UNIVERSIDADE ESTADUAL DE LONDRINA

HUMBERTO FERREIRA DA LUZ JUNIOR

GAIADOC:
UMA FERRAMENTA DE DOCUMENTAÇÃO DE CÓDIGO FONTE DIRECIONADA AOS REQUISITOS DE NEGÓCIO

LONDRINA–PR

2013

L979g Luz Junior, Humberto Ferreira da

GAIADOC: uma Ferramenta de Documentação de Código Fonte Direcionada aos Requisitos de Negócio/ Humberto Ferreira da Luz Junior. – Londrina, 2013.

66f. : il.

Orientador: Prof. Dr. Rodolfo Miranda de Barros.

Dissertação (Mestrado em Ciência da Computação) – Universidade Estadual de Londrina, Centro de Ciências Exatas, Programa de Pós-Graduação em Ciência da Computação, 2013.

Inclui bibliografia.

1. GaiaDoc – Teses. 2. Engenharia de requisitos – Teses. 3. Engenharia de software – Teses. 4. Software – Desenvolvimento – Teses. I. Barros, Rodolfo Miranda de. II. Universidade Estadual de Londrina. Centro de Ciências Exatas. Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDU 519.68.02

HUMBERTO FERREIRA DA LUZ JUNIOR

GAIADOC:

UMA FERRAMENTA DE DOCUMENTAÇÃO DE CÓDIGO FONTE DIRECIONADA AOS REQUISITOS DE NEGÓCIO

Dissertação apresentada ao programa de Pós-Graduação em Ciência da Computação, Departamento de Computação da Universidade Estadual de Londrina, para a obtenção do título de Mestre em Ciência da Computação.

Orientador:
Prof. Dr. Rodolfo Miranda de Barros

LONDRINA – PR
2013

HUMBERTO FERREIRA DA LUZ JUNIOR

GAIADOC:
UMA FERRAMENTA DE DOCUMENTAÇÃO DE CÓDIGO FONTE DIRECIONADA AOS REQUISITOS DE NEGÓCIO

Dissertação apresentada ao programa de Pós-Graduação em Ciência da Computação, Departamento de Computação da Universidade Estadual de Londrina, para a obtenção do título de Mestre em Ciência da Computação.

BANCA EXAMINADORA

Prof. Dr. Rodolfo Miranda de Barros
Universidade Estadual de Londrina
Orientador

Prof. Dr. Alan Salvany Felinto
Universidade Estadual de Londrina
Membro titular

Prof. Dr. Bruno Bogaz Zarpelão
Universidade Estadual de Londrina
Membro titular

Prof. Dr. Lourival Aparecido de Góis
Universidade Tecnológica Federal do Paraná - Campus Ponta Grossa
Membro titular

Londrina, 01 de Julho de 2013

AGRADECIMENTOS

Agradeço aos meus pais, Humberto Ferreira da Luz e Claudia Cristina Baptista da Luz, que me deram suporte ao longo de todos estes anos desde minha mudança para Londrina em razão da graduação até os dias atuais.

À minha noiva Karen Lumi Nakano, maior amiga e companheira, por sua paciência e compreensão, em especial nas vésperas de submissão de artigos e em outros momentos críticos durante o período de mestrado.

Agradeço a Rodolfo Miranda de Barros, meu orientador desde a graduação até o mestrado, que tornou possível a aprovação desse trabalho por meio de seus conselhos e supervisão.

E à Universidade Estadual de Londrina, em especial à Luciana Tomasi Carli Soares e Simone Yuriko Kobayashi, que tornaram possível a obtenção da licença parcial de minhas atividades como Analista de Informática na Assessoria de Tecnologia de Informação (ATI).

LUZ JUNIOR, Humberto Ferreira da. **GAIADOC:** uma Ferramenta de Documentação de Código Fonte Direcionada aos Requisitos de Negócio. 2013. Dissertação (Mestrado em Ciência da Computação) – Universidade Estadual de Londrina, Londrina, 2013.

RESUMO

Com o objetivo de preencher uma lacuna nos métodos tradicionais de documentação de código fonte, que se concentram principalmente na documentação de API para outros programadores, esta pesquisa apresenta uma nova abordagem para a documentação dos requisitos de negócio, mapeando-os através de um conjunto de anotações. Essas anotações, por sua vez, são interpretadas pela ferramenta GaiaDoc, que é especificada neste trabalho e é capaz de gerar a documentação em forma de especificações de caso de uso em uma linguagem e formato de fácil compreensão pelos participantes do projeto. Junto com a proposta da ferramenta GaiaDoc, um fluxo de requisitos baseado no RUP é desenvolvido para se encaixar com as necessidades da nova abordagem de documentação de código fonte e é validado por meio das áreas de processo de requisitos do CMMI. Dois estudos de caso da aplicação da metodologia proposta são apresentados antes das considerações finais.

Palavras-Chaves: Documentação de código fonte; engenharia de requisitos; processo unificado rational; javadoc; gerador de documentação.

LUZ JUNIOR, Humberto Ferreira da. **GAIADOC:** a Source Code Documentation Tool Focused on Business Requirements. 2013. Dissertação (Mestrado em Ciência da Computação) – Universidade Estadual de Londrina, Londrina, 2013.

ABSTRACT

Aiming to fill a gap in traditional methods of source code documentation, which focus mainly on the API documentation for other programmers, this research presents a new approach for business requirements documentation, mapping them through a set of annotations. These annotations, in turn, are interpreted by the GaiaDoc tool, which is specified in this paper and is able to generate documentation in form of use case specifications in a language and format easily understandable by the project stakeholders. Along with the GaiaDoc tool proposal, a RUP based requirements flow is developed to meet the needs of the new approach for source code documentation and it is validated by means of the CMMI requirements process areas. Two case studies of the proposed methodology's application are presented before the final considerations.

Keywords: Source code documentation; requirements engineering; rational unified process; javadoc; documentation generator.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API *Application Programming Interface*

ATI Assessoria de Tecnologia de Informação

CMMI *Capability Maturity Model Integration*

DCF Documentação de Código Fonte

DDS Diretoria de Desenvolvimento de Sistemas

DSAC Divisão de Sistemas Acadêmicos

DSAD Divisão de Sistemas Administrativos

DSF Divisão de Sistemas Financeiros

DSRH Divisão de Sistemas de Recursos Humanos

DSRS Diretoria de Suporte a Redes e Sistemas

DSU Diretoria de Suporte ao Usuário

DSW Divisão de Sistemas Web

ECU Especificação de Caso de Uso

EJB *Enterprise JavaBeans*

HTML *HyperText Markup Language*

IDE *Integrated Development Environment*

JEE *Java Enterprise Edition*

JPA *Java Persistence API*

JSF *JavaServer Faces*

MVC *Model View Controller*

PDF *Portable Document Format*

PDS Processo de Desenvolvimento de Software

POJO *Plain Old Java Object*

RUP *Rational Unified Process*

SVN *Subversion*

UBS Unidade Básica de Saúde

UML *Unified Modeling Language*

XML *Extensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

A Documentação de Código Fonte (DCF) é considerada uma das principais boas práticas para qualquer linguagem de programação. Permite que outros desenvolvedores aprendam a utilizar o código sem a necessidade de analisar centenas, ou, até mesmo, milhares de linhas de algoritmos complexos e propicia que o conhecimento sobre o seu funcionamento não fique armazenado apenas nas pessoas.

Quase todas as práticas recomendadas para esse tipo de documentação focam em usuários de API (*Application Programming Interface*), ou seja, em outros programadores, que geralmente adotam uma linguagem excessivamente técnica e, muitas vezes, somente descrevem um algoritmo que deveria ser legível apenas por seu próprio código com classes, variáveis e métodos bem nomeados.

No RUP (*Rational Unified Process*), as regras de negócio de projetos de software são especificadas principalmente como requisitos funcionais no modelo de caso de uso, que representa interações entre os atores e o sistema, e nas descrições detalhadas de cada um dos casos de uso, que são definidos como Especificações de Caso de Uso (ECU) [1].

Em muitos casos, a DCF está presente em classes que representam regras de negócio do projeto. Um claro exemplo das classes que implementam essas regras reside na camada *controller* da arquitetura *Model View Controller* (MVC), que não costuma ser utilizada para fornecer serviços, como no caso de uma API, sendo perfeitamente possível associá-la a casos de uso da UML (*Unified Modeling Language*).

A problemática encontra-se no fato de que as especificações de documentação de código, assim como suas ferramentas, foram criadas especialmente para documentação de APIs, induzindo os programadores a documentar classes que implementam as regras de negócio de forma incorreta.

Nas classes que representam regras de negócio uma documentação essencialmente técnica pode torná-la insuficiente para compreender por qual razão determinada classe existe, o que ela faz, quais suas restrições, entre outras informações cabíveis a este contexto.

Por outro lado, se a classe for documentada em excesso com os requisitos de regra de negócio, se torna comum encontrar casos em que há pouca integridade e rastreabilidade entre a ECU (ou documentação equivalente dos requisitos), a DCF e a própria implementação. Isto ocorre porque os requisitos são modificados com muita frequência, fazendo com que o código fonte seja atualizado e, por falta de tempo, sua documentação e requisitos documentados na ECU passem a ser desprezados [2].

Em consequência, a versão corrente da implementação raramente é equivalente à da documentação, pois se demanda demasiado esforço para mantê-la atualizada. Para tratar essa questão, diversas ferramentas, tais como a Javadoc, surgiram para concentrar a DCF e exportá-la para outros formatos mais

acessíveis para seus leitores.

Métodos atuais de DCF como Javadoc [3], Sandcastle [4] e Doxygen[5] atendem bem os projetos de software que fornecem serviços ou bibliotecas para outros programadores. Exemplos incluem *web services*, como o de empresas responsáveis pelo envio e entrega de correspondências em uma região, que pode ser utilizado para o cálculo de preços e prazos de entregas para determinado endereço, e bibliotecas, como aquelas da própria linguagem Java.

Tendo em vista o cenário descrito, a motivação para a concepção deste trabalho surgiu desses casos em que boa parte do código está diretamente relacionado às regras de negócio inerentes ao próprio projeto, como no caso de projetos corporativos, que podem, inclusive, utilizar os serviços e bibliotecas já mencionados.

Visando tratar os problemas citados, é proposta, neste trabalho, uma nova abordagem para a documentação de código focando apenas na regra de negócio. Este enfoque é dado através de um conjunto próprio de anotações mapeadas em ECUs e disponibilizadas por meio de uma linguagem e um formato facilmente compreensíveis para os *stakeholders* do projeto.

Este trabalho apresenta, ainda, a ferramenta GaiaDoc, responsável pelo mapeamento das anotações em ECU, e algumas vantagens em armazenar a DCF. Dentre as vantagens, há o versionamento, que pode ser realizado por intermédio de um sistema de gerenciamento de configuração, tal como o Subversion [6] ou o GIT [7], e o rastreamento de erros nos requisitos, que pode ser obtido por meio de sistemas de *bug tracking*, como o Bugzilla ou o JIRA.

Além da geração de ECU, a GaiaDoc também faz a leitura de uma anotação que registra as lições aprendidas associadas ao escopo do caso de uso, persistindo a informação em um banco de dados. Então as lições podem ser vistas e gerenciadas por meio do portal corporativo Gaia [8].

A adoção do conjunto de anotações e da ferramenta GaiaDoc resultou em modificações no fluxo tradicional de requisitos do RUP, inserindo em seu escopo artefatos adicionais e alterando alguns procedimentos no desenvolvimento e na gestão dos requisitos. As alterações efetuadas no fluxo de requisitos foram validadas para determinar se a conformidade com as áreas de processo de requisitos do CMMI (*Capability Maturity Model Integration*) existia e/ou foi mantida em relação ao RUP.

Essa abordagem permite que se torne mais fácil manter a integridade dos requisitos em relação à sua implementação e que seja mantida a rastreabilidade dos requisitos, o que garante que todos os requisitos compreendidos pelo analista de requisitos e homologados pelo cliente estejam presentes na implementação e no produto final, além de evitar que funcionalidades desnecessárias sejam construídas [9].

A abordagem deste trabalho é validada de forma técnica, por meio de um estudo de caso aplicado ao sistema de gestão de clínica odontológica, e de forma organizacional, por intermédio de um estudo caso da aplicação do GaiaDoc na Diretoria de Desenvolvimento de Sistemas (DDS) na Assessoria de Tecnologia de Informação (ATI) da Universidade Estadual de Londrina (UEL).

O capítulo 2 apresenta o contexto que motivou o desenvolvimento deste trabalho, incluindo os conceitos teóricos introdutórios sobre DCF, RUP, MVC e CMMI, respectivamente. O capítulo 3 descreve

a metodologia utilizada para mapear a DCF para a ECU e trabalhos relacionados a documentação dos requisitos no código fonte, assim como funcionamento da ferramenta responsável por analisar o código e gerar o documento de saída. O capítulo 3 também define o fluxo de requisitos para um projeto que siga a abordagem proposta e o valida segundo as áreas de processo de requisitos do CMMI. O capítulo 4 descreve um estudo de caso de aplicação da GaiaDoc em um sistema de gestão de clínica odontológica com o intuito de tornar mais compreensível o funcionamento da ferramenta e na DDS da UEL para verificar a aplicabilidade da proposta em um ambiente já consolidado de desenvolvimento de software. O capítulo 5 estabelece as considerações finais e considera trabalhos futuros que podem ser realizados a partir deste estudo. O apêndice A exibe mais 2 exemplos de documentação de código fonte no padrão GaiaDoc e a ECU resultante. O apêndice B contém os resultados completos da pesquisa realizada no estudo de caso aplicado à ATI.

2 CONTEXTUALIZAÇÃO

Esse capítulo apresenta os conceitos sobre a DCF e o RUP, que representam a base desse estudo, o MVC, que introduz um padrão de arquitetura que se encaixa bem à metodologia proposta, e o CMMI, que foi utilizado na validação do fluxo de requisitos modificado para atender às mudanças causadas pelo uso da ferramenta GaiaDoc.

2.1 Documentação de Código Fonte

A documentação de código fonte está entre as mais básicas boas práticas ensinadas nos cursos de programação, consistindo em uma descrição em linguagem natural de funções, métodos, classes, variáveis e constantes.

Esse tipo de documentação reside no código fonte dentro de comentários, que consistem em frações de texto identificado pelo compilador que são completamente ignorados por não representar ações válidas dentro do código [10].

Novos programadores ou usuários de APIs aproveitam essa documentação para entender o funcionamento da aplicação sem criar a necessidade de análise minuciosa de centenas ou milhares de linhas de código fonte para compreender a sua lógica.

O próprio autor do código fonte, após um longo período de tempo, pode utilizar a documentação para se recordar de suas decisões sobre a forma em que o código foi implementado e evitar que essas informações sejam perdidas, mesmo após sua saída da organização.

A legibilidade da documentação pode ser prejudicada por estar misturada com o código fonte. Por esta razão diversas linguagens de programação dispõem de ferramentas[1] que permitem exportá-la para outros formatos como HTML, DOC, PDF, entre outros. Já o problema da legibilidade do código em meio ao excesso de documentação pode ser contornado pelo uso de IDEs (Integrated Development Environment) que fornecem a opção de ocultar comentários.

Os programadores estão sempre próximos do desastre quando estão comentando o código [11]. Isso ocorre em razão do hábito deles descreverem toda a funcionalidade do código com mais detalhes que o necessário, o que leva a uma documentação, quase em sua totalidade, inútil e difícil de se manter, ou irem pelo caminho contrário, documentando menos que o necessário para a compreensão de uma classe ou

[1] Como exemplos de ferramentas de DCF podemos citar a Doxygen, a phpdoc, a Sandcastle e a Javadoc, sendo esta última abordada na seção 8 deste trabalho.

método.

De acordo com Spinellis [11], após a análise do código fonte dos projetos da plataforma FreeBSD, foi possível constatar que havia uma variação muito grande na quantidade média de documentação a cada 100 linhas de código. A pesquisa demonstrou que cada programador tem sua própria visão de como seu código deve ser documentado, o que leva a uma grande inconsistência na DCF de projetos de software.

O código deve ser auto-descritivo, por meio de variáveis e funções bem nomeadas e de simples compreensão. Os blocos de comentário devem ser aproveitados para descrever as razões que levaram o programador a resolver o problema de um modo específico ou descrever as regras que se aplicam a aquela classe ou função, circunstâncias em que o uso da linguagem natural é imprescindível.

Segundo os estudos de Schreck [12] a má documentação não é um resultado dos processos de documentação, mas de desenvolvedores que não aderem aos processos especificados. Uma supervisão rigorosa por parte do gerente de projetos poderia resolver este problema, entretanto em grandes projetos essa tarefa se torna tediosa e custosa.

O fato de ter de parar de escrever código, que é algo que programadores geralmente gostam de fazer, para ter que se preocupar com a documentação, que em alguns casos os programadores não sabem como escrever, é uma das razões prováveis para a falta de vontade dos desenvolvedores para documentar código fonte. Este desconforto aumenta ainda mais quando a documentação está fora do código [10].

A maior atratividade exercida sobre os programadores pelo código pode ser justificada pelas regras bem definidas, rigor matemático, linguagem semântica não ambígua, sintaxe altamente estruturada e pensamento criativo e disciplinado exigido pelas linguagens de programação e projeto de algoritmos [13].

Por outro lado, a documentação em geral oferece uma grande liberdade de expressão e níveis de abstração, a qual os programadores tem grande dificuldade de tirar total proveito. A documentação pode indicar, por exemplo, o que o código deve fazer, qual o histórico do método, quais as possíveis exceções do código, quais as implicações de uma determinada solução, qualquer interação externa e premissa que o programador poderia ter [13].

Com tanta liberdade se torna difícil escolher como deveria ser o comentário apropriado em cada situação e contexto. Os programadores preferem se dedicar às atividades em que eles possuem maior domínio causando como consequência documentação incompreensível, mal escrita, desatualizada ou incompleta.

Todos estes fatos dão origem a projetos com documentação de má qualidade, difíceis de ser mantidos e estendidos visto que não é possível garantir que seus desenvolvedores continuem disponíveis para a organização que os gerencia.

2.2 Rational Unified Process

O Rational Unified Process é um Processo de Desenvolvimento de Software (PDS) iterativo, incremental, orientado a planejamento e a riscos, além de focado na arquitetura do sistema a ser desenvolvido

[14]. Seu objetivo é guiar e orientar organizações de desenvolvimento de software em seus empreendimentos [1].

Além de um PDS, o RUP também é um *framework* de processo que pode ser utilizado exatamente da forma que é ou então ser adaptado e estendido para se moldar aos requisitos da organização, acomodando necessidades especiais, características, limitações, cultura e domínio. Isto é possível, pois se seguido fielmente, o processo pode gerar trabalho inútil e produzir artefatos que agregam pouco valor à organização [1].

O processo do RUP possui duas dimensões:

- A horizontal que representa o tempo e aspecto dinâmico do RUP através de seu ciclo de vida que consiste em fases (as mesmas do modelo cascata), iterações e pontos de controle;
- A vertical representa as nove disciplinas principais do RUP (estas disciplinas também são conhecidas como fluxos de trabalho), que agrupa suas atividades de engenharia de software de acordo com sua natureza.

As quatro fases do RUP que representam sua dimensão horizontal são:

- Concepção;
- Elaboração;
- Construção;
- Transição.

As nove disciplinas de engenharia de software e suporte que o compõem estão abaixo:

- Modelagem de negócios;
- Requisitos;
- Análise e projeto;
- Implementação;
- Teste;
- Distribuição;
- Ambiente;
- Configuração e gerência de mudança;
- e gerência de projeto.

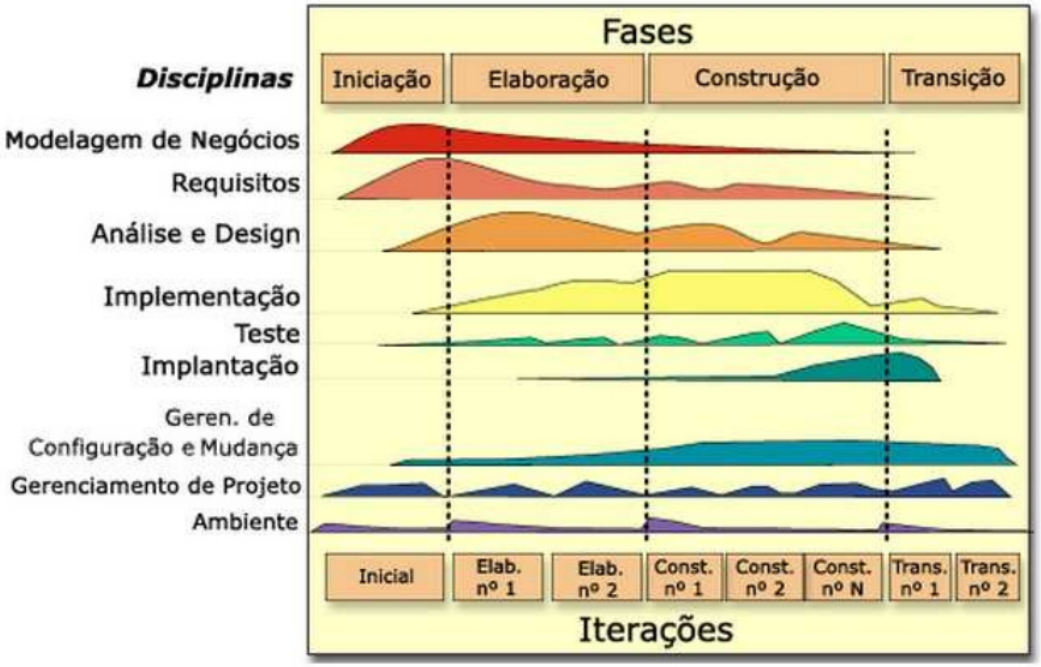


Figura 2.1 – Ciclo de vida do RUP que representa a intensidade dos fluxos de trabalho durante cada uma de suas fases.

Na figura 2.1 é possível visualizar o ciclo de vida de cada um destes fluxos, incluindo o momento em que são iniciados, suas durações e seus términos [1].

As seções seguintes se preocupam em descrever melhor o ciclo de vida e as disciplinas de engenharia de software e de suporte relevantes ao escopo deste trabalho.

2.2.1 Ciclo de Vida

O ciclo de vida do RUP, como citado anteriormente, possui quatro fases: concepção, elaboração, construção e transição. Elas acontecem iterativamente e concorrentemente, pois é possível que a fase seguinte inicie antes de sua predecessora se encerrar.

Durante a fase de concepção são identificados os requisitos de negócio para o software através de casos de uso, é desenhado um rascunho preliminar da arquitetura do sistema e é criado o plano do projeto que será seguido. Em geral, esta fase abrange atividades de planejamento e comunicação com o cliente, já que eles colaboram com o desenvolvimento do plano do projeto e com as outras atividades desta fase [15].

A fase de elaboração é a fase que prossegue com a comunicação com o cliente de forma que os casos de uso preliminares e a representação arquitetural sejam refinados e expandidos. Esta fase também inclui modificações e uma revisão cuidadosa no plano do projeto para assegurar que o escopo, os riscos e as datas de entrega sejam razoáveis antes de iniciar a fase de construção [15].

A fase de construção é aquela que implementa em código fonte os casos de uso utilizando o modelo arquitetural como entrada e adicionando a implementação de forma incremental no sistema a ser desenvolvido. Além da construção dos casos de uso, durante esta fase também são projetados e realizados

testes unitários, de integração e de aceitação nas entregas finalizadas da iteração [15].

Durante a fase de transição, são realizados testes beta da entrega da iteração junto aos usuários finais e é criado como resultado um relatório contendo defeitos encontrados e mudanças necessárias para corrigi-los. Materiais de apoio tais como manuais de usuário, guias para solução de problemas e procedimentos de instalação que possivelmente precisem ser entregues dependendo do que foi definido no escopo do projeto. Na conclusão desta fase, o incremento produzido na fase de construção é integrado ao sistema e torna-se utilizável [15].

2.2.2 Disciplinas de Engenharia de Software e de Suporte

Disciplina de Modelagem de negócios: A modelagem de negócios é importante para garantir que se saiba o domínio do negócio ao realizar o projeto de engenharia de software e que as aplicações construídas agreguem valor ao serviço do cliente. Segundo Krutchten [1], ela tem como finalidade entender os problemas atuais, a estrutura e a dinâmica da organização na qual um sistema será distribuído, assegurar que os clientes, usuários finais e desenvolvedores tenham um entendimento comum da organização alvo e derivar seus requisitos de sistema necessários para seu suporte.

As atividades de modelagem de negócios não são recomendadas para qualquer trabalho de engenharia de software. Elas são indicadas principalmente quando há mais pessoas diretamente envolvidas no uso do sistema e mais informação precisa ser manipulada pelo software, já que é nessa situação que ela mais agrega valor [1].

Na modelagem de negócios, atores representam os usuários de negócio, casos de uso representam os processos de negócio, papéis exercidos por pessoas na organização são representados por trabalhadores de negócio e entidades de negócio representam tudo o que for administrado ou produzido pela organização.

Este fluxo gera como artefatos:

- O documento de visão de negócio, que define os objetivos e metas do trabalho de modelagem de negócio;
- Um modelo de caso de uso de negócio, que modela as funções planejadas de negócio, com o objetivo principal de identificar entregas e papéis na organização;
- Um modelo de objeto de negócio, que descreve a realização de casos de uso de negócio;
- A avaliação da organização alvo;
- As regras de negócio;
- As especificações de negócio suplementares;
- E um glossário de negócio.

Disciplina de Requisitos: O fluxo de requisitos tem por objetivo captar e gerenciar efetivamente os requisitos funcionais e não-funcionais do sistema e com base nas necessidades do usuário projetar

interfaces que por ele serão utilizadas. Além disso este fluxo se concentra em fornecer um melhor entendimento dos requisitos do sistema após ser acordado entre os interessados sobre o que o sistema deveria fazer e também define o escopo do sistema [1].

As ferramentas para atingir estas metas se baseiam em especificações adicionais em modelos de caso de uso, definindo os requisitos do sistema de forma detalhada. Isto auxilia na administração da extensão e das mudanças dos requisitos por meio de descrições sobre o modo de usar os atributos dos requisitos do sistema [1].

Dentro do fluxo de requisitos do RUP, a modelagem de caso de uso vem após a definição do documento de visão, que contém o escopo geral, as necessidades fundamentais e as características do projeto, além da análise de expectativas das partes interessadas. Ela ocorre simultaneamente ao projeto de interface de usuário e a administração dos requisitos documentados ocorre ao longo de todo o projeto em virtude de mudanças e refinamentos.

A tradução das necessidades e dos interesses dos *stakeholders* do projeto contidos no documento de visão origina o detalhamento dos requisitos na forma de casos de uso. Esses casos de uso servem de base para projetar e construir o sistema, identificar os casos de teste e validar o resultado final do projeto, garantindo que todos os requisitos acordados sejam cumpridos adequadamente [1]. Na primeira iteração do PDS, a ECU deve ocorrer antes que seja iniciada a fase de construção do projeto e em conjunto com outras especificações adicionais, como os outros diagramas da UML.

O modelo de caso de uso descreve o comportamento e o relacionamento entre o sistema e os atores, que representam papéis desempenhados pelos usuários, através de interações, sem entrar em detalhes sobre como o sistema será implementado [16].

Para cada caso de uso identificado, é criada sua respectiva especificação de caso de uso em linguagem natural, que contém as pré-condições, as pós-condições, as extensões, os requisitos especiais, a descrição básica e o fluxo passo-a-passo de eventos executados pelo ator naquele caso de uso. Esse fluxo de eventos se divide entre o básico, que é o que deve ocorrer normalmente, e o alternativo, que se inicia quando eventos inesperados, como, por exemplo, falhas no sistema acontecem.

Requisitos especiais abrangem requisitos não-funcionais que tem influência no escopo da ECU como usabilidade, confiabilidade, desempenho. Um exemplo é a especificação do limite de tempo de resposta de um determinado sistema.

É importante destacar que a modelagem de caso de uso é um método bastante popular para documentação de requisitos funcionais de projetos orientados a objetos nas organizações. A ECU é usualmente criada e mantida como um documento de texto, o que pode causar problemas de integridade entre o código fonte, a especificação, os outros artefatos do projeto e o próprio produto final.

Outros artefatos que também podem ser desenvolvidos são o glossário, que define uma terminologia comum no projeto, a *storyboard* de caso de uso e o protótipo de interface do usuário que auxiliam a envolver os usuários do sistema na fase de requisitos.

Disciplina de Teste: O fluxo de testes se inicia no começo do projeto com a avaliação da

qualidade da arquitetura até do produto ou resultado final do projeto, através da verificação das interações e integração de componentes, da implementação correta de todos os requisitos, além da identificação e garantia que os relatos de falhas sejam encaminhados para as pessoas corretas e que isto ocorra previamente à distribuição do produto [1].

Diversos tipos de teste podem ser realizados [1], dentre eles estão: (i) instalação; (ii) integridade; (iii) carga, que verifica o comportamento do sistema sob grandes quantidades de usuários e transações, sempre com a mesma configuração; (iv) desempenho, que realiza os testes sob configurações diferentes; e (v) tensão, que realiza os testes em condições anormais ou extremas.

Como artefatos desse fluxo há:

- O plano de teste, que contém informações acerca de como os testes serão realizados, seus objetivos e os recursos necessários para executá-los;
- O modelo de teste, que descreve os testes e representa o que será testado e de que forma;
- Os resultados de teste, que são os dados obtidos através de sua realização;
- O modelo de carga de trabalho, que possui os valores das simulações realizadas, bem como também seu carregamento e volume;
- Os defeitos, que são testes que falham e assim geram solicitações de mudança;
- Os pacotes e classes de teste;
- E os subsistemas e componentes de teste.

Disciplina de Configuração e Gerência de Mudança

O fluxo de configuração se preocupa com a estrutura do produto e com os artefatos importantes e suas inter-dependências, controlando o versionamento e o histórico das mudanças de forma que se obtenha rastreabilidade nas mudanças realizadas no decorrer do projeto [1].

Já o fluxo de mudanças lida com propostas documentadas de mudanças que interferem em um ou mais artefatos do projeto com o objetivo de corrigir falhas encontradas, melhorar a qualidade, acrescentar algum novo requisito solicitado, entre outros motivos. Após a análise de impacto de mudança, uma decisão será tomada indicando se a mudança será implementada, e em caso afirmativo também definirá quando e em que versão essa modificação ocorrerá [1].

Durante estes fluxos, informações podem ser obtidas acerca da quantidade de mudanças aprovadas ainda não implementadas, sobre a severidade das falhas localizadas, e do progresso do projeto quanto a esse trabalho que deverá ser realizado em relação ao que já foi implementado. Estas informações compõem relatórios de medida e estado para o projeto [1].

Os principais artefatos envolvidos no fluxo de gerenciamento de configuração e mudanças são:

- O plano de gerenciamento de aplicação, que descreve as práticas que devem ser adotadas durante este fluxo;
- As solicitações de mudança;
- E os relatórios de medida e estado.

2.3 Model View Controller

Model View Controller é um padrão de projeto arquitetural desenvolvido para a linguagem Smalltalk que propõe que as regras de negócio devem ser desacopladas da visão e do domínio da aplicação, sendo portanto uma arquitetura de três camadas. Seu objetivo é o de simplificar a implementação e facilitar o reuso de código.

O modelo (*model*) representa os dados e o comportamento que serão exibidos ao usuário. Uma visão (*view*) é criada para cada interface gráfica que poderá ser apresentada ao usuário e seu conteúdo depende do domínio. O controlador (*controller*) recebe e processa uma determinada entrada do usuário para uma função específica e se responsabiliza por manter o domínio sempre sincronizado com a *view*, coordenando o fluxo de dados entre eles [17].

Em aplicações Web, as quais habitualmente adotam a arquitetura MVC, o usuário envia uma requisição pelo navegador Web. No lado do servidor a requisição é tratada pelo controlador, que atualiza o modelo e, em seguida, determina qual visão deverá ser apresentada ao usuário. Para finalizar, o servidor envia uma resposta ao cliente e o resultado é exibido ao usuário. A figura 2.2 demonstra essa sequência, no formato de um diagrama de sequência da UML [15].

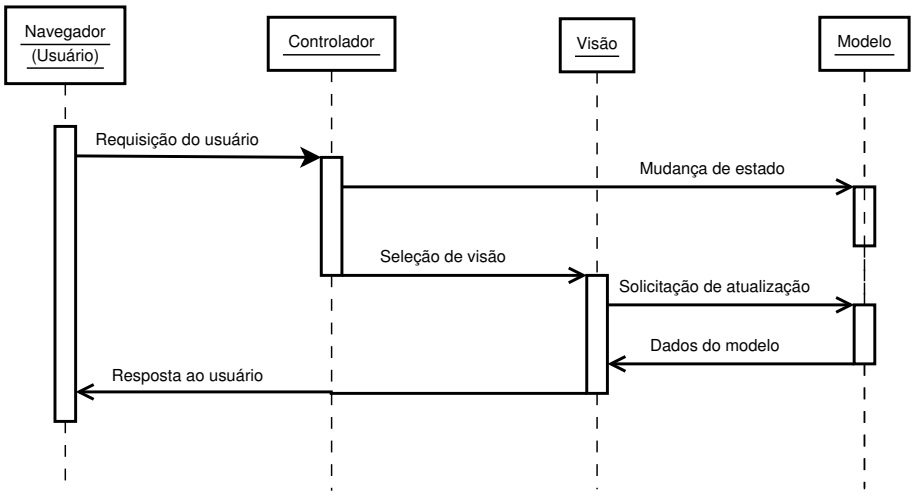


Figura 2.2 – Diagrama de sequência que demonstra o funcionamento da arquitetura MVC.

Esta arquitetura foi escolhida devido ao desacoplamento da regra de negócio, o que torna a arquitetura MVC um exemplo ideal de como o GaiaDoc pode ser utilizado de forma eficiente. Apesar do domínio de regra de negócio pertencer ao modelo, os casos de uso podem ser melhor representados na camada de controle, pois é a camada que gerencia os fluxos de eventos do sistema. Sua documentação é representada, posteriormente, no formato de ECU.

2.3.1 Arquitetura Java Enterprise Edition

Originalmente, a arquitetura MVC foi projetada para ser utilizada em aplicações Desktop e por esta razão não atende algumas necessidades de aplicações modernas que são implantadas em servidores e executadas no cliente.

A figura 2.3 exibe a arquitetura da plataforma Java Enterprise Edition (JEE) 6 . Ela consiste em três camadas [18]:

- Cliente: é executada por navegadores Web na máquina do cliente, sendo representada principalmente por páginas Web dinâmicas em linguagem de marcação como HTML ou XML;
- Servidor JEE: é subdividida na camada Web, que contém controladores de fluxo e determina o que é exibido no cliente, e na camada de negócios, que implementa o domínio de negócios em *beans* de entidade e de sessão do *framework* Enterprise Javabeans (EJB);
- Servidor de Banco de Dados: possui a definição e o conteúdo das tabelas de banco de dados utilizadas para mapear as entidades da camada de negócios.

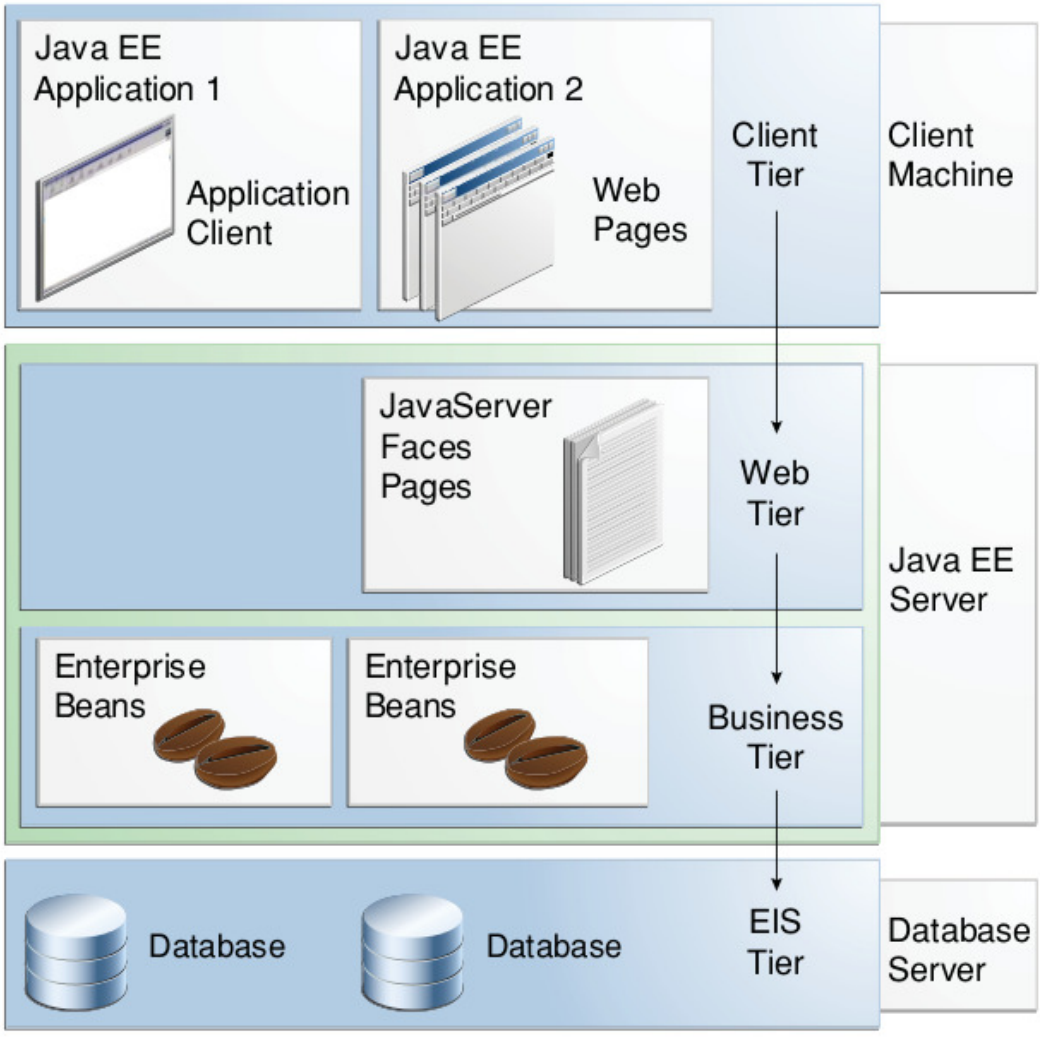


Figura 2.3 – Arquitetura da plataforma JEE 6.

De modo superficial, pode-se dizer que a camada Cliente do Java EE é equivalente à Visão do MVC, a camada Web é análoga ao Controle e as camadas de Negócios e de Banco de dados correspondem ao Modelo.

2.4 CMMI

O CMMI é um *framework* que consiste nas melhores práticas que tratam das atividades de desenvolvimento aplicadas a produtos e serviços em 22 áreas de processo. Cada área possui seus objetivos específicos com suas respectivas práticas que auxiliam a alcançá-los.

As práticas do CMMI abrangem desde a adoção de processos em uma organização até sua melhoria contínua e evolutiva. Partindo de processos *ad hoc* e imaturos para processos maturos, disciplinados, eficazes e com qualidade aprimorada.

É importante destacar, entretanto, que o CMMI fornece diretrizes sobre o que deve ser feito, porém não descreve como os objetivos devem ser alcançados. Por esta razão, PDSs podem ser utilizados para complementar e auxiliar a cumprir os objetivos do CMMI em determinadas áreas de processo.

O escopo das áreas de processo abrange desde práticas de planejamento, engenharia e gerenciamento de projeto de software até engenharia de hardware, cobrindo o ciclo de vida do produto de sua concepção até sua entrega e manutenção.

O resultado esperado da correta adoção das práticas do CMMI inclui aumento de produtividade e qualidade, além de cronogramas mais precisos e execução do projeto dentro do orçamento estimado.

Esse trabalho abordará apenas duas áreas de processo: Desenvolvimento de Requisitos e Gerenciamento de Requisitos, pertencentes aos níveis de maturidade 3 e 2 do CMMI, respectivamente.

Todas as informações relacionadas aos processos do CMMI descritas neste capítulo foram baseadas no CMMI-Dev, publicado em [19].

2.4.1 Desenvolvimento de Requisitos

A área de processo Desenvolvimento de Requisitos pertence ao nível 3 de maturidade do CMMI e abrange os requisitos de cliente e os requisitos de produto e de seus componentes. Agregados, eles tratam as necessidades relevantes dos *stakeholders*, incluindo restrições causadas pela seleção do padrão de arquitetura de projeto e requisitos não-funcionais (segurança, tempo de resposta, manutenibilidade, entre outros).

Esta área possui três objetivos específicos:

- Desenvolver Requisitos de Cliente;
- Desenvolver Requisitos de Produto;
- Analisar e Validar Requisitos.

As duas primeiras estão diretamente associadas aos requisitos a serem desenvolvidos pelos processos, enquanto a terceira serve como suporte na análise e validação destes requisitos.

Cada um dos objetivos possui passos a serem seguidos e estão descritos nas subseções seguintes.

Desenvolver Requisitos de Cliente: Os requisitos de cliente são representados pelas necessidades dos *stakeholders* e seu desenvolvimento é importante por poder levantar requisitos que possivelmente não seriam identificados diretamente pelo desenvolvimento de requisitos de produto.

As necessidades, expectativas, restrições, interfaces, conceitos operacionais e conceitos de produto são iterativamente analisadas, harmonizadas, refinadas e elaboradas para serem traduzidas como um conjunto de requisitos de cliente. Estes requisitos são, então, derivados e refinados em requisitos de produto e seus componentes.

Um substituto para os usuários finais ou clientes pode ser definido para representar suas necessidades com o objetivo de facilitar a interação com os *stakeholders* e evitar conflitos. Um exemplo é o *product owner* no processo de desenvolvimento ágil Scrum.

Este objetivo possui duas práticas específicas:

- Elicitar necessidades: abrange a coleta de requisitos, expectativas e restrições relacionados ao produto e às atividades relacionadas ao seu ciclo de vida por meio de técnicas como entrevistas, protótipos, *brainstorming*, casos de uso, estória e engenharia reversa;
- Transformar necessidades dos *stakeholders* em requisitos do cliente: transforma e refina as informações obtidas pelo passo anterior em requisitos de cliente funcionais e não funcionais classificados por prioridade. Também inclui nos procedimentos de validação e verificação as restrições identificadas.

Desenvolver Requisitos de Produto: A análise dos requisitos de cliente e dos conceitos operacionais são derivados e refinados, dando origem a requisitos mais técnicos, denominados requisitos de produto e componentes de produto.

Os requisitos de produto e de seus componentes tratam das necessidades relacionadas ao ciclo de vida do produto, que pode consistir-se, por exemplo, nas seguintes etapas:

- Conceito e visão;
- Análise de viabilidade;
- Projeto e desenvolvimento;
- Produção;
- Término gradual.

No caso de projetos iterativos ou incrementais, os requisitos são atribuídos a cada iteração/incremento de acordo com sua prioridade.

Abaixo, estão detalhados as três práticas específicas pertencentes ao desenvolvimento de requisitos de produto:

- Estabelecer requisitos do produto e de seus componentes: os requisitos de cliente, geralmente expressos em linguagem não técnica, são traduzidos em requisitos de produto, que representam os requisitos de cliente em linguagem técnica para as decisões de projeto. Os requisitos são derivados para seleção de arquitetura e tecnologias a ser aplicadas no produto e o relacionamento entre os requisitos é estabelecido e consultado durante o gerenciamento de mudanças;
- Alocar requisitos de componentes do produto: os requisitos de produto são alocados em componentes do produto. O significado de componente de produto pode variar de acordo com o contexto e a arquitetura de produto, porém basicamente representa uma parte do produto, como por exemplo aquela que é entregue em uma determinada iteração, caso de uso ou estória;
- Identificar requisitos de interface: são identificadas e definidas as interfaces entre funções (ou outras entidades lógicas de acordo com a arquitetura adotada). As interfaces são utilizadas na integração entre os componentes do produto.

Analisar e Validar Requisitos: Analisar e validar requisitos é utilizada como suporte para os dois outros objetivos específicos citados anteriormente.

A análise se concentra em verificar se os requisitos coletados e decisões tomadas terão o impacto desejado para atender as necessidades dos *stakeholders*. Já a validação compara o ambiente desejado para o usuário final e as necessidades dos *stakeholders* (*baseline* do produto) com o que já foi realizado.

Portanto, a análise procura assegurar que o produto a ser desenvolvido ou em desenvolvimento estará em conformidade com o que é esperado dele, enquanto a validação faz a verificação do produto já finalizado (ou de um de seus componentes) para determinar se os objetivos foram ou estão sendo atingidos.

Os cinco processos que fazem parte da análise e validação de requisitos são:

- Estabelecer conceitos e cenários operacionais: definição de sequências e ambiente de eventos que podem ocorrer no desenvolvimento, instalação, manutenção ou uso do produto. Estes cenários operacionais podem estar representados por casos de uso ou casos de teste;
- Estabelecer uma definição das funcionalidades requisitadas e dos atributos de qualidade: os conceitos e cenários operacionais passam por análise para definir a funcionalidade requerida e os atributos qualitativos objetivando descrever o que é esperado do produto. A descrição funcional obtida inclui ações, sequências, entradas, saídas e outras informações que comuniquem como o produto será utilizado;
- Analisar requisitos: análise dos requisitos para determinar se eles são necessários e suficientes para atender os objetivos do produto e do projeto;
- Analisar requisitos para alcançar equilíbrio: busca obter e manter o equilíbrio entre custo, cronograma, qualidade e outras restrições do projeto;

- Validar requisitos: os requisitos são validados desde o início do projeto junto aos *stakeholders* por meio de protótipos, simulações, cenários, entre outras técnicas. O *feedback* obtido pela validação dá origem a outras necessidades e requisitos não expostos anteriormente.

2.4.2 Gerenciamento de Requisitos

Todos os requisitos gerados pela área de processo descrita na seção anterior são gerenciados pela área denominada Gerenciar Requisitos, que pertence ao nível 2 de maturidade do CMMI. O objetivo desta área de processo é garantir que os requisitos criados auxiliem o planejamento e a execução das necessidades do projeto.

Parte do gerenciamento de requisitos envolve a documentação de mudanças (incluindo sua razão) e a manutenção bidirecional da rastreabilidade de requisitos entre o requisito fonte e todos os requisitos de produto e de componente que dele foram derivados.

As mudanças ocorrem a partir do melhor entendimento dos requisitos, da evolução das necessidades, de mudanças tecnológicas, de regulamentações, entre diversos motivos. Se a mudança gerar um novo requisito ele é direcionado aos processos da área de Desenvolvimento de Requisitos, caso ela se aplique a um requisito já existente os processos utilizados pertencerão à área de Gerenciamento de Requisitos.

Ao contrário da área de processo anterior, esta área possui apenas um objetivo específico que está detalhado na subseção seguinte.

Uma vez que este é o único objetivo específico desta área de processo, ela abrange essencialmente o que já foi referido para a própria área.

Os processos de gerência de requisitos possuem relação muito próxima aos de desenvolvimento de requisitos, especialmente os dois primeiros processos entre os cinco desta área, que podem ocorrer concomitantemente e estão detalhados abaixo:

- Compreender os requisitos: define critérios que determinam as fontes apropriadas de requisitos, além de critérios de avaliação e aceitação de requisitos. Inclui também a análise dos requisitos para verificar se os critérios estabelecidos foram atendidos;
- Obter comprometimento com os requisitos: enquanto o processo anterior foca em obter e analisar os requisitos de fontes adequadas, este processo procura obter compromisso daqueles que vão executar as atividades necessárias para implementar os requisitos;
- Gerenciar mudanças nos requisitos: sabendo-se a fonte e a razão de determinado requisito, este processo analisa o impacto de mudanças a ele aplicadas e mantém um histórico de mudanças associado ao requisito;
- Manter rastreabilidade bidirecional dos requisitos: procura garantir que a rastreabilidade possa ser obtida de requisitos de sua origem até requisitos de baixo nível e de requisitos de baixo nível de volta a sua origem. Rastreabilidade bidirecional auxilia a determinar se todos os requisitos foram tratados e se todos os requisitos de baixo nível podem ser relacionados a uma fonte válida.

- Garantir o alinhamento entre trabalho do projeto e os requisitos: busca por inconsistências entre requisitos e outros produtos de trabalho do projeto e define ações corretivas para resolvê-las.

2.5 Trabalhos Relacionados

Ramos et al [10] definem um padrão de encapsulamento e modularização de requisitos como uma solução para o problema de duplicação de requisitos e apresenta exemplos práticos aplicados a casos de uso. Quando um requisito duplicado é identificado, um novo caso de uso é criado e passa a ser referenciado onde ocorria a duplicação. A solução é baseada na ideia de orientação a aspectos utilizada no desenvolvimento do código fonte.

O trabalho de Ramos aborda duplicação de requisitos em um mesmo tipo de documento, seja especificação de caso de uso, estória ou outro padrão de documentação, entretanto não aborda a duplicação que ocorre entre vários tipos de documentos como por exemplo a relação entre especificação de caso de uso e documentação de código fonte, que é descrita nesta dissertação.

A abordagem para tratamento de duplicação entre formatos e padrões distintos de documentação mais próxima ao estudo deste trabalho está relacionada a ferramentas que obtém requisitos não pertencentes à regra de negócio à partir do código fonte com o objetivo de gerar documentação de API para desenvolvedores de software.

Embora existam diversas ferramentas deste gênero para várias linguagens, o Javadoc, uma ferramenta que analisa os blocos de comentário em um conjunto de classes Java e produz sua respectiva documentação no formato de páginas HTML, foi escolhido na comparação porque funciona com Java, a mesma linguagem de programação a que se aplica o Gaiadoc.

2.5.1 Javadoc

A criação do Javadoc, de acordo com Kramer [20], deu-se em virtude da necessidade de se evitar que a documentação de especificação da API Java seja fragmentada. Apesar de possuir uma única documentação, por diversos motivos, documentações secundárias existiam em outras fontes, algumas vezes, complementando a principal e, outras vezes, sobrepondo-a.

O primeiro passo na especificação do Javadoc foi decidir para quem a documentação seria escrita. Os possíveis públicos identificados eram as centenas de milhares de desenvolvedores de aplicações, as centenas de organizações que utilizavam a licença Java para incorporar a linguagem em seus sistemas operacionais e as dezenas de engenheiros de testes de conformidade.

Na sequência, foi definido de forma unânime, que a documentação deveria residir no código-fonte com a finalidade de gerar uma documentação atualizada e precisa. Considerando que há uma ligação forte entre a documentação e o código, há uma maior conveniência para os desenvolvedores da API, tendo em vista ser mais simples escrever a documentação no mesmo local em que escrevem o código.

O último passo consistiu na definição de diretrizes que instruíssem o modo em que os

desenvolvedores deveriam proceder na implementação da API e também a forma em que a documentação deveria ser escrita.

A figura 2.4 demonstra um exemplo de documentação de código fonte da classe String da biblioteca Java o padrão de anotações do Javadoc e a figura 2.5 o resultado gerado pela ferramenta em HTML.

```
/**
 * The <code>String</code> class represents character strings. All string
 * literals in Java programs, such as <code>"abc"</code>, are implemented as
 * instances of this class.
 * Strings are constant; their values cannot be changed after they are created.
 * String buffers support mutable strings. Because String objects are immutable
 * they can be shared. For example:
 * @author Lee Boynton
 * @author Arthur van Hoff
 * @author Martin Buchholz
 * @see java.lang.Object#toString()
 * @since JDK1.0
 */
public final class String {
    /**
     * Returns <tt>true</tt> if, and only if, {@link #length()} is <tt>0</tt>.
     *
     * @return <tt>true</tt> if {@link #length()} is <tt>0</tt>, otherwise
     *         <tt>false</tt>
     *
     * @since 1.6
     */
    public boolean isEmpty() {}
    /**
     * Returns the <code>char</code> value at the specified index. An index
     * ranges from <code>0</code> to <code>length() - 1</code>. The first
     * <code>char</code> value of the sequence is at index <code>0</code>, the
     * next at index <code>1</code>, and so on, as for array indexing.
     *
     * <p>
     * If the <code>char</code> value specified by the index is a <a
     * href="Character.html#unicode">surrogate</a>, the surrogate value is
     * returned.
     *
     * @param index
     *            the index of the <code>char</code> value.
     * @return the <code>char</code> value at the specified index of this
     *         string. The first <code>char</code> value is at index
     *         <code>0</code>.
     * @exception IndexOutOfBoundsException
     *                if the <code>index</code> argument is negative or not less
     *                than the length of this string.
     */
    public char charAt(int index) {}
}
```

Figura 2.4 – Parte da classe String da API Java documentada por meio do Javadoc.

Class String

java.lang.Object
    java.lang.String

All Implemented Interfaces:

Serializable, CharSequence, Comparable<String>

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

The `String` class represents character strings. All string literals in Java programs, such as `"abc"`, are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

```
String str = "abc";
```

is equivalent to:

```
char data[] = {'a', 'b', 'c'};
String str = new String(data);
```

Here are some more examples of how strings can be used:

```
System.out.println("abc");
String cde = "cde";
System.out.println("abc" + cde);
String c = "abc".substring(2,3);
String d = cde.substring(1, 2);
```

The class `String` includes methods for examining individual characters of the sequence, for comparing strings, for searching strings, for extracting substrings, and for creating a copy of a string with all characters translated to uppercase or to lowercase. Case mapping is based on the Unicode Standard version specified by the `Character` class.

The Java language provides special support for the string concatenation operator (+), and for conversion of other objects to strings. String concatenation is implemented through the `StringBuilder`(or `StringBuffer`) class and its append method. String conversions are implemented through the method `toString`, defined by `Object` and inherited by all classes in Java. For additional information on string concatenation and conversion, see Gosling, Joy, and Steele, *The Java Language Specification*.

Unless otherwise noted, passing a `null` argument to a constructor or method in this class will cause a `NullPointerException` to be thrown.

A `String` represents a string in the UTF-16 format in which *supplementary characters* are represented by *surrogate pairs* (see the section Unicode Character Representations in the `Character` class for more information). Index values refer to `char` code units, so a supplementary character uses two positions in a `String`.

The `String` class provides methods for dealing with Unicode code points (i.e., characters), in addition to those for dealing with Unicode code units (i.e., `char` values).

Since:

JDK1.0

Figura 2.5 – Documentação da classe String gerada pelo Javadoc em HTML.

3 PROCESSO DE DOCUMENTAÇÃO DE CÓDIGO FONTE GAIADOC

3.1 Metodologia

Os objetivos iniciais da metodologia proposta foram:

- Preencher uma lacuna nos métodos de documentação de código utilizados atualmente para classes que representam a regra de negócio;
- Evitar a duplicação da documentação dos requisitos da regra de negócio dentre os artefatos do projeto, resultando em menor esforço para sua manutenção e maior rastreabilidade entre os requisitos, a implementação e o produto entregue ao cliente;
- Estabelecer um ambiente com regras mais claras para DCF que possa orientar melhor o modo em que a documentação deve ser escrita.

Visto que as anotações presentes no código fonte exigem regras bem definidas, o passo seguinte consistiu em determinar qual seria o documento de requisitos resultante. Devido à popularidade da UML na elicitação dos requisitos de projetos de desenvolvimento de software, casos de uso e suas respectivas especificações foram estabelecidas como os principais artefatos para a abordagem proposta.

Como consequência, apesar de a UML não se prender a um PDS específico, foi natural adotar a disciplina de requisitos do RUP, considerando que o RUP foi especificamente desenvolvido utilizando a UML.

3.1.1 Mapeamento entre a Documentação do Código Fonte e a Especificação de Caso de Uso

O mapeamento entre a documentação de código das classes e a ECU é feito através de anotações dentro dos blocos de comentários da linguagem Java, de modo similar ao Javadoc.

Anotações localizadas em blocos precedidos por /** e encerrados por */ são analisadas pela ferramenta, assim como pelo Javadoc. Qualquer bloco de comentário contido por /* e */ e comentário de linha antecedido por // é ignorado.

A tabela 3.1 apresenta e descreve o conjunto de anotações adotado, incluindo o contexto e a descrição de cada anotação.

As seguintes regras são definidas para o mapeamento do escopo de classe:

Tabela 3.1 – Especificação das anotações utilizadas pela ferramenta GaiaDoc.

Nome	Contexto	Descrição
@name	Classe.	Nome do caso de uso.
@description	Classe, método e variável.	Descrição do caso de uso, do atributo ou do passo dentro de um fluxo de eventos.
@writer	Classe.	Um escritor da documentação (que não é necessariamente o programador, que seria melhor definido através da anotação @author, do Javadoc).
@performer	Classe e método.	Um ator que executa as ações do caso de uso.
@extension	Classe.	Descreve uma extensão do caso de uso.
@specialRequirement	Classe.	Descreve um requisito especial do caso de uso.
@main	Classe.	Quando um caso de uso é mapeado em múltiplas classes essa anotação indica qual é a principal entre elas.
@lessonLearned	Classe.	Descreve uma lição aprendida do caso de uso.
@preCondition	Classe e método.	Descreve uma pré-condição do caso de uso.
@postCondition	Classe e método.	Descreve uma pós-condição do caso de uso.
@basicFlow	Método.	Indica que o método faz parte do fluxo básico de eventos do caso de uso.
@alternativeFlow	Método.	Indica que o método faz parte de um determinado fluxo alternativo de eventos do caso de uso.

1. Toda classe que represente requisitos de regra de negócio devem conter a anotação *@name* em seu bloco de comentário, indicando a partir de que caso de uso foi mapeada;

2. Sempre que um caso de uso é mapeado em mais de uma classe, uma delas deve conter a anotação *@main*;

3. A anotação *@Description* é requerida quando o caso de uso é mapeado em apenas uma classe. Se o caso de uso é mapeado em diferentes classes, ao menos uma delas deve conter esta anotação, e se mais de uma classe tiver a anotação, então aquela que tiver a anotação *@main* será a adotada na ECU resultante;

4. As anotações *@writer* e *@performer* são obrigatórias e podem ser repetidas várias vezes, caso a documentação seja escrita por mais de uma pessoa ou há mais de um ator no caso de uso;

5. No escopo de classe há também as anotações *@extension*, que representa uma extensão do caso de uso, e *@specialRequirement*, que representa um requisito especial do caso de uso; *@lessonLearned* Lições aprendidas relacionadas ao caso de uso podem ser inseridas no escopo da classe para posteriormente ser cadastradas no sistema de gerenciamento de lições aprendidas do portal corporativo Gaia. Um identificador único deve ser passado como parâmetro da anotação para impedir que uma lição aprendida seja cadastrada múltiplas vezes sempre que a GaiaDoc for executada;

6. No caso das anotações *@writer*, *@performer*, *@extension* e *@specialRequirement*, quando o caso de uso é mapeado em diversas classes, estas propriedades são mescladas.

7. As anotações *@preCondition* e *@postCondition* incluídas no escopo de classe são incluídas no início da ECU, acompanhando os requisitos especiais e as extensões.

A figura 3.1 exibe o modelo de documentação para o escopo de classe.

```
/**
 * @main
 * @name Nome do caso de uso.
 * @description Descrição do caso de uso.
 * @writer Nome do escritor da especificação de caso de uso
 * @performer Ator do caso de uso.
 * @performer Segundo ator do caso de uso.
 * @extension Extensão do caso de uso.
 * @lessonLearned('id') Lição aprendida relacionada ao caso de
 * uso.
 */
public class NomeDaClasse{}
```

Figura 3.1 – Modelo de documentação no bloco de comentário da classe.

As variáveis da classe, em conjunto com sua descrição, são utilizadas para gerar o Glossário, que é um dicionário de termos relacionados ao caso de uso no formato de uma tabela. As seguintes regras são adotadas para este escopo:

1. Apenas a anotação *@description* é permitida. Caso esta anotação não esteja presente, o atributo é ignorado durante a geração da ECU;
2. O nome do atributo é obtido do nome da variável a que pertence o bloco de comentário.

O modelo de documentação no escopo da variável é exemplificado pela figura 3.2.

```
/**
 * @description Descrição do atributo.
 */
private int NomeDoAtributo;
```

Figura 3.2 – Modelo de documentação no bloco de comentário da variável.

Os métodos da classe representam o fluxo de eventos do caso de uso. Um fluxo de eventos pode ser dividido em diferentes classes, desde que todas pertençam ao mesmo caso de uso. As regras para seu bloco de comentário estão listadas abaixo:

1. O bloco de comentário do método deve conter a anotação *@description*, que não pode ser repetida no mesmo bloco;
2. Por padrão, é considerado que o ator que dispara o evento é aquele definido pela anotação *@performer* no escopo da classe. Se a mesma anotação estiver presente no escopo do método, esse valor será sobrescrito.
3. Se o evento tiver pré ou pós-condições, as anotações *@preCondition* e *@postCondition* podem ser utilizadas quantas vezes for necessário. Quando incluídas no escopo de método ambas as anotações são inseridas na ECU como notas de rodapé com seu índice adicionado no passo do fluxo de eventos ao qual pertencem;
4. É obrigatório, no escopo do método, que a anotação *@basicFlow* ou *@alternativeFlow* esteja presente. Entretanto ambas não podem ser utilizadas juntas;

5. A anotação *@alternativeFlow* tem um parâmetro indicando para que fluxo alternativo de eventos o método pertence.

De acordo com as delimitações especificadas acima, o modelo para documentação de método é o demonstrado na figura 3.3.

```
/**
 * @basicFlow(1)
 * @description Descrição do evento 1 no fluxo básico de eventos.
 * @preCondition Pré-condição do evento.
 */
public String NomeDoMetodo(int parametro1, ...){}
```

Figura 3.3 – Modelo de documentação no bloco de comentário do método.

Após a leitura das anotações no arquivo que contém a classe, o GaiaDoc converte estes dados para a estrutura de classes demonstrada pela figura 3.4.

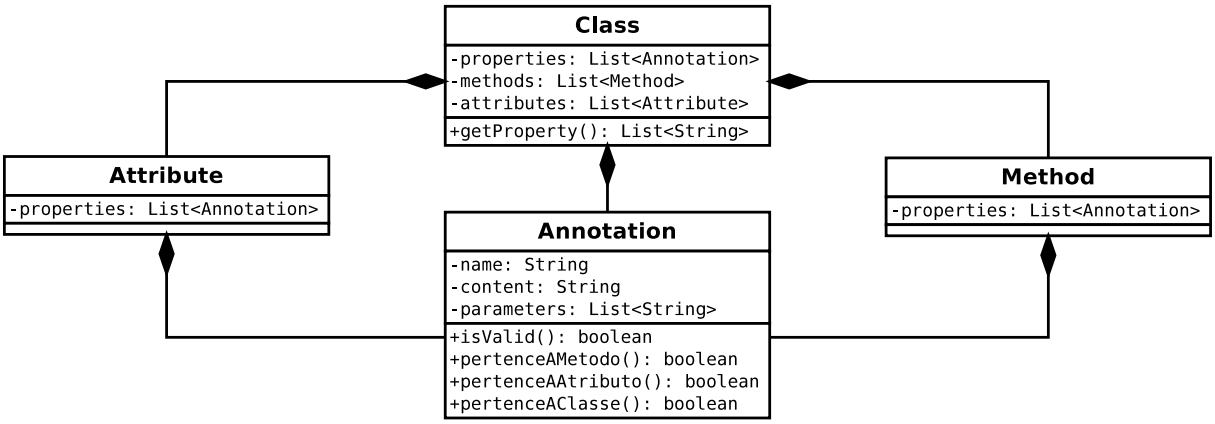


Figura 3.4 – Diagrama de classe da estrutura que representa os dados lidos do arquivo Java pela ferramenta GaiaDoc.

Os objetos da classe *Annotation* contém os dados referentes a cada anotação individualmente, como seu nome, seu conteúdo e seus parâmetros, caso se apliquem. Além disso ela disponibiliza métodos que auxiliam na identificação de seu escopo e também na verificação de sua validade.

Os objetos de *Class* armazenam um conjunto de propriedades, que são na realidade objetos do tipo *Annotation*, que pertencem ao seu escopo. Ela também possui um conjunto de métodos e de atributos. Cada anotação de método encontrada na classe representa um objeto do tipo *Method* que armazena as propriedades relacionadas a seu escopo. O mesmo ocorre com o conjunto de objetos do tipo *Atribute* que contém as anotações associadas à variável.

3.1.2 A Ferramenta GaiaDoc

A ferramenta GaiaDoc foi desenvolvida para fazer o mapeamento entre as anotações descritas na seção anterior, presentes no código fonte, e a ECU, que deve ser gerada como saída.

Ela foi projetada para ser executada via linha de comando, recebendo dois parâmetros (como demonstrado pela figura 3.5): o primeiro é o diretório raiz em que estão contidas todas as classes que

devem ser analisadas; o segundo é o diretório de saída no qual são salvas as especificações de caso de uso geradas, a partir das anotações das classes, em PDF (Portable Document Format).

Figura 3.5 – Ferramenta GaiaDoc executada via terminal no linux.

Há, também, parâmetros opcionais para remover partes específicas da especificação de caso de uso como, por exemplo, o glossário. Neste caso deve-se adicionar o parâmetro *-r glossary*.

O PDF foi escolhido como formato de saída com o objetivo de dificultar a modificação dos requisitos por meio do documento gerado, incentivando a alteração diretamente no código fonte.

No momento em que o comando é executado, a ferramenta percorre todos os arquivos do diretório indicado e seus subdiretórios. Cada classe encontrada é, então, analisada sintaticamente para verificar se contém todas as anotações exigidas e, em caso afirmativo, a documentação é obtida do arquivo e utilizada para gerar sua especificação de caso de uso.

A análise léxica é realizada, pois qualquer anotação inválida impede a produção da especificação de caso de uso. Algumas anotações fora do escopo do GaiaDoc podem também ser aceitas, o que é útil no caso da utilização de outras anotações para informações complementares como, por exemplo, o uso da anotação *@author*, que pertence ao domínio do Javadoc, para especificar quem é o autor do código, enquanto a anotação *@writer* indica quem é o autor da documentação dos requisitos de regra de negócio.

Durante a execução da análise sintática, as anotações são localizadas nos blocos de comentário e classificadas como *tokens* pelo analisador. A leitura do arquivo que comporta a classe deve ocorrer na sequência esquematizada pelo diagrama de estados representado pela figura 3.6.

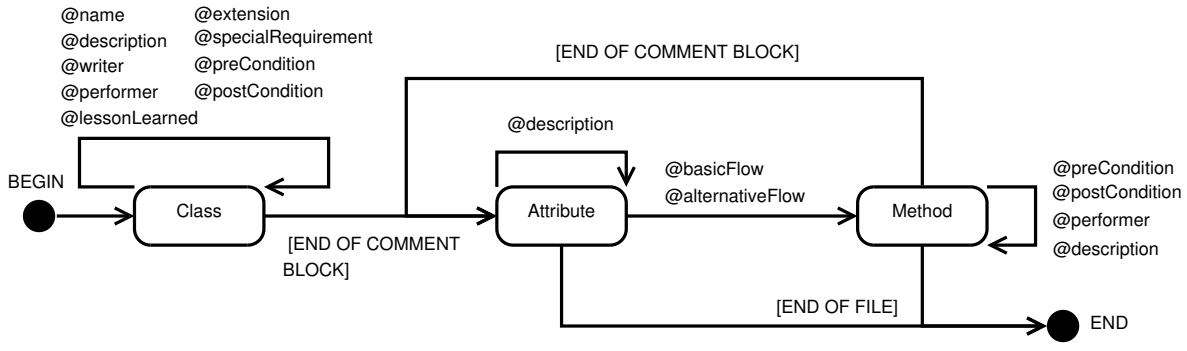


Figura 3.6 – Diagrama de Estados do analisador sintático.

Caso o analisador sintático não siga o fluxo previsto pelo diagrama de estados, a documentação é considerada inválida e não é gerada a ECU para a classe. Visto que a análise léxica não é realizada, apenas as anotações GaiaDoc são levadas em consideração no avanço de um estado para outro e na determinação de documentação inválida.

Quando ocorrem alterações na documentação dos requisitos no código fonte, é necessário apenas re-executar o comando para sobreescrever as especificações geradas ou armazená-las em outro diretório. O versionamento da documentação é gerenciado diretamente pelo código fonte, portanto para obter uma

versão prévia da ECU, é necessário apenas baixar a versão desejada do repositório SVN e executar a ferramenta GaiaDoc sobre esse código.

3.1.3 Comparação entre o Javadoc e o GaiaDoc

Ambas são ferramentas que funcionam com a linguagem Java e apesar de a fonte dos requisitos ser a mesma, o público-alvo de cada uma é distinto. O GaiaDoc se foca nos requisitos da regra de negócio e tem como alvo os diversos *stakeholders* do projeto, incluindo o cliente, e portanto utiliza linguagem simples e clara. Já o Javadoc se foca em usuários de APIs e, portanto, sua documentação possui uma abordagem mais técnica. Deste modo as duas ferramentas não são concorrentes, mas complementares. É responsabilidade do analista de requisitos, em conjunto com a equipe de desenvolvimento, estabelecer se a classe deve ser documentada com o uso do Javadoc, do GaiaDoc, ou, em alguns casos, com a utilização de ambos.

Em relação ao PDS, apesar de poder ser adaptado a outros processos, o GaiaDoc, à princípio, foi especificado para ser utilizado exatamente em conjunto com o RUP e com a UML, possuindo inclusive alguns pré-requisitos, como os diagramas de caso de uso e de classe, assim como a especificação de caso de uso como saída. O controle de mudanças no código-fonte é mais rigoroso, pois a especificação de caso de uso passa a estar integrada ao código-fonte.

Já o Javadoc pode ser utilizado tanto com processos prescritivos, quanto com processos ágeis, não existindo nenhuma limitação com relação ao processo. Há apenas um guia de boas práticas sobre o modo em que a documentação deve ser escrita e como as anotações devem ser utilizadas para se tirar um maior proveito da ferramenta.

A tabela 3.2 sumariza os atributos utilizados na comparação entre as duas ferramentas.

Tabela 3.2 – Comparativo entre as ferramentas Javadoc e GaiaDoc.

Atributo de comparação	Javadoc	GaiaDoc
Conteúdo da documentação	Documentação de API.	Requisitos de regras de negócio do projeto.
Público-alvo	Usuários de APIs	*Stakeholders* do projeto
Tipo de linguagem	Técnica	Simples, clara e compreensível aos *stakeholders*
Escritor da documentação	Programador	Analista de requisitos
Dependências	Java	Java, UML e RUP
Mapeamento da documentação	Baseado em anotações.	Baseado em anotações.
Formato do documento de saída	HTML, extensível a outros formatos por meio de doclets	PDF.

3.2 Fluxo de Requisitos

Apesar da modelagem de caso de uso não limitar a escolha do Processo de Desenvolvimento de Software (PDS), é frequente que seja utilizada em conjunto com o RUP, sendo este o principal motivo de

sua escolha como base do fluxo de requisitos do GaiaDoc.

A adoção da nova abordagem apresentada neste trabalho exige algumas alterações no fluxo padrão de requisitos do RUP, que é tratado essencialmente pela Disciplina de Requisitos.

O primeiro artefato gerado por esta disciplina é o documento de visão, que estabelece qual o problema a ser resolvido pelo sistema de software, qual a delimitação do escopo, como também quem são as pessoas interessadas no projeto.

Os requisitos são, então, identificados através de entrevistas com os *stakeholders* e classificados em funcionais, que são especificados por intermédio de casos de uso, e não funcionais, que são documentados como uma especificação suplementar.

Seguindo a modelagem tradicional do RUP, são, então, refinados os requisitos funcionais, representados pela ECU em uma linguagem compreensível para todas as partes interessadas no projeto. Essa especificação é usualmente disponibilizada no formato de documento de texto, compreendendo a descrição do caso de uso, a lista de atores que dele participam e o fluxo de eventos básico e alternativos.

Embora não esteja no escopo do RUP, posteriormente boa parte dos requisitos já documentados são duplicados na documentação de código fonte, aumentando o esforço para manter ambos atualizados com a última versão do código.

Por esta razão, a abordagem da metodologia proposta neste trabalho diverge da tradicional recomendada pelo RUP depois da definição do diagrama de caso de uso. Após a identificação dos casos de uso, é definido um diagrama de classes preliminar e, posteriormente, é criado o esqueleto do projeto, não havendo sua especificação em documentos de texto.

O esqueleto do projeto possui as classes especificadas no diagrama, seus atributos e métodos (sem implementação), como também a documentação por este trabalho sugerida.

O ato de criar o esqueleto do projeto melhora a compreensão dos requisitos do projeto por parte dos analistas, e como consequência, o diagrama de classes (e outros artefatos da UML) é refinado. Estas melhorias são propagadas de volta ao esqueleto e ao diagrama de classes, gerando, conseguintemente, um processo de refinamento. A figura 3.7 sumariza o fluxo de criação dos artefatos na disciplina de requisitos.

Após a implementação do esqueleto do projeto, as classes estão prontas para ser documentadas pelo analista de requisitos através do uso das anotações descritas previamente. Quando encerrada a documentação, as especificações são, então, geradas no formato PDF e distribuídas aos *stakeholders*.

No início da fase de construção do sistema, a maior parte dos requisitos já se encontra documentada no código fonte. Durante todo o desenvolvimento do projeto, os requisitos são iterativamente refinados, alterados e incrementados, desde que as mudanças tenham sido aprovadas pelo Comitê de Gerenciamento de Mudanças. Como consequência, todos os artefatos podem ser modificados, incluindo os casos de uso, o diagrama de classes, as especificações de caso de uso localizadas no código fonte e até o próprio código.

As correções de falhas identificadas na documentação podem ser registradas em um sistema

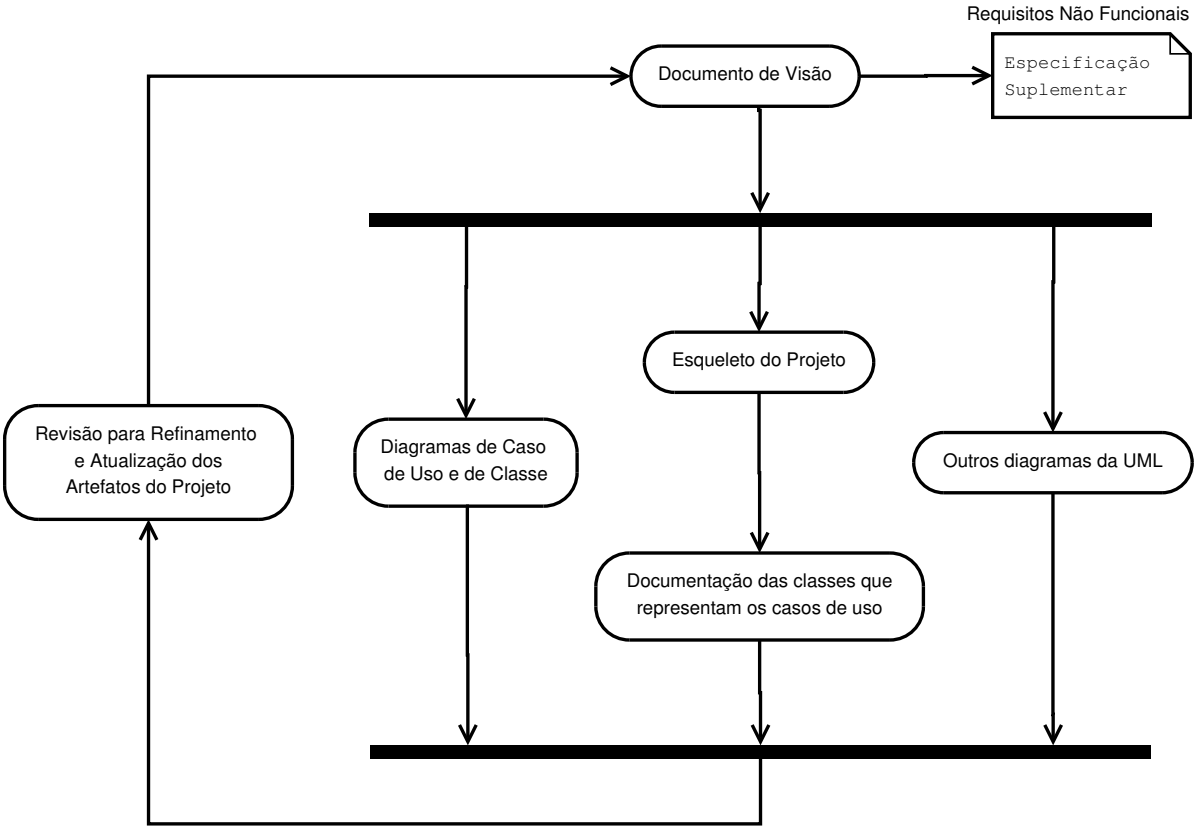


Figura 3.7 – Diagrama de atividade que representa o fluxo de criação dos artefatos na disciplina de requisitos.

de *bug tracking*, auxiliando a manter a rastreabilidade das mudanças efetuadas nos requisitos documentados e também definindo as prioridades dessas modificações em relação a correções do próprio código fonte. Desse modo, é mantido um repositório centralizado de todas as mudanças realizadas tanto no código, quanto na documentação.

Atualmente, o versionamento é quase sempre aplicado ao código fonte e sistemas de rastreamento de erros estão sempre presentes nos projetos de desenvolvimento de software, portanto a adição deste recurso ao gerenciamento de requisitos representaria pouco ou nenhum custo para a organização.

O resultado das práticas descritas é um código muito bem documentado, atualizado e em conformidade com os outros artefatos do projeto.

Embora o RUP não aborde a documentação de código fonte, em projetos reais é comum que a documentação dos requisitos seja duplicada, ainda que em linguagens e níveis de profundidade distintos, e, portanto, a atualização dos requisitos se torna mais dispendiosa para a organização.

Caso a organização opte por documentar um artefato em detrimento de outro, com o objetivo de diminuir os gastos, a integridade entre os artefatos do projeto é quebrada.

No método proposto, o custo de atualização dos artefatos que contêm os requisitos do projeto é diminuído, pois a especificação em linguagem natural dos requisitos funcionais se concentra em apenas um local. O analista de requisitos atualiza a documentação situada no código fonte e utiliza a ferramenta GaiaDoc para gerar novamente a ECU em sua versão atualizada.

A desvantagem dessa abordagem, assim como a de diversas outras técnicas de engenharia

de software, é que a duração das fases de análise e planejamento se torna maior. Apesar disso, a expectativa é que a duração das fases subsequentes seja consideravelmente reduzida, gerando um custo-benefício positivo para o projeto.

3.2.1 Rastreabilidade de Requisitos

Segundo Lamsweerde [21] a evolução dos requisitos levanta um complicado problema no gerenciamento da informação. Grandes quantidades de informação precisam ser versionadas e mantidas em um estado consistente. Mudanças no documento de requisitos devem ser propagadas aos outros itens que dele dependem para a manutenção da consistência entre eles.

O objetivo geral do gerenciamento da rastreabilidade de requisitos é auxiliar na manutenção da consistência dos requisitos na presença de mudanças, assegurando que seu impacto seja facilmente localizável por meio de *links* de rastreabilidade entre os artefatos do projeto. Um artefato é rastreável se podemos afirmar de onde ele vem, para onde vai, para que e como será utilizado [21].

Os *links* de rastreabilidade podem ser classificados em quatro tipos:

- Dependência: é o *link* entre um determinado item de requisitos que afeta um segundo item, ou seja, ocorre dependência do segundo item, que é afetado, para o primeiro;
- Revisão: resulta da evolução ao longo do tempo para aprimorar ou corrigir o documento e deve conter a data de modificação, seu autor, seus colaboradores e a razão da mudança;
- Uso: ocorre se a alteração de um determinado item de requisitos torna um segundo item incompleto, inadequado, inconsistente ou ambíguo;
- Derivação: existe se um item é construído à partir de outro.

O *link* de revisão, de acordo com a abordagem proposta, pode ter os colaboradores e a razão da mudança gerenciados por um sistema de rastreamento de erros (adaptado para ser usado como um sistema de rastreamento de requisitos). Todas outras informações relacionadas a uma nova versão da ECU são facilmente gerenciadas por uma ferramenta de versionamento como mencionado anteriormente. O gerenciamento de qualquer outro artefato além do ECU requer que o versionamento seja aplicado externamente ao código fonte ou que eles sejam incluídos em um diretório no projeto de implementação.

Além da gestão do versionamento dos documentos de requisitos, a dependência entre os requisitos contidos em cada artefato deve ser monitorado. No âmbito do presente documento um vínculo de rastreabilidade vertical pode ser encontrado entre:

- O diagrama de classe e o esqueleto do projeto indicando derivação;
- O diagrama de caso de uso e o ECU indicando derivação;
- O ECU e o código fonte que representa dependência;
- O esqueleto do projeto e o código fonte que também representa dependência.

Um link de rastreabilidade horizontal pode ser encontrado entre o esqueleto do projeto e da ECU.

O grafo da figura 3.8 ilustra as relações de vínculo de rastreabilidade entre os artefatos do projeto.

A vantagem desta proposta é que a rastreabilidade entre os artefatos pode ser facilmente identificada, o que é muito importante uma vez que, especialmente no início da especificação, muitas alterações podem ser feitas no esqueleto do projeto enquanto os requisitos estão sendo melhor compreendidos.

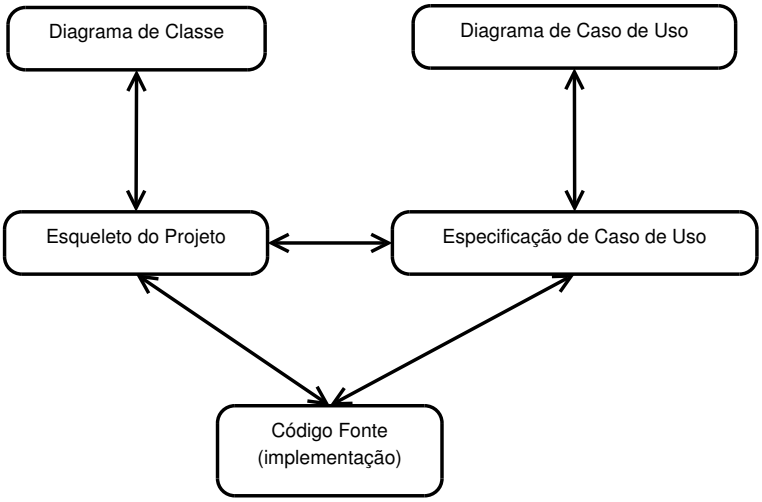


Figura 3.8 – Grafo de rastreabilidade entre os artefatos no fluxo de requisitos do GaiaDoc.

Uma das questões importantes relacionada ao tema é até que ponto a gestão de rastreabilidade traz um custo-benefício positivo ao projeto. Alguns dos fatores determinantes nesta decisão são:

- O tamanho estimado do projeto (medido em pessoa-hora);
- A suscetibilidade de mudança nos requisitos;
- O tempo em que o produto ficará disponível ao usuário após entregue;
- A quantidade de requisitos críticos. Alguns destes fatores (como o (ii) e o (iv)) podem ser utilizados para selecionar apenas parte dos artefatos do projeto que serão rastreados durante sua execução, sendo esta uma solução intermediária que pode ser a ideal em diversos casos.

Quanto à manutenção da rastreabilidade de requisitos, o RUP propõe o uso de uma matriz de rastreabilidade que pode ser representada no formato de um grafo de dependências, auxiliando assim a decidir se a mudança é viável e qual seu custo, além de facilitar seu planejamento.. O processo da GaiaDoc mantém a rastreabilidade dos requisitos pela técnica de referência cruzada entre implementação e ECU, inserindo *links* na ECU que indica quais classes foram utilizadas em sua criação.

A técnica de referência cruzada adotada na GaiaDoc pode ser eficiente em projetos pequenos que não queiram acrescentar o custo adicional na gerência de rastreabilidade ou que considere apenas a ECU como artefato crítico. Se o projeto exigir uma gestão mais rigorosa da rastreabilidade a referência cruzada,

assim como a matriz de rastreabilidade ou grafos de rastreabilidade, não serão técnicas eficientes, sendo preferível adotar listas ou banco de dados de rastreabilidade [21].

3.2.2 Conformidade do Novo Fluxo de Requisitos em Relação ao CMMI

Este trabalho não focou na relação e integração entre o fluxo de requisitos do GaiaDoc e o CMMI. Isso se deve ao fato de que o fluxo de requisitos apresentado é baseado no RUP, sendo usual que empresas certificadas CMMI o adotem como a base de seu PDS e que diversos estudos (como [22] e [23]) já foram realizados constatando que, de fato, o RUP atende boa parte dos objetivos das áreas de processo de requisitos do CMMI.

O objetivo foi o de analisar e validar possíveis impactos e benefícios que as modificações efetuadas sobre o fluxo de requisitos do RUP podem causar em projetos de organizações que buscam se certificar pelo CMMI.

Além de não haver foco na integração entre RUP e CMMI, o escopo da análise de relação e impacto entre o fluxo de requisitos do GaiaDoc e o CMMI se restringiu apenas à área de requisitos, sendo identificadas as áreas de processo do CMMI e as disciplinas do RUP relevantes.

Foi necessário identificar as áreas de processo do CMMI e as disciplinas do RUP que entram no escopo da definição do fluxo de requisitos para adoção da ferramenta GaiaDoc. Duas áreas de processo estão relacionadas aos requisitos: Desenvolvimento de Requisitos e Gerência de Requisitos. Do lado do RUP inicialmente foi identificada apenas a disciplina de Requisitos, assim como no estudo de [24], entretanto, após a análise das duas áreas de processo do CMMI foi identificado que a disciplina de Modelagem de Negócios tem relação com as práticas iniciais de desenvolvimento de requisitos de cliente, a disciplina de Teste tem relação com práticas de validação do produto e a disciplina de Configuração e Gerência de Mudança possui práticas em comum com a área de processo de gerência dos requisitos.

Apesar do CMMI apresentar a área de processo de Desenvolvimento de Requisitos antes da área de Gerência de Requisitos (mesma ordem adotada neste trabalho), a segunda área é adotada primeiro por estar no segundo nível de maturidade, apresentando principalmente práticas de gerência de mudanças e rastreabilidade, enquanto o Desenvolvimento de Requisitos pertence ao terceiro nível e introduz mais práticas para gerência de requisitos, como análise e validação, além de diretrizes para a definição de requisitos de cliente e de produto.

Na área de processos **Desenvolvimento de Requisitos**, os requisitos de cliente, de acordo com o RUP, são tratados essencialmente pela disciplina de modelagem de negócios. As necessidades dos *stakeholders* são registradas no documento de visão de negócio e representadas por meio de casos de uso preliminares, que identificam seus papéis na organização e na relação com o produto a ser desenvolvido.

Os casos de uso são refinados e o diagrama preliminar de classes é desenvolvido. Assim como descrito anteriormente, depois da criação do esqueleto de um projeto Java correspondente ao diagrama de classes, é realizada a especificação de caso de uso no padrão GaiaDoc. Este conjunto de artefatos, que representa os requisitos de produto, é dividido em componentes que serão entregues a cada iteração do

processo.

O cenário e o fluxo de eventos presentes nas especificações de caso de uso são utilizados na análise de impacto dos requisitos para determinar se as necessidades dos *stakeholders* serão atendidas e para estabelecer a *baseline* do produto. Na validação de protótipos, testes unitários e de integração e *storyboards* dos casos de uso são utilizados para verificar se o componente entregue está em conformidade com os requisitos documentados.

Em suma, de acordo a análise realizada, as mudanças efetuadas no RUP para o uso da ferramenta GaiaDoc não trouxeram nenhum impacto positivo ou negativo no desenvolvimento de requisitos de cliente, de produto ou da análise e validação dos requisitos, visto que duplicação ou reaproveitamento de requisitos não é o objetivo desta área de processos do CMMI.

Se o desenvolvimento de requisitos possui práticas de gerência de requisitos associados à análise e validação dos requisitos, a área de **Gerência de Requisitos** se preocupa com a gestão de mudanças e na manutenção da rastreabilidade entre os artefatos do projeto.

No processo tradicional do RUP, as mudanças, sejam por falhas ou por novos requisitos, são identificadas e submetidas à aprovação de um Comitê de Gerenciamento de Mudanças. Se aprovada ela é encaminhada às fases iniciais de desenvolvimento de requisitos até ser integrada ao produto.

O fluxo de requisitos proposto não visa nenhuma mudança na abordagem de como os requisitos devem ser identificados e tratados, apenas sugere o uso de sistemas de gerenciamento de erros (*bug tracking*) na gestão das falhas e adaptação a novos requisitos identificados que causam impacto a requisitos existentes.

3.2.3 Relação entre o Escritor da Documentação e o Programador

Algumas regras devem ser estabelecidas na relação entre o programador e o escritor da documentação - usualmente, o analista de requisitos.

O programador pode e deve realizar comentários simples (precedidos por //), que não são mapeados pelas ferramentas geradoras de documentação, ao passo que o escritor da documentação pode editar livremente a documentação localizada no código fonte, que é utilizada para gerar as especificações dos casos de uso, porém, não deve, em hipótese alguma, modificar o próprio código.

Quanto à documentação das classes mapeadas pelo Javadoc, que é utilizada por outros programadores como uma API, consiste em uma decisão estratégica da organização se a documentação deve ser escrita pelo programador ou pelo analista de requisitos. Leslie [25] recomenda que nos casos em que os requisitos são técnicos, o programador deve ter a permissão de escrever os comentários, contudo, o analista de requisitos também pode editá-los e complementá-los.

Programadores podem e devem revisar a documentação escrita pelo analista de requisitos, entretanto devem propor as alterações através de um sistema de gerenciamento de mudanças, tal como um sistema de *bug tracking*, que se ajusta perfeitamente à situação, considerando que a documentação está presente no código fonte. As mudanças aprovadas são, então, efetuadas pelo analista de requisitos.

4 ESTUDO DE CASO

Dois estudos de caso são apresentados nas duas seções deste capítulo:

- O primeiro foi aplicado ao projeto já existente do Sistema de Gerenciamento de Clínica Odontológica da Universidade Estadual de Londrina (UEL), que foi construído entre os anos de 2009 a 2011, e teve por objetivo demonstrar a aplicação das anotações GaiaDoc em classes que representam regras de negócio do projeto com concentração na redução da duplicação e maior facilidade de gestão dos requisitos;
- O segundo estudo de caso não visou sua aplicação prática, foi apenas um estudo de aplicação e impacto causado pela adoção da metodologia proposta nos projetos de uma divisão de desenvolvimento de software. A divisão escolhida foi a Divisão de Desenvolvimento de Software (DDS) da Assessoria de Tecnologia de Informação (ATI), também da UEL, com a meta de verificar quais seriam as principais dificuldades em um ambiente real e sugestões para solucioná-las e permitir a adoção da abordagem proposta.

4.1 Aplicação no Projeto de Sistema de Gerenciamento de Clínica Odontológica da UEL

O primeiro estudo de caso foi aplicado ao projeto do Sistema de Gerenciamento de Clínica Odontológica da Universidade Estadual de Londrina, desenvolvido pelo Departamento de Ciência da Computação da mesma universidade.

A equipe do projeto consistiu em quatro alunos de graduação, um aluno de pós-graduação e dois professores que atuaram na gerência do projeto. O processo adotado foi o RUP e o padrão de documentação de código escolhido foi o Javadoc. Apesar disso não foi definido nenhum padrão sobre *como* os requisitos deveriam ser escritos ou validados.

O projeto foi implementado na arquitetura JEE 6. Foram identificadas 12 classes centrais que implementaram as regras de negócio:

1. ManterAgenda;
2. ManterAgendamento;
3. ManterClínicas;
4. ManterControleAtendimento;

5. ManterDatasClinicas;
6. ManterListaEspera;
7. ManterProntuario;
8. ManterTabelaSUS;
9. ManterUBS;
10. ManterUsuario;
11. ManterGraficoAuxiliar.

Para avaliar o nível de documentação de cada uma dessas classes centrais foi estabelecida a seguinte métrica:

- Muito bem comentada (em média 3 a 5 linhas de comentário por método): 4 classes;
- Comentada (em média 1 a 2 linhas de comentário por método): 6 classes;
- Não comentada: 2 classes.

É importante destacar entretanto que a métrica adotada visa apenas identificar o quanto cada classe foi documentada e não a qualidade dos requisitos nela contidos.

Para a verificação e validação da qualidade de requisitos poderia ser escolhido um conjunto de classes por amostragem de acordo com diversos critérios tais como instabilidade e criticidade das regras de negócio ou até mesmo por aleatoriedade. Em cada classe selecionada poderiam ser aplicados métodos direcionados a qualidade de requisitos como o de [26].

De acordo com a análise das classes foi observado que quanto mais simples a regra de negócio implementada, menor era a chance da classe conter documentação descrevendo a si própria e seus métodos. Quando a complexidade aumentava, o número médio de linhas de comentário por método era proporcionalmente incrementado.

A documentação contida no código fonte foi comparada com aquela presente na ECU, buscando por sobreposição de requisitos, o que leva a um maior esforço de manutenção e maior probabilidade de inconsistência de requisitos.

Classes comentadas em mais detalhes foram aquelas que tiveram mais requisitos duplicados em relação a sua respectiva ECU. O bloco de comentário do método verificaUbs (figura 4.1) da classe ManterListaEspera exemplifica este cenário descrevendo muito bem as ações necessárias para verificar se existem vagas disponíveis para agendamento em uma Unidade Básica de Saúde (UBS). Entretanto, ele é quase uma cópia da ECU Manter Lista de Espera. No caso de qualquer alteração no procedimento de verificar vagas, o impacto da mudança atingirá no mínimo três artefatos do projeto.

```
/**
 * Método que verifica se existem vagas livres para agendamento em uma UBS. Se existirem
 * armazena o mês e o ano para posterior agendamento. Se não existirem coloca o paciente
 * em lista de espera.
 *
 * As operações que o método executa são:
 *              1) Remove da lista de UbsVagas os itens que não correspondem ao ID da UBS
 * determianda pelo prontuário do paciente e os meses anteriores a data atual;
 *              2) Para todas as clínicas que o paciente foi encaminhado:
 *                      2.1) Verifica se existem vagas livres em qualquer mes;
 *                      2.2) Verifica se na lista de espera existem pacientes para a mesma clínica na mesma UBS;
 *              3) A partir dos resultados de 2.1 e 2.2:
 *                      3.1) Se existirem pacientes na lista de espera e existirem vagas livres para
 *  a clínica então armazena o paciente na lista de espera, armazena uma mensagem para que o
 *  setor de agendamento possa posteriormente fazer o agendamento do(s) paciente(s) que estão
 *  na lista de espera;
 *              3.2) Se existirem vagas livres é feita a verificação para saber em qual mês e ano
 *  estão essas vagas, são feitas as subtrações necessárias das vagas de cada UBS e esses dados
 *  são armazenados para posterior agendamento.
 *              3.3) Se não existirem vagas livres em nenhum mês o paciente é colocado em lista de espera
 *
 */
public void verificaUbs(){
        Prontuario prontuarioTemp = new Prontuario();
        listaAgendamentoEspera = new ArrayList<ListaEspera>();
        AgendamentoClinica ac = new AgendamentoClinica();
```

Figura 4.1 – Documentação de código fonte Javadoc do método verificaUbs na classe ManterListaEspera.

Classes com poucos comentários continham pouca ou nenhuma sobreposição, porém foi identificado que em muitos casos apenas a documentação do código fonte não era o bastante para compreender a sua regra de negócio sem o exame do próprio código ou do documento de ECU.

Na sequência, os casos de uso identificados e as classes de regra de negócio do projeto foram correlacionadas. Cada caso de uso foi mapeado em apenas uma classe de regra de negócio (ou *managed bean*, no caso da arquitetura desse projeto), sendo desconsideradas classes auxiliares como entidades, validadores, conversores, entre outras, que tenham relação com a classe mapeada.

Apesar de ser possível ocorrer e de estar inclusive previsto na especificação do GaiaDoc, nenhum caso de uso do projeto abordado nesse caso de uso foi mapeado em mais de uma classe de regra de negócio.

As classes foram re-documentadas seguindo a abordagem GaiaDoc tomando o cuidado para manter toda a informação contida na ECU. Todavia, considerando que a GaiaDoc em sua atual versão não suporta a adição de imagens, os diagramas de caso de uso contidos na ECU não puderam ser incluídos na versão gerada pela ferramenta. Portanto esta foi a única informação identificada que foi perdida na re-documentação das classes.

Visando demonstrar, de fato, o funcionamento da ferramenta GaiaDoc, é apresentada a re-documentação da classe que representa a ECU Manter Agenda. O diagrama de caso de uso representado na figura 4.2 indica que o caso de uso possui apenas um ator, a secretária da clínica, que pode cadastrar tanto prontuários quanto atendimentos. Quando um atendimento é cadastrado no sistema, é requerido que o prontuário do paciente atendido já esteja no banco de dados, caso contrário, deverá ser inserido no sistema antes do cadastro do atendimento.

O caso de uso Manter Agenda possui ainda relação de inclusão com o caso de uso Validar Usuário, e relação de extensão com o caso de uso Cadastrar Paciente.

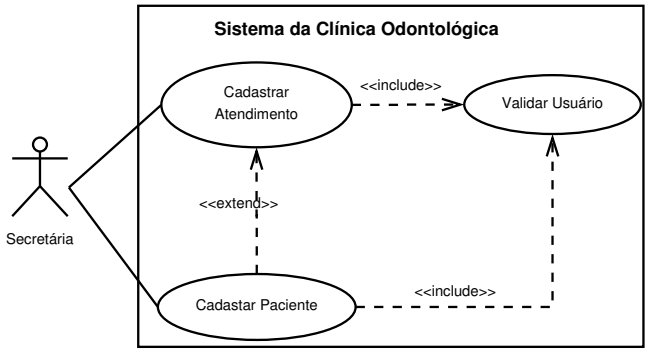


Figura 4.2 – Caso de Uso do Sistema de Clínica Odontológica.

Em qualquer operação de persistência deve ocorrer a validação do usuário para determinar se ele é do tipo secretária e, portanto, possui permissão de cadastrar pacientes e atendimentos, ou se a operação é inválida.

A figura 4.3 apresenta a parte do diagrama de classes do projeto que é relevante a este caso de uso e à sua documentação de código fonte gerada pelo GaiaDoc.

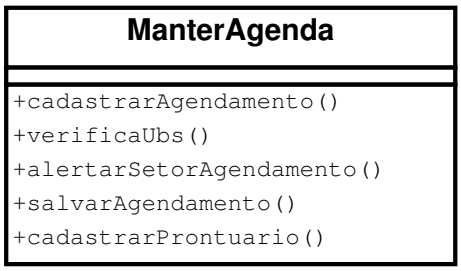


Figura 4.3 – Diagrama da classe ManterAgenda.

A figura 4.4 representa a classe ManterAgenda, na linguagem de programação Java, que contém a documentação de código fonte no formato GaiaDoc e representa o controlador na camada MVC. A camada de visão pode ser criada utilizando bibliotecas de componentes do framework JavaServer Faces (JSF) ou somente com HTML. A camada de modelo pode ser montada com classes POJO (Plain Old Java Objects) simples ou com entidades do framework JPA (Java Persistence API). Isso demonstra que o controlador fica com um baixo acoplamento em relação aos componentes das outras camadas da arquitetura.

É importante observar, em projetos desenvolvidos com a abordagem GaiaDoc desde sua concepção, que as classes contêm apenas as assinaturas dos métodos (parâmetros e tipos de retorno são permitidos), sem sua implementação. O conteúdo dos métodos é trabalhado somente na fase de construção pela equipe de desenvolvimento. No caso do sistema da Clínica Odontológica, essa implementação já existia.

Após a execução da ferramenta GaiaDoc sobre a classe ManterAgenda, é gerada como saída a especificação de caso de uso demonstrada na figura 4.5, onde podemos perceber que como benefício adicional a ferramenta ainda gera um padrão de documento de especificação de caso de uso para a organização.

Visto que a GaiaDoc foi aplicada a um projeto já desenvolvido, o estudo se preocupou apenas em verificar a eficiência da proposta relacionada à duplicação dos requisitos no código fonte e na ECU, como também na garantia de que os requisitos estejam consistentes com os artefatos do projeto.

```
/**
 * @name Manter Agenda
 * @description Este caso de uso descreve o cadastro de agendamento de pacientes
 *              na agenda de uma clínica odontológica realizado por meio da
 *              página de Prontuários.
 * @writer Humberto Ferreira da Luz Junior
 * @performer Secretária
 * @performer Administrador do sistema
 * @specialRequirement O sistema deve exibir mensagem de erro caso o cadastro
 *                     não seja completado com sucesso.
 * @preCondition O usuário deve estar autenticado no sistema e deve pertencer ao
 *               grupo de usuários secretária ou administrador.
 * @extension Se o paciente atendido pela UBS não estiver cadastrado, esse
 *            cadastro deve ser realizado antes do cadastro de agendamento.
 */
public class ManterAgenda {
        /**
         * @basicFlow(1)
         * @description A secretária seleciona o paciente e a clínicas em que ele
         *              deve ser agendado no sistema. Ela clica no botão para
         *              cadastrar o agendamento.
         */
        public void cadastrarAgendamento() {}
        /**
         * @basicFlow(2)
         * @description O sistema verifica se a UBS da secretária possui vagas
         *              livres para agendamento na clínica solicitada. Se não houver
         *              vagas o paciente é colocado em lista de espera.
         */
        public void verificaUbs() {}
        /**
         * @alternativeFlow('Pacientes que podem ser agendados estão na lista de
         *                              espera',1)
         * @description Se no passo 2 do fluxo básico é identificado que há vagas
         *              disponíveis para agendamento de determinada UBS e também há
         *              pacientes na lista de espera aguardando agendamento, é
         *              enviado um alerta ao setor de agendamento da Clínica
         *              Odontológica.
         */
        public void alertarSetorAgendamento() {}
        /**
         * @basicFlow(3)
         * @description O sistema salva o agendamento.
         */
        public void salvarAgendamento() {}
        /**
         * @alternativeFlow('Prontuário não cadastrado',1)
         * @description Se no passo 1 do fluxo básico, o prontuário do paciente não
         *              estiver disponível para o cadastro do agendamento, a
         *              secretária é direcionada ao cadastro de prontuários.
         */
        public String cadastrarProntuario() {}
```

Figura 4.4 – Documentação de código fonte da classe ManterAgenda.

O próximo passo consiste na aplicação da GaiaDoc em um projeto desde sua concepção e a validação do processo de requisitos proposto neste trabalho, como também analisar o relacionamento entre o analista de requisitos e o programador de forma prática.

Outros exemplos de classes controladoras e suas respectivas especificações de caso de uso resultantes podem ser encontradas no apêndice A deste trabalho ou no repositório SVN do projeto GaiaDoc localizado no Google Code, cujo endereço é referenciado na seção 6.

Especificação de Caso de Uso: Manter Agenda

Autor: Humberto Ferreira da Luz Junior
Descrição: Este caso de uso descreve o cadastro de atendimento de pacientes na agenda de uma clínica odontológica realizado por meio da página de Prontuários.
Ator: Secretária, administrador do sistema.
Classe: ManterAgenda

Extensão

1. Se o paciente atendido pela UBS não estiver cadastrado, esse cadastro deve ser realizado antes do cadastro de agendamento.

Requisito Especial

1. O sistema deve exibir mensagem de erro caso o cadastro não seja completado com sucesso.

Fluxo Básico

1. A secretária seleciona o paciente e as clínicas em que ele deve ser agendado no sistema. Ela clica no botão para cadastrar o agendamento.
2. O sistema verifica se a UBS da secretária possui vagas livres para o agendamento na clínica solicitada. Se não houver vagas o paciente é colocado em lista de espera.
3. O sistema salva o atendimento.[1]

Fluxos Alternativos

1. **Pacientes que podem ser agendados estão na lista de espera**
   1. Se no passo 2 do fluxo básico é identificado que há vagas disponíveis para agendamento de determinada UBS e também há pacientes na lista de espera aguardando agendamento, é enviado um alerta ao setor de agendamento da Clínica Odontológica.
2. **Prontuário não cadastrado**
   1. Se no passo 1 do fluxo básico, o paciente não estiver disponível para cadastrar o agendamento, a secretária é direcionada ao cadastro de pacientes.

1 **Pré-Condição:** O usuário deve estar autenticado no sistema e deve pertencer ao grupo de usuários secretária ou administrador.

Figura 4.5 – Especificação de caso de uso Manter Agenda.

4.2 Aplicação na Assessoria de Tecnologia de Informação da UEL

A ATI da UEL tem sua organização administrativa dividida em três diretorias:

- Diretoria de Desenvolvimento de Sistemas (DDS): é responsável por projetar, desenvolver, implantar e manter os sistemas de informação da instituição;
- Diretoria de Suporte ao Usuário (DSU): fornece apoio à comunidade universitária na utilização de serviços de e-mail, acesso à internet, sistemas de informação e atendimento técnico nos computadores ligados à rede;
- Diretoria de Suporte a Redes e Sistemas (DSRS): tem por função a configuração, manutenção e implantação da infraestrutura de redes, servidores de aplicação, banco de dados, sistemas operacionais e softwares de apoio aos sistemas de informação da instituição.

A DDS é composta por 19 analistas, separados por 5 divisões que são focadas em domínios específicos de requisitos:

- Divisão de Sistemas Acadêmicos (DSAC);
- Divisão de Sistemas Administrativos (DSAD);
- Divisão de Sistemas Financeiros (DSF);
- Divisão de Sistemas de Recursos Humanos (DSRH);
- Divisão de Sistemas Web (DSW).

Nas divisões DSAD e DSF ocorre predomínio de sistemas implementados por meio da solução proprietária Oracle Forms & Reports (OFR) e os sistemas Web são implementados na linguagem de programação PHP. Nas divisões DSAC e DSRH também há predomínio do OFR, com a maior parte dos sistemas Web em PHP, porém já há alguns sistemas implementados e migrados para a plataforma JEE. Na divisão DSW ocorre predomínio do PHP e alguns dos sistemas são implementados com a plataforma JEE.

Os programas escritos em OFR seguem o paradigma estruturado e orientado a eventos. Seu código fonte é armazenado em um arquivo binário, com a extensão *.fmb* para Forms e *.rdf* para Reports, que não permite sua leitura como arquivo de texto, assim como não permite que a ferramenta de versionamento realize a comparação entre 2 versões de uma mesma fonte.

Os programas escritos em PHP seguem o paradigma orientado a objetos na DSW e o paradigma estruturado nas outras divisões da DDS, enquanto as aplicações Java são todas implementadas com orientação a objetos e desenvolvidas por meio da plataforma JEE.

Nenhum PDS é adotado e o padrão oficial de documentação de requisitos da DDS consiste apenas na escrita de atas, que são validadas e assinadas pelos membros do projeto que participaram da reunião. Algumas divisões adotam meios alternativos de documentação, sem qualquer forma de padronização, como *wikis*, ferramentas de gerenciamento de processos ágeis e documentos de texto.

As ferramentas de apoio utilizadas pela DDS consistem em: (i) Subversion, um sistema de gerenciamento de versões, que é aplicado ao código fonte e a outros artefatos dos projetos; ferramenta solicitação de serviços desenvolvida pela DDS e utilizada também pela DSU; (iii) Mantis, uma ferramenta de rastreamento de mudanças que é utilizada somente para solicitação de serviços à DSRS; e (iv) OTRS, que está sendo adotado progressivamente com o intuito de unificar as solicitações de serviços para 3 diretorias da ATI em uma única ferramenta.

Para melhor compreender os problemas e necessidades da DDS com relação à engenharia de requisitos foi elaborado um questionário, respondido por 13 analistas. [1].

A pesquisa indicou que 52% dos analistas utilizam apenas atas de reunião para documentar os requisitos e também 52% consideram a ata de reunião como uma forma de comprometimento de requisitos

[1] O apêndice B disponibiliza os resultados completos da pesquisa, realizada com os analistas da DDS na ATI

acordados com o usuário. Apenas 4% vê a ata de reunião como um documento que deve ser utilizado para validação.

Essa parte da pesquisa indica que a maior parte dos analistas utiliza apenas a ata para documentação de requisitos e que, ainda que seja o único documento de requisitos, ele não é utilizado para validação do produto entregue.

Quanto à resposta a mudanças, 77% dos analistas afirmou que os requisitos não são todos atualizados e se tornam inconsistentes no decorrer do projeto.

De acordo com as respostas dos analistas, 54% não versionam seus documentos de requisitos, entretanto 85% afirmam que as mudanças são registradas por meio de documentos de texto, planilhas ou sistema de solicitação de serviço.

Os documentos de requisitos são disponibilizados aos stakeholders do projeto por meio de e-mail ou compartilhamento de pasta, mas em alguns casos também por *wiki*, ata impressa ou conversa pessoal.

Mais da metade dos analistas consideram que ocorre duplicação de requisitos entre os documentos do projeto e 85% não estão satisfeitos com os procedimentos adotados atualmente para a documentação de requisitos.

Entre os problemas alegados estão:

- Falta de padronização na documentação de requisitos;
- Falta de atualização dos documentos;
- Falta de metodologia para gerenciamento dos requisitos;
- Dificuldade para sintetizar as informações das atas em dados relevantes aos gerentes de projeto, diretores e reitores;
- Falta de homologação;
- Falta de ferramenta que auxilie na gestão dos requisitos;

Como consequência da situação crítica da gestão dos requisitos, é comum que na ocorrência de problemas em um sistema, caso o analista que possui domínio das regras de negócio esteja ausente, haja grandes dificuldades para resolvê-los.

Ocorreu, nos anos de 2006 e 2007, a definição de um fluxo para gestão de projetos e de requisitos, baseado no RUP, com a utilização de ECUs, protótipos de interface de usuário e atas de reunião, que desempenham o papel do documento de visão e já eram utilizados para obter comprometimento com o cliente.

A iniciativa não seguiu adiante e, segundo os analistas, as principais razões para o fracasso na fase de implantação do processo especificado foram:

- Falta de comprometimento;
- Falta de comunicação e tentativas isoladas;
- Super-estimativa do tempo e esforço que seriam gastos na criação e manutenção de todos os documentos;
- Preconceito quanto ao retorno prático da adoção do RUP nas atividades e tarefas da ATI;
- Documentação excessiva, falta de análise e discussão interna;
- Falta de tempo;
- Descontinuidade da implementação, pois deveria ter sido um processo de melhoria contínua;
- Falta de organização.

De acordo com o cenário descrito e levando em consideração que não existe um fluxo de requisitos definido até o momento, o fluxo especificado neste trabalho pode ser utilizado como base para a definição de um processo que atenda as necessidades particularidades da ATI.

As razões apontadas para a não adoção do RUP devem ser utilizadas como lições aprendidas para a implantação de um novo processo. A falta de comunicação poderia ser solucionada pela adoção de uma ferramenta de gestão de projetos, que facilitaria o acompanhamento do processo de definição do processo de desenvolvimento, além de ampla divulgação por meio dos canais de comunicação disponíveis na organização.

A falta de comprometimento deve ser tratada por meio de reuniões e incentivos para a participação de todos os analistas na elaboração do processo.

Para evitar a desmotivação e descontinuidade da implantação do processo, devem ser estabelecidos objetivos razoáveis e graduais, seguindo um modelo similar aos níveis do CMMI.

O GaiaDoc, nesse contexto, soluciona os problemas de duplicação e versionamento dos requisitos, atenua a inconsistência entre os artefatos e, de certo modo, define um padrão de documentação de código fonte e um formato, também padronizado, de ECU.

A adoção do GaiaDoc, entretanto, é possível apenas em uma pequena porção dos projetos (Java e PHP, ainda se adaptado, no caso da segunda linguagem), não podendo ser utilizado com o OFR, que segue o paradigma estruturado de programação e, além disso, armazena o código fonte em arquivos binários, o que impede que as anotações sejam lidas pelo GaiaDoc.

É estimado pelos gestores e analistas da ATI que sejam necessários mais de 10 anos para a migração de todo o sistema em OFR para a arquitetura JEE, o que inviabiliza a adoção do GaiaDoc nesse ambiente, como padrão oficial de documentação de requisitos de forma mais abrangente.

Este estudo de caso deixa evidente a dificuldade na adoção do GaiaDoc como padrão de documentação de regra de negócio em organizações de desenvolvimento de software que possuem sistemas legados desenvolvidos por meio do paradigma estruturado de programação.

Organizações que possuem a maior parte de seus projetos implementados em linguagens de programação orientadas a objetos não devem enfrentar grandes dificuldades para a utilização da metodologia

proposta, além da necessidade de adotar um processo de desenvolvimento compatível com as exigências da ferramenta GaiaDoc.

5 CONSIDERAÇÕES FINAIS

Neste trabalho, foi proposta uma nova perspectiva no fluxo de requisitos do RUP por meio da utilização da ferramenta GaiaDoc. A ferramenta tem por objetivo gerar a ECU de forma automatizada à partir de anotações presentes nos blocos de comentário do código fonte.

Segundo a opinião de especialistas, membros avaliadores de conferências para as quais este trabalho foi submetido, e por meio de observações nos estudo de caso aplicados ao sistema de Clínica Odontológica e à DDS da ATI, ambos pertencentes à Universidade Estadual de Londrina, foi constatado que a metodologia e o fluxo de requisitos desenvolvidos para a ferramenta GaiaDoc são relevantes dentro da área de engenharia de requisitos.

Ainda que alguns itens como a utilização de ferramentas de versionamento e sistemas de gerenciamento de mudanças tenham sido abordados de forma superficial nos estudos de caso apresentados e possam ser melhor explorados futuramente, as seguintes contribuições foram identificadas como proporcionadas pela ferramenta GaiaDoc:

- Preenchimento da lacuna nos métodos de documentação de código utilizados atualmente para classes que representam a regra de negócio;
- Estabelecimento de um ambiente com regras mais claras para DCF, por meio da ECU, que orienta o modo em que a documentação deve ser escrita;
- Menor esforço na manutenção da documentação dos requisitos funcionais voltados à regra de negócio que é resultado da diminuição de duplicação de requisitos nos artefatos do projeto;
- Melhor controle nas mudanças aplicadas a estes requisitos provido pelo aproveitamento de técnicas já adotadas no código fonte como versionamento e sistemas de gerenciamento de mudanças;
- Maior facilidade para rastrear as mudanças dos requisitos e associá-los aos outros artefatos como o diagrama de classes e o código fonte;
- Maior consistência e integridade entre os artefatos que documentam os requisitos e a implementação;
- E uma maior padronização nos documentos de especificação de caso de uso fornecida pela geração automática da GaiaDoc.

Todas as contribuições mencionadas estão relacionadas ao controle e gerenciamento dos requisitos e não à sua qualidade. Outros métodos e técnicas devem ser utilizados para esse propósito.

As recomendações na DCF usualmente se limitam à escrita de um código auto-explicativo e inclusão de comentários apenas quando necessários, não estabelecendo regras ou padrões sobre a forma que ela deve ocorrer. Este trabalho aproveita um padrão de escrita de requisitos altamente consolidado e amplamente utilizado e por esse motivo é também esperado que a DCF por meio do GaiaDoc exerça maior atratividade ao desenvolvedor de software.

A análise de conformidade do fluxo de requisitos proposto em relação às áreas de processo do CMMI indicou que as alterações realizadas não causaram impacto no desenvolvimento de requisitos e introduziu soluções práticas em alguns aspectos da gestão de mudanças e de rastreabilidade de requisitos trazidas pela melhor integração da documentação dos requisitos com práticas já adotadas na gestão de código fonte.

O estudo de caso da aplicação do GaiaDoc na ATI evidenciou as dificuldades encontradas na adoção de PDSs em equipes de desenvolvimento de software, além de obstáculos impostos por ferramentas e tecnologias já adotadas, que seguem outros paradigmas de programação.

O objetivo deste trabalho não foi propor uma ferramenta que substitua as tradicionais voltadas à documentação de APIs, mas disponibilizar uma alternativa que pode se adequar melhor às necessidades do projeto em determinados casos.

No caso da linguagem Java, o ideal seria utilizar ambas as ferramentas (Javadoc e GaiaDoc) no projeto, adotando uma em detrimento da outra (ou a combinação de ambas), de acordo com o objetivo da classe implementada.

5.1 Trabalhos Futuros

Atualmente, apenas classes da linguagem Java são suportadas pela ferramenta GaiaDoc, contudo, estuda-se o suporte a outras linguagens orientadas a objeto de forma extensível. O objetivo consiste em possibilitar, pelo simples intermédio da configuração de um arquivo com as características sintáticas da linguagem em questão, adicionar seu suporte à ferramenta.

A API GaiaDoc pode ser também estendida para a criação de plugins para IDEs amplamente utilizadas como, por exemplo, o Netbeans e/ou o Eclipse.

Do ponto de vista do fluxo de requisitos pode ser realizado um estudo para a adaptação do GaiaDoc para outros PDSs além do RUP, tanto para os ágeis quanto para os prescritivos, adotando a ECU e o diagrama de classes como artefatos para os requisitos ou estabelecendo um novo conjunto de anotações para o mapeamento de artefatos de requisitos no código fonte.

O GaiaDoc pode ser aplicado a um projeto desde sua concepção até sua conclusão, de modo que o fluxo de requisitos especificado neste trabalho seja validado.

É possível também aprofundar o modo em sistemas de *bug tracking* podem efetivamente auxiliar na gestão dos requisitos do projeto. De que forma eles atendem essa necessidade e que aprimoramentos poderiam ser desenvolvidos para melhorar seu suporte.

REFERÊNCIAS

[1] KRUCHTEN, P. *The Rational Unified Process: An Introduction*. 3. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN 0321197704.

[2] KINIRY, J. R.; FAIRMICHAEL, F. Ensuring consistency between designs, documentation, formal specifications, and implementations. In: *Proceedings of the 12th International Symposium on Component-Based Software Engineering*. [S.l.: s.n.], 2009. (CBSE '09), p. 242–261. ISBN 978-3-642-02413-9.

[3] MICROSYSTEMS, S.; ORACLE. *Javadoc*. 1995–2013. `http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html`.

[4] MICROSOFT. *Sandcastle*. 2008–2013. `http://sandcastle.codeplex.com/`.

[5] HEESCH, D. van. *Doxygen*. 1997–2013. `http://www.doxygen.org/`.

[6] FOUNDATION, A. S. *Subversion*. 2004–2013. `http://subversion.apache.org/`.

[7] TORVALDS, L. *Git*. 2005–2013. `http://git-scm.com/`.

[8] GOES, A.; BARROS, R. Gerenciamento do conhecimento em uma fabrica de software: Um estudo de caso aplicando a ferramenta gaia - l.a. In: *Proceedings of the XXXVIII Conferencia Latinoamericana en Informatica*. Medellin, Colombia: [s.n.], 2012. (CLEI).

[9] WRIGHT, S. Requirements traceability - what? why? and how? In: *In Proceedings of the Colloquium by the Institution of Electrical Engineers Professional Group C1 (Software Engineering)*. London, UK: [s.n.], 1991. p. 1–2.

[10] RAMOS, R. et al. Um padrão para requisitos duplicados. In: *V Latin American Conference on Pattern Languages of Programming*. Porto de Galinhas, PE, Brazil: SugarLoafPLop, 2007. (SugarLoafPLop '07).

[11] SPINELLIS, D. Code documentation. *IEEE Software*, v. 27, n. 4, p. 18–19, 2010.

[12] SCHRECK, D.; DALLMEIER, V.; ZIMMERMANN, T. How documentation evolves over time. In: *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*. New York, NY, USA: ACM, 2007. (IWPSE '07), p. 4–10. ISBN 978-1-59593-722-3. Disponível em: <http://doi.acm.org/10.1145/1294948.1294952>.

[13] DEPASQUALE, P. et al. // todo: Help students improve commenting practices. In: *Frontiers in Education Conference (FIE), 2012*. [S.l.: s.n.], 2012. p. 1–6. ISSN 0190-5848.

[14] HEIJSTEK, W.; CHAUDRON, M. R. V. Evaluating rup software development processes through visualization of effort distribution. In: *Software Engineering and Advanced Applications, 2008. SEAA '08. 34th Euromicro Conference*. [S.l.: s.n.], Sept. p. 266–273. ISSN 1089-6503.

[15] PRESSMAN, R. *Software Engineering: A Practitioner's Approach*. 7. ed. New York, NY, USA: McGraw-Hill, Inc., 2010. ISBN 0073375977, 9780073375977.

[16] BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. *Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series)*. [S.l.]: Addison-Wesley Professional, 2005. ISBN 0321267974.

[17] JACYNTHO, M. D.; SCHWABE, D.; ROSSI, G. A software architecture for structuring complex web applications. *J. Web Eng.*, Rinton Press, Incorporated, Paramus, NJ, v. 1, n. 1, p. 37–60, out. 2002. ISSN 1540-9589. Disponível em: <http://dl.acm.org/citation.cfm?id=2011098.2011104>.

[18] JENDROCK, E. et al. *The Java EE 6 Tutorial: Basic Concepts*. 4th. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2010. ISBN 0137081855, 9780137081851.

[19] DEVELOPMENT, C. Cmmi for development, version 1.3 cmmi-dev, v1.3. *Engineering*, Carnegie Mellon University, n. November, p. 482, 2010. Disponível em: <http://www.sei.cmu.edu/library/abstracts/reports/10tr033.cfm>.

[20] KRAMER, D. API documentation from source code comments: a case study of Javadoc. In: *Proceedings of the 17th annual international conference on Computer documentation*. New York, NY, USA: ACM, 1999. (SIGDOC '99), p. 147–153. ISBN 1-58113-072-4. Disponível em: <http://doi.acm.org/10.1145/318372.318577>.

[21] LAMSWEERDE, A. van. *Requirements Engineering - From System Goals to UML Models to Software Specifications*. [S.l.]: Wiley, 2009. ISBN 978-0-470-01270-3.

[22] MANZONI, L.; PRICE, R. Identifying extensions required by rup (rational unified process) to comply with cmm (capability maturity model) levels 2 and 3. *Software Engineering, IEEE Transactions on*, v. 29, n. 2, p. 181–192, Feb. ISSN 0098-5589.

[23] SMITH, J. *Reaching CMM Levels 2 and 3 with the Rational Unified Process*. Cupertino, CA: Whitepaper, 2000.

[24] CINTRA, C. C.; PRICE, R. T. Experimenting a requirements engineering process based on rational unified process (rup) reaching capability maturity model integration (cmmi) maturity level 3 and considering the use of agile methods practices. In: ALENCAR, F. M. R.; SáNCHEZ, J.; WERNECK, V. (Ed.). *WER*. [S.l.: s.n.], 2006. p. 153–159.

[25] LESLIE, D. M. Using Javadoc and XML to produce API reference documentation. In: *Proceedings of the 20th annual international conference on Computer documentation*. New York, NY, USA: ACM, 2002. (SIGDOC '02), p. 104–109. ISBN 1-58113-543-2. Disponível em: <http://doi.acm.org/10.1145/584955.584971>.

[26] DORIGAN, J. *Um Modelo de Processo de Engenharia de Requisitos para Padronização e Aumento da Qualidade*. Dissertação (Mestrado) — Departamento de Computação, UEL, 2013.

APÊNDICES

APÊNDICE A - EXEMPLOS DE DOCUMENTAÇÃO GAIADOC

Este apêndice apresenta dois exemplos de classes documentadas no código fonte seguindo o padrão de documentação GaiaDoc e a ECU resultante.

```
/**
 * @name Receber foto
 * @description Este caso de uso descreve o recebimento de uma foto que será
 *              utilizada na confecção da carteirinha estudantil.
 * @writer Humberto Ferreira da Luz Junior
 * @performer Estudante
 * @performer Atendente da Pró-Reitoria de Graduação (PROGRAD)
 * @preCondition O usuário deve ter sido redirecionado à partir do portal do
 *               estudante de graduação, seu IP deve ser o mesmo e o tempo
 *               entre o redirecionamento e o carregamento da página deve
 *               ser inferior a 1 minuto.
 * @preCondition O número de matrícula do estudante deve ser válido e ativo.
 */
public class FotoController {
  /**
   * @basicFlow(1)
   * @description Envio e carregamento da foto por arquivo.
   */
  public void carregarArquivo(FileUploadEvent event)
      throws FileNotFoundException, IOException {}
  /**
   * @alternativeFlow('Envio por Webcam', 1)
   * @description No passo 1 do fluxo básico, a foto pode ser enviada por
   *              Webcam, ao invés de por arquivo, e carregada no sistema.
   */
  public void oncapture(CaptureEvent captureEvent) {}
  /**
   * @basicFlow(2)
   * @description Recorte da foto 3x4 à partir da foto enviada.
   */
  public String recortar() {}
  /**
   * @basicFlow(3)
   * @description Aceitação dos termos e confirmação do envio da foto 3x4.
   */
  public String confirmar() {
  }
}
```

Figura A1 – Documentação da classe FotoController do projeto Identificação Institucional da ATI no padrão GaiaDoc.

Especificação de Caso de Uso: Receber foto

Autor: Humberto Ferreira da Luz Junior
Descrição: Este caso de uso descreve o recebimento de uma foto que será utilizada na confecção da carteirinha estudantil.
Ator: Estudante, atendente da Pró-Reitoria de Graduação (PROGRAD).
Classe: FotoController

Pré-Condição

1. O usuário deve ter sido redirecionado à partir do portal do estudante de graduação, seu IP deve ser o mesmo e o tempo entre o redirecionamento e o carregamento da página deve ser inferior a 1 minuto.
2. O número de matrícula do estudante deve ser válido e ativo.

Fluxo Básico

1. Envio e carregamento da foto por arquivo.
2. Recorte da foto 3x4 à partir da foto enviada.
3. Aceitação dos termos e confirmação do envio da foto 3x4.

Fluxos Alternativos

1. **Envio por Webcam**
   1. No passo 1 do fluxo básico, a foto pode ser enviada por Webcam, ao invés de por arquivo, e carregada no sistema.

Figura A2 – Especificação de caso de uso Receber foto.

```java
/**
 * @name Renovar matrícula
 * @description Este caso de uso descreve a renovação de matrícula de um
 *              estudante da universidade.
 * @writer Humberto Ferreira da Luz Junior
 * @performer Estudante
 */
public class RenovacaoMatriculaController implements Serializable {
  /**
   * Caso o aluno esteja em sua última tentativa antes de jubilar, indica a
   * data provável do júbilo.
   */
  private CalendarioEscolar dataProvavelJubilando;
  /**
   * Data que indica o período em que o comprovante de renovação pode ser
   * emitido.
   */
  private CalendarioEscolar periodoComprovanteRenovacao;
  /**
   * @basicFlow(1)
   * @description Os dados do aluno, incluindo habilitações em curso e
   *              disciplinas optativas, são carregados pelo sistema.
   */
  public void carregarRenovacao() {}
  /**
   * @basicFlow(2)
   * @description O estudante seleciona as habilitações para ser cursadas em
   *              continuidade ou concomitância, se disponíveis.
   */
  public void adicionarHabilitacao(AgRenovacaoHabilitacao h) {}
  /**
   * @basicFlow(3)
   * @description O estudante seleciona as disciplinas optativas, se
   *              disponíveis.
   */
  public void adicionarOptativa(GradeCurricular gc) {}
  /**
   * @basicFlow(4)
   * @description O estudante renova sua matrícula com as habilitações e
   *              disciplinas optativas escolhidas.
   */
  public void salvarRenovacao() {}
  /**
   * @alternativeFlow('Remoção de habilitação', 1)
   * @description O estudante remove uma habilitação de sua lista de
   *              habilitações selecionadas.
   */
  public void removerHabilitacao(AgRenovacaoHabilitacao h) {}
  /**
   * @alternativeFlow('Remoção de disciplina optativa', 1)
   * @description O estudante remove uma disciplina optativa de sua lista de
   *              optativas selecionadas.
   */
  public void removerOptativa(GradeCurricular gc) {}
}
```

Figura A3 – Documentação da classe RenovacaoMatriculaController do projeto Portal do Estudante da ATI no padrão GaiaDoc.

Especificação de Caso de Uso: Renovar matrícula

Autor: Humberto Ferreira da Luz Junior
Descrição: Este caso de uso descreve a renovação de matrícula de um estudante da universidade.
Ator: Estudante.
Classe: RenovacaoMatriculaController

Fluxo Básico

1. Os dados do aluno, incluindo habilitações em curso e disciplinas optativas, são carregados pelo sistema.
2. O estudante seleciona as habilitações para ser cursadas em continuidade ou concomitância, se disponíveis.
3. O estudante seleciona as disciplinas optativas, se disponíveis.
4. O estudante renova sua matrícula com as habilitações e disciplinas optativas escolhidas.

Fluxos Alternativos

1. **Remoção de habilitação**
   1. O estudante remove uma habilitação de sua lista de habilitações selecionadas.
2. **Remoção de disciplina optativa**
   1. O estudante remove uma disciplina optativa de sua lista de optativas selecionadas.

Glossário

Data Provavel Jubilando	Caso o aluno esteja em sua última tentativa antes de jubilar, indica a data provável de júbilo.
Periodo Comprovante Renovacao	Data que indica o período em que o comprovante de renovação pode ser emitido.

Figura A4 – Especificação de caso de uso Renovar matrícula.

APÊNDICE B - RESULTADOS DA PESQUISA NA DDS/ATI

Resultado gráfico da pesquisa realizada com os analistas da DDS/ATI gerado por meio do Google Forms.

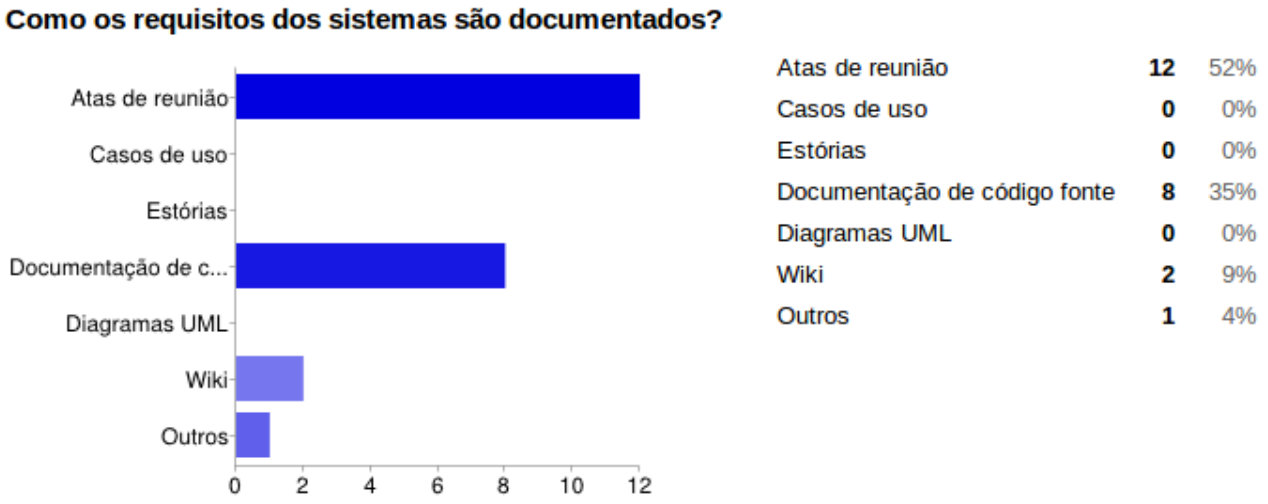
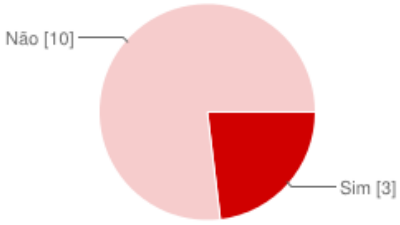


Figura B1 – Como os requisitos são documentados.

Figura B2 – O que a ata de reunião representa para os analistas.

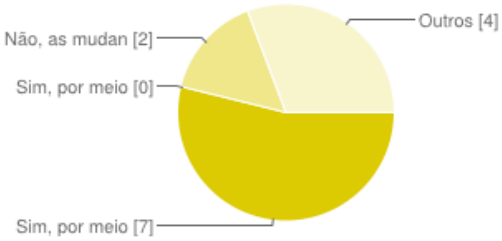
Quando ocorrem mudanças nos requisitos do sistema, a documentação de requisitos é sempre atualizada e permanece consistente com o que está implementado no código fonte?



Sim	**3**	23%
Não	**10**	77%

Figura B3 – Atualização e consistência dos documentos de requisitos.

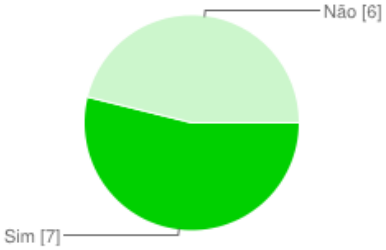
As mudanças ocorridas nos requisitos são registradas de algum modo?



Sim, por meio de documentos de texto, planilhas ou outro tipo de documento	**7**	54%
Sim, por meio de ferramentas como Mantis, Redmine, Bugizilla e outras ferramentas de rastreamento de mudanças	**0**	0%
Não, as mudanças nos requisitos não são registradas	**2**	15%
Outros	**4**	31%

Figura B4 – Registro das mudanças.

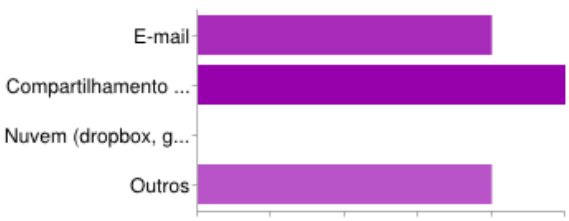
Os documentos de requisitos são versionados (i.e. gerenciados por uma ferramenta de versionamento, como o SVN)?



Sim	**7**	54%
Não	**6**	46%

Figura B5 – Versionamento dos requisitos.

De que forma os documentos de requisitos são disponibilizados para todas as pessoas do projeto (analistas, diretores e usuários)?



E-mail	**4**	31%
Compartilhamento de pasta	**5**	38%
Nuvem (dropbox, google drive, skydrive, etc)	**0**	0%
Outros	**4**	31%

Figura B6 – Disponibilização dos requisitos aos *stakeholders* do projeto.

Ocorre duplicação entre no conteúdo dos diversos documentos de requisitos? Por exemplo, um mesmo requisito documentado tanto no código fonte quanto na wiki ou na ata, de modo que qualquer mudança no requisito exija que ambos sejam atualizados para que a consistência entre os documentos de requisitos seja mantida.

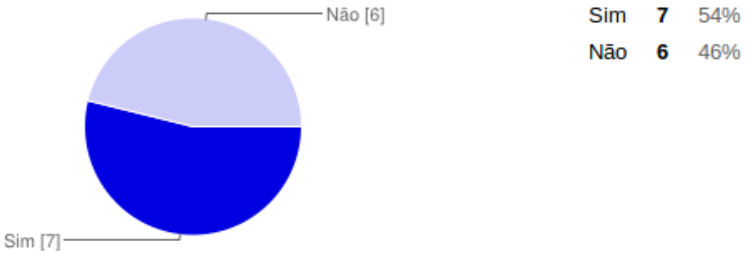


Figura B7 – Duplicação de requisitos.

Você está satisfeito com os procedimentos adotados atualmente para a documentação dos requisitos?

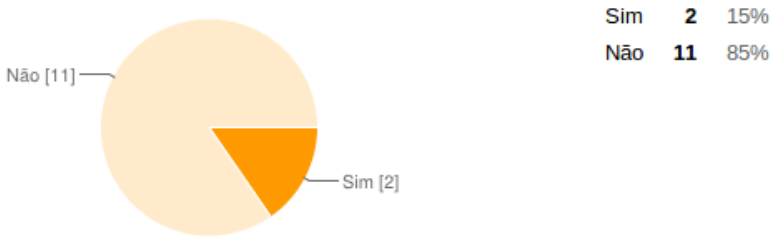


Figura B8 – Satisfação com os procedimentos adotados para a documentação dos requisitos.

Se não está satisfeito, quais são os principais problemas decorrentes do método de documentação de requisitos adotado atualmente?

Falta uma padronização de documento de requisitos | falta de atualização, falta de uma metodologia padrão para documentação | Ausência de Documento Formal | Principal problema é sintetizar as informações das atas em dados relevantes aos gerentes de projeto, diretores e reitores. Falta a padronização de medidas para planejamento dos projetos, no que se refere a recursos de tempo e pessoal. | falta de centralizacao, atualização, consistencia, rastremento de requisitos e de de classes, funcoes e bd | Falta de padronização e inexistência de homologação | Falta padronização | Precisa adotar um método simples, centralizado em uma ferramenta de gestão de projetos de fácil acesso, que armazene as informações de maneira prática para recuperação e atualização. | Padronização | não tem

Figura B9 – Satisfação com os procedimentos adotados para a documentação dos requisitos.

Houve uma tentativa de adoção de um processo de desenvolvimento de software na DDS alguns anos atrás. Na sua opinião, quais os fatores que impediram a adoção do RUP com sucesso?

falhas de comunicação e tentativas isoladas, sem a participação de todos os envolvidos no processo | Excesso de Controles | Comprometimento de todos os chefes e diretores. Superestimar o tempo e esforço que se gastaria na criação e manutenção de todos os documentos. Preconceito quanto o retorno prático da adoção do RUP nas atividades e tarefas da ATI. Subestimar a validade da implantação de um processo de desenvolvimento ao longo do tempo. | Agora deixou de ser anonimoauahuahuahu. eu nao estava aqui na época. | Não estava na época, mas creio que faltou apoio e maior cobrança por parte dos chefes e diretores, devido isso tomar tempo e influenciar o andamento do trabalho; | Documentação excessiva e falta de análise e discussão interna. | Não atende a realidade do setor. | Falta de tempo para implantação de novas metodologias, pois , foi feito um estudo de metodologias e ferramentes a serem adotadas, e isso deveria ser um processo contínuo, infelizmente na UEL, a cada troca de gestão, alguns estudos são suspensos, não tem continuidade. | Faltou tempo da equipe se organizar e adotar o procedimento viável para mudança. | Falta de tempo, e com o aumento de sistemas e falta de analistas no desenvolvimento. E tambem muita documentação.

Figura B10 – Satisfação com os procedimentos adotados para a documentação dos requisitos.

TRABALHOS PUBLICADOS PELO AUTOR

1. LUZ JUNIOR, H.F.; BARROS, R.M.; Source Code Documentation: a Tool Focused on Business Requirements. Proceedings of the IADIS International Conference Applied Computing, 2012, Madrid, Espanha. (IADIS'12), 2012, p. 267-274;

2. LUZ JUNIOR, H.F.; BARROS, R.M.; GaiaDoc: a Tool Focused on Business Requirements for Code Documentation. Proceedings of the 25th International Conference on Computer Applications in Industry and Engineering, 2012, New Orleans, EUA. (CAINE'12), 2012, p. 63-68;

3. LUZ JUNIOR, H.F.; BARROS, R.M.; GaiaDoc: a Tool Focused on Business Requirements for Code Documentation with a CMMI Compliant and RUP Based Requirement Flow. Journal of Comunication and Computer, 2013, Chicago, EUA. (JCC'13).