UNIVERSIDADE FEDERAL DE PELOTAS

Bacharelado em Ciência da Computação

Trabalho acadêmico

Modelagem de um processo de desenvolvimento de software baseado nas técnicas e metodologias de projetos open source

Lucas Diego Zwirtes Cavalheiro

Pelotas, 2007

LUCAS DIEGO ZWIRTES CAVALHEIRO

MODELAGEM DE UM PROCESSO DE DESENVOLVIMENTO DE SOFTWARE BASEADO NAS TÉCNICAS E METODOLOGIAS DE PROJETOS OPEN SOURCE

Trabalho acadêmico apresentado ao
Curso de Ciência da Computação da
Universidade Federal de Pelotas, como
requisito parcial à obtenção do título de
Bacharel em Ciência da Computação

.....
Orientadora: Profª. MSc. Flávia Braga de Azambuja

Pelotas, 2007

Dados de catalogação na fonte:
Ubirajara Buddin Cruz – CRB-10/901
Biblioteca de Ciência & Tecnologia - UFPel

C376m Cavalheiro, Lucas Diego Zwirtes

Modelagem de um processo de desenvolvimento de software baseado nas técnicas e metodologias de projetos open source / Lucas Diego Zwirtes Cavalheiro ; orientador Flávia Braga de Azambuja. – Pelotas, 2007. – 112f. : il. color. - Monografia (Conclusão de curso). Curso de Bacharelado em Ciência da Computação. Departamento de Informática. Instituto de Física e Matemática. Universidade Federal de Pelotas. Pelotas, 2007.

1.Informática. 2.Engenharia de software. 3.Desenvolvimento de software. 4.Processo de software. 5.Modelagem de processo de software. 6.Software open source. 7.Software livre. I.Azambuja, Flávia Braga de. II.Título.

CDD: 005.1

Banca examinadora:

.....

Profª. Flávia Braga de Azambuja, MSc. (Orientadora)

.....

Prof. Dr. Carlos Antônio Pereira Campani

.....

Prof. Guilherme Tomaschewski Netto, MSc.

.....

Prof. Dr. Lucas Ferrari de Oliveira

À minha família, por todo amor recebido nesses vinte e seis anos.

Agradecimentos

Ao Pai Celestial, Deus, Javeh, GADU, Buda, Alá, Jah ou Olorum (vai saber quem realmente está lá em cima) pela Criação.

Aos meus pais que me deram amor, estudo e toda dedicação que um filho poderia querer ter durante todos esses anos, e sempre me ensinaram que não existe bem material que supere o valor do conhecimento.

Aos meus avós, pelo carinho e apoio durante os primeiros anos de vida e durante minha estada em Pelotas.

A minha irmã, por sempre incomodar me mandando ir estudar e não jogar no computador.

Aos meus tios e tias, as puxadas de orelha valeram a pena.

Aos meus primos, nossas “artes” também valeram a pena.

À minha orientadora Flávia Braga de Azambuja, pela sua dedicação ao aluno.

A todos os professores, de alguma forma vocês são responsáveis pelo o que eu sou hoje e pelo que serei amanhã.

À Raquel, pelos incentivos constantes nas nossas conversas além das consultorias gramaticais.

À Nescafé e ao Guaraná do Amazonas, sem os quais seria impossível atravessar as gélidas madrugadas pelotenses para concluir este trabalho.

À embaixada russa.

A todos os meus amigos de Ijuí, Santa Maria e Porto Alegre pelos agitos nas férias.

Aos meus amigos do condado, que quando eu precisava me distrair me levavam pra festa, e quando eu precisava estudar também me levavam pra festa.

A todos meus colegas, em especial Helen Franck, Leandro Carvalho, Mario Heber Lopez Filho e Roberta F. Correa.

E a todos aqueles que me esqueci de citar aqui, obrigado!

“*I do not fear computers. I fear the lack of them.*”

Isaac Asimov

Resumo

CAVALHEIRO, Lucas Diego Zwirtes. **MODELAGEM DE UM PROCESSO DE DESENVOLVIMENTO DE SOFTWARE BASEADO NAS TÉCNICAS E METODOLOGIAS DE PROJETOS OPEN SOURCE**. 2007. 111fl. Trabalho acadêmico (Graduação) – Instituto de Física e Matemática. Universidade Federal de Pelotas.

A adoção de um processo de desenvolvimento de software é vital para o bom andamento do projeto e a garantia de que o sistema será entregue funcionando e dentro do tempo estipulado no início do projeto. Desde a "crise" do software diversas metodologias foram criadas sempre buscando adaptar o processo de desenvolvimento a novas tecnologias e modelos de negócio. Comunidades de desenvolvimento *open source* há tempos produzem software de boa qualidade e com uma quantidade de falhas admissível. Cada projeto open source possui técnicas e métodos particulares, porém compartilham características como a distribuição geográfica dos desenvolvedores e a intensa comunicação entre os membros da equipe de desenvolvimento e a comunidade de usuários. Grandes projetos como GNU/Linux, Apache e Mozilla são alguns exemplos que impulsionaram o interesse acadêmico e comercial no processo de desenvolvimento de software *open source*. No entanto existem lacunas e características muito específicas no seu processo de desenvolvimento que impedem que essa metodologia seja adotada completamente por outras equipes de desenvolvimento. Todavia, essas lacunas proporcionam oportunidades para se criar processos híbridos de desenvolvimento. Este trabalho parte da análise qualitativa de processos de desenvolvimento de projetos open source, visando formalizar um modelo de processo de desenvolvimento de software baseado nas técnicas e metodologias identificadas. O modelo proposto será formalizado na linguagem de metamodelo de processo de software SPEM a fim de obter melhor compreensão das iterações entre as atividades do processo.

Palavras-chave: Processo de desenvolvimento de software. Software open source. Modelagem de processo de software.

Abstract

CAVALHEIRO, Lucas Diego Zwirtes. **MODELAGEM DE UM PROCESSO DE DESENVOLVIMENTO DE SOFTWARE BASEADO NAS TÉCNICAS E METODOLOGIAS DE PROJETOS OPEN SOURCE**. 2007. 111fl. Trabalho acadêmico (Graduação) – Instituto de Física e Matemática. Universidade Federal de Pelotas.

The usage of a software process development model is all-important to the course of a project and to guarantee the delivery of a full working system in the deadline estimated at the beginnng of the projetct. Since the software 'crisis' various metodologies was created fetching to match the development process with the new technologies and the new business models. The communities of open source development produces good quality software and with a admissible ratio of bugs. Each open source project has specific techniques and methodologies, nevertheless they shares some characteristics like the geographic distribution of the developers and the active communication among development team members and the community. Large projects like GNU/Linux, Apache and Mozilla are instances that stimulated both academics and commercial interest in the open source software development process. However there is lacks into the development process and the very specific characteristics avoid other projetcts from usage this methodology, this problems are chances to create hybrids development processes. This work starts from a qualitative analysis of the open source projects develpment processes aim at to formalize a software development process model based on identified techniques and methodologies. The proposed model will be formalized with the SPEM software process metamodel language to acquire a better understanding of the iterations among process activities.

Keywords: Software development process. Open source software. Software process modeling.

Lista de figuras

Lista de abreviaturas e siglas

CASE – Computer-aided software engineering
CMM – Capability Maturity Model
CORBA – Common Object Request Broker Architecture
COTS – Commercial off-the-shelf
CVS – Control Version System
E-MAIL – Eletronic Mail
FTP – File Transfer Protocol (protocolo de transferência de arquivos)
GCC – Gnu Compiler Collection
GNU – Gnu is Not Unix
IDE – Integrated Development Environment
IP – Internet Protocol
IRC – Internet Relay Chat
NCSA – National Center for Supercomputing Applications
OSI – Open Source Initiative
PHP – PHP Hypertext Processor
RAD – Rapid application development
RUP – Rational unified process
SPEM – Software Process Engineering Metamodel
TI – Tecnologia da Informação
UML – Unified Modelling Language
XP – Extreme programming
WEB – Referente à World Wide Web (Internet)

Sumário

1 Introdução

À medida que novas tecnologias são desenvolvidas e que novas oportunidades são criadas, a necessidade de criar e/ou adaptar um novo processo de software aparece. Por exemplo, com o advento da Internet surgiu a necessidade de adaptar os processos de software existentes para se obter um melhor desenvolvimento de sites de comércio eletrônico, que demandam uma segurança extrema. O processo de software está intimamente ligado com a qualidade, desempenho e funcionalidade do sistema. Sem a adoção de um modelo de processo de desenvolvimento de software, um projeto possui maiores chances de falhar visto que a utilização de práticas e metodologias define um conjunto padrão de regras e ações para organizar e gerenciar o processo como um todo, evitando assim que o caos se instale. Com a ascensão cada vez maior da utilização de softwares open source, pesquisadores e engenheiros de software voltaram suas atenções para o processo de desenvolvimento deste tipo de software. Para a surpresa da grande maioria alguns pontos do desenvolvimento de software open source vão contra os modelos clássicos de desenvolvimento de software. Desde então estudos de projetos de software open source têm sido realizados. Cada projeto possui particularidades em seu desenvolvimento, tornando muito difícil propor um modelo de processo que englobe totalmente todos os projetos. Contudo, alguns pesquisadores acreditam que o próximo “salto” em desenvolvimento de software será quando for possível aplicar o processo de software open source, seja no meio acadêmico ou nos grandes desenvolvedores de software. Então algumas modelagens do processo têm sido propostas, abstraindo particularidades e buscando identificar atividades comuns nos processos da maioria dos projetos open source (ou pelo menos nos projetos de maior êxito).

1.1 Motivação

Desde a “crise do software” os sistemas passaram a ser desenvolvidos utilizando-se da organização e padronização das atividades do desenvolvimento para elevar a qualidade e reduzir o custo do software. A escolha correta do processo de software mais adequado à aplicação que se deseja desenvolver é fundamental. Os modelos de processo de software são uma descrição simplificada das atividades de desenvolvimento e são importantes para facilitar a escolha do processo mais adequado ao problema analisado. Não existe um modelo aplicável a todos os tipos de sistemas e novos processos de desenvolvimento foram sendo desenvolvidos conforme as necessidades do cliente ou dinamismo dos modelos de negócio que regem o mercado. Para a obtenção de melhores softwares, é necessária uma melhoria dos processos de desenvolvimento. Sommerville (2003, p.477) observa que “A melhoria do processo significa compreender os processos existentes e modificá-los, a fim de melhorar a qualidade do produto e/ou reduzir custos e o tempo de desenvolvimento”.

Alguns autores não configuram o desenvolvimento de software open source entre os modelos tradicionais de processo de software. Como grande parte de seus desenvolvedores são imbuídos da cultura *hacker*, ao invés de seguir um modelo de processo de software tradicional, cada projeto de software open source utiliza várias técnicas e métodos diferentes, algumas oriundas de modelos de desenvolvimento existentes outras constituindo novas abordagens para problemas existentes. Oficialmente, open source significa que o código fonte deve estar disponível para redistribuição sem restrições e sem encargos, e a licença deve permitir a criação de modificações e trabalhos derivados, e deve garantir que essas derivações sejam redistribuídas sob os mesmos termos do trabalho original (O’REILLY, 1999, p.1). O movimento open source criou um modo diferenciado de desenvolver software competitivo e com baixo custo, atraindo assim a atenção das grandes empresas de TI. O desenvolvimento open source tipicamente tem uma pessoa central (ou um “corpo” de pessoas) que seleciona uma coleção de códigos desenvolvidos para compor a versão final do software, tornando-a amplamente disponível para distribuição além de que o grande número de colaboradores voluntários é que se encarregam do trabalho que desejam realizar – caracterizando um processo de desenvolvimento diferente dos projetos comerciais. Outra grande diferença é que no

processo open source não existem explicitamente modelos detalhados em nível de sistema, nem mesmo um plano de projeto. Apesar da ausência dos mecanismos tradicionais para coordenar o desenvolvimento de software - planejamento, modelagem do sistema e definição dos processos - o desenvolvimento de software open source tem produzido aplicações de boa qualidade e funcionalidade. Onde aparentemente parece ser o caos na verdade constitui um caso de desenvolvimento distribuído de software, em que os desenvolvedores trabalham em diferentes localidades, raramente ou nunca se encontram pessoalmente, e coordenam suas atividades quase que exclusivamente por mensagens eletrônicas (MOCKUS, FIELDING e HERBSLEB, 2002, p.1).

A motivação deste trabalho reside justamente na busca por um processo de desenvolvimento de software que através da melhor comunicação entre os participantes do processo em conjunto com iterações dinâmicas se obtenha um processo híbrido com menores custos sem abrir mão da qualidade. Devido ao potencial econômico e por dimensionar um novo modelo de negócio, pode ser uma tendência o software open source dar lugar a novos modelos híbridos (FELLER, J., FITZGERALD, B., 2000, p.68). Também é possível que o modelo proposto seja uma alternativa para empresas e grupos de desenvolvedores que desejem implementar o processo de desenvolvimento de software distribuído visto que a distância física entre as equipes de desenvolvimento, a comunicação entre equipes que falam diferentes idiomas, a integração de software e utilização de diferentes processos de desenvolvimento constituem problemas que limitam a utilização dessa estratégia de desenvolvimento (BENNATAN, 2002, p.7-10). Outro fator que impulsiona este trabalho é a falta de documentação relativa ao processo de desenvolvimento de software open source. Nenhum site dos projetos de software open source disponibiliza de forma explícita algum tipo de modelo ou esquema detalhado do processo de software empregado. Isto acaba por afastar o interesse das equipes de desenvolvimento de software (LONCHAMP, 2005, p.2). A modelagem de um processo de software baseado no processo open source pode ser um meio de incentivar futuros trabalhos nesta área.

1.2 Objetivos

O objetivo desse trabalho é analisar diversas pesquisas sobre projetos de software open source para definir um processo de desenvolvimento de software que utiliza as prerrogativas do desenvolvimento open source, identificando os pontos que tornam inviável sua aplicação em empresas e equipes de desenvolvimento. Definido esses pontos, propomos um conjunto de atividades alternativas para preencher as carências do modelo open source gerando assim um novo modelo. Para modelar esse processo de desenvolvimento diferenciado será adotado o metamodelo SPEM, utilizado para descrever processos de desenvolvimento de software através de uma abordagem orientada a objeto.

1.3 Metodologia

A metodologia utilizada neste trabalho consiste em uma análise qualitativa de diferentes estudos etnográficos e baseados em questionários/entrevistas sobre os processos de desenvolvimento de software open source. Esta análise qualitativa é fundamentada em dois tipos de procedimentos de análise. Um dos procedimentos utilizados é o *Grounded Theory*, desenvolvido na sociedade norte-americana nas décadas de 1950 e 1960, em que ainda no levantamento de informações se admite passos de construção de conceitos (geralmente indutivos) e teorias, visto que os pesquisadores já refletem e analisam os conceitos implícitos durante a coleta de dados. Desta forma, levantamento e análise de dados se sobrepõem. O outro procedimento utilizado é a *análise fenomenológica*, caracterizada pela descrição dos fenômenos do ponto de vista do sujeito e tomando suas intenções como ponto de partida. Comparando um fenômeno em diversas situações, a parte invariante pode indicar a natureza do fenômeno e possibilita uma descrição mais detalhada do núcleo essencial (MAYRING, 2002).

Este trabalho foi então estruturado em mais quatro capítulos além desta introdução. No **capítulo dois** será feito um estudo sobre o estado da arte na modelagem de processo de software, procurando abordar seus fundamentos, objetivos e algumas metodologias relevantes para a realização deste trabalho. No

capítulo três será realizada uma abordagem histórica sobre o software open source e a análise de trabalhos que buscam identificar as práticas e metodologias utilizadas no processo de desenvolvimento de software deste tipo de projeto. Já no **capítulo quatro** será proposto um modelo de desenvolvimento de software que é baseado no processo identificado no capítulo três. O **capítulo cinco** será colocado em debate as conclusões obtidas com a realização deste trabalho.

1.4 Contribuições esperada

Desenvolver um modelo de processo de desenvolvimento de software caracterizado pelas técnicas e atividades desenvolvidas em projetos open source, melhorando o desenvolvimento de software. Espera-se que este modelo proposto possa ser utilizado por empresas do ramo de software como também qualquer outro grupo de desenvolvedores que buscam uma alternativa aos processos de softwares existentes, redução do trabalho de desenvolvimento ou uma maior flexibilização do processo de software. Há também a expectativa de que o modelo proposto possa contribuir com pesquisas realizadas na área de desenvolvimento distribuído de software, visto que grandes projetos open source apresentam em seu processo de desenvolvimento de software características de natureza distribuída.

2 Processo de software

Desde o surgimento da engenharia de software estudiosos e profissionais têm estudado incessantemente (e desenvolvido) novos processos de desenvolvimento de software. Não é para menos. O software está intrinsecamente incorporado ao cotidiano de nossas vidas. A importância e a complexidade dos sistemas de software atualmente atingiram um patamar tão elevado que o menor descuido no processo de desenvolvimento pode levar ao fracasso ou a perdas intangíveis.

Sommerville (2003, pg.7) define processo de software como "conjunto de atividades e resultados associados que geram um produto de software". Pode-se imaginar o processo de software como sendo um roteiro, onde cada etapa é descrita detalhadamente.

> Um processo de software define a abordagem que é adotada quando o software é elaborado. A elaboração de software de computador é um processo interativo de aprendizado, e o resultado é um conhecimento personificado acumulado, destilado e organizado, à medida que o processo é conduzido (PRESSMAN, 2002, p.17).

Antes de abordar o processo de desenvolvimento de software, é importante contextualizar o desenvolvimento de software na época em que surgiram as primeiras idéias da disciplina de engenharia de software bem como seus objetivos e abrangência. Assim compreende-se melhor a importância da criação de processos de desenvolvimento de software

2.1 A engenharia de software

As primeiras idéias de engenharia de software surgiram pela primeira vez no final de década de 60, quando o desenvolvimento de software passava por uma "crise". Nessa época, ocorreu uma grande revolução em termos de hardware. Os

computadores estavam ficando cada vez menores e aumentando o poder de processamento – os chamados computadores de terceira geração. Conseqüentemente, a demanda por softwares que explorassem esse potencial também cresceu. No entanto, o desenvolvimento de software continuava ocorrendo de modo informal e desorganizado. Os resultados dessa prática não eram nada animadores: softwares apresentavam custos maiores que os previstos, os sistemas não eram confiáveis, de difícil manutenção e desempenho inferior ao esperado (SOMMERVILLE, 2003, p. 4).

Diante deste quadro, especialistas da área reuniram-se em uma conferência organizada para debater esta crise no desenvolvimento de software. Os custos do software subiam enquanto os custos de hardware caíam. Os sistemas estavam cada vez maiores e mais complexos do que os softwares produzidos anteriormente. Novas técnicas e metodologias de desenvolvimento de software eram necessárias para superar esses obstáculos.

A engenharia de software, baseada nos princípios de engenharia, surgiu para definir, orientar e compartilhar processos eficazes de desenvolvimento de software. Como bem ressalta Sommerville (2003, p.5), a engenharia de software "se ocupa com todos os aspectos da produção de software, dada à especificação do sistema até a manutenção. [...] gerenciamento de projetos de software e o desenvolvimento de ferramentas, técnicas e teorias que dêem apoio à produção".

Pressman (2002, p.18) reforça esta definição dizendo que a engenharia de software é a "aplicação de uma abordagem sistemática, disciplinada e quantificável, para o desenvolvimento, operação e manutenção do software". Pressman ainda complementa lembrando que também cabe a engenharia de software o estudo de novas abordagens para o desenvolvimento de software.

Para produzir software de qualidade, os engenheiros de software utilizam métodos e ferramentas de engenharia de software.

> *Métodos* de engenharia de software fornecem a técnica de como fazer para construir software. Os métodos incluem um amplo conjunto de tarefas que abrange análises de requisitos, projeto, construção de programas, teste e manutenção. Conjunto de princípios básicos, que regem cada área da tecnologia e incluem atividades de modelagem e outras técnicas (PRESSMAN, 2002, p.19).

Os métodos de engenharia de software começaram a ser desenvolvidos na década de 70 e buscam desenvolver modelos gráficos de um sistema que possam ser utilizados como uma especificação ou projeto de sistema. Para Sommerville

(2003, p.10) os métodos devem incluir 4 componentes principais. **Descrição de Modelos de Sistema**, consiste no detalhamento dos modelos a serem desenvolvidos e a notação utilizada para definir esses modelos, como por exemplo modelos de objetos, modelo de fluxo de dados, etc. **Regras** são as restrições que sempre se aplicam a modelos de sistemas, como “cada entidade em um modelo de sistemas deve ter um único nome”. **Recomendações** são heurísticas que caracterizam a boa prática de projeto nesse método. Seguindo essas recomendações, deve-se chegar a um modelo de sistema bem organizado. Um tipo de recomendação seria “nenhum objeto deve ter mais do que sete subobjetos associados a ele”. Outro componente essencial são as **diretrizes de processo**, que descreve as atividades que podem ser seguidas para desenvolver os modelos de sistema e a organização dessas atividades. Um exemplo seria “atributos de objetos devem ser documentados”.

Ferramentas de engenharia de software, também chamadas de ferramentas CASE (*engenharia de software apoiada por computador*) são definidas por Sommerville (2003, p.11) como “uma ampla gama de diferentes tipos de programas utilizados para apoiar as atividades de processo de software, como a análise de requisitos, a modelagem de sistema, a depuração e os testes”. Todos os métodos de desenvolvimento de software possuem algum tipo de apoio automatizado ou semi-automatizado para o seu processo.

> Quando ferramentas são integradas, de modo que a informação criada por uma ferramenta pode ser usada por outra, um sistema para o apoio de desenvolvimento de software, chamado *engenharia de software apoiada por computador*, é estabelecido. Combinam software, hardware e uma base de dados de engenharia de software (um depósito que contém informação importante sobre análise, projeto, construção de programas e teste) (Pressman, 2002, p. 19).

Estes aparatos fornecidos pela engenharia de software somente irão conduzir a um software de qualidade, seguro e de fácil manutenção se associados a um processo de software.

2.2 Processo de software e suas atividades

Após vários anos de desenvolvimento de sistemas buscando formar práticas de engenharia de software, diversos processos de software foram criados. Algumas vezes por causa dos requisitos muito específicos do software a ser desenvolvido

(por exemplo, um sistema de controle de um satélite) outras vezes por causa do tempo de realização do trabalho (muito curto ou falta de mão-de-obra para desenvolver).

No entanto o processo de desenvolvimento de software tem quatro atividades fundamentais, que são encontradas em todos os processos: **Especificação do Software** é a fase de definição de funcionalidades e restrições. Geralmente é realizado o processo de engenharia de requisitos, com a participação do cliente e/ou usuário final do sistema. Também se determina qual solução computacional será utilizada (definição de entradas, saídas e entidades bem como suas relações). Como resultado, produz um documento de especificação dos requisitos e uma descrição dos componentes e suas iterações utilizando-se de textos e diagramas (por exemplo, uma modelagem UML). O **Desenvolvimento do Software** é a fase onde o software é produzido atendendo as suas especificações. Esta etapa envolve gerar, depurar e integrar código fonte. **Validação do Software** é a atividade progressiva e sistemática. Deve garantir que o software faz o que o cliente deseja. Destacam-se os testes de unidade (funções, módulos, classes e interfaces), testes de integração (todas essas partes combinadas devem seguir funcionando corretamente) e o teste de sistema (aspectos funcionais, desempenho e estabilidade). Na **Evolução do Software** o sistema, naturalmente, deve evoluir para atender necessidades mutáveis do cliente. Muitos erros/falhas costumam ocorrer após a entrega do software. A manutenção de um software geralmente demanda mais tempo – e custo – do que o desenvolvimento do produto. Além disso, os requisitos do cliente também tendem a modificar devido às inovações tecnológicas, mudanças nos negócios ou para atender algum tipo de demanda dos seus usuários (SOMMERVILLE, 2003, p.7).

Os diversos processos de software organizam essas atividades de várias maneiras e são descritos em diferentes níveis de detalhe. Além disso, os prazos das atividades variam, assim como os resultados que cada etapa produz. Um ponto interessante a se considerar é que diferentes organizações podem utilizar processos diferentes para obterem o mesmo tipo de software.

Atividades auxiliares (também conhecidas como atividades *guarda*-chuva) podem ser usadas para complementar as atividades fundamentais a fim de otimizar recursos e minimizar os riscos. Pressman (2002, p.21) exemplifica atividades

auxiliares como “[...] garantia da qualidade do software, gestão de configuração de software e medição cobrem o modelo de processo”.

No entanto, essa variedade de processos de desenvolvimento de software não significa que qualquer processo possa ser utilizado em qualquer circunstância. Alguns processos são mais adequados do que outros, dependendo do tipo de aplicação que se pretende desenvolver. A utilização de um processo inadequado provavelmente reduzirá a qualidade ou utilidade do produto a ser desenvolvido. Cabe aos engenheiros de software identificar qual modelo é mais adequado diante dos requisitos identificados.

2.3 Modelos de processo de software

Um modelo de processo de software é uma representação abstrata das atividades e métodos envolvidos no desenvolvimento de software. Pressman (2002, p.24) define que “modelo de processo de software (ou paradigma de engenharia de software) é a estratégia de desenvolvimento que abrange as camadas de processo, os métodos e ferramentas CASE”. A modelagem de processo de software ajuda na compreensão e comunicação do processo, comparação, reuso e aperfeiçoamento de processos além de guiar às atividades da equipe de desenvolvimento através de suas representações do processo (Lonchamp, 2005, p. 28). Em seu outro trabalho, Lonchamp (1993, p.45) afirma que “Muitos tipos de informações podem ser integradas em um modelo de processo de software para responder perguntas básicas como o que, quem, como, onde e por que”. Os modelos podem auxiliar no desenvolvimento de novos processos (descrevendo a estrutura), facilitar a compreensão do processo, simular e aperfeiçoar um processo (identificação de problemas, gargalos e oportunidades) além da possibilidade de aplicação educacional (Fuggetta, 2000, p. 27). A seguir são abordados os modelos de processo de software de maior importância para o nosso estudo.

2.3.1 Modelo cascata

É o modelo de processo de software mais antigo – e mais utilizado. Originalmente publicado por Walker Royce em 1970 como “ciclo de vida do

software", passou a ser conhecido também por "modelo cascata" devido à forma como ocorre à transição de uma fase para a outra (Fig. 1). Uma fase não inicia até que a fase precedente tenha terminado, com a aprovação dos documentos gerados. Isto acarreta um alto custo das iterações. Na prática, no entanto, ocorre certa iteração entre as fases. Neste modelo o processo de desenvolvimento de software é distribuído em cinco etapas (SOMMERVILE, 2003, p.37). O processo inicia com a etapa de **análise e definição de requisitos** que ocorre mediante consulta ao usuário do software, que define as funções, restrições e os objetivos do sistema. Serve como uma especificação. A segunda etapa do modelo cascata é o **projeto de sistemas de software**, que agrupa os requisitos em sistemas de hardware ou software. Envolve a identificação e descrição das abstrações fundamentais do sistema de software e suas relações.

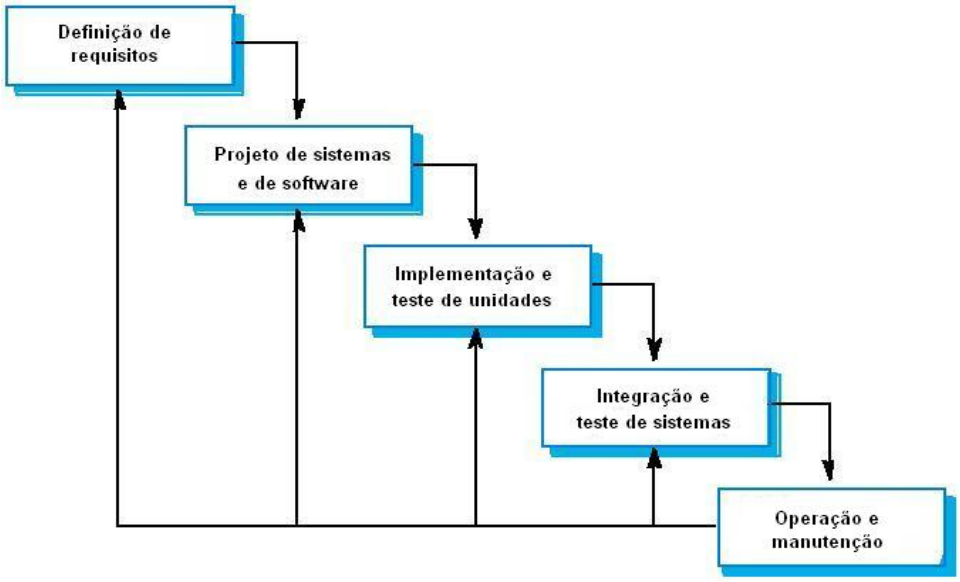

Figura 1 – Modelo cascata de processo de software.

Fonte: SOMMERVILLE, 2003, p.38.

A terceira etapa do modelo cascata de desenvolvimento é a **implementação e teste de unidades**, sendo o projeto de software compreendido como um conjunto de programas ou unidades de programas. Os testes verificam que cada unidade atenda suas especificações. A quarta etapa do processo é a **integração e teste de sistema**. As unidades/programas individuais são integrados e testados como um sistema completo, para garantir que os requisitos de software foram atendidos. Depois dos testes, o sistema é entregue ao cliente. A última etapa é conhecida por **operação e manutenção**. Normalmente é a fase mais longa. Correção de erros não

descobertos nos estágios anteriores, melhorando a implementação das unidades e aumentando as funções do sistema. O suporte/manutenção do software reaplica cada uma das fases precedentes ao sistema desenvolvido.

Partes iniciais do projeto vão sendo suspensas com o andamento do processo e problemas encontrados têm sua resolução postergada. Por exemplo, na fase de codificação a análise de requisitos pode ter sido suspensa e o cliente pode desejar novas funcionalidades. Haverá um grande retrabalho visto que a solução desse problema é adiada e todas as fases do ciclo de vida do software terão que ser executadas novamente. Outro problema verificado é o tempo de bloqueio que pode ocorrer entre uma atividade e outra.

> Numa análise interessante de projetos reais, Bradac [BRA94] descobriu que a natureza linear do ciclo de vida clássico leva a “estado de bloqueio”, nos quais alguns membros da equipe de projeto precisam esperar que outros membros completem as tarefas dependentes. Na realidade, o tempo gasto em espera pode exceder o tempo gasto no trabalho produtivo! O estado de bloqueio tende a ocorrer mais no início e no fim de um processo seqüencial linear. (PRESSMAN, 2002, p.27-28)

É possível observar que no modelo cascata há uma inflexível divisão do projeto nesses estágios distintos. Como a análise e especificação do projeto devem ser feitos em um estágio inicial do processo, é difícil responder a todos os requisitos do cliente, que estão em constante modificação. Esse modelo de processo é recomendado quando os requisitos do projeto forem bem compreendidos.

2.3.2 Modelo evolucionário

O princípio deste modelo de processo de software é desenvolver uma implementação inicial, expor o resultado a avaliação do usuário e fazer as modificações necessárias, aprimorando a solução por meio de muitas versões até que um sistema adequado tenha sido desenvolvido. Sommerville (2003, p.39) comenta que “ao invés de ter as atividades de especificação, desenvolvimento e validação em separado, todo esse trabalho é realizado concorrentemente com um rápido *feedback* por meio dessas atividades”. Duas formas de desenvolvimento evolucionário são identificadas na literatura. O *desenvolvimento exploratório* procura trabalhar com o usuário, explorando seus requisitos na busca de um sistema final. O desenvolvimento inicia com as partes do sistema que são compreendidas. A evolução ocorre com o acréscimo de novas características, à medida que são

propostas pelo cliente (Fig. 2). A outra forma é o uso de *protótipos*, que são usados para compreender melhor os requisitos do cliente e assim desenvolver um sistema mais adequado. O protótipo esclarece partes dos requisitos que não estejam bem compreendidos, como bem observa Pressman (2002, p.28) ao dizer que “Interações ocorrem à medida que o protótipo é ajustado para satisfazer às necessidades do cliente, enquanto, ao mesmo tempo, permitem ao desenvolvedor entender melhor o que precisa ser feito”. Na maioria dos casos os protótipos são descartados após cumprirem sua finalidade, pois seu propósito não é segurança, desempenho ou estabilidade, e sim apenas auxiliar a definição dos requisitos do sistema.

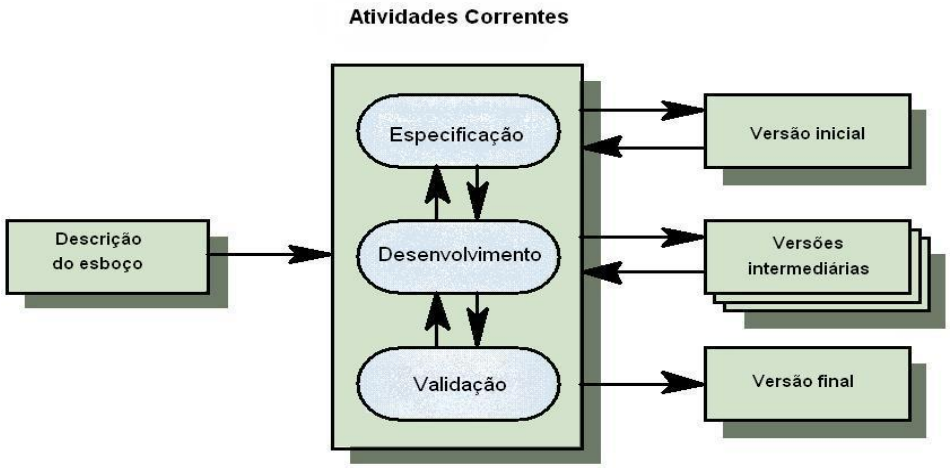

Figura 2 – Modelo de processo de software evolucionário.

Fonte: SOMMERVILLE, 2003, p.39.

O modelo de processo de software evolucionário é mais vantajoso do que o modelo cascata na questão de análise e especificação de requisitos, pois enquanto no modelo cascata só ocorre na etapa inicial, no modelo evolucionário ocorre em todo o processo de desenvolvimento. Desta forma, o retrabalho ao final do processo é muito reduzido. Além disso, o usuário já recebe uma primeira versão do sistema logo no início do desenvolvimento, enquanto que no modelo cascata é preciso esperar a fase de operação/manutenção. No entanto, para sistemas de grande complexidade, não é recomendado.

> Não é ideal para sistemas de grande porte, de longo tempo de vida. Recomenda-se usar um processo misto, que incorpore as melhores características dos modelos cascata e evolucionário. Pode-se realizar a especificação e interface do usuário utilizando o desenvolvimento evolucionário, através de protótipos descartáveis, e nas outras fases utilizar modelo cascata a fim de obter um sistema melhor estruturado. (SOMMERVILLE, 2003, p.40)

Outras desvantagens que são observadas no modelo evolucionário é que o processo não é visível e a geração de documentos é inviável devido à alta interação entre os estágios. As mudanças constantes tendem a corromper o sistema e podem exigir conhecimento especializado de algum tipo de técnica ou ferramenta especializada (SOMMERVILLE, 2003, p.39-40).

2.3.3 Modelo de desenvolvimento formal

Também conhecido como “métodos formais”, esse processo de software é semelhante ao modelo cascata e baseia-se na transformação matemática formal (utilizando-se métodos como o método-B e as redes de Petry) de uma especificação de sistema em um programa executável. Pressman (2002, p.41) diz que “Métodos formais permitem ao engenheiro de software especificar, desenvolver e verificar um sistema baseado em computador, pela aplicação de uma rigorosa notação matemática”.

> Quando são usados métodos formais durante o desenvolvimento, um mecanismo para eliminação de vários problemas difíceis de resolver usando outros paradigmas de engenharia de software é fornecido. Ambigüidade, inconclusão e inconsistência podem ser descobertas e corrigidas mais facilmente, não através das revisões comuns, mas através da aplicação da análise matemática. (PRESSMAN, 2002, p.41)

Esse modelo possui duas diferenças em relação ao modelo cascata. A primeira é em relação à especificação de requisitos. No desenvolvimento formal ocorre uma especificação formal detalhada e expressa em notação matemática (a teoria dos autômatos e a representação de máquinas de estado finitas podem ser utilizadas). A outra diferença refere-se às atividades de projeto, implementação e teste de unidades. No lugar dessas atividades ocorre um processo de transformações formais.

Sommerville (2003, p.41) comenta que neste processo de transformação “[...] a representação matemática formal do sistema é sistematicamente convertida em uma representação de sistema mais detalhada, mas ainda matematicamente correta. Cada etapa acrescenta mais detalhes, até que a especificação formal seja convertida em um programa [...]” (Fig. 3). Recomenda-se o uso deste tipo de modelo de processo para o desenvolvimento de sistemas críticos em termos de segurança, confiabilidade e garantia, por exemplo, software para aviões e ônibus espaciais.

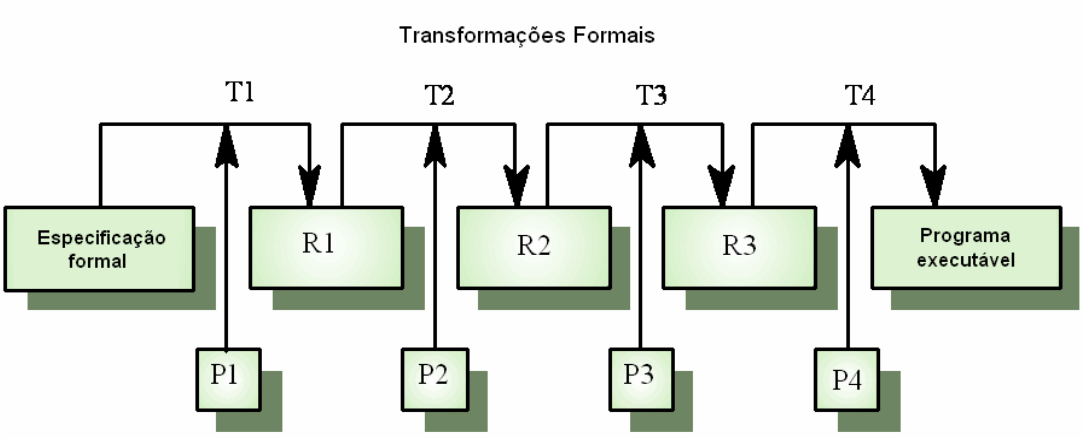

Figura 3 – Transformações formais.

Fonte: SOMMERVILLE, 2003, p.40.

O exemplo mais conhecido desse modelo de processo de desenvolvimento de software é o processo *Cleanroom*, desenvolvido em 1987 pela IBM (Mills e Selby) e aprimorado posteriormente (Linger, 1994; Prowell et al., 1999). O Cleanroom realiza um desenvolvimento incremental do software, em que cada estágio é desenvolvido e sua correção é demonstrada através de uma abordagem formal. Não são executados testes para encontrar defeitos no processo e o teste do sistema busca medir a confiabilidade do sistema (SOMMERVILLE, 2003, 41).

Destacam-se três problemas na aplicabilidade comercial do modelo de processo de desenvolvimento formal. O desenvolvimento de métodos formais é muito lento e dispendioso (a interação do sistema não é sensível à especificação formal), existem poucos desenvolvedores capacitados a aplicar métodos formais e é difícil utilizar os métodos formais para como um meio de comunicação com os clientes – visto que eles não possuem conhecimentos em métodos formais (PRESSMAN, 2002, p.41).

2.3.4 Desenvolvimento orientado ao reuso

A reutilização de software é uma prática que há algum tempo já ocorre nos centros de desenvolvimento de software, sendo geralmente realizada de modo informal pelos desenvolvedores que possuem conhecimento do código-fonte de outros sistemas previamente desenvolvidos. Mesmo que às vezes seja necessário realizar adaptações no código, essa prática proporciona um desenvolvimento mais

rápido de software. O paradigma de programação orientado a objetos se utiliza fortemente deste conceito de reuso, como destaca Pressman (2002, p.39) ao dizer que “Classes orientadas a objetos, se adequadamente projetadas e implementadas, são reusáveis ao longo de diferentes aplicações e arquiteturas [...]”.

Também conhecido como “modelo de desenvolvimento baseado em componentes”, o processo de software orientado a reuso dispõe de uma larga base de componentes de software reutilizáveis, que podem ser acessados, e com algum tipo de infra-estrutura para integração desses componentes (SOMMERVILE, 2003, p.1). As fases de especificação de requisitos e validação não são diferentes dos outros modelos de processo de desenvolvimento de software. Porém os estágios intermediários são diferentes, pois é necessário um processo para reutilizar os componentes já existentes (Fig. 4).

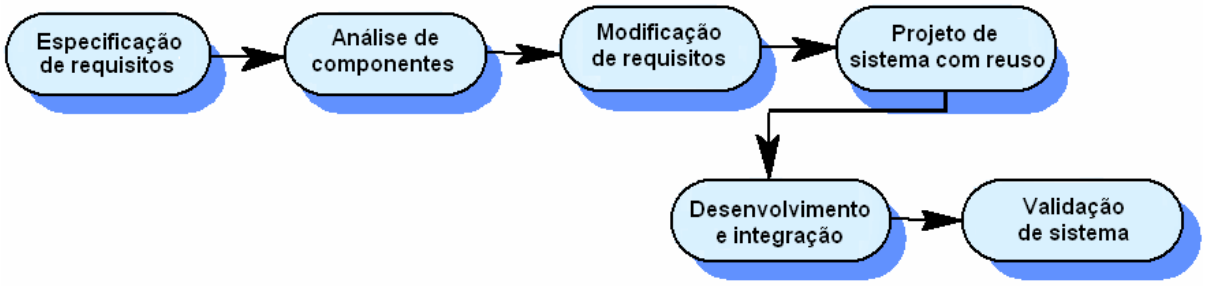

Figura 4 - Modelo de processo de desenvolvimento de software orientado a reuso

Fonte: SOMMERVILLE, 2003, p.42.

Após a atividade de especificação dos requisitos, é feita uma busca por componentes já implementados em outros projetos e cada componente é avaliado se pode ser reutilizado para implementar alguma das funcionalidades requeridas. Em muitos casos os componentes disponíveis não satisfazem os requisitos exigidos. É importante destacar o uso de diagramas UML para definição dos componentes e de suas interfaces.

Depois de selecionados os componentes, os requisitos do sistema são reavaliados, considerando os componentes que foram encontrados, e modificados quando possível a fim de conseguir reutilizar esses componentes. Se não for possível realizar modificações nos requisitos, uma nova análise pode ser feita no repositório de componentes em busca de uma solução alternativa. Tendo todos os componentes selecionados, um projeto de sistema deve ser feito. Uma infra-

estrutura do sistema é projetada (ou uma existente deve ser reutilizada). Devem-se considerar os componentes que são reutilizados para organizar a infra-estrutura. Se os componentes não estiverem disponíveis, um novo software poderá ser projetado. A integração dos componentes pode ser feita através de algum software disponível no mercado ou poderá ser desenvolvido pela equipe. Os componentes e sistemas COTS (comercialmente disponíveis) são integrados, a fim de criar um sistema. No modelo orientado ao reuso, a integração de sistemas pode ser incorporada ao processo de desenvolvimento (SOMMERVILLE, 2003, p.42).

O modelo de desenvolvimento orientado ao reuso é muito vantajoso. Com o reuso a quantidade de código novo desenvolvido pela equipe diminui, reduzindo também o tempo de entrega do sistema, os custos (menos trabalho) e os riscos (os componentes existentes já foram testados e aprovados em outros projetos). A QSM Associates, Inc. (http://www.qsma.com), empresa da área de gerência de projetos de software, realizou um estudo sobre reuso de software e constatou que a montagem de componentes leva a uma redução de 70% no prazo do ciclo de desenvolvimento, uma redução de 84% no custo do projeto e um aumento do índice de produtividade (PRESSMAN, 2002, p.40-41).

Apesar dessas vantagens, alguns problemas podem surgir durante o desenvolvimento orientado a reuso. A modificação de requisitos, por exemplo, pode levar a um sistema que não cumpra os requisitos do usuário. Deve haver algum tipo de controle de versões dos componentes, há o risco de perder-se o controle da evolução do sistema.

2.4 Iteração de processo

A constante mudança nos requisitos do usuário e nas tendências do mercado (que cobra inovação com rapidez e baixo custo) contribui para dificultar o término de um sistema, pois é necessário retornar às etapas iniciais do processo de desenvolvimento e executa-las novamente. Os modelos de processo abordados anteriormente, apesar de cobrirem um amplo número de situações, nem sempre são totalmente aplicáveis na prática. Dependendo do tamanho e da complexidade do sistema se faz necessário usar mais de um modelo de processo em diferentes partes do software, formando um modelo híbrido de desenvolvimento. Para apoiar

essa iteração que ocorre nos processos, existem dois principais modelos de desenvolvimento híbridos, o *desenvolvimento incremental* e o *desenvolvimento em espiral*. Segundo Pressman (2002, p. 32) “são caracterizados de forma a permitir aos engenheiros de software desenvolver versões cada vez mais completas do software”. A premissa desses modelos de desenvolvimento iterativos é que a especificação do sistema é construída paralelamente ao desenvolvimento do software. Isto gera um conflito com as práticas comerciais, visto que geralmente a especificação completa do sistema faz parte do contrato para o seu desenvolvimento. Pode haver resistência de grandes clientes devido a esse fato e talvez seja necessário o desenvolvimento de um novo tipo de contrato (SOMMERVILLE, 2003, p.43).

2.4.1 Desenvolvimento incremental

Sugerido por Mills em 1980, esse modelo de processo é uma combinação dos modelos cascata e modelo evolucionário baseado em protótipos, como uma tentativa de reduzir o retrabalho e postergar decisões do usuário quanto aos requisitos do sistema. Ao contrário dos protótipos, os incrementos são desenvolvidos objetivando um software operacional a cada acréscimo.

No desenvolvimento incremental a análise dos requisitos é realizada com o usuário, definindo, em uma escala de prioridades, quais requisitos e funcionalidades são mais essenciais e quais são menos importantes. De posse dessa avaliação, se estabelece quantos incrementos serão desenvolvidos e quais os requisitos que cada um irá implementar. Um plano de entrega de incrementos é então elaborado, obedecendo às prioridades do cliente. O primeiro incremento (também chamado de núcleo) contém os requisitos essenciais do cliente, sanando as necessidades mais urgentes.

Após a entrega do primeiro incremento, os demais incrementos – contendo as funcionalidades menos importantes – vão sendo desenvolvidos, validados e integrados, conforme o plano de entregas. Cada incremento pode ser desenvolvido com um modelo de processo diferente. Após a validação do sistema com o incremento integrado, o sistema do usuário é atualizado com o novo incremento

(SOMMERVILLE, 2003, p.43). Este processo ocorre diversas vezes, até que todos os incrementos sejam entregues e o sistema esteja finalizado (Fig. 5).

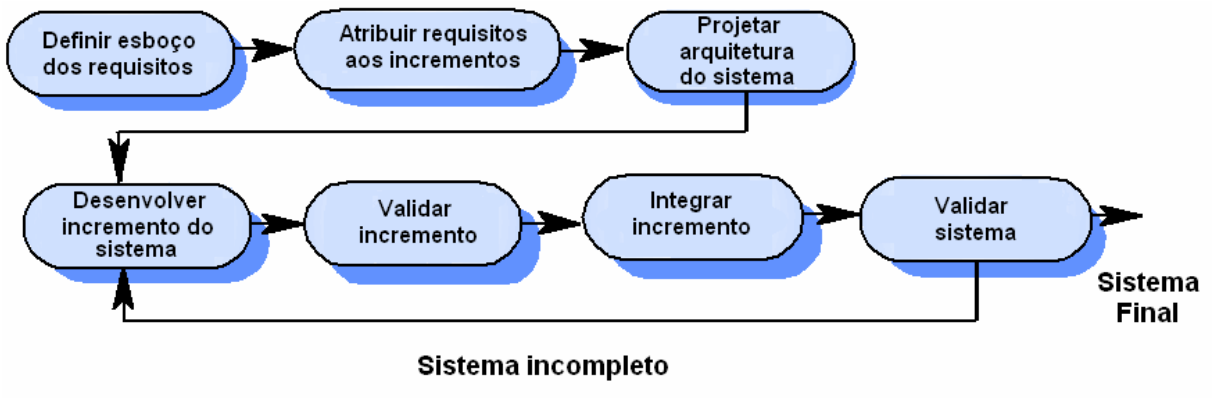

Figura 5 – Modelo de desenvolvimento incremental de software.

Fonte: SOMMERVILLE, 2003, p.43.

Este modelo de desenvolvimento proporciona diversas vantagens, como utilização imediata do software (visto que as principais funções já estão disponíveis no primeiro incremento), os clientes podem esclarecer os requisitos através do primeiro incremento e o número de falhas na parte mais importante do sistema é reduzida (pois é exaustivamente testada) (SOMMERVILLE, 2003, p.44). Pressman (2002, p.33) observa que “O desenvolvimento incremental é particularmente útil quando não há mão de obra disponível para uma implementação completa, dentro do prazo comercial de entrega estabelecido para o projeto”.

Em contra partida pode ser difícil definir os requisitos que cada incremento vai implementar, visto que os incrementos devem ser pequenos. Também fica difícil identificar pontos que sejam comuns a vários incrementos de diferentes partes do sistema. A seguir, são abordadas duas metodologias baseadas no modelo de processo de desenvolvimento incremental que estão sendo muito utilizadas atualmente e merecem ser comentadas.

2.4.1.1 Método XP

A metodologia de desenvolvimento de software *XP* (programação extrema) é um exemplo desenvolvimento incremental. Criada em 1996 por Kent Beck, procura

desenvolver software com simplicidade e maior eficiência. A filosofia do XP se concentra em quatro pontos que podem aperfeiçoar qualquer projeto de software: melhorar a comunicação, procurar simplicidade, obter resposta de quão bem você está desenvolvendo e sempre avançar com coragem. Ideal para pequenos grupos de desenvolvimento e para quando os requisitos do usuário são vagos (BECK, 2000, p.29-33). Muitas falhas de projetos podem estar ligadas a problemas de comunicação da equipe. Essa comunicação envolve engenheiros, cliente, gerentes e programadores. Deve-se desenvolver o software com os requisitos que estão disponíveis. A prática de tentar “adivinhar o que o cliente vai precisar” pode resultar em retrabalho e custos extras de desenvolvimento. O feedback é essencial para que o projeto avance na direção correta (NEWKIRK, 2002, p.695). O método de desenvolvimento XP define *doze práticas* que de alguma forma estão dentro do contexto das quatro idéias da filosofia da programação extrema. Estas práticas relacionam-se entre si para formar o que alguns autores chamam de “ciclo de vida” do XP (Fig.6).

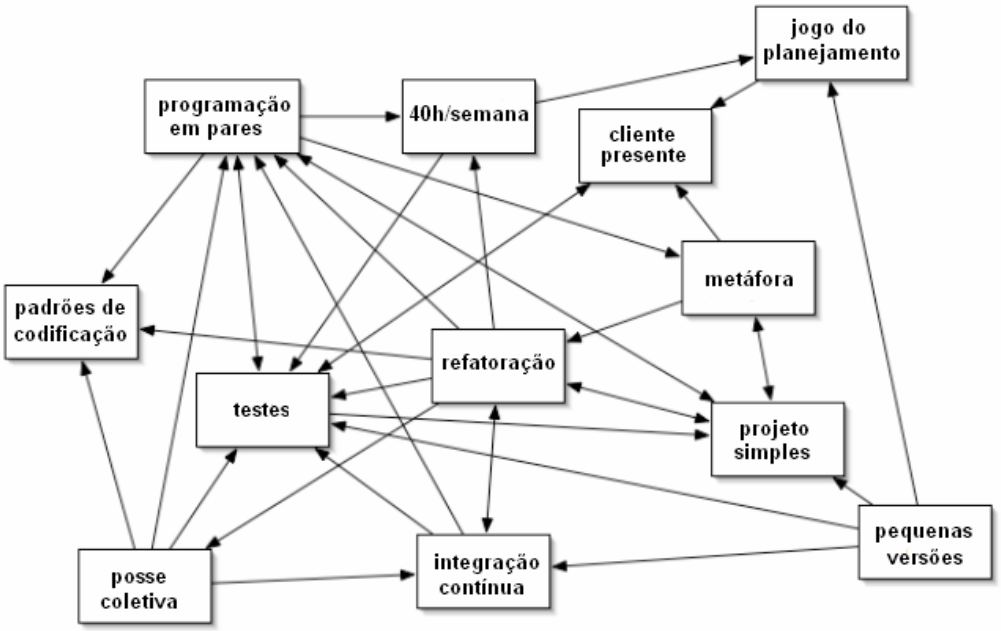

Figura 6 – As 12 práticas do método XP e suas dependências.

Fonte: VANDERBURG, 2005, p.540.

A prática de *planejamento* define o escopo do projeto, sendo o cliente responsável por definir as prioridades do negócio e os programadores responsáveis por fazer as estimativas técnicas. O uso de p*equenas versões* do software é uma

prática chave para obter uma resposta rápida e concreta se o sistema que está sendo desenvolvido corresponde às expectativas do cliente. O uso de *metáforas* é uma prática adotada para fornecer uma visão geral sobre o funcionamento do sistema como um todo. Serve tanto para ajudar o cliente a identificar os requisitos como também aos programadores e projetistas a obterem um conceito mais abstrato de como o sistema funciona. A prática de *projeto simples* é bastante valorizada, não se deve tentar adivinhar o futuro – a probabilidade de errar é alta – e sim trabalhar com as informações que estão disponíveis no momento.

A prática de *testes* indica que o desenvolvimento de software, na metodologia XP, é fortemente orientado a testes. Há os testes de unidade, para avaliação técnica dos incrementos desenvolvidos, bem como os testes de aceitação, em que os clientes fornecem o feedback sobre as partes do sistema que vão sendo disponibilizadas. *Refatoração* é a prática desenvolvida pelos programadores para alterar o software existente sem perda de funcionalidades. A alteração do software é praticamente inevitável durante o seu ciclo de existência, porém a freqüência com que essas mudanças ocorrem pode levar a uma deterioração do sistema caso não sejam bem planejadas e executadas. A *programação em pares* é uma prática adotada baseada na idéia de que “duas cabeças pensam melhor do que uma”.

Apesar de parecer uma idéia contraditória (dois programadores, sentados lado a lado, programando em um único computador) essa prática tem se mostrado mais produtiva visto que enquanto um programador produz código e escreve os testes de unidade, o seu parceiro fica revisando e pensando sobre o código. Outra prática importante é a *posse coletiva*. Qualquer parte do código pode ser modificada por qualquer membro da equipe. A prática de *integração contínua* é realizada diversas vezes pelos programadores durante o projeto, assim o sistema é exaustivamente testado aumentando a qualidade do software.

A prática de *40 horas por semana* procura dar um ritmo sustentável ao desenvolvimento do projeto, sem desgastar a equipe e obtendo assim um rendimento melhor. A prática de *cliente presente* reforça a idéia de que o cliente faz parte da equipe de desenvolvimento e que deve estar sempre disponível para elucidar possíveis dúvidas durante o processo de desenvolvimento. Também cabe ao cliente o desenvolvimento do teste de aceitação. Finalizando as práticas do método XP há a prática de *padrões de codificação*, que é essencial para a boa

comunicação entre os membros da equipe de desenvolvimento, além de evitar que o caos domine no código-fonte (NEWKIRK, 2002, p.695).

O desenvolvimento e a entrega de incrementos de funcionalidades devem ser muito pequenos. O envolvimento do cliente no processo deve ser intenso ("*the customer is always available*") [1] e o cliente deve ser tratado como parte da equipe de desenvolvimento (e não apenas como uma parte do processo). O modelo XP incentiva a constante melhoria de código e a programação impessoal. Os programadores trabalham em pares para desenvolver o código, sendo ambos responsáveis pelos softwares que a dupla produz. Requer pessoal com alta qualificação, visto que trabalha com tecnologias e ferramentas específicas (por exemplo, linguagem de programação Java e conjunto de ferramentas para projetos de desenvolvimento de software). O engajamento dos membros da equipe também deve ser grande devido a grande dependência que existe em relação a participação do cliente em todo o processo de desenvolvimento. (SOMMERVILLE, 2003, p.428).

2.4.1.2 Processo unificado da Rational

É um modelo proprietário de processo de desenvolvimento de software popularmente conhecido por *RUP*, que foi desenvolvido pela Rational Software Corporation – posteriormente adquirida pela IBM – para guiar o processo de desenvolvimento a um software de alta qualidade. Emprega uma abordagem orientada a objetos, sendo que a linguagem UML é utilizada para modelar diversas etapas do processo. O RUP possui duas dimensões que representam a organização do projeto no tempo (eixo horizontal) e as áreas de trabalho – chamadas *disciplinas* – no processo de desenvolvimento (eixo vertical).

O RUP define nove disciplinas (eixo vertical). A *modelagem de negócio* descreve o processo de negócio e sua estrutura interna para obter melhor compreensão do negócio e estar capacitado a identificar os principais requisitos do software a ser construído. A disciplina de *gerenciamento de requisitos* tem por objetivo extrair, organizar e documentar os requisitos do usuário. A *análise e projeto* criam a arquitetura e o projeto do sistema. A *implementação* escreve e depura código fonte, teste de unidade e gerenciamento. A disciplina de *teste* realiza os

[1] "O cliente está sempre disponível" – tradução livre.

testes de integração, de sistema e de aceitação. Na *implantação* ocorre o empacotamento do software e a criação de scripts de instalação. Nesta disciplina se elabora a documentação do usuário final e são realizadas outras tarefas necessárias para tornar o software disponível para todos os usuários. Na disciplina de *gerenciamento do projeto* acontece o planejamento e o monitoramento do projeto. A disciplina de *gerenciamento de configuração e mudanças* cobre todas as tarefas relativas ao gerenciamento de versões, lançamentos e requisição de mudanças. A última disciplina, chamada *ambiente*, adapta o processo às necessidades do projeto e/ou da organização, selecionando, introduzindo e apoiando ferramentas de desenvolvimento (HIRSCH, 2002, p.1-2).

A altura de cada barra associada a cada uma das nove disciplinas representa o quanto de trabalho é despendido para esta disciplina em um determinado instante de tempo (Fig. 7).

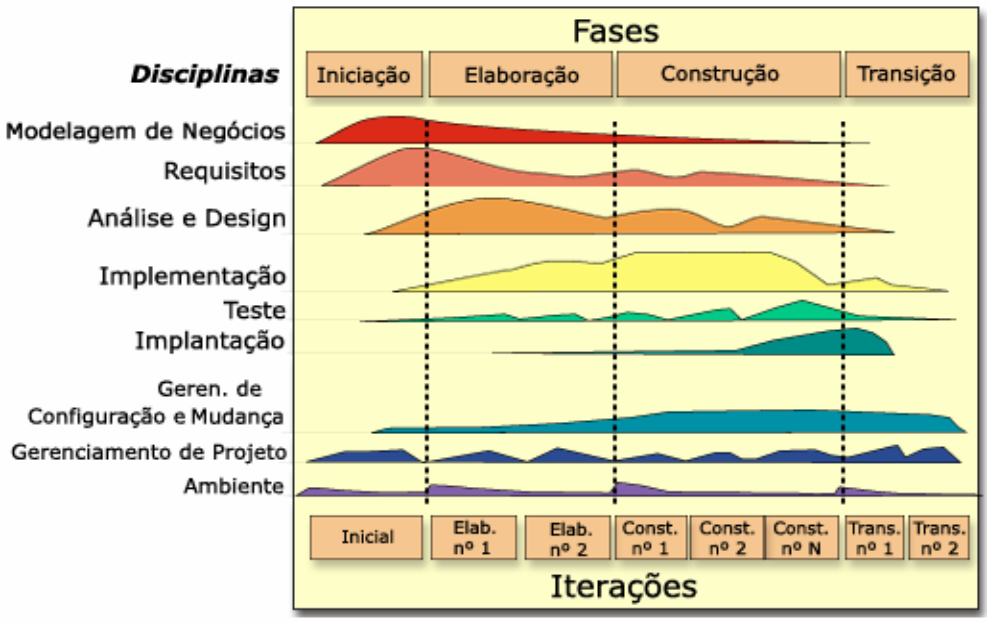

Figura 7 – Fases do RUP e áreas de trabalho

Fonte: HIRSCH, 2002, p.2.

Observe que na fase inicial do projeto, a disciplina de implantação não consome trabalho da equipe – não há o que ser implantado, pois nada foi desenvolvido no sistema. Em compensação a disciplina de requisitos demanda uma alta carga de trabalho da equipe na fase inicial do projeto, o que é esperado visto que esta disciplina é essencial para o sucesso do projeto.

Ao avançarmos no tempo, é possível perceber que a quantidade de trabalho da disciplina de requisitos vai diminuindo gradualmente, o que é esperado, pois quanto mais avançarmos em direção do final do projeto menor será a quantidade de requisitos do cliente que ainda não foram implementadas. Por outro lado a disciplina de implantação vai demandando uma carga de trabalho cada vez maior, que é já era esperado – o sistema já está sendo implementado. Para cada disciplina o modelo define um artefato, uma atividade e funções (“papéis”) (HIRSCH, 2002, p.2).

Um artefato é o resultado de um trabalho como um documento, um código-fonte ou um diagrama UML. Uma atividade é o detalhamento de uma pequena unidade de trabalho que pode criar, modificar, adicionar ou revisar um artefato. Um “papel” é uma responsabilidade que uma ou mais pessoas envolvidas no projeto se comprometem (gerente de projeto, engenheiro de requisitos, codificador,...). No eixo horizontal o projeto é dividido em quatro fases: *iniciação* (onde se define objetivos do projeto), *elaboração* (criação e validação da arquitetura do sistema de software), *construção* (implementação do sistema baseado na arquitetura criada na fase anterior) e *transição* (beta-teste do sistema e preparação de possíveis lançamentos). Em cada uma dessas fases é possível ocorrer uma ou mais iterações, sendo que o objetivo e o tempo de duração de cada iteração são definidos antes da mesma iniciar (HIRSCH, 2002, p.2).

O RUP tem sido largamente utilizado em grandes equipes e grandes projetos. Devido ao seu poder de customização pode ser utilizado para o desenvolvimento de pequenos projetos, porém se sua adaptação não for corretamente implementada o projeto pode demorar mais tempo para ser finalizado.

2.4.2 Desenvolvimento em espiral

Proposto por Boehm em 1988 no artigo *A spiral model of software development and enhancement*, representa o processo de software como uma espiral em que cada loop representa uma fase do processo. O loop mais interno está relacionado à viabilidade do sistema; o loop seguinte, à definição de requisitos do sistema; o próximo loop ao projeto do sistema, e assim por diante. O modelo espiral difere dos outros modelos de processo devido a explicita consideração dos riscos no modelo em espiral (SOMMERVILLE, 2002, p.45).

> Fornece potencial para o desenvolvimento rápido de versões incrementais do software. Usando o modelo espiral, o software é desenvolvido numa série de versões incrementais. A versão incremental, durante as primeiras iterações, pode ser um modelo de papel ou protótipo. Durante as últimas iterações, são produzidas versões cada vez mais complexas do sistema submetido à engenharia (PRESSMAN, 2002, p.33).

Um modelo espiral é dividido em um número de atividades – regiões de tarefas. Geralmente há de três a seis regiões de tarefas. Cada uma das regiões é constituída de um conjunto de tarefas de trabalho. Projetos pequenos demandam menor quantidade de tarefas de trabalho. Projetos maiores, mais complexos, cada região de tarefas contém mais tarefas de trabalho, que são definidas para alcançar um alto nível de formalidade (PRESSMAN, 2002, p.34).

A equipe de desenvolvimento de software move-se em volta da espiral, no sentido horário, a partir do centro. A espiral foi dividida em quatro quadrantes, sendo que cada quadrante representa uma atividade (Fig. 8). O primeiro quadrante é a definição de objetivos. Neste quadrante são executadas as atividades relativas à definição de requisitos, identificação de restrições (no processo e no sistema) e planejamento de ações alternativas (SOMMERVILLE, 2003, p.45). O segundo quadrante é relativo à avaliação das alternativas, identificação e resolução de riscos, sendo desenvolvidas as atividades de análise de riscos. Protótipos são desenvolvidos para esclarecer dúvidas quanto aos requisitos e assim reduzir os riscos do projeto (SOMMERVILLE, 2003, p.45).

O terceiro quadrante é relativo ao desenvolvimento e validação, onde um modelo de desenvolvimento é escolhido para o sistema baseado nas informações da avaliação dos riscos. Por exemplo, se o maior risco é a questão da segurança, o modelo de desenvolvimento formal pode ser escolhido (SOMMERVILLE, 2003, p.45). O quarto quadrante trata do planejamento, o projeto é revisto e decide-se pela execução do próximo loop da espiral. Caso se decida avançar para o próximo loop, os planos para a próxima fase do projeto são elaborados (SOMMERVILLE, 2003, p.45).

Apesar da maioria dos projetos de sistemas de software terem aumentado sua produtividade em cerca de 50%, o uso do modelo em espiral em um projeto pode apresentar, pelo menos, três dificuldades. A primeira dificuldade é sua adequação com os contratos de aquisição de software. Assim como os modelos evolucionários, como não há uma especificação completa do software até que este esteja finalizado, fato que dificulta a elaboração de contratos visto que estes

geralmente exigem a especificação do sistema antes da assinatura do contrato (BOEHM, 1988, p.70-71).

A segunda dificuldade é em relação à confiança da análise de riscos, pois esta depende da experiência e conhecimento da equipe de desenvolvedores. Uma equipe de desenvolvedores inexperiente pode não identificar riscos importantes, resultando em um software com uma grande falha. Além disso, não adianta ter uma equipe experiente analisando os riscos se a equipe que irá fazer a implementação for inexperiente, essa dependência das pessoas que compõe o time de desenvolvedores torna necessário ter mão-de-obra bem qualificada (BOEHM, 1988, p.70-71). A terceira dificuldade que pode surgir devido à natureza do próprio modelo em espiral, que necessita aperfeiçoar algumas definições de modo mais detalhado. Por exemplo, a elaboração de indicadores de estado do projeto na espiral, procedimentos que rastreiem a relação custo-progresso. Quanto maior a complexidade do sistema, mais necessário se torna o refinamento do modelo em espiral, criando-se níveis mais detalhados de loops da espiral (BOEHM, 1988, p.71).

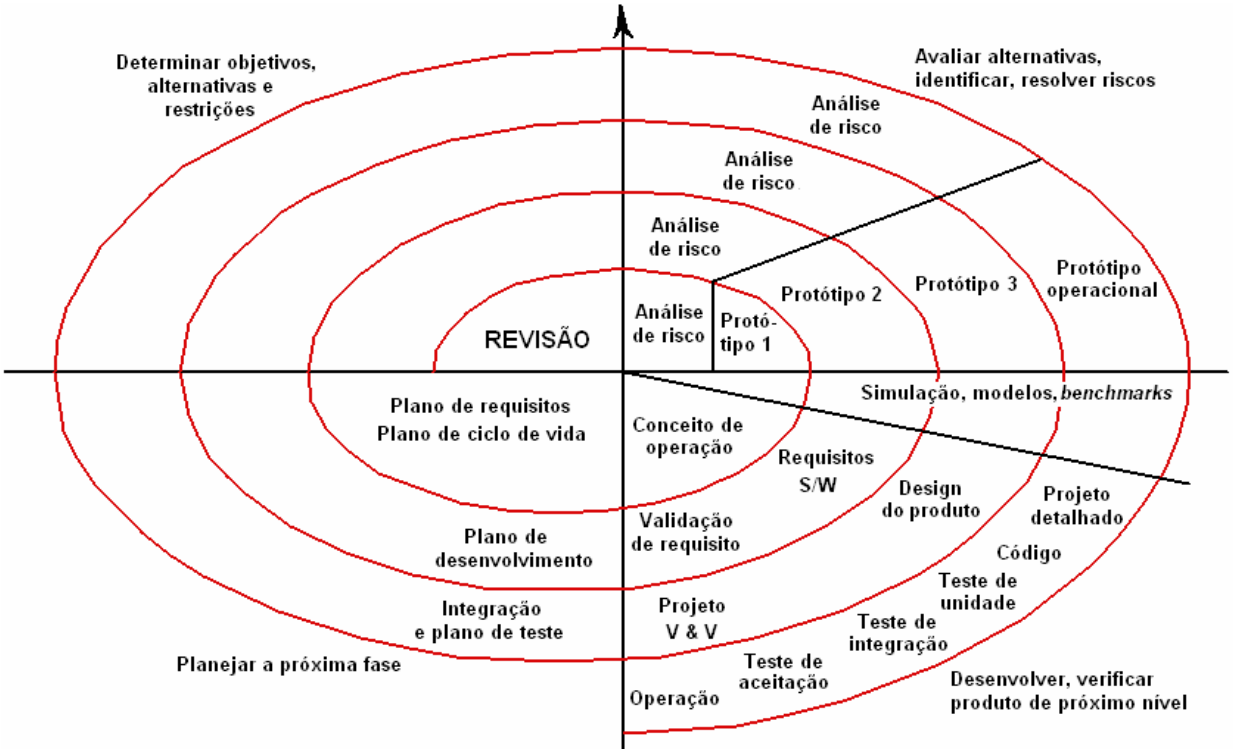

Figura 8 – Exemplo de modelo de processo de desenvolvimento de software em espiral.

Fonte: SOMMERVILLE, 2003, p.45.

2.4.2.1 Desenvolvimento rápido de aplicações

Trata-se de um modelo de processo de desenvolvimento de software incremental, baseado no modelo de desenvolvimento em espiral e no modelo de desenvolvimento evolucionário com uso de protótipos. Também conhecido como *RAD*, este modelo de processo utiliza um ciclo de desenvolvimento muito curto, proporcionando uma entrega rápida do sistema ao cliente que, neste modelo, tem papel ativo no desenvolvimento do software. Não há especificação detalhada do sistema, o projeto e a implementação ocorrem em paralelo baseados no documento de requisitos obtido com o usuário, que define somente as funcionalidades essenciais. O desenvolvimento do software ocorre em diversas etapas utilizando ferramentas CASE e linguagens de quarta geração, além de uma participação ativa de usuários finais, investidores, colaboradores e clientes. Quando é possível, também reutiliza componentes de softwares previamente desenvolvidos. Caso o sistema seja aprovado ele é entregue ao cliente, senão novos protótipos são desenvolvidos e uma nova avaliação do sistema é feita (SOMMERVILLE, 2003, p.148).

O RAD também tem suas restrições. Para grandes projetos é necessário um grande número de recursos humanos para se criar o número necessário de equipes de desenvolvimento. O modelo de desenvolvimento RAD é adequado para aplicações de pequeno e médio porte. RAD também exige comprometimento de desenvolvedores e clientes durante as atividades de desenvolvimento. Se não existir uma iteração dinâmica entre os membros do projeto, a abordagem RAD falhará. Também existem riscos técnicos, como no caso de uma nova tecnologia ser adotada ou quando a interoperabilidade com outros softwares existentes é muito grande. Nestes casos o modelo RAD não é recomendado.

2.5 Contribuições de um processo de software

A criação de processos de desenvolvimento de software (e principalmente a efetiva execução dos processos) demonstrou ser a saída para o caos que vivia o desenvolvimento de software no final da década de 60. Sem dúvida a padronização das atividades, independente do modelo adotado, fez com que o software atingisse

um alto nível de segurança, qualidade e eficiência, proporcionando que todos os profissionais das mais diversas áreas de trabalho aumentassem sua produtividade com a aplicação da computação. No entanto quatro contribuições merecem destaque. A primeira é o desenvolvimento de ferramentas CASE que proporcionaram infra-estrutura e ambiente de desenvolvimento para criação de sistemas de software cada vez mais complexos.

A segunda contribuição importante foi a criação de métodos e técnicas de desenvolvimento, que definiram diretrizes para o uso da tecnologia em prol do desenvolvimento de software. De nada adiantaria dispor da mais avançada tecnologia se esta não fosse corretamente utilizada. A terceira contribuição foi no modo de definir o comportamento das pessoas e da organização dentro de um processo de software. Tanto a organização que desenvolve o projeto como os membros da equipe precisam ser gerenciados e coordenados.

A quarta notória contribuição dos processos de desenvolvimento de sistemas de computador foram as idéias de marketing e economia aplicadas ao desenvolvimento de software. Algumas atividades do processo de software (por exemplo, análise de requisitos e especificação do projeto) consideram, como qualquer outro tipo de produto que precisa ser vendido, tendências de mercado, procurar por nichos de mercado não explorados e realizar algum tipo de marketing para atingir seu foco de consumidores.

Para construção de um modelo de desenvolvimento de software baseado no processo open source se faz necessário analisar os diversos aspectos que envolvem este tipo de projeto. Além de compreender as diferenças entre software open source e software proprietário, o próximo capítulo deste trabalho procura avaliar os processos de desenvolvimento de alguns projetos open source considerados consistentes pela literatura analisada na revisão bibliográfica deste trabalho. O estudo dos modelos de desenvolvimento open source propostos na literatura poderá auxiliar na identificação de contribuições e/ou inovações, bem como possíveis pontos críticos, para a modelagem do processo proposto neste trabalho.

3 Desenvolvimento de software open source

Antes de abordar o processo de software open source, é necessário primeiramente esclarecer alguns conceitos deste universo – principalmente quanto à diferença entre os termos *open source* e *software livre*. Tendo em mente o conceito de open source, alguns projetos considerados de sucesso serão apresentados para compreender por que o uso de software livre está deixando o meio acadêmico e ganhando espaço entre usuários comuns, grandes multinacionais, governos de países e até mesmo órgãos militares. Esse "movimento" pode ser observado na declaração de Lonchamp (2005, p.1-2) onde apresenta a informação de que "Atualmente, Linux e o servidor Apache são utilizados em respectivamente 30% e 60% dos servidores públicos de Internet". Conhecendo os projetos de software open source será possível identificar as atividades que caracterizam este processo de desenvolvimento de software e assim obter a base que será a estrutura do processo de software proposto neste trabalho.

3.1 Características de software open source

Apesar de o termo open source ser novo, compartilhamento e redistribuição de código-fonte são práticas antigas. A disponibilização do código-fonte era uma prática normal e visto com "bons olhos" até o início da década de 80, tanto no meio acadêmico quanto no comercial (o sistema operacional Unix original é um exemplo). O hardware detinha o foco comercial, e não o software, qualquer melhoria no software agradava as grandes companhias, pois tornava o objeto computador mais "atraente". Outro fator relevante para essa aparente passividade dos grandes desenvolvedores de software era a grande variedade de arquiteturas e organização de computadores. Um programa que era desenvolvido para uma arquitetura não

funcionava nas outras, restringindo bastante à cópia dos softwares nesta época. No início da década de 80 as mudanças quanto à licença do software começaram, devido ao surgimento das linguagens de alto nível (possibilitando que um código-fonte fosse compilado em diferentes arquiteturas) e a redução drástica da variedade de arquiteturas existentes (FOGEL, 2005, p.3). As licenças passaram a ser restritivas não permitindo modificação e redistribuição do software, separando o código-fonte do código executável (REIS, 2003, p.16), caracterizando o software *proprietário*. Richard Stallman – que trabalhava no MIT – iniciou um movimento em contrapartida criando em 1985 a Fundação Software Livre (FSF) e iniciou um projeto (GNU) que visava criar um sistema operacional completamente livre e compatível com o Unix.

Há uma diferença entre as denominações *free software* (software livre) e *software open source* (fonte aberto) (Fogel, 2005, p.3-4). O termo *open source* foi criado em 1998 visto que a expressão software livre poderia ser um empecilho para os negócios de TI. A Iniciativa Open Source (OSI) é uma tentativa de atrair empresas e mais investimentos para o desenvolvimento de software open source. Normalmente o termo open source é associado aos dois movimentos, visto que têm os mesmos objetivos e possuem desenvolvimentos de software semelhantes (LONCHAMP, 2005, p.3-4).

Reis (2003, p.14) define software livre como “[...] qualquer software cuja licença garanta ao seu usuário liberdades relacionadas ao uso, alteração e redistribuição. Seu aspecto fundamental é o fato do código-fonte estar livremente disponível para ser lido, estudado ou modificado [...]”. A mesma definição pode ser usada para open source. Contudo quando é feita uma comparação entre as licenças dos softwares livre e open source, há uma diferença visível. A licença de software GNU GPL é a mais importante no universo software livre e possui um caráter não-permissivo: a redistribuição só é permitida se o usuário que receber a cópia também tem liberdade de usar, modificar e redistribuir sua versão (obriga que as cópias modificadas do software também sejam livres, com código fonte aberto). A licença de open source definida pela OSI (http://www.opensource.org/licenses/category) “não pode colocar restrições sobre outro software que é distribuído ao longo da licença de software”, ou seja, é não-restritiva. Permite que a licença de programas maiores que utilizam software (ou trecho de software) open source possam incluir na sua licença algum tipo de restrição (REIS, 2003, p.16).

Neste trabalho foi adotada a nomenclatura open source visto que na literatura encontrada (maioria dos casos) ele foi o termo escolhido para descrever as características do processo de desenvolvimento deste tipo de software. Sendo assim o software open source deve então ter total liberdade de redistribuição do software (gratuita ou não), o código-fonte deve ser disponibilizado (e sua distribuição deve ser permitida) em algum tipo de linguagem de programação, modificações e trabalhos derivados devem ser permitidos e não pode discriminar pessoas ou grupos. Existem outras características que são consideradas específicas de software open source: distribuição do código-fonte e do código executável via Internet; desenvolvimento do software de forma descentralizada; usuários participantes (comunicação intensa entre usuário-desenvolvedor); uso de ferramentas de comunicação (para atender essa intensidade) e interesse pessoal do criador do projeto de software (REIS, 2003, p.15). No entanto, o software open source é uma estrutura mais complexa que compreende além do produto software todo um conjunto de entidades que formam um *projeto* de software open source, abordado a seguir.

3.2 Projetos de software open source

Devido à grande diversidade de projetos de software open source torna-se vital o seu estudo para conseguir estabelecer o processo de desenvolvimento de software open source. Reis (2003, p.15) define um projeto de software open source como sendo um conjunto “[...] composto por pessoas, código-fonte, processo e ferramentas de desenvolvimento”. Três tipos de projetos open source são distinguidos quanto a sua organização. *Meta-projetos* é um conjunto de projetos de software open source com uma estrutura de liderança composta pelos mantenedores de cada um desses projetos. Tem como objetivo organizar os projetos individuais em torno de algum objetivo maior (além de simples desenvolvimento de software), como a padronização de interfaces que o metaprojeto GNOME busca. *Distribuição* é um conjunto de softwares open source que se agregam para formar um produto maior, procurando prover qualidade, facilidade e um acesso facilitado ao software open source. Um exemplo de distribuição é a existência de uma imensa variedade de distribuições de sistemas operacionais GNU/Linux, cada qual possuindo um conjunto diferente de softwares open source de acordo com seu

propósito original (segurança, interface com usuário, maior quantidade de utilitários, gerenciamento de redes, entre outros). A diferença é que no meta-projeto existe uma promoção e auxílio aos projetos que o compões enquanto que na distribuição o relacionamento é apenas de usuário. O terceiro tipo de projeto são os *grupos de usuários*. Como o próprio nome diz, são um conjunto de usuários de software open source que se unem para auxiliar novos usuários a utilizar um metaprojeto ou distribuição, promover o uso de software livre na comunidade local e dar suporte de qualidade. Geralmente concentram os usuários de uma determinada região geográfica (REIS, 2003, p.22-23).

3.2.1 Características de projetos open source

O **código-fonte** é mantido em um repositório público na Internet (página na web ou FTP), podendo ser copiado por qualquer pessoa. O desenvolvimento do código-fonte é geralmente efetuado de forma isolada, cada desenvolvedor trabalha na sua própria cópia local. Ao concluir suas modificações, é feita uma revisão do código, sendo submetido a vários membros do projeto. Se aprovadas, as possíveis alterações podem ser integradas de duas formas. Ou por uma única pessoa, chamada de mantenedor ou “ditador benevolente”; ou por um comitê formado pelos principais desenvolvedores do projeto. Caso não sejam aprovadas as mudanças, o código deve ser reescrito. Este procedimento é realizado para todas as alterações (REIS, 2003, p.23-24).

As **pessoas envolvidas** no projeto podem ser classificadas de quatro formas (REIS, 2003, p.24). Os *usuários não-participantes* apenas utilizam o software algumas vezes, não reportando erros. Os *usuários participantes* utilizam o software além de que contribuem frequentemente com o projeto reportando problemas e com idéias sobre funcionalidades. Outra categoria são os *desenvolvedores esporádicos*, possuem conhecimento de programação e capacidade para fazer alterações no código-fonte. Contribuem resolvendo pequenas falhas ou adicionando pequenas extensões. O último conjunto de participantes são os *desenvolvedores ativos*, responsáveis por módulos do código fonte ou partes mais complexas. Essa divisão não é rígida, os usuários podem mudar de grupo conforme o andamento do projeto ou motivação pessoal (REIS, 2003, p.24).

A **comunicação** entre os participantes do projeto ocorre principalmente nas listas de correio eletrônico (*e-mail*) e programas de comunicação em tempo real (IRC). É possível observar também o crescente uso de *blogs* (espécie de diário virtual), apesar deste tipo de comunicação ocorrer em menor escala. As mensagens normalmente consistem de textos simples, para garantir o fácil acesso a informação e evitar incompatibilidade de algum padrão. Os principais tópicos abordados entre os participantes são para comunicar experiências, problemas e solicitações. Questões sobre implementação e projeto também são discutidas pelos desenvolvedores. Constituem a uma rica fonte de documentação do projeto, visto que ferramentas dão suporte ao armazenamento dessas discussões que posteriormente podem ser encontradas na Internet através de páginas de busca de conteúdo (REIS, 2003, p.25). Para Fogel (2005, p.77) a habilidade de escrever é mais importante do que a capacidade de programar, visto que todas as mensagens são trocadas através de textos por e-mail. Um programador fraco, mas com comunicação excelente, tende a ser mais produtivo do que um ótimo programador, porém com deficiência de comunicação, visto que consegue coordenar e mobilizar muitas pessoas para executarem diversas tarefas.

A questão de **gerência do código-fonte** é muito importante em um projeto caracterizado por um ambiente de desenvolvimento distribuído de software, como os projetos open source. Devido a essa característica existe uma falsa idéia de que a anarquia impera no projeto open source, quando na verdade diversos mecanismos são utilizados para evitar inconsistência do código-fonte e retrabalho da equipe de desenvolvimento. A grande maioria dos projetos utiliza um sistema de controle de versão para incluir, alterar, excluir e gerenciar código-fonte. A ferramenta mais utilizada é o CVS, devido a sua simplicidade e disponibilidade (REIS, 2003, p.26-27). Fogel (2005, p.38) define CVS como “[...] uma combinação de tecnologias e práticas para rastrear e controlar mudanças nos arquivos do projeto, em particular ao código-fonte, documentação e páginas web”. Foi o primeiro sistema de controle de versão open source, atualmente existem outros como o Subversion. O uso do CVS proporciona que os desenvolvedores realizem o trabalho de modo isolado, fazendo suas modificações em uma cópia local do código-fonte do software – também conhecida por *sandbox*. É importante ressaltar que o CVS faz o controle de qualquer arquivo de texto simples, sendo útil também para manter arquivos relativos a documentação do projeto. Existe um conjunto de arquivos constituindo um

repositório, armazenados em um servidor normalmente conectado à Internet. São armazenados os arquivos mais atuais bem como as diferenças entre as versões anteriores. Este repositório de arquivos é mantido intacto, todas as alterações são feitas nas cópias locais, que posteriormente são integradas no repositório (fig. 9).

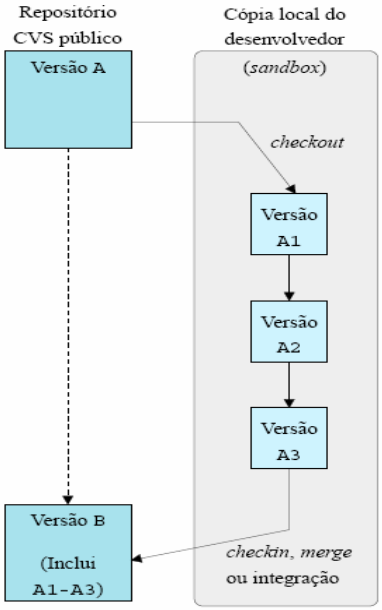

Figura 9 – Diagrama do fluxo de trabalho normal utilizando uma ferramenta para controle de versão do tipo CVS.

Fonte: REIS, 2003, p.27.

A ferramenta CVS também permite que o desenvolvedor atualize sua cópia local com as modificações que vão sendo integradas nos arquivos do repositório, através da operação de *update*. O mantenedor (ou núcleo) responsável pelo código-fonte utiliza o CVS para integrar as mudanças na base do repositório, realizando uma operação de commit ou *checkin*. Se mais de uma pessoa possui autorização para alterar o código-fonte do repositório, pode ocorrer um **conflito** na hora do commit. Isto ocorre caso mais de um desenvolvedor alterar o mesmo trecho de código. Neste caso, o primeiro desenvolvedor a integrar o trecho de código desconsidera o trabalho dos outros desenvolvedores, que recebem uma notificação para atualizarem a sua base de código. Ao atualizar a base outra mensagem alerta que o mesmo trecho de código foi alterado. Uma revisão cuidadosa deve ser feita para avaliar qual solução deve permanecer (o recomendado é manter as duas alterações em um trecho único). Cada arquivo alterado recebe um número de

versão, permitindo assim que alterações nos arquivos do repositório sejam desfeitas (REIS, 2003, p.28). *Branches* (bifurcações) são linhas paralelas de desenvolvimento que o CVS permite criar a partir do código principal (tronco), sendo particularmente útil quando se deseja aplicar mudanças de grande impacto na base do código. As alterações efetuadas no branch não refletem no código principal, porém podem ser integradas e caracterizar uma operação denominada *merge*. Podem ser criadas diversas bifurcações e cada versão do branch terá seu próprio código.

As **ferramentas** de apoio ao desenvolvimento de software open source devem ter interoperabilidade, respeito às características individuais dos desenvolvedores e suporte ao modelo *offline* de desenvolvimento. Dividem-se em ferramentas de *desenvolvimento* que suportam formatos padrões (editores de texto vi e Emacs são os mais utilizados), ferramentas de *configuração e compilação* para tornar simples o procedimento de compilar um novo software (make, autoconf, autoheader e automake são um exemplo), ferramentas de *visualização de código* para fazer comparações de arquivos (CVSWeb, LXR e Bonsai representam essa classe) e ferramentas para *acompanhamento de defeitos* (Bugzilla).

Esse conjunto de características é muito bem representado quando pensamos em projetos como Linux, GNOME, Apache e Mozilla, onde é encontrado um grande número de pessoas envolvidas do projeto (ao ponto de surgir verdadeiras comunidades), desenvolvimento do código-fonte é distribuído e um conjunto de ferramentas para dar suporte ao desenvolvimento (REIS, 2003, p.21). Ao contrário do que a maioria das pessoas possa imaginar, nem todos os projetos software livre são assim. Um estudo de Krishnamurthy (2002) avaliou os 100 projetos open source mais ativos da página web Sourceforge.net[1] e foi constatado que grande parte dos projetos é desenvolvida por apenas um indivíduo (em média a equipe é composta por quatro desenvolvedores), produz pouca atividade nas suas listas de e-mail e os projetos mais antigos possuíam maior número de desenvolvedores. Os esforços deste trabalho serão concentrados em torno de projetos de software open source com nível de maturidade avançado e que são influências para projetos menores.

[1] Serviço de hospedagem de projetos open source, fornece espaço para armazenamento, repositório CVS com controle de acesso, ferramenta para acompanhamento de defeitos e lista de discussões própria.

3.3 Modelos de processo de desenvolvimento open source

A modelagem de um processo de desenvolvimento de software open source é muito difícil. Além da grande diversidade de projetos (cada um tendo suas particularidades de desenvolvimento), a falta de documentação explícita sobre o processo por parte dos próprios projetos colabora para que interessados no assunto desistam de estudá-lo. A maioria dos estudos que concernem o desenvolvimento de software open source apresenta uma descrição narrativa do processo de desenvolvimento, não podendo ser analisada, comparada ou reaproveitada em outros projetos. Desse modo, o desenvolvedor que deseja integrar um projeto open source é obrigado a recorrer aos repositórios de informação pública na Internet. Por isso a descrição, compartilhamento, modificação e distribuição de modelos detalhados é uma contribuição importante para a difusão e aprimoramento desse tipo de processo (LONCHAMP, 2005, p.2). Dentre vários projetos existentes, destacamos quatro estudos que são correntemente citados na literatura sobre processo de desenvolvimento de software livre.

3.3.1 A catedral e o bazar

Considerado por muitos autores como o primeiro trabalho a tentar descrever o processo de desenvolvimento de um projeto open source, A Catedral e o Bazar consiste de uma narrativa de Eric Raymond sobre o processo de desenvolvimento do kernel Linux e do projeto Fetchmail. Apesar de não constituir uma modelagem do processo de desenvolvimento, este trabalho é muito importante por fornecer práticas comuns no desenvolvimento de software open source. Essas práticas serão muito importantes para detalharmos o processo de desenvolvimento proposto. Raymond define dois estilos de desenvolvimento de software.

3.3.1.1 Os estilos de desenvolvimento

O estilo *Catedral* é referente ao desenvolvimento de software proprietário e até mesmo de alguns projetos de software livre. O projeto desenvolvido pelo estilo

Catedral é individualmente construído por mágicos hábeis ou por um bando de magos trabalhando no mais esplêndido isolamento, sem nenhuma versão beta para ser lançada antes do tempo. O outro estilo de desenvolvimento é o *Bazar*, referente aos projetos open source como Linux. Este estilo é caracterizado por lançamentos prévios e freqüentes, delegar o máximo possível de tarefas e estar aberto ao ponto da promiscuidade, assemelhando-se com um barulhento bazar. Esse ambiente de desenvolvimento de software era contrário à experiência profissional de Raymond, que decidiu fazer um teste: desenvolver um projeto de software open source no estilo bazar, o fetchmail. Ao longo do desenvolvimento do projeto, Raymond produziu algumas "máximas" sobre esse estilo de desenvolver software que caracterizam o processo de desenvolvimento open source, sendo que algumas podem ser utilizadas como práticas de desenvolvimento (RAYMOND, 2001, p.21-22).

3.3.1.2 O desenvolvedor e o código

A primeira observação que Raymond faz é que *todo bom trabalho de software inicia coçando uma ferida do desenvolvedor*, ou seja, o software nasce de uma necessidade do desenvolvedor – sendo essa uma característica predominante nos projetos open source. Como não encontrava um bom cliente de mensagem eletrônica, Raymond resolveu pesquisar pelos programas existentes para encontrar um que estivesse o mais próximo das suas necessidades, para assim adicionar as funcionalidades que ele necessitava. Segunda observação: *bons programadores sabem o que escrever, os grandes sabem o que reescrever (e reusar)*. Como exemplo ele cita Linus Torvalds, que no início do desenvolvimento do linux reutilizou idéias e códigos do Minix, pequeno sistema operacional do tipo Unix. Outra lição compartilhada por Raymond e que demonstra o espírito de troca de códigos-fonte do mundo open source é que *quando você perde o interesse em um programa, seu último dever é entregá-lo para um sucessor competente*.

3.3.1.3 A Lei de Linus

Uma das “máximas” mais importantes de Raymond e que é de fundamental importância no desenvolvimento open source é *tratar seus usuários como desenvolvedores é sua rota menos atribulada para um aperfeiçoamento rápido do código e uma depuração eficaz*. Tanto é importante que Raymond julga o modelo de desenvolvimento do linux uma contribuição maior do que o próprio kernel. Com o auxílio da internet e com uma política de reconhecimento do trabalho de seus desenvolvedores, Linus revolucionou ao disponibilizar várias versões do kernel em curtos espaços de tempo (às vezes até mesmo uma versão por dia!). Ou como diz Raymond, *libere cedo, libere frequentemente e escute os seus clientes*. Isso fez Raymond perceber que Linus pretendia ter o máximo de pessoas trabalhando o máximo do tempo no projeto. Desta forma, *dado uma base larga o suficiente de beta-testadores e de co-desenvolvedores, quase todos os problemas serão caracterizados rapidamente e a solução será óbvia para alguém* (RAYMOND, 2001, p.23-30). Raymond denominou de “A Lei de Linus” e na sua visão esta é a principal diferença entre os estilos bazar e catedral. No estilo catedral de programação, erros e problemas de desenvolvimento são complicados, traiçoeiros. São meses de pesquisa de poucos desenvolvedores dedicados para ter confiança de que todos os problemas foram resolvidos. Assim, longos intervalos de lançamento e a certeza da decepção de uma versão tão esperada não ser perfeita. No entanto o estilo bazar aceita que os erros e problemas são fúteis, ou pelo menos se tornam fúteis quando expostos a um grande número de co-desenvolvedores que analisam detalhadamente cada novo lançamento (RAYMOND, 2001, p.31).

3.3.1.4 Restrições

Existem algumas restrições na implementação do estilo bazar. Não se pode iniciar um projeto “a partir do zero”, algum tipo de software tem que ser oferecido para os usuários experimentarem mesmo que seja grosseiro ou com poucas funcionalidades. Outro requisito é quanto às características do coordenador do projeto. Conhecimentos de programação, boa capacidade de gerenciamento e principalmente de comunicação com sua base de co-desenvolvedores são elementos essenciais para o sucesso de um projeto no estilo bazar.

3.3.2 Processo de desenvolvimento do Apache e do Mozilla

Audris Mockus e James Herbsleb, enquanto membros do Production Research Department at Lucent Technologies' Bell Laboratories, realizaram uma pesquisa juntamente com Roy Fielding para avaliar se o processo de desenvolvimento open source poderia competir com sucesso contra os métodos de desenvolvimento de software comercial. Para tal, eles escolheram dois grandes projetos de software open source, o servidor de páginas na internet Apache e o navegador de páginas para Internet Mozzilla, e quantificaram aspectos dos seus processos de desenvolvimento como participação dos desenvolvedores, tamanho do núcleo da equipe de desenvolvimento, propriedade do código fonte, produtividade, densidade de defeitos e intervalo de tempo para resolução de problemas.

A metodologia utilizada consiste em uma análise qualitativa junto da equipe de desenvolvedores e uma análise quantitativa dos dados encontrados nos repositórios de informações dos projetos (lista de mensagens eletrônicas, log[2] das ferramentas de controle de versão e da ferramenta de relatório de erros).

Considerando características típicas de um processo de desenvolvimento open source é possível observar que geralmente uma única pessoa ou um núcleo que seleciona quais alterações podem ser incorporadas ao código base do projeto. Os softwares open source são desenvolvidos por uma grande maioria de voluntários, que escolhem qual trabalho desejam ficar encarregados. Além disso, não há um design em nível de sistema explícito ou mesmo detalhado, também não existe planejamento, plano de trabalho ou lista de entregas.

Mockus observou que estes elementos configuram um ambiente de distribuído de desenvolvimento, com os desenvolvedores espalhados em diferentes locais, nunca se encontrando face a face e tendo suas atividades coordenadas quase que exclusivamente através da internet, ao invés dos mecanismos tradicionais de comunicação. Também é possível perceber que o código dos desenvolvedores de software open source é produzido com mais atenção e criatividade visto que eles estão trabalhando em software que realmente tem interesse e prazer, resultando em softwares de excelente qualidade e funcionalidade (MOCKUS, FIELDING e HERBSLEB, 2002, p.263).

[2] Tipo de arquivo que registra os eventos e operações de um determinado sistema.

3.3.2.1 Desenvolvimento no projeto Apache

Na análise qualitativa do projeto Apache, foi solicitado aos membros do núcleo de desenvolvimento que escrevessem um rascunho descrevendo o processo de desenvolvimento até obterem a descrição completa e deste documento foram extraídas algumas conclusões. O projeto Apache iniciou em 1995, em uma tentativa de coordenar as propostas de correções do programa *httpd* da NCSA, desenvolvido por Rob McCool. O núcleo de pessoas responsáveis por "guiar" o desenvolvimento do projeto se chama Apache Group, um comitê informal constituído inteiramente de voluntários que não dedicam todo o seu tempo para o desenvolvimento do projeto, necessitando assim de um processo de tomada de decisão descentralizado e com comunicação assíncrona. O grupo implementou seu sistema de comunicação exclusivamente através de listas de mensagens eletrônicas e um sistema de votação com quorum mínimo para resolução de discordâncias. Qualquer membro do grupo pode votar pela inclusão de alterações no código. No início do projeto eram oito membros, durante a análise do processo eram em torno de doze pessoas. Não há um processo de desenvolvimento estabelecido, os desenvolvedores executam um conjunto de tarefas comum a todos os membros: checar se existem problemas, determinação se um voluntário trabalhará nesse problema, identificação de soluções, desenvolver e testar código em sua cópia local do código-fonte, submeter às mudanças no código ao Apache Group para revisão e atualizar o código-fonte base com as alterações e documentação relacionada. Pode haver diversas iterações entre as tarefas até que a atualização do código seja efetivada (MOCKUS, FIELDING e HERBSLEB, 2002, p.265-266).

A coleta de erros e falhas pode ocorrer na lista de mensagens eletrônicas dos desenvolvedores, através do sistema de aviso de problemas BUGDB e no grupo de discussões sobre o Apache na USENET – sistema de troca de mensagens através de fóruns de discussão categorizados por assunto. Os erros reportados na lista de mensagens eletrônicas dos desenvolvedores têm alta prioridade e recebem a atenção de todos os desenvolvedores ativos. O BUGDB permite que qualquer pessoa reporte erros e falhas através da internet ou de correio eletrônico, mas seu manuseio é considerado complicado e por isso não é muito utilizado pelos usuários

e desenvolvedores, poucos se comprometem a fazer verificações periódicas em busca de avisos de problemas neste sistema. Assim que um problema é descoberto, procura-se por um voluntário para resolvê-lo; os desenvolvedores tendem a trabalhar em trechos do código-fonte em que estão mais familiarizados (MOCKUS, FIELDING e HERBSLEB, 2002, p.266-267).

Para manter um histórico do desenvolvimento do projeto, um arquivo é armazenado em cada repositório contendo uma lista com os problemas com maior prioridade, as questões ainda não resolvidas e os planos de liberação de versões do software. Existe uma propriedade do código implícita visto que os desenvolvedores são responsáveis por partes do código que tenham criado ou que já mantêm há algum tempo, porém essa condição não dá direitos especiais em relação ao controle das alterações do código-fonte base (MOCKUS, FIELDING e HERBSLEB, 2002, p.266-267).

Após a definição de quem trabalhará em qual problema, a próxima etapa é decidir por uma dentre as diversas soluções propostas analisando quesitos como desempenho, portabilidade e complexidade. As soluções mais adequadas são compartilhadas com os demais membros do projeto através da lista de mensagens eletrônicas. Definida a solução, ela é implementada e testada pelo desenvolvedor na sua cópia local do código-fonte, e então ela é ou incorporada diretamente no código-fonte do Apache ou é criado um pacote de atualização para a correção do erro, que é submetido para a lista de mensagens eletrônicas dos desenvolvedores para revisão (MOCKUS, FIELDING e HERBSLEB, 2002, p.266-267).

Como qualquer pessoa pode se inscrever na lista, muitos “olhos” estão revisando o código e retornando contribuições para o pacote – um exemplo da Lei de Linus. Assim que o pacote for aprovado, qualquer desenvolvedor do núcleo pode atualizar o código-fonte. Quando uma versão do software está próxima de ser liberada, um dos desenvolvedores do núcleo é voluntário para ser o gerente do lançamento da versão e fica responsável por identificar problemas críticos que possam impedir o lançamento, determinar quando esses problemas foram reparados e a versão se tornou estável, além de controlar o acesso ao repositório do código-fonte base. O papel de gerenciador de lançamento é alternado entre os membros do núcleo com maior experiência no projeto (MOCKUS, FIELDING e HERBSLEB, 2002, p.266-267).

Para a análise quantitativa do processo de desenvolvimento do projeto Apache foram utilizadas três fontes de dados. A *lista de mensagens eletrônicas dos desenvolvedores*, na qual qualquer pessoa pode participar e as mensagens abordam os mais diversos assuntos como discussões técnicas, propostas de alterações, notificações sobre alterações no código e alertas sobre problemas encontrados. Foram utilizados scripts na linguagem Pearl para extrair data, remetente, assunto e conteúdo das mensagens, a inspeção manual foi utilizada para casos em que a técnica falhou (por exemplo, múltiplos endereços eletrônicos). A segunda fonte utilizada foram os arquivos da ferramenta de controle de versões concorrentes *CVS*. Cada alteração no código gera automaticamente uma mensagem (que é arquivada no servidor do CVS) eletrônica contendo a data e hora da modificação, o nome de identificação do desenvolvedor, os arquivos modificados, número de linhas adicionadas e excluídas de cada arquivo, e um resumo explicando a modificação. A terceira fonte de dados usada foi a base de dados da ferramenta BUGDB, utilizada para reportar problemas. Para cada operação realizada, uma mensagem é automaticamente gerada e armazenada, sendo que os autores extraíram dessas mensagens o código do relatório do problema, o módulo afetado, a atual situação do problema (se está em aberto, sendo analisado, resolvido, etc.), o nome do autor da notificação, a data e o comentário (MOCKUS, FIELDING e HERBSLEB, 2002, p.265).

Para analisar quantitativamente o processo, os autores compararam os dados obtidos com os dados de vários projetos comerciais. Estes projetos não eram exatamente iguais ao projeto Apache, porém possuem características muito semelhantes. Os resultados dessa comparação sugeriram aos autores várias conclusões importantes sobre o processo de desenvolvimento do Apache e sobre o produto final. A abrangência da comunidade do projeto é realmente enorme, visto que aproximadamente quatrocentas pessoas contribuíram com código que foi realmente incorporado. No entanto os quinze principais desenvolvedores do projeto foram responsáveis por 83% das modificações requisitadas, 88% das linhas de código adicionadas e por 91% das linhas de código excluídas. Isso demonstra que o trabalho dos desenvolvedores do núcleo do projeto é muito maior, em termos de produção de código, do que o trabalho dos não-membros do núcleo do projeto. Por outro lado, ao analisar as notificações de problemas, a participação dos não-membros do núcleo com código para correções de problemas é mais significativa do

que em relação ao desenvolvimento de novas funcionalidades. Os desenvolvedores do núcleo do projeto são responsáveis por 66% das correções. Assim pode-se concluir que os desenvolvedores do núcleo são os principais responsáveis pelo desenvolvimento e manutenção de novas funcionalidades do projeto. Quanto à regularidade das contribuições dos desenvolvedores do núcleo, os dados mostraram que uma pequena parcela envia contribuições para o projeto regularmente. Em projetos comerciais a contribuições dos desenvolvedores também variam, no entanto não são tão maiores do que as variações encontradas no projeto Apache. As taxas de produtividade do Apache se mostraram em mesmo nível do que as taxas dos desenvolvedores de projetos comerciais (MOCKUS, FIELDING e HERBSLEB, 2002, p.267-269).

Em projetos comerciais os defeitos são geralmente relatados pelas equipes de codificação, de teste e de manutenção de software. Os dados coletados no projeto Apache demonstraram que grande parte dos erros que afetam os usuários finais foram notificados através do sistema BUGDB. É evidente que a operação de teste de software é em grande parte exercida pelos membros da comunidade do projeto. Devido à característica informal do desenvolvimento do Apache, acaba sendo implementado um sistema de “propriedade de código”, a fim de coordenar as alterações e evitar/resolver possíveis conflitos. Também é possível verificar que apesar desta responsabilidade não há impedimentos para que um coordenador de um módulo contribua com código para diferentes módulos (MOCKUS, FIELDING e HERBSLEB, 2002, p.269).

. Para avaliar a densidade de defeitos no código, a medida de *defeitos pós-lançamento por mil linhas de código* mostrou-se ineficiente podendo induzir a resultados errôneos. Por exemplo, se em uma pequena modificação de um módulo considerar todas as linhas ao invés de somente as adicionadas, a taxa de defeitos será reduzida. Para contornar esse problema, foram utilizadas duas medidas diferentes e os resultados obtidos foram comparados com as mesmas medidas de projetos de software comercial. Tanto se considerarmos a densidade de defeito pós-lançamento como a densidade pré-lançamento (versões beta), as taxas do Apache se mostraram menores do que as taxas de software comercial. Os autores atribuem a esse resultado ao fato de que ou poucos defeitos são produzidos pelos desenvolvedores do projeto, ou a atividade de reportar defeitos é maior ou então a base de conhecimento dos desenvolvedores sobre o domínio da aplicação – afinal

também são usuários do software – resulta em uma taxa reduzida. O intervalo de resolução dos defeitos reportados varia de acordo com a prioridade que é atribuída ao problema. Os problemas que afetam os usuários são prioridade no projeto Apache. Quanto maior o número de usuários atingidos (ou quanto maior o número de falhas que podem ocorrer devido ao problema), mais alta é a prioridade. Problemas que envolvem sistemas operacionais muito específicos ou componentes adicionais possuem uma prioridade menor. Do ponto de vista do usuário essa abordagem é muito vantajosa, pois ele quer que os problemas sejam resolvidos o mais rápido possível. Problemas com a documentação do software não de extrema urgência, embora também sejam muito importantes (MOCKUS, FIELDING e HERBSLEB, 2002, p.269-271).

3.3.2.2 Desenvolvimento no projeto Mozilla

A mesma metodologia foi aplicada ao projeto Mozilla pelos autores. No entanto os resultados foram ligeiramente diferentes, devido que a origem do projeto Mozilla ser comercial. Em 1998 a empresa Netscape liberou o código-fonte do seu navegador de Internet (*Navigator*), ficando a cargo do grupo mozilla.org o papel de "ditador benevolente" tornando-se assim o ponto de contato. O projeto Mozilla possui um trabalho mais diversificado em relação ao Apache, apoiando várias tecnologias e ferramentas de desenvolvimento que não são ligadas diretamente com o navegador. A ferramenta Bugzilla é usada para submissão e controle de erros, o CVS para controle de versão, o Bonsai (junto com o Tinderbox) facilitam o acesso ao repositório CVS e as mudanças de código em tempo real, e o LXR transforma os arquivos do código-fonte em páginas HTML (REIS, FORTES, 2002, p.12-7). O início do desenvolvimento foi conturbado devido à falta de documentação e de suporte por parte da Netscape, afastando assim interesse da comunidade e de alguns desenvolvedores inclusive. Após um esforço para produzir documentação e tutoriais, detalhando as ferramentas de desenvolvimento e o próprio processo, o projeto evoluiu atraindo vários desenvolvedores (Mockus et al., 2002, p.331). Atualmente o navegador Mozilla Firefox obteve um sucesso considerável, segundo dados da

empresa NetApplications[3], especializada no desenvolvimento de ferramentas para monitoração e medição de tráfego em páginas na internet, no ano de 2004 o navegador Microsoft Internet Explorer detinha 91,35% do mercado enquanto que o navegador Mozilla Firefox aparecia com apenas 3,66%. No entanto esse quadro mudou. De acordo com a pesquisa da NetApplications, considerando os dados até o mês de maio de 2007, o Microsoft Internet Explorer detém 78,67% do mercado, enquanto que o Mozilla Firefox detém 14,54%.

Assim como o Apache, o grupo mozilla.org é constituído de uma equipe central (em 2002 eram doze pessoas) que coordenam e guiam o projeto, definindo processos e escrevendo código. Apenas cerca de um terço da equipe central escreve código para o navegador de internet, os demais desempenham outras funções na comunidade como assegurar qualidade do software, liberação de versões, manutenção da página na internet e de ferramentas de apoio ao desenvolvimento. A autoridade para tomar decisões sobre determinados módulos é repassada para determinados indivíduos da comunidade que possuam bastante conhecimento sobre o código-fonte do módulo em questão. No entanto a decisão final ainda cabe à equipe central, que se resguarda o direito de alterar o responsável por algum módulo como também resolver possíveis conflitos. A equipe central, juntamente com todos os membros da comunidade do projeto, mantém um documento contendo diversas informações do projeto como especificações do que deverá ser incluído em versões futuras e datas agendadas para liberação dessas versões. Através da ferramenta Bugzilla, qualquer pessoa pode reportar problemas, solicitar melhoramentos do software e enviar correções de problemas. Também é mantida uma lista de todos os problemas, corrigidos ou não, e de quem está trabalhando em cada um, evitando trabalhos duplicados. Através desse sistema os membros da comunidade podem escolher em qual frente de trabalho incorporar, de acordo com seu nível de conhecimento do domínio do problema e também de acordo com o tipo de trabalho que gosta de fazer. As listas de discussão completam esse sistema, tornando-se um importante canal de comunicação entre os membros da comunidade para a troca de idéias. Antes da liberação da versão estável, no mínimo um “teste fumaça” (identificação e resolução de erros simples, a fim de reduzir problemas de integração) é realizado diariamente em uma versão, compilada

[3] Página <http://marketshare.hitslink.com/report.aspx?qprid=0> visitada em 17/07/2007.

em diversas plataformas para assegurar que a versão é suficientemente estável e o trabalho de desenvolvimento possa prosseguir. Caso erros sejam detectados pelos testes fumaça, eles são divulgados diariamente para que os desenvolvedores estejam alertas quanto a problemas críticos na versão. O projeto Mozilla conta com diversas equipes de teste responsáveis por avaliar aspectos do software em relação ao comprometimento com padrões, clientes de mensagem eletrônica e de notícias, e internacionalização. A equipe de testes mantém também testes de caso, plano de testes e outros materiais como um guia de passos para verificação de defeitos e um guia para resolução de problemas. A revisão do código é executada em duas etapas. Primeiro o responsável pelo módulo avalia as atualizações propostas no escopo do módulo. Depois alguns membros da equipe central, com profundo conhecimento técnico, avaliam as atualizações em relação com o código base. Para o gerenciamento de liberação de versões utilizam a ferramenta Tinderbox, que demonstra quais partes do código foram usadas em quais versões e para qual plataforma. Também destaca qual código foi alterado e por quem. A decisão sobre quando uma versão pode ser liberada oficialmente é humana e não um processo automatizado (Mockus et al., 2002, p.332-333).

Os resultados qualitativos, obtidos utilizando-se a mesma metodologia do projeto Apache, demonstram que o projeto Mozilla é muito maior do que o Apache sendo constituído por 78 módulos[4] - alguns deles maiores do que o projeto Apache inteiro. O autor escolheu então sete módulos para realizar a análise. A partir de mensagens e códigos armazenados no sistema CVS, o autor observou que os participantes da comunidade tendem a contribuir com menos mudanças e menor número de linhas de código-fonte no projeto. Historicamente o Mozilla teve mais equipes principais de desenvolvimento do que participantes externos, apresentando números muito próximos dos projetos comerciais (Mockus et al., 2002, p.333-335).

Diferentemente do Apache, a equipe central de desenvolvimento é quem reporta maior parte dos problemas encontrados na fase de testes. Outra característica particular do projeto Mozilla em relação ao Apache é que a maior parte dos membros do núcleo de desenvolvimento dedica tempo integral ao projeto, obtendo assim níveis de produtividade semelhantes ao de projetos comerciais. (Mockus et al., 2002, p.336). Mockus et al. (2003, p.337), na análise dos resultados

[4] Dado obtido na época em que o autor realizou o estudo.

obtidos, chega a afirmar que “os dados sugerem que a produtividade neste particular projeto híbrido é comparável a ou melhor do que os projetos comerciais examinados”.

A propriedade do código se faz necessária, cada mantenedor de um módulo é responsável pelos problemas notificados, aperfeiçoamento de requisições, submissão de pacotes de correções (*patch*) e revisar o código antes de inserir na base do CVS. O “proprietário” do módulo deve atuar como um facilitador do bom desenvolvimento do software (Mockus et al., 2003, p.337).

Na análise da quantidade de defeitos após a liberação da versão, o projeto Mozilla obteve resultados muito parecidos com os do projeto Apache, sendo levemente menor a taxa de defeitos do Mozilla. Em relação a software comercial, o módulo do Mozilla com as mais altas taxas de defeitos, ainda assim, estava abaixo das taxas de defeitos dos projetos comerciais analisados. Isso pode ser devido a baixa taxa de defeitos no código-fonte. O tempo médio de resolução destes defeitos foi maior do que no projeto Apache por causa das inspeções de código obrigatórias que ocorrem no projeto Mozilla, todavia é importante ressaltar que há uma relação entre a prioridade da resolução do defeito e o tempo que leva para ser resolvido o problema. Os problemas com prioridade elevada foram resolvidos no máximo em trinta dias – dependendo do módulo analisado. Quanto menor a prioridade maior o tempo de correção do defeito (Mockus et al., 2003, p.337-p.339).

Mockus et al. termina concluindo em sua pesquisa que é tentador hibridizar processos de desenvolvimento de software comerciais com processos open source. Uma diferença importante de se observar entre processos de desenvolvimento de software comerciais e processos open source é na forma de coordenação, seleção e atribuição do trabalho. Neste caso é possível notar diferenças mesmo entre o projeto Apache e o projeto Mozilla – devido a sua origem comercial. O esquema de coordenação do projeto Apache aparenta ser adequado para projetos pequenos, o próprio servidor Apache é um projeto pequeno visto que todas as demais funcionalidades formam um conjunto de vários projetos auxiliares. Desta forma o Apache Group apenas precisa saber quem está trabalhando em quê e manter um canal de comunicação. Somando o fato de que todos os membros do núcleo de desenvolvimento têm acesso ao código-fonte base dispensando esperas por aprovações de um único indivíduo, o projeto consegue grandes benefícios com produtividade e qualidade.

No Mozilla, em contrapartida, os módulos são menos independentes do que os projetos auxiliares do Apache. O Mozilla é um projeto muito grande para apenas uma pequena equipe de desenvolvimento, necessitando de mecanismos mais formais de coordenação do trabalho para que a interdependência dos módulos não afete o desenvolvimento. No caso particular do navegador Mozilla, a demanda do mercado influência muito o desenvolvimento do software (tanto comercial como open source) gerando maior pressão sobre a equipe em termos de inovação e datas de liberação de novas versões. Esses fatores podem levar a uma maior complexidade do código e menor modularização, prejudicando a coordenação do trabalho. Para evitar esses problemas é sugerido que novos projetos híbridos sejam baseados em pequenas equipes desenvolvendo módulos bem definidos, além de reuso de código – tanto open source como comercial. O autor também observa que seria interessante o experimento de um projeto comercial utilizar a coordenação e delegação do trabalho no mesmo estilo dos projetos Apache e Mozilla (Mockus et al., 2003, p.342-p.344).

Mockus et al. (2003, p.344) finaliza dizendo que espera que pesquisas futuras possam demonstrar “novas abordagens que elegantemente combinariam as melhores tecnologias de todos os tipos de ambientes de desenvolvimento de software”.

3.3.3 Modelo de Reis

Esse modelo propõe estabelecer o modo como o software é construído nos projetos open source levando em consideração as atividades fundamentais do processo de software. Baseado em um estudo da bibliografia existente sobre o desenvolvimento de software open source – bem como sobre a morfologia dos seus projetos – Reis criou e submeteu um questionário para desenvolvedores de diversos projetos hospedados no servidor sourceforge.net além de ter participado ativamente nos projetos open source Mozilla, Bugzilla, PyGTK e Kiwi. No projeto Mozilla Reis desempenhou as atividades de teste funcional, testes de aceitação pública (*smoketests*), além de notificação, discussão e triagem de informes de erro. No projeto Bugzilla tornou-se desenvolvedor pleno, contribuindo com submissão e revisão de código. Nos projetos PyGTK e Kiwi Reis desenvolveu as atividades de

desenvolvimento de código e documentação, sendo que o projeto Kiwi é de sua própria iniciativa (REIS, 2003, p. 91-93). Essa participação ativa é uma característica muito importante do seu trabalho, pois poucos estudos sobre o processo de desenvolvimento de software open source foram realizados por participantes ativos dos projetos analisados. Desta forma Reis procurou estabelecer de que forma os projetos de software open source trabalham, se existe um processo comum entre os projetos e se existem diferenças entre este processo e os processos de desenvolvimento tradicionais da engenharia de software (REIS, 2003, p. 57-61).

Reis (2003, p. 66-69) propôs algumas hipóteses, baseadas nos trabalhos anteriormente realizados e na sua experiência como participante ativo de projetos open source, para confrontar com os dados obtidos através da aplicação do questionário. Foi tomado o cuidado para que o questionário fosse enviado para desenvolvedores com bom conhecimento do fluxo de trabalho e da equipe de desenvolvimento, sempre dando preferência ao mantenedor do projeto. A forma de aplicação do questionário foi pela internet, através de um sistema web desenvolvido por Reis na linguagem PHP.

Os projetos analisados foram selecionados através dos sites sourceforge.net e freshmeat.net, duas das principais páginas da internet especializadas em armazenar projetos open source. Reis buscou projetos com longevidade, intensa atividade ou publicamente conhecidos. Essas características levam a acreditar que os projetos possuem um processo de desenvolvimento consistente com grande participação da comunidade, inclusive os próprios sites sourceforge e freshmeat atribuem índices relativos à estabilidade e maturidade do projeto (variam de 1 a 6, sendo que o valor 5 corresponde a projetos estáveis e o valor 6 a projetos maduros), porém estes índices não possuem relação com os critérios estabelecidos pelo Capability Maturity Model (CMM[5]). Desta forma foram escolhidos 600 projetos do freshmeat.net e 200 projetos do sourceforge.net. O autor ainda acrescentou os 50 projetos mais populares dos grupos GNOME, KDE e GTK+, além de outro conjunto de projetos formado pelos projetos recorrentes na literatura e outros com participação ativa do autor.

Após a exclusão de projetos repetidos (um projeto pode estar hospedado nos dois sites simultaneamente), chegou-se ao total de 1102 projetos os quais o

[5] Regras adotadas pelo Software Engineering Institute para medir a maturidade do processo de desenvolvimento de software em uma empresa. Atualmente é denominado CMMI <http://www.sei.cmu.edu/cmmi/>

autor obteve o endereço eletrônico do mantenedor e entrou em contato por e-mail solicitando a participação na pesquisa, obtendo resposta de 570 projetos (REIS, 2003, p. 74-75).

O autor também desenvolveu uma ferramenta – chamada *process_tarball* – em linguagem de programação *Python*, para obter pacotes com os códigos-fonte e assim poder realizar medições com o auxílio da ferramenta *sloccount* desenvolvida por David Wheeler (REIS, 2003, p. 84-87).

Após filtrar os questionários incompletos, os que apresentavam respostas inconsistentes e os questionários relativos à meta-projetos e distribuições (que podem gerar inconsistência nos dados), Reis obteve 519 respostas válidas. É importante ressaltar que dentre estas respostas válidas encontramos projetos de renome como o kernel Linux, Mozilla, Perl, Tcl e MySQL, lembrando que os questionários foram enviados por e-mail para os próprios criadores/mantenedores dos projetos (REIS, 2003, p. 97-98).

Os questionários mostraram que 71% dos entrevistados iniciaram o projeto por motivação pessoal, corroborando a idéia lançada por Eric Raymond e apresentada na seção **3.3.1**. Apenas 13% dos projetos iniciaram com apoio ou através de uma organização e meros 6% dos projetos eram software proprietário e se tornaram open source (REIS, 2003, p. 105). Quanto ao perfil dos principais usuários na utilização do software, apesar de 75% terem indicado o próprio autor ou pessoas com elevado conhecimento técnico, 21% declarou as opções “Cliente” ou “Organização Específica”, fato que levou Reis a perceber uma aceitação comercial do software open source. Este fato também sugere a potencial utilização do processo de desenvolvimento de software open source para fins comerciais ou específicos. Para determinação da “idade” do projeto foi adotada a data de liberação pública da primeira versão do software. Neste ponto os questionários demonstraram que 40% dos projetos analisados possuem entre dois e cinco anos, contudo não representam o tempo total de desenvolvimento (REIS, 2003, p. 107).

Um dado importante para este trabalho é o resultado em relação ao tamanho da equipe do projeto. Desconsiderando o fato de que nos projetos open source é difícil determinar *quem* faz parte da equipe de desenvolvimento devido à volatilidade dos seus participantes (tanto o mantenedor como a comunidade pode perder o interesse no projeto), cabe aqui apresentar os dados obtidos. Dentre os questionários válidos 42% dos projetos apresentam entre dois a cinco membros na

equipe e em 32% dos projetos a equipe constitui uma única pessoa, no entanto é importante destacar que entre os dez maiores projetos (considerando número de linhas de código) apenas três possuem menos de 25 pessoas na equipe de desenvolvimento. Essas equipes utilizam o modelo de liderança “ditador benevolente” (como o Linux) e o modelo de comitê (como o Apache) na mesma proporção. Reis acredita que o modelo ditador é uma evolução natural dos projetos de software livre, enquanto que o modelo comitê exige um maior esforço para manter a comunicação entre os membros e a coordenação do trabalho. Poucos projetos indicaram não haver estrutura de liderança (apenas 3%) o que desmistifica o rótulo de ambiente caótico e informal de desenvolvimento.

Outros aspectos em relação a equipe de desenvolvimento também são pertinentes e merecem atenção. Em relação à distribuição geográfica, 64% dos projetos possuem equipes em que a maioria dos membros se conhecem apenas através da internet e nunca se encontraram pessoalmente, o que ratifica a idéia de um ambiente distribuído de desenvolvimento. Outro ponto interessante é que em 40% dos projetos um contribuidor geralmente participa de mais de uma atividade no processo de desenvolvimento e em média 55% das equipes possuem membros com mais de cinco anos de experiência em desenvolvimento de software. (REIS, 2003, p. 109-113).

No que concerne a definição de requisitos, 36% dos projetos replicam a funcionalidade de outro software e 32% dos projetos possuem seus requisitos definidos total ou parcialmente por um padrão prévio. Apesar de o autor justificar a ausência de um processo de requisitos com estes dados, fica claro a lacuna existente nessa atividade do processo de software e a possibilidade potencial de hibridização do processo. Outro dado importante sobre a definição de requisitos é que 40% dos mantenedores de projetos open source afirmam que as funcionalidades do software são implementadas de acordo com o que a equipe pensa ser correto, demonstrando certa independência das equipes (REIS, 2003, p. 114).

Em relação à documentação, 70% dos projetos afirmaram ter documentos que descrevem ao menos as funcionalidades e comportamentos esperados no software produzido. O autor, baseado na sua experiência como participante ativo de projetos open source, lembra que pouca documentação formal é encontrada. Outro fator interessante é a grande quantidade de comentários que existe no código-fonte

do software open source, seguindo sua filosofia de que com o código aberto é possível as pessoas aprenderem e alterarem o software de acordo com suas necessidades (REIS, 2003, p. 118-119).

A seção de perguntas relativas a testes apresentou dados inesperados. Somente 27% dos projetos afirmaram utilizar um conjunto de testes automatizados para validar o software e apenas 10% afirmaram existir um plano de testes (documento escrito) para o software desenvolvido. Contraditoriamente, 56% dos projetos afirmaram existir uma tendência/política de somente liberar uma versão ao público quando ela tiver sido extensivamente testada pela equipe.

O autor credita estes resultados ao fato de a maioria dos projetos considerarem os usuários como a equipe de testadores beta, dando muita importância para os testes funcionais realizados nas versões alfa e beta de seu software. Outro ponto relevante em relação a testes é que apenas 23% dos projetos afirmaram possuir um processo de revisão de código, porém Reis acredita que tal número se deve ao fato que em uma grande quantidade de projetos a equipe de desenvolvimento é formada por apenas uma pessoa. O autor afirma que em contrapartida os grandes projetos (como GCC, Apache e Mozilla) utilizam algum tipo de processo de revisão de código (REIS, 2003, p. 120-122).

O questionário também procurava elucidar os tipos de ferramentas usados para auxiliar o desenvolvimento de software open source. O gráfico com os resultados obtidos com o questionário ilustra bem os tipos de ferramentas e a quantidade de projetos que as utilizam (Fig. 10).

O autor observa que realmente o processo de desenvolvimento open source utiliza um conjunto de ferramentas simples para implementar a comunicação assíncrona, o controle de versão do software, a coordenação do trabalho, bem como a divulgação e disponibilização do software produzido.

Reis também realizou alguns agrupamentos e correlações dos resultados com intuito de obter mais características do processo de desenvolvimento open source, sendo mais relevante para este trabalho a relação entre esforço de engenharia de software e número de linhas de código (*sloc*). Se práticas de engenharia de software forem identificadas, é possível formular um processo.

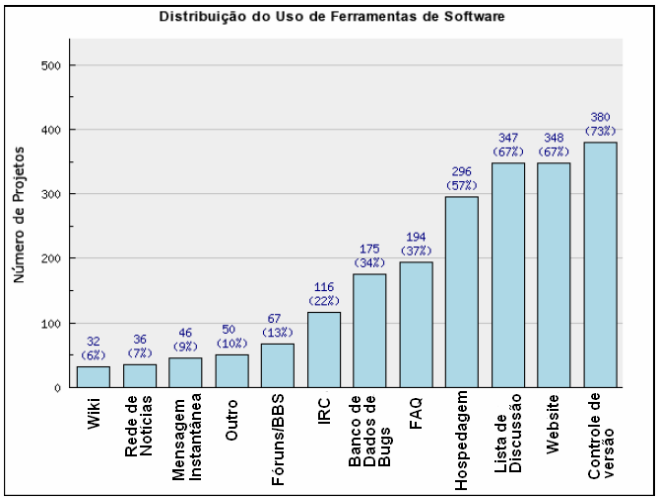

Figura 10 – Utilização de ferramentas no desenvolvimento de software open source.

Fonte: REIS, 2003, p. 124.

A Fig. 11 demonstra que apesar da distribuição esparsa dos pontos comprovar a diversidade de processos de desenvolvimento, a concentração de pontos indica um conjunto comum de atividades que possibilitam formular um processo de desenvolvimento aplicável em boa parte dos projetos (REIS, 2003, p. 129-130). Após a análise dos resultados obtidos, Reis propõe um ciclo de vida resumido para projetos open source.

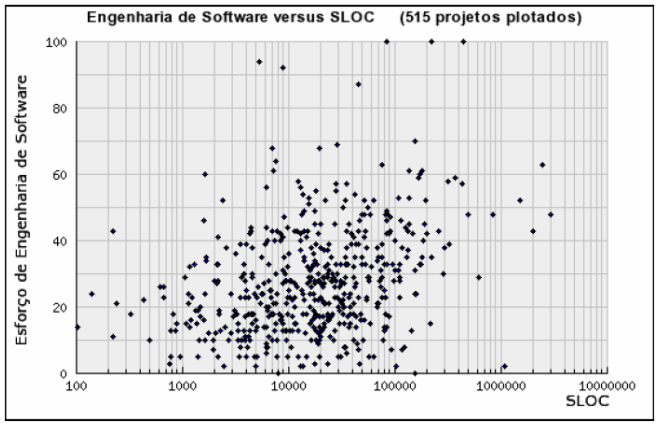

Figura 11 – Scatter relacionando esforço de engenharia de software com número de linhas de código de cada projeto.

Fonte: REIS, 2003, p. 130.

Este processo de desenvolvimento caracteriza-se por ter maior ênfase nas atividades de operação/manutenção do software, as atividades podem ser a qualquer instante (depende da motivação dos desenvolvedores) e o software produzido está em constante evolução. É importante ressaltar que os mesmos mecanismos utilizados para a comunicação entre os membros da equipe de desenvolvimento também é utilizado para estabelecer um canal de comunicação com os usuários do software, onde diversos temas relativos ao projeto são debatidos, como novas funcionalidades, alterações, notificação de defeitos, suporte, entre outros.

A Fig. 12 ilustra a comunicação dentro de um projeto e a forma como o software é disponibilizado pela equipe de desenvolvimento. Observe que além de desenvolver e liberar versões do software cabe a equipe de desenvolvimento se relacionar com os usuários para manter o interesse e a motivação dos mesmos no projeto. Essa relação é essencial para manter o feedback e atender solicitações ou reparos no software de modo rápido e seguro (REIS, 2003, p. 141-142).

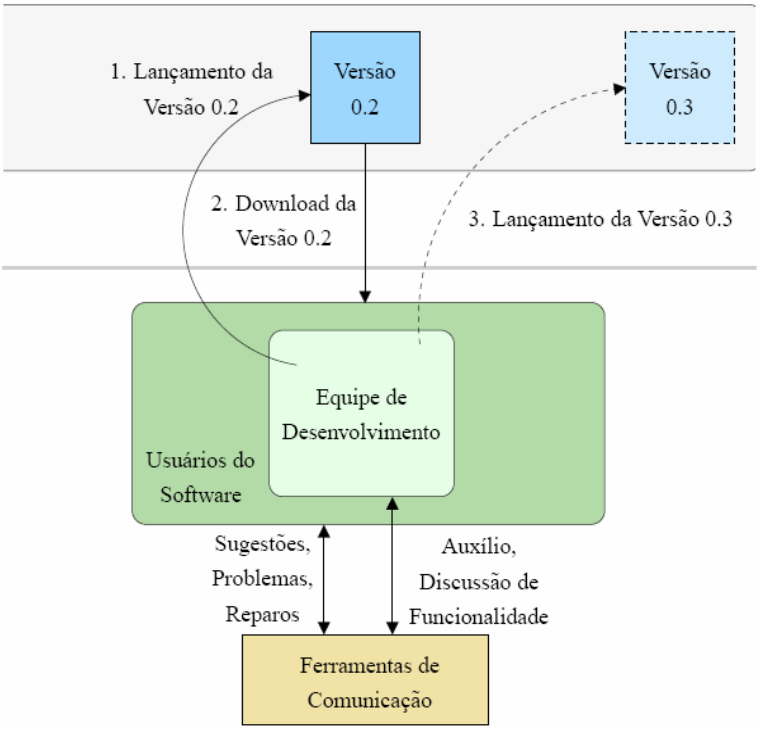

Figura 12 – Descrição de um projeto open source.

Fonte: REIS, 2003, p. 142.

No desenvolvimento de código nos projetos open source cada usuário é um desenvolvedor em potencial visto que tendo acesso ao código fonte ele pode

contribuir com reparos ou novas funcionalidades, caracterizando assim um ambiente com desenvolvimento concorrente. No entanto as alterações no código-fonte base são restritas ao mantenedor do projeto ou ao comitê responsável. Apesar de o trabalho simultâneo poder gerar algum retrabalho proporciona a possibilidade de comparar mais de uma solução para um problema - duas cabeças pensam melhor do que uma. A Fig. 13 mostra o modelo de processo de desenvolvimento de software open source que foi proposto por Reis, cobrindo as principais características identificadas em projetos de maior porte e que tornam o processo open source peculiar: o software, a equipe de desenvolvimento e as ferramentas de apoio ao processo (principalmente comunicação e desenvolvimento) (REIS, 2003, p. 141-145).

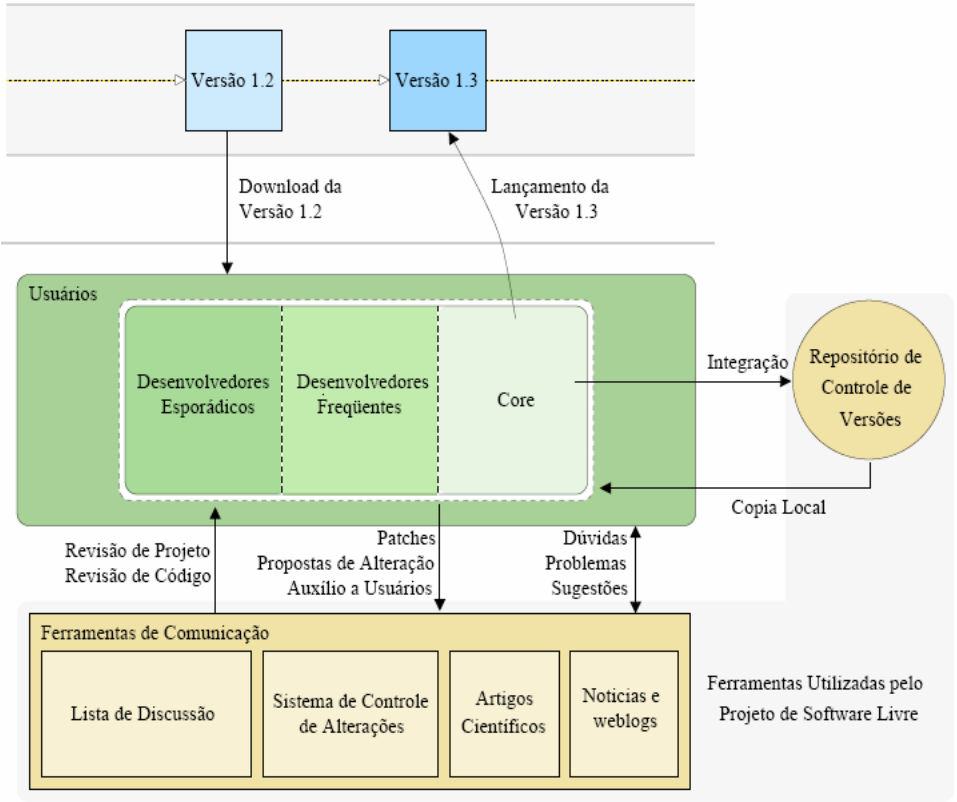

Figura 13 – Principais elementos do processo de desenvolvimento de um projeto open source consolidado.

Fonte: REIS, 2003, p. 145.

Obviamente esse modelo não é limitado, podendo projetos maiores utilizar-se de outros recursos em termos de ferramentas e comunicação com os usuários.

Apesar de REIS (2003, p.146) caracterizá-lo como “[...] um resumo bastante simplificado do processo [...]”, é de extrema valia para este trabalho no tocante da descrição da dinâmica dos membros da equipe de desenvolvimento e de suas atividades, além de demonstrar a ausência de artefatos formalizados nas atividades de requisitos e planejamento. O autor também ressalta os aspectos humanos envolvidos nos projetos open source, principalmente na motivação que conduz uma grande maioria de voluntários a se tornarem desenvolvedores, e na estrita relação usuário-desenvolvedor que existe nas comunidades open source personificada na ação dos beta-testadores do software (REIS, 2003, p.146-148).

3.3.4 Modelagem de Lonchamp

Jacques Lonchamp propõe um modelo de processo de desenvolvimento de software open source (OSSDP) com três camadas. Uma camada *genérica* e uma camada de *definição* são especificadas para descrever as características comuns a todos os projetos open source “iniciantes”. Uma camada em nível *específico* é estabelecida a fim de permitir a descrição de particularidades (características específicas) encontradas nos processos de desenvolvimento de diferentes projetos open source (LONCHAMP, 2005, p.1). Para fazer a modelagem do processo de desenvolvimento Lonchamp utilizou a simbologia do metamodelo SPEM para ilustrar os componentes do processo de desenvolvimento open source (papéis, atividades, artefatos...) aliada a diagramas de casos de uso, diagramas de seqüência e diagramas de atividades para representar as iterações entre as etapas do processo de desenvolvimento de software open source. É importante para a compreensão da importância dessa modelagem abordar o metamodelo SPEM.

3.3.4.1 SPEM

É um metamodelo para processo de engenharia de software – SPEM – desenvolvido a partir de 1999 e lançado no ano de 2005 (versão 1.1) pela Object Management Group (OMG), organização internacional fundada em 1989 para incentivar o uso do paradigma da orientação a objetos, tanto em termos de pesquisa como também comercialmente, sendo responsável pela especificação padrão da

UML e da tecnologia CORBA[6]. Dentre as várias empresas que apóiam o SPEM estão IBM, Fujitsu, Rational Software, Siemens e Alcatel. O metamodelo utiliza a notação da UML para representar a arquitetura e modelagem do software, sendo todo estruturado na UML 1.4. O SPEM deve ser empregado para descrever um processo de desenvolvimento de software consolidado (ou uma família de processos de desenvolvimento de software associados) utilizando uma abordagem orientada a objetos, fugindo do seu escopo a forma como é executado o processo.

A segunda versão da especificação já engloba o comportamento do processo e apesar de ter sido pré-disponibilizada para download na página da Internet da OMG, ainda existem questões e correções a serem solucionadas pelo grupo quanto ao comportamento do processo, falta de algumas associações e extensões dos estereótipos dos componentes[7]. O lançamento oficial da versão 2.0 está previsto para outubro de 2008 e por isso este trabalho foi desenvolvido na especificação da versão 1.1.

O SPEM procura definir um conjunto de elementos suficiente para que qualquer processo de desenvolvimento possa ser instanciado na sua especificação. A Fig. 14 ilustra a arquitetura de modelagem de quatro camadas que foi definida, sendo que a camada mais baixa representa o processo como realmente é executado na prática e cada nível superior representa uma abstração do processo.

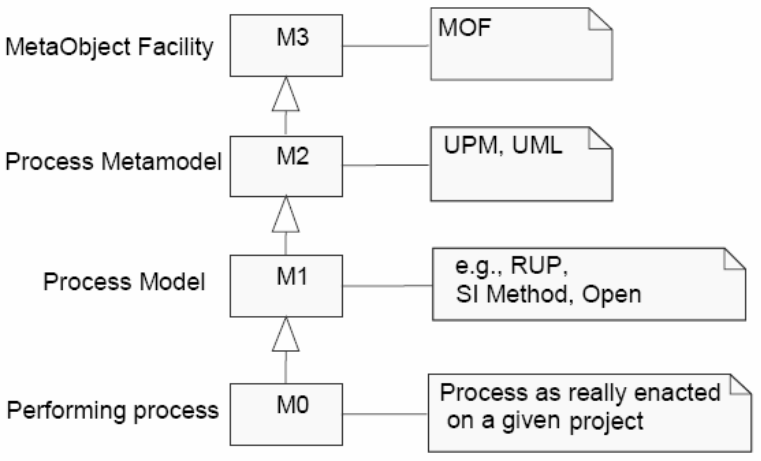

Figura 14 – Níveis de modelagem de processo.

Fonte: SPEM, 2005.

Observe que os metamodelos de processo de desenvolvimento situam-se na camada M2 e os processos de desenvolvimento como XP, RUP e modelo cascata

[6] Arquitetura criada pela OMG para troca de dados entre sistemas distribuídos diferentes.
[7] Os problemas não resolvidos podem ser acompanhados em http://www.omg.org/issues/spem2-ftf.open.html

situam-se ma camada M1. Na camada M3 situa-se as facilidades de metaobjetos (MOF), tecnologia criada pela OMG para a definição e representação de metadados como objetos CORBA (SPEM, 2005). Um dos benefícios de utilizar SPEM é a facilidade de modelar o processo e a rapidez em efetuar alterações no processo de desenvolvimento (MÄKILÄ, T.; JÄRVI, A., 2006, p.93).

A idéia central do SPEM é que o processo de desenvolvimento de software consiste em uma colaboração entre entidades abstratas ativas (*Process Roles*) que executam operações (*Activities*) em entidades tangíveis (*Work Products*).

Um processo (*Process*) é a descrição completa de um processo de engenharia de software, sendo que uma disciplina (*Discipline*) é organizada a partir da visão de uma das disciplinas da engenharia de software: especificação de software, desenvolvimento de software, teste e evolução. Cada elemento modelo (*Model Element*) descreve um aspecto da engenharia de software. Um conjunto de papéis (*Roles*) interage através de associações e colaborações, permutando produtos de trabalhos e acionando a execução de determinadas operações (SPEM, 2005). Alguns modelos de processos denominam *work products* como artefatos de trabalho, podendo ser um documento, ou um código-fonte ou uma modelagem, e podem ser produzidos, utilizados ou modificados por um processo (Fig. 15).

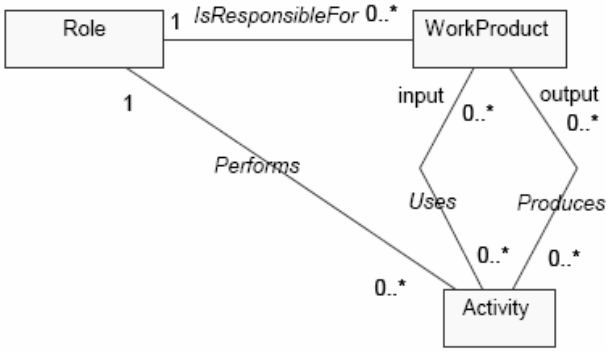

Figura 15 – Um modelo conceitual simplificado.

Fonte: SPEM, 2005.

Uma descrição externa em linguagem natural (*ExternalDescription*) pode ser associada aos elementos do modelo para melhor compreensão dos membros da equipe de desenvolvimento que participam do processo. Também são utilizados guias (G*uidance*) com disposições para suporte aos desenvolvedores. Técnicas, lista de tarefas a serem executadas, regras e recomendações (*guidelines*) e manuais das

ferramentas utilizadas no processo (*ToolMentor*) são alguns exemplos de guias (SPEM, 2005).

Seis tipos de dependências (*Dependencies*) foram especificadas no SPEM: classificação, impacto, importação, precedência, referência e *trace*. Elas constituem relações entre os elementos modelos, por exemplo, uma dependência do tipo precedência entre duas atividades pode indicar relações início-início, fim-início ou fim-fim. *Work definition* é um modelo elemento que descreve a execução, as operações realizadas e as transformações que são efetuadas nos produtos de trabalho pelos Process Roles, sendo exemplos de work definition fase, iteração (*Iteration*), atividade e ciclo de vida. Uma fase (*Phase*) é uma definição de trabalho em alto nível limitada por um marco. Um *Process Performer* é um elemento modelo que descreve o "proprietário" de um work definition. Deve ser utilizado com work definitions que não podem ser associadas com um papel do processo, tal como um ciclo de vida e uma fase. As atividades são o principal elemento de trabalho, sendo decompostas em um conjunto de passos (Step). O passo é um elemento modelo atômico e bem detalhado. A Fig. 16 demonstra a estrutura do ciclo de vida (Lifecycle), que é um work definition associado a um processo e contém todo o trabalho a ser feito em um processo de engenharia de software (SPEM, 2005).

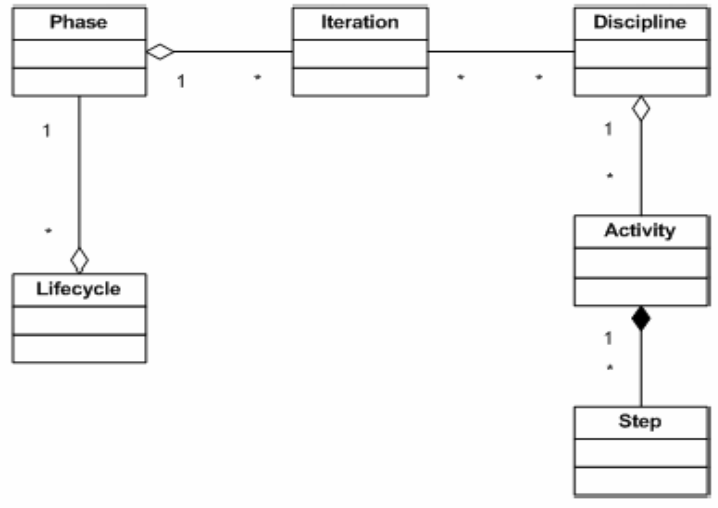

Figura 16 – Organização estrutural da classe do ciclo de vida (*Lifecycle*) do SPEM.

Fonte: MARTINS et al., 2004, p.182.

Na Fig. 17 é possível visualizar a estrutura de um processo no SPEM. O uso deste metamodelo para instanciar o processo de desenvolvimento apresenta

diversas vantagens. Para Lonchamp (2005, p. 7) “[...] o SPEM tem os benefícios dos diagramas UML para apresentar diferentes perspectivas de um modelo de processo de software [...]”. Para BÉZIVIN e BRETON (2004, p.32) “Usando SPEM como plataforma permite o reuso de transformações com diferentes cenários”.

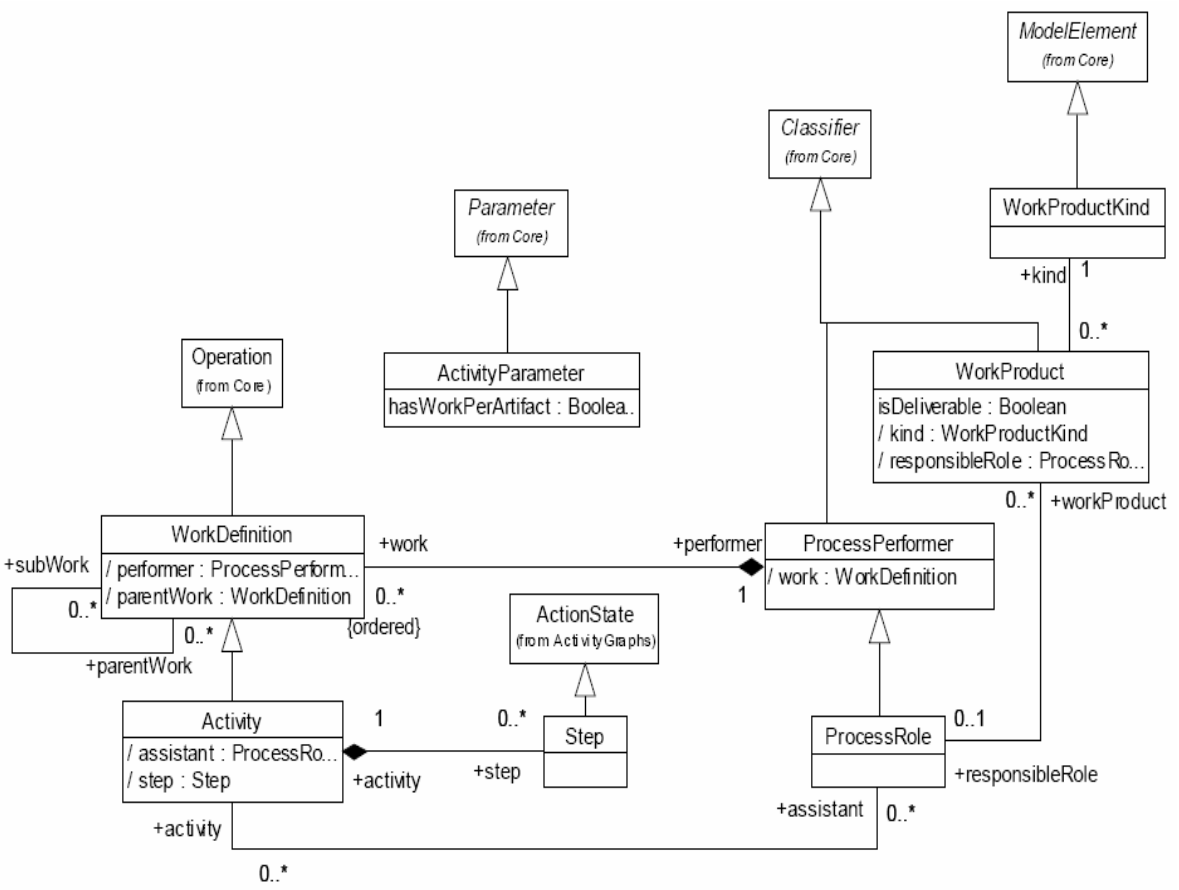

Figura 17 – Pacote de Estrutura do Processo (Process_Structure_Package) no SPEM.

Fonte: SPEM, 2005.

O SPEM faz uso de estereótipos (*stereotypes*) não só para representar suas classes na UML, mas também para introduzir novos símbolos gráficos (Fig. 18). Aproveitando-se de ter sua estrutura montada em cima da UML, o metamodelo SPEM utiliza uma notação própria e permite o uso de indicações visuais nos seus modelos em uma linguagem própria do seu domínio de aplicação. Dessa forma torna-se muito claro a diferença entre seus diagramas e os diagramas da UML, a qual sem o uso de estereótipos não seria possível.

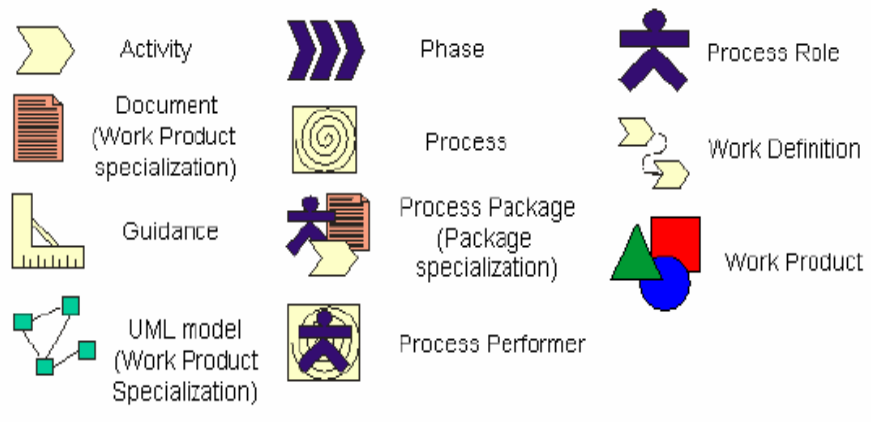

Figura 18 – Principais estereótipos do SPEM.

Fonte: LONCHAMP, 2005, p.7.

3.3.4.2 Modelo OSSDP

Lonchamp primeiramente propôs uma modelagem em nível de definição e em nível genérico, abrangendo assim tanto projetos open source pequenos como os grandes projetos que já atingiram um grau de maturidade elevado. O autor ainda definiu uma camada específica, que é bem detalhada e utilizada para modelar detalhadamente fragmentos característicos de determinados projetos open source. O nível de definição é o mais abstrato, definindo apenas duas disciplinas: o processo de desenvolvimento do software e o processo realizado na comunidade do projeto (Fig. 19). O autor reconhece que o modelo implica em existir uma grande comunidade participando do processo, no entanto sua abordagem é voltada para projetos maduros (LONCHAMP, 2005, p.8).

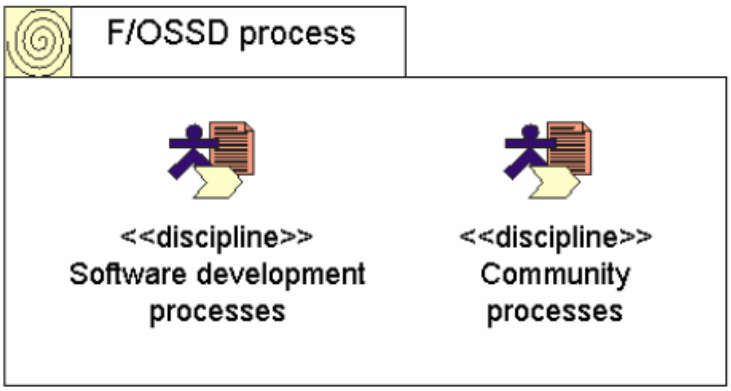

Figura 19 – Nível de definição do OSSDP.

Fonte: Fonte: LONCHAMP, 2005, p.8.

O nível genérico foi modelado através de uma composição de diversos estudos sobre projetos open source e também através de uma sondagem realizada em páginas na internet especializadas em hospedar projetos open source (como o sourceforge, comentado na seção anterior). Para contemplar esta visão genérica o autor modelou as duas disciplinas do nível de definição e construiu o diagrama de casos de uso para as duas disciplinas.

A Fig. 20 demonstra a organização da disciplina de processo de desenvolvimento de software. Esse modelo proporciona a identificação de características do processo open source que já foram abordadas neste capítulo, mas algumas merecem destaque. A primeira observação de destaque é a participação do usuário no desenvolvimento do software, contribuindo diretamente com o processo através de ações que em outros modelos de desenvolvimento geralmente são executadas na fase de manutenção e/ou operação.

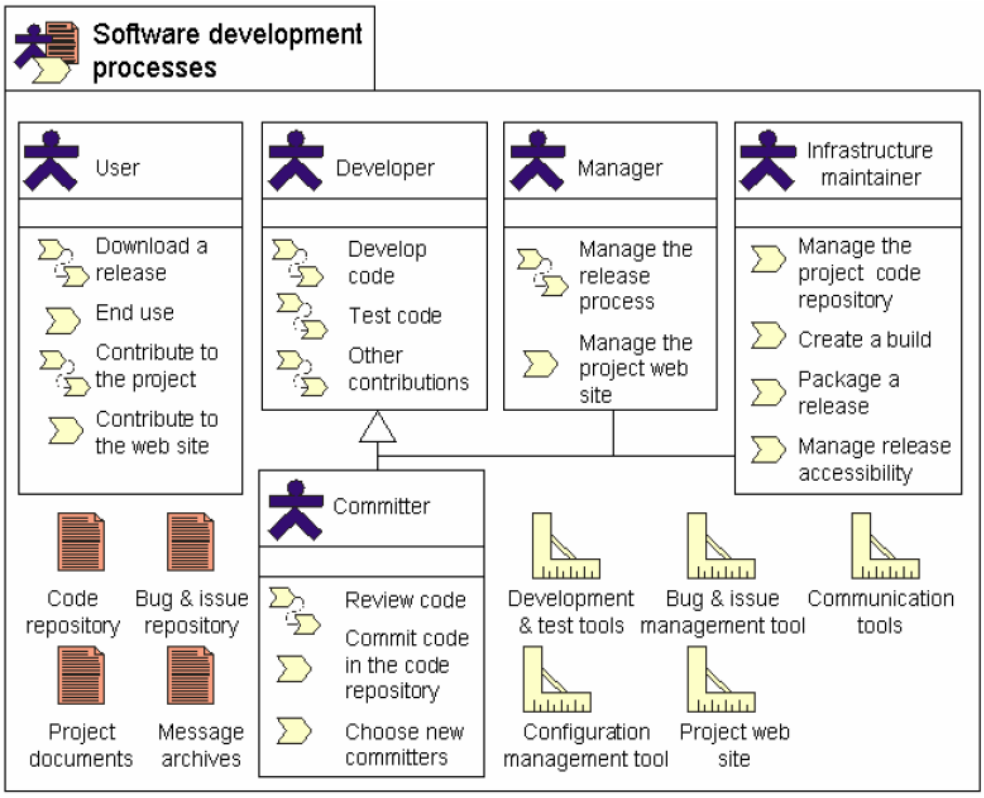

Figura 20 – Disciplina de Desenvolvimento de Software

Fonte: LONCHAMP, 2005, p. 9.

Também é percebida a ausência de um artefato de trabalho com uma especificação mais formalizada, é possível apenas identificar um obscuro “Project

documents” além de alguma informação armazenada nos arquivos de mensagens e que provavelmente deve estar em linguagem natural.

O terceiro ponto interessante desse modelo é que todos os guias representam ferramentas e políticas que direta ou indiretamente auxiliam a coordenação do trabalho e a comunicação entre os membros do projeto e com os usuários. Outro fato peculiar desse processo é a ausência de fases distintas, visto que os participantes do projeto trabalham nas mais variadas tarefas concorrentemente de acordo com o seu interesse pessoal. A Fig. 21 apresenta a disciplina de desenvolvimento da comunidade, que abrange as atividades da comunidade em relação à divulgação do projeto, gerenciamento de subprojetos e manutenção da página na internet para disponibilizar arquivos do projeto.

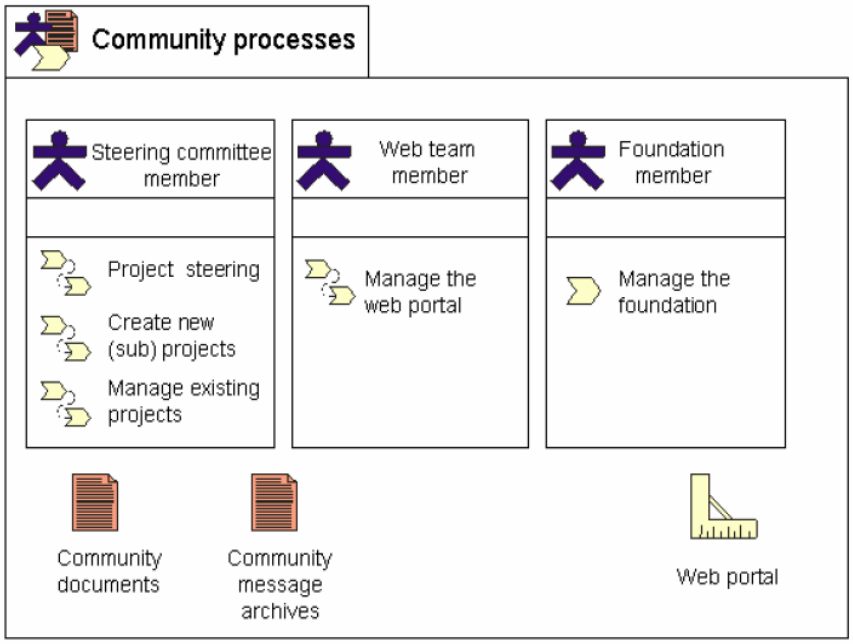

Figura 21 – Disciplina de Desenvolvimento da Comunidade

Fonte: LONCHAMP, 2005, p. 9.

Cada *work definition* das disciplinas representadas nas figuras 20 e 21 constituem um conjunto de atividades que são executadas por determinados papéis do processo (*Process Role*) que também podem auxiliar outros papéis na execução de alguma atividade. Os diagramas de casos de uso foram divididos em quatro partes. O primeiro diagrama de caso de uso é orientado ao usuário focando duas ações: a obtenção de uma cópia da versão a partir da internet e sua contribuição para o projeto (Fig. 22).

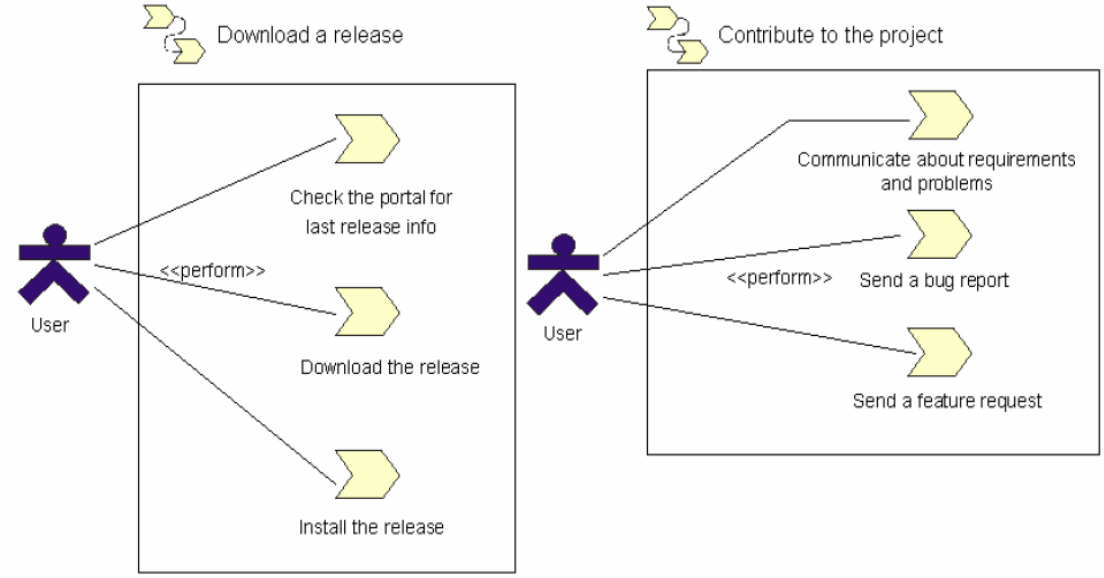

Figura 22 – Casos de uso orientado ao Usuário.

Fonte: LONCHAMP, 2005, p.11.

Lonchamp ressalta que este diagrama demonstra a importância dos usuários no OSSDP através da notificação de defeitos, da sugestão de novas funcionalidades ao software e participando de debates sobre pontos importantes nos canais de comunicação do projeto (LONCHAMP, 2005, p.11).

O diagrama de casos de uso para os desenvolvedores do projeto reflete uma realidade já comprovada na dissertação de Reis e apresentada na seção anterior: na maioria dos projetos open source a equipe de desenvolvimento é pequena, com experiência na área de desenvolvimento e desenvolve a maior parte das novas funcionalidades do software. Os membros dessa equipe foram denominados Committers, pois possuem permissão para alterar o código fonte do repositório. Outra grande parcela de contribuidores (denominados Developers) notifica defeitos encontrados no software através da lista de discussão ou de uma ferramenta para reportar erros.

O diagrama de casos de uso dos Desenvolvedores e dos Committers, bem como o diagrama apresentado anteriormente, demonstra a falta de um processo formalizado de análise e especificação de requisitos, contudo como os desenvolvedores também são usuários – com grande conhecimento no domínio da aplicação – a qualidade do software não é prejudicada. O autor também ressalta a ausência de atividades de design no diagrama de casos de uso dos desenvolvedores (Fig. 23).

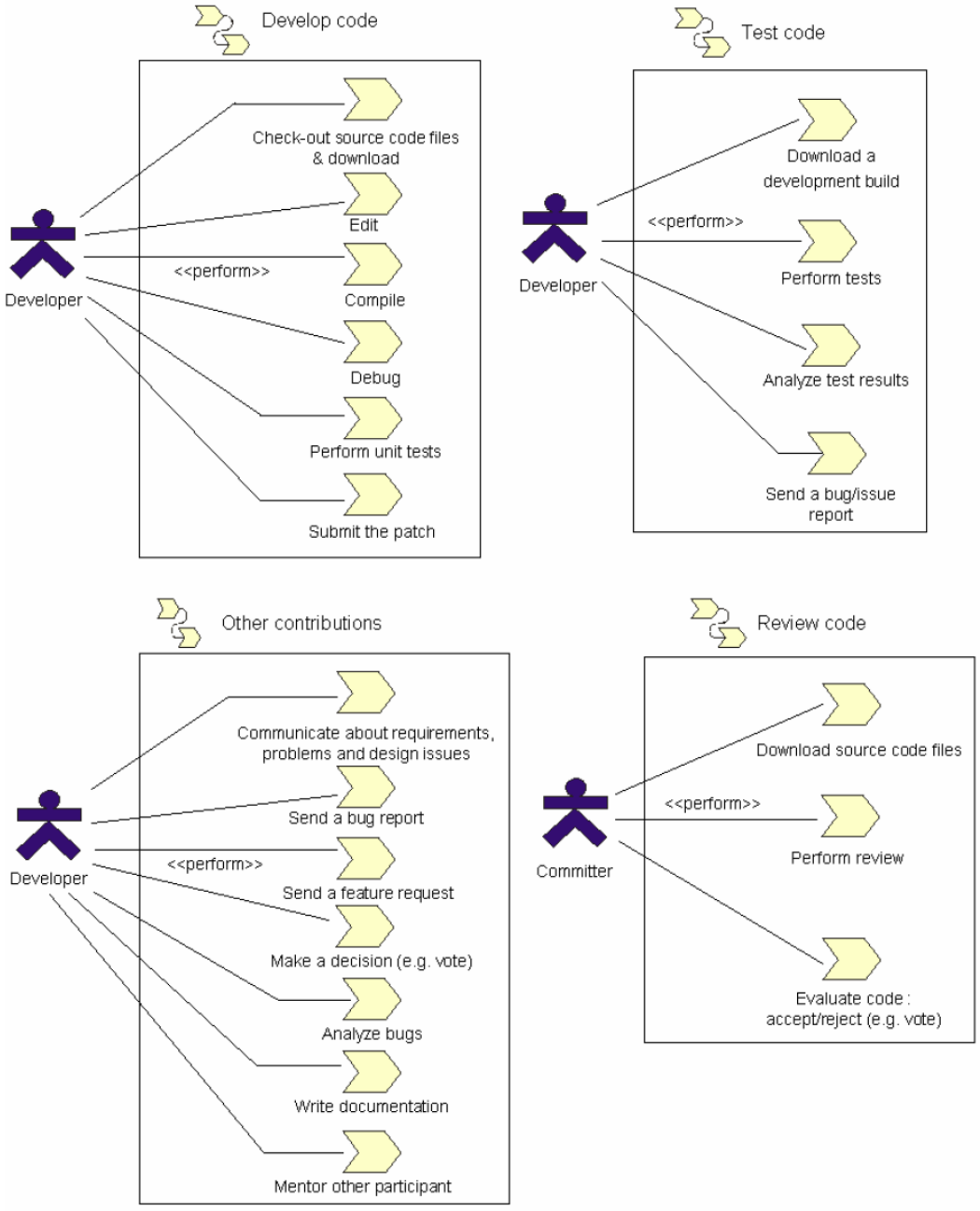

Figura 23 – Casos de uso orientado ao Desenvolvedor e ao Committer.

Fonte: LONCHAMP, 2005, p.12.

Tal fato pode ser tanto devido aos projetos iniciarem do esforço de uma única pessoa, como bem observa o autor, como também pode ser devido à grande parte dos projetos serem provenientes de sistemas de software já existentes, como Reis demonstrou na seção anterior. O autor credita o sucesso da divisão do trabalho e do alto grau de paralelismo de desenvolvimento dos projetos (alguns projetos mantém várias versões de um software) devido à modularização do design do

software em desenvolvimento. Todo trabalho desenvolvido em paralelo é combinado para formar uma nova versão (LONCHAMP, 2005, p.11-12).

A Fig. 24 ilustra o diagrama de casos de uso para o gerente do projeto, representando o processo de gerenciamento da liberação de uma versão do software. Novamente temos a participação dos usuários na atividade de teste de beta-versões, consistindo numa rica fonte de informação para os desenvolvedores avaliarem se todas as funcionalidades oferecidas estão sendo cumpridas.

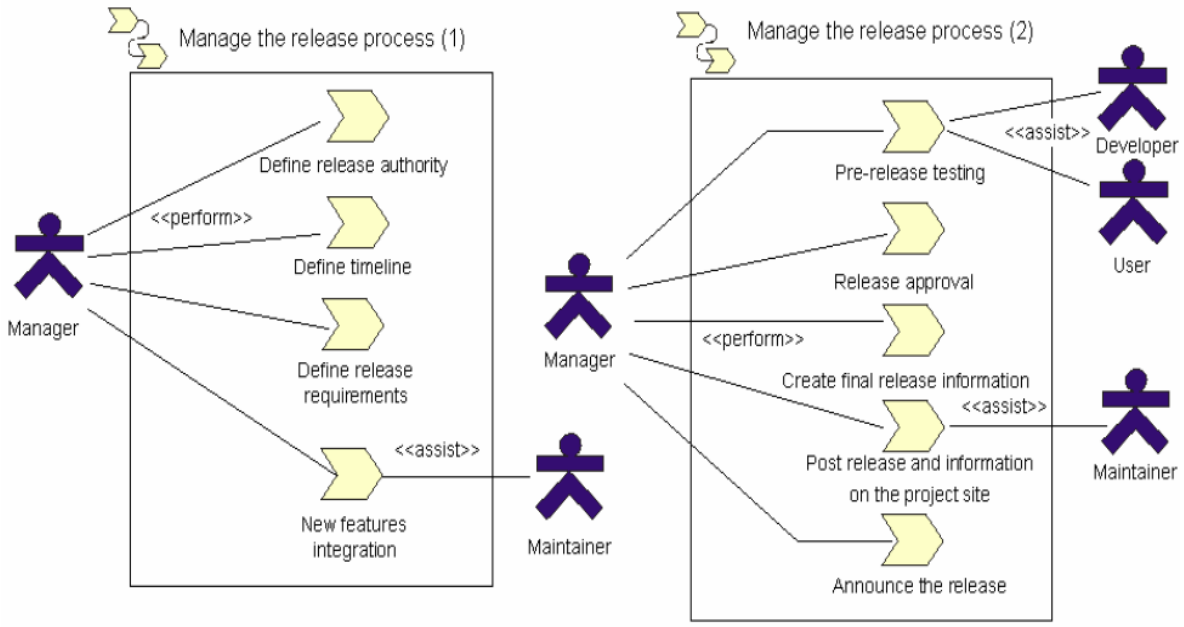

Figura 24 – Casos de uso orientado ao Gerente.

Fonte: LONCHAMP, 2005, p.13.

Essas atividades sugerem que o processo de desenvolvimento de software open source é mais informal que o desenvolvimento de softwares proprietários, a ausência de planos e de cronogramas observada por MOCKUS, FIELDING e HERBSLEB (2000, p.272) reforça essa idéia. Alguns projetos open source que possuem um processo de desenvolvimento mais consolidado já desenvolvem um roteiro de atividades e um breve resumo sobre o projeto e os propósitos do software que desenvolvem (LONCHAMP, 2005, p.13).

O diagrama de casos de uso da comunidade deixa evidente que suas principais atividades são para facilitar a participação dos usuários do software e sua comunicação com os demais membros da comunidade (Fig. 25 e Fig. 26).

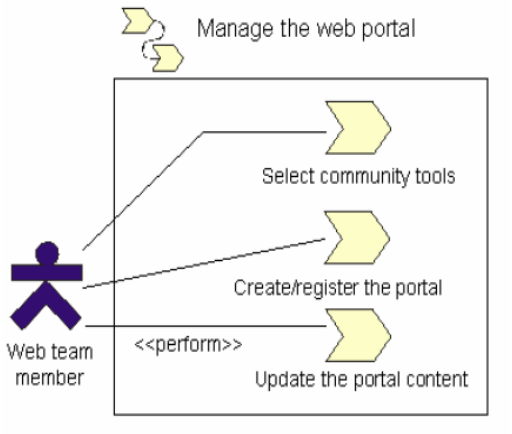

Figura 25 – Diagrama de casos de uso orientado à Comunidade (parte 1).

Fonte: LONCHAMP, 2005, p.14.

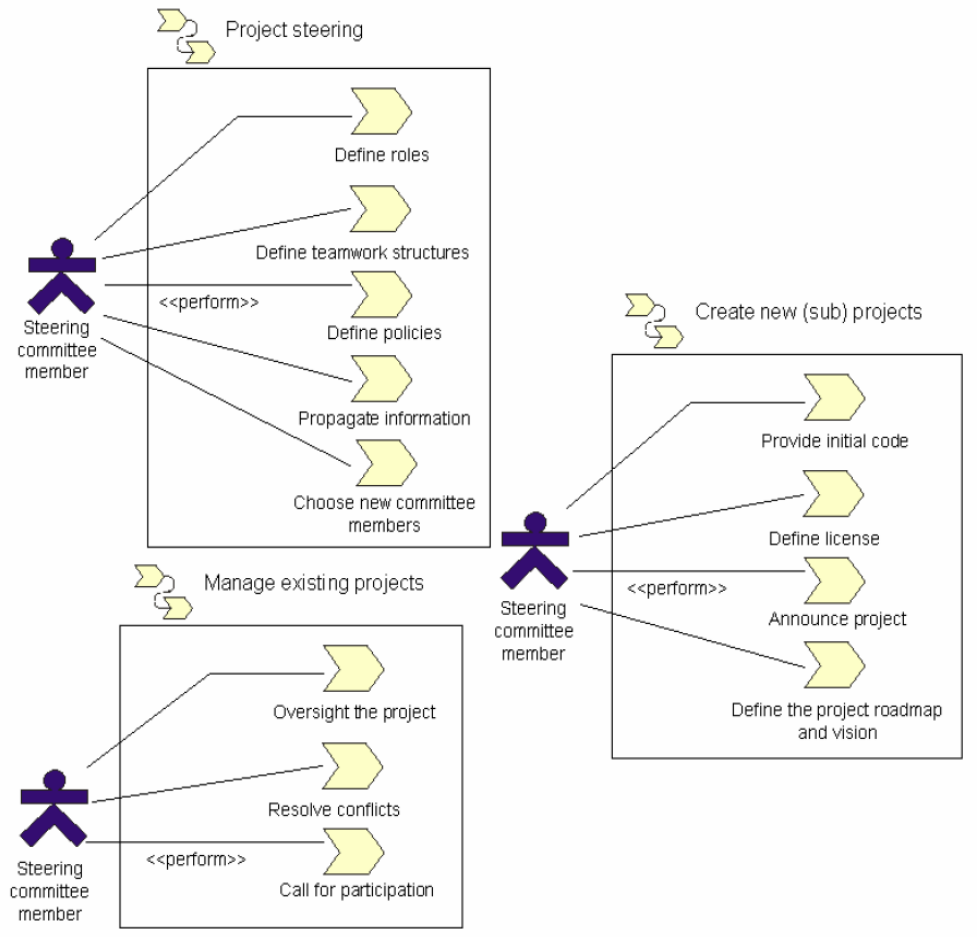

Figura 26 – Diagrama de casos de uso orientado à Comunidade (parte 2).

Fonte: LONCHAMP, 2005, p.14.

Na camada de nível específico Lonchamp utiliza diagramas de atividades SPEM para modelar processos de desenvolvimento de projetos que disponibilizam informações em páginas na internet e que definem seus próprios papéis no processo, documentos, guias e até mesmo detalhes de menor importância sobre o comportamento desses elementos do processo (LONCHAMP, 2005, p.16).

Observe um exemplo ilustrado na sua modelagem do processo de desenvolvimento do servidor Apache. Primeiramente Lonchamp definiu alguns exemplos de especialização dos Process Roles (Fig. 27). Os Process Roles do projeto Apache são valorizados de acordo com os seus méritos, suas responsabilidades e direitos estão diretamente ligados as suas contribuições dentro da comunidade. Sendo assim fica claro que os membros do comitê estão há muito tempo liderando os esforços na sua área de trabalho (LONCHAMP, 2005, p.24).

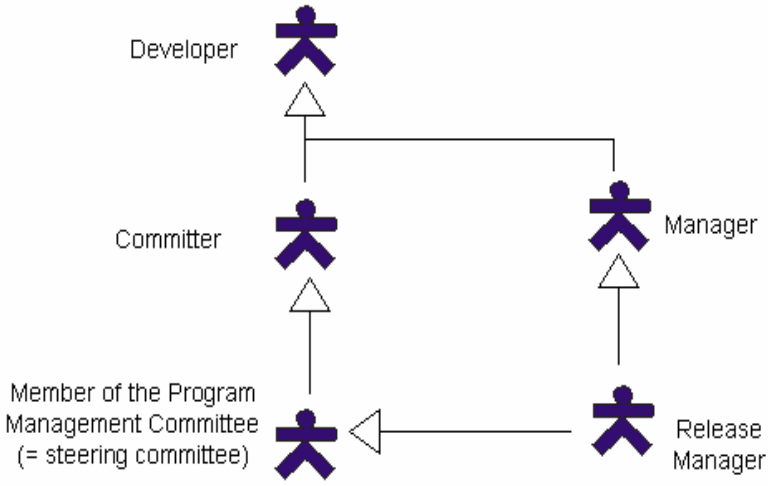

Figura 27 – Definição de alguns Process Roles do servidor Apache.

Fonte: LONCHAMP, 2005, p.24.

A atividade de teste de versão beta (*pre release testing*) vista na figura 24 é para Lonchamp a principal atividade na liberação de uma nova versão de software open source. Semelhantemente os testes operacionais são decisivos, em termos de validação, no desenvolvimento de um software comercial.

No projeto Apache tudo é feito democraticamente, no seu estilo de gerenciamento distribuído. A Fig. 28 demonstra o diagrama de atividades que representam essa atividade.

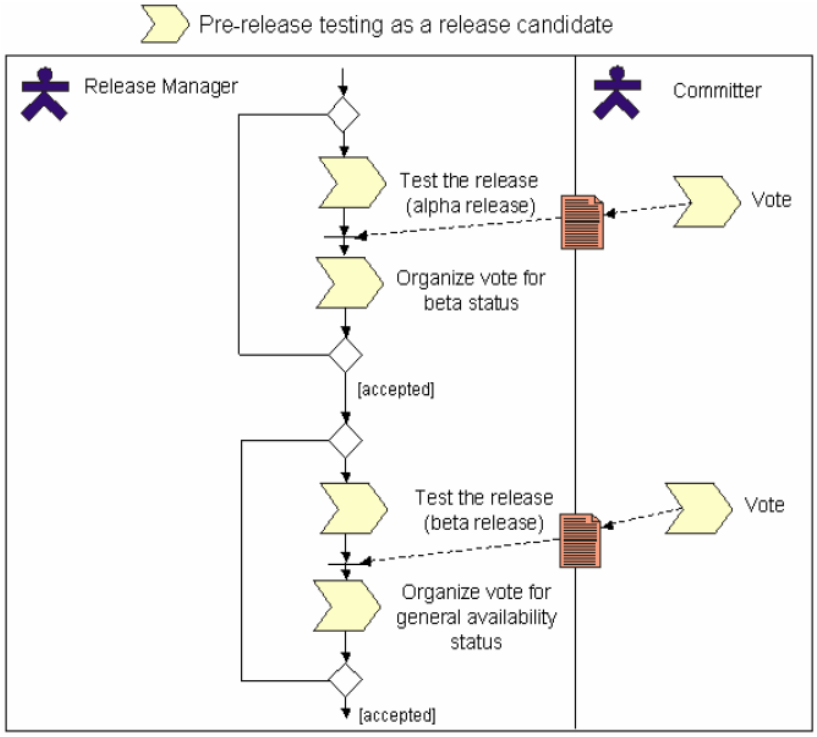

Figura 28 – Diagrama de atividades da fase de testes de uma versão beta do servidor Apache.

Fonte: LONCHAMP, 2005, p. 26.

Todas as transições entre os estágios do software (versão alfa para versão beta, e versão beta para versão final) são decididas pela maioria coletiva através de um sistema de votação, ainda que exista a restrição de pelo menos três membros do comitê (Apache Group) votar positivamente. Para garantir a qualidade, cada versão "candidata" à versão final deve rodar no principal servidor apache.org por dois ou três dias (Lonchamp, 2005, p.26).

Lonchamp conclui seu trabalho dizendo que o desenvolvimento open source não é solução para todos os problemas, mas uma boa alternativa que usa a Internet para melhorar suas atividades de desenvolvimento e propor soluções tanto para problemas comuns do processo de software como também para o processo de desenvolvimento distribuído (LONCHAMP, 2005, p.32).

Quanto ao uso do SPEM para a modelagem, Lonchamp ressalta que apesar das qualidades do metamodelo é necessária uma evolução dessa abordagem, visto que a modelagem das ferramentas teve que ser improvisada através do conceito de *Guidance*, pois o SPEM define como um tipo de guia os manuais de especificação

das ferramentas utilizadas, denominados *Tool Mentor* (LONCHAMP, 2005, p.29). Na versão 2.0 do SPEM, cujo lançamento está previsto neste ano de 2007, já propõe uma classe chamada *Tools*, com notação própria, para modelar e definir a atuação das ferramentas no processo de desenvolvimento de software. Essa prática alternativa utilizada por Lonchamp será muito importante para a modelagem proposta no próximo capítulo.

4. Processo de software baseado nos modelos open source

A partir da análise dos modelos de processo de desenvolvimento de software open source e tendo conhecimento dos processos de desenvolvimento tradicionais da engenharia de software, será proposto um modelo de desenvolvimento híbrido baseado nas práticas identificadas no desenvolvimento de software open source, pois conforme visto no capítulo anterior a metodologia open source possui lacunas que necessitam ser preenchidas para que este modelo possa ser utilizado tanto em projetos comerciais como por equipes de projetos de desenvolvimento de software que necessitem de um modelo diferenciado, documentado e que ao mesmo tempo cubra todas as etapas do processo. Após a análise do capítulo anterior fica evidente que a metodologia open source não possui explicitamente as atividades de análise e especificação de requisitos, como também não possuí um planejamento detalhado (ou muitas vezes nem existe um planejamento).

A definição de requisitos que é realizada, no entanto, apresenta a característica única de que o software se origina a partir de uma necessidade do seu próprio desenvolvedor, além de que os requisitos vão sendo propostos conforme a base de usuários do software cresce. Isto raramente é possível de ser implementado, pois, independente da área de atuação (acadêmico ou comercial) em questão, o usuário/cliente não têm obrigação nenhuma de ter noções de programação e desenvolvimento de software, contudo o usuário exerce papel fundamental em qualquer processo de desenvolvimento à medida que é dele que parte as principais funcionalidades do sistema e é a opinião do cliente/usuário que determina os resultados dos testes operacionais. Uma situação em que esse modelo open source de definição de requisitos poderia ser implementado é quando ocorre

desenvolvimento de software para uso interno de uma empresa ou grupo de pesquisa, onde os desenvolvedores provavelmente também utilizarão o sistema.

O processo proposto irá se preocupar principalmente em modelar a disciplina de *especificação do software* e a intensa comunicação existente entre os membros da equipe de desenvolvimento e o usuário. As demais disciplinas do processo de software serão modeladas baseadas na metodologia open source, conforme as narrativas e diagramas vistos no capítulo anterior. Visto que muitas das práticas identificadas nos projetos open source são inviáveis de se aplicar em projetos comerciais, o modelo proposto buscará adaptá-las de forma que possam ser utilizadas por qualquer equipe de desenvolvimento, porém sem relegar seu "passado" open source. Essas lacunas da metodologia open source serão preenchidas com uma análise de requisitos aos moldes do desenvolvimento evolucionário exploratório (utilizando-se versões alfa e beta) e incremental, com um planejamento detalhado das etapas do projeto do software.

A modelagem do processo será realizada adotando-se como padrão o metamodelo SPEM, abordado no capítulo anterior. Dessa forma utilizaremos a notação do SPEM para representação dos diagramas relativos ao processo híbrido, além de diagramas UML de atividades quando se fizer necessário prover maiores detalhes das iterações das atividades do processo. A Fig. 29 representa o mais alto nível de abstração do modelo proposto, doravante denominado como **DSBOS** (**D**esenvolvimento de **S**oftware **B**aseado em **O**pen **S**ource).

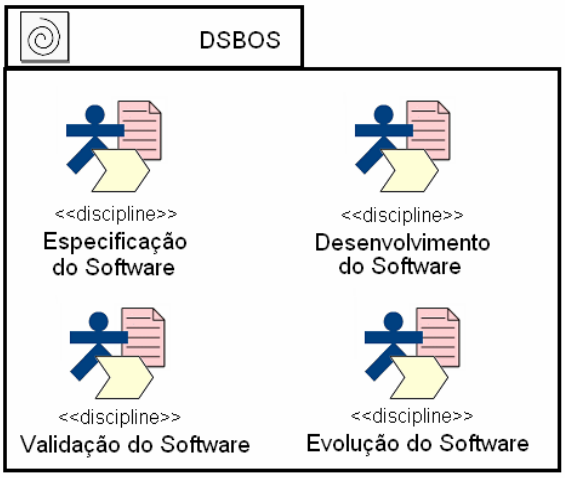

Figura 29 – Definição do modelo de desenvolvimento baseado em open source.

Em relação ao processo open source foi criada a disciplina de Especificação de Software, para definir e especificar os requisitos do sistema. Na disciplina de Desenvolvimento de Software foram extraídas as atividades que possuem relação com definição de requisitos, especificação de software e testes operacionais.

A definição dos papéis do processo é muito importante para que o gerente do projeto possa controlar as tarefas que estão sendo desenvolvidas e por qual membro de se sua equipe de desenvolvimento. A Fig. 30 apresenta a configuração de uma equipe de desenvolvimento em relação aos papéis do processo. Essa configuração serve tanto para projetos pequenos como também é possível, para grandes projetos, instanciar várias equipes de desenvolvimento sendo que cada equipe de desenvolvimento seria responsável por um módulo do projeto. Nesse caso se faz necessária a criação do papel de gerente de projeto, já que o mantenedor do projeto ficaria responsabilizado apenas em coordenar as atividades executadas dentro do módulo. O gerente do projeto seria responsável por executar as atividades necessárias para coordenar e integrar o trabalho dos módulos, atuando como um elo de ligação. Essa configuração de equipe também é apenas uma sugestão, outros papéis podem ser acrescentados ou funções acumuladas. Apesar de não ser aconselhável, o Arquiteto de Software pode assumir as funções do Analista de sistemas. Ou, ao invés de um engenheiro de testes, podemos ter vários tipos de “testadores” (testadores de mesa, testadores de interface de módulo, testadores de interface com o usuário, entre outros). Diversas generalizações são possíveis, de acordo com as necessidades do projeto ou da equipe de desenvolvimento.

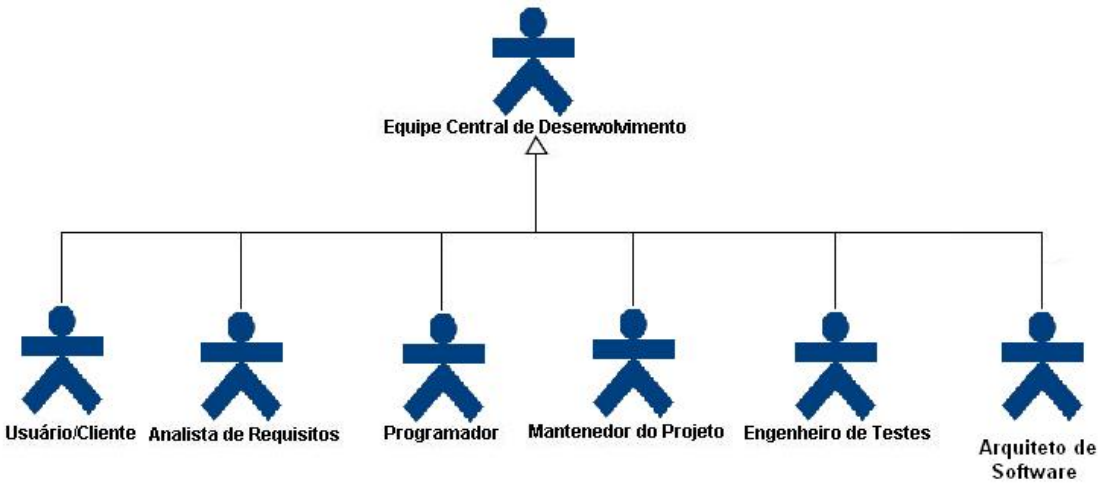

Figura 30 – Configuração hipotética de possíveis *Roles* no modelo de DSBOS.

4.1 Especificação do software

No processo open source os requisitos do software são definidos pelo fundador do projeto. Novas funcionalidades serão inseridas somente por necessidades do fundador do projeto ou da base de usuários do software, ou seja, os requisitos são adicionados de forma incremental e evolutiva. Em termos comerciais poucas companhias de software investem nesse tipo de feedback dos seus usuários, sendo uma prática recente no software proprietário. A prova disso é a crescente liberação de versões beta de softwares proprietários, fato que alguns anos atrás não acontecia.

Devido ao fato de não haver uma disciplina de requisitos bem definida no processo open source há a possibilidade de utilizar a análise de requisitos de qualquer outro processo de software. Entretanto é interessante realizar uma análise de requisitos que se assemelhe com o modelo open source para assim obter os mesmos benefícios e facilidades que os projetos possuem. No modelo DSBOS é realizada uma análise de requisitos com participação do usuário e muito bem documentada (Fig. 31).

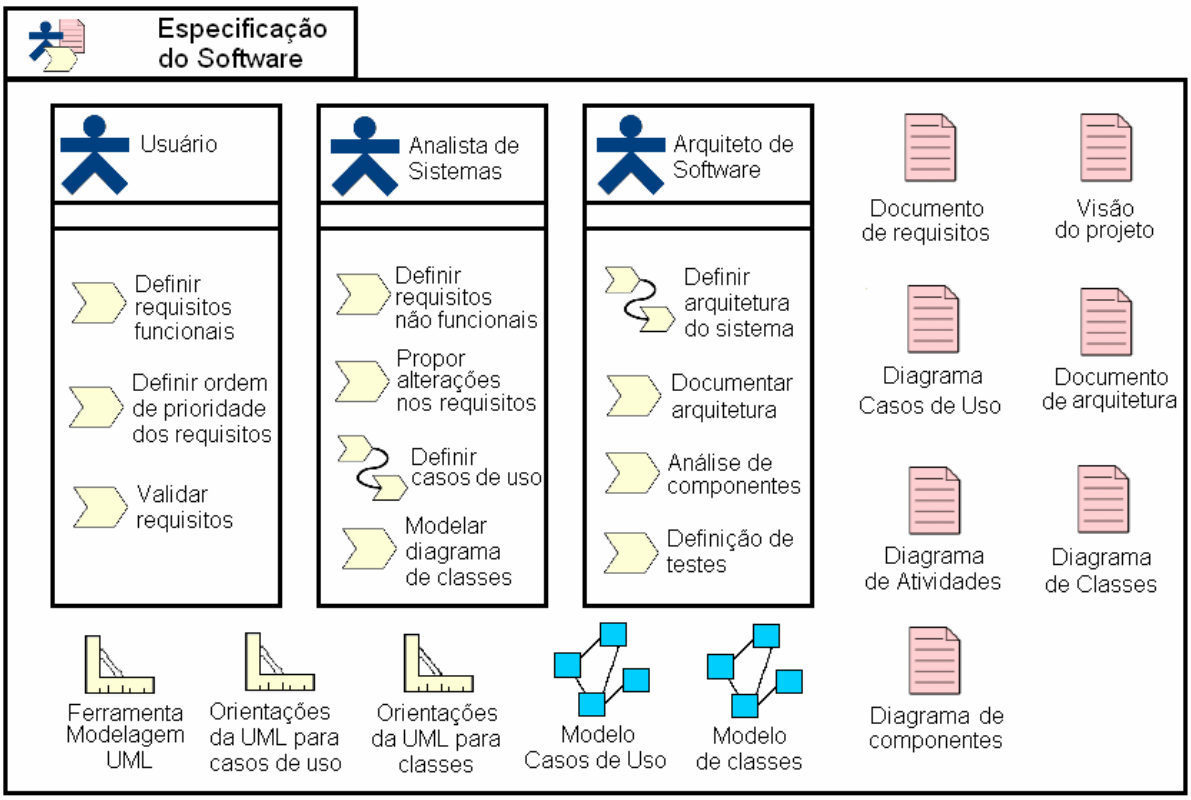

Figura 31 – Disciplina de especificação de software

Nesta especificação de software o usuário possui duas funções “novas” em relação aos modelos de processo da engenharia de software tradicional. Ao invés do Analista de Sistemas extrair os requisitos do usuário, o próprio usuário *determina* os requisitos funcionais do sistema além de definir a *prioridade* que cada requisito possui, descrevendo em linguagem natural as funções que o sistema poderia executar, quais pessoas do seu ambiente utilizarão o sistema (ou qual o público alvo), induzindo-o a uma descrição de cenários. Essa descrição forma o artefato “Visão do Projeto”, que guiará toda a equipe de desenvolvimento ao longo do processo. Dessa forma a probabilidade de erros e inconsistências no documento de requisitos pode ser reduzida. O Analista de Sistemas irá definir os requisitos não-funcionais do sistema e juntamente com os requisitos funcionais construir o diagrama de casos de uso, observando as dependências (Fig. 32).

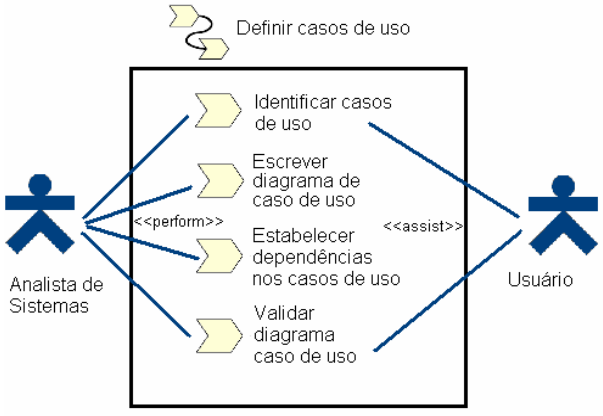

Figura 32 – O diagrama de casos de uso para o Analista de Sistemas.

Observe que o usuário participa do processo de construção.

O diagrama de casos de uso pode ser validado junto ao usuário ao mesmo tempo em que pode ser utilizado para validar seus requisitos. Também cabe ao Analista do sistema propor alterações nos requisitos funcionais do usuário, caso ocorra problemas no desenvolvimento do sistema ou se alguma inconsistência for identificada no andamento do projeto, porém a decisão final quanto aos requisitos permanece com o usuário. Essa atividade também pode ser utilizada para auxiliar/esclarecer a definição dos requisitos funcionais.

O diagrama de casos de uso e a descrição em linguagem natural feita pelo usuário irão auxiliar a construção do diagrama de atividades bem como a elaboração

do diagrama de classes. Como visto no capitulo anterior, os processos open source iniciam a partir de projetos existentes ou reutilizam código de projetos existentes. A utilização de modelos UML sugere o uso de orientação a objetos, que foi escolhida por causa da grande facilidade de reuso que proporciona além de que a UML é uma linguagem relativamente fácil e que proporciona a produção de uma grande variedade de diagramas para a documentação formalizada de desenvolvimento de sistemas. Através do diagrama de atividades é possível obter uma maior percepção dos passos que guiam essa atividade visto que as dependências estão representadas (Fig.33).

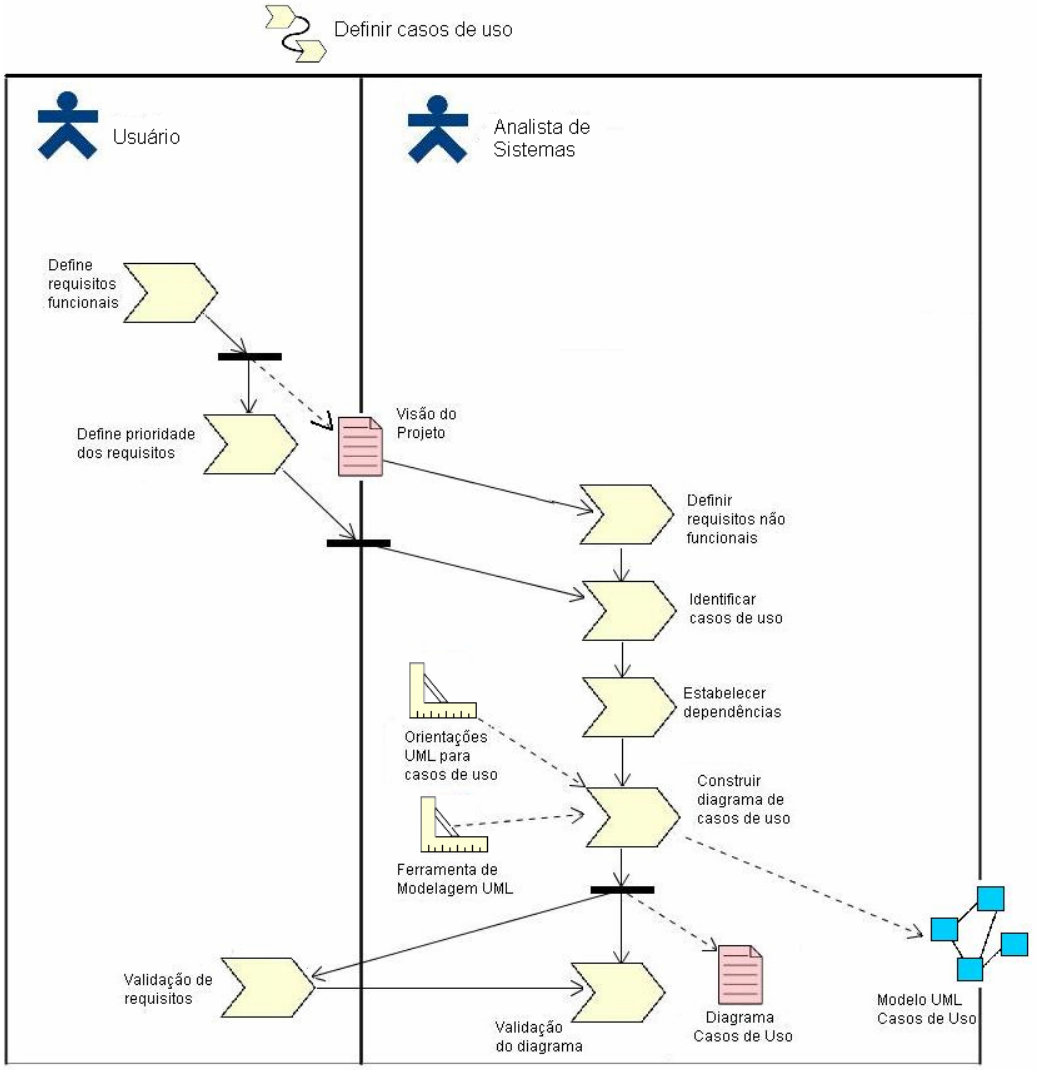

Figura 33 – Diagrama de atividades da construção
do diagrama de casos de uso.

O fluxo das atividades nestes diagramas pode ser vistos da mesma forma que um workflow. Os modelos UML obtidos são construídos seguindo o guia com as orientações da especificação da UML. Para representar o uso da ferramenta de

modelagem UML, foi utilizada uma *guidance*, do mesmo modo que Lonchamp realizou no trabalho exposto no capítulo anterior. O conjunto dos documentos produzidos forma o *work product* “documento de requisitos”.

O Arquiteto de Software é responsável por estabelecer *como* o sistema será desenvolvido, do hardware ao software, da estrutura física à implantação do sistema. A definição da arquitetura passa primeiramente por uma definição conceitual, mais abstrata. Gradativamente a arquitetura vai sendo refinada, identificando-se restrições de hardware e estabelecendo qual conjunto de softwares é a melhor alternativa para o desenvolvimento do projeto. A definição das tecnologias que serão usadas durante a execução do projeto será determinante para a definição dos requisitos técnicos da equipe que irá realizar a implementação do sistema de software. Baseado em todos estes levantamentos, o Arquiteto de Software também pode montar um guia dos testes que deverão ser executados durante o desenvolvimento do projeto.

Todas essas informações técnicas irão compor o documento de arquitetura, uma especificação técnica formal que deve abranger os diferentes pontos de vista do projeto. A Fig. 34 apresenta o caso de uso para o Arquiteto de Software relativo ao conjunto de atividades que envolvem a definição da arquitetura que o sistema do projeto apresentará.

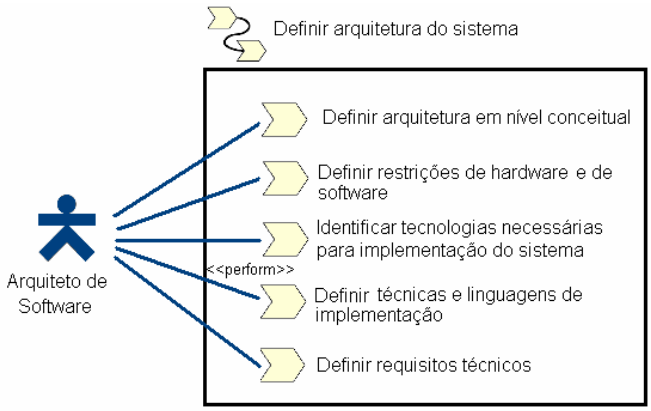

Figura 34 – Diagrama de casos de uso orientado ao Arquiteto de Software.

O fluxo destas atividades pode ser refinado através do diagrama de atividades (Fig. 35). O diagrama de classes e o diagrama de casos de uso são as bases na qual o arquiteto de software vai construir toda a estrutura do sistema. O Arquiteto também realiza uma busca por possíveis componentes que possam ser utilizados e que já tenham sido desenvolvidos em outros projetos, gerando assim o

diagrama de componentes. Dependendo do nível de complexidade do projeto cada uma dessas atividades ainda pode ser decomposta em passos ou outras subatividades, a fim de deixar o sistema o mais detalhado possível e minimizar as chances de ocorrerem erros por falta de informação dos membros da equipe. Nesse caso o arquiteto de software pode ter o auxílio de um projetista técnico ou de uma equipe inteira para agilizar o processo.

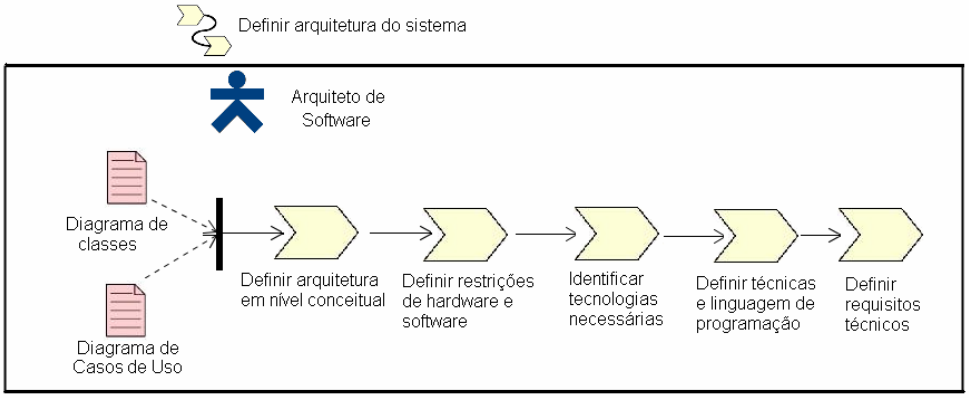

Figura 35 – Diagrama de atividades para a definição da arquitetura de um sistema.

Também é possível refinarmos uma atividade em um conjunto de passos (*Step*) e representá-los através de um diagrama de atividades. A Fig. 36 é um exemplo desta decomposição.

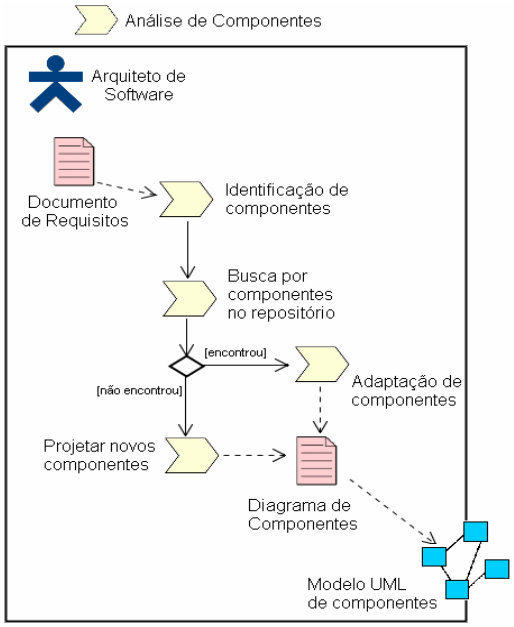

Figura 36 – Decomposição da atividade *Análise de Componentes* em passos.

Nos dois diagramas de atividades apresentados a documentação com especificação do sistema exerce papel fundamental no desempenho de atividades-chave, deixando explicito essa lacuna encontrada na maioria dos processos de desenvolvimento utilizados nos projetos open source. A introdução da disciplina de especificação de requisitos e a produção de diversos documentos que contemplam a formalização do sistema podem tornar mais atrativo a adoção deste modelo de desenvolvimento. Na próxima seção será abordado como as práticas da disciplina de desenvolvimento de software open source abordadas no capítulo anterior podem ser adotadas em outros tipos de projetos com desenvolvimento de software.

4.2 Desenvolvimento do software

Tomando como base o modelo proposto por Lonchamp na seção 3.3.4.2, a disciplina de desenvolvimento de software é levemente adaptada para poder comportar as principais características dos projetos open source (Fig. 37).

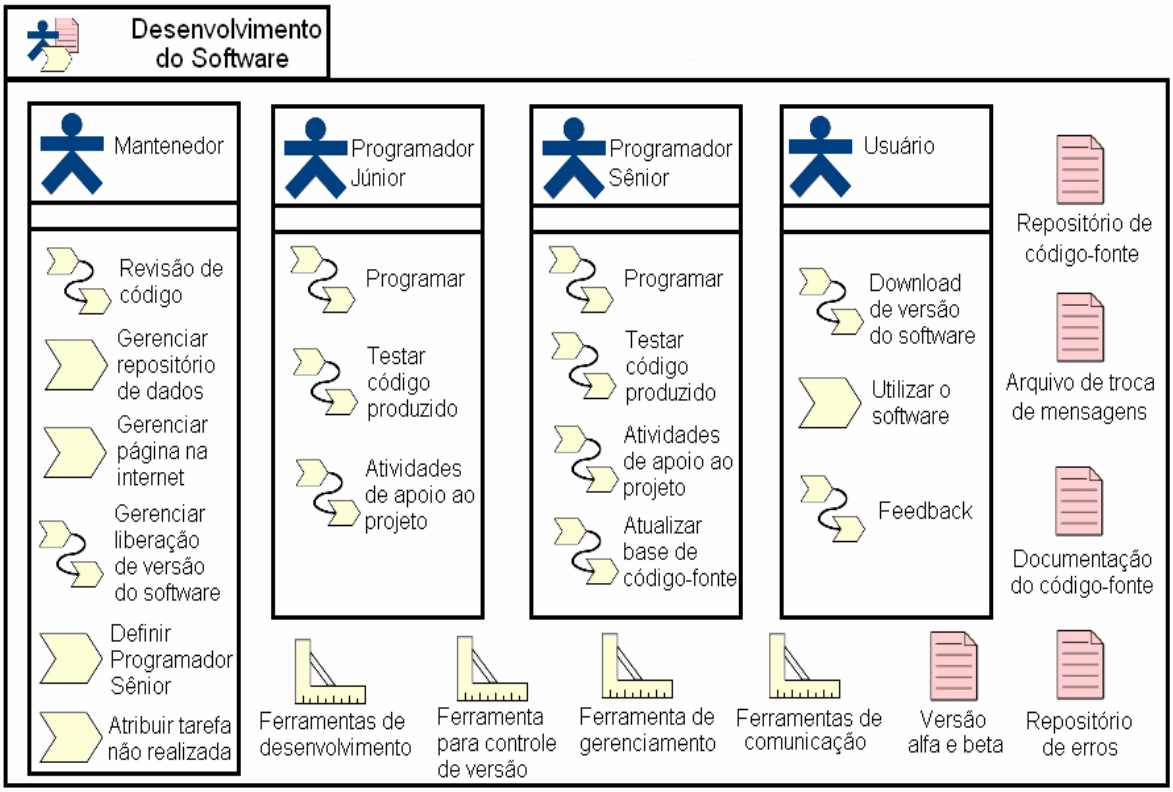

Figura 37 – Disciplina de Desenvolvimento de Software

A idéia consiste em incluir o usuário como desenvolvedor além de construir toda a execução do processo baseado no poder de comunicação que a Internet proporciona atualmente. Com a disseminação das conexões de alta velocidade (banda larga) é interessante rever o processo de desenvolvimento para adaptá-lo visando usufruir de benefícios e facilidades, buscando uma melhor integração com o usuário final do sistema. Assim como no processo open source, é viável instanciar um portal a fim de montar uma estrutura *online* que disponibilize informações sobre o projeto, em tempo de desenvolvimento, tanto para o cliente como para os desenvolvedores do sistema. Para um ambiente distribuído de desenvolvimento de software pode ser criada uma página na Internet. Para uma equipe de desenvolvimento centralizada esta página pode ser disponibilizada dentro da própria intranet – ficando somente os documentos relativos ao usuário oferecidos na Internet. Tal página (seja na internet ou na intranet), assim como nos projetos open source, disponibilizaria o código-fonte da versão mais estável, o pacote de instalação da versão mais estável, o documento de "Visão do Projeto", o "Documento de Requisitos", a distribuição das tarefas entre os membros da equipe (lista de tarefas e qual membro é responsável por cada uma delas), um mecanismo para comunicação direta entre os membros da equipe de desenvolvimento (grupo de discussão, por exemplo) além de uma área especial para notificação de defeitos e questionamentos quanto à usabilidade do software. O acesso a esses artefatos pode ser controlado através de um mecanismo de autenticação de usuário, identificando-o conforme suas atividades no processo de desenvolvimento e disponibilizando as informações do sistema conforme o perfil registrado (Usuário, Programador, Mantenedor, entre outros). Desta forma fica configurado um ambiente de desenvolvimento distribuído sendo possível prover comunicação síncrona entre seus participantes como também assíncrona, uma possibilidade a mais além do correio eletrônico.

É importante ressaltar o papel fundamental que as ferramentas de comunicação exercem para viabilizar toda essa estrutura. Os projetos open source utilizam ferramentas simples para implementar a comunicação assíncrona (através de e-mail e fóruns de discussão) e a comunicação síncrona (através de programas de mensagem instantânea), porém os avanços tecnológicos nestas áreas permitem que a comunicação seja aperfeiçoada com o uso de videoconferências e voz sobre IP. Quanto mais intrínseca e ativa ocorrer a troca de informações com os usuários finais do sistema, menor a probabilidade de acontecer falta – ou distorção – de

requisitos. Se as equipes de desenvolvimento se comunicar efetivamente, menos traumático será o processo de implementação do software e maior será a qualidade do produto final. Em ambientes de desenvolvimento distribuído, como nos grandes projetos open source, a comunicação é fundamental para evitar retrabalho e reduzir falhas. No modelo proposto na figura 37 as ferramentas para implementar as redes de comunicações estão representadas através de *guidance*, o que de certa forma não deixa de ser a idéia base dos modelos de desenvolvimento open source estudados e que também é a idéia central deste modelo de processo proposto: um desenvolvimento orientado ao cliente, maximizando sua participação no processo através das ferramentas de comunicação. Por este motivo é que se abre mão da simplicidade do open source para atuar em novos modelos de negócio, ampliando a área de atuação da equipe de desenvolvimento.

Outra novidade é a tentativa de simular a atribuição de trabalho semelhantemente ao modo como ocorre nos projetos open source. Cada Programador, ao tomar conhecimento do documento de requisitos, deve candidatar-se a uma tarefa de acordo com suas habilidades e motivações profissionais. Através de uma *ferramenta de gerenciamento* o Mantenedor deve controlar as tarefas existentes e os desenvolvedores alocados para cada tarefa, evitando que tarefas fiquem sem responsáveis bem como não deixando mão-de-obra ociosa. Se uma atividade não está sendo executada, um membro da equipe de desenvolvimento que não esteja alocado para alguma tarefa deve ser responsabilizado pela realização dessa tarefa pelo Mantenedor. Assim cresce a probabilidade dos desenvolvedores trabalharem com uma motivação maior, aumentando a produtividade e a qualidade.

Uma equipe de desenvolvimento, por menor que seja, conta com vários programadores que através de um conjunto de ferramentas de desenvolvimento – IDE – implementam a especificação do sistema. Nos projetos open source a experiência do desenvolvedor no projeto determina o seu nível de responsabilidade dentro do ciclo de vida do desenvolvimento do software. Da mesma forma serão considerados dois tipos de programadores no modelo DSBOS. O Programador Júnior irá produzir código de acordo com o trabalho que se propôs a desenvolver (Fig. 38). Caso já tenha concluído suas atividades, pode obter uma cópia do código-fonte e auxiliar no desenvolvimento de outra tarefa do projeto. Ao concluir seu trabalho, envia os arquivos e a documentação para o Programador Sênior. Os

desenvolvedores também se ocupam de outras atividades durante o desenvolvimento do software (Fig. 39).

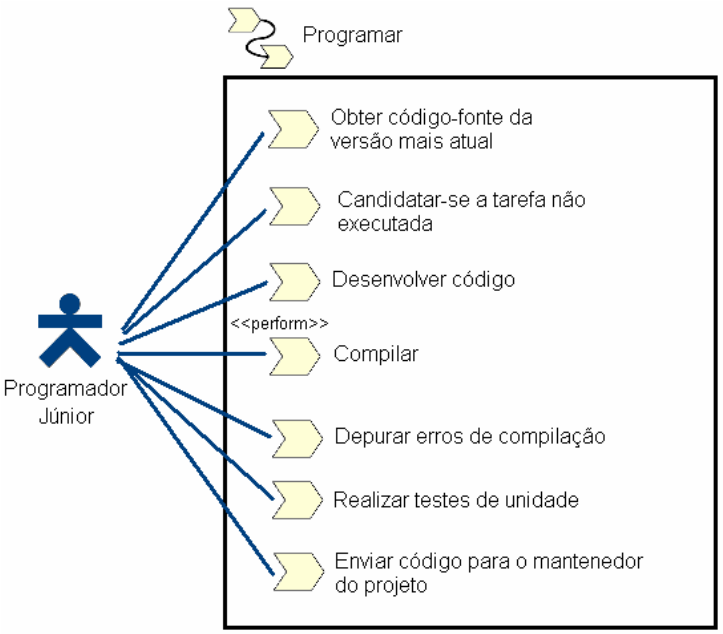

Figura 38 – Casos de uso relativos à atividade de desenvolver código-fonte

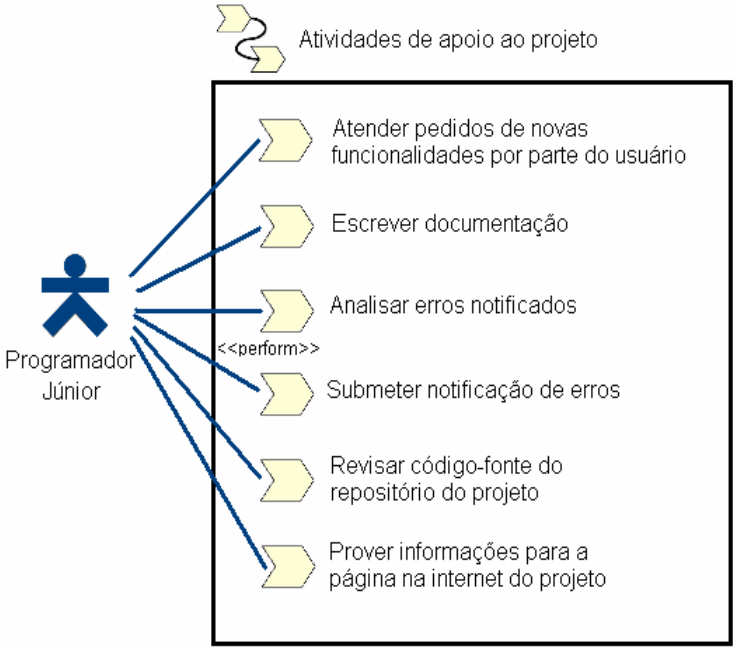

Figura 39 – Atividades de apoio ao desenvolvimento do software

Essas atividades que auxiliam o desenvolvimento não possuem fluxo de trabalho ou seqüência pré-determinados, ou seja, podem estar sendo desenvolvidas

em qualquer momento do ciclo de vida do desenvolvimento do sistema. Apesar de algumas dessas atividades estarem tradicionalmente ligadas a disciplina de Manutenção e/ou Evolução de Software, seguindo o modelo open source, também podem ser executadas durante a disciplina de Desenvolvimento de Software.

O Programador Sênior é mais experiente e além de desenvolver as mesmas atividades que o Programador Júnior, terá o poder para efetivar alterações no código-fonte base do projeto (Fig. 40). É importante observar que na prática muitas empresas adotam esses dois níveis de experiência entre os programadores, contudo não necessariamente desempenham as mesmas atividades que neste modelo efetuarão.

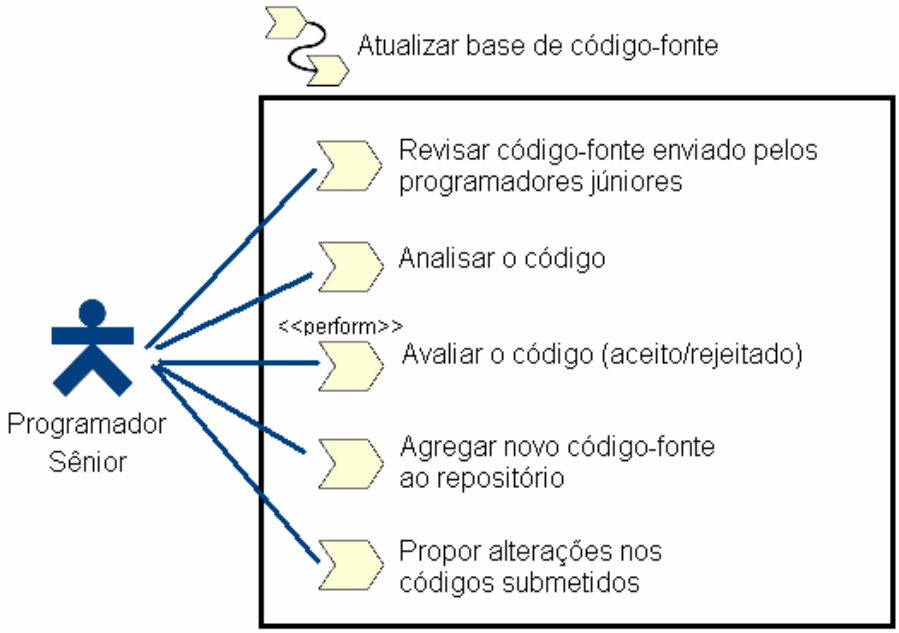

Figura 40 – Casos de uso referentes à atualização do repositório de arquivos do código-fonte

A Fig. 41 exibe o diagrama de atividades para o caso de uso de atualização do código-fonte base que é disponibilizado no repositório. Várias vezes a atividade de revisão de código aparece no processo como um mecanismo para garantir não só a qualidade do software como também a segurança do sistema de software a ser desenvolvido, sendo abordada na próxima seção. A análise do código é relativa a compreensão da solução implementada pelos programadores para determinado problema. A avaliação determina se a solução implementada é a mais adequada, seja em termos de complexidade, desempenho, usabilidade ou segurança. Em

qualquer uma das circunstâncias, não sendo aprovada pelo Programador Sênior, novas alternativas devem ser desenvolvidas para garantir a consistência do projeto.

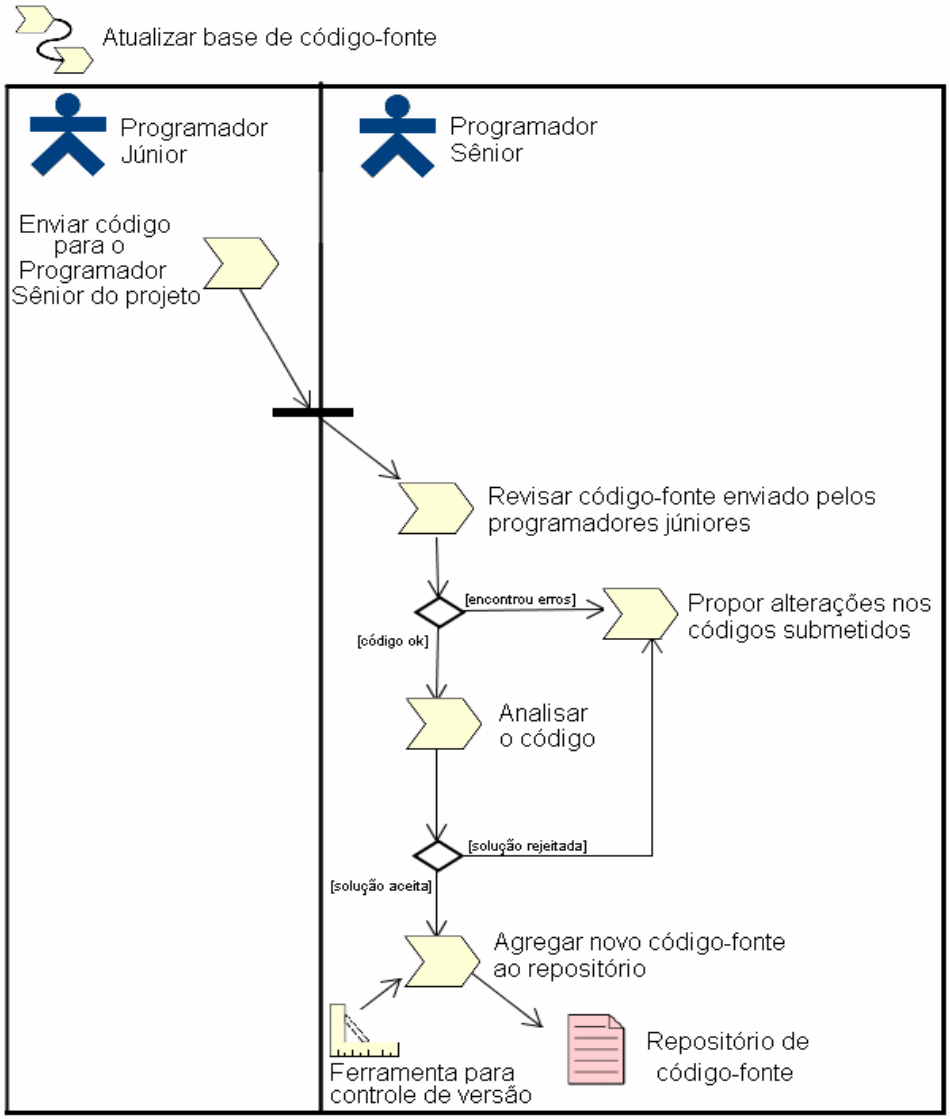

Figura 41 – Fluxo de atividades para atualização de código no repositório

O Mantenedor atua como o gerente do projeto (ou do módulo, dependendo da dimensão do sistema em construção). É ele que gerencia os conteúdos da página na Internet, pois como responsável direto pelo projeto deve fornecer para os usuários/clientes informações sobre o andamento do desenvolvimento. Como líder do projeto também é dever do Mantenedor escolher, dentro de sua equipe de desenvolvimento, qual programador deverá ficar responsável pela submissão de arquivos-fonte no repositório de código do projeto. Essa escolha deve ser baseada na experiência do programador dentro da organização e de suas contribuições em

projetos anteriores, sempre devendo privilegiar aqueles que geralmente se dedicam e contribuem ativamente nos trabalhos.

O Mantenedor também é responsável pelo gerenciamento dos arquivos do repositório de dados do projeto, auxiliando o Programador Sênior a controlar a produção da equipe com a ferramenta de controle de versão (evitando assim o retrabalho e controlando a produção da equipe de desenvolvimento) e garantindo que toda a documentação associada ao sistema em construção esteja disponível no repositório. Enquanto não há código suficiente para a construção de uma versão, ele atua como um revisor de código e contribui eventualmente com correções e/ou soluções alternativas (Fig. 42).

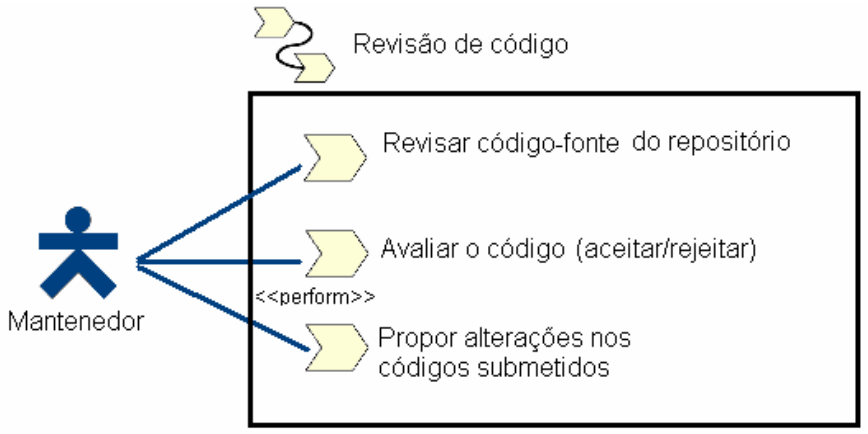

Figura 42 – Caso de uso do Mantenedor para *work definition* Revisão de Código

A atividade de liberação de uma versão do software tem a participação de toda a equipe de desenvolvimento, com o gerenciamento do Mantenedor e o auxílio dos programadores e do usuário em algumas etapas (Fig. 43). Assim que o documento de requisitos é finalizado pelo Analista de Sistemas e pelo Arquiteto de Software, o Mantenedor elabora um cronograma com datas para liberação da versão alpha e da versão beta. Na versão alfa os requisitos funcionais definidos pelo usuário como sendo de alta prioridade devem estar implementados, visto que são a parte mais importante do sistema e por isso precisam passar por uma bateria de testes maior. Na versão beta os requisitos funcionais com maior prioridade já devem estar estáveis, pois nesta versão os requisitos funcionais de baixa prioridade e os requisitos não-funcionais do software é que devem ser implementados. A versão

beta é quase como uma versão candidata a versão estável, os testes fumaça e operacional já poderiam ser realizados nesta fase.

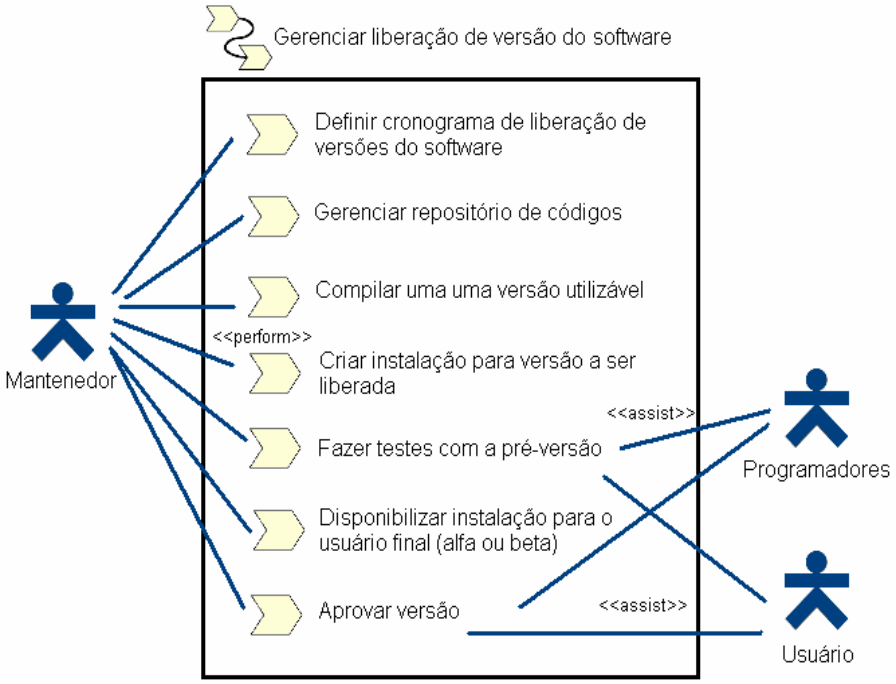

Figura 43 – Casos de uso orientado a liberação de versão de software.

A Fig. 44 demonstra como seria o fluxo das atividades para a liberação de uma versão do software. A partir do momento que o repositório de dados contém código suficiente – ao menos dos requisitos funcionais – a compilação de uma versão alfa é feita pelo Mantenedor, acrescentando ao código-fonte a documentação relacionada ao sistema. A proposta deste modelo, tal como nos projetos open source, é que o usuário também participe dos testes desta versão. Apesar de não se tratar da fase operacional, o usuário é quem define os requisitos funcionais e ninguém melhor do que o próprio usuário para avaliá-los.

A *guidance* ferramenta de notificação de erros é uma especialização de ferramentas de desenvolvimento e consiste em um sistema para notificar e gerenciar os erros encontrados nas versões do software, tendo o seu funcionamento nos mesmos moldes que a ferramenta Bugzilla, comentada no capítulo anterior. A utilização de uma ferramenta deste tipo além de facilitar o controle dos erros e indicar quem está trabalhando na solução dos mesmos, fornece uma interface simples que permite até mesmo o usuário utilizá-la sem maiores dificuldades, além

de que é possível integrar a uma página da internet fornecendo um meio poderoso de comunicação de erros entre os membros de equipes distribuídas de desenvolvimento – assim como as equipes de desenvolvimento open source abordadas na seção 3.3.

O tempo desta atividade de testes é variável e depende do projeto em execução, pode ser um período de tempo pré-estabelecido no cronograma definido pelo mantenedor como também pode ser um número máximo de erros durante um determinado período de tempo.

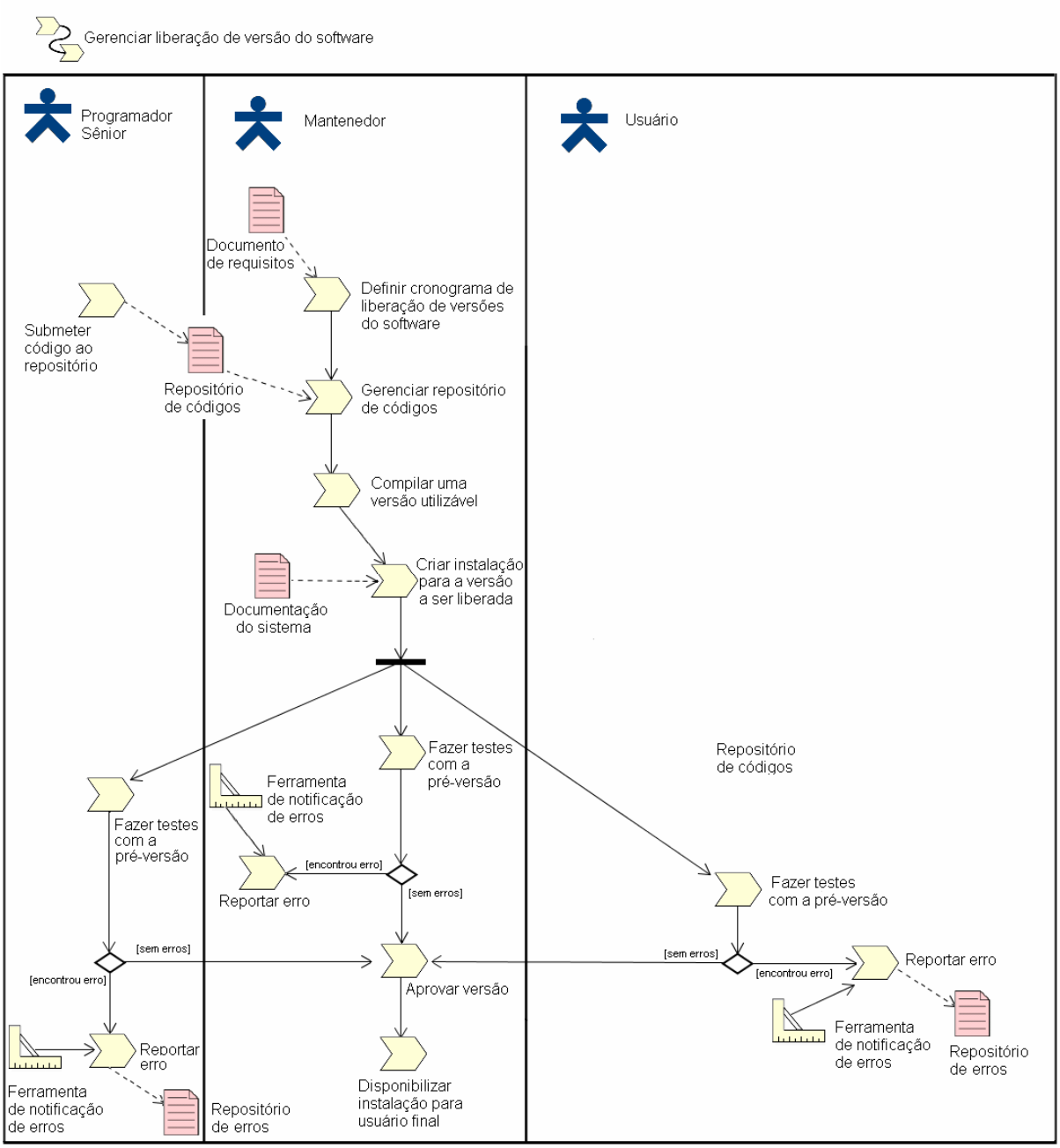

Figura 44 – Diagrama de atividades para a liberação de uma versão do software

Caso a versão testada seja uma versão alfa, o Mantenedor deve começar a preparar o lançamento de uma versão beta, dependendo da produtividade da equipe de desenvolvimento. No caso de uma versão beta ter sido testada, após todas as correções terem sido aplicadas ao software o Mantenedor deve disponibilizar uma versão final do software, para ser amplamente distribuída e utilizada por todos os usuários do sistema. A versão sendo aprovada encerra o ciclo de vida da disciplina de desenvolvimento do software.

4.3 Validação do software

A Fig. 45 representa a disciplina de validação do software no modelo de desenvolvimento proposto. Suas atividades são muito semelhantes em relação ao desenvolvimento se diferenciando pelo fato de o código produzido ser bem menor e apenas para corrigir falhas. Também compartilha das mesmas ferramentas e meios de comunicação da disciplina de desenvolvimento. Ela não é uma disciplina com início e fim demarcados. Seguindo a metodologia dos modelos open source, a proposta deste trabalho é um desenvolvimento orientado a validação do software a partir do momento em que o usuário passa a fazer parte da equipe de desenvolvimento.

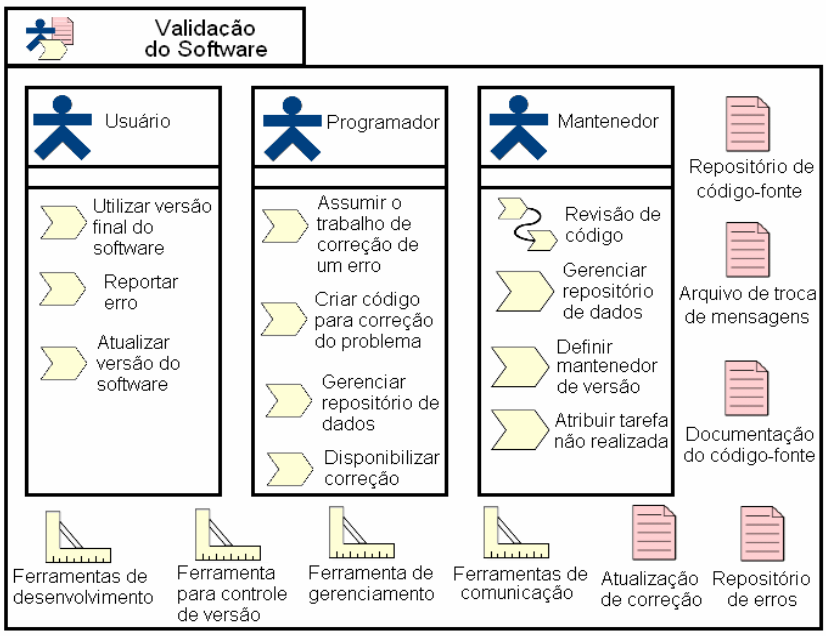

Figura 45 – Disciplina de validação do software

Durante a disciplina de desenvolvimento o usuário já começa a testar o software através das pré-versões alfa e beta, e os próprios programadores podem validar o código produzido no momento que é disponibilizado no repositório de dados, conforme apresentado na seção anterior. Aliando a isso um canal de comunicação direta entre usuário e desenvolvedores através da página na internet, correio eletrônico e da ferramenta de notificação de erros, o software produzido tende a satisfazer os requisitos do usuário com maior sucesso.

Ao final da disciplina de desenvolvimento a versão do software liberado já está validada pelo cliente e pelos demais desenvolvedores. No entanto, após o lançamento da versão final do sistema, a atividade de testes não pára, apoiando-se na ferramenta de notificação de erros para gerenciar o trabalho de suporte. A Fig. 46 demonstra a proposta de aplicação de correções no software final. Erros que eventualmente não tenham acontecido durante o desenvolvimento, erros devido à atualização de softwares indiretamente utilizados pelo sistema (atualização do navegador de páginas na Internet, por exemplo) e erros causados por falhas em sistemas que trabalham integrados com o software desenvolvido (um defeito no sistema operacional, por exemplo) são resolvidos nesta etapa.

Esta disciplina não tem previsão de encerramento, pois enquanto o software permanecer em uso a equipe de desenvolvimento deve estar preparada para atender aos erros que eventualmente possam ocorrer. No caso de um grande projeto, em que vários usuários utilizam diferentes versões do software desenvolvido, o Mantenedor pode designar um Programador Sênior para assumir as funções de Mantenedor para uma versão específica do software, evitando que os usuários que ainda usam alguma versão anterior do software não sejam prejudicados. Caso contrário, devido à diminuição do fluxo de trabalho, o próprio Mantenedor passa a ser responsável pelas atualizações no repositório de códigos.

Uma prática que não está representada nos diagramas, mas que pode ser aplicada ao modelo proposto, são os pacotes de atualizações. Com o tempo diversas atualizações são produzidas e para facilitar a operação de atualização do software do ponto de vista do usuário, um conjunto de atualizações pode ser agrupado pelo Mantenedor para formar um arquivo de instalação de atualizações (update). Dependendo do número de atualizações produzidas pela equipe, elas podem ser incorporadas ao código base do software, se integrando aos módulos já

desenvolvidos e formando uma nova versão do software. Neste caso recomenda-se adotar uma nomenclatura semelhante à encontrada nos projetos de software livre: o número antes do ponto representa a versão do código fonte base e o número após o ponto representa a versão estável (se for um número par) ou a versão instável do software (caso seja um número ímpar). Um terceiro número pode indicar quantas liberações já foram realizadas. Por exemplo, a versão número 3.4.3 indica que se trata do código base estável 3.4 e que três versões desta base já foram liberadas para os usuários.

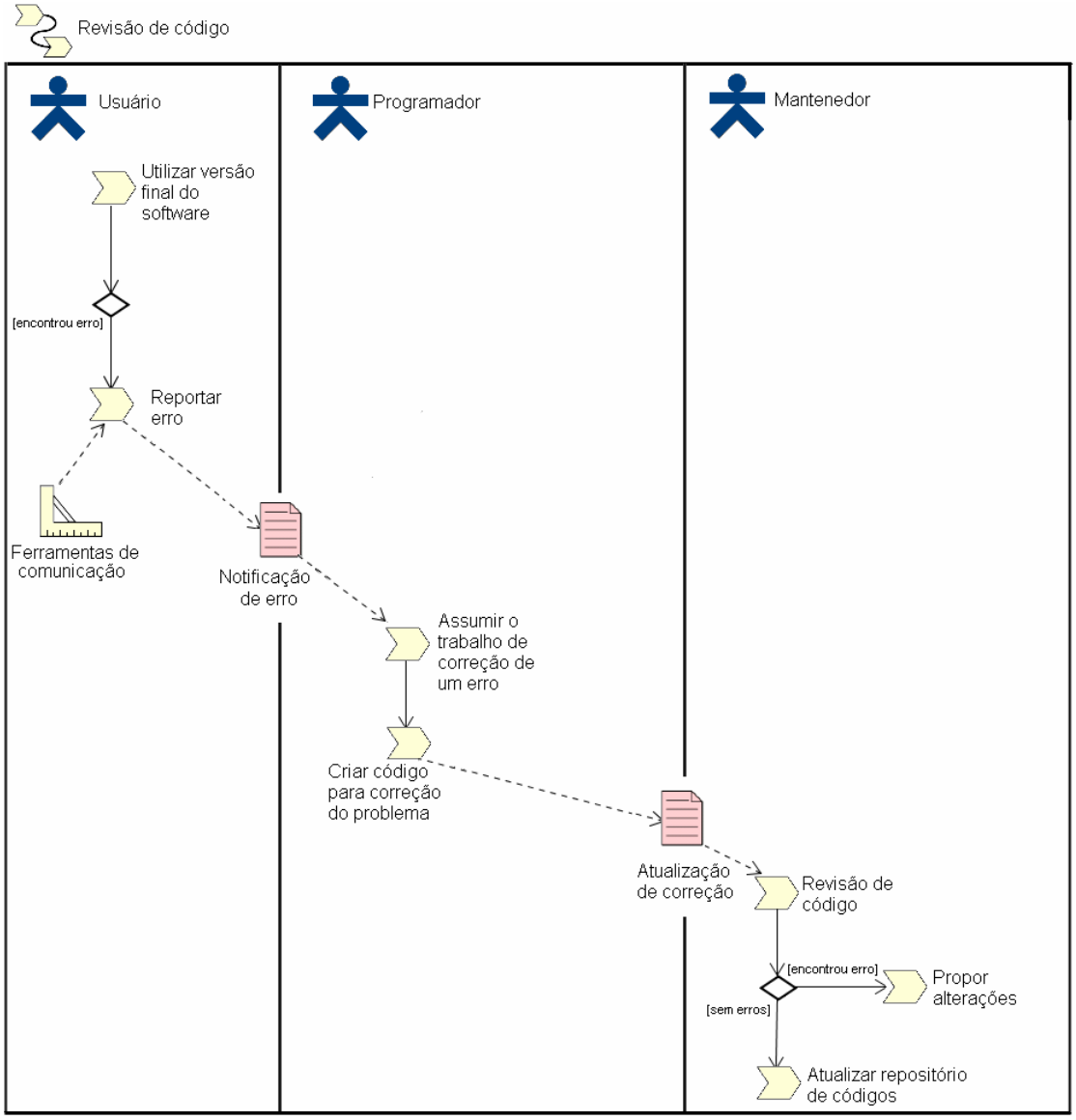

Figura 46 – Especificação dos passos para criar uma atualização do software

4.4 Evolução do software

No modelo proposto por este trabalho, a disciplina de evolução do software também ocorre concomitantemente com a disciplina de desenvolvimento do software e com a disciplina de manutenção do software, não constituindo assim uma etapa distinta das demais disciplinas no DSBOS. Os canais de comunicação existentes entre o usuário e a equipe de desenvolvimento permitem que a qualquer momento o usuário possa elucidar algum requisito mal implementado pela equipe. Do mesmo, modo novos requisitos podem ser propostos pelo usuário à medida que as versões alfa e beta do software são liberadas. A Fig. 47 exibe a proposta de atividades referente a uma solicitação de novos requisitos por parte do usuário.

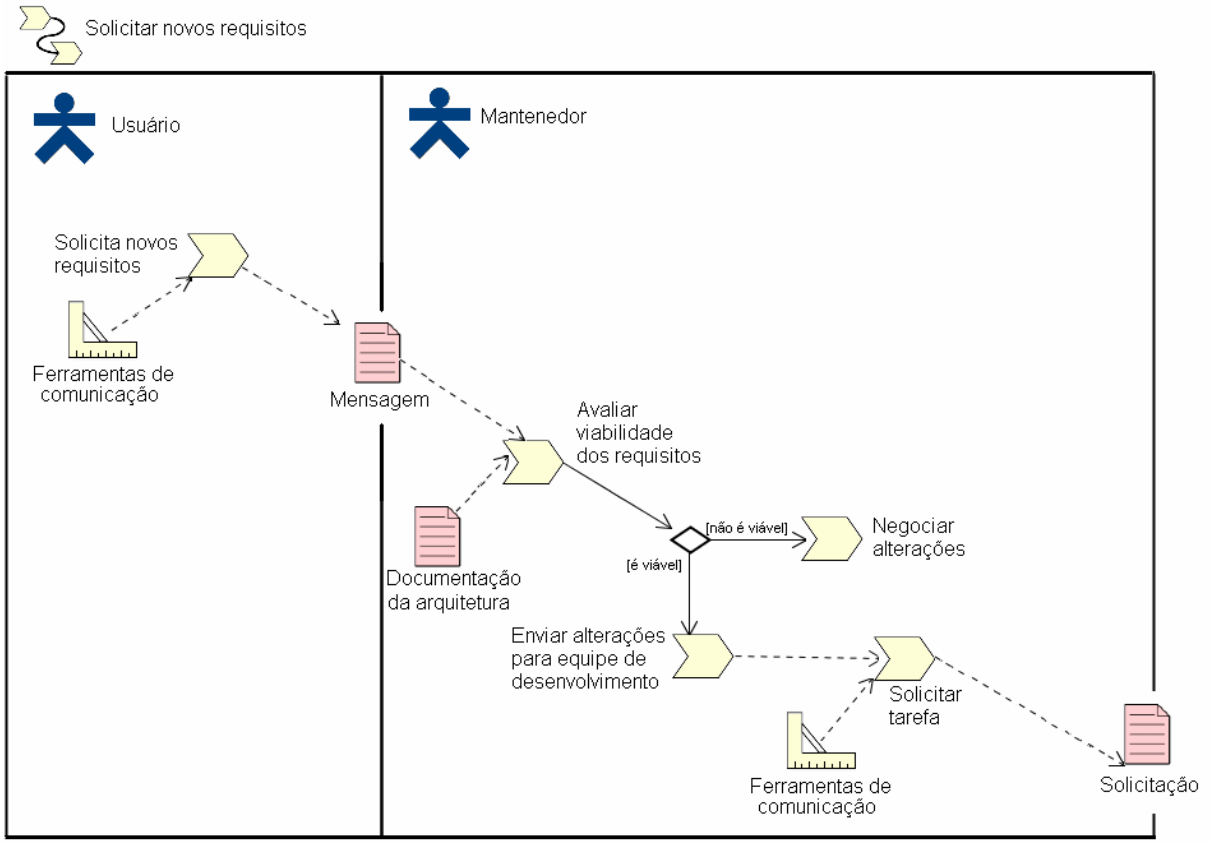

Figura 47 – Diagrama de atividades para a solicitação de novos requisitos pelo usuário

A solicitação do usuário é realizada através de uma das ferramentas de comunicação com preferência para o correio eletrônico da lista de discussão do projeto, pois assim o requerimento fica registrado no repositório de troca de mensagens do projeto e todos os membros da equipe de desenvolvimento tomam conhecimento das necessidades do cliente. Também pode ser realizada através da ferramenta para notificação de erros, apesar de não ser o propósito deste mecanismo.

A avaliação da viabilidade dos requisitos solicitados é executada pelo Mantenedor do projeto. O Arquiteto de Software também estaria qualificado para realizar essa atividade, porém como o a solicitação de novos requisitos supõe que o projeto já está em desenvolvimento ou até mesmo concluído, o Mantenedor possui uma visão mais ampla do sistema como um todo e assim estaria mais capacitado a avaliar, tanto em nível conceitual como em nível de implementação, a viabilidade da solicitação. Para tanto se utiliza do documento de arquitetura, escrito pelo Arquiteto de Software na disciplina de especificação do software, para determinar se é possível implementar os novos requisitos.

Este é um exemplo claro de quão importante é o planejamento feito durante a especificação do sistema pelo Arquiteto de Software. Sem o documento de arquitetura ficaria mais difícil determinar a viabilidade de um novo requisito solicitado pelo usuário, podendo inclusive acontecer de no meio do desenvolvimento a equipe chegar à conclusão de que a arquitetura adotada não permite a implementação do novo requisito. O custo adicional gerado por tal acontecimento é um ônus que as organizações relacionadas ao desenvolvimento de software não podem arcar no competitivo modelo de negócio vigente.

Caso as alterações não sejam viáveis, o Mantenedor deve negociar com o usuário possíveis adaptações no requisito solicitado. Como a solicitação foi enviada para a lista de discussão do projeto um debate pode ser organizado pelo Mantenedor sobre o novo requisito solicitado a fim de tentar solucionar o problema. Em última instância, o Arquiteto de Software pode fazer uma avaliação sobre a possibilidade de alterar a arquitetura do sistema de software em desenvolvimento.

O Mantenedor dando o parecer favorável ao requisito solicitado pelo usuário, o processo de desenvolvimento de código ocorre do mesmo modo que no restante do projeto (compartilhando as mesmas atividades e *work definitions*) e encerrando

assim o ciclo de vida do processo de desenvolvimento. Esta prática proposta pelo modelo DSBOS torna a evolução do software uma etapa mais natural: o progresso do sistema já ocorre durante sua implementação e validação. Esta dinâmica atende bem a projetos em que há mudanças constantes nos requisitos do usuário, reduzindo os impactos no projeto e no desenvolvimento do software sem diminuir a qualidade e a segurança do sistema final.

Este trabalho conclui assim a tentativa de modelar um processo de desenvolvimento de software utilizando práticas open source, dando atenção especial para a disciplina de especificação de software, onde poucos projetos open source apresentam documentação e planejamento satisfatórios.

As conclusões e demais observações do modelo DSBOS serão debatidas no próximo capítulo.

5 Conclusões

O desenvolvimento de um modelo de processo de software baseado no open source não se mostrou uma tarefa fácil. Este trabalho iniciou analisando o estado da arte no desenvolvimento de modelos que representassem o conjunto de atividades que abrangem a produção de software. A grande variedade de modelos e suas diferentes abordagens comprovaram a idéia intuitiva de que o desenvolvimento de software está se tornando uma tarefa cada vez mais complexa. Os modelos de negócio estão em constante mutação, sendo que o uso de modelos de desenvolvimento é uma técnica para controlar o fluxo de execução das tarefas e garantir que o sistema final atenderá as necessidades iminentes dos seus usuários com qualidade e baixo overhead. Cada modelo estudado apresentou características e aplicações específicas, sugerindo que realmente não existe um modelo definitivo e universal para ser aplicado no desenvolvimento de software.

O estudo dos projetos open source demonstrou que realmente existem softwares de qualidade que estão sendo produzidos e largamente utilizados pelos diversos segmentos ligados à computação. O crescimento do uso do sistema operacional GNU/Linux, a utilização do servidor Apache, a utilização em conjunto da linguagem de programação para web PHP e do banco de dados relacional MySQL para produção de páginas com conteúdo dinâmico na Internet e a crescente difusão do navegador de páginas para Internet Mozilla Firefox são alguns exemplos que corroboram a idéia de softwares de sucesso. Grandes projetos indicam processos de desenvolvimento maduros e consistentes. A investigação realizada na literatura retornou diversas pesquisas que possibilitaram um estudo abrangente do processo de desenvolvimento de software open source. De acordo com o objetivo deste trabalho, foi possível identificar algumas lacunas no processo bem como práticas

interessantes de serem replicadas no modelo proposto. Ficou evidente a ausência de uma disciplina explícita de especificação de software, a falta de documentação sobre o planejamento do projeto e a conseqüente ausência de um cronograma de atividades. O problema é que nos projetos open source o desenvolvedor também é usuário, o que é inviável de implementar na maioria das equipes de desenvolvimento. A partir desta constatação é que este trabalho buscou abordar mais detalhadamente essa etapa. De fato, quando foi modelado o conjunto de atividades para a solicitação de novos requisitos por parte do usuário, na disciplina de evolução do software na seção 4.4, comprovou-se a importância da documentação produzida na especificação do software.

Em contrapartida, outras características do desenvolvimento open source foram escolhidas para estruturar o modelo DSBOS, como o desenvolvimento e validação do software com participação ativa do usuário e a utilização de ferramentas de comunicação pela Internet para prover a troca de mensagens entre os membros da equipe de desenvolvimento (tanto de forma síncrona como assíncrona) possibilitando assim a configuração de um ambiente distribuído de software. Também foi proposto um novo modo de atribuição de trabalho aos membros da equipe de desenvolvimento: cada membro se compromete a realizar um trabalho que desejava realizar de acordo com seus conhecimentos e motivações pessoais, ao invés de o gerente do projeto designar as tarefas, mantendo assim essa característica dos projetos open source. Definidos esses pontos, foi modelado um conjunto de atividades de acordo com as quatro disciplinas fundamentais ao processo de desenvolvimento de software, originando assim a proposta de modelo DSBOS.

A utilização da linguagem de metamodelo SPEM mostrou-se ideal para facilitar a descrição e compreensão de processos de desenvolvimento de software, visto que é estruturada na linguagem UML (orientada a objetos) e aplica uma notação própria com o uso de estereótipos. Estas características facilitaram não só a instanciação do processo como a representação das iterações entre as diversas atividades que o compõe. O único fator que dificultou a modelagem foi a ausência de uma ferramenta gratuita. A modelagem deste trabalho foi realizada através da ferramenta de modelagem UML Jude Community versão 2.5.1 e a aplicação da

notação dos estereótipos do SPEM ocorreu através do editor gráfico que acompanhava o sistema operacional do computador usado.

Alguns trabalhos futuros muito interessantes podem ser extraídos a partir deste trabalho acadêmico. Esta proposta de modelo baseado no processo open source pode incentivar novos estudos, utilizando-se outras abordagens para suprir as lacunas que existem neste tipo de processo de software. Com o advento da versão 2.0 do SPEM os processos de desenvolvimento de software serão ainda melhor descritos e um dos trabalhos possíveis seria a descrição do comportamento deste processo de desenvolvimento, enriquecendo este modelo proposto e aumentando sua compreensão. Devido ao fato de não ter sido encontrada uma ferramenta gratuita para a modelagem do processo na linguagem SPEM, também é sugerido o desenvolvimento de uma ferramenta open source que seja SPEM *compliant*, ou seja, que esteja em concordância com todas as determinações da especificação do metamodelo.

O trabalho futuro mais interessante seria a análise da aplicação deste modelo sugerido por uma equipe de desenvolvimento. A validação de um processo de software exige um tempo de aplicação e avaliação do processo que extrapola o limite de tempo de um trabalho acadêmico, mas que poderia ser desenvolvido em uma atividade de mestrado ou ser alvo de estudo de um grupo de pesquisa. Neste caso é fundamental a criação de uma ferramenta para instanciar e controlar projetos que adotarem este modelo proposto, viabilizando assim uma análise quantitativa do projeto. Devido ao fato de diversas ferramentas serem utilizadas ao longo do processo, um ambiente integrado de desenvolvimento orientado à comunicação usuário-desenvolvedor também seria de grande relevância devido à natureza distribuída que o processo permite.

Referências Bibliográficas

BECK, Kent. **Extreme Programming Explained: Embrace Change**. 1.ed. Addison Wesley, 2000.

BENNATAN, E.M. What is Happening to the Global Software Village?. **Agile Project Management Advisory Service, Executive Report**, v.3, n.1, Jan. 2002.

BÉZIVIN, J.; BRETON, E. Applying The Basic Principles Of Model Engineering to The Field of Process Engineering. **UPGRADE: The European Journal for the Informatics Professionals**. vol. V, n. 5, p. 27-33, Oct. 2004.

BOEHM, B. W. A spiral model of software development and enhancement. **IEEE Computer**; v. 21, p. 61-72, May 1988.

FELLER, J.; FITZGERALD, B. A framework analysis of the open source software development paradigm. In: INTERNATIONAL CONFERENCE ON INFORMATION SYSTEMS, 21, 2000, Austrália. **Proceedings of the...** Austrália: ICIS, p.58-69.

FOGEL, Karl. **Producing Open Source Software**. 1.ed. 2005.

FUGGETTA, A. Software Process: A Roadmap. In: INTERNATIONAL CONFERENCE ON THE FUTURE OF SOFTWARE ENGINEERING. 22., 2000, Limerick. **Proceedings of the...** Limerick: ICSE ’00; ACM Press, p. 25-34.

HIRSCH, M. Making RUP agile. **Conference on Object Oriented Programming Systems Languages and Applications - Pratitioner Report**; p.1-8, 2002.

KRISHNAMURTHY, Sandeep. Cave or Community? An empirical examination of 100 Mature Open Source Projects. **First Monday**, v.7, n. 6, jun. 2002.

LINGER, R. C. Cleanroom process model. **IEEE Software**. v.11. n.2. Mar. 1994. p.50-58.

LONCHAMP, Jacques. A Structured Conceptual and Terminological Framework for Software Process Engineering. In: INTERNATIONAL CONFERENCE ON THE SOFTWARE PROCESS, 2., 1993, Berlin. **Proceedings of the...** Berlin: ICSP2, IEEE Computer Society Press, p. 41-53.

LONCHAMP, Jacques. Open Source Software Development Process Modeling. In: **Software Process Modeling**. Spring, 2005, p.1-33.

MÄKILÄ, T.; JÄRVI, A. Spemmet - A Tool for Modeling Software Processes with SPEM. In: INTERNATIONAL CONFERENCE ON INFORMATION SYSTEMS IMPLEMENTATION AND MODELLING, 9., 2006, Přerov, Czech Republic. **Proceedings of the...** Přerov, Czech Republic : ISIM, 2006, p. 87-94.

MARTINS, P. V.; SILVA, A. R. Comparação de Metamodelos de Processos de Desenvolvimento de Software. In: CONFERÊNCIA PARA A QUALIDADE NAS TECNOLOGIAS DA INFORMAÇÃO E COMUNICAÇÕES, 5., 2004, Porto. **Actas da...** Porto: QUATIC, 2004. p.179-186.

MAYRING, Philipp. **Introdução à Pesquisa Qualitativa. Uma introdução para pensar qualitativamente.** 5a ed. Weinheim: Beltz, 2002.

MOCKUS, A.; FIELDING, R. T.; HERBSLEB, J. D. A Case Study of Open Source Software Development: The Apache Server. In: INTERNATIONAL CONFERENCE OF SOFTWARE ENGINEERING, 22., 2000, Limerick. **Proceedings of the...** Limerick, ICSE'00, ACM Press, p.263-272.

MOCKUS, A.; FIELDING, R. T.; HERBSLEB, J. D. Two Case Studies of Open Source Software Development: Apache and Mozilla. **ACM Transactions on Software Engineering and Methodology**; v.11, n.3, p.309-346, July 2002.

NEWKIRK, J. Introduction to Agile Process and Extreme Programming. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 24., 2002, Orlando. **Proceedings of the...** Orlando, ICSE 2002, ACM Press, p. 695-696.

O'REILLY, Tim. Lessons from Open Source Software Development. **Communications of the ACM**, v.42, n.4, p.33-37, abr. 1999.

PRESSMAN, Roger S. **Engenharia de Software**. 5.ed. Rio de Janeiro: McGraw-Hill, 2002.

PROWELL, S. J. et al. **Cleanroom Software Engineering: Technology and Process**. Reading, MA: Addison-Wesley. 1999.

RAYMOND, Eric S. **The Cathedral and the Bazaar. Musings on Linux and Open Source Software by an Accidental Revolutionary**. Revised edition. O'Reilly, 2001.

REIS, Christian Robottom. **Caracterização de um Processo de Software para Projetos de Software Livre.** 2003. 158f. Dissertação (Mestrado em Ciências da Computação e Matemática Computacional)-Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos.

REIS, C. R.; FORTES, R. P. M. An Overview of the Software Engineering Process and Tools in the Mozilla Project. In: WORKSHOP ON OPEN SOURCE SOFTWARE DEVELOPMENT, 2002, Newcastle. **Proceedings of...** Newcastle, OSSDW, p.162-182.

SOMMERVILLE, Ian. **Engenharia de Software**. 6.ed. Addison Wesley, 2003.

SPEM. **Software Process Engineering Metamodel Specification**. v1.1. Object Management Group, jan. 2005.

VANDERBURG, G. A Simple Model of Agile Software Processes -- or -- Extreme Programming Annealed. In: ACM SIGPLAN CONFERENCE ON OBJECT ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES AND APPLICATIONS, 20., 2005, San Diego. **Proceedings of the...** San Diego, OOPSLA'05, p.539-545.