Caracterização de um Modelo de Processo para Projetos de Software Livre

Christian Reis
kiko@acm.org

Orientação:
Profa. Dra. Renata Pontin de Mattos Fortes
renata@icmc.sc.usp.br

Monografia apresentada ao Instituto de Ciências Matemáticas e de Computação para o Exame de Qualificação, como parte dos requisitos para a obtenção do título de Mestre na Área de Ciências da Computação e Matemática Computacional.

São Carlos, São Paulo
Abril de 2001

Resumo

Software Livre, que neste texto abrange software também conhecido como Open Source, é software que é fornecido acompanhado de código fonte e que pode ser livremente modificado e redistribuído. Uma conseqüência indireta desta liberdade é o aparecimento de comunidades de desenvolvimento de software que trabalham de forma descentralizada por meio da Internet, desenvolvendo e mantendo os diferentes projetos de software livre. Estas comunidades, à primeira vista, parecem estar caoticamente organizadas do ponto de vista de um processo de engenharia de software; no entanto, grande parte do software produzido é de alta qualidade, assim como é alta a produtividade e satisfação dos desenvolvedores. O objetivo deste trabalho é verificar esta aparente inconsistência, e desenvolver um modelo de processo para este tipo de software.

Palavras-Chave: Software Livre, Open Source, Modelo de Processo de Software, Engenharia de Software, Desenvolvimento de Software Descentralizado.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Engenharia de Software e Software Livre são termos que são raramente abordados de forma associada em trabalhos científicos; no entanto, esforços combinados nestas áreas têm grande potencial para enriquecer o conhecimento geral do processo de desenvolvimento de software. Neste capítulo é feita uma breve apresentação dos temas e objetivos do trabalho.

1.1 Contextualização

A dependência e demanda crescentes da sociedade em relação à Informática e, em particular, a Software, tem ressaltado uma série de problemas relacionados ao processo de desenvolvimento de software: alto custo, alta complexidade, dificuldade de manutenção, e uma disparidade entre as necessidades dos usuários e o produto desenvolvido [1, 2, 3].

Estes problemas afligem a área de Software desde sua criação; Pressman os identifica como uma 'aflição crônica', e não uma crise pontual [4]. Para administrá-los, a comunidade de Engenharia de Software, ao longo dos últimos 30 anos, estuda e implementa práticas de desenvolvimento de software bem organizadas e documentadas.

Parte do trabalho dos pesquisadores envolvidos com a Engenharia de Software tem sido descrever e abstrair modelos que descrevem processos de software. Estes modelos permitem que se compreenda o processo de desenvolvimento dentro de um paradigma conhecido. A existência de um modelo é apontada como um dos primeiros passos em direção ao gerenciamento e à melhoria do processo de software [5].

A maior parte dos modelos existentes é baseada em premissas bastante tradicionais da área: uma divisão discreta das atividades do processo de software, focando em gerenciamento, medidas e registros destas atividades. No entanto, não existe um modelo uniforme que possa descrever com precisão o que de fato acontece durante todas as fases da produção de um software; os processos implementados são muito variados, e as necessidades de cada organização diferem substancialmente [6].

Além disso, na última década, um segmento crescente da comunidade de Engenharia de Software vem defendendo a existência de problemas fundamentais da aplicação sistemática e institucionalizada de processos de software convencionais [7, 8, 9] . Estes proponentes advogam processos simplificados, focados nas pessoas que compõem o processo, e principalmente no programador. Nas palavras de Bach:

‘Nas conferências e nos periódicos, a atenção extraordinária dada ao processo de desenvolvimento de software é mal dirigida. Demasiado é escrito sobre processos e métodos para desenvolver software; muito pouco sobre o cuidado e alimentação das mentes que de fato escrevem o software.”

Na última década, a popularização de softwares classificados como Software Livre [10], que são distribuídos livremente acompanhados de código fonte, tem gerado curiosidade por parte de pesquisadores e profissionais de alguma forma envolvidos com software. Esta curiosidade em parte se deve à forma particularmente simples pela qual estes softwares são produzidos.

Apesar de apresentarem alta confiabilidade, evolução rápida, e uma comunidade produtiva [11, 12, 13], os projetos de software livre apresentam um processo de desenvolvimento aparentemente caótico, com algumas características particulares: total descentralização, política liberal em relação a alterações, e discussão aberta sobre todo assunto técnico pertinente ao projeto [14]. Os projetos sobrevivem utilizando um mínimo de infra-estrutura provida pela Internet, sendo o email a ferramenta mais utilizada [15].

Em 1998, Raymond publicou um conjunto de artigos que descrevem informalmente algumas características do processo de desenvolvimento que muitos projetos seguem [14, 16], as mais importantes sendo: *releases*[1] freqüentes, propriedade de código compartilhada, e revisão e verificação de código maciça. Nestes artigos, Raymond utiliza o termo “Bazar”para descrever a forma de organização dos projetos.

Os artigos de Raymond geraram uma série de críticas e respostas, e ele é considerado um pioneiro por ser o primeiro a documentar, ainda que informalmente, um processo para este tipo de software. Apesar da recepção pela comunidade de Engenharia de Software aos conceitos apresentados em [14] não ser totalmente positiva [17, 18, 19], a indústria tem reconhecido o potencial do software livre e apoiado substancialmente os principais projetos.

Nos últimos anos, IBM, Apple, Sun, Netscape/AOL e Microsoft têm reconhecido e investido em projetos de software livre [20, 21, 22, 13]. Em 2001, será realizada o primeiro Workshop para Desenvolvimento Open Source como parte do International Conference of Software Engineering, em Toronto, Canadá, e é a primeira vez que a conferência tratará deste assunto em um fórum específico [23].

O interesse da indústria e da comunidade acadêmica não é sem motivo: em alguns segmentos importantes, projetos de software livre são líderes de mercado. O servidor Web mais utilizado atualmente é o Apache HTTPD, que é software livre [24]. Edwards atribui 80% do volume total de email Internet ao Sendmail, um software livre para transmissão de correio eletrônico [12]. O núcleo de sistema operacional Linux, associado a um conjunto enorme de componentes de software livre, tem mais de 10 milhões de instalações estimadas mundialmente[25].

A popularização do Linux, associada à criação de um conjunto de publicações e sites Web, gerou interesse por software livre em milhares de desenvolvedores ao redor do mundo. O Freshmeat.net, um site Web dedicado a projetos de software livre, atualmente arquiva uma lista de mais de 1.000 projetos em desenvolvimento [26]. Este conjunto de softwares representa um tema muito interessante para estudo, especialmente levando em conta a natureza aberta e transparente destes projetos.

[1] Um *release* é o lançamento de uma versão consolidada de um software.

O aparente paradoxo entre os aspectos positivos de software livre e sua estrutura pouco formal traz a tona uma questão fundamental: de que forma estes projetos se organizam, e como podem ter sucesso frente ao aparente caos em que existem?

1.2 Motivação

Dos trabalhos científicos publicados que tratam do assunto Software Livre, poucos abordam o assunto de modelo de processo e metodologia; no entanto, deve existir algum processo pelo qual software deste tipo é desenvolvido. Por este motivo, há oportunidade de realizar um trabalho potencialmente inovador.

Como descrito na Introdução, o assunto é uma área fértil para pesquisa e experimentalismo; existem centenas de projetos operando de forma aberta e que podem ser analisados de forma conveniente. Além disso, a comunidade tem se mostrado bastante aberta a pesquisadores interessados em compreender o seu funcionamento [27].

Uma motivação adicional para o trabalho é tornar conhecido este processo entre os cientistas brasileiros. Software livre é um assunto que tem especial importância para países em desenvolvimento; no caso particular do Brasil, no entanto, é ainda pouco explorado. Se for possível reproduzir o sucesso destes projetos internacionais em empresas e instituições acadêmicas brasileiras, uma oportunidade para promover um avanço importante na área de software pode ser gerada.

1.3 Objetivos

Este trabalho tem como objetivo levantar as características principais , do ponto de vista do processo de software, a partir de um conjunto de projetos de software livre. O trabalho busca determinar se estas características bastam para compor um modelo de processo de software para software livre, e caso seja possível, descrever este modelo.

Como conseqüência do levantamento destas características, devem ser identificadas, dentre o conjunto total de práticas aplicadas entre os diferentes projetos, uma coleção de melhores práticas de desenvolvimento de software livre. Além disso, será formada uma base de dados descrevendo o processo, permitindo que outros pesquisadores e organizações possam aproveitar o trabalho realizado.

1.4 Organização da Monografia

Este texto está dividido em cinco capítulos principais, sendo o primeiro esta Introdução. A revisão bibliográfica é abordada nos dois capítulos seguintes: o segundo capítulo trata da teoria desenvolvida na área de Processos de Software, dentro da Engenharia de Software; o terceiro capítulo descreve, com base na literatura existente, os aspectos principais referentes a software livre e o desenvolvimento de software baseado neste princípio. Este capítulo traz um comentário dos trabalhos relacionados.

O quarto e último capítulo descreve o projeto proposto, e aborda a metodologia que se pretende utilizar para executar o projeto. Esta parte inclui um detalhamento e cronograma das atividades essenciais.

Capítulo 2

O Processo de Software

Muito do que será investigado neste trabalho diz respeito a um Processo de Software, e para defini-lo, cabe alguma discussão. Existe alguma sobreposição em relação aos termos Processo, Modelo, Método e Metodologia, gerando confusão em algumas circunstâncias. Embora não sejam sinônimos, é comum observar na literatura o uso de um termo em lugar do outro. Assim, é necessário buscar definições para fundamentar o objetivo deste trabalho, que envolve o entendimento de um processo para software livre.

Neste trabalho, é usada a palavra modelo para descrever um enfoque conceitual ao Processo de Software; é usado o termo metodologia para descrever uma seqüência de atividades práticas a ser executadas durante o desenvolvimento.

2.1 Definições

O Processo de Software é definido por Sommerville em [28]:

> "[O processo é] um conjunto de atividades e resultados associados que produzem um produto de software."

Pressman, em [29] oferece a seguinte definição:

> "[...definimos um] processo de software como um *framework* para as tarefas que são necessárias para a construção de software de alta qualidade."[1]

Estas definições oferecem uma idéia mais clara do que é considerado um processo. Diretamente delas, podemos retirar os seguintes pontos importantes:

- O processo reúne um conjunto de atividades. Como existem atividades que englobam outras atividades, vamos usar o termo fase para descrever atividades de nível mais alto.
- O processo tem como objetivo desenvolver um produto de software. Pressman [2] restringe o termo a processos que geram "produtos de alta qualidade", mas se ignoramos esta restrição, podemos aplicar o termo a qualquer conjunto de atividades que é aplicada com o objetivo de desenvolver software.
- A definição não se restringe a processos estruturados; qualquer forma de desenvolvimento constitui um processo, por mais imatura ou caótica que possa ser.

[1] O autor segue com a seguinte indagação: "Um 'processo' é sinônimo de engenharia de software? Sim e Não."

Não há processo correto ou incorreto; dependendo da sua aplicação, ambiente e objetivo, o uso de um processo específico pode ser vantajoso ou não. Um ponto importante a ressaltar é que cada autor e organização coloca e classifica processos e atividades de forma diferente, tornando difícil uma uniformidade completa. As seções seguintes discutem visões alternativas, e se baseiam em características comuns encontradas na literatura para classificar.

2.2 Fases do Processo de Software

Pela definição, podemos entender o que é o processo; no entanto, de que fases é composto?

Em meados dos anos 70, Schwartz já apontava como fases principais do processo de produção de um sistema de software [30]:

1. Especificação de Requisitos: tradução da necessidade ou requisito operacional para uma descrição da funcionalidade a ser executada.

2. Projeto de Sistema: tradução destes requisitos em uma descrição de todos os componentes necessários para codificar o sistema.

3. Programação (Codificação): produção do código que controla o sistema e realiza a computação e lógica envolvida.

4. Verificação e Integração (*Checkout*): verificação da satisfação dos requisitos iniciais pelo produto produzido.

A definição moderna oferecida por Sommerville [28] é similar; define as atividades como Especificação, Desenvolvimento, Validação e Evolução. Este último ponto não é descrito diretamente por Schwartz:

5. Evolução: alteração do Software para atender a novas necessidades do usuário.

É interessante observar que é destacada a questão da longevidade do software no ítem Evolução; neste contexto, podemos perceber que a definição de Schwartz omite uma particularidade importante: o software continua sendo desenvolvido mesmo depois de entregue. Pressman [2] oferece uma visão compatível com esta, ainda que simplificada: as fases descritas são Definição, Desenvolvimento e Manutenção.

É importante ressaltar que não existe uma seqüencialidade obrigatória de fases. Tradicionalmente, as fases tem sido vistas como passos discretos e seqüenciais [31]; no entanto, diversos autores apontam a natureza não-simultânea das fases como uma realidade na aplicação de processos de software [32, 33].

Além da questão de seqüencialidade, existem autores que defendem que o processo de software é muito mais iterativo e cíclico do que a idéia de fases simples pode sugerir. Em particular, Boehm, Davis, Rising et Al., e Beck sugerem processos onde existem ciclos contínuos e repetidos, onde alguma forma de produto é desenvolvida a cada ciclo [34, 32, 35, 8]. A extensão de cada ciclo, no entanto, não é um consenso.

2.3 Atividades do Processo de Software

Para cada fase do processo de desenvolvimento de software existe uma série de atividades que são executadas. Estas atividades constituem um conjunto mínimo para se obter um produto de software, segundo Pressman [2]. Observando as fases individuais e suas atividades associadas, combinando classificações de Schwartz, Pressman e Sommerville [30, 2, 28], é possível identificar as seguintes atividades:

1. Especificação
   (a) Engenharia de Sistema: estabelecimento de uma solução geral para o problema, envolvendo questões extra-software.
   (b) Análise de Requisitos: levantamento das necessidades do software a ser implementado. A Análise tem como objetivo produzir uma especificação de requisitos, que convencionalmente é um documento.
   (c) Especificação de Sistema: descrição funcional do sistema. Pode incluir um plano de testes para verificar adequação.
2. Projeto[2]
   (a) Projeto Arquitetural: onde é desenvolvido um modelo conceitual para o sistema, composto de módulos mais ou menos independentes.
   (b) Projeto de Interface: onde cada módulo tem sua interface de comunicação estudada e definida.
   (c) Projeto Detalhado: onde os módulos em si são definidos, e possivelmente traduzidos para pseudo-código.
3. Implementação
   (a) Codificação: a implementação em si do sistema em uma linguagem de computador.
4. Validação
   (a) Teste de Unidade e Módulo: a realização de testes para verificar a presença de erros e comportamento adequado a nível das funções e módulos básicos do sistema.
   (b) Integração: a reunião dos diferentes módulos em um produto de software homogêneo, e a verificação da interação entre estes quando operando em conjunto.
5. Manutenção e Evolução
   (a) Nesta fase, o software em geral entra em um ciclo iterativo que abrange todas as fases anteriores.

O Processo de Software pode ser visto como um gerador de produtos, sendo que o produto final, ou principal, é o Software em si. É importante perceber que existem subprodutos que são gerados para cada fase — ao final da fase de Especificação, por exemplo, é comum ter sido desenvolvido e entregue um ou mais documentos que detalham os requisitos do sistema. Estes subprodutos também são chamados na literatura de *deliverables*.

[2] Projeto, Implementação e Validação podem ser vistos como uma fase única de Desenvolvimento.

Todo modelo de software deve levar em consideração as fases descritas; no entanto, cada um organiza estas fases de uma forma particular de acordo com sua filosofia de organização. Na próxima seção são analisados alguns modelos mencionados na literatura.

2.4 Modelos de Processo de Software

Existem alguns modelos teóricos desenvolvidos que buscam descrever a forma com que as fases seguem e interagem. Nesta seção estão descritos alguns dos modelos mais conhecidos [28, 2, 31, 34]. Existe alguma flexibilidade no que diz respeito à definição do termo "modelo"; neste trabalho são considerados, dentre os modelos descritos na literatura, os que têm um caráter estratégico, e não específico. Em outras palavras, um modelo é uma filosofia do andamento das fases, e não uma descrição de como cada atividade deve ser executada.

A seção 2.5, por sua vez, descreve algumas metodologias, que são formas práticas de organizar o processo de desenvolvimento. Uma metodologia traz conceitos bastante específicos em relação ao desenvolvimento, como exemplifica Beck [8] em sua descrição da metodologia XP — programação em pares, ciclos de 15 dias e equipes de menos de 10 pessoas.

2.4.1 O Modelo Cascata

Este modelo foi idealizado em 1970 por Royce, e tem como característica principal a seqüencialidade das atividades: sugere um tratamento ordenado e sistemático ao desenvolvimento do software. Cada fase transcorre completamente e seus produtos são vistos como entrada para a nova fase; o software é desenvolvido em um longo processo e entregue ao final deste. O autor sugere laços de *feedback*, que permitem realimentar fases anteriores do processo, mas em geral o modelo cascata é considerado um modelo linear [31]. A figura 2.1 fornece uma descrição visual do modelo.

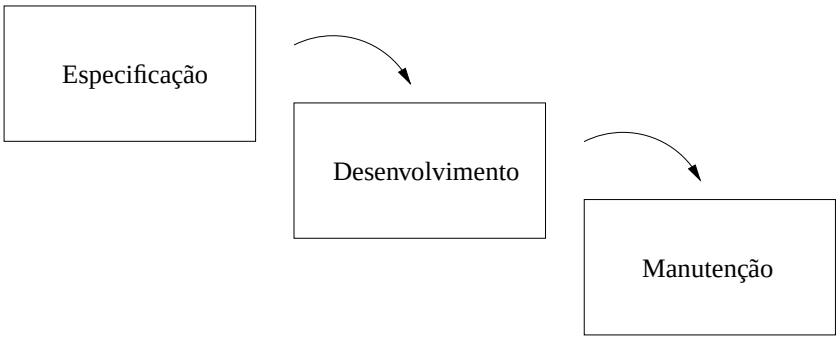

Figura 2.1: Diagrama simplificado do modelo Cascata

Críticas ao modelo Cascata sugerem a inadequação deste a processos reais; em geral, há muito intercâmbio de informações entre as fases, e raramente ocorrem projetos onde não há concorrência das fases em si. [32]. Além disso, o modelo Cascata não leva em consideração questões modernas importantes ao desenvolvimento: prototipação, aquisição de software e alterações constantes nos requisitos, por exemplo [8, 36].

2.4.2 O Modelo Espiral

Boehm, em 1986, sugeriu um modelo evolucionário para o desenvolvimento de software, baseado em uma seqüência de fases que culminam em *releases* incrementais do software [34]. Esta característica incremental é confirmada no conceito de prototipação de Brooks [37], e em metodologias de processo mais modernas, como descrito por Beck [8] e Rising [35].

Em geral, modelos incrementais têm o objetivo de lidar melhor com um conjunto de requisitos incerto ou sujeito a alterações. Gancarz, por exemplo, afirma que a filosofia original do Sistema Operacional Unix leva em conta o fato de requisitos raramente se adequarem ao produto inicial [38]. Por este motivo, o modelo espiral parece melhor adequado a projetos reais que o modelo cascata — em geral, modela melhor a interação real entre cliente e fornecedor do software [34]. A figura 2.2 apresenta um diagrama do modelo.

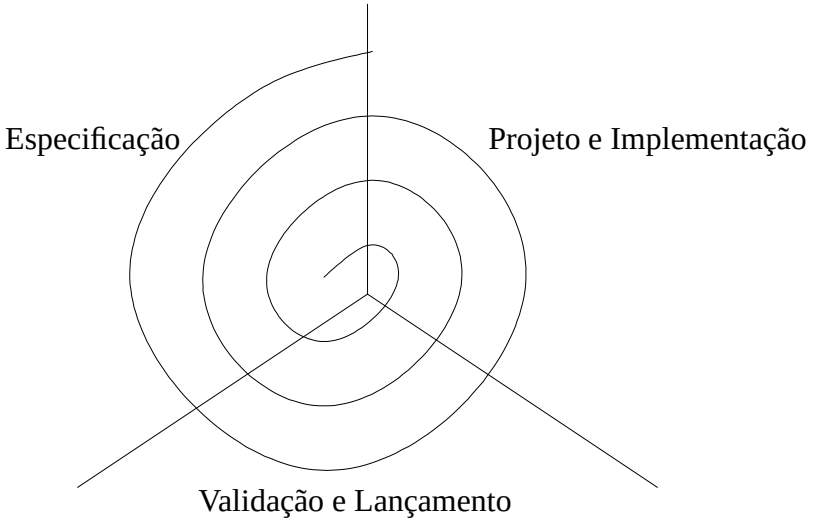

Figura 2.2: Diagrama simplificado do modelo Espiral

O Modelo Espiral, ainda assim, assume que existe alguma seqüencia entre as fases: não há suporte para fases que ocorrem simultaneamente, ou que necessitam de intercomunicação contínua para operarem.

2.4.3 Um Modelo baseado em Componentes Comerciais

Embora Brooks, em [39], tenha defendido a compra de componentes de software como um dos meios para melhorar a produtividade substancialmente, Hirai e Saeki apontam como uma dificuldade significativa, em projetos de software modernos, a interação da organização desenvolvedora com seus fornecedores de software [36]. Um dos maiores riscos de desenvolvimento apontados por Lawson em [40] é a interdependência de um produto com componentes de software produzidas em outras organizações, sobre as quais não se tem controle algum.

O modelo proposto por Hirai e Saeki utiliza dois mecanismos externos que não são contemplados nas fases tradicionais: um Sensor, que captura informações da Internet que tratam de atualização e problemas nos componentes comerciais utilizados; e um Controlador de Processo que fornece aos

membros do projeto informação relacionada aos componentes utilizados. Existe uma base de dados onde ficam armazenadas informações relacionadas aos componentes e ao seu uso na organização.

O Controlador alimenta todas as fases do desenvolvimento, da Especificação à Integração e Teste, com dados referentes aos componentes. Na fase de Especificação, por exemplo, o Controlador fornece características e recursos dos componentes utilizáveis, para que a equipe possa decidir pelo uso de algum. A figura 2.3 mostra uma descrição simplificada este modelo.

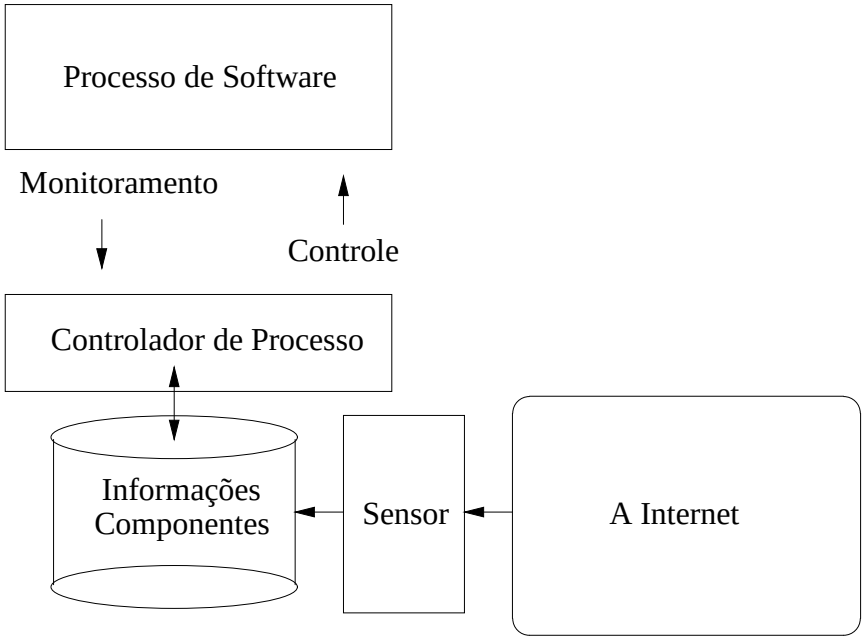

Figura 2.3: Diagrama simplificado do modelo baseado em Componentes Comerciais

Este modelo é interessante por abordar um fato até então ignorado nos modelos de processo: que componentes comercialmente disponíveis podem influenciar substancialmente o processo de desenvolvimento. Os padrões de qualidade, prazo e ritmo de lançamento dos componentes utilizados influem diretamente na forma com a qual o software é desenvolvido dentro de uma organização.

2.4.4 O Modelo Concorrente

Davis e Sitaram defendem um outro ponto importante: o de que as fases de um processo de desenvolvimento não ocorrem seqüencialmente, e sim, concorrentemente. A descrição do modelo é feita usando um estudo de caso modelado em Statecharts [32]. O mecanismo pelo qual o processo ocorre é baseado em eventos que sinalizam alterações de estado dentro de cada fase. O modelo representa atividades simultâneas de todos os membros da equipe de desenvolvimento, e os eventos que alteram o estado são gerados por necessidades do usuário, decisões da gerência, e resultados de revisões técnicas.

Por exemplo, a fase de Especificação pode estar em um dentre diversos estados: em desenvolvimento, completo, revisado, e controlado no repositório. A criação, e toda revisão à especificação original, ativa a fase de Desenvolvimento, de forma que há um ciclo constante entre os estados. A figura 2.4 apresenta um diagrama de estados para esta fase.

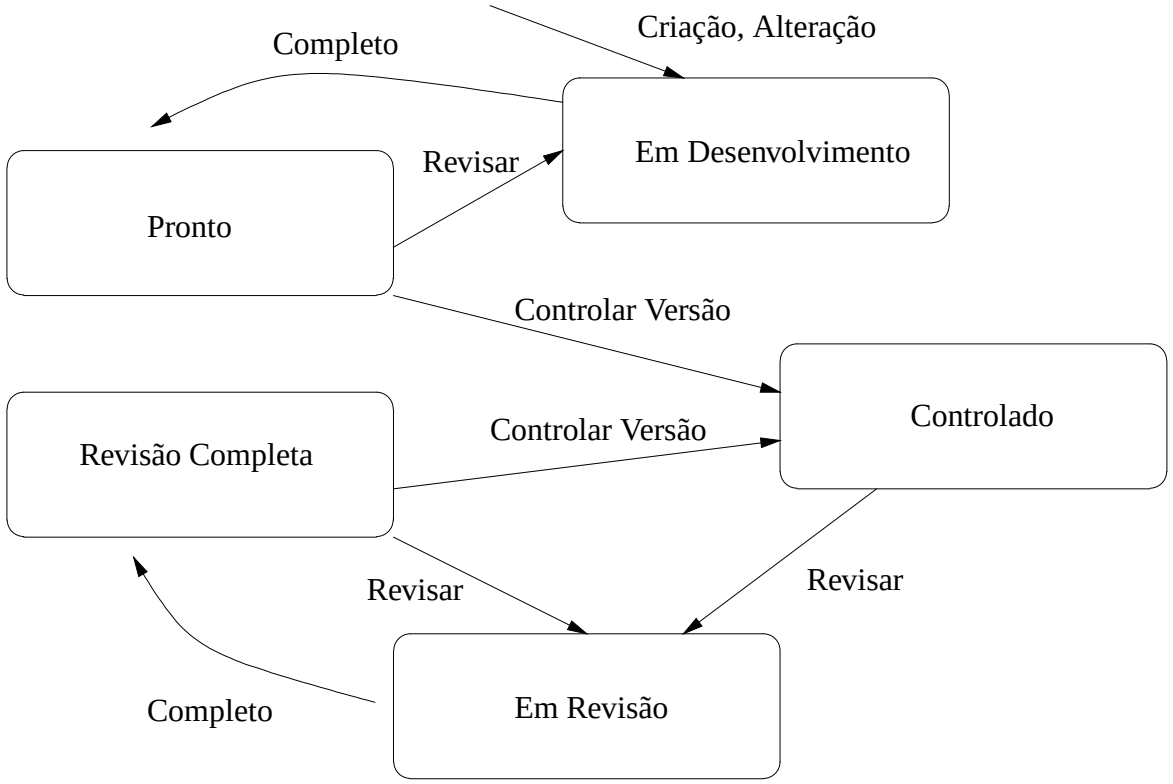

Figura 2.4: Diagrama simplificado da fase de Especificação do modelo Concorrente

Embora o modelo apresente complexidade elevada, é bem adaptado a situações reais de desenvolvimento, onde realmente existe uma divisão temporal menos discreta entre as fases em execução. O modelo caótico, descrito a seguir, elabora em outro nível esta filosofia.

2.4.5 O Modelo Caótico

Este modelo foi sugerido por Racoon, que coloca o programador como a figura mais importante do seu modelo de processo [33]. Este sentimento é confirmado por DeMarco e Bach em seus trabalhos [7, 9].

A descrição do modelo é interessante e espirituosa. O princípio básico é que cada uma das fases de desenvolvimento descritas na seção 2.2 consiste na realidade em um ciclo de vida completo. Cada fase do ciclo é composta, por sua vez, de um conjunto de fases idênticos. Por exemplo, a fase de Especificação é um ciclo onde consideramos como implementar a especificação, a implementação em si, e a evolução desta especificação. A figura 2.5 apresenta um ciclo para esta fase.

Além disto, Racoon coloca, “cada fase ocorre em todas as fases”. Na fase de Especificação, são analisados os requisitos para criar um documento de requisitos; na fase de Implementação, são criados requisitos de implementação; na fase de Manutenção, são revisados e alterados os requisitos iniciais. Este padrão fractal se repete para todas as fases.

Embora o autor seja informal em boa parte de sua descrição, seu enfoque é interessante no sentido de que enuncia a complexidade como uma realidade do processo de software, e não um problema a ser resolvido. Em parte, este é um motivo pelo qual as metodologias mais específicas focam em aspectos

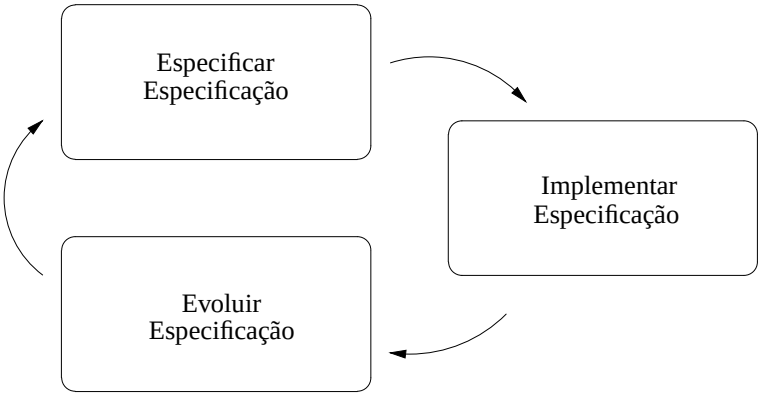

Figura 2.5: Diagrama simplificado da fase de Especificação do modelo Caótico

práticos do desenvolvimento, e não em questões filosóficas profundas. Na próxima seção são analisadas algumas metodologias de desenvolvimento, também chamadas de processos leves (*lightweight processes*) ou processos ágeis [8, 41].

2.5 Metodologias Ágeis

Os modelos apresentados anteriormente têm como objetivo prover um mapa conceitual do processo de desenvolvimento. Existe discussão até que ponto modelar e especificar um processo é relevante [42, 43]; o certo é que modelos abordam essencialmente questões conceituais da organização do desenvolvimento. Fowler coloca em [41] que as metodologias modernas de desenvolvimento, como XP e SCRUM, são uma reação a modelos extremamente conceituais e a metodologias "monumentais" [41]. Nas suas palavras:

"Estas metodologias [monumentais] existem há muito tempo. Elas não são conhecidas por serem particularmente de sucesso [...] A crítica mais freqüente a estas metodologias é que são burocráticas [...] Como uma reação a estas metodologias, um grupo novo de metodologias apareceu nos últimos anos [...] Estes novos métodos tentam estabelecer um compromisso útil entre nenhum processo e processo demasiado, provendo apenas processo suficiente para fornecer uma vantagem razoável."

Nesta seção estão descritas metodologias práticas para o desenvolvimento, que são formas específicas de organizar o processo de software para obter vantagens de qualidade e produtividade. A metodologia se foca em especificar tarefas aplicando um conceito específico (*refactoring* e programação em pares em XP, por exemplo) a cada fase do desenvolvimento, e propor soluções práticas para problemas comuns.

2.5.1 Extreme Programming (XP)

O trabalho de Beck [8] descreve um processo minimalista onde existe muito pouca burocracia envolvida no desenvolvimento. Equipes pequenas, de até 10 desenvolvedores, trabalham em iterações curtas, produzindo software incrementalmente, e analisando requisitos à medida que são descritos pelo cliente.

XP se apóia em um contínuo refinamento do projeto e da implementação deste no código. Para realizar este refinamento, aplica alguns princípios básicos:

- Propriedade Coletiva: Todos os desenvolvedores são responsáveis por qualquer parte do software, e têm liberdade para alterar e reparar o código caso seja necessário.
- Programação em Duplas: dois programadores trabalham juntos no mesmo teclado e monitor.
- *User Stories*: requisitos são elaborados na forma de cenários de operação do software pelo usuário, e o software é implementado a partir destes cenários.
- *Refactoring*: o código é continuamente alterado para que se reduza à sua forma mais simples possível.
- Testes de Unidade: a programação é iniciada por testes de unidade, que são implementados *antes* do código que devem testar. A coleção de testes permite aos desenvolvedores alcançarem confiança o suficiente para continuamente alterar o código implementado.

O interesse que XP tem gerado entre a comunidade de desenvolvimento — este é o segundo ano em que a Extreme Programming Conference é realizada [44] — pode ter relação com sua atitude de "menor esforço" face a problemas complexos do desenvolvimento. Por exemplo, a documentação do sistema deve ser mantida junto com o próprio código fonte, e deve ser mínima, forçando o desenvolvedor a escrever código auto-explicativo [45] e a evitar complexidade.

2.5.2 SCRUM

Uma outra metodologia de desenvolvimento que é classificada como ágil é o SCRUM[3]. Esta metodologia foi criada na Easel, e posteriormente desenvolvida por duas empresas em conjunto: Advanced Development Methods e VMARK. Seu objetivo é fornecer um processo conveniente para projeto e desenvolvimento orientado a objeto [46].

A metodologia é baseada em princípios semelhantes aos de XP: equipes pequenas, requisitos pouco estáveis ou desconhecidos, e iterações curtas para promover visibilidade para o desenvolvimento. No entanto, as dimensões em SCRUM diferem de XP.

SCRUM divide o desenvolvimento em *sprints* de 30 dias. Equipes pequenas, de até 7 pessoas, são formadas de projetistas, programadores, engenheiros e gerentes de qualidade. Estas equipes trabalham em cima de funcionalidade (os requisitos, em outras palavras) definidas no início de cada sprint. A equipe é responsável pelo desenvolvimento desta funcionalidade.

Todo dia, é feita uma reunião de 15 minutos onde o time expõe à gerência o que será feito no próximo dia, e nestas reuniões os gerentes podem levantar os fatores de impedimento (*bottlenecks*), e o progresso geral do desenvolvimento.

SCRUM é interessante porque fornece um mecanismo de informação de status que é atualizado continuamente, e porque utiliza a divisão de tarefas dentro da equipe de forma explícita. Uma discussão mais profunda do método é feita em [47].

2.5.3 Crystal/Clear

Crystal/Clear faz parte, na realidade, de um conjunto de metodologias criado por Cockburn [48]. As premissas apresentadas para a existência deste conjunto são:

[3] A palavra Scrum vem do nome de uma tática utilizada em rugby para recuperar uma bola perdida.

- Todo projeto tem necessidades, convenções e uma metodologia diferentes.
- O funcionamento do projeto é influenciado por fatores humanos, e há melhora neste quando os indivíduos produzem melhor.
- Comunicação melhor e lançamentos freqüentes reduzem a necessidade de construir produtos intermediários do processo.

Crystal/Clear é uma metodologia direcionada a projetos pequenos, com equipes de até 6 desenvolvedores. Assim como com SCRUM, os membros da equipe tem especialidades distintas. Existe uma forte ênfase na comunicação entre os membros do grupo, e a organização do espaço de trabalho deve permitir este tipo de trabalho.

Toda a especificação e projeto são feitos informalmente, utilizando quadros publicamente visíveis. Os requisitos são elaborados utilizando *use cases*, um conceito similar às *user stories* em XP, onde são enunciados os requisitos como tarefas e um processo para sua execução. Os *releases* de software são feitos em incrementos regulares de um mês, e existem alguns subprodutos do processo que são responsabilidade de membros específicos do projeto.

Grande parte da metodologia é pouco definida, e segundo o autor, isto é proposital; a idéia de Crystal/Clear é permitir que cada organização implemente as atividades que lhe parecem adequadas, fornecendo um mínimo de suporte útil do ponto de vista de comunicação e documentos.

2.6 Sobre Atividades Auxiliares

Nesta seção é retomada a descrição das atividades que são realizadas durante um processo genérico de software; estas atividades se aplicam tanto a modelos conceituais, quanto às metodologias práticas abordadas na última seção.

Além das atividades envolvidas com a produção do software em si, existem atividades auxiliares, citadas por Pressman [29] como *umbrella activities*. Estas atividades existem com o objetivo de garantir que o software construído seja de qualidade e bem documentado, e que a produtividade do grupo de desenvolvimento seja mantida.

Atividades auxiliares do processo são aplicadas ao longo de todas as fases do desenvolvimento. O conjunto específico de atividades varia de acordo com cada organização, mas existe um consenso de que as seguintes atividades são importantes [29, 49, 28]:

- Gerenciamento de Configuração: um registro de alterações e motivos para estas para cada subproduto do processo de desenvolvimento, incluindo o projeto do sistema, requisitos, documentação e o código em si.
- Garantia da Qualidade: um processo completo, abrangendo atividades que existem para manter qualidade no processo; inclui revisões técnicas (inclusive de código, projeto, requisitos), medição do processo de desenvolvimento e análise destas medidas, elaboração da estratégia de testes, e elaboração e aplicação do processo de software em si.
- Acompanhamento e Controle de Progresso: um mecanismo para explicitar o estado atual do projeto, comunicar este estado aos diferentes membros do projeto, e aplicar alterações ao cronograma, requisitos e organização do grupo de desenvolvimento para garantir produtividade.

- Documentação: a atividade de registrar em língua natural instruções, detalhes e explicações a respeito de cada um dos produtos do processo de software.

Todo processo de software tem benefícios com a aplicação destas atividades, e em muitos casos, mesmo inconscientemente, estas são partes implícitas do desenvolvimento de software. Um projeto que utiliza uma ferramenta para controle de versão como o CVS [50] está, estritamente falando, gerenciando configuração; um projeto que mantém um cronograma público com atualizações freqüentes está acompanhando e gerenciando o seu processo.

2.7 Desenvolvimento de Software Descentralizado

Até a última década, a maior parte do software era desenvolvido em grupos geograficamente reunidos, trabalhando como uma equipe altamente coesa. No entanto, há uma tendência crescente por utilizar equipes geograficamente dispersas para desenvolver software. Em parte, é responsável por esta tendência o desejo de aproveitar ao máximo as habilidades do pessoal disponível para o projeto, não importando sua localização.

Esta alteração importante na organização da equipe de desenvolvimento exige alterações no processo de software [51]. Nesta seção são tratados aspectos da Engenharia de Software relacionados ao desenvolvimento de software descentralizado ou distribuído.

2.7.1 Problemas Relacionados

Grundy et al. apontam alguns problemas criados ou exacerbados por estas alterações [52]. É interessante notar que, embora muito se fale dos problemas, existe pouca discussão das vantagens que o desenvolvimento distribuído oferece. A principal, e óbvia, é a possibilidade de reunir em uma equipe um conjunto de desenvolvedores geograficamente dispersos.

Os problemas apontados por Grundy são:

- A atribuição das tarefas do desenvolvimento aos membros da equipe.
- A comunicação e colaboração entre desenvolvedores.
- O compartilhamento dos documentos e artefatos do projeto, e a consistência destes.
- A necessidade de suportar plataformas heterogêneas.
- O acompanhamento do progresso dos grupos distintos.
- A integração entre os módulos desenvolvidos por estes grupos.

Estes problemas são tratados em diversos outros trabalhos da área [53, 54], e algumas soluções são discutidas. O trabalho de Herbsleb et al., em particular, é um estudo de caso que faz uma análise bastante completa das dificuldades do desenvolvimento distribuído, e será discutido na próxima seção.

2.7.2 O Estudo de Caso de Herbsleb et al.

Herbsleb et. al realizaram um estudo de um projeto de desenvolvimento de software, distribuído entre dois centros da Lucent na Alemanha e Inglaterra, para o desenvolvimento de um software para comutação telefônica. Neste estudo, é feita uma discussão interessante das questões práticas enfrentadas pelas equipes [54].

Uma das questões que o artigo discute é a divisão em módulos de um sistema. Os autores defendem que esta divisão reflete diretamente em uma divisão do trabalho entre os grupos dispersos, o que torna a responsabilidade dos módulos e suas interfaces muito importantes, e freqüentemente, um fonte de problemas.

Outra questão é que o plano e o modelo de processo representam mal aspectos informais que sustentam o desenvolvimento centralizado: habilidade individual, criatividade, o uso de redes pessoas, e do espaço físico. Um exemplo citado é a lanchonete da empresa, onde desenvolvedores se encontram para discutir assuntos gerais informalmente; este ponto de encontro serve como um mecanismo de troca de informação vital porem informal dentro do grupo de desenvolvimento. Em projetos dispersos, não há este tipo de recurso.

O estudo aponta dificuldades enfrentadas no projeto; entre elas estão as seguintes:

- A integração entre os módulos foi problemática. Os motivos principais foram um plano de integração falho (não contemplava disparidade das entregas efetivas dos módulos) e pouca experiência com a plataforma de hardware terceirizada.
- Manter a consistência entre os repositórios de versão se mostrou difícil; e quando se escolheu consolidar o processo em um repositório único, a dificuldade de julgar alterações se mostrou grande; não havia conhecimento suficiente do código desenvolvido pelo outro grupo.
- A comunicação e contato entre os grupos era restrita, especialmente durante a fase inicial do projeto, onde existia pouca intimidade entre os desenvolvedores, e grandes diferenças culturais para superar. É apontado como um problema especialmente sério o custo alto para iniciar a comunicação, que envolvia descobrir quem seria o responsável por um módulo, encontrá-lo, e efetivamente iniciar a comunicação.

Os autores concluem com as seguintes observações:

- A descentralização trabalha contra os mecanismos informais de comunicação.
- A divisão modular do sistema tem importância fundamental para a divisão do trabalho, e deve ser feita com atenção na estabilidade e compreensão das interfaces.
- É fundamental estabelecer mecanismos para comunicar a situação atual do desenvolvimento e localizar os responsáveis por cada parte do processo.
- Investir na transposição das barreiras de comunicação iniciais é de extrema importância; um plano de viagens é importante para vencer a timidez e desconfiança inicial; ferramentas adequadas para comunicação são essenciais ao processo de desenvolvimento.

2.7.3 Soluções Propostas

Para os problemas apontados por Grundy, Herbsleb et al., existe discussão em alguns trabalhos, sendo que as soluções são apresentadas em dois níveis: primeiro, na elaboração de um processo que seja bem adaptado às necessidades de desenvolvimento descentralizado [55, 56]; segundo, no desenvolvimento de ferramentas específicas para apoiar projetos distribuídos [53, 57].

Uma possibilidade de apoio ao processo é o uso de sistemas de workflow e automação de processo; no entanto, a maior parte dos trabalhos estudados aponta que sistemas de *workflow* não se adaptam bem ao desenvolvimento de software. Grundy et al. descrevem o problema da seguinte forma [52]:

‘Sistemas de *Workflow* e planejamento de projeto [...] provêm recursos mais acessíveis para modelar processos de trabalho, mas em geral não dispõem da flexibilidade para especificar mecanismos de coordenação de trabalho e integração de ferramentas.”

Cook descreve um mecanismo baseado em eventos para capturar informação de um projeto de software distribuído em [56], e neste trabalho ele levanta uma característica importante: se todo diálogo é registrável, é possível acumular dados e posteriormente analisá-los para compreender melhor o processo. Esta parece ser uma vantagem importante que projetos distribuídos têm: o fato de toda comunicação ser registrável e estruturável, já que é transmitida por canais passíveis de armazenamento e medição. A possibilidade de se utilizar este registro para levantar um histórico de alterações nos diferentes produtos de um projeto é bastante interessante.

Ferramentas

Possivelmente porque seja tão importante um conjunto de ferramentas apropriadas para o tipo de tarefa [54] existe uma preocupação tão grande com o assunto de software de suporte a desenvolvimento distribuído. Fielding et al. apontam o surgimento da Internet como uma solução importante para o problema da Engenharia de Software em projetos descentralizados, e sugerem um conjunto de alterações à tecnologia Web para permitir bom suporte à engenharia de software:

- O uso de ligações (*links*) como entidades independentes, permitindo a autores definirem links separadamente do texto. Inclui a possibilidade de prover anotações e caminhos específicos.
- Notificações para clientes (através de um modelo tipo *push*) para comunicar alterações em dados do servidor.
- O uso de pequenos clientes independentes, ao invés dos clientes Web monolíticos (*browsers*) existentes hoje.
- Autoria distribuída e controle de versões para documentos Web.

Parte destes requisitos tem sido desenvolvidos pelo W3C [58] nos últimos anos como parte do desenvolvimento de novos padrões para a Web. Em particular, a tecnologia WebDAV, que existe para permitir autoria e controle de versões para documentos Web, já está desenvolvida e é utilizada em alguns produtos para Web[57].

Existem diversos projetos para desenvolver ferramentas colaborativas para Engenharia de Software através da Web, a exemplo dos citados em [52, 55]. No entanto, parece faltar ainda um conjunto de ferramentas que seja pouco restritivo, multiplataforma, e que suporte os recursos mínimos necessários à colaboração de fato para o desenvolvimento de software descentralizado.

2.8 Considerações Finais

Grande parte das pesquisas feitas na área de Engenharia de Software, e em particular no desenvolvimento de Processos de Software, continuam sendo desenvolvidas e contribuindo para melhorias na construção de produtos de software. No entanto, existe uma tendência atual para a simplificação e pragmatização do processo para acomodar novas necessidades de desenvolvimento; os requisitos e demandas por novos sistemas são muito diferente dos conhecidos e estabelecidos nos anos 70 e 80.

Neste contexto, vem a tona uma nova forma de produção de software, baseada em elementos tradicionais de desenvolvimento aplicados à nova realidade de desenvolvimento descentralizado que a Internet traz. Esta nova forma é baseada em torno de código livremente disponível, Software Livre, e é abordada no próximo capítulo.

Capítulo 3

Software Livre

Neste capítulo é abordado o tópico principal do trabalho, que é a classe de softwares distribuídos como software livre. Para esclarecer a terminologia que será adotada neste trabalho, serão inicialmente apresentadas algumas definições importantes.

3.1 Definições

O termo Software Livre assume um conjunto de significados, dependendo do contexto. Pode significar:

- A forma com a qual o Software é licenciado; ou
- O conjunto de todos os Softwares licenciados desta forma; ou ainda
- Uma comunidade organizada em torno de Softwares licenciados desta forma.

Neste texto, vamos utilizar o primeiro significado, que é definido em [10] como sendo Software que é distribuído acompanhado de código fonte, e que pode ser livremente modificado e redistribuído. O termo tem uma conotação filosófica marcante que, segundo Stallman [59], é proposital: o aspecto a ser reforçado é a liberdade. Usaremos o termo 'não-livre' para descrever software que é fornecido em outros termos.

Um dos aspectos interessantes desta filosofia é que em nada restringe o preço do software. Isto significa que é perfeitamente possível cobrar um valor pelo software ou por sua distribuição. Na prática, o fato da redistribuição do software ser irrestrita resulta em um preço a baixo o suficiente para motivar as pessoas a não copiarem o software de alguém que já o tem. De qualquer forma, o que se observa é que existem duas formas principais de se adquirir o software:

1. Transferindo o software de um repositório online na Internet.
2. Adquirindo o software de um distribuidor, que cobra uma taxa pela distribuição deste software. Em muitos casos o software é oferecido em uma embalagem, e acompanhado de manuais e mídia digital.

Open Source

O termo *Open Source* é bastante conhecido e descreve melhor a forma de desenvolvimento que é aplicada em grande parte dos projetos de software livre. Em português a tradução direta é Código Aberto; no

entanto, a definição é demasiado vaga para ser utilizada consistentemente, e por isso será usada apenas a expressão Software Livre neste trabalho.

Raymond define Open Source em [14]. O termo é usado para descrever software livre que é desenvolvido de forma colaborativa e aberta, e é aplicado como um sinônimo de Software Livre em muitos casos[1].

***Freeware, Shareware,* Domínio Público**

A palavra *Freeware* sugere a idéia do software ser grátis. Como exposto anteriormente, este termo se aplica mal a software livre, por não reforçar a idéia de liberdade (que é a essência do software livre), e por indicar preço zero [61]. A tradução imediata da palavra é *software gratuito*.

Shareware é um termo que descreve software que pode ser redistribuído de forma liberal, mas cujo uso é limitado a uma licença que obriga o usuário a pagar após um período de tempo. Não se aplica a software livre, que não inclui restrições deste caráter [61].

Domínio Público é uma classificação: todo software que não tem copyright (ou seja, cujo autor o abandonou voluntária ou automaticamente), e que pode ser utilizado e modificado sem nenhuma restrição, é classificado como Domínio Público. software livre não é sinônimo de Domínio Público, e usar o termo é incorreto [62]; a existência de um copyright associado ao software livre é um de seus fatores essenciais.

3.2 Copyright e Licenças

Software, como toda produção intelectual, é protegido por copyright, que determina quem é seu proprietário. O dono do copyright tem total controle sobre a forma com a qual seu bem pode ser distribuído, e este controle é usado de uma maneira peculiar com software livre [63].

A forma com a qual o dono do copyright permite o uso e distribuição do seu bem é descrita através de uma Licença. No caso de software, existem inúmeras licenças; o uso do software é ditado por regras descritas nestas licenças[2].

Software Livre utiliza licenças e copyright com o objetivo de garantir a liberdade do software a seus usuários. Cada licença oferece alguns detalhes diferentes, e existem hoje algumas dezenas [64]. A tabela 3.1 apresenta as principais diferenças entre as licenças mais utilizadas [65]. Feller faz na sua introdução em [66] uma boa análise das questões envolvidas com licenciamento.

O assunto de licenciamento é um tema bastante controverso dentro da comunidade de software livre, e a questão de compatibilidade é importante: em muitos casos, software distribuído com uma licença específica não pode ser usado em conjunto com software licenciado de forma incompatível. O que exatamente quer dizer ‘usado em conjunto’, no entanto, é motivo de debate. Torvalds, por exemplo, decidiu que o Linux — embora seja licenciado através do GPL — não proibiria que chamadas do sistema fossem utilizadas por software não-livre. Isto, na prática, significa que software que executa no Linux não tem necessariamente que ser software livre [72].

[1] O Free Software Foundation [60] de Stallman, no entanto, condena este uso.

[2] Em geral, a licença é exibida durante a instalação de um software não-livre, para garantir que o usuário a conhece e aceita.

Exemplos de Licença	Restrições
Domínio Público	Efetivamente qualquer software sem licença é de domínio público, que permite que seu usuário faça o que queira com ele, inclusive modificar e redistribuir. Não há nenhum tipo de restrição sobre o software, e pode ser modificado e distribuído sem código fonte, por exemplo.
BSD [67], X [68], MIT	Permite redistribuição livre do software. Ocasionalmente inclui uma cláusula que obriga cópias redistribuídas a manterem um aviso do copyright. Não obriga versões modificadas a serem livres, e nem a fornecerem código fonte.
GPL [69]	Permite redistribuição desde que mantida a garantia de liberdade inalterada aos usuários da cópia redistribuida. Obriga versões modificadas a serem livres, e portanto, a serem fornecidas acompanhadas de código fonte.
LGPL [70], MPL [71]	Permite redistribuição do código em si mantida a garantia de liberdade inalterada. No entanto, permitem que este código seja usado em um "produto maior" sem que este tenha que ser licenciado livremente. Se modificações forem feitas ao código em si, devem ser fornecidas acompanhadas de código fonte. Esta restrição não cobre o código fonte do "produto maior".

Tabela 3.1: Principais Licenças de Software Livre

3.3 Histórico

Software Livre é um conceito bastante antigo, embora não tivesse este nome específico. Software, durante as décadas de 60 e 70 era desenvolvido de forma colaborativa e aberta em diversas instituições e empresas; o Unix original é um exemplo desta tendência [73]. Embora as licenças não explicitassem claramente liberdade, a redistribuição do software era vista como positiva, e o software geralmente era fornecido com o seu código fonte [74, 75]

A década de 80 trouxe uma alteração importante: a incremental mudança para licenças restritivas, que não permitiam redistribuição. O software fornecido, que até então era visto como uma combinação de código fonte com código executável, passou a significar apenas o executável [75]. O Unix, que era visto como um software único até então, se dividiu entre uma série de fornecedores que introduziram suas alterações proprietárias, reduzindo a sua interoperabilidade [76].

Em 1985, Stallman criou o Free Software Foundation (FSF), cuja finalidade era promover a criação de um sistema operacional completamente livre. Para garantir esta liberdade, Stallman também criou uma licença para o software, a GPL, descrita na seção 3.2. O sistema operacional tinha o objetivo de ser compatível com Unix, e foi chamado de GNU. O desafio de construir o sistema operacional envolvia não apenas a criação de um núcleo de sistema operacional (que até hoje não se encontra em estado de produção), mas a criação de uma coleção de aplicativos e bibliotecas que permitissem ao sistema compatibilidade com o Unix original. Durante esta década, o FSF trabalhou desenvolvendo entre outros a suite de compiladores GCC [77], o editor de textos Emacs [78] e as bibliotecas padrão libc e glibc [79].

Raymond reforça que grande parte do software livre desenvolvido até então era desenvolvido de forma fechada e pouco colaborativa, e usa o termo "Catedral" para descrever o processo de desenvolvimento [14].

Em 1991, um estudante da Universidade de Helsinki, Linus Torvalds, iniciou o desenvolvimento de um núcleo de sistema operacional, o Linux. Este é considerado o mais importante exemplo moderno de um software livre desenvolvido de forma aberta. Torvalds convidou outros desenvolvedores a participarem ativamente da codificação e manutenção do núcleo, e, de forma surpreendentemente rápida, este evoluiu para se tornar um software com funcionalidade similar a núcleos Unix comerciais [80].

Um fator que se considera importante para o sucesso do Linux do ponto de vista do processo de desenvolvimento é a escolha pela licença GPL, e pela decisão de utilizar a Internet como o canal principal de desenvolvimento.

3.3.1 Relação com Unix

Um detalhe interessante é que a cultura em torno de software livre é bastante ligada à cultura em torno do Unix. Os motivos para esta tendência parecem derivar de um longo histórico de uso pelos principais representantes do movimento, e pela natureza exploracional e acadêmica que o Unix original trouxe.

Um tema recorrente entre desenvolvedores é uma admiração pela filosofia do Unix original [81], que é sumarizada por Gancarz [38] como sendo composta dos seguintes pontos:

1. Pequeno é belo.
2. Faça cada programa perfazer uma tarefa apenas, bem.
3. Construa um protótipo assim que possível; requisitos mudam.
4. Escolha portabilidade sobre eficiência.
5. Armazene dados numéricos em arquivos texto.
6. Faça reuso do software para sua vantagem.
7. Use *scripts* shell para aumentar portabilidade e reuso.
8. Evite interfaces restritivas com o usuário.
9. Faça cada programa ser um filtro de dados para que possa ser usado em conjunto com outros.

Os autores originais do Unix colocam estes pontos repetidas vezes como o motivo principal de sucesso do Unix [82, 83]. Esta filosofia é adotada informalmente como a filosofia de grande parte dos projetos de software livre, e são citados em comunicações em listas de discussão de diversos projetos. Desta tradição Unix provavelmente deriva a grande concentração de software livre desenvolvida para esta plataforma e em seus derivados compatíveis.

3.4 Definição de um Projeto de Software Livre

Embora não exista publicada uma definição do que é um Projeto de software livre, existe um consenso informal de que o software em conjunto com seus desenvolvedores, usuários e repositórios de código e documentos constitui um Projeto, e é usada esta nomenclatura neste trabalho. A forma de categorização do software nos diretórios online de software livre Freshmeat.net [26] e Sourceforge [84] corroboram

Maturidade de Desenvolvimento	Número de Projetos
Em Planejamento	9
Pré-Alpha	37
Alpha	177
Beta	387
Produção/Estável	611
Maduro	79

Tabela 3.2: Software Livre em Freshmeat.net por Maturidade

esta definição.

Em síntese, o Projeto de Software Livre, nesta visão, abrange:

- Código fonte, que é o software propriamente dito, mas que existe em forma de inúmeras cópias entre todos seus usuários e repositórios de dados. Software Livre em geral tem uma versão bem definida e distribuída a partir de um ponto central conhecido para o Projeto.
- Um grupo de desenvolvedores, que trabalham para codificar e corrigir este software. Estes desenvolvedores em todos os casos, sem exceção, trabalham colaborativamente através da Internet. Os desenvolvedores podem ter status diferenciado dentro do projeto [85].
- Os usuários do software. Apesar de todo software ter usuários, a participação do usuário no Projeto de software livre é essencial; usuários discutem inovações e apresentam erros encontrados com freqüência, e através deste mecanismo os desenvolvedores são incentivados a trabalhar por um produto melhor [14].
- Os repositórios de documentos e código online; todo projeto tem algum Site central que é uma referência para desenvolvedores e usuários buscarem código e informações atualizados. Além disso, é comum que existam outros repositórios, que podem ser classificados entre:
  - Sites específicos ao projeto. Em ocasiões, é criado um Site para atender a um segmento dos usuários do software, ou algum aspecto técnico [86].
  - Sites direcionados a software livre, onde são informadas versões novas, ou oferecidos serviços aos usuários e desenvolvedores [26, 84].
  - Repositórios CVS públicos, que serão discutidos na seção 3.6.3.

É interessante perceber que todo o desenvolvimento e distribuição depende essencialmente do uso da Internet, e que esta é um pré-requisito para a existência de um projeto de software livre. Com base nesta explicação, serão fornecidos exemplos reais de projetos de software livre.

3.5 Exemplos de Projetos de Software Livre

Existem hoje centenas projetos de software livre em algum estado de desenvolvimento [26], sendo que a maior parte destes já está em algum estado de implementação. Este volume já é suficiente para denotar a importância do conceito. Uma listagem recente de categorias do Freshmeat.net [26] é fornecida na tabela 3.2[3]:

[3]Maturidade neste contexto não se refere ao conceito de Maturidade dos Modelos de Melhoria de Processo.

Dos diversos projetos de sucesso que existem hoje, existem alguns com pertinência especial por sua forma de organização, sua base de usuários, ou pela percepção subjetiva de qualidade. Para que se possa entender a difusão do conceito, foram escolhidos alguns projetos representativos para discussão na próxima seção.

3.5.1 Núcleo de Sistema Operacional: Linux

Existe alguma ambiguidade quando é usada a palavra Linux, que pode tanto se referir ao núcleo de sistema operacional como ao sistema operacional como um todo, incluindo aplicações básicas e de sistema. Neste texto, usaremos Linux para descrever o núcleo apenas.

O Linux [87] é um núcleo multi-tarefa preemptivo baseado na API definida pelo grupo Posix do IEEE [88]. Foi inicialmente implementado por Torvalds [72], e é inteiramente escrito em C e Assembler. O núcleo é multi-plataforma; hoje suporta mais de dez arquiteturas diferentes, a mais importante do ponto de vista do desenvolvimento sendo a arquitetura ia32 da Intel [89].

O fato de ser o projeto de software livre mais famoso também faz com que muito se tenha estudado do ponto de vista de sua evolução e organização, e que existam diversos sites Web dedicados ao tema. O Linux, no entanto, é um dos projetos mais desprovidos de infra-estrutura de desenvolvimento: utiliza como ferramenta apenas formas de comunicação eletrônica — listas de discussão e email. Não há repositório de informes de erros, e tampouco controle de versões público.

Tamanho

O tamanho do núcleo em linhas de código é motivo de algum debate; em [90] o núcleo 2.4, a versão estável atual, é medido em mais de 2 milhões de linhas de código. Wheeler fez um estudo que é um dos mais completos existentes do sistema operacional, e aponta o tamanho do núcleo 2.0, em 1999, em mais de 500.000 ‘linhas de código lógicas’, o que implica contagem omitindo comentários e linhas em branco. [91]. Godfrey e Tu traçam um gráfico do crescimento do núcleo em [92] com detalhes individuais da evolução individual de cada módulo do núcleo; o crescimento foi classificado por eles como “super-linear”.

Participação Externa

Moon e Sproull fazem uma análise histórica do núcleo em [93] em três diferentes perspectivas: dos indivíduos, do grupo e da comunidade; neste trabalho, é colocada claramente a importância do desenvolvimento em comunidade e fornecem estatísticas históricas da participação de desenvolvedores externos no núcleo:

‘...em duas semanas do anúncio de Torvalds [a respeito lançamento do núcleo] em Outubro de 1991, 30 pessoas tinham contribuído cerca de 200 informes de erros, contribuições de utilitários e drivers e melhoras para ser adicionadas ao núcleo [...] em Julho de 1995, mais de 15.000 pessoas de 90 países e 5 continentes tinham contribuído comentários, informes de erro, correções, alterações, reparos e melhoras.”

Uma observação da lista de discussão principal do núcleo na semana de 16 de Abril de 2001 aponta 615 participantes em 1597 mensagens [94]. O tamanho da comunidade tem impacto direto sobre a liderança do núcleo, que é surpreendentemente convencional.

Organização

O núcleo hoje tem em Torvalds seu “dono”, e ele decide quais alterações são aceitas e a filosofia geral do núcleo. No entanto, crescentemente a responsabilidade pelos módulos principais do núcleo tem sido atribuídos a outros, e de certa forma o núcleo tem diversos “donos” em um dado momento [14]. A manutenção da versão estável, há alguns anos, é responsabilidade de Alan Cox.

3.5.2 Núcleo de Sistema Operacional: FreeBSD

O outro exemplo importante é o núcleo FreeBSD. Este núcleo é um derivado direto do núcleo BSD original, que originou como uma série de alterações ao Unix AT&T original e eventualmente evoluiu para uma base de código quase inteiramente nova [73].

O FreeBSD é administrado por um grupo de 9 pessoas, que faz a maior parte da codificação. Há um grupo de mais 200 desenvolvedores que tem acesso de escrita ao repositório de código do projeto. A maior parte das alterações não-triviais são discutidas na lista de discussão antes de integração, de qualquer forma, o que permite que seja feita revisão técnica do projeto e da implementação da alteração.

O FreeBSD possui um conjunto de listas de discussão, um repositório de controle de versões público, e uma interface para informes de defeitos. O projeto é considerado mais conservador que o Linux por discutir mais profundamente nova funcionalidade e suporta a hardware, e este fator é apontado como um motivo para sua relativa lentidão a suportar dispositivos novos.

3.5.3 Servidor Web: Apache

O Apache é um servidor HTTPD, desenvolvido a partir do servidor NCSA original, e que é administrado por um grupo sem fins lucrativos. O Apache é um dos projetos livres mais estudados [27, 85] e sobre o qual existe bastante material descrevendo sua organização.

O projeto de organiza em torno de um núcleo de desenvolvedores, que fazem a maior parte da codificação. Estes desenvolvedores interagem constantemente com os usuários e com os desenvolvedores secundários, que tem suas alterações revisadas publicamente através de listas de discussão. Não há um líder forte é o caso do Linux; o modelo de liderança é semelhante ao do FreeBSD.

O Apache possui um sistema para informes de defeitos para a Web, uma série de listas de discussão, e um repositório CVS público. No último ano, houveram duas conferências Internacionais Apachecon, que tratam especificamente do servidor Web e seus usos [95].

3.5.4 Navegador Web: Mozilla

O Mozilla é um projeto criado pela Netscape para desenvolver um navegador Web. O projeto é um dos maiores entre os projetos de software livre existentes, e é o que conta atualmente com maior apoio financeiro formal.

Existe uma preocupação nítida com a garantia de qualidade no Mozilla [96]; uma série de ferramentas de engenharia de software foi desenvolvida internamente, e o conjunto de documentos detalhando processo, autoridade e status [97] do projeto exemplifica o interesse que o grupo tem pelo processo pelo qual o software é desenvolvido.

O Mozilla conta com uma rede própria de IRC (descrito na seção 3.6.3), uma série de listas de discussão integradas com Usenet[4], e um repositório de informes de defeitos bastante utilizado, o Bugzilla [98].

A autoridade e responsabilidade no Mozilla é dividida. Existem proprietários dos módulos principais do sistema [99], e além disso existe um cargo de *driver*, que é um responsável por decidir e incentivar o reparo dos defeitos mais importantes. Além destes, há ainda um grupo que faz revisão técnicas das alterações apresentadas, utilizando o Bugzilla integralmente.

Um ponto interessante com o Mozilla é que o desenvolvimento é dirigido pelos informes de erros, e as discussões em torno destes abrangem as salas de IRC e as listas de email. A participação de engenheiros da Netscape continuamente nestes meios de comunicação garante resposta rápida a dúvidas e problemas informados, e é interessante observar como a comunidade trabalha utilizando as ferramentas de forma integrada.

3.5.5 Editor de Gráficos Bitmap: Gimp

O Gimp é um editor bitmap avançado [100] , que em muito se assemelha ao Adobe Photoshop. Possui um bom conjunto de ferramentas e filtros, e uma arquitetura de plug-ins que permitem que seja estendido livremente.

De todos os projetos descritos, o Gimp é um dos softwares mais completo de recursos; no entanto, seu processo de desenvolvimento é extremamente simples. É mantido por um conjunto de desenvolvedores pequeno, concentrados na Universidade da Califórnia em Berkeley. Embora tenha uma base de usuários muito grande, o Gimp não utiliza uma grande variedade de ferramentas de Engenharia de Software: utiliza as listas de discussão primordialmente para comunicação, e um canal de IRC para manter discussões em tempo real entre desenvolvedores e usuários. Não há um relatório público de status, e até pouco tempo não uma interface independente para informes de erro[5].

3.6 Características do Processo de Software Livre

De todos os projetos que foram analisados, e de publicações que tratam do processo, é possível levantar algumas premissas básicas para o desenvolvimento de software livre. Essas premissas não são uma regra geral que deve ser seguida, mas um levantamento entre membros da comunidade e seus projetos. Uma profundidade melhor desta análise, com base em dados quantitativos, seria importante.

O primeiro tópico a ser abordado trata de atividades do processo. Deve ficar claro que a lista e o conteúdo das atividades descritas são ainda incompletas pelo fato de existir pouco literatura descritiva.

3.6.1 Metodologia de Desenvolvimento

Existem algumas atividades básicas que são realizadas em grande parte dos projetos de software livre estudados. Estas atividades estão aqui classificadas de acordo com as fases descritas em 2.2.

[4] Usenet é um serviço de distribuição de mensagens Internet que utiliza replicação entre servidores para transmití-las.
[5] o projeto recentemente adotou o bugzilla juntamente com o projeto GNOME.

Especificação de Requisitos

De forma geral, há pequena ênfase do projeto em especificação de requisitos, e muito raramente há um documento formal de especificação [101]. São conhecidos três fatores principais que motivam esta tendência:

1. A maior parte dos projetos existentes replica em alguma forma o comportamento de um ou mais softwares não-livres. Esta tendência é apontada por Cook, que afirma que os projetos não são *cutting-edge*, no sentido de apresentarem pouca inovação [18]. Os exemplos principais que citamos mostram claramente esta tendência; Unix, por exemplo, é um sistema operacional de 30 anos de idade, com grande quantidade de pesquisa e experiência disponível. Além disso, existem muitos padrões publicados nos segmentos onde existem exemplos de software livre de sucesso: o Posix é um exemplo para o Linux, e os padrões do W3C norteiam o Apache e o Mozilla.

2. Como colocado por Raymond e O'Reilly, grande parte dos projetos nasce de uma motivação pessoal do autor inicial, que tem um problema a resolver [14, 102]. Por este motivo, os requisitos são implicitamente conhecidos, discutidos em listas de discussão com outros interessados, e considerados completos o suficiente para se construir uma primeira versão funcional [101]. Esta primeira versão pode já ser o suficiente para gerar interesse pelo projeto, segundo o próximo ítem.

3. O princípio de crescimento evolutivo. Como descrito por Raymond, a melhor técnica de lançamento de software livre é "Faça *releases* [do software] cedo, e com freqüência" [14], e grande parte dos projetos adota esta técnica. A própria filosofia do Unix, colocada por Gancarz [38], afirma que os requisitos são raramente estáveis, e que o usuário dificilmente saberá o que quer. Em parte, este princípio ecoa as idéias colocadas por Brooks de prototipação [37].

Da forma como estes fatores são colocados, é difícil determinar se a inexistência de requisitos formalizados é um fator negativo de importância para projetos de software livre.

Projeto

Durante a fase de projeto, em geral se coloca um projeto a nível de sistema para um software. No entanto, na maior parte dos sistemas de software livre, não existe uma definição clara da arquitetura do sistema [101]. Existe pouco material publicado discutindo esta ausência, e aparentemente é uma falta grave.

É possível que a alta qualidade dos desenvolvedores envolvidos com os projetos, citada por Cook como o fator mais importante para o sucesso de software livre [18], ajude a minimizar este problema. O fato de existir um líder ou grupo de líderes com experiência nos projetos de sucesso, como listado acima, pode ser outra explicação. O fato resta de que existem produtos de software livre sem especificação clara de seu projeto, e no entanto, de sucesso impressionante, como é o caso do Linux.

Codificação: Desenvolvimento Colaborativo e Distribuído

A codificação de um software livre inicia em geral imediatamente após sua idealização. Ecoando as idéias colocadas na seção 3.6.1, a prototipação é uma prática comum, e existe grande impulso para se chegar a uma versão funcional inicial.

O ponto mais importante em relação ao desenvolvimento a ser colocado é que é feito integralmente por meio da Internet. Poucos projetos envolvem desenvolvedores fisicamente próximos, e em geral

estes utilizam a infra-estrutura online para discutir suas idéias de qualquer forma.

Segundo o que se pode levantar do trabalho de Raymond [14], Yamauchi et al. [15], e o processo de codificação se dá da seguinte maneira:

1. Um desenvolvedor (ou um grupo) cria uma versão inicial do código em particular, testando e implementando em relativo isolamento.

2. Um anúncio é feito em algum veículo de comunicação da comunidade: em geral é colocada em uma ou mais listas de discussão, e no Freshmeat.net ou outro site associado.

3. Outros desenvolvedores se interessam pelo projeto, transferem o código, e experimentam com o que já foi desenvolvido. Caso tenham dúvidas ou sugestões, contatam os desenvolvedores por meio eletrônico.

4. Alterações ao código são enviadas por email aos autores, e às listas de discussão relevantes. Estas alterações são feitas na forma de deltas ou *patches*[6] textuais [50]. Estas alterações são geralmente analisadas e discutidas (embora muitas vezes sejam ignoradas), e um acordo é estabelecido sobre a qualidade e pertinência da alteração. Caso seja julgada uma alteração positiva, é integrada ao repositório de código e será incluída em uma nova versão do software.

5. Alterações feitas por membros do círculo de liderança do projeto em geral podem ser integradas sem discussão. No entanto, existe o costume de se utilizar uma lista de discussão para registrar automaticamente as alterações no repositório, de forma que existirá mais de um local onde a alteração estará publicamente visível. O projeto Wine de emulação Windows possui uma lista de *patches* neste estilo, como exemplo [103].

É importante deixar claro que o desenvolvimento é visto como uma atividade bastante individual [102]. Isto significa que durante o processo de codificação de uma alteração, o desenvolvedor trabalha em isolamento; apenas quando deve ser feita a verificação e integração de sua alteração ao repositório de código principal é que entra o caráter colaborativo do software.

Evolução e Manutenção

A seção anterior descreve implicitamente o que é uma característica definitiva para software livre: O software está, em quase toda a sua vida, em produção. A partir do primeiro lançamento, já existirão usuários interessados (caso o projeto tenha despertado algum interesse) e todo o impacto de uma versão lançada se aplica; em especial, compatibilidade de interfaces, desatualização da documentação e resistência a mudanças. Skillcorn et al. fazem uma revisão sobre o efeito de alterações de requisitos sobre software em manutenção, e colocam o problema da seguinte forma [104]:

‘Adicionar novos requisitos, remover requisitos parcialmente implementados, ou alterar significativamente requisitos existentes após um conjunto mínimo de requisitos ter sido estabelecido, é um problema comum em manutenção de software. Infelizmente, há poucos métodos, ferramentas, ou enfoques que tratem do problema hoje...”

Um ponto importante a ser levantado é como projetos de software livre minimizam o impacto de alterações pós-lançamento, já que passam a maior parte de sua vida em manutenção.

[6] Um patch é um texto contendo a variação entre duas versões de um arquivo ou programa.

3.6.2 Teste e Garantia da Qualidade

Se as atividades básicas do processo de software estão descritas, ainda resta descrever os processos pelos quais estes projetos são controlados. Software Livre desenvolvido de forma colaborativa e distribuída, de acordo com o modelo "Bazar" proposto por Raymond [14] utiliza alguns mecanismos de garantia de qualidade. O artigo de Zhao e Elbaum é base importante para esta seção, e é o único estudo direcionado a atividades de qualidade em projetos de software livre [105].

Uso de linguagens de alto nível

A maior parte do software desenvolvido, hoje, na comunidade, é escrito em linguagens de alto nível [91]. O uso de linguagens de alto nível é apontado por Brooks como sendo um dos principais fatores para a melhora de qualidade e produtividade no desenvolvimento de software [39]. Grande parte dos projetos novos listados no Freshmeat.net [26] são desenvolvidos em Python, Java, PHP e Perl, todas estas sendo linguagens dinâmicas, interpretadas, modernas.

Uso de controle de versão

Grande parte dos projetos de software livre hoje utilizam alguma forma de controle de versão [106], e a absoluta maioria utiliza apenas uma ferramenta, o CVS [50]. O controle de versão é visto como uma extensão natural do processo de desenvolvimento, e permite que se possa paralelizar o desenvolvimento de forma conveniente, especialmente se tratando de um conjunto muito grande de desenvolvedores.

É interessante notar que o Linux até hoje não utiliza nenhuma forma oficial de controle de versões, e todas as alterações são trocadas por email [107]. Torvalds é conhecido por pessoalmente analisar a fundo as alterações que são enviadas e integrá-las ao seu repositório principal, embora parte desta tarefa seja hoje dividida com o chefe da versão estável, Cox.

Inspeção e Revisão de Código

Todos os projetos implementam extensivamente revisão de código. A revisão acaba acontecendo quase automaticamente, já que as pessoas são forçadas pela distância a enviar modificações a outras para integração. Yamauchi et al. estudaram dois casos distintos de projetos de software, e levantaram respectivamente as proporções de 14% e 34.4% do total de comunicação como sendo referentes a contribuições a nível de alterações (*patches*) ou resultados de testes [15].

Raymond coloca esta revisão de uma forma peculiar: "Dado olhos suficientes, todos os defeitos são rasos". O sentido implícito na frase é que, embora o desenvolvimento seja feito independentemente, a revisão de código utiliza a seu favor a grande massa de interessados para analisar alterações.

Yamauchi et al. colocam de forma interessante o mecanismo social pelo qual é feita a revisão de código: todas as decisões são negociadas com argumentos estritamente técnicos, e o processo de discussão é essencialmente racional. Os autores indicam que esta racionalidade equaliza os participantes e reduz a necessidade de uma hierarquia [15]. Certamente o que se observa nas listas de discussão é que grande parte das mensagens é repleta de justificativas técnicas para defender um ponto de vista.

Teste Beta

O uso do termo Beta pode não ser estritamente correto, mas permite um entendimento mais fácil do modelo de testes usado em projetos de software livre. Em resumo, todo software é tratado como permanentemente em Beta, e resultados e opiniões dos usuários são sempre recolhidos. A natureza do desenvolvimento, que tem caráter contínuo em projetos ativos, permite que esta atitude seja mantida ao longo do tempo. É muito infrequente um cenário onde alterações estão expostas a um grupo pequeno de desenvolvedores; em geral, qualquer usuário pode escolher usar a versão que desejar [14].

Existe, no entanto, uma divisão ideal em muitos projetos entre uma versão estável, e uma versão instável. A versão estável permite apenas a inclusão de correções de defeitos, enquanto a versão instável permite adição de nova funcionalidade [92]. É possível que esta divisão torne aos usuários menos custoso, efetivamente, estar usando software em desenvolvimento (e portanto teste) contínuo.

Teste Funcional

Embora não seja uma regra geral, e projetos importantes como o Linux não tenham um plano de testes formal, existe uma consciência de que o software deve ser testado antes de seu lançamento. Em geral, este teste é implementado através de uma suite de testes funcionais que é executada de forma automática imediatamente antes da instalação do software.

Zhao e Elbaum apontam a ausência de um plano de testes como sendo uma fraqueza potencial do modelo de desenvolvimento utilizado. Segundo estes, a confiança no processo de Revisão de Código e Teste Beta pode não ser justificada, já seu estudo sugere que ocorre menos do que se acreditava previamente [105].

3.6.3 Ferramentas

Os projetos de software livre que foram analisados nesta revisão implementam o uso de pelo menos uma das ferramentas a seguir. É interessante notar que existem muito poucas ferramentas, e grande uniformidade entre os projetos de quais são aplicadas. Wilson coloca a ausência de ferramentas apropriadas como um defeito grave do modelo de desenvolvimento de software livre [108]; no entanto, Yamauchi e Bollinger et al. concordam no ponto que o minimalismo auxilia e simplifica o processo de desenvolvimento de software livre [15, 42].

Como visto na seção 2.7, que tratou de Desenvolvimento de Software Descentralizado, o uso de ferramentas é essencial para software que é desenvolvido desta forma. Toda comunicação tem que ser feita explicitamente por meio de alguma ferramenta, e a habilidade de controlar e verifificar o progresso de um dado projeto, também. As ferramentas discutidas a seguir não representam uma lista exaustiva; foram escolhidas pela sua popularidade e utilização em projetos importantes como o Linux, o Mozilla e o Apache.

Email

A ferramenta de trabalho mais utilizada, e que é usada em todo e qualquer projeto de software livre, é o email. Toda correspondência direta entre desenvolvedores, e entre desenvolvedores e os usuários, é feita desta forma [15].

O email é bastante utilizado por ser um bom denominador comum entre plataformas e línguas ; é esperado que qualquer desenvolvedor tenha acesso conveniente, e que possa se comunicar com clareza através deste meio. O email é utilizado para enviar alterações de código através de *patches*, e para posteriormente discutir estas mudanças [27].

Listas de Discussão

Uma lista de discussão é um mecanismo de distribuição de emails para um conjunto de emails de usuários assinantes. Grande parte dos projetos de software livre tem uma ou mais listas associadas, e a lista se torna um dos veículos mais importantes de comunicação com os desenvolvedores [15].

Através das listas, usuários solicitam inovações e comunicam defeitos encontrados. A maior parte das listas é armazenada em um repositório indexado, de forma que é possível pesquisar uma solução nelas posteriormente [84].

IRC

Nos últimos dois anos, a comunidade de desenvolvimento de software livre tem passado a utilizar o IRC como uma forma de comunicação instantânea. O IRC é um serviço de comunicação em tempo real que implementa um paradigma de salas (ou canais), onde membros de uma mesma sala podem comunicar abertamente. Existem diversas redes de IRC, cada rede suportando um conjunto de canais.

O IRC oferece a possibilidade de obter resposta e discutir assuntos relacionados ao desenvolvimento sem os problemas gerados pela natureza assíncrona do email. O projeto Mozilla utiliza extensivamente o IRC como ferramenta de projeto [98].

CVS

O CVS é um software para controle de versões. O CVS é um exemplo de sucesso entre projetos de software livre; a maior parte dos projetos o utiliza, e por as características de simplicidade e compatibilidade que o email oferece [50]. Existe uma interface web para consultar CVS chamada CVSWeb.

O CVS é largamente documentado e utilizado; no entanto, possui limitações e inconveniências, como apontado por McDonald [109]. A falta de integração com outras ferramentas também é vista como um problema. No entanto, segue sendo largamente utilizado, e após anos sem lançamentos, o desenvolvimento de uma nova versão foi registrado recentemente [110]. O sucesso do CVS pode se dever à sua característica de bom denominador comum, similarmente ao email.

Bugzilla e GNATS

Existem alguns softwares para controlar informes de erros que são utilizados em projetos de software livre. O Bugzilla é um produto da organização Mozilla, e é um mecanismo integrado de informes feito para a Web [111]. O Bugzilla tem um interface interessante de consultas, pode ser customizado à instalação, e utiliza uma base de dados relacional associada. O Bugzilla permite controlar informes a nível de módulos e produtos, tem um sistema integrado de comentários e anexos, e fornece um mecanismo de controle de acesso.

O GNATS [112] é um outro sistema de controle de informes, sendo este baseado principalmente em email e ferramentas de comando de linha, com uma interface para geração de relatórios baseada na Web.

3.7 Trabalhos Relacionados

Existem alguns trabalhos científicos publicados que tratam software livre do ponto de vista do processo pelo qual é construído:

Yamauchi et al. [15] fazem uma discussão da dependência de projetos de software livre a meios eletrônicos limitados. Em seu trabalho, caracterizam os tipos de meio usado nos projetos, e verificam as formas pelas quais os projetos se coordenam, inovam e concordam baseados nestes meios. Seu estudo de caso é realizado aplicando entrevistas e monitoram correspondência dos desenvolvedores em dois projetos de software livre.

Zhao e Elbaum [105] fazem uma análise das atividades de qualidade em projetos de software livre, aplicando questionários aos desenvolvedores e medindo defeitos para caracterizar o uso de testes, inspeção, linguagens e ferramentas nos projetos estudados.

Cubranic apresenta um trabalho sintético em [113], descrevendo alguns projetos, e fazendo críticas e elogios a características específicas do modelo. No artigo, o autor apresenta algumas sugestões de mudanças ao processo geral de desenvolvimento que visam permitir a software livre ser aplicado a outros nichos que não somente software básico.

Feller e Fitzgerald fazem uma análise da metodologia de desenvolvimento de software livre em [66], utilizando um framework para classificação de arquiteturas de Sistemas de Informação definido por Zachman em [114]. Os autores ainda fazem uma breve comparação entre o modelo RAD [2] e o processo de desenvolvimento de software livre.

Godfrey e Tu realizam um estudo da evolução do núcleo Linux em [92], usando medições sobre o código fonte deste. Sua conclusão é que o perfil de crescimento é “inesperadamente forte”, e que o núcleo não apresenta os problemas comuns a projetos de software tradicionais, como apresentado por Lehman[7] em [115].

Mockus et al. analisam em [27] o servidor Apache, aplicando um questionário a líderes do projeto, e analisando listas de discussão e uma base de dados de defeitos. Os autores identificaram as métricas a ser usadas utilizando o método Goal Question Metric (GQM) de Basili e Weiss [116].

3.8 Considerações Finais

Existe ainda uma grande quantidade de projetos a ser analisada, de onde podem ser retiradas conclusões importantes sobre o processo pelo qual são desenvolvidos e administrados. Software Livre apresenta algumas características interessantes do ponto de vista dos produtos gerados, e na baixa dependência de infra-estrutura e recursos formais. Tendo em vista estes pontos, será apresentado no próximo capítulo o plano de trabalho, incluindo a metodologia a ser utilizada e as atividades previstas.

[7] Lehman, neste trabalho, defende que o ritmo de crescimento de um sistema diminui à medida que o próprio sistema cresce.

Capítulo 4

Plano de Trabalho

Revistos os tópicos relacionados a Processos de Software, e a Projetos de Software Livre, nesta seção são abordados o projeto e a metodologia propostos.

4.1 Descrição do Projeto

Como colocado na Introdução, o objetivo do projeto é caracterizar o processo de desenvolvimento que projetos de Software Livre utilizam, e verificado um processo homogêneo, descrever um modelo para este processo.

O trabalho constitui um levantamento (*survey*) de atividades de Engenharia de Software para projetos de software livre. Existem trabalhos anteriores deste tipo; no entanto, são restritos a uma área específica da Engenharia de Software, como o trabalho de Zhao e Elbaum [105], que aborda questões relacionada à garantia de qualidade.

Metodologia

O trabalho aplicará pesquisa de campo na forma de observação direta intensiva [117, 118], de forma exploratória, para levantar e analisar dados quantitativos e qualitativos a respeito dos projetos. A intenção é obter dos líderes e participantes das comunidades de desenvolvimento os aspectos particulares de cada projeto, e correlacionar estes dados para sintetizar um conjunto comum de atividades.

Para realizar o levantamento, é necessário definir o conjunto de projetos a ser estudados, e com este conjunto aplicar a pesquisa de campo. Os aspectos importantes que os projetos escolhidos devem contemplar foram determinados do conjunto de projetos apresentados no Capítulo 3. Estes aspectos são:

- Uso de um canal comum para comunicação, como uma lista de discussão.
- Uso de um repositório de controle de versões público.
- Uso de ferramentas online para gerenciamento do projeto, como os sistemas de acompanhamento de defeitos GNATS e Bugzilla, descritos no capítulo anterior.

O levantamento poderá ser feito através de questionários aplicados individualmente. Para identificar as métricas pode ser utilizada a metodologia Goal Question Metric (GQM), que já foi aplicada anteriormente a projetos de Software Livre com sucesso [27]. Definir a metodologia a ser aplicada é

uma tarefa a ser cumprida já ao início do trabalho.

Além do levantamento de dados através de entrevistas individuais, existe a possibilidade de utilizar os repositórios de dados públicos de cada projeto. Pode ser analisado e quantificado destes dados um conjunto de padrões que podem corroborar as conclusões obtidas dos dados individuais.

4.2 Atividades Previstas

As atividades que deverão ser executadas para o cumprimento do trabalho seguem:

1. Pesquisa bibliográfica, contemplando estudo e análise dos modelos de processo existentes na literatura; estudo de Projetos de Software Livre, focando na sua organização, participação e mecanismos de Engenharia de Software implícitos.
2. Pesquisa e definição de uma metodologia para coleta de informações a ser aplicada no trabalho.
3. Desenvolvimento de uma base para o levantamento de dados, constituindo questionários, entrevistas e software conforme necessário.
4. Aplicação, coleta e síntese dos dados coletados em um repositório para análise.
5. Análise dos dados, e documentação dos resultados.
6. Escrita e defesa da dissertação.

A tabela 4.1 detalha um cronograma para estas atividades; a data de início coincide com a entrega do monografia de qualificação, Abril de 2001.

	Meses (2001/2002)													
Ativ.	Abr	Mai	Jun	Jul	Ago	Set	Out	Nov	Dez	Jan	Fev	Mar	Abr	Mai
1														
2														
3														
4														
5														
6														

Tabela 4.1: Resumo das Atividades do Projeto

Bibliografia

[1] DEMARCO, T. Software development: State of the art vs. state of the practice (1993). In: *Why does Software Cost so Much?* New York: Dorset House, 1st. ed., 1995.

[2] PRESSMAN, R. S. *Software engineering: A practitioner's approach*. 4th. ed. McGraw-Hill, 1997. p. 22–53.

[3] BOEHM, B. W. The high cost of software. In: *Practical Strategies for Developing Large Systems*. Menlo Park: Addison-Wesley, 1st. ed., 1975.

[4] PRESSMAN, R. S. *Software engineering: A practitioner's approach*. 4th. ed. McGraw-Hill, 1997. p. 16.

[5] KELLNER, M. I. Software process modeling experience. In: Proceedings of ICSE, 11., 1989, Pittsburgh. IEEECSP. p. 400–401.

[6] LINDVALL, M., RUS, I. Process diversity in software development. *IEEE Software*, v. 17, n. 4, p. 14–18, July/August 2000.

[7] DEMARCO, T. Mad about measurement (1995). In: *Why does Software Cost so Much?* New York: Dorset House, 1st. ed., 1995.

[8] BECK, K. *Extreme programming explained*. Reading: Addison-Wesley, 2000.

[9] BACH, J. Enough about process: What we need are heroes. *IEEE Software*, v. 12, n. 2, p. 96–98, March 1994.

[10] The Free Software Foundation. What is Free Software? Disponível em `http://www.gnu.org/philosophy/free-sw.html`. 2001. Visitado em Abril de 2001.

[11] MILLER, B. P. et al. Fuzz revisited: A re-examination of the reliability of unix utilities and services. Disponível em `ftp://grilled.cs.wisc.edu/technical_papers/fuzz-revisited.ps`. 1995. Visitado em Fevereiro de 2001.

[12] EDWARDS, J. The changing face of freeware. *IEEE Computer*, v. 31, n. 10, p. 11–13, 1998.

[13] VALLOPPILIL, V. V. Open source software – a (new?) development methodology. Disponível em `http://www.opensource.org/halloween1.html`. 1998. Visitado em Fevereiro de 2001.

[14] RAYMOND, E. S. The Cathedral and The Bazaar. In: *The Cathedral and The Bazaar*. Sebastopol: O'Reilly and Associates, 1st. ed., 1999. p. 27–78.

[15] YAMAUCHI, Y., YOKOZAWA, M., SHINOHARA, T., ISHIDA, T. Collaboration with Lean Media: How Open Source Succeeds. In: Proceedings of CSCW, 2000. ACM Press. p. 329–338.

[16] RAYMOND, E. S. Homesteading the Noosphere. In: *The Cathedral and The Bazaar*. Sebastopol: O'Reilly and Associates, 1st. ed., 1999. p. 79–135.

[17] MCCONNELL, S. C. Open source methodology: Ready for prime time? *IEEE Software*, v. 16, n. 4, p. 6–8, July/August 1999.

[18] COOK, J. E. Open source development: An arthurian legend. In: Position Papers for the ICSE 2001 Workshop: Open Source Development, 2001. IEEECSP.

[19] GLASS, R. L. The sociology of open source: Of cults and cultures. *IEEE Software*, v. 17, n. 3, p. 104–105, May/June 2000.

[20] IBM. IBM Linux Technology Center. Disponível em `http://oss.software.ibm.com/developerworks/opensource/linux/`. 2001. Visitado em Fevereiro de 2001.

[21] Apple Computer. Apple - Public Source: Open Source Projects at Apple. Disponível em `http://www.apple.com/darwin/`. 2001. Visitado em Abril de 2001.

[22] KARPINSKI, R. AOL's Case Supports Open Source. Disponível em `http://www.techweb.com/wire/story/TWB19981130S0013`. 1998. Visitado em Janeiro de 2001.

[23] ICSE 2001. Workshop on Open Source Development. Disponível em `http://opensource.ucc.ie/icse2001/`. 2001. Visitado em Fevereiro de 2001.

[24] Netcraft. Netcraft Web Server Survey. Disponível em `http://www.netcraft.com/survey/`. 2001. Visitado em Janeiro de 2001.

[25] Linux International. Estimates of the Number of Linux Users. Disponível em `http://counter.li.org/estimates.html`. 2001. Visitado em Abril de 2001.

[26] Freshmeat.Net. Freshmeat II. Disponível em `http://freshmeat.net/`. 2001. Visitado em Abril de 2001.

[27] MOCKUS, A., FIELDING, R., HERBSLEB, J. A case study of open source software development: The Apache Server. In: Proceedings of ICSE, 2000. IEEECSP. p. 263–272.

[28] SOMMERVILLE, I. *Software engineering*. 5th. ed. Addison-Wesley, 1995. p. 7.

[29] PRESSMAN, R. S. *Software engineering: A practitioner's approach*. 4th. ed. McGraw-Hill, 1997. p. 22–23.

[30] SCHWARTZ, J. I. Construction of software. In: *Practical Strategies for Developing Large Systems*. Menlo Park: Addison-Wesley, 1st. ed., 1975.

[31] ROYCE, W. W. Managing the development of large software systems. In: Proceedings of IEEE WESCON, 1970. p. 1–9.

[32] DAVIS, A. M., SITARAM, P. A concurrent process model of software development. *ACM SIGSOFT Software Engineering Notes*, v. 9, n. 2, p. 38–51, April 1994.

[33] RACOON, L. The chaos model and the chaos life cycle. *ACM SIGSOFT Software Engineering Notes*, v. 20, n. 1, p. 55–66, January 1995.

[34] BOEHM, B. W. A spiral model of software development and enhancement. *ACM Software Engineering Notes*, v. 11, n. 4, p. 14–24, August 1986.

[35] RISING, L., JANOFF, N. S. The Scrum Software Development Process for Small Teams. *IEEE Software*, v. 17, n. 4, p. 11–13, July/August 2000.

[36] HIRAI, C., SAEKI, N. A proposal of an internet-based software development process model for cots-based systems development. In: Proceedings of ICSE Workshop: Software Engineering Over the Internet, 1998. IEEECSP.

[37] BROOKS, F. P. Plan to throw one away. In: *The Mythical Man-Month*. Reading: Addison-Wesley, 2nd. ed., 1995.

[38] GANCARZ, M. *Unix Philosophy*. Reading: Prentice-Hall, 1995.

[39] BROOKS, F. P. No silver bullet. In: *The Mythical Man-Month*. Reading: Addison-Wesley, 2nd. ed., 1995.

[40] LAWSON, H. W. From busyware to stableware. *IEEE Computer*, v. 31, n. 10, p. 117–119, October 1998.

[41] FOWLER, M. The New Methodology. Disponível em `http://www.martinfowler.com/articles/newMethodology.html`. 2001. Visitado em Abril de 2001.

[42] BOLLINGER, T., NELSON, R., SELF, K. M., TURNBULL, S. J. Open source methods: Peering through the clutter. *IEEE Software*, v. 16, n. 4, p. 8–11, July/August 1999.

[43] LEHMAN, M. M. Some Reservations on Software Process Programming. In: Proceedings of the IEEE/ACM Software Process Workshop, 4., 1988. p. 111–112.

[44] XP2001. Extreme Programming Conference. Disponível em `http://www.xp2001.org/`. 1999. Visitado em Abril de 2001.

[45] KERNIGHAN, B. W., PIKE, R. *The practice of programming*. Reading: Addison-Wesley, 1999.

[46] SUTHERLAND, J. SCRUM Software Development Process. Disponível em `http://jeffsutherland.com/scrum/index.html`. 2001. Visitado em Abril de 2001.

[47] SCWABER, K. SCRUM Development Process. In: Proceedings of OOPSLA, 1995. Springer-Verlag.

[48] COCKBURN, A. Crystal Methodologies. Disponível em `http://crystalmethodologies.org/`. 2001. Visitado em Abril de 2001.

[49] MCCONNELL, S. *Code complete*. 1st. ed. Redmond: Microsoft Press, 1995.

[50] BERLINER, B. CVS II: Parallelizing software development. In: Proceedings of the USENIX Winter 1990 Technical Conference, Berkeley, CA. USENIX Association, c1990. p. 341–352.

[51] MAURER, F., KAISER, G. Software engineering in the internet age. *IEEE Internet Computing*, v. 2, n. 5, p. 22–24, September/October 1998.

[52] JOHN GRUNDY, J. H., MUGRIDGE, R. Coordinating distributed software development projects with integrated process modelling and enactment environments. In: Proceedings of IEEE WETICE, Stanford. c1998. p. 39–44.

[53] FIELDING, R. T., JR., E. J. W., ANDERSON, K. M., BOLCER, G. A., OREIZY, P., TAYLOR, R. N. Software engineering and the www: The cobbler's barefoot children, revisited. Technical report, UCI 96-53, November 1996.

[54] HERBSLEB, J. D., GRINTER, R. E. Splitting the organization and integrating the code: Conway's law revisited. In: Proceedings of ICSE, 1999, Los Angeles. IEEECSP. p. 85–95.

[55] ALHO, K., SULONEN, R. Supporting Virtual Software Projects on the Web. In: Proceedings of IEEE WETICE, 1998. IEEECSP. p. 10–14.

[56] COOK, J. E. Internet-based software engineering enables and requires event-based management tools. In: Proceedings of ICSE Workshop: Software Engineering over the Internet, 1998, Los Angeles. IEEECSP.

[57] WHITEHEAD, Jr., E. J., WIGGINS, M. Webdav: Ietf standard for collaborative authoring on the web. *IEEE Internet Computing*, v. 2, n. 5, p. 34–40, September/October 1998.

[58] W3C. The WWW Consortium. Disponível em `http://www.w3c.org/`. 2000. Visitado em Janeiro de 2001.

[59] The Free Software Foundation. Open source. Disponível em `http://www.gnu.org/philosophy/free-software-for-freedom.html`. 2001. Visitado em Abril de 2001.

[60] The Free Software Foundation. The free software foundation. Disponível em `http://www.gnu.org/`. 26 Mar. 2001. Visitado em Abril de 2001.

[61] The Free Software Foundation. Categories of Free and Non-Free Software. Disponível em `http://www.gnu.org/philosophy/categories.html`. 2001. Visitado em Fevereiro de 2001.

[62] JASSIN, L. J. What is the Public Domain? Disponível em `http://www.gigalaw.com/articles/jassin-2000-11-p2.html`. 2001. Visitado em Abril de 2001.

[63] FIELD Jr., T. G. Copyright for Computer Authors. Disponível em `www.fplc.edu/tfield/copySof.htm`. Sept. 2000. Visitado em Abril de 2001.

[64] The GNU Project. Various Licenses and Comments about Them. Disponível em `http://www.gnu.org/philosophy/license-list.html`. 2001. Visitado em Abril de 2001.

[65] PERENS, B. The Open Source Definition. In: *Open Sources*. Sebastopol: O'Reilly and Associates, 1st. ed., 1999. p. 171–188.

[66] FELLER, J., FITZGERALD, B. A framework analysis of the open source software development paradigm. In: , 21., 2000, Brisbane.

[67] The FreeBSD Project. The 4.4BSD License. Disponível em `http://www.freebsd.org/copyright/license.html`. 1994. Visitado em Fevereiro de 2001.

[68] The X Consortium. Terms and Conditions. Disponível em `http://www.x.org/terms.htm`. 2000. Visitado em Abril de 2001.

[69] The GNU Project. The GNU General Public License. Disponível em `http://www.gnu.org/copyleft/gpl.html`. June 1991. Visitado em Abril de 2001.

[70] The GNU Project. The GNU Lesser General Public License. Disponível em `http://www.gnu.org/copyleft/lesser.html`. June 1999. Visitado em Abril de 2001.

[71] The Mozilla Organization. The Mozilla Public License. Disponível em `http://www.mozilla.org/MPL/MPL-1.1.html`. 2001. Visitado em Abril de 2001.

[72] TORVALDS, L. The Linux Edge. In: *Open Sources*. Sebastopol: O'Reilly and Associates, 1st. ed., 1999. p. 101–111.

[73] MCKUSICK, M. K. Twenty years of Berkeley Unix. In: *Open Sources*. Sebastopol: O'Reilly and Associates, 1st. ed., 1999. p. 31–46.

[74] RAYMOND, E. S. A brief history of hackerdom. In: *The Cathedral and The Bazaar*. Sebastopol: O'Reilly and Associates, 1st. ed., 1999. p. 5–26.

[75] STALLMAN, R. M. The GNU Operating System and the Free Software Movement. In: *Open Sources*. Sebastopol: O'Reilly and Associates, 1st. ed., 1999. p. 53–70.

[76] Warren Toomey. UNIX History Graphing Project. Disponível em `http://minnie.cs.adfa.edu.au/Unix_History/`. 2001. Visitado em Abril de 2001.

[77] The Free Software Foundation. The gnu compiler collection. Disponível em `http://gcc.gnu.org/`. 26 Mar. 2001. Visitado em Fevereiro de 2001.

[78] The Free Software Foundation. GNU Emacs. Disponível em `http://www.gnu.org/software/emacs/emacs.html`. 2001. Visitado em Fevereiro de 2001.

[79] The Free Software Foundation. The GNU C Library. Disponível em `http://www.gnu.org/software/libc/libc.html`. 2001. Visitado em Abril de 2001.

[80] Thomas Schenk. Linux: Its history and current distributions. Disponível em `http://www.developer.ibm.com/library/articles/schenk1.html`. 26 Mar. 2001. Visitado em Abril de 2001.

[81] KERNIGHAN, B. W., PIKE, R. *The UNIX Programming Environment*. Reading: Prentice-Hall, 1984.

[82] KERNIGHAN, B. W. The Unix System and Software Reusability. *IEEE Transactions on Software Engineering*, v. 10, n. 5, p. 513–518, September 1984.

[83] PIKE, R. Systems Software Research is Irrelevant. Disponível em `plan9.bell-labs.com/cm/cs/who/rob/utah2000.pdf`. 2000. Visitado em Abril de 2001.

[84] Sourceforge. Sourceforge. Disponível em `http://sourceforge.net`. 2001. Visitado em Janeiro de 2001.

[85] FIELDING, R. T. Shared leadership in the Apache Project. *Communications of the ACM*, v. 42, n. 4, p. 42–43, April 1999.

[86] The Linux-MM Team. Linux Memory Management. Disponível em http://www.linux-mm.org. 2001. Visitado em Abril de 2001.

[87] Transmeta Corp. The Linux Kernel Archives. Disponível em http://www.kernel.org. 2001. Visitado em Abril de 2001.

[88] IEEE. Portable Operating Systems Interface (POSIX (R)). Disponível em http://standards.ieee.org/catalog/posix.html. 1995. Visitado em Fevereiro de 2001.

[89] DEURZEN, P. A. Linux Architectures. Disponível em http://home.hccnet.nl/p.van.deurzen/linuxweb/docs/architectures.htm. 2001. Visitado em Janeiro de 2001.

[90] BEIRNE, P. Pat Beirne's Linux FAQ. Disponível em http://patb.dyndns.org/Programming/patb_linux.html. 1999. Visitado em Abril de 2001.

[91] WHEELER, D. A. Estimating Linux's Size. Disponível em http://www.dwheeler.com/sloc. 1999. Visitado em Abril de 2001.

[92] GODFREY, M. W., TU, Q. Evolution in open source software: A case study. In: Proceedings of ICSM, 2000.

[93] MOON, J. Y., SPROULL, L. Essence of Distributed Work: The Case of the Linux Kernel. Disponível em http://www.firstmonday.dk/issues/issue5_11/moon/index.html. 2000. Visitado em Janeiro de 2001.

[94] BROWN, Z. Kernel Traffic. Disponível em http://http://kt.zork.net/kernel-traffic/kt20010416_114.html. 2001. Visitado em Abril de 2001.

[95] The Apache Group. Apache Conferences. Disponível em http://www.apache.org/foundation/conferences.html. 2001. Visitado em Abril de 2001.

[96] The Mozilla Organization. Mozilla QA Home Page. Disponível em http://www.mozilla.org/quality.html. 2001. Visitado em Fevereiro de 2001.

[97] The Mozilla Organization. Mozilla Roadmap. Disponível em http://www.mozilla.org/roadmap.html. 2001. Visitado em Abril de 2001.

[98] The Mozilla Organization. The mozilla.org Community. Disponível em http://www.mozilla.org/community.html. 2001. Visitado em Fevereiro de 2001.

[99] The Mozilla Organization. Mozilla Module Owners. Disponível em http://www.mozilla.org/owners.html. 2001. Visitado em Abril de 2001.

[100] The GIMP Group. GIMP Home. Disponível em http://www.gimp.org. 2001. Visitado em Abril de 2001.

[101] VIXIE, P. Software engineering. In: *Open Sources*. Sebastopol: O'Reilly and Associates, 1st. ed., 1999. p. 91–100.

[102] O'REILLY, T. Lessons from open source software development. *Communications of the ACM*, v. 42, n. 4, p. 33–37, April 1999.

[103] The Wine Project. Wine Development HQ. Disponível em `http://www.winehq.com`. 2001. Visitado em Janeiro de 2001.

[104] STARK, G., SKILLCORN, A., OMAN, P., AMEELE, C. R. An examination of the effects of requirements changes on software maintenance releases. *Journal of Software Maintenance Research and Practice*, v. 11, p. 293–309, 1999.

[105] ZHAO, L., ELBAUM, S. A survey on quality related activities in open source. *ACM SIGSOFT Software Engineering Notes*, v. 25, n. 3, p. 54–57, May 2000.

[106] VAN DER HOEK, A. Configuration management and open source projects. In: Proceedings of ICSE Workshop: Software Engineering over the Internet, 2000. IEEECSP.

[107] CUBRANIC, D. Coordinating open source software development. In: Proceedings of IEEE WETICE Workshop: Coordinating Distributed Software Projects, 1999.

[108] WILSON, G. Is the open-source community setting a bad example? *IEEE Software*, v. 16, n. 1, p. 23–25, January/February 2000.

[109] MCDONALD, J., HILFINGER, P. N., SEMENZATO, L. PRCS: The Project Revision Control System. In: Proceedings of the International Symposium on System Configuration Management, 8., Brussels. c1998.

[110] Collab.net. CVSHome.org. Disponível em `http://www.cvshome.org`. 2001. Visitado em Abril de 2001.

[111] The Mozilla Organization. Bugzilla. Disponível em `http://bugzilla.mozilla.org`. 2001. Visitado em Abril de 2001.

[112] The Free Software Foundation. GNATS. Disponível em `http://www.gnu.org/software/gnats/gnats.html`. 2001. Visitado em Abril de 2001.

[113] CUBRANIC, D. Open-source software development. In: Proceedings of ICSE Workshop: Software Engineering over the Internet, 1999. IEEECSP.

[114] ZACHMAN, J. A framework for IS Architecture. *IBM Systems Journal*, v. 26, n. 3, p. 276–292, 1987.

[115] LEHMAN, M. M., RAMIL, J. F., WERNICK, P., PERRY, D., TURSKI, W. Metrics and laws of software evolution: The nineties view. In: Proceedings of ISMS (Metrics'97), 4., 1997, Albuquerque.

[116] BASILI, V. R., WEISS, D. M. A methodology for collecting valid software engineering data. Technical report, TR-1235, College Park, Maryland, 1982.

[117] LAKATOS, E. M., DE ANDRADE MARCONI, M. *Fundamentos de metodologia científica*. 3rd. ed. São Paulo: Atlas, 1991.

[118] DE BARROS, A. P., DE SOUZA LEHFELD, N. *Fundamentos de metodologia: Um guia para a inciação científica*. 1st. ed. São Paulo: McGraw-Hill, 1986. p. 90–110.