UNIVERSIDADE DO VALE DO RIO DOS SINOS
CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
CURSO DE INFORMÁTICA

Uma Ferramenta para Gerenciamento de Requisitos em Projetos Baseados em Extreme Programming

Pablo Dall'Oglio

Prof. Sérgio Crespo Coelho da Silva Pinto
Orientador

Monografia submetida como requisito parcial para a obtenção do título de Bacharel em Informática.

São Leopoldo, Novembro de 2006

Resumo

Muitas metodologias surgiram nas últimas décadas para organizar o desenvolvimento de projetos de software. A Maioria delas inspiradas em metodologias tradicionais da engenharia, cujos princípios baseiam-se na previsibilidade dos requisitos. Como previsibilidade em software nem sempre é possível, nos últimos anos vemos um interesse crecente na adoção de metodologias que suportem processos adaptativos de concepção de software, as chamadas Metodologias Ágeis.

A Metodologia ágil mais proeminente atualmente é *Extreme Programming* (XP). Projetos em *Extreme Programming* são divididos em ciclos de desenvolvimento e seu planejamento é realizado ciclo à ciclo, de forma evolutiva. A principal característica desta metodologia é a aplicação de um conjunto de princípios que garantem agilidade ao processo.

Extreme Programming é uma metodologia que introduz uma dinamicidade grande ao processo de concepção de software. Entretanto, poucos esforços são feitos para se manter o registro dos requisitos de software, bem como garantir a rastreabilidade dos mesmos. Uma solução para garantir tal rastreabilidade, é organizar a informação já produzida por estes projetos, através da utilização de uma ferramenta que venha a prover armazenamento e gerenciamento da especificação de requisitos e consequentes decisões de projeto.

O presente trabalho busca desenvolver uma ferramenta para auxiliar o gerenciamento de requisitos de software em projetos que utilizam a metodologia *Extreme Programming*, de forma que suporte os papéis que interagem no processo e venha a prover rastreabilidade, acompanhamento das iterações, estatísticas e pontos de verificação, de acordo com os princípios da metodologia.

Abstract

Many methodologies appeared in the last decades to organize the development of software projects. Most of them inspired in traditional engineering methodologies, whose principles are based on requirements previsibility. Once that previsibility in software development is not always feasible, in the last years we've seen a growing interest in the adoption of methodologies that support adaptive process for software conception, also known as Agile Methodologies.

Te most proeminent methodology nowadays is Extreme Programming (XP). Projects that follow Extreme Programming principles are divided in development cicles and its planning is done cicle by cicle, in an evolutionary way. The main feature of this methodology is the application of a group of principles that ensure the agility of the process.

Extreme Programming is a methodology that introduces a great dinamicity to the process of software creation. However, just a few effords are done to keep the record of software requirements, as well as to ensure its traceability. One solution to ensure this traceability is to organize the information that is already produced by these projects, through the usage of a tool that provides storage and management of the requirements specification and also, consequent project decisions.

The goal of this work is the development of a tool to support the requirements management of software projects that use the Extreme Programming methodology, supporting the roles that interact in the process, providing traceability, iteractions tracking, statistics and checkpoints, according to the methodology principles.

Sumário

Índice de Figuras

Índice de Tabelas

1 Introdução

Para [Leite, 2001], nos últimos anos a sociedade tem passado a ver cada vez mais software como parte de suas vidas e isto tem gerado também uma maior demanda por qualidade, que por sua vez possui um custo, ainda não compreendido. O maior desafio da engenharia de software atual está justamente em prover tal qualidade com os recursos limitados disponíveis.

De acordo com uma pesquisa publicada no ano de 2002 pelo *Standish Group*, apenas 34% dos projetos em software são considerados como concluídos com sucesso, 15% são tidos como fracassados e 51% como não atingindo completamente os objetivos para os quais foram propostos.

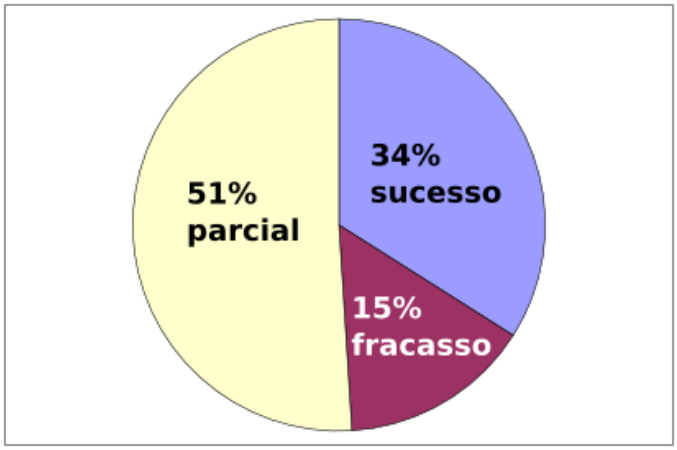


Figura 1 -Pesquisa sobre projetos do Standish Group

Muitas metodologias surgiram ao longo do tempo para organizar o desenvolvimento de projetos de software, provendo mais qualidade. A maioria delas inspiradas em metodologias tradicionais da engenharia, cujos princípios baseam-se na separação entre projeto e construção e na previsibilidade dos requisitos. Ou seja, todos os requisitos devem ser levantados na etapa de projeto [Fowler, 2003].

Em se tratando de software, nem sempre é possível obter com exatidão todos os requisitos de um projeto antes de sua construção. Muitos fatores subjetivos conspiram neste sentido. Em geral, os requisitos mudam, uma vez que o cliente só consegue validar um processo após vê-lo implementado. Neste caso, é necessáiro saber o que fazer quando da mudança [Beck, 2003].

Como a previsibilidade nem sempre é possível, uma das soluções é adotar metodologias que suportem processos adaptativos ou evolutivos de concepção de software, as chamadas Metodologias Ágeis. Um dos princípios chave das Metodologias Ágeis é o *feedback* do cliente após curtas iterações através do uso da prototipação. Desta forma, temos uma ferramenta eficiente de validação de requisitos [Highsmith, 2001].

A Metodologia ágil de maior adoção atualmente é *Extreme Programming* (XP). Formada por um conjunto de valores, é centrada em codificação ao invés de

documentação, utiliza uma série de técnicas da engenharia de software, como *refactoring*, *unit tests*, todas de forma incremental, e possui a figura do cliente como priorizador de tarefas, o qual guia o desenvolvimento de acordo com os interesses do negócio [Leite, 2001].

Como principal ferramenta para análise de requisitos, *Extreme Programming* se utiliza de *user stories*, que são fichas contendo relatos do cliente produzidas a partir de seções de *brainstorming*. Cada relato deve ser priorizado, testado, estimado e codificado, utilizando as técnicas acima descritas, de forma incremental, alternando ciclos de iteração e produzindo constante *feedback* para o cliente [Horriant, 2004].

Projetos em *Extreme Programming* são divididos em ciclos de desenvolvimento, o planejamento de implementação das *user stories* é feito ciclo a ciclo, tomando por base a produtividade do ciclo anterior e se adaptando de forma flexível à possíveis mudanças de requisitos por parte do cliente [Ambler, 2004].

Como visto, a mudança de requisitos é bem vinda em projetos *Extreme Programming*, uma vez que o processo se adapta à mudança [Beck, 1999]. *Extreme Programming* é uma metodologia que introduz uma dinamicidade grande ao processo de concepção de software. Entretanto, poucos esforços são feitos para se manter o registro dos requisitos de software, bem como garantir a rastreabilidade dos mesmos [Leite, 2001].

Uma solução para garantir tal rastreabilidade, é a organização da informação já produzida por estes projetos, através da utilização de uma ferramenta que venha a prover armazenamento e gerenciamento da especificação de requisitos e consequentes decisões de projeto. As informações produzidas, armazenadas e processadas a partir das *user stories*, podem ser utilizadas para avaliação de qualidade, seguindo determinadas métricas [Duncan, 2001].

1.1 Objetivos

O presente trabalho busca desenvolver uma ferramenta para auxiliar o gerenciamento de requisitos de software em projetos que utilizam a metodologia *Extreme Programming*, de forma que suporte os papéis que interagem no processo e venha a prover rastreabilidade, acompanhamento das iterações, estatísticas e pontos de verificação, de acordo com os princípios da metodologia.

1.2 Método

1) Identificar e analisar as principais técnicas de elicitação de requisitos em projetos *Extreme Programming*;
2) Identificar e analisar as principais abordagens de mudança de requisitos em projetos *Extreme Programming*;
3) Identificar e analisar as principais técnicas de validação de requisitos em projetos *Extreme Programming*;
4) Construir uma ferramenta que auxilie na gerência de requisitos em projetos *Extreme Programming*;

2 Metodologias Ágeis

2.1 Introdução

De acordo com [Fowler, 2004], a maioria do desenvolvimento de software atualmente é realizado sem muito planejamento e permeado por decisões rápidas. Este estilo de desenvolver software pode funcionar muito bem enquanto que o software tem pequenas dimensões, mas a medida que vai crescendo, torna-se muito difícil corrigir seus bugs ou mesmo adicionar novas funcionalidades.

Felizmente, a engenharia de software nos oferece uma gama de metodologias para organizar o desenvolvimento de aplicações. Tais metodologias geralmente impõem a utilização de algum processo formal e detalhado, com a intenção de tornar a tarefa de desenvolvimento mais previsível.

As metodologias tradicionais apresentam processos que objetivam a definição dos requisitos de uma aplicação antes da mesma ser construída. Geralmente a falha nestes projetos estão ligadas à definições erroneas sobre estes requisitos ou ao não entendimento entre as partes, cliente e fornecedor. Para [Fowler, 2004], é um erro pensar que é possível elencar e estimar com precisão todos requisitos de uma aplicação antes de sua construção. Os requisitos sempre mudam, por que o cliente gera novas demandas e novas interpretações sobre o que considera desejável na medida em que vê os resultados e usa as primeiras versões do software. Mesmo que se pudesse definir um conjunto de requisitos com precisão, a própria natureza dinâmica da economia atual determina constantes mudanças sobre o que realmente tem valor para os negócios, o que influencia diretamente nas aplicações, que devem suportar estes processos.

Para [Miller, 2001], há pouquíssimos projetos em software que não passam por mudanças devido aos negócios, à tecnologia ou riscos. Neste caso, há duas opções: resistir ou adaptar. A primeira opção geralmente é escolhida por aqueles cujos projetos devem seguir um planejamento. Esta opção muitas vezes desagrada ao cliente. Como a meta de todo software deve ser a adequação às necessidades do cliente, se as necessidades mudam, o software também deve mudar. Assim, a segunda opção, a adaptação, é a melhor saída nestes casos.

Para [Fowler, 2004], a forma mais adequada de se estar preparado para as mudanças é através da adoção de um processo que suporte constante *feedback* através do desenvolvimento iterativo, ou também chamado de espiral. Esta abordagem remonta à prototipação, na qual são apresentadas ao cliente pequenas porções do software contendo um pequeno conjunto de funcionalidades, para que o mesmo as valide já nos estágios iniciais do projeto. O desenvolvimento iterativo é uma forma adequada de se lidar com requisitos em um ambiente adaptativo, onde o planejamento é realizado iteração pós iteração.

Diferentemente das metodologistas tradicionais, onde existe claramente uma grande etapa inicial de análise (*big up front*) e uma certa resistência à mudança de requisitos, surge uma nova onda na engenharia de software, que são as Metodologias Ágeis. Um

dos pontos que a diferenciam das outras é justamente à abordagem referente ao que fazer quando da mudança de requisitos. Para [Highsmith, 2001], a nova estratégia não é evitar a mudança, mas aceitá-la como parte do processo, estando preparada para quando ela ocorrer, reduzindo os custos de sua implementação. Uma das metodologias desta nova onda é *Extreme Programming*, que através de suas práticas e princípios dá uma resposta rápida às necessidades dinâmicas dos negócios atuais.

Os princípios das metodologias ágeis foram elaborados a partir de um manifesto escrito pelos profissionais mais proeminentes nesta área (Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland e Dave Thomas) . Alguns dos valores propostos no manifesto, de acordo com [Highsmith, 2001] são:

- Indivíduos e interações ao invés de processos e ferramentas;
- Aplicação em funcionamento ao invés de documentação;
- Colaboração do cliente ao invés de negociações de contrato;
- Responder à mudança ao invés de seguir o planejamento;

Uma das grandes contribuições das metodologias ágeis é a colocação da pessoa em primeiro plano e o reconhecimento de que ela é o fator primário do sucesso de um projeto. O foco nas interações entre os indivíduos ao invés do foco no processo facilita o compartilhamento da informação e do conhecimento do grupo, o que torna mais rápidas as mudanças, além de minimizar os impactos da não utilização extensiva de documentação. Outro fator que suaviza o impacto da mudança é a presença constante do cliente como membro da equipe, o que permite reagir com velocidade às mudanças, ao invés de seguir o planejamento inicial, que na maioria das vezes se torna obsoleto rapidamente [Highsmith, 2001].

Para [Miller, 2001], a indústria de software ainda não assimilou os novos conceitos do movimento ágil. A maioria das pessoas ainda pensa em software como um *commodity*. Enquanto a maioria ainda pensa em variáveis como preço, o movimento ágil se preocupa com as interações entre as pessoas, na criação de um ambiente favorável e principalmente na satisfação do cliente.

2.2 Exemplos de Metodologias

A seguir, temos relacionadas algumas metodologias ágeis.

2.2.1 Scrum

Scrum foi aplicado pela primeira vez no ano de 1993 por Jeff Sutherland, John Scumniotales e Jeff McKenna em uma empresa chamada Easel [Fowler, 2004]. Scrum é uma metodologia voltada ao gerenciamento do processo de desenvolvimento de software baseada em princípios como flexibilidade, adaptabilidade e produtividade. Seu foco é o desenvolvimento de um trabalho de qualidade em um ambiente sucetível à constantes mudanças. Paral tal, Scrum se utiliza de técnicas como *backlog, sprints, e daily scrums*. O *backlog* consiste em uma lista priorizada de requisitos, melhorias e bugs do sistema. *Sprint* é uma jornada, pode ser comparada à uma iteração, e é geralmente de 30 dias. A priorização de tarefas por parte do cliente pode acontecer de um *Sprint* para outro, mas não durante a execução do mesmo [Paetsch, 2003].

2.2.2 Crystal

Crystal é um conjunto de metodologias criadas por Alistair Cockburn, que iniciou seus primeiros ensaios à pedido da IBM no início dos anos 90. A metodologia compartilha alguns princípios como a orientação à pessoa, de *Extreme Programming* [Fowler, 2004]. A partir de uma "família de metodologias", se escolhe a mais apropriada para cada projeto. Dentre estas, as mais utilizadas compartilham alguns princípios, como: entrega incremental em períodos de tempo; testes de regressão e funcionais; *reviews* e *workshops* de produto e metodologia. O espírito da metodologia é escolher o caminho menos disciplinado pelo qual o projeto possa ser um sucesso [Paetsch, 2003].

2.2.3 FDD

FDD (*Feature Driven Development*) foi criado por Jeff De Luca e Peter Coad [Fowler, 2004]. É um processo de software baseado em curtas iterações, focado nas etapas de projeto e implementação. Em uma primeira etapa, o sistema é modelado por experts de cada área de negócios, através de diagramas de classes contendo métodos, relacionamentos e atributos. Os métodos são a base para a construção de uma lista de *features,* que são características de valor para o cliente, priorizadas pela equipe. Equipes que utilizam FDD se reunem uma vez por semana para discutir o estado de cada *feature*, como uma forma de acompanhamento de requisitos [Paetsch, 2003].

2.2.4 DSDM

DSDM (*Dynamic System Development Method*) começou a ser desenvolvido na Inglaterra em 1994 por um consórcio de empresas. Possui grande organização, treinamentos e suporte comercial [Fowler, 2004]. Consiste em um *framework* para o desenvolvimento rápido de aplicações. Durante seus primeiros estágios, são descobertos os requisitos básicos do sistema, que, no decorrer do processo de desenvolvimento são detalhados. Assim como em *Extreme Programming*, testes permeiam todo o processo de desenvolvimento [Paetsch, 2003].

2.2.5 ASD

ASD (*Adaptive Software Development*) consiste em um *framework* para desenvolvimento de aplicações complexas e de grande porte. Imprime diversos princípios ágeis, como o desenvolvimento incremental e a constante prototipação. Seus estágios iniciais são curtos e objetivam garantias e envolvimento do cliente. Após o término de cada ciclo ocorrem reuniões com o cliente e revisões em grupo [Paetsch, 2003]. A metodologia foi criada por Jim Highsmith após ter passado muitos anos ensinando metodologias formais e acabando por descobrir que elas não são o caminho correto a seguir em projetos atuais do mundo dos negócios. Sua visão é fortemente baseada na natureza adaptativa das metodologias e na teoria do caos [Fowler, 2004].

2.2.6 XP

XP (*Extreme Programming*) é a metodologia ágil de maior destaque e notoriedade. Suas raízes estão na comunidade Smalltalk ainda no final dos anos 80, em pessoas como Kent Beck e Ward Cunningham que foram refinando suas práticas ao longo de vários projetos que participaram ao longo do tempo. A primeira grande prova de uso da metodologia ocorreu em um projeto liderado por Kent na Chrysler no ano de 1996. A partir de então a metodologia vem crescendo em adoção à cada ano [Fowler, 2004]. O próximo capítulo será dedicado exclusivamente à metodologia *Extreme Programming*.

3 Extreme Programming

3.1 Introdução

Extreme Programming é uma metodologia ágil para times pequenos e médios face à rápida mudança de requisitos. Para [Teles, 2004], a metodologia parte da premissa de que o cliente aprende sobre suas necessidades à medida em que manipula o sistema que está sendo produzido. De acordo com [Beck, 2004], *Extreme Programming* é baseada em alguns princípios de senso comum, dentre eles:

1. Constante revisão de código através da programação em pares;
2. Constantes testes de unidade, o desenvolvimento é dirigido por testes;
3. Constante *design* (*refactoring*);
4. Simplicidade no *design* (utilizar o *design* mais simples possível);
5. Constante refinamento de arquitetura;
6. Constantes testes de integração;
7. Iterações curtas;

Extreme Programming se diferencia das metodologias tradicionais por prover *feedback* contínuo através de curtos ciclos de iteração, ter uma abordagem incremental, tornar o cronograma flexível respondendo às mudanças nas regras de negócio, e sua confiança em testes automatizados, comunicação oral, *refactoring* e a colaboração entre os desenvolvedores [Beck, 2004].

Extreme Programming, através de seus princípios, procura combater os possíveis riscos existentes em um projeto de software de forma simples, através de sua abordagem incremental e iterativa, na qual o cliente participa ativamente do desenvolvimento do software, tendo sempre a melhor percepção de retorno possível (*feedback*) e tomando decisões que definam quais são os requisitos a serem implementados que lhe garantam maior benefício, se adequando às mudanças no negócio [Teles, 2004].

De acordo com o pensamento clássico de Boehm, o custo de uma mudança em um projeto de software cresce exponencialmente ao longo do tempo, ou seja, quanto mais tarde um problema for detectado, mais recursos serão consumidos para sua resolução.

Para [Highsmith, 2001], a preocupação atualmente reside em como estar preparado para as mudanças que ocorrem durante a execução do projeto, e não em evitar tais mudanças. Para os metodologistas tradicionais, se conseguirmos elencar um conjunto concreto de requisitos no início do projeto e seguirmos esta análise durante sua execução, praticamente anulamos a necessidade de mudança em um projeto. Mas para [Highsmith, 2001], esta rigidez em responder à mudanças do ambiente de negócios do cliente é o que leva muitas vezes um projeto ao fracasso. Abaixo, temos a tradicional curva do custo da mudança.

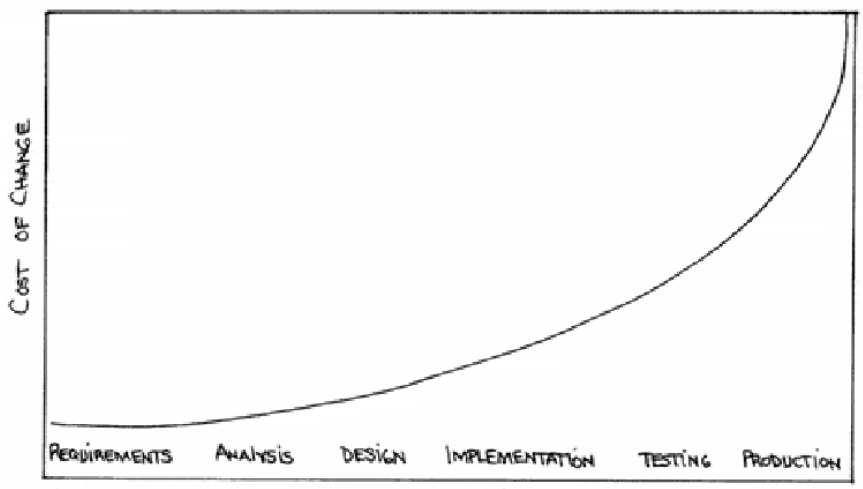


Figura 2 -Curva tradicional do custo da mudança [Beck, 2004]

Para [Beck, 2004], atualmente contamos com uma série de ferramentas e técnicas que minimizam o custo da mudança mesmo em estágios avançados do desenvolvimento. Ambientes de simulação, ferramentas de prototipação, padrões de projeto, testes de unidade, constante *refactoring*, dentre outros, fazem com que a mudança seja inofensiva dentro do contexto do sistema. Uma das premissas de *Extreme Programming* é a de que o custo da mudança cresce suavemente ao longo do tempo. Dessa forma, o cliente não teria medo de deixar para um segundo momento a tomada de uma decisão importante dentro do projeto, para diminuir o risco de errar.

[Fowler, 2004] acredita que a curva possa ser "achatada" através da adoção destes vários princípios e práticas que tornam possível a adoção do *design* evolucionário sem acarretar grandes custos da mudança. Dentre estas práticas, os testes automatizados, a integração contínua e os *refactorings*, dão subsídios à equipe para que esta possa fazer as mudanças necessárias tendo a confiabilidade de que o sistema ainda está íntegro. Abaixo, a curva do custo da mudança em *Extreme Programming*.

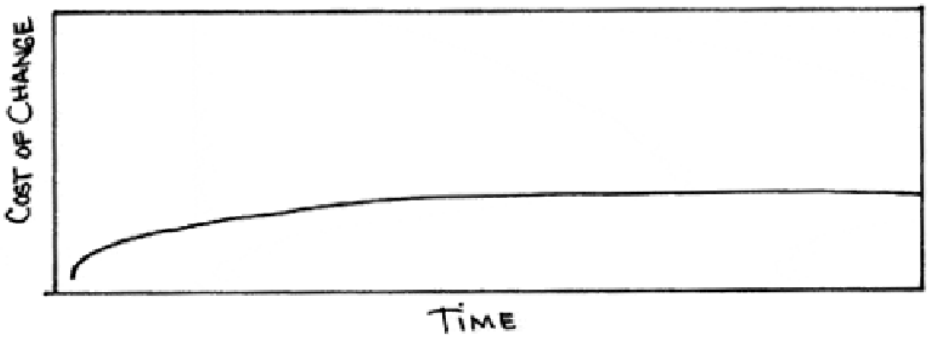


Figura 3 -Curva do custo da mudança em XP [Beck, 2004]

Diferentemente das metodologias tradicionais de engenharia de software, nas quais o processo de desenvolvimento tem muitos inputs, *Extreme Programming*, prevê o mínimo necessário para que o processo inicie. Após iniciado o próprio cliente o acompanha, iteração após iteração, alimentando o processo com seu *feedback*. Assim, o processo *Extreme Programming* tende a ser mais adaptável às mudanças surgidas durante o desenvolvimento do software, visto que o processo se realimenta [Beck, 2004].

3.2 Valores

De acordo com [Beck, 2004], são quatro os valores de *Extreme Programming*:

3.2.1 Comunicação

Uma considerável quantidade de falhas em projetos de software é decorrente de problemas de comunicação. Muitas vezes um programador não comunica o outro de uma mudança importante, alguém esquece de perguntar algo relevante ou faz uma interpretação errada. *Extreme Programming* faz uso de diversas técnicas que simplesmente não funcionam em ambientes sem comunicação, como *pair programming*. Para [Teles, 2004], XP busca garantir que a comunicação ocorra da forma mais direta e eficaz possível.

3.2.2 Simplicidade

Um dos valores mais difíceis de serem assimilados. Muitas vezes os programadores pensam à frente, arquitetando soluções para possíveis problemas futuros. Para *Extreme Programming* funcionar, é necessário focar nos problemas presentes e elaborar soluções da maneira mais simples, desde que elas atendam os requisitos do cliente.

3.2.3 Feedback

Extreme Programming baseia-se em implementar para o cliente primeiramente os requisitos mais importantes para o seu negócio, para que o mesmo possa dar sua opinião, gerando novos inputs, novos requisitos em um fluxo constante. Este *feedback*, dá maior credibilidade às decisões tomadas, uma vez que se toma ciência rapidamente dos erros e dos acertos, diferentemente de outras abordagens que permitem esta verificação somente no final. Para [Teles, 2004], o *feedback* é o mecanismo que permite o cliente conduzir o desenvolvimento do software diariamente.

3.2.4 Coragem

Extreme Programming incentiva a mudança. Para mudar, é necessário coragem. Muitas vezes um sistema grande e complexo está funcionando bem, mas não da melhor maneira. Nestes casos é necessário muita coragem para mudar o que já está funcionando, para facilitar futuras alterações, tornando o código mais legível.

3.3 Princípios

Os valores até agora mencionados são um tanto quanto vagos e sua interpretação pode variar de pessoa para pessoa. Portanto, [Beck, 2004] sugere princípios baseados nestes valores. Estes princípios transmitem uma idéia mais concreta e são de fácil aplicação.

- **Feedback Rápido:** O tempo decorrido dentre uma ação e sua reação é fundamental para o aprendizado. Um dos princípios de XP é prover um rápido *feedback*, do cliente em relação às mudanças do software e dos programadores em relação à si mesmos, para que o aprendizado ocorra rapidamente.
- **Assuma a Simplicidade:** A maioria dos problemas em aplicações de negócio podem ser solucionaods de forma simples. Os programadores tendem à projetar soluções complexas que possam atender à demandas futuras que ainda nem surgiram. Em XP, deve-se procurar fazer bem o serviço de hoje de forma simples, confiando na capacidade de se adicionar novos recursos.

- **Mudança Incremental:** Grandes mudanças realizadas de uma só vez não funcionam. XP adota a visão incremental. As mudanças devem ser feitas aos poucos, desde mudanças no planejamento, quanto no projeto ou no código.
- **Adote a Mudança:** As mudanças devem ser bem vindas em qualquer etapa do projeto.
- **Trabalho de Qualidade:** XP assume que a variável de Qualidade de um projeto não é tão variável assim e pode assumir valores de ótimo e excelente. Todos gostam de desenvolver bons trabalhos. Então as pessoas devem ser motivadas continuamente a desenvolver bons trabalhos, o que as deixam motivadas também.
- **Ensine Aprendendo:** Ao invés de oferecer uma série de dogmas, XP foca em ensinar estratégias onde se aprenda o quanto projetar, refactorar, dentre outros.
- **Pequeno Investimento Inicial:** Em XP, um grande investimento financeiro inicial, que proporciona um grande time é muito mais prejudicial que um pequeno investimento. Uma grande equipe no início se torna ingerenciável. O trabalho deve começar bem focado e crescendo aos poucos, bem como o investimento do negócio, que garanta a continuidade e o crescimento.
- **Jogue para Ganhar:** Uma equipe XP deve estar motivada e sempre disposta a realizar o melhor dentro de um projeto. Em projetos tradicionais, muitas equipes realizam trabalhos baseadas em extensas documentações de projetos, apenas para garantir que estão fazendo um bom trabalho e ter desculpas caso o projeto falhe. XP não busca isto. A equipe XP sabe que o projeto dará certo e não buscará artefatos para se proteger no caso de falhas.
- **Experimentos Concretos:** Quanto mais decisões forem tomadas sem testes em um projeto XP, maior a probabilidade de ocorrerem problemas. Quaisquer decisões precisam ser testadas antes de serem colocadas em produção.
- **Comunicação Honesta:** A comunicação entre os membros da equipe de um projeto XP deve ser honesta e franca. Muitas vezes, conversas privadas ou paralelas são sinal de que um projeto não vai bem. Pessoas gostam de confiar e serem confiadas, gostam de trabalhar em grupo e realizar bons trabalhos. As pessoas devem exercitar a liberdade de expressar suas opiniões, principalmente aquelas que externam críticas, conselhos e sugestões que apontem possíveis falhas em um projeto.
- **Aceite a Responsabilidade:** Em equipes XP a responsabilidade na execução de tarefas deve ser aceita pelos seus membros, não dada sob pressão. Uma tarefa delegada pode gerar grande frustração. Em uma equipe, os membros devem assumir seus papéis e aceitar as tarefas de acordo com seus perfis e suas capacidades.
- **Adaptação Local:** XP não oferece um conjunto de regras a serem seguidas a risca. Pelo contrário, equipes que forem adotar XP, devem adaptá-lo a sua realidade e construir seu próprio processo.

- **Seja Nomade:** Esteja sempre preparado para mudanças, tanto tecnológicas, quanto em relação aos ambientes de trabalho, equipes, clientes, etc. Leve consigo apenas o que realmente seja simples e tenha valor.
- **Medidas Honestas:** É necessário ter muito cuidado ao se utilizar medidas em projetos XP. Porque muitas das medidas tradicionais simplesmente não são condizentes com os princípios defendidos por XP. Medidas como número de linhas de código ou número de horas exatas para uma tarefa, não refletem a realidade proposta pela metodologia.

3.4 Práticas

De acordo com [Beck, 2004], para um projeto *Extreme Programming* funcionar é necessário seguir algumas práticas. A não utilização de uma das práticas de XP, pode comprometer as demais, tendo em vista sua inter-dependência, além de adicionar mais riscos ao projeto. [Beck, 2004] acredita que projetos tenham ganhos reais em usar apenas algumas práticas de XP, mas acredita também que os ganhos são muito maiores ao se adotar todas as práticas conjuntamente, pois sua sinergia faz com que a soma das partes seja maior que o todo. As práticas são as seguintes:

- **Jogo do Planejamento:** Para [Teles, 2004], um projeto XP é dividido em "*Releases*" (que são versões bem definidas contendo módulos de valor para cliente) e estes em "Iterações" (que são os ciclos de desenvolvimento de um "*Release*". O Jogo do Planejamento, originalmente chamado por "Planning Game", consiste nas atividades necessárias para elaborar o escopo do próximo *release* ou iteração, baseado nas prioridades de negócio, restrições e estimativas de desenvolvimento.
- **Stand Up Meeting:** Para [Teles, 2004], *Stand Up Meetings* são reuniões rápidas no início de cada dia de trabalho, onde a equipe reavalia o que foi realizado no dia anterior e dá as prioridades necessárias
- **Pequenos Releases:** Para [Beck, 2000], todo *release* deve ser o menor possível, contendo as *stories* com maior valor para o cliente. XP tem o objetivo de gerar constantemente valor para o cliente, produzindo um conjunto reduzido de funcionalidades para que o cliente possa utilizar no seu dia-a-dia, produzindo *feedback* [Teles, 2004].
- **Metáfora:** Para [Beck, 2000], o desenvolvimento de um projeto XP deve seguir uma metáfora. Calculadoras, planilhas e ícones são exemplos de metáforas utilizadas para simplificar o entendimento de idéias. Para [Teles, 2004], as metáforas tem a vantagem de transmitir idéias complexas de forma simples.
- **Design Simples:** O *design* de um sistema deve ser simples o suficiente que permita rodar os testes automatizados, não tenha duplicação de código, tenha a menor quantidade possível de classes e métodos e represente o que seja realmente significativo. A primeira meta em se utilizar um *design* simples é comunicar a intenção do desenvolvedor e em segundo, prover uma estrutura lógica que permita o sistema ter longa-vida [Beck, 2000]. Para [Teles, 2004],

utilizado o *design* simples, evita-se a criação excessiva de generalizações dentro do código fonte, preparando o sistema para modificações futuras.

- **Testes:** Para [Teles, 2004], o desenvolvimento de sistemas que utilizam XP é guiado por testes, ou seja, cada funcionalidade tem um teste escrito antes de ser codificada. Para [Beck, 2004], testes automatizados são escritos para garantir a qualidade e aumentar a confiança da equipe, uma vez que os resutados dão subsidios à equipe continuar no desenvolvimento.
- **Refactoring:** Para [Beck, 2000], muitas vezes, ao implementar determinada tarefa, o programador se confronta com a necessidade de mudar o código existente para suportar uma nova característica. *Refactoring* significa, dentre outras coisas, simplificar a estrutura, mudar a organização para que a manutenção do código seja uma tarefa menos árdua, sem que isto altere seu comportamento. Para [Teles, 2004], *refactoring* é utilizado para facilitar o entendimento e a manipulação do software e utiliza fortemente os testes automatizados para garantir que a mudança não irá ter impactos negativos.
- **Pair Programming:** Toda produção de código em XP é realizada por dois programadores utilizando o mesmo equipamento, enquanto que um escreve o código, o outro age de forma estratégica, supervisionando o trabalho efetuado, pensando em possíveis testes a serem construídos e possíveis *refactorings*. A programação em pares traz muita produtividade à dupla, devido à comunicação estreita entre os pares. Um novo aprendizado tende a se espalhar bem mais rápido em equipes que utilizam esta prática. Esta prática tende também, a diminuir as diferenças técnicas entre os membros da equipe e ao longo do tempo, a elevar todos à um nível superior.
- **Propriedade Coletiva:** Em projetos XP, qualquer pessoa que vislumbre uma possível melhoria em alguma porção de código, deve ter a liberdade de realizar tal alteração. Diferentemente da propriedade individual, esta forma provê uma dinâmica maior, já que não é necessário submeter as modificações aos detentores da propriedade daquele código. Esta prática tende a trazer o caos, mas em XP funciona devido à utilização de testes automatizados que indicam à qualquer momento uma possível falha no sistema. A propriedade coletiva diminui a quantidade de código complexo no sistema, uma vez que todos precisam compreender o sistema. Sempre que um desenvolvedor encontrar um código complexo, deve tentar simplificá-lo através de *refactoring* [Beck, 2000].
- **Ritmo Sustentável:** Para garantir a agilidade de um projeto XP é fundamental que a equipe esteja focada em seus objetivos e principalmente determinada, capaz e com criatividade para solucionar problemas [Teles, 2004]. Neste sentido, é de suma importância manter um ritmo sustentável de trabalho, evitando-se extrapolar as horas de trabalho, o que é essensial para se trabalhar na plenitude da mente.
- **Integração Contínua:** Para [Beck, 2000], o código deve ser integrado e testado constantemente após o desenvolvimento de novas características. Tal integração pode ser automatizada através de um ambiente preparado para tal. Cada alteração no sistema geralmente reflete no código de outro desenvolvedor. A integração dá garantias para o desenvolvedor prosseguir com o desenvolvimento

enquanto a integridade do sistema se mantém, garantida através de um conjunto de testes. Assim, a integração contínua reduz o risco do projeto.

- **Semana de 40 horas:** Para que a equipe esteja sempre em sintonia com o projeto, traga novas idéias e esteja constantemente motivada e preparada para o trabalho, é necessário tomar cuidado com a carga de trabalho. Embora diferentes pessoas tenham diferentes níveis de tolerância, à médio prazo o trabalo em excesso é prejudicial e geralmente indica problemas no projeto. Deve-se evitar trabalhar mais do que 40 horas semanais.
- **Cliente Presente:** Todo projeto XP deve ter um cliente presente. Este cliente deve ser alguém que realmente represente os interesses e saiba dos requisitos que devem ser contemplados no sistema. Segundo [Huo, 2004], o cliente presente auxilia o desenvolvedor a refinar e a corrigir os requisitos durante todo o processo de desenvolvimento, enquanto que no modelo em cascata, os clientes são envonlvidos durante a fase de definição de requisitos e projeto do sistema.
- **Padrões de Código:** XP se baseia na propriedade coletiva, onde temos programadores atuando em todas partes do sistema. Assim, não se pode permitir diferentes padrões de escrita de código, uma vez que o entendimento do mesmo fica muito prejudicada. Para tal, XP adota padrões de código que devem ser adotados por todos os membros da equipe [Beck, 2000].

3.5 Papéis

De acordo com [Beck, 2000], o poder entre os responsáveis em um projeto de software deve ser balanceado. Homens de negócio devem tomar decisões de negócio e técnicos devem tomar decisões técnicas. Um dos problemas de falha em projetos de software são os prazos estimados de forma incondizente com a realidade. Estimativas devem ser realizadas por quem tem estas condições (técnicos), baseados em sua experiência prévia. Homens de negócio podem tomar decisões relacionadas à datas, escopo e priorização. Em *Extreme Programming*, o cliente é o responsável por tomar as decisões de negócio e o cliente é parte da equipe do projeto.

3.5.1 Cliente

O cliente, bem como o programador, é uma das duas principais figuras em *Extreme Programming*. Como o cliente nem sempre sabe exatamente o que quer no início do projeto, este deverá se envolver e aprender algumas atividades, como escrever *stories*, testes funcionais, trazer para o projeto mundanças no ambiente dos negócios e tomar decisões, como definir o que é mais importante, quais são as prioridades, quando os requisitos representam fielmente o que o sistema deve implementar, dentre outros. Para [Beck, 2000], o melhor cliente é aquele que irá utilizar o sistema a ser desenvolvido e possui conhecimento sobre o problema a ser resolvido. O cliente terá de trabalhar conjuntamente com a equipe de desenvolvimento, aprendendo quais são as informações mais relevantes que devem ser fornecidas e quais são desnecessárias.

Para [Beck, 2000], são deveres do cliente:

- Definir o escopo e as datas dos *releases*;
- Definir as prioridades das características propostas;

- Definir o escopo exato do que foi proposto;

3.5.2 Coach

O "*coach*", ou "treinador" é o responsável pelo processo. Responsável por chamar atenção de alguém da equipe quando esta desvia dos seus objetivos e de sua metodologia. Como todos na equipe devem conhecer *Extreme Programming*, o "*coach*" deve possuir grande domínio sobre a metodologia, fornecendo alternativas para resolução de problemas e traçando analogias com outras equipes que já utilizaram XP, além de tornar mais claros os fundamentos da metodologia. Para [Beck, 2000], o *coach* deve saber quando deve intervir, pois, segundo ele, a intervenção freqüênte na resolução de problemas pode causar uma grande dependência da equipe, que perderá um pouco da capacidade de trabalhar sem o "*coach*".

O papel do "*coach*" é fundamentalmente técnico, voltado à execução e à evolução do processo. Neste sentido, são importantes as habilidades relativas à comunicação. São suas principais funções:

- Servir como desenvolvedor parceiro para programadores iniciantes, novos membros da equipe;
- Incentivar práticas como *refactoring*, observando sempre objetivos de integração de longo prazo;
- Auxiliar os programadores tecnicamente em funções de testes, definição e *refactoring*;
- Comunicar o processo para as posições mais altas na hierarquia;

3.5.3 Tracker

O papel do "*tracker*" é de realizar estimativas e controlar se as atividades completadas estão conforme as expectativas iniciais de tempo. Seu papel é prover *feedback* sempre que o time estiver realizando estimativas, provendo informações sobre a *baseline* e maximizando a acuracidade das estimativas, tendo por base o que já foi feito, alertando sobre estimativas muito otimistas ou pessimistas. O "*tracker*" também é responsável por manter a equipe sintinizada com os macro objetivos do projeto, mantendo-se sempre atualizado em relação ao progresso dos *releases*, sendo capaz de alertar os desenvolvedores sobre possíveis desvios. O "*tracker*" é também a "alma" da equipe, pois ele mantém o histórico dos testes realizados, dos bugs, mantém o registro de quais equipes assumiram quais *stories* e quais tarefas foram adicionadas no decorrer da execução. Enfim, deve ter a habilidade de coletar a informação a partir dos membros da equipe e de mantê-los informados sobre os tempos decorridos e restantes, registrando as informações necessárias para alimentar as métricas escolhidas.

3.5.4 Testador

O papel do testador em *Extreme Programming* é voltado para auxiliar o cliente a escrever *stories* e testes funcionais, uma vez que grande parte da tarefa que engloba testes já é atribuída ao próprio programador, responsável por desenvolver orientado à testes. Também é função do testador rodar os testes funcionais e os testes de integração frequentemente, destinando os resultados para quem convir, garantindo assim o bom funcionamento dos testes.

3.5.5 Programador

Para [Beck, 2000], os programadores exercem papel fundamental em XP. Possuem papel similar ao de programadores como em outras metodologias, entretanto seu papel vai bem além deste, envolvendo comunicação com outros membros da equipe, como o cliente a fim de discutir a necessidade de implementar determinadas tarefas, envolve também confiabilidade que permita a colaboração mútua e a propriedade coletiva do código, envolve a escrita de testes para demonstrar o funcionamento do software, o uso de metáforas no desenvolvimento, sempre visando construir o software com o maior valor agregado possível para o cliente. Algumas habilidades, como a programação em pares, simplesmente não são necessárias em outras metodologias. Estas são algumas das principais funções do desenvolvedor:

- Estimar o tempo necessário para impementar uma funcionalidade;
- Indicar as consequênicas de alternativas técnicas;

3.6 Requisitos

A forma como *Extreme Programming* lida com seus requisitos é principalmente através de *story cards*. Este assunto será aprofundado nos capítulos 4 e 5.

3.7 As quatro variáveis

De acordo com [Schwaber, 2001], a gerência de projetos tradicional elenca quatro variáveis a serem controladas, Custo, Tempo, Qualidade e Escopo. É muito difícil traçar uma correlação entre estas variáveis. Sabe-se porém que tempo é uma variável sob domínio do cliente e que influencia escopo e qualidade. A qualidade tem duas faces, a qualidade interna do projeto (visível pelos programadores) e a externa (visível pelo cliente). Em certas ocasiões, pode-se "relaxar" os graus de qualidade, para alcançar maior velocidade. Para [Beck, 2004], mais recursos financeiros (equipe maior) nem sempre aceleram o desenvolvimento. Uma equipe muito grande no início do projeto pode se tornar ingerenciável. O time deve começar pequeno, entregando pequenas porções de software significativas para o cliente. Os recursos devem crescer ao longo do projeto, quando se gastará mais com qualidade e escopo, dentre outros.

De acordo com [Beck, 2004], no desenvolvimento de software a variável mais importante a ser controlada é o escopo. O escopo é a variável que muda com a maior freqüência e a de maior dificuldade de consenso. Os clientes tem dificuldade em formular os requisitos e os fornecedores tem dificuldade em entendê-los. Em *Extreme Programming*, como o cliente vê os resultados iteração após iteração, ele tem uma noção melhor dos requisitos já desenvolvidos, bem como terá um melhor embasamento para formular os próximos a serem implementados. *Extreme Programming* vê o escopo como um aliado, e baseia-se em sua flexibilidade. Como o planejamento é feito iteração após iteração, o escopo pode variar se alguma das outras variáveis mudar.

3.7.1 Custo

O custo é a variável mais independente dentre as quatro. O custo pode ser reduzido ou aumentado alterando-se as restrições de qualidade, o prazo ou até o escopo.

Para [Beck, 2000], uma forma de alterar o custo de um projeto é colocando mais pessoas na equipe do projeto. Para ele, no entanto, esta alternativa à longo prazo acaba

provocando efeitos colaterais como atrasos e excesso de comunicação advindos da equipe aumentada. “Dobrar o tamanho da equipe não irá dobrar a velocidade do projeto, pois isto aumenta a comunicação que precisa ser realizada”. Para o autor, o projeto pode até vir a se tornar mais lento, tendo em vista o aprendizado que o(s) novo(s) integrante da equipe terão de assimilar de outros membros e também sobre o sistema. Outras formas de se adicionar recursos ao projeto, como o investimento em ferramentas também pode tornar o processo lento, visto que os programadores terão que aprender à utilizá-la.

Para [Beck, 2000], a melhor forma de aumentar os recursos de um projeto é investir em formas de se manter a motivação, como em computadores mais rápidos ou monitores maiores.

3.7.2 Qualidade

Para [Beck, 2000], há dois tipos de qualidade, a externa (relativa à percepção do cliente) e a interna (relativa à qualidade técnica da aplicação). Para ele, os requisitos não-funcionais devem ser trazidos para o escopo do projeto em forma de *stories*, uma vez que o escopo é facilmente manipulado.

A qualidade interna do sistema reflete o quão bem planejada é sua arquitetura, quão bem escritos são seus testes, enfim, a qualidade interna permite manter a curva do custo da mudança estável. Relaxar os níveis de qualidade significa gasto em tempo à médio e longo prazo para corrigir problemas.

3.7.3 Tempo e Escopo

As variáveis Tempo e Escopo são as que apresentam maior facilidade de manipulação em um projeto, uma vez que as anteriores são mais rígidas. *Extreme Programming* assume que as variáveis Qualidade e Custo são fixas, o que permite trabalhar em nível de Tempo e Escopo, que são as variáveis mais importantes para o planejamento.

Para [Beck, 2000], é necessário tomar muito cuidado com a variável Escopo, já que desenvolvedores são constantemente tentados à adicionar novos recursos nas aplicações. Já a variável Tempo sofre consequências graves quando adicionamos mais Escopo, geralmente só percebidas quando o projeto está atrasado e não há muito o que fazer.

Para [Beck, 2000], a variável Tempo deve ser controlada com muita cautela para que se perceba facilmente os efeitos de novas *stories* e de novos requisitos. Este é um dos motivos pelos quais XP utiliza marcações bem definidas dentro dos *releases* do projeto, as iterações, pois segundo o autor, as iterações levam a equipe à observar constantemente a variável tempo, tendo uma noção clara das consequências advindas das mudanças no escopo.

4 Planejamento de Requisitos

4.1 Introdução

Quando trabalhamos em projetos de software razoavelmente grandes, se torna imprescindível planejar. Não planejamos para prever o futuro, principalmente quando tratamos de projetos de software, onde a única constante é a mudança. De acordo com [Beck, 2000], planejamos para garantir que a tarefa mais importante seja realizada no momento correto, para melhorar a coordenação entre pessoas e para dar respostas rápidas ao inesperado. Planos devem ser construídos de forma que as etapas nele contidas sejam plausíveis de serem realizadas, e não como instrumento de pressão sobre aqueles que irão realizá-las, quando os prazos são incondizentes com a realidade.

Para [Highsmith, 2001], de acordo com pesquisas realizadas sobre projetos de desenvolvimento de software, em mais da metade dos casos é difícil encontrar o planejamento inicial do mesmo. Isto porque, o sucesso da maioria dos projetos passa na maioria das vezes por satisfazer as necessidades do cliente, o que implica em grande parte dos casos na readequação do projeto (em termos de requisitos, escopo e tecnologia) durante a sua execução.

De acordo com [Beck, 2000], durante a realização de um projeto ocorrem eventos que mudam o seu planejamento. Não podemos evitar estes eventos, mas devemos adaptar o planejamento à estas ocorrências. Um projeto que está perfeitamente adequado ao seu planejamento pode indicar sinal de problemas por não ter se adaptado à estes eventos. Esta distorção entre o real e o planejado tende à ser desastrosa quando percebida pelo cliente.

Quando agimos sob qualquer tipo de planejamento, seja no trabalho ou seja na vida, é imprescindível que saibamos em que ponto estamos para atingir nossos objetivos. Em se tratando de software, esta necessidade fica ainda mais evidente. É necessário que todos envolvidos possam ter uma visão clara do que falta para atingir os objetivos. Paral tal, é necessário delinear pontos claros de verificação no decorrer do projeto (*milestones*) [Beck, 2000].

4.2 Planejamento em XP

Para [Miller, 2001], metodologias ágeis não são voltados à entrega de uma grande solução em um dado momento. Ao invés disto, focam em pequenos ciclos onde um conjunto de atividades são completadas. Estes pequenos ciclos se repetem inúmeras vezes, permitindo refinamentos, até a entrega final da aplicação. Estes ciclos ou iterações se constituem na unidade de tempo ideal para o planejamento (*milestones*), uma vez que possuem duração pré-determinada.

De acordo com [Beck, 2000], projetos de desenvolvimento de software funcionam em ciclos e um dos grandes desafios é sincronizar dois ciclos diferentes, o de negócios e o de desenvolvimento. O ciclo de negócios é dirigido por atividades relacionadas à publicações na imprensa, instalação, treinamento e cobranças. É um período que pode

levar meses de trabalho até sua finalização e que chamamos de *release*. Já os ciclos de desenvolvimento precisam ser menores que o ciclo de negócios para que o mesmo se adapte à ocorrência de eventos externos. Geralmente estes ciclos duram de uma à três semanas e são chamados de *iterações*.

Figura 4 -Releases e iterações

Como um período de semanas ainda é grande para o acompanhamento do cliente, utiliza-se uma medida de fácil entendimento para ambas as partes, a *story*. De acordo com [Beck, 2000], a *story* descreve uma funcionalidade com valor para o cliente, e não aspectos puramente técnicos. Características não funcionais levantadas pelo desenvolvedor podem vir a se tornar *stories*, mas somente com o consentimento do cliente, através da sua percepção de valor. A *story* deve ser facilmente compreendida por ambos clientes e desenvolvedores e escrita em linguagem natural. Por isto devem ser extremamente simples, ao passo que se implemente várias delas em uma iteração (ciclo de desenvolvimento), e que tenham a completeza necessária que permita sua implementação.

Número:_____	***Tipo [New/Fix/Enhance]***	
Data: _____	***Priority: _____***	
Requisito: _____	***Risk: _____***	***Estimate: _____***
Descrição:		
Notas:		

Date	Status	ToDo	Comment

Figura 5 -Exemplo de *user story* [Breitman, 2002]

4.3 Planejamento dos releases

De acordo com [Beck, 2000], o planejamento de cada iteração do desenvolvimento é realizado conjuntamente entre o cliente e a equipe de projeto. Nele decide-se como irá ser realizada a sincronização entre os ciclos de desenvolvimento e os *releases*, observando detalhes cruciais como datas contratuais. Esta etapa é chamada de

planejamento de *releases*. Nela, as *stories* são alocadas em iterações e estas em *releases*, bem como, se escolherá quais serão trabalhadas primeiramente.

Figura 6 -*Planning game* [Beck, 2000]

O planejamento em projetos XP é baseado principalmente no histórico do projeto. Assim ocorre com o planejamento das *releases*, onde cada *release* e suas iterações são baseadas na velocidade obtida no *release* anterior, tendo assim, um bom nível de precisão nas estimativas. De acordo com [Beck, 2000], a etapa menos precisa de um projeto XP, é o planejamento inicial, quando o projeto ainda não possui histórico que sirva de *baseline*. Apesar de ineficaz em relação à estimativas, principalmente em relação à velocidade da equipe e ao tamanho das *stories*, o planejamento inicial é de grande importância, pois mantém a equipe focada em seus objetivos, apesar de que talvez não sejam todos cumpridos, servirá de *baseline* para os planejamentos subsequentes.

[Leite, 2001] acredita que todo o processo de construção de software deve lidar com os aspectos evolutivos de maneira organizada, estando ancorado sobre uma *baseline*, que nada mais é do que um conjunto de cenários (*stories*). Para o autor, tais representações evoluem e a complexidade reside justamente em registrá-las para garantir sua rastreabilidade, principalmente num contexto volátil.

A cada ciclo ou iteração, novas funcionalidades são desenvolvidas, de acordo com uma priorização dada pelo cliente e definida através de critérios de maior valor para o negócio. Quando ocorrerem dependências técnicas, estas devem ser sanadas através de técnicas como *scaffolding* que produzem artifícios que simulam de forma simplificada a funcionalidade em falta para que as funcionalidades dependentes possam ser desenvolvidas [Miller, 2001].

De acordo com [Beck, 2000], o planejamento de uma iteração é feito detalhando-se uma *user story* em suas tarefas, requisitando programadores para trabalharem nelas e solicitando à estes, que estimem tempos. Enquanto o planejamento dos *releases* é sincronizado com o negócio do cliente, o planejamento das iteração é uma tarefa muito mais voltada aos programadores envolvidos.

Story	Time Estimate (Ideal Weeks)	Assigned Iteration #	Assigned Release #
Find lowest fare.	3	2	1
Show available flights.	2	1	1
Sort available flights by convenience.	4		2
Purchase ticket.	2	1	1
Do customer profile.	4		
Do simple customer profile.	2	1	1
Do full customer profile.	3		
Review itineraries.	1	2	1
Cancel itinerary.	2		2

Tabela 1 -Exemplo de planejamento de releases [Beck, 2000]

De acordo com [Beck, 2000], este é o fluxo de atividades natural de um projeto que utiliza a metodologia *Extreme Programming*:

4.3.1 Exploração

Esta fase dá à todos uma noção do que o sistema deve fazer. Um dos seus objetivos é tornar o cliente confiante de que há suficientes *stories* para um primeiro *release*, bem como ter a certeza de que os desenvolvedores não podem estimar melhor sem ter o sistema implementado. Nesta fase, os desenvolvedores também experimentam algumas das ferramentas e tecnologias que podem ser usadas durante a execução do projeto, bem como realizar testes de carga na rede e no hardware. É dividida em:

- **Escrever a story**: o cliente escreve as *stories* nos cartões, descrevendo o que o sistema deve fazer.
- **Estimar a story**: o desenvolvedor estima quanto tempo a *storie* precisa para ser implementada.
- **Dividir a story**: se o desenvolvedor não puder estimar, o cliente pode dividir a *storie* em outras partes menores.

4.3.2 Entrega

O cliente escolhe o escopo e as datas para o desenvolvimento. É dividida em:

- **Priorizar por valor**: O cliente categoriza as *stories* em três segmentos. (1) Fundamentais para o funcionamento do sistema, (2) Menos essenciais, mas com grande valor de negócio, (3) desejáveis.
- **Priorizar por risco**: O desenvolvedor categoriza as *stories* em três segmentos. (1) Podem ser estimadas com precisão, (2) Podem ser razoavelmente estimadas e (3) Não podem ser estimadas.
- **Definir a velocidade**: O desenvolvedor informa o quanto pode desenvolver em dias ideiais.
- **Escolher Escopo**: O cliente escolhe um conjunto de cartões para o *release,* definindo datas.

4.3.3 Controle

Guiar o desenvolvimento e atualizar o planejamento, baseado no aprendizado de cada iteração. É dividida em:

- **Iteração**: No início de cada iteração, o cliente seleciona as *stories* de maior valor para implemenar.
- **Recuperação**: Caso o desenvolvedor tenha superestimado a velocidade, pode questionar o cliente sobre as *stories* de maior valor para manter o *release* de acordo com a nova velocidade.
- **Nova story**: Caso o cliente perceba a necessidade de novas *stories* durante o desenvolvimento, ele pode escrever a *storie*, o desenvolvedor estima e o cliente pode trocar outra *storie* já submetida pela nova.
- **Re-estimativa**: Caso o desenvolvedor perceba que o planejamento não reflete mais a realidade, pode re-estimar a *storie* e definir novamente a velocidade.

4.4 Métricas

Temos uma tendência natural de medir a eficiência dos processos nas disciplinas da ciência. Na engenharia de software não é muito diferente. De acordo com [Fowler, 2004], apesar dos esforços, ainda somos incapazes de medir variáveis como a produtividade em software. Para ele, metodologias ágeis, que por sua vez adotam o foco na pessoa, e não no processo, devem ter o seu gerenciamento baseado na delegação e responsabilidades, dada o alto grau de conhecimento dos profissionais envolvidos e na dificuldade de medir as saídas do processo. O gerenciamento baseado em métricas econtra seu campo de atuação em processos envolvendo baixo grau de conhecimento e facilidade na métrica das saídas do processo. Para [Fowler, 2004], a utilização do gerenciamento baseado em métricas pode levar à distorções geradas pela equipe, para produzir melhores indicadores, mesmo que a efetividade do trabalho produzido seja reduzida.

Em *Extreme Programming*, a unidade de medida mais utilizada, chama-se "esforço em dias" é baseada na contagem de dias úteis em determinada iteração. Para se chegar ao "esforço em dias" é necessário multiplicar os dias úteis de determinada iteração (20, caso utilizemos 4 semanas) pelo número de pessoas da equipe. [Beck, 2000]

Apesar de ser extremamente simples de gerenciar este tipo de medida, as pessoas geralmente não produzem 100% do tempo disponível do calendário. Durante o desenvolvimento, ocorrem diversos eventos inesperados, além da distração, que fazem a produtividade ser geralmente menor. Este fato corroborou para a adoção de uma outra medida em XP, a do "tempo ideal". O "tempo ideal" é a medida em dias do tempo que leva a conclusão de uma determinada tarefa sem interrupções, em um ambiente sem intempéries. [Beck, 2000]

A fim de medir e estimar, utiliza-se o "tempo ideal" justamente por esta abstração de aspectos externos, embora ele raramente seja equivalente ao "esforço em dias". O importante é utilizar a mesma unidade de medida para as *stories* atuais e as passadas, a fim de utilizar o que já foi concluído para estimar as novas tarefas.

4.4.1 Velocity

A proporção entre o tempo estimado de desenvolvimento (em dias ideais) e o tempo em dias úteis decorridos no calendário para conclusão de tal tarefa é uma das medidas básicas utilizadas no planejamento em XP e chama-se "*Velocity*" ou Velocidade. Caso esta proporção aumente significa que são consumidos menos dias úteis para uma determinada tarefa estimada em dias ideais e pode indicar uma equipe trabalhando

melhor como também pode significar que os requisitos não estão sendo atingidos de maneira satisfatória. [Beck, 2000]

4.4.2 Member Load

"*Member Load*" ou carga do colaborador é uma forma de medir o quanto em dias ideias de trabalho, determinado programador aceitou. Esta medida é importante para garantir que programadores não assumam uma carga excessiva de trabalho. Medir sua velocidade ao longo das iterações é importante para garantir que ele não aceite tarefas que correspondem à uma quantidade maior de dias ideais do que a aceita na iteração anterior.

De acordo com [Beck, 2000], a velocidade não é uma medida de produtividade e deve ser utilizada para fins de previsibilidade. Para ele, determinado programador pode ter um indicador menor em razão de uma visão muito otimista no momento de realizar as estimativas ou devido à tempo investido ajudando aos outros.

4.4.3 Story Progress

Em diversos momentos durante a execução de um projeto, precisamos ter uma "fotografia" de sua atual situação, percebendo como está o desenvolvimento de cada membro da equipe e de cada *story*. Quem faz este papel, geralmante é o "*tracker*", visitando cada programador e questionando sobre:

- Quantos dias ideais já foram trabalhados em determinada *story*;
- Quandos dias restam até a conclusão da mesma.

Veja abaixo, um exemplo de progresso das *stories*, neste caso, temos cada *story*, quem está trabalhando nela, quantos dias ideais já foram concluídos e quantos são necessários para concluir a tarefa. Tal medida é importante para comparar a quantidade de dias que o programador necessita para concluir a tarefa e a quantidade de dias úteis disponíveis para o fim da iteração.

Task	Who	Done	To Do
UI cleanup	KA	2	0
KA Total (out till end)		2	0
Alternative fare finder object	KB	2	0
Update planet ports to find alternatives	KB	1	1
Find candidate fares for alternative ports	KB	0	1
Network performance improvement	KB	2	2
Investigate using IPv84	KB	0	1
KB Total		5	5
Special offers—major space lines	MF	0	2
Find candidate fares by date range	MF	0	1
Hotel booking interfacer	MF	1	0
Interface to IHAB	MF	3	2

Tabela 2 -Exemplo de Progresso da Iteração. [Beck, 2000]

4.4.4 Bug Density

Para [Pohren, 2004], *Bug Density* é a "quantidade de erros encontrados pelo usuário para cada classe multiplicado pelo tempo que cada um levou para ser corrigido".

5 Gerência de Requisitos

5.1 Introdução

Para [Leite, 2001], muitas das razões que levam à produção de softwares que não satisfazem as necessidades dos clientes são derivadas da falta de atenção para com a tarefa de definir e acompanhar a evolução dos requisitos de software durante a execução de um projeto. Aspectos de qualidade são tratados no próprio processo de definição do software, através de processos como o da gerência de requisitos.

Durante muito tempo, o centro das atenções da engenharia de software não foi o processo. Quando a produção era focada em fases sequenciais com produtos bem definitos, a qualidade deveria estar no produto e nas suas formas de representação (linguagens), e não no processo de concepção do mesmo [Leite, 2001].

Atualmente, sabe-se que o processo de produção é fundamental para que tenhamos produtos de qualidade, neste sentido, procura-se cada vez mais a obtenção e transformação de dados sobre o processo produtivo em conhecimento [Leite, 2001].

Através da passagem a seguir, vemos a importância da escolha dos métodos de produção, uma vez que estes terão relação direta para com a qualidade resultante. Também, vemos pontos em comum entre este trecho em destaque com metodologias ágeis, em especial *Extreme Programming*, onde temos como central a participação do ser humano no processo, sendo a qualidade da execução do projeto extremamente ligada com a qualidade dos membros da equipe.

> "É importante ressaltar que a escolha de quais métodos serão utilizados no processo de produção é uma escolha gerencial ligada a qualidade, porque a qualidade resultante será função dos métodos de produção escolhidos e seguidos. É também importante não esquecer que a produção de software é um processo que envolve, como parte fundamental, seres humanos. As tecnologias de produção de software, nas quais estão incluídas as tecnologias de gerência, tem por objetivo automatizar ao máximo a produção de software, mas ainda são fundamentalmente dependentes da qualidade das equipes" [Leite, 2001].

5.2 Gerência Tradicional

Para [Nuseibeh, 2000], a medida básica de sucesso para um software é o quão ele atende os propósitos para o qual foi projetado. Neste contexto, o papel da engenharia de requisitos é o de descobrir este propósito através de identificação e documentação das necessidades dos clientes. A engenharia de requisitos conduz à um processo que leva desde o reconhecimento de um problema a ser resolvido até sua especificação detalhada.

Basicamente suas técnicas visam assegurar um conjunto consistente e significativo de requisitos antes da construção do sistema, para se evitar mudanças posteriores. Estes

idéias baseiam-se no tradicional pressuposto de que quanto mais tarde os erros forem identificados, mais custosos serão [Paetsch, 2003].

Desde os anos 60, muitas metodologias de desenvolvimento de software foram criadas, utilizadas e aprimoradas pelos seus usuários, atingindo um nível de maturidade. Estas metodologias, voltadas muitas vezes para diferentes aspectos do desenvolvimento, são conhecidas na engenharia de software como "metodologias tradicionais". Dentre estas, a que mais se destaca é o modelo em cascata (*waterfall*), que tem sido utilizado com sucesso em projetos grandes e complexos. O modelo em cascata difide o processo de desenvolvimento em estágios lineares. Apesar do sucesso de metodologias tradicionais, como o modelo em cascata, em projetos complexos, tais metodologias são inflexíveis face à mudança de requisitos e possuem um processo burocrático independente do tamanho do projeto. As metodologias ágeis foram criadas para endereçar estes problemas inerentes às metodologias tradicionais [Huo, 2004].

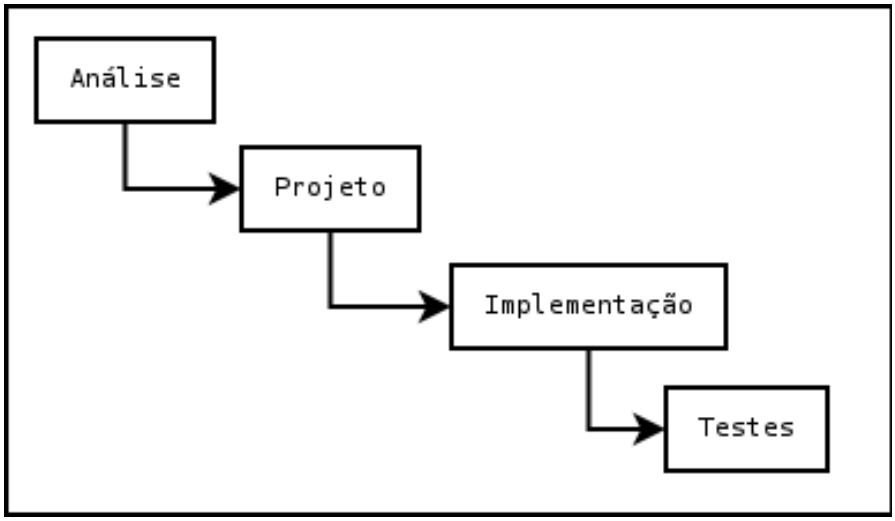


Figura 7 -Modelo em cascata

5.2.1 Elicitação

Para [Horrian, 2003], o objetivo da elicitação é descobrir os requisitos de um sistema no domínio da aplicação e dos negócios, bem como o escopo do mesmo, suas necessidades e restrições através de seus usuários. Para [Nuseibeh, 2000], a elicitação de requisitos também consiste em identificar os usuários do sistema, bem como os desenvolvedores envolvidos no processo, as tarefas que seus usuários desempenham e como eles gostariam de desempenhar estas mesmas tarefas. De acordo com [Paetsch, 2003], algumas técnicas utilizadas para tal, são:

- **Entrevistas**: É um método que pode ser aplicado informalmente ou através de questionários com perguntas pré-definidas. Sua utilização traz à tona fatos e opiniões dos usuários do sistema. Por um lado as entrevistas resultam em um conjunto rico em informações, por outro, demandam grandes esforços de análise, tendo em vista que podem trazer visões conflitantes.
- **Casos de Uso/Cenários**: Casos de Uso descrevem as interações entre o sistema e seus atores, trazendo à tona suas relações e os diferentes papéis desempenhados pelos usuários no sistema. Casos de Uso são usados nos primeiros estágios de análise para demonstrar os requisitos funcionais do

sistema. Cenários provêm descrição de um determinado conjunto de atividades entre o sistema e o usuário, simulando o fluxo de interações entre os mesmos após um estado inicial.

- **Observação**: A observação consiste na observação do usuário do sistema em suas atividades cotidianas, visando descrever suas tarefas, através de meios diretos, como a anotação, ou indiretos como a gravação.
- **Brainstorming**: Brainstorming é uma técnica de criatividade para geração de idéias com vistas a resolução de problemas. As idéias são geradas de forma rápida, coletadas e então discutidas e avaliadas pelo grande grupo.
- **Prototipação**: Um protótipo é uma versão inicial de um sistema com vistas à validação de requisitos ainda em estágios iniciais de desenvolvimento. O protótipo de um sistema pode ser descartado ou evoluir para uma versão final do sistema.

5.2.2 Análise

Para [Horrian, 2003], a análise de requisitos verifica a consistência, completeza e aplicabilidade dos mesmos, observando se são contraditórios, apresentam informações necessárias para sua implementação e estão dentro das restrições impostas pelo cliente. Esta fase envolve resolução de conflitos através de técnicas como negociação de priorização e levantamento de pré-requisitos técnicos.

- **Priorização**: O principal objetivo da priorização é a entrega dos requisitos de maior valor para o cliente antes, observando restrições de agenda e de investimentos. A priorização é realizada em conjunto entre o cliente e o desenvolvedor, onde o cliente irá escolher os de maior benefício para o seu negócio e o desenvolvedor irá ressaltar pré-requisitos técnicos, quando estes existirem.
- **Modelagem**: Modelos são uma importante ferramenta para destacar diferentes visões do sistema. Os modelos refletem praticamente todas fases do desenvolvimento de um sistema, passando pela análise e projeto e são utilizados com sucesso para maior entendimento do sistema proposto, bem como a validação do mesmo [Nuseibeh, 2000].

5.2.3 Documentação

A documentação de requisitos é utilizada para comunicar os requisitos do sistema entre cliente e desenvolvedor, através da descrição do domínio da aplicação e do sistema a ser desenvolvido [Horrian, 2003].

Para [Nuseibeh, 2000], a forma pela qual os requisitos de um sistema são documentados é de extrema importância, para garantir que os mesmos possam ser lidos, analizados e validados.

5.2.4 Validação

Para [Horrian, 2003], o processo de validação objetiva garantir que os requisitos reflitam as necessidades do cliente em relação ao sistema (completeza), através da inspeção por padrões organizacionais, conhecimento sobre a organização e do sistema a ser desenvolvido, garantindo que os requisitos estejam livres de inconsistências. Para

[Nuseibeh, 2000], se torna necessário não apenas validar os requisitos, mas também resolver possíveis conflitos entre membros com idéias divergentes.

5.2.5 Gerenciamento

Durante o ciclo de vida de um projeto, é necessário gerenciar as mudanças de requisitos. Neste sentido, dependências e relações entre os requisitos são controlados via matriz de rastreabilidade, que permite ao gerente conhecer quais outros requisitos serão afetados por uma mudança. Para tal, utiliza-se versionamento de requisitos, para que se mantenha o registro de quem solicitou a alteração do requisito, quais foram as alterações e qual foi o impacto decorrente [Horrian, 2003].

Para [Nuseibeh, 2000], o sucesso de um software depende do quanto ele irá se adequar as mudanças do ambiente. Neste sentido, gerenciar as mudanças de requisitos, tais como a adição ou remoção de requisitos é fundamental, uma vez que focar a gerência da mudança apenas no código-fonte, através de controles de versões, por exemplo, leva à dificuldades na manutenção.

5.3 Gerência em Extreme Programming

A forma através da qual as metodologias ágeis lidam com a volatilidade e instabilidade dos requisitos é através da adoção de uma série de técnicas como: planejamento simples, iterações curtas, *releases* recentes e frequente *feedback*. Tais técnicas permitem a entrega de versões de um produto em intervalos de tempo menores do que nas metodologias tradicionais [Huo, 2004].

Para [Paetsch, 2003], as metodologias ágeis tem se tornado populares nos últimos anos, introduzindo uma série de métodos com o objetivo de entregar software rapidamente e de assegurar que mudanças nas necessidades dos clientes sejam atendidas. Para tal, são adotados uma série de princípios, como já vistos até aqui, como entregas constantes, cliente presente e aceitação da mudança de requisitos. Esta abordagem ágil é constantemente vista como incompatível com a tradicional engenharia de requisitos, onde se identifica, analisa, documenta e validam os requisitos de um sistema, em um processo mais “pesado” e baseado na documentação escrita.

Para [Paetsch, 2003], metodologias ágeis são por natureza menos orientados à documentação e mais centrados no código da aplicação. *Extreme Programming*, por exemplo, não aborda explicitamente técnicias de requisitos em detalhes e sua documentação é tida como um custo a ser evitado.

[Horrian, 2003] coloca que a prática de documentação de requisitos não é encontradas em equipes XP, onde o conhecimento depende muito mais do cliente e do desenvolvedor e práticas como a propriedade coletiva do código levam à um conhecimento do sistema por parte de todos os integrantes da equipe.

A fase que antecede a escrita dos *story cards* em conjunto entre o cliente e o desenvolvimento consistem em um processo que pode ser visto como um *brainstorming*, onde os desenvolvedores requisitam maiores detalhes sobre cada *story* de cliente, com o objetivo de ter subsídios para realizar as estimativas de tempo, que por sua vez, serão utilizadas posteriormente pelo cliente para priorização de tarefas para cada iteração [Paetsch, 2003].

Para [Horrian, 2003], a etapa de planejamento existente em XP, chamada de “Planning Game”, cumpre o papel para a elicitação de requisitos, através da utilização de técnicas como os *story cards*, entrevista e priorização. Para ele, as seções de brainstorming desenvolvidas conjuntamente com o cliente provêm uma *baseline* de requisitos. Também o uso de roteiros não é necessário para simular, como no processo tradicional de elicitação de requisitos, uma vez que o cliente presente é um membro constante na equipe, para sanar dúvidas. Da mesma forma, protótipos perdem o sentido, uma vez que o código é entregue constantemente, permitindo que o cliente tenha uma visão real do sistema a cada *release*, aumentando o valor do sistema.

Em relação à validação de requisitos, metodologias ágeis usam testes de aceitação, reuniões de revisão, mas principalmente a entrega da aplicação ao cliente, através da qual, o cliente pode utilizar o software e decidir por si mesmo, se o conjunto de requisitos foram implementados de acordo com a necessidade do negócio [Paetsh, 2003]. Práticas como o desenvolvimento guiado por testes garantem que todos os requisitos sejam validados e entendidos antes de sua implementação [Horrian, 2003].

[Paetsch, 2003] compara ainda a escrita de testes em conjunto com *releases* frequentes presentes em XP, com a revisão de requisitos em conjunto com prototipação evolucionária da engenharia tradicional. Entretanto, diferentemente da prototipação tradicional, XP exige um código claro e suficientemente testado.

5.3.1 Extreme Requirements

[Leite, 2001] observa que *Extreme Programming* não oferece definições claras de como lidar com os requisitos de software além de sugerir a utilização de *story cards*. Portanto, sugere a criação de um conjunto de processos complementares às práticas de XP, chamados de Extreme Requirements (XR). De acordo com o autor, estes são alguns problemas observados na metodologia XP:

- A certeza de que as necessidades de negócio possam ser representadas por apenas um cliente;
- A falta de consideração para com os requisitos não-funcionais da perspectiva dos negócios;
- A falta de elos entre *stories*, tarefas e o código;
- A falta de um processo para produzir testes funcionais;
- A falta de um processo para produzir *stories* e testes;

Para [Leite, 2001], quando tratamos de ambiente de negócios, é muito incomum encontrar alguém que possa representar diferentes interesses e percepções. Adotar a prática de “cliente presente” é uma forma simples de delegar a responsabilidade de resolver inconsistências internas à este membro da equipe, entretanto é um risco alto ao projeto.

[Leite, 2001] propõe a utilização de roteiros, uma técnica que visa facilitar o entendimento e a comunicação entre os membros de uma equipe, devido à sua facilidade de simular o pensamento. A utilização de roteiros demanda a aplicação de uma linguagem descritiva de fácil aprendizado. Note que termos em maiúsculos indicam a existência de um sub-roteiro, uma forma simples de se manter o elo entre os artefatos.

TITLE: **Organize the Meeting**
GOAL: Assure an efficient development of the meeting.
CONTEXT: The meeting has been previously scheduled.
RESOURCES: equipment, physical space.
ACTORS: requester, secretary.

EPISODES:
The requester instructs the secretary about the meeting call.
CALL TO THE MEETING.
NOTIFY ASSISTANCE.
NOTIFY ABSENCE.
[ASK FOR EQUIPMENT.]
IF the convoking date was made with anticipation **THEN** REMIND THE MEETING.
[The secretary assures that the equipment is available for the meeting date .]

Figura 8 -Roteiro [Leite, 2001]

Para [Leite, 2001], uma vez escrito o roteiro pelo "cliente presente", ele pode ser verificado por outros representantes do cliente, para garantir que o roteiro seja validado por clientes com diferentes pontos de vista. Este processo é chamado de processo de inspeção de roteiros.

O adiamento da preocupação com os requisitos não-funcionais pode levar à um aumento do custo do projeto, assim como problemas de qualidade. A forma de se lidar com isto é trazendo os requisitos não-funcionais às etapas iniciais do projeto, anexando-os à requisitos funcionais, de preferência, juntamente à descrição das etapas, utilizando para tal uma linguagem especial. [Leite, 2001] sugere a utilização do termo "*constraint*" (restrição) para tal.

5.4 Gerência Tradicional *versus* XP

O modelo em cascata apresenta cinco estágios diferentes. Em princípio, cada estágio deve estar finalizado antes de se prosseguir para o seguinte.:

1) Análise e definição de requisitos;
2) Projeto do sistema;
3) Implementação e testes de unidade;
4) Integração e testes de sistema;
5) Operação e manutenção;

As metodologias ágeis, de outra maneira, passam por todos os seus estágios constantemente durante execução do projeto. Apesar de não apresentar estágios tão claramente separados quando no modelo em cascata, a sequência das fases de reconhecimento dos requisitos, planejamento, implementação e integração são as mesmas em ambos os modelos [Huo, 2004].

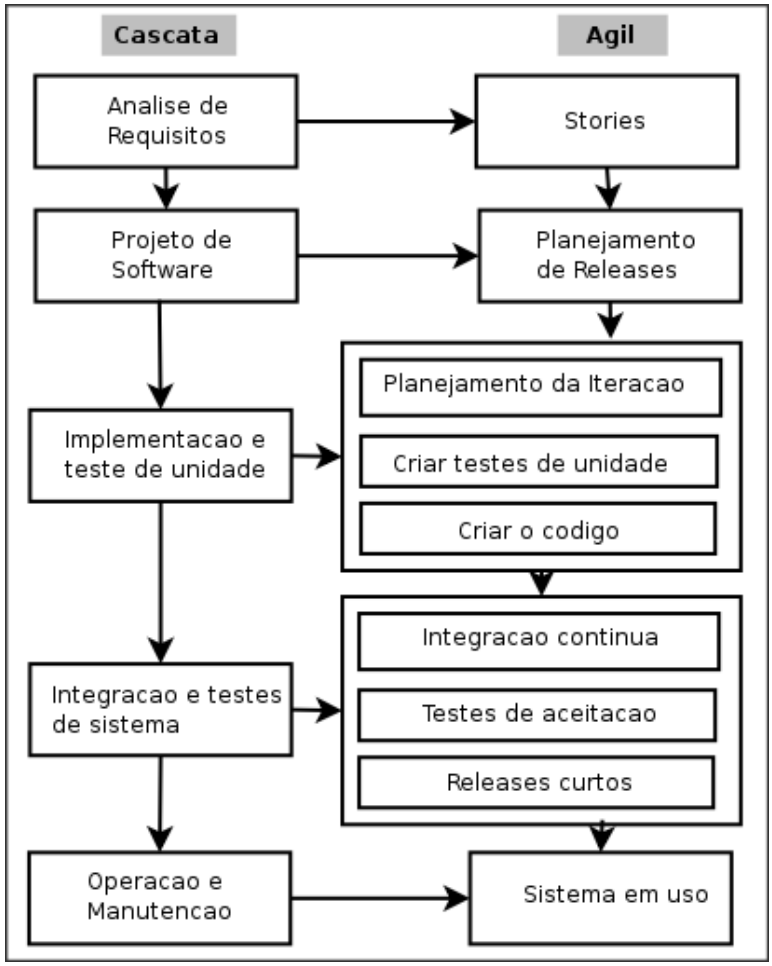


Figura 9 -Modelo em cascata *versus* modelo ágil [Huo, 2004]

Na tabela abaixo, as fases de um processo em cascata. Cada fase não pode iniciar até que a fase anterior esteja completa, uma vez que ela utiliza os resultados da fase anterior como seu *input*. Estes mesmos resultados são verificados por um processo de garantia de qualidade correspondente [Huo, 2004].

Modelo em Cascata	Técnicas de Qualidade
Definição de requisitos	1. Revisão de requisitos 2. Prototipação 3. Validação do modelo
Projeto do sistema	1. Questionários 2. Métricas 3. Validação de Cenários 4. Checagem do modelo
Implementação e teste de unidade	1. Revisão de código
Integração e teste de sistema	1. Simulação 2. Testes de integração 3. Testes de aceitação
Operação e manutenção	1. Controle Requisição de mundança

Tabela 3 -Técnicas de qualidade do modelo em cascata [Huo, 2004]

As metodologias ágeis, diferentemente, não possuem fases distintas. Apesar da sequência do desenvolvimento ser basicamente a mesma, um ciclo corresponde à uma versão (*release*), que se repete várias vezes durante o projeto, o que não existe no modelo em cascata. Nas metodologias ágeis, algumas atividades de desenvolvimento possuem também a funcionalidade de garantia de qualidade, movendo algumas responsabilidades para os desenvolvedores, o que é mais responsável se comparado a ter uma equipe em separado para tal. A troca de resultado entre as fases é agilizada pela rápida comunicação, garantindo a velocidade no processo. Na tabela abaixo, cada sequência do processo ágil, possui um processo de garantia de qualidade [Huo, 2004].

Metodologias Ágeis	Técnicas de Qualidade
Definição de requisitos Projeto do sistema	1. Metáforas 2. Cliente presente
Implementação e teste de unidade	1. Refactoring 2. Pair Programming 3. Reuniões matinais 4. CRC's
Integração e teste de sistema	1. Integração contínua 2. Testes de integração 3. Testes de aceitação

Tabela 4 -Técnicas de qualidade das metodologias ágeis [Huo, 2004]

6 Rastreabilidade de Requisitos

6.1 Introdução

Para acompanhar o ritmo do ambientes de negócios, projetos de software devem lidar com a constante mudança de objetivos e necessidades do cliente. Para tal, é necessário ouvir o cliente durante todas as fases do desenvolvimento, tornando as decisões de *design*, bem como a especificação de requisitos rastreáveis. A rastreabilidade é fator chave para proporcionar o rápido acesso à informação, diminuindo o impacto da mudança [Lee, 2003].

A rastreabilidade de requisitos consiste em ligações entre as informações produzidas no desenvolvimento de software. Para [Pinheiro, 2004], tais ligações são essenciais no desenvolvimento de sistemas, uma vez que é grande o volume de informações produzidas e estas devem estar relacionadas, ainda mais em ambientes distribuídos com com várias equipes. Neste contexto, a rastreabilidade é inevitável pois precisamos dela para procurar informações que nos ajudem a tomar decisões e a descobrir fatos como investigar as causas de um erro em testes de aceitação, estudar o impacto da mudança de um requisito, dentre outros.

De acordo com [Pinheiro, 2004], a rastreabilidade é uma tarefa auxiliar que serve tanto para propósitos técnicos quanto gerenciais. Rastrear significa procurar, seguir um caminho, manter o histórico, como em um mapa, onde podemos ir de coordenada em coordenada, indo e voltando, desde suas origens até o seu desenvolvimento. A rastreabilidade deve ser produzida de maneira natural, dentro do processo e deve permitir ao interessado visualizar somente os rastros de seu interesse.

Os requisitos são vinculados aos documentos que os geraram, pois tais documentos podem conter a descrição exata dos motivos que levaram à criação de tal requisito, bem como podem conter informações valiosas acerca deste requisito. Nestes casos, ambos devem ser relacionados para que um leve ao outro, para que possamos seguir os rastros deixados pelos requisitos em outros documentos (navegar para frente), bem como os rastros deixados por outros documentos nos requisitos (navegar para trás). O mesmo acontece entre os requisitos e as decisões de *design* e os testes de aceitação, os quais só existem devido aos requisitos que os geraram [Pinheiro, 2004].

[Sayao, 2006] destaca como sendo vantagens advinda do uso da rastreabilidade:

- Verificação de alocação entre requisitos e sua implementação;
- Identificação das origens de requisitos conflitantes;
- Verificação de requisitos para os quais não foram previstos testes;
- Validação do sistema, verificando se o mesmo atende ao conjunto de requisitos proposto;
- Estimativas de custos de prazos quando da inserção de uma nova funcionalidade;
- Identificação de riscos que possam impactar os requisitos;

- Visualização de relacionamentos entre hardware e software que auxiliam em mudanças de ambiente operacional;
- Identificação de ligações entre código e documentos de análise que possam proporcionar futuro reaproveitamento de código;

6.2 Tipos de Rastreabilidade

Rastreabilidade para frente (*forward traceability*) permite que se chegue até a implementação e componentes de *design*, a partir dos requisitos. É útil quando mudam os requisitos e desejamos visualizar qual será o impacto da mudança no *design*, nos testes para que possamos antever possíveis mudanças nestas rotinas [Pinheiro, 2004].

Rastreabilidade para trás (*backward traceability*), permite que se chegue até as origens do requisito (pessoas, processos, documentos), a partir dos requisitos. É útil quando os requisitos mudam e não entende-se os motivos que levaram à isto. Desta forma, podemos investigar os documentos, decisões que levaram à especificação deste requisito [Pinheiro, 2004].

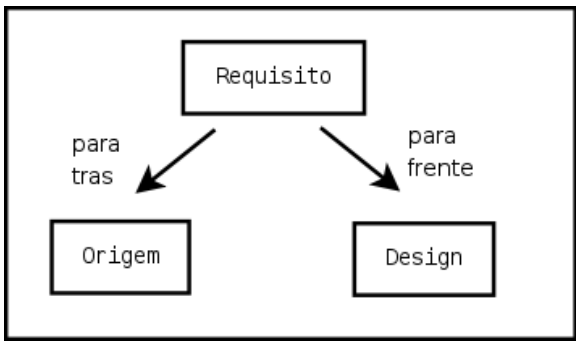


Figura 10 -Tipos de rastreabilidade [Pinheiro, 2004]

O processo de especificação de requisitos é cercado de muito trabalho e com grandes detalhes. Muitas vezes os requisitos são refinados, alguns novos são criados e outros deixam de existir. Neste contexto, é importante manter os vínculos entre os requisitos e os artefatos que os geraram. Para [Pinheiro, 2004], a rastreabilidade "intra-requisitos" consiste nos relacionamentos entre os requisitos e a rastreabilidade "extra-requisitos" consiste nos relacionamentos entre os requisitos e outros artefatos.

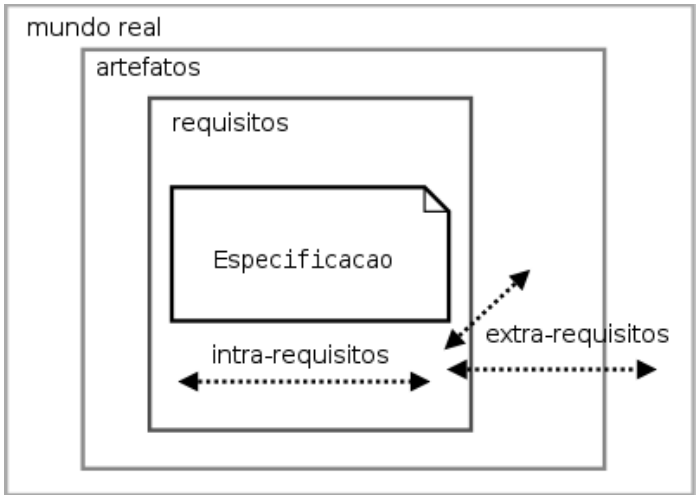


Figura 11 -Rastreabilidade entre requisitos [Pinheiro, 2004]

6.3 Rastreabilidade funcional e não-funcional

A rastreabilidade funcional é relacionada aos aspectos do desenvolvimento de software que podem ser descritos em termos de estados, testes, modelos, diagramas, artefatos e notações. A rastreabilidade é derivada das interligações entre estes modelos. Podemos destacar, os modelos de análise, que referenciam requisitos como entrevistas e documentos (memorandos, manuais), os modelos de *design* que referenciam classes, atributos, métodos e sequências, dentre outros e a rastreabilidade entre um requisito e seus testes funcionais. A rastreabilidade dentro de um mesmo modelo, entre os componentes de um mesmo modelo é chamada de *rastreabilidade vertical* e a entre diferentes modelos é hamada de *rastreabilidade horizontal* [Pinheiro, 2004].

A rastreabilidade não-funcional está relacionada à aspectos de qualidade e à conceitos intangíveis, tais como objetivos, metas, decisões e intenções. Geralmente são vinculados à requisitos não-funcionais. Dentre os aspectos não-funcionais que merecem destaque, estão a definição de metas, a interpretação e justificativa de artefatos, descrição do ambiente de desenvolvimento, bem como do ambiente de negócios, incluindo aspectos sociais. Para [Pinheiro, 2004], estes precisam ser traduzidos para uma forma funcional para que possam ser verificados, através de modelos organizacionais que reflitam aspectos como políticas, metas e papéis.

6.4 Aspectos da rastreabilidade

Para [Pinheiro, 2004], três aspectos devem ser cobertos pela rastreabilidade: a definição (o que será rastreável), a produção (registro dos objetos e seus relacionamentos) e a extração (processo de seleção e visualização do que foi rastreado).

Na definição devemos escolher quais tipos de caminhos devem existir entre os objetos a serem rastreados, qual o significado de suas ligações e sob quais condições eles serão rastreados. O significado das ligações é importante para evitar má interpretação, que pode causar problemas maiores do que a falta de rastreabilidade. A produção ocorre quando da existência de um evento. Tão cedo isto acontecer, ele deve ser registrado.

A extração consiste em meios de se obter a informação registrada. Para [Pinheiro, 2004], a extração pode ser:

- **Seletiva**: restringe a rastreabilidade à certos padrões de objetos e seus relacionamentos, contexto ou fase do desenvolvimento.
- **Interativa**: permite navegar entre os objetos relacionados, guiado pelos seus relacionamentos. Neste caso, pode-se utilizar marcações do tipo <traces-from> e <traces-to> na especificação dos requisitos.
- **Não-guiada**: permite ao usuário ir de um objeto ao outro, investigando seu conteúdo, quando se tem pouca informação sobre como prover rastreabilidade.

A aplicação proposta provê extração seletiva e interativa entre as *stories* e os documentos de análise, entre as decisões de *design* e as *stories* e entre os testes de aceitação as *stories*.

6.5 Classificações e Modelos

De acordo com o apresentado por [Ramesh. 2001], é proposto um modelo de rastreabilidade que possibilita a captura de informações sob as seguintes dimensões:

- **Fontes (*Sources*)**: A origem dos requisitos, documentos, leis, normas;
- **Interessados (*Stakeholders*)**: Pessoas envolvidas no processo;
- **Objetos (*Objects*)**: Objetos relacionados ao produto, como requisitos ou ao processo, como a arquitetura;

Pela proposta de [Toranzo, 1999], as informações a serem rastreadas são classificadas em quatro níveis que podem impactar o sistema sendo desenvolvido:

- **Ambiental/Organizacional**: Contém as informações do ambiente no qual a organização está inserida e as informações da organização (missão, valores, padrões);
- **Gerencial**: Contém informações associativas entre requisitos e tarefas, valiosas para a gerência do projeto (tarefas, recursos, riscos e checkpoints);
- **Desenvolvimento**: Contém informações sobre os artefatos provindos do processo de desenvolvimento (modelos, diagramas, requisitos, programas);

6.6 Rastreabilidade em XP

Para que a rastreabilidade seja criada com sucesso, deve iniciar nos primeiros estágios do projeto e o método para sua criação deve ser não-intrusivo à equipe de desenvolvimento, aproveitando o andamento do processo de elicitação de requisitos, como conversas entre os desenvolvedores e o cliente, criando ligações entre as decisões de *design* e os requisitos [Lee, 2003].

Projetos em *Extreme Programming* usam técnicas não-estruturadas para reunir os requisitos, conhecidas como *user stories*, que são geralmente escritas em cartões. Os cartões podem ser organizados por prioridade, risco, custo, dentre outros. A principal função dos cartões é guiar as discussões com o cliente sobre as funcionalidades do sistema. Entretanto, as conversas ocorridas como resultado das *stories* são raramente capturadas e ficam na memória coletiva, o que pode ser uma fraqueza da metodologia, se aplicada em projetos complexos [Lee, 2003].

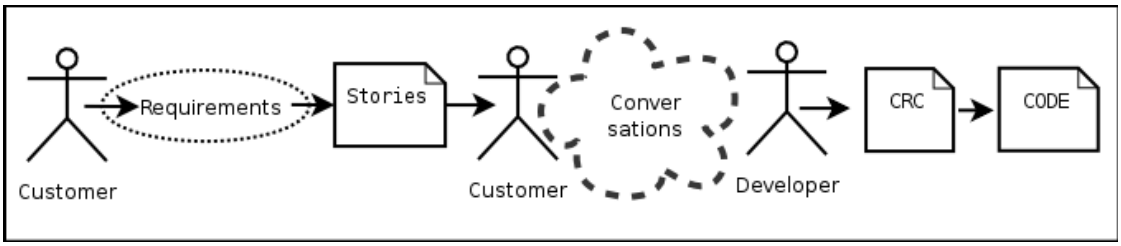


Figura 12 -Fluxo dos requisitos em um projeto XP [Lee, 2003].

A natureza ágil dos artefatos utilizados em XP fazem com que seja difícil aplicar a tradicional documentação estática e centralizada para suportar as necessidades de rastreabilidade. Para [Lee, 2003], a única forma de atingir tal rastreabilidade, é através de ligações entre as mudanças de requisitos e as decisões de *design*.

Atualmente há pouca informação acerca do que acontece com as *stories* após estas serem codificadas. Para [Breitman, 2002], os requisitos do sistema (neste caso, as *stories*) são importantes artefatos e devem estar disponíveis mesmo após o sistema desenvolvido e entregue para garantir a rastreabilidade. As mudanças no código também devem estar em sincronia com os requisitos que as geraram.

Para [Lee, 2003], soluções que venham prover gerenciamento das mudanças em projetos ágeis devem suportar práticas focadas em promover a construção coletiva do conhecimento, onde o conteúdo é construído colaborativamente através de estruturas como planos, *stories* e tarefas.

7 Trabalhos relacionados

7.1 Introdução

Neste capítulo, serão apresentadas ferramentas similares à desenvolvida no presente trabalho. Para tal, foi realizada uma seleção a partir de pesquisa na internet e também com base no trabalho desenvolvido por [Pohren, 2004]. Desta forma, foram considerados somente aqueles projetos cujas páginas web encontravam-se disponíveis durante a fase de seleção (02 à 08 de outubro de 2006) e também cujos projetos demonstravam certo grua de vitalidade (notícias atuais, versões recentes, etc).

7.2 Ferramentas

7.2.1 XPManager

O XPManager (http://www.naphta.com.br/xpmanager.html) é uma ferramenta criada durante uma monografia de graduação [Pohren, 2004] com o objetivo de suportar a gerência de projetos segundo a ótica da metodologia *Extreme Programming*. Desta forma, o XPManager se configura como sendo uma ferramenta completa para gerência de projetos, oferecendo funcionalidades como o controle do projeto através das *user stories* e também fornecendo estatísticas financeiras do projeto [Pohren, 2004].

De acordo com [Pohren, 2004], "o XPManager foi desenvolvido com a intenção de ser um *software* que auxilie o *coach* e o *tracker* das equipes XP no seu dia-a-dia. Para isto, foram levados em conta no seu desenvolvimento os valores e as características da metodologia, a fim de que o mesmo não fosse muito 'invasivo' na rotina da equipe de desenvolvimento."

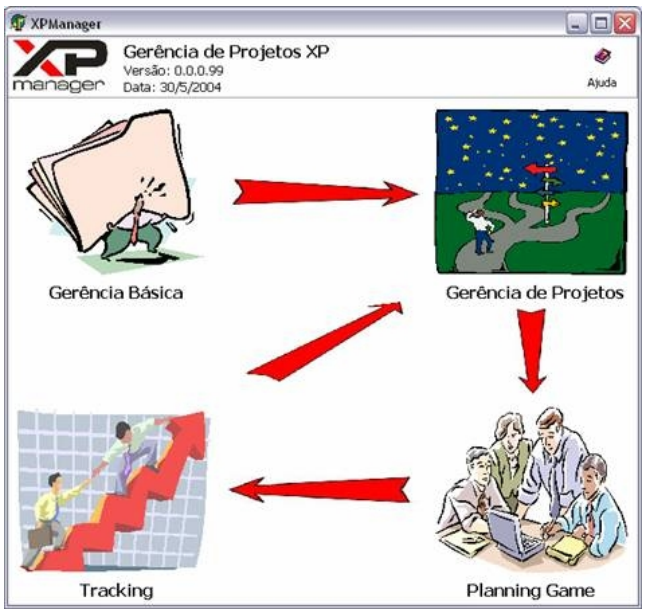


Figura 13 -Fluxo dos processos no XPManager

7.2.2 VersionOne

VersionOne (http://www.versionone.net) é uma ferramenta comercial que de acordo com seu site oficial é: "Completa e fácil de utilizar para o planejamento e gerenciamento de projetos ágeis". O VersionOne é uma ferramenta baseada na web e se constitui em um *framework* intuitivo para organizações introduzirem suas práticas ágeis suportando os mais diversos papéis: desenvolvedores, testadores, gerentes e clientes trabalhando colaborativamente.

O VersionOne possui a gerência integrada de requisitos, características, tarefas, bugs e testes. Também suporta todas as práticas chave do gerenciamento ágil, como: planejamento de features, planejamento dos *releases*, planejamento das iterações e rastreabilidade, gerenciamento de tarefas e testes, além de relatórios de *Member Load* e *Velocity*, dentre outros. Permite exportar algumas informações no formato XLS.

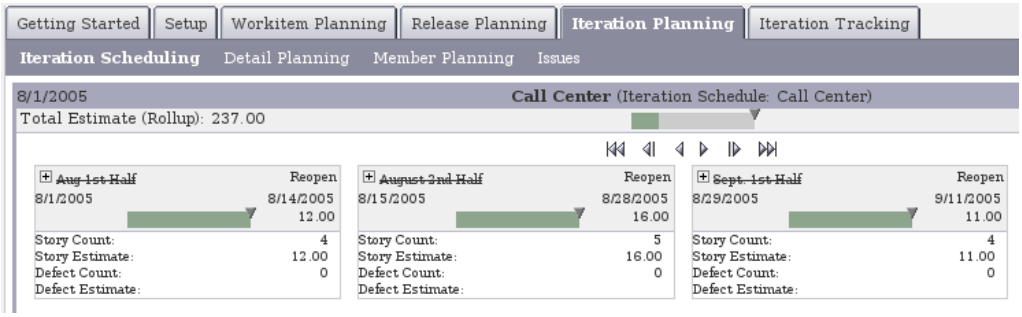


Figura 14 -*Iteration planning* no VersionOne

7.2.3 Scope Manager

O Select Scope Manager (http://www.selectbs.com/products/select-scope-manager.htm) objetiva auxiliar desenvolvedores a automatizar o desenvolvimento baseado em *Extreme Programming*, reduzindo o ciclo de desenvolvimento, tempo e custos através da aplicação de práticas simples de gerenciamento de escopo para o desenvolvimento de software. O Scope Manager automatiza o projeto XP proporcionando decisões rápidas através do gerenciamento do escopo de *stories*, tarefas, idéias e documentos de análise, permitindo assim, à equipe de desenvolvimento estimar e detalhar as tarefas requisitadas. Desta forma, as *stories* podem ser selecionadas e atribuídas à *releases*.

De acordo com a página web da ferramenta, cada *story* segue seu próprio processo de desenvolvimento. Durante o desenvolvimento, seu progresso é monitorado e o escopo é ajustado de acordo. Desta forma, listagens de tarefas, plano de projeto, controle de horas, relatórios de *velocity* e gráficos do andamento do projeto são gerados automaticamente.

O Scope também possui funcionalidades que permitem exportar os projetos desenvolvidos nele no formato XML e também importar usuários.

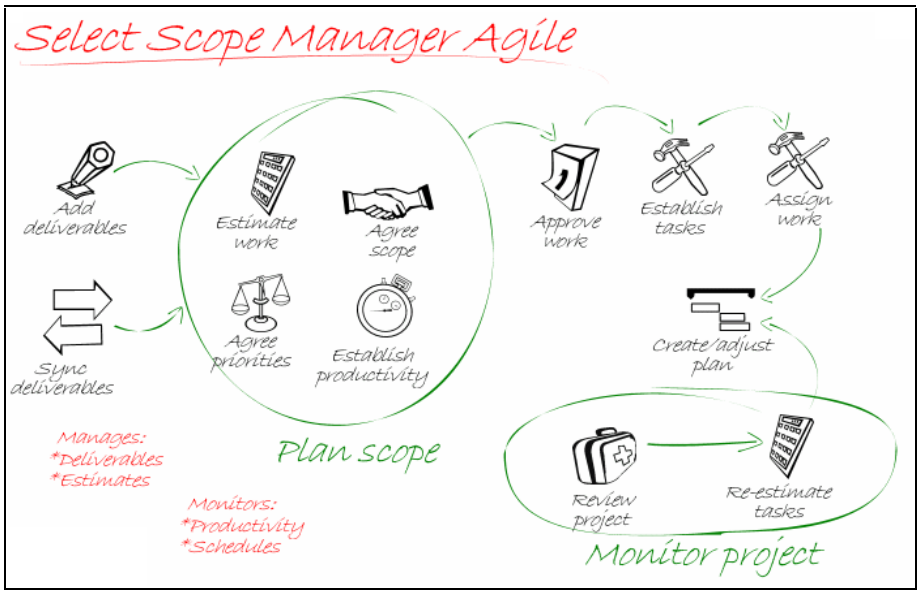


Figura 15 -Fluxo dos processos no Scope Manager

7.2.4 XPlanner

O Xplanner (www.xplanner.org) é um projeto em software livre, web, voltado ao planejamento e controle de equipes que utilizam *Extreme Programming*. Como diz sua página na web: "O Xplanner é um trabalho em andamento. Seus desenvolvedores esperam que a ferramenta evolua concomitantemente com o entendimento e maior compreensão da metodologia XP por parte da comunidade de software."

O Xplanner possui diversas funcionalidades, dentre elas: Permite o registro de projetos, iterações, *stories* e tarefas. Permite a inter-ligação entre tarefas através de hiperlinks. geração de planilhas de horas trabalhadas, relatório de *Velocity* e a possibilidade de anexar recursos (documentos) às *stories*. Permite exportar os dados do projeto e das iterações em formato: XML, MPX (MS Project), PDF, e iCal. Suporta ferramentas de colaboração como o Twiki e interfaces SOAP para integração

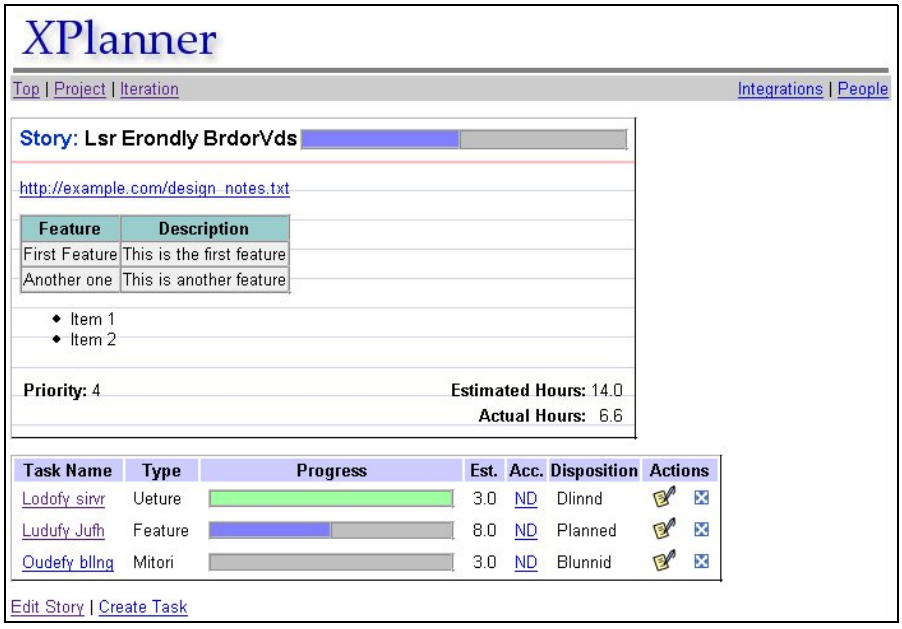


Figura 16 -Progresso das *stories* no Xplanner

7.3 Critérios de Classificação

Para classificar as ferramentas estudadas e traçar um paralelo com o trabalho sendo desenvolvido pelo presente trabalho, elaboramos alguns critérios de avaliação. Alguns destes são técnicos (web, multi-plataforma), os demais são funcionalidades consideradas importantes pela literatura. A seguir, veremos quais foram estes critérios. Ao final, apresentamos uma matriz de correlação apresentando tais funcionalidades.

- **Web:** O sistema contemplará este item se puder ser rodado via web, através de um browser (Internet Explorer, Mozilla, Opera, dentre outros).

- **Multi Plataforma:** O sistema contemplará este item se puder rodar nos sistemas operacionais mais utilizados do mercado (Windows, Linux e MacOS).

- **Cadastros de Colaboradores:** Indica se o sistema apresenta cadastros contemplando informações detalhadas a respeito de clientes e desenvolvedores, permitindo inclusões, alterações e listagens;

- **Cadastro da Story:** O sistema contemplará este item se contemplar o cadastro detalhado da *story*, permitindo editar informações como datas, prioridades, resumo, prioridades, testes, dentre outros.

- **Anexar Recursos:** O sistema contemplará este item se permite relacionar documentação suplementar ao projeto, como modelos, documentos e gráficos.

- **Gerência de Releases:** O sistema contemplará este item se permite cadastrar as diferentes *releases* de um projeto, com informações como datas de início, término e projeto a qual o *release* pertence.

- **Gerência de Iterações:** O sistema contemplará este item se permite cadastrar iterações em um projeto, bem como seus detalhes: datas, *release* a qual pertence, velocidade, número de dias, dentre outros.

- **Visão geral do Processo:** O sistema contemplará este item se apresentar uma visão geral do fluxo de atividades dentro do sistema, em forma de fluxograma.

- **Planning Game:** O sistema contemplará este item se permitir alocar *stories* dentro de iterações de forma dinâmica, disponibilizando ao usuário informações sobre a alocação de tempo atual e tempo disponível.

- **Rastreabilidade:** O sistema contemplará este item se oferecer ao usuário algum tipo de rastreabilidade (para frente, para trás, intra-requisitos, extra-requisitos) seja ela seletiva, interativa ou não-guiada.

- **Relatórios Estatísticos:** O sistema contemplará este item se apresentar algum tipo de gráfico ou relatório estatístico que auxilie o gerente na tomada de decisões. Gráficos que demonstrem a evolução do projeto ou mesmo o status de *stories* e tarefas estão dentre os relatórios estatísticos mais comuns.

- **Perfis Diferenciados:** O sistema contemplará este item se oferecer diferentes níveis de permissão para cada tipo de perfil de usuário (gerente, desenvolvedor, cliente). Neste quesito são avaliadas questões como: quem pode cadastrar, estimar, gerenciar, cadastrar testes, dentre outros.

- **Relatório Velocity:** O sistema contemplará este item se apresentar o relatório de *Velocity*, que demonstra a proporção entre o tempo estimado de desenvolvimento e o tempo em dias úteis decorridos no calendário para conclusão de uma tarefa.

- **Relatório de Bugs:** O sistema contemplará este item se apresentar algum relatório ou gráfico estatístico que demonstre a evolução da quantidade de bugs no projeto ao longo das iterações.

- **Relatório Member Load:** O sistema contemplará este item se apresentar algum relatório ou gráfico estatístico que demonstre a carga de trabalho de cada membro da equipe.

- **Relatório Progresso da Story:** O sistema contemplará este item se apresentar algum relatório ou gráfico estatístico que demonstre o progresso individual das *stories*, analisando o desenvolvimento de suas tarefas.

- **Relatório Release Status:** O sistema contemplará este item se apresentar algum relatório ou gráfico estatístico que demonstre o status do *release* atual.

- **Grafo de Precedências:** O sistema contemplará este item se apresentar grafo de precedência entre os requisitos de software (*stories*). Indicando qual deve ser implementada antes.

- **Relatório Horas Trabalhadas:** O sistema contemplará este item se apresentar algum relatório ou gráfico estatístico que demonstre a quantidade de horas trabalhadas pelos membros da equipe.

- **Interoperabilidade:** O sistema contemplará este item se apresentar alguma forma de intercâmbio de conteúdo entre outra ferramenta de mercado voltada à gerência de projetos, através de arquivos padronizados como XML ou mesmo através de interfaces públicas como Web Services.

7.4 Matriz de funcionalidades

Nesta matriz abaixo, foram classificadas as aplicações estudadas, juntamente com o trabalho desenvolvido no presente trabalho (XP3), na última coluna.

Recurso	XPManager	VersionOne	Scope	XPlanner	XP3
Web	Não	Sim	Não	Sim	Sim
Multi-plataforma	Não	Sim	Não	Sim	Sim
Cadastros de colaboradores	Sim	Sim	Sim	Sim	Sim
Cadastro da Story	Sim	Sim	Sim	Sim	Sim
Anexar Recursos	Não	Sim	Sim	Sim	Sim
Gerência de releases	Sim	Sim	Sim	Não	Sim
Gerência de iterações	Sim	Sim	Sim	Sim	Sim
Visão geral do processo	Sim	Sim	Sim	Não	Sim
Planning Game	Sim	Sim	Sim	Não	Sim
Rastreabilidade	Não	Sim	Não	Sim	Sim
Relatórios Estatísticos	Sim	Sim	Sim	Sim	Sim
Perfis diferenciados	Não	Sim	Sim	Não	Sim
Relatório “Velocity”	Sim	Sim	Sim	Sim	Sim
Relatório de Bugs	Sim	Sim	Sim	Não	Não
Relatório “Member Load”	Sim	Sim	Sim	Sim	Sim
Rel. Progresso da Story	Sim	Sim	Sim	Sim	Sim
Relatório “Release Status”	Sim	Sim	Sim	Sim	Sim
Grafo de Precedências	Não	Não	Não	Não	Sim
Registro Horas Trabalhadas	Sim	Sim	Sim	Sim	Não
Interoperabilidade	Não	Sim	Sim	Sim	Sim

Tabela 5 -Matriz de funcionalidades das ferramentas estudadas

8 Ferramenta Proposta

8.1 Introdução

Para [Nawrocki, 2002], as três principais fontes de conhecimento em projetos XP são o código, os testes automatizados e a memória do programador. Caso o recurso do programador se torne indisponível por qualquer que seja a razão, sobram o código e os testes automatizados, o que pode tornar a manutenção difícil quando os requisitos não estão documentados. [Nawrocki, 2002] também coloca que a comunicação oral, preferida à escrita ou eletrônica é sucetível à erros ou falhas de memória. Para ele, é difícil lembrar exatamente porque e sob quais circunstâncias se tomou determinada decisão em uma comunicação oral, uma vez que não há documentação escrita.

Um dos objetivos da engenharia de requisitos tradicional é gerenciar as mudanças, registrando-as e permitindo posterior rastreabilidade, criando resultados que demonstrem sua principais causas. Metodologias Ágeis fornecem uma base para o gerenciamento de requisitos através dos *storie cards*. A única diferença é o nível de detalhamento dos requisitos, que só aumenta no momento da implementação da *storie*, i.e., dentro de sua iteração, enquanto que na engenharia de requisitos tradicional, há um esforço inicial para elicitação de requisitos com maior grau de detalhamento [Paetsch, 2003].

Para [Beck, 2000], deve-se evitar utilizar técnicas complexas para sua organização (*stories*), uma vez que as mesmas podem dificultar desde a sua escrita até o seu entendimento.

XP é uma metodologia ágil baseada fortemente na comunicação oral sobre a escrita. Para [Nawrocki, 2002], uma forma de conciliar a documentação escrita em projetos XP sem perder a agilidade é delegando a responsabilidade pela documentação e gerenciamento dos requisitos ao testador, que acumula papel de testador e de analista, uma vez que requisitos e testes funcionais estão no mesmo nível. Para este papel, é indicado a utilização de ferramentas para gerenciamento de requisitos, para tornar a tarefa mais ágil, bem como disponibilizar os requisitos na web, para que fiquem visíveis tanto para o cliente quanto para o desenvolvedor, bem como transferir todos os formulários, desenhos, tabelas do cliente para o meio eletrônico, de preferência a internet. [Nawrocki, 2002] acredita ainda que documentando os requisitos em um projeto XP através da figura do testador pode acrescentar ainda mais agilidade do ponto de vista do desenvolvedor.

Para [Nuseibeh, 2000], é de importância crescente a necessidade de que os requisitos possam não somente ser escritos, mas também rastreados, para que se possa gerenciar sua evolução no tempo. Vários padrões de documentação de requisitos já foram desenvolvidos, entretanto, sua estrutura necessita ser construída para cada contexto em particular.

De acordo com [Pinheiro, 2004], o uso de modelos formais auxilia o processo de automatizar a rastreabilidade, através de sua geração automática, permitindo a criação inclusive de procedimentos que venham a permitir a verificação de sua consistência.

Para [Pinheiro, 2004], em muitos casos, a informação que lidamos está representada de forma não-estruturada, em linguagem natural, como em entrevistas, documentos e imagens. A tradução de tais informações para um meio estruturado pode facilitar sua manipulação, no entanto, não atende às necessidades relativas à rastreabilidade. Nestes casos, vincular os objetos é uma importante decisão para torná-los rastreáveis, no entanto, o raciocínio por trás de cada decisão só será descoberto quando se chegar à tal objeto. Para [Pinheiro, 2004], o registro de objetos não-funcionais deve permitir a inserção de objetos hiper-mídia, como: documento, texto, vídeo e imagem. Também pode-se utilizar determinadas marcações em partes de textos em documentos, para organizar posterior verificação.

Baseados nos pressupostos vistos até o momento, propõe-se a criação de uma ferramenta para gerir a informação gerada naturalmente em projetos *Extreme Programming*.

8.2 Implementação

Para o desenvolvimento da ferramenta proposta, serão utilizadas as seguintes tecnologias:

- **Apache 2:** O Apache (www.apache.org) é um servidor de páginas web mundialmente reconhecido por sua qualidade e segurança.
- **PHP 5 :** O PHP (www.php.net) é uma linguagem de programação de última geração voltada originalmente para a web com grandes recursos de orientação a objetos e flexibilidade devido à sua tipagem dinâmica.
- **SQLite 2:** O SQLite (www.sqlite.org) é um banco de dados em formato de arquivos que possuem grande poder de armazenamento de informações (até 2 terabytes) e implementa o padrão SQL92. Possui grande portabilidade pois dispensa a utilização de um servidor.

8.3 Especificação

A ferramenta proposta será projetada com a utilização da linguagem de modelagem UML (Unified Modeling Language). A UML é uma notação, e não um método. A UML define uma linguagem gráfica que permite visualizar, especificar, construir e documentar [Matos, 2002].

Da UML serão utilizados os seguintes artefatos: Diagrama de casos de uso (que irá modelar os principais usuários, seus papéis e as respectivas operações desempenhadas no sistema), diagrama de classes (quer irá modelar a estrutura, bem como as relações entre entidades do sistema), diagrama de atividades (que irá modelar o fluxo do processo de planejamento), diagrama de estados (que irá demonstrar o estado dos objetos e suas transições durante o ciclo do sistema).

8.3.1 Arquitetura

No diagrama de arquitetura abaixo, representamos características físicas da arquitetura do ambiente onde o sistema será instalado. Detalhes como bancos de dados, tecnologia utilizada e componentes utilizados podem ser aqui demonstrados [Matos, 2002].

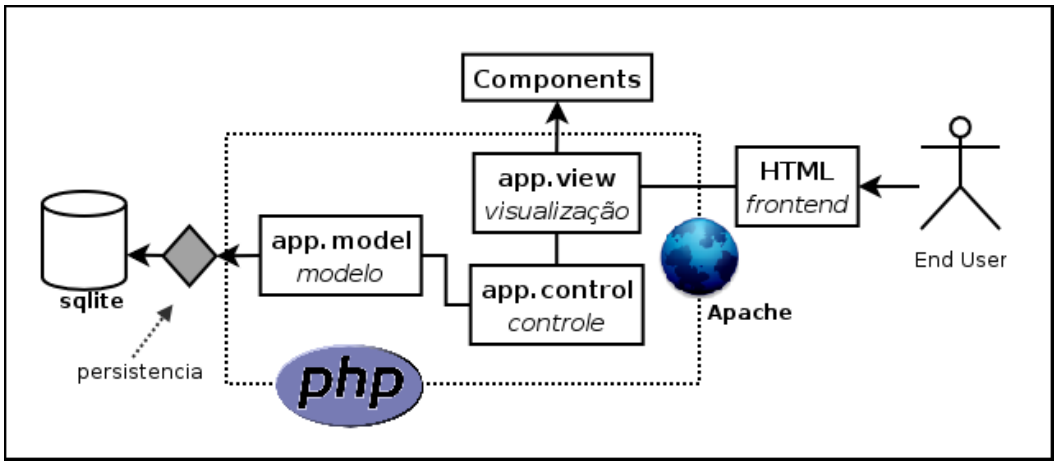


Figura 17 -Arquitetura da ferramenta

Para o desenvolvimento do seguinte trabalho, foi utilizado o *design pattern* MVC (*Model, View, Controller*), que permite uma segmentação dos aspectos de programação. O padrão MVC apresenta uma forma de estruturar a aplicação de forma que os aspectos de programação fiquem divididos em três camadas, de acordo com a seguinte arquitetura [Gamma, 1995] :

Model (app.model)

- Abstrações do mundo real;
- Contém os objetos de negócio;
- Contem dados e métodos que atuam sobre estes objetos;
- Alerta objetos “observadores” quando os dados sofrem alterações;

View (app.view)

- Provê a interface do usuário;
- Renderiza os modelos de dados na interface;
- Responsável pela atualização da interface quando da alteração do modelo;

Controller (app.control)

- Responsável por controlar o fluxo da aplicação;
- Coordena as camadas Model e View para realizar operações dadas pela interação do usuário;

8.3.2 Casos de Uso

Casos de Uso (*Use Cases*) são modelos de cenários que representam situações nas quais o usuário interage com o sistema para atingir um determinado objetivo. É uma técnica voltada à identificação dos requisitos do sistema. O modelo de Use Cases é a base para o desenvolvimento orientado a objetos proposto e também central na utilização de UML [Matos, 2002].

Até o momento, de acordo com as atividades sugeridas por [Beck, 2000], acrescidas de funcionalidades básicas (cadastros) que são necessárias na gerência de um projeto, são os seguintes os papéis e atividades a serem implementados na ferramenta proposta (XP3).

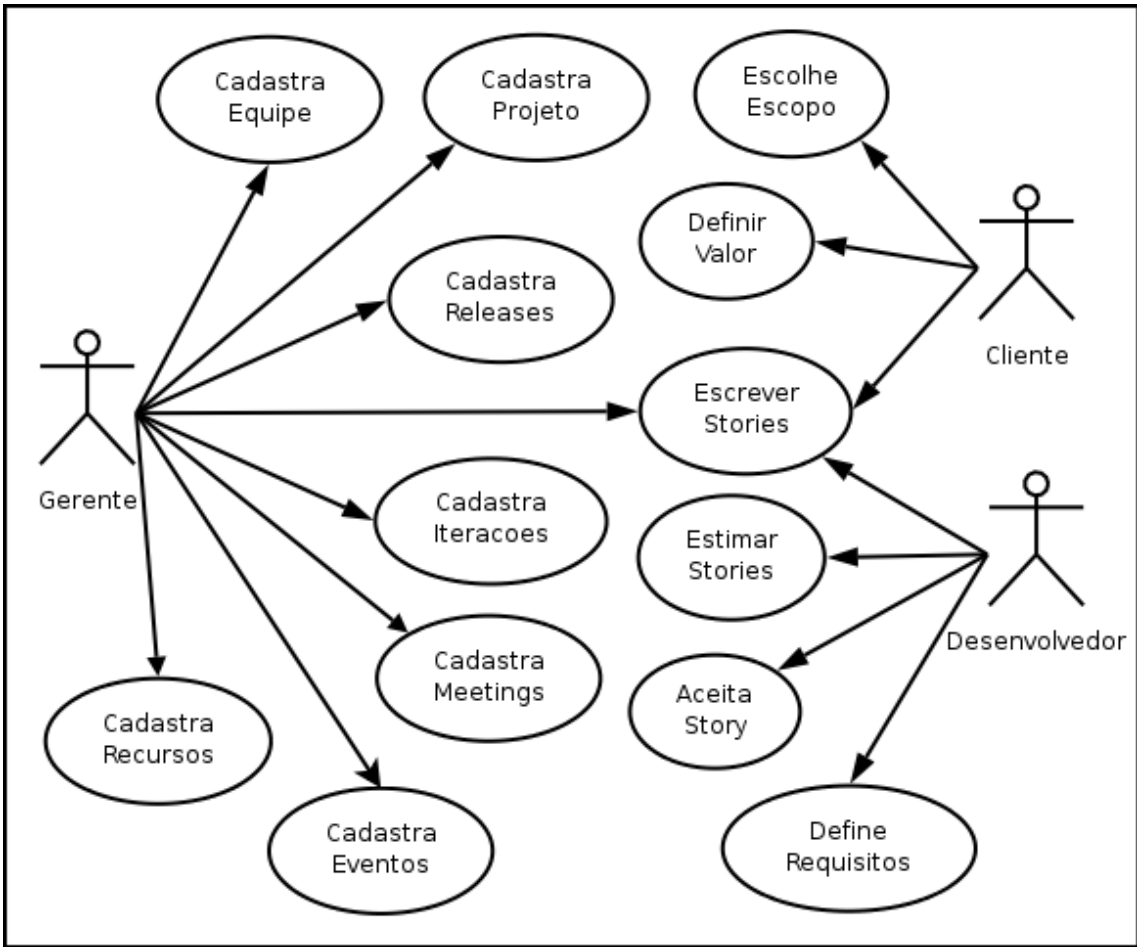


Figura 18 -Diagrama de casos de uso

Caso de Uso	Cadastrar Stakeholders
Atores	Gerente
Descrição	Operação de Cadastro de Stakeholder. Stakeholder pode ser cliente, desenvolvedor ou gerente.
Pré-condições	• Usuário logado no sistema; • Usuário com perfil de gerente;
Pós-condições	• Stakeholder cadastrado com sucesso;
Fluxo Básico	1. Usuário Preenche cadastro detalhado; 2. Usuário seleciona perfil do novo stakeholder; 3. Usuário remete dados; 4. Sistema armazena novo usuário;
Fluxo de excessão	1. Caso o usuário tenha deixado algum campo validado vazio: 2. Sistema emite mensagem de alerta; 3. Usuário repete preenchimento;

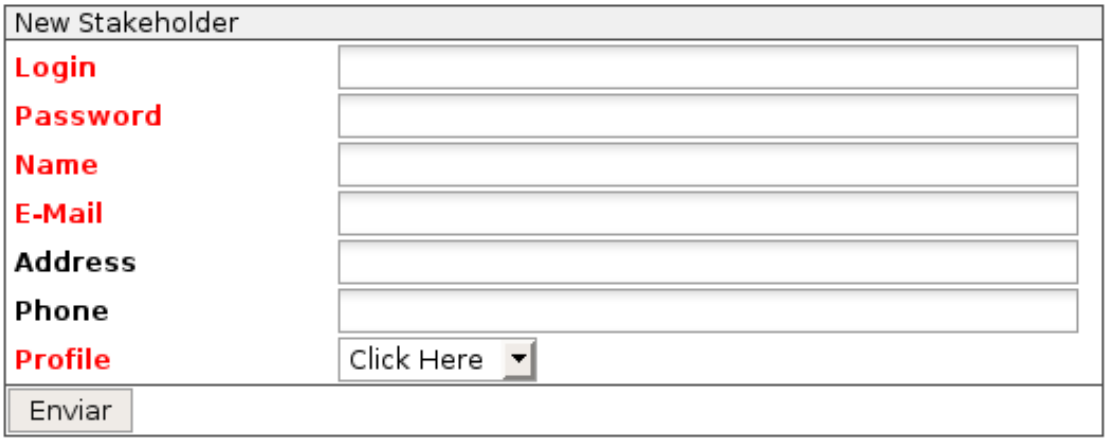


Figura 19 -Cadastro de stakeholders

Caso de Uso	Cadastrar Projeto
Atores	Gerente
Descrição	Operação de cadastro de novo projeto.
Pré-condições	• Usuário logado no sistema; • Usuário com perfil de gerente;
Pós-condições	• Projeto cadastrado com sucesso;
Fluxo Básico	1. Usuário preenche cadastro do projeto; 2. Usuário seleciona o cliente do projeto; 3. Usuário remete dados;
Fluxo de excessão	1. Caso o usuário tenha deixado algum campo validado vazio: 2. Sistema emite mensagem de alerta; 3. Usuário repete preenchimento;

Figura 20 -Cadastro de projeto

Caso de Uso	**Cadastrar Release**
Atores	Gerente
Descrição	Operação de cadastro de nova release.
Pré-condições	• Usuário logado no sistema; • Usuário com perfil de gerente;
Pós-condições	• Release cadastrado com sucesso;
Fluxo Básico	1. Usuário preenche cadastro da release; 2. Usuário seleciona o projeto da release; 3. Usuário remete dados;
Fluxo de excessão	1. Caso o usuário tenha deixado algum campo validado vazio: 2. Sistema emite mensagem de alerta; 3. Usuário repete preenchimento;

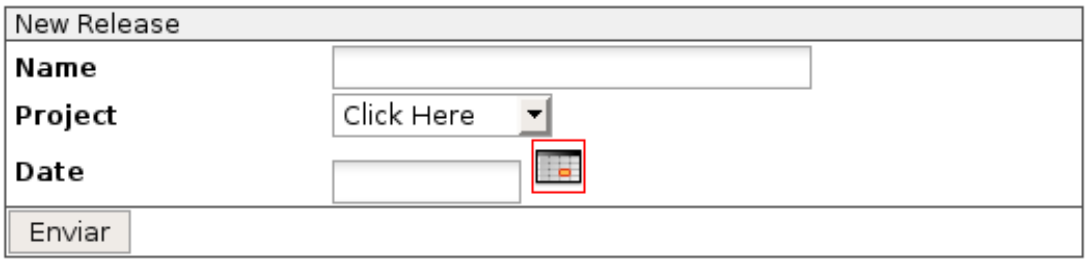


Figura 21 -Cadastro do release

Caso de Uso	Cadastrar Iteração
Atores	Gerente
Descrição	Operação de cadastro de nova iteração.
Pré-condições	• Usuário logado no sistema; • Usuário com perfil de gerente;
Pós-condições	• Iteração cadastrada com sucesso;
Fluxo Básico	1. Usuário preenche cadastro da iteração; 2. Usuário seleciona a release da iteração; 3. Usuário remete dados;
Fluxo de excessão	1. Caso o usuário tenha deixado algum campo validado vazio: 2. Sistema emite mensagem de alerta; 3. Usuário repete preenchimento;

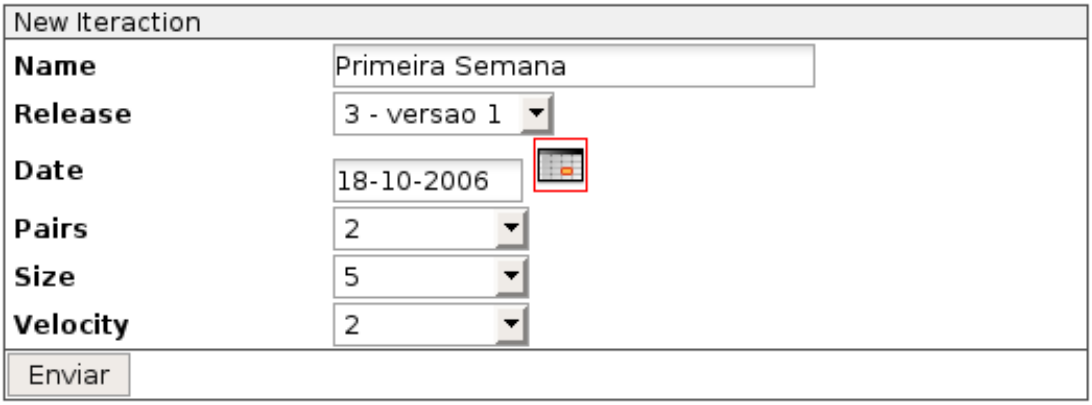


Figura 22 -Cadastro de iteração

Caso de Uso	Cadastrar Stand Up Meeting
Atores	Gerente
Descrição	Operação de cadastro de *stand up meeting*
Pré-condições	• Usuário logado no sistema; • Usuário com perfil de gerente;
Pós-condições	• *Stand Up Meeting* cadastrado com sucesso;
Fluxo Básico	1. Usuário preenche cadastro do encontro; 2. Usuário seleciona o projeto do encontro; 3. Usuário remete dados;
Fluxo de excessão	1. Caso o usuário tenha deixado algum campo validado vazio: 2. Sistema emite mensagem de alerta; 3. Usuário repete preenchimento;

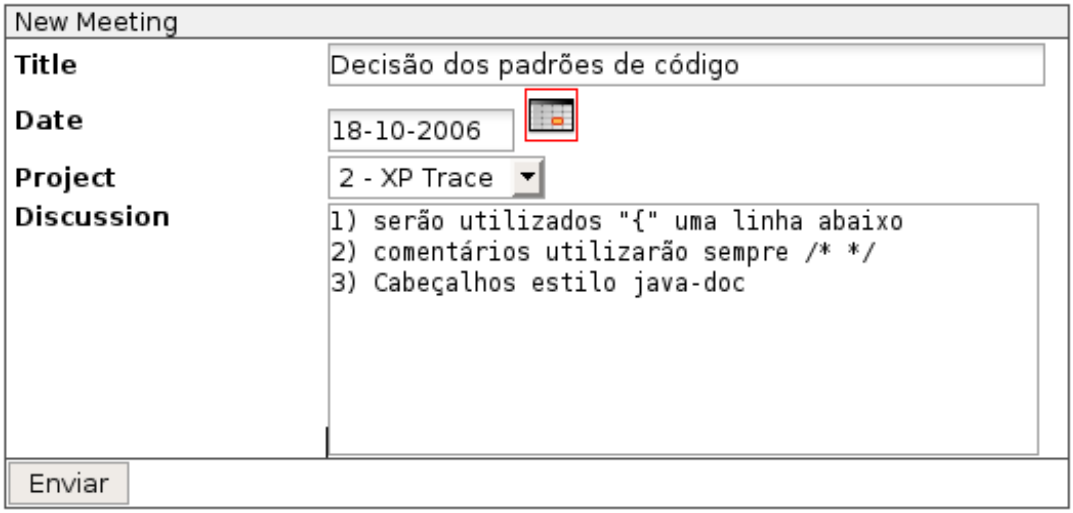


Figura 23 -Cadastro de *stand up meeting*

Caso de Uso	**Cadastrar Eventos**
Atores	Gerente
Descrição	Cadastra eventos que acontecem no projeto
Pré-condições	• Usuário logado no sistema; • Usuário com perfil de gerente;
Pós-condições	• Evento cadastrado com sucesso;
Fluxo Básico	1. Usuário preenche cadastro do evento; 2. Usuário seleciona o projeto do evento; 3. Usuário remete dados;
Fluxo de excessão	1. Caso o usuário tenha deixado algum campo validado vazio: 2. Sistema emite mensagem de alerta; 3. Usuário repete preenchimento;

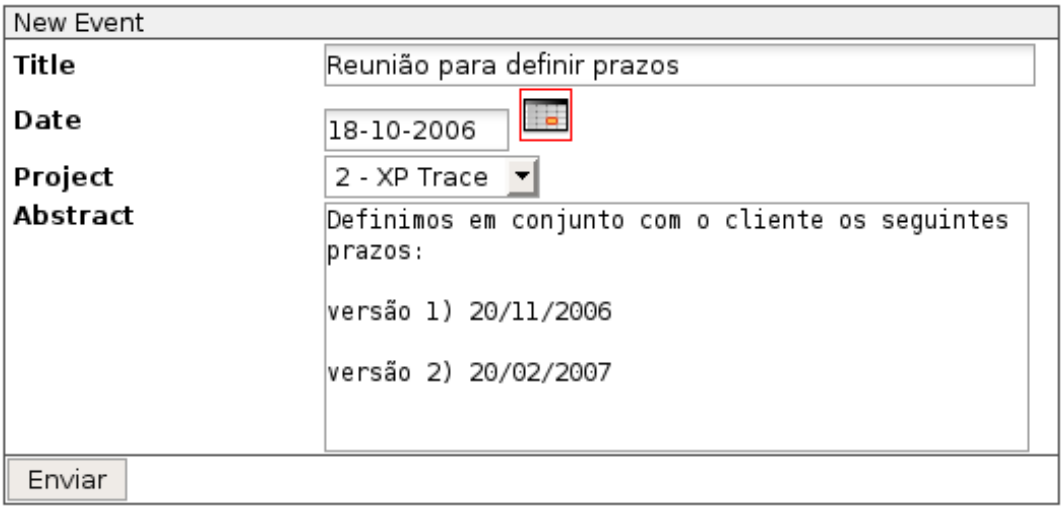


Figura 24 -Cadastro de eventos

Caso de Uso	Cadastrar Recursos
Atores	Gerente
Descrição	Cadastra um determinado recurso para o projeto. Pode ser documentação de arquitetura, decisões, hierarquia, dentre outros “adornos” ao projeto.
Pré-condições	• Usuário logado no sistema; • Usuário com perfil de gerente; • Projeto padrão definido pelo usuário;
Pós-condições	• Recurso cadastrado com sucesso;
Fluxo Básico	1. Usuário preenche cadastro do recurso; 2. Usuário seleciona a fonte do recurso (arquivo); 3. Usuário remete dados;
Fluxo de excessão	1. Caso o usuário tenha deixado algum campo validado vazio: 2. Sistema emite mensagem de alerta; 3. Usuário repete preenchimento;

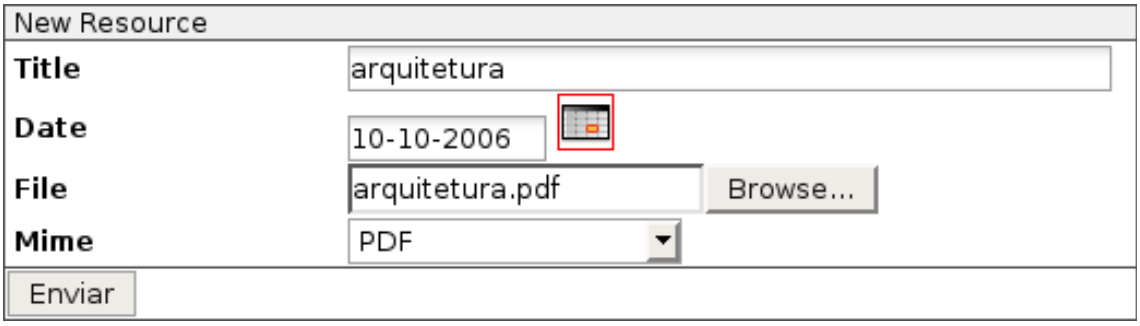


Figura 25 -Cadastro de recursos

Caso de Uso	Cadastrar nova Story
Atores	Gerente Desenvolvedor Cliente
Descrição	Cadastra uma nova *story* no sistema, juntamente com todos seus detalhes.
Pré-condições	• Usuário logado no sistema;
Pós-condições	• *Story* cadastrado com sucesso; • Registro de rastreamento efetuado;
Fluxo Básico	1. Usuário preenche cadastro básico da *story*; 2. Usuário preenche histórico da *story*; 3. Usuário preenche tarefas da *story*; 4. Usuário preenche testes da *story*; 5. Usuário remete dados;
Fluxo de excessão	1. Caso o usuário tenha deixado algum campo validado vazio: 2. Sistema emite mensagem de alerta; 3. Usuário repete preenchimento;

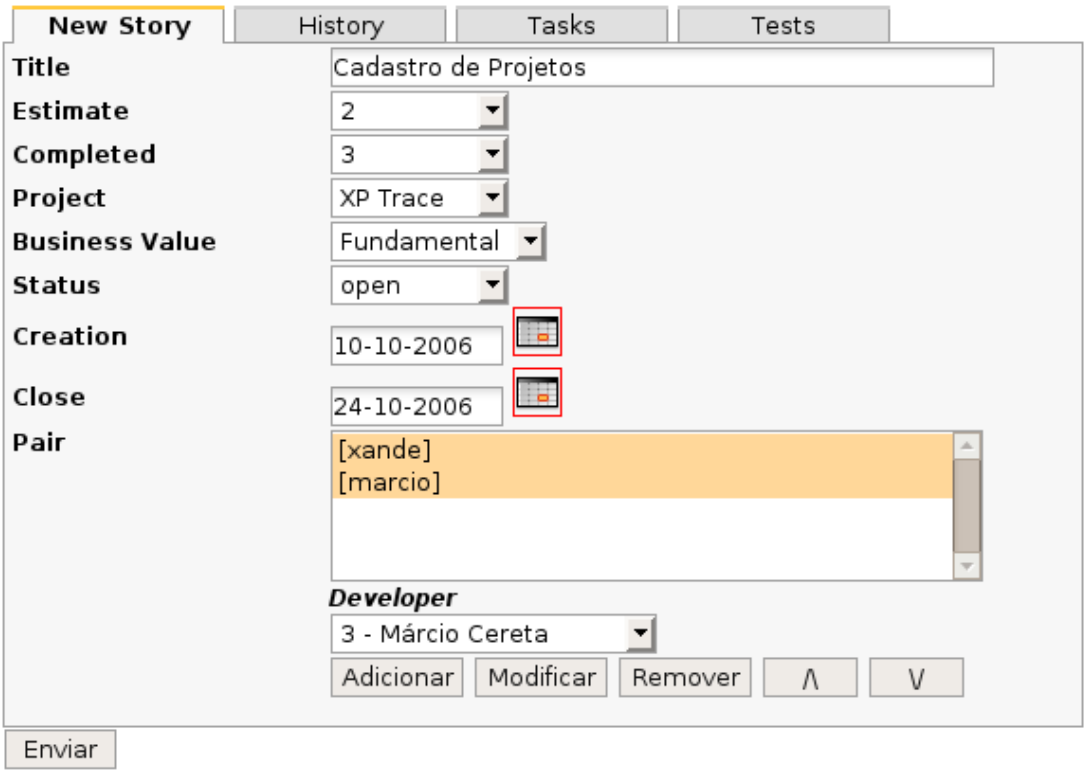


Figura 26 -Cadastro básico da *story*

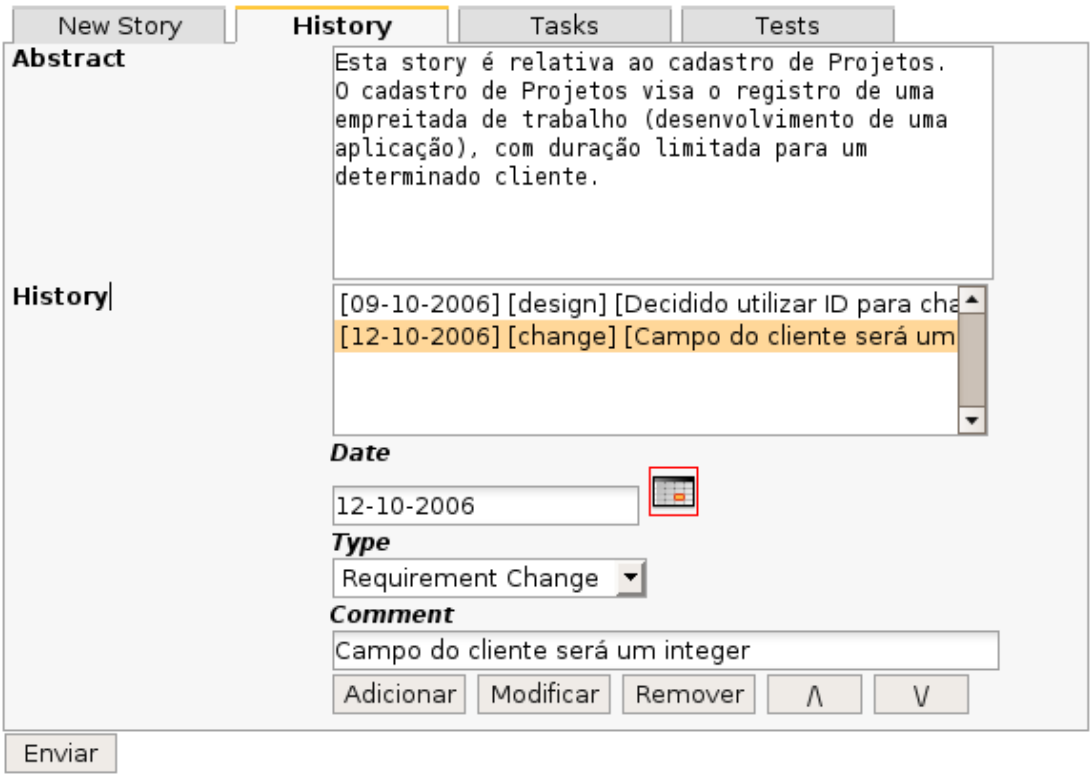


Figura 27 -Cadastro do histórico da *story*

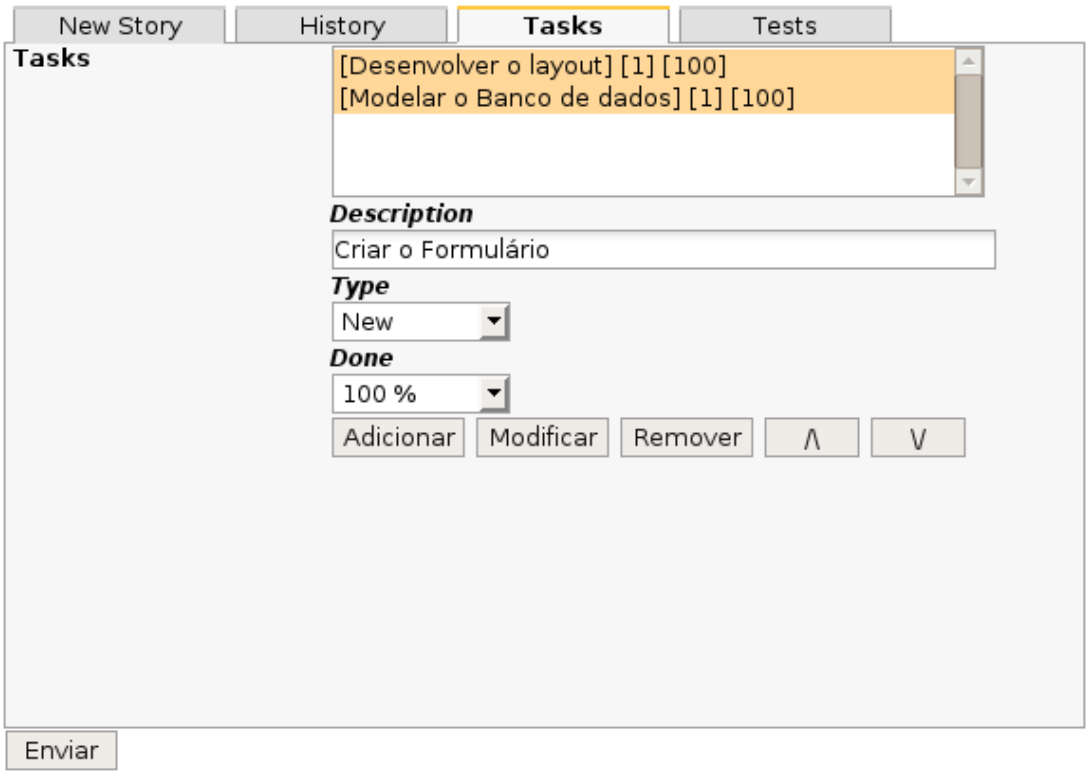


Figura 28 -Cadastro das tarefas de uma *story*

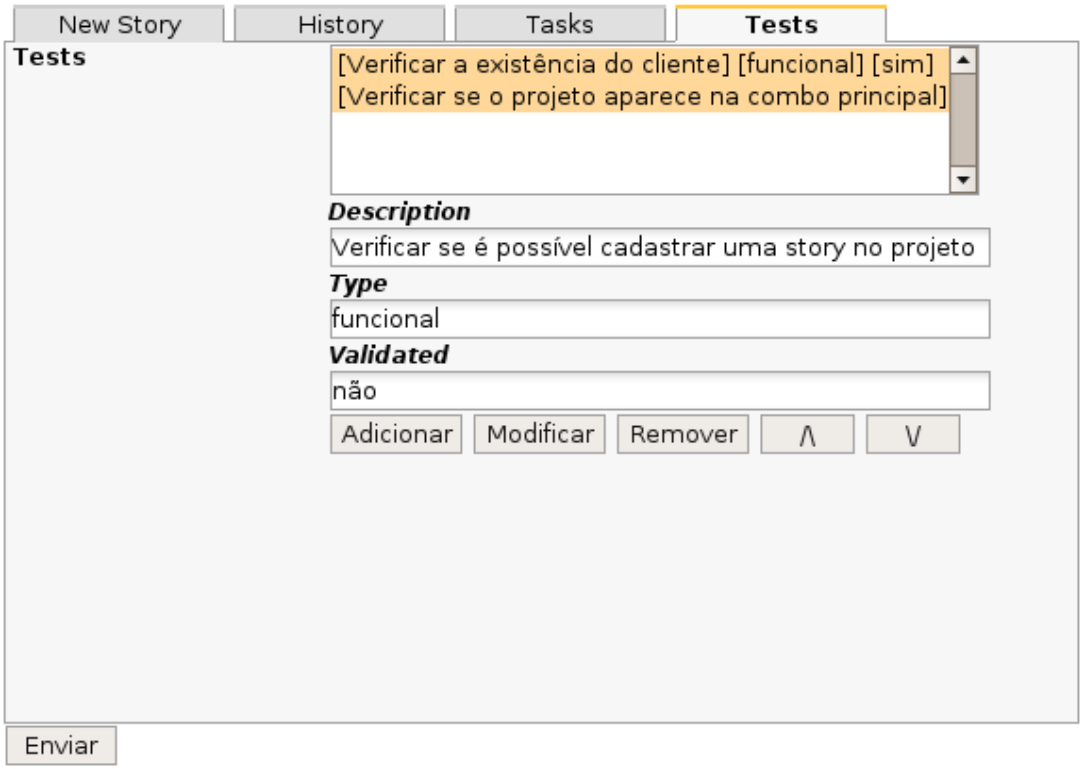


Figura 29 -Cadastro de testes da *story*

Caso de Uso	**Estimar Story**
Atores	Desenvolvedor
Descrição	Define a estimativa de tempo para implementação de uma *story*.
Pré-condições	• Usuário logado no sistema; • Projeto padrão definido pelo usuário;
Pós-condições	• Estimativa gravada com sucesso; • Registro de rastreamento efetuado;
Fluxo Básico	1. Sistema pré-seleciona stories do projeto padrão que ainda não foram estimadas; 2. Usuário estima stories; 3. Usuário remete dados;
Fluxo de excessão	Não aplicável.

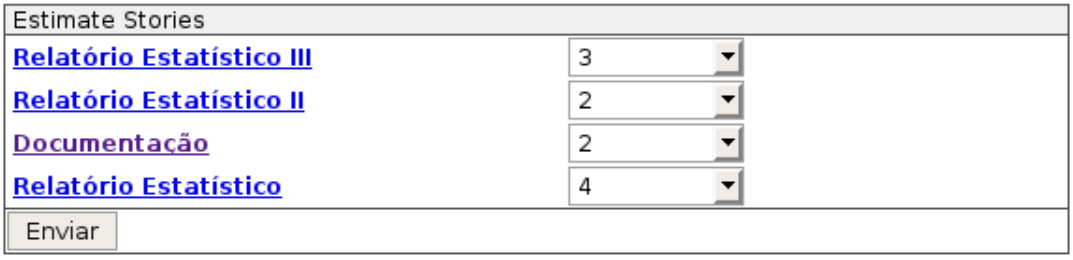


Figura 30 -Estimar *story*

Caso de Uso	Definir Valor da Story
Atores	Cliente
Descrição	Define o valor de negócio de uma *story*.
Pré-condições	• Usuário logado no sistema; • Projeto padrão definido pelo usuário;
Pós-condições	• Valor gravado com sucesso; • Registro de rastreamento efetuado;
Fluxo Básico	1. Sistema pré-seleciona stories do projeto padrão que ainda não foram valoradas; 2. Usuário define o valor de negócio das stories; 3. Usuário remete dados;

Figura 31 -Definir o valor da *story*

Caso de Uso	Definir Pré-requisitos da Story
Atores	Desenvolvedor
Descrição	Define o pré-requisito de uma *story*.
Pré-condições	• Usuário logado no sistema; • Projeto padrão definido pelo usuário;
Pós-condições	• Pré-requisito gravado com sucesso; • Registro de rastreamento efetuado;
Fluxo Básico	1. Sistema pré-seleciona stories do projeto padrão; 2. Usuário define pré-requisitos das stories; 3. Usuário remete dados;

Figura 32 -Definir Pré-requisitos da *story*

Caso de Uso	Seleção de Escopo
Atores	Cliente
Descrição	Define o escopo de uma iteração.
Pré-condições	• Usuário logado no sistema; • Projeto padrão definido pelo usuário;
Pós-condições	• Stories alocadas em iterações; • Registro de rastreamento efetuado;
Fluxo Básico	1. Sistema lista iterações do projeto padrão; 2. Usuário seleciona iteração que deseja planejar; 3. Sistema lista stories que ainda não foram alocadas; 4. Usuário seleciona stories para alocar naquela iteração; 5. Usuário remete dados;
Fluxo de excessão	Não aplicável.

Release	Iteraction	
Versão 1.0		
	Primeira Semana	Data 12-07-2006
	Segunda Semana	Data 19-07-2006
	Terceira semana	Data 26-07-2006
Versão 2.0		
	Quarta Semana	Data 02-08-2006
	Quinta semana	Data 07-08-2006
	Sexta Semana	Data 15-08-2006
	Sétima Semana	Data 21-08-2006
Versão 3.0		
	Oitava Semana	Data 04-09-2006
	Nona Semana	Data 11-09-2006

Figura 33 -Seleção da iteração

Após selecionar a iteração à qual se deseja alocar as *stories*, o sistema irá exibir a quatidade todal de dias disponíveis na iteração, a quantidade de dias alocados e a quantidade de dias restantes, seguindo a fórmula sugerida por [Beck, 2000]:

- **`Available Days = tamanho iter * pares`**
- **`load_factor = tamanho iter  * pares / velocity`**
- **`Alocated Days = sum (stories->estimate * load_factor)`**

Available Days: 10
Alocated Days: 0
Left: 10

Choose Scope
Pesquisar ferramentas de fórum **(5 days)** ☐ Instalar now-rau **(7.5 days)** ☐
Enviar

Figura 34 -Seleção do escopo da iteração

Caso de Uso	Aceite de Story
Atores	Desenvolvedor
Descrição	Aceita uma *story*
Pré-condições	• Usuário logado no sistema; • Projeto padrão definido pelo usuário;
Pós-condições	• Formação dos pares registrado na *story*; • Registro de rastreamento efetuado;
Fluxo Básico	1. Sistema lista stories sem par formado; 2. Usuário seleciona stories a trabalhar; 3. Usuário remete dados;
Fluxo de excessão	Não aplicável.

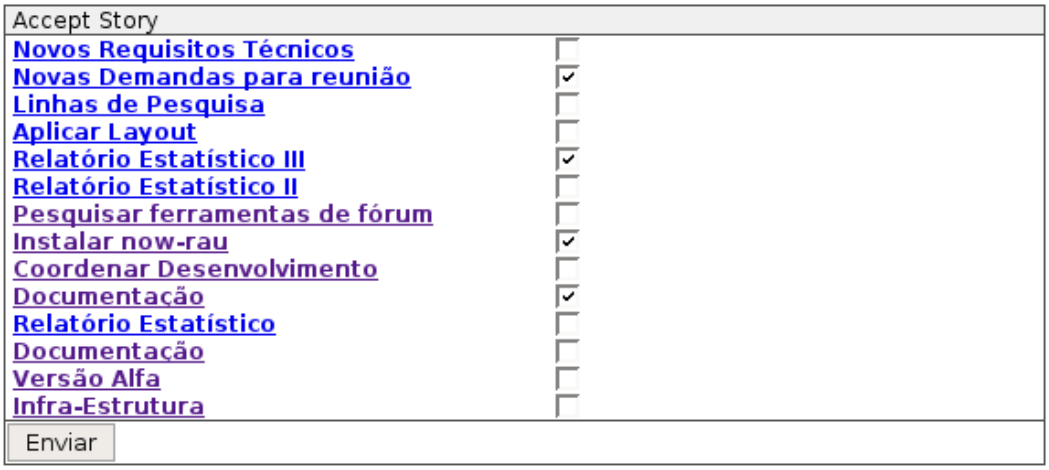


Figura 35 -Aceite de *story*

8.3.3 Atividades

O Diagrama de atividades nos dá uma visão macro de processo de um Use Case. Nos auxilia na tarefa de visualizar quem participa do processo. É basicamente um fluxograma adaptado para a realidade da Orientação a Objetos. Ele focaliza o fluxo de atividades envolvidas em um processo. Mostra quais atividades dependem uma da outra. Também levanta classes candidatas para participar do processo [Matos, 2002].

De acordo com as atividades sugeridas por [Beck, 2000], pode-se modelar o fluxo básico da aplicação, em conjunto com as atividades de planejamento de iterações, da seguinte maneira:

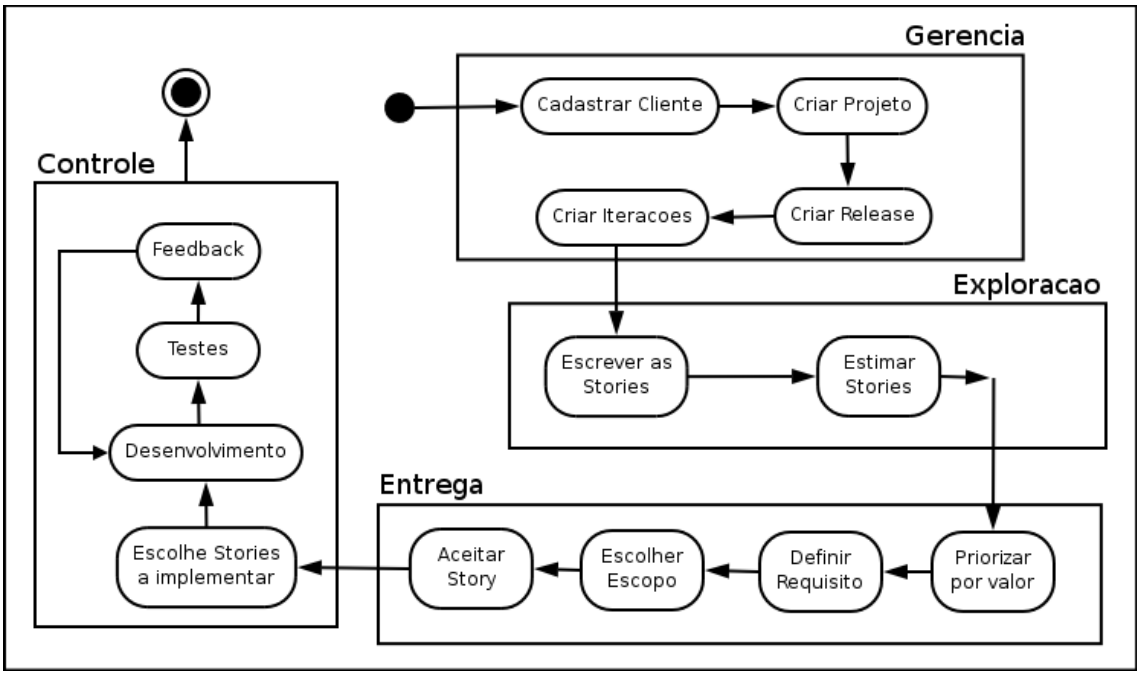


Figura 36 -Diagrama de atividades

8.3.4 Classes

O modelo de classes representa conceitos do mundo real importantes para o sistema através de uma estrutura estática onde são vistos classes, relacionamentos, atributos e operações. Representa os requisitos que levantamos e não deve ser utilizado para representar entidades do modelo de dados, como muitas vezes é utilizado erroneamente. Lembre-se que o modelo de dados server para armazenar os dados com segurança e confiabilidade de forma estruturada. Neste momento, nossa preocupação deve ser levantar as classes de negócio a partir dos requisitos iniciais levantados. Além disso, surgirá a necessidade de classes de limite (interface, persistência) e controladoras (para gerenciar a interação entre as classes de negócio e as classes de limite), que devem então ser adicionadas [Matos, 2002].

[Breitman, 2002] propõe enriquecer o atual processo de documentação em projetos XP, agregando as *user stories* ao produto final, que é o código, tendo ganhos como: o armazenamento e a pesquisa desta informação; rastreabilidade através de informações como data, prioridade e risco e a possibilidade de derivar testes a partir das *stories*. As *stories* são o ponto de partida do desenvolvimento em um projeto XP. Elas podem ser decompostas em tarefas (*tasks*) e estimadas, permitindo assim, quantificar o esforço de desenvolvimento.

Para [Beck, 2000], as *stories* devem ser descartadas após o uso, pois o código está sempre sujeito à mudanças e é custoso sincronizar código e *stories*. [Breitman, 2002] discorda de Beck, pois o conjunto de *stories* é a documentação do projeto, e o principal argumento que sustenta a idéia de que a documentação deve persistir, é para prover rastreabilidade. Também argumenta que o problema da manutenção de tais informações pode ser simplificado pela adoção de uma estratégia de gerenciamento que garanta o histórico das mudanças. [Beck, 2000] argumenta ser desnecessário manter duas fontes basicamente com a mesma informação. Já [Breitman, 2002] discorda pelo fato da

natureza da informação contida em ambos (*stories* e código) é diferente. O código não contém informações como prioridade e risco.

[Breitman, 2002] sugere a criação de um modelo onde, além de informações das *stories* (roteiros), exista um glossário de termos utilizados no projeto (*glossary*), garantindo que ambos desenvolvedores e clientes tenham o mesmo entendimento.

Abaixo, a idéia de modelo apresentada por [Breiman, 2002]. O modelo é representado pelo roteiro (*scenario*) que contém atributos como meta, contexto, prioridade e risco. As demais classes, quando instanciadas, irão capturar informações extra no processo. As *tags* capturam as informações relacionadas às versões do cenário, como versão, data, atividade e propriedade. A classe *Trace* possibilita criar links com outras *stories* para descrever relacionamentos. A classe *Rationale* armazena decisões de *design*, que podem ser descritas em linguagem natural.

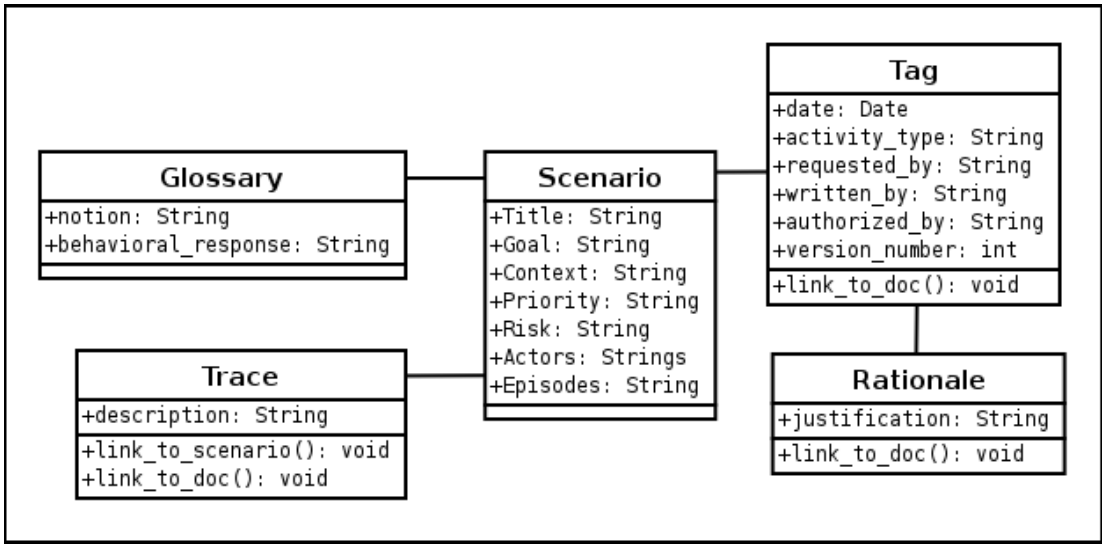


Figura 37 -Modelo segundo [Breitman, 2002]

Para atingir os objetivos da aplicação, é proposto o seguinte diagrama de classes para suportar as entidades básicas (*Customer, Project, Release, Iteration, Story, Pair e Developer*), bem como entidades necessárias para garantir a rastreabilidade dos requisitos (*Resources, History, Tasks e Tests*), de acordo com o diagrama a seguir:

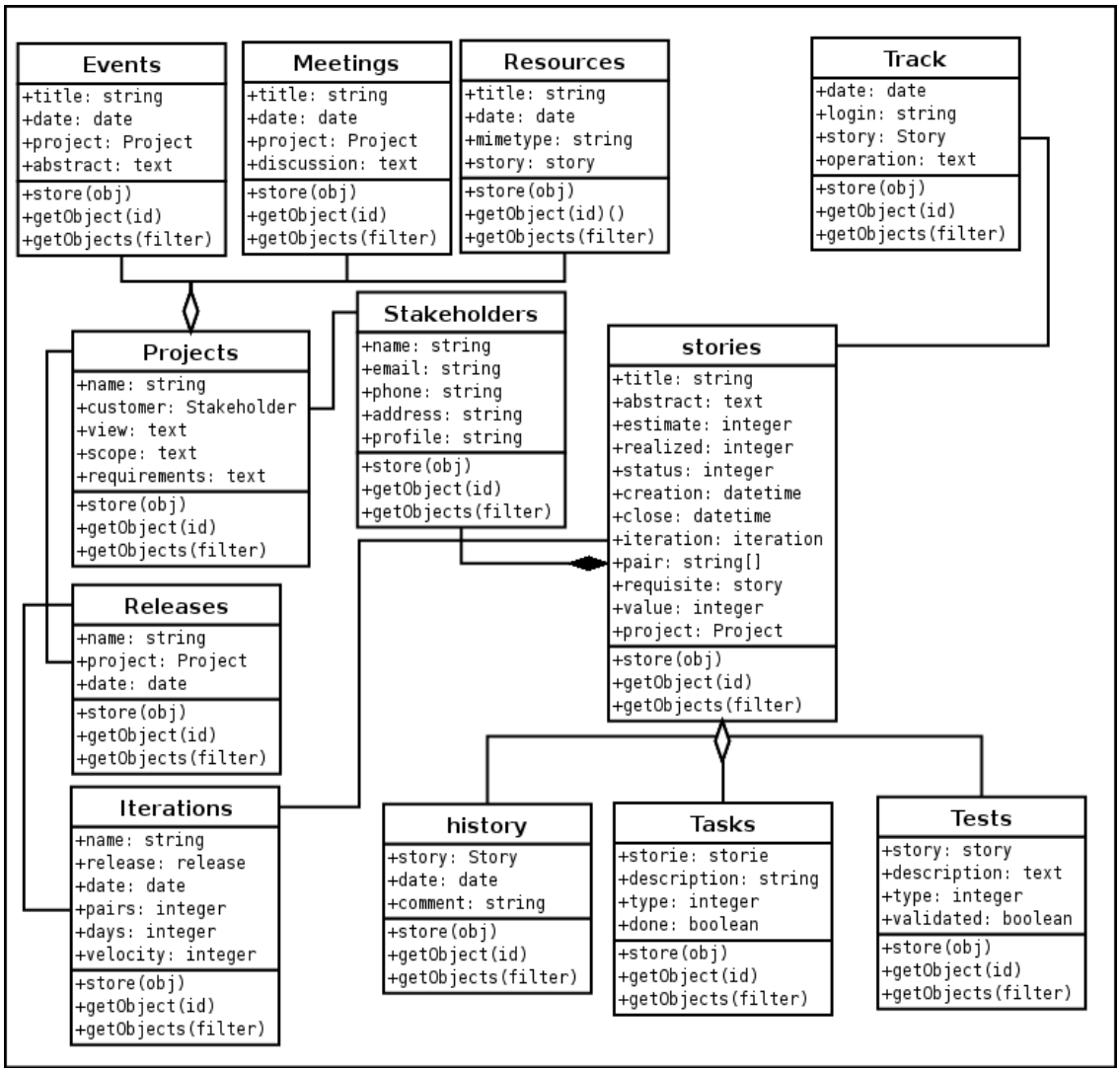


Figura 38 -Diagrama de classes

Descrição das Classes:

- **Stakeholders**: São os membros do projeto. Podem ser cliente, gerente ou desenvolvedor. Possui alguns dados básicos de contato como telefone e email, dentre outros atributos.
- **Projects**: São os diferentes projetos manipulados pelo sistema. O sistema suporta múltiplos projetos, cada projeto tem seu cliente, além de informações gerenciais como visão, escopo e requisitos sucintos.
- **Releases**: São os diferentes *releases* de um projeto. Possuem informações essenciais como nome e data de lançamento.

- **Iterations**: São as diferentes iterações de um *release*. Cada iteração possui um nome, data de início, quantidade de pares alocados, dias de trabalho e velocidade de trabalho em dias ideais.
- **Stories**: São os requisitos do sistema. Cada *story* possui informações como seu título, resumo, estimativa, tempo realizado, data de criação, data de conclusão, par de programadores, pré-requisitos e valor de negócio. Além disto, possui alguns objetos agregados como *History, Tasks e Tests*, vistos abaixo.
- **History**: Armazena o histórico de decisões de uma *story*, através de informações como a data, um comentário e o tipo de decisão (*design*, alteração de requisitos, etc...)
- **Tasks**: São as diferentes tarefas nas quais uma *story* é dividida. Cada tarefa possui descrição, tipo (Nova, *Bug Fix*, Melhoria) e percentual de conclusão.
- **Tests**: São os testes de aceitação da *story*. Provê descrição, tipo (funcional, não funcional) e se está validado ou não.
- **Track**: Provê rastreabilidade. Armazena todas operações processadas no sistema, bem como data e quem fez, juntamente com uma "fotografia" da *story* no sistema no momento em que esta sofreu alguma alteração.
- **Events**: Armazena os diferentes tipos de eventos em um projeto, como: comunicações importantes, emails trocados, reuniões com o cliente, dentre outros.
- **Meetings**: Armazena os *Stand Up Meetings*, que são as rápidas reuniões matinais realizadas antes de se iniciar um dia de trabalho.
- **Resources**: Armazena um "recurso". Em princípio, um recurso é qualquer arquivo que possa prover de subsídios (artefatos) que levaram à concepção de um determinado requisito (*story*). Estes artefatos podem ser documentos contendo decisões ou modelos de arquitetura, dentre outros.

8.3.5 Objetos

O diagrama de objetos é uma variação do diagrama de classes. Ao invés de exibir a estrutura estática das classes, seu foco é a demonstração de um conjunto de instâncias de objetos em tempo real, como uma "fotografia" da memória da aplicação enquanto estiver em execução.

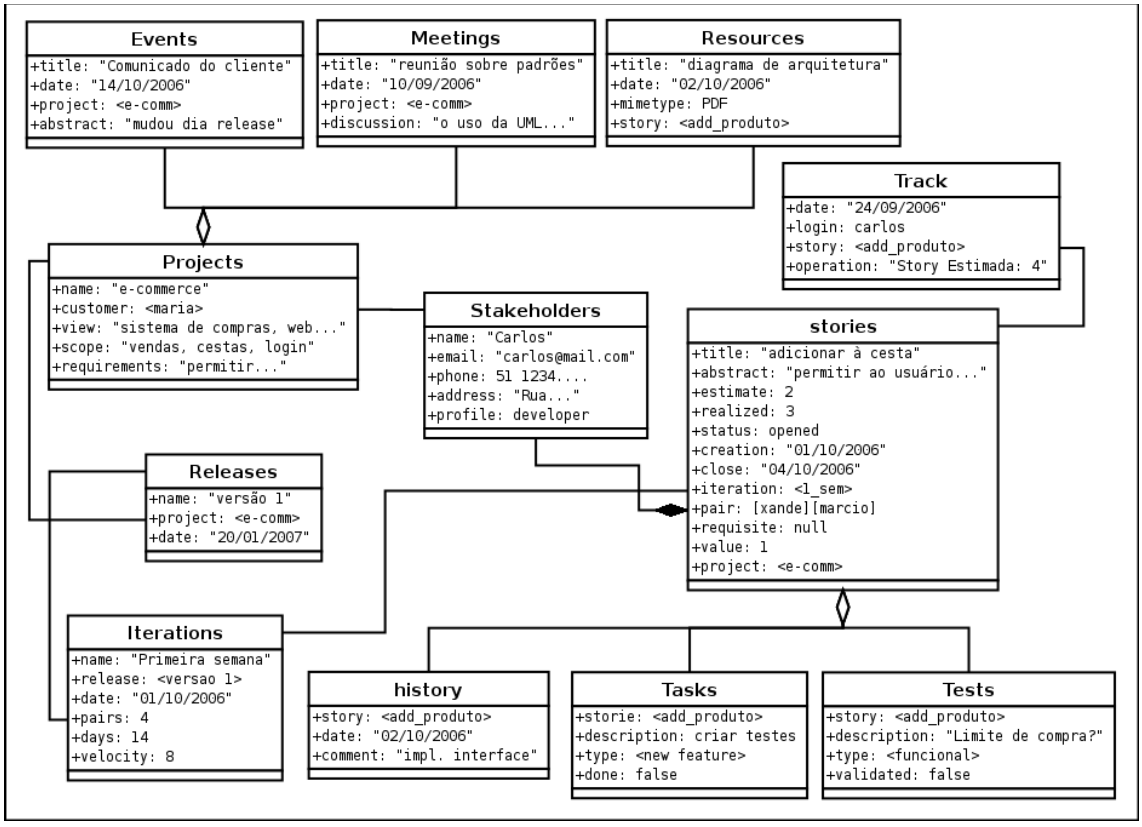


Figura 39 -Diagrama de objetos

8.3.6 Estados

Mostra os possíveis estados de um objetos durante o ciclo de vida dele no sistema, bem como aos eventos que o levaram a trocar de estado (transições). O estado é a situação de um objeto em memória em um determinado momento. O diagrama de estados nos esclarece a participação de um determinado objeto no sistema ao longo de seu ciclo de vida, a fim de obter um maior entendimento do mesmo [Matos, 2002].

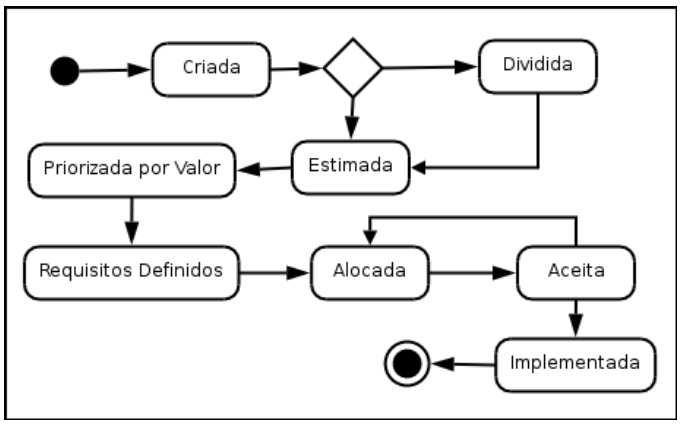


Figura 40 -Diagrama de estados da *story*

8.4 Gestão do Processo

8.4.1 PERT/CPM

Após a priorização dada pelo cliente, não há ordem específica para implementação das tarefas de uma *story*. Sugere-se o uso de ferramentas tradicionais como PERT para navegar entre as tarefas e tomar as decisões que levem à um melhor aproveitamento de tempo durante sua implementação [Beck, 2000].

Na ferramenta desenvolvida no presente trabalho, através da utilização da biblioteca para geração de grafos *graphviz*, reunimos o conjunto de *user stories*, passando as informações de precedência (que irá definir a direção das setas) e conclusão das *stories* (que irá definir a cor das *stories*) para um algorítmo específico que desenha os círculos e suas relações entregando uma imagem pronta para a aplicação. As *stories* preechidas são aquelas ainda não realizadas e podem indicar um caminho crítico dentro do projeto.

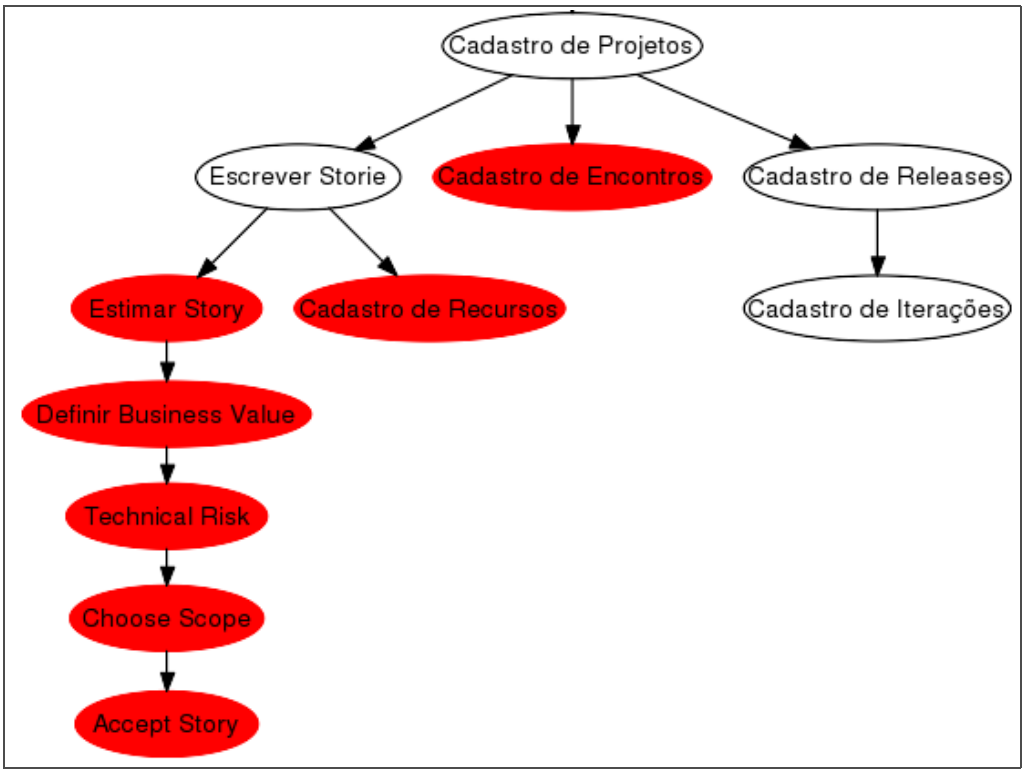


Figura 41 -Gráfico de Pert/CPM gerado

8.4.2 Workflow

Para [Carvalho, 2000], o *Workflow* permite coordenar os processos de negócios de uma empresa, determinando o fluxo do processo e permitindo o acompanhamento de todas as atividades que o constituem.

Dentro da aplicação desenvolvida para o presente trabalho, existe uma seção chamada "*Workflow*", onde o gerente, o desenvolvedor e o cliente podem consultar o status geral do processo, bem como visualizar a quantidade de *stories* "pendentes" em cada etapa. Na imagem abaixo, vemos que faltam 4 *stories* para serem estimadas, 7 para serem definido o valor de negócios, 10 para definir os pré-requisitos, 2 para definir o escopo e 14 para serem aceitas por membros da equipe.

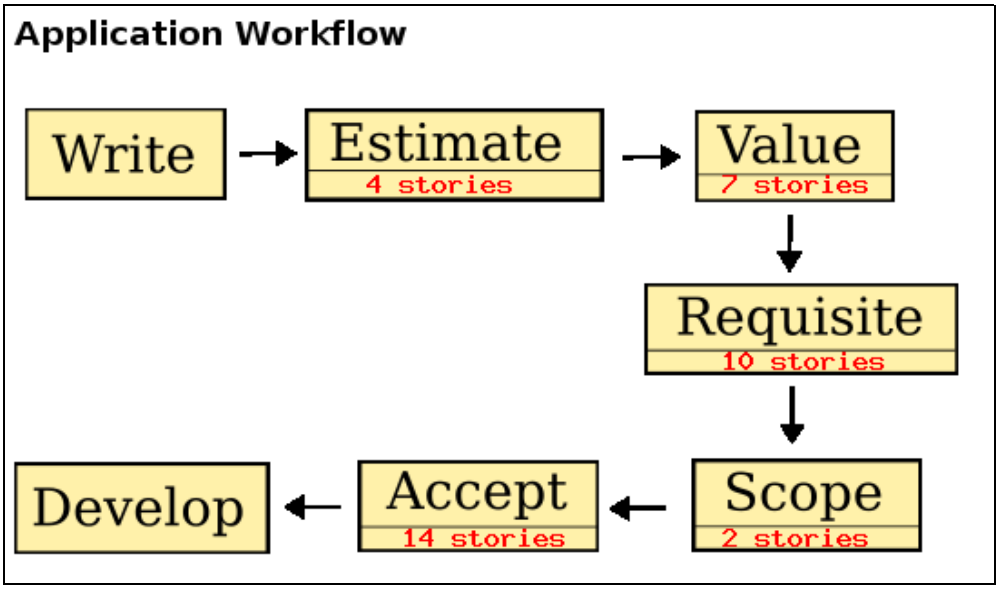


Figura 42 -*Workflow* da aplicação

8.4.3 Rastreabilidade

Para [Lee, 2003], a habilidade de rastrear através de artefatos como código, testes de aceitação, requisitos e *design* é fator crítico para o sucesso de projetos complexos.

Recapitulando o que foi dito no capítulo sobre Rastreabilidade: para [Breitman, 2002], "os requisitos do sistema são importantes artefatos e devem estar disponíveis mesmo após o sistema desenvolvido e entregue para garantir a rastreabilidade.

Pensando nisto, na presente ferramenta foi desenvolvido um *log* de operações, onde as principais operações no sistema armazenam informações vitais como : quem realizou a operação, quando a operação foi realizada, uma descrição do que foi realizado, um link para o estado atual da *story* e também uma "fotografia" da *story* no momento anterior ao que a operação fora realizada. Temos alguns exemplos de operações que são armazenadas: Escrever a *story*, definir o valor de negócio, definir os pré-requisitos, definir o escopo, aceitar a *story*, dentre outros. Veja abaixo a tela que permite consultar o *log* de rastreabilidade.

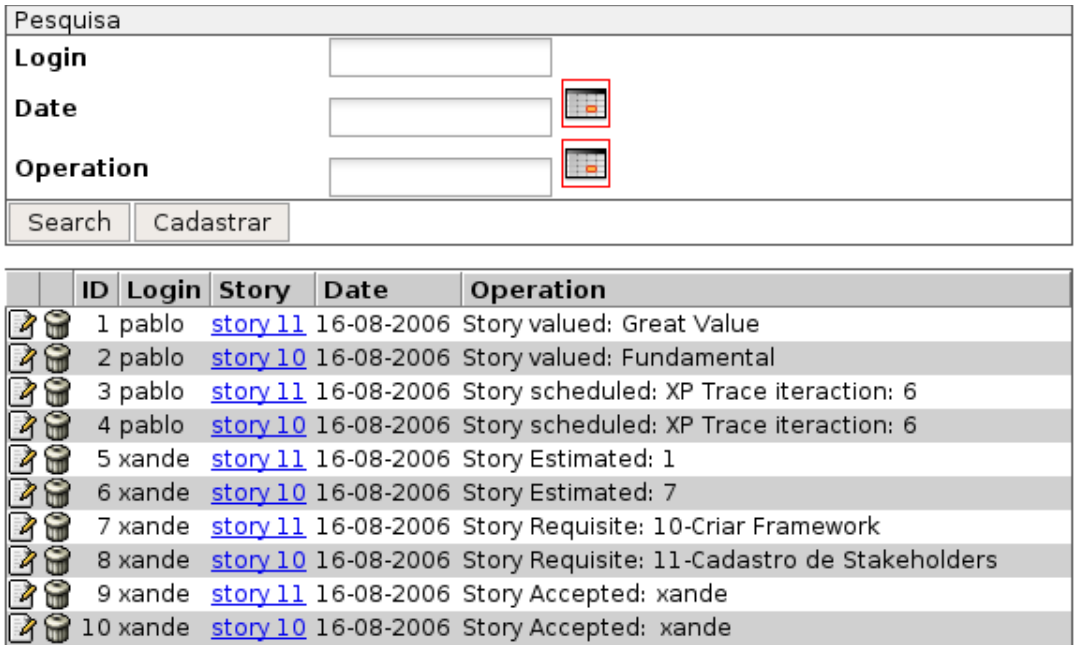


Figura 43 -Log de rastreabilidade

Aspectos de Rastreabilidade Implementadas:

- **Seletiva**: "*restringe a rastreabilidade à certos padrões de objetos e seus relacionamentos, contexto ou fase do desenvolvimento*". Neste caso, permitindo realizar filtro sobre o registro de *logs* de rastreabilidade deixado no sistema.

Para [Lee, 2003], enquanto que as conversações são dificilmente capturadas por causa de sua natureza ad-hoc, as ferramentas devem prover mecanismos para armazenar requisitos não-estruturados em um meio mais apropriado. Para ele, recursos como navegação entre os requisitos e anotações que permitam um caminho de duas mãos entre os requisitos e as decisões de *design*, são fundamentais.

Para aumentar o grau de rastreabilidade, [Leite, 2001] sugere a utilização de "Name Spaces", técnica que visa mapear o vocabulário dos negócios a fim de que se utilize os mesmos termos nas *stories*, assim como no código. A utilização de roteiros reforça a necessidade de destacar a nomenclatura presente no vocabulário léxico. Neste contexto, a ferramenta desenvolvida, permite livremente a utilização da linguagem de marcação HTML para destacar texto dentro dos seus campos descritivos da *story*.

Por definição, um vocabulário léxico, deve ser restrito à utilização de termos que representem um conjunto mínimo das palavras mais utilizadas em linguagem natural, assim como maximizar à utilização de símbolos e outros descritores que garantam a conectividade por hipertexto [Leite, 2001]. Neste contexto, a presente ferramenta permite a utilização de dois conectores especiais entre os requisitos de software: o conector <story> que cria um link para determinada *story* e o conector <resource> que cria um link para determinado recurso, que pode ser um documento PDF com a descrição da arquitetura ou uma imagem PNG contendo um diagrama de classes.

A figura a seguir, representa uma seção da aplicação chamada de "Desktop do Desenvolvedor". Neste painél, serão exibidas as *stories* que participa o desenvolvedor identificado pelo login. As *stories* são exibidas de forma sucinta e conectadas através de hiperlinks entre elas e entre seus artefatos.

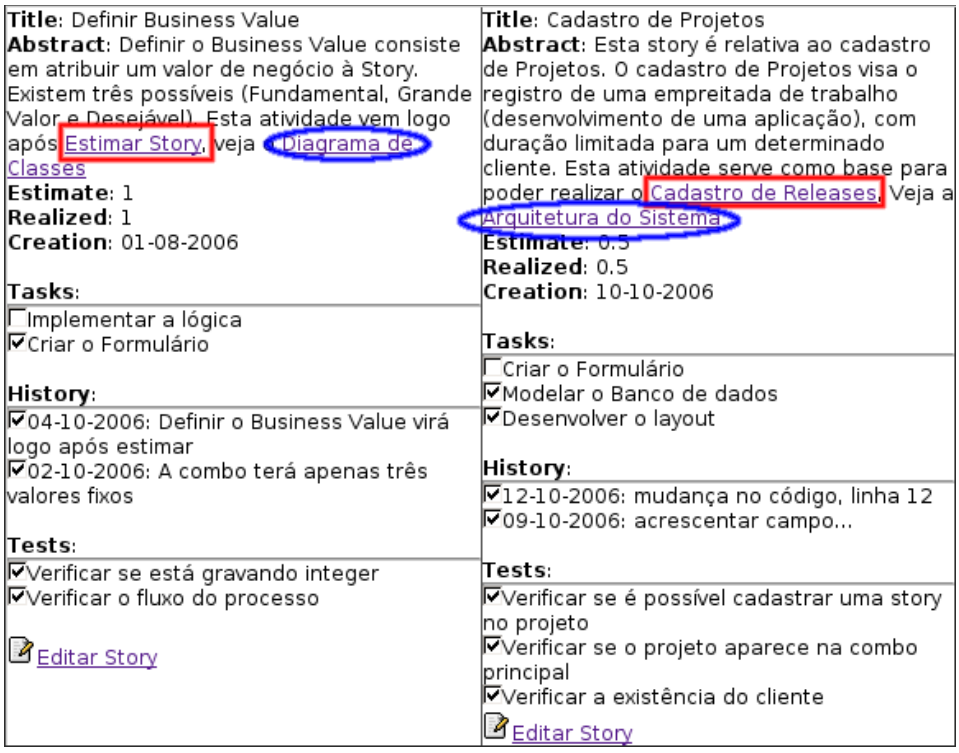


Figura 44 -Desktop do desenvolvedor

Tipos de Rastreabilidade Implementadas:

- **Para trás:** "*permite que se chegue até as origens do requisito (pessoas, processos, documentos), a partir dos requisitos"*. Implementada através do conector <resource>.
- **Intra-requisitos:** "*consiste nos relacionamentos entre os requisitos*". Implementada através do conector <story>.
- **Extra-requisitos:** "*consiste nos relacionamentos entre os requisitos e outros artefatos*." Implementada através do conector <resource>.

Aspectos de Rastreabilidade Implementadas:

- **Interativa**: "*permite navegar entre os objetos relacionados, guiado pelos seus relacionamentos*". Implementada através do conector <story> e <resource>.

Abaixo a descrição real das *stories* acima, utilizando as *tags* ou conectores criados especialmente para realizar as ligações entre requisitos (destacado através de um retângulo) e entre estes e seus artefatos ou recursos (destacado através de uma elipse).

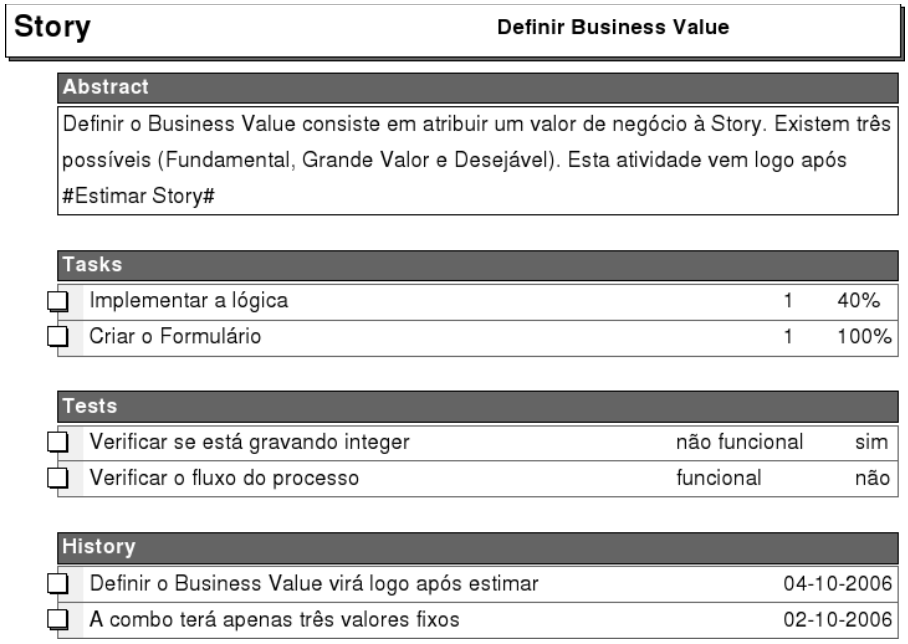
- **Story "Definir Business Value"**: "*Definir o Business Value consiste em atribuir um valor de negócio à Story. Existem três possíveis (Fundamental, Grande Valor e Desejável). Esta atividade vem logo após* ***<story>****Estimar Story****</story>****, veja o* ***<resource>****Diagrama de Classes****</resource>****"*.
- **Story "Cadastro de Projetos"**: "*Esta story é relativa ao cadastro de Projetos. O cadastro de Projetos visa o registro de uma empreitada de trabalho (desenvolvimento de uma aplicação), com duração limitada para um determinado cliente. Esta atividade serve como base para poder realizar o* ***<story>****Cadastro de Releases****</story>****, Veja a* ***<resource>****Arquitetura do Sistema****</resource>****"*.

8.5 Relatórios e Métricas

8.5.1 Imprimir Stories

A principal mídia utilizada para armazenar as *stories* em projetos XP são cartões de papel. [Beck, 2000] ainda ressalta a liberdade de se realizar anotações nos cartões, bem como a facilidade de se passar um cartão de um desenvolvedor para outro, para ser implementado. [Beck, 2000] recomenda que se permita a fácil impressão dos cartões caso venha a se implementar alguma solução baseada em software para tal. E também que essa não gere trabalho adicional que venha a comprometer a agilidade da equipe.

A funcionalidade "Imprimir Stories" gera um relatório em PDF contendo o relato de todas as *stories* de um projeto, seu resumo, suas tarefas, testes de aceitação e histórico de atividades.



Story — **Definir Business Value**

Abstract

Definir o Business Value consiste em atribuir um valor de negócio à Story. Existem três possíveis (Fundamental, Grande Valor e Desejável). Esta atividade vem logo após #Estimar Story#

Tasks

☐ Implementar a lógica	1	40%
☐ Criar o Formulário	1	100%

Tests

☐ Verificar se está gravando integer	não funcional	sim
☐ Verificar o fluxo do processo	funcional	não

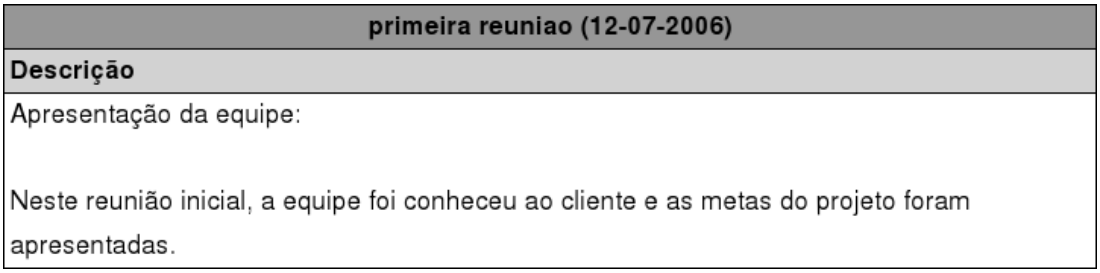
History

☐ Definir o Business Value virá logo após estimar	04-10-2006
☐ A combo terá apenas três valores fixos	02-10-2006

Figura 45 -*Story* impressa

8.5.2 Imprimir Meetings

A funcionlidade de Imprimir os "*Stand Up Meetings*" permite gerar um relatório impresso com todas as decisões de projeto tomadas nas reuniões matinais que antecedem um dia de trabalho. Assim como os "*Stand Up Meetings*", é possível também imprimir os eventos de um projeto, que podem ser frutos de reuniões com o cliente ou mesmo comunicados institucionais, dentre outras fontes de informação que se queira manter o registro no sistema.



primeira reuniao (12-07-2006)
Descrição
Apresentação da equipe: Neste reunião inicial, a equipe foi conheceu ao cliente e as metas do projeto foram apresentadas.

Figura 46 -Imprimir *meetings*

8.5.3 Release Status

O “Release Status” ou relatório do progresso das *stories*, mostra uma “fotografia” do atual estado do *release*, indicando o quanto das *stories* estão concluídas e permite ao gerente ter uma visão de quais são aquelas *stories* que serão críticas, merecendo uma atenção especial.

Sexta Semana		Data 11-09-2006
	Accept Story	
	Criar o modelo de dados	100%
	Criar a interface	100%
	Choose Scope	
	Criar o modelo de dados	100%
	Criar a interface	100%
Sétima Semana		Data 18-09-2006
	Fluxo da Aplicação	
	Estudar geração da imagem com o fluxo	100%
	Implementar geração do fluxo	100%
	Implementar interação com o usuário	100%
Oitava Semana		Data 25-09-2006
	Criar Grafo de Precedência	
	Estudar biblioteca para renderização	100%
	Criar interface	70%
	Implementar	40%

Figura 47 -*Release status*

8.5.4 Velocity

De acordo com o que foi visto no capítulo 4, *Velocity* é uma métrica que repesenta: “*A proporção entre o tempo estimado de desenvolvimento e o tempo em dias úteis decorridos no calendário para conclusão de tal tarefa*”. Neste relatório temos uma visão do *release* iteração à iteração, demonstrando uma visão que permite ao gerente perceber quais são as *stories* críticas em relação ao encerramenta da iteração, uma vez que são fornecidas informações como a quantidade de dias estimados e a quantidade de dias efetivamente utilizados.

Release	Iteraction	Story	Tasks
versao 1			
	Segunda Semana		Date 14-08-2006
		Cadastro de Releases	
		Estimated *(ideal * load)*	4.67 days
		Completed *(calendar days)*	0.5 days
		Cadastro de Iterações	
		Estimated *(ideal * load)*	0.67 days
		Completed *(calendar days)* **open!**	0.5 days
		Cadastro de Encontros	
		Estimated *(ideal * load)*	0.67 days
		Completed *(calendar days)*	0.5 days
		Available	4 days
		Work Time	1.5 days

Figura 48 -Relatório de *velocity*

8.5.5 Member Load

De acordo com o que foi visto no capítulo 4, "*Member Load*" é uma "*forma de medir o quanto em dias ideias de trabalho, determinado programador aceitou*". Desta forma, atuando como ferramenta gerencial para monitorar a carga de trabalho das pessoas envolvidas no projeto, evitando excessos que são prejudiciais para o andamento do projeto, de acordo com a metodologia. Neste gráfico, é indicada a quantidade de dias ideias aceitos por cada desenvolvedor em cada iteração.

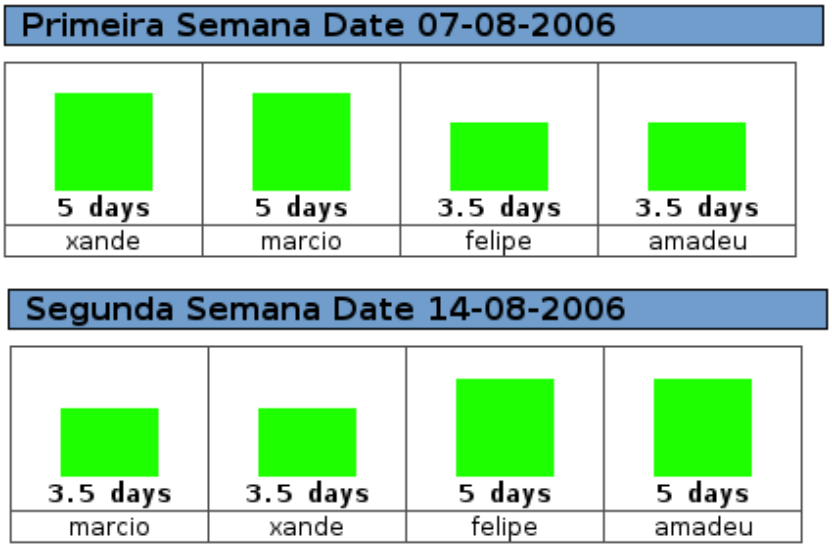


Figura 49 -Gráfico de *member load*

8.6 Interoperabilidade

A forma que iremos dispor para prover interoperabilidade para a aplicação sendo desenvolvida é através da utilização de Web Services. Web Services são serviços disponibilizados pela internet, através de um conjunto de tecnologias independentes de plataforma, que permite interoperabilidade através da entrega de serviços [Vaughan, 2002] e a comunicação entre aplicações modularizadas, que podem ser descritas, publicadas e invocadas pela internet para utilização imediata, ou mesmo composição de novos serviços [Hansen, 2002] intregrando tudo através de padrões abertos e conhecidos como XML, SOAP, WSDL e UDDI [Chung, 2003].

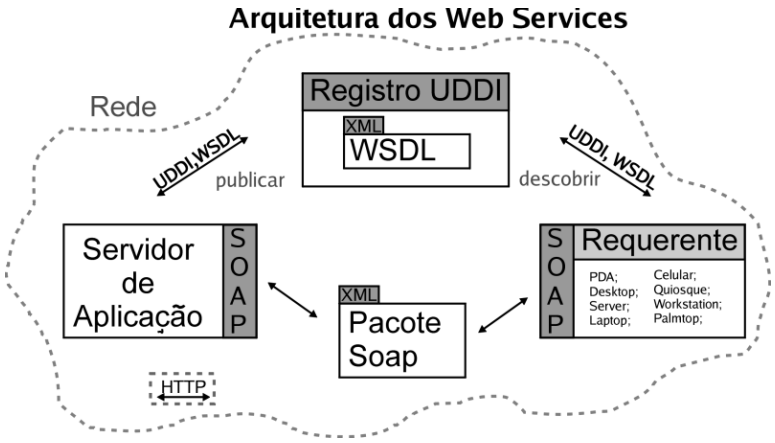


Figura 50 -Arquitetura dos *web services* [Vaughan, 2002]

Através da figura acima, temos um panorama geral do funcionamento de Web Services, através de um conjunto de tecnologias de padrão aberto, interagindo sob uma plataforma de internet. No meio de todo o processo, provendo a comunicação entre as aplicações, está o protocolo SOAP (*Simple Object Access Protocol*). SOAP é um protocolo herdeiro do padrão XML que encapsula um conjunto de regras para descrição de dados e processos, através de um mecanismo simples para definir a semântica de uma aplicação através de um modelo de empacotamento e um mecanismo de codificação [Chavda, 2004]. É projetado para a troca de informações em um ambiente descentralizado [Hansen, 2002] através do protocolo de comunicação HTTP e do formato XML [Vaughan, 2002]. Dessa forma, para uma aplicação trabalhar com WS, basta a compatibilidade com SOAP, tanto no lado do cliente (criando o documento XML com a informação necessária para invocar o serviço) quanto no lado do servidor (responsável por executar a mensagem como um interpretador) [Hansen, 2002].

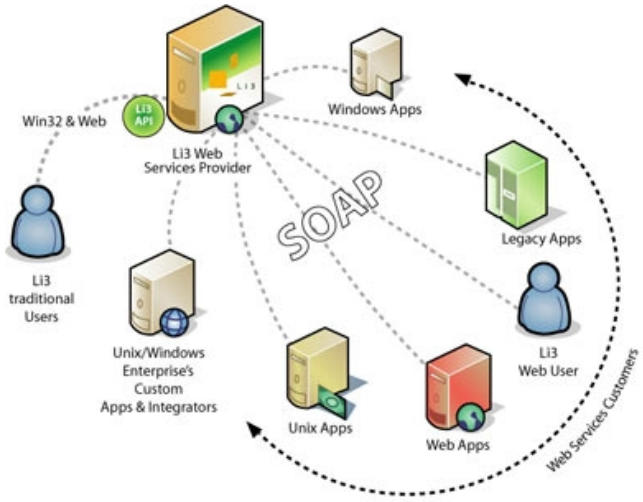


Figura 51 -Ilutração do funcionamento do protocolo SOAP

8.6.1 Introdução

Como trabalhos relacionados, encontramos a proposta de [Breitman, 2002], que envolve a implementação de seu *framework* através de um DTD descrevendo os componentes do *framework*, bem como arquivos XML, que podem ser instanciados contendo os componentes do *framework*. O link com o código pode ser feito através de um hiperlink, que pode inclusive apontar para a versão exata do código, disponível on-line através de um sistema de controle de versões. Abaixo a proposta do autor:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE Scenario>
<Scenario number="1275">
    <title>Calculate COLA rate</title>
    <goal>provide the COLA values</goal>
    <context>Blw pay period</context>
    <priority type=>high</priority>
    <risk>-----</risk>
    <actor> main frame program, plant,
    </actor>
    <resources>OLD COLA rate, NEW COLA rate</resources>
    <episodes>
```

```
        1. when the COLA rate changes in the middle of ...
        2. pay 2nd. week of the Pay Period at the NEW.
        3. run a main frame program that will pay or calc...
        4. plant retransmits the hours data for the 2nd week exclusively
        5. calculate COLA as a "2144" COLA gross pay adjustment.
        6. create RM Boundary.
        7. place in DEEntExcessCOLA BIN.
    </episodes>
    <trace type="homogeneous">
        <description>table containing the
        COLA rates</description>
        <link>
            http://www.xr.com/colarates.xml
        </link>
    </trace>
    <tag requestedBy="Karin"
        writtenBy="Otavio"
        authorizedBy="Julio">
        <date day="19" month="3" year="1998">03/19/1998</date>
        <activityType type="new">new</activityType>
        <version>02</version>
        <rationale>
            <justification>
                This procedure is the result of a new
                company policy regarding
                the calculations of the
                COLA rates.
            </justification>
            <link>http://www.abc.com/memorandum67.xml</link>
        </rationale>
    </tag>
</scenario>
```

Para [Breitman, 2002], a utilização de XML traz ganhos, uma vez que pode-se tirar maior proveito das tecnologias da internet, ao passo que se realiza verificações de consistência entre cenários.

8.6.2 Descritor WSDL

Para descrever um Web Service, é utilizada a linguagem WSDL (*Web Services Description Language*), baseada no formato XML [Vaughan, 2002]. A descrição se dá através da definição das interfaces e os mecanismos de interação, contendo informações como o protocolo, formato de dados, segurança, dentre outros [Hansen, 2002]. O WSDL descreve um conjunto de mensagens SOAP e fornece informações necessária para os clientes interagirem com ele [Chavda, 2004]. WSDL especifica a localização do WebService, as operações que estão disponíveis, os tipos de dados intercambiados e os protocolos de comunicação que serão utilizados. Abaixo o WSDL utilizado no presente trabalho, ele provê um método chamado getStory(), responsavel por retornar os dados de ums *story*, bem como de seus objetos agregados.

```
<?xml version ='1.0' encoding ='ISO-8859-1' ?>
<definitions name=''
  targetNamespace='http://example.org/'
  xmlns:tns='http://example.org/'
  xmlns:soap='http://schemas.xmlsoap.org/wsdl/soap/'
  xmlns:xsd='http://www.w3.org/2001/XMLSchema'
  xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
  xmlns:wsdl='http://schemas.xmlsoap.org/wsdl/'
  xmlns='http://schemas.xmlsoap.org/wsdl/'>

<message name='getStoryRequest'>
  <part name='codigo' type='xsd:string'/>
</message>
<message name='getStoryResponse'>
  <part name='resultado' type='xsd:string'/>
</message>

<portType name='PortType'>
```

```
  <operation name='getStory'>
    <input message='tns:getStoryRequest'/>
    <output message='tns:getStoryResponse'/>
  </operation>
</portType>

<binding name='Binding' type='tns:PortType'>
  <soap:binding style='rpc'
    transport='http://schemas.xmlsoap.org/soap/http'/>
  <operation name='getStory'>
    <soap:operation soapAction='xptrace#getStory'/>
    <input>
      <soap:body use='encoded' namespace='xptrace'
        encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'/>
    </input>
    <output>
      <soap:body use='encoded' namespace='xptrace'
        encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'/>
    </output>
  </operation>
</binding>

<service name='Service'>
  <port name='Port' binding='Binding'>
    <soap:address location='http://www.xp.local/servidor.php'/>
  </port>
</service>
</definitions>
```

8.6.3 Servidor SOAP

O servidor de aplicação provê o serviço e se comunica com o requerente através da camada de pacotes SOAP. A aplicação servidora processa a requisição enviada pelo cliente (codificada em XML) de acordo com suas regras internas de negócio e retorna ao cliente a resposta também através de um pacote XML pelo protocolo SOAP. [Chavda, 2004]

Para a aplicação desenvolvida no presente trabalho, iremos prover o método getStory(). Primeiramente iremos verificar a consistência na passagem dos parâmetros. Em um segundo passo, iremos obter as classes do modelo de dados. Em cada caso, iremos obter os objetos relacionados à *story* passada como parâmetro através do método getObject(), passando o ID do objeto a ser obtido. Em uma terceira etapa iremos obter alguns objetos agregados através do método getObjects(), que recebe um filtro para critério de seleção. Por fim, iremos construir um vetor que irá ser traduzido em um documento XML seguindo a mesma estrutura definida pelos seus índices associativos. Esta tradução será realizada pela classe TxmlArray.

O serviço é configurado através da utilização da classe “SoapServer”, nativa do PHP, que provê o método addFunction(), que habilita um determinado método a responder por requisições de clientes.

```
<?php
function getStory($codigo)
{
    // verifica a passagem do parâmetro
    if (!$codigo)
    {
        throw new SoapFault('Client','Parametro nao preenchido');
    }

    // carrega bibliotecas necessárias
    include_once('loadlibraries.php');

    // obtém modelos
    $StoriesModel     = TMaster::getModel('Stories');
    $IteractionsModel = TMaster::getModel('Iteractions');
```

```
$ReleasesModel    = TMaster::getModel('Releases');
$ProjectsModel    = TMaster::getModel('Projects');
$StakeholdersModel= TMaster::getModel('Stakeholders');
$TasksModel       = TMaster::getModel('Tasks');
$TestsModel       = TMaster::getModel('Tests');
$HistoryModel     = TMaster::getModel('History');

// obtém objetos
$story      = $StoriesModel->getObject($codigo);
$iteraction = $IteractionsModel->getObject($story->iteraction);
$release    = $ReleasesModel->getObject($iteraction->release);
$project    = $ProjectsModel->getObject($story->project);
$customer   = $StakeholdersModel->getObject($project->customer);

// obtém objetos agregados
$filter->story = array('=', $codigo);
$tasks  = $TasksModel->getObjects($filter);
$tests  = $TestsModel->getObjects($filter);
$history= $HistoryModel->getObjects($filter);

// compõe array com os dados dos objetos
$return['Story']['id']          = $story->id;
$return['Story']['title']       = $story->title;
$return['Story']['abstract']    = $story->abstract;
$return['Story']['estimate']    = $story->estimate;
$return['Story']['realized']    = $story->realized;
$return['Story']['status']      = $story->status;
$return['Story']['creation']    = $story->creation;
$return['Story']['close']       = $story->close;
$return['Story']['pair']        = $story->pair;
$return['Story']['requisite']   = $story->requisite;
$return['Story']['value']       = $story->value;
$return['Story']['iteraction']['code']      = $story->iteraction;
$return['Story']['iteraction']['name']      = $iteraction->name;
$return['Story']['iteraction']['date']      = $iteraction->date;
$return['Story']['iteraction']['pairs']     = $iteraction->pairs;
$return['Story']['iteraction']['days']      = $iteraction->days;
$return['Story']['iteraction']['velocity']  = $iteraction->velocity;
$return['Story']['release']['code']         = $iteraction->release;
$return['Story']['release']['name']         = $release->name;
$return['Story']['project']['code']         = $story->project;
$return['Story']['project']['name']         = $project->name;
$return['Story']['project']['view']         = $project->view;
$return['Story']['project']['scope']        = $project->scope;
$return['Story']['project']['requirements'] = $project->requirements;
$return['Story']['project']['customer']['code'] = $project->customer;
$return['Story']['project']['customer']['name'] = $customer->name;

$i=0;
// percorre as tarefas da story e armazena no array
foreach ($tasks as $task)
{
    $i++;
    $return['Story']['tasks']["task$i"]['code'] = $task->id;
    $return['Story']['tasks']["task$i"]['description'] = $task->description;
}

$i=0;
// percorre os testes da story e armazena no array
foreach ($tests as $test)
{
    $i++;
    $return['Story']['tests']["test$i"]['code'] = $test->id;
    $return['Story']['tests']["test$i"]['description'] = $test->description;
}

$i=0;
// percorre o histórico da story e armazena no array
foreach ($history as $hist)
{
    $i++;
    $return['Story']['history']["history$i"]['code'] = $hist->id;
    $return['Story']['history']["history$i"]['comment'] = $hist->comment;
    $return['Story']['history']["history$i"]['date'] = $hist->date;
}
```

```
    // converte array para XML e retorna para o SOAP client
    return TXmlArray::Array2Xml($return);
}

// instancia servidor SOAP
$server = new SoapServer("exemplo.wsdl", array('encoding'=>'ISO-8859-1'));
$server->addFunction("getStory");
$server->handle();
?>
```

8.6.4 Cliente SOAP

O cliente ou requerente é a aplicação que interage com o Servico [Hansen, 2002], podendo ser desde um desktop ou servidor, até um celular ou palmtop. A aplicação cliente envia para aplicação servidora um pacote XML através do protocolo SOAP, para que este pacote seja processado e posteriorment devolvido com o resultado.

Neste caso, iremos simplemente criar um cliente que irá acionar o serviço prestado pelo servidor criado acima. Através da utilização da classe "SoapClient", que disponibiliza a API da aplicação servidora. Neste caso, estamos buscando a *story* 12 e imprimindo seu XML na tela. Caso ocorra alguma exceção, esta será impressa na tela através do controle de erros try/catch.

```
<?php
// instancia cliente SOAP
$client = new SoapClient("exemplo.wsdl", array('encoding'=>'ISO-8859-1'));
try
{
    // realiza chamada remota de método
    $retorno = $client->getStory(12);

    // Imprime os dados de retorno
    echo '<pre>';
    echo htmlspecialchars($retorno);
    echo '</pre>';
}
catch (SoapFault $excecao) // ocorrência de erro
{
    echo "Erro: ";
    echo "<b> {$excecao->faultstring} </b>";
}
?>
```

8.6.5 Documento XML

Este é o resultado da execução da aplicação cliente listada acima. Temos um documento XML contendo toda descrição da *story*. Primeiramente vemos os atributos únicos da *story* em si, como seu título, resumo, estimativa, tempo realizado, data de criação, data de conclusão, par de programadores, pré-requisitos e valor de negócio. Posteriormente temos a iteração, o *release* e o projeto a qual ela pertence, bem como alguns dados descritivos. Ao final, temos as tarefas nas quais a *story* foi dividida (*tasks*), os seus testes de aceitação (*tests*), bem como o seu histórico de alterações (*history*).

```
<Story>
    <id>12</id>
    <title>Cadastro de Projetos</title>
    <abstract>Esta story é relativa ao cadastro de Projetos.
O cadastro de Projetos visa o registro de uma empreitada de trabalho (desenvolvimento de
uma aplicação), com duração limitada para um determinado cliente.</abstract>

    <estimate>0.5</estimate>
    <realized>0.5</realized>
    <status>open</status>
    <creation>10-10-2006</creation>
    <close>24-10-2006</close>
    <pair>xande#marcio</pair>
```

```
    <requisite>11</requisite>
    <value>1</value>
    <iteraction>
        <code>6</code>
        <name>Primeira Semana</name>
        <date>07-08-2006</date>
        <pairs>1</pairs>
        <days>4</days>
        <velocity>3</velocity>
    </iteraction>
    <release>
        <code>3</code>
        <name>versao 1</name>
    </release>
    <project>
        <code>2</code>
        <name>XP Trace</name>
          <view>Este projeto visa criar uma infra-estrutura que registre os requisitos
gerados em um projeto Extreme Programming, bem como gere informações gerenciais a partir
destes dados gerados naturalmente.</view>
           <scope>O projeto visa criar estruturas para armazenar projetos, releases,
iterações, reuniões, eventos, recursos externos, stores, além de gerar os relatórios que
forneçam as métricas mais comuns utilizadas na metodologia.</scope>
         <requirements>Para tal, será necessário utilizarmos PHP, Apache, Banco de dados
SQLite e a biblioteca SOAP, disponível no PHP.</requirements>
        <customer>
            <code>2</code>
            <name>Candido Fonseca da Silva</name>
        </customer>
    </project>
    <tasks>
        <task1>
            <code>146</code>
            <description>Criar o Formulário</description>
        </task1>
        <task2>
            <code>145</code>
            <description>Modelar o Banco de dados</description>
        </task2>
    </tasks>
    <tests>
        <test1>
            <code>21</code>
            <description>Verificar se é possível cadastrar uma ...</description>
        </test1>
        <test2>
            <code>20</code>
            <description>Verificar se o projeto aparece na combo principal</description>
        </test2>
    </tests>
    <history>
        <history1>
            <code>26</code>
            <comment>mudança no código, linha 12</comment>
            <date>12-10-2006</date>
        </history1>
        <history2>
            <code>25</code>
            <comment>acrescentar campo...</comment>
            <date>09-10-2006</date>
        </history2>
    </history>
</Story>
```

9 Conclusão

O gerenciamento de projetos de software, em muitos casos pode se tornar uma tarefa mais complexa do que o próprio desenvolvimento em si. Na maioria das vezes a dificuldade reside em manter o registro dos requisitos funcionais, bem como as decisões de projeto em sintonia com a ferramenta que decide-se utilizar para prover tal gerenciamento, seja esta ferramenta eletrônica ou não.

O gerenciamento do processo de software pode ser facilitado por ferramentas que utilizem em maior grau as informações produzida naturalmente em um projeto, evitando introduzir novas práticas que visem a produção de formalismos que diminuem a agilidade do processo.

Extreme Programming é uma metodologia que imprime uma grande agilidade no desenvolvimento de software através de práticas que devem ser utilizadas em conjunto. Durante o seu ciclo de desenvolvimento, a metodologia produz naturalmente uma série de informações que podem ser utilizadas para prover informações gerenciais, como as *stories* ou mesmo os jogos de planejamento.

Como proposta do presente trabalho, o desenvolvimento de uma ferramenta para gerenciar os requisitos de software em um projeto *Extreme Programming* deveria levar em consideração todos estes fatores, para tal foi realizado um extenso trabalho de revisão bibliográfica para que evitasse construir algo que iria contra os princípios pregados.

Como resultado, acreditamos que todas as metas foram cumpridas através do desenvolvimento de uma ferramenta simples e que provê ao mesmo tempo informações básicas para acompanhamento dos requisitos, bem como acompanhamento das operações de acordo com os princípios da rastreabilidade e também gráficos gerenciais que proporcionam uma visão ampla da atual situação de um projeto.

Esta ferramenta, fruto do presente trabalho fora utilizada com sucesso em dois projetos, o primeiro, o desenvolvimento da própria ferramenta, monitorando os trabalhos desenvolvidos iteração após iteração até que se chegasse à conclusão do presente trabalho, produzindo relatórios e gráficos que representam o esforço desempenhado para a produção do mesmo. A ferramenta também foi utilizada com sucesso em um outro projeto profissional no qual o gerenciamento da equipe se deu de forma remota, mostrando também bons resultados que possibilitaram maior compreensão do planejamento de trabalho.

Pessoalmente o desenvolvimento de tal ferramenta foi de grande valia face ao grande aprendizado proporcionado pelos conceitos que tiveram de ser compreendidos a partir desta metodologia que tem crescido em adoção a cada dia.

10 Trabalhos Futuros

Durante o desenvolvimento do presente trabalho, a área de gerenciamento de projetos se mostrou de grande riqueza e inúmeras idéias sobre possíveis recursos foram surgindo. Devido ao tempo restrito para execução do presente trabalho, procurou-se manter o foco naqueles recursos que demonstravam de forma mais clara e explícita a utilização dos conceitos da metodologia. A partir disto, prentende-se dar continuidade ao presente trabalho, implementando todos recursos que foram vislumbrados no sentido de melhorar a produtividade e prover maiores informações gerenciais, dentre eles:

- Aumentar o grau de interoperabilidade com outras aplicações, exportando a informação armazenada na atual aplicação para formatos de ferramentas reconhecidas como reqPro, Rational Rose, dentre outras;
- Aumentar o número de métodos oferecidos pela API Web Service, tornando a interface mais rica e dando maior controle à aplicações externas que possam manipulá-la;
- Criar uma aplicação cliente para registrar o número de horas trabalhadas pelo programador, possibilitando assim um controle mais preciso para o gerente da equipe e diminuindo também o trabalho que existe atualmente para o registro das operações realizadas;

Bibliografia

[1] AMBLER, Scott. **Agile Modeling: Effective Practices for Extreme Programming and the Unified Process**. 224 pages. Wiley. 1st edition (March 22, 2002).

[2] BECK, Kent. FOWLER, Martin. **Planning Extreme Programming**. 160 pages. Addison-Wesley Professional; 1st edition (October 13, 2000)

[3] BECK, Kent. ANDRES, Cynthia. **Extreme Programming Explained** : Embrace Change (2nd Edition). 224 pages. Publisher: Addison-Wesley Professional; 2 edition (November 16, 2004)

[4] BREITMAN, K. K. ; LEITE, J. C. S. P. . **Managing User Stories**. In: TCRE Workshop, 2002, Essen, Alemanha. Proceedings of the Time Constrained Requirements Engineering Workshop. Rio de Janeiro : Papel&Virtual, 2002. p. 49-56.

[5] CARVALHO, R. Ferreira, M. A. T. **Análise de software de gestão do conhecimento.** XXI Simpósio de gestão da inovação tecnológica. São Paulo: Nov. 2000.

[6] CHAVDA, K. F., **Anatomy of Web Service**, Journal of Computing Sciences in Colleges, Kennesaw State University, January 2004.

[7] CHUNG, J., Lin, K., Mathieu, R. **Web Services Computing: Advancing Software Interoperability**. IEEE Computer, 35-37, 2003

[8] COHN, Mike, FORD, Doris. **Introducing an Agile Process to an Organization**. IEEE Computing, 2003.

[9] DUNCAN, Richard. **The Quality of Requirements in Extreme Programming**. CrossTalk, June 2001: 19-22, 31.

[10] FOWLER, Martin. **Is design dead?** XP2000 Conference. Disponível em: <http://www.martinfowler.com>, Acesso em: Abril de 2006.

[11] FOWLER, Martin. **New Methodology.** XP2000 Conference. Disponível em: <http://www.martinfowler.com>, Acesso em: Abril de 2006.

[12] GAMMA, Erich, HELM, Richard, JOHNSON, Ralph, VLISSIDES, John. **Design patterns: elements of reusable object-oriented software**, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1995.

[13] HANSEN, R. Santos, C. Crespo, S. Lanius, G. Massen, F. **Web Services: An Architecture Overview**. First Seminar on Advanced Research in Electronic Business, Rio de Janeiro, Brasil, 2002.

[14] HIGHSMITH, Jim. COCKBURN, Alistair. **Agile Software Development: The Business of Inovation**. Prepared by the IEEE Computer Society/ACM joint task force. 2001.

[15] HJERLING, Johan; LJUNGQVIST, Patrik, **Capability Maturity Model Integration (CMMI) and Agile Methods (XP)** - A course paper. 2004.

[16] HORRIAN, Hossein. MAHMUD, Shafquat. KARTHIKEYAN, Srinivasan. **Requirements Engineering in Agile methods.** Dept. of Computer Science, University of Calgary, Canada, 2003.

[17] HUO, Ming. VERNER, June. ZHU, Liming. BABAR, Muhammad Ali. **Software Quality and Agile Methods.** Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04) 0730-3157/04 © 2004 IEEE

[18] KAINDL, H. ; Brinkkemper, S. ; Bubenko, J. ; Farbey, B ; Greenspan, S ; Heitmeyer, C. ; LEITE, J. C. S. P. ; Mead, N. ; Mylopoulos, J ; Siddiqi . **Requirements Engineering and Technology Transfer: Obstacles, Incentives and Improvement Agenda**. Requirements Engineering Journal, Springer Verlag, Londres, v. 7, n. 3, p. 113-123, 2002.

[19] LEE, Christopher. Guadagno, Luigi. Jia, Xiaoping. **An Agile Approach to Capturing Requirements and Traceability**, Proc. of the 2nd International Workshop on Traceability in Emerging Forms of Software Engineering, October 7, 2003, Montreal, Canada, pp.17-23.

[20] LEITE, Júlio Cesar Sampaio do Prado. **Gerenciando a qualidade de software com base em requisitos**. In: Ana Regina Cavalcanti da Rocha, José Carlos Maldonado, Kival Chaves Weber. (Org.). Qualidade de Software Teoria e Prática. 1 ed. São Paulo: Prentice - Hall, 2001, v. 1, p. 238-246.

[21] LEITE, Júlio Cesar Sampaio do Prado. **Extreme Requirements**. In: Jornadas de Ingeniería de Requisitos Aplicada, 2001, Sevilha. Jornadas de Ingeniería de Requisitos Aplicada, 2001. p. 1-13.

[22] LEITE, J. C. S. P. ; GLADYS KAPLAN, GRACIELA HADAD JORGE HORACIO DOORN . **A Scenario Construction Process**. Requirements Engineering, Springer Verlag -- Londres, v. 5, n. 1, p. 38-61, 2000.

[23] LUBARS, Mitch. POTTS, Colin, RICHTER, Charlie. **A Review of the State of the Practice in Requirements Modeling**. Proc. IEEE Int'l Symp. Requirements Eng., IEEE Computer Soc. Press, Los Alamitos, Calif., 1993, pp. 2-14.

[24] MATOS, Alexandre Veloso de. **UML: prático e descomplicado**. São Paulo: Érica, 2002.

[25] MILLER, Randy. **The Dynamics of Agile Software Processes**. Disponível em: <http://www.borland.com>, acesso em: Abril de 2006.

[26] NAWROCKI, Jerzy. JASIŃSKI, Michal. WALTER, Bartosz. WOJCIECHOWSKI, Adam. **Extreme Programming Modified.** IEEE Joint International Conference on Requirements Engineering. 2002.

[27] NUSEIBEH, Bashar. EASTERBROOK, Steve. **Requirements Engineering: A Roadmap**. Proceedings of International Conference on Software Engineering (ICSE-2000), 4-11 June 2000, Limerick, Ireland.

[28] PAETSCH, Frauke. EBERLEIN, Armin. MAURER, Frank. **Requirements Engineering and Agile Software Development**. Proceedings of the Twelfth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'03). 2003 IEEE.

[29] PAULK, M. C. **Extreme Programming from a CMM Perspective**. IEEE Software, 18(6):19-26, Nov./Dec. 2001.

[30] PINHEIRO, Francisco. **Requirements Traceability**. Capítuo 5: Julio Cesar S. P. Leite, Jorge H. Doorn. (Org.). Perspectives on Software Requirements. Boston: Kluwer Academic Publishers, 2004, v. 1, p. 91-110.

[31] POHREN, Daniel. XPManager: **Uma Ferramenta de Gerência de Projetos Baseados em Extreme Programming**. Monografia de Graduação. Feevale, 2004.

[32] RAMESH, B. Jarke, M. . **"Towards Reference Models for Requirements Traceability"** in IEEE Transactions on Software Engineering, 27(1). 2001. pp. 58--93

[33] SCHWABER, Ken. BEEDLE, Mike. **Agile Software Development with SCRUM**. 150 p. Prentice Hall; 1st edition (October 15, 2001).

[34] TORANZO, M. Castro, J. **A Comprehensive Traceability Model to Support the Design of Interactive Systems.** Proceedings of the Workshop on Object-Oriented Technology. Springer Verlag, Londres, vol. 1743, p.283-284, 1999.

[35] VAUGHAN, Nichalos. **WEB Services: Beyond the Hype**. IEEE Computer, 35:2, 2002.

Anexo A – Avaliação da Aplicação

A ferramenta desenvolvida no presente trabalho (entitulada XP3), estará disponível no endereço: http://xp.dalloglio.net. Para acessá-la, basta a utilização de um navegador compatível com os padrões da W3C e acesso à internet.

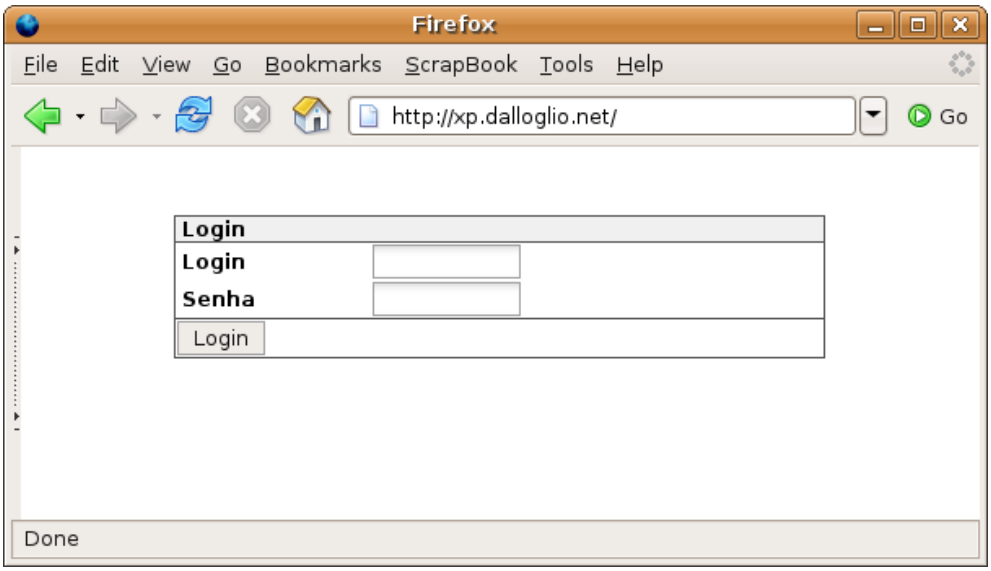


Figura A1 - Tela de login

Perfis

Ao logar na aplicação, a mesma se comportará de forma distinta a partir do perfil de usuário, exibindo opções diferentes no seu menu.

- **Gerente**: Usuário "pablo", senha "123";
  Terá acesso aos menus "Manage", "Planning" e "Reports";
- **Desenvolvedor**: Usuário "xande", senha "123";
  Terá acesso aos menus "Planning" e "Reports";
- **Cliente**: Usuário "sergio", senha "123";
  Terá acesso aos menus "Planning" e "Reports";

Obs: As opções do menu "Planning" se adaptam ao perfil do usuário selecionado. Poderá definir o valor de negócios e escolher o escopo quem for cliente ou gerente. Poderá estimar, definir pré-requisitos e aceitar a story quem for desenvolvedor ou gerente. As demais opções estão disponíveis de acordo com as macros permissões explicadas acima.

Obs: O desenvolvedor "xande" é o único dos 3 que possui stories alocadas para desenvolver, portanto somente para ele o "Desktop do desenvolvedor" terá informações.

Obs: O banco de dados possui algumas informações incompletas propositalmente para permitir uma visão geral de todas as etapas do processo.

Após o login efetuado com sucesso, é necessário escolher um projeto de trabalho, existem dois disponíveis:

- **XPTrace**: Possui registrado todo o banco de requisitos do presente trabalho;
- **Fiocruz**: Possui registrado os requisitos de um projeto a parte, desenvolvido para a Fiocruz;

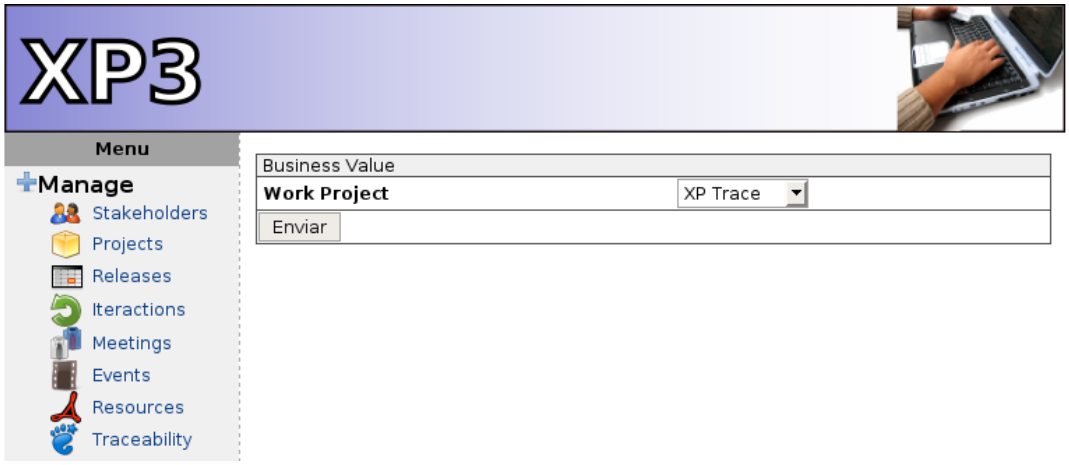


Figura A2 – Definição de projeto

Após definir o projeto de trabalho, o usuário será direcionado para a tela de *workflow*, onde poderá escolher uma etapa do processo a trabalhar ou mesmo acessar as opções do menu.

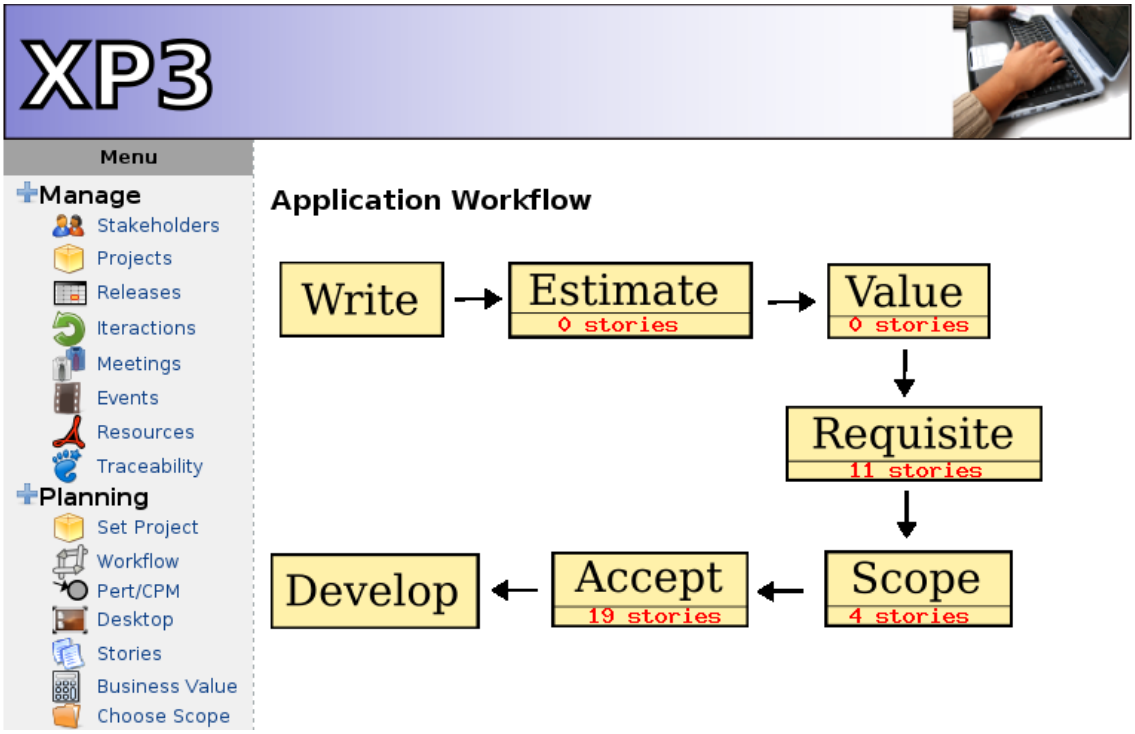


Figura A3- Apresentação do *workflow*

Anexo B – Configuração da Aplicação

Para instalar a aplicação, é necessário ter instalado as seguintes ferramentas:

- Apache2;
- PHP5 (com os seguintes módulos) :
  - libxml
  - mcrypt
  - PDO
  - pdo_sqlite
  - SimpleXML
  - soap
  - SQLite
  - xml
  - xmlreader
  - xmlwriter
  - zlib
- Descompacte na pasta httpdocs do Apache;
    - tar -xzvf xp.tar.gz
- Pronto, basta acessar pelo navegador;