FUNDAÇÃO DE ENSINO EURÍPIDES SOARES DA ROCHA
CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA – UNIVEM
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

**FRANCIENE DUARTE GOMES**

# PARFAIT/EA – PROCESSO ÁGIL DE DESENVOLVIMENTO BASEADO EM *FRAMEWORK*

MARÍLIA
2007

FRANCIENE DUARTE GOMES

# PARFAIT/EA – PROCESSO ÁGIL DE DESENVOLVIMENTO BASEADO EM *FRAMEWORK*

Dissertação apresentada ao Programa de Mestrado do Centro Universitário Eurípides de Marília, mantido pela Fundação de Ensino Eurípides Soares da Rocha, para obtenção do Título de Mestre em Ciência da Computação (Área de Concentração: Engenharia de *Software*).

Orientadora:
Prof[a]. Dr[a]. Maria Istela Cagnin Machado

MARÍLIA
2007

Aos meus pais, Francisco e Iene.

# AGRADECIMENTOS

Agradeço a Deus pela minha vida maravilhosa e por ter me dado forças para enfrentar todas as dificuldades e conseguir atingir mais um objetivo. Muito obrigada!

À minha orientadora professora Dr.ª Maria Istela Cagnin Machado por confiar em mim, pela dedicação e apoio e pelos seus ensinamentos. Muito obrigada!

Aos meus pais Francisco e Iene por sempre acreditarem em mim, pelo carinho e dedicação constante em toda a minha vida e principalmente por existirem. Muito obrigada!

Ao meu irmão Eney, pelo carinho, atenção, por acreditar em mim e pelos seus ensinamentos que me fizeram crescer profissionalmente. Muito obrigada mano!

Ao meu namorado, Edgard (Di) pela paciência, pelo carinho, pelo companheirismo, pela compreensão nos momentos em que não estive presente e pela confiança. Muito obrigada meu amor!

A minha tia querida Tânia, que sempre me ajudou com seu carinho e atenção. Muito obrigada!

Aos meus amigos Rodrigo Fraxino Araujo, Silvio Ricardo Rodrigues Sanches e Giulianna Marega Marques pela amizade, pela paciência comigo e pelos momentos de descontração únicos que jamais esquecerei. Muito obrigada!

As minhas amigas Gisele Fabiana, Eliane de Matos, Kely Mazzo e Vânia Somaio pela amizade, companheirismo, carinho e pela confiança depositada em mim. Muito obrigada!

A todos os meus amigos da turma de 2004, 2005 e 2006 pelos momentos de descontração e de tensão. Muito obrigada!

Ao corpo docente do Programa de Mestrado em Ciência de Computação – UNIVEM.

À CAPES pelo apoio financeiro.

*No final tudo dá certo. Se não deu ainda é porque não chegou o final*

***Fernando Sabino***

Gomes, Franciene, D. **Processo Ágil de Desenvolvimento Baseado em *Framework*.** 163 f. Dissertação (Mestrado em Ciência da Computação) – Centro Universitário Eurípides de Marília. Fundação de Ensino Eurípides Soares da Rocha, Marília, 2007.

## RESUMO

Atualmente, uma importante técnica de Engenharia de *Software* que está sendo freqüentemente utilizada é a reutilização de *software*, a fim de aumentar a produtividade das equipes e diminuir o custo e o tempo de desenvolvimento. Dentre os meios para praticá-la têm-se os *frameworks*, que são utilizados para a geração de aplicações. A construção de *software* apoiado por *frameworks* vem aumentando consideravelmente em decorrência da grande abrangência do mesmo, uma vez que é possível reutilizar tanto o código como também o projeto. Além disso, têm-se também os padrões de *software* como sendo mais uma forma de reutilização. O conceito de padrão de *software* é também explorado na literatura para apoiar a construção de *frameworks* durante as fases de análise e projeto, facilitando seu uso, bem como entendimento e a comunicação entre as pessoas durante o desenvolvimento de *software* gerado a partir do *framework*. Para apoiar tal desenvolvimento muitos processos são propostos. No entanto, nenhum estudo sobre processo baseado em métodos ágeis voltado para o desenvolvimento de *software* com apoio computacional de *framework* foi encontrado na literatura. Assim, é proposto neste trabalho um processo ágil de desenvolvimento de *software* apoiado por *frameworks*, tendo como base o processo ágil de reengenharia PARFAIT, o qual também é apoiado por *frameworks*, pertence ao domínio de Sistemas de Informação e representa um dos recursos do arcabouço ARA (Arcabouço Ágil de Reengenharia). Esse arcabouço foi inicialmente idealizado para ser utilizado na reengenharia de sistemas legados procedimentais para o paradigma orientado a objetos e é utilizado neste trabalho para apoiar somente a engenharia avante. Para gerenciar processos ágeis baseados em *frameworks,* a ferramenta existente P.DOCTool, utilizada apenas para documentar processos com a documentação estruturada no formato estabelecido pelo RUP (*Rational Unified Process*), foi evoluída neste trabalho para atender o planejamento e a execução de projetos de *software*. Um estudo de caso planejado de um sistema real foi conduzido para analisar e evoluir o processo de *software* proposto e utilizado como exemplo para avaliar a ferramenta.

**Palavras chaves:** Processo de Desenvolvimento, Métodos Ágeis, Padrão de *Software*, Linguagem de Padrões, *Frameworks.*

Gomes, Franciene, D. **Processo Ágil de Desenvolvimento Baseado em *Framework*.** 163 f. Dissertação (Mestrado em Ciência da Computação) – Centro Universitário Eurípides de Marília. Fundação de Ensino Eurípides Soares da Rocha, Marília, 2007.

## ABSTRACT

An important technique of Software Engineering frequently used nowadays is the software reusing, in order to increase productivity and decrease development cost and time. Frameworks, used for the generation of several applications, can be used to practice this technique. The construction of software by frameworks is increasing considerably due to its great embracement, once that it is possible to reuse the code as well as the project. Moreover, there are also the software patterns, another form of reusing. The concept of software pattern is also explored in literature supporting the construction of frameworks during the analysis and project phases, facilitating its use, as well as the communication and understading between people during the software development generated by the framework. A lot of processes are considered to support such development. However, any study based on agile methods aiming software development with computation support of framework was found in the literature. In this context, the creation of an agile process of development supported by frameworks is proposed, having the PARFAIT agile process of engineering as basis. It is supported by frameworks, belongs to the information systems domain and represents one of the ARA (Agile Reengineering Architecture) resources. This architecture was initially idealized to be used in the reengineering of procedural legacies systems to the object oriented paradigm. To manage the agile processes based on frameworks, the P.DOCTool tool, created only to document process with structured documentation in the RUP (Rational Unified Process) format, was evolved in this work to take care of planning and execution of software projects. A real system case study was conducted to analyze and evolve the software process proposed and used as an example to evaluate the tool.

**Key-words:** Development Proccess, Agile Methods, Software Patterns, Pattern Languages, Frameworks.

# LISTA DE FIGURAS

# LISTA DE QUADROS

# SUMÁRIO

# CAPÍTULO 1. INTRODUÇÃO

## 1.1 Contexto

Um dos principais objetivos da Engenharia de *Software* é o desenvolvimento de *software* com qualidade. Nesse sentido, o termo qualidade engloba maior confiabilidade, redução dos riscos durante a aplicação do processo, conformidade com padrões, por exemplo, padronização na criação de interfaces com o usuário, extensibilidade, flexibilidade, redução tanto de custos quanto de esforços, bem como o aumento da produtividade das equipes de desenvolvimento. Para atingir esses objetivos, diversos métodos, técnicas e processos são utilizados no desenvolvimento do *software*.

Dentre os métodos têm-se os ágeis que seguem princípios e práticas ágeis definidos pela Aliança Ágil[1]. Uma das características marcantes desses métodos é o desenvolvimento do *software* de modo iterativo e incremental e a disponibilização de uma versão do *software* em um tempo menor de desenvolvimento.

Os métodos tradicionais de desenvolvimento de *software* fornecem um processo um tanto burocrático e são usados, em geral, para o desenvolvimento de sistemas complexos, cujos requisitos não mudam com freqüência. Já os métodos ágeis podem ser aplicados em sistemas em que os requisitos não são estáveis. Alguns métodos ágeis são mais voltados para o desenvolvimento de sistemas pequenos, como XP (BECK, 2000) e *Scrum* ((SCHWABER e BEEDLE, 2002) **apud** (ABRAHAMSSON *et al*, 2002)), outros para o desenvolvimento de sistemas mais complexos como ASD (*Adaptive Software Development*) ((HIGHSMITH,

[1] http://www.agilealliance.com. Acesso em Outubro/2005

2000) **apud** (ABRAHAMSSON *et al*, 2002)), mas a maioria dos métodos permite adaptações para seu uso (ABRAHAMSSON *et al*, 2002).

Várias técnicas que viabilizam o reúso também vêm sendo utilizadas na Engenharia de *Software,* como *frameworks* (BOSCH *et al*, 1999; MATTSSON, 1996; PREE *et al*, 1995; TALIGENT, 1997)*,* padrões de *software* (JACOBSON *et al*, 1997; MATTSSON, 1996), linguagens de padrões (APPLETON, 1997; BRAGA, 2003; COPLIEN, 1998; SCHMIDT *et al,* 1996), componentes (SOMMERVILLE, 2003), entre outros. O uso dessas técnicas proporciona bons resultados ao processo de desenvolvimento e, conseqüentemente, ao *software*. Alguns *frameworks* são baseados em linguagens de padrões de análise, como, por exemplo, o *framework* GREN (BRAGA, 2003). O uso de linguagens de padrões de análise na construção de *frameworks* facilita o entendimento do *framework* e sua utilização, bem como a análise e a documentação do *software* por ele gerado.

Com o avanço da tecnologia e o grande interesse pela qualidade do *software*, tem-se intensificado a utilização de processo de desenvolvimento, sendo ele um dos principais fatores que colabora para o sucesso do *software* (SOMMERVILLE, 2003). Entretanto, uma carência observada no desenvolvimento de *software* é a falta de um arcabouço que forneça subsídios e apoio ao desenvolvimento do *software* ágil com o uso de *framework.*

Nesse contexto, um arcabouço denominado ARA (**A**rcabouço de **R**eengenharia **Á**gil), proposto por Cagnin (2005a), utilizado na reengenharia[2] de sistemas procedimentais para o paradigma orientado a objetos, é utilizado neste trabalho para ser aplicado somente no desenvolvimento de *software,* apoiando somente a engenharia avante[3]. Em todo o trabalho a palavra engenharia avante tem o mesmo significado que a palavra desenvolvimento de *software*.

[2] É a reconstrução de um produto, com funcionalidades adicionais, melhor desempenho, confiabilidade e manutenibilidade (PRESSMAN, 2002).
[3] Engenharia avante abrange a transferência das abstrações, modelos lógicos e projeto do sistema para uma implementação física (CHIKOFSKY e CROSS, 1990).

Para isso, é definido neste trabalho, um novo processo ágil para apoiar somente o desenvolvimento de *software*, denominado PARFAIT/EA, abstraído a partir do processo ágil PARFAIT (**P**rocesso **Á**gil de **R**eengenharia baseado em **F**r**A**meworks no domínio de sistemas de **I**nformação com técnicas de VV**&T)** que apóia somente a reengenharia, e que é um dos recursos do ARA. Práticas de outros métodos ágeis são estudadas para serem incorporadas ao novo processo. Uma ferramenta existente denominada P.DOCTool (BIANCHINI, 2004) utilizada na documentação de processos, em que a estrutura do processo seja baseada na do arcabouço RUP (*Rational Unified Process*), foi evoluída neste trabalho para permitir o gerenciamento de processos e o planejamento e execução de projetos.

## 1.2 Motivação e Justificativa

Vários recursos são disponibilizados pelo ARA para apoiar a reengenharia, como exemplo o processo PARFAIT. Para apoiar somente a engenharia reversa[4], um processo abstraído do PARFAIT foi proposto por Cagnin *et al* (2003c), denominado PARFAIT/RE. Ambos os processos são aplicados em sistemas legados[5] no domínio de Sistemas de Informação[6].

O processo PARFAIT/RE foi definido a partir de um estudo de caso (Cagnin *et al* (2003c)), em que foram observados resultados positivos no uso do PARFAIT somente para conduzir a engenharia reversa, com: a utilização da linguagem de padrões de análise para apoiar o entendimento do domínio do sistema legado, bem como para apoiar a elaboração da

[4] "Processo de transformação de um código em um modelo pelo mapeamento a partir de uma linguagem de implementação específica" (BOOCH *et al,* 2000).
[5] Sistemas Legados são sistemas de *softwares* antigos que estão em funcionamento em uma empresa. Fornecem serviços fundamentais para o dia-a-dia, mas possuem tecnologia ultrapassada (SOMMERVILLE, 2003).
[6] "Sistema de Informação pode ser definido tecnicamente como um conjunto de componentes inter-relacionados que coleta (ou recupera), processa, armazena e distribui informações destinadas a apoiar a tomada de decisões, a coordenação e o controle de uma organização" (LAUDON, 2004).

sua documentação no paradigma orientado a objetos o uso do *framework* para gerar um protótipo do sistema o mais rápido possível, a fim de identificar novos requisitos (não presentes no sistema legado, mas fornecidos pelo *framework* e presentes no protótipo) e/ou aprimorar os requisitos anteriormente identificados no sistema legado com o apoio da execução de casos de teste no protótipo gerado, utilizados anteriormente para validar o sistema legado durante o seu desenvolvimento, participação constante do cliente durante o processo, colaborando para produzir documentação consistente, entre outros.

Com base nesses dois processos, um utilizado na reengenharia e o outro na engenharia reversa, surgiu o interesse em criar um novo processo voltado somente para a engenharia avante.

Ferramentas computacionais vêm sendo utilizadas para facilitar o uso e a gerência de processos. Como o PARFAIT e todos os demais processos abstraídos a partir dele possuem documentação baseada na estrutura do RUP, observou-se na literatura a existência da ferramenta P.DOCTool. Essa ferramenta apóia a documentação de processos com esse tipo de documentação e possui código fonte disponível, por ser uma ferramenta acadêmica desenvolvida no grupo de pesquisa ICMC-USP. Nesse contexto e observada a importância de apoio computacional também para a gerência de processos, houve a motivação neste trabalho em estender a ferramenta P.DocTool para permitir isso.

## 1.3 Objetivo do trabalho

Este trabalho tem como objetivo definir um processo ágil para o desenvolvimento de *software*, baseado no uso de *frameworks* que utilizem linguagem de padrões de análise em sua construção. *Framework* é utilizado na construção de cada versão do *software* e a linguagem

de padrões de análise facilita o entendimento e a documentação do *software*. Tal processo é abstraído a partir do processo PARFAIT. Além disso, outras práticas ágeis estudadas são adicionadas ao processo a fim de aumentar a sua agilidade. Com a utilização do processo no desenvolvimento de *software,* espera-se obter *software* com qualidade em um tempo menor de desenvolvimento, diminuindo prazos e, conseqüentemente, os custos do desenvolvimento. Para automatizar a gerência de projetos utilizando processos ágeis, uma ferramenta que apóia somente a documentação de processos é evoluída neste trabalho.

## 1.4 Organização do Trabalho

Esta dissertação está organizada em sete capítulos. Neste capítulo foi apresentada a importância da qualidade de *software* e de técnicas de reúso utilizadas no desenvolvimento de produtos de *software*. Discutiu-se também a necessidade de definir um processo de desenvolvimento a partir do processo PARFAIT com o apoio do arcabouço ARA e com a utilização de práticas ágeis estudadas na literatura, além daquelas existentes no ARA. Foram apresentados o contexto, a motivação, a justificativa e os objetivos para a realização deste trabalho. Os próximos capítulos estão organizados da seguinte maneira:

No Capítulo 2 apresentam-se conceitos sobre processo de desenvolvimento de *software* e de alguns métodos ágeis existentes na literatura.

No Capítulo 3 apresentam-se algumas técnicas que permitem reúso de *software* e que são importantes para o entendimento do trabalho, sendo: padrões, linguagens de padrões e *frameworks.*

No Capítulo 4 apresentam-se o arcabouço ARA e o processo PARFAIT, itens de grande relevância no trabalho.

No Capítulo 5 apresentam-se ferramentas para gerenciar projetos de *software* que utilizam métodos ágeis e uma ferramenta, denominada P.DOCTool, para documentar e consultar processos cuja documentação seja baseada no formato da documentação do RUP(*Rational Unified Process*)

No Capítulo 6 apresentam-se a definição e a evolução do processo ágil de desenvolvimento baseado em *framework* PARFAIT/EA.

No Capítulo 7 apresenta-se a extensão da ferramenta P.DOCTool, evoluída neste trabalho, para atender o planejamento e a execução de projetos de *software* e, conseqüentemente, o gerenciamento de processos.

No Capítulo 8 apresentam-se a conclusão do trabalho, as limitações e os trabalhos futuros.

No Apêndice A apresenta-se a documentação completa do processo PARFAIT/EA.

# CAPÍTULO 2. DESENVOLVIMENTO DE *SOFTWARE*

## 2.1. Considerações Iniciais

Neste capítulo apresenta-se o embasamento teórico para apoiar o entendimento do trabalho apresentado no Capítulo 6. Na Seção 2.2 apresentam-se algumas definições de processo de *software* de acordo com alguns autores encontrados na literatura. Na Seção 2.3 apresentam-se alguns métodos ágeis, os quais preocupam se em desenvolver, o *software* com qualidade e rapidez. Na Seção 2.4 algumas comparações entre os métodos ágeis são apresentadas em quadro comparativo. Na Seção 2.5 apresentam-se alguns trabalhos correlatos ao trabalho desenvolvido e na Seção 2.6, são feitas as considerações finais sobre este capítulo.

## 2.2. Processo de *Software*

Existem diversas definições de processos de *software* que são encontradas na literatura. De acordo com Sommerville (2003), processo é um conjunto de atividades e de resultados associados que levam à produção de um produto de *software*. Processos são complexos e dependem de julgamento humano. Essa definição pode ser complementada pela de Fuggeta (2000), que aborda processo de *software* como um conjunto coerente de políticas, estruturas organizacionais, tecnologias, procedimentos e artefatos necessários para conceber, desenvolver, entregar e manter um produto de *software*. Sommerville (2003) divide o processo em atividades ou fases e ao final de cada atividade obtém-se uma parte do produto, já Fuggeta (2000) engloba ao processo questões administrativas e artefatos que podem ser

utilizados em todo o processo de desenvolvimento do produto. As duas definições buscam o mesmo resultado, que é a obtenção do produto de *software*.

Similarmente a Sommerville (2003), Pressman (2002) relata processo como uma estrutura, composta de atividades que são aplicadas em todo o projeto de *software* e Pfleeger (2004) aborda processo como um conjunto de tarefas ordenadas. Os autores Sommerville e Pressman relacionam ao processo atividades, restrições e recursos.

O processo de *software* envolve várias etapas que compõem o ciclo de desenvolvimento do produto, podendo ser chamado, de acordo com Pfleeger (2004), como ciclo de vida do *software*, pois abrange desde a concepção do produto até a implementação, entrega, utilização e manutenção.

Não há um processo considerado ideal e diferentes empresas desenvolvem abordagens distintas para o desenvolvimento de *software* (SOMMERVILLE, 2003). Este fato está relacionado ao grande número de processos existentes, para atender os diversos tipos de *software*.

A evolução de um processo depende da capacidade das pessoas em explorá-lo (SOMMERVILLE, 2003) e a qualidade do *software* é incorporada durante o processo com a aplicação adequada de métodos, procedimentos e ferramentas (PRESSMAN, 2002). A evolução do processo leva à maturidade. Isso é possível devido ao uso constante do processo em projetos de *software*, o que possibilita a identificação de pontos fracos ou não cobertos pelo processo e a melhoria das suas atividades. A utilização de uma ferramenta para gerenciar projetos de *software*, contribui tanto para o desempenho do projeto quanto para a execução das atividades do processo.

A maturidade dos processos de *software* geralmente é baseada em projetos anteriores e na troca de experiência entre os participantes do projeto (PFLEEGER, 2004). Esse fato relaciona-se à experiência adquirida pela equipe, por já possuir uma idéia das atividades

necessárias para o desenvolvimento e das estimativas de prazos, custos e os possíveis riscos que podem vir a acontecer. Alguns processos fornecem critérios que permitem avaliar a execução de suas atividades ou fases, como o processo de reengenharia PARFAIT, proposto por Cagnin (2005a), que ao final de cada fase possui marcos de referência (*milestones*) com critérios para avaliação, indicando ao engenheiro de *software* a situação do projeto e fornecendo opção de cancelar a execução do projeto, alterar ou continuar.

De acordo com Sommerville (2003), existem grandes diferenças entre processos de desenvolvimento de *software*, mas algumas atividades são fundamentais e comuns entre todos eles, como:

1. Especificação de *software* – define as funcionalidades e as restrições de operação do *software*.

2. Projeto e implementação de *software* - produz o *software* de modo que cumpra sua especificação.

3. Validação de *software* – valida o *software* para garantir que faz realmente o que foi especificado.

4. Evolução de *software* – evolui para atender as novas necessidades do cliente.

Com base nestas atividades, diversos modelos de processos ou também chamados, segundo Carvalho (2001), de paradigmas de desenvolvimento, seguem esta seqüência em sua definição, sendo que vários são os modelos de processos encontrados na literatura (SOMMERVILLE, 2003; PRESSMAN, 2002; PFLEEGER, 2004). Esses autores abordam modelos de processo de *software* como uma descrição para o desenvolvimento do *software* e apresentam alguns modelos de processos mais populares, como: modelo seqüencial linear ou modelo cascata, modelo incremental, modelo em espiral, modelo de prototipagem, entre outros.

Em modelos de processos destacam-se atividades que são parte do processo de *software*, produtos de *software* e os papéis dos responsáveis envolvidos no processo.

Embora a utilização de processos de *software* e métodos no desenvolvimento de *software* tenham vários objetivos comuns*,* como exemplo a qualidade do *software*, existem diferenças entre os conceitos. Segundo Pressman (2002), método é um conjunto de técnicas utilizadas no desenvolvimento de *software*, que incluem um conjunto de tarefas (análise de requisito, projeto, implementação, entre outras) por meio dos princípios básicos da Engenharia de *Software*, já o processo de *software*, abrange métodos, ferramentas e procedimentos. Chan *et al* (2007), afirmam que decidir as tarefas que irão compor um processo de desenvolvimento, não é uma tarefa fácil, pois no processo de desenvolvimento devem-se considerar algumas características, como a equipe do projeto, o projeto, o conhecimento em Engenharia de *Software* entre outras. Em ambas as definições, verifica-se a importância do conhecimento em Engenharia de *Software* e da maneira como é aplicado esse conhecimento em um projeto de *software.* Alguns métodos ágeis são apresentados na próxima seção.

## 2.3. Métodos Ágeis

Os métodos ágeis surgiram com um manifesto, denominado "Manifesto para o Desenvolvimento Ágil de *Software*[7]" (BECK *et al,* 2001), organizado por um grupo de dezessete pessoas que se reuniu no estado de Utha, Estados Unidos. O objetivo deste manifesto foi determinar características comuns entre os métodos ágeis em confronto com os métodos tradicionais, rompendo as resistências aos processos usuais, proporcionando um

[7] http://www.agilemanifesto.org. Acesso em Outubro/2005

processo de desenvolvimento de *software* menos burocrático, mais simples e com possibilidade a mudanças de requisitos durante todo o projeto, refletindo as necessidades do cliente.

O manifesto resultou em alguns princípios e valores adotados pela "Aliança Ágil", nome atribuído à união das metodologias formadas no encontro. Os quatros princípios básicos do manifesto ágil são listados a seguir (BECK *et al,* 2001):

- **Indivíduos e interações** são mais importantes que processos e ferramentas.
- ***Software* funcionando** é mais importante do que documentação completa e detalhada.
- **Colaboração com o cliente** é mais importante do que negociação de contratos.
- **Adaptação a mudanças** é mais importante do que seguir o planejamento inicial.

Os métodos ágeis enfatizam o ponto de vista humano do desenvolvimento de *software* ao invés do ponto de vista de Engenharia (LYCETT *et al*, (2003) **apud** (GARCIA *et al*, 2005)). Uma outra preocupação dos métodos está relacionada à forma iterativa de desenvolvimento e não somente na documentação suficiente, o que demanda menos tempo quando comparado com os métodos tradicionais.

As práticas referentes aos métodos ágeis são apresentadas a seguir (BECK *et al,* 2001):

- Satisfazer os clientes entregando versões contínuas o mais cedo possível.
- Permitir mudanças de requisitos em qualquer fase do desenvolvimento.
- Entregar versões de *software* que funcionem adequadamente em curtos períodos de tempo.
- Proporcionar trabalho em conjunto entre desenvolvedores e clientes.
- Fornecer infra-estrutura necessária para que os indivíduos se tornem motivados e desempenhem o trabalho esperado.

- Proporcionar comunicação direta com os participantes do projeto.
- Alcançar a medida primária de progresso, que é *software* funcionando adequadamente.
- Manter harmonia entre clientes, desenvolvedores e usuários.
- Preocupar-se com a qualidade técnica e com a execução de bons projetos.
- Projetar com simplicidade.
- Permitir que equipes da própria organização participem da definição da arquitetura, requisitos e projeto do sistema.
- Permitir que as equipes, em intervalos regulares, expressem como podem se tornar mais efetivas.

Como pode ser notada por meio das práticas citadas, a idéia principal dos métodos ágeis é diminuir a burocracia existente nos métodos tradicionais, facilitar a comunicação entre os participantes do projeto e produzir *software* com qualidade, diminuindo os riscos e prazos.

Os métodos ágeis devem ser aplicados no desenvolvimento de *software* que não possuem requisitos estáticos, com equipes pequenas e que exijam um tempo menor de desenvolvimento. Os métodos tradicionais devem ser aplicados apenas em situações em que os requisitos de sistema são estáveis e requisitos futuros são previsíveis (SOARES, 2004b).

Uma outra característica, de acordo com Garcia *et al* (2005), é que os métodos ágeis não são centrados em artefatos e utilizam documentação apropriada para evitar redundâncias e excessos de documentos no desenvolvimento do *software*.

Segundo Highsmith (2002), desenvolvimento ágil não é definido por somente um jogo pequeno de práticas e técnicas, mas com uma potencialidade estratégica para adaptação à mudanças, à flexibilidade, à criatividade entre os participantes do projeto e à definição de uma equipe de desenvolvimento com o objetivo de conduzir um projeto por meio de turbulências e incertezas.

Existem hoje diversos métodos ágeis sendo utilizados em empresas, universidades e centros de pesquisas. Dentre os métodos temos: XP (KENT BECK, 2000), Scrum (K.SCHAWABER e BEEDLE, 2002), FDD (*Feature Driven Development)* (PALMER; FELSIN, 2002)*,* família *Crystal* (COCKBURN, 1998*),* DSDM (*Dynamic System Development Method)* (STAPLETON, 1997), ASD (*Adaptative Software Development*) (HIGHSMITH, 2000). Entre os métodos ágeis, o XP é o mais conhecido (SOARES, 2004b). As subseções a seguir fornecem uma visão geral de cada um desses métodos.

### 2.3.1. eXtreme Programming (XP)

Criado por Kent Beck, o método XP é utilizado no desenvolvimento de *software,* em que os requisitos são vagos e estão em constante mudanças, formado por equipes de desenvolvedores de tamanho pequeno e médio (BECK, 2000). XP é um método ágil, pois procura responder com velocidade às mudanças nas especificações do projeto, com base nos princípios, valores e práticas definidas pelo método.

XP utiliza alguns valores que o diferenciam de outros métodos, como: comunicação, simplicidade, *feedback* e coragem, e doze práticas simples que são utilizadas pela equipe de um projeto de *software*.

O método XP enfatiza o desenvolvimento rápido do projeto com objetivo de garantir a satisfação do cliente e cumprir estimativas do projeto. Realiza-se o máximo possível a comunicação pessoal, evitando o uso de telefone e o envio de mensagens por correio eletrônico (SOARES, 2004a). A comunicação é um dos valores principais do XP, que visa a manter um melhor relacionamento entre clientes e desenvolvedores, encorajando a comunicação entre gerente e desenvolvedores. Segundo Nawrocki (2002), com o incentivo na

participação do cliente em um projeto XP e a liberação de versões freqüentes, existe uma probabilidade menor de erro no projeto e permite solucionar muitos problemas.

O valor simplicidade representa o desenvolvimento de um projeto XP da maneira mais simples possível, que vai desde a implementação de requisitos atuais até a criação de linhas de código, evitando funções desnecessárias. O código fonte é analisado por meio de testes aplicados constantemente em um projeto XP.

A finalidade do *feedback* é manter atualizados o desenvolvedor e o cliente do andamento do projeto, podendo, assim, o cliente sugerir novas características, evitando a barreira entre cliente, gerente de projeto e desenvolvedor.

O valor coragem se enquadra na capacidade de implantar os valores anteriores, já que muitas pessoas não possuem bom relacionamento e comunicação. Segundo Soares (2004a), a coragem dá suporte à simplicidade quando a equipe percebe que é possível simplificar o *software*, permitindo a construção de projetos mais simples.

#### 2.3.1.1 Práticas

As práticas do XP são apresentadas a seguir, de acordo com Beck (2000):

- Jogo do planejamento – usa estimativas de custo fornecidas pelos programadores para determinar o que necessita ser feito e o que pode ser adiado no projeto, permitindo que o cliente decida o que é menos importante e o que pode ser desenvolvido em uma próxima versão do *software*.
- Versões Pequenas - disponibiliza versão do sistema em funcionamento o mais rápido, atualizando o sistema freqüentemente em um ciclo pequeno de desenvolvimento.

- Metáfora – cria descrições comuns do *software*, evitando termos técnicos para facilitar o desenvolvimento e a comunicação.
- Projeto simples – orienta a simplicidade no desenvolvimento do sistema, satisfazendo os requisitos atuais, sem a preocupação de requisitos futuros.
- Teste – valida todo o projeto de *software* com o uso de teste. Testes de unidade são criados antes do código e são utilizados em todo o projeto. Os clientes criam estórias que demonstram as características do sistema. Essas estórias são validadas por meio de teste funcional. Teste funcional é também conhecido como teste de caixa preta (MYERS, 2004). Esta técnica preocupa-se com as funções do sistema, sem se preocupar com os detalhes da implementação (MALDONADO *et al*, 2004). Para essa técnica são utilizados alguns critérios, como: Particionamento de Equivalência, Análise do Valor Limite, Grafo de Causa Efeito (MYERS, 2004), entre outros. Os testes de aceitação também são utilizados e devem ser especificados pelo cliente, assumindo a responsabilidade de determinar de que forma aceitará o produto que está sendo desenvolvido.
- Refatoração – reestrutura o sistema sem mudar seu comportamento, com o objetivo de remover duplicação de código, melhorar a comunicação entre os programadores e simplificar ou adicionar flexibilidade ao código.
- Programação em pares – permite a existência de dois programadores para a construção do código trabalhando em uma única máquina, para que um ajude o outro. Enquanto um escreve o código o outro analisa, verificando se existe a necessidade de mudança e a identificação de erros.
- Propriedade coletiva do código – mantém todo o código disponível a todos os membros da equipe, podendo qualquer membro adicionar um trecho de código no sistema.

- Integração contínua - integra o sistema várias vezes ao dia, cada vez que uma tarefa é finalizada.
- 40 horas de trabalho semanal – exige que cumpra uma carga horária de 40 horas semanais para que todos permaneçam descansados.
- Cliente presente – requer participação do cliente no desenvolvimento do projeto para sanar todas as dúvidas que possam surgir.
- Padrões de codificação – padroniza a escrita do código para que a equipe trabalhe eficazmente em pares e que possa compartilhar todo o código.

Segundo Campelo (2003), a questão da manutenção de sistemas produzidos a partir de um projeto XP é bastante questionada quanto à sua eficácia devido a pouca documentação pregada pela metodologia.

Neste contexto, Nawrocki (2002) relata que as fontes de conhecimento em projetos XP são: código fonte, casos de teste e a memória dos programadores. O risco em relação à pouca documentação está nas alterações em projetos antigos e na saída de programadores da equipe. Nawrocki (2002) afirma, ainda, que a única base para a manutenção é o código fonte e os casos de teste.

### 2.3.2. *Scrum*

Criado por Jeff Suttherland e Ken Schwaber, o método *Scrum* fornece um processo para o desenvolvimento do *software,* concentrando em como a equipe deve trabalhar em um projeto *Scrum* em constante mudança (ABRAHAMSSON *et al*, 2002).

O termo *Scrum*, de acordo com SCHWABER e BEEDLE, ((2002) **apud** (ABRAHAMSSON *et al*, 2002)) origina da estratégia do jogo de *rugby*[8], em que as equipes lutam pela posse da bola em um círculo, com o objetivo de atingir uma meta. As equipes atuam em conjunto, ocorrendo freqüentes trocas de bola entre os companheiros. Analogamente, no desenvolvimento, o termo *Scrum* é formado por uma equipe pequena, em que existe uma comunicação entre todos os integrantes da equipe, seguindo o mesmo objetivo.

*Scrum* apresenta uma abordagem empírica formada por algumas idéias da teoria de controle de processos industriais para o desenvolvimento de *software*, reintroduzindo as idéias de flexibilidade, adaptabilidade e produtividade. O foco do método é encontrar uma forma de trabalho dos membros da equipe para produzir o *software* de forma flexível e em um ambiente em constante mudança.

O método é baseado em princípios semelhantes aos de XP: equipes pequenas, requisitos pouco estáveis ou desconhecidos e iterações curtas para promover visibilidade para o desenvolvimento. As equipes de um projeto podem ser formadas por cinco a nove pessoas (ABRAHAMSSON *et al*, 2002).

Segundo Abrahamsson *et al* (2002), existem grandes esforços em juntar XP e *Scrum*, visto que *Scrum* fornece uma estrutura para a gerência de projeto, suportado pelas práticas de XP, formando um pacote integrado de desenvolvimento de *software*.

Seu ciclo de vida é baseado em três fases principais. A fase de pré-planejamento, desenvolvimento e a fase de pós-planejamento. O pré-planejamento é dividido em duas fases secundárias: a fase de planejamento e a de arquitetura do projeto. As fases são apresentadas a seguir ((SCHWABER, 1995; SCHWABER; BEEDLE, 2002) **apud** (ABRAHAMSSON *et al,* 2002)).

[8] http://www.rugbymagazine.com.br/Historia.asp. Acesso em Novembro/2005

**Pré-planejamento (*pre-game phase*)**: nesta fase é criada a lista de *backlog* das funcionalidades (requisitos) do produto a ser construído. Esta lista é atualizada constantemente, aplicando detalhes aos itens. A fase de planejamento inclui outras atividades como a definição da equipe de desenvolvimento, as ferramentas a serem usadas, os possíveis riscos do projeto e as necessidades de treinamento; e a fase de arquitetura do projeto, inclui a arquitetura baseada no planejamento corrente da lista de *backlog*.

**Desenvolvimento (*development phase*)**: nesta fase são identificadas as variáveis técnicas e do ambiente. Essas variáveis são consideradas em todo o projeto, aumentando a flexibilidade para o acompanhamento das mudanças. O *software* é desenvolvido em unidades básicas (S*prints*) às quais novas funcionalidades são adicionadas. As S*prints* seguem o desenvolvimento tradicional, ou seja: análise, em seguida o projeto, implementação e testes. Para cada *Sprint* é feito um planejamento que deve durar no máximo um mês.

**Pós-planejamento (*post-game phase*)**: nesta fase é realizada a liberação do produto ao cliente. Verificam-se todas as funcionalidades foram implementadas e se estão de acordo com as exigências do cliente. Nesta fase são feitas as etapas de integração, testes finais no *software* e documentação.

#### 2.3.2.1 Práticas

*Scrum* não possui práticas ou métodos específicos de engenharia a serem usados no desenvolvimento e requer a utilização de determinadas práticas de gerência nas várias fases do desenvolvimento (ABRAHAMSSON *et al*, 2002). Os itens que compõem o método *Scrum* são: *Backlog* do produto, *Sprint,* Reunião de planejamento, *Sprint Backlog,* Reuniões de *Scrum* diárias, Equipes *Scrum* e Revisão, e estão descritos a seguir:

***Backlog* do produto –** representa uma lista de todas as funcionalidades desejadas para a construção do *software*. Nesta lista também são especificados os novos requisitos, tarefas, funções e tecnologias que poderão ser utilizados. A lista é priorizada pelo responsável do projeto.

***Sprint* –** representa o ciclo de desenvolvimento do *Scrum* (unidades básicas) com um tempo de desenvolvimento de até 30 dias. A meta do *Sprint* é a entrega de uma parte de um incremento do *software* para o cliente.

**Reunião de planejamento** – iniciada antes da execução do *Sprint* em que são definidos os itens que comporão a lista de *backlog* do *Sprint* e a ordem de prioridade que eles terão.

***Sprint Backlog* –** representa a execução da lista de *backlog* durante o ciclo do *Sprint.*

**Reuniões de *Scrum* diárias –** reúne a equipe uma vez por dia para discutir sobre o projeto e para manter continuamente o progresso da equipe. São respondidas algumas perguntas **"O que foi feito desde a última reunião?", "O que deve ser feito?**" **"Quais os problemas encontrados?"**. As reuniões não devem ultrapassar quinze minutos.

**Equipes *Scrum* –** formada por desenvolvedores e usuários. Estas equipes são pequenas e geralmente possuem sete participantes.

**Revisão** – realizam-se revisões sobre as últimas reuniões realizadas no último *Sprint.* Nas reuniões podem ser acrescentadas novas funcionalidades na lista de *backlog* ou alterações nas funcionalidades existentes.

Na Figura 2.1 apresenta-se o método *Scrum*: na etapa 1 são definidas as funcionalidades que irão compor a lista de *Backlog,* na etapa 2 são criados os *Sprints* com base na lista de *backlog,* na etapa 3 são distribuídas as tarefas, compostas na lista de *backlog* para os membros do projeto *Scrum,* a etapa 4 representa a execução do *Sprint,* na etapa 5 são

realizadas reuniões diárias para verificar o andamento do *Sprint* e na etapa 6 tem-se o produto disponível para uso.

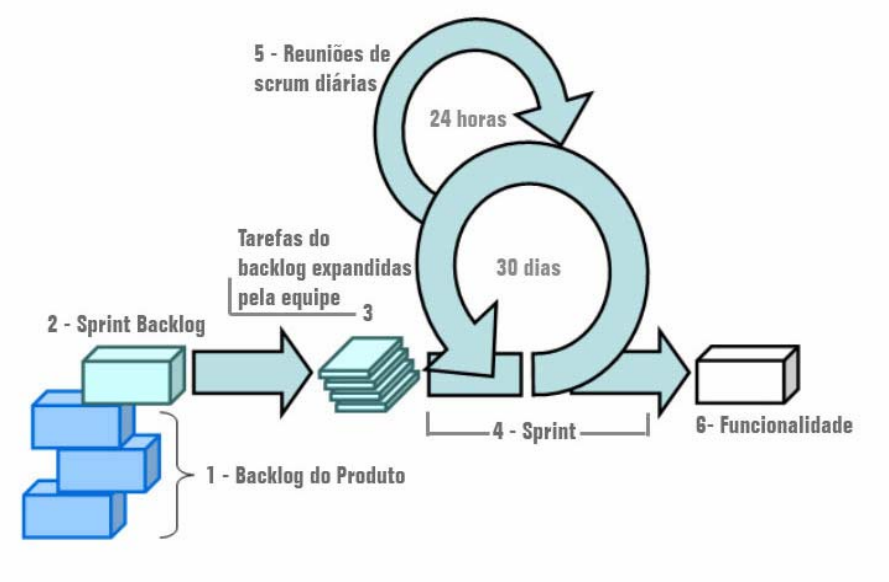


Figura 2.1: Representação do método *Scrum*[9] (adaptado de MULLER, 2004)

### 2.3.3 *Feature Driven Development* (FDD)

Criado por Jeff de Luca e Peter Code ((PALMER e FELSING, 2002) **apud** (ABRAHAMSSON *et al*, 2002)), o método FDD é um método ágil e adaptável ao sistema. Não cobre o processo inteiro de desenvolvimento do *software*, mas focaliza-o particularmente no projeto e nas fases de construção (ABRAHAMSSON *et al*, 2002).

FDD incorpora o desenvolvimento iterativo e as melhores práticas da modelagem ágil. Os aspectos de qualidade são enfatizados durante todo o processo de desenvolvimento, incluindo entregas freqüentes e tangíveis, bem como monitoração do progresso do projeto no período de desenvolvimento (ABRAHAMSSON *et al*, 2002).

[9] http:\\www.controlchaos.com. Acesso em Outubro/2005

FDD possui cinco processos seqüenciais durante o projeto e o desenvolvimento do sistema, como ilustrado na Figura 2.2 e logo em seguida descrito. A parte iterativa dos processos de FDD (“projetar por característica” e “construir por característica”) suporta o desenvolvimento ágil com adaptações rápidas às mudanças, de acordo com as exigências e as necessidades do negócio (ABRAHAMSSON *et al*, 2002). As iterações do projeto e construção de uma característica seguem por um período de uma a três semanas de trabalho.

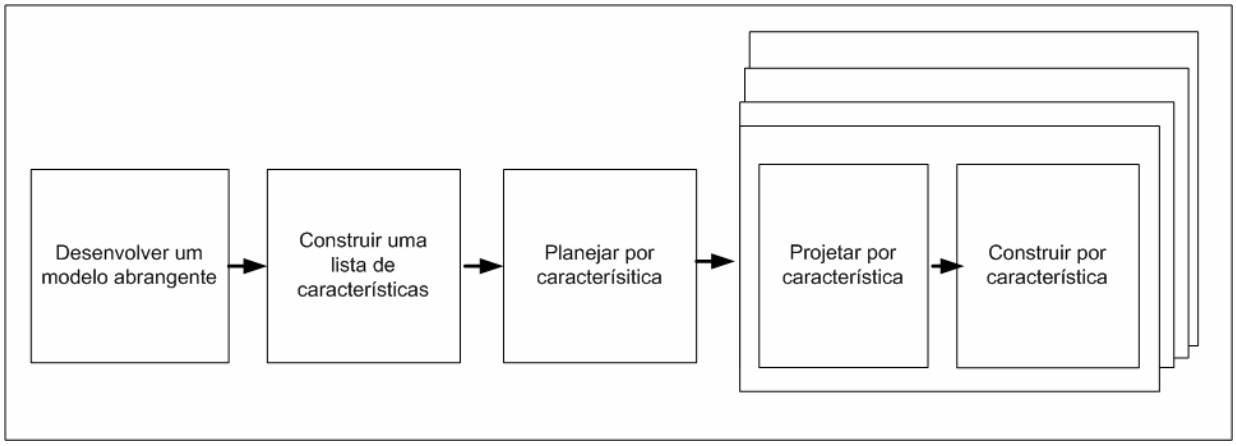


Figura 2.2: Processos FDD (adaptado de ABRAHAMSSON *et al,* 2002)

**Processo 1: “Desenvolver um modelo abrangente”** – Os membros de um projeto devem estar cientes do contexto e das exigências do sistema a ser construído logo no início do desenvolvimento do projeto. Isso é alcançado por meio de casos de uso ou especificações funcionais exigidos neste processo.

**Processo 2: “Construir uma lista de Características”** – A equipe identifica as características, agrupa-as hierarquicamente e atribui prioridades e tamanho. Entre as tarefas deste processo incluem a formação da equipe que irá projetar a lista de características.

**Processo 3: “Planejar por Características”** – Um plano de projeto é construído e usado nos processos seguintes, determinando a seqüência de desenvolvimento com as prioridades e as datas que cada característica deve ser completada.

**Processo 4: "Projetar por Características" –** Um pequeno grupo de características é selecionado do conjunto de características. Deste grupo são identificadas as classes que estão envolvidas e os seus respectivos proprietários. Cada característica selecionada irá passar por esta etapa, em que a equipe de características define um diagrama de seqüência (BOOCH *et al*, 2000) detalhado para ela. Os proprietários das classes estruturam suas classes e métodos. No final a equipe faz uma inspeção no projeto. Entre as tarefas deste processo incluem a formação da equipe de projeto e a definição de um guia de domínio, a construção do diagrama de seqüência, a estruturação das classes e métodos e a inspeção do projeto.

**Processo 5: "Construir por Características" –** Neste processo são realizados a implementação das classes e métodos, a inspeção do código, os testes de unidade e o desenvolvimento de cada característica ou conjunto delas.

A iteratividade entre os dois últimos processos é representada de forma incremental, em que as características vão sendo concluídas e novas características vão sendo inicializadas, dando origem a uma versão do *software.*

FDD possui requisitos mais formais e um mecanismo mais preciso para acompanhamento do projeto. Segundo Khramtchenko (2004), o desenvolvimento baseado em FDD consiste de dois estágios principais: descobrir uma lista de características a serem implementadas e realizar a implementação baseada em característica.

Khramtchenko (2004) relata que descobrir a lista de características é o processo mais crítico e o estágio principal para o sucesso do projeto. O fator crítico deste estágio está relacionado às dificuldades encontradas na especificação dos requisitos e na interpretação de determinados requisitos ambíguos. O sucesso do projeto pode ser atribuído à qualidade identificada na lista de características.

O método FDD tem participação integral do cliente junto à equipe de desenvolvimento.

### 2.3.3.1 Práticas

Algumas práticas do FDD possuem semelhanças às práticas do método *Scrum*, por exemplo, a lista de funcionalidades. No método *Scrum* a lista de funcionalidades é denominada "*Backlog* do Produto" e no método FDD é denominada "Desenvolvimento por Características". Tanto no método *Scrum* como no método FDD a lista de funcionalidades é um requisito muito importante, pois representa todo o sistema. Uma outra prática é com relação à equipe de projeto. No método *Scrum* é denominada "Equipes *Scrum"* e no método FDD "Equipes de característica"; em ambos os métodos as equipes são pequenas. Uma outra semelhança é com relação à maneira como é realizada a revisão. No método *Scrum* são realizadas reuniões diárias com o objetivo de verificar o andamento do projeto e no método FDD são realizadas configurações regulares, que visam assegurar que o sistema está disponível para adição de novas funcionalidades.

FDD possui algumas boas práticas apresentadas a seguir, segundo Abrahamsson *et al* (2002):

- Modelo de objeto do domínio: faz a exploração e a explanação do domínio do problema.
- Desenvolvimento por características: desenvolve uma lista de características, acompanhando seu progresso.
- Posse individual da classe: possui uma única pessoa nomeada para ser responsável por uma classe e pela sua consistência, desempenho e integridade.
- Equipes das características: formada por equipes pequenas que orientam o projeto.
- Inspeção: realiza inspeções no projeto ou no código fonte durante o desenvolvimento do projeto.

- Configurações regulares: utiliza configurações regulares para garantir a demonstração do sistema disponível, fornecendo base às novas características que poderão ser adicionadas.
- Gerência de configuração: realiza um controle das versões permitindo a identificação do histórico das últimas modificações, mantendo um histórico de todas as versões.

A equipe de projeto deve utilizar todas as práticas descritas anteriormente, obedecendo as regras do desenvolvimento e podendo ainda adaptá-las de acordo com a necessidade do projeto (ABRAHAMSSON *et al*, 2002).

### 2.3.4. Família *Crystal*

Criado por Alistair Cockburn, a família de métodos *Crystal* prioriza a comunicação entre os participantes do projeto e inclui um número diferente de métodos que atendem projetos com características diferentes (ABRAHAMSSON *et al*, 2002). A família *Crystal* é formada por *Crystal Clear, Crystal Yellow, Crystal Orange, Crystal Red e Crystal Orange/Web.* De acordo com Cockburn (2000), os principais métodos da família *Crystal* são o *Crystal Clear*, *Crystal Orange* e o *Crystal Orange/Web*. Nesta seção serão apresentados os métodos *Crystal Clear* e *Crystal Orange*, porque atendem ao domínio deste trabalho, ou seja, Sistema de Informação.

Abrahamsson *et al* (2002) relatam que a escolha de um método deve ser baseada no tipo de projeto. A dimensão e o tamanho de um projeto são representados por símbolos em que cada símbolo representa uma categoria que especifica o tipo de projeto com relação ao tamanho e à complexidade.

As regras, características e valores são comuns em todos os métodos da família *Crystal*. Segundo Cockburn (2000) dois valores próprios da família *Crystal* são a alta tolerância e a comunicação centrada nas pessoas. A tolerância relaciona-se ao comportamento humano com relação às ferramentas e produtos de trabalho utilizados em um projeto *Crystal.*

Todos os projetos utilizam ciclos de desenvolvimento incrementais com um tamanho máximo de incremento de quatro meses, mas preferivelmente entre um a três meses.

Os métodos *Crystal* não limitam nenhuma prática de desenvolvimento, ferramentas ou produtos de trabalho e também permitem a adaptação de outros métodos, como por exemplo, XP ou *Scrum* (ABRAHAMSSON *et al*, 2002).

A idéia principal de *Crystal* é o desenvolvimento do *software* visto como um jogo cooperativo para a invenção e a comunicação, com o objetivo da entrega rápida de *software*. Nos métodos *Crystal* cada projeto é diferente e evolui com o tempo (ABRAHAMSSON *et al,* 2002).

Para grandes projetos recomendam-se métodos ágeis mais "pesados", mais complexos, por exemplo, ASD (*Adaptative Software Development)*, ao invés dos métodos *Crystal* (ABRAHAMSSON *et al*, 2002).

### 2.3.4.1 Crystal Clear

*Crystal Clear* é um método ágil direcionado a projetos pequenos. Os membros da equipe têm especialidades distintas, existindo uma forte ênfase na comunicação entre os membros sendo que a organização do espaço de trabalho deve permitir isso.

Toda especificação e projeto são feitos informalmente utilizando um quadro branco, que deve estar em um lugar onde todos os participantes do projeto tenham acesso. Os

requisitos são elaborados utilizando o diagrama de casos de uso da linguagem de modelagem UML (*Unified Modeling Language*) (BOOCH *et al*, 2000), representando as tarefas que serão desenvolvidas em um projeto. A documentação de um projeto *Crystal Clear* é exigida, mas não existem regras ou documentos para apoio, permitindo que a equipe decida como documentá-lo (COCKBURN, 2000). As versões de *software* são liberadas em incrementos regulares de um mês.

COCKBURN, ((2002) **apud** (ABRAHAMSSON *et al*, 2002)) identificaram limitações a respeito dos métodos individuais usados na família *Crystal.* Por exemplo, o *Crystal Clear* tem uma estrutura relativamente restrita de comunicação e é apropriado somente para uma única equipe situada em um único espaço de trabalho. Além disso, faltam elementos para validação do sistema e não é apropriado para sistemas de vida crítico, como por exemplo, sistemas de tempo real que monitoram, analisam e controlam eventos do mundo real (PRESSMAN, 2002).

A idéia do *Crystal Clear* é permitir que cada organização implemente as atividades que lhe parecem adequadas, fornecendo o mínimo de suporte necessário do ponto de vista de comunicação e documentos.

### 2.3.4.2 Crystal Orange

*Crystal Orange* também é designado para projetos pequenos como *Crytal Clear*. Permite a participação de 10 a 40 pessoas no projeto com uma duração de um a dois anos.

*Crystal Orange* também requer que a equipe do projeto fique situada no mesmo ambiente de trabalho, facilitando assim a comunicação. De acordo com Abrahamsson *et al* (2002) as atividades de projeto e verificação de código não são bem definidas, o que também

não é recomendado para projetos que possuem vida crítica. Como já foi dito, também ocorre no *Crystal Clear.*

Em *Crystal Orange* o período de entrega de um incremento pode ser estendido para no máximo quatro meses (ABRAHAMSSON *et al*, 2002). Esse período representa uma prática que é definida na política padrão pelos métodos da família *Crystal*. As políticas padrões são apresentadas a seguir.

*Crystal Orange* inclui atividades de incremento que podem ser associadas na execução das práticas. Essas atividades são: planejar, monitorar, revisar e paralelizar e estão apresentadas a seguir (ABRAHAMSSON *et al*, 2002):

**Planejar:** inclui o planejamento do próximo incremento no sistema, de acordo com as exigências selecionadas pela equipe. É realizada uma reunião para a liberação de uma versão do sistema entre três ou quatro meses.

**Monitorar**: monitora o progresso durante o desenvolvimento do processo respeitando o progresso e a estabilidade. O progresso é medido por marcos de referência.

**Revisar**: incrementa diversas iterações. Cada iteração inclui as seguintes atividades: construção, demonstração e revisão dos objetivos.

**Paralelizar:** permite a integração de uma nova tarefa quando a tarefa anterior foi monitorada, o que produz estabilidade ao resultado.

### 2.3.4.3 Políticas Padrões (Práticas)

Políticas padrões são as práticas que precisam ser aplicadas no processo de desenvolvimento. Tanto o *Crystal Clear* quanto *Crystal Orange* sugerem as seguintes políticas padrão ((COCKBURN, 2002) **apud** (ABRAHAMSSON *et al*, 2002)).

- realizar entrega incremental e regular do *software* a cada 2 ou 3 meses.
- utilizar marcos de referência para medir o progresso do projeto, por meio de critérios de avaliação, baseando nas entregas do *software* e nas principais decisões que são escritas nos documentos.
- permitir a participação direta do usuário no projeto.
- utilizar teste de regressão de cada funcionalidade.
- disponibilizar duas visões do sistema para conferência do usuário.
- criar *workshop* para ajustes no produto e na metodologia no início e meio de cada incremento.

As políticas padrões do método *Crystal* são obrigatórias, mas podem, contudo, restabelecer práticas equivalentes de métodos, como XP ou *Scrum* ((COCKBURN, 2002) **apud** (ABRAHAMSSON *et al*, 2002)).

### 2.3.5 *Dynamic Systems Development Method* (DSDM)

Criado pela Consortium[10] no Reino Unido em 1994, o método DSDM é um processo de desenvolvimento baseado no RAD (*Rapid Application Development)* (PRESSMAN, 2002). A idéia fundamental do DSDM é o ajuste do tempo e dos recursos, em vez de definir funcionalidades para um produto. As funcionalidades vão sendo ajustadas de acordo com o andamento do processo, forçando a cooperação e a colaboração entre todos os membros interessados no projeto (ABRAHAMSSON *et al*, 2002).

Norfolk (2004) ressalta que DSDM é baseado na participação contínua do usuário em um processo iterativo de desenvolvimento.

[10] http://www.dsdm.org. Acesso em Novembro/2005

O ciclo de vida DSDM se divide em cinco fases: estudo de viabilidade, estudo do negócio, modelo de iteração funcional, construção da iteração e implementação (ABRAHAMSSON *et al*, 2002).

A fase do estudo de viabilidade tem como objetivo avaliar o projeto, verificar o tipo de projeto e toda a organização, já na fase de estudo do negócio são analisadas as características essenciais do negócio e a tecnologia disponível.

A fase de modelo de iteração funcional é a primeira fase iterativa e incremental. Cada iteração é planejada e os resultados são analisados para iterações adicionais.

Na fase de construção da iteração o sistema é construído. A saída é um sistema testado que cumpra pelo menos o mínimo das exigências definidas na fase modelo de iteração funcional. O projeto e a gerência de configuração são iterativos e os protótipos funcionais são revistos pelos usuários. O desenvolvimento adicional é baseado nos comentários do usuário.

Na fase de implementação, o sistema é transferido do ambiente de desenvolvimento para o ambiente de produção. São realizados treinamentos com o usuário após a entrega do sistema.

Em DSDM a atividade de teste ocorre durante todo o ciclo de vida, tendo como princípio a remoção do defeito quando possível, tornando o projeto mais barato (NORFOLK, 2004). Os testes utilizados pelo método são teste funcional e teste de aceitação (HORRIAN *et al*, 2003).

De acordo com Abrahamsson *et al* (2002), uma característica relevante do método é a utilização da prototipação no seu ciclo de desenvolvimento. Nas três últimas fases são priorizadas as funcionalidades para cada iteração e um protótipo funcional é construído. A última etapa corresponde à transferência do sistema do ambiente de desenvolvimento para o ambiente de produção. Na Figura 2.3, ilustra-se o processo DSDM.

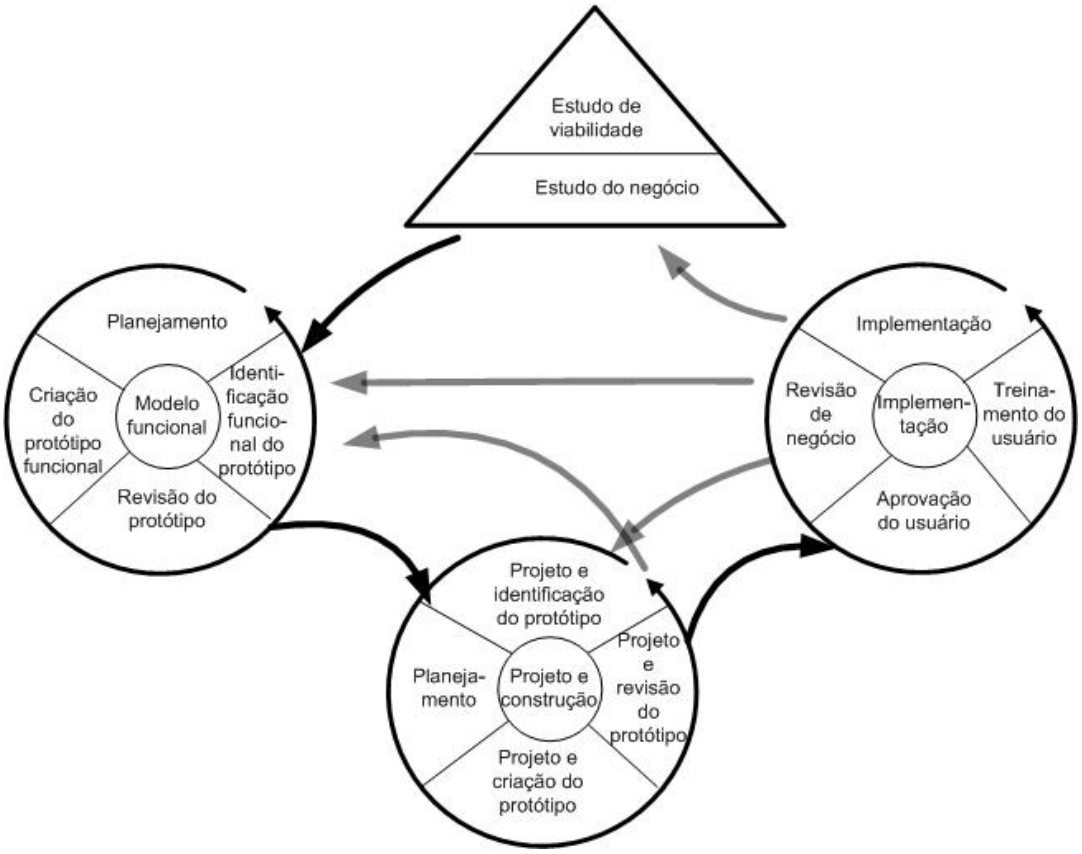


Figura 2.3: Diagrama do processo DSDM (adaptado de ((STAPLETON, 1997) **apud** (ABRAHAMSSON *et al*, 2002))

A equipe em um projeto DSDM, segundo Abrahamsson *et al* (2002), pode variar entre duas a seis pessoas, podendo existir várias equipes pequenas em um projeto. Em uma equipe de duas pessoas deve existir pelo menos um usuário e um colaborador. Os autores ressaltam que o ideal para a utilização do método DSDM em projetos grandes e pequenos é uma equipe formada por seis pessoas.

### 2.3.5.1 Práticas

O DSDM baseia-se em nove práticas que também são chamadas de princípios e estão apresentadas a seguir:

- usuário tem participação ativa no projeto.
- as equipes no DSDM devem ter autonomia para tomar decisões.
- o foco é voltado nas entregas freqüentes do produto.
- capacidade para o propósito do negócio pela equipe do projeto é um critério essencial para a aceitação das especificações do produto.
- desenvolvimento iterativo e incremental.
- todas as mudanças durante o desenvolvimento são reversíveis.
- o detalhamento nas funcionalidades definidas é uma das exigências estabelecidas, permitindo que funcionalidades essenciais sejam definidas logo no início do projeto.
- teste funcional e teste de aceitação são integrados ao ciclo de vida.
- aproximação colaborativa e cooperativa entre todos os participantes é essencial.

### 2.3.6 *Adaptative Software Development* (ASD)

Criado por Jim Highsmith (2000), o método *Adaptative Software Development,* baseia-se em práticas derivadas do RAD (PRESSMAN, 2002), orientando o desenvolvimento para aceitar as mudanças. Tem seu foco voltado principalmente para resolver problemas no desenvolvimento de sistemas grandes e complexos. O método incentiva fortemente o desenvolvimento incremental, iterativo e com prototipação constante (ABRAHAMSSON *et al,* 2002).

O processo de desenvolvimento é guiado por meio de ciclos, compostos por três fases: especulação, colaboração e aprendizado. Abrahamsson *et al* (2002) ressaltam que as fases são nomeadas conforme enfatizam o papel da mudança no processo, sendo: especulação,

colaboração e aprendizado. ASD permite mudanças no projeto, não visualizando como um problema e sim como uma vantagem (HIGHSMITH, 2002). A não resistência a mudanças enfatiza uma característica dos métodos ágeis, que é ser adaptativo.

Na Figura 2.4 apresenta-se o ciclo de vida do método ASD. Na fase de especulação é realizado o planejamento do projeto, a fase de colaboração apóia a equipe de trabalho nas mudanças do projeto e a fase de aprendizado representa o conhecimento envolvido no projeto, enfatizando o reconhecimento de erros e mudanças durante o desenvolvimento.


Figura 2.4: Ciclo ASD (adaptado de ABRAHAMSSON *et al*, 2002)

O enfoque no ciclo ASD é mais voltado aos resultados com qualidade, do que às tarefas a serem desempenhadas (HIGHSMITH, 2002). As tarefas representam as atividades existentes para o desenvolvimento das funcionalidades.

Na Figura 2.5 apresentam-se as fases do ciclo de vida do método ASD. A primeira fase define a atividade para a iniciação do projeto e é a atividade responsável pelo ciclo de desenvolvimento adaptável (fase de **especulação**). A fase de **colaboração** possui uma atividade utilizada para o desenvolvimento do componente (módulo) e a fase de **aprendizado** engloba as atividades de revisão, qualidade e liberação de versões. Na fase de **aprendizado**, existe um retorno da atividade de **revisão de qualidade** para a atividade **ciclo adaptável** da fase de **especulação**, representando o ciclo de aprendizado do método.

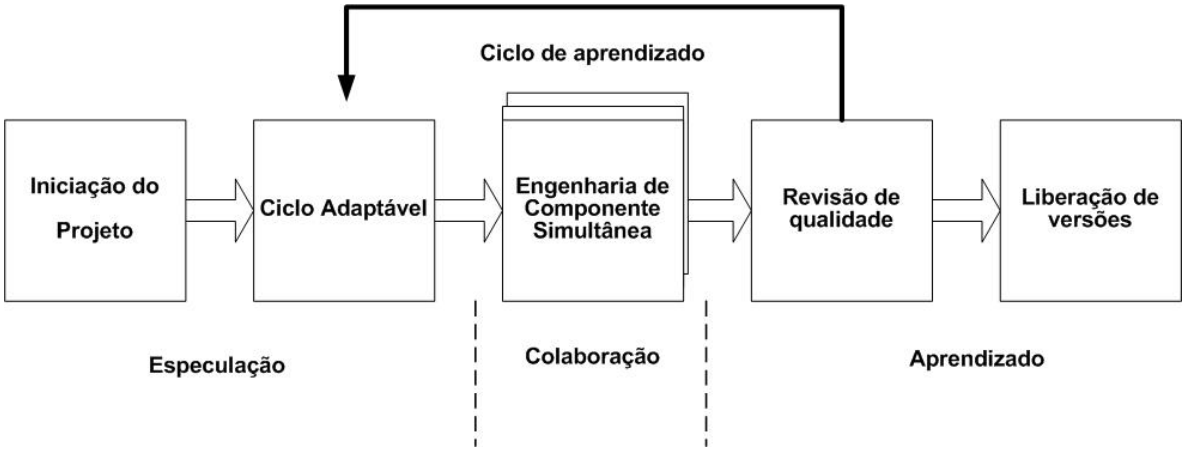


Figura 2.5: Fase do ciclo de vida ASD (adaptado de ((HIGHSMITH, 2000) **apud** (ABRAHAMSSON *et al*, 2002)))

Os ciclos de desenvolvimento adaptáveis contêm a fase de colaboração, em que diversos módulos podem estar sendo desenvolvidos simultaneamente. Planejar os ciclos é uma parte do processo iterativo, porque as definições dos módulos são refinadas continuamente refletindo a nova informação e fornecendo novas mudanças (ABRAHAMSSON *et al*, 2002). Os ciclos duram em média de 4 a 8 semanas.

O ciclo ASD possui seis características básicas que devem ser seguidas em um projeto. Essas características são apresentadas a seguir (ABRAHAMSSON *et al*, 2002):

**Conduzido à missão** - as atividades em cada ciclo de desenvolvimento devem ser ajustadas de acordo com o projeto.

**Baseado em módulos** – as atividades não devem ser orientadas a tarefas, mas de preferência ao desenvolvimento do *software*, construindo pequenas versões em pequenos períodos.

**Iterativo** – o desenvolvimento deve ser bem compreendido e bem definido.

**Quadro do tempo** – a ambigüidade em projetos complexos com relação a prazos pode ser evitada com o uso de históricos de projetos anteriores. A gerência do projeto força os participantes a tomarem decisões inevitáveis no início do projeto.

**Dirigido aos riscos** – as mudanças são freqüentes no desenvolvimento do *software* e devem ser avaliadas constantemente para sua adaptação.

**Tolerante a mudanças** – as mudanças que proporcionam risco ao projeto devem começar o mais rápido possível.

#### 2.3.6.1 Práticas

ASD propõe poucas práticas para o trabalho de desenvolvimento do *software*, sendo basicamente três: desenvolvimento iterativo, planejamento (baseado em módulos) e revisões de grupo voltadas para o cliente.

O problema mais significativo com ASD é que suas práticas são difíceis de identificar e alguns detalhes das práticas não possuem uma origem específica (ABRAHAMSSON *et al*, 2002).

## 2.4 Comparação entre Métodos Ágeis

Cada método ágil possui características que influenciam no funcionamento e no desenvolvimento do projeto de *software*. Algumas características podem ser encontradas em vários métodos e outras são específicas para cada um.

A seguir serão apresentados três quadros. No Quadro 2.1 apresenta-se um estudo comparativo dos métodos ágeis, realizado por Abrahamsson *et al* (2002), apontando os pontos chaves, as principais características e as falhas.

No Quadro 2.2 são apresentadas as fases do processo de desenvolvimento de *software* e as práticas dos métodos ágeis aplicadas em cada fase. Esse quadro foi criado seguindo o processo de desenvolvimento de *software* de cada método, apresentando as informações inerentes a cada fase do ciclo de vida.

No Quadro 2.3, apresentam-se as técnicas e os documentos produzidos em cada fase do desenvolvimento do *software* dos métodos ágeis estudados. Esse quadro foi criado com base no estudo realizado sobre os métodos ágeis e nos trabalhos propostos por Nawrocki (2002), Paetsch *et al* (2003) e Horrian *et al* (2003).

A idéia de construir o Quadro 2.3 objetiva facilitar a definição da documentação que deve ser elaborada durante a aplicação do processo de desenvolvimento criado neste trabalho (Capítulo 6), com o objetivo de apoiar a manutenção, no desenvolvimento do *software* e após o desenvolvimento. Para cada fase são apresentados as técnicas e os documentos produzidos.

Uma das técnicas apresentadas por Paetsch *et al* (2003) para a fase de levantamento de requisitos pode ser atribuída a todos os métodos ágeis, como a utilização de entrevistas, que é classificada por esse autor em dois tipos: uma com perguntas específicas para determinadas pessoas e outra sem perguntas definidas e realizada em grupos. Para a documentação da entrevista, Nawrocki (2002) afirma que todas as informações e documentos criados junto ao cliente devem ser transferidos para um formulário eletrônico. Já para as outras fases do desenvolvimento do *software*, observou-se que as técnicas utilizadas e os documentos produzidos são específicos para cada método ágil.

Desta forma, os quadros facilitam o entendimento do funcionamento dos métodos ágeis, das práticas utilizadas e dos documentos produzidos.

Quadro 2.1: Estudo dos métodos ágeis (adaptado de Abrahamsson *et al*, 2002)

| Métodos | Pontos Chaves | Principais características | Falhas |
|---|---|---|---|
| **XP** | Desenvolvimento dirigido pelo cliente, equipes pequenas e versões freqüentes. | Refatoração do sistema melhora o desempenho e é responsável pelas mudanças. | Pouca atenção no uso de prática de gerenciamento. |
| **Scrum** | Pequeno, auto-organizável, ciclo de desenvolvimento de 30 dias. | Visão do produto bem definida e repetível. | Falta de testes de integração e de aceitação no ciclo de desenvolvimento. |
| **FDD** | Formado por cinco processos e iterações curtas. | Método simples, desenvolvimento por características e modelagem de objeto. | Foco somente no projeto e na implementação. |
| **Crystal** | Vários métodos com características diferentes. | Capacidade de selecionar o método mais adaptável ao projeto. | Dificuldade no uso de estimativas. |
| **DSDM** | Uso do RAD, equipe com autonomia para tomar decisões. | Utiliza a prototipação e possui vários papéis (responsável) para execução de uma atividade no método. | Somente os membros da equipe têm acesso aos procedimentos do método. |
| **ASD** | Foca no ciclo adaptável, colaborativo e no desenvolvimento iterativo. | Sistemas adaptativos. | Baseia-se mais nos conceitos e na cultura do que em práticas ágeis. |

Quadro 2.2: Técnicas e ráticas utilizadas em cada fase do ciclo de vida do *software*

| Métodos | Levantamento de Requisitos | Análise | Projeto | Implementação | Teste | Implantação | Mudanças no desenvolvimento |
|---|---|---|---|---|---|---|---|
| **XP** | Executa por meio de estórias criadas pelo cliente com base na funcionalidade do novo sistema. | Baseia-se nas estórias escritas pelo cliente. | Utiliza a prática "Jogo do Planejamento". | Motiva que seja em pares de programadores. | Utiliza teste de unidade e de aceitação. Os testes de unidade são criados antes do código e utilizados em todo o projeto. | Utiliza a prática "Versões Freqüentes" com *software* funcionando. | Utiliza a prática "Refatoração". |
| **Scrum** | Elabora uma lista informal das prováveis funcionalidades. | Define a lista de *Backlog* do produto. | Organiza a lista de *Backlog* conforme o nível de prioridade exigido pelo cliente. | Inicia o *Sprints.* | Executa no ciclo do *Sprints* na fase de desenvolvimento. | Executa na fase de pós-planejamento. | Não explícito. |
| **FDD** | Define no primeiro processo "Desenvolver um modelo abrangente". | Elabora no segundo processo "Construir uma lista de características". | Define nos processos "Planejar por características" e "Projetar por características". | Executa no processo "Construir por característica". | Testes de unidades são criados e aplicados no desenvolvimento das características no processo "Construir por característica". | Executa no processo "Construir por características". | Não explícito |
| **Crystal Orange** | Define por meio de entrevista com os usuários (HORRIAN *et. al*, 2003). | Não explícito. | Utiliza ferramenta para acompanhamento de projeto. | Utilizam algumas ferramentas para controle de versão, programação e comunicação. | Aplica a atividade de verificação/teste e usa teste de aceitação e teste de regressão e utiliza ferramenta de teste. | Não explícito. | Não explícito. |
| **Crystal Clear** | Define por meio de entrevista com os usuários Horrian *et. al* (2003) | Utilizam quadro branco visível a toda equipe e diagrama de caso de uso. | Utiliza ferramenta para acompanhamento de projeto. | Utiliza algumas ferramentas para compilar o código fonte, gerar versões e controlar versões. | Utiliza teste de regressão e teste de aceitação. | Não explícito. | Não explícito. |
| **DSDM** | Define na fase do estudo de viabilidade. | Elabora na fase do estudo do negócio. | Define na fase do estudo de iteração funcional. | Executa na fase de construção de iteração. | Aplica teste funcional e de aceitação na fase de construção de iteração. | Executa na fase de implementação após a aprovação do usuário. | Não explícito. |
| **ASD** | Define na fase de especulação. | Elabora na fase de especulação. | Define na fase de colaboração. | Executa na fase de colaboração. | Executa na fase de aprendizado. | Libera versões pequenas em um curto período. | Não explícito. |

Quadro 2.3: Técnicas Utilizadas/Documentos Produzidos.

| Métodos | Levantamento de Requisitos | Análise | Projeto | Implementação | Teste | Implantação | Mudanças no desenvolvimento |
|---|---|---|---|---|---|---|---|
| **XP** | Entrevistas/ Formulário preenchido. | Cartão de estórias/Diagrama Classes. | Jogo do planejamento/Lista de estórias priorizadas definidas pelo cliente. | Projeto simples/Código fonte. | Teste de unidade e aceitação/Documentação dos casos de teste. | Versões freqüentes/ *Software* executável. | Realizada por meio do código fonte e casos de teste/ -. |
| **Scrum** | Entrevistas/ Formulário preenchido. | Lista de *Backlog*/Documentação da lista *Backlog* | Reunião Scrum diárias/Lista de Backlog. | *Sprint*/Código fonte. | Não explícito. | Fim do *Sprint*/ *Software* executável. | Revisão/Lista de *Backlog* atualizada. |
| **FDD** | Entrevistas/ Formulário preenchido. | Diagrama de casos de usos ou especificações funcionais/Documentação dos diagramas. | Diagrama de seqüência e de classes/Documentação dos diagramas. | Implementação das classes/Código fonte. | Teste de unidade aplicado nas classes/Documentação dos casos de teste. | Fechamento da versão/ *Software* executável. | Não explícito. |
| **Crystal Orange** | Entrevistas/ Formulário preenchido. | Diagrama de casos de uso/Documentação do diagrama. | Usa o documento de requisito /Documenta outros requisitos e os não funcionais. | Nenhuma/Código fonte. | Teste de regressão/Documentação dos casos de teste. | Não explícito. | Não explícito. |
| **Crystal Clear** | Entrevistas/ Formulário preenchido. | Diagrama de casos de uso/Documentação do diagrama. | Usa o documento de requisito/Documenta outros requisitos e os não funcionais | Nenhuma /Código fonte. | Teste de regressão/Documentação dos casos de teste | Não explícito. | Não explícito. |
| **DSDM** | Entrevistas/ Formulário preenchido. | Planejamento/ Plano de projeto. | Planejamento/ Plano de projeto do protótipo | Aprovação do Protótipo/Diretrizes. | Teste Funcional/Documentação dos casos de teste. | Não explícito. | Não explícito. |
| **ASD** | Entrevistas/ Formulário preenchido. | Planejamento/Protótipo funcional baseado em módulos. | Gerência de Projeto/Documentação do projeto. | Baseado em módulos/Código-fonte. | Não explícito. | Liberação de Versão/ *Software* executável. | Tolerante a mudanças/ -. |

## 2.5 Trabalhos Correlatos

Alguns trabalhos relacionados a processos ágeis de desenvolvimento são discutidos nesta seção.

No trabalho proposto por Zwartjes *et al* (2005), foi definido um processo ágil de desenvolvimento para componentes. Nesse trabalho, os autores utilizaram como base o padrão de Engenharia de *Software* ESA[11] (Agência Espacial Européia). Segundo Zwartjes *et al* (2005), a utilização desse padrão baseou-se na experiência em outros projetos. O objetivo principal do trabalho proposto por Zwartjes *et al* (2005) é aumentar a produtividade e diminuir a documentação. Algumas modificações foram criadas para tornar o processo ágil, como a eliminação da documentação utilizada na gerência do projeto e a utilização de algumas práticas de métodos ágeis. O processo atende às características dos métodos ágeis, sendo: iterativo, incremental, adaptativo, formado por equipes pequenas e permite a interação entre clientes e os integrantes da equipe de desenvolvimento.

Para a definição do processo, Zwartjes *et al* (2005) conduziram estudos de casos e definiram o processo em quatro fases, sendo: exploração, projeto, construção e manutenção.

Zwartjes *et al* (2005) concluíram que um processo ágil para desenvolvimento de componentes é mais barato e mais eficiente, desde que haja investimento para treinar a equipe de trabalho, e mais produtivo, quando comparado com os métodos tradicionais.

O XwebProcess (SAMPAIO *et al*; 2004) é um processo ágil para desenvolvimento de aplicações *web*, baseado no método ágil XP e documentado por meio do meta-modelo

[11] http://esapub.esrin.esait/bulletin/bullet90/b90jones.htm

SPEM[12] (*Software Process Engineering Metamodel*), que é utilizado para descrever processos de desenvolvimento de *software*.

Algumas características do processo XwebProcess são: permite a participação do cliente; descreve o projeto e a aplicação de teste *web* para garantir a qualidade do produto desenvolvido; libera versões o mais rápido possível, por meio da comunicação entre o cliente e a equipe de desenvolvimento e com o uso de técnicas de prototipação; e possui uma estrutura de fácil compreensão.

Alguns resultados comparados com o método XP e XWebProcess em um estudo de caso de desenvolvimento *web,* demonstraram que este último produz documentação suficiente e favorece a comunicação em equipe (SAMPAIO *et al*; 2004).

Um outro processo ágil de desenvolvimento de *software* encontrado na literatura é o easYProcess (DANTA *et al*, 2004), que é apoiado por práticas ágeis do arcabouço RUP (*Rational Unified Process*) Kurchten (2000), do método XP e da Modelagem Ágil (AMBLER, 2004a). Foi elaborado em ambiente acadêmico com o objetivo de facilitar o aprendizado da disciplina de Engenharia de *Software,* do curso de Ciência da Computação da Universidade Federal de Campina Grande, no estado da Paraíba.

As características principais do processo easYProcess (DANTA *et al*, 2004) são: participação do cliente, desempenho de diversos papéis por uma única pessoa, liberação de *releases* e execução de iterações curtas de acordo com o ambiente acadêmico, aplicação de testes de práticas ágeis durante o projeto e criação de repositório de código para controle de versão. O easYProcess também é denominado de YP e utiliza várias ferramentas livres como exemplo: Xplanner[13] utilizada para gerência de projetos, Eclipse[14] para o desenvolvimento em Java, JUnit[15] para implementação de teste de unidade, entre outras.

[12] http://www.omg.org/technology/documents/formal/spem.htm
[13] http://www.xplanner.org/
[14] http://www.eclipse.org/
[15] http://www.junit.org/index.htm

Um outro processo ágil é o processo de reengenharia PARFAIT (CAGNIN, 2005a), que migra sistemas legados procedimentais para o paradigma orientado a objetos. Para isso utiliza *frameworks* baseados em linguagens de padrões de análise para apoiar na construção do sistema alvo, sendo possível obter a documentação de análise do sistema legado bem como o entendimento do domínio ao qual pertence, a partir da aplicação da linguagem de padrões. O processo PARFAIT, que é de interesse deste trabalho, será apresentado no Capítulo 4, na Seção 4.3.

## 2.6 Considerações Finais

Neste capítulo foram apresentadas algumas definições sobre processo de desenvolvimento de *software*, enfatizando a maneira como o processo deve ser conduzido, juntamente com os recursos necessários para o desenvolvimento do *software.* Apresentaram-se a origem da formação da Aliança Ágil, seus princípios e as práticas ágeis comuns a todos os métodos ágeis. Alguns métodos ágeis foram apresentados, abordando seu processo de desenvolvimento e as práticas existentes em cada método. Além disso, foram apresentados também processos de desenvolvimento ágil encontrados na literatura.

No próximo capítulo apresentam-se algumas abordagens de reúso: padrão de *software*, linguagens de padrões e *frameworks*.

# CAPÍTULO 3. PADRÃO DE *SOFTWARE* E *FRAMEWORK*

## 3.1 Considerações Iniciais

Neste capítulo apresentam-se algumas técnicas de reúso existentes na literatura. Na Seção 3.2 apresentam-se os conceitos de padrões de *software* e linguagens de padrões, além de exemplos de algumas linguagens de padrões que pertencem ao domínio de Sistemas de Informação, que é de interesse deste trabalho. Na Seção 3.3 apresentam-se diversas definições de *framework* encontradas na literatura e as características de alguns *frameworks* de aplicação. E na Seção 3.4 apresentam-se as considerações finais deste capítulo.

## 3.2. Padrão de *Software*

A origem de padrões é proveniente de um trabalho de Christopher Alexander, um arquiteto que desenvolveu um conjunto de teorias sobre padrões com a idéia de permitir que as pessoas projetassem suas próprias casas e prédios (CHRISTOPHER, 1999). Na década de 90 essa idéia foi utilizada na Engenharia de *Software,* na área de desenvolvimento de *software,* com o objetivo de criar padrões para solucionar um problema particular seguindo uma determinada estrutura. Vários trabalhos foram realizados na Engenharia de *Software*, utilizando o mesmo padrão de especificação proposto por Christopher Alexander, apresentado em seu livro *A Pattern Language* (1977). Os padrões propostos por Christopher Alexander foram refinados, influenciando a criação de padrões análogos para análise e projeto orientado a objetos (MATTSSON, 1996).

De acordo com Jacobson *et al* (1997), padrões são baseados nos princípios básicos de um modelo. Esses modelos são descrições utilizadas para resolver problemas específicos dentro de um determinado contexto. Cada padrão fornece a solução para um problema e são utilizados em diferentes estágios do desenvolvimento de sistema.

Uma característica relatada por Devedzic (1998) é em relação à facilidade de reúso fornecida pelos padrões, que compartilham modelos e conhecimento do projeto, permitindo sua adaptação em problemas específicos.

Ainda no contexto da característica de reúso, Braga (2003) classifica padrões em diversas categorias, sendo: padrões de análise, padrões de projeto, padrões de processo, padrões arquiteturais e afirma que esta não é uma classificação original, pois pode haver padrões que se enquadram em mais de uma categoria.

Padrões de análise descrevem os modelos do processo do negócio na fase de análise do desenvolvimento do *software*. Já padrões de projeto baseiam-se na solução de um problema específico de projeto, descrito pela interação de objetos e classes (JACOBSON *et. al*, 1997). Padrões de processo são formados por soluções para problemas nos processos envolvidos na Engenharia de *Software* e padrões arquiteturais definem a organização estrutural fundamental de sistemas de *software* ou *hardware* (BRAGA, 2003).

Seguindo o contexto de reutilização, uma caracterização do uso de padrões relatada por Jacobson *et al* (1997) é no desenvolvimento de aplicações e em *frameworks.* Uma aplicação corresponde a um sistema desenvolvido e o *framework* a maneira como é gerada essa aplicação. Um estudo de *frameworks* é apresentado na Seção 3.3.

Com base nessa reutilização houve a necessidade de um estudo de padrões e linguagens de padrões, pelo fato deste trabalho utilizar como apoio computacional no processo de desenvolvimento *frameworks,* cuja construção foi baseada em linguagem de padrões de análise.

### 3.2.1. Exemplos de Padrões de *Software*

Fowler (1997) apresenta vários padrões de análise, os quais abordam a estrutura conceitual do processo de negócios de um *software* ao invés da sua implementação. Os padrões são baseados na experiência pessoal de aplicação da modelagem orientada a objetos. Alguns dos padrões definidos por Fowler (1997) são: *Accountability* que define responsabilidades entre partes, como por exemplo, contratos formais ou informais, o *Observation* utilizado para tratar informações qualitativas dos objetos, já o padrão *Measurement* trata informações quantitativas dos objetos e os padrões *Inventory* e o *Accounting* são utilizados para descrever como uma rede de contas e regras de lançamento pode formar um sistema de contabilidade.

Os padrões propostos por Souza *et al* (2005), são padrões de requisitos para Sistemas de Informação, fundamentados no conceito de caso de usos. Cada padrão aborda soluções de problemas na especificação de casos de uso. Os padrões definidos são: casos de uso CRUD (*Create*, *Read*, *Update* e *Delete*), caso de uso transação, casos de uso relatórios e caso de uso assistente. O objetivo do padrão caso de uso CRUD é documentar requisitos, por meio de modelos e especificações de casos de uso. Para cada padrão são definidos o contexto, o problema, as forças, a solução, a estrutura dividida em fluxo básico, subfluxo e conseqüências.

O padrão de projeto *AgregaComponente* definido por Freitas *et al* (2005), tem como objetivo facilitar o acoplamento de novas funcionalidades em sistemas baseados no padrão arquitetural *Layer* (BUSCHMANN *et al*, 1999). O *AgregaComponente* é utilizado nas situações em que uma nova funcionalidade deve ser adicionada em um sistema em camadas, no caso em que o fluxo de requisições do sistema for tratado pelo padrão de projeto *Command*

(GAMMA *et al*, 2000) e quando o sistema organizado em camadas, possuir módulos desenvolvidos por pessoas distintas.

Freitas *et al* (2005) citam algumas vantagens e desvantagens do padrão *AgregaComponente*. Dentre as vantagens têm-se: facilidade na identificação e definição do problema, no qual auxilia a tarefa de adicionar componentes ao sistema e na expansão do sistema. Uma desvantagem é em utilizar o padrão é que o desenvolvedor deve possuir também conhecimento do padrão *Command* para tratar a conexão entre a interface gráfica e a lógica de negócio do sistema.

O padrão de arquitetura MVC (*Model View Controller*) (KRASNER e POPE, 1998) tem como objetivo separar a camada de apresentação – *View* da camada de negócio – *Model* e do fluxo de aplicação - *Controller*, permitindo que a mesma camada de negocio possa ser utilizada por várias interfaces (*View)*. O padrão MVC é muito utilizado em domínio de aplicações *web*, com objetivo de mapear as tarefas de entrada, processamento e saída, para o modelo de interação com o usuário. Uma das vantagens do padrão de arquitetura MVC é separar os conceitos de apresentação, controle de fluxo e lógica de negócios, o que diminui a duplicação de código, tornando a aplicação mais robusta, portável e de fácil manutenção (RIEHLE, 1997).

### 3.2.2. Linguagem de Padrões

Uma linguagem de padrões é formada por uma coleção estruturada de padrões, que se relacionam entre si, podendo ser utilizados para transformar requisitos e restrições em uma arquitetura (COPLIEN, 1998). Segundo Schmidt *et al* (1996), uma linguagem de padrões é uma linguagem formal, pois é formada por um conjunto de padrões relacionados para

solucionar um problema particular. Os dois autores descrevem a formação de uma linguagem de padrões por meio de padrões que se relacionam entre si. Schmidt *et al* (1996) apontam a utilização de uma linguagem de padrões para resolver um problema específico.

Uma vantagem apresentada por Mattsson (1996) é a facilidade que as linguagens de padrões têm em ensinar o projetista como projetar, criando construções flexíveis e de fácil integração a novos padrões. Já Coplien (1998) afirma que o uso de linguagens de padrões pelo projetista evita a construção de arquiteturas ruins. Essa afirmação ocorre porque as linguagens de padrões fornecerem uma estrutura, que guia o projetista numa seqüência de vários passos até chegar ao fim, inibindo-o de eliminar passos obrigatórios definidos na linguagem e importantes para o projeto.

Uma das utilizações das linguagens de padrões está na construção de *frameworks.* Segundo Braga (2003), quando um *framework* utiliza uma linguagem de padrões, é possível saber o que encontrar no *framework*, tornando fácil seu entendimento e sua manutenção.

Algumas linguagens de padrões de análise serão apresentadas resumidamente, com um enfoque maior na linguagem de padrões GRN. A apresentação somente de linguagens de padrões de análise refere-se ao fato do *framework* GREN, utilizado neste trabalho, ser baseado em sua construção neste tipo de linguagem de padrões, sendo possível obter a documentação de análise do sistema bem como o entendimento do domínio ao qual pertence, a partir da aplicação da linguagem de padrões. O *framework* GREN será apresentado na próxima seção.

### 3.2.1.1 Exemplos de Linguagens de Padrões

A linguagem de padrão LV (Leilões Virtuais), proposta por Ré (2002), aplica-se ao desenvolvimento de sistemas para gestão de vendas por intermédio de leilões virtuais. Seu desenvolvimento foi baseado em três sistemas de leilões existentes na Internet, sendo: *eBay*, *iBazar* e *Arremate.com.* A linguagem é formada por dez padrões, divididos em duas categorias principais. A primeira categoria – Padrões Requeridos – representa requisitos essenciais para que um recurso possa ser leiloado e a segunda categoria – Padrões Opcionais – possui padrões desejáveis, mas seu uso não é obrigatório.

Outra linguagem de padrões de análise é a linguagem de padrões SiGCli (Sistema de Gerenciamento de Clínicas de Reabilitação), proposta por Pazin (2004), pertencente ao domínio de clínicas de reabilitação, permitindo o controle de atendimento a pacientes, controle de vendas e controle de compras. É composta por 9 padrões, sendo: Identificar Pacientes, Definir Serviços, Realizar Vendas, Processar Guias, Agendar Atendimento, Identificar Atendentes, Realizar Acompanhamento, Realizar Compras e Controlar Faturamento.

As linguagens de padrões LV e a SiGCli são consideradas extensões da linguagem de padrões GRN porque foram construídas seguindo o mesmo processo de elaboração e possuem alguns padrões semelhantes ao da linguagem de padrões GRN, que foram utilizados e adaptados para seu domínio. A linguagem de padrões GRN está descrita a seguir.

### 3.2.1.2 GRN

A GRN, proposta por Braga *et al* (2002), é uma linguagem de padrões de análise, utilizada no domínio de gestão de recursos de negócios. É formada por quinze padrões de análise, sendo que alguns deles são aplicações ou extensões de padrões de análise propostos na literatura. Os padrões podem ser usados para locação, comércio e manutenção de recursos. A palavra recurso refere-se a um bem ou serviço gerenciado pela aplicação, que define todas as suas características importantes.

A GRN oferece aos desenvolvedores inexperientes informações de análise suficientes para o desenvolvimento de novos sistemas e de soluções alternativas.

Os padrões principais desta linguagem são: LOCAR O RECURSO, COMERCIALIZAR O RECURSO e MANTER O RECURSO. Esses padrões podem ser utilizados em conjunto, dependendo do sistema a ser modelado (BRAGA *et al*, 2002).

O primeiro padrão a ser aplicado é IDENTIFICAR O RECURSO. Os padrões ITEMIZAR A TRANSAÇÃO DO RECURSO, PAGAR PELA TRANSAÇÃO DO RECURSO e IDENTIFICAR O EXECUTOR DA TRANSAÇÃO são mostrados dentro de uma caixa, denotando que são aplicáveis a todas as situações nas qual uma seta chega até a borda dessa caixa. A seta sem origem que chega ao padrão 11 significa que esse é o primeiro padrão a ser verificado, seguido dos padrões 12 e 13.

Na Figura 3.1 apresenta-se a estrutura dos padrões da GRN, bem como a seqüência que podem ser aplicados. Os padrões são divididos em três grupos. O **Grupo 1** possui padrões para Identificação de Recurso de Negócios. É formado pelos três primeiros padrões (1 a 3) usados para a identificação e possível qualificação, quantificação e armazenamento dos recursos gerenciados pelo negócio. O **Grupo 2** possui padrões para Transação de Negócios. É formado por quatro padrões, seguindo a numeração do 4 a 10 e são responsáveis

pela manipulação dos recursos de negócio. O **Grupo 3** possui padrões dos Detalhes da Transação de Negócio. Contém cinco padrões seguindo a numeração 11 a 15 e são responsáveis pelos detalhes das transações efetuadas com o recurso.

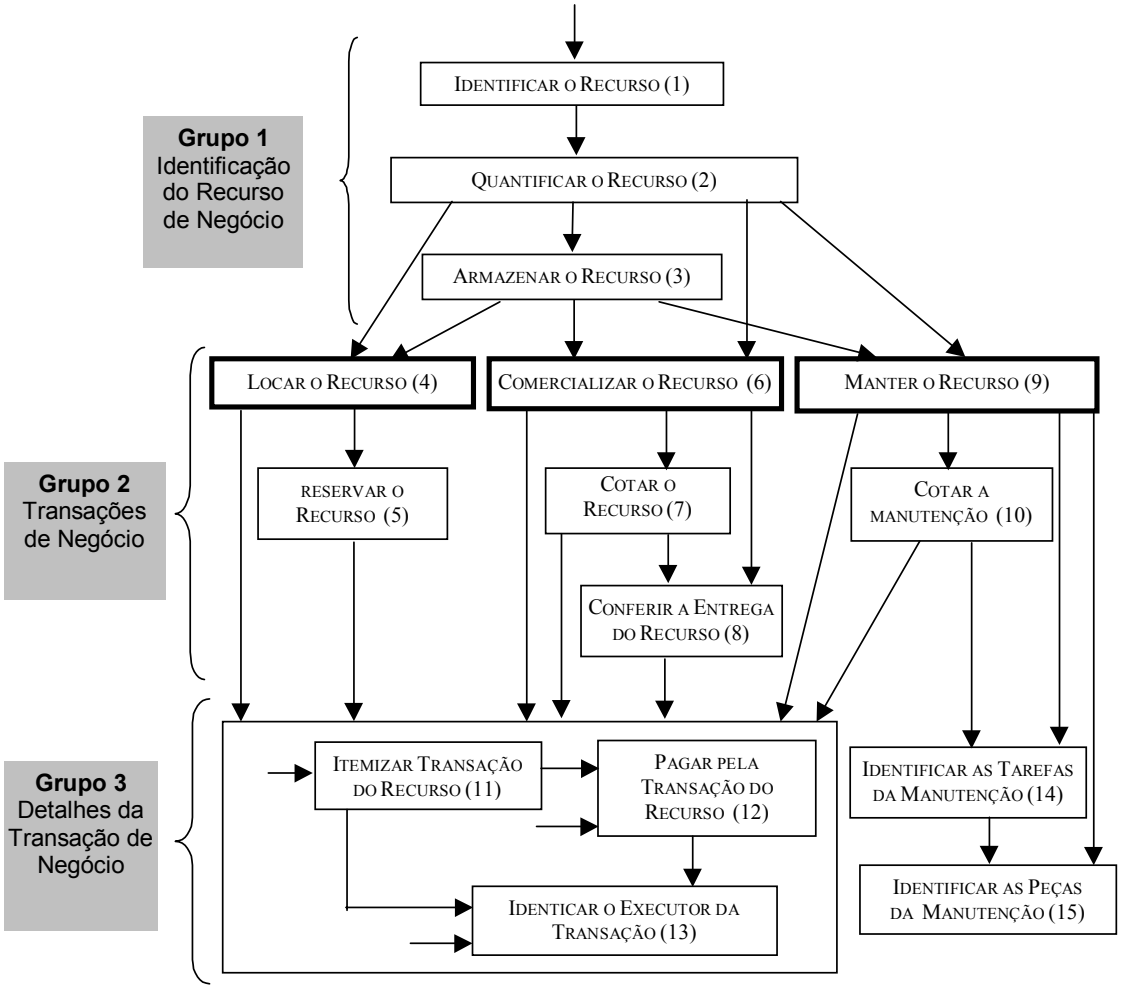


Figura 3.1: Estrutura da linguagem GRN (BRAGA *et al*, 2002)

## 3.3 *Frameworks*

Diversas definições de *frameworks* são encontradas na literatura. De acordo com Taligent (1997), *framework* é uma estrutura formada por blocos pré-fabricados de *software* que os programadores podem usar, estender ou adaptar para uma solução específica. Bosch *et al* (1999) afirmam que *framework* é um conjunto de classes que incorporam um projeto

abstrato e Johnson (1992) complementa que é uma estrutura reusável. Appleton (1997) também define *framework* como uma arquitetura reusável em que fornece estrutura e comportamento comuns para uma família de abstrações de *software*.

Os autores descrevem que *framework* são usados para soluções de problemas em um determinado domínio. Já Mattsson (1996) apresenta *framework* como um tipo de arquitetura reusável de *software* que compreende o projeto e o código.

Dentre as definições apresentadas, Mattsson enfatiza uma característica para o *framework*, como sendo orientado a objetos. As outras definições abordam a estrutura do *framework* de uma maneira mais genérica.

Uma outra característica de *framework,* de acordo com Pree *et al* (1995) e Braga (2003), é o reúso que pode ser fornecido não somente pelo código fonte do sistema, mas também pela arquitetura do projeto.

Para a geração de um sistema com o uso de um *framework,* é necessário que este seja instanciado. De acordo com Booch *et al* (2000) a definição da palavra instância refere-se a uma manifestação concreta de uma abstração, ou seja, abstrai-se somente o que for necessário, ignorando o restante e no final obtêm-se um produto de acordo com a abstração.

Segundo Braga (2003), para a instanciação de *frameworks* o engenheiro de *software* deve possuir conhecimento considerável sobre o projeto e implementação do *framework.* Tal conhecimento facilitará a construção de *software* com menor quantidade de código escrito possível, além de fornecer boas soluções que possam ser reutilizadas dentro de um mesmo domínio do *software*.

*Frameworks* permitem especializações, ou seja, adaptações que são chamadas de *hot spots*. *Hot Spots* são as partes variáveis do *framework*, contendo componentes que podem ser adaptados. As partes fixas são chamadas de *frozen spots* e representam a arquitetura geral de uma aplicação, seus componentes básicos e os relacionamentos entre eles. Nas instanciações

do *framework* as partes fixas não são alteradas, mas as partes variáveis podem ser alteradas de acordo com as necessidades do sistema que está sendo desenvolvido.No mercado existem diversos tipos de *frameworks* para diversas situações. De acordo com essa diversidade, Fayad e Schmidt (1997) classificam *frameworks* em três grupos, com base em seu escopo:

- *Frameworks* de infra-estrutura: simplificam o desenvolvimento da infra-estrutura de sistemas portáveis e eficientes, como por exemplo, sistemas operacionais, sistemas de comunicação, entre outros.
- *Frameworks* de integração-*middleware*: são usados em geral para integrar aplicações e componentes distribuídos. Eles são projetados para melhorar a habilidade dos desenvolvedores em modularizar, reutilizar e estender sua infra-estrutura de *software* para funcionar em um ambiente distribuído.
- *Frameworks* de aplicação empresarial: tem como objetivo gerar aplicações de acordo com o seu domínio, sendo flexíveis para atender aos requisitos específicos do domínio. Geralmente são voltados para a área de negócio, são pagos e possuem um custo alto.

O processo ágil de desenvolvimento apresentado neste trabalho é baseado em *frameworks* de aplicação. O *framework* utilizado é o *framework* GREN porque pertence ao nível acadêmico, sendo de fácil acesso; pela sua construção ser baseada em uma linguagem de padrões de análise que atende ao domínio de Sistemas de Informação, e do seu uso pelo processo PARFAIT, que é utilizado neste trabalho com base para a definição do processo de desenvolvimento e está apresentado no Capítulo 4, na Seção 4.3. Alguns *frameworks* são apresentados na Seção 3.3.1, como GREN (BRAGA, 2003), IBM SanFrancisco (BOHER, 1998) e Qd+ (RÉ, 2002).

Uma outra classificação atribuída aos *frameworks* é com relação a sua extensão (FAYAD e JOHNSON, 2000). Os *frameworks* são classificados em: caixa branca, caixa preta

e caixa cinza. Os *frameworks* caixa branca possuem características de linguagens de programação orientadas a objetos, permitindo sua extensão por meio de herança, ou seja, a criação de classes concretas a partir de classes abstratas e de ligação dinâmica; *frameworks* caixa preta baseiam-se na composição de objetos, definindo interfaces com os componentes que podem ser conectados ao *framework* por meio da composição de objetos. Já os *frameworks* caixa cinza são uma mistura de *frameworks* caixa branca e caixa preta, permitindo a extensão por meio de herança e composição.

Com o uso de *frameworks* espera-se aumentar a produtividade na Engenharia de *Software* (MATTSSON, 1996). Essa afirmação está relacionada à agilidade dos *frameworks* em proporcionar versões do *software* em um período menor de desenvolvimento, diminuindo os prazos e, conseqüentemente, os custos.

### 3.3.1 Exemplos de *Frameworks*

Alguns *frameworks* de aplicação são descritos nesta seção, como: o *framework* GREN (BRAGA, 2003), o IBM SanFrancisco (MONDAY *et al*, 2000) e o Qd+ (RE, 2002). O *framework* GREN e o *framework* QD+ possuem a estrutura baseada em uma linguagem de padrões de análise, ao contrário do *framework* IBM SanFrancisco que é formado por um conjunto de componentes. Cada um dos *framework* possui um subdomínio específico, mas todos pertencem ao domínio de Sistemas de Informação.

### 3.3.1.1 GREN

O *framework* GREN foi construído com base na linguagem de padrões GRN (Gestão de Recursos e Negócios) (BRAGA, 2003) e desenvolvido na linguagem de programação Smalltalk. Esse *framework* cria sistemas pertencentes ao domínio de gestão de negócios, que incluam locação, compra, venda ou manutenção de bens ou serviços.

As classes e os relacionamentos contidos em cada padrão da GRN possuem implementação correspondente na hierarquia de classes do GREN (CAGNIN, 2005a). O *framework* GREN é do tipo caixa cinza e o armazenamento dos objetos é realizado no Sistema Gerenciador de Banco de Dados (SGBD) MySQL[16].

GREN possui uma arquitetura dividida em três camadas: a de persistência, a de negócios e a de interface com os usuários (engenheiro de *software*). A camada de persistência trata a conexão com o banco de dados e a persistência dos objetos em Smalltalk para o SGBD MySQL. A camada de negócio trata a implementação das classes e relacionamentos de cada padrão da linguagem de padrões GRN, comunicando-se com a camada de persistência. A camada de interface gráfica com o usuário trata da entrada e saída de dados, permitindo a interação com o usuário final.

Uma outra camada definida é a GREN-Wizard, que gera o código da aplicação a partir da especificação baseada nos padrões da GRN e está localizada acima da camada do GREN, pois utiliza todas as outras camadas. Após a aplicação da linguagem de padrões é obtido o diagrama de classes da aplicação e, com base nesse diagrama, utilizam-se classes pré-programadas do *framework* GREN para gerar o código do *software* sendo construído.

Para facilitar a instanciação do *framework* GREN, Braga (2003) criou uma ferramenta chamada GREN-Wizard que possui o mesmo nome da última camada da

[16] http://www.mysql.com

arquitetura do GREN, anteriormente mencionada. A ferramenta GREN-Wizard é considerada como um gerador de aplicações, pois gera classes e métodos a partir da especificação de uma aplicação, criada por meio dos padrões da GRN utilizados para modelá-la (CAGNIN, 2005a).

O objetivo da GREN-Wizard é facilitar o entendimento para utilizar o *framework* e automatizar os passos da instanciação do *framework*. Na Figura 3.2 apresenta-se a arquitetura do GREN, conforme descrito.

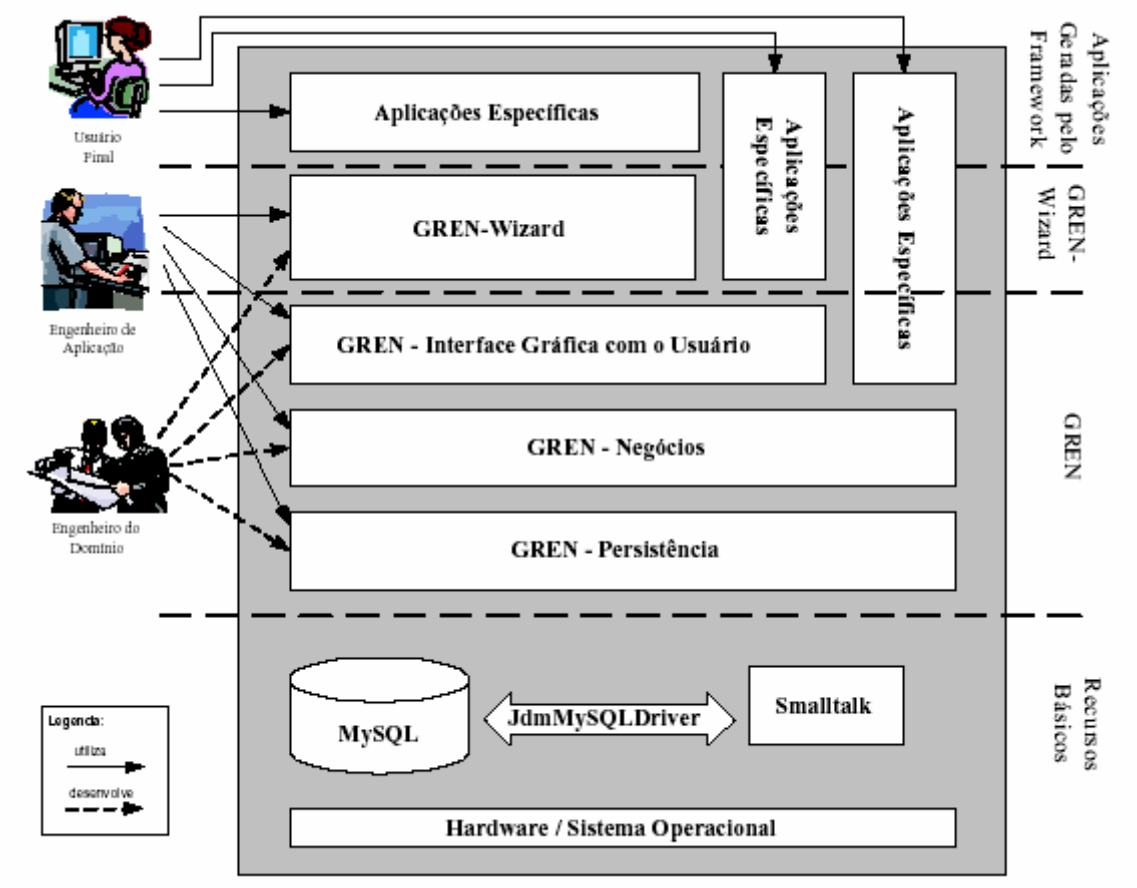


Figura 3.2: Arquitetura do *framework* GREN (BRAGA, 2003)

### 3.3.1.2 IBM SanFrancisco

IBM SanFranciso (*SanFranciso Component Framework*) é formado por uma coleção de componentes, desenvolvidos na linguagem Java, o que permite a independência de plataforma (MONDAY *et al*, 2000). Tem como objetivo construir aplicações de negócio, fornecendo um modelo orientado a objetos integrado ao domínio. Este *framework* é formado por três camadas.

A camada mais específica do SanFrancisco é denominada "Processos de Negócios Centrais" e trata dos principais processos e classes de um domínio particular. A camada intermediária, denominada "Objetos de Negócio Comuns", consiste de classes, processos e mecanismos comuns a vários domínios de aplicação, divididos em três grupos principais: objetos gerais de negócio, objetos financeiros de negócios e mecanismo generalizado ou padrões.

Nas partes intermediárias do SanFrancisco são fornecidas partes independentes que podem ser escolhidas e utilizadas de maneira similar a uma biblioteca de componentes, desde que seja respeitado o relacionamento entre processos e classes do *framework*.

A camada inferior é denominada "Fundação e Utilitária", fornecendo os serviços básicos necessários a uma aplicação de negócio. Sua classificação quanto à extensão é do tipo caixa cinza (MONDAY *et al*, 2000). Na Figura 3.3 apresenta-se a arquitetura completa do IBM SanFrancisco.

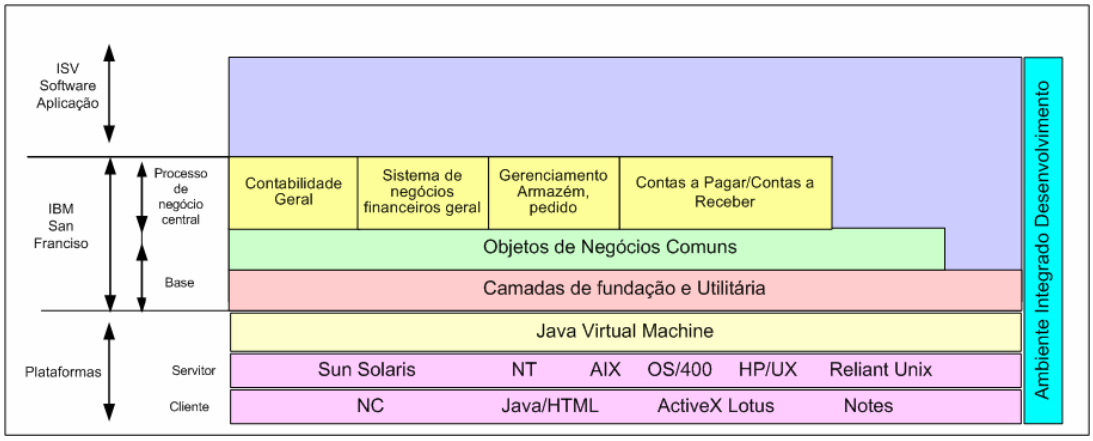


Figura 3.3: Arquitetura do *framework* IBM SanFrancisco (adaptada de MONDAY *et al*, 2000)

### 3.3.1.3 Qd+

O *framework* Qd+ foi baseado em sua construção na linguagem de padrões de análise LV (Leilões Virtuais). A linguagem de padrões é utilizada para capturar o domínio da aplicação e guiar o desenvolvimento do *framework*, facilitando também o aprendizado (RÉ, 2002). Por meio da instanciação do *framework,* obtêm-se sistemas no domínio de leilões virtuais.

Qd+ possui uma interface *Web* e arquitetura em três camadas. A arquitetura possui um navegador na máquina cliente, que corresponde à camada de apresentação; um servidor de *Web* ou servidor HTTP, que corresponde à camada de aplicação; e um servidor de banco de dados, que corresponde à camada de persistência.

Foi desenvolvido na linguagem de programação Smalltalk e utiliza o SGBD MySQL, responsável pelo armazenamento físico dos objetos do domínio em uma base de dados permanente. De acordo com Ré (2002), caso exista a necessidade de adição, criação ou a destruição de algum objeto armazenado, essa atividade deve ser realizada na camada de persistência. Sua classificação quanto à extensão é do tipo caixa branca.

O *framework* Qd+ é considerado uma extensão do GREN, pois possui algumas abordagens semelhantes, como: o processo de construção e instanciação de *frameworks* baseado numa linguagem de padrões de análise é o mesmo processo utilizado para construção do *framework* GREN, foi construído na mesma linguagem de programação do *framework* GREN e a linguagem de padrões LV, utilizada pelo *frameworks* Qd+, possui alguns padrões que foram adaptados da linguagem de padrões GRN pertencente ao *framework* GREN. Na Figura 3.4 apresenta-se a arquitetura do Qd+.

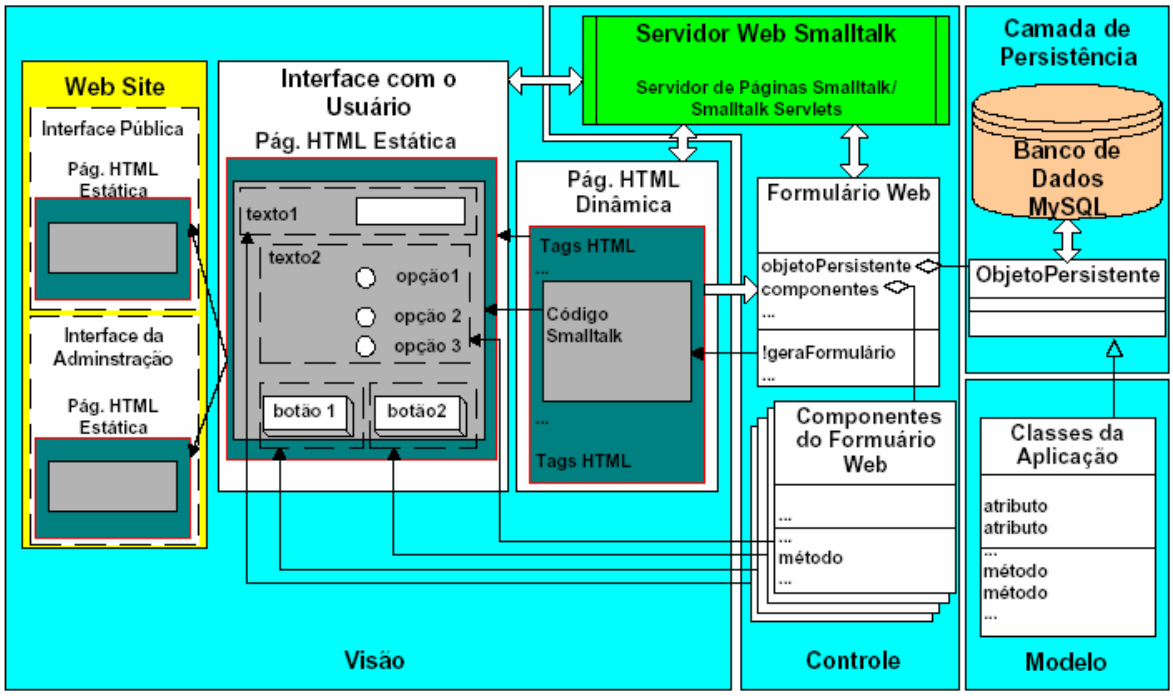


Figura 3.4: Arquitetura do *framework* Qd+ (RÉ, 2002)

## 3.4 Considerações Finais

Neste capítulo foram apresentadas algumas abordagens de reúso relacionadas ao trabalho como: a definição de padrão, seu uso e alguns tipos de padrões abordados por Braga

(2003). No contexto de padrão, apresentou-se a definição de linguagens de padrões, sua importância e alguns exemplos de linguagens de padrões de análise que atendem ao domínio de Sistemas de Informação.

Uma das utilizações de linguagens de padrões foi apresentada no uso para a construção de *frameworks.* Um estudo sobre *framework* foi apresentado, destacando suas características, os tipos existentes na literatura, sua classificação e alguns exemplos de *framework* de aplicação. Sua apresentação ocorre em decorrência do trabalho ser baseado no uso de *frameworks* como apoio computacional.

O próximo capítulo apresenta um arcabouço ágil utilizado na reengenharia de sistemas procedimentais para o paradigma orientado a objetos, com destaque para o processo PARFAIT, pertencente ao arcabouço, que foi utilizado no desenvolvimento deste trabalho.

# CAPÍTULO 4. ARCABOUÇO DE REENGENHARIA ÁGIL

## 4.1 Considerações Iniciais

Neste capítulo apresenta-se um arcabouço ágil utilizado para apoiar a reengenharia de *software* de sistemas procedimentais para o paradigma orientado a objetos, apoiado por ferramentas, processos, abordagens de reúso e práticas ágeis.

O capítulo está organizado da seguinte forma: na Seção 4.2 é apresentado o arcabouço ARA (Arcabouço de Reengenharia Ágil), na Seção 4.3 é mostrado o processo PARFAIT (**P**rocesso **Á**gil de **R**eengenharia baseado em **FrA**meworks no domínio de sistemas de **I**nformação com técnicas de VV&**T**), que é um dos elementos que compõe tal arcabouço, sendo muito importante seu entendimento neste trabalho, pois o processo apresentado no Capítulo 6, foi abstraído a partir dele. E, finalmente na Seção 4.4 as considerações finais deste capítulo são relatadas.

## 4.2.Arcabouço de Reengenharia Ágil (ARA)

Proposto por Cagnin (2005a), o ARA é um **A**rcabouço de **R**eengenharia **Á**gil baseado em *framework* que tem como objetivo migrar sistemas legados procedimentais para o paradigma orientado a objetos. O ARA pode também ser utilizado somente para realizar a engenharia reversa (CAGNIN *et al*, 2003c). Além disso, de acordo Cagnin (2005a), pode ser adaptado para apoiar o desenvolvimento ágil de *software,* constatado durante o desenvolvimento deste trabalho.

O ARA baseia-se na reengenharia incremental e a prioridade dos requisitos para o desenvolvimento do sistema alvo é estabelecida pelo usuário. Uma característica importante apontada por Cagnin (2005a) é a entrega de uma versão inicial do produto com qualidade em um curto espaço de tempo. Esse fato relaciona-se a agilidade do arcabouço, por possuir algumas práticas ágeis, como: permite a participação do cliente em todo o processo de reengenharia, libera versões rápidas do sistema alvo, incentiva programação em pares, pratica jogo do planejamento a cada iteração do processo, garante propriedade coletiva do código, incentiva jornada de trabalho de no máximo quarenta horas semanais, utiliza teste em toda reengenharia e usa metáfora por meio da linguagem de padrão de análise. Ainda no contexto de agilidade, um outro recurso coberto pelo ARA é o uso de *frameworks* baseados em linguagens de padrões de análise.

Na Figura 4.1 apresenta-se a estrutura do arcabouço ARA e todos os seus recursos disponíveis. Observando o lado esquerdo dessa figura tem-se a engenharia reversa que está representada na figura por um triângulo, formada por uma seqüência de passos que devem ser efetuados, representando a abstração realizada em um sistema existente até chegar na engenharia avante, que está representada no lado direito da figura. Nesse último caso, o produto gerado por meio da instanciação do *framework* em cada iteração do processo estabelecido no arcabouço vai sendo refinado até atender as expectativas do usuário do sistema legado. O usuário do sistema legado é representado no centro da figura oval, cercado por um ciclo incremental, formado por *framework* baseado em LPA (linguagem de padrões de análise), a própria LPA e a atividade de adaptação que deve ser realizada no sistema alvo. A LPA é utilizada no entendimento do domínio do sistema legado e também apóia a sua documentação para que se adequar às funcionalidades do sistema legado.

Ressalta-se que durante o uso do ARA na reengenharia não há delimitação das fases de engenharia reversa e engenharia avante devido à sua característica incremental e iterativa.

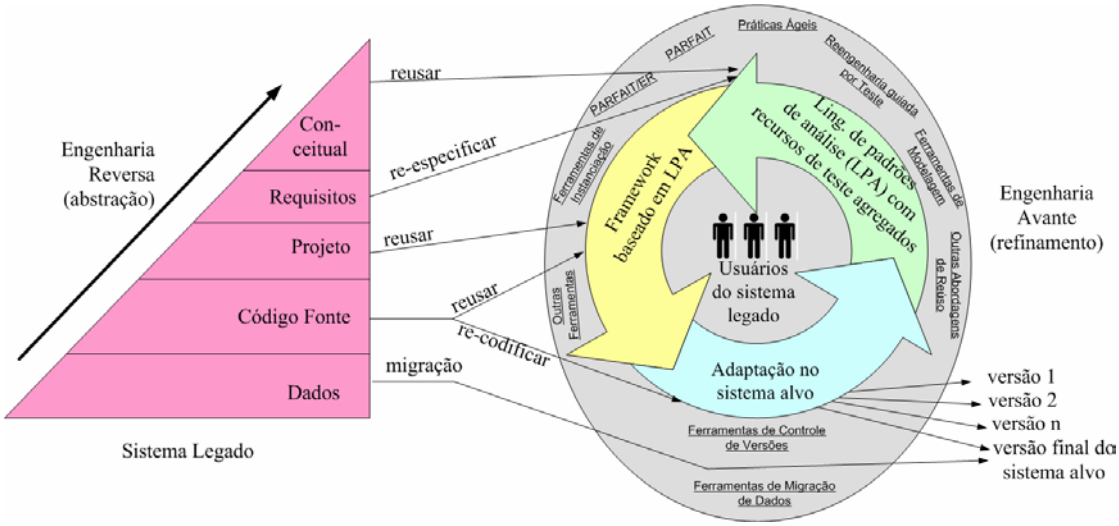


Figura 4.1: Estrutura do ARA (adaptada de CAGNIN, 2005a)

A base do triângulo é representada pelos dados (informações) existentes no sistema legado. Os próximos passos são: análise do **código fonte**, recuperação do **projeto** do sistema legado, identificação dos **requisitos** do sistema legado e entendimento do seu domínio (base **conceitual**), apoiados pela aplicação da linguagem de padrões de análise disponível. O entendimento do sistema legado e a identificação de regras de negócio também são apoiados pela execução de casos de teste, que são criados com o apoio de critérios de teste funcional e documentados para serem posteriormente utilizados durante a engenharia avante, a fim de validar as funcionalidades do sistema alvo. A essa prática dá-se o nome de reengenharia guiada por teste. Para executar os passos mencionados anteriormente é necessário interagir com vários recursos disponíveis, que estão apresentados na parte oval da figura, por exemplo: processos, abordagens de reúso, práticas ágeis, reengenharia guiada por teste, ferramentas de modelagem; ferramentas de controle de versões utilizadas para armazenar versões do sistema, devido ao ciclo ser incremental; ferramentas de migração de dados, utilizadas para popularizar a base de dados do sistema alvo a partir da base de dados do sistema legado; ferramentas de instanciação, que podem ser disponibilizadas juntamente com os *frameworks*, com o objetivo

de facilitar a geração do código fonte do sistema alvo e outras ferramentas que podem ser utilizadas conforme a necessidade do projeto.

A fase de implementação do sistema alvo reutiliza código por meio da instanciação do *framework* baseado em LPA disponível. Adaptações no código fonte do sistema alvo, obtido do *framework,* são realizadas a fim de implementar regras de negócio do sistema legado. Esse passo é apoiado por ferramentas de controle de versões. As versões do sistema alvo vão sendo liberadas para teste de aceitação conforme termina o ciclo incremental ou iterativo, representado na figura oval. Esse ciclo prossegue até gerar uma versão final do sistema alvo, que é implantada.

Uma das abordagens disponível no ARA é a abordagem de reúso de teste, denominada ARTe (CAGNIN, 2005a). Essa abordagem tem como objetivo associar requisitos de teste a padrões de linguagens de padrões de software para que sejam posteriormente reutilizados junto com a solução do padrão, a fim de reduzir o tempo da atividade de VV&T (Verificação, Validação e Teste). ARTe é composta por duas etapas. A Etapa 1 fornece uma estratégia que apóia a definição de requisitos e casos de teste funcional para cada padrão de uma determinada linguagem de padrões. A Etapa 2 fornece diretrizes para apoiar o reúso das classes de equivalência, dos requisitos e dos casos de teste criados pela estratégia em conjunto com a solução dos padrões.

Inúmeras vantagens são encontradas na utilização do ARA tanto na reengenharia quanto somente na engenharia reversa, como por exemplo: usa *frameworks* baseados em LPA, diminuindo o tempo do desenvolvimento do sistema e a vantagem da associação da LPA que ajuda na análise e na documentação do *software*; utiliza apoio computacional em várias fases, facilitando o desempenho de várias atividades; disponibiliza práticas ágeis; usa ferramentas para controle de versões, migração de dados, de modelagem, de instanciação do *framework* quando disponível, permitindo o uso de outras ferramentas quando houver

necessidade; usa a reengenharia guiada por teste podendo ser adaptada para a engenharia reversa para identificar funcionalidades fornecidas pelo *framework,* mas não presentes no sistema legado e fornece um ciclo incremental e iterativo.

Observa-se a importância do ARA como apoio em ambos os processos e, neste trabalho, verificou-se a possibilidade do ARA de apoiar também o desenvolvimento do *software*, pois embora o processo de desenvolvimento de *software* envolva a construção de todas as funcionalidades a partir de uma especificação, quando comparado com o processo de reengenharia e engenharia reversa existem várias semelhanças. Todos os processos possuem algumas fases (levantamento de requisitos, análise, projeto, implementação, teste, implantação e manutenção) em comum entre eles. O que muda tanto no processo de reengenharia, quanto engenharia reversa e desenvolvimento do *software* é somente a maneira como serão iniciadas e utilizadas as atividades ao longo do projeto. Complementando esta afirmação, de acordo com Sommerville (2003), a principal diferença entre a reengenharia e o desenvolvimento de um novo *software* é o ponto de partida.

## 4.3. Processo PARFAIT

De acordo com Cagnin, (2005b), PARFAIT é um **P**rocesso **Á**gil de **R**eengenharia baseado em **F**r**A**meworks no domínio de sistemas de **I**nformação com técnicas de VV**&T**, com o objetivo de migrar sistemas procedimentais para o paradigma orientado a objetos. O processo PARFAIT pertence ao arcabouço ARA e utiliza vários recursos disponíveis do arcabouço, como: ferramentas para controle de versões, *framework* baseado em LPA, práticas ágeis, ferramenta para migração de dados entre outros recursos e ferramentas.

As características principais do processo PARFAIT, segundo Cagnin (2005b) são:

- incremental, iterativo e baseado em *framework.*
- considera diversas práticas de métodos ágeis, como: versões pequenas, cliente presente, testes constantes, jogo do planejamento, programação em pares, propriedade coletiva do código, integração contínua, metáfora e semana de 40 horas.
- dirigido ao cliente e ao risco, ou seja, o cliente prioriza os requisitos que julga serem mais importantes. Com relação ao risco, o processo PARFAIT concentra em criar uma arquitetura estável logo no início do projeto, evitando diminuir os riscos da reengenharia.
- utiliza "reengenharia guiada por teste", com o objetivo de apoiar o entendimento do sistema legado, em que os testes são escritos antes do desenvolvimento e posteriormente utilizados durante a adaptação do sistema alvo para se adequar ao sistema legado e durante o teste do sistema alvo, para garantir a sua qualidade.
- executa paralelamente o sistema alvo com o sistema legado para avaliar a compatibilidade funcional entre eles.
- não se limita a reproduzir a funcionalidade do sistema legado, mas evolui o sistema de acordo com as necessidades dos usuários.
- a documentação tem seu formato baseado em diversos elementos fundamentais do arcabouço RUP (*Rational Unified Process*) (KURCHTEN, 2000), como por exemplo: atividades, passos, artefatos de entrada, artefatos de saída, papel, apoio computacional, gabaritos, diretrizes, entre outros (CAGNIN, 2005b).

Pelo fato do processo PARFAIT utilizar algumas técnicas, como por exemplo os critérios de teste funcionais (MYERS, 2004): Particionamento de Equivalência, Análise do

Valor Limite e utilizar como apoio computacional um *framework* baseado em LPA, torna-se necessário que o engenheiro de *software* tenha conhecimento dessas técnicas e dos recursos que são utilizados por essas técnicas, como exemplo no uso do *framework,* obtendo conhecimento da LPA, da linguagem de programação utilizada pelo *framework* e do SGBD (CAGNIN, 2005b).

O processo PARFAIT é dividido em quatro fases: CONCEPÇÃO, ELABORAÇÃO, CONSTRUÇÃO e TRANSIÇÃO. Para cada fase existem diversas atividades, diretrizes, inspeções, artefatos de entrada e artefatos de saída e o papel do responsável pela execução de cada atividade. As atividades podem ser executadas por apenas uma pessoa para os diversos papéis.

Na Figura 4.2 apresentam-se somente as fases, as atividades e as iterações entre as fases do processo PARFAIT. Todas as fases são formadas por atividades e cada atividade possui passos que são utilizados para executar a atividade por completo.

No final de cada fase existe um marco de referência, que é representado por uma pirâmide localizada no interior do quadro de cada fase. O marco de referência é composto por questões com o objetivo de avaliar se a execução da fase obteve um resultado satisfatório.

Em algumas fases, quando esse resultado não é atingido, há a necessidade de revisar todas as atividades executadas ou voltar para a fase anterior. Como exemplo, tem-se a fase de **CONCEPÇÃO**, em que é feita uma avaliação para decidir se continua ou cancela o projeto de reengenharia. Para cada fase existem atividades, sendo que algumas são obrigatórias, como por exemplo, na fase de **CONSTRUÇÃO** a atividade "Adaptar o sistema alvo*"* é obrigatória e deve ser executada pelo programador. Outras atividades não são obrigatórias e estão representadas na figura por meio de retângulo com linhas tracejadas.

Uma outra observação em relação às atividades do processo PARFAIT é que algumas atividades são iterativas e incrementais, a fim de refinar os artefatos produzidos. Essas atividades são representadas na figura por um retângulo de linha dupla.

As diretrizes são representadas pela letra (D), utilizada para apoiar o uso de técnicas durante a aplicação do processo e as inspeções são representadas pela letra (I), que são utilizadas para validar os artefatos produzidos. Algumas fases possuem em suas atividades tanto diretrizes como inspeções, como exemplo na fase de **ELABORAÇÃO**, na atividade "Desenvolver o diagrama de casos de uso e elaborar os casos de teste".

A sigla SL representa sistema legado e só aparece nas atividades em que é necessário ter o sistema legado para a execução da atividade.

A primeira fase do processo PARFAIT é a fase de **CONCEPÇÃO**. O objetivo desta fase é identificar o domínio do sistema legado em relação ao do *framework* disponível. Nesta fase é realizada a elaboração do projeto de reengenharia, que geralmente é baseado em projetos anteriores ou na experiência do responsável pelo projeto. Esta fase é composta por quatro atividades, sendo: "Familiarizar-se com o domínio do *framework"*, "Observar o domínio do sistema legado em relação ao do *framework*", "Confrontar as características não funcionais do *framework* x sistema legado" e "Elaborar o planejamento do projeto de reegenharia".

A próxima fase é a **ELABORAÇÃO**, cujo objetivo é elaborar a documentação do sistema legado, seguindo a prioridade dos requisitos definida pelo usuário. Essa fase é composta pelas atividades: "Desenvolver o diagrama de casos de uso e elaborar os casos de teste", "Desenvolver o diagrama de classes do sistema alvo", "Documentar as modificações realizadas no diagrama de classes" e "Documentar as regras do negócio do sistema". Os casos de teste são criados nessa fase para apoiar o entendimento do sistema legado e para

futuramente auxiliar na validação do sistema alvo. A documentação orientada a objeto é elaborada com o apoio da linguagem de padrões de análise.

A fase seguinte é a fase de **CONSTRUÇÃO.** O objetivo desta fase é realizar a instanciação do *framework,* para obter o sistema alvo. Nesta fase são executados os casos de teste, criados anteriormente no sistema alvo e realizadas as adaptações nesse sistema para que fique com funcionalidades semelhantes ao sistema legado. As atividades desta fase são: "Criar o sistema alvo no paradigma orientado a objetos", "Executar os casos de teste no sistema alvo" e "Adaptar o sistema alvo". A última fase é de **TRANSIÇÃO**. O objetivo desta fase é garantir que o sistema alvo possa ser disponibilizado para o usuário. Nesta fase são realizados testes no sistema alvo, a fim de garantir sua integridade bem como os ajustes finais. É composta pelas seguintes atividades: "Elaborar o manual do usuário do sistema alvo", "Converter a base de dados do sistema legado", "Testar o sistema alvo" e "Treinar os usuários finais".

O processo PARFAIT apresenta-se importante para a reengenharia, principalmente com o apoio das atividades de VV&T, o que facilita o entendimento do sistema legado, a elicitação e refinamento dos requisitos e a validação do produto final (CAGNIN, 2005a).

Um fator importante apontando por Cagnin (2005a), é que o processo PARFAIT foi utilizado em apenas sistemas de pequeno porte, não havendo resultados que comprove sua consolidação em sistemas de maior porte.

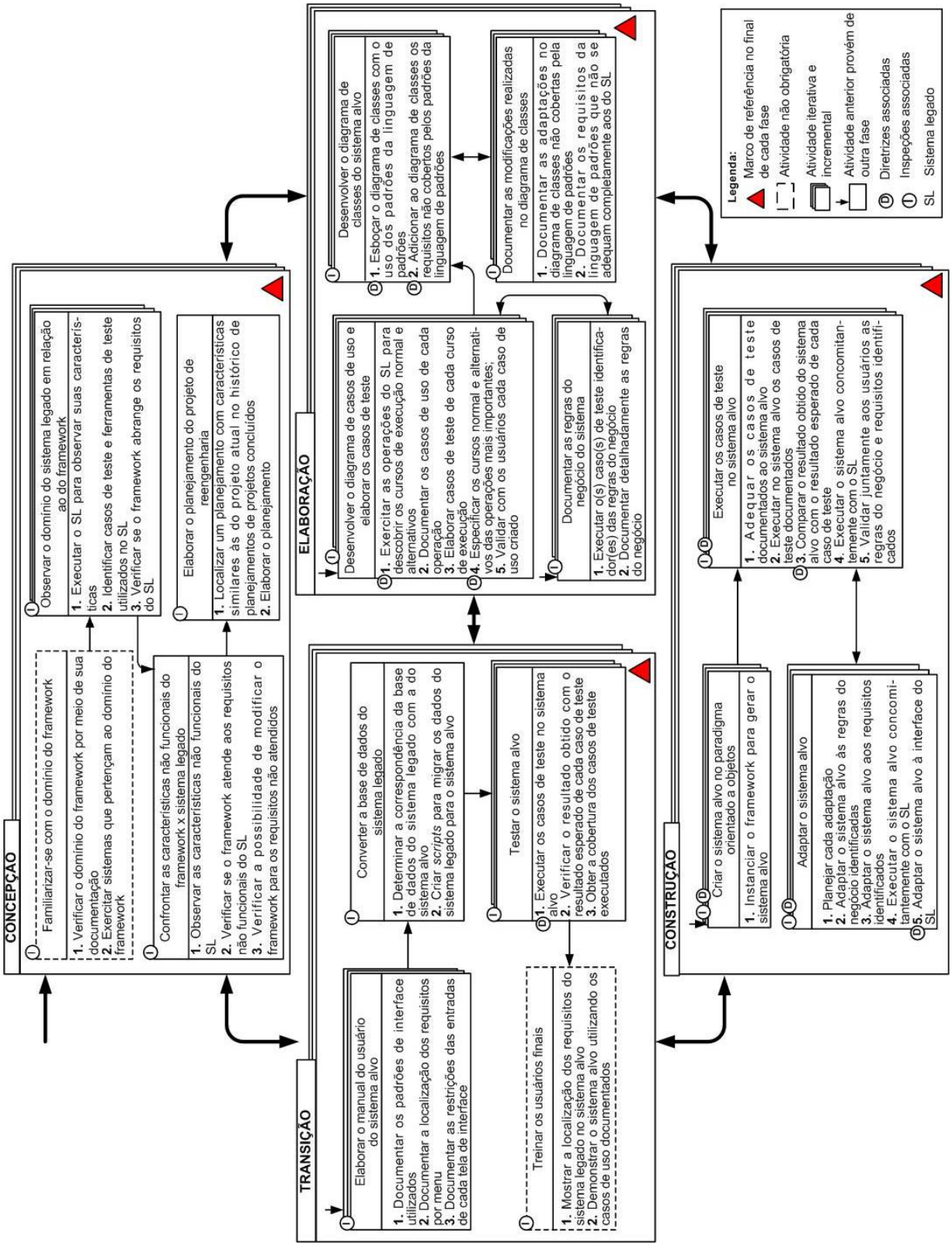


Figura 4.2: Visão completa do PARFAIT (CAGNIN, 2005a)

## 4.4. Considerações Finais

Neste capítulo foi apresentado o arcabouço ARA e, mais detalhadamente, o processo PARFAIT. O arcabouço ARA é utilizado neste trabalho para apoiar o processo de desenvolvimento de *software* abstraído a partir do processo PARFAIT, apresentado no capítulo seguinte.

No próximo capítulo apresenta-se uma análise comparativa de um estudo de caso de reengenharia para a definição da versão inicial do processo PARFAIT/EA e os resultados de um estudo de caso da aplicação do processo PARFAIT/EA no desenvolvimento de *software* a fim de analisá-lo e evoluí-lo.

# CAPÍTULO 5. FERRAMENTAS PARA GERENCIAR PROJETOS E DOCUMENTAR PROCESSOS

## 5.1 Considerações Iniciais

Na busca pela qualidade de *software*, o uso de processo de *software* ou de uma metodologia tem mostrado ser algo inevitável.

Para ajudar na melhoria do processo de *software,* é vantajosa a utilização de um apoio automatizado, por meio de uma ferramenta que seja capaz de orientar o uso do processo e armazenar em um repositório as experiências adquiridas, podendo ser reutilizadas em projetos futuros para apoiar, por exemplo, em estimativas de tempo e custo. O uso de uma ferramenta favorece tanto a evolução do processo quanto a evolução do *software.*

Nesse contexto, este capítulo está organizado da seguinte forma: na Seção 5.2 apresentam-se algumas ferramentas para gerenciar projetos utilizando métodos ágeis, na Seção 5.3 apresenta-se a ferramenta P.DOCTool (BIANCHINI, 2004), utilizada para documentar processos de *software,* cuja estrutura seja baseada naquela do processo RUP (*Rational Unified Process*), como é o caso do processo PARFAIT/EA e na Seção 5.4 apresentam-se as considerações finais deste capítulo.

## 5.2 Ferramentas para Gerenciar Projetos de Métodos Ágeis

Existem diversas ferramentas para documentar métodos ágeis. Algumas delas como XPlanner (XPLANNER, 2006), XPWeb (CHIROUZE *et al*, 2006), VersionOne (VERSIONONE, 2006) e ExtremePlanner (EXTREMEPLANNER, 2006) estão no escopo deste trabalho e são apresentadas a seguir.

A ferramenta **XPlanner** é uma ferramenta para planejamento e acompanhamento de projetos XP (eXtreme Programming) (BECK, 2000). Ela possui código livre, seu acesso é via *Web* e foi desenvolvida em JSP (*Java Server Pages*). Para a utilização da ferramenta é necessário conhecimento das práticas ágeis do método XP.

Algumas características da ferramenta são: cadastrar as estórias dos usuários agrupadas por iterações, cadastrar as tarefas desenvolvidas, projetar um modelo simples de planejamento, exportar o arquivo do projeto para vários formatos, como exemplo XML (e*Xtensible Markup Language*), e permitir que o gerente de projeto possa distribuir as tarefas entre os participantes do projeto, com a devida carga horária planejada. Os desenvolvedores podem ter acesso as informações e verificar as tarefas sendo executadas, favorecendo uma melhor comunicação entre toda a equipe.

Na Figura 5.1 apresenta-se a tela de visualização da iteração de um projeto de *software*. Essa tela exibe a iteração, com a respectiva carga horária, as estórias da iteração numeradas, indicando o grau de prioridade e uma barra que apresenta o progresso de implementação da estória, com base no tempo estimado inicialmente. Na parte inferior, tem-se opções para: editar uma iteração, criar uma nova estória, visualizar métricas de tempo por equipe e individual e estatísticas sobre as informações da iteração corrente.

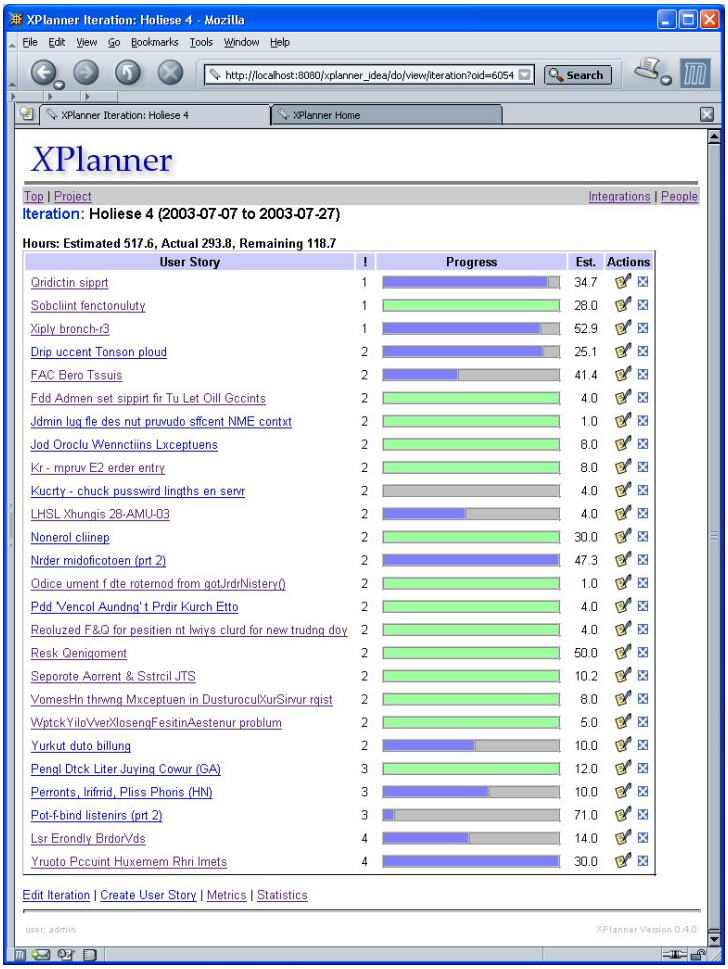


Figura 5.1: Tela de visualização de iteração da ferramenta XPlanner (XPLANNER, 2006)

A ferramenta **XPWeb** é utilizada para o gerenciamento de projetos que utilizem o método ágil XP, permitindo controlar as iterações, as estórias de usuários e as tarefas definidas nos projetos.

Entre algumas funcionalidades da ferramenta XPWeb tem-se: exportar o arquivo do projeto para o formato XML (e*Xtensible Markup Language*), autenticar usuários, criar vários projetos, manipular a documentação gerada, utilizar calendário para o agendamento das tarefas cadastradas e emitir relatórios para visualização do andamento do projeto.

XPWeb é uma ferramenta com interface *Web* e foi desenvolvida na linguagem de *script* PHP (*Hypertext Preprocessor*) (WELLING e THOMSON, 2005).

Na Figura 5.2 apresenta-se a tela para o cadastro de estória de usuário, com as opções para definir a prioridade e o risco, modificar e apagar a estória. Essa tela foi captada no modo de demonstração da ferramenta.

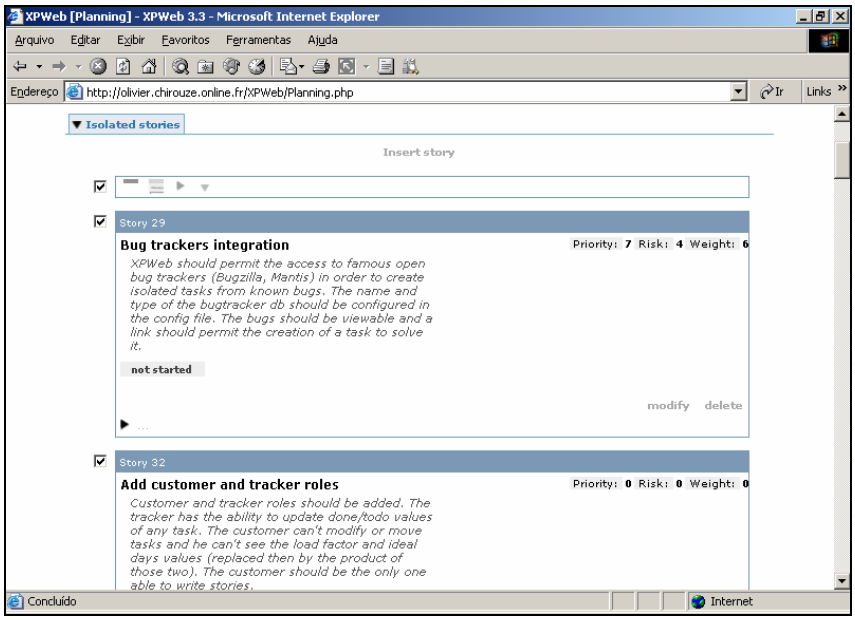


Figura 5.2: Tela para cadastro de estórias do usuário da ferramenta XPWeb (CHIROUZE et al, 2006)

A ferramenta **VersionOne** é uma ferramenta para gerenciamento e planejamento de projetos de *software*, com suporte para os métodos ágeis: *Scrum*, XP, DSDM e AgileUp (AMBLER, 2006b). A ferramenta utiliza várias práticas ágeis como: jogo do planejamento, testes constantes, versões freqüentes e oferece um modelo de configuração para o cliente selecionar o método que melhor se adapta as suas necessidades.

Os participantes do projeto, como: gerente de projeto, cliente, programadores, testadores, entre outros, podem trabalhar juntos no projeto, por meio de prioridades e responsabilidades estabelecidas na ferramenta.

Algumas características da ferramenta são: gerenciamento de vários projetos e versões, gerenciamento de requisitos (estórias de usuários, lista de *backlog*, lista de características, etc), planejamento das iterações, notificação das modificações do projeto por meio da tecnologia RSS (*Rich Site Summary*) e oferece uma API (*Application Programming*

*Interfaces)* para integrar outras ferramentas e dados. Além disso, a ferramenta também oferece relatórios que permitem a visualização do andamento do projeto e das iterações.

Na Figura 5.3 apresenta-se a tela principal da ferramenta VersionOne, em que foi escolhido o método ágil XP. Nessa tela tem-se acesso a várias opções oferecidas pela ferramenta, como por exemplo: os projetos, as tarefas, as versões, os casos de testes, o planejamento, as opções de relatórios, entre outras. A ferramenta oferece ainda uma opção chamada sumário, que disponibiliza um gráfico que mostra o progresso do projeto. A VersionOne é de domínio proprietário.

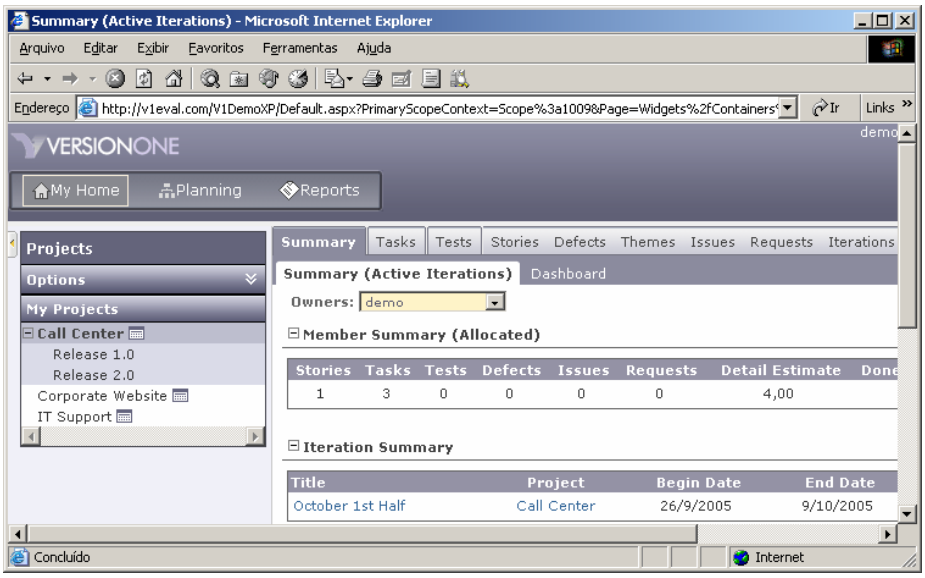


Figura 5.3: Tela principal da ferramenta VersionOne (VERSIONONE, 2006)

A ferramenta **ExtremePlanner** tem como objetivo controlar e coordenar o desenvolvimento de *software* iterativo, incremental e flexível, com suporte para os métodos ágeis XP e *Scrum*. ExtremePlanner proporciona uma melhor comunicação da equipe de projeto, permitindo que todos possam colaborar de forma eficaz.

Algumas características da ferramenta ExtremePlanner são: controlar as estórias, as tarefas e os casos de teste com facilidade, verificar o progresso do projeto permitindo incluir novos requisitos, adicionar documentos, imagens, notas nas estórias, enviar notificações por

*e-mail* das mudanças do projeto, permitir múltiplos projetos, iterações e versões e manter a equipe informada das mudanças do projeto.

Na Figura 5.4 apresenta-se a tela principal da ferramenta ExtremePlanner. Dentre as opções disponíveis no formato de *link,* têm-se: visualizar as versões correntes, as iterações e as tarefas ativas distribuídas por usuário; ter acesso as estórias cadastradas com opção para alterá-las, removê-las e exportá-las no formato de arquivo do *Excel* ou RTF (*Rich Text Format*); ter acesso aos casos de testes e selecionar outros projetos cadastrados. A ferramenta ExtremePlanner é de domínio proprietário.

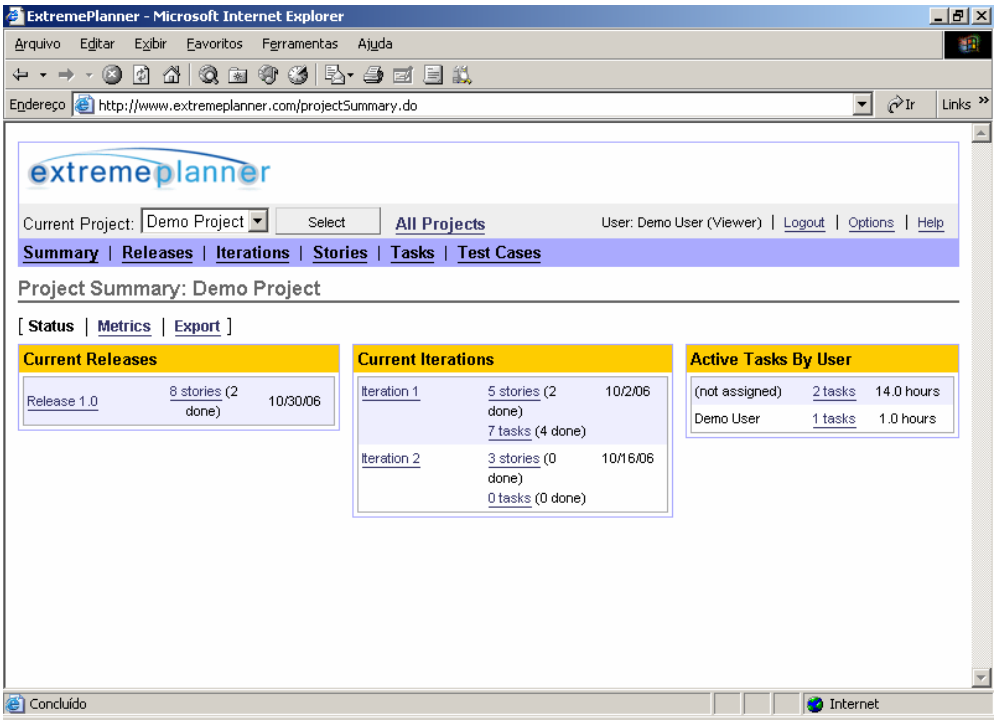


Figura 5.4: Tela principal da ferramenta ExtremePlanner (EXTREMEPLANNER, 2006)

No Quadro 5.1 destacam-se algumas características das ferramentas estudadas, como: a linguagem de programação, sendo que todas as ferramentas utilizam linguagem voltada para desenvolvimento *web*, facilitando o acesso à ferramenta e as informações pela equipe de projeto; o(s) método(s) ágil(eis) suportado pela ferramenta, sendo que todas suportam o método XP: a disponibilidade da ferramenta, ou seja, gratuita (código aberto) ou

proprietária, o SGBD utilizado e as principais funcionalidades existentes em cada ferramenta. Observou-se que todas as ferramentas gratuitas utilizam SGBD's gratuitos. Outra observação identificada é de todas as ferramentas estudadas, nenhuma delas apóia processo de desenvolvimento de *software*. No Quadro 5.1 apresenta-se um resumo das ferramentas descritas anteriormente.

Quadro 5.1: Resumo das ferramentas

| **Ferramentas/ Características** | **XPlanner** | **XPWeb** | **VersionOne** | **ExtremePlanner** |
|---|---|---|---|---|
| **Método ágil suportado** | XP, *Scrum* | XP | *Scrum*, XP, DSDM, *AgileUp* | XP, *Scrum* |
| **Linguagem desenvolvida** | JSP | Java | ASP.Net *Framework* Microsoft .NET 1.1 | Java |
| **Disponibilidade** | Código aberto | Código aberto | Proprietária | Proprietária |
| **Estrutura** | *Web* | *Web* | *Web/Desktop* | *Web* |
| **SGBD alvo** | MySQL | MySQL | Microsoft SQLServer 2005 | *SQL Database* |
| **Principais funcionalidades** | Cadastrar e controlar as estórias agrupadas por iterações, cadastrar as tarefas, projetar um modelo simples de planejamento, e gerar gráficos para acompanhar o projeto | Cadastrar e controlar as estórias, gerar calendário para o agendamento das tarefas e emitir relatórios para acompanhamento do projeto | Cadastrar e controlar os requisitos, versões, planejamento das iterações, controlar as modificações do projeto, fornecer API para integrar outras ferramentas e dados e emitir relatórios para acompanhamento do projeto | Cadastrar e controlar as estórias, as tarefas, os casos de teste, as iterações, versões e permitir a visualização de gráficos para acompanhamento do projeto |
| **Apóia processo de desenvolvimento** | Não | Não | Não | Não |

## 5.3 Ferramenta P.DOCTool para Documentar Processos

A ferramenta P.DOCTool (BIANCHINI, 2004) foi desenvolvida para apoiar na documentação e consulta *on-line* de processos de *software* que seguem a estrutura do processo RUP (*Rational Unified Process*) (KRUCHTEN,2000).

Em um estudo de caso conduzido por Bianchini (2004) para avaliar a ferramenta, a documentação do processo PARFAIT foi cadastrada e posteriormente consultada a fim de verificar se a visualização dos dados estava sendo apresentada corretamente. Isso foi possível, pois a documentação do processo PARFAIT é baseada na estrutura do RUP.

P.DOCTool é baseada na *Web*, desenvolvida na linguagem PHP (WELLING e THOMSON, 2005) e utiliza o SGBD MySQL (WELLING e THOMSON, 2005).

As principais funcionalidades da ferramenta são: autenticar usuários, armazenar documentos de processos (fases, atividades, diretrizes, artefatos de entrada e artefatos de saída) e permitir *upload* e *download* de gabaritos referentes aos artefatos que devem ser produzidos durante o uso do processo.

A P.DOCTool possui três visões de usuários, sendo que em cada visão há restrição de algumas operações da ferramenta. Os tipos de usuários da ferramenta são:

- administrador do sistema (*system administrator*): possui uma visão de todo o sistema com funcionalidades exclusivas para cadastrar usuários e tem acesso integral nas permissões para editar e excluir os cadastros armazenados.
- gerente de projetos (*project manager*): possui acesso a todos os cadastros relacionados aos processos criados por ele.
- usuário simples (*simple user*): possui acesso a ferramenta apenas para visualização da documentação de processos cadastrados.

A estrutura da base de dados da ferramenta está ilustrada na Figura 5.5 e cada elemento é formado por um conjunto de elementos, sendo: processo possui um tipo (classe *TypeProcesses*) e é formado por fases (classe *Phases*). Cada fase possui atividades (classe *Activities*) que são executadas por meio de passos (classe *Steps*), com apoio de ferramentas computacionais existentes (classe *Tools*). Cada atividade é executada por profissionais (classe *WorkerRoles*) e pode ter diretrizes (classe *Guidelines*) e gabaritos (classe *Template)* para facilitar a elaboração dos artefatos de saída (classe *OutPut*). Os artefatos de entrada (classe *Input*) apóiam a execução de cada atividade, as verificações (classe *Verifications*) são utilizadas para validar os artefatos produzidos e os marcos de referências (classe *Milestones*) são aplicados no final de cada fase, para observar o andamento do projeto.

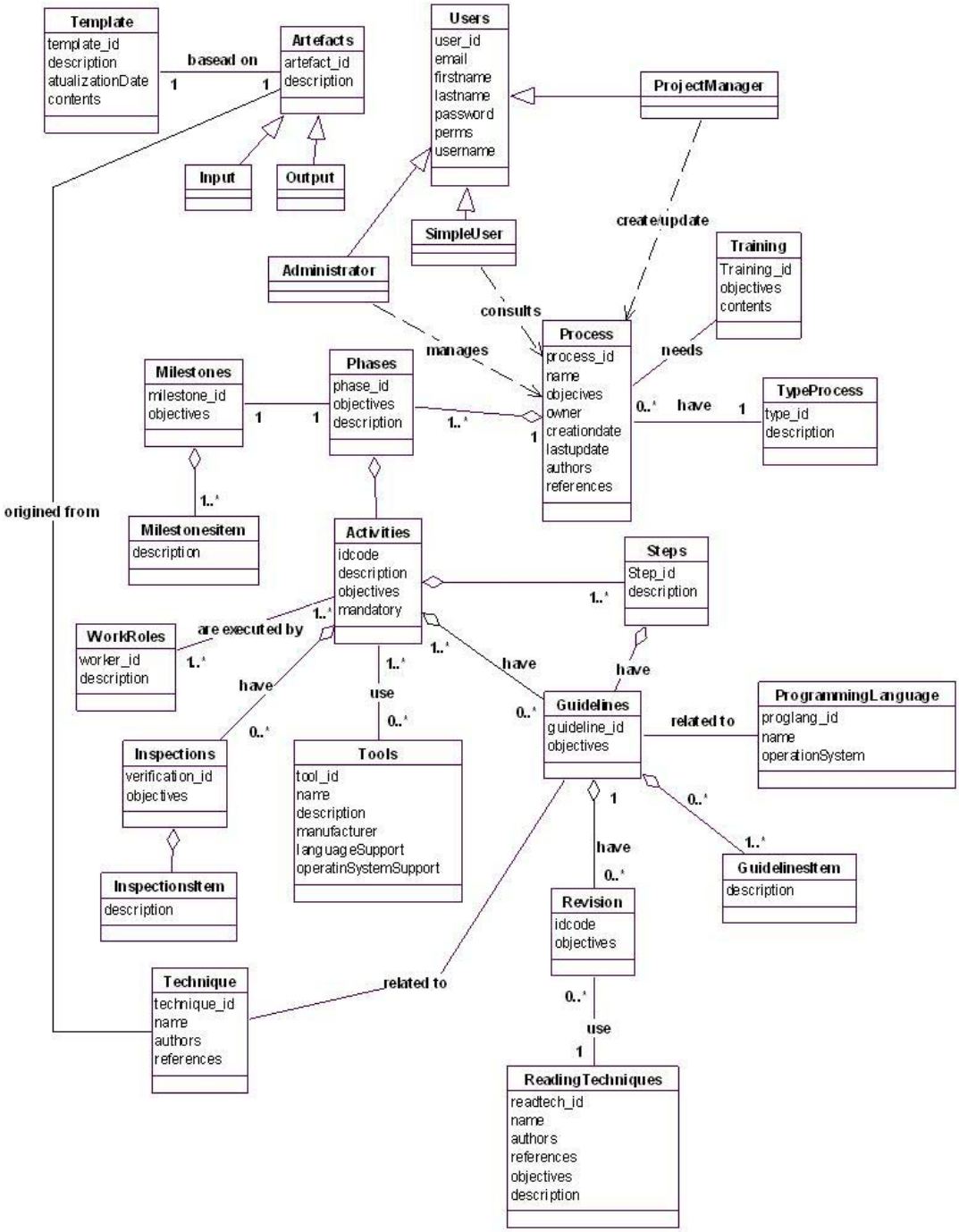


Figura 5.5: Diagrama de classes da base de dados da P.DOCTool (adaptada de BIANCHINI, 2004)

## 5.4 Considerações Finais

Com base nas ferramentas apresentadas para gerenciar projetos de *software* que apóiam métodos ágeis, conclui-se que nenhuma delas apóia processo de desenvolvimento de *software*.

De acordo com essa observação, com a disponibilização do código fonte da ferramenta P.DOCTool, apresentada neste capítulo, uma vez que foi desenvolvida no ICMC-USP e possui fins acadêmicos, e o seu apoio a documentação de processos cuja a estrutura do processo seja baseada no RUP, evidenciou-se a possibilidade de estendê-la neste trabalho. A extensão da ferramenta P.DOCTool é apresentada no Capítulo 7.

# CAPÍTULO 6 DEFINIÇÃO E EVOLUÇÃO DO PARFAIT/EA

## 6.1 Considerações Iniciais

Várias são as atividades envolvidas no processo de desenvolvimento de *software.* Dentre essas atividades estão aquelas em que os engenheiros de *software* buscam aumentar a produtividade e a qualidade do *software* com o uso de componentes e *frameworks* (PEDRYCZ e PETERS, 2001). Nesse contexto, este capítulo apresenta a definição e evolução de um processo ágil de desenvolvimento baseado em *frameworks*, cuja construção do *framework* seja baseada em uma linguagem de padrões de análise. Esse processo é denominado PARFAIT/EA.

Na Seção 6.2 apresenta-se o processo PARFAIT/EA, juntamente com suas fases, atividades, diretrizes, inspeções e os documentos de entrada e saída. Na Seção 6.3 apresenta-se uma discussão sobre a análise realizada para a definição do esboço do processo PARFAIT/EA, abstraído a partir de um processo ágil de reengenharia baseado em *framework*, denominado PARFAIT. Na Seção 6.4 apresenta-se um estudo de caso planejado com o objetivo de avaliar a aplicabilidade do processo PARFAIT/EA no desenvolvimento de *software,* definido na Seção 6.3 e, finalmente, na Seção 6.5 apresentam-se as considerações finais deste capítulo.

## 6.2 Processo PARFAIT/EA

O processo PARFAIT/EA foi definido por meio de estudos de casos. O primeiro estudo de caso teve como objetivo definir o esboço do processo PARFAIT/EA (GOMES *et al*, 2006) por meio da análise dos resultados de um estudo de caso de reengenharia conduzido anteriormente por Cagnin *et al* (2003a). O segundo estudo de caso (GOMES e CAGNIN, 2007a) teve como objetivo avaliar a utilização do processo no desenvolvimento de *software*, em que foi constatado que PARFAIT/EA possui características para ser aplicado no desenvolvimento de *software* e representa mais um recurso do arcabouço ARA.

Para cada atividade, o processo fornece gabaritos para apoiar na elaboração dos artefatos de saída da atividade, para orientar o responsável na coleta das informações e conseqüentemente gerar documentação. Os gabaritos disponíveis no processo são simples e de fácil entendimento tanto para o responsável como para o cliente.

Em relação ao uso de ferramentas computacionais, PARFAIT/EA utiliza ferramentas para: modelagem UML, para editoração de texto, para instanciação do *framework* disponível, entre outras ferramentas que sejam necessárias para o projeto de *software*.

A documentação do processo PARFAIT/EA é formada por: papel do responsável para a execução da atividade, documentos de entrada necessários para execução da atividade, documentos de saída produzidos na atividade, seqüência de passos para facilitar o uso da atividade, ferramentas computacionais para apoiar a execução da atividade e aplicação de diretrizes e inspeções para facilitar a execução da atividade e para validar os artefatos produzidos, respectivamente.

A escolha dos papéis em cada atividade foi baseada nos papéis indicados pelo processo PARFAIT (CAGNIN, 2005a).

A implantação do *software* é realizada de forma incremental e no final de cada ciclo de desenvolvimento um novo ciclo é definido, de acordo com as necessidades exigidas pelo cliente. Nos Quadros 6.1, 6.2, 6.3 e 6.4 apresentam-se um resumo do processo contendo o objetivo de cada fase, as atividades e os documentos de entrada, documentos de saída e as práticas aplicadas em cada atividade.

Quadro 6.1: Fase de CONCEPÇÃO

| **Fase de CONCEPÇÃO**<br>Observar as condições do projeto de *software* em relação ao *framework* disponível e os riscos em utilizar o *framework* | | | | | |
|---|---|---|---|---|---|
| **Atividade** | **Objetivo** | **Papéis** | **Documento(s) de entrada** | **Documento (s) de saída** | **Prática(s)** |
| Familiarizar-se com o domínio do framework | Entender o domínio do framework disponível | Analista de Sistemas e Programadores | Exemplos de aplicações apoiadas pelo *framerwork*, documentação do *framework,* da LPA utilizada pelo *framework* e da ferramenta para instanciar o *framework* | Formulário para avaliação de conhecimento do domínio do framework | - |
| Observar o domínio do projeto de software em relação ao framework | Analisar se o domínio do projeto de *software* se encaixa no domínio do framework disponível. Associar a essa atividade a atividade "Confrontar as características não funcionais do framework x projeto de software" para identificar os requisitos não funcionais. Essa atividade permite identificar a viabilidade de iniciar ou não o projeto de *software* | Engenheiro de *software*, Analista de Sistemas | Exemplos de projetos de *software* no mesmo domínio do projeto a ser desenvolvido | Documentos coletados com o cliente e um Resumo de cada reunião realizada com a presença do cliente | Cliente presente e uso de metáforas |
| Elaborar o planejamento do projeto de software | Elaborar o planejamento do projeto de *software* de acordo com as informações coletadas na atividade "Observar o domínio do projeto de software em relação ao framework" e associar a atividade "Confrontar as características não funcionais do framework x projeto de software" | Analista de Sistema | Histórico de planejamentos de projetos de software similares ao projeto atual | Lista de Requisitos e Planejamento do Projeto de Software | Jogo do planejamento e Detalhamento nos requisitos definidos |
| Confrontar as características não funcionais do framework x projeto de software | Verificar as características não funcionais do *framework* e do projeto de software. | Analista de Sistemas | Documentos coletados na atividade "Observar o domínio do projeto de software em relação ao do framework", sendo: Documentos coletados com o cliente e um Resumo de cada reunião realizada na presença do cliente | Lista dos requisitos não funcionais | - |
| **Reuniões freqüentes** fornecem questões para analisar e identificar informações do projeto de *software.* | | | | | |
| **Marcos de referência** tem o objetivo de avaliar se o *framework* disponível apresenta os requisitos funcionais e não funcionais do projeto de *software*. Essa avaliação é realizada por *checklist.* | | | | | |

Quadro 6.2: Fase de ELABORAÇÃO

| **Fase de ELABORAÇÃO**<br>Produzir documentação suficiente para apoiar a construção do *software*, as adaptações e futuras manutenções. | | | | | |
|---|---|---|---|---|---|
| **Atividade** | **Objetivo** | **Papéis** | **Documento(s) de entrada** | **Documento (s) de saída** | **Prática(s)** |
| Desenvolver o diagrama de casos de uso e elaborar os casos de teste | Descrever cada requisito identificado na fase de CONCEPÇÃO e elaborar para cada requisito os casos de teste | Analista de Sistemas | Lista de requisitos priorizada pelo cliente de acordo com ciclo de desenvolvimento, definido no gabarito de planejamento | Diagrama de casos de uso, documentação dos casos de teste e documentação das classes de equivalência | Cliente Presente |
| Documentar as regras de negócio do software | Documentar todas as regras de negócio identificadas para facilitar as adaptações necessárias no *software* | Analista de Sistemas | Diagrama de classe e diagrama de caso de uso | Documentação das regras de negócio | - |
| Desenvolver o diagrama de classes do software | Desenvolver o diagrama de classe de acordo com o ciclo de desenvolvimento, com base na linguagem de padrões de análise disponível pelo framework | Analista de Sistemas | Diagrama de casos de uso e linguagem de padrões utilizada pelo *framework* | Diagrama de classe | Metáfora |
| Documentar as modificações realizadas no diagrama de classes | Documentar os padrões utilizados no diagrama de classes que não atenderam aos requisitos do *software* e as classes que não foram atendidas pelos padrões | Analista de Sistemas | Diagrama de Classe do *software* | Gabarito da documentação das adaptações no diagrama de classe não cobertas pela linguagem de padrões e Gabarito da documentação dos requisitos da linguagem de padrões que não se adequam ao projeto de *software* | - |
| **Reuniões freqüentes** têm como objetivo discutir a documentação que está sendo produzida, as prioridades definidas pelo cliente, os casos de testes produzidos e as regras de negócios documentadas. | | | | | |
| **Marcos de referência** têm como objetivo avaliar o ciclo corrente. Algumas questões são consideradas. | | | | | |

Quadro 6.3: Fase de CONSTRUÇÃO

| **Fase de CONSTRUÇÃO**<br>Criar o *software* de acordo com os requisitos identificados e priorizados pelo cliente para o ciclo corrente e permitir, quando necessário, que os requisitos sejam adaptados para atender as exigências do projeto de *software.* | | | | | |
|---|---|---|---|---|---|
| **Atividade** | **Objetivo** | **Papéis** | **Documento(s) de entrada** | **Documento (s) de saída** | **Prática(s)** |
| Criar o software | Criar o *software* por meio da instanciação do *framework*, de acordo com os requisitos priorizados para o ciclo corrente | Programador | Documentação do *framework*, da LPA e das ferramentas de instanciação do *framework* | *Software* | Versões freqüentes, integração contínua, propriedade coletiva do código |
| Executar os casos de teste no software | Aplicar os casos de teste no *software* com objetivo de identificar erros e falhas operacionais | Testador, Cliente | Documentação dos casos de teste, *software* | Relatório resumo de teste | Testes constantes |
| Adaptar o software | Adaptar todas modificações não fornecidas pelo *framework* e necessárias ao projeto de *software,* como: regras de negócio e sugestões solicitadas pelo cliente, importantes ao projeto | Programador | Documentação das regras de negócios, *software* | *Software* modificado | Projetar (adaptar) com simplicidade, propriedade coletiva do código. |
| **Reuniões freqüentes** têm como objetivo discutir com a equipe de projeto e o cliente as versões liberadas e as modificações realizadas no projeto de *software.* | | | | | |
| **Marcos de referência** avalia a *interface* do *software* e a atividade de teste de acordo com um *checklist* disponível. | | | | | |

Quadro 6.4: Fase de TRANSIÇÃO

| **Fase de TRANSIÇÃO**<br>Garantir que a versão corrente do *software* esteja de acordo com as exigências do ciclo para que possa ser disponibilizada para o cliente | | | | | |
|---|---|---|---|---|---|
| **Atividade** | **Objetivo** | **Papéis** | **Documento(s) de entrada** | **Documento (s) de saída** | **Prática(s)** |
| Testar o software | Executar teste de aceitação juntamente com o cliente para verificar se o *software* atende aos requisitos identificados para o ciclo | Testador | *Software* e documentação dos casos de testes | *Software* testado. | Testes constantes |
| Treinar os usuários finais | Treinar os usuários para que possam utilizar adequadamente o *software* | Analista de Sistemas, Suporte Técnico | *Software* | - | - |
| Elaborar o manual do usuário do software | Elaborar o manual do usuário para facilitar o uso do *software,* podendo ser *on-line* | Analista de Sistemas | *Software* | Manual do usuário | - |
| **Reuniões freqüentes** têm com objetivo discutir questões que envolvem o *software* e as expectativas do cliente em relação ao *software.* | | | | | |
| **Marcos de referência** tem o objetivo de verificar se todos os requisitos foram atendidos para a implantação do *software* na empresa cliente, seguindo um critério para avaliação. | | | | | |

A documentação completa do processo PARFAIT/EA está disponível no (Apêndice A).

## 6.3 Análise do PARFAIT para abstrair o PARFAIT/EA

Para melhor definição do esboço do processo de desenvolvimento PARFAIT/EA, foram utilizados os resultados de um estudo de caso de reengenharia (CAGNIN *et al*, 2003a) conduzido anteriormente com a aplicação do processo de reengenharia PARFAIT.

Com base nos resultados desse estudo de caso de reengenharia, foi realizada uma análise nas atividades de cada fase do processo PARFAIT, procurando identificar quais atividades e passos atendiam ao contexto de desenvolvimento de *software,* em que algumas atividades foram aproveitadas, outras foram adaptadas, retiradas ou adicionadas para compor o novo processo.

Esse estudo de caso de reengenharia foi conduzido em um sistema de biblioteca de uma universidade, implementado originalmente na linguagem de programação Clipper, contendo aproximadamente 6 KLOC (linhas de código fonte) (CAGNIN *et al,* 2003a).

Ressalta-se que o conteúdo de algumas atividades do PARFAIT não foi alterado, no entanto apenas seus nomes foram modificados para se adequar ao contexto de desenvolvimento de *software* e estão citados nesta seção.

A classificação das atividades do processo PARFAIT, como: “não obrigatória”, “iterativa e incremental” e “atividade anterior provém de outra fase”, foram mantidas no processo PARFAIT/EA. Com a análise do estudo de caso, observou-se que essa classificação é importante, pois orienta o engenheiro de *software* no uso das atividades, sendo forçado, de

uma certa maneira, a executar todos os passos necessários para o desenvolvimento do *software*.

Para a definição das atividades da fase de CONCEPÇÃO do processo PARFAIT/EA, todas as atividades dessa fase do processo PARFAIT foram utilizadas. A primeira atividade "Familiarizar-se com o domínio do framework" foi utilizada na íntegra, pois o processo PARFAIT/EA também é baseado no uso de *frameworks* de aplicação. As outras atividades dessa fase foram adaptadas. Por exemplo, tem-se a atividade "Observar o domínio do projeto de software em relação ao framework", com o nome já alterado para o contexto do processo PARFAIT/EA. Os dois primeiros passos dessa atividade do processo PARFAIT foram retirados, pois não atendem ao contexto de desenvolvimento. Esses passos são: "Executar o sistema legado para observar suas características" e "Identificar casos de teste e ferramentas de teste utilizados no sistema legado". Para complementar essa atividade visando ao desenvolvimento de *software*, alguns passos foram adicionados, são eles: "Realizar entrevistas com o cliente para identificar o domínio do projeto do software" e "Coletar documentos para apoiar a identificação do domínio do projeto de software". Nessas atividades, o engenheiro de *software* realiza entrevistas e coleta informações a respeito do domínio do sistema por meio de documentos (relatórios, manuais, procedimentos, etc) ou de entrevistas com o cliente.

Nas outras duas atividades da fase de CONCEPÇÃO do PARFAIT foram alterados apenas os nomes para atender ao contexto de desenvolvimento de *software*, que são "Confrontar as características não funcionais do framework x projeto de software" e "Elaborar o planejamento do projeto de software", foram adaptadas. A seqüência de execução dessas duas atividades foi alterada porque no caso do desenvolvimento do *software* o ideal é que na elaboração do planejamento do projeto de *software* as características não funcionais do

projeto já estejam identificadas, a fim de serem confrontadas com as do *framework,* assim essas atividades devem ser executadas de forma paralela.

Ainda na fase de CONCEPÇÃO, na atividade já definida para o processo PARFAIT/EA "Elaborar o planejamento do projeto de software", foram adicionados três novos passos: "Realizar entrevistas e fornecer questionário(s) para a coleta de requisitos", "Definir a lista de requisito do novo projeto junto ao cliente" e "Permitir que o cliente liste os requisitos mais importantes". Com a adição desses novos passos, pode-se comparar essa atividade à prática ágil "Jogo do Planejamento" do método ágil XP (BECK, 2000), pois segue a mesma idéia, em que o cliente prioriza os requisitos para a próxima iteração e os programadores utilizam estimativas de custo para determinar o que precisa ser feito e o que pode ser adiado. Uma das técnicas que pode ser utilizada nessa atividade é o uso de questionário, que deve ser preenchido pelo próprio cliente ou por um dos integrantes do projeto. O questionário, depois de preenchido, é analisado pela equipe de projeto e documentado.

Na fase de ELABORAÇÃO, somente a primeira atividade "Desenvolver o diagrama de casos de uso e elaborar os casos de teste" foi adaptada para atender ao desenvolvimento do *software*. Nessa atividade o primeiro passo do processo PARFAIT é "Exercitar as operações do Sistema Legado para descobrir os cursos de execução normal e alternativo". Este passo foi retirado, levando-se em conta que o sistema ainda não existe e que será desenvolvido. O segundo passo já definido no processo pelo nome "Documentar os casos de uso de cada requisito" foi aproveitado. Nele, todos os requisitos do *software* são documentadas por meio de casos de uso. O outro passo alterado e definido para o PARFAIT/EA, foi "Elaborar casos de teste dos casos de uso priorizados pelo cliente". Esse passo foi alterado, pois na fase de ELABORAÇÃO o projeto já possuirá uma lista dos requisitos priorizadas pelo cliente identificadas na fase de CONCEPÇÃO. O último passo "Validar com os clientes cada caso de

uso criado" foi utilizado no PARFAIT/EA, pois é uma atividade importante uma vez que esse processo permite a participação do cliente em todo o projeto, principalmente para validar os artefatos produzidos. A participação do cliente em todo projeto baseia-se na prática ágil do método XP "Cliente Presente".

Como o processo é adaptativo e requer a participação do cliente, nesse momento o cliente pode modificar e/ou atualizar os requisitos especificados por meio dos casos de uso, caso haja necessidade.

Uma outra questão relevante analisada no estudo de caso, é com relação a atividade de teste. Para o desenvolvimento do *software* optou-se também por utilizar os mesmos critérios de teste funcional utilizados pelo PARFAIT (CAGNIN, 2005a), ou seja, Particionamento de Equivalência (MYERS, 2004) e Análise do Valor Limite (MYERS, 2004). Embora a maioria dos métodos ágeis utilize critérios estruturais para o teste de unidade, a escolha dos critérios de teste funcional baseou-se na definição dos requisitos, com objetivo de testar cada requisito separadamente.

A criação dos testes é realizada durante a fase de ELABORAÇÃO e esses são utilizados na fase de CONSTRUÇÃO para testar cada versão do sistema e, durante a fase de TRANSIÇÃO, para testar o sistema completamente antes de ser liberado para uso. A criação de testes antes de construir o sistema assemelha-se à prática "Testes constantes" do método ágil XP, em que os testes são criados com base nos cartões de estórias escritos pelo cliente e todo o s*oftware* é validado por meio de teste.

Com base no estudo de caso de reengenharia, observa-se que as outras atividades, "Desenvolver o diagrama de classes do sistema", "Documentar as modificações realizadas no diagrama de classes" e "Documentar as regras de negócio do software" são importantes também para o desenvolvimento do *software*, pois a utilização dessas atividades favorece a produção da documentação do projeto do *software*, contribuindo para futuras manutenções.

Ressalta-se que a atividade “Desenvolver o diagrama de classes do sistema” é apoiada pela linguagem de padrões de análise, cuja construção do *framework* é baseada. Com a especificação do sistema baseada na linguagem de padrões é possível, na fase de CONSTRUÇÃO, obter uma versão do novo sistema o mais rápido possível, a partir da instanciação do *framework.*

No caso em que os requisitos e as regras de negócio do sistema não pertencem à linguagem de padrões e, conseqüentemente, ao *framework,* é necessário documentar essas restrições nas atividades “Documentar as modificações realizadas no diagrama de classes” e “Documentar as regras de negócio do software”, a fim de que seja possível posteriormente adaptar a versão do *software* gerado a partir do *framework*. Isso é feito com a finalidade de se obter uma versão final do *software* de acordo com os requisitos definidos pelo cliente.

A fase de CONSTRUÇÃO do PARFAIT/EA é composta pelas atividades da fase de CONSTRUÇÃO do PARFAIT, com poucas adaptações. Na atividade “Executar os casos de teste no software”, o passo 4 “Executar o sistema alvo concomitantemente com o SL” foi retirado, pois não atende ao desenvolvimento do *software* e sim à reengenharia. Na atividade “Adaptar o software”, os dois últimos passos foram adaptados. No processo PARFAIT/EA esses passos são apoiados pelo cliente, sendo renomeados para: “Executar o software junto ao cliente” e “Adaptar o software de acordo com as mudanças exigidas pelo cliente”. Esses passos são importantes, pois é nesta atividade que são implementadas as regras de negócio já documentas na fase anterior. O uso desta atividade pode ser associada à prática ágil do método XP “Programação em Pares”, permitindo facilitar a implementação das adaptações, evitando erros ocorridos na escrita do código fonte. Torna-se possível, ainda nesta atividade a aplicação de outra prática ágil do XP, sendo a prática “Propriedade Coletiva de Código”, em que todos os integrantes do projeto têm acesso ao código fonte, podendo também adicionar trechos de código no *software*, quando necessário.

Ainda na fase de CONSTRUÇÃO, o estudo de caso de reengenharia apresenta versões do sistema alvo para serem confrontadas com as funcionalidades do sistema legado. Para apoiar o desenvolvimento do *software*, torna-se interessante a liberação de versões funcionais, em que essas versões serão validadas pela equipe de projeto junto ao cliente e liberadas para teste de aceitação. A liberação de versões funcionais pode ser comparada com a prática ágil "Versões Pequenas" do método ágil XP, em que versões freqüentes do *software* são liberadas. Para complementar esta fase, a prática ágil do método XP "Integração Contínua" deve ser aplicada, pois permite a integração do *software* cada vez que uma tarefa é finalizada, evitando perdas de funcionalidades na liberação da versão.

No processo de reengenharia PARFAIT, a fase de TRANSIÇÃO é composta por quatro atividades, sendo uma delas "Converter a base de dados do sistema legado". Em uma análise realizada, definiu-se que essa atividade ficasse isolada no processo PARFAIT/EA, pois ela só será utilizada quando realmente houver necessidade. A razão disso é a possibilidade de ocorrer casos em que exista um *software* operando na empresa cliente e seja necessário a conversão da base de dados do *software* antigo para a do novo *software* desenvolvido.

Uma outra adaptação necessária para o desenvolvimento foi realizada na atividade "Treinar os usuários finais". Essa atividade no processo PARFAIT é definida como não obrigatória, pois depende de recursos financeiros cedidos pela empresa cliente. Analisando o estudo de caso de reengenharia e adaptando para o desenvolvimento, essa atividade passou a ser obrigatória, pois o cliente não possui nenhum *software* com essas funcionalidades instalado, ao contrário da reengenharia em que o cliente já tem familiaridade com o sistema legado. O primeiro passo dessa atividade, que é "Mostrar a localização dos requisitos do sistema legado no sistema alvo" foi retirado, pois enquadra na mesma questão em que o *software* ainda não existe.

Uma outra adaptação necessária para essa última fase é a ordem na seqüência de execução das atividades. Com o isolamento da atividade “Converter a base de dados”, a seqüência de execução ficou da seguinte maneira: “Testar o software”, “Treinar os usuários finais” e por último “Elaborar o manual do usuário do software”. No projeto de reengenharia, com o uso do processo PARFAIT, uma versão funcional para uso é liberada somente no final permitindo a entrega do manual do usuário. Já no desenvolvimento de *software*, com o processo PARFAIT/EA versões funcionais são liberadas com freqüência, possibilitando a escrita do manual do usuário no final de cada ciclo de desenvolvimento ou somente na versão final do *software*.

Observando os resultados do estudo de caso de reengenharia, verificou-se a vantagem do uso de marcos de referência em todas as fases do processo PARFAIT/EA, o que possibilita ajustes necessários no planejamento do projeto. Com isso, definiu-se o uso de marcos de referência em cada fase do PARFAIT/EA, sendo também adicionada uma outra atividade denominada “Reuniões freqüentes”, e que também será executada em todas as fases do processo PARFAIT/EA. Esta atividade está representada na Figura 6.1 por um círculo vermelho, podendo ser realizada no início ou no fim da fase ou na conclusão de cada atividade.

Na Figura 6.1 apresentam-se todas as atividades alteradas (fonte do texto vermelho), as não alteradas (fundo amarelo) e as retiradas (fonte do texto azul) do processo PARFAIT para abstrair o esboço do processo de desenvolvimento PARFAIT/EA. Além disso, houve a necessidade de adicionar algumas atividades intrínsecas ao contexto de desenvolvimento ágil de *software* e estão apresentadas na Figura 6.1 na cor laranja.

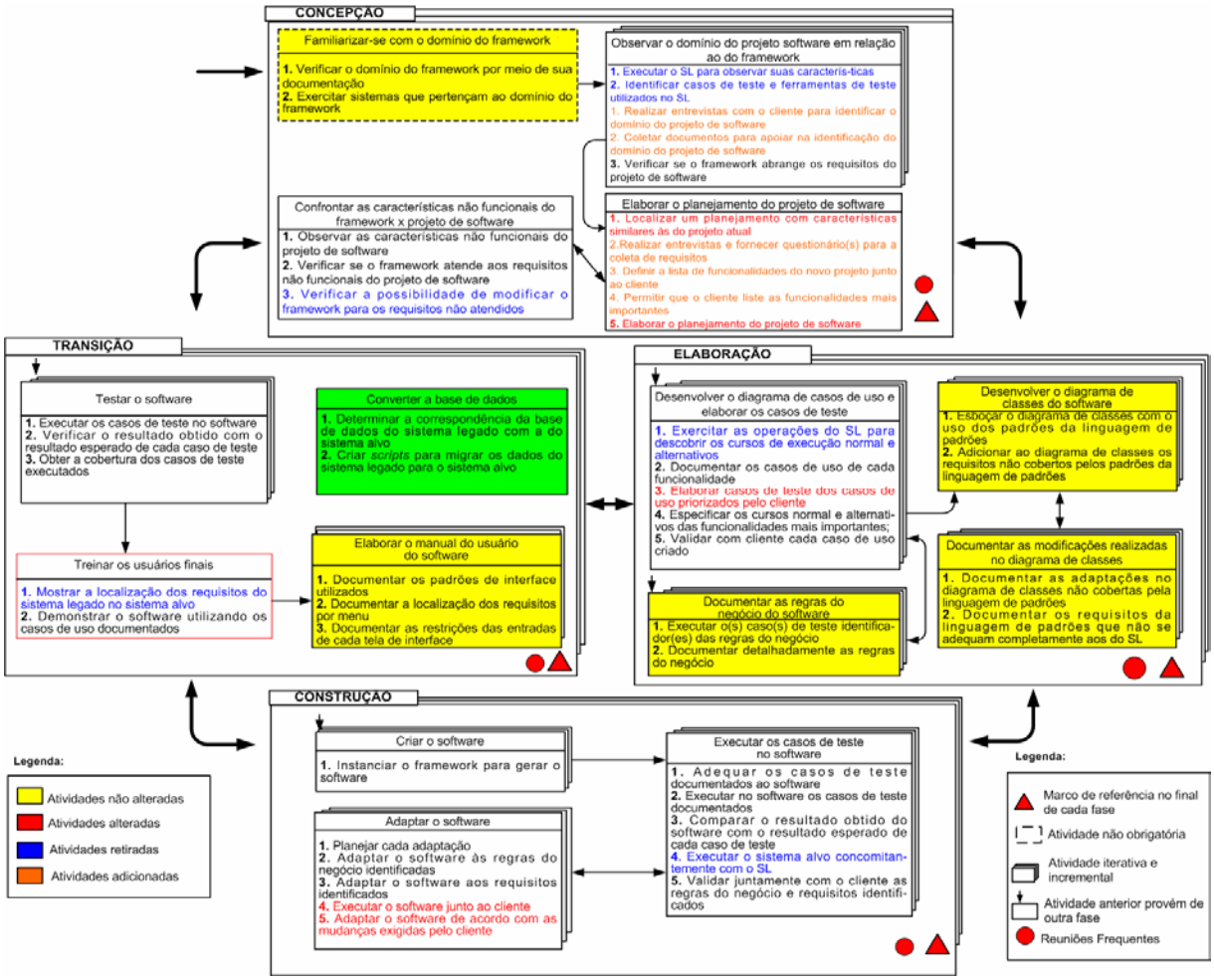


Figura 6.1: Alterações no PARFAIT para obtenção do esboço do PARFAIT/EA (GOMES *et al,* 2006)

A partir do esboço do processo (apresentado na Figura 6.2), verificou-se que as atividades retiradas atendiam somente a engenharia reversa no contexto de reengenharia, já às atividades adaptadas atendiam também a engenharia reversa e puderam ser utilizadas no desenvolvimento de *software,* na qual abrangem as tarefas de levantamento de requisitos/análise. As atividades não alteradas compreendem as atividades de análise no contexto de engenharia reversa e são utilizadas na reengenharia e no desenvolvimento de *software*.

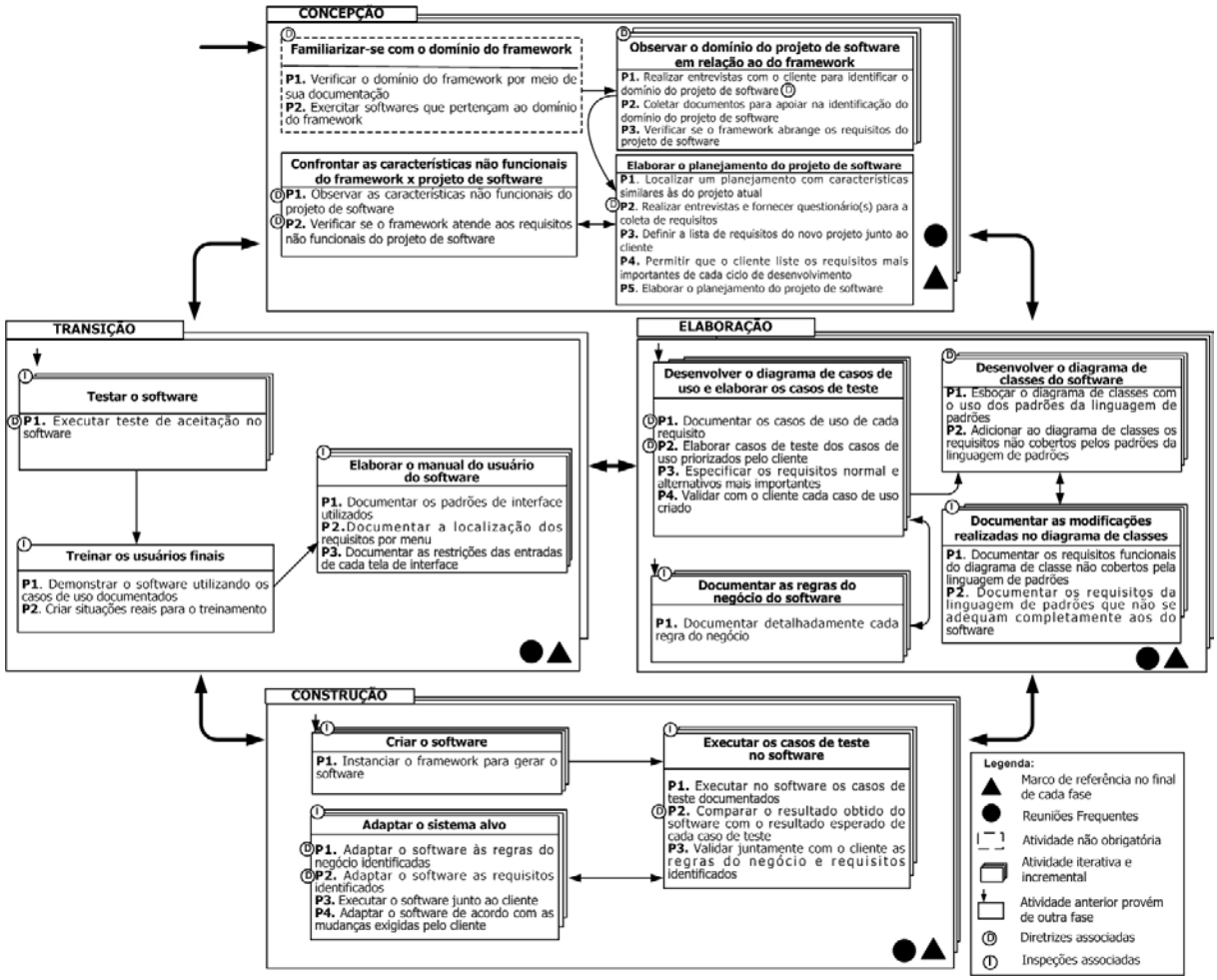


Figura 6.2: Esboço do processo PARFAIT/EA (adaptada GOMES *et al,* 2006)

No Quadro 6.5 mostram-se os documentos de entrada e documentos de saída de cada atividade em cada fase do processo PARFAIT/EA.

Quadro 6.5: Documentos de entrada e saída de cada atividade

| Fase CONCEPÇÃO | Documentos de Entrada | Documentos de Saída |
|---|---|---|
| Familiarizar-se com o domínio do framework | Exemplos de aplicações apoiadas pelo *framerwork*, documentação do *framework,* da LPA utilizada pelo *framework* e da ferramenta para instanciar o *framework* | Formulário para avaliação de conhecimento do domínio do framework |
| Observar o domínio do projeto de software em relação ao framework | Exemplos de projetos de *software* no mesmo domínio do projeto a ser desenvolvido | Documentos coletados com o cliente e um Resumo de cada reunião realizada com a presença do cliente |
| Elaborar o planejamento do projeto de software | Histórico de planejamentos de projetos de software similares ao projeto atual | Lista de Requisitos e Planejamento do Projeto de Software |

| | | |
|---|---|---|
| Confrontar as características não funcionais do framework x projeto de software | Documentos coletados na atividade "Observar o domínio do projeto de software em relação ao do framework", sendo: Documentos coletados com o cliente e um Resumo de cada reunião realizada na presença do cliente | Lista dos requisitos não funcionais |
| **Fase ELABORAÇÃO** | | |
| Desenvolver o diagrama de casos de uso e elaborar os casos de teste | Lista de requisitos priorizada pelo cliente de acordo com ciclo de desenvolvimento, definido no gabarito de planejamento | Diagrama de casos de uso, documentação dos casos de teste e documentação das classes de equivalência |
| Documentar as regras de negócio do software | Diagrama de classe e diagrama de caso de uso | Documentação das regras de negócio |
| Desenvolver o diagrama de classes do software | Diagrama de casos de uso e linguagem de padrões utilizada pelo *framework* | Diagrama de classe |
| Documentar as modificações realizadas no diagrama de classes | Diagrama de Classe do *software* | Gabarito da documentação das adaptações no diagrama de classe não cobertas pela linguagem de padrões e Gabarito da documentação dos requisitos da linguagem de padrões que não se adequam ao projeto de *software* |
| **Fase CONSTRUÇÃO** | | |
| Criar o software | Documentação do *framework*, da LPA e das ferramentas de instanciação do *framework* | *Software* |
| Executar os casos de teste no software | Documentação dos casos de teste, *software* | Relatório resumo de teste |
| Adaptar o software | Documentação das regras de negócios, *software* | *Software* modificado |
| **Fase TRANSIÇÃO** | | |
| Testar o software | *Software* e documentação dos casos de testes | *Software* testado. |
| Treinar os usuários finais | *Software* | - |
| Elaborar o manual do usuário do software | *Software* | Manual do usuário |

## 6.4 Estudo de Caso para avaliar o PARFAIT/EA

Para avaliação do processo PARFAIT/EA (GOMES e CAGNIN, 2007a), o planejamento de um estudo de caso foi definido e documentado de acordo com o formato proposto por (WHOLIN *et al*, 2000), em que estudos de caso são utilizados para monitorar projetos, atividades ou exercícios, com objetivo de acompanhar um atributo ou definir relacionamentos entre diferentes atributos. Os dados do estudo de caso são coletados para um propósito específico do estudo.

Esse planejamento tem como objetivo observar os resultados da aplicação do processo PARFAIT/EA e sua evolução no contexto de desenvolvimento de *software*.

### 6.4.1 Definição do Estudo de Caso

Esta fase do planejamento tem como objetivo evidenciar o que realmente pretende-se avaliar, estabelecendo as possíveis hipóteses para analisar o processo e avaliar sua aplicabilidade no desenvolvimento de *software.*

**Objeto de estudo**: PARFAIT/EA - Processo Ágil de Desenvolvimento baseado em *Framework*.

**Objetivo:** Conduzir um estudo de caso no contexto de desenvolvimento de *software* para avaliar o uso do processo PARFAIT/EA.

**Foco qualitativo:** Produção de documentação necessária para apoiar na produção do *software* que atenda as necessidades dos usuários e clientes, seguindo várias práticas de métodos ágeis e apoio computacional baseado em *framework.*

**Perspectiva:** A perspectiva do estudo de caso baseia-se no uso do processo pelo engenheiro de *software*.

**Contexto**: O estudo de caso foi realizado pela própria autora do processo a qual já havia executado uma análise de um estudo de caso de reengenharia para a definição das fases e atividades do processo PARFAIT/EA (Seção 6.2). O material necessário para o estudo foi: documentação da LPA GRN (BRAGA, 2002), documentação do *framework* GREN (BRAGA, 2003a), da ferramenta de instanciação GREN-Wizard (BRAGA, 2003b), da ferramenta de controle de versões GREN-WizardVersionControl (CAGNIN, 2004a), manuais do SGBD MySQL[17] e da linguagem de programação Smalltalk. O estudo de caso baseia-se em uma empresa real, no ramo de prestação de serviços. O *software* desenvolvido controla os serviços e produtos oferecidos por uma empresa incubadora[18] às empresas incubadas. Alguns dos serviços oferecidos por essa empresa são: cópias de xérox e envio de fax, bem como empréstimos de revistas de informática, administração, negócios, entre outras. Participou do estudo de caso a secretária, como usuário da empresa, e a autora do processo, como desenvolvedor.

[17] http://dev.mysql.com/doc/refman/4.1/pt/

[18] Empresa deste tipo tem como objetivo oferecer um ambiente flexível e facilidades em recursos como: qualificação, infra-estrutura, serviços básicos, como por exemplo: telefonia e acesso *web.*

### 6.4.2 Planejamento do Estudo de Caso

**Seleção do Contexto:** Para a condução do estudo de caso, um formulário foi fornecido para anotar o tempo gasto na execução de cada atividade e de cada ciclo de desenvolvimento. A autora do processo já possuía algum conhecimento dos recursos e ferramentas necessários para a condução do estudo, adquirido em um treinamento. Esse estudo é válido em um contexto específico do domínio de Engenharia de *Software*.

**Formulação das hipóteses:**

**Nulas:**

1. A documentação fornecida pelo processo, para a instanciação do *framework* e para a adaptação da versão gerada pelo *framework* é suficiente para apoiar o desenvolvimento de *software*.
2. Na definição da lista de requisitos a maior parte das regras de negócios é identificada.
3. Todas as práticas ágeis definidas em cada atividade do processo PARFAIT/EA são utilizadas em todas as iterações do processo.

**Seleção das variáveis:** As variáveis independentes são variáveis que podem ser manipuladas e controladas e as variáveis dependentes são as variáveis nas quais se observa o efeito das mudanças das variáveis independentes (WHOLIN *et al,* 2000).

- **Variáveis independentes:** A experiência do participante com os recursos necessários para a execução do estudo de caso (como: o uso do *framework* GREN, da linguagem de padrões GRN, da ferramenta de instanciação GREN-Wizard e da ferramenta de controle de versões GREN-WizardVersionControl, do SGBD MySQL, da linguagem de programação Smalltalk, dos critérios de teste funcional:

Particionamento de Equivalência e Análise do Valor Limite) e no domínio de *softwares* de prestação de serviços ajuda no desempenho e no uso do processo.

- **Variáveis dependentes:** 1) analisar se a lista de padrões de análise, o diagrama de classes e a lista de requisitos são suficientes tanto para apoiar a instanciação do *framework* quanto para apoiar as adaptações no *software* gerado, a fim de adequá-lo ao projeto de *software* definido na fase de CONCEPÇÃO; 2) quantidade de regras de negócio identificada em cada ciclo de desenvolvimento durante a definição da lista de requisitos; e 3) quantidade e identificação de práticas ágeis utilizadas em cada iteração do processo.

A primeira variável baseia-se na documentação produzida, verificando se essa documentação é suficiente para um projeto de *software*, a segunda variável relaciona-se na identificação de regras de negócio com a liberação de versões funcionais ao cliente e a terceira variável baseia-se na freqüência de uso de práticas ágeis em cada iteração do processo.

**Seleção dos indivíduos:** A técnica utilizada no estudo de caso é a amostragem por conveniência, executada pela própria autora do processo.

**Projeto do estudo de caso:** O estudo de caso é definido como objeto único, pois foi realizado por apenas um participante em um único estudo de caso.

**Instrumentação:** Para a condução do estudo de caso é necessário que a participante possua conhecimento em algumas técnicas. Os documentos são utilizados para treinamento, como a documentação de trabalho do processo PARFAIT/EA (GOMES *et al*, 2006), documentação da LPA GRN e do *framework* GREN, manual da linguagem de programação Smalltalk, dos critérios de teste funcionais Particionamento de Equivalência e Análise do Valor Limite, da ferramenta de instanciação GREN-Wizard e da ferramenta para controle de versões GREN-WizardVersionControl.

**Avaliação da validade:** Como ameaça à avaliação dos resultados tem-se: a validade de conclusão em que algumas informações e recursos estabelecidos dependem do conhecimento do engenheiro de *software*; a validade interna em que o estudo de caso foi realizado pela autora do trabalho, que já possui experiência nos recursos necessários pelo processo PARFAIT/EA adquirido em um treinamento, o que permite um melhor desempenho na execução desse estudo e a validade de construção em que estudos de casos mais complexos são necessários para avaliar o processo e a utilização do processo por uma empresa desenvolvedora de *software* pode contribuir para sua maturidade.

### 6.4.3 Operação do Estudo de Caso

**Execução:** O estudo de caso foi conduzido em duas etapas. A primeira etapa teve como objetivo identificar o domínio do projeto de *software* em relação ao *framework* disponível e avaliar a viabilidade em realizar o projeto. Para a realização desta etapa foram executadas as duas primeiras atividades da fase de CONCEPÇÃO do processo, sendo: "Familiarizar-se com o domínio do framework" e "Observar o domínio do projeto de software em relação ao do framework". Para a execução destas atividades foram gastas 72 horas, devido ao projeto apresentar uma particularidade diferente a projetos anteriores, como o controle de dois recursos, produtos e serviços, o que exigiu aprendizado.

A segunda etapa consistiu na confirmação do início do projeto de *software* e na aplicação por completo do PARFAIT/EA no desenvolvimento do *software*. Durante esta etapa foi registrado o tempo gasto na execução de cada atividade e em cada ciclo de desenvolvimento, como apresentado no Quadro 6.6.

Para este estudo de caso não foi utilizada nenhuma ferramenta de teste para apoiar a atividade de teste da fase de CONSTRUÇÃO. A técnica aplicada foi a técnica de Teste Funcional e os critérios foram "Particionamento de Equivalência" e "Análise de Valor Limite" (MYERS, 2004). O Teste de Aceitação (MYERS, 2004) foi também constantemente aplicado durante o estudo de caso, a fim de avaliar o *software* produzido juntamente com o cliente. Na atividade de teste da fase de TRANSIÇÃO do processo, apenas o Teste de Aceitação foi aplicado.

De acordo com o planejamento do estudo de caso, definido na Seção 6.4.2, três ciclos de desenvolvimento foram necessários. No primeiro ciclo foram disponibilizados os módulos do *software* cadastro e seus respectivos relatórios. Já, no segundo ciclo foram disponibilizados os módulos principais do *software*, sendo: o módulo de "Empréstimo de Revistas" e o de "Lançamento de Serviços", juntamente com os relatórios gerenciais. No terceiro ciclo foram realizadas apenas adaptações nas telas para uma melhor apresentação do *software.*

No Quadro 6.6 apresentam-se as atividades executadas em cada ciclo de desenvolvimento, bem como o tempo gasto em cada atividade. Já no Quadro 6.7 apresentam-se os dados coletados referentes às variáveis dependentes, definidas na Seção 6.4.2.

Quadro 6.6: Dados coletados nos ciclos de desenvolvimento

| **Fase** | **Atividade** | **Ciclo 1** | **Ciclo 2** | **Ciclo 3** |
|---|---|---|---|---|
| 1 | Observar o domínio do projeto de software em relação ao *framework* | 72h00 | 01h00 | - |
| 1 | Elaborar o planejamento do projeto de software | 00h20 | 00h30 | 00h20 |
| 1 | Confrontar as características não funcionais do *framework* x projeto de software | 00h20 | 00h20 | - |
| 1 | Reuniões freqüentes | 00h15 | 00h15 | - |
| 2 | Desenvolver o diagrama de casos de uso e elaborar os casos de teste | 19h30 | 18h00 | - |
| 2 | Documentar as regras de negócio do *software* | - | 00h06 | - |
| 2 | Desenvolver o diagrama de classes do *software* | 00h30 | 00h20 | - |
| 2 | Documentar as modificações realizadas no diagrama de classes | 2h30 | 00h20 | - |
| | Reuniões freqüentes | 00h15 | 00h15 | - |
| 3 | Criar o *software* | 00h20 | 01h06 | - |

| | | | | |
|---|---|---|---|---|
| 3 | Executar os casos de teste no *software* | 03h00 | 03h00 | - |
| 3 | Adaptar o *software* | 01h00 | 04h00 | 02h30 |
| 3 | Reuniões freqüentes | 00h15 | 00h15 | 00h15 |
| 4 | Testar o *software* | 01h00 | 01h00 | 01h00 |
| 4 | Treinar os usuários finais | 00h20 | 00h30 | 00h15 |
| 4 | Reuniões freqüentes | 00h15 | 00h30 | 00h15 |
| | **Total Geral do ciclo** | **100h55** | **31h12** | **04h20** |

Quadro 6.7: Dados coletados durante o estudo de caso

| **Dado coletado nas hipóteses nulas** |
|---|
| *Lista de padrões de análise utilizados, diagrama de classes e lista de requisitos foram suficiente para instanciar o framework em cada iteração do processo?*<br>**Resultado:** Sim, suficiente. |
| *Lista de padrões de análise utilizados, diagrama de classes e lista de requisitos foram suficientes para as adaptações realizadas no software em cada iteração do processo?*<br>**Resultado:** Sim, tais artefatos foram suficientes para adaptações realizadas no desenvolvimento. No entanto são necessários novos estudos para as adaptações (manutenção) após a entrega do *software.* |
| *Quantidade de regras de negócio identificada em cada ciclo de desenvolvimento*<br>**Resultado:** Apenas uma regra de negócio foi identificada no segundo ciclo, devido às mudanças ocorridas na empresa. |
| *Quantidade e identificação de práticas ágeis utilizadas em cada iteração do processo?*<br>**Resultado:** No primeiro ciclo foram utilizadas 100% das práticas ágeis definida no processo, sendo: cliente presente, metáforas, jogo do planejamento, detalhamento dos requisitos definidos, versões freqüentes, integração contínua e testes constantes, já no segundo ciclo foram utilizados 50 % das práticas definidas, sendo: cliente presente, jogo do planejamento, detalhamento dos requisitos definidos, integração continua e testes constantes e no terceiro ciclo foram utilizadas 30 % das práticas definidas, sendo: cliente presente, jogo do planejamento e testes constantes. |
| *Quantidade de regras de negócio identificada na definição da lista de requisitos?*<br>**Resultado**: Nenhuma regra identificada. |

## 6.4.5 Análise e Interpretação dos Resultados

A condução do estudo de caso não foi realizada com horários contínuos e todo o tempo gasto nas atividades em cada ciclo de desenvolvimento foi anotado.

Observou-se que a execução da primeira atividade da fase de CONCEPÇÃO, "Familiarizar-se com o domínio do framework" ajuda não somente no entendimento do

*framework,* como também permite analisar o uso dos padrões da LPA em projetos semelhantes, contribuindo para o entendimento do novo projeto.

Verificou-se que a maior parte do tempo gasto foi na atividade "Desenvolver o diagrama de casos de uso e elaborar os casos de teste", principalmente na elaboração dos casos de teste, como constatado por Cagnin (CAGNIN, 2004d). O uso de *framework* no projeto de *software* permite reusar diagramas de casos de uso de projetos anteriores semelhantes ao projeto atual, devido às funcionalidades semelhantes produzidas do domínio do *framework*, diminuindo o tempo para a descrição da funcionalidade. Na descrição dos casos de uso, foram gastas 3h10 e na elaboração dos casos de teste 34h20, o que corresponde a um total de 37h20 gastos nessa atividade em todo o projeto.

Pelo fato do projeto de *software* ser simples e possuir a maioria dos requisitos do *framework*, algumas atividades foram executadas em pouco tempo, como exemplo a atividade "Documentar as modificações realizadas no diagrama de classes", em que foi gasto 00h30 para a descrição dos atributos adicionados no diagrama de classe e a descrição do padrão que não atendeu completamente a regra de negócio do *software*. Tal regra determina que o prazo de empréstimo de uma revista seja de três dias e o atraso na devolução da revista é de dois dias de suspensão para cada dia de atraso. Essa regra de negócio foi identificada somente no segundo ciclo após sua implantação. No início do projeto essa regra de negócio não existia.

Neste estudo de caso não foi marcado o tempo gasto com a aplicação das inspeções nas atividades e o uso dos marcos de referência foi associado à atividade "Reuniões freqüentes", executada no final de cada atividade.

Com a totalização do tempo gasto, verificou-se que no primeiro ciclo foi gasto o maior tempo do projeto, o que caracteriza o processo "dirigido ao risco", pelo fato desse ciclo conter o núcleo do projeto de *software*, o qual define uma arquitetura sólida com o apoio do *framework*.

No terceiro ciclo, foram realizados apenas alguns ajustes nas telas do *software,* desabilitando funcionalidades presentes no *framework* e ausentes no projeto. Nesse ciclo foi gasto um total de 04h20.

A utilização da ferramenta GREN-WizardVersionControl apoiou a iteratividade do processo PARFAIT/EA e permitiu a entrega das versões do *software* funcionando mais rapidamente.

Um ponto importante do PARFAIT/EA definido neste estudo de caso é a utilização de diretrizes e inspeções aplicadas em algumas atividades do processo. O objetivo das diretrizes é apoiar a execução do passo correspondente e as inspeções de avaliar os artefatos produzidos, como exemplo a atividade "Criar o software", da fase de CONSTRUÇÃO, que possui uma inspeção com o objetivo de verificar se todos os padrões da LPA, utilizados para criar o diagrama de classes e os atributos adicionais, foram considerados na instanciação do *framework*. Algumas diretrizes (D) e inspeções (I) apóiam apenas um único passo na atividade e outras apóiam toda a atividade.

Com a aplicação do estudo de caso e o resultado apresentado em relação às práticas ágeis, observou-se a falta da definição de práticas ágeis existentes na literatura em algumas atividades, como na atividade "Adaptar o software". Após a condução do estudo de caso, definiu se para a essa atividade as práticas do método XP (BECK, 2000): projetar com simplicidade e propriedade coletiva do código, em que a primeira atividade foi modificada para "Projetar (adaptar) com simplicidade".

Pelo fato do estudo de caso ser um estudo observacional (WHOLIN *et al,* 2000), não foi possível obter dados conclusivos com relação às hipóteses formuladas. No entanto, verificou-se que a utilização de diretrizes e inspeções em algumas atividades do processo garante a Verificação e Validação (V&V) e contribuiu para a evolução do processo e para a qualidade do *software.*

De acordo com os resultados apresentados no Quadro 6.7, a diminuição do uso das práticas ágeis do ciclo um para o ciclo dois, é devido ao fato de que algumas práticas ágeis, como por exemplo, o uso de metáforas já estarem definidas no projeto. Já a diminuição das práticas ágeis do ciclo dois para o três, baseia-se no contexto do ciclo, em que foram realizadas apenas adaptações nas telas.

Em relação à não identificação de regras de negócios na lista de requisitos, pode ser justificada pelo fato do projeto ser simples. Vale ressaltar que a regra de negócio descrita no estudo de caso foi definida durante o andamento do projeto, após uma reunião com os funcionários da empresa incubadora.

Com a evolução do processo PARFAIT/EA a atividade "Converter a base de dados do software" da fase de TRANSIÇÃO foi retirada, no contexto em que o cliente não possui dados para conversão.

### 6.4.6 Discussão

Na condução do estudo de caso algumas dificuldades foram identificadas, como: utilização do *framework* GREN, em que é necessário ter conhecimento prévio na linguagem de padrões GRN e na linguagem de programação Smalltalk para a implementação das regras de negócio e dos requisitos específicos do sistema não fornecidos pelo *framework.* Adicionalmente, tem-se a não utilização da abordagem ARTe (CAGNIN, 2005a), em que optou-se somente pela utilização dos critérios de teste funcionais, Particionamento em Classe de Equivalência e Análise do Valor Limite.

## 6.5 Considerações Finais

Neste capítulo apresentou-se um processo ágil de desenvolvimento baseado em *framework,* o qual foi definido por meio de uma análise de um estudo de caso de reengenharia conduzido anteriormente e evoluído em um estudo de caso de desenvolvimento de *software*.

Tal processo tem como base o uso de *framework* de aplicação baseado em LPA, o qual facilita o engenheiro de *software* no entendimento do domínio do *framework,* na elaboração da documentação do projeto de *software* e na instanciação do *framework* para gerar a aplicação, como relatado por (GOMES e CAGNIN, 2007b).

Algumas restrições com a utilização do *framework* GREN em relação à ausência de alguns requisitos não funcionais (segurança, *backup*, etc) foram observadas.

Em relação aos trabalhos correlatos, observou-se que ao contrário do processo PARFAIT/EA, que é baseado em *framework,* o processo XwebProcess proposto por Sampaio *et al* (2004) não é baseado em nenhuma tecnologia de reúso ou ferramenta específica, já o processo easYProcess de Danta *et al* (2004), utiliza várias ferramentas livres para apoiar o desenvolvimento de *software*, mas não apoio computacional de *framework* baseado em LPA.

Para melhor avaliação da aplicabilidade do processo, o ideal é que um outro participante aplique o processo no desenvolvimento de *software* e utilize ferramentas de testes para apoiar a execução dos casos de testes.

No próximo capítulo, apresenta-se a extensão da ferramenta P.DOCTool (BIANCHINI, 2004) para apoiar a gerência de projetos.

# CAPÍTULO 7 EXTENSÃO DA FERRAMENTA P.DOCTOOL

## 7.1 Considerações Iniciais

Um dos sucessos de um *software* pode ser atribuído a sua evolução, com modificações necessárias para atender a sua especificação.

Nesse contexto, torna-se necessário o planejamento do projeto de *software*, em que é possível elaborar prazos, custos, gerenciar tempo e controlar projetos. A extensão da ferramenta P.DOCTool conduzida neste trabalho, visa proporcionar o planejamento e a execução de projetos, permitindo gerenciar processos ágeis, mantendo organizados os projetos.

Esse capítulo está organizado da seguinte forma: na Seção 7.2 apresenta-se a definição da extensão da ferramenta P.DOCTool, descrita no Capítulo 6, na Seção 7.3 apresenta-se um exemplo da utilização da ferramenta P.DOCTool e na Seção 7.4 apresenta-se as considerações finais desse capítulo.

## 7.2 Definição da extensão da Ferramenta P.DOCTool

A extensão da ferramenta P.DOCTool tem como objetivo gerenciar projetos de *software* que utilizem processos ágeis, cuja estrutura da documentação do processo seja baseada na do RUP. Essa é a principal diferença entre a ferramenta P.DOCTool e as ferramentas estudadas na (Seção 6.2).

A estrutura da base de dados da ferramenta segue a mesma definida na versão original da P.DOCTool, com alterações necessárias para sua extensão, discutidas nesta seção.

O objetivo em estender a ferramenta P.DOCTool é devido ao fato de a ferramenta já fornecer toda a parte de cadastro de processos de *software*, sendo necessário apenas adicionar a parte de gerência de projetos, além disso por ser baseada na *web,* facilita acesso por todos os integrantes do projeto.

Os requisitos da ferramenta foram divididos em três módulos: gerenciamento de usuários, processos e planejamento e execução de projetos. Os requisitos pertencentes aos módulos gerenciamento de usuários e processos já estão disponíveis na versão original da ferramenta, proposta por Bianchini (2004). Os requisitos do módulo planejamento e execução de projetos foram adicionados na extensão da ferramenta e são apresentados a seguir:

1. Cadastrar projetos de *software:* para isso é preciso selecionar qual processo será utilizado, como por exemplo, PARFAIT/EA ou PARFAIT, que já deve estar previamente cadastrado e associar a cada projeto os participantes.
2. Cadastrar as iterações do projeto e associar a cada iteração as atividades do processo que serão, (no caso do planejamento do projeto) e que já foram executadas (no caso da execução de projetos).
3. Cadastrar as reuniões associadas a cada iteração do projeto de *software.*
4. Cadastrar as versões funcionais do *software,* que serão liberadas em uma determinada iteração.
5. Cadastrar os requisitos do *software* e associar cada requisito a iteração (ões) na qual será desenvolvido, permitindo obter informações sobre a versão funcional na qual será disponibilizado. Para cada requisito cadastrado, associar as seguintes informações:

   a. os casos de teste necessários para garantir a corretude do requisito .
   b. as regras de negócios envolvidas com o requisito.
   c. os comentários que poderão ser adicionados a cada requisito pelos participantes do projeto na iteração corrente.
6. Disponibilizar opções para visualizar informações do projeto, das iterações, das versões funcionais e da iteratividade do projeto de *software*.
7. Fornecer aos programadores uma lista com as atividades do processo planejadas em cada iteração que servirá de guia para a execução do projeto.
8. Gerenciar as respostas obtidas durante os marcos de referência, ao final de cada iteração executada, as quais servirão de base para projetos futuros.

Com a evolução da ferramenta P.DOCTool e com os requisitos definidos no módulo de planejamento e execução de projetos, mantém-se um repositório dos projetos finalizados, facilitando a elaboração de estimativas de tempo e custo para novos projetos, o reúso de casos de testes e de planejamento de projetos.

Na Figura 7.1 apresenta-se o diagrama de classes da extensão da ferramenta P.DOCTool, no qual algumas classes do módulo processos são mostradas na figura com cor de fundo cinza e estão relacionadas com as classes do módulo de planejamento e execução de projeto.

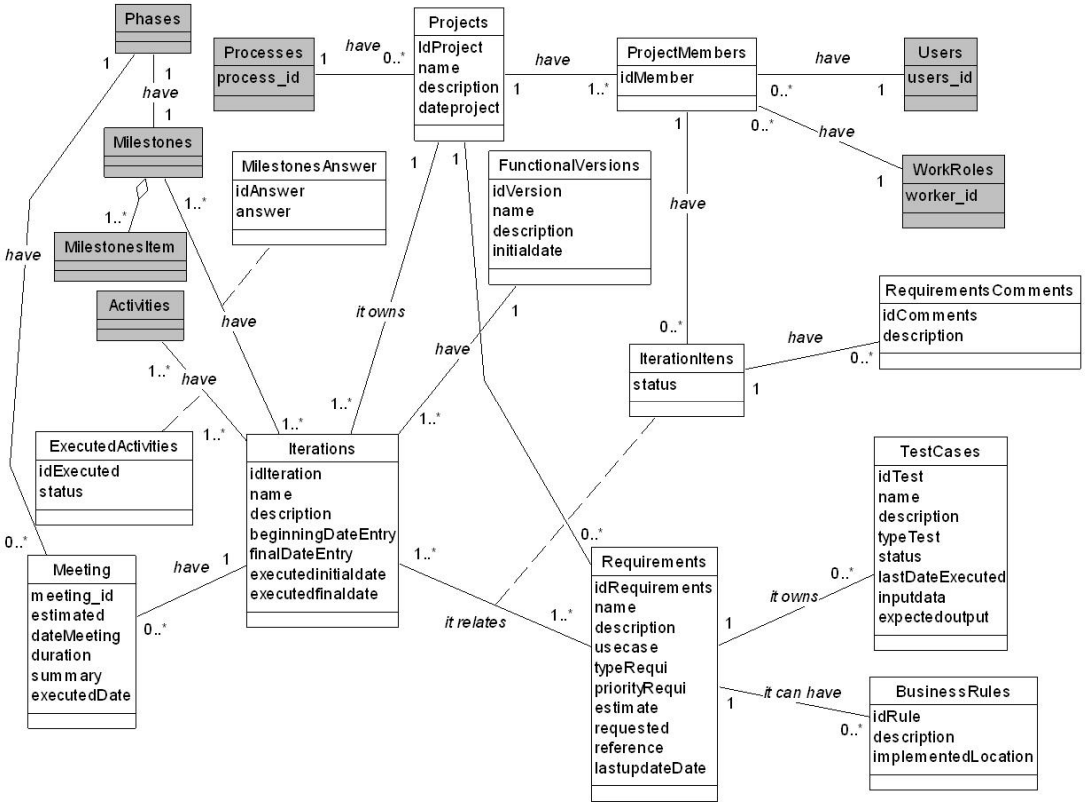


Figura 7.1: Diagrama de classes da extensão da ferramenta P.DOCTool

Na Figura 7.2 ilustra-se a arquitetura da ferramenta P.DOCTool, dividida em três camadas: a camada de usuários, a camada de funcionalidades e a camada de persistência. A camada de usuários é responsável pela entrada e saída de dados de acordo com as visões do usuário, a camada de funcionalidades representa os requisitos disponíveis na ferramenta, agrupados por módulos e a camada de persistência, que é responsável pela conexão do banco de dados e pelo armazenamento dos dados no SGBD MySQL.

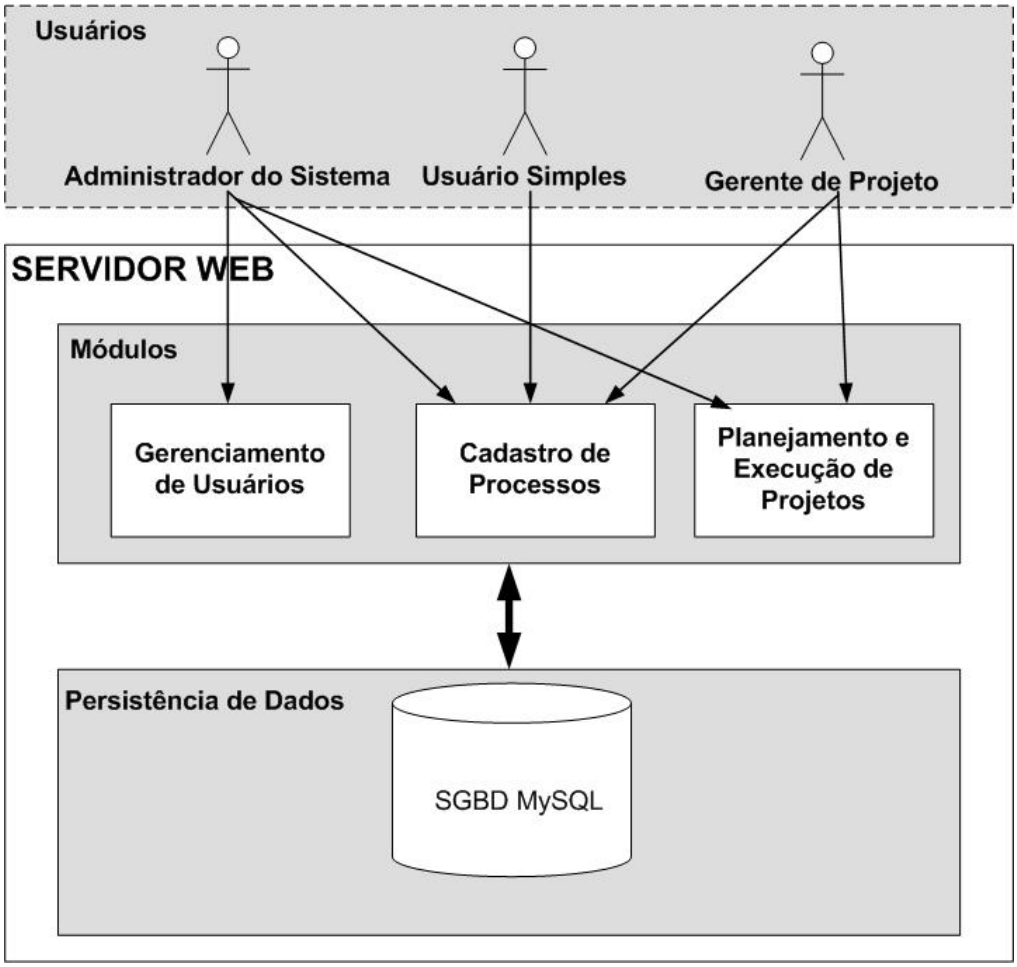


Figura 7.2: Arquitetura da ferramenta P.DOCTool

## 7.3 Exemplo de uso da Ferramenta P.DOCTool

Para exemplificar o uso da ferramenta P.DOCTool, foi cadastrado na ferramenta o projeto de *software* do estudo de caso apresentado na Seção 6.4.

Com a extensão da ferramenta P.DOCTool foi criada uma tela para ter acesso aos módulos da ferramenta: “Processes” e “Project Planning/Execution”. Na opção “Project Planning/Execution”, encontram-se todos os itens definidos na extensão da ferramenta. Essa tela é apresentada após o *login* na ferramenta e está ilustrada na Figura 7.3.

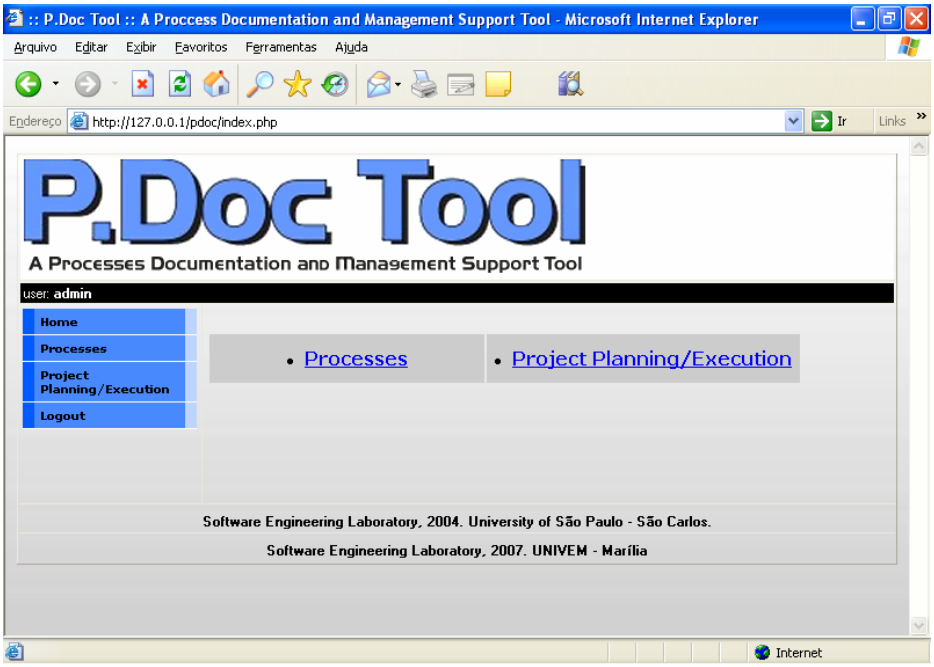


Figura 7.3: Tela de acesso aos módulos da ferramenta P.DOCTool

Para cadastrar projetos na ferramenta, é necessário que o processo a ser utilizado já esteja cadastro. Para isso, o primeiro módulo a ser acessado é o de processos, que já fazia parte da ferramenta. Na Figura 7.4 apresenta-se o cadastro de processos, com os botões: “New Process”, utilizado para cadastrar um novo processo, “Remove Selected”, utilizado para remover um processo selecionado e “Edit Selected”, utilizado para editar informações de um processo selecionado e inserir as suas fases.

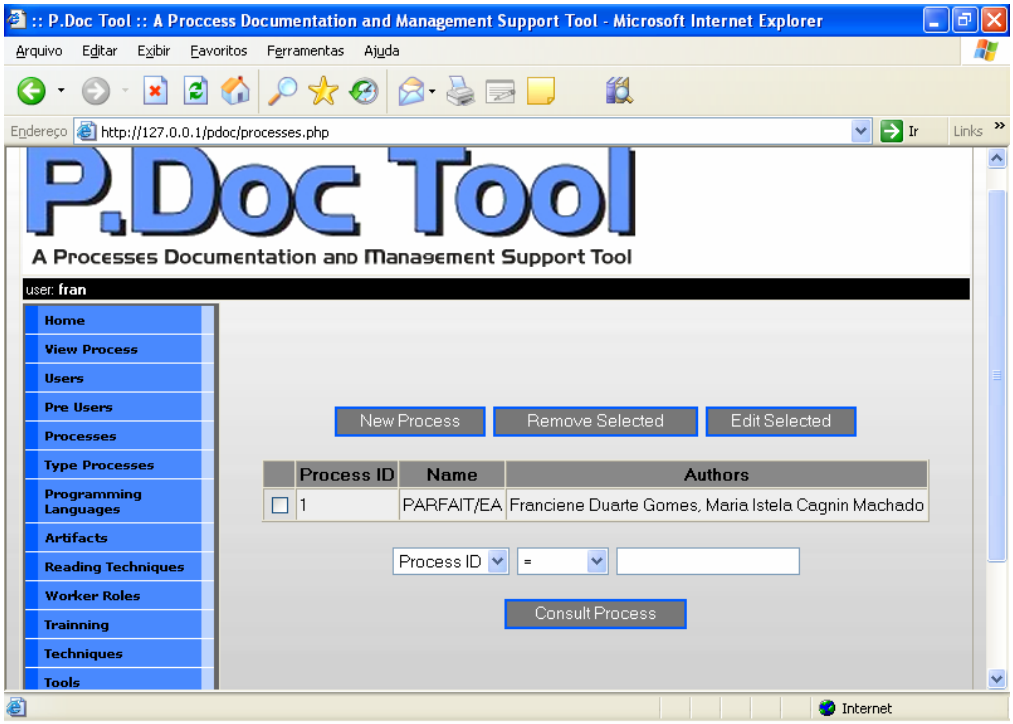


Figura 7.4: Tela do cadastro de processos

Conforme o usuário interage com a ferramenta no cadastro de processos, outras opções são disponibilizadas, além daquela para cadastrar as fases do processo, como o cadastrado dos marcos de referência, das reuniões e das atividades. Para cada atividade, devem ser associadas: as ferramentas utilizadas, os papéis, os artefatos de entrada e de saída e as diretrizes e inspeções de cada atividade.

Na Figura 7.5 apresenta-se a tela para visualização do processo PARFAIT/EA cadastrado na ferramenta, existente no módulo de cadastro de processo. A visualização de cada informação do processo é possível por meio de *links*, por exemplo, o *link* "CONCEPÇÃO" mostra informações da fase de CONCEPÇÃO do processo PARFAIT/EA, como ilustrado no lado direito da figura (marcos de referência e atividades).

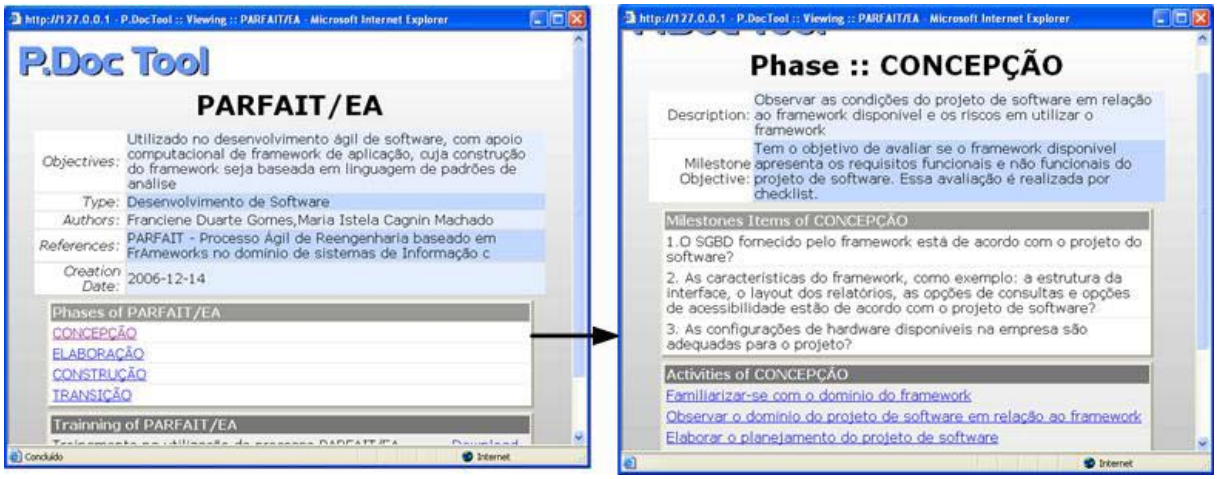


Figura 7.5: Tela para visualizar documentação de processos

Assim que o processo estiver cadastrado, o próximo passo é efetuar o planejamento do projeto. O objetivo do planejamento é organizar o projeto de *software* em iterações, associando a cada iteração, os participantes, os requisitos, as atividades executadas e uma versão funcional.

As Figuras 7.6, 7.7, 7.8, 7.9, 7.10 fazem parte do módulo de planejamento e execução de projetos e representam a extensão da ferramenta P.DOCTool.

Na Figura 7.6 apresenta-se o cadastro do projeto, que permite incluir outras informações como os participantes do projeto e as iterações. Na Figura 7.7 apresenta-se a tela para o cadastro dos participantes do projeto.

Neste exemplo foi cadastrado o projeto “Controle Empréstimo de Revista” e apenas um participante, sendo a própria autora do trabalho.

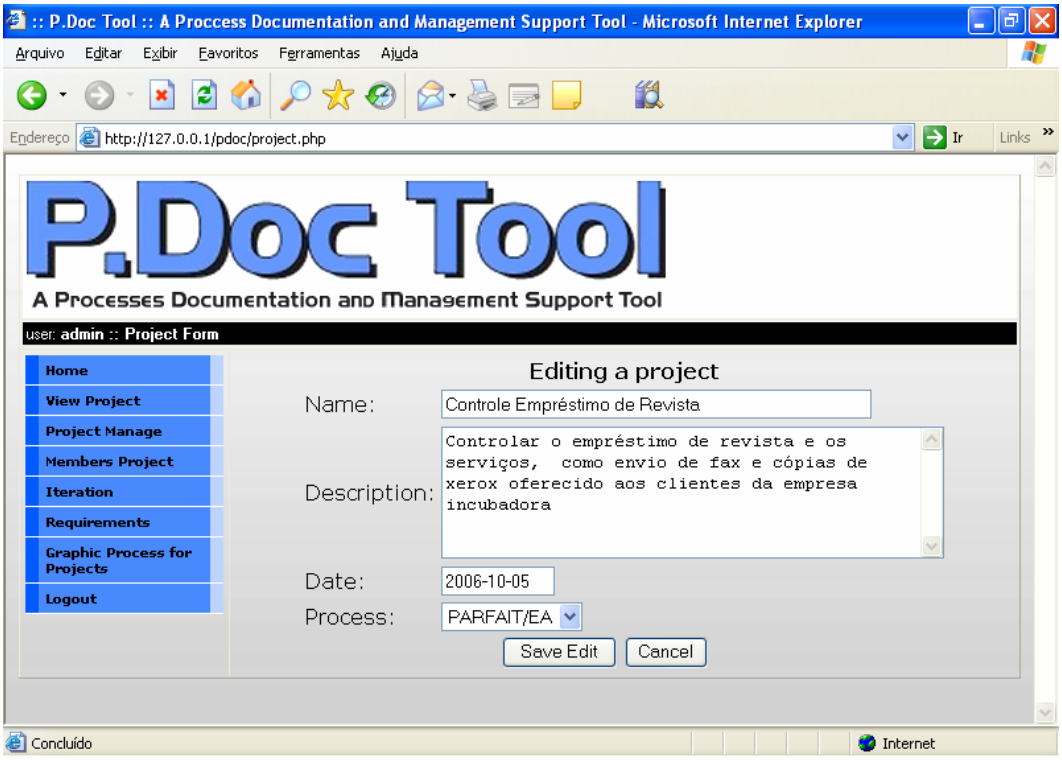


Figura 7.6: Tela do cadastro de projeto

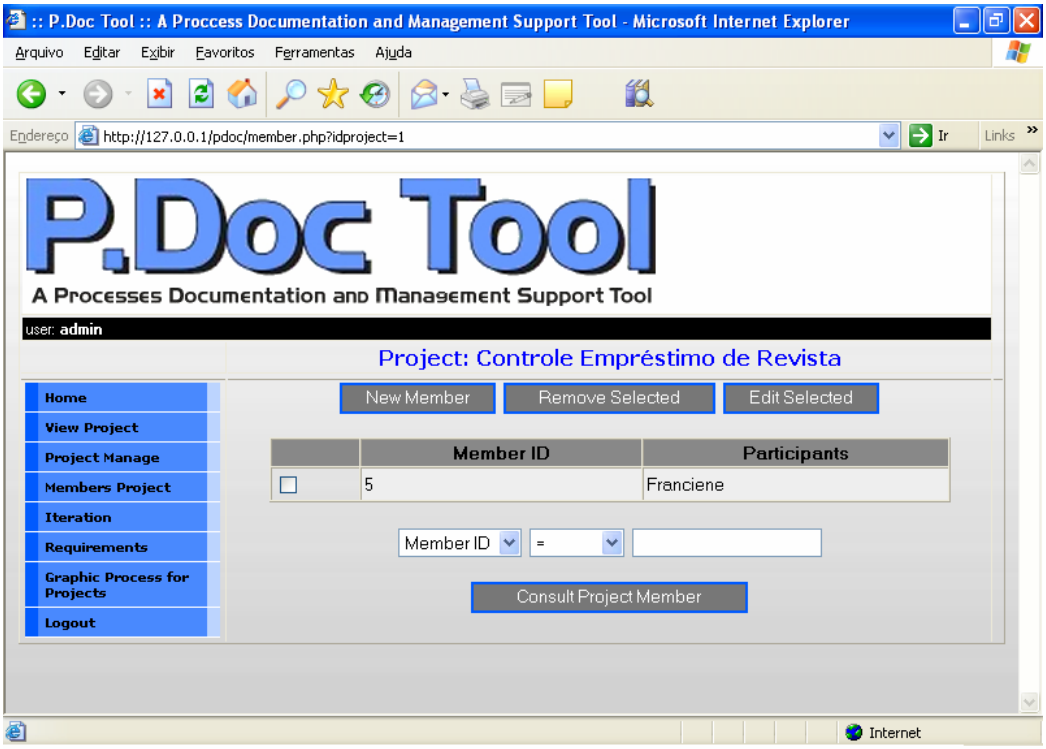


Figura 7.7: Tela do cadastro de participantes

Após o cadastro do projeto e dos participantes, o próximo passo é cadastrar as iterações definidas para o projeto. Para o projeto do exemplo foram cadastradas três iterações,

com os respectivos nomes: “Ciclo 1”, “Ciclo 2”e “Ciclo 3”. Na Figura 7.8 apresenta-se a tela para o cadastro de iteração com as opções para cadastrar: as atividades que serão executadas, as versões funcionais, os requisitos que irão compor a iteração e as respostas dos marcos de referência.

No exemplo, as atividades cadastradas na ferramenta são as apresentadas no Quadro 6.5, na Subseção 6.4.3 e estão ilustradas na Figura 7.9.

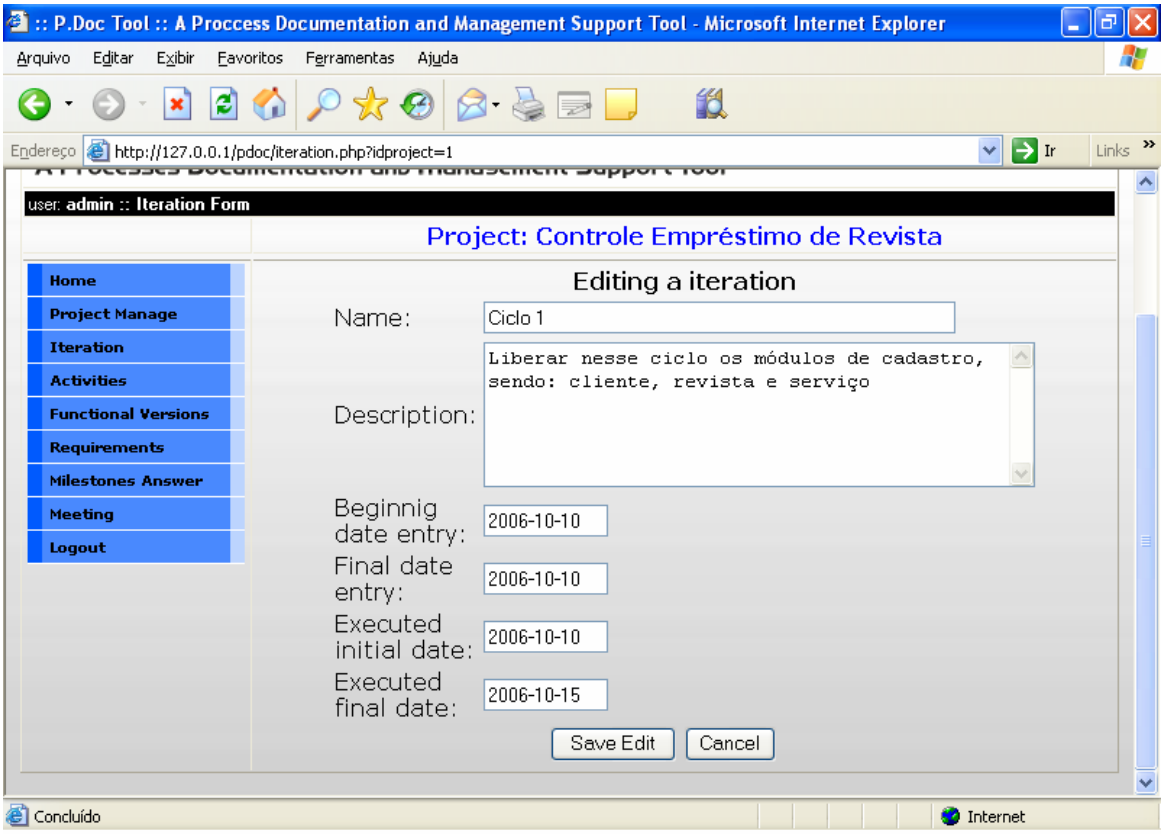


Figura 7.8: Tela cadastro de iteração

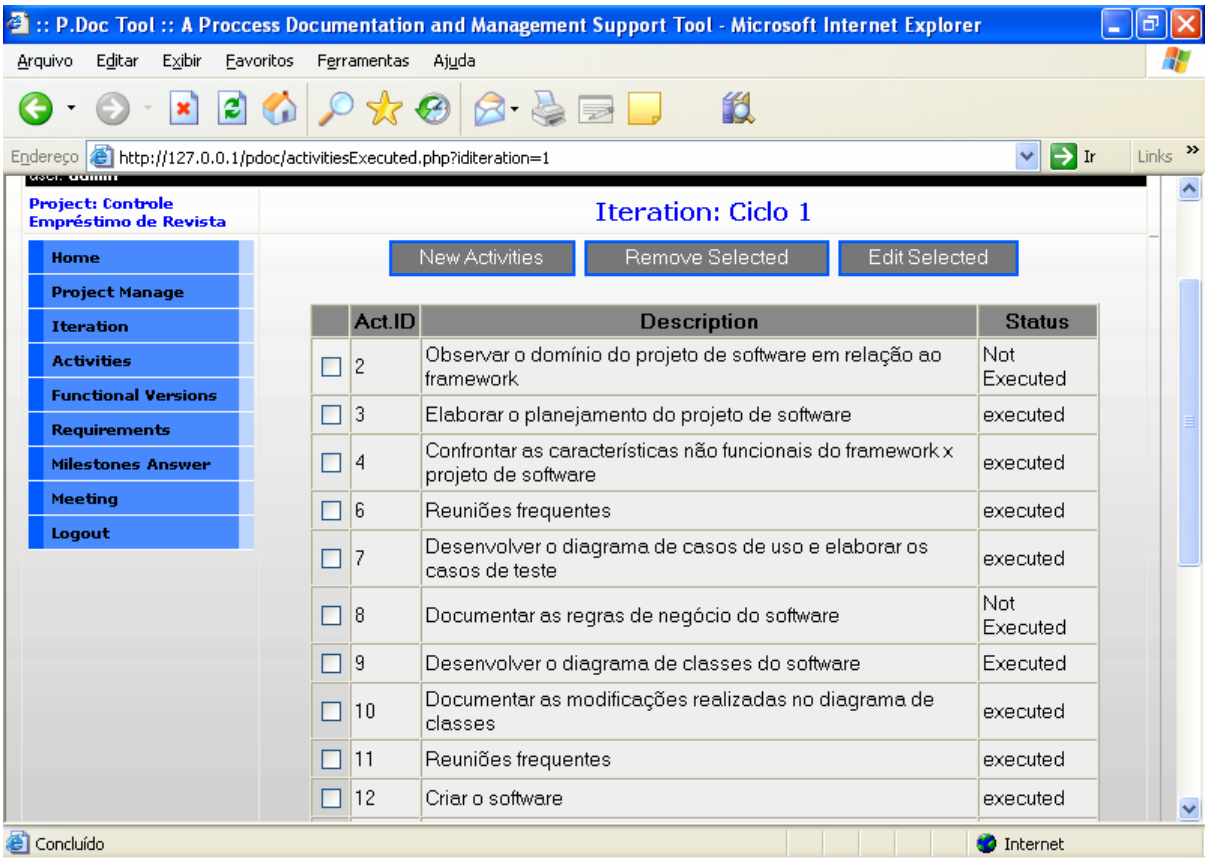


Figura 7.9: Tela cadastro de atividades executadas e não executadas

No cadastro de requisito, têm-se as opções para cadastro de: regras de negócios, casos de teste e comentários.

Os participantes do projeto podem descrever comentários sobre os requisitos em cada iteração do processo. Isso garante a comunicação entre os participantes do projeto. Na Figura 7.10 apresenta-se o comentário sobre o requisito “Cadastro de Cliente” da iteração “Ciclo 1”.

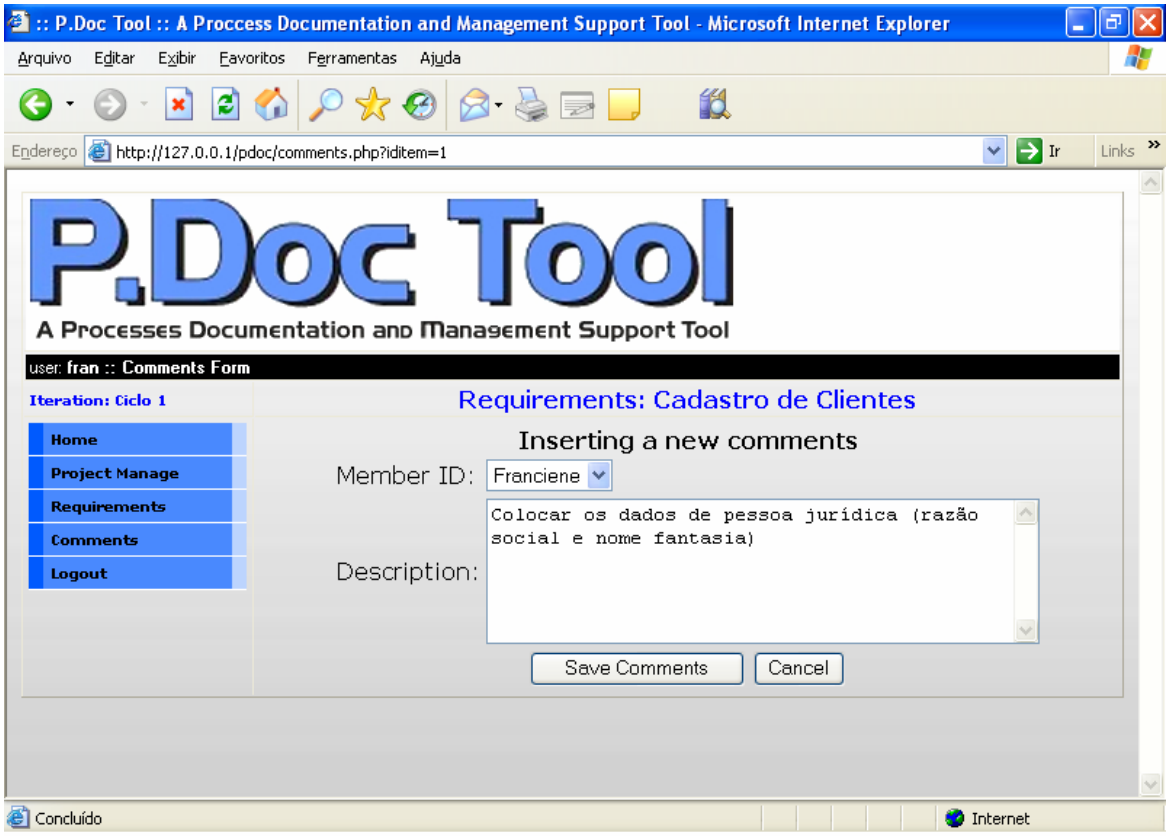


Figura 7.10: Tela comentário de requisito

Durante o andamento do projeto uma regra de negócio foi definida pelo cliente. Na Figura 7.11 apresenta-se a tela de cadastro do requisito “Cadastro de Empréstimo de Revista”, na Figura 7.12 a tela do cadastro da regra de negócio do requisito, descrita na Subseção 6.4.5 e na Figura 7.13 a tela do cadastro do caso de teste de aceitação.

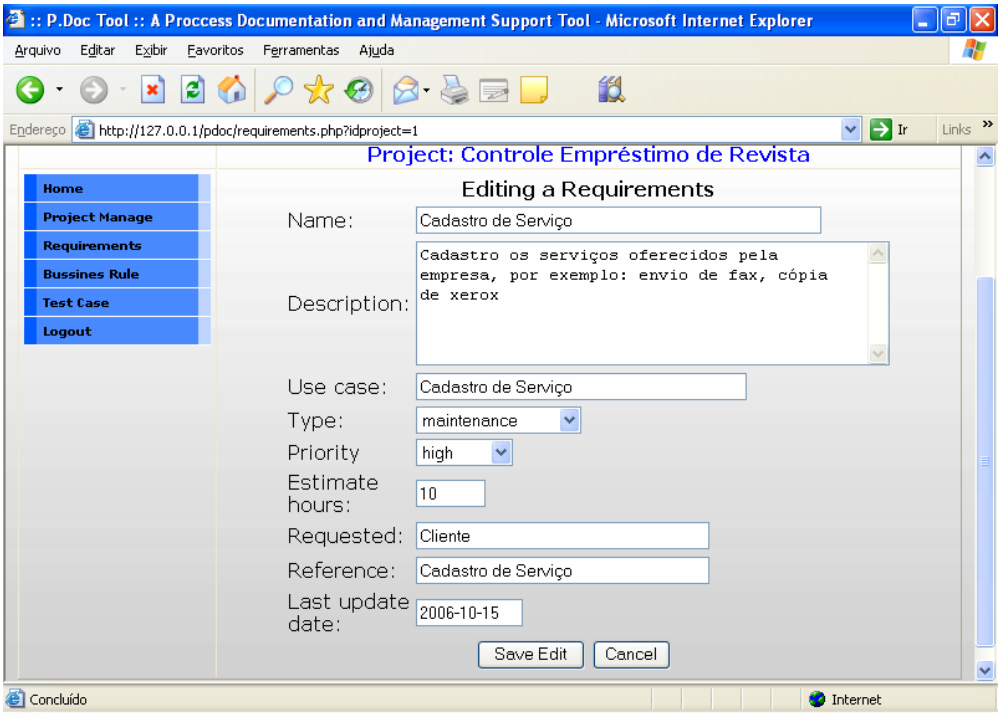


Figura 7.11: Tela cadastro de requisito

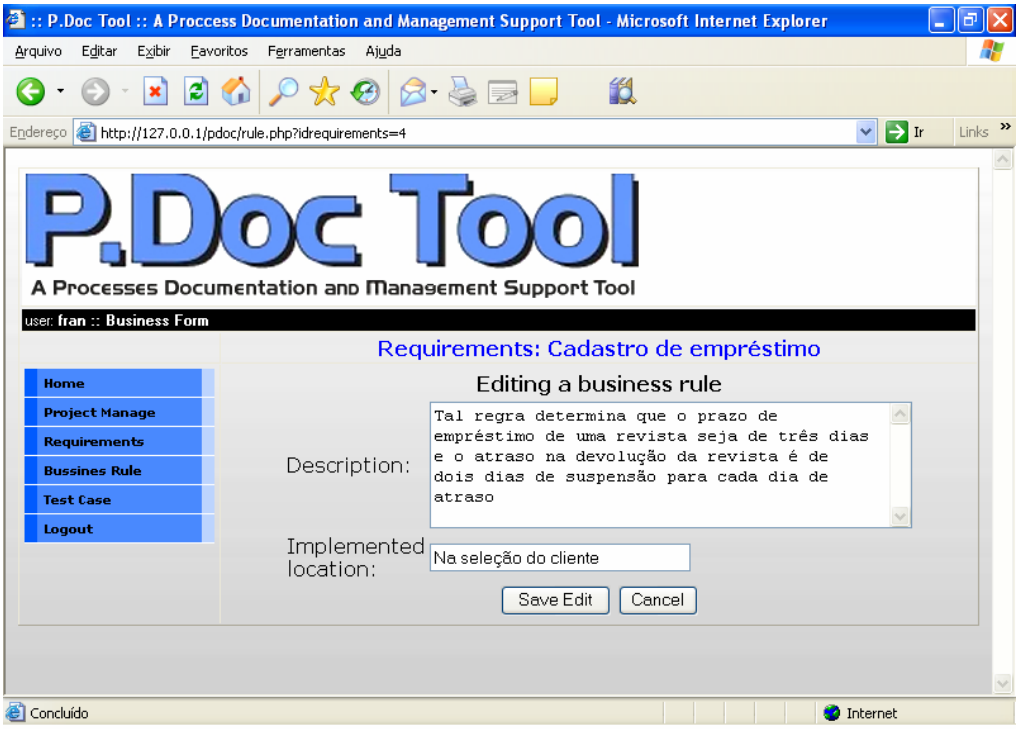


Figura 7.12: Tela cadastro de regra de negócio

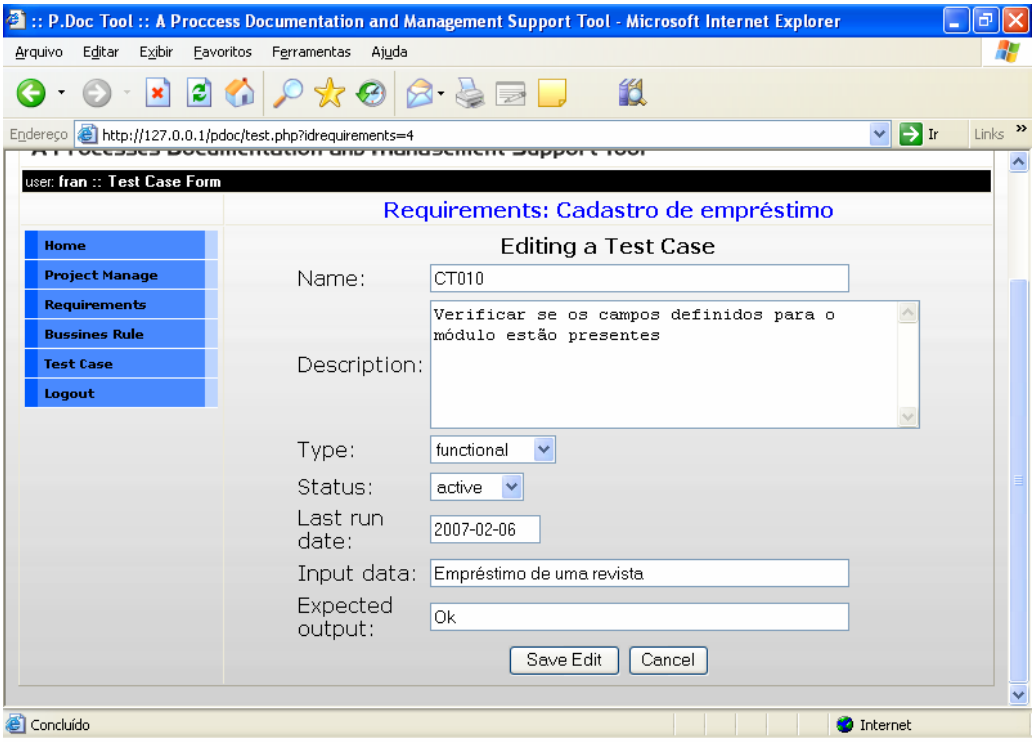


Figura 7.13: Tela cadastro de caso de teste

Com o planejamento e execução do projeto cadastrado, a ferramenta disponibiliza uma opção de relatório nomeada “View Project” (Figura 7.14), na qual todas as informações do projeto são apresentadas. Para cada iteração, apresentam-se as seguintes informações: os requisitos, as atividades executadas, as respostas dos marcos de referência e o conteúdo das reuniões freqüentes. Quando um requisito é selecionado, a ferramenta mostra as informações referentes as regra de negócio, aos casos de teste e aos comentários associados.

Na Figura 7.15 apresenta-se um relatório no formato de arquivo PDF (*Portable Document Format*), referente as atividades executadas e não executadas da iteração “Ciclo 1”.


Figura 7.14: Tela “View Project”

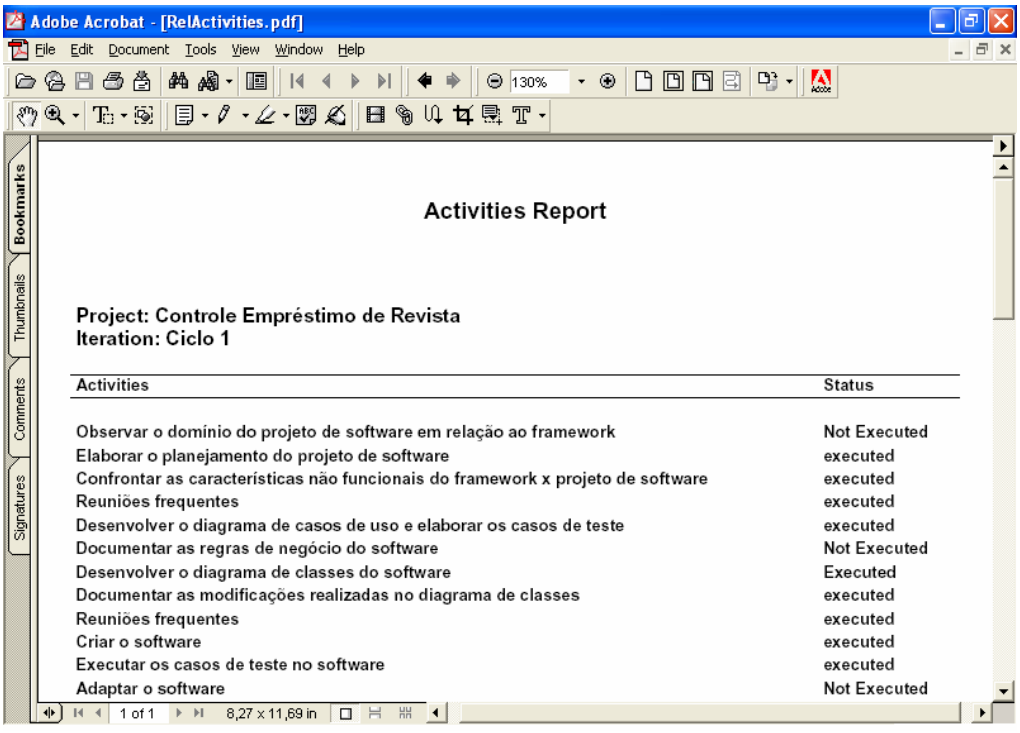


Figura 7.15: Relatório de atividades executadas e não executadas

A ferramenta disponibiliza também um gráfico (Figura 7. 16) que tem como objetivo mostrar, em porcentagem, a quantidade de projetos executados por processo.

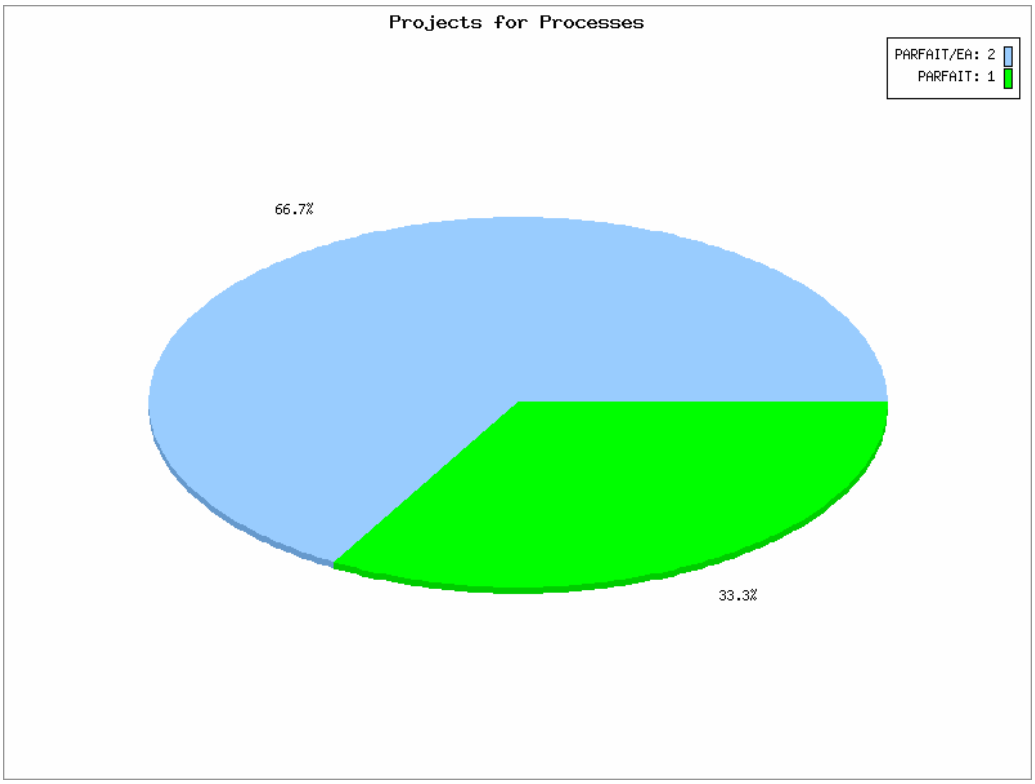


Figura 7.16: Gráfico de projetos por processo

## 7.4 Considerações Finais

Com o armazenamento de projetos tanto de reengenharia quanto de desenvolvimento de *software* na ferramenta P.DOCTool, os dados servirão de base para futuros projetos, como exemplo, para a atividade de planejamento, em que o responsável pela execução da atividade poderá estimar prazos com mais precisão, baseado em projetos semelhantes ao que estiver sendo executado.

Para uma melhor apresentação das informações mais importantes dos projetos, como por exemplo, o andamento das iterações, gráfico para representar de forma estatística o

andamento da iteração e barras de progresso para acompanhar a iteração deverão ser adicionados na ferramenta.

Uma característica presente na ferramenta P.DOCTool e ausente nas ferramentas apresentadas na Seção 5.2 é que a P.DOCTool apóia cadastro e gerência de processos de desenvolvimento de *software,* cuja estrutura da documentação do processo seja baseada na do RUP.

No próximo capítulo discutem-se as conclusões do trabalho desenvolvido, as limitações e sugestões de trabalhos futuros.

# CAPÍTULO 8. CONCLUSÃO

Neste trabalho foi definido um processo ágil de desenvolvimento baseado em *framework* (Seção 6.2), o qual fornece: atividades descritas por passos para facilitar sua execução, indicação de papéis e apoio computacional para cada atividade, documentos de entrada para orientar a aplicação da atividade, aplicação de diretrizes e inspeções específicas para cada atividade e para cada passo, a fim de validar os artefatos produzidos, utilização de gabaritos para facilitar a elaboração dos artefatos de saída para cada atividade e elaboração de documentação suficiente para documentar um projeto de *software* de acordo com o desenvolvimento ágil.

Neste contexto, o processo PARFAIT/EA representa mais um recurso do arcabouço ARA (Seção 4.2), atendendo somente ao desenvolvimento de *software* com apoio computacional de *framework.*

Com a condução de estudos de casos descritos nas Seções 6.3 e 6.4, foi possível observar que o processo possui características próprias para o desenvolvimento com o uso computacional de *framerworks,* por exemplo: atividades para analisar o domínio e as características funcionais e não funcionais disponíveis no *framework,* documentos de entrada para execução de cada atividade e critérios de avaliação definidos nos marcos de referência, específicos para *frameworks.*

A definição de práticas ágeis no PARFAIT/EA contribui para a entrega rápida do *software,* para a comunicação entre todos os participantes e para o incentivo na adaptação do *software* quando cliente e/ou desenvolvedor percebem que algo pode ser melhorado.

Uma outra contribuição do processo com a utilização de *framework* baseado em linguagem de padrões de análise, é a diminuição do tempo gasto em programação e o apoio na análise e na documentação do projeto.

A extensão da ferramenta P.DOCTool, apresentada na Seção 7.2, surgiu da necessidade de criar um repositório de projetos para facilitar a execução da atividade de planejamento, tanto do processo PARFAIT, quanto do processo PARFAIT/EA e permitir o cumprimento de prazos, já que com o acesso ao repositório torna-se mais fácil e previsível a condução do projeto com base em experiências adquiridas.

Observou-se que a ferramenta P.DOCTool gerencia a execução de projetos de desenvolvimento de *software* e de reengenharia, facilitando o acesso a documentação, bem com na inclusão ou alteração de documentação.

## 8.1 Limitações

Uma das limitações do trabalho realizado refere-se ao uso apenas do *framework* GREN para exercitar o processo PARFAIT/EA, o que não garante que as atividades do processo são genéricas para a utilização de outros *frameworks* e, consequentemente, para o uso de outras linguagens de padrões de análise além da GRN.

Neste trabalho, os critérios de teste funcional Particionamento em Classes de Equivalência e Análise do Valor Limite foram utilizados no estudo de caso conduzido para apoiar as atividades relacionadas a VV&T do PARFAIT/EA ao invés do uso da abordagem ARTe (CAGNIN, 2005a). Devido a isso, não é possível garantir que essa seja a melhor opção para a elaboração dos casos de teste durante o desenvolvimento de software com o apoio do PARFAIT/EA.

Uma outra limitação relaciona-se às hipóteses formuladas durante o planejamento do estudo de caso ,pois avaliam apenas algumas características do processo. Referente à condução de estudo de caso observa-se que há limitação em relação às pessoas que participaram do estudo e ao ambiente em que foi conduzido. Outra limitação é referente à utilização do PARFAIT/EA em apenas sistemas de pequeno porte.

## 8.2 Sugestões de Trabalhos Futuros

Para continuidade do trabalho realizado, a seguir são citadas algumas indicações de trabalhos futuros.

- Adaptar e melhorar o processo PARFAIT/EA para apoiar o desenvolvimento baseado em MDA (*Model Driven Architecture*) Unger *et al* (2003), com o objetivo de produzir documentação e modelos suficientes para construção de *software*s gerados a partir de ferramentas MDA.
- Reproduzir os experimentos apresentados neste trabalho utilizando um outro *framework* ou um gerador de aplicação cuja construção seja baseada em uma linguagem de padrões, com objetivo de obter novos resultados em relação à aplicabilidade do processo.
- Planejar um estudo de caso para aplicar as diferentes atividades de manutenção de *software* (corretiva, adaptativa, perfectiva e preventiva) no desenvolvimento de *software*, com o objetivo de avaliar se a documentação produzida pelo processo PARFAIT/EA é suficiente e se as atividades do processo apóiam a fase de manutenção.

- Aprimorar a ferramenta P.DOCTool com o objetivo de facilitar a apresentação das informações disponíveis no repositório de projetos, implantando barras progressivas para demonstrar o progresso na execução de cada iteração e acrescentar novos gráficos para uma melhor visualização das informações.
- Conduzir estudo de casos utilizando o PARFAIT/EA no desenvolvimento de sistemas de médio e grande porte, a fim de avaliar sua aplicabilidade.
- Planejar um estudo de caso de desenvolvimento de *software* para aplicar a abordagem ARTe (CAGNIN, 2005a) na elaboração dos casos de teste, por meio do reúso dos casos de teste disponíveis, com o objetivo de comparar os resultados obtidos com o uso desta abordagem com os resultados obtidos com o uso apenas dos critérios de teste funcionais utilizados neste trabalho.

# REFERÊNCIAS


ABRAHAMSSON, P.; SALO, O.; RONKAINEN, J.; WARSTA, J. **Agile Software Development Methods**. **review and analysis**. ESPOO (Technical Research Centre of Finland)' 2002. VTT Publications n. 478, 112p, 2002.

AMBLER, S. W. **Modelagem Ágil.** 1ª ed. Bookman, 352p, 2004a.

AMBLER, S. W. **The Agile Unified Process**. Site, http://www.ambysoft.com/unifiedprocess/ Acesso em Dezembro/2006b.

APPLETON, B. **Patterns and Software: Essential Concepts and Terminology**. Site, http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html. Acesso em Dezembro/2005, 1997.

BECK, K. **Extreme Programming explained: Embrace change**. 2ª ed. Addison-Wesley, 224p, 2000.

BECK, K.; BEEDLE, M.; VAN BENNEKUM, A.; COCKBURN, A.; CUNNINGHAM, W.; FOWLER, M.; GRENNING, J.; HIGHSMITH, J.; HUNT, A.; JEFFRIES, R.; KERN, J.; MARICK, B.; MARTIN, R. C.; MELLOR, S.; SCHWABER, K.; SUTHERLAND, J. THOMAS, D. **Manifesto for agile software development**. Site, http://www.agilemanifesto.org. Acesso em Outubro/2005,2001.

BIANCHINI, S. L. **Aspectos de Automatização da Documentação de Processos.** 2004**.** 72 f. Grau: Monografia (Graduação Bacharelado em Ciências da Computação) - Instituto de Ciências Matemáticas de Computação, Universidade de São Paulo, São Carlos, SP, 2004.

BOHER.K.A. **Architecture of the San Francisco frameworks.** IBM SYSTEMS JOURNAL, Vol 37*,* N.º 2, 1998. p. 156-169, 1998.

BRAGA, R. T. V.; GERMANO, F. S.; MASIERO, P. C. **GRN: Uma Linguagem de Padrões para Gestão de Recursos de Negócios**. Relatório Técnico, 2002.

BRAGA, T. V. **Um Processo para Construção e Instanciação de Frameworks baseados em uma linguagem de Padrões para um Domínio Específico.** 2003. 232 f. Grau: Tese (Doutorado em Computação) - Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos, SP, 2003.

BRAGA, R. T. V.; MASIERO, P. C. **Building a wizard for framework instantiation based on a pattern language**. In: OOIS'2003, International Conference on Object-Oriented

Information Systems, Geneva, Switzerland: Lecture Notes on Computer Science, LNCS 2817, Springer, p. 95-106, 2003.

BUSCHMANN, F. M.; ROHNERT, H.; SOMMERLAND, P. e STAL, M. **Pattern – oriented software architeture: a system of patterns**. John Wiley & Sons Ltd, New York, 1999.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. **UML - guia do usuário**; tradução de Fábio Freitas da Silva. Rio de Janeiro: Campus, p. 180-184, 2000.

BOSCH, P. M.; MATSSON, M.; BENGTSSON, P.; FAYAD, M. E. **Building Application Frameworks: Object-Oriented Foundations of Framework Design.** Nova Iorque: John Willey and Sons, p. 33-54, 1999.

CAGNIN, M. I. **PARFAIT: uma contribuição para reengenharia de software baseada em linguagens de padrões e frameworks.** 2005. 241 f. Grau: Tese (Doutorado em Computação) - Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos, SP, 2005a.

CAGNIN, M. I. **Documentação do PARFAIT**. Documento de Trabalho, Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2005b.

CAGNIN, M. I.; MALDONADO, J. C.; GERMANO, F.S.R.; CHAN, A. **Um Estudo de Caso de Reengenharia Utilizando o Processo PARFAIT.** In: SDMS'S 2003, Simpósio de Desenvolvimento e Manutenção de Software da Marinha, Rio de Janeiro, RJ, 10 p, 2003a. CD-ROM.

CAGNIN, M. I.; PENTEADO R. D. GERMANO, F.S.R **PARFAIT: Towards a Framework-based Agile Reengineering Process.** In: ADC'2003, Agile Development Conference, Salt Lake City, UTHA, USA: IEEE, p. 22-31, 2003b.

CAGNIN, M. I, MALDONADO, J.C.; GERMANO, F.S.; PENTEADO, R.D. **An Agile Reverse Engineering Process basead on a Framework**. In: WER'2003, 6th International Workshop on Requirements Engineering, Piracicaba, SP, p.240-254, 2003c.

CAGNIN, M. I.; MALDONADO, J. C.; PENTEADO, R.; BRAGA, R.T.V.; GERMANO, F. **GREN-WizardVersionControl: Uma Ferramenta de Apoio ao Controle de Versão das Aplicações Criadas pelo Framework GREN.** In: Sessão de Ferramentas'2004, Simpósio Brasileiro de Engenharia de Software, Brasília, DF, p.73-78, 2004a.

CAGNIN, M. I.; MALDONADO, J. C.; MASIERO, P. C.; PENTEADO, R. D.; BRAGA, R. T. **An Evolution Process for Application Frameworks.** In: I Workshop de Manutenção

Moderna de Software, em conjunto com o XVIII Simpósio Brasileiro de Engenharia de Software, Brasília, DF, CD-ROM, 8 p., 2004b.

CAGNIN, M. I.; MALDONADO, J. C.; BRAGA, R. T. V.; GERMANO, F. S.; PENTEADO, R. D. **Uma Ferramenta de Apoio ao Controle de Versão das Aplicações Criadas por um Framewok**. In: XXX Conferência Latino-Americana de Informática, Arequipa, Peru, publicado no CLEI Electronic Journal, p. 414-425, 2004c.

CAGNIN, M. I.; MALDONADO, J. C.; CHAN, A.; PENTEADO, R. D.; GERMANO, F. S. **Reuso na Atividade de Teste para Reduzir Custo e Esforço de VV&T no Desenvolvimento e na Reengenharia de Software**. In: XVIII Simpósio Brasileiro de Engenharia de Software, Brasília, DF, p. 71-85, 2004d.

CAMPELO, R. E. C. **XP-CMM2: Um Guia para Utilização de Extreme Programming em um Ambiente Nível 2 do CMM.** 2003. 207 f. Grau: Dissertação (Mestrado em Ciência da Computação) - Universidade Federal de Pernambuco, Recife, 2003.

CARVALHO, A. M. B. R; CHIOSSI, T. C. S. **Introdução à engenharia de software**. Campinas, SP: Editora da Unicamp, p. 21-41, 2001.

COAD, P.; LEFEBVRE, E.; LUCA, J. D**. Java Modeling In Color With UML: Enterprise Components and Process.** 1999, Cap. 6. p. 182-203.

COPLIEN, J.O. **Software Design Patterns: Common Questions and Answers**. In: Rising L., (Ed.), The Patterns Handbook: Techniques, Strategies, and Applications, Cambridge University Press, New York, 1998, p. 311-320.

COCKBURN, A. **Crystal Clear/A Human-Powered Methodology For Small Teams, including The Seven Properties of Effective Software Projects**. http://st- http://st-www.cs.uiuc.edu/users/johnson/427/2004/crystalclearV5d.pdf. Acesso em Novembro/2005, 1998.

COCKBURN, A. **Agile Software Development Draft version: 3b.** http://zsiie.icis.pcz.pl/ksiazki/Agile%20Software%20Development.pdf/. Acesso em Novembro/2005, 2000.

CHAN, A.; CAGNIN, M, I.; MALDONADO, J. C.; BRAGA, R. T. V. **Um ambiente para apoiar a utilização de padrões de software e requisitos de teste no desenvolvimento de aplicações.** In: SugarLoafPLOP' 2007, 7th Latin American Conference on Pattern Language of Programming, Porto de Galinha, PE - Brasil, 16p., (submetido). 2007.

CHRISTOPHER, A**. Christopher Alexander's Pattern Language.** IEEE Transactions on Professional Communication, Vol. 42, n.º 2, p. 117-122, 1999.

CHIROUZE, O.; DEJANOVSKI, A.; FULLER, B.; GIROLAMI, P. Site: http://xpweb.sourceforge.net. Acesso em Setembro/2006.

CHIKOFSKY, J. E.; CROSS, J. H. **Reverser engineering and design recovery: A taxonomy**. IEEE Software, Vol. 7, n.º 1. p. 13-17, 1990.

DANTA, V., GARCIA, F.P., LIMA. ***easYProcess*: Um Processo De Desenvolvimento De Software para Uso No Ambiente Acadêmico.** XII WEI - Workshop de Educação em Computação, Salvador, BA, 2004.

DEVEDZIC, V. **Software Patterns.** FON - School of Business Administration, University of Belgrade, Yugoslavia, 1998, 29 p. http://citeseer.ist.psu.edu/572487.html. Acesso em Dezembro/2005.

EXTREMEPLANNER. Site: http://www.extremeplanner.com/index.html. Acesso em Outubro/2006.

FAYAD, M. E., SCHMIDT, D. C. **Object-oriented Application *frameworks***. Communications of the ACM, Vol. 40, 10 p., 1997.

FAYAD, M. E., JOHNSON, R. E. **Domain-specific application framework: Frameworks experience by industry**. 1ª ed. John Wiley e Sons, 2000.

FUGGETTA, A. **Software Process: A Roadmap.** In: ICSE' 2000, Conference on Software Engineering, Limerick, Ireland, 8 p., 2000.

FREITAS, G. D.; WELFER, D.; D'ORNELLAS, M. C. **Padrão de projeto *AgregaComponente*.** In: SugarLoafPLOP' 2005 5th Latin American Conference on Pattern Language of Programming, Campos do Jordão - Brasil, 2005, p. 286-292.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Design patterns elements of reusable of object-oriented software**. 2ª ed. Addison-Wesley, 1995.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. Padrões de Projeto – Soluções Reutilizáveis de Software Orientado a Objetos. Bookman, Porto Alegre, 2000.

GARCIA, E. C.F.; PENTEADO, R. C, SILVA, A., BRAGA, R.T.V. **Padrões e Métodos Ágeis: agilidade no processo de desenvolvimento de software**. In: SugarLoafPLOP' 2005 5th Latin American Conference on Pattern Language of Programming, Campos do Jordão - Brasil, 2005, p. 156-169.

GOMES, F. D.; CAGNIN, M. I.; MALDONADO, J. C. **Esboço de um Processo Ágil de Desenvolvimento baseado em Framework**. In: CLEI'2006, 32th Conferência Latino Americano de Informática, Santiago – Chile, CD-ROM, 12 p., 2006**.**

GOMES, F. D.; CAGNIN, M. I**. Evolução de um Processo Ágil de Desenvolvimento baseado em framework.** In: JIISIC'07, 7th Jornada Ibero Americano de Engenharia de Software, Lima – Peru, 2007,10 p., 2007a**.**

GOMES, F. D.; CAGNIN, M. I**. Experiências de Uso de Padrões de Software/Frameworks de Aplicação na Reengenharia e no Desenvolvimento de Sistemas.** In: SugarLoafPLOP' 2007 7th, Latin American Conference on Pattern Language of Programming, Porto de Galinha, PE - Brasil, 11p., (submetido). 2007b.

HIGHSMITH. J. **What Is Agile Software Development?** Agile Software Development.CROSSTALK The Journal of Defense Software Engineering.Outubro, 2002. p. 4-9.

HORRIAN, H.; MAHMUD, S.; KARTHIKEYAN, S. **Requirements Engineering in Agile methods**. University of Calgary, 9 p., 2003.

JACOBSON . E. E, KRISTENSEN. B. B, NOWACK P. **Patterns in the Analysis, Design and Implementation of Frameworks.** Technical Report**,** Department of Computer Science, Aalborg University, 5 p., 1997.

JOHNSON, R. E. **Documenting Frameworks using Patterns**. In: OOPSLA'92, Conference on Object-Oriented Programming, Systems, Languages, and Applications., Vancouver, British Columbia, Canada, 14 p., 1992.

KHRAMTCHENKO. SERGUEI Comparing eXtreme Programming and Feature Driven **Development in academic and regulated environments.** CSCIE-275: Software Architecture and Engineering, Harvard University, 29 p., 2004.

KRUCHTEN, P. **The Rational Unified Process: An Introduction**. 2ª ed. Addison-Wesley, 2000.

LAUDON, K. C. **Sistemas de Informação gerenciais: administrando a empresa digital.** Tradução Arlete S. M. revisão técnica Erico V. M.; Belmiro, J. São Paulo: Prentice Hall, p: 7, 2004.

MATTSSON, M. "**Object-Oriented *Framework*s A survey of metholological issues**".1996. 128 f. Grau: Tese (Licenciatura) - Department of Computer Science, Lund University, 1996.

MALDONADO.J.C.; BARBOSA, E. F.; VINCENZI, A. M. R.; DELAMARO, M. E.; SOUZA, S. R. S.; JINO, M. **Introdução ao Teste de Software (Versão 2004-01)**. Notas Didáticas do ICMC. ISSN – 0103-2585, 2004.

MYERS, G. J. **The art of software testing**. 2ª ed. Wiley, 2004.

MONDAY, P.; CAREY, J.; DANGLER, M. **SanFrancisco Component Framework: An Introduction.** Addison-Wesley, 2000.

MÜLLER, E. **Métodos Ágeis: Uma aplicação do Scrum no SIMUPLAN**. 2004. 92 f. Grau: Monografia (Bacharelado em Ciência da Computação) - Universidade de Passo Fundo, Passo Fundo, 2004.

NAWROCKI, J.; JASINSKI, M.; WALTER, B.; WOJCIECHOWSKI**. Extreme Programming Modified: Embrace Requirements Engineering Practices.** In: RE' 2002, International Conference on Requirements Engineering, IEEE, 8 p., 2002.

NORFOLK, D. **Understanding DSDM**. PC Network Advisor. Vol. 067, p.15-18, 2004

PAETSCH, F.; EBERLEIN, A.; MAURER, F. **Requirements Engineering and Agile Software Development**. In:Proceedings of the 25th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'03), 6 p., 2003.

PAZIN. A. **SiGCli:A Pattern Language for Rehabilitation Clinics Management.** In: SugarLoafPLOP' 2004 Latin American Conference on Pattern Language of Programming, Porto das Dunas - Ceará, Brasil, 25 p., 2004.

PEDRYCZ, Witold; PETERS, James F. **Engenharia de software: teoria e prática**. Rio de Janeiro: Campus, 2001. 602p.

PFLEEGER, S. L. **Engenharia de software: teoria e prática**. 2ª ed. São Paulo: Prentice Hall, 2004.

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

PREE, E, W.; POMBERGER, G.; SCHAPPERT, A.; SOMMERLAD, P. **Active guidance of framework development**. *Software* - Concepts and Tools, Springer – Verlag, Vol. 16, n.º 3, p. 94-103, 1995.

RÉ, R. **Um processo para construção de frameworks a partir da engenharia reversa de sistemas de informação baseados na Web: Aplicação ao domínio dos leilões virtuais.** 2005. 143 f. Grau: Dissertação (Mestrado em Computação) - Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos, SP, 2002.

RIEHLE, D. **Composite Design Patterns.** In: OOPSLA'97, Communications of the ACM, 218-228 p., 1997.

SAMPAIO, A., VASCONCELOS, A., SAMPAIO, P. R. F. **Design and Empirical Evaluation of an Agile Web Engineering Process.** XVIII Simpósio Brasileiro de Engenharia de Software, Brasília – DF, Brasil, p. 194-209, 2004.

SOARES, M. S. **Comparação entre Metodologias Ágeis e Tradicionais para Desenvolvimento de Software**. INFOCOMP Journal of Computer Science, Vol. 3, n.º 2, p. 8-13, 2004.

SOARES, M. S. **Metodologias Ágeis Extreme Programming e Scrum para o Desenvolvimento de Software**. Revista Eletrônica de Sistemas de Informação, Vol. 3 n.º 1 Campo Largo - PR, 2004.

SOUZA, G. T.; PIRES, C. G.; BELCHIOR, A. D. **Padrões de Requisitos para Especificação de Casos de Uso em Sistemas de Informação.** In: SugarLoafPLOP' 2005, 5th Latin American Conference on Pattern Language of Programming, Campos do Jordão, SP - Brasil, 42-57p., 2005.

SOMMERVILLE, I. **Engenharia de Software**; tradução André Maurício de Andrade Ribeiro; revisão técnica kecki Hirama. São Paulo: Addison Wesley, 2003.

SCHMIDT, D. C.; JOHNSON, R.; E.; FAYAD, M. **Software Patterns.** Site: http://www.cs.wustl.edu/~schmidt/CACM-editorial.html. Acesso em Dezembro/2005, 2005.

TALIGENT. **Building object-Oriented *frameworks*.** Taligent Inc. Write paper, 1997.

UNGER, K. H.; FLOR, T.; VÖGLER, G. **Model Driven Architecture Development Approach for Pervasive Computing.** In: OOPSLA'03, California - USA. ACM, p 314-315, 2003.

VERSIONONE. Site: http://www.versionone.net. Acesso em Setembro/2006.

WELLING, L.; THOMSON, L. **PHP e MySQL desenvolvimento Web**. 3ª ed. Rio de Janeiro: Campus, 2005.

WHOLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M.; REGNELL, B.; WESSLÉN, A. **Experimentation in software engineering**: An introduction. Kluwer, 2000.

XPLANNER. Site: http://www.xplanner.org/. Acesso em Setembro/2006.

ZWARTJES, G; GEFFEN, J.V. **An Agile Approach Supported by a Tool Environment for the Development of Software Components.** 2005. 128 f. Grau**:** Dissertação (Mestrado em Ciência da Computação), Universidade de Pretoria, Pretoria – South Africa, 2005.

KRASNER, G. E.; POPE, S. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. Journal of Object Oriented Programming, 1 (3): 26-49, August/September, 1998.

# GLOSSÁRIO

| Prática Ágil | Significado | Método Origem |
|---|---|---|
| Cliente presente | Permite a participação do cliente em todo o desenvolvimento do *software,* contribuindo para o desentendimento de dúvidas e evolução do *software* | XP (BECK, 2000), *Scrum,* FDD, DSD, ASD, *Crystal Clear e Crystal Orange* ( ABRAHAMSSON *et al*, 2002) |
| Detalhamento nos requisitos definidos | Exige que os requisitos essenciais sejam definidos logo no início do projeto | DSDM (ABRAHAMSSON *et al*, 2002) |
| Integração contínua | Integram o *software* várias vezes ao dia, cada vez que uma tarefa é finalizada | XP (BECK, 2000) |
| Jogo do planejamento | Utiliza estimativas de custo fornecidas pelos programadores para determinar o que necessita ser feito e o que pode ser adiado no projeto, permitindo que o cliente decida o que é menos importante e o que pode ser desenvolvido em uma próxima versão do *software* | XP (BECK, 2000) |
| metáforas | Cria descrições comuns do *software*, evitando termos técnicos | XP (BECK, 2000) |
| Propriedade coletiva do código | Mantêm todo o código disponível a todos os membro da equipe, podendo qualquer membro da equipe adicionar um trecho de código no *software* | XP (BECK, 2000) |
| Testes constantes | Valida todo o projeto de *software* com o uso de teste, em que teste de unidade são criados antes do código e são utilizados em todo o projeto. Os clientes criam estórias que demonstram as características do sistema | XP (BECK, 2000) |
| Versões freqüentes | Libera versão do *software* em funcionamento o mais rápido em ciclos pequenos de desenvolvimento | XP (BECK, 2000) |
| Projetar com simplicidade | Orienta a simplicidade no desenvolvimento do *software* | XP (BECK, 2000) |

# APÊNDICE A

A seguir, descrevem-se cada fase que compõem o processo PARFAIT/EA, bem como suas atividades, os passos para executar cada atividade, diretrizes e inspeções.

## FASE DE CONCEPÇÃO

**Atividade: Familiarizar-se com o domínio do framework (opcional)**

**Papel:** Analista de Sistemas e Programadores.
**Apoio Computacional:** Editor de texto.
**Documentos de Entrada:** Exemplos de aplicações instanciadas no *framerwork*, documentação do *framework,* da LPA (Linguagem de Padrões de Análise) utilizada para a construção do *framework* e da(s) ferramenta (s) utiliza para instanciação do *framework*.
**Prática utilizada: -**
**Documentos de Saída:** Formulário para avaliação de conhecimento.

**Passos**:

**P1.** Verificar o domínio do *framework* por meio de sua documentação.
**P2.** Exercitar *softwares* que pertençam ao domínio do *framework*.

**D.** O analista de sistemas e os programadores devem responder as questões do "Formulário de avaliação de domínio" (Quadro 1), para avaliar o conhecimento sobre o domínio do *framework*.

Quadro 1 - Formulário de avaliação conhecimento do *framework*

| | | | |
|---|---|---|---|
| 1.Como você classifica seu conhecimento no domínio do *framework*? | a. ( ) Bom | b. ( )Regular | c. ( )Não possui conhecimento |
| 2. Já utilizou o *framework* em algum projeto de desenvolvimento de *software*? | a. ( ) Muitas vezes | b. ( )Raramente | c. ( ) Nunca usou |
| 3. Qual seu conhecimento no dominio da linguagem de padrões do framework? | a. ( ) Bom | b. ( )Regular | c. ( )Não possui conhecimento |
| 4. Qual seu conhecimento nos recursos necessários para o uso do *framework*? | a. ( ) Bom | b. ( )Regular | c. ( )Não possui conhecimento |
| 5. Qual seu conhecimento na linguagem de programação e na linguagem de padrão de análise do *framework*? | a. ( ) Bom | b. ( )Regular | c. ( )Não possui conhecimento |
| 6. Qual seu conhecimento na linguagem de modelagem UML? | a. ( ) Bom | b. ( )Regular | c. ( )Não possui conhecimento |
| 7. Já leu a documentação do *framework* disponível? | a. ( ) Sim | b. ( )Não | c. ( ) Comecei a ler |
| 8. Qual seu conhecimento no SGBD fornecido pelo *framework*? | a. ( ) Bom | b. ( )Regular | c. ( )Não possui conhecimento |

| **Pontos:** | **Pesos:** | **Avaliação:** |
|---|---|---|
| **a** = 2 ponto(s)<br>**b** = 1 ponto(s)<br>**c**= 0 ponto(s) | Questão 1 = 2,<br>Questão 2 = 1<br>Questão 3 = 2<br>Questão 4 = 1<br>Questão 5 = 1<br>Questão 6 = 1<br>Questão 7 = 1<br>Questão 8 = 2 | **16 – 22 pontos:** possui conhecimento suficiente para utilização do *framework*.<br>**8 –15 pontos:** possui pouco conhecimento para utilização do *framework*.<br>**0 – 7 pontos:** não possui conhecimento suficiente no domínio e na utilização do *framework* |

**Atividade: Observar o domínio do projeto de software em relação ao do framework (obrigatória)**

**Papel:** Engenheiro de *software*, Analista de Sistemas.

**Apoio Computacional:** Editor de texto.

**Documentos de Entrada:** Exemplos de projetos de *software* que atendam o desenvolvimento com o apoio do *framework* no mesmo domínio do projeto a ser desenvolvido.

**Práticas utilizadas:** Cliente presente e uso de metáforas.

**Documentos de Saída:** Documentos coletados com o cliente e um Resumo de cada reunião realizada com a presença do cliente.

**Passos:**

**P1.** Realizar entrevistas com o cliente para identificar o domínio do projeto de software. No final de cada entrevista, o responsável deve fazer um resumo do que foi conversado com o cliente. Esse resumo pode ser realizado utilizando o Resumo da Reunião (Quadro 2).

**D.1**.**1** As reuniões com os clientes devem ser tranqüilas, comunicativas e devem permitir que os clientes expressem suas vontades.

Quadro 2 – Resumo da Reunião

**Resumo da Reunião do dia: __/__/____**

| Responsável (s): |
|---|
| |
| **Descrição** |
| |
| |
| |
| |
| |
| |

Cliente — Responsável pelo projeto

**P2.** Coletar documentos necessários como, por exemplo: relatórios utilizados para controle, anotações realizadas em papel, formulários utilizados no preenchimento de alguma atividade, entre outros documentos para apoiar na identificação do domínio do projeto de *software*.

**P3.** Verificar se o *framework* abrange os requisitos do projeto de *software*.

**D**. Na realização da entrevista com o cliente, o engenheiro de *software* deve observar se o projeto de *software* se enquadra no domínio do *framework* disponível. Caso não seja possível identificar o domínio do projeto de *software*, isso é feito na próxima atividade.

**Atividade: Elaborar o planejamento do projeto de software (obrigatória)**

**Papel:** Analista de Sistema**.**

**Apoio Computacional:** Editor de texto, *Software* específico para planejamento de projeto.

**Documentos de Entrada:** Histórico de planejamentos de projetos de *software* similares ao projeto atual.

**Prática utilizada:** Jogo do planejamento e Detalhamento nos requisitos definidos.

**Documentos de Saída:** Lista de Requisitos e Planejamento do Projeto de *Software*.

**Passos:**

**P1.** Localizar um planejamento com características similares ao do projeto atual. Caso não exista, basear-se na experiência profissional da equipe.

**P2.** Realizar entrevistas e aplicar questionário(s) para a coleta de requisitos.

**D.2.1** Nas entrevistas com os clientes é importante que o responsável pela condução da entrevista faça um planejamento, preparando uma lista de questões direcionada ao objetivo da entrevista. Recomenda-se que o responsável comunique o entrevistado sobre a data, a duração da entrevista e sobre o assunto.

**P3.** Definir a lista de requisitos do novo projeto junto ao cliente.

**Gabarito Lista de Requisitos (Quadro 3)**

- Uso do gabarito para a descrição dos requisitos do projeto de *software*. O preenchimento do gabarito pode ser realizado pelo próprio cliente. Nesse caso, o analista de sistema deve deixar o cliente à vontade, sem definir restrições para a escrita. Caso o cliente não queira descrever os requisitos no gabarito, o engenheiro que deverá preencher.
- Para tornar as reuniões mais ágeis o analista de sistemas pode apenas fazer anotações da maneira que ele entenda o que deve ser feito para depois transcrever para o gabarito.
- Caso o cliente tenha dificuldades em transmitir as informações, é aconselhável montar um protótipo do sistema, apresentando todos os possíveis módulos em forma de menu.

Quadro 3 – Lista de Requisitos

| Cliente: | | | Data: __/__/____ |
|---|---|---|---|
| Projeto: | | | |
| Cód.: | | Prioridade: ( ) Baixa ( )Média ( ) Alta | |
| Categoria: ( ) Funcional ( ) Não Funcional | | | |
| Nome do caso de uso: | | | |
| Descrição do Caso de Uso | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| R.N (Regra de negócio): | | | |
| | | | |
| | | | |

**P4.** Permitir que o cliente priorize os requisitos mais importantes de cada ciclo de desenvolvimento.

**P5.** Elaborar o planejamento do projeto de *software*.

**Gabarito Planejamento do Projeto de *Software* (Quadro 4)**.

- Com os requisitos priorizados pelo cliente, é necessário definir os ciclos de desenvolvimento, observando se os requisitos priorizados dependem de outros requisitos que não foram priorizados no desenvolvimento do ciclo anterior.

Quadro 4 – Planejamento do Projeto de *Software*

| Projeto: | | | Ciclo: | Tempo estimado: |
|---|---|---|---|---|
| Período Previsto | Data Inicial: | | Data Final: | |
| Período Executado | Data Inicial: | | Data Final: | |
| Caso de uso: | | | | |
| Par Responsável: | | | Cód. casos de uso que depende: | |
| | | | | |

**Atividade: Confrontar as características não funcionais do framework x projeto de software (obrigatória)**

**Papel:** Analista de Sistemas**.**

**Apoio Computacional:** Editor de texto.

**Documentos de Entrada:** Documentos coletados na atividade "Observar o domínio do projeto de *software* em relação ao do framework", ou seja, documentos coletados com o cliente e um resumo de cada reunião realizada na presença do cliente.

**Documentos de Saída: Lista dos requisitos não funcionais.**

**Práticas:** -

**Passos:**

**P1.** Observar as características não funcionais do projeto de *software*.

**D.1.1**. É necessário que as características não funcionais sejam identificadas e que seja analisado o grau de prioridade. A lista de requisitos sugerida pelo processo PARFAIT/EA possui opções para marcar o requisito como funcional e não funcional no gabarito Lista de Requisitos (Quadro 3).

**P2.** Verificar se o *framework* atende aos requisitos não funcionais do projeto de *software*.

**D.2.1** Caso existam muitos requisitos não funcionais não fornecidos pelo *framework*, marcar uma reunião com a equipe de projeto e posteriormente com o cliente para avaliar a viabilidade em continuar o projeto de *software*.

## REUNIÕES FREQÜENTES – FASE CONCEPÇÃO

Nesta fase o ideal é que as reuniões sejam executadas no início e no final de cada atividade para facilitar no entendimento e especificação do projeto de *software*. Os participantes devem estar preparados para discutirem sobre o projeto e sobre suas evoluções. PARFAIT/EA determina que as reuniões tenham um tempo curto de duração e sugere algumas questões para serem discutidas:

1. O que foi realizado desde a última reunião?
2. O que deve ser no próximo ciclo do projeto?
3. Quais as dificuldades encontradas no projeto em relação ao *framework* disponível?
4. O projeto tem condições de prosseguir?
5. Dentre o(s) requisito (s) descrito(s), existem informações ambíguas, informações conflitantes ou ausentes?
6. Os requisitos não funcionais do *framework* podem ser adaptados ao projeto?

## MARCOS DE REFERÊNCIA – FASE CONCEPÇÃO

O objetivo do marco de referência nesta fase é avaliar se o *framework* disponível apresenta os requisitos funcionais e não funcionais do projeto de *software*.

1. O SGBD fornecido pelo *framework* está de acordo com o projeto do *software*?
2. As características do *framework*, como exemplo: a estrutura da *interface,* o *layout* dos relatórios, as opções de consultas e opções de acessibilidade estão de acordo com o projeto de *software?*
3. As configurações de *hardware* disponíveis na empresa são adequadas para o projeto?
4. As configurações de *software* (segurança de acesso, regras de negócio), fornecidas pelo *framework*, são adequadas para o projeto de *software*?
5. As configurações de *software* adicionais, necessárias para o funcionamento do *framework*, são adequadas para o projeto de *software*?
6. Os requisitos não funcionais do projeto são fornecidos pelo *framework?* Caso os requisitos funcionais não sejam fornecidos pelo *framework,* o que é necessário fazer?
    a) Adaptar o *framework*?
    b) Adaptar o *software* gerado pelo *framework*?
    c) Aceitar as restrições impostas pelo *framework*?

# FASE DE ELABORAÇÃO

**Atividade: Desenvolver o diagrama de casos de uso e elaborar os casos de teste (obrigatória)**

**Papel:** Analista de Sistemas.
**Apoio Computacional:** Ferramenta CASE para modelagem em UML.
**Documentos de Entrada:** Lista de requisitos priorizada pelo cliente de acordo com ciclo de desenvolvimento definido no gabarito de Planejamento do Projeto de *Software* (Quadro 4).
**Prática utilizada:** Cliente Presente.
**Documentos de Saída:** Diagrama de casos de uso, documentação dos casos de teste e documentação das classes de equivalência.

**Passos:**

**P1.** Documentar os casos de uso dos requisitos específicos de acordo com a lista de requisitos.

**D.1.1.** Elaborar somente um diagrama de casos de uso para requisitos semelhantes, ou seja, que possuem funcionalidades distintas.

**D.1.2.** Especificar detalhadamente os requisitos específicos do projeto de *software* que não são cobertos pelo domínio do *framework.*

**P2.** Elaborar casos de teste dos casos de uso priorizados pelo cliente para o ciclo corrente.

**D 2.1.** Uso da abordagem ARTe (CAGNIN, 2005a), quando o *framework* utilizado é baseado em uma linguagem de padrões.

**Para o uso da abordagem ARTe, seguir os passos:**

a) definir classes de equivalências para os requisitos de teste de consistência e de integridade comuns a todos os padrões da linguagem de padrões especificando: tipo de operação (inserção, alteração ou remoção), tipo de requisito, as classes válidas e as classes inválidas.
b) documentar os requisitos de teste de consistência e de integridade com base nas classes definidas no passo (a).
c) definir classes de equivalências para os requisitos de teste relacionado ao negócio, especificando: tipo de operação (inserção, alteração ou remoção), tipo de requisito, as classes válidas e as classes inválidas.
d) documentar os requisitos de teste de negócio e de integridade com base nas classes definidas no passo (c).
e) derivar e documentar casos de teste com base nos requisitos de teste de consistência, de integridade de dados e de negócio.
f) mapear os casos de teste comuns de integridade de dados e consistência a todos os padrões comuns da LPA.

**D 2.2.** Uso da técnica funcional com a utilização dos critérios "Análise do Valor Limite" e "Particionamento de Equivalência".

**Para uso dos critérios "Análise do Valor Limite" e "Particionamento de Equivalência", seguir os passos:**

a) elaborar para cada caso de uso, especificar os possíveis dados de entrada (classes válidas) e (classes inválidas).

b) utilizar o Gabarito da documentação das classes de equivalência do Quadro 6, para documentar as classes de equivalência e o Gabarito da documentação dos casos de testes, apresentado no Quadro 7 para a documentação dos casos de teste (CAGNIN, 2005b).

Quadro 6 - Gabarito da documentação das classes de equivalência

| Caso de uso: | | |
|---|---|---|
| **Restrições de Entrada** | **Classes Válidas** | **Classes Inválidas** |
| | | |
| | | |

Quadro 7 - Gabarito da documentação dos casos de teste

| Caso de uso | | | | |
|---|---|---|---|---|
| **Id. Caso de Uso** | **Cód. Caso de Teste** | **Id.C.Equivalência** | **Entrada Esperada** | **Saída Esperada** |
| | | | | |
| | | | | |

**P3.** Especificar os cursos normais e alternativos dos requisitos mais importantes.
**P4.** Validar com cliente cada caso de uso criado.

**Atividade: Documentar as regras de negócio do software (obrigatória)**

**Papel:** Analista de Sistemas.
**Apoio Computacional:** Editor de texto
**Documentos de Entrada:** Diagrama de classe do projeto de *software*, diagrama de casos de uso.
**Prática utilizada: -**
**Documento de Saída:** Documentação das regras de negócio.

**Passo:**

**P1.** Documentar detalhadamente cada regra de negócio.

Quadro 8 – Gabarito para documentar regras de negócio

| Projeto: | | |
|---|---|---|
| Nome R.N: | | Ciclo: |
| Caso de uso: | | |
| Cód. Caso de uso que depende: | | |
| Algoritmo da R.N: | | |
| | | |
| | | |
| | | |
| | | |

**I.** Certificar que todas as classes, atributos e métodos utilizados na documentação da regra de negócio foram atualizados no diagrama de classe do projeto de *software.*

**Atividade: Desenvolver o diagrama de classes do software (obrigatória)**

**Papel:** Analista de Sistemas.
**Apoio Computacional:** Ferramenta CASE para modelagem em UML.
**Documentos de Entrada:** Diagrama de casos de uso e linguagem de padrões utilizada para a construção do *framework*.
**Prática utilizada:** Metáfora.
**Documento de Saída:** Diagrama de classes do *software*.

**Passos:**

**P1.** Esboçar o diagrama de classes com o uso dos padrões da linguagem de padrões.
**P2.** Adicionar ao diagrama de classes os requisitos não cobertos pelos padrões da linguagem de padrões.

**D.** Utilizar um *Script* dos padrões da LPA para facilitar a construção do diagrama de classes.

**Atividade: Documentar as modificações realizadas no diagrama de classes (obrigatória)**

**Papel:** Analista de Sistemas.
**Apoio Computacional:** Ferramenta CASE para modelagem em UML.
**Documento de Entrada:** Diagrama de Classes do *software.*
**Prática utilizada: -**
**Documentos de Saída:** Gabarito da documentação das adaptações no diagrama de classe não cobertas pela linguagem de padrões e Gabarito da documentação dos requisitos da linguagem de padrões que não se adequam ao *software*.

**Passos:**

**P1.** Documentar os requisitos funcionais do diagrama de classes não cobertos pela linguagem de padrões, utilizando o gabarito apresentado no Quadro 9.

Quadro 9 – Gabarito da documentação das adaptações no diagrama de classes não cobertas pela linguagem de padrões**.**

| Elemento | Tipo de alteração no diagrama de classes: (I)nserção, (R)emoção ou (M)odificação | Descrição da alteração no diagrama de classe |
|---|---|---|
| Cardinalidade ou classe ou relacionamento alterado | I ou R ou M | Descrição da alteração ocorrida no diagrama de classe |

**P2.** Documentar os requisitos da linguagem de padrões que não se adequam completamente aos do *software*, utilizando o Gabarito apresentado no Quadro 10.

Quadro 10 – Gabarito da documentação dos requisitos da linguagem de padrões que não se adequam ao *software*.

| Padrão/Variante | Descrição do requisito fornecido pelo padrão | Descrição do requisito não coberto pelo padrão |
|---|---|---|
| Nome do padrão e da variante utilizada no diagrama de classes, que não se adequam completamente ao software | Descrição sucinta do requisito que o padrão representa | Descrição sucinta do requisito do software não coberto pelo padrão utilizado |

**I1.** Certificar que todas as modificações necessárias no projeto de *software* que não atendem aos padrões da LPA e ao projeto de *software* foram documentadas.

**REUNIÕES FREQUENTES – FASE DE ELABORAÇÃO**

As reuniões nesta fase têm como objetivo discutir a documentação que está sendo produzida, as prioridades definidas pelo cliente, os casos de testes produzidos e as regras de negócios documentadas. Algumas questões devem ser discutidas, como:

1. Com a documentação produzida, os participantes do projeto obtiveram uma visão geral do produto?
2. Com a documentação produzida até o momento é possível gerar uma versão do *software*?
3. O que ainda é necessário acrescentar ao projeto de *software*?

**MARCO DE REFERÊNCIA – FASE DE ELABORAÇÃO**

No final desta fase algumas questões devem ser consideradas com o objetivo de avaliar o ciclo corrente.

1. Todas as regras de negócio do projeto de *software* foram identificadas?
2. O *framework* atende aos requisitos identificados até o momento?
3. Foi elaborada estimativa de prazo/custo para o ciclo de desenvolvimento corrente?
4. É viável utilizar o *framework* para o desenvolvimento do *software*, após a elaboração e refinamento do diagrama de classes no ciclo corrente?

# FASE DE CONSTRUÇÃO

**Atividade: Criar o software (obrigatória)**

**Papel:** Programador.
**Apoio Computacional:** *Framework*, ferramenta de apoio para instanciar o *software*.
**Documentos de Entrada:** Documentação do *framework*, da LPA e das ferramentas de instanciação do *framework*.
**Práticas utilizadas:** Versões freqüentes, integração contínua, propriedade coletiva do código.
**Documento de Saída:** *Software*.

**Passos**:

**P1.** Instanciar o *framework* para gerar o *software*.

**I1.** Verificar se todos os padrões da LPA utilizados para criar o diagrama de classes foram utilizados na instanciação do *framework* e se os atributos adicionais aos padrões representados no diagrama de classes, foram considerados na instanciação do *framework*.

**Atividade: Executar os casos de teste no software (obrigatória)**

**Papel:** Testador, Cliente.
**Apoio Computacional:** Ferramenta de teste.
**Prática utilizada:** Testes constantes.
**Documentos de Entrada:** Documentação dos casos de teste, *software*.
**Documento de Saída:** Relatório resumo de teste.

**Passos:**

**P1.** Executar no *software* os casos de teste documentados.

**P2.** Comparar o resultado obtido do *software* com o resultado esperado de cada caso de teste.

**D.1.1** Caso o resultado dos casos de teste seja diferente do esperado, analisar o *software* e elaborar novos casos de testes**.**

**P3.** Validar juntamente com o cliente as regras de negócio e os requisitos identificados.

**D.1.2** Incentivar o cliente para que faça testes de aceitação na versão corrente do *software*.

**I.** Garantir que todos os casos de testes documentados sejam executados.

**Atividade: Adaptar o software (obrigatória)**

**Papel:** Programador.
**Apoio computacional:** Framework, ferramenta de controle de versão dos *softwares* gerado pelo framework e ferramenta para instanciação do framework.
**Documentos de Entrada:** Documentação das regras de negócios, *software*.
**Prática utilizada:** Projetar (adaptar) com simplicidade, propriedade coletiva do código.
**Documento de Saída:** *Software* modificado.

**Passos**:

**P1.** Adaptar o *software* às regras do negócio identificadas.

**D.1.1 Adaptações no *software* em relação às regras de negócio.**

a) Caso a regra de negócio exista na LP, pesquisar na hierarquia de classes do framework os métodos que implementam a regra de negócio.

b) Caso a regra de negócio do projeto não é coberta pela LP, implementar a regra de negócio no *software* com a criação de novas classes e/ou novos métodos.

**P2.** Adaptar o *software* aos requisitos identificados

**D2.1 Adaptações no *software* em relação aos requisitos:**

a) Estudar as superclasses (classes do *framework*).

b) Caso o requisito não é desejado, é necessário desabilitá-lo ou torná-lo invisível.

c) Para os requisitos não existentes na LP é necessário implementá-los no *software* por meio da criação de métodos em classes adequadas.

**D1.** Após cada adaptação no *software* é necessário atualizar o "Diagrama de classes do sistema" adicionando a documentação de novos métodos criados.

**D2.** Com o uso do framework GREN, é necessário utilizar a ferramenta GREN-WizardVersionControl (CAGNIN, 2005a), para que as adaptações realizadas no código fonte não sejam perdidas nas próximas instanciações do *framework.*

**P3.** Executar o *software* junto ao cliente.
**P4.** Adaptar o *software* de acordo com as mudanças exigidas pelo cliente.

**I.** Observar se as regras de negócio e se os requisitos foram implementados ou modificados corretamente.

## REUNIÕES FREQUENTES – FASE DE CONSTRUÇÃO

Nesta fase discuta com a equipe de projeto e o cliente as versões liberadas e as modificações realizadas no projeto de *software.* Faça sugestões em pontos fracos identificados no *software.*

1. A última versão do *software* gerada pelo *framework* atendeu as expectativas dos requisitos coletadas até o final desta fase?
2. O que pode ser melhorado no *software*?
3. As adaptações realizadas foram satisfatórias?
4. Os casos de testes aplicados foram bem sucedidos?
5. A versão do *software* liberada possui os requisitos definidos para esse ciclo?
6. Qual o próximo passo a ser realizado no projeto de *software?*

## MARCOS DE REFERÊNCIA – FASE DE CONSTRUÇÃO

Avaliar algumas características do *software* de acordo com o *checklist* a seguir:

1. **Verificar a *interface* do *software*, avaliando:**
   - o desempenho das *interfaces* e relatórios.
   - a combinação de cores, tamanho, alinhamento das caixas de texto e apresentação das informações.

- a facilidade de operar o *software,* oferecendo opções para o uso do mouse e, quando possível, do teclado.
- o *software* gerado é amigável ao cliente?

2. **Com relação a atividade de teste**:
   - Continuar os testes mesmo sem a presença do cliente, verificando o que funciona e o que está errado.
   - Ajustar ou adaptar algumas características do *software* para consertar o que está errado.
   - Indicar ao cliente o que precisa ser modificado, oferecendo opções de mudanças.

## FASE DE TRANSIÇÃO

**Atividade: Testar o software (obrigatória)**

**Papel:** Testador
**Apoio computacional: -**.
**Documentos de Entrada:** *Software*.
**Prática utilizada:** Testes constantes.
**Documento de Saída:** *Software* testado.

**Passos:**

**P1.** Executar teste de aceitação no *software*.
**D.1.2.** Para as situações em que o *software* não apresentou o resultado esperado, realizar anotações utilizando o gabarito Resumo da Reunião (Quadro 2).

**I.** Caso seja o cliente que teste o *software* ou que o mesmo esteja presente, permitir que ele sinta-se à vontade para identificar novos requisitos ou modificar os requisitos existentes.

**Atividade: Treinar os usuários finais (obrigatória)**

**Papel:** Analista de Sistemas, Suporte Técnico.
**Apoio computacional**: Ultima versão do *software*.
**Documentos de Entrada:** *Software*
**Prática utilizada:** -
**Documento de Saída: -.**

**Passos:**

**P1.** Demonstrar o *software* utilizando os casos de uso documentados.
**P2.** Criar situações reais para o treinamento.

**D1.** Treinar os usuários do *software* conforme a(s) atividade(s) desempenhada(s) na empresa.

**D2.** Certificar que todos os usuários estejam utilizando o *software* adequadamente, não introduzindo dados errôneos na base de dados.

**D3.** Exemplificar situações utilizando quadro branco e permitir a interação com os usuários.

**Atividade: Elaborar o manual do usuário do software (obrigatória)**

**Papel:** Analista de Sistemas.
**Apoio computacional**: Editor de texto.
**Documento de Entrada:** *Software.*
**Prática utilizada:** -
**Documento de Saída:** Manual do usuário.

**Passos:**

**P1.** Documentar os padrões de interface utilizados
**P2.** Documentar a localização dos requisitos por menu.
**P3.** Documentar as restrições das entradas de cada tela de interface.

**I.** Verificar se todos os requisitos e operações foram documentados.

## REUNIÕES FREQÜENTES – FASE DE TRANSIÇÃO

Na reunião desta fase, algumas questões devem ser discutidas como:

1. Existe algum requisito não apresentado no *software* que deveria ser adicionado?
2. O *software* produzido atendeu as necessidades do cliente?
3. A versão entregue ao cliente possui os requisitos da versão anterior?
4. Os usuários do *software* obtiveram treinamento adequado?
5. Foram definidos um conjunto de critérios de aceitação do *software* junto ao cliente, para obter um grau de qualidade?

## MARCO DE REFERÊNCIA – FASE DE TRANSIÇÃO

O objetivo desta fase é verificar se todos os requisitos foram atendidos para a implantação do *software* na empresa cliente. Considere alguns itens a ser conferidos em cada atividade, de acordo com o *checklist* a seguir.

Para cada novo item identificado na atividade, atualizar o *checklist* para que esses itens possam ser reutilizados em outros projetos.

| | |
|---|---|
| | |
| **Verificar os itens na atividade "Testar o software" em relação a:** | |
| ( ) | Os requisitos estão todos implementados corretamente |
| ( ) | Os resultados estão corretos ou conforme o esperado |
| ( ) | O *software* apresentou falhas durante sua execução |
| | |
| **Verificar os itens a seguir na atividade "Treinar os usuários finais" em relação a:** | |
| ( ) | Todas as operações dos módulos de cadastros foram explicadas |
| ( ) | Os relatórios referentes aos módulos de cadastros foram explicados |
| ( ) | As regras de negócios foram demonstradas no *software* |
| ( ) | A utilização do mouse e teclado nos módulos do *software* |
| | |
| | |
| **Verificar os itens na atividade "Elaborar o manual do usuário" em relação a:** | |
| ( ) | Organização do documento para facilitar o entendimento |
| ( ) | Facilidade com a identificação das operações |
| ( ) | Uso de recursos para destacar as informações relevantes |
| (..) | Exemplos para ajudar no aprendizado do assunto |