Cristina Bona

AVALIAÇÃO DE PROCESSOS DE SOFTWARE: UM ESTUDO DE CASO EM XP E ICONIX

Dissertação apresentada ao
Programa de Pós-Graduação em
Engenharia de Produção da
Universidade Federal de Santa Catarina
como requisito parcial para obtenção
do grau de Mestre em
Engenharia de Produção

Orientador: Prof. Marcello Thiry Comicholi da Costa, Dr.

Florianópolis
2002

Cristina Bona

AVALIAÇÃO DE PROCESSOS DE SOFTWARE: UM ESTUDO DE CASO EM XP E ICONIX

Esta dissertação foi julgada e aprovada para a obtenção do grau de **Mestre em Engenharia de Produção** no **Programa de Pós-Graduação em Engenharia de Produção** da Universidade Federal de Santa Catarina

Florianópolis, 25 de outubro de 2002.

Prof. Edson Pacheco Paladini, Dr.
Coordenador do Programa

BANCA EXAMINADORA

Prof. Marcello Thiry C. da Costa, Dr.
Universidade do Vale do Itajaí
Orientador

Prof. Alejandro Martins Rodriguez, Dr.
Universidade Federal de Santa Catarina

Prof. Vinícius Medina Kern, Dr.
Universidade Federal de Santa Catarina

Aos meus pais, Fedele e Luiza
pelo carinho constante.
Aos meus irmãos Ione, Marcos e Daiane.

Agradecimentos

Agradeço primeiramente àquela luz maior,
por me conduzir e me dar forças para alcançar meus objetivos.
À Universidade Federal de Santa Catarina.
Ao orientador Prof. Marcello Thiry, pela dedicação incondicional,
apoio, paciência e também por ter acreditado em mim.
Aos meus pais Fedele e Luiza, pelo carinho, educação e lição de vida.
Vocês me ensinaram a confiar e sonhar.
Ao meu marido Marcello, pelo seu amor, carinho, compreensão
e constante ajuda nesta caminhada.
À Decka, pela amizade e momentos de descontração.
À CEFET/SC e à Secretaria Municipal da Saúde,
pela disponibilização das informações.
À todos que direta ou indiretamente
contribuíram para a realização
desta pesquisa.

“Quando os ventos da mudança sopram,
alguns constroem abrigos,
outros, moinhos”.
Clauss Möller

Sumário

Lista de Figuras

Lista de Tabelas

Resumo

BONA, Cristina. **Avaliação de Processos de Software: Um estudo de caso em XP e ICONIX**. Florianópolis, 2002. 122f. Dissertação (Mestrado em Engenharia de Produção) – Programa de Pós-Graduação em Engenharia de produção, UFSC, 2002.

O presente trabalho busca fornecer, através de exemplos práticos e representações gráficas, uma estrutura capaz de orientar as organizações na escolha da metodologia mais apropriada para seus projetos de software. Um dos principais esforços dos pesquisadores envolvidos com a Engenharia de Software tem sido apresentar e abstrair modelos que descrevem processos de software. Estes modelos permitem que se compreenda o processo de desenvolvimento dentro de um paradigma conhecido. A existência de um modelo é apontada como um dos primeiros passos em direção ao gerenciamento e à melhoria do processo de software. Na última década, um novo segmento da comunidade de Engenharia de Software vem defendendo processos simplificados, também conhecidos como "processos ágeis", focados nas pessoas que compõem o processo e, principalmente, no programador. O trabalho apresenta a aplicação e avaliação de dois processos. O primeiro a ser discutido é o *Extreme Programming* (XP), o qual está entre os "processos ágeis" e que vai contra uma série de premissas adotadas por métodos mais tradicionais de desenvolvimento. O segundo processo é o ICONIX**,** o qual é definido como sendo um processo prático que utiliza a notação UML. Como produto final, além da aplicação de cada processo, são discutidos os seus pontos positivos e negativos. Um estudo comparativo entre os modelos é também elaborado, onde são examinados simultaneamente os recursos de ambos os processos, para desta forma, determinar quais características devem ser inerentes a um processo de software que garanta a produtividade e qualidade.

Palavras-chave: Processo de Software, Extreme Programming, ICONIX, UML.

Abstract

BONA, Cristina. **Avaliação de Processos de Software: Um estudo de caso em XP e ICONIX**. Florianópolis, 2002. 122f. Dissertação (Mestrado em Engenharia de Produção) – Programa de Pós-Graduação em Engenharia de produção, UFSC, 2002.

This work intends to offer, by means of practical examples and graphic notations, a document guide to lead the choice in organizations of the proper methodology for its software projects. One of the main researchers' efforts involved with the Software Engineering has been to present and to abstract models that describe software processes. These models allow the understanding about the development process inside a well-known paradigm. The existence of a model is pointed as one of the first steps in direction to the management and to the improvement of the software process. In the last few years, a new faction of the Software Engineering community is defending the necessity of simpler processes, also known as “agile processes”, which focus on the people that compose the process and, mostly, on the programmer. This work introduces the application and evaluation of two processes. The first one to be discussed is the Extreme Programming (XP), one of the more known among the agile processes. XP goes against a set of premises adopted by more traditional development methods. The second process to be presented and to be discussed is ICONIX, which is defined as a practical process that uses the UML notation. After the presentation of the two processes, it is presented a practical application of each process where their positive and negative aspects are pointed out. A comparative study between the two models is also elaborated, where are examined the resources from both processes. In this way, the work concludes how to determine which aspects should be inherent to a software process that guarantees productivity and quality.

Keywords: Software Process, Extreme Programming, ICONIX, UML.

1 INTRODUÇÃO

1.1 Contextualização

O impacto e a rápida evolução ao longo dos últimos 40 anos das tecnologias relacionadas com os sistemas de informação têm colocado sucessivos desafios às empresas. A dependência e demanda crescentes da sociedade em relação à Informática e, em particular, a software, tem ressaltado uma série de problemas relacionados ao **processo** de desenvolvimento de software: alto custo, alta complexidade, dificuldade de manutenção, e uma disparidade entre as necessidades dos usuários e o produto desenvolvido.

Empresas de software em todo o mundo empregam perto de 7 milhões de técnicos e geram anualmente uma receita de mais de 600 bilhões de dólares, com taxa de crescimento anual de mais de 25% nos últimos três anos. A indústria de software é vista atualmente como um dos segmentos mais promissores, com um enorme potencial futuro (CORDEIRO, 2000). Desta forma, desenvolver projetos de software eficientes é de fundamental importância para a indústria de software como um todo.

A necessidade de se encontrar métodos que pudessem ajudar na evolução do processo de construção de software, culminaram com o desenvolvimento da Engenharia de Software. Essa nova abordagem ao processo de construção de software trouxe consigo métodos e técnicas que ajudaram na administração da complexidade inerente do software. Conforme Silva & Videira (2001), a comunidade de Engenharia de software, desde os finais da década de 60, estuda e implementa práticas de desenvolvimento de software bem organizadas e documentadas.

Os processos usados para desenvolver um projeto de software têm a maior importância na qualidade do software produzido e na produtividade alcançada pelo projeto. No entanto, não existe um modelo uniforme que possa descrever com precisão o que de fato acontece durante todas as fases da produção de um software; os processos implementados são muito variados, e as necessidades de cada organização diferem substancialmente (SILVA & VIDEIRA, 2001).

Além disso, na última década, um segmento crescente da comunidade de Engenharia de Software vem defendendo a existência de problemas fundamentais da aplicação sistemática e institucionalizada de processos de software convencionais (HIGHSMITH, 2002), (BECK, 2000), (FOWLER, 2001) e (SCHWABER, 1997). Estes proponentes advogam processos simplificados, focados nas pessoas que compõem o processo, e principalmente no programador.

O processo ***Extreme Programming*** (XP) está entre os denominados "processos ágeis" (HIGHSMITH, 2002), que vai contra uma série de premissas adotadas por métodos mais tradicionais de desenvolvimento. XP consiste numa série de práticas e regras que permitem aos programadores desenvolver software de uma forma dinâmica e ágil, com mínimo de documentação.

Neste contexto, o processo **ICONIX** define-se como um "processo" de desenvolvimento de software prático. O ICONIX está entre a complexidade e abrangência do RUP (*Rational Unified Processes*) e a simplicidade e o pragmatismo do XP, mas sem eliminar as tarefas de análise e de desenho que o XP não contempla.

1.2 Objetivos do Trabalho

1.2.1 Objetivo geral

O objetivo principal deste trabalho é realizar um estudo comparativo entre os modelos de desenvolvimento de sistemas o XP e o ICONIX, buscando fornecer uma estrutura capaz de orientar as organizações na escolha da metodologia mais apropriada para seus projetos de software.

1.2.2 Objetivos específicos

- Estudar os principais processos de desenvolvimento de software;
- Estudar, através da revisão bibliográfica, o modelo *Extreme Programming* ;
- Estudar, através da revisão bibliográfica, o modelo ICONIX;
- Especificar a estrutura para aplicação dos modelos;

- Descrever os pontos positivos e negativos dos dois processos avaliados;
- Apresentar um esquema comparativo dos modelos;
- Validar a aplicação;
- Análise dos resultados.

1.3 Justificativa do Trabalho

A Engenharia de Software foi sempre muito mais aplicada no ambiente acadêmico que no mercado de desenvolvimento. Entretanto, a competitividade e a necessidade crescente por um desenvolvimento de qualidade em prazos apertados tem feito com que as empresas mudem sua forma de trabalhar e procurem soluções para a organização do processo de desenvolvimento de software. No Brasil, cerca de 87% das empresas de desenvolvimento de software são de pequeno e médio porte (SEMEGHINI, 2001). Sendo assim, existem limitações de recursos para serem aplicados em treinamento e consultorias voltadas à implantação de qualidade de software e de um processo que garanta resultados adequados.

Atualmente, a prática tem mostrado resultados que estimulam a revisão dos procedimentos tradicionais de engenharia de requisitos. O trabalho em grupo é outra importante ferramenta para aumentar o conhecimento sobre um determinado assunto. O processo de discussão faz com que novas idéias sejam consideradas e oferece diversas visões sobre o mesmo problema. Estas novas tendências de trabalho são interessantes na área de desenvolvimento de software, uma vez que a característica de resolver problemas colaborativamente (em grupo) é uma realidade (BECK, 2001), (COCKBURN, 2000).

A diversidade de processos de desenvolvimento de software (FOWLER, 2000), (SUTHERLAND, 2000), (EVANS, 2001), (ROSENBERG & SCOTT, 1999) também faz com que seja importante um bom entendimento sobre como desenvolver software de qualidade e encontrar qual o processo mais adequado para o tipo de software que será desenvolvido.

1.4 Estrutura do Trabalho

O trabalho está estruturado da seguinte forma:

Capítulo 1: Neste capítulo é apresentada a contextualização da pesquisa deste trabalho, enfatizando os objetivos propostos e a justificativa.

Capítulo 2: Aborda uma revisão sobre processo de software. Apresentando uma separação em fases do processo, atividades do processo, modelos de clico de vida e metodologias ágeis.

Capítulo 3: Este capítulo descreve o modelo de processo XP. Enfatizando os quatro valores e apresentando detalhes das doze práticas do processo. Apresenta também o clico de vida do XP e, destaca alguns pontos positivos e negativos.

Capítulo 4: Este capítulo apresenta o modelo de processo ICONIX. Mostra as principais tarefas e marcos do processo, destacando os alertas do mesmo. Aborda também os modelos utilizados exemplificando-os.

Capítulo 5: Identifica os procedimentos metodológicos utilizados para a realização do trabalho. Apresenta o ambiente da aplicação e como foi realizada a coleta das informações através da aplicação do XP e da aplicação do ICONIX.

Capítulo 6: Este capítulo apresenta a análise dos resultados. Destaca os pontos positivos e negativos do XP e do ICONIX. Além de exibir uma tabela comparativa entre ambos os processos. Adicionalmente, é realizado um questionário e o resultado representado graficamente.

Capítulo 7: Apresenta as conclusões alcançadas no desenvolvimento desta pesquisa, e as recomendações e trabalhos futuros.

2 PROCESSO DE SOFTWARE

2.1 Introdução

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

O Processo de Software é definido por Sommerville (1995) como:

“O processo é um conjunto de atividades e resultados associados que produzem um produto de software”.

Pressman (1997), oferece a seguinte definição:

“...definimos um processo de software como um *framework* para as tarefas que são necessárias para a construção de software de alta qualidade”.

De acordo com Thiry (2001), o processo de software tem a seguinte definição:

“O processo define *quem* irá fazer *o que* e *como* será atingido o objetivo. O objetivo é construir um software ou melhorar um existente".

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

- O processo reúne um conjunto de atividades. Como existem atividades que englobam outras atividades, pode-se usar o termo fase para descrever atividades de nível mais alto;
- A definição contempla na prática: procedimentos e métodos, ferramentas e equipamentos, comunicação e pessoas;
- O processo tem como objetivo desenvolver um produto de software. Pressman (1997) restringe o termo a processos que geram “produtos de alta

qualidade", mas se esta restrição for ignorada, pode-se aplicar o termo a qualquer conjunto de atividades que é aplicada com o objetivo de desenvolver software;

- O processo direciona as tarefas individuais e do time como um todo.

O nível de detalhamento de cada processo depende da equipe envolvida no desenvolvimento do software. Rezende (2002) usa a metáfora da receita de bolo para representar o processo. Quem segue uma receita tem a liberdade de adicionar ou suprir partes, assim também pode ocorrer com o processo, estabelecendo um dinamismo na execução.

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

2.2 Fases do Processo de Software

Pela definição, podemos entender o que é o processo; no entanto, torna-se importante conhecer quais as fases que o compõe.

Em meados dos anos 70, Schwartz (1975) já apontava como fases principais do processo de produção de um sistema de software:

- **Especificação de Requisitos**: tradução da necessidade ou requisito operacional para uma descrição da funcionalidade a ser executada;
- **Projeto de Sistema**: tradução destes requisitos em uma descrição de todos os componentes necessários para codificar o sistema;
- **Programação** (codificação): produção do código que controla o sistema e realiza a computação e lógica envolvida;
- **Verificação e Integração** (c*heckout*): verificação da satisfação dos requisitos iniciais pelo produto produzido.

A definição moderna oferecida por Sommerville (1995) é similar; define as fases como Especificação, Desenvolvimento, Validação e Evolução. Este último ponto não é descrito diretamente por Schwartz (1975):

- **Evolução** (manutenção): alteração do software para atender a novas necessidades do usuário.

É interessante observar que é destacada a questão da longevidade do software no item Evolução. Neste sentido, pode-se perceber que a definição de Schwartz (1975) omite uma particularidade importante: o software continua sendo desenvolvido mesmo depois de entregue. Pressman (1997) oferece uma visão compatível com esta, ainda que simplificada: as fases descritas são definição, desenvolvimento e manutenção.

Neste contexto, Leach (2000) destaca o processo de desenvolvimento de software usado na Microsoft. Onde a maior parte dos novos softwares desenvolvidos tem três fases: o planejamento, o desenvolvimento e a estabilização. A última fase compreende os testes realizados dentro da Microsoft e os testes realizados por usuários externos a partir de uma versão beta. A estabilização acontece quando as mudanças são visíveis e o número de erros está bem reduzido, em um nível aceitável. Este nível é baseado no conhecimento do time em saber se o software é seguro. A evolução está presente nas constantes melhorias que a empresa realiza em seus produtos.

É importante ressaltar que não existe uma seqüência nem um número obrigatório de fases. Rezende (2002) ressalta que mesmo considerando a ISO 9000-3, não existe um padrão universalmente aceito, podendo esta divisão ter mais ou menos fases.

Além da questão de seqüencialidade, existem autores que defendem que o processo de software é muito mais iterativo e cíclico do que a idéia de fases simples pode sugerir. Em particular, Microsoft® (1996), Leach (2000), Beck (2000) e o próprio *Rational Unified Process* (RUP) sugerem processos onde existem ciclos contínuos e repetidos, onde alguma forma de produto é desenvolvida a cada ciclo. A extensão de cada ciclo, no entanto, não é um consenso.

2.3 Atividades do Processo de Software

Para cada fase do processo de desenvolvimento de software existe uma série de atividades que são executadas. Segundo Pressman (1997) as atividades constituem um conjunto mínimo para se obter um produto de software. Observando as fases individuais e suas atividades associadas, combinando classificações de Schwartz (1975), Pressman (1997) e Sommerville (1995), é possível identificar as seguintes atividades:

Especificação

- Engenharia de Sistema: estabelecimento de uma solução geral para o problema, envolvendo questões de tecnologia e equipamento;
- Análise de Requisitos: levantamento das necessidades do software a ser implementado. A Análise tem como objetivo produzir uma especificação de requisitos, que convencionalmente é um documento;
- Especificação de Sistema: descrição funcional do sistema. Pode incluir um plano de testes para verificar adequação.

Projeto

- Projeto Arquitetural: onde é desenvolvido um modelo conceitual para o sistema, composto de módulos mais ou menos independentes;
- Projeto de Interface: onde cada módulo tem sua interface de comunicação estudada e definida. Pode resultar em um protótipo;
- Projeto Detalhado: onde os módulos em si são definidos, e possivelmente traduzidos para pseudocódigo[1].

Implementação

- Codificação: a implementação em si do sistema em uma linguagem de computador, de programação.

[1] Instruções do computador escritas pelo programador em linguagem simbólica.

Validação

- Teste de Unidade e Módulo: a realização de testes para verificar a presença de erros e comportamento adequado relacionado às funções e módulos básicos do sistema;
- Integração: a reunião dos diferentes módulos em um produto de software homogêneo, e a verificação da interação entre estes quando operando em conjunto.

Evolução e Manutenção

- Nesta fase, o software em geral entra em um ciclo iterativo que abrange todas as fases anteriores.

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

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

2.4 Modelos de Ciclo de Vida

O ciclo de vida é a base para a gerência e controle o projeto, é o coração do processo, pois define as fases e atividades a serem seguidas. Entretanto, um ciclo de vida não define notação nem se preocupa em estabelecer claramente os artefatos a serem produzidos.

Thiry (2001) considera que “...modelo é uma coleção de artefatos, cada um expressando uma visão do sistema. ....artefato é qualquer informação produzida por um participante (modelo, documentos). Um artefato pode ter ainda, suas versões controladas”.

Existem alguns modelos teóricos desenvolvidos que buscam descrever a forma com que as fases seguem e interagem. Nesta seção estão descritos alguns dos modelos de clico de vida mais conhecidos, como: modelo cascata, modelo espiral, modelo protótipo e modelo iterativo e incremental (ICONIX, RUP). Existe alguma flexibilidade no que diz respeito à definição do termo “modelo”, neste trabalho são considerados, dentre os modelos descritos na literatura, os que têm um caráter estratégico, e não específico. Em outras palavras, um modelo é uma filosofia do andamento das fases – ciclo de vida, e não uma descrição de como cada atividade deve ser executada.

A seção 2.5, por sua vez, descreve algumas metodologias, que são formas práticas de organizar o processo de desenvolvimento. Uma metodologia traz conceitos bastante específicos em relação ao desenvolvimento, como exemplifica Beck (2000) em sua descrição da metodologia *Extreme Programming* (XP): programação em pares, ciclos pequenos e equipes com até 10 programadores.

2.4.1 Modelo cascata

O modelo mais comum de processo de desenvolvimento de software é chamado de modelo cascata (LEACH, 2000). Este modelo foi idealizado em 1970 por Royce (THIRY, 2001), e tem como característica principal à seqüencialidade das atividades: sugere um tratamento ordenado e sistemático ao desenvolvimento do software. Cada fase transcorre completamente e seus produtos são vistos como entrada para a nova fase. O software é desenvolvido em um longo processo e entregue ao final deste. O modelo sugere laços de *feedback*, que permitem realimentar fases anteriores do processo, mas em geral o modelo cascata é considerado um modelo linear (LEACH, 2000). A figura 1 fornece uma descrição visual do modelo.

A maior relevância deste modelo está na reunião de diversos conceitos que são adotados até hoje. Além disso, ele estabeleceu a necessidade de definir uma sistemática para o desenvolvimento de software. Entretanto, é possível observar que a rigidez da sistemática proposta inicialmente resulta na inadequação deste modelo para processos reais. Geralmente, há muito intercâmbio de informações entre as fases, e este modelo não permite flexibilidade de retorno nas sequências, tornando o desenvolvimento do software altamente engessado (REZENDE, 2002). A

manutenção é ainda considerada uma fase por si só, enquanto a forma moderna de desenvolvimento estabelece que a manutenção deve ser feita através da reaplicação do processo sob as novas demandas. Além disso, o modelo cascata não leva em consideração questões modernas importantes ao desenvolvimento: prototipação, aquisição de software e alterações constantes nos requisitos, por exemplo às apresentadas por Beck (2000) e Cockburn (2000).

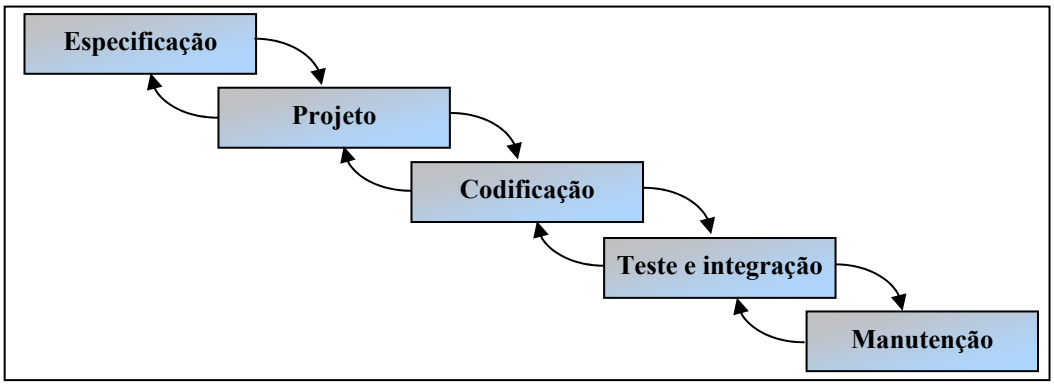


Figura 1: Diagrama simplificado do modelo cascata

Apesar das suas limitações, o modelo em cascata foi à base para o surgimento de diferentes modelos de ciclo de vida.

2.4.2 Modelo espiral

O modelo espiral é uma forma elaborada do modelo cascata, introduzido por Barry Boehm em um artigo publicado na *IEEE Computer* em Maio de 1988. Boehm sugeriu um modelo evolucionário para o desenvolvimento de software, baseado em uma seqüência de fases que culminam em versões incrementais do software (CANTOR, 1998). Esta característica incremental é confirmada no conceito do modelo de prototipação rápida (LEACH, 2000), e em metodologias de processo mais modernas, como RUP, ICONIX e *Extreme Programming* descrita por Beck (2000).

O modelo espiral define quatro importantes atividades representadas em quatro quadrantes conforme representado na figura 2:

- Determinação dos objetivos: definição que será desenvolvido, restrições impostas à aplicação, tais como desempenho, funcionalidade, capacidade de acomodar mudanças, meios alternativos de implementação;

- Análise de risco: análise das alternativas e identificação/resolução dos riscos. Uma vez avaliados os riscos, pode-se construir protótipos para verificar se estes são realmente robustos para servir de base para a evolução futura do sistema;
- Desenvolvimento: detalhe do projeto, codificação, integração;
- Planejamento e próxima iteração: avaliação dos resultados pelo cliente, entrega ao cliente.

Segundo Schneider (1999), um aspecto bastante intrigante do modelo espiral torna-se aparente quando passa a se considerar a sua dimensão radial. Cada iteração ao redor da espiral (começando pelo centro e crescendo para fora), indica que uma meta do projeto foi executada. Durante o primeiro ciclo ao redor de uma espiral, objetivos, alternativas e restrições são definidas, riscos são identificados e analisados. Se a análise de risco indicar que há dúvidas nos requisitos, a prototipação pode ser usada no quadrante de desenvolvimento para assegurar o projeto. Entretanto, em muitos casos, o fluxo ao redor da espiral continua, onde cada rotação indica uma evolução significativa do projeto, até a sua completa conclusão. Cada circuito ao redor da espiral, envolve desenvolvimento que pode ser alcançado com o modelo cascata ou prototipação.

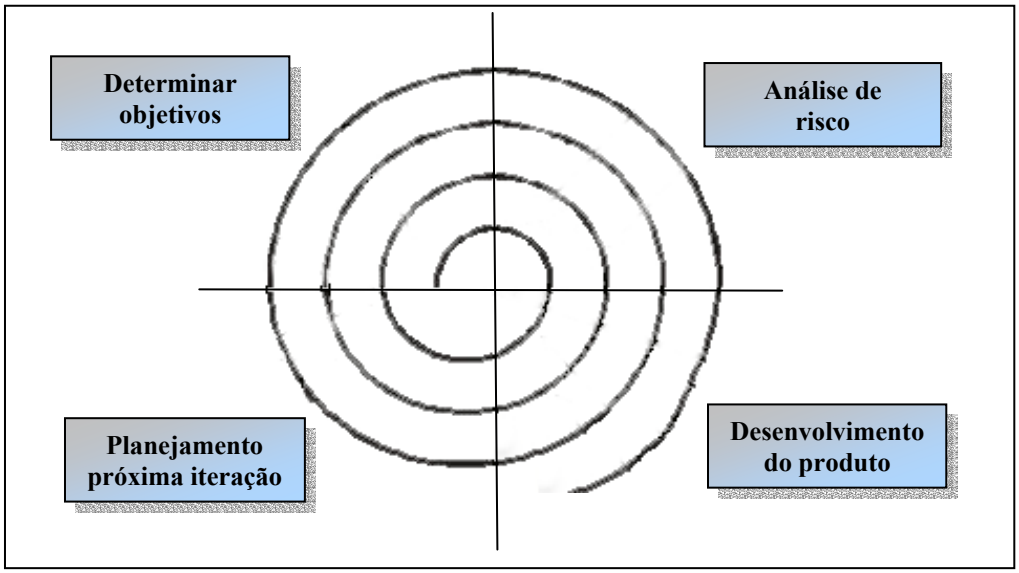


Figura 2: Diagrama simplificado do modelo espiral

Geralmente, modelos incrementais têm o objetivo de lidar melhor com um conjunto de requisitos incertos ou sujeitos a alterações. O modelo espiral parece mais bem adequado a projetos reais que o modelo cascata.

Entretanto, Cantor (1998) faz algumas recomendações sobre a forma que o modelo é apresentado, que pode ser muito detalhado para atender as necessidades de um projeto. O modelo espiral assume a existência de alguma seqüência entre as fases: não há suporte para fases que ocorrem simultaneamente, ou que necessitam de intercomunicação contínua para operarem.

2.4.3 Modelo de prototipação

Segundo Leach (2000), o modelo de prototipação é iterativo e requer a criação de um ou mais protótipos como parte de um processo de desenvolvimento de software.

O modelo de prototipação se baseia na utilização de um protótipo do sistema real, para auxiliar na determinação de requisitos. Um protótipo deve ter custo reduzido e rápida obtenção, para que possa ser avaliado. Para isto, uma parte do sistema é desenvolvida com o mínimo de investimento mas sem perder as características básicas, para ser analisada juntamente com o usuário (ver figura 3).

De acordo com Pascoal (2001), a prototipação é um processo que habilita o desenvolvedor a criar um modelo do software que deve ser construído. O modelo pode ter diferentes formas, como:

- uma no papel ou em um modelo baseado em computador, no qual a interação homem-máquina é desenhada para permitir ao usuário entender como tal interação irá ocorrer;
- uma versão funcional que contém apenas um subconjunto das funcionalidades requeridas no software desejado;
- um protótipo em execução, o qual executa parte ou todas a funções desejadas, mas tem outras características que serão melhoradas no desenvolvimento.

Por sua vez, o protótipo pode servir como um "primeiro sistema". Neste sentido, Pascoal (2001) destaca que alguns problemas podem surgir quando o cliente

visualiza o que parece ser uma versão do software, sem perceber que na realidade, não foram considerados aspectos de qualidade e de manutenção. Quando é informado de que o produto deve ser reconstruído, o cliente "implora" para que não mude. Outro problema, é o desenvolvedor assumir um compromisso de implementação para obter um protótipo funcional rapidamente. Então, um sistema operacional ou linguagem de programação inadequados podem ser usados simplesmente porque estão disponíveis e são conhecidos.

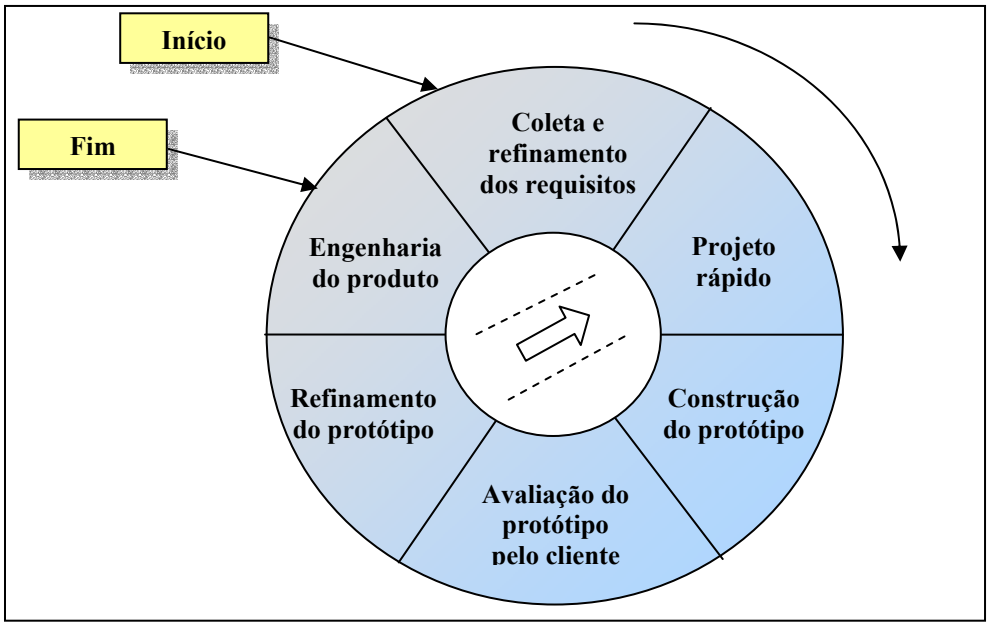


Figura 3: Diagrama simplificado do modelo de prototipação

2.4.4 Modelo iterativo e incremental

Larman (2000) apresenta que “Um ciclo de vida iterativo se baseia no aumento e no refinamento sucessivo de um sistema através de múltiplos ciclos de desenvolvimento de análise, de projeto, de implementação e de teste” (ver figura 4).

O modelo **iterativo** corresponde à idéia de melhorar pouco-a-pouco, ou seja, refinar o sistema. A essência do sistema não é alterada, mas o seu detalhe vai aumentando em iterações sucessivas. Por sua vez, o modelo **incremental** corresponde à idéia de aumentar pouco-a-pouco a esfera do sistema, ou seja, alargar o sistema em sucessivos incrementos (SILVA & VIDEIRA, 2001).

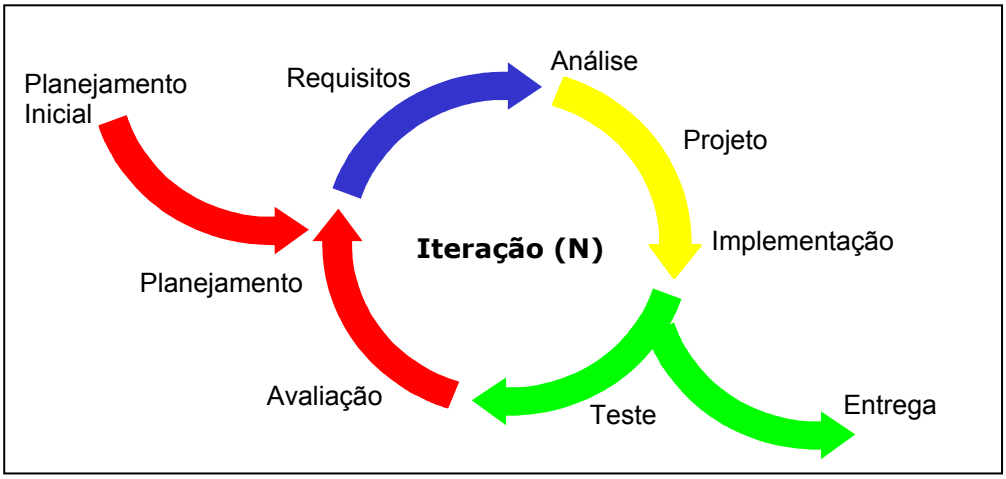


Figura 4: Modelo Iterativo

Desta forma, Silva & Videira (2001) definem que “O princípio subjacente ao modelo **iterativo** e **incremental** é que a equipe envolvida possa refinar e alargar pouco-a-pouco a qualidade, detalhe e âmbito do sistema envolvido”.

Neste sentido, a equipe seleciona o que deve ser feito em cada iteração baseada em dois fatores. Primeiro, a iteração deve trabalhar com um grupo de casos de uso que juntos estendam a usabilidade do produto em desenvolvimento. Segundo, a iteração deve tratar os riscos mais importantes (MARTINS, 1999).

Cantor (1998) apresenta as atividades do modelo iterativo e incremental distribuídas em quatro fases: concepção, elaboração, construção e transição (ver figura 5). Estas fases são organizadas em uma série de atividades. Entretanto, o término de cada fase não é determinado pela execução completa de suas atividades. Desta forma, o projeto pode avançar mesmo quando ainda houver alguma pendência. Estas pendências vão sendo resolvidas ao longo das demais fases. Cada fase tem como propósito o seguinte:

- Concepção: entendimento inicial e concordância da definição do produto, isto é, o que será entregue. A principal atividade é a revisão do escopo do projeto, através de uma modelagem de negócio e revisão dos requisitos do sistema. O principal artefato desta fase é um documento com a visão geral do sistema em desenvolvimento;
- Elaboração: entendimento inicial e concordância do projeto detalhado, isto é, como será construído. O objetivo desta fase é preparar a documentação

necessária para que a codificação do sistema possa acontecer de forma organizada, reduzindo a distância entre os requisitos e os resultados. As principais atividades são a revisão do empacotamento dos requisitos, detalhamento dos requisitos, definição das entidades (inclusive, as persistentes), montagem inicial do plano de testes, organização da arquitetura do sistema, orientação para a codificação e definição da interface;

- Construção: criação do primeiro *build*[2] totalmente funcional. Durante o processo são desenvolvidos *builds* incrementais. Esta fase está associada principalmente com a codificação do sistema. Entretanto, possui também atividades relacionadas com análise e projeto. É necessário realizar uma revisão dos modelos gerados pela fase de elaboração. Além disso, testes serão feitos com base nos componentes desenvolvidos;
- Transição: entrega do produto de acordo com os requisitos iniciais. A fase de transição está relacionada com a implantação e estabilização final do sistema. Além disso, são aplicados os testes de aceite e os ajustes finais.

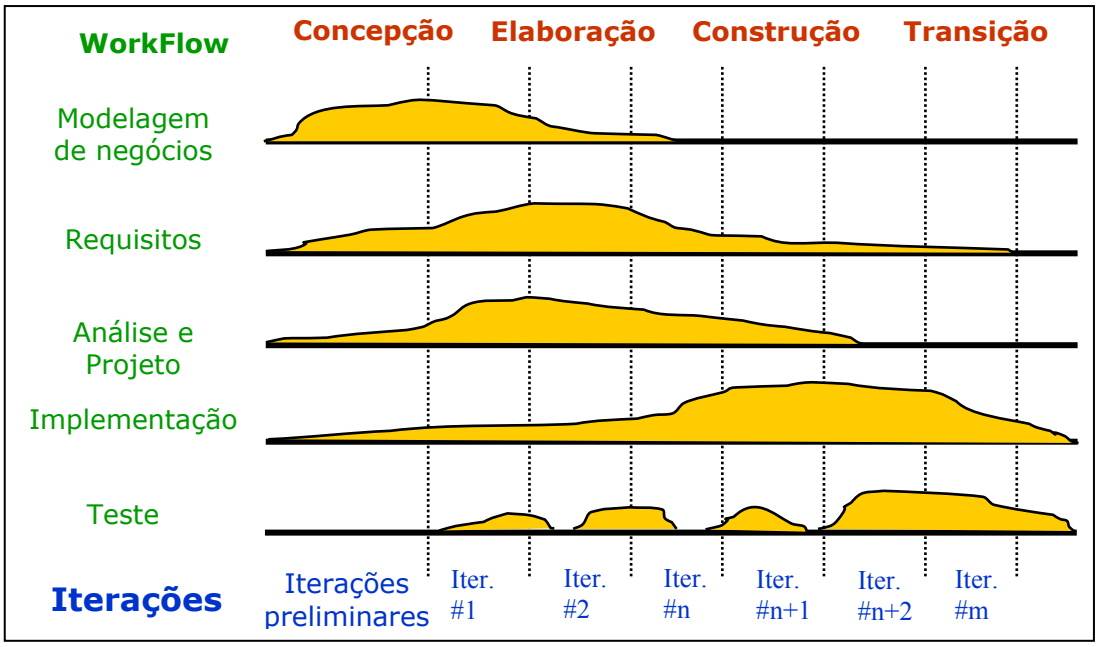


Figura 5: Distribuição das atividades nas fases do modelo iterativo

[2] Um *build* é uma pré-versão do sistema que atende um conjunto de requisitos funcionais.

Há vários benefícios em se adotar um processo iterativo e incremental , entre os quais pode-se destacar (MARTINS, 1999):

- Redução dos riscos envolvendo custos a um único incremento. Se a equipe precisar repetir a iteração, perde-se somente o esforço mal direcionado de uma iteração, não o valor de um produto inteiro;
- Redução do risco de lançar o produto no mercado fora do cronograma previsto. Identificando os riscos na fase inicial do projeto o tempo gasto para gerenciá-los ocorre cedo, quando as pessoas estão sob menos pressão do que numa fase final de projeto;
- Aceleração do tempo de desenvolvimento do projeto como um todo, porque os desenvolvedores trabalham de maneira mais eficiente quando buscam resultados de escopo pequeno e claro;
- Reconhecimento de uma realidade freqüentemente ignorada: as necessidades dos usuários e os requisitos correspondentes não podem ser totalmente definidos no início do processo. Eles são tipicamente refinados em sucessivas iterações. Este modelo de operação facilita a adaptação a mudanças de requisitos.

O modelo iterativo e incremental é um modelo emergente. Seu enfoque é interessante no sentido de que usa a flexibilidade e modularidade do desenvolvimento orientado a objeto. Assim, provê um ciclo de vida que combina ambas a preocupação com “como as pessoas trabalham” e concede o controle de gerenciamento (CANTOR, 1999). Estas características estão presentes em grande parte das metodologias atuais, que focam os aspectos práticos do desenvolvimento, e não em questões filosóficas profundas. Na próxima seção são analisadas algumas metodologias de desenvolvimento, também chamadas de processos leves (*lightweight processes*) ou processos ágeis (BECK, 2000), (FOWLER, 2001).

2.5 Metodologias Ágeis

Fowler (2001) coloca que as metodologias modernas de desenvolvimento, como *Extreme Programming* (XP) e SCRUM, são uma reação a modelos extremamente conceituais e a metodologias “monumentais”. Nas suas palavras:

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

Segundo Evans (2001), por décadas o processo de desenvolvimento de software tem sido um tópico interessante. Mas nos últimos anos, o debate retornou de forma intensa. Pois uma nova abordagem os “processos ágeis ou leves” vem de encontro à abordagem dos “processos pesados” como a tradicional abordagem cascata, comentada na seção anterior, e o RUP, processo introduzido em 1990, para prover produtos baseados em UML (*Unified Modeling Language*) (OMG®, 2001).

Nesta seção estão descritas metodologias ágeis para o desenvolvimento, que são formas específicas de organizar o processo de software para obter vantagens de qualidade e produtividade. A metodologia tem por objetivo especificar tarefas aplicando um conceito específico (*refactoring* e programação em pares no XP, por exemplo) a cada fase do desenvolvimento, e propor soluções práticas para problemas comuns. Posteriormente, serão detalhas em seção independente, a metodologia ICONIX – considerada uma metodologia intermediária – e a metodologia XP, que são as metodologias analisadas neste trabalho, com enfoque maior na análise de requisitos.

2.5.1 Extreme Programming (XP)

O trabalho de Beck (2000) descreve um processo minimalista onde existe muito pouca burocracia envolvida no desenvolvimento. Equipes pequenas, de até 10

desenvolvedores, trabalham em iterações curtas, produzindo software de modo incremental, e analisando requisitos à medida que estes são descritos pelo cliente.

O XP se apóia em um contínuo refinamento do projeto e da implementação deste no código. Para realizar este refinamento, aplica alguns princípios básicos como: propriedade coletiva, programação em pares, histórias do usuário, *refactoring* e testes de unidade. A metodologia XP será detalha posteriormente no capítulo 3.

O interesse que XP tem gerado entre a comunidade de desenvolvimento - desde o ano de 2000 é realizada anualmente a *Extreme Programming Conference* - pode ter relação com sua atitude de "menor esforço" em face de problemas complexos do desenvolvimento (XP2002, 2000). Por exemplo, a documentação do sistema deve ser mantida junto com o próprio código fonte e deve ser mínima, forçando o desenvolvedor a escrever código auto-explicativo e a evitar complexidade.

2.5.2 SCRUM

Uma outra metodologia de desenvolvimento que é classificada como ágil é o SCRUM[3] (SUTHERLAND, 2000). Esta metodologia foi criada na Easel, e posteriormente desenvolvida por duas empresas em conjunto: *Advanced Development Methods* e VMARK. Seu objetivo é fornecer um processo conveniente para projeto e desenvolvimento orientado a objeto (SUTHERLAND, 2000).

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

SCRUM divide o desenvolvimento em iterações (chamadas de *sprints*) de 30 dias. Equipes pequenas, de até 07 (sete) pessoas, são formadas por projetistas, programadores, engenheiros e gerentes de qualidade. Estas equipes trabalham em cima de funcionalidade (os requisitos, em outras palavras) definidas no início de cada *sprint*. A equipe é responsável pelo desenvolvimento desta funcionalidade.

[3] A palavra Scrum vem do nome de uma tática utilizada em *rugby*, onde um grupo de jogadores precisa trabalhar em equipe para recuperar uma bola perdida.

Neste sentido, Fowler (2001) destaca que o ponto é estabilizar os requisitos durante o *sprint*.

Ainda de acordo com Fowler (2001), todo dia, é feita uma reunião de 15 minutos onde o time expõe à gerência o que será feito no próximo dia, e nestas reuniões os gerentes podem levantar os fatores de impedimento (*bottlenecks*), além de avaliar o progresso geral do desenvolvimento. SCRUM é interessante porque fornece um mecanismo de informação de status que é atualizado continuamente, e porque utiliza a divisão de tarefas dentro da equipe de forma explícita.

Schwaber (1997) ressalta que a análise, o projeto, e os processos de desenvolvimento na fase de *sprint* são inconstantes. Entretanto, um mecanismo de controle é usado para administrar a imprevisibilidade e controlar o risco. O resultado é flexibilidade, receptividade e confiabilidade (ver figura 6).

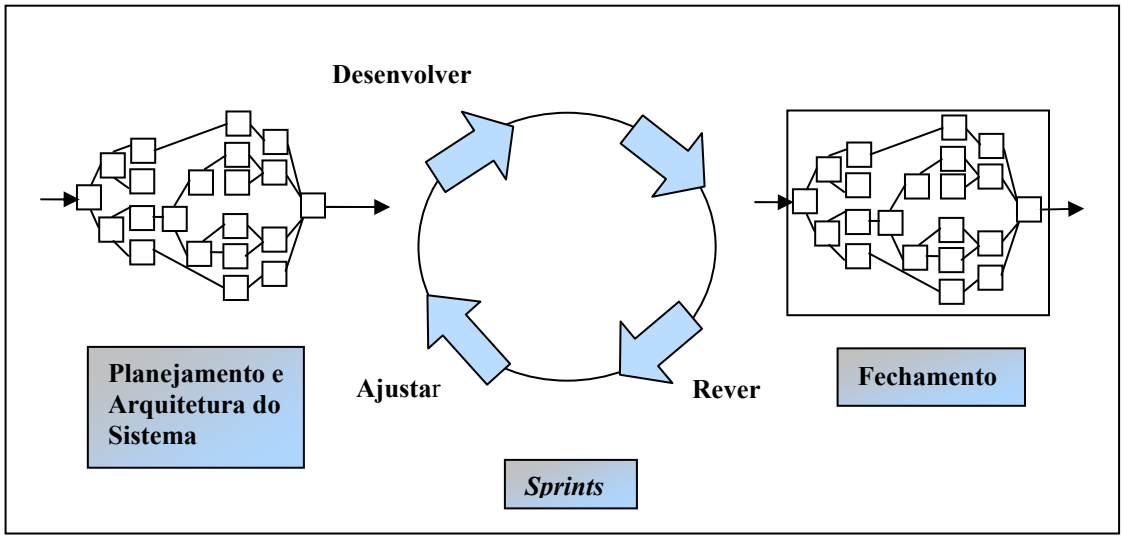


Figura 6: Metodologia SCRUM (adaptado de SCHWABER, 1997)

Segundo Schwaber (1997), as características da metodologia SCRUM são:

- A primeira fase e a última fase (planejamento e fechamento) consistem em definir processos. Todos os processos de entrada e de saída são bem definidos e a forma de como executar o processo é explicito. O fluxo é linear, com algumas iterações na fase de planejamento;
- A fase de *sprint* é um processo empírico, ou seja, baseado somente na experiência do time. Alguns processos na fase de *sprint* não são identificados ou controlados. Então, são tratados como controles de requisitos externos.

Estes controles, incluindo os riscos de gerenciamento, são colocados em cada iteração da fase de *sprint* para evitar o caos enquanto maximizam flexibilidade;

- *Sprints* são flexíveis e não-lineares. Se disponível, o conhecimento explícito do processo é usado. Caso contrário, o conhecimento tácito, tentativa, e erro são usados para construir o conhecimento do processo. Os *sprints* servem para evoluir no produto final;
- O projeto permanece aberto ao ambiente até a fase de fechamento. O produto a ser entregue pode ser mudado a qualquer momento durante o planejamento e as fases de *sprint* do projeto. O projeto permanece aberto à complexidade do ambiente, inclusive à competitividade, tempo, qualidade, e pressões financeiras, ao longo destas fases;
- O produto a ser entregue é determinado durante o projeto, baseado no ambiente.

Nos últimos anos, o método de desenvolvimento SCRUM tem rapidamente ganhado reconhecimento como uma ferramenta eficaz para desenvolvimento produtivo de software. Mesmo quando, padrões de SCRUM são combinados com outros padrões organizacionais existentes, eles são altamente adaptáveis, ainda que em organizações de desenvolvimento de software bem estruturadas (SUTHERLAND, 2000).

2.5.3 Crystal/Clear

Crystal/Clear faz parte, na realidade, de um conjunto de metodologias criado por Cockburn (2000). As premissas apresentadas para a existência deste conjunto são:

- todo projeto tem necessidades, convenções e uma metodologia diferente;
- o funcionamento do projeto é influenciado por fatores humanos e há melhora neste quando os indivíduos produzem melhor;
- uma comunicação melhor e lançamentos freqüentes reduzem a necessidade de construir produtos intermediários do processo.

Crystal/Clear é uma metodologia direcionada a projetos pequenos, com equipes de até 6 (seis) desenvolvedores. Assim como com o SCRUM, os membros da equipe têm especialidades distintas. Existe uma forte ênfase na comunicação entre os membros do grupo, e a organização do espaço de trabalho deve permitir este tipo de colaboração. O enfoque do *Crystal/Clear* está em alcançar o sucesso do projeto por realçar o trabalho das pessoas envolvidas.

Toda a especificação e projeto são feitos informalmente, utilizando quadros publicamente visíveis. Os requisitos são elaborados utilizando casos de uso - *use case* em UML (OMG®, 2001), um conceito similar aos cartões de histórias em XP, onde são enunciados os requisitos como tarefas e um processo para sua execução. A liberação das versões de software são feitos em incrementos regulares de um mês, e existem alguns subprodutos do processo que são responsabilidade de membros específicos do projeto.

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

Fowler (2001) enfatiza que *Crystal* compartilha uma orientação humana com XP, mas não segue um processo disciplinado. Embora *Crystal* seja menos produtivo que XP, mais pessoas poderão seguir esta metodologia.

Ainda de acordo com Fowler (2001), em fevereiro de 2001 Alistair Cockburn anunciou que ele e Jim Highsmith - autor da metodologia de Desenvolvimento de Software Adaptável (ASD) - estão unindo suas metodologias. Ambos, contribuem e continuam a fundir suas percepções sobre princípios e práticas para executar um projeto tão rápido e tão eficazmente quanto permitem de circunstâncias.

2.6 Considerações Finais

Grande parte das pesquisas feitas na área de Engenharia de Software, e em particular no desenvolvimento de Processos de Software, continuam sendo desenvolvidas e contribuindo para melhorias na construção de produtos de software. No entanto, existe uma tendência atual para a simplificação e pragmatização do

processo para acomodar novas necessidades de desenvolvimento, os requisitos e demandas por novos sistemas são muito diferente dos conhecidos e estabelecidos nos anos 70 e 80 (EVANS, 2001).

Neste contexto, vem à tona o destaque da literatura nos requisitos, sejam eles documentados em cartões de histórias (segundo a abordagem do XP) ou documentados através da análise de requisitos diretamente associada aos casos de uso (segundo a abordagem ICONIX). Na seção seguinte é abordado o processo XP e na seção subseqüente o processo ICONIX.

3 EXTREME PROGRAMMING (XP)

3.1 Introdução

Extreme Programming (XP) é uma metodologia leve, eficiente, flexível e de baixo risco para times pequenos e médios, que desenvolvem software com requisitos dinâmicos ou em constante mudança (BECK, 2000).

A metodologia XP foi criada por Kent Beck, que no início dos anos 1990 pensava sobre caminhos melhores para desenvolver software. Em março de 1996 Kent começou um projeto na Daimler Chrysler usando novos conceitos em desenvolvimento de software. O resultado era a metodologia XP (WEELS, 2002).

De acordo com Beck (2000), XP se distingue de outras metodologias por:

- apresentar *feedback* (retornos) contínuos e concretos em ciclos curtos;
- abordar planejamento incremental, apresentando rapidamente um plano global, que evolui durante o ciclo de vida do projeto;
- ter habilidade flexível de programar implementação de funcionalidade, respondendo as mudanças das regras de negócio;
- confiar nos testes automatizados escritos pelos programadores e clientes, para monitorar o progresso do desenvolvimento, permitindo a evolução do sistema e detectando antecipadamente os problemas;
- acreditar na comunicação oral, na colaboração íntima dos programadores, nos testes e no código fonte, definindo a estrutura do sistema e os objetivos;
- confiar no processo de evolução do projeto, que dura tanto quanto o sistema;
- acreditar nas práticas que trabalham tanto com as aptidões, a curto prazo, dos programadores, quanto os interesses, a longo prazo, do projeto.

XP é uma disciplina de desenvolvimento de software baseado em valores de simplicidade, comunicação, feedback e coragem. XP envolve o time inteiro para um trabalho de equipe com práticas simples, com feedback suficiente para capacitar o

time a ver onde eles estão e a convergir para práticas em uma solução única (JEFFRIES, 2001).

No XP, cada contribuinte do projeto é um integrante do time inteiro. O time trabalha com um representante de negócio chamado de “o Cliente”, que participa ativamente com o time diariamente.

Os times usam uma forma simples de planejamento e acompanhamento para decidir qual a próxima tarefa a ser realizada e predizer quando o projeto será feito. Definidas as regras de negócio, o time produz o software em uma série de pequenas versões, que passam por todos os testes que o cliente definiu.

Ainda de acordo com Jeffries (2001), os programadores trabalham em pares e em grupo, com projeto simples e código obsessivamente testado, melhorando o projeto continuamente para manter sempre as necessidades correntes e o sistema integrado. Os programadores escrevem todo código de produção em pares, e todos trabalham junto o tempo todo. Eles codificam de forma consistente para que todos possam entender e melhorar o código quando necessário.

3.2 Os Quatro Valores do XP

É importante ressaltar que, o XP segue um conjunto de valores, princípios e regras básicas que visam alcançar eficiência e efetividade no processo de desenvolvimento de software (BECK, 2000). Os valores são quatro: comunicação, simplicidade, *feedback* e coragem. Nestes valores, estão fundamentados alguns princípios básicos: *feedback* rápido, simplicidade assumida, mudanças incrementais, compreensão às mudanças e qualidade do trabalho. A partir destes princípios, foram definidas as doze práticas básicas que são adotadas pelo XP.

Os quatro valores do XP são:

- **Comunicação**: É o primeiro valor do XP. Deve-se preferir sala de discussão (*chat*) a correio (*e-mail*), telefonemas a *chat*, conversar pessoalmente a telefonemas, trabalhar na mesma sala a ter salas isoladas, trabalhar em conjunto a revisar o resultado final (WUESTEFELD, 2001). Jeffries (2001) enfatiza que deve existir comunicação em todos os sentidos. O time deve escrever o código usando as mesmas convenções de formatação, nomeando

convenções, e definindo os padrões de codificação. Desta forma, independente que quem escreveu o método é familiar e fácil de entender. A comunicação deve ser facilitada para todo o time do XP;

- **Simplicidade**: É o segundo valor do XP. O objetivo é simplificar continuamente o software. É isso que sustenta a premissa de extremo, pois a simplicidade não é fácil. O processo em si também é adaptado, a cada dia, se houver como torná-lo mais simples. Simplicidade e comunicação estão diretamente relacionadas. Pois quando mais comunicação existe no time, mais claro fica de se ver o que realmente precisa ser feito e, mais seguro de ver o que não precisa ser feito. Assim, se faz o software mais simples quando é possível de se trabalhar;
- ***Feedback***: É o terceiro valor do XP. Todo problema é evidenciado o mais cedo possível para que possa ser corrigido o mais rápido possível. Toda oportunidade é descoberta o mais cedo possível para que possa ser aproveitada o mais rápido possível (WUESTEFELD, 2001) (ver figura 7);
- **Coragem**: Este valor somente tem peso se estiver combinado com os três primeiros valores (BECK, 2000). É preciso coragem para apontar um problema no projeto, pedir ajuda quando necessário, simplificar código que está funcionando, expor para o cliente que o prazo estimado para implementar determinado requisito não é suficiente, fazer alterações no processo de desenvolvimento. Ou seja, fazer a coisa certa mesmo que não pareça o mais correto naquele momento.

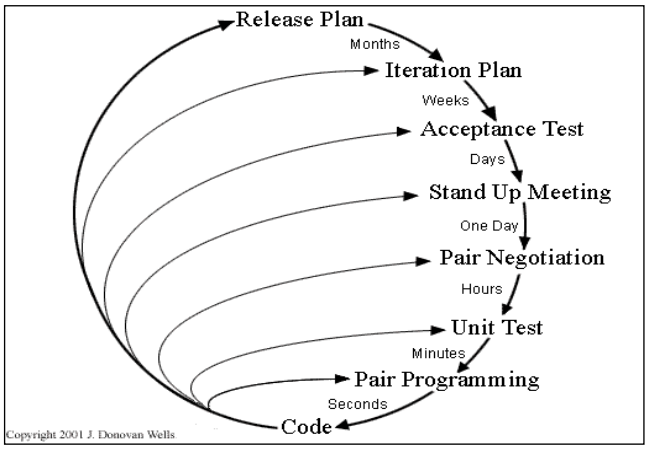


Figura 7: Diagrama de planejamento e *feedback* (WELLS, 2001)

3.3 As Doze Práticas do XP

1. **Jogo de Planejamento** (*Planning Game*): Rapidamente determina o escopo das próximas versões combinando a prioridades do negócio e estimativas técnicas. Como prática, estrutura o plano e atualiza o plano.
2. **Pequenas versões** (*Small releases*): O time XP coloca rapidamente um sistema simples em produção, e o atualiza freqüentemente em ciclos bastante curtos.
3. **Metáforas**: Guiam todo o desenvolvimento e a comunicação com o cliente utilizando um “sistema de nomes” e uma descrição do sistema.
4. **Projeto simples** (*Simple design*): O sistema precisar ser projetado o mais simples possível satisfazendo os requisitos atuais.
5. **Teste**: Os times XP focalizam a validação do software durante todo o processo. Os programadores escrevem primeiro os testes, e só então continuam o desenvolvimento que deve atender os requisitos destes testes. Os clientes escrevem testes para validar se os requisitos estão sendo atendidos.
6. ***Refactoring***: Os times procuram aperfeiçoar o projeto do sistema durante todo o desenvolvimento, mantendo a clareza do software: sem ambigüidade, com alta comunicação, simples, porém completo.
7. **Programação em pares** (*Pair Programming*): Todo código produzido é feito em duplas, ou seja, dois programadores trabalhando juntos na mesma máquina.
8. **Propriedade coletiva** (*Colletive ownership*): Todo o código pertence a todo o time. Então, qualquer um pode alterar qualquer código em qualquer tempo.
9. **Integração contínua** (*Continnuous integration*): Integra e constrói o sistema de software várias vezes por dia. A todo o momento, uma tarefa é completada.
10. **Semana de 40-horas** (*40-hour week*): Como regra, não se deve trabalhar mais que 40 (quarenta) horas na semana. Programadores exaustos cometem mais erros.

11. **Cliente dedicado** (*On-site customer*): Os times XP tem "o cliente" real, disponível todo o tempo, que determina os requisitos, atribui as prioridades, e reponde as dúvidas.
12. **Código padrão** (*Coding standards*): Todos os programadores escrevem o código da mesma forma, de acordo com regras que asseguram a clareza e a comunicação através do código.

A maioria das regras XP causa polêmica à primeira vista e muitas não fazem sentido se aplicadas isoladamente. É a sinergia de seu conjunto que sustenta o processo XP, encabeçando uma verdadeira revolução de metodologias ágeis.

Nas próximas seções, cada prática será delineada e, examinada a conexão entre elas, que permite que as fraquezas de uma prática sejam superadas pelas forças de outras práticas (BECK, 2000).

A figura 8 é um diagrama que representa a colaboração entre as práticas. Cada prática é uma parte simples. A riqueza está na interação das partes. A linha de ligação entre duas práticas significa que elas dão suporte uma a outra.

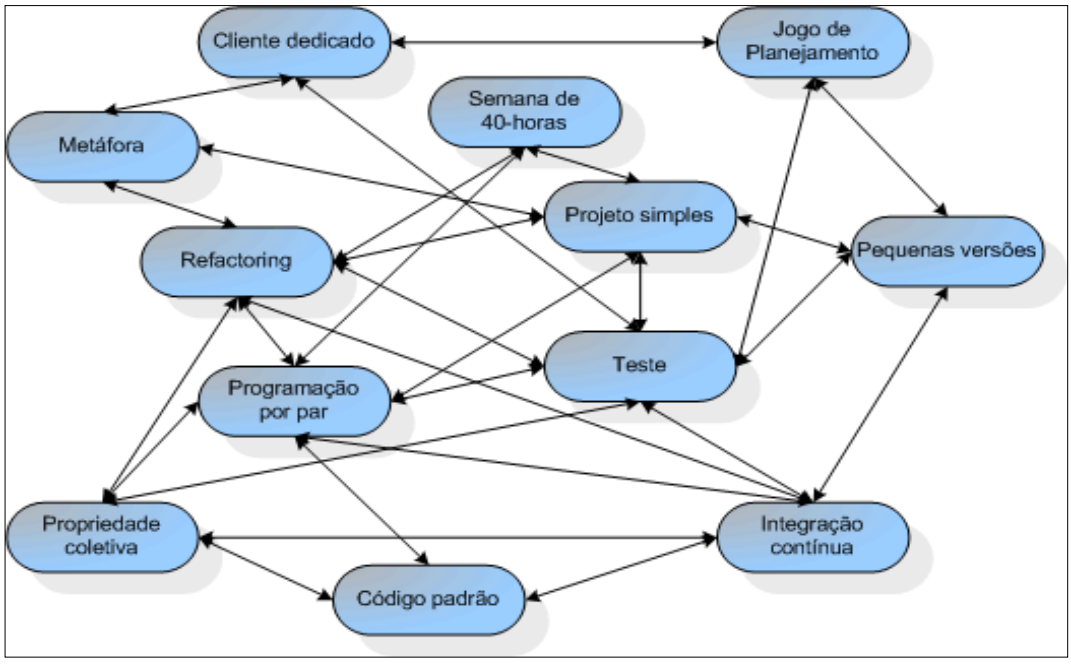


Figura 8: Colaboração entre as práticas (adaptado de BECK, 2000)

3.3.1 Jogo de planejamento

Conforme Beck (2000), o jogo de planejamento é suportado tanto pelas regras de negócio quanto pelas considerações técnicas. As regras de negócio são estimadas pelo cliente de negócio, que decide sobre: escopo, prioridade, composição das versões e datas das versões. As considerações técnicas são estimadas pelos técnicos, que decidem sobre: tempo, riscos técnicos e processo.

Planejamento no XP responde duas perguntas chaves em desenvolvimento de software: predizer o que será realizado até determinada data, e determinar qual a próxima tarefa a ser realizada. A ênfase está em guiar o projeto, em lugar de especificar exatamente o que será necessário e quanto tempo levará, o que já é bastante difícil (JEFFRIES, 2001). Existem dois passos chave em planejamento XP, abordando estas duas perguntas:

Versão de Planejamento (*Release Planning*) é o passo onde o cliente apresenta as características desejadas aos programadores, e os programadores estimam sua dificuldade. A figura 9 mostra a estrutura de um cartão de história. XP preconiza versões freqüentes e pequenas a cada dois ou três meses. Com as estimativas das dificuldades, e com conhecimento da importância das características, o cliente atinge um plano para o projeto. Inicialmente, as versões dos planos são imprecisas. Uma vez que, as prioridades e as estimativas não são verdadeiramente sólidas, até que o time comece a trabalhar. Então, pode-se saber o quão rápidos eles irão executar o trabalho.

Cartão de História e Tarefa

Data: ___/___/_____ Tipo de Atividade: Nova:____ Dificuldade: _____ Valor: _____

Número da História: _____ Prioridade: Usuário: _____ Técnico: _____

Referência Anterior: _____ Risco: _____ Estimativa do Técnico: _____

Descrição da Tarefa:

Notas:

Acompanhamento da Tarefa:

Data	Estado	Para Realizar	Comentário

Figura 9: Modelo de cartão de história (adaptado de BECK, 2000)

Iteração de Planejamento (*Iteration Planning*) é o passo em que o time recebe orientação através de cartões de tarefas (BECK, 2001). A figura 10 é um modelo de cartão de tarefa. Os times XP constroem software em "iterações" de duas a três semanas, entregando um software executável e útil no fim de cada iteração. Durante a Iteração de Planejamento, o cliente apresenta as características desejadas para as próximas duas semanas. Os programadores quebram as características em tarefas, e estimam o tempo (em um nível mais detalhado que na Versão de Planejamento). O time baseado no volume de trabalho realizado na iteração prévia, estima o tempo que será empreendido na iteração corrente.

Nesta proposta, a cada duas semanas, a quantia de progresso é completamente visível. Não existe "noventa por cento feitos" em XP: um conjunto de características é totalmente completado, ou não é. Este enfoque pode resultar em um bom e pequeno paradoxo: por um lado, com tanta visibilidade, o cliente está em uma posição para cancelar o projeto se o progresso não for suficiente. Por outro lado, o progresso é tão visível, e a habilidade de decidir o que será feito na próxima iteração é tão completa, que projetos XP tendem a entregar mais do que é necessário, com menos pressão e tensão (JEFFRIES, 2001).

Cartão de Tarefa

Data: ___/___/_____

Número da História: _____ Autor do Software: _____ Estimativa da Tarefa: _____

Descrição da Tarefa:

Notas do Autor do Software:

Acompanhamento da Tarefa:

Data	Realizado	Para Realizar	Comentário

Figura 10: Modelo de cartão de tarefa (adaptado de BECK, 2000)

3.3.2 Pequenas versões

Segundo Jeffries (2001), os times XP trabalham com pequenas versões em dois modos importantes:

Primeiro, o time disponibiliza uma versão executável, considerando que o software está testado, com as características escolhidas pelo cliente naquela iteração. O cliente pode usar este software para qualquer propósito, tanto para avaliação, quanto para liberar aos usuários finais (altamente recomendado). O aspecto mais importante é que o software é visível, e dado ao cliente no final de cada iteração. Assim, mantém o projeto aberto e tangível.

Segundo, o time XP freqüentemente disponibiliza para seus usuários finais uma versão. Nos projetos XP Web, com duração mensal ou menor, são disponibilizadas versões diárias, quando possível.

Jeffries (2001) argumenta que pode parecer impossível criar boas versões nesta freqüência, mas times XP por toda parte estão fazendo isso. Pode-se ver mais adiante em “Integração Contínua”, que versões freqüentes são mantidas confiáveis com teste obsessivos do XP, como descrito na seção “Teste”.

Todas as versões devem ser tão pequenas quanto possíveis, e devem conter os requisitos mais importantes (BECK, 2000).

3.3.3 Metáfora

O projeto de software em XP é guiado por uma metáfora simples (BECK, 2000). A metáfora ajuda a manter todos os desenvolvedores em sintonia com o projeto e auxilia a definir nomes a objetos ou classes. É bastante importante como uma prática para manter o código padrão.

Jeffries (2001) comenta que, às vezes uma metáfora suficientemente poética não surge. Entretanto, com ou sem uma imagem explícita, os times XP usam um sistema comum de nomes para que todos entendam como o sistema trabalha. Assim, pode-se localizar a funcionalidade que se está procurando, ou saber o lugar certo para pôr a funcionalidade que se quer adicionar.

Por exemplo, um cliente necessita de um sistema de *HelpDesk* com características similares ao Microsoft Outlook®. Então, o “Outlook” pode ser a metáfora (ULIANA, 2001).

3.3.4 Projeto simples

Os times XP constroem o software de acordo com um projeto simples. Desta forma, o time XP mantém o projeto exatamente delineado com a funcionalidade corrente do sistema. Sendo assim, não existe nenhum movimento perdido, e o software está sempre pronto para a próxima iteração (JEFFRIES, 2001).

Beck (2000) enfatiza que o projeto certo para um software em qualquer tempo é aquele que:

- executa todos os testes;
- não tem lógica duplicada (deve-se ter cuidado com hierarquia de classes paralelas);
- expõe claramente cada intenção para os programadores;
- tem o menor número possível de classes e métodos.

Projeto em XP não é algo que possa ser realizado uma única vez. É necessário ser feito o tempo todo. Pois, existem passos do projeto feitos na versão de planejamento e na iteração de planejamento, e cada vez mais, o time toma parte nestas sessões e na revisão rápida do projeto por *refactoring*. Em processos incrementais e iterativos, como *Extreme Programming*, um bom projeto é indispensável. É por isso que existe tanto enfoque em projeto ao longo do curso de todo o desenvolvimento (JEFFRIES, 2001).

Wells (2002) aborda também que um projeto simples sempre requer menos tempo para terminar que um complexo. Desta forma, o objetivo é fazer um projeto mais simples, mas suficiente para se trabalhar. Isto parece coerente uma vez que é mais rápido e mais barato substituir um código complexo durante o desenvolvimento do que desperdiçar muito mais tempo no futuro.

3.3.5 Teste

Os programadores primeiramente escrevem os testes de unidade e o cliente escreve os testes funcionais, de forma que a confiança na funcionalidade possa se tornar parte do programa desenvolvido (BECK, 2000).

Fowler & Foemmel (2000) destacam que os testes de unidade escritos pelos programadores têm por finalidade testar uma classe individual ou um grupo de classes. Já os testes funcionais (também chamados de testes de aceitação) escritos pelo cliente têm por finalidade testar o sistema de ponta-a-ponta. É importante ressaltar que os testes são escritos antes do código ser construído.

Criar testes de unidade ajuda o desenvolvedor a considerar o que realmente tem que ser feito. Os requisitos são reforçados pelos testes. Não pode haver erros de interpretação em uma especificação escrita na forma de código executável.

Nesta proposta, Oshiro (2001) destaca que as atividades de teste são realizadas durante todo o processo de desenvolvimento, e o código é construído com a finalidade de satisfazer os resultados esperados. E à medida que um novo código é adicionado, novos testes devem ser executados e assim garantir que não ocorram impactos negativos.

Testes de Unidade são considerados elementos chaves em XP (WELLS, 2002), pois são criados antes do código e armazenados em um repositório junto ao código que será testado. Um fator importante para a utilização dos testes de unidade, sobretudo se for automatizado, é a economia de custo que podem proporcionar ao identificar erros. Desta forma, um conjunto completo de testes de unidade deve estar disponível no início do projeto, e não no final.

Testes Funcionais constituem a primeira indicação que as histórias dos clientes são válidas, mostram que cada história pode ser testada (ASTELS,2002). Normalmente, são escritos pelos clientes ou usuários finais através dos cartões de história (WELLS, 2002), com suporte de um membro do time responsável por testar o sistema. Testes funcionais também são usados como testes de validação, anteriores à liberação de uma versão do sistema.

3.3.6 Refactoring

O enfoque do XP está em entregar as regras de negócios a cada iteração. Para realizar isto no decorrer do projeto inteiro, o software deve estar bem projetado. Então, o XP usa um processo contínuo de melhoria de projeto chamado *refactoring*,

do título do livro de Martin Fowler, “*Refactoring: Improving the Design of Existing Code*” (JEFFRIES, 2001).

Refactoring é a técnica empregada na reestruturação do código, cujo principal propósito é fazer com que um programa fique mais reutilizável e fácil de entender, sem que haja mudança de comportamento (FOWLER, 2001).

O objetivo do processo de *refactoring* é remover duplicação (um sinal certo de projeto pobre) e acrescentar a "coesão" ao código, enquanto baixa o acoplamento. “Alta coesão e baixo acoplamento” é reconhecida pelo menos nos últimos 30 (trinta) anos como marca de código bem projetado. O resultado é que os times XP começam com uma vantagem, projeto simples, e continuam com essa vantagem durante todo o processo. Fowler (2001) enfatiza que bom projeto é essencial para manter a velocidade do desenvolvimento do software. Nesta proposta, a técnica de *refactoring* ajuda a desenvolver o código mais rapidamente.

Refactoring é o processo de melhorar o projeto de um código que já está funcionando. Entretanto, fica a questão, afinal qual o melhor projeto para um software? Wuestefeld (2001) responde que, “O melhor projeto para um software é aquele que, por prioridade:

- passa 100% nos testes de unidade;
- tem performance aceitável;
- é o mais claro, o mais fácil de entender”.

Ainda de acordo com Wuestefeld (2001), deve-se fazer *refactoring* onde for possível e sempre que possível. Reescrever o código atual, parte por parte, fazendo com que ele fique cada vez mais manutenível. Os testes de unidade oferecem segurança para o que o que já estava funcionando continue funcionando.

3.3.7 Programação em pares

Segundo Jeffries (2001), “Todo software produzido em XP é construído por dois programadores, sentados lado a lado, na mesma máquina. Esta prática assegura que todo código produzido é revisado por, pelo menos outro, programador”. O resultado desta prática é um projeto rápido, com testes mais eficazes e um código

melhor. Pode parecer ineficiente ter dois programadores fazendo o trabalho de um, mas Williams (2001) ressalta que o contrário é verdadeiro. Pesquisas mostram que o código produzido por dois programadores é melhor e a velocidade da produção é quase equivalente ao trabalho realizado por um programador isolado (WILLIAMS, 2000). A figura 11 mostra a comparação do tempo de conclusão de projetos realizados por pares de programadores e por programadores individuais.

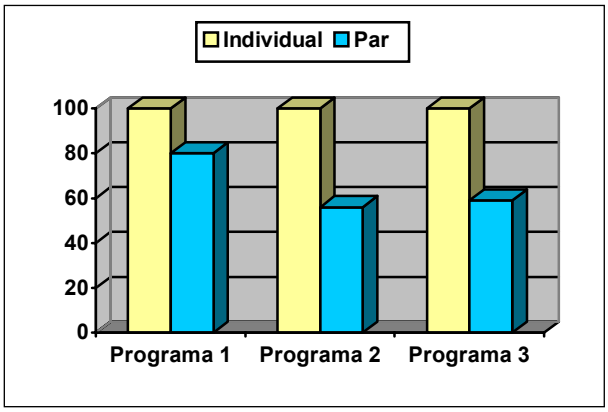


Figura 11: Comparação: tempo de programação (adaptado de WILLIAMS, 2000)

É importante observar que existem dois papéis em cada par. O parceiro que está com o teclado e o mouse, pensa na melhor forma de implementar o método corrente. Enquanto que, o outro parceiro pensa mais estrategicamente em: se aquela abordagem irá funcionar, se existe algum teste que ainda não foi trabalhado e se existe alguma forma que possa simplificar o sistema corrente para que problema desapareça (BECK, 2000).

A programação em pares necessita de prática para ser realizada bem. É necessário praticar algumas semanas para ver se o resultado é satisfatório. De acordo com Jeffries (2001), noventa por cento dos programadores que aprende programação em pares prefere esta à programação isolada. Então, o XP recomenda a programação em pares para todos os times. Esta técnica, além de prover código e testes melhores, também serve para transferir conhecimento a todo o time. Assim, todos conseguem o benefício do conhecimento especializado do outro. Os programadores aprendem, suas habilidades melhoram e se tornam mais valiosos para o time e para a companhia.

3.3.8 Propriedade coletiva

Em um projeto XP, qualquer par de programadores pode melhorar qualquer código em qualquer hora. Isto significa que todo código consegue o benefício de muita atenção das pessoas, o que acrescenta qualidade ao código e reduz defeitos (JEFFRIES, 2001).

Segundo Beck (2000) o time XP, como um todo, é responsável por cada programa de código. Não é necessário pedir autorização para alterar qualquer programa. Entretanto, isso somente é possível com os testes de unidade bem elaborados, permitindo segurança aos programadores que trabalham no código. Além disso, a propriedade coletiva também tende a espalhar conhecimento de todo o sistema para o time.

Wuestefeld (2001) aborda que a prática de “integração contínua” auxilia a prática de propriedade coletiva, pois, faz com que os programadores que trabalham neste ambiente, nem percebam se o código que eles programaram foi modificado ou estendido.

3.3.9 Integração contínua

Os times XP mantêm o sistema integrado o tempo todo. O código é integrado e testado depois de algumas horas, no máximo depois de um dia de desenvolvimento. A forma mais simples para fazer isto, é ter uma máquina dedicada para a integração (BECK, 2000). Ferramentas de controle de versão podem ser úteis neste momento.

De acordo com Wells (2002), a integração contínua freqüentemente evita a divergir ou a fragmentar os esforços de desenvolvimento. Nesta proposta, os programadores devem estar em constante comunicação um com os outros, pois assim, eles sabem o que pode ser reutilizado, ou o que pode ser compartilhado. Todo o time precisa trabalhar com a versão mais recente, evitando perder tempo com versão obsoleta. Então, a integração deve ser feita vária vezes por dia. Cada par de programação é responsável por integrar seu próprio código. Isto pode ser feito quando os testes de unidade tenham sido executados 100% em cada funcionalidade planejada. A integração contínua evita ou descobre problemas de compatibilidade cedo.

3.3.10 Semana de 40-horas

Não é produtivo fazer horas extras por períodos maiores que uma semana. Como também não é produtivo passar a noite acordado e trabalhar no dia seguinte.

Wuestefeld (2001) enfatiza que as pessoas não possuem o mesmo ritmo de trabalho, por isso não existe um número de horas ideal para todas as pessoas. Algumas serão mais produtivas com 07 (sete) horas e outras com 08 (oito) ou com 09 (nove) horas. Os excessos é que estão fora de questão, como trabalhar 01 (uma) hora por dia ou 20 (vinte) horas por dia.

O time XP trabalha intensamente de modo a maximizar a produtividade. Um programador quando está cansado, raciocina mais lentamente e fica mais distraído. Um programador cansado é ótimo para inserir erros (*bugs*) em um programa, erros que vão dar trabalho para serem corrigidos e exigir mais horas extras.

De acordo com Beck (2000), a sobrecarga de trabalho é sintoma de sérios problemas no projeto. A regra XP é simples, não se deve trabalhar uma segunda semana fazendo horas extras. Uma semana trabalhando horas a mais, para por o planejamento em ordem, é admissível. Entretanto, se isso se repetir na semana seguinte, pode indicar que algo está errado.

3.3.11 Cliente dedicado

Um “cliente real” precisa sentar com o time, deve estar sempre disponível, não somente para auxiliar, mas para fazer parte da equipe. Beck (2000) define por “cliente real”, aquele cliente que realmente irá utilizar o sistema quando ele estiver em produção. O objetivo desta regra é ter o usuário real do sistema, de preferência junto com o desenvolvimento. O cliente é muito valioso para o time.

A principal vantagem (WUESTEFELD, 2001) é que o cliente poderá fornecer detalhes do sistema quando surgirem dúvidas. E surgem muitas dúvidas quando os programadores estão implementando uma funcionalidade. Se o cliente estiver sempre por perto, significa que toda e qualquer dúvida pode ser prontamente esclarecida e que o cliente tem um controle maior do que está realmente sendo implementado.

Entretanto, algumas vezes, não é possível que o cliente esteja no mesmo local de trabalho que os programadores. Ainda assim, é possível trabalhar com XP, desde que se tenha um canal aberto de comunicação e que o cliente tenha disponibilidade para:

- atender os programadores quando surgirem às dúvidas;
- produzir valor para o projeto escrevendo testes funcionais;
- definir o escopo e definir as prioridades para os programadores.

Nestes casos seria indicado o uso de ambientes colaborativos através da Internet. Por sua vez, se o time não incluir um cliente real, torna-se necessário adicionar um risco ao projeto para o futuro, pois os programadores irão codificar sem saber exatamente que testes eles devem satisfazer e quais eles devem ignorar.

3.3.12 Código padrão

Os times XP seguem um padrão de codificação comum a todos os membros. Desta forma, todo código no sistema pode ser visto como se fosse escrito por um único programador. As particularidades do padrão não são importantes: o que é importante é que todo o código deve parecer familiar, em defesa da propriedade coletiva (JEFFRIES, 2001).

De acordo com Wells (2002) a padronização do código mantém o mesmo consistente e fácil para todo o time entender e/ou fazer o *refactoring*. Não importa qual é o padrão, desde que haja um. Ele não precisa ser arbitrariamente definido, podendo emergir naturalmente como resultado da propriedade coletiva (programação em pares).

3.4 Ciclo de Vida e as Fases do Processo XP

O projeto ideal em XP é aquele que inicia por uma curta fase de desenvolvimento, seguida de anos de produção e refinamentos simultâneos e finalmente encerra quando o projeto não faz mais sentido (BECK, 2000).

O ciclo de vida e as fases do processo XP são abordagens muito discutidas por diversos autores que adotam essa metodologia. O ciclo de vida XP é bastante curto

e, à primeira vista, difere dos padrões dos modelos de processo convencionais. Entretanto, esta abordagem faz sentido somente em um ambiente onde as mudanças de requisitos do sistema são fatores dominantes (OSHIRO, 2001). No caso extremo, os requisitos podem mudar no meio da versão, para atender funcionalidades mais importantes do que as definidas no planejamento original.

3.4.1 Exploração

De acordo com Wake (2002), o objetivo da fase de Exploração é entender o que o sistema deve fazer, bem o suficiente para que possa ser estimado. Sendo que, o cliente escreve e administra histórias (*story cards*) e o programador estima as histórias. Encerra quando todas as histórias necessárias para a próxima fase – fase de planejamento - tenham sido estimadas (ver figura 12).

A fase de Exploração deve proporcionar confiança suficiente para o time de forma que, com as ferramentas que possui consiga iniciar e finalizar o programa. O time deve acreditar que tem (ou pode aprender) os conhecimentos necessários para o projeto. Enfim, o time XP precisar aprender a confiar em todos os membros da equipe (BECK, 2000).

A fase de Exploração começa com as regras de negócio. O cliente escrevendo cartões de história que descrevem o que o sistema precisa fazer. Wake (2002) considera que, histórias menores tendem a ter risco menor. Então, se uma história for muito grande, o cliente deve quebrar a história para ficar mais flexível ou, quebrar a pedido dos programadores, que podem estar sentindo dificuldades de estimar. Se os programadores não souberem como estimar alguma coisa, eles podem fazer uma rápida programação da história, ou seja, uma pontuação (*spike*), que pode durar minutos, horas ou no máximo dois dias (ver figura 5). O resultado de uma pontuação é conhecimento suficiente para tentar uma estimativa.

Em paralelo as histórias dos clientes os programadores estão experimentando ativamente as diferentes tecnologias e as possíveis configurações, explorando as possibilidades para a arquitetura do sistema. Leva-se de uma a duas semanas testando a arquitetura, mas deve-se testar de três a quatro maneiras, ou seja, diferentes pares podem testar diferentes tecnologias e comparar ou, dois pares

podem testar a mesma tecnologia e comparar as diferenças emergentes. Se este período não for suficiente para testar a tecnologia, então a tecnologia deve ser classificada como um risco para o projeto. Entretanto, durante a fase de exploração pode-se convidar um especialista na tecnologia escolhida, o qual não irá perder tempo em pequenos problemas, pois sabe por experiência como resolvê-los.

Beck (2000) ressalta que esta pequena exploração na arquitetura, auxilia os programadores a terem uma idéia da implementação quando o usuário apresentar seus cartões de história. Pois, os programadores precisam estimar cada tarefa de programação durante a exploração. Quando uma tarefa é feita, eles precisam repassar para o calendário o tempo gasto para executar aquela tarefa. A prática da estimativa cria confiança no time para quando eles realmente tiverem que estimar o tempo.

3.4.2 Planejamento

O objetivo da fase de Planejamento é definir a menor data e o maior conjunto histórias que serão realizadas na primeira versão. Esta definição é feita de acordo com estimativas entre cliente e programadores (BECK, 2000).

Assim que as histórias são coletadas, o Jogo de Planejamento é conduzido. Como apresentado anteriormente, o Jogo de Planejamento é a melhor maneira de executar esta fase. O cliente decide quais histórias são vitais e que devem fazer parte da primeira versão. Desta forma, pode-se elaborar uma lista priorizada das histórias. Se houver uma boa preparação durante a fase de exploração é necessário apenas alguns dias para a fase de planejamento.

Wake (2002), apresenta alguns passos, que também podem ser visualizados na figura 12, para auxiliar a fase de *release* no jogo de planejamento:

- o cliente seleciona as histórias por valor: alto, médio ou baixo;
- os programadores qualificam as histórias por risco (opcional): alto, médio ou baixo;
- os programadores declaram a velocidade: calculada sobre a estimativa realizada sobre as histórias dos clientes. A velocidade é empiricamente determinada, ou seja, baseada na experiência dos programadores;

- clientes escolhem o escopo: escolhem as histórias para a próxima versão.

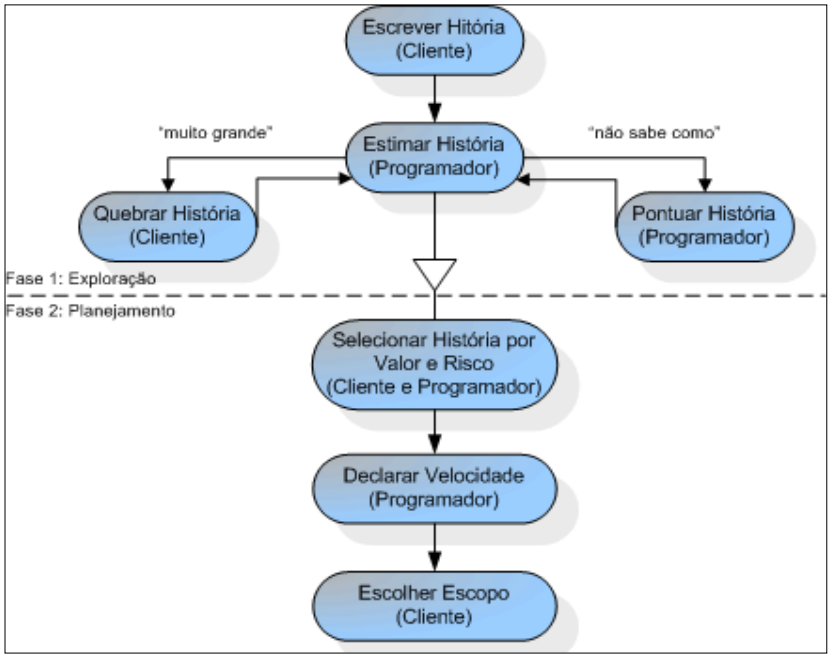


Figura 12: Jogo de Planejamento: *Release* (adaptado de WAKE, 2002)

Wake (2002) ressalta que as histórias dos clientes são quebradas em pequenas tarefas (*task cards*) e, definidos quais os programadores que irão trabalhar em cada tarefa. No primeiro dia de cada iteração, o time decide quais as histórias que serão trabalhadas. Os programadores selecionam as tarefas, estimam (pontuam em dias) e aceitam as tarefas que irão programar. A figura 13 mostra como funciona a fase de iteração no Jogo de Planejamento.

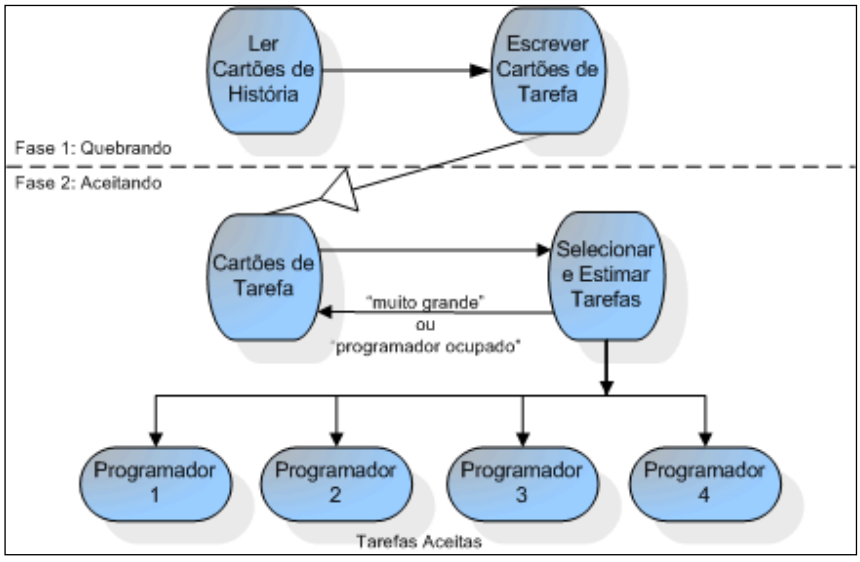


Figura 13: Jogo de Planejamento: Iteração (adaptado de WAKE, 2002)

3.4.3 Iteração para primeira versão

Segundo Beck (2000), os compromissos são divididos para serem executados em iterações que duram de uma a quatro semanas. Para cada história executada naquela iteração é produzido um conjunto de testes funcionais.

A primeira iteração, mostra como a arquitetura do sistema irá se comportar. Então, as histórias devem ser escolhidas de forma que representem força para criar todo o sistema. A pergunta chave para ser trabalhada nesta fase é: Qual a coisa de maior valor para o time para ser trabalhada nesta iteração? (BECK, 2000).

Uma importante característica do XP (WELL, 2001) é somente implementar uma funcionalidade que esteja marcada para a interação corrente. Pois, sempre existirá tempo para implementar uma funcionalidade em uma próxima iteração, ou seja, quando a história do cliente for selecionada como a mais importante na versão de planejamento.

Desta forma, o prazo final de cada interação será seriamente respeitado. Consegue-se analisar qual a real velocidade do projeto durante aquela interação. Então, o conjunto de histórias que farão parte da nova iteração estará mais bem estimado.

Conforme Reis (2000), uma seqüência de ciclos iterativos conduzem o desenvolvimento, que concentra o projeto, a codificação, os testes e as versões do produto. Normalmente, há alguns ciclos iterativos antes da fase de produção.

No final de cada iteração o cliente terá completado a execução de todos os testes funcionais e, no final da última iteração, o sistema estará pronto para entrar em produção.

3.4.4 Produção

A produção do sistema pode iniciar quando o ciclo de *feedback* é encerrado, ou seja, no final de uma versão, como representado anteriormente na figura 7.

De acordo com Beck (2000), existem alguns processos para avaliar se o *software* realmente está pronto para entrar em produção. Pode-se implementar novos testes

para provar se o sistema está estável o suficiente para entrar em produção. Testes são freqüentemente aplicados nesta fase.

Pode ser necessário apenas ajustar o desempenho. E o melhor momento para realizar estes ajustes seria no final da versão. Pois, se tem mais conhecimento do projeto, assim como existe a possibilidade de realizar estimativas reais de sobrecarga de produção do sistema. E provavelmente, este teste poderá ser realizado na máquina de produção.

3.4.5 Manutenção

A fase de manutenção é o estado normal de um projeto XP. Deve-se simultaneamente produzir novas funcionalidades, manter o sistema existente rodando, substituir membros do time que partem e incorporar novos membros ao time (BECK, 2000).

Nesta fase, pode-se tentar fazer *refactorings* maiores, os quais, nas versões anteriores, causaram certo receio. Pode-se testar novas tecnologias que se pretende incorporar nas próximas versões, ou migrar a tecnologia que está em uso para versões mais atualizadas. O cliente pode escrever novas histórias que tragam para o seu negócio grandes conquistas.

Quando o sistema está em produção, é provável que a velocidade de desenvolvimento mude. Enquanto se está explorando, se pode mensurar o real efeito que produz o tempo gasto com suporte sobre as atividades de desenvolvimento. Beck (2000) acredita que há um aumento proporcional em 50% do tempo requerido antes da produção (de 2 dias requeridos para 3 dias). Entretanto, não se deve adivinhar, mas estimar.

A estrutura do time provavelmente será alterada, ou seja, voltada para a produção. Talvez seja necessário um *help-desk*, e já não se tenha a necessidade de um grande quantidade de programadores. Entretanto, se deve ter cuidado com a rotação da posição de todos os programadores, pois há coisas que se aprende dando suporte a produção e que não se aprende de outra forma. O suporte pode ser tão interessante quanto o desenvolvimento. Então, para diminuir riscos, o time pode ser mudado gradualmente. Isto é importante, tanto para comunicar a estrutura de

todo o projeto, quanto os detalhes de planejamento e implementação, e que somente pode ser feito através do contato diário entre o time.

3.4.6 Fim do Projeto

Beck (2000) considera que “Morrer bem é tão importante quanto viver bem. Isto é uma verdade para o XP como para as pessoas”.

Quando não mais existir novas histórias, é o momento de finalizar o projeto. É o momento de escrever algumas páginas (de 5 a 10 páginas) sobre a funcionalidade do sistema, um documento que auxilie no futuro a saber como realizar alguma alteração no sistema. Uma boa razão para finalizar o projeto é o cliente estar satisfeito com o sistema e não ter mais nada que consiga prever para o futuro.

Toda a equipe que trabalhou no sistema deve ser reunida para reavaliação. Aproveite a oportunidade de analisar o que pode ter causado queda no sistema e o que fez o projeto avançar. Assim, o time saberá melhor o que fazer no futuro, o time executará tarefas de formas diferentes da próxima vez.

3.5 Papéis do Time

Certamente, alguns papéis são fundamentais e precisam existir em um time XP como: o programador, o cliente, o treinador e o supervisor (BECK, 2000). Outros papéis podem combinar responsabilidades na mesma pessoa. Um exemplo claro é o caso do gerente e do supervisor (WUESTEFELD, 2001).

O **programador** é o coração do XP, responsável por:

- estimar prazos das histórias do cliente;
- definir os cartões de tarefas a partir dos cartões de histórias;
- estimar os cartões de tarefa;
- implementar testes unitários. Deve ser feito antes do código;
- implementar o código de produção;
- trabalhar em par;
- fazer *refactoring* sempre que necessário;

- solicitar ao cliente que esclareça ou divida uma história quando necessário.

O **cliente** é outro papel essencial do *Extreme Programming.* Enquanto o programador sabe como programar, o cliente sabe o que deve ser programado. O cliente é quem paga pelo desenvolvimento do projeto e também precisa estar disposto a aprender. Pois, não é fácil executar funções como:

- definir os requisitos do software;
- escrever os cartões de história;
- definir as prioridades para os cartões de história;
- validar e definir os testes funcionais sempre que solicitado;
- esclarecer dúvidas sempre que solicitado.

O **testador** em XP tem o papel realmente focado no cliente. É a pessoa do time que aplicar os testes. Tem por responsabilidade:

- definir com o cliente os testes funcionais do projeto;
- escreve os testes funcionais;
- executa os testes e publica os resultados para o time.

O **supervisor** (*tracker*[4]) é a consciência do time, sua responsabilidade é:

- coletar sinais vitais do projeto (métricas) uma ou 2 vezes por semana;
- manter todos informados do que esta acontecendo;
- tomar atitudes sempre que as coisas parecerem ir mal.

O **treinador** é responsável pelo processo como um todo. Notifica as pessoas quando elas estão se desviando do processo e conduz o time novamente para o processo. O treinador tem a função de:

- garantir que o projeto permaneça extremo;
- ajudar com o que for necessário;

[4] A tradução literal seria “batedor” ou “aquele que segue uma trilha”, mas no contexto deste trabalho se optou por usar “supervisor”.

- manter a visão do projeto;
- formular e comunicar uma tarefa que um programador pode querer trabalhar.

O **gerente** é a pessoa que precisa transmitir coragem, confiança e saber cobrar o que é de responsabilidade de cada um. O gerente tem vários tipos de trabalho como:

- gerenciar o time e os problemas do time;
- agendar as reuniões de planejamento;
- garantir que as reuniões fluam como planejado;
- escrever o que foi definido nas reuniões;
- manter o supervisor (*tracker*) informado dos acontecimentos das reuniões;
- buscar recursos.

3.6 Considerações Finais

Nesta seção foi apresentado o XP, que é um processo orientado por princípios e práticas que guiam um projeto e sua equipe de desenvolvimento. O projeto deve ser ágil, incremental e aperfeiçoado com *refactoring*. Desta forma, o XP defende que não se deve criar um grande volume de documentos ou diagramas que podem ficar desatualizados. Uma vez que, o objetivo é ganhar tempo para ir mais rápido. Astels (2002) enfatiza que a natureza cooperativa do XP e a sua orientação a resultados garantem a longevidade. O próximo capítulo apresenta o modelo de processo ICONIX. Ele procura mostra as principais tarefas e marcos do processo, destacando os alertas do mesmo. Aborda também os modelos utilizados exemplificando-os.

4 ICONIX

4.1 Introdução

O ICONIX é um processo simplificado que unifica conjuntos de métodos de orientação a objetos em uma abordagem completa, com o objetivo de dar cobertura ao ciclo de vida. Foi elaborado por Doug Rosenberg e Kendall Scott a partir da síntese do processo unificado pelos “três amigos” - Booch, Rumbaugh, e Jacobson o qual tem dado suporte e conhecimento a metodologia ICONIX desde 1993 (Rosenberg & Scott, 1999). Ver representação na figura 14.

Silva & Videira (2001), apresentam o ICONIX como uma metodologia prática, intermediária entre a complexidade do RUP (*Rational Unified Process*) e a simplicidade do XP (*Extreme Programming*). O ICONIX está adaptado ao padrão da UML (OMG®, 2001), é dirigido por casos de uso e o seu processo é iterativo e incremental.

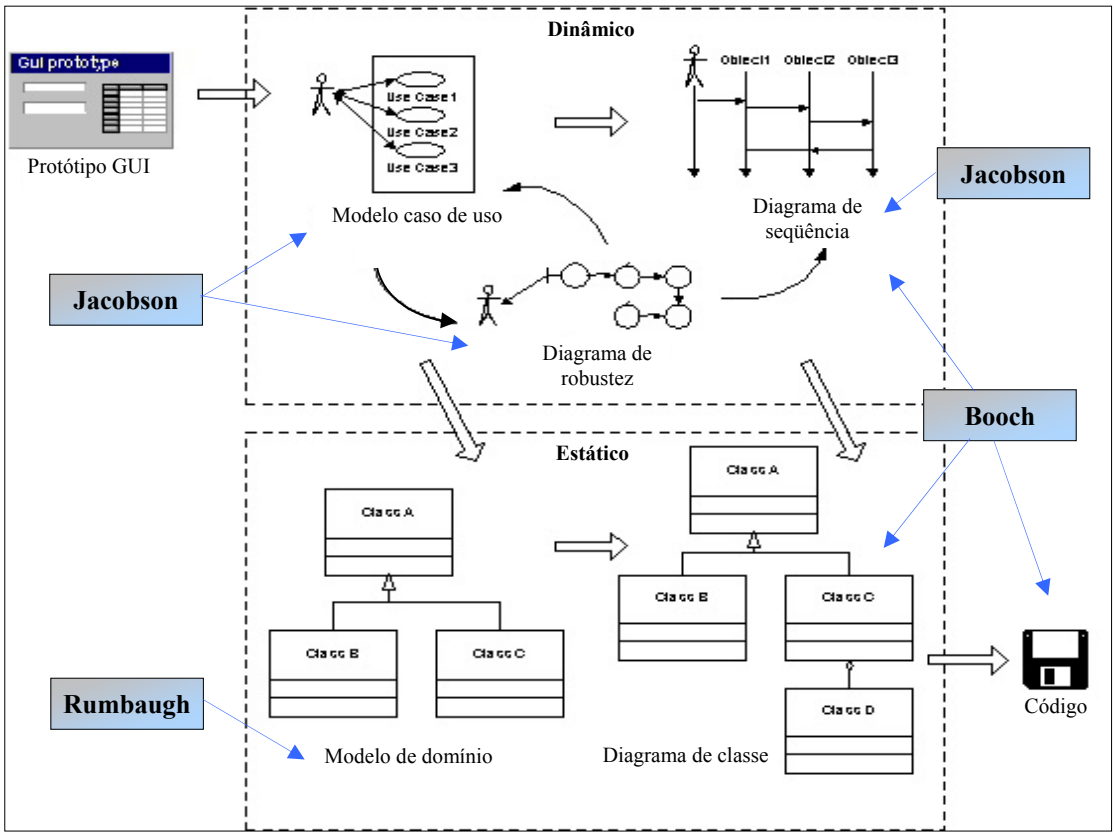


Figura 14: Processo ICONIX, mostrando a contribuição dos “três amigos”

De acordo com Rosenberg & Scott (1999), o ICONIX tem como base responder algumas questões fundamentais sobre o software. Desta forma, utiliza técnicas da UML (OMG®, 2001) que auxiliam a prover a melhor resposta. As questões e as técnicas são:

1. Quem são os usuários do sistema (ou atores), e o que eles estão tentando fazer? Utilizar **casos de uso**;
2. O que são, no "mundo real" (chamado domínio de problema), os objetos e as associações entre eles? Utilizar **diagrama de classe** de alto nível;
3. Que objetos são necessários para cada caso de uso? Utilizar **análise de robustez**;
4. Como objetos estão colaborando e interagindo dentro de cada caso de uso? Utilizar **diagrama de seqüência e de colaboração**;
5. Como será manipulado em tempo-real aspectos de controle? Utilizar **diagrama de estado**;
6. Como realmente será construído o sistema em um nível prático? Utilizar **diagrama de classe** de baixo nível.

Borillo (2000), destaca três características fundamentais no ICONIX:

- **Iterativo e incremental**: várias iterações ocorrem entre o desenvolvimento do modelo de domínio e a identificação dos casos de uso. O modelo estático é incrementalmente refinado pelo modelo dinâmico (ver figura 14);
- **Rastreabilidade** (*traceability*): cada passo referência para os requisitos de alguma forma. Silva e Videira (2001) definem rastreabilidade como sendo a capacidade de seguir a relação entre os diferentes artefatos produzidos. Desta forma, pode-se determinar qual o impacto que a alteração de um requisito tem em todos os artefatos restantes;
- **Aerodinâmica da UML** : a metodologia oferece o uso “aerodinâmico” da UML (OMG®, 2001) como: os diagramas de casos de uso, diagramas de seqüência e colaboração, diagramas de robustez.

4.2 Tarefas e Marcos do ICONIX

Rosenberg & Scott (1999) apresentam as seguintes tarefas principais: análise de requisitos, análise e projeto preliminar, projeto e implementação. Estas tarefas incluem a abordagem completa da metodologia ICONIX tendo marcos específicos associados, conforme será apresentado a seguir.

4.2.1 Análise de requisitos

A realização das seguintes atividades compõe a tarefa de análise de requisitos, representada também na figura 15:

- Identificar no “mundo real” os objetos e todas as relações de agregação de generalização entre eles. Utilizar para representar o diagrama de classe de alto nível definido como **modelo de domínio**;
- Apresentar, se possível, uma **prototipação** rápida de interface do sistema, ou diagramas de navegação, etc., de forma que o cliente possa compreender melhor o sistema proposto;
- Identificar os casos de uso do sistema mostrando os atores envolvidos. Utilizar para representar o **modelo de caso de uso**;
- Organizar os casos de uso em grupos, ou seja, utilizar **diagrama de pacote**;
- Associar requisitos funcionais aos casos de uso e aos objetos de domínio.

Primeiro Marco: Revisão dos requisitos

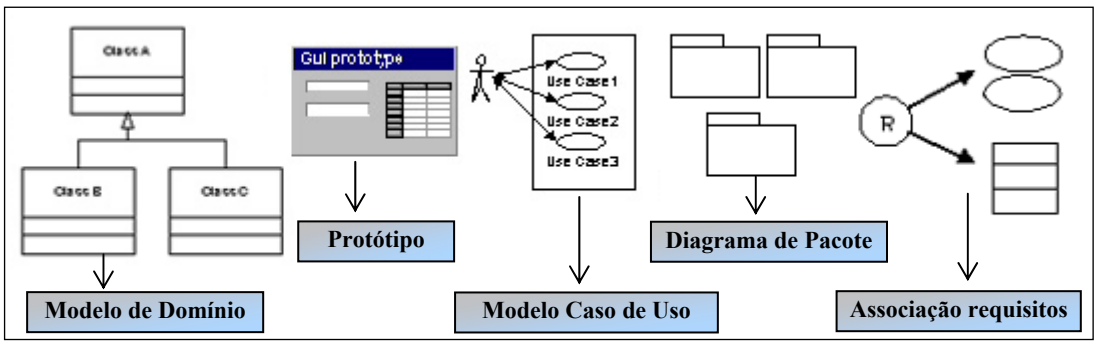


Figura 15: ICONIX - Atividades da análise de requisitos

Um importante aspecto do ICONIX é que um requisito se distingue explicitamente de um caso de uso (SILVA & VIDEIRA, 2001). Neste sentido, um caso de uso descreve um comportamento; um requisito descreve uma regra para o comportamento. Além disso, um caso de uso satisfaz um ou mais requisitos funcionais; um requisito funcional pode ser satisfeito por um ou mais casos de uso. Ainda de acordo com os autores Silva & Videira (2001), a relação entre casos de uso e requisitos é um assunto em discussão na comunidade de OO (Orientação a Objeto), não existindo qualquer consenso até o momento.

4.2.2 Análise e projeto preliminar

As atividades que fazem parte da tarefa de análise e projeto preliminar são (ver figura 16):

- Escrever os **casos de uso**, com fluxo principal das ações, podendo conter o fluxo alternativo e o fluxo de exceção;
- Apresentar a **análise de robustez**. Sendo que, para cada caso de uso se deve identificar um conjunto de objetos (usar os estereótipos de classes) e atualizar o diagrama de classes do modelo de domínio;
- Terminar a atualização do **diagrama de classe**.

Segundo Marco: Revisão do projeto preliminar

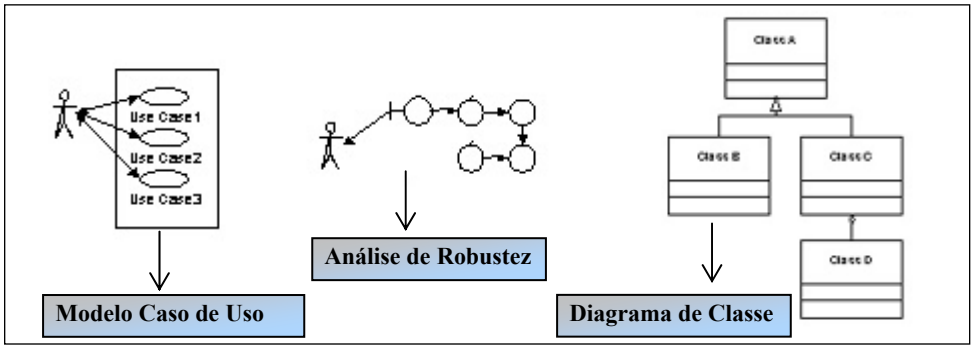


Figura 16: ICONIX - Atividades da análise e projeto preliminar

4.2.3 Projeto

A tarefa de projeto consiste na realização das seguintes atividades (ver figura 17):

- Especificar comportamento através do **diagrama de seqüência**. Para cada caso de uso, identificar as mensagens entre os diferentes objetos. Se necessário utilizar diagrama de colaboração para representar as transações chaves entre os objetos. Pode-se complementar utilizando diagrama de estado para mostrar o comportamento em tempo real;
- Terminar o modelo estático, adicionando detalhes ao projeto no **diagrama de classe;**
- Verificar com o time se o projeto satisfaz todos os requisitos identificados.

Terceiro Marco: Revisão detalhada/Crítica do projeto

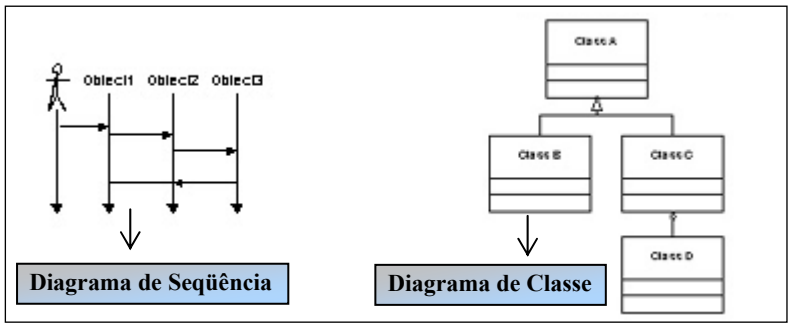


Figura 17: ICONIX - Atividades do projeto

4.2.4 Implementação

As atividades que dão suporte a tarefa de implementação são:

- Utilizar diagrama de componente, se for necessário para apoiar a fase de desenvolvimento;
- Escrever/Gerar o **código**;
- Realizar testes de unidade e de integração;
- Realizar testes de aceitação do usuário.

Quarto Marco: Entrega

A metodologia ICONIX apresenta um conjunto de “alertas” para cada tarefa realizada (SILVA & VIDEIRA, 2001). A seção seguinte apresenta os alertas do ICONIX, e nas seções posteriores são detalhadas as técnicas de modelo de domínio, modelo de use case, análise de robustez, modelo de interação (diagrama de seqüência) e finalmente endereçando requisitos.

4.3 Alertas do ICONIX

Silva & Videira (2001) e Borillo (2000) destacam os alertas do ICONIX por advertirem sobre problemas e dúvidas comuns em times de desenvolvimento de software. Os alertas estão relacionados com cada técnica, como a seguir:

- **Alertas do modelo de domínio**: (1) não perder muito tempo com verificação gramatical; (2) não adicionar multiplicidade muito cedo ao projeto; (3) endereçar agregação e composição apenas na fase do projeto detalhado; (4) não desenhar um diagrama com mais de 7-9 classes;
- **Alertas do modelo de use case**: (1) não tentar escrever casos de uso antes de saber o que os usuários realmente fazem; (2) não desperdiçar tempo construindo modelos elegantes de casos de uso que não servem para construir o projeto; (3) não perder tempo discutindo se vai usar *includes* ou *extends*; (4) não usar *templates* textuais de caso de uso longos ou complexos;
- **Alertas da análise de robustez**: (1) não tentar fazer um projeto detalhado do diagrama de robustez; (2) não perder tempo aperfeiçoando o diagrama de robustez à medida que o projeto evolui;
- **Alertas do diagrama de seqüência**: (1) não tentar alocar comportamento aos objetos antes de saber realmente o que os objetos são; (2) não iniciar o diagrama de seqüência antes de completar o diagrama de robustez associado; (3) não focar a atenção na configuração de métodos *get* e *set* em vez de focalizar a atenção em métodos reais.

4.4 Modelo de Domínio

Rosenberg & Scott (1999) definem a modelagem de domínio como sendo a tarefa de descobrir objetos (classes), que representam coisas e conceitos no “mundo real”. Na abordagem do ICONIX, a modelagem de domínio envolve palavras externas dos requisitos de dados, para construir o modelo estático.

O modelo de domínio serve como um glossário de termos, que pode ser utilizado na primeira fase para escrever os caso de uso, destaca Borillo (2000).

As regras para se conseguir um modelo de domínio são:

- substantivos e frases com substantivos se tornam **objetos** e **atributos**;
- verbos e frases com verbos se tornam **operações** e **associações**;
- frases possessivas indicam que substantivos devem ser **atributos** em vez de objetos.

Jim Rumbaugh (*apud* ROSENBERG & SCOTT, 1999) define uma classe como “uma descrição de um grupo de objetos com propriedades similares, comportamento comum, relações comum, e semântica comum”. Borillo (2002) apresenta alguns passos para auxiliar a construir o modelo estático, ou seja, através da abstração real do problema de domínio localizar as classes apropriadas, sendo:

- o primeiro passo é descobrir as classes: as melhores fontes possíveis para descobrir as classes são os requisitos de baixo nível, declaração do problema de alto nível, e conhecimento de perito;
- o segundo passo é eliminar da lista de classes candidatas os itens que não são necessários;
- o terceiro passo é construir a relação de generalização: a generalização é a relação em que uma classe é o refinamento de outra classe. Nesta relação à antiga classe é chamada de superclasse e a última classe é chamada de subclasse;
- o quarto passo é construir associação entre as classes: a associação é uma relação estática entre duas classes. Mostra a dependência entre as classes, mas não as ações.

Os dez erros de mais freqüentes na modelagem de domínio, destacados por Rosenberg & Scott (1999), são:

1. Atribuir multiplicidade nas associações logo no início da modelagem;
2. Fazer análise de verbo e substantivo exaustivamente;
3. Atribuir operações para classes sem explorar casos de uso e diagramas de seqüência;
4. Aperfeiçoar o código para reusabilidade antes de satisfazer os requisitos;
5. Debater sobre usar agregação (compartilhada) ou composição para cada parte de associação;
6. Presumir uma estratégia de implementação específica nesta fase;
7. Usar nomes difíceis de entender para as classes;
8. Ir diretamente para construção da implementação;
9. Criar um mapeamento de “um-para-um” entre as classes do modelo de domínio e as tabelas do banco de dados;
10. Utilizar padrões de projeto de forma prematura.

Desta forma, o objetivo do modelo de domínio é realizar um primeiro levantamento das entidades que fazem parte do problema. Sendo assim, os erros levantados anteriormente pretendem alertar os desenvolvedores para que eles evitem uma precipitação na busca de detalhes.

4.5 Modelo de Caso de Uso

Borillo (2000) destaca que o modelo de caso de uso é o centro conceitual do desenvolvimento, porque guia todo o processo do ICONIX. Neste sentido, Rosenberg & Scott (1999) apresentam alguns elementos chaves, onde os modelos de casos de uso são desenvolvidos em cooperação com o modelo de domínio; a análise de robustez identifica, um conjunto de objetos que satisfazem cada caso de uso; o diagrama de seqüência traça o fluxo das mensagens entre os objetos conforme especificação dos casos de uso. Além disso, o endereçamento de requisitos conecta requisitos de usuário com casos de uso e classes e finalmente, os

casos de uso servem de base para os testes de funcionalidade durante a fase de implementação.

Um caso de uso é uma seqüência de ações que um ator realiza no sistema para alcançar um objetivo (ROSENBERG & SCOTT, 1999). Um caso de uso descreve e valida o que o sistema irá fazer, serve de controle entre o usuário, cliente e desenvolvedores. De acordo com Larman (2000), casos de uso não são exatamente a especificação de requisitos ou a especificação funcional, mas ilustra e implica em requisitos nos documentos que descreve. São compostos por atores (entidades externas) e casos de uso (cenários, texto).

Ainda de acordo com LARMAN (2000), a estrutura e os cabeçalhos dos casos de uso são usualmente descritos como apresentado na figura 18. É interessante iniciar com a construção de um modelo de caso de uso de alto nível. Nesta visão, os casos de uso procuram descrever sucintamente um processo. O objetivo é obter rapidamente uma compreensão dos processos principais. Em um segundo momento, este modelo deve ser expandido para garantir uma compreensão mais profunda dos requisitos e dos processos.

Caso de uso:	**Verifica conferência**
Atores:	Professor
Tipo:	Primário
Descrição:	O professor verifica a existência de uma conferência. Valida se esta pode fazer parte de outra conferência geral. Especifica também o prazo para submissão de artigos. No término, esta conferência estará disponível para edição.

Figura 18: Exemplo textual de caso de uso de alto nível

É importante ressaltar que a UML (OMG®, 2001) não especifica uma forma padronizada para descrever casos de uso. Esta atividade deve ser definida pelo fluxo de trabalho estabelecido para o processo em uso. Entretanto, é comum utilizar seções como pré-condições (o que precisa ser verdadeiro antes do início do caso de uso) e pós-condições (o que passa a ser verdadeiro após o término do caso de uso). Além disso, os fluxos de ações são normalmente representados através de uma lista enumerada de passos a serem seguidos (ver figura 19). Este formato é adotado no trabalho de Fowler & Scott (2000), onde a descrição de um cenário é feita através de

uma seqüência de passos e de seqüências alternativas (variações do fluxo principal).

Verifica conferência
1. O professor verifica a existência de uma conferência
2. Se existir realiza manutenção dos dados
3. Senão inclui dados
4. Valida se esta conferência é composta por outra conferência principal
5. Se houver deve ser diferente da conferência em questão.
6. Confirma o sucesso da operação
7. Disponibiliza conferência para edição

Alternativa: Conferência principal não existe.
No item 4, não existir a conferência principal, permitir a inclusão desta e retornar ao item 4.

Figura 19: Exemplo textual de caso de uso

Os **atores** representam o papel de uma entidade externa ao sistema, como um usuário, o hardware, ou outro sistema que interage com o sistema modelado. É importante ressaltar que, os atores não fazem parte do sistema, mas identificam seus limites. Fowler & Scott (2000) enfatizam que, quando se falar em atores, é importante pensar nos papéis e não nas pessoas ou em cargos. Um único ator pode desempenhar vários casos de uso, e um caso de uso pode reciprocamente ter muitos atores.

Atores e casos de uso são classes. Um ator está conectado a um ou mais casos de uso através de associações, e podem possuir relacionamentos de generalização que definem um comportamento de herança. Para a visualização do diagramas de caso de uso ver a representação da figura 20.

De acordo com Silva & Videira (2001) “um **pacote** (*package*) em UML (OMG®, 2001) é um elemento meramente organizacional. Permite agregar diferentes elementos de um sistema em grupos de forma que semântica ou estruturalmente faça sentido”. Então, um grupo de casos de uso relacionado é definido como pacote de casos de uso.

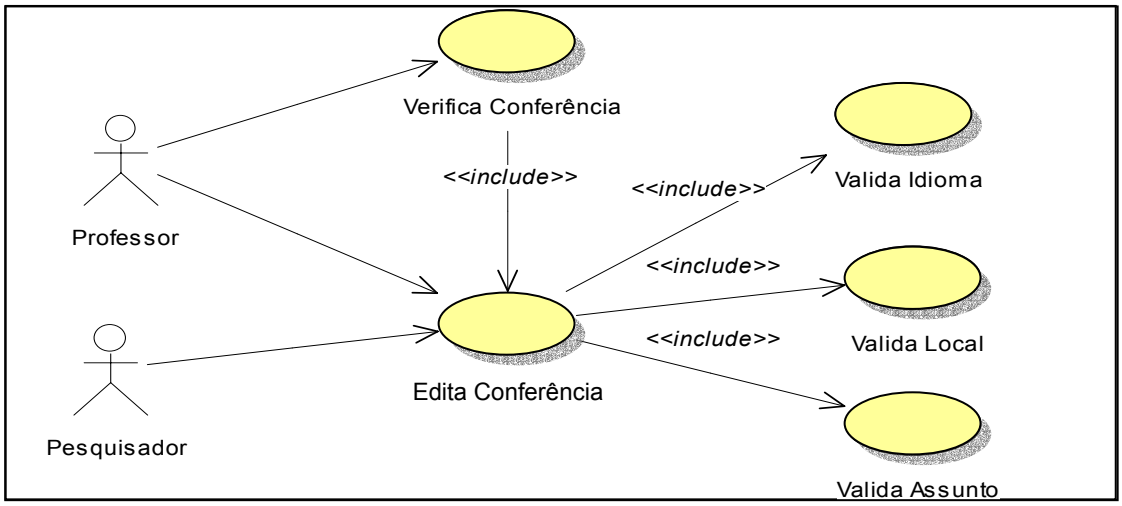


Figura 20: Diagrama de caso de uso

Pode-se aplicar algumas regras para definir qual associação utilizar entre os casos de uso:

- Usar **inclusão** (<<*include*>>) quando estiver repetindo o mesmo fluxo em dois ou mais casos de uso separados e deseja evitar a repetição. Por exemplo, vários casos de uso validam usuário (login). Então, se cria um caso de uso para "validar_usuário", onde, os casos de uso que utilizam esse procedimento, vão incluir (<<*include*>>) uma referência ao caso de uso "validar_usuário" (ver figura 20);
- Usar **generalização** quando estiver descrevendo uma variação semelhante à outra, mas que faz um pouco mais, e deseja descrevê-la sem muito controle;
- Usar **extensão** (<<*extend*>>) quando descrever uma variação em comportamento normal e deseja utilizar a forma mais controlada, explicando os pontos de extensão no *use-case* geral. Por exemplo, matrícula de pós-graduação regular e especial. O caso de uso "realiza_matrícula_especial" é uma extensão (<<*extend*>>) do caso de uso geral "realiza_matrícula".

Os dez enganos a serem evitados quando se escreve caso de uso, destacados por Rosenberg & Scott (1999), são:

1. Escrever requisitos funcionais em vez de texto de cenário de uso;
2. Descrever atributos ou métodos no lugar de uso;
3. Escrever os casos de uso de forma muito sucinta;

4. Desvincular completamente da interface com o usuário;
5. Evitar nomes explícitos para os objetos de limite;
6. Escrever usando uma perspectiva diferente do usuário, em voz passiva;
7. Descrever somente interações de usuário, ignorando respostas de sistema;
8. Omitir texto para fluxos alternativos de ação;
9. Enfocar em algo diferente do que está "dentro de" um caso de uso, como pré-condições ou pós-condições;
10. Gastar muito tempo decidindo se é melhor usar *includes* ou *extends*.

Após a conclusão dos casos de uso, pode-se iniciar a sua análise, por meio dos diagramas de robustez.

4.6 Análise de Robustez

O conceito de análise de robustez foi introduzido por Ivar Jacobson para o mundo de OO em 1991. Análise de robustez envolve analisar o texto narrativo de cada caso de uso e identificar um primeiro conjunto de possíveis objetos que participarão do caso de uso (ROSENBERG & SCOTT, 1999). Estes objetos são classificados em três estereótipos (ver figura 21):

- Objetos de **limite** (*boundary objects*): os atores usam para se comunicar com o sistema (freqüentemente referidos como objetos de **interface** ou fronteira);
- Objetos de **entidade** (*entity objects*): são normalmente objetos do modelo de domínio, geralmente mapeados em tabelas de um banco de dados;
- Objetos de **controle** (*control objects*): funcionam como integradores entre os objetos de limite e os objetos de entidade. Geralmente, são convertidos em métodos de objetos de entidade ou de objetos de limite.

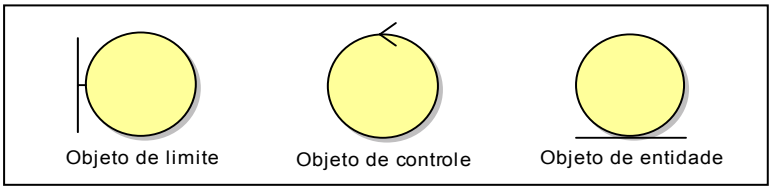


Figura 21: Símbolos do diagrama de robustez

Borillo (2000) e Silva & Videira (2001) enfatizam que a análise de robustez é o link entre a análise (o quê) e o projeto (como), tendo as seguintes regras chaves:

- **Verificar razoabilidade**: ajuda a ter certeza que o texto de caso de uso está correto e se a especificação do comportamento do sistema está razoável. Pode-se substituir os substantivos genéricos do texto do caso de uso para nomes adequados dos objetos que aparecem nos diagramas de robustez;
- **Verificar a perfeição**: ajuda a validar se todas as referências de execução do sistema estão descritas no fluxo principal e alternativo dos casos de uso;
- **Identificar objetos**: permite identificar novos objetos que não foram esquecidos durante a modelagem de domínio. Esta fase ajuda a achar discrepâncias e conflitos entre nomes. Deve-se usar os nomes do modelo de domínio para os objetos de entidade. Quando se termina a análise de robustez, deve-se ter identificado, a maioria das classes de domínio;
- **Projeto preliminar**: os diagramas de robustez são menos complexos e mais fáceis ler que diagramas de seqüência.

Uma importante sugestão do ICONIX é desenvolver os diagramas de análise de robustez antes, ou em paralelo, com a descrição textual dos casos de uso (SILVA & VIDEIRA, 2001). Desta forma, pode-se influenciar a identificação dos objetos e a escolha dos nomes usados. A finalidade é substituir os nomes genéricos por nomes dos objetos que aparecem os diagramas de robustez. Ainda de acordo com Silva e Videira (2001), outra sugestão relevante é evitar fazer desenho detalhado nesta fase e neste tipo de diagrama. Uma vez que, o objetivo fundamental é encontrar para cada caso de uso, os principais objetos e a relação de comunicação entre eles.

O ICONIX apresenta dois princípios para guiar a análise de robustez que são: (1) deve-se trabalhar com um número pequeno (entre dois e cinco) de controles por caso de uso. Se existir mais de 10 controles por caso de uso, então, deve-se dividir em casos de uso menores; (2) as setas podem seguir uma ou duas direções entre diferentes tipos de objetos, e indicam associações lógicas (BORILLO, 2000). A figura 22 mostra o que é permitido e o que não é permitido ser feito no diagrama de robustez.

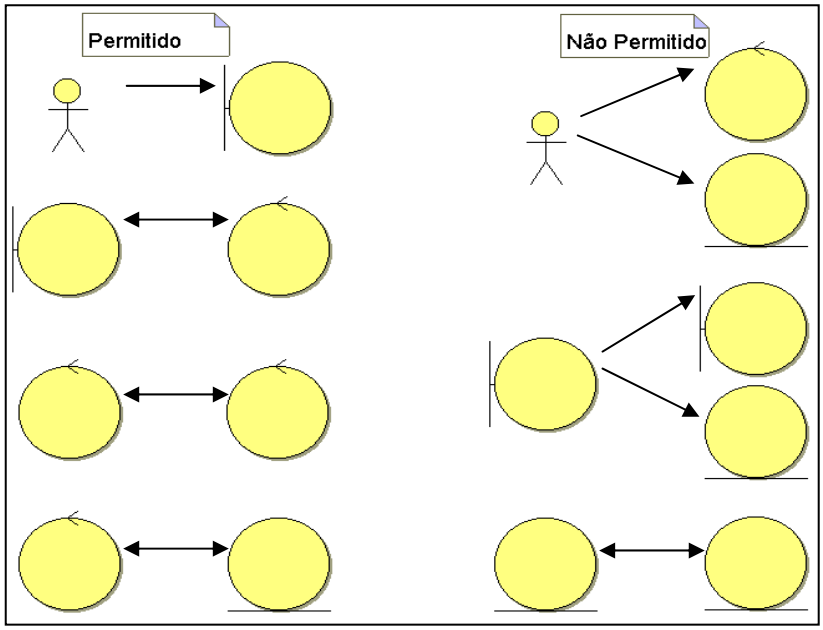


Figura 22: Regras do diagrama de robustez

A essência do diagrama de robustez pode ser capturada pelas seguintes regras:

- **atores** podem se comunicar somente com objetos de **limite**;
- objetos de **limite** podem conversar apenas com **atores** e objetos de **controle**;
- objetos de **entidade** se comunicam apenas com objetos de **controle**;
- objetos de **controle** podem conversar somente com objetos de **limite** e de **controle**.

Segundo Rosenberg & Scott (1999), o próximo passo necessário, antes de passar para a modelagem de interação (diagrama de seqüência), é atualizar o modelo de domínio (estático). De fato, é muito importante atualizar continuamente o modelo estático enquanto se trabalha com os casos de uso e durante a análise de robustez.

Os dez benefícios da análise de robustez, destacados por Rosenberg & Scott (1999) são:

1. Obriga a escrita dos casos de uso de forma concisa;
2. Força a escrita dos casos de caso na pessoa correta;
3. Provêem mecanismos de checagem dos casos de uso;

4. Ajuda a definir regras de sintaxe do tipo “O ator interage somente com objetos do tipo limite”, para os casos de uso;
5. A realização dos diagramas de robustez é mais fácil de entender e mais rápido de desenhar do que os diagramas de seqüência;
6. Permite o esboço de um *framework* para GUI-Lógica-Repositório para sistemas cliente/servidor;
7. Permite o mapeamento entre o que o sistema faz (casos de uso) e como o sistema irá funcionar (diagrama de seqüência);
8. Preenche a lacuna semântica entre a análise (caso de uso) e o projeto (diagrama de seqüência);
9. Permite que seja feito um repasse para identificação de reutilização de estrutura definidas na realização dos casos de uso;
10. Permite a divisão entre o paradigma de Modelo – Visão – Controle.

Uma vez terminada a modelagem de domínio e a análise de robustez, tem-se a maior parte dos objetos e definidos alguns atributos para os objetos. Então, pode-se partir para a especificação de como o software realmente irá trabalhar. Utilizando para isto o modelo de interação, que será visto na próxima seção.

4.7 Modelo de Interação

A modelagem de interação é a fase na qual se constrói as linhas de execução que unem os objetos e capacitam visualizar inicialmente, como o novo sistema apresentará comportamento útil (ROSENBERG & SCOTT, 1999).

O comportamento de um caso de uso no modelo de interação é detalhado através do diagrama de seqüência. O diagrama de seqüência mostra a colaboração dinâmica entre os vários objetos do sistema. Um importante aspecto deste diagrama é que, a partir, dele percebe-se a seqüência de mensagens enviadas entre os objetos, pois mostra a interação entre os mesmos (SILVA & VIDEIRA, 2001). Além disso, aloca comportamento nos objetos de controle, que normalmente se transformam em objetos de limite e/ou objetos de entidade e, ajuda finalizar a distribuição das operações nas classes. Então, pode-se atualizar o modelo estático e

completar os métodos utilizando os atributos identificados nas fases anteriores (BORILLO, 2000).

Rosenberg & Scott (1999) destacam quatro tipos de elementos no diagrama de seqüência (ver figura 23):

1. **Texto** do caso de uso (fluxo de ação). Copiar este item para a margem esquerda do diagrama de seqüência;
2. **Objetos** dos diagramas de robustez. Eles são representados com o nome do objeto e (opcionalmente) o nome da classe a que pertence (*objeto::classe*);
3. **Mensagens** representadas como setas entre objetos;
4. **Métodos** (implementação de operações) são mostrados como retângulos em cima das linhas pontilhadas que pertence aos objetos em que se está atribuindo os métodos.

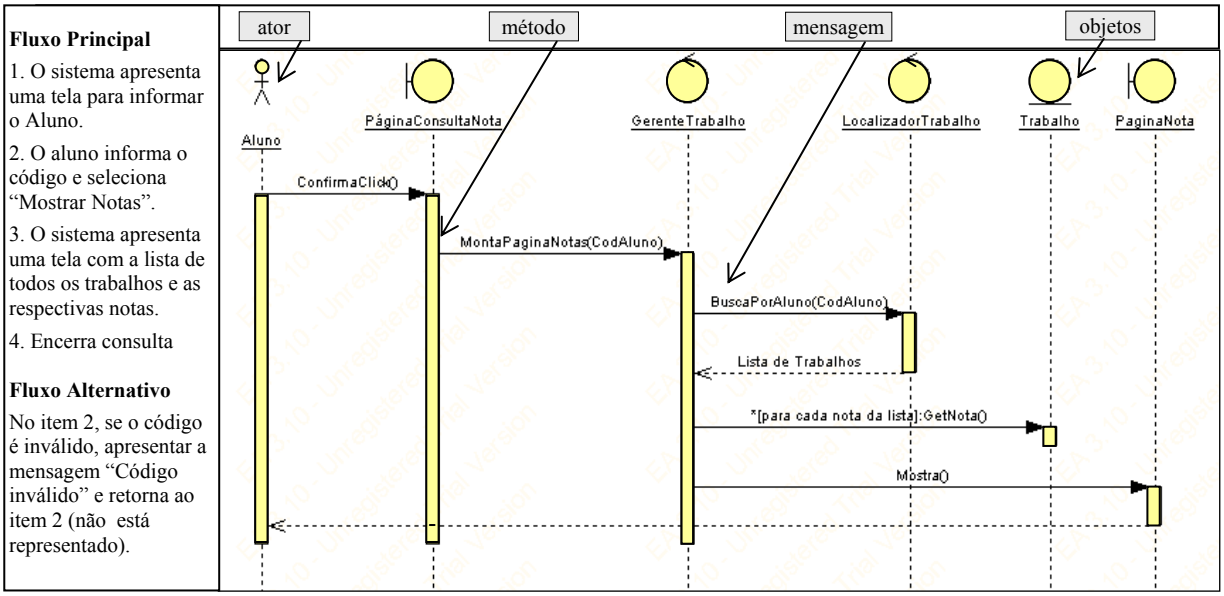


Figura 23: Elementos do diagrama de seqüência

Decidir que métodos continuam e que classes são a essência da modelagem de interação, não é tarefa fácil de realizar e exige muito esforço e experiência (ROSENBERG & SCOTT, 1999). O ICONIX apresenta algumas sugestões para realizar esta tarefa, como:

- Iniciar convertendo os objetos de controle, dos diagramas de robustez, para conjuntos de métodos e mensagens que incluam o comportamento desejado;
- Usar o diagrama de robustez como uma lista de checagem, para ter certeza que todo o comportamento requerido do sistema está no diagrama de seqüência;
- Conseguir responder a questão: “Que objetos são responsáveis por quais funções?”;
- Desenhar mensagens entre objetos é equivalente a atribuir métodos e/ou operações para os objetos.

É importante ressaltar que, no modelo de interação pode-se usar o diagrama de colaboração e diagrama de estado da UML (OMG®, 2001) em conjunto com o diagrama de seqüência (são opcionais). Estes diagramas auxiliam a modelar aspectos adicionais do comportamento dinâmico do sistema que se está projetando (ROSENBERG & SCOTT, 1999). Silva e Videira (2001) consideram que os diagramas de colaboração ajudam a ilustrar as principais transações entre objetos, em particular situações com padrões de desenho. O ICONIX dá ênfase à utilização de diagramas de estados associados a casos de uso em vez de, como é mais comum, a objetos e classes.

Os dez pontos mais importantes do modelo de interação, destacados por Rosenberg & Scott (1999) são:

1. Fazer um diagrama de seqüência através dos casos de uso;
2. Combinar o fluxo narrativo do caso de uso associado com o diagrama;
3. Construir um diagrama de seqüência único para cada caso de uso, inclusive com os fluxos alternativos;
4. As mensagens entre objetos invocam as operações nas classes;
5. Ter dificuldades em iniciar a construção do diagrama de seqüência indica que o caso de uso pode estar escrito de forma incorreta ou a análise de robustez não foi completada;

6. Explorando o comportamento dinâmico do sistema se identifica atributos e métodos dos diagramas de classe;
7. O diagrama de seqüência é o veículo primário para tomar decisões de alocação de comportamento;
8. Adicionar solução dos objetos de limite (interface) para os objetos de domínio do problema, como se explora o uso do sistema em um nível detalhado;
9. Os padrões de projeto, nesta atividade, são freqüentemente úteis;
10. Escrever o texto original, nível de requisitos, do caso de uso na margem do diagrama de seqüência provê requisitos visuais de rastreabilidade.

De acordo com Silva & Videira (2001), a última atividade para completar o modelo de interação é atualizar o modelo estático. Para tanto, deve-se adicionar informações detalhadas ao diagrama de classe, com base nos diagramas de seqüência que identificam as operações que de deverão fazer parte das classes correspondentes. Além disso, o diagrama de classe deverá incluir todos os atributos, operações, valores de visibilidade e de navegabilidade.

4.8 Endereçando Requisitos

O ICONIX reserva a abordagem da análise de requisitos por último, por duas razões principais (BORILLO, 2000):

- para evitar a possível confusão entre requisitos, casos de uso e funções (operações). O conceito de casos de uso e funções foi explanado anteriormente através da modelagem de casos de uso e da modelagem de interação. Desta forma, o ICONIX acredita que é mais fácil falar sobre requisitos e como eles contrastam com esses itens;
- o conceito de rastreabilidade se torna mais importante no momento anterior a codificação. É preciso evitar codificar um sistema que não esteja de acordo com os requisitos reais do cliente.

Um **requisito** é um critério especificado pelo usuário que o sistema precisa satisfazer (ROSENBERG & SCOTT, 1999).

Pressman (1997) define requisito como uma condição ou capacidade que o software deve possuir para atender uma necessidade do usuário, ou para resolver um problema, ou para atingir um objetivo, ou para atender as restrições da organização ou dos outros componentes do sistema.

Segundo Rosenberg & Scott (1999), os requisitos são normalmente expressos em sentenças que incluem as palavras *deve ter* ou *é necessário*. Existem vários tipos diferentes de requisitos. Borillo (2002) considera os seguintes requisitos como os mais usados:

- os **requisitos funcionais**: que representam a funcionalidade que o sistema deve ter. Podem ser especificados, por exemplo, na forma de *pré-condição* (estado), *ação* e *pós-condição* (resultado);
- os **requisitos não-funcionais**: que representam as qualidades do produto. Podem incluir requisitos de desempenho, de capacidade, de teste e de segurança. Geralmente associados a *critérios* que servem como parâmetro para quantificar o requisito;
- as **restrições**: que fixam os limites do projeto e do sistema. Normalmente associadas a uma *razão* por que uma restrição está limitando o sistema.

Ainda de acordo com Borillo (2000), o modo que o ICONIX aborda os requisitos em relação aos casos de uso e funções é a seguinte: um caso de uso descreve uma unidade de comportamento para um sistema; os requisitos descrevem as leis que governam o comportamento; as funções são as ações individuais que acontecem dentro do comportamento. Neste sentido, não existe nenhuma razão por que um caso de uso não possa endereçar mais que um requisito. Também é perfeitamente aceitável que um conjunto de casos de uso satisfaçam apenas um requisito.

Na abordagem do ICONIX, os requisitos do usuário precisam ser endereçados em conjunto com cada artefato produzido na análise e projeto. Par tal, torna-se necessário:

- revisar as alocações dos requisitos para os casos de uso, usando os diagramas de casos de uso original e os diagramas de robustez. Cada requisito deve ser alocado, pelo menos, para um caso de uso;

- verificar que cada requisito seja tratado, pelo menos, por uma classe no modelo estático;
- trilhar para que o nível dos requisitos descritos nos casos de uso satisfaça o projeto atual através dos diagramas de seqüência.

Os dez itens mais importantes para lembrar sobre requisitos, destacados por Rosenberg & Scott (1999) são:

1. O time deve fazer uma lista completa dos requisitos, tão cedo quanto possível, em vez de iniciar diretamente com o código;
2. Os requisitos são requisitos; os casos de uso são casos de uso. Os requisitos não são casos de uso; os casos de uso não são requisitos;
3. Existem vários tipos de requisitos, incluindo funcionais, não funcionais e restrições;
4. O time deve demonstrar conexão direta com, pelo menos, um caso de uso para cada requisito durante a revisão dos requisitos;
5. O time deve demonstrar conexão direta com, pelo menos, uma classe para cada requisito durante a revisão dos requisitos;
6. Cada caso de uso deve servir para ambas as entradas, tanto para o processo de desenvolvimento quanto para os testes de aceitação de usuário;
7. O time deve localizar a alocação de requisitos para os casos de uso e classes de domínio como parte da revisão de requisitos;
8. Quando todos os requisitos do usuário estiverem mapeados em algum lugar do projeto, o processo de implementação pode ser iniciado;
9. O projeto detalhado, como refletido nos diagramas de seqüência, deverá ser certificado com o texto de cada caso de uso como parte da revisão do projeto;
10. Deve-se ter, pelo menos, um teste para verificar cada requisito.

4.9 Considerações Finais

Nesta seção foi apresentado o ICONIX, o qual é um processo iterativo e incremental, dirigido por casos de uso e com modelagem visual baseada em UML (OMG®, 2001). Silva & Videira (2001) destacam que o ICONIX, por meio de uma abordagem essencialmente prática, ensina a modelar um sistema de software segundo o paradigma OO. Ainda de acordo com Silva & Videira (2001), o objetivo global do ICONIX é, a partir de um conjunto de requisitos inicialmente definido, a construção do modelo de classes que suporte a implementação de determinado sistema. O próximo capítulo apresenta os procedimentos metodológicos utilizados para a realização do trabalho. É apresentado anda, o ambiente da aplicação e como foi realizada a coleta das informações por meio da aplicação do XP e da aplicação do ICONIX.

5 PROCEDIMENTOS METODOLÓGICOS

Neste capítulo, será apresentada a metodologia aplicada no trabalho. A pesquisa de campo será realizada na forma de observação direta intensiva, de forma exploratória (GIL *apud* SILVA, 2001) para levantar e analisar dados quantitativos e qualitativos sobre os processos XP e ICONIX. A intenção é obter, dos líderes e participantes das comunidades de desenvolvimento, os aspectos particulares de cada processo, e correlacionar estes dados para sintetizar um conjunto comparativo entre os dois modelos.

Para realizar o levantamento, foram definidos dois projetos e serem estudados. O modelo XP será analisado no projeto “Gratificação de Incentivo a Docência (GID)” do Centro Federal de Educação Tecnológica de Santa Catarina (CEFET/SC); o modelo ICONIX será analisado no projeto “Prontuário Médico e Odontológico (InfoSaúde)” da Secretaria Municipal da Saúde de Florianópolis.

Além do levantamento de informações, foi elaborado um questionário, o qual foi aplicado individualmente nos especialistas de domínio (clientes) e nos analistas de desenvolvimento (técnicos e alunos). Este questionário indicou as preferências, facilidades e dificuldades encontradas em cada processo.

Para a execução do trabalho, os processos foram aplicados paralelamente em ambos os projetos. Desta forma, acreditou-se conseguir uma forma de comparação mais precisa.

5.1 Ambiente da Aplicação e Escalonamento dos Processos

O ambiente da aplicação, o escalonamento dos processos e, a infra-estrutura tecnológica e de pessoal foram determinados conforme a disponibilidade e as necessidades do cliente. Também foi realizado remanejamento de pessoal para que fosse possível atender os requisitos necessários em cada modelo proposto.

5.1.1 Infra-estrutura e tecnologia de desenvolvimento

Para a aplicação do XP foi adotada a seguinte estrutura:

- banco de dados o Oracle9i;
- ferramenta de programação o Oracle Forms Builder e PL/SQL;
- ferramentas de modelagem o ERwin 4.0 e o Microsoft Visio 2000;
- ferramenta de teste de aceitação o Microsoft Excel;
- time composto por 4 membros.

Para a aplicação do ICONIX foi adotada a seguinte estrutura:

- banco de dados o SQL Server 2000;
- linguagem de programação o Borland Delphi 6;
- ferramentas de modelagem o ERwin 4.0 e o Enterprise Architect (EA);
- time composto por 5 membros.

Embora o EA permita realizar o modelo de dados (entidade-relacionamento) através de uma extensão do modelo de classe, foi utilizado o ERwin 4.0 por ser uma ferramenta mais completa e devido ao time ter experiência com ela.

5.1.2 Escalonamento dos processos X sistemas

A escolha da utilização do processo XP no sistema GID se deu pelo fato que, o XP se destina a projetos nos quais o cliente não necessita de uma documentação formal, o time envolvido deve ser pequeno e o prazo de execução dura poucos meses. Além disso, o XP trabalha em ambientes dinâmicos, ou seja, os requisitos podem sofrer freqüentes mudanças (ZABEU, 2002) (WELLS, 2001).

Por sua vez, a escolha da utilização do processo ICONIX no sistema InfoSaúde se deu pelo fato que, a necessidade de uma documentação de requisitos e de uma modelagem formal era fundamental para a execução do projeto. É importante destacar que, os requisitos e os casos de uso foram elaborados de forma que o especialista de domínio pudesse compreender e validar a funcionalidade dos mesmos.

Nas seções seguintes será apresentado como foi aplicado o processo XP e o processo ICONIX.

5.2 Aplicação do XP

O trabalho foi iniciado com a fase de exploração, conceituada anteriormente na seção 3.4.1, cujo objetivo foi compreender o que o sistema precisava fazer, bem o suficiente para que pudesse ser estimado (WAKE, 2002). Essa estimativa foi preliminar, e partiu das primeiras histórias escritas pelo cliente. Em paralelo as histórias dos clientes, foram avaliadas as diferentes tecnologias e exploradas as possibilidades para a arquitetura do sistema. Nesta etapa, foi utilizado também, o diagrama de atividades da UML (OMG®, 2001).

Durante a fase de planejamento, foi especificado o projeto, as iterações e o dia-a-dia. Foi elaborada uma estimativa com base nos cartões de história, na metáfora e em uma solução simples (ASTELS et al., 2002). O objetivo desta fase foi estimar o menor tempo e o maior número de histórias para a primeira versão (BECK, 2000). O projeto também poderia ser replanejado se uma alteração significativa, repassada pelo cliente ou pelos desenvolvedores, fosse identificada.

5.2.1 Composição e tarefas do time

Todos os contribuintes do projeto sentavam-se juntos como membros de um time (JEFFRIES, 2001). O time incluiu um especialista de domínio - “o cliente” - que definia os requisitos, fixava as prioridades e guiava o projeto. O time possuía uma dupla de programadores, os quais também absorveram a tarefa de ajudar o cliente a definir os testes de aceitação e os requisitos. Além destes, existia um gerente que provia recursos, mantinha a comunicação externa e coordenar as atividades. Nenhum destes papéis foi necessariamente de propriedade exclusiva de um só indivíduo. Todos os membros do time contribuíram de todas as formas que puderam, conforme suas capacidades. Jeffries (2001) destaca que os melhores times não têm nenhum especialista, mas contribuintes gerais com habilidades especiais.

5.2.2 Descrição do sistema

O aplicativo foi construído para calcular a Gratificação de Incentivo a Docência (GID) e teve como base o regulamento elaborado pelo Comitê de Avaliação Docente

do CEFET/SC. Este regulamento estabelece os critérios e procedimentos de avaliação e desempenho docente para a implementação da GID.

O sistema mantém uma lista de todos os docentes de carreira de 1º e 2º Graus que ministram aulas nos cursos regulares do CEFET/SC ou que desempenham outras atividades previstas no regulamento. Uma lista simples de atividades também é mantida.

Para efeito de pontuação, os professores são classificados em grupos. Conforme o grupo, existe um critério para avaliação, ou seja, uma seqüência de procedimentos e cálculos que o sistema executa. O centro do sistema é o gerenciamento dos pontos obtidos pelo professor, que é proporcional ao seu desempenho docente. A principal finalidade do sistema foi gerar informações para os recursos humanos.

Neste momento, identificou-se a necessidade de um diagrama que mostrasse de forma genérica o fluxo do negócio. Neste sentido, foi escolhido o diagrama de atividades que ilustra o fluxo de atividades, representado na figura 24. Torna-se importante ressaltar que este diagrama é parte da especificação UML (OMG®, 2001).

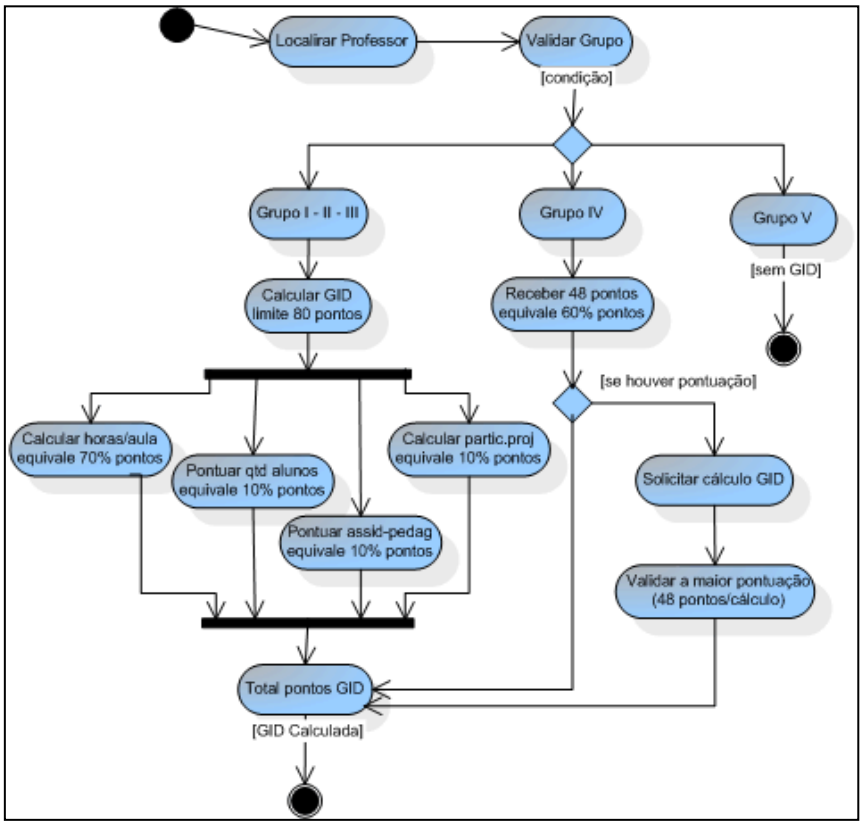


Figura 24: Diagrama de atividades da GID

5.2.3 Metáfora

A metáfora foi usada para superar os problemas de conceituação e comunicação inicial com o cliente (ASTELS, 2002). Quando se começou a pensar sobre o sistema, se achou que ele era direto o suficiente para se adotar simplesmente uma metáfora ingênua, um sistema de nomes baseado no domínio. Entretanto, o cliente em determinado momento, da fase de exploração, comentou:

“Pensei inicialmente em utilizar uma planilha do Excel para realizar os cálculos. Pois, a entrada dos dados é facilitada e consigo visualizar os critérios de avaliação e, rapidamente tenho o resultado”.

Então, este comentário passou a ser a metáfora para o sistema da GID. A principal vantagem da metáfora foi que o time conseguiu, rapidamente, pensar em uma interface para o usuário. Além disso, o formulário para coletar as informações nas gerências dos cursos foi exibido em formato tabular (ver figura 25).

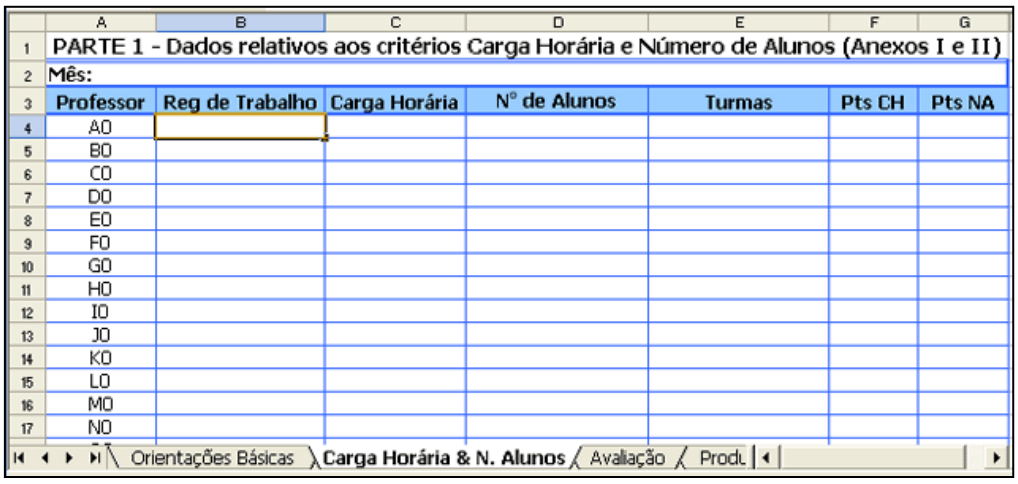


Figura 25: Planilha para coleta e simulação da GID

5.2.4 Histórias do usuário

Nesta seção, estão relacionados os cartões de história elaborados pelo cliente. Todas as histórias de interesse foram anotadas e discutidas pelo time. Inicialmente, houve uma pequena dificuldade do cliente em elaborar uma história. Entretanto, não existiu resistência. Partiu-se com as histórias centrais e, posteriormente, adicionou-se algum detalhe.

É importante ressaltar que um cartão de história é apenas um lembrete de uma conversação com o cliente (ASTELS, 2002). Uma história não contém todos os detalhes que são necessários para codificar o comportamento. Para isto, foram realizadas conversas diretas com o cliente.

A primeira história, representada pela figura 26, está em seu formato original. As demais histórias, representadas pela figura 27, estão em formato resumido para melhor organização deste trabalho.

CARTÃO DE HISTÓRIA E TAREFA

Data:	23/04/2002	Tipo de Atividade	Nova: X	Dificuldade	Valor:
Número da História	001 – Calcular GID			Prioridade do Usuário:	
Referência Anterior:		Risco:		Estimativa do Técnico:	
Descrição da Tarefa:	Calcular a GID com base no grupo que o professor está. O grupo é especificado pelo regime de trabalho e número de horas/aula do professor, num total de 5 grupos. Sendo que para os grupos I-II-II calcular a GID, para o grupo IV dar 48 pontos (calcular somente se o professor solicitar) e o grupo V não recebem GID.				
Notas:	Os professores dos grupos I-II-III são atribuídos um total de pontos multiplicando-se 80 pontos vezes o número de professores de cada grupo.				

Acompanhamento da Tarefa:

Data	Estado	Para Realizar	Comentário
23/04	Questionar	OK	Quais informações guardar mensal/semestral
30/04		OK	Guardar mensal: horas/aula e semestral: avaliação quantitativa e participação projetos

Figura 26: Cartão de história e tarefas

Nº história	Descrição
002	Coletar os dados dos professores através de formulários que estarão disponíveis para cada gerência. Estes dados servem de base para o cálculo da GID. Os dados são referentes à carga horária, número de alunos, avaliação quantitativa, e participação em projetos.
003	Considerar no cálculo da GID: - se o professor se afastar: terá como gratificação à pontuação obtida no período anterior. - se não houver pontuação anterior, ou se o afastamento for superior ao de avaliação, a GID será calculada com base em 60% do limite máximo de pontos . - nos meses de férias será considerada média dos últimos 12 meses. afastamento por doença e/ou invalidez, será calculado pela média dos últimos 12 meses.
004	Gerar o cálculo com informações dos últimos 6 meses para as gerências, com possibilidade de alterar os resultados no período de 21 dias (7 dias recurso / 14 dias julgamento). Relatório final deverá ser entregue a GDRH – Gerencia de Recursos Humanos que irá vigorar no semestre subseqüente ao cálculo.
005	Imprimir o resultado do total dos pontos e a conversão em reais. Imprimir dados do professor para conferência.

Figura 27: Cartão de história e tarefas resumido

5.2.5 Estimativa, priorização e planejamento

Assim que as histórias foram coletadas, o cliente as classificou por prioridade: alta, média, baixa. As histórias sinalizadas como “alta” eram imediatamente necessárias, o sistema não teria validade sem elas. As histórias marcadas como “média” eram requeridas, mas a sua ausência poderia ser contornada por algum tempo. As histórias sinalizadas como “baixa” seriam interessantes, mas apenas após a conclusão das outras histórias.

Posteriormente, o time de programação estimou as histórias pontuando em semanas, se uma história consumisse mais de três semanas o programador retornava a história para que o cliente a dividisse em histórias menores. Definidas as estimativas e prioridades, foram designadas as iterações. Decidiu-se por duas iterações (ver Tabela 1).

É importante observar que os cartões de história formaram a funcionalidade básica do aplicativo. Desta forma, optou-se por constituir a primeira versão do sistema. Provavelmente, outras versões serão necessárias para aperfeiçoamento e complemento de recursos.

Tabela 1: Dados dos cartões de história

História	Prioridade	Estimativa	Iteração
001 - Calcular GID	alta	2 semanas	1º
002 - Coletar dados	alta	1 semana	1º
003 – Exceções do cálculo	média	1 semana	1º
004 – Gerar pontos	média	2 semanas	1º
005 – Imprimir resultados	média	1 semana	2º

Logo após, a realização das prováveis estimativas, as histórias foram divididas em tarefas. Com o objetivo de maximizar o trabalho, foi elaborada uma lista simplificada de tarefas para os cartões de história (ver Tabela 2). Esta decisão foi tomada para facilitar o trabalho do time, uma vez que, existia apenas um par de programadores.

Tabela 2: Divisão das histórias em tarefas

História	Tarefa (s)	Realizada
001 - Calcular GID	Criar e manipular professores	OK
	Criar e manipular projetos	OK
	Criar e manipular grupos	OK
	Manipular dados mensais de carga horária e alunos	OK
	Manipular dados semestrais avaliação quantitativa	OK
	Manipular dados semestrais projetos de pesquisa	OK
	Mapear as fórmulas do cálculo	OK
002 - Coletar dados	Elaborar formulário para coleta dos dados	OK
	Gerenciar lista de informações atualizadas	OK
003 – Exceções do cálculo	Guardar informações de períodos anteriores	OK
	Prever meses de férias	OK
	Guardar média dos últimos 12 meses	OK
004 – Gerar pontos	Gerar os pontos de todos os professores validando o grupo	OK
	Listar todas as informações geradas	OK
	Pesquisar resultado por professor	OK
005 – Imprimir resultados	Imprimir dados na tela e em papel	OK
	Imprimir lista de resultados: matricula – nome – pontos – valor	OK
	Imprimir dados individuais para conferência	OK

5.2.6 Teste de aceitação

Para validar cada característica desejada, o cliente realizou os testes de aceitação manualmente por meio da interface gráfica do usuário (GUI – *Graphical User Interface*). Foi assentado que a GUI era suficientemente simples para que os testes de aceitação fossem executados à mão. Além disso, um teste de simulação da GID foi construído na planilha do Microsoft Excel, cujo objetivo era mostrar que os resultados esperados estavam corretamente implementados.

Para Jeffries (2001), os melhores times XP valorizam os testes do cliente da mesma forma que consideram os testes do programador. Uma vez que os testes funcionem, o time deve manter esta versão como correta.

5.2.7 Desenvolvimento orientado por teste

O time realizou o “desenvolvimento orientado por teste”, trabalhando em ciclos muito pequenos, onde foram acrescentados os testes. Primeiramente, se escreveu o teste que especificou e validou o comportamento desejado e, em seguida, foi criado e desenvolvido o código que o implementou. Desta forma, o time produziu o código com grande parte dos testes cobertos, o que foi um grande avanço.

A construção dos testes foi iniciada com a definição de “*procedures*”. O objetivo do primeiro teste foi examinar a pontuação docente obtida através da fórmula para calcular à carga horária. O teste seguinte foi modelado para validar os pontos obtidos através da relação de responsabilidades docentes, conforme o número de alunos. Um terceiro teste foi elaborado para a avaliação quantitativa das aulas ministradas, limitando pontos por assiduidade e atribuições didático-pedagógicas. O último teste realizado para atender a história “001 – Calcular GID”, foi atribuir pontos pela participação em projetos de pesquisa utilizando a fórmula conforme o grupo do professor.

Cada teste, inclusive o código, foi desenvolvido de forma incremental, incluindo um teste de cada vez e fazendo com que ele fosse validado. Isto ajudou os programadores a terem um retorno imediato de como eles estavam procedendo. Esse padrão de desenvolvimento se repetiu em toda a aplicação. Embora, tenham sido realizados testes somente para as histórias e tarefas que o time considerou como cruciais para o sistema.

Neste contexto, Jeffries (2001) enfatiza que o retorno dos testes realizados é muito importante no desenvolvimento do sistema, pois um bom resultado exige bons testes. Desta forma, o sistema da GID foi construído.

5.2.8 Entrega da versão para produção

O sistema entrou em produção após a execução de todos os testes de aceitação realizados pelo cliente. Desta forma, um projeto pequeno e simples foi desenvolvido no modelo do XP. Parte da interface pode ser visualizada na figura 28.

É importante ressaltar que alguns requisitos surgiram à medida que o desenvolvimento progrediu. Um requisito que se pode destacar, foi à habilidade de simular o cálculo do professor com dados fictícios. Neste caso, não foi necessário alterar muito o planejamento. A nova história foi incorporada adicionando-se mais uma iteração.

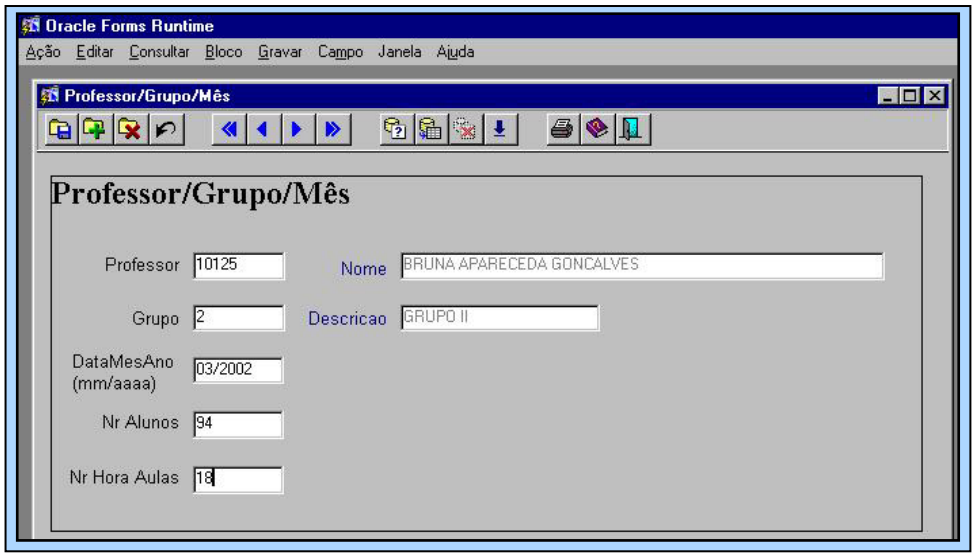


Figura 28: Tela de entrada dos dados mensais da GID

5.3 Aplicação do ICONIX

O trabalho foi iniciado com um levantamento informal de todos os requisitos que, a princípio, deveriam fazer parte do sistema InfoSaúde. Posteriormente, conforme as etapas de análise de requisitos, análise e projeto preliminar, projeto e implementação foram acontecendo, esses requisitos foram transformando-se em outros artefatos. As tarefas mais importantes contempladas nesta seção referem-se à fase de concepção, em particular as tarefas de identificação de requisitos, de análise e de desenho.

Quando se optou por utilizar o ICONIX neste projeto, estava claro que havia a necessidade de produzir alguns documentos. Inicialmente, estes documentos não pareceram muito úteis. Entretanto, logo se percebeu que estes artefatos eram essenciais para se construir o projeto dentro do melhor caminho. Borca (2000) considera que cada documento ajuda a desenvolver uma parte do projeto. O projeto InfoSaúde foi dividido em múltiplos passos. Sendo que, um artefato foi produzido como resultado de cada passo. Estes artefatos são apresentados nas seções seguintes.

5.3.1 Descrição do sistema

O sistema foi desenvolvimento para automatizar o processo de agendamento de consultas e acompanhamento do paciente através do prontuário médico e

odontológico. Além de gerar informações estatísticas e financeiras para Secretaria da Saúde. O prontuário médico mantém o histórico referente à saúde do paciente (exames, vacinas, partos, medicamentos, etc.) e o prontuário odontológico contém os dados relativos à situação dentária (odontograma) e a higiene bucal do paciente.

Integram o InfoSaúde um conjunto de profissionais, os quais pertencem a uma especialidade. Para cada especialidade é definida a duração da consulta, que automaticamente gera a estrutura da agenda com a quantidade de consultas disponíveis para o profissional. Adicionalmente, cada especialidade dá acesso a determinadas partes (fichas) do prontuário.

O InfoSaúde ainda integra várias estruturas de codificação utilizadas pelo Ministério da Saúde, como a tabela do Sistema de Informações Ambulatoriais (SIA) e a tabela do Código Internacional de Doenças (CID). Com base nessas codificações são geradas informações para o Relatório Ambulatorial de Atendimento Individual (RAAI). O RAAI resume um atendimento e possui dados sobre o tipo de consulta (1º consulta, gestantes), procedimentos realizados no paciente (curativo, pressão arterial), vacinas aplicadas e medicamentos fornecidos.

5.3.2 Atores (utilizadores)

Como foi discutido na seção 4.5, os atores são entidades externas que interagem com o sistema. Neste contexto, Silva & Videira (2001) destacam que o conjunto de todos os atores reflete todos os elementos que interagem com o sistema.

O InfoSaúde suporta os seguintes tipos de atores, que foram identificados no início do levantamento dos requisitos:

- Corpo clínico: Conjunto de profissionais registrados no sistema que possuem um identificador próprio (CRM, CRO). Estes atores têm acesso a todas as informações que fazem parte do prontuário do paciente. Podem incluir e modificar os dados. São responsáveis pelo conteúdo e sigilo das informações.
- Enfermeiro: Conjunto de profissionais cadastrados no sistema que tem um identificador próprio (CREA). Acessam pacientes que já possuem consulta agendada para realizar procedimentos de triagem (colher informações básicas

do paciente). Também acessam o prontuário para realizar consultas de enfermagem. Tem restrição de acesso ao receituário e atestados.

- Recepcionista: São usuários do sistema com acesso para manipular a agenda dos profissionais e cadastrar pacientes do posto.
- Coordenador: Atores responsáveis pela gestão e configuração das informações do sistema. Controla as operações introdução, remoção e alteração de atores, define a estrutura da agenda, tipos de exames, etc. São ainda responsáveis pela emissão de todas as informações gerenciais.

5.3.3 Análise de requisitos

5.3.3.1 Modelo de domínio

A primeira tarefa que a equipe desempenhou foi à coleta de todos os documentos utilizados no Posto de Saúde. Com base nestes documentos foi realizado uma leitura e interpretação das informações. Desta forma, se identificou todos os objetos do mundo real e todas as relações de generalização e associação possíveis. Além disso, foi realizado um conjunto restrito de entrevistas com os atores e especialistas de domínio. O resultado desta tarefa correspondeu ao diagrama de domínio (ver figura 29).

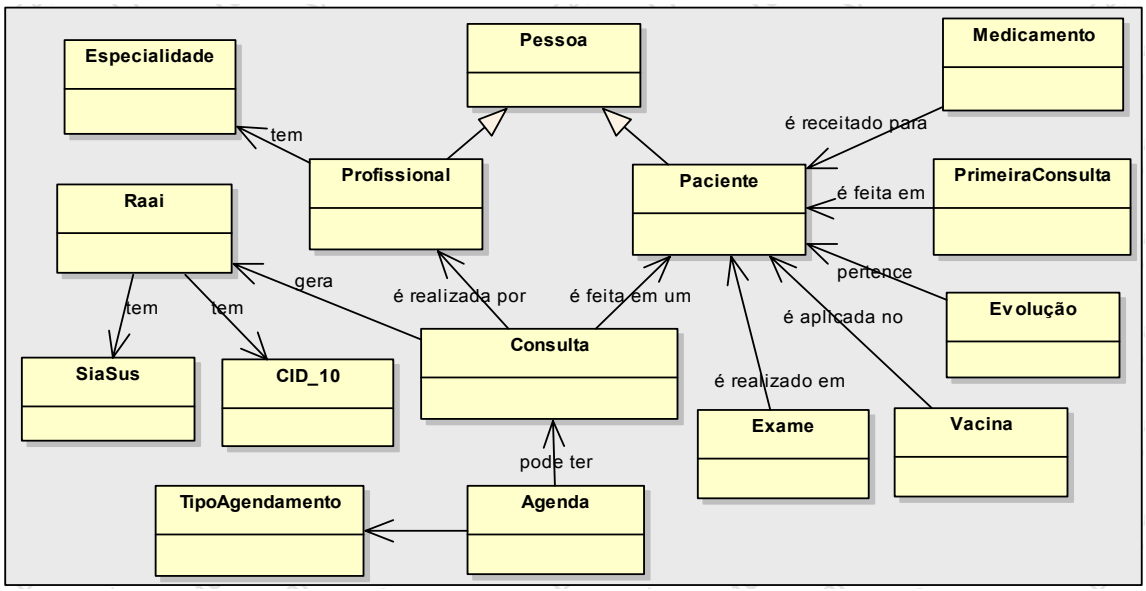


Figura 29: Diagrama de domínio do InfoSaúde

Silva & Videira (2001) destacam que uma prática comum para a identificação das classes candidatas é abstrair dos textos e documentos os nomes e os substantivos. Por sua vez, a identificação de relacionamento de associação entre as classes pode ser captada a partir de verbos e expressões verbais.

5.3.3.2 Prototipagem de GUI

O protótipo de GUI permitiu ao cliente ter uma idéia visual do sistema InfoSaúde. A idéia foi produzir uma interface mais próxima da realidade do usuário, ou seja, intuitiva, priorizando a simplicidade e a facilidade de uso.

Os primeiros esboços da interface foram produzidos a partir de técnicas simples, sem a utilização de ferramentas gráficas. A representação foi feita através de *storyboards*. Cybis (1997) conceitua que "Um *storyboard* é uma seqüência de desenhos contando uma estória sobre um usuário e a tarefa a ser realizada em uma determinada unidade de apresentação". A realização do desenho das principais telas, como o agendamento de consulta e abertura de prontuário, permitiu que se realizassem, rapidamente, alguns testes de usabilidade junto ao cliente.

5.3.3.3 Diagramas de caso de uso

Esta atividade consistiu na identificação da funcionalidade do sistema a partir da atuação dos atores envolvidos. Os casos de uso foram estruturados em agrupamentos lógicos em função dos atores, representados através do diagrama de casos de uso (ver figura 30 e 31).

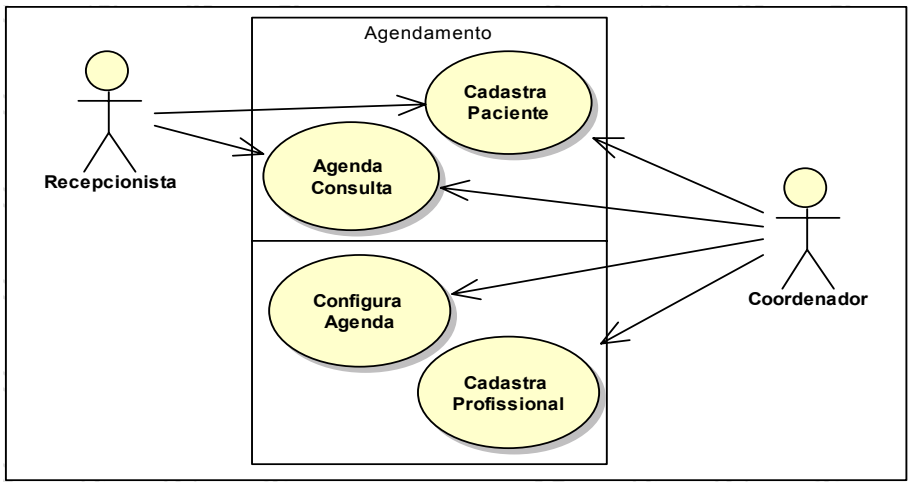


Figura 30: Diagrama de casos de uso da recepcionista e coordenador

A figura 30 apresenta os casos de uso do recepcionista e do coordenador. É importante destacar que o caso de uso “Configura Agenda” da figura 30, consiste na montagem da agenda conforme os horários e a especialidade do profissional. Desta forma, viabilizam o procedimento do caso de uso “Agenda Consulta”. Na figura 31, pode-se notar uma delimitação chamada de “Prontuário Médico”, onde o objetivo foi representar claramente os casos de uso que fazem parte do prontuário do paciente. Os casos de uso “Atende Paciente” e “Preenche RAAI” fazem parte do atendimento com um todo. Isso é definido pelo ICONIX como sendo a classificação dos casos de uso em pacotes.

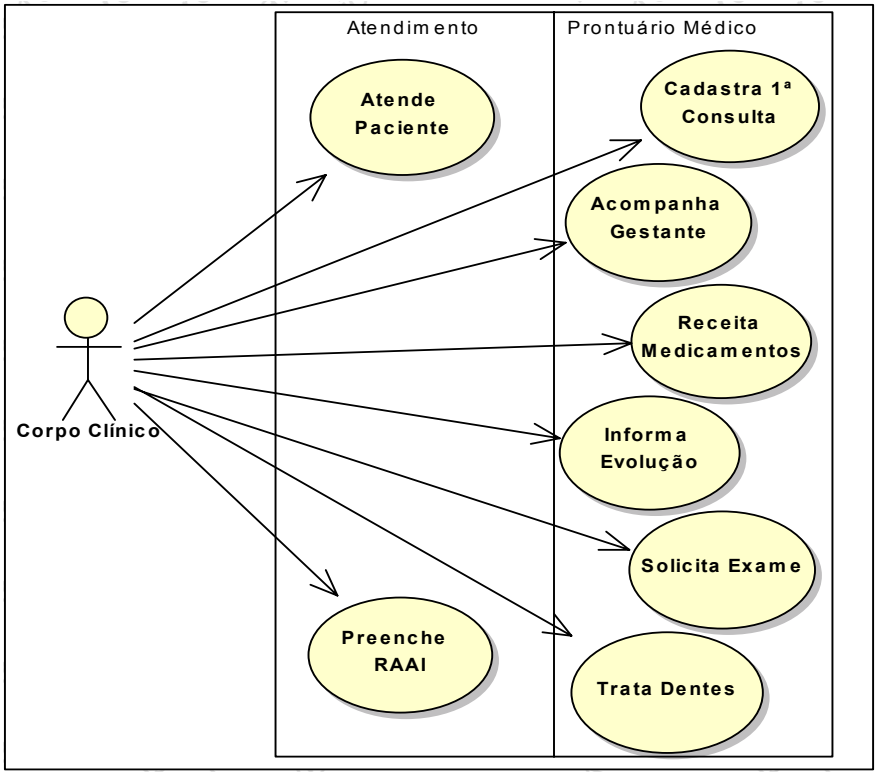


Figura 31: Diagrama de casos de uso do corpo clínico

5.3.3.4 Requisitos Funcionais

Como apresentado na seção 4.8, no ICONIX, os requisitos e os casos de uso são conceitos distintos, existindo uma relação de muito para muitos entre si. Além de realizar a associação entre requisitos e casos de uso. Robertson (1999) enfatiza que os “requisitos são coisas a descobrir antes de começar a construir um produto.” Desta forma, foi elaborada uma lista de requisitos funcionais. A figura 32 apresenta uma visão parcial desta lista.

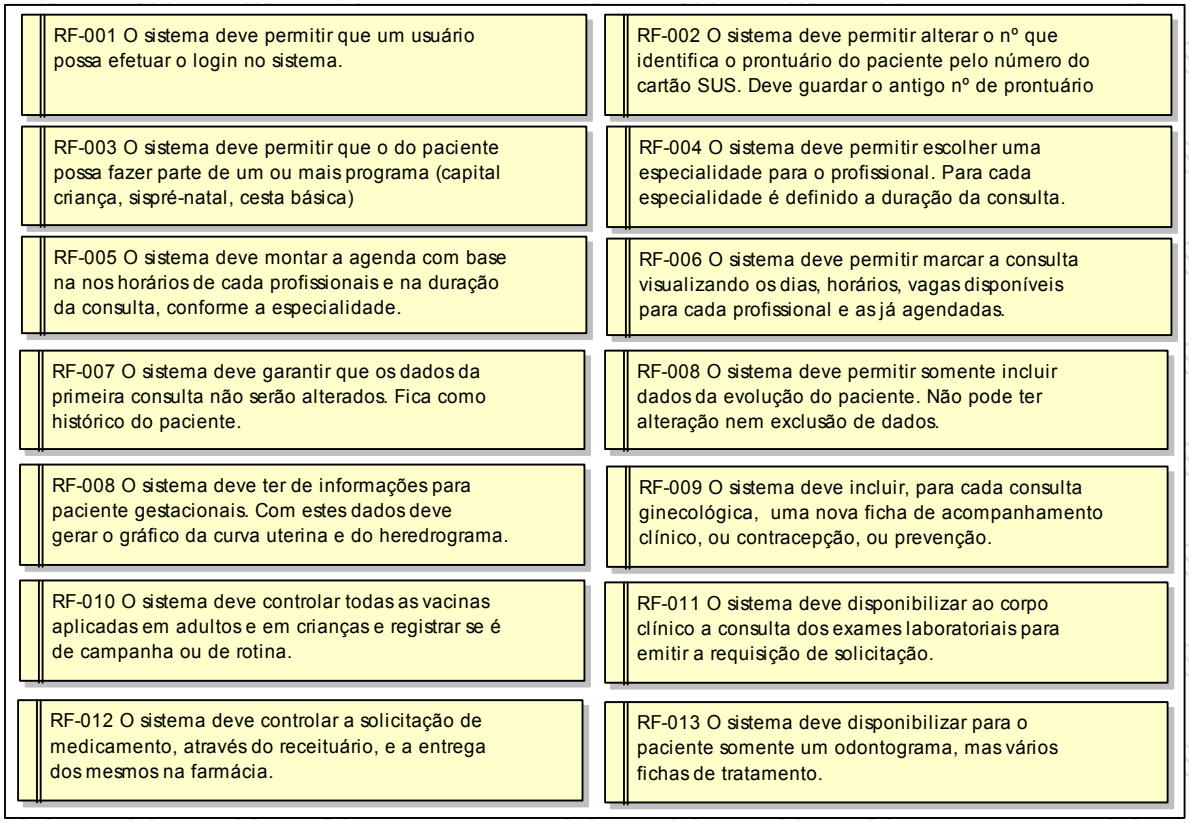


Figura 32: Lista de requisitos funcionais

5.3.4 Projeto preliminar

Nesta etapa, a primeira atividade consistiu em detalhar os casos de uso, que foram identificados anteriormente. Para tal, foram descritos textualmente todos os cenários correspondentes a cada caso de uso, sendo em geral contemplados os cenários do fluxo principal, alternativo e de exceção.

A principal sugestão do ICONIX, nesta atividade, é que não se deve perder muito tempo com descrição textual. Entretanto, se deve usar um estilo consistente que seja adequado ao contexto do projeto (SILVA & VIDEIRA, 2001).

Será apresentado, como exemplo, a descrição textual de um caso de uso, que pode ser visualizado na figura 33. A formatação deste modelo, foi resultado da compilação de vários livros (FOWLER, 2000), (LARMAN, 2000) e (KULAK & GUINEY, 2000), sugestão de profissionais da área e, experiência da própria autora. Também é utilizado no caso de uso, a especificação da versão, para o controle de versões.

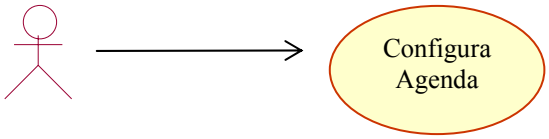
<table>
<tr><td>Versão: 2.0</td><td>Fase: Análise</td><td>Autor(es): Decka Cortese
Cristina Bona</td><td>Data Criação:</td><td>19/10/2001 17:02</td></tr>
<tr><td colspan="3"></td><td>Data Atualização:</td><td>26/12/2002 21:04</td></tr>
<tr><td colspan="5">USE CASE UC-008 – Configura Agenda
 
</td></tr>
<tr><td colspan="2">Breve Descrição</td><td colspan="3">Configurar a agenda do profissional conforme o horário de trabalho no posto de saúde</td></tr>
<tr><td colspan="2">Ator</td><td colspan="3">Coordenador</td></tr>
<tr><td colspan="2">Pré-condições</td><td colspan="3">A especialidade que o profissional pertence deve possuir agenda.</td></tr>
<tr><td colspan="2">Fluxo Principal</td><td colspan="3">1. O sistema apresenta uma tela para informar o Profissional e as opções de incluir, remover ou editar um horário de trabalho.
2. O coordenador informa o código do profissional e seleciona a edição.
3. O sistema apresenta uma tela de manipulação dos horários, com a seleção de data de início e fim.
4. O coordenador preenche os horários que o profissional selecionado estará disponível ou não, e solicita a gravação.
5. O sistema valida as informações.
6. O sistema grava as informações fornecidas pelo coordenador.</td></tr>
<tr><td colspan="2">Fluxos Alternativos e Exceções</td><td colspan="3">No item 2, o coordenador pode solicitar a busca de um profissional ou pode preencher parcialmente um código:
2.1 O sistema apresenta uma tela de busca.
2.2 O coordenador preenche o código ou o nome (pode ser parcial) e confirma.
2.3 O sistema busca os profissionais de acordo com as informações fornecidas e mostra o resultado na própria tela de busca.
2.4 O coordenador seleciona o profissional e vai para o passo 3.
No item 2, o coordenador pode optar pela exclusão
No item 2, o coordenador pode optar pela inclusão
A qualquer momento o Coordenador pode cancelar a operação e retornar ao passo 1.</td></tr>
<tr><td colspan="2">Pós-condições</td><td colspan="3">Montar a agenda de atendimento do profissional.</td></tr>
<tr><td colspan="2">Pendências</td><td colspan="3"></td></tr>
<tr><td colspan="2">Fonte ou documentos relacionados</td><td colspan="3">Diretor de planejamento: Silvio
Modelo de Agenda utilizada no posto. Anexo 12.</td></tr>
<tr><td colspan="2">Regras de negócio</td><td colspan="3">A data de fim somente deve ser preenchida se o período de configuração do trabalho é por tempo determinado.
O tipo de horário deve ser marcado com uma das opções.
1. Normal: horário padrão de trabalho. Enquanto for o horário válido não deve possuir data de fim. Tem prioridade 0 (zero), ou seja, se existir outro registro com outro tipo de horário, estes vão prevalecer.
2. Férias: período de férias. Tem prioridade 1 (um), vai sobrepor o Normal. No item 4 se o tipo de horário for marcado como Férias, deverá apresentar na agenda para cada dia de férias a informação “Férias”, destacado em vermelho. Não deve mostrar os horários das consultas.
3. Outros: horário de exceção. Por exemplo: Licença, folga, etc. Deve ter horário de início e fim, se não for informado o horário de fim vai ficar valendo este horário como padrão. Pode se Tem prioridade 2 (dois), sobrepõe o Normal e Férias.
Deve ser informado para cada dia da semana se o profissional trabalha e qual o horário de inicio e fim do turno da manhã e da tarde.</td></tr>
</table>

Figura 33: Modelo textual de caso de uso do InfoSaúde

A segunda atividade desta etapa, foi ilustrar graficamente as interações entre os objetos participantes do caso de uso, através do diagrama de robustez. No projeto do InfoSaúde, foram realizados diagrama de robustez somente para os casos de uso que a o time sentiu necessidade de validar a descrição textual com a especificação do comportamento dos objetos. Na figura 34, é apresentado um exemplo do diagrama de robustez do caso de uso “Configura Agenda”, mostrado na figura 33.

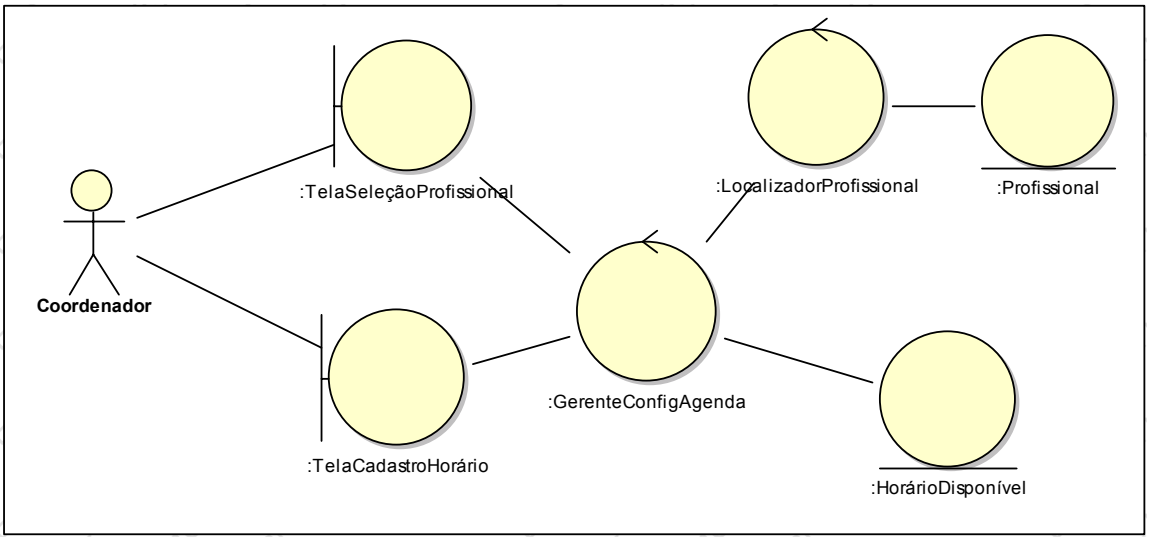


Figura 34: Diagrama de robustez do caso de uso “Configura Agenda”

Posteriormente à realização dos diagramas de robustez, foi atualizado o diagrama de domínio. Além das classes já definidas, foram descobertas novas classes como “LocalizadorProfissional” e “GerenteConfigAgenda”. Também foram descobertos atributos e incluídos no diagrama de classe (domínio). A representação detalhada do diagrama de classe poderá ser visualizada na seção seguinte.

5.3.5 Projeto detalhado

O principal objetivo desta fase é especificar o comportamento detalhado do sistema, através da construção o diagrama de seqüência e atualização do modelo estático. Para a realização desta atividade, é necessário considerar a infra-estrutura computacional e a tecnologia de desenvolvimento envolvida, que foi apresentada anteriormente na seção 5.1.1.

No projeto preliminar, o comportamento de um caso de uso foi especificado através do diagrama de robustez. Nesta fase, a primeira atividade foi especificar o

comportamento detalhado através do diagrama de seqüência. Pois, o objetivo é evidenciar o fluxo de mensagens trocadas entre os objetos e os atores. A figura 35 ilustra o diagrama de seqüência relativo ao caso de uso “Configura Agenda”.

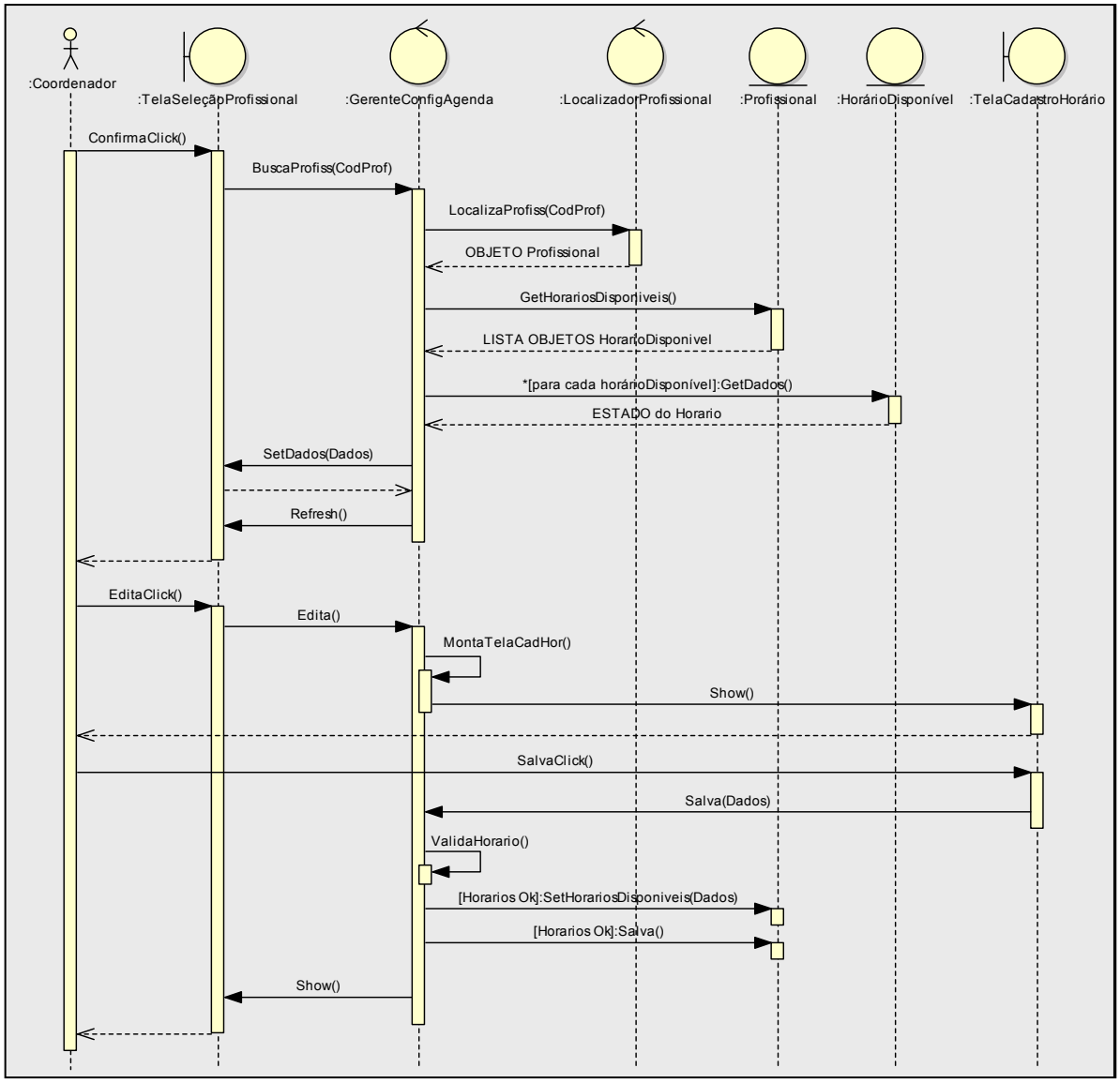


Figura 35: Diagrama de seqüência do caso de uso “Configura Agenda”

A atividade seguinte à construção do diagrama de seqüência, foi o término do modelo estático. Para isso, o diagrama de classe foi atualizado com informações detalhadas, baseado nos diagramas de seqüência que foram desenvolvidos, que por sua vez, identificaram as operações que fazem parte das classes correspondentes. A figura 36 apresenta parte do diagrama de classe do InfoSaúde. Por motivos de simplicidade e facilidade de leitura não se introduziu todas as classes identificadas.

A definição e utilização de padrões de projeto também faz parte desta fase final do projeto do diagrama de classe. Fernandes & Lisboa (2001) destacam que o grande atrativo da utilização de padrões de projetos, é permitir a reutilização de soluções previamente encontradas. Uma vez que, documentam um problema, suas características e soluções. Mais informações sobre padrões de projeto pode ser encontradas em (GAMMA et al, 1995), (LARMAN, 2000) e (APPLETON, 2000).

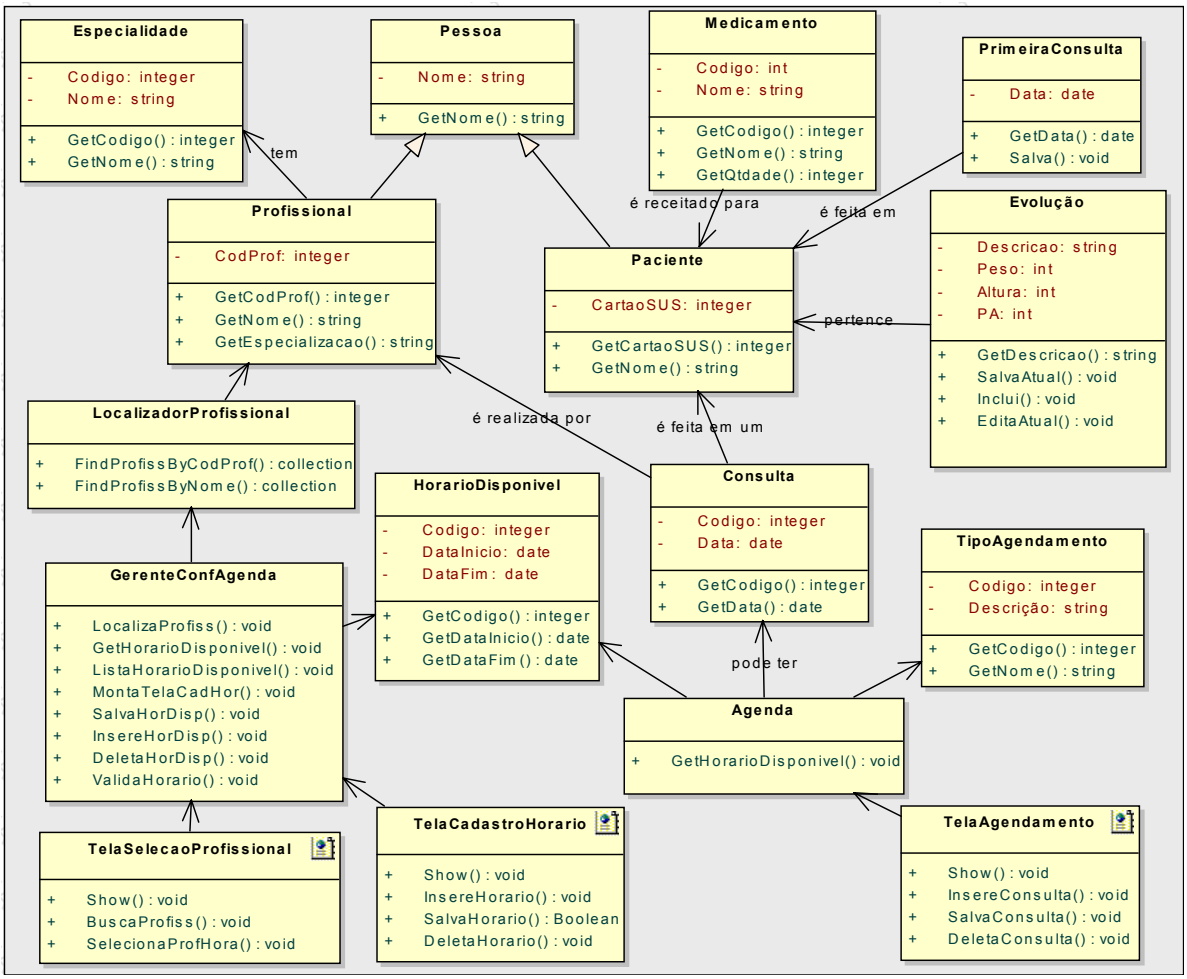


Figura 36: Diagrama de classe, parcial, do InfoSaúde

5.3.6 Implementação

O ICONIX considera que a atividade de implementação está fora do seu âmbito e foco de interesse (SILVA & VIDEIRA, 2001). Entretanto, disponibiliza em conjunto restrito de sugestões. Algumas destas sugestões auxiliaram o time a atribuir determinadas tarefas e atividades aos diferentes membros da equipe. Como por exemplo, o ICONIX sugere que:

- os casos de uso sejam escritos por pessoas com experiência em desenho de interface ou por técnicos com experiência na produção de manuais de usuário;
- os modelos de domínio e diagramas de classe detalhados sejam construídos por pessoas com experiência em projeto de base de dados;
- os programadores de sistema devem pensar em aspectos como desempenho, segurança, e serem responsáveis pelos diagramas de estado e colaboração, se forem utilizados.
- a tarefa de desenho detalhado, em especial o diagrama de seqüência, seja realizada, ou pelo menos supervisionada, por pessoas com experiência em modelação OO.

Desta forma, o sistema Infosaúde foi implementado e testado. Os testes foram realizados com base nos casos de uso. O resultado dos testes, foram descritos em arquivos textos que retornavam a equipe de programadores. Os programadores realizaram as devidas correções e as enviam novamente os testadores. Este ciclo se repetiu até que o sistema estivesse livre de erros.

5.3.7 Entrega da versão para produção

O InfoSaúde entrou em produção após a execução dos testes e uma demonstração do funcionamento para toda a equipe de especialistas de domínio e usuários finais. Além disso, foi dado treinamento de utilização do sistema. As turmas foram montadas conforme a ação que o usuário irá realizar no sistema. A ação do usuário, estava quase sempre relacionada à ação do ator. Por sua vez, o sistema foi disponibilizado para produção. Parte da interface pode ser visualizada na figura 37 e 38. A figura 37 é o resultado dos artefatos anteriormente detalhados – “Configura Agenda”. A figura 38 mostra a agenda gerada com base nos dados da configura;ao da agenda.

Para a incorporação de novos requisitos no InfoSaúde, uma nova versão é planejada. Então, o time parte novamente da análise, projeto, implementação e teste, ou seja, aplica-se o conceito de processo iterativo e incremental.

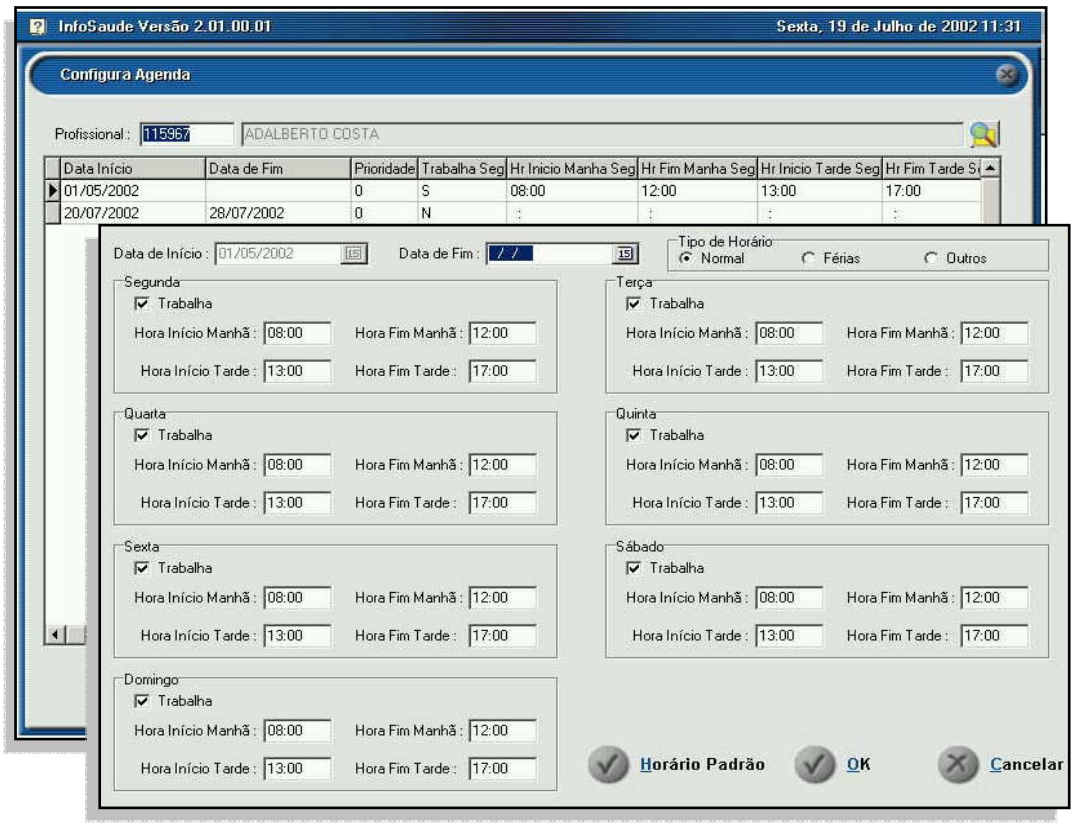


Figura 37: Tela de configuração da agenda do InfoSaúde

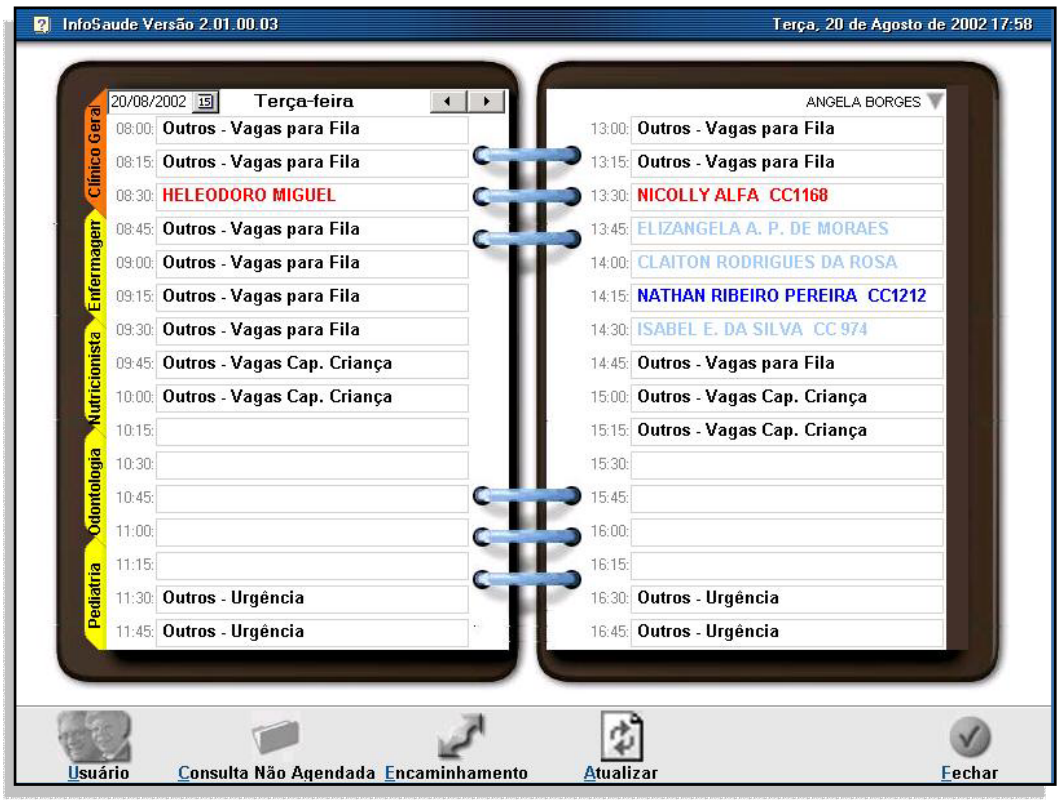


Figura 38: Tela de agendamento de consultas do InfoSaúde

6 ANÁLISE DOS RESULTADOS

Aplicando-se os passos descritos no item 5.2 e no item 5.3, procedeu-se a avaliação dos processos XP e ICONIX, com o objetivo produzir subsídios para abstrair os pontos positivos e negativos. Além disso, o comportamento dos recursos de cada processo foi comparado, sendo um dos pontos de destaque deste trabalho. Também foi elaborado um questionário, em que se procurou indagar alguns pontos polêmicos do XP, e finalmente foi realizada uma entrevista com clientes sobre a preferência entre os dois processos.

6.1 Pontos Positivos do XP

Pode-se considerar que XP é adequado em projetos onde os clientes não sabem exatamente por onde iniciar o desenvolvimento e que a probabilidade de mudarem de idéia é grande. Uma vez que o software é rapidamente produzido e que o projeto de desenvolvimento recebe *feedback* rápido, apenas parte do planejamento é realmente perdido. Então, se o projeto é feito para atender as variações internas e de requisitos, o XP parece o mais adequado.

Isto encurta o ciclo para entrega de uma versão, sendo uma das forças primárias do XP. Realmente, isto é muito positivo: os clientes vêem rapidamente por aquilo que eles pagaram, embora incompleto, e os desenvolvedores permanecem mais interessados e na trilha certa. Os desenvolvedores têm mais liberdade para escolher suas tarefas e para consertar problemas, o que incorpora motivação para o time.

Outra importante característica e que também é um dos principais *slogans* do XP é: sempre faça a coisa mais simples que poderia possivelmente funcionar. O pensamento que Beck (2000) mostra, ao longo do seu livro, que a simplicidade deve ser sempre maximizada. Percebeu-se que, a busca da simplicidade no sistema GID trouxe redução de tempo para o seu término, quando comparado a um projeto complexo. Além disso, ele é também fácil de entender e modificar.

A integração contínua é outra idéia muito interessante para desenvolver um projeto. O fato de o XP trabalhar com pequenas versões e que todo o código é

testado, dá uma visão confortável de progresso e mantém os desenvolvedores entusiasmados tendo as tarefas à mão.

Beck (2000) argumenta que os testes extensivos e a integração contínua, são tarefas que devem ser executadas pelos próprios membros do time. Realmente, este argumento mantém a entrega da versão do produto simplificada, e os desenvolvedores podem conhecer todo o sistema.

6.2 Pontos Negativos e Problemas com XP

Um problema importante de implementação com XP é que ele é um modelo difícil de ser usado quando se pretende vender serviços para outras companhias. Isto fica evidente quando o cliente deseja um levantamento explícito de requisitos ou, pelo menos, queira um preço fixo e o tempo de produção estimado. Convencer o cliente a contratar um serviço sem estas informações é muito difícil.

Em relação ao custo de mudanças ser baixo com XP, existem controvérsias. McBreen (2002), afirma que não existe nenhum dado real, para uma afirmação ou para a outra, que estime o custo de mudanças em software moderno. Realmente, estimar o custo de uma mudança de requisitos não é fácil porque depende do impacto que irá refletir no sistema. O XP parece combinar com negócios onde a mudança de requisitos é rápida e emergente, porque esta mudança é nova e não poderia ter sido prevista. A abordagem tradicional é provavelmente melhor em ambientes mais estáveis, em que os requisitos são bem conhecidos e a mudança é mais fácil de ser prevista. Por sua vez, quando se olha para custo de mudança, é necessário olhar tanto para o time de desenvolvedores quanto para a comunidade de usuários.

A metodologia XP tem uma propensão para tentar convencer por choque. Nega muitas práticas que foram tentadas por décadas, algumas consideradas eficientes como a documentação e o levantamento de requisitos, somente porque o desenvolvimento de software em geral tem problemas. Entretanto, isto não significa que tudo o que foi realizado até o momento não funciona.

A análise do problema, por exemplo, nunca é mencionada como algo para ser realizado. Os clientes devem ter uma boa idéia das “histórias” que eles precisam

escrever inicialmente. Isto poderia ser facilmente contornado com a ajuda do analista de sistemas, antes de iniciar o processo XP. Mas isto não é citado. Somente quando o cliente já está executando seu papel, ele teria suporte de um desenvolvedor. A intenção de integrar o cliente no processo, escrevendo histórias, pode exigir muito esforço do cliente.

Neste contexto, destaca-se também o problema da ausência de uma fase de engenharia de requisitos mais específica. Os requisitos não são construídos para engessar desenvolvedores, mas uma fase na qual o problema é continuamente refletido e refinado, até que uma solução coerente e simples seja tomada. É importante pensar sobre o problema, especialmente porque se não for gasto uma porção significativa de tempo antes de iniciar o projeto, a chance de planejar corretamente para evitar *refactoring* é perdida. Isto não significa dizer que se deva passar tempo excessivo com requisitos e com planejamento, especialmente se eles mudam com freqüência. Porém, uma idéia global do problema a ser resolvido e um caminho para a solução acordados são sempre boas vantagens.

O XP parece ser adequado para projetos onde o ritmo durante o desenvolvimento é rápido. Quando o processo de desenvolvimento amadurece e menos funcionalidade é solicitada, não existe nenhuma especificação sobre como o XP se comporta em uma manutenção reduzida no desenvolvimento. O XP parece não suportar projetos em que o desenvolvimento segue um ritmo mais lento.

O problema descrito acima pode não parecer óbvio, então pode-se esclarecer que, pouca ou nenhuma documentação significa “o código” e que o conhecimento do projeto deve estar confiado “nas pessoas”. Infelizmente, estes recursos não são sempre confiáveis. É comum alguns desenvolvedores deixarem o projeto (isto é previsto em XP), mas ocasionalmente todos ou a maioria dos desenvolvedores deixam o projeto. Então, como ninguém sabe explicar como o código funciona, a barreira para a entrada de um novo desenvolvedor é grande. Porém, se existir documentação e que esteja corretamente mantida, mesmo que informal, pode ajudar a reduzir este problema. Esse é um problema que o XP evita mencionar.

Boehm (*apud* Ambler, 2002) mencionou que um projeto documentado ajuda um perito externo a diagnosticar problemas. Entretanto, Beck (*apud* Ambler, 2002) discorda, e expõe que um perito externo passa tempo considerável diagnosticando

projetos e acaba incomodando as pessoas como detalhes pouco significativos e não técnicos na documentação.

O XP confia muito na possibilidade de aplicar a técnica de *refactoring* e *retrofitting* (reajustar) no código existente com o objetivo de aumentar a funcionalidade. Isto pode funcionar se as mudanças forem simples. Mas, muitas vezes se encontram problemas significativos durante o desenvolvimento e adaptação do código. Em alguns casos, podem ser encontrados problemas até em mudanças simples. Neste sentido, destaca-se que 'simples' é um valor percebido e não um objetivo, quando o assunto é desenvolvimento de software. Pequenas mudanças podem causar sérios impactos e um planejamento adequado de pré-codificação poderia ajudar a resolver este problema.

Em relação aos testes, se uma grande parte do projeto exigir implementação de interface com o usuário, pode ser um problema para os desenvolvedores criarem testes automatizados. De acordo com Myers & Rosson (*apud* REIS, 2000), os estudos mostram que normalmente 50% de todo código é dedicado para interfaces com os usuários. Obviamente, os testes podem ser trabalhados manualmente, embora mais incômodos. Porém, o XP conta com desenvolvimento orientado por teste; problemas com teste podem significar desenvolvimento lento.

McBreen (2003) critica a posição de que um bom time XP é pelo menos seis vezes mais produtivo que um time de engenharia de software tradicional. Além disso, os programadores do XP são considerados todos profissionais capacitados e com muita disposição para executar uma tarefa. Realmente, pode-se dizer que alguém não capacitado ou não disposto não deve fazer parte do time. Entretanto, em algum momento, um desenvolvedor poderá passar por uma fase ruim ou não estar disposto a executar determinada tarefa que lhe foi designada. Em um grupo, isto comumente pode acontecer. Uma solução para tarefas que ninguém quer, ou que estão sendo mal executadas é realizar uma votação ou criar normas de negociação. Mas, isto pode gerar conflito com a idéia de livre escolha e de propriedade (*ownership*) que o XP sugere.

É importante observar que a integração contínua não é uma idéia que surgiu com o XP e todos processos modernos que adotam ciclos de vida como o modelo em

espiral, o RAD (*Rapid Application Development*) ou o modelo iterativo, empregam a filosofia de integração contínua.

6.3 Pontos Positivos do ICONIX

O ICONIX é adequado a sistemas em que não se descarta análise e projeto. Neste contexto, o ICONIX destaca-se, pois apresenta claramente as atividades de cada fase. Exibe a seqüência de passos que devem ser seguidos e oferece suporte através da abordagem UML (OMG®, 2001).

Duas características importantes do processo consistem na identificação e representação do modelo de domínio e dos casos de uso. Positivamente, a partir destes dois modelos tudo se desenrola de forma iterativa e incremental. Pois, cada caso de uso é detalhado através de uma descrição textual e com o respectivo diagrama de robustez. Em seguida, são identificados novos objetos e detalhes, os quais são incorporados ao diagrama de domínio (versão de alto nível do diagrama de classe). Logo após, se elaboram os diagramas de seqüência, onde se identifica o comportamento (operações) dos objetos. Finalmente, as operações são adicionadas na versão detalhada e final do diagrama de classe.

Outro ponto positivo do ICONIX é ser orientado por casos de uso. Ou seja, pode haver rastreamento a partir dos casos de uso para qualquer outro artefato gerado no processo. Os casos de uso são criados especialmente nas fases de concepção e elaboração, onde 80% deles são identificados (SANTIAGO, 2000). Por sua vez, um caso de uso é mais eficaz quando elaborado do ponto de vista do usuário. Desta forma, os casos de uso representam o fluxo das ações do usuário e as respostas do sistema para estas ações. Portanto, o foco deve ser na visão externa do sistema, na visão que os atores tem dele.

A rastreabilidade, também foi avaliada como sendo um ponto positivo no ICONIX. Pois, em todo o caminho percorrido deve ser traçado uma referência aos requisitos identificados. Desta forma, a equipe se mantém próxima às necessidades do usuário e assegura que não vai omitir quaisquer requisitos durante a implementação. A rastreabilidade é chave, pois elucida cada rastro de requisito em um ou mais casos de uso e, em uma ou mais classes do modelo de domínio e como trabalham juntos.

Neste caso, quando for analisado o choque de propor uma mudança em um requisito específico, a rastreabilidade revelará realmente quais elementos do sistema que podem ser afetados com a mudança.

6.4 Pontos Negativos e Problemas com ICONIX

O ICONIX não sugere explicitamente nenhum diagrama para modelar processo de negócio na fase preliminar do projeto. Mesmo sendo o ICONIX um processo que pretende ser prático e simples, poderia no entanto, se beneficiar do diagrama de atividades disponível na UML (OMG®, 2001) para modelar processos de negócios. Da mesma forma que, outros processos igualmente simples e práticos como o Grapple, abordado por McConnell (*apud* SILVA & VIDEIRA, 2001) sugere a utilização do diagrama de atividades para modelar processos de negócio na fase preliminar.

Um aspecto crítico de modelagem de casos de uso envolve os fluxos alternativos. É fundamental pensar sobre todos os fluxos alternativos possíveis para cada caso de uso, sempre que possível. Considerando que o fluxo principal é mais fácil de identificar e escrever, não significa, porém, que o fluxo alternativo deva ser postergado até que o projeto detalhado seja implantado. De fato, se isto acontecer pode causar omissões sérias nestes pontos, gastando muito tempo para escrever o fluxo alternativo, frente aos demais artefatos já definidos. Este é um cuidado que o próprio ICONIX ressalta. Quando importantes fluxos alternativos não são descobertos até a fase de codificação e depuração, o programador responsável por escrever o código tender a tratar isto de forma mais conveniente no momento. Isto, não é saudável para o projeto. Então, pergunte várias vezes: Existe alguma coisa que pode acontecer? Existe outra forma tratamento? Isto está correto? Garanta com isto um conjunto rico de fluxos alternativos.

Outra característica forte do ICONIX é a distinção entre requisitos e casos de uso. Uma desvantagem clara desta posição do ICONIX é obrigar a equipe de projeto identificar e elaborar de uma lista de requisitos, assim como a manter as associações entre os requisitos e os casos de uso. Isto requer um esforço adicional e um acréscimo do volume de trabalho, que poderia ser evitado em um processo que pretende ser rápido e simples.

6.5 Comparação do XP e do ICONIX

Serão examinados simultaneamente os recursos do processo XP e do ICONIX para determinar as semelhanças, as diferenças e/ou as relações entre ambos. A comparação pode ser visualizada na Tabela 3.

Tabela 3: Comparação dos processos

RECURSO	XP	ICONIX
Requisitos	• próprios **cartões** de história: não especifica formalmente.	• **lista** de requisitos bem definidos.
Regras de negócio	• definidas pelo **cliente**; • estimadas pelo **cliente**: escopo, prioridade, composição de versão e data das versões.	• definidas pelo **cliente**; • estimadas pelo **time**: não especifica detalhes.
Regras técnicas	• estimadas pelos **técnicos**: tempo, riscos técnicos, tecnologia.	• **não** especifica formalmente.
Papéis time	• especifica.	• **não** especifica.
Usuários do sistema	• **não** especifica.	• especifica através dos **atores**.
Cliente	• deve fazer parte do time (on-site)	• não precisa fazer parte do time.
Documentação	• cartões de história (temporária); • direto no código fonte.	• lista de requisitos; • casos de uso (texto); • diagramas da UML.
Programação	• em **pares**.	• **individual**.
Refactoring	• utiliza.	• não especifica.
Orientado por	• **testes**.	• **casos de uso**.
Testes	• testes de unidade (obrigatório); • testes funcionais (preferencialmente automatizados).	• testes de unidade (sugerido); • testes de funcionais (não sugere automatização).
Metáfora	• utiliza	• não utiliza
Interface	• não especifica	• protótipo de GUI
Ciclo de vida	• iterativo incremental.	• iterativo incremental.
Diagramas	• **não** utiliza.	utiliza padrão **UML**: • modelo de domínio; • diagrama de casos de uso; • diagrama de robustez; • diagrama de seqüência; • diagrama de classe.
Fases do processo	especifica, mas as fases são **confusas**: • exploração; • planejamento; • iteração da primeira versão; • produção; • manutenção.	especifica, as fases são **claras**: • análise de requisitos; • projeto preliminar; • projeto detalhado; • implementação.

Tempo estimado	• especifica nos cartões de história.	• não especifica formalmente, mas os casos de uso podem ser pontuados.
Adotar	• mudança de paradigma.	• procedimento normal.
Número de programadores	• de 2 a 10 programadores.	• não especifica limite.
Implementação	**é o foco** do XP, e as atividades que auxiliam são: • propriedade coletiva; • programação em pares; • histórias do usuário; • *refactoring*; • testes de unidade.	**não é o foco** do ICONIX, mas apresenta atividades que auxiliam: • ar diagrama de componente (se for necessário); • testes de unidade; • testes de integração; • testes de aceitação do usuário.
Definição da arquitetura	• na fase de exploração.	• antes da fase de análise de requisitos.
Foco do processo	• baseado nas **pessoas**.	• baseado em **artefatos**.
Conhecimento	• programação OO; • ***refactoring***.	• programação OO; • **UML**.

6.6 Questionário

A partir de um questionário, foi aplicada uma série de 6 (seis) perguntas ao grupo de alunos da Universidade de São Paulo (USP) e aos Professores que ministraram a disciplina de Laboratório de Programação Extrema. A proposta da disciplina é o desenvolvimento de um sistema de médio porte utilizando as técnicas do XP (mais informações podem ser encontradas em http://www.ime.usp.br/~xp/). Também foi aplicado o questionário ao time da CEFET/SC que utilizou o XP. As questões são listadas a seguir e o resultado de cada uma representado graficamente nas figuras 39, 40, 41, 42, 43 e 44. Participaram do questionário 13 (treze) pessoas.

1ª Questão - Gosta de programar em par (programação pareada)?

Figura 39: Gosta de programação em pares

Comentário: existe sempre uma condição, dos próprios programadores, que se o par é bom, a programação pareada é agradável e as experiências podem ser compartilhadas. Porém, se o par não é tão capacitado, acaba atrapalhando o fluxo do trabalho, pois perde-se tempo ensinando ou supervisionando, ou ainda, assume-se a programação sozinho. Enfim, é difícil conseguir um time de programadores com conhecimento homogêneo e disposição para executar as tarefas no mesmo ritmo.

2ª Questão: A velocidade de produção da programação em par é equivalente maior que a velocidade da programação individual, ou seja, a dupla produz quase o dobro que um programador isolado?

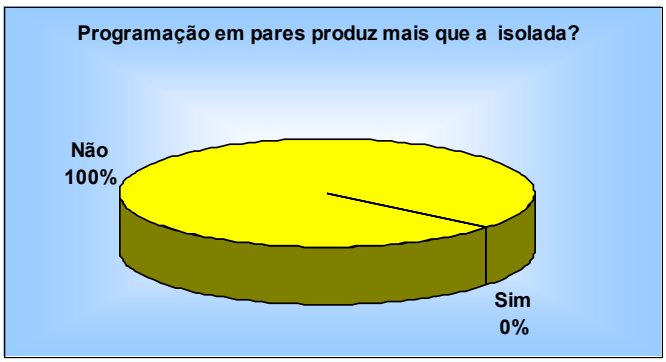


Figura 40: Velocidade de programação

Comentário: um argumento dos programadores é que, se o critério for o número de linhas a resposta deve ser “não”, mas se o critério for à qualidade do software gerado a resposta deve ser “sim”. Este é um dos pontos mais discutidos entre defensores e críticos do XP. A programação pareada faz bastante sentido no

contexto do XP, uma vez que a atividade de projeto é realizada juntamente com a atividade de teste. Entretanto, se você tiver um time de programadores experientes trabalhando isolados, é provável que a qualidade das linhas de código seja tão aceitável quando a programação pareada.

3ª Questão: O “cliente”, teve dificuldade em escrever as histórias (*story cards*)?

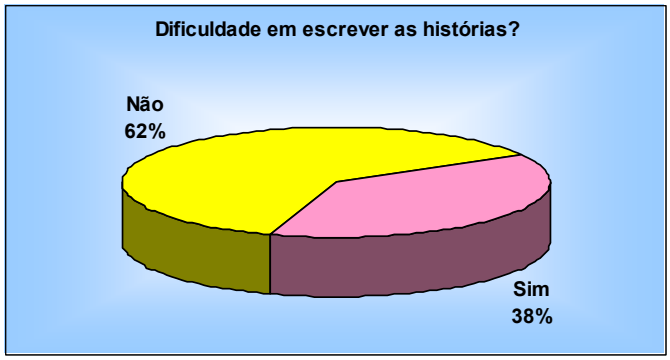


Figura 41: Dificuldade de escrever histórias

Comentário: quando o cliente tem alguma experiência, mesmo que informal, de escrever requisitos, as primeiras versões rapidamente são elaboradas, e como XP recomenda as histórias vão evoluindo com o tempo. Porém, se o cliente não tem muita noção do que realmente é importante estar escrito nas histórias, para que o desenvolvedor execute seu trabalho, pode exigir um esforço inicial desgastante.

4ª Questão: Sentiu necessidade de alguma documentação mais formal, como diagramas?

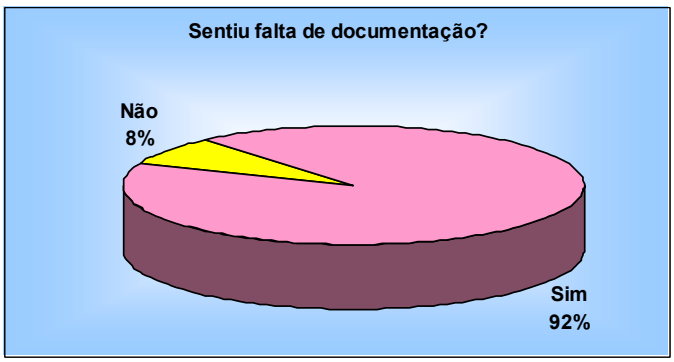


Figura 42: Necessidade de documentação

Comentários: A necessidade de documentação de projeto foi grande. Para suprir a carência, os times acabaram utilizando os diagramas da UML. Na verdade, construir software sem projeto documentado é muito complicado, é o mesmo que retroceder e negar anos de estudo os da engenharia de software.

5ª Questão: Você aplicou técnica de refatoramento (*refactoring*) durante a programação?

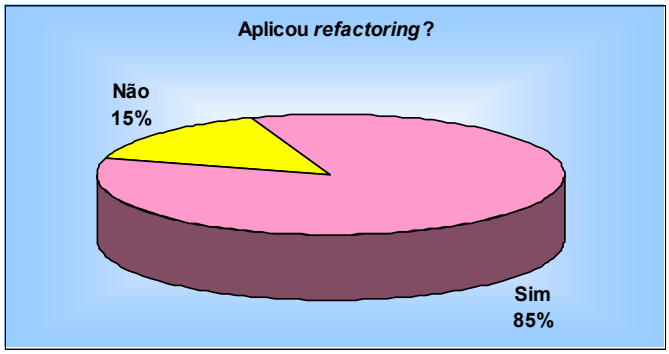


Figura 43: Aplicação de *refactoring*

Comentários: aplicar a técnica de *refactoring* ajudou os grupos melhorarem o código constantemente. Considerando sempre, pequenas mudanças.

6ª Questão: Observando a sistemática do time, atendeu a prática de integração contínua (o sistema construído é integrado cada vez que uma tarefa é completada)?

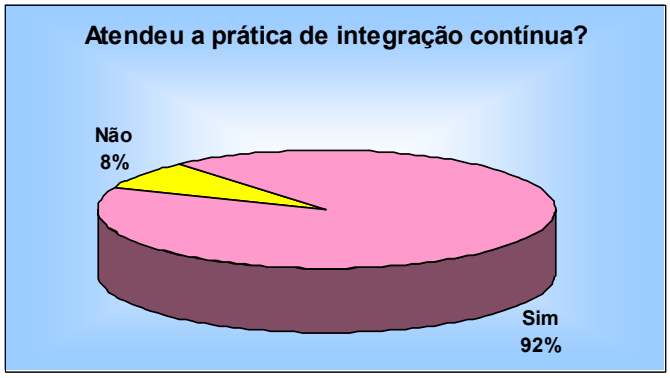


Figura 44: Aplicação da integração contínua

Comentários: os grupos utilizaram ferramentas de controle de versão para auxiliar esta prática. Realmente, estas ferramentas ajudam muito, pois pode-se facilmente atualizar a última versão, além disso, todos sabem quem está modificando o que. Entretanto, o suo destas ferramentas é sempre recomendado, independentemente do processo adotado.

Além do questionário, foram entrevistados 10 (dez) clientes, contratantes de serviços de desenvolvimento de software, sobre a preferência de trabalharem com XP ou com ICONIX. Para tanto, foram explanadas as principais características de cada processo; do XP foram relatadas as 12 práticas, comentadas anteriormente na seção 3.3; e do ICONIX foram explanadas as tarefas e os marcos, discutido anteriormente na seção 4.2. O resultado pode ser visualizado na figura 45.

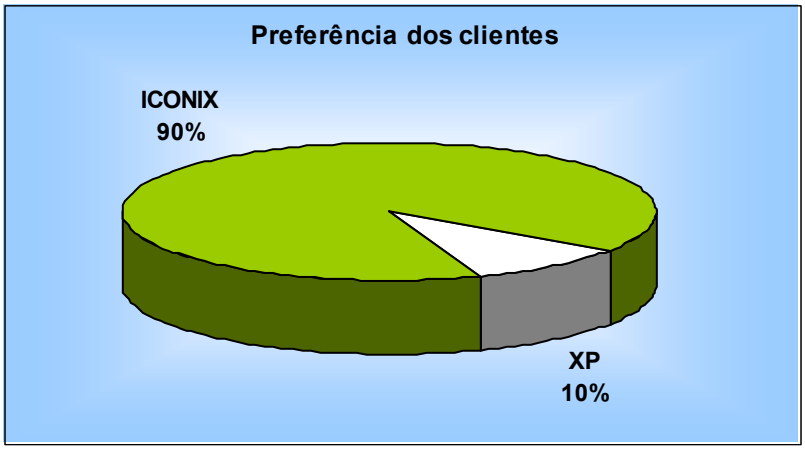


Figura 45: Preferência dos clientes entre o XP e o ICONIX

Comentários: Quando são expostas as características como, pouca ou nenhuma documentação, cliente *on-site* e escrevendo histórias, a rejeição e desconfiança no processo são grandes. Talvez por ser uma mudança de paradigma muito grande e ainda ser extremamente recente se comparado ao tempo de existência das metodologias chamadas de “tradicionais”. Por sua vez, o ICONIX imprime ao cliente o aspecto de um trabalho mais profissional, além de uma transparência maior no progresso do desenvolvimento e nas fases bem definidas.

6.7 Considerações Finais

No decorrer deste capítulo, foi representado, através da especificação dos pontos positivos e negativos e dos gráficos e tabelas, as avaliações obtidas com a aplicação dos processos descritos metodologicamente no capítulo 5. Pode-se observar que, ambos os processos podem ser utilizados dependendo das características exigidas pelos clientes e dos conhecimentos do time. Entretanto, existe ainda um longo caminho a ser percorrido em busca do estado da arte para o processo de software. O capítulo seguinte apresenta as conclusões, recomendações e trabalhos futuros.

7 CONCLUSÃO

Neste trabalho foram apresentadas algumas reflexões fundamentais relativas à importância do processo de software, principalmente como base para o levantamento de requisitos, para o planejamento de projeto e para o apoio ao desenvolvimento. Além de dar suporte aos especialistas de domínio (clientes) e aos desenvolvedores (técnicos) na escolha do processo que melhor se adapta a necessidade da organização.

Foram apresentados também, os fundamentos teóricos do dois processos aplicados, o XP e o ICONIX, bem como os conceitos relacionados à Engenharia de Software, como as fases do processo, as atividades e os modelos de ciclo de vida. As metodologias ágeis foram abordadas como sendo metodologias modernas de desenvolvimento e como uma reação a modelos extremamente conceituais e a metodologias “monumentais” (FOWLER, 2001). Por fim, foi aplicado e demonstrado o processo XP e o processo ICONIX, mostrando todo o ciclo evolutivo, desde a concepção até a entrega da primeira versão. Esta aplicação gerou subsídios para avaliar os processos. Uma vez que, o objetivo foi abstrair os pontos positivos e negativos e comparar o comportamento dos recursos de cada processo, contemplando a proposta desta pesquisa.

Adicionalmente, foi elaborado um questionário, indagando alguns pontos polêmicos do XP. Um ponto importante a destacar é que a falta de documentação prejudica a adoção do XP. Essa necessidade é sentida tanto pelos clientes, que ficam inseguros tendo como documentação somente com o código, quanto pelos desenvolvedores, que não tem visualização gráfica de projeto. Finalmente, foi realizada uma entrevista com clientes, contratantes de serviço de software, sobre a preferência entre os dois processos (ver figura 45). Esse levantamento mostrou claramente que, 90% dos clientes preferem o ICONIX ao XP. Mais uma vez, a falta de documentação foi determinante para a escolha do processo. Neste caso, também pode-se considerar que a necessidade do cliente *on-site* foi um agravante.

Esse trabalho espera ter deixado duas contribuições principais. A primeira delas está relacionada à exemplificação textual e gráfica da aplicação dos dois processos. A aplicação, além de servir como instrumento de estudo para os acadêmicos, pode

ser utilizada como suporte para a tomada de decisão sobre qual processo melhor de adapta a realidade da empresa ou para a construção de um determinado sistema, que pretenda empregar o processo XP ou o ICONIX.

A segunda contribuição está relacionada à busca constante de um modelo de processo que possa ser aplicado de forma “produtiva” em um ambiente de desenvolvimento de softwares. Uma das barreiras na aplicação de determinadas metodologias está relacionada ao excesso de recursos e controles que a mesma requer, pois acaba demandando um tempo excessivo à construção e atualização de artefatos, gerando custos. Considerando esse tipo de dificuldade, a comunidade de Engenharia de Software busca alternativas, como às metodologias ágeis (XP) e as metodologias práticas ou intermediárias (ICONIX). A definição de um processo ideal, que seja produtivo e que proporcione garantia de qualidade ao software produzido, ainda é um desafio a ser enfrentado e vencido pelos pesquisadores da Engenharia de Software. Enfim, existe um longo caminho a ser percorrido na busca do estado da arte para o processo de software.

7.1 Trabalhos Futuros

Apesar deste trabalho ter alcançado os objetivos propostos, verificou-se que alguns detalhes e acréscimos podem ser elaborados a partir dos resultados obtidos. Sendo assim, foram feitas as seguintes recomendações e propostas de trabalhos futuros:

- Aprofundar a pesquisa, aplicando e avaliando outros processos de software. A partir desta aplicação, adicionar a avaliação a tabela comparativa de recursos dos processos, proporcionando novos subsídios;
- O teste de unidade foi conceituado e explicado o seu funcionamento. A partir destas informações, explorar o uso de teste de unidade automatizado, e avaliar os resultados;
- Pode-se avaliar um conjunto de ferramentas de *refactoring* fornecendo procedimentos de utilização e recursos disponíveis;
- Elaborar um novo processo de software, baseado na ponderação dos aspectos que estabeleceram características de produtividade e qualidade.

O propósito destas recomendações resumem-se em tornar a avaliação de processos mais abrangente e com maior potencial de escolha.

REFERÊNCIAS BIBLIOGRÁFICAS

AMBLER, Scott W. **The Agile Edge: Duking it Out. Software Development**. Canadá, v.10, n.7, p.53-55, jul. 2002.

APPLETON, Brad. **Patterns and Software: Essential Concepts and Terminology**, fev, 2000. Disponível em: <http://www.enteract.com/~bradapp/docs/patterns-intro.html>. Acesso em: 12 ago. 2002.

ASTELS, David; MILLER, Granville; NOVAK, Miroslav. **Extreme Programming Explained**: Guia Prático. Rio de Janeiro: Ed. Campus, 2002.

BECK, Kent. **Extreme Programming Explained**: Embrace change. Reading, Massachusetts: Ed. Addison-Wesley, 2000.

BORCA, Oliver. **Project Engineering Process**, set, 2000. Disponível em: http://pst.web.cern.ch/PST/HandBookWorkBook/Handbook/SoftwareEngineering/IC-ProjectManagement.pdf>. Acesso em: 29 jul. 2002.

BORILLO, Dante. **The ICONIX approach**, set, 2000. Disponível em: <http://pst.cern.ch/PST/HandBookWorkBook/Handbook/SoftwareEngineering/iconix.html>. Acesso em: 09 mai. 2002.

CANTOR, Murray R. **Object-Oriented Project Management with UML**. New York: Ed. John Wiley & Sons, 1998. Cap.03, p.93-95.

COCKBURN, Alistair. **Crystal Methodologies**, 2001. Disponível em: <http://crystalmethodologies.org/index.html>. Acesso em: 18 abr. 2002.

CORDEIRO, Marco A. **Bate Byte 100 – Foco no Processo**, ago, 2000. Disponível em: <http://www.pr.gov.br/celepar/celepar/batebyte/>. Acesso em: 20 jun. 2002.

CYBIS, Walter. Apostila do LabUtil: Recomendações para Design Ergonômico de Interfaces. In: ___. **Técnicas de Projeto**. Programa de Pós-graduação em Engenharia de Produção. Florianópolis: UFSC, 1997. Cap.06, p.73-78.

EVANS, Gary K. Palm-Sized Process – Point-of-sale gets agile. **Software Development**. Canada, v.9, n.9, p.29-32, set. 2001.

FERNANDES, Acauan P.; LISBOA, Maria L. B. **Implementação Reflexiva de um Padrão de Projeto para Recuperação de Estado de Objetos. Revista do CEEI**. Bagé, v.5, n.8, p.42-43, ago. 2001.

FOWLER, Martin. Refactoring: Improving the Design of Existing Code. In: ___. **Principles in Refactoring**. Massachusetts: Addison-Wesley Longman, 1999. Cap.02, p.53-71.

FOWLER, Martin. **The New Methodology**, nov, 2001. Disponível em: <http://www.martinfowler.com/articles/newMethodology.html>. Acesso em: 18 abr. 2002.

FOWLER, Martin; FOEMMEL, Matthew. **Continuous Integration**, 2000. Disponível em: <http://martinfowler.com/articles/continuousIntegration.html>. Acesso em: 09 abr. 2002.

FOWLER, Martin; SCOTT, Kendall. **UML Essencial: Um breve guia para a linguagem-padrão de modelagem de objetos**. Porto Alegre: Ed. Bookman, 2000.

GAMMA, Erich et al. **Design Patterns: Elements of Reusable Object-Oriented Software.** Massachusetts: Addison-Wesley, 1995.

JEFFRIES, Ron. XP Magazine Contents: What is Extreme Programming?. **XProgramming.com – an Extreme Programming Resource**, nov, 2001. Disponível em: <http://www.xprogramming.com/xpmag/index.htm>. Acesso em: 11 mar. 2002.

HIGHSMITH, Jim. **Does Agility Work? Software Development**. Canadá, v.10, n.6, p.30-36, jun. 2002.

KULAK, Daryl; GUINEY, Eamonn. **Use Cases: Requirements in Context**. New York: ACM Press, 2000.

LARMAN, Craig. Utilizando UML e Padrões: Uma introdução à análise e ao projeto orientados a objetos. In: ___. **Casos de Uso: Descrevendo Processos**. Porto Alegre: Bookman, 2000. Cap.01, p.66-86.

LEACH, Ronald J. **Introduction to Software Engineering**. Florida: CRC Press LLC, 2000. Cap.01, p.1-27.

MARTINS, Vidal. **Bate Byte 89 - O Processo Unificado de Desenvolvimento de Software**, 1999. Disponível em: <http://www.pr.gov.br/celepar/celepar/batebyte/>. Acesso em: 15 jun. 2002.

MCBREEN, Pete. **Questioning Extreme Programming**. Indianapolis: Ed. Person Education, 2003.

MICROSOFT®. **Microsoft Solutions Framework: Solutions Development Discipline**. California, Silicon Valley Campus: Microsoft Corporation, 1996.

OMG®. Object Management Group **- Unified Modeling Language Specification (2.0)**, 2001. Disponível em: <http://www.omg.org>. Acesso em: 18 jul. 2002.

OSHIRO, Adriane et al. **Extreme Programming, um novo modelo de processo para o desenvolvimento de software**, jul, 2001. Disponível em: <http://www.icmc.sc.usp.br/~percival/>. Acesso em: 03 abr. 2002.

PASCOAL, Sandro M. et al. **Tutorial sobre ciclo de vida dos sistemas**, jun, 2001. Disponível em: <http://www.inf.cesumar.br/sandro/ciclo_vida.htm>. Acesso em: 30 abr. 2002.

PRESSMAN, Roger. S. **Software Engineering**: A practitioner's approach. New York: Ed. McGraw-Hill, 1997. p. 22-53.

REIS, Christian. **A commentary on the Extreme Programming development process**, ago, 2000. Disponível em: <http://www.async.com.br/~kiko/papers/xp/>. Acesso em: 04 abr. 2002.

REZENDE, Denis A. **Engenharia de Software e Sistemas de Informação**. Rio de Janeiro: Brasport, 2002. p.122-151.

ROBERTSON, James & Suzanne. **Mastering the Requirements Process**. [s.l.] : ACM Press ; Addison-Wesley, 1999. ISBN 0201 360462.

ROSENBERG, Doug; SCOTT, Kendall. **Use Case Driven Object Modeling with UML**: A Practical approach. Massachusetts: Addison-Wesley Longman, 1999.

ROSENBERG, Doug; SCOTT, Kendall. **Software Development Online: Give Them What They Want**, jun, 2001. Disponível em: <http://www.sdmagazine.com/>. Acesso em: 09 set. 2002.

SANTIAGO, Ricardo M. **XPers – Perguntas e Respostas**, 2000. Disponível em: <http://www.xpers.hpg.ig.com.br>. Acesso em: 09 set. 2002.

SCHNEIDER, Ricardo L. **Modelagem de Sistemas de Informação II**, dez, 1996. Disponível em: < http://www.dcc.ufrj.br/~schneide/>. Acesso em: 15 jun. 2002.

SCHWABER, Ken. **SCRUM Development Process** – Advanced Development Methods, 1997. Disponível em: <http://jeffsutherland.com/oopsla/schwapub.pdf.> Acesso em: 19 abr. 2002.

SCHWARTZ, Jonathan I. Construction of software. In: **Practical Strategies for Developing Large Systems**. Menlo Park: Ed. Addison-Wesley, 1975.

SEMEGHINI, Júlio. **Núcleo Softex Campinas – Missão Japão 2001**, nov, 2001. Disponível em: < http://www.cps.softex.br/relatorio.htm>. Acesso em: 24 jul. 2002.

SILVA, Alberto M. R.; VIDEIRA, Carlos A. E. **UML, Metodologias e Ferramentas Case**. Lisboa: Centro Atlântico, 2001.

SILVA, Edna L. da; MENEZES, Estera. **Metodologia da Pesquisa e Elaboração de Dissertação**. Florianópolis: Laboratório de Ensino a Distância da UFSC, 2001.

SOMMERVILLE, Ian. **Software Engineering**. Lancaster University, Lancaster: Ed. Addison-Wesley, 1995. p. 7.

SUTHERLAND, Jeff. **SCRUM Software Development Process**, jan, 2000. Disponível em: <http://jeffsutherland.com/scrum/index.html>. Acesso em: 19 abr. 2002.

THIRY, Marcello. **Processo de Desenvolvimento de Software com UML**, jun, 2001. Disponível em: <http://www.eps.ufsc.br/disc/procuml/>. Acesso em: 29 abr. 2002.

ULIANA, Ronie M. **SystemMetaphor by Objective Wiki**, 2001. Disponível em: <http://www.xispe.com.br/wiki/wiki.jsp?topic=SystemMetaphor>. Acesso em: 21 jun. 2002.

XP2002. **Extreme Programming Conference**, out, 2000. Disponível em: <http://www.xp2002.org/.>. Acesso em: 18 abr. 2001.

WAKE, William C. **Extreme Programming Explored**. Reading, Massachusetts: Ed. Addison-Wesley, 2002.

WELLS, Don. **Extreme Programming: A gentle introduction**, 2002. Disponível em: <http://www.extremeprogramming.org>. Acesso em: 18 mar. 2002.

WILLIAMS, Laurie et al. **IEEE Software – Strengthening the Case for Pair Programming**, jul/ago, 2000. Disponível em: <http://collaboration.csc.ncsu.edu/laurie/Papers/ieeeSoftware.pdf>. Acesso em: 25 jun. 2002.

WILLIAMS, Laurie. **Pair Programming, an Extreme Programming practice**, mar, 2001. Disponível em: <http://www.pairprogramming.com/>. Acesso em: 24 jun. 2002.

WUESTEFELD, Klaus. **Xispê – Extreme Programming**, 2001. Disponível em: <http://www.xispe.com.br/index.html>. Acesso em: 18 mar. 2002.

ZABEU, Sheila B. **PC World – XP: Bom senso ao extremo**, fev, 2002. Disponível em: <http://pcworld.terra.com.br/pcw/testes/programacao/0016.html>. Acesso em: 04 jul. 2002.