FUNDAÇÃO DE ENSINO “EURÍPIDES SOARES DA ROCHA”
CENTRO UNIVERSITÁRIO “EURÍPIDES DE MARÍLIA” – UNIVEM
FACULDADE DE CIÊNCIA DA COMPUTAÇÃO

WLADIMIR HENRIQUE MANGELADO BARBOSA

A METODOLOGIA EXTREME PROGRAMMING: UM ESTUDO DE CASO

MARÍLIA
2005

WLADIMIR HENRIQUE MANGELADO BARBOSA

A METODOLOGIA EXTREME PROGRAMMING: UM ESTUDO DE CASO

Monografia apresentada ao Curso de Ciência da Computação da Fundação de Ensino "Eurípides Soares da Rocha", Mantenedora do Centro Universitário Eurípides de Marília – UNIVEM, como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientadora:

Profa. Dra. Maria Istela Cagnin

MARÍLIA
2005

BARBOSA, Wladimir Henrique Mangelardo Barbosa.
A Metodologia Extreme Programming: Um Estudo de Caso / Wladimir Henrique Mangelardo Barbosa; orientadora: Maria Istela Cagnin. Marília, SP: [s.n], 2005.
70 f.

Monografia (Graduação em Ciência da Computação) – Centro Universitário Eurípides de Marília – Fundação de Ensino Eurípides Soares da Rocha.

1. *Extreme Programming*. 2. Desenvolvimento de Software. 3. Metodologia Ágil.

CDD: 005.1

WLADIMIR HENRIQUE MANGELADO BARBOSA

A METODOLOGIA EXTREME PROGRAMMING: UM ESTUDO DE CASO

Banca examinadora da Monografia para obtenção do grau de Bacharel em Ciência da Computação.

Resultado: _____ (_____)

ORIENTADORA: _____
Profª. Drª. Maria Istela Cagnin

1º EXAMINADOR: _____
Prof. Dr. Edmundo Sérgio Spoto

2º EXAMINADOR: _____
Prof. André Luiz Satoshi Kawamoto

Marília, ____ de _____ de 2005.

BARBOSA, Wladimir Henrique Mangelardo. **A Metodologia Extreme Programming: Um estudo de caso.** 2005. 70 f.
Dissertação (Graduação em Ciência da Computação) – Centro Universitário Eurípides de Marília – Fundação de Ensino Eurípides Soares da Rocha, Marília, 2005.

RESUMO

Este trabalho apresenta as características básicas dos métodos de desenvolvimento de software tradicionais, cascata e espiral. Além disso, apresenta a metodologia ágil de desenvolvimento de software *Extreme Programming* (XP), abordando as principais características dos seus valores e suas práticas. Também são discutidas as principais diferenças entre XP e as metodologias tradicionais. Finalmente é apresentado o desenvolvimento de uma aplicação baseada na metodologia XP como um estudo de caso.

Palavras-chave: *Extreme Programming,* Desenvolvimento de Software, Metodologia Ágil.

BARBOSA, Wladimir Henrique Mangelardo. **A Metodologia Extreme Programming: Um estudo de caso.** 2005. 70 f.
Dissertação (Graduação em Ciência da Computação) – Centro Universitário Eurípides de Marília – Fundação de Ensino Eurípides Soares da Rocha, Marília, 2005.

ABSTRACT

This research presents the basic characteristics of the traditional methods of software development, cascade and spiral. Moreover, the software development *Extreme Programming* (XP) agile methodology is presented, approaching the main characteristics of its values and practices. Also, the main differences between XP and the traditional methodologies are discussed. Finally, the development of an application based on the XP methodology is presented as a case study.

Keywords: *Extreme Programming*, Software Development, Agile Methodologies.

LISTA DE FIGURAS

LISTA DE QUADROS

SUMÁRIO

1 INTRODUÇÃO

1.1 Contexto

Não é de hoje que a indústria de software vem passando por grandes transformações e novos desafios, dentre os quais estão: a busca pela qualidade do software em um desenvolvimento no menor tempo possível e que atenda as necessidades dos clientes.

Para conseguir superar estes desafios, a indústria de software passou a dar mais valor a engenharia de software e a qualidade de software, com o intuito de atender as exigências do mercado. Como conseqüência disso, passou a utilizar metodologias de desenvolvimento de software, adotou métricas e padrões para se chegar aos níveis aceitáveis de qualidade, para prever custos e prazos em seus projetos. Porém, ainda são poucos os projetos que conseguem obter êxito em seu desenvolvimento, em que os prazos e os orçamentos estabelecidos sejam cumpridos e as necessidades do cliente sejam realmente atendidas.

Dados de 2004 (*Standish Group International*[1]) mostram que apenas 36% dos projetos de softwares atingiram sucesso (cumprimento de prazos, orçamentos e funcionalidades). Dentre os projetos pesquisados, as metodologias mais utilizadas eram cascata e espiral, tratadas como metodologias tradicionais neste trabalho.

Analisando os motivos para a baixa taxa de sucesso dos projetos de desenvolvimento de software cita-se: o elevado tempo entre cada fase do projeto; falta de conhecimento por parte do cliente da sua real necessidade; forte linearidade no desenvolvimento do projeto.

Como respostas as fragilidades do modelo tradicional, surgiram as metodologias para o desenvolvimento ágil de software, trazendo características como: enfoque nas pessoas e não

[1] *Standish Group International* empresa que realiza, desde de 1994, um levantamento bastante sofisticado sobre projetos de desenvolvimento de software em empresas norte-americanas.
Disponível em: http://www.standishgroup.com

em processos ou algoritmos; foco nas reais necessidades do cliente; ausência de linearidade no desenvolvimento do projeto; aprendizado do cliente em relação as suas necessidades durante o projeto e repassando suas novas necessidades a equipe de desenvolvimento.

Dentre as várias metodologias de desenvolvimento ágil está o *Extreme Programming*, metodologia esta que prima a qualidade do software desenvolvido, atendendo as reais necessidades do cliente e o cumprimento dos prazos estabelecidos.

1.2 Objetivos

Este trabalho tem como objetivo apresentar a metodologia ágil de desenvolvimento de software *Extreme Programming* (conhecido como XP), mostrando quais são os seus valores e as suas práticas. Além disso, visa também o desenvolvimento de uma aplicação como um estudo de caso, para que seja possível analisar melhor as características, os valores e as práticas que a metodologia *Extreme Programming* possue..

1.3 Estrutura da Monografia

O primeiro capítulo da monografia fornece ao leitor uma compreensão do contexto, objetivos e estrutura do trabalho desenvolvido.

No segundo capítulo são apresentados os dois métodos de desenvolvimento de software mais utilizados atualmente e quais são seus paradigmas.

O terceiro capítulo apresenta as principais características da metodologia de desenvolvimento *Extreme Programming*.

O quarto capítulo apresenta o estudo de caso, que é o desenvolvimento de uma aplicação baseada nos valores e práticas do XP, e também os resultados obtidos com o desenvolvimento.

No quinto capítulo são discutidas as conclusões obtidas com o desenvolvimento do trabalho.

2 MÉTODOS DE DESENVOLVIMENTO DE SOFTWARE TRADICIONAL

2.1 Considerações Iniciais

A engenharia de software é produto do conceito da engenharia de sistemas e de hardware. Ela abrange um conjunto de três etapas fundamentais: métodos, ferramentas e procedimentos. Estas etapas possibilitam ao gerente de projeto o controle do processo de desenvolvimento do software e ao desenvolvedor uma base para a construção de software de alta qualidade.

Este capítulo apresenta parte da revisão bibliográfica, precisamente no que se diz respeito sobre a engenharia de software tradicional. Na Seção 2.2 são apresentadas as características do método de desenvolvimento de software em cascata. Já na Seção 2.3 são apresentadas as características do método de desenvolvimento de software em espiral.

2.2 Desenvolvimento em Cascata

É o desenvolvimento mais antigo e o mais amplamente usado na engenharia de software. Também conhecido como ciclo de vida clássico, o modelo em cascata possui uma abordagem sistemática, eminentemente seqüencial ao desenvolvimento do software (PRESSMAM, 2000, p. 34). Essa abordagem sugere que o sistema seja construído linearmente, como mostrado na Figura 1, seguindo uma seqüência de fases:

1. Análise e engenharia de sistemas – esta fase envolve a coleta dos requisitos ao nível do sistema, uma análise global, com uma pequena quantidade de projeto e análise de alto nível.
2. Análise de requisitos do software – levantamento dos requisitos é feito pela equipe de desenvolvimento nesta fase. O analista deve compreender o domínio da informação para o software, assim como a função, desempenho e interfaces que serão necessárias. Os requisitos coletados, tanto para o sistema quanto para o software, são documentados e revistos com o cliente.
3. Projeto – com base nos requisitos coletados na fase de análise, a equipe projeta a arquitetura do sistema: estrutura de dados, detalhes procedimentais e caracterização de interface. Assim como os requisitos, o projeto também é documentado passando a fazer parte da configuração do software.
4. Codificação – baseando-se no projeto e na análise de requisitos, a equipe passa a implementar, ou seja, traduzindo o projeto em uma forma legível por máquina, as diversas partes do software.
5. Testes – com o código gerado inicia-se a fase de teste do programa para verificar se o sistema atende às necessidades especificadas pelo cliente. Caso descubram-se erros, as correções necessárias devem ser feitas.
6. Manutenção – Até o fim de sua vida, o software poderá sofrer alterações. Algumas mudanças serão feitas porque erros foram encontrados, porque o software deve ser adaptado por causa de mudanças externas, ou porque novas funcionalidades serão adicionadas ao software por solicitação do cliente.

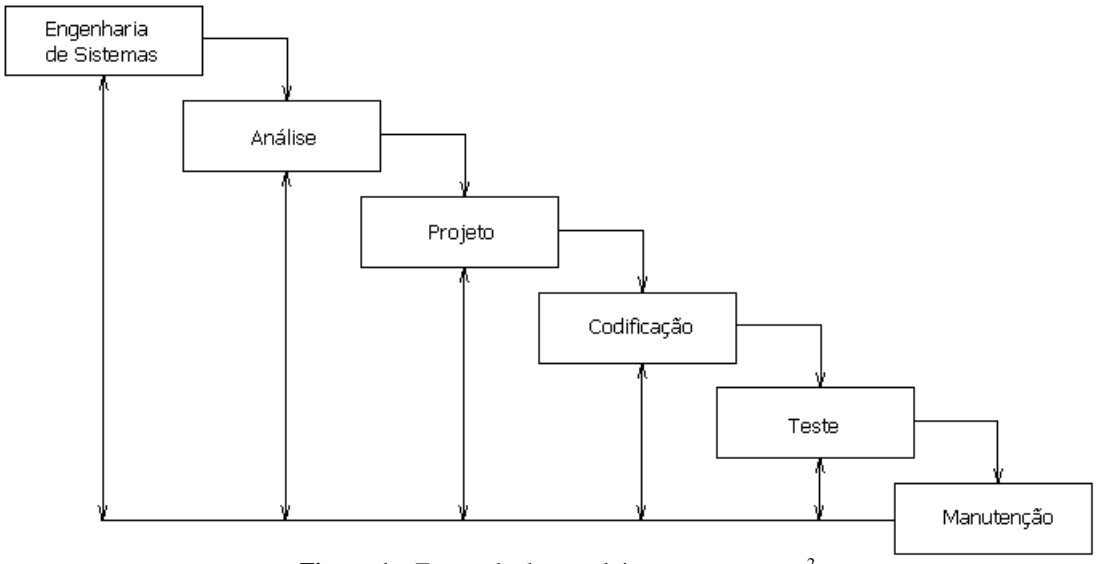

Figura 1 – Etapas do desenvolvimento em cascata[2]

2.3 Desenvolvimento em Espiral

O desenvolvimento em espiral é a abordagem mais realística para o desenvolvimento de sistemas. Essa metodologia utiliza uma abordagem "evolucionária", como descrita na Figura 2, em que versões mais completas do software são construídas progressivamente. Segundo Pressman (2000, p.38), cada iteração ao redor da espiral (partindo-se do centro e avançando para fora) passa por quatro importantes atividades:

1. Planejamento – determinação dos objetivos, alternativas e restrições.
2. Análise dos riscos – análise de alternativas e identificação/resolução dos riscos.
3. Engenharia – desenvolvimento do produto no "nível seguinte".
4. Avaliação feita pelo cliente – avaliação dos resultados da engenharia.

[2] Figura adaptada de: http://simat.inescn.pt/doc/doc_tecnicos/rer101/contexto/v101/introducao.html

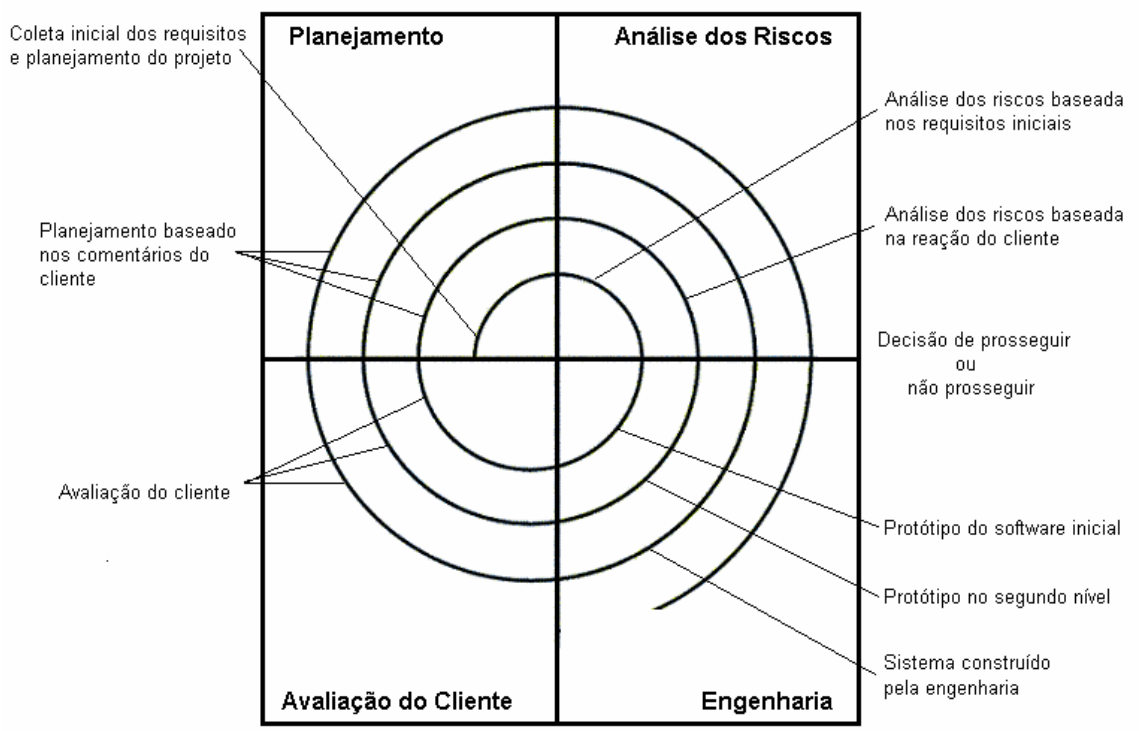

Figura 2 – Fases do desenvolvimento em espiral

Durante o primeiro giro em torno da espiral, na fase de planejamento, ocorre a coleta inicial dos requisitos e do software. Na fase seguinte existe uma análise dos riscos baseada nos requisitos iniciais fornecidos pelo cliente. Em seguida, na fase de engenharia, é construído um protótipo inicial do software que será avaliado pelo cliente na fase de avaliação.

O cliente avaliando o trabalho de engenharia apresenta sugestões para modificações que serão os requisitos utilizados nos próximos ciclos de planejamento, análise dos riscos e engenharia. Chegando novamente a fase de avaliação, o cliente re-avalia o produto gerando novos requisitos e o ciclo se inicia novamente até que o software atinja seu objetivo.

2.4 Considerações Finais

Neste capítulo foram descritos dois dos principais métodos de desenvolvimento de software: Cascata e Espiral.

Nota-se que ambos possuem etapas similares, porém o que os difere é como essas etapas são abordadas durante todo o processo de desenvolvimento de software.

3 *EXTREME PROGRAMMING*

3.1 Considerações Iniciais

Este capítulo apresenta a metodologia de desenvolvimento de software *Extreme Programming*. Na Seção 3.2 é apresentado uma visão geral sobre XP. Na Seção 3.3 mostram-se quais são os valores de XP e na Seção 3.4 abordam-se quais são as práticas que XP propõe.

3.2 Visão Geral do XP

Segundo Beck (2000), "*Extreme Programming* é uma metodologia ágil para equipes pequenas e médias desenvolvendo software com requisitos vagos e em constante mudança".

Extreme Programming (XP) é uma metodologia de desenvolvimento de software incremental, em que o software começa a ser implementado no inicio do projeto e ganha novas funcionalidades ao longo do tempo de desenvolvimento.

Na Figura 3 é apresentado um gráfico de desenvolvimento incremental, em que, o software ganha novas funcionalidades com o passar das semanas de desenvolvimento.

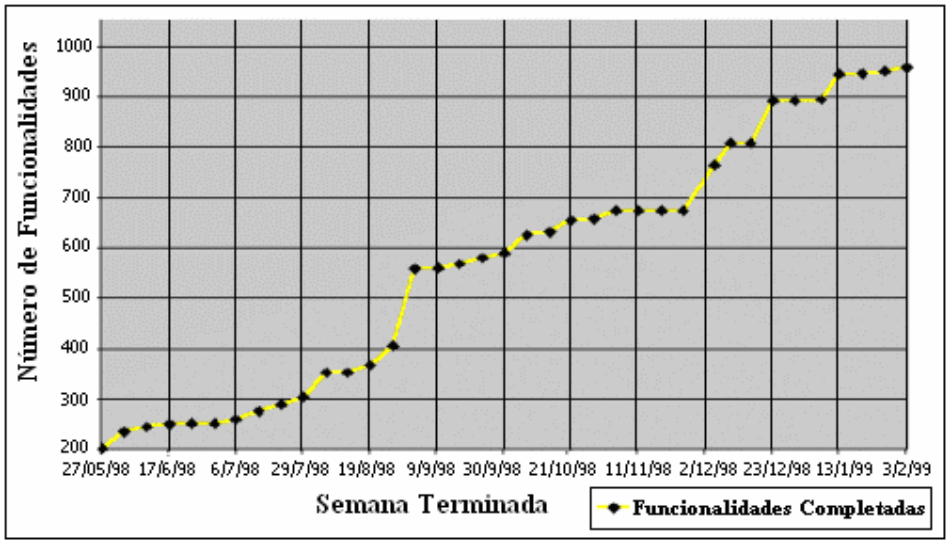

Figura 3 – Número de funcionalidades aumentando com as semanas[3]

O XP é uma metodologia de desenvolvimento de software que se enquadra em uma categoria conhecida como Metodologias Ágeis de Desenvolvimento. Esta categoria assume que as necessidades do cliente são por ele aprendidas à medida que ele é capaz de manipular o sistema que está sendo produzido. O cliente pode reavaliar as suas necessidades de acordo com o *feedback* gerado pelo software, podendo assim saber quais são as suas reais prioridades que deverão ser incorporadas ao sistema.

Com esse tipo de abordagem o cliente pode direcionar o desenvolvimento do software de modo que a equipe produza sempre aquilo que tem mais valor para o seu negócio. Isso faz com que o retorno do investimento no software feito pelo cliente seja rápido.

3.3 Valores do XP

Os valores em XP são as suas diretrizes, ou seja, eles irão definir as atitudes da equipe e as principais prioridades do método.

[3] Figura adaptada de: http://www.step-10.com/process/FDDandXP.html

Para uma equipe estar realmente utilizando o XP, ela deve respeitar e utilizar todos os valores e práticas. O XP baseia-se em quatro valores fundamentais. São eles: f*eedback*, comunicação, simplicidade e coragem.

3.3.1 *Feedback*

O *feedback* nada mais é do que a retro-alimentação que o cliente fornece à equipe de desenvolvimento quando aprende algo novo a respeito do sistema, seja sobre os requisitos ou sobre a forma como foram implementados. As metodologias ágeis de desenvolvimento de software, assim como o XP, se baseiam fortemente no *feedback* rápido (*fast feedback*).

Quanto mais rápido for gerado o *feedback* do cliente em relação ao software, mais rápida será a solução do problema ou a implementação de uma nova funcionalidade.

O processo de *feedback* está presente também no desenvolvimento de software tradicional. A diferença é que, no desenvolvimento de software tradicional, o *feedback* do cliente para a equipe de desenvolvimento demora muito para ser gerado, pois o cliente só terá contato com o software depois que ele estiver finalizado.

Isso já não acontece com o XP, uma vez que o ciclo de desenvolvimento ocorre várias vezes e o *feedback* do cliente para a equipe de desenvolvimento é gerado rapidamente. O fato de inúmeros *feedbacks* gerados pelo cliente faz com que o software convirja para um produto final que atenda às necessidades do mesmo.

3.3.2 Comunicação

A comunicação é um elemento extremamente necessário para que o *feedback* exista. Os integrantes da equipe de desenvolvimento e o cliente devem trocar informações e idéias para que o software ganhe forma e atinja os objetivos desejados.

Existem várias formas de ocorrer a comunicação entre as pessoas envolvidas no projeto, sendo uma mais rica que a outra. Partindo da comunicação mais rica para a mais pobre, temos a comunicação face-a-face, que além da interpretação da nossa fala, o interlocutor pode observar nossos gestos, a nossa expressão facial, o tom de voz, entre outros. Agora, em uma comunicação via telefone, o interlocutor continua tendo acesso à fala e ao tom de voz, porém os gestos e a expressão facial deixam de ser acessíveis a ele. Quando uma mensagem é transmitida através de um e-mail, bilhetes, mensagens instantâneas ou qualquer outro tipo de meio escrito, a perda dos elementos da comunicação é ainda mais acentuada, pois o leitor tem acesso a somente o conteúdo do texto.

O XP procura utilizar o meio mais rico de comunicação que existe: a conversa face-a-face. Explora tanto quanto possível a interação direta entre as pessoas envolvidas no projeto, diminuindo assim as falhas de comunicação e as falhas no desenvolvimento do software que são decorrentes dos problemas na comunicação.

3.3.3 Simplicidade

A simplicidade em XP é o valor que diz que as ações de cada membro da equipe de desenvolvimento devem ser simples e restritas para que logo após a execução de uma ação, possa-se obter *feedback* sobre ela rapidamente.

As funcionalidades solicitadas pelo cliente devem ser desenvolvidas com simplicidade, ou seja, desenvolver somente o suficiente para que atenda o pedido do cliente. Desta forma, os desenvolvedores não cometem um erro muito freqüente: o trabalho especulativo.

Trabalho especulativo é aquele trabalho que é realizado baseando-se em premissas incertas. Quando um desenvolvedor está implementando uma funcionalidade e se depara com uma dúvida, ele assume uma resposta que lhe parece razoável. Mas, freqüentemente, aquilo que ele assumiu como resposta verdadeira está incorreto e a única maneira de descobrir a resposta correta é levar a dúvida ao cliente, que nem sempre está por perto. A melhor alternativa diante desse tipo de problema é implementar a funcionalidade da forma mais simples possível.

Outra maneira de trabalho especulativo é quando o desenvolvedor assume que o cliente terá no futuro determinadas necessidades e implementa a funcionalidade de forma genérica. Na maioria das vezes essas generalizações são desnecessárias e o esforço gasto na implementação dessa funcionalidade foi em vão.

De acordo com Beck (1999), XP está fazendo uma aposta. Está apostando que é melhor fazer hoje algo simples e pagar pouco mais amanhã para mudá-lo, caso necessário, do que fazer hoje algo mais complicado que talvez nunca seja usado.

Portanto, o principal objetivo da simplicidade é evitar o desperdício de tempo e dinheiro em algo sobre o qual não temos certeza, evitando-se também o re-trabalho fruto do desconhecimento ou da precipitação.

3.3.4 Coragem

Sobre o valor “coragem” em XP, Teles (2004) apresenta a seguinte declaração:

> "O XP é uma metodologia de desenvolvimento de software nova e se baseia em diversas premissas que contrariam os processos tradicionais de desenvolvimento. Sendo assim, a adoção de XP exige que a equipe de desenvolvimento tenha coragem para: desenvolver o software de forma incremental, manter o sistema simples, permitir que o cliente priorize as funcionalidades, fazer os desenvolvedores trabalharem em par, investir tempo em refatoração, investir tempo em testes automatizados, estimar as estórias (funcionalidades do sistema descrita em cartões) na presença do cliente, expor o código a todos os membros da equipe, integrar o sistema diversas vezes ao dia, adotar um ritmo sustentável, abrir mão de documentações que servem como defesa, propor contratos de escopo variável e propor a adoção de um processo novo" (TELES,2004, p. 50).

A seguir será detalhado cada tipo de coragem.

- **Coragem para desenvolver o software de forma incremental**

Utilizando o método em espiral, o sistema é implementado de maneira incremental, em que novas funcionalidades são adicionadas fazendo com que, por motivos de acomodação, partes já implementadas sejam alteradas.

Isso requer coragem dos desenvolvedores pois erros poderão aparecer em partes que estavam funcionando corretamente.

- **Manter o sistema simples**

A recomendação que o XP faz aos desenvolvedores é que implemente as funcionalidades de maneira simples, atendendo somente aquilo que foi solicitado e que se conhece no presente, evitando assim generalizações. Fazer isso demanda muita coragem, porque a equipe deve crer que ela será capaz de implementar possíveis necessidades futuras que ela já consegue visualizar no presente, porém não foram especificadas pelo cliente.

- **Permitir que o cliente priorize as funcionalidades**

Normalmente, quem decide a ordem de implementação das funcionalidades dos sistemas são os desenvolvedores, pois levam em consideração algumas relevâncias técnicas,

como por exemplo, dependência de funcionalidades. Entretanto, essa ordenação lógica dos desenvolvedores nem sempre é que irá gerar maior valor para o cliente.

Por isso, no XP, a ordem de implementação das funcionalidades, que são descritas através de estórias registradas em pequenos cartões, são especificadas pelo cliente, pois ele conhece as necessidades do negócio e sabe o que lhe trará maior retorno a cada momento. Isso exige coragem por parte da equipe, em permitir que o cliente defina as prioridades mesmo não sendo a ordem mais conveniente em relação ao ponto de vista dos desenvolvedores.

- **Fazer os desenvolvedores trabalharem em par**

É necessário muita coragem por parte da equipe de desenvolvimento para adotarem essa técnica, pois, por ser uma técnica incomum nos projetos tradicionais de desenvolvimento e por deixar a impressão que encarece o projeto, uma vez que dois desenvolvedores são colocados para fazer o trabalho que um único poderia fazer (mais adiante, na seção 3.4.4, será discutido a questão da programação em par e sobre a impressão de que essa técnica encarece o projeto).

- **Investir tempo em refatoração**

De acordo com Martin Fowler (1999), "refatoração é o processo de alterar um sistema de software de tal forma que ele não altere o comportamento externo do código e melhore a sua estrutura interna. Essa é uma forma disciplinada de limpar o código que minimiza as chances de introdução de bugs".

Essa técnica é indispensável para que o código do sistema possa ser desenvolvido de forma simples e clara ao longo do projeto e é uma prática essencial do XP. Um código de alta qualidade se faz necessário quando um sistema é desenvolvido de forma incremental, pois só assim, a equipe será capaz de alterá-lo continuamente.

Inicialmente a refatoração parece ser um desperdício de tempo, mas na verdade ela gera retornos a médio e a longo prazo, permitindo uma rapidez nas alterações do sistema. Portanto, é preciso ter coragem para adotar uma técnica que, aparentemente, leva a um consumo maior no tempo de desenvolvimento do sistema.

- **Investir tempo em testes automatizados**

Outra técnica vista como um desperdício de tempo é o teste automatizado. O XP recomenda a criação de testes de unidades para as classes de sistema e testes de aceitação a partir das estórias.

A criação destes testes pelos desenvolvedores não é um gasto desnecessário de tempo como parece, muito pelo contrário, é um investimento de tempo que trará retorno a médio e a longo prazo. A criação dos testes previne a equipe da ocorrência e permanência de falhas no sistema, o que evita que a mesma gaste tempo depurando o software na procura de erros. É necessário coragem da equipe para investir tempo em testes automatizados e compreender que ela não deve apenas pensar no curto prazo, pois em se tratando de produtividade, ela deve olhar o projeto como um todo.

- **Estimar as estórias na presença do cliente**

No XP, as estórias são estimadas, ou seja, são definidas a quantidade de tempo que as estórias irão levar para ser desenvolvidas pela equipe sempre na frente do cliente, para que assim ele possa tirar dúvidas, tornando assim as estórias mais precisas. Essa prática deixa muita gente, principalmente os gerentes de projeto, com medo que o cliente perceba insegurança dentro da equipe. Por isso é necessário coragem da equipe em expor suas dúvidas ou inseguranças para o cliente para que ele possa ajudar os desenvolvedores.

- **Expor o código a todos os membros da equipe**

A técnica em programação em par e a prática do código coletivo (em que todos os desenvolvedores têm acesso a todas as partes do código) expõe o código de um desenvolvedor a todas as outras pessoas da equipe de desenvolvimento, deixando-o assim, sujeito a eventuais críticas e avaliações. Por isso, o desenvolvedor deve ter coragem para expor seu código e ter humildade e serenidade para tratar eventuais críticas como um aprendizado.

- **Integrar o sistema diversas vezes ao dia**

Como o XP trabalha com integração contínua, ou seja, sugere que os desenvolvedores integrem todos os módulos do sistema várias vezes ao dia, surge o risco de que partes do sistema não funcionem corretamente, devido a erros decorrentes da integração. Por isso, é preciso ter coragem da equipe para integrar continuamente e também utilizar práticas como uso intensivo dos testes.

- **Adotar um ritmo sustentável**

É preciso coragem para contrariar a lógica tradicional, em que se recomenda que pessoas trabalhem mais que o normal para alcançar produtividade mais alta e permitir que os desenvolvedores trabalhem apenas oito horas por dia e evitem fazer horas extras.

- **Abrir mão de documentações que servem como defesa**

A equipe de desenvolvimento precisa ter muita coragem para abandonar os documentos que servem como defesa e encarar a documentação de forma leve e gerar apenas documentos relevantes para o projeto. Muitas equipes de desenvolvimento não jogam para perder, e por isso, utilizam de todo o tipo de documentação que possa ser usada para livrar a

equipe de qualquer responsabilidade. O XP tem uma abordagem diferente, jogando para ganhar, preocupada em produzir o melhor software possível.

- **Propor contrato de escopo variável**

Como a maioria dos projetos de desenvolvimento de software é contratada de escopo fechado, onde impedem que o cliente faça alterações no escopo ao longo do projeto, é necessário muita coragem para adotar uma prática pouco disseminada como é o caso dos contratos de escopo variável. O contrato de escopo variável permite que o cliente faça alterações no escopo, alterações que foram geradas principalmente do *feedback* do cliente e de seu aprendizado no sistema, fazendo com que se construa um software que atenda plenamente as suas necessidades.

- **Propor a adoção de um processo novo**

Por ser uma metodologia de desenvolvimento de software nova e principalmente por parecer seguir na direção oposta dos desenvolvimentos de softwares que as equipes estão habituadas a trabalhar, é necessário muita coragem para adotar o XP, apostando em novas premissas.

3.4 Práticas do XP

As práticas são um conjunto de atividades que deverão ser seguidas pelas equipes que utilizam XP.

O XP baseia-se nas seguintes práticas: cliente presente, jogo do planejamento, *stand up meeting*, programação em par, desenvolvimento guiado pelos testes, refatoração, código coletivo, código padronizado, *design* simples, metáfora, ritmo sustentável, integração

contínua e *releases* curtos. Muitas das práticas são polêmicas e aplicadas isoladamente não fazem o menor sentido. Porém, há uma confiança muito grande na sinergia entre elas, onde a fraqueza de uma é compensada por outra e assim fecha-se um ciclo fortemente dependente.

3.4.1 Cliente Presente

Por ser o XP uma metodologia de desenvolvimento ágil, que se baseia no paradigma em espiral, onde o tempo de *feedback* é fundamental, ele sugere que o cliente esteja sempre presente durante o dia-a-dia do projeto, pois sua ausência pode significar o fracasso do projeto.

A participação do cliente evita que mudanças bruscas sejam feitas ao longo do projeto, pois permite que o projeto seja conduzido através de uma série de pequenos ajustes.

As principais razões que tornam essencial a presença do cliente são:

- Respostas rápidas para eventuais dúvidas de implementação, evitando assim o trabalho especulativo que ocorre quando a equipe não consegue ter as dúvidas respondidas e assume premissas que podem gerar funcionalidades de forma incorreta.
- Quando a implementação de uma funcionalidade está terminada, os desenvolvedores podem rapidamente mostrar o resultado para o cliente, permitindo assim que o cliente tenha *feedback* rápido. Com esse *feedback*, o cliente pode informar aos desenvolvedores alguns detalhes que devem ser modificados para que a funcionalidade fique a mais fiel possível da sua idéia original.

Além disso, a constante presença do cliente contribui para melhorar o relacionamento entre as partes envolvidas, aumentando assim a confiança entre eles.

Embora a presença do cliente seja altamente recomendável ao longo do desenvolvimento, é difícil encontrar um cliente que possua disponibilidade para estar sempre presente, pois na maioria das vezes, ele é uma pessoa ocupada.

Para tentar conciliar a presença do cliente no desenvolvimento do sistema com seus outros afazeres, existem algumas estratégias que podem ser tomadas. Por exemplo, se existir a possibilidade, a equipe de desenvolvimento pode se instalar junto a empresa do cliente e transferir o mesmo para a sala de projeto. Assim a equipe e o cliente estarão em constante contato.

Em alguns casos não é possível deslocar a mesa de trabalho do cliente para a sala onde estará a equipe de desenvolvimento, mas é possível disponibilizar para a equipe uma sala no mesmo prédio que o cliente se encontra. Nestes casos, é necessário acertar uma agenda mínima que garanta a participação do cliente durante alguns momentos por dia.

Existem casos onde não será possível disponibilizar uma sala para a equipe de desenvolvimento no mesmo prédio do cliente. Nestes casos uma solução pode ser montar um escritório próximo ao prédio para alojar a equipe. Em casos assim, é ideal que o cliente visite a equipe de desenvolvimento pelo menos uma vez ao dia para responder as dúvidas levantadas pela equipe.

Teles (2004, p. 66) diz que quanto mais freqüente for a presença do cliente, melhor. Se realmente não for possível ter a presença do cliente em um dado projeto, é provável que o XP não seja o processo de desenvolvimento para ele. O XP realmente precisa de participação do cliente no desenvolvimento.

3.4.2 Jogo do Planejamento

Beck e Fowler (2001, **apud** Teles, 2004) dizem que o XP utiliza o planejamento para assegurar que a equipe de desenvolvimento esteja sempre trabalhando naquilo que irá gerar maior valor para o cliente a cada dia de trabalho. Esse planejamento é feito diversas vezes ao longo do projeto, para que todos tenham a oportunidade de revisar as prioridades.

- **Escrevendo estórias**

Todas as funcionalidades que o sistema possui são levantadas através de estórias que são registradas em pequenos cartões, como mostrado na Figura 4.

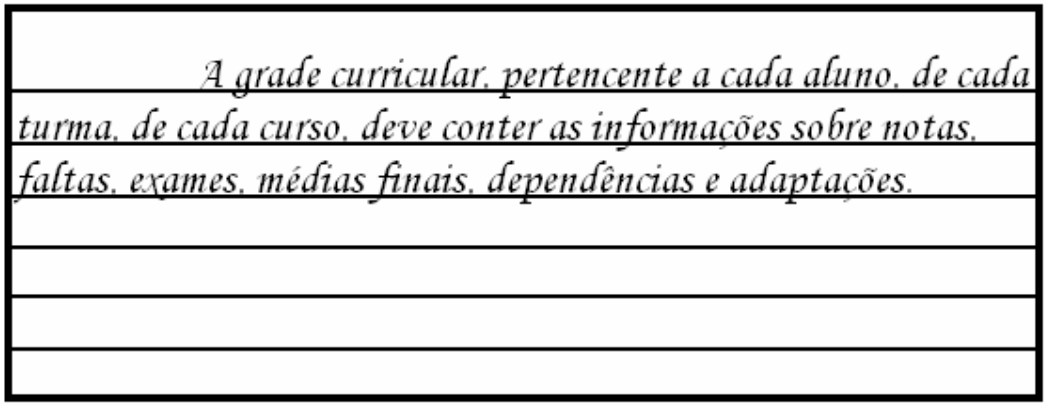

Figura 4 – Uma estória escrita em um cartão[4]

As estórias sempre são escritas pelo cliente, com suas próprias palavras. Não devem existir regras para escrever uma estória, a não ser respeitar o espaço do cartão, para que assim as estórias sempre fiquem da forma mais simples possível.

As estórias são extremamente importantes pois, ao escrever uma estória, o cliente é obrigado a pensar melhor na funcionalidade, formalizando o pensamento e analisando o assunto sobre o qual irá tratar. Elas devem ser simples e curtas, uma vez que as estórias têm o objetivo de servir como um lembrete.

[4] Esta estória, que trata da grade curricular do aluno e seus atributos, foi descrita pelo usuário do sistema desenvolvido, durante este trabalho. Maiores detalhes sobre o desenvolvimentos são obtidos no capítulo 4.

Ao escrever um cartão, o cliente cria uma espécie de vínculo psicológico com ele, mostrando ter uma responsabilidade sobre aquilo que está sendo solicitado à equipe.

O cartão também tem a função de fazer o cliente compreender que existe um certo custo em tudo aquilo que está sendo solicitado. Assim é possível que o cliente saiba selecionar aquilo que está sendo solicitado a cada momento, ou seja, para que ele possa priorizar.

- **Tarefas**

Os cartões são utilizados pela equipe de desenvolvimento para saber quais são as funcionalidades desejadas pelo cliente. Os desenvolvedores escolhem quais serão as estórias que serão implementadas a cada dia de trabalho. Porém, em projetos grandes, em que as estórias costumam consumir muito tempo de trabalho, como uma ou mais semanas, os cartões são divididos em tarefas. As tarefas são registradas em novos cartões que são distribuídos entre os desenvolvedores, assim eles escolhem as tarefas que irão implementar e não estórias.

Teles (2004, p. 72) considera preferível adotar essa abordagem sempre que for viável no projeto, visto que estórias menores costumam ser mais simples de estimar e administrar.

- **Estimando as estórias**

Toda estória deve ser estimada para que o cliente saiba o custo previsto para cada uma delas. Para estimar cada estória, a equipe de desenvolvimento deve adotar uma unidade de medida. Essa unidade de medida deverá ser usada para todas as estórias.

> "O XP utiliza o conceito de dia ideal de desenvolvimento, que representa um dia no qual o desenvolvedor só trabalha na implementação de funcionalidades, sem ter que se preocupar com nenhum atividade extra. No XP, todas as estimativas são feitas considerando dias ideais de trabalho. Ou seja, quando um desenvolvedor estima, ele deve fazer a seguinte pergunta: se eu tiver o dia todo apenas para implementar esta estória, sem ter que atender ao telefone, participar de reuniões etc, quantos dias eu levarei para finalizá-la?" (TELES, 2004, p. 73)

- **Usando pontos para estimar**

O ponto é a unidade de medida usada para estimar, para facilitar a comunicação na equipe e refere-se ao termo dia ideal de desenvolvimento. Inicialmente, ponto representa um dia ideal de desenvolvimento, contudo, em projetos onde as estórias sejam muito longas, é comum adotar que um ponto seja equivalente a uma semana ideal de desenvolvimento.

O ponto representa o tempo de desenvolvimento ideal de um ou mais desenvolvedores, ou seja, período no qual se pode implementar uma certa funcionalidade sem ter que se preocupar com outras atividades.

Além de simplificar a comunicação entre os membros da equipe de desenvolvimento, o ponto é usado para registrar as estimativas nos cartões de estórias, onde em cada cartão, a equipe registra a quantidade de pontos estimados para aquela funcionalidade. Geralmente os pontos são registrados no canto superior esquerdo, enquanto o canto direito é usado para registrar quantos foram os pontos realmente gastos na implementação daquela funcionalidade.

Na Figura 5 é apresentado um exemplo de um cartão contendo uma estimativa.

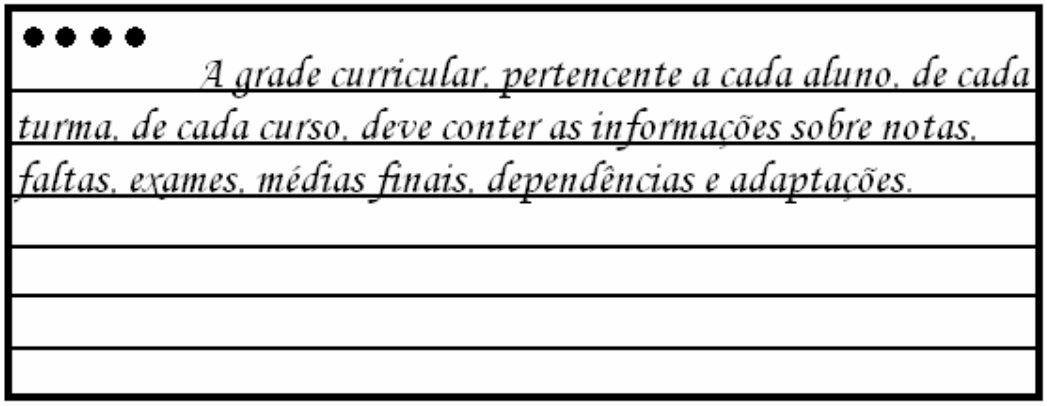

Figura 5 – Um cartão contendo uma estimativa

- **Estimando por comparação**

Beck (2000), diz que uma maneira simples e eficaz de se estimar o tamanho de uma estória é procurar uma estória similar que já foi estimada. Sempre que possível, a equipe deve procurar estimar por comparação. Ou seja, buscar cartões que já tenham sido

implementados e que possuam funcionalidades semelhantes ao que está sendo estimado. Caso esse cartão seja encontrado, a equipe deve utilizar os pontos realmente consumidos pelo cartão como sendo a estimativa do novo cartão.

- **Estimando em equipe**

O XP recomenda que as estimativas sejam feitas em equipe, pois tende a ser mais precisa, uma vez que se baseia em um conjunto maior de conhecimentos e experiências do que se fosse estimado por uma única pessoa.

É muito importante que o cliente esteja presente na elaboração das estimativas, visto que permite que a equipe tire eventuais dúvidas sobre as estórias, aumentando a qualidade das estimativas e evitando que os desenvolvedores assumam premissas que podem estar incorretas.

Um aspecto negativo em se estimar com a presença do cliente é que é comum que ele faça pressão sobre a equipe de desenvolvimento para que reduza o tempo das estimativas. Para contornar esse problema é necessário um trabalho constante de conscientização para que o cliente não faça pressão durante a geração das estimativas e, se a pressão vier, a equipe deve ter a disciplina de ignorá-la e informar a estimativa correta para o cliente.

- **Planejando os *releases***

O XP trabalha com o objetivo de gerar valor, ao longo de todo o projeto, para o cliente. Para que isso ocorra é necessário que o software seja entregue de forma incremental, para que assim, a cada entrega o cliente possa começar a utilizar o sistema, obtendo dessa maneira os benefícios que ele oferece. Estas entregas recebem o nome de *release*.

Em XP, os projetos são divididos em *releases*, onde cada *release* implementa um certo conjunto de funcionalidades que possui um valor bem definido para o cliente. Um

projeto de oito meses pode ser dividido em quatro *releases*, onde cada *release* terá oito semanas, por exemplo.

Projetos em XP costumam trabalhar com o conceito de *releases* pequenos, de curta duração, em torno de dois meses. Segundo Beck (1999), cada *release* deve ser tão pequeno como possível, contendo as exigências mais valiosas do negócio para o cliente. Beck (1999) também afirma que o *release* tem que fazer o sentido ao todo, isto é, não se pode executar a metade de uma funcionalidade e enviá-la, para finalizar o *release*.

Isto é importante, pois o cliente usufrui do sistema rapidamente, que gerando um valor de retorno, serve de incentivo no prosseguimento do projeto. A equipe de desenvolvimento recebe *feedback* dos usuários finais do sistema, o que permite que ele faça ajustes para aprimorar a qualidade dos próximos *releases*.

- **Priorizando as estórias de cada *release***

Como dito anteriormente, o cliente e a equipe de desenvolvimento dividem o projeto em *releases* de tamanhos fixos e que sejam pequenos. Após a identificação do valor que será gerado em cada *release*, é necessário que o cliente estabeleça uma ordem dos mesmos, com base em suas necessidades e também considerando aspectos técnicos apresentados pela equipe de desenvolvimento.

Uma vez determinados os objetivos de cada *release*, o cliente deve distribuir as estórias referentes a cada um deles. Teles (2004, p. 78) diz que não é necessário escrever todas as estórias do sistema no início do projeto. Diz também que é recomendável que o cliente se concentre apenas nas estórias deste *release*, deixando as estórias dos *releases* posteriores para o futuro, visto que tratar deles muito no início normalmente é um trabalho especulativo.

No início de cada *release*, a equipe informará ao cliente o número de pontos que ela acredita que será capaz de implementar, ou seja, informando ao cliente a quantidade de pontos que ele terá a sua disposição. Caso o número de estórias referentes àquele *release* superar a quantidade de pontos disponível, o cliente deverá indicar aquelas que ele julga mais prioritárias. As estórias que ficaram de fora da implementação poderão ser alocadas nos próximos *releases*.

Ao contrário dos processos tradicionais de software, o XP não busca proteger a equipe de desenvolvimento das mudanças, pois permite que, durante um *release*, o cliente altere as estórias se considerar necessário. Ele pode criar estórias que substituirão estórias existentes, pode remover estórias já previstas, ou pode alterar o conteúdo de algumas previamente identificadas. Assim, o cliente incorpora o seu aprendizado no sistema, gerando benefícios diretos para si próprio.

Nos processos tradicionais, onde o escopo é fechado no início do projeto, o cliente terá que esperar até o final do processo para poder incorporar qualquer mudança no sistema, tendo que renegociar o contrato com a equipe de desenvolvimento, renegociações que costumam ser caras.

Mesmo que seja pequeno, um *release* dura um tempo muito longo para o desenvolvimento em XP. Por isso, cada *release* é dividido em iterações, que são unidades de tempo menores.

- **Planejando iterações**

Basicamente, uma iteração é um pequeno espaço de tempo dedicado para a implementação de um conjunto de estórias. Geralmente, os projetos em XP utilizam iterações de duas semanas, mas dependendo das características do projeto, as iterações podem variar de uma a três semanas.

Para facilitar a gerência e o planejamento, uma iteração deve, de preferência, manter-se inalterada ao longo do projeto, uma vez que foi definido o seu tamanho.

O início de cada iteração é caracterizado por uma reunião. Essa reunião recebe o nome de reunião de planejamento da iteração e é nesta reunião que o cliente e os desenvolvedores definem as estórias que serão implementadas na iteração que se inicia. Para que esta reunião ocorra é necessário saber a velocidade da iteração, que significa quantos pontos espera-se que a equipe será capaz de implementar.

É necessário efetuar algumas operações simples para saber a quantidade de pontos ideais que podem ser implementados em uma iteração. No Quadro 1 é apresentado um exemplo, em que, assumindo-se que um ponto equivale a um dia ideal de um par de desenvolvedores, uma iteração de duas semanas e uma equipe de seis desenvolvedores, temos:

- Tamanho da iteração = 2 semanas = 10 dias úteis
- Deve-se descontar:
  - 1 dia útil para a reunião de planejamento da iteração
  - 1 dia útil para a reunião de encerramento da iteração
- Dias úteis disponíveis para o desenvolvimento = 10 – 2 = 8
- Número de desenvolvedores = 6 = 3 x 2 = 3 pares
- 1 par/dia = 1 ponto
- 1 par em 8 dias = 1 x 8 = 8 pontos
- 3 pares em 8 dias = 3 x 8 = 24 pontos

Quadro 1 – Cálculo da quantidade de pontos ideais para uma iteração

Com a iteração de duas semanas, ela terá dez dias úteis, sendo um reservado para a reunião de planejamento da iteração (realizada no primeiro dia da iteração) e outro reservado para reunião de encerramento da iteração (realizada no último dia da iteração).

É essencial que toda a equipe participe efetivamente das reuniões (de planejamento e encerramento das iterações), pois se trata de um momento onde existe uma grande interação com o cliente, no qual ele apresenta diversas informações que a equipe usará ao longo da iteração.

Um acompanhamento do número de pontos que a estória está consumindo deve ser feito pelos desenvolvedores porque, ao final da iteração, a equipe irá somar a quantidade total de pontos ideais implementados. Esta quantidade representa a velocidade da iteração e é o número de pontos que a equipe irá oferecer para o cliente na iteração que se inicia.

Para estimar quantos pontos a equipe irá implementar na iteração que se inicia ela soma quantos pontos foram efetivamente implementados na iteração anterior e assume que a equipe terá a capacidade de implementar a mesma quantidade de pontos na nova iteração. Depois a equipe informa ao cliente sobre a quantidade de pontos que ela será capaz de implementar que, por sua vez, distribui os cartões do *release* em questão e escolhe os cartões mais prioritários. Somados, totalizam o número de pontos oferecidos pela equipe para a iteração, o cliente informa a ordem que deseja que os cartões sejam implementados.

Ao final da iteração, se a equipe implementar menos pontos que o estimado significa que a velocidade diminuiu e serão oferecidos menos pontos na iteração seguinte (as estórias que não foram implementadas serão implementadas na iteração seguinte, caso o cliente queira). Porém, se a equipe tiver terminado de implementar os pontos antes do fim da iteração, ela deverá solicitar mais estórias ao cliente. Isso indica que a velocidade aumentou e serão oferecidos mais pontos para o cliente na próxima iteração.

Além das iterações estarem contidas nos *releases*, existe uma diferença importante entre *releases* e iterações. Nos *releases*, o cliente pode alterar as estórias que serão implementadas, porém no caso da iteração, isso não é permitido. Depois que um conjunto de estórias foram priorizadas para uma iteração, a equipe irá trabalhar somente naquelas estórias e não aceitará que o cliente faça mudanças. Ele terá que esperar o fim da iteração para solicitar mudanças.

- **Dependências técnicas**

Algumas vezes, a ordem que o cliente deseja que as estórias sejam implementadas é afetada por dependências técnicas que a equipe deve identificar e lhe apresentar, o que impede que o cliente obtenha o desenvolvimento na ordem em que ele gostaria.

O XP recomenda que a ordem indicada pelo cliente seja respeitada pela equipe de desenvolvimento, visto que, na maioria dos casos, isso é possível, desde que a equipe implemente algumas simplificações, contornando assim as dependências técnicas.

> "Quando a equipe respeita a ordem, ela gera *feedback* rápido para o cliente sobre uma estória que é prioritária para ele. Desta forma, ele poderá utilizar a funcionalidade no sistema e, eventualmente, aprender algo novo que irá gerar modificações que farão com que a funcionalidade se torne mais útil e gere maior valor" (TELES, 2004, p. 84).

Entretanto, em algumas situações é impossível contornar as dependências técnicas. Quando isso acontecer, é necessário mostrar a dificuldade ao cliente durante a reunião de planejamento da iteração e será preciso negociar uma ordem que viabilize a implementação das funcionalidades desejadas com agilidade.

- **Encerrando uma iteração**

O último dia da iteração, como dito anteriormente, é reservado para a reunião de encerramento. É nesta reunião que o cliente faz todos os testes de aceitação que foram escritos para as estórias da iteração. O objetivo é que o cliente valide todo o sistema e verifique se o resultado da iteração está satisfatório. Utilizando o sistema, o cliente pode aprender mais sobre os requisitos e detectar erros.

- **Encerrando um *release***

O final de um *release* se dá ao final da última iteração prevista para ele. Depois disso, o sistema é colocado em produção para que todos os usuários passem a ter acesso as suas funcionalidades. Quanto mais entrada em produção o projeto tiver, melhor será o seu resultado para o usuário, uma vez que a cada *release* eles terão a oportunidade de gerar *feedback* para a equipe de desenvolvimento, que através desse *feedback*, pode direcionar seus esforços para fazer um sistema cada vez mais adequado as necessidades do cliente.

3.4.3 *Stand Up Meeting*

Um dia de trabalho de uma equipe XP começa sempre com um *stand up meeting*, que significa "reunião em pé" em português. Refere-se a uma reunião rápida que envolve todos os membros da equipe de desenvolvimento. Esta reunião leva o nome de reunião em pé pois a idéia é que seja uma reunião rápida, em torno de dez minutos.

O *stand up meeting* tem que ocorrer todos os dias e não de tempos em tempos. Elas devem fazer parte do dia-a-dia da equipe de desenvolvimento.

O *stand up meeting* serve, primeiramente, para que todos os membros da equipe comentem rapidamente sobre o trabalho que executaram no dia anterior. Assim, a equipe toda toma parte sobre o andamento do projeto naquele dia. A reunião é uma oportunidade que todos têm de apresentar as dificuldades que surgiram e as soluções que encontraram para resolvê-las. Dessa forma, no aparecimento de problemas semelhantes, todos saberão o que fazer, ou pelos menos, saberão a quem recorrer.

A reunião também é utilizada para decidir o que será feito no dia que se inicia. No *stand up meeting*, a equipe decide em conjunto quais cartões serão tratados naquele dia e

quem cuidará de cada cartão. Outras atividades que não estão diretamente ligadas aos cartões também são divididas e priorizadas.

Embora seja aconselhável que os participantes da reunião estejam em pé, isso não é obrigatório. O importante mesmo é que as pessoas se concentrem na reunião, que todos participem ativamente dela, que seja realizada todos os dias e que seja sempre objetiva e rápida.

3.4.4 Programação em Par

Em XP, os desenvolvedores nunca trabalham sozinhos, mas sempre em pares. Trata-se da programação em par.

A programação em par é uma técnica onde, diante de cada computador, existem sempre dois desenvolvedores trabalhando juntos em um mesmo problema, para produzir o mesmo código. A pessoa que assume o teclado e digita é chamada de condutor, a outra que acompanha o condutor é chamada de navegador e faz um trabalho de estrategista.

A técnica da programação em par permite que o código seja revisado permanentemente enquanto é construído. Enquanto o condutor digita os comandos, o navegador está sempre revisando o código e tentando evitar que eventuais erros do condutor passem despercebidos, ou seja, faz com que as correções sejam aplicadas imediatamente, assim que os erros apareçam no código.

Outro benefício que a programação em par proporciona é a influência que ela tem na modelagem de uma solução. Quando uma solução é modelada por dois programadores ela tende a ser mais eficiente do que quando modelada por uma só pessoa. Isso acontece principalmente porque ao modelar uma solução, um programador utiliza um conjunto de

conhecimentos e experiências que acumulou ao longo de sua carreira e dois programadores utilizam dois conjuntos diferentes de conhecimentos e experiências.

Raramente a solução idealizada somente por um programador é adotada no sistema quando se programa em par. A maioria das soluções é fruto da conversa entre os parceiros, gerando idéias que possuem contribuições de ambas as partes, utilizando o melhor de cada idéia.

Além de mais eficientes, as soluções adotadas costumam ser mais simples. Se um parceiro sugere algo muito complicado, o outro rapidamente percebe a complexidade da idéia e poderá sugerir simplificações. Por isso, o resultado final costuma ser uma solução mais clara e, ao mesmo tempo, mais simples.

- **Os efeitos sobre a produtividade**

A princípio, a idéia da programação em par é estranha, pois enquanto um desenvolvedor programa o outro fica parado. É claro que não é isso que ocorre porém é isso que muita gente acaba interpretando de maneira incorreta.

Na verdade, ambos estão programando juntos, porém um programador digita, enquanto o outro não digita.

> "A programação em par pressupõe uma comunicação contínua entre os desenvolvedores que formam o par. Através da conversa, eles discutem as melhores idéias para a resolução de cada problema. Portanto é absolutamente incorreto acreditar que um trabalha enquanto o outro fica parado. Ambos estão trabalhando o tempo todo" (Teles, 2004, p. 91).

Ainda assim uma dúvida fica no ar. Não seria um desperdício de uma pessoa quando se aloca duas pessoas para fazerem o trabalho que apenas uma poderia fazer sozinha? Essa é uma preocupação válida, porém a prática da programação em par demonstra que o resultado é muito diferente daquele que pode ser imaginado a princípio.

O tempo gasto na implementação de uma certa funcionalidade sempre é diferente quando se está programando em par. Se um programador programa sozinho leva, por exemplo, um dia para implementar uma funcionalidade, porém, se estivesse programando em par, levaria, aproximadamente, a metade do dia.

Estudos científicos demonstraram (LAURI WILLIAN et al, 2000) que praticamente não existem diferenças de produtividade em equipes que programam em par e outras que não programam, elas produzem basicamente a mesma coisa. A diferença está quando se fala em longo prazo. O código produzido pela equipe que programa em par é quase isento de defeitos devido ao processo contínuo de revisão a que o código é submetido. É certo que defeitos sempre podem existir, porém a prática da programação em par reduz a quantidade de erros, assim a equipe gastará menos tempo futuramente tentando descobrir a causa dos defeitos. Além disso, como as soluções geradas pelos pares tendem a ser mais simples e eficazes, é mais fácil dar manutenção no código.

- **A pressão do par**

A pressão do par é um elemento sutil, mas que influencia em muito a produtividade da equipe.

O ato de programar exige muita concentração, porém, no dia-a-dia de um programador, ele se depara com diversas fontes de distração: e-mail, mensagens instantâneas, sites, cansaço, que provoca uma diminuição do seu ritmo de trabalho no código.

Quando o programador está trabalhando em um problema que exige uma solução mais elaborada, porém ele está cansado, ou está desmotivado, ele acaba adotando uma solução insuficiente, porém mais rápida. Outras vezes ele trava diante de um problema e não consegue sair do lugar.

A programação em par corrige estes desvios através da pressão do par. “Quando o programador está acompanhado de outra pessoa, ele deixa de ter um compromisso consigo mesmo. O seu compromisso se expande e passa a englobar o seu colega” (TELES, 2004, p. 92).

A conseqüência disso é que diversos focos de distração são eliminados. Dificilmente um programador ficará lendo os seus e-mails com um colega ao seu lado. Quando ele se depara com um problema que exige uma solução mais elaborada ou “trave” diante de um problema e ele não se sentir disposto para implementar o que tem que ser feito, o seu par o ajudará na resolução do mesmo, seja incentivando ou assumindo o teclado.

- **Revezamento**

A condução da programação deve ser feita sempre por ambos os programadores que formam o par. Para isso é fundamental que haja um revezamento diversas vezes ao longo de um dia de trabalho.

O revezamento não se aplica apenas a quem conduzirá o teclado e quem ficará de navegador, mas também de quem serão os pares. Em um projeto deve-se trocar sempre de par, pois isso é importante para a disseminação do conhecimento.

- **A disseminação do conhecimento**

A programação em par também contribui para a disseminação do conhecimento. Entre os desenvolvedores de um mesmo projeto é possível encontrar diferenças em termos do conhecimento técnico, conhecimento do projeto e experiência própria. A programação em par faz com que o conhecimento geral da equipe seja mais homogêneo.

Quando um programador mais experiente faz par com um novato, que está conduzindo o teclado, o programador mais experiente poderá identificar melhorias no código

do seu colega. Essas melhorias contribuem para um aumento do conhecimento do novato. Outras vezes, um programador poderá corrigir aquilo que seu colega costuma fazer de errado.

Desta forma, toda a equipe é nivelada por alto, pois busca potencializar o que há de melhor em cada um e suprimir as falhas.

3.4.5 Refatoração

Nos modelos tradicionais de desenvolvimento, quando um desenvolvedor se depara com um código mal escrito ou pouco legível dificilmente ele efetua alterações neste código para deixá-lo mais simples, mesmo que tiver que implementar novas funcionalidades.

O XP recomenda que os desenvolvedores, ao encontrar um código pouco legível, mal codificado, duplicado, sem padronização, devem alterar esse código para que ele fique mais legível e simples, porém esta alteração não pode mudar o seu comportamento observável e sua estrutura. Esta prática recebe o nome de refatoração (MARTIN FOWLER, 1999).

Agindo desta forma, é possível assegurar que o código estará sempre simples e legível, o que facilita a manutenção futura, seja ela corretiva, evolutiva, preventiva ou adaptativa. Um código mal formulado cria enormes dificuldades para quem um dia precisar modificá-lo ou compreendê-lo, porém quando está limpo e legível, é possível alterá-lo rapidamente.

A refatoração anda de mãos dadas com a prática do código coletivo, uma vez que todo desenvolvedor tem a possibilidade de atuar em qualquer parte do sistema. Com o código simples e legível os desenvolvedores conseguem ler facilmente qualquer parte do código.

A prática da refatoração está apoiada pelos testes automatizados (prática abordada posteriormente), pois com ela é possível saber se o código continua produzindo as mesmas respostas mesmo depois de ter sido alterado.

3.4.6 Desenvolvimento guiado pelos testes

> **“**Testar é a parte do desenvolvimento de software que todo mundo sabe que precisa ser feita, mas ninguém quer fazer. Testar costuma ser uma atividade chata, que consome tempo e só é valorizada depois que o sistema entra em produção e diversos problemas começam a surgir” (TELES, 2004, p. 104).

A maioria dos projetos deixa os testes para serem feitos no final mas, como os projetos costumam atrasar, o período previsto para testes acaba sendo usado para terminar o desenvolvimento. Aqueles projetos que conseguem chegar ao ponto de fazer teste no final são prejudicados porque testar no final é menos eficiente do que fazer testes durante o projeto.

Testar não deve ser uma atividade complicada e chata, ela dever ser algo simples de ser feito e não deve consumir um tempo excessivo. Além disso, é importante que o ato de testar seja parte natural do ato de programar.

- **Testar é investir**

Um projeto de desenvolvimento de software pode ser separado em duas categorias básicas: desenvolvimento e depuração. Todo projeto gasta um certo tempo com depuração quando o software está com um ou mais *bugs* e é um processo extremamente custoso.

O que deixa a depuração custosa é o tempo existente entre o instante que o *bug* é inserido no sistema e o instante em que ele é detectado. Quanto maior for esse tempo mais o custo de depuração cresce. Isso acontece porque o desenvolvedor precisa entender o que tinha sido feito no código, o porquê foi feito daquela maneira, o que isso tentava resolver e o que pode estar gerando esse erro. Ou seja, o custo para reaprender sobre a funcionalidade é alto.

O desenvolvimento guiado pelos testes faz com que a correção de erros da funcionalidade seja mais barata por dois motivos: primeiro que o teste expõe o defeito assim que ele entra no sistema, ou seja, o teste aponta o erro imediatamente, o que evita uma perda

enorme de tempo fazendo depuração e segundo que, quando o *bug* aparece em uma parte do sistema que não é aquela na qual se está trabalhando, o teste informa exatamente onde está o defeito, permitindo uma correção rapidamente.

A idéia principal da programação guiada pelos testes é reduzir o tempo dedicado à depuração, reduzindo também o custo da depuração em um projeto.

- **Testando no XP**

O XP trabalha com dois tipos de testes: testes de unidade e testes de aceitação.

O teste de unidade é realizado sobre cada classe do sistema a fim de verificar se os resultados gerados estão ou não corretos. O teste de aceitação é feito sobre cada funcionalidade, ou estória do sistema, verificando assim a interação entre um conjunto de classes que a implementam.

Tanto os testes de unidade quanto os testes de aceitação devem ser gerados primeiros, ou seja, os testes de unidade devem ser escritos antes da classe e os testes de aceitação devem ser criados antes da implementação das estórias. Além disso, os dois testes devem ser automatizados para que possam ser executados inúmeras vezes no decorrer do desenvolvimento.

- **Testes de Unidade**

Os testes de unidade servem para verificar se os métodos das classes do sistema fornecem respostas corretas. Os testes são escritos pelos desenvolvedores enquanto codificam o sistema, ou seja, faz parte da atividade de programação desde a primeira classe codificada do sistema até a última.

Todos os testes de unidades devem ser automatizados e devem executar corretamente ao longo de todo o desenvolvimento. Caso um teste venha a falhar, a equipe passa a ter como

tarefa prioritária a correção do erro que está presente no código. O XP busca garantir que o sistema esteja operando sempre com o máximo de qualidade e segurança.

Os testes de unidade têm como principal objetivo gerar um código limpo, claro e que funcione corretamente. Os testes não só fazem a validação de todo o sistema como também, pelo fato de terem sido escritos antes das classes, auxilia a análise, o *design* e a codificação do sistema.

Para se testar uma classe, é necessário compreender muito bem as suas características. É preciso conhecer quais são os parâmetros aceitos em cada método e conhecer quais seriam as respostas corretas, as incorretas e as que não poderiam ocorrer em nenhuma circunstância. É importante conhecer bem o problema assim como o tipo de solução para ele.

Teles (2004) diz que:

> "Quando um desenvolvedor pensa no teste antes de pensar na implementação, ele é forçado a compreender melhor o problema e que ele se vê diante da necessidade de se aprofundar o entendimento, entrando nos detalhes e levantando hipóteses". Diz também que em momentos como estes, inúmeras dúvidas podem surgir e que é importante que o cliente esteja por perto para esclarecê-las, evitando assim que o desenvolvedor assuma premissas incorretas.

Quando se aprofunda no problema, o desenvolvedor está fazendo uma análise mais detalhada fazendo com que o desenvolvimento guiado pelos testes se torne uma técnica de análise. Após esta etapa, o desenvolvedor pode elaborar um ou mais testes para verificar a classe. Assim que essas etapas estejam concluídas, o desenvolvedor poderá implementar a classe orientando-se pelos testes.

O desenvolvedor deverá codificar somente o suficiente para que os testes executem perfeitamente. Ou seja, codificar somente o necessário para que os testes passem. Nem mais, nem menos. Sendo assim, os testes direcionam o desenvolvedor mostrando aquilo que deve

ser feito e também quando é a hora de parar. Portanto, os testes auxiliam o processo de codificação e ao mesmo tempo, a criação de um *design* simples.

Como os testes servem para validar o sistema a qualquer momento durante o projeto, todos eles deverão ser automatizados e agrupados para que assim possam ser executados inúmeras vezes ao longo do desenvolvimento.

- **Testes de Aceitação**

Utilizar os testes de unidade para verificar se cada classe do sistema está funcionando corretamente é fundamental, porém não é suficiente. É preciso verificar também se uma estória, isto é, uma funcionalidade, que freqüentemente é implementada por diversas classes, também gera resultados corretos.

Para verificar se as classes englobadas por uma estória estão se relacionando corretamente, o XP utiliza os testes de aceitação. Estes testes procuram simular uma interação do usuário com o sistema a fim de verificar se o comportamento do mesmo está ou não correto.

Os testes de aceitação são como um roteiro que tem vinculado a ele um conjunto de respostas esperadas. Para cada estória é escrito um roteiro que contém as ações que o usuário deverá executar e, para cada ação, existe uma resposta esperada. O usuário deve executar as ações e comparar os resultados com os resultados esperados que estão no roteiro. Se o sistema apresenta um resultado diferente do esperado é porque existe um erro nele ou o próprio teste está incorreto.

O XP recomenda que os testes de aceitação sejam criados pelo cliente, pois ele é quem melhor conhece a estória e seu funcionamento correto. Caso o cliente tenha dificuldade de escrever os testes, um membro da equipe poderá ajudá-lo, para que ele consiga transformar

suas idéias sobre a funcionalidade em testes que possam ser executados de preferência de forma automatizada.

"Executar o teste de aceitação de cada estória é essencial, mas também é fundamental executar todos os testes algumas vezes ao longo da iteração" (TELES, 2004). Para que se possa executar os testes de aceitação diversas vezes é necessário que eles sejam automatizados. Porém, automatizar os testes de aceitação não costuma ser uma tarefa fácil. Cabe aos desenvolvedores procurarem mecanismos e ferramentas que possam ajudá-los na automação dos testes.

Quando a automação dos testes de aceitação não pode ser concretizada, a equipe deve procurar executar o teste a cada funcionalidade assim que ela for concluída. E, ao final de cada iteração, deve-se executar todos os testes de aceitação das estórias que fazem parte daquela iteração. Se possível, executar também os testes das iterações anteriores que ainda não foram alteradas dentro do sistema.

3.4.7 Código Coletivo

Nos projetos tradicionais, o sistema costuma ser dividido em partes, de modo que cada desenvolvedor fique responsável por uma ou mais partes. Assim, cada membro é responsável por um subconjunto de funcionalidades que formam todo o sistema.

Quando se trabalha desta maneira, é comum que uma pessoa acabe se tornando ilha de conhecimento, ou seja, ela possui um domínio sobre uma certa área do projeto e somente ela conhece aquela parte ou aquele código.

As ilhas de conhecimento podem se tornar sérios problemas em um projeto. Elas podem se tornar um gargalo, caso a parte do código que está sob sua responsabilidade precise ser alterado várias vezes e ela não consiga atender a todas as demandas.

Outro problema acontece quando essa pessoa precise se ausentar durante um certo período de tempo ou deixe o projeto. A pessoa que ficará encarregada de substitui - lá poderá ter dificuldade para compreender o código quando este tiver que ser alterado.

Por estas razões que o XP trabalha com o código coletivo, onde não existe uma pessoa responsável por um subconjunto de código. "Cada desenvolvedor tem acesso a todas as partes do código e total liberdade para alterar o que necessitar, ou seja, o código é coletivo" (TELES, 2004, p. 145).

O código coletivo fornece uma maior agilidade no desenvolvimento, pois não existindo um responsável pelo código, qualquer pessoa pode alterá-lo, caso sinta a necessidade. Outra vantagem do código coletivo é que cria mais um mecanismo de revisão do código, uma vez que aquilo que é escrito por um par hoje, acaba sendo manipulado por outro amanhã, que, caso perceba algo confuso, poderá fazer uma refatoração.

3.4.8 Padrões de Codificação

Em projetos que adotam a prática do código coletivo é importante que a equipe consiga se comunicar de forma clara através do código. Ou seja, um código gerado por um par deve ser fácil de ser compreendido pelos outros pares. Para ajudar a se ter uma melhor compreensão de um código, o XP sugere a utilização de padrões de codificação.

Um exemplo de diferença de codificação é o programador que prefere colocar o "{" no final da declaração de um método, enquanto outro prefere colocar na linha abaixo da declaração, como mostrado no Quadro 2:

```
public int subtração ( int x, int y ){
     return x – y;
}
```

```
public int subtracao ( int x, int y )
{
       return x – y;
}
```

Quadro 2 – Diferença de codificação.

Para melhorar a compreensão de um código é importante que o padrão adotado pela equipe seja fácil, simples e leve em consideração aspectos como: identação, letras maiúsculas e minúsculas, comentários e nomes de variáveis e métodos. Desta forma, qualquer desenvolvedor poderá ler o código com velocidade, sem se preocupar em compreender estilos de formatação especiais.

3.4.9 *Design* Simples

Para que o cliente possa obter um *feedback* logo e gerar maior valor possível, o XP prega um *design* do sistema da forma mais simples possível para que possa atender as necessidades do cliente.

Nos projetos tradicionais, o custo de uma alteração no sistema cresce exponencialmente ao longo do projeto. Para tentar evitar alterações futuras, os desenvolvedores procuram criar generalizações dentro do código, o que acaba dando margens para especulações e implementações que, na maioria das vezes, não terá utilidade para o cliente.

O XP parte do princípio que o custo de uma alteração tende a crescer lentamente e se estabilizar ao longo do tempo de modo a se tornar praticamente linear. Esta premissa é dita

em função dos avanços nas linguagens e práticas de programação, novos ambientes e ferramentas de desenvolvimento, avanços na tecnologia de banco de dados, utilização de orientação a objetos, que, em conjunto com as outras práticas de XP, deixa os códigos simples, legíveis e com facilidade de alteração a qualquer momento.

3.4.10 Metáfora

Às vezes, uma pessoa está tentando explicar um assunto ou problema a outra pessoa e não há nada que a faça compreender, ou seja, simplesmente ela não consegue entender as mensagens que está se tentando passar. Depois de algumas tentativas, a pessoa faz uma comparação e analogia com o assunto que está em questão e a outra pessoa passará a entender deste assunto de uma forma muito mais rápida, chegando até a não esquecer mais. Esse artifício é chamado de metáfora.

A metáfora deve ser utilizada com intensidade durante um projeto, uma vez que facilita a comunicação e a fixação dos assuntos entre as partes. Ela tem um enorme efeito sobre o *design* do sistema, pois ajuda a equipe a criar um sistema coeso, consistente e adequado às necessidades do cliente.

Para que os desenvolvedores possam criar boas metáforas, utilizando-se de sua criatividade, é imprescindível ter uma boa condição física e um bem estar. Por isso, a prática de metáforas anda em conjunto com o ritmo sustentável.

3.4.11 Ritmo Sustentável

Um grande problema nos projetos de desenvolvimento de software é a falta de tempo para finalizar o projeto. Para compensar a falta de tempo, algumas equipes acabam submetendo os desenvolvedores a trabalharem até mais tarde e, às vezes, chegando a sacrificarem seus finais de semana e feriados.

Trabalhar além do horário pode até acelerar o desenvolvimento no primeiro momento, mas passado alguns dias, o rendimento da equipe cai drasticamente. O desenvolvedor cansado não consegue se concentrar bem no problema que está resolvendo, dando margens a erros no código, fruto da desatenção do desenvolvedor.

O XP recomenda que os desenvolvedores trabalhem apenas oito horas por dia e evitem fazer horas-extras. Adotando um ritmo sustentável de quarenta horas semanais e respeitando a individualidade e o físico de cada desenvolvedor, a equipe estará sempre concentrada e muito menos propensa a pequenos erros durante a implementação das funcionalidades.

3.4.12 Integração Contínua

No projeto de um sistema, principalmente nos métodos tradicionais, o sistema é dividido de modo que um módulo fique sob responsabilidade de um desenvolvedor. Ela fica responsável pela codificação e testes de tudo que diz respeito a sua parte. Esta estratégia é interessante porque reduz a complexidade e as preocupações de um desenvolvedor.

Para que os módulos se comuniquem os desenvolvedores padronizam interfaces, assim torna-se possível a integração dos módulos em um dado momento futuro. O problema é que

essa estratégia funciona mais na teoria do que na prática, pois costumam surgir erros na integração, principalmente se os períodos em que os módulos são integrados e testados são extremamente longos.

O XP propõe uma estratégia diferente. No XP, a equipe deve integrar os módulos continuamente, se possível deve ser efetuada diversas vezes ao dia. Adotando-se essa prática o *feedback* sobre a alteração efetuada será retornado em menor espaço de tempo. Com isso, caso erros apareçam eles serão corrigidos rapidamente para que a equipe possa realizar os testes necessários.

3.4.13 *Releases* Curtos

No modelo tradicional de desenvolvimento de software, o projeto é dividido em fases e o software só entra em produção quando essas fases estão finalizadas. Ou seja, o valor gerado pelo software só começa a ser recebido pelo cliente quando o sistema está finalizado. A maior parte do investimento do projeto é feita antes que o software esteja concluído, portanto antes que ele possa gerar algum tipo de valor econômico ao cliente.

O XP tem como objetivo gerar um fluxo contínuo de valor para o cliente, fazendo com que ele comece a receber os benefícios do sistema muito antes deste estar finalizado. A prática de *releases* curtos dá o máximo de valor econômico ao cliente em um curto espaço de tempo.

Como dito anteriormente, um *release* é um conjunto de funcionalidades implementadas que possui um valor bem definido para o cliente e que é colocado em produção para que todos os usuários possam obter seus benefícios. No seu ciclo de desenvolvimento, um projeto em XP pode ter um ou mais *releases*. Ou seja, o software pode

entrar em produção uma ou mais vezes ao longo do seu desenvolvimento. O ideal é que o projeto possua um grande número de *releases* e que eles sejam o mais curto possível.

3.5 Considerações Finais

Este capítulo apresentou o método de desenvolvimento ágil de software *Extreme Programming* (XP). Foram abordados todos os seus valores e suas práticas, que somados formam um conjunto de procedimentos que devem ser adotados pelas equipes que usam XP.

Analisando os dois capítulos anteriores é possível observar as principais diferenças entre os métodos tradicionais e o *Extreme Programming*. Basicamente a principal diferença entre XP e as metodologias tradicionais são o enfoque e os valores.

A idéia do *Extreme Programming* é o enfoque nas pessoas e não em processos ou algoritmos. Além disso, há uma preocupação maior em se gastar mais tempo com a implementação do que com a documentação, para fornecer uma versão do software funcionando o mais rápido possível. Outra característica é que o XP é adaptativo, ou seja, se adapta a novos fatores decorrentes do desenvolvimento do projeto (recebendo, avaliando e respondendo às mudanças), ao contrário das metodologias tradicionais que procuram analisar previamente tudo o que pode acontecer no decorrer do desenvolvimento sendo que essa análise prévia é difícil e apresenta um alto custo.

As metodologias tradicionais devem ser aplicadas apenas em situações em que os requisitos do software são estáveis e os requisitos futuros são previsíveis. Como os requisitos para o desenvolvimento de um software, na maioria das vezes, são mutáveis, essas situações são difíceis de serem alcançadas. Segundo Pressman (2000), uma alteração feita na fase de

implementação pode ter um custo de sessenta a cem vezes maior do que se a alteração tivesse sido feita na fase de análise de requisitos, quando usado o desenvolvimento em cascata.

No XP, em que se tem um *feedback* constante, é possível adaptar rapidamente o projeto às mudanças nos requisitos, o que a torna ideal para projetos com requisitos vagos e em constantes mudanças.

Outra diferença é que no XP são feitas constantes entregas de partes operacionais do software ao cliente. Com isso, ele pode usufruir dos benefícios do software mais rapidamente. O que não acontece nas metodologias tradicionais, em que o cliente precisa esperar muito para ver o software funcionando.

Um ponto negativo em XP é que não existe uma preocupação formal em fazer a análise e o planejamento dos riscos e também XP não é indicado para grandes equipes e empresas. Já nas metodologias tradicionais é possível fazer a análise e o planejamento dos riscos, assim como utilizá-las em grandes equipes.

4 ESTUDO DE CASO USANDO *EXTREME PROGRAMMING*

4.1 Considerações Iniciais

Este capítulo aborda o desenvolvimento de uma a aplicação baseando-se nos valores e práticas do *Extreme Programming* apresentado no capítulo anterior. Na Seção 4.2 é apresentada a linguagem de programação escolhida para o desenvolvimento da aplicação. Na Seção 4.3 apresentam-se as características da aplicação desenvolvida.

A aplicação foi desenvolvida com o objetivo de tentar verificar as principais vantagens e desvantagens do método ágil XP, porém, muitas práticas não foram utilizadas no desenvolvimento. *Extreme Programming* é um método voltado para equipes de desenvolvimento, mas este trabalho foi abordado individualmente, o que impossibilitou o uso de práticas coletivas como s*tand up meeting*, programação em par, código coletivo e integração contínua, assim como alguns itens de outras práticas como estimar em equipe as estórias durante o jogo do planejamento.

4.2 Linguagem de Programação Utilizada

A linguagem escolhida para o desenvolvimento do sistema utilizando XP foi a linguagem de programação Clarion. Clarion é uma Linguagem de 4ª Geração (L4G) que é tanto voltada a negócios quanto a propósitos gerais. É focalizada nos negócios no sentido de que contém estruturas de dados e comandos que são altamente otimizados para manutenção de arquivos de dados e necessidades do mundo dos negócios. É também utilizada a propósitos

gerais porque é completamente compilada (não interpretada) e possui um conjunto de comandos comparável funcionalmente com outra linguagem de 3ª geração (L3G), como C/C++, Modula-2, Pascal, etc.

O Clarion é a base da linha de produtos da *SoftVelocity* e apóia sua reputação para um desenvolvimento de aplicações de banco de dados de forma rápida e eficiente. Somado à linguagem de quarta geração (L4G) Clarion, o produto Clarion também inclui um compilador para C++ e para Modula2. Todas as linguagens compartilham um otimizador comum e podem ser misturados em uma aplicação simples.

Tanto desenvolvedores de *software* independentes quanto corporativos possuem necessidades similares: aumentar a produtividade para acompanhar a demanda por novas aplicações de banco de dados. O Clarion é um ambiente de desenvolvimento rápido com ênfase na geração de código e metadados, que são partes ou trechos de códigos que podem ser reutilizados para diversas funções durante o processo de geração de código de uma aplicação. Os metadados são usados para criar rapidamente aplicações com qualidade para gerenciar dados comerciais.

O ambiente do Clarion pode ser observado na Figura 6, em que o rótulo “A” ilustra a parte do ambiente que apóia a definição da aplicação, com a especificação dos procedimentos. No rótulo “B” é mostrado o controle de eventos, em que o programador digita os códigos necessários e no rótulo “C” é apresentada a área para a criação das janelas da aplicação.

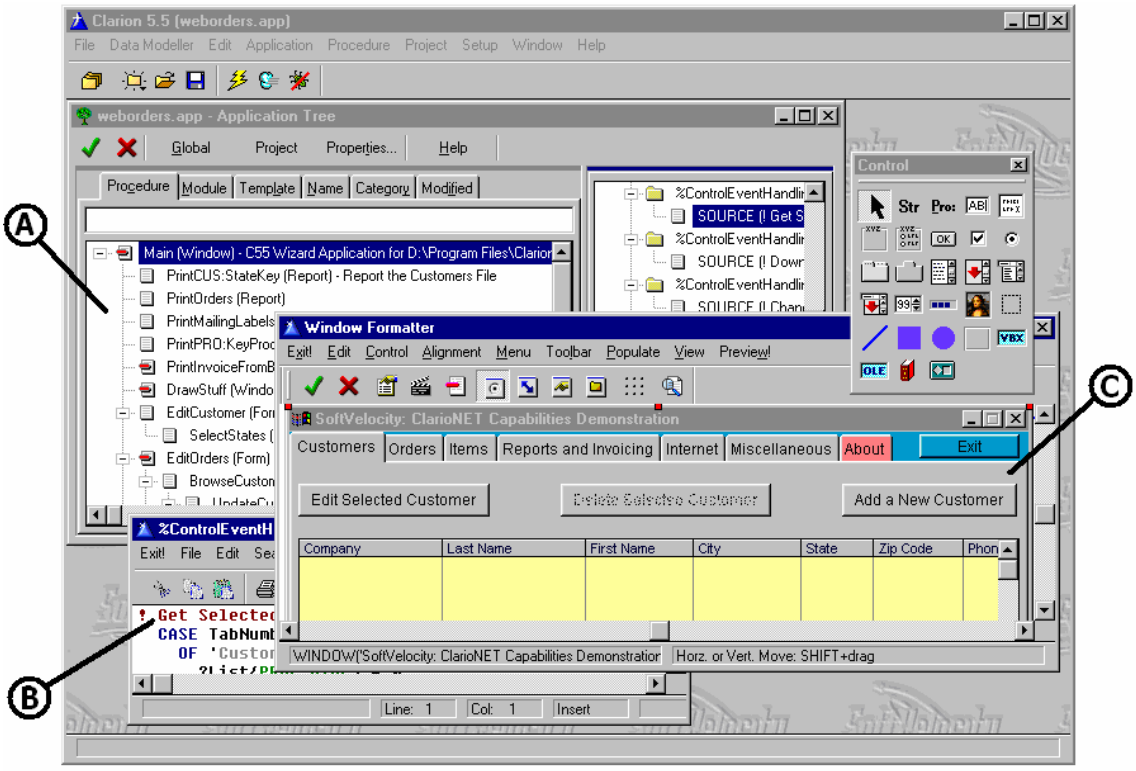

Figura 6 – Ambiente de desenvolvimento Clarion.

As metabases Clarion, que são bancos de dados utilizados para armazenar os metadados, permitem a novos desenvolvedores gerar códigos para projetos específicos usando *templates* criados por programadores mais avançados. Os *templates* são uma série de modelos prontos e pré-testados que definem o código-fonte a ser gerado para implementar as mais diversas funcionalidades que são requeridas por uma aplicação típica de acesso a bancos de dados. A metabase contém a maioria da informação que o Clarion precisa para criar uma aplicação totalmente funcional, que é aplicada diretamente aos requisitos específicos dos seus projetos. O benefício para o programador é que ele pode usar as ferramentas adicionais do Clarion para personalizar a aplicação gerada, a fim de ajustá-la às suas necessidades especiais.

As metabases do Clarion armazenam o metadado, aplicação e camadas de interface do usuário para todos os projetos, referenciando dados corporativos. Além disso, Clarion é um grande sistema de geração de código gera aplicações completas através das descrições armazenadas nos arquivos da metabase. Estas metabases Clarion são chamadas de Dicionário

de Dados (*Data Dictionary*) e Registro de Templates (*Template Registry*). Programadores podem desenvolver muitas aplicações a partir de um mesmo metadado.

4.3 Aplicação Desenvolvida: Sistema Acadêmico

A aplicação escolhida pelo cliente para desenvolvimento utilizando *Extreme Programming* foi um sistema acadêmico. O sistema tem como objetivo fazer o controle de alunos, professores, cursos, disciplinas, turmas e notas dos alunos referentes a cada disciplina.

As informações abaixo sobre o sistema foram descritas em estórias, porém, por motivos de visualização, elas foram inseridas no trabalho na forma textual.

Segundo o cliente, para o controle de alunos é necessário que as seguintes informações sejam armazenadas: RA (único), endereço residencial, número residencial, bairro, cidade, complemento, CEP, telefones, sexo do aluno, identificação se o aluno é canhoto ou destro, data de nascimento, data que o aluno ingressou na faculdade, observações e uma foto.

Todos os cursos da instituição devem ser cadastrados fazendo-se necessário armazenar o nome do curso, sua duração (em anos), uma sigla ou abreviação do nome e uma descrição sobre o curso.

As informações armazenadas para o cadastro das disciplinas são: nome, sigla ou abreviação do nome, carga horária, a qual série a disciplina está relacionada e um *status* indicando se a disciplina está ativa ou inativa naquele ano.

No cadastro de professores, as informações necessárias são: nome do professor, endereço residencial, número residencial, bairro, cidade, complemento, CEP, CPF, telefones, sexo do professor, data de nascimento, um *login* para acesso ao sistema e observações caso sejam necessárias.

No início de cada ano letivo, é necessário cadastrar as turmas de todos os cursos, informando qual é o curso daquela turma, a série ou termo, o ano letivo e o período da turma (escolher se é matutino ou noturno).

As disciplinas devem ser atribuídas às turmas e terá um professor associado a cada atribuição disciplina-turma. Cada aluno terá relacionado a ele uma grade curricular. A grade do aluno irá conter as notas bimestrais, faltas, exames, médias, dependências e adaptações sobre cada disciplina.

O sistema, segundo o cliente, deverá ter um controle de acesso sendo que somente o administrador do sistema terá acesso ao cadastro dos alunos, professores, cursos, disciplinas, turmas e notas. Os professores terão acesso somente às disciplinas ministradas por ele e às respectivas notas dos alunos. Os professores poderão também alterar o seu *login* e senha para acesso ao sistema. O acesso dos alunos fica restrito a consultar suas notas do ano em curso e alterar sua senha de acesso ao sistema.

O administrador e os professores poderão gerar relatórios de notas dos alunos. Os professores poderão gerar relatórios somente das disciplinas por eles ministradas. O administrador do sistema poderá gerar o boletim do aluno (notas e faltas em cada disciplina) e um relatório na forma gráfica sobre o desempenho do aluno nas disciplinas durante o ano.

Na Figura 7 mostra-se o Diagrama de Entidade-Relacionamento (DER), diagrama que permite especificar os dados e seus relacionamentos, da base de dados do sistema: o Aluno pode possuir muitos telefones e também possui muitas Grades. A Grade é formada pelo Aluno e por Atribuições de aulas, que por sua vez, são formadas por Professor, Disciplina e Turma. Um Curso possui muitas Turmas. As tabelas de Administrador e ConsultaNotaTemp são tabelas sem relacionamento com nenhuma outra tabela. A tabela Administrador é referente ao cadastro de administradores do sistema e ConsultaNotaTemp é uma tabela para armazenar dados temporários.

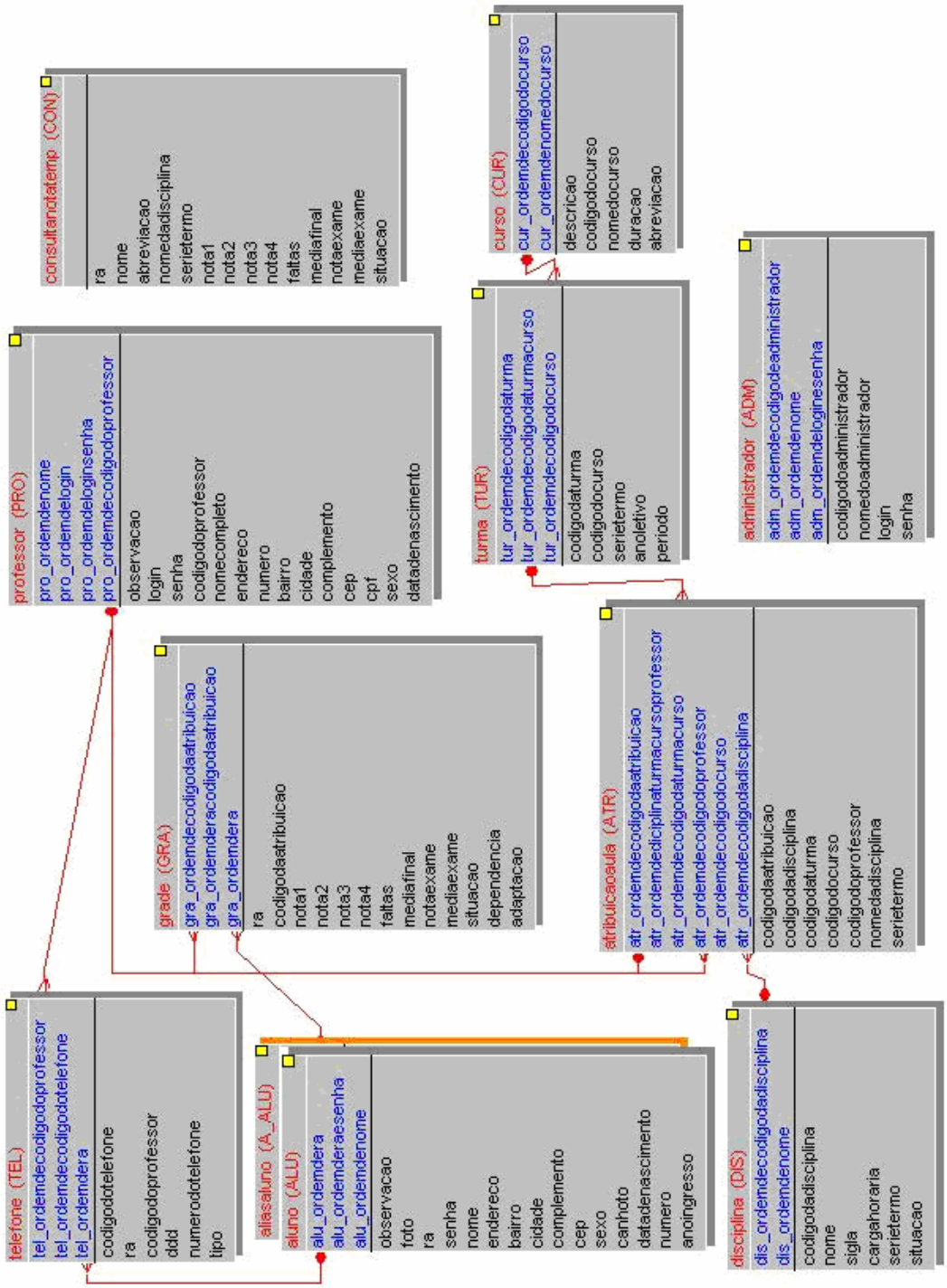

Figura 7 – Diagrama Entidade-Relacionamento do sistema acadêmico

Como em XP o software é entregue de forma incremental, o desenvolvimento do sistema foi dividido em quatro *releases*, descritos a seguir.

4.3.1 Primeiro *Release*

Durante o primeiro *release* foram implementados as interfaces e funcionalidades referentes ao cadastro de alunos e ao cadastro de cursos.

As funcionalidades desenvolvidas foram inclusão, alteração e exclusão de registros do sistema, validação de informações obrigatórias e restrições de campos que requerem valor único (RA do aluno, por exemplo, é um valor que não admite repetições).

Nas Figuras 8 e 9 são apresentadas a interface principal do sistema acadêmico e a interface do cadastro de aluno, no primeiro *release*.

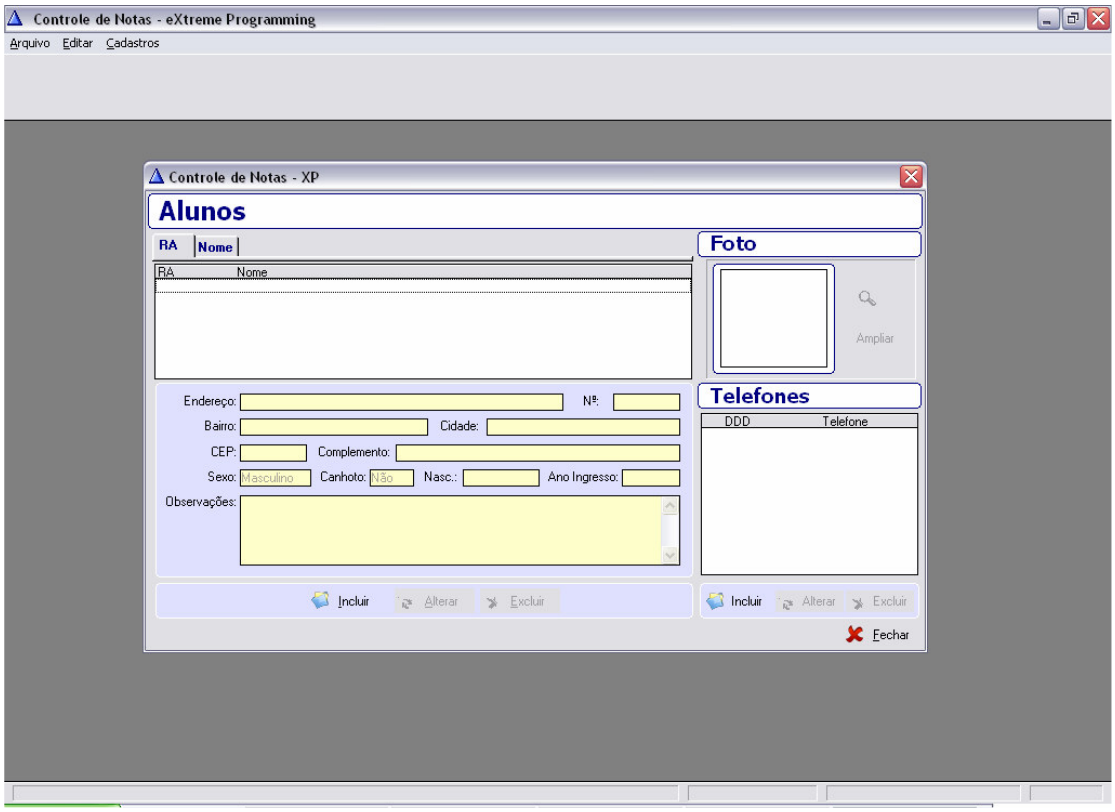

Figura 8 – Interface principal e o cadastro de alunos

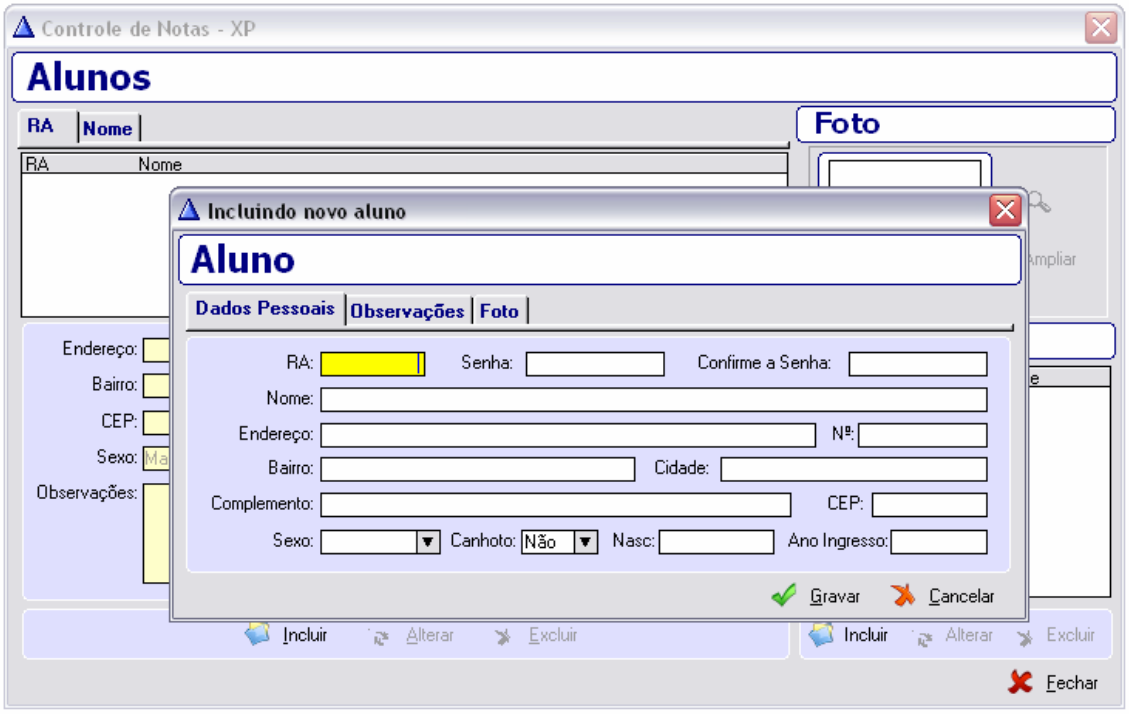

Figura 9 – Interface da operação de inclusão de um novo aluno.

Após o primeiro *release* entrar em produção, um *feedback* foi gerado pelo cliente solicitando que os campos na tela de visualização das informações do aluno fossem desabilitados.

Os campos estavam marcados como somente leitura, para evitar alteração indevida das informações, porém estavam habilitados para serem selecionados com um clique do mouse. Isso poderia causar confusão nos usuários do sistema que, ao clicar com o mouse no campo e verificar que o mesmo foi selecionado, poderiam achar que a alteração se dá por esse modo e quando percebessem que nada foi alterado concluiriam que o sistema está com um *bug*.

Outro *feedback* gerado pelo cliente foi a solicitação para que não permitisse a abertura da mesma tela diversas vezes, se caso ela já estivesse aberta. Ele percebeu que era possível invocar várias vezes a tela de cadastro de alunos ao clicar mais de uma vez no menu “Cadastro” e depois em “Alunos”.

Ambos os *feedbacks* foram corrigidos e antes da entrega do segundo *release*.

4.3.2 Segundo *Release*

O segundo *release* foi formado pelo controle das disciplinas, professores e turmas.

Assim como no *release* anterior, nesse *release* foram implementadas as interfaces para cada um dos cadastros e suas referentes funcionalidades como inclusão, alteração e exclusão de registros, validação de informações obrigatórias e informações únicas.

Na Figura 10 mostra-se a presença de um menu inicial na interface principal do sistema durante o segundo *release*.

Figura 10 – Menu presente na interface principal do segundo *release*.

Na Figura 11 é apresentada a interface do cadastro de professores.

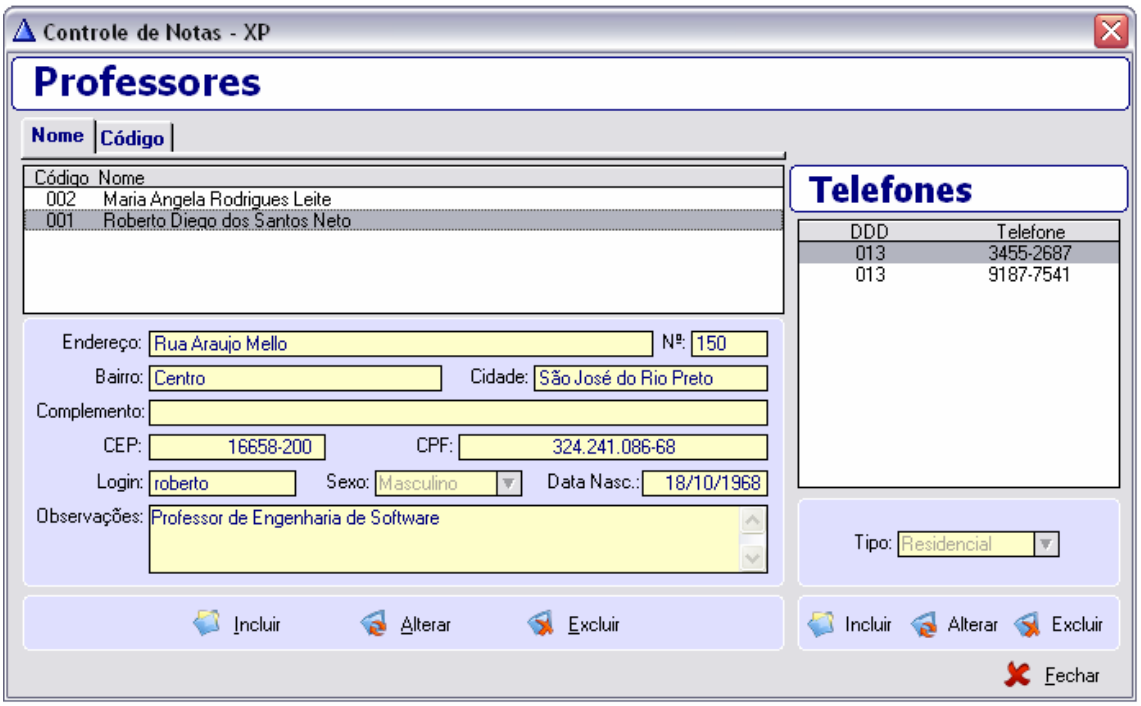

Figura 11 – Cadastro de professores.

Os *feedbacks* gerados após a entrega desse *release* foram com relação aos campos obrigatórios e a validação do CPF do professor.

O cliente solicitou que os campos obrigatórios tivessem algum tipo de marcação diferente para facilitar o cadastro. A solução encontrada para uma melhor visualização foi deixar os campos obrigatórios sublinhados. Assim os campos que tiverem com um sublinhado no seu nome são informações obrigatórias para uma inclusão do registro com sucesso. Os demais campos podem ou não ser preenchidos.

No cadastro de um novo professor, o sistema fazia a validação do CPF digitado, emitindo uma mensagem na tela avisando quando um CPF era inválido. Porém se o usuário quisesse, o professor era cadastrado com o CPF inválido mesmo. O cliente solicitou que isso fosse alterado, para que só se cadastrasse os professores que fossem digitados com um CPF válido.

4.3.3 Terceiro *Release*

Composto pelo cadastro de atribuição de aulas para os professores, cadastro da grade curricular do aluno, módulo de acesso do docente, módulo de acesso do aluno e *login* no sistema.

Na Figura 12 mostra-se a interface do *login*, que permite fazer o controle de acesso e permissões no sistema.

Figura 12 – Tela de acesso ao sistema (login e senha).

O menu varia de acordo com cada tipo de usuário que acesso o sistema, pois cada tipo de usuário possui funcionalidades diferentes. Na Figura 13 é apresentado o menu referente ao usuário do tipo administrador, com as seguintes funcionalidades: cadastro de alunos, cadastro de cursos, cadastro de disciplinas, cadastro de professores, cadastro de turmas, atribuição de disciplinas a professores, grade do aluno, relatório (implementado no quarto *release*) e sair. Na Figura 14, mostra-se o menu de funcionalidade par um usuário do tipo professor: disciplinas por ele ministrada, relatório (implementado no quarto *release*), alterar *login* e/ou senha e sair da aplicação. Na Figura 15 é apresentado o menu para um usuário do tipo aluno, com as funcionalidades de visualização de notas, alteração de senha e sair da aplicação.

Figura 13 – Menu referente ao usuário do tipo administrador.

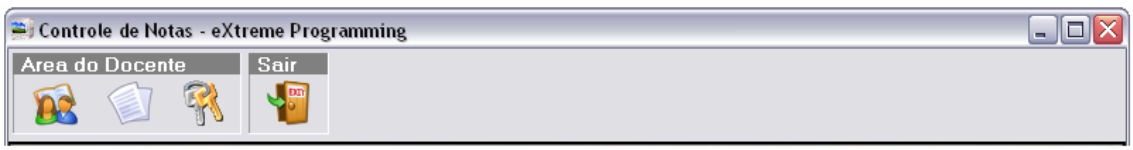

Figura 14 – Menu referente ao usuário do tipo professor.

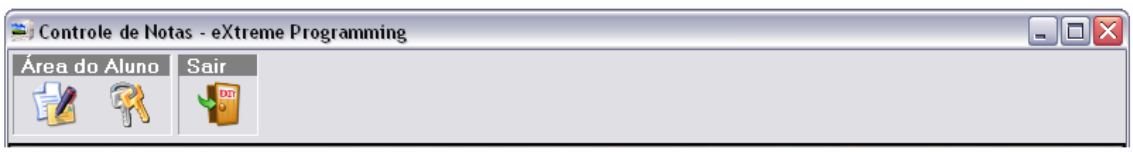

Figura 15 – Menu referente ao usuário do tipo aluno.

O cadastro de atribuição de aulas para os professores é formado pelas disciplinas, turmas e professores. É responsável por atribuir, para cada professor, uma ou mais disciplinas para ministrar e para quais turmas a disciplina será ministrada (turma do diurno, turma do noturno, ou ambas as turmas).

Na Figura 16 é mostrado a atribuição de aulas para um professor.

Figura 16 – Atribuição de disciplinas para professores

A grade curricular é formada pelos alunos e pela atribuição de aulas. Na grade são especificadas quais são as disciplinas cursadas pelo aluno naquele ano vigente, quais são as suas notas, médias e faltas para aquela disciplina, quem é o professor e em qual turma ele a cursa. Na Figura 17 apresenta o cadastro de uma grade para um aluno.

Figura 17 – Cadastro da grade curricular de um aluno

O módulo de acesso do docente somente pode ser visualizado pelo professor através de um *login* e uma senha. É nessa área restrita que o professor pode visualizar suas disciplinas e os alunos que as cursam, podendo também atribuir suas as notas e faltas. Na Figura 18 mostra-se a área restrita ao docente.

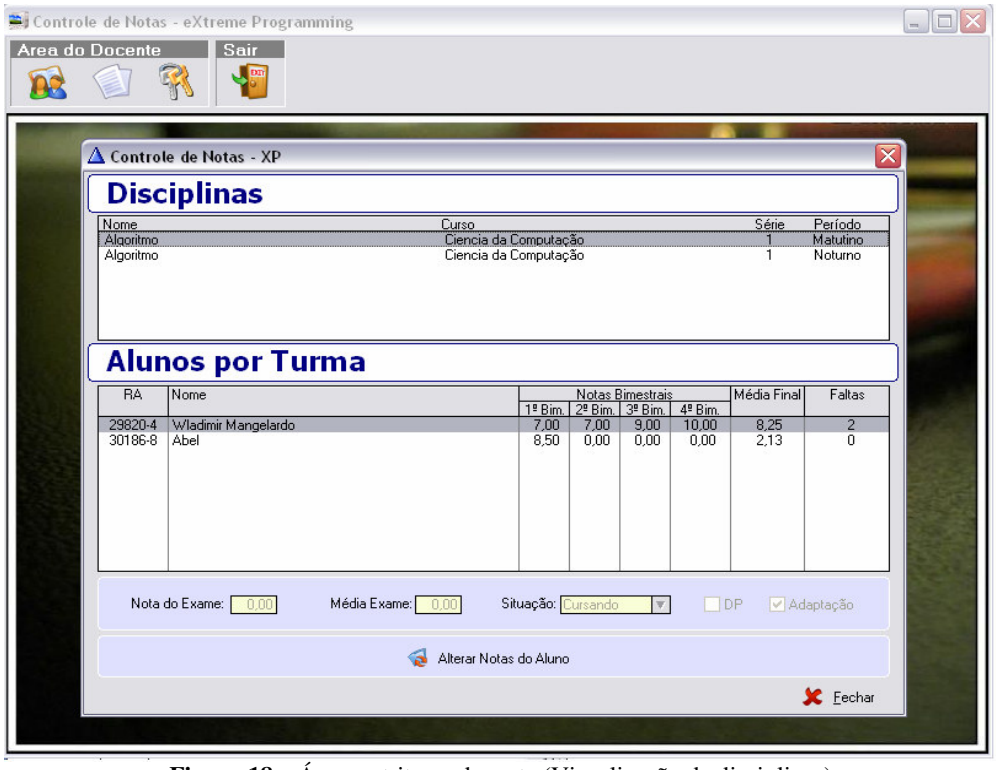

Figura 18 – Área restrita ao docente (Visualização de disciplinas)

A área restrita ao aluno (Figura 19) é o módulo acessível ao aluno onde é possível visualizar somente suas notas e somente as notas do ano vigente. O acesso a essa área se dá através do *login* do aluno pelo seu RA e sua senha.

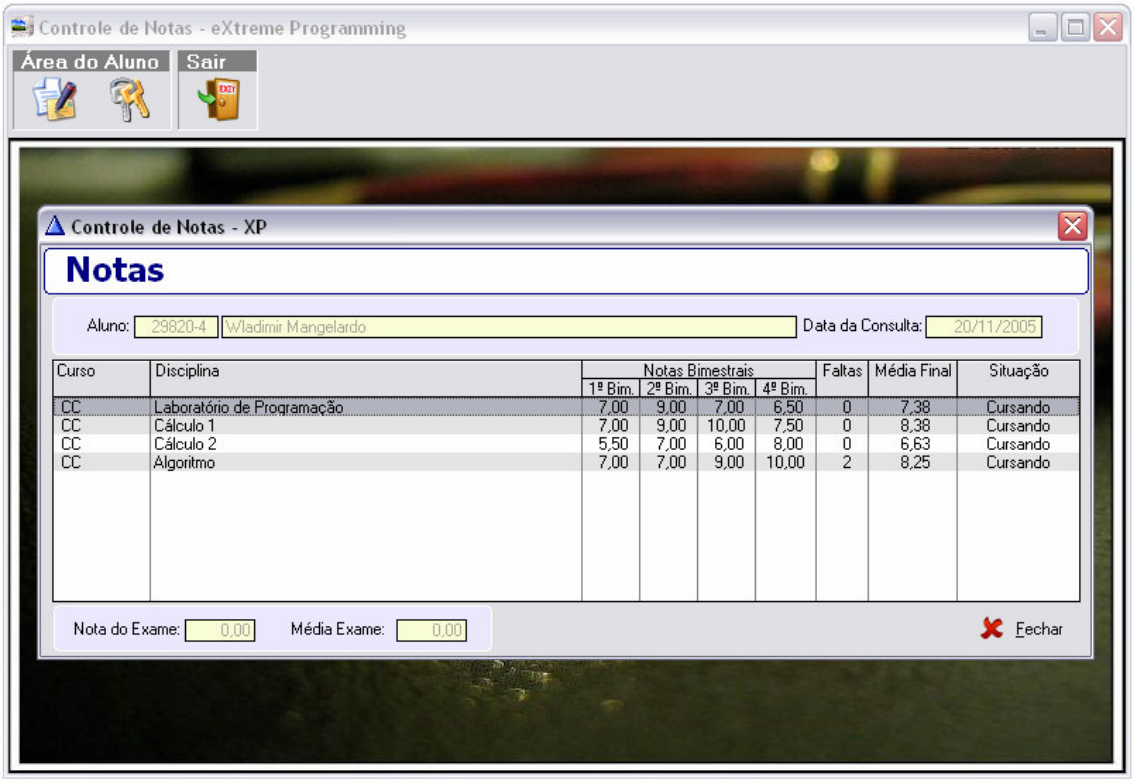

Figura 19 – Área restrita ao aluno para visualização de notas e faltas

4.3.4 Quarto *Release*

O quarto e último *release* é composto pelos relatórios em geral.

Nesse período de desenvolvimento foi implementada uma tela na área restrita ao docente para a geração de relatórios com as notas e faltas dos alunos que cursam suas disciplinas. O professor escolhe a disciplina que deseja e depois visualiza as informações na tela, podendo gerar ou não o relatório.

No módulo do administrador também foi implementada uma tela para relatórios. O administrador pode gerar um relatório contendo as notas e faltas em todas as disciplinas

cursadas de todos os alunos ou só de um determinado aluno. O administrador pode gerar também um gráfico, mostrando o desempenho do aluno em uma determinada disciplina.

Na Figura 20 mostra-se uma visualização prévia de um relatório gerado pelo administrador, contendo todas as notas e faltas de um determinado aluno. Na Figura 21, é apresentado um gráfico de desempenho disciplinar para um determinado aluno.

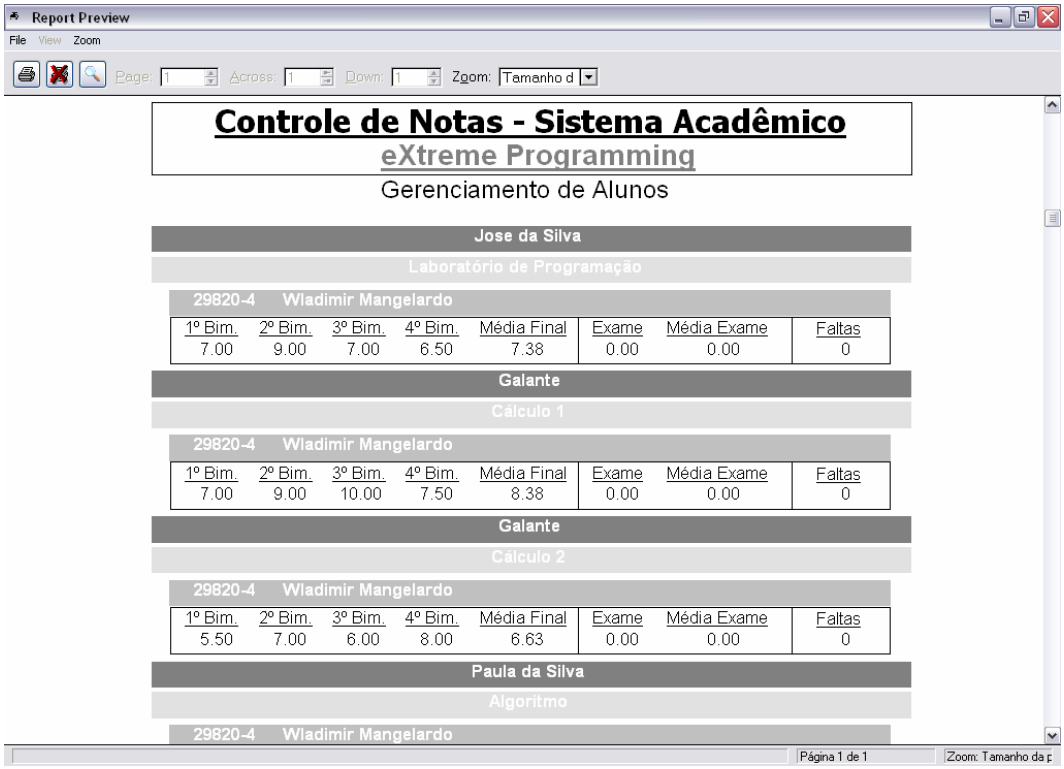

Figura 20 – Visualização prévia de um relatório de notas e faltas.

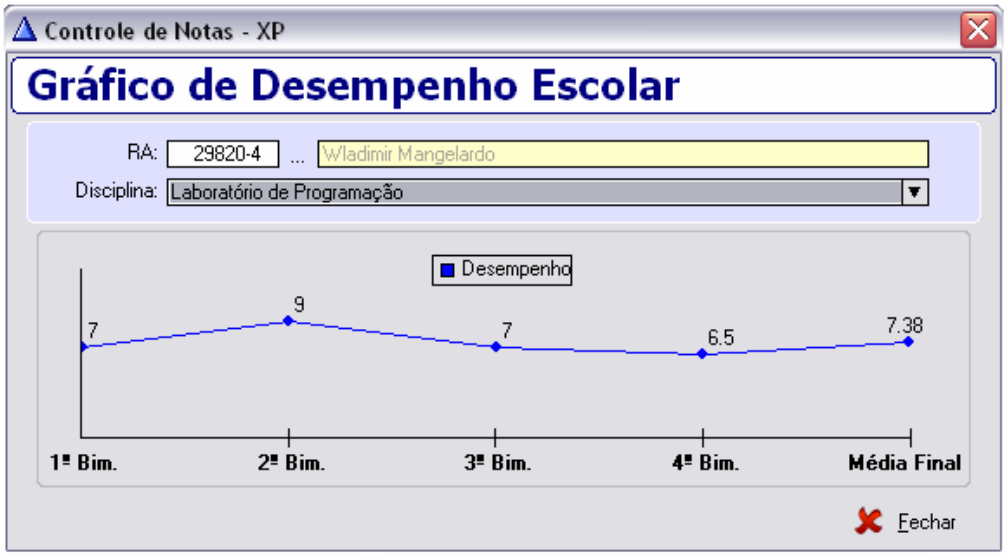

Figura 21 – Gráfico do desempenho escolar de um aluno.

4.4 Considerações Finais

Foi descrita neste capítulo a linguagem de programação utilizada no desenvolvimento da aplicação para o estudo de caso. Também foram apresentadas as funcionalidades que fizeram parte dos quatro *releases* em que o desenvolvimento da aplicação foi dividido.

Nos itens referentes a cada *release* foi apresentada a evolução da aplicação, tanto em sua interface quanto em suas funcionalidades, bem como os *feedbacks* recebidos de cada *release*.

É importante ressaltar que não foi possível abordar todas as práticas em que o XP se baseia, como s*tand up meeting*, programação em par, código coletivo e integração contínua, assim como alguns itens de outras práticas como estimar em equipe as estórias durante o jogo do planejamento. Essas práticas são práticas coletivas, que envolvem toda uma equipe de desenvolvimento, porém o trabalho foi desenvolvido individualmente.

5 CONCLUSÃO

A metodologia de desenvolvimento ágil de software *Extreme Programming* tem sido bem aceitas pela indústria de software e por pesquisadores da Engenharia de Software. De certa forma ela pode ser considerada nova, porém ela não apresenta muitos pontos revolucionários, pois ela agrupa uma série de práticas que tem sido utilizada por outras metodologias como programação em par e código coletivo.

Especialistas em tecnologia da informação vêm aprimorando as concepções do método *Extreme Programming* para atender as necessidades do mercado e principalmente das pessoas. Porém, um desafio futuro do XP é encontrar maneiras de eliminar alguns de seus pontos fracos, como a falta de análise de riscos, sem torná-la uma metodologia muito complexa.

Ainda faltam casos de sucesso no uso do XP em projetos grandes e críticos, porém os resultados iniciais nos demais projetos tem sido promissores, principalmente em termos de qualidade, confiança, cumprimentos nas datas de entrega e custo.

O desenvolvimento da aplicação como um estudo de caso não foi suficiente para se perceber diretamente as vantagens e desvantagens do uso do *Extreme Programming*. O principal motivo dessa insuficiência é a coletividade que o método exige. Como dito anteriormente, o XP é voltado para o desenvolvimento em equipe (preferencialmente pequena, com até 12 desenvolvedores), porém, o trabalho foi desenvolvido individualmente.

Contudo, é possível tirar algumas conclusões dos valores e práticas da metodologia *Extreme Programming*. De uma forma geral, durante todo o desenvolvimento foram utilizados alguns valores e práticas só que mais superficialmente. Com relação aos valores, foram utilizados o *feedback*, a comunicação, a simplicidade e a coragem. Em relação as

práticas, foram utilizadas o cliente presente, o jogo do planejamento, o desenvolvimento guiado pelos testes, o *design* simples e *releases* curtos.

Dos valores pode-se dizer que o *feedback* e a comunicação são de grande valia em um projeto de desenvolvimento de software. Os *feedbacks* recebidos do cliente foram essenciais para que o sistema ficasse de acordo com as suas necessidades. A comunicação também foi importante no desenvolvimento, principalmente para respostas a pequenas dúvidas que iam surgindo durante o projeto. As formas de comunicação utilizadas entre o cliente e o programador foram a mensagem instantânea e *e-mails*. A simplicidade é um valor muito importante em XP, porém não é tão fácil de ser utilizado. É possível visualizar novas necessidades que o cliente terá futuramente, inclusive necessidades que, se fossem implementadas no presente, seriam muito mais simples de serem feitas, porém, as mesmas não foram solicitadas pelo cliente.

Das práticas, o cliente presente é muito importante, principalmente para que o *feedback* e a comunicação ocorram e o trabalho especulativo não venha a acontecer. O jogo do planejamento não foi totalmente utilizado. Durante esse período foi decidido em quantos *releases* o projeto seria divido, quais funcionalidades iriam fazer parte de cada *release* e quais seriam os prazos de entrega de cada *release*. Uma dificuldade encontrada é que a idéia de desenvolvimento incremental não é muito animadora. Saber que a inclusão de outras funcionalidades no sistema provavelmente fará com que outras parem de funcionar faz com que o programador antecipe o desenvolvimento de uma funcionalidade em um *release* anterior para facilitar a sua inclusão da mesma no seu *release* correto. Ou seja, o programador faz um trabalho especulativo, tentando facilitar a inclusão da futura funcionalidade ele já inicia a codificação antes da hora, deixando de lado a prática do *design* simples. O desenvolvimento guiado pelos testes parece ser muito útil, porém quando não automatizado deixa a desejar, não cumpri o papel de ajudar na simplicidade do desenvolvimento. Acaba-se

desenvolvendo a funcionalidade da mesma forma como se os testes ainda não existissem. Ou seja, pensar no teste antes de desenvolver a funcionalidade só parece ser válido quando o mesmo for automatizado, senão vem a ser um tempo perdido.

É importante ressaltar que o desenvolvimento do trabalho foi feito individualmente, pode ser que se o mesmo projeto for desenvolvido em equipe, as dificuldade e facilidades encontradas nos valores e nas práticas utilizadas sejam totalmente diferentes dos citados acima. A mesma ressalva vale para um outro projeto. Pode ser que, mesmo desenvolvendo individualmente um outro projeto, os resultados dos valores e das práticas também sejam diferentes dos anteriores.

Como trabalhos futuros, pode-se pensar em um estudo de caso para cada uma das metodologias apresentadas. Assim é possível comparar o método tradicional e o XP, principalmente se o desenvolvimento da aplicação for realizado em equipe.

REFERÊNCIAS

TELES, V. M. **Extreme Programming: Aprenda como encantar seus usuários desenvolvendo software com agilidade e alta qualidade.** São Paulo: Novatec Editora Ltda, 2004.

PRESSMAN, R. S. **Engenharia de Software.** São Paulo: Makron Books, 2000.

BECK, K. **Extreme Programming Explained.** Addison Wesley, 1999.

FOWLER, M. et al. **Refactoring: Improving the Design of Existing Code.** Addison Wesley, 1999.

BECK, K.; FOWLER, M. **Planning Extreme Programming.** Addison Wesley, 2000.

LAURI WILLIAN et al. **Strengthning the Case for Pair Programming.** IEEE Software, Julho/Agosto de 2000, p.19-25, 2000.

Extreme Programming. Disponível em: <http://www.xispe.com.br>. Acesso em: 16 abr. 2005.

Clarion – Características Técnicas. Disponível em: <http://www.clarion.com.br>. Acesso em: 10 set. 2005.

Extreme Programming: A Gentle Introduction. Disponível em: <http://www.extremeprogramming.org/>. Acesso em: 1 mai. 2005.

ABRAHAMSSON, P.; SALO, O.; RONKAINEN, J. **Agile Software Development Methods. Review and Analysis.** Dissertação – University of Oulu, Finland, 2002.

SOARES, M. S. **Metodologias Ágeis Extreme Programming e Scrum para o Desenvolvimento de Software.** Artigo – Faculdade de Tecnologia e Ciências de Conselheiro Lafaiete. Unipac - Universidade Presidente Antônio Carlos, Conselheiro Lafaiete, MG, Brasil, 2004.

SOARES, M. S. **Comparação entre Metodologias Ágeis e Tradicionais para o Desenvolvimento de Software.** Artigo – Faculdade de Tecnologia e Ciências de Conselheiro Lafaiete. Unipac - Universidade Presidente Antônio Carlos, Conselheiro Lafaiete, MG, Brasil, 2004.

AGUIAR, A. et al. **Projecto SIMAT.** Disponível em: <http://simat.inescn.pt/doc/doc_tecnicos/rer101/contexto/v101/introducao.html>. Acesso em: 27 mar. 2005