

LIGA DE ENSINO DO RIO GRANDE DO NORTE
FACULDADE NATALENSE PARA O DESENVOLVIMENTO DO RIO
GRANDE DO NORTE

Emmanuel Richard de Moura

UMA PROPOSTA DE METODOLOGIA DE DESENVOLVIMENTO DE
SOFTWARE PARA UMA PEQUENA SOFTWARE HOUSE

NATAL
2007

EMMANUEL RICHARD DE MOURA

UMA PROPOSTA METODOLOGIA DE DESENVOLVIMENTO DE
SOFTWARE PARA UMA PEQUENA *SOFTWARE HOUSE*

Monografia apresentada ao Curso de Graduação em Sistemas de Informação da Faculdade Natalense para o Desenvolvimento do Rio Grande do Norte, como requisito obrigatório para obtenção do grau de Bacharel em Sistemas de Informação.

Orientador:

Gleydson de A. F. Lima

Natal

2007

Catálogo na Publicação – Biblioteca da FARN
Setor de Processos Técnicos

Moura, Emmanuel Richard de

Uma proposta de metodologia de desenvolvimento de software para uma pequena software house / Emmanuel Richard de Moura. – Natal: FARN, 2007.
66f.

Orientador: Gleydson de A. F. Lima (M. Sc.)

Monografia (Bacharelado em Sistemas de Informação) – Faculdade Natalense para o Desenvolvimento do Rio Grande do Norte.

1. Desenvolvimento de Softwares – Monografia. 2. Projetos de Software – Monografia. I. Título.

RN/FARN/BC

CDU 004.4

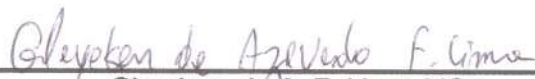


004.4
M885 p
2007
2x.04

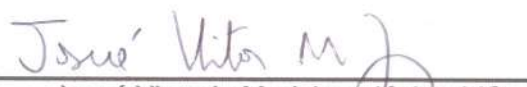
EMMANUEL RICHARD DE MOURA

UMA PROPOSTA METODOLOGIA DE DESENVOLVIMENTO DE
SOFTWARE PARA UMA PEQUENA *SOFTWARE HOUSE*

Monografia apresentada ao Curso de Sistemas de Informação da Faculdade
Natalense para o Desenvolvimento do Rio Grande do Norte, aprovada em 10 de
dezembro de 2007, pela banca examinadora composta pelos seguintes membros:



Gleydson de A. F. Lima, MSc.
Orientador



Josué Vitor de Medeiros Júnior, MSc.
Membro Examinador



Nilda Maria Teixeira de Serqueira, MSc.
Membro Examinador

RESUMO

O Objetivo desse trabalho é sugerir uma metodologia de desenvolvimento de software para uma determinada empresa de desenvolvimento de softwares; O mercado competitivo exige das pequenas empresas de desenvolvimento de software agilidade e qualidade; segundo o MCT, a indústria de software do Brasil rendia um total de 1,1 bilhão de dólares anuais em 1991 e passou a render 4,3 bilhões de dólares em 1999. Um crescimento total de 291%. Todavia, as pequenas empresas de desenvolvimento de softwares nacionais podem deixar de ser competitivas em um breve espaço de tempo se não derem a devida atenção para qualidade de software. Os aspectos analisados foram: Estrutura da empresa; Metodologias existentes e sua aplicabilidade à empresa em questão; e Adaptação das metodologias existentes. A metodologia aplicada foi estudo de caso analisando como a empresa está estruturada; Entrevistas com a equipe de desenvolvimento e pesquisa bibliográfica. Atingindo-se como resultado uma metodologia adequada às necessidades da empresa.

PALAVRAS-CHAVE: Qualidade, desenvolvimento, software,
metodologia.

ABSTRACT

The Objective of this work is to suggest a methodology of development of software for one determined company of development of softwares; The competitive market demands of the small companies of software development agility and quality; according to MCT, the industry of software of Brazil relieved a total of 1,1 billion of annual dollars in 1991 and started to relieve 4,3 billion dollar in 1999. A total growth of 291%. However, the small national companies of development of softwares can leave of being competitive in a brief space of time if not to give the due attention for quality of software. The analyzed aspects had been: Structure of the company; Existing methodologies and its applicability to the company in question; e Adaptation of the existing methodologies. The applied methodology was case study analyzing as the company is structuralized; Interviews with the development team and bibliographical research. Reaching itself as resulted an adequate methodology to the necessities of the company.

KEYWORDS: Quality, development, software, methodology.

LISTA DE ILUSTRAÇÕES

Ilustração 1 Porte das empresas, segundo a força de trabalho efetiva.....	13
Ilustração 2-Qualidade de software no Brasil.....	16
Ilustração 3Visão Geral do Iconix.....	21
Ilustração 4Atividades da tarefa de análise de requisitos.....	21
Ilustração 5Atividades da tarefa de análise e desenho preliminar	23
Ilustração 6Atividades da tarefa de desenho	24
Ilustração 7Atividades da tarefa de implementação.....	24
Ilustração 8Visão Geral do Método Fusion	27
Ilustração 9Verificação de intersecções entre modelos	29
Ilustração 10Modelos de Comunicação	33
Ilustração 11Uma user story em um cartão.....	37
Ilustração 12O custo de uma alteração não cresce muito ao longo do tempo	39
Ilustração 13Quadro de acompanhamento diário	48
Ilustração 14-Quadro de Tarefas.....	48
Ilustração 15Quadro de acompanhamento diário	61
Ilustração 16-Quadro de acompanhamento diário preenchido.....	62

SUMÁRIO

Introdução.....	9
Justificativa	10
Objetivo Geral.....	17
Objetivos Específicos	17
Referencial teórico.....	18
Metodologia RUP	18
Metodologia Iconix.....	20
Análise de Requisitos	21
Análise e Desenho Preliminar.....	22
Desenho	23
Implementação	24
Avisos do Processo ICONIX.....	25
Metodologia Fusion	26
Fase de Análise	27
Fase de Projeto	30
Fase de Implementação	30
Extreme Programming (Processo de Desenvolvimento Ágil)	31
Pontos do Manifesto de Desenvolvimento de Software Ágil:.....	31
Os Quatro valores do XP	32
Valores Fundamentais do XP:.....	32
Comunicação.....	32
Feedback.....	33
Simplicidade	34
Coragem.....	34
Princípios do Extreme Programming	35
Trabalhando com o Cliente.....	35
Usar metáforas para descrever conceitos difíceis	35
Planejamento.....	36
Manter as reuniões curtas	38
Teste.....	38
Design Simples.....	38
Programação em par	39
Codifique dentro dos padrões.....	39

Código Coletivo	40
Integração Contínua	40
Refactoring	41
Releases Curtos	41
Ritmo Sustentável.....	41
Metodologia	43
Resultados.....	44
Papéis dos membros da equipe	44
Planejamento.....	45
Releases.....	46
Iterações.....	46
Planejamento.....	47
Conclusão da iteração.....	48
Reunião diária.....	49
Codificação.....	49
Testes.....	50
Analogias.....	52
Ritmo Sustentável.....	53
Integração contínua.....	53
Conclusões.....	55
Sugestões para Trabalhos Futuros.....	55
Referências	56

INTRODUÇÃO

A empresa X foi criada em 1999, em Natal/RN, com o propósito de desenvolver softwares de gestão empresarial que se moldem às necessidades dos usuários, levando em consideração rapidez no desenvolvimento, facilidade de uso e adequação ao negócio do cliente.

A empresa tem crescido continuamente e, hoje, tem a maior parte de seus clientes fora do estado do Rio Grande do Norte.

Apesar de ter criado muitas ferramentas para agilizar o desenvolvimento dos softwares, a empresa carece de um método formal que oriente sua equipe.

O método de trabalho da empresa tem sido aprimorado de forma empírica. Isso tem causado alguns problemas como:

- Gastos que superam o orçamento;
- Consumo de tempo que supera o cronograma;
- Funcionalidades que não resolvem os problemas dos usuários. Necessitando gastar com retrabalho, refazendo o código para se adequar às necessidades do cliente.

A empresa tem uma equipe pequena composta de 3 programadores. Apesar disso, a produtividade é tão grande que a empresa tem uma gama de mais de 20 sistemas diferentes e poucas solicitações levam mais de um mês para serem implementadas.

As requisições dos clientes são relacionadas em uma lista e são priorizadas pelo desenvolvedor líder, que leva em consideração a complexidade da solicitação, dando prioridade às mais simples e às mais antigas.

Mensalmente é criado um release e atualizado no cliente que avalia se está de acordo com seu desejo. Caso não esteja, a equipe faz os ajustes e o processo se repete.

A empresa necessita de um método formal que oriente seus diversos setores relacionados com o desenvolvimento de software.

JUSTIFICATIVA

A literatura tem alertado os pesquisadores e os profissionais da área de sistemas de informação para uma série de fatores que têm efetivamente ameaçado o sucesso dos processos de desenvolvimento de software. São comuns os relatos de taxas relativamente altas de projetos atrasados, com custos que ultrapassaram as estimativas, cuja implementação não correspondeu ao que foi exigido ou que simplesmente foram abortados antes da sua conclusão (OSMUNDSON et al, 2002; ROPPONEN ; LAITINEN, 2000).

Segundo estudo feito por Barki et al (2001), mais de 50% dos projetos analisados ultrapassaram os valores previstos no orçamento, enquanto que 42% extrapolaram o tempo previsto para sua conclusão. De acordo com o Standish Group (1995), apenas 16,2% dos projetos de software eram completados dentro do cronograma e do prazo estipulado, sendo que a taxa de cancelamento antes da sua conclusão atingiu 31,1%. Boehm (1981) enfatiza que os efeitos de outputs negativos se refletem em várias dimensões, atingindo clientes, usuários, desenvolvedores e mantenedores dos projetos.

Desde os anos 1960s, quando a frase "the software crisis" foi pronunciada, muitos problemas desta área foram identificados, e muitos deles ainda persistem, tais como (GIBBS, 1994):

- Previsão pobre – desenvolvedores não prevêm adequadamente quanto tempo e esforço serão necessários para produzir um sistema de software que satisfaça as necessidades (requisitos) dos clientes/usuários. Sistemas de software são geralmente entregues muito tempo depois do que fora planejado;
- Programas de baixa qualidade – programas de software não executam o que o cliente deseja, consequência talvez da pressa para se entregar o produto. Os requisitos originais podem não ter sido completamente especificados, ou podem apresentar contradições, e isto pode ser descoberto muito tarde durante o processo de desenvolvimento;
- Alto custo para manutenção – a manutenção pode ser corretiva, quando ocorrem enganos (erros, falhas) no sistema já entregue, ou evolutiva, quando novas características são adicionadas ao sistema de software. Ambas são caras quando o sistema original foi construído sem uma arquitetura clara e visível.

- Duplicação de esforços – é difícil compartilhar soluções ou reusar código, em função das características de algumas linguagens adotadas, por falta de confiança no código feito por outra pessoa e até mesmo pela ausência/deficiência de documentação das rotinas e dos procedimentos já construídos.

Para solucionar alguns destes problemas, muitas pequenas *software houses* têm adotado metodologias de desenvolvimento de software. Todavia, os paradigmas metodológicos para desenvolvimento de software têm sido considerados somente em termos de “um método” de análise/projeto/implementação, isto é, um conjunto de conceitos, técnicas e notações. Esta visão elimina os aspectos tecnológicos, contextuais e organizacionais que potencialmente existem dentro de um processo de software.

Os ambientes tradicionais das pequenas empresas de desenvolvimento de softwares geralmente têm suportado somente a engenharia do produto, assumindo um processo implícito e tendo como foco principal o produto.

Esta visão tem limitado as pequenas empresas de desenvolvimento de softwares no que diz respeito à tomada de decisões, ao estabelecimento e arquivamento de metas organizacionais, à determinação de pontos para melhoria, à estipulação de prazos para entrega de produtos e à obtenção de uma certificação.

Já em 1989, Humphrey (1989) constatou que: “a ausência de práticas administrativas no desenvolvimento de software é a principal causa de sérios problemas enfrentados pelas organizações, tais como: atrasos em cronogramas, custo maior do que o esperado e presença de defeitos, ocasionando uma série de inconveniências para os usuários e enorme perda de tempo e de recursos”. Ainda hoje esta afirmação tem sido confirmada por diversos autores.

Na atual cultura das pequenas empresas de desenvolvimento de softwares, o planejamento, quando ocorre, é feito de forma superficial (WEBER, 1999) (SANCHES, 2001). A maioria dos projetos de software é realizada sem um planejamento de como a idéia modelada pelo levantamento de requisitos e necessidades dos clientes pode ser transformada em produto.

Os gerentes de projeto estão desacostumados a estimar (VIGDER, 1994). Quando estimam, costumam basear-se em estimativas passadas, mesmo sabendo que elas podem estar incorretas (não sabem também precisar o quanto elas estão incorretas).

Há gerentes que se recusam a estimar somente por julgarem perda de tempo, uma vez que correm o risco de obterem resultados incorretos e, portanto, estarem desperdiçando tempo.

Estimar, medir e controlar um projeto de software são tarefas difíceis, pois o desenvolvimento de software é uma atividade criativa e intelectual, com muitas variáveis envolvidas (como metodologias, modelos de ciclo de vida, técnicas, ferramentas, tecnologia, recursos e atividades diversas). Os gerentes inexperientes tentam cumprir prazos dando a máxima velocidade na fase inicial e estão despreparados para os momentos de impasse, quando os ajustes são inevitáveis.

A dinamicidade do processo de software dificulta também o gerenciamento efetivo de projetos de software, devido às alterações constantes nos planos de projetos, redistribuição de atividades, inclusão/exclusão de atividades, adaptação de cronogramas, realocação de recursos, novos acordos com os clientes, entregas intermediárias não previstas, etc. Um projeto de software também deve adaptar-se ao ambiente, dependendo da disponibilidade de recursos, ferramentas e habilidades do pessoal ou equipe.

Segundo Laitinen (2000), as pequenas empresas de desenvolvimento de softwares são as que mais urgentemente necessitam de formalização de procedimentos para gerenciamento e controle de seus projetos. Também afirma que, a utilização de padrões e normas está sendo pouco considerada por elas (Lindvall, 2000).

Devido ao fato destes modelos terem sido originalmente desenvolvidos para o âmbito de empresas bem estruturadas e departamentalizadas, ou seja, tipicamente para o âmbito de grandes empresas, dificultando ainda mais a orientação para empresas de pequeno e médio porte (Belloquim, 1999).

Outros fatores ainda agravam os problemas de gerenciamento de projetos de software em uma pequena empresa de desenvolvimento, tais como: inexistência de um processo definido, recursos pessoais e financeiros limitados, falta e/ou pouca cultura em processos, pouco treinamento em engenharia de software, imaturidade metodológica, crescimento ocorrido por demanda, falta de experiência administrativa por parte dos gerentes e diretores e falta de definição das metas organizacionais.

A classificação do porte de uma pequena empresa de desenvolvimento de software pode ser realizada considerando diversos fatores, entre eles podemos citar: faturamento, força de trabalho e comercialização anual bruta. Nesta pesquisa é

adotada a classificação do porte da empresa baseada na força de trabalho, dada pelo Ministério da Ciência e Tecnologia (BRASIL, 2001).

A grande maioria das pequenas empresas de desenvolvimento de softwares atualmente são micros ou pequenas empresas. A figura abaixo apresenta um gráfico, retirado do (BRASIL, 2001), que mostra a predominância de micros e pequenas empresas com percentuais superiores a 70% desde 1995.

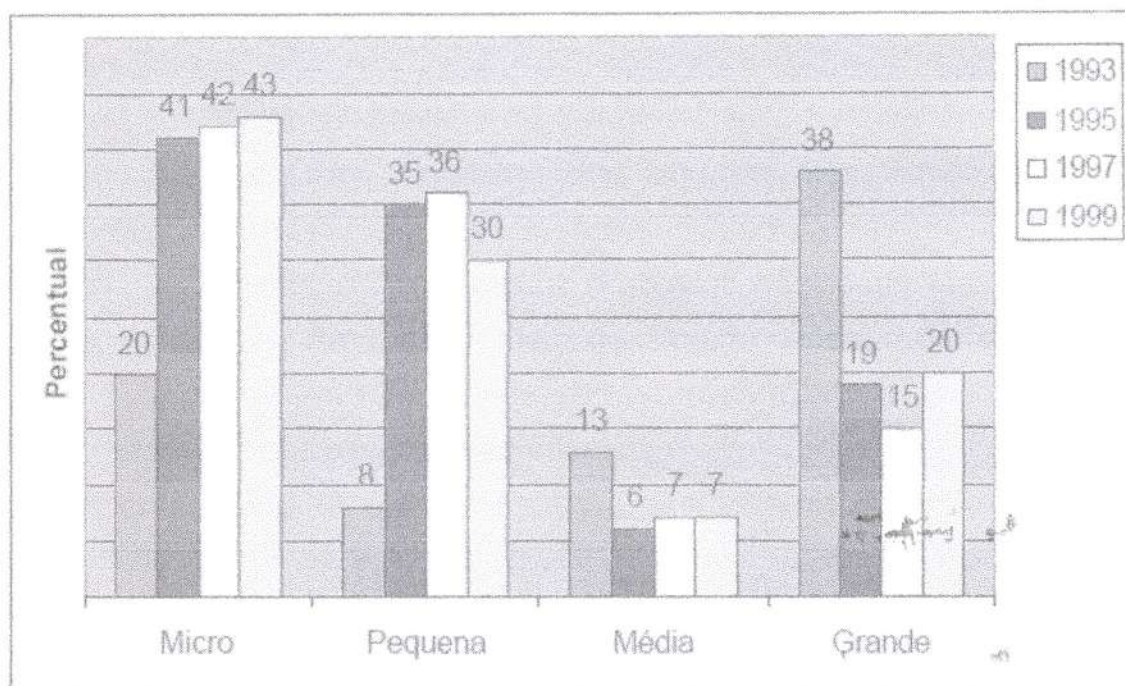


Ilustração 1 Porte das empresas, segundo a força de trabalho efetiva

O MCT ainda preconiza que se for considerado o serviço terceirizado, que é muito comum na área de produção de serviços e produtos de software, o percentual de empresas micro diminuem significativamente, aumentando o rol das pequenas empresas de desenvolvimento de softwares. Estudos realizados também em outros países, como o (STANDISH, 1995), também estabeleceram a predominância de empresas de pequeno porte no setor computacional.

Atualmente, uma grande parte da população mundial depende de aplicações de software para realizar suas atividades diárias. Segundo Reed (2000), se alguns sistemas de uso global deixarem de funcionar, aproximadamente 40% da população mundial sofrerá as consequências do problema.

Fazer software não é uma tarefa fácil. Fazer software de qualidade é mais difícil. Os resultados obtidos ao longo do tempo nos mostram padrões de baixa qualidade e de custos e prazos completamente ultrapassados.

Desde 1968 fala-se em metodologias de desenvolvimento de software, mas resta muito para ser feito na busca de qualidade e produtividade no desenvolvimento de software.

Ainda hoje, qualidade do software é tema muito discutido. Temos mercados cada vez mais competitivos e clientes cada vez mais exigentes, fazendo com que as empresas busquem desenvolver produtos cada vez melhores. Fuggetta (2000) afirma que a qualidade do software está fortemente relacionada à qualidade do processo de software.

As empresas buscam constantemente o melhoramento de seus processos, exigindo profissionais qualificados para desenvolver softwares de qualidade. Assim, vale a pena analisar os diversos esforços que foram efetuados pelas empresas ao longo do tempo, e perceber por que alguns processos não foram efetivos na resolução dos problemas. As organizações que forem capazes de integrar, harmonizar e acelerar os seus processos de desenvolvimento e também de manutenção de software terão a primazia do mercado.

Segundo Lindvall (2000), a preocupação com o processo de software está relacionada à necessidade de entender, avaliar, controlar, aprender, comunicar, melhorar, prever e certificar o trabalho dos engenheiros de software. Para isso é preciso documentar, definir, medir, analisar, avaliar, comparar e alterar os processos.

Para Fuggetta (2000), a pesquisa em processo de software trata dos métodos e tecnologias utilizados para avaliar, apoiar e melhorar as atividades de desenvolvimento e manutenção de software. A primeira contribuição importante da pesquisa na área de processo de software é o convencimento de que desenvolver software é fruto de um esforço coletivo, complexo e criativo e de que a qualidade do software depende das pessoas, da organização e dos procedimentos usados em seu desenvolvimento. Os problemas do desenvolvimento e da manutenção de software não podem ser solucionados com a introdução de ferramentas e ambientes de desenvolvimento de software, embora estes sejam aspectos importantes. Também não podem ser solucionados com a seleção de um modelo de ciclo de vida, mesmo que este seja adequado ao desenvolvimento. A definição e o uso de processo de software envolve a complexa inter-relação de fatores organizacionais, culturais, tecnológicos e econômicos.

Questões como porte da empresa e a cultura organizacional, objetivos de projetos específicos, recursos disponíveis, tecnologias de desenvolvimento, conhecimento e experiência da equipe impõem características aos processos.

Assim, as situações é que definem como devem ser os processos de software.

Segundo Coleman (1996), os problemas relacionados ao desenvolvimento de software são caracterizados, freqüentemente, pela baixa capacidade de previsão, pela baixa qualidade dos softwares, pelos altos custos de manutenção e pela duplicação de esforços.

Os responsáveis pelo desenvolvimento têm sérios problemas na previsão do tempo e do esforço necessários para produzir um sistema que satisfaça os requisitos do usuário. Segundo Brooks (1982), medidas quantitativas de tempo e esforço não são facilmente convertidas entre si. Os erros são encontrados apenas nas etapas finais do processo de codificação, fruto de uma análise falha combinada com a pressa para a entrega do software. Tanto a manutenção corretiva quanto a evolutiva são dispendiosas quando o sistema não foi construído com uma arquitetura clara e visível. Os projetos são freqüentemente iniciados "da estaca zero", compartilhando pouco código de projetos anteriores. Mesmo no âmbito de um mesmo projeto esta situação pode ser encontrada.

Portanto, nesse sentido, é importante estabelecer uma metodologia simplificada e flexível, com atividades detalhadas e que seja de fácil implantação, buscando melhorar o processo de desenvolvimento das empresas e a qualidade do software produzido.

Segundo o MCT, a indústria de software do Brasil rendia um total de 1,1 bilhão de dólares anuais em 1991 e passou a render 4,3 bilhões de dólares em 1999. Um crescimento total de 291%. Todavia, as pequenas empresas de desenvolvimento de softwares nacionais podem deixar de ser competitivas em um breve espaço de tempo se não derem a devida atenção para qualidade de software. As informações disponibilizadas pelo MCT, conforme Ilustração 2, mostram esta preocupação das pequenas empresas de desenvolvimento de softwares no Brasil.

Indicadores seleccionados de qualidade de software	1995	1997	1999
Percentual de companhias de software com certificação ISO 9000	2%	8%	17%
Percentual de companhias de software usando CMM	3%	5%	10%
Percentual de companhias de software usando ISO/IEC 9126 para avaliação do produto	-	7%	10%
Percentual da renda anual investida em treinamento para melhoria da qualidade de software	-	2,5%	2,3%
Percentual das companhias de software aplicando sistematicamente surveys de satisfação do cliente	19%	25%	29%

Ilustração 2-Qualidade de software no Brasil

OBJETIVO GERAL

Propor uma metodologia de desenvolvimento de software para uma determinada software house.

OBJETIVOS ESPECÍFICOS

- Comparar metodologias de desenvolvimento;
- Descrever a realidade e problemas enfrentados pela empresa;
- Definir a metodologia de desenvolvimento mais adequada à empresa;
- Definir modelos e procedimentos a serem adotados pela empresa.

REFERENCIAL TEÓRICO

A quantidade de metodologias propostas para o desenvolvimento de software atingiu um número demasiado elevado, o que torna virtualmente impossível a sua apresentação. Por isso, enumerou-se algumas metodologias conhecidas, que maior relevância e divulgação tiveram. Para além destas, existiram outras contribuições importantes que não estão incluídas aqui por não apresentarem uma perspectiva integrada de todo o processo de desenvolvimento, mas apenas sugerirem anotações ou técnicas de modelagem.

METODOLOGIA RUP

Durante a década de 90, tornou-se óbvia para muitos teóricos da engenharia de software a vantagem do paradigma da orientação por objetos e, nesse sentido, começaram a proliferar as metodologias que tinham por base os respectivos conceitos. O exagerado número de metodologias que, entretanto foram propostas fez recordar a experiência negativa que já tinha ocorrido com as metodologias estruturadas, pelo que ganhou força a idéia da criação de uma metodologia de desenvolvimento de software "unificada", que tirasse partido da experiência e dos conceitos da linguagem UML.

O RUP é uma metodologia de engenharia de software desenvolvida e comercializada pela empresa Rational Software. Tendo em conta que a construção de software de qualidade de uma forma repetitiva e previsível é difícil, e que as causas dos problemas associados a este tipo de desenvolvimento têm sido uma constante ao longo do tempo, o RUP propõe várias boas práticas para o desenvolvimento de software aplicadas de forma integrada.

O RUP é mais do que uma "simples" metodologia de desenvolvimento de software, uma vez que pode funcionar como um conjunto de princípios genéricos utilizados para instanciar e configurar várias metodologias concretas, conforme o tipo de organização, o domínio de aplicação, o nível de competências, etc.

O RUP suporta diversas boas práticas do desenvolvimento de software: (1) é uma metodologia de desenvolvimento de software iterativa; (2) propõe a gestão integrada de requisitos desde a sua identificação até a implementação; (3) propõe o desenvolvimento de software baseado em arquiteturas de software e em componentes; (4) defende a modelação visual; e (5) o controle de qualidade permanente. Além destas características, o RUP integra outras idéias fundamentais, nomeadamente o fato de ser orientado por casos de utilização.

Existem diferentes perspectivas segundo as quais o processo de desenvolvimento de software e o produto dele resultante podem ser encarados, sobretudo tendo em conta os interesses particulares dos diversos intervenientes no processo, que resultam das suas funções na organização. A definição de uma arquitetura sólida é a chave de sucesso para qualquer projeto que utilize uma abordagem orientada por objetos, segundo Booch (1994). O RUP apresenta um modelo segundo o qual considera a existência de 4+1 visões do sistema, definidas ao longo do processo, e cuja representação é a seguinte: Visão Lógica, Visão Processo, Visão Implementação, Visão Distribuição e por fim a Visão Casos de Utilização. Cada visão é concretizada através de diversos diagramas do UML.

Para os usuários o mais importante é a satisfação dos seus requisitos funcionais, expressos numa visão lógica (Logical View) do sistema. A este nível é necessário garantir que se tem informação sobre o que o sistema deve fazer. A arquitetura lógica do sistema é representada através dos diagramas de classes que modelam as principais abstrações do sistema (classes, relações e pacotes). Para os técnicos que participam no desenvolvimento do sistema, interessa ter a visão da organização do mesmo em termos dos módulos de software (desde o código fonte até aos executáveis), bem como resolver questões relacionadas com a gestão da configuração, a facilidade do desenvolvimento, a reutilização e restrições relacionadas com as linguagens de programação. Esta é a visão de implementação (Implementation View), que reflete a estrutura estática do sistema. Os principais elementos de modelação nesta visão incluem os pacotes e os diagramas de componentes.

A visão de processamento (Process View) do sistema preocupa-se em representar os conceitos relacionados com a implementação do sistema no seu ambiente de produção, como tarefas, atividades e processos, bem como com as suas interações. Nesta visão, são consideradas questões como paralelismo e concorrência na execução, tolerância a falhas, distribuição de objetos, tempos de resposta e escalabilidade, desempenho, confiabilidade e integridade do sistema. Esta visão é modelada por diagramas de componentes.

A visão de instalação (Deployment View) representa a correspondência entre os componentes desenvolvidos pelo projeto e o respectivo suporte tecnológico, modelando a configuração dos elementos de processamento em tempo de execução. As principais preocupações são questões como a migração e instalação

do sistema e o respectivo desempenho, disponibilidade, confiabilidade e escalabilidade. São utilizados diagramas de componentes e de instalação, que permitem visualizar a localização física de componentes na infra-estrutura física e/ou computacional da organização.

Finalmente, a visão dos casos de utilização (Use Case View) tem em consideração os casos de utilização importantes, de forma a definir, por um lado, a arquitetura nas fases iniciais do processo e, por outro, a validar mais tarde a percepção das restantes visões. Funciona assim como a visão integradora de todas as restantes, e reforça a idéia do RUP enquanto metodologia conduzida por casos de utilização, que são naturalmente os principais diagramas de modelação desta visão.

METODOLOGIA ICONIX

O ICONIX é uma metodologia de desenvolvimento de software promovido pela empresa ICONIX Software Engineering, cujo foco de negócio reside na formação e produção de material para educação em tecnologias de objetos, em particular em CORBA, COM, Java e UML. O principal evangelista do ICONIX é Doug Rosenberg (1999), que define o ICONIX como um "processo" de desenvolvimento de software prático, algures entre a complexidade e abrangência do RUP (Rational Unified Process) e a simplicidade e o pragmatismo do XP (Extreme Programming).

O ICONIX é conduzido por casos de utilização, iterativo e incremental, tal como o RUP, mas sem a complexidade que este preconiza. Por outro lado, é relativamente pequeno e simples, tal como o XP, mas sem eliminar as tarefas de análise e de desenho que aquele não contempla. Por fim, o ICONIX usa o UML como linguagem de modelação e apresenta um alto grau de rastreabilidade (traceability).

O ICONIX é uma metodologia de desenvolvimento de software, englobando as seguintes tarefas principais: (1) análise de requisitos; (2) análise e desenho preliminar; (3) desenho; e (4) implementação, conforme se introduz sumariamente em seguida.

A metodologia consiste na produção de um conjunto de artefatos que retratam as visões dinâmica e estática de um sistema, e que vão sendo desenvolvidos incrementalmente e em paralelo, os modelos da estática e os modelos da dinâmica.

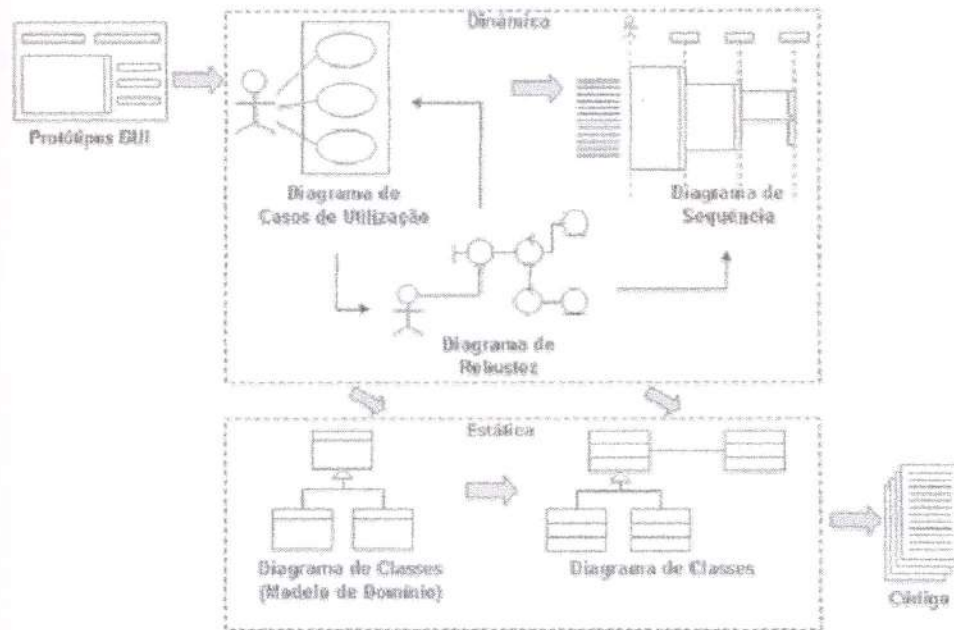


Ilustração 3 Visão Geral do Iconix

A Figura acima ilustra a visão geral do ICONIX. Esta figura revela um aspecto importante da utilização do UML: o fato da implementação de um sistema apenas depender da versão detalhada do diagrama de classes final. (Parece óbvio, mas muitos indivíduos pensam ainda que se poderiam usar, por exemplo, diagramas de sequência para gerar código automaticamente!).

Análise de Requisitos

A tarefa de análise de requisitos consiste na realização das seguintes atividades mostradas na figura abaixo:

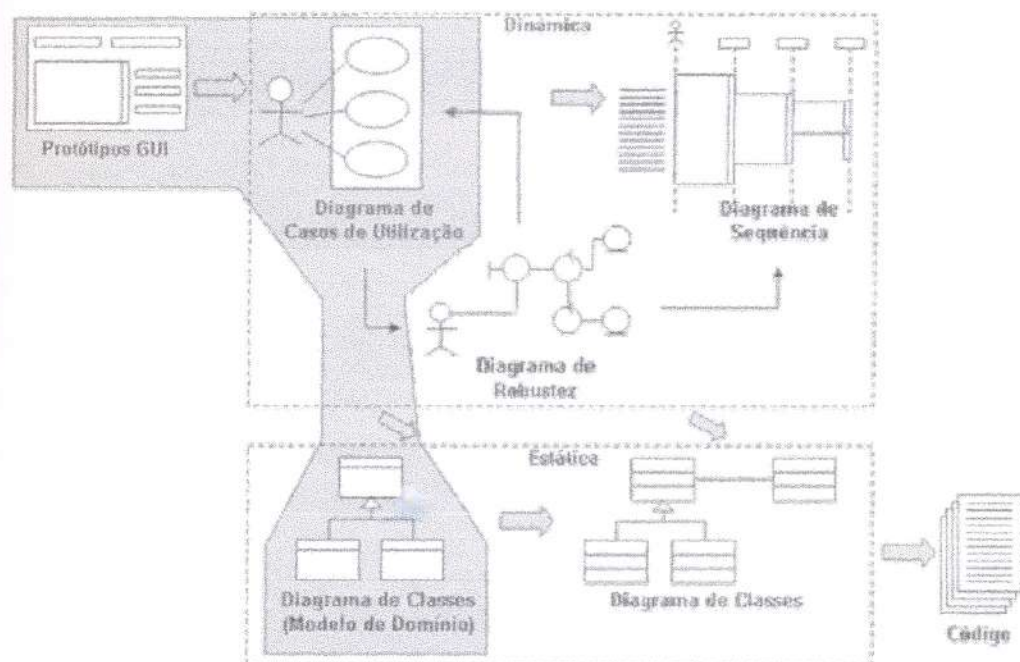


Ilustração 4 Atividades da tarefa de análise de requisitos

- Identificar os objetos do mundo real e todas as relações de generalização, associação e agregação entre esses objetos.
- Desenhar o correspondente diagrama de classes de alto nível, designado por modelo de domínio.
- Se for razoável (e. g. , se for pertinente ou se houver orçamento para tais atividades), desenvolver protótipos de interface homem-máquina (GUI), diagramas de navegação, etc. , de forma que os utilizadores e clientes possam entender melhor o sistema pretendido.
- Identificar os casos de utilização envolvidos no sistema.
- Desenhar os diagramas de casos de utilização realçando os atores envolvidos e as suas relações.
- Organizar em grupos os casos de utilização.
- Capturar essa organização através de diagramas de pacotes (packages).
- Associar requisitos funcionais aos casos de utilização e aos objetos do domínio.

Observação: O ICONIX distingue explicitamente um requisito de um caso de utilização. Em particular, o seu autor distingue-os da seguinte forma:

- Um caso de utilização descreve uma unidade de comportamento.
- Um requisito descreve uma regra que governa o comportamento.
- Um caso de utilização satisfaz um ou mais requisitos funcionais.
- Um requisito funcional pode ser satisfeito por um ou mais casos de utilização.

Ou seja, segundo o ICONIX, há uma relação de muitos-para-muitos entre casos de utilização e requisitos, pelo que tem sentido a última atividade desta tarefa, de associação entre estes dois conceitos. Note-se que existem outros autores que consideram, por outro lado, os casos de utilização o mecanismo ideal para se especificarem os próprios requisitos de um sistema. (Esta relação entre casos de utilização e requisitos é um assunto em discussão na comunidade OO, não existindo até ao momento qualquer consenso).

Análise e Desenho Preliminar

Consiste na realização das seguintes atividades (ver Ilustração 5):

- Fazer as descrições dos casos de utilização com os cenários principais, cenários alternativos e cenários de exceções.
- Fazer a análise de robustez. Para cada caso de utilização: Identificar um primeiro conjunto de objetos. Usar os estereótipos de classes definidos no perfil "Processos de Desenvolvimento de Software" especificado no UML 1. 3 (boundary, control, e entity).
- Atualizar o diagrama de classes do modelo do domínio, com os novos objetos e atributos, entretanto descobertos.
- Terminar a atualização do diagrama de classes de modo a refletir a conclusão da fase de análise.

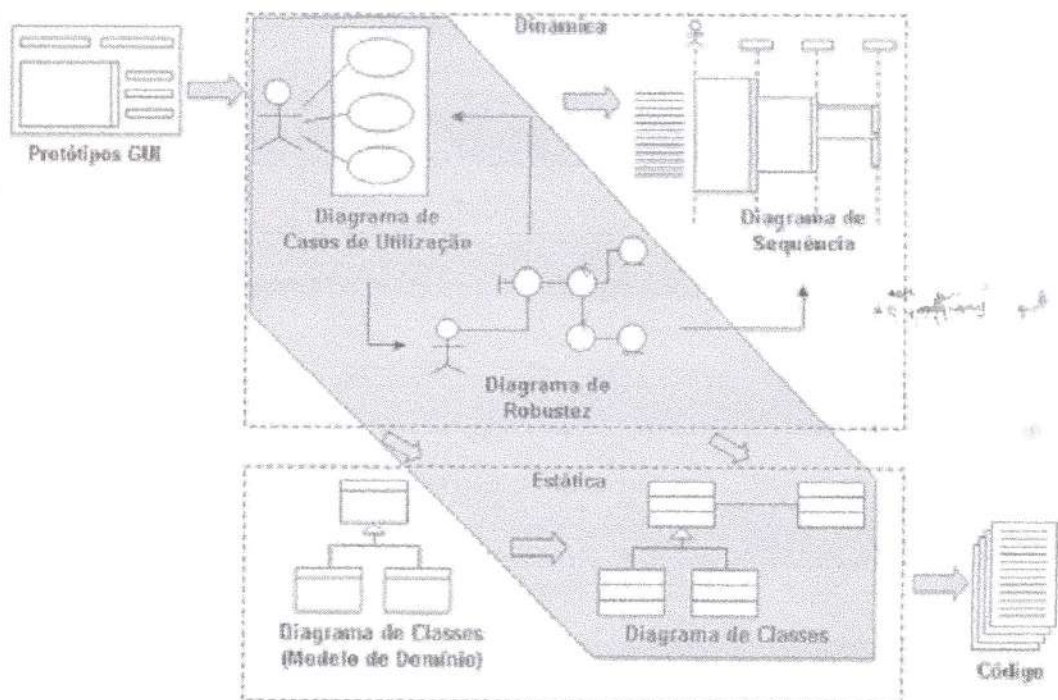


Ilustração 5 Atividades da tarefa de análise e desenho preliminar

Desenho

A tarefa de desenho consiste na realização das seguintes atividades (ver Figura abaixo):

- Especificar o comportamento. Para cada caso de utilização: Identificar os objetos, as mensagens trocadas entre objetos e os métodos associados que são invocados. Desenhar um diagrama de sequência com o texto do caso de utilização do lado esquerdo, e informação do desenho do lado direito. Continuar a atualizar o diagrama de classes com os objetos e atributos, entretanto descobertos.

- Se for relevante, usar diagrama de colaboração para ilustrar as transações principais entre objetos.
- Terminar o modelo estático, adicionando informação detalhada sobre o desenho (e. g. , visibilidade e padrões de desenho).
- Verificar que o desenho satisfaz todos os requisitos identificados.

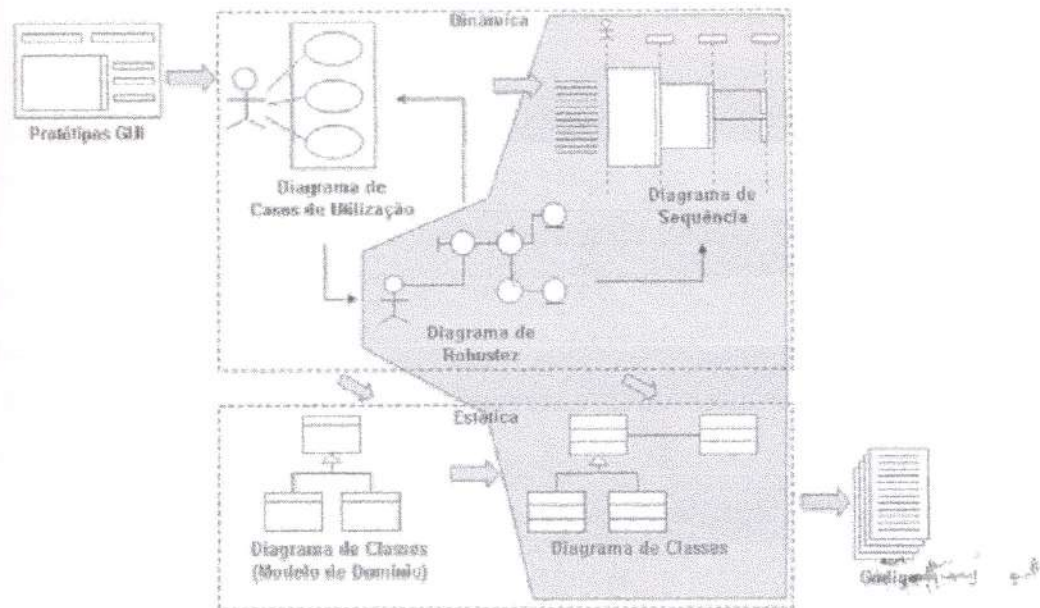


Ilustração 6 Atividades da tarefa de desenho

Implementação

A tarefa de implementação consiste na realização das seguintes atividades (ver Figura abaixo):

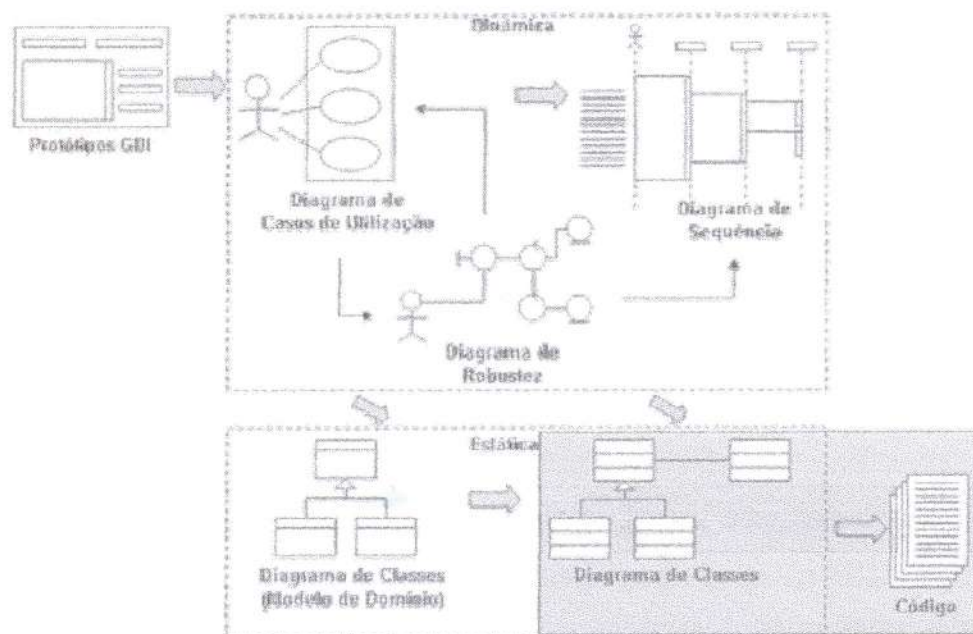


Ilustração 7 Atividades da tarefa de implementação

- Consoante as necessidades, produzir diagramas de arquitetura, tais como, diagramas de instalação e de componentes, que apóiem a tarefa de implementação.
- Escrever e, eventualmente, gerar o código.
- Realizar testes unitários e de integração.
- Realizar testes de sistema e de aceitação do utilizador.

Avisos do Processo ICONIX

O ICONIX lança um conjunto de avisos relativamente a problemas e dúvidas comuns, que deverão ser tidos em conta pelos intervenientes do processo. Os avisos relacionados com a realização da tarefa de análise de requisitos têm como principal objetivo evitar a perda de tempo com detalhes desnecessários. Designadamente:

Na produção dos modelos de domínio:

- Não perder demasiado tempo com a inspeção gramatical.
- Não endereçar o desenho da multiplicidade, demasiado cedo no projeto.
- Endereçar a agregação e composição apenas na fase do desenho detalhado.

Na produção dos modelos de casos de utilização:

- Não começar a escrever os casos de utilização até se conhecer bem como é que os utilizadores irão atuar.
- Não passar semanas a construir modelos de casos de utilização elaborados e bem desenhados, mas a partir dos quais não é possível construir-se um adequado desenho de classes.
- Não perder muito tempo em discussões sobre quando e onde usar relações "include" ou "extend".
- Não usar templates textuais de casos de utilização muito longos ou complexos.

Avisos a ter em conta na realização da tarefa de análise e desenho preliminar:

- Não procurar fazer desenho detalhado nos diagramas de robustez.
- Não perder tempo a tentar aperfeiçoar os diagramas de robustez à medida que o desenho evolui.

Avisos a ter em conta na realização da tarefa de desenho:

Na produção dos diagramas de seqüência:

- Não procurar associar comportamento aos objetos antes de se ter um idéia concreta sobre o seu significado e interesse para o sistema.
- Não começar a desenhar um diagrama de seqüência antes de se ter completado o diagrama de robustez correspondente.
- Não focar a atenção (e esforço) na definição de métodos "get" e "set" em detrimento dos métodos reais. (Estes métodos de acesso e alteração dos atributos podem ser facilmente gerados automaticamente a partir de uma ferramenta CASE).

Na produção dos diagramas de estado:

- Não desenhar diagramas de estados para objetos com apenas dois estados.
- Não modelar o que não é necessário modelar.
- Não desenhar diagramas de estados só porque se consegue desenhá-los.

METODOLOGIA FUSION

A Metodologia (ou Método) Fusion de orientação a objetos foi proposta por Derek Coleman, Patrick Arnold, Stephanie Bodoff, Chris Dollin, Helena Gilchrist, Fiona Hayes e Paul Jeremaes no ano de 1994, embora seu "nascimento" deu-se nos laboratórios da HP em meados de 1993. O método compreende as fases de análise, projeto e implementação (COLEMAN, 1996).

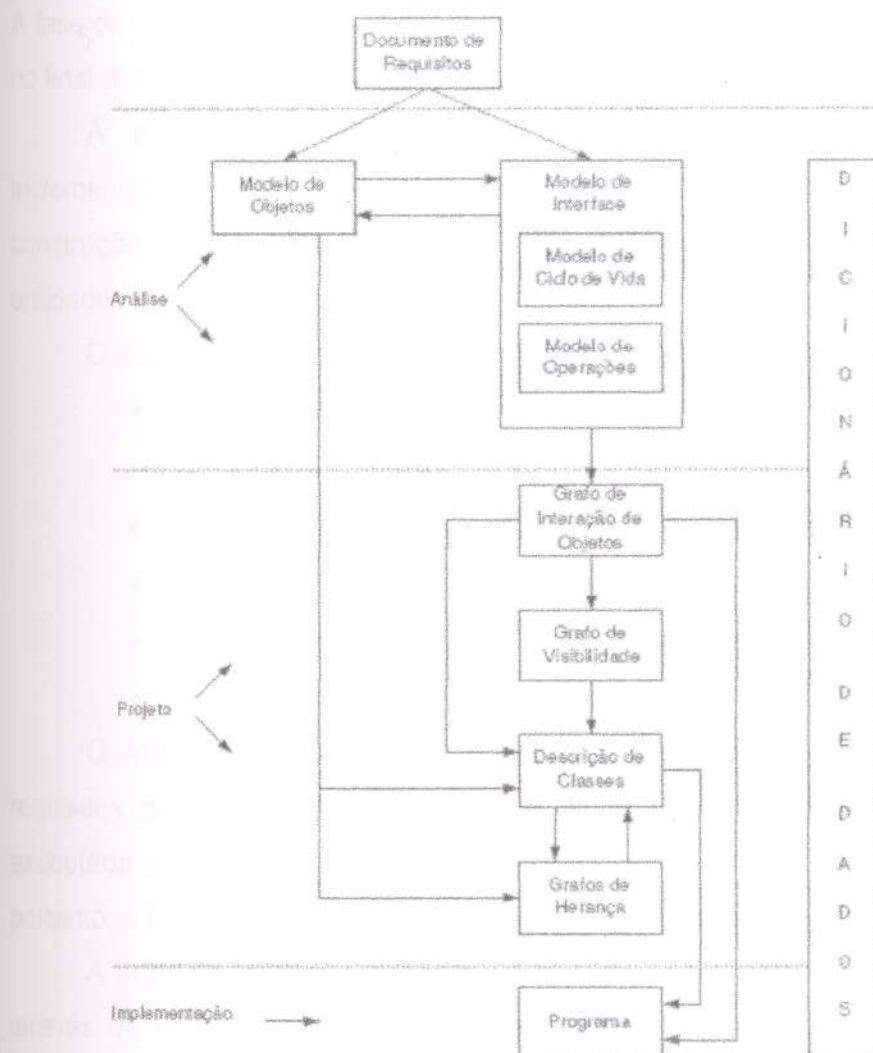


Ilustração 8 Visão Geral do Método Fusion

O Fusion foi pensado para solucionar problemas encontrados nos métodos orientados a objeto de primeira geração, que são a "notação" e o "processo". Portanto, o Fusion fornece um processo para desenvolvimento de software bastante conciso, propondo notações bem definidas, completas e ferramentas para gerenciamento do processo que fazem a verificação e a consistência do mesmo.

O Fusion não encontra dificuldades em desenvolver sistemas seqüenciais. O mesmo não pode ser dito dos sistemas concorrentes, os quais nem todos podem ser bem representados. Neste caso, o próprio Derek Coleman sugere outras técnicas e métodos.

Fase de Análise

A função da fase de análise é, baseada no documento de requisitos, identificar as classes e os objetos existentes no sistema, descrever os relacionamentos e definir as operações que poderão ser executadas pelo sistema.

A fase de análise mostra o “que” o sistema faz e não “como” ele é feito. Obtém-se, no final desta fase, um “documento de especificações”.

A análise é uma fase que deve ser executada de forma iterativa e incremental. Durante todo o processo da análise ela deve ser acompanhada pela construção do dicionário de dados, que tem a função de identificar e descrever as entidades existentes.

O desenvolvimento incremental consiste em:

- Especificações não completas do sistema no início do processo;
- Domínio do sistema crescente com o passar do tempo;
- Análise e projeto realizados iterativamente;
- Modelos são detalhados a cada etapa, sendo baseados em anteriores;
- Alterações ou correções realizadas no decorrer do processo;
- Uso de protótipos.

O Método Fusion parte do princípio de que já exista um documento de requisitos pronto. “Vale lembrar que a captura de requisitos será normalmente executada por um usuário, o qual deve fornecer o documento inicial de requisitos; portanto, o Fusion não possui uma fase de captura de requisitos” (Coleman, 1996).

A importância de um bom levantamento dos dados pode ser exemplificada através de uma frase do físico Albert Einstein, na obra *A Evolução da Física* de 1938: “A formulação de um problema é muitas vezes mais essencial do que sua solução, a qual pode ser meramente uma questão de habilidade matemática ou experimental”.

Dois modelos são gerados pela fase de análise, os quais tratam de aspectos diferentes: o Modelo de Objetos – define a estrutura estática das informações do sistema – e o Modelo de Interfaces – define a comunicação (comportamento) das entradas e saídas do sistema.

Esta fase do processo é responsável pela descrição de:

- Classes de objetos, sem associar quaisquer métodos às classes;
- Relacionamentos entre classes;
- Operações que podem ser executadas;
- Sequência de operações permitidas no sistema.



Ilustração 9 Verificação de intersecções entre modelos

Os modelos da fase de análise se inter-relacionam (Figura acima) através de algumas características, sendo a verificação destes resumida nos seguintes tópicos (COLEMAN, 1996):

Inteireza em relação aos requisitos:

- Todos os cenários estão no ciclo-de-vida;
- Todas as operações do sistema foram definidas;
- Todas as informações estáticas estão compreendidas no modelo de objetos do sistema;
- Todas as outras informações se encontram no dicionário de dados.

Consistência simples:

- Todos os conceitos (classes, relacionamentos, atributos, predicados) estão definidos no dicionário de dados ou no modelo de objetos do sistema;
- A fronteira do modelo de objetos do sistema é consistente com o modelo de ciclo-de-vida;
- Todos os identificadores encontram-se no dicionário de dados.

Consistência semântica:

- Os eventos gerados pelo modelo ciclo-de-vida e pelo modelo de operações devem estar consistentes;
- Modelo de operações deve preservar as definições do modelo de objetos do sistema;
- Conferir os cenários manualmente através dos esquemas.

Vale lembrar que as verificações citadas acima devem ser observadas durante todo o processo de análise.

Fase de Projeto

Decide como representar as operações do sistema através de interpretações entre objetos relacionados e como estes obterão acesso entre si. Como resultado desta fase, obtém-se um "documento de arquitetura". O resultado da fase de projeto também pode ser interpretado como a estrutura de software orientada a objetos contendo as mesmas informações e preservando os relacionamentos do modelo de objetos do sistema. O projeto ainda define como a funcionalidade definida no modelo de operações será implementada. A fase de projeto utiliza-se de 4 modelos:

- Grafos de Interação de Objetos (GIO) - têm a finalidade de definir a seqüência de mensagens que são trocadas entre os objetos de um determinado grupo, quando da realização de uma determinada operação. Para o desenvolvimento de um software mais confiável, deve-se ter um GIO para cada operação do sistema, pois estes fornecem uma representação visual para a futura implementação;
- Grafos de Visibilidade - O objetivo de um Grafo de Visibilidade (GV) é definir as estruturas de referência entre as classes existentes no sistema. Um objeto pode ser tanto cliente como servidor. Todo objeto servidor deve estar visível a todo e qualquer objeto cliente, para que este seja capaz de enviar mensagens aos objetos servidores;
- Descrição de Classes - as informações do modelo de objetos, dos grafos de interação de objetos e dos grafos de visibilidade são consolidadas nas descrições de classes. Neste estágio são estabelecidos para cada classe: superclasse(s) imediata(s), seus métodos, seus atributos de dados e atributos-objeto. Após isto, serão atualizadas as informações relativas à herança;
- Grafos de Herança - O conceito de grafos de herança significa que uma classe pode ser definida como uma especialização de outra.

Fase de Implementação

A Fase de Implementação consiste em codificar o projeto em uma linguagem de programação. A base para isto é composta pela descrição de classes, grafos de interação de objetos e dicionário de dados, da fase de projeto e o ciclo-de-vida gerado na fase de análise.

Embora não haja uma ordem determinada a ser seguida na fase de implementação, é conveniente utilizar-se uma seqüência de eventos. Para ordenar

os eventos, dando preferência primeiramente às operações mais simples, utiliza-se o ciclo-de-vida. A partir de então, implementam-se as classes que irão suportar estas operações. Após, implementa-se os outros eventos, o que poderá vir a forçar a modificação das classes já implementadas, até que se complete as definições preestabelecidas. Como resultado desta fase, tem-se o "código fonte" do sistema requerido.

Uma ferramenta CASE que suporta as notações do Fusion é a WinA&D.

EXTREME PROGRAMMING (PROCESSO DE DESENVOLVIMENTO ÁGIL)

Os processos de desenvolvimento ágil são baseados no "Manifesto de Desenvolvimento de Software Ágil" onde são definidos quatro pontos. Contudo é importante entender que o Manifesto define preferências, não alternativas para o desenvolvimento de software dando ênfase a certas áreas, mas não eliminando outras (AMBLER, 2002) (TELES, 03).

Pontos do Manifesto de Desenvolvimento de Software Ágil:

- Indivíduos e interações são mais importantes que processos e ferramentas: Em um grupo de desenvolvimento de software todos têm que trabalhar efetivamente juntos nas várias fases do processo (programação, teste, modelagem, projeto e customização), isto significa dizer que a comunicação e o entrosamento entre os participantes do projeto são de suma importância, pois isso proporciona uma ação em conjunto da equipe;
- A implementação do software é mais importante do que uma documentação compreensível: ao se perguntar ao usuário se ele quer um documento de mais de 50 páginas que descreva como se pretende construir o software ou se começa logo a construção, o que ele dirá? 99 % de chances de que comece logo a construir o software. Isso significa que é primordial a construção do software e depois pensar em documentação. Havendo, então, uma ênfase muito grande na programação e conseqüentemente uma grande insistência nos testes do sistema.
- A colaboração do cliente é mais importante do que a negociação de um contrato: só o cliente pode dizer o que ele quer, isso significa que os clientes participam diretamente da criação do software, indicando prioridades, ordem das telas, e outros detalhes. É verdade que o

cliente não sabe exatamente especificar o sistema, tão pouco, vai querer participar disso, mas é uma tarefa difícil que terá que ser feita, acostumando e educando o usuário. É claro que o contrato é importante, mas não é mais que a comunicação.

- Resposta à mudança é mais importante do que seguir um plano: o usuário muda de idéia por várias razões, a tecnologia também muda o tempo todo, embora não seja sempre para melhor, e isso reflete no sistema em construção. Por isso o sistema deve ser maleável e os desenvolvedores devem sempre estar dispostos a efetuar mudanças no sistema perante qualquer fato revés que possa vir a ocorrer.

Os Quatro valores do XP

Segundo Teles (2003), o XP (Extreme Programming) é um processo de desenvolvimento ágil que foi idealizado por Kent Beck voltado para:

- Projetos cujos requisitos são vagos e mudam com frequência;
- Desenvolvimento de sistemas orientados a objeto;
- Equipes pequenas, preferencialmente até doze desenvolvedores.
- Desenvolvimento incremental (ou iterativo), onde o sistema começa a ser implementado logo no início do projeto e vai ganhando novas funcionalidades ao longo do tempo.

O XP possui quatro valores fundamentais: Comunicação, Feedback, Simplicidade e Coragem.

Valores Fundamentais do XP:

Comunicação

A comunicação é algo que está contido em todo o ser humano e é fundamental para troca de conhecimento e idéias. Pensando nisso o XP adota a comunicação como um de seus valores. "A Comunicação entre o cliente e a equipe permite que todos os detalhes do projeto sejam tratados com a atenção e a agilidade que merecem" (TELES, 2003). Teles (2003) exemplifica vários tipos de comunicação e como elas influenciam no entendimento de uma pessoa, como conversa face-a-face, via telefone ou via escrita.

O XP procura evitar problemas de comunicação utilizando a conversa face-a-face com o usuário. Procura explorar tanto quanto possível a interação entre as pessoas, de modo a diminuir as falhas de comunicação. Diferente disso, nos

métodos tradicionais de desenvolvimento, o meio escrito é utilizado extensivamente para a comunicação de idéias. Contudo os documentos ainda serão utilizados, mas para documentar o que já foi executado e não o que ainda será executado.

Segundo (AMBLER, 2002) a comunicação é dividida em duas partes: a primeira chamada de Opções de Documentação e a outra chamada de Opções de Modelagem (Figura abaixo), onde o valor de cada tipo varia conforme a situação em que será utilizada.

Nesse caso a comunicação face-a-face também é a melhor, contudo particularmente quando utilizada com outros meios como um quadro magnético, flip chart, papel, etc.

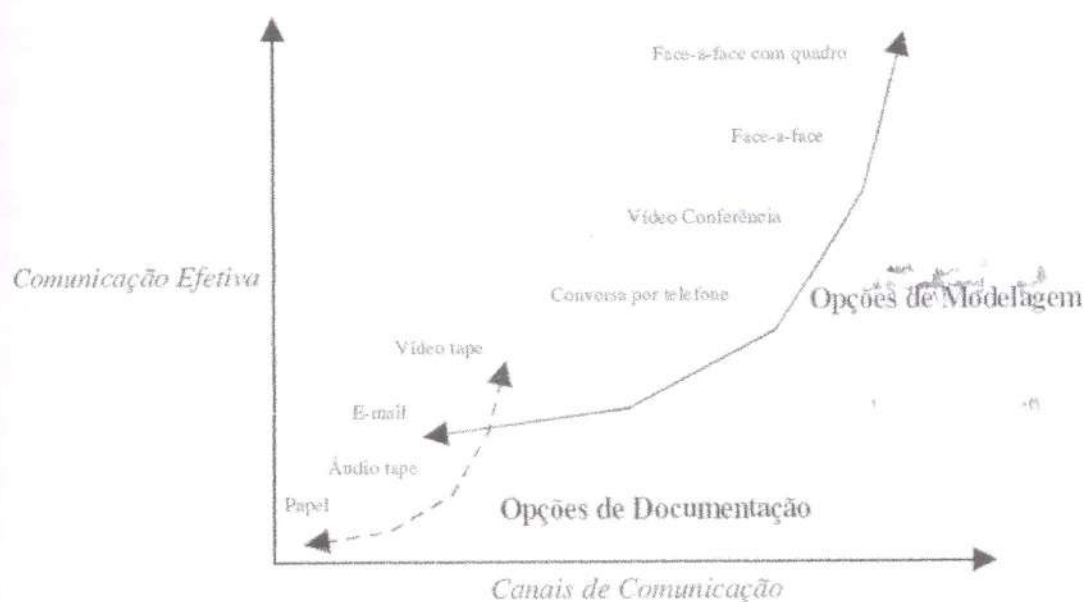


Ilustração 10 Modelos de Comunicação

Feedback

Como saber se o trabalho está no caminho certo? Quando o cliente gera feedback para equipe de desenvolvimento. Ou seja, ele realimenta a equipe com alterações nas necessidades que ainda serão implementadas e, eventualmente, naquelas que já fazem parte do software. Segundo (TELES, 2003) o feedback é o mecanismo fundamental que permite que o cliente conduza o desenvolvimento diariamente e garanta que a equipe direcione as suas atenções para aquilo que irá gerar mais valor.

A idéia principal do XP é ter um feedback rápido, isto significa que o usuário produz uma nova idéia e rapidamente apresenta ao grupo de desenvolvimento, que a analisa, faz algumas alterações, põem-nas no sistema e apresenta ao usuário, que

pode gerar mais idéias; e assim prossegue indefinidamente. Contudo cada ciclo deve ocorrer em dias ou semanas.

O interessante da técnica do feedback é que com um grupo trabalhando junto o resultado é imediato na escala de segundo ou minutos (AMBLER, 2002), e as revisões informais do sistema estão na escala de minutos ou horas. Isso é importante, pois deixa o desenvolvimento mais dinâmico.

Simplicidade

A simplicidade está muito ligada ao feedback, e segundo Teles (2003) "É muito importante que as ações de cada membro da equipe sejam suficientemente simples e restritas. Isso permite que, logo após a execução de uma ação, possa-se obter feedback sobre ela rapidamente". Quando há uma funcionalidade que o usuário deseja que seja implementada, essa deve ser feita de maneira mais simples possível, para que o usuário tenha logo seu pedido atendido e o valide de forma positiva.

Se o projeto for simples e limpo, ele permanece ágil e maleável. Retardando o projeto até que ele seja requerido, podendo fazer com que algo valioso funcione agora. Isso é chamado de "projeto just in time" e vai contra o desenvolvimento tradicional que prescreve o projeto generalizado. O projeto generalizado obriga a criação de um projeto suficientemente flexível para lidar com os requisitos futuros, aumentando as possibilidades de fazer algo que não seja preciso no futuro. E para o XP é melhor, então, esperar até que seja necessário projetar esses requisitos futuros.

Coragem

A coragem é o requisito fundamental no desenvolvimento ágil de software (AMBLER, 2002). Primeiro porque precisa ter coragem para escolher o desenvolvimento ágil, já que é algo que difere dos padrões atuais. Segundo porque durante o desenvolvimento deve-se ter coragem para tomar importantes decisões, na parte de arquitetura, programação, entre outros. Terceiro, reconhecer que você pode falhar e que você comete erros. E finalmente o quarto motivo: confie que você vai superar esses problemas futuros.

Segundo Teles (2003) a equipe de desenvolvimento deve ter coragem para:

- a) Desenvolver o software de forma incremental;
- b) Manter o sistema simples;
- c) Permitir que o cliente priorize as funcionalidades;

- d) Fazer os desenvolvedores trabalharem em par;
- e) Investir tempo em *Refactoring*;
- f) Investir tempo em testes automatizados;
- g) Estimar as user stories na presença do cliente;
- h) Expor o código a todos os membros da equipe;
- i) Integrar o sistema diversas vezes ao dia;
- j) Adotar um ritmo sustentável;
- k) Abrir mão de documentações que servem como defesa;
- l) Propor contratos de escopo variável;
- m) Propor a adoção de um processo novo.

Princípios do Extreme Programming

É importante compreender os valores que regem o XP (ASTELS, 2002). Pois eles são base para a maioria dos processos que serão desenvolvidos no futuro, contudo alguns desses valores são apenas bom-senso.

Trabalhando com o Cliente

Há uma insistência no XP em fazer o cliente real trabalhar diretamente no projeto.

Esse cliente tem que ser capaz de tomar decisões no âmbito de recursos, riscos, e assim por diante e de responder perguntas da equipe. E como já dito anteriormente, o feedback será constante (circular), proporcionando um melhor andamento do desenvolvimento, pois o cliente também é quem aceita ou não o que está sendo produzido naquele momento (teste de aceitação).

Usar metáforas para descrever conceitos difíceis

Uma metáfora é uma forma poderosa de relacionar uma idéia difícil em uma área desconhecida (ASTELS, 2002). Os projetos em XP usam metáforas para orientá-los e fornecerem texto compartilhado. Dessa forma todas as pessoas que estão envolvidas no desenvolvimento compartilham uma visão geral do sistema e do problema que deverá ser solucionado.

A metáfora também serve como uma definição da arquitetura conceitual geral do sistema. A metáfora pode ser uma representação simples de um conceito ou recurso, mas não significa que serão precisas em todos os aspectos. Em vez disso, fornecem um contexto para discutir problemas e suas soluções.

Planejamento

Antes de se construir um software, deve-se planejá-lo. Isso ajuda na compreensão mútua por todos os envolvidos no projeto. Também serve para medir o tamanho do projeto em relação à capacidade da organização, ou seja, se o projeto é maior que a capacidade da equipe e quanto tempo vai levar.

O planejamento serve também para saber por onde se está indo, como se chegará ao final e quais os riscos podem ser encontrados no decorrer do caminho. Mas o mais importante: ele mudará à medida que os requisitos e os riscos mudarem e quanto mais longe do futuro for o planejamento, maiores as chances de que ele seja impreciso.

Escrevendo User Stories

Uma user story é a menor quantidade de informações (uma etapa) necessária para que o cliente defina um caminho através do sistema. É através das user stories que são extraídos os requisitos do sistema. Todas as funcionalidades do sistema são levantadas através dessas user stories que são registradas em pequenos cartões (Ilustração 11) . Esses cartões são escritos pelo cliente, por algumas razões:

- a) Ao escrever uma user story, o cliente é forçado a pensar melhor na funcionalidade, pois ele formaliza o pensamento e analisa melhor o assunto sobre qual irá tratar. Sendo recomendáveis user stories simples e curtas, com apenas o suficiente para transmitir a idéia desejada;
- b) Quando o cliente escreve o cartão, ele cria um vínculo com ele. Trata-se de um vínculo psicológico, mas de grande importância, fazendo o cliente sentir responsabilidade sobre aquilo que está sendo solicitado;
- c) Através do cartão o cliente compreende que existe um custo em tudo que é pedido, ou seja, que ele tenha consciência deste custo para que ele saiba selecionar aquilo que deve ou não ser solicitado a cada momento, ou que ele possa priorizar.

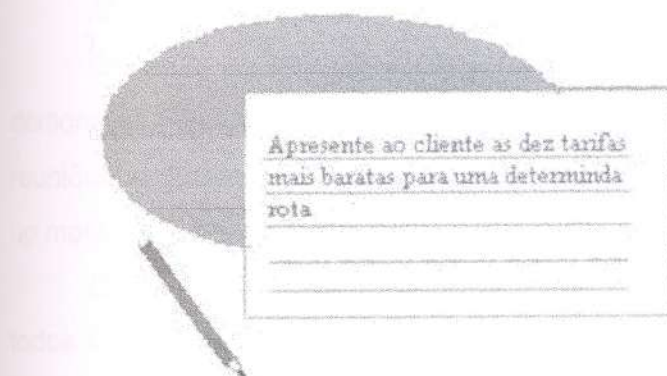


Ilustração 11 Uma user story em um cartão

É importante ressaltar que a descoberta de user stories é um processo constante durante todo o projeto XP. Um conjunto inicial de user stories é criado no início do projeto. Em seguida, novas user stories podem ser criadas como resultado de novos requisitos. Elas também podem ser criadas para refinar os itens a serem entregues à medida que novas expectativas de funcionalidades existentes são descobertas.

A equipe de desenvolvimento utiliza os cartões para saber quais são as funcionalidades desejadas pelo cliente. Os desenvolvedores escolhem as user story que irão implementar a cada dia de trabalho. Contudo, quando o projeto é muito grande, os cartões podem representar um grande esforço para serem implementados. Neste caso é comum dividir os cartões em tarefas, que são registradas em novos cartões. Assim os desenvolvedores implementam tarefas ao invés de user story.

Estimando User Stories

Para que o cliente possa priorizar as funcionalidades, é preciso que elas sejam registradas na forma de user story, as quais precisam ser estimadas, para que ele possa saber o custo previsto de cada uma delas. Essa estimativa deve ser feita através de uma unidade de medida.

O XP utiliza o conceito de dia ideal de desenvolvimento, que representa um dia no qual o desenvolvedor só trabalha na implementação de funcionalidades. Ou seja, quando um desenvolvedor estima, ele quer saber quanto tempo irá levar para implementar aquela user story, sem ser atrapalhado por uma atividade externa (reuniões, atender telefone, etc.).

Manter as reuniões curtas

Em geral as reuniões em um projeto de desenvolvimento convencional são demoradas, desgastantes e aborrecem os participantes. O XP também possui reuniões, contudo são rápidas, por volta de dez minutos e realizadas em pé (stand up meeting) e no início do dia.

O objetivo é informar a todos o que foi feito no dia anterior, fazendo com que todos fiquem atualizados sobre o andamento do projeto. Por consequência, isso traz a oportunidade de avaliar o trabalho que está sendo feito.

O segundo objetivo é decidir o que será feito no dia que se inicia, priorizando as atividades dos membros da equipe. É importante frisar que essas reuniões devem ocorrer todo dia.

Teste

O teste é outra peça central da filosofia XP (ASTELS, 2002) porque a partir dos testes se fornece uma definição ou documentação do comportamento desejado do sistema e se a funcionalidade está corretamente implementada podendo ir ao próximo problema.

Os testes ocorrem em diversos níveis:

- a) Teste de unidade: é escrito antes do código pelos desenvolvedores de software. A finalidade desse teste é avaliar se uma pequena parte da funcionalidade funciona como devido.
- b) Teste de aceitação: esse teste procura simular a interação do usuário com o sistema e verifica se o seu comportamento está correto. É como um roteiro que tem vinculado a ele um conjunto de respostas esperadas.

Design Simples

A premissa de um projeto simples está focada segundo Astels (2002) em projeto para o momento, ou seja, fazer as coisas o mais simples possível.

Segundo Beck (2000), nas últimas décadas a comunidade de desenvolvimento investiu, no sentido de reduzir custos de alteração de software e a consequência é:

- a) Melhores linguagens de programação;
- b) Avanços tecnológicos de banco de dados;
- c) Melhores práticas de programação;
- d) Novos ambientes e ferramentas de desenvolvimento;
- e) Novas notações;

- f) Computadores mais velozes, baratos e com grande capacidade de armazenamento;
- g) Orientação a objetos.

Para somar a isso o XP parte do princípio que o custo de uma alteração no software cresce mais lentamente e tende a se estabilizar ao longo do tempo se tornando quase linear.

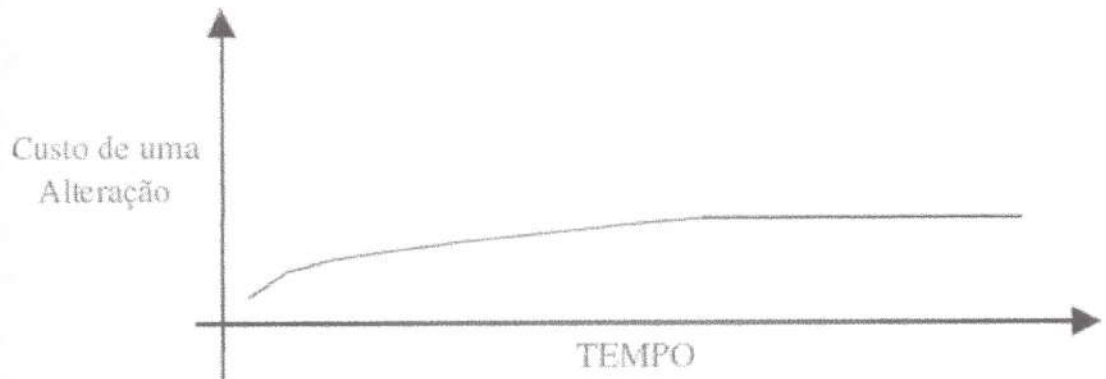


Ilustração 120 O custo de uma alteração não cresce muito ao longo do tempo

Isso significa que se o custo se mantém estável ao longo do tempo, não se faz necessário colocar todas as funcionalidades futuras possíveis. Isso torna o desenvolvimento mais rápido, simples e com maior feedback do usuário.

Programação em par

A programação em par, segundo Teles (2003) e Astels (2002), é uma técnica na qual dois programadores trabalham em um mesmo problema, ao mesmo tempo em um mesmo computador. Onde uma pessoa (o condutor) tem controle do teclado e do mouse e o outro (o navegador) observa-o e ajuda-o.

O primeiro escreve ativamente os testes, codifica, integra ou faz *Refactoring*, o outro observa, procurando possíveis erros (sintaxe e grafia), oferece novas sugestões e idéias, além de pensar em outros tipos de teste. É importante lembrar que os papéis não são fixos, pode haver troca, com navegador virando condutor e vice-versa.

Codifique dentro dos padrões

Significa utilizar o padrão mais comum na hora do desenvolvimento propriamente dito (codificação). Por exemplo, se utilizar Java usar o padrão "Sun Microsystems" ou decidir coisas mais simples como abrir chave "{" no final da declaração ou uma linha abaixo.

A consequência dessa padronização é (ASTELS, 2002):

- a) Você pode programar aos pares porque ambos os parceiros trabalham no mesmo formato;
- b) Todos podem ter a propriedade do código porque todos escreveram o código com o mesmo estilo;
- c) O *Refactoring* pode ser feito, pois o programador entende o formato do código que ele está alterando e os demais entenderão o formato do seu trabalho;
- d) A velocidade total aumenta, porque não é necessário reformatar o código durante o trabalho, nem estar tentando alterar o código para outro estilo ou tentando entender a formatação e o estilo, bem como o próprio código.

Código Coletivo

Nos métodos tradicionais, há uma divisão do trabalho na construção do código, ocasionando inúmeros problemas, como pode acontecer se o responsável por aquela parte do código, ficar doente ou sair da equipe de desenvolvimento, ninguém será capaz de mexer naquele código que era de sua responsabilidade, ou se for capaz terá grande dificuldade no início. Outro problema é qualidade e a legibilidade do código, pois com apenas um na construção do código não é possível saber se ele é legível ou não.

A idéia do XP é então fazer um código coletivo, onde cada parte dele pode eventualmente sofrer a visita de diferentes desenvolvedores ao longo de sua vida. Como consequência o código será sempre revisado.

Integração Contínua

Para trabalhar com maior velocidade deve-se fazer a integração contínua do trabalho feito ao código fonte base de produção. Isso deve ocorrer diariamente. Em outras palavras, após o término de uma tarefa deve-se unir com o "programa principal". Fazendo testes e certificando-se de que todo o sistema está funcionando com sucesso.

Segundo Jeffries (2001) o processo de integração é feito corretamente quando ele não obstrui o avanço da equipe:

- a) Deve ser fácil e rápido obter o código fonte do qual você necessita;
- b) Deve ser fácil e rápido armazenar as mudanças;
- c) O repositório (ferramenta que armazena o código fonte do sistema e controla a versão dos arquivos) deve detectar conflitos e é importante que seja simples resolvê-los

- d) Não deve haver espera, isto é, se um par de programadores precisa ditar alguma coisa, ele deve seguir adiante sem que nada o impeça.

Refactoring

Segundo Flower (1999) *Refactoring* é o processo de alterar um sistema de software de tal forma que ele não altere o comportamento externo do código apenas melhorando sua estrutura interna. Essa é uma forma disciplinada de limpar o código que minimiza as chances de introdução de bugs. Em outras palavras, se um desenvolvedor tem uma dúvida em uma parte do código ele deve procurar compreendê-lo e, em seguida, reescrevê-lo de modo que fique mais legível, apenas alterando a funcionalidade, dado que o código continua fazendo a mesma coisa.

No XP isso vai um pouco além, pois o desenvolvedor é obrigado a fazer *Refactoring* o tempo todo de qualquer parte do código que esteja pouco legível. As vantagens do *Refactoring* é que ele mantém a legibilidade do código, torna mais simples a prática do código coletivo, pois todos os desenvolvedores conseguem ler qualquer parte do código facilmente e igualmente o código coletivo ajuda a identificar a necessidade de *Refactoring*.

Releases Curtos

Um release é um conjunto de funcionalidades implementadas que representam um valor bem definido para cada cliente e que é colocado em produção para que todos os usuários possam se beneficiar dele. Em outras palavras, colocar nas mãos do cliente uma versão pequena do sistema.

Diferente das abordagens tradicionais onde o cliente recebe o sistema quando ele está finalizado, o XP fornece releases pequenos ao cliente em um curto espaço de tempo em relação ao desenvolvimento tradicional de um a seis meses em geral. A vantagem disso é que o cliente começa a receber os benefícios do sistema muito antes de estar completamente finalizado, além de fornecer um valioso feedback do usuário se o sistema está indo no caminho certo, e finalmente detecta todos os defeitos o mais cedo possível.

Ritmo Sustentável

No desenvolvimento tradicional a restrição no tempo leva as equipes de desenvolvimento a trabalharem em excesso, trabalhando diariamente até mais tarde e invadindo fins-de-semana e feriados. A grande maioria dos projetos de software passa por dificuldades de cumprimento de cronograma e, invariavelmente, a solução adotada é fazer as pessoas trabalharem além do horário.

Ninguém faz seu melhor trabalho quando está estressado, sob pressão, cansado. O desenvolvedor com esses problemas não consegue se concentrar bem no problema que está resolvendo. Em vez de dez minutos para resolver um problema acaba ficando uma hora para encontrar a solução. Além da possibilidade de que o código seja agraciado com "pequenos" errinhos.

Por isso o XP tenta evitar que as pessoas tenham estafa, fazendo-as trabalhar apenas oito horas por dia com uma certa quantidade de descontração no trabalho, procurando fazer com que a equipe se livre do estresse mantendo todos relaxados. A idéia é não trabalhar mais do que aquilo que funciona para a equipe

METODOLOGIA

Estudo de caso, analisando como a empresa em questão está estruturada;

Entrevistas com a equipe e desenvolvimento;

Pesquisa bibliográfica com a finalidade de levantar contribuições de autores sobre o assunto.

RESULTADOS

Para propor-se esta metodologia verificou-se como se dá o processo de desenvolvimento de software na empresa e, baseando-se na análise de diversas metodologias existentes, sugere-se uma que se adequará às necessidades da empresa.

PAPÉIS DOS MEMBROS DA EQUIPE

Na empresa há um desenvolvedor que mantém contato com os clientes, recebendo as solicitações e as repassando aos outros desenvolvedores, de acordo com a sua experiência. Todos os desenvolvedores além de implementarem, também testam e documentam, tanto o código quanto o que foi desenvolvido.

Dessa forma, recomendam-se os seguintes papéis a serem desempenhados pelos membros da equipe:

- Gerente de projeto

Responsável pelos assuntos administrativos do projeto, mantendo um forte relacionamento com o cliente para que o mesmo participe das atividades do desenvolvimento. Deverá filtrar assuntos não relevantes aos desenvolvedores e aspectos que só devam ser implementados em iterações futuras.

É também responsável pelas questões técnicas do projeto. Recomenda-se que esta pessoa seja a que tem o maior conhecimento do processo de desenvolvimento, dos valores e práticas desta metodologia apresentada, pois deverá verificar o comportamento da equipe, sinalizando os eventuais erros cometidos.

Também é de sua responsabilidade definir os testes de aceitação dos recursos solicitados. Devendo ajudar o cliente a elaborar os casos de testes e, no final de cada iteração, verificar se o software atende todos os testes pré-estabelecidos.

Orientará o Redator Técnico quanto à documentação do software e testes.

É importante que o Gerente de Projeto tenha um estreito relacionamento com o cliente e que acompanhe os passos dos desenvolvedores, onde a sua ausência representará sérios riscos ao projeto.

- Redator técnico

Responsável pela documentação do sistema, evitando um forte trabalho dos desenvolvedores neste tipo de atividade, permitindo uma maior dedicação ao trabalho de codificação.

Esta pessoa deve estar em plena sintonia com os desenvolvedores e Gerente de projeto para que a documentação reflita o código escrito e as regras de negócio atendidas pelo sistema.

É importante salientar que essa documentação é destinada ao cliente.

O Redator Técnico atuará, também, como testador. Recomenda-se que não conheça o código escrito e, preferencialmente, que não tenha conhecimento de desenvolvimento.

- Desenvolvedor

Responsável por analisar, projetar e codificar o sistema.

PLANEJAMENTO

As funcionalidades do sistema devem ser descritas brevemente em histórias em conjunto com os testes conceituais e serão estes os indicadores para uma boa implementação.

Ao terminar uma história, a mesma deverá ser validada rapidamente com o Gerente de Projeto e a equipe deve receber o *feedback* necessário sobre a funcionalidade, criando ciclos rápidos e precisos.

Quando um *release* estiver pronto, o Gerente de Projeto deverá entrar em contato com o cliente para lhe apresentar o que foi implementado e colher seu *feedback*, gerando a partir daí novas histórias.

Como a empresa tem clientes em todo o país é impossível tê-los fisicamente presentes. Sugere-se, então, que sejam agendadas reuniões em que se use ferramentas de vídeo-conferência para que a equipe possa fazer os debates necessários.

Deve-se ter bastante atenção ao planejamento para assegurar que a equipe de desenvolvimento esteja trabalhando naquilo que gere o máximo de valor para o cliente. Este planejamento deve ser feito várias vezes durante o projeto. É o momento onde o Gerente apresenta as funcionalidades desejadas através de histórias, onde a equipe estima o custo de cada história e planeja os *releases* e as iterações.

Todas as funcionalidades do sistema devem ser descritas em histórias, pequenos cartões em que o cliente deve descrever o que deseja com suas palavras e da forma mais simples possível.

Após a definição das histórias é necessário estimar o tempo das mesmas para que o Gerente priorize o que deve ser implementado. Deve-se utilizar uma unidade

referente a um *dia de trabalho ideal* do desenvolvedor, onde o mesmo não precisaria atender telefonemas, participar de reuniões, ou seja, estaria preocupado apenas com a história em questão.

Muitas vezes algumas histórias consumirão semanas de trabalho, oferecendo uma certa dificuldade de serem estimadas. Recomenda-se que estas histórias sejam quebradas em tarefas menores e que as mesmas não utilizem mais que alguns dias de um desenvolvedor. Recomenda-se 4 dias no máximo.

Em cada história deve ser anotada a quantidade de dias estimada pelo desenvolvedor. Recomenda-se que as estimativas sejam efetuadas em equipe e se possível com a presença do cliente para que eventuais dúvidas sejam sanadas.

RELEASES

O objetivo é gerar valor para o cliente ao longo do projeto, para isto o software deve ser desenvolvido de forma incremental, onde a cada entrega o cliente possa utilizar o sistema e obter benefícios do mesmo. Estas entregas são denominadas *releases*, e devem ser feitas em no máximo 1 mês.

A divisão dos *releases* deve ser efetuada no início do projeto, com tamanhos fixos e pequenos. Após esta divisão o Gerente definirá as histórias que farão parte dos *releases* tentando-se evitar um planejamento muito longo, pois na entrega de cada *release* o cliente aprenderá com o sistema e possivelmente irá alterar as histórias para o próximo *release*.

ITERAÇÕES

Mesmo os *releases* sendo efetuados em um curto espaço de tempo, continua representando um tempo muito longo para o *feedback* do cliente. Por esta razão os *releases* devem ser divididos em espaços menores, chamados de *iterações*.

Uma iteração contém um conjunto de histórias a serem implementadas, com duração de uma semana, onde ao final da mesma o cliente possa validar as implementações efetuadas e fornecer o *feedback* necessário à equipe.

Durante as iterações, o desenvolvimento das histórias deve seguir uma ordem bem definida. A cada nova história, os desenvolvedores devem executar as seguintes etapas:

- A história é discutida em conjunto por toda a equipe que busca identificar o maior número possível de detalhes. Nessa fase, normalmente se identificam questões em aberto que são anotadas para serem perguntadas ao requerente.

- Os desenvolvedores ajudam o Gerente a identificar testes de aceitação para a história, os quais são anotados e validados mais tarde com o requerente. Este, por sua vez, eventualmente acrescentará outros testes de aceitação para a história.
- Finalmente, a equipe começa a implementar os aspectos dinâmicos da funcionalidade, adicionando código ao sistema. Esse processo assegura que os testes de aceitação sejam planejados antes da implementação da funcionalidade, o que leva os desenvolvedores a identificar cedo todos os fluxos aceitáveis e todas as exceções que poderiam ser geradas e, portanto, teriam que ser tratadas. Assim, quando a implementação começa, já é direcionada para cuidar dos fluxos aceitáveis, bem como para tratar as exceções, evitando que os usuários sejam surpreendidos por bugs do sistema caso tentem executar operações indevidas. Tais operações devem ser previstas no sistema e tratadas de modo que o usuário possa sempre receber uma mensagem adequada quando tentar executá-las, ao invés de uma mensagem de erro genérica.

Além disso, a utilização de protótipos cada vez mais refinados será útil para permitir que o cliente visualize detalhes da funcionalidade de forma rápida, permitindo que ele peça ajustes e estes sejam incorporados com baixo custo. Ainda assim, é comum o requerente identificar ajustes após a funcionalidade ser completamente implementada.

As iterações devem começar em uma terça-feira e terminar em uma segunda-feira. No primeiro dia de cada iteração deve haver uma reunião de planejamento que é usada para que o Gerente priorize as histórias a serem implementadas na iteração. No último dia, o Gerente deve validar todas as funcionalidades e percorrer todo o plano de testes de aceitação elaborado e refinado ao longo da iteração que está sendo finalizada. Estando de acordo com o planejado, o Gerente deve agendar a entrega ao cliente.

PLANEJAMENTO

Nas reuniões de planejamento, o Gerente deve selecionar um conjunto de cartões contendo as histórias consideradas candidatas para serem implementadas na iteração. Se necessário, deve criar novos cartões em função do aprendizado obtido nas iterações anteriores. Ou eliminar cartões que passe a considerar

desnecessários. Os desenvolvedores devem estimar os cartões candidatos. Isso deve ser feito em conjunto, de modo que todos os desenvolvedores possam opinar sobre a quantidade de esforço necessária para cada cartão. Finalmente, todos os cartões devem ser limitados a um máximo de três dias de trabalho. Cartões estimados acima deste número devem ser divididos.

De posse das estimativas, o Gerente deve selecionar um conjunto de cartões que possa gerar o máximo de benefício para o Cliente dentro do orçamento disponibilizado pela equipe de desenvolvimento.

O orçamento é representado pela velocidade da equipe na iteração anterior. Para descobrir este valor, a equipe deve registrar seu progresso diariamente no **quadro de acompanhamento diário**, conforme exemplificado na ilustração abaixo.

História	Estimativa	1	2	3	4	5	6	7	8	9	10	Total	Desvio
1. Listar vãos entre duas cidades	3												
2. Ordenar por preço	1.75												
3. Ordenar por número de conexões	3												
4. Ordenar por número de escalas	1.75												
5. Buscar tarifa de um voo	2												
6. Buscar tarifa mais barata de um trecho	2.5												
7. Buscar 10 tarifas mais baratas de um trecho	2												
8. Marcar assento de um passageiro	1												
Total	17												
Tarefas Extras													
Total													

Ilustração 13 Quadro de acompanhamento diário

Feita a priorização, os cartões da iteração devem ser posicionados em um **quadro de tarefas**. A ordem em que são colocados no quadro deve refletir a ordem definida pelo Gerente para o desenvolvimento dos mesmos. Tal ordenação deve ser respeitada pelos desenvolvedores ao longo da iteração, de modo que as funcionalidades mais importantes sejam sempre as primeiras implementadas.

Data de solicitação	Tarefa	Situação (Espera, Andamento, Finalizado)

Ilustração 14-Quadro de Tarefas

Ao final da reunião de planejamento, o Gerente deve atualizar o quadro de tarefas e o quadro de acompanhamento diário.

CONCLUSÃO DA ITERAÇÃO

A finalização de uma iteração deve ser feita com a aceitação do requerente, utilizando as funcionalidades implementadas na iteração e percorrendo o plano de

testes de aceitação para assegurar que todas as histórias tenham sido implementadas corretamente. Caso o requerente peça ajustes de última hora, estes devem ser transformados em cartões para iterações seguintes.

Na reunião de finalização o Gerente deve fazer uma retrospectiva com a participação dos Clientes. Trata-se de uma técnica estruturada para avaliação da iteração, de modo a indicar os aspectos da iteração que funcionaram bem e aqueles que precisam ser melhorados. Ao final da retrospectiva, faz-se a priorização dos principais pontos que necessitam de melhorias e todos propõem ações para solucionar os problemas na iteração que se inicia. Tais ações devem ser registradas em um quadro branco de modo que a equipe possa recordar-se delas ao longo da iteração e colocá-las em prática. Este mecanismo é útil para melhorar continuamente o sistema e a forma como a equipe se estrutura para implementá-lo.

REUNIÃO DIÁRIA

Diariamente a equipe de desenvolvimento deve realizar uma reunião por volta das 8:30h da manhã, com uma duração em torno de 10 a 20 minutos, para discutir tudo o que foi feito no dia anterior e priorizar as atividades do dia que se inicia. Além disso, essa reunião também deve ser usada para que os desenvolvedores atualizem o Quadro de Acompanhamento Diário.

Estas reuniões são úteis e devem ser feitas diariamente, durante todo o projeto. Muitos problemas serão resolvidos rapidamente nestas reuniões. Além disso, os desenvolvedores conseguem ter uma boa noção do todo a partir dos relatos de seus colegas.

A reunião deve ser realizada com os integrantes em pé. Segundo estudos uma reunião em pé é mais rápida, evita bate-papos paralelos e faz os integrantes irem diretamente ao assunto e tem por objetivo atualizar a equipe sobre o que foi implementado no dia anterior e trocar experiências das dificuldades enfrentadas. Neste momento também são decididas as histórias que serão implementadas no dia e, em conjunto, definido o responsável por cada uma delas.

CODIFICAÇÃO

Todo desenvolvedor ao encontrar um código duplicado, pouco legível, mal codificado, sem padronização, lento, com código legado ou uso incorreto de outras implementações, tem por obrigação alterar este código deixando-o mais legível e simples, porém esta alteração não pode mudar o comportamento do código em questão.

Esta prática anda de mãos dadas com o código coletivo, já que todo desenvolvedor tem a possibilidade de melhorar qualquer código do sistema.

A padronização oferece facilidades aos desenvolvedores no momento de implementar novas funcionalidades ou efetuar qualquer tipo de manutenção, uma vez que o código se encontra simples e claro.

Uma questão importante é que a prática de refatoração está apoiada pelos testes, pois facilmente o desenvolvedor terá um *feedback* se a alteração por ele efetuada irá gerar qualquer tipo de comportamento anormal no sistema, sofrendo o aprendizado sobre a alteração por ele efetuada.

É importante que a equipe utilize a refatoração em seu dia-a-dia. Entretanto, as refatorações são desprovidas de um mecanismo de proteção essencial. Por isso, poderá haver situações em que refatorações gerem bugs que não são detectados rapidamente.

Durante um dia de trabalho, é comum que os desenvolvedores executem uma grande quantidade de pequenas refatorações à medida que evoluam. Poucas serão as situações em que os desenvolvedores farão grandes refatorações que consomem muito tempo e geram grandes impactos na arquitetura.

TESTES

Todo código implementando deve coexistir com um teste, assim garantindo o pleno funcionamento da nova funcionalidade.

É com base nestes testes que qualquer desenvolvedor terá coragem para alterar uma parte do código que não tenha sido implementada por ele, já que executando os testes poderá verificar se a modificação efetuada alterou o comportamento do sistema.

Com a implementação de testes o desenvolvedor poderá amadurecer o entendimento sobre o problema que irá solucionar, já que os testes devem ser codificados antes da nova implementação.

Há dois tipos de testes, os testes de unidade e de aceitação. O teste de unidade tem por objetivo verificar se os resultados gerados por cada classe estão corretos, já o teste de aceitação tem por objetivo verificar se a interação entre as classes que implementam uma funcionalidade (estória) atende a necessidade de forma correta. Os testes de aceitação são descritos pelo cliente e implementados pelos desenvolvedores, facilitando ainda mais a interação entre as partes.

O desenvolvimento orientado a testes normalmente representa uma parte técnica difícil, especialmente para pessoas habituadas a trabalhar dentro da forma tradicional de desenvolvimento, isto é, implementar e depois testar. Geralmente os desenvolvedores não possuem experiência, nem conhecimentos sobre como implementar testes.

Portanto, é possível que haja deficiência nos testes. Por isso é importante que a equipe execute testes manuais com boa frequência, o que evitará uma incidência elevada de bugs.

A deficiência nos testes fará com que a equipe produza um número de bugs considerado significativo a cada iteração. Tais bugs devem ser detectados com rapidez através dos testes.

Na empresa X, é comum dividir o projeto em partes e colocar responsáveis por cada uma destas partes. Porém apenas uma pessoa torna-se conhecedora daquela parte.

Este tipo de divisão traz sérios problemas ao projeto, uma vez que se aquela parte necessitar inúmeras alterações, apenas uma pessoa estará capacitada para alterá-la. Com estas inúmeras alterações, esta pessoa pode se tornar um gargalo no projeto.

Deve-se travar uma batalha contra este tipo de divisão, já que não existe uma pessoa responsável por uma parte do código. Cada desenvolvedor deve ter acesso a qualquer parte do sistema e ter liberdade para alterá-la a qualquer momento e sem qualquer tipo de aviso.

Esta prática tem como consequência um código revisado por diversas pessoas e caso algo não esteja claro, com certeza será alterado por alguma pessoa (*Refatoração*) para que o mesmo torne-se legível.

O código coletivo permite que a equipe avance com agilidade, na medida em que não será necessário esperar por outros desenvolvedores sempre que haja a necessidade de editar um arquivo do sistema. Além disso, o revezamento entre as diversas funcionalidades permitirá que os desenvolvedores tenham vivência em todos os aspectos do projeto.

O código é uma forma da equipe se comunicar. Uma forma clara de se comunicar através do código, é manter um padrão no projeto para que qualquer um possa rapidamente entender o código lido.

Recomenda-se a adoção de um padrão desde o início do projeto, preferencialmente padrões utilizados pela comunidade da linguagem de desenvolvimento. Havendo a necessidade de modificações e adaptações durante o projeto, que visam agilizar o desenvolvimento, as mesmas devem ser efetuadas.

Além disso, a padronização do código ajudará os desenvolvedores a manter o código coletivo.

Para que o maior valor possível seja gerado para o cliente, um *design* do sistema deve ser implementado da forma mais simples possível para que atenda à necessidade do cliente.

Umas das premissas do desenvolvimento tradicional é que o custo de uma alteração no sistema cresce exponencialmente ao longo do projeto, com isto tenta-se criar soluções genéricas preparando o sistema para futuras alterações.

Este tipo de pensamento dá margens para especulações e implementações que na maioria dos casos não terá utilidade para o cliente. Deve-se partir do princípio que o custo de uma alteração tende a crescer lentamente e se estabilizar ao longo do projeto, esta premissa é dita em função dos avanços nas linguagens e práticas de programação, novos ambientes e ferramentas de desenvolvimento, utilização de orientação a objetos no desenvolvimento e em conjunto com estes novos avanços existe o fruto do código ficar simples, legível e passível de alteração a qualquer momento.

A cada iteração, a equipe de desenvolvimento deve implementar novas características na arquitetura do sistema que sejam suficientes para comportar apenas as funcionalidades da iteração. Ou seja, mesmo que cartões de iterações futuras sejam conhecidos, a equipe não deve criar mecanismos para sustentar a construção futura dos mesmos.

Se a equipe investir muito cedo na criação de uma infra-estrutura para suportar cartões futuros, grande parte do esforço acabará se perdendo.

ANALOGIAS

Muitas vezes pessoas tentam explicar um assunto ou problema a outras pessoas por um período sem obter o êxito necessário na explicação dada. Simplesmente as outras pessoas não conseguem entender a mensagem que está se tentando repassar.

Ao criar comparações e analogias com o assunto em questão as pessoas passarão a entender deste assunto de uma forma muito mais rápida e possivelmente

não a esquecerão mais. Este tipo de artifício é chamado de *metáfora*, e deve ser utilizado com intensidade durante o projeto, uma vez que facilita a comunicação e fixação dos assuntos entre as partes.

Esta prática anda em conjunto com o ritmo sustentável, já que para o desenvolvedor criar boas metáforas exige criatividade e desenvolvimento de idéias, o que torna necessário uma boa condição física e bem estar por parte do desenvolvedor.

No início do projeto, a equipe e o cliente devem definir algumas metáforas para se referirem a aspectos chaves dos processos de aferição. Estas metáforas devem ser usadas nas conversas com o requerente e as classes do sistema devem ser nomeadas de acordo com elas.

O uso de metáforas é útil para facilitar a comunicação dos conceitos mais importantes do projeto. Para criar tais metáforas, a equipe deve imaginar como os processos modelados para o sistema seriam executados no dia-a-dia caso não existisse computador.

RITMO SUSTENTÁVEL

Um grande problema nos projetos de software é a falta de tempo para encerrar o mesmo, e uma das técnicas mais adotadas para compensar a falta de tempo é submeter os desenvolvedores (humanos) a trabalharem até mais tarde e muitas vezes sacrificarem seus finais de semana.

Nos primeiros momentos este tipo de atitude tem efeitos positivos, porém passados alguns dias o rendimento da equipe cai drasticamente, dando margens a erros pela falta de concentração no desenvolvimento, justamente pelo cansaço físico do desenvolvedor.

Deve-se proibir que os desenvolvedores trabalhem até mais tarde. Sugere-se um ritmo sustentável de 44 horas semanais, de segunda-feira a sexta-feira das 8:00h às 12:00h e das 14:00h às 18:00h, e aos sábados das 8:00h às 12:00h respeitando assim a individualidade e o físico de cada desenvolvedor. Abolindo-se completamente o trabalho além desse período. Desta forma a equipe estará sempre concentrada e muito menos propensa a pequenas falhas na implementação.

INTEGRAÇÃO CONTÍNUA

No desenvolvimento tradicional geralmente as equipes são organizadas de modo que uma parte (módulo) fique sob responsabilidade de um desenvolvedor,

cabendo a esta pessoa efetuar testes e codificação que dizem respeito à sua parte. Esta estratégia reduz a complexidade e as preocupações de um desenvolvedor.

Interfaces de integração são convencionadas para que futuramente estas partes possam se comunicar. Este tipo de desenvolvimento está propenso a sérios erros de integração, uma vez que os períodos em que as partes são integradas e testadas são extremamente longos.

Propõe-se uma integração contínua, se possível efetuada diversas vezes ao dia para que toda a equipe tenha conhecimento do código recém desenvolvido. Com esta prática o *feedback* sobre a alteração efetuada será retornado em um menor espaço de tempo.

O processo de integração contínua deve ser apoiado pela utilização de um repositório. Um repositório único para o código e uma base de dados única para todos trabalharem ao mesmo tempo.

De tempos em tempos, os desenvolvedores devem integrar o que desenvolveram com o que foi gerado pelos demais desenvolvedores, utilizando-se para isso, uma pasta comum para os *builds* gerados.

É possível que ocorram conflitos de integração que deverão ser corrigidos com rapidez porque todos os desenvolvedores se sentam próximos uns dos outros. Assim, os programadores devem chamar seus colegas para ajudar a solucionar os conflitos rapidamente, o que normalmente dará bons resultados.

CONCLUSÕES

O estágio atual, em que se encontra o mercado de desenvolvimento de softwares, exige que as empresas busquem novas fórmulas alternativas de competitividade para poder continuar a desenvolver suas atividades, cujas alternativas devem ser rápidas e eficazes.

Neste trabalho procurou-se mostrar a importância e a utilidade da aplicação de métodos de desenvolvimento de software e, ao mesmo tempo, demonstrar que a metodologia de desenvolvimento de software proposta é uma ferramenta eficaz e que proporcionará a melhoria da qualidade do software produzido. Além de reduzir custos e tornar os clientes mais satisfeitos.

A metodologia proposta é facilmente implantável pela empresa, podendo-se, através da monitoração de seus processos estabelecer uma constante melhoria no método. Em vista disso, o problema mencionado na Introdução deste trabalho pode ser reduzido sensivelmente.

O objetivo geral foi alcançado, pois a metodologia de desenvolvimento de software proposta, foi modelada baseando-se na estrutura laboral da empresa.

Quanto aos objetivos específicos, também obteve-se êxito, uma vez que foi identificado na fundamentação teórica, que o método de desenvolvimento de software XP é o que melhor se adapta ao modelo de fábrica de software da empresa estudada. Além disso foram feitos ajustes, baseados nas outras metodologias estudadas, para que o método proposto adeque-se à rotina de trabalho da empresa.

SUGESTÕES PARA TRABALHOS FUTUROS

A Metodologia proposta restringiu-se à teoria. Para que se tenha uma maior adequação à realidade da empresa, faz-se necessário o seu acompanhamento do uso.

REFERÊNCIAS

- AMBLER, Scott W. **Agile Modeling: Effective Practices for Extreming Programming and the Unified Process**. New York: Ed. John Wiley & Sons, Inc, 2002.
- ASTELS, David; MILLER, Granville; NOVAK, Miroslav. **Extreme Programming: Guia Prático**. Rio de Janeiro: Ed. Campus, 2002.
- BECK, Kent. **Extreme Programming Explained: Embrace change**. Reading, Massachusetts: Ed. Addison-Wesley, 2000.
- BOEHM, Barry W. **Software Engineering Economics**. Prentice Hall, 1981.
- BOOCH, Grady Booch. **Object-Oriented Analysis and Design with Applications**. 2ª edição. Addison Wesley, 1994.
- BROOKS, F. P. **The Mythical Man Month: Essays in Software Engineering**. Yourdon Press, Englewood Cliffs, New Jersey, 1982.
- COLEMAN, Derek et al. **Desenvolvimento orientado a objetos: o método fusion**. Rio de Janeiro: Campus, 1996.
- FLOWLER, Martin. **Refactoring: Improving the Design of Existing Code**. Ed. Addison-Wesley Longman, 1999.
- FUGGETTA, A. **Software process: a roadmap**. In: **The Future of Software Engineering**. A. Finkelstein (ed), 2000.
- GIBBS, W. **Software's Chronic Crisis**. Scientific American Magazine, setembro, 1994.
- HUMPHREY, W. S. **Managing the Software Process**. Addison Wesley Longman Inc, 1989.
- JEFFRIES, Ron; Anderson, Ann; Hendrickson. **Chet. Extreme Programming Installed**. Ed. Addison Wesley, 2001.
- Laitinen, M. & Fayad, M & Ward. R. **Software Engineering in the Small**. IEEE Software, setembro/outubro, 2000.
- LINDVALL, M. e RUS, I. **Process diversity in software development**. IEEE Software, v.17, n. 4, jul./ago. 2000.
- MCT. <<http://www.mct.gov.br>>. Acessado em maio de 2007.
- REED, K. **Software Engineering - a new millenium ?** IEEE Software, jul-ago 2000.
- Sanches, R. & Júnior, W. T. **Proposta de um Modelo de Processo de Planejamento de Projeto de Software para Melhoria de Gerenciamento de Projetos**. XII CITS - Conferência Internacional de Tecnologia de Software, junho, 2001.
- TELES, Vinícius Manhães. **Extreme Programming**. São Paulo: Ed. Novatec, 2003.
- Vigder, M. R. & Kark, A. W. **Software Cost Estimation and Control**. National Research Council Canada. Institute for Information Technology, 1994.
- Weber, K. C. **Qualidade e Produtividade em Software**. 3 ed. São Paulo: Makron Books do Brasil Ltda, 1999.

XISPE, Extreme Programming. Abril, 2007. < <http://www.xispe.com.br> >. Disponível em 28/05/2007.

XProgramming.com – an Extreme Programming Resource. Nov, 2001. Disponível em: <<http://www.xprogramming.com/xpmag/index.htm>>. Acesso em: 28 mai. 2007.

ANEXOS

ANEXO A - Quadro de Acompanhamento Diário

O quadro de acompanhamento diário é uma ferramenta de monitoramento do projeto e envolve os seguintes conceitos:

- História;
- Cartão;
- Estimativa;
- Dia ideal;
- Dia real;
- Velocidade e
- Tarefas extras.

Uma **história** é uma pequena funcionalidade que pode ser desenvolvida por um desenvolvedor em poucos dias. Um software é formado por um conjunto de histórias, as quais são escritas pelo cliente, com suas próprias palavras, em pequenos **cartões**.

Cada história deve ser estimada pela equipe de desenvolvimento, que registra a **estimativa** no canto superior esquerdo dos cartões (mais tarde, o realizado é colocado no campo superior direito). Desta forma, observando cada cartão, o cliente será capaz de identificar a história e seu respectivo custo na forma de tempo de desenvolvimento.

As estimativas utilizam como unidade o conceito de **dia ideal**. Se uma história é estimada em um dia ideal, isso significa que o desenvolvedor será capaz de desenvolver a história em um dia de trabalho, desde que não seja interrompido para executar outras atividades. Portanto, um dia ideal representa um dia de trabalho no qual o programador pode se dedicar integralmente apenas ao desenvolvimento de histórias, sem se preocupar em atender telefonemas, participar de reuniões, corrigir bugs etc.

Ao fazer uma estimativa, o desenvolvedor deve projetar um mundo ideal, no qual ele possa se abstrair de interrupções e dedicar-se plenamente ao desenvolvimento da funcionalidade. Portanto, a estimativa não leva em conta fatores externos, nem qualquer tipo de interrupção. O que se busca é o melhor caso. Entretanto, infelizmente o desenvolvedor vive em um **dia real**, no qual existem interrupções que afetam a quantidade de histórias produzidas.

Para lidar com isso, utiliza-se o conceito de **velocidade** da iteração. Um dia real é composto por horas dedicadas ao desenvolvimento de histórias e horas dedicadas a **tarefas extras**, as quais representam um overhead para o projeto. A velocidade indica a quantidade de horas efetivamente úteis, ou seja, que realmente levaram à implementação de novas histórias. Portanto, a velocidade procura indicar quantos dias ideais estiveram contidos dentro de uma determinada iteração.

Para compreender estes conceitos, pode-se utilizar um exemplo. Suponha-se uma iteração de duas semanas e uma equipe de dois desenvolvedores. Neste caso:

1 iteração = 2 semanas = 10 dias úteis.

1 desenvolvedor / dia trabalhando exclusivamente em histórias = **1 dia ideal**.

2 desenvolvedores / dia trabalhando exclusivamente em histórias = 2 dias ideais.

10 dias úteis x 2 dias ideais = 20 dias ideais.

A cada iteração deste projeto, se tudo correr de maneira perfeita, a equipe será capaz de implementar histórias que somem um máximo de 20 dias ideais. Este é o limite máximo da iteração e só é alcançado caso não existam tarefas extras ao longo da mesma.

Se em uma determinada iteração a equipe tiver consumido 4 dias para executar atividades extras, isso significará que a quantidade de dias ideais da iteração terá sido apenas 16, que representa o resultado de 20 (limite máximo) – 4 (tarefas extras). O valor 16, ou seja, a quantidade de dias ideais da iteração representa a sua velocidade.

As histórias são estimadas em número de dias ideais. Por sua vez, a velocidade indica a quantidade de dias ideais de uma iteração. Portanto, este valor é utilizado pelo Gerente para indicar a quantidade de histórias que serão alocadas a cada iteração. Ele poderá alocar histórias até que a soma de suas estimativas alcance a velocidade da iteração.

A velocidade não é um valor constante. Uma iteração pode ter uma quantidade maior ou menor de tarefas extras em relação à outra. Para que a equipe possa se planejar adequadamente, é necessário que seja capaz de fornecer a velocidade da iteração que se inicia com a maior precisão possível.

A equipe deve verificar quantos dias ideais foram obtidos na iteração anterior e assumir que esta quantidade se repetirá na iteração corrente. Entretanto, para fazer isso, é necessário que a equipe seja capaz de medir a velocidade de cada

iteração. Em outras palavras, é necessário que ela monitore o tempo dedicado às funcionalidades e o tempo dedicado às tarefas extras.

O **quadro de acompanhamento diário** é uma ferramenta que pode ser utilizada de maneira simples para monitorar os acontecimentos da iteração. Ele é dividido em duas partes: histórias e tarefas extras. A parte associada às histórias é utilizada para registrar o tempo gasto no desenvolvimento de cada história da iteração, enquanto a outra parte é usada para indicar o tempo alocado às atividades extras.

A tabela contém as seguintes colunas:

- Estimativa – utilizada apenas na parte associada às histórias. Armazena a quantidade de dias ideais estimados para cada história da iteração. As histórias são listadas nas linhas da tabela em ordem de prioridade.
- Dias da iteração – considerando-se uma iteração de duas semanas, haverá dez dias úteis de trabalho. Para cada dia, do primeiro ao décimo, cria-se uma coluna.
- Total – indica a quantidade total de tempo gasta com uma determinada história ou tarefa extra.

História	Estimativa	1	2	3	4	5	6	7	8	9	10	Total	Desvio
1. Listar voos entre duas cidades	3												
2. Ordenar por preço	1,75												
3. Ordenar por número de conexões	2												
4. Ordenar por número de escalas	1,75												
5. Buscar tarifa de um voo	2												
6. Buscar tarifa mais barata de um trecho	2,5												
7. Buscar 10 tarifas mais baratas de um trecho	2												
8. Marcar assento de um passageiro	1												
Total	17												
Tarefas Extras													
Total													

Ilustração 15Quadro de acompanhamento diário

Algumas funcionalidades consomem mais esforço que o previsto. O desvio entre a estimativa e o realizado também é armazenado na tabela, na extrema direita. Essa informação é útil para que a equipe estude as funcionalidades onde houve erros de estimativa e procure compreender o que os gerou, de modo a melhorar as estimativas futuras.

História	Estimativa	1	2	3	4	5	6	7	8	9	10	Total	Desvio
1. Listar vãos entre duas cidades	3	1	0,5	1								2,5	-0,5
2. Ordenar por preço	1,75	1	0,75									1,75	0
3. Ordenar por número de conexões	3		0,25	1	0,75	1	0,5					3,5	0,5
4. Ordenar por número de escalas	1,75				1	0,5						1,5	-0,25
5. Buscar tarifa de um vão	2					0,5	0,5	1				2	0
6. Buscar tarifa mais barata de um trecho	2,5								1	1		2	-0,5
7. Buscar 10 tarifas mais baratas de um trecho	2								0,75	1		1,75	-0,25
8. Marcar assento de um passageiro	1										1	1	0
Total	17	2	1,5	2	1,75	2	1	1	1,75	2	1	16	-1
Tarefas Extras													
Reunião de padronização do banco de dados			0,5									0,5	
Ajustes no link da internet				0,25								0,25	
Falta de luz							1	1				2	
Deployment em produção									0,25			0,25	
Correção do bug #25											1	1	
Total		0	0,5	0	0,25	0	1	1	0,25	0	1	4	

Ilustração 16-Quadro de acompanhamento diário preenchido

O quadro de acompanhamento diário deve ser desenhado a cada iteração em um quadro branco, de modo que seja permanentemente visível e acessível por todos os membros da equipe. A atualização deve ser feita de forma rápida durante a reunião diária. Cada desenvolvedor escreve no quadro a quantidade de tempo gasta no dia anterior com cada história ou atividade extra. Para isso, utiliza-se a regra de que o máximo de tempo alocado a uma história ou atividade extra em um dia é 1 dia ideal. Por sua vez, o mínimo é 0,25 dia ideal que equivale a duas horas de trabalho de um desenvolvedor.

Fazendo isso, o quadro não fornece precisão absoluta sobre o que foi feito na iteração, mas o processo de registro da informação é simplificado para diminuir ou eliminar resistências à sua utilização. O objetivo é facilitar para que se obtenha, ao menos, um bom retrato da iteração, mesmo que a precisão seja ligeiramente comprometida.

O preenchimento do quadro durante a reunião diária também é utilizado como forma de assegurar que a equipe mantenha a disciplina em conjunto. Quando um desenvolvedor possui uma ficha para preencher tudo o que faz durante o dia e o tempo alocado a cada atividade, por exemplo, é comum que ele se esqueça de preencher ou simplesmente se recuse a fazê-lo. Por outro lado, quando todos os desenvolvedores criam o hábito de preencherem o quadro, juntos, manter a disciplina torna-se mais fácil.

Ao final da iteração, a equipe pode registrar as informações do quadro para que possam ser referenciadas futuramente. Isso pode ser feito com o uso de uma planilha eletrônica ou um editor de texto, por exemplo. Assim, os membros do projeto são capazes, entre outras coisas, de verificar todas as histórias

implementadas no passado, o tempo efetivamente gasto em cada uma, os desvios em relação às estimativas e as tarefas extras realizadas.

As informações extraídas do quadro ao final da iteração atendem a vários propósitos. O mais básico é indicar a velocidade que será adotada na iteração seguinte. Entretanto, o quadro também é usado para fornecer subsídios à retrospectiva. Ela representa um processo estruturado de reflexão da equipe de desenvolvimento, executado com a participação do cliente, sempre que possível. A primeira etapa da retrospectiva consiste em enumerar e analisar os pontos positivos da iteração para que eles sejam lembrados pela equipe na iteração seguinte e sirvam de inspiração para manter as práticas que vem funcionando.

Em seguida, a equipe enumera e analisa os pontos da iteração que precisam melhorar. Todas as pessoas indicam os problemas que vivenciaram e as conseqüências dos mesmos. A equipe prioriza em conjunto os pontos a melhorar, de modo a indicar quais os dois ou três mais críticos (não mais que isso). Para cada um destes problemas, os participantes da retrospectiva propõem e discutem ações que possam resolvê-los na iteração seguinte.

As informações do quadro de acompanhamento diário são importantes durante a retrospectiva, pois geram pistas para identificar os problemas mais críticos, como por exemplo uma iteração com excesso de atividades extras, ou uma história que teve um desvio muito acentuado em relação ao estimado, ou um conjunto de histórias que acabou não sendo implementado. A retrospectiva é um mecanismo de melhoria contínua.

ANEXO B – Modelo de Cartão

Número	Data	Cliente	Estimativa	Realizado
1	3/12/2007	Funsep	5	

História: O sistema deve ter um cadastro de contribuintes.

Detalhamento: Utilizar os seguintes campos: Código, nome do contribuinte / Razão social, nome de fantasia, Tipo (Física ou Jurídica), CPF/CNPJ, Ocorrências, Receita <Selecionado de uma tabela>, endereço completo, telefone, situação, Sócios, Situação (Ativo, suspenso ou inativo).

A partir do cadastro do contribuinte, poder cadastrar Tipo de Receita.

Pode ter vários contribuintes cadastrados com mesmo nome/ razão social e nome de fantasia.

Se o contribuinte for pessoa física, o primeiro sócio é igual ao contribuinte.

Todos os campos são obrigatórios, com exceção de Ponto de Referência e telefone.

Sem limite máximo de sócios.

Sem limite máximo de telefones.

O campo "Código" deve ser gerado automaticamente e de forma sequencial

O campo "Ocorrências" deve ser numérico, começando por 1

O campo "Sócios" deve permitir o cadastro ilimitado de sócios.

O cadastro de sócios deve ter, pelo menos, os seguintes campos: Nome, CPF, endereço, telefones.

O cadastro de Receita deve ter os seguintes campos: código <informado manualmente>, tipo de receita <Selecionado de uma tabela>, descrição, valor em UFIR e valor em real.

O cadastro de tipo de receita deve ter os seguintes campos: código <informado manualmente > e descrição.

Deve-se importar o cadastro de contribuintes e o de receitas do sistema antigo.

Testes de aceitação

Testeld	Ação	Resultado
H1.T1	Digitar um CPF ou CNPJ inválido	Uma mensagem é exibida indicando que o CPF ou CNPJ é inválido
H1.T1	Confirmar cadastro com campo obrigatório sem preenchimento	Uma mensagem é exibida indicando que há um campo obrigatório não preenchido

[] – Campos de detalhamento

() – Campos de seleção

{ } – Campos de repetição

<> - Observação

ANEXO C – Quadro de Tarefas

Quadro de Tarefas				
Data	Cartão	Tarefa	Responsável	Situação
3/12/2007	1	Criação da base de dados	Richard	
3/12/2007	1	Importação do cadastro de receitas	Richard	
3/12/2007	1	Importação do cadastro de Contribuintes	Richard	
3/12/2007	1	Formulário de cadastro de receitas	Richard	
3/12/2007	1	Formulário de cadastro de sócios	Richard	
3/12/2007	1	Formulário de cadastro de contribuintes	Richard	

ANEXO C – Quadro de Acompanhamento Diário 2

Quadro de Acompanhamento Diário								
Tarefa	Estimativa	1	2	3	4	5	Total	Desvio
Criação da base de dados	2							
Importação do cadastro de receitas	1							
Importação do cadastro de Contribuintes	3							
Formulário de cadastro de sócios	2							
Formulário de cadastro de contribuintes	2							
Total								
Extras								
Total								