CENTRO UNIVERSITÁRIO DO SUL DE MINAS - UNIS/MG

SISTEMAS DE INFORMAÇÃO

EDYENE CELY AMARO OLIVEIRA

METODOLOGIA DE DESENVOLVIMENTO DE SISTEMAS

Belo Horizonte

2010

EDYENE CELY AMARO OLIVEIRA

METODOLOGIA DE DESENVOLVIMENTO DE SOFTWARE

Monografia apresentada ao curso de Sistemas de Informação do Centro Universitário do Sul de Minas UNIS/MG como pré-requisito para obtenção do grau de bacharel, sob orientação do Prof. Rafael Rodrigues de Souza.

Belo Horizonte

2010

EDYENE CELY AMARO OLIVEIRA

METODOLOGIA DE DESENVOLVIMENTO DE SOFTWARE

Monografia apresentada ao curso de Sistemas de Informação do Centro Universitário do Sul de Minas UNIS/MG como pré-requisito para obtenção do grau de bacharel, sob orientação do Prof. Rafael Rodrigues de Souza.

Aprovado em / /

_____

Prof. Ms.

_____

Prof. Ms.

_____

Prof. Ms.

OBS.:

Dedico este trabalho ao meu marido Antonio que sempre me apoiou e ajudou em todo este caminho, onde construí uma boa base de conhecimento.

AGRADECIMENTOS

Agradeço a Deus que me deu muitas oportunidades em minha vida para que fosse possível chegar ao ponto em que estou hoje.

Ao meu marido e filhos que sempre tiveram paciência em finais de semana e feriados de estudos e trabalhos, renegando muitas horas de lazer.

Aos meus professores que me deram apoio e tiveram paciência ao mesmo tempo em que cobravam mais empenho.

A alguns em especial que tive mais contato:

Prof. Tomás que me abriu as portas da faculdade através de um link esquecido aberto. Professor Wanderson, sempre incentivando e acreditando na educação à distancia e professora

Simone que me ensinou exaustivamente atividades de probabilidade.

Prof. Demétrio que me mostrou que era possível escrever um artigo, mesmo a distancia, sobretudo para alguém que nunca havia escrito nada do gênero.

Prof. Francisco que foi um herói em levar um trabalho de TCC até a conclusão a distancia.

E meu orientador prof. Rafael Rodrigues, onde através de sua disciplina defini minha área de atuação profissional, além de me orientar no TCC e estágio.

Obrigada aos professores, orientadores, tutores e colegas de classe, pois todos participaram direta e indiretamente do meu crescimento como profissional e como pessoa.

RESUMO

Este trabalho tem o objetivo de demonstrar uma metodologia desenvolvida com o melhor dos processos existentes no mercado atual. Trata-se da Metodologia Prática, um processo extraído de metodologias consideradas pesadas como o *RUP (Rational Unified Process)* e das metodologias ágeis *XP (eXtreme Programming) e SCRUM*.

O estudo demonstra o funcionamento dos três processos, a começar pelo RUP. É explicado o funcionamento, a sua base, disciplinas e papéis, pontos fortes e fracos. Este é um processo bem aceito no mercado atual inclusive sendo usado como base para certificações como MPS-BR e CMMI.

Logo em seguida é feita uma demonstração com explicações sobre as metodologias ágeis XP e SCRUM. É definido o funcionamento de cada uma e em que tipo de projeto esta abordagem será melhor aplicada, seus valores e objetivos.

E finalmente, o objeto deste trabalho que é a Metodologia Prática. É justificada sua criação, como poderá ser usada e em que tipo de equipe de TI se aplica. É demonstrado como a Metodologia Prática poderá ser usada aliando rapidez no desenvolvimento e qualidade no produto final.

A Metodologia Prática foi implementada em uma empresa de TI e aplicada em uma equipe com conhecimentos básicos de análise de sistemas. Este trabalho traz os resultados obtidos a partir de treinamentos realizados em 15 projetos onde foram todos desenvolvidos com total utilização desta metodologia.

Palavras-chave: Metodologia, Processo, RUP, XP, SCRUM.

ABSTRACT

This paper aims to demonstrate a methodology developed with the best of existing processes in the current market. This is the Practical Methodology, a process of extraction methodologies considered heavy like RUP (Rational Unified Process) and agile XP (eXtreme Programming) and SCRUM.

The study demonstrates the operation of the three cases, starting with the RUP. It explained how it works, it is based, disciplines and roles, strengths and weaknesses. This is a process well accepted in the market today including being used as a basis for certification as MPS-BR and CMMI.

Shortly afterwards is a demonstration that explains the SCRUM agile methodologies and XP. It defined the operation of each one, what kind of project approach is best applied, their values and goals.

And finally, the object of this work is the Methodology and Practice. It justified its creation, which may be used and what kind of staff to apply. It shows how the methodology can be used combining practice development speed and quality in the final product.

The Practical Methodology was implemented in an IT company invested in a team with basic knowledge of systems analysis and this paper presents the results obtained from training carried out in 15 projects which were all developed with full use of this methodology.

Key Words: Methodology, Process, RUP, XP, SCRUM

SUMÁRIO

1 INTRODUÇÃO

Desde os primórdios da programação de computadores que a busca por qualidade se tornou uma necessidade. A engenharia de software vem desempenhando um papel muito importante, pois visa sistematizar a produção, manutenção e recuperação de produtos de software de modo que ocorra dentro de prazos e custos estimados. Com base em seus objetivos, fica demonstrada a necessidade da adoção de um modelo sistêmico para padronizar e gerenciar os processos de desenvolvimento de software com o intuito de buscar qualidade nos processos e produtos de software. Mas o que são processos? O *Software Engineering Institute (SEI),* da *Carnegie Melon University* propõe o seguinte:

> "Um processo de software é uma seqüência de passos realizados para um dado propósito. Colocado de maneira simples, processo é aquilo que as pessoas fazem, usando procedimentos, métodos, ferramentas e equipamentos para transformar matéria prima (entradas) em produto (saída) que tenha valor para o cliente." (Paulk, 1995).

Este trabalho tem por objetivo demonstrar uma nova metodologia de desenvolvimento de software que visa praticidade aliando boas práticas à engenharia de software. No mercado atual existem várias metodologias, dentre elas o *Rational Unified Process (RUP),* trata-se de uma metodologia com disciplinas e papéis bem definidos. Pode ser usado tanto em projetos complexos e grandes, quanto em pequenos e simples, pois ele é bem versátil. Nos últimos anos devido às constantes mudanças no mundo como um todo, surgiram outros processos, estes são conhecidos como metodologias ágeis. Entre elas as mais conhecidas e usadas são o eXtreme Programming (XP) e o SCRUM. Ambas são mais leves, ou seja, não usam tantas documentações como o antigo modelo Cascata e o RUP. O objetivo das metodologias ágeis é a produção rápida, isso é conseguido fazendo pleno uso das iterações e interações. Isso quer dizer que a produção é feita por partes e com total acompanhamento do cliente.

No contexto da engenharia de software, uma metodologia é a adoção de processos de desenvolvimento de software e a partir de sua utilização, o processo deve possuir seqüência de passos ordenados. Além de ser constituída por técnicas e ferramentas, onde a técnica tem o objetivo de obter informação para ser usada com o processo e a ferramenta facilita e modela o sistema.

Assim uma metodologia visa dividir todo o contexto de desenvolvimento em várias fases onde cada uma possui outras atividades com o objetivo de construir ou manter um software de qualidade. E por trás deste software de qualidade tem toda uma seqüência de passos que foram seguidos de acordo com o processo adotado pela metodologia. Como existem vários processos para desenvolvimento de software, existem diversas formas de conduzir um projeto e seu ciclo de vida, podendo adotar desde o modelo Cascata ao Orientado a Objetos ou pode-se criar um processo personalizado. A Metodologia Prática é resultado do estudo dos três processos citados (RUP, XP e SCRUM), onde foi avaliado o que cada uma tem de melhor.

O objetivo da Metodologia Prática é aliar as disciplinas e definições de papéis presentes no RUP à versatilidade e agilidade dos processos ágeis – XP e SCRUM. Trata-se de uma metodologia com processos simples e claros, porém bem documentada. Contempla todas as fases do projeto, mas apenas com documentação necessária. Vindo do XP e SCRUM, a Metodologia Prática adotou métodos como prioridade ao cliente e reuniões freqüentes. O projeto é desenvolvido desde o início focando no cliente, tendo-o por perto, fazendo pleno uso de interações. A cada parte desenvolvida ou acrescentada, o cliente é chamado para dar seu feedback.

Segundo (PRESMAN, 2001), o modelo Cascata é o mais usado em todo o mercado, mas não o mais eficaz, pois além dos projetos não seguirem o fluxo linear, as mudanças de requisitos que ocorrem durante o desenvolvimento do projeto não são fáceis para serem adaptadas, pois altera toda a documentação já desenvolvida, implicando em retrabalho.

A Metodologia Prática reúne o que há de melhor em processos atuais do mercado, que são usados já por muito tempo e que visa contribuir para o desenvolvimento de software com rapidez e qualidade.

2 DESCRIÇÃO DAS METODOLOGIAS EXISTENTES

2.1 Histórico

Processo, sempre foi uma preocupação no desenvolvimento de software e até uma necessidade de padronizar e automatizar o processo. No início dos anos 1970 em plena crise do software que surgiu um modelo que segundo (PRESSMAN, 2001), trata-se do processo Cascata, este foi o mais usado em todo o mercado. O RUP surgiu em 1998, veio com o propósito de suprir as falhas do modelo clássico, ele tem as disciplinas e papeis bem definidos. O RUP foi criado pelos mesmos criadores da *Unified Modeling Language (UML)*, a saber, Grady Booch, Ivar Jacobson e James Rumbaugh e o objetivo de sua criação foi justamente de apoiar o RUP.

O XP e o SCRUM são metodologias ágeis e mais usadas no mercado atualmente, surgiram por volta de 2001. Nesta época dezessete especialistas em processos de desenvolvimento de software representando principalmente os métodos SCRUM (SCHWABER, 2002) e o XP (BECK, 1999), estabeleceram princípios comuns compartilhados por todos os métodos, sendo assim criada a Aliança Ágil. As metodologias ágeis tiveram seu embasamento no fato de serem utilizadas de maneira adequada em equipes pequenas e em requisitos vagos que esse modificam-se rapidamente.(BECK, 2004).

> A agilidade pode ser aplicada a qualquer processo de software. Entretanto para conseguir isso, é essencial que o processo seja projetado de modo que permita a equipe do projeto adaptar tarefas e aperfeiçoá-las, conduzir o planejamento para que se entenda a fluidez de uma abordagem de desenvolvimento ágil, eliminar tudo menos os produtos de trabalho mais essenciais e mantê-los simples, enfatizar uma estratégia de entrega incremental que forneça o software funcionando ao cliente o mais rápido possível para o tipo de produto e ambiente operacional. (PRESSMAN, 2001, p. 60)

2.2 XP - eXtreme Programming

2.2.1 Descrição

O *XP (eXtreme Programming)* é conhecido como Modelo Ágil, ou seja é uma metodologia leve para que equipes pequenas ou médias desenvolvam softwares em face a requisitos vagos ou que mudem constantemente. De acordo com seu autor, *Kent Beck*, *[Beck(2004)]*, o *XP* é focado na realização das tarefas que criem valor para o cliente. Sendo seu principal objetivo o desenvolvimento de software com qualidade por meio de um estilo de desenvolvimento focado em práticas de programação, comunicação clara e trabalho em equipe.

Visto que em desenvolvimento de software a única constante são as mudanças, o *XP* traz a abordagem de que elas são comuns de ocorrer, porém, é necessário saber lidar com elas.

> "Assim, um processo ágil de software deve ser adaptado incremental mente. Para realizar a adaptação incremental, uma equipe ágil requer o feedback do cliente (de modo que adaptações apropriadas possam ser feitas). Um catalisador efetivo para o feedback do cliente é um protótipo ou uma porção do sistema operacional. Assim, uma estratégia de desenvolvimento incremental deve ser instituída. Incrementos de softwares (protótipos, executáveis ou partes de um sistema operacional devem ser entregues em curtos períodos de tempo de modo que a adaptação acerte o passo com as modificações (imprevisibilidade). Essa abordagem interativa habilita o cliente a avaliar o incremento de software regularmente, fornecer o feedback necessário à equipe de software e influenciar as adaptações do processo feitas para acomodar o feedback". (PRESSMAN, 2001, pg. 61)

O XP é mais utilizado em equipes pequenas e médias, onde os requisitos são vagos e mudam rapidamente. O XP prega que é importante utilizar um código simples, isso implica em escrever o mínimo de classes, apenas o necessário. Sem contar que é melhor escrever o mínimo hoje e acrescentar se necessário, do que criar algo complicado que nunca venha a ser usado. As interações é o ponto forte do XP, em que a partir de pequenas entregas, a equipe terá respostas quanto ao software que está sendo produzido. Além de agregar valor ao produto, pois a qualquer momento, se o cliente parar a produção, ele já terá uma parte funcionando corretamente. O *XP* é um processo focado nas pessoas, com ele é definido como as pessoas vão interagir para gerar um produto de qualidade.

A Metodologia Prática adotou do XP os seguintes valores:

Comunicação – Um dos maiores problemas de um projeto ocorre por falta de comunicação. É necessário que haja uma comunicação clara e direta entre os membros da equipe.

Simplicidade – Ao invés de procurar soluções mirabolantes, importante focar em soluções simples a um dado problema. Este valor orienta a equipe a obter resposta sobre as ações realizadas através do *Feedback*.

Respeito – Os membros da equipe devem se importar uns com os outros e com as ações realizadas.

A Metodologia Prática adotou do XP os seguintes princípios:

Auto semelhança - A estrutura de uma solução deve ser utilizada em outros contextos, mesmo que em diferentes escalas.

Aperfeiçoamento - Deve-se sempre buscar a realização do melhor trabalho possível no presente.

Diversidade - As equipes devem ser formadas por pessoas com diferentes perfis. Os conflitos que possam surgir dessa escolha são compensados pelo benefício das múltiplas visões sobre um problema.

Oportunidade - Problemas devem ser vistos como oportunidades de mudança

Qualidade - Este princípio não deve ser visto como uma variável de controle negociável. A qualidade deve ser sempre buscada.

Pequenos passos - Ao dar grandes passos, leva-se muito tempo para realizá-los e, caso tenham sido dados na direção errada, é mais difícil voltar atrás. Agindo dessa maneira, é freqüente o temor da necessidade de mudanças. Pequenos passos são uma postura mais adequada em processos complexos. **Obs.** Neste ponto, vamos adotar a iteração e incrementação, pois à medida que o projeto vai sendo entregue, o cliente vai percebendo a forma que ele está tomando e ficará mais fácil gerenciar alguma divergência.

Sentar junto fisicamente - O ambiente de trabalho da equipe deve ser compartilhado. Há sim a necessidade de espaços privados, mas a equipe deve trabalhar junta, fisicamente, a maior parte possível do tempo.

Time completo - As equipes de XP devem ter pessoas com diversas habilidades diferentes, que atendam todas as necessidades de um projeto

Área de trabalho informativa - Uma pessoa que entre no espaço de uma equipe deve poder, num curto espaço de tempo, ter noção do estado em que se encontra o projeto em desenvolvimento. O ambiente deve propiciar também espaços coletivos para a programação e espaços individuais para a privacidade. Nas paredes, é interessante manter gráficos grandes, bem como outras informações pertinentes sobre o estado do projeto e da equipe.

Design incremental - O design de um sistema deve ser trabalhado diariamente, levando-se em consideração o melhor a ser feito naquele momento.

2.2.2 Críticas ao XP

O XP tem alguns pontos fracos, a começar pela falta de análise de riscos. Não existe a preocupação formal em fazer análise e o planejamento do futuro sistema e como os riscos acontecem normalmente em projetos de desenvolvimento de software, este se torna um ponto negativo. Também o fato de tratar a comunicação, caso a metodologia seja usada em grandes equipes e estarem muitas vezes separados geograficamente.

2.3 Scrum

2.3.1 Descrição

SCRUM é um método iterativo, incremental e ágil para o desenvolvimento de software. Ele é colaborativo na sua essência e é ideal para pequenos e complexos projetos de desenvolvimento de software onde os requisitos mudam constantemente. É escalável, pois suporta grandes projetos e é compatível com *CMM3 e ISO9001*. É extremamente simples, mas muito resistente.

Do *Scrum* esta Metodologia prática vai buscar boas práticas para ajudar a conduzir o processo de desenvolvimento de software.

2.3.2 Comunicação

Segundo o Scrum a equipe precisa ter uma comunicação efetiva e direta. O Scrum incentiva o uso do quadro branco para clarear as dúvidas.

2.3.3 Planejamento Contínuo

É importante construir o todo aos poucos, refinando os requisitos, clareando o entendimento sobre eles e corrigindo o rumo sempre que uma mudança provocar alterações significativas. O projeto de software deve ser um processo criativo, não linear, por isso planos menos detalhados e feitos com freqüência maior se mostram mais adequados. É importante entender que é impossível planejar e manter estática as fases do projeto até o fim, pois as mudanças são necessárias e podem impactar no projeto. É necessário saber gerenciar corretamente as mudanças. A fim de contribuir com o planejamento contínuo, é importante identificar desde o início os stakeholders que são os envolvidos no projeto e mantê-los acompanhando o projeto, eles precisam comprometer-se com o projeto também. Assim, através de iterações curtas, será garantido ritmo a todos os envolvidos e levam ao feedback real e imediato dado pelo cliente.

2.3.4 Gerenciamento de riscos

Em metodologias ágeis, a identificação e gerência dos riscos são feitos através de reuniões diárias e a cada iteração, o controle da qualidade dos trabalhos é avaliado. Esta tarefa pode ser executada através de testes, revisão por pares, inspeção contínua e acompanhamento pelo cliente, que seguem o mesmo processo.

É importante lembrar que o sucesso de um projeto de software não é somente medido se comparando projeções iniciais de prazo, custo e escopo e sim através de entrega de software de valor ao cliente. Estes três pilares – prazo, custo e escopo - devem ter seus pesos definidos

O interessante deste tipo de abordagem é que não se tem a procura por um culpado. Todos estão imbuídos na busca da melhor solução para a empresa, seja o analista,

desenvolvedor, gerente de projeto ou o cliente. E cada um tem o seu papel claramente determinado frente ao processo de desenvolvimento. De acordo com o Scrum é sempre bom ter o cliente por perto e fazê-lo um participante ativo. É importante que ele entenda suas responsabilidades e sua grande parcela de contribuição para o sucesso do projeto.

2.3.5 Críticas ao SCRUM

O Scrum precisa ser seguido à risca caso contrário, não vai ser Scrum. Precisa respeitar a quantidade de membros das equipes e fazer uso dos artefatos como o quadro branco. O fato da comunicação também precisa ser bem resolvido, para não haver dispersão.

2.4 Processo RUP

2.4.1 Descrição

O *RUP* é um processo iterativo e incremental resolvendo o problema de entregar um produto e só então o cliente perceber que não é o que ele esperava. Segundo *Kruchten[4]* uma abordagem iterativa consiste em construir um pouco, implementar um pouco, projetar um pouco, validar e então assumir mais requisitos, construir um pouco mais, implementar um pouco mais e assim por diante, até que tenha acabado.

O *RUP* deve ser implementado para projetos grandes e complexos, podendo ser adaptado para projetos pequenos também.

As vantagens do desenvolvimento iterativo são:

- Os riscos são reduzidos mais cedo.
- Os requisitos variáveis serão melhor gerenciáveis.
- A capacidade de reutilização aumentada.
- Os diversos membros da equipe de desenvolvimento trabalham de forma uniforme.
- Os testes são utilizados desde o início do projeto.
- As mudanças são melhores acomodadas.

O *RUP* se preocupa em definir quais diagramas serão gerados, e quais workflows os consumirão. Sendo, portanto baseado em processos e documentos.

Faz pleno uso de diagramas *UML (Unified Modeling Language*), sobretudo de Use Case. O *RUP* é praticamente guiado por Caso de Uso, este é o principal artefato desta metodologia.

A UML independe de processo, ela é uma linguagem para modelagem de sistemas orientados a objetos. Ela indica como criar e ler modelos bem formados, mas não quais deverão ser criados. Neste momento que entra a metodologia, o processo que a empresa ou desenvolvedor de software irá fazer uso. Os criadores do Modelo Unificado são os mesmos criadores da UML de forma que se este processo oferece as melhores práticas de desenvolvimento de software atual.

2.4.2 Características

O RUP possui características próprias como ser orientado a objetos, guiado por caso de uso, centrado na arquitetura, incremental e interativo. Essas características o torna único e um dos melhores processos a ser implementado.

O fato do RUP ter essas três características, faz dele um processo realmente forte com disciplinas bem definidas. Porém essas características não podem ser alteradas, pois faria com que o RUP perdesse seu valor. O forte dele são as três características e não podem nem serem alteradas nem retiradas, mesmo que seja uma delas.

2.4.3 RUP - Orientado por Caso de Uso

O RUP é um processo orientado por caso de uso, isso quer dizer que desde o início, ele é concebido com a visão de atender especificamente às necessidades de cada usuário. Pois o caso de uso é um cenário onde são definidos os limites do sistema, suas funcionalidades e usuários de cada uma delas.

Um caso de uso não é ligado apenas aos requisitos, mas também ao projeto, implementação e testes do mesmo. Todo o processo é guiado por caso de uso, ou seja, segue um fluxo de trabalho desde o projeto, culminando no sistema em produção.

2.4.4 RUP - Centrado na Arquitetura

Ser centrado na arquitetura quer dizer que o sistema é projetado para evoluir, não só no projeto presente, mas também nas versões futuras. O RUP, como já foi dito, é guiado por caso de uso e este fator é o cerne para criar uma arquitetura correta.

Os casos de uso estão ligados às funcionalidades do sistema e a arquitetura está ligada a como eles estão formados. A arquitetura deve ser projetada utilizando alguns casos de usos, cerca de 5% a 10% que são os casos de uso chaves do sistema. Os casos de uso devem estar de acordo com a arquitetura, os dois se completam, os primeiros geram as funcionalidades e a arquitetura o suporte para eles se acomodarem e terem condições de realizarem todos os requisitos necessários ao sistema.

2.4.5 RUP – Interativo e Incremental

Estas duas disciplinas do RUP o tornam muito interessante, pois no antigo modelo Cascata, o sistema era projetado e implementado de forma linear. Desta forma, o cliente tinha acesso a algo concreto apenas quando lhe era entregue o sistema já em produção. No máximo ele conseguia visualizar alguma parte do sistema na fase de testes. Isso gerava atritos, pois o mundo muda a cada instante e tais mudanças são necessárias. De forma que muitas funcionalidades ficavam sem utilidade ou obsoletas.

De acordo com o processo do RUP, o projeto deve ser realizado de forma incremental. Isso quer dizer que o projeto é apresentado ao cliente em partes para que ele vá acompanhado o que está sendo feito. Logo no início são definidos os módulos e à medida que a implementação vai ocorrendo, o projeto é entregue ao cliente. Desta forma obtém feedback por parte dele e reduz os riscos, pois o módulo entregue estará funcionando.

Trabalhar desta forma também tem outro fator importante que é o de gerar valor para o cliente. Além de entregar o produto ou parte dele mais rápido, o cliente receberá um produto testado e aprovado. Se ele desejar parar a produção por algum motivo, ele terá um produto de valor às mãos, em funcionamento. Pelo menos já foi entregue alguma parte para ele que poderá ser usada. Se fizer de outras formas, isso não ocorrerá, pois o sistema será entregue apenas no final da codificação e testes.

2.4.6 Críticas ao RUP

O *RUP (Rational Unified Process*) tem disciplinas pesadas e, sobretudo papeis bem definidos. Não tem como aplicar esta metodologia sem cumprir todas as fases do processo. O que muitas vezes pode ser difícil devido às constantes mudanças. E o ponto negativo maior para a aplicação do *RUP* é que ele é melhor aplicado em grandes equipes. Os papéis devem ser definidos, em suma é preciso ter pessoal qualificado em postos adequados e as fases devem ser cumpridas à risca. Isso se torna um ponto negativo, pois não é qualquer equipe e projeto que poderá aplicá-lo. E em relação às mudanças ele pode ser pesado e não adequado ao desenvolvimento e acompanhamento de todo tipo de projeto.

3 JUSTIFICATIVA DA METODOLOGIA PRÁTICA

É cada vez mais necessário que as empresas produzam software de qualidade, com o menor custo e dentro do prazo estipulado. Ter um processo bem definido significa ter maior possibilidade para se obter, entender e validar as necessidades e expectativas do cliente para, posteriormente, documentá-las. O principal objetivo deste processo é concluir, com êxito, um acordo entre quem solicita e quem desenvolve, estabelecendo clara e rigorosamente o que deverá ser produzido.

Não utilizar processo ou uma metodologia é equivalente a pensar "não há tempo para descobrir o que se deve construir, devemos começar a construir agora" (HEU,2001).

A Metodologia Prática vem em busca de melhoria na produção de software, extraindo o melhor dos três processos existentes no mercado de TI - *XP (eXtreme Programming), Scrum e RUP (Rational Unified Modeling).*

4 METODOLOGIA PRÁTICA

Para desenvolver o software, esta metodologia aplica processos que gera artefatos que serão usados nas fases de desenvolvimento e testes. Documentos e artefatos são pré-requisitos para um software de qualidade.

De acordo com o *XP*, *"*a maior prioridade é satisfazer ao cliente, lançando versões em pouco tempo e continuamente." E o *RUP* também tem o mesmo princípio, de forma que a produção de software se dará de forma iterativa e incremental. Através deste modelo, vai ser possível mostrar valor para o cliente desde o início, pois processo não é visto como valor para ele. A partir do momento que começa a entrega dos módulos, ele vai percebendo o valor investido, além de servir de *feedback* para a equipe.

O processo de planejamento da Metodologia Prática é constituído por várias etapas que devem ser seguidas, lembrando que o bom senso deve ser seguido sempre. E o propósito dela não é de "engessar" a produção de software, mas sim padronizar, buscando qualidade. Estão sendo definidos abaixo artefatos que devem ser gerados na produção do software, mas podem ser usados menos ou mais diagramas, resultando em outros artefatos. É importante usar de bom senso, responsabilidade e experiência para definir a necessidade de alteração na seqüência abaixo.

4.1 Processos

4.1.1 Levantamento de Requisitos:

- Após o primeiro contato com o cliente, o líder do sistema deverá criar o documento de requisito.
- Assim que o documento estiver pronto, o líder deve retornar para o cliente para validação.
- Após aprovação do cliente, o líder deve então agendar uma reunião com os desenvolvedores que estarão envolvidos no projeto para repassar todo o levantamento do sistema.

- Caso haja alguma dúvida por parte de alguém, o líder deve marcar outra reunião com o cliente para sanar as dúvidas. O documento de requisito não pode deixar nenhuma margem de dúvidas para ambos os lados, equipe do projeto e cliente.

4.1.2 Caso de Uso

- Após ter validado com o cliente e estar bem claro com os membros da equipe, o documento de requisitos será utilizado para produzir outro artefato, o Use Case. O líder do projeto ou o analista desenhará todas as funcionalidades do projeto que foram levantados e estão descritos no documento de requisitos.
- Assim que os documentos de caso de uso estiverem prontos ou à medida que forem ficando prontos, o analista deverá validar junto ao cliente. Lembrando que não é necessário ser todos os casos de uso, mas aqueles casos críticos, que precisam de exatidão, para que fiquem bem claros frente ao cliente. Por se tratar de modelo (desenho), alguma regra pode ser melhor visualizada e entendida.

4.1.3 Diagrama de Classes

- Quando os casos de uso estiverem prontos, ou de preferência à medida que forem ficando prontos (no caso de muitos), é importante começar a desenhar os diagramas de classes. Apenas é importante lembrar que o projeto precisa ser visto como um todo, conforme for a criticidade é interessante terminar os casos de uso e depois criar os diagramas de classes.
- No caso de diagrama de classe, não é importante mostrar ao cliente, visto ser um artefato interno, é usado para a equipe de desenvolvedores.

4.1.4 Modelo de Dados

O modelo de dados é usado após ter os diagramas de classes prontos, pois geralmente cada classe será uma tabela do banco de dados. A diferença é que no modelo de dados, existe a cardinalidade e chaves primárias e estrangeiras.

4.2 A Fase de Planejamento

O planejamento do software é sem dúvida a parte mais importante da criação de um sistema. Estudos mostram que a grande soma de investimentos na produção de software está concentrada em testes e manutenção, ou seja, gasta-se mais para consertar o software do que para produzi-lo. Segundo o *Standish Group*, órgão pesquisador americano, 85% dos erros são introduzidos na etapa de planejamento, sobretudo na análise de requisitos e projeto. A etapa de planejamento desde o primeiro contato com o cliente até a finalização do projeto deve ser feito com cuidado, atenção e muita responsabilidade.

É importante seguir as etapas que compõe o planejamento e que a Metodologia Prática adotou, porém em tudo é preciso bom senso. Afinal o objetivo da metodologia é ajudar na construção de software de qualidade, mas ela por si não é garantia de sucesso. É importante lembrar que os problemas comuns na elicitação de requisitos são:

- Problemas de escopo
  - ✓ O limite do sistema é mal definido ou detalhes técnicos desnecessários confundem os objetivos globais
- Problemas de entendimento
  - ✓ O analista precisa entender (e muito bem) o domínio do negócio. Caso o domínio seja completamente desconhecido do analista, é necessário gastar mais tempo nesta fase, até que todos os pontos estejam completamente entendidos.
- Problemas de volatilidade
  - ✓ Gerenciar as mudanças é muito importante, pois elas sempre existirão. Não tem como trabalhar cada fase do processo até o final sem que seja necessário alterar alguma informação. E tais mudanças influenciam todas as etapas, sendo necessário alterar desde o documento de requisito ao código.

4.3 Entrevista

A entrevista é uma das técnicas mais comuns para obter informações, podendo ser classificadas como estruturadas ou não estruturadas. Uma entrevista estruturada requer um prévio conhecimento sobre o contexto onde se aplica a entrevista. A entrevista não-estruturada é aplicada quando se inicia o contato com o universo de informações e serve para garimpar informações iniciais. O entrevistador deixa o entrevistado falar, mas deve tomar o cuidado de não perder o foco no tópico em questão. Segundo (LEITE, 2006), o papel do entrevistador na entrevista não estruturada é um papel de aprendizado. Muitas vezes a equipe pode usar perguntas preparadas por outros atores, mas o ideal é que os entrevistadores sejam aqueles que elaborem as perguntas.

4.3.1 Como proceder na entrevista

Entrevista é uma disciplina própria do Jornalismo, mas em Sistemas de Informação é muito útil para elicitar um requisito. As entrevistas se tornam uma técnica e muito importante. Apenas alguns cuidados devem ser tomados como:

- ✓ Marcar a entrevista e avisar ao responsável sobre o assunto a ser tratado e não se atrasar neste dia, procurar chegar na hora combinada, se possível com certa antecedência;
- ✓ Procurar ser bem humorado no diálogo, porém sem exageros que destoem;
- ✓ Agir com segurança e naturalidade, mostrando que sabe o que quer;
- ✓ Ajudar o entrevistado, se necessário, a expor as suas opiniões. Conduzir a entrevista;
- ✓ Não cortar as respostas. Esperar que cada uma delas termine para formular a próxima pergunta;
- ✓ Não se devem misturar várias perguntas ao mesmo tempo;
- ✓ Você deve acompanhar com total atenção as respostas do entrevistado;

- ✓ Faça as perguntas no mesmo nível de quem responde: às vezes trata-se de pessoa humilde que tem informações sobre determinado fato, mas se ficar amedrontada não agirá da mesma forma que antes.

- ✓ As perguntas podem ser:

  a) de esclarecimento.
  Exemplo: Quantos usuários acessarão o sistema?

  b) de análise
  Exemplo: Visto já existir na empresa um sistema, quais motivos o levaram a ver a necessidade de outro sistema?

  c) de ação
  Exemplo: O que você espera desta funcionalidade específica?

4.4 Levantamento de Requisitos

O levantamento de requisitos é uma das etapas mais importantes na produção do software, pois conforme já foi falado anteriormente, é nessa fase que mais erros são introduzidos. De acordo com pesquisa realizada pelo *Standish Group*, os requisitos aparecem em primeiro lugar como problema onde requisito incompleto é responsável por 13,1% da causa dos fracassos. Sendo seguido por falta de envolvimento do usuário com 12,4%, mudança de requisitos e especificações com 8,7% e falta de planejamento com 8,1%. Os requisitos realmente são muito importantes além de que se trata de um trabalho social, é necessário que o analista seja responsável ao lidar com outras pessoas.

Entre os objetivos do levantamento de requisitos estão:

- ✓ Estabelecer uma base de concordância entre o cliente e o fornecedor sobre o que o sistema fará.
- ✓ Fornecer uma referência para a validação do produto final.
- ✓ Reduzir o custo do desenvolvimento.

- ✓ Manter planos, artefatos e atividades de software consistentes com os requisitos aprovados.

4.4.1 Requisitos Funcionais

São requisitos diretamente ligados a comportamento do software. Descrevem as funções que o software deve executar. Um requisito funcional descreve uma interação entre o sistema e seu ambiente

Ao escrever um requisito procurar escrever de forma clara as funcionalidades do sistema e as pessoas que vão interagir com ele, pois as funcionalidades são realizadas para alguém. Ao fazer isso durante a escrita do documento de requisitos, ficará bem mais fácil identificar os casos de uso e os atores ao escrever os diagramas de caso de uso.

Exemplos:

Ex.1 O software deve permitir que o atendente efetue cadastro de clientes

Ex.2 O software deve permitir que o caixa efetue o registro de itens vendidos

Ex.3 O software deve permitir que o administrador gere o relatório de vendas por mês.

Os itens grifados são os atores e as informações em vermelho são as funcionalidades.

4.4.2 Requisitos não Funcionais

São requisitos que expressam condições que o software deve atender ou qualidades específicas que o software deve ter. Em vez de informar o que o sistema fará, os requisitos não funcionais colocam restrições no sistema.

Os requisitos não funcionais definem propriedades e restrições do sistema, porém eles podem ser mais críticos que requisitos funcionais. A partir do momento que não satisfaz, se torna um sistema inútil.

Um fato muito importante é que os requisitos não funcionais devem ser mensuráveis. Deve-se associar forma de medida/referência a cada requisito não funcional elicitado.

Exemplo: velocidade, tamanho, facilidade de uso, confiabilidade, robustez, portabilidade.

Nunca usar palavras não verificáveis como: amigável, portável, pequeno, flexível – todas elas geram ambigüidade. Deixam margem para discussão com cliente no futuro, pois requisitos como rápido não é mensurável.

4.5 Caso de Uso

Caso de uso é um dos diagramas mais importantes da *UML*, certos processos como o *RUP* dão tanta importância que é praticamente guiado por caso de uso. Um caso de uso descreve um conjunto particular de funcionalidade do sistema, modelando o diálogo que uma entidade externa chamada de ator realiza com o sistema.

Os objetivos do caso de uso são:

- Identificar os objetos na fase de desenvolvimento de planos de testes e documentação;
- Aprimorar a comunicação entre os projetistas da aplicação e os clientes
- Utilizado para representar as funcionalidades do sistema
- Representa o que o sistema faz (não como)

Este é um artefato que esta metodologia incentiva fortemente a usar, pois se trata de um dos diagramas que mais esclarece dúvidas referentes ao domínio do problema. E um dos únicos que é entendido tanto pelo cliente quanto pela equipe. Ele poderá ser usado também pela área de qualidade nos testes, pois ele detalha as funcionalidades e atores.

4.6 Diagrama de Classes

Classe é uma descrição de um conjunto de objetos que compartilham os mesmos atributos, operações, relacionamentos e semântica. Elas podem ser concretas ou abstratas. Uma classe é uma cópia da tabela do banco de dados com acréscimo de operações. Uma classe é um dos principais diagramas estruturais da UML. Um diagrama de classes lista todos os conceitos do domínio que serão implementados no sistema e define a estrutura do sistema a desenvolver. Os diagramas de classes não surgem do nada, eles são conseqüência do prévio levantamento de requisitos e definição de casos de uso.

Os diagramas de classes ilustram os atributos e operações de uma classe e as restrições como que os objetos podem ser conectados, descrevem também os tipos de objetos no sistema e os relacionamentos entre estes objetos que podem ser: associações e abstrações. Para representar a visibilidade dos atributos e operações em uma classe utilizam-se as seguintes marcas e significados:

+ público – visível em qualquer parte do sistema em qualquer classe

 # protegido – visível apenas para a classe que o chamou – as descendentes

-privado – visível somente dentro da classe

4.7 Modelo de Dados

O modelo de dados é o artefato mais importante da modelagem de um sistema. Pois se trata do modelo do banco de dados, onde serão armazenadas todas as informações do cliente. Seguindo as etapas anteriores, ao chegar nesta fase o modelo estará bem claro na mente dos envolvidos no projeto. Além de que terão sido criados todos os artefatos necessários para geração do modelo de forma correta.

Esta metodologia usará um sistema criado pela própria empresa que é o gerador de códigos. Visto que este programa tem padrão próprio, a metodologia ajustou-se para atendê-

lo, pois se trata de um mecanismo já testado e estável. Foi usado em variados projetos passando por desktop e web, simples e complexos, pequenos e gigantes. O código gerado está em sincronia com o Framework próprio, que também será usado em todas as aplicações criadas na empresa.

Visando trabalhar em total conformidade com o gerador de códigos e com o Framework da empresa, a metodologia adotará o padrão do framework. (vide o item padronização).

A ferramenta para trabalhar com banco de dados foi o *Visio Enterprise Architets*, que é uma ferramenta Microsoft e robusta. Tem todas as funcionalidades para geração do modelo de dados e geração de scripts. Ao usá-la, o analista criará o artefato (documento) e criará o script para a geração do banco, de forma que ao passar para o programador a especificação funcional, já estará tudo pronto para começar a geração de códigos. E para manutenção, o *Visio Enterprise Architets* faz engenharia reversa de bancos já existentes, bastando informar o caminho ou endereço do servidor onde o banco de dados está instalado.

4.8 Prototipação

Segundo o dicionário, protótipo significa***: "primeiro tipo ou exemplar, original, modelo"***. Essa definição nos desperta para a sua funcionalidade, que nos faz pensar e decidir qual é a melhor maneira de utilizá-lo. O protótipo pode ser interpretado como uma maquete, que contém os conceitos do que se pretende obter do produto final. Pode ser considerado também como a primeira versão de um produto que, mesmo com as funcionalidades incompletas, permite ser criticado e aperfeiçoado a fim de se obter qualidade no que está sendo produzido.

O objetivo da prototipação é validar os requisitos, abordar questões de interface e avaliar a visibilidade e complexidade do sistema. Visto que durante a prototipação, usuários e clientes ficam em constante interação, facilita as alterações nas funcionalidades do sistema bem como modificações de requisitos. Neste período, o esforço para qualquer intervenção, tanto na documentação quanto na implementação da aplicação, é muito menor, reduzindo mão-de-obra e, conseqüentemente, custos de produção. Com prototipação, desde cedo os

requisitos do sistema se tornam bem perceptíveis e facilita o seu esboço. Além de que o cliente já acompanha sabendo como ficará o sistema, ele entende muito mais protótipo do que documentos.

Assim que o documento de requisito estiver pronto ou caso de uso, já deve ser iniciada a prototipação.

A Metodologia Prática adotou um software comercial do mercado para construir os protótipos. A opção foi o software *Axure*, pois se trata de uma verdadeira revolução em prototipação de sistemas. Ele é relativamente fácil para aprender, em pouco tempo qualquer profissional da área de TI consegue aprender os primeiros passos. Sem contar que o fabricante mantém um blog e tutoriais para auxiliar seus clientes.

As ferramentas de prototipação são de grande valor atualmente, sobretudo as que trazem a experiência de navegação. Este que é o maior ganho, pois por ser navegável, ele simula realmente o que acontece a cada clique. Antes em outras abordagens como o uso de ferramentas como o PowerPoint, era necessário criar inúmeros slides e o resultado não era satisfatório como é hoje. Após criar todos eles e obter aprovação do cliente, era necessário passar por um design e interface. E somente após esta etapa é que era possível ter um protótipo navegável. Com ferramentas tipo *Axure*, esta experiência é obtida já na primeira apresentação, simula realmente o que acontece a cada clique. Além do dimensionamento do projeto, no *Axure*, por exemplo, é possível acessar com facilidade a lista de páginas e até mesmo gerar um mapa totalmente navegável a partir dela.

Os protótipos são de grande ajuda, sobretudo no sentido de gerar valor para o usuário. Pois ele não entende e logo não vai ver o investimento dele em vários documentos gerados após vários dias ou meses de trabalho da equipe de TI. Enquanto que mostrando um projeto mesmo que em HTML, ele vai conseguir ver algo concreto e conseqüentemente ficará satisfeito.

4.9 Padronização

A padronização é fundamental no controle de qualidade de qualquer empresa e visto que as atividades de codificação são responsáveis pelo maior esforço do desenvolvimento de software, quanto mais fácil for o entendimento do código da aplicação, mais produtiva será a equipe de desenvolvimento. Freqüentemente as pessoas que escrevem o código não são as mesmas que o mantém e, quando são, geram uma dependência com o código desenvolvido que dificilmente é dissolvida. Um padrão de codificação visa minimizar esses problemas, pois estabelece regras definindo como o código deve ser escrito para favorecer a impessoalidade do artefato sem modificar a funcionalidade da aplicação.

4.10 Classes

4.10.1 Nomenclatura

A Metodologia Prática foi desenvolvida para um processo orientado a objetos, desde a concepção do software, passando pela escolha da linguagem e implementação. Portanto, foi definido que o nome da classe deve ser o mesmo nome da tabela no banco de dados, pois uma classe é a representação desta tabela e o framework ao gerar o código para ela vai seguir este padrão.

Se a tabela no banco de dados for fruto de duas tabelas, no banco ela estará criada com o nome da primeira seguida de "_" (underscore), e, portanto, o nome da classe deverá ser o mesmo.

4.10.2 Declaração

Se for necessário criar uma classe sem o uso de um gerador de código, obedecer as seguintes normas:

- Após a declaração do nome da classe e seus supertipos, são declaradas as constantes, variáveis de classe (usam-se os termos “variáveis de classe” e “métodos de classe” com o mesmo significado de “variáveis estáticas” e “métodos estáticos”, respectivamente), propriedades de classe, variáveis de instância, propriedades de instância, Construtores estáticos de classe, construtores, finalizador, métodos de classe e métodos de instância, nesta seqüência.
- Os métodos são agrupados por funcionalidade e não pela forma de acesso ou sua condição de estático ou de instância.

4.10.3 Documentação

A codificação deve ser documentada, para isso as linguagens de programação tem vários recursos para este fim. No caso do .NET tanto em páginas quanto classes, usa-se a opção de comentário Sumary.

A linha imediatamente anterior à definição da classe deve conter um comentário descrevendo seu propósito. Esse texto deve estar dentro de uma tag <summary>.

Podem ser acrescentados lembretes sobre possíveis melhoramentos e defeitos existentes na classe. Esses textos devem estar dentro da tag <remarks>. Esta tag vem depois da tag <summary>.

4.11 Namespaces

4.11.1 Nomenclatura

Os identificadores que são as palavras que compõem o nome de um namespace – são separados por pontos e são iniciados com letras maiúsculas e não devem conter caracteres especiais, como underscores, ou caracteres específicos de uma língua.

Os namespaces são nomeados de acordo com os conceitos que agrupam.

Esta metodologia usará como definição de namespaces o nome da empresa seguido do nome da aplicação.

4.11.2 Declaração

Após o comentário de início do arquivo, são listados os namespaces importados (utilizando a cláusula using) e declarado o namespace (utilizando a cláusula namespace) a que a classe a ser definida pertence.

- Só deve existir uma declaração de namespace por arquivo.
- Os namespace pertencentes ao *.NET Framework Class Library* são listados primeiro. Depois são listados os namespaces da aplicação e de outras *APIs* utilizadas.

4.12 Interfaces

4.12.1 Nomenclatura

O nome de uma interface é um adjetivo ou substantivo, e segue as mesmas regras para nomenclatura de classes.

4.12.2 Documentação

A documentação segue as mesmas regras das classes.

4.13 Enumerações, Delegates, Estruturas, Constantes, Variáveis De Classe, De Instância E Locais, Parâmetros E Propriedades De Classe E De Instancia.

4.13.1 Nomenclatura

Todas as nomenclaturas devem respeitar as regras abaixo:

- Os nomes não são abreviados, exceto nos casos em que sua abreviação seja tão ou mais sugestiva que o nome completo.
- Quando apropriado, devem ser utilizados nomes complementares como mínimo/Maximo, começo/fim e aberto/fechado.
- Não se devem usar termos como Flag para nomear variáveis ou propriedades de estados que não são binários, ou seja, que podem assumir mais de dois valores.
- Qualificadores Media, Soma, Mínimo, Maximo, Índice devem ser anexados ao final da palavra quando apropriado.
- Não se misturam palavras de mais de uma língua, exceto para nome de classes que representam exceções e, portanto, finalizam com "Exception".
- Não se utiliza nenhum caractere especial nem específico de uma língua

A nomenclatura das *enumerações, delegates*, estruturas e propriedades de classe e de instância devem atender aos requisitos abaixo:

- Todas as palavras que compõem o nome iniciam com letras maiúsculas.
- Os campos definidos dentro de uma estrutura devem seguir a "notação camelo".
- O nome dos tipos especificados por uma enumeração também tem todas as palavras que o compõe iniciando com letras maiúsculas.
- Os tipos pelo comando *enum* devem todos ser formados por letras maiúsculas separadas pelo *caracter* '_'.
- É recomendada a utilização de vírgula mesmo após o último elemento de uma enumeração. Isso melhora a manutenção do código e passa sem problemas pelo processo de compilação. A nomenclatura das variáveis locais e parâmetros devem atender aos requisitos abaixo:
- A primeira letra das palavras, exceto da primeira palavra é maiúscula.
- Não utilizar o mesmo nome de parâmetro para as variáveis locais
- Mesmo variáveis que aparecerão em poucas linhas de código devem ter um nome significativo.
- Os contadores de laços são nomeados com letras do alfabeto. A letra *i* é a primeira a ser utilizada; sendo necessário o uso de outros contadores, declara-se a letra *j* e assim sucessivamente até a última letra do alfabeto.
- Variáveis locais correspondendo a exceções devem ser declaradas apenas se forem ser utilizadas. De preferência, deve ser utilizado o nome *ex*.

Os nomes das constantes são compostos de palavras com todas as letras maiúsculas utilizando underscores como separadores.

Não se devem usar números ou string literais, tais números em lugares que deveriam ser constantes ou mesmo variáveis.

4.13.2 Documentação

A linha imediatamente anterior à definição da enumeração, estrutura, constante, variável de classe, variável de instância, variável local, parâmetro, propriedade de classe ou propriedade de instância deve conter um comentário descrevendo seu propósito. Esse texto deve estar dentro das tag <summary>.

Faz parte da documentação a descrição do invariante que se aplique à variável.

Por exemplo: “O intervalo de valores possíveis para a variável diaMes é de 1 a 31” ou “a soma desta variável com a variável complemento deve dar 100”. Esse texto deve estar dentro da tag <value> ou <summary>.

Caso alguma variável ou constante não seja declarada privada, deve-se documentar a justificativa para isso. Esse texto deve estar dentro da tag <remarks>.

4.13.3 Nomenclatura do Métodos

Os nomes dos métodos devem atender aos requisitos abaixo:

- A primeira letra das palavras é maiúscula.
- Os nomes não são abreviados, exceto nos casos que a sua abreviação seja tão ou mais sugestiva que o nome completo.
- Não se misturam palavras de mais de uma língua.
- Não se utiliza nenhum caractere especial nem específico de uma língua.
- A primeira palavra deve ser um verbo no infinitivo representando a utilidade do método.
- Quando possível, deve-se utilizar overload de métodos para representar vários métodos que executam a mesma operação.
- Devem-se utilizar verbos nos nomes dos métodos que executam alguma operação no objeto, por exemplo, CadastrarFuncionario().
- É redundante repetir o nome da classe no nome do método. Assim, deve usar Livro.CalcularDesconto() ao invés de Livro.CalcularDescontoLivro().
- Deve-se evitar utilizar nomes que são sujeitos a interpretação pessoal, como AnaliseIsto().

Apesar da aplicação deste padrão, geralmente, resultar em nomes maiores, necessitando de digitação extra, o efeito da sua conformidade é um código mais fácil de compreender, pois o propósito do método já é esclarecido no seu nome.

Métodos de acesso a variáveis iniciam com get ou set finalizam com o nome da variável tendo a primeira letra de cada palavra maiúscula.

4.13.4 Declaração dos Métodos

Todas as declarações de variáveis locais são realizadas no início do código do método.

4.13.5 Documentação dos Métodos

A linha imediatamente anterior a definição do método deve conter um comentário contendo informações suficientes para seu entendimento e uso adequado (utilizando a *tag <summary>),* todos os parâmetros necessários para chamar o método (utilizando a *tag* <*param*>), sua cláusula de retorno (utilizando a tag <*returns*> e as exceções levantadas (utilizando a tag <*exception*>)

Caso a decisão de visibilidade (*public, protected e private*) do método possa ser questionada, documenta-se a razão pela qual foi tomada esta decisão (utilizando a *tag* <*remarks*>).

Se necessário, são declaradas ao final do comentário referências a outras classes e métodos (utilizando a *tag* <*see*> *e* <*seealso*>).

Caso necessário, outros itens são acrescentados ao cabeçalho acima, como, por exemplo, pré-condições e pós-condições, histórico de alterações do método, questões de concorrência, limitações e erros detectados no método (utilizando a *tag <summary>).*

Não é aconselhável o uso de comentários dentro do corpo dos métodos. Ao invés disto, é sugerido produzir métodos simples, com comentários acima de sua assinatura.

Caso o método possa ser melhorado ou corrigido, embora esteja funcionando corretamente, deve-se comentar isso informando a possível melhoria (utilizando a *tag* <*remarks*>).

4.13.6 Espaços em Branco

Uma palavra reservada seguida por parênteses não é separada destes por um espaço em branco. Um espaço em branco aparece depois de cada vírgula em uma lista de parâmetros ou na inicialização dos elementos de um *array*:

Todos os operadores binários, exceto o ".", são separados dos operadores por um espaço em branco, conforme os exemplo abaixo:

As expressões de um comando for são separadas por espaços em branco. Um cast é seguido por um espaço em branco.

4.14 Padronização para interfaces (telas)

Os componentes da interface ASP.NET devem obedecer às seguintes regras:

Os nomes dos componentes visuais possuem um prefixo e um sufixo. Os prefixos devem ser: lbl – label, Txt – textbox, ddl – drop down list, chk – checkbox, rb – radio button, img – image, dg – datagrig, gv – gridview.

O sufixo deve corresponder o que o campo representa em termos de requisitos. Por exemplo: NomePassageiro, TelefonePassageiro..

Exemplos de possíveis nomes: txtNomePassageiro e chkMasterCartaoCredito

O nome dos validadores é formado por duas partes: o nome do componente a que o validador se aplica + a string “Validator”.

Por exemplo:

Um validador para um componente chamado txtIdade deve se chamar *txtIdadeValidator*.

Caso o componente possua mais de um validador, o nome dos validadores deve conter também a indicação do tipo do validador.

Os possíveis tipos de validadors são: RQ – Required Field, Rv – Range Validator, Cv – Custom Validator. m Validator

Logo se o componente txtIdade possui um Required Field Validator e um Range Validator

Estes teriam os seguintes nomes:

Para o Required Field Validator: txtIdadeRQValidator;

Para o Range Validator : txtIdadeRVValidator

Caso existam mais de validator do mesmo tipo para um campo, adiciona-se uma string que descreve o tipo de validação entre o tipo de validator e o sufixo do nome do campo.

Por exemplo: validaRVMenorQue18IdadePassageiro.

Caso exista uma ValidationSummary na página ele deve se chamar sumarioValidacao

4.15 Padronização de dados

A padronização do banco de dados visa facilitar o entendimento por todos os envolvidos e evitar ambigüidades.

4.15.1 Tabelas

As tabelas devem ser nomeadas de forma simples e clara, de modo que seu nome não precise de mais informação complementar para descrever seu conteúdo.

Ex: Funcionário (Tabela que contém os dados dos funcionários).

Tabelas que são fruto de relacionamento entre duas tabelas devem conter o nome das duas tabelas separadas pelo caractere “_” (underscore). O nome da tabela mais importante deve vir primeiro. Caso ambas tenham a mesma importância, deve-se utilizar a primeira em ordem alfabética.

4.15.2 Campos

O nome dos campos deve iniciar com duas iniciais conforme o tipo do campo (ver tabela 1) seguido pelo caractere "_" (underscore) e do nome do campo. Novamente o nome do campo deve descrevê-lo de forma simples e clara. Os campos só devem conter letras maiúsculas, sem nenhum tipo de acentuação.

TIPO	DATA TYPE	ABREVIAÇÃO	EXEMPLO
Chave primária	Int identity	ID	ID_FUNCIONARIO
Número	Int	NR	NR_CALCADO
Data	DateTime	DT	DT_NASCIMENTO
Nome	Varchar	NM	NM_FUNCIONARIO
Boleano	Bit	IN	IN_ATIVO
Texto	Varchar	TX	TX_OBSERVACOES
Quantidade	Int	QN	QN_FILHOS
Valor	Float	VL	VL_CONTA
Percentual	Float	PR	PR_PRIMEIRO_AUXILIAR
Arquivo de Áudio	Varbinary(Max)	AD	AD_LAUDO
Arquivo de Imagem	Varbinary(Max)	IM	IM_EXAME

Tabela 1 – Abreviatura dos tipos de dados

4.16 Padronização ADO.NET

Os nomes das variáveis de tipos ADO.NET devem possuir um prefixo de acordo com o tipo do objeto.

O prefixo deve ser seguido pelo nome da tabela a qual a variável está associada. Por exemplo, um data reader que percorra a tabela "Clientes" deve-se se chamar drClientes.

4.17 Testes

4.17.1 Apresentação dos testes buscando qualidade

Um importante fator de sucesso de um software é sua qualidade. Existem diversas maneiras de avaliar a qualidade de um produto, uma dessas maneiras é a realização de atividades de testes de software por uma equipe especializada no assunto. O principal objetivo dessas atividades é descobrir se o produto está de acordo com as exigências do cliente. Segundo *Watts S. Humphrey*, criador do CMM (*Capability Maturity Model*), a qualidade do produto final é diretamente proporcional à qualidade do processo utilizado no seu ciclo de vida. Dessa forma, os testes fazem parte do processo de construção do software e esta metodologia é a primeira de muitas outras que virão e a cada versão ela vai atingindo um grau maior de maturidade. E o mais importante é que se trata de um reflexo do processo, ou seja, à medida que a equipe amadurece a metodologia também, pois ambas andarão juntas a partir de agora.

Nesta primeira versão, não serão adotados testes automático, e sim manual. Não se trata de tarefa menos importante ou fácil, muito pelo contrário, os testes automáticos não substituem os manuais. Os testes automáticos podem ser usados em tarefas repetitivas e longas. Um software gigante em que levaria três ou quatro dias para fazer os testes, através de uma ferramenta de testes levaria horas, por exemplo. Teste manual é um processo criativo, com ele não se testa somente o que deveria funcionar, mas o que não deveria também.

Para desempenhar corretamente o processo de testes, é importante iniciar as atividades o quanto antes em um projeto de desenvolvimento, se possível no primeiro dia do projeto. Dessa maneira, defeitos podem ser identificados mais cedo no produto, minimizando os custos da sua correção. Isso nos leva a concluir que não é uma prática vantajosa tratar testes como uma única atividade de execução, e deixá-las para o final do projeto. Isso poderá gerar uma corrida por correções de defeitos, e estudos mostram que quanto mais modificado é um código, maior a propensão desse código ao aparecimento de mais defeitos. Nesta primeira versão, o objetivo é ganhar experiência e aprender a usar a intuição e julgamento do que deve testar, quando, porque testar certas áreas e porque não testar outras. Como já foi dito, testes é um processo criativo, onde é necessário realizar inclusive testes exploratórios. Estes são ocorrem quando o testador não tem informações detalhadas sobre o que vai tesar e como vai

testar. O testador baseia-se na sua experiência, assim como no conhecimento que ele vai adquirindo durante o trabalho com testes. A partir dessa perspectiva, podemos afirmar que o teste exploratório é uma atividade iterativa e empírica de exploração que exigem idas e vindas num processo de investigação contínuo onde a intuição, a criatividade e a experiência do testador são indispensáveis para garantir a eficiência do teste.

O grande benefício da utilização de testes exploratórios está ligado à sua natureza empírica. Os testes automatizados repetem sempre os mesmos testes, os mesmos caminhos, as mesmas operações. Por meio dos testes exploratórios temos a oportunidade de empiricamente e iterativamente criar novos testes e seguir novos caminhos, que por sua vez, nos leva a descoberta de novos defeitos. E é isso que buscamos nesta primeira versão da Metodologia Prática, maturidade para avançar para um próximo passo, na próxima versão.

4.17.2 Equipe de testes

Ao contrário do que se pensava até pouco tempo, a equipe de testes deve ser composta por pessoas com habilidades, conhecimentos e atitudes. É necessário que o analista de testes conheça engenharia de software. Ele não precisa ser um analista de sistemas nem engenheiro de software, mas precisa entender os artefatos gerados para o software, pois a partir deles que será gerado o plano de testes. O analista precisa ter boa comunicação escrita e falada, pois além do plano de teste, ele vai gerar relatórios, incluir informações na ferramenta de gerenciamento de testes e, portanto é necessário escrever corretamente. Quanto à comunicação falada, deve ser clara, pois muitas situações vão ser resolvidas através de diálogos. É importante fazer perguntas bem elaboradas e manter interação com os desenvolvedores para evitar mal entendidos. Lembrando que entre a equipe de testes e desenvolvimento, existe uma competição saudável, mas de maneira nenhuma deve ser levada para o lado pessoal. O objetivo das duas equipes é a qualidade do produto e quando elas atingem esta maturidade, o projeto e a empresa ganham. Outra atitude que a equipe de testes precisa ter é pro atividade. Geralmente os testes são efetuados na última etapa antes da entrega do produto. E, portanto, pode ocorrer de já começar com atraso e prazo reduzido devido às fases anteriores. O analista de testes precisa ser proativo e não ficar parado ao encontrar obstáculos, pois o tempo está correndo e pode não conseguir recuperação. Sem contar que é justamente nesta fase (manutenção) que ocorrem as maiores inclusões de erros.

Diplomacia é outra característica do analista de testes, pois como já foi dito, a interação tester/desenvolvedor pode ser tenso. De forma que é importante ter delicadeza para conduzir uma questão, pois o analista de testes é na maioria das vezes portador de más notícias. Outra tarefa realizada pela área de qualidade que é muito importante é a usabilidade, é importante que o analista de testes aprenda a ter olhos críticos, simplificando e facilitando as interfaces. Ele precisa ser uma espécie de "advogado" do usuário.

4.17.3 Detalhamento dos modelos de documentos a serem usados

Como já dito anteriormente, nesta primeira versão, a Metodologia Prática implementará os testes manuais. O artefato que será usado será apenas uma planilha com testes pré-estabelecidos e usará uma ferramenta para gerenciamento de testes.

4.17.4 Apresentação de ferramentas de apoio para testes

Para este primeiro momento a ferramenta de apoio será o *Mantis*. Será usado juntamente com a planilha de testes. Trata-se de uma ferramenta *Open Source* desenvolvida em *PHP* e usa o *MySql* como base de dados. É uma ferramenta que possibilita gerar níveis de perfil.

5 CONCLUSÃO

Ao desenvolver um software, sempre houve a preocupação de construir um produto de qualidade, que atenda aos requisitos do cliente e que seja construído com o mínimo de erros possíveis. O presente trabalho demonstrou a Metodologia Prática, em todos os seus conceitos particularidades. Esta metodologia é na verdade um processo híbrido, pois é fruto de modelos tradicionais e ágeis, usando técnicas de ambos, para atingir maiores níveis de satisfação e qualidade, diminuindo o tempo de desenvolvimento.

A Metodologia Prática foi desenvolvida para padronizar o desenvolvimento de software e automatizar o processo, além obrigar o analista/desenvolvedor seguir boas práticas de desenvolvimento de softwares. Foi escrita de forma bem clara e simples, pensando em atingir profissionais que encontrarão facilidade em sua implementação, e iniciantes, que tem poucos conceitos sobre processo.

Os processos convencionais como o RUP, XP, SCRUM, citam muitos recursos, sobretudo em análise de sistemas, ferramentas, notações, linguagens de definição, como a UML (Unified Modeling Language), mas por outro lado estes processos não dizem como se fazer uma análise completa, ou como modelar um projeto ou até mesmo, quando ou como usar a UML. Cabe a uma metodologia esta função. E neste estudo foi demonstrado as funcionalidades da Metodologia Prática que mostra detalhadamente como usar a UML com seus diagramas necessários; a padronização da codificação, detalhando as regras de construção de classes, métodos, interfaces, tabelas e campos do banco de dados; como modelar um projeto usando ferramentas Case, gerando artefatos para documentação do software; traz a padronização de classes, métodos e interfaces; a criação do plano de testes e a definição das ferramentas de apoio e artefatos gerados para a sua validação; a prototipação das telas, definindo as ferramentas de apoio a prototipação.

A Metodologia Prática não será eficaz, nem um bom processo a ser seguido, se não for usada com critério e bom senso, levando em conta o projeto. Cabe ao gestor do projeto adotá-la e seguir o processo definido por ela em todo o projeto.

6 REFERÊNCIAS BIBLIOGRÁFICAS

PRESSMAN, Roger S. Engenharia de Software, 6ª edição, Rio de Janeiro: McGraw Hill, 2002

NOGUEIRA, Marcelo - Engenharia de Software: Um Framework para Gestão de Riscos em Projetos de Software, Rio de Janeiro: Editora Ciência Moderna Ltda, 2009.

HEINRIK Kniberg – Scrum e XP direto das Trincheiras, Rio de Janeiro: C4Media Inc, 2007.

KRUCHTEN , Philippe. Introdução ao RUP Rational Unified Process. ISBN: 85-7393-275-9. Ciência Moderna, 2001.

BECK, K.; ANDRES, C. Extreme Programming Explained : Embrace Change (2nd Edition). [S.l.]: Addison-Wesley Professional, 2004. ISBN 0321278658.

HEU, Heumann, J. "What does "No time for Requirements" mean?". The Rational Edge,Nov. 2001, 7p.

LEITE, J.C.S.P., "<"Entrevistas">","<"07/2007 ">", em Livro Vivo : Engenharia de Requisitos, http://livrodeengenhariaderequisitos.blogspot.com/, 2007]

http://msdn.microsoft.com/library/default.asp?url=/library/en-

7 APÊNDICE

Anexo a este trabalho, está o produto desenvolvido: Uma Metodologia aplicada a uma Empresa de Tecnologia da Informação, com todos os processos detalhados e explicados passo a passo.