FACULDADE DE TECNOLOGIA DE SÃO PAULO

ANDRÉ NOBREGA MARQUES

Metodologias ágeis de desenvolvimento:

Processos e Comparações

São Paulo

2012

FACULDADE DE TECNOLOGIA DE SÃO PAULO

ANDRÉ NOBREGA MARQUES

Metodologias ágeis de desenvolvimento:

Processos e Comparações

Monografia submetida como exigência parcial para a obtenção do Grau de Tecnólogo em Processamento de Dados

Orientador: Prof. Valter Yogui

São Paulo

2012

Dedicatória

Este trabalho é dedicado a Meire Cristina Lemos Rossi (13/05/1966 – 07/08/2012), minha sogra, a qual nos deixou muito cedo e sempre foi uma excelente mãe, esposa, filha, irmã e sogra.

Agradecimentos

Agradeço primeiramente a Deus, que me auxiliou ao longo de toda a minha trajetória acadêmica desde meus primeiros passos na pré-escola, me ajudou a atingir as notas necessárias para cursar meu colegial na escola federal de São Paulo e também na FATEC-SP.

Também agradeço aos meus pais que sempre se esforçaram para pagar pelos meus estudos desde o maternal até a oitava série, além de sempre me apoiarem e me ajudarem nos momentos de dificuldade dessa longa trajetória.

Agradeço também a minha esposa e futura mãe dos meus filhos, se Deus quiser, que sempre me apoiou e esteve ao meu lado desde antes de eu iniciar meu curso nesta faculdade. Além dela merecem destaque todos meus familiares que também sempre me apoiaram (irmãos, tios, primos e avós).

Todos os professores que me ajudaram também em toda a minha vida acadêmica para que chegasse até esse momento merecem destaque também, desde a Lua de Cristal (escola na qual cursei o maternal e jardins), passando pelo Colégio Orlando Garcia, Colégio Módulo, Centro Federal de Educação Tecnológica de São Paulo até chegar à FATEC-SP. Dentre esses, agradeço especialmente ao Valter Yogui, meu instrutor de TCC, que deu sua maior contribuição no momento crucial da escolha do tema, que, se não fosse por ele, resultaria em um trabalho muito mais generalista.

Sumário

Resumo

Há diversas metodologias ágeis surgindo nos últimos anos, e cada vez mais se entra em discussão sobre a validade e efetividade delas, muitas empresas já as adotaram e, boa parte, relata ter ótimos resultados.

Este trabalho visa detalhar algumas das principais metodologias ágeis (*Scrum, XP, FDD, Lean Software Development* e *Crystal Family*) e fazer a comparação dos processos das que mais estão sendo utilizadas hoje em dia (*Scrum*, *XP* e *FDD*), além de relatar alguns casos de sucesso e fracasso das metodologias ágeis em geral. Dessa forma, será possível conhecer essas metodologias além de identificar os pontos que tornam cada uma delas interessantes e também verificar a efetividade que está sendo relatada por quem as utiliza.

Abstract

There are many agile methodologies arising in the last few years, and each day we get more into the discussion about their validity and effectiveness, there are many companies that already adopt them and, many of them relate good results.

This work intent to detail some of the principal agile methodologies (Scrum, XP, FDD, Lean Software Development and Crystal Family) and compare the processes of the most used ones (Scrum, XP and FDD), in addition to report some successful cases and some failure ones of the agile methodologies. This way, it will be possible to know these methodologies in addition to identify the principles that make each of them so interesting and also check the effectiveness related by the users.

Introdução

A era da computação iniciou-se nos anos 40, e os maiores investimentos eram voltados a hardware. No início dos anos 50, passou-se a ter o domínio da tecnologia de hardware e então os investimentos se voltaram ao desenvolvimento dos sistemas operacionais o que possibilitou o surgimento das primeiras linguagens de programação de alto nível, permitindo assim que usuários pudessem se concentrar mais no desenvolvimento sem se preocuparem com questões técnicas do funcionamento do hardware. O surgimento de sistemas operacionais com multiprogramação, no início dos anos 60, possibilitou um aumento na eficiência destes, contribuindo para a queda de preço dos hardwares.

Com o desenvolvimento dos hardwares e dos sistemas operacionais, passou-se a haver a necessidade de sistemas mais complexos e maiores em substituição aos pequenos aplicativos que existiam até o momento e foi aí que se iniciou a “crise do software” pela incompatibilidade dos métodos utilizados até então com os métodos necessários para a criação desses sistemas. Em 1968 foi realizada uma conferência pelo Comitê de Ciência da NATO (*North Atlantic Treaty Organization*) com o nome “Engenharia de Software” e foi aonde, pela primeira vez, foi utilizado esse termo. Nessa conferência foi discutido sobre a existência de uma real crise de software, e o que se constatou foram, ao menos, os seguintes problemas:

- Cronogramas não observados;
- Projetos com tantas dificuldades que são abandonados;
- Módulos que não operam corretamente quando combinados;
- Programas que não fazem exatamente o que era esperado;
- Programas tão difíceis de usar que são descartados;
- Programas que simplesmente param de funcionar.

E foi a partir desse cenário que passou a se desenvolverem os processos e metodologias de desenvolvimento de software. Um dos primeiros processos, que surgiu em 1970, foi o que ficou conhecido como processo em cascata, ele possui sete fases e uma só poderia ser iniciada após o término da anterior, daí o nome, isso acaba causando alguns transtornos, afinal em muitas vezes o cliente gostaria de

fazer alguma mudança no sistema, porém descobre isso quando a fase de requerimentos já passou o que acaba deixando o projeto defasado ou mais caro por não prever de início esse tipo de mudança.

Nos anos 80, surgiu outro processo para desenvolvimento de software, o processo em espiral, que apareceu como a solução para os problemas existentes no modelo em cascata, a exemplo do "*ciclo de Demming*" (ciclo PDCA), esse modelo surgia com apenas quatro fases (Planejamento, Avaliação, Análise de Risco e Engenharia), o processo inicia no planejamento, vai para a avaliação, análise de risco, engenharia e, seguindo a ideia do espiral, volta para o planejamento fazendo todo o ciclo novamente, idealizando assim um modelo iterativo e incremental, permitindo que os erros ocorridos em uma fase possam ser revistos.

Ainda assim, com o passar do tempo, a complexidade dos sistemas tem aumentado ainda mais, os sistemas possuem mais usuários e tornam-se cada vez mais importantes, podendo, em muitas vezes, gerenciar a vida ou a morte de pessoas, como é o caso de softwares que calculam quantidades de medicamentos a serem aplicados em pacientes ou que monitoram o estado de saúde dos pacientes, os sistemas também controlam a economia e as armas do mundo, é possível ver isso através do pânico que foi causado pela possibilidade do "bug" do milênio no final dos anos 90.

Unindo-se aos problemas até então encontrados e à importância atual dos sistemas informatizados, surgiram alguns teóricos discordando da ideia de tratar o desenvolvimento de software como uma fábrica de produção em série, um exemplo é Cockburn que compara o desenvolvimento de software ao ato de escrever uma poesia épica em conjunto com diversas pessoas: seriam diversas pessoas, com diversos argumentos, tentando dar seu melhor sem talento, tempo ou recursos suficientes.

Foi a partir desses pensamentos que surgiram as metodologias ágeis de desenvolvimento, a partir de 2001, com a assinatura do manifesto ágil, que estabeleceu os princípios das metodologias ágeis:

> "Nossa maior prioridade é satisfazer o cliente, através da entrega adiantada e contínua de software de valor.
>
> Aceitar mudanças de requisitos, mesmo no fim do desenvolvimento. Processos ágeis se adequam a mudanças, para que o cliente possa tirar vantagens competitivas.

Entregar software funcionando com frequência, na escala de semanas até meses, com preferência aos períodos mais curtos.

Pessoas relacionadas a negócios e desenvolvedores devem trabalhar em conjunto e diariamente, durante todo o curso do projeto.

Construir projetos ao redor de indivíduos motivados. Dando a eles o ambiente e suporte necessário, e confiar que farão seu trabalho.

O Método mais eficiente e eficaz de transmitir informações para, e por dentro de um time de desenvolvimento, é através de uma conversa cara a cara.

Software funcional é a medida primária de progresso.

Processos ágeis promovem um ambiente sustentável. Os patrocinadores, desenvolvedores e usuários, devem ser capazes de manter indefinidamente, passos constantes.

Contínua atenção a excelência técnica e bom design, aumento da agilidade.

Simplicidade: a arte de maximizar a quantidade de trabalho que não precisou ser feito.

As melhores arquiteturas, requisitos e designs emergem de times auto-organizáveis.

Em intervalos regulares, o time reflete em como ficar mais efetivo, então, se ajustam e aperfeiçoam seu comportamento de acordo".

SCRUM

História

Ao que se é possível encontrar, a primeira vez em que esse termo, originário do *rugby*, foi utilizado como comparativo a metodologias de desenvolvimento foi em 1986 no artigo "*The New New Product Development Game*" (O novo jogo de desenvolvimento de produtos novos), por Hirotaka Takeuchi e Ikujiro Nonaka. Nesse artigo os autores diziam: "A nova ênfase em velocidade e flexibilidade demandam uma abordagem diferente no gerenciamento do desenvolvimento de novos produtos. A abordagem da sequência tradicional ou "corrida de revezamento" para o desenvolvimento de produtos - exemplificado pelo sistema de planejamento do programa em fases da "*National Aeronautics and Space Administration*" – podem conflitar com os objetivos de velocidade e flexibilidade máximas. Ao invés disso, uma abordagem integral ou de *rugby* - na qual um time tenta avançar como uma unidade, passando a bola para trás e adiante – possa atender melhor aos requisitos competitivos de hoje em dia".

No início dos anos 90, mais precisamente em 1991, Peter DeGrace e Leslie Hulet Stahl, utilizaram o termo "abordagem *Scrum*" para fazer referência aos processos e métodos, outrora descritos por Takeuchi e Nonaka, em seu artigo "*Wicked Problems, Righteous Solutions - A Catalogue of Modern Software Engineering Paradigms*" (Problemas cruéis, soluções justas – Um catálogo dos paradigmas da engenharia de software moderna). Ainda nos anos 90, Ken Schwaber começa a utilizar essa abordagem em sua empresa, Jeff Sutherland também desenvolve uma abordagem similar, a qual se refere apenas como "*Scrum*".

A apresentação oficial do *Scrum*, bem como sua publicação ocorreu em 1995, na OOPSLA (*Object-Oriented Programming, Systems, Languages & Applications* – Programação, Sistemas, Linguagens e Aplicações orientadas a objeto), e então passaram a haver atualizações constantes com contribuições significativas nos cinco anos seguintes de Mike Beadle e Martine Devos.

Valores

Conforme Schwaber e Sutherland (2011), ele é baseado nas teorias empíricas de controle de processo e, como tal, acredita que o conhecimento vem da experiência e da tomada de decisões baseadas naquilo que é conhecido. Assim como na maioria das metodologias ágeis, o *Scrum* prega um processo iterativo e incremental para o desenvolvimento de um projeto, de forma que se torne mais previsível e tenha riscos mais controláveis.

Dentre os diversos valores e regras que regem o *Scrum*, há três que são mais importantes:

- **Transparência:** Todos os aspectos mais relevantes do processo tem que estar ao alcance dos responsáveis pelos resultados, sendo esses definidos de uma forma que estejam claros e causando o mesmo entendimento para estes, ou seja, todos devem “falar a mesma língua”.
- **Inspeção:** Os usuários da metodologia devem verificar, com certa frequência, se os mecanismos *Scrum* e o progresso do projeto para identificar qualquer andamento que não esteja indo de acordo com o esperado. O ideal é que tal verificação seja feita por inspetores especializados em tal trabalho para que tenha um resultado melhor e também para que não venha a causar prejuízo de tempo àqueles que deveriam estar executando outras tarefas.
- **Adaptação:** Caso seja identificado, durante a inspeção, que os aspectos de algum processo foram desviados além do que é aceitável, assim o será o produto, portanto o processo deve ser adaptado o quanto antes para eliminar possíveis futuros desvios.

Há quatro oportunidades formalmente descritas pelo *Scrum* em que a inspeção e a adaptação podem ser aplicadas:

- Reunião de planejamento do *Sprint*
- Reunião diária (*Daily Scrum*)
- Reunião de revisão do *Sprint*
- Retrospectiva do *Sprint*

Papéis

Os times *Scrum* foram feitos para se organizarem sozinhos e terem diversas funções. A ideia é que uma equipe que se organiza por si só saiba definir de forma melhor como completarem seu trabalho em relação a quando são dirigidos por uma pessoa de fora. Quanto às diversas funções, a ideia é que todas as habilidades necessárias para resolver todas as questões que surgirem estejam dentro da equipe, sem a dependência de outros que não participam ativamente da equipe.

Os profissionais integrados a esse time são divididos em três papéis:

Product Owner ("Dono do produto")

Ele é o responsável por fazer com que o produto tenha o maior valor possível. As estratégias para que esse objetivo seja atingido podem ser as mais variadas possíveis e devem levar em consideração os times *Scrum*, suas organizações e os indivíduos que fazem parte.

É também o *Product Owner* o responsável por atualizar e gerenciar o *Backlog* do produto (termo explicado a frente), o que inclui, conforme o manual do *Scrum* [2011]:

> "- Expressar claramente os itens do Backlog do Produto;
> - Ordenar os itens do Backlog do Produto para alcançar melhor as metas e missões;
> - Garantir o valor do trabalho realizado pelo Time de Desenvolvimento;
> - Garantir que o Backlog do Produto seja visível, transparente, claro para todos, e mostrar o que o Time Scrum vai trabalhar a seguir; e,
> - Garantir que a Equipe de Desenvolvimento entenda os itens do Backlog do Produto no nível necessário." (SCHWABER, SUTHERLAND; 2011)

É importante ressaltar que o *Product Owner* é responsável por esses itens, porém não necessariamente precisa executá-los, podendo delegá-los, por exemplo, à equipe de desenvolvimento.

É de extrema importância para o bom desenvolvimento e sucesso do produto

que a figura do *Product Owner* seja respeitada bem como suas decisões, ninguém poderá alterar prioridades ou dar ordens à equipe de desenvolvimento sem passar por ele, assim como a equipe de desenvolvimento não pode acatar a alguma ordem que não venha dele.

Equipe de Desenvolvimento

A equipe de desenvolvimento é responsável pela realização das tarefas "executáveis" do *Backlog*, deixando-as "prontas" e entregando uma versão incrementada utilizável do produto ao final de cada *Sprint*, ela é estruturada e tem autonomia de organizar e gerenciar seu próprio trabalho como considerar melhor.

Conforme Schwaber e Sutherland (2011), uma equipe de desenvolvimento deve possuir as seguintes características:

> "- Elas são auto-organizadas. Ninguém (nem mesmo o Scrum Master) diz à Equipe de Desenvolvimento como transformar o Backlog do Produto em incrementos de funcionalidades potencialmente utilizáveis;
> - Equipes de Desenvolvimento são multifuncionais, possuindo todas as habilidades necessárias, enquanto equipe, para criar o incremento do Produto.
> - O Scrum não reconhece títulos para os integrantes da Equipe de Desenvolvimento que não seja o Desenvolvedor, independentemente do trabalho que está sendo realizado pela pessoa; Não há exceções para esta regra.
> - Individualmente os integrantes da Equipe de Desenvolvimento podem ter habilidades especializadas e área de especialização, mas a responsabilidade pertence à Equipe de Desenvolvimento como um todo; e,
> - Equipes de Desenvolvimento não contém subequipes dedicadas a domínios específicos de conhecimento, tais como teste ou análise de negócio."

Para o bom funcionamento da metodologia, é recomendado que as equipes de desenvolvimento possuam entre três e nove integrantes, menos que isso pode significar uma menor interação que pode resultar menor produtividade, além de, por possuir poucos integrantes, pode apresentar uma deficiência nas habilidades necessárias para que uma versão seja entregue realmente funcional. Mais de nove integrantes em uma equipe de desenvolvimento pode gerar uma complexidade acima da capacidade de um processo empírico gerenciar.

Scrum Master

O *Scrum Master* é algo parecido com um auditor da metodologia, o qual fará parte do time para assegurar que a teoria, a prática e as regras do *Scrum* estão sendo aplicadas de forma correta. Além desse papel de auditor, o *Scrum* máster é o elo entre o time e o mundo exterior, permitindo que apenas ocorram interações que de fato agreguem valor ao produto.

Suas interações com o *Product Owner* ocorrem, entre outras maneiras, das seguintes:

> "- Encontrando técnicas para o gerenciamento efetivo do Backlog do Produto;
> - Claramente comunicar a visão, objetivo e itens do Backlog do Produto para a Equipe de Desenvolvimento;
> - Ensinar a Time Scrum a criar itens de Backlog do Produto de forma clara e concisa;
> - Compreender a longo-prazo o planejamento do Produto no ambiente empírico;
> - Compreender e praticar a agilidade; e,
> - Facilitar os eventos Scrum conforme exigidos ou necessários." (SCHWABER, SUTHERLAND; 2011)

Já quando se trata da equipe de desenvolvimento, essas interações ocorrem, entre outras maneiras, das seguintes:

> "- Treinar a Equipe de Desenvolvimento em autogerenciamento e interdisciplinaridade;
> - Ensinar e liderar a Equipe de Desenvolvimento na criação de produtos de alto valor;
> - Remover impedimentos para o progresso da Equipe de Desenvolvimento;
> - Facilitar os eventos Scrum conforme exigidos ou necessários; e,
> - Treinar a Equipe de Desenvolvimento em ambientes organizacionais nos quais o Scrum não é totalmente adotado e compreendido." (SCHWABER, SUTHERLAND; 2011)

Enfim, com a organização, as interações, devem ocorrer das seguintes maneiras, entre outras:

> "- Liderando e treinando a organização na adoção do Scrum;
> - Planejando implementações Scrum dentro da organização;
> - Ajudando funcionários e partes interessadas a compreender e tornar aplicável o Scrum e o desenvolvimento de produto empírico;
> - Causando mudanças que aumentam a produtividade do Time Scrum; e,
> - Trabalhando com outro Scrum Master para aumentar a eficácia da aplicação do Scrum nas organizações." (SCHWABER, SUTHERLAND; 2011)

Eventos *Scrum*

Para uma melhor organização, o *Scrum* se divide em eventos com duração máxima estabelecida.

Sprint

O *Sprint* é um dos principais componentes do *Scrum*, esse evento tem como missão entregar uma versão incremental utilizável ao seu final, para que isso ocorra, o *Sprint* deve possuir uma definição do que deve ser construído e um plano para que isso ocorra. Ele possui duração máxima de um mês, afinal em um tempo maior que esse poderia ocorrer mudanças no que deveria ser construído, aumentando a complexidade e o risco, além disso, se houver qualquer problema apenas o orçamento de um mês estará comprometido e não todo ele.

Conforme Schwaber e Sutherland (2011) há algumas regras que devem ser seguidas para que um *Sprint* tenha o andamento correto:

> "- Não são feitas mudanças que podem afetar o objetivo do Sprint;
> - A composição da equipe de desenvolvimento permanece constante;
> - As metas de qualidade não diminuem; e,
> - O escopo pode ser clarificado e renegociado entre o Product Owner e a equipe de desenvolvimento quanto mais for aprendido."

Um *Sprint* pode ser cancelado a qualquer momento, antes de sua finalização, pelo *Product Owner*, que normalmente o faz quando a organização muda sua direção ou quando as condições do mercado ou da tecnologia mudam, fazendo com que o objetivo do *Sprint* se torne obsoleto.

Há quatro outros eventos em que se divide o *Scrum*, são eles: reunião de planejamento do *Sprint*, reuniões diárias, revisão do *Sprint* e retrospectiva do *Sprint*.

Reunião de planejamento do Sprint

Todo o time *Scrum* participa da reunião de planejamento do *Sprint*, que deve durar, no máximo, 8 horas (considerando um *Sprint* de um mês) para responder a duas perguntas: “O que ficará ‘pronto’ nesse *Sprint*?” e “Como ficará ‘pronto’ o que foi escolhido para esse *Sprint*?”. O objetivo é que ao final da reunião a equipe de desenvolvimento possa explicar ao *Product Owner* e ao *Scrum Master* como pretende trabalhar de forma auto-organizada para atingir o objetivo do *Sprint* e criar um incremento previsto.

O que ficará ‘pronto’ nesse *Sprint*?

A equipe de desenvolvimento faz uma previsão de tudo que conseguirá desenvolver durante o *Sprint* levando em consideração e selecionando os itens do *Backlog* do produto. Feito isso o time *Scrum* determina a meta do *Sprint*, que é o objetivo, dentro do *Sprint*, da implementação de cada item selecionado do *Backlog* do produto.

Como ficará “pronto” o que foi escolhido para esse *Sprint*?

A equipe de desenvolvimento deve determinar como executará e quando entregará cada um dos itens selecionados para a entrega da versão ao fim do *Sprint*, o conjunto desse plano de entrega com os itens selecionados é chamado de *Backlog* do *Sprint*.

É interessante que durante essa parte da reunião o *Product Owner* esteja presente para negociar, junto à equipe de desenvolvimento, o que entrará no *Sprint* e o que pode ser trocado, além de ajudar a esclarecer quaisquer itens que não estejam tão claros quanto deveriam.

Reunião diária

A reunião diária também é chamada de "*Stand-up meeting*" (reunião em pé), pois em alguns casos é recomendável que ela seja feita literalmente dessa forma, "em pé", para que possa respeitar o limite de 15 minutos. A ideia dessas reuniões é que a equipe de desenvolvimento sincronize as atividades e crie um plano para o próximo dia. Durante essa reunião, cada desenvolvedor deve responder três perguntas:

> "- O que foi completado desde a última reunião?
> - O que será feito até a próxima reunião?
> - Quais os obstáculos que estão no caminho?" (SCHWABER, SUTHERLAND; 2011)

O *Scrum Master* deve fiscalizar a equipe de desenvolvimento para verificar se essa reunião realmente está sendo feita de forma diária e se apenas os membros dessa equipe estão participando, afinal somente aqueles que de fato transformarão os itens do *Backlog* na versão do *Sprint* tem importância para isso (principalmente se considerarmos que a equipe de desenvolvimento deve ser auto-organizada).

Revisão do Sprint

Finalizado o *Sprint*, é feita a reunião de revisão do *Sprint*. Nessa reunião, que deve durar quatro horas para um *Sprint* de 30 dias, os seguintes elementos são verificados, conforme Schwaber e Sutherland (2011):

> "- O Product Owner identifica o que está "Pronto" e o que não está;
> - A Equipe de Desenvolvimento discute o que foi bem durante o Sprint, quais problemas ocorreram dentro do Sprint, e como esses problemas foram resolvidos;
> - A Equipe de Desenvolvimento demonstra o trabalho que está "Pronto" e responde as questões sobre o incremento;
> - O Product Owner discute o Backlog do Produto tal como está. Ele (ou ela) projeta as prováveis datas de conclusão baseado no progresso até a data; e,
> - O grupo todo colabora sobre o que fazer a seguir, e é assim que a Reunião de Revisão do Sprint fornece valiosas entradas para a Reunião de Planejamento do próximo Sprint."

Como resultado dessa reunião, obtém-se o *Backlog* do produto revisado.

Retrospectiva do Sprint

A retrospectiva do *Sprint* tem como objetivo verificar as formas de trabalho e práticas utilizadas no último *Sprint* e identificar o que deve passar a ser utilizado também nos próximos *Sprints* por terem sido elementos que comprovadamente contribuíram para o bom desenvolvimento do trabalho e também para melhor qualidade do produto.

A ideia é que, como produto dessa reunião, tenham sido identificadas todas as melhorias que serão implementadas no próximo *Sprint*.

Artefatos do *Scrum*

Os artefatos do *Scrum* são elementos que auxiliam no desenvolvimento do produto, visando facilitar a transparência das informações que são utilizadas no projeto para todo o time *Scrum*.

Backlog do Produto

O *Backlog* do produto é uma lista ordenada que concentra todos os requisitos, características, funções, requisitos, melhorias e correções do produto. Ele fica em constante atualização e suas primeiras versões contêm apenas os requisitos conhecidos inicialmente e melhor entendidos. Enquanto o produto existir esse *Backlog* ganhará novas versões para tornar-se mais apropriado à realidade existente. A ordenação dessa lista, normalmente, respeita os critérios de valor, risco, prioridade e necessidade.

Esse artefato é utilizado para determinar tudo o que pode ser feito dentro de cada *Sprint*, esses itens são os que podem ser desenvolvidos (ou “prontos”) pela equipe de desenvolvimento no *Sprint* são considerados “disponíveis” ou “executáveis” para seleção na reunião de planejamento do *Sprint*.

Backlog do Sprint

O *Backlog* do *Sprint* é o produto da reunião de planejamento do *Sprint*, nele estão contidos todos os itens selecionados do *Backlog* do produto que serão produzidos no *Sprint* em questão.

No *Backlog* do *Sprint*, além dos itens que serão desenvolvidos, está contido um plano detalhado para que as mudanças envolvidas sejam devidamente entendidas por todo o time.

Incremento

O incremento é a versão utilizável do produto que surge como resultado do *Sprint*, contendo todos os itens listados no *Backlog* do *Sprint*.

XP (Extreme Programming)

História

Os principais colaboradores e criadores da *XP* são Kent Beck e Ward Cunningham, diversas das principais características da *XP* foram herdadas da comunidade que ambos faziam parte, a comunidade SmallTalk (linguagem de programação orientada a objetos criada em meados dos anos 1960), características como adaptação à mudança, desenvolvimento iterativo, ênfase nos testes, integração contínua, programação pareada e refatoração.

No ano de 1999, foi a primeira vez em que o termo "*XP*" (*extreme programming*) foi utilizado, quando o programador Kent Beck trabalhava como líder de um projeto na "*Chrysler Comprehensive Compensation*", um projeto de longo prazo que visava reescrever a aplicação de folha de pagamento da Chrysler Corp. Segundo Wells (1999), certo dia Beck se perguntou "Há algumas atividades que contribuem para o sucesso do desenvolvimento de software, o que aconteceria se as fizéssemos o mais intensamente possível?" e daí veio o nome "Programação Extrema".

Após essa experiência, em 1999, Beck lançou o primeiro livro sobre *XP*, "*Extreme Programming: Explained: Embrace Change*", o qual recebeu o prêmio JOLT de produtividade da revista "*Software Development*".

Valores

Os valores aplicados pelo *XP* são nada mais do que uma forma de trabalhar em harmonia com a equipe e com os valores da empresa. Ele ainda permite que, além de seus valores padrões, cada um que for aplicar a metodologia incorpore seus valores a ela, conforme as mudanças que forem feitas nas regras do *XP*. Os valores padrões são:

- **Simplicidade**: Não se deve ser feito nada além do que foi solicitado, afinal se deve trabalhar sobre a solicitação pela qual foi feita o investimento, isso aumentará o valor gerado até a data final. Dessa forma, pode-se aos poucos caminhar rumo ao objetivo analisando calmamente cada uma das falhas que ocorrerem durante o percurso.
- **Comunicação**: Os membros da equipe devem trabalhar próximos, comunicando-se diariamente e pessoalmente, todos contribuindo desde o levantamento de requisitos até a codificação e entrega ao cliente.
- **Retorno**: Devem ser entregues versões funcionais do programa periodicamente. Essas versões periódicas serão demonstradas para que todas as alterações nos processos ou no projeto sejam feitas conforme a necessidade.
- **Respeito**: Todos os membros da equipe devem ser igualmente respeitados, assim como seus conhecimentos. Dessa forma o cliente deve respeitar os desenvolvedores acerca de seus conhecimentos técnicos, assim como os desenvolvedores devem respeitar o cliente quanto aos seus conhecimentos do negócio.
- **Coragem**: Sempre se deve dizer a verdade e trabalhar em cima dela. Deve-se ter a coragem também de fazerem as mudanças necessárias para que o projeto tenha sucesso.

Papéis

Há apenas alguns papéis descritos no *XP*, os quais serão descritos a seguir.

Cliente

É responsável por criar e priorizar as “histórias do usuário” que devem ser implementadas no projeto. Como ele é o responsável por priorizar as histórias, ele que deve controlar o que será entregue em cada versão e que deve decidir quais histórias devem ser removidas ou adicionadas para que o prazo da versão possa ser cumprido ou para que ela possa ter mais valor.

Desenvolvedor

São responsáveis por, em conjunto, estimarem o custo de cada história e os recursos necessários para que sejam desenvolvidas, além de transformá-las em “tarefas”. Quando se juntam em pares, devem responsabilizar-se pelas tarefas que escolherem desenvolver, escrever os testes unitários e o código.

Técnico

O *coach* é responsável por monitorar o processo de desenvolvimento, o modo que as técnicas e processos *XP* estão sendo aplicados e manter o foco do time em possíveis problemas e melhorias.

Rastreador

É responsável por verificar o progresso do time e alertá-lo quando for necessário alterar os prazos ou balancear a tarefas.

Regras

As regras do *XP* são dividas em cinco categorias: Planejamento, Gerenciamento, Projeto, Codificação e Testes.

Planejamento

O planejamento é iniciado com a “história do usuário”, que serve para o mesmo propósito dos “casos de uso” (da UML), porém não é a mesma coisa. A história do usuário é escrita pelo cliente com cerca de três frases na linguagem dele próprio. Ele deverá conter, basicamente, o que o usuário faz, o que ele quer e os benefícios que a funcionalidade trará.

A história do usuário é parecida com casos de uso, porém ela não se limita a descrever a interface do usuário. Ela também é constantemente confundida com o levantamento de requisitos, porém ela deve ser bem mais simples do que este afinal esse documento servirá, basicamente, para planejar as datas de entrega da próxima versão do sistema, não contendo um grande nível de detalhes como algoritmos, especificação de tecnologia ou desenho do banco de dados.

Após escritas as histórias do usuário, é feita uma reunião de planejamento de versões que serve para discutir e planejar as versões de cada iteração. Dessa reunião devem participar tanto as pessoas competentes para a tomada de decisões técnicas (desenvolvedores) quanto às pessoas que possam tomar as decisões do negócio (clientes e analistas de negócio), enfim, todos envolvidos com o projeto e que podem tomar decisões que influenciarão no produto final devem participar.

Na reunião de planejamento da versão, os desenvolvedores irão aprazar em semanas ideais de programação todas as histórias de usuários (por ideais, entende-se que não haverá nada além dessas histórias para fazer, ou seja, uma equipe totalmente dedicada a isso). Já os clientes irão determinar tudo o que é mais importante ou prioritário para ele. Após essa reunião, o projeto deverá ter seus prazos baseados em quatro variáveis: escopo (quanto será feito), recursos (quantas pessoas estão disponíveis), tempo (quando o projeto ou versão será entregue) e

qualidade (quanto planejamento será feito para a codificação e quantidade de testes). A alteração de qualquer uma dessas variáveis influencia diretamente alguma outra.

Cada uma das versões, planejada durante a reunião, deve ser um pequeno pacote de funcionalidades entregue periodicamente (de uma a três semanas). Cada versão também deverá ter uma reunião antes de se iniciar para que os desenvolvedores possam verificar cada uma das tarefas que terão para desenvolver as histórias dos usuários, dessa forma as tarefas duplicadas podem ser removidas e cada um dos desenvolvedores pode selecionar a tarefa que deseja fazer e estimar o tempo que demorará a fazer, contando também com os testes de cada tarefa.

Essas reuniões também servem para uma melhor avaliação do tempo que será gasto para o desenvolvimento da versão inteira e se novas funcionalidades podem ser incluídas no pacote ou se alguma deve ser removida. Os prazos dessas versões devem ser estritamente respeitados e o progresso feito ao longo delas deve ser medido para quantificar o andamento do projeto, sempre que for identificado um prazo que não pode ser cumprido, deve-se voltar do início fazendo uma nova reunião de planejamento da versão.

Gerenciamento

A comunicação é muito importante para uma equipe de *XP*, simplesmente remover os obstáculos físicos entre os membros da equipe já é um grande avanço para melhorá-la, sendo o ideal colocar todos os colaboradores em uma área central, de forma que se vejam como iguais em valores e em contribuição para o projeto, colocar ainda algumas pessoas ao redor dessa área central permite que elas trabalhem mais concentradas, porém sem se desintegrar da equipe.

Além das barreiras físicas, é ideal criar espaços para que as discussões sobre o projeto possam ocorrer com a participação de todos, ao menos, como ouvintes e também locais em que possa haver reuniões diárias (preferencialmente em pé, para que sejam rápidas). Incluir quadros e paredes em que possam ser afixados avisos, regras ou detalhes sobre o projeto também é uma boa coisa para que todos tenham

fácil acesso a tais informações.

Essas reuniões diárias “em pé” são muito importantes para o desenvolvimento do projeto e conhecimento do andamento por todos os envolvidos, afinal a ideia é que toda a equipe participe, mesmo sendo apenas como ouvinte, para que sejam evitadas outras reuniões menores e isoladas, deixando todos com as informações necessárias para a continuidade do desenvolvimento. Nessas reuniões, deve ser abordado, ao menos, um resumo do que foi feito no dia anterior, o que será focado no dia atual e quais são os problemas que estão causando atrasos.

É importante criar um ritmo de trabalho estável na equipe para saber de fato o que se pode desenvolver e em quanto tempo, portanto, se algo não pode ser concluído na versão que está sendo desenvolvida, é mais interessante fazer uma nova reunião acerca da versão do que fazer horas extras ou envolver mais pessoas no projeto (o que, em tese, “aceleraria” o ritmo), afinal isso maquiará a produção da equipe, prejudicando futuras previsões, além de desmotivar a equipe obrigando-os a fazer um trabalho forçado ou de perder muito tempo colocando novos membros a par do que está sendo desenvolvido.

Sempre que uma versão do sistema for fechada, a “velocidade do projeto” deve ser medida. Esse indicador é baseado na soma dos tempos estimado de todas as histórias dos usuários desenvolvidas durante a versão em questão e também na soma do tempo estimado de cada uma das tarefas indicadas pelos desenvolvedores e servirá para auxiliar na determinação do escopo das próximas versões e, consequentemente, dará melhor estimativa do tempo total do projeto.

Na reunião de planejamento da próxima versão, os usuários poderão selecionar o mesmo número de histórias a serem desenvolvidas referentes às horas das histórias da versão anterior, assim como os desenvolvedores poderão se propor a realizarem o mesmo número de horas de tarefas que concluíram na última versão. Dessa forma os desenvolvedores terão certa folga para trabalhar caso a última versão tenha sido muito corrida e, caso terminem antes do prazo, poderá ser feita outra reunião para acrescentar mais histórias ainda na versão em questão.

Um conceito muito importante também do *XP* é a ideia de “movimentar” os integrantes da equipe, ou seja, não deixar alguém permanente com alguma parte do código ou do projeto, afinal isso fará com que aquela pessoa seja extremamente

conhecedora daquela parte e não das outras, assim como fará com que os outros integrantes não conheçam aquela parte, portanto o ideal é que todos participem em algum momento de todas as diferentes partes do projeto, dessa forma uma perda na equipe não será tão sentida e também não há o risco de alguém ficar sobrecarregado por só ele ter conhecimento de alguma determinada parte.

O *XP* deve também ser flexível, ou seja, as regras dele devem ser seguidas de início, porém, conforme o desenvolvimento e as características do projeto, ele pode ser adaptado com novas regras ou alterações de algumas outras para melhor se adequar, porém é interessante que as regras sejam debatidas com os membros da equipe e que todos tenham conhecimento delas para não haver nenhum desentendimento.

Projeto

O projeto do sistema deve sempre buscar a maior simplicidade possível, afinal um projeto simples é bem mais fácil de concluir do que um complexo, sempre que houver algo complexo, deve-se tentar tornar isso o mais simples possível, afinal é muito mais rápido e barato substituir algo complexo durante o projeto do que após a codificação já tiver sido feita. Levando em conta isso, não se deve pensar em soluções futuras, focando-se nos problemas atuais e sem adicionar funcionalidades desnecessárias, por mais que pareça que um dia o deixarão de ser. Tentar pensar em tudo no momento do projeto só o tornará mais complexo e caro.

Deve-se lembrar de que a simplicidade é algo subjetivo, portanto a equipe que está trabalhando deve se reunir para definir aquilo que é simples e o que não é. Algumas características subjetivas, recomendadas pelo *XP*, que devem ser analisadas para ajudar a decidir isso são:

- **Testável**: Deve ser possível desenvolver testes unitários para cada função, procedimento ou método escrito para verificar problemas, portanto é importante que cada um desses seja simples para que os testes também sejam.
- **Navegável**: Um sistema é considerável “navegável” quando qualquer

programador consegue encontrar o que deseja sempre que precisar, ou seja, as variáveis, classes, atributos, métodos, funções e procedimentos possuem nomes intuitivos e também são utilizadas outras técnicas de programação (herança, polimorfismo, etc.) que facilitam tal navegação.

- **Compreensível**: Uma equipe que já trabalha junto há algum tempo não deve ter dificuldades em reconhecer e entender uma parte do sistema desenvolvido por outro integrante, porém um código leva a qualidade de compreensível quando novos integrantes também não possuem essa dificuldade.
- **Explicável**: O projeto deve conseguir ser explicado a um novo integrante da equipe de forma simples e rápida.

Um dos métodos que auxilia na melhor compreensão do sistema é a criação de metáforas, preferencialmente com algo que todas as pessoas estão acostumadas a ver no dia a dia, por exemplo [colocar exemplo]. Há vezes em que não é possível encontrar uma metáfora coerente, e por isso sempre é bom reforçar as outras formas de facilitar a compreensão do sistema, como criar nomes coerentes para as classes e objetos.

Cartões CRC

Outro método utilizado são os cartões de classe, responsabilidade e colaboração (CRC *cards*) e as sessões CRC, utilizados para projetar o sistema como um time. Uma grande contribuição desse método é realizar a transição das pessoas acostumadas com a programação estruturada para a orientada a objetos. Cada cartão é utilizado para representar um objeto, responsabilidades são listadas no lado esquerdo do cartão e as classes colaborativas são listadas ao direito de cada responsabilidade.

Tendo todos os cartões devidamente escritos, a sessão CRC continua com um representante simulando o sistema com esses cartões demonstrando qual objeto envia mensagens e qual as recebe. Feito isso, as fragilidades do sistema são facilmente expostas e alternativas podem ser encontradas com maior velocidade. O cuidado que deve haver nessas reuniões é o de elas fugirem do controle com muitos cartões sendo movimentados de um lado para outro, além de pessoas falando e se

movimentando demais, para evitar isso se pode limitar a quantidade de pessoas na reunião ou até mesmo criar regras de convivência como “apenas uma pessoa deve ficar de pé por vez” ou “deve-se pedir a palavra antes de se manifestar”.

Um dos pontos importantes e também bastante criticado dessa técnica é a falta de documentação escrita, porém, segundo Wells (1999), ela torna-se desnecessária se for levada em conta a obviedade gerada pelos cartões e suas descrições.

Soluções de “pico”

É recomendado que sejam criadas soluções de pico, ou seja, programas simples que em muitos casos serão descartados, porém que ajudarão a encontrar problemas maiores em uma solução e também a mitigar os riscos de problemas técnicos e aumentar a confiança no sistema.

Refatoração

A refatoração é uma prática necessária para manter a simplicidade do projeto, manter o código limpo e conciso é uma necessidade para que ele seja fácil de entender, modificar e estender. Por melhor que um código tenha sido feito, chega o momento em que ele se torna obsoleto e deve ser refeito.

Codificação

Novamente é reforçada a presença do cliente em todas as fases do desenvolvimento do sistema, afinal, como não são dados muitos detalhes específicos durante a história do usuário é necessário que ele esteja presente para auxiliar na determinação das tarefas da programação.

Determinadas todas as tarefas, chega-se, finalmente, ao código. Este deve

seguir padrões determinados para que qualquer um da equipe possa manipulá-lo sem grandes dificuldades, ou seja, para que ele seja uma propriedade coletiva, isso faz com que todos se sintam à vontade para contribuir com novas ideias, sejam elas para qualquer parte do projeto, além de facilitar para qualquer desenvolvedor corrigir bugs ou fazer a refatoração no projeto, além disso, há o fato de ser muito mais seguro confiar o projeto do sistema e suas particularidades a diversos componentes da equipe em vez de apenas a um membro, afinal dificilmente alguém conseguirá lembrar-se de todas as partes e também não se corre o risco de um membro da equipe sair e todo o conhecimento do projeto do sistema acompanhá-lo.

Testes unitários

Antes de codificar o projeto em si, uma boa prática aconselhada pelo *XP*, é que primeiramente sejam codificados os testes unitários. Segundo Wells (1999), fica muito mais fácil e rápido codificar o projeto além de ajudar o desenvolvedor a visualizar o que realmente deve ser feito por aquilo que será codificado. Quando os códigos de testes já estão prontos, não é necessário esperar mais nenhuma etapa também após a conclusão da codificação, podendo testá-la logo após. Além disso, é possível verificar se realmente todas as funcionalidades esperadas estão contempladas pelo código e somente elas.

Programação em par

Uma boa prática que é muito contestada e difícil de implementar em uma empresa pela justificativa do custo é a da programação em par. Todo o código é desenvolvido por duas pessoas ao mesmo tempo em um mesmo computador, essa prática faz com que a qualidade do software aumente e, apesar de parecer um contrassenso, torna o desenvolvimento mais rápido se comparado a dois programadores desenvolvendo em máquinas separadas.

Apesar de parecer simples, a programação em par exige uma grande habilidade social, afinal os dois evolvidos devem lidar com suas vaidades e

aceitarem sugestões assim como aceitarem deixar de lado suas convicções em alguns momentos. Deve-se estar claro também que a programação em par não serve como uma relação de professor-aluno, pelo contrário, deve ser realizada por pessoas em níveis semelhantes de conhecimento.

Laurie Williams, uma professora do departamento de ciências da computação, da faculdade de engenharia, integrante da Universidade do Estado da Carolina do Norte, nos EUA, fez um estudo com 42 programadores sênior na Universidade de Utah, nos EUA. O estudo consistiu em separar 14 programadores para trabalharem sozinhos e fazer ou outros 28 trabalharem em pares, sendo que todos deveriam cumprir as mesmas tarefas.

Na primeira tarefa, os pares gastaram 60% mais horas por programador que os programadores individuais, na segunda 20% e na terceira 10%, ou seja, se um programador demorou 10 horas para desenvolver algo, o par demorou 5 horas e 15 minutos. Em todos os casos, os sistemas desenvolvidos pelos pares passaram em 15% mais testes que os desenvolvidos individualmente, além disso, 90% disseram ter gostado da programação em par e também que se sentiram mais confiantes no trabalho que realizaram.

Integração de códigos

O código de um par de programadores pode funcionar muito bem, porém quando ele é combinado com outros códigos, criados por outros desenvolvedores, verdadeiras catástrofes podem acontecer e é por isso que a integração de códigos é muito importante sempre que há um desenvolvimento paralelo.

Para isso, o *XP* aconselha que os pares utilizem turnos para a integração de seus códigos com o código final, ou seja, cada par terá sua vez de integrar suas alterações e homologá-las, além de, é claro, utilizar uma ferramenta para gerenciar as versões dos códigos. Fazer com que essa integração seja frequente, faz com que elas sejam menores e mais rápidas, além de um par não alocar uma classe, ou um código qualquer por muito tempo.

Testes

A parte de testes fica mais simples a partir do momento em que todas as regras anteriores foram seguidas, pois todos os testes unitários já estarão prontos e o código terá uma ótima qualidade homologada pela programação em par. Para o código, é necessário apenas que se tome o cuidado de testar todos os testes unitários e de, sempre que uma falha for encontrada, sejam criados novos testes para evitar que se repita.

Para verificar se as “histórias dos usuários” estão de acordo com o que foi desenvolvido, há os testes de aceitação. O próprio cliente ditará os cenários que demonstrarão que sua história foi implementada corretamente, podendo ser criados diversos testes de aceitação ou apenas um, desde que seja possível verificar se a funcionalidade está de acordo com o que era esperado em sua plenitude. Esses testes devem ser automatizados ao máximo, de forma que possam ser refeitos sempre que necessário sem uma grande necessidade de interação.

O resultado desses testes de aceitação deve ser verificado pelo cliente que dirá se está de acordo ou não e também determinará a prioridade de correção dos que não estiverem.

Todos os testes citados, segundo Wells (1999), podem ser feitos pela própria equipe de desenvolvimento ou por uma equipe separada especializada na garantia da qualidade do sistema, ainda assim é recomendável que ambas as equipes possuam um relacionamento próximo.

Feature Driven Development (FDD)

História

O FDD (Desenvolvimento dirigido a funcionalidade) foi criado ao longo de um projeto, em 1997, de desenvolvimento de software em Singapura que contou com Jeff de Luca como gerente de projeto, Peter Coad como arquiteto chefe e Stephen Palmer como gerente de desenvolvimento.

Era um projeto ambicioso, porém complexo e que já havia falhado uma vez, o que deixou os usuários céticos e uma equipe de desenvolvimento desmoralizada e frustrada. A partir do desenvolvimento e utilização dos processos desta metodologia no projeto, os funcionários passaram a se divertir com o que estavam aplicados e o trabalho passou a fluir muito melhor.

Abordagem

Segundo Palmer e Felsing (2002), muitos dos processos existentes até a época focavam em documentações e outras coisas que faziam as pessoas se importarem muito mais em seguir os processos descritos na metodologia utilizada do que realmente no desenvolvimento do software. Para ele, um desenvolvimento bem feito não deve ter diversos processos que tem que ser consultados diversas vezes durante todo o processo, mas sim apenas alguns que consigam ser decorados e, assim, enraizados no desenvolvedor, além de **muita** comunicação.

O FDD foi pensado para projetos de software de tamanho moderado a grande e ele tem como foco uma “característica” que, neste caso, é uma função valorizada pelo cliente e que pode ser implementada em uma semana ou menos. Essas características são pequenos blocos de funcionalidades que serão entregues periodicamente (em iterações), sendo assim, é mais fácil para os usuários descrevê-las, revisá-las e relacioná-las. Como se tratam de pequenos blocos, os projetos e codificações serão mais simples e fáceis de monitorar.

Tendo todas as características listadas (ou boa parte delas) é possível organizá-las hierarquicamente conforme a relevância e tamanho para o negócio, dessa forma todo o planejamento e cronograma serão guiados por essa hierarquia.

A “fórmula” para definir uma característica é a seguinte:

<ação> o <resultado> <por | para | de | a> um <objeto>

Com essa fórmula podem-se gerar exemplos como:

- Internar o paciente em um leito.
- Agendar a consulta para um profissional.

Papéis

Para melhor definir as atribuições de cada integrante de um projeto, o FDD descreve seis papéis principais (gerente de projetos, arquiteto chefe, gerente de desenvolvimento, programadores chefes, donos de classes, especialistas de domínio) seis papéis de apoio (gerente de domínio, gerente de versão, “guru” da linguagem, engenheiro construtor, criador de ferramentas, administrador de sistemas) e três papéis adicionais (testador, implantador e escritor técnico).

Gerente de Projetos

É ele o responsável por gerenciar o projeto, seus recursos, seus prazos, as tecnologias utilizadas. É ele também que deve sofrer todas as interações externas (cobranças do cliente ou da diretoria, por exemplo) blindando todo o restante da equipe envolvida no desenvolvimento do projeto. Basicamente, pode-se dizer que ele é o responsável por permitir que o trabalho de todos outros envolvidos flua sem maiores problemas e que o projeto seja entregue corretamente.

Arquiteto Chefe

O arquiteto chefe é o responsável pelo projeto geral do sistema, ou seja, ele que dará a palavra final sempre que houver qualquer discussão acerca dos modelos a serem utilizados para o desenvolvimento, servindo muitas vezes como um intermediador entre os programadores chefes.

Gerente de Desenvolvimento

Cabe ao gerente de desenvolvimento fazer com que o desenvolvimento em si tenha um andamento normal, sem interrupções. A pessoa que estiver nesse papel

deve ter boas habilidades de desenvolvimento e deve resolver qualquer conflito de recursos que surja entre os programadores chefes no dia-a-dia.

Programadores Chefes

São desenvolvedores mais experientes que já participaram de alguns ciclos completos de desenvolvimento de software. Eles lideram times pequenos de desenvolvedores e atuam na análise e projeto de atividades mais complexas. Cabe a eles também entenderem-se com os outros programadores chefes em conflitos técnicos ou por recursos.

Donos de Classes

Dentro dos times liderados pelos programadores chefes, são as figuras mais importantes (após, é claro, o próprio programador chefe), pois são responsáveis por projetar, codificar, testar, documentar e consolidar tudo que for feito em suas determinadas classes.

Especialistas de Domínio

Podendo ser representado por clientes, analistas de negócio, ou por uma mistura deles, o especialista de domínio é, como o nome sugere, alguém que possui muitos conhecimentos no negócio para o qual o sistema será desenvolvido ou sobre algum negócio com o qual o sistema deverá se relacionar.

Gerente de versão

Basicamente, o gerente da versão é o responsável por uma versão, ou

iteração, que está sendo desenvolvido, monitorando o trabalho dos programadores chefes e verificando se será entregue tudo o que foi planejado. Ele trabalha em conjunto com o gerente de projetos, relatando a ele como está o desenvolvimento da versão.

"Guru" da linguagem

O "guru" da linguagem, como o nome sugere, é alguém que saiba muito ou tudo sobre uma linguagem ou tecnologia específica que será utilizada no projeto, esse papel é especialmente importante nos casos em que será utilizada uma linguagem ou tecnologia pela primeira vez, tornando-se não tão importante nos casos em que a maior parte da equipe (ou ela toda) já conhece a tecnologia em questão.

Engenheiro Construtor

O engenheiro construtor deve preparar tudo o que for necessário para que a próxima versão possa funcionar sem problemas (scripts preparatórios, geração de documentação ou relatórios) além de garantir que a versão do sistema esteja íntegra (controlar a versão para que nada seja perdido durante as consolidações e que as novas alterações sejam consolidadas corretamente).

Criador de ferramentas

Cria sistemas e ferramentas para auxiliarem no desenvolvimento do projeto, seja para o próprio desenvolvimento ou para a fase de testes ou conversão de dados.

Administrador do Sistema

Configura e gerencia o ambiente de rede dos desenvolvedores e de testes, garantindo que o trabalho deles possa ocorrer sem problemas de infraestrutura.

Testador

O testador é o responsável por verificar se o que foi desenvolvido está de acordo com o que foi solicitado e se atende plenamente a necessidade do cliente (assim como nas metodologias tradicionais e nas outras ágeis).

Implantador

Faz a conversão necessária de dados para que o banco de dados antigo fique compatível com o novo sistema. Enfim, faz a atualização do ambiente para que o novo sistema possa rodar sem maiores problemas.

Escritor técnico

Escreve e prepara toda a documentação impressa e digital para o usuário.

Regras

Desenvolver um modelo geral

Esse o primeiro passo de modelagem do sistema que será desenvolvido (apesar de, em alguns casos, no momento em questão já terem sido buscadas justificativas e protótipos para conseguir aval para a realização do projeto). Como produto deste passo, deve haver um modelo geral do projeto que contenha as classes principais, suas descrições e responsabilidades, além das relações entre elas, deve haver também diagramas de sequência de alto nível e um conjunto de anotações que expliquem e analisem as decisões tomadas no modelo em questão.

Para que as metas desse passo sejam alcançadas, há algumas recomendações feitas pelos autores da FDD:

- Envolver pessoas da análise e do desenvolvimento;
- Aplicar diversas cabeças a um problema concorrentemente;
- Padronização rápida no uso de terminologias;
- Identificação rápida de suposições, preconceitos e omissões dos membros principais dos times;
- Compreensão total das propostas e metas do projeto;
- Um conjunto compartilhado, entre todos envolvidos, dos requisitos detalhados e aberto a discussões.

Além dessas recomendações, esta metodologia aconselha detalhadamente como tudo deve ser feito, desde o ambiente em que devem trabalhar os programadores (formato da sala, leiaute das mesas, materiais utilizados) até a forma que deve ser desenhado o diagrama de classes.

Como participantes deste passo, é criado um time de modelagem do qual devem participar o gerente de projeto, o arquiteto chefe, os programadores chefes, os especialistas de domínio e alguns dos melhores desenvolvedores.

Construir uma lista de funcionalidades

Desenvolvido o modelo geral inicial do projeto, deve-se criar uma lista de funcionalidades nas quais se focarão todos os próximos passos do desenvolvimento das versões das iterações. Como resultado deste passo, devemos ter uma lista de funcionalidades hierarquizadas e ordenadas no formato:

<ação> <resultado> <objeto>

Para melhor organizar essas funcionalidades, elas são agrupadas em um conjunto de funcionalidades (também chamado de “atividade”) e esses conjuntos são agrupados em outro conjunto superior, chamado de “área”.

Deste passo, participam o gerente de projetos, o gerente de desenvolvimento e, sempre que possível, o arquiteto chefe.

Planejar por funcionalidade

Após ter o modelo geral do projeto e todas as funcionalidades que se espera desenvolver, deve-se planejar em quais versões cada funcionalidade será entregue e quanto tempo será reservado para o desenvolvimento de cada versão.

Projetar por funcionalidade

Para projetar por funcionalidade, devem ser selecionadas pelos chefes dos programadores as funcionalidades que serão desenvolvidas. A partir disso serão estudados os documentos de referência, desenvolvidos os diagramas de sequência, refinados os modelos que serão utilizados e então escritas as classes e métodos.

Como resultado deste passo, estarão prontas todas as classes e métodos que serão utilizados para o desenvolvimento das funcionalidades escolhidas pelos programadores chefes.

Desenvolver por funcionalidade

Finalmente, no desenvolvimento, o código e os testes unitários são produzidos, inspecionados, testados e então, enfim, as funcionalidades são integradas ao sistema.

Lean Software Development

História

A “*Lean Software Development*” (Desenvolvimento “Magro” de Software) tem sua origem, como o nome sugere, nas “*lean techniques*” (técnicas “magras”) do ambiente industrial sendo aplicadas ao desenvolvimento de software.

Mary Poppendieck, que idealizou a metodologia, juntamente com Tom Poppendieck, foi programadora por diversos anos, até se deparar com uma oportunidade e seguir a carreira de gestora de TI. Após alguns anos, ao voltar ao cenário de desenvolvimento de software, deparou-se com algo que considerou assustador: uma ênfase em definição e detalhamento de processos sendo justificada pelo movimento de manufatura “magra” (o qual ela acompanhou na própria indústria).

Como sabia que as técnicas que estavam sendo aplicadas eram exatamente o oposto do que a manufatura “magra” pregava (“um fluxo rápido e essencial”, “as pessoas devem pensar mais para fazerem seus trabalhos”, “testar incessantemente é a melhor forma de fazer as coisas funcionarem corretamente”), Mary resolveu tentar mudar o paradigma do desenvolvimento de software escrevendo um livro que explica a forma correta de aplicar as técnicas “magras” ao desenvolvimento de software.

Abordagem

O desenvolvimento de software é algo muito abrangente, indo desde “Web Design” até o envio de um satélite à órbita, e sendo assim tão abrangente e variável, há diversas práticas que podem ser aplicadas a alguns casos e a outros não. Esta metodologia cuida de aplicar as práticas corretas para cada caso levando em conta os princípios “magros”.

Há 22 ferramentas essenciais a um líder de projetos segundo Poppendieck e Poppendieck (2003):

- Verificação de desperdícios;
- Mapeamento de valor “*stream*”;
- *Feedback*;
- Iterações;
- Sincronização;
- Desenvolvimento baseado em “peça”;
- Pensamento nas opções;
- Último momento responsável;
- Tomada de decisões;
- Sistemas “*Pull*”
- Teoria das filas;
- Custo de atraso;
- Autodeterminação;
- Motivação;
- Liderança;
- *Expertise*;
- Integridade “*perceived*”;
- Testes;

- *Refactoring*;
- Medições;
- Contratos;

Princípios

Conforme já foi dito, a *Lean Software Development* teve sua origem na manufatura magra e, por isso, herdou alguns de seus princípios e os adaptou. Os sete princípios fundamentais dessa metodologia são:

- **Eliminar desperdícios:** entende-se por desperdício tudo que não agrega valor a um produto, se há algo ou alguma funcionalidade em um projeto que não é imediatamente necessário é considerado um desperdício. O ideal é entender exatamente a necessidade do cliente e então desenvolver e entregar exatamente aquilo, nada a mais, nada a menos, tudo o que entra no caminho de uma satisfação rápida do cliente é **desperdício**.
- **Ampliar os aprendizados:** desenvolvimento é um exercício de descoberta, já produção é um exercício de redução de variações, portanto a aplicação da abordagem "*lean*" em um cenário de desenvolvimento é completamente divergente da aplicação em um cenário de produção. Desenvolvimento é comparável a criar uma receita, enquanto produção é como fazer um bolo. As receitas são feitas por chefs experientes que já desenvolveram um instinto para o que funciona e a capacidade de adaptar os ingredientes disponíveis para a ocasião. O desenvolvimento de software também é um processo de aprendizado, porém com desafios maiores como grandes equipes e resultados mais complexos que de uma receita. A melhor abordagem para melhorar um desenvolvimento de software é ampliar o aprendizado.
- **Decida o mais tarde possível:** esse princípio é muito efetivo em ambientes em que há a incerteza, pois ele oferece uma abordagem baseada em opções. Dessa forma, conforme o tempo passa, há mais fatos, o que torna mais certa alguma das opções pensada no início, podendo ter uma decisão mais correta. A principal estratégia utilizada em sistemas complexos para postergar decisões é fazer uma capacidade de mudança no sistema.
- **Entregue o mais rápido possível:** quanto mais rápido se entrega algo ao cliente, mais rápido saber-se-á se foi feito de acordo com sua necessidade e expectativa e mais rápido arrumar-se-á caso não atenda o desejado. Tudo isso faz com que não se percorram diversas etapas fazendo coisas que o

cliente não queria realmente, ou que não era, de alguma forma, o que ele queria. O desenvolvedor conseguirá ter um retorno mais confiável e mais rápido, além de não entregar amanhã a necessidade do cliente de ontem (e que não é mais).

- **Fortaleça o time:** a execução de excelência depende da compreensão ideal dos detalhes, e ninguém pode entender os detalhes melhor que os que executarão o trabalho. Quando guiados por um bom líder e tendo o conhecimento necessário, os desenvolvedores poderão tomar decisões técnicas melhores do que as de qualquer outro.
- **Torne-o íntegro:** um sistema pode ser considerado íntegro quando o usuário recebe exatamente aquilo que esperava. Para um sistema realmente ser considerado íntegro ele deve manter-se completamente funcional durante muito tempo, adaptando-se às mudanças futuras. O software íntegro deve possuir uma arquitetura coerente e possuir uma ótima usabilidade, além de conseguir receber manutenções, ser adaptável e extensível.
- **Veja o "todo":** Para construir um sistema complexo íntegro, é necessário o conhecimento de especialistas em diversas áreas, o problema é que esses diversos especialistas tendem a aumentar o desempenho das partes do produto que lhes compete, em vez de focar no desempenho geral do sistema, sendo assim, é normal que o bem comum acabe sofrendo as consequências. É importante que os envolvidos em um projeto possam ver e pensar no todo em vez de ficarem focados em suas áreas.

Crystal Family

História

A família “Crystal” foi desenvolvida a partir de uma necessidade da IBM em 1991. Ela precisava de uma metodologia “*object-technology*” e para isso contratou Alistair Cockburn como seu desenvolvedor.

Cockburn iniciou o processo de desenvolvimento da metodologia a partir de entrevistas com times de projetos e percebeu que as práticas utilizadas eram bem diferentes das descritas nos livros da época, como o uso excessivo de comunicação próxima e direta, moral, diálogo com o usuário final, etc. Pelo que pode perceber, esses aspectos diferenciavam os projetos que obtinham sucesso dos que não.

Utilizando-se dessas técnicas, Cockburn foi consultor de um projeto com preço e escopo fixo, e as teve como principal fator de sucesso. Partindo das lições aprendidas com a experiência deste e dos projetos que participou posteriormente, Cockburn escreveu um livro, em 2004, descrevendo a família “Crystal”.

Aspectos

"Crystal" é uma família de metodologias com um código genético comum, que tem ênfase em entrega frequente, comunicação próxima e melhoria recíproca. Deve-se lembrar de que "Crystal" não é simplesmente uma metodologia, mas sim uma família, portanto existe uma metodologia diferente para cada tipo de projeto, sempre utilizando o "código genético" da família.

O nome "Crystal" parte de duas dimensões dos projetos: tamanho e criticidade, correspondendo, respectivamente, as propriedades cor e rigidez do mineral (cristal). Projetos maiores, que exigem maior coordenação e comunicação, correspondem a cores mais escuras (branco, amarelo, laranja, vermelho e assim por diante). Projetos para sistemas que possuem um maior número de regras de validação e verificações correspondem a cristais mais rígidos. A metodologia quartzo é utilizada para sistemas inofensivos com poucos desenvolvedores, já um mesmo time criando um sistema para um reator nuclear exige uma metodologia diamante, demandando assim por verificações repetitivas tanto no design quanto na implementação dos algoritmos.

Cada projeto poderá, então, utilizar uma metodologia de cor ou rigidez diferente dependendo de seu tamanho e criticidade, porém todos, conforme já mencionado, carregam o código genético da linguagem, que engloba:

O modelo de jogo da economia cooperativa;

Segundo Cockburn (2004), há dois objetivos para esse "jogo" que competem pelos mesmos recursos: entregar o software e preparar-se para o próximo jogo. Como o jogo nunca se repete, cada projeto exige estratégias ligeiramente diferentes dos jogos anteriores. Esse "jogo" faz com que os envolvidos pensem em seus trabalhos de uma forma muito específica e focada.

- Prioridades selecionadas;
    - Segurança no resultado do projeto;
    - Eficiência no desenvolvimento;
    - Capacidade de convivência com as convenções.
- Propriedades selecionadas;

    - Entrega frequente;
    - Melhora “refletiva”;
    - Comunicação próxima;
    - Segurança pessoal;
    - Foco;
    - Acesso fácil a usuários experientes;
    - Ambiente técnico com testes automatizados, gerenciamento de configurações e integração frequente.
- Princípios selecionados;
    - A quantidade de detalhes necessários nos requisitos, desenho e documentação do planejamento varia de acordo com as circunstâncias do projeto, levando em conta o dano que pode ser causado por um erro desconhecido no sistema e a frequência de colaboração pessoal utilizado pela equipe.
    - Muito provavelmente não seja possível eliminar todos os intermediários entre o início do desenvolvimento e o sistema, porém eles devem ser reduzidos para que contenham apenas as informações que realmente agregam, de forma que a equipe possua.
    - A convenção de trabalho da equipe deve ajustar-se continuamente para que fique compatível com as personalidades dos integrantes.
- Técnicas de amostras selecionadas;
- Exemplos de projetos.

Crystal Clear

A “*Crystal Clear*” é a variação mais conhecida da família Crystal, trata-se de um aperfeiçoamento que pode ser aplicado em equipes que contenham de 2 a 8 pessoas, instaladas em uma mesma sala ou escritórios adjacentes. A comunicação próxima é reforçada, ocorrendo de uma forma natural e quase imperceptível, afinal os integrantes acabam participando como ouvintes de discussões que não fariam parte.

Comparações

Para fazer as comparações entre as metodologias ágeis, serão utilizadas apenas as que são mais utilizadas e que, consequentemente, foram mais bem detalhadas neste trabalho.

Metodologia	Ano do surgimento
XP	1999
Scrum	1995
FDD	1997

Metodologia	Motivo de desenvolvimento
XP	Utilização para administrar um projeto
Scrum	Utilização para administrar projetos de duas empresas
FDD	Utilização para administrar um projeto

Metodologia	Definição dos requisitos
XP	Um esboço é feito no momento inicial do planejamento, por meio das "histórias do usuário" que, conforme já descrito, é mais simples do que o levantamento de requisitos das metodologias tradicionais, porém mais complexas do que os casos de uso que se limitam a descrever a interface dos usuários.
Scrum	Todos os requisitos e funcionalidades conhecidos são descritos no momento inicial do planejamento formando assim o *Backlog* do produto. A cada reunião de revisão do *Sprint* o *Backlog* do produto é revisado.
FDD	No início do planejamento, são listados e documentados os requisitos por meio de casos de uso ou histórias dos usuários (como no *XP*), além disso, é ideal que sejam construídos diagramas de classe e de sequência UML para maior compreensão do projeto.

Metodologia	Implementações dos requisitos
XP	A cada nova versão a ser entregue é feita uma reunião de planejamento dela na qual são determinadas as histórias de usuário que serão implementadas e em quanto tempo o serão.
Scrum	Cada *Sprint* possui uma versão que é desenvolvida até o seu final, nessa versão estão contidos todos os itens do *Backlog* do *Sprint*, o qual foi gerado com itens do B*acklog* do produto mais bem detalhados durante a reunião de planejamento do *Sprint*.
FDD	Com as características já definidas durante o planejamento inicial, estas são agrupadas e ordenadas em relação à importância e dependência, dessa forma são determinados conjuntos de características que serão implementadas a cada versão.

Metodologia	Validação das versões
XP	Conforme as regras da metodologia, a validação tem sua primeira fase antes mesmo da versão ser codificada, afinal a primeira fase do desenvolvimento é a codificação dos testes unitários, os quais serão utilizados para validar o código após o desenvolvimento. Finalizado o desenvolvimento, há os testes de aceitação, que são desenvolvidos em conjunto com o cliente para validar se as histórias foram implementadas corretamente.
Scrum	Não há métodos especificados para fazer a validação, apenas sugere-se que ela seja feita na fase final do *Sprint*.
FDD	Ao fim do desenvolvimento de cada versão, os próprios programadores executam testes unitários para verificar se todas as características foram implementadas corretamente.

Metodologia	Integração da versão ao sistema
XP	É ideal que toda a alteração de código a ser integrada ao sistema ocorra de forma rápida e seja a menor possível, para que isso possa acontecer, essas integrações devem ocorrer com uma frequência alta.
Scrum	Não há métodos especificados para fazer a validação, apenas sugere-se que ela seja feita na fase final do *Sprint*.
FDD	Após a validação ser realizada, é feita a integração do conjunto de características desenvolvidas na versão em questão.

Metodologia	Validação do sistema
XP	Não há um momento ou métodos especificados para a validação do sistema como um todo, portanto isso ocorrerá, basicamente, quando o sistema for disponibilizado para o cliente.
Scrum	Não há um momento ou métodos especificados para a validação do sistema como um todo, portanto deve-se entender que isso será feito no momento da implementação do sistema com a nova versão ao cliente.
FDD	Feita a integração da versão ao sistema, são realizados novamente todos os testes unitários do sistema para verificar se ainda está tudo funcionando corretamente.

Metodologia	Finalização do projeto
XP	O projeto pode ser considerado finalizado no momento em que o cliente estiver de fato satisfeito e não houver mais funcionalidades a serem acrescentadas
Scrum	Assim que todos os itens do *Backlog* do produto estiverem "prontos", pode-se concluir que o projeto está finalizado.
FDD	Quando o sistema já estiver com todas as características que foram listadas implementadas, pode-se concluir que o projeto está finalizado.

Casos de sucesso

Há diversos casos de sucesso com o uso de metodologias ágeis relatados, para exemplificação neste trabalho foi utilizado um caso de uma empresa nacional e um caso de uma empresa internacional.

Dextra

Uma empresa brasileira, a Dextra, relatou, em 2010, ter alcançado cem mil horas de projetos com metodologias ágeis após ter sido a pioneira na adoção do *Scrum* no Brasil e afirma ter obtido resultados impressionantes. Dentre esses projetos, destacam-se sistemas para a GLOBOSAT, a Força Aérea Brasileira e o Grupo Confidence, nos quais a empresa afirma ter obtido sucesso e satisfação totais de seus clientes.

FBI

Na comunidade ágil, há um caso famoso de um projeto que foi “salvo” pelas metodologias ágeis, o projeto um sistema de gerenciamento de casos (“Sentinela” - o qual passou a ser desenvolvido após os atentados da cidade de Oklahoma). Houve um projeto desenvolvido, inicialmente, com metodologias tradicionais, o qual foi cancelado em 2005 com um orçamento de cento e setenta milhões de dólares. Após esse fracasso, foi feita uma nova tentativa utilizando ainda a metodologia em cascata, agora com o orçamento de quatrocentos e vinte e cinco milhões de dólares, em 2010 foi verificado que o orçamento estouraria em trezentos e trinta e um milhões de dólares e o prazo aumentaria mais seis anos, ainda com a possibilidade de o sistema não estar completo.

Nesse momento, Chad Fulgham, o *CIO*, decidiu pausar o projeto e passar a utilizar o *Scrum*. Para isso ele reduziu a equipe de desenvolvimento para cinquenta e cinco pessoas (antes havia cento e vinte e cinco) e organizou todo o trabalho em

“histórias de usuários” e determinou a quantidade dos *Sprints* que seriam realizados e a duração de cada um. Como resultado dessa mudança, o projeto foi entregue e disponibilizado para todos os funcionários do FBI em primeiro de julho de dois mil e doze, conforme memorando publicado no site do FBI.

Casos de fracasso

A maior parte dos casos de fracasso que é possível encontrar nos sites e livros é relatada por defensores das metodologias ágeis, ou seja, em todos esses casos há alguma falha na utilização dos métodos por parte dos fracassados, ou algum tipo de resistência apresentada pela equipe. Ainda assim, esses casos podem, ao menos, deixar claros os erros mais comuns que ocorrem e dá uma ideia de como impedi-los de acontecerem.

Bless – Projeto Canon

Essa empresa foi extremamente ambiciosa no *Projeto Canon*, tendo sido criada no mês de Julho, comprometeu-se com a entrega completa de um site totalmente funcional para o dia trinta de setembro. Após um dia de apresentação acerca da metodologia *XP* por Laurent Bossavit (autor do caso) a empresa decidiu utilizar a metodologia para parte do projeto. Inicialmente, essa iniciativa mostrou-se efetiva, permitindo até mesmo que algumas iterações fossem entregues sem grandes problemas, porém os verdadeiros obstáculos apareceram quando foi necessário expandir o processo para o restante do projeto após serem verificados diversos problemas de qualidade nessas outras partes, além do fato de o cliente não estar ciente de tudo isso que estava acontecendo (as diferenças nos processos de trabalho entre as equipes e a consequente falha na qualidade e nos prazos).

Basicamente, o autor resume o fracasso desse projeto em três características que conflitaram, em parte, com os princípios do *XP*: paixão, ousadia e glamour. A paixão fez com que acreditassem poder desenvolver projetos grandes para grandes companhias. A ousadia fez com que aceitassem um projeto muito complexo em um prazo extremamente curto e irreal. Finalmente o glamour fez com que dessem mais atenção a “perfumarias” em vez de realmente se preocuparem com a estrutura interna do sistema, além de fazer com que deixassem a coragem (um dos valores pregados pelo *XP*) de lado para acomodarem-se em linguagens e sistemas de gerenciamento de banco de dados tradicionais em vez de realmente procurarem o que melhor os atenderia para o projeto em questão. Ainda assim, o que acabou

fazendo com que o fracasso fosse iminente foi a má administração da empresa, que acabou deixando um pouco de lado a opinião e o pensar dos colaboradores, para tentar resolver tudo “de cima” (algo não recomendado por todas as metodologias ágeis).

Conclusão

A intenção deste trabalho foi demonstrar algumas das diversas metodologias ágeis que já surgiram e os diversos processos que elas propõem não apenas para administrar um projeto, mas sim para toda uma forma diferente de se trabalhar e valorizar o profissional.

O *XP* é uma das metodologias ágeis mais completas se considerarmos todas as linhas que suas regras abrangem, mencionando desde o relacionamento com o cliente, até o desenvolvimento do sistema e sua entrega, o *Scrum* não possui tantos detalhamentos como o *XP* (não especifica processos para a codificação em si, por exemplo), porém passou a ser uma das metodologias mais utilizadas, afinal suas regras para a administração de um projeto são bem definidas e por não terem especificações tão “fortes” em algumas áreas (como é o caso da mencionada codificação) enfrenta menos resistência para ser implantada, além de poder ser utilizada com mais facilidade em qualquer tipo de projeto (não apenas em desenvolvimentos de sistemas), enfim, o *Scrum* pode servir como a porta de entrada menos traumática para as metodologias ágeis. Assim como o *XP*, o *FDD* também é muito abrangente em suas regras, detalhando minunciosamente cada etapa do desenvolvimento do projeto e, portanto, pode passar pelos mesmos problemas do *XP*.

Apesar de terem muito em comum, não há uma metodologia ágil que seja boa para todos, aqueles que desejam utilizá-las devem estudar o cenário e as culturas em que serão implantadas e então tentar identificar qual melhor se encaixa. É importante ressaltar também que não necessariamente devem-se seguir todas as regras de apenas uma metodologia ágil, podendo, como acontece muito com o *Scrum*, mesclar regras e princípios de diversas metodologias ágeis, conforme o que melhor se adaptar para o ambiente em que for implantado.

Se compararmos com o tempo que utilizamos as metodologias tradicionais e o surgimento das metodologias ágeis, realmente elas ainda possuem um longo caminho a percorrer antes de dar um veredito sobre a sua real efetividade, porém devem ser observados os resultados (assertividade de prazos, satisfação do cliente, satisfação dos colaboradores) que as empresas que as utilizam relatam e também a

quantidade de pessoas e empresas que estão seguindo essa tendência.

É importante lembrar que já houve uma quebra de paradigma em outro momento no mundo do desenvolvimento de sistemas no momento em que se passou a utilizar mais a orientação a objeto do que a programação estruturada, e se deve lembrar que essa existia desde meados dos anos 60, e só tornou-se de fato uma tendência a ser seguida no início dos anos 90.

Bibliografia

BECK, Kent et al. **Manifesto para Desenvolvimento Ágil de Software.** 2001. Disponível em: http://www.agilemanifesto.org/iso/ptbr/ Acesso em: 07 de setembro de 2012.

BOSSAVIT, Laurent. The Unbearable Lightness of Programming: a tale of two cultures. **Cutter IT Journal,** Massachusetts, v.15, n.9, 2002. Disponível em: http://cf.agilealliance.org/articles/system/article/file/1052/file.pdf. Acesso em: 18 de julho de 2012.

COCKBURN, Alistair. **Crystal Clear:** A Human-Powered Method Small Teams. New Jersey: Addison Wesley, 2004.

KOSCIANSKI, André; SOARES, Michel dos Santos. Capítulo 10: Metodologias ágeis. In: **Qualidade de Software:** Aprenda as metodologias e técnicas mais modernas para o desenvolvimento de software. 1 ed. São Paulo: Novatec, 2006.

PALMER, Stephen R.; FELSING, John M. **A Practical Guide to Feature-Driven Development.** New Jersey: Prentice Hall PTR, 2002.

POPPENDIECK, Mary; POPPENDIECK, Tom. **Lean Software Development:** An Agile Toolkit. New Jersey: Addison Wesley, 2003.

PRESSMAN, Roger. Capítulo 4: Desenvolvimento Ágil. In: **Engenharia de Software.** 6 ed. São Paulo: McGraw Hill Interamericana, 2006.

SÃO PAULO. Dextra. **Dextra alcança 100 mil horas de projetos com metodologias ágeis com resultados impressionantes**, 2012. Disponível em: http://www.dextra.com.br/noticias/Dextra-alcanca-100-mil-horas-de-projetos-com-metodologias-ageis-com-resultados-impressionantes.htm. Acesso em: 12 de outubro de 2012.

SÃO PAULO. USP. **Cooperativa de Desenvolvimento Ágil de Software – AgilCoop.** Instituto de Matemática e Estatística (IME), 2012. Disponível em: http://ccsl.ime.usp.br/agilcoop/casos_de_sucesso. Acesso em: 10 de setembro de 2012.

SCHWABER, Ken; SUTHERLAND, Jeff. **The Scrum Guide –** The Definitive Guide to Scrum: The rules of the game. Estados Unidos, 2011. Disponível em:

http://www.scrum.org/Portals/0/Documents/Scrum%20Guides/Scrum_Guide.pdf. Acesso em 25 de outubro de 2012.

WASHINGTON DC. FBI. **FBI Announces Deployment of Sentinel**, 2012. Disponível em: http://www.fbi.gov/news/pressrel/press-releases/fbi-announces-deployment-of-sentinel. Acesso em: 12 de outubro de 2012.

WELLS, Don. **The Rules of Extreme Programmin**. Estados Unidos, 1999. Disponível em: http://www.extremeprogramming.org/rules.html. Acesso em 09 de julho de 2012.