CENTRO PAULA SOUZA

FACULDADE DE TECNOLOGIA DE AMERICANA

Curso de Tecnologia em Análise e Desenvolvimento de Sistemas

Priscila Vilela da Costa

SCRUM COMO METODOLOGIA DE DESENVOLVIMENTO DE SOFTWARE

Americana, SP

2014

CENTRO PAULA SOUZA

FACULDADE DE TECNOLOGIA DE AMERICANA

Curso de Tecnologia em Análise e Desenvolvimento de Sistemas

Priscila Vilela da Costa

SCRUM COMO METODOLOGIA DE DESENVOLVIMENTO DE SOFTWARE

Trabalho de Conclusão de Curso desenvolvido em cumprimento à exigência curricular do Curso de Tecnologia em Análise e Desenvolvimento de Sistemas, sob a orientação do Prof. Mestre Wagner Siqueira Cavalcante

Área de concentração: Engenharia de *Software*

Americana, SP

2014

Costa, Priscila Vilela da

C87s Scrum como metodologia de desenvolvimento de software. / Priscila Vilela da Costa. – Americana: 2014.

50f.

Monografia (Graduação em Tecnologia em Análise e Desenvolvimento de Sistemas). - - Faculdade de Tecnologia de Americana – Centro Estadual de Educação Tecnológica Paula Souza.

Orientador: Prof. Me. Wagner Siqueira Cavalcante

1.Desenvolvimento de software I. Cavalcante, Wagner Siqueira II. Centro Estadual de Educação Tecnológica Paula Souza – Faculdade de Tecnologia de Americana.

CDU: 681.3.05

Priscila Vilela da Costa

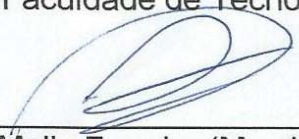
SCRUM COMO METODOLOGIA DE DESENVOLVIMENTO DE SOFTWARE

Trabalho de graduação apresentado como exigência parcial para obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas pelo CEETEPS/Faculdade de Tecnologia – FATEC/ Americana.
Área de concentração: Engenharia de *Software*

Americana, 05 de dezembro de 2014.

Banca Examinadora:

Wagner Siqueira Cavalcante (Presidente)
Mestre
CEETEPS/Faculdade de Tecnologia – FATEC/ Americana



Alexandre Mello Ferreira (Membro)
Mestre
CEETEPS/Faculdade de Tecnologia – FATEC/ Americana

Clerivaldo José Roccia (Membro)
Mestre
CEETEPS/Faculdade de Tecnologia – FATEC/ Americana

AGRADECIMENTOS

Em primeiro lugar gostaria de agradecer a Deus por ter me concedido esta oportunidade.

Ao meu orientador, Prof. Wagner Siqueira Cavalcante, que me guiou ao longo do desenvolvimento desta monografia com suas dicas.

A todos os mestres, especialistas e professores, que durante todo o curso se dedicaram e transmitiram seus conhecimentos, contribuindo para meu aprendizado.

DEDICATÓRIA

A minha mãe, que sempre me apoiou nesta difícil caminhada até a conclusão do presente curso, mesmo nos momentos em que eu quis desistir, sempre me dando forças para continuar.

RESUMO

O presente texto conceitua as metodologias para desenvolvimento de *software*, com o foco principal no *Scrum*, abordado como tema principal deste trabalho. Assim, para possibilitar uma abordagem adequada, são apresentadas algumas sínteses dos modelos Cascata, Evolucionário, Prototipação, Espiral, Métodos Ágeis e XP (*Extreme Programming*). Também detalham-se as etapas que fazem parte da criação de um sistema com o *Scrum*, dando um maior destaque aos tópicos principais, que são o *Backlog* do Produto e a Retrospectiva de *Sprint*. Além disso, são apresentados três estudos de caso, reais e com sucesso a respeito desta metodologia, sendo eles o Noshster, um site relacionado ao mundo gastronômico, o ANKOS, que funciona como um banco de informações para a saúde pública e, por fim, o Conferous, que é uma aplicação de conferências *online*. Ao final, apresenta-se um breve comparativo entre a metodologia em estudo e as demais presentes no mercado, com o objetivo de constatar se o *Scrum* realmente possui mais vantagens do que os outros métodos no desenvolvimento de software, e em que situação torna-se adequado aplicá-lo.

Palavras Chave: *Metodologia; Modelo; Scrum.*

ABSTRACT

This paper conceptualizes the methodologies for software development, with the main focus on Scrum, addressed as the main topic of this work. Thus, to enable a proper approach, some summaries of Cascade, Evolutionary, Prototyping, Spiral, Agile and XP (Extreme Programming) models are presented. It is also detail the steps that are part of creating a system with Scrum, giving greater prominence to the main topics, which are the Product Backlog and Sprint Retrospective. In addition, three study cases, real and successful regarding this methodology are presented, the Noshster, a website related to the gastronomic world, ANKOS, which acts as a database of information for public health and the last, Conferous, which is an application to online conferences. At the end, a brief comparison between the method under study and the others on the market is presented, aiming to mark if Scrum actually has more advantages than other methods in software development, and in which situation it is appropriate to apply it.

Keywords: *Methodologie; Mode; Scruml*

SUMÁRIO

LISTA DE FIGURAS E DE TABELAS

1 INTRODUÇÃO

Quando os *softwares* começaram a ser desenvolvidos, não havia um método padrão que pudesse ser seguido, o que fez com que cada desenvolvedor tivesse sua metodologia própria, o que criou muitos problemas, como atrasos dos projetos, códigos difíceis de serem interpretados, projetos com preços superiores ao orçado, cronogramas impossíveis de se obedecer, entre tantos outros problemas. Tudo isso foi chamado, durante a década de 1970, de “crise do *software*”, porém esta não passou e está presente até hoje (FURTADO).

Isto pode se tornar um problema visto que nos dias atuais há *softwares* não só nos computadores, mas também em *smartphones*, *tablets*, embutidos em automóveis e eletrodomésticos e, mais recentemente, chegaram aos dispositivos vestíveis, como é o caso do Google *glass* (óculos) e dos *smart* relógios, que terão o Android (Sistema Operacional do Google). Sundar Pichai, vice-presidente das divisões Android e Chrome do Google, afirmou em uma palestra, que tem a intenção de lançar pequenos pacotes de desenvolvimento, que facilitem a adaptação do Android a esses gadgets (equipamento com funções especifica que normalmente são usados no dia a dia) (BARROS, 2014).

Dados do *The Standish Group* (2013) mostram que aproximadamente 18% de todos os projetos de criação de *software* falham e que 43% deles tem algum imprevisto, com um total de apenas 39% dos projetos sendo realizados sem nenhum contratempo. Ainda segundo o *The Standish Group* (2013), aproximadamente 59% dos projetos tem um aumento de custo, e 74% ultrapassam o tempo inicial para o desenvolvimento.

“A raiz comum para a maioria das falhas, na verdade, está na (ausência de) utilização de metodologias de desenvolvimento e como elas interagem com os indivíduos envolvidos em um projeto de *software*“ (FURTADO).

Por este e outros motivos, questiona-se se realmente existe uma metodologia ideal para o desenvolvimento de *software*, independente de seu porte.

Sendo assim, o **objetivo geral** deste trabalho é apresentar explicitamente o *Scrum*, explorar a criação de *softwares* com o mesmo e como ele funciona neste

processo, mostrando fatos que comprovam que este é o método mais vantajoso para o desenvolvimento de *softwares*.

Os **objetivos específicos** são, portanto:

- Fazer uma breve introdução sobre desenvolvimento de *software* (como é o processo, por que surgiu o conceito, etc.).

- Apresentar um pouco sobre os outros métodos de desenvolvimento de *software* (cascata, desenvolvimento incremental, etc.), mostrando parte de seu funcionamento e estrutura, também onde eles melhor se aplicam.

- Explicar como o *Scrum* surgiu e como ele é usado atualmente.

- Explicar como o *Scrum* funciona, o passo a passo do seu processo e todos os agentes envolvidos em um projeto com uso do *Scrum*.

- Por fim, comparar o *Scrum* com os outros métodos apresentados anteriormente por meio de critérios específicos.

O **método cientifico** empregado para o desenvolvimento deste trabalho foi de caráter qualitativo, por meio de notícias e artigos eletrônicos ou impressos, livros e sites. Também se empregaram alguns estudos de casos como uma forma de pesquisa de caráter experimental, nos quais o tema da proposta do trabalho foi empregado.

O trabalho foi estruturado em quatro capítulos, o primeiro conceitua as outras metodologias existentes para o desenvolvimento de *software*, o segundo apresenta o *Scrum*, discutindo todas as etapas envolvidas no mesmo.

O terceiro capítulo exemplifica três casos reais de uso do *Scrum* no desenvolvimento de *software*. Por último o quarto capítulo compara o *Scrum* com os outros métodos, levando assim aos argumentos para às considerações finais.

2 DESENVOLVIMENTO DE *SOFTWARE* ALÉM DO *SCRUM*

Antes mesmo de existir o *Scrum* ou qualquer um dos métodos de desenvolvimento de *software*, os mesmos eram desenvolvidos de forma livre, sem nenhum processo específico a ser seguido. O desenvolvimento de *software* aconteceu desta forma até a crise do *software* em 1968.

“A crise de software resultava diretamente da introdução de novo *hardware* de computador baseado em circuitos integrados” (SOMMERVILLE, 2007, p.3).

Segundo Sommerville (2007), esses novos *hardwares* apresentavam as possibilidades de se ter aplicações de computador mais complexo que os feitos até aquele momento, e o modo informal de se fazer sistemas que era adotado naquele período começou a representar atrasos de até mesmos anos e custos muito elevados.

Este cenário caótico levou à criação da Engenharia de *Software*, proposto como uma forma de solucionar a crise do *software*. É no contexto Engenharia de *Software* que as metodologias começaram a surgir, visto ser uma disciplina de engenharia relacionada a todos os aspectos de produção de *software* (SOMMERVILLE, 2007).

As metodologias ou modelos de desenvolvimento de *software* são formas de interpretação do processo de *software*. O processo pode ser interpretado como “uma coleção de padrões que definem um conjunto de atividades, ações, tarefas de trabalho, [...] necessários ao desenvolvimento de *software* de computador” (PRESSMAN, 2010, p.24).

Esse processo de *software* deve possuir alguns passos básicos, de acordo com Sommerville (2007) eles são: especificação de *software*, desenvolvimento de *software*, validação de *software* e evolução de *software*. E a partir desses passos e do processo que se surgem os modelos/métodos específicos para cada proposta.

> Modelos [...] têm sido aplicados durante muitos anos em um esforço de trazer ordem e estrutura para o desenvolvimento de *software*. Cada um desses modelos convencionais sugere um fluxo de processo de *software* um tanto diferente, mas todos realizam o mesmo conjunto de atividades [...] (PRESSMAN, 2010, p.55).

Partindo do princípio que há muitos modelos que foram propostos como uma tentativa de solucionar a crise do *software*, será abordado neste capitulo um pouco sobre o funcionamento de alguns modelos além do *Scrum*.

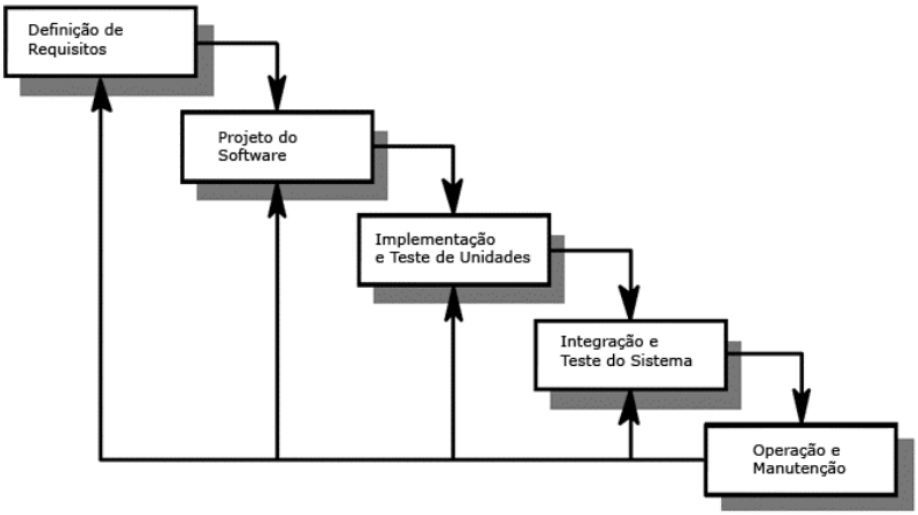
2.1 MODELO CASCATA

De acordo com Pressman (2010) o modelo mais antigo da Engenharia de *Software* é o cascata. Ele funciona de forma linear e sequencial, composto por cinco etapas para o desenvolvimento do *software* conhecidas como ciclo de vida clássico.

Essas etapas, segundo Sommerville (2007), são a definição de requisitos, projeto de sistema de *software*, implementação e teste de unidade, testes de integração de sistema, operação e manutenção. Essas etapas devem ser realizadas em sequência e as fases só podem ter inicio após o fim da fase anterior (SOMMERVILLE, 2007).

O nome “cascata” (ou *waterfall*) dado a este modelo se deve a essa ordem de execução dos passos, pois estes são organizados desta forma sequencial em escada (semelhante a uma cascata) como é mostrado na Figura 1.

Figura 1 - Ciclo de Vida Clássico



Fonte: SOUSA (2012)

A primeira etapa é a de definição de requisitos, que pode ser explicada pelo próprio nome, pois tem como propósito definir os objetivos e restrições do sistema por meio de consulta ao usuário (SOMMERVILLE, 2007).

A segunda etapa é o projeto de *software*, que cuida de criar a arquitetura do sistema. Ele “envolve a identificação e a descrição das abstrações fundamentais do sistema de *software* e suas relações” (SOMMERVILLE, 2007, p.44).

A terceira etapa, a implementação e testes unitários, verifica se cada unidade do sistema está atendendo aos seus objetivos. A quarta etapa, Integração e Testes do Sistema, procura constatar se o sistema como um conjunto inteiro está atendendo aos requisitos (SOMMERVILLE, 2007).

Por último tem-se a fase de operação te manutenção, que é quando o sistema é implantado, e são feitas as correções de erros não encontrados anteriormente (SOMMERVILLE, 2007).

Porém, esse funcionamento sequencial no desenvolvimento com a utilização do modelo cascata gera alguns problemas, e seu uso passou a ser questionado nos últimos anos. Os problemas giram em torno de mudanças ao longo do projeto que podem ocorrer, pois dificilmente o cliente conseguirá definir todos os requisitos no início do projeto. Outro ponto é a falta de uma versão executável antes do fim do sistema. (PRESSMAN, 2010)

“Hoje em dia, o trabalho de *software* é em ritmo rápido e sujeito a uma torrente sem fim de modificações [...] modelo cascata é frequentemente inadequado para esse tipo de trabalho” (PRESSMAN, 2010, p.39).

Pode-se concluir que o modelo cascata é um modelo sequencial, lógico, mas que uma modificação não prevista ao longo do seu uso pode causar danos catastróficos e gerar custos elevados para os projetos.

2.2 MODELO EVOLUCIONÁRIO

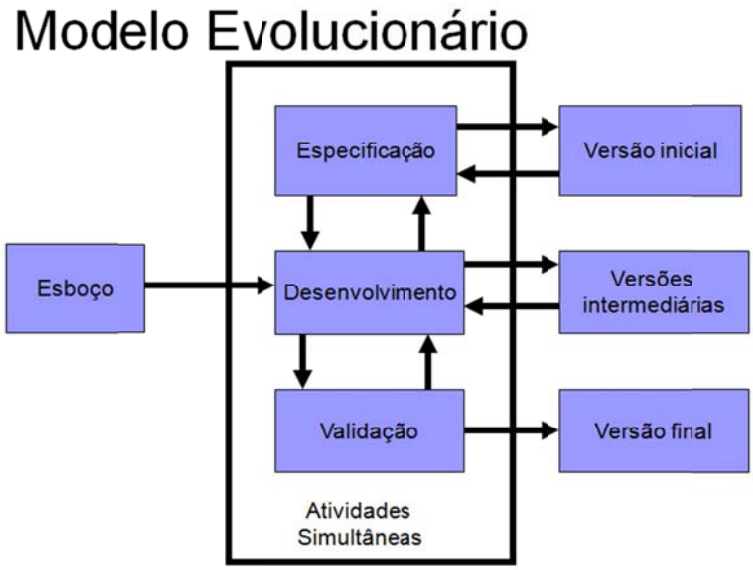
O desenvolvimento evolucionário é um modelo no qual existe uma implementação inicial e, ao longo do processo de desenvolvimento do *software*, os requisitos e os resultados são refinados de acordo com os comentários do cliente em relação ao que está feito até aquela etapa. De acordo com Sommerville (2007), existem dois modelos para o desenvolvimento evolucionário.

O primeiro deles é o exploratório, que tem como objetivo absorver os comentários do cliente na forma de requisitos ao longo do desenvolvimento, incorporando isto ao sistema.

O segundo é o de prototipação *throwaway* (descarte), que busca entender os requisitos do cliente e expor no protótipo aqueles que não foram bem compreendidos.

Esse modelo pode ser exposto na Figura 2, na qual fica claro o processo que tem uma versão inicial e depois são criadas as intermediárias de acordo com as interversões do cliente, durante o período de validação, até se chegar à versão final.

Figura 2 - Desenvolvimento Evolucionário



Fonte: MAROTA (2012)

Porém, Sommerville (2007) também afirma que esse modelo não é recomendado para sistemas complexos, com longos ciclos de vida e grande porte. Isso pode estar relacionado ao fato dele não possuir um processo visível, onde não há produtos regulares para se medir o progresso do projeto. Também existe uma conexão com a má estruturação que ocorre no modelo, devido às mudanças continuas, causadas pela adição de novos requisitos apontados pelo cliente no processo de desenvolvimento do *software*.

“A vantagem [...] é que a especificação pode ser desenvolvida de forma incremental. À medida que os usuários compreendem melhor seu problema, isso pode ser refletido no sistema de *software”* (SOMMERVILLE, 2007, p.45).

2.3 PROTOTIPAÇÃO

Um problema muito comum durante o desenvolvimento de *software* é a alteração constante dos requisitos do projeto, pois muitas vezes o cliente não identificou os requisitos de entrada, processamento e saída. Existem também casos em que o desenvolvedor não consegue identificar se um algoritmo é eficiente. Para esses casos recomenda-se uma abordagem de prototipação. (PRESSMAN, 2010)

A prototipação fornece ao desenvolvedor um modo de criar um modelo do *software* final do projeto. Mazzola (2010) afirma que o foco principal deste modelo é eliminar o “congelamento” dos requisitos do projeto do sistema.

A ideia é executar a fase de coleta de requisitos normalmente, e após ela se criar um protótipo, chamado de projeto rápido, que deve apresentar as abordagens de entrada, processamento e saída para os usuários, desta forma redefinindo os requisitos para obter um sistema que satisfaça as necessidades do cliente. A Figura 3 ilustra o ciclo de como funciona a prototipação.

Figura 3 - Ciclo de Prototipação

Fonte: ARAÚJO (2011)

Idealmente o protótipo funciona como uma forma de refinar os requisitos de *software*, podendo também ser considerado como o primeiro sistema, que se trata da primeira versão que normalmente é descartada (PRESSMAN, 2010).

Normalmente o protótipo é apresentado ao cliente de três formas diferentes. A primeira delas é um modelo em papel ou executado no computador que retrate a *interface* do sistema. O segundo modo se trada da implementação de um (ou mais) subconjunto de requisitos. Por último, ele é feito na forma de um programa existente que represente todas ou parte das funções a serem construídas no *software* (MAZZOLA, 2010).

Assim, em resumo, a ideia da prototipação seria apresentar essas pequenas versões do projeto, que são aprovadas pelo cliente, de modo a evitar que seja entregue um produto final que não atenda os requisitos do projeto.

2.4 MODELO ESPIRAL

O modelo espiral foi originalmente proposto por Barry Boehm e é considerado um modelo relativamente novo, que abrange as melhores características do modelo cascata e do uso de prototipação. A representação do modelo é feita em formato espiral.

“Cada *loop* na espiral representa uma fase do processo de *software*. [...] O *loop* mais interno pode estar relacionado à viabilidade [...] o próximo, ao projeto de sistema e assim por diante” (SOMMERVILLE, 2007, p.48).

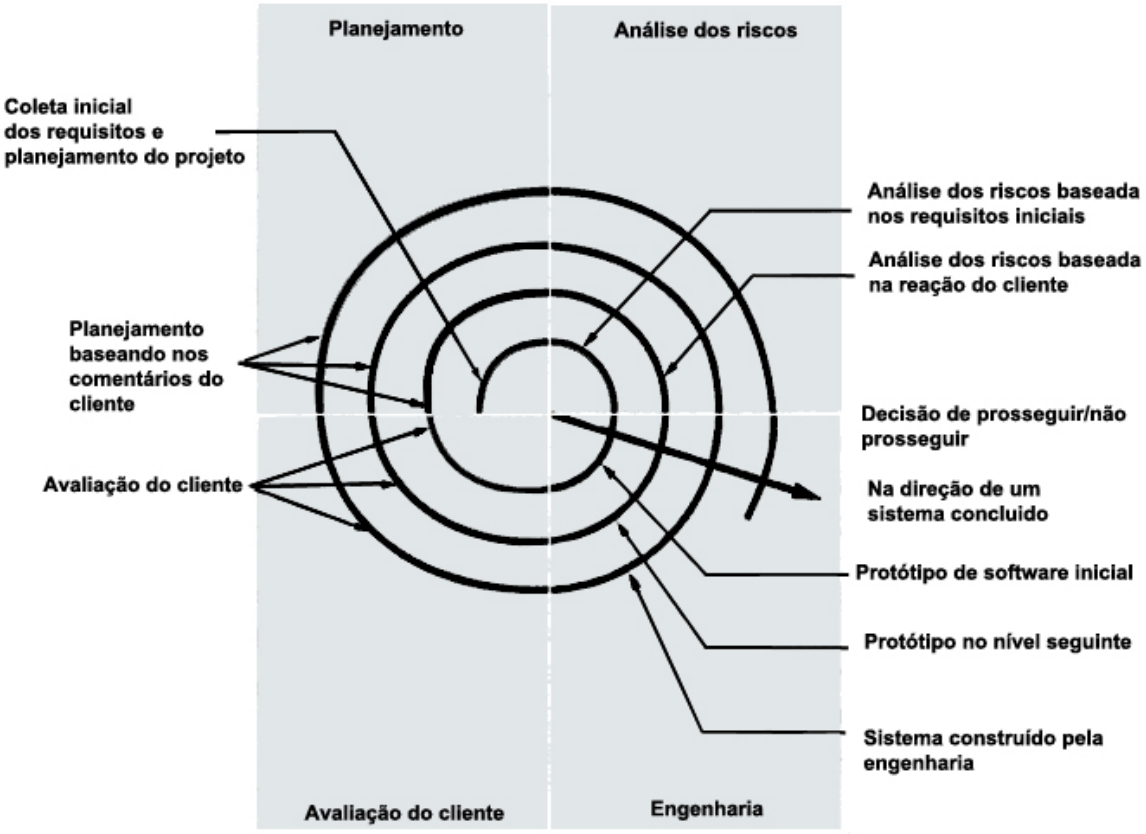
Segundo Pressman (2010), existem quatro atividades importantes no modelo espiral:

- Planejamento, que é a determinação dos objetivos;
- Análise dos riscos, que são as alternativas para solucionar os riscos;
- Engenharia, que representa o produto final;
- Avaliação, feita pelo cliente, que se entende pela avaliação dos resultados da engenharia.

“Com cada interação ao redor da espiral (iniciando-se ao centro e avançando para fora), versões progressivamente mais completas do *software* são construídas” (PRESSMAN, 2010, p. 39).

As atividades do modelo (planejamento, análise dos riscos, engenharia e avaliação) possuem suas próprias etapas, que fazem parte do processo ao longo da espiral, elas podem ser vistas na Figura 4, onde estão divididas as etapas e seus componentes.

Figura 4 - Modelo Espiral



Fonte: ARAÚJO (2011)

Seguindo o modelo, ao final de cada interação, o cliente vai avaliando o que foi feito por meio da qual define-se o planejamento da próxima fase (PRESSMAN, 2010).

“Em cada arco da espiral, a conclusão da análise dos riscos resulta numa decisão de “prosseguir/não prosseguir”. Se os riscos forem muito grandes, o projeto pode ser encerrado” (PRESSMAN, 2010, p.40).

De acordo com Sommerville (2007), a diferença entre o modelo espiral e os outros modelos é que ele reconhece de forma explicita a existência de riscos que podem causar atrasos e problemas de custo no projeto.

Porém, caso o risco passe despercebido, incontestavelmente irão ocorrer problemas. Assim, o aspecto importante desse modelo é que só se segue para o planejamento de uma nova atividade após a aprovação do cliente e avaliação dos riscos (PRESSMAN, 2010).

2.5 MÉTODOS ÁGEIS

Na década de 1980 e início da década de 1990, os métodos usados para desenvolvimento de *softwares* eram robustos e complexos, com muito tempo dedicado ao planejamento. Assim, para evitar gastar mais tempo com planejamento do que com o desenvolvimento e testes do sistema em si, engenheiros de *software* criaram alguns métodos ágeis no começo da década de 1990 (SOMMERVILLE, 2007).

Entre esses métodos se encontra o *extreme programming* (que pode ser considerado como o mais conhecido), o *Scrum,* DSDM (Metodologia de Desenvolvimento de Sistemas Dinâmicos), entre outros. Devido ao sucesso desses métodos, eles são comumente incorporados aos métodos tradicionais.

Todos os métodos ágeis são baseados em entregas incrementais, nas quais o sistema é entregue por partes (incrementos), esses incrementos são criados baseados nos requisitos do cliente.

De acordo com Sommerville (2007), os princípios dos métodos ágeis são: envolvimento do cliente (muito presentes durante todo o processo, avaliando passo a passo), entrega incremental, pessoas e não processo (habilidades da equipe reconhecidas e exploradas), aceitação de mudanças e manutenção a simplicidade.

Esses métodos podem ser vantajosos, mas apresentam dificuldades, principalmente para conseguir a disponibilidade do cliente de estar presente e contribuir o tempo todo com o projeto. Sommerville (2007) não considera estes

métodos como adequados para o desenvolvimento de sistemas de larga escala ou críticos.

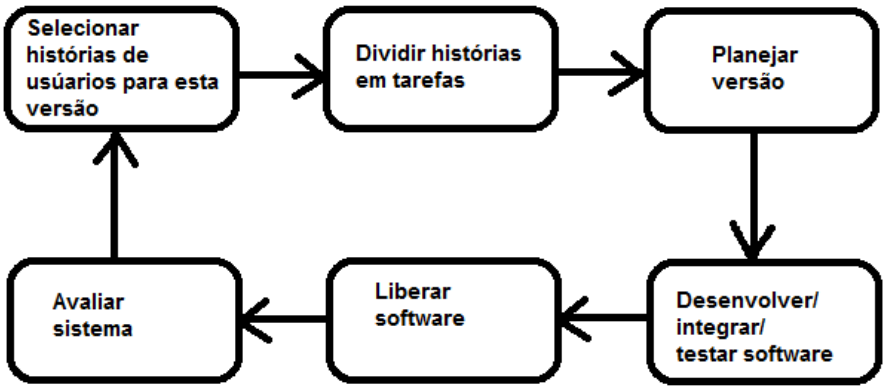
2.6 EXTREME PROGRAMMING

O nome desse método ágil surgiu devido ao fato de o mesmo envolver o cliente em níveis extremos no projeto. Neste método, os requisitos são chamados de cenários (histórias), que são divididos em tarefas. Em resumo, de acordo com Sommerville (2007), esse método possui os seguintes princípios:

- Planejamento incremental (através de cartões de histórias divididas em tarefas);
- Pequenos *releases* (o projeto é liberado aos poucos);
- Projeto Simples (feito o suficiente para atender os requisitos atuais);
- Desenvolvimento *test-first* (existe um *framework* para criação de testes unitários);
- *Refactoring* (todos os desenvolvedores devem recriar o código continuamente);
- Programação em pares*;*
- Propriedade coletiva (os pares de programadores trabalham em todas as áreas do sistema);
- Integração contínua (assim que o trabalho é concluído o mesmo se integra ao *software*);
- Ritmo sustentável (não é aceitável haver muita hora extra);
- Cliente *on-site* (o cliente é parte da equipe).

No ciclo do XP que é visto na Figura 5, é explicado as suas etapas que são a coleta das histórias, depois seu planejamento, a execução do projeto em si, e depois de liberado a versão do sistema se tem uma avaliação do que foi entregue.

Figura 5 - Ciclo de Vida Extreme *Programming*



Fonte: SOMMERVILLE, 2007, p.264

Assim, nesse método, o cliente é tido como parte ativa da equipe, na qual ele discute os cenários com a equipe e prioriza os mesmos. Os programadores devem programar em pares, para que um possa testar o trabalho do outro. Durante a programação, novas versões do sistema são compiladas às vezes mais de uma vez por dia, e cada vez que uma compilação é feita, o programador deve testá-la e somente se a mesma passar por todos os testes, é incorporada ao resto do *software* (SOMMERVILLE, 2007).

No caso deste método, os programadores não se antecipam para futuras mudanças no *software* e isso é um problema, pois elas contribuem para a degradação da estruturado *software*, dificultando cada vez mais a implementação de mudanças (SOMMERVILE, 2007).

O princípio do *extreme programming* é que o *software* deve ter um código de fácil entendimento, para que as alterações sejam rápidas para se implantar novas histórias.

3 *SCRUM*

O *Scrum* é uma metodologia que se encaixa em sistemas de diferentes portes e complexidades. Ele se baseia em reuniões diárias entre a equipe de desenvolvimento e o cliente, exigindo um contato constante entre ambos (*Product Owner*), gerando como resultado final um *software* que está em concordância com o *Product Backlog* (lista de requisitos do cliente). (DESENVOLVIMENTO ÁGIL, 2013)

Scrum pode ser definido como:

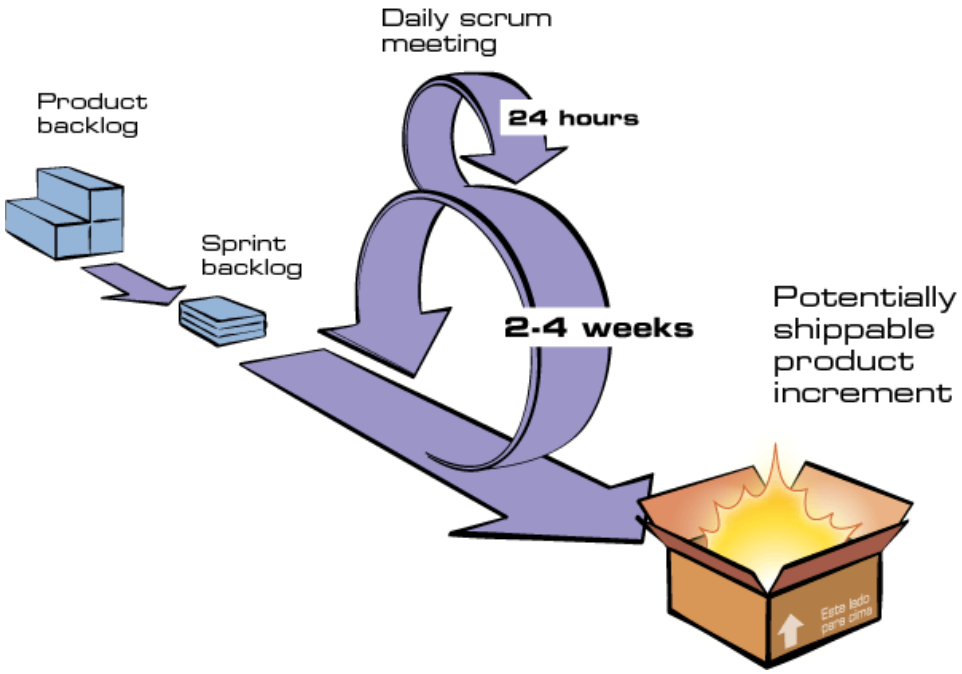
“Tal qual no esporte rugby, é uma forma de reiniciar o jogo, seja após um incidente, seja depois que a bola sai de jogo [...] manter o jogo (desenvolvimento de *software*) rolando” (PHAM, 2011, p.37).

A origem desse método é de um artigo publicado por Hirotaka Takeuchi e Ikujiro Nonaka, na *Harvard Business Review* de 1986, no qual eles descreveram uma abordagem em que as equipes subdividiam-se em pequenas equipes de multifunções, trabalhando juntas para um objetivo comum, e compararam isso ao *Scrum* do rugby (PHAM, 2011).

Anos depois, Jeff Sutherland e Ken Schwaber aplicaram algumas dessas medidas simultaneamente em suas respectivas empresas, e em 1995, a pedido da *Object Management Group* (OMG), eles criaram o *Scrum*, uma nova metodologia descrita no artigo “*Scrum and the perfect storm*” (*Scrum* e a tempestade perfeita) (PHAM, 2011).

No método, há o *backlog* do produto (*Product Backlog*), que contém os requisitos em que a equipe seleciona as atividades da s*print*. Todos os dias são feitas as *Daily Scrum*. No final das s*prints*, realizam-se as reuniões de revisão de *Sprint*, e depois, uma reunião de retrospectiva, que antecede o planejamento da próxima *sprint*, esse processo é representado graficamente na Figura 6 (DESENVOLVIMENTO ÁGIL, 2013).

Figura 6 - Processo do *Scrum*



Fonte: DESENVOLVIMENTO ÁGIL (2013)

3.1 *BACKLOG* DE PRODUTO (*PRODUCT BACKLOG*)

Backlog de produto é uma lista das necessidades expostas pelo cliente, e que precisa sempre ser revista e atualizada.

“O *product backlog é o coração do Scrum*. É aqui que tudo começa. O *product backlog* é basicamente uma lista de requisitos, estórias. Coisas que o cliente deseja, descritas utilizando a terminologia do cliente” (KNIBERG, 2007, p.9).

Essa lista de requisitos deve ser mantida pelo dono do produto que faz a priorização dos requisitos da lista, além disso, ela é uma lista altamente dinâmica, visto que os itens podem ser adicionados e removidos de acordo com o desenvolvimento das *sprints* (COHN, 2011)

O objetivo dos métodos ágeis é fazer com que haja um equilíbrio entre a escrita e a discussão, logo, as histórias são uma descrição curta e simples do usuário sobre as funcionalidades que o sistema deve ter. Frequentemente elas são escritas em arquivos ou notas adesivas (chamadas de fichas), que são armazenadas em caixas de fácil acesso para auxiliar no planejamento (COHN, 2011).

Essas fichas funcionam como uma forma de afirmar o compromisso entre a equipe e o dono do produto, visto que nas mesmas não há nada muito detalhado, e esses detalhes surgem e são compreendidos nas conversas entre o dono do produto e a equipe (COHN, 2011).

Um exemplo de como essas fichas podem ser preenchidas, é dado por Kniberg (2007), sugerindo que as fichas contenham um ID (identificador da ficha), nome (algo que a equipe e o dono do produto entendam), importância (a importância dela para o dono do produto), estimativa inicial (o tempo que a história ira demorar), como demonstrar (como ela é apresentada na *sprint*) e, por último, as notas (informações úteis).

Essas histórias ou itens de *backlog* de produto, formam um *iceberg* na forma como são dispostas, começando no topo com requisitos pequenos, que são implementados em uma única Sprint, e vão ficando maiores à medida que se desce no "*iceberg*" *(*COHN, 2011).

Também é importante lembrar que os requisitos sempre mudam-se e surgem novos requisitos ao longo do projeto, assim é essencial que o *backlog* de produto seja constantemente arrumado, Cohn (2011) recomenda que 10% dos esforços de cada *sprint* sejam para reorganizar o *backlog*.

Assim chega-se à questão: Como esse *backlog* de produto deve ser coletado? Uma possível resposta é apresentada por Pham (2011), identificando-se os *stakeholders* (seus objetivos) e depois adentra-se a uma "floresta de requisitos".

No primeiro passo são utilizadas as regras SMART (Específico, Mensurável, Alcançável, Realista e Programado, em Português), para identificar os objetivos dos *stakeholders* e seus requisitos (PHAM, 2011).

Depois, é criada a floresta de requisitos, compreendendo que cada árvore é uma funcionalidade (uma parte do produto) e cada folha é uma história do produto. Dessa forma, ao final da floresta tem-se como resultado o *backlog* do produto (PHAM, 2011).

3.2 DONO DO PRODUTO (*PRODUCT OWNER*)

O dono do produto é quem vai garantir que a equipe esteja dedicada ao objetivo correto. Ele é responsável por priorizar e definir o *backlog* do produto, assegurando que o projeto tenha um bom retorno em relação ao que foi investido nele (COHN, 2011).

Segundo Pham (2011), o dono do produto tem de ter sete qualidades:

- Saber gerenciar as expectativas dos *stakeholders*;
- Ele deve ter uma visão clara do produto;
- Saber coletar os requisitos;
- Tem de estar completamente disponível para a equipe do projeto;
- Saber manusear várias atividades;
- Deve comunicar a visão do produto;
- Ser um líder que consiga guiar, treinar e dar suporte à equipe.

Pham (2011) acredita que, para se ter uma visão clara do produto, deve-se perguntar cinco questões: Para quem, por que, o quê, onde e quando? De acordo com Cohn (2011), o dono do produto deve passar esta visão para o restante da equipe, sendo esta uma das duas coisas que ele deve fornecer a equipe.

O compartilhamento da visão com a equipe motiva os integrantes a criarem uma conexão de longo prazo com os usuários do produto. A segunda coisa que o dono do produto deve fornecer a equipe é a compreensão dos limites, os quais devem descrever a realidade sobre a qual a visão será realizada (COHN, 2011).

Dentro destes limites, ele irá estipular prazos, o que deve ser feito, entre outras atividades. Esses limites devem servir de motivadores para a equipe superar os problemas difíceis encontrados ao longo do projeto. Em suma, o limite é uma caixa dentro da qual a equipe deve pensar, e ela deve se basear nas restrições de negócio. Assim, esta caixa impede que a equipe se perca dentre as muitas soluções dadas aos membros para comparação de escolhas (COHN, 2011).

Além disso, o dono do produto tem como principal atividade saber gerenciar as expectativas dos *stakeholders*, dando prioridade àqueles que possuem mais interesse e influência sobre o projeto (PHAM, 2011).

Por fim, é importante que o dono do produto esteja disponível, preferencialmente todos os dias, para a interação com a equipe, comparecendo a todas as reuniões de revisão (PHAM, 2011).

Apesar do dono do produto ser um papel único, existem alguns casos em que a complexidade do projeto exige mais de uma pessoa para esta função. Neste caso cria-se uma equipe de donos de produto. Porém esta equipe deve ter uma pessoa com a responsabilidade de todas as atividades desempenhadas pelas equipes delegadas a ela, ou seja, uma pessoa de autoridade máxima (COHN, 2011).

3.3 *SCRUM MASTER*

O *Scrum Master* é semelhante a um *personal trainer*, que não pode forçar a praticar os exercícios, o que ele faz é lembrar dos objetivos e como se quer atingí-los, ou, em outras palavras, ele não tem autoridade sobre os membros da equipe e sim sobre o processo (COHN, 2011).

Ele deve ter conhecimentos técnicos, de mercado ou conhecimentos específicos que auxiliem a equipe, mas, acima de tudo, deve ter um conhecimento prático de *Scrum*, adquirido em projetos anteriores (PHAM, 2011).

Pham (2011) defende que um bom *Scrum Master* tem sete características: Deve ter conhecimento, ser um ótimo líder-servidor, ter fortes habilidades organizacionais, ser bom em apresentações, saber resolver conflitos e ter boas habilidades de desenvolvimento humano.

“Um dos papéis mais importantes do *Scrum Master* é servir à equipe durante os *sprints*, removendo o máximo de impedimentos possível e protegendo a equipe, o máximo que puder, de distúrbios externos” (PHAM, 2011, p.182).

Em outras palavras, o *Scrum Master* deve assumir a responsabilidade de ajudar os membros da equipe a usarem o *Scrum*. Também deve sempre colocar as

necessidades da equipe para atingir o objetivo em primeiro lugar. E, para garantir que ele limpe os obstáculos, deve ser comprometido, nunca deixando estes problemas muito tempo (ou dias) sem solução (COHN, 2011).

Além disso, segundo Cohn (2011), o ideal é que o *Scrum Master* não seja uma pessoa vinda de uma contratação externa, e, sim, da própria empresa, o que pode auxiliar o mesmo em uma das suas atividades, que é a resolução de conflitos entre os membros da equipe, para evitar comprometer a capacidade de entrega da mesma (PHAM, 2011).

Outro papel importante é que ele deve garantir que os membros da equipe levem questões para uma discussão aberta, e que eles recebem apoio para isso (COHN, 2011).

Por fim, ele serve de ponte entre o dono do produto e os relatórios que devem ser apresentados nas reuniões com a gerencia, além de ser responsável por organizar as reuniões que ocorrem durante o projeto (*Daily Scrum,* reuniões de revisão de *Sprint*, etc.) (PHAM, 2011).

Por esses motivos, um dos papéis mais importantes dentro do *Scrum* é o do *Scrum Master.*

3.4 EQUIPE (*SCRUM TEAM)*

A equipe no *Scrum* deve ser auto-organizada, de modo que seus membros sejam capazes de decidir como vão interagir com o projeto, assim elas também devem ser capazes de gerenciar os conflitos existentes (PHAM, 2011).

“O trabalho em equipe ainda é um elemento muito importante em um projeto *Scrum*. [...] Se os membros da equipe [...] não se entendem e nada é feito a respeito disso, percebemos que quase sempre o projeto termina em problemas” (PHAM, 2011, p.139).

É importante levar em conta alguns fatores na hora de se formar uma equipe *Scrum*, como a familiaridade com o assunto e o tamanho da equipe. De acordo com Cohn (2011), é crucial manter a equipe pequena e as distribuições de funcionalidades “*end-to-end*” (fim a fim) visíveis para o usuário.

A questão de se manter a equipe pequena é um dos tópicos mais criticados quando o *Scrum* está para ser implementado em uma organização, porém existem muitas vantagens que serão levantadas a seguir. Sugere-se manter equipes de 5 a 9 pessoas, ou, como Cohn (2011) prefere, equipes de duas pizzas (equipes com tamanho que até duas pizzas são suficientes para o almoço de todos).

Estas vantagens, segundo Cohn (2011), são:

- Menos riscos de que as pessoas deixem de assumir responsabilidades;
- Existem mais chances de se ter uma interação construtiva;
- Gasta-se menos tempos para coordenar os esforços da equipe;
- Todos participam do projeto, ninguém fica de lado;
- Os membros ficam mais satisfeitos;
- Normalmente ninguém fica dedicado a uma única coisa a ponto de se tornar algo prejudicial.

Cohn (2011) assegura que as equipes menores normalmente são mais produtivas, mas ele também afirma que não se deve iniciar um projeto sem se ter pelo menos uma equipe formada, e que em casos de projetos maiores o ideal é se trabalhar com várias equipes pequenas, que deem foco em cada uma trabalhar com um requisito do trabalho por *sprint.*

De acordo com Pham (2011), existem cinco condições para existir um bom trabalho em equipe:

- Deve haver abertura: para se discutir qualquer assunto entre os membros da equipe;
- Respeito: deve haver um respeito mutuo entre os integrantes da equipe;
- Confiança: os membros da equipe devem confiar uns aos outros;
- Solicitude: os integrantes devem estar sempre dispostos a ajudar os outros membros do time;
- Estabilidade no emprego: as pessoas têm que sentir que estão seguras, que não serão descartadas pela empresa.

Além disso, Pham (2011) afirma que existem equipes de alto, médio e baixo desempenho.

> Equipes de alto desempenho são divertidas, receptivas e solícitas. Em contraste, equipes de baixo desempenho costumam caracterizar-se pelo silêncio nas reuniões, sorrisos forçados e uma atitude fechada. Os membros de uma equipe de desempenho médio apenas seguem o fluxo diariamente, fazendo apenas o que é necessário para permitir que as horas passem e sejam pagos por isso, mas não agregam nenhum tipo de valor à empresa (PHAM, 2011, p.147).

Se considerarem-se as características de uma equipe *Scrum*, ela deve ser pequena, resolver seus conflitos e ser auto-organizada. O ideal é buscar sempre uma equipe de alto desempenho quando se trata de projetos *Scrum*, sempre lembrando que esta equipe será formada por profissionais técnicos (desenvolvedores, analistas, etc.), mas ela também pode incluir especialistas na área da qual o *software* pertence.

3.5 REUNIÃO DE PLANEJAMENTO DE *RELEASES*

Antes de explicar o funcionamento da reunião de planejamento de *releases*, é importante levar em conta que, de acordo com Cohn (2011), as equipes de *Scrum* geralmente planejam de uma forma mais precisa que as equipes de processo sequencial.

Isso é muito importante, visto que nesta reunião deve-se responder a questão de quando, na pior hipótese, será entregue cada uma das versões (*releases*) do sistema (KNIBERG, 2007).

Uma das funções do dono do produto é definir a importância dos requisitos, ou seja, em qual versão cada um deles deverá se entregue. Para isso é importante que exista um esforço cooperativo entre o dono do produto e a equipe. O ideal é que a equipe estime e dono do produto esteja presente para responder às perguntas e garantir que nenhuma das decisões gere problemas, como quebra de contrato (KNIBERG, 2007).

Para isto é importante que a equipe tenha um conhecimento prévio a respeito da arquitetura geral do produto, para que, desta forma, exista uma melhor colaboração entre a equipe e o dono do produto (PHAM, 2011).

Estas reuniões ocorrem normalmente em uma sala entre a equipe e o dono do produto, que fornece distrações e pede que a equipe destaque uma determinada quantidade de histórias mais importantes. O dono do produto deve ficar na sala para responder às dúvidas e esclarecer o escopo dos itens (KNIBERG, 2007).

Também é necessário determinar a velocidade de cada *sprint*, além de ser essencial revisar e atualizar, quando necessário, o plano de *release*, conforme as *sprints* são finalizadas. Nestes casos, podem ocorrer atrasos ao plano inicial, e assim, o dono do produto pode negociar com o cliente, alternativas para o problema, como tirar uma funcionalidade daquele *release* e encaixá-la em uma posterior (KNIBERG, 2007).

3.6 REUNIÃO DE PLANEJAMENTO DE *SPRINT*

Antes de entender como funciona e o que deve ser definido na reunião de planejamento de *Sprint*, é importante entender o que é um *sprint*. *Sprint* é uma iteração na qual a equipe de desenvolvimento transforma os requisitos (histórias) em um incremento de um produto passível de entrega (PHAM, 2011).

No final dos *sprints*, espera-se que seja entregue uma versão funcional e de valor para o cliente ou usuário final. É importante que as *sprints* terminem dentro do prazo estipulado (COHN, 2011).

Logo pode-se concluir que o planejamento da *sprint* é uma reunião muito crítica e, de acordo com Kniberg (2007), é o evento mais importante do *Scrum*, pois, caso esse planejamento seja mal feito, o *sprint* será totalmente bagunçado e talvez não se conclua da forma esperada.

Assim, ao fim desta reunião, deve ser definido o objetivo da *sprint*, os membros da equipe, o *sprint backlog* (que será explicado na próxima seção), a data de apresentação do *sprint* e os locais e horários para a *Daily Scrum*, que é o momento em que a equipe sincroniza o quanto cada evoluiu (KNIBERG, 2007).

Nestas reuniões espera-se que o dono do produto faça um resumo dos objetivos das histórias mais importantes e que ele deixe a equipe definir o prazo da *sprint*, assim, se o prazo estimado não for o esperado, ele pode mudar a importância da história (KNIBERG, 2007).

O tamanho da *sprint* depende do tempo que ela irá demorar a ser concluída. Para isto, o ideal é fazer testes de tempo até se chegar a um valor ideal para cada *sprint*. Também é importante se perguntar qual o propósito de se fazer a *sprint*, desta forma delimitando um objetivo para a mesma (KNIBERG, 2007).

Além disto, é importante definir quais itens do *backlog* de produto entrarão na *sprint*, o que pode ser feito com o uso de duas técnicas: usarem uma estimativa por

instinto de que a história deve entrar, ou fazer um cálculo de velocidade para saber que histórias podem entrar no *sprint (*KNIBERG, 2007).

Por ser uma reunião em que muitas decisões são tomadas, é comum que a reunião demore mais que o esperado. Nestes casos, Kniberg (2007) aconselha que a reunião seja interrompida na hora em que ela deveria acabar e que o *sprint* fique incompleto, assim a equipe e o dono do produto apreendem uma lição e passam a respeitar melhor os prazos, pois no *Scrum* tempo é algo muito importante.

3.7 SPRINT *BACKLOG*

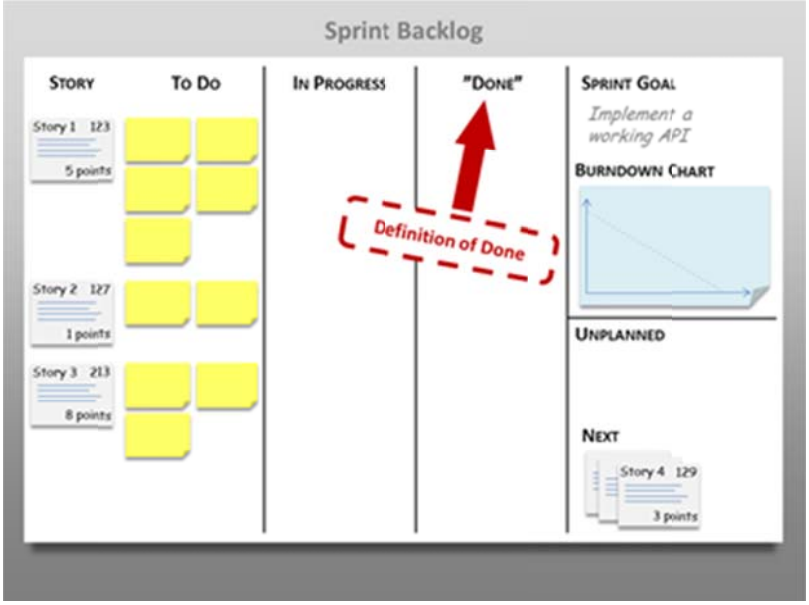
> O *Sprint Backlog* é uma lista de tarefas que o *Scrum Team* se compromete a fazer em um *Sprint*. Os itens do *Sprint Backlog* são extraídos do *Product Backlog*, pela equipe, com base nas prioridades definidas pelo *Product Owner* e a percepção da equipe sobre o tempo que será necessário para completar as várias funcionalidades (DESENVOLVIMENTO ÁGIL, 2013).

O *sprint backlog* deve ser feito depois da reunião de planejamento do *sprint* e antes da primeira *Daily Scrum*. De acordo com Kniberg (2007), a forma mais produtiva para se fazer o *sprint backlog*, é usar um quadro de tarefas (Figura 7) pregado em uma parede.

Além disso, o *sprint backlog* deve ser simples evitando novas adições durante o *sprint*. Isso se deve ao fato de que a cada novo *sprint*, um novo *sprint backlog* deverá ser criado (KNBERG, 2007).

Por fim, é papel do *Scrum master* atualizar o *sprint backlog*, de forma que as tarefas completadas sejam refletidas no mesmo, e de forma que fique claro quanto tempo será necessário para terminar as outras tarefas na opinião da equipe.

Figura 7 - Sprint *Backlog*



Fonte: NORD (2012)

3.8 *DAILY SCRUM*

No *Scrum*, a equipe se reúne diariamente para inspecionar o progresso da equipe de forma a atingir o objetivo do *sprint*. Assim, o *Daily Scrum* não define-se como sendo uma reunião de *status*. O *Daily Scrum* é quando os membros da equipe se reúnem e sincronizam o quanto cada membro progrediu rumo ao objetivo da *sprint* (PHAM, 2011).

Normalmente essas reuniões são feitas na mesma hora e lugar, com todos de pé, como uma forma de evitar que se ultrapasse o tempo da reunião de 15 minutos. Ela também pode ser um espaço para atualizar o quadro de tarefas (*sprint backlog*), com o que já foi feito e o que falta ser feito, apesar de que algumas equipes preferem fazer isto antes da reunião (KNIBERG, 2007).

Pham (2011) afirma que existem três perguntas que devem ser feitas durante a *Daily Scrum*:

- O que você fez ontem?
- O que você vai fazer amanhã?
- O que impede seu progresso?

Durante essas perguntas, é comum existir alguém que não saiba responder o que vai fazer amanhã, e neste caso, Kniberg (2007) sugere que a pessoa seja “pulada” e, no final, após todos falarem, seja feita uma revisão do quadro de tarefas, talvez adicionando novas, e após isto que seja perguntado novamente para quem não sabia o que fazer, se agora ela sabe.

Se for frequente que a pessoa não saiba o que fazer, Kniberg (2007) recomenda que seja ponderada a importância desta pessoa para a equipe. Caso ela não seja muito importante, deve ser retirada da equipe, porém, se ela for importante, deve ser alocada para trabalhar junto com alguém que vai guiá-la na tarefa de descobrir o que ela deve fazer.

No final da reunião, uma pessoa deve totalizar a nova estimativa de prazos e anotar a mesma no gráfico de *burndown* (descrito a seguir).

3.9 GRÁFICO DE *BURNDOWN*

O gráfico de *burndown* representa diariamente o progresso da equipe. Em outras palavras, ele mostra a proporção trabalhada em relação ao total planejado (CAMPOS, 2012).

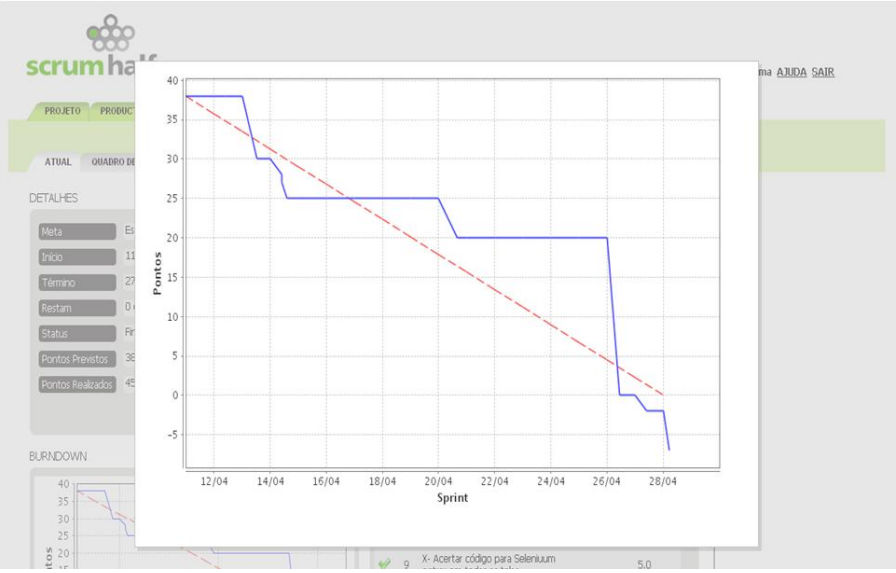
Este gráfico é simples de ser representado Em seu eixo X (horizontal) representa-se o número de dias da *sprint* e, no eixo Y (vertical), representa-se o trabalho restante, que pode ser representado por horas, dias, pontos, etc (CASTRO, 2009).

Depois de desenhados os eixos X e Y, é preciso traçar uma linha diagonal (linha base) com início no ponto máximo ao ponto zero do eixo Y. Esta linha funciona como um guia para se analisar se a equipe está atrasada ou adiantada em relação ao projeto (CAMPOS, 2012).

Paralela à linha base, traça-se uma segunda linha (linha real), que mostra o desempenho real da equipe e, se esta linha fica à direita da linha base, significa que o projeto está atrasado, todavia, se ela ficar à esquerda, o projeto está adiantado. Desta forma é possível, ao longo do *sprint*, ser analisada a velocidade da equipe e verificar porque o projeto não está caminhado como deveria (CAMPOS, 2012).

A seguir, apresenta-se um exemplo de um gráfico de *burndown* (Figura 8), no qual a linha base é representada pela linha vermelha e a linha real, pela cor azul.

Figura 8 - Gráfico de *Burndown*



Fonte: CAMPOS (2014)

3.10 REUNIÃO DE REVISÃO DE *SPRINT*

Ao final de cada uma das *sprints*, são feitas as reuniões de revisão, que normalmente ocorrem na forma de demonstrações do produto apresentadas ao cliente. Os participantes são o dono do produto, o *Scrum master*, clientes, equipe e *stakeholders* em geral (DESENVOLVIMENTO ÁGIL, 2013).

Esta reunião é crítica para o processo de aceitação do produto no *Scrum*. A participação do cliente, mais especificamente do usuário final, aumenta o valor do *feedback* recebido durante a reunião, por isso, por mais que seja desafiador envolver o usuário, é muito importante sua participação (SCHATZ, 2009).

Os papéis dos envolvidos nesta reunião podem ser explicados de modo simples. O dono do projeto tem como responsabilidade ser quem apresenta o produto para os clientes e *stakeholders* para mostrar que ele faz parte da equipe. A equipe deve aproveitar a oportunidade deste contato direto com os usuários finais para mostrar o que foi feito e se isto está de acordo. Os usuários finais vão servir como verdadeiros defensores do produto, quando este for implantado na empresa.

Os *stakeholders* aprendem como o time trabalha. E, por fim, o *Scrum master* deve garantir que tudo ocorra bem, que as pessoas certas estão presentes, entre outros detalhes (SCHATZ, 2009).

Em resumo, é nesta reunião que será avaliado se o que foi planejado foi cumprido, de acordo com os objetivos do s*print.*

“Idealmente, a equipe completou cada um dos itens do *Product Backlog* trazidos para fazer parte do *Sprint*, mas o importante mesmo é que a equipe atinja o objetivo geral do *Sprint”* (DESENVOLVIMENTO ÁGIL, 2013).

3.11 RETROSPECTIVA DE *SPRINT*

A retrospectiva de *Sprint* ajuda o time a se adaptar e melhorar constantemente. Nela, o *feedback* da equipe é dado e também são testadas a comunicação dos membros entre si, com o *Scrum master* e o dono do produto (ANAND, 2011).

Por essas e outras razões, Kninberg (2007) considera que este é o segundo evento mais importante do *Scrum*. É muito comum existir uma resistência inicial da equipe em relação à reunião de retrospectiva do s*print*, mas devido sua importância isto deve ser contornado. Sem as retrospectivas, as equipes continuam a cometer os mesmos erros de forma repetitiva.

A retrospectiva normalmente é feita durante uma reunião que dura de 1 a 3 horas, com a participação do dono do produto, do *Scrum master* e da equipe. O objetivo da mesma gira em torno da seguinte questão: o que pode ser feito melhor para o próximo *sprint*? (KNINBERG, 2007).

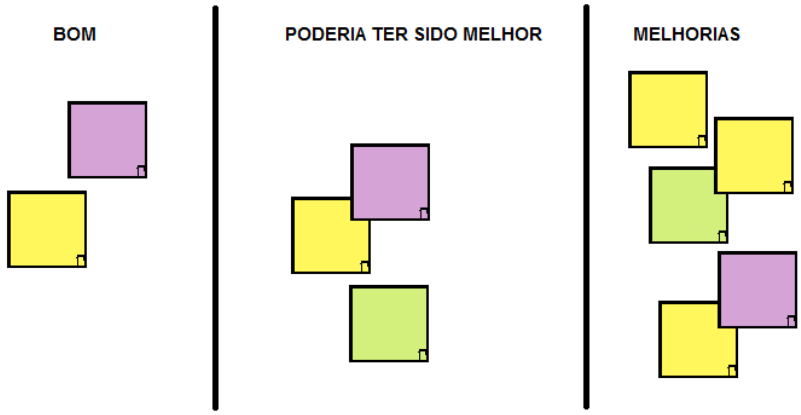
Uma forma de responder esta pergunta é apresentada por Kninberg (2007), em que ele faz um quadro (Figura 9), onde os membros da equipe vão adicionando *post-its* em três colunas: a coluna bom (o que pode ser repetido), a coluna poderia ter sido melhor (se o *sprint* fosse repetido o que seria diferente) e uma coluna de melhorias (o que pode ser melhorado no futuro).

Depois de fazer a análise das três colunas, cada membro da equipe deve escolher três melhorias que eles gostariam de ver no próximo *Sprint*, depois a

equipe deve entrar em consenso e escolher as cinco melhorias que deverão receber mais atenção e que serão validadas na próxima retrospectiva. É importante que o foco seja mantido em apenas algumas melhorias por s*print (*KNINBERG, 2007).

Assim, conclui-se que a retrospectiva é importante, porque é durante ela que os erros e problemas da equipe vêm à tona para serem melhorados nos próximos *sprints* do projeto.

Figura 9 - Quadro de Retrospectiva de Sprint



Fonte: Próprio autor

4 *SCRUM* NA PRÁTICA

A seguir serão apresentados três exemplos do uso de *Scrum* na prática, mostrando alguns dos pontos abordados anteriormente aplicados nestes casos. O primeiro e o último foram desenvolvidos por equipes de cinco pessoas e com *sprints* de uma semana cada. O segundo exemplo foi desenvolvido por um grupo de dez pessoas com *sprints* de 15 dias cada.

4.1 EXEMPLO I - NOSHSTER

O primeiro exemplo é um estudo de caso apresentado por Pham (2011) sobre o uso do *Scrum* e das práticas apresentadas no livro “*Scrum* em Ação”, para se criar a página *web* Noshster.

“Noshster é uma rede social mundial em que gastrônomos escrevem a respeito de pratos que comeram, compartilhando e analisando os restaurantes em que experimentaram tais pratos” (PHAM, 2011, p.197).

Neste estudo de caso, o foco descrito no livro são as práticas de coleta de requisito e desenvolvimento das *sprints*, porém também foram usadas as reuniões clássicas do *Scrum* como *Daily Scrum*, reunião de planejamento de *Sprint*, etc.

Primeiramente, Pham (2011) apresenta uma visão geral do sistema, em que são respondidas as perguntas clássicas desta fase:

Quem? Gastrônomos (usuários).

Por quê? Encontrar os melhores pratos.

O que? Pratos classificados e não restaurantes.

Onde? Verificar as classificações de pratos do mundo todo.

Quando? 24x7 (24 horas por dia, 7 dias por semana).

Após ter a visão definida, foi criada uma “floresta” com quatro árvores, cada qual com suas folhas. Após a criação do *backlog* de produto, a equipe convenceu o

CEO de que o desenvolvimento fosse feito de forma vertical, assim cada árvore representaria um s*print* (PHAM, 2011).

O primeiro *sprint* foi voltado para as funções do usuário, pois desta forma a equipe seria capaz de certificar que o *software* teria um escopo mundial. Assim, neste primeiro momento foram criadas as funções de gerenciamento de usuário (cadastro, *login*, *logout*, visualização de perfil, edição de perfil, exclusão de perfil e navegação de usuários) (PHAM, 2011).

O segundo *sprint* foi voltado para o gerenciamento de pratos (inclusão, edição, visualização e navegação de pratos). O terceiro foi voltado para o gerenciamento de restaurantes (inclusão, edição, visualização e navegação de restaurantes). Por fim, o quarto e último s*print* foi para desenvolver a parte de análise do sistema (inclusão, edição e visualização de análise) (PHAM, 2011).

Por fim, foram descritos os testes realizados no sistema: o unitário, o de integraçãoe o de aceitação. Neste últimо foi simulado um usuário de vida real, para testar as funcionalidades e verificar se tudo estava respondendo como esperado (PHAM, 2011).

4.2 EXEMPLO II - ANKOS

ANKOS (*A New Kind Of Simulator*, ou um novo tipo de simulador), foi desenvolvido por dez estudantes do curso de pós-graduação do centro de informática da Universidade Federal de Pernambuco. Para isto, os estudantes utilizaram as práticas do *Scrum* em um processo de desenvolvimento *open source* com características de desenvolvimento distribuído (SOARES, 2007).

Sendo assim, neste exemplo deu-se foco às práticas de *Scrum* que foram utilizadas no desenvolvimento do ANKOS e nas conclusões que a equipe chegou ao final do projeto.

ANKOS é um sistema da área de saúde pública.

> Um sistema capaz de organizar as informações coletadas por pesquisadores em áreas de estudo da esquistossomose, [...] formando uma base sólida para a aplicação de autômatos celulares que possibilitem a

> geração de cenários capazes de levar a tomada de ações estratégicas de combate e prevenção da doença (SOARES, 2007, p.4).

A primeira coisa que a equipe fez foi montar o *backlog* do produto com base nos requisitos levantados pelo dono do produto. Este *backlog* foi priorizado e dividido entre as *sprints*, e, para saber quais tarefas tinham mais valor de negócio dentro da s*print* e deveriam ser executados primeiro, foram realizadas reuniões assíncronas com o dono do produto no decorrer do s*print* (SOARES, 2007).

De acordo com Soares (2007), as atividades foram divididas em tarefas de até uma semana para cada participante e, para realizar as reuniões diárias, o andamento do s*print backlog, feedback* do dono do produto, etc., foi feito uso de um fórum que era acessado e as informações trocadas.

No final de cada s*print* aconteceram reuniões síncronas por *Skype*, para apresentar ao dono do produto o que foi completado do sistema. Após esta reunião sempre acontecia a retrospectiva do *sprint,* que era feita, na maioria das vezes, presencialmente. Durante essa reunião, um membro da equipe coletava os pontos fortes e fracos, além de terem algumas métricas coletadas (SOARES, 2007).

Algumas dessas métricas foram a variação média do esforço por *sprint*, que ficou em torno de 15% do esperado, e a estimativa de produtividade, que havia sido estimada em 18 homem-hora e acabou sendo 10 homem-hora, para cada ponto de caso produzido (SOARES, 2007).

Por fim, Soares (2007) conclui que nem todas as práticas do *Scrum* se aplicavam ao contexto do projeto, além de encontrar algumas dificuldades, como a união da equipe (neste contexto, geograficamente distribuída), a participação do dono do produto e o papel do *Scrum master* (neste caso um desenvolvedor). Apresenta-se, então, a conclusão final de Soares (2007) sobre o uso do *Scrum*:

> Apesar do *Scrum* não cobrir todas as características específicas para equipes geograficamente distribuídas, foi possível fazermos uso de diversos aspectos de desenvolvimento ágil sem, no entanto, comprometer as particularidades exigidas por esses tipos de projetos (SOARES, 2007, p.8).

4.3 EXEMPLO III - CONFEROUS

O terceiro e último exemplo é um estudo de caso apresentado por Pham (2011) sobre o uso do *Scrum* e das práticas apresentadas no livro “*Scrum* em Ação” para criar a página web Conferous.

“Conferous é uma aplicação web de gerenciamento de chamadas e conferência *on-line* que permite a equipes e grupos colaborarem facilmente, simplificando a conFiguração de chamadas de conferência” (PHAM, 2011, p.244).

Neste estudo de caso, o foco descrito no livro são as práticas de coleta de requisito e desenvolvimento das *sprints*, porém também foram usadas as reuniões clássicas do *Scrum* como *Daily Scrum*, reunião de planejamento de *Sprint*, etc.

Primeiramente, Pham (2011) apresenta uma visão geral do sistema na qual são respondidas as perguntas clássicas desta fase:

Quem? Usuários de conferência corporativos.

Por quê? Facilitar o uso das chamadas de conferência.

O que? Chamadas de conferência

Onde? Estados Unidos da América.

Quando? 24x7.

Após ter a visão definida, criou-se uma floresta com três árvores e suas respectivas folhas. Neste caso usou-se o desenvolvimento horizontal, de modo que, a cada s*print*, foi desenvolvido mais de um subconjunto de funcionalidades. Isso se deve ao fato que, para se ter um *release* inicial do sistema, era necessário que o usuário conseguisse se inscrever nas funções básicas das salas de teleconferência (PHAM, 2011).

Este sistema foi dividido em dois *sprints*. Durante o primeiro s*print* foram criadas as funções básicas de usuário e sala, sendo elas: cadastro, *login*, *logout*, inclusão de sala, exclusão de sala e navegação de salas. Já no segundo foram

criadas as funções restantes do sistema: navegação de registros, edição de perfil e cancelamento de conta (PHAM, 2011).

De certo modo, este exemplo pode ser considerado como o sistema de menor porte dentre os três aqui citados.

5 XP X *SCRUM*

Até este ponto já foram apresentados diversos aspectos do *Scrum*, de outros métodos para o desenvolvimento de *software* e alguns casos reais, sendo que o *Scrum* foi à base para o desenvolvimento de projetos de *softwares*. Este capitulo é dividido em duas etapas, a primeira delas faz uma comparação entre os métodos clássicos (Cascata, Evolucionário, Prototipação, etc.) e os métodos ágeis. Em um segundo momento é comparado somente o *Scrum* como XP, que são as metodologias ágeis mais conhecidas.

O primeiro critério avaliativo é o porte do projeto, projetos com custos de trabalho inferiores a US$ 1 milhão são considerados projetos de pequeno porte, enquanto que os projetos com custos superiores a US$ 10 milhões são considerados projetos de grande porte (THE STANDISH GROUP, 2013).

Dentre o critério do porte o analisado são os sistemas de pequeno porte, pois cerca de 66% dos projetos pequenos tem sucesso, enquanto que 44% dos grandes projetos falham. Além disso os CIOs (*Chief Information Officer*) das empresas estão sempre à procura de simplificar os projetos, fazendo mais por menos. Isso se deve à tendência dos projetos de se tornarem muito grandes para atingir o sucesso. Assim, o ideal é dividir projetos muito grandes em vários projetos menores (*THE STANDISH GROUP*, 2013).

Assim o segundo critério tem relação com a facilidade do método auxiliar na quebra dos projetos em subprojetos.

O terceiro e último critério, se refere ao controle sobre o projeto, fator decisivo na hora de um projeto de desenvolvimento ter sucesso. Tudo isso é analisado na Tabela 1.

Tabela 1 - Métodos Clássicos X Métodos Ágeis

Critérios	Métodos Clássicos	Métodos Ágeis
Projetos de porte pequeno	14% dos projetos tem sucesso	45% dos projetos tem sucesso

Quebrar projetos em projetos menores	Não oferecem recursos para a divisão, pois os processos são fixos (não sofrem alterações).	Facilitam a divisão dos projetos, pois o processo não é fixo, ou seja, pode sofrer alterações.
Controle sobre o projeto	Existe um controle, porém geralmente sequencial, causando altos riscos se uma alteração é implementada no fim do projeto.	Constante, inclusive em relação às alterações, que são constantes e eles se adaptam a elas sem maiores prejuízos.

Fonte: Próprio Autor

Após a análise destes três critérios, se chega à conclusão que os modelos ágeis podem ser uma solução para os problemas de falha nos projetos. Isto, porque estes métodos possuem dois fatores cruciais para o sucesso de um projeto atualmente: o envolvimento do cliente (dono do produto) e o envolvimento dos usuários (*stakeholders*) (THE STANDISH GROUP, 2013).

Assim sendo, tudo pode ser resumido em uma comparação básica entre alguns aspectos dos dois principais métodos de desenvolvimento de *software* apresentados neste trabalho o *Scrum* e o XP.

O primeiro critério usado na comparação do XP e do *Scrum* é o modo como às tarefas são divididas dentro do projeto, o que pode impactar seu andamento se não for feito de forma adequada. O segundo ponto analisado entre ambas é o nível de envolvimento do cliente, característica base para os métodos ágeis.

O terceiro critério se trata da forma como os requisitos são priorizados durante o projeto. A quarta abordagem diz respeito ao código, como ele é preparado para alterações, fator crítico quando se trata de desenvolvimento de *software*.

Por fim o último critério se relaciona as reuniões que ocorrem em ambas as metodologias e como elas podem impactar o sistema. Esta comparação, com os

critérios pré-determinados, ocorre na Tabela 2, dividida em três colunas, o critério avaliado, XP e *Scrum*.

Tabela 2 - XP X *Scrum*

Critérios	**XP**	***Scrum***
Tarefas	Divididas entre pares de programadores.	Divididas entre todos os membros da equipe, de forma individual.
Cliente	Envolvido em todas as atividades do projeto (100% do tempo).	Envolvido nas principais reuniões e mantenedor do *backlog* de produto.
Priorizações	Requisitos e atividade priorizados em pareceria entre cliente e equipe.	Requisitos e atividades priorizados pelo dono do produto (cliente).
Código	Código simples para alterações rápidas.	Busca de código que funcione mesmo após falhas.
Reuniões	Nenhuma reunião especifica ou muito frequente.	Reuniões diárias, para o acompanhamento constante do projeto, além de reuniões mais longas pré-determinadas.

Fonte: Próprio Autor

Após a análise desta tabela, ficam claro que o *Scrum* possibilita uma flexibilidade controlada para mudanças ao longo do projeto (como ficou claro no estudo de caso ANKOS, item 4.2) devido suas reuniões diárias, além do envolvimento do cliente não ser em níveis extremos, o que aumenta as chances do cliente ter tempo para disponibilizar para o projeto.

Apesar de o Scrum apresentar algumas vantagens sobre o XP o uso combinado de ambas é algo muito vantajoso, como próprio Kninberg (2007) destaca em seu livro.

6 CONSIDERAÇÕES FINAIS

Considerando os dados apresentados nesta monografia, conclui-se que ao longo dos anos muitos modelos e métodos foram propostos com o objetivo de solucionar a chamada crise do software, que no passado se relacionava com a introdução de novos hardwares e hoje pode ser associada aos percalços encontrados ao longo do desenvolvimento de um novo sistema.

Dentre estes modelos destacou-se o Scrum, que é uma metodologia que funciona por meio de reuniões diárias (Daily Scrum) e entregas incrementais (Sprints). Estas características fornecem uma correlação entre esta metodologia e alguns outros métodos da Engenharia de Software, como é o caso do modelo espiral, que é baseado inteiramente em suas entregas incrementais. O modelo espiral, por sua, vez possui características de outros dois modelos, que são o cascata (método sequencial) e prototipação (entrega de protótipos ao longo do desenvolvimento do sistema).

Assim, conclui-se que, de certa forma, o Scrum surge de uma série de evoluções dos métodos, que acabam por se interligar com algumas de suas características.

Para o Scrum, tempo é algo muito importante para que os Sprints sejam entregues dentro do prazo, que foram previamente estipulados em cada Sprint backlog, e é controlado através do gráfico de burndown. Assim como todo método ágil, no Scrum, a participação do cliente é muito importante, o qual é representado como o dono do produto, responsável por priorizar as atividades que devem entrar nas Sprints de acordo com a prioridade para suas entregas.

Dentre todos os aspectos do Scrum, o que mais o destaca, e corresponde à sua maior vantagem sobre os outros métodos, são suas reuniões diárias, que controlam o andamento do projeto ao ponto de corrigir, com uma maior facilidade, os problemas que possam atrasar ou afetar o sucesso do projeto de qualquer modo.

Uma questão importante, que pode ser vista como uma possível desvantagem do Scrum é o fato que, para seu sucesso, o ideal é se trabalhar com equipes pequenas (entre 5 e 9 pessoas). Porém, devido a pesquisas realizadas pelo

The Standish Group (2013), ficou claro que isso não é um empecilho, já que recomenda-se que em casos de grandes projetos, o ideal é dividí-lo em projetos menores, pois, 66% dos projetos pequenos tem sucesso, enquanto apenas 7% dos grandes projetos não têm nenhuma alteração ou problema em seu percurso.

Além disso, foi levantado em pesquisas do The Standish Group (2013), que nos últimos anos 45% dos projetos que usaram métodos ágeis gastaram cerca de US$ 1 milhão a menos em seus custos.

Atrelado às questões acima citadas, pode-se afirmar que o Scrum é uma das metodologias mais vantajosas para o desenvolvimento de software, visto que ele é um método ágil, entre os que estão cada vez mais sendo adotados, e é uma metodologia que, com seus princípios básicos, possibilita modos de simplificar a documentação e o processo de criação de softwares sem perder totalmente a essência de métodos tradicionais.

7 REFERÊNCIAS BIBLIOGRÁFICAS

ANAND, Bachan. **Keep your Agile Retrospectives fresh and lively!** 2011. Disponível em: <http://agile.conscires.com/2011/06/20/keep-your-agile-retrospectives-fresh-and-lively/>. Acesso em: 26 jun. 2014.

ARAÚJO, José. **Paradigmas da Engenharia de *Software* [Parte 2].** 2011. Disponível em: <http://centraldaengenharia.wordpress.com/2011/02/09/paradigmas-prototipacao/>. Acesso em: 25 set. 2014.

ARAÚJO, José. **Paradigmas da Engenharia de *Software* [Parte 3]**. 2011. Disponível em: <https://centraldaengenharia.wordpress.com/2011/02/12/paradigmas-espiral/>. Acesso em: 19 jun. 2014.

CAMPOS, Ester Lima de. **Burndown chart**: Mede o progresso da *sprint* e dá indicativos do processo de trabalho da equipe. 2012. Disponível em: <http://blog.my*Scrum*half.com/2012/01/burndown-chart-medindo-o-progresso-de-sua-s*print*-e-trazendo-indicativos-do-processo-de-trabalho-da-equipe/>. Acesso em: 25 jun. 2014.

CASTRO, Flavio Steffens de. **Gráfico Burndown**: Sugestão de uso. 2009. Disponível em: <http://www.agileway.com.br/2009/08/18/grafico-burndown-sugestao-de-uso/>. Acesso em: 25 jun. 2014.

COHN, Mike. **Desenvolvimento de *Software* com *Scrum***: Aplicando Métodos Ágeis com Sucesso. Trad. Aldir José Coelho Corrêa da Silva. Porto Alegre/RS: Artmed, 2011. 496 p.

DESENVOLVIMENTO ÁGIL. ***Scrum***. 2013. Disponível em: <http://desenvolvimentoagil.com.br/*Scrum*/>. Acesso em: 03 abr. 2014 (Projeto Open Source feito pela comunidade de desenvolvimento ágil do Brasil)

FURTADO, André. Pontas de Iceberg do Caos no Desenvolvimento de *Software*. **Microsoft**, São Paulo. Disponível em: <http://www.microsoft.com/brasil/msdn/Tecnologias/Carreira/Desenvolvimento*Software*.mspx>. Acesso em: 25 abr. 2014.

KNIBERG, Henrik. ***Scrum* e XP direto das Trincheiras**: Como nós fazemos *Scrum*. Trad. Vários Tradutores. São Paulo: Infoq, 2007. 148 p (Desenvolvimento de *Software* Corporativo).

MAROTA, Bruno. **Engenharia de *Software* Para Concursos**: Processos e Modelos de Processos de *Software*. 2012. Disponível em: <http://brunomarota.blogspot.com.br/2012/04/engenharia-de-*software*-para-concursos_19.html>. Acesso em: 19 jun. 2014.

MAZZOLA, VitÓrio Bruno. **Engenharia de *Software***. 2010. Disponível em: <http://jalvesnicacio.files.wordpress.com/2010/03/engenharia-de-*software*.pdf>. Acesso em: 25 set. 2014.

NORD, Magnus. **Zen of *Scrum***: *Sprint* Planning. 2012. Disponível em: <http://www.devoteddeveloper.com/2012/09/zen-of-*Scrum*-*sprint*-planning.html>. Acesso em: 24 jun. 2014.

PHAM, Andrew; PHAM, Phuong-van. ***Scrum* em Ação**: Gerenciamento e Desenvolvimento Ágil de Projetos de *Softwares*. Trad. Edgard B. Damiani. São Paulo: Novatec, 2011. 288 p.

PRESSMAN, Roger S.. **Engenharia de *Software***. Trad. Rosângela Ap. d. Penteado. 6. ed. São Paulo: Pearson, 2010. 762 p.

SCHATZ, Bob. **The Sprint Review**: Mastering the Art of Feedback. 2009. Disponível em: <http://www.*Scrum*alliance.org/community/articles/2009/april/the-*sprint*-review-mastering-the-art-of-feedback>. Acesso em: 26 jun. 2014.

SOARES, Felipe S. Furtado et al. Adoção de *SCRUM* em uma Fábrica de Desenvolvimento Distribuído de *Software*. **Universidade Federal de Pernambuco**, Recife/PE, 2007 Disponível em: <http://afinaimagem.googlecode.com/svn/trunk/PGP/Rodrigo/Documentacao*Scrum*/adocao-de-*Scrum*-em-uma-fabrica-de-desenvolvimento.pdf>. Acesso em: 01 abr. 2014.

SOARES, Michel dos Santos. Metodologias Ágeis Extreme Programming e *Scrum* para o Desenvolvimento de *Software*. **Revista Eletrônica de Sistemas de Informação**, Curitiba/PR, v. 3, n. 1, 2004. Anual. Disponível em: <http://revistas.facecla.com.br/index.php/reinfo/article/viewArticle/146>. Acesso em: 01 abr. 2014.

SOMMERVILLE, Ian. **Engenharia de *Software***. Trad. Vários Autores. 8. ed. São Paulo: Pearson, 2007. 568 p.

SOUSA, Marcos Morais de. **ENGENHARIA DE *SOFTWARE***. 2012. Disponível em: <http://marcosmoraisdesousa.blogspot.com.br/2012/04/engenharia-de-*software*.html>. Acesso em: 03 jun. 2014.

THE STANDISH GROUP INTERNATIONAL. **Chaos Manifesto 2013**. p.4-6, 2013. Disponível em: <http://versionone.com/assets/img/files/CHAOSManifesto2013.pdf>. Acesso em: 25 abr. 2014.