REVISTA
CIÊNCIA E SUSTENTABILIDADE
ISSN 2447-4606

Metodologias ágeis para o desenvolvimento de softwares

Agile methodologies for software development

Thiago Bessa
Universidade Federal do Cariri (UFCA) e Universidade de Lisboa. Doutorando em Tecnologias da Informação e Comunicação com ênfase em Technology Enhanced Learning and Societal Challenges pelo Instituto de Educação da Universidade de Lisboa. E-mail: thiago@bessapontes.com

Daniel Dias Branco Arthaud
Universidade Federal do Ceará (UFC). Graduado em Ciências da Computação pela Universidade Federal do Ceará. E-mail: daniel@vizir.com.br

Recebido em: 01/10/2018 I Aceito em: 06/02/2018

RESUMO

O desenvolvimento de software é uma atividade complexa. Devido a essa complexidade a Engenharia de Software foi criada na década de sessenta do século XX. Ela tem por objetivo aumentar a produtividade no desenvolvimento e melhorar a qualidade do produto gerado. Diversas metodologias e práticas surgiram com a intenção de direcionar o desenvolvimento. Inicialmente vieram as metodologias chamadas tradicionais: cascata, prototipação, incremental. Elas traziam junto consigo muita rigidez e cada passo delas gerava diversos artefatos e documentações, mesmo assim os resultados entregues pelos projetos geravam resultados abaixo do esperado. Com isso, metodologias ágeis surgiram com a proposta de fornecer agilidade de resposta e flexibilidade de adaptação no desenvolvimento trazendo um diferencial competitivo através de velocidade e qualidade dos resultados. Esta revisão técnica tem como objetivo apresentar um mecanismo para desenvolvimento ágil de software de modo a atender com mais rapidez e qualidade, reduzindo dessa forma o consumo desnecessário de recursos, auxiliando na promoção de um sistema sustentável. Compreendeu-se que essas técnicas ágeis trouxeram uma grande evolução na troca de experiência, comunicação, transmissão de conhecimento, confiança das pessoas, confiança do cliente. E isso faz com que a produtividade da equipe cresça e também faz com que a satisfação do cliente seja maior.

Palavras-chave: Software. Desenvolvimento. Metodologias. Práticas ágeis.

REVISTA CIÊNCIA E SUSTENTABILIDADE
ISSN 2447-4606

ABSTRACT

Software development is a complex activity. Due to this complexity of Software Engineering was created in the 60 years of the 20th century. She aims to increase productivity in development and improve the quality of the product generated. Several methodologies and practices arose with the intention of driving the development. Initially came from the traditional calls methodologies: cascade, prototyping, incremental. They brought along a lot of stiffness and every step of which generated several artifacts and documentation, yet the results delivered by the projects generated results below expectations. With that, agile methodologies have emerged with the proposal to provide agility and flexibility of adaptation in the development bringing a competitive advantage through speed and quality of the results. This technical review aims to present a mechanism for Agile software development in order to meet with more speed and quality, reducing in this way the unnecessary consumption of resources, assisting in the promotion of a sustainable system. Understand that these agile techniques have brought a great evolution in the exchange of experience, communication, transmission of knowledge, trust of people, customer trust. And that makes the team grow productivity and also makes customer satisfaction is greater.

Keywords: Software. Development. Methodologies. Agile practices.

1 INTRODUÇÃO

Este trabalho tem por objetivo apresentar uma revisão técnica de algumas práticas ágeis e mostrar como aplicá-las durante o desenvolvimento de software. Ao realizarmos o desenvolvimento de um software, nos deparamos com vários problemas que muitas vezes nos impedem de entregá-lo dentro do prazo ou orçamento estipulado, ou até mesmo com qualidade inferior à que gostaríamos. O que resulta num alto consumo de recursos, quer sejam a nível técnico, financeiro ou humano. A revisão aqui apresentada, justifica-se como uma forma de corroborar com o debate acerca dos estudos, que levarão a melhores práticas de desenvolvimento de software.

REVISTA CIÊNCIA E SUSTENTABILIDADE
ISSN 2447-4606

Para muitos, desenvolver um software é uma incógnita. Não existe um plano a ser seguido, os cronogramas são feitos com suposições, as mudanças são feitas tardiamente, dificultando-as cada vez mais.

Através dos tempos, foram criadas metodologias para fazer com que este processo seja algo disciplinado. Mas também se tornou algo muito burocrata, onde muito trabalho inútil é realizado.

O manifesto ágil chegou para encontrar um meio-termo entre esses dois mundos, criando apenas documentações necessárias e focando mais no produto funcional. Como afirmam Fadel e Silveira (2010), as metodologias ágeis são iterativas e incrementais, resultando em um produto desenvolvido com base na melhoria continua, e como o cliente participa de todo o projeto, a sua satisfação normalmente é garantida.

No processo de criação de software, os requisitos estão sendo alterados constantemente, por isso, os métodos ágeis, trabalham com iterações, onde são apresentadas partes do sistema para o cliente para aprovação, tornando o processo bastante adaptativo.

Implantar uma metodologia ágil em uma equipe de desenvolvimento de software que frequentemente tem problemas com cronogramas pode ser uma solução bem útil. Com o decorrer do tempo e prática, o processo ganha agilidade nas entregas dos produtos.

Mas como qualquer metodologia, o processo precisa estar em constante alteração e, assim, sendo melhorado sempre. O problema em questão pretende entender como os métodos ágeis podem melhorar a produtividade da equipe de desenvolvimento e melhorara a qualidade do software gerado.

Entende-se que com o uso de práticas e metodologias ágeis, conseguiremos fazer um desenvolvimento mais voltado a colaboração entre os desenvolvedores, a flexibilidade de adaptação a mudanças e a agilidade nas respostas. Deste modo o objetivo deste estudo está em apresentar algumas práticas utilizadas nas metodologias ágeis. E mostrar como aplicá-las no desenvolvimento de software.

REVISTA CIÊNCIA E SUSTENTABILIDADE
ISSN 2447-4606

2 MÉTODOS ÁGEIS

2.1 ENGENHARA DE SOFTWARE

O termo Engenharia de Software foi criado na década de 60 e oficializado em 1968, na conferência sobre Engenharia de Software da OTAN, organizada para discutir a chamada "crise do software". Durante essa crise, percebeu-se que novas técnicas e métodos eram necessários para controlar a complexidade inerente aos novos sistemas de software.

Dentre as definições de Engenharia de Software, podemos destacar as seguintes:

a) Criação e utilização de sólidos princípios de engenharia a fim de obter software de maneira econômica, que seja confiável e que trabalhe eficientemente em máquinas reais (BAUER, 1969 *apud* NAUR; RANDELL, 1969);
b) Disciplina da engenharia que se ocupa de todos os aspectos da produção de um software, desde os estágios iniciais de especificação até a manutenção desse sistema, depois que ele entrou em operação (SOMMERVILLE, 2003);
c) Estudo e aplicação de uma abordagem sistemática, disciplinada e quantificável para o desenvolvimento, operação e manutenção de software (IEEE, 1993);
d) Conjunto de métodos, técnicas e ferramentas necessárias à produção de software de qualidade para todas as etapas do ciclo de vida do produto (KRAKOWIAK,1985 *apud* CHAGAS, 2008).

Das definições acima podemos concluir que Engenharia de Software é um conjunto de métodos, técnicas, padrões e ferramentas aplicados ao processo de desenvolvimento, com intenção de produzir softwares de alta qualidade e confiabilidade.

De acordo com Pressman (2007), a Engenharia de Software abrange três componentes básicos: métodos, que proporcionam os detalhes de como construir

REVISTA CIÊNCIA E SUSTENTABILIDADE
ISSN 2447-4606

um software, tratando de planejamento, estimativas, análises de requisitos e arquitetura, entre outros; ferramentas, que sustentam cada um dos métodos; e procedimentos, que definem a sequencia em que os métodos são aplicados e fazem o elo entre os métodos e as ferramentas.

Segundo Sommerville (2003), houve um grande progresso desde 1968 e o desenvolvimento da engenharia de software melhorou consideravelmente o software produzido. Métodos eficazes de especificação, modelagem e implementação de software foram desenvolvidos.

Segundo Ghezzi, Jazayeri e Madrioli (1991), o grande objetivo da engenharia de software é viabilizar maior produtividade na construção de aplicações e qualidade dos artefatos de software.

2.2 METODOLOGIAS DE DESENVOLVIMENTO DE SOFTWARE

Dentre as definições de Metodologias de Desenvolvimento de Software (Processos de software) podemos ressaltar as seguintes:

a) Conjunto de atividades e resultados associados que levam à produção de um software (SOMMERVILLE, 2003);
b) Framework para as tarefas que são necessárias para a construção de software de alta qualidade (PRESSMAN, 2007);
c) Conjunto de passos parcialmente ordenados, constituídos por atividades, métodos, práticas e transformações, usado para atingir uma meta. Esta meta geralmente está associada a um ou mais resultados concretos finais, que são produtos da execução do processo (PÁDUA, 2001);
d) Conjunto de atividades, métodos, práticas e transformações que são empregados para desenvolver e manter softwares e produtos associados (sendo estes, planos de projeto, documentos de projeto, projetos de software, código, casos de teste e manual do usuário). Atuam neste processo, ferramentas e modelos que automatizam e facilitam os trabalhos dos envolvidos (analista, programador, gerente, cliente e outros), possibilitando o

REVISTA CIÊNCIA E SUSTENTABILIDADE
ISSN 2447-4606

desenvolvimento de software com qualidade, obedecendo a prazo e orçamento determinados (VASCONCELOS, 2005).

Analisando as definições acima, podemos concluir que processos de software são as diversas fases necessárias para produzir e manter um produto de software. Requerem a organização lógica de diversas atividades técnicas e gerenciais envolvendo agentes, métodos, ferramentas, artefatos e restrições que possibilitam disciplinar, sistematizar e organizar o desenvolvimento e manutenção de produtos de software. Diversos foram os modelos e processos que surgiram com tais perspectivas, fornecendo descrições de como o software deveria ser criado e mantido (ORTIGOSA, 1995).

Existem vários processos de software definidos na literatura da Engenharia de Software. É comum mesmo algumas organizações criarem seu próprio processo ou adaptar algum processo à sua realidade. Mesmo existindo diversas metodologias de desenvolvimento, há atividades que podemos citar como sendo fundamentais para qualquer metodologia: Especificação de Software, Projeto e Implementação de Software, Validação de Software e Evolução de Software.

2.2.1 Metodologias Tradicionais

As metodologias tradicionais são também chamadas de pesadas ou orientadas a documentação. Essas metodologias surgiram em um contexto de desenvolvimento de software muito diferente do atual (ROYCE,1970). Podemos chamar de metodologias tradicionais de desenvolvimento de software as metodologias de desenvolvimento em cascata (do inglês *waterfall*) e processos de desenvolvimento que nela se baseiam (TELES, 2004).

O termo tradicional, nesse contexto, é utilizado para identificar metodologias ou processos que são baseados na metodologia clássica (Modelo em Cascata). O que melhor caracteriza estas metodologias é a separação bem rígida das fases de projeto, que consistem em: Levantamento de Requisitos, Análise, Desenho, Implementação, Testes e Implantação. Cada fase tem suas especificidades e possuem entre si interdependência, isto é, a próxima fase só começa quando a

REVISTA CIÊNCIA E SUSTENTABILIDADE
ISSN 2447-4606

anterior estiver pronta. Elas seguem também algumas premissas, que são, linearidade, determinismo, especialização, foco na execução e crescimento exponencial do custo de alteração.

Nas metodologias tradicionais, o desenvolvimento se divide em duas grandes partes, a concepção e a construção. É na fase de concepção que se tem a maior interação com o cliente e é gasto um grande tempo identificando as necessidades para evitar que apareçam novas necessidades ou durante a fase de construção. Essa fase de concepção é considerada a mais importante do projeto, pois dependendo dela, este terá sucesso ou não.

Se surgirem imprevistos durante a construção acarretará em retrabalho e aumento no tempo de projeto e isso causará impactos no preço, prazo e qualidade, o que pode gerar conflitos com o cliente, que além de sofrer com os impactos possui uma grande possibilidade de o sistema não atender suas necessidades.

Durante a construção não há interação com o cliente, impedindo que sejam feitas correções ou modificações no escopo do projeto. As metodologias tradicionais, devem ser utilizadas apenas em situações em que os requisitos são estáveis e previsíveis. Em geral isso não acontece, pois, os requisitos para desenvolvimento de software são altamente mutáveis.

"A percepção que os usuários têm de suas necessidades também evolui à medida que eles conhecem o sistema. É difícil compreender o valor de uma determinada funcionalidade até que ela seja efetivamente usada, principalmente porque não se pode requerer de um usuário comum a mesma capacidade de abstração que um desenvolvedor possui ao olhar um conjunto de requisitos" (OLIVEIRA, 2003 *apud* FERREIRA; LIMA, 2006).

2.2.2 Metodologias Ágeis

As metodologias ágeis constituem uma nova classe de metodologias de desenvolvimento de software criada para atender à crescente pressão do mercado por processos mais ágeis e leves, com ciclos de desenvolvimento cada vez mais curtos (ABRAHAMSSON, 2003).

REVISTA CIÊNCIA E SUSTENTABILIDADE
ISSN 2447-4606

Ludvig e Reinert (2007), citam que elas surgiram como uma reação às metodologias pesadas (FOWLER, 2005) e tiveram como principal motivação criar alternativas para o Modelo Tradicional em Cascata (HILMAN, 2004).

Carvalho, Abrantes e Cameira (2011) falam que estas metodologias, focam nos indivíduos e na interação entre eles, no funcionamento do software, na colaboração com o cliente e na resposta rápida às mudanças. Isto é alcançado através de ciclos iterativos de desenvolvimento de sistemas com objetivos de curto prazo, tonando possível a adaptação dos requisitos até o momento em que entram em fase de desenvolvimento.

Segundo Fowler (2005), as metodologias ágeis de desenvolvimento são conjuntos de práticas e métodos de desenvolvimento, criados e desenvolvidos ao longo das últimas duas décadas, que têm por objetivo tornar o desenvolvimento de software rápido, com custo controlável e melhorar a qualidade do software.

Para ele, a maior diferença entre as metodologias tradicionais e as metodologias ágeis são as premissas de que: **Metodologias ágeis são adaptativas ao invés de predeterminantes**. Metodologias tradicionais tentam prever detalhes do processo de desenvolvimento no início do projeto, e acabam tendo alguns problemas quando são necessárias mudanças no projeto, como atrasos na entrega do projeto.

Nas metodologias ágeis a premissa é de que sempre haverá mudanças no projeto que não podem ser previstas em seu início, e essas mudanças são bem-vindas, estando todos os envolvidos no projeto preparados para elas. E as **Metodologias ágeis são orientadas a pessoas e não a processos**. Metodologias tradicionais definem um processo que irá funcionar bem, independente de quem os utilize. Métodos ágeis afirmam que nenhum processo jamais será equivalente à habilidade da equipe de desenvolvimento. Portanto, o papel do processo é dar suporte à equipe de desenvolvimento e seu trabalho.

Uma importante característica presente em todos os Métodos Ágeis é o reconhecimento do desenvolvimento de software como um processo empírico (WILLIAMS; COCKBURN, 2003). Existem dois tipos básicos de processos: os processos definidos e os processos empíricos (SCHWABER; BEEDLE, 2002).

Em linhas gerais, os processos definidos são aqueles onde é possível a criação de um plano a priori que direcionará todo o projeto, produzindo sempre o mesmo

REVISTA CIÊNCIA E SUSTENTABILIDADE
ISSN 2447-4606

resultado. Processos empíricos, por outro lado, são aqueles onde há grande incerteza, pesquisa e descoberta, tornando impraticável a definição completa do projeto num único momento (SCHWABER; BEEDLE, 2002).

2.3 MANIFESTO ÁGIL

O termo "Metodologias Ágeis" tornou-se popular em 2001 quando um grupo de especialistas em processos de software representando diversos métodos ágeis se reuniu para discutir maneiras de melhorar o desempenho de seus projetos. Nessa reunião eles perceberam que em suas experiências anteriores sempre havia um conjunto de princípios comuns que, quando respeitados, os projetos davam certo. Dessa reunião surgiu a aliança ágil e estabeleceu-se o Manifesto ágil com suas filosofias, valores e princípios.

Das definições existentes de métodos ágeis, a mais aceita é: Métodos ágeis são um conjunto de práticas que seguem os princípios do Manifesto Ágil (BECK *et al*., 2001).

Os princípios básicos de métodos ágeis compreendem honestidade ao código de trabalho, eficácia das pessoas que trabalham em conjunto e foco no trabalho em equipe. Assim, o grupo de desenvolvimento, incluindo desenvolvedores e representantes do cliente, deve ser bem informado, competente e autorizado a considerar o eventual ajuste das necessidades emergentes durante o processo de ciclo de vida do desenvolvimento. Isto significa que os participantes estão preparados para fazer mudanças, e que também os contratos existentes são formados com as ferramentas que suportam e permitem que essas melhorias sejam feitas (LEITÃO, 2010).

Ludvig e Reinert (2007) disseram que o Manifesto Ágil não é contra os modelos adotados pela abordagem tradicional. Os métodos ágeis são fundamentados no desenvolvimento incremental (SOMMERVILLE, 2003), incluem técnicas utilizadas na Engenharia de Software, porém não seguem o padrão proposto pelas metodologias tradicionais (HIGHSMITH, 2000).

REVISTA CIÊNCIA E SUSTENTABILIDADE
ISSN 2447-4606

Portanto, não rejeitam os processos e ferramentas, a documentação, a negociação de contratos ou o planejamento, apenas mostram que eles têm importância secundária comparado com os indivíduos e iterações, como o software estar executável, com a colaboração e as respostas rápidas a mudanças e alterações.

Estudos como os de Hamed *et al.* (2013) apontam os métodos ágeis mais utilizados, que destes escolhemos o *Scrum*, *Extreme Programming* e o *Lean Software Development* para apresentarmos neste trabalho, pois além de possuírem base comum de suas concepções, como Engenharia e Gestão, têm apresentado resultados satisfatórios quando aplicados em times de todos os tamanhos (HAMED *e. al*., 2013).

2.3.1 *Scrum*

O *Scrum* é uma metodologia ágil de desenvolvimento de software, criado na década de 1990 por Jeff Sutherland, com base em conceitos tradicionais da engenharia de produção como *Lean* e a Teoria das Restrições Explorada por Takeuchi e Nonaka (1986) no artigo "*The new product development game*", no qual descrevem as vantagens da utilização de times pequenos, multidisciplinares e auto gerenciáveis no desenvolvimento de produtos.

O objetivo do *Scrum* é entregar a maior qualidade de software possível dentro de uma série de pequenos intervalos de tempo fixo, chamados *Sprints*, que tipicamente duram menos de um mês (SUTHERLAND *et al*., 2000). Ou seja, o objetivo maior do *sprint* é entregar o máximo valor de negócio possível no menor tempo.

Schwaber e Sutherland (2011), falam que o *Scrum* é fundamentado nas teorias empíricas de controle de processo, ou empirismo. O empirismo afirma que o conhecimento vem da experiência e de tomada de decisões baseadas no que é conhecido. Ele se apoia em três pilares: (i) **Transparência** - Aspectos significativos do processo devem estar visíveis aos responsáveis pelos resultados; (ii) **Inspeção** - Os usuários *Scrum* devem, frequentemente, inspecionar os artefatos *Scrum* e o progresso em direção ao objetivo, para detectar indesejáveis variações. A inspeção não pode sobrepor a execução das atividades; (iii).**Adaptação** - Se for visto que algum aspecto do processo se desviou dos limites aceitáveis e que o produto será

REVISTA CIÊNCIA E SUSTENTABILIDADE
ISSN 2447-4606

inaceitável, o processo deve ser reajustado para produzir o que era esperado. E isso deve ser feito o mais breve possível para minimizar os desvios.

Há 4 (quatro) eventos principais onde se pode fazer a inspeção e adaptação. São eles: Reunião de planejamento da Sprint, Daily *Scrum* (*Daily Meeting*), Reunião da revisão da Sprint e Retrospectiva da Sprint.

Os valores aplicados no *Scrum* estão diretamente relacionados aos descritos no manifesto ágil. Eles foram descritos por Sliger e Broderick (2008) em seu livro da seguinte forma:

a) Compromisso: esteja disposto a assumir o compromisso de uma meta. *Scrum* fornece a toda pessoa autoridade necessária para cumprir os seus compromissos;
b) Foco: faça o seu trabalho. Centre todos os seus esforços e competências em fazer o trabalho a que lhe foi empenhado. Não se preocupe com nada além disso;
c) Abertura: o *Scrum* mantém tudo sobre um projeto visível para todos;
d) Respeito: os indivíduos são moldados por seus antecedentes e as suas experiências. É importante respeitar os diferentes povos que compõem uma equipe;
e) Coragem: tenha a coragem de se comprometer, de agir, de estar aberto e aguardar o respeito.

O *Scrum* possui três papéis bem definidos que são apresentados no Quadro 1 a seguir:

Quadro 1 - Papéis da metodologia *Scrum*

Termo	Definição
Scrum Master	É o responsável por garantir que tudo vai ocorrer de acordo com as regras do *Scrum* (práticas, valores) e que o projeto tenha o progresso esperado. Ele também é responsável por eliminar ou mitigar qualquer impedimento que possa atrapalhar a produtividade da equipe.
Product Owner	É o responsável por gerenciar as atividades que serão desenvolvidas (*Product Backlog*). Ele toma as decisões quanto a priorização das atividades e participa da estimativa do esforço de desenvolvimento.
Scrum Team	É a equipe de desenvolvimento propriamente dita.

Fonte: elaborado pelos autores baseado em Vieira (2014).

REVISTA CIÊNCIA E SUSTENTABILIDADE
ISSN 2447-4606

O *Scrum* utiliza ainda algumas práticas de gestão para diminuir o caos gerado pela imprevisibilidade e complexidade dos processos. O Quadro 2 apresenta tais informações.

Quadro 2 - Práticas de gestão metodologia *Scrum*

Termo	Definição
Product Backlog	É a definição de tudo que é necessário para o produto final baseado no conhecimento atual. É uma lista de prioridades que é constantemente atualizada pelo *Product Owner*.
Effort Estimation	É o momento em que o time juntamente com o *Product Owner* e o *Scrum* Master, define o tempo estimado para cada item do *Product Backlog* e verificam se as informações existentes sobre o item são suficientes.
Sprint	É o período de desenvolvimento, de tempo predeterminado, para as atividades que foram selecionadas para aquele sprint.
Sprint Planning Meeting	É uma reunião organizada em duas fases. A primeira fase participa todos os envolvidos da equipe além de clientes, usuários e gerentes. Nesta fase são decididas metas e funcionalidades do sprint. Na segunda fase o *Scrum* Master e o time se reúnem para definir como serão implementadas as atividades no sprint.
Sprint Backlog	É a lista de atividades que foram escolhidas para ser desenvolvida no Sprint. Esta lista não muda até que termine o Sprint. Quem deve garantir que isso aconteça é o *Scrum* Master.
Daily Meeting	São reuniões diárias organizadas entre a equipe e conduzidas pelo *Scrum* Master com objetivo de controlar o andamento do projeto e verificar a existência de algum impedimento.
Sprint Review Meeting	Reunião entre a equipe e o *Scrum* Master, para apresentar os resultados daquele Sprint para o *Product Owner*, gerentes e clientes. Como resultado dessa reunião pode ser que o *Product Backlog* seja modificado, aumentando ou modificando alguns itens já existentes.

Fonte: Fonte: elaborado pelos autores baseado em Vieira (2014).

2.3.2 XP - *Extreme Programming*

Extreme Programming é uma metodologia de desenvolvimento de software que combina rapidez, produtividade, qualidade de forma simples e que atende as necessidades do cliente.

É uma metodologia voltada para o desenvolvimento onde os requisitos se modificam constantemente. Ela busca gerar o máximo de valor possível para o cliente num curto espaço de tempo. E durante esse espaço de tempo o cliente tem a possibilidade de analisar o produto recebido e ver se é realmente o desejado após uso dele.

REVISTA CIÊNCIA E SUSTENTABILIDADE
ISSN 2447-4606

Segundo Beck (2004), seu criado, a *Extreme Programming* é uma metodologia ágil para equipes médias e pequenas, onde os requisitos para o desenvolvimento de software são vagos e em constante mudança.

Extreme Programming traz em sua base valores e práticas que sempre procuram garantir ao cliente versatilidade e satisfação com o produto final. Os valores do XP são definidos como: (i) **comunicação**: a comunicação é um ponto extremamente importante para o bom andamento de um projeto utilizando XP, a conversação entre o cliente e a equipe tem que ser bem clara e objetiva; (ii) **simplicidade**: tudo no XP deve ser feito da forma mais simples possível. Faça a coisa da forma mais simples que funcione. E dessa forma o cliente terá a funcionalidade rapidamente da forma desejada; (iii) **feedback**: o cliente está sempre aprendendo com o produto que está recebendo durante o projeto. Com isto ele consegue tomar decisões de priorização de atividades, consegue verificar se o que está sendo feito é realmente o que ele necessitava; (iv) **coragem**: algumas práticas no XP, tais como testes unitários, integração contínua, programação em par entre outras, aumentam a confiança da equipe e ajudam a ter coragem para: melhorar o código que está funcionando para torná-lo mais simples (refatorar); investir tempo no desenvolvimento de testes; mexer no design em estágio avançado; pedir ajuda aos que sabem mais; e abandonar processos formais e ter o projeto e a documentação em forma de código.

O *Extreme Programming* possui 11 práticas que a tornam uma metodologia mais próxima do cliente, e que a equipe avance com celeridade na resolução dos problemas. A primeira prática é o ***Planning Game***, que trata de uma reunião entre cliente e desenvolvedores feita no início de um ciclo para priorização e atividades. Na visão do XP, o cliente deve estar presente no dia a dia do projeto, para poder trazer a equipe uma melhor compreensão das funcionalidades que estão sendo desenvolvidas. Esta prática é definida como: **cliente disponível ou presente**.

O ***Peer Programming*** ou **Programação em Duplas** é o desenvolvimento das atividades feito em duplas. Conseguindo, dessa forma, otimizar a produção de código com qualidade. Há também o **Código Coletivo** onde todo o código produzido por

REVISTA CIÊNCIA E SUSTENTABILIDADE
ISSN 2447-4606

qualquer dupla pertence a equipe inteira, dessa forma qualquer outra dupla pode modificar o código feito por outros.

A **Metáfora** é utilizada para facilitar a comunicação com o cliente. A equipe busca ter uma maior compreensão do contexto através de analogias com outros assuntos que são mais conhecidos. Assim como o ***Stand up Meeting*** que são reuniões rápidas e que são feitas de pé com o objetivo de compartilhar o andamento do projeto e problemas e soluções encontradas. Disseminando conhecimento entre os membros da equipe.

Os **Releases Curtos** são conjuntos de funcionalidades que geram valo ao cliente. Devem ser tão curtas quando possíveis para que entregue o máximo de valor ao cliente com o mínimo de tempo e esforço possível. Para modificar o código que está funcionando, usa-se o ***Refactoring***. Ela busca tornar o código mais simples, mais eficiente, melhorar o Design do projeto entre outras coisas. Em geral, sistemas que estão sendo produzidos, são divididos em partes e entregues a equipe para o desenvolvimento. O XP prega que a integração e testes entre essas partes devem ser feitas diversas vezes durante o dia, conhecido como a **Integração Contínua**.

Por fim apresentam-se o **Padrão de Desenvolvimento** necessário pra que seja adotado algum padrão de código para conseguir que a comunicação seja feita através dele, e o **Desenvolvimento guiado por testes** que para todo código ser gerado deve existir um teste automatizado, que o verifique. Dessa forma garantindo o funcionamento da funcionalidade.

2.3.3 *Lean Software Development*

Lean Software Development é uma aplicação dos princípios e práticas de *Lean* aplicados no contexto de Desenvolvimento de Software.

Lean é uma metodologia de produção que tem foco em eliminar os desperdícios, aumentar a velocidade de processos e a qualidade do produto final. Ele considera desperdício qualquer gasto de recurso com outro objetivo que não seja a criação de valor para o cliente final.

Pantalião (2009), trouxe em seu texto, "O que é Lean?" Algumas definições bem esclarecedoras:

REVISTA CIÊNCIA E SUSTENTABILIDADE
ISSN 2447-4606

a) Allan Shalloway disse: “Lean é uma abordagem para entregar valor mais rapidamente para seus clientes focando em melhorar o workflow dos produtos sendo entregues. Em particular, times devem sempre reduzir atrasos, que estão ligados a desperdício e baixa qualidade";
b) Mary Poppendieck complementou citando a opinião de John Shook sobre o assunto: “Como sabemos, muito e se não a maior parte, se não quase tudo do "Lean" é essencialmente procurarmos meios de executar o P-D-C-A em tudo que fazemos";
c) Kent Beck resumiu muito bem sua opinião assim: "Ao aplicar o desenvolvimento Lean eliminamos tempo e esforço desperdiçado. O que sobra é produtividade. Quando isto tiver sido feito, estará pronto para ver o próximo nível de esforço e tempo desperdiçado. Desta forma, o ciclo de melhoria nunca termina";
d) Mary Poppendieck comenta novamente e faz algumas observações na mesma linha de Kent Beck. Ela estava pensando sobre o que John Shook queria dizer e conclui: "Lean é sobre sempre melhorar o que você é agora através de testes e métricas dos resultados. Então é sobre nunca estar confortável com o status quo, e sempre aprender como melhorar utilizando o método científico."

Os clientes de um sistema de software irão perceber a integridade de um sistema no momento em que ele resolve o seu problema de uma forma fácil de usar e rentável. Não importa se o problema é mal compreendido, se existiram mudanças ao longo do tempo, ou é dependente de fatores externos, um sistema com integridade percebido é que resolve o problema de forma eficaz (POPPENDIECK; POPPENDIECK, 2003).

Para a implementação do *Lean*, é necessária uma mudança de cultura na organização. Não basta chegar com uma nova série de regras e achar que automaticamente a organização se tornará enxuta. Esta mudança na forma de pensar da equipe é certamente o maior desafio na execução desta iniciativa.

Princípios do *Lean* aplicados ao Desenvolvimento de Software:

REVISTA CIÊNCIA E SUSTENTABILIDADE
ISSN 2447-4606

I. Elimine os desperdícios

"Desperdício é tudo aquilo que não agrega valor ao cliente" Taiichi Ohno.

Devemos eliminar toda e qualquer coisa que não traga valor ao cliente. É muito difícil entender o conceito real de valor e identificá-lo. Alguns exemplos de desperdício são: Funcionalidades extras; Documentação produzida apenas para "regularizar" o processo; Funcionalidades incompletas ou com defeito.

É essencial identificar a necessidade do cliente e entregá-la da forma mais rápida possível e com qualidade.

II. Amplifique o aprendizado

Desenvolvimento de Software pode ser considerado um exercício de constantes descobertas. Não se espera que um desenvolvedor consiga gerar o melhor código de primeira. Ele vai "experimentando" e aprendendo coisas novas, e conforme ganha experiência vai surgindo um software melhor. A melhor forma de melhorar ambiente de desenvolvimento é amplificando o conhecimento amplificado através da criação do conhecimento.

III. Decida o mais tarde possível

A prática de adiar a decisão garante que estas não sejam tomadas em um momento de incerteza, adiando-as, até que haja uma maturidade maior e as deliberações possam ser proferidas.

IV. Entregue o mais rápido possível

"A moral da história é que devemos encontrar uma maneira de entregar software tão rápido, que nossos clientes não tenham tempo de mudar de ideia". (Mary Poppendieck, 2003)

O *Feedback* é um item essencial para o *Lean*. No desenvolvimento, há um ciclo de descoberta que é fundamental para a aprendizagem: estórias, implementação, feedback e melhorias. Quanto menor esse ciclo, mais poderá ser aprendido. Após a solicitação do cliente, deve haver um esforço para entregá-lo software em funcionamento o mais rápido possível. A rapidez deste ciclo faz com que o cliente tenha o que necessita o quanto antes. Em uma organização madura em desenvolvimento de software, tudo isso acontece em um fluxo rápido em resposta a uma necessidade dos clientes.

REVISTA CIÊNCIA E SUSTENTABILIDADE
ISSN 2447-4606

V. Dê autonomia à equipe

Um dos doze princípios do manifesto ágil é "Construir projetos ao redor de indivíduos motivados. Dando a eles o ambiente e suporte necessário, e confiar que farão seu trabalho." Uma equipe deve ser suficientemente auto organizada e multidisciplinar, com a capacidade de tomar decisões, assumir compromissos, riscos, ir atrás de seus próprios objetivos.

VI. Construa com integridade

Qualidade é inegociável. Entregue qualidade intrínseca e explícita aos seus clientes, se eles perceberem isso, significa que foi uma entrega de qualidade. De acordo com Poppendieck (2003), existem duas dimensões de integridade: integridade percebida e integridade conceitual. A integridade percebida significa que a totalidade do produto alcança um equilíbrio entre as funcionalidades, usabilidade, confiabilidade, economia e isso encanta o cliente. A integridade conceitual significa que os conceitos centrais do sistema de trabalho em conjunto são facilitados e coesos. Essa última é fator crítico de sucesso para a integridade percebida.

V. Visualize o todo

O desenvolvimento de sistemas focando o todo visa diminuir problemas de integridade. Especialistas de áreas específicas tendem a tratar as partes do sistema das quais estão envolvidos de forma separada. Uma visão mais complexa faz com que o equilíbrio entre todas as particularidades de um sistema seja mantido e haja um melhor aproveitamento.

3 UM *OVERVIEW* DE PRÁTICAS ÁGEIS

3.1 *USER STORIES*

A técnica de *User Stories*, assim como o próprio nome diz, utiliza uma breve descrição do usuário para o que deve ser feito.

Ela é constituída em apenas um parágrafo sucinto a respeito da funcionalidade do sistema e pode ser escrita em cartões (*Stories Cards*), para ser facilmente manuseável. O conteúdo dos cartões deve ser simples, limpo e breve.

REVISTA CIÊNCIA E SUSTENTABILIDADE
ISSN 2447-4606

Uma das vantagens de se utilizar as *User Stories* é que ela pode ser feita por qualquer membro da equipe, não há necessidade de tem um profundo conhecimento em levantamento de requisitos, como o Caso de Uso. Dessa forma, ela pode ser utilizada por equipes que não conhecem o Caso de Uso, entretanto, quando a equipe é mais experiente, também pode ser utilizada em conjunto.

A ideia de alterar o foco da escrita para a fala é fazer com que o usuário final receba o que ele realmente precisa, e não exatamente o que ele quer. A partir de pequenas histórias, o *software* é criado com o propósito de atender os objetivos da melhor forma possível. Ou, na pior das hipóteses, o cliente vai receber o que ele escreveu nas histórias.

Após a obtenção das histórias, elas são agrupadas de acordo com suas prioridades, originando o *Product Backlog*. E ao final do processo, testes de aceitação ajudam a confirmar se a história foi codificada corretamente.

3.1.1 Escrevendo histórias

Como explica Cohn (2004), para escrever as melhores histórias devemos buscar em seis propriedades apresentadas no Quadro 3 a seguir:

Quadro 3 - Propriedades de como escrever histórias usando *User Stories*

Norteadores	Descrição
Independente	Deve-se ter cuidado com a dependência entre as histórias, pois entre histórias dependentes entre si, pode se tornar difícil fazer o planejamento do tempo de cada atividade, assim como sua priorização.
Negociável	As histórias não devem conter muitos detalhes. Elas devem servir apenas como um lembrete para uma negociação entre o desenvolvedor e o cliente.
Valiosa ao Comprador ou Usuário	As *User Stories* não têm necessariamente o dever de serem valiosas sempre aos usuários do sistema, elas podem ser valiosas somente ao comprador do sistema, como, por exemplo, configurações do software para os usuários ou estruturas e metodologias a serem seguidas.
Estimável	"É importante que desenvolveres sejam capazes de estimar (ou pelo menos ter uma ideia) do tamanho de uma história ou o tempo que será levado para transformá-la em código", cita Cohn (2004).
Pequena	O tamanho da história deve ser correspondente com a equipe e as tecnologias que serão utilizadas. Uma história muito grande ou muito pequena, como citado anteriormente, é difícil de ser estimada. Quando uma história é muito abrangente, ela deve ser dividida em outras histórias, para que possa ser desenvolvida.
Testável	Se uma história deixa claro o que deve ser testado, o desenvolvedor não pode saber com certeza o que deve ser feito ou quando a história está toda codificada.

Fonte: Cohn (2004).

REVISTA CIÊNCIA E SUSTENTABILIDADE
ISSN 2447-4606

Finalmente, o desenvolvedor é responsável por auxiliar o cliente/usuário a escrever as histórias e se certificar que elas sigam as seis propriedades. Lembrando que a melhor maneira de escrever uma *User Story* é deixar o próprio cliente escrevê-la, mas isso pode deixá-los desconfortáveis, pensando que eles serão responsabilizados pelo que escreverem, porém se o desenvolvedor acha necessário pedir alguma informação sobre alguma tecnologia, ele deve escrevê-la de forma que o cliente possa entendê-lo.

3.2 *REFACTORING*

A refatoração é um meio de alterar o código e/ou a estrutura de um software sem alterar a sua funcionalidade, fazendo com que o código seja mais limpo e fácil de ser entendido.

Refatoração é arriscada. Quanto mais você começa a examinar o código, descobre que mais ainda pode ser refatorado, e se não for feito sistematicamente, ela pode te envolver e cada vez mais, fazendo com que você perca o foco.

Ela segue o caminho inverso do processo de produção de um software - onde criamos primeiramente a arquitetura e depois codificamos - utilizando o código já existente para desenharmos uma arquitetura melhor compatível.

Antes de começar a refatoração, deve-se construir uma base sólida de testes para o código que vai ser alterado. Para que possamos ter certeza que o resultado final da refatoração está exatamente igual ao anterior.

Ao se realizar uma refatoração, deve ser observada a clareza do código, pois o compilador pode não ligar se o código está feio ou confuso, mas as pessoas que farão a manutenção do código se importarão com sua clareza. O que significa também que o código será mais fácil de ser compreendido, pois do contrário, o desenvolvedor teria dificuldade em achar o pondo a ser alterado, aumentando a chance de ocorrer erros.

Eis alguns pontos importantes que devem ser levados em consideração ao realizar uma refatoração em um sistema. Deve-se levar em consideração 3 (três) pontos importantes que devem ser levados em consideração ao realizar uma

REVISTA CIÊNCIA E SUSTENTABILIDADE
ISSN 2447-4606

refatoração de um sistema. São eles: **(i)** Quebrar em métodos menores: Muitas vezes, o primeiro alvo a ser buscado é a divisão de métodos longos em outros menores. Para dar início à este processo, deve ser buscado toda variável global do método. Quando o valor de uma variável não é alterado, ela pode ser utilizada como parâmetro. Porém, as que são utilizadas necessitam de mais cuidado, se ela for alterada apenas uma vez, pode ser utilizada como retorno; **(ii)** Alterando a nomenclatura: os nomes das variáveis devem indicar claramente a função dela no código; e **(iii)** Reposicionar métodos: os métodos devem ficar na classe onde ele utiliza seus dados. Em alguns casos, esse ato de mover o método pode inutilizar algum parâmetro, pois a informação já está lá.

Além dessas, existem outras inúmeras técnicas para fazer uma boa refatoração, além de ferramentas que auxiliam nesse processo. Como escreve Fowler (1999), "qualquer tolo pode escrever código que um computador possa entender, bons programadores escrevem códigos que humanos podem entender". Mas é importante lembrar que não são todos os casos que podem ser refatorados. Por exemplo, quando o prazo está curto, ou até mesmo quando um código está tão ruim que rescrever é melhor.

3.3 KANBAN

Kanban é uma palavra de origem japonesa que significa cartão visual. Ele é baseado numa ideia bem simples: o total de atividades em andamento (WIP) deve ser limitado e uma atividade só deve ser iniciada quando for possível iniciá-la, ou seja, quando uma atividade em andamento for liberada ou caso o limite imposto esteja disponível. Ele se utiliza de um mecanismo visual para acompanhar o fluxo das atividades.

Conforme mencionou Yoshima (2010, *apud* VALE, 2009), Kanban Systems for Software Development é uma filosofia de gestão que pode se aplicar a qualquer tipo de processo com o intuito de melhorá-lo de maneira incremental. O Kanban não tem regras. Você é livre para mudar o Kanban, mas ele tem propriedades que podem efetivamente melhorar um sistema complexo, gerando fluxo e coordenação, mesmo entre vários times. Essas propriedades são: a) visualize o fluxo de trabalho; b) limite

REVISTA CIÊNCIA E SUSTENTABILIDADE
ISSN 2447-4606

o trabalho em andamento; c) meça e gerencie o fluxo; d) torne as políticas explícitas, e; d) use modelos para melhoria.

Anderson ([201-] *apud* YOSHIMA, 2010) cita ainda que essas simples medidas são poderosas para que uma equipe com um trabalho possa explorar melhor suas potencialidades de suas limitações oferecendo indicadores claros dos problemas para que a empresa mude na direção correta, de maneira incremental, e principalmente, gerando o mínimo de impacto emocional na organização.

Segundo VALE (2009), esta técnica começou a ser utilizada em desenvolvimento de software com a introdução das metodologias ágeis. Um quadro com cartões e post-its é utilizado para sinalizar o status do trabalho em andamento. Quando a equipe move, por exemplo, um cartão para área de concluído do quadro, ela sinaliza que está pronta para o próximo trabalho. Isso permite a criação de fluxo dentro de uma iteração e impede que a demanda seja empurrada para a equipe em grandes lotes. As vantagens do Kanban são: Permite um controle visual das atividades existentes, em andamento ou na fila para iniciar. Permite identificar diversos problemas no processo de desenvolvimento. Promove uma equipe multifuncional. Estimula a melhoria contínua (Kaizen) e a redução dos desperdícios de todos os tipos (*Lean*). Diminui a burocracia na metodologia de desenvolvimento. E por fim, estimula que as atividades sejam feitas no tempo correto (*Just In Time*).

3.4 TEST DRIVEN DEVELOPMENT (TDD)

Test Driven Development (TDD) é uma prática adotada por diversos desenvolvedores há décadas. Atualmente, vem ganhando maior visibilidade como prática dentro do XP (*Extreme Programming*). Também conhecida pelos nomes de *Test First Design* (TFD), *Test First Programming* (TFP) e *Test Driven Design* (TDD) (GEORGE; WILLIAMS, 2003), esta prática consiste de pequenas iterações onde novos casos de teste são escritos contemplando uma nova funcionalidade ou melhoria e, depois, o código necessário e suficiente para passar esse teste é implementado. Então, o software é refatorado para contemplar as mudanças de forma que os testes continuem passando (FEITOSA, 2007).

REVISTA CIÊNCIA E SUSTENTABILIDADE
ISSN 2447-4606

Segundo Beck (2010), TDD tem por objetivo obter código limpo que funciona. Ele diz também que o ciclo básico do TDD é formado pelas seguintes etapas:

a) adicione um pequeno teste;
b) rode todos os testes e falhe;
c) faça uma pequena mudança;
d) rode os testes e seja bem-sucedido;
e) refatore para remover duplicação.

Nesse contexto, os testes atuam como uma rede de segurança, pois defeitos gerados no processo de refatoração são facilmente e rapidamente detectados. Devido a essa segurança na melhoria contínua do código, ele fica mais simples e mais fácil de ser mantido. Segundo um estudo comparativo com a criação dos testes depois da codificação usando TDD, a complexidade do código-fonte acaba sendo menor (CANFORA *et al.*, 2009 *apud* VIEIRA; VIÉGAS, 2010).

Segundo Aniche, Ferreira e Gerosa (2010), TDD é uma prática com foco no feedback. Uma definição mais formal sugerida por eles é que TDD é a arte de produzir testes automatizados para código de produção e utilizar esse processo para direcionar o design e a programação.

As pessoas costumam discutir se TDD é uma técnica de teste, pois os programadores precisam escrever testes unitários o tempo todo; ou uma técnica de Design, pois os testes gerados dão feedback sobre o Design e os programadores utilizam isso para melhorar o Design do software. Alguns especialistas alegam que TDD é de fato uma técnica de Design, e os testes são simplesmente uma consequência (MARTIN, 2002).

No trabalho de Aniche, Ferreira e Gerosa (2010), um participante mencionou o efeito do feedback do teste. Como os desenvolvedores recebem de forma rápida e constante feedback sobre o design do código, eles estão aptos a encontrar problemas e consertá-los enquanto é barato e fácil: TDD faz você escrever o código desde o início. Depois disso você percebe que o que foi feito não é tão bom assim, então você começa a melhorar seu código. É dessa forma que TDD acaba influenciando o design, evoluindo-o constantemente.

REVISTA CIÊNCIA E SUSTENTABILIDADE
ISSN 2447-4606

O desenvolvimento guiado por testes apresenta uma grande quantidade de benefícios entre eles (GOLD *et al*. 2005):

a) Desenvolvimento simples e incremental: um dos principais benefícios desta abordagem é que temos um software funcionando quase que imediatamente. A primeira iteração do software é simples e não tem muita funcionalidade, porém a funcionalidade irá melhorando conforme o desenvolvimento for progredindo. É uma abordagem menos arriscada do que tentar construir um sistema que somente funcionará quando colocarmos todas as partes dele juntas;

b) Desenvolvimento menos estressante: Os desenvolvedores que utilizam TDD precisam se preocupar apenas em ter o próximo teste executado com sucesso. O desenvolvedor foca sua atenção em pequenas partes do software, faz com que ele funcione e parte para outra parte. As decisões de projeto e implementação vão sendo tomadas aos poucos, conforme os testes são criados;

c) Testes constantes de regressão: No desenvolvimento de software muitas vezes passamos pelo efeito dominó, onde uma simples alteração num módulo do sistema pode ocasionar consequências nas demais partes do sistema. Na metodologia TDD a cada alteração realizada no código são executados todos os testes gerados para o sistema até o momento, desta forma qualquer alteração que ocasione um efeito indesejado em outras partes do código e detectado imediatamente. Outra vantagem é que a cada iteração sempre temos o código funcionando;

d) Melhora da comunicação: Muitas vezes é difícil expressar em palavras como deve ser o funcionamento de uma parte do software. Os testes de unidade servem como uma linguagem que pode ser usada para comunicar o comportamento exato de uma parte do software;

e) Melhora da compreensão do comportamento requerido do software: A escrita de testes de unidade antes da escrita do código ajuda a focar e entender o comportamento requerido do software;

REVISTA CIÊNCIA E SUSTENTABILIDADE
ISSN 2447-4606

f) Centralização do conhecimento: Muitas empresas costumam deixar o desenvolvimento dos módulos de um software para uma única pessoa, desta forma todo o conhecimento a respeito da implementação daquele código está na cabeça de um único funcionário. A documentação e a implementação de um código limpo permitem que outros programadores entendam como o código foi implementado, porém nem sempre isto é realizado. Com TDD, os testes de unidade constituem um repositório que provê informações sobre as decisões que foram tomadas para o projeto do módulo.

Test Driven Development tem implicações sociais, dentre elas: com a densidade de defeitos reduzidos, a *Quality Assurance* (um programa de acompanhamento sistemático e avaliação dos diferentes aspectos de um projeto, serviço ou facilidade para garantir que os padrões de qualidade estão sendo cumpridos) do projeto pode ter um trabalho mais pró-ativo; gerentes podem estimar com precisão suficiente para envolver clientes no processo; engenheiros de software podem trabalhar em colaboração minuto-a-minuto; e há software potencialmente entregável com novas funcionalidades todos os dias (BECK, 2002 *apud* BALLE, 2011).

TDD é uma prática que pode ser combinada com outras técnicas e utilizada em qualquer metodologia de desenvolvimento de software.

3.5 *PAIR PROGRAMMING*

Pair Programmming é uma técnica de desenvolvimento no qual dois programadores trabalham juntos em uma mesma estação de trabalho, compartilhando códigos, ideias e soluções. Ela possui dois papeis bem definidos, o piloto e o copiloto. Enquanto o piloto trabalha codificando o sistema, o copiloto trabalha analisando o código gerado, detectando erros, problemas e possíveis melhorias. Durante o desenvolvimento os programadores alternam entre os papeis. Trabalhando dessa forma a atividade desenvolvida tende a ter uma melhoria na sua qualidade.

A formação das duplas é dinâmica. Se duas pessoas pareiam pela manhã, à tarde eles poderiam facilmente trabalhar com outras pessoas. Se um dos membros

REVISTA CIÊNCIA E SUSTENTABILIDADE
ISSN 2447-4606

do time tem a responsabilidade de uma tarefa em uma área que lhe é estranha, pode pedir a alguém com experiência nessa área para parear. É uma forma de passar conhecimento para todos no time (BECK, 1999 *apud* BALLE, 2011).

Programação pareada valoriza principalmente a comunicação, especialmente ao fazer um rodízio entre os membros de todos os pares, alternando pessoas entre os pares com certa frequência, para que o conhecimento sobre as diferentes partes do sistema seja compartilhado entre todos os membros da equipe e a diversidade do time seja potencializada. Esta prática também valoriza a coragem, pois duas pessoas têm mais coragem do que uma e valoriza o feedback através do diálogo ativo entre o par (SILVA, 2007).

Faria e Yamanaka (2010), falaram que *Pair Programming* não é uma prática nova. Coplien e Schmidt (1995) publicou seu livro sobre processo de produção de software sugerindo um padrão organizacional de desenvolvimento emparelhado. Em seu trabalho Faria e Yamanaka (2010) citam alguns benefícios do uso de Pair Programming. O Quadro 4 apresenta duas listas. Uma com os benefícios unânimes de Pair Programming e outra com os que são questionáveis.

Quadro 4 - Benefícios unânimes e questionáveis do Pair Programming *User Stories*

Benefícios unânimes	Benefícios questionáveis
Melhora a qualidade global do software.	Reduz o tempo de desenvolvimento.
Promove a transferência de informação e conhecimento entre os membros de equipes.	Reduz o custo de desenvolvimento.
Aumenta o moral e a confiança dos programadores na própria solução.	Produz código mais eficiente.
Aumenta a satisfação dos desenvolvedores durante as atividades de programação.	Reduz a taxa de erros encontrados nos testes de integração devido às revisões e testes contínuos durante o desenvolvimento.
Reduz a taxa de erros encontrados nos testes de unidade devido às revisões e testes contínuos durante o desenvolvimento.	
Aumenta a responsabilidade dos programadores devido à pressão dos pares.	
Aprimora as habilidades colaborativas dos programadores.	
Aprimora as habilidades de comunicação dos programadores.	
Provê aos programadores, ao observarem seus sócios, o conhecimento de diferentes abordagens para a solução de problemas.	

REVISTA CIÊNCIA E SUSTENTABILIDADE
ISSN 2447-4606

Resulta em código menor.	

Fonte: Faria e Yamanaka (2010).

Porém há desvantagens desta técnica, que devem ser lavadas em consideração na adequação no momento de aplicar em algum projeto real. Algumas desvantagens desta técnica. São elas: Problemas culturais: A aplicação dessa técnica, deve ser gradual, pois se for aplicada de uma forma brusca, a equipe pode não se adaptar e a produtividade diminuir ao invés de aumentar. E o Cálculo da produtividade: Métricas de produtividade são bastante obscuras quando se trata de programação em dupla. Os critérios de avaliação podem ser muito subjetivos, indo desde a afinidade entre a dupla, até mesmo o processo de ensinar um método criado a outra pessoa. Mensurar o rendimento da equipe nessa situação é possível, mas na mesma linha é também trabalhoso.

3.6 *DAILY MEETING*

Daily Meeting é uma reunião de curta duração realizada diariamente pela equipe, ela é sempre feita no mesmo horário e no mesmo local durante as iterações e tem duração máxima de 15 minutos (SCHWABER, SUTHERLAND, 2009 *apud* BALLE, 2011), na qual cada membro fala sobre o seu progresso e reporta quaisquer problema que ocorreu para que o responsável pela equipe possa mitigar ou até eliminar o problema.

É comum nessa reunião serem feitas três perguntas para cada membro:

1. O que você fez desde a última reunião?
2. Quais foram as suas dificuldades durante o trabalho?
3. Quais as atividades você pretende fazer até a próxima reunião?

Segundo Balle (2011) o sentido da Daily *Scrum* não é ser uma reunião de status, é uma inspeção do progresso na direção da meta do Sprint. Geralmente acontecem reuniões subsequentes para realizar adaptações ao trabalho que está por vir na Sprint. A intenção é otimizar a probabilidade de que o time alcance sua meta. Essa é uma reunião fundamental de inspeção e adaptação no processo empírico do *Scrum*.

REVISTA CIÊNCIA E SUSTENTABILIDADE
ISSN 2447-4606

Concentrando-se no que cada pessoa fez ontem e no que ela irá fazer hoje, a equipe ganha uma excelente compreensão sobre que trabalho foi feito e que trabalho ainda precisa ser feito. O *Daily Scrum* não é uma reunião de status report na qual um chefe fica coletando informações sobre quem está atrasado. Ao invés disso, é uma reunião na qual membros da equipe assumem compromissos perante os demais.

Quando é relatado um problema na reunião diária de interesse de outros membros da equipe ou quando há a necessidade de assistência de outros membros da equipe, qualquer membro de equipe pode se organizar imediatamente após a *Daily Meeting* (SCHWABER, 2004).

Ludvig e Reinert (2007) citam que esta reunião apesar de ser curta, é extremamente útil e Schwaber destaca alguns benefícios: **(i)** problemas identificados são apresentados a gerência para uma possível solução assim que ocorrerem mantendo o desenvolvimento em dia; **(ii)** evitar duplicação de trabalho; **(iii)** melhor compreensão sobre as interdependências entre os membros da equipe; **(iv)** trabalho em equipe; **(v)** produção dinâmica entre a equipe; e **(vi)** produção mais eficaz com a pressão de estar à frente da equipe e dizer o que você tem feito.

Alguns objetivos da reunião diária são: Ajudar a começar o dia bem; apoiar melhorias; manter o foco nas coisas certas; manter o senso de equipe; e comunicar o que está acontecendo.

3.7 INTEGRAÇÃO CONTÍNUA

Segundo Teles (2005) equipes de desenvolvimento normalmente são compostas por diversos programadores, trabalhando no mesmo projeto. Isso cria dois problemas práticos. O primeiro é que sempre que diversos indivíduos estão trabalhando na mesma coisa, ocorre uma necessidade de sincronização (POPPENDIECK; POPPENDIECK, 2003). O segundo é que todos precisam ser capazes de evoluir rapidamente sem interferir no trabalho uns dos outros. Portanto, enquanto desenvolve, você quer fingir que é o único programador no projeto. Você quer marchar adiante em velocidade máxima ignorando a relação entre as mudanças que você efetua e as mudanças que os demais estão fazendo (BECK, 2000).

REVISTA CIÊNCIA E SUSTENTABILIDADE
ISSN 2447-4606

Esses problemas podem ser resolvidos com a utilização da técnica de Integração Contínua. "Integração Contínua é uma prática de desenvolvimento de software onde os membros de um time integram seu trabalho frequentemente, geralmente cada pessoa integra pelo menos diariamente – podendo haver múltiplas integrações por dia. Cada integração é verificada por um build automatizado (incluindo testes) para detectar erros de integração o mais rápido possível. Muitos times acham que essa abordagem leva a uma significante redução nos problemas de integração e permite que um time desenvolva software coeso mais rapidamente (FOWLER, 2006).

Para Balle (2011) a definição da frequência, na qual ocorre a integração contínua, dependerá de cada time, sendo que diversos fatores devem ser levados em consideração, como tamanho da equipe, grau de distribuição, entre outros. Com Integração Contínua segue-se o princípio do Agile que quanto mais cedo forem detectados os problemas, mais simples solucioná-los. Devem ser verificadas quebras de build – versão "compilada" de um software ou parte dele que contém um conjunto de recursos que poderão integrar o produto final – e, de preferência, deve ser feito uso de um build automatizado para a construção.

A prática da integração contínua é a atividade de unir o trabalho feito por um par de programadores, ao código como um todo. Logo após terminar determinada atividade, o par deve testar e juntar seu código à versão mais recente do código coletivo. Isso deve ser feito várias vezes ao dia, para sincronizar as atividades individuais (BECK, 2004).

Duvall *et al.* (2007), descreve o cenário de Integração Contínua da seguinte maneira: O desenvolvedor envia o código para o controle de versão. Enquanto isso a máquina de integração (Servidor de Integração Contínua) está "ouvindo" o repositório buscando por modificações. Logo após um commit ser efetuado, o servidor identificará a mudança no código. Inicia-se o processo de build baixando os arquivos do Servidor de Controle de Versão. Assim, o script de build é executado, testes são realizados, relatórios gerados e todo o projeto é integrado. O servidor de integração envia por e-mail ou outros dispositivos o feedback sobre o build para usuários específicos do projeto. E o servidor volta ao estado de Pull buscando por mudanças no repositório.

REVISTA CIÊNCIA E SUSTENTABILIDADE
ISSN 2447-4606

No processo de Integração Contínua existem cinco fases principais. Que são:

I. Compilação: é o elemento mais básico de todo o processo de construção e integração. A geração de código binário é requisito para o sucesso das outras fases. Dependendo da linguagem o processo de criação de um binário pode ser substituído por uma atenção mais minuciosa a fase de teste e inspeção;
II. Teste: é imprescindível o uso de testes automatizados (testes unitários, testes de carga/performance, teste de segurança, teste de integração entre outros) para efetuar a validação de requisitos de negócio, funcionalidades e correções de bugs e com isso garantir a confiabilidade do software;
III. Inspeção: automatizar a inspeção de código, análise dinâmica e estática, pode ser utilizado para auxiliar na melhoria da qualidade de software através do estabelecimento de regras;
IV. Implantação: esta é a forma de garantir que a qualquer momento pode entrar em produção uma versão testada, compilada (dependendo da linguagem) e pronta para uso;
V. Feedback Contínuo e Documentação: automatizar a criação das documentações diminui a dificuldade dos desenvolvedores em criar ou atualizar as documentações.

Algumas vantagens da integração contínua são descritas no Quadro 5 a seguir:

Quadro 5 - Vantagens na integração contínua

Vantagens	Descrição
Redução de Riscos	Integrando o código várias vezes ao dia, é possível reduzir alguns dos riscos do projeto, facilitando detecção de defeitos, mensuração da saúde do código e redução do número de incertezas.
Redução de Processos Repetitivos	Processos repetitivos, que são executados por seres humanos, no ambiente de desenvolvimento, devem ser automatizados. Com isso, o custo, esforço e tempo necessários para execução dessas atividades podem ser repassadas para o processo de criação e produção de software.
Gerar software funcionando	A integração contínua permite que seja entregue software funcionando a qualquer momento. Para os clientes, essa é a maior vantagem da Integração Contínua.
Ampliar a Visibilidade do Projeto	O fato de ter um feedback rápido e constante, fará com que os desenvolvedores consigam tomar decisões mais acertadas e concisas

REVISTA CIÊNCIA E SUSTENTABILIDADE
ISSN 2447-4606

Ampliar a Confiança no Produto	Em um ambiente com integração contínua, os desenvolvedores se sentem confiantes para entregar sistemas que funcionam e com qualidade.

Fonte: elaborado pelos autores (2018).

3.8 *PLANNING POKER*

Planning Poker é uma técnica ágil que busca estimar o esforço necessário para conclusão ou o tamanho relativo das atividades. O método foi descrito inicialmente por James Greening em 2002 e popularizado por Mike Cohn no seu *livro Agile Estimating and Planning*.

Para iniciar uma sessão de estimativa, o *Product Owner* ou o cliente lê uma *User Story* ou descreve um recurso para os estimadores, que deve incluir todos na equipe. Cada estimador está segurando um baralho de cartas com valores. Os mais comumente utilizados são os primeiros números da sequência de Fibonacci (0, 1, 2, 3, 5, 8, 13, 20, 40 e 100), porém, podem ser utilizados outros valores que expressem a mesma ideia. Os valores representam o número de Story Points, dia ideal, ou outra unidade de estimativa que faça sentido para o time (SCHWABER; SUTHERLAND 2010 *apud* BALLE, 2011).

Os estimadores discutem a função, fazendo perguntas para o *Product Owner*, conforme necessário. Quando o recurso tiver sido amplamente debatido, cada estimador seleciona uma carta para representar a sua estimativa, sem mostrá-la aos demais. Todos os cartões são, então, revelados ao mesmo tempo. Se todos os estimadores houverem selecionado o mesmo valor, ele se torna a estimativa. Se não, os estimadores discutem as suas estimativas. As estimativas mais altas e baixa deve principalmente partilhar as suas razões. Após um debate, cada estimador seleciona novamente um cartão de estimativa e todas as cartas são novamente reveladas ao mesmo tempo (SCHWABER; SUTHERLAND, 2010 apud BALLE, 2011).

O processo é repetido até que o consenso é alcançado ou até que os estimadores decidirem que a estimativa de um determinado item deve ser adiada até que informações adicionais possam ser adquiridas (BALLE, 2011).

Cohn (2005), nos mostra em Agile Estimating and Planning, porque Planning Poker funciona:

REVISTA CIÊNCIA E SUSTENTABILIDADE
ISSN 2447-4606

Primeiro, ele reúne opiniões de múltiplos especialistas para fazer a estimativa. Porque estes especialistas formam uma equipe multifuncional de todas as disciplinas em um projeto de software, eles são mais adequados para a tarefa de estimativa do que ninguém.

Em segundo lugar, um diálogo vivo segue durante o planejamento de poker, e estimadores são chamados pelos seus pares para justificar as suas estimativas. Sendo pedido para justificar estimativas também tem sido mostrado para resultar em estimativas que melhor compensar as informações em falta (BRENNER *et al.,* 1996). Isso é importante em um projeto ágil porque as histórias de usuário a ser estimados são muitas vezes intencionalmente vagos.

Em terceiro lugar, os estudos mostraram que uma média de estimativas individuais conduz a melhores resultados como fazer discussões de grupo de estimativas. Grupo de discussão é a base do planejamento de poker, e essas discussões levar a uma média de tipos de as estimativas individuais.

4 PRÁTICAS ÁGEIS: PRÁTICA NO PROJETO

Nesse capítulo, será mostrada uma forma de utilização das técnicas ágeis discutidas nos capítulos anteriores. Usaremos um exemplo bem simples de uma locadora de filmes para que fique o mais claro possível.

Começaremos a criação do software obtendo informações do cliente do que especificamente ele vai precisar no sistema. Utilizando as técnicas de *User Stories*, vamos buscar conhecer o sistema através de uma conversa com o cliente, fazendo com que ele mesmo escreva cada funcionalidade que ele espera do sistema. Lembrando sempre de fazer com que as histórias sigam as regras descritas na técnica: Independente, Negociável, Valiosa ao Comprador ou Usuário, Estimável, Pequena e Testável.

Suponhamos que o cliente tenha descrito duas histórias da seguinte maneira:

O consumidor deve poder alugar DVD.

e

O consumidor deve poder alugar Blu-rays.

REVISTA CIÊNCIA E SUSTENTABILIDADE
ISSN 2447-4606

As duas histórias são de suma importância, porém uma história é muito dependente da outra, ferindo uma das regras descritas anteriormente. Orientamos o nosso cliente sobre este caso, e ele combina as duas histórias em uma só:

> O consumidor deve poder alugar DVDs e Blu-rays. Mas quando o consumidor alugar dois itens deve receber um desconto de 5%. Se o consumidor alugar quatro itens, ele recebe 10% de desconto. E se levar seis ou mais itens, ele recebe 15%.

Dessa vez ferindo outras duas regras da *User Stories*, a que diz que ele deve ser Pequena, Estimável e Negociável. As duas primeiras andam juntas, quanto maior mais difícil de estimar um tempo para ela história. A terceira deve-se ao fato de que quando uma história contém muitos detalhes, ou existem detalhes que não precisariam estar aí (a não ser que fosse apenas uma pequena observação), ou ela deveria ser separada em diferentes histórias. No nosso caso, ficaria melhor se corrigíssemos da seguinte maneira:

> "O consumidor deve poder alugar DVDs e Blu-rays.
> Obs.: Para cada par de itens ele recebe 5% de desconto".

Após a obtenção das *User Stories* esteja completa, começamos a criar cartões visuais a partir das *User Stories*, com os campos Desenvolvedor e *Tester*, seguidos por espaços para datas e horas de início e término (como mostra a Figura 1) e, após isso, organizamos cada cartão conforme suas prioridades em um quadro com linhas que separam as áreas da nossa fábrica de software (Figura 2).

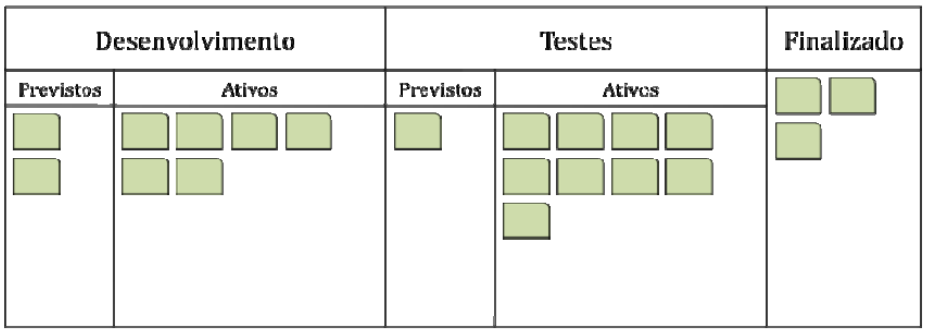
REVISTA CIÊNCIA E SUSTENTABILIDADE
ISSN 2447-4606

Figura 1 - Ficha para aluguel de filmes

O funcionário poderá gerar relatório com os filmes alugados por um determinado cliente		Tempo Estimado: 8h
Responsável		
Data/Hora Início		
Data/Hora Fim		
Duração		

Fonte: elaborado pelos autores (2018).

Figura 2 - Áreas da fábrica



Fonte: elaborado pelos autores (2018).

Para fazermos a estimativa do tempo necessário para cada *User Story*, reunimos todos os integrantes da equipe de desenvolvimento em uma sala, onde todos têm em mãos cartas com os dez primeiros números do Fibonacci. E, a cada história que é lida para os presentes na sala, eles escolhem uma carta com o valor da sua estimativa sobre a história. Quando todos já escolheram, as cartas são mostradas de uma única vez e aí começa uma conversa, até se chegar em uma conclusão sobre a melhor estimativa possível para o caso.

Feita a estimativa, como primeiro passo para começarmos a codificação do sistema, pegamos o cartão visual que contém a história a ser codificada e a colocamos na área do quadro destinava ao Desenvolvimento, colocando o nome do

REVISTA CIÊNCIA E SUSTENTABILIDADE
ISSN 2447-4606

colaborador no campo Desenvolvedor, seguido pela data e hora em que está sendo iniciado.

Mas antes de começarmos a codificação propriamente dita, utilizamos a técnica descrita no TDD e escrevemos os testes pontuais para a história que será codificada. No nosso sistema de locadora, vamos utilizar o *framework* JUnit para escrevermos os testes, porém, ele não é único. Muitas linguagens têm o *framework* seu correspondente para que possa escrever e executar testes.

Utilizando como exemplo a história descrita na Figura 1, vamos continuar o processo aplicando técnicas do TDD. Para esta história devemos começar a desenvolver da seguinte maneira:

```
Cliente maria = ListaCliente.getCliente("Maria");
maria.getFilmesAlugados();
```

A princípio, o sistema vai acusar alguns erros, como por exemplo, não existe a classe Cliente. Mas isso é previsto, pois ela não foi criada ainda. Então criamos as classes *Cliente* e *ListaCliente*:


```
class Cliente
{
string Nome;
string Endereco;
}
class ListaCliente { }
```

Após isso o sistema vai acusar outro erro, não existem os métodos *getCliente()* e *getFilmesAlugados()*. Codificaremos então, os dois métodos e, assim, sucessivamente. Dessa maneira, não será criado código desnecessariamente, ou seja, tudo que for escrito, terá sua utilidade no sistema.

Com o desenvolvimento sendo feito, na maioria das vezes, de forma focada em um único teste, geralmente é necessário utilizar algumas técnicas de *Refactoring* nos intervalos de codificação das *User Stories*.

REVISTA CIÊNCIA E SUSTENTABILIDADE
ISSN 2447-4606

Assim que o desenvolvedor termina sua participação na história atual, ele preenche o campo do cartão que contém a história em que ele está trabalhando com a data e hora do término de sua tarefa e, então, passa o cartão retira o cartão da área destinada ao Desenvolvimento e reposiciona na área de Teste. E dessa maneira cada pessoa da equipe de teste deve fazer quando assumir ou terminar uma tarefa de cada cartão posicionado em sua área no quadro.

Em alguns casos, a história escrita no cartão detém um grau de complexidade alto. Quando isso ocorre, utilizamos o *Pair Programming*, aumentando a qualidade do código e, consequentemente, abaixarmos a possibilidade de erros. Essa técnica também é utilizada em alguns casos em que um dos desenvolvedores envolvidos não possui a experiência necessária para assumir toda a responsabilidade da funcionalidade, fazendo com que eles troquem informações e conhecimentos.

Todos os dias, logo pela manhã, a equipe de desenvolvimento se reúne durante 15 minutos para discutir o que cada um está fazendo e o que irá fazer durante o dia. Dessa maneira, todos da equipe ficam cientes do que está acontecendo durante o projeto e, mais importante ainda, discutem, quando necessário, qual é a melhor maneira de resolver um problema, ou uma situação, em que uma das pessoas esteja enfrentando dificuldade.

Esta reunião, ou *Daily Meeting* pode ser transformar em um rápido *Brainstorm*, porém, se ele for se estender muito, a reunião deve ser continuada e finalizada, para que o *Brainstorm* seja continuado posteriormente, não necessariamente com todos os membros da equipe. Mas é importante lembrar-se de informar a equipe, na reunião do dia seguinte, qual foi a decisão tomada para solucionar o problema anterior, caso futuramente outra pessoa passe por uma situação parecida.

Para auxiliar no processo de integração entre os vários desenvolvedores codificando ao mesmo tempo, utilizamos uma ferramenta de Integração Contínua chamada *Hudson*. Porém ela não é a única que encontramos no mercado, podemos obter um resultado parecido com o *Cruise Control* e o *Continuum*.

Dentre as vantagens dessa ferramenta, ela possui um repositório de código e se encarrega de atualizar o repositório a cada *commit* que o desenvolvedor faz. Ela também faz construções de projetos, monitoramento e execução de *jobs*, assim

REVISTA CIÊNCIA E SUSTENTABILIDADE
ISSN 2447-4606

como criação de builds automatizadas e envio de e-mails para as pessoas que fazem parte da *build* quando ela quebra, assim como várias outras.

CONSIDERAÇÕES FINAIS

Desenvolvimento de software é uma atividade extremamente complexa. Ele traz consigo uma quantidade muito grande de variáveis. A Engenharia de Software foi criada para tentar trazer ordem à essa atividade. Ela, inicialmente, veio com a burocracia das engenharias e trouxe uma diversidade muito grande de documentos e artefatos.

Com o tempo, a quantidade excessiva de burocracia dentro das metodologias existentes na Engenharia de Software, foram se tornando um fardo em certos projetos e, viu-se a necessidade da criação de Técnicas Ágeis para que o tempo seja otimizado e melhor aplicado, trazendo também novas formas de gerenciar a equipe e desenvolver o software.

Essas novas técnicas trouxeram uma grande evolução na troca de experiência, comunicação, transmissão de conhecimento, confiança das pessoas, confiança do cliente. E isso faz com que a produtividade da equipe cresça e também faz com que a satisfação do cliente seja maior. As Técnicas Ágeis têm por princípios básicos a simplicidade e a fácil adaptação às mudanças. Fazendo com que sejam muito bem aceitas pelas equipes de desenvolvimento de sistemas e ganhando cada vez mais espaço entre elas no cenário atual.

Uma das características das técnicas ágeis, é que elas costumam funcionar melhor em equipes pequenas. Quando existem equipes muito grandes, geralmente elas são divididas em menores para que possam ser trabalhadas da melhor forma. Essas técnicas não precisam ser utilizadas simultaneamente para um bom resultado, mas o ideal é conhecer o projeto e, principalmente, as pessoas que estão envolvidas nele e, dessa maneira, aplicar as melhores técnicas para cada situação.

Embora exista a crescente utilização de metodologias ágeis, ainda existem alguns pontos fracos com relação a elas. Um exemplo disso é a falta de uma documentação consistente sobre o projeto, ou ainda, a falta de uma análise de risco. Porém, a adaptabilidade das técnicas ágeis com relação a cada situação em que ela

REVISTA CIÊNCIA E SUSTENTABILIDADE
ISSN 2447-4606

será usada, pode fazer com que essas falhas possam ser contornadas, adicionando novas práticas a elas, ou não as aplicando em sua totalidade.

Contudo, assim como acontecia antes da criação dessas técnicas ágeis, não existe uma fórmula mágica para o desenvolvimento de sistemas. O sucesso da aplicação de novas técnicas e metodologias vai depender do conhecimento da técnica pretendida, da percepção da equipe caso algo não esteja saindo como o planejado e da atuação contínua dessa equipe com relação à possíveis melhorias no processo. Afinal, em nenhum caso existe uma verdade absoluta que atenda todos os projetos.

REFERÊNCIAS

ABRAHAMSSON, P. *et al*. New directions on agile methods: a comparative analysis. *In:* INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 25., 2003, Portland. **Anais eletrônicos** [...]. Portland: ICSE, p. 244-254, 2003.

ANICHE, M. F.; FERREIRA, T. F.; GEROSA, M. A. **what concerns beginner test- driven development practitioners:** a qualitative analysis of opinions in an agile conference. 2011. Disponível em: http://www.ime.usp.br/~aniche/files/wbma2011.pdf. Acesso em: 20 fev. 2018.

BALLE, A. R. **Análise de metodologias ágeis**: conceitos, aplicações e relatos sobre XP e *Scrum*. 2011. Disponível em: http://www.lume.ufrgs.br/handle/10183/31028. Acesso em: 25 fev. 2018.

BECK, K. **Extreme programming explained**: embrace change. Boston: Addison-Wesley, 2000.

BECK, K *et al*. **Manifesto para desenvolvimento ágil de software**. 2001. Disponível em: http://agilemanifesto.org/iso/ptbr/manifesto.html. Acesso em: 10 set. 2018.

BECK, K. **Programação extrema (XP) explicada**. [Porto Alegre]: Bookman. 2004.

BECK, K. **TDD:** desenvolvimento guiado por testes. [Porto Alegre]: Bookman, 2010.

CARVALHO, C. E. C.; ABRANTES, C. T.; CAMEIRA, R. F. **Métodos ágeis de desenvolvimento de software**: um caso prático de aplicação do *Scrum*. 2011. Disponível em: http://biblioteca.gpi.ufrj.br/jspui/bitstream/1/305/1/M%C3%89TODOS%20%C3%81GEIS%20DE%20DESENVOLVIMENTO%20DE%20SOFTWARE.pdf. Acesso em: 10 fev. 2018.

REVISTA CIÊNCIA E SUSTENTABILIDADE
ISSN 2447-4606

COHN, M. **Agile estimating and planning**. [New Jersey]: Prentice Hall, 2005.

COHN, M. **User stories applied for agile software development**. [Boston]: Addison-Wesley Professional, 2004.

COPLIEN, J. O.; SCHMIDT, D.; **Pattern languages of program design**. [Boston]: Addison-Wesley Professional,1995.

DUVALL, P.; MATYAS, S.; GLOVER, A. **Continuous integration**: improving software quality and reducing risk. USA: Pearson. 2007.

FADEL, A. C.; SILVEIRA, H. M. **Metodologias ágeis no contexto de desenvolvimento de software**: XP, *Scrum* e Lean. São Paulo: Universidade Estadual de Campinas, 2010.

FARIA, E. S. J.; YAMANAKA, K. **Programação em dupla**: estado de arte. 2010. Disponível em: http://revistas.unicentro.br/index.php/RECEN/article/view/616/1112. Acesso em: 10 fev. 2018.

FEITOSA, D. S. **Um estudo sobre o impacto do uso de desenvolvimento orientado por testes na melhoria da qualidade de software**. 2007. Disponível em: http://disciplinas.dcc.ufba.br/pub/MATA67/TrabalhosSemestre20072/monografia_Daniela Soares_Feitosa.pdf. Acesso em:22 fev. 2018

FERREIRA, R. B.; LIMA, F. P. A. **Metodologias ágeis**: um novo paradigma de desenvolvimento de software. 2006. Disponível em: http://www.cos.ufrj.br/~handrade/woses/woses2006/pdfs/10-Artigo10WOSES-2006.pdf. Acesso em: 20 jan. 2018.

FOWLER, M. **Continuous integration**. 2006. Disponível em: http://martinfowler.com/articles/continuousIntegration.html. Acesso em: 01 mar. 2018.

FOWLER, M. **UML essencial**: um breve guia para a linguagem-padrão de modelagem de objetos. EUA: Bookman, 2005.

FOWLER, M. **Refactoring**: improving the design of existing code. [Boston]: Addison-Wesley Professional,1999.

GEORGE, B.; WILLIAMS, L. **An initial investigation of test-driven development in industry**. 2003. Disponível em: http://collaboration.csc.ncsu.edu/laurie/Papers/TDDpaperv8.pdf. Acesso em: 22 fev. 2018.

GHEZZI, C.; JAZAYERI, M.; MANDRIOLI, D. **Fundamentals of Software Engineering**. [New Jersey]: Prentice Hall, 1991.

REVISTA CIÊNCIA E SUSTENTABILIDADE
ISSN 2447-4606

GOLD, R.; HAMMEL, T.; SNYDER, T. **Test-driven development**: a J2EE example. [s.l.]: Apress, 2005.

HAMED, A. M. M.; ABUSHAMA, H. Popular agile approaches in software development: review and analysis. *In:* INTERNATIONAL CONFERENCE ON COMPUTING, ELECTRICAL AND ELECTRONIC ENGINEERING (ICCEEE), 2013. **Anais eletrônicos** [...]. [s.l.]: IEEE, p. 160-166, 2013.

HIGHSMITH, J.; Reytiing lifecycle dinosaurs: using adaptive software development to meet the challenges of a high-speed, high-change environment. **Software Testing & Quality Engineering**, july-august. 2000.

HILMAN. **Metodologias ágeis**. 2004. Disponível em: http://www.redes.unb.br/material/ESOO/Metodologias%20%c1geis.pdf. Acesso em: 20 abr. 2018.

KNIBERG, H. **Scrum e XP direto das Trincheiras**: como nós fazemos Scrum. 2007 Disponível em: http://www.infoq.com/resource/minibooks/scrum-xp-from-thetrenches/pt/pdf/ScrumeXPDiretodasTrincheiras.pdf. Acesso em: 10 jul. 2018

LEITÃO, M. V. **Aplicação de Scrum em ambiente de desenvolvimento de software educativo**. 2010. Disponível em: http://tcc.dsc.upe.br/20101/TCC_final_Michele.pdf. Acesso em: 12 fev. 2018.

LUDVIG, D.; REINERT, J. D. **Estudo do uso de metodologias ágeis no desenvolvimento de uma aplicação de governo eletrônico**. 2007. Disponível em: http://projetos.inf.ufsc.br/arquivos_projetos/projeto_589/Artigo_Diogo_Jonatas.pdf . Acesso em: 05 fev. 2018.

MARTIN, R. C. **Agile software development, principles, patterns, and practices**. 5. ed. [New Jersey]: Prentice Hall, 2002.

NAUR. P.; RANDELL, B. **Software Engineering**: a report on a Conference Sponsored by de NATO Science Committee. Garmisch, Germany: NATO, 1969. Disponível em: http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF. Acesso em: 22 de fev. 2018.

ORTIGOSA, A. **Proposta de um ambiente adaptável de apoio ao processo de desenvolvimento de software**. Dissertação (Mestrado em Computação). Universidade Federal do Rio Grande do Sul (URFGS), 1995.

PÁDUA, W. **Engenharia de Software**: fundamentos, métodos e padrões. Rio de Janeiro: LTC, 2001.

REVISTA CIÊNCIA E SUSTENTABILIDADE
ISSN 2447-4606

PANTALIÃO, A. **O que é Lean?** 2009. Disponível em: http://www.infoq.com/br/news/2009/08/lean-30-segundos. Acesso em: 18 mar. 2018.

POPPENDIECK, M; POPPENDIECK, T. **Lean software development**: an agile toolkit. Upper Saddle River, NJ: Addison-Wesley, 2003.

PRESSMAN, R. S. **Engenharia de Software**. 6. ed. [s.l.]: Ed.Mc Graw Hill, 2007.

ROYCE, W.W. **Managing the development of large software systems**: concepts and techniques. Los Angeles, CA: Proc. IEEE Westcon, 1970.

SCHWABER, K. **Agile Project Management with Scrum**. [s.l.]: Microsoft Press, 2004.

SCHWABER, K.; BEEDLE, M. **Agile software development with Scrum**. [New Jersey]: Prentice- Hall, Inc., 2002.

SCHWABER, K.; SUTHERLAND, J. **Guia do Scrum**, 2011. Disponível em: http://www.scrum.org/storage/scrumguides/Scrum%20Guide%20%20Portuguese%20BR. pdf. Acesso em: 18 mar. 2018.

SILVA, A. F. **Reflexões sobre o ensino de metodologias ágeis na academia, na indústria e no governo**. 2007. Disponível em: http://www.teses.usp.br/teses/disponiveis/45/45134/tde-17122007-175223/pt-br.php. Acesso em: 15 jan. 2018.

SLIGER, M.; BRODERICK, S.; **The software project manager´s bridge to agility.** [Boston]: Addison-Wesley. 2008

SOMMERVILLE, I. **Engenharia de Software**. 6. ed. São Paulo: Addison Wesley, 2003.

SUTHERLAND, J. *et al.* **Scrum**: an extension pattern language for hyper productive software development. [s.l.]: [s.e.], 2000.

TAKEUCHI, H; NONAKA, I.; The new product development game. **Harvard Business Review**, Cambridge, n. 1, jan./feb., 1986.

TELES, V. M. **Extreme Programming**: aprenda como encantar seus usuários desenvolvendo software com agilidade e alta qualidade. Rio de Janeiro: Novatec Editora, 2004.

TELES, V. M. **Um estudo de caso da adoção das práticas ágeis e valores do Extreme Programming**. 2005. Disponível em: http://improveit.com.br/xp/dissertacaoXP.pdf. Acesso em: 01 mar. 2018.

VALE, A.; **Software Zen**. 2009. Disponível em: https://softwarezen.me/alisson-vale/. Acesso em: 22 set 2018.

REVISTA CIÊNCIA E SUSTENTABILIDADE
ISSN 2447-4606

VASCONCELOS, A. M. L., **Produção de Software**: com ênfase em Software Livre: processos de desenvolvimento de software 1. Lavras: UFLA/FAEPE, 2005.

VIEIRA, D. **Scrum:** a metodologia ágil explicativa de forma definitiva. 2014. Disponível em: http://scrum.mindmaster.com.br/. Acesso em: 25 mar. 2018.

VIEIRA, H. O.; VIÉGAS, J. R. **Ferramenta para inclusão do código de testes na documentação das classes**. 2010. Disponível em: http://junitindoc.googlecode.com/files/TCC_Ferramenta%20para%20Inclus%C3%A3o%20do%20CC3%B3digo%20de%20Testes%20na%20Documenta%C3%A7%C3%A3o%20das%20Classes.pdf. Acesso em: 23 fev. 2018.

WILLIAMS, L.; COCKBURN, A. **Agile software development**: it's about feedback and change. [s.l.]: IEEE Computer, 2003.

YOSHIMA, R. **Desmistificando o Método Kanban**. 2010. Disponível em: http://info.abril.com.br/noticias/rede/gestao20/gestao/desmistificando-o-metodokanban. Acesso em: 06 ago. 2018.