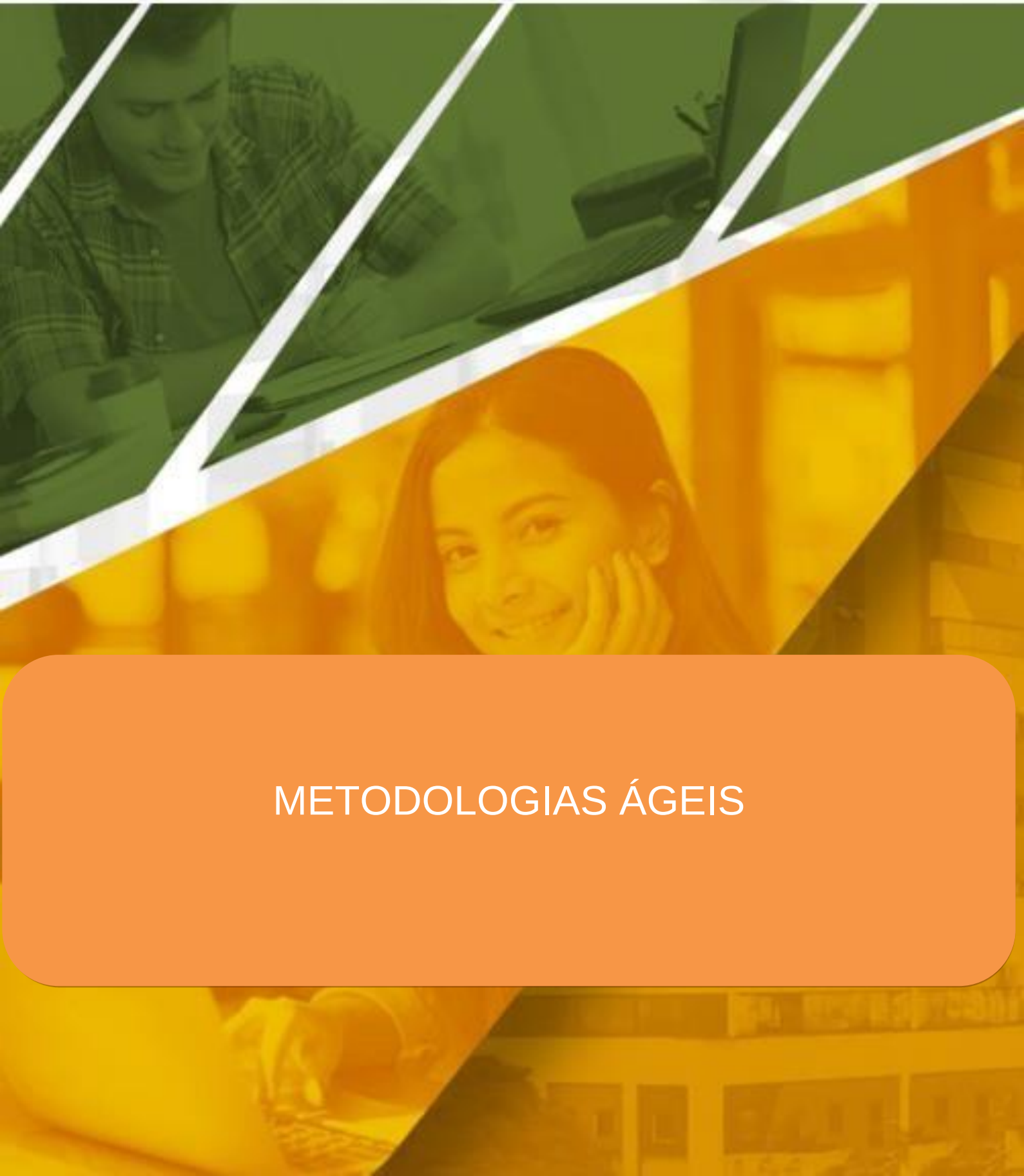
UniFCV
Centro Universitário Cidade Verde



METODOLOGIAS ÁGEIS

UNIDADE I

INTRODUÇÃO AOS MÉTODOS ÁGEIS

Professora Especialista Camila Sanches Navarro

Plano de Estudo:

- Princípios ágeis: O Manifesto Ágil e o que é ser ágil hoje;
- Software ágil: valores e princípios fundamentais;
- Desafios e evolução da Engenharia de Software.

Objetivos de Aprendizagem:

- Compreender o processo de desenvolvimento de um *software*;
- Entender como funcionam as metodologias ágeis, identificando os benefícios, os princípios e os valores do desenvolvimento ágil;
- Obter entendimento da evolução da engenharia de software, compreendendo também todos os processos de software;
- Compreender como surgiu o manifesto ágil e qual a sua finalidade

INTRODUÇÃO

Por muito tempo caro (a) aluno (a), o desenvolvimento de software tradicional foi aplicado em diversos projetos. A metodologia tradicional consiste em definir inicialmente todas as etapas a serem percorridas, como o planejamento do projeto, uma estimativa de prazo e custos, a execução e a entrega do projeto. (SOMMERVILLE, 2011).

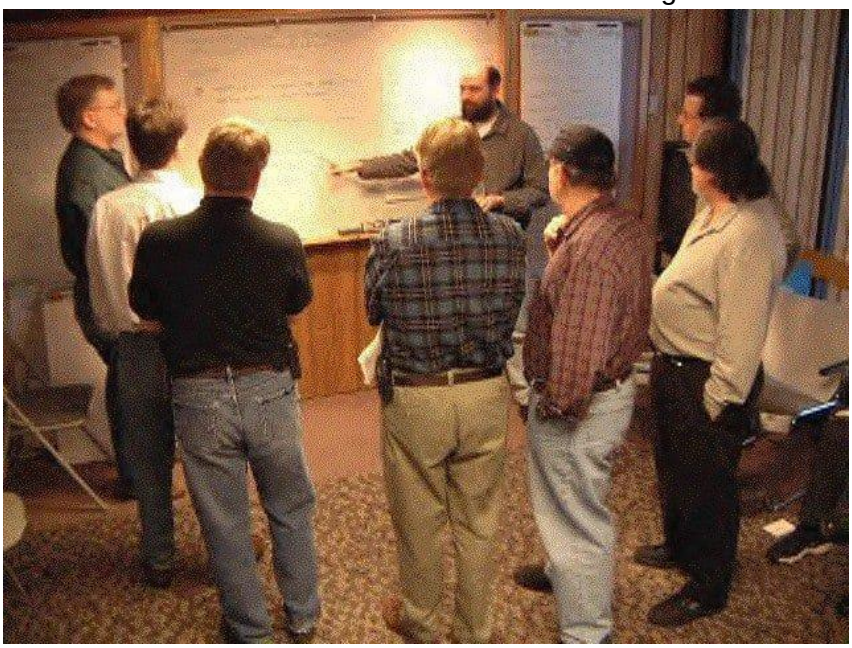
No entanto, os modelos clássicos e em cascatas os quais eram utilizados pelas equipes não atendiam todas as necessidades, tendo em vista que todos os processos a serem percorridos pela equipe eram planejados e elaborados com antecedência, não podendo sofrer alterações, uma vez que o projeto já havia sido iniciado. (SOMMERVILLE, 2011). Por não se adequar à realidade vivida pelas equipes de desenvolvimento, surgiu a ânsia por elaborar novas metodologias mais dinâmicas, que atendessem com maior eficácia suas necessidades, considerando que poderia ocorrer alterações de escopo sem que comprometesse todo o projeto, visando sempre satisfazer cada vez mais o cliente. (AMBLER; HOLITZA, 2012)

Portanto, os processos de desenvolvimento sofreram diversas mudanças desde a década de 60, tendo como grande marco o Manifesto Ágil, ocorrido em 2001, em que um grupo de 17 pessoas inovaram a forma como conduzir o desenvolvimento de um *software*. Atualmente, todas as etapas são bem estruturadas, sendo definido cada processo de desenvolvimento, o que gera maior confiabilidade. (AMBLER; HOLITZA, 2012)

Para todo *software* que venha a ser desenvolvido, se faz necessário, inicialmente, se atentar a alguns processos do projeto, como custos, prazos, qualidade e satisfação do cliente. Processos estes que, se não cumpridos corretamente, acarretará diversos problemas, tanto para a equipe de desenvolvimento quanto para o cliente. Sendo assim, é de extrema importância um bom gerenciamento de projetos, reconhecendo que, a utilização de algumas metodologias ágeis contribui significativamente para uma boa gerência, resultando na satisfação do cliente. (PRESSMAN, 2010)

1 PRINCÍPIOS ÁGEIS: O MANIFESTO ÁGIL E O QUE É SER ÁGIL HOJE

FIGURA 01 – Reunião do Manifesto Ágil



Fonte: https://shortest.link/1X9E

Manifesto Ágil

Um grupo de 17 pessoas se reuniram nos dias 11, 12 e 13 de fevereiro do ano de 2001 na *The Lodge* da estação de esqui *Snowboard*, nas montanhas de *Wasatch* em Utah, nos Estados Unidos. O encontro serviu para conversar, esquiar, relaxar, comer e tentar encontrar um terreno comum. Como resultado desse encontro surgiu o Manifesto Ágil de Desenvolvimento de *Software*. (AMBLER; HOLITZA, 2012)

O encontro foi marcado pela presença de representantes da *Extreme Programming*, SCRUM, DSDM, *Adaptive Software Development, Crystal, Feature-Driven Development, Pragmatic Programming* entre outros, com o intuito de encontrar uma alternativa para os processos de desenvolvimento de *software*. (AMBLER; HOLITZA, 2012)

De acordo com Jim Highsmith (2001), seria difícil encontrar uma reunião maior de anarquistas organizacionais, em que se deu como resultado dessa união um Manifesto para Desenvolvimento Ágil de *Software*, que, por sua vez, foi assinado por todos os participantes. Martin Fowler, britânico, ressaltou na reunião que existia uma única preocupação, ela seria com o termo ágil, pois grande parte dos americanos não conseguiam pronunciar a palavra ágil.

Segundo Jim Highsmith (2001), Alistair Cockburn relatou que suas preocupações inicialmente foram refletidas nos pensamentos de muitos participantes.

> "Eu, pessoalmente, não esperava que esse grupo específico de agilites algum dia concordasse com qualquer coisa substantiva." Mas seus sentimentos pós-reunião também foram compartilhados: "Falando por mim mesmo, estou encantado com a frase final [do Manifesto]. Fiquei surpreso que os outros pareciam igualmente encantados com a frase final. Portanto, concordamos em algo substantivo." (HIGHSMITH, 2001)

Ao final da reunião, a qual durou dois dias, Bob Martin disse, em tom de brincadeira, que estava prestes a fazer uma declaração "piegas". Declarou que todos se sentiram lisonjeados por trabalharem com um grupo de pessoas que possuíam os mesmos valores, um conjunto de valores que se baseiam na confiança e respeito uns pelos outros e também na elaboração de modelos organizacionais fundamentados em pessoas, colaboração e construção dos tipos de comunidades organizacionais nas quais, qualquer um gostaria de trabalhar. Bob acreditava que os metodologistas ágeis eram realmente a respeito de coisas "piegas", sobre como entregar um bom resultado aos clientes, podendo operar em um ambiente que faz mais do que falar sobre. Embora Bob tenha dito tudo com um toque de humor, foram poucos os que discordaram dos sentimentos de Bob. (HIGHSMITH, 2001)

Jim Highsmith (2001) acredita que o *Extreme Programming* cresceu demasiadamente em utilização e interesse, não por conta de ser realizado a programação em pares ou refatoração, mas sim, por um todo, em que as práticas apontam uma equipe de desenvolvedores livres da bagagem das corporações Dilbertesque.

Kent Beck contou uma história, em que um trabalho anterior, foi estimado um prazo para a programação de seis semanas para duas pessoas desenvolverem. Após o gerente atribuir ao outro programador outra função, no início do projeto, ele concluiu o *software* em doze semanas. Ele informou que se sentiu péssimo consigo mesmo. O chefe, por sua vez, falou a Kent que ele foi muito lento durante as segundas seis semanas, isso fez com que ele ficasse mais desanimado, se sentindo um fracassado como programados, quando, finalmente percebeu que ele estava correto em afirmar que levaria seis semanas para entregar o projeto, levando em conta que, se duas

pessoas trabalharem juntas, e não somente uma, como havia ocorrido. (HIGHSMITH, 2001)

Jim Highsmith (2001), acrescentou ainda, que é muito frequente acontecer situações tal qual a citada anteriormente, devido ao fato de que, muitas vezes, a equipe (gestão, desenvolvedores) impõem demandas irracionais através de imposições de estruturas de poder por não quererem tomar decisões complicadas de *trade-off*. Problema este encontrado em todas as organizações, Dilbertesque (empresas que constantemente promovem os colaboradores para funções de gestão, com o intuito de delimitar os danos que serão capazes de cometerem), e não apenas de desenvolvimento de *software*.

Embora muitos não tenham percebido, caro (a) aluno (a), o movimento *Agile* não é contra metodologias, pelo contrário, o movimento tem o intuito de enaltecer a palavra metodologia, fazendo jus a sua real finalidade. A modelagem no movimento *Agile* não é utilizada apenas para acumular diagramas, não tem a finalidade de possuir centenas de páginas que raramente são utilizadas. Realiza-se o planejamento, levando em consideração todos os limites que possam ser encontrados em um ambiente conturbado. (HIGHSMITH, 2001)

Jim Highsmith (2001), menciona que a reunião que ocorreu em *Snowbird* foi incubada em uma reunião anterior de policitantes da Programação Extrema, coordenada por Kent Beck no *Rogue River Lodge* em Oregon, no ano de 2000. Reunião em que os participantes declararam o seu apoio a diversas metodologias "*Light*", embora nada formal tenha ocorrido. No decorrer do ano de 2000, diversos artigos foram produzidos referindo à categoria de processos “leves”, muitos deles apontavam o *Extreme Programming*, a *Adaptive Software Development*, o *Crystal* e o *SCRUM* tais como metodologias leves. Embora a denominação “*Light*” não agradava muito o grupo, ela estava sendo muito apontada de maneira positiva.

Jim Highsmith (2001) conta que Bob Martin, da *Object Mentor*, em Chicago, convocou em setembro de 2000, via e-mail, diversos líderes para mais uma reunião.

No e-mail, Bob Martin dizia:

> (…) Eu gostaria de convocar uma pequena conferência (dois dias) no período de janeiro a fevereiro de 2001 aqui em Chicago. O objetivo

> desta conferência é reunir todos os líderes de métodos leves em uma sala. Todos vocês estão convidados; e eu estaria interessado em saber quem mais devo abordar. (HIGHSMITH, 2001).

Jim Highsmith (2001) relata que Bob criou um site *Wiki*, onde as discussões acirraram. Alistair Cockburn expressou-se com uma mensagem onde constata a sua insatisfação com a palavra *'Light'*:

> (…) Não me importo que a metodologia seja chamada de leve, mas não tenho certeza se quero ser referido como um peso leve participando de uma reunião de metodologistas de peso leve. De alguma forma, soa como um bando de gente leve e magricela tentando se lembrar que dia é. (HIGHSMITH, 2001).

A maior polêmica foi devido à localização, Jim Highsmith (2001) conta que existia sérias apreensões com o inverno de Chicago, pois o caracterizavam como frio e sem diversão. Por fim, Jim Highsmith (2001) diz que:

> (...) Esperamos que nosso trabalho conjunto como *Agile Alliance* ajude outras pessoas em nossa profissão a pensar sobre desenvolvimento de *software*, metodologias e organizações, de maneiras novas e mais ágeis. Nesse caso, alcançamos nossos objetivos. (HIGHSMITH, 2001).

Dezessete (17) participantes do manifesto ágil, denominados "Aliança Ágil":

Segundo Ambler e Holitza (2012), segue a relação dos 17 participantes do manifesto ágil:

- Alistair Cockburn, criador da Metodologia *Agil Crystal*.
- Andrew Hunt, co-autor de O Programador Pragmático.
- Arie van Bennekum, da *Integrated Agile*.
- Brian Marick, cientista da computação e autor de vários livros sobre programação.
- Dave Thomas, programador e co-autor de *The Pragmatic Programmer*.
- James Grenning, autor de *Test Driven Development.*
- Jeff Sutherland, o inventor do *SCRUM.*
- Jim Highsmith, criador do *Adaptive Software Development* (ASD).

- Jon Kern, atuante até os dias de hoje em assuntos de agilidade.
- Ken Schwaber, co-criador do *SCRUM*.
- Kent Back, co-criador da *Extreme Programming* (XP).
- Martin Fowler, desenvolvedor parceiro da *Thoughtworks*.
- Mike Beedle, co-autor de Desenvolvimento *Agil de Software* com *SCRUM*.
- Robert C. Martin, o *"Uncle Bob"*
- Ron Jeffries, co-criador da *Extreme Programming* (XP).
- Steve Mellor, cientista da computação e um dos idealizadores da Análise de Sistema Orientado a Objetos (OOSA).
- Ward Cunningham, criador do conceito *wiki*.

Como o Manifesto Ágil contribui para as empresas de desenvolvimento

Para maior entendimento, caro (a) aluno (a), vou demonstrar para você o quão importante foi o manifesto ágil, como ele contribui para o dia a dia de uma empresa e seus colaboradores.

Vamos imaginar um cenário em que, você, aluno (a), está gerenciando um projeto, você determina que precisará de 4 anos para entregar o *software* (essa prática é extremamente comum em todas as empresas). Você acredita que as necessidades do cliente com esses anos será exatamente igual às necessidades dele hoje? Eu já te adianto que não, pois é impossível prever como estará o mercado daqui a 4 anos, basta ver o quanto o mercado mudou de tantos anos para cá.

Nessa situação, antes do manifesto ágil, você teria duas alternativas, ou entregaria um *software* que não satisfaria o cliente ou teria que voltar e alterar todo o projeto, acarretando em perda de dinheiro e de tempo. Após o manifesto ágil, você, aluno (a), utilizando uma metodologia ágil não enfrentaria problema algum, pois durante os 4 anos de projeto o cliente estaria ao "lado" da equipe, em contato frequente com você. No decorrer do projeto você entregaria ao cliente versões para que ele fosse testando, com os feedbacks do cliente você daria continuidade ao projeto, garantindo, assim, a satisfação do cliente.

As metodologias ágeis são muito bem aceitas pelas equipes e desenvolvimento de sistemas e vem conquistando cada vez mais o seu espaço no mercado, pois tem como princípios básicos a simplicidade e a facilidade a adaptação a mudanças.

2 SOFTWARE ÁGIL: VALORES E PRINCÍPIOS FUNDAMENTAIS

Neste capítulo, caro (a) aluno (a) vamos abordar os 12 princípios do desenvolvimento de software ágil juntamente com os seus 4 valores de desenvolvimento.

Figura 02 – METODOLOGIAS ÁGEIS

SHUTTER: 1376317913

Desenvolvimento de *Software*

O desenvolvimento de *software* ágil consiste em entregar para o cliente o “produto certo”, por meio de entrega incremental e frequente de pequenas funcionalidades do sistema, permitindo, assim, um *feedback* constante do cliente, que, por sua vez, possibilita a realização de correções do *software*, quando necessário. Assim, o desenvolvimento ágil conquistou o seu espaço nas empresas, pois a sua aplicação em um projeto acarreta em um melhor desempenho na produtividade, um produto com maior qualidade e um projeto com um custo reduzido.

Existem diversos aspectos que diferenciam o desenvolvimento de *software* tradicional do ágil, sendo os principais: o ciclo de vida do projeto, a entrega, o planejamento e a execução do projeto. Em outras palavras, a metodologia ágil possibilita a equipe “quebrar” o projeto em partes menores, podendo realizar entregas frequentes, até que todo o projeto seja concluído. Dessa forma, o desenvolvimento ágil possibilitou solucionar um dos maiores desafios encontrados pelas metodologias tradicionais, quando anteriormente se entregava o produto pronto para o cliente, depois de um longo processo de desenvolvimento, processos tão longos que naturalmente faziam com que os requisitos do cliente mudassem, muitas vezes, com frequência, fazendo com o que a equipe de desenvolvimento entregasse o produto final “errado”.

Utilizar as metodologias ágeis não significa que a equipe não deva se preocupar com a qualidade, com os custos e com os prazos do projeto, pelo contrário, são levantamentos indispensáveis em qualquer *software* a ser desenvolvido, as metodologias ágeis da autonomia para a equipe controlar e gerenciar as mudanças necessárias que provavelmente vão surgir no decorrer do *software*. O planejamento da metodologia ágil acontece de forma iterativa e incremental, enquanto nas metodologias tradicionais tudo é planejado com muita antecedência.

Metodologia Ágil

Quando utilizamos no desenvolvimento de um *software* uma metodologia ágil, temos como meta entregar um produto totalmente funcional, que atenda todas as necessidades do cliente, e, para que isso aconteça, é indispensável que exista entre a equipe de desenvolvimento, uma excelente comunicação, (muitas vezes com uso de ferramentas que possibilitam maior facilidade e comunicação) como descrito na Figura 02, sendo indispensável também a motivação e que cada integrante da equipe tenha como objetivo entregar um produto de qualidade. Durante o processo de desenvolvimento, o cliente é chamado frequentemente para intervir, tendo, assim, um papel decisivo na definição de novos requisitos. (TOMÁS, 2009, p. 5)

Caro aluno (a), você precenciará no decorrer da sua carreira com situações em que o plano definido inicialmente pela equipe de desenvolvimento quase sempre não chegará ao final, exatamente igual ao que foi proposto inicialmente, pois não existe projeto previsível, sendo assim, ser “ágil” é aceitar que os requisitos podem sofrer alterações constantes e a função da equipe é estar sempre apta para atender às novas necessidades de forma simples e precisa.

Contudo, ao utilizarmos as metodologias ágeis, visamos a redução dos ciclos de entrega e uma maior adaptabilidade e flexibilidade em relação a alterações ou, até mesmo, o surgimento de novos requisitos dos stakeholders (segundo o filósofo norte-americano Robert Edward Freeman) (1980), *stakeholder* é quando um integrante ou a equipe de desenvolvimento é impactado pelas ações tomadas por uma empresa. *Stakeholders* é o mesmo que parte interessada, assim como o cumprimento de todos os prazos de entrega. (TOMÁS, 2009)

Atualmente existem alguns modelos ágeis de processo de desenvolvimento de software, entre eles, as três mais utilizadas são:

- *SCRUM;*
- *XP (Extreme Programming) ;*
- *Kanban.*

Tanto os *stakeholders* quanto os gestores do projeto tendem a ser extremamente exigentes, reagindo com antecedência ao mercado. Com a utilização de uma metodologia ágil, espera-se, então, que alcancemos uma maior produtividade, maior qualidade, menores custos, versões do sistema sendo entregue com antecedência (mesmo que não seja a versão final), uma conclusão mais rápida do projeto e, por fim, mas não menos importante, a satisfação do cliente. (TOMÁS, 2009)

Devemos considerar os clientes sendo uma parte de extrema importância da equipe de desenvolvimento, tendo em vista que são eles que testam as versões disponibilizadas e nos fornecem um *feedback* em relação aos requisitos e funcionalidades do sistema. Assim, a equipe de desenvolvimento consegue fornecer ao cliente uma versão do *software* bem mais funcional.

Entretanto, caro aluno (a), podemos afirmar que as funcionalidades e um sistema vão literalmente de encontro com as reais necessidades do cliente. Não podemos em hipótese alguma entregar um projeto ao cliente que ele já não o tenha testado. Quando realizamos todos os processos, reduzimos drasticamente a possibilidade de o software não atender às necessidades do cliente.

Benefícios do desenvolvimento Ágil

Dentre inúmeros benefícios para as equipes e desenvolvimento que adotam as metodologias ágeis, está o fato de que os erros que encontramos no desenvolvimento ágil são corrigidos imediatamente durante o andamento natural do projeto, enquanto nas metodologias tradicionais, uma alteração necessária em um requisito geraria uma nova cascata de etapas, o que tornaria o projeto mais demorado e com um custo

elevado, contudo, segue mais alguns benefícios que a equipe encontra dentro do desenvolvimento ágil:

- Levamos um tempo menor para desenvolvimento e consumimos menos recursos, isso só é possível, pois conseguimos ter uma boa comunicação entre todos os envolvidos no projeto;
- É uma abordagem mais flexível, o que torna mais fácil alterar o rumo do projeto;
- É possível realizarmos teste em todos os ciclos do projeto, com eles podemos obter insights ou informações novas, o que reduz muito a possibilidade do software não satisfazer o cliente;
- Com o desenvolvimento ágil podemos melhorar muito o relacionamento entre a equipe, uma vez que é indispensável uma frequente comunicação entre os integrantes;
- As equipes de desenvolvimento tem uma maior autonomia, isso faz com que trabalhem com mais motivação, engajamento e satisfação, tornando-se, assim, muito mais produtivo;
- Ao concluirmos o projeto garantimos a satisfação do cliente, uma vez que todo o desenvolvimento foi direcionado a atender às suas reais necessidades.

Princípios do Desenvolvimento Ágil

Segundo Ambler e Holitza (2012), o Desenvolvimento de *Software* Ágil é composto por 12 princípios elaborados pela Aliança Ágil. Esses princípios têm como objetivo ser utilizado como um guia o qual possibilita um direcionamento para a equipe de desenvolvimento de um projeto, a fim de potencializar seus *softwares* a obterem resultados finais satisfatórios. Contudo, existem, então, 12 princípios a serem seguidos no processo de desenvolvimento de software, sendo eles: (AMBLER; HOLITZA, 2012)

- Satisfação do cliente: considera-se como prioridade satisfazer o cliente, com isso podemos alcançar, através de uma entrega antecipada e contínua de *software* de valor;

- Mudança em favor da vantagem competitiva: consideramos que mudanças de requisitos serão sempre muito bem-vindas, mesmo que o desenvolvimento esteja em fases tardias;
- Prazos curtos: devemos sempre entregar módulos do software em funcionamento com frequência, o intervalo pode ser a cada quinze dias ou, até mesmo, a cada dois meses, levando em consideração que existe uma preferência por prazos mais curtos;
- Trabalho em conjunto: Todos integrantes da equipe de desenvolvimento de *software*, sejam elas as pessoas relacionadas a negócios como os desenvolvedores devem trabalhar sempre em conjunto, diariamente, durante todos os processos do projeto;
- Ambientação e suporte: acredita-se que para construir projetos de qualidade necessitamos ter desenvolvedores motivados, proporcionado a todos da equipe o ambiente e o suporte que precisam, sempre confiando que farão o seu trabalho com excelência;
- Falar na cara: a maneira mais eficiente de transmitir as informações necessárias para uma equipe de desenvolvimento, tanto as informações externas como as internas, se dá por meio de uma conversa cara a cara;
- Funcionalidade: consideramos um *software* funcional sendo a medida primária de progresso;
- Ambiente de sustentabilidade: os processos ágeis proporcionam um ambiente sustentável, com patrocinadores, com desenvolvedores e com os usuários sendo capazes de permanecerem com passos constantes.
- Padrões altos de tecnologia e design: considera-se que manter a atenção à excelência técnica e a um bom design aumenta consideravelmente a agilidade da equipe;
- Simplicidade: quando realizamos algo simples estamos dominando a arte de aumentar consideravelmente a quantidade e trabalho que não necessitou ser feito;
- Autonomia: os melhores requisitos, as melhores arquiteturas e os melhores designs surgem de equipes que trabalham de forma cooperativa, que possuem autonomia e confiança para decidirem o melhor caminho para realizar seus projetos;

- Reflexões para otimizações: com intervalos regulares, o time reflete em como podem ficar mais efetivo, assim, se ajustam e otimizam seu comportamento de acordo com a necessidade.

Os valores do Desenvolvimento Ágil

- Valor ágil I - Indivíduos e interações acima de processos e ferramentas

Durante o desenvolvimento de um *software,* devemos levar em consideração que por ser uma atividade humana a grande aliada no período de desenvolvimento é uma boa comunicação entre a equipe durante todo o processo, isso faz com que seja reduzida todas as dúvidas e ainda aproxima a equipe. As ferramentas e os processos utilizados são de extrema importância, só que devem ser utilizadas de maneira simples. (AMBLER; HOLITZA, 2012)

- Valor ágil II - *Software* funcionando é melhor que documentação abrangente.

A melhor forma de identificar que o trabalho foi executado perfeitamente é visualizar o *software* em perfeito funcionamento, a documentação é muito importante, desde que seja só o necessário, podendo, assim, agregar valor. Vale ressaltar que os clientes estão pagando pelo resultado. (AMBLER; HOLITZA, 2012)

- Valor ágil III - Colaboração com o cliente acima de negociação de contratos.

Devemos ter em mente que sempre temos que trabalhar em parceria com o cliente e em hipótese alguma contra o cliente. Assim o cliente contribuirá muito em cada tomada de decisão, tornando-se todos uma só equipe em busca de um objetivo em comum. (AMBLER; HOLITZA, 2012)

- Valor ágil IV – Responder a mudanças ao invés de seguir um plano

Devemos ficar atentos a todos os feedbacks coletados do cliente para que possamos adaptar o projeto sempre que necessário. (AMBLER; HOLITZA, 2012)

Sendo assim, caro (a) aluno (a), quando fazemos parte de uma equipe de desenvolvimento temos a consciência que estamos em um ambiente de muitas incertezas, e que os planos inicialmente calculados nem sempre serão executados, o que não significa que não se deve planejar os processos com antecedência, só que

necessitamos estar sempre preparados para executar qualquer alteração que for necessária.

3 DESAFIOS E EVOLUÇÃO DA ENGENHARIA DE SOFTWARE

Neste capítulo, caro (a) aluno (a) vamos abordar sobre a engenharia de *software*, tratando sobre os processos de *software* e os modelos de ciclo de vida de um *Software.*

Engenharia de Software

A Engenharia de *Software* é nada mais que a aplicação de métodos, modelos, padrões, princípios científicos e teorias que possibilitam gerenciar, planejar, modelar, projetar, implementar, medir, analisar, manter e refinar um *software* (SOMMERVILLE, 2011). A Engenharia de *Software* proporciona um maior rendimento no decorrer do desenvolvimento, garantindo que todos os processos que foram cuidadosamente elaborados sejam cumpridos com eficiência, tendo como objetivo principal obter um *software* de qualidade, satisfazendo, assim, o cliente. (PRESSMAN, 2010)

Processo de Software

O processo de desenvolvimento de *software* é um conjunto de tarefas que são capazes de proporcionar resultados que irão conduzir à produção de um produto de software. O processo de *software* fica compreendido como sendo as etapas a serem estruturadas para construir, implantar e manter o *software*. Algumas atividades são frequentes dentro do processo de desenvolvimento de *software*, como os Requisitos, Projeto e Implementação, Validação e a Evolução. (SOMMERVILLE, 2011)

- **Requisitos**

A tarefa de levantamento de requisitos, também conhecida como Elicitação de requisitos, consiste em identificar e extrair todas as funções que o usuário necessita executar dentro do *software* a ser desenvolvido, bem como informações pertinentes a

restrições e acesso, premissas e entre outras. De acordo com Sommerville (2011), o êxito de todas as etapas posteriores depende exclusivamente da qualidade dos requisitos conseguidos. Portanto, é notável que se faz necessário executar o processo de elicitação com extrema cautela, abstraindo todas as informações necessárias.

- **Projeto e Implementação**

É nesta fase, caro (a) aluno (a), que o *software* passa a ser decodificado, respeitando todas as especificações levantadas anteriormente. A equipe de desenvolvimento codifica os modelos oferecidos em forma de diagramas, utilizando a linguagem de programação necessária, já estabelecida inicialmente no projeto. (SOMMERVILLE, 2011)

- **Validação**

No processo de validação, são realizados todos os testes necessários, a fim de garantir que o *software* desenvolvido opera todas as suas funcionalidades com eficiência. (SOMMERVILLE, 2011). Nessa fase, é analisado também se o *software* foi desenvolvido de acordo com o levantamento realizado inicialmente com o cliente, tendo em vista a qualidade do produto e a satisfação do cliente. São exemplos de alguns testes de software:

- Teste de instalação e configuração: é verificado como o sistema se comporta em diversas configurações de *hardware* e *software*;
- Teste de integridade: momento em que é verificado se os componentes vão manter-se íntegros quando expostos a um grande volume de dados, quando, por exemplo, é verificado o comportamento de uma tabela com milhões de registros;
- Teste de Segurança: momento em que o sistema é testado a fim de identificar se existem falhas de segurança, avaliando as vulnerabilidades do *software*;
- Teste funcional: período utilizado, a fim de identificar se o *software* está apto para efetuar as operações que foi desenvolvida para realizar;

- Teste de unidade: momento em que fornecemos alguns valores, sendo eles válidos ou inválidos, a fim de identificar se o retorno está de acordo com o esperado;
- Teste de integração: momento em que os módulos são integrados e testados em grupo. Como, exemplo, podemos citar o momento em que um *software* acessa um determinado banco de dados;
- Teste de volume: o sistema é submetido a uma determinada quantidade de dados, a fim de analisar o comportamento do *software*;

Podemos citar como, por exemplo, quando executamos uma consulta ao banco de dados, solicitando que retorne todo o conteúdo que nele contém.

- Teste de performance: momento em que o *software* é testado sendo avaliado a sua capacidade de resposta, a sua disponibilidade, confiabilidade e robustez, de acordo com a quantidade de conexões simultâneas, considerando o seu comportamento em situações normais e em situações de alta carga de trabalho;
- Teste de usabilidade: período em que é considerado se o *software* é, em geral, fácil de ser utilizado por um cliente, possuindo botões com ações facilmente e rapidamente identificadas. Aqui se faz necessário analisar também se o *software* necessita de alguma alteração quanto às suas cores, fontes ou imagens, pois os itens citados podem prejudicar a usabilidade do sistema;
- Teste de regressão: técnica utilizada que consiste na aplicação de versões mais recentes do sistema, garantindo, assim, que não vá surgir nenhum problema em componentes que já foram testados anteriormente.

- **Evolução**

De acordo com Sommerville (2011), todo *software* em funcionamento necessita de manutenção, sendo que ele precisa sofrer atualizações, a fim de continuar sendo útil ao usuário.

Modelos de ciclo de vida de *Software*

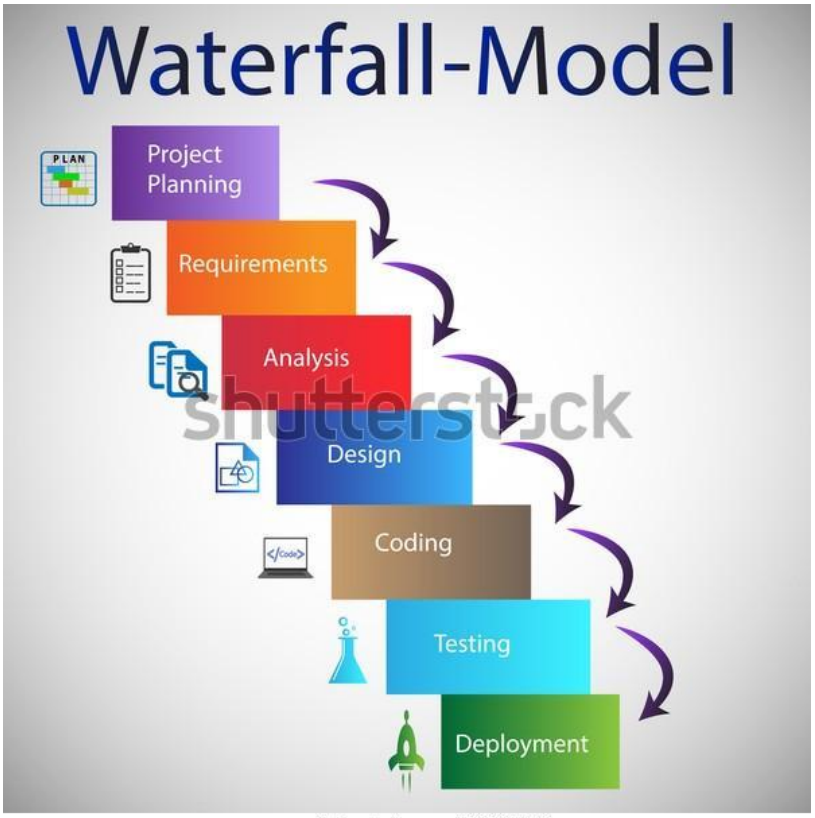
Existem inúmeros modelos de ciclo de vida de *software* e as suas intitulações podem diferenciar conforme os autores Sommerville (2010) e Pressman (2001) relatam em suas obras alguns modelos de ciclo de vida, iremos, agora, retratar os principais abaixo:

- **Modelo em Cascata**

Modelo, também conhecido como clássico, foi a primeira tentativa de se oficializar uma metodologia de desenvolvimento de *software*, o qual consiste em respeitar cada processo, em que uma etapa necessita ser encerrada para que outra etapa possa ser iniciada, ou seja, se uma etapa do projeto ainda não foi encerrada não se pode iniciar outra até que ela seja concluída. Tendo também a executar exatamente o que foi planejado inicialmente, sendo assim, o Modelo Cascata, representado na Figura 01, não possui uma flexibilidade quanto a alterações no projeto quando já iniciado. (PRESSMAN, 2010)

Embora seja um modelo considerado fácil de ser utilizado pelas equipes de desenvolvimento, pois possui etapas bem definidas, o Modelo Cascata não é muito dinâmico, sendo assim, não muito indicado a sua utilização. Acontece que, caso seja necessário corrigir um erro identificado na fase de testes, por exemplo, o ato geraria um custo elevado para a correção do projeto. (PRESSMAN, 2010). Estudos apontam que, o custo gerado em correções de um *software* que se encontra em desenvolvimento aumenta exponencialmente 10 vezes a cada fase que o projeto progride sem que seja feita a correção necessária. Portanto, Myers (1979) destaca que os problemas que são identificados nos processos iniciais do desenvolvimento acabam saindo muito mais em conta de serem corrigidos, em comparação quando os problemas são encontrados na fase de produção.

Figura 02 – MODELO CASCATA



Fonte: 419198770

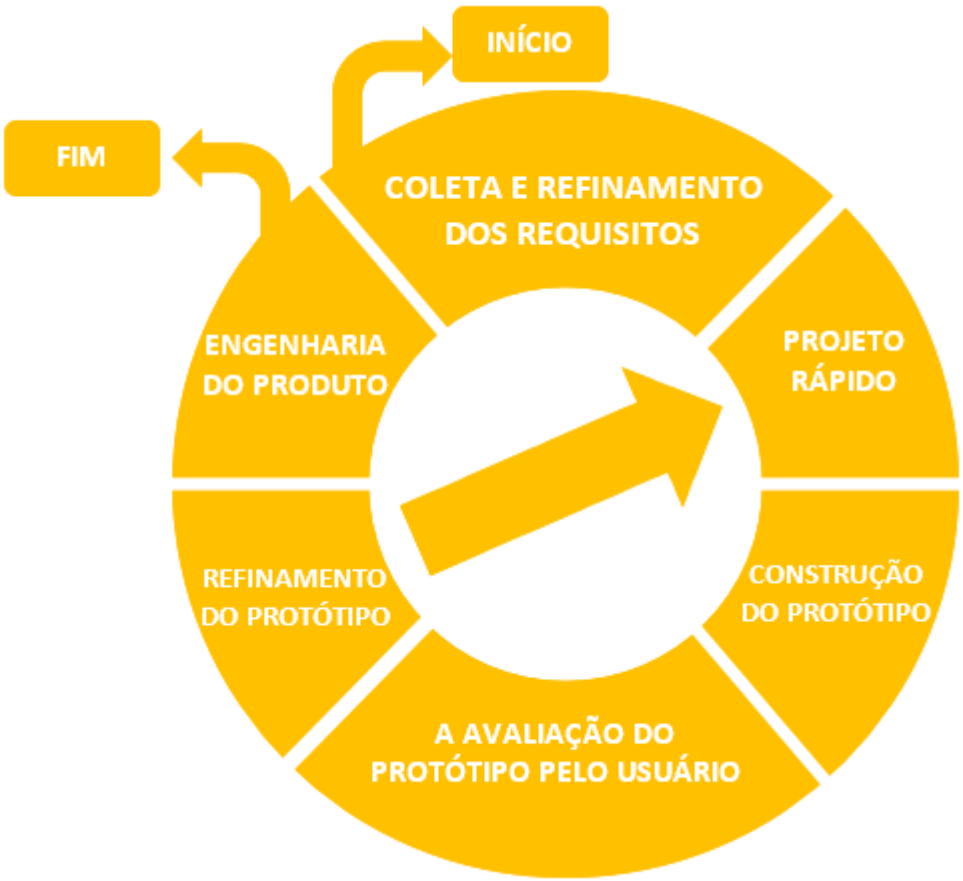
- **Prototipação**

A prototipação, segundo K. Laudon e J. Laudon (1999), envolve a construção de um *software* experimental ou somente parte dele, de maneira rápida, para que assim os usuários finais possam avaliá-lo, como mostra a Figura 03. Com a utilização do protótipo pelo usuário, o qual consegue identificar de maneira mais precisa quais são as reais necessidades da empresa, assim, as funcionalidades do *software* podem ser adaptadas de acordo com as exigências identificadas durante a utilização do protótipo. Podemos, então, classificar a prototipação como um artifício para visualizar as necessidades do cliente antes mesmo de tirá-la do papel, sendo assim, se torna uma forte aliada para as startups, considerando que a prototipação evita gastos elevados no processo de desenvolvimento, e tem como objetivo validar todos os requisitos do usuário. (PRESSMAN, 2010)

Podemos relatar, caro (a) aluno (a), que a prototipação possui alguns pontos fortes, tais como, por exemplo, a eficácia da sua utilização quando os requisitos fornecidos pelo usuário não são tão claros e precisos, assim, possibilitando, a participação do usuário em todas as etapas de desenvolvimento, ela permite testes

de interface com os usuários, permitindo também o contato imediato do usuário com determinadas telas do sistema, além ainda de facilitar o processo de solução de problemas identificados na utilização dos protótipos. Contudo, não se recomenda a utilização da prototipação em sistemas de grande porte e ela não substitui a análise detalhada necessária para a construção de um software. (PRESSMAN, 2010)

FIGURA 03 – PROTOTIPAÇÃO



Fonte: Elaborada pela autora.

- **Modelo Iterativo e Incremental**

O desenvolvimento incremental baseia-se no contexto de desenvolver uma implementação inicial, colocá-la à disposição dos usuários para que possam efetuar as observações necessárias e dar continuidade através de diversas versões até que, por fim, um *software* apropriado seja desenvolvido. (SOMMERVILLE, 2011)

De acordo com Sommerville (2011) podemos citar algumas características do Modelo Iterativo e Incremental, podemos enfatizar que:

- Raramente se estabelece uma solução completa do problema com antecedência;
- Geralmente os processos são realizados rumo a uma solução, recuando quando se torna perceptível a ocorrência de algum erro;
- Torna o projeto mais barato e mais fácil realizar alterações no *software*;
- Cada versão ou incremento do *software* acrescenta alguma funcionalidade necessária para o usuário;
- A versão inicial inclui as funcionalidades mais importantes para os usuários;
- Possibilita que o usuário avalie o *software* em um estágio considerado inicial, podendo informar a equipe se ele oferece o que foi requisitado;
- Caso o usuário identifique que as suas necessidades não estão sendo atendidas, só o incremento que estiver em desenvolvimento no momento necessitará ser alterado;
- O Desenvolvimento Iterativo e Incremental é a abordagem mais comum;
- Ela pode ser dirigida a planos, ágil ou a junção de ambas;

✔ Dirigida a planos: cada incremento do *software* é apontado antecipadamente;

✔ Ágil: os incrementos iniciais são apontados, só que os incrementos posteriores serão desenvolvidos de acordo com o progresso e prioridades dos usuários.

Ainda de acordo com Sommerville (2011) podemos citar algumas limitações do modelo iterativo e incremental, podemos enfatizar que:

- O processo não é visível, sendo necessário entregas constantes para assim poder mensurar o progresso do desenvolvimento;
- A estrutura do *software* acaba se degradando ao adicionar novos incrementos.

Contudo, no Modelo Iterativo e Incremental, apresentado na Figura 04, o usuário realiza a avaliação antecipadamente com as entregas, assim os detalhes que necessitam de correções podem ser alterados mais cedo, reduzindo, assim, as chances de o projeto não atender às necessidades do cliente. Todas as iterações

seguintes serão planejadas com base na avaliação realizada pelo usuário no incremento anterior. (SOMMERVILLE, 2011)

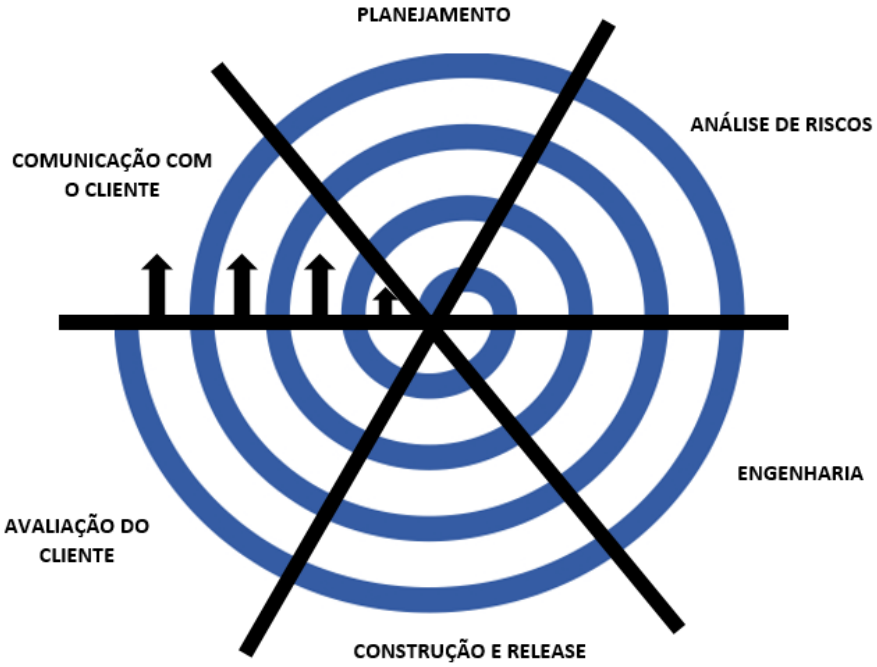
Figura 04 – MODELO ITERATIVO E INCREMENTAL

Fonte: Elaborada pela autora.

- **Modelo Espiral**

Com o intuito de melhorar as características dos modelos já abordados anteriormente, Barry Boehm propôs em 1988 o Modelo Espiral, apresentando um formato muito mais rápido de desenvolvimento das versões, sendo elas mais completas (PRESSMAN, 2011).

FIGURA 05 – MODELO EM ESPIRAL



Fonte: Elaborada pela autora.

O modelo espiral adotou alguns aspectos gerenciais, como planejamento e análise de riscos. (PRESSMAN, 2011). De acordo com a Figura 05, o modelo apresenta uma divisão no processo de avaliação de *software* que se resume em 4 principais atividades:

- Planejamento: momento em que se define os objetivos, possibilidades e restrições;
- Análise de riscos: análise de possibilidades e levantamento de riscos;
- Engenharia: desenvolvimento do projeto;
- Atualização realizada pelo cliente: avaliação dos resultados da engenharia.

No modelo espiral, o processo se inicia a partir do centro, cada volta que se completa é uma nova iteração que acontece e, consequentemente, uma nova versão, cuja versão é disponibilizada de maneira mais completa que a anterior. Cada *loop* da espiral é definido de acordo com o que for mais necessário. (PRESSMAN, 2011).

SAIBA MAIS

As metodologias ágeis não param de crescer, e para ser bem sincera com você, elas nem devem. Segundo o *State of Scrum* de 2021, levantamento elaborado pela *Scrumalliance.org*, o qual realizaram uma pesquisa com 4.182 empresas para identificar a quanto tempo elas estão utilizando metodologias ágeis, identificaram que 32% dos entrevistados utilizam a mais e 5 anos, 62% utilizam de 1 até 5 anos, 4% respondeu não saber o tempo exato e somente 2% informou que não faz uso de nenhuma metodologia ágil. (STATE OF SCRUM, 2021)

REFLITA

As metodologias ágeis têm como função apoiar a equipe de desenvolvimento quando os requisitos sofrem alterações durante o processo, lembre-se, é indispensável que a equipe mantenha boa comunicação. (HIGHSMITH, 2001)

CONSIDERAÇÕES FINAIS

Podemos afirmar que o desenvolvimento de *software* é uma tarefa muito complexa, pois traz consigo diversas variáveis. Um levantamento de requisitos realizado inicialmente muitas vezes não coincide com as necessidades do cliente ao final do projeto, pois não é possível prever como vai estar o mercado no futuro. Diante dessa situação, anteriormente, teríamos que realizar todas as correções necessárias, o que levaria um tempo maior e um gasto elevado. Problemas como este, estavam se tornando comum entre as empresas, foi o que aconteceu com o manifesto ágil em que 17 pessoas se reuniram para encontrar uma maneira de inovar e sancionar todos os problemas enfrentados no decorrer dos tempos. A solução encontrada foi a utilização das metodologias ágeis pelas equipes de desenvolvimento dentro das empresas.

O manifesto ágil trouxe, sem dúvidas, uma evolução gigantesca para as equipes de desenvolvimento, possibilitando uma troca de experiência, melhor comunicação, confiança e segurança entre as equipes e uma confiabilidade maior quanto ao cliente. Todos esses pontos fazem com que a produtividade da equipe aumente e que a satisfação do cliente seja maior.

As metodologias ágeis têm como princípios básicos a simplicidade e a fácil adaptação às mudanças necessárias no decorrer do processo de desenvolvimento. (AMBLER; HOLITZA, 2012). Isso porque o cliente se faz presente em praticamente todos os processos, fornecendo *feedbacks* que auxiliam no andamento do projeto. Seus princípios básicos fizeram com que fossem muito bem aceitas pelas equipes de desenvolvimento e, consequentemente, vem ganhando cada vez mais espaço no mercado.

Contudo, a chegada das metodologias ágeis trouxe maior confiabilidade e garantia de que a equipe de desenvolvimento vai realizar todos os processos como esperado e que o cliente receberá seu produto realmente funcionando, pois, esse é realmente o resultado que toda empresa precisa e espera.

REFERÊNCIAS

MYERS, Glenford J. **The art of software testing**. New York: John Wiley & Sons, 177p, 1979.

AMBLER, Scott and HOLITZA,Matthew. **Agile for Dummies**. Ed 2. John Wiley &Sons,Inc, 2012. ISBN: 978-1-118-30554-6.

PRESSMAN, R. **Engenharia de Software**, 6 edição, AMGH Editora. 2010.

LAUDON, K. C.; LAUDON, J. P. **Sistemas de informação**. 4. ed. Rio de Janeiro: LTC, 1999.

SOMMERVILLE. **Engenharia de Software**, São Paulo: Addison-Wesley, 9 ed., 2011. ISBN-10: 8579361087 ISBN-13: 9788579361081.

HIGHSMITH, J. **History: The Agile Manifesto.** 2001. Disponível em: https://agilemanifesto.org/history.html. Acesso em: 12 de setembro de 2021.

SCRUM ALLIANCE. **State of Scrum de 2017/2018** - Disponível em: Scrumalliance.org. Acesso em:12 de setembro de 2021.

UNIDADE II

PRINCIPAIS PRÁTICAS DOS MÉTODOS ÁGEIS

Professora Especialista Camila Sanches Navarro

Plano de Estudo:

- Desenvolvimento Dirigido por Testes;
- Programação pareada;
- Refatoração;
- Integração contínua;
- Ciclo de vida do desenvolvimento Ágil de software;
- Ferramentas de apoio: User Story, Planning Poker e Burndown;
- O processo visionário.

Objetivos de Aprendizagem:

- Compreender como funciona o desenvolvimento dirigido por testes, a programação pareada, a fatoração e a integração contínua;
- Entender como devemos utilizar as ferramentas de apoio *User Story, Planning Poker* e o gráfico *Burndown;*
- Obter entendimento sobre o processo visionário.

INTRODUÇÃO

Existem diversos métodos que podemos adotar em um processo de desenvolvimento de *software*, dentre eles, citamos no decorrer do material o desenvolvimento dirigido por testes, considerado um método muito útil, pois cada funcionalidade deve ser testada de forma individual. (FREEMAN, 2012). A programação pareada consiste em duas pessoas trabalharem juntas em uma única funcionalidade, em que uma escreve o código, enquanto a outra revisa todo o código que está sendo digitado. (BECK, 2000)

De acordo com Beck (2010), a Refatoração é o processo de realizar alterações em um *software* sem que seja alterado o seu comportamento externo, melhorando a sua estrutura interna. Já o método de integração contínua é explicado por Fowler (2006), ele ressalta que é o processo cujo os desenvolvedores juntam todas as suas alterações no código em um repositório central, e este processo ocorre com frequência.

Segundo Norén (2021), o início e o final de cada fase de um ciclo de vida de *software* ágeis, representa um ponto de revisão do trabalho feito e do que será produzido. Dessa forma, fica bem mais simples identificar e corrigir erros que possam ter surgido.

Existem algumas ferramentas que auxiliam de forma significativa a equipe de desenvolvimento, um exemplo seria a *User Story* (história de usuário), que consiste em descrever com precisão as necessidades do cliente de acordo com o ponto de vista dele, sendo essa uma forma de centralizar o cliente nas tomadas de decisões. (COHN, 2004b). Outra ferramenta muito utilizada é o *Planning Poker*, ele auxilia o time a estimar quanto tempo cada tarefa necessita para ser cumprida, isso através de um “jogo de cartas”. Rubin (2013) apresenta uma outra ferramenta, o gráfico de *Burndown*, ele é muito útil para prever quando o processo será concluído, ele é frequentemente utilizado na metodologia *Scrum*.

Por fim, trataremos sobre o processo visionário,em que Filion (1991) explica que para ser visionário é preciso possuir algumas habilidades bem específicas.

1 DESENVOLVIMENTO DIRIGIDO POR TESTES

O desenvolvimento dirigido a testes, popularmente conhecido como TDD (*Test-Driven Development*), é uma abordagem utilizada no desenvolvimento de *softwares* em que alternamos entre testes e o próprio desenvolvimento. (BECK, 2002; JEFFRIES e MELNIK, 2007). Basicamente, caro (a) aluno (a), você desenvolve um código de forma incremental, juntamente com um teste para esse incremento e, dessa forma, você não pode passar para o próximo incremento até que tenha realizado o teste no incremento atual. O desenvolvimento dirigido a testes foi mostrado como sendo parte dos métodos ágeis, como o *Extreme Programming*, porém ele pode ser utilizado também em processos de desenvolvimento dirigido a planos.

Contudo, podemos afirmar que o TDD transforma o desenvolvimento de *software*, levando em conta que antes de implementar o sistema devemos primeiro escrever os testes. Segundo Freeman (2012), utilizamos os testes, a fim de clarear a ideia em relação ao que se deseja com o código.

Podemos considerar que a elaboração de testes unitários é de extrema importância para o TDD, de acordo com Pressman (2010), devemos testar os componentes individuais, a fim de garantir que estão operando corretamente. Os componentes são testados de forma independente, sem os outros componentes do *software*.

Já Ian Sommerville (2011), afirma que os componentes são integrados com a finalidade de compor o sistema, assim, não devemos realizar somente os testes unitários, necessitamos também testar o *software* como um todo. Esse processo é fundamental para identificarmos se o sistema apresenta algum erro gerado pelas interações.

Para maior entendimento, caro (a) aluno (a), vamos imaginar o processo e montagem de um carro, todas as peças devem ser testadas antes de montar o automóvel, seja ela um volante, uma roda, uma porta, um acento e todas as demais partes. Após montar o carro, devemos testar se ele está funcionando da maneira esperada e, se ao fazermos uma curva, por exemplo, ele vai realizar a operação com

perfeição. Ou seja, não adianta eu testar separadamente a roda, o volante e a porta se eu não testar o carro após montado, pois pode ocorrer algum erro no processo de montagem (integração) dos componentes. Assim como no exemplo citado, um *software* é um conjunto de unidades integradas, sendo necessário realizar os testes individuais e também os testes para analisar a integração de dois ou mais componentes. (MASSOL e HUSTED, 2003)

Processo fundamental de TDD

Podemos dizer que o ciclo do desenvolvimento dirigido a testes é bem simples, o iniciamos, criando um teste, na sequência, fazemos a codificação para passar no teste e, por fim, refatoramos o nosso código. Observe, caro (a) aluno (a), que esse teste tem por finalidade auxiliar a codificação, reduzindo, assim, os problemas que poderiam ocorrer no processo de desenvolvimento. (SOMMERVILLE, 2011)

Vamos voltar ao exemplo do carro citado anteriormente, imagine que você necessita testar um pneu que foi criado para o automóvel, você precisa compreender que o pneu é redondo para criar um teste para pneu redondo, sem essa compreensão, por exemplo, você criaria um teste para pneu triangular, assim não seria possível realizar o teste. Por isso, é muito importante que haja uma compreensão completa dos requisitos do cliente, pois, para que seja possível implementar os testes iniciais, devemos entender todas as especificações do sistema, além da regra de negócio.

Segundo Ian Sommerville (2011), para que possamos utilizar o TDD (*Test-Driven Development*), necessitamos seguir o modelo F.I.R.S.T, o qual é explicitado abaixo:

- F (*Fast*): os testes precisam ser rápidos, pois vão testar apenas uma unidade;
- I (*Isolated*): os testes unitários são isolados, testando, assim, as unidades individualmente, e não a sua integração;
- R (*Repeatable*): devemos repetir os testes, com resultados de comportamento frequente;
- S (*Self-verifying*): a auto verificação deve verificar se o teste passou ou falhou;
- T (*Timely*): O teste deve ser apropriado, sendo um só teste por unidade.

Como implementar no desenvolvimento o processo de TDD

É indispensável para implementar o processo de TDD (*Test-Driven Development*) no desenvolvimento de um *software* a utilização de um ambiente de testes automatizados, como, por exemplo, o ambiente *JUnit,o qual* é um *framework* utilizado para escrever testes repetíveis e automatizados. Esse *framework* possui suporte para a linguagem de desenvolvimento *Java* -, devido ao fato do código ser desenvolvido em pequenos incrementos, pois devemos executar todos os testes toda vez que adicionarmos uma nova funcionalidade ou após refatorar o *software*. (MASSOL e HUSTED, 2003)

Além da *JUnit*, temos também outras bibliotecas *XUnit's*, podemos citar como exemplo as seguintes:

- TesteNG: ferramenta de teste unitário, disponível para a linguagem de desenvolvimento *Java*;
- *PHPUnit*: ferramenta de teste unitário, disponível para a linguagem de desenvolvimento PHP;
- *SimpleTest*: ferramenta de teste unitário, disponível para a linguagem de desenvolvimento PHP;
- *NUnit:* ferramenta de teste unitário, disponível para a linguagem de desenvolvimento. NET;
- Jasmine: ferramenta de teste unitário, disponível para a linguagem de desenvolvimento *JavaScript;*
- *CUnit*: ferramenta de teste unitário, disponível para a linguagem de desenvolvimento C;
- *PyUnit*: ferramenta de teste unitário, disponível para a linguagem de desenvolvimento *Python*.

Assim sendo, caro (a) aluno (a), após escolher a ferramenta que vai ser utilizada, você deve embutir os testes nela, pois é onde o teste será executado. (SOMMERVILLE, 2011). Quando utilizamos essa abordagem, podemos rodar centenas de testes em questão de segundos. Como observamos, existem ferramentas

para diversas linguagens de desenvolvimento, escolha a sua e lembre-se, primeiro você deve criar os testes, para só depois efetuar a sua implementação.

Vantagens do Desenvolvimento Dirigido a Testes

De acordo com Ian Sommerville (2011), o desenvolvimento dirigido a testes auxilia os desenvolvedores a identificar o que um trecho do código supostamente deve fazer. Para que o programador possa escrever um teste, ele necessita compreender para que ele se destina. Por exemplo, se o seu cálculo envolve divisão, você precisa averiguar se não está dividindo o número por zero, caso você não se lembre de escrever um teste para essa funcionalidade, o código não será incluído no *software*. Em resumo, o TDD (*Test-Driven Development*), possibilita uma melhor compreensão em relação ao problema.

Kent Beck (2002), aponta mais quatro benefícios do desenvolvimento dirigido a testes, sendo eles:

- Cobertura de código: como todo segmento de código necessita ter pelo menos um teste associado, conseguimos garantir que todo o código no sistema foi realmente executado. Assim, podemos descobrir as falhas logo no início do processo de desenvolvimento, pois o código é testado enquanto está sendo escrito;
- Teste de regressão: sempre que necessário podemos executar testes de regressão, a fim de averiguar se as mudanças no sistema não introduziram novos *bugs*, isto só é possível devido ao fato de que, enquanto um *software* é desenvolvido, simultaneamente desenvolvemos um conjunto de testes;
- Depuração simplificada: quando um teste falha, nós conseguimos localizar o problema com muita facilidade, sem ser necessário utilizar ferramentas de depuração para identificar o erro, pois a cada “pedaço” de código escrito é executado os testes, assim o erro encontra-se no código recém escrito;
- Documentação de sistema: ler os testes torna mais simples a compreensão do código, pois eles descrevem o que o código deve estar fazendo. Assim, podemos dizer que, os testes escritos servem como uma forma de documentação;

- Redução de custos: um projeto que utiliza o desenvolvimento dirigido a testes não necessita realizar testes de regressão - o teste de regressão identifica se as mudanças realizadas não ocasionaram em novos *bugs* no sistema, além de verificar se o código recém-escrito interage como esperado com o código já existente -. O teste de regressão tem um custo muito alto.

O desenvolvimento dirigido a testes mostrou ser uma abordagem de sucesso para projetos de pequenas e médias empresas. Muitas equipes que adotaram a abordagem TDD (*Test-Driven Development*) ficaram satisfeitas com o resultado, pois é uma forma mais eficaz de desenvolver *softwares*. (JEFFRIES e MELNIK, 2007).

2 PROGRAMAÇÃO PAREADA

Figura: Programadores desenvolvendo em par

Fonte: 1016500210

Você já ouviu falar em programação em par?

Essa prática consiste em duas pessoas trabalharem lado a lado no mesmo projeto, no mesmo algoritmo, código ou testes, de forma colaborativa, compartilhando o mesmo computador (BECK e ANDRES, 2004). Essa prática de programação pareada por ser integrada em qualquer processo de *software*.

Sabe aquele ditado, que diz que duas cabeças pensam melhor do que uma? Ele se aplica na programação pareada. A programação em pares é dividida em dois papéis, o de piloto e o de navegador.

O piloto é a pessoa que digita o código e pensa de forma tática em como vai completar o processo atual, explicando suas ações sempre em voz alta, enquanto digita. Já o navegador fica responsável por rever as linhas de código enquanto estão sendo escritas pelo piloto, fornecendo mais segurança e confiabilidade ao piloto. O navegador precisa pensar de forma estratégica nos problemas futuros e sugerir soluções para o piloto. Essa prática requer muito diálogo e concentração, e é de extrema importância que os pares realizem troca de papéis com frequência. (BECK e ANDRES, 2004).

Kent Beck (2000), ressalta que a programação em par não consiste em uma sessão de tutoria, a qual um desenvolve e a outra pessoa fica apenas, observando. Pelo contrário, a programação em par é uma prática dinâmica e colaborativa. Existem

algumas orientações sobre como funciona a dinâmica da programação em par, sendo elas:

- Quando formar pares: os pares podem ser formados para qualquer coisa dentro do projeto. (BECK e ANDRES, 2004, p. 42);
- Como compor os pares: a formação dos pares deve acontecer de maneira natural, procurando trocar de pares aleatoriamente com uma certa frequência, buscando assim trabalhar com todos da equipe. (TELES, 2004);
- Como agir durante o pareamento: para programar em par é necessário existir muita comunicação entre a dupla, conversar sobre objetivos em curto-prazo, orientações sobre qualquer coisa que seja relevante para o projeto. (SHORE e WARDEN, 2008, TELES, 2004);
- Revezamento: é necessário trocar de papel com frequência, revezando entre os papéis de piloto e de navegador. (BECK e ANDRES, 2004);
- Rodízio de par: ao término de cada tarefa, pode ser feita a troca de pares, isso pode acontecer também quando o projeto é muito extenso, podendo trocar de par, em média, a cada 4 horas, ou ainda, quando é necessário ter uma nova perspectiva do problema ou do processo. (SHORE e WARDEN, 200)

Contudo, caro (a) aluno (a), existem alguns benefícios que a programação em par no processo de desenvolvimento de *software* pode proporcionar para a equipe, são elas: (POPPENDIECK e POPPENDIECK, 2014; SHORE e WARDEN, 2008; TELES, 2004):

- Como dito anteriormente, que duas cabeças pensam melhor do que uma, a programação pareada possibilita a solução de problemas complexos em um tempo reduzido e com mais qualidade;
- Como é realizado a revisão do código que está sendo escrito com frequência, reduzimos drasticamente a possibilidade de erros no código;
- Por estarem em pares, os programadores conseguem focar integralmente na atividade em desenvolvimento, resultando, assim, em um trabalho mais produtivo;
- Alcançamos uma maior produtividade, principalmente em relação ao tempo de desenvolvimento do projeto;

- Conseguimos praticamente igualar o conhecimento da equipe, tendo em vista que ocorre a troca de conhecimentos no rodízio de pares e de papéis, nivelando, assim, o potencial para aprendizado, devido a comunicação constante entre os pares;
- Alcançamos uma maior satisfação no trabalho, isso se deve a maior colaboração entre os integrantes da equipe.

3 REFATORAÇÃO

De acordo com Kent Beck (2010), o desenvolvimento dirigido a testes utiliza a refatoração de maneira inovadora. Normalmente, em uma refatoração, não se pode mudar a semântica do *software* sob qualquer circunstância, mas em TDD (*Test-Driven Development*), as circunstâncias com as quais nos importamos são os testes que já passaram. A única variável na qual a semântica é preservada pode ser o caso de teste em particular. Essa “equivalência observacional” impõe um dever para que tenha testes suficientes, pois uma refatoração com respeito aos testes é o mesmo que uma refatoração com respeito a todos os testes possíveis. Esses modelos mostram como alterar o projeto do sistema, mesmo que drasticamente.

Para maior compreensão e aprendizado, segue os padrões que podem ser utilizados segundo Kent Beck (2010):

- **Reconciliar diferenças (*Reconcile Differences*):** É uma maneira de unificar dois trechos simples de um código, aproximando-os de modo que apenas quando eles forem idênticos ocorra a unificação. Esse processo pode ser estressante, já que, se feito de forma mecanicamente correta, existe pouquíssima possibilidade de alterar o comportamento do sistema. Algumas refatorações levam a analisar os fluxos de controle e valores de dados com muita atenção. Um padrão de ideias pode conduzir você a crer que as modificações que foram realizadas não alteraram o resultado. (BECK, 2010). Kent Beck (2010) ressalta que a refatoração, com esse tiro no escuro, é o que devemos a todo custo evitar com as estratégias de respeitar os passos necessários e *feedbacks* precisos. Devemos considerar que as refatorações saltadas, nem sempre serão evitadas, muitas das vezes conseguimos apenas reduzir a suas incidências.
- **Isolar mudança (*Isolate Change*):** Kent Beck (2010) relata que, para modificar uma parte específica de um método ou de um objeto multi-parte, é necessário isolar a parte a qual será modificada. Como em um procedimento cirúrgico que a equipe médica isola toda e qualquer área, exceto aquela que será operada, isso reduz as variáveis. Você pode imaginar que, quando isolada

a mudança e, então, realizada a modificação, o resultado é tão banal que pode desfazer o isolamento de maneira simples.

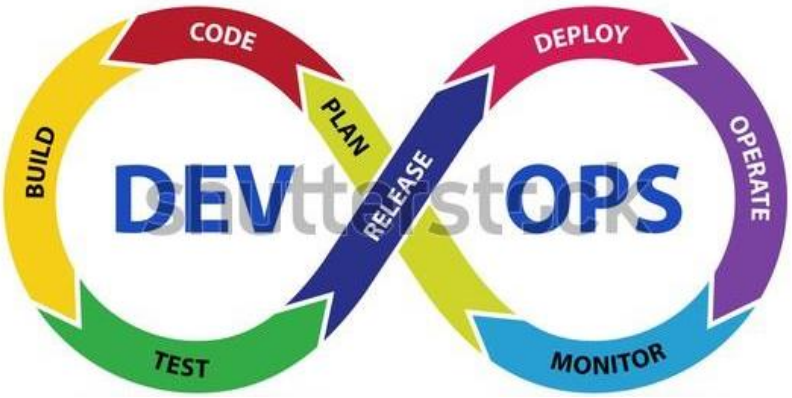
- **Migrar dados (*Migrate Data*):** Para trocar uma apresentação, é necessário trocar os dados temporariamente. Kent Beck (2010), relata que você precisa mudar a representação internamente e só então muda a interface visível externamente, essa versão é conhecida como interna-para-externa:
- **Extrair método (*Extract Method)*:** Kent Beck (2010) deixa evidente que para tornar um método longo e complexo em um método fácil de ler é necessário tornar uma pequena parte dele em um método separado e chamá-lo de um novo método. Extrair método é uma complexa refatoração atômica, mas, felizmente é pouco provável que esse processo seja feito manualmente, já que é comum sua implementação na refatoração automática.
- **Método em uma linha (*Inline Method*):** utilizado para simplificar fluxos e dados que podem se tornar complexos, retorcidos ou espalhados, é necessário substituir uma invocação de método pelo próprio método. (BECK, 2010)
- **Extrair interface (*Extract Interface*):** para introduzirmos uma segunda implementação de operações, devemos criar uma interface contendo todas as operações compartilhadas. Muitas vezes, quando é necessário extrair uma interface, nós saímos da primeira implementação e vamos para a segunda implementação. Um exemplo seria: quando você tem um *Rectangle* e pretende adicionar Oval, então, você cria uma interface chamada *Shape*. Muitas vezes, encontrar os nomes para as interfaces é um processo considerado fácil. (BECK, 2010)
- **Mover método *(Move Method)*:** consiste em invocar um método para a classe que ele pertence. Quando for necessário mover apenas uma parte do método, você deve primeiro extrair o método, mover o método inteiro, após colocar em uma linha o método dentro da classe original**.** (BECK, 2010)
- **Objeto método (*Method Object*):** de acordo com Beck (2010), este Objeto Método é útil na preparação para inserir um tipo de lógica nova no sistema. É muito utilizado também para simplificar um código que não pode ser submetido a Extrair Método.
- **Adicionar parâmetro (*Add Parameter)*:** quando adicionamos um parâmetro, estamos frequentemente realizando um passo de extensão. Se

aplica quando você, por exemplo, executou o primeiro teste sem necessitar de parâmetros, mas, posteriormente, você tem que levar em consideração mais informações, assim adicionamos um parâmetro. (BECK, 2010)

- **Parâmetro de método para parâmetro de construtor (*Method Parameter to Constructor Parameter*):** quando passamos o mesmo parâmetro a diversos métodos distintos dentro do mesmo objeto, podemos simplificar a API passando o parâmetro uma única vez, assim, eliminamos a duplicação. (BECK, 2010)

4 INTEGRAÇÃO CONTÍNUA

FIGURA – Processos da Integração contínua



Fonte: 1642498102

De acordo com Fowler (2006), a integração contínua tem como principal característica a integração constante de todas as modificações realizadas no software pela equipe de desenvolvedores. O processo de integração necessita de uma *build* automatizada, a fim de realizar todos os testes necessários para, assim, identificar os erros de integração com mais agilidade. Podemos dizer que ela está diretamente relacionada com a qualidade do *software*, levando em conta que a *build* precisa ser aprovada em todos os testes especificados pela equipe.

Como apresentado na Figura em abertura do tópico, a equipe de desenvolvimento realiza o planejamento de todos os requisitos necessários (*Plan*), em seguida, os desenvolvedores desenvolvem/codificam o que foi planejado inicialmente (*Code*), para na, sequência, construir uma versão (*Build*) que vai ser testada (*Test*). Caso seja aprovada nos testes, essa nova versão do sistema vai ser lançada (*Release*), sendo colocada no ambiente de produção para que o cliente possa utilizar (*Deploy*), com a utilização do sistema (*Operate*), o cliente poderá fornecer um *feedback* para a equipe que vai estar monitorando todas as funcionalidades do sistema (*Monitor*). Enquanto o sistema estiver em operação, esses processos nunca se encerram. (FOWLER, 2006)

Princípios a serem seguidos

Fowler (2006) apresentou alguns princípios que devem ser seguidos, sendo princípios básicos para auxiliar a implantação, esses princípios são apresentados a seguir.

- **A equipe deve manter um único repositório de código:** Todos os artefatos que são indispensáveis para realizar uma *build* do *software*, necessitam estar nesse repositório, em um local em que todos os desenvolvedores tenham conhecimento. (FOWLER, 2006)
- **Automatizar a *build*:** a construção dos artefatos de um sistema completo e executável deve ser realizada de forma automática, em um ambiente diferente das IDEs. (FOWLER, 2006)
- **Fazer com que a *build* seja auto testável:** deve-se deixar os testes automatizados, assim o código vai ser verificado, com o intuito de encontrar problemas que venham a surgir. (FOWLER, 2006)
- **Cada desenvolvedor deve lançar suas modificações todos os dias:** essas modificações necessitam manter o código perfeitamente executável, com essa prática, a equipe de desenvolvedores consegue encontrar com maior agilidade os conflitos que podem surgir entre versões. (FOWLER, 2006)
- **Cada *commit* (conjunto de modificações) deve atualizar o repositório principal em uma máquina de integração:** O desenvolvedor vai considerar finalizada uma alteração no repositório quando a build for realizada com sucesso após o lançamento de todas as modificações. (FOWLER, 2006)
- **Manter a *build* rápida:** O tempo de execução da *build* precisa ser curto, assim, conseguimos manter o *feedback* ágil. (FOWLER, 2006)
- **Testar em uma cópia do ambiente de produção:** A configuração do ambiente de testes precisa ser configurada exatamente igual ao do ambiente de produção. (FOWLER, 2006)
- **Todos podem ver o que está acontecendo:** é indispensável a comunicação entre a equipe de desenvolvimento em relação ao estado atual do *software*, pode ser utilizado, por exemplo, alertas luminosos ou sonoros para sinalizar o resultado ou o estado das últimas *builds*. (FOWLER, 2006)
- **Automatize a implantação do sistema:** a implantação deve rodar naturalmente dentro de qualquer ambiente, assim as implantações automáticas

fazem com que esse processo se torne bem mais rápido, reduzindo ainda a possibilidade de erros. (FOWLER, 2006)

Algumas ferramentas para Integração contínua

Existem diversas ferramentas para realizar a integração contínua, dentre elas, podemos citar como exemplo as mais conhecidas, sendo: (DEVOPS, 2018)

- ***Jenkins*:** considerada uma das melhores ferramentas, ela possui o código aberto, possui diversas extensões de recursos, podendo ser acessada através de *plugins*;
- ***Bamboo*:** foi desenvolvida pela *Atlassian*, e tem disponibilidade para uma versão de nuvem (hospedada na instância do *Amazon EC2*) e em servidor;
- ***Apache Gump*:** auxilia na compilação do código, identificando as alterações que não são compatíveis com o código, isso ocorre no momento em que as alterações são incluídas no sistema de controle de versão;
- ***CircleCI1*:** essa ferramenta suporta diversas linguagens e faz uso de contêineres para oferecer os seus serviços. Ela é hospedada no *GitHub*;
- ***Buildbot*:** possui o código aberto, ele ajuda a integrar, construir e testar todos os processos, de maneira automatizada. O *Buildbot* realiza a integração com diversas ferramentas de gerenciamento de *software*.

Vantagens da Integração contínua

A integração contínua redefiniu as práticas antigas de desenvolvimento de *software*, além da implantação para o cliente, que ficou totalmente diferente. De acordo com Fowler (2006), a integração contínua proporciona diversos benefícios para a equipe de desenvolvimento, sendo eles:

- Aumenta a produtividade do desenvolvedor;
- Auxilia a identificar os erros mais rapidamente;
- Possibilita entregar atualizações com mais agilidade;
- Não possui incompatibilidade, assim a equipe não será surpreendida no final;
- Possui feedback de todas as alterações feitas em todo o sistema;

- A compilação fica automatizada;
- Os processos de integração não são longos;
- Analisa a viabilidade do código sem ser necessário esperar;
- Todos os códigos são modulares e não são complexos, isso devido aos *check ins* frequentes;
- Conseguimos levar um tempo menor de depuração, possibilitando ter mais tempo para adicionar novos recursos;
- Possui um sistema de *feedback* contínuo e máquina de integração dedicada.

5 CICLO DE VIDA DO DESENVOLVIMENTO ÁGIL DE SOFTWARE

Figura- Ciclo de vida do desenvolvimento ágil

Fonte: 709541407

Quando vamos desenvolver um *software*, a primeira escolha a ser feita é o modelo de ciclo de vida que vai ser utilizado. O ciclo de vida consiste em uma estrutura formada por processos, atividades e tarefas a ser desenvolvida, operação e manutenção do *software*, envolvendo todo o processo que o sistema vai passar, desde o levantamento de requisitos, até a entrega do produto ao cliente. Em outras palavras, os modelos de ciclo de vida de um *software* são como uma estrutura pré-definida para ser seguida pela equipe de desenvolvimento. (SOMMERVILLE, 2011)

Um processo de *software* é caracterizado por um conjunto de atividades que necessitam ser executadas para que resulte em um *software* de qualidade. Cada atividade pode ser agrupada em fases, como o levantamento de requisitos, a análise, o projeto, o desenvolvimento, os testes e a implantação do sistema. Por sua vez, em cada uma das fases são definidas as atividades, funções e responsabilidades de cada membro da equipe de desenvolvimento. (PRESSMAN, 2010)

Posso afirmar, caro (a) aluno (a), que não existe um modelo de ciclo de vida ideal, mas sim o que se encaixa no perfil do negócio do cliente, no tempo que se tem disponível para desenvolvimento, no custo e na equipe disponível para o projeto. São fatores primordiais para a escolha do modelo a ser adotado. (SOMMERVILLE, 2011)

Quando utilizamos um ciclo de vida de *software* ágeis, o início e final de cada fase significa um ponto de revisão do trabalho feito e do que será produzido na

sequência. Dessa forma, fica bem mais simples e eficaz identificar e corrigir erros que possam ter surgido no decorrer do processo. Os ciclos de vida ágeis se enquadram no modelo iterativo e no incremental, o qual define as fases as quais serão incrementadas o *software*, e no ciclo de vida adaptativo e no ágil, em que o produto é desenvolvido após múltiplas iterações. (NORÉN, 2021)

Como apresentado na Figura (Ciclo de vida do Desenvolvimento Ágil), a equipe de desenvolvimento realiza o planejamento de todos os requisitos necessários (*Plan*) dentro da *Sprint 1* e, em seguida, os desenvolvedores codificam o que foi planejado inicialmente (*Develop*), para na sequência realizar os testes necessários (*Test*). Caso seja aprovada nos testes, essa nova versão do sistema vai ser lançada no ambiente de produção para que o cliente possa utilizar (*Deploy*), e, após ser revisado todo o processo (*Review*) e ele for satisfatório, podemos ir para a próxima tarefa (*Sprint 2)*. (AUDY, 2015)

Segundo Nóren (2021), atualmente podemos utilizar dois modelos de ciclo de vida ágil, sendo eles:

- Centrado no fluxo, como, por exemplo, o Kanban;
- Centrados em ciclos iterativos e incrementais, como, por exemplo, o Scrum.

Centrado no fluxo, são estabelecidas algumas limitações de forma muito óbvias para a aprovação das atividades, conhecida como Work in Progress. No centrado em ciclos iterativos e incrementais as iterações são bem mais rápidas, podendo variar de uma a quatro semanas entre cada sprints. (NORÉN, 2021)

6 FERRAMENTAS DE APOIO: *USER STORY*, *PLANNING POKER* E *BURNDOWN*

- ***User Story***

User Story, consiste basicamente em compreender de maneira clara e direta todas as necessidades do cliente, levando em consideração cada detalhe por ele apresentado. Quando vamos desenvolver um projeto, necessitamos que exista um alinhamento entre o que está sendo desenvolvido pela equipe e o que o cliente espera da aplicação. Por esse motivo, todos os requisitos obtidos com o cliente são considerados de extrema importância. (ROZENFELD, 2006).

Segundo Cohn (2004a) as *User Stories* são adquiridas através de conversas com os usuários, para assim serem compiladas como breves descrições a partir da perspectiva do usuário. Essas descrições, por sua vez, representam as funcionalidades que o sistema deve ter, (REES, 2002), essas funcionalidades englobam muito mais do que as características técnicas, podemos dizer que abrangem fatores que retornam muito valor ao cliente. (OGLIO, 2006).

Características da *User Story*

User Story possui cinco características básicas, sendo elas: (LEFFINGWELL; BEHRENS, 2010)

- **Dar valor:** (atributo mais importante), mostra como uma determinada funcionalidade vai atribuir muito valor ao usuário, este é o atributo considerado mais importante;
- **Independente:** testada e desenvolvida por si só;
- **Negociável:** de acordo com os feedbacks dos usuários pode haver colaboração e alteração nos requisitos;
- **Estimável:** entendimento da dificuldade de implementação;
- **Testável:** é validada por testes, a fim de garantir que os requisitos foram devidamente implementados.

Vantagens da *User Story*

A utilização da *User Story* possibilita uma linguagem comum entre os usuários e a equipe de desenvolvimento, trabalhando juntos, a fim de identificar o que realmente é relevante e requerido pelo *software*. (LEFFINGWELL; BEHRENS, 2010).

Conseguimos ter *feedbacks* constantes, pois trata-se de um método que pode ser utilizado por meio de iterações. (ALEXANDER; MAIDEN, 2004). Como os requisitos sofrem alterações constantes, o fato de trabalhar com iterações facilita muito a vida da equipe de desenvolvimento. (OGLIO, 2006).

Com essa prática, torna-se possível uma compreensão maior do *design* em termos mais práticos, auxiliando, assim, o processo de desenvolvimento, o qual facilita compreender o que acontece agora, o que os usuários gostariam que acontecesse, além dos problemas associados, possibilitando, assim, refinar uma funcionalidade do sistema. (ALEXANDER; MAIDEN, 2004).

Modelo genérico de *User Stories*

Cohn (2004b) sugere um plano para a aplicação do *User Stories* no desenvolvimento de *software*, plano esse dividido em quatro fases principais:

- Fase dos usuários;
- Fase das histórias;
- Fases das estimativas;
- Fase do plano de lançamento.

Fase dos usuários

Na fase denominada Fase dos Usuários, de acordo com Cohn (2004b), devemos seguir as seguintes etapas:

- **Identificação dos usuários e de suas funções em cartões:** o time precisa identificar todos os integrantes da equipe, sendo eles internos ou externos, isso pode acontecer por meio de uma sessão de *brainstorming* (tempestade de ideias, caracterizado por um debate), coletando os possíveis usuários em *post-its*;

- **Validação dessas funções e eliminação de duplicatas:** após a sessão de *brainstorming*, a equipe precisa analisar a lista de nomes coletados, a fim de manter nela somente os usuários estritamente necessários;
- **Avaliação de cada função adicionando novas informações:** momento em que cada usuário precisa ser considerado individualmente, a fim de coletar novas informações sobre a frequência de uso, do domínio de *software*, do objetivo de uso e entre outros dados;
- **Criação de algumas personas:** para ajudar na criação de histórias, pode utilizar personas. Essa etapa é opcional;

Após seguir todos esses passos da fase dos usuários, irá resultar em uma lista de usuários claramente identificados, com suas funções bem definidas e algumas possíveis personas. (COHN, 2004b)

Fase das histórias

Na fase denominada Fase das Histórias, de acordo com Cohn (2004b), devemos seguir as seguintes etapas:

- **Realização de um *workshop* para elaboração das histórias**: consiste em reunir a equipe em um *workshop* com o propósito de criar histórias para cada usuário e persona, identificados na fase do usuário;
- **Criação das histórias para cada persona**: inicia-se pelas personas identificadas, a equipe começa a elaborar uma história por vez.
- **Criação das histórias para os outros usuários:** repetimos os procedimentos da etapa anterior para todos os usuários identificados;
- **Verificar se as histórias estão adequadas:** é de extrema importância que seja revisado todas as histórias criadas, pois assim, os desenvolvedores vão conseguir segui-las da maneira esperada.

Após seguir todos esses passos da fase das histórias, irá resultar em uma lista de histórias para todos os usuários da fase anterior. (COHN, 2004b)

Fase das estimativas

Na fase denominada Fase das Estimativas, de acordo com Cohn (2004b), devemos seguir as seguintes etapas:

- **Discussão sobre as histórias com os desenvolvedores:** momento em que as histórias podem ser reavaliadas ou desmembradas;
- **Cálculo da estimativa de tempo de cada história:** momento em que a equipe deve estimar o tempo que vai ser utilizado em cada história, o tempo pode ser estimado em horas, dias, ou como acharem conveniente;

Após seguir todos esses passos da Fase das Estimativas, irá resultar em uma lista de histórias com suas devidas estimativas de tempo. (COHN, 2004b)

Fase do Plano de Lançamento

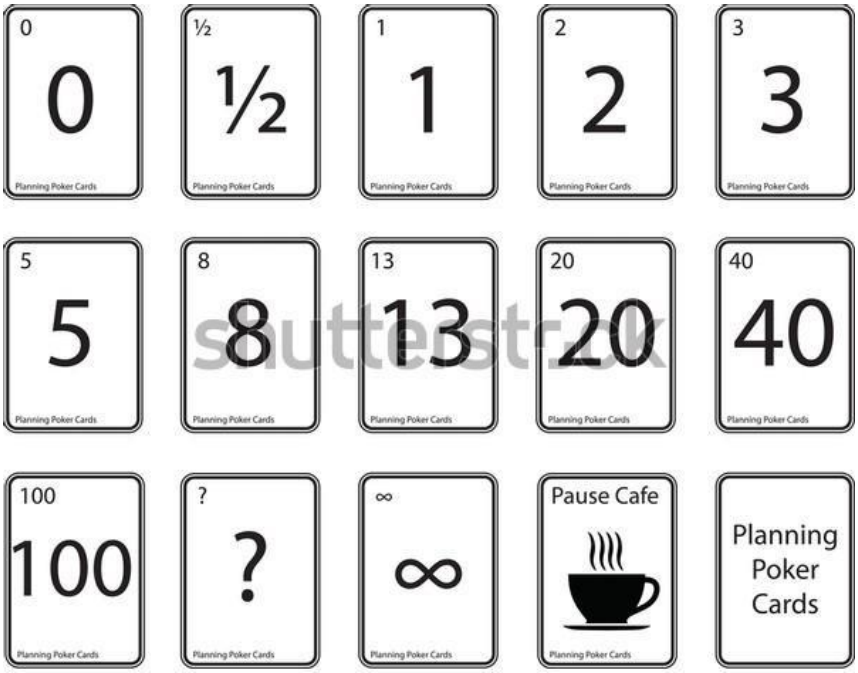
Na fase denominada Fase do Plano de Lançamento, de acordo com Cohn (2004b), devemos seguir as seguintes etapas:

- **Seleção do tamanho da iteração:** neste momento, precisamos definir o tamanho de cada iteração. O tamanho vai depender do tipo de projeto, do prazo para a entrega final e de outras entregas pequenas;
- **Cálculo da estimativa da velocidade:** neste momento, podemos estimar a velocidade do projeto fazendo uso de projeções históricas, e isso só é possível quando o time de projeto é o mesmo e quando o projeto é parecido. Nos demais casos, o time provavelmente vai fazer a estimativa com base na sua experiência.
- **Priorização das histórias**: podemos utilizar diversos métodos diferentes, porém, o mais comum é a categorização das histórias por relevância.

Após seguir todos esses passos da Fase do Plano de Lançamento, irá resultar em histórias ordenadas em cada iteração.

- ***Planning Poker***

Figura 01 - Cartas *Planning Poker*



Fonte: 1923982988

A técnica *Planning Poker* originou-se no *Extreme Programming* (XP), é utilizada para que o time consiga estimar de maneira mais ágil o que será realizado em breve, podendo discutir e compreender melhor cada funcionalidade a ser desenvolvida. Quem criou esta técnica foi o James W. Grenning, um dos integrantes do manifesto ágil, em abril de 2002. Ela foi incorporada nos treinamentos de *Scrum* e acabou se tornando extremamente popular. (COHN, 2005)

Essa técnica é aplicada no início de cada iteração (*Sprint*), durante a reunião de planejamento ou em reuniões de refinamento de *backlog*, ou ainda, para estimar funcionalidades de um modo mais macro. O integrante responsável pelo negócio (cliente ou *Product Owner*) faz a leitura e explica a história do usuário, para que a equipe de desenvolvimento possa conversar sobre a história e estimar com uma carta do *Planning Poker*. (COHN, 2005)

Benefícios do *Planning Poker*

- É uma abordagem de estimativa mais rápida que abordagens tradicionais, mantendo um bom resultado. (COHN, 2005);
- Faz com que não tenha discussões em "*loop* infinito" (analysis paralysis). (COHN, 2005);
- Exibe os requisitos que ainda não estão prontos para desenvolvimento (*Definition of Ready*). (COHN, 2005);
- Possibilita que todos da equipe façam as estimativas. (COHN, 2005);

- Possibilita engajamento e senso de pertencimento à equipe, pois todos integrantes discutem e trazem as suas próprias estimativas. (COHN, 2005)

Como estimar

É importante ressaltar que, estimativas são só estimativas, pode ser que o que você venha a estimar seja exatamente o tempo que você vai levar em um projeto, mas acostume-se, as estimativas nem sempre estarão corretas, pelo simples fato de não ser possível prever o futuro. Pode ser que algumas estimativas vão levar um tempo menor para serem executadas, outras um tempo maior, contudo, na média costumam funcionar relativamente bem. (COHN, 2005)

Como Jogar o *Planning Poker*

Antes de iniciar, é necessário definir qual será a unidade de medida, se vai ser em horas ou em pontos (*Story Points*). *Story Point* é uma estimativa relativa de complexidade, risco e esforço, sendo assim, para que você consiga estimar com mais precisão, deve levar em consideração essas três variáveis. (COHN, 2005)

Regras

- Somente as pessoas que forem participar da construção podem jogar; (COHN, 2005);
- O *Scrum Master* deve ser o facilitador do jogo; (COHN, 2005)
- O *Product Owner* deve estar disponível para tirar dúvidas; (COHN, 2005)
- Todos da equipe devem ter os baralhos de *Planning Poker,* tal qual é representado na Figura 01. (COHN, 2005)
- *Product Owner* deve apresentar os itens priorizados a equipe. (COHN, 2005)

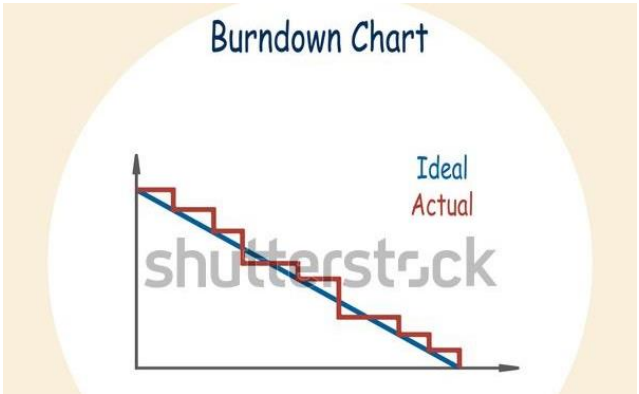
Regras adicionais

- Quando uma só pessoa da equipe estiver com uma pontuação diferente das demais, devemos fazer a média; (COHN, 2005)
- Quando jogamos pela terceira vez o mesmo item e não conseguimos atingir um consenso, devemos adotar o que mais de 70% do time acredita; (COHN, 2005)

- Caso não exista uma maioria de 70%, deve-se quebrar o item, isso ajudará em novas rodadas de *planning poker*; (COHN, 2005)
- As estimativas devem seguir exclusivamente a sequência de Fibonacci (0, ½, 1, 2, 3, 5, 8, 13, …); (COHN, 2005)
- O cliente/*product owner* apresenta a história de usuário; (COHN, 2005)
- A equipe de desenvolvimento discute as histórias de usuário com o intuito de entender os requisitos; (COHN, 2005)
- Cada integrante da equipe decide uma estimativa, colocando a carta virada para baixo, escondendo a sua estimativa. (COHN, 2005)
- Quando todos os integrantes da equipe colocarem suas estimativas na mesa, viram-se as cartas ao mesmo tempo, apresentando, assim, as estimativas. Sendo essa a origem do nome "*poker*"; (COHN, 2005)
- Quando as cartas possuem os mesmos valores, a estimativa está pronta. Nesse momento, podemos ir para a próxima história de usuário, quando existir. Se não houver, a sessão de *Planning Poker* é finalizada; (COHN, 2005)
- Quando as estimativas não são iguais, existindo uma discrepância significativa, todos que estimaram os extremos devem discutir os seus pontos de vista. Após finalizar a discussão, o processo de *poker* se repete, com cada integrante do time jogando novamente a carta de estimativa. (COHN, 2005)

- ***Burndown***

Figura 02 – Gráfico de Burndown



Fonte: 1039492567

Utilizamos o gráfico *Burndown* com o intuito de acompanhar a evolução de uma *Sprint*, além de servir como um indicador, a fim de prever quanto tempo ainda resta para que o trabalho seja concluído, e isso é possível através da linha atual

(representada na Figura 02 pela cor vermelha), com ela, verificamos se o processo está prestes a terminar, dentro da meta estimada pela equipe. (RUBIN, 2013)

De acordo com a Figura 02, o eixo vertical retrata a estimativa de trabalho (esforço/horas) restante, e o eixo horizontal representa os dias de uma *Sprint*. A linha representada pela cor azul significa o tempo ideal estimado pela equipe para a conclusão da *Sprint*, já a linha na cor vermelha vai indicar o progresso do time dentro da *Sprint*. A cada dia o gráfico é atualizado, disponibilizando, assim, o total estimado, para que possa ser acompanhado o desempenho da equipe na *Sprint* atual. Com os resultados obtidos com o gráfico, podemos tomar medidas decisivas para a conclusão da *Sprint*. (RUBIN, 2013).

O gráfico de *Burndown* se tornou indispensável no *Scrum*, ele é utilizado para uma melhor compreensão do quadro de tarefas, além de ser um grande indicador de tendência (conhecido como previsão e trabalho), com o intuito de atingir os objetivos estimados pela equipe. (AUDY, 2015).

Preenchimento do gráfico Burndown

- A cada dia, marcar o ponto no gráfico equivalente ao momento atual (Decaimento representa o quanto de trabalho previsto que já foi feito);
- Conectar o ponto atual com o ponto do dia anterior com uma linha;
- Analisar a situação atual do desenvolvimento e tomar as medidas necessárias.

Análise do gráfico *Burndown*

- O que significa quando o gráfico está mais alto que a linha de decaimento linear? Irá gerar atraso no cronograma;
- O que significa quando o gráfico está mais baixo que a linha de decaimento linear? Irá gerar adiantamento no cronograma.

O que devemos fazer em cada um desses casos?

- Mitigar riscos;

- Refazer as estimativas;
- Analisar a possibilidade de adicionar ou remover tarefas.

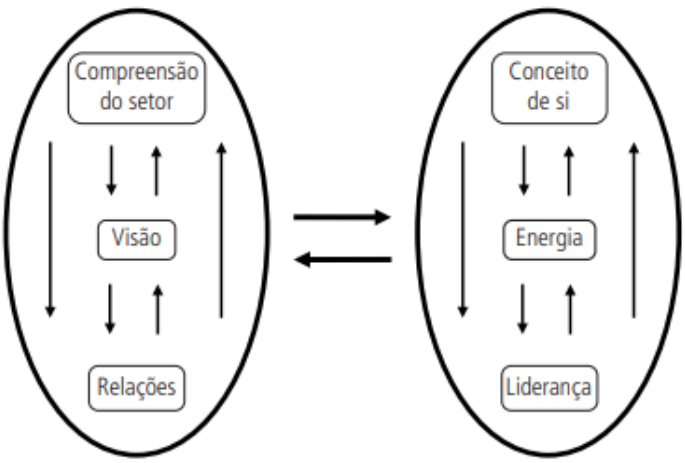
7 O PROCESSO VISIONÁRIO

De acordo com Filion (1991), o conceito de si ou a chamada auto imagem é a principal fonte de criação. O autor afirma que, um indivíduo só realiza algo quando se julga capaz de fazê-lo. Isso está diretamente ligado sobre como a pessoa se enxerga, é a imagem que ela tem de si mesma, contudo, podemos dizer, então, que a auto-imagem tem forte influência no desempenho de cada pessoa.

Acredita-se que o conceito de si, inspira e condiciona o processo visionário, sendo assim, o empreendedor deve se conhecer verdadeiramente, pois suas características influenciam a empresa. Se um indivíduo é desorganizado, por exemplo, a empresa tende a incorporar essa característica. Contudo, acredita-se também que a energia é uma forte aliada do processo visionário, tendo em vista que a energia está relacionada a quanto uma pessoa está determinada a investir em um uma situação. (CARMO, 2011)

Como apresentado na Figura 03, a energia é considerada a fonte almejada por um empreendedor, pois ela oferece o ânimo indispensável para que possa entender um determinado setor, desenvolver uma visão, estabelecer as relações, aprofundar-se nas características do produto ou serviço e dedicar-se à organização e ao controle gerencial. Assim sendo, a energia torna-se um dos elementos fundamentais na formação das condições para o exercício da liderança. (CARMO, 2011)

Figura 03 – Elementos que dão suporte ao processo visionário



Fonte: Filion (1991)

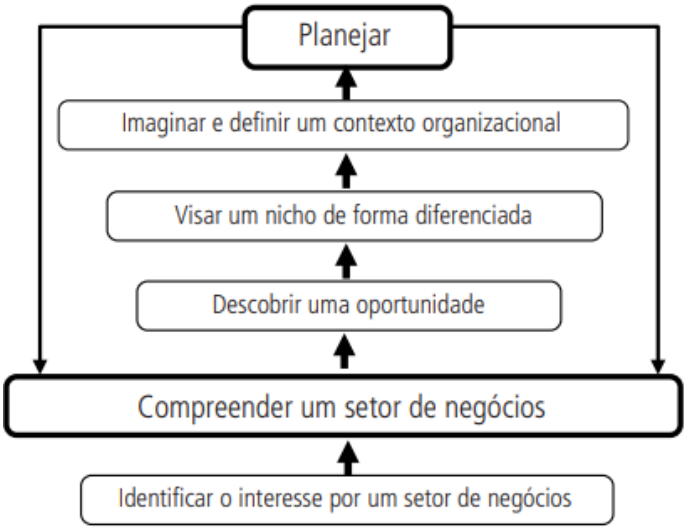
Você, caro (a) aluno (a), imagina como pode ter a visão de oportunidade real sem que você conheça a área de negócios em que se pretende atuar? Como conseguir desenvolver uma visão sem a compreensão do setor? Pois é, necessitamos pensar sobre essas questões! Quando decidimos abrir uma empresa da qual não temos conhecimento do seu setor, nós não estamos empreendendo, estamos nos aventurando. (DOLABELA, 2008)

Filion (1991) afirma que, quando compreendemos um determinado setor, passamos a entender como funcionam as empresas que atuam no ambiente em que pretendemos entrar, entendemos também quem são os clientes, como eles se comportam e quais são os pontos fortes e fracos da concorrência, dentre outras análises necessárias para que alcancemos os nossos objetivos.

Filion (1991) ressalta ainda que é indispensável também conhecer a tecnologia envolvida, as tendências de mercado, as políticas de exportações, as barreiras de entrada e a lucratividade. É necessário também conhecer os fornecedores e insumos, as necessidades de recursos humanos e as melhores maneiras de sua contratação, devemos também saber quais são as ameaças e oportunidades apresentadas, tendências tecnológicas e o funcionamento do mercado concorrencial. (CARMO, 2011)

Todos os elementos do Processo Visionário contribuem diretamente para a compreensão do setor. A Figura 04 apresenta as etapas do Processo Visionário. O mesmo se inicia a partir da identificação do interesse do indivíduo (empreendedor) por um setor. (CARMO, 2011)

Figura 04 – Etapas do Processo Visionário



FONTE: Filion (1991)

Carmo (2011), acredita que motivação para empreender surge inicialmente dentro do círculo de relações primárias, em outras palavras, surgem a partir das relações familiares. Quando um indivíduo inicia o seu processo visionário, ele busca por relações que possam lhe agregar contribuindo para o aprimoramento da sua visão. Dessa forma, todo empreendedor passa a enxergar as suas relações como um auxílio para que possam melhorar, desenvolver e implementar sua visão

SAIBA MAIS

O gráfico de *Burndown* tornou-se uma ferramenta muito importante dentro da metodologia ágil *Scrum*, mostrando visualmente quanto trabalho ainda precisa ser feito para concluir uma sprint. A equipe de desenvolvimento indica no gráfico o início e o final de uma Sprint, isso levando em consideração a tarefa que precisa ser realizada nesta Sprint. (AUDY, 2015)

REFLITA

Antes de definir quais ferramentas você vai utilizar durante o seu desenvolvimento é importante entender em que elas poderão auxiliar a equipe, tendo em vista que cada particularidade do projeto. (NAVARRO, 2021)

CONSIDERAÇÕES FINAIS

Podemos observar que existem inúmeros métodos que podem ser adotados em um projeto, todos eles proporcionam de alguma forma, facilidades durante o processo de desenvolvimento. Quando tratamos dos ciclos de vida de desenvolvimento ágeis identificamos que eles se enquadram no modelo iterativo, incremental e no modelo adaptativo. (NORÉN, 2021)

O desenvolvimento dirigido por testes, citado por Freeman (2012), é um método muito utilizado pelas equipes, pois com ele podemos identificar e solucionar os erros com muito mais agilidade e praticidade, pois cada funcionalidade é testada assim que é desenvolvida. Beck (2000), apresentou a programação em par, que por sua vez, previne que erros de códigos possam vir a acontecer, sendo que todo o código que é digitado por um membro da dupla é corrigido simultaneamente pelo outro.

De acordo com Beck (2010), a Refatoração é muito utilizada para que seja possível manter um *software* bem projetado, mesmo com o passar do tempo, levando em conta todas as mudanças que ele irá sofrer. A integração contínua abordada tem como objetivo identificar os erros mais rapidamente, podendo melhorar assim a qualidade do *software* e reduzir o tempo gasto para validar e disponibilizar as atualizações. (FOWLER, 2006)

Cohn (2004b) abordou a ferramenta *User Story* e a *Planning Poker*, enquanto Rubin (2013) retratou o gráfico *Burndown*. O que cada ferramenta dessa tem em comum é que elas auxiliam o processo de desenvolvimento de *software*, garantindo assim uma maior confiabilidade de que o processo de desenvolvimento cumprirá todos os requisitos inicialmente planejados. O *Planning Poker* evita discussões prolongadas, agilizando os processos, o gráfico de *Burndown* possibilita o acompanhamento do desenvolvimento do projeto e o *User Story* garante que o *software* atenderá todas as necessidades do cliente.

Por fim, conclui-se que o conceito de si, inspira e condiciona o Processo Visionário, dessa forma, o empreendedor deve se autoconhecer, pois suas características influenciam diretamente na empresa. (FILION, 1991)

REFERÊNCIAS

SOMMERVILLE. **Engenharia de Software**, São Paulo: Addison-Wesley, 9 ed., 2011. ISBN-10: 8579361087 ISBN-13: 9788579361081.

BECK, K. ***Test Driven Development: By Example. Boston***: Addison-Wesley, 2002. JEFFRIES, R.; MELNIK, G. TDD: The Art of Fearless Programming. IEEE Software, v. 24, 2007, p. 24-30.

MASSOL, V.; HUSTED, T. JUnit in Action. Greenwich, Conn.: Manning Publications Co., 2003.

Pressman R. **Engenharia de Software**, 6a edição, AMGH Editora. 2010.

FREEMAN, Steve. Pryce, Nat. **Desenvolvimento de Software Orientado a objetos, Guiado por Testes**. Rio de Janeiro: Alta Books, 2012.

BECK, K.; ANDRES, C. ***Extreme Programming Explained***: Embrace chance. Boston: Addison-Wesley, 2004.

TELES, V. M. ***Extreme Programming:*** aprenda como encantar seus usuários desenvolvendo software com agilidade e alta qualidade. São Paulo: Novatec, 2004.

SHORE, J.; WARDEN, S. **A Arte do Desenvolvimento Ágil.** Tradução: Bianca Capitânio. Rio de Janeiro: Alta Books, 2008.

POPPENDIECK, M.; POPPENDIECK, T. **Implementando o Desenvolvimento Lean de *Software***: Do Conceito ao Dinheiro. 1. ed. Porto Alegre: Bookman, 2011.

FOWLER, M. ***Continuous integration***. 2006. Disponível em: https://martinfowler.

com/articles/continuousIntegration.html. Acesso em: 23 set. 2016.

FILION, Louis Jacques. **O planejamento de seu sistema de aprendizagem empresarial: identifique uma visão e avalie o seu sistema de relações**. Revista de Administração de Empresas, São Paulo, v. 31, n. 3. p. 63-71, jul./set. 1991.

Empreendedorismo ISBN: Cintia Tavares do Carmo - **Curso Técnico em Informática** - Colatina – ES – 2011 – INSTITUTO FEDERAL ESPIRITO SANTO

RUBIN, Kenneth S. ***Essencial Scrum: A Practical Guide To The Most Popular Agile Process***. Arbor : Pearson , 2013.

AUDY, Jorge. ***Scrum 360***: **Um guia completo e prático de agilidade.** São Paulo : Casa do Código, 2015.

Mundo DevOps: **Desenvolvimento de *Software* Eireli** - 30.328.325/0001-69 © Copyright – 2018.

REES, M. J. A ***Feasible User Story Tool for Agile Software Development*** ? p. 0–8, 2002.

COHN, M. ***Advantages of User Stories for Requirements Why User Stories? User Stories Aren ' t Use Cases***. 2004a

Mike Cohn - Editora : Prentice Hall PTR; Illustrated edição (1 novembro 2005) ***Agile Estimating and Planning*** Capa comum – Ilustrado, 1 novembro, 2005.

OGLIO, P. D. UNIVERSIDADE DO VALE DO RIO DOS SINOS ***Uma Ferramenta para Gerenciamento de Requisitos em Projetos Baseados em Extreme Programming*** Resumo. UNIVERSIDADE DO VALE DO RIO DOS SINOS, 2006.

COHN, M. ***User Stories Applied: For Agile Software Development.*** Addison Wesley, 2004b.

ALEXANDER, I. A. N.; MAIDEN, N. ***Scenarios, stories, use cases.*** John Wiley &Sons Ltd. 2004. England.

LEFFINGWELL, B. D.; BEHRENS, P. By ***Dean Leffingwell with Pete Behrens***. p. 1–16, 2010.

NÓREN. A. **Como é o ciclo de vida de um projeto ágil.** 2021. Disponível em : https://blog.mastertech.com.br/negocios/como-e-o-ciclo-de-vida-de-um-projeto-agil/ . Acesso em:

UNIDADE III

EXEMPLOS DE MÉTODOS ÁGEIS

Professora Especialista Camila Sanches Navarro

Plano de Estudo:

- Programação Extrema XP;
- Kanban.

Objetivos de Aprendizagem:

- Compreender todos os conceitos da Programação Extrema (XP);
- Entender como funciona o método Kanban, identificando como implementá-lo dentro de uma empresa.

INTRODUÇÃO

Nos dias atuais, para se entregar um projeto, é necessário ficar atento a quantia de riscos, os quais podem vir a ocorrer, dentre estes riscos, podemos citar o atraso na entrega, muitas vezes entregar um sistema que não atende a solicitação do cliente, devido a negócios mal compreendidos e entre outros diversos problemas que podemos nos deparar. (BECK, 2004)

Com a metodologia XP podemos praticamente extinguir a possibilidade de nos depararmos com problemas que comprometem o nosso projeto, como, por exemplo, as taxas de erros, no desenvolvimento de sistema utilizando a XP realizamos testes constantemente, abrangendo tanto a equipe de desenvolvedores quanto os clientes, estes testes verificam a funcionalidade de cada módulo do *software*. Em geral, a XP traz consigo quatro valores, sendo eles a comunicação, a simplicidade, o *feedback* e a coragem. (BECK, 2004)

Podemos caracterizar o *Kanban* sendo como um “método sem metodologia”, podendo ser implementado em diversas áreas. É de extrema importância compreender o sistema (método) *Kanban*, pois, quando é aplicado em uma empresa as suas práticas e princípios devem ser levados em consideração a maneira atual de trabalhar da organização, pois na maioria dos casos já possui um fluxo de trabalho, de rotinas. (KANBAN UNIVERSITY, 2021)

1 PROGRAMAÇÃO EXTREMA

Neste capítulo, caro (a) aluno (a) vamos abordar a programação extrema, muito conhecida como XP, esclarecendo o ciclo de vida de desenvolvimento de software utilizando essa metodologia e as suas 12 práticas.

Figura 01 – Programação Extrema

Fonte: 1607934496

Conceitos da programação extrema (XP)

De acordo com Beck (2004), existem diversos fatores que podem acarretar em sérios problemas para a entrega de um *software*, ocasionando na insatisfação do cliente. Dentre diversos riscos, podemos citar: a taxa de erros existentes em um projeto, levantamento de requisitos incompletos e mal compreendidos, atrasos na entrega de versões, alterações nos negócios, dentre outros riscos existentes. Beck (2004), garante que, muitos desses riscos possam ser evitados com a utilização da metodologia ágil XP, citando como exemplo a taxa de erros, tendo em vista que, no XP, ela é praticamente extinta, pois são realizados testes, tanto para os programadores quanto para os usuários testarem as funcionalidades requisitadas inicialmente.

Contudo, a adoção pela metodologia ágil XP aumentou drasticamente desde o seu surgimento, e isso ocorreu devido a inúmeros depoimentos de empresas que obtiveram excelentes resultados com o uso do XP. A junção dos seus princípios com os seus valores e suas práticas resultam em uma equipe trabalhando integralmente focada no projeto, para que existam entregas constantes de funcionalidades do sistema para o cliente.

Ciclo de vida de um projeto XP

Beck (2004) explica que um projeto desenvolvido com o XP passa por determinadas etapas durante seu ciclo de vida, sendo necessário executar diversas tarefas. O ciclo de vida de um projeto XP é composto por: Fase da Exploração, a Fase de Planejamento Inicial, a Fase das Iterações do *Release*, a Fase da Produção, a Fase de Manutenção e a Fase da Morte, todas explicadas a seguir.

- **Fase de Exploração:** consiste em realizar todos os levantamentos de requisitos necessários, devemos identificar nesta fase todos os detalhes, incluindo também o ambiente tecnológico cujo sistema vai rodar ao ser entregue. Recomenda-se não utilizar mais que duas semanas na fase de exploração. Esse momento é indispensável para que obtenhamos a confiança e segurança necessária para desenvolver um produto que venha a atender todas as necessidades do usuário. (BECK, 2004)
- **Fase de Planejamento Inicial:** consiste em definir juntamente com o cliente, por exemplo, a data do lançamento do primeiro *release* (*release* é o lançamento de uma - popularmente conhecida -, versão do sistema, com uma nova funcionalidade para o cliente. (SOMMERVILLE, 2011)). Para que ocorra o planejamento, a equipe, incluindo o cliente, deve definir as estórias (pequena narração da necessidade do cliente, de forma bem simples) que vão ser desenvolvidas, classificando as suas prioridades. Dentro do planejamento é feito a análise de quantas estórias vão ser implementadas em uma iteração. Recomenda-se concluir uma iteração dentro do prazo de uma a três semanas. (BECK, 2004)
- **Fase das Iterações do *Release*:** momento em que os programadores vão implementando o sistema de acordo com o planejamento realizado

anteriormente, seguindo todos os valores, todos os princípios e todas as práticas do XP para que possa ser lançado o primeiro release. O intervalo recomendado para cada release é de dois até quatro meses. (BECK, 2004)

- **Fase da Produção:** momento em que se desenvolve todas as funcionalidades do sistema, lançando o *release* conforme vão sendo finalizados, vale ressaltar que, nesse processo é realizado diversos testes, para assim garantir que o módulo do sistema está apto para ser entregue ao cliente. (BECK, 2004)
- **Fase de Manutenção:** consiste em realizar as manutenções necessárias, muitas vezes introduzindo novas funcionalidades. É de extrema importância evidenciar que, quando realizamos uma manutenção em um *software* que já está em produção (o cliente está utilizando), devemos ter muita atenção para não danificar uma funcionalidade do sistema, atentando-se que qualquer falha neste procedimento acarretará em problemas para o usuário. (BECK, 2004)
- **Fase da Morte:** consiste em finalizar o projeto, isso pode acontecer ao momento em que o cliente se encontra completamente satisfeito com o *software*, ou na pior das hipóteses, quando o projeto se torna inviável economicamente, normalmente isso acontece quando ocorrem muitos erros durante o desenvolvimento. (BECK, 2004)

Práticas da Programação Extrema - XP

Práticas da programação extrema correspondem ao trabalho realizado pelas equipes no dia a dia. Elas necessitam ser utilizadas, a fim de agregar valores no projeto. Um exemplo seria a programação pareada, ela é utilizada, pois permite uma melhor comunicação, *feedback* constante, simplifica o *software*, identifica os erros com mais agilidade, agregando, assim, valores ao projeto. Quando você precisa adotar uma prática, antes de qualquer coisa, tem de compreender para qual projeto ela será utilizada, pois sua eficácia depende muito do contexto em que será exposta. (TELES, 2006)

Doze (12) práticas da Programação Extrema

De acordo com Kent Beck (2004), apresentaremos a seguir as 12 práticas que envolvem a Programação Extrema.

- **O Jogo do Planejamento:** Consiste em elaborar um escopo do processo completo a ser percorrido no decorrer do projeto, decidindo as prioridades, qual o caminho a ser percorrido e as datas de entrega. Esse planejamento faz com que todos os processos aconteçam da maneira mais organizada possível. (BECK, 2004)
- **Entregas Frequentes:** Beck (2004) afirma que o cliente deve estar a par do processo do sistema, recebendo frequentemente algumas versões (módulos) do *software*. Acredita-se que, inicialmente, é indispensável colocar o sistema em produção, mesmo que com poucas funcionalidades e, posteriormente, vão sendo liberadas outras versões.
- **Metáfora:** Para que os integrantes da equipe de desenvolvimento falem a mesma linguagem, todos devem compreender os requisitos do sistema. Como forma de facilitar, essa compreensão pode ser criado histórias para o *software*, auxiliando o entendimento de como este sistema será utilizado pelo usuário final. (BECK, 2004)
- **Projeto Simples:** O sistema precisa ser arquitetado (planejado) da maneira mais simples possível, levando-se em conta que geralmente o mais simples é o que realmente necessita ser feito. Caso haja alguma complexidade considerada desnecessária, ela será removida no mesmo instante que for identificada. (BECK, 2004)
- **Testes:** Após ser finalizado um módulo, tendo sido implementado todas as funcionalidades solicitadas pelo cliente, os programadores escrevem alguns testes para que o novo módulo possa ser testado pela equipe e pelo usuário. Esses testes verificam se o que realmente foi solicitado e implementado, proporcionando assim uma maior confiabilidade e segurança. (BECK, 2004)
- **Refatoração:** Após concluir um módulo, o desenvolvedor analisa a maneira que o código foi digitado, vendo se consegue, de alguma forma, simplificar como o *software* foi implementado, sem que altere as funcionalidades do sistema. Com essa análise, pode ser removido algumas duplicidades, melhorando, assim, a comunicação e consequentemente fazendo com o que sistema responda de forma mais ágil a cada requisição solicitada. (BECK, 2004)

- **Programação em Pares:** consiste no desenvolvimento de um *software* em dupla, em que um vai escrevendo os códigos (conhecido como piloto) e o outro (conhecido como navegador) vai revisando todo o código que vem sendo digitado. (BECK, 2004)
- **Propriedade Coletiva:** podemos dizer que a comunicação e padronização garante o sucesso da propriedade coletiva, pois aqui é permitido que qualquer integrante da equipe realize alterações no código quando acreditarem ser necessárias. Todos os desenvolvedores precisam compreender cada linha de código, desenvolvimento esse que necessita seguir um padrão de codificação. (BECK, 2004)
- **Integração Contínua:** consiste em integrar e atualizar as versões que vão sendo lançadas frequentemente, visando sempre a melhoria da forma como as funcionalidades vão sendo implementadas. (BECK, 2004)
- **Semana de 40 horas:** Acredita-se que a equipe de desenvolvimento não pode trabalhar mais de 40 horas semanais por duas semanas seguidas, caso isso esteja ocorrendo, serve como um indicador de que o projeto está caminhando de forma errada. (BECK, 2004)
- **Cliente Presente:** Durante o processo de desenvolvimento devemos incluir o cliente na equipe, pois será de grande serventia ao time, respondendo todas as dúvidas que possam vir a surgir durante a codificação. Isso faz com que todas as tomadas de decisões se tornem muito mais ágeis e certeiras. (BECK, 2004)
- **Padrões de Codificação:** Todos os desenvolvedores precisam escrever os códigos, seguindo os padrões estipulados pela equipe, assim todos os integrantes compreendem cada linha codificada, podendo, assim, realizar alterações quando necessário. (BECK, 2004)

Valores e Princípios de XP

Segundo Beck (2004), o XP segue quatro valores: a comunicação, a simplicidade, a coragem e o *feedback*. Esses valores direcionam os integrantes da equipe durante todo o processo de desenvolvimento.

- **Comunicação:** o primeiro valor consiste em manter a comunicação constante entre todos os envolvidos no projeto, desde os desenvolvedores até os

clientes. Para ajudar a estimular a comunicação, temos algumas práticas, como a programação em pares, os testes, a comunicação com o cliente e, ainda, com o intuito de facilitar a comunicação, um ambiente livre de paredes, divisórias, barreiras, como demonstrado na Figura 02. (BECK, 2004)

- **Simplicidade:** o segundo valor consiste em manter a simplicidade em todos os processos de desenvolvimento. Um ambiente mais simples possibilita com mais agilidade modificações quando necessárias. Assim todos da equipe devem seguir a mesma linha de simplicidade. (BECK, 2004)
- ***Feedback*:** o terceiro valor consiste em manter um *feedback* constante, isso ocorre principalmente com os testes que são executados. Esse valor é considerado muito importante, pois ele é quem direciona o desenvolvimento com mais facilidade, possibilitando a correção de erros com frequência. (BECK, 2004)
- **Coragem:** o quarto e último valor consiste em aplicar a XP como ela realmente deve ser aplicada. Quando for necessário alterar um código já escrito e que está funcionando, por exemplo, é preciso que essa tarefa seja realizada, muitas vezes deveremos jogar linhas de código fora para que possa ser reescrita de forma mais simplificada. (BECK, 2004)

Figura 02 – Modelo de ambiente de trabalho estimado pela Programação Extrema

Fonte: 572805157

Segundo Beck (2004), o XP segue cinco princípios, o *feedback* rápido, o assumir a simplicidade, a mudança incremental, o abraçando as mudanças e o trabalho de qualidade. Esses princípios direcionam os integrantes da equipe durante todo o processo de desenvolvimento.

- ***Feedback* rápido:** o primeiro princípio consiste em manter uma constante comunicação entre os integrantes envolvidos no projeto, a fim de facilitar possíveis tomadas de decisões. (BECK, 2004)
- **Assumir a simplicidade:** o segundo princípio consiste em resolver qualquer situação da forma mais simples possível. (BECK, 2004)
- **Mudança incremental:** o terceiro princípio consiste em realizar as alterações, quando necessárias, de forma incremental, sendo realizadas aos poucos, e não todas de uma só vez. (BECK, 2004)
- **Abraçando as mudanças (*Embracing Changes*):** o quarto princípio consiste em aceitar que o processo de desenvolvimento,o qual passa por diversas alterações no decorrer do projeto, e essas mudanças são sempre muito bem-vindas em qualquer período do desenvolvimento. (BECK, 2004)
- **Trabalho de qualidade:** o quinto e último princípio consiste em entregar um produto ao cliente que atende 100% dos requisitos solicitados por ele, tendo todas as suas funcionalidades rodando com perfeição, pois só assim será agregado valor ao negócio do usuário. (BECK, 2004)

Quando não se deve usar XP

Beck (2004), explica que existem projetos os quais não são indicados a utilização da metodologia XP, pois algumas características da equipe, dos clientes e da tecnologia dificultariam a execução da metodologia, tais como:

- **Cultura:** Em alguns casos, a equipe está adaptada à elaboração de muita documentação ou, até mesmo, a utilizar muito tempo para analisar e projetar todos os passos que o sistema precisa seguir à risca. Em outras palavras, pode ser que a equipe de desenvolvimento seja adepta ao desenvolvimento de sistema tradicional, dificultando a aplicação das práticas do XP. (BECK, 2004)

- **Tamanho da equipe:** Equipes com mais de 12 pessoas não são recomendadas para a adoção das práticas do XP. Imagine como ficaria a comunicação entre uma equipe com muitos integrantes, seria difícil manter a qualidade no processo de desenvolvimento do projeto. (BECK, 2004)
- **Tecnologia:** Existem casos em que a tecnologia usada por uma determinada equipe seja muito complexa para escrever os testes necessários e, muitas vezes, não possibilitam o *feedback* com frequência. Essas tecnologias raramente são adeptas a aceitarem as mudanças que venham a ser solicitadas no decorrer do projeto. (BECK, 2004)
- **Espaço físico:** o ambiente de trabalho não pode ser um espaço onde a equipe de desenvolvimento não fique próxima, pois a comunicação seria prejudicada e consequentemente isso se repetiria no projeto de forma negativa. (BECK, 2004)
- **Cliente:** se o cliente (ou algum responsável que compreenda a fundo todo o processo da empresa do cliente) não pode estar à disposição da equipe de desenvolvimento para esclarecer as dúvidas que possam vir a surgir, não se recomenda a utilização da metodologia XP. (BECK, 2004)

2 KANBAN

Figura 03 – Método Kanban

Fonte: 1389111368

O termo *Kanban* (traduzido seria “sinal” ou “cartão”) nasceu dos “sistemas de cartão” que eram utilizados em indústrias, eles serviam para facilitar a organização da série de trabalhos que tinham que ser executados. (ANDERSON, 2010)

Para facilitar a compreensão, caro (a) aluno (a), podemos dizer que com o *Kanban* você consegue gerenciar qualquer trabalho de conhecimento que venha precisar. O modelo *Kanban*, possui um sistema de sinalização, o que possibilita visualizar uma atividade de trabalho em andamento, ou em outras palavras, a quantidade de cartões (atividades) em andamento equivale à capacidade do sistema. (MARIOTTI, 2012)

Outro aspecto do modelo *Kanban* é que ele permite que uma tarefa seja “puxada” por um membro da equipe. Essas tarefas ficam na lista de *backlog* (lista de espera), e assim que um membro da equipe esteja disponível, ele puxa uma tarefa para executá-la, diferente de outros métodos, cujo as atividades são distribuídas pela equipe. Esse formato aumenta consideravelmente o desempenho da equipe de desenvolvimento. (BENSON, 2011)

Em geral, o método *Kanban* possibilita a visualização de todo o trabalho em fase de desenvolvimento, auxiliando a operar cada funcionalidade com mais

eficiência, com a utilização dele em times, conseguimos aliviar a sobrecarga e ter total controle sobre o trabalho realizado. (BENSON, 2011)

Para que possamos implantar o modelo *Kanban* devemos respeitar três estágios:

- O estágio de visualizar os processos;
- O estágio de limitar o trabalho em processo (WIP: *work in progress)*;
- O estágio de gerenciamento do *lead-time* (o tempo que uma atividade vai levar para que passe por todos os processos até a sua entrega).

Os princípios e as práticas do sistema *Kanban*

Vamos trabalhar agora caro (a) aluno (a) alguns princípios e algumas práticas do sistema (método) *Kanban*.

Princípios da gestão e mudanças

O *Kanban* faz uso de uma abordagem transformadora, para a sua implantação podemos nos basear no jeito de trabalho que a empresa já utiliza, analisando como podemos melhorá-lo. Com a implantação do quadro *Kanban*, conseguimos gerar mudanças evolutivas, pois durante o uso do quadro observamos que os atos de lideranças dos membros da equipe auxiliam o modo de trabalhar, tornando-a cada vez melhor. Sendo assim, de acordo com Benson (2011), para implantar o sistema *Kanban* em uma empresa devemos levar em conta três princípios de gestão:

- Princípio 01: Inicie a partir do que você já faz hoje;
- Princípio 02: Utilizar mudanças evolucionárias para melhorar o que já existe;
- Princípio 02: Deve encorajar os atos de liderança.

Princípios da Entrega de Serviços

O *Kanban* incentiva uma organização a utilizar uma abordagem voltada a serviços, pois acredita-se que se uma empresa pretende aumentar a qualidade do serviço prestado, ela precisa seguir rigorosamente um alguns princípios. Princípios esses citados a seguir: (KANBAN UNIVERSITY, 2021)

- Princípio 01: É necessário compreender e se atentar nas necessidades e nas expectativas do cliente;
- Princípio 02: É indispensável gerenciar todo o trabalho, fazendo com que os integrantes da equipe se auto organizem em torno dele;
- Princípio 03: Sempre rever com frequência a rede de serviços e as suas normas, com o intuito de melhorar os resultados.

Práticas Gerais do Kanban

Vamos trabalhar agora, seis práticas gerais do método Kanban.

- **Visualizar**

Acredita-se que, para identificarmos onde podemos realizar melhorias dentro de uma empresa, a chave seja uma boa visualização. Na maioria das vezes a forma como executa um determinado trabalho fica escondida, e quando exibimos todos os processos conseguimos tomar decisões que realmente vão agregar valor ao negócio. (KANBAN UNIVERSITY, 2021)

- **Limite de Trabalho em Progresso (WIP - *Work in Progress*)**

No método *Kanban* utilizamos o WIP (trabalho em progresso ou a escrita em inglês, vista no subtópico), ele mostra a quantidade de itens de trabalho que estão em realização no momento. Dentro do *Kanban*, limitamos uma determinada quantidade de “atividades”, para assim garantir o fluxo de trabalho. (KANBAN UNIVERSITY, 2021)

- **Gerenciando o Fluxo**

Quando limitamos o WIP (*Work In Progress*), estamos garantindo que o projeto vai correr de forma previsível com um fluxo contínuo. Acompanhar o fluxo de trabalho

proporciona aos gestores uma visibilidade de tudo que pode ser melhorado e da onde chegará trabalhando da forma atual. (KANBAN UNIVERSITY, 2021)

- **Tornando as Políticas Explícitas**

Diariamente, é necessário tomar decisões a respeito da organização do trabalho, isso se torna mais simples através de algumas políticas explícitas, sendo elas: (KANBAN UNIVERSITY, 2021)

- Políticas explícitas 01: como (quando, por quem) acontecerá o de reabastecimento do quadro;
- Políticas explícitas 02: quando uma atividade de trabalho é concluída, e o item de trabalho pode seguir em frente, puxando a próxima atividade;
- Políticas explícitas 03: os limites de WIP *(Work In Progress)*;
- Políticas explícitas 04: como ocorrerá o tratamento de itens de trabalho de diferentes classes de serviço;
- Políticas explícitas 05: qual será os horários das reuniões e conteúdo;
- Políticas explícitas 06: demais princípios e acordos de colaboração.

Recomenda-se deixar essas políticas em uma área bem visível, podendo ficar ao lado do quadro *Kanban*, assim elas vão auxiliar a auto organização dentro do grupo de trabalho. Essas políticas devem ser de pouca quantidade, bem simples, bem definidas, visíveis, muito aplicáveis, modificadas com facilidade por quem presta o serviço e devem ser definidas em conjunto, envolvendo todas as partes, sendo eles os clientes, partes interessadas e os colaboradores. (KANBAN UNIVERSITY, 2021)

- **Implementando Ciclos de Feedback**

Dentro do método *Kanban* conseguimos obter ciclos de *feedback* através dos quadros, das revisões (cadências) e das reuniões constantes. Esses *feedbacks* são extremamente importantes para que ocorra uma entrega com qualidade. É importante ressaltar que um *feedback* tem grande impacto dentro de uma organização, com ele podemos também obter uma grande evolução de aprendizagem e melhorias necessárias. (KANBAN UNIVERSITY, 2021)

- **Melhorar Colaborativamente, Evoluir Experimentalmente**

O Kanban é utilizado de forma colaborativa para realizar mudanças contínuas através de experimentos baseados no método científico. O *feedback* e as métricas são de extrema importância para direcionar para o caminho evolutivo, criando experimentos considerados seguros, assim, se o que acreditam estiver correto e a experiência proporcionar um resultado positivo, mantém-se a mudança, caso contrário, se os resultados forem negativos, podemos voltar ao estado anterior. (KANBAN UNIVERSITY, 2021)

Práticas Específicas

Existem algumas práticas específicas do Kanban, que serão explicadas a seguir.

- **STATIK**

Uma abordagem Statik é considerada uma forma consistente de implantar o método Kanban em uma determinada empresa. Ela precisa ser aplicada de forma iterativa em cada serviço, respeitando 6 passos básicos em sua abordagem. Conforme vai aplicando cada passo estipulado pode ser descoberto alguma informação nova, tendo muitas vezes que repetir os passos anteriores. Assim, o STATIK deve ser utilizado como um ciclo de *feedback*, alimentando as atividades existentes e futuras, e não utilizado somente uma vez como um processo sequência. (KANBAN UNIVERSITY, 2021)

Cada um destes 6 processos pode levar de 4 horas até 4 dias para ser executado, devendo ser considerado que os processos devem ser feitos com grande parte dos envolvidos no projeto, e não de forma isolada. (KANBAN UNIVERSITY, 2021)

- **Passo 01: Identificar fontes de insatisfação**: para que tenhamos sucesso com o método *Kanban*, precisamos identificar se algum membro da equipe está insatisfeito com algo em específico, é necessário também saber com o que os clientes estão insatisfeitos. Identificando cada insatisfação

conseguimos realizar as mudanças necessárias com mais motivação. (KANBAN UNIVERSITY, 2021)

- **Passo 02: Analisar a demanda:** tendo em vista que o *Kanban* presa por gerenciar o trabalho e não os trabalhadores, precisamos saber o que os clientes solicitam e qual o meio que ele utiliza para fazer a solicitação, além de saber exatamente quais sãos os tipos de trabalhos e padrões de demanda. Com isso, teremos uma visão completa do trabalho que está sendo desenvolvido. (KANBAN UNIVERSITY, 2021)

- **Passo 03: Analisar as capacidades do sistema:** com alguns dados (relatórios) históricos conseguimos saber como se encontra a equipe, identificando a capacidade de cada um, além de saber se as demandas dos clientes estão sendo entregues de acordo com o prazo determinado. (KANBAN UNIVERSITY, 2021)

- **Passo 04: Modelar o fluxo de trabalho:** necessitamos identificar como ocorre o trabalho no momento, se ele acontece de forma sequencial, paralelo ou não possui uma ordem particular, com essas informações podemos definir as colunas que utilizaremos no quadro Kanban. (KANBAN UNIVERSITY, 2021)

- **Passo 05: Identificar as classes de serviço:** é importante compreendermos como os itens entram e se mantêm (são tratados) no sistema atual. (KANBAN UNIVERSITY, 2021)

- **Passo 06: Projetando o sistema Kanban:** Projetamos o sistema Kanban de acordo com todas as informações adquiridas anteriormente, ele consiste principalmente em um quadro e cartões, possuindo alguns elementos como métricas, cadências e políticas. (KANBAN UNIVERSITY, 2021)

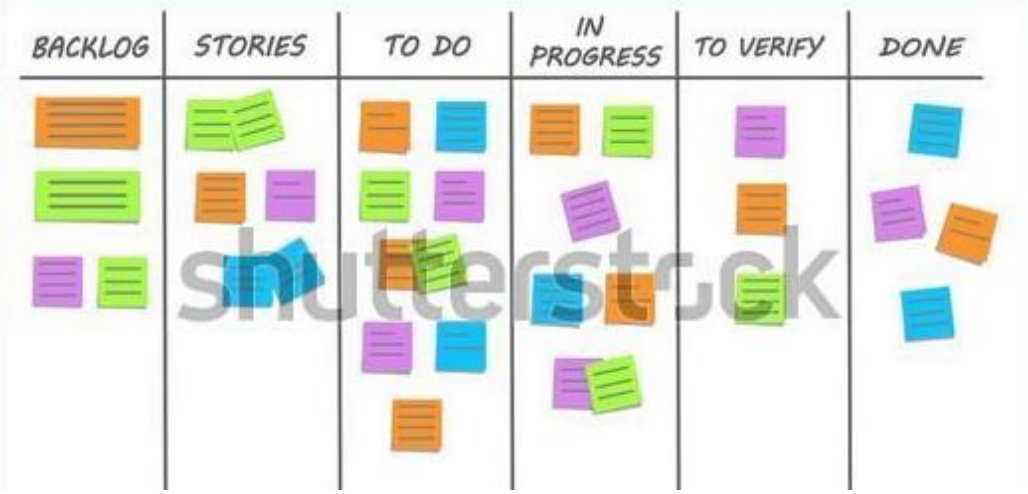
- **Quadros *Kanban***

Para que possamos visualizar o sistema *Kanban*, utilizamos os quadros, que, no caso, consiste em puxar o cartão (trabalho) da esquerda para a direita conforme os processos vão sendo concluídos e, assim que puxamos um cartão, um novo cartão entra na esquerda do quadro. Assim que um cartão sai pelo lado direito do quadro, significa que o produto foi entregue ao cliente. (KANBAN UNIVERSITY, 2021). É importante ressaltar que o sistema (método) *Kanban* pode ser aplicado em qualquer empresa, de qualquer ramo. (BENSON, 2011)

Dentro do método *Kanban*, precisamos definir um ponto de entrega e a quantidade de "itens de trabalho" permitido (*Work In Progress*), esses itens de trabalho podem possuir diferentes tamanhos e serem de diversos tipos. Eles são comumente exibidos em notas individuais (pedaço de papel), conhecidos como cartões ou *sticky notes*. (KANBAN UNIVERSITY, 2021)

Denominamos como fluxo de trabalho as atividades que cada item de trabalho passa. Baseando-se no conceito de que o *Kanban* preza por iniciar pelo que você faz hoje na empresa, assim o fluxo de trabalho real passa a ser modelado no *Kanban*. Exibimos através de colunas cada passo individual do fluxo de trabalho.

Para que você consiga compreender esse conceito, vamos analisar a Figura 03. Nós temos o *Backlog*, onde fica localizados os cartões (cada um com seus requisitos) esperando em fila para entrar no quadro, na sequências as histórias, limitadas pela quantidade permitida (WIP), a coluna seguinte contém as pendências a serem resolvidas, seguida pela coluna das atividades em progresso. Na sequência vem a coluna de verificar, onde podemos executar todos os testes necessários, para por fim, ir para a última coluna, onde o trabalho foi concluído. (MARIOTTI, 2012)



Fonte: 1809436141

A quantidade de cartões disponibilizados no painel *Kanban* representa a quantidade máxima permitida em cada fase de um sistema. Os cartões funcionam como uma forma de sinalizar, permitindo uma nova tarefa ser executada somente quando um cartão se encontra disponível. Este método é considerado de baixo custo e simples de ser implementado, com ele, é possível administrar o limite de atividades que estão em andamento, a fim de garantir que a equipe tenha um bom desempenho, sem se sentirem sobrecarregados. (MARIOTTI, 2012)

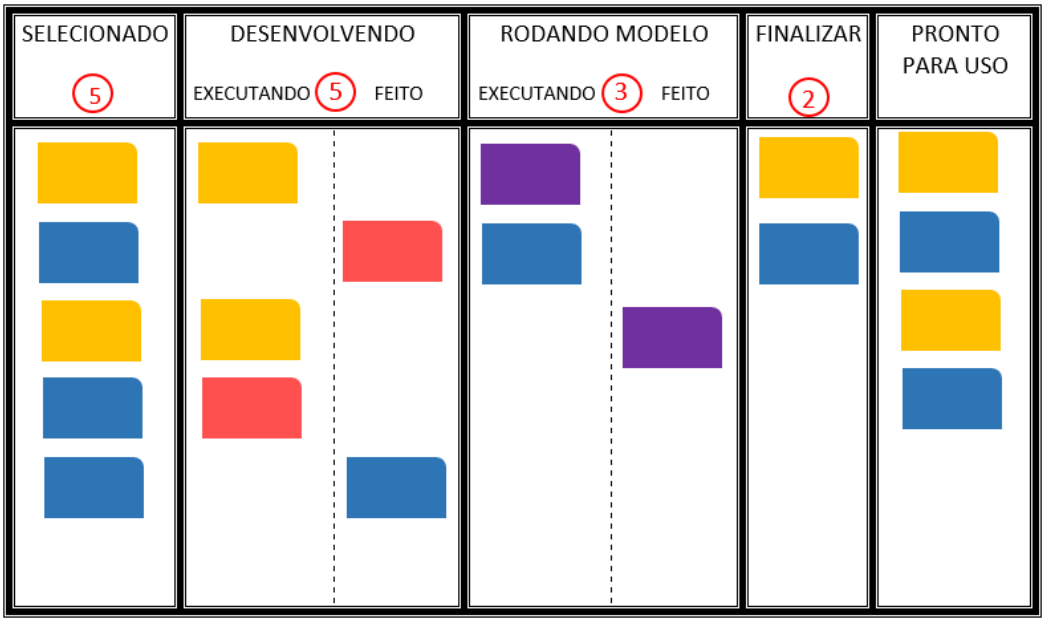
É extremamente importante ressaltar que, cada empresa formará o seu quadro *Kanban* de acordo com a sua realidade atual, sendo assim, para cada ramo, denominaremos as colunas de forma distinta, a fim de se adequar a empresa. Assim, cada sistema *Kanban* e cada quadro *Kanban* são considerados únicos. (KANBAN UNIVERSITY, 2021)

- **Limites de WIP e Sistema Puxado**

O número máximo de itens de trabalho permitido *(Work In Progress*) pode ser definido por pessoas, por colunas, por estados do trabalho ou ainda por tipo de trabalho. Anderson (2010) acrescenta que, quando aplicamos os limites (WIP) em todas as colunas do quadro, consequentemente reduzimos o *lead-time* (tempo de espera). Os limites têm como objetivo proporcionar um equilíbrio entre a demanda e

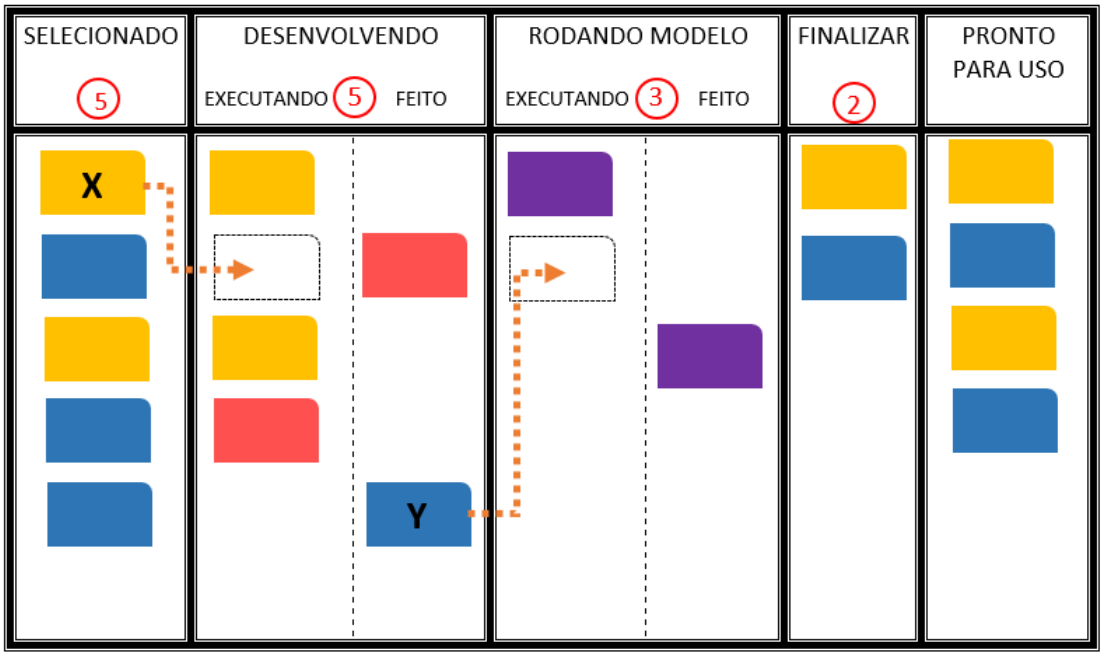
a capacidade de desenvolvimento ao longo do tempo. Esses limites de WIP consistem em um número dentro de um círculo acima de cada coluna, como apresentado na Figura 04. (KANBAN UNIVERSITY, 2021)

Figura 04 – Modelo de quadro Kanban com seus limites de WIP



Fonte: Adaptado de KANBAN UNIVERSITY (2021)

Quando limitamos o trabalho permitido, automaticamente aplicamos o princípio de puxar, assim, conforme um cartão vai se movendo para a direita do quadro, liberamos espaço, pois não estamos utilizando a capacidade máxima de WIP, então, devemos puxar um novo cartão da esquerda. Como apresentado na Figura 05, um cartão “X” da coluna “selecionado” é puxado para a coluna “desenvolvimento”, pois o cartão “Y”, que antes estava na coluna "desenvolvimento", foi puxado para a coluna “rodando modelo”. Assim, podemos observar que o trabalho concluído é considerado mais importante do que um novo trabalho, assim iniciamos um trabalho e o finalizamos. (KANBAN UNIVERSITY, 2021)



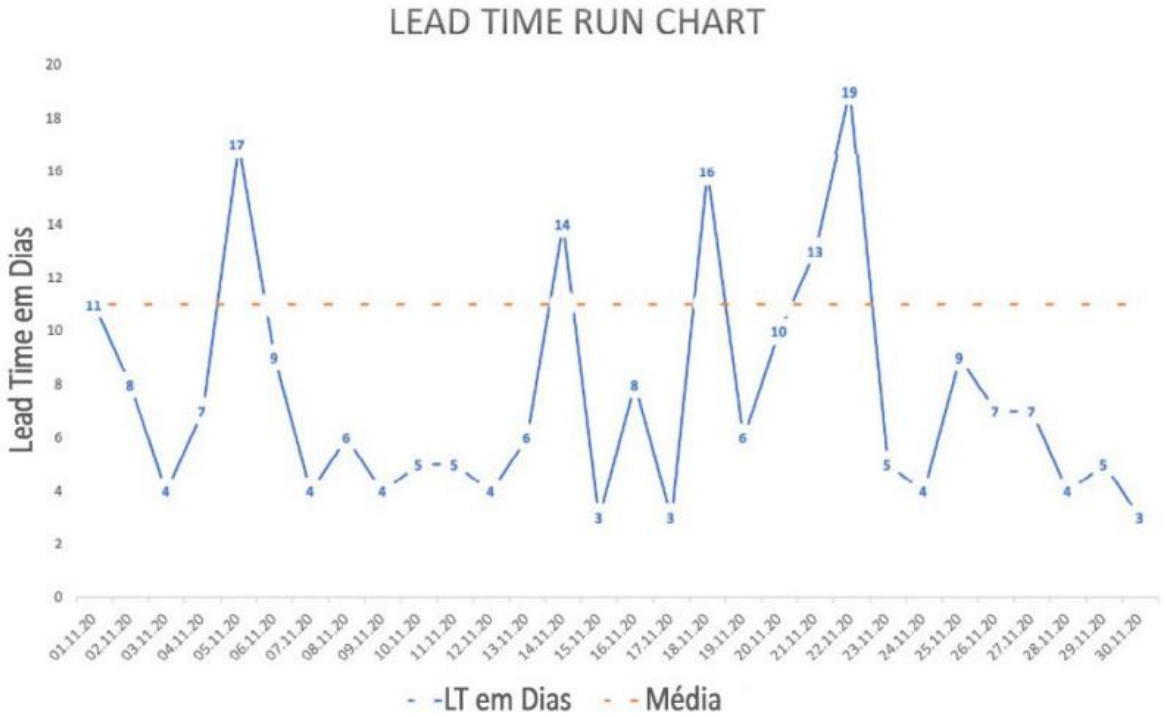
Fonte: Adaptado de KANBAN UNIVERSITY (2021)

Principais Métricas do *Kanban*

Existem diversas métricas dentro do *Kanban*, dentre elas podemos citar as três principais, o *Lead Time*, o *Delivery Rate e o WIP (Work In Progress).* Essas métricas são aplicadas em diversas representações gráficas, assim, torna-se possível compreender como o sistema está se comportando e ainda identificar as melhorias necessárias. (KANBAN UNIVERSITY, 2021)

- **Métrica 01 - O *Lead Time*:** representa o tempo que um único item de trabalho leva desde o início até a sua conclusão. (KANBAN UNIVERSITY, 2021)
- **Métrica 01 - *Delivery rate* (Taxa de entrega):** representa a quantidade de itens de trabalho que foram completados por unidade de tempo, podendo ser novas contratações por mês ou treinamentos por meses. (KANBAN UNIVERSITY, 2021)
- **Métrica 01 - WIP (*Work in Progress* ou em português, trabalho em progresso):** representa a quantidade de itens de trabalho estipulada para cada fase do sistema. (KANBAN UNIVERSITY, 2021)

Figura 06 – Modelo de gráfico de execução



Fonte: KANBAN UNIVERSITY (2021)

A Figura 06 retrata um gráfico de execução (*run chart*), utilizado pelo *Lead Time* dos itens de trabalhos concluídos, cujo são traçados sequencialmente na linha do tempo, tornando uma ferramenta muito importante para analisar as tendências do *lead time.*

Cadências do *Kanban*

Quando implantamos o sistema *Kanban*, podemos observar que não possuímos inicialmente tantos *feedbacks*, que por sua vez, com o passar do tempo, o ciclo de *feedback* evolui, melhorando assim a maturidade da equipe. Quando se aplica o sistema *Kanban*, precisamos compreender que cada elemento do *Kanban* vai ser personalizado de acordo com o contexto atual da empresa, sendo assim, com as cadências não são diferentes, ou seja, será preciso: (KANBAN UNIVERSITY, 2021)

- Verificar se já existem reuniões ou revisões dentro da empresa;
- Identificando as reuniões podemos manter os nomes existentes, usando o nome da cadência padrão ou ainda criar um outro;
- Por fim, devemos escolher qual será a frequência e a duração das reuniões, tendo como base o contexto atual.

Na maioria dos casos, adotar reuniões frequentes com uma duração menor, aumentam a agilidade ao longo do tempo. Podemos observar ainda que com menos participantes conseguimos reuniões regulares mais focadas, mais estruturadas e melhor gerenciadas. (KANBAN UNIVERSITY, 2021)

Tabela 01 – Reuniões

CADÊNCIA	EXEMPLO DE FREQUÊNCIA	PROPÓSITO
Team Kanban Meeting (Reunião do Time Kanban)	*Daily (*Diária)	Observar e acompanhar o status e o fluxo de trabalho (não os trabalhadores). Como podemos entregar os itens de trabalho no sistema rapidamente? A capacidade ficou disponível? O que devemos fazer a seguir?
Team Retrospective (Retrospectiva do Time)	Quinzenal ou mensal	Refletir sobre como os times gerenciam seu trabalho e como eles podem melhorar.
Internal Team Replenishment Meeting (Reunião de Reabastecimento Interno do Time)	Semanal ou quando necessário	Selecione os itens da lista de trabalho para fazer a seguir.

Fonte: KANBAN UNIVERSITY (2021)

O modelo *Kanban* aplicado no desenvolvimento de *software*

Para que seja possível fazer o uso do modelo *Kanban* no desenvolvimento de *software* a empresa necessita de um painel, seja ele físico (loura, painéis, paredes, tabuleiro) ou digital, tendo o intuito de manter esse painel bem visível, pois o princípio do *Kanban* é a sinalização visual. (MARIOTTI, 2012)

Anderson (2010) ressalta que para que seja considerado que a equipe realmente faz uso do modelo *Kanban*, é preciso ser possível puxar tarefas, aplicando três etapas cruciais:

- Etapa 01: é preciso criar o painel de visualização;
- Etapa 02: é preciso limitar os processos WIP (*Work In Progress*);

- Etapa 03: é preciso gerenciar o *lead-time*, aplicando o conceito de ter que puxar uma nova tarefa quando um cartão se encontra disponível.

- **Priorização**

Anderson (2010), ressalta que para que o modelo *Kanban* traga resultados positivos, é necessária uma dedicação maior da gestão, gerenciando os limites, tendo como foco sempre o desenvolvimento do que foi planejado inicialmente. Quando a equipe consegue implementar os três primeiros passos citados anteriormente, os resultados começam a aparecer, sendo eles: (MARIOTTI, 2012)

- Os códigos de grande qualidade;
- *Lead-time* de desenvolvimento considerado curto;
- Maior controle do desempenho de produção.

- **Maturidade na produção**

Com a produção de códigos de alta qualidade e o cumprimento de todos os prazos da entrega, consideramos que a equipe conseguiu atingir a maturidade durante a produção. (BENSON, 2011) Com a maturidade eliminamos os retrabalhos, o que ocasiona em um ritmo de produção elevado. (MARIOTTI, 2012)

- **Comportamento emergente**

Atualmente, existe uma lista de comportamentos emergentes, lista essa elaborada por Anderson (2010). Os comportamentos que compõem essa lista são: (MARIOTTI, 2012)

- Item 01: Os processos são limitados e de acordo com cada fluxo do projeto;
- Item 02: Não é necessário o uso de iteração durante o desenvolvimento
- Item 03: Conseguimos gerenciar o custo de implementação;

- Item 04: Os valores são otimizados para classes de serviços;
- Item 05: Possui um gerenciamento de risco com alocação de capacidade;
- Item 06: Uma gestão quantitativa;
- Item 07: Tem fortes chances de atingir outros departamentos;
- Item 08: Mesclando pequenas equipes, proporcionamos um maior grupo de trabalho.

- **Modelo *Kanban* e os sinalizadores visuais**

Os sinalizadores visuais dentro do *Kanban*, são utilizados para identificar as tarefas existentes e utilizado também para mostrar o fluxo de valor. O painel do *Kanban* é o que possibilita a visualização de todas as tarefas. Podemos montar um quadro simples e funcional dentro do *Kanban*, contendo as seguintes etapas (colunas): (MARIOTTI, 2012)

- Etapa 01: Análise;
- Etapa 02: Desenvolvimento;
- Etapa 03: Aceitação;
- Etapa 04: Implantação.

As colunas dentro do *Kanban* representam as diferentes etapas de uma tarefa durante o período de desenvolvimento, no exemplo temos a etapa de análise, desenvolvimento, aceitação e a etapa de implantação. Já o cartão (ou sinalizador) deve ser puxado (movido) dentro do quadro de uma etapa para outra, até que o projeto seja finalizado. Cada cartão contém uma breve história do usuário, com o intuito de apresentar os requisitos necessários. Inicialmente todos os cartões entram para a fila de *backlog*, ali permanecem até que seja liberada a entrada nas colunas de produção. A quantidade de cartões durante o processo de desenvolvimento é limitada, sendo assim, um cartão só sai da fila de *backlog* para entrar nas colunas de desenvolvimento quando as tarefas de um cartão (que está atualmente nas colunas de desenvolvimento) forem finalizadas. Esse procedimento seria o de “puxar” um novo cartão para iniciar o desenvolvimento. (MARIOTTI, 2012)

- **Mitos do modelo Kanban**

Costumam dizer que o *Kanban* não é um processo com iterações, isso é um mito. Existe a iteração no *Kanban* e ela pode ser utilizada quando necessário, sendo assim um recurso opcional. Comenta-se ainda que o *Kanban* não usa estimativas, sendo também um mito, pois existem sim a possibilidade de utilizar estimativas no *Kanban* quando necessárias, sendo também um recurso opcional. (MARIOTTI, 2012)

É comum ouvirmos falar ainda que o *Kanban* é melhor do que *Scrum* e o XP, quando na verdade tudo não passa de um mito. O *Kanban* é apenas mais uma ferramenta que pode ser utilizada no processo de desenvolvimento de um *software*, ela não veio para substituir ou sobressair outra ferramenta, mas, sim para implementar os conceitos de mudança, fazendo uso do modelo de visualizações, estipulando limites de WIP (*Work In Progress*) e obter evolução nos resultados positivos. (MARIOTTI, 2012)

SAIBA MAIS

Podemos realizar combinações entre os métodos ágeis, sendo assim o Kanban, pode ser implementado na sua empresa juntamente com o processo atual. o Kanban caro (a) aluno (a) tem como propósito agregar, sendo assim, após compreender o processo atual adotado pela empresa, implementamos os conceitos kanban a fim de identificar os problemas existentes no processo. (NAVARRO, 2021)

CONSIDERAÇÕES FINAIS

Quando falamos de programação extrema (XP), não podemos deixar de falar dos seus 4 princípios, tendo notado que quando se implementa a metodologia respeitando todos os princípios, fica impossível não entregar um produto final ao cliente com qualidade. O primeiro princípio trata sobre a comunicação, pois foi identificado que boa parte do sucesso de um software tem como base a comunicação entre os integrantes da equipe de desenvolvimento, incluindo sempre os clientes. O segundo princípio do XP é a simplicidade, aqui preza-se por desenvolver um software totalmente funcional, da maneira mais simples possível. O terceiro princípio é o feedback, pois defende-se que quanto mais feedbacks tivermos em um sistema, melhor será a comunicação entre a equipe, ocasionando na simplicidade do projeto. Por último, temos o princípio da coragem, em que muitas vezes é necessário descartar uma linha de código já pronta a fim de refazê-la de maneira mais simples, mais eficaz. Podemos acrescentar ainda que a coragem está diretamente ligada à comunicação, pois é preciso compreender tudo que está sendo feito para que possa realizar as alterações necessárias. (BECK, 2004)

De acordo com Anderson (2010), com o uso do Kanban, conseguimos eliminar os problemas que afetam o desempenho, além de desafiar o time para resolver todos os problemas, com o intuito de não perder o ritmo do trabalho. Anderson (2010) acrescenta ainda que o Kanban possibilita a equipe a ter mais visibilidade em todos os processos, deixando bem claro os problemas existentes e focando toda a equipe no quesito qualidade, podendo, assim, prevenir problemas futuros. O modelo Kanban sendo aplicado a um projeto tende a reduzir o estresse e melhorar a colaboração da equipe, fazendo com que todos os prazos de entrega de tarefas sejam cumpridos. Isso tudo reflete no fortalecimento da confiança dos clientes com a equipe de desenvolvimento. (MARIOTTI, 2012)

REFERÊNCIAS

BECK, K. **Programação extrema explicada**: acolha as mudanças. Porto Alegre, RS: Bookman, 2004

SOMMERVILLE. **Engenharia de Software,** São Paulo: Addison-Wesley, 9 ed., 2011. ISBN-10: 8579361087 ISBN-13: 9788579361081.

ANDERSON. D. J. **Kanban: Successful Evolutionary Change for Your Technology Business**, 2010.

BENSON. J, BARRY. T. M. **Personal Kanban**. 2011.

MARIOTTI, F. S. **Kanban: o ágil adaptativo Introduzindo Kanban na equipe ágil-** Edição 45 - Engenharia de Software Magazine – 2012.

UNIDADE IV

APLICANDO O SCRUM NO DESENVOLVIMENTO DE SISTEMA UTILIZANDO MÉTODOS E PRÁTICAS ÁGEIS

Professora Especialista Camila Sanches Navarro

Plano de Estudo:

- O framework SCRUM;
- Os mitos das metodologias ágeis;
- Ferramentas que auxiliam as metodologias ágeis.

Objetivos de Aprendizagem:

- Compreender a *Framework Scrum;*
- Visualizar alguns mitos das metodologias ágeis;
- Conhecer algumas ferramentas que auxiliam a utilização do método ágil.

INTRODUÇÃO

O *Framework Scrum* auxilia o desenvolvimento de produtos desde os anos 90 e começou inicialmente de forma primária por um determinado grupo de especialistas em tecnologia, após trinta anos no mercado continua sendo um tema extremamente forte. (MJV, 2020)

Com o *Framework Scrum*, conseguimos resolver problemas com um grau elevado de complexidade, isso com a ajuda de diversas técnicas e processos que precisam ser implementados, sempre visando entregar um produto de qualidade para o cliente. (SCHWABER E SUTHERLAND, 2013)

De acordo com Schwaber e Sutherland (2013), o *Scrum* é sustentado por três pilares, sendo eles: A transparência, em que todos os processos precisam ser sempre bem claros, fazendo com que o time trabalhe o mais alinhado possível. O próximo ponto é a inspeção, que consiste em inspecionar todas as tarefas, com uma certa frequência, assim garantimos que tudo está sendo feito como previsto. E, por fim, a adaptação, em que qualquer irregularidade identificada durante a inspeção necessita ser ajustada o quanto antes, pois, só assim, evitaremos futuros erros. (ENACTUS, 2017)

O *Scrum*, mesmo sendo implementado há muitos anos, ainda é motivo de muita desconfiança por diversas empresas, na maioria das vezes, por falta de entendimento e de conhecimento, sendo assim, é comum encontrarmos alguns mitos referentes à metodologia *Scrum*, esclarecemos cada um deles no decorrer da apostila. (PAVANI, 2020)

Existem diversas ferramentas que permitem a utilização das metodologias ágeis com mais precisão, em que todos os integrantes do time conseguem ter acesso independente de onde estejam. Nos dias atuais, é comum ouvirmos falar no *Home Office*, muito aderido por empresas de tecnologia e, nesse cenário, utilizar uma ferramenta como o *Trello*, por exemplo, se torna indispensável, pois todo o time *Scrum* consegue trabalhar de forma clara e alinhada.

1 O FRAMEWORK SCRUM

Figura 01 – Scrum

Fonte: 1262757337

O *Scrum* é um *framework* que permite solucionar problemas complexos desde o início de 1990, com ele, podemos implementar diversas técnicas e processos, permitindo entregar um produto final com qualidade. De acordo com Schwaber e Sutherland (2013), podemos caracterizar o *Scrum* como:

- Leve;
- Fácil de compreender;
- Muito difícil de dominar.

Pilares do Scrum

Segundo Schwaber e Sutherland (2013), existem três pilares que sustentam o *Scrum*: a transparência, a inspeção e a adaptação.

- **Transparência**

Todos os processos do desenvolvimento precisam estar sempre visíveis, assim, todos os integrantes da equipe conseguem trabalhar de forma mais alinhada, compreendendo cada etapa. (ENACTUS, 2017). Para que isso seja possível, é necessário seguir um padrão comum, em que todos devem utilizar uma linguagem comum quando se trata do projeto. (SCHWABER E SUTHERLAND, 2013)

- **Inspeção**

Todas as tarefas devem ser inspecionadas com uma certa frequência, a fim de averiguar como está o andamento do projeto e se todos os processos estão sendo seguidos corretamente, com isso, caso algum membro da equipe precise de ajuda, é possível direcionar alguém para auxiliar. (ENACTUS, 2017)

- **Adaptação**

Caso durante a inspeção, for identificado que o processo não está acontecendo como deveria (está se desviando, fora dos limites permitidos) ou ainda o produto resultado não se apresentar como o esperado, deve-se, então, ajustar o processo ou o projeto a ser desenvolvido. (SCHWABER E SUTHERLAND, 2013). Quando identificado, é necessário realizar os ajustes o quanto antes, para, assim, evitar que haja mais desvios. (ENACTUS, 2017)

O *Time Scrum*

Já ouviu falar em times auto-organizáveis e multifuncionais? Os *Times Scrum* possuem as duas características, sendo elas as auto-organizáveis e multifuncionais, cujo o time decide qual o melhor caminho para finalizarem o seu trabalho, tendo todas as habilidades necessárias, não dependendo assim de nenhuma outra pessoa além da sua equipe. Os produtos são entregues pelo *Time Scrum* de modo iterativo e incremental, o que possibilita as realimentações, devido aos *feedbacks*. Compõe o *Time Scrum*, o *Product Owner* (PO), o *Scrum Master* (SM) e o time de

desenvolvimento (ST), como apresentado na Figura 02. (SCHWABER E SUTHERLAND, 2013)

Figura 02 – Time Scrum

Fonte: 408767248

- **O *Product Owner***

Segundo Schwaber e Sutherland (2013), quando falamos do *Product Owner* (dono do produto), estamos falando do responsável por potencializar o valor do time de desenvolvimento e potencializar também o valor do produto. Contudo, o *Product Owner* fica responsável por fazer o gerenciamento do *Backlog* do Produto, tendo que realizar as seguintes funções:

- O *Product Owner* precisa expor de forma clara todos os itens do *Backlog* do Produto;
- O *Product Owner* necessita ordenar todos os itens do *Backlog* do produto, a fim de atingir as suas metas com mais precisão;
- O *Product Owner* precisa assegurar o valor do trabalho que o time de desenvolvimento realiza;

- O *Product Owner* precisa fazer com que o *Backlog* do Produto seja transparente e muito visível a todos da equipe, deixando claro o que o *Time Scrum* irá trabalhar na sequência;
- O *Product Owner* precisa fazer com que todos integrantes do time de desenvolvimento compreendam todos os itens do *Backlog* do produto.

Todos esses trabalhos são do *Product Owner*, embora quando necessário, ele pode solicitar que o time de desenvolvimento execute as tarefas necessárias, ainda assim, ele continua sendo o único responsável pelo trabalho. O cargo de *Product Owner* é de extrema importância, sendo assim, é indispensável que todos os membros da equipe tenham muito respeito por ele, quando algum outro membro do time desejar realizar alguma alteração no sistema, deve inicialmente, convencer o *Product Owner*, pois ele é o único responsável por gerenciar o *Backlog* do produto (consiste em uma lista com todos os itens indispensáveis para entregar o projeto). (SCHWABER E SUTHERLAND, 2013)

- **O Time de Desenvolvimento**

O Time de Desenvolvimento, caro (a) aluno (a), é formado por programadores que tem como objetivo entregar uma versão usável do sistema, sendo os únicos responsáveis por criarem incrementos, eles têm autonomia de gerenciar e organizar o seu trabalho. É indispensável que o time de desenvolvimento tenha as seguintes características: (SCHWABER E SUTHERLAND, 2013)

- O Time de Desenvolvimento: decide como vai transformar o *Backlog* do Produto em incrementos de funcionalidades, sendo eles auto-organizados;
- O Time de Desenvolvimento: possui todas as aptidões necessárias para criar incrementos do projeto, sendo considerados multifuncionais;
- O Time de Desenvolvimento: são todos conhecidos como desenvolvedores, não existindo distinção entre os integrantes do time de desenvolvimento;

- O Time de Desenvolvimento: a responsabilidade do projeto a ser desenvolvido pertence a toda a equipe, mesmo que algumas funcionalidades sejam realizadas de forma individual.

Quanto à quantidade de membros no Time de Desenvolvimento, é recomendável que não seja tão grande, pois geraria uma complexidade maior para gerenciar, embora também não possa ser muito pequeno, pois poderá ocasionar na incapacidade de entregar o incremento potencialmente utilizável. Contudo, Schwaber e Sutherland (2013) recomenda que:

> *"*O tamanho ideal do Time de Desenvolvimento é pequeno o suficiente para se manter ágil e grande o suficiente para completar uma parcela significativa do trabalho dentro dos limites da Sprint.*"* (SCHWABER E SUTHERLAND, 2013, pg. 21)

- **O *Scrum Master***

O *Scrum Master*, caro (a) aluno (a), faz com que todos os membros da equipe compreendam e apliquem o *Scrum*, verificando se seguem as teorias, as práticas e as regras do *Scrum*. Visando potencializar o valor do *Time Scrum*, o *Scrum Maste*r auxilia os membros do time a compreenderem quais das suas interações são úteis e quais são dispensáveis, ajudando ainda a alterarem as interações necessárias. (SCHWABER E SUTHERLAND, 2013)

Para maior compreensão, vamos entender quais são as funções do *Scrum Maste*r perante o *Time Scrum*:

- *Scrum Master* trabalhando para o *Product Owner*: O papel do *Scrum Master*, neste caso, é identificar as melhores técnicas para gerenciar o *Backlog* do Produto de forma efetiva, além de passar para o time de desenvolvimento de forma clara e objetiva todos os itens presentes no *Backlog*. O *Scrum Master*, precisa ainda, instruir o time a criar de forma clara os itens necessários do *Backlog* e, por fim, compreender a longo prazo o planejamento do produto e aplicar todas as práticas da

agilidade, devendo ainda facilitar todos os eventos *Scrum*, conforme for sendo necessário. (SCHWABER E SUTHERLAND, 2013)

- *Scrum Master* trabalhando para o Time de Desenvolvimento: precisa realizar um treinamento com o time em relação ao autogerenciamento e interdisciplinaridade e instruir e liderar os desenvolvedores durante a criação de produtos de alto valor. É indispensável o *Scrum Master* remover todos os impedimentos que possam surgir no time de desenvolvimento, facilitando também todos os eventos *Scrum*, sempre que necessário. Por fim, aqui, o *Scrum Master* necessita ainda treinar todo o time de desenvolvimento em ambientes onde o Scrum não foi compreendido e aplicado por completo. (SCHWABER E SUTHERLAND, 2013)

- *Scrum Master*, trabalhando para a Organização: O *Scrum Master* precisa liderar e treinar a Organização para aderir de forma correta o *Scrum*, planejando também implementações *Scrum* dentro da Organização. O *Scrum Master* necessita auxiliar todos os envolvidos a compreender o *Scrum* e o desenvolvimento do produto, causando algumas mudanças que ocasionaram em um aumento na produtividade do *Time Scrum* e, por fim, ele precisa trabalhar com outros *Scrum Master*, a fim de aprimorar a eficácia da implantação do *Scrum* nas organizações. (SCHWABER E SUTHERLAND, 2013)

Tabela 01: Postos-chaves a ser gerenciado

PONTO-CHAVE A SER GERENCIADO	**PRODUCT OWNER**	**TIME DE DESENVOLVIMENTO**	**SCRUM MASTER**
Retorno sobre o investimento	x		
Necessidades/objetivos de negócios	x		
Clientes e demais partes interessadas	x		
Visão do produto	x		
Releases	x		
Tarefas de desenvolvimento do produto		x	
Qualidade interna do produto		x	
Qualidade externa do produto	x	x	
Estimativas ou previsões		x	
Processos (funcionamento do Scrum)			x
Impedimentos no trabalho			x
Relacionamento e motivação do time		x	x
Riscos	x	x	x
Comunicação	x	x	x

Fonte: SABBAGH (2013)

Eventos *Scrum*

Segundo Schwaber e Sutherland (2013), existem quatro eventos formais no *Scrum* dentro de uma *Sprint*, sendo eles:

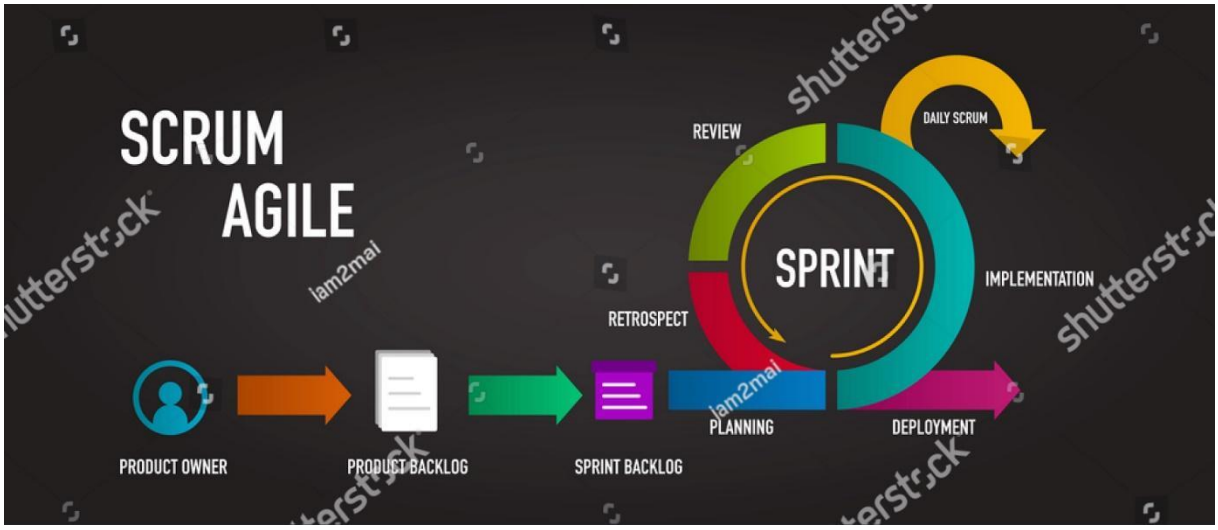
- Evento 01: Reunião de planejamento da *Sprint;*
- Evento 02: Reunião diária;
- Evento 03: Reunião de revisão da *Sprint;*
- Evento 04: Retrospectiva da *Sprint.*

Estes eventos são conhecidos como eventos prescritos e são utilizados no *Scrum* com o intuito de criar uma rotina, assim minimizamos as reuniões não definidas. Cada evento acontece no formato *time-boxed*, em outras palavras, cada evento desse possui uma duração máxima, assim que se inicia uma *Sprint* a sua duração é determinada, não podendo sofrer alterações (reduzir ou aumentar o tempo). Assim que se alcança a finalidade de um evento, os eventos restantes podem encerrar, assim só será gasto o tempo realmente necessário. (SCHWABER; SUTHERLAND, 2013)

Os eventos são de extrema importância para o *Scrum*, eles possibilitam que, quando necessário, possamos realizar algumas alterações, adaptando o sistema para que atenda às necessidades do cliente, pois cada evento tem por finalidade inspecionar cada funcionalidade de forma minuciosa, tornando todos os processos muito transparente. (SCHWABER E SUTHERLAND, 2013)

- **Sprint**

Figura 03 – Processos de uma Sprint



Fonte: 1833995965

Schwaber e Sutherland (2013), consideram uma *Sprint* como sendo o coração do *Scrum*, em que cada *Sprint* tem a duração (*time-boxed*) de até 4 semanas, assim que o time finaliza uma *Sprint* (momento em que um projeto “pronto” é lançado, cujo o *software*, por exemplo, finalizou uma funcionalidade e já pode ser utilizada pelo cliente), ele já inicia uma nova *Sprint*.

Como representado na Figura 03, as *Sprints* possuem uma reunião de planejamento da *Sprint* (*Planning*), reuniões diárias (*Daily Scrum*), o processo de desenvolvimento (*Implementation*), uma revisão da *Sprint* (*Review*) e a reunião de retrospectiva da *Sprint* (*Retrospect*). Ainda que durante uma *Sprint* não sejam realizadas alterações que possam colocar em risco o objetivo da *Sprint* e as estimativas quanto à qualidade não reduz. Por fim, o escopo pode ser renegociado entre o Time de Desenvolvimento e o *Product Owner*. Devemos ter em mente, que cada *Sprint* tem uma finalidade, ao final da *Sprint* (que não pode ser maior do que quatro semanas) é entregue ao cliente uma versão nova do sistema. (SCHWABER E SUTHERLAND, 2013)

Reunião de Planejamento da *Sprint*

Durante a reunião de Planejamento da *Sprint*, os membros da equipe precisam planejar o que vai ser feito durante a *Sprint*, é importante ressaltar que todo o *Time Scrum* participa dessa reunião. O *time-box* de uma reunião de planejamento

da *Sprint* pode durar, no máximo, oito horas, quanto o de uma *Sprint* for menor (inferior a 4 semanas) e, consequentemente, essa reunião inicial terá uma duração menor. (SCHWABER E SUTHERLAND, 2013)

Durante a reunião de planejamento da *Sprint*, o *Scrum Master* precisa garantir que todos os participantes compreendam com exatidão tudo o que vai ocorrer na *Sprint* e qual será a sua finalidade. O *Scrum Master* instrui como o *Time Scrum* precisa comportar-se, a fim de não passar os limites do *time-box*. (SCHWABER E SUTHERLAND, 2013)

De acordo com Schwaber e Sutherland (2013), uma reunião de planejamento da *Sprint* precisa responder a duas questões:

- O que podemos entregar como resultado do incremento da próxima *Sprint*?

Durante a reunião de Planejamento da *Sprint* o time de desenvolvimento faz um levantamento das funcionalidades que deverão ser implementadas durante a *Sprint* e, por sua vez, o *Product Owner* expõe qual o objetivo da *Sprint* e quais os itens de *Backlog* do Produto, para que assim, todos os integrantes do time compreendam exatamente o que deverá ser trabalhado na *Sprint*. Fica por conta do Time de Desenvolvimento decidir quais serão os itens do *Backlog* do Produto que serão desenvolvidos na *Sprint*. Na sequência, o *Time Scrum* estabelece qual será a meta da *Sprint*, assim fica claro o motivo de estar desenvolvendo o novo incremento. (SCHWABER E SUTHERLAND, 2013);

- Como será realizado o trabalho para entregar o incremento?

Após ser definido o objetivo da *Sprint* e ter selecionado os itens de *Backlog* do Produto da Sprint, o time de Desenvolvedores estabelece como serão construídas as funcionalidades necessárias durante a *Sprint*, para que possa resultar em um incremento de produto denominado “Pronto”. Chamamos de *Backlog* da *Sprint* quando juntamos os itens de *Backlog* do produto com o plano de entrega. Após o Time de Desenvolvimento concluir o planejamento de como o trabalho acontecerá durante a *Sprint*, eles se auto-organizam para colocar em prática todo o trabalho do *Backlog* da *Sprint*. O *Product Owner*, por sua vez, auxilia os participantes da reunião a

compreenderem todos os itens selecionados do *Backlog* do Produto, caso o Time de Desenvolvimento identifique que os itens selecionados são considerados poucos, ou ainda, muitos para desenvolver durante a *Sprint*, o time pode renegociar os itens de *Backlog*, a fim de aumentar ou reduzir a quantidade para a *Sprint* atual. Por fim, fica por conta do Time de desenvolvimento esclarecer todos os processos que vão ocorrer durante a *Sprint* para o *Product Owner* e para o *Scrum Master*, tendo em vista que o intuito é completar o objetivo da *Sprint*, podendo criar o incremento esperado. (SCHWABER E SUTHERLAND, 2013)

Reunião Diária

A reunião diária acontece sempre antes de iniciar o trabalho, e este evento tem o *time-boxed* de 15 minutos. O objetivo da reunião diária é identificar o que foi feito desde a última reunião e o que será necessário fazer até a próxima reunião, ela acontece sempre no mesmo horário e diariamente. De acordo com Schwaber e Sutherland (2013), durante a reunião, os participantes esclarecem alguns pontos importantíssimos, sendo eles:

- O que eu realizei ontem contribuiu para que o Time de Desenvolvimento atingisse a meta da *Sprint*?

- O que eu posso fazer hoje para contribuir com Time de Desenvolvimento, a fim de atingir a meta da *Sprint*?

- Existe algum empecilho (obstáculo) que dificulte o meu trabalho ou o trabalho do time de desenvolvimento?

Sendo assim, as reuniões diárias contribuem muito para que o Time de Desenvolvimento possa visualizar o seu progresso em relação ao objetivo da *Sprint*, se o mesmo vai ser realizado dentro do prazo esperado, assim, consequentemente, a probabilidade do Time de Desenvolvimento atingir a sua meta é extremamente grande. (SCHWABER E SUTHERLAND, 2013)

As reuniões diárias são conduzidas pelo Time de Desenvolvimento, que, por sua vez, são instruídos pelo *Scrum Master* a não ultrapassar o *time-boxed* de 15 minutos. É importante ressaltar que somente os integrantes do Time de Desenvolvimento participam da Reunião Diária. Considerada por Schwaber e Sutherland (2013) como uma reunião chave para a inspeção e para a adaptação do projeto, a reunião diária aperfeiçoa a comunicação entre o time, podendo identificar e remover com agilidade qualquer impedimento que venha a surgir, além de possibilitar tomadas de decisões com mais precisão.

Revisão da Sprint

Considerando uma *Sprint* de quatro semanas, a reunião de Revisão da *Sprint* deve ter o seu *time-boxed* de quatro horas de duração (vale ressaltar que, para *Sprints* menores, o tempo da reunião de Revisão da *Sprint* também será menor). Essa reunião acontece sempre ao final de *Sprint*, a fim de esclarecer tudo que foi feito na *Sprint*, apresentando o incremento desenvolvido para que assim todos consigam visualizar o que foi realizado na *Sprint*. (SCHWABER E SUTHERLAND, 2013)

De acordo com Schwaber e Sutherland (2013), a Reunião de Revisão inclui alguns elementos, sendo eles:

- Participaram da reunião de revisão o *Time Scrum* e os *Stakeholders* (partes interessadas)
- O *Product Owner* precisa listar quais os itens do *Backlog* do Produto foram “Prontos” e quais itens não foram “Prontos”;
- Momento em que o Time de Desenvolvimento tem para conversar sobre as dificuldades encontradas durante a *Sprint* e, como conseguiram solucionar esses problemas, além de falarem sobre o que desenvolveu muito bem também durante a *Sprin*t;
- Durante a reunião, o Time de Desenvolvimento apresenta o incremento que está “Pronto” e tira as dúvidas referentes a nova funcionalidade;

- Momento em que o *Product Owner* conversa sobre o *Backlog* do Produto, de acordo como ele se encontra atualmente, podendo estimar as possíveis datas de conclusão do projeto, com base no progresso até o presente momento;
- Durante a reunião o *Time Scrum* discute sobre o que deve ser feito na sequência, contribuindo, assim, com informações valiosas para a reunião de Planejamento da próxima *Sprint*;
- Discute-se também sobre como o mercado do produto desenvolvido pode ter sofrido mudanças e levanta-se o que é indispensável fazer na sequência;
- É feito uma análise da linha do tempo, do orçamento disponível e do mercado para o próximo incremento (versão) a ser desenvolvida.

A Reunião de Revisão da *Sprint* resulta em um *Backlog* do Produto verificado, que provavelmente definirá o próximo *Backlog* do Produto da *Sprint*. (SCHWABER E SUTHERLAND, 2013)

Retrospectiva da Sprint

Segundo Schwaber e Sutherland (2013), a retrospectiva da *Sprint* é uma excelente oportunidade para o *Time Scrum* observar todo o percurso da *Sprint*, pontuando todos os detalhes possíveis, a fim de elaborar um propósito de melhorias, para que seja aplicado na próxima *Sprint*. A Retrospectiva da *Sprint* acontece após a Revisão da *Sprint*, e antecede a Reunião de Planejamento da próxima *Sprint*. O *time-boxed* dessa reunião é de três horas, quando a *Sprint* é de quatro semanas.

Schwaber e Sutherland (2013), afirmam que a finalidade da Retrospectiva da *Sprint* é:

- Examinar como a *Sprint* foi em relação aos processos, às ferramentas, às pessoas e aos relacionamentos;
- Apontar os itens que acontecerem da forma esperada e onde podemos executar algumas melhorias;
- Elaborar um plano com o intuito de realizar melhorias na forma como o *Time Scrum* realiza o seu trabalho.

No decorrer da Retrospectiva da *Sprint* o *Scrum Master* procura entusiasmar o *Time Scrum,* a fim de procurar sempre melhorar o processo de desenvolvimento e as práticas do *Scrum* para que na próxima *Sprint* seja possível alcançar um resultado ainda melhor. Ao final da retrospectiva, fica por conta do time identificar onde podem ocorrer melhorias para a próxima *Sprint*. (SCHWABER E SUTHERLAND, 2013)

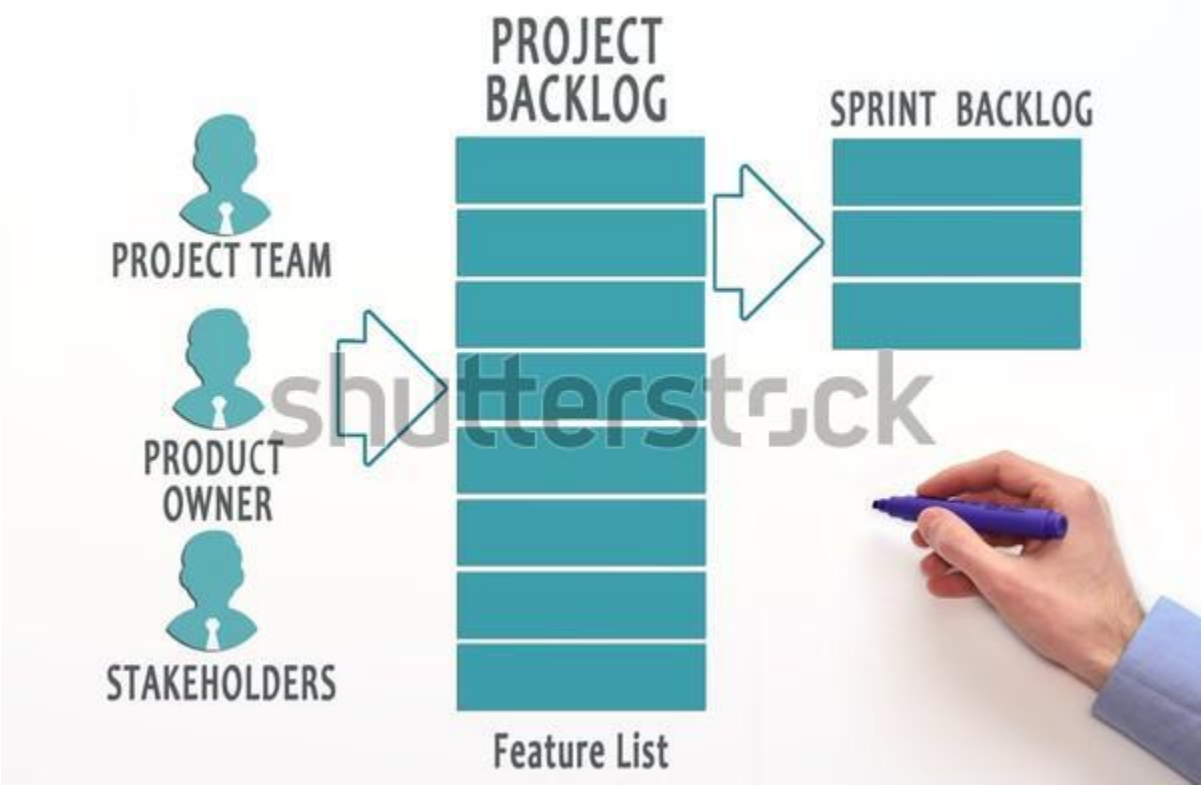
- **Artefatos do *Scrum***

Os artefatos do *Scrum* são projetados com o intuito de deixar bem claro a todos os integrantes do time as informações chaves, fazendo com que todos tenham o mesmo entendimento de todos os artefatos. Podemos dizer ainda, que os artefatos fornecem transparência para inspeções e adaptações. (SCHWABER E SUTHERLAND, 2013)

***Backlog* do Produto**

Backlog do Produto consiste em uma lista com todas as funcionalidades que o sistema necessita ter, cujo *Product Owner* é o responsável. Podemos falar que um *Backlog* do Produto nunca vai estar completo, acrescentando ainda que ele é dinâmico, pois sofre alterações com frequência, a fim de identificar qual a necessidade do produto para torná-lo mais útil. Em resumo, o *Backlog* do Produto é um dos três artefatos do *Scrum*, sendo caracterizado como uma lista de todas as funcionalidades e requisitos necessários para o sistema, essa lista é formada por itens de *Backlog*, podendo ser representadas por melhorias ou algumas correções, quando necessário, contudo, o *Backlog* do Produto é a fonte de todos os requisitos necessários do Produto. (SCHWABER E SUTHERLAND, 2013)

Figura 04 – Backlog do Produto



Fonte: 416242063

Como apresentado na Figura 04, os itens do *Backlog* que se encontram no topo da lista precisam ser bem claros e minuciosamente detalhados, ao contrário dos itens que se encontram no final da lista (parte de baixo), sendo assim, quanto menor a colocação na lista, menos detalhes são necessários no momento. Os itens do *Backlog* que vão ser implementados na próxima *Sprint* necessitam ser bem definidos, para que, assim, seja possível desenvolver todos dentro do time-boxed da Sprint. (SCHWABER E SUTHERLAND, 2013)

- ***Backlog* da *Sprint***

Os itens do *Backlog* selecionados para a *Sprint* se compõem do *Backlog* da *Sprint*, simultaneamente com o plano para a entrega do incremento, como apresentado na Figura 03. O time de Desenvolvimento realiza modificações no *Backlog* da *Sprint* no decorrer da *Sprint*, assim, o *Backlog* da *Sprint* vai surgindo no decorrer da *Sprint*. Contudo, podemos afirmar que o *Backlog* da *Sprint* consiste na previsão que o Time de Desenvolvimento tem sobre qual será a próxima

funcionalidade necessária no próximo incremento. É importante ressaltar que somente o Time de Desenvolvimento pode realizar alterações no *Backlog* da *Sprint*. (SCHWABER E SUTHERLAND, 2013)

Incremento

De acordo com Schwaber e Sutherland (2013), o incremento é formado por todos os itens do *Backlog* do Produto finalizados na *Sprint* atual e nas anteriores, pois sempre que se finaliza uma *Sprint*, obtemos um incremento “Pronto”, em outras palavras, a nova funcionalidade do sistema, por exemplo, deve ser uma versão utilizável. Cada incremento definido como “Pronto” já passou por testes, a fim de garantir que está funcionando perfeitamente, individualmente e em conjunto com os incrementos “Prontos” anteriormente.

Definição de “Pronto”

Schwaber e Sutherland (2013), afirma que, sempre que um incremento ou um *Backlog* do Produto é definido como “Pronto”, todos os integrantes do time precisam compreender o que o “Pronto” significa. Sabemos que, embora o seu significado possa variar de uma *Time Scrum* para outra, em geral, o “Pronto” significa que o trabalho realizado dentro de uma *Sprint*, por exemplo, foi finalizado.

2 OS MITOS DAS METODOLOGIAS ÁGEIS

Figura 05 – Desenvolvimento Ágil

Fonte: 576494164

Existem diversas organizações que ainda visualizam as metodologias ágeis com muita desconfiança, fazendo com que fiquem estagnadas, não aderindo ao método ágil. Vamos agora, caro (a) aluno (a) descrever alguns mitos e erros sobre o uso de metodologias ágeis. (PAVANI, 2020)

Mitos das Metodologias Ágeis

- **A metodologia ágil é somente para times e projetos pequenos**

Mito, pois as metodologias ágeis podem ser utilizadas por times maiores e projetos grandes, desde que seja respeitado os conceitos para a formação dos times. (PAVANI, 2020)

- **Métodos ágeis só funciona em times de tecnologia**

Mito, pois a agilidade nada mais é do que a forma do time pensar, e não a forma de pensar em tecnologia. Esse mito originou-se pelo fato de que as metodologias ágeis nasceram na área de tecnologia, contudo, toda e qualquer empresa tem condições e permissão para adotar os métodos ágeis. (MJV, 2020)

- **Utilizar o método Ágil significa entregar o produto mais rápido**

Mito, utilizar o método ágil significa que será entregue pequenas versões do projeto constantemente, ao final de cada *Sprint* um novo incremento será disponibilizado para o cliente. O fato de entregar ao cliente as funcionalidades que vão ficando prontas dá a sensação de que os métodos ágeis são mais rápidos, quando na verdade o que é feito é entregar um produto de valor ao cliente, garantindo que aquela funcionalidade atenderá as necessidades do mesmo. (MJV, 2020)

- **Não utilizamos documentação nos métodos ágeis**

Mito, todo e qualquer projeto precisa ser documentado, só que é necessário entender qual o volume realmente necessário de documentação, assim que definido, a documentação se torna um forte aliado no desenvolvimento. (MJV, 2020)

- **O *Framework Scrum* resolve todos os problemas**

Mito, *Framework Scrum* é o modelo ágil mais conhecido atualmente, ele trabalha com *Sprints* (ciclos) curtas, o que facilita a detecção de erros, quando existentes, com mais agilidade, contudo, ele não soluciona todos os problemas, ele auxilia o time a detectar os problemas existentes. (MJV, 2020)

- **Quem utiliza o *Scrum* não pode utilizar o *Kanban***

Mito, tanto o *Kanban* quanto o *Scrum* podem ser utilizados em conjunto, o *Framework Scrum* já consolidado é ideal para projetos com início, com meio e final, já

o *Kanban*, por sua vez, proporciona uma maior visibilidade com os seus quadros e cartões. (MJV, 2020)

- **Não existe planejamento em métodos ágeis**

Mito, todo e qualquer projeto que venha a ser executado necessita de um planejamento, podemos dizer que é um dos princípios de qualquer projeto, acontece que, com as metodologias ágeis não se realiza um planejamento rico em detalhes do projeto por completo, tendo em vista que cada incremento entregue pode acarretar em alterações no próximo incremento. (MJV, 2020)

- **Métodos Ágeis são muito bagunçados, pois não possui um Gerente de Projetos**

Mito, de bagunçado as metodologias ágeis não tem em nada, pelo contrário, os times de desenvolvimento trabalham com tempo e objetivo bem definido inicialmente. Os times de desenvolvimento são auto gerenciáveis, embora tenha a figura de um líder, líder esse que tem a missão de facilitar a resolução de qualquer impedimento, fazendo com o que time consiga manter o foco integralmente no desenvolvimento. (MJV, 2020)

- **Não existe disciplina nas Metodologias Ágeis**

Mito, é de extrema importância que o time tenha muita disciplina, pois, para manter contato com o cliente constantemente, remover impedimentos, fazer o levantamento de riscos, levantamento de requisitos, garantir que os princípios básicos da metodologia sejam seguidos necessita de muita disciplina, caso contrário não obteríamos sucesso com as metodologias ágeis. (MJV, 2020)

- **Não existe flexibilidade nas metodologias ágeis**

Mito, a grande vantagem das metodologias ágeis é que elas proporcionam uma maior flexibilidade nos seus projetos, o contato frequente com o cliente possibilita uma

maior compreensão do que o cliente realmente necessita, fazendo com que o projeto, após ser finalizado, atenda todas as reais necessidades do cliente. Sendo assim, sempre que identificado que é necessário realizar alguma alteração ela é feita o mais breve possível. (MJV, 2020)

- **Requer muito retrabalho**

Mito, na verdade o uso das metodologias ágeis tende a reduzir o trabalho, pois com as entregas constantes conseguimos identificar com mais facilidade o surgimento de algum erro, podendo solucioná-lo com mais agilidade.

- **Metodologias ágeis não tem foco na resolução de impedimentos**

Mito, com a entrega frequente de funcionalidades de valor para o cliente conseguimos garantir uma maior assertividade, pois são realizados testes que garantem que as funcionalidades operem como o esperado. Contudo, dentro do *Scrum,* por exemplo, existe um líder, mais conhecido como facilitador, ele remove qualquer empecilho que possa prejudicar a equipe de desenvolvimento. Sendo assim, podemos garantir que o uso de metodologias ágeis faz com que o time tenha o foco integralmente em entregar um produto de qualidade ao cliente. (MJV, 2020)

- **Metodologias ágeis tem um custo elevado para sua adoção**

Mito, pois não se faz necessário adquirir ferramentas pagas para adotar uma metodologia ágil, considerando que o método ágil é uma forma de pensar, trabalharemos otimizando o tempo com reuniões bem definidas e acompanhando todo o processo do projeto, assim, o único investimento necessário é a dedicação dos integrantes do time. (MJV, 2020)

- **Metodologias ágeis não são auto gerenciáveis**

Mito, pois dentro das metodologias ágeis não existe um chefe para definir, por exemplo, o prazo para entregar um projeto, o que existe é um time, time esse que

estima o tempo necessário e escolhe a melhor forma de desenvolver cada projeto., o que os torna um time auto gerenciável. (MJV, 2020)

- **Dentro das metodologias ágeis a qualidade depende de um dos integrantes do time**

Mito, dentro das metodologias ágeis trabalhamos com times, assim não existe responsabilidade individual e sim coletiva, todos são responsáveis pelo projeto, quando um time estima um tempo para a entrega, é responsabilidade dele cumprir os prazos estimados, garantindo a entrega de um produto final de qualidade. (MJV, 2020)

- **A grande estrela dentro do time ágil é o projeto**

Mito, tendo em vista que são as pessoas que executam as funções necessárias não tem como não as considerar como as estrelas, sendo assim, dentro dos métodos ágeis as pessoas são mais importantes do que qualquer processo.

3 FERRAMENTAS QUE AUXILIAM AS METODOLOGIAS ÁGEIS

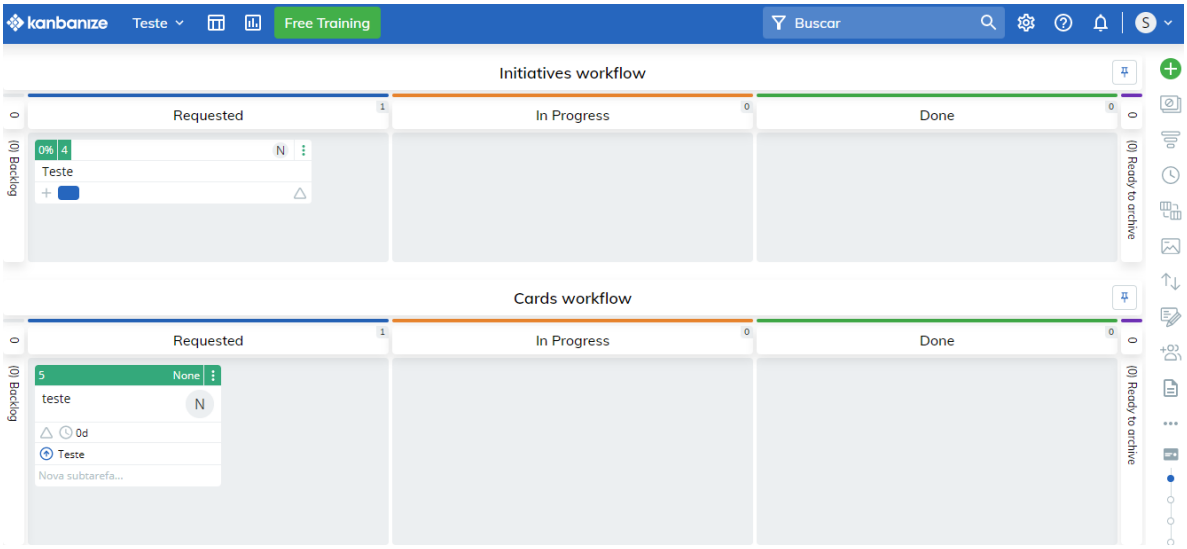
Atualmente, existem diversas ferramentas que possibilitam a utilização de metodologias ágeis de forma virtual, podendo substituir, por exemplo, um quadro físico do Kanban por um quadro virtual dentro de uma plataforma, fazendo com que todos os integrantes do time consigam visualizar todos os processos. As ferramentas são indispensáveis para times de desenvolvimento que exercem suas funções no *Home Office* (pessoas que trabalham de casa exercem o trabalho *home office*), por exemplo. Contudo, vamos apresentar agora algumas ferramentas utilizadas.

Kanbanize

O *software Kanbanzine* possibilita o gerenciamento do método *Kanban*, sua plataforma permite que sejam feitos o planejamento e a organização do projeto, além de possibilitar o acompanhamento de todo o projeto. O *software* é muito flexível, como apresentado na Figura 04, e o mesmo tem uma versão gratuita, o que facilita a adesão da plataforma por qualquer empresa. *Kanbazine* possui alguns recursos, sendo eles: (KANBANIZE, 2021)

- Possui uma linha do Tempo do Projeto;
- Possui quadros *Kanban*;
- Possibilita o acompanhamento de tempo;
- Possui automação do fluxo de trabalho;
- Permite ter previsão do projeto;
- Possui relatórios automatizados referente ao status do projeto;
- Permite o gerenciamento de dependência;
- Possui um poderoso módulo de análise.

Figura 04 – *Kanbanize*, ferramenta que auxilia o *Kanban*



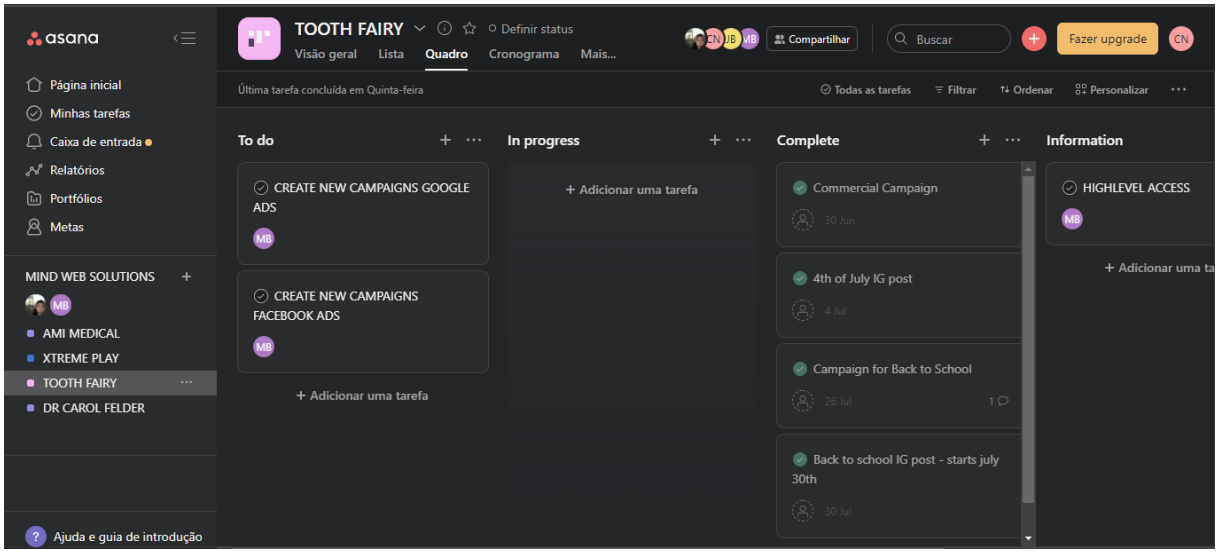
Fonte: kanbanize.com

Asana

Consiste em uma ferramenta totalmente flexível, muito utilizada para gerenciar o fluxo de trabalho, como apresentado na Figura 05. As atualizações feitas são visualizadas em tempo real, e possui a funcionalidade de atribuir tarefas a um grupo específico. O *Asana* possui alguns recursos, sendo eles: (KANBANIZE, 2021)

- Possui um *feed* de atividades;
- Os integrantes recebem por e-mail todas as atualizações de forma automáticas;
- Permite criar calendários e possibilita uma visualização personalizada;
- Possui uma lista de Minhas Tarefas;
- Permite acompanhar as tarefas e adicionar seguidores;
- Permite visualizar as tarefas e prioridades dos membros da equipe.

Figura 05 – Asana, ferramenta que auxilia o *Kanban* e o *Scrum*



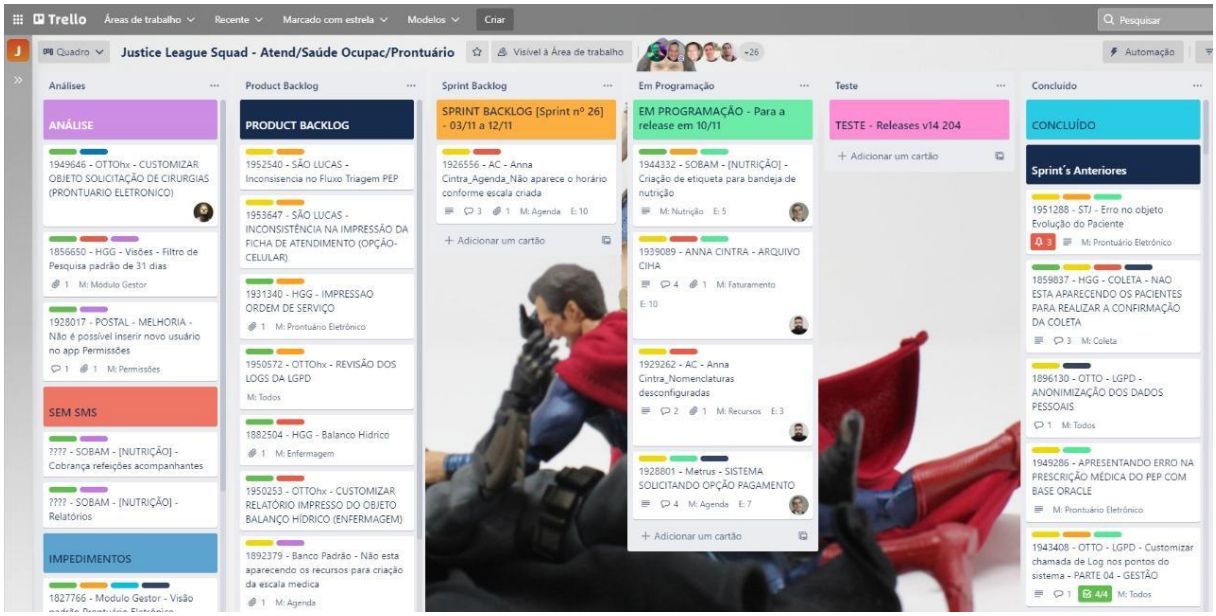
Fonte: kanbanize.com

Trello

Considerado de extrema facilidade de utilização, o *Trello* vem agradando muito os seus usuários. Os times podem personalizar o *Trello* de acordo com as suas necessidades, requisitos e processos de trabalho, como apresentado na Figura 06. O *Trello* integra mais de 100 ferramentas digitais e possui alguns recursos, sendo eles: (KANBANIZE, 2021)

- Permite a personalização das *Sprints*;
- Permite a criação de quadros *Kanban*;
- Possui registros de atividades;
- Possui uma automação integrada;
- Possui um painel de controle e uma visualização da Linha do Tempo;
- Permite a visualização do Calendário;
- Permite atribuir funções;
- Possui login com *Google Apps*.

Figura 06 – *Trello*, ferramenta que auxilia o *Scrum* e o *Kanban*

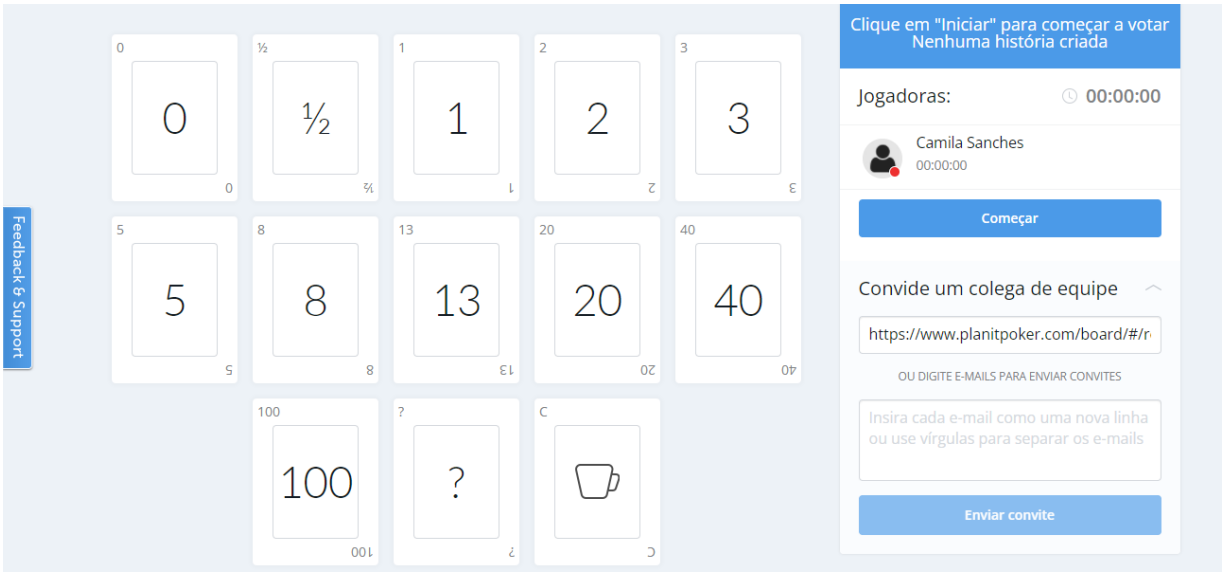


Fonte: www.kanbanize.com

PlanITPoker

Ferramenta muito utilizada para criar sessões de jogadas *Planning Poker*, ela não possui um limite de participantes por sessão. O site é responsivo e permite compartilhar o link da sessão para todos os interessados, como apresentado na Figura 07. Ao final, os resultados são exportados para o Excel. (BANDARRA, 2018)

Figura 07 – *PlanITPoker*, ferramenta que auxilia jogar o *Planning Poker*

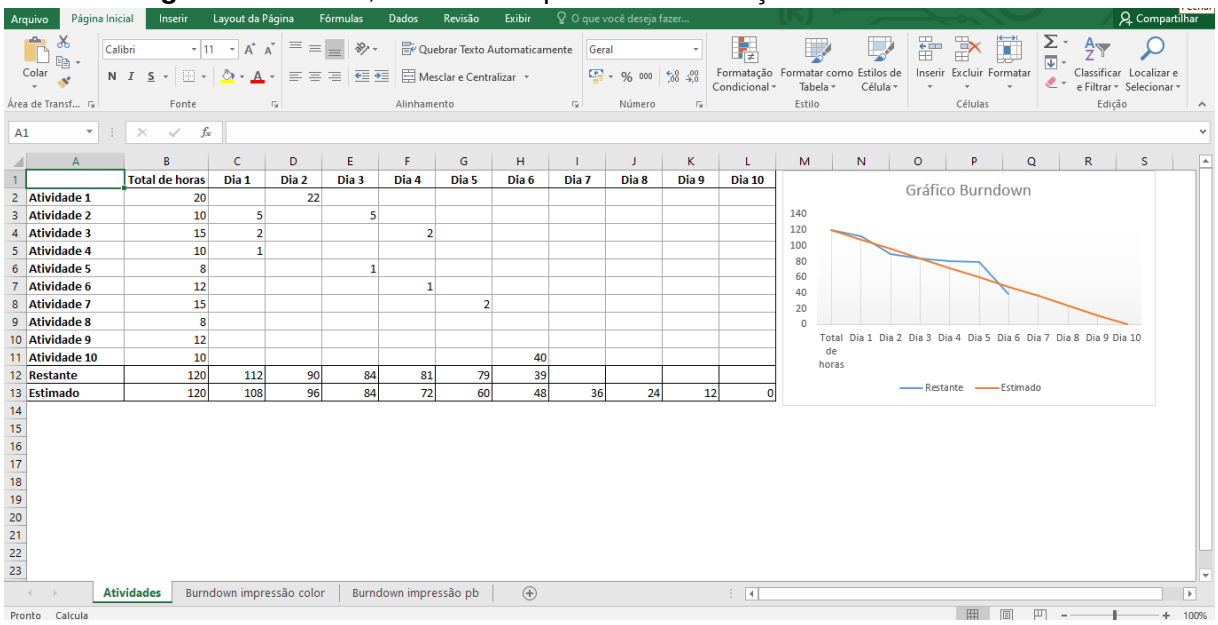


Fonte: www.planitpoker.com

Excel

O Excel permite a criação de Gráficos *Burndown*, permitindo uma visualização ampla e clara do processo do projeto, como apresentado na Figura 08.

Figura 08 – Excel, ferramenta que auxilia a criação do Gráfico *Burndown*



	Total de horas	Dia 1	Dia 2	Dia 3	Dia 4	Dia 5	Dia 6	Dia 7	Dia 8	Dia 9	Dia 10
Atividade 1	20		22								
Atividade 2	10	5		5							
Atividade 3	15	2			2						
Atividade 4	10	1									
Atividade 5	8			1							
Atividade 6	12				1						
Atividade 7	15					2					
Atividade 8	8										
Atividade 9	12										
Atividade 10	10						40				
Restante	120	112	90	84	81	79	39				
Estimado	120	108	96	84	72	60	48	36	24	12	0

Fonte: elaborado pela autora

SAIBA MAIS

O Scrum não é considerado um processo utilizado para construir um produto, e sim um framework que foca nos princípios elaborados no manifesto ágil.O Scrum permite que os times desenvolvem e gerenciam projetos complexos, e a partir das regras e artefatos, dividindo em etapas e entregas frequentes. (PAVANI, 2020)

REFERÊNCIAS

MJV. **Conheça 10 mitos e verdades sobre a agilidade.** 2020. Disponível em: https://www.mjvinnovation.com/pt-br/blog/10-mitos-e-verdades-sobre-agil/. Acesso em: 10 de outubro de 2021.

PAVANI. B. **ÁGIL É SÓ SER RÁPIDO? CONHEÇA.** 2020. Disponível em: https://blog.tivit.com/agil-e-so-ser-rapido-conheca-alguns-mitos-sobre-este-modelo. Acesso em: 10 de outubro de 2021.

KANBANZINE. **9 Melhores Ferramentas Kanban e Como Escolher o Melhor Software Kanban para a Sua Empresa.** Disponível em: https://kanbanize.com/pt/recursos-kanban/guia-do-software-kanban. Acesso em: 08 de outubro de 2021.

BANDARRA. M. **Ferramentas Planning Poker gratuitas. Agora não tem desculpa para escapar a estimativas.** 2018. Disponível em: https://onceyougoagile.com/pt/blog/2018/07/28/ferramentas-planning-poker-gratuitas-agora-nao-tem-desculpa-para-escapar-a-estimativas/. Acesso em: 13 de outubro de 2021.