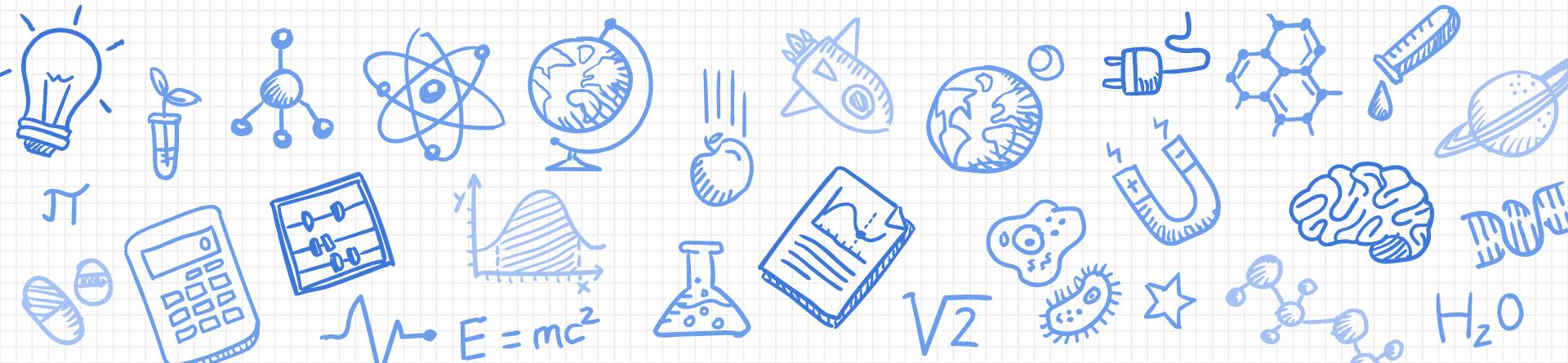


Metodologias Ágeis



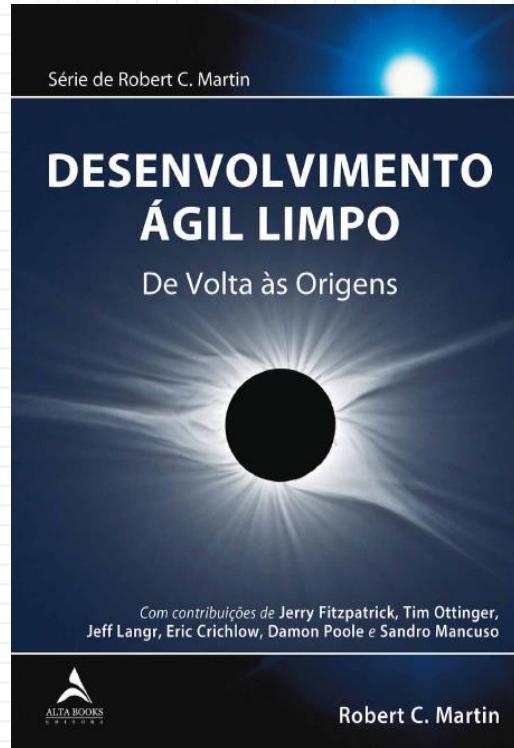


Olá!

Eu sou Willian Brito

- ❖ Desenvolvedor Full Stack na Msystem Software
- ❖ Formado em Analise e Desenvolvimento de Sistemas.
- ❖ Pós Graduado em Segurança Cibernética.
- ❖ Certificação SYCP (Solyd Certified Pentester) v2018

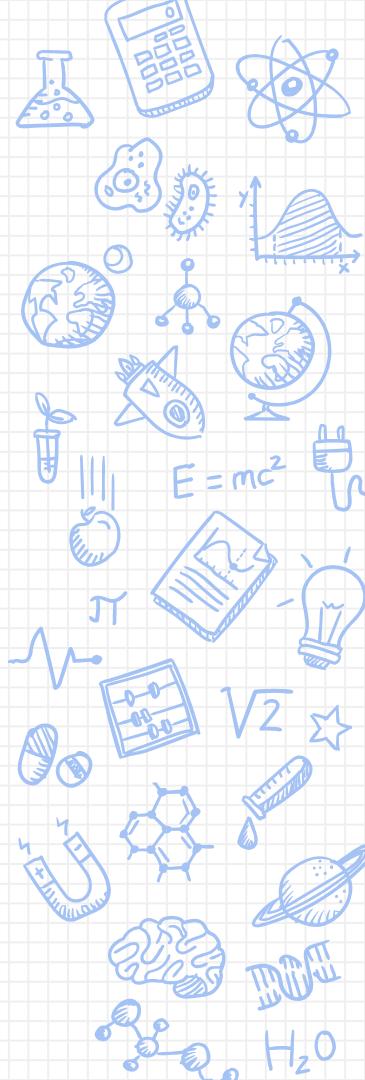
Este conteúdo é baseado nessas obras



Este conteúdo será focado no modelo de gestão de projetos da metodologia SCRUM e algumas práticas técnicas do eXtreme Programming (XP).

O que é Metodologia Ágil

Metodologias ágeis são conjuntos de **práticas** que proporcionam uma forma de **gerenciar projetos** mais **adaptável às mudanças**. Elas são estruturadas em **ciclos curtos** sendo que, a cada novo ciclo, é entregue um conjunto de funcionalidades pré-determinado. Portanto, as metodologias ágeis têm como principal restrição o tempo e são caracterizadas por produzirem **entregas rápidas e frequentes**.



O Surgimento

Desde seus primórdios, a indústria de desenvolvimento de software seguiu uma metodologia tradicional. Nela, temos as seguintes fases: levantamento e análise de requisitos, desenho da arquitetura, codificação, testes, implantação e manutenção. Embora tecnicamente correta, essa metodologia é vista como desnecessariamente rígida. Ao tornar o processo tão formal, ela impede que os clientes tenham o desenvolvimento na velocidade que necessitam.

Além disso, muitas empresas utilizavam e algumas ainda usam a metodologia Cascata, umas das mais tradicionais formas de se gerenciar projetos. Na abordagem surgida nos anos 1970, todas as etapas são seguidas de forma sequencial. Mas o modelo pode gerar muitos problemas de gestão, pois uma etapa só é iniciada quando a anterior for inteiramente concluída.

Com a maturação da indústria de software, problemas com um possível delay entre as necessidades do cliente e as entregas começaram a surgir, e a necessidade de encontrar novas soluções se tornou clara. Foi então criada a metodologia ágil, que tinha, como o nome indica, a função de agilizar o processo de desenvolvimento, principalmente no que diz respeito à utilização do software pelo cliente.

Ágil é entregar mais valor para o cliente

Assim, em 2001, um grupo de programadores lançou o Manifesto Ágil, pregando uma metodologia que tem como objetivo satisfazer os clientes entregando com rapidez e com maior frequência versões do software conforme as necessidades.

Ou seja, a partir de uma versão publicada, embora não absolutamente completa, pode-se evoluir o software de acordo com as necessidades do cliente e das demandas da sociedade. Do contrário, o produto final demoraria tanto para ficar pronto que poderia se tornar obsoleto.

Afinal, é melhor ter esse produto para entregar e com a possibilidade dele ser melhorado com o tempo do que passar todo o processo sem uma oferta, e depois ela ser ultrapassada pela de um concorrente.

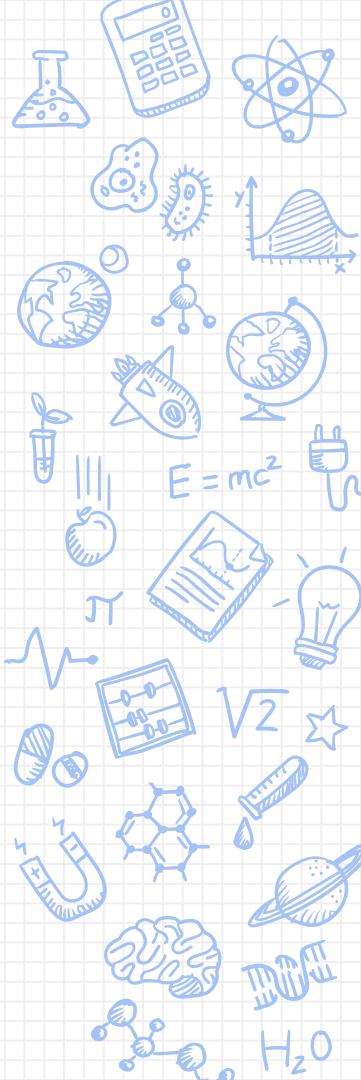
Manifesto Ágil

O **Manifesto Ágil** foi publicado nos dias **11 a 13 de fevereiro de 2001**, em **Snowbird - Utah**, como um trabalho de 17 profissionais experientes (consultores, desenvolvedores e líderes) da comunidade de software que já praticavam métodos ágeis como **XP, Crystal, DSDM, SCRUM, FDD e Pragmatic Programming** se reuniram para discutir alternativas aos processos e às práticas burocráticas utilizadas nas abordagens tradicionais de Engenharia de Software e Gerência de Projetos. Assim nasceu o Manifesto Ágil, destacando as diferenças em relação às abordagens tradicionais.

Durante a reunião, foram observados os pontos comuns de projetos que tiveram sucesso em suas metodologias e com base nesses pontos foi criado o Manifesto para Desenvolvimento Ágil de Software, no qual chamamos de Manifesto Ágil.

O Manifesto Ágil aborda valores que todos os profissionais ali reunidos acordaram em seguir e disseminar.

Valores do Manifesto Ágil



Indivíduos e interações mais que processos e ferramentas

Software em funcionamento mais que documentação abrangente

Colaboração com o cliente mais que negociação de contratos

Responder a mudanças mais que seguir um plano

Ou seja, mesmo havendo **valor** nos **itens à direita**,

valorizamos mais os itens à esquerda.

Valores

Indivíduos e interações mais que processos e ferramentas

Devemos entender que o desenvolvimento de software é uma atividade humana e que a qualidade da interação entre as pessoas pode resolver problemas crônicos de comunicação. Processos e Ferramentas são importantes, mas devem ser simples e uteis.

Software em funcionamento mais que documentação abrangente

O maior indicador de que sua equipe realmente construiu algo é software funcionando. Clientes querem é resultado e isso pode ser com software funcionando. Documentação também é importante, mas que seja somente o necessário e que agregue valor.

Colaboração com o cliente mais que negociação de contratos

Devemos atuar em conjunto com o cliente e não “contra” ele ou ele “contra” a gente. O que deve acontecer é colaboração, tomada de decisões em conjunto e trabalho em equipe, fazendo que todos sejam um só em busca de um objetivo.

Responder a mudanças mais que seguir um plano

Desenvolver software e produtos é um ambiente de alta incerteza e por isso não podemos nos debruçar em planos enormes e cheio de premissas. O que deve ser feito é aprender com as informações e feedbacks e adaptar o plano a todo momento.

Princípios do Manifesto Ágil

- Satisfação do cliente:** A maior prioridade está em satisfazer o cliente por meio da entrega adiantada e contínua de software de valor.
- Mudança em favor da vantagem competitiva:** Mudanças de requisitos são bem-vindas, mesmo em fases tardias do desenvolvimento.
- Prazos curtos:** Entregar software em funcionamento com frequência, desde a cada duas semanas até a cada dois meses, com uma preferência por prazos mais curtos.
- Trabalho em conjunto:** Tanto pessoas relacionadas a negócios como desenvolvedores devem trabalhar em conjunto, diariamente, durante todo o curso do projeto.
- Ambientação e suporte:** Para construir projetos ao redor de indivíduos motivados, é preciso dar a eles o ambiente e o suporte necessários, confiando que farão seu trabalho.
- Falar na cara:** O método mais eficiente de transmitir informações tanto externas como internas para um time de desenvolvimento é por meio de uma conversa cara a cara.
- Funcionalidade:** Um software funcional é a medida primária de progresso.
- Ambiente de sustentabilidade:** Processos ágeis promovem um ambiente sustentável, com patrocinadores, desenvolvedores e usuários sendo capazes de manter passos constantes.
- Padrões altos de tecnologia e design:** A contínua atenção à excelência técnica e ao bom design aumenta a agilidade.
- Simplicidade:** Fazer algo simples é dominar a arte de maximizar a quantidade de trabalho que não precisou ser feito.
- Autonomia:** As melhores arquiteturas, os requisitos e os designs emergem de times auto organizáveis.
- Reflexões para otimizações:** Em intervalos regulares, o time reflete em como ficar mais efetivo, então, se ajustam e otimizam seu comportamento de acordo.

Fundadores do Manifesto Ágil

Eis a lista de nomes da “**Aliança Ágil**”, sim, esse foi o termo que eles usaram para dirigir a si mesmos enquanto assinantes do manifesto:

Robert C. Martin, o “Uncle Bob”

Ken Schwaber, co-criador do Scrum.

Jeff Sutherland, o inventor do Scrum.

Kent Back, co-criador da eXtreme Programming (XP).

Ron Jeffries, co-criador da eXtreme Programming (XP).

Mike Beedle, CEO, Enterprise Scrum Inc.

Arie van Bennekum, da Integrated Agile.

Alistair Cockburn, criador da Metodologia Ágil Crystal.

Ward Cunningham, desenvolveu o primeiro Wiki.

Martin Fowler, Chief Scientist da ThoughtWorks.

James Grenning, autor da técnica de estimativa Planning Poker.

Jim Highsmith, criador do Adaptive Software Development (ASD).

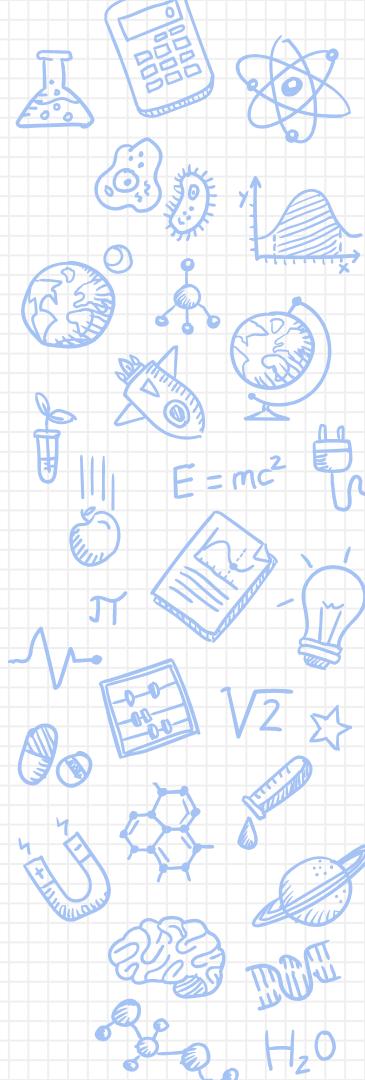
Andrew Hunt, co-autor de O Programador Pragmático.

Jon Kern, atuante até os dias de hoje em assuntos de agilidade.

Brian Marick, cientista da computação e autor de vários livros sobre programação.

Steve Mellor, cientista da computação e um dos idealizadores da Análise de Sistema Orientado a Objetos (OOA).

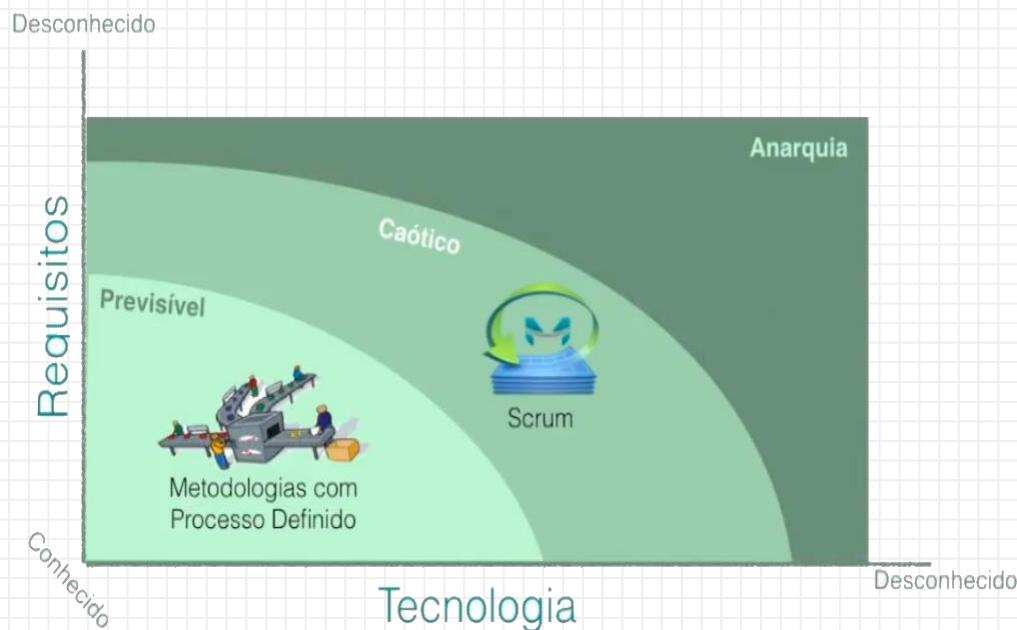
Dave Thomas, programador e co-autor de O Programador Pragmático.



Como saber quando devo utilizar Metodologias Ágeis?

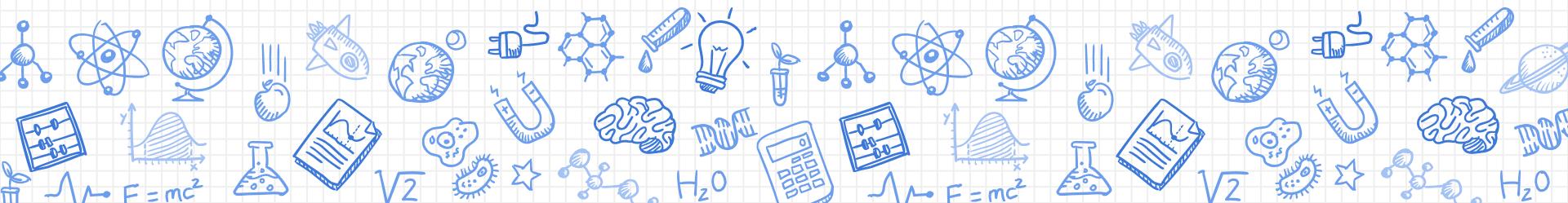
Este gráfico mostra todas as relações possíveis entre conhecimento dos requisitos e das tecnologias envolvidas no projeto que será justificável a utilização de Metodologias Ágeis.

- ❑ Quanto maior o conhecimento dos requisitos e das tecnologia utilizadas, mais **previsíveis** são as atividades necessárias para realizar o projeto, então quando tem-se domínio da tecnologia e um bom conhecimento dos requisitos podemos, utilizar metodologias com processos bem definidos, como por exemplo metodologias em cascata.
- ❑ Agora, conforme conhecemos menos os requisitos e a tecnologia fica mais complexa o que dificulta o domínio da mesma, o cenário é **caótico**. É nesse cenário caótico que as metodologias ágeis é melhor aplicada.
- ❑ Entretanto, se não há conhecimento dos requisitos e tão pouco a tecnologia, temos uma total **anarquia**. Aqui é impossível realizar o projeto.



SCRUM

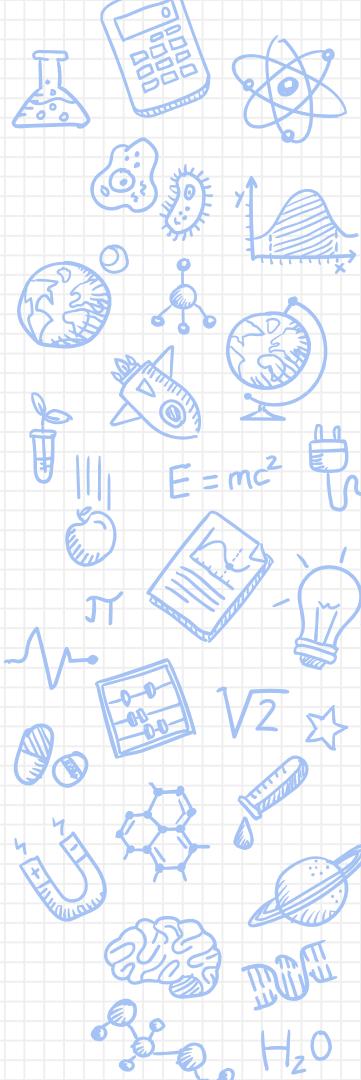
A arte de fazer o dobro do trabalho
na metade do tempo



O que é o SCRUM

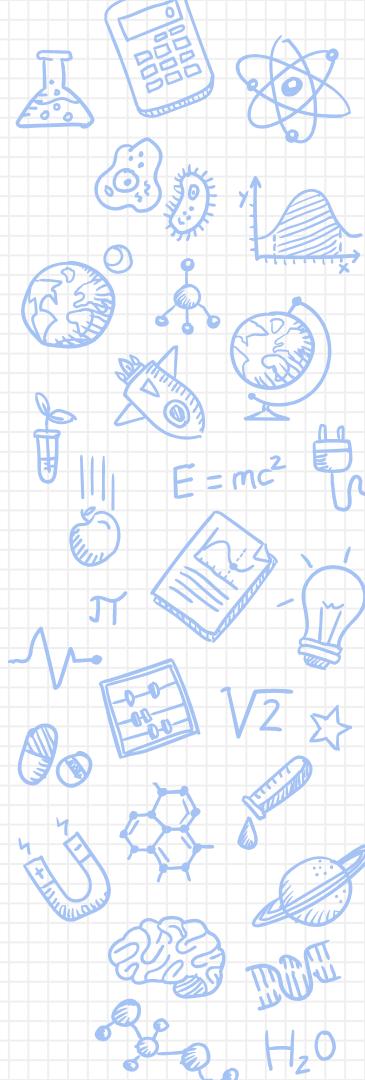
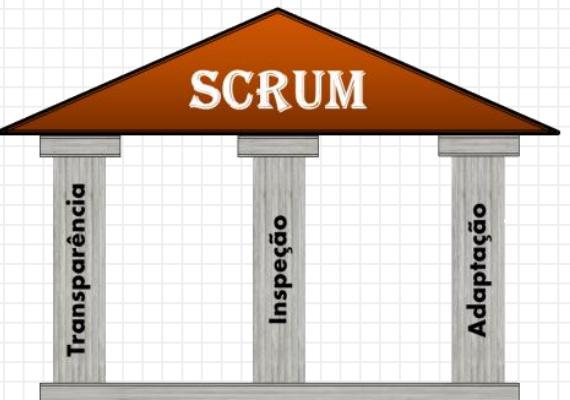
O Scrum é um framework, ou seja, um modelo a ser seguido, mas que pode (e deve) ser customizado conforme as necessidades. O Scrum é composto por:

- Pilares** (Transparência, Inspeção e Adaptação)
- Valores** (Comprometimento, Coragem, Foco, Abertura e Respeito)
- Papéis** (Scrum Master, Product Owner e Time de Desenvolvimento)
- Eventos** (Sprint, Planejamento da Sprint, Reuniões Diárias, Revisão da Sprint e Retrospectiva da Sprint)
- Artefatos** (Backlog do Produto, Backlog da Sprint e Incremento)



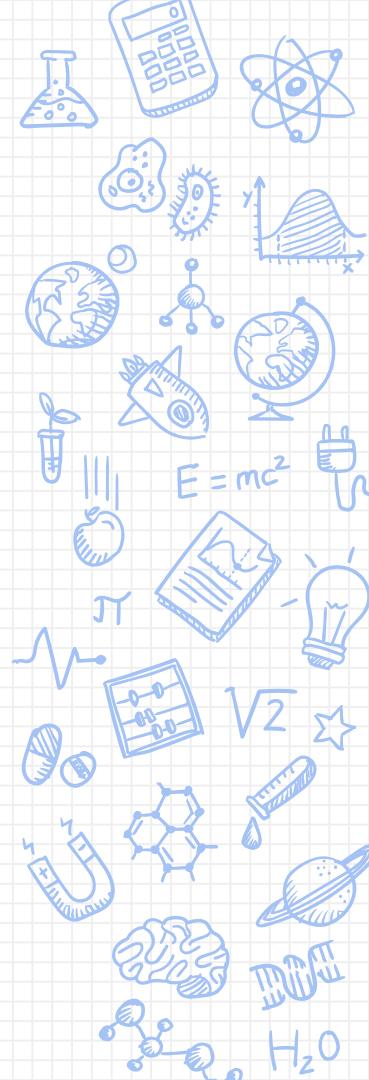
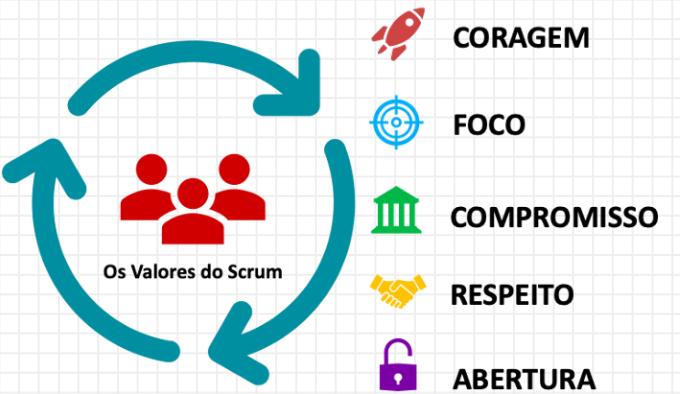
Pilares

- ❑ **Transparência:** A transparência requer que os aspectos significativos do processo estejam visíveis aos responsáveis pelo resultado.
- ❑ **Inspeção:** A importância de inspecionar o progresso em direção ao objetivo da Sprint serve para detectar variações indesejadas.
- ❑ **Adaptação:** O processo ou material sendo produzido deve ser ajustado, caso haja falhas encontradas na inspeção dos projetos.



Valores

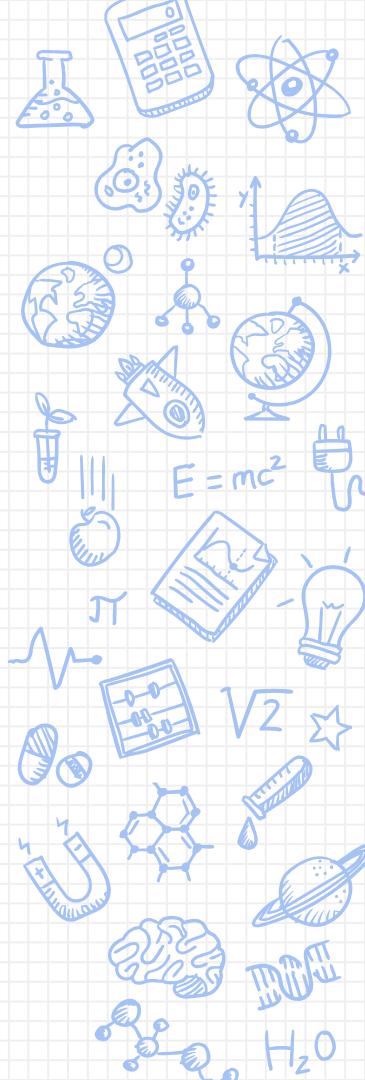
O sucesso no uso do SCRUM depende das pessoas se tornarem mais proficientes na vivência destes 5 valores **comprometimento, coragem, foco, abertura e respeito**. Quando incorporados e vividos pelo time, os pilares de transparência, inspeção e adaptação constroem a confiança para Todos.



Papéis

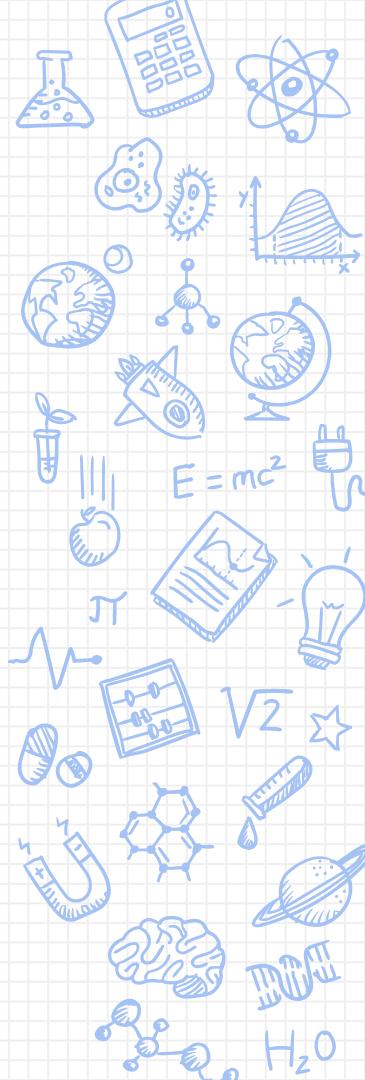
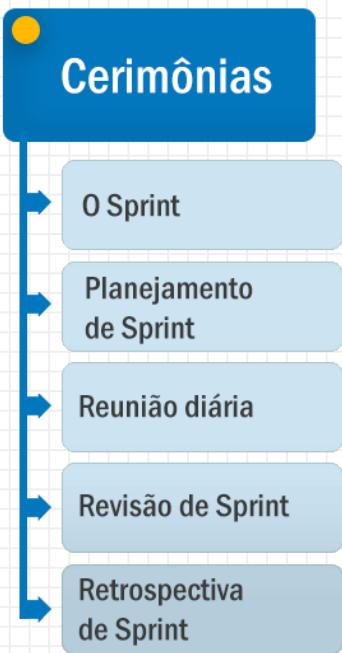
- ❑ **Product Owner:** O Product Owner possui a visão do produto e do retorno que o projeto trará para a empresa e para os envolvidos. Dessa forma sua missão é cuidar do Product Backlog, planejar releases, priorizar requisitos e passar ao time uma visão clara sobre os objetivos do projeto.
- ❑ **Scrum Master:** O Scrum Master exerce um papel de liderança no processo, mas ele não centraliza decisões, seu papel é guiar o time e a empresa para a auto-gestão. O papel de Scrum Master não possui autoridade alguma perante o P.O ou o Time. Assim sendo as responsabilidades do Scrum Master são manter o foco no processo, remover impedimentos da equipe e auxiliar na comunicação entre equipe e P.O.
- ❑ **Developer Team:** O time é uma equipe multidisciplinar que realizará a implementação do produto / projeto. O time deve manter a auto-gestão de suas atividades. Devem ser comprometidos e responsáveis em realizar o trabalho necessário para atingir a meta da Sprint.

SCRUM TIME



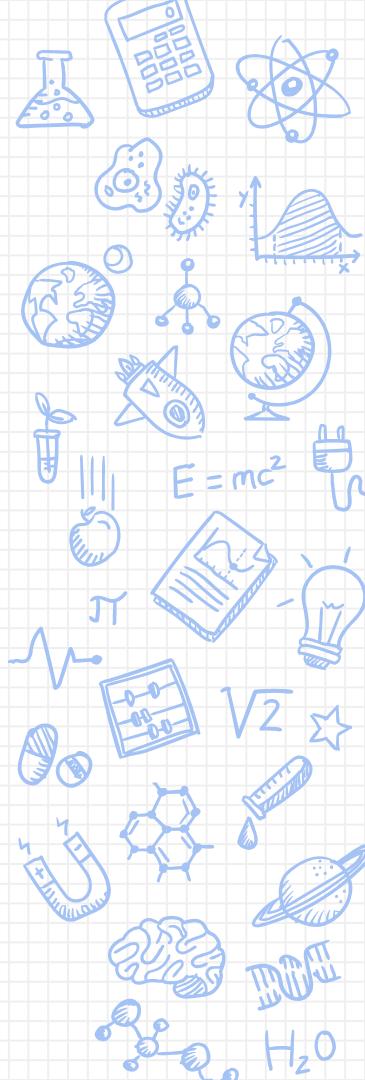
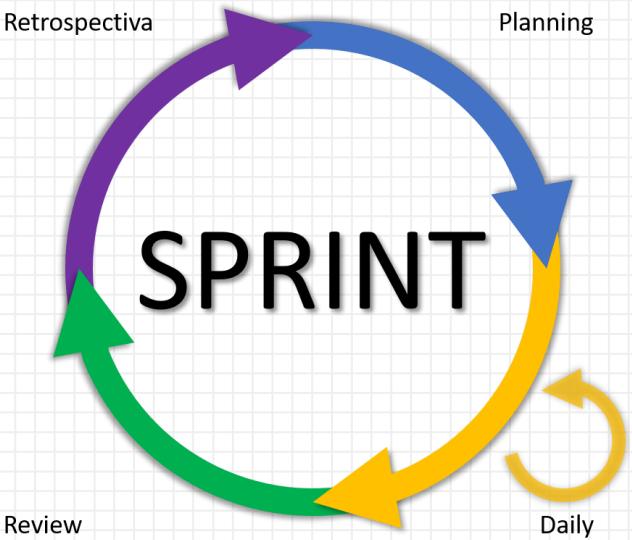
Eventos

Os eventos usados no SCRUM servem como oportunidade para que os três pilares (Transparência, Inspecção e Adaptação) sejam aplicados com sucesso. São eles Sprint, Planejamento da Sprint, Reuniões Diárias, Revisão da Sprint e Retrospectiva da Sprint.



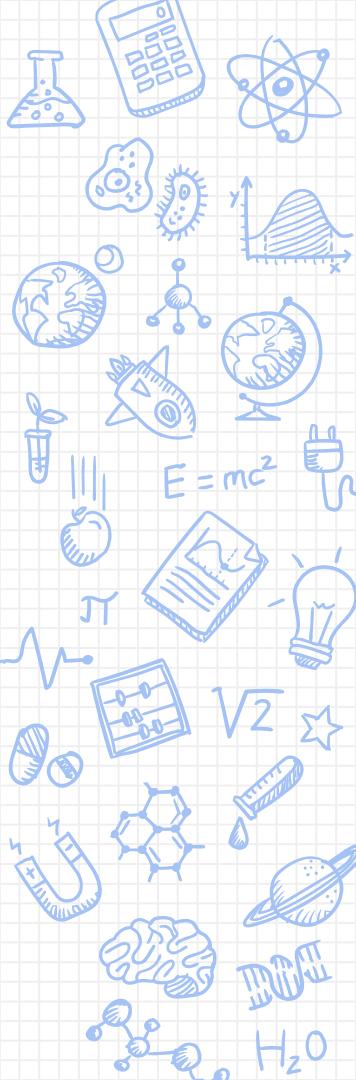
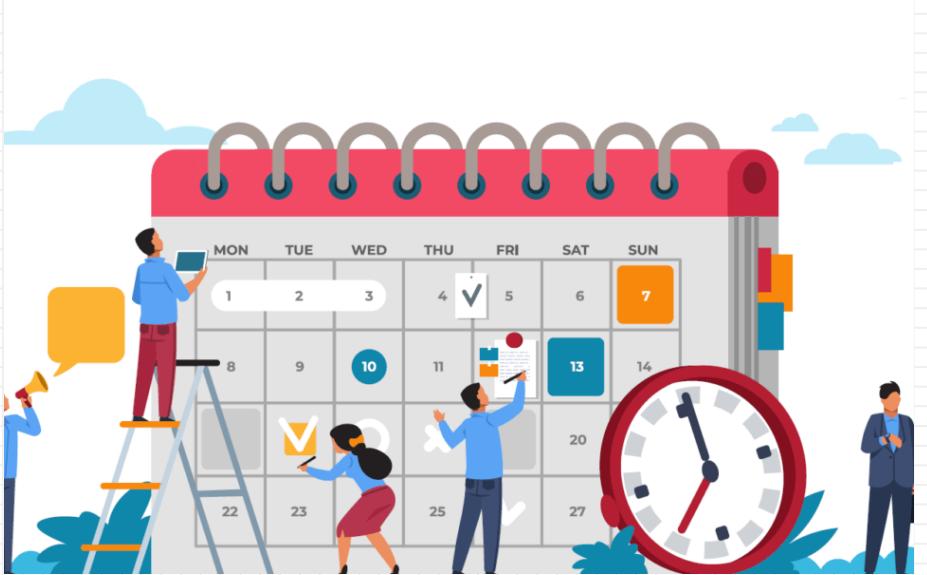
Sprint

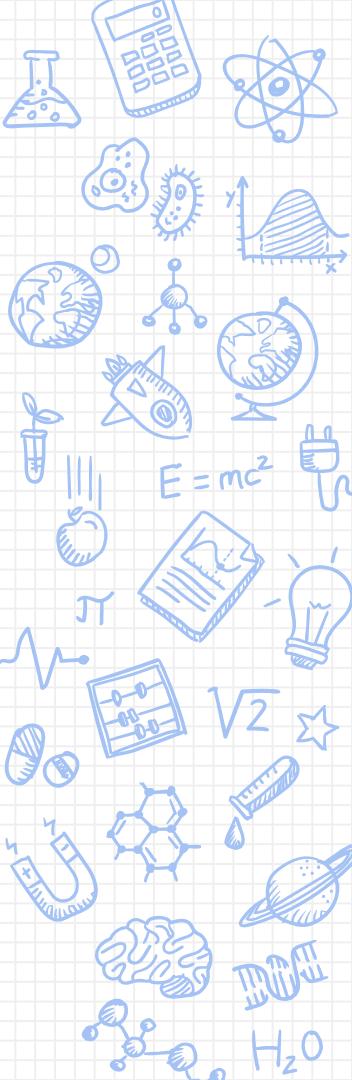
A Sprint possui um time boxed de um mês ou menos, durante o qual um “incremento de produto potencialmente liberável é criado uma nova Sprint inicia imediatamente após a conclusão da Sprint anterior. A meta da Sprint é um objetivo definido para a Sprint que pode ser satisfeita através da implementação do Backlog do Produto durante a Sprint não são feitas mudanças que possam por em perigo o objetivo da Sprint as metas de qualidade não diminuem e, o escopo pode ser clarificado e renegociado entre o Product Owner e o Time de Desenvolvimento.



Planejamento da Sprint

Este plano é criado com o trabalho colaborativo de todo o Time Scrum. O Planejamento da Sprint é um time boxed com, no máximo, 8h para uma Sprint de um mês de duração. O Scrum Master garante que o evento ocorra e que os participantes entendam seu propósito.

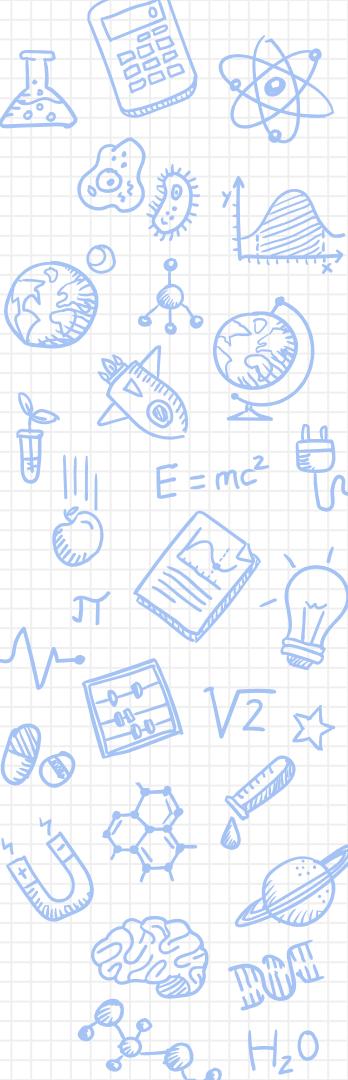




Reunião Diária

Evento time boxed de 15 minutos para o Time de Desenvolvimento, realizada em todos os dias da Sprint. Nela o Time de Desenvolvimento planeja o trabalho para as próximas 24 horas, isso otimiza a colaboração e a performance do time e é mantida no mesmo horário e local todo dia para reduzir a complexidade

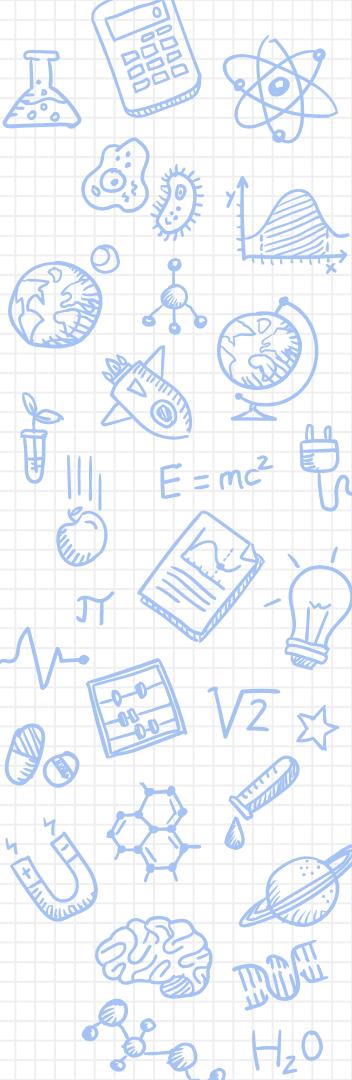




Revisão da Sprint

Realizada no final da Sprint para inspecionar o incremento e adaptar o Backlog do Produto, se necessário. Durante a Revisão da Sprint, o Time Scrum e as partes interessadas colaboram sobre o que foi feito na Sprint esta é uma reunião de, no máximo, 4 h de duração para uma Sprint de um mês.





Retrospectiva da Sprint

Ocorre depois da Revisão da Sprint e antes do planejamento da próxima Sprint. Esta é uma reunião de no máximo três horas para uma Sprint de um mês. É uma oportunidade para o Time Scrum inspecionar a si próprio e criar um plano para melhorias a serem aplicadas na próxima Sprint.



Fluxo da Sprint

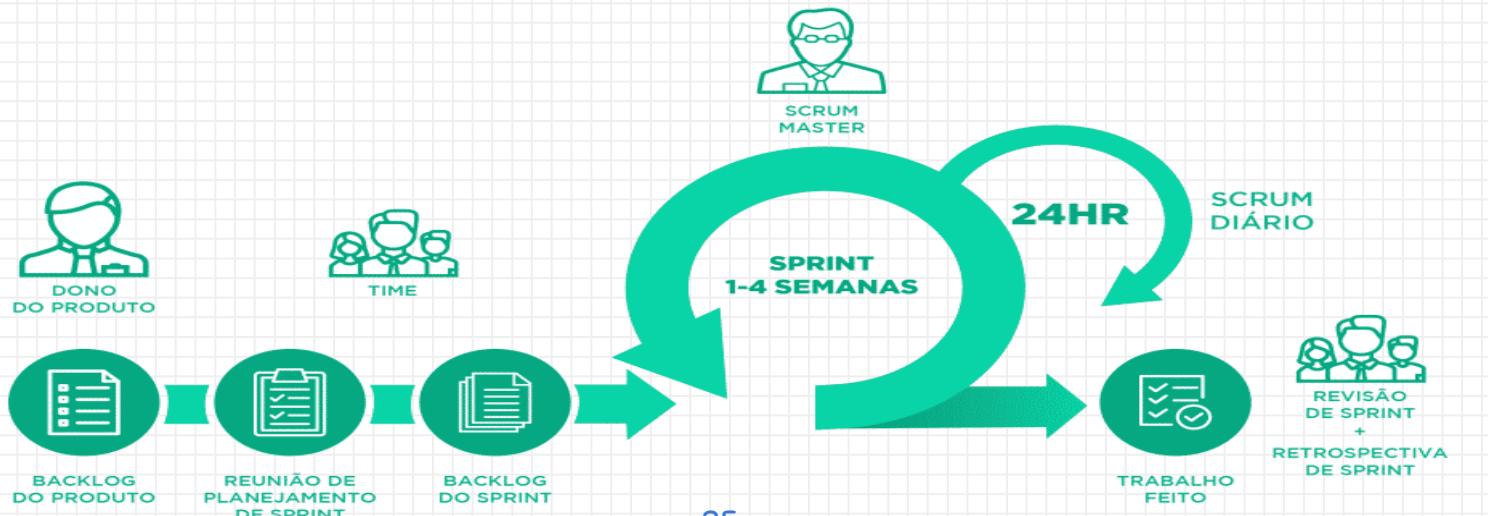
1- Planejamento de Sprints: Sprints são ciclos rápidos de tempo pré determinado que devem gerar um produto ao final. Todo o trabalho a ser realizado durante o Sprint vai ser planejado nesse evento.

2- Reunião Diária: A reunião diária é uma reunião de 15 minutos na qual o time deve responder três principais perguntas:

1. O que você fez ontem ?
2. O que você vai fazer hoje ?
3. Que obstáculos a equipe está enfrentando?

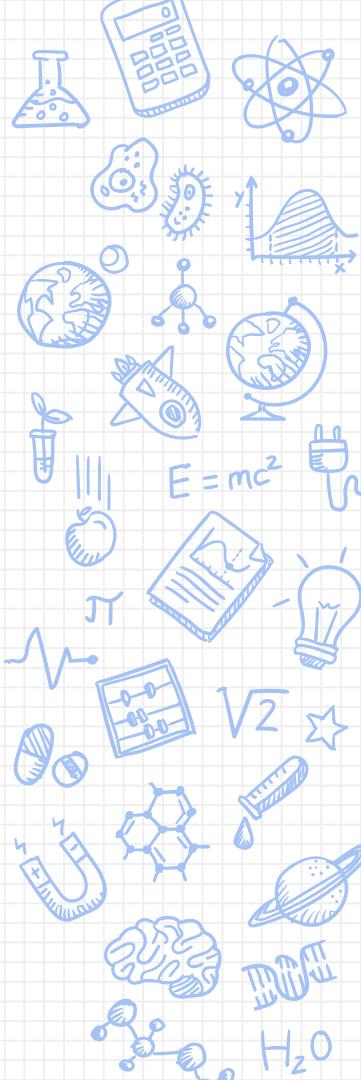
3- Revisão do Sprint: Acontece no final de cada Sprint para inspecionar o produto gerado até então e adaptar o prazo do projeto, caso necessário.

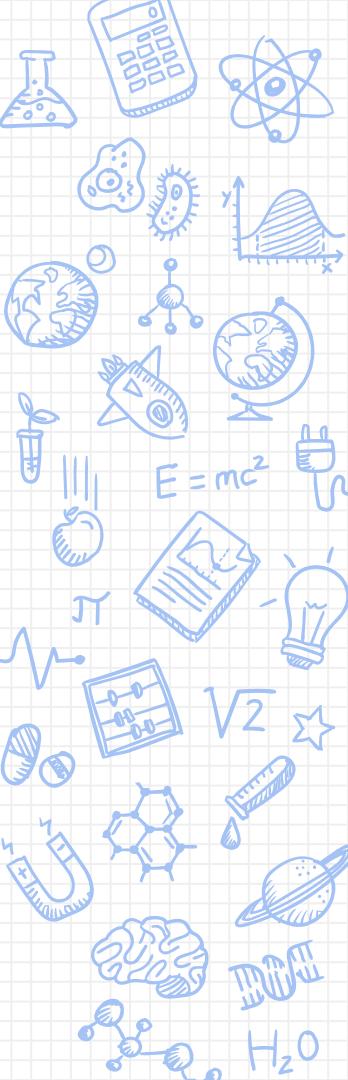
4- Retrospectiva do Sprint: É uma oportunidade para o time avaliar a si mesmo e construir um plano de melhorias para o próximo Sprint.



Artefatos

Os artefatos do Scrum têm a função de fornecer transparência ao processo, assegurando que todas as informações estão sendo transmitidas às equipes de desenvolvimento para promover sucesso completo na entrega do produto.

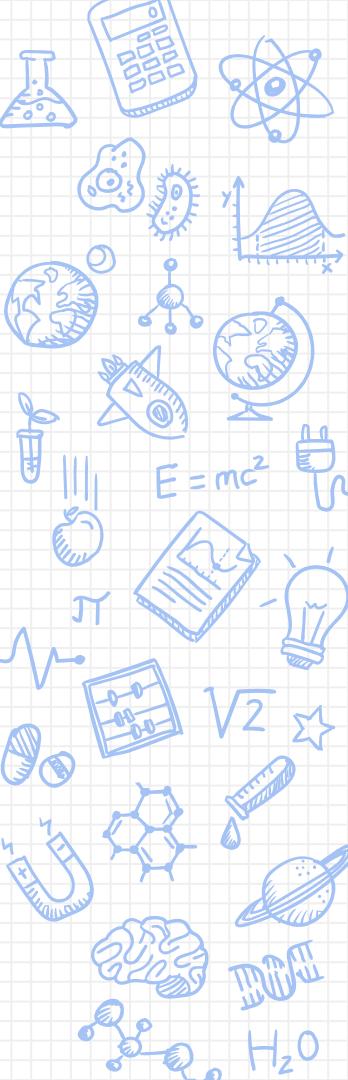




Backlog do Produto

Na metodologia ágil Scrum, a backlog do produto é uma lista ordenada de tudo o que é necessário para o produto. Deve ser a única origem para qualquer solicitação de mudança no produto ou sistema. Pode-se dizer que uma backlog do produto nunca está pronta, porque à medida que é estabelecido um desenvolvimento inicial a partir das informações primordiais, essa lista de produtos pendentes vai evoluindo para que o produto seja mais competitivo, útil e apropriado.

A lista de produtos pendentes vai ter todos os recursos, requisitos, funções, correções e aprimoramentos para as versões futuras. É possível dizer que à medida que um produto ou serviço é usado vai ganhar valor com o feedback do mercado, ou seja, a backlog do produto é dinâmica, está sempre em construção. Por isso, é realizado o refinamento do backlog do produto, que vai agregar mais detalhes e estimativas ao produto. Esse trabalho é monitorado a cada revisão da Sprint, junto com o Product Owner e as partes interessadas.

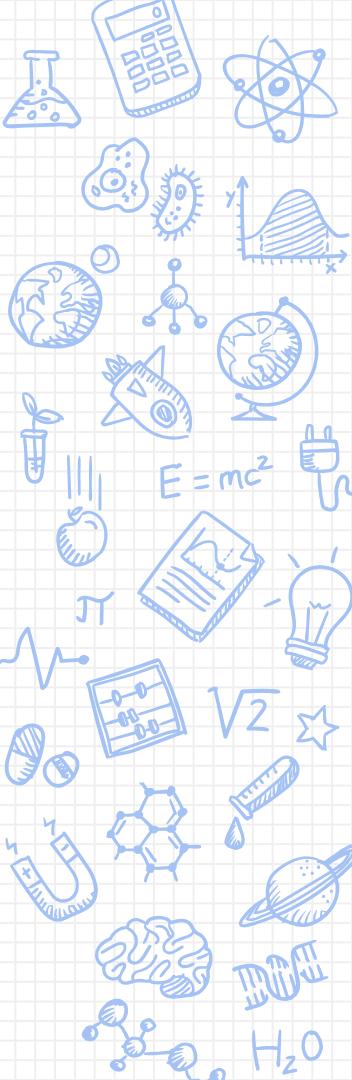


Backlog da Sprint

A Sprint, que são intervalos de tempos definidos, é o evento principal do Scrum, todas as atividades ocorrem durante esse período. No Scrum, cada Sprint vai definir uma entrega e o processo da próxima será sempre melhorado. O início de cada Sprint acontece quando é realizada o Planejamento da Sprint, e o fim será quando realizar a Retrospectiva da Sprint.

Portanto, o backlog da Sprint, que é o conjunto de itens do backlog do produto selecionados para o Sprint, será a previsão do time de desenvolvimento a respeito de qual funcionalidade estará neste próximo incremento. Neste artefato, pelo menos uma melhoria de processo de alta prioridade é incluída na Sprint atual a partir do que foi identificado na Retrospectiva anterior.

À medida que a tarefa é executada e concluída, o trabalho restante é atualizado e melhorado. Caso alguns elementos na Sprint anterior sejam considerados irrelevantes para o próximo processo, eles são removidos.



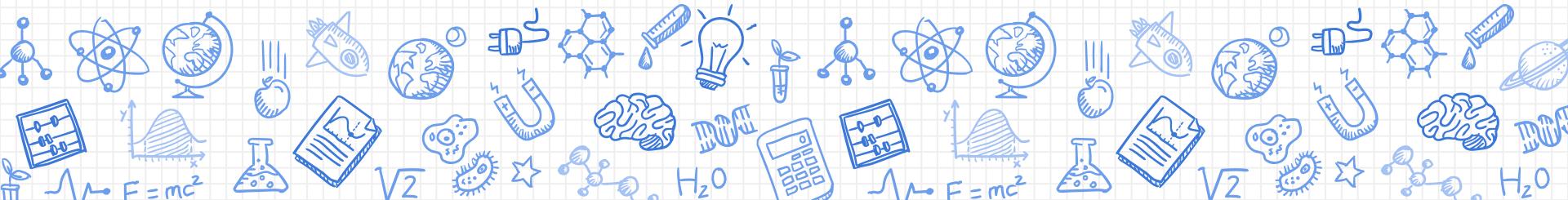
Incremento

O Incremento é um artefato oriundo do resultado dos esforços do Time de Desenvolvimento durante a Sprint. Ele geralmente é composto por software com potencial de ser liberado para produção, situação na qual o time terá gerado valor real para o cliente, tal qual como descrito no Manifesto Ágil.

A avaliação do Incremento ocorre durante a Sprint Review, a fase de Inspeção do desenvolvimento da sprint, sendo o artefato que fortalece com a experiência do Scrum, onde apenas resultados obtidos pelo time podem ser tomados como base para julgamentos futuros.

Implementando o SCRUM

Aqui está um resumo sobre a melhor forma de começar um projeto Scrum. Esta é uma descrição resumida do processo, mas deve ser suficiente para você começar.



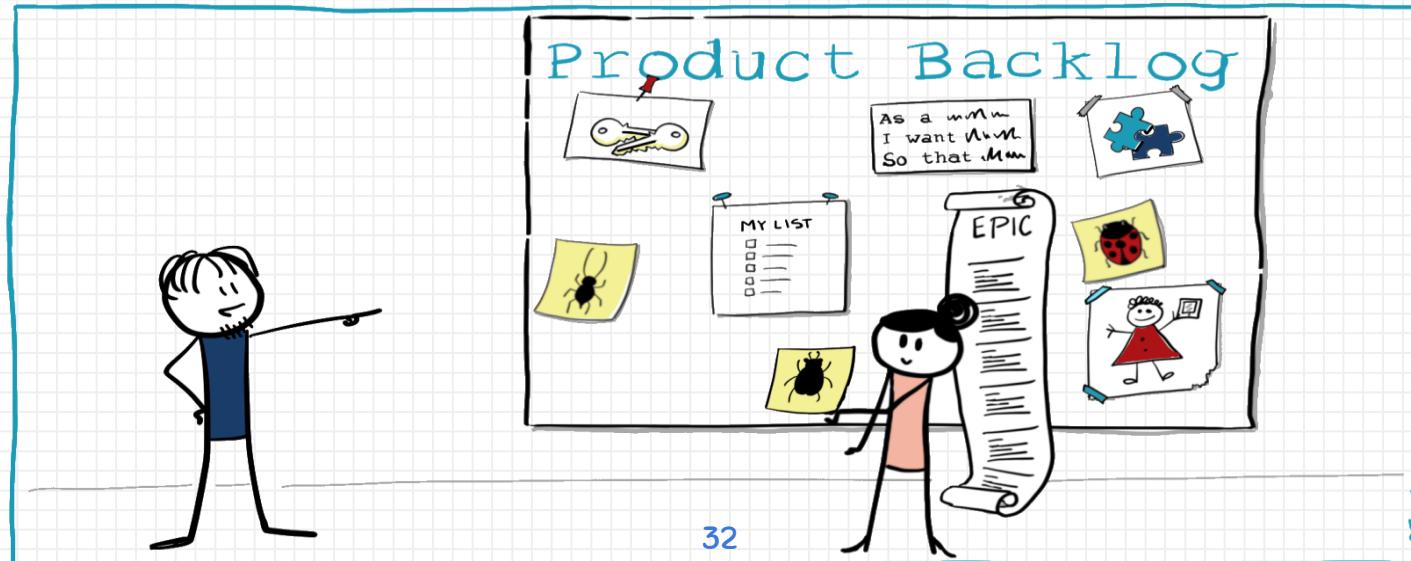
Escolha o Time Scrum

- 1. Escolha um Dono do Produto:** Essa pessoa é a responsável pela visão do que você vai fazer ou conseguir. Ela leva em consideração os riscos e os benefícios, o que é possível, o que pode ser feito e o que desperta a paixão na equipe.
- 2. Escolha uma equipe:** Quem serão as pessoas que realmente trabalharão no projeto? Essa equipe precisa ter todas as habilidades necessárias para pegar a visão do Dono do Produto e transformá-la em realidade. As equipes devem ser pequenas; entre três e nove pessoas.
- 3. Escolha um Mestre Scrum:** Essa pessoa vai orientar o restante da equipe em relação à estrutura do Scrum, além de ajudar a eliminar qualquer obstáculo que os esteja deixando mais lentos.



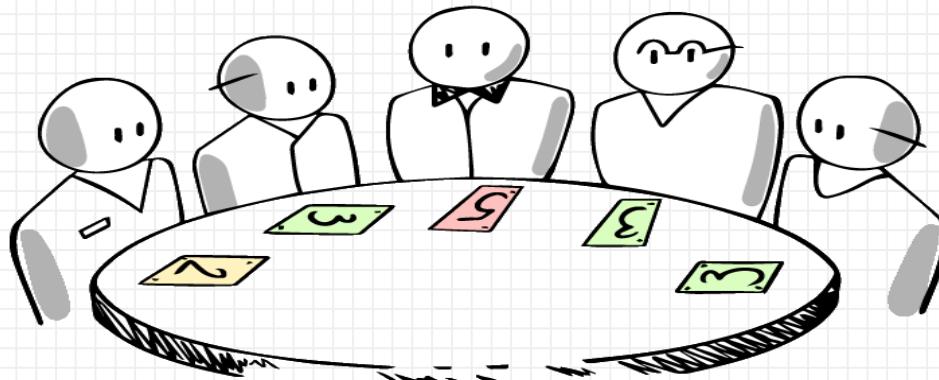
Crie o Backlog do Produto

4. Crie e priorize uma lista de Pendências do Produto: Trata-se de uma lista detalhada de tudo que precisa ser feito ou construído para transformar a visão em realidade. Essas Pendências existem e evoluem durante o desenvolvimento do produto; elas são o mapa dele. Em qualquer fase do projeto, são a única e definitiva visão de “tudo que precisa ser feito pela equipe a qualquer momento, em ordem de prioridade”. Só existe uma lista de Pendências; isso significa que o Dono do Produto precisa tomar decisões em relação às prioridades durante todo o processo; ele deve consultar todos os stakeholders e a equipe para se certificar de que elas representam tanto o que as pessoas querem, quanto o que pode ser construído.



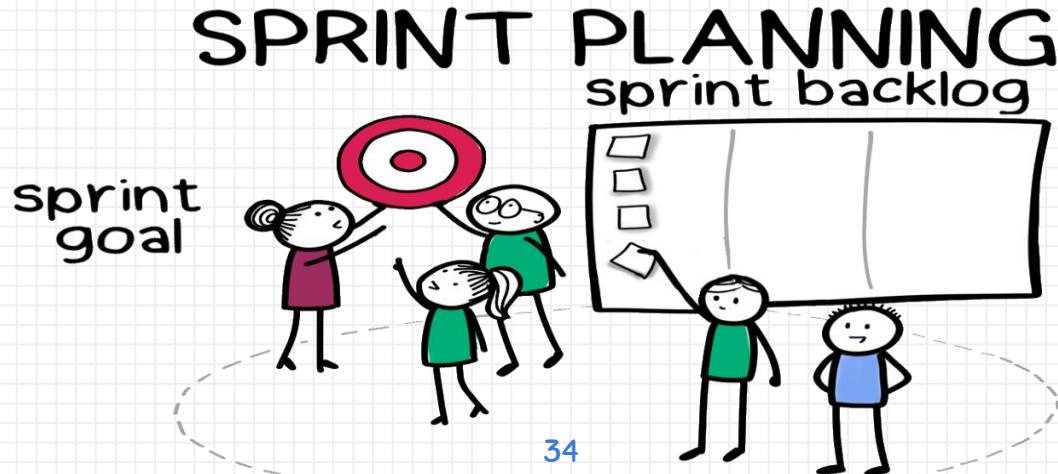
Pontue as Histórias de Usuarios

5. Aperfeiçoe e faça estimativas para as Pendências do Produto: É crucial que as pessoas que irão realmente concluir os itens da lista façam as estimativas de quanto esforço eles exigirão. A equipe deve olhar para cada item das Pendências e ver se aquilo é factível. Existem informações suficientes para concluir-lo? Ele é pequeno o suficiente para ser estimado? Existe uma definição de “Feito”? Ele cria valor visível? Cada item deve poder ser mostrado, demonstrado e, esperançosamente, ser enviado. Não estime as Pendências em horas, porque as pessoas são péssimas nesse tipo de previsão. Faça isso usando uma classificação relativa por tamanho: Pequeno, Médio ou Grande. Ou, melhor ainda, use a sequência de Fibonacci e faça estimativas de pontos para cada item: 1, 2, 3, 5, 8, 13, 21 etc.



Cerimônia: Planejamento da Sprint

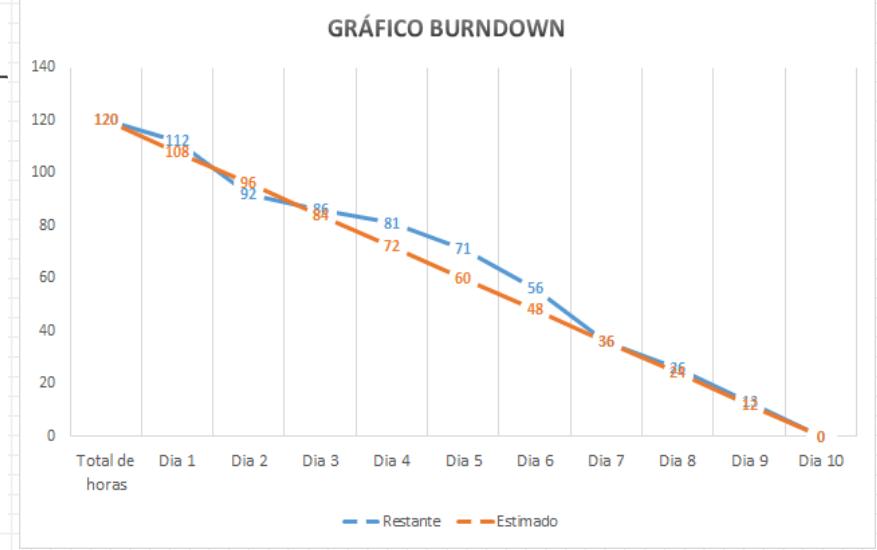
6. Planejamento do Sprint. Esta é a primeira das reuniões Scrum: A equipe, o Scrum Master e o Product Owner se reúnem para planejar o Sprint, que sempre tem uma duração definida de tempo menor que um mês. A maioria das pessoas define Sprints de uma ou de duas semanas. As equipes olham para as tarefas no topo das Pendências e estimam o quanto podem fazer naquele Sprint. Se a equipe já está trabalhando a alguns Sprints, ela deve pegar tarefas que totalizem o mesmo número de pontos do Sprint anterior. Esse número é conhecido como a Velocidade da equipe. O Mestre Scrum e a equipe devem tentar aumentar o número de pontos a cada Sprint. Essa é outra chance para a equipe e o Dono do Produto se certificarem que todos entendem como os itens vão satisfazer a visão. Além disso, durante essa reunião todos devem concordar com um Objetivo do Sprint. Um dos pilares do Scrum é que, uma vez que a equipe se comprometeu com o que acredita ser capaz de fazer em um Sprint, é isso. Ele não pode ser mudado, nada pode ser acrescentado. A equipe deve trabalhar de forma autônoma durante o Sprint para concluir o que foi previsto.



Quadro Scrum

7. Torne o trabalho visível: O melhor jeito para se fazer isso no Scrum é criar um Quadro Scrum com três colunas: A Fazer, Fazendo, Feito. post-its representam os itens que precisam ser concluídos e a equipe os move pelo Quadro Scrum à medida que forem concluídos, um a um. Outro modo de tornar o trabalho visível é criar um Gráfico de Burn-Down. Um eixo é o número de pontos que a equipe definiu para o Sprint, e o outro é o número de dias. Todos os dias, o Mestre Scrum soma o número de pontos concluídos e os marca no gráfico. O ideal é que haja uma ladeira descendo pelo gráfico até chegar ao zero no último dia do Sprint.

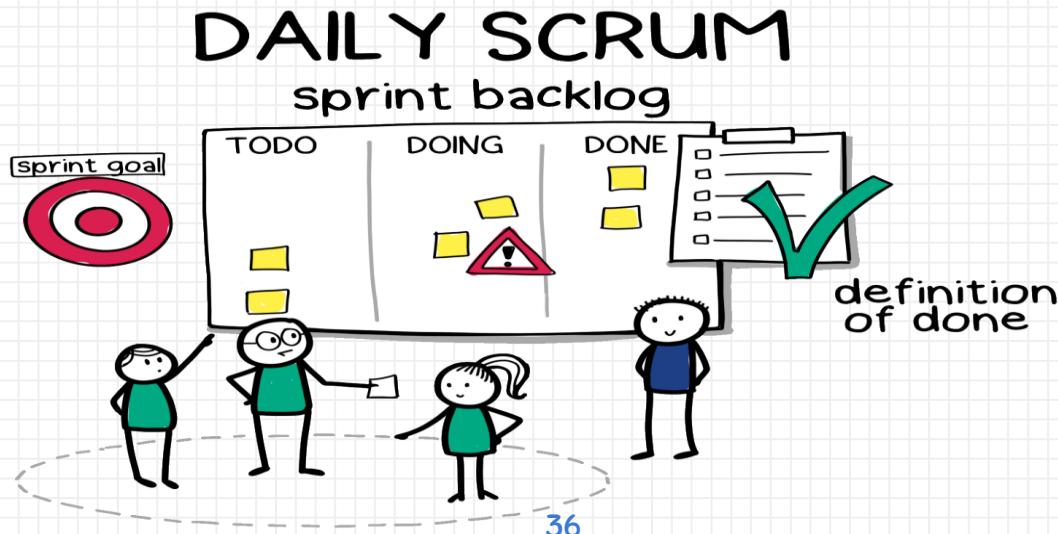
Item de Backlog	A Fazer	Fazendo	Feito	
Cadastro de Usuários	Validar campos obrigatórios nascimento (> 1900) e (< hoje) Validar Alteração - Ok Montar tabela de B.D.	Validar CPF Incluir - Ok Deleção - Ok Usuário só é ativo com concordo selecionado	Validar confirmação da senha Captcha Montar Tela Montar Controlador	Montar Model



Cerimônia: Reunião Diária

8. Reuniões Diárias ou Scrum Diário: Este é o ritmo do Scrum. Todos os dias, no mesmo horário, durante não mais do que 15 minutos, a equipe e o Mestre Scrum se reúnem para responder a três perguntas:

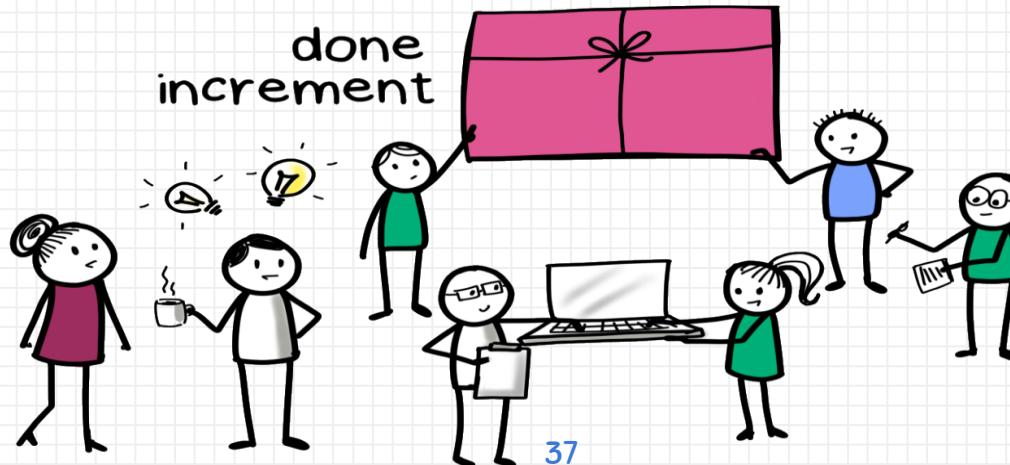
- O que você fez ontem para ajudar a equipe a concluir o Sprint?
- O que você vai fazer hoje para ajudar a equipe a concluir o Sprint?
- Existe algum obstáculo impedindo você ou a equipe de alcançar o objetivo do Sprint?



Cerimônia: Revisão da Sprint

9. Revisão ou Demonstração do Sprint: Trata-se da reunião na qual a equipe mostra o que conseguiu fazer durante o Sprint. Qualquer pessoa pode participar, não apenas o Dono do Produto, o Mestre Scrum e a equipe, mas também os stakeholders, os gestores, os clientes, e qualquer outra pessoa. Esta é uma reunião aberta na qual a equipe demonstra o que conseguiu colocar na coluna Feito. A equipe só deve demonstrar o que satisfaz a Definição de Feito. O que está total e completamente concluído e pode ser entregue sem qualquer trabalho adicional. Pode não ser o produto completo, mas deve ser um atributo concluído do produto.

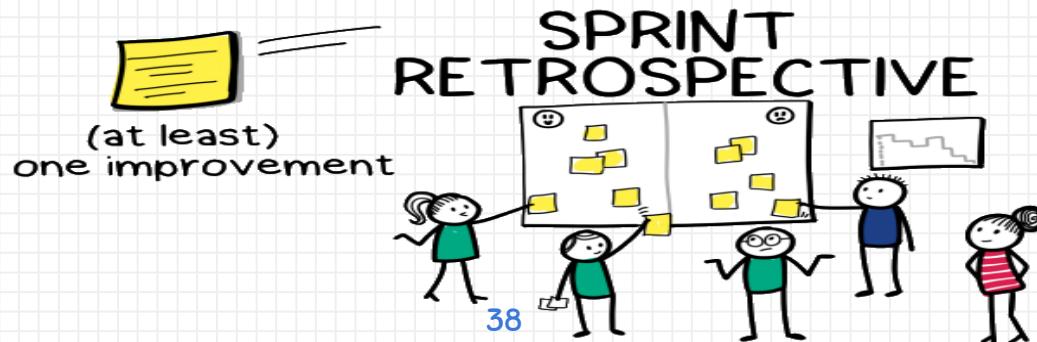
SPRINT REVIEW



Cerimônia: Retrospectiva da Sprint

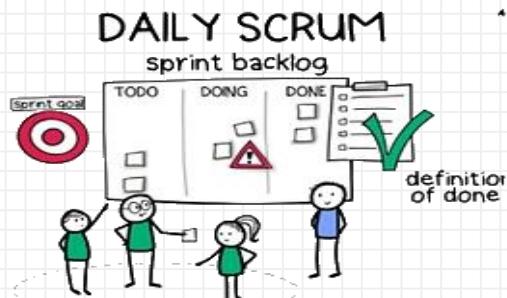
10. Retrospectiva do Sprint: Depois que a equipe mostrou o que conseguiu fazer no Sprint anterior aquilo que está “Feito” e pode ser entregue para clientes para obtenção de feedback, eles se reúnem e pensam no que deu certo e o que poderia ter sido melhor, e o que podem melhorar no próximo Sprint. Qual é o aprimoramento no processo que eles, como uma equipe, podem implementar de forma imediata?

Para ser eficaz, essa reunião requer certa dose de maturidade emocional e atmosfera de confiança. O importante é lembrar-se sempre de que você não está procurando culpados; está olhando para o processo. Por que aquilo aconteceu assim? Por que você não percebeu aquilo? O que poderia ter acontecido para sermos mais ágeis? É essencial que as pessoas na equipe assumam a responsabilidade pelo processo e seus respectivos resultados, e que busquem soluções como uma equipe. Ao mesmo tempo, elas têm de ter coragem de levantar as questões que realmente as incomodam, de forma que a solução seja orientada, em vez de acusadora. E o restante da equipe precisa ter maturidade para ouvir o feedback, absorvê-lo e procurar uma solução, em vez de assumir uma postura defensiva. No final da reunião, a equipe e o Mestre Scrum devem chegar a um acordo sobre um aprimoramento no processo que será implementado no Sprint seguinte. Tal aprimoramento no processo, às vezes, é chamado kaizen, e deve ser colocado nas pendências do próximo Sprint, acompanhado de testes de aceitação. Desse modo, será fácil para a equipe verificar se o aprimoramento realmente foi implementado, e que efeito ele teve sobre a velocidade.



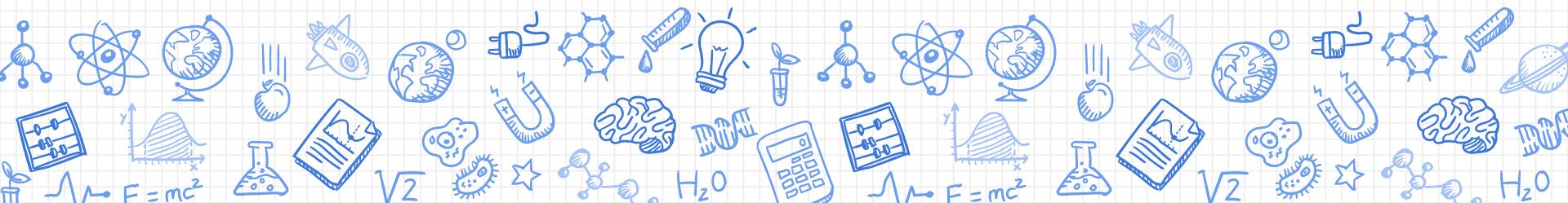
Próxima Sprint

11. Comece imediatamente o próximo Sprint, considerando a experiência da equipe com os impedimentos e os aprimoramentos no processo.



eXtreme Programming (XP)

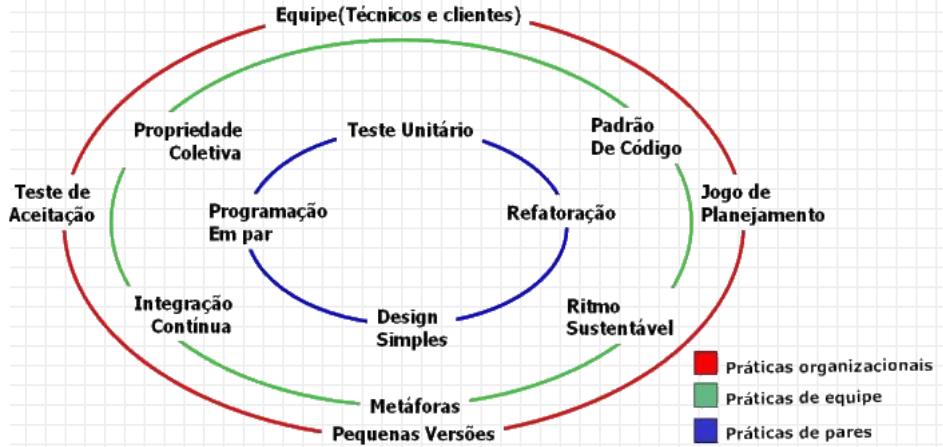
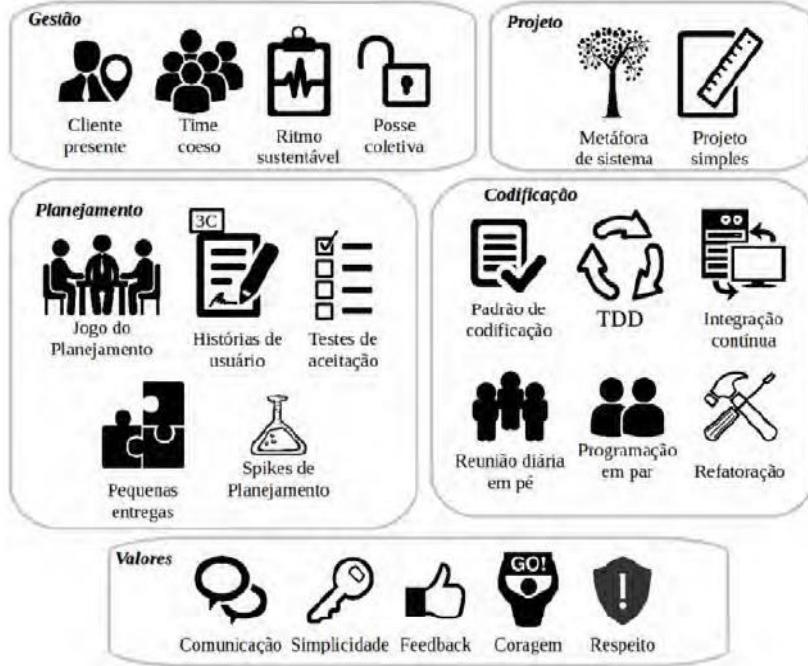
Práticas para o dia a dia no
desenvolvimento ágil de software.



A Metodologia XP foi criada em 1997 para focar mais em **práticas de engenharia**.
Por isso, é mais comum na área de **desenvolvimento de software**.

eXtreme Programming (XP)

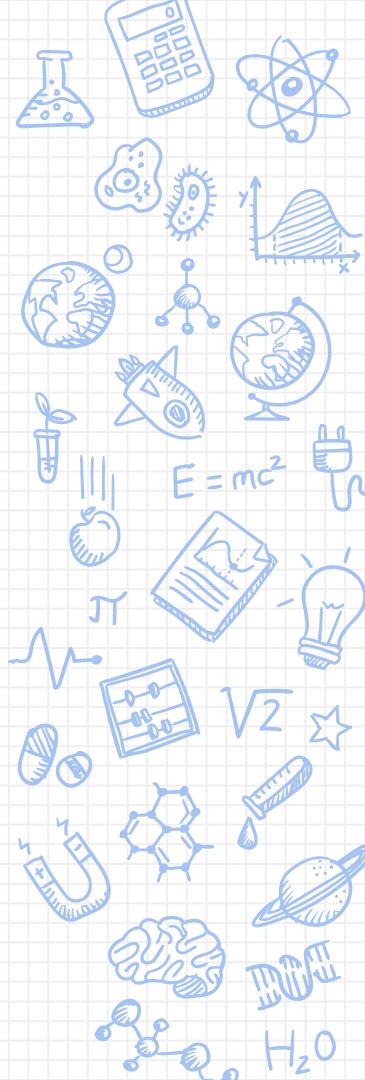
O eXtreme Programming é uma metodologia ágil de desenvolvimento de software desenvolvida por **Kent Beck, Ward Cunningham e Ron Jeffries**, o XP tem como principal tarefa a codificação com ênfase menor nos processos formais de desenvolvimento e com uma maior disciplina de engenharia ágil de software para codificação e testes.



Valores do XP

O eXtreme Programming define cinco valores para que seus papéis e práticas funcionem em sinergia de acordo com sua essência ágil: a comunicação, o feedback, a simplicidade, a coragem e o respeito.

- Comunicação
- Feedback
- Simplicidade
- Coragem
- Respeito

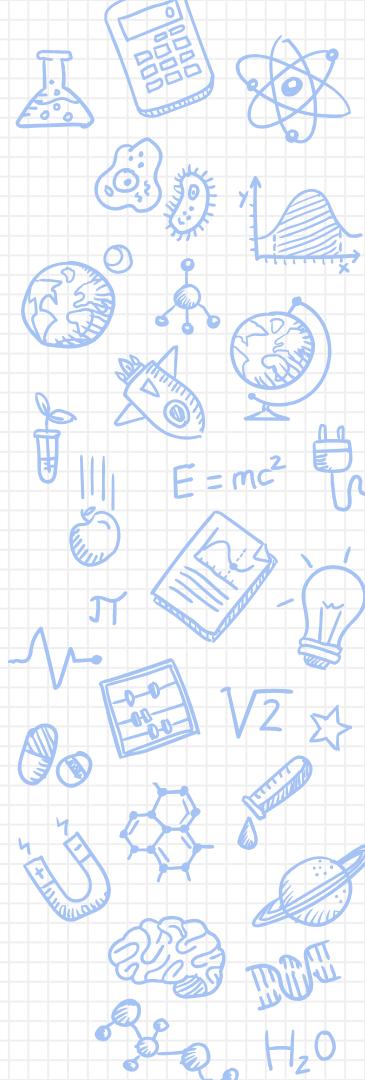


Comunicação

Ter foco na comunicação é essencial em projetos de software, pois é a principal forma de transmitir e trocar informações e conhecimentos. Por essa razão, é importante incentivar meios eficazes de comunicação, cujo valor está presente no Manifesto Ágil e na maioria das práticas de XP.

Ela pode ocorrer de várias formas, como, por exemplo: e-mail, telefone, skype (ou outras ferramentas de chat instantâneo), teleconferência, conversa face a face etc. Porém, é preciso entender que nem todas possuem a mesma efetividade.

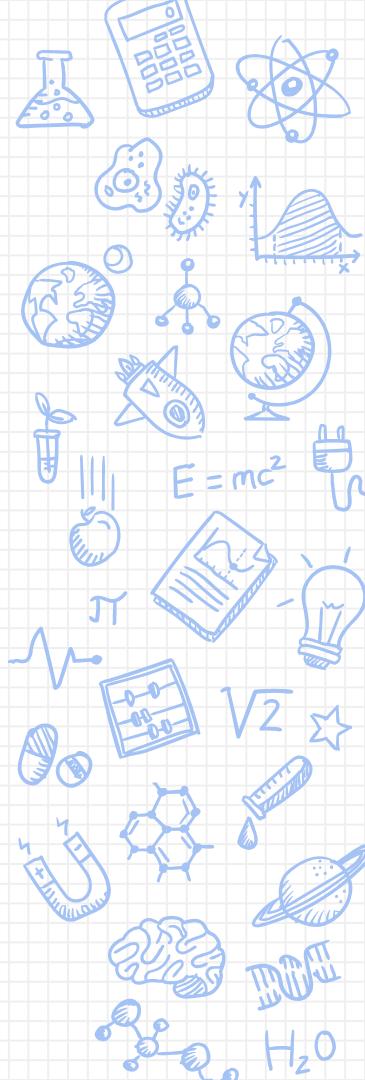
A forma mais eficiente é a conversa face a face, usando um quadro branco como apoio para rabiscar ideias e/ou rascunhos sobre arquitetura de software. Distribuir quadros brancos no ambiente de desenvolvimento pode funcionar muito bem para estimular discussões de arquitetura, fluxos, algoritmos e, assim, aumentar a possibilidade de implementar certo logo na primeira tentativa. A comunicação incentiva diretamente outro valor essencial no XP: o feedback.



Feedback

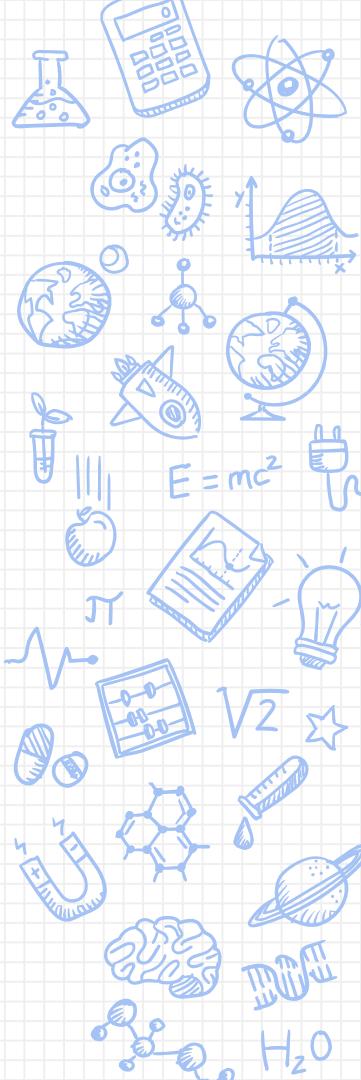
Na adoção das práticas, o feedback é realizado a todo momento, seja em relação aos requisitos do cliente, ao resultado da execução de testes unitários, ou na compilação do código na integração contínua. A compreensão das necessidades dos usuários e do negócio propriamente dito é um aprendizado constante.

A razão de adotarmos estratégias iterativas e incrementais é que isso permite que os inevitáveis erros das pessoas sejam descobertos o mais cedo possível e reparados de forma metódica.

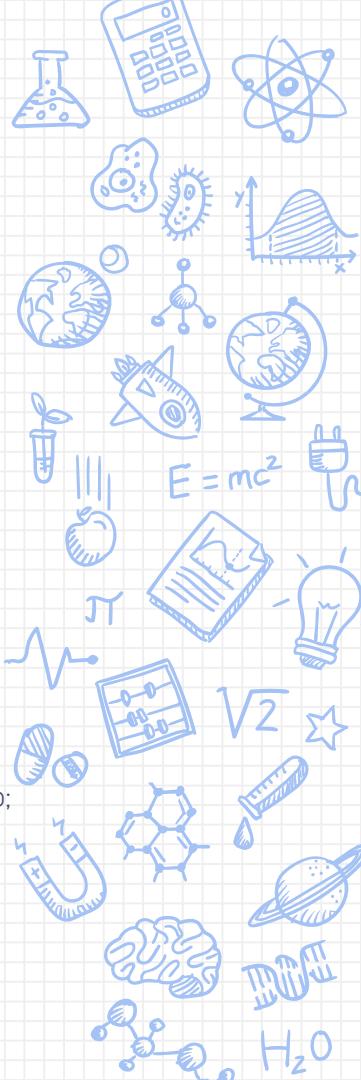


Simplicidade

A simplicidade é outro valor presente nas práticas XP, e a chave e a diretriz de um projeto ágil. Ela visa manter o trabalho o mais simples e focado possível, entregando somente o que realmente agrega valor. Acrescentar suporte para futuras funcionalidades de forma desnecessária complica o design e eleva o esforço para desenvolver incrementos subsequentes. Infelizmente, projetar um design simples não é algo fácil, pois muitos times já estão acostumados coma prática de futurologia, sofrendo da Síndrome de Nostradamus; ou seja, mesmo sabendo o que o cliente quer hoje, eles insistem em tentar desenvolver uma solução que também resolva problemas futuros. O ponto é: não temos certeza sobre o que vai acontecer no futuro, então precisamos nos focar apenas nos problemas e necessidades atuais do nosso cliente. Lembre-se: busque sempre desenvolver o suficiente de forma simples e com qualidade.



Coragem



A coragem também é incentivada por meio do uso das práticas do XP. Por exemplo, o desenvolvedor apenas se sentirá confiante para refatorar o código criado por outro colega caso o time tenha um padrão de codificação e posse coletiva do código. Além dessa prática, o uso de controle de versão e testes unitários também encorajam isso.

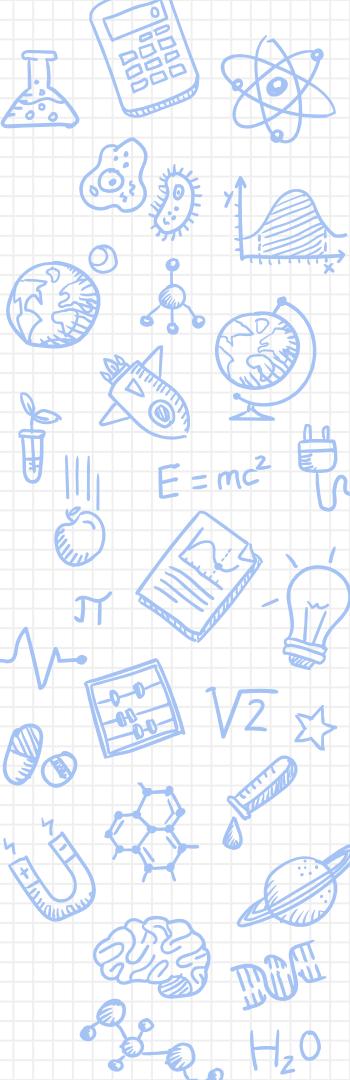
O cliente teme:

- Não obter o que foi solicitado/contratado;
- Pedir a coisa errada;
- Pagar demais e receber pouco;
- Jamais ver um plano relevante;
- Não saber o que está acontecendo (falta de feedback);
- Ater-se às primeiras decisões de projeto e não ser capaz de reagir à mudança do negócio.

Já o desenvolvedor teme:

- Ser solicitado a fazer mais do que sabe fazer;
- Ser ordenado a fazer coisas que não façam sentido;
- Ficar defasado tecnicamente;
- Receber responsabilidades, mas sem autoridade;
- Não receber definições claras sobre o que precisa ser feito;
- Sacrificar a qualidade em função do prazo;
- Resolver problemas complexos sem ajuda;
- Não ter tempo suficiente para fazer um bom trabalho.

Respeito

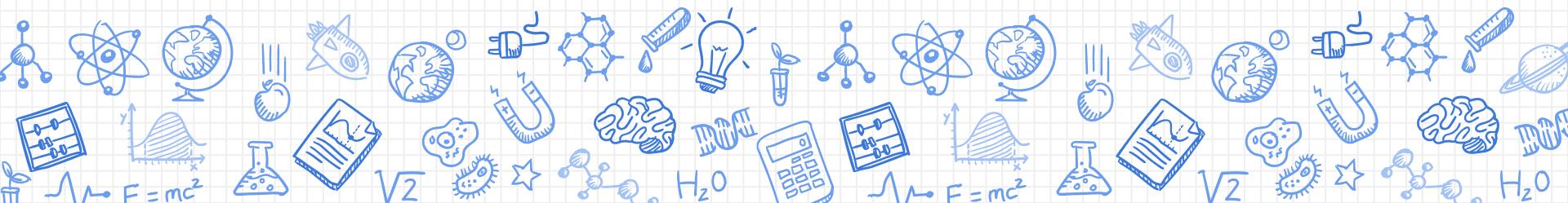


O respeito pelos colegas de time e pelo cliente é muito importante. Pessoas que são respeitadas sentem-se valorizadas. Os membros do time precisam respeitar o cliente; o cliente precisa respeitar os membros do time; e, além disso, o cliente deve fazer parte das decisões. Os desenvolvedores nunca devem realizar mudanças que quebrem a compilação e que fazem os testes de unidade falharem ou realizarem outras ações que possam atrasar o trabalho ou o progresso do time. Todos no time devem respeitar a posse coletiva do código, sempre primando pela qualidade e buscando um design simples por meio da refatoração. A adoção dos demais quatro valores do XP conduzirá a este quinto valor: o respeito. Isso vai garantir um alto nível de motivação, além de incentivar a lealdade entre todas as pessoas envolvidas no projeto, criando um verdadeiro senso de trabalho em equipe.

O ponto é: como se adquirem confiança e respeito? Obviamente, entregando softwares que funcionam. E mais que isso, o respeito é fundamental para uma relação transparente e duradoura. É isso que realmente forma a parceria e a colaboração de todos envolvidos, algo essencial para a entrega contínua de valor.

Engenharia Ágil

Algumas práticas de Gestão, Projeto,
Planejamento e Codificação de Software do XP.



Práticas de Engenharia Ágil

Quando se fala em práticas de **Engenharia Ágil do XP** podemos elencar algumas que devem ser levadas “**ao extremo**” durante a condução do projeto:

Fases pequenas (small releases): dividir o projeto em ciclos curtos.

Jogo do planejamento (planning game): priorizar quais funcionalidades serão desenvolvidas antes de cada ciclo.

Metáfora (metaphor): entender a realidade do cliente e traduzir as necessidades dele nos requisitos do projeto.

Design simples (simple design): entregar apenas o que o cliente pediu, em vez de tentar reinventar a roda.

Testes de aceitação (customer tests): cada nova funcionalidade disponibilizada deve ser validada pelo cliente.

Semana de 40 horas (sustainable pace): programar é uma atividade que exige que a mente esteja descansada, por isso, o XP defende que não seja feito trabalho extra.

Propriedade coletiva (collective ownership): ninguém deve ter que pedir permissão para poder modificar o código-fonte, pois o código-fonte é uma criação colaborativa.

Programação em par (pair programming): a programação do código deve ser feita em duplas, preferencialmente composta por um profissional iniciante e outro mais experiente, no mesmo computador.

Padronização do código (coding standards): a equipe de desenvolvimento precisa definir padrões para a construção do código, pois isso fará com que o código fique uniformizado e mais comprehensível.

Desenvolvimento orientado a testes (test driven development): toda funcionalidade deve passar por um rigoroso processo de testes antes de ser validada, a fim de que não haja retrabalho para a equipe.

Refatoração (refactoring): o código deve passar por revisões periódicas, para ser continuamente aperfeiçoado.

Integração contínua (continuous integration): depois que uma funcionalidade for testada ela deve ser imediatamente sincronizada entre a equipe, evitando conflitos.

Dwight David “Ike” Eisenhower:

“Planos não são nada. Planejamento é tudo.”

O Jogo do Planejamento

O Jogo do Planejamento é uma prática que desempenha o papel fundamental do XP. Ele nos informa como dividir um projeto em funcionalidades, histórias e tarefas. Fornece orientação para a estimativa, a priorização e o cronograma dessas funcionalidades, histórias e tarefas.

Esta etapa envolve os clientes e os desenvolvedores para planejarem as entregas de uma forma colaborativa. O nome jogo mostra a essência da dinâmica: tratar o planejamento como um jogo que contém um objetivo, jogadores, peças e regras. Seu resultado é maximizado por meio da colaboração de todos os jogadores. Ele agrupa todo o processo necessário para a entrega de software como a escrita de histórias de usuário, a priorização pelo cliente e a criação das estimativas.

As pessoas de negócio definem o escopo, a prioridade, a composição e as datas das entregas. As decisões de negócio têm suporte das pessoas de desenvolvimento. Estas tomam decisões técnicas por meio de estimativas e de análises de consequências técnicas no negócio, e decidem seu processo de trabalho. Tanto o Negócio quanto o Desenvolvimento têm voz nas decisões das entregas. Não há bola de neve no XP.

O desenvolvimento iterativo e incremental é utilizado no jogo do planejamento. Os incrementos são definidos no planejamento de releases e as iterações, no planejamento de iterações. Cada incremento representa uma entrega de valor em produção ao cliente, sendo desenvolvido em uma ou mais iterações. A divisão em iterações auxilia o time a quebrar as entregas em pedaços.

Definindo o Jogo e suas Regras

Os jogos e as regras do jogo do planejamento são:

- ❑ **Objetivo:** maximizar o valor do software produzido pelo time, ou seja, colocar em produção a maior quantidade de histórias de usuário com maior valor ao longo do projeto.
- ❑ **Estratégia:** o time deve investir o menor esforço para colocar a funcionalidade de maior valor em produção tão rápido quanto possível, em conjunto com estratégias de projeto e programação para reduzir o risco. Dividir para conquistar é a estratégia utilizada ao dividir o software em entregas frequentes, e cada entrega em iterações pequenas.
- ❑ **As peças:** as peças básicas do jogo são histórias de usuário e as tarefas de implementação. As histórias são as peças no planejamento das releases, já as tarefas, no planejamento das iterações.
- ❑ **Os jogadores:** as pessoas do Desenvolvimento e as pessoas de Negócio.
- ❑ **Os movimentos:** as regras dos movimentos existem para lembrar a todos de como eles gostariam de agir em um melhor ambiente com confiança mútua e dar uma referência para quando as coisas não estiverem indo bem.

Entendendo Regras e Comprometimentos

Os movimentos do jogo são dados por suas regras e são divididos em duas partes: a primeira trata dos movimentos para as entregas (planejamento de releases); a outra dos movimentos para as iterações (planejamento de iterações), conforme apresentado na figura a seguir.



Planejamento de releases

Para uma release, o jogo do planejamento define as regras para o cliente direcionar o desenvolvimento na frequência de uma a três semanas. Seus movimentos para uma release são dados em três fases: **Exploração, Comprometimento e Direcionamento.**

Exploração: ambos os jogadores descobrem novos itens que o sistema pode fazer, dividindo-se em três movimentos:

- Escrever uma história de usuário:** o Negócio escreve algo que o sistema deverá fazer.
- Estimar uma história de usuário:** o Desenvolvimento estima o tempo ideal de programação para implementar a história.
- Quebrar uma história de usuário:** caso o Desenvolvimento não consiga estimar a história ou sua estimativa for muito grande, o Negócio identifica sua parte mais importante para que o Desenvolvimento possa estimá-la e desenvolvê-la.

Planejamento de releases

Comprometimento: o Negócio decide o escopo e o tempo para a próxima entrega de acordo com as estimativas, e o Desenvolvimento se compromete com a entrega, dividindo-se em quatro movimentos:

- Ordenar por valor:** o Negócio separa as histórias em três categorias: as essenciais, as menos essenciais, mas de alto valor; e as que seriam agradáveis de se ter.
- Ordenar por risco:** o Desenvolvimento separa as histórias em três categorias: aquelas que podem ser estimadas com precisão, aquelas que podem ser estimadas razoavelmente bem, e aquelas que não podem ser estimadas de qualquer modo.
- Ordenar por velocidade:** o Desenvolvimento mostra ao Negócio quanto de tempo ideal de programação o time poderá trabalhar em um mês do calendário.
- Selecionar escopo:** o Negócio escolhe as histórias para a entrega (trabalhando por tempo) ou uma data para as histórias a serem desenvolvidas (trabalhando por escopo),

Planejamento de releases

Direcionamento: o plano é atualizado no que foi aprendido pelo Negócio e pelo Desenvolvimento, dividindo-se em quatro movimentos:

- Iteração:** a cada iteração (uma a três semanas), o Negócio seleciona uma com as histórias de maior valor a serem implementadas;
- Recuperação:** caso o Desenvolvimento descubra que sua velocidade foi superestimada, ele pode pedir ao Negócio para encaixar apenas as histórias de maior valor de acordo com a nova velocidade;
- Nova história:** caso o Negócio descubra uma nova história a ser adicionada à entrega que já está sendo desenvolvida, ele pode escrevê-la, o Desenvolvimento estimá-la, e, então, trocá-la por outra selecionada pelo Negócio;
- Reestimar:** caso o Desenvolvimento descubra que o plano não é exato, ele pode reestimar todas as histórias restantes.

Planejamento de iterações

As peças são as tarefas de implementação e os jogadores são os programadores. As decisões sobre as iterações são mais flexíveis em questão de escopo e de prazos, pois dependem mais do Desenvolvimento; enquanto as sobre entregas dependem mais do Negócio. Suas fases são semelhantes às de uma entrega: **Exploração, Comprometimento e Direcionamento**.

Exploração: o Desenvolvimento cria suas tarefas e identifica o tempo de trabalho, dividindo-se em três movimentos:

- Escrever uma tarefa:** transformar as histórias da iteração atual em tarefas.
- Quebrar ou unir tarefas:** caso a tarefa esteja grande demais, quebre-a em tarefas menores. Caso ela esteja pequena demais, una-a a outra relacionada.
- Estabeleça um fator de carga:** cada programador escolherá seu fator de carga para a iteração, o qual é a porcentagem de tempo que ele realmente estará desenvolvendo; ou seja, é removido o tempo em que se realiza outras, tais como reuniões do time.

Planejamento de iterações

Comprometimento: o Desenvolvimento compromete-se com suas tarefas, dividindo-se em três movimentos:

- Aceitar uma tarefa:** o programador puxa a tarefa para si e aceita a responsabilidade por ela.
- Estimar a tarefa:** o programador responsável pela tarefa a estima em Dias Ideais de programação.
- Balanceamento:** cada programador soma suas tarefas e multiplica pelo fator de carga, balanceando sua carga de trabalho, caso esteja maior ou menor do que a estabelecida.

Direcionamento: o Desenvolvimento desenvolve as tarefas para verificar a história, dividindo-se em quatro movimentos:

- Implementar uma tarefa:** o programador pega o cartão da tarefa, encontra um par para programar, desenvolve a tarefa com TDD (Desenvolvimento Guiado por Testes) e ao final integra o código fazendo passar a suíte de teste relacionada.

Planejamento de iterações

- ❑ **Registrar progresso:** a cada dois ou três dias, um membro do time atualiza o andamento das tarefas de cada membro, com o tempo gasto e o tempo que falta para cada tarefa;
- ❑ **Recuperação:** programadores que ficaram sobrecarregados pedem ajuda reduzindo o escopo das tarefas, reduzindo o escopo das histórias, removendo aquelas não essenciais ou pedindo ao cliente para postergar a história para outra iteração;
- ❑ **Verificar a história:** caso estiverem prontos, rodar os testes funcionais para as tarefas que já foram concluídas, complementando a suíte de testes funcionais com os novos casos de testes.

As histórias de usuários são estimadas antes de serem desenvolvidas. Já na iteração, o programador pega a tarefa para si e depois a estima. Nem todas estarão ligadas diretamente com o Negócio, mas sim com as necessidades do Desenvolvimento; por exemplo, para configurar um servidor de integração.

Mantendo o foco

Você lembra-se do objetivo do jogo do planejamento? Maximizar o valor do software construído pelo time para o cliente é o foco do jogo. É importante que todos participem (Desenvolvimento e Negócio) com colaboração. A simplicidade (a arte de maximizar a quantidade de trabalho não realizado) é essencial para o feedback contínuo, sendo importante também para o time sentir-se valorizado por fazer entregas frequentes de valor. O Desenvolvimento e o Negócio devem manter um ritmo sustentável e constante de trabalho para manter o foco.

Todo momento é um momento de aprendizagem

No jogo do planejamento, até quando se erra, aprende-se. O erro não é visto como algo ruim, desde que se aprenda com ele. Quando o time reconhecer que superestimou sua velocidade, ele deve comunicar ao Negócio para rearranjar as histórias dentro de sua nova velocidade. Quando houver problemas com as estimativas, devem-se reavaliar essas estimativas das histórias. É preciso ter coragem para buscar um processo que seja transparente. A comunicação e o feedback são importantes para que os dois jogadores aprendam a melhor forma de maximizar o jogo.

Esther Derby:

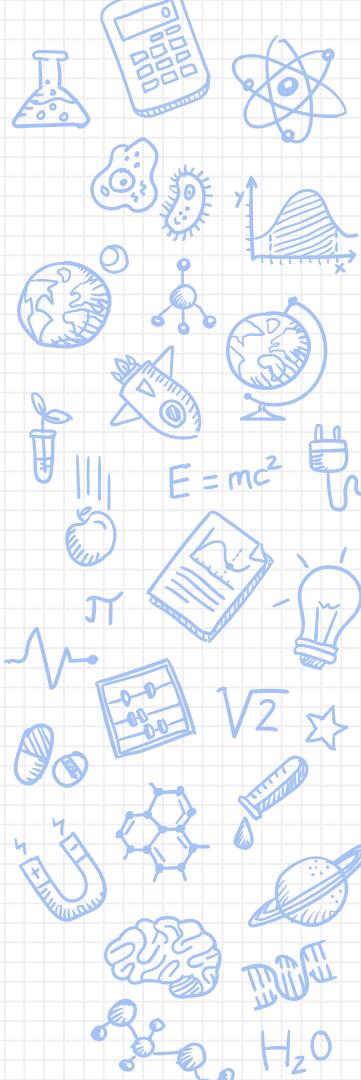
“98% de nosso pensamento está realmente acontecendo em um nível inconsciente, grande parte através de metáforas.”

Metáfora de Sistema

Metáforas em geral estão no nosso cotidiano. No dia a dia, frequentemente as usamos para expressarmos ideias de um modo mais simples, fácil e rápido; tentamos encontrar algo semelhante que a pessoa já conheça para ela conseguir entender o que queremos explicar.

No XP o uso de Metáforas é a prática que cria e promove a terminologia e para que todos da equipe e da empresa falem a mesma língua com o objetivo de se comunicar a respeito do sistema.

A metáfora traz uma visão comum que auxilia o time e o cliente a entender os elementos do sistema. Ela funciona como um pattern de alto nível e explica seu design sem uma documentação pesada. Ela é semelhante à linguagem ubíqua, porém, enquanto esta se concentra em uma linguagem comum entre o negócio e os desenvolvedores, aquela foca na linguagem da arquitetura da solução.



Ron Jeffries:

“Simplicidade é mais complicada do que você pensa. Mas ela é bem valiosa.”

Design Simples

Obter um projeto simples não é uma tarefa fácil, porém, manter a simplicidade agiliza no desenvolvimento de novas funcionalidades e na resposta a mudanças. O projeto simples está presente em todas as fases do XP. A simplicidade é abordada de um modo extremo. Ele começa assim e mantém-se por meio de testes automatizados e refatoração.

Tentar prever o futuro é antiXP. Nenhuma funcionalidade adicional é implementada, pois desvia do caminho da solução e aumenta a complexidade do software. Apenas tenha cuidado com as dívidas técnicas que poderão ser geradas, ou com funcionalidades inacabadas. A programação em pares evita o design extra, porque um membro da dupla não vai querer esperar que o outro desenvolva coisas sem relevância.

As regras de Kent Beck para o Design Simples são:

- Roda todos os testes;
- Expressa todas as ideias necessárias;
- Não contém código duplicado;
- Possui a menor quantidade de classes, métodos e variáveis;
- É conciso e legível.

Em todos os níveis de granularidade de software deve-se pensar na simplicidade, ou seja, do código ao produto.

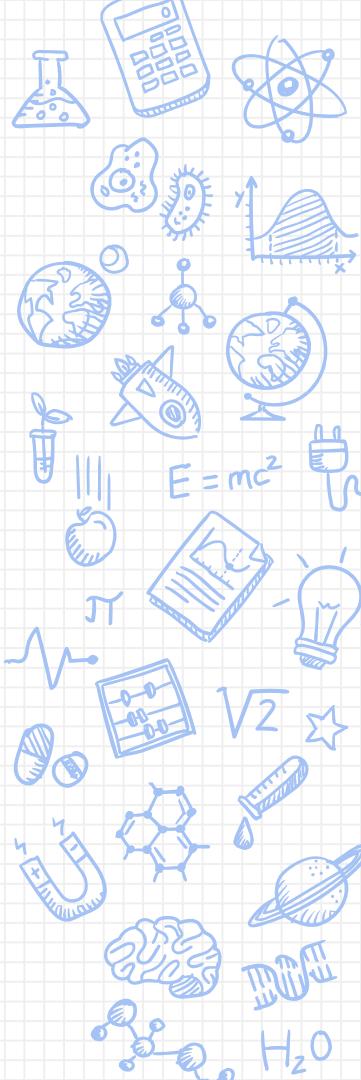
Lisa Crispin:

“Testes de aceitação são mapas da estrada para a iteração, dizendo ao time aonde é preciso ir e quais pontos de referência olhar.”

Testes de Aceitação

O propósito dos testes de aceitação é a comunicação, a transparência e a precisão. Quando os desenvolvedores, os testadores e o cliente concordarem com eles, todos entenderão qual é o plano para o comportamento do sistema. Chegar a esse ponto é responsabilidade de todas as partes. Eles validam como o cliente aceitará as funcionalidades prontas, pois são testes funcionais que guiam o time no desenvolvimento para, então, colocar em produção o que foi decidido que o sistema deve conter.

Esses testes são criados a partir das histórias de usuários selecionadas pelas pessoas de negócio no planejamento de iterações. A história de usuário apenas estará completa quando todos seus critérios de aceitação estiverem passando. Auxiliado pelo testador do time, o cliente é quem especifica os exemplos dos critérios de aceitação na forma de cenários, que podem conter dados e variações que criticam a história. Os exemplos de cenários esclarecem as histórias ao time.



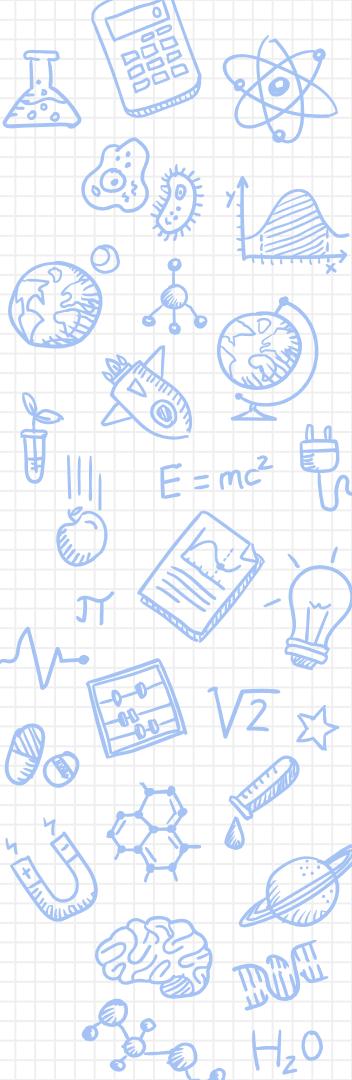
Automatizando Teste de Aceitação

Testes de aceitação sempre têm de ser automatizados por uma razão simples: custo. Caso você ache custoso ter testes de aceitação automatizados, aguarde para ver o custo da confusão e do debug de código ao longo prazo. Para fazer pequenas entregas frequentes, é necessário que os testes de aceitação estejam automatizados e que rodem no build gerado pelo servidor de integração contínua. O cliente poderá decidir se uma história desenvolvida poderá ir para produção mesmo quando alguns testes de aceitação não estiverem passando. Apenas ele tem esse poder.

Validando com critérios de aceitação

Cada história de usuário possui um ou mais critério de aceitação, criado também pelo cliente. O testador ajuda-o a pensar em cenários bons de teste, sendo escritos já prevendo a automatização. Esses critérios descrevem como a história será aceita pelo cliente e servem de guia para o desenvolvimento. Como exemplo, considerando a história “*Como um repositor de estoque, quero localizar quais as prateleiras do supermercado que contêm menos de 50% de sua capacidade de produtos*”, podemos escrever os critérios de aceitação:

- Deve exibir uma lista de prateleiras com o setor e a seção de cada uma, e o tipo e o nome do produto contido;
- Deve exibir as prateleiras com menos de 50% da capacidade de itens;
- Não deve exibir as prateleiras com 50% ou mais da capacidade de itens.



BDD (Behaviour Driven Development)

Os critérios de aceitação também podem seguir um modelo. Comumente, o utilizado é o de cenários de BDD em inglês (Behaviour-Driven Development) traduzido para o português fica Desenvolvimento Guiado por Comportamento. Esse modelo possui três informações:

- Dado que:** precondição, são os passos para preparar para ação a ser validada.
- Quando:** ação que vai disparar o resultado a ser validado.
- Então:** resultado a ser validado.

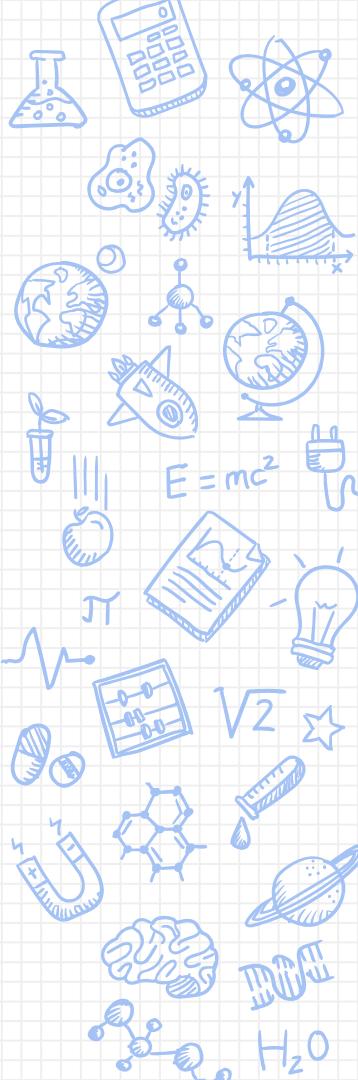
Como um exemplo, o critério de aceitação deve exibir o setor e a seção da prateleira como tipo e o nome do produto usando o modelo, ficará desta forma:

- Dado que** o usuário logado é um repositor de estoque.
- Quando** o item de menu “Consultar prateleiras para reposição” for clicado.
- Então** a lista de todas as prateleiras para reposição é exibida.

Robert C. Martin (Uncle Bob):

“Desenvolvimento de software é uma maratona, não uma sprint.”

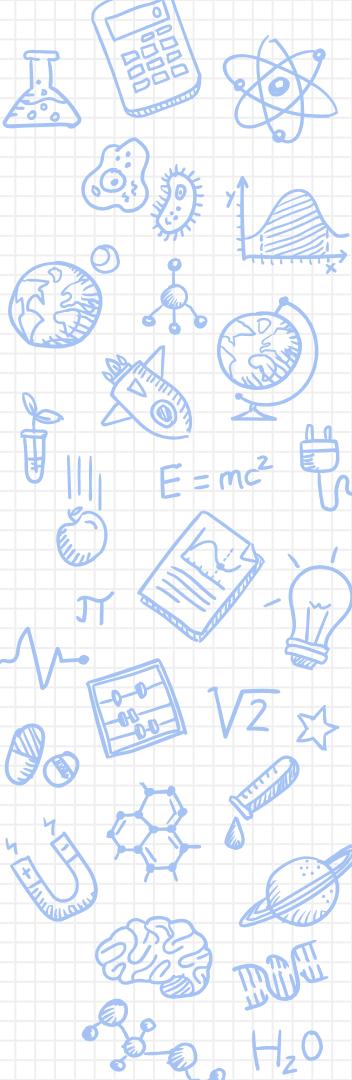
Ritmo Sustentável



Imagine uma pessoa correndo em uma maratona. O caminho é longo, então a melhor estratégia é correr em um ritmo constante até a chegada. Todos os corredores profissionais sabem disso. Caso alguém queira correr mais rápido que seu ritmo, em poucos minutos ele estará ofegante e terá que parar para descansar, também perdendo sua concentração na respiração. O mesmo ocorre ao desenvolver software.

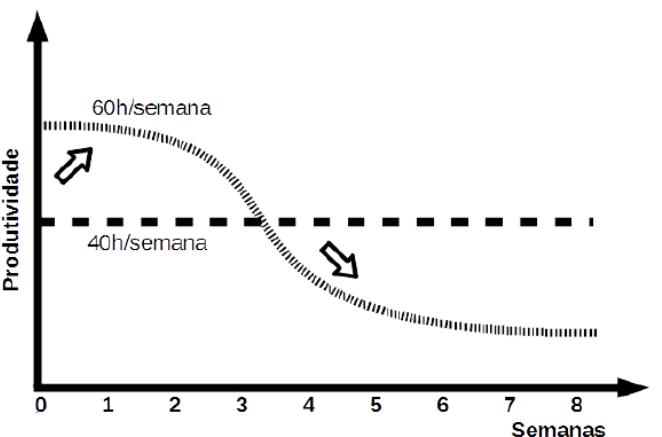
Ter ritmo sustentável é uma regra criada pelo XP para balancear o desenvolvimento com as demandas do negócio. Todos têm seus ritmos equilibrados: o time XP, o cliente e pessoas envolvidas com o projeto. Utilizar a simplicidade traz soluções mais efetivas ao cliente, mais ainda quando há mais demandas do negócio do que a capacidade do time. A qualidade do software é essencial para manter esse fluxo de trabalho normalizado, porque defeitos gerariam retrabalho para as próximas iterações.

Produtividade em longo prazo é o ponto chave aqui. Durante uma semana ou pouco mais, o time consegue produzir mais trabalhando com horas extras. Porém, a longo prazo é inviável, porque o rendimento vai decaindo semana a semana.



40 Horas Semanais

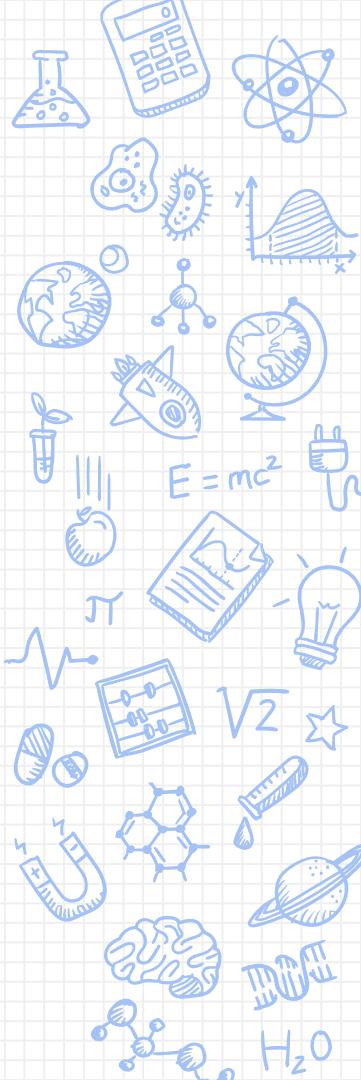
O conceito de ritmo sustentável iniciou-se com a prática de 40 horas semanais (40-hour week) no livro de origem do eXtreme Programming. Ron Jeffries abstraiu esse conceito e o renomeou para ritmo sustentável. Curiosamente, há várias décadas, em 1926, Ford já havia estabelecido a jornada de trabalho de 40 horas semanais, pois sabia que o problema seria resolvido com uma boa organização, e não com mais horas trabalhadas. A imagem a seguir ilustra a relação da produtividade por semana de trabalho em um ritmo sustentável (40 horas semanais) e insustentável (60 horas semanais, fazendo horas extras).

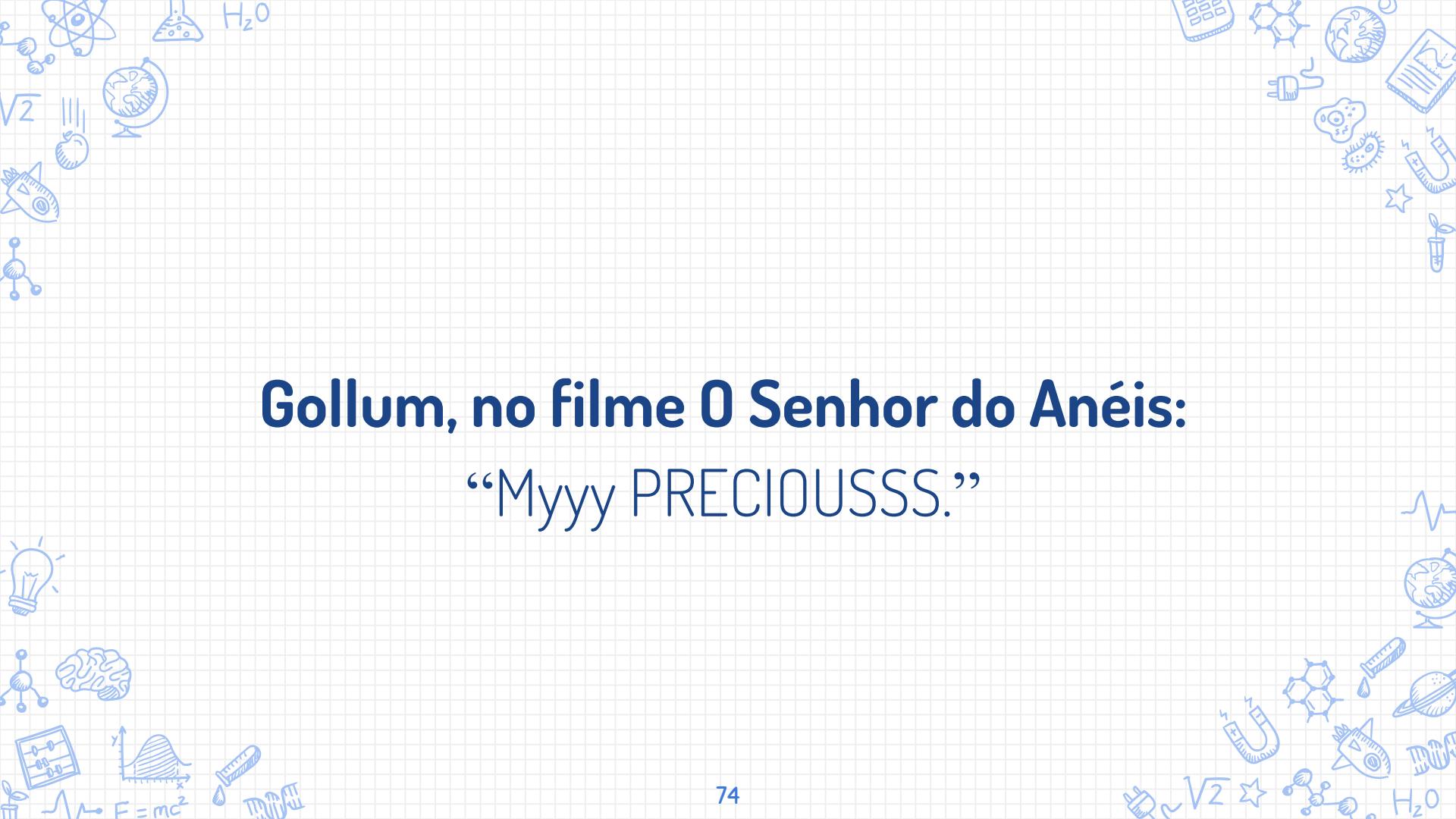


40 Horas Semanais

Pessoas não são máquinas, e desenvolver software não é uma atividade simples. É necessário raciocínio aprimorado para produzir cada funcionalidade que é diferente da anterior já feita. Programadores não são mão de obra, mas sim trabalhadores do conhecimento (chamados de cérebro de obra). Sabe-se que, nos humanos, o cérebro é o órgão que consome mais energia, totalizando até 20% do total da energia do corpo. Comumente, um dia de trabalho rende pelo trabalho normal de uma semana, assim como o inverso uma semana inteira não rende nem por um dia normal de trabalho.

Algumas pessoas têm capacidade de trabalhar em todo seu potencial criativo, atento e confiante em 45 horas por semana, outras em 35 horas, mas raramente alguém conseguirá isso em 60 horas, em longo prazo. Um time cansado trabalhará menos, não importa quanto tempo trabalhem a mais.



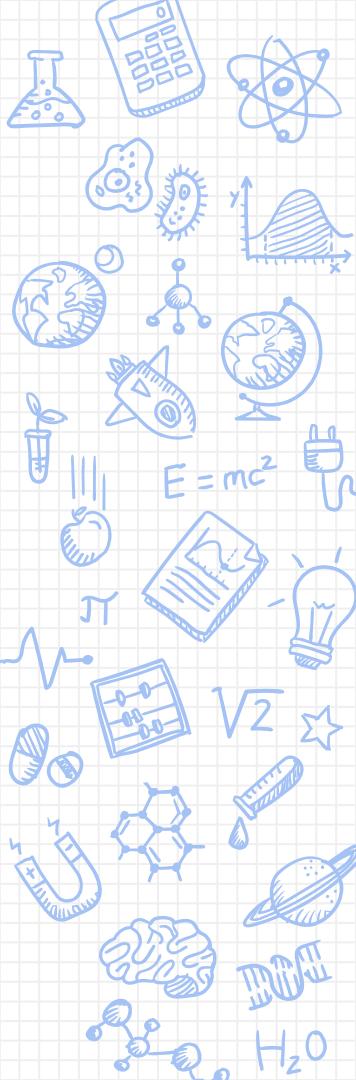
A background filled with various blue line-art icons related to science and education, including a atom, a globe, a rocket, a brain, a lightbulb, a graph, a test tube, a magnet, a DNA helix, a plug, a virus, a heart rate monitor, a planet, and chemical structures.

Gollum, no filme O Senhor do Anéis: “Myyy PRECIOUSSS.”

Propriedade Coletiva

Propriedade coletiva é uma prática indispensável no XP, pois envolve colaboração, comunicação e feedback. Com a posse coletiva, qualquer membro ou par do time pode implementar uma funcionalidade, arrumar um bug ou refatorar em qualquer parte do código, a qualquer momento. No XP, todos têm a responsabilidade pelo sistema. Isso encoraja cada membro do time a sentir-se responsável pela qualidade do todo. É claro que nem todos saberão sobre todas as partes igualmente, mas todos saberão, ao menos, um pouco de cada parte do sistema.

Quando um par do time encontra uma oportunidade de melhorar o código, ele pode tranquilamente refatorar e melhorar seu design. A posse coletiva gera bastante sinergia com as outras práticas do XP, e o versionamento com a integração contínua avisará a todos o que está sendo alterado. O trabalho em time com a posse coletiva encoraja a difusão de ideias e de conhecimento. A programação em par ajuda a distribuir esse conhecimento por meio do aprendizado entre os membros do grupo, deixando a posse coletiva mais refinada. Porém, para tudo isso funcionar, é necessário que nenhum integrante ache que é o dono de partes do sistema. É preciso colaboração.

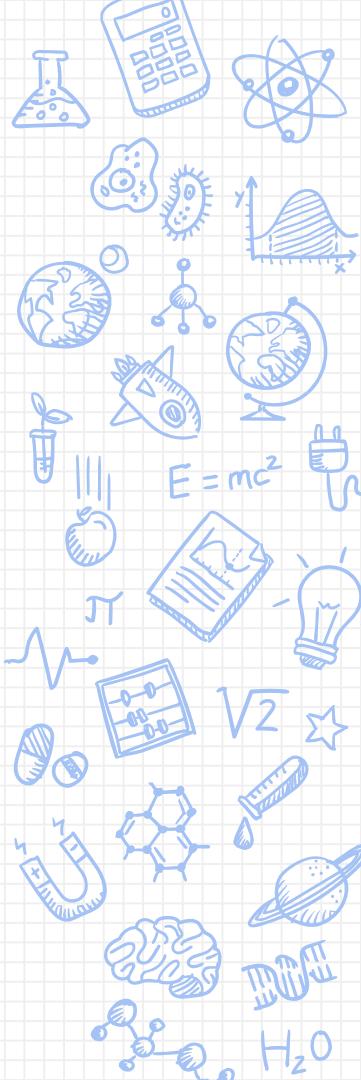


My Precious

Você assistiu ao filme O Senhor dos Anéis? O personagem Gollum (ou Sméagol) chamava o anel de my precious (em português, “meu precioso”). Não faça isso com o seu código. Deixe seu time colaborar! O objetivo é evitar as ilhas de conhecimento, reduzindo o risco de depender apenas de uma pessoa. A posse coletiva é um benefício para todos os membros (cada um poderá sair de férias ou tirar uma folga com mais tranquilidade por não dependerem tanto dele, por exemplo).

Quando a posse não é coletiva, somente uma pessoa é dona do código e os programadores devem submetê-lo para revisão e aprovação. Isso é, basicamente, o oitavo desperdício no Lean, que é relativo ao capital humano e à criatividade dos funcionários. Consequentemente, isso gerará diversos outros desperdícios: espera (o time aguardará a revisão do dono do código).

transporte (enviar o código ao dono); defeito (nem toda a decisão do dono será a melhor); super processamento (a revisão para controle pelo dono do código); e estoque (a fila de itens a revisar pelo dono). Uma pessoa é notavelmente um gargalo no time ou no processo quando não consegue atender todas suas demandas e cria estoque de tarefas, nas quais ninguém pode ajudar. E atenção: qualquer membro poderá sair da empresa a qualquer momento, o que pode impactar no negócio do cliente.



Lei de Linus, Eric Steven Raymond:

“Dados olhos suficientes, todos os erros
são óbvios.”

Programação em par

No XP, todo o código de produção é criado por programação em par (pair programming). Isso significa ter duas pessoas piloto e copiloto trabalhando juntas em apenas um computador, com foco em uma única tarefa e ao mesmo tempo. O que o XP indica é programar em par quando o código de produção for escrito. Não necessariamente se deve programar em par 100% do tempo, pois atividades, como pesquisas e leituras, não têm essa necessidade. Trabalhar dia a dia ao lado de um colega exige habilidades sociais que levam tempo em aprender, porém aumenta a cooperação no time, independente do status da função de cada integrante. Programação em par não é mentoria, uma vez que é um trabalho colaborativo entre ambas as partes, independente de terem muita diferença nas experiências. Não é aceitável dizer “O design que você fez possui um erro”, mas sim “O design que nós fizemos possui um erro”.

A Programação em par é uma prática que traz muitos benefícios:

- **Revisão de código contínua:** muitos erros são pegos quando eles estão sendo codificados, em vez de serem descobertos por um testador, tornando a quantidade de erros consideravelmente menor;
- **Discussão contínua da solução:** o design é aprimorado e o código fica mais sucinto;
- **Afinamento do par:** o time resolve problemas mais rapidamente por estar coeso também em pares;
- **Aprendizagem:** as pessoas aprendem significativamente sobre o sistema e sobre engenharia de software;
- **Gestão do conhecimento:** o projeto acaba com várias pessoas entendendo cada pedaço do sistema;
- **Trabalho colaborativo:** os indivíduos aprendem a trabalhar em time e a conversar mais frequentemente, dando mais fluxo à informação e à dinâmica do grupo;

Robert C. Martin (Uncle Bob):

“A quantia de tempo gasto lendo código versus escrevendo é bem mais de 10 para 1. Então, fazer o código mais fácil de ler, o torna mais fácil de escrever.”

Padrão de Codificação

Pelo fato de todos no time XP estarem trabalhando juntos em cada parte do sistema, refatorando código e trocando de pares frequentemente, não se pode haver diferentes formas de codificação. Um padrão torna-se necessário, pois deve facilitar a comunicação entre o time, encorajando a posse coletiva, e evitando problemas na programação em pares e na refatoração.

O padrão deve ser estabelecido e concordado pelo time, pois faz com que todos o utilizem efetivamente. Caso contrário, a equipe ficará insatisfeita e evitará ao máximo seu uso. Sua existência vale muito mais do que a sua forma, porque ele pode ser revisto e evoluído de acordo com seu uso. Para descrever um padrão de código, diversos itens podem ser considerados:

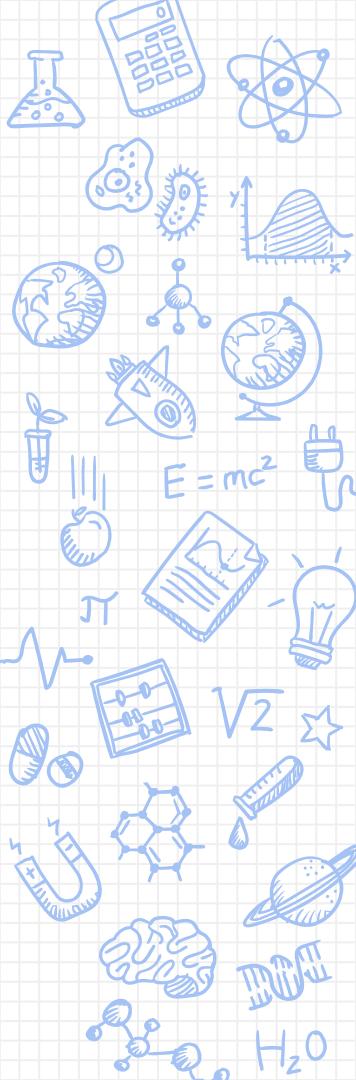
- Nomenclatura de variáveis;
- Nomenclatura de métodos;
- Nomenclatura de classes;
- Nomenclatura de pacotes;
- Indentação da tabulação;
- Indentação de chaves;
- Indentação das estruturas condicionais;
- Uso de parênteses em expressões;
- Uso de tratamento de exceções;
- Uso de estruturas de dados específicas;
- Importação de classes e bibliotecas;
- Diretrizes de Orientação a Objetos;
- Uso de padrões de projeto;
- Boas práticas de programação;
- Comentários de código;
- Comentários nos commits;
- Logging;
- Detalhes específicos da linguagem de programação;
- Detalhes específicos do ambiente de desenvolvimento.

Kent Beck:

“Eu não sou um excelente programador;
eu sou apenas um bom programador
com excelentes hábitos.”

O que é o TDD

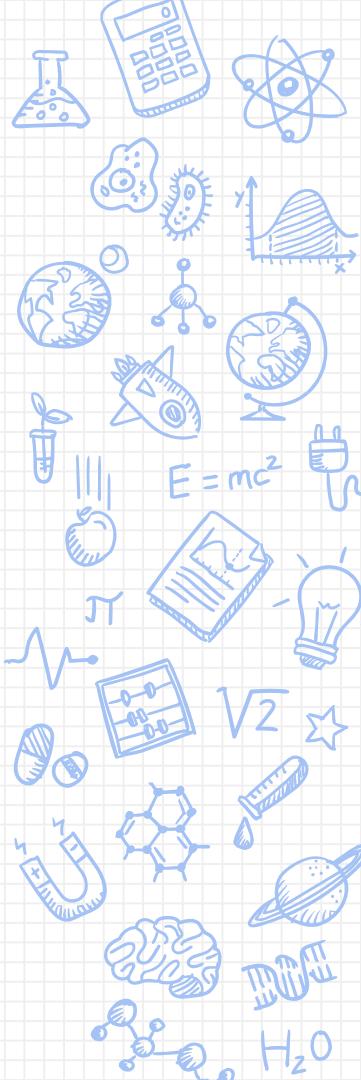
O TDD do inglês (Test Driven Development) em português (Desenvolvimento Guiado por Testes) é uma técnica para construção de software que guia seu desenvolvimento por meio da escrita de testes. Sua essência está em seu mantra: testar, codificar e refatorar. Essa técnica surgiu a partir da refatoração, dando base para guiar a codificação. TDD gera aprendizado, porque ajuda a aprender boas práticas de programação e a criar testes automatizados.

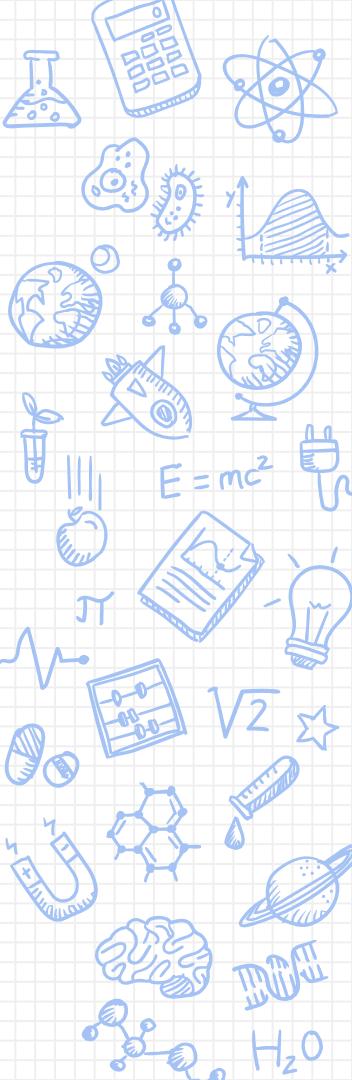


Porque Utilizar TDD (Test Driven Development)

O Desenvolvimento Orientado a Testes é um assunto rico e complexo, digno de um conteúdo inteiro só sobre esse assunto. Este conteúdo será somente uma visão geral que foca mais a justificação e a motivação do que os aspectos técnicos mais aprofundados da prática.

Na contabilidade. Os contadores produzem uma quantidade gigantesca de documentos com teor profundamente técnico, cheio de símbolos. Cada símbolo registrado em seus documentos deve estar correto, para que não se percam fortunas e, possivelmente, até vidas. Como os contadores asseguram que todos os símbolos estejam corretos?

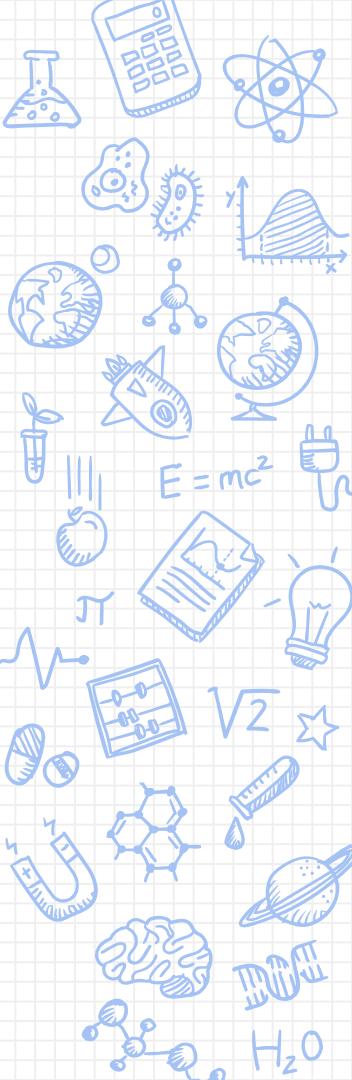




Método das Partidas Dobradas

Os contadores têm um método que foi inventado há mil anos. Chama-se **método das partidas dobradas**. Cada transação que eles registram em seus livros é inserida duas vezes: uma como crédito em um conjunto de contas e outra como débito complementar em outro conjunto de contas. Essas contas acabam sendo disponibilizadas em um único documento chamado balanço patrimonial, que subtrai a soma dos passivos e ações a partir da soma dos ativos. Essa diferença deve ser zero. Se não for zero, algum erro foi cometido.

Logo no início de sua formação, contadores são ensinados a registrar as transações uma por vez e sempre calcular o saldo. Isso lhes possibilita detectar rapidamente os erros. Eles aprendem a evitar um registro de um lote de transações entre as verificações de saldo, pois seria difícil identificar os erros. Essa prática é tão essencial para a contabilidade propriamente dita do dinheiro que se tornou um cânones em basicamente todas as partes do mundo.



TDD (Test Driven Development)

O **Desenvolvimento Orientado a Testes** é o **Método das Partidas Dobradas** para os programadores. Todo comportamento exigido é inserido duas vezes: uma vez como teste e depois como código de produção, para que o teste seja aprovado. As duas entradas são complementares, assim como os ativos são complementares aos passivos e ações. Quando executadas juntas, as duas entradas geram zero como resultado: zero falhas nos testes.

Os programadores que aprendem sobre TDD são ensinados a registrar todos os comportamentos, um de cada vez uma vez como um teste que falha e, depois, como um código de produção que passa nos testes. Isso lhes permite detectar rapidamente os erros. Eles aprendem a evitar escrever muito código de produção e a acrescentar um lote de testes, pois seria difícil identificar os erros.

Esses dois métodos, o método das partidas dobradas e o TDD, são equivalentes. Ambos têm o mesmo propósito: evitar erros em documentos de importância crucial, nos quais todos os símbolos devem ser registrados de forma correta. A despeito do grau de relevância da programação em nossa sociedade, ainda não infundimos o TDD como prática de lei. Mas, considerando que vidas e fortunas já foram perdidas pelos caprichos de softwares mal desenvolvidos, o quanto estamos longe de integrar essa prática?

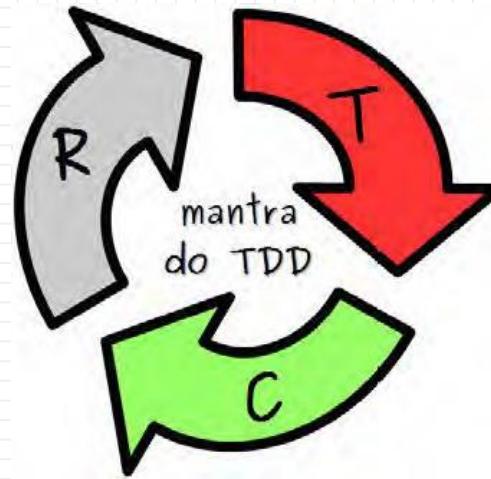
As 3 regras do TDD

O TDD pode ser descrito em três regras simples:

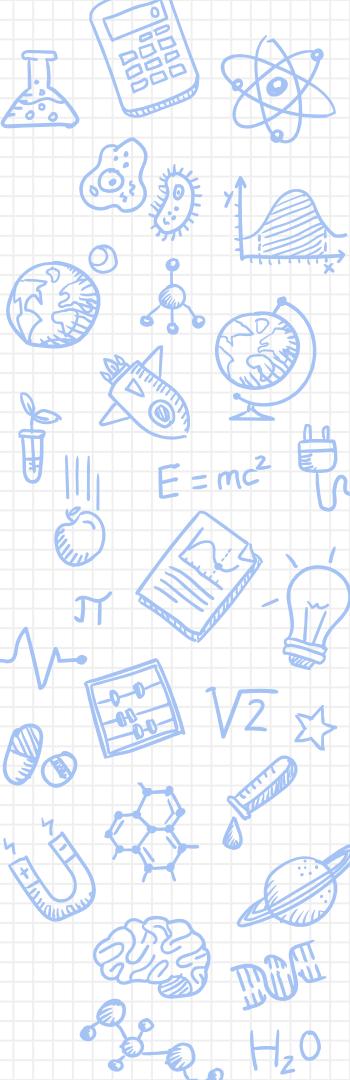
- Não escreva nenhum código de produção antes de elaborar um teste que falhou devido à falta desse mesmo código.
- Não escreva mais testes do que o suficiente para identificação da falha e falhas na compilação ainda contam como falhas.
- Não escreva mais códigos de produção do que o suficiente para passar nos testes.

O processo de refatoração está intimamente relacionado com as 3 Regras do TDD e é conhecido como ciclo Vermelho/Verde/Refatore:

1. Primeiro, escreva um teste que falhe.
2. Depois, faça com que o teste passe.
3. Em seguida, limpe o código.
4. Volte para a etapa 1.



Mantra do TDD



A ideia aqui é que escrever um código que funcione e escrever um código que seja limpo são duas dimensões separadas da programação. Tentar controlar ambas as dimensões ao mesmo tempo é, na melhor das hipóteses, complicado, então, as separamos em duas atividades distintas.

Em outras palavras, já é custoso o bastante fazer o código funcionar, imagine fazer sua limpeza. Desse modo, primeiro nos concentramos em fazer com que o código funcione por meio de quaisquer ideias loucas vindas de nossas cabeças. Em seguida, depois que estiver funcionando, com a aprovação dos testes, limpamos a bagunça que fizemos. Isso deixa claro que a refatoração é um processo contínuo, e não um processo executado de forma planejada. Não ficamos bagunçando as coisas por dias e depois tentamos limpar a bagunça. Ao contrário, fazemos o mínimo de bagunça, durante um minuto ou dois, e depois a limpamos.

A palavra Refatoração nunca deve aparecer em um cronograma. Esse não é o tipo de atividade que aparece em um planejamento. Não reservamos tempo para refatorar. A refatoração simplesmente faz parte da nossa abordagem diária para escrever o software.

Martin Fowler:

“Qualquer tolo pode escrever código que o computador possa entender. Bons programadores escrevem código que humanos possam entender.”

Refatoração

Uma refatoração é uma mudança feita na estrutura interna do software que o faz ficar mais fácil de entender e mais barato de modificá-lo, sem mudar seu comportamento observável. Obter essa simplicidade não é algo fácil, por isso a fazemos por meio de uma série de pequenas alterações, uma a uma sem alterar esse comportamento.

Sabemos que alguns tipos de código são difíceis para alterar, como: código com lógica duplicada ou complexa, código ilegível ou código com comportamento adicional e desnecessário. Estes comumente possuem muitos bad smells (maus cheiros), também chamados code smells (cheiros de código).

Quais são seus benefícios?

- Economiza tempo e aumenta a qualidade;
- Melhora o design do software;
- O código torna-se mais legível e reutilizável;
- Fica mais fácil encontrar defeitos;
- Ajuda você a programar mais rápido.

Alguns exemplos de code smells a serem refatorados são:

- Uma classe ou método muito longo;
- Uma classe que utiliza muitos métodos de outra classe;
- Uma classe que faz muito pouco;
- Subclasses muito semelhantes;
- Nome de variável não comunicativo;
- O não tratamento de uma exceção;
- Código duplicado;
- Código inútil.

Martin Fowler:

“Integração contínua não livra os bugs,
mas os tornam dramaticamente mais
fáceis de encontrar e de remover.”

Integração Contínua

Integração contínua (CI) é uma prática, na qual o código que está sendo desenvolvido pelo time é integrado, versionado, construído e verificado diversas vezes ao dia em um ambiente dedicado. Os programadores XP devem integrar e fazer commit em somente uma versão no repositório de código. Cada integração é verificada por um build com testes automatizados, detectando erros o mais cedo possível. Essa prática gera sinergia com as outras do XP. Os builds das pequenas entregas são construídos por meio da CI. Ela auxilia na refatoração, pois verificará o build automaticamente a cada commit realizado no repositório. Em cada um, o padrão de codificação pode ser verificado automaticamente.

A integração contínua é um processo simples que traz muitos benefícios:

- Aumenta o feedback, a comunicação na equipe e a moral do time;
- Todos veem o que está acontecendo;
- Previne e descobre os problemas de integração mais cedo;
- Reduz riscos e evita a baixa qualidade;
- Não é necessário um integrador dedicado para a equipe;
- Todos têm acesso à versão mais atualizada;
- Auxilia na reutilização de código: os desenvolvedores terão sempre o código mais atualizado;
- Evita problemas de merge, fazendo pequenas integrações ao longo do tempo.

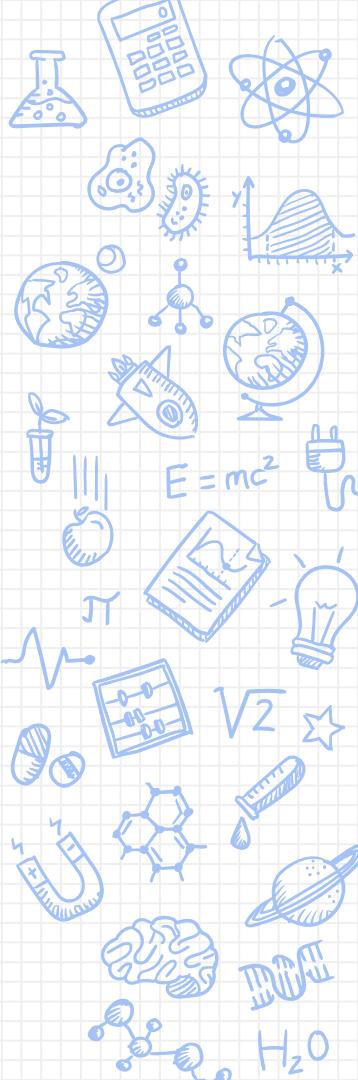
Entrega Contínua

Entrega contínua (CD) é um conjunto de princípios e práticas com o objetivo de compilar, testar e liberar software ao cliente de forma mais rápida e frequente. Essa abordagem encaixa-se muito bem com as práticas do XP de liberação constante de pequenas versões e de integração contínua. Podemos dizer que entrega contínua (CD) também é uma prática extrema, pois está bastante alinhada com o primeiro princípio do Manifesto Ágil: *nossa maior prioridade é satisfazer o cliente por meio da entrega contínua e adiantada de software com valor agregado.*

Princípios da entrega contínua de software: Seguindo tais princípios, você terá um processo aprimorado de entrega contínua.

- Criar um processo de confiabilidade e repetitividade de entrega de versão;
- Automatize quase tudo;
- Mantenha tudo sob o controle de versão;
- Se é difícil, faça com mais frequência e amenize o sofrimento;
- A qualidade deve estar presente desde o início;
- “Pronto” quer dizer “entregue”;
- Todos são responsáveis pelo processo de entrega;
- Melhoria contínua.

Comparação entre SCRUM e XP



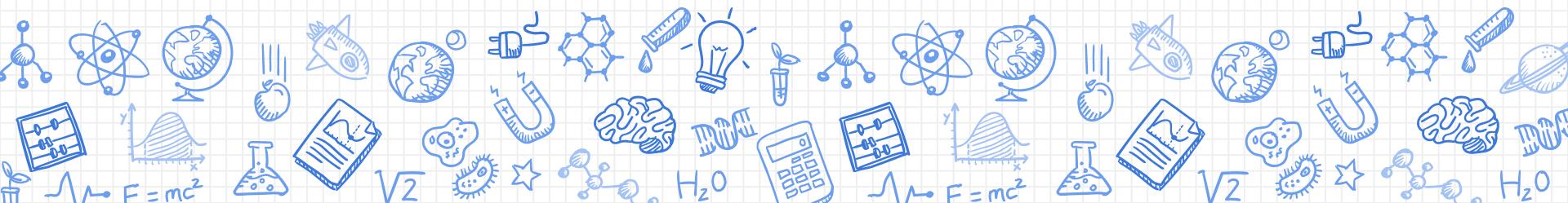
O SCRUM, como é um método ágil, e os métodos ágeis acabam tendo várias semelhanças e pontos em comum, ele tem um forte relacionamento com o XP como por exemplo, eles tem a raiz fundamentada no manifesto ágil.

O SCRUM é uma forma de gestão ampla para projetos que não depende da área de conhecimento. Já o XP tem sua aplicação mais restrita, focada basicamente no mundo de desenvolvimento de softwares.

Entretanto, quando usamos o SCRUM como forma de gestão e para criação de software, muitas das práticas contidas no XP são de grande competência, como por exemplo a criação de testes automatizados ou o uso de refatoração de código, o uso dessas práticas nos trás um ganho de qualidade e garantia que a manutenção futura seja simplificada.

Outras Metodologias

Algumas metodologias que podem ser complementares ao SCRUM e XP.



Kanban

Kanban é um sistema de controle e gestão do fluxo de produção em empresas e projetos que usa de cartões coloridos (post-its) e também recebe o nome de gestão visual, em razão do uso de cores como sinalizadores.



Lean

A **Metodologia Lean** é anterior ao manifesto ágil, tendo surgido no Japão do pós-guerra, em indústrias automobilísticas que desejavam ser mais produtivas. Por compreender modelos de processos enxutos, com poucos desperdícios, essa abordagem é compatível com as metodologias ágeis. Essa, inclusive, é uma abordagem excelente para aplicar no chão de fábrica: o **lean manufacturing** é um exemplo bem-sucedido disto. Há também o **lean construction**, específico para área de construção civil!

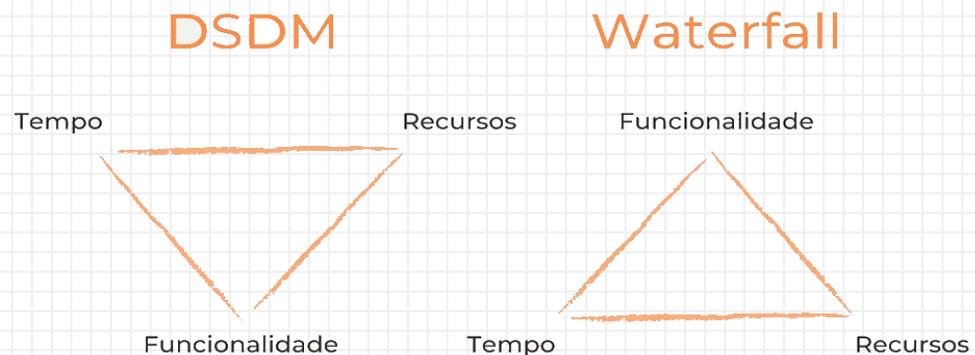


DSDM (Dynamic Systems Development Method)

O DSDM (Metodologia de Desenvolvimento de Sistemas Dinâmicos) é uma metodologia antigamente conhecida como RAF (Rapid Application Development). O DSDM fornece uma fundação para implementação da metodologia ágil no projeto, passando por planejamento, gerenciamento, execução e dimensionamento.

Ao aplicar o DSDM, a empresa baseia-se em 6 princípios fundamentais:

- Valor
- Envolvimento ativo do usuário
- Equipes capacitadas
- Entregas frequente
- Testes integrado
- Colaboração do cliente



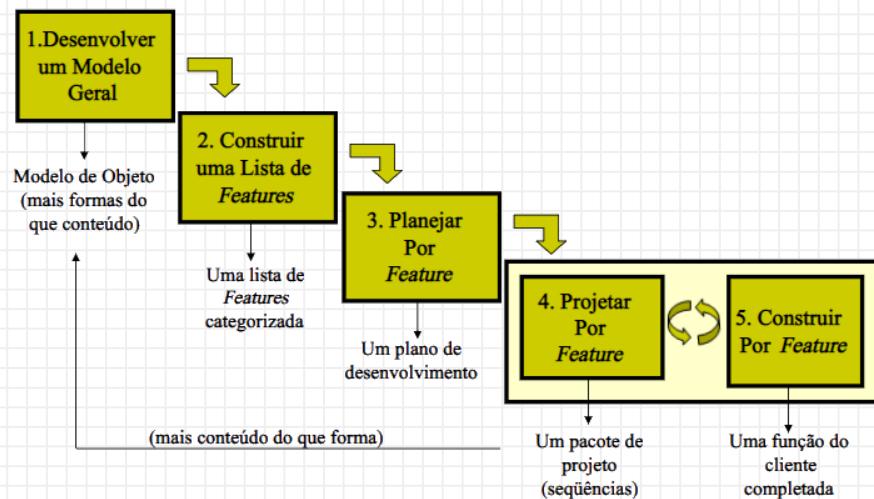
Um elemento interessante do DSDM é sua abordagem mais comercial no que diz respeito à entrega. Por isso, segue uma fórmula que busca 80% de implantação do sistema ou solução em 20% do tempo.

FDD (Feature Driven Development)

O FDD (Feature Driven Development) trata-se do desenvolvimento orientado a funcionalidades. É um processo de iteração mais curto, cuja estrutura está mais ligada à forma que o modelo ágil segue. Assim, as tarefas são decompostas em pequenas funcionalidades, pulverizando o trabalho. As vantagens desta forma de gestão ágil se originam principalmente do fato de cada feature ser uma unidade mínima do projeto total. Isso faz com que cada tarefa, descrição, teste e alteração seja sempre minimalista, dando agilidade ao processo e gastando menos tempo e recursos humanos.

No FDD são aplicados 5 princípios, sendo eles:

- 1. Desenvolver um Modelo Abrangente
- 2. Construir uma Lista de Funcionalidades
- 3. Planejar por Funcionalidade
- 4. Detalhar por Funcionalidade
- 5. Construir por Funcionalidade



ASD (Adaptive Software Development)

O ASD é **A**daptive **S**oftware **D**evelopment (ou Desenvolvimento Adaptativo de Softwares), com foco no desenvolvimento de soluções mais complexas. Com o ASD, o objetivo é evitar o caos nas entregas, apostando na colaboração humana e na auto-organização.

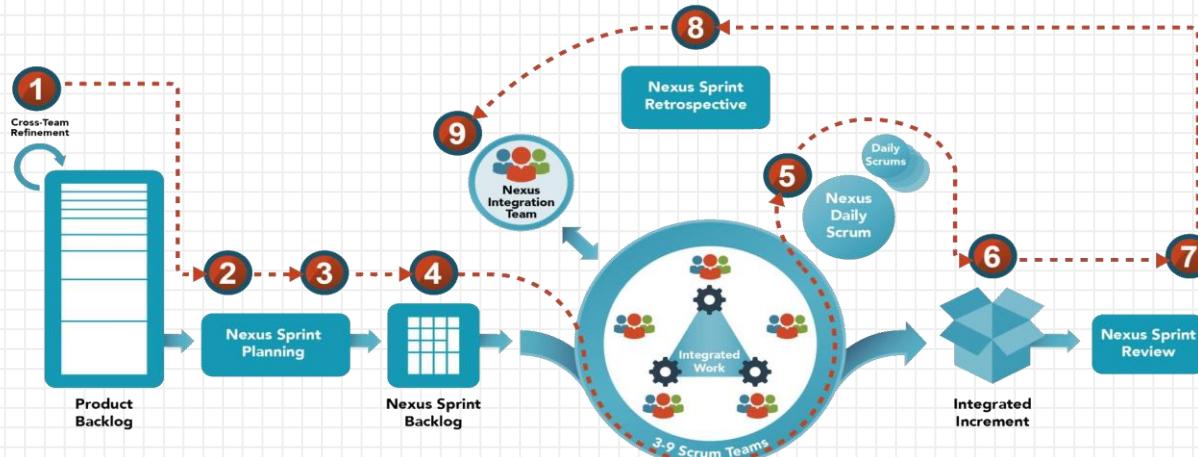
Baseia-se em 6 princípios:

- Orientado a missões
- Baseado em componentes
- Iterativo
- Prazos pré-fixados
- Tolerância a mudanças
- Orientado a riscos



Nexus Framework

O Nexus é um framework que possibilita implementar o Scrum em larga escala em uma empresa, unindo de três a nove times de Scrum para trabalhar na entrega de um único produto ou resultado no final de cada Sprint. O Nexus também prevê três responsáveis por entrelaçar as equipes, o chamado Time de Integração, que é composto por: um Product Owner, Scrum Master e pelo menos um membro de cada time Scrum. A composição desse time, entretanto, não é fixa: ela pode mudar ao longo do tempo de forma a refletir as necessidades do Nexus. As atividades comuns dessa equipe incluem: mentoria, consultoria, e destacar as dependências e problemas entre os times.

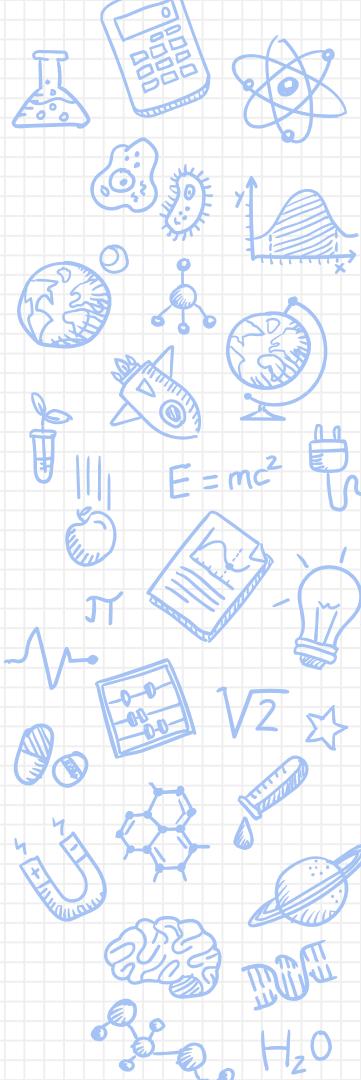
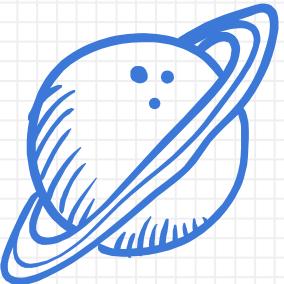
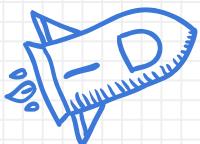


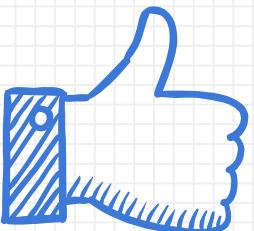


Conclusão

A metodologia ágil é um conceito fascinante ao desenvolvimento de produtos e soluções. Trata-se de uma maneira eficiente de revolucionar a rotina operacional, integrando o time de desenvolvimento com os clientes. Isso resulta em uma experiência mais gratificante e enriquecedora para todos os envolvidos, bem como em uma melhor qualidade de entrega.

O que resulta, em melhores resultados para a empresa, sua aplicação adequada, no entanto, depende de uma série de fatores, um deles é a tecnologia, porém, não existe bala de prata quando se fala em gestão de projetos. Você pode e deve adaptar as coisas conforme a realidade da sua empresa.





Obrigado!

Alguma Pergunta?

Você pode me encontrar em:

- willianbrito05@gmail.com
- www.linkedin.com/in/willian-ferreira-brito
- github.com/Willian-Brito