METODOLOGIAS ÁGEIS

AULA 2

Prof.ª Mariana Lucia Kalluf Dakkache Leal

CONVERSA INICIAL

Os métodos ágeis são caracterizados por sua flexibilidade e adaptabilidade, e encorajam o envolvimento dos clientes e a entrega frequente de *software* funcional. Nesta abordagem, veremos alguns dos métodos ágeis mais conhecidos. Exploraremos as metodologias:

- Extreme Programming (XP);
- Feature-Driven Development (FDD);
- Test-Driven Development (TDD);
- Crystal;
- Scrum.

Entenderemos suas origens, princípios, processos e benefícios para equipes de desenvolvimento ágil. Com o conhecimento adquirido nesta abordagem será possível escolher o método mais adequado às necessidades e características de seu projeto.

TEMA 1 – EXTREME PROGRAMMING (XP)

Extreme Programming (XP), criada em 1996 por Kent Beck, Ward Cunningham e outros desenvolvedores, tornou-se uma das metodologias mais populares no desenvolvimento de *software* ágil.

De acordo com Beck (2000), "extreme programming é uma metodologia ágil de desenvolvimento de *software* que enfatiza a simplicidade, a comunicação e o *feedback* rápido. Ela visa produzir *software* de alta qualidade de forma rápida e eficiente, mantendo um ritmo sustentável de trabalho."

Crédito: ktsdesign/Shutterstock.

Esta metodologia é dinâmica e versátil, mas exige que a equipe de desenvolvimento seja rigorosamente disciplinada, pois possui uma série de valores, princípios e práticas que precisam ser seguidos para que a eficácia no projeto seja atingida.

1.1 Valores do XP

> Extreme Programming se concentra em quatro valores fundamentais: comunicação, simplicidade, *feedback* e coragem. A comunicação é importante porque a programação é uma atividade social e colaborativa. A simplicidade é importante porque sistemas simples são mais fáceis de manter e adaptar. O *feedback* é importante porque permite ajustar continuamente o projeto às necessidades do cliente. E a coragem é importante porque o XP envolve constantes mudanças e riscos. (Beck, 2000)

Crédito: Unitone Vector/Shutterstock.

O método Extreme Programming (XP) está focado em quatro valores fundamentais que orientam a equipe do projeto.

1. **Comunicação**: é a chave para a resolução de conflitos, para a solução de problemas. Por meio dela, é possível encontrar possibilidades de melhorias no projeto.
2. **Simplicidade**: o código-fonte deve ser sempre organizado e de fácil compreensão. Não devem ser mantidas funcionalidades desnecessárias.
3. ***Feedback***: a equipe do projeto tem retornos constantemente, seja dos testes, seja dos outros colegas ou dos clientes. Com esses *feedbacks* é possível realizar os ajustes com maior celeridade.

4. **Coragem**: a equipe do projeto precisa tomar decisões difíceis, sair da zona de conforto e testar novas ideias, permitindo, com isso, que decisões assertivas para o projeto sejam tomadas.

Esses valores norteiam as práticas do XP, auxiliando no trabalho colaborativo, eficaz e com foco no cliente da equipe do projeto.

1.2 Princípios do XP

> Ao seguir os princípios do XP, a equipe de desenvolvimento de *software* pode produzir *software* de alta qualidade que atenda às necessidades do cliente de forma eficaz. Estes princípios incluem *feedback* rápido, simplicidade, mudança incremental, qualidade de trabalho, pequenos passos, melhoria contínua, diversidade e reflexão constante. (Chen, 2008)

Crédito: VectorMine/Shutterstock.

Na metodologia XP, existem 12 princípios fundamentais para o desenvolvimento ágil de *software*.

1. ***Feedback* rápido**: o *feedback* dos clientes e usuários finais norteia a equipe de desenvolvimento para realizar os ajustes necessários em tempo real.
2. **Presunção de simplicidade**: buscar sempre as soluções mais simples para entregar um produto de qualidade; é mais fácil entender, manter e atualizar um código mais simples e organizado.
3. **Mudança incremental**: o desenvolvimento deve ser realizado de maneira incremental e, a cada etapa, deve entregar um *software* funcional; a equipe deve estar pronta para mudanças sempre que exigido.
4. **Trabalho de alta qualidade**: a equipe do projeto deve garantir que o sistema seja testado, organizado e documentado.

5. **Pequenos passos**: devem ser definidos pequenos ciclos de entrega do produto, para garantir que seja entregue valor ao cliente com maior rapidez e, com isso, que os ajustes sejam realizados à medida que são identificados.
6. **Melhoria contínua**: a equipe do projeto deve buscar constantemente aprimoramento no processo de desenvolvimento do *software,* com o intuito de melhorar a qualidade, a *performance* e a eficiência do projeto.
7. **Diversidade:** a equipe deve ter várias habilidades, experiências e visões para agregar ao produto soluções criativas e inovadoras.
8. **Reflexão constante**: a cada término de iteração, a equipe deve identificar o que funcionou bem e o que precisa melhorar e, com isso, ajustar o processo para as próximas iterações.
9. **Comunicação**: a comunicação entre a equipe e com o cliente deve ser clara e direta para o bom entendimento das necessidades e objetivos do projeto.
10. **Respeito**: as opiniões devem ser respeitadas para que o ambiente de trabalho seja agradável e produtivo.
11. **Coragem**: a equipe deve ter coragem para fazer os questionamentos necessários, tomar decisões difíceis e realizar mudanças de projeto, mesmo que audaciosas.
12. **Presença/disponibilidade do cliente**: durante o projeto é primordial que o cliente esteja presente para orientar e entregar seu *feedback*.

1.3 Práticas do XP

> O XP define várias práticas que as equipes de desenvolvimento de *software* devem seguir para alcançar seus objetivos. Estas práticas incluem, por exemplo, o desenvolvimento orientado a testes, a programação em par, a integração contínua e o planejamento em conjunto com o cliente. (Larman, 2005)

Crédito: LadadikArt/Shutterstock.

O XP possui várias práticas definidas. Dentre elas, temos:

- **jogo do planejamento –** reunião entre o cliente e a equipe de desenvolvimento para definir quais funcionalidades serão desenvolvidas em um determinado período;
- **pequenos *releases* –** *releases* curtos e frequentes para entregar funcionalidades que agreguem valor ao cliente;
- **metáfora –** explicar o *software* por meio de metáforas para facilitar o entendimento da equipe;
- **projeto simples –** construir um *software* claro e objetivo para facilitar sua manutenção e evolução;
- **testes –** criar testes automatizados para garantir a qualidade do *software*;
- **programação em par –** adotar o *pair programming* explicado no Tópico 5 do capítulo 1;
- **refatoração –** adotar a refatoração explicada no Tópico 5 do capítulo 1;
- **integração contínua –** integração contínua do código desenvolvido para que seja testado e validado com frequência;
- **ritmo sustentável –** definir um ritmo de trabalho que seja saudável para a equipe;
- **cliente presente –** trazer o cliente para trabalhar próximo à equipe;
- **programação em duplas rotativas –** mudar as duplas de programação com frequência.
- **propriedade coletiva do código –** todos os desenvolvedores são responsáveis pelo código e podem alterá-lo sempre que necessário;

- **padrões de codificação –** definir um padrão de codificação a ser aplicado pela equipe de desenvolvimento para que seja mais fácil sua manutenção;
- **integração do usuário –** utilizar as histórias de usuário (*user stories*) para documentar as funcionalidades;
- **refatoração em larga escala –** melhorar a qualidade do código em grande escala; pode abranger alterações consideráveis;
- **programe o que você precisa hoje –** o desenvolvedor deve codificar somente o que foi acordado para aquele momento; evitar desenvolver funcionalidades que podem ser necessárias no futuro;
- **uso de metodologias ágeis –** outras metodologias ágeis, Scrum, Kanban e Lean podem ser utilizadas em conjunto com o XP.

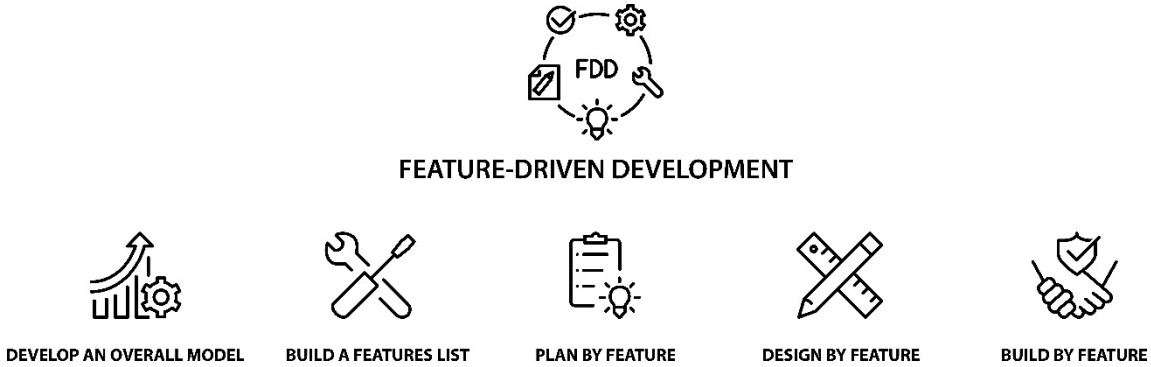
As práticas da Extreme Programming (XP) podem ser adequadas e ajustadas de acordo com as necessidades da equipe ou do projeto. A equipe pode trabalhar com as práticas que forem mais aderentes ao seu processo de desenvolvimento, buscando sempre entregar valor ao cliente.

TEMA 2 – DESENVOLVIMENTO DIRIGIDO POR FUNCIONALIDADES (FDD)

O Feature-Driven Development (FDD) foi criado por Jeff De Luca e Peter Coad na década de 1990. Foi apresentado pela primeira vez no livro *Java Modeling in Color with UML*, escrito por Peter Coad, Eric Lefebvre e Jeff De Luca. Baseia-se em práticas específicas que enfatizam a entrega rápida de funcionalidades de valor para o cliente.

Para Pereira (2016):

> O FDD é uma abordagem mais estruturada e orientada a processo para o desenvolvimento ágil. Ele define um processo de cinco etapas para o desenvolvimento de *software* e oferece orientação detalhada sobre cada etapa. O FDD é uma metodologia altamente adaptável, que pode ser usada em projetos de qualquer tamanho ou complexidade.

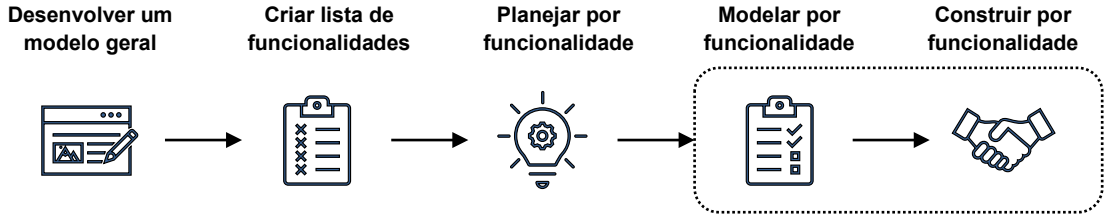


Crédito: Vector street/Shutterstock.

2.1 Etapas do FDD

O processo do Feature-Driven Development (FDD) é dividido em cinco etapas (Figura 1).

Figura 1 – Etapas do FDD



Etapa 1 – Desenvolvimento do modelo geral

O objetivo desta etapa é conhecer o sistema; para isso, a equipe de desenvolvimento:

- constrói um modelo que apresenta a estrutura geral do sistema e os relacionamentos dos objetos de negócio;
- identifica os requisitos funcionais do sistema.

Etapa 2 – Criar lista de funcionalidades

Nesta etapa deve-se elaborar, em conjunto com o cliente, a lista de funcionalidades do sistema e fazer sua priorização. Essa lista é atualizada no decorrer do projeto.

Etapa 3 – Planejar por funcionalidade

Nesta etapa, devem ser planejadas as tarefas, a estimativa de tempo e os recursos necessários para execução de cada uma das funcionalidades.

Etapa 4 – Modelar por funcionalidade

Neste momento, deve ser construído o *design* detalhado de cada uma das funcionalidades. Nessa documentação deve ser incluída a especificação da arquitetura, diagramas de classe e de sequência.

Etapa 5 – Construir por funcionalidade

Na última etapa, as funcionalidades são implementadas e testadas em pequenos incrementos. Esta etapa se repete até que todas as funcionalidades tenham sido implementadas, testadas e integradas ao sistema.

> O FDD (Feature-Driven Development) é um processo iterativo e incremental que se concentra em desenvolver funcionalidades incrementais, pequenas e tangíveis, em vez de desenvolver grandes sistemas em uma única vez. O FDD enfatiza a importância das etapas

de planejamento, *design* e construção em pequenas iterações, o que ajuda a garantir que o sistema seja desenvolvido com alta qualidade e entregue dentro do prazo estipulado. (Sachetto; Turbias, 2014)

2.2 Características do FDD

Algumas das principais características do FDD incluem:

- **foco nas funcionalidades** – desenvolver pequenas funcionalidades que agreguem valor ao cliente e a cada iteração complementar as funcionalidades do produto;
- **equipes multidisciplinares** – a equipe deve ser composta de vários perfis diferentes para garantir uma visão completa das necessidades do projeto;
- **processo iterativo e incremental** – define ciclos de desenvolvimento curtos e que fazem com que o sistema evolua a cada entrega;
- **ênfase no planejamento** – o FDD enfatiza a importância do planejamento, especialmente quanto à definição de funcionalidades, *design* e construção;
- **inspeções de código** – para garantir a qualidade do *software* são realizadas inspeções no código e, com isso, os problemas são encontrados e solucionados proativamente;
- **ênfase na comunicação** – realização de reuniões diárias e outras formas de interação da equipe certificam que todos os integrantes da equipe estão trabalhando em sintonia e com os mesmos objetivos;
- **entrega contínua** – garantir que as entregas forneçam um produto funcional.

Crédito: Diyajyoti/Shutterstock.

2.3 Vantagens e desvantagens do FDD

O Feature-Driven Development (FDD) possui diversas vantagens e desvantagens que devemos considerar, antes de optar por essa metodologia.

2.3.1 Vantagens

> O FDD é uma abordagem pragmática e altamente visível para desenvolvimento de *software* que tem benefícios inegáveis para equipes de todos os tamanhos. O foco em entregar recursos e gerenciar processos de desenvolvimento tem sido um sucesso comprovado em muitas organizações em todo o mundo. (De Luca, 2002)

A seguir explicaremos algumas das vantagens da utilização do FDD.

- **Abordagem orientada a objetos**: com a abordagem orientada a objetos do FDD é possível criar a modelagem dos objetos de negócio com maior eficiência e adaptar o desenvolvimento às expectativas e necessidades do cliente.
- **Equipe multidisciplinar**: encoraja o trabalho em conjunto de profissionais de diferentes áreas de especialização para entregas de *software* com qualidade e no prazo definido.
- **Entrega contínua de recursos**: ressalta a entrega contínua de versões testáveis do projeto; com isso é possível realizar os ajustes de acordo com os *feedbacks* do cliente constantemente.
- **Adaptabilidade**: a metodologia pode ser adaptada às necessidades do projeto.

2.3.2 Desvantagens

"O FDD pode ser menos adequado para projetos de grande escala, já que pode haver dificuldades em manter a visibilidade e a clareza em projetos complexos com muitas equipes e recursos." (Chisolm, 2007).

Citaremos agora algumas das desvantagens da utilização do FDD.

- **Falta de foco em documentação**: apesar de a documentação ser clara e objetiva, não é detalhada, o que pode causar alguns problemas em projetos que demandem uma documentação com mais informação.

- **Dependência de habilidades técnicas**: a equipe do projeto deve possuir as habilidades técnicas necessárias para modelar os objetos de negócio e implementar os recursos com qualidade e eficiência.
- **Necessidade de planejamento detalhado**: ter um planejamento detalhado, em um projeto que os requisitos podem mudar com frequência, é o desafio da metodologia.
- **Potencial para conflitos**: como a equipe do projeto e o cliente estão em contato constante, podem ocorrer desentendimentos entre eles, principalmente quando o cliente tiver expectativas equivocadas quanto ao tempo ou custo do projeto.

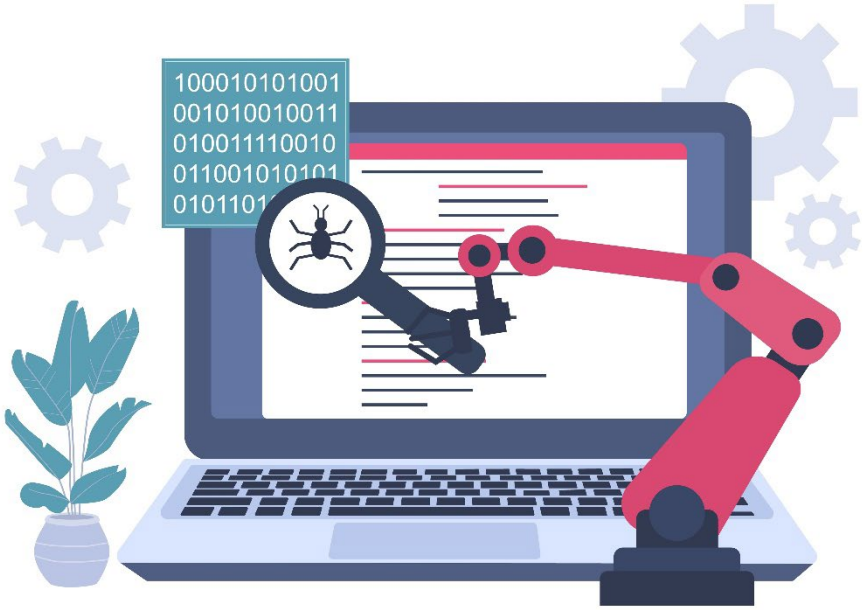
TEMA 3 – DESENVOLVIMENTO ORIENTADO A TESTES (TDD)

"O TDD é uma prática de desenvolvimento de *software* que foi criada com o objetivo de garantir a qualidade do código produzido, permitindo que os desenvolvedores escrevam testes antes mesmo de escreverem o código em si" (Freeman; Freeman, 2014).

Nos anos 1990, Kent Beck regulamentou e difundiu o Test-Driven Development (TDD) como uma prática, participante da metodologia Extreme Programming (XP), que tem por objetivo escrever um teste, falhar propositalmente, escrever o código para passar no teste e refatorar o código.

Essa prática popularizou-se por minimizar os *bugs* e assegurar a qualidade do código. Atualmente, o TDD é frequentemente utilizado com outras práticas ágeis, com o objetivo de intensificar o ciclo de desenvolvimento e elevar a qualidade do *software*.

O método de programação em pares (*pair programming*), visto no capítulo 1 deste material, auxiliou no desenvolvimento do conceito de TDD. A qualidade do *software* aumenta e o tempo de desenvolvimento reduz quando dois programadores trabalham em conjunto no mesmo código.



Crédito: Igogosha/Shutterstock.

O Test-Driven Development (TDD) propõe que, mesmo antes de implementar o código de cada funcionalidade, devem ser escritos os testes automatizados, e o desenvolvedor fará a verificação do *software* por meio dos testes.

> O TDD é uma técnica de programação que ajuda a melhorar a qualidade do *software*, reduzindo o número de defeitos e facilitando a manutenção do código. Ao escrever testes antes de escrever o próprio código, o desenvolvedor consegue validar continuamente se o *software* está funcionando corretamente e se comportando conforme o esperado. Isso resulta em um código mais seguro, confiável e fácil de manter. (Caroli, 2019)

3.1 Características do TDD

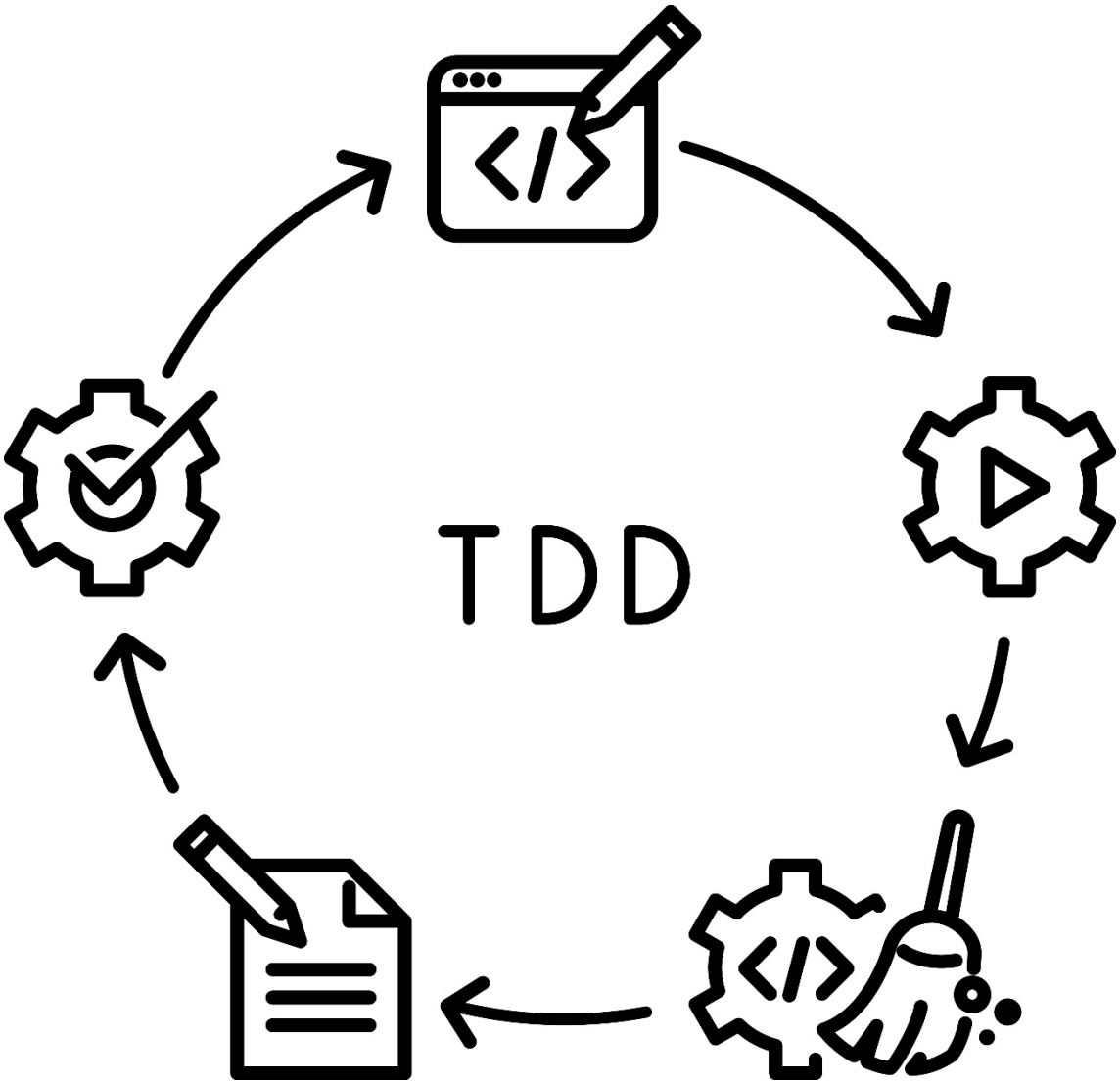
Algumas das principais características do Test-Driven Development (TDD) são:

- **foco em testes** – a escrita dos testes automatizados, primeira etapa do desenvolvimento do *software*, auxilia na especificação das funcionalidades do projeto e certifica que está funcionando adequadamente;
- **desenvolvimento iterativo** – os testes automatizados de cada funcionalidade são escritos e realizados em iterações curtas que

garantem a identificação e correções dos erros encontrados com mais celeridade;

- **código mais seguro** – as funcionalidades são minuciosamente testadas antes de serem liberadas;
- **refatoração constante** – a frequente refatoração do código garante um código limpo e de fácil manutenção; essa prática contribui para o aprimoramento progressivo da qualidade do código;
- **equipe mais produtiva** – os erros são identificados precocemente, o que garante a qualidade do código desenvolvido e reduz o tempo gasto no desenvolvimento das funcionalidades.

3.2 Como funciona o TDD



Crédito: Vector street/Shutterstock.

Para garantir que o código criado está de acordo com os testes escritos, o processo de desenvolvimento com TDD normalmente segue cinco passos.

1. **Escrever um teste**. O programador escreve um teste automatizado que descreve a funcionalidade desejada. Inicialmente, esse teste irá falhar, pois não temos a funcionalidade codificada.
2. **Escrever o código**. O programador codifica o mínimo necessário para o teste ser executado com sucesso. Nesse momento, não deverá implementar nenhuma funcionalidade adicional.
3. **Executar o teste.** O programador verifica se o teste foi bem-sucedido, caso contrário retorna para o passo 2 e faz as correções no código.
4. **Refatorar o código**. Se o teste foi aprovado, o programador, sem alterar a funcionalidade do código, pode refatorá-lo para melhorar sua qualidade.
5. **Repetir o processo**.

   A cada funcionalidade do projeto, esses passos devem se repetir.

3.3 Vantagens e desvantagens do TDD

O Test-Driven Development (TDD) possui diversas vantagens e desvantagens que devemos considerar, antes de optar por essa metodologia.

3.3.1 Vantagens do TDD

De acordo com (Jeffries, 2003): "Testes são tão importantes quanto o próprio código. A abordagem TDD garante que o código esteja funcionando corretamente desde o início, o que resulta em menos erros e um código mais confiável".

Apresentaremos a seguir mais algumas vantagens do TDD.

- **Redução do número de defeitos**: com essa prática, o programador pode validar constantemente se o funcionamento do *software* está de acordo com o esperado e, com isso, reduzir erros.
- **Melhoria na qualidade do código**: induz o programador a escrever um código em módulos e a tratar os testes como parte do desenvolvimento.
- **Facilidade em detectar regressões**: a criação de testes de unidade para cada uma das funcionalidades permite detectar mais facilmente quando

uma funcionalidade, que antes funcionava normalmente, para de funcionar.

- **Maior confiança no *software***: com uma cobertura de testes abrangente, os erros são mais facilmente detectados e, com isso, sua manutenção é mais simples.

3.3.2 Desvantagens do TDD

Maxim (2019, p. 13) menciona que: "O TDD pode levar a uma ênfase excessiva na cobertura de teste e a testes mal escritos, que não detectam erros críticos. Além disso, pode ser demorado e aumentar o custo do desenvolvimento".

Temos a seguir mais algumas desvantagens da utilização do TDD.

- **Possível aumento no tempo de desenvolvimento**: cada uma das funcionalidades demanda a criação de testes automatizados, podendo aumentar o tempo de desenvolvimento.
- **Possível aumento da complexidade do código**: a criação de testes para as funcionalidades pode tornar o código complexo, devido às validações.
- **Pode requerer maior habilidade do desenvolvedor**: a habilidade em escrever os testes automatizados pode ser um obstáculo para programadores com menos experiência.

TEMA 4 – CRYSTAL

De acordo com Cockburn (2002, p. 14), "Crystal é uma metodologia ágil que se concentra em diferentes aspectos do desenvolvimento de *software*, dependendo das características do projeto e da equipe. Ela oferece um conjunto de diretrizes flexíveis e adaptáveis que podem ser ajustadas para diferentes contextos e objetivos".

Crystal é uma família de metodologias com características similares. Foi criada, no fim dos anos 1990, por Alistair Cockburn, que antes estudou diversas metodologias de desenvolvimento de *software*, tais como o método RUP (Rational Unified Process) e a metodologia Extreme Programming (XP), mas identificou que essas abordagens possuíam muitas regras e não eram compatíveis com diferentes projetos e equipes. Com isso, criou a metodologia

Crystal, que salienta a simplicidade, o trabalho em equipe, a comunicação entre equipe e cliente e a capacidade de adaptação.

> A Crystal Methodology entende que cada projeto pode exigir um conjunto de políticas, práticas e processos ligeiramente adaptados para atender às características exclusivas do projeto. A família de metodologias usa cores diferentes com base no "peso" para determinar qual metodologia usar. O uso da palavra *cristal* vem da pedra preciosa, onde as várias "faces" representam os princípios e valores fundamentais subjacentes. As faces são uma representação de técnicas, ferramentas, padrões e funções listadas. (Project Management Institute, 2018)

Seu objetivo é disponibilizar uma coleção de princípios flexíveis e ajustáveis para o desenvolvimento de *software* que possam ser empregados em variados contextos e projetos. A metodologia fundamenta-se nos sete pilares que veremos a seguir.

4.1 Pilares

Os sete pilares, também conhecidos como *sete propriedades* ou *sete aspectos fundamentais* da metodologia Crystal, são:

1. **comunicação** – muito valorizada na metodologia Crystal, a comunicação direta e objetiva entre a equipe do projeto e o cliente ajuda a alinhar as expectativas e a reduzir a chance de erros e mal-entendidos;
2. **refinamento constante** – a melhoria contínua do processo de desenvolvimento é primordial; para isso, a metodologia Crystal propõe revisar e adaptar frequentemente as práticas empregadas pela equipe e garantir que as metodologias utilizadas estejam de acordo com as necessidades de cada projeto;
3. **entrega frequente** – realizar entregas frequentes e incrementais garante *feedbacks* rápidos, identificação e correção de problemas mais rapidamente;
4. **ambiente seguro** – para garantir um ambiente de trabalho seguro e acolhedor, a equipe é encorajada a expressar suas ideias e opiniões sem constrangimentos;
5. **foco nas pessoas** – as pessoas são o elemento central do processo de desenvolvimento de *software*; a cooperação, a sinergia e o trabalho colaborativo são reconhecidos;

6. **acesso fácil** – as informações e ferramentas necessárias para o desenvolvimento do *software* devem ser acessadas com facilidade pela equipe do projeto;
7. **habilidade técnica** – a capacitação contínua e a troca de experiências e conhecimentos auxiliam na melhoria das habilidades técnicas dos membros da equipe do projeto.

Esses pilares ajudam a guiar a equipe de desenvolvimento de *software* na adoção de práticas ágeis que possam melhorar a eficiência, a qualidade e a colaboração no desenvolvimento de *software*.

4.2 Família Crystal

Cada um dos métodos da família Crystal é identificado por uma cor, de acordo com o número de pessoas envolvidas.

- **Crystal Clear**: indicada normalmente para equipes de até seis pessoas, mas, excepcionalmente, pode chegar até 12 integrantes.
- **Crystal Yellow**: recomendada para equipes com dez a vinte participantes.
- **Crystal Orange**: indicada para equipes de vinte a cinquenta pessoas.
- **Crystal Red**: ideal para equipes de cinquenta a cem membros.

Das metodologias da família Crystal, as mais utilizadas são a Clear e a Crystal Orange, que abordaremos a seguir.

4.3 Crystal Clear

> Crystal Clear é uma metodologia leve, projetada para projetos pequenos e equipes colocalizadas. É altamente adaptável e enfatiza a comunicação cara a cara, com um foco na entrega de *software* que seja útil e satisfatório para os usuários finais. Crystal Clear se concentra em pessoas, não em processos ou ferramentas, e promove a colaboração, a simplicidade e a qualidade técnica como valores fundamentais. (Cockburn, 2004, p. 5)

As atividades executadas nos incrementos devem durar de um a três meses e a Crystal Clear não possui estrutura para comportar mais de um time simultaneamente.

> Crystal Clear ajuda a equipe a se organizar em torno de um conjunto de objetivos compartilhados, fornecendo orientação para decidir quais práticas adotar e quais ferramentas usar para atingir estes objetivos. Isso ajuda a equipe a manter a disciplina sem impor processos pesados e burocráticos. (Cockburn, 2004, p. 30)

4.4 Crystal Orange

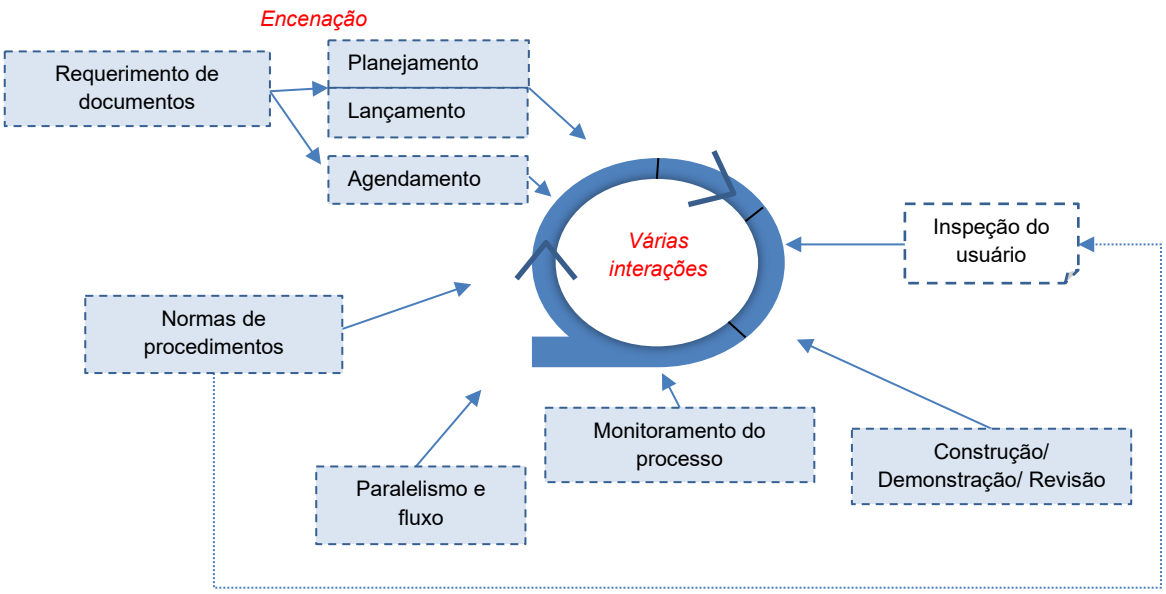
A metodologia Crystal Orange é indicada para projetos de médio porte e agrega mais de um time multifuncional simultaneamente.

> Crystal Orange é uma metodologia ágil adaptativa que enfatiza a comunicação e a colaboração como elementos fundamentais. Ele é projetado para equipes de desenvolvimento de *software* com até 50 membros e é altamente adaptável e flexível. O Crystal Orange incentiva o uso de prototipagem rápida, testes frequentes e *feedback* contínuo como parte do processo de desenvolvimento, e enfatiza a importância da qualidade técnica e do trabalho em equipe. (Cohn, 2005, p. 141)

4.5 Ciclo de vida da família Crystal

O ciclo de vida da família Crystal está fundamentado nas práticas apresentadas na Figura 2.

Figura 2 – Ciclo de vida da metodologia Crystal



- **Encenação (*staging*)**: planejamento do próximo incremento do sistema; são definidos os requisitos que serão implementados na iteração e seu prazo de entrega.
- **Edição e revisão**: construção, demonstração e revisão dos objetivos do incremento.

- **Monitoramento**: o processo de desenvolvimento é monitorado com relação ao avanço e à segurança da equipe e é medido em marcos (*milestones*) e em estágios de estabilidade.
- **Paralelismo e fluxo**: no Crystal Orange, o monitoramento da estabilidade e colaboração entre as equipes possibilita que as diversas equipes trabalhem paralelamente sem que sejam impostos limites.
- **Inspeções de usuários**: geralmente são realizadas de duas a três inspeções por usuários a cada incremento.
- ***Workshops* refletivos:** são reuniões que ocorrem antes e depois de cada interação, com o objetivo de analisar o progresso do projeto.
- **Assuntos locais (*local matters*)**: são os procedimentos a serem aplicados, que variam de acordo com o tipo de projeto; por exemplo, *templates* de código, formatações e padrões para interfaces.
- **Produtos de trabalho (*work products*)**: ordem planejada para a disponibilização das funcionalidades; nas duas metodologias são disponibilizados manual do usuário, sequência de *releases*, casos de testes, migração de código e modelos de objetos; o Crystal Orange apresenta alguns artefatos que não existem no Crystal Clear, pois permite a operação em múltiplas equipes.
- **Padrões (*standards*)**: padrões de notação, convenções de produto, formatação e qualidade usados no projeto.
- **Ferramentas (*tools*)**: ferramentas mínimas utilizadas; para Crystal Clear, são ferramentas de apoio menores, tais como compiladores, gerenciadores de versão e configuração e *printing whiteboards* (quadros brancos que permitem a impressão); o Crystal Orange utiliza ferramentas mais elaboradas, tais como ferramentas para testes, comunicação, programação, monitoramento do projeto, desenho e medição de desempenho.

4.6 Vantagens e desvantagens da metodologia Crystal

4.6.1 Vantagens

- **Adaptação à equipe**: as equipes conseguem adaptar a metodologia as suas habilidades e experiências específicas.

- **Flexibilidade**: a metodologia é altamente adaptável às mudanças no projeto e nos requisitos do cliente.
- **Foco em pessoas:** enfatiza a importância das pessoas e da colaboração para o sucesso do projeto.
- **Simplificação do processo**: os processos e a documentação da metodologia são mais leves, o que pode ser útil para equipes menores ou menos experientes.

4.6.2 Desvantagens

- **Falta de estrutura**: o planejamento e a execução de projetos complexos podem ser prejudicados pela flexibilidade da metodologia.
- **Menos documentação**: o compartilhamento de conhecimento na equipe ou com o cliente pode ficar prejudicado pela documentação menos criteriosa, já que o foco é o desenvolvimento do *software*.
- **Necessidade de *expertise***: os membros da equipe devem possuir conhecimento técnico e experiência em desenvolvimento de *software*, por isso equipes com menos experiência podem ter dificuldades com esse método.
- **Pouco controle de qualidade**: a qualidade do *software* pode ser comprometida devido à pouca rigorosidade da metodologia quanto aos testes e verificações.

TEMA 5 – SCRUM

"Scrum é um *framework* que permite gerenciar projetos complexos de forma adaptativa e produtiva, utilizando uma abordagem ágil de desenvolvimento de *software*." (Sabbagh; Barcaui, 2015).

Crédito: Good_Stock/Shutterstock.

O Scrum foi criado no início dos anos 1990 por Jeff Sutherland, John Scumniotales e Jeff McKenna como uma abordagem para gerenciamento de projetos de desenvolvimento de *software*. A princípio, o Scrum foi implantado em empresas de tecnologia nos Estados Unidos, mas ao longo dos anos popularizou-se em outras áreas e no mundo inteiro.

> Scrum é o nome de uma jogada do *rugby*; um jogo britânico com oito jogadores em cada time, acontece na disputa pela bola em casos de penalidades ou faltas. O termo vem do *rugby*, e se refere à maneira como o time se une para avançar com a bola pelo campo. Tudo se alinha: posicionamento cuidadoso, unidade de propósito e clareza de objetivo. (Sutherland, 2016, p. 16)

Nesse contexto, a primeira utilização da palavra *scrum* foi no artigo intitulado *The New Product Development Game*, escrito em 1986 por Hirotaka Takeuchi e Ikujiro Nonaka. O Scrum que conhecemos na atualidade é uma evolução do conceito original que se concentra em desenvolvimento de *software* ágil.

De acordo com Vale (2014), o "Scrum tem como objetivo maximizar o retorno sobre o investimento (ROI) e a satisfação do cliente por meio do desenvolvimento iterativo e incremental de produtos ou projetos."

O Scrum é uma metodologia ágil e flexível de gerenciamento de projetos que entrega valor gradualmente e de forma consistente e permite que as equipes se adaptem rapidamente às mudanças que podem ocorrer nas necessidades do cliente.

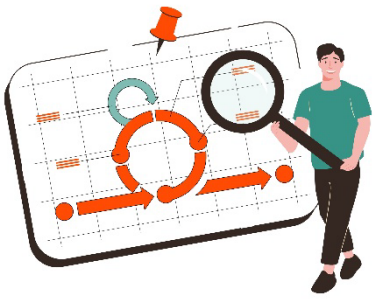
5.1 Objetivos do Scrum

- **Entregar valor ao cliente**: entregar valor de forma incremental e iterativa, garantindo que as necessidades do cliente sejam atendidas e que o produto atenda às expectativas do mercado.
- **Possibilitar adaptação às mudanças**: permite que as equipes se adaptem rapidamente às mudanças nas necessidades do cliente ou do mercado, garantindo que o produto esteja sempre alinhado com os requisitos e as expectativas do cliente.
- **Maximizar a colaboração**: enfatiza a colaboração entre os membros da equipe, garantindo que todas as partes interessadas estejam alinhadas e trabalhando juntas para atingir os objetivos do projeto.

- **Proporcionar melhoria contínua**: promove a melhoria contínua em todas as etapas do processo, permitindo que as equipes aprendam com seus erros e identifiquem oportunidades de melhoria para o próximo ciclo de desenvolvimento.

5.2 Características do Scrum

- **Time auto-organizado**: equipes auto-organizadas com autonomia para tomar decisões sobre como realizar o trabalho e gerenciar suas atividades.
- **Ciclos de trabalho curtos (*sprints*)**: o trabalho é dividido em ciclos curtos e repetitivos chamados *sprints*, com duração de uma a quatro semanas.
- ***Backlog* do produto**: lista priorizada de funcionalidades e requisitos que precisam ser entregues no projeto.
- ***Daily scrum***: reunião diária de 15 minutos em que a equipe compartilha o progresso do trabalho e identifica obstáculos.
- ***Sprint review***: ao término de cada *sprint*, é realizada uma revisão da *sprint* para apresentar o trabalho realizado e obter *feedback* do cliente.
- **Retrospectiva da *sprint***: depois de realizada a revisão da *sprint*, a equipe faz uma retrospectiva para refletir sobre o que funcionou e o que não funcionou durante a *sprint* e identificar oportunidades de melhoria.
- **Papéis definidos**: os papéis dos membros da equipe são claramente definidos, incluindo o Scrum *Master*, o *Product Owner* e a equipe de desenvolvimento.
- **Transparência**: a transparência em todas as etapas do processo é enfatizada, o que significa que todas as atividades e decisões são visíveis e podem ser monitoradas.



Crédito: Vector_Bird/Shutterstock.

5.3 Vantagens do Scrum

- **Maior flexibilidade**: as equipes são mais flexíveis e adaptáveis às mudanças de requisitos do projeto, permitindo entregas mais frequentes e maior valor para o cliente.
- **Foco no valor para o cliente**: as equipes trabalham para entregar valor ao cliente a cada *sprint*.
- **Transparência**: todas as atividades e decisões são visíveis e podem ser monitoradas.
- **Colaboração**: a colaboração e a comunicação constante da equipe permitem melhor compreensão do trabalho e rápida resolução dos problemas.
- **Melhoria contínua**: incentivo à melhoria contínua do processo, permitindo que as equipes aprendam e evoluam ao longo do tempo.

5.4 Desvantagens do Scrum

- **Difícil de entender e implementar**: requer um entendimento profundo e pode ser difícil de implementar, especialmente para equipes sem experiência prévia em metodologias ágeis.
- **Requer muita disciplina e comprometimento**: todos os membros da equipe devem estar comprometidos com o processo e devem seguir as práticas estabelecidas.
- **Exige um alto nível de comunicação**: comunicação constante e aberta entre todos os membros da equipe, o que pode ser desafiador em equipes grandes ou distribuídas.
- **Não é adequado para todos os projetos**: é mais adequado para projetos complexos e em constante mudança e pode não ser a escolha ideal para projetos simples e bem definidos.
- **Dificuldade para medir o progresso**: o progresso pode ser difícil de medir em termos de prazos e entregas finais, pois é um processo iterativo e incremental.

  No capítulo 3, abordaremos esse assunto com mais profundidade.

FINALIZANDO

Nesta abordagem, apresentamos um *overview* dos principais métodos ágeis utilizados no desenvolvimento de *software*. Foram abordadas as metodologias Extreme Programming (XP), Feature-Driven Development (FDD), Test-Driven Development (TDD), Crystal e Scrum.

O XP é um método ágil que se concentra na entrega de *software* de qualidade rapidamente, usando práticas como programação em par, integração contínua e testes automatizados.

Já o FDD se concentra em entregar recursos específicos e de forma iterativa, com ênfase em *design* orientado a objetos, modelagem e inspeção de código.

O TDD é uma técnica de desenvolvimento de *software* que consiste em escrever testes antes de escrever o código do programa. No capítulo 5 retornaremos a esse tema para falarmos da aplicabilidade do TDD.

A metodologia Crystal é uma família de metodologias ágeis que reconhece que cada projeto pode exigir um conjunto de políticas, práticas e processos ligeiramente adaptados para atender a características.

Por fim, o Scrum é um processo iterativo e incremental que se concentra na entrega contínua de valor ao cliente. No capítulo 3, aprofundaremos esses conhecimentos.

Cada método ágil apresenta suas próprias características, vantagens e desvantagens e a escolha do método mais adequado dependerá das necessidades e características do projeto em questão.

REFERÊNCIAS

BECK, K. **Programação extrema**: explorando os limites da programação ágil. Porto Alegre: Bookman, 2000.

CAROLI, P. **Direto ao ponto**: criando produtos de forma enxuta. Casa do Código, 2019.

CHEN, R.; LIAO, Y.; YU, H. XP for Embedded System Development: An Industrial Case Study. In: **2008 International Symposium on Computer Science and Its Applications**, IEEE, 2008, p. 385-391.

CHISOLM, P. **Feature-Driven Development**: An Agile Alternative to Agile Methodologies. Upper Saddle River: Pearson Education, 2007.

COCKBURN, A. **Agile *Software* Development**: The Cooperative Game. 2. ed. Boston: Addison-Wesley, 2002.

_____. **Crystal Clear**: A Human-Powered Methodology for Small Teams. Boston: Addison-Wesley Professional, 2004.

COHN, M. **Agile Estimating and Planning**. Prentice Hall, 2005.

DE LUCA, J. **Feature-Driven Development**: The Next Step in Agile Methods. Boston: Addison-Wesley, 2002.

FREEMAN, S.; FREEMAN, N. **Use a cabeça!** desenvolvimento ágil com TDD (Test-Driven Development). Rio de Janeiro: Alta Books, 2014.

JEFFRIES, R. **Test-driven Development**: by Example. Boston: Addison-Wesley, 2003.

LARMAN, C. **Utilizando UML e padrões**: uma introdução à análise e ao projeto orientados a objetos e ao desenvolvimento iterativo. 3. ed. Porto Alegre: Bookman, 2005.

MAXIM, R. **The Effective Engineer**: How to Leverage Your Efforts in Software Engineering to Make a Disproportionate and Meaningful Impact. San Francisco: O'Reilly Media, 2019.

PEREIRA, T. **Gerenciamento de projetos ágeis com Scrum e Lean**. Brasport, 2016.

PROJECT MANAGEMENT INSTITUTE. **Guia Ágil**. Newtown Square: Project Management Institute, 2018.

SABBAGH, R.; BARCAUI, A. **Scrum**: gestão ágil para projetos de sucesso. Rio de Janeiro: Elsevier, 2015.

SACHETTO, R.; TURBIAS, T. **Desenvolvimento ágil com FDD – Feature-Driven Development**. São Paulo: Novatec, 2014.

SUTHERLAND, J. **Scrum**: a arte de fazer o dobro do trabalho na metade do tempo. 2. ed. São Paulo: Leya, 2016.

VALE, T do. **Scrum**: gestão ágil para desenvolvimento de *software*. São Paulo: Novatec, 2014.