QUALIDADE DE SOFTWARE

AULA 4

Prof.ª Maristela Regina Weinfurter Teixeira

CONVERSA INICIAL

As metodologias ágeis surgiram devido à recorrente crise do software que se estendia desde os anos 1950. Com o Manifesto Ágil (2001), a ideia era de que os modelos rígidos e burocráticos fossem deixados de lado e fossem substituídos por modelos de processos que prezavam pela entrega incremental, rápida, com melhoria contínua e que atendessem ao cronograma, aos custos e que garantissem a qualidade do processo e do software.

Esse manifesto foi proposto como uma declaração de valores e princípios essenciais para a área de desenvolvimento de software. Tal declaração foi descrita num documento, em fevereiro de 2001, com o suporte de profissionais que já praticavam métodos ágeis tais como XP, DSDM, Scrum, entre outros. Eles reuniram-se no norte dos Estados Unidos e conceberam um acordo sobre quais aspectos seriam importantes para que o desenvolvimento de software fluísse com qualidade.

O manifesto estabelece os valores listados a seguir (Ágil, 2022).

1. Os indivíduos e as interações são mais importantes que processos e ferramentas;
2. Software funcionando é mais importante que documentação completa;
3. Colaboração com as partes envolvidas é mais importante do que negociações contratuais;
4. Resposta rápida a mudanças é mais importante que seguir planos meticulosos.

Esse manifesto também recomenda 12 princípios (Agil, 2022).

> 1. Nossa maior prioridade é satisfazer o cliente através da entrega contínua e adiantada de software com valor agregado.
> 2. Mudanças nos requisitos são bem-vindas, mesmo tardiamente no desenvolvimento. Processos ágeis tiram vantagem das mudanças visando vantagem competitiva para o cliente.
> 3. Entregar frequentemente software funcionando, de poucas semanas a poucos meses, com preferência à menor escala de tempo.
> 4. Pessoas de negócio e desenvolvedores devem trabalhar diariamente em conjunto por todo o projeto.
> 5. Construa projetos em torno de indivíduos motivados. Dê a eles o ambiente e o suporte necessário e confie neles para fazer o trabalho.
> 6. O método mais eficiente e eficaz de transmitir informações para e entre uma equipe de desenvolvimento é através de conversa face a face.
> 7. Software funcionando é a medida primária de progresso.
> 8. Os processos ágeis promovem desenvolvimento sustentável. Os patrocinadores, desenvolvedores e usuários devem ser capazes de manter um ritmo constante indefinidamente.

9. Contínua atenção à excelência técnica e bom design aumenta a agilidade.
10. Simplicidade: a arte de maximizar a quantidade de trabalho não realizado é essencial.
11. As melhores arquiteturas, requisitos e designs emergem de equipes auto-organizáveis.
112. Em intervalos regulares, a equipe reflete sobre como se tornar mais eficaz e então refina e ajusta seu comportamento de acordo.

Não há como falarmos sobre qualidade de software sem estabelecermos uma conexão com algumas das mais importantes metodologias ágeis. É muito comum que as empresas utilizem um misto de artefatos e conceitos de várias metodologias, sempre com o objetivo de agilidade no processo de desenvolvimento e garantia da qualidade (processos e software).

TEMA 1 – MÉTODOS E FRAMEWORKS ÁGEIS

Metodologias ágeis vieram para ficar e já nasceram com o propósito não somente de fazer entregas rápidas e interativas, mas também entregas com qualidade. Qualidade e agilidade caminham juntas, pois ambas estão ligadas ao processo de gestão de riscos, requisitos e qualidade. Uma metodologia ágil, além da entrega no prazo, compreende que toda a mudança deve ser capaz de manter produtividade, garantindo a qualidade do processo e do produto.

Para que as entregas sejam ágeis e com qualidade é importante que o time se comunique com facilidade, de forma interativa, mantendo o foco nos problemas sendo resolvidos para que o retrabalho seja menor. Mudanças constantes, cliente colaborativo e comunicação são os pontos principais para que as metodologias ágeis caminhem com o foco na qualidade, pois, sem esse, item, seriam apenas entregas rápidas.

Há quem diga que métodos ágeis são a plena resposta para todos os problemas quando falamos em liberação de software de alta qualidade. De fato, elas permitem a entrega de soluções mais precisas para o mercado, envolvendo clientes ao longo de todo o processo de desenvolvimento de software, permitindo com isso que os times consigam reagir às novas necessidades e mudanças no decorrer do percurso. Iterações permitem que, após a liberação de cada nova funcionalidade, rapidamente recebamos feedback da entrega. Elas ainda permitem que a melhoria contínua seja algo natural no dia a dia dos times de desenvolvimento (Rezvani, 2019).

Um dos principais propósitos do mundo ágil é a entrega de “valor” ao cliente de forma “rápida”. Com isso, a quebra de problemas complexos em pequenas partes é uma técnica poderosa e ajuda muito na obtenção da qualidade. Construir a coisa certa pode ser facilitado quando temos uma forte cultura de qualidade e uma mentalidade ágil, bem como uma estrutura de processo que leva à entrega de valor incremental.

A parte mais importante do ágil é trabalhar de forma iterativa e colaborativa, com foco no valor incremental. Muitas organizações que se autodenominam “ágil” focam mais nas cerimônias e assumem que, seguindo tais processos, estão automaticamente prontas para entregar software com mais rapidez e qualidade. Muitas vezes, as ferramentas e a estrutura necessárias, bem como a qualidade, são completamente ignoradas. Geralmente, isso ocorre porque o investimento necessário para desenvolver pessoas, processos e ferramentas não é considerado como parte da estratégia geral.

1.1 Conceitos básicos em métodos ágeis

As metodologias ágeis, quando utilizadas da forma correta, ou seja, com o foco em entregas ágeis com qualidade, de fato mudarão radicalmente os resultados dos projetos, tanto a nível de produto de software quanto de ciclo de vida do processo de desenvolvimento. Um framework que pode ajudar na implementação correta de um modelo ágil é o Agile Software Soution Framework (ASSF1), que também inclui um toolkit para quantificação de todas as partes do método. Já o Scaled Agile Framework (SAFe2) auxilia na criação de uma estrutura organizacional ágil, a nível de portfólio, programa e times (Rezvani, 2019).

Os principais modelos ágeis do mercado são:

- SCRUM;
- KANBAN;
- Extreme Programming (XP);
- FDD (Feature Driven Development);
- DSDM (Dynamic System Development Model);
- BDD (Behavior Driven Development);
- TDD (Test Driven Development).

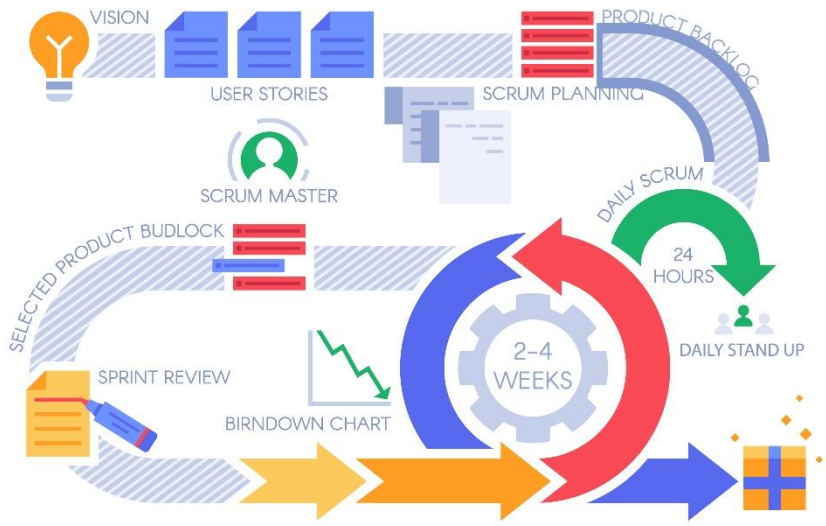
Um dos modelos mais utilizados no mercado é o SCRUM, porém, na maioria das vezes, ele não é aplicado integralmente. O que percebemos no mercado é uma mistura de conceitos existentes nos vários tipos de métodos ágeis. No entanto, utilizando a estrutura do SCRUM, temos algumas terminologias importantes que serão utilizadas daqui em diante. A figura 1 demonstra a ideia central do SCRUM.

Scrum é um uma estrutura (um framework) de regras, papéis, eventos e artefatos utilizados para o desenvolvimento de software ágil e iterativo. O nome desse framework é baseado nas jogadas de Rugby, na qual os participantes reiniciam a jogada juntos e empurrando uns ao outros, com o objetivo de tomar posse da bola.

Em termos gerais, ele é composto por (Rezvani, 2019):

- **Iteração/Sprint:** um período de trabalho predefinido dentro do qual o produto é produzido. A duração geralmente varia entre duas e quatro semanas;
- **Equipe Scrum:** uma equipe Scrum consiste em um time multifuncional de pessoas que executam o trabalho. Eles auto-organizam-se, estimam o trabalho, comprometem-se com ele e cumprem seu compromisso. As funções associadas a cada time são:
    - desenvolvimento;
    - engenharia de qualidade;
    - documentação técnica;
    - proprietário do produto (PO);
    - Scrum Master (SM).

Figura 1 – Estruturas e componentes do SCRUM



Crédito: -artila/Shutterstock.

- **Product Owner (PO):** PO é a ligação entre o time Scrum e os clientes externos, bem como entre as funções internas, como gerenciamento, finanças, vendas e suporte;
- **Scrum Master (SM):** SM é um líder da equipe SCRUM. O SM coordena e facilita todos os eventos do time e trabalha na defesa de bloqueios que possam atrasar o time. SM é o guardião do processo SCRUM;
- **DoR e DoD:** Definição de Pronto (DoR) é fundamental para garantir o sucesso das equipes ágeis. Como parte do refinamento regular, as equipes continuarão refinando as histórias e, quando estiverem claros os objetivos e os critérios de aceitação, poderão marcá-las como "prontas". Isso acelerará a fase de implementação assim que as equipes iniciarem sua iteração. Algumas equipes confundem Definição de Pronto (DoD) e critérios de aceitação. O DoD aplica-se a todas as histórias, mas cada história tem seus próprios critérios de aceitação exclusivos. Muitas equipes perdem a preparação do backlog, o DoR e um DoD claro e perguntam-se por que a implementação do Agile não é tão eficaz;
- **Refinamento do Backlog (Grooming):** o fluxo de trabalho no Agile pode ser um pouco confuso no início. A priorização e o refinamento do backlog são liderados pelo PO, enquanto a entrada é fornecida pelos membros da equipe SCRUM continuamente. A preparação regular do backlog

permitirá que a equipe colabore, faça perguntas esclarecedoras e estime o esforço necessário para as histórias. Essas sessões de preparação de pendências são realizadas pelo menos uma vez por sprint ou, preferencialmente, uma vez por semana. Esse esforço permite que o backlog da equipe tenha histórias prontas para implementação com risco e surpresa mínimos;

- **Planejamento de Sprint (Iteração):** uma vez que as histórias são preparadas, pode-se esperar que o planejamento da Sprint corra bem. A sessão de planejamento da Sprint leva de duas a quatro horas e é facilitada por um Scrum Master. O principal objetivo dessa reunião de planejamento é esclarecer objetivos e prioridades e garantir que todos os objetivos sejam viáveis. Também é importante ter o feedback capturado das iterações anteriores para discutir os pontos-chave que precisam ser abordados nas próximas Sprints. A equipe, então, escolherá um pequeno número de itens da lista para melhorar e incluir no backlog da Sprint conforme achar adequado. Se a lista priorizada de melhorias não for acordada pela equipe SCRUM e escrita, não é garantido que seja feito;
- **Levantamento Diário:** durante a Sprint, os membros da equipe trabalham duro para cumprir seu compromisso. Reuniões diárias (não mais que 15 a 20 minutos) são quando os membros da equipe se comunicam sobre o que trabalharam, o que resta a ser feito e trazem à tona quaisquer problemas de bloqueio que precisem de atenção. O Scrum Master e os membros da equipe ficam de olho no progresso e no trabalho restante para garantir a conclusão bem-sucedida da Sprint;
- **Demonstração de fim de sprint:** no último dia da Sprint, a equipe realiza uma demonstração para mostrar o excelente trabalho que realizaram. Isso fornece visibilidade sobre o trabalho que foi feito e permite que grandes feedbacks sejam considerados para os próximos sprints;
- **Retrospectiva da Sprint:** após a demonstração, uma sessão retrospectiva é conduzida para focar nas melhorias desejadas. Essa reunião é para os membros do SCRUM, incluindo o PO e o SM. As equipes consideram o progresso e os obstáculos que tiveram durante a Sprint e apresentam recomendações a serem seguidas para que os sprints futuros funcionem de forma mais eficaz em equipe. O SM é um dos principais colaboradores nas reuniões de retrospectiva e

acompanhará quaisquer impedimentos e áreas que precisem de ajuda da organização mais ampla. Aqui estão três áreas principais que são discutidas em uma sessão retrospectiva:

  - O que foi bem?
  - O que não correu bem?
  - Que mudanças a equipe quer fazer para a próxima Sprint?

- **Priorizando o Backlog para um melhor fluxo de trabalho:** a maioria das equipes ágeis desenvolve com cadência e libera sob demanda. Isso mostra que, apesar de algumas crenças, as equipes ágeis se preocupam com datas e cumprem seus compromissos. Os proprietários de empresas precisam entender como a priorização funciona na metodologia ágil e como é importante corrigir a qualidade, mas não o escopo.

Um dos requisitos principais para uma empresa tornar-se ágil encontra-se na mudança de mentalidade das lideranças e da estrutura organizacional. A estrutura organizacional precisa tornar-se colaborativa, e as lideranças devem adotar a ideia do método ou dos métodos adotados. Uma das vantagens de considerarmos frameworks ágeis é que processos se tornam mais dinâmicos, escalonáveis e, especialmente, produtivos.

1.2 Desafios de liderança em uma transformação de qualidade ágil

Para que uma transformação funcione, a liderança deve se apropriar dos resultados. Isso inclui um compromisso pessoal com maior qualidade e melhoria contínua. A cultura certa não surgirá se os líderes não estiverem a bordo do mesmo barco. Os executivos da empresa precisam estar de acordo com a nova realidade ágil, o que se dá por meio de um contrato social entre executivos e membros dos times de desenvolvimento, garantindo que a cultura organizacional deve ser mudada. Com isso, os times de desenvolvimento garantem o compromisso de apoio na nova jornada ágil.

Muitos executivos querem implementar o modelo ágil simplesmente porque ouviram que isso permitirá entregas mais rápidas, mas não compreendem que seu papel é fundamental nessa iniciativa, uma vez que precisamos tornar a empresa culturalmente alinha ao ágil. Não somente a rápida entrega, mas sobretudo o valor ao cliente é uma meta importante.

Para implementar qualquer mudança, a equipe de liderança precisa concentrar-se no “porquê”, especialmente se a mudança envolver toda a organização em todos os níveis. O problema em muitos casos é que vamos direto ao como e ao quê e tentamos implementar novos processos sem saber por quê!

Requisitos das lideranças para que o modelo ágil seja bem-sucedido:

- Comunicar-se claramente com todos os níveis de sua organização para garantir que a lógica da mudança seja compreendida;
- Presença garantida a todas as reuniões do modelo adotado (Daily, Retro, Planning etc.);
- Compromisso pelo exemplo;
- Reforço constante sobre a mudança;
- Abertura para feedback construtivo;
- Acreditar no modelo adotado.

Um erro grave na implantação dos modelos ágeis é quando as equipes passam horas priorizando itens que precisavam detalhar e analisar mais para, então, conseguirem uma cadência adequada de entregas. Esquecer-se do tempo de manutenções e melhorias também é algo preocupante, uma vez que, por vezes, não conseguimos fazer a entrega da forma como deveríamos e precisamos considerar nossas listas de dívidas técnicas dentro do backlog do produto e de próximas Sprints.

Para ganhar agilidade nos negócios, o processo de tomada de decisão precisa ser agilizado, caso contrário, não é possível minimizar o tempo de solução. Cada membro da equipe é responsável pelas decisões e deve demonstrar a capacidade de liderar. Os líderes precisam criar um ambiente que capacite indivíduos e equipes para liderar. Um lembrete de que todo o foco deve estar em encontrar as melhores soluções para os clientes permitirá que os mais próximos da obra façam as melhores escolhas. Ter a capacidade de mudar e girar como uma organização com pouca resistência será a chave para se manter competitivo. Essa é, muitas vezes, a mudança de mentalidade mais difícil para líderes e organizações como um todo.

Uma das coisas mais importantes na utilização de metodologias ágeis para desenvolver softwares é saber gerenciar o backlog e reduzir o trabalho em andamento (WIP).

Muitas vezes, um líder chega com uma nova solicitação que foi prometida a um cliente depois que as equipes já discutiram sua lista de pendências e estão a caminho de executar a iteração/liberação planejada. Quando eles perguntam se esse novo trabalho tem prioridade maior do que o que a equipe planejou fazer, a resposta não é clara. Há uma expectativa tácita de fazer esse novo pedido em cima de tudo o que está planejado. É aqui que é necessário um empresário decisivo; eles devem reorganizar as prioridades e remover a ambiguidade sobre o que entregar em seguida. É fundamental que os líderes entendam como um backlog é criado e priorizado.

Lembre-se, não é suficiente apenas “fazer ágil”. Criar uma mentalidade de qualidade ágil, pensar ágil e ser ágil precisam ser o foco. Estabelecer o melhor e mais relevante alinhamento organizacional, entender as melhores abordagens para medir o sucesso e promover uma cultura de transparência e confiança são elementos-chave de uma jornada de transformação bem-sucedida. Um roteiro, a visão, uma meta de iteração e os objetivos de qualidade trarão clareza para todo o time de desenvolvimento ágil, além de trazer visibilidade para os stakeholders.

A cultura ágil já traz a qualidade de software integrada à sua concepção. Sem os objetivos da qualidade clarificados, nenhum time consegue caminhar com seus projetos. Transparência, confiança e compreensão clara dos objetivos de qualidade são elementos essenciais para que o projeto flua e seja entregue de acordo com o planejado dentro da ideia agilista.

TEMA 2 – TDD – TEST DRIVE DEVELOPMENT

O Test Driven Development (TDD) é uma concepção ágil para o desenvolvimento de software orientado a testes. Nele, os desenvolvedores escrevem seus casos de testes e, depois, programam as funcionalidades. O TDD encoraja que o projeto do código deva ser simples e inspire confiança. O livro mais completo e original que aborda o TDD foi escrito por Kent Beck. O TDD é um conceito atrelado à metodologia ágil Extreme Programming (XP) e baseia-se em pequenos ciclos repetitivos, por meio da criação da funcionalidade do software com base em um teste unitário.

Todo processo inicial falha, e, a cada nova linha de código implementada, rodamos novamente o teste, até que ele fique totalmente sem erros. TDD caminha lado a lado com as boas práticas de desenvolvimento de software, para garantir código limpo, menos acoplado e mais coeso.

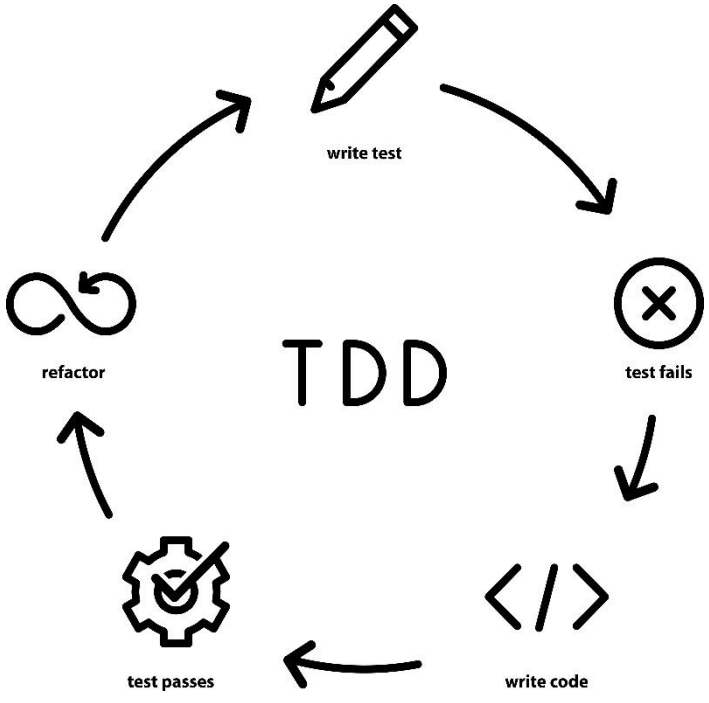
Com TDD, conseguimos:

- código limpo (sem código desnecessário e/ou duplicado);
- código fonte dos testes como documentação dos casos de testes;
- código confiável, logo, com mais qualidade;
- suporte para teste de regressão;
- ganho de tempo na depuração e correção de erros;
- desenvolvimento refatorado constantemente e baixo acoplamento do código.

O Ciclo de Desenvolvimento é composto por sinalizadores “**red”, “green” e “refactor”**.

1. Escrita do teste inicial. **Flag red**;
2. Adição de nova funcionalidade;
3. Execução do teste passar. **Flag green**;
4. Refatoração do código;
5. Escrita do próximo teste.

Figura 2 – Ciclo de desenvolvimento do TDD



Crédito: Vector street/Shutterstock.

Com a implantação do TDD, o feedback ágil e a entrega de novas funcionalidades e sem quebras é garantida.

Para o TDD, as quebras são os problemas que ocorrem quando um teste não passa. Os testes unitários orientam o desenvolvimento simples e refatorado de forma atômica e iterativa. Com isso, a produtividade do desenvolvedor, inicialmente parece burocrática, pois exige que ele mude sua forma de pensamento e crie primeiramente os testes.

Conforme o dev adquire experiência, o processo de desenvolvimento também é mais rápido e produtivo. Tais testes ajudam-nos a não tomar caminhos errados.

Em um primeiro momento, o desenvolvedor tem a sensação de que algo está sem lógica, pois criar o código de testes sem ao menos termos uma linha de código implementada parece algo totalmente estranho. Mas é justamente isso que Beck (2010) propõe, que o primeiro teste sempre vai falhar, porque não encontrará código algum. E, logo depois da primeira escrita de um pequeno teste unitário e que pode não estar totalmente completo também, a escrita do código da funcionalidade é criada com mais clareza e segurança pelo desenvolvedor. A isso chamamos **Keep it Simple, Stupid** (KISS), que conduz à escrita de um código limpo, simples e funcional. Inevitavelmente, ao utilizarmos boas práticas e padrões de programação, garantimos que nossa refatoração caminhe da forma mais correta possível.

Uma vez superada a técnica do TDD, passamos agora de forma mais confortável para refatoração. Depois que fizermos o teste passar, é momento de refatorar, retirando duplicidades, melhorando as nomenclaturas, criando métodos e classes e utilizando os padrões necessários. Junto à refatoração, é hora também de observarmos que cada classe deve ter uma única responsabilidade, a isso chamamos Single Responsibility Principle (SRP).

2.1 TDD e Extreme Programming

Quando falamos em XP, estamos dizendo que precisamos pensar “fora da caixa” ao falarmos sobre hábitos e padrões clássicos de desenvolvimento de software. Precisamos focar na produtividade saudável. Para compreender e usar o XP, é necessário o uso de técnicas e bons relacionamentos com nossos coworkers.

XP é um estilo de programação que foca na excelência do software por meio de técnicas de programação aliadas a uma comunicação clara e sem ruídos entre os desenvolvedores.

Com a aplicação do XP em nosso dia a dia, podemos observar que:

- conquistamos uma filosofia de desenvolvimento de software baseada nos valores de comunicação, feedback, simplicidade, coragem e respeito;
- utilizamos um conjunto de práticas comprovadamente úteis para melhorar o desenvolvimento de software. As práticas complementam-se, amplificando seus efeitos. Eles são escolhidos como expressões dos valores;
- criamos um conjunto de princípios complementares, técnicas intelectuais para traduzir os valores em prática, úteis quando não há uma prática útil para o seu problema específico;
- e, finalmente, desenvolvemos uma comunidade que compartilha esses valores e muitas das mesmas práticas.

XP é um caminho de melhoria para a excelência da sinergia entre pessoas que se unem para desenvolver softwares. São detalhes importantes que a distinguem de outras metodologias:

- ciclos de desenvolvimento curtos, resultando em feedback precoce, concreto e contínuo;
- abordagem de planejamento incremental, que, rapidamente, apresenta um plano geral que deve evoluir ao longo da vida do projeto;
- capacidade de agendar com flexibilidade a implementação da funcionalidade, respondendo às necessidades de negócios em constante mudança;
- dependência de testes automatizados escritos por programadores, clientes e testadores para monitorar o progresso do desenvolvimento, permitir que o sistema evolua e detectar defeitos antecipadamente;
- dependência de comunicação oral, testes e código-fonte para comunicar a estrutura e a intenção do sistema;
- dependência de um processo de design evolutivo que dura tanto quanto o sistema;
- confiança na estreita colaboração de indivíduos ativamente engajados com talento comum;

- confiança em práticas que trabalham tanto com os instintos de curto prazo dos membros da equipe quanto com os interesses de longo prazo do projeto.

"O XP é uma metodologia leve para equipes pequenas e médias que desenvolvem software diante de requisitos vagos ou que mudam rapidamente" (Kent, 2004).

No XP, construímos software que faz o necessário para criar valor para o cliente. O corpo de conhecimento técnico necessário para ser uma equipe de destaque é grande e crescente. Ela trata de restrições no desenvolvimento de software. Não aborda gerenciamento de portfólio de projetos, justificativa financeira de projetos, operações, marketing ou vendas. XP tem implicações em todas essas áreas, mas não aborda essas práticas diretamente. Metodologia é frequentemente interpretada como um conjunto de procedimentos e regras a serem seguidos e teremos sucesso no final, mas, infelizmente, metodologias não funcionam como programas. As pessoas não são computadores. Cada equipe faz o XP de maneira diferente com graus variados de sucesso.

Em XP, podemos trabalhar com times de quaisquer tamanhos e de ampla variedade de projetos. Os valores e princípios do XP são aplicáveis em qualquer escala. As práticas precisam ser aumentadas e alteradas quando muitas pessoas estão envolvidas. Ele adapta-se a requisitos vagos ou que mudam rapidamente, o que é extremamente necessário num cenário de empresas que mudam rapidamente para se adaptarem às novas realidades do mercado. Porém, isso não o restringe a esse cenário.

Com o XP reconciliamos a humanidade dos desenvolvedores com sua produtividade como a boa prática de desenvolvimento de software. A chave para o sucesso encontra-se na aceitação de que somos pessoas em um negócio de pessoa para pessoa e que a técnica também importa, pois somos pessoas técnicas em um campo técnico. Olhando para os processos, existem maneiras melhores e piores de trabalhar, e a busca da excelência na técnica é fundamental. A técnica apoia as relações de confiança, logo, se pudermos estimar com precisão nosso trabalho e entregar qualidade na primeira vez, criaremos ciclos de feedback rápidos, tornando-nos, assim, em desenvolvedores XP confiáveis. No XP, todos os participantes aprendem um alto nível de técnica a serviço dos objetivos do time como um todo.

Os times XP jogam para vencer e aceitam a responsabilidade pelas consequências. Quando a autoestima não está vinculada ao projeto, somos livres para fazer o nosso melhor trabalho em qualquer circunstância.

Manter um pouco de distância nos relacionamentos, reter o esforço por falta ou excesso de trabalho, adiar o feedback para outra rodada de difusão de responsabilidade: nenhum desses comportamentos têm lugar em um time XP.

Quando pensamos em qualidade de software, o XP é uma disciplina de desenvolvimento de software que aborda o risco em todos os níveis do processo de desenvolvimento. Além de ser produtivo, produz software de alta qualidade e é muito divertido de executar.

Abordagem dos riscos no processo de desenvolvimento com XP:

- **Agendamentos:** o XP exige ciclos de lançamento curtos, alguns meses no máximo, portanto, o escopo de qualquer atraso é limitado. Dentro de uma versão, usa iterações de uma semana de recursos solicitados pelo cliente para criar um feedback detalhado sobre o progresso. Dentro de uma iteração, planeja com tarefas curtas, para que a equipe possa resolver problemas durante o ciclo. Por fim, o XP exige a implementação dos recursos de prioridade mais alta primeiro, de forma que quaisquer recursos que passarem do lançamento terão um valor menor;
- **Projeto cancelado:** o XP pede que a parte do time voltada para os negócios escolha a menor versão que faça mais sentido para os negócios, a fim de que haja menos erros antes da implantação e o valor do software seja maior;
- **O sistema literalmente "azeda"**: o XP cria e mantém um conjunto abrangente de testes automatizados, que são executados e executados novamente após cada alteração (muitas vezes ao dia) para garantir uma linha de base de qualidade. Além disso, sempre mantém o sistema em condições de implantação, para o qual os problemas não podem se acumular;
- **Taxa de defeitos-testes:** o XP tem a perspectiva de programadores escrevendo testes (função por função) e de clientes escrevendo testes (programa-recurso-por-programa-recurso);
- **Negócios incompreendidos:** o XP exige que pessoas orientadas para negócios sejam membros de primeira classe do time. A especificação do

projeto é continuamente refinada durante o desenvolvimento, para que o aprendizado do cliente e da equipe possam ser refletidos no software;

- **Mudanças nos negócios:** o XP reduz o ciclo de lançamento, portanto, há menos mudanças durante o desenvolvimento de um único lançamento. Durante uma versão, o cliente pode substituir uma nova funcionalidade por uma funcionalidade ainda não concluída. A equipe nem percebe se está trabalhando em funcionalidades recém-descobertas ou recursos definidos anos atrás;
- **Falso rico em recursos:** o XP insiste que apenas as tarefas de prioridade mais alta sejam abordadas;
- **Rotatividade de pessoal**: o XP pede aos programadores que aceitem a responsabilidade de estimar e completar seu próprio trabalho, dando-lhes feedback sobre o tempo real gasto para que suas estimativas possam melhorar e respeitar essas estimativas. As regras para quem pode fazer e alterar estimativas são claras. Assim, há menos chance de um programador ficar frustrado ao ser solicitado algo impossível. XP também incentiva o contato humano entre a equipe, reduzindo a solidão que, muitas vezes, está no centro da insatisfação no trabalho. Finalmente, o XP incorpora um modelo explícito de rotatividade de pessoal. Os novos membros da equipe são incentivados a aceitar gradualmente mais e mais responsabilidades e são auxiliados ao longo do caminho uns pelos outros e pelos programadores existentes.

A metodologia XP pressupõe que sejamos parte de uma equipe, idealmente com metas claras e um plano de execução. Também define que os times trabalhem em pares. Dentro da metodologia, a mudança deve ser barata e presume que temos a aspiração de crescimento e melhoria de nossas habilidades e relacionamentos, bem como estarmos dispostos a atingir esses objetivos.

O XP aconselha que se você estiver cansado, é melhor você descansar e, quando estiver bem, continuar trabalhando. Quando você fica muito tempo sobre um código ou um teste e nada funciona, é sinal de que é hora de descansar.

Uma coisa bem interessante no TDD é a integração contínua. Com os testes sempre funcionando, garante que o ciclo de programação se torne rápido e eficiente. Quanto aos projetos, quanto mais simples, mais fáceis de

continuidade e de testes, podendo ser automatizado. Já a refatoração é uma forma de simplificar quaisquer comportamentos do código. Gera maior confiança ao time de desenvolvimento na escrita da próxima rodada de testes e de código.

Finalmente, a entrega contínua é garantida de forma mais eficiente com o uso do TDD. Obviamente, como qualquer técnica, possui suas limitações. Não há como testar UIs automaticamente com TDD, nem de código de terceiros, além de haver uma limitação quanto aos testes relacionados a bancos de dados.

Aprendemos que TDD é uma técnica muito interessante e pode auxiliar em muito na construção de código limpo, organizado, com padrões e testado. Mas como toda e qualquer técnica, sempre há a curva de aprendizado e adaptação pelo time de trabalho.

É importante que o time esteja maduro em relação à compreensão do motivo de se utilizarem técnicas que, no início, parecem mais atrapalhar do que ajudar. Mas é justamente dessa curva de aprendizado e amadurecimento que falamos. Para ganharmos produtividade e melhorias em qualidade, vamos ter sempre um primeiro passo que nos parece não sair do lugar. Mas os resultados a médio e longo prazo sempre são bons. Tudo é uma questão de unanimidade e persistência do time.

TEMA 3 – DDD – DOMAIN-DRIVEN DESIGN

Para entendermos como o DDD pode nos ajudar em projetos de software para um domínio não trivial, é necessário que compreendamos as dificuldades de criação e manutenção do software. Para Millett (2015), um dos pontos importantes do DDD encontra-se no domínio do problema e na criação de uma solução sustentável e útil para resolução de problemas. O autor utiliza-se de uma série de padrões estratégicos e táticos.

Nem todo grande produto de software precisa ser perfeitamente projetado. Na verdade, tentar fazer isso seria um desperdício de esforço. Equipes de desenvolvimento e especialistas em domínio usam padrões de análise e conhecimento para detalhar grandes domínios de problemas em subdomínios mais gerenciáveis. Esse detalhamento revela o subdomínio principal – o motivo pelo qual o software está sendo escrito. O domínio central equivale ao domínio do software em desenvolvimento. O DDD enfatiza a necessidade de concentrar esforços e talentos nos subdomínios principais, pois essa é a área que possui mais valor e é a chave para o sucesso do aplicativo. A

compreensão do domínio principal gera delimitações sobre as fronteiras do negócio que será utilizado pelos times de desenvolvimento compreenderem para que serve o software.

O investimento em qualidade de código para as principais áreas de um aplicativo ajudará a mudar com os negócios. Se as áreas-chave do software não estiverem em sinergia com o domínio de negócios, com o tempo, é provável que o design apodreça e se transforme em uma grande bola de lama, resultando em software de difícil manutenção.

No espaço de solução, um modelo de software é construído para cada subdomínio a fim de lidar com problemas de domínio e alinhar o software com os contornos do negócio. Esse modelo não é um modelo da vida real, mas mais uma abstração construída para satisfazer aos requisitos dos casos de uso de negócios, mantendo as regras e a lógica do domínio de negócios. A equipe de desenvolvimento deve concentrar a energia e esforço no modelo e na lógica de domínio, bem como nos aspectos técnicos puros do aplicativo. Para evitar complexidade técnica acidental, o modelo é mantido isolado do código de infraestrutura.

Cada modelo adota padrões de projeto mais apropriados e são usados com base nas necessidades de complexidade de cada subdomínio. Modelos para subdomínios que não são essenciais para o sucesso do produto ou que não são tão complexos não precisam ser baseados em designs ricos orientados a objetos, podendo, em vez disso, utilizar arquiteturas mais processuais ou orientadas a dados.

Os modelos são construídos por meio da colaboração de especialistas de domínio e da equipe de desenvolvimento. A comunicação é alcançada usando uma linguagem compartilhada em constante evolução, conhecida como Linguagem Ubíqua (UL), para conectar de forma eficiente e eficaz um modelo de software a um modelo de análise conceitual. O modelo de software está vinculado ao modelo de análise usando os mesmos termos do UL para sua estrutura e design de classe. Insights, conceitos e termos descobertos em um nível de codificação são replicados no UL e, portanto, no modelo analítico. Da mesma forma, quando o negócio revela conceitos ocultos no nível do modelo de análise, esse insight é realimentado no modelo de código; essa é a chave que permite que os especialistas do domínio e as equipes de desenvolvimento evoluam o modelo em colaboração.

Os modelos ficam dentro de um contexto limitado que define a aplicabilidade do modelo e garante que sua integridade seja mantida. Modelos maiores podem ser divididos em modelos menores e definidos em contextos delimitados separados, em que existe ambiguidade na terminologia ou várias equipes estão trabalhando para reduzir ainda mais a complexidade. Esses modelos são isolados do código de infraestrutura para evitar a complexidade acidental de mesclar conceitos técnicos e de negócios. Os contextos limitados também evitam que a integridade dos modelos seja corrompida, isolando-os do código de terceiros.

O DDD não requer uma estrutura ou banco de dados especial, pois é um modelo implementado no código e segue um princípio Plain Old C# Object (POCO) que garante que ele seja desprovido de qualquer código de infraestrutura para que nada distraia seu propósito centrado no domínio. Uma metodologia orientada a objetos é útil para construir modelos, mas não é obrigatória.

O DDD é arquitetonicamente agnóstico, pois não há um único estilo de arquitetura que você deva seguir para implementá-lo. Os estilos de arquitetura podem variar porque devem ser aplicados no nível de contexto limitado e não no nível do aplicativo. Um único produto pode incluir um contexto limitado que segue uma arquitetura centrada em eventos, outro que utiliza um modelo de domínio rico em camadas e um terceiro que aplica o padrão de registro ativo.

O Domain-Driven Design (DDD) é uma filosofia de desenvolvimento projetada para gerenciar a criação e manutenção de software escrito para domínios de problemas complexos. Agrega uma coleção de padrões, princípios e práticas que podem ser aplicados ao projeto de software para gerenciar a complexidade. Utiliza dois tipos de padrões. Padrões estratégicos moldam a solução, enquanto padrões táticos são usados para implementar um modelo de domínio rico. Padrões estratégicos podem ser úteis para qualquer aplicação, mas padrões táticos só são úteis se seu modelo for suficientemente rico em lógica de domínio. Um modelo abstrato é criado para cada subdomínio a fim de gerenciar problemas de domínio.

Uma linguagem ubíqua é usada para vincular o modelo de análise ao modelo de código para que a equipe de desenvolvimento e os especialistas de domínio colaborem no design de um modelo. Aprender e criar uma linguagem

para comunicar sobre o domínio do problema é o processo de DDD. O código é o artefato.

Para manter a integridade de um modelo, ele é definido dentro de um contexto limitado. O modelo é isolado das preocupações de infraestrutura do aplicativo para separar as complexidades técnicas das complexidades de negócios. Se houver ambiguidade na terminologia para um modelo ou várias equipes no trabalho, o modelo pode ser dividido e definido em contextos limitados menores.

O DDD não dita nenhum estilo de arquitetura específico para desenvolvimento, apenas garante que o modelo seja mantido isolado das complexidades técnicas para que possa se concentrar nas preocupações da lógica do domínio. Também valoriza o foco no domínio principal, a colaboração e a exploração com especialistas do domínio, a experimentação para produzir um modelo mais útil e a compreensão dos vários contextos em jogo em um domínio de problema complexo.

Não é uma linguagem de padrões, é uma filosofia de colaboração focada na entrega, com a comunicação desempenhando um papel central com uma abordagem centrada na linguagem e no domínio para o desenvolvimento de software.

Uma vez que a metodologia de desenvolvimento tenha foco na definição clara e detalhada do domínio do negócio, contribui fortemente para a garantia da qualidade em relação aos requisitos do software.

TEMA 4 – BDD – BEHAVIOR DRIVEN DEVELOPMENT

O Desenvolvimento Orientado ao Comportamento (Behavior-Driven Development, BDD) é um processo de desenvolvimento de software, baseado no TDD, que se concentra em capturar o comportamento de um sistema e, em seguida, direcionar o design de fora para dentro, no qual especialistas e partes interessadas descrevem os comportamentos de um software.

Assim como o DDD, o BDD não se concentra nos aspectos técnicos de um aplicativo. A diferença é que o BDD se concentra no comportamento do software, enquanto o DDD se concentra no modelo de domínio no centro do software que é usado para cumprir os comportamentos – uma distinção pequena, mas importante.

Esse modelo utiliza sua própria forma de UL para especificar requisitos – uma linguagem de análise, se preferir, conhecida como Given, When, Then (GWT). O formato GWT ajuda a estruturar conversas com especialistas de domínio e revelar os comportamentos reais de um domínio.

Um recurso descreve um comportamento que agrega valor ao negócio. Em uma reportagem, um papel e um benefício também estão incluídos. A clareza da função relacionada ao recurso permite que a equipe de desenvolvimento, juntamente aos especialistas do domínio, entenda com quem falar ou quem procurar. O benefício justifica a existência do recurso, ajudando a esclarecer por que o usuário empresarial deseja o recurso.

Para entender melhor um recurso e seu comportamento, usamos cenários BDD para descrição do recurso em diferentes casos de uso. Os cenários começam com uma condição inicial, os Givens. Os cenários contêm um ou mais eventos, os Quandos, e descrevem os resultados esperados, os Então.

Além de ser uma maneira leve de capturar requisitos, os cenários fornecem critérios de aceitação que desenvolvedores e testadores podem usar para determinar quando um recurso está completo e os usuários de negócios podem usar para confirmar que a equipe entende o recurso.

O uso desse método de captura de requisitos remove a ambiguidade que a documentação tradicional de requisitos pode resultar, ao mesmo tempo em que enfatiza fortemente a linguagem do domínio. Os recursos e cenários são um produto da colaboração entre a equipe de desenvolvimento e especialistas de negócios e podem ajudar a moldar o UL.

Da mesma forma que o DDD, o BDD auxilia a condução dos requisitos e comportamentos que nosso software deve ter. Com isso, agregamos da mesma forma que no DDD questões muito importantes para garantia da qualidade do software que está sendo projetado.

TEMA 5 – UX E O DESENVOLVIMENTO ÁGIL

A abordagem ágil para desenvolvimento de software inclui vários recursos, tais como (Voil, 2020):

- equipes multidisciplinares, idealmente colocalizadas;
- uma forte ênfase na comunicação interpessoal;

- a substituição de requisitos formalmente documentados por um backlog continuamente gerenciado e priorizado, com as histórias de usuários e foco na prototipagem;
- desenvolvimento iterativo organizado em períodos fixos curtos, conhecidos como iterações, sprints ou timeboxes;
- entrega incremental frequente de software o mais ágil possível.

Como resultado após o uso dos vários recursos anteriormente citados, conseguimos a redução de entregas não adequadas ao propósito do software. Uma vez que o desenvolvimento ocorre iterativamente e permite feedback contínuo, a garantia da qualidade do software/projeto ocorre de forma natural.

O pensamento ágil está profundamente enraizado em muitas organizações de desenvolvimento de produtos de software. De certa forma, o design ágil e o centrado no usuário complementam-se lindamente: ambos são baseados em iteração e prototipagem contínuas, reconhecendo a importância do aprendizado validado e de ciclos de feedback curtos. No entanto, na prática pode haver tensões e dificuldades que surgem das variações de abordagem entre desenvolvedores, analistas e designers.

O processo de design atualmente está relacionado com o desenvolvimento de software, e não é diferente com as metodologias ágeis. Os profissionais de UX devem manter-se sempre dois passos à frente dos desenvolvedores, trabalhando em sprints que subsidiam sprints posteriores. Se a equipe usa um método como o Método de Desenvolvimento de Sistemas Dinâmicos (DSDM), isso inclui uma fase de fundamentos em que a pesquisa pode ser realizada, mas o que realmente é necessário é uma forma de realizar todas as atividades de pesquisa e design de UX continuamente, ao longo do projeto.

Devemos tratar a pesquisa, o design e o desenvolvimento como projetos ou subprojetos separados e criar oportunidades para que eles se informem à medida que o trabalho avança. Como um dos pontos fortes comprovados do Agile é sua insistência em equipes multidisciplinares, sem dúvida a abordagem mais produtiva é manter toda a equipe unida em um único projeto.

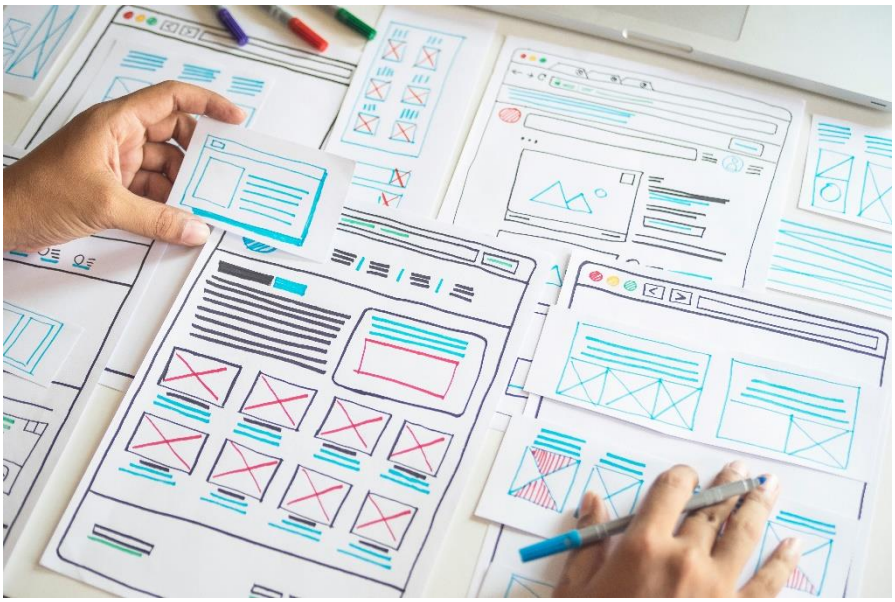
A prototipação surge com o desenvolvimento de software com a ideia de melhorar processos relativos a requisitos. Eles podem tanto ser descartados quanto evolutivos durante o processo de implementação do software.

O protótipo permite a melhora na comunicação com os usuários, pois, de forma rápida, as User Interface (UIs) são desenhadas, inicialmente em baixa fidelidade, chegando à alta fidelidade com até algumas funcionalidades.

Com a prototipação, o processo de desenvolvimento torna-se incremental e rápido, assim como todas as metodologias ágeis prezam. Mas como toda técnica, a prototipagem pode conduzir o desenvolvimento do produto de forma a reter requisitos insuficientes. Então, é bem importante que a prototipagem seja utilizada sempre em conjunto com outras técnicas que não lancem fora todos e quaisquer requisitos importantes para o sucesso do software.

Os protótipos de baixa fidelidade (*wireframes*) retratam a ideia inicial de como a UI ficará. Podem ser feitos com papel e lápis, com esboços iniciais tanto sobre a UI quanto sobre suas funcionalidades. A figura 3 exemplifica um *wireframe*.

Figura 3 – Protótipo de baixa fidelidade



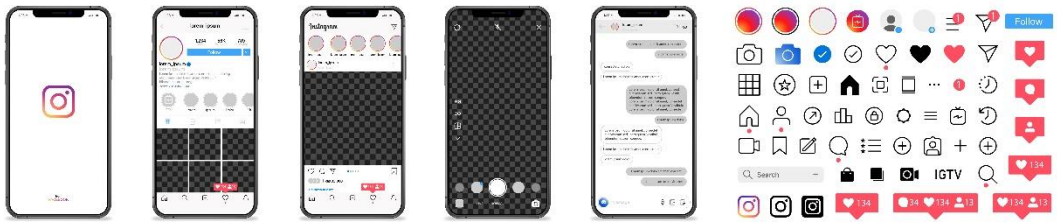
Crédito: Chaosamran_Studio/Adobe Stock.

Esse tipo de protótipo, geralmente, é mais barato, por conta dos materiais utilizados. Embora inicialmente eles serem feitos em papel, já temos algumas ferramentas que auxiliam quem não gosta de trabalhar com papel, ou trabalha remotamente: Balsamiq e OmniGraffle.

Uma vez elaborados os protótipos de baixa fidelidade e validados com os usuários, podemos passar para os de média fidelidade (mockups). Esses já são melhores que os de baixa fidelidade, pois, por meio de ferramentas

automatizadas, conseguem simular comportamentos de interação das UIs. A sensação do usuário já melhora muito em relação ao tipo de protótipo anterior. Exemplo de mockup na figura 4.

Figura 4 – Mockup – protótipo de média fidelidade

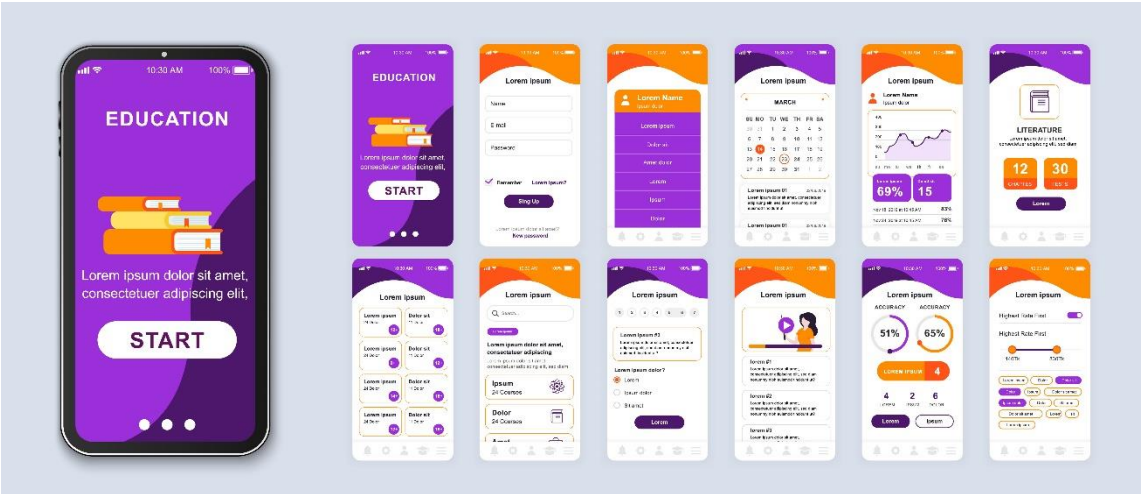


Crédito: 123levit/Adobe Stock.

Para desenvolvermos esses mockups, há várias ferramentas no mercado, tais como Sketch, UXPin, Adobe XD, InVision, entre outras.

Finalmente, o protótipo de alta fidelidade já é uma versão similar ou a própria UI final. Essas UIs são desenvolvidas diretamente em linguagem de programação, linguagem de marcação e com frameworks alinhados com as linguagens escolhidas. Nesse momento, o realismo é bem próximo do que será o software com todas as suas *features* implementadas. A figura 5 exemplifica uma UI em alta fidelidade.

Figura 5 – Protótipo de alta fidelidade – UI



Crédito: alexdndz/Adobe Stock.

Mesmo que nessa etapa do protótipo já estejamos utilizando a linguagem de programação definida para o projeto, há ferramentas que auxiliam no design

dessas UIs finais, tais como Bootstrap, BootPly, Adobe XD e IDEs próprias para o desenvolvimento de software. Lembrando que grande parte da prototipagem é feita por UI/UX Designers ou por desenvolvedores que tenham especialidade em Design de UI/UX.

Prototipagem é uma técnica muito interessante no que se refere à qualidade de software, pois traz consigo o hábito de testes. Todo protótipo deve vir acompanhado da definição dos stakeholders, muitas vezes, utilizando-se personas. São escolhidas tarefas que o usuário usará para realizar os testes com a equipe técnica. Com a prototipagem, utilizamos testes de usabilidade.

5.1 Usabilidade e testes de usabilidade

Usabilidade é um assunto tratado dentro da área de Interação Humano-Computador (IHC). IHC, por sua vez, é abordada dentro da ISO 13407 (Projeto Centrado no Usuário), que explica como a qualidade no uso pode incrementar a qualidade do software. Em linhas gerais, ela estabelece:

1. Especificação e entendimento do contexto de uso;
2. Especificação e entendimento dos requisitos da empresa e dos usuários;
3. Desenvolvimento da solução de interação (IHC);
4. Avaliação da interação em confronto com os requisitos do software.

Ou seja, tais tarefas tornam os requisitos do software mais consistentes e detalhados para a produção de software de qualidade. Dentro da mesma linha de uso, existe a ISO 9241-11, que descreve como a qualidade no uso do software pode ser definida, documentada e avaliada, também levando em consideração a melhoria na qualidade do software.

A identificação do contexto de uso, a especificação da qualidade nos requisitos de uso, o monitoramento da qualidade no uso e a avaliação da qualidade no uso são atributos alicerçados nessa ISO.

Antes de prosseguirmos, é importante compreendermos o que é usabilidade. É um termo que define a facilidade com que as pessoas podem empregar um objeto para realizar uma tarefa. Ela é acompanhada de métodos de mensuração para o estudo dos princípios de eficiência no uso de um objeto. Aqui, podemos trocar a palavra “objeto” por “software”, visto que usabilidade não é exclusividade da área de engenharia de software. Dentro de IHC, usabilidade refere-se à simplicidade e facilidade que um software apresenta para uso de

suas IUs. A ISO 9241, além do que já conversamos, também possui um tópico específico chamado ergonomia de software de escritório, que aborda as características necessárias para a usabilidade de software.

O teste de usabilidade é uma estratégia de teste formalizada para que os usuários sejam envolvidos como amostra-alvo no uso de um determinado software. Os usuários devem proceder à execução de tarefas típicas e críticas. Ele tem diferentes técnicas para avaliação da ergonomia de software interativo, tais como:

1. Avaliação Heurística;
2. Inspeção por checklists;
3. Testes de Percurso;
4. Entrevistas e Questionários;
5. Percurso cognitivo.

Normalmente, essas técnicas envolvem usuários reais, o que pode tornar o processo bastante caro, caso não estejamos desenvolvendo um software para usuários específicos de uma empresa, por exemplo. Quando expandimos para o mundo de software mobile ou web, que vão atingir milhares de usuários, os testes de usabilidade precisam passar por todo um preparo, desde a seleção dos usuários, que, na maioria das vezes, são pagos, até a busca de locais para aplicação dos testes, geralmente empresas especializadas em pesquisas e testes de usabilidade. Também é preciso providenciar todo o preparo das atividades que serão desenvolvidas para que os usuários participem do processo de testes de usabilidade e ergonomia.

Ao falarmos em usabilidade e testes de usabilidade, não há como não nos referirmos ao pai da heurística, Jakob Nielsen. Ele formulou dez heurísticas que são utilizadas no teste de avaliação heurística e por todos os profissionais de UX Design & Researcher.

As dez heurísticas de Nielsen:

1. Visibilidade do status do software;
2. Correspondência entre o software e a vida real;
3. Liberdade e controle do usuário;
4. Consistência e padrões;
5. Prevenção de erros;
6. Reconhecimento e não lembrança;

7. Flexibilidade e Eficiência;
8. Estética e Design minimalista;
9. Auxiliar usuários no reconhecimento, no diagnóstico e na recuperação de erros;
10. Ajuda e Documentação.

A aplicação da avaliação heurística utiliza-se desses dez tópicos e, geralmente, é feita por mais de um profissional, pois cada um possui perspectivas diferentes e engrandecedoras aos testes.

FINALIZANDO

As metodologias ágeis nasceram com a intenção de serem simples, iterativas e interativas, cumprindo cronograma e custos, mas com toda orientação em relação à garantia da qualidade. Processos orientados à colaboração, processos orientados ao domínio do negócio, ao comportamento ou a quaisquer que forem seus focos estão sempre atrelados à qualidade e também aos aspectos de usabilidade e UX.

Cada vez mais multidisciplinar, as equipes de desenvolvimento também têm suas atividades apoiadas por técnicas de verificação e validação para garantia da construção de requisitos corretos e códigos bem elaborados. A UI não se compromete somente com estética, cores e fontes. A experiência do usuário é uma característica relativa à interação humano-computador que deve permear todo processo de desenvolvimento do front-end ao focar o usuário no centro do processo.

Com o aumento de técnicas e ferramentas que apoiam tanto o desenvolvimento quanto os testes, a automação de testes é algo valioso e que vem ganhando maior importância no processo de garantia de qualidade de software.

Ainda não chegamos ao modelo ideal, mas já evoluímos bastante para conseguirmos aplicar desde simples testes unitários até as pesquisas mais complexas na área de experiência do usuário. O que importa é que o processo de desenvolvimento e a qualidade do software evoluam constantemente por meio dos feedbacks e da melhoria contínua.

REFERÊNCIAS

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

BECK, K.; ANDRES, C. **Extreme programming explained**: embrace change. 2. ed. Addison-Wesley Professional, 2004.

MANIFESTO Ágil. Disponível em: <https://agilemanifesto.org/iso/ptbr/principles.html>. Acesso em: 27 ago. 2022.

MILLETT, S. T. N. **Patterns, principles, and practices of Domain-Driven Design**. Birmingham: Wrox Press, 2015.

REZVANI, N. **An executive's guide to software quality in an agile organization**: a continuous improvement journey. Los Altos: CA Press/Apress, 2019.

VOIL, N. **User experience foundations**. Swindon: BCS, 2020.