**QUALIDADE DE SOFTWARE**

AULA 4

Profª Maristela Weinfurter

**CONVERSA INICIAL**

Vamos conversar agora um pouco sobre a metodologia ágil e como ela contribui para agregar conceitos de qualidade e melhorar nosso produto e nossos processos de desenvolvimento.

Passamos algumas décadas falando sobre crise do *software*, engenharia de *software*, normas e padrões em qualidade de *software* entre 1960 e o ano 2000. Porém, testadas e por vezes refutadas, as metodologias de desenvolvimento mais clássicas não conseguiam dar a efetividade que precisávamos, com menores custos e com maior rapidez nas entregas. Foi assim que surgiu, então, o manifesto ágil, nos idos de 2001.

O manifesto ágil foi a proposta de uma declaração de valores e princípios essenciais para a área de desenvolvimento de *software*. Tal declaração foi descrita em um documento, em fevereiro de 2001, com o suporte de profissionais que já praticavam métodos ágeis, tais como XP, DSDM, Scrum, entre outros. Eles se reuniram no norte dos Estados Unidos, mais precisamente em Utah, e chegaram a um acordo sobre quais aspectos seriam importantes para que o desenvolvimento de *software* fluísse.

Esse manifesto aborda quatro valores a saber (Manifesto, 2022):

Valores:

1. 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.

Além disso, esse manifesto aborda 12 princípios (Manifesto, 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.
12. Em intervalos regulares, a equipe reflete sobre como se tornar mais eficaz e então refina e ajusta seu comportamento de acordo.

Mas, afinal de contas, a metodologia ágil resolve todos os problemas em relação à qualidade de *software* e de processo? A resposta é não, porém, ela busca criar abordagens que sejam mais conectadas com questões de qualidade. O crescimento de *startups* também auxiliou na ideia de velocidade nas entregas, mas com a necessidade de um mínimo de qualidade.

Algumas práticas resultam em excelente valor, tais como padronização de código, rituais de atividades, teste A/B, entre outras. Note que a metodologia ágil não é a salvação para os problemas em qualidade de *software*, mas agrega muitos conceitos e práticas que dão leveza às questões de qualidade.

**TEMA 1 – DOCUMENTOS, FEEDBACK DO CLIENTE E DEFINIÇÃO DE PRONTO**

Para falarmos sobre documentos, *feedback* do cliente e definição de pronto, estamos pisando em terreno de métodos ágeis. Para tanto, vamos iniciar abordando um pouco sobre alguns métodos, seus conceitos e como eles podem agregar conceitos importantes para a qualidade de *software*.

**1.1 MÉTODOS E FRAMEWORKS ÁGEIS E O DESTAQUE AO SCRUM**

O ano de 2001 foi um marco na era das metodologias ágeis, porém, não necessariamente foi quando elas surgiram. Elas já vinham sendo implementadas e melhoradas há um bom tempo. E quando falamos em metodologias, isso corresponde a dizer que temos vários *frameworks* considerados ágeis. Alguns exemplos são:

1. SCRUM;
2. KANBAN;
3. Extreme Programming (XP);
4. FDD (*Feature Driven Development*);
5. DSDM (*Dynamic System Development Model*);
6. BDD (*Behavior Driven Development*);
7. TDD (*Test Driven Development*).

Ao analisarmos o mercado, percebemos muito mais a denominação SCRUM em evidência, porém, o que mais ocorre dentro das organizações é justamente a adequação e adaptação de um misto entre os métodos. Mais raramente iremos encontrar algum lugar que utilize exclusivamente um dos modelos citados acima, por exemplo.

Um dos requisitos principais para uma empresa tornar-se Ágil encontra-se na mudança de mentalidade das lideranças e da estrutura organizacional em redesenho. A estrutura organizacional precisa tornar-se colaborativa e as lideranças entrarem na ideia do método ou dos métodos adotados.

Na sequência, uma vez adotada a metodologia ágil, é importante o olhar sobre outras propostas de metodologias que auxiliam em muito o método adotado, tais como *Design Thinking*, *Design Sprint* e o *Lean*.

Uma das vantagens de adotarmos *frameworks* ágeis é que os processos se tornam mais dinâmicos, escalonáveis, além de gerarem mais produtividade ao time todo.

Agora falando um pouco sobre o SCRUM (figura 1), este utiliza-se de etapas ou ciclos de desenvolvimento chamados Sprints, as quais permitem maior qualidade nas entregas e mais agilidade nas mudanças dos requisitos durante todo o ciclo de vida do produto.

Figura 1 – Estrutura e Componentes do SCRUM

Diagrama

Descrição gerada automaticamente

Créditos: Asha Sreenivas/Adobe Stock.

O *framework* Scrum é baseado em pilares e papéis, no qual os clientes tornam-se parte integrante da equipe de desenvolvimento, os quais validam ou até mesmo redefinem em conjunto as entregas. Neste caso, é excelente a forma como os riscos são administrados em conjunto, bem como o cronograma é monitorado, também em conjunto. Ou seja, tudo é muito transparente, conta com inspeções monitoradas e priorizadas em conjunto, através de reuniões diárias (*daily*), além de o projeto estar em constante adaptação por conta das inspeções, atividades refinadas e ajustadas dentro das Sprints. Separamos um momento para falarmos sobre os papéis dentro da metodologia Scrum e das demais, até mesmo para fazermos uma análise em conjunto. O Scrum é recomendado para projetos que dependam de definições de prioridades nos processos e otimização do desenvolvimento de gestão para vários segmentos de negócios.

Kanban não é propriamente um *framework* ágil, é uma ferramenta que pode ser aplicada em conjunto com quaisquer metodologias ágeis. Ela teve início nas linhas de montagem do sistema Toyota de Produção para acompanhar os estágios das atividades:

1. *to do*: a realizar;
2. WIP (*work in progress*): em andamento;
3. *done*: já realizado.

Basicamente, ele obedece a alguns princípios básicos, tais como: gestão à vista, desenvolvimento adaptativo e estágios de trabalho. Tudo isso aplicado em um *board*(quadro) com os três estágios citados anteriormente.

O método XP (*Extreme Programming*) tenta garantir a qualidade de projetos de *software* com um conjunto de atividades, tais como: testar, revisar, desenvolver, integrar, simplificar e realizar tudo isso sempre em iterações curtas, focando na redução de custos. Os princípios básicos do XP são: comunicação, simplicidade, *feedback*, coragem e coach. Vamos comentar sobre um dos elementos muito importante para qualquer ambiente ágil, o *feedback*. O *feedback*, feito através de reuniões diárias (*daylies*), de retrospectivas e de revisões de produto, garante ao cliente se o produto funciona adequadamente e dentro do prazo. Como o processo de *feedback* é constante, há uma garantia maior de que ao término do projeto, de fato, o produto esteja o mais alinhado possível com as necessidades do cliente. A figura 2 demonstra visualmente as características do XP.

O FDD trabalha com a ideia de orientação do desenvolvimento de pequenas funcionalidades. Uma *feature* corresponde a uma pequena funcionalidade. Quando um domínio de negócio é bastante grande, o FDD se encaixa bem, pois o domínio de negócio é decomposto até que processos, atividades estejam bem refinados.

Já o DSDM tem por objetivo entregar *software* com cronograma e custo ajustados aos requisitos ao longo do desenvolvimento. Ele é uma extensão do RAD (*Rapid Application Development*, que surgiu nos anos 1991, substituindo o termo prototipação rápida). Ele é um *framework* interativo e incremental que alia envolvimento do usuário no processo, autonomia do time, entregas pequenas, eficácia, *feedback*, reversibilidade, previsibilidade, testes tratados fora do ciclo de vida e comunicação.

O TDD (*Test Driven Development*) é uma técnica de desenvolvimento de *software* que preza pela verificação e validação do *software* em ciclos curtos de repetições. O processo inicia-se com a escrita de casos de testes automatizados, garantindo que o código que será escrito esteja de acordo com o caso de teste. O código é refatorado até que esteja de acordo com os padrões aceitáveis.

Já o BDD (*Behavior Driven Development*) é outra técnica de desenvolvimento ágil alinhada à colaboração do time de desenvolvimento, com os times de qualidade e outros profissionais que possam estar relacionados ao conceito de verificação e validação. Mesmo controverso, pode ser visto como uma extensão da técnica TDD. A proposta do BDD encontra-se em onde inicia o processo, o que se deve testar e não testar, quanto devemos testar de uma única vez, como as chamadas de testes devem ocorrer e a compreensão do porquê um teste falha.

Dentre as práticas do BDD, temos o envolvimento das partes interessadas, a descrição do comportamento da aplicação ou de uma unidade de código, a automação de testes para obtenção de *feedback* rápido, testes de regressão, a escrita de comportamentos com *Should* (deve) e o uso de simuladores de teste (*mocks*, *fakes*, *spies*) para a colaboração entre módulos não escritos ainda.

Figura 2 – Extreming Programming (XP)

Diagrama

Descrição gerada automaticamente

Créditos: Kostiantyn/Adobe Stock.

Conforme vamos avançando rapidamente sobre os conceitos dos vários *frameworks* e ferramentas, observamos várias características em comum, como é o caso de prazos, custos, pequenas partes, *feedback*, testes. Enfim, a realidade é que qualquer método ou ferramenta ágil tem como objetivo final a agregação de rapidez na entrega com qualidade no processo e no produto final.

**1.2 OS PAPÉIS DO PO E DO PM EM MÉTODOS ÁGEIS**

Seguindo a ideia da seção anterior, na qual falamos sobre algumas metodologias ágeis, agora vamos comparar os papéis encontrados dentro dos vários métodos ágeis. Vamos iniciar pela metodologia Scrum.

No método Scrum temos três papéis em destaque:

1. *Scrum Master* (SM): responsável pelo monitoramento das atividades da equipe e dos processos. Administra prazos e impedimentos;
2. *Product Owner* (PO): responsável pelo projeto e representa as partes interessadas, bem como responde pelo negócio;
3. *Development Team* (DT): equipe multidisciplinar responsável em trabalhar em todo o ciclo de desenvolvimento do produto.

A metodologia Scrum é a que mais vem ganhando espaço dentro das equipes de desenvolvimento, então, por mais que não haja a adoção completa do Scrum, seus papéis são bastante utilizados em equipes ágeis.

Vamos ao caso do PO (*Product Owner*). Este é membro do time, geralmente baseado no Scrum, que define histórias do *backlog* de produtos e depois os prioriza. O PO mantém a integridade conceitual sobre as novas *features*, *bugs* e melhorias, para que o produto realmente esteja de acordo com a visão definida junto aos clientes. O PO ainda é responsável pela qualidade das entregas finais, para as quais ele deve fazer as devidas validações.

O PO deve ter conhecimento sobre o domínio do negócio e conseguir mediar as necessidades dos diferentes clientes e usuários do produto/projeto. Suas responsabilidades diárias encontram-se entre o planejamento e a comunicação eficaz. Se o time adotar, por exemplo, o BDD, o PO deve participar no desenvolvimento dos requisitos e critérios de aceitação das histórias.

Agora, vamos tratar de um assunto muito corriqueiro dentro das organizações e que gera muita confusão. De um lado temos a figura do PO (*Product Owner*), de outro temos a figura do PM (*Product Manager*). Quando falamos em empresas ainda pequenas, o mesmo profissional pode até agregar as duas funções. Mas quando falamos em empresas já em modo escalando, é necessária a divisão de atividades.

Um PM está mais envolvido com os clientes, logo, com o negócio, enquanto o PO, mesmo tendo que conhecer do negócio, está mais alinhado aos times de tecnologia e soluções. O PM trabalha em uma visão de *roadmap* do produto, *backlog* de projeto, orçamentos, avaliações ROI, entre outros. Já o PO trabalha com a visão do *backlog* para o time, bem como sua implementação. O foco do PM é mais organizacional e a nível macro, enquanto o PO está alinhado com o dia a dia da implementação e testes dos produtos por Sprints.

**1.3 DOR, DOD, CRITÉRIOS DE ACEITE E COMO FUNCIONAM**

Agora que já conhecemos um pouco sobre metodologias ágeis e as funções do *Product Owner* (PO), vamos nos aprofundar em dois conceitos muito importantes para que possamos garantir entregas com qualidade. Estamos falando sobre DoR (*Definition of Ready*), DoD (*Definition of Done*) e sobre critérios de aceite.

Um parênteses: até aqui vimos várias definições e todas estão em inglês. E isto é assim mesmo, pois no dia a dia, quando você for trabalhar, as pessoas falarão exatamente dessa forma. Dificilmente você trabalhará em um ambiente que utiliza a metodologia ágil e alguém traduzirá Sprint, PO, entre outros tantos vocábulos técnicos que estudamos. Sem entrarmos em uma discussão se isso é bom ou ruim, é o que ocorre. Então, vamos continuar utilizando tais jargões técnicos no idioma inglês.

O que é e como funciona a definição de pronto (DoR)?

Nada mais é que um acordo entre os times de desenvolvimento e o *Product Owner*, utilizando todas as histórias dos usuários para que o *Backlog* esteja o mais refinado possível. Isso melhora muito o planejamento e a especificação das *cards* do *backlog*. Quanto mais refinada uma história, mais *cards* pequenos e autocontidos. *Card*? Sim, cada desenvolvedor ou outro profissional que esteja trabalhando no mesmo produto recebe uma *card* ou várias por Sprint, e nela contém a descrição da atividade que o dev irá fazer. Pode ser uma atividade para o desenvolvimento de uma *feature*, pode ser uma card para a documentação de uma feature, ou quaisquer outras atividades inerentes ao produto. Para que um item do *backlog* faça parte de uma *Sprint*, o time deve ter certeza de que ele está “*Ready*”, ou seja, bem descrita e compreensível para que o desenvolvimento ocorra da forma correta.

É importante que 10% do tempo de cada *Sprint* seja reservado para o planejamento e a adequação da atividade, pois isso reduz o impacto de itens mal especificados, os quais geralmente surpreendem de forma ruim o time e podem gerar mudanças, o que impacta na entrega e no objetivo da *Sprint*.

A definição de pronto (DoD) é outro acordo, definido pelos membros do time (devs e PO), aplicável às histórias dos usuários, com o objetivo de compreensão do que significa “*Done*”. Isso garante transparência.

Uma definição de pronto ideal explicita que cada item passe por testes de aceitação com critérios de aceitação acordados para cada item de atividade. Cada definição de pronto exige que o incremento do produto tenha qualidade para a entrega.

Para a construção de definição de pronto, deve-se considerar todas as restrições organizacionais que possam afetar o time de Scrum, que podem ser restrições em relação aos processos, ao negócio, culturais ou tecnológicos. Se tais situações não forem bem estabelecidas, os times podem iniciar seus projetos com uma definição de pronto longe da ideal e comprometer as entregas.

Vamos a algumas exemplificações:

*Definition of Ready*:

1. Utilizar *wireframe* de baixa fidelidade para projetos de UI;
2. As histórias de usuários serem escritas de acordo com o BDD;
3. Cada história deve vir acompanhada de um critério de aceite;
4. Cada história deve vir acompanhada de dados para os testes.

*Definition of Done*:

1. Para ir para produção, a *feature* precisa passar por todos os testes automatizados (TDD);
2. No caso de IU, as *features* devem funcionar em qualquer tipo de browser ou no caso de *mobile* (tanto para iOS quanto para Android);
3. Código deve ter sido revisado por outro dev via Pull Request no GitHub.

Critérios de Aceite:

1. O usuário normalmente esquece se fez seu *SignUp* através de uma plataforma social (LinkedIn, Google, Facebook ou Instagram). Caso ele tente fazer o *login* por uma forma que não foi a que fez o *SignUp*, o *login* deverá ser inteligente para “forçá-lo” a fazer novo *SignUp*, sem perder os feitos anteriormente.
2. Durante a utilização de uma API específica, é necessário o tratamento de exceção para identificar se há registros, se houve erro, se a API está fora do ar e criar um *log* para futura inspeção.

Vimos até aqui uma série de conceitos atrelados ao método ágil e à qualidade de *software*. Se utilizarmos tudo o que estudamos até aqui, vamos garantir a qualidade total do produto e dos processos? Não, mas certamente estaremos no caminho mais correto, mais próximo do menor número possível de erros e falhas em nosso produto.

**TEMA 2 – PROTOTIPAÇÃO, USABILIDADE, HEURÍSTICAS DE NIELSEN**

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

O protótipo permite a melhoria na comunicação com os usuários, pois de forma rápida, as UIs (*User Interface*) 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 4 exemplifica um *wireframe*.

Figura 4 – Wireframe: Protótipo de Baixa Fidelidade

Calendário

Descrição gerada automaticamente

Créditos: Chaosamran\_Studio/Adobe Stock.

Este tipo de protótipo geralmente é mais barato, por conta dos materiais utilizados. Apesar de inicialmente eles serem feitos em papel, hoje 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*). Estes já são melhores que os de baixa fidelidade, pois através 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 5.

Figura 5 – Mockup: Protótipo de Média Fidelidade

Interface gráfica do usuário

Descrição gerada automaticamente

Créditos: 123levit/Adobe Stock.

Para desenvolvermos estes *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. Estas UIs são desenvolvidas diretamente em linguagem de programação, linguagem de marcação e com *frameworks* alinhados com as linguagens escolhidas. Neste momento, o realismo é bem próximo do que será o *software* com todas as suas *features* implementadas. A figura 6 exemplifica uma UI em alta fidelidade.

Figura 6 – Protótipo de Alta Fidelidade: User Interface

Interface gráfica do usuário, Aplicativo

Descrição gerada automaticamente

Créditos: alexdndz/Adobe Stock.

Mesmo que nesta etapa do protótipo já estejamos utilizando a linguagem de programação definida para o projeto, há ferramentas que auxiliam no *design* destas 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 em conjunto com a equipe técnica, juntamente com prototipagem, utilizamos testes de usabilidade.

**2.1 USABILIDADE E TESTES DE USABILIDADE**

Usabilidade é um assunto tratado dentro da área de IHC (Interação Humano-Computador). 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 então o que é usabilidade. É um termo que define a facilidade que as pessoas podem empregar para utilizar 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 se refere à simplicidade e facilidade que um *software* possui 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 possui 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, estas 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 irão 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, à busca de locais para aplicação dos testes, geralmente empresas especializadas em pesquisas e testes de usabilidade, bem como 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 tem como não nos referirmos ao pai da heurística, Jakob Nielsen. Ele formulou 10 heurísticas que são utilizadas no teste de avaliação heurística e por todos os profissionais de UX Design & Researcher.

As 10 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. Auxílio a usuários no reconhecimento, no diagnóstico e na recuperação de erros;
10. Ajuda e documentação.

A aplicação da avaliação heurística utiliza-se destes 10 tópicos e geralmente é feita por mais de um profissional, pois cada um possui perspectivas diferentes e engrandecedoras aos testes.

**TEMA 3 – TDD, PROGRAMAÇÃO E REFATORAÇÃO EM PARES**

O TDD (*Test Driven Development*) é uma forma de desenvolvimento de *software*orientado por testes. Os desenvolvedores escrevem seus casos de testes e depois partem para a programação das funcionalidades, as quais devem ser testadas pelos testes implementados anteriormente.

Kent Beck é o autor do livro que fala sobre TDD, sendo que o TDD é um dos pilares da metodologia XP (*Extreme Programming*). O TDD baseia-se em ciclos atômicos e repetitivos, no qual cada funcionalidade do *software* possui um teste criado. Todo processo inicial falha, e a cada nova linha de código implementada, rodamos novamente o teste, até que ele fique totalmente sem erros.

Lembrando que mesmo com a utilização de TDD, é importante a adoção de boas práticas de desenvolvimento de *software*, para garantir código limpo, menos acoplado e mais coeso.

O Ciclo de Desenvolvimento é composto por sinalizadores “**r*ed***”**,**“***green***” e“***refactor***”. Exemplificando o processo (Figura 7):

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 7 – Ciclo de desenvolvimento do TDD

Diagrama

Descrição gerada automaticamente

Uma vantagem no uso do TDD encontra-se na garantia de *feedback* rápido sobre as novas *features* e suas quebras. Quebras são os problemas que ocorrem e um teste não passa. O processo de refatoração fica muito mais fácil de ser implementado, uma vez que os testes vão orientando e ajudando na refatoração do código.

O TDD também auxilia na escrita de códigos mais simples e consequentemente mais limpos e na segurança na correção de erros e falhas, gerando maior produtividade para o desenvolvedor. O fato de os testes serem construídos antes da codificação das funcionalidades, obriga que o código seja planejado antes de ser implementado. Isto é muito bom, pois garante que não tomemos o caminho errado. Muitos falam que não é produtivo porque no início realmente a curva de aprendizagem é grande, no entanto, quando a técnica é aperfeiçoada, se torna intuitiva e segura.

É estranho pensar que vamos construir um código de testes sem ao menos termos uma linha de código implementada para testar. Mas é assim mesmo. O primeiro teste sempre irá falhar, porque o teste não encontrará o código. E isto é a base do KISS (*Keep it Simple, Stupid*). Quer dizer que devemos nos esmerar para escrevermos um código limpo, simples e funcional. Utilizarmos boas práticas e padrões de programação. Isso garante 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 fizemos 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 isto chamamos de SRP (*Single Responsibility Principle*).

Para Kent Beck (2010), o TDD auxilia na arquitetura limpa do código, pois aprendemos a escrevê-lo de forma mais simples (fazendo um teste passar), mais clara (refatorando o código após ele passar pelo teste) e mais seguro (com testes).

**3.1 TDD E EXTREME PROGRAMMING**

O TDD adiciona melhorias ao XP através da programação em pares, trabalho descansado, integração contínua, projetos mais simples, melhorias na refatoração e entrega contínua.

A programação em pares aliada ao TDD traz a ideia de que dois programadores vão concebendo o código. Isso é bom porque o código melhora muito e quando um programador está cansado, o outro assume a atividade. O ritmo de programação é contínuo, mesmo que cada um tenha seu tempo de descanso.

O XP aconselha que se você estiver cansado, é melhor você descansar e quando estiver bem, continue 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, gerando 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 e também há ainda 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 utilizar técnicas, que no início parecem mais atrapalhar do que ajudar, contudo é justamente a 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 prazos sempre são bons. Tudo é uma questão de unanimidade e persistência do time.

**TEMA 4 – INTEGRAÇÃO CONTÍNUA**

A integração contínua é um conceito abrangente, mas simples na sua concepção. Este exige comprometimento dos times de desenvolvimento e infraestrutura e uma vez alinhada tal situação, a implantação se torna algo que traz ganhos de produtividade para ambos os times. É uma das práticas da XP.

Cada desenvolvedor envia suas alterações de código a um repositório compartilhado e o procedimento deve ocorrer no mínimo uma vez por dia, mesmo que o código não esteja completo (@wip => *working in progress*).

Isto ocorre normalmente sobre um repositório, como é o caso do mais difundido, o GitHub. O Git é um sistema de controle de versionamento, altamente difundido entre os devs mundo afora. Todos os códigos e demais artefatos ficam em um repositório, o qual contém geralmente um repositório central (*main*) e cada *feature* pode ser trabalhada em repositórios que depois são mesclados ao código principal. O Github é uma plataforma que auxilia no gerenciamento do código em ambiente visual e facilita o uso do Git.

Contudo, como falado anteriormente, não existe somente o GitHub, mas o GoCD, Gitlab-CI, CircleCI e muitas outras ferramentas que por vezes trabalham em conjunto ou agregam as outras ferramentas. Lembrando que a ferramenta central é o Git. As demais são apenas de apoio e automatização.

É interessante observar que não é porque um time utiliza várias ferramentas (*git*, *containers*, *kubernetes*, *Jenkins* etc.) ele está com a integração contínua efetiva.  Apesar de que o conceito de integração contínua vem sendo falado e discutido desde os anos 1993, foi justamente o XP que a colocou em prática.

Para Beck (2010), a integração contínua é uma prática na integração e testes do código com frequência de apenas algumas horas. Integração pode ser assíncrona ou síncrona. A assíncrona é uma abordagem na qual cada dev do time pode integrar modificações a qualquer momento e continuar seu trabalho. Uma vez que as modificações foram testadas e se estiver tudo em ordem, uma rotina de *deploy* realizará automaticamente a atualização do *software* sem intervenção humana (figura 8).

Figura 8 – Integração Assíncrona

Diagrama

Descrição gerada automaticamente

Em caso de anormalidades com o *deploy*, o time de desenvolvimento é notificado.

A integração síncrona difere da assíncrona por utilizar a ideia de que o desenvolvedor integra seu código e executa a *build* automaticamente em um ambiente de testes ou *staging*. Uma vez assegurado de que os testes de regressão passaram, então é possível passar para a próxima tarefa e continuar o desenvolvimento. A figura 9 exemplifica o processo de integração síncrona.

Figura 9 – Integração Síncrona

Diagrama

Descrição gerada automaticamente

Não é difícil encontrar times com um cenário caótico com conflitos de *commits* e que necessite realizar vários retrabalhos em merges. A melhor forma de se resolver este problema é através da prática de integração contínua para minimizar tais problemas.

A integração contínua tem por missão a simplificação do processo de desenvolvimento de *software*. Algumas ferramentas podem automatizar e ajudar no processo da integração contínua, tais como um servidor Jenkins ou GoCD. Elas permitem que em pouco tempo o processo esteja implantado, mas sempre lembrando de que o time precisa estar comprometido com esta ideia.

**TEMA 5 – CRITÉRIOS DE ACEITAÇÃO E USER STORIES**

Critérios de aceitação possuem a mesma ideia independentemente de qual metodologia de trabalho se utilize. Mas vamos continuar na linha das metodologias ágeis. Uma *User Story* (US) é uma ferramenta para auxiliar no detalhamento dos requisitos do *software*. O foco das histórias é a descrição das necessidades e dos objetivos do *software* segundo a visão do usuário. É uma maneira de expressar os requisitos do *software* dentro da metodologia Scrum. Existe um padrão para escrita das US chamado INVEST.

INVEST define que toda *user story* precisa:

* I (Independente): cada história não pode depender de outras histórias;
* N (Negociável): cada requisito precisa de um espaço para discussão;
* V (Valioso): deve ter valor para o cliente;
* E (Estimável): a história deve ser mensurável;
* S (Sensata): a história deve ser coerente;
* T (Testável): deve haver uma forma clara de testes para verificar se a história está ou não funcionando.

As *User Stories* efetivam a comunicação entre as partes envolvidas e o time de desenvolvimento do software. Ela também faz com o que o desenvolvimento seja focado no cliente. Seu formato básico para escrita segue a seguinte ideia:

COMO UM <persona/ator>, EU QUERO UM <recurso/funcionalidade> PARA QUE EU <tenha um objetivo>.

Ao criarmos uma história de usuário, devemos observar para quem estamos escrevendo (personas ou atores), o que vamos escrever e não o como o faremos. Os atores ou personas são baseados nas características dos nossos usuários. Por exemplo: gerente de *marketing*, CEO, cliente de varejo *online* etc.

Geralmente quem escreve as histórias é o *Product Owner* para que ele possa centralizar todas as histórias e assim não sairmos do alvo que queremos atingir com nosso *software*. Para garantirmos uma boa escrita, podemos fazer uso dos 5 Por ques (5 WHYS).

Os *user stories* possuem alguns elementos importantes: cartão, conversa e confirmação. Ainda temos a ideia do Épico.

O cartão, ou *card*, é justamente o espaço dentro do nosso *backlog* (auxiliado por algum *software* como Jira, Trello ou outros) para que esteja em local de acesso a quem está escrevendo ou para quem foi direcionado para implementação ou teste. Ela descreve de forma simples, com linguagem natural, o que pretendemos com determinada funcionalidade.

Algumas recomendações para escrita de *user stories*:

1. Como: define a persona que utilizará a *feature* que está em planejamento;
2. Quero: exemplifica a funcionalidade no *software*;
3. Para: responde à questão “para que a funcionalidade foi criada?”.

Um épico é uma *User Story* abrangente, que geralmente precisa de mais de uma única Sprint para ser desenvolvido. Ele geralmente é quebrado em várias *User Stories* para que caiba dentro de uma Sprint.

A Conversa é outro elemento de uma US. Durante a conversa, todas as pessoas envolvidas no projeto devem ler em conjunto as *User Stories* (US) para que fique claro para todos. É muito comum escrevermos algo e depois outras pessoas interpretarem de formas diferentes. Ou seja, a colaboração é um dos grandes benefícios com *User Stories*. Após a análise em conjunto e levantados os pontos de ajustes, passamos para a confirmação.

A confirmação traz consigo o momento de finalização acerca da solução criada a partir de uma *User Story*, dividida por vezes em várias tarefas para ganharmos em produtividade junto à equipe de desenvolvimento. Dentro da confirmação, encontramos os critérios de aceite, os quais auxiliarão na diminuição do retrabalho, na agilidade de testes e na validação das *features*.

Estes critérios de aceite são geralmente escritos com apoio dos profissionais ligados aos testes, pois eles conseguirão antecipar muitas situações em relação à prevenção de *bugs*, desvios no desenvolvimento, entre outros problemas aos quais o *software* está exposto.

Eles são descritos de forma simples, mas que abrangem todo o comportamento esperado ao definirmos as funcionalidades. Ainda, são responsáveis para que possamos descrever a definição de pronto. Eles têm objetivo de delimitar as fronteiras de cada *User Story*, auxiliam o PO no detalhamento em alto nível das entregas, colaboram com o entendimento e objetivo da US e melhoram o planejamento dos testes.

Bons critérios de aceitação são escritos em um bom nível de detalhamento, são independentes da implementação e principalmente definem o que não devemos fazer na implementação.

**FINALIZANDO**

A qualidade de *software* vai além de normas e padrões. Ela conta muito com técnicas e *frameworks* de metodologias ágeis e também a área de IHC.

Métodos ágeis já nasceram com a inclusão de critérios de qualidade de *software* através de testes, de codificação limpa, com a ideia de automatização de processos de qualidade e de implantação de *software*.

O desenvolvimento de *software* não está relacionado apenas à codificação, ele deve considerar desde o momento dos requisitos, acordos com os usuários, o bem-estar dos desenvolvedores, formas de colaboração, formas de especificação e detalhamento das funcionalidades, entre tantas outras atividades.

Observamos que para o que chamamos de UI vai além de um desenho de *interface*, observando-se estética e cores. Precisamos melhorar a experiência do usuário e para isso utilizamos todas as características da Interação Humano-Computador. Pensarmos nos usuários faz-nos refletir sobre como construir *software* com foco em nossos usuários.

Devido ao grande número de técnicas e ferramentas que apoiam o desenvolvimento de *software* e testes, a automatização de processos tanto no desenvolvimento, quanto nos testes e na implantação, é valiosa para garantirmos produtividade e assertividade.

A evolução das metodologias ágeis com suas características de *software* bem-testado e definido tem conseguido um bom êxito na entrega de *software*.

**QUALIDADE DE SOFTWARE**

AULA 5

Profª Maristela Weinfurter

**CONVERSA INICIAL**

**TESTES AUTOMATIZADOS DE SOFTWARE**

O teste é parte integrante do desenvolvimento de software, juntamente com a codificação, as operações, os requisitos do cliente, entre outros.  E para a definição dos testes, é necessário que pensemos de forma clara, o que queremos testar. Para tanto, há muitos fatores para definição de testes ágeis e algumas práticas que auxiliam times de desenvolvimento e qualidade, tais como:

* Qualidade no desenvolvimento: os times devem se concentrar na prevenção de mal-entendidos sobre o comportamento dos recursos, bem como na prevenção de defeitos no código.
* Orientação do desenvolvimento com exemplos concretos: usar práticas como desenvolvimento orientado a testes de aceitação (ATDD), desenvolvimento orientado a comportamento (BDD) ou especificação por exemplo (SBE).
* Inclusão de atividades de teste, como boa comunicação e reuniões para construir um entendimento compartilhado, elaboração de perguntas para testar ideias e suposições; automatização de testes; realização de testes exploratórios; testes de atributos de qualidade como desempenho, confiabilidade e segurança; e aprendendo com o uso da produção.
* Utilização das retrospectivas de toda a equipe e pequenos experimentos para melhorar continuamente os testes e a qualidade e descobrir o que funciona em seu contexto.

Em Testes Ágeis, há 10 princípios que não são necessariamente apenas para o time de qualidade, mas para qualquer profissional que esteja envolvido com desenvolvimento de software. São eles:

1. Fornecer feedback contínuo;
2. Entregar valor ao cliente;
3. Habilidades em comunicação com os clientes;
4. Coragem;
5. Simplicidade;
6. Praticar a melhoria contínua;
7. Responder à mudança;
8. Organização;
9. Foco nas pessoas;
10. Aproveitar o caminho.

O Manifesto do Teste Ágil declara valores e princípios que são essenciais para o desenvolvimento de software e para o ciclo de vida ágil.

Figura 1 – Manifesto do Teste Ágil



Seguindo o esquema da Figura 1, temos da esquerda para direita que:

1. Testing throughout OVER testing at the end: O retrabalho gera aumento de trabalho exponencial para corrigir “bugs” depois que o ciclo de vida do software foi concluído. Teste é algo que se deve fazer do início ao final do ciclo e frequentemente.
2. Preventing bugs OVER finding bugs: O software deve ser construído com qualidade do início ao fim, sendo pensado tanto em produção quanto em manutenção.
3. Testing understanding OVER checking functionality: A propriedade compartilhada gera maior qualidade. Logo, avaliar e verificar o entendimento do time no que estamos construindo é essencial. Pair Programming, code reviews podem ajudar na coleta de feedbacks.
4. Team responsibility for quality OVER tester responsibility: toda a equipe de desenvolvimento é responsável pela qualidade do produto que está sendo desenvolvido.

Dentro da cultura de testes, podemos observar algumas relações entre o manifesto do teste ágil e o ciclo de vida ágil. Os times trabalham com ciclos incrementais e que se repetem, prevenindo o teste do software do início ao término. Desde a criação dos planos de testes e critérios de aceitação, as práticas de desenvolvimento orientadas a testes garantem o alinhamento do que está sendo construído desde o início. E a responsabilidade é sempre de todos pela qualidade, não somente do time de QA.

Para conseguirmos atingir os objetivos e ideais da área de testes ágeis, devemos prestar atenção na automação de testes. Mas para que precisamos automatizá-los? Porque garantem que não há regressões no software, o feedback é mais rápido, economia de tempo executando testes repetidos, trabalhos manuais podem gerar erros pela fadiga do profissional, entre outros.

Os principais testes que podemos automatizar são: testes de regressão, de tarefas repetitivas, de funcionalidades críticas e testes de cálculos matemáticos. Embora automatizamos os testes, ainda assim será necessário executar os testes manuais. Testes ágeis e automação têm uma grande relação, pois garantimos um feedback contínuo e rápido, bem como entrega de software com qualidade.

**TEMA 1 – MÉTRICAS DE TESTES E COBERTURA**

A atividade de testar algo envolve sempre algum tipo de métrica. As métricas servem para nos direcionar naquilo que precisamos de alguma forma melhorar dentro do processo ou do produto de software.

Quanto às métricas de testes, que são um padrão de medidas para a verificação da efetividade e eficiência do processo de desenvolvimento de software, são utilizadas para promoção das estimativas do esforço alocado para os testes. Estas são capturadas e utilizadas para o acompanhamento da melhoria do processo.

As métricas são divididas em básicas e derivadas. As básicas são obtidas pelo esforço de testes sobre os dados brutos reunidos por analistas e utilizadas para o acompanhamento do status da evolução do projeto. Enquanto as derivadas são obtidas pelo gestor de testes, através da conversão das métricas básicas para a necessidade de mudanças nos processos.

As métricas básicas estão associadas à quantidade de casos de testes criados, executados, bloqueados e reexecutados, que passaram ou falharam e estão sob investigação. Já as métricas derivadas estão relacionadas ao percentual dos testes concluídos, da cobertura de testes e de casos de testes que passaram ou não, bem como das falhas na primeira execução, da efetividade e da eficiência dos testes, da taxa de defeitos descobertos e do custo de remoção dos defeitos, dos defeitos corrigidos, de todo o retrabalho utilizado.

As medidas de testes mais importantes encontram-se na cobertura e na qualidade. A de cobertura associa a abrangência dos testes e a qualidade, a confiabilidade, estabilidade e o desempenho dos objetivos dos testes. Uma avaliação da cobertura fornece uma medida que avalia a conclusão dos testes e a avaliação dos defeitos indica qual a qualidade do software.

Quando falamos em cobertura, temos dois tipos: a cobertura do código (code coverage) e a cobertura de testes (test coverage).

A cobertura de código tem por objetivo encontrar código não testado, e este tipo de atividade não é utilizada para metrificar a qualidade do software. Ela irá auxiliar na avaliação do conjunto de ferramentas de testes escrita para o código em questão. A cobertura de código nos ajuda na exploração de caminhos felizes, infelizes e alternativos, indicando código inútil, desnecessário e mal escrito. Ela ainda auxilia no aumento da cobertura de testes, pois identifica cenários não explorados, e permite que gaps nos requisitos, nos casos de testes e defeitos, sejam prevenidos defeitos nos estágios iniciais do ciclo de vida do software.

Agora falando sobre a cobertura de testes, que é diferente da cobertura de código, esta utiliza-se de métricas qualitativas, visando à medição da eficácia dos testes em confronto com os requisitos testados, e determina que os casos de testes cubram os requisitos a serem testados.

Voltando à cobertura de código, que nos diz o quanto nosso software está testado de forma subjetiva, temos alguns tipos importantes:

1. **Function**: responsável pela verificação da quantidade de funções que são chamadas.
2. **Statement**: responsável pela verificação da quantidade de instruções executadas pelo código.
3. **Branch**: verifica se todas as ramificações de uma estrutura de controle são executadas (if/else, switch case, for, while, repeat etc.).
4. **Condition**: verificação de cada subexpressão booleana. Se tanto verdadeiras quanto falsas foram realmente testadas.

Algo muito importante a se considerar nos temas métricas e cobertura é que por mais que planejemos, testemos, melhoremos nosso código e nossos requisitos, sempre esbarramos na grande pergunta: quando devemos parar de testar?

Percebemos que por mais que tenhamos métricas e coberturas de testes e de código, são questões que ainda deixam muitas equipes de desenvolvimento e testes de software inquietas. Há muitas questões em aberto, tais como:

* Qual o tempo restante para término dos testes?
* Quanto já testamos?
* Vamos conseguir encerrar os testes no prazo?
* Já testamos o suficiente?
* Quantos defeitos de fato foram corrigidos?
* Temos um bom plano de testes e uma boa ferramenta de testes?
* Quais defeitos são nossa prioridade?
* Quem encontrou qual defeito?
* Qual analista de testes encontrou mais defeitos?
* Quantos defeitos são encontrados pelos usuários?
* Qual o custo para encontrar e corrigir defeitos?
* Quais defeitos podem esperar?

Percebemos que por nossos questionamentos, eles de fato geram certa angústia para os responsáveis pelos testes. Nunca temos a resposta exata, porque estamos trabalhando com um nível de incertezas muito grande.

**TEMA 2 – BDD & CODE COVERAGE**

O Agile Testing possui quadrantes (Figura 2) que mostram a visão em quatro partes dos testes durante o ciclo de desenvolvimento de software. Dois desses quadrantes estão relacionados à tecnologia e outros dois aos negócios. Dentro dos quadrantes, há uma referência se os testes são manuais ou automatizados.

Figura 2 – Quadrantes dos métodos de testes ágeis

Diagrama

Descrição gerada automaticamente

Fonte: <adaptworks.com.br>.

No quadrante 1 (Q1), temos testes unitários e componentes de testes, representados por TDD como a principal técnica. Geralmente está anexo ao CI/CD. O quadrante 2 (Q2) está relacionado à qualidade externa da aplicação, que está visível aos usuários do software, enquanto o do quadrante 1 é interno. A principal técnica estabelecida neste quadrante é a BDD. Já o quadrante 2 (Q3) assegura que os requisitos foram atingidos. Neste caso, podemos utilizar UAT (User Acceptance Testing), testes exploratórios e de usabilidade. Aqui os usuários devem ser envolvidos. Finalmente, o quadrante 4 (Q4) corresponde a uma avaliação técnica com testes de segurança, carga e performance.

**2.1 BEHAVIOR DRIVEN DEVELOPMENT (BDD) E USER STORIES**

O BDD, do quadrante 2, segundo Agile Testing, é uma abordagem que funciona bem com métodos ágeis,  projetado para que a gestão e a entrega dos projetos de software tenham uma melhor comunicação entre clientes e desenvolvedores. Ele garante que projetos permaneçam sempre focados na entrega do que o negócio realmente precisa, e que todas as necessidades do usuário estejam atendidas. Nesta metodologia, os testes são importantes, mas os testes não são os elementos que conduzem o desenvolvimento. Seu objetivo é que as metas e os resultados para o cliente sejam definidos de forma clara.

Essa abordagem envolve pessoas no processo através de Outside-in Development (desenvolvemos de fora para dentro), do uso de exemplos para escrita do comportamento da aplicação ou unidades de código, na automatização dos exemplos para que o feedback seja rápido, do uso do verbo “deve” quando escrevemos o comportamento do software, para que as responsabilidades fiquem claras e permitem que as funcionalidades do software possam ser questionadas, e também pelo uso de mocks, stubs, fakes e dummies para substituição de códigos que ainda não foram escritos.

As funcionalidades são escritas seguindo um padrão conforme a Figura 3. Logo após, utilizamos a estrutura (GWT - Given, When, Then), conforme a Figura 4. Na estrutura GWT, definimos os cenários do negócio para criarmos os seus critérios de aceitação.

Figura 3 – Features

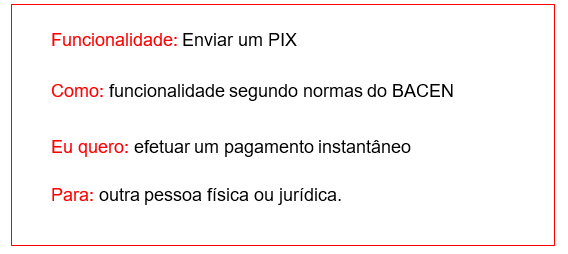
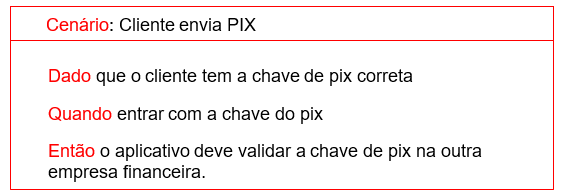


Figura 4 – Estrutura GWT



Podemos incrementar tais cenários com User Stories, e deixá-los mais precisos. Isso é bem interessante porque torna o critério de aceitação mais claro.

Cada User Story pode ter vários cenários, assim como um cenário pode ter várias User Stories. Tudo depende do que é abordado em cada US. O que importa mesmo é não abrirmos mão da objetividade para conseguirmos os melhores resultados.

Todas as User Stories que não possuam critérios de aceitação acabam suscetíveis a falhas durante o processo de aceite, além de possibilitarem erros do software em relação ao negócio para o qual ele está sendo construído. O método BDD auxilia na redução de falhas no momento da entrega do software.

**2.2 SONARQUBE UMA FERRAMENTA DE AUTOMATIZAÇÃO DE TESTES**

SonarQube é uma ferramenta open source que avalia a qualidade do código que está em desenvolvimento. Ela abrange arquitetura, código duplicado, potenciais problemas de erros no código, nível de complexidade do código e a cobertura dos testes. Ela possui um sistema de feedback de qualidade muito bom para todo o time de desenvolvimento.

Durante o processo de compilação do software, a SonarQube vai realizando análises e detectando por exemplo: partes do código que possam gerar bugs, duplicidade das linhas de comando e problemas na segurança. Ela possui versões Community (gratuita), developer, enterprise, data center e Sonar Cloud.

SonarQube é uma ferramenta multilinguagem (C/C++, Go, Java, PL/SQL, Python, dentre 29 diferentes linguagens), tem integração com Devops, integração contínua e qualidade centralizada, com ferramentas como Jenkins, Azure DevOps, Bitbucket, GitHub, Gitlab, entre outras.

Adota as métricas de complexidade, código duplicado, tamanho, cobertura de testes, índice de confiabilidade e segurança, índice de manutenibilidade e quantidade de problemas.

Esta ferramenta baseia-se em regras pré-definidas para a análise de código, geração de issues toda vez que um código quebra. Tais quebras de código são classificadas em:

1. **Bugs**: erros no código que precisam ser corrigidos.
2. **Vulnerabilities**: fraquezas de segurança.
3. **Code Smells**: problemas com a manutenção do código e que podem gerar dificuldade para quem precise alterar o código.

Além das categorias das issues, o SonarQube classifica os problemas por grau de severidade:

1. **Blocker**: impacto no comportamento do software e a correção deve ser imediata.
2. **Critical**: erro que pode afetar ou não o comportamento do software, mas pode representar falha de segurança e que deve ser corrigido imediatamente.
3. **Major**: falha que pode impactar na produtividade do desenvolvedor.
4. **Minor**: falha que pode afetar pouco na produtividade do desenvolvedor.
5. **Info**: uma descoberta que não é bug e nem falha.

Possui versões Community, Developer, Enterprise e Data Center dividido em dois produtos: Code Quality (Release Quality Code & Maintainability) e Code Security (Security Analysis e OWASP Top 10). São milhares de regras automatizadas para análise de código estático, que protege o software em várias frentes orientando o trabalho dos times de testes e desenvolvimento. Entre suas principais características, temos:

1. **Release Quality Code**: característica de captura de bugs complicados para auxiliar no tratamento de comportamentos indefinidos do software e que afetem os usuários.
2. **Application Security**: funcionalidade que corrige a vulnerabilidade de segurança que compromete o software durante o acesso.
3. **Technical Debt**: característica para promover um código limpo e de fácil manutenção para aumento de produtividade do desenvolvimento do software.

SonarQube é uma ferramenta com o maior número de integrações e também a mais complexa por conta de todas suas possibilidades de customizações. Além do SonarQube, temos outras ferramentas interessantes: Codacy e Code Climate.

**2.3 CODE COVERAGE**

Code coverage é uma métrica dentro dos testes automatizados e indica o percentual de código em produção que está coberto por testes. A ferramenta SonarQube cobre práticas de CI/CD como build/deploy nas pipelines de automação, dando assim cobertura para o resultado de análise de cobertura de testes.

Nossa agilidade na análise de cobertura de testes em builds aumenta e a visibilidade dos resultados em percentuais de cobertura é estabelecida. Caso haja algum problema neste limite de cobertura aceitável, podemos barrar a pipeline em determinada atividade.

É importante notar que há dois termos muito parecidos: cobertura de código (Code Coverage) e cobertura de testes. A cobertura de código é uma medida de código executada durante o teste e a cobertura de teste é uma medida de quanto do recurso que está sendo testado e que é coberto pelos testes.

A cobertura de código é objetiva, mas não nos diz o quão bem testado o software foi, enquanto a cobertura de teste é subjetiva e não quantifica a atividade.

A métrica de code coverage deve servir como um indicador de que as coisas estão indo mal, e o percentual muito baixo como um grande alerta de que o código está pouco testado, mas busque identificar a raiz do problema. Não há números mágicos para indicar um percentual muito baixo, apenas que quedas de 70% para baixo devem ser investigadas.

A qualidade do código, segundo novas tendências como o Clean Code, nos diz que um bom código é simples e eficiente. O SonarQube auxilia nesta inspeção contínua da qualidade do código para realização de revisões automáticas com análise estática em mais de 20 linguagens de programação.

**2.4 TESTES UNITÁRIOS**

Teste unitário é a fase de teste de cada unidade do software. O objetivo neste momento é o isolamento de cada parte do software com a ideia de garantir que cada pequena parte esteja funcionando conforme o especificado.

Este tipo de teste, assim como todos os demais, carece de um bom planejamento. Logo, o desenvolvedor deve fazer a avaliação sempre que pensar nos requisitos para cada funcionalidade a ser testada. Pensar sobre quais entradas e saídas queremos diante do processamento do fluxo dos dados. Normalmente conhecido como Unit, possui uma estrutura de testes automáticos unitários e consiste na verificação da menor unidade do projeto de software.

Unit Test é de responsabilidade dos desenvolvedores durante o processo de implementação do código. Ou seja, após a programação de uma classe, deve-se executar um teste unitário. No entanto, mesmo sendo responsabilidade de um dev, um QA deve estar comprometido na criação em conjuntos de testes unitários para contribuição do melhor desempenho do software. Sem robustez nos testes, a técnica falhará.

A redução da quantidade de bugs é considerável ao longo da implementação do software. Estes funcionam através da comparação de resultados esperados das funcionalidades com o código escrito.

É algo que vem se tornando cada dia mais elementar dentro da programação, e com isso, várias linguagens já possuem suas ferramentas de automatização de testes unitários, tais como:

1. Unit Testing Framework, Pytest e Locust para linguagem Python.
2. XCTest para Swift.
3. Test::Unit, RSpec e Minitest para Ruby.
4. Mocha, Jasmine, Jest, Protractor e Qunit para JavaScript.
5. PHPUnit para PHP.
6. NUnit para C#.
7. JUnit para Java.

Geralmente os testes de software são pensados antes da implementação do código, pois eles acabam nos orientando no desenvolvimento de cada classe do software.

**TEMA 3 – TESTES AUTOMATIZADOS, CYPRESS E CUCUMBER**

Mas afinal de contas, para que automatizar testes?

São inúmeras as razões, mas há uma pequena lista que já nos garante o entendimento dos principais motivos:

1. Testes manuais demoram muito.
2. Processos manuais são mais propensos a erros.
3. Os profissionais são liberados para outras atividades.
4. Testes automatizados de regressão fornecem uma rede de segurança.
5. Testes automatizados fornecem feedback antecipado e frequente.
6. Testes e exemplos orientam a codificação.
7. Testes geram documentações.
8. Testes podem ter um bom retorno sobre os investimentos.

Testes automatizados caracterizam o sucesso no mundo das metodologias ágeis. Isso porque um teste demora muito para ser executado, além claro de todo o seu planejamento. Processos manuais também são mais propensos a erros. A automação libera as pessoas para fazerem outras atividades e melhorarem seus trabalhos. Testes de regressão automatizados oferecem uma rede de segurança. Testes oferecem feedback com antecedência e com mais frequência. Exemplos e testes orientam na codificação. Testes são fontes de documentações. A automação traz um excelente retorno para os investimentos. Mas enfim, de fato, o que mais impulsiona uma empresa à automação dos testes é geralmente por conta da demora.

À medida que o software vai crescendo, o tempo para testes aumenta exponencialmente. As equipes ágeis entregam software a cada final das interações curtas e a aplicação dos testes de regressão automatizados, pelo menos diariamente, é uma prática indispensável. Se estes não forem automatizados, o tempo de execução dos testes ficará cada vez maior, bem como a frustração em relação às dívidas técnicas que não serão alcançadas a tempo ou talvez nunca. Lembrando que testes de regressão são retestes do software que foi modificado. É a maior parte de esforço nos testes de desenvolvimento de software e também uma parte essencial. Geralmente esse tipo de teste é incorporado à integração contínua. Tais testes geralmente são realizados por ferramentas como Selenium, SUnit, TestNG, PHPUnit, entre outros.

Na automatização dos testes, é bom lembrar que técnicas como TDD e SDD (desenvolvimento orientado a testes e orientados a testes de histórias) auxiliam muito no planejamento. Eles auxiliam no projeto do código e os testes garantem um conjunto de código refatorado e limpo. Eles colaboram com a questão das dívidas técnicas.

E o que poderíamos automatizar?

Quaisquer atividades tediosas e repetitivas dentro do desenvolvimento de software são uma candidata potencial à automação. Por exemplo, dentro do processo de integração contínua, compilações e implantações. O processo de implantação automatizada acelera os testes e reduz os erros.

Testes unitários e de componentes, utilizados geralmente com o TDD, são uma excelente opção de automatização. Eles criam projetos de código robusto e com alta qualidade. Testes de APIs ou Web Services são fortes candidatos à automação com execução de testes interativos e semiautomáticos. Testes por trás das Interfaces do Usuário, os quais automatizam regras de negócio. Testes das Interfaces de Usuário, Testes de carga e criação de dados e configurações iniciais.

Mas o que não seria bom automatizarmos?

Testes de usabilidade, que dependem de experiência e aspectos de julgamento de pessoas que entendem de usabilidade. Testes exploratórios que justamente são feitos para que aprendamos mais sobre o produto, usando informações para melhorar nosso software.

Porém, por mais que queiramos automatizar testes, nem sempre será fácil automatizá-los. Código mal escrito ou mal planejado produzem características não testáveis. Código legado que não separe regra de negócio, acesso ao banco de dados.

Escolha as ferramentas certas para cada situação e características do software a ser testado. Dê preferência à aplicação de princípios de testes ágeis para escolha de ferramentas. Mantenha tudo o mais simples possível, faça pequenas interações e mantenha o feedback. E finalmente, o time todo deve estar engajado em testes, metodologias ágeis de testes e ferramentas de testes.

**3.1 A FERRAMENTA CYPRESS**

O Cypress é uma ferramenta de testes de ponta a ponta para front-end (E2E) baseada em JavaScript.  Ele permite a execução de testes ponta a ponta, integração e unitários.

Conseguimos testar através de um navegador, código construído em React, Angular, Vue e outros frameworks que permitem a renderização pelo servidor. Ele tem acesso nativo a todos os objetos DOM (Document Object Model), suas funções e quaisquer outros elementos associados ao DOM. DOM está associado ao HTML.

Esta ferramenta permite uma viagem no tempo, fazendo prints durante a execução dos testes. Permite a depuração diretamente em ferramentas como Chrome DevTools. Recarrega automaticamente a cada alteração do teste. Espera automaticamente comandos e afirmações para prosseguir nos testes e em caso de falhas, a tela é capturada automaticamente, ou um vídeo é gravado com todo o conjunto de testes executado.

**3.2 A APLICAÇÃO DA FERRAMENTA CYPRESS NA EMPRESA GODADDY**

A GoDaddy conta com 18 milhões de clientes em todo o mundo e 77 milhões de nomes de domínio sob gestão. Ela conseguiu criar uma cultura de qualidade através da automação de testes com Cypress.

Ela possui uma equipe de trabalho complexa, pois seu construtor de websites em nuvem GoCentral é dedicado a pequenas empresas independentes. A carga de trabalho é grande e exaustiva e garantir a qualidade tornou-se algo imprescindível. É neste momento que entra a escolha do Cypress.

A escolha do Cypress se deu por alguns motivos:

1. Testes avançados de ponta a ponta;
2. Gravação dos testes;
3. Cobertura de testes bem abrangente;
4. Facilidade de adição de novas pipelines de integração contínua (CI).

Além, claro, da facilidade de uso, do painel para revisão dos resultados de testes e o suporte que obtiveram com a documentação e a grande comunidade. Algo impressionante e relatado pelo time foi a aproximação entre time de engenheiros e de controle de qualidade da empresa GoDaddy. O quanto a ferramenta os aproximou e os fez colaborar, garantindo uma excelente experiência.

Ao término da implantação da ferramenta, os times observaram que o Cypress tornou a vida de todos mais fácil. Eles agora são capazes de escrever testes automatizados de ponta a ponta obtendo mais recursos e fazendo muito menos testes manuais. Fazem as entregas do software com mais confiança, reduzem os bugs e ganham eficiência no uso da Dashboard que demonstra as execuções de testes e a identificação rápida de falhas de testes.

Com os ganhos obtidos na implantação da ferramenta Cypress em apenas um projeto da GoDaddy, ela agora planeja expandir o uso desta para todos os demais projetos da empresa.

**3.3 A FERRAMENTA CUCUMBER**

Cucumber é uma ferramenta focada no BDD (Behavior Driven Development) e que permite a descrição das necessidades reais dos usuários de tal forma que técnicos e não técnicos compreendam os fluxos de testes e fluxos de trabalho.

O Cucumber tem origem na comunidade Ruby e depois passou a ser aplicado em outras linguagens como Java e JavaScript. Desenvolvedores, analistas de negócios e clientes fazem parte do uso desta ferramenta.

É importante considerarmos a organização dos dados referentes aos testes, armazenando-os e formatando-os para os testes.

Para realização dos testes em BDD, precisamos seguir algumas etapas:

1. Escrever o comportamento esperado pelo código a ser testado.
2. Definir os passos que irão realizar o teste.
3. Escrever o código a ser testado.
4. Executar o teste.

O ponto central do Cucumber é estabelecer a inteligibilidade das especificações das features e dos cenários de testes. Tudo isso aumenta o nível de conhecimento dos requisitos e produz muitos benefícios para os times.

**TEMA 4 – TESTES AUTOMATIZADOS**

A arquitetura de microsserviços vem tomando um espaço considerável no âmbito da engenharia de software, e com ele as preocupações em adequação de testes que extraiam todas as possibilidades de bugs e falhas. No caso de microsserviços, a estratégia e o ambiente de testes devem se adaptar às muitas variáveis que são consideradas em relação à arquitetura do tipo monolito.

Quando testamos um monolito, podemos dividir as atividades de teste e testarmos módulos individualmente ou em grupos de componentes. Com microsserviços não funciona da mesma forma. Os componentes são gerenciados de forma interdependente para os testes, pois isso torna-os mais econômicos, uma vez que é necessário o uso de testes duplicados gerarem a necessidade de dependência como no caso real.

De um lado a arquitetura de microsserviços e de outro a infraestrutura baseada em containers (docker, por exemplo) gera uma combinação que exige a observação de vários detalhes. São dependências remotas e menos componentes processados.

Há uma combinação maior de técnicas associadas à arquitetura de microsserviços. Elas se comunicam pela rede, precisam de testes de impacto das conexões em detalhes. Normalmente temos as seguintes situações no ambiente de teste:

1. Equipes multifuncionais com desenvolvedores front-end, middleware, back-end, administrador de banco de dados e DevOps.
2. A governança descentralizada faz com que os times escolham suas ferramentas de acordo com suas necessidades.
3. Testes, deploy e infraestrutura altamente automatizados, com quase nenhuma intervenção manual.

Todas essas questões interferem na escolha das técnicas de testes a serem adotadas. Inevitavelmente, muitas das alternativas de testes estão intimamente relacionadas à arquitetura monolítica e precisam ser adaptadas à nova arquitetura. Dentre as quais utilizam soluções de testes duplicados com stubs, mocks ou serviços virtuais.

Os métodos disponíveis para arquiteturas de microservices são testes de containers, como testes de containers de banco de dados, testes de containers de virtualização de serviço e testes de containers de serviços terceiros (por exemplo, um teste de container Redis, um teste de container ESB ou um teste de container de dispositivo virtual) e os legados em sandbox.

As técnicas bem focadas em testes de micro serviços são:

1. Testes de containers;
2. Testes de containers de banco de dados;
3. Testes de containers de virtualização de serviço;
4. Testes de containers de serviços de terceiros (por exemplo, container ESB, container REDIS);
5. Testes de legados em sandbox.

A arquitetura de microsserviços, de fato, possui características peculiares e diferentes da arquitetura tradicional, porém, a grande questão encontra-se em adaptarmos aquilo que faz sentido e explorarmos todo o universo de testes ao redor dos containers para conseguirmos rastrear todo nosso processo de desenvolvimento, validando nosso produto através do que chamamos de *observabilidade*. Ou seja, uma visão 360 graus sobre tudo o que ocorre com o software, incluindo infraestrutura, entradas e saídas, gestão de logs e criando métricas apropriadas a este novo contexto.

Algumas ferramentas comerciais que seguem a ideia de observabilidade são chamadas de APM (*Application Performance Management*). Essa ferramenta possui um monitoramento unificado que rastreia e analisa front-end e back-end. Ela prepara um diagnóstico que facilita a correção dos problemas pensando inclusive na melhor experiência do nosso usuário. A APM descobre gargalos de desempenho em aplicações web e monitora a velocidade do ponto de vista do usuário e do back-end.

Dentre as ferramentas de APM encontradas no mercado, temos:

1. New Relic;
2. Datadog;
3. AppDynamics;
4. Scouter;
5. Pintpoint;
6. Loupe;
7. Stackify Retrace.

O contexto distribuído da arquitetura de microsserviços realmente requer ferramentas que auxiliem no processo de testes, pois há uma combinação muito perigosa de possíveis gaps que podem gerar bugs, defeitos ou problemas de experiência com nosso usuário. A rastreabilidade através de ferramentas de observabilidade, como as citadas anteriormente, auxiliam muito num trabalho que manualmente seria praticamente impossível.

**TEMA 5 – TESTES DE VULNERABILIDADE**

Web Application Vulnerability Scanners são ferramentas de automação de testes que verificam aplicativos da Web. Buscam por brechas e vulnerabilidades de segurança, tais como SQL Injection, Command Injection, configuração insegura de servidor e Cross-site Scripting. Esse tipo de ferramenta, também chamadas de *DAST* (Dynamic Application Security Testing), pode ser encontrada comercialmente ou de forma open source. O Projeto OWASP Benchmark aborda várias ferramentas do tipo DAST considerando sua eficácia na detecção de vulnerabilidades.

Algumas ferramentas que podemos encontrar dentro do projeto OWASP Benchmark:

1. APIsec (SaaS - comercial);
2. AppScan (Windows - comercial);
3. AppTrana Website Security Scan (Free - multiplataforma);
4. CloudDefense (SaaS / OnPremises - comercial - integra CI/CD);
5. Grabber (Open Source);
6. Zed Attack Proxy (Open Source, multiplataforma).

Essas ferramentas e muitas outras não listadas aqui prometem descobrir as vulnerabilidades presentes no ambiente no qual o software está inserido, buscando a identificação de quaisquer pontos fracos que o torne suscetível a ataques ou tentativas de hackers.

Os riscos que são apurados são classificados numa escala numérica e separados por níveis de gravidade.

Vulnerabilidade pode ser uma configuração ou qualquer outra característica específica dentro do software ou no ambiente no qual ele esteja inserido que pode se tornar um ponto de acesso ilegal. Hackers podem utilizar essas vulnerabilidades para roubar dados confidenciais, ou até mesmo manipular o software para trabalhar de acordo com suas vontades. É neste ponto que as ferramentas de testes de vulnerabilidade nos auxiliam alertando sobre todas as falhas preexistentes e onde elas se localizam.

Segundo o projeto OWASP, as vulnerabilidades mais comuns são:

1. Falhas de criptografia;
2. Injeção;
3. Quebra de controle de acesso;
4. Configuração insegura;
5. Projeto inseguro;
6. Falha da integridade dos dados;
7. Falha de identificação e autenticação;
8. Falsificação de solicitação do lado do servidor.

Ao utilizarmos os testes de vulnerabilidade, descobrimos nossos problemas de segurança e permitimos que sejam resolvidos antes que sejam atacados por hackers ou até mesmo por usuários mal-intencionados que descobriram brechas ao acaso.

Lembrando que estas questões de vulnerabilidade vão de encontro com a LGPD (Lei Geral de Proteção de Dados), HIPAA (Lei de Portabilidade e Responsabilidade de Seguros de Saúde), ISO 27001 (Norma Internacional para padrões de segurança) e a PCI-DSS (Padrão de segurança de dados da indústria de cartões de pagamento).

Para cobertura dos testes de vulnerabilidades, temos:

1. Avaliação de Rede e Wireless;
2. Verificação de Aplicativos;
3. Avaliação de Host;
4. Avaliação de Banco de Dados.

Refletindo que estamos com a maior parte de nossos produtos de software orientados a equipamentos móveis (incluindo smartphone que é o mais comum), IoT e Web Application, os testes de vulnerabilidade não são apenas mais uma opção em testes, mas são primordiais para garantir nossa segurança.

**FINALIZANDO**

A complexidade do nosso produto de software, seja ele para equipamentos móveis, IoT ou Web Application, nos direciona à utilização de ferramentas de testes automatizados. Diante do quadro de produtos e da necessidade de melhoria da experiência de nosso usuário, não há como fugirmos da adoção de ferramentas que garantam a maior cobertura de erros e de testes possíveis para nossos projetos de software.

O teste é mais que parte integrante do desenvolvimento de software, ele hoje garante maior segurança aos times de desenvolvimento, aos times de qualidade e à empresa como um todo. O transtorno com bugs, defeitos e quaisquer outras vulnerabilidades em relação à segurança e performance de nossos produtos pode ser catastrófico para as empresas, principalmente para as empresas que já nasceram digitais.

Técnicas, metodologias e ferramentas precisam estar alinhadas ao tipo de software para garantirmos a qualidade do software e também do processo de desenvolvimento.

Considerar técnicas como BDD, TDD, SBE e outras em paralelo aos métodos ágeis Scrum, XP e Kanban, por exemplo, pode nos auxiliar no ganho de produtividade de nossas equipes de desenvolvimento e dos times de testes. Essas metodologias proporcionam também maior sinergia entre os times de desenvolvimento e testes.

**QUALIDADE DE SOFTWARE**

AULA 6

Profª Maristela Weinfurter

**CONVERSA INICIAL**

**MÉTRICAS DE QUALIDADE APLICADAS EM DESENVOLVIMENTO COM MÉTODOS ÁGEIS**

O incrível mundo ágil traz consigo técnicas, ferramentas e métricas que nos auxiliam na construção de *software* com qualidade. Às vezes, temos a impressão de que trabalhar num modelo ágil é algo bagunçado. Talvez pela dinâmica acelerada tanto do desenvolvimento quanto das mudanças que ocorrem no dia a dia. E o contexto piora ainda se estivermos falando de uma *Startup* e de um MVP (*Minimum Viable Product*), lugar e produto totalmente orientado a mudanças constantes.

Vamos conversar e analisar algumas ferramentas e práticas relacionadas à qualidade de *software*, tais como *Sprint Burdown Chart*, *Velocity*, *Controle Chart*, *Lead Time*, *Blocked Time*, índice de débito técnico e dívida técnica.

Para falarmos sobre tais técnicas, ferramentas e artefatos, vamos elucidar conceitos inerentes especialmente à Metodologia *Scrum*, falando um pouco sobre os diversos papéis do time de desenvolvimento de *software* e também sobre as etapas que temos dentro da metodologia.

Mesmo que nosso foco seja mais a nível de *Scrum*, é importante lembrar que muitas empresas acabam não adotando unicamente uma metodologia ágil, mas tudo o que há de bom e alinhado ao negócio em questão. Encontraremos no mercado times utilizando termos e técnicas do *Scrum*, XP e Kanban no mesmo projeto. Isto porque já pudemos notar que há várias coisas interessantes em cada uma destas metodologias até aqui demonstradas.

A grande ideia é que possamos utilizar sempre o que for de melhor e mais aderente ao nosso projeto, não importando muito se é de uma metodologia ou de outra. Nosso foco é sempre construir um *software* com qualidade e dentro de processos com qualidade também.

Medir, discutir, mudar, validar, verificar e colaborar são os verbos mais utilizados para desenvolvermos *software* com foco nos requisitos de nosso cliente, com qualidade e bons resultados no todo.

**TEMA 1 – SPRINT BURDOWN CHART E O MÉTODO ÁGIL SCRUM**

*Sprint Burdown Chart* é uma ferramenta gráfica com origem na metodologia ágil *Scrum*. Ken Schwaber foi quem a criou para mostrar aos times de desenvolvimento o seu progresso dentro de cada *Sprint*. Apesar de ter sido criado junto à metodologia *Scrum*, é utilizado da mesma forma com Kanban, Lean ou XP. É possível visualizar o progresso das atividades da *Sprint* em geral (iterações, épicos, projetos ou releases). A Figura 1 lista os elementos do *Scrum Framework*.

Esse *framework* é dividido em papéis, cerimônias e artefatos. Os papéis ficam a encargo do *Product Owner* (PO), *Scrum master* e o próprio time. As cerimônias são do tipo: *Sprint planning*, *Sprint review*, *Sprint retrospective* e *Daily Scrum meeting*. E, finalmente, os artefatos atrelados ao *Scrum* são *Product backlog*, *Spring backlog* e *Burndown charts*. A Figura 2 ilustra como é o comportamento do gráfico.

Falando rapidamente sobre as cerimônias, temos o que se segue.

* ***Sprint Planning*:** é uma reunião que corresponde à primeira atividade de qualquer *Sprint*. Durante a *planning*, todos os objetivos desse novo ciclo são estabelecidos e o *Sprint Backlog* é criado, bem como o gráfico *Burdown* é inicializado. Nesta reunião, são feitas todas as negociações do time e definido o objetivo da *Sprint*. Uma vez tudo definido e aceito pelo time, o PO garante que a equipe mantenha suas atividades alinhadas aos itens do *Sprint Backlog*. Os três papéis listados na Figura 1 participam da *planning*. O PO aponta o *backlog* do produto e a equipe desenha em conjunto o *backlog* da *Sprint*. O *Scrum master* media as discussões. Uma vez todos de acordo, a *Sprint* é iniciada.
* ***Sprint Review***: uma *Sprint* de *Review* marca o encerramento da *Sprint* e o planejamento desta, que não deve durar mais do que 2 horas. Neste momento, a equipe apresenta o que foi desenvolvido, mostra novos recursos funcionando e demais artefatos. Aqui, vale ressaltar que, como se trata de entrega de *software*, deve estar funcionando. Não se pode considerar apenas apresentações estáticas do que seria o produto. Além do time envolvido na *Sprint*, é bem comum que outras partes interessadas apareçam nesta reunião para celebrarem as entregas, visto que, geralmente, estão ansiosos para verem os resultados, ou seja, novas *features* implementadas.
* ***Sprint Retrospective***: essa etapa corresponde de 15 a 30 minutos que o time todo discute o que está funcionando e o que não está funcionando na dinâmica das atividades. Ela é realizada na sequência da *Sprint Review*, e todos devem participar, inclusive as partes interessadas (clientes, CEO etc.). O *Scrum master* é responsável por essa atividade e questiona o grupo da seguinte forma:
  + O que gostaríamos de fazer na sequência?
  + O que gostaríamos de parar de fazer?
  + O que vamos continuar fazendo na próxima *Sprint*?
* ***Daily Scrum Meeting*:** a equipe se junta em um horário e local para reunião diária, que deve durar no máximo 15 minutos. Quando presencial, a *daily* é feita com os participantes de pé (*Stand Up Meeting*). Assim, garante-se que ela não leve mais de 15 minutos. O *Scrum master* facilita a *Daily* e questiona o time de forma simples:
  + O que fizeram ontem?
  + O que irão fazer hoje?
  + O que está bloqueado?

Um gráfico de *Burdown* é importante porque nos mostra como o trabalho está progredindo, mantém a equipe coesa em relação ao que está ocorrendo com as atividades e com cada envolvido e ajuda na observação do progresso que pode estar muito lento em relação ao planejado.

Ele é relativamente simples de ser interpretado e sua simplicidade o torna uma ferramenta excelente para o rastreamento do progresso das atividades. O gráfico marca no eixo horizontal os dias da *Sprint* e no eixo vertical os pontos do planejamento que compõem a *Sprint*. Sempre indo do máximo de pontos da *Sprint*, que corresponde à velocidade da equipe até zerar.

Figura 1 – *Scrum Framework*

Texto

Descrição gerada automaticamente

Crédito: Dmitry/Adobe Stock.

A primeira linha do gráfico é traçada por uma diagonal que vai do ponto máximo ao ponto zero do eixo vertical. A linha cinza do gráfico da Figura 2 representa essa diagonal.

Figura 2 – *Burndown Chart*

Interface gráfica do usuário

Descrição gerada automaticamente

Crédito: Bakhtiarzein/Adobe Stock.

Ao término do primeiro dia de trabalho, após a *daily*, o time deve calcular o total de pontos das US (*User Stories*, representada na Figura 3) concluídas para que a linha azul (Figura 2) comece sua descida. Esses pontos são o somatório das US finalizadas.

Porém, como toda ferramenta, possui algumas limitações. Por exemplo, os gráficos geralmente focam um pequeno pedaço do contexto total. Ou seja, se houver alguma alteração no escopo do projeto, nós não teremos clareza e indicação das mudanças no nosso escopo mais específico.

Eles demonstram apenas as horas de trabalho e não se as US foram completadas ou ainda estão em andamento. Sua utilidade está relacionada com a exatidão das estimativas, porém estimativas nem sempre são bem elaboradas. Outro problema é que o gráfico não demonstra os motivos do aumento de trabalho restante, que pode vir por conta do aumento da estimativa ou remoção de trabalhos concluídos.

Figura 3 – Exemplo de como descrever uma *User Story*

Texto

Descrição gerada automaticamente com confiança média

Crédito: AmaiaL/Shutterstock.

Agora, podemos tornar o *Burndown Chart* mais efetivo seguindo alguma dicas.

1. Escrever atividades pequenas no *Sprint Backlog*. Refine o quanto puder para que ela caiba dentro da *Sprint* e seja útil.
2. Ao escrever suas atividades, pense com clareza em todos os itens do *Product Backlog* (para o PO). Qualidade não deve ser só compromisso do time de SQA. Com isto, escreve de forma clara, objetiva e que realmente entregue valor.
3. Objetivos simples e curtos garantem um bom planejamento diário, gestão do *Sprint Backlog* e gestão de riscos.
4. Não parar de questionar e sanar todas as dúvidas para que a US seja desenvolvida com clareza nos objetivos.

O *Burndown Chart*, se aplicado adequadamente, é uma excelente ferramenta para auxiliar o PO e todo o time de desenvolvimento. Ele ajuda a promover reflexões sobre o processo, bem como a capacidade do time, objetivos diários e absorver demandas não planejadas.

**TEMA 2 – VELOCIDADE (VELOCITY) E O MÉTODO ÁGIL SCRUM**

*Velocity* é uma métrica bastante utilizada por times de desenvolvimento de *software* que utilizam o método *Scrum*. Geralmente, o *Scrum master*, por vezes, chamado de *Agile Coach*, é responsável pela mensuração do projeto e uma das perguntas a qual ele mais responde é sobre a velocidade de seu time.

Mas como podemos mensurar as atividades de nosso time? Em qual nível de detalhe precisamos descer? A forma mais simples e comum é a medição por meio do número de horas, embora a experiência de muitos diga que é uma forma falha. Muito empregada numa metodologia precursora do *Scrum*, chamada *Waterfall*.

Outra visão é a medição por meio de ***Story Points*** e, aqui, não devemos confundir com pontos por função (forma mais antiga de medir produtividade em *software*). *User Points* tem origem nas *User Stories*, também da metodologia *Waterfall*. *Story Points* são medidos com a utilização da sequência de Fibonacci como métrica. Juntamente, são feitas algumas questões, como: o projeto é parecido com algum anterior?

Como foi o esforço para executá-lo? Existe algum bloqueio que possa causar atrasos? E se dependemos de outras áreas ou fornecedores que podem afetar a conclusão de nossas atividades? Uma técnica utilizada para o planejamento das estimativas é o *Planning Poker*. Cada membro do time recebe uma pilha de cartas que *Story Points* diferentes. Uma pessoa faz a leitura dos projetos que a equipe tem definido. E aí cada integrante do time exibe uma carta que acredita que represente o esforço para concluir determinado projeto. A seguir, pontos importantes que devem serf levados em consideração.

1. *Story points* não são horas.
2. Média não é um bom resultado.
3. Ele não funciona muito bem com atividades pequenas ou enormes.
4. Manter os pontos sempre atualizados é muito importante.

A Figura 4 representa graficamente todos os conceitos mais importantes da metodologia *Scrum*.

Figura 4 – Processos da Metodologia *Scrum*

Interface gráfica do usuário, Diagrama, Aplicativo

Descrição gerada automaticamente

Crédito: Elaine333/Shutterstock.

Uma forma que nos parece engraçada no primeiro momento é a ***t-shirt size***, ou, traduzindo, tamanho de camiseta. Sim, você leu corretamente. Tomamos as medidas P, M, G etc. para cada item a ser mensurado, como no caso de camisetas para uma pessoa.

Também se faz uso da ***issue count***, a qual conta o número de itens, que é bastante utilizada no *framework* Kanban e sugere-se que seja utilizada com times mais experientes.

Aliás, a mensuração, de forma geral, sempre fica melhor quando já temos profissionais experientes no time, que conhecem o dia a dia do desenvolvimento, do controle de qualidade, do gerenciamento de projeto, enfim, de tudo na vida!

Agora, qual métrica vamos utilizar? O *Scrum Master* é quem adotará as técnicas conforme fiquem mais aderentes ao projeto e ao time. Para que a medição funcione, a grande questão encontra-se em sermos coerentes com nossas escolhas e não mudarmos constantemente.

E como já comentamos, queremos conhecer o ritmo do nosso time, em qual velocidade ele trabalha e consegue fazer as entregas com qualidade. De nada nos vale que haja uma entrega rápida se ela é parcialmente ou nada testada, se ela realmente entrega aquilo que estava proposto no *backlog*.

Uma vez que nos adequamos à nossa forma de medir, essa métrica irá auxiliar em muito no planejamento do *roadmap* do PO. Novamente, vamos frisar que é muito importante a experiência dentro do time, até porque metodologia ágil utiliza muito do empirismo. Ou seja, é baseado nas experiências passadas. Após a adoção da métrica, é importante que se passem duas a três *Sprints* para que então consigamos ter uma ideia se as métricas estão fazendo sentido.

Quando falamos em velocidade do time, devemos levar em consideração não somente as habilidades individuais de nossos devs, mas todo o contexto do projeto, a cultura da empresa e a coesão do time.

Algumas ferramentas de gerenciamento de projetos ágeis, como no caso do Jira, a velocidade pode ser calculada automaticamente no relatório de Gráfico de Velocidade, desde que seja parametrizado adequadamente. Ele pode ser configurado para utilizar *Story Points*, Horas ou *Issue Count*. O importante é que seja adequadamente configurado pelo *Scrum master* ou PO.

O que podemos fazer se nossos cálculos demonstrarem que nossa velocidade não está bem? O importante não é que saiamos tirando conclusões precipitadas, mas que possamos refletir junto ao time, tentando levantar o que está causando o problema de velocidade em nossas entregas.

E muito importante é sempre lembrarmos que somos um time e não um único desenvolvedor. Por vezes, um desenvolvedor experiente e que entregue rápido pode ser atrasado em suas entregas por outros motivos gerais.

**TEMA 3 – LEAD TIME & CYCLE TIME NO MÉTODO KANBAN**

*Lead Time* é um conceito muito difundido dentro da área de negócio de SCM (*Supply Chain Management*, Figura 5). Mais precisamente, dentro da área de logística, que, por sua vez, faz parte do SCM. Dentro desse contexto, *Lead Time* nos remete a custos de operações logísticas e que estão associados por exemplo a estoques de segurança.

No caso de um atraso dos processos logísticos, o processo fabril de um cliente gera falhas na produção. Sendo assim, a fábrica aplica o controle *Lead Time* com seu fornecedor. Logo, tanto a fábrica quanto o fornecedor acabam compartilhando informações-chave para que o processo não sofra atrasos.

Figura 5 – *Supply Chain Management* (SCM)

Interface gráfica do usuário, Aplicativo

Descrição gerada automaticamente

Crédito: Genestro/Adobe Stock.

**3.1 MÉTODO ÁGIL KANBAN**

O método ágil Kanban tem o objetivo de trabalhar com projetos que catalisem resultados enxutos dentro das empresas. O que queremos ao adotar uma metodologia Kanban é que nossos sistemas complexos se adaptem à cultura Kaizen. Esta é focada em melhoria contínua para que os resultados econômicos, sociológicos e dos negócios. Ou seja, usa cinco propriedades como condições para o estímulo do comportamento enxuto.

1. Visualização do fluxo de trabalho.
2. Limite do WIP.
3. Métricas e gerenciamento para o fluxo de trabalho.
4. Adoção de políticas de processo explícitas.
5. Adoção de modelos para reconhecimento de oportunidades de melhoria.

A metodologia ágil Kanban tem foco na qualidade, na redução dos trabalhos em progresso, nas entregas frequentes, no equilíbrio entre demanda e receita, gestão de prioridades e fontes de variabilidade para melhoria e previsibilidade.

Falando rapidamente sobre o formato do Kanban, temos:

1. *Delivery Planning*: reunião para planejamento das entregas das funcionalidades do *software*.
2. Kanban *Delivery*: (*downstream*) conversão das opções e execução do *backlog*.
3. Kanban *Meeting*: reunião curta e rápida, de preferência de pé como no *Scrum* e que tem por finalidade tomar decisões para que o trabalho flua.
4. Kanban *Discovery*: (*upstream*) é a discussão das opções de priorização das atividades para o time e para as entregas.

**3.2 MÉTRICAS *LEAN***

Para que haja efetividade no uso da metodologia Kanban, é importante que as métricas sejam levadas a sério. Estas precisam estabelecer que o time de desenvolvimento meça e quantifique sua produtividade. Para isto, devemos estabelecer KPIs (indicadores-chave de desempenho). Figura 6 exemplifica elementos importantes para construção e utilização de KPIs.

Figura 6 – Elementos de KPIs

Diagrama

Descrição gerada automaticamente

Crédito: Buffaloboy/Adobe Stock.

Kanban, aliada aos KPIs bem construídos geram impactos profundos na redução de prazos de entrega, aumento de produtividade e melhoria contínua.

Exemplos de KPIs dentro do *Lean* Kanban:

1. *Lead Time* (Tempo de Espera);
2. WIP (*Work in Progress*); e
3. *Throughput*(vazão de atividades entregues).

Cada time deve evoluir e experimentar métricas mais adequadas ao seu tipo de projeto e ao seu time de profissionais.

**3.3 WIP (WORKING IN PROGRESS)**

WIP (*Working In Progress*) compreende atividades de desenvolvimento que estão em execução ou parcialmente acabadas e ficam em uma fila de espera de concluídas para implantação (*deploy*).

Práticas *Lean* e metodologias ágeis sempre comentam que o ideal é limitarmos o trabalho em andamento, porque serve como um inventário que agrega valor ao cliente e pode se transformar em desperdício caso não seja mais necessário no futuro.

Então, aquilo que chamamos de sistema empurrada passa a ser chamado de puxado, no qual não é mais possível iniciarmos atividades antes de finalizarmos outra em andamento.

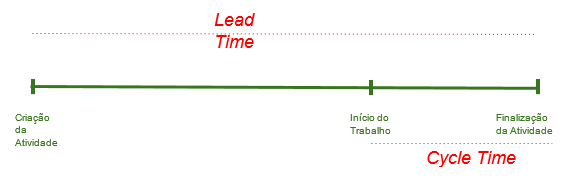
Por exemplo, no WIP podemos evitar desperdício trocando 10 atividades em andamento com 50% de conclusão, porém nenhuma concluída por cinco atividades completas e cinco que irão iniciar.

**3.4 LEAD & CYCLE TIME**

Dentro da área de desenvolvimento, o *lead time* está associado ao *framework* Kanban e que dentro do quadro de *backlog*, traça a tarefa em todas as suas fases: de sua criação até o *status* de DONE.

Do outro lado, temos o conceito de *Cycle Time* que determina o tempo que uma atividade será trabalhada, a partir do momento em que ela entrou em DOING até o momento da finalização (DONE). Neste caso, sempre menor ou igual ao *Lead Time*, conforme a Figura 7.

Figura 7 – Representação dos termos *Lead Time* e *Cycle Time*



Tanto *Lead Time* quanto *Cycle Time* são métricas que presumem a utilização de horas ou dias para finalização de cada atividade. O cliente se importa com o *Lead Time,* e o time técnico foca geralmente no *Cycle Time*. Lembrando que o esforço é sempre diferente do prazo. Ao concluirmos uma atividade em um dia, não representa que ela será entregue neste mesmo dia.

Ela será entregue ao cliente dentro do tempo do *Lead Time*, logo, esse *Lead Time* pode estar alto. E o que pode gerar esse número alto de horas ou dias para entrega? Bloqueios por conta de processos de infraestrutura, por exemplo, a atividade faz parte de uma demanda que deve ser entregue com outras demandas de um determinado tipo e WIP (*work in progress*) não limitado.

Voltando-nos novamente à origem do *Lead Time* e do *Cycle Time*, esbarramos novamente no Kanban. O sistema Kanban (sistema puxado com origem no método de produção Toyota) permite a visualização do fluxo de trabalho na busca de geração de valor. Já o método Kanban é uma abordagem evolutiva e incremental de mudanças.

O objetivo do Kanban é sempre reunir pessoas comprometidas em mudanças de problemas que sejam claras e explícitas. Com ele, conseguimos otimizar nossos fluxos de trabalho, melhorar nossa comunicação e colaboração, além de transformarmos o ambiente de trabalho para que seja mais previsível e estável. Para gerenciamento das atividades, utilizamos um quadro Kanban, conforme a Figura 8:

Figura 8 – Kanban *Board*

Gráfico

Descrição gerada automaticamente com confiança média

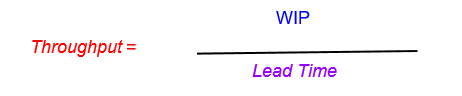
Crédito: Viktoriia Ablohina/Shutterstock.

*Lead Time* e *Cycle Time* fazem parte do *Lean* Kanban, que combina práticas para geração de valor e trabalho criativo de forma ágil com foco na maximização da entrega de valor para o cliente. Outros termos importantes para *Lean* Kanban são WIP e *Throughput*.

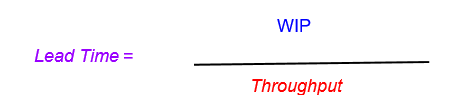
WIP (*Work in Progress*) define o trabalho em progresso, em execução naquele ponto do processo. É importante sua sinalização no quadro Kanban porque conseguimos visualizar a quantidade de tarefas que estão em andamento e confrontarmos com a capacidade de entrega do time. Quando limitamos o WIP, nosso trabalho torna-se mais estável e previsível, gerando confiabilidade nas entregas. O contrário cria filas maiores e aumento de risco nas entregas.

Outro termo importante dentro do Kanban é a métrica de *Throughput*. Essa métrica demonstra a quantidade de tarefas entregues num determinado período. A visão da *performance* do time fica em evidência e tudo aquilo que atrapalha a produtividade, as filas de atividades, é melhor compreendido e melhorado.

Para calcularmos o *Throughput* temos:



E para calcularmos o *Lead Time* temos:



Para melhorarmos o *Throughput,* precisamos melhorar a qualidade na validação das atividades anteriores, que deve ser menor, priorizá-las, eliminarmos etapas se for possível e classificarmos as atividades para identificação das prioridades.

Quando o WIP (*Work in Progress*) não é adequado e limitado, os *cards* são muito grandes e pouco concisas, entre outras situações, a variação do *Throughput* se torna instável.

Como podemos resolver o problema da variabilidade de *Throughput*? Por exemplo, determinando que *deploys* ocorrem num determinado dia da semana, estabelecer melhor o espaçamento e periodicidade das reuniões, bem como limitar seu tempo e também procurar manter o WIP mais enxuto.

**3.3 *CUMULATIVE FLOW DIAGRAM***

O CFD (*Cumulative Flow Diagram*) é um diagrama de fluxo acumulado, que representa o registro de forma cumulativa da quantidade de demandas que passam pelas etapas de fluxo de trabalho (TODO, DOING e DONE), Figura 6. Sua função é ser uma ferramenta analítica. Um dos artefatos mais comuns e utilizados no método Kanban.

Sua estrutura é muito simples, o qual propicia a visualização da capacidade de entrega por meio do *status* das tarefas. Muito bom para ajudar na identificação de gargalos ou bloqueios do time de desenvolvimento. No Eixo vertical, temos a quantidade de tarefas e no eixo horizontal a linha do tempo. Um bom fluxo inclina suavemente, sem quebras ou saltos.

Figura 9 – CFD

Gráfico, Gráfico de linhas

Descrição gerada automaticamente

É vantajoso utilizarmos o CFD quando precisamos deixar visualmente expostas às disfunções do fluxo de trabalho. Os gargalos são bem visíveis, o desbalanceamento entre as etapas do fluxo e a previsão de que o time está fora do caminho para entrega dos itens do *backlog*. São situações importantes para gerenciarmos e que, no caso de estarem fora do desejável, o gráfico o deixará bem exposto.

Cada etapa é representada por uma faixa gráfica de cor diferente. Os projetos normalmente estão divididos em estágios do desenvolvimento para acompanhamento, como em fila, em execução, em testes ou completadas. É um modelo que não é engessado e pode variar de acordo com a natureza de cada projeto, seguindo suas especificidades.

Ganhamos na visualização do andamento do projeto, no rastreamento de projetos ágeis e melhoria de foco, aumento de produtividade.

**TEMA 4 – DÍVIDA TÉCNICA**

Há dois conceitos importantes dentro do mundo ágil e é muito importante a discussão sobre estes, pois estão alinhadíssimos à agilidade e qualidade.

Apesar de adotarmos todos os critérios de qualidade possível, nem sempre conseguimos desenvolver o código ou requisitos considerando todos os fatores importantes, especialmente quando falamos do lançamento de um MVP. Às vezes, também precisamos lidar com custos, outras necessidades mais urgentes, enfim, são muitas as situações que podem nos levar a não implementarmos o *software* como desejávamos.

E nessa linha de raciocínio, temos o que resta disso tudo chamada dívida técnica, pois não temos condições de implementar tudo de uma única vez. Quanto mais complexo é a ideia de nosso código, maior será nossa dívida técnica. Precisamos contornar caminhos dentro do desenvolvimento de software com abordagens mais simplistas e menos completas para conseguirmos avançar rapidamente nas entregas. Porém, tais atalhos nos geram dívidas quase como dívidas financeiras na vida pessoal. Caso não sejam pagas, elas se acumularão e poderão alcançar níveis incontroláveis. Nosso código não deu muito certo, vai nos cobrar juros dessa dívida.

Esse conceito tomou largas proporções no mundo de desenvolvedores e consegue descrever exatamente a entropia (mundo **dev** em desordem) inevitável ao desenvolvimento de qualquer *software*. Às vezes, essa desordem ocorre por conta do projeto ao ser otimizado em decorrência de fatores externos, ou então por conta de escolhas não boas, ausência de informações importantes, revisões feitas sem a devida atenção e erros mesmo do desenvolvedor. Indiferentemente das razões, a dívida técnica é uma consequência inevitável no desenvolvimento de *software*.

Quanto mais a dívida técnica aumenta, o custo para efetuarmos a mudança cresce na mesma proporção. A dívida técnica é uma fonte de complexidade que aumenta além dos níveis ideais. A capacidade de resposta do cliente diminui e perdemos a agilidade nas entregas. O que é essencial num modelo ágil.

Um dos grandes desafios da dívida técnica é que é difícil quantificar e priorizar aquilo que ficou para trás. À primeira vista, todas as atividades parecem-nos essenciais e importantes. Há uma vertente que tenta criar uma espécie de monetização para dívida técnica e seu custo de reparo está relacionado ao número de linhas.

Por exemplo, se o código possui 10.000 linhas, o custo para corrigi-lo será de 20.000 dólares. Com essa situação de monetização da dívida técnica, estabelecemos um limite de crédito para a dívida técnica. E depende de como estejamos utilizando-a, podemos estabelecer que quando a dívida chega a 0,25 de dólar por linha de código, está na hora de suspendermos o desenvolvimento de novas *features*. A equipe então se concentra na refatoração para redução do débito técnico aceitável.

A dívida técnica foi cunhada nos anos 1992 por Ward Cunningham, como uma metáfora em relação ao prejuízo que temos com as coisas que ficam para trás no desenvolvimento de *software* e retornam no formato de correções ou adaptações e implementações.

Apesar de falarmos anteriormente sobre a dívida técnica a nível de código, esta pode ocorrer em função do planejamento do projeto, do desenho da arquitetura, da documentação, dos testes, dos defeitos, dos requisitos funcionais e não funcionais, da infraestrutura, do corpo técnico de profissionais, da automação de testes, da usabilidade, enfim, ela pode surgir de diversas fontes relacionadas ao desenvolvimento de *software*.

A métrica do débito técnico permite que a inspeção contínua do código ocorra por meio de um *feedback* muito rápido dentro do processo de desenvolvimento de *software*. Pode mudar a proficiência do processo de *software* para saída do processo. A avaliação qualitativa é substituída pela mediação quantitativa da qualidade de *software* com o uso do débito técnico. A introdução desse conceito melhora muito o gerenciamento do processo de desenvolvimento de *software*.

Vimos que o processo de “consertar” um *software* é caro, porém ignorar esse assunto pode custar muito mais caro. Uma prática da análise ágil diz que devemos priorizar as dívidas técnicas. Equipes ágeis então identificam, rastreiam, priorizam, monitoram e pagam sua dívida técnica. Geralmente, essas dívidas estão associadas às *User Stories*, equilibrando, assim, entre o novo desenvolvimento e o acerto da dívida.

Um rastreador então é definido sempre que uma dívida técnica é apresentada ou descoberta. Tanto o sistema de rastreamento de dívidas como o sistema de rastreamento de defeitos, não precisam ser complicados. Um gráfico simples na parede usando cartões de índice para capturar “histórias de dívidas técnicas” é uma maneira eficaz de um time gerenciar sua dívida técnica.

Geralmente o PO, com auxílio do time, prioriza as histórias de dívidas técnicas, bem como estima o valor da eliminação de cada uma. Ele ainda garante que a prioridade das novas histórias de usuário nem sempre supera a prioridade das histórias de dívida técnica. Os líderes de equipe técnica devem defender oportunidades para agendar histórias de dívidas em iterações de desenvolvimento. Podemos elencar alguns aspectos que nos auxiliam na mensuração da dívida técnica:

1. Duplicidade de código;
2. Coesão de código;
3. Fragilidade de *builds;*
4. Fragilidade de testes automatizados;
5. Tamanho dos métodos das classes; e
6. Cobertura ineficiente de testes unitários.

Algumas equipes de projetos ágeis alocam de 15 a 20% da capacidade da equipe em cada iteração para redução da dívida, deixando 80 a 85% para o desenvolvimento de histórias de usuários. Outras equipes ágeis ocasionalmente designam iterações explícitas de redução da dívida que evitam o desenvolvimento da história do usuário em favor do trabalho de redução da dívida. Ainda outras equipes ágeis encontram maneiras de eliminar a dívida técnica durante o desenvolvimento da história do usuário. Seja qual for a abordagem, é essencial monitorar e pagar continuamente a dívida técnica.

**TEMA 5 – OKR: (OBJECTIVES AND KEY RESULTS)**

Os objetivos e os resultados-chave (OKR) são a maneira como uma empresa pode atingir seus objetivos medidos pelos indicadores. OKRs garantem previsibilidade e mitigação para as chances de fracasso a longo prazo.

Objetivos relacionados à estratégia da empresa devem ser audaciosos e os resultados voltados aos produtos devem ser focados na operação. E tudo o que se transforma em objetivos deve ser mensurável, para que seja avaliado e as ações possam ser efetivas.

A Google foi a empresa que popularizou o OKR, seguida por inúmeras *startups* do Vale do Silício. Com a OKR, é possível vincular os objetivos estratégicos, indicadores, ações operacionais e riscos em uma mesma visão. Elementos da OKR são apresentados na Figura 10.

Figura 10 – Elementos de uma OKR

Diagrama

Descrição gerada automaticamente

Crédito: buffaloboy/Shutterstock.

A OKR é um processo de planejamento estratégico com resultados esperados para o negócio como um todo. Quando começamos a planejá-las, elas são estabelecidas inicialmente em alto nível com metas realistas para o negócio.

Estabelecemos uma KPI (*Key Performance Indicator*) com relação direta entre objetivos, metas, ações e indicadores. Ela é um modelo vivo, portanto, deve ser revista periodicamente para a avaliação das KPIs. As ações são avaliadas e adequadas para que sejam atingidas de fato.

O procedimento de elaboração da OKR segue um processo linear no qual estabelecemos objetivos, metas, indicadores, ações e riscos. Há um quadro no qual deve ser guiado por um interlocutor experiente na área técnica e que ajuda no entendimento do fluxo e na melhoria da reflexão sobre o que é prioritário para a organização. Para melhorar nossa compreensão, vamos observar o modelo do OKR Canvas na Tabela 1:

Tabela 1 – OKR Canvas

|  |  |
| --- | --- |
| Objetivos | Incluir: objetivos estratégicos e período de aferição. Trimestre, Quarter, Semestre. |
| Metas | Incluir: conjunto das metas a partir dos objetivos estratégicos e período de aferição. |
| Indicadores | Criar: a partir do cruzamento dos objetivos e metas traçadas como estratégias da organização. Forma: quantitativa para representar ganhos ou perdas realizadas. |
| Ações | Ações devem ser acompanhadas e consolidadas para períodos determinados. |
| Riscos | Sobre a possibilidade de não ocorrer da forma planejada. |

**5.1 DEFINIÇÃO DE KPIS**

KPIs devem ser parametrizáveis de forma que se consiga atingir as metas estabelecidas e seja possível a aferição disto. A ideia é proceder ao mapeamento das metas estratégicas para identificação dos resultados a serem obtidos. Incluindo os limites para que a aferição dos resultados estabeleça validade das ações propostas.

O Canvas apresentado anteriormente é a melhor forma de alinhar o entendimento comum dos times sobre as metas e objetivos a serem atingidos. E todo esse trabalho pode ser feito por meio de um roteiro.

1. Reunir pessoas que possam de fato contribuir para construção das OKRs.
2. Preparar um *Agile Think* OKR Canvas, técnica para mapear e estabelecer as OKRs.
3. Revisitar resultados obtidos, pois podem direcionar a riscos.
4. Divulgar os resultados das OKRs.

Organizar e implementar OKRs depende do conhecimento de seus conceitos, técnicas e ferramentas, porém a experiência sobre o negócio e sobre o que cada área faz toda diferença na definição, planejamento e obtenção dos resultados.

**FINALIZANDO**

Nesta aula, trabalhamos o lado das métricas ágeis. Conhecemos ferramentas, técnicas e conceitos que nos direcionam a pensar e planejar *software*, dentro da dinâmica de agilidade, mas com a garantia da qualidade que almejamos, tanto para o processo quanto para o produto.

As metodologias e *frameworks* ágeis, que foram evoluindo desde os anos 2000, nos trazem técnicas e ferramentas, de certa forma simples e fáceis de serem implantadas para quantificarmos o quanto estamos sendo produtivos e entregando produtos na hora certa com os requisitos certos e com um código limpo.

Vimos algumas ferramentas, como: *Sprint Burdown Chart*, *Velocity*, *Controle Chart*, *Lead Time*, *Blocked Time*, índice de débito técnico e dívida técnica. Também conversamos sobre o mundo das OKRs, que nos auxiliam a colocar foco estratégico em nossos KPIs.

Também foi possível observarmos que, na grande maioria das atividades relacionadas às metodologias ágeis, acabamos fazendo um *mix* de conceitos, técnicas e ferramentas, o que nos faz observar que dificilmente vamos utilizar apenas uma metodologia ágil.

O interessante é justamente adaptarmos tais metodologias ao nosso time de desenvolvimento e o time de SQA, aquilo que nos torna realmente ágeis com qualidade no desenvolvimento de *software*. *Scrum*, Kanban, XP e todos os demais possuem características similares, porém, com técnicas diferentes. É aí que o mercado acaba evoluindo para *Scrumban*, ou coisas do tipo.

Temos várias empresas trabalhando com a metodologia *Scrum* no sentido de processo de desenvolvimento de *software*, com adaptações, com programação em pares do XP para garantir melhoria na construção e validação do código, o quadro Kanban para estabelecer *done*, *doing*, *testing* e todas as etapas do desenvolvimento de *software* e um *mix* de ferramentas de qualidade ágeis que estudamos até aqui.

O importante, tanto no uso do modelo ágil quanto das métricas ágeis, é justamente aprender a adaptá-las à realidade do time e da cultura da empresa, além, claro, das questões financeiras e cronogramas que possuímos.

Questão 1/5 - Qualidade de Software

Algumas ferramentas comerciais que seguem a ideia de observabilidade são chamadas de APM (Application Performance Management). Esta ferramenta possui um monitoramento unificado que rastreia e analisa front-end e back-end. Ela prepara um diagnóstico que facilita a correção dos problemas pensando inclusive na melhor experiência do nosso usuário. A APM descobre gargalos de desempenho em aplicações web e monitora a velocidade do ponto de vista do usuário e do back-end.

Qual ferramenta abaixo representa uma ferramenta APM?

Nota: 0.0

|  |  |  |
| --- | --- | --- |
|  | A | Angular |
|  | B | React |
|  | C | Cucumber |
|  | D | Cypress |
|  | E | New Relic |

Questão 2/5 - Qualidade de Software

A norma ISO/IEC 15504 surgiu em 1991 e trazia consigo o pensamento da normatização de conceitos de facilitar as definições de características dos processos. Na sequência, em 1993, surgiu um projeto chamado SPICE (Software Process Improvement and Capability dEtermination) considerando três aspectos: auxiliar o projeto de norma 15504, inclusão de testes de campo e coleta de dados para consolidação da nova norma.  Como outras normas e padrões, o seu objetivo é focado na melhoria de processos. Utiliza-se da análise de resultados como pontos fortes e fracos, bem como sobre o gerenciamento de riscos.

SPICE é dividida em etapas, sendo assim, a seguir temos a definição da parte 5:

Nota: 20.0

|  |  |  |
| --- | --- | --- |
|  | A | Estabelece os requisitos mínimos para realização da avaliação com boa granularidade para o processo. Nesta fase há classificação dos dados, implantação de responsabilidades e o desenvolvimento de modelos para acompanhamento dos processos. |
|  | B | Traz uma introdução sobre os conceitos e avaliação dos processos, com destaque a um glossário que define os principais termos avaliativos do processo. |
|  | C | Estabelece um exemplo de PRM (Process Assessment Model), o qual resume num modelo de referência de processos o ciclo de vida dos processos estabelecidos pela norma.  **Você acertou!**  CORRETA: Conforme texto da Aula 03 no Tema 05 Parte 1 (Conceitos e vocabulário): traz uma introdução sobre os conceitos e avaliação dos processos, com um destaque para o glossário que define os principais termos avaliativos do processo.   Parte 2 (Avaliação de processos): estabelece quais requisitos mínimos para realização da avaliação para que haja uma boa granularidade para o processo. Nesta fase há a classificação de dados, implantação de responsabilidades, e o desenvolvimento de modelos para o acompanhamento dos processos.   Parte 3 (Recomendações de avaliação): fornece recomendações para avaliação dos requisitos.   Parte 4 (Recomendações para melhoria dos processos): Insere uma perspectiva para especificação dos processos, identificando os pontos fortes, fracos e os riscos.   Parte 5 (Exemplo de avaliação): esta parte estabelece um exemplo de PRM (Process Assessment Model), o qual resume num modelo de referência de processos o ciclo de vida dos processos estabelecido pela norma ISO/IEC 12207. Além do PRM, há também a exemplificação do PAM (Process Assessment Model), que estabelece fatores de avaliação e medição dos processos conforme o planejado no cronograma. |
|  | D | Insere uma perspectiva para especificação dos processos, identificando pontos fortes, fracos e riscos. |
|  | E | Fornece recomendações para avaliação dos requisitos. |

Questão 3/5 - Qualidade de Software

Usabilidade é um assunto tratado dentro da área de IHC (Interação Humano-Computador). 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. Jakob Nielsen formulou 10 heurísticas que são utilizadas nos testes de avaliação heurística e por todos os profissionais de UX Design & Researcher.

Quais heurísticas abaixo encontram-se dentro destas 10 elaboradas por Nielsen?

Nota: 20.0

|  |  |  |
| --- | --- | --- |
|  | A | Visibilidade do status do software.  **Você acertou!**  CORRETA: Conforme texto da Aula 04 no Tema 2.1  Usabilidade é um assunto tratado dentro da área de IHC (Interação Humano-Computador). 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:  Especificação e entendimento do contexto de uso  Especificação e entendimento dos requisitos da empresa e dos usuários  Desenvolvimento da solução de interação (IHC)  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.  Ao falarmos em usabilidade e testes de usabilidade, não tem como não nos referirmos ao pai da heurística, Jakob Nielsen. Ele formulou 10 heurísticas que são utilizadas no teste de avaliação heurística e por todos os profissionais de UX Design & Researcher.              As 10 heurísticas de Nielsen:  Visibilidade do status do software  Correspondência entre o software a vida real  Liberdade e controle do usuário  Consistência e padrões  Prevenção de erros  Reconhecimento e não lembrança  Flexibilidade e Eficiência  Estética e Design minimalista  Auxiliar usuários no reconhecimento, no diagnóstico e recuperação de erros  Ajuda e Documentação.              A aplicação da avaliação heurística utiliza-se destes 10 tópicos, e geralmente é feita por mais de um profissional, pois cada um possui perspectivas diferentes e engrandecedoras aos testes |
|  | B | Correspondência de qualidade. |
|  | C | Liberdade de testes. |
|  | D | Padrões de código. |
|  | E | Flexibilidade no desenvolvimento. |

Questão 4/5 - Qualidade de Software

As medidas de testes mais importantes encontram-se na cobertura e na qualidade. A de cobertura associa a abrangência dos testes e a qualidade, a confiabilidade, estabilidade e o desempenho dos objetivos dos testes.  Uma avaliação da cobertura fornece uma medida que avalia a conclusão dos testes e a avaliação dos defeitos indica qual a qualidade do software. Quando falamos em cobertura, temos dois tipos: a cobertura do código (code coverage) e a cobertura de testes (test coverage).

A cobertura de código tem por objetivo:

Nota: 20.0

|  |  |  |
| --- | --- | --- |
|  | A | Encontrar código não testado, atividade que é utilizada para metrificar a qualidade de software. Exploração de caminhos felizes, infelizes e alternativos, indicando código inútil, desnecessário e mal escrito. Previne defeitos nos estágios iniciais do ciclo de vida do software.  **Você acertou!**  CORRETA: Conforme texto da Aula 05 no Tema 1  As medidas de testes mais importantes encontram-se na cobertura e na qualidade. A de cobertura associa a abrangência dos testes e a qualidade, a confiabilidade, estabilidade e o desempenho dos objetivos dos testes.  Uma avaliação da cobertura fornece uma medida que avalia a conclusão dos testes e a avaliação dos defeitos indica qual a qualidade do software.  Quando falamos em cobertura, temos dois tipos: a cobertura do código (code coverage) e a cobertura de testes (test coverage).  A cobertura de código tem por objetivo encontrar código não testado, e este tipo de atividade não é utilizada para metrificar a qualidade do software. Ela irá auxiliar na avaliação do conjunto de ferramentas de testes de testes escrita para o código em questão. A cobertura de código nos ajuda na exploração de caminhos felizes, infelizes e alternativos, indicando código inútil, desnecessário e mal escrito. Ela ainda auxilia no aumento da cobertura de testes, pois identifica cenários não explorados, e permite que gaps nos requisitos, nos casos de testes e defeitos sejam prevenindo defeitos nos estágios iniciais do ciclo de vida do software.  Agora falando sobre a cobertura de testes, que é diferente da cobertura de código, esta utiliza-se de métricas qualitativas, visando a medição da eficácia dos testes em confronto com os requisitos testados, e determina que os casos de testes cubram os requisitos a serem testados. |
|  | B | Repassar código testado, atividade que é utilizada para metrificar a qualidade de software. Exploração de caminhos felizes, infelizes e alternativos, indicando código inútil, desnecessário e mal escrito. Previne defeitos nos estágios iniciais do ciclo de vida do software. |
|  | C | Encontrar código não testado, atividade que é utilizada para metrificar a qualidade de software. Exploração de unicamente caminhos alternativos, indicando código inútil, desnecessário e mal escrito. Previne defeitos nos estágios finais do ciclo de vida do software. |
|  | D | Encontrar código não testado, atividade que é utilizada para metrificar a qualidade de software. Exploração de unicamente caminhos alternativos, indicando código inútil, desnecessário e mal escrito. Previne defeitos nos estágios finais do ciclo de vida do software. |
|  | E | Repassar código testado, atividade que é utilizada para metrificar a qualidade de software. Exploração de unicamente caminhos alternativos, indicando apenas código e mal escrito. Previne defeitos nos estágios finais do ciclo de vida do software. |

Questão 5/5 - Qualidade de Software

Testes de integração são feitos para que todas as funcionalidades já testadas considerem agora requisições de HTTP, servidores, SGBDs, APIs externas, gerações de arquivos, envio de mensagens e outras ações que façam parte do conjunto de funcionalidades num contexto real.

Pensando em testes de integração, é correto afirmar que:

Nota: 20.0

|  |  |  |
| --- | --- | --- |
|  | A | Testes de integração devem ser totalmente automatizados. |
|  | B | Testes de integração são apenas manuais. |
|  | C | Testes de integração podem ser feitos de forma automatizada e/ou manual.  **Você acertou!**  CORRETA: Conforme texto da Aula 02 no Tema 3.2  Os testes de integração, como o próprio nome já deixa claro, sãos os testes que vamos fazendo na  integração entre todas as funcionalidades já testadas nos testes unitários considerando agora as  requisições HTTP, servidores, SGBDs, APIs externas, gerações de arquivos, envio de mensagens,  entre tantas outras ações que possam ser feitas no conjunto de funcionalidades totais do nosso  software.  Além de obviamente ser importante no momento do desenvolvimento de um novo software, é  também, não menos importante, no momento de quaisquer manutenções ou evoluções no software,  pois a probabilidade de alguma nova funcionalidade ser modificada e alterar o comportamento de  outras partes já existentes é consideravelmente grande.  Assim como quaisquer testes, os testes de integração podem ser feitos de forma manual, mas  também através de ferramentas. Estas open-source ou não, facilitam e agilizam este processo de  testes. |
|  | D | Ferramentas de automação de testes de integração são apenas open-source. |
|  | E | Ferramentas de automação de testes de integração são apenas proprietárias (comerciais). |

Questão 1/5 - Qualidade de Software

Todas as metodologias ágeis têm em seu cerne a ideia de desenvolvimento de software com base sólida na qualidade de software. O BDD (Behavior Driven Development) é uma técnica ágil alinhada à colaboração do time de desenvolvimento com times de qualidade e trabalham fortemente na verificação e validação. Dentre as práticas do BDD, temos o envolvimento das partes interessadas, a descrição do comportamento da aplicação ou de uma unidade de código, a automação de testes para obtenção de feedback rápido, testes de regressão, a escrita de comportamentos com Should (deve) e o uso de simuladores de teste (mocks, fakes, spies) para colaboração entre módulos não escritos ainda.

Mesmo que controverso, há quem diga que o BDD é uma extensão de qual técnica ágil?

Nota: 20.0

|  |  |  |
| --- | --- | --- |
|  | A | FDD |
|  | B | TDD  **Você acertou!**  CORRETA: Conforme texto da Aula 04 no Tema 1.1  Já o BDD (Behavior Driven Development), é outra técnica de desenvolvimento ágil alinhada à colaboração do time de desenvolvimento, com os times de qualidade e outros profissionais que possam estar relacionados ao conceito de verificação e validação. Mesmo controverso, pode ser visto como uma extensão da técnica TDD. A proposta do BDD encontra-se em onde inicia o processo, o que deve-se testar e não testar, quanto devemos testar de uma única vez, como as chamadas de testes devem ocorrer e a compreensão do porquê um teste falha.              Dentre as práticas do BDD, temos o envolvimento das partes interessadas, a descrição do comportamento da aplicação  ou de uma unidade de código, a automação de testes para obtenção de feedback rápido, testes de regressão, a escrita de comportamentos com Should (deve) e o uso de simuladores de teste (mocks, fakes, spies) para colaboração entre módulos não escritos ainda. |
|  | C | Kanban |
|  | D | Scrum |
|  | E | DSDM |

Questão 2/5 - Qualidade de Software

A norma ISO/IEC 15504 surgiu em 1991 e trazia consigo o pensamento da normatização de conceitos de facilitar as definições de características dos processos. Na sequência, em 1993, surgiu um projeto chamado SPICE (Software Process Improvement and Capability dEtermination) considerando três aspectos: auxiliar o projeto de norma 15504, inclusão de testes de campo e coleta de dados para consolidação da nova norma.  Como outras normas e padrões, o seu objetivo é focado na melhoria de processos. Utiliza-se da análise de resultados como pontos fortes e fracos, bem como sobre o gerenciamento de riscos.

SPICE é dividida em etapas, sendo assim, a seguir temos a definição da parte 5:

Nota: 20.0

|  |  |  |
| --- | --- | --- |
|  | A | Estabelece os requisitos mínimos para realização da avaliação com boa granularidade para o processo. Nesta fase há classificação dos dados, implantação de responsabilidades e o desenvolvimento de modelos para acompanhamento dos processos. |
|  | B | Traz uma introdução sobre os conceitos e avaliação dos processos, com destaque a um glossário que define os principais termos avaliativos do processo. |
|  | C | Estabelece um exemplo de PRM (Process Assessment Model), o qual resume num modelo de referência de processos o ciclo de vida dos processos estabelecidos pela norma.  **Você acertou!**  CORRETA: Conforme texto da Aula 03 no Tema 05 Parte 1 (Conceitos e vocabulário): traz uma introdução sobre os conceitos e avaliação dos processos, com um destaque para o glossário que define os principais termos avaliativos do processo.   Parte 2 (Avaliação de processos): estabelece quais requisitos mínimos para realização da avaliação para que haja uma boa granularidade para o processo. Nesta fase há a classificação de dados, implantação de responsabilidades, e o desenvolvimento de modelos para o acompanhamento dos processos.   Parte 3 (Recomendações de avaliação): fornece recomendações para avaliação dos requisitos.   Parte 4 (Recomendações para melhoria dos processos): Insere uma perspectiva para especificação dos processos, identificando os pontos fortes, fracos e os riscos.   Parte 5 (Exemplo de avaliação): esta parte estabelece um exemplo de PRM (Process Assessment Model), o qual resume num modelo de referência de processos o ciclo de vida dos processos estabelecido pela norma ISO/IEC 12207. Além do PRM, há também a exemplificação do PAM (Process Assessment Model), que estabelece fatores de avaliação e medição dos processos conforme o planejado no cronograma. |
|  | D | Insere uma perspectiva para especificação dos processos, identificando pontos fortes, fracos e riscos. |
|  | E | Fornece recomendações para avaliação dos requisitos. |

Questão 3/5 - Qualidade de Software

As medidas de testes mais importantes encontram-se na cobertura e na qualidade. A de cobertura associa a abrangência dos testes e a qualidade, a confiabilidade, estabilidade e o desempenho dos objetivos dos testes.  Uma avaliação da cobertura fornece uma medida que avalia a conclusão dos testes e a avaliação dos defeitos indica qual a qualidade do software. Quando falamos em cobertura, temos dois tipos: a cobertura do código (code coverage) e a cobertura de testes (test coverage).

A cobertura de código tem por objetivo:

Nota: 20.0

|  |  |  |
| --- | --- | --- |
|  | A | Encontrar código não testado, atividade que é utilizada para metrificar a qualidade de software. Exploração de caminhos felizes, infelizes e alternativos, indicando código inútil, desnecessário e mal escrito. Previne defeitos nos estágios iniciais do ciclo de vida do software.  **Você acertou!**  CORRETA: Conforme texto da Aula 05 no Tema 1  As medidas de testes mais importantes encontram-se na cobertura e na qualidade. A de cobertura associa a abrangência dos testes e a qualidade, a confiabilidade, estabilidade e o desempenho dos objetivos dos testes.  Uma avaliação da cobertura fornece uma medida que avalia a conclusão dos testes e a avaliação dos defeitos indica qual a qualidade do software.  Quando falamos em cobertura, temos dois tipos: a cobertura do código (code coverage) e a cobertura de testes (test coverage).  A cobertura de código tem por objetivo encontrar código não testado, e este tipo de atividade não é utilizada para metrificar a qualidade do software. Ela irá auxiliar na avaliação do conjunto de ferramentas de testes de testes escrita para o código em questão. A cobertura de código nos ajuda na exploração de caminhos felizes, infelizes e alternativos, indicando código inútil, desnecessário e mal escrito. Ela ainda auxilia no aumento da cobertura de testes, pois identifica cenários não explorados, e permite que gaps nos requisitos, nos casos de testes e defeitos sejam prevenindo defeitos nos estágios iniciais do ciclo de vida do software.  Agora falando sobre a cobertura de testes, que é diferente da cobertura de código, esta utiliza-se de métricas qualitativas, visando a medição da eficácia dos testes em confronto com os requisitos testados, e determina que os casos de testes cubram os requisitos a serem testados. |
|  | B | Repassar código testado, atividade que é utilizada para metrificar a qualidade de software. Exploração de caminhos felizes, infelizes e alternativos, indicando código inútil, desnecessário e mal escrito. Previne defeitos nos estágios iniciais do ciclo de vida do software. |
|  | C | Encontrar código não testado, atividade que é utilizada para metrificar a qualidade de software. Exploração de unicamente caminhos alternativos, indicando código inútil, desnecessário e mal escrito. Previne defeitos nos estágios finais do ciclo de vida do software. |
|  | D | Encontrar código não testado, atividade que é utilizada para metrificar a qualidade de software. Exploração de unicamente caminhos alternativos, indicando código inútil, desnecessário e mal escrito. Previne defeitos nos estágios finais do ciclo de vida do software. |
|  | E | Repassar código testado, atividade que é utilizada para metrificar a qualidade de software. Exploração de unicamente caminhos alternativos, indicando apenas código e mal escrito. Previne defeitos nos estágios finais do ciclo de vida do software. |

Questão 4/5 - Qualidade de Software

Teste unitário é a fase de teste de cada unidade do software. O objetivo neste momento é o isolamento de cada parte do software com a ideia de garantir que cada pequena parte esteja funcionando conforme o especificado. Unit Test é de responsabilidade dos desenvolvedores durante o processo de implementação do código.

Quais ferramentas abaixo são para testes unitários e correspondem à linguagem de programação de forma correta?

Nota: 20.0

|  |  |  |
| --- | --- | --- |
|  | A | Jasmine para JavaScript, PHPUnit para PHP e JUnit para Java.  **Você acertou!**  CORRETA: Conforme texto da Aula 05 no Tema 2.4 Teste unitário é a fase de teste de cada unidade do software. O objetivo neste momento é o  isolamento de cada parte do software com a ideia de garantir que cada pequena parte esteja  funcionando conforme o especificado. Este tipo de teste, assim como todos os demais, carece de um bom planejamento. Logo, o  desenvolvedor deve fazer a avaliação sempre que pensar nos requisitos para cada funcionalidade a  ser testada. Pensar sobre quais entradas e saídas queremos diante do processamento do fluxo dos  dados. Normalmente conhecido como Unit, possui uma estrutura de testes automáticos unitários e  consiste na verificação da menor unidade do projeto de software. Unit Test é de responsabilidade dos desenvolvedores durante o processo de implementação do  código. Ou seja, após a programação de uma classe, deve-se executar um teste unitário. No entanto,  mesmo sendo responsabilidade de um dev, um QA deve estar comprometido na criação em  conjuntos de testes unitários para contribuição do melhor desempenho do software. Sem robustez  nos testes, a técnica falhará. A redução da quantidade de bugs é considerável ao longo da implementação do software. Estes  funcionam através da comparação de resultados esperados das funcionalidades com o código escrito.  É algo que vem se tornando cada dia mais elementar dentro da programação e com isto, várias  linguagens já possuem suas ferramentas de automatização de testes unitários, tais como: Unit Testing Framework, Pytest e Locust para linguagem Python. XCTest para Swift. Test::Unit, RSpec e Minitest para Ruby. Mocha, Jasmine, Jest, Protractor e Qunit para JavaScript. PHPUnit para PHP. NUnit para C#. JUnit para Java. |
|  | B | NUnit para Java, RSpec para Swift, XCTest para Ruby. |
|  | C | PyUnit para Python, XCTest para Java e Mocha para C#. |
|  | D | Jasmine para Java, PHPUnit para PHP e JUnit para Python. |
|  | E | Pytest para Python, XCTest para Java e Mocha para C++. |

Questão 5/5 - Qualidade de Software

Testes de integração são feitos para que todas as funcionalidades já testadas considerem agora requisições de HTTP, servidores, SGBDs, APIs externas, gerações de arquivos, envio de mensagens e outras ações que façam parte do conjunto de funcionalidades num contexto real.

Pensando em testes de integração, é correto afirmar que:

Nota: 0.0

|  |  |  |
| --- | --- | --- |
|  | A | Testes de integração devem ser totalmente automatizados. |
|  | B | Testes de integração são apenas manuais. |
|  | C | Testes de integração podem ser feitos de forma automatizada e/ou manual.  CORRETA: Conforme texto da Aula 02 no Tema 3.2  Os testes de integração, como o próprio nome já deixa claro, sãos os testes que vamos fazendo na  integração entre todas as funcionalidades já testadas nos testes unitários considerando agora as  requisições HTTP, servidores, SGBDs, APIs externas, gerações de arquivos, envio de mensagens,  entre tantas outras ações que possam ser feitas no conjunto de funcionalidades totais do nosso  software.  Além de obviamente ser importante no momento do desenvolvimento de um novo software, é  também, não menos importante, no momento de quaisquer manutenções ou evoluções no software,  pois a probabilidade de alguma nova funcionalidade ser modificada e alterar o comportamento de  outras partes já existentes é consideravelmente grande.  Assim como quaisquer testes, os testes de integração podem ser feitos de forma manual,  mas também através de ferramentas. Estas open-source ou não, facilitam e agilizam este  processo de testes. |
|  | D | Ferramentas de automação de testes de integração são apenas open-source. |
|  | E | Ferramentas de automação de testes de integração são apenas proprietárias (comerciais). |

Questão 1/5 - Qualidade de Software

As medidas de testes mais importantes encontram-se na cobertura e na qualidade. A de cobertura associa a abrangência dos testes e a qualidade, a confiabilidade, estabilidade e o desempenho dos objetivos dos testes.  Uma avaliação da cobertura fornece uma medida que avalia a conclusão dos testes e a avaliação dos defeitos indica qual a qualidade do software. Quando falamos em cobertura, temos dois tipos: a cobertura do código (code coverage) e a cobertura de testes (test coverage).

A cobertura de código tem por objetivo:

Nota: 20.0

|  |  |  |
| --- | --- | --- |
|  | A | Encontrar código não testado, atividade que é utilizada para metrificar a qualidade de  software. Exploração de caminhos felizes, infelizes e alternativos, indicando código inútil,  desnecessário e mal escrito. Previne defeitos nos estágios iniciais do ciclo de vida do  software.  **Você acertou!**  CORRETA: Conforme texto da Aula 05 no Tema 1  As medidas de testes mais importantes encontram-se na cobertura e na qualidade. A de cobertura  associa a abrangência dos testes e a qualidade, a confiabilidade, estabilidade e o desempenho dos  objetivos dos testes.  Uma avaliação da cobertura fornece uma medida que avalia a conclusão dos  testes e a avaliação dos defeitos indica qual a qualidade do software.  Quando falamos em cobertura, temos dois tipos: a cobertura do código (code coverage) e a  cobertura de testes (test coverage).  A cobertura de código tem por objetivo encontrar código não testado, e este tipo de atividade não é  utilizada para metrificar a qualidade do software. Ela irá auxiliar na avaliação do conjunto de  ferramentas de testes de testes escrita para o código em questão. A cobertura de código nos ajuda na  exploração de caminhos felizes, infelizes e alternativos, indicando código inútil, desnecessário e  mal escrito. Ela ainda auxilia no aumento da cobertura de testes, pois identifica cenários não  explorados, e permite que gaps nos requisitos, nos casos de testes e defeitos sejam prevenindo  defeitos nos estágios iniciais do ciclo de vida do software.  Agora falando sobre a cobertura de testes, que é diferente da cobertura de código, esta utiliza-se de  métricas qualitativas, visando a medição da eficácia dos testes em confronto com os requisitos  testados, e determina que os casos de testes cubram os requisitos a serem testados. |
|  | B | Repassar código testado, atividade que é utilizada para metrificar a qualidade de software.  Exploração de caminhos felizes, infelizes e alternativos, indicando código inútil,  desnecessário e mal escrito. Previne defeitos nos estágios iniciais do ciclo de vida do  software. |
|  | C | Encontrar código não testado, atividade que é utilizada para metrificar a qualidade de  software. Exploração de unicamente caminhos alternativos, indicando código inútil,  desnecessário e mal escrito. Previne defeitos nos estágios finais do ciclo de vida do  software. |
|  | D | Encontrar código não testado, atividade que é utilizada para metrificar a qualidade de  software. Exploração de unicamente caminhos alternativos, indicando código inútil,  desnecessário e mal escrito. Previne defeitos nos estágios finais do ciclo de vida do  software. |
|  | E | Repassar código testado, atividade que é utilizada para metrificar a qualidade de software.  Exploração de unicamente caminhos alternativos, indicando apenas código e mal escrito.  Previne defeitos nos estágios finais do ciclo de vida do software. |

Questão 2/5 - Qualidade de Software

A ISO 12207 é uma norma que certifica sistemas de gestão de qualidade. Esta norma especifica fatores relacionados aos requisitos dentro das atividades de desenvolvimento de software. Ela possui quatro níveis principais: processos fundamentais, processos de apoio, processos organizacionais e processos de adaptação.

Quais são os subníveis do nível processos de adaptação?

Nota: 20.0

|  |  |  |
| --- | --- | --- |
|  | A | Processo de certificação, validação e verificação de erros, defeitos e falhas. |
|  | B | Identificação do ambiente do projeto, solicitação de informações, seleção de processos,  atividades e tarefas e documentação das decisões e motivos de adaptação  **Você acertou!**  CORRETA: Conforme texto da Aula 03 no Tema 2 Processos de Adaptação são os processos que observam a realidade de cada organização e definem  seus ciclos de desenvolvimento de acordo com as peculiaridades mais convenientes aos processos  do ciclo de vida, e é subdividido em:   Identificação do ambiente do projeto: refere-se à análise e adaptação de fatores relacionados a  pessoas, requisitos, software, políticas de qualidade, pontos estratégicos e cultura organizacional.   Solicitação de informações: refere-se às informações decorrentes para o processo de implantação e  adaptação dos requisitos da norma ao ciclo de vida do desenvolvimento.   Seleção de processos, atividades e tarefas: refere-se aos processos, suas atividades e tarefas que  devem possuir documentos contendo características e responsáveis por desenvolvê-las e executá-las.   Documentação das decisões e motivos de adaptação: refere-se ao plano de viabilidade adaptativa,  análises de custos e riscos. |
|  | C | Processo de gerência, de infraestrutura, de melhoria e de treinamento. |
|  | D | Processo de aquisição, fornecimento, desenvolvimento, operação e manutenção. |
|  | E | Documentação, gerência de configuração, gerência de qualidade, processo de verificação, processo de validação, processo de revisão conjunta, processo de auditoria, processo de resolução de problemas. |

Questão 3/5 - Qualidade de Software

Testes de integração são feitos para que todas as funcionalidades já testadas considerem agora requisições de HTTP, servidores, SGBDs, APIs externas, gerações de arquivos, envio de mensagens e outras ações que façam parte do conjunto de funcionalidades num contexto real.

Pensando em testes de integração, é correto afirmar que:

Nota: 0.0

|  |  |  |
| --- | --- | --- |
|  | A | Testes de integração devem ser totalmente automatizados. |
|  | B | Testes de integração são apenas manuais. |
|  | C | Testes de integração podem ser feitos de forma automatizada e/ou manual.  CORRETA: Conforme texto da Aula 02 no Tema 3.2  Os testes de integração, como o próprio nome já deixa claro, sãos os testes que vamos fazendo na  integração entre todas as funcionalidades já testadas nos testes unitários considerando agora as  requisições HTTP, servidores, SGBDs, APIs externas, gerações de arquivos, envio de mensagens,  entre tantas outras ações que possam ser feitas no conjunto de funcionalidades totais do nosso  software.  Além de obviamente ser importante no momento do desenvolvimento de um novo software, é  também, não menos importante, no momento de quaisquer manutenções ou evoluções no software,  pois a probabilidade de alguma nova funcionalidade ser modificada e alterar o comportamento de  outras partes já existentes é consideravelmente grande.  Assim como quaisquer testes, os testes de integração podem ser feitos de forma manual, mas  também através de ferramentas. Estas open-source ou não, facilitam e agilizam este processo de  testes. |
|  | D | Ferramentas de automação de testes de integração são apenas open-source. |
|  | E | Ferramentas de automação de testes de integração são apenas proprietárias (comerciais). |

Questão 4/5 - Qualidade de Software

O CMMI (Capability Maturity Model Integration) é um modelo com práticas importantes para maturidade de disciplinas específicas em engenharia de software, engenharia de sistemas, desenvolvimento integrado de processo e produto e escolha de fornecedores. Modelo administrado pelo Instituto CMMI, baseia-se em melhores práticas para o desenvolvimento e manutenção de software. Dividido em cinco níveis de maturidade, as quais atestam o grau de maturidade dos processos de uma empresa.

O Nível 5 está relacionado a:

Nota: 20.0

|  |  |  |
| --- | --- | --- |
|  | A | Todos os critérios estabelecidos no nível um já foram satisfeitos e todos os processos estão  de acordo com a política definida para a organização. |
|  | B | Todos os critérios estabelecidos no nível três já foram satisfeitos e o processo é gerenciável,  fazendo uso de medição e avaliação quantitativa. |
|  | C | Todos os critérios do nível quatro foram satisfeitos e os processos são adaptados e  otimizados, fazendo uso de meios quantitativos para atender à mudança de necessidades do  cliente e melhoria contínua do processo.  **Você acertou!**  CORRETA: Conforme texto da Aula 03 no Tema 3  O CMMI (Capability Maturity Model Integration) é um modelo que contém práticas  importantes para maturidade de disciplinas específicas tais como engenharia de software,  engenharia de sistemas, desenvolvimento integrado de processo e produto e escolha de fornecedores.  Este modelo é administrado pelo Instituto CMMI. Ele baseia-se em melhores práticas para o  desenvolvimento e manutenção de software e está dividido em cinco níveis de maturidade, as  quais atestam o grau de maturidade dos processos de uma empresa. (Figura 6)  Seu foco é melhorar continuamente o processo de uma empresa. Seu surgimento foi na  década de 1980, tendo sido o ano de 1991 o marco para o desenvolvimento das CMMs, as quais  antecedem o CMMI. A constituição do CMMI tem por base o SW-CMM (SEI Software CMM),  EIA SECM (Electronic Industries Alliances’s Systems Engineer Capability Model) e IPD-CMM  (Integrated Product Development CMM).  “Nível 0: Incompleto: os processos não funcionam ou não atingem todas as metas e objetivos  definidos pela CMMI.  Nível 1: Executado: os processos definidos pela CMMI já estão sendo executados com tarefas que  produzem artefatos definidos.  Nível 2: Controlada: todos os critérios estabelecidos no nível 1 já foram satisfeitos e todos os  processos estão de acordo com a política definida para a organização. Os colaboradores estão  executando seu trabalho com acesso a recursos adequados e  as partes interessadas estão envolvidas ativamente nos processos de desenvolvimento. Além de que  as tarefas e o software são monitorados, controlados e  revisados em conformidade com os processos da CMMI para este nível.  Nível 3: Definido: todos os critérios estabelecidos no nível 2 já foram satisfeitos e o processo  de desenvolvimento é adaptado com base no conjunto de processos agora padronizados de acordo com a cultura organizacional da empresa. O software, a  mensuração e outras questões de melhoria do processo já agregam valor ao processo organizacional.  Nível 4: Gerenciado quantitativamente: todos os critérios estabelecidos no nível 3 já foram  satisfeitos e o processo é gerenciável fazendo uso de medição e avaliação quantitativa.  Objetivos quantitativos foram estabelecidos e o desempenho do processo é  amparado em critérios de controle do processo.  Nível 5: Otimizado: todos os critérios do nível 4 foram satisfeitos e os processos agora são  adaptados e otimizados, fazendo uso de meios quantitativos (estatísticos) para atender à mudança de necessidades do cliente e melhoria contínua do processo. |
|  | D | Todos os critérios do nível dois já foram satisfeitos e o processo de desenvolvimento é  adaptado com base no conjunto de processos padronizados de acordo com a cultura organizacional da empresa. |
|  | E | Processos que não funcionam ou não atingem todas as metas e objetivos definidos pela  CMMI. |

Questão 5/5 - Qualidade de Software

O protótipo permite que a comunicação entre usuários e times de desenvolvimento melhorem e desta forma contribuindo para implementação de software mais aderente às necessidades reais dos usuários. O protótipo pode ser feito em baixa fidelidade, média e alta fidelidade. Em processo incremental, a prototipação vai melhorando de acordo com as reuniões com os usuários.

Como um protótipo de média fidelidade também é conhecido e quais suas características?

Nota: 20.0

|  |  |  |
| --- | --- | --- |
|  | A | O protótipo de média fidelidade é muito similar ao de baixa fidelidade com o uso apenas de  papel e lápis. |
|  | B | Mockup e utiliza-se de ferramentas automatizadas, porém o comportamento de interação da  UI fica para o protótipo de alta fidelidade. |
|  | C | Mockup e que através de ferramentas automatizadas (Sketch, Adobe XD, InVision)  conseguem simular comportamentos de interação das UIs.  **Você acertou!**  CORRETA: Conforme texto da Aula 04 no Tema 2  O protótipo permite a melhora na comunicação com os usuários, pois de forma rápida, as UIs  (User Interface) 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.  …              Uma vez elaborados os protótipos de baixa fidelidade e validados com os usuários, podemos  passar para os de média fidelidade (mockups). Estes já são melhores que os de baixa fidelidade,  pois através de ferramentas automatizadas, conseguem simular comportamentos de interação das  UIs. A sensação do usuário já melhora muito em relação ao tipo de protótipo anterior.  Para desenvolvermos estes 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. Estas UIs  são desenvolvidas diretamente em linguagem de programação, linguagem de marcação e com  frameworks alinhados com as linguagens escolhidas. Neste momento, o realismo é bem próximo do  que será o software com todas as suas features implementadas. A figura 6 exemplifica uma UI  em alta fidelidade. |
|  | D | Mockup através da utilização de papel e lápis com simulação da interação das UIs  manualmente. |
|  | E | Balsamiq através da utilização de papel e lápis com simulação da interação das UIs  manualmente. |