QUALIDADE DE SOFTWARE

AULA 6

Profª Maristela Regina Weinfurter Teixeira

CONVERSA INICIAL

O teste é parte integrante do desenvolvimento de *software*, juntamente com a codificação, as operações, os requisitos do cliente, entre outros. 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 das práticas que auxiliam times de desenvolvimento e qualidade, por exemplo:

- 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:

- fornecer *feedback* contínuo;
- entregar valor ao cliente;
- habilidades em comunicação com os clientes;
- coragem;
- simplicidade;
- praticar a melhoria contínua;
- responder à mudança;
- organização;
- foco nas pessoas;

- 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.

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. Além disso, a responsabilidade é sempre de todos pela qualidade, não somente do time de *Quality Assurance* (QA).

Para conseguirmos atingir os objetivos e ideais da área de testes ágeis, devemos prestar atenção na automação de testes. Mas por que precisamos automatizá-los? Porque isso garante que não haja regressões no *software*, bem como o *feedback* é mais rápido, há economia de tempo executando testes repetidos, e ainda, trabalhos manuais podem gerar erros pela fadiga do profissional, entre outras razões.

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

TEMA 1 – ESTRATÉGIAS DE TESTES

O modelo ágil orienta que não há a necessidade de a criação de um plano de testes criar um plano de teste. É importante ainda ter em mente que criar um plano de testes muito elaborado com detalhes exatos não é necessário, mas sim ter uma compreensão clara de quais testes são, comunicando o escopo com a equipe de lançamento e salvando-o em uma área compartilhada.

Sendo assim, cria-se um plano mestre de testes que corresponde à estratégia em testes segundo o modelo ágil, contendo objetivos (Nader-Rezvani, 2018) como:

- fornecer uma estrutura para testes dentro do desenvolvimento do ciclo de vida para que o esforço permaneça focado e dentro do cronograma;

- identificar as tarefas necessárias para preparar e conduzir testes manuais e automatizados de nível de lançamento entre as equipes;
- renunciar a uma abordagem tradicional de testes em silos pela comunicação clara e frequente de coordenação com todos os membros da equipe de lançamento ágil;
- incluir nos itens a serem comunicados quais equipes adicionaram histórias específicas para o *backlog* do produto e, posteriormente, na lista de pendências de iteração. Isso inclui a coordenação de atividades de teste para evitar a duplicação de esforços e reduzir a oportunidade de ocorrência de erros;
- garantir o compartilhamento de dados de teste em todas as camadas de teste;
- representar requisitos funcionais e não funcionais do produto/componente e garantir que esses requisitos foram atendidos;
- indicar claramente por meio das histórias derivadas, se o projeto tiver um componente de terceiros ou de código aberto, quem realiza testes funcionais, de estresse e de sistema e especificar os critérios de aceitação para o componente.

Com o desenvolvimento da estratégia de testes, é necessário considerar algumas questões importantes (Nader-Rezvani, 2018):

- expectativas de marcos, por exemplo, pode haver requisitos específicos para alcançar grandes marcos, como demonstração do cliente, certificação de plataforma, infraestrutura, requisitos etc. que precisam ser considerados;
- garantir que os objetivos de incremento/liberação do programa sejam claramente compreendidos e levados em consideração ao criar uma estratégia de teste holística;
- usar uma abordagem de desenvolvimento orientado a testes (TDD) com forte ênfase na automação desde os estágios iniciais do lançamento;
- determinar as metodologias de teste que serão mais eficazes em encontrar defeitos críticos com antecedência e eficiência;
- realizar testes de API ou testes de linha de comando para testar o código no início do ciclo em vez de confiar apenas na “caixa preta” teste.

Considerar a melhor forma de alavancar e integrar ferramentas estáticas de análise de código para detecções precoces;

- as abordagens podem incluir testes baseados em risco, baseados em mudanças testes, testes baseados em requisitos, testes baseados em cenários, teste baseado em história, teste baseado em modelo, teste baseado em ataque testes, injeção de falhas, testes exploratórios, e assim por diante;
- *expertise* e as ferramentas disponíveis para a equipe para testes;
- como abordaremos o teste de regressão. Um produto novo em folha pode ser diferente de um produto existente com um grande número de testes automatizados disponíveis. Certa funcionalidade em produtos lançados pode ser comprovada em campo, e se nada foi alterado, o risco de regressões pode ser mínimo;
- considerar fortemente testes baseados em mudanças com um claro mapa para indicar onde o foco adicional é necessário;
- ter em mente que pode não haver tempo suficiente para executar todo o conjunto de testes de regressão;
- Para dependências, riscos e lacunas, ROAMing (Resolver, Possuir, Aceitar ou Mitigar), o exercício que identificará riscos e ações necessárias para incluir na estratégia de teste. Por exemplo, se um componente principal será preterido, é importante indicar claramente qual nível de teste de regressão será necessário para avaliar possíveis danos colaterais.

Vários tipos de teste são fornecidos como exemplo nas seções a seguir. É importante utilizar essas diretrizes de estratégia de teste, quando aplicável, e adicionar quaisquer outros tipos de teste que são adequados à estratégia.

Também é fundamental entender completamente os diferentes tipos de testes e suas várias camadas para aplicar a melhor estratégia de teste para as diferentes etapas de desenvolvimento de recursos. A fundação começa no nível de teste de unidade no inferior, e à medida que o código se torna mais estável, espera-se que camadas adicionais sejam projetadas e executadas.

Uma combinação de testes automatizados contínuos, misturando técnicas de teste, integrando a cobertura de código e utilizando análises para se concentrar em regressão nos dará uma estratégia de teste sólida. Sem uma estratégia de automação de teste adequada, um modelo de teste contínuo não

será eficaz ou mesmo possível. Também é importante considerar a integração da análise de código estático no modelo de teste para que a detecção precoce seja possível.

1.1 Cobertura de testes em camadas

É importante que as equipes reconheçam que é fisicamente impossível profissionais de teste validarem cada configuração e cenário. Definir estratégias, como o nível apropriado de testes em cada estágio de desenvolvimento, é fundamental para garantir que a cobertura adequada seja alcançada. O risco de não cobrir certas combinações de teste terá que ser discutido e internalizado pelas equipes de produto.

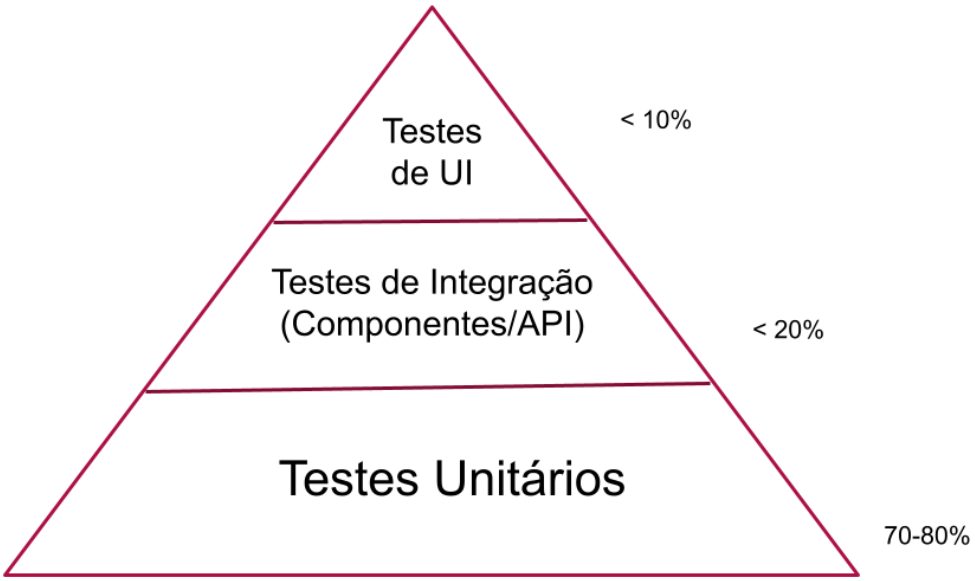
Todos nós vimos várias variações da Pirâmide de Teste que foi introduzida por Martin Fowler e Michael Cohn. Um exemplo está refletido na Figura 1, a seguir.

A principal conclusão é que as equipes devem se concentrar em produzir muito mais testes de unidade de baixo nível do que testes automatizados e manuais baseados em interface de usuário de alto nível, incluindo testes exploratórios e de usabilidade. Essa estrutura de camada de teste é usada para integração contínua e entrega contínua, que são essenciais para desbloquear os benefícios do teste ágil.

Na realidade, porém, muitas organizações têm uma estrutura de teste automatizada semelhante ao mostrado na Figura 1, em que a maior parte do foco está no manual e testes automatizados de interface do usuário. Isso é conhecido como o *Ice Cream Cone of Testing*.

Isso não possibilita um modelo de CI/CD bem-sucedido, que é um pré-requisito para ter um modelo de teste contínuo com boa cobertura. Essas equipes não investem tempo no desenvolvimento de testes unitários, porque não é “fácil” fazê-lo. O que eles não percebem é que a implementação sólida de cobertura de teste de unidade automatizada economiza muito dinheiro e tempo por encontrar problemas no início do ciclo de desenvolvimento.

Figura 1 – Pirâmide de testes



Fonte: Elaborado por Teixeira, 2022 com base em Nader-Rezvani, 2018.

Figura 2 – Cobertura de testes em camada

Fonte: Elaborado por Teixeira, 2022 com base em Nader-Rezvani, 2018.

TEMA 2 – TIPOS DE TESTES ÁGEIS

2.1 Teste de unidade

Testes de unidade para todos os novos códigos precisam fazer parte da Definição de Pronto, e como tais, são obrigatórios. As ferramentas de cobertura de código validarão a eficácia.

2.2 Teste de recurso/funcional

Essa camada de teste se concentra nos recursos de um módulo, componente ou nível do produto. Ele garante que o teste de regressão impactado seja incorporado ao teste de novos recursos.

2.3 Teste de regressão

Esse nível se concentra em garantir que novas funcionalidades/recursos não introduzam *bugs* ou impactem negativamente a funcionalidade do produto existente. Consideremos usar testes automatizados contínuos focados nas áreas de mudança, como novos funcionalidades são implementadas. Orientação para áreas de teste de regressão geralmente vem do arquiteto de QA, com a contribuição de toda a equipe *Scrum*. **Estratégia de regressão** impactada para garantir que novos recursos não violem os existentes funcionalidades devem ser discutidas em detalhes.

2.4 Teste de documentação

Testes serão feitos para verificar se a documentação do produto visível ao usuário é suficiente para instalar, administrar e usar o produto. Cada equipe *Scrum* tem a responsabilidade de revisar a documentação quanto à integridade e precisão, conforme nova funcionalidade é desenvolvida.

2.5 Teste do sistema

Este nível se concentra em testar o sistema integrado de módulos ou componentes dentro de uma área de produto para verificar se produz os resultados desejados. É importante descrever a abordagem de teste do sistema para as áreas no escopo e garantir abordagem ao longo do lançamento.

2.6 Teste de integração

Esse nível se concentra em testar o sistema integrado de produtos ou soluções (incluindo *software* de terceiros) para verificar se os produtos integrados atendem aos requisitos desejados. Ele fornece uma abordagem de teste para garantir se o produto fornece mecanismos uniformes para integração com produtos. Revisão da metodologia de teste de integração com toda a versão A equipe fornecerá uma abordagem de teste consistente em várias equipes de *Scrum*.

2.7 Teste do Cenário do Cliente E2E

Também conhecido como *Business Process Testing* ou *Model-Based Testing*, deve incluir clientes externos e internos.

2.7.1 Externo

É importante estabelecer um programa de validação do cliente para envolver pelo menos três clientes durante todo o ciclo de lançamento, quando novos recursos são disponibilizados para buscar *feedback* cedo e frequentemente. Isso possibilitará que os clientes executem seu E2E casos de uso e destaquem desafios ou forneçam opções para aprimorar os recursos. Algumas equipes construíram parcerias com clientes específicos e têm acesso aos detalhes do ambiente e conjunto de dados para executar casos de uso genéricos. Essa é uma ótima maneira de testar novos recursos, garantir que não haja regressões e receber *feedback* antecipado.

2.7.2 Interno

Testes exploratórios por clientes internos (ou seja, Suporte, Pré-vendas, *Services* e SWAT) é uma ótima abordagem a seguir aqui.

2.8 Testes não funcionais (habilidades)

Testes que determinam até que ponto o aplicativo atende aos requisitos não funcionais esperados são muitas vezes referidos como “habilidades”. A seguir estão os tipos específicos de “habilidades” que as equipes devem revisar e considerar conforme os requisitos para sua aplicação.

2.8.1 Teste de acessibilidade

Testes que serão feitos para garantir que o produto/componente esteja em conformidade com as normas de acessibilidade. O teste será concluído com o envio de um relatório de Conformidade.

2.8.2 Teste de interoperabilidade

Testes para garantir que o produto é capaz de coexistir de forma graciosa com componentes e agentes comuns que provavelmente estão na mesma lógica servidor em ambientes comuns de clientes. Isso inclui a capacidade de ser instalado e desinstalado sem afetar outros componentes, bem como ser capaz de compartilhar os recursos disponíveis de forma eficiente.

2.8.3 Teste de segurança

Existem vários tipos de metodologias de teste de segurança a serem usados. Análise de código estático de vulnerabilidade e segurança, testes de penetração, ética *hacking* e avaliação de risco estão entre as principais técnicas usadas pela maioria das empresas empresariais. Isso é para garantir que o produto/componente passe por análise adequada de segurança e risco para documentar vulnerabilidades e formular um plano de remediação adequado. O teste de segurança não é mais considerado uma reflexão tardia e deve ser integrado ao ciclo de desenvolvimento de código. As vulnerabilidades de segurança devem ser priorizadas e abordadas ao longo do caminho. A maneira como poderíamos descrever a importância de lidar com vulnerabilidades durante todo o ciclo de vida da versão é como se tivéssemos, por exemplo, usado blocos de construção para arquitetar uma pirâmide. Se, depois de completar a pirâmide, percebermos que um bloco incorreto foi usado na base e tentemos substituí-lo, toda a estrutura será impactada e possivelmente destruída. Dessa maneira, identificar problemas e resolvê-los à medida que eles surgem economizará dinheiro.

2.8.4 Teste de localização

A “localizabilidade” consiste em internacionalização (i18n) e localização (l10n) teste. Depois que o conteúdo é traduzido, é importante fazer a verificação

do conteúdo para garantir que o material seja traduzido adequadamente de acordo com o idioma e a cultura.

2.8.5 Teste de prontidão para internacionalização

Testes que verificam se o componente/recurso e, em geral, o produto como um todo, está devidamente internacionalizado para que possa ser localizado de forma rápida e econômica em qualquer idioma. O teste deve incluir testes de pseudolocalização que pretendemos realizar para garantir que o produto atenda aos requisitos de i18n.2.

2.8.6 Teste de localização

Testes que verificam se o material traduzido está linguisticamente correto e apropriado e que o produto/componente localizado pode ser instalado e parece e se comportar conforme o esperado em ambientes localizados e em inglês dos EUA (sistema operacional, navegador, banco de dados, terceiros e tecnologias subjacentes). Embora o foco esteja principalmente na interface do usuário da *web*, é importante também planejar tradução da documentação do produto, ajuda *on-line*, interface de linha de comando, arquivos de *log*, e assim por diante.

2.8.7 Teste de suporte

Testes que garantem que seu produto implemente um conjunto conhecido e consistente de capacidades para possibilitar que os clientes consumam produtos de forma eficaz em todo o ciclo de vida da implantação. Por exemplo, fornecer erro/aviso significativo por mensagens permitirão que os clientes e o departamento de suporte resolvam problemas mais rápido. Uma grande consideração de suporte é incorporar a telemetria ou o chamado Recurso de casa. Isso possibilita um diagnóstico mais rápido, bem como muitas vezes leva a melhorias e resolve os problemas dos clientes antes que eles o percebam – uma ótima maneira de melhorar a satisfação do cliente.

2.8.8 Teste de capacidade de atualização

Testes que garantem que o produto possa ser atualizado para a nova versão sem um grande investimento de esforço/*hardware* por parte do cliente. Os produtos básicos e as personalizações com suporte devem poder ser

atualizados dentro de prazos predefinidos (os quais precisam ser discutidos e acordados durante o planejamento da liberação).

2.8.9 Teste de usabilidade

Teste do aplicativo para garantir que os usuários pretendidos de um sistema possam realizar suas tarefas de forma eficiente, eficaz e satisfatória. Teste de usabilidade é realizado no pré-lançamento para que quaisquer problemas significativos sejam identificados, como também para validar o aplicativo em um limite de iteração lógica, para novos recursos.

2.8.10 Testes de desempenho e escalabilidade

Testes para garantir que o comportamento do sistema seja o desejado durante a carga normal e determinar o limite superior de transações ou cálculos permitidos do produto. Isso inclui as abordagens de teste de escalabilidade para vários níveis de teste do produto. Por exemplo, testes que verificam o comportamento do programa durante picos intensos de atividade, ou imprimindo 1.000 documentos de uma vez ou abrindo o número máximo de arquivos. É importante, no entanto, estabelecer limites adequados e aceitáveis com antecedência. Com certas aplicações e sem quaisquer desses limites, o sistema eventualmente falha. É melhor relaxar os limites como os casos de uso do cliente são mais bem compreendidos, se necessário.

2.9 Evolução de testes ao longo do tempo

Enquanto o conjunto anterior de diretrizes, tipos de testes, planejamento, e assim por diante, será relevante por muitos anos, os departamentos de desenvolvimento devem se manter sempre atualizados com as novas tecnologias emergentes para atualizar continuamente e ajustar sua estratégia de teste. Nos últimos anos, as tendências mencionadas a seguir e tecnologias ganharam muita publicidade. Manter a atenção nessas áreas específicas pode mudar a paisagem da sua estratégia de teste. Alguns começaram a influenciar o espaço mais do que outros.

2.9.1 Internet das Coisas (IoT)

Por melhor que pareça, o mundo dos dispositivos conectados e a integração pesadelo que vem com ele trará desafios interessantes para lidar. Como exemplo, vulnerabilidades resultantes de produtos conectados criarão muitas dimensões adicionais a serem consideradas em sua estratégia de teste.

2.9.2 *Big Data*

Clientes e usuários coletam e carregam *terabytes* de dados em várias plataformas. A consideração cuidadosa para testes com uma grande quantidade de dados precisa ser feita para um melhor alinhamento com o ambiente semelhante ao do cliente. Devemos nos lembrar de que nem sempre é possível construir um ambiente de cliente exato para validação interna, mas chegar ao mais próximo possível da configuração do cliente é o objetivo aqui.

2.9.3 Usuários móveis e automação de testes

Cada vez mais clientes exigem suporte móvel para seus aplicativos. Testando aplicativos móveis e considerando ângulos únicos em relação ao número de atualizações regulares de *software* e tipos de dispositivos, eles merecem suas próprias estratégias e planejamento de testes.

2.9.4 API e microsserviço

Microsserviços e ênfase em um *design* rico baseado em API se encaixam bem com o Agile, disciplina e metodologia. Existem muitas diferenças em relação ao teste tradicional de todo o sistema em uma abordagem em cascata. Receber pedaços e peças disponíveis para teste exigirão melhor planejamento, coordenação e otimização do ambiente utilizado. Cada serviço independente terá de ser testado separadamente e, em seguida, como parte de uma estrutura interconectada. A melhor abordagem aqui é voltar ao básico da Pirâmide de Teste e começar com uma forte estratégia de teste de unidade seguida por testes funcionais, de integração e E2E, que já falamos.

2.9.5 Adoção de ferramentas de código aberto

Ferramentas de código aberto têm sido populares em muitos setores para testar seus formulários. Os dias em que uma ferramenta era usada para testar todo o aplicativo se foram. Com a otimização que determina qual ferramenta de código aberto usar para qual camada de teste, podemos reduzir custos e melhorar a produtividade.

2.9.6 IA e aprendizado de máquina

Muitas organizações conseguiram otimizar seus testes e cobertura em todos os níveis, aproveitando várias fontes de dados que estão disponíveis para eles. Tais dados residem no gerenciamento de casos de teste e nos relatórios de defeitos existentes das equipes sistemas (registro, detalhes de resolução e informações de regressão), juntamente com seus dados do repositório de código-fonte. A utilização desses dados pode ajudar a identificar mapa de calor e áreas problemáticas no produto. Os dados históricos são um tesouro que pode ser usado para treinar algoritmos de IA para medir riscos, classificar defeitos e prever entrega de soluções aos clientes.

2.9.7 Teste de *Crowdsource*

Isso está se tornando cada vez mais popular, especialmente com orçamento em constante pressão. Não ser capaz de contratar recursos internos ou contratados para fazer testes exploratórios levou algumas organizações a adotar essa abordagem. O maior desafio aqui são as preocupações de segurança que existem e precisam ser tratadas.

TEMA 3 – TESTES ÁGEIS SEGUNDO O MANIFESTO DE TESTES ÁGEIS

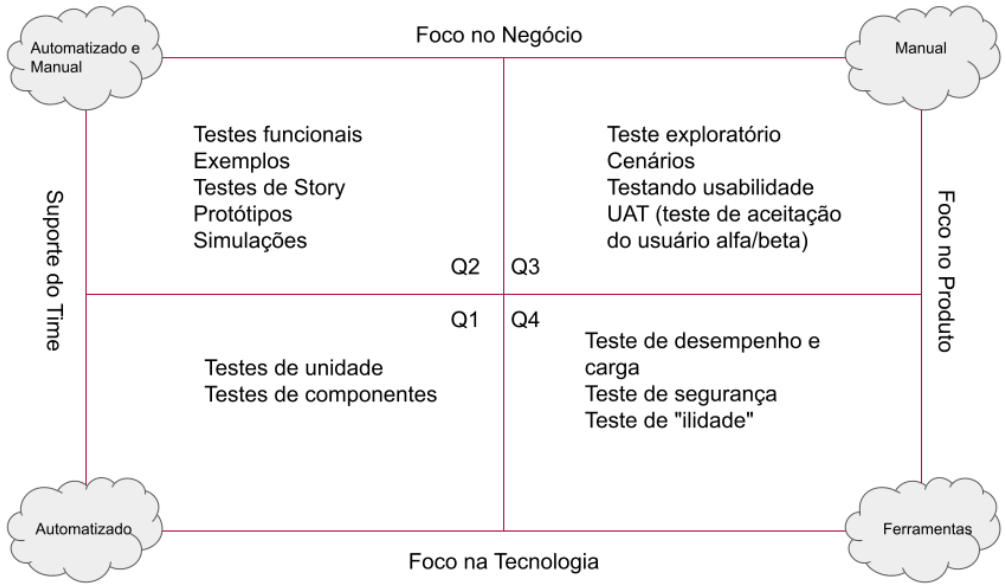
Testes ágeis surgiram em complemento aos métodos ágeis, pois o trabalho da área de SQA, da mesma forma que a de desenvolvimento de *software*, também precisa evoluir e conquistar agilidade com qualidade. Os pontos principais do manifesto de testes ágeis estão relacionados a seguir.

- Testar cada etapa e não somente no término do projeto.
- Prevenir *bugs* e não encontrar *bugs*.
- Testar o entendimento e não verificar as funcionalidades.

- Construir o melhor *software* e não deixá-lo quebrar.
- O time é responsável pela qualidade, não somente o time de SQA.

Na Figura 3, a seguir, são demonstrados os quadrantes dos testes ágeis, que dão uma ideia de como os testes acontecem em conjunto com o ciclo de desenvolvimento de *software*.

Figura 3 – Quadrantes de testes ágeis



Fonte: Teixeira, 2022.

3.1 Quadrante 1

O quadrante 1 encontra-se no canto esquerdo inferior e representa o desenvolvimento orientado a testes, que é uma prática de desenvolvimento ágil. Os testes de unidade verificam a funcionalidade de um pequeno subconjunto do *software*, um objeto ou método. Testes de componentes verificam o comportamento de uma parte maior do *software*, como um grupo de classes que presta algum serviço. Ambos os tipos de testes geralmente são automatizados com um membro da família de testes xUnit, ferramentas de automação. Esses testes de programador são voltados para tecnologia e possibilitam que a qualidade interna do código seja garantida. Um *framework* ágil que aborda o uso de testes unitários como principal objetivo é o Desenvolvimento orientado a testes – TDD.

O processo de escrever testes antes da escrita do código principal oportuniza que os códigos sejam escritos com mais confiança e as entregas das

Stories fiquem bem alinhadas. Testes unitários e de componentes são automatizados e escritos na mesma linguagem de programação do projeto que está em curso. Um especialista de negócios provavelmente não é treinado para compreendê-lo, até porque esses testes não se destinam ao uso do cliente. Tais testes são parte do processo automatizado que é executado para o *check-in* do código, o qual dá à equipe um *feedback* instantâneo e contínuo sobre a qualidade interna.

3.2 Quadrante 2

Os testes do quadrante 2 também apoiam o trabalho do time de desenvolvimento, mas em um nível mais alto. Estes são voltados para os negócios e para o cliente, como também definem a qualidade externa e as características desejadas pelos clientes. Assim como os testes de Q1, segundo a Figura 3, estes impulsionam o desenvolvimento em um nível mais alto, pois descrevem em detalhes as especificações de cada Story voltado ao negócio. São escritos de forma que os especialistas em negócios entendam e tenham domínio sobre sua verificação. Por vezes, duplicam testes do nível de unidade, porém são orientados a confirmar o comportamento desejado no nível do negócio. Também podem ser automatizados com o propósito de fornecer informações rápidas e possibilitar resoluções ágeis dos problemas. São executados com frequência para que o time seja alimentado antecipadamente com comportamentos inesperados. Os testes verificam as UIs do usuário e APIs dos aplicativos que serão utilizados pelos clientes. Tais testes devem ser executados como parte do processo contínuo automatizado.

Há outro grupo de testes que também pertence a esse quadrante no qual o usuário e especialistas em interação usam *wireframes* para ajudar na validação das propostas de projetos de GUI (interface gráfica do usuário). Com isso, *designers* auxiliam os desenvolvedores *front-end* antes que eles comecem a codificá-los. Os testes com esse grupo ajudam a apoiar a equipe para que o produto seja construído corretamente, mas não são automatizados. O quadrante 2 também corresponde ao termo “testes de aceitação”, que abrange uma gama mais ampla de testes em conjunto com os quadrantes 3 e 4. Esses testes de aceitação verificam se todos os aspectos do sistema atendem de fato aos requisitos do cliente, incluindo testes de usabilidade e desempenho.

3.3 Quadrante 3

Os especialistas em negócios podem ignorar a funcionalidade ou o time pode simplesmente interpretar mal os requisitos. Mesmo quando os programadores desenvolvem o código que faz os testes focados nos negócios passarem, eles podem não estar totalmente completos ou no caminho adequado relativo ao negócio. O quadrante 3 classifica os testes voltados para os negócios. Quando fazemos testes voltados para os negócios para validar o *software*, testamos “simulando” a maneira como um usuário real trabalharia no aplicativo. Esse teste é manual, pois somente o ser humano pode fazê-lo.

Podemos usar alguns recursos automatizados para nos ajudar a configurar os dados de que precisamos, mas temos que usar nossos sentidos e nossa intuição para verificar se o time de desenvolvimento entregou o valor de negócio exigido pelos clientes. Muitas vezes, usuários e clientes realizam esses tipos de testes de aceitação. O teste (UAT) oferece aos clientes a chance de dar um bom treino aos novos recursos e ver quais mudanças eles podem querer no futuro, e é uma boa maneira de reunir novas ideias de *Stories*. Se o time está entregando *software* com base em contrato para um cliente, o UAT pode ser uma etapa necessária para aprovar as *Stories* concluídas. O teste de usabilidade é um exemplo de um tipo de teste que estuda o uso do aplicativo à medida que é utilizado por grupos focais, por exemplo. Grupos focais reúnem amostras de possíveis clientes que são entrevistados e observados para colher suas reações. O teste de usabilidade também inclui navegação de página em página ou mesmo algo tão simples como a ordem de tabulação. O teste exploratório é central para esse quadrante. Durante o teste exploratório das sessões, o testador projeta e executa testes simultaneamente, usando os resultados e pensando em analisá-los. Isso oferece uma oportunidade muito melhor para aprender sobre o aplicativo do que testes com *script*. O teste exploratório é mais abordagem ponderada e sofisticada do que testes *ad hoc* e trabalha o sistema da mesma forma que os usuários finais farão. Os testadores usam sua criatividade e intuição, e como resultado, é por meio desse tipo de teste que muitos dos mais problemas sérios são normalmente encontrados.

3.4 Quadrante 4

Os tipos de testes que se enquadram no quarto quadrante são igualmente críticos para a agilidade e para o desenvolvimento de *software*. Os testes voltados para a tecnologia no quadrante 4 destinam-se a criticar as características do produto, tais como desempenho, robustez e segurança. Criar e executar esses testes pode exigir o uso de ferramentas especializadas e conhecimentos adicionais. Se conhecermos os requisitos de desempenho, segurança, interação com outros sistemas e outros atributos não funcionais antes de começarmos a codificar, é mais fácil de projetar e codificar a aplicação. Alguns deles podem ser mais importantes do que a funcionalidade real. Esses tipos de testes geralmente são apoiados por ferramentas que automatizam o processo, configurando cenários de testes manuais, segurança, bem como de carga e desempenho.

Os quatro quadrantes servem como diretrizes para garantir que todas as situações possíveis sejam cobertas no processo de teste e desenvolvimento. Testes que dão suporte ao time podem ser usados para direcionar os requisitos. Testes que criticam o produto nos ajudam a pensar em todas as possibilidades de aplicação da qualidade. Os quadrantes são utilizados para sabermos quando o *software* estará pronto e como garantir que todo o time compartilhe a responsabilidade de cobrir os quatro quadrantes da matriz. Os quadrantes servem para pensarmos sobre as diferentes dimensões, bem como sobre o contexto para o os esforços de teste.

TEMA 4 – BDD, *CODE COVERAGE* E TESTES UNITÁRIOS

O BDD, que dentro do diagrama dos quadrantes de testes ágeis corresponde ao Q2, é uma abordagem de projeto que garante que o projeto de *software* tenha 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. Nessa metodologia, os testes são importantes, mas os testes não são os elementos que conduzem ao desenvolvimento. Seu objetivo é que metas e resultados para o cliente sejam definidos de forma clara.

Essa abordagem envolve pessoas no processo por meio de *Outside-in Development* (em que 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. Escrever o comportamento do *software*, para que as responsabilidades fiquem claras, possibilita que suas funcionalidades 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 apresentado na Figura 3 anterior. Logo após, utilizamos a estrutura GWT – *Given*, *When*, *Then*), conforme é mostrado na Figura 4 a seguir. Na estrutura GWT, definimos os cenários do negócio para criarmos os seus critérios de aceitação.

Figura 4 – *Features*

GWT - Given, When, Then	
FUNCIONALIDADE	Enviar um PIX
COMO	funcionalidade segundo normas do BACEN
EU QUERO	efetuar um pagamento instantâneo
PARA	outra pessoa física ou jurídica.

Fonte: Teixeira, 2022.

Figura 5 – Estrutura GWT

GWT - Given, When, Then	
CENÁRIO	Cliente envia PIX
DADO	que o cliente tem a chave de pix correta
QUANDO	entrar com a chave do pix
ENTÃO	o aplicativo deve validar a chave de pix na outra empresa financeira.

Fonte: Teixeira, 2022.

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.

4.1 *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 nesse 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 está sendo testado e 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 busquemos 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.

4.2 Testes unitários

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

Esse 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, bem como pensar sobre quais entradas e saídas queremos diante do processamento do fluxo dos dados. Normalmente conhecido como Unit, apresenta 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 por meio 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 têm 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 5 – TESTES AUTOMATIZADOS E TESTES DE VULNERABILIDADE

A arquitetura de microsserviços vem tomando um espaço considerável no âmbito da engenharia de *software*, e com ela 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 os torna mais econômicos, uma vez que é necessário o uso de testes duplicados para 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) geram 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. governança descentralizada, que 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, entre 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 microsserviços 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 tem 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 dentro 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 esse novo contexto.

Algumas ferramentas comerciais que seguem a ideia de observabilidade são chamadas de *Application Performance Management* – APM. Essa ferramenta apresenta 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*.

Entre 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 por meio de uma ferramenta de observabilidade como as citadas anteriormente auxiliam muito um trabalho que manualmente seria praticamente impossível.

5.1 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 chamada de Dynamic Application Security Testing – Dast, 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 tornem suscetível a ataques ou tentativas de *hackers*.

Os riscos que são apurados são classificados em uma 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 nesse 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 possibilitamos que eles sejam resolvidos antes que sejam atacados por *hackers* ou até mesmo por usuários mal-intencionados que descubram brechas ao acaso.

Lembrando que essas questões de vulnerabilidade vão de encontro com a Lei Geral de Proteção de Dados – LGPD, a Lei de Portabilidade e Responsabilidade de Seguros de Saúde – HIPAA), a 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, como 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 sim 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 causado por *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 com o tipo de *software* para garantirmos a sua qualidade e também a do processo de desenvolvimento.

Considerar técnicas como BDD, TDD, SBE e outras em paralelo com os métodos ágeis *Scrum*, XP e *Kanban*, por exemplo, podem 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.

REFERÊNCIAS

NADER-REZVANI, N. A**n Executive's Guide to Software Quality in an Agile Organization**: A Continuous Improvement Journey. Apress, 2018.