DISCIPLINA DE
TESTE DE SOFTWARE

Sumário

APRESENTAÇÃO

Olá, sou Eliane Collins, eu possuo mais de 13 anos de experiência na área de Teste de Software (Pesquisa, Gerenciamento, Execução de Processo de Teste e Automação de Teste). Eu também sou Doutoranda de Ciência da Computação na USP e trabalho na linha de pesquisa de teste de software e inteligência artificial.

Durante minha experiência profissional trabalhei com sistemas web, sistemas móveis, metodologias ágeis e também com pesquisa e inovação. Eu possuo certificação CTAL-TM – Certified Test Advanced Level (BSTQB/ISTQB), CSD – Certified Scrum Developer (Scrum Alliance), COBIT – Control Objectives for Information and Related Technology e ITIL – Information Technology Infrastructure Library.

Eu costumo participar de eventos acadêmicos e profissionais e já realizei publicações em eventos SBQS, CBSOFT, ICTSS, AST, ICGSE, COMPUSAC, Global Scrum Gathering, Revista Engenharia de Software Magazine – ESM (DEVMedia). Eu também contribuí em capítulos do livro: Introdução ao Teste de Software e Automação de Teste de Sofware da SBC com professores da USP, UFAM e UFSCAR.

O material a seguir foi elaborado ressaltando autores e trabalhos que são referências na área com exemplos para facilitar o aprendizado, assuntos adicionais foram incluídos no livro para complementar o conhecimento, em caso de dúvidas ou se você quiser entrar em contato comigo, eu estou disponível no endereço de e-mail elianecollins@gmail.com .

.

UNIDADE I
DEFINIÇÕES DE TESTE DE SOFTWARE

Objetivo

Este capítulo aborda uma rápida revisão pelos principais ciclos de vida de desenvolvimento de software para assim definir a importância de se testar software, os princípios de teste de software, os níveis de teste de software, os principais tipos e técnicas de teste que auxiliam no processo para garantir a qualidade de sistemas.

Ciclo de Vida de Software

Processo de software pode ser definido segundo Pressman (PRESSMAN, 2005) como uma coleção de padrões que definem um conjunto de atividades, ações, tarefas de trabalho, produtos de trabalho e/ou comportamentos relacionados necessários ao desenvolvimento de softwares de computador. Em termos gerais um processo para desenvolvimento de software nos fornece um guia com estágios, métodos e ferramentas consistentes para uma equipe de projeto construir um software.

Todo processo define uma estrutura que deve ser estabelecida para a efetiva utilização da tecnologia da engenharia de software, ele forma a base para o controle gerencial dos projetos de software e estabelecem o contexto no qual métodos técnicos são aplicados, os produtos de trabalho são produzidos, os marcos são estabelecidos, a qualidade é assegurada e a modificações são geridas de maneira adequada (PRESSMAN, 2005).

A partir dessa idéia vários processos de desenvolvimento de software foram criados ao longo do tempo, mas devido à complexidade da construção de sistemas, fatores ambientais e culturais que influenciam em equipes e em projetos de desenvolvimento nenhum pode ser considerado ideal, não

existindo, portanto, uma fórmula única que garanta o sucesso e a qualidade de todo software produzido.

Desenvolvimento em Cascata

A metodologia de desenvolvimento Cascata ou Modelo Clássico segue uma abordagem sistemática e sequencial que inicia como visto na Figura 1 com a especificação de requisitos, depois progredindo para o planejamento do projeto do sistema, a implementação culminando na manutenção do software acabado.

Figura 1 Fases do Modelo Sequencial

Sendo este o paradigma mais antigo da engenharia de software, nas últimas décadas várias críticas têm provocado questionamentos principalmente pelo fato de que projetos reais raramente seguem um fluxo sequencial, dificilmente o cliente estabelece todos os requisitos no início do projeto e também o fato da entrega ser feita ao cliente apenas no período final do projeto (PRESSMAN, 2005).

Modelo Rapid Application Development (RAD)

Essa metodologia é incremental com ênfase em um ciclo de desenvolvimento curto, sendo uma adaptação do Modelo Clássico onde o desenvolvimento rápido é conseguido através do desenvolvimento baseado em componentes. Sua aplicação se dá nos casos em que o sistema pode ser modularizado. Cada uma das equipes RAD de desenvolvimento fica responsável por um módulo, enquanto que outras equipes desenvolvem outros módulos, de forma concorrente (PRESSMAN, 2005). Ao final da construção de cada módulo há uma integração entre eles e este ciclo se repete até que o software esteja pronto como mostra na Figura 2.

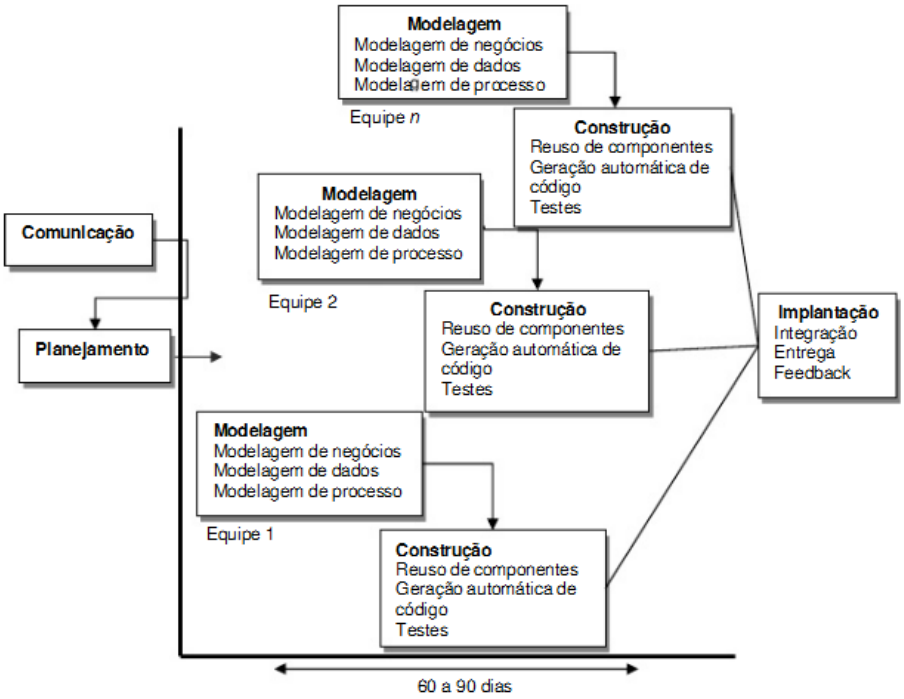


Figura 2 Modelo RAD de PRESSMAN, 2006

Essa metodologia não é adequada caso o sistema não possa ser modularizado, sem um projeto grande o suficiente para formar uma equipe RAD onde a equipe possui expertise para produzir com alto desempenho, se cliente e equipe de desenvolvimento não estiverem comprometidos com as atividades contínuas e rápidas e quando os riscos técnicos são muito grandes (PRESSMAN, 2005).

Prototipagem Evolutiva

Em um processo evolucionário mostrado na Figura 3, é feito primeiro um protótipo do sistema que é exposto para o cliente e refinado até que o sistema esteja de acordo com as necessidades desejadas. O protótipo é uma versão inicial usada para demonstrar conceitos, experimentar opções de projeto e, para saber mais sobre o problema e suas possíveis soluções. A partir do protótipo o sistema final é construído e testado para finalmente ser entregue (SOMMERVILLE 2006).

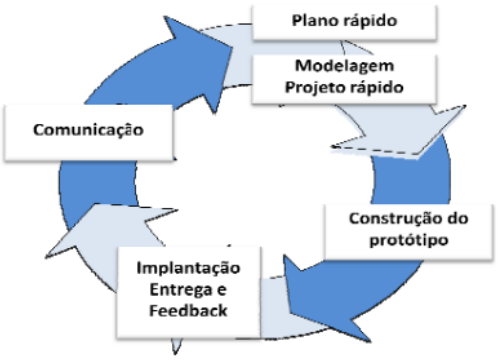


Figura 3 Prototipagem, de PRESSMAN 2005.

A Prototipagem pode ser problemática devido aos seguintes fatores: O Cliente cria expectativa de um produto pronto ou quase pronto ao visualizar o protótipo, exigindo apenas alguns acertos para que ele seja entregue, ignorando que o protótipo ainda não estaria funcional.Outro fator é que o desenvolvedor frequentemente faz concessões na implementação a fim de gerar um protótipo executável. Uma implementação inapropriada pode ser feita apenas para disponibilizar o protótipo rapidamente, com a pressão do tempo para entrga, nem toda codificação é retrabalhada e acaba fazendo parte do sistema definitivo (PRESSMAN, 2006).

Rational Unified Process (RUP)

A metodologia RUP foi desenvolvida e criada pela empresa IBM Rational e se caracteriza por exigir disciplina no desenvolvimento utilizando um conjunto de ferramentas, modelos e entregáveis. Esta metodologia é iterativa e incremental, guiada por casos de uso, análises de risco e possuindo ciclos de desenvolvimento sucessivos como iniciação, elaboração, construção e transição mostrados na Figura 4 (KRUCHTEN, 2000).

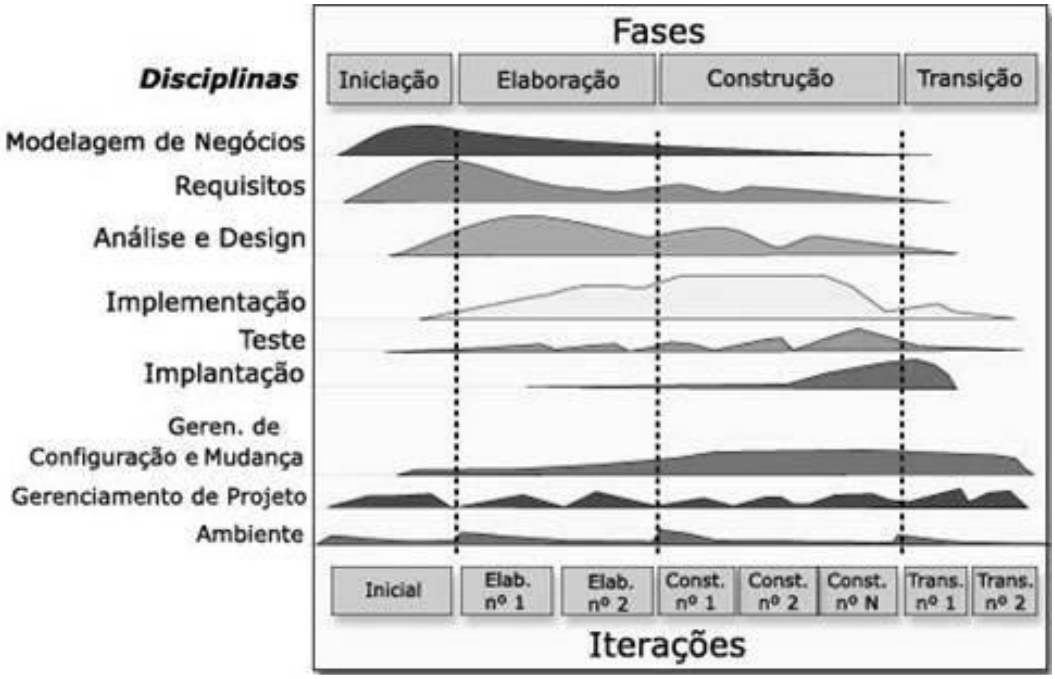


Figura 4 Ciclo de construção usando RUP de: Kruchten, 2000.

As equipes que adotam o RUP, devem ter em mente que os benefícios dessa metodologia não virão de maneira imediata, pois para sua implementação é necessário ter treinamento adequado, comprar ferramentas apropriadas, conseguir apoio especializado para as equipes de desenvolvimento e tempo para a absorção da metodologia. Sendo um investimento justificável apenas para projetos de grande porte e complexos.

Metodologia Ágil de Desenvolvimento Scrum

É uma metodologia de desenvolvimento ágil que se tornou muito popular na indústria. O Scrum é descrito como um processo de gestão de software e desenvolvimento de produtos, que segundo o autor, utiliza iterações e práticas incrementais, para produzir produtos que agregam valor ao negócio, podendo ser também aplicada a projetos de outras naturezas, bem como gestão de programas.

Os papéis no Scrum são (BERTHOLDO e BARBAN, 2010):

Scrum Master: O responsável por garantir o entendimento do processo e sua execução, além de motivar e treinar a equipe.

Product Owner: É o responsável pela priorização de requisitos que devem ser implementados, ou seja, gerencia o Product Backlog (lista de funcionalidades que serão implementadas no projeto).

Time (Equipe): Responsável pela execução e implementação das funcionalidades.

A figura 5 apresenta o ciclo de vida de projetos sob a metodologia Scrum, a metodologia controla as funcionalidades que deverão ser implementadas em forma de estórias (User Story), através de uma lista chamada Product BackLog. Para cada iteração de desenvolvimento (Sprint), é feita uma reunião inicial de planejamento (Sprint Planning Meeting), onde itens desta lista são priorizados pelo cliente (Product Owner). A equipe Scrum então define quas funcionalidades poderão ser atendidas dentro da iteração de acordo com a prioridade. Essa lista planejada para a iteração é chamada de Sprint Backlog (TAVARES 2008).

Figura 5 Ciclo de Vida no SCRUM (Collins E., 2013)

Durante o Sprint as atividades definidas são divididas em tarefas que são acompanhadas pela equipe através da reunião diária (Daily meeting), que dura no máximo quinze minutos, onde problemas e impedimentos para a execução são identificados e resolvidos através da resposta de cada membro do time a 3 perguntas (O que foi feito desde a última daily Scrum? O que se espera fazer até a próxima daily Scrum? O que está impedindo / atrapalhando o progresso?).

Por que testar?

Softwares estão no nosso dia a dia seja para realizar uma compra, uma operação bancária, um agendamento de consulta, nos aparelhos médicos e

para comunicação. Neste sentido, o desenvolvimento desses softwares deve garantir que funcionem corretamente para evitar falhas que causem prejuízos financeiros e de vida.

As Atividades que garantam a qualidade de um software a ser desenvolvido são essenciais para sua entrega ao cliente. Para Pressman (PRESSMAN, 2005) a Qualidade de Software é estar em conformidade com os requisitos, tanto funcionais quanto não funcionais, tendo as características implícitas e explícitas do sistema atendidas. Segundo o autor, um conjunto de fatores compõem a Qualidade de Software e estes variam de acordo com o contexto no qual se está trabalhando. Para um sistema on-line, o tempo de processamento da informação é requisito fundamental e pode afetar a qualidade do sistema, enquanto um sistema de gestão pode não ter o tempo de processamento como um fator essencial que irá afetar sua qualidade diretamente.

As atividades de garantia de qualidade que fornecem uma métrica quantitativa do projeto de software são as atividades de V&V (Verificação e Validação). São importantes e efetivas quando aplicadas desde o início do projeto para (SOMMERVILLE, 2006) elas são definidas como:

- Verificação: avalia se o software desenvolvido está em conformidade com os padrões previamente estabelecidos e está sendo construído corretamente, a verificação pode ser feita em documentação e código através de técnicas de revisão e inspeção.

- Validação: avalia se o software desenvolvido está em conformidade com os requisitos do cliente. O teste faz parte dessa atividade e examina se o comportamento do software está correto, através de sua execução.

Mas por que ocorrem defeitos no software?

O ser humano está sujeito a cometer um engano, que produz um defeito (bug), no código, em um software ou sistema ou em um documento, e que é demonstrado através de uma falha para o usuário.”

Figura 6 Principais definições engano, defeito e falha

Entre as principais causas dos defeitos de software, destacam-se:

- Pressão no prazo
- Códigos complexos
- Complexidade na infraestrutura
- Mudanças na tecnologia e de requisitos não documentados
- Muitas interações de sistema
- Processo de desenvolvimento ainda imaturo
- Falhas de comunicação
- Pessoas com pouco ou nenhum treinamento em sua função

"Testar é analisar um programa com a intenção de descobrir erros e defeitos." (Myers)

O Teste de Software é considerado o elemento crítico da garantia de qualidade de software e deve ser conduzido por uma equipe de profissionais especializados. É uma atividade que pode ser encarada, do ponto de vista psicológico, como uma anomalia do processo de desenvolvimento de software, pois o engenheiro deve criar testes com o objetivo de "demolir" o software que ele construiu (PRESSMAN, 2005).

Testar o software é diferente de "debugar" o código. Pois o primeiro é responsável por mostrar Falhas e o segundo identifica a causa de um defeito e é realizado pelo desenvolvedor.

O Teste de software possui 7 princípios gerais, que são:

1. Teste demonstra a presença de defeitos

- O Teste demonstra a presença de erros, mas não pode provar que eles não existam. Teste reduz a probabilidade que os defeitos permaneçam em um software. Se seus testes não encontram defeitos, reveja seu processo de teste.

2. Teste Exaustivo é impossível

    - Testar todas as combinações de entradas e pré-condições não é viável, exceto para sistemas muito triviais

3. Teste Antecipado

    - A Atividade de teste deve começar no início do projeto de software. “Quanto mais cedo os defeitos forem encontrados e corrigidos, menor é seu custo para o projeto” (Myers, 1979)

4. Agrupamento de defeitos

    - Um número pequeno de módulos contém a maioria dos defeitos descobertos durante o teste. 80% dos defeitos são causados por 20% do código (Lei de Pareto).

5. Paradoxo do Pesticida

    - Após um determinado momento, um mesmo conjunto de testes executados podem não encontrar mais falhas. Por isso, casos de teste devem ser constantemente revisados e atualizados.

6. Teste depende do contexto

    - Diferentes tipos de aplicações exigem aplicação de técnicas diferentes de teste. Um sistema de controle de tráfego aéreo possui um nível de risco diferente de um software de biblioteca.

7. A ilusão da ausência de Erros

    - Os sistemas devem fazer o que o usuário deseja. Sistemas “sem bugs” que não satisfazem às expectativas e necessidades dos usuários não serve.

Você deve estar se perguntando quanto de Teste deve ser necessário?

A resposta é depende. É necessário levantar alguns pontos em consideração:

- Nível do risco (risco técnico, do negócio e do projeto)
- Restrições do projeto como tempo e orçamento
- Completude de requisitos e funcionalidades
- Satisfação do cliente com o sistema

Níveis de Teste

Devido à necessidade de se testar o software sob vários prismas, para que assim seja possível identificar o máximo de erros possíveis dando assim maior cobertura e confiabilidade ao software, as atividades de teste devem passar por fases, são elas (MALDONADO, 2007):

Teste de Unidade:

O teste de unidade consiste, em testar as menores unidades de um programa, como funções, métodos e sub-rotinas. Cada unidade deve ser testada separadamente e pode ser aplicado pelo próprio desenvolvedor à medida que ocorre a implementação (MALDONADO, 2007). Um teste de software de unidade deve verificar (INTHURN C. 2001):

- A interface com o módulo. As informações de entrada e de saída devem ser consistentes;
- A estrutura de dados local, ou seja, os dados armazenados temporariamente devem manter a sua integridade durante os passos de geração do código;
- As condições de limite. Os limites que foram estabelecidos na demarcação ou restrição do processamento têm que garantir a execução correta da unidade;
- Os caminhos básicos. Instruções devem ser executadas para verificar se os resultados obtidos estão corretos.
- Os caminhos de tratamento de erros. Estes também devem ser testados para validar se valores não verdadeiros estão sendo corretamente tratados.

Como, apenas um módulo do sistema não é um programa, principalmente no desenvolvimento orientado a objetos, são desenvolvidos para essa fase Stubs e Mocks. Stubs são classes criadas para substituir um código e simular seu comportamento, se preocupando em testar o estado dos objetos. Já Mocks, simulam o comportamento de objetos de forma controlada para verificar a interação entre eles (FREEMAN et al. 2004).

Teste de Integração:

É o teste que verifica se os módulos avaliados individualmente e aprovados, funcionam corretamente quando juntos, ou seja, integrados. Existe a integração de maneira incremental, onde as unidades são gradativamente integradas e testadas. Há a integração não incremental que seria a combinação das unidades e o sistema completo é testado. Já a integração incremental é dita como a mais eficiente devido à facilidade de isolar as causas de defeitos quando se testa pequenas partes, do que ao se testar o sistema inteiro (INTHURN C., 2001).

Nesse caso é necessário o conhecimento das estruturas dos módulos e interações no sistema, por conta disso, essa fase de teste tende a ser executada pelos desenvolvedores do sistema (MALDONADO, 2007).

Existem as seguintes abordagens:

- Top-Down: Módulos integrados de cima para baixo.
    - Drivers são utilizados como modulo de controle e os módulos reais são substituídos por stubs.
    - Permite a verificação antecipada de todo o comportamento de alto nível.
    - Retarda a verificação das implementações de baixo nível.
    - Entradas e saídas podem ser difíceis de formular e analisar.
- Bottom-up: Cada modulo no nível inferior da hierarquia do sistema é testado individualmente.

    - A cada integração drivers são utilizados para a integração de novos módulos e substituídos pelos módulos correspondentes.
    - Stubs geralmente não são necessários.
    - Mais fáceis de formular e de interpretar os dados de saída.
    - Muitos elementos são integrados e combinados de uma só vez.
- Big Bang: Todos os components são combinados com antecedência
    - O sistema inteiro é testado de uma vez
    - Para cada modulo, constrói-se stubs e drivers
    - Correção difícil por conta da dificuldade de isolar a causa do erro

Os Stubs são unidade que substitui a unidade final que será testada. Simulam um comportamento da classe. Os Drivers são operações que automatizam testes de acordo com caso de teste. É responsável pela ativação do teste de uma unidade.

Teste de Sistema:

É o teste feito após a integração do sistema, ele explora as funcionalidades como um todo, o seu objetivo é verificar se o software e os demais componentes, como hardware e banco de dados, funcionam corretamente juntos de acordo com os requisitos do cliente e com o desempenho satisfatório.

Esse tipo de teste verifica várias características funcionais e não funcionais do sistema.

Testes ditos Funcionais podem vir de: especificação de requisitos, casos de uso, especificação funcional, ou estórias e fluxo de usuário. Alguns tipos:

- Teste de Segurança: avalia se o sistema protege dados de usuário contra ameaças e invasões.
- Teste de Recuperação: avalia a capacidade do sistema de se recuperar de falhas externas do ambiente e internas.

- Teste de Regressão: A cada mudança de versão do software, todos os casos de testes funcionais elaborados devem ser executados novamente para garantir que o sistema continua funcionando corretamente.

- Os Testes não funcionais validam o comportamento do sistema e como ele está funcionando. Alguns tipos:

- Teste de Desempenho: avalia se o tempo de resposta do sistema a uma solicitação do usuário está adequado.

- Teste de carga, stress: avalia como o sistema se comporta em uma situação extrema de grande quantidade de solicitações e operações ao mesmo tempo.

- Teste de usabilidade: avalia o quanto a interface e aparência do sistema está amigável, fácil e adequada ao usuário.

- Teste de interoperabilidade: avalia como o sistema se comporta e interage em um ambiente com outras aplicações e sistemas interagindo com o usuário ao mesmo tempo.

- Teste de manutenção: avalia a capacidade do sistema de ser modificado e atualizado.

- Teste de confiabilidade: avalia a capacidade do sistema de se manter estável por um longo período de execução.

Teste de Aceitação

Testes de aceitação, são testes funcionais realizados antes da disponibilização do sistema para o cliente onde a preocupação é de validar as regras de negócio e os requisitos originais especificados, normalmente são realizados por um grupo de usuários e no ambiente mais próximo possível dos usuários do sistema (BLACK, 2008). Podem ser também Operacional, Alfa e beta.

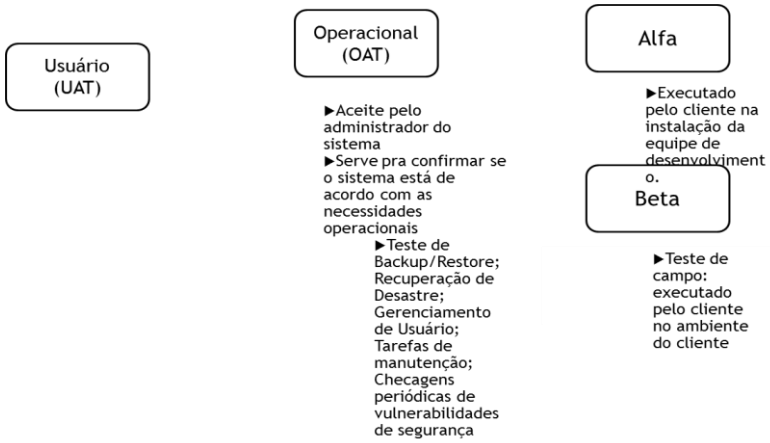


Figura 7 Definições de tipos de Teste de Aceitação (Collins E., 2013)

Técnicas de Teste

Técnica Estrutural ou Caixa Branca

A técnica de teste estrutural, também chamada de teste de caixa branca valida o software em nível de implementação, testa os caminhos lógicos do software, condições, laços e uso de variáveis.

Existem basicamente 3 níveis a serem aplicados:

- Nível de Componente: A estrutura é o próprio código, ex.: comandos, decisões e desvios.
- Nível de Integração: A estrutura pode ser uma árvore de chamadas (um diagrama em que um módulo chama outros módulos).
- Nível de Sistema: A estrutura pode ser uma estrutura de menu, processos de negócios ou estruturas das páginas Web.

Há vários critérios utilizados para essa técnica, são classificados em (MALDONADO, 2007):

- Critérios baseados na complexidade: Utilizam informações sobre a complexidade do programa para derivar requisitos de teste como o critério Caminho básico que utiliza a complexidade ciclomática, essa métrica oferece uma medida quantitativa da dificuldade de conduzir os testes e uma indicação de confiabilidade final.
- Critérios baseados em fluxo de controle: utilizam a análise de fluxo de dados como fonte de informação para derivar os requisitos de

teste. Requerem testes nas interações que envolvam definições de variáveis e subsequentes de referências a essas definições como os critérios Todas-Arestas e Todos-Caminhos que visam um teste estrutural rigoroso.

- Critérios baseados em fluxo de dados: Destacam-se critérios Rapps e Weyuker, derivam requisitos de teste usando os conceitos de Grafo Def-Uso que é extensão do grafo de fluxo de controle. Nele são adicionadas informações a respeito do fluxo de dados do programa, caracterizando associações entre pontos do programa nos quais é atribuído um valor a uma variável e pontos onde esse valor é utilizado.

- Cobertura de Sentença: Seu objetivo é executar todos os comandos (linhas executáveis) não importando se todos os desvios dos códigos foram exercitados. Para realizar seu cálculo, utilizamos a fórmula:

$$\text{cobertura de sentença} = \frac{\text{nº de comandos exercitados}}{\text{total de comandos}} * 100\%$$

- Cobertura de Decisão (cobertura de desvios): É avaliada pela porcentagem dos resultados da decisão que foram exercitados em um conjunto de casos de teste. Por exemplo, as opções de "Verdadeiro" ou "Falso" de uma expressão condicional - IF. Exemplo de fluxos de controle que possuem desvios:

    - IF, ELSE, DO WHILE, CASE, UNTIL, REPEAR, etc

    - Objetivo: testar todos os resultados possíveis de todas as decisões. Para facilitar o cálculo de cobertura, é utilizada a seguinte fórmula:

$$\text{cobertura de decisão} = \frac{\text{nº de decisões exercitados}}{\text{total de decisões}} * 100\%$$

Técnica Funcional (Caixa Preta)

É uma técnica utilizada para criar casos de teste em que o sistema é avaliado como uma caixa preta e as entradas e saídas são avaliadas para saber se estão em conformidade com os requisitos do cliente.

Como submeter o sistema a todas as possíveis entradas pode fazer com que as s execuções de teste sejam muito grandes e até mesmo infinitas (teste exaustivo), critérios de teste funcionais foram definidos para avaliar da melhor maneira possível o sistema. Dentre os principais critérios de teste funcional destacam-se (MALDONADO, 2007):

- Particionamento de classes de equivalência: Tem como objetivo determinar um subconjunto de dados finitos dividindo o domínio de entrada em classes de equivalência, que podem ser condições de entrada/saída válidas ou inválidas. Uma vez essas classes definidas, assume-se que qualquer elemento da classe pode ser um representante desta, pois todos eles devem se comportar de maneira.

Exemplo:

Derivar casos de teste com base na seguinte regra:

0 – 16	Não empregar.
16 – 18	Pode ser empregado tempo parcial.
18 – 55	Pode ser empregado tempo integral.
55 – 99	Não empregar.

Seria muito trabalhoso testar todos os valores de 0 a 99, certo?

Se partirmos do princípio que a implementação é feita como abaixo:

```
1 if (idade >= 0 && idade <= 16)
2         empregar = "NAO";
3 if (idade >= 16 && idade <= 18)
4         empregar = "PAR";
5 if (idade >= 18 && idade <= 55)
6         empregar = "INT";
7 if (idade >= 55 && idade <= 99)
8         empregar = "NAO";
```

Assim, usando a partição de equivalência apenas 4 testes satisfazem as regras com os valores de 0, 20, 38 e 46.

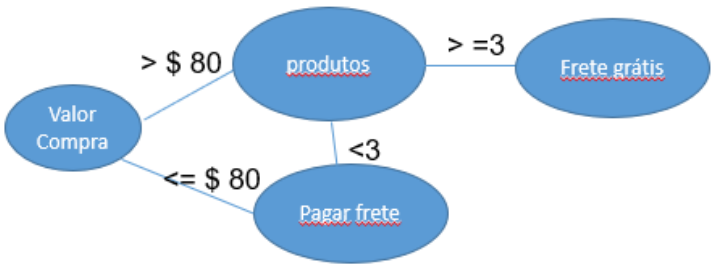
- **Análise do valor limite:** Complementa a partição de classes de equivalência usando os valores fronteiras das classes. Os dados de teste são escolhidos de forma que o limitante da classe de equivalência seja explorado observando o domínio de saída. Usando as mesmas regras do exemplo anterior, porém analisando os valores limites, temos:

Tabela 1 Exercício Análise de Valor Limite

Regra	Limite Inferior	Limite Superior
Não empregar	-1 e 0	15 e 16
Empregar em tempo parcial	15 e 16	17 e 18
Empregar em tempo integra	17 e 18	54 e 55
Não empregar	54 e 55	99 e 100

- **Tabela de Decisão (Grafo causa-efeito):** Ajuda na definição de um conjunto de casos de teste que exploram as ambiguidades e incompletudes da especificação. Para criação dos casos de teste deve-se realizar os seguintes passos: 1) Dividir a especificação em partes; 2) Identificar causas e efeitos (entradas e saídas) na especificação; 3)Analisar a semântica da especificação e transformar em grafo booleano; 4)Adicionar anotações ao grafo com as combinações de causas e efeito; 5) Converter o grafo em tabela de decisão e 6) Converter as colunas da tabela de decisão em casos de teste.

  Exemplo: Grafo representando regra de frete x quantidade e valor de compra.



	Valor da ompra	>80	>80	<=80
CAUSA	produtos	<3	>=3	
EFEITO	Pagar frete	X		X
	Frete grátis		X	

Figura 8 Tabela de Decisão

- **Transição de Estados:** Representa sistemas que esperam muitas respostas diferentes dependendo da sua condição atual ou de estado anterior. Os estados do sistema, ou objetos em teste, são isolados, identificáveis e finitos. Permite ao testador visualizar o software em: Termos de estados, Transições entre estados, as entradas ou eventos que disparam as mudanças de estado (transição), As ações que podem resultar daquelas transições. Uma tabela de estado exibe a relação entre estados e entradas, e pode destacar possíveis transições inválidas. É muito utilizada em softwares industriais embarcados e automações técnicas em geral. Exemplo, a imagem abaixo mostra os diferentes estados para acessar uma conta.

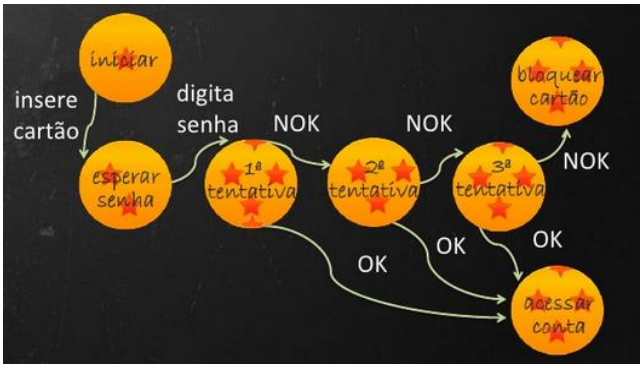


Figura 9 Gráfico de Transição (Qualidadebr, 2016)

- **Caso de Uso:** É a descrição de um uso particular do sistema feito por um ator (usuário do sistema), deriva testes a partir do diagrama de caso de uso da UML. Exemplo:

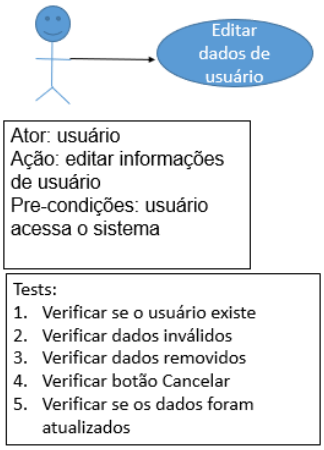


Figura 10 Caso de Uso

- Experiência: Utiliza o conhecimento e experiência de pessoas são utilizados para derivar os casos de testes. Utiliza a experiência do profissional sobre defeitos prováveis e sua distribuição.

- Suposição de Erros: Depende da experiência do testador para supor os prováveis erros do sistema.

- Ataque de falha: Uma abordagem estruturada da técnica de dedução de erros que enumera uma lista de possíveis erros e constrói testes com objetivo de atacar/cobrir estes erros.

- Testes Exploratórios: Baseia-se nos objetivos do teste, onde a especificação é rara ou inadequada. Pode servir como uma checagem do processo de teste, assegurando que os defeitos mais importantes sejam encontrados, também ocorre simultaneamente à modelagem, execução e registro de teste. Serve de complemento para outras técnicas mais formais.

Segundo James Bach no artigo “Exploratory Testing Explained” aplicamos testes exploratórios quando:

- Quando se quer ter um feedback rápido sobre um novo produto ou nova funcionalidade;

- Quando se quer aprender sobre um produto rapidamente;

- Quando se quer buscar diversidade após executar os testes tradicionais;

- Quando se quer encontrar defeitos críticos rapidamente;

- Quando comparar e investigar de forma independente e rápida o trabalho de outro testador;

- Quando se quer investigar o status de um risco em particular.

Considerações Finais

Neste capítulo foram abordados os tópicos relacionados aos principais ciclos de desenvolvimento de software e as definições básicas de teste de software, assim como foi mostrado que o teste de software está presente em todas as camadas e níveis de software e que o tipo de teste aplicado está

relacionado ao propósito do teste e as características do sistema que precisam ser avaliadas.

UNIDADE II

PROCESSO DE TESTE DE SOFTWARE

Objetivo

Neste capítulo será mostrado em detalhes as atividades de processo de teste no contexto de desenvolvimento de software além de mostrar processos de apoio ao teste de software como a gerência de configuração e o gerenciamento de incidentes.

Processo de Teste

O processo de teste, de maneira geral, consiste na condução das seguintes atividades ao longo de todo o desenvolvimento do software: planejamento e controle, análise e modelagem de casos de teste, implementação e execução de testes, avaliação dos critérios de saída e relatórios, e por último, atividade de encerramento de teste (BLACK, 2008).

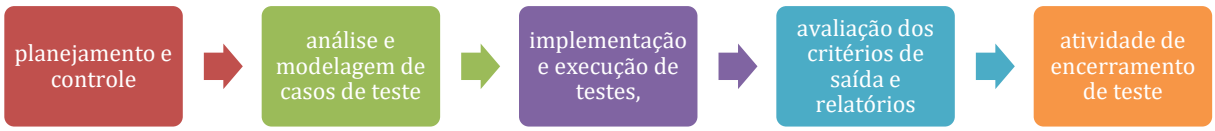


Figura 11 Processo de Teste

Planejamento e Controle

No planejamento de teste, gerentes de teste ou líderes de teste trabalham junto com o cliente para estabelecer os objetivos de teste. É elaborado um documento de plano de teste, onde estão definidas as estratégias, o escopo de teste, recursos, métodos, técnicas de teste, critérios de finalização de atividades e cronograma das atividades. O plano de teste pode ser baseado no formato padrão estabelecido pela norma IEEE 829.O gerente de testes deve acompanhar todo o processo para garantir o controle da realização das tarefas estabelecidas.

Os critérios de Entrada no planejamento definem quando começar um teste, no início de um nível de teste ou quando um conjunto de testes está pronto para execução:

- Disponibilidade do ambiente de teste;
- Preparação da ferramenta de no ambiente de teste;
- Disponibilidade de código a ser testado;
- Disponibilidade dos dados de teste.

Os critérios de saída definem quando parar de testar, no final de um nível de teste ou quando um conjunto de testes realizados atingiu um objetivo específico:

- Métricas como a cobertura de código, riscos ou funcionalidades;
- Estimativa da densidade de defeitos ou segurança nas medições;
- Custos;
- Riscos residuais, como defeitos não solucionados ou falta de cobertura de teste em algumas áreas;
- Cronograma, baseado na data de entrega do produto;

Parte importante do planejamento é a estimativa de esforço do teste, realizar estimativa é uma atividade que dá suporte ao cálculo de orçamento do projeto, alocação ou contratação de recursos e elaboração de cronograma. Estimativas não são exatas, no entanto, boas estimativas dão previsões realistas dentro de limites de tolerância.

As estimativas devem ser feitas usando como base, métricas de projetos anteriores similares e o conhecimento técnico dos executores ou especialistas.

Existem fatores para levar em consideração na hora da estimativa, como:

- Características do produto: A qualidade da especificação ou outra informação usada por projetos de teste, o tamanho do produto, a complexidade do problema, os requisitos para segurança e os requisitos para documentação.

- Características do processo de desenvolvimento: A estabilidade da organização, ferramentas usadas, processos de teste, experiência das pessoas envolvidas e pressão no prazo.

- As saídas do teste: O número de defeitos e a quantidade de retrabalho necessária.

A partir do planejamento, é necessário fazer o monitoramento e controle das atividades. O objetivo é uma visibilidade sobre as atividades do teste. As informações a serem monitoradas podem ser coletadas manualmente ou automaticamente. É necessário para tomar ações preventivas ou corretivas. Métricas são utilizadas também para medir os critérios de saída, como cobertura e também podem ser usadas para avaliar o progresso em relação ao orçamento e cronogramas planejados.

Pode ser medido no projeto a porcentagem de trabalho na preparação do caso de teste (ou porcentagem de casos de testes devidamente planejados), a porcentagem de trabalho na preparação do ambiente, a execução dos casos de testes (números de casos de teste executados ou não, testes com resultados positivos e negativos), as datas dos pontos de controle, as informações dos defeitos (densidade do defeito, defeitos encontrados e resolvidos, taxas de falha e resultado de retestes), a cobertura de requisitos, riscos ou código, a confiança subjetiva do testador sob o produto, e também o custo do teste, incluindo o custo comparado ao benefício de encontrar o próximo erro ou de executar o próximo teste.

Existem tipos de métricas que ajudam no monitoramento e controle, como:

- Métricas de Produto: Referente ao produto final que será entregue ao mercado. Exemplo: cobertura de testes, Índice de densidade de defeitos.

- Métricas de Projeto: Referente ao projeto de teste planejado para validar o produto. Exemplo: Tempo de Teste estimado X tempo de teste utilizado

- Métricas de Processo: Referente ao processo de teste executado para validar o produto. Exemplo: Defeitos encontrados X Defeitos corrigidos

Quando os testes detectam defeitos, eles mitigam o risco de qualidade. Risco é a probabilidade de um evento acontecer. O planejamento de teste baseado deve levar em conta o risco para selecionar as condições de teste, alocar os trabalhos de teste àquelas condições e priorizar os casos de teste resultantes.

Para gerenciar o risco no planejamento, existem em quatro atividades principais:

- Identificação de riscos: através de entrevistas, retrospectivas, checklists e experiências anteriores os riscos são identificados;
- Avaliação de riscos: é a categorização de cada risco determinando a probabilidade de acontecer e seu impacto. Categorização do risco significa colocar cada risco em um tipo adequado, como desempenho, confiabilidade, funcionalidade e assim por diante
- Mitigação de riscos: maneira de reduzir o impacto do risco, por exemplo, se houver problemas nos requisitos durante a identificação de riscos, a equipe do projeto pode fazer revisões de especificações de requisitos como ação de mitigação;
- Gerenciamento de riscos: planejar as respostas ao risco, monitorar e acompanhar os riscos identificados.

Análise e Modelagem

A fase de análise de testes é a atividade que define “o que” será testado na forma de condições de teste. As condições de teste podem ser identificadas pela análise da base de teste, dos objetivos de teste e dos riscos de produto. Podem ser consideradas medidas e alvos detalhados para o sucesso e devem proceder da base de teste e de objetivos estratégicos definidos, inclusive dos objetivos de teste e de outros critérios de sucesso de projetos ou stakeholders. As condições de teste também devem conduzir a modelagens de testes e

outros produtos de trabalho de testes à medida que os produtos de trabalho são criados.

Na análise e modelagem de casos de teste, os objetivos de teste são transformados em condições e casos de teste tangíveis. Casos de teste são roteiros completos e independentes para testar um cenário. Um caso de teste contém uma entrada e uma saída esperada, eles devem ser projetados com o intuito de descobrir defeitos no software, um bom caso de teste é aquele que consegue identificar um defeito não esperado no software. Nessa atividade os casos de teste são criados em um documento chamado Especificação de casos de teste (BLACK, 2008).

A modelagem de teste inclui:

- Determinar em quais áreas de teste os casos de teste de baixo nível (concretos) ou de alto nível (lógicos) são mais adequados;
- Determinar a/s técnica/s de modelagem de casos de teste que propiciam a cobertura de teste necessária;
- Criar os casos de teste que exercem as condições de teste identificadas

Os casos de teste concretos são úteis quando os requisitos são bem definidos, quando o pessoal da área de teste é menos experiente e quando a verificação externa dos testes, eles proporcionam uma reprodutibilidade excelente, mas exige trabalho de manutenção.

Os casos de teste lógicos podem propiciar uma cobertura melhor do que os casos de teste concretos porque variam um pouco sempre que são executados. Os casos de teste lógicos são úteis quando os requisitos não são bem definidos, quando o analista de teste que executará o teste é experiente.

A modelagem de casos de teste inclui a identificação do seguinte:

- Objetivo;
- Pré-condições, como requisitos de projeto ou de ambiente de teste localizado e os planos de entrega, estado do sistema etc.;

- Requisitos de dados de teste (tanto os dados de entrada do caso de teste quanto os dados que devem existir no sistema para a execução do caso de teste);
- Resultados esperados;
- Pós-condições, como dados afetados, estado do sistema, causas de processamento subsequente etc.

É importante lembrar o nível de teste, o alvo do teste e o objetivo do teste

Exemplo de caso de teste:

CT01- Cadastrar usuário válido

Pré-condição: Estar na tela de cadastro de usuário, o CPF 15362311425 não deve ter sido previamente cadastrado

Passos:

1. Clicar no campo de texto "Nome"
2. Digitar "Maria Ramos"
3. Clicar no campo de texto "CPF"
4. Digitar "15362311425"
5. Clicar no campo de texto "E-mail"
6. Digitar "mramos@yahoo.com.br"
7. Clicar no campo de texto "Senha"
8. Digitar "xy7694207"
9. Clicar no botão "Salvar"

Resultado esperado:

A mensagem "usuário cadastrado com sucesso" é exibida

Pós condição: Retornar a tela principal do sistema

Implementação e Execução

Na fase de Implementação e execução, podemos listar as seguintes tarefas a serem realizadas:

- Finalizar, implementar e priorizar os casos de teste
- Criar procedimentos de teste

- Agrupar os casos em suítes de teste para testes eficientes
- Escrever scripts automatizados para teste
- Executar testes manuais ou automatizados
- Registrar resultados da execução do teste
- Analisar e reportar incidentes
- Comparar resultados

Na fase de execução, a comparação dos resultados reais com os resultados esperados é a alma da atividade de execução de testes. O Analista de Teste deve prestar atenção a estas tarefas e se concentrar nelas, senão todos os trabalhos de modelagem e implementação de testes podem ser em vão quando as falhas não são detectadas ou um comportamento correto é classificado como incorreto (resultado falso positivo). Se os resultados esperados e os reais não baterem, houve um incidente e os resultados do teste devem ser devidamente registrados.

Na fase de avaliação dos critérios de saída e relatórios um documento de Relatório de Execução de testes e os defeitos encontrados devem ser reportados para a equipe de desenvolvimento. Resumindo temos os seguintes pontos:

- Checar logs de teste mediante critério de encerramento especificado no planejamento
- Avaliar se são necessários testes adicionais
- Elaborar relatório resumido de teste para os interessados
- E na fase de encerramento de teste temos as seguintes atividades a serem consideradas:
- Checar os entregáveis entregues
- Documentar aceite do sistema
- Levantar lista de mudanças em aberto para futuras correções

- Finalizar e armazenar o testware para reutilização
- Escrever e analisar lições aprendidas
- Aplicar melhorias na maturidade do processo a partir das informações coletadas

Atividades de Encerramento de Teste

Por fim, no encerramento das atividades de teste, assim que se determinar a conclusão da execução do teste, as principais saídas dos trabalhos de teste devem ser captadas e repassadas ao responsável ou arquivadas.

As Atividades de Fechamento possuem 4 grupos:

- Verificação de conclusão de teste: Os testes planejados foram executados.
- Transferência de artefatos de teste: Defeitos conhecidos que tenham sido delegados ou aceitos devem ser comunicados às pessoas que utilizarão e apoiarão o uso do sistema.
- Aprendizados: Realização de ou a participação em reuniões retrospectivas.
- Arquivamento de Resultados: Registros, relatórios e outros documentos e produtos de trabalho no sistema de gestão de configurações também deve ser realizado.

Gerenciamento de Configuração

O gerenciamento de configuração tem como objetivo estabelecer e manter a integridade dos produtos (componentes, dados e documentação) do software ou sistema durante todo o projeto ou ciclo de vida do produto.

É parte muito importante em um projeto de software, sua ausência pode acarretar vários problemas de projeto, como:

- As falhas reportadas pelos clientes não são reproduzidas;
- Alterações acabam sobrescrevendo outras;

- Falhas reaparecem tempos depois de terem sido corrigidas;
- Testes são executados em versões ainda não fechadas;
- Builds de instalação podem não funcionar corretamente no ambiente de teste;
- Códigos sem integração;
- Retrabalho alto;

A gestão de configuração deve estar presente desde o planejamento do projeto, deve-se estimar o tempo de sua implantação e como os artefatos do projeto, seja código ou documentos serão organizados e versionados. Uma boa gestão de configuração resulta em benefícios, como:

- Dificilmente defeitos são reabertos;
- Facilita a manutenção do software;
- Builds estáveis podem ser geradas automaticamente e com código já integrado a qualquer momento;
- Itens de software e suas funcionalidades estão organizados (baseline);
- Facilidade em voltar versões;
- Estabilidade de versões intermediárias do software;
- Artefatos organizados e com identificadores definidos (baseline).

Gerenciamento de Incidente

Um Incidente é qualquer evento (anomalia) ocorrido diferente do esperado na execução do software. Esse evento pode vir a ser causado por:

- Um defeito no software
- Um defeito no processo de teste

- Um ambiente de teste mal configurado
- Falta de um oráculo de teste definido

O Relatório de Incidentes é um documento que contém a lista dos incidentes identificados durante a execução de teste. Este documento deve conter informações detalhadas com a descrição dos incidentes, as características, classificações do incidente e em qual parte do software o problema ocorreu. Este relatório possui os seguintes benefícios:

- Prover aos desenvolvedores e outros envolvidos um retorno sobre o problema para permitir a identificação, isolamento e correção se necessário
- Prover aos líderes de teste um meio para se rastrear a qualidade do sistema em teste e o progresso do teste
- Prover ideias para aprimorar o processo de testes
- Todos os incidentes devem ser registráveis, rastreáveis, classificáveis e gerenciáveis.

Os incidentes possuem atributos, como estado, severidade, categoria, prioridade, origem, sistema operacional, reprodutibilidade e etc. Dependendo do projeto a equipe de teste pode definir mais ou menos atributos como na figura 12.

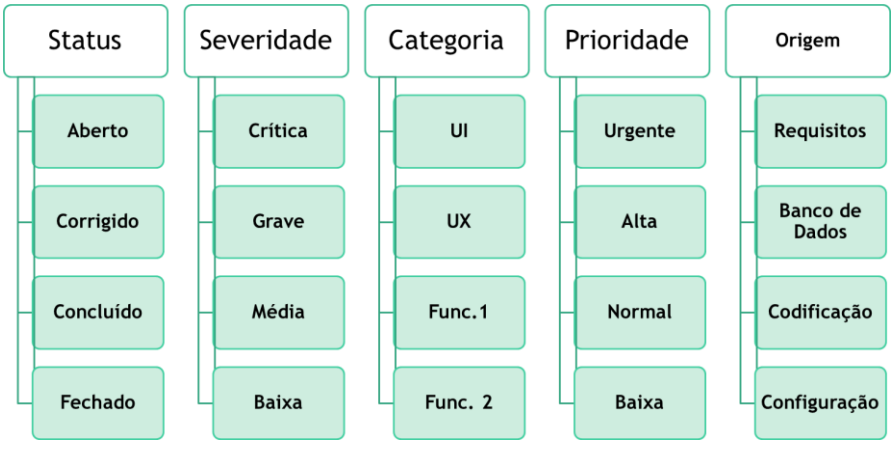


Figura 12 Atributos de Incidentes

O Status (ou estado) de um incidente pode ser Aberto se foi encontrado pelo testador e registrado no sistema de gestão de incidentes, corrigido, quando o desenvolvedor faz a correção do incidente, concluído se o incidente

foi testado novamente pelo testador e não apresentou problema e então é fechado, pois ele não ocorre mais.

A Severidade pode ser crítica, se o incidente causa um dano crítico que paralisa o sistema, grave se o incidente causa erro no sistema em que suas funcionalidades não respondem corretamente, média caso o incidente ocorra em apenas uma parte da funcionalidade e baixa quando o incidente é um erro cosmético, um botão ou uma imagem fora do lugar.

Quanto ao atributo de categoria, este deve ser definido de acordo com a natureza do projeto, podem ser listadas as funcionalidades ou as plataformas do sistema como web, servidor, mobile, interface UI, interação UX e etc.

A prioridade é a ordem de correção do incidente, é definida de acordo com o escopo e o objetivo do projeto, normalmente um incidente crítico é de prioridade urgente para ser corrigido, o incidente grave de prioridade alta e assim por diante, porém há casos de incidentes considerados de baixa severidade, porém possuem prioridade alta caso o projeto priorize a interface do sistema.

A origem seria onde está localizado a origem do incidente, pode ser um requisito mal escrito, um erro na modelagem de uma tabela no banco de dados, um defeito de codificação ou configuração do ambiente.

O processo de gerenciamento de incidentes possui um fluxo que é seguido pela maioria das ferramentas, basicamente este fluxo se apresenta como na figura 13. O início do processo o Testador realiza a execução de teste, encontra o defeito e o registra na ferramenta. O desenvolvedor verifica o defeito registrado, analisa e poderá aceitar como defeito ou considera-lo rejeitado caso não seja um defeito ou seja um registro duplicado. Quando o desenvolvedor aceita, o defeito é priorizado e corrigido. O Testador recebe a correção e executa o teste novamente (reteste), caso o defeito persista, ele é reaberto no sistema para o desenvolvedor corrigir, mas se o defeito não ocorreu novamente, então o registro é fechado.

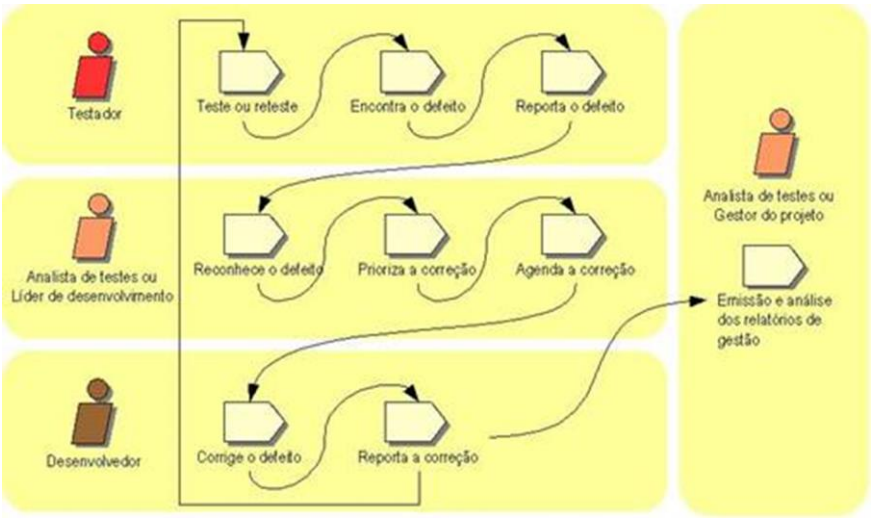


Figura 13 Processo genérico de Gerenciamento de Incidentes

Considerações Finais

Neste capítulo vimos sobre o processo de teste no contexto do desenvolvimento de software. O processo de teste deve estar presente desde o planejamento do projeto, vimos que as principais atividades de teste são planejamento e controle, análise e modelagem, implementação e execução, avaliação dos critérios de saída e relatórios, e encerramento de teste. Parte importante desse processo são o gerenciamento de configuração e o gerenciamento de incidentes, o primeiro garante que os artefatos não serão perdidos e serão corretamente versionados e sincronizados, o segundo garante o monitoramento e controle dos incidentes encontrados na execução e promove informações para o relatório de teste.

UNIDADE III
FERRAMENTAS DE APOIO

Objetivo

Neste capítulo, as ferramentas para automação de teste que facilitam o processo e execução de testes serão discutidas, primeiramente será mostrado o conceito de automação de testes e suas principais técnicas, em seguida os tipos de ferramentas mais encontradas para suporte aos projetos de software, os benefícios que as ferramentas trazem para o projeto e como se dá o processo de implantação da ferramenta.

Automação de teste

Automatizar testes significa fazer uso de outros *software*s que controlem a execução dos casos de teste. O uso desta prática pode reduzir o esforço necessário para os testes em um projeto de *software*, ou seja, executar maiores quantidades de testes em tempo reduzido. Testes manuais que levariam horas para serem totalmente executados poderiam levar minutos caso fossem automatizados (TUSCHLING, 2008).

Apesar dos testes manuais encontrarem falhas em uma aplicação, é um trabalho desgastante que consome muito tempo. Automatizar seus testes significa executar mais testes, com mais critérios, em menos tempo e sem muito esforço na execução (COSTA, 2006). Sem contar as inúmeras possibilidades que um *script* automático traz, como: cobertura de requisito e profundidade nos testes (extensibilidade), além de ser efetivo quanto à quantidade de dados de entrada que podem ser processadas (confiabilidade). Outras vantagens, que podem ser destacadas ao transformar suas rotinas de testes em *scripts* automáticos, são: segurança, reusabilidade e manutenibilidade (BERNARDO e KON, 2008).

Testes melhores e mais elaborados contribuem para um aumento na qualidade do produto final. Porém, construir uma suíte de *scripts* de teste

automáticos requer uma padronização de atividades e conhecimento em codificação e análise por parte do testador. Para fazer testes automáticos eficientes, caso o teste seja de unidade, é necessário entender a arquitetura do *software* e a base do código. Caso o teste seja de sistema, é necessário conhecer as regras de negócio, as funcionalidades, e os valores e situações permitidas como resultado.

Técnicas de Automação

Existem várias abordagens para se automatizar casos de teste, dentre elas se destaca a utilização de Ferramentas baseadas em Interface Gráfica que possuem capacidade de gravar e executar casos de teste. São as ferramentas conhecidas como rec-and-play (Record and Playback). Nessa abordagem a ferramenta interage diretamente com a aplicação, simulando um usuário real. Na medida em que a aplicação está sendo executada manualmente, a ferramenta oferece um suporte de gravação: ela captura as ações e as transforma em scripts. Estes podem ser executados posteriormente (Fantinato et al., 2009).

É bom frisar que para testes utilizando esta abordagem, o software que será testado não precisa sofrer alteração alguma. Não há necessidade de modificação, no sentido de padronizar o código, para que a aplicação se torne fácil de testar (testabilidade). Os testes nessa abordagem são baseados na mesma interface gráfica que o usuário irá utilizar. O trabalho aqui é encontrar uma ferramenta que ofereça o recurso necessário para testar sua aplicação específica.

Existem ferramentas rec-and-play para aplicações Desktop (Java Swing, interface gráfica como o KDE) e Web. A maior desvantagem de se utilizar esta técnica de automação é que o script se torna “escravo” da interface da aplicação. Para que scripts, utilizando esta abordagem, sejam criados, as ferramentas fazem uso dos nomes, posições e das propriedades dos componentes e objetos que estão dispostos na interface da aplicação. Se alguma mudança de posição, nome, tipo do componente, ocorrer, o script “quebra”, ou seja, não funciona mais (Fantinato et al., 2009).

Outra maneira, muito utilizada, de automatizar casos de teste é com o uso de Lógica de Negócio (Script Programming). Neste caso são observadas as funcionalidades da aplicação, sem interagir com a interface gráfica. Com esta abordagem serão testadas das maiores até as menores porções do código: funções, métodos, classes, componentes, entre outros.

Geralmente é pedido que se faça uma alteração no código para que o trabalho de automação fique mais simples e produtivo. Muitas vezes o código é melhorado (Refactoring) e padronizado para que se consiga testar mais facilmente e sem muitos problemas. São necessários profissionais com conhecimento em código e programação para criar os scripts automatizados. O uso deste método traz muitos benefícios, por exemplo, testes que necessitam de centenas de repetições, cálculos complexos e até mesmo integração entre sistemas são feitos facilmente utilizando esta abordagem.

Existem bibliotecas, ferramentas e frameworks que suportam esta abordagem. Bons exemplos são: JUnit, para testar unidade de código Java; FitNesse, que é um framework para testes de aceitação; MbUnit, para testar unidade em códigos .NET; Nester, para fazer testes de mutação em códigos C#; httpUnit, para testar aplicações WEB; Cactus, para testar EJB; jfcUnit e Abbot, para testar aplicações baseadas em interfaces gráficas (Collins E. e Lobão L., 2010).

Entre as maneiras de se automatizar, o presente artigo tem como base a abordagem rec-and-play. É com ela que compartilharemos as experiências, destacando suas vantagens em um projeto Web que foi desenvolvido utilizando SCRUM.

Ferramentas

Existem ferramentas de suporte a teste de software para atender às fases do processo de teste e os níveis e tipos de teste. Elas podem gerar dados para teste, podem automatizar procedimentos de teste, comparar resultados e apoiar o planejamento, modelagem e relatório de teste.

Algumas ferramentas suportam múltiplas atividades e atacam mais de um nível de teste. Há também os “frameworks de teste” que são utilizados de para pelo menos três propósitos:

- Construir ferramentas de teste
- Design de código (ex.: data-driven)
- Processo de execução de teste

A classificação das ferramentas é de acordo com as atividades de teste que elas suportam:

- Ferramentas para gerenciamento do teste
- Ferramentas de suporte para teste estático
- Ferramenta de suporte para especificação de teste
- Ferramenta de suporte para execução e registro
- Ferramenta de performance e monitoração
- Ferramenta de suporte para áreas de aplicações específicas
- Ferramenta de suporte utilizando outras ferramentas

Tipo	Objetivo/Descrição	Exemplo
Gerenciamento de Teste	Fornecem interfaces para a execução de teste, rastreamento de defeitos e gestão de requisitos juntamente com suporte para análise quantitativa e relatório dos objetos de teste	TestLink, Quality Center

Gerenciamento de Requisitos	Armazenam termos de requisitos, armazenam os atributos para os requisitos (incluindo prioridade), fornecem identificadores únicos e dão suporte ao rastreamento dos requisitos aos testes individuais	OSRMT, Spider-CL, DotProject, SIGERAR, OpenReq
Gerenciamento de Incidentes	armazenam e gerenciam relatórios de incidentes (ex.: defeitos, falhas, problemas e anomalias) e ajudam no gerenciamento do ciclo de vida de incidentes	Mantisbt, Bugzilla, Jira
Gerenciamento de Configuração	são necessárias para manter o controle da versão de diferentes builds de softwares e testes	Mercurial, Subervion, TRAC,
Suporte ao processo de revisão	Essas ferramentas auxiliam com o processo de revisão, check-lists, diretrizes de revisão, revisões on line para times grandes	GRIP, IBIS, ISPIS
Análise Estática (D)	Ajudam os desenvolvedores, testadores e os envolvidos com a qualidade a encontrar defeitos antes dos testes dinâmicos, através da aplicação de padrões de codificação	FindBugs, FxCop, Moose, cppcheck
Modelagem (D)	Ferramentas de modelagem validam o modelo do software (ex.: modelo físico de dados, para um banco de dados relacional), enumerando inconsistência e encontrando defeitos	Erwin, DBDesign

Modelagem de Teste	Geram entradas de teste ou testes a partir dos requisitos, de uma interface gráfica do usuário, dos diagramas de modelagem (estado, dados ou objetos) ou do código	ALLPAIRS, TestComposer, AGEDIS
Preparação de dados de teste (D)	Manipulam a base de dados, arquivos ou transmissão de dados para ajudar na preparação dos dados de teste a serem utilizados durante a execução de testes	Apache Jmeter, Optim, Datatect, DTM Data Generator
Ferramentas de execução do teste	Suportam linguagens de script ou parametrização de dados baseados em telas e outras customizações do teste	Selenuim, SiKuLi, BadBoy
Framework de teste de unidade (D)	Facilitam o teste de componente ou parte de um sistema por simular o ambiente em que o objeto de teste será executado, através do fornecimento de simuladores, como stubs ou drivers	Junit, TestNG, NUnit, HSQLDB
Comparadores de teste	Determinam as diferenças entre os arquivos, banco de dados ou resultados dos testes. Normalmente incluem comparações dinâmicas. Pode ser usado como Oráculo de teste.	OZJ
Ferramentas de medição de cobertura (D)	Medem a porcentagem de estruturas de código que são exercitados (comandos, decisões, ramos, módulos e chamada de funções) por um dado conjunto de testes	Rational Code Coverage, JCoverage
Ferramentas de Segurança	São utilizadas para avaliar as características de segurança do software. Isso inclui a avaliação	Netsparker, Websecurify, Wapiti, N-Stalker

	da habilidade do software em proteger confidencialidade, integridade, autenticação, autorização, disponibilidade e não repúdio de dados	
Ferramenta de Análise dinâmica (D)	Encontram defeitos que só são evidenciados quando o software está em execução, assim como dependência de tempo ou vazamento de memória. São normalmente utilizados em testes de componentes, testes de integração dos componentes e teste de middleware.	Dynamic Analysis
Ferramentas de Teste de Performance/Teste de Carga/Teste de Estresse	Monitoram e relatam como um sistema se comporta sob uma variedade de condições simuladas, em termos de número de usuários concorrentes, ou usuários virtuais	Apache Jmeter, OpenSTA, WEBLoad
Ferramentas de monitoração	Ferramentas de monitoração continuamente analisam, verificam e reportam a respeito do uso de recursos específicos do sistema	Bandwidthd, vnstat
Avaliação de qualidade de dados	Responsáveis pela avaliação da qualidade dos dados para revisar e verificar a versão de dados e regras de migração, para garantir que os dados processos estão corretos, completos e aderentes com padrões pré-definidos específicos ao contexto	Cognos Data Manager, Pentahoo

Principais Benefícios das Ferramentas (Syllabus CTFL, 2011):

- Reduzir trabalhos repetitivos (executar os testes de regressão, entrar os mesmos dados do teste e
- checar a padronização do código).
- Maior consistência e possibilidade de repetições (ex.: testes executados por uma ferramenta e testes
- derivados de requisitos).
- Avaliação dos objetivos (ex.: medidas estáticas, cobertura e comportamento do sistema)
- Facilidade do acesso a informações sobre o teste (ex.: estatísticas e gráficos referente ao progresso de
- teste, taxas de incidente e performance).

Riscos das Ferramentas:

- Expectativas falsas referentes à ferramenta (incluindo funcionalidades e facilidade de uso).
- Subestimação do tempo, custo e esforço necessário para a introdução inicial da ferramenta (incluindo treinamento e expertise externo).
- Subestimação do esforço e tempo necessários para obter benefícios significantes e contínuos do uso da ferramenta (incluindo a necessidade para mudanças no processo de teste e melhoria contínua na forma como a ferramenta é utilizada).
- Subestimação do esforço para manter os artefatos de teste gerados pela ferramenta.
- Confiança excessiva na ferramenta (quando o emprego de teste manual e modelagem de teste são mais recomendados).
- Negligência no controle de versão de ativos de teste dentro da ferramenta.
- Negligência na manutenção dos relacionamentos e questões de interoperabilidade entre ferramentas críticas, como ferramentas de gestão de requisitos, controle de versão, gerenciamento de

incidente, rastreamento de defeitos e ferramentas de múltiplos fornecedores.

- Risco de o fornecedor da ferramenta abandonar o negócio, aposentar a ferramenta, ou vender a ferramenta para outro fornecedor.
- Baixa capacidade do fornecedor para suporte, upgrades e correções de defeitos.
- Risco de suspensão de projetos de código aberto e ferramentas livres.
- Imprevistos, como a falta de habilidade para dar suporte a uma nova plataforma.

O seguinte processo poderá ajudar a organização a implantar ferramentas de teste:

Avaliar a maturidade da organização:

Ferramentas devem:

- Ajudar os projetos no esforço de teste
- Ajustar-se a cultura da empresa
- É necessário avaliar o problema e o que a empresa quer alcançar com a aplicação da ferramenta
- Automação é bom para:
- Atividades repetitivas
- Atividades que são mais seguras
- Atividades lentas

Identificar os requisitos das ferramentas

- Requisitos devem ser identificados porque serão a base para a avaliação das ferramentas dos diversos fornecedores
- Após a identificação, critérios de aceitação para essa ferramenta deve ser definido

Pré-Selecionar Fornecedores

- Elaborar lista de fornecedores
- Avaliar ferramentas com base nos requisitos e critérios definidos no passo anterior
- Fazer ranking de fornecedores que mais se adequam ao requisite buscado
- Realizar prova de conceito (ou seja, selecionar algumas pessoas e fazer o uso da ferramenta por um breve período)

Realizar Projeto Piloto

- Fazer projeto piloto para garantir que os benefícios são obtidos a partir de um ambiente controlado
- Os principais objetivos de realizar um projeto piloto são:
- Ganho de experiência
- Identificação de mudanças no processo de teste
- Avaliar custos reais e benefícios de implementação
- Fazer avaliação em ambiente controlado
- Após um bom resultado, partir para o próximo passo

Implantar

- Deverá ser baseado numa avaliação de sucesso do projeto piloto
- Implantar em toda a organização e projetos de forma gradual e aos poucos
- Treinar todos que irão usar a ferramenta

- Observe que possíveis mudanças no processo podem ocorrer. Fique atento!
- Usar adequadamente a ferramenta

Fatores de Sucesso

- Implementar a ferramenta ao restante da organização incrementalmente.
- Adaptar e aprimorar o processo para combinar com o uso da ferramenta.
- Providenciar treinamentos para novos usuários.
- Definir diretrizes de utilização.
- Implementar um modo de aprender lições com o uso da ferramenta.
- Fornece suporte ao time de teste para uma determinada ferramenta.
- Monitorar o uso e os benefícios da ferramenta

Considerações Finais

Neste capítulo vimos a definição de automação de teste e para complementar foram mostradas as principais técnicas para automatizar testes. Em seguida conhecemos os principais tipos de ferramentas e como elas ajudam o suporte a um projeto de desenvolvimento. Foram levantados pontos sobre o risco das ferramentas e os principais benefícios em um projeto, assim como o processo de implantação de uma ferramenta em um projeto.

UNIDADE IV

MELHORIAS DE PROCESSO DE TESTE

Objetivo

Neste capítulo os principais modelos para aprimoramento de processo de teste serão explicados. Estes modelos apresentam referências e com isso garantem que a organização possui um nível de maturidade em teste de software, garantindo a qualidade de seus produtos. É importante para uma organização que busca um diferencial competitivo em qualidade de software.

Aprimoramento de Processos de Teste

Deve fazer parte da organização a política de avaliação e melhorias de processos para desenvolvimento de software e de teste.

Reuniões de retrospectiva e lições aprendidas são exemplos de atividades que permitem analisar, identificar erros passados e aprender com os próprios erros, com isso possibilita o aprimoramento dos processos que as organizações utilizam para desenvolver e testar software.

Os modelos de aprimoramento de processos são divididos em duas categorias:

- O modelo de referências de processos: Avalia os recursos de uma organização em comparação com o modelo, avaliar a organização dentro da estrutura para criar um roteiro para o aprimoramento do processo;
- O modelo de referências de conteúdo: Faz avaliações orientadas para negócios das oportunidades de aprimoramento de uma organização.

Modelo TMMI

Modelo de processo estagiado, prescritivo, composto por 5 níveis de maturidade, cada nível precisa estar 85% completo:

- Nível 1: Inicial
  Representa um estado em que não há processo de teste formalmente documentado ou estruturado.
- Nível 2: Gerenciamento
  O segundo nível é atingido quando os processos de teste são claramente separados da depuração. Podem ser alcançados ao definir a política e as metas de teste, introduzindo as etapas encontradas em um processo de teste fundamental (por exemplo, o planejamento de testes) e implementando técnicas e métodos básicos de teste.
- Nível 3: Definição
  O terceiro nível é alcançado quando um processo de teste é integrado ao ciclo de vida de desenvolvimento de software e documentado em normas, procedimentos e métodos formais.
- Nível 4: Medição
  O nível quatro é atingido quando o processo de teste consegue ser medido e gerenciado eficazmente na organização em favor de projetos específicos.
- Nível 5: Otimização
  Representa um estado de maturidade em que os dados do processo de teste podem ser utilizados para ajudar a impedir defeitos e a ênfase é depositada na otimização do processo estabelecido.

TPI-Next

O modelo TPI-Next é um modelo de processo prescritivo contínuo, que define 16 áreas importantes e cada qual cobre um aspecto específico do processo de teste.

Checkpoints específicos são definidos para avaliar a área principal de cada um dos níveis de maturidade.

Os níveis de maturidade desse modelo são:

- Controlado
  - Todo o processo é executado de acordo com uma estratégia de testes, técnicas de especificação de casos de testes são utilizadas, defeitos são registrados e reportados. O testware (artefatos, ferramentas, etc.) e o ambiente de testes são bem controlados;
  - Defeitos registrados/reportados;
  - Especificação de casos de teste definida;
  - Ambiente e testware organizados.

- Eficiente
  - A automação do processo de testes pode ser uma forma de alcançar a eficiência.
- Em otimização
  - O objetivo deste nível é garantir uma melhoria contínua.

Os Artefatos para Avaliação nesse modelo são:

- Checkpoints/Checklists
  - Requisitos de cada nível;
  - Perguntas que devem ser respondidas pela equipe e classificam o processo de forma objetiva.
- Matriz de Maturidade
  - Relação entre áreas de conhecimento e níveis

CTP – Critical Test Process

O modelo CTP é avaliação dos processos críticos de teste, é uma modelo referência de conteúdo, não-prescritivo e contínuo. Certos processos de teste são cruciais para um processo e teste bem-sucedido.

O modelo identifica doze processos de teste críticos:

- Teste;
- Estabelecer contexto;
- Análise de risco de qualidade;
- Estimativa de teste;
- Plano de teste;
- Desenvolvimento de equipe de teste;
- Desenvolvimento do sistema de teste;
- Gerenciamento de versão de teste;
- Execução de teste;
- Relatório de erro;
- Relatório de resultados;
- Gestão de mudanças.

Este modelo de avaliação pode ser adaptado de acordo com as necessidades das organizações para incluir:

- Identificando desafios específicos;
- Reconhecendo características de bons processos de teste;
- Priorizando melhorias sugeridas que são importantes para a organização;
- O modelo CTP pode ser integrado em qualquer modelo;
- O modelo CTP usa métricas para comparar empresas com as melhores práticas e médias do setor, derivadas de entrevistas com participantes.

STEP - Systematic Test and Evaluation Process

O STEP é um modelo criado em 1985, é continuo, referência de conteúdo sistemático não prescritivo de teste. Ele não exige que as melhorias aconteçam em uma ordem específica.

Por ser um modelo de referência de conteúdo, ele se baseia na ideia de que o teste é uma atividade de ciclo de vida que começa durante a elaboração de requisitos e continua até a desativação do sistema.

Estas são algumas das suposições básicas do modelo STEP:

- Estratégia de teste baseada em requisitos;
- O teste começa quando o ciclo de vida de desenvolvimento de software começa;
- Os testes estão alinhados aos requisitos, bem como ao uso;
- Design de software é liderado pelo design de testware;
- Os defeitos devem ser identificados nos estágios iniciais ou evitados completamente;
- Os defeitos devem ser analisados minuciosamente;
- Os testadores devem trabalhar em equipe com os desenvolvedores.

A figura 14, ilustra o modelo STEP nos níveis de teste. Cada nível de teste possui o detalhe do plano (plan detail) , adquirir (acquire) e métrica (measure).

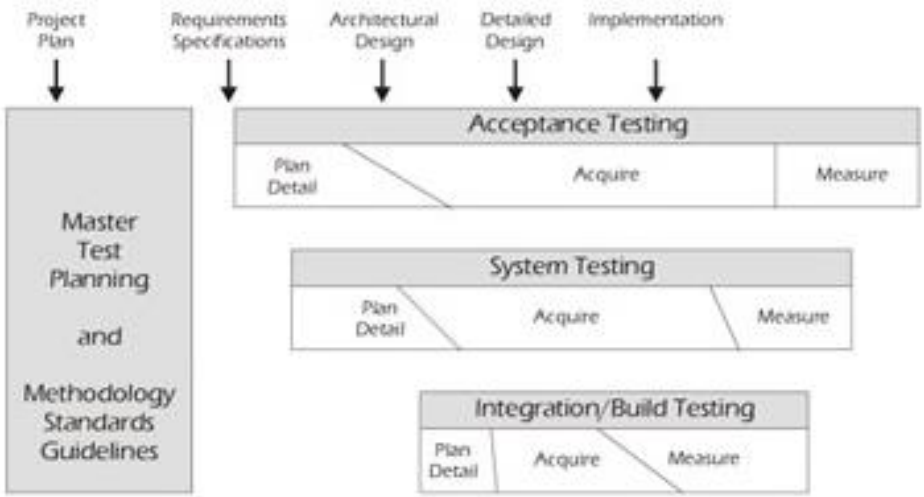


Figura 14 Step nos Níveis de Teste

TIM - Test Improvement Model

Modelo criado por T. Ericsson, A. Subotic e S. Ursing. Possui os seguintes objetivos principais:

- Identificação do estado atual;

- Implementação de pontos fortes;
- Avaliação dos pontos fracos:
    - Eliminar;
    - Melhorar.

O Modelo TIM é baseado em 4 níveis:

Baselining

- Padronização dos documentos, métodos e políticas;
- Análise e classificação dos problemas.

Cost-efectiveness

- Detectar bugs desde o início do projeto;
- Treinamento;
- Reuso.

Risk-lowering

- Envolvimento no início do projeto;
- Gastos justificados;
- Análise de produtos;
- Métricas de produtos, processos e recursos;
- Análise e gerenciamento dos riscos;
- Comunicação com as partes dos projetos.

Optimizing

- Conhecimento e entendimento através de experimentação e modelagem;
- Melhoria contínua;
- Análise dos principais problemas;
- Cooperação com todas as partes do projeto;
- Presente em todas as fases do ciclo de desenvolvimento.

O TIM utiliza para avaliação, questionários, entrevistas com pessoas chave e discussões. Depois é feita uma análise para identificar o perfil de

maturidade. A partir daí as melhorias são identificadas e sugestões para atingir a melhoria são indicadas de acordo com a necessidade da organização alinhando o custo, tempo e recursos.

MPTBR (Melhoria de Processo de Teste de Software Brasileiro)

Este é um modelo de Melhoria do Processo de Teste concebido para apoiar organizações através dos elementos essenciais para o desenvolvimento da disciplina de teste, inserida no processo de desenvolvimento de software. Ele não exige que as melhorias aconteçam em uma ordem específica, tem como objetivos:

- Aumentar a qualidade dos produtos de software através da otimização e melhoria contínua dos processos de teste;
- Fornece visibilidade relativa a maturidade do processo de teste de uma organização para o mercado de software; e
- Fomentar a melhoria contínua dos processos de teste no âmbito do desenvolvimento de software.

Figura 15 Níveis MPTBR

A Figura 15 ilustra os níveis do MPTBR, são 5 níveis: parcialmente gerenciado, gerenciado, definido, prevenção de defeitos e automação e otimização. O MPTBR utiliza o referencial teórico de modelos como TMMI, MPSBR e CMMI, a base de conhecimento são norma da ISSO 29119-2 e as certificações ISTQB (international software testing qualifications board), CSTE (certified software testing) e CBTS(certificação brasileira de teste de software).

Considerações Finais

Neste capítulo conhecemos os principais modelos para aprimoramento de processo de teste de software, os modelos tanto de referência quanto de conteúdo avaliam os processos das organizações identificando o nível de maturidade e sugerindo melhorias para atingir o nível máximo. Nesse sentido, organizações que possuem maturidade em seus processos, garantem um diferencial competitivo e qualidade de seus produtos e processos.

UNIDADE V
TESTES ÁGEIS

Objetivo

Neste capítulo serão mostrados os princípios do teste ágil, como os testes se organizam nesse ambiente, o papel do testador em projetos de metodologias ágeis e as principais práticas ágeis como a Integração Contínua, TDD (test driven development), ATDD (Acceptance test driven development) e BDD (behavior driven development).

Testes Ágeis

Segundo Crispin e Gregory (2010) testes ágeis não significam apenas testes em projetos ágeis, mas testar uma aplicação com um plano para aprender sobre ela e deixar as informações dos clientes guiar os testes, tudo respeitando os valores ágeis. Segundo as autoras, para uma abordagem de teste bem sucedida e ágil, deve-se levar em conta os seguintes fatores: Olhar o processo no seu alto nível (Big Picture); Colaborar com o Cliente; Fundamentar as práticas ágeis com o time de projeto; Fornece e obter feedback contínuo; Automatizar testes de regressão; Adotar uma mentalidade Ágil, e Usar a abordagem considerando que todos são um único time com um único objetivo.

Testes ágeis envolvem a perspectiva do cliente o mais cedo possível, testando mais cedo e frequentemente, tornando assim, o código mais estável, já que incrementos do software são liberados continuamente no desenvolvimento ágil. Isto é normalmente feito usando script automatizado para acelerar o processo de execução do teste e reduzir o custo de todo o processo de execução de testes (RAZAK e FAHRURAZI, 2011).

Automação de teste é considerada a atividade chave para metodologias ágeis e o elemento principal dos testes ágeis (CRISPIN e GREGORY, 2010). Enquanto que testes tradicionais focam principalmente em testes manuais e exploratórios criticando o produto, mas não desempenhando um papel

produtivo no processo de desenvolvimento apoiando a criação do produto. Além disso, seu foco é revelar falhas. Nos testes ágeis, a equipe não está envolvida apenas em identificar as falhas, mas também em preveni-las. Assim, teste ágil é um desafio para testadores acostumados à estratégia tradicional, principalmente porque eles não precisam esperar pela entrega do sistema para iniciar suas tarefas, mas sim precisam ser proativos e iniciar os testes desde o início do projeto junto com os desenvolvedores (MAIA et al., 2012).

Diferentes testes possuem diferentes propósitos, na figura 3 é mostrado um diagrama de quadrantes que descrevem a forma como cada tipo de teste reflete as razões diferentes para testar um software (COLLINS et al, 2012).

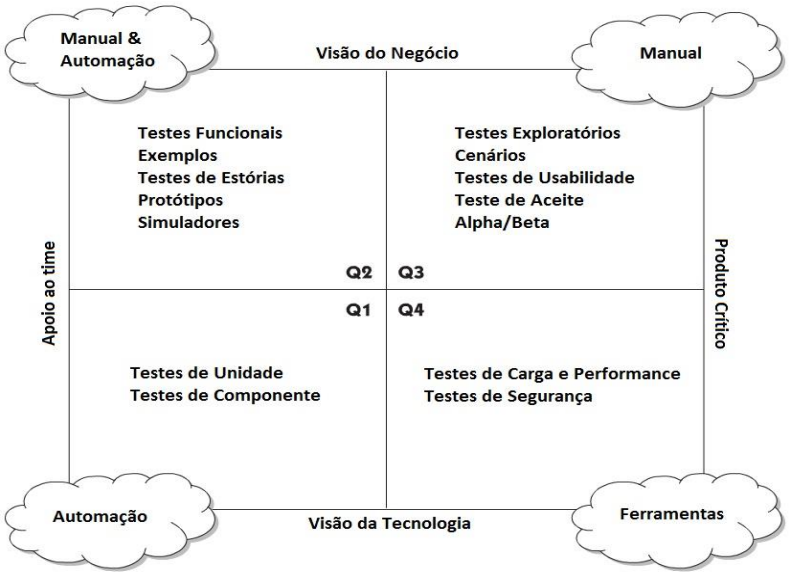


Figura 16 Quadrantes de Testes Ágeis (apud SANTOS, 2011)

- (Q1) Testes de apoio à programação: teste do nível técnico - Será que este método de fazer o que o deveria?.
- (Q2): Testes de funcionalidades que apoiam à programação: teste de implementação da lógica do negócio - O código faz o que deveria?.
- (Q3) Testes de regras de negócio do cliente para criticar o produto: valida as regras de negócio - O código faz algo que não deveria Há falta de requisitos?.

- (Q4) Testes para criticar o produto: teste de características não funcionais - Existem vulnerabilidades? O sistema consegue lidar com uma carga limite? É rápido o suficiente?.

O teste ágil é um desafio para os testadores que estão acostumados a trabalhar em projetos tradicionais, principalmente por causa da proatividade e da colaboração com os desenvolvedores como uma equipe única e ser bem sucedido neste desafio requer experiência, superar barreiras técnicas e comportamentais.

A estratégia de teste amplamente difundida é a pirâmide de teste idealizada por Mike Cohn (COHN, 2004) que está ilustrada na figura. Segundo essa pirâmide de automação de teste, a base da automação em um projeto ágil devem ser testes unitários ou de componente, pois não há código sem um ou mais testes unitários associados a ele.

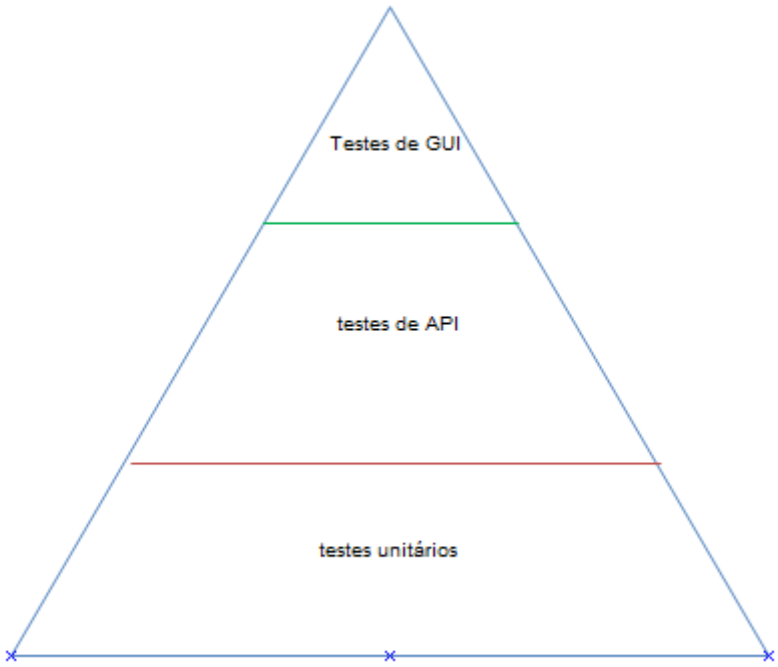


Figura 17 Pirâmide de automação de teste de Mike Cohn (adaptado de CRISPIN e GREGORY, 2010)

Na metade da pirâmide são localizados os testes de aceitação em nível de API, que são testes funcionais, mas não são baseados na interface e sim no código das regras de negócio. Por fim, em menor quantidade estariam os testes

baseados na interface e poucos testes manuais não automatizados. Para elas, os seguintes princípios ágeis devem estar presentes na estratégia:

- Um único time de projeto: todo time é responsável pela qualidade do projeto e pelas atividades de teste.
- Aprendendo e fazendo: usar práticas ágeis para codificar testes.
- Simplicidade: tentar a solução mais simples primeiro.
- Feedback iterativo: realizar tarefas de automação a cada iteração e usar reuniões de retrospectiva para avaliar as tarefas.
- Quebrar histórias em pedaços menores.

A figura 18 mostra uma estratégia aplicada no desenvolvimento ágil. No Planejamento de uma interação (Sprint Planning) são definidos os testes que serão executados, as responsabilidades, as ferramentas a serem utilizadas, o risco e a definição de Pronto (estória implementada, integrada e testada). Para seguir a pirâmide ideal de automação de teste (Figura 17) durante a interação devem ser feitos testes de unidade, testes de API e testes de sistema automatizados, para isso pode-se utilizar uma abordagem como TDD, ATDD ou BDD e no fim da interação há a reunião de retrospectiva para que a equipe aponte o que foi bom, o que deu errado e o que poderia ser melhorado para a próxima iteração.

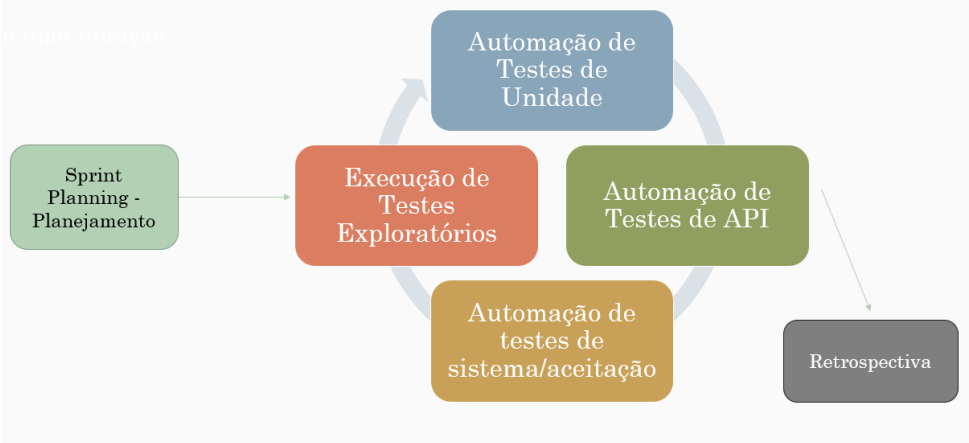


Figura 18 Estratégia de Processo de Teste no Desenvolvimento Ágil

A Configuração do ambiente de automação de teste é uma tarefa importante que necessita do envolvimento de toda equipe do projeto para que

o conhecimento não fique apenas concentrado em apenas uma pessoa, o que causa risco ao projeto em caso de ausência ou substituições de time

Vários autores consideram que para um projeto executar uma metodologia de desenvolvimento ágil, é necessário possuir um ambiente de integração contínua. O Processo de Integração Contínua é a prática no onde toda equipe deve integrar seu trabalho sempre a cada alteração. A figura 13 mostra como o ambiente de integração contínua funciona e interage no projeto. Profissionais de Teste e Desenvolvimento enviam código e scripts de teste (unitários, API ou se sistema) para o servidor controlador de versão. Este é monitorado pelo servidor de Integração Contínua que realiza as seguintes tarefas: Compila o código; executa scripts de teste; caso os testes passem, ele empacota o código testado e envia para o servidor de produção; Lá são executados os testes de performance, segurança e testes manuais exploratórios. Caso algum teste automático apresente falha, um e-mail é enviado para toda equipe reportando a falha e o código não é empacotado e enviado ao servidor de produção.

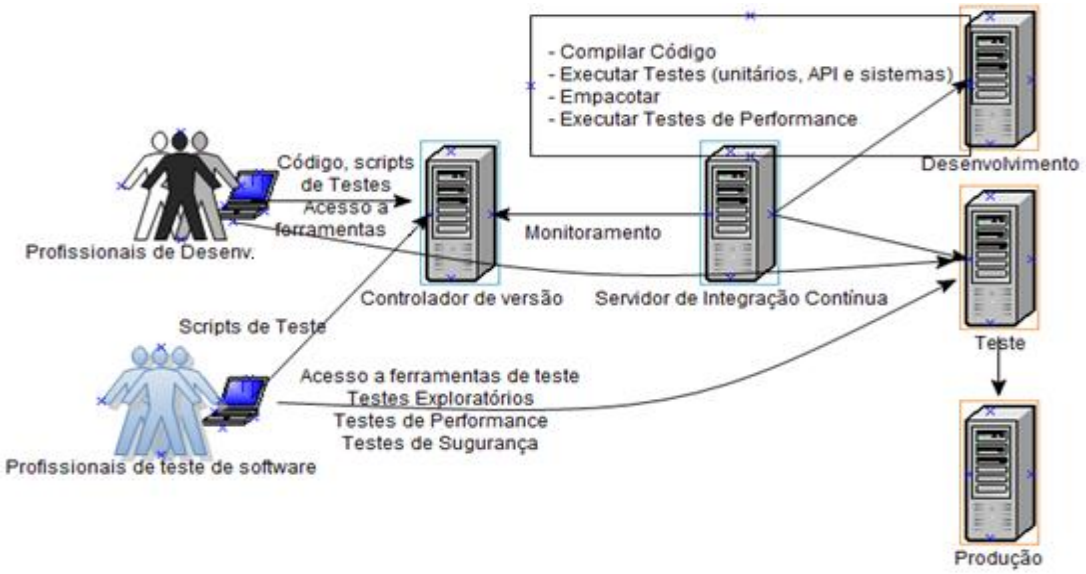


Figura 13: Estrutura de Ambiente de Integração Contínua (Collins et al, 2012)

Destaca-se que a facilidade deste ambiente está em promover integração das ferramentas usadas no projeto, respostas rápidas, dados estatísticos para o projeto e fácil acesso às ferramentas e seus resultados.

A partir desse ambiente é possível que todo time do projeto tenha acesso rápido aos resultados de execuções de teste e logs de execução facilitando decisões rápidas quando surge algum problema no software.

As principoais vantagens da Integração Contínua são:

- A redução de riscos obtida através da detecção e correção de defeitos mais cedo;
- Automação de processos manuais e repetitivos (compilação, integração de base de dados, teste, inspeção, distribuição e feedback);
- Permitir uma melhor visibilidade de projeto (estatísticas reais e recentes fornecidas nos relatórios gerados);
- Verificações automáticas de cobertura, validação de padrões de design de código e padronização.

O Papel do Testador Ágil

Em projetos que utilizam metodologias ágeis, cada integrante da equipe é responsável por testar os resultados de seu trabalho (TALBY et al. 2006). Todos os membros do time de projeto testam o sistema e este deve ser verificado em todos os seus níveis e camadas.

A literatura técnica indica que os testes de unidade sejam feitos pelos programadores antes e o código deve ser escrito para que o caso de teste criado seja bem sucedido, essa técnica se chama Desenvolvimento Dirigido a Testes. Em testes de sistema e de aceitação, os profissionais especializados em teste realizam essa atividade, porque os programadores não possuem o perfil indicado e geralmente não querem encontrar falhas nas funcionalidades que eles mesmos criaram (CRISPIN e GREGORY, 2010).

O testador que atua em projetos de metodologia ágil não exerce as mesmas responsabilidades que exerceria em metodologias tradicionais pois o seu papel deve ser desempenhado de acordo com os critérios a seguir (SANTOS et al, 2011):

- Negociar qual o nível de qualidade esperado pelo cliente, não o seu padrão de qualidade, mas o padrão que cliente desejar e estiver disposto a pagar;
- Clarificar histórias e esclarecer suposições;
- Prover estimativas para as atividades de desenvolvimento e testes;
- Garantir que os testes de aceitação verificam se o software foi construído conforme o nível de qualidade definido pelo cliente;
- Ajudar o time a automatizar os testes e a desenvolver código testável;
- Prover feedback contínuo para manter o projeto no rumo certo.

TDD – Test Driven Development

O Paradigma de programação Test Driven Development foi criado por Kent Beck. Nele, o desenvolvedor escreve um caso de teste automatizado que define uma funcionalidade ou melhoria. Então, é produzido código que possa ser validado pelo teste para posteriormente o código ser refatorado para um código sob padrões aceitáveis. O fluxo basicamente consiste em Adiconar primeiro um teste unitário, depois executar, este deve falhar, então o código é implementado com o objetivo de passar o teste, o teste é executado novamente e deve passar, por fim o código é refatorado e recomeça o ciclo como na figura 19.

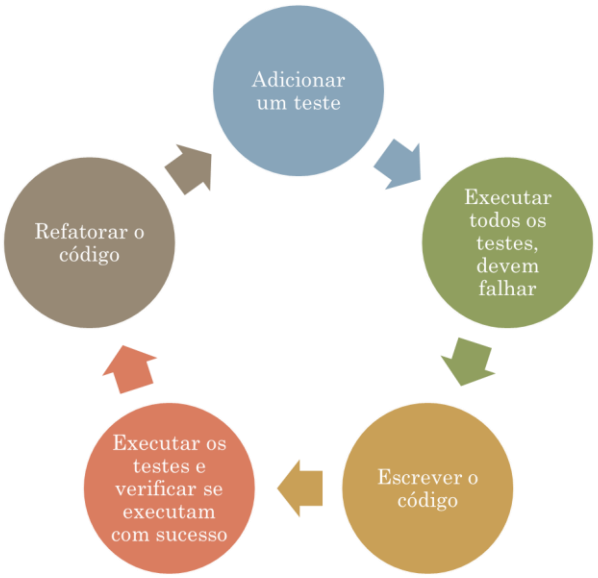


Figura 19 Fluxo TDD

O TDD possui o benefício de que o código implementado se torna menos acoplado, mais modularizado, mais flexível, extensível e fácil de realizar manutenção. Desenvolvedores pensam no software em pequenas unidades que podem ser re-escritas, desenvolvidas e testadas independemente e integradas depois. Com isso, se garante a prevenção de defeitos de código.

O TDD também possui limitações, como:

- Pode ser difícil e complexo devido a dependência de stubs e mocks
- Precisa de suporte gerencial da organização
- Os próprios testes se tornam parte da manutenção do projeto.
- Os testes precisam ser bem escritos para evitar “pontos cegos” no código
- O alto número de testes de unidades pode trazer um falso senso de segurança, resultando em menor nível de atividades de garantia de qualidade, como testes de integração e aceitação.

ATDD – Acceptance Test Driven Development

Esta prática busca desenvolver dirigido a testes de aceitação, ou seja, os requisitos executáveis são os guias do desenvolvimento. Para isso cenários de teste de usuário são implementados primeiro por testadores e em seguida o teste de unidade, os testes falham e então o código deve ser implementado primeiro para passar nos testes unitários e depois para passar no teste de

aceitação e finalmente ser refatorado, como mostra a figura 20. É um ciclo onde deve ser feito o debate dos cenários de teste de aceitação para desenvolver e revisar.

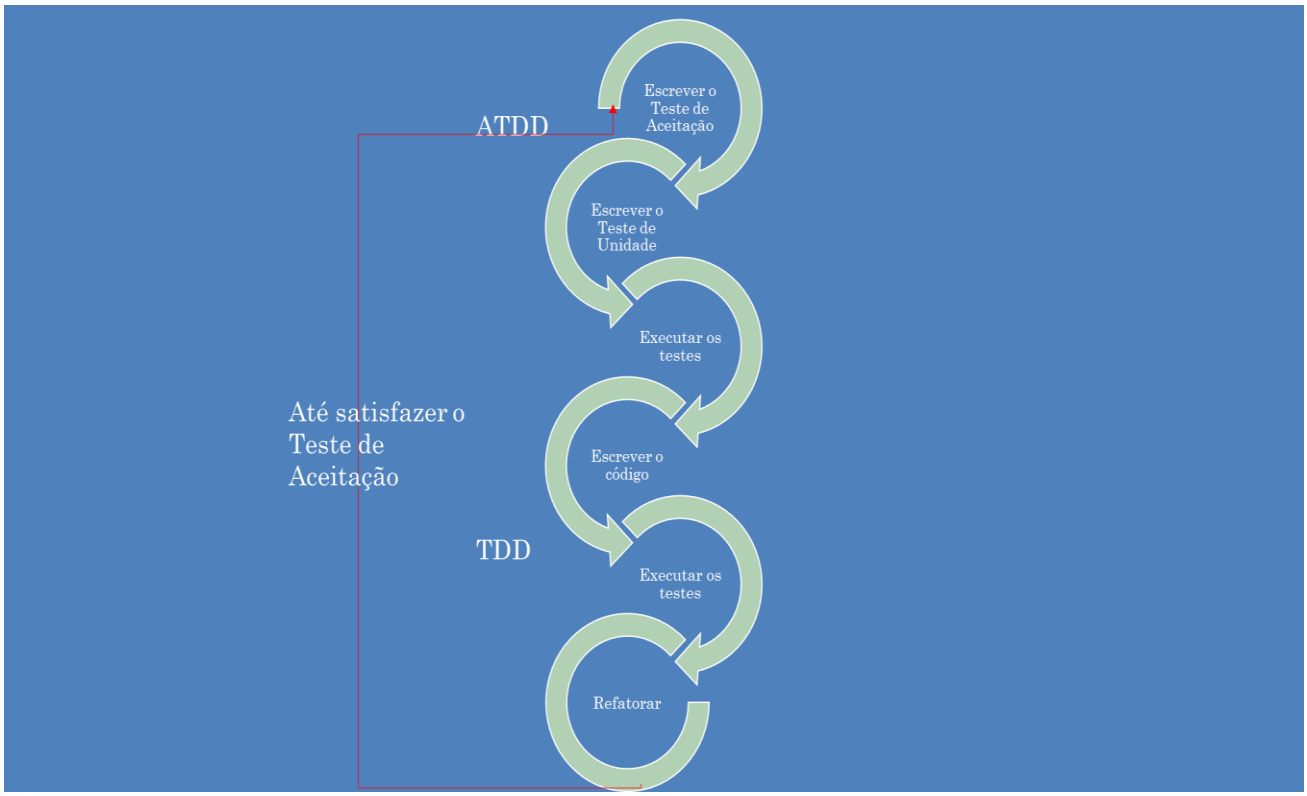


Figura 20 ATDD

Exemplo de cenário de teste de aceitação:

Cenário: Receber Notificação

Dado usuário está logado
E usuário cadastrado no site
Quando usuário confirma o recebimento do alerta
Então o alerta é criado para envio de e-mail.

O teste de aceitação deve ser bem detalhado representando um cenário de uso real da aplicação. Ferramentas que podem ser utilizadas de apoio: cucumber, RSpec, fitnesse.

BDD – Behavior Driven Development

No BDD, o desenvolvimento é dirigido a comportamento. Testes descritos em sentenças e frases que representam o comportamento esperado. Nesta abordagem todos os envolvidos participam da definição do comportamento do software. Todo time participa da elaboração da descrição do comportamento, o que favorece o aprendizado e o foco em atender às regras de negócio.

Exemplo de cenário de comportamento :

Funcionalidade: Alerta de Recebimento
Em ordem de comprar um produto disponível
Como cliente do site
Eu quero ser notificado quando um produto voltar a ficar disponível.

(ATDD)

Cenário: Receber Notificação
Dado usuário está logado
E usuário cadastrado no site
Quando usuário confirma o recebimento do alerta
Então o alerta é criado para envio de e-mail.

No exemplo podemos observar que o ATDD complementa do BDD dando detalhamento ao comportamento. As Ferramentas que podem ser usadas são Cucumber, Rspec, Jbehaviour.

Considerações Finais

Neste capítulo, foi visto que o desenvolvimento ágil para desenvolvimento de software requer uma estratégia de teste de abordagem proativa, técnica e colaborativa. Nesse sentido, o papel do testador passa a ser mais técnico e desafiador, a comunicação e a colaboração são fatores muito

importantes para que o ambiente de teste automático possa ser implementado e executado. As abordagens de TDD, ATDD e BDD integram a equipe de desenvolvimento, teste e negócio fazendo com que o time se torne multitarefa para atender e agregar valor ao cliente final.

Referências

LOBÃO Luana, Preparatório CTFL – Testing and Play, 2017.

MALDONADO, J.; DELAMARO, M.; JINO, M., Introdução ao Teste de Software, ed. Campos 2007.

PRESSMAN, R. S., "Software Engineering: A Practitioner's Approach", McGraw-Hill, 6th ed, Nova York, NY, 2005.

Syllabus, BSTQB, ctal- ta 2012. Disponível em: http://www.bstqb.org.br/ Arquivo capturado em 02/01/2019

Syllabus, BSTQB, ctal- tm 2012. Disponível em: http://www.bstqb.org.br/ Arquivo capturado em 02/01/2019

Crispin, L.; Gregory, J.; Agile Testing: A Practical Guide for Testers and Agile Teams, Addison-Wesley

ROCHA, A. R. C., MALDONADO, J. C., WEBER, K. C. et al., "Qualidade de software – Teoria e prática", Prentice Hall, São Paulo, 2001.

Qualidade BR, disponível em: https://qualidadebr.wordpress.com/, Arquivo capturado em 02/01/2018

Rick D. et al, Systematic Software Testing, 2002 disponível em: https://flylib.com/books/en/2.174.1/an_overview_of_the_testing_process.html#fastmenu_13 Arquivo capturado em 02/01/2019

Sandro Ronaldo Bezerra Oliveira, UFPA disponível em http://www.ufpa.br/srbo/Disciplinas/CBCC_CBSI_Mestrado_Qualidade/Aulas/Aula05.pdf Arquivo capturado em 02/01/2019

SOMMERVILLE, I. Engenharia de Software. São Paulo: Prentice Hall: 2003. 606p.

KRUCHTEN, P. The Rational Unified Process An Introdution. Massachusetts, Addison Wesley, 2000.

BERTHOLDO, L.; BARBAN, L., "Adaptação do Scrum ao Modelo Incremental", 2010.

TAVARES, A. Gerência de Projetos com PMBOK e SCRUM Um estudo de caso. Faculdade Cenecista Nossa Senhora do Anjos. Gravataí, 2008.

INTHURN, C., Qualidade e Teste de Software, ed. Visual Books, 2001.

FREEMAN, S.; PRYCE, N.; MACKINNON, T.; WALNES, J.; Mock Roles, not Objects, 2004, disponível em: <http://www.mockobjects.com/files/mockrolesnotobjects.pdf>, acessado em Dezembro de 2018.

BLACK R.; MITCHEL J., Advanced Software Testing, Vol. 3, ed. Rockynook, 2008

TUSCHLING, O., Software Test Automation, 2008. Disponível em: <http://www.stickyminds.com/getfile.asp?ot=XML&id=14908&fn=XDD14908filelistfilename1%2Epdf>, acessado em abril de 2018.

BERNARDO, P. ; KON, F. (2008) “A Importância dos Testes Automatizados”, Disponível em: <http://www.ime.usp.br/~kon/papers/EngSoftMagazine-IntroducaoTestes.pdf>, Acessado em Outubro de 2012.

Fantinato, M. et al. (2009) “AutoTest – Um Framework Reutilizável para a Automação de Teste Funcional de Software”, disponível em http://www.sbc.org.br/bibliotecadigital/download.php?paper=255, acessado em Outubro 2018.

Collins E. e Lobão L.. Experiência em Automação do Processo de Testes em Ambiente Ágil com SCRUM e ferramentas OpenSource. IX Simpósio Brasileiro de Qualidade de Software - 2010, 303 - 310

RAZAK, R.A.; FAHRURAZI, F.R., "Agile testing with Selenium," Software Engineering (MySEC), 2011 5th Malaysian Conference in, vol., no., pp.217,219, 13-14 Dec. 2011. doi: 10.1109/MySEC.2011.6140672.

CRISPIN, L.; Gregory, J. Agile Testing: A Practical Guide for Testers and Agile Teams. Addison-Wesley, 2010.

MAIA, N.; MACEDO,G.; COLLINS, E.; DIAS-NETO, A. C.;”Aplicando Testes Ágeis com Equipes Distribuídas: Um Relato de Experiência” In: Simpósio Brasileiro de Qualidade de Software, 2012, Fortaleza.

TALBY, D., Keren, A., Hazzan, O., and Dubinsky, Y. 2006. Agile Software Testing in a Large-Scale Project. Software, IEEE 23, no. 4, 30-37.

COLLINS, E.; DIAS-NETO, A.; DE LUCENA, V.F.; , "Strategies for Agile Software Testing Automation: An Industrial Experience," Computer Software and Applications Conference Workshops (COMPSACW), 2012 IEEE 36th Annual , vol., no., pp.440-445, 16-20 Julho de 2012. doi: 10.1109/COMPSACW.2012.84.

SANTOS, A. M.; Karlsson, B. F.; Cavalcante, A. M. Uma Abordagem Empírica para o Tratamento de Bugs em Ambientes Ágeis. Anais do Workshop Brasileiro de Métodos Ágeis 2011 (WBMA) , Brazil 2011.