Certified Tester

Agile Technical Tester Syllabus

versão 1.1 (2019)

International Software Testing Qualifications Board

Direitos autorais

Histórico da Revisão

Versão	Data	Observações
v0.1	11/01/2017	Standalone sections
v0.2	24/05/2017	WG review comments on v01 incorporated
v0.3		WG review comments on v02 incorporated
v0.7		Alpha review comments on v03 incorporated
v0.71		Working group updates on v07
v0.9	30/12/2017	Alpha candidate
v0.91	6/01/2018	Alpha release
v0.98	12/01/2019	Beta candidate
v0.99		Beta release
2018		GA version
2019	12/02/2019	Minor changes in Text, Added several LOs (K1, K2 levels), Added Copyright information, Added learning time in all chapters
2019	14/03/2019	Changed LOs to adapt to official short name “ATT” Modified “Acknowledgements”
2019	06/05/2019	Some minor changes
2019	07/06/2019	Minor changes
2019	08/06/2019	Redactional changes
2019	10/07/2019	Preparation for Beta Review
2019	14/09/2019	Enhancements, corrections post Beta Review
2019	14/11/2019	Release date
2019	09/12/2019	Changed the ISTQB logo ®; Changed the K-level training duration (Chapter 0.6)

Histórico da versão BSTQB

Versão	Data	Observações
1	14/06/2023	Padronização do layout com o ISTQB
2	28/06/2024	Adequação visual
3	06/01/2025	Novo padrão visual do ISTQB

Índice

Direitos autorais 2
Histórico da Revisão 3
Índice 4
Agradecimentos 5
0 Introdução 6
0.2 Objetivo deste documento 6
0.3 Visão geral 6
0.4 Objetivos de aprendizagem examináveis 6
0.5 O exame de certificação 6
0.6 Credenciamento 7
0.7 Como esse syllabus está organizado 7
1 Engenharia de Requisitos [120 min] 8
1.1 Técnicas de engenharia de requisitos 9
1.1.1 Analisar as histórias de usuários e épicos usando técnicas de engenharia de requisitos 9
1.1.2 Identificar os critérios de aceite usando a engenharia de requisitos e as técnicas de teste 10
2 Teste no Ágil [180 min] 12
2.1 Desenvolvimento ágil e técnicas de teste 13
2.1.1 Desenvolvimento orientado por teste (TDD) 13
2.1.2 Desenvolvimento orientado pelo comportamento (BDD) 14
2.1.3 Desenvolvimento orientado por testes de aceite (ATDD) 15
2.2 Teste baseado na experiência no Ágil 16
2.2.1 Combinando as técnicas baseadas na experiência e testes caixa-preta 16
2.2.2 Criando cartas de teste e interpretando seus resultados 16
2.3 Aspectos da qualidade do código 17
2.3.1 Refatoração 17
2.3.2 Revisões de código e análise de código estático para identificar defeitos e débito técnico 18
3 Automação de teste [180 min] 20
3.1 Técnicas de automação de teste 21
3.1.1 Teste orientado por dados 21
3.1.2 Teste orientado por palavras-chave 21
3.1.3 Aplicando a automação de teste a uma determinada abordagem de teste 23
3.2 Nível de automação 24
3.2.1 Compreender o nível necessário de automação de teste 24
4 Implantação e entrega [120 min] 27
4.1 Integração contínua, teste contínuo e entrega contínua 28
4.1.1 Integração contínua e seu impacto nos testes 28
4.1.2 O papel do teste contínuo na entrega e na implantação contínua 29
4.2 Virtualização de serviço 29
Referências 31
Terminologia Ágil 33
Apêndice 34

Agradecimentos

Este documento foi produzido pela equipe *International Software Testing Qualifications Board Agile Working Group*, liderada por Rex Black (presidente), Michaël Pilaeten (vice-presidente e presidente da AI) e Renzo Cerquozzi (product owner).

A equipe do *Advanced Level Agile* agradece à equipe de revisão e aos Conselhos Nacionais por suas sugestões.

No momento em que o syllabus do *Advanced Technical Agile Technical Tester* foi concluído, o grupo de trabalho tinha a seguinte composição: Michaël Pilaeten (Presidente), Renzo Cerquozzi (product owner), Alon Linetzki (grupo de trabalho de marketing), Leanne Howard (grupo de trabalho do glossário) e Klaus Skafte (grupo de trabalho de exames).

Autores: Leo van der Aalst, Renzo Cerquozzi, Bertrand Cornanguer, István Forgács, Jani Haukinen, Noam Kfir, Sammy Kolluru, Alon Linetzki, Tilo Linz, Michaël Pilaeten, Marie Walsh.

Revisores internos: Michael Arefi, Vojtěch Barta, Renzo Cerquozzi. Graham Bath, Laurent Bouhier, Anders Claesson, Alessandro Collino, David Janota, David Evans, Leanne Howard, Matthias Hamburg, Kari Kakkonen, Tor Kjetil Moseng, Meile Posthuma, Salvatore Reale, Marko Rytkönen, Sarah Savoy, Klaus Skafte, Mike Smith e Chris van Bael.

Criação e revisão de exemplos de perguntas: Armin Born, Renzo Cerquozzi, Alon Linetzki, Tilo Linz, Jamie Mitchell, Jani Haukinen, Tobias Horn, Chris van Bael, Suruchi Varshney.

A equipe também agradece às seguintes pessoas, dos Conselhos Nacionais e da comunidade de especialistas em Ágil, que participaram da revisão, comentários e votação do *Foundation Agile Extension Syllabus*: Adam Roman, Armin Beer, Beata Karpinska, Chris Van Bael, Erwin Engelsma, Giancarlo Tomasig, Gary Mogyorodi, Ingvar Nordström, Jana Gierloff, Jörn Munzel, Jurian van de Laar, Kari Kakkonen, Laurent Bouhier, Marko Rytkönen, Martin Klonk, Matthias Hamburg, Meile Posthuma, Paul Weymouth, R. Green, Richard Seidl, Rik Marselis, Stephanie Ulrich, Stephanie van Dijck, Tal Pe'er, Tilo Linz, Veronica Seghieri e Wim Decoutere.

Um agradecimento especial a Galit Zucker, secretário geral do *ISTQB®*, pela orientação e suporte.

Este documento foi formalmente aprovado para divulgação pela Assembleia Geral do ISTQB® em 18 de outubro de 2019, e sua versão na Língua Portuguesa em 9 de março de 2020.

O BSTQB® agradece aos voluntários do WGT BSTQB® pelo empenho e esforço na tradução e revisão deste material: Eduardo Medeiros Rodrigues, George Fialkovitz Jr, Irene Nagase, Paula Renata Z. de Souza, Osmar Higashi, Yuri Fialkovitz.

0 Introdução

0.2 Objetivo deste documento

Este documento forma a base de conhecimento para a certificação *ISTQB® CTAL-ATT Agile Technical Tester*. O ISTQB® fornece este *syllabus* da seguinte forma:

1. Aos Conselhos Nacionais, para traduzirem ao seu idioma local e credenciar os provedores de treinamento. Os Conselhos Nacionais podem adaptar o syllabus às suas necessidades linguísticas específicas e modificar as referências para se adaptarem às suas publicações locais.
2. Aos Conselhos de Exame, para derivar questões de exame em sua língua local com base nos objetivos de aprendizagem para cada módulo.
3. Aos Provedores de Treinamento, para que produzam o material didático e determinem métodos apropriados de ensino.
4. Aos Candidatos à Certificação, como uma fonte para se prepararem para o exame.
5. À Comunidade Internacional de Software e à Engenharia de Sistemas, para avançar a profissão de software e testes de sistema, e servir como base para livros e artigos.

O *ISTQB®* permite que outras entidades usem esse syllabus para outros fins, desde que obtenham autorização prévia por escrito.

0.3 Visão geral

O documento de visão geral do *ISTQB® CTAL-ATT Agile Technical Tester* [ISTQB_ATT_OVIEW] inclui as seguintes informações:

- Resultados de negócios para o *syllabus*;
- Resumo do *syllabus*;
- Relações entre os *syllabi*;
- Descrição dos níveis cognitivos (níveis K);
- Anexos de testes de software e sistemas.

0.4 Objetivos de aprendizagem examináveis

Os Objetivos de Aprendizagem suportam os Resultados de Negócios e são usados para criar o exame para obter a Certificação *ISTQB® CTAL-ATT Agile Technical Tester*. Em geral, todos os capítulos desse syllabus são examináveis no nível K1. Ou seja, o candidato reconhecerá, lembrará e recordará de um termo ou conceito. Os objetivos de aprendizagem nos níveis K2, K3 e K4 são mostrados no início do capítulo pertinente.

0.5 O exame de certificação

O exame de certificação *ISTQB® CTAL-ATT Agile Technical Tester* será baseado nesse *syllabus*. As respostas às perguntas do exame podem exigir o uso de material com base em mais de um capítulo desse syllabus. Todos os capítulos são examináveis, exceto a Introdução e os Apêndices. Normas, livros e outros conteúdos programáticos do ISTQB® estão incluídos como referência, mas seu conteúdo não é passível de exame, além do que é resumido neste programa, a partir de tais normas, livros e outros conteúdos programáticos do ISTQB®.

O formato do exame é de múltipla escolha.

Os exames podem ser realizados como parte de um curso de treinamento credenciado ou de forma independente (p. ex., em um centro de exames ou em um exame público). A conclusão de um curso de treinamento credenciado não é um pré-requisito para o exame.

0.6 Credenciamento

Um Conselho Nacional do *ISTQB*® pode credenciar Provedores de Treinamento cujo material do curso segue este syllabus.

Os Provedores de Treinamento devem obter as diretrizes de credenciamento da diretoria ou do órgão que realiza o credenciamento. Um curso credenciado é reconhecido como estando em conformidade com este material, podendo ter um exame ISTQB® como parte do curso.

0.7 Como esse syllabus está organizado

Existem quatro capítulos com conteúdo examinável.

No cabeçalho do nível superior de cada capítulo está especificado entre colchetes o tempo mínimo que deve ser dedicado ao estudo do capítulo.

Para cursos de treinamento credenciados, o syllabus exige um mínimo de 10 horas de instrução, distribuídas pelos capítulos da seguinte forma:

- Capítulo 1: Engenharia de Requisitos, 120 minutos
- Capítulo 2: Teste no Ágil, 180 minutos
- Capítulo 3: Automação de teste, 180 minutos
- Capítulo 4: Implantação e entrega, 120 minutos

1 Engenharia de Requisitos [120 min]

Palavras-chave

critérios de aceite, épico, história do usuário

Objetivos de aprendizagem

ATT-1.x (K1) Palavras-chave

1.1 Técnicas de engenharia de requisitos

ATT-1.1.1-1 (K4): Analisar as histórias de usuários e épicos usando técnicas de engenharia de requisitos.

ATT-1.1.1-2 (K2): Descrever as técnicas de engenharia de requisitos e como elas podem ajudar os testadores.

ATT-1.1.2-1 (K4): Criar e avaliar os critérios de aceite testáveis para uma determinada história de usuário usando técnicas de engenharia e teste de requisitos.

ATT-1.1.2-2 (K2): Descrever as técnicas de elicitação.

1.1 Técnicas de engenharia de requisitos

A aplicação de técnicas de engenharia de requisitos permite que as equipes ágeis aprimorem as histórias de usuário e os épicos, adicionem contexto, considerem os impactos e as dependências, e identifiquem quaisquer lacunas como a falta de requisitos não funcionais. [ISTQB_FL_AT]

Embora a maioria das técnicas de engenharia de requisitos discutidas nesta seção sejam provenientes de abordagens tradicionais de desenvolvimento, elas também são eficazes no desenvolvimento ágil.

Geralmente nos projetos tradicionais, as atividades e técnicas de engenharia de requisitos são formalizadas, executadas sequencialmente, e de responsabilidade dos designados como Analistas de Negócios, Analistas Funcionais, Arquitetos Técnicos, Arquitetos Corporativos e Analistas de Processos. Por outro lado, nos projetos ágeis, as técnicas de engenharia de requisitos são aplicadas em todo o projeto e durante cada iteração por meio de uma abordagem menos formal. Essas técnicas de engenharia de requisitos são realizadas com mais frequência, usando ciclos de feedback contínuo, com a participação de todos os membros da equipe do Ágil, e não do Analista de Negócio ou do proprietário do produto (product owner) da equipe.

1.1.1 Analisar as histórias de usuários e épicos usando técnicas de engenharia de requisitos

O testador precisa conhecer, entender, selecionar e utilizar as várias técnicas de engenharia de requisitos para poder esclarecer (e possivelmente melhorar) as histórias, épicos e outros requisitos ágeis.

Exemplos dessas técnicas são storyboards, mapeamento de histórias, personas, diagramas e casos de uso.

Storyboards

Um storyboard (que não deve ser confundido com o painel de tarefas do Ágil ou o storyboard do usuário do Ágil) fornece uma representação visual do sistema. Os storyboards ajudam os testadores a:

- Observar o raciocínio por trás das histórias de usuário e a “história” sob uma visão geral, fornecendo contexto, tornando possível analisar rapidamente o fluxo funcional do sistema e identificar quaisquer lacunas na lógica.
- Visualizar os grupos de histórias de usuários relacionados a uma área comum do sistema (temas) que podem ser considerados para inclusão na mesma iteração, pois provavelmente utilizarão o mesmo trecho de código.
- Auxiliar no mapeamento de histórias e priorizar os épicos e histórias de usuários relacionados no backlog do produto.
- Auxiliar na identificação de critérios de aceite para histórias e épicos de usuários.
- Auxiliar na seleção da abordagem de teste correta, baseando-se nos aspectos visuais do desenho do sistema.
- Juntamente com o Mapeamento de História, ajuda na priorização dos testes e na identificação da necessidade de simuladores, drivers ou emuladores (mocks).

Mapeamento de Histórias

O Mapeamento de Histórias (ou Mapeamento de Histórias de usuário) é uma técnica que consiste em usar duas dimensões independentes para organizar as histórias de usuário. O eixo horizontal do mapa representa a ordem de prioridade de cada uma das histórias de usuário, enquanto o eixo vertical representa a sofisticação da implementação. O uso do Mapeamento de História pode ajudar os testadores a:

- Determinar as funcionalidades mais básicas de um sistema para derivar o smoke *teste*;
- Identificar a ordem das funcionalidades para determinar as prioridades dos testes.
- Visualizar o escopo do sistema;
- Determinar o nível de risco de cada história de usuário.

Personas

Personas são usadas para definir personagens fictícios ou arquétipos que ilustram como os usuários típicos irão interagir com o sistema. O uso de personas pode ajudar os testadores a:

- Identificar as lacunas nas histórias de usuários, identificando os diferentes tipos de usuários que podem usar o sistema;
- Identificar as inconsistências nas histórias de usuários sobre como um determinado tipo de usuário pode usar o sistema em comparação com outros;
- Elicitar os critérios de aceite das histórias de usuários;
- Descobrir caminhos adicionais de teste durante os testes exploratórios;
- Revelar as condições de teste, especialmente aquelas relacionadas aos grupos específicos de usuários, ajudando assim a garantir uma cobertura suficiente do grupo e testar as diferenças entre os grupos de usuários.

Diagramas

Diagramas como os de Entidade Relacionamento, diagramas de classe, diagramas UML entre outros, podem mostrar a estrutura ou o fluxo dos dados; e os atributos funcionais ou o comportamento do sistema, podem ser usados para identificar as lacunas na funcionalidade do sistema.

Casos de uso

Casos de uso (diagramas e especificações) [ISTQB_FL] podem ajudar os testadores a:

- Garantir que as histórias de usuários sejam testáveis e dimensionadas adequadamente;
- Determinar se as histórias de usuários precisam ser refinadas ou decompostas;
- Revelar *stakeholders* esquecidos;
- Identificar interfaces e pontos de integração, que devem ser considerados durante a modelagem do teste;
- Visualizar os relacionamentos entre épicos e histórias de usuários, para verificar se o épico não possui histórias de usuários ausentes.

1.1.2 Identificar os critérios de aceite usando a engenharia de requisitos e as técnicas de teste

A engenharia de requisitos é um processo que consiste nas seguintes etapas:

Elicitação

Processo de descobrir, entender, documentar e revisar as necessidades e restrições dos usuários para o sistema. As técnicas de elicitação devem ser usadas para derivar, avaliar e aprimorar os critérios de aceite.

Documentação

Processo de registrar as necessidades e restrições dos usuários de forma clara e precisa. As histórias de usuários e os critérios de aceite devem ser documentados e alinhados ao nível de adesão da equipe aos princípios do Manifesto Ágil. O tipo de documentação depende da abordagem da equipe e dos *stakeholders*. Os critérios de aceite podem ser documentados usando linguagem natural, usando modelos (p.e., diagramas de transição de estado) ou usando exemplos.

Negociação e Validação

Para cada história de usuário, vários *stakeholders* podem ter outras percepções ou preferências. Como essas percepções e preferências podem ser inconsistentes ou até conflitantes, também podem ser os critérios de aceite de cada *stakeholder*. Cada um desses conflitos deve ser identificado, negociado e resolvido entre todos os envolvidos. Todo conflito esquecido ou não resolvido pode comprometer o sucesso do projeto. No final desta etapa, o conteúdo de cada história de usuário é validado pelos seus respectivos *stakeholders* (p.e., como uma definição de feito).

Gestão

À medida que os projetos evoluem, as opiniões e circunstâncias podem mudar. Embora os critérios de aceite tenham sido adequadamente elicitados, documentados, negociados e validados, os critérios de aceite ainda estão sujeitos a alterações. As histórias de usuários devem ser gerenciadas usando bons processos de configuração e gerenciamento de alterações porque elas são potencialmente expostas às mudanças.

Para identificar os critérios de aceite, várias técnicas de elicitação estão à disposição do testador, incluindo:

Questionários quantitativos

Usar dados quantitativos retirados de perguntas fechadas é uma excelente maneira de fazer comparações claras entre vários pontos de dados. Isso geralmente fornece dados numéricos que podem ser incluídos em uma conclusão numérica para um critério de aceite. O questionário quantitativo pode ser usado como uma técnica de elicitação para um grande número de stakeholders e, especificamente, para os critérios de aceite não-funcionais.

Questionários qualitativos

As perguntas abertas são uma maneira extremamente eficaz de adicionar mais qualidade à pesquisa quantitativa. As perguntas abertas são mais bem utilizadas como acompanhamento das principais perguntas. Isso pode gerar informações adicionais para as quais novas histórias de usuário precisam ser criadas ou adicionadas às existentes. O questionário qualitativo pode ser usado como uma técnica de elicitação para um menor número de *stakeholders* - à medida que o processamento leva mais tempo - e é adequado para critérios de aceite funcional.

Entrevista qualitativa:

A entrevista qualitativa é mais flexível que uma consulta quantitativa e é usada principalmente para obter informações sobre antecedentes, contextos e causas. É improvável retornar dados concretos, mas os critérios de aceite podem ser derivados das respostas relacionadas ao contexto de uma história de usuário. A entrevista qualitativa pode ser o complemento de qualquer tipo de questionário, para aprofundar os critérios de aceite derivados.

Existem várias outras técnicas de elicitação, no conjunto de técnicas de observação (p.ex., aprendizagem), técnicas de criatividade (p.ex., 6 Chapéus de Pensamento) e técnicas de apoio (p.ex., prototipagem baixa fidelidade). O conjunto de ferramentas técnicas do testador influenciará a qualidade dos critérios de aceite elicitados.

As técnicas INVEST e SMART [INVEST] também podem ser usadas para identificar e avaliar os critérios de aceite, juntamente com técnicas de teste como particionamento de equivalência, análise de valor de limite, tabelas de decisão e teste de transição de estado [ISTQB_FL].

2 Teste no Ágil [180 min]

Palavras-chave

Desenvolvimento orientado por teste, desenvolvimento orientado pelo comportamento, desenvolvimento orientado por teste de aceite, especificação por exemplo, regulamento de teste.

Objetivos de aprendizado

ATT-2.x (K1) Palavras-chave

2.1 Desenvolvimento ágil e técnicas de teste

ATT-2.1.1-1 (K3) Aplicar TDD (desenvolvimento orientado por teste) no contexto de um determinado exemplo em um projeto ágil.

ATT-2.1.1-2 (K2) Entender as características de um teste de unidade.

ATT-2.1.1-3 (K2) Entender o significado do acrônimo FIRST.

ATT-2.1.2-1 (K3) Aplicar desenvolvimento orientado pelo comportamento (BDD) no contexto de uma determinada história de usuário em um projeto ágil.

ATT-2.1.2-2 (K2) Entender como gerenciar diretrizes para a formulação de um cenário.

ATT-2.1.3 (K4) Analisar uma lista de pendências de produtos em um projeto ágil para determinar uma maneira de introduzir o desenvolvimento orientado por testes de aceite (ATDD).

2.2 Teste baseado na experiência no Ágil

ATT-2.2.1-1 (K4) Analisar a criação de uma abordagem de teste usando automação de teste, testes baseados em experiência e testes caixa-preta, criados usando outras abordagens (incluindo testes baseados em risco) para um determinado cenário em um projeto ágil.

ATT-2.2.1-2 (K2) Explicar as diferenças entre Missão Crítica e Não Crítica.

ATT-2.2.2-1 (K4) Analisar as histórias e épicos de usuários para criar cartas de teste.

ATT-2.2.2-2 (K2) Compreender o uso de técnicas baseadas na experiência.

2.3 Aspectos da qualidade do código.

ATT-2.3.1-1 (K2) Entender a importância da refatoração de casos de teste em projetos ágeis.

ATT-2.3.1-2 (K2) Compreender a lista de tarefas práticas para refatoração de casos de teste.

ATT-2.3.2-1 (K4) Analisar o código como parte de uma revisão de código, para identificar os defeitos e débitos técnicos.

ATT-2.3.2-2 (K2) Compreender a análise de código estática.

2.1 Desenvolvimento ágil e técnicas de teste

No desenvolvimento de software, podem ocorrer defeitos em código mal escrito e falhas em atender às necessidades do cliente. As técnicas de desenvolvimento ágil lidam com esses problemas aplicando os conceitos de desenvolvimento orientado por teste (TDD), desenvolvimento orientado pelo comportamento (BDD) e desenvolvimento orientado por teste de aceite (ATDD). O TDD é uma técnica para melhorar a qualidade do produto do software, enquanto o BDD e o ATDD ajudam a melhorar sua qualidade de utilização (características e funcionalidade).

2.1.1 Desenvolvimento orientado por teste (TDD)

O desenvolvimento orientado por teste é um método de desenvolvimento de software que combina modelagem, teste e codificação em um processo iterativo rápido. O TDD adota uma abordagem disciplinada de testar primeiro (*test first*), para criar um teste que expressa a funcionalidade pretendida do código antes de ser codificada. O teste é executado antes que o próprio código seja escrito, para verificar se o teste falha. Depois que o código é escrito, o teste é executado novamente e deve passar.

O TDD é geralmente considerado a primeira e a principal metodologia de programação de teste a partir da qual derivam as demais metodologias, como desenvolvimento orientado pelo comportamento (BDD), desenvolvimento orientado por teste de aceite (ATDD) e especificação por exemplo (SBE).

O TDD oferece uma abordagem evolutiva ao desenvolvimento de software que trata o desenho como um processo contínuo. Cada iteração constitui uma pequena alteração no código de produção e oferece uma oportunidade para o desenvolvedor melhorar um pouco o seu desenho. À medida que as mudanças e as decisões de projeto cuidadosamente consideradas se acumulam passo a passo, surge um projeto robusto e bem testado.

Os profissionais de TDD usam várias práticas e técnicas para escrever códigos de alta qualidade e evitar o acúmulo de débito técnico, incluindo:

- Testes de unidade e uma abordagem prescritiva de testar primeiro;
- Ciclos iterativos curtos;
- Feedback imediato e frequente;
- Uso eficaz de ferramentas, controle de fontes, e integração contínua;
- A aplicação orientada de princípios de programação e padrões de modelagem.

Os praticantes de TDD escrevem testes de unidade para verificar o comportamento esperado do código em produção. Um teste de unidade executa uma função sob condições específicas e verifica se produz um resultado esperado ou não. As expectativas são descritas em declarações que verificam se o estado do sistema muda conforme o esperado ou se um comportamento específico é exibido. As declarações podem verificar, por exemplo:

- Que uma função executa um cálculo e retorna um resultado esperado;
- Que uma função modifica o estado do sistema de uma maneira específica;
- Que uma função chama outra função de uma maneira específica.

Os testes de unidade devem ser fáceis para os desenvolvedores implementarem e manterem. Eles são automatizados e geralmente são escritos usando a mesma linguagem que o código em produção. Os testes de unidade devem ter as seguintes características:

- **Determinístico**: toda vez que um teste de unidade é executado nas mesmas condições, ele deve produzir os mesmos resultados.
- **Atômico**: um teste de unidade deve testar apenas a funcionalidade relacionada a ele.
- **Isolado:** um teste de unidade deve se esforçar para exercitar apenas o código específico para o qual foi inicialmente planejado. Um teste de unidade não deve depender de outro e deve evitar dependências no código em produção sempre que possível.
- **Rápido:** um teste de unidade deve ser pequeno e rápido para fornecer feedback imediato. Deve ser possível executar muitos testes de unidade em um curto período.

Outro acrônimo para descrever um bom teste de unidade é o FIRST: Rápido (Fast), Isolado (Isolated), Reproduzível (Repeatable), Autovalidado (Self-validating), Completo (Thorough).

Existem muitas estruturas de teste de unidade disponíveis para muitas linguagens diferentes. Eles diferem em suas APIs, abordagens e terminologia, mas todos compartilham de alguns elementos comuns. Independentemente da linguagem ou estrutura de teste de unidade, ele normalmente segue um padrão de três etapas:

1. **Organizar/Configurar**: preparar o ambiente de execução. Esta etapa pode instanciar objetos, inicializar o estado do sistema, e injetar ou inserir dados conforme necessário.
2. **Atuar:** executar a operação que está sendo testada e verificar o resultado produzido pela operação. Essa etapa é obrigatória e pode consistir em apenas uma linha de código que aciona a operação que está sendo testada.
3. **Afirmar**: realizar a verificação real das saídas esperadas ou outras pós-condições.

O processo iterativo é a pedra angular do TDD. O objetivo de cada iteração é fazer uma alteração pequena, focada e cuidadosamente considerada no programa. Seguindo o padrão de codificação por cores, o ciclo iterativo é frequentemente chamado de ciclo vermelho-verde-refatoração:

- **Vermelho**: escrever um teste com falha que descreva uma expectativa não implementada. Executar o teste para garantir que ele falhe.
- **Verde**: escrever um código de produção que satisfaça apenas a expectativa descrita pelo teste e fazer com que seja aprovado. Executar todos os testes relevantes para garantir que todos sejam aprovados. Se houver falha, fazer as alterações necessárias para que todos os testes passem.
- **Refatorar**: Melhorar o desenho e a estrutura do código de teste e do código em produção sem alterar a funcionalidade, garantindo que todos os testes relevantes continuem a passar. Os desenvolvedores podem aplicar uma sequência de refatoração para alterar o código sem alterar o comportamento até que o código seja otimizado. Os mais modernos ambientes de desenvolvimento integrado (IDE) e muitos editores de código podem aplicar a refatoração de forma automática e segura.

2.1.2 Desenvolvimento orientado pelo comportamento (BDD)

De acordo com o syllabus do ISTQB® Foundation Agile Tester [ISTQB_FL_AT], o BDD é uma técnica na qual os desenvolvedores, os testadores e os representantes do negócio trabalham juntos para analisar os requisitos de um sistema de software, desenvolvê-los usando uma linguagem compartilhada e verificá-los automaticamente.

O BDD é fortemente influenciado por duas disciplinas distintas: desenvolvimento orientado por teste (TDD) e desenvolvimento orientado pelo domínio (DDD) [Evans03]. Ele incorpora muitos dos princípios básicos de ambas as disciplinas, incluindo colaboração, desenho, linguagem universal, execução automatizada de testes, ciclos curtos de feedback e muito mais. O TDD conta com testes de unidade para verificar detalhes da implementação, enquanto o BDD conta com cenários executáveis para verificar os comportamentos e muito mais. O BDD geralmente segue passos pré-determinados:

- Criar histórias de usuário colaborativamente;
- Formular histórias de usuários como cenários executáveis e comportamentos verificáveis;
- Implementar comportamentos e executar os cenários para verificá-los.

As equipes que aplicam o BDD extraem um ou mais cenários de cada história do usuário e os formulam como testes automatizados. Um cenário representa um único comportamento sob condições específicas.

Os cenários geralmente são baseados nos critérios de aceite, exemplos e casos de uso da história do usuário. Os cenários do BDD devem ser escritos usando uma linguagem que possa ser entendida por todos os membros da equipe, técnicos ou não técnicos.

Portanto, é dada uma forte preferência ao uso de linguagem natural, como a língua portuguesa, para expressar cenários. Além disso, as equipes que aplicam o BDD estabelecem uma linguagem universal [Evans03], que essencialmente constitui uma terminologia clara e inequívoca para todos na equipe e é usada em todos os lugares.

Os cenários do BDD são tipicamente compostos por três seções principais [Gherkin]:

1. **Given** (dado): descreve o estado do ambiente (pré-condições) antes que o comportamento seja acionado.
2. **When** (quando): descreve as ações que acionam o comportamento.
3. **Then** (então): descreve os resultados esperados do comportamento.

Extrair cenários de histórias de usuários envolve:

- Identificar todos os critérios de aceite especificados por uma história de usuário e escrever os cenários para cada um deles. Alguns podem exigir vários cenários.

- Identificar os casos de uso e exemplos funcionais e escrever os cenários para cada um deles. Muitos casos de uso e exemplos são condicionais, por isso é crucial identificar cada uma das condições.
- Criar cenários durante o teste exploratório, o que pode ajudar a identificar relacionamento entre os comportamentos existentes, os comportamentos em potencial conflito, os estados mínimos, os fluxos alternativos etc.
- Procurar o uso repetido de etapas ou grupos de etapas para evitar trabalho repetitivo.
- Identificar as áreas que requerem dados aleatórios ou sintéticos.
- Identificar as etapas que exigem emuladores, simuladores ou *drivers* para manter o isolamento e evitar a execução de integrações ou processos possivelmente dispendiosos.
- Garantir que os cenários sejam atômicos e não afetem o estado um do outro (isolado).
- Decidir se deve limitar as seções “When” a uma etapa, de acordo com o princípio de que todos os testes verificam apenas uma coisa ou otimizar por outras considerações, como velocidade de execução do teste.

As diretrizes recomendadas para a elaboração de cenários incluem:

- O cenário deve descrever um comportamento específico que o sistema suporta a partir da perspectiva de um usuário específico.
- O cenário deve usar a terceira pessoa ao descrever as etapas (Give, When e Then) para descrever o estado e as interações a partir da perspectiva do usuário.
- Os cenários devem ser isolados e atômicos para que possam ser executados em qualquer ordem e não afetem ou dependam um do outro. Os passos Given devem colocar o sistema no estado necessário para que os passos *When* sejam executados de forma consistente, como esperado.
- Os passos *When* devem descrever as ações semânticas que um usuário executa, em vez das ações técnicas específicas, a menos que haja uma necessidade específica de testar uma ação específica. Por exemplo, “O usuário confirma o pedido” (uma ação semântica) geralmente é preferível a “O usuário clica no botão Confirmar” (uma ação técnica), a menos que o próprio botão precise ser testado.
- Os passos Then devem descrever observações ou estados específicos. Eles não devem especificar estados genéricos de sucesso ou erro.

2.1.3 Desenvolvimento orientado por testes de aceite (ATDD)

O ATDD suporta a coordenação de projetos de software para atender à entrega com base nas necessidades do cliente. Testes de aceite são especificações do comportamento e funcionalidade desejados de um sistema. Uma história de usuário representa uma parte da funcionalidade valorizada pelo cliente. Os testes de aceite verificam se essa funcionalidade está implementada corretamente. Essa história do usuário é dividida pelos desenvolvedores em um conjunto de tarefas necessárias para implementar essa funcionalidade. Os desenvolvedores podem implementar essas tarefas usando TDD. Quando uma determinada tarefa é concluída, os desenvolvedores passam para a próxima tarefa, e assim por diante, até que a história do usuário seja concluída, o que é indicado por testes de aceite executados com sucesso. Tanto o BDD quanto o ATDD são focados no cliente, enquanto o TDD é focado no desenvolvedor. O BDD e o ATDD são semelhantes, pois ambos produzem o mesmo resultado: um entendimento compartilhado do que deve ser construído e como construir a coisa certa. O BDD é uma maneira estruturada de escrever casos de teste (p.ex., testes de aceite) usando a sintaxe Given/When/Then discutida anteriormente.

O ATDD separa a intenção do teste, que é expressa em um formato legível por humanos como um texto sem formatação, da implementação do teste, que é preferencialmente automatizada. Essa importante separação significa que os membros da equipe que estão lidando com o negócio e outros não técnicos, como proprietários de produtos (product owner), analistas e testadores, podem desempenhar um papel ativo em redigir exemplos testáveis (ou testes de aceite) para direcionar o desenvolvimento. Os stakeholders com diferentes papéis e perspectivas, como cliente, desenvolvedor e testador, se reúnem para discutir de forma colaborativa um potencial história do usuário (ou outra forma de requisito), a fim de alcançar um entendimento compartilhado do problema a ser resolvido e fazer perguntas abertas sobre a funcionalidade e explorar exemplos concretos e testáveis do comportamento necessário.

Os exemplos acordados (e as discussões que os levam a eles) são resultados valiosos em si mesmos; portanto, é importante lembrar que, embora eles também possam ser automatizados como testes de aceite, nem sempre é necessário ou econômico fazê-lo. Essa ênfase na ideia de exemplos valiosos, em vez de testes, é intencional e é um truque importante para incentivar todos os membros de uma equipe a se envolverem no processo de descoberta.

Isso ilustra um papel importante para o testador ágil nessa situação. Durante as discussões, o testador pode pensar em termos de técnicas de teste, como particionamento de equivalência, análise de valor de limite e assim por diante. Eles podem usar essa abordagem analítica para incentivar perguntas a serem feitas ao proprietário do produto

(product owner) sobre onde os comportamentos interessantes e casos extremos podem ser encontrados. A intenção deve ser sempre a de incentivar todo o grupo a considerar e encontrar os exemplos principais. Apresentar uma combinação interessante de entradas e perguntar se o grupo está de acordo sobre os resultados esperados é uma boa maneira de explorar as áreas potenciais de incerteza ou de análise incompleta.

A especificação por exemplo (ou SBE) refere-se a uma coleção de padrões úteis que permitem que uma equipe do Ágil descubra, discuta e confirme os resultados importantes exigidos pelos stakeholders nos negócios e os comportamentos de softwares necessários para alcançar esses resultados. O termo tem sido amplamente usado para incluir e ampliar o significado do termo desenvolvimento orientado por teste de aceite (ATDD).

2.2 Teste baseado na experiência no Ágil

As várias características dos projetos ágeis, como abordagem, duração da iteração, nível de teste aplicável, nível de risco do projeto e produto, qualidade dos requisitos, nível de experiência e conhecimento dos membros da equipe, organização do projeto etc., influenciarão o equilíbrio de testes automatizados, testes exploratórios e manuais caixa-preta em um projeto ágil.

2.2.1 Combinando as técnicas baseadas na experiência e testes caixa-preta

Durante a análise de risco, os níveis de risco (p. ex., alto, médio, baixo) são determinados para os recursos e funcionalidades específicas do sistema. A próxima etapa é encontrar a combinação e o equilíbrio de testes automatizados, testes exploratórios e testes manuais caixa-preta para um nível de risco específico. Abaixo está uma tabela que descreve essa ideia. Nesta tabela, os níveis de risco são listados verticalmente e os três testes são abordados horizontalmente. Os seguintes termos são usados para descrever:

altamente recomendado / recomendado / indiferente / não recomendado / não usar

A tabela abaixo é um exemplo de uma combinação de diferentes técnicas de teste (exploratória, automatizada e manual) que podem ser usadas em um sistema crítico de segurança. Esta tabela pode ser adaptada a outros projetos específicos.

Nível de Risco	Testes Automatizados	Testes Exploratórios	Testes Caixa-Preta
ALTO	altamente recomendado	recomendado	altamente recomendado
MÉDIO	recomendado	recomendado	recomendado
BAIXO	indiferente	altamente recomendado	recomendado

Tabela 1 - Sistemas de Missão/Segurança Crítica

Examinando a situação da primeira linha da Tabela 1, uma combinação de teste automatizado e teste caixa-preta é sugerido como altamente recomendado, além de uma abordagem de teste exploratório. A decisão de automatizar (ou não) também será influenciada por muitos outros fatores.

A seguir temos um exemplo de uma combinação de diferentes técnicas de teste quando utilizadas em um sistema que não seja de segurança crítica.

Nível de Risco	Testes Automatizados	Testes Exploratórios	Testes Caixa-Preta
ALTO	recomendado	altamente recomendado	recomendado
MÉDIO	indiferente	altamente recomendado	indiferente
BAIXO	não usar	altamente recomendado	não usar

Tabela 2: Sistema que não são de missão e/ou segurança crítica.

Examinando a situação da última linha da Tabela 2, uma abordagem de teste exploratório é altamente recomendada, enquanto outras podem não ser usadas.

Em qualquer situação (crítica ou não à segurança), a combinação dependerá das características do projeto.

2.2.2 Criando cartas de teste e interpretando seus resultados

Antes que uma determinada carta de teste possa ser criada, os épicos e histórias de usuários existentes devem ser avaliados primeiro. Veja o capítulo 1 para técnicas.

Ao analisar épicos ou histórias de usuários para criar uma carta de teste, os seguintes itens devem ser considerados:

- Quem são os usuários neste épico ou história de usuário?
- Qual é a principal funcionalidade do épico ou da história de usuário?
- Quais são as ações que um usuário pode executar? (Isso pode ser obtido na lista de critérios de aceite, definida para uma história de usuário.)
- O objetivo da história do usuário é satisfeito quando o recurso ou funcionalidade é concluído? (Ou existem outras tarefas de teste que afetam a definição de concluído.)

A granularidade de uma carta de teste é importante. Não deve ser muito pequeno, pois deve explorar uma área em torno de um problema identificado (reativo, regressão) ou uma área em torno de uma história de usuário ou um épico (proativa, reveladora).

Ela não deve ser muito grande, pois deve caber dentro de um período de execução de 60 a 120 minutos. O objetivo de executar uma sessão de teste exploratório é ajudar na tomada de decisão de uma boa qualidade em relação a um fenômeno, agrupamento de bugs etc. Os resultados dessa exploração devem fornecer informações suficientes para tomar essa decisão.

As cartas de teste podem ser criadas usando flipcharts, planilhas, documentos, sistemas existentes de gerenciamento de testes, personas, mapa mental e uma abordagem de equipe inteira. Os testadores exploratórios usam heurísticas para direcionar sua criatividade ao escrever e executar sessões de teste exploratório. Eles também pensam de forma criativa ao analisar histórias e épicos de usuários, podendo ser usados para criar as cartas de teste. Alguns exemplos de heurísticas podem ser encontrados em [Whittaker09] e [Hendrickson13].

Todas as descobertas encontradas durante os testes exploratórios devem ser documentadas. Os resultados dos testes exploratórios devem fornecer informações sobre uma melhor modelagem de teste, ideias para testar o produto e ideias para qualquer melhoria adicional. As descobertas que devem ser documentadas durante os testes exploratórios incluem defeitos, ideias, perguntas, sugestões de melhoria etc.

Ferramentas podem ser usadas para documentar as sessões de teste exploratório. Isso inclui ferramentas de captura e registro de tela, ferramentas de planejamento etc. A documentação deve incluir o resultado esperado. Em alguns casos, caneta e papel são suficientes, dependendo do volume de informações a serem coletadas.

Ao resumir a sessão de teste exploratório, durante a reunião de balanço, as informações são coletadas e agregadas para apresentar um status de progresso, cobertura e eficiência da sessão. Esse resumo de informações pode ser usado como um relatório de gerenciamento ou em reuniões de retrospectivas em qualquer nível e escala (equipe única, várias equipes e implementação ágil em larga escala). No entanto, pode ser bastante desafiador determinar as métricas precisas de teste relacionadas a sessões de teste exploratórias.

2.3 Aspectos da qualidade do código

O controle do débito técnico é muito importante nos projetos ágeis, especialmente em termos de manutenção de altos níveis de qualidade do código durante o lançamento (*release*). Várias técnicas são usadas para atingir esse objetivo.

2.3.1 Refatoração

A refatoração é uma maneira de limpar o código de maneira eficiente e controlada, esclarecendo e simplificando o desenho do código e dos casos de teste existentes, sem alterar seu comportamento. Nos projetos ágeis, as iterações são curtas, o que cria um curto ciclo de feedback para todos os membros da equipe. Iterações curtas também criam um desafio para os testadores que tentam obter uma cobertura adequada. Devido à natureza das iterações, o fato de que a funcionalidade está aumentando e os recursos são adicionados e aprimorados ao longo do tempo, os casos de teste que foram escritos para um recurso em uma iteração anterior, geralmente precisam de manutenção ou até de um novo desenho completo nas iterações posteriores. Usando uma abordagem evolutiva da modelagem de teste, a atualização e refatoração dos testes podem compensar as alterações das funcionalidades e garantir que os testes permaneçam alinhados com a funcionalidade do produto.

Depois que as histórias de usuário são entendidas e os critérios de aceite são escritos para cada uma delas, o impacto da funcionalidade da iteração atual nos testes de regressão existentes (manual e automatizado) pode ser analisado, podendo ser necessária a refatoração ou aprimoramento dos testes. As equipes estão mantendo e estendendo seu código extensivamente de iteração para iteração, e sem a refatoração contínua será difícil a implementação.

A refatoração dos casos de teste pode ser feita da seguinte maneira:

- **Identificação**: identificar os testes existentes que requerem refatoração por revisão ou análise causal.
- **Análise**: analisar o impacto das alterações dos testes no conjunto geral de testes de regressão.
- **Refatoração**: alterar a estrutura interna dos testes para facilitar a compreensão e baratear a modificação sem alterar o seu comportamento observável.
- **Execução**: reexecutar os testes, verificando seus resultados e documentando os defeitos quando relevante. A refatoração não deve influenciar o resultado da execução do teste.
- **Avaliação**: verificar os resultados da repetição dos testes, encerrando esta fase quando o limite de qualidade definido e aceito pela equipe for ultrapassado.

2.3.2 Revisões de código e análise de código estático para identificar defeitos e débito técnico

Uma revisão de código é um exame sistemático do código por duas ou mais pessoas (uma das quais geralmente é o autor). A análise de código estático é o exame sistemático do código por uma ferramenta. Ambas são práticas eficazes e difundidas para expor e identificar problemas que afetam a qualidade do código. As revisões de código e a análise de código estático fornecem feedback construtivo que ajuda a identificar os defeitos e gerenciar débitos técnicos.

Impedimentos, assim como restrições de recursos, maior complexidade técnica do que o esperado, rápida mudança de prioridades e limitações técnicas podem dificultar os esforços para escrever um código de qualidade e forçar os programadores a fazer concessões que diminuirão a qualidade do código em favor de resultados mais imediatos. Essas concessões podem introduzir defeitos e incorrer em débitos técnicos.

O débito técnico refere-se ao aumento do esforço necessário para implementar uma solução futura melhor (incluindo-se a remoção de defeitos que estejam latentes no código), devido a consequência da escolha de uma solução inferior e mais fácil de implementar agora. O débito técnico é muitas vezes incorrido involuntariamente, por concessões sutis ou pelo acúmulo gradual de pequenas ou despercebidas mudanças à medida que o software evolui. As revisões de código e a análise de código estático ajudam a identificar diferentes causas de débito técnico, como aumento na complexidade, dependências circulares, conflitos entre diferentes módulos de código, baixa cobertura de código, código inseguro e assim por diante. Outros tipos de débitos técnicos também podem ocorrer, por exemplo, em artefatos de teste, infraestrutura e o pipeline de integração contínua.

Quando o débito técnico é incorrido deliberadamente (como consequência de outras decisões ou concessões) ou identificado por revisões de código ou análise estática, deve-se fazer um esforço para reduzi-lo. É preferível lidar com isso imediatamente. Se não puder ser tratado imediatamente, as tarefas para lidar com o débito técnico devem ser adicionadas ao *backlog* (produto).

A relação custo x benefício entre o tempo adicional necessário para analisar e revisar o código, e incorrer em débito técnico, quase sempre favorece a análise e as revisões de código. Quando o código com defeito e débito técnico se espalha, torna-se cada vez mais difícil, oneroso e demorado corrigi-lo sem prejudicar outras partes do sistema. A análise e as revisões de código podem melhorar a qualidade do código e provavelmente reduzir o tempo total gasto.

Além de ajudar a identificar os defeitos e gerenciar débitos técnicos, a análise e as revisões de código oferecem benefícios adicionais:

- Treinamento e compartilhamento de conhecimento;
- Melhoria na robustez, na capacidade de manutenção e na legibilidade do código;
- Fornecimento de supervisão e manutenção dos padrões consistentes de codificação.

Revisões de código

Ao participar das revisões de código, os testadores podem usar sua perspectiva diferenciada para contribuir valiosamente na qualidade do código, trabalhando em conjunto com os programadores para identificar possíveis defeitos e evitar débitos técnicos em um estágio muito inicial. Os testadores devem ser competentes para compreender a linguagem de programação usada no código analisado, mas não precisam ter habilidades significativas de codificação para participar efetivamente das análises de código. Eles podem trazer seus conhecimentos de várias maneiras, como questionar sobre o comportamento do código, sugerir casos de uso que podem não ter sido considerados ou monitorar as métricas de código que possam indicar problemas de qualidade. As revisões de código também oferecem uma oportunidade de compartilhar conhecimento entre programadores e testadores. Um dos principais desafios das revisões de código em projetos ágeis são as curtas iterações e o tempo

necessário para realizar as revisões de código. É importante planejar revisões de código, reservando o tempo necessário para revisá-las durante cada iteração.

As revisões de código são atividades manuais, possivelmente suportadas por ferramentas, e executadas por ou com outras pessoas (além do autor). Normalmente, líderes de equipe ou programadores mais experientes da equipe farão a revisão do código, embora também possam ser feitos com outros membros da equipe. Muitas vezes, é benéfico para testadores e outros não programadores participarem das revisões de código.

Diferentes abordagens para revisões de código variam em seu nível de formalidade e rigor [Wiegers02]. As abordagens mais formais e rigorosas tendem a ser mais completas, mas também consomem mais tempo. As abordagens menos formais e rigorosas são menos completas, mas podem ser muito mais rápidas. Existem diferentes tipos de revisões por pares, e as equipes ágeis tendem a preferir as mais rápidas, realizadas com frequência, e geralmente antes da integração.

As revisões de código podem ser realizadas com o revisor e o autor (desenvolvedor) sentados lado a lado. Esta modalidade, comum em revisões ad-hoc e programação em pares, facilita uma excelente comunicação e encoraja as análises mais profundas e um melhor compartilhamento de conhecimentos. Também pode contribuir para a coesão e moral da equipe.

Em equipes distribuídas ou em equipes que preferem uma abordagem mais desconectada, o processo de revisão de código é facilitado pelo sistema de gerenciamento de configuração. O processo geralmente é parcialmente automatizado como parte do processo de integração contínua. Esses processos podem suportar revisões de código com um único revisor em cada rodada de revisão ou revisões de equipe colaborativa.

Análise de código estático

Na análise de código estático, uma ferramenta analisa o código e procura por problemas específicos sem executá-lo. Os resultados da análise estática do código podem apontar problemas claros no código ou fornecer indicadores indiretos que requerem avaliação adicional.

Muitas ferramentas de desenvolvimento, especialmente as de ambientes de desenvolvimento integrado (IDEs), podem executar análises estáticas de código enquanto escrevem o código. Isso fornece o benefício de *feedback* imediato, embora seja possível apenas aplicar um subconjunto da análise realizada durante a integração contínua.

3 Automação de teste [180 min]

Palavras-chave

teste orientado por dados, teste orientado por palavras-chave, procedimento de teste, abordagem de teste

Objetivos de aprendizagem para o processo de teste

ATT-3.x (K1) Palavras-chave

3.1 Técnicas de automação de teste

ATT-3.1.1 (K3) Aplicar as técnicas de teste orientadas por dados e palavras-chave para desenvolver scripts de teste automatizados.

ATT-3.1.2 (K2) Entender como aplicar a automação de teste a uma determinada abordagem de teste em um ambiente ágil.

ATT-3.1.3-1 (K2) Entender a automação de teste.

ATT-3.1.3-2 (K2) Entender as diferenças entre as várias abordagens de teste.

3.2 Nível de automação

ATT-3.2.1-1 (K2) Entender os fatores a serem considerados ao determinar o nível de automação de teste necessário para acompanhar a velocidade da implantação.

ATT-3.2.1-2 (K2) Entender os desafios da automação de testes em configurações ágeis.

3.1 Técnicas de automação de teste

3.1.1 Teste orientado por dados

Motivação

O teste orientado por dados é uma técnica de automação de teste que minimiza os esforços necessários para desenvolver e manter casos de teste que possuem etapas de teste idênticas, mas que possuem combinações diferentes de entradas de dados. O teste orientado por dados é uma técnica bem estabelecida que não é específica para testes em projetos ágeis devido à sua capacidade de reduzir os esforços de desenvolvimento e manutenção da automação de testes, essa técnica deve ser considerada como parte da estratégia de automação de teste em todos os projetos ágeis.

Conceito

A ideia básica do teste orientado por dados é separar a lógica do teste dos dados de teste. Em vez disso, o procedimento de teste nomeia variáveis de dados de teste que referenciam valores de dados de teste que são fornecidos em uma lista ou tabela de dados de teste separados. O procedimento de teste pode ser executado repetidamente usando diferentes conjuntos de dados. Mais detalhes e exemplos podem ser encontrados em [ISTQB_AL_TAE].

Benefícios para equipes ágeis

- As equipes ágeis podem se ajustar rapidamente à mudança ou ao crescimento das funcionalidades de um produto, de iteração em iteração, uma vez que mudar ou adicionar novas combinações de dados é simples e tem pouco ou nenhum impacto sobre automação existente.
- As equipes ágeis podem facilmente escalar a cobertura de teste necessária para cima ou para baixo, adicionando, mudando, ou removendo as entradas da tabela de dados de teste. Isto também ajuda as equipes ágeis a controlar os tempos de execução dos testes para atender às restrições de implantação contínua.
- A técnica suporta a filosofia de trabalhar de forma multifuncional, porque as tabelas de dados de teste são mais fáceis de entender do que os roteiros de teste e, portanto, permitem uma participação mais efetiva de membros de equipes menos técnicas.
- Como as tabelas de dados de teste são facilmente compreendidas, essa técnica suporta *feedback* precoce nos casos de teste e critérios de aceite de membros e clientes da equipe.
- Esta técnica ajuda as equipes ágeis a completar as tarefas de automação de testes de forma mais eficiente porque não só reduz o esforço para desenvolver novos testes, mas também reduz a manutenção dos existentes orientados a dados.

Limitações para equipes ágeis

Embora não haja limitações específicas ao uso dessa técnica em um contexto ágil, as equipes devem estar cientes das limitações gerais ao uso dessa abordagem. Mais detalhes podem ser encontrados em [ISTQB_AL_TAE].

Ferramentas

- A edição, armazenamento e gerenciamento dos dados de teste geralmente são feitos utilizando planilhas ou arquivos de texto.
- A maioria das ferramentas ou linguagens de automação de teste oferecem comandos internos para ler os dados de teste de planilhas ou arquivos de texto.

3.1.2 Teste orientado por palavras-chave

Motivação

Uma desvantagem da automação de teste é que os scripts de teste automatizados são muito mais difíceis de entender do que os procedimentos de teste manual descritos em linguagem natural. A aplicação de uma técnica de automação de teste orientado por palavras-chave ajuda a melhorar a legibilidade, a compreensibilidade e a manutenção dos scripts de teste automatizados.

Conceito

A ideia básica do teste orientado por palavras-chave é definir um conjunto de termos (palavras-chave) retiradas dos casos de uso de um produto ou de um domínio comercial do cliente e usá-las como vocabulário para definir procedimentos de teste. Devido à sua linguagem seminatural, os scripts de teste automatizados resultantes são mais fáceis de entender do que o texto simples da linguagem de programação ou script.

Mais detalhes e exemplos podem ser encontrados em [ISTQB_AL_TAE].

Existem diferentes estilos de como um procedimento de teste pode ser escrito usando palavras-chave (consulte [Linz 14], capítulo 6.4.2 Teste orientado por palavras-chave):

- **Estilo Lista**: um procedimento de teste é uma sequência ou lista simples de palavras-chave;
- **Estilo BDD**: um procedimento de teste (ou cenário) é escrito como uma sentença semelhante à linguagem natural, usando palavras-chave como partes 'executáveis' da sentença (consulte a seção 2.1.2 (BDD));
- **Estilo DSL**: uma gramática formal baseada no conceito de "linguagem específica de domínio" (DSL) define como as palavras-chave e elementos potencialmente adicionais de linguagem, podem ser combinados (consulte [Fowler / Parsons10]).

Benefícios para equipes ágeis

- Ao definir palavras-chave ou DSL, uma equipe ágil cria e padroniza o vocabulário da equipe relacionado ao domínio, o que ajuda os membros da equipe a se comunicarem de forma mais clara e precisa, ajudando a evitar mal-entendidos;
- As equipes ágeis podem reunir rapidamente comentários mais qualificados de clientes ou usuários. Os procedimentos de teste compostos por palavras-chave ou escritos no estilo BDD/DSL podem ser mais bem compreendidos pelos clientes do que o código de teste simples da linguagem do programa. Os critérios informais de aceite podem ser formalizados sem perder a "legibilidade da linguagem natural". Isso pode ajudar a entender a própria lógica de negócios e, assim, a interpretação dos critérios de aceite;
- A criação e gerenciamento de uma documentação viva determina a forma de como os casos de teste serão criados.
- A técnica ajuda as equipes ágeis a realizar suas tarefas de automação de teste de forma multifuncional, incluindo os membros não técnicos, pois a composição de procedimentos de teste a partir de palavras-chave existentes não requer habilidades de programação;
- Alterar o comportamento de uma palavras-chave definida requer muito menos esforço do que alterar o mesmo comportamento em vários procedimentos de teste. Isso pode reduzir bastante os esforços de manutenção e liberar recursos para utilizar o tempo na implementação de novos testes;
- Uma equipe ágil pode aplicar a técnica (e, portanto, obter seus benefícios) em toda a pirâmide de teste, porque as palavras-chave podem ser implementadas para conduzir o objeto de teste por interfaces arbitrárias, desde o nível de teste de interface até o nível de API (p. ex., através de chamadas de API, chamadas REST, chamadas SOAP etc.). Isso significa que o conceito também é aplicável em casos de teste de unidade e integração e não se limita apenas ao teste do sistema por meio da interface do usuário, como costuma ser assumido. No entanto, ele pode adicionar um nível de abstração desnecessário aos testes de unidade.

Limitações para equipes ágeis

Além das limitações gerais dessa abordagem, conforme descrito em [ISTQB_AL_TAE], as equipes ágeis também devem estar cientes das seguintes limitações e possíveis armadilhas:

- Para executar procedimentos de teste orientado por palavras-chave, é necessária uma estrutura de execução apropriada (p. ex., incluindo um interpretador de palavras-chave). Não é recomendado reinventar a estrutura do zero. A equipe ganhará maior velocidade usando uma estrutura ou ferramenta existente que suporte teste orientado por palavras-chave. Esse é especialmente o caso se a equipe seguir um estilo BDD ou DSL.
- Implementar uma nova palavras-chave de forma estável e de fácil manutenção é difícil e requer experiência e boa capacidade de programação. As tarefas de implementação de palavras-chave também competem contra as tarefas de codificação do produto. A velocidade de automação de testes resultante pode, portanto, ser inferior ao esperado.
- O conjunto de palavras-chave (nome, nível de abstração) ou a DSL (regras gramaticais) devem ser bem concebidas. Caso contrário, a compreensibilidade ou escalabilidade não irá ao encontro das expectativas.

- O conjunto de palavras-chave também deve ser gerenciado adequadamente. Se isto não for feito, a implementação de palavras-chave não compensa porque podem ocorrer múltiplas implementações de sinônimos, ou porque as palavras-chave não são ou são raramente utilizadas por casos de teste. Para evitar essas armadilhas, um membro da equipe deve fazer um glossário das palavras-chave.
- A equipe deve estar ciente de que a aplicação de testes orientados por palavras-chave requer alguns investimentos iniciais (p. ex., para definir as palavras-chave ou linguagem de domínio, selecionar uma estrutura apropriada, implementar o primeiro conjunto de palavras-chave) e, no início de um projeto, a velocidade de automatização pode ser lenta.

Ferramentas

- Tal como nos testes orientados por dados, uma abordagem comum, mas limitada, é utilizar planilhas de cálculo e arquivos de texto simples para editar, armazenar ou gerenciar os testes orientados por palavras-chave.
- Diversas estruturas de teste, ferramentas de execução de testes e ferramentas de gerenciamento de testes oferecem suporte integrado para teste orientado por palavras-chave (denominados "teste orientado por palavras-chave", "testes baseados em interação", "testes de processos de negócios" ou "testes orientados por comportamento", dependendo do fornecedor da ferramenta ou estrutura). As ferramentas disponíveis podem ser encontradas em [ToolList].

3.1.3 Aplicando a automação de teste a uma determinada abordagem de teste

A automação de teste não é um objetivo de teste. A automação de teste é uma estratégia que, quando adotada adequadamente, pode promover objetivos estratégicos maiores de teste, aumentando a eficiência do teste, tornando-o eficaz contra certos tipos de defeitos (p. ex., defeitos de desempenho e confiabilidade) ou permitindo a descoberta antecipada de defeitos. A estratégia está alinhada com o contexto e, portanto, está em constante evolução.

Dependendo do contexto e das necessidades da equipe a automação de teste geralmente assume formas diferentes e envolve ferramentas diferentes. Em projetos grandes, geralmente não existe apenas uma solução que atenda a todas as necessidades e, portanto, várias estratégias de automação de teste são relevantes. A aplicação da automação de teste deve ser apropriada à estratégia de teste da organização e à abordagem de teste em um determinado projeto.

A automação de teste é mais do que apenas a automação da execução do teste. A automação de teste pode desempenhar um papel importante na configuração do ambiente de teste, aquisição de liberação de teste, gerenciamento de dados de teste e comparação de resultados de teste, para dar apenas alguns exemplos. Ao planejar e projetar o uso de tais ferramentas, considere a abordagem de teste, as implicações do Ciclo de Vida Ágil implementado, os recursos dessas ferramentas de automação de teste e a sua integração com várias outras ferramentas (p. ex., automação de teste na estrutura de integração contínua).

Em alguns casos, a automação de teste atende diretamente aos objetivos de uma iteração; ou seja, criando funcionalidades. Em outros casos, a automação de teste suporta esses objetivos indiretamente, por exemplo, reduzindo o risco de regressão associado às alterações no sistema. Conforme discutido no syllabus ISTQB® CTFL-AT Agile Tester [ISTQB_FL_AT], algumas organizações optam por colocar esses esforços de suporte de automação de teste em equipes fora das próprias equipes de iteração. Em tais organizações, você pode, por exemplo, ver uma equipe separada fornecendo a criação e manutenção da estrutura de automação de teste de regressão como um serviço para várias equipes de iteração. Essa abordagem pode ser bem-sucedida se a equipe externa fornecer serviços úteis para as equipes de iteração que ajudam essas equipes a se concentrarem em seus objetivos de iteração imediata. Dependendo das equipes externas, é possível influenciar o comprometimento da equipe, enquanto transferem (parte) o controle sobre o comprometimento.

A seguir, são apresentados exemplos de considerações de automação de teste para as principais abordagens de teste, conforme declarado nos syllabi ISTQB® CTFL Foundation [ISTQB_FL], CTAL-TM Test Manager [ISTQB_AL_TM] e CTEL-TM Expert Test Manager:

Analítico:

O desenvolvimento orientado pelo comportamento (BDD) e desenvolvimento orientado por teste de aceite (ATDD) são técnicas que podem ser usadas dentro de uma abordagem analítica em um contexto ágil, por

exemplo, aplicado à automação de teste. O BDD e o ATDD podem ser usados para produzir testes automatizados em paralelo (ou mesmo antes) da implementação da história do usuário.

Baseado em modelo:

O teste baseado em modelo de comportamento funcional pode suportar a criação automatizada de testes durante a implementação da história do usuário, fornecendo uma fonte rápida de testes eficientes. O teste baseado em modelo também pode ser usado para a criação de histórias de usuários, pois os modelos podem ser usados para testar requisitos e ajudar nas revisões estáticas. Os modelos são frequentemente usados para testar comportamentos não funcionais, como confiabilidade e desempenho, que são características importantes para muitos sistemas e que são propriedades emergentes do sistema como um todo.

Metódico:

Como existem iterações curtas e múltiplas em projetos ágeis, os *checklists* de teste automatizado (com todas as atividades) podem ser usados como uma abordagem metódica para a execução eficiente de um conjunto estável de testes.

Compatível com processo:

Em projetos que devem estar em conformidade com padrões ou regulamentos definidos externamente, esses padrões ou regulamentos podem influenciar como os testes automatizados são usados ou como os resultados dos testes automatizados são capturados. Por exemplo, em projetos regulamentados pela FDA (Food & Drug Administration; alto risco), os testes automatizados e seus resultados devem ser rastreáveis aos requisitos, e os resultados devem conter detalhes suficientes para provar que o teste foi aprovado.

Reativo (ou heurístico):

O teste reativo desempenha um papel importante de validação no teste ágil, enquanto a maioria dos testes automatizados desempenha principalmente um papel de verificação. Embora as estratégias reativas sejam principalmente manuais (p. ex., testes exploratórios, detecção de erros etc.), uma proporção maior de cobertura de testes automatizados geralmente leva a muitos testes manuais que seguem estratégias reativas, pois muitos dos testes preparados podem ser automatizados. Além disso, os demais testes manuais podem cobrir áreas de maior risco.

Dirigido (ou consultivo):

Quando a cobertura do teste é especificada por stakeholders externos e a automação do teste precisa ser usada, a capacidade da equipe de teste em responder à solicitação é importante. Portanto, as equipes de teste que seguem uma estratégia de teste direcionada devem considerar o tempo e as habilidades necessárias para concluir suas tarefas nas iterações.

Contra regressão:

Nos projetos ágeis, uma característica principal da estratégia de contra regressão é o conjunto grande, estável e crescente de testes de regressão automatizados. A cobertura adequada, a capacidade de manutenção e a análise eficiente de resultados são críticas, especialmente à medida que o número de testes de regressão aumenta. Em vez de focar em um conjunto cada vez maior de testes de regressão, uma abordagem de contra regressão bem-sucedida se concentra na melhoria contínua e na refatoração dos testes criados.

3.2 Nível de automação

3.2.1 Compreender o nível necessário de automação de teste

A automação é um elemento importante nos projetos ágeis, porque não abrange apenas a automação de teste, mas também a automação do processo de implantação (consulte o Capítulo 4). A implantação contínua é a implantação automática de uma nova versão no ambiente de produção. A implantação contínua ocorre em intervalos regulares e curtos.

Como a implantação contínua é um processo automatizado, os testes automatizados devem ser suficientes para manter o nível exigido de qualidade do código. A simples execução de testes de unidade automatizados não é suficiente para obter uma cobertura suficiente de teste. Um conjunto de teste automatizado, executado como parte do processo de implantação, também deve incluir testes de integração e no nível do sistema. Na prática, alguns testes manuais ainda podem ser necessários.

A seguir, alguns desafios enfrentados pela automação de teste em uma configuração ágil:

- **Volume do conjunto de testes**: cada iteração (exceto as de manutenção ou fortalecimento) implementa características adicionais. Para manter o conjunto de testes alinhado com a funcionalidade adicional do produto, a equipe do Ágil deve aprimorar seu conjunto de testes em cada iteração. Isso significa que o número de casos de teste no conjunto aumenta por padrão de iteração para iteração. Ele requer esforços cuidadosos e deliberados para refatoração dos testes visando aumentar a cobertura sem aumentar significativamente o seu tamanho. De qualquer forma, o esforço e o tempo necessário para manter, preparar e executar o conjunto completo de teste aumentará com o tempo.
- **Tempo de desenvolvimento do teste**: os testes necessários para verificar a funcionalidade nova ou alterada do produto precisam ser projetados e implementados. Isso inclui criar ou atualizar os dados de teste necessários e preparar quaisquer atualizações no ambiente de teste. A manutenção do teste também afeta o tempo de desenvolvimento.
- **Tempo de execução do teste**: aumentar o volume do conjunto de testes levará a uma quantidade crescente de tempo necessária para a execução dos testes.
- **Disponibilidade de pessoal**: a equipe necessária para criar, manter e executar um conjunto de testes deve estar disponível para cada implantação. Pode ser difícil ou impossível garantir isso ao longo do projeto, especialmente durante feriados, fins de semana ou se as implantações ocorrerem fora do horário comercial.

Uma estratégia para enfrentar esses desafios é minimizar o conjunto de testes selecionando, preparando e executando apenas um subconjunto de testes, priorizando ou usando a análise de risco. Essa estratégia tem a desvantagem de aumentar os riscos, pois implica em um número reduzido de testes executados.

A automação de teste, tanto quanto possível deve aumentar a frequência ou a velocidade das implantações. Outra estratégia para acompanhar (ou até aumentar) a frequência ou a velocidade da implantação é automatizar o maior número possível de testes.

A automação de teste é um instrumento operacional para acompanhar ou aumentar a velocidade da implantação em qualquer projeto e é premissa para a implantação contínua. Para encontrar o volume certo de automação de teste que pode acompanhar a velocidade da implantação, os seguintes benefícios e limitações devem ser analisados e equilibrados:

Benefícios

- A automação de teste pode garantir um nível definido e repetível de cobertura de teste para cada ciclo de implantação;
- A automação de teste pode diminuir o tempo de execução do teste e ajudar a aumentar a velocidade da implantação;
- A automação de teste pode diminuir os limites para obter maior frequência de implantação;
- A implantação contínua suporta a redução do tempo de colocação no mercado (time to market) e dos ciclos de *feedback* do usuário;
- A automação de teste pode ocorrer com mais frequência, o que permite que todos os testes sejam executados a cada compilação, oferecendo uma linha principal estável na integração contínua, onde o software é mesclado à linha principal o mais rápido possível.

Limitações

- A automação de teste requer esforços de desenvolvimento e manutenção de teste, que podem prolongar a duração do desenvolvimento e, assim, diminuir a frequência de implantação;
- Testes no nível do sistema e, especialmente, testes de carga ou desempenho (e possivelmente outros testes não funcionais) podem precisar de muito tempo para serem executados, mesmo que automatizados;
- Testes automatizados podem falhar devido a muitos fatores. Um caso de teste automatizado aprovado pode não ser confiável (falso negativo). Um caso de teste automatizado com falha pode falhar devido a um erro não relacionado à qualidade dos produtos ou a um falso positivo.

A frequência de lançamento deve corresponder às necessidades dos usuários do produto. Muitas versões entregues em pouco tempo podem desagradar ao cliente mais do que uma frequência mais lenta. Portanto, em situações em que é tecnicamente possível aumentar ainda mais a frequência de implantação, pode não ser um benefício adicional do ponto de vista do cliente.

4 Implantação e entrega [120 min]

Palavras-chave

Virtualização de serviço, teste contínuo

Objetivos de aprendizagem

ATT-4.x (K1) Palavras-chave

4.1 Integração contínua, teste contínuo e entrega contínua

ATT-4.1.1 (K3) Aplicar a integração contínua e resumir seu impacto nas atividades de teste.

ATT-4.1.2 (K2) Entender o papel dos testes contínuos na entrega contínua e na implantação contínua.

4.2 Virtualização de serviço

ATT-4.2.1-1 (K2) Entender o conceito de virtualização de serviço e seu papel em projetos ágeis.

ATT-4.2.1-2 (K2) Entender os benefícios da virtualização de serviço.

4.1 Integração contínua, teste contínuo e entrega contínua

4.1.1 Integração contínua e seu impacto nos testes

O objetivo da integração contínua é fornecer *feedback* rápido para que, se os defeitos forem introduzidos no código, eles sejam encontrados e corrigidos o mais rápido possível. Os testadores ágeis devem contribuir para o desenho, implementação e manutenção de um processo de integração contínua eficaz e eficiente, não apenas em termos de criação e manutenção de testes automatizados que se encaixam na estrutura da integração contínua, mas também em termos de priorização dos testes, dos ambientes necessários, configurações e assim por diante.

Em uma situação ideal da integração contínua, uma vez que o código é criado, todos os testes automatizados são executados para verificar se o software continua se comportando conforme definido e se não foi danificado pelas alterações recebidas. No entanto, existem dois objetivos conflitantes:

1. Executar o processo de integração contínua frequentemente para obter *feedback* imediato sobre o código;
2. Verificar o código o mais minuciosamente possível após cada compilação.

Quando não são tomados cuidados adequados na modelagem, implementação e manutenção dos testes automatizados (conforme discutido no capítulo anterior), a execução de todos os testes automatizados levará muito tempo para que o processo de integração contínua seja concluído várias vezes por dia. Mesmo com uma automação cuidadosa de testes, pode ser que a execução completa de todos os testes automatizados em todos os níveis de teste atrase excessivamente o processo de integração contínua. Diferentes organizações têm prioridades diferentes e projetos diferentes precisam de soluções diferentes para encontrar o equilíbrio certo entre os objetivos mencionados acima. Por exemplo, se um sistema é estável e muda com menos frequência, menos ciclos de integração contínua podem ser necessários. Se o sistema estiver sendo atualizado constantemente, provavelmente serão necessários mais ciclos de integração contínua.

Existem soluções que suportam os dois objetivos, que são complementares e podem ser usadas em paralelo.

A primeira é priorizar o teste para que os testes mais básicos e mais importantes sejam sempre executados usando uma abordagem de teste baseada em risco.

A segunda solução é permitir que diferentes configurações de teste no processo de integração contínua sejam usadas para diferentes tipos de ciclos de integração contínua. Para o processo diário de compilação e teste, apenas os testes básicos selecionados com base na priorização inicial são executados. Para o processo noturno de integração contínua, um número maior ou possivelmente todos os testes funcionais que não exigem o ambiente de pré-produção são executados. Antes do lançamento, são realizados testes funcionais e não funcionais mais detalhados em um ambiente de pré-produção com entradas reais de usuário, incluindo testes de integração com bancos de dados, diferentes sistemas ou plataformas.

A terceira solução é acelerar a execução do teste, diminuindo sua quantidade na interface do usuário. Normalmente, não há problema de tempo para concluir a execução dos testes de unidade ou integração, que geralmente são executados muito rapidamente. Pode surgir uma situação em que existem tantos testes de unidade que eles não podem ser completamente executados durante o processo de integração contínua. Os problemas reais com o tempo geralmente estão relacionados ao uso de casos de teste de interface do usuário de ponta a ponta como parte do processo de integração contínua. A solução é aumentar a quantidade de API, linhas de comando, camada de dados, camada de serviço e outros testes de lógica de negócios que não são da interface do usuário e diminuir o teste da interface do usuário, diminuindo o esforço de automação na pirâmide de automação de teste. Isso garante mais testes de manutenção, mas também reduz o tempo de execução do teste.

A quarta solução pode ser usada quando execuções de teste são muito frequentes em um sistema de integração contínua e é impossível executar todos os casos de teste. Com base nas modificações de código e no conhecimento do rastreamento de execução dos casos de teste existentes, um desenvolvedor ou testador pode selecionar e executar apenas os casos de teste afetados pelas alterações, ou seja, utilizar-se da análise de impacto para selecioná-los. Como apenas uma pequena fração de toda a base de código é modificada durante um curto ciclo, relativamente poucos casos de teste precisam ser executados. No entanto podem ser perdidos defeitos significativos de regressão.

Uma quinta solução é dividir o conjunto de testes em pedaços de tamanho igual e executá-los em paralelo em vários ambientes. Isso é muito comum em empresas que usam a integração contínua, porque elas já precisam de muita capacidade de servidores para o processo de *build* (montar uma versão compilada do código).

A integração contínua funciona em uma variedade de plataformas tecnológicas. À medida que as ferramentas de desenvolvimento e implantação avançam em direção à nuvem, os produtos de integração contínua também. Embora a maioria dos produtos de integração contínua ainda seja projetada para baixar e executar em ambientes locais, a nuvem criou uma geração de novos produtos que fornecem serviços de integração contínua em plataformas hospedadas que as equipes podem usar rapidamente. Agora, as equipes podem evitar os custos e o tempo desnecessários de criar um ambiente novo, baixar, instalar e configurar o software de integração contínua. Ao mudar para a nuvem, a equipe pode configurar e começar a trabalhar. Quando for prático, a nuvem é uma maneira flexível de acelerar a construção de testes e as suas execuções quando necessário.

As ferramentas atuais de integração contínua oferecem suporte não apenas a ela, mas também à entrega e implantação contínuas. A execução de testes automatizados em um ambiente que não replica completamente a produção pode levar a falsos negativos, mas a clonagem do ambiente de produção pode custar caro. Algumas soluções incluem o uso de ambientes de teste em nuvem que replicam os ambientes de produção conforme a necessidade, ou a criação de um ambiente de teste que seja uma versão reduzida, mas realista, da produção.

Os bons sistemas de integração contínua precisam ter o poder de serem implantados automaticamente em ambientes mais complexos e plataformas diferentes. A tarefa dos testadores é planejar quais casos de teste incluir; priorizar quais devem ser executados em algumas ou todas as plataformas para manter uma boa cobertura; e projetar os casos de teste para validar eficientemente o software em um ambiente semelhante à produção.

4.1.2 O papel do teste contínuo na entrega e na implantação contínua

O teste contínuo é uma abordagem que envolve um processo antecipado de teste, com frequência, em qualquer lugar, e automatizado para obter feedback sobre os riscos de negócios, associados a um candidato a lançamento de *softwares* o mais rápido possível. Em testes contínuos, uma modificação feita no sistema aciona os testes necessários, ou seja, aqueles que cobrem a alteração para serem executados automaticamente, fornecendo *feedback* imediato ao desenvolvedor. O teste contínuo pode ser aplicado em diferentes situações (p. ex., no IDE, na integração contínua, na entrega contínua e na implantação contínua etc.), no entanto, o conceito é o mesmo, ou seja, para executar automaticamente os testes o mais cedo possível.

A entrega contínua requer a integração contínua. A entrega contínua estende a integração contínua, implantando todas as alterações de código em um ambiente de teste ou pré-produção ou em um ambiente semelhante à produção após o estágio de construção. Esse processo de preparação possibilita a execução de testes funcionais aplicando entradas reais do usuário e testes não funcionais, como testes de carga, estresse, desempenho e portabilidade. Como todas as alterações incorporadas são entregues ao ambiente de armazenamento temporário usando a automação completa, a equipe do Ágil pode confiar que o sistema pode ser implantado na produção com um simples toque de um botão quando a definição de concluído é alcançada.

A implantação contínua leva a entrega contínua um passo adiante com a noção de que todas as alterações são implantadas automaticamente na produção. A implantação contínua visa minimizar o tempo decorrido entre a tarefa de desenvolvimento de escrever um novo código e utilizá-lo na produção por usuários reais. Para mais informações, consulte [Farley].

4.2 Virtualização de serviço

A virtualização de serviço refere-se ao processo de criação de um serviço de teste compartilhável (um serviço virtual) que simula o comportamento, dados e desempenho relevantes de um sistema ou serviço conectado. Esse serviço virtual permite que as equipes de desenvolvimento e teste executem suas tarefas, mesmo que o serviço real ainda esteja em desenvolvimento ou não esteja disponível.

É difícil realizar testes no início do ciclo de desenvolvimento com as equipes e sistemas de desenvolvimento altamente conectados e interdependentes de hoje. Ao desacoplar equipes e sistemas usando a virtualização de serviço, as equipes podem testar seu software no início do ciclo de vida do desenvolvimento usando casos de uso e cargas mais realistas.

Embora simuladores, drivers e emuladores sejam valiosos para permitir testes iniciais, os simuladores criados manualmente geralmente são sem estado e fornecem respostas simples a solicitações com tempos de resposta fixos. Os serviços virtuais podem suportar transações com estado e manter o contexto de elementos dinâmicos (IDs de sessão ou cliente, datas e horários etc.), além de possuir tempos de resposta variáveis que modelam o fluxo de mensagens através de vários sistemas.

Os serviços virtuais são criados com a ajuda da ferramenta de virtualização de serviço, geralmente de uma das seguintes maneiras:

- **Interpretar a partir dos dados**: arquivos XML, dados históricos de logs do servidor ou simplesmente amostras da planilha de dados;
- **Monitorar o tráfego da rede**: o teste do SUT (sistema alvo de teste) acionará o comportamento relevante do sistema, capturado e modelado pela ferramenta de virtualização de serviço;
- **Capturar por agentes**: a lógica interna ou do lado do servidor pode não estar visível nas mensagens de comunicação, forçando os dados relevantes e as mensagens a serem enviadas do servidor dependente a serem recriadas como um serviço virtual.

Se nenhuma das opções acima se aplicar, o serviço virtual precisará ser criado dentro da equipe do projeto, com base no protocolo de comunicação apropriado.

Os benefícios da virtualização de serviço incluem:

- Atividades paralelas de desenvolvimento e teste para o serviço em desenvolvimento;
- Testes anteriores de serviços e APIs;
- Configuração de dados de teste anteriores;
- Compilação paralela, integração contínua e automação de testes;
- Descoberta antecipada de defeitos;
- Sobreutilização reduzida de recursos compartilhados (sistema COTS, mainframe, data warehouses);
- Custos de teste reduzidos, reduzindo o investimento em infraestrutura;
- Permitir testes não funcionais iniciais do SUT;
- Simplificando o gerenciamento do ambiente de teste;
- Menos trabalho de manutenção para ambientes de teste (não é necessário manter o *middleware*);
- Menor risco para gerenciamento de dados (o que ajuda na conformidade com o GDPR).

Observe que o serviço virtual não precisa incluir todas as funcionalidades e dados do serviço real, apenas as partes necessárias para o teste do SUT.

A introdução de uma virtualização de serviço pode ser um empreendimento complexo e potencialmente caro. A introdução de uma ferramenta de virtualização de serviço deve ser tratada de maneira semelhante à introdução de qualquer nova ferramenta de teste para a equipe e organização. Este tópico é abordado no ISTQB® CTFL Foundation Syllabus [ISTQB_FL] e no ISTQB® CTAL-TM Test Manager [ISTQB_AL_TM].

Referências

Normas e Padrões

[ISO29119-5] ISO/IEC/IEEE 29119-5 Software and systems engineering - Software testing - Part 5: Keyword-Driven Testing

Documentos ISTQB®

[ISTQB_FL] ISTQB® CTFL Foundation Level, 2018br, https://www.bstqb.org.br/sobre-ctfl

[ISTQB_FL_AT] ISTQB® CTFL-AT Agile Tester, 2014br, https://www.bstqb.org.br/sobre-ctfl-at

[ISTQB_FL_UT] ISTQB® CTFL-UT Usability Testing, 2018br, https://www.bstqb.org.br/sobre-ctfl-ut

[ISTQB_FL_PT] ISTQB® CTFL-PT Performance Testing, 2018br, https://www.bstqb.org.br/sobre-ctfl-pt

[ISTQB_FL_MAT] ISTQB® CTFL-MAT Mobile Application Testing, 2019br, https://www.bstqb.org.br/sobre-ctfl-mat

[ISTQB_FL_MBT] ISTQB® CTFL-MBT Model-Based Testing, 2015br, https://www.bstqb.org.br/sobre-ctfl-mbt

[ISTQB_ATT_OVIEW] ISTQB® Advanced Agile Technical Tester Overview, 2019br

[ISTQB_AL_TA] ISTQB® CTAL-TA Test Analyst,2019br, https://www.bstqb.org.br/sobre-ctal-ta

[ISTQB_AL_TM] ISTQB® CTAL-TM Test Manager, 2012br, https://www.bstqb.org.br/sobre-ctal-tm

[ISTQB_AL_SEC] ISTQB® CTAL-SEC Security Testing, 2016br, https://www.bstqb.org.br/sobre-ctal-sec

[ISTQB_AL_TAE] ISTQB® CTAL-TAE Test Automation Engineer, 2017br, https://www.bstqb.org.br/sobre-ctal-tae

[ISTQB_GLOSSARY] ISTQB® Glossary of Terms used in Software Testing, v3.2, 2019br, https://glossary.istqb.org

Literatura

[Adzic09] Adzic, Gojko. “Bridging the Communication Gap” Neuri Ltd, 2009

[Adzic11] Adzic, Gojko. “Specification by Example” Manning, 2011.

[Beck02] Kent Beck, “Test Driven Development: By Example”, 2002

[Beck99] Beck, Kent. “Extreme Programming Explained” Addison Wesley, 1999.

[Carkenord08] Barbara A. Carkenord, “Seven Steps to Mastering Business Analysis”, J. Ross Publishing, 2008.

[Cohn09] Mike Cohn: Succeeding with Agile: Software Development Using Scrum. Addison-Wesley Professional, 2009;

[Crispin08] Crispin, L. and Gregory, J. (2008) Agile Testing: A Practical Guide for Testers and Agile Teams. Crawfordsville : Addison-Wesley Professional

[Elfriede99] Dustin Elfriede, Automated Software Testing: Introduction, Management, and Performance: Introduction, Management, and Performance, Addison-Wesley Professional, 1999

[Evans03] Eric Evans, “Domain-Driven Design: Tackling Complexity in the Heart of Software”, 2003

[Fewster12] Mark Fewster, Dorothy Graham, Experiences of Test Automation: Case Studies of Software Test Automation, Addison-Wesley Professional, 2012

[Fowler/Parsons 10] Martin Fowler, Rebecca Parsons, Domain-Specific Languages, Addison-Wesley Signature, Series, 2010

[Hendrickson13] Elisabeth Hendrickson, “Explore It!: Reduce Risk and Increase Confidence with Exploratory Testing”, Pragmatic Bookshelf, 2013

[Jeffries00] Ron Jeffries, Ann Anderson, and Chet Hendrickson, “Extreme Programming Installed,” Addison-Wesley Professional, 2000

[Jorgensen13] Paul C. Jorgensen, Software Testing: A Craftsman's Approach, Auerbach Publications; 4th edition, 2013

[Linz14] Tilo Linz, Testing in Scrum, A Guide for Software Quality Assurance in the Agile World, Rocky Nook, 2014

[Meszaros07] Gerard Meszaros, "xUnit Test Patterns: Refactoring Test Code", Addison-Wesley, 1st Edition, Apress, 2007

[Michelsen12] John Michelsen and Jason English, "Service Virtualization: Reality is Overrated", Apress, 1st Edition, 2012.

[Osherove09] Roy Osherove, "The Art of Unit Testing", 2009

[Paskal15] Greg Paskal, "Test Automation in the Real World: Practical Lessons for Automated Testing", MissionWares, 2015

[Smart15] Smart, John Ferguson; "BDD in action", Manning, 2015

[Wein89] Weinberg, Gerald & Gause, Donald. "Exploring Requirements: Quality Before Design" Dorset House, 1989.

[Whittaker09] James A Whittaker, "Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design", Addison-Wesley Professional, 2009

[Wiegers02] Karl Wiegers, "Peer Reviews in Software: A Practical Guide (Paperback)", 2002.

Outras referências

As seguintes referências apontam para informações disponíveis na Internet e em outros locais. Embora essas referências tenham sido verificadas no momento da publicação deste syllabus, o *ISTQB®* não poderá ser responsabilizado se elas não estiverem disponíveis com o tempo.

[Cohn09] The Forgotten Layer of the Test Automation Pyramid, www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid

[CyclomaticComplexity] https://en.wikipedia.org/wiki/Cyclomatic_complexity

[Farley] http://www.davefarley.net/?cat=5

[DSL] https://en.wikipedia.org/wiki/Domain-specific_language

[Fowler07] https://martinfowler.com/articles/mocksArentSimuladores.html

[Fowler04] Fowler, Martin. https://martinfowler.com/bliki/SpecificationByExample.html

[Fowler03] Martin Fowler, https://martinfowler.com/bliki/TechnicalDebt.html

[Gherkin] https://docs.cucumber.io/gherkin

[INVEST] Bill Wake, "INVEST in Good Stories, and SMART Tasks", http://xp123.com/articles/investin-good-stories-and-smart-tasks

[IQBBA] International Qualifications Board for Business Analysts, http://www.iqbba.org

[IREB] International Requirements Engineering Board, https://www.ireb.org

[IIBA] International Institute of Business Analysts (IIBA), https://www.iiba.org

[Marick01] Marick, Brian, http://www.exampler.com/old-blog/2003/08/21/#agile-testing-project-1

[Marick03] Marick, Brian. http://www.exampler.com/old-blog/2003/08/22.1.html

[North06] Dan North, "Introducing BDD", blog post, 2006

[TESTDOUBLES] Wojciech Bulaty, Bill Wake, "Stubbing, Mocking and Service Virtualization Differences for Test and Development Teams", https://www.infoq.com/articles/stubbing-mockingservice-virtualization-differences

[ToolList] Test tool review, information platform on the international market of software testing tools, www.testtoolreview.de/en/

[xUnit] https://en.wikipedia.org/wiki/XUnit, https://en.wikipedia.org/wiki/Unit_testing

Terminologia Ágil

As palavras-chave são encontradas no Glossário do *ISTQB*® e são identificados no início de cada capítulo. Para termos comuns do Ágil, as seguintes fontes da internet são aceitas e fornecem definições: https://www.agilealliance.org/agile101/agile-glossary/. No caso de definições conflitantes, o Glossário do *ISTQB*® é a principal fonte.

Apêndice

Termos específicos para o *CTAL-ATT Agile Techinical Tester*:

Persona

Um personagem fictício que representa um determinado tipo de usuários e como ele irá interagir com o sistema.

Storyboard

Uma representação visual do sistema no qual as histórias de usuários são representadas no contexto com o objetivo de entender os processos de negócio.

Mapeamento de histórias

Técnica que ordena histórias de usuários em duas dimensões, o eixo horizontal representando sua ordem de execução e o eixo vertical representando a sofisticação do produto implementado.