Unidade III

5 VERIFICAÇÃO E VALIDAÇÃO DE *SOFTWARE*

5.1 Definições de verificação e validação

As técnicas de verificação e validação são essenciais para o processo de qualidade no desenvolvimento de *software* e são chamadas popularmente de técnicas de V&V. Existem diversas técnicas que serão apresentadas nos próximos capítulos descrevendo as várias formas de garantia e controle de qualidade dos produtos de construção da aplicação. Porém, antes de detalhar essas técnicas, são descritas algumas definições básicas de qualidade.

Como já visto, a garantia da qualidade é executada durante a construção do produto, e o controle da qualidade, após o produto ter sido construído.

As técnicas de V&V abrangem ambos os cenários, por meio de:

- aplicação de ferramentas que podem automatizar a revisão de determinados produtos; por exemplo, a simples revisão ortográfica do editor de texto é uma ferramenta automatizada de garantia da qualidade;
- utilização do processo de revisão de artefatos, como revisão por pares;
- adoção de normas e padrões, como um padrão de melhores práticas de codificação;
- controle sistemático e formal das mudanças de requisitos que podem ocorrer durante o ciclo de construção; por exemplo, a inclusão de uma nova funcionalidade no *software* deve ser avaliada e aprovada pelo cliente antes da sua incorporação ao projeto;
- manutenção de registros de todas as alterações realizadas nos artefatos, identificados, no mínimo, com o nome, a data da alteração e o motivo da alteração;
- o mais importante, que é o processo de medição para avaliar se a qualidade está se mantendo, melhorando ou piorando.

Uma das principais finalidades da V&V é garantir que o produto seja construído corretamente, ou seja, atuar de forma preventiva para que os esforços de correção de problemas causem o menor impacto possível.

Há diversas questões relacionadas à garantia da qualidade de que todo esse processo preventivo é muito caro, e não compensaria financeiramente a uma empresa de desenvolvimento de *software* aplicar

tudo o que deveria. No entanto, essa equação não é difícil de resolver. Basta analisarmos se os custos para realizar as ações de garantia da qualidade (GQ1), somados aos custos para corrigir problemas não identificados pelo processo de garantia da qualidade (GQ2), são menores que os custos para corrigir os problemas em um cenário sem nenhuma ação de qualidade (NGQ).

$$GQ1 + GQ2 < NGQ$$

O problema que as empresas enfrentam para analisar essa simples equação, porém, é definir como coletar e analisar os custos da correção desses problemas sem o processo de qualidade, porque muitas vezes não há processos definidos, os custos não são informados/controlados ou são ocultados pelas equipes de construção para evitar a busca de culpados e/ou mascarar os problemas internos. Isso acaba levando as empresas a chegarem a conclusões inadequadas ou inverídicas sobre os custos do processo de garantia de qualidade.

Nesse contexto sobre a questão de fazer ou não fazer a V&V para garantir a qualidade do *software*, os resultados da aplicação das técnicas devem ser levados em consideração para essa tomada de decisão. Em um conjunto de benefícios, podem-se destacar os seguintes:

- permite encontrar erros mais cedo;
- aumenta a integração da equipe;
- permite o acompanhamento contínuo da qualidade;
- facilita o gerenciamento;
- melhora a qualidade;
- aumenta a interação das equipes;
- avalia se o sistema está apto a ser usado em situação operacional.

O processo de verificação e validação deve ser aplicado durante todo o ciclo de desenvolvimento do *software* e deve fazer parte do planejamento das demais atividades do processo de construção, ou seja, essas atividades de qualidade devem estar explícitas no cronograma de trabalho do projeto. A falta desse planejamento pode fazer a qualidade ser verificada somente na fase de testes, gerando custos mais elevados para encontrar falhas introduzidas no início do desenvolvimento e aumentando o tempo necessário para a realização dos testes. Portanto, as ações de V&V devem ser iniciadas o mais cedo possível e focar as partes mais críticas do sistema.

5.1.1 Verificação

As atividades de verificação consistem nas ações realizadas ao final de cada fase, interação ou artefato, produzido durante o ciclo de desenvolvimento do *software* com o objetivo de atestar que o produto está sendo desenvolvido corretamente, conforme ilustrado na Figura 27.

5.1.2 Validação

As atividades de validação consistem nas ações realizadas ao final ou no decorrer do processo de desenvolvimento do *software* com o objetivo de avaliar se o produto está de acordo com as especificações de requisitos iniciais fornecidas pelo cliente e garantir que o produto tenha sido desenvolvido corretamente, conforme ilustrado na Figura 27.

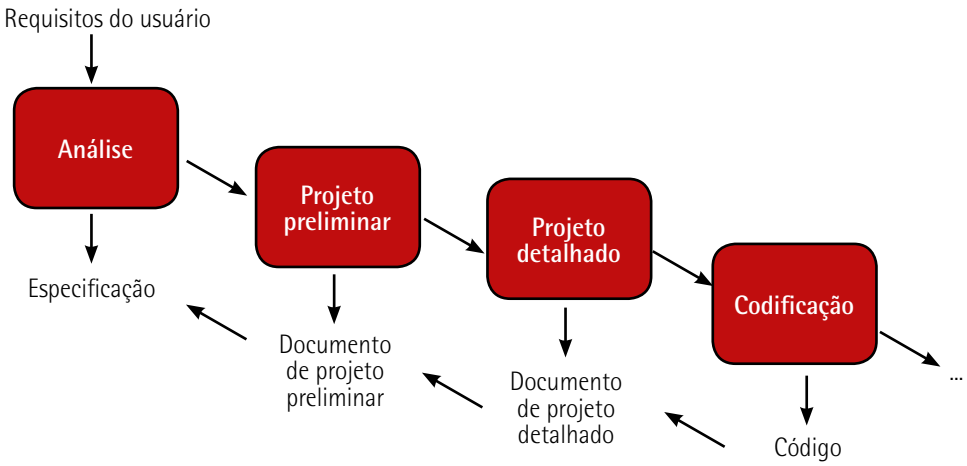


Figura 27 – Atividades de verificação da qualidade

Lembrete

A verificação e a validação dos artefatos e códigos produzidos durante o ciclo de desenvolvimento de *software* é a principal ferramenta de garantia da qualidade.

5.1.3 Técnicas de V&V

Para realizar as atividades de V&V podem ser utilizadas técnicas que se classificam em duas categorias: técnicas estáticas e técnicas dinâmicas.

As **técnicas estáticas** são aquelas realizadas, de forma manual ou automática, sobre os artefatos de documentação e modelagem do sistema e não necessitam da execução do *software*. Por exemplo:

- avaliação de diagramas de casos de uso;
- avaliação de diagramas de classes;
- avaliação do modelo de dados;
- inspeções de código-fonte.

As **técnicas dinâmicas** são aquelas realizadas, de forma manual ou automática, sobre o *software* construído e que necessitam de sua execução, por exemplo:

- execução de simulação;
- realização de testes.

Nos próximos tópicos são detalhadas as principais técnicas de verificação e de validação.

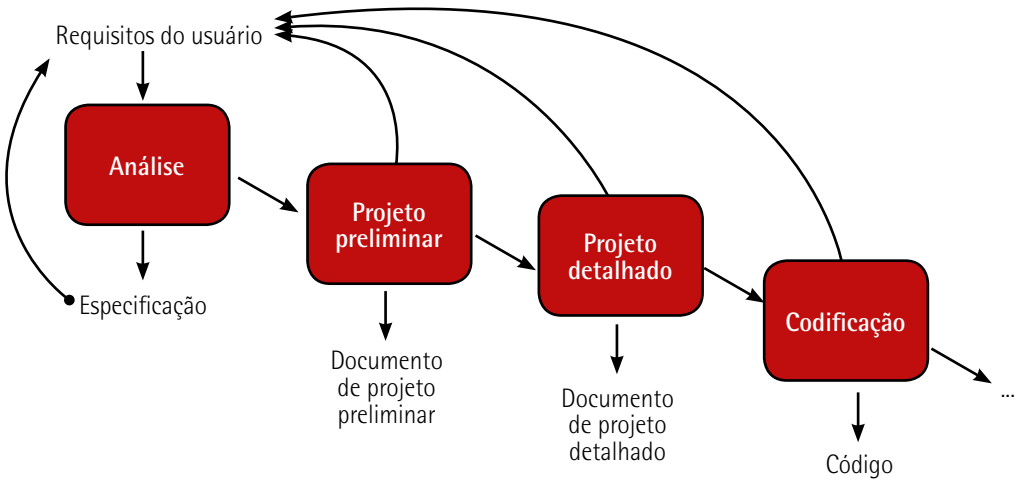


Figura 28 – Atividades de validação da qualidade

5.2 Revisões técnicas

Segundo o SEI-CMMI (2003), o processo de revisão é uma avaliação crítica de todos os artefatos produzidos durante o desenvolvimento de um *software*, não apenas sobre o código-fonte, em pontos predefinidos do ciclo de vida com o objetivo de encontrar e corrigir eventuais erros inseridos durante o processo. O fato de encontrar esses erros mais cedo proporciona uma redução de custos no processo de desenvolvimento.

Pressman (2006, p. 585) destaca uma série de benefícios à qualidade do *software* que podem ser alcançados com a utilização do processo de revisão:

- verificação se o produto de trabalho está em conformidade com os padrões, as especificações e os requisitos definidos;
- identificação de melhorias necessárias em um produto de trabalho;
- documentação e geração de histórico de erros;
- consenso entre cliente e equipe sobre os produtos de trabalho;

- aumento do conhecimento da equipe;
- "via" gerencial para, formalmente, completar uma tarefa.

As revisões técnicas são atividades de garantia da qualidade realizadas durante o processo de desenvolvimento e são uma forma de envolver outros membros da equipe e/ou externos a esta, com o objetivo de chegar ao consenso de que o produto de *software* está de acordo com as expectativas. A partir da revisão, podem ser identificados desvios em relação ao padrão definido e realizadas correções e melhorias no produto.

O objetivo principal da revisão não é encontrar erros, mas sim permitir o alinhamento de conhecimento entre os envolvidos no processo e o atendimento às expectativas do cliente.

Uma revisão técnica pode ter características informais ou formais, dependendo das necessidades e/ou dos objetivos que se deseja atingir. São apresentadas neste tópico as características de uma Revisão Técnica Formal (RTF).

5.2.1 O processo de revisão

O processo de RTF, ilustrado na Figura 29, deve ser estruturado, conduzido em uma reunião e será tão bem-sucedido quanto for planejado e controlado. Deve ser realizado sobre artefatos que tratem de assunto único ou correlato para permitir melhor avaliação e aumentar as probabilidades de sucesso.

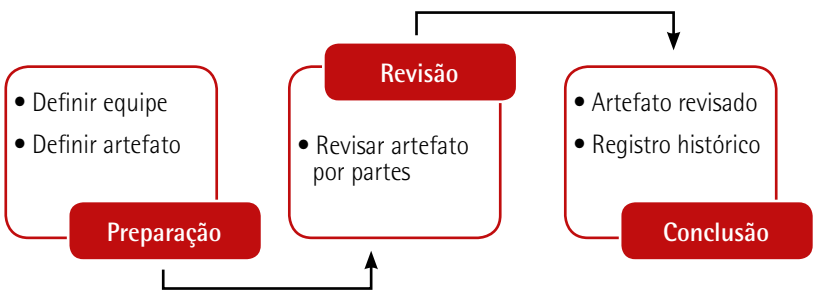


Figura 29 – O processo de revisão técnica

Para a obtenção de melhores resultados durante uma sessão de RTF, os seguintes procedimentos devem ser observados, mas não se limitando a estes:

- a equipe de revisão deve ter de três a cinco pessoas internas e externas ao projeto;
- a equipe deve ser convidada e informada sobre o objetivo da revisão;
- o artefato a ser revisado deve ser enviado com antecedência para que todos leiam previamente;
- ter um *checklist* (lista de verificação) preparado;

- os papéis devem estar claramente definidos: moderador, autor e revisores;
- iniciar pela leitura do documento pelo autor, de acordo com a estrutura do documento;
- a cada parte do documento, a equipe de revisores faz os comentários;
- o líder faz a mediação em eventuais conflitos e discordâncias;
- todos os comentários devem ser registrados para histórico.

A reunião deve ter, no máximo, a duração de 2 horas.

5.2.2 Diretrizes básicas de uma RTF

Pressman (2006, p. 586) apresenta ainda um conjunto de diretrizes mínimas a serem seguidas durante uma RTF:

- fixe e mantenha uma agenda com os participantes;
- revise o produto, não o autor do artefato;
- faça anotações por escrito;
- enuncie os problemas, mas não tente resolver cada um deles;
- limite o debate;
- realize um treinamento sobre revisões para todos os revisores;
- reveja suas antigas revisões.

As revisões técnicas formais são procedimentos relativamente simples que produzem resultados significativos na garantia da qualidade de um produto de *software*. Planeje e utilize sempre a técnica de RTF.

5.3 Passeio (*walkthrough*)

Os w*alkthroughs* são revisões técnicas informais de um artefato de *software* visando à garantia da qualidade. Normalmente são chamados de **revisão por pares**, mas podem ter até três participantes: autor, revisor e moderador. O revisor pode ser um técnico, um cliente ou uma pessoa externa ao projeto que domine o assunto em revisão. O moderador, preferencialmente, não deve ser do mesmo nível hierárquico do autor e do revisor.

Os passeios são bem mais simples que as RTFs, e seu alto grau de informalidade torna esta técnica muito aceita pela maioria das equipes de projeto e uma prática recomendada pelos métodos ágeis de desenvolvimento. Apoia a recomendação da qualidade a qual determina que tudo deve ser revisado antes de ser entregue ao cliente.

Saiba mais

Métodos ágeis são processos de desenvolvimento que se caracterizam pela velocidade e pela simplicidade da construção de um *software*. Veja mais detalhes em: <http://desenvolvimentoagil.com.br/>.

5.3.1 O processo de *walkthrough*

Por se tratar de reuniões informais para avaliação dos produtos, o processo é extremamente simples e não envolve agendamento, preparação ou planejamento. A Figura 30 ilustra esse processo.

Figura 30 – O processo de *walkthrough*

Possui basicamente as seguintes características:

- pouca ou nenhuma preparação requerida;
- objetivo de comunicar ou receber aprovação do artefato;
- papéis específicos não são estabelecidos;
- o autor guia os presentes pelo artefato;
- o revisor lê o documento e faz suas considerações;
- o autor nunca pode ser o leitor.

Observação

Os w*alkthroughs* são muito úteis, principalmente, para revisar artefatos de requisitos e de modelagem do *software*, como especificações de casos de uso, diagramas de classes, dentre outros.

5.3.2 Diretrizes básicas de um w*alkthrough*

Algumas recomendações importantes que devem ser observadas e obedecidas durante uma sessão de w*alkthrough* são apresentadas a seguir:

- o autor seleciona os revisores e convida-os para a reunião;
- o autor entrega o artefato aos revisores na reunião;
- o foco deve ser o artefato, e não o autor;
- o autor deve apresentar o artefato durante a reunião;
- os revisores apresentam possíveis falhas, comentários e sugestões de melhoria;
- com base nas informações apresentadas, o autor faz as devidas correções.

5.3.3 Revisões progressivas por pares

As revisões progressivas por pares constituem uma variação do w*alkthrough* e apresentam características deste e da RTF. Basicamente, o artefato é dividido em partes e distribuído aos revisores. O seguinte procedimento deve ser cumprido:

- o artefato deve ser separado, e suas partes, distribuídas aleatoriamente para os revisores (por exemplo, numa revisão de código, cada revisor pode ficar com um trecho de código);
- no procedimento de revisão, cada revisor faz a leitura do artefato, como em um *walkthrough*;
- todos os revisores fazem suas considerações, que são registradas;
- ao término de cada trecho revisado, o revisor passa a vez para o próximo, e assim sucessivamente, até que todo o material seja revisado;
- se possível, o autor já faz as alterações sugeridas para verificar a eficácia destas.

5.4 Técnica de inspeção

Trata-se de uma técnica de verificação extremamente formal, em que os envolvidos examinam os artefatos produzidos contra uma especificação inicial com o objetivo de encontrar incoerências, inconsistências e erros.

As inspeções podem ser realizadas a qualquer momento dentro do ciclo de vida de um produto de *software* (análise, projeto, codificação e testes), não necessitando da execução do sistema para serem feitas. São efetivas para encontrar, documentar e corrigir erros em qualquer artefato produzido, pois

devem envolver pessoas com conhecimento e domínio do assunto que está sendo inspecionado e possuir um *checklist* dos pontos que devem ser verificados no artefato.

Em um processo de inspeção, qualquer produto resultado do trabalho de desenvolvimento pode ser analisado, mas não se limitando a estes:

- documento de requisitos;
- especificação de casos de uso;
- diagrama de classes;
- protótipo de telas;
- especificações de telas;
- diagrama de arquitetura do sistema;
- modelo de dados;
- especificação de projeto;
- código-fonte;
- casos de testes;
- roteiro de testes;
- plano de implantação.

5.4.1 Processo de inspeção

Segundo Pressman (2006), um processo de inspeção é composto de seis etapas, conforme ilustrado na Figura 31 e detalhado a seguir:

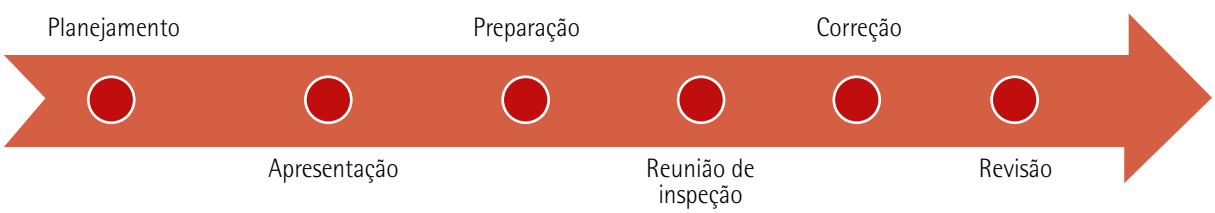


Figura 31 – O processo de inspeção

5.4.1.1 Planejamento

Fase inicial que compreende a previsão da inspeção dos artefatos no cronograma do processo de desenvolvimento de *software*, sem o qual a formalidade prevista não será atendida.

A seguir, o artefato a ser revisado, bem como todos os documentos de referência para a reunião de inspeção, são reunidos e apresentados, para que a equipe de inspeção seja dimensionada e o número de reuniões seja definido.

Os artefatos e demais subsídios são distribuídos à equipe de inspeção.

5.4.1.2 Apresentação

Trata-se de uma etapa opcional diretamente ligada ao tamanho e à complexidade do artefato a ser inspecionado.

No caso de ocorrer, a equipe de inspeção é reunida, e o autor faz uma explicação sobre os pontos principais do artefato.

5.4.1.3 Preparação

Os inspetores devem fazer a leitura prévia de toda a documentação distribuída dentro do tempo planejado e fazer suas anotações de correções necessárias e eventuais melhorias identificadas. Devem ser treinados e capacitados no processo de inspeção e ter domínio do assunto a que o artefato inspecionado se refere.

5.4.1.4 Inspeção

Consiste na reunião propriamente dita. O leitor faz a apresentação do artefato por partes, para que os inspetores façam os questionamentos, que são respondidos pelos autores, e os apontamentos de correção, que são registrados pelo escritor. Os apontamentos devem ser classificados quanto a sua gravidade. O processo se repete até que todo o artefato seja inspecionado.

Ao final, a equipe faz as considerações de melhoria e decide se o artefato será aceito, aceito com ressalvas ou se necessita de nova inspeção.

5.4.1.5 Correção

As correções e melhorias apontadas durante a sessão de inspeção devem ser feitas pelos autores e registradas as respectivas soluções.

5.4.1.6 Revisão

Os autores fazem a apresentação das soluções aplicadas aos apontamentos, e o moderador deve dar o aceite a cada item registrado e determinar a situação final da inspeção. O coordenador de inspeções recebe o documento final da inspeção.

5.4.2 Papéis e responsabilidades

Por se tratar de um processo formal, uma inspeção define claramente quem são os integrantes da inspeção e quais são as responsabilidades de cada um deles, com o objetivo de melhorar a comunicação e evitar conflitos durante a sessão. A sessão deve, obrigatoriamente, envolver um usuário final do artefato inspecionado.

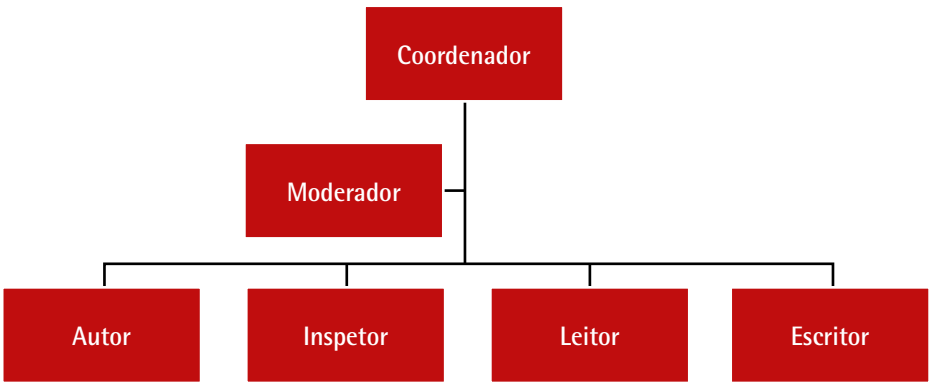


Figura 32 – Papéis em uma inspeção

Segundo Pressman (2006), os papéis mais comuns em uma sessão de inspeção, conforme a Figura 32, são:

- Coordenador das inspeções: é o responsável pelo registro de toda a sessão, pela geração do relatório final da inspeção e pela documentação das métricas de erros e correções que precisam ser realizadas visando à melhoria contínua. Também faz a aceitação final do produto inspecionado.

- Moderador: é o responsável pelo planejamento, pela montagem da equipe de inspeção junto com o autor e pela condução da sessão a partir do *checklist*. Trata-se do facilitador do processo.

- Autor: é o elaborador do artefato a ser inspecionado; distribui o documento aos participantes, monta a equipe junto com o moderador, tira dúvidas durante a sessão e faz as correções dos erros identificados.

- Leitor: faz a leitura gradual do artefato para que a equipe faça as observações, os questionamentos e o apontamento de incorreções.

- Inspetor: examina o artefato antes da reunião de inspeção, aponta erros e faz sugestões de melhoria. Todos os participantes da reunião de inspeção têm o papel de inspetor.

- Escritor: registra as incorreções apontadas pelos inspetores.

Esses são os papéis mínimos em uma inspeção, e não quer dizer que tenhamos apenas uma pessoa em cada papel. A cada inspeção, definem-se os participantes de forma coerente com o que precisa ser verificado.

5.4.3 *Checklist* de uma inspeção

Os *checklists* são listas de verificação elaboradas previamente com base em informações históricas de outras inspeções, lições aprendidas, dados fornecidos por especialistas e itens retirados de padrões preexistentes contra os quais o artefato será avaliado. Um *checklist* é incrementado a cada inspeção realizada, registrando as informações para inspeções futuras, e deve ser único por artefato elaborado.

O Quadro 16 apresenta alguns exemplos de itens de verificação para alguns artefatos típicos do processo de desenvolvimento de *software*.

Quadro 16 – Exemplos de itens de *checklists* para artefatos de *software*

Artefato	Itens de verificação
Especificação dos casos de uso	As precondições foram descritas? A especificação está de acordo com a tela? Os fluxos alternativos foram corretamente indicados? As regras de negócio foram apontadas? Na descrição dos casos de uso não há referências de navegação? Os atributos de entrada e saída estão descritos? As pós-condições estão explícitas?
Diagrama de classes	As classes identificadas referem-se a objetos reais do sistema? Os relacionamentos de agregação e herança fazem sentido no contexto do sistema? Métodos privados foram identificados? As classes têm menos de vinte atributos? O requisito de baixo acoplamento foi respeitado? O requisito de alta coesão foi atendido?
Diagrama de dados	Todas as tabelas possuem chaves primárias? A chave primária está correta? Existem índices secundários criados? As chaves secundárias estão corretamente relacionadas às tabelas?
Documento de arquitetura	Foram considerados os padrões do cliente? Existe uma implementação do padrão definido para servir de referência? Foram considerados os requisitos de segurança? Foram considerados os requisitos de desempenho? Foram considerados os padrões de implementação da linguagem?
Codificação	As variáveis estão com nomes adequados? Todas as variáveis são inicializadas? Todas as variáveis são do tipo privado? Os métodos estão com nomes adequados? Todos os métodos retornam valores? Condições e *loops* estão corretamente grafados? Existem mais de duas condicionais aninhadas?

Uma inspeção é extremamente eficaz para a identificação de erros e a verificação de suas correções, tornando-se uma das principais ferramentas para a garantia da qualidade de um produto de *software*. Por não necessitar de execução da aplicação, seu caráter preventivo e de completude a faz mais efetiva que os testes unitários e integrados realizados pelas equipes de desenvolvimento.

O planejamento e a realização de inspeções devem ser parte integrante do cronograma de desenvolvimento de um sistema por equipes maduras e experientes com foco na qualidade de seus produtos.

5.5 Sala limpa (*clean room*)

O nome é derivado do processo *clean room*, usado na fabricação de circuitos integrados e proposto para a engenharia de *software* em 1980 por Mill, Dyer e Linger. Essa técnica é baseada em especificações formais matemáticas e destinada ao desenvolvimento de *software* de alta confiabilidade como os utilizados no controle de usinas nucleares, na navegação de aviões, trens, metrôs e navios, dentre outros.

O foco da **sala limpa** é a realização de ações preventivas, e não a correção de erros. A verificação se dá por meio de inspeções rigorosas, provas matemáticas e testes estatísticos para determinar a confiabilidade do sistema, tornando-a diferente de outras técnicas (SOMMERVILLE, 2013).

Embora esteja comprovado que os resultados da confiabilidade do *software* aumenta consideravelmente com o processo sala limpa, Pressman (2006) atribui três situações que fazem a técnica não ser largamente utilizada no mercado de desenvolvimento de *software*:

- a crença de que é uma metodologia muito teórica, matemática e radical para o desenvolvimento de *software*;
- a substituição de testes unitários por verificação de correção e controle estatístico de qualidade é muito diferente do método costumeiramente empregado pelos desenvolvedores de *software*;
- as empresas de desenvolvimento de *software* ainda não possuem nível de maturidade suficiente para aplicar o processo sala limpa.

Agrega-se a essas três situações o fato de ser um processo muito mais custoso que os tradicionais, tanto na especificação como na verificação, além de exigir maior qualificação técnica e matemática dos profissionais envolvidos no processo e manutenção de treinamentos constantes das equipes para a garantia dos padrões exigidos e para o tratamento de mudanças nos membros das equipes.

5.5.1 Processo sala limpa

Segundo Sommerville (2013), o processo sala limpa é estruturado em quatro processos e suas respectivas atividades, ilustrados na Figura 33:

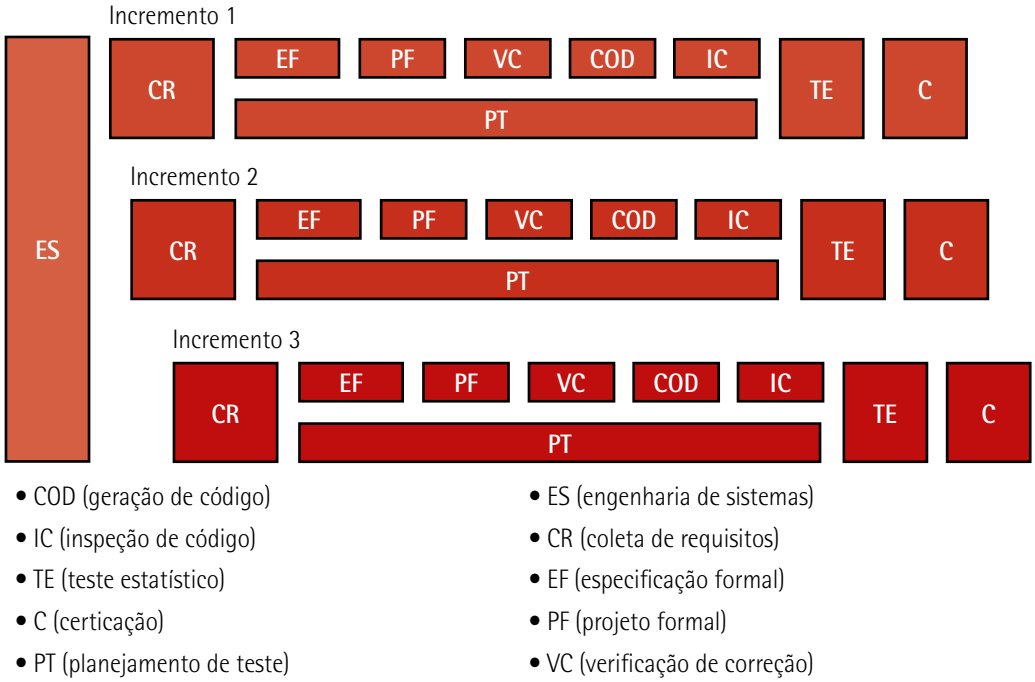


Figura 33 – O desenvolvimento com o processo sala limpa

- Processo de gerenciamento: define as atividades relacionadas à gestão do projeto com as ações de planejamento e de gestão das mudanças durante o processo.

- Processo de especificação: descreve as atividades de coleta de requisitos, a especificação formal dos requisitos, a especificação formal do projeto, a definição da arquitetura do sistema, o plano de desenvolvimento incremental do projeto e a verificação formal das especificações.

- Processo de desenvolvimento: define as atividades de engenharia relativas à codificação e à inspeção formal do código produzido.

- Processo de certificação: define o uso dos modelos de referência, os testes estatísticos formais, a execução da aplicação e a certificação da confiabilidade do *software* em desenvolvimento.

5.5.2 Testes estatísticos

São utilizados com o objetivo de avaliar as condições operacionais do *software* relacionadas à robustez, ao desempenho e à confiabilidade a partir de registros em arquivos em tempo de execução (*logs*) para gerar as ocorrências que possam ser avaliadas por meio de dados estatísticos e matemáticos, tais como: número de falhas observadas, tempos de resposta, tempos de execução, dentre outras. Com essas medidas estatísticas registradas, a confiabilidade é definida com base nos resultados dos testes realizados. Se uma longa sequência de testes for conduzida sem falha ou ocorrência de defeitos, o tempo médio de falhas – *Mean Time to Failure* (MTTF) será baixo, e a confiabilidade do produto de *software* poderá ser considerada alta.

Por exemplo, vamos supor que você tenha testado três sistemas exatamente iguais e de forma paralela, até que ocorresse uma falha ou um defeito em cada um deles. O primeiro sistema apresentou uma falha após 20 horas de execução, o segundo sistema falhou após 18 horas e o terceiro sistema falhou após 24 horas. Neste caso, o MTTF é igual à média aritmética dos três tempos registrados, ou seja, é calculado como (20 + 18 + 24)/3 que apresenta o resultado igual a 20,66.

Observação

Logs são arquivos físicos ou tabelas que registram as diversas atividades realizadas pelo *software* durante a sua execução, mediante previsão de gravação dessas informações.

5.5.3 Certificação de qualidade

Para que a certificação de qualidade ocorra, os cenários de testes são criados e especificados, e os casos de testes são gerados e executados. O resultado dessa execução gera os dados das falhas que permitem o cálculo da confiabilidade, e, estando dentro da média esperada (MTTF), o *software* recebe a certificação.

Observação

MTTF é um indicador de confiabilidade de *software* que mostra o tempo médio em que uma falha não recuperável acontece durante os testes de um *software*.

Saiba mais

Para obter mais informações sobre confiabilidade de *software* e especificações formais, visite o *site*: <www.reliasoft.com.br>.

6 TESTES DE *SOFTWARE*

A atividade de teste de *software* sempre foi considerada como um gasto de tempo desnecessário, uma atividade de segunda classe, uma vez que a programação é a atividade principal no desenvolvimento de *software*. Os testes tinham apenas o objetivo de mostrar que o sistema funcionava.

Somente a partir da década de 1980 começaram a ser elaborados métodos de testes que passaram a fazer parte do processo de desenvolvimento de *software*, de maneira formal e como uma atividade essencial ao processo de construção, que passou a ter o objetivo de garantir que o sistema atendesse aos requisitos especificados.

No final da década de 1990 começaram a ser criadas as funções de gerente de testes, analista de testes e operador de testes, que são especialistas no planejamento, na elaboração e na execução dos testes, tornando estes cada vez mais relevantes no ciclo de vida do *software* e assumindo um perfil de prevenção de problemas, e não apenas de localização de erros.

6.1 Fundamentos sobre testes de *software*

Atualmente, o grau de exigência dos usuários com a qualidade dos produtos de *software* está cada vez maior. A complexidade das aplicações em relação à granularidade, a segurança e a integração com outros sistemas e o aumento da concorrência no mercado de *software* tornam os testes uma atividade essencial e indispensável, com equipes exclusivas e dedicadas a essas ações, com o objetivo de garantir e controlar a qualidade do sistema.

Nos modelos de qualidade de processo de *software*, o processo de testes e as atividades de V&V aparecem apenas nos níveis de maturidade intermediários, como o Nível 3 para o CMMI e o Nível D do MPS.BR, conforme ilustrado na Figura 34.

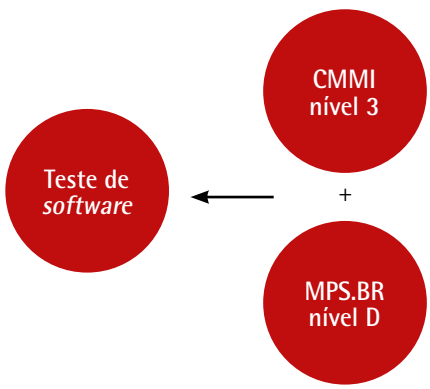


Figura 34 – O processo de testes nos modelos de qualidade

6.1.1 Conceitos de testes de *software*

As definições sobre testes de *software* variam, mas todas convergem para os conceitos básicos de encontrar defeitos em um *software* que está sendo testado, conferir se está de acordo com os requisitos definidos pelo usuário e verificar se realiza o que deveria ser feito.

Segundo Myers (2004), testar um *software* é um processo de executar um programa ou sistema com a intenção de encontrar defeitos.

Para Dijkstra (1970), os testes podem mostrar a presença de falhas em um *software*, mas nunca a sua ausência.

Para Hetzel (1988), testes são uma atividade que, a partir da avaliação de um atributo ou capacidade de um programa, torna possível determinar se ele alcança os resultados esperados.

Para o Instituto de Engenheiros Eletricistas e Eletrônicos (IEEE), teste é um processo de execução de um sistema ou programa, sob condições específicas, para detectar diferenças entre os resultados obtidos e os esperados.

Saiba mais

IEEE é uma entidade internacional que publica trabalhos sobre Tecnologia da Informação, dentre ouras áreas. Acesse o *site*: <www.ieee.org.br>.

É importante ressaltar que muitos desenvolvedores ainda confundem o processo de testes com o processo de depuração de um programa, mas ambos são completamente diferentes, pois enquanto os testes são uma busca estruturada para encontrar erros em um programa pronto, a depuração, mais conhecida como *debug*, é um processo de busca de erros de forma não estruturada durante a execução de um programa, por meio de uma ferramenta de apoio ao desenvolvimento. Uma vez encontrado, o erro deve ser corrigido. A depuração, normalmente, ocorre durante a atividade de programação e antes da execução dos testes propriamente ditos.

Os testes devem ser realizados: pelos desenvolvedores, por meio dos testes unitários; pelos testadores, mediante os testes integrados guiados por roteiro de testes elaborados a partir da especificação inicial; e pelos usuários, por meio dos testes de aceitação, nos quais é verificado se o *software* atende às necessidades. Em outras palavras, testes de *software* são uma atividade de validação, de responsabilidade de todos os que participam do processo de desenvolvimento, para garantir a qualidade.

6.1.2 Conceituação de defeito, erro e falha

Constantemente há confusão entre o uso das palavras erro, defeito e falha pelos profissionais de Tecnologia da Informação, principalmente, pela tradução dos termos da língua inglesa. Para esclarecer esses conceitos, o IEEE, por meio da norma 610.12 (1990), definiu os seguintes fundamentos para esses três termos descritos no Quadro 17.

Quadro 17 – Definições de termos segundo o IEEE 610.12

Termo	Termo em inglês	Definição
Engano	*Mistake*	Ação humana que produz um resultado incorreto
Falha	*Fault* ou *bug*	Manifestação, no *software*, de um engano cometido pelo desenvolvedor
Erro	*Error*	Diferença entre o valor obtido e o valor esperado, ou seja, qualquer estado inesperado na execução do *software*
Defeito	*Failure*	Incapacidade de o *software* fornecer o serviço conforme especificado

Fonte: IEEE (1990).

Portanto, a partir das falhas inseridas no código pelos desenvolvedores, acontecem os erros e os defeitos do *software*, que são detectados pelo processo de testes. As principais falhas do *software* ocorrem em virtude dos enganos cometidos nas especificações iniciais da fase de requisitos, que mapeiam as necessidades dos usuários e que não são detectadas pelo processo de testes. Além dessa falha original, novos erros e defeitos são inseridos por meio de:

- alterações e mudanças constantes que afetam o *software*, tornando a manutenção cada vez mais difícil;
- tempo reduzido de implementação que pressiona o desenvolvedor a cometer mais erros;
- código mal-escrito, fora de padrões estabelecidos;
- falta de documentação do *software*.

Segundo Rios e Moreira (2013), diversos tipos de defeitos podem ser encontrados em um *software*. Esses defeitos podem ser evitados, desde que sejam do conhecimento do desenvolvedor, e podem fazer parte de um *checklist* de boas práticas para serem tratados nas fases iniciais da codificação, juntamente com os padrões de código. Os principais defeitos que podem ocorrer em um *software* são apresentados no Quadro 18.

Quadro 18 – Principais tipos de defeitos de *software*

Tipo de defeito	Definição
Funcionalidade	Quando o *software* não faz o que o usuário espera que ele faça
Usabilidade	Quando há dificuldade de navegação, a cor do texto está muito clara, dificultando a leitura, ou o conteúdo é muito extenso, obrigando o usuário a usar barra de rolagem constantemente
Desempenho	O *software* não atende com a rapidez necessária às solicitações do usuário, especialmente, no caso dos sistemas muito interativos
Prevenção de defeitos	O programa não se protege das entradas de dados não previstas, que, posteriormente, são processadas de forma inadequada. O *software* pode, por exemplo, aceitar valores em branco em campos numéricos
Detecção e recuperação de defeitos	O programa não trata as operações, como: *overflow*, *flags* de defeitos, ou trata de forma inadequada
Limites	O *software* não consegue tratar ou trata inadequadamente valores extremos (o maior, o menor, o primeiro, o último) ou fora dos limites
Cálculo	O *software* executa um cálculo e produz um resultado errado. Muitas vezes, por questões de aproximação, uma fórmula não produz os resultados esperados
Inicialização ou fechamento	Alguns *softwares* ou rotinas devem ser inicializados quando usados pela primeira vez ou sempre que são chamados para execução. Exemplo de inicialização de programa: na primeira vez em que executa, o *software* deve criar um arquivo em disco. A ausência deste arquivo poderá causar problemas nas etapas seguintes do processamento
Condições de disputa	Ocorre quando o *software* espera pela resposta dos eventos A e B, sendo suposto que A sempre termine primeiro. Se por algum problema B terminar primeiro, o *software* poderá não estar preparado para esta situação e apresentar resultados inesperados

Carga	O *software* pode não suportar um pico de serviço em um determinado momento (estresse) ou uma carga alta de serviço por um tempo muito prolongado
Hardware ou *software*	Capacidade do equipamento ou do *software* básico de suportar as condições de operação da aplicação por tempo prolongado

Fonte: Rios; Moreira (2013).

6.1.3 Por que devemos testar um *software*?

Além do fator qualidade, que foi exposto até aqui, afeta a satisfação do cliente e é um dos principais indicadores de uma organização, o fator custo é um grande incentivador para que as organizações apliquem e utilizem o processo de testes no ciclo de desenvolvimento de *software*.

Segundo um dos maiores especialistas em desenvolvimento de *software* no mundo, Boehm (1976), quanto mais tarde um defeito for identificado, mais caro ficará para corrigi-lo. Além disso, os custos para se identificar e corrigir os defeitos em um *software* aumentam exponencialmente na proporção em que o trabalho evolui em suas fases de desenvolvimento.

De acordo com Myers (2004), quanto mais cedo for descoberto e corrigido o defeito, menor será o seu custo para o projeto, pois esse custo cresce até dez vezes a cada fase para a qual o projeto do *software* avança. Essa afirmação é conhecida como a Regra 10 de Myers.

É do conhecimento do mercado de Tecnologia da Informação que as organizações chegam a gastar até 30% do esforço total do desenvolvimento de *software* realizando testes, e, no caso de construção de *softwares* críticos, como *software* de controle de voo, trens e navios, a fase de testes pode custar de três a cinco vezes mais que todas as demais fases de um projeto tradicional.

Garantir que o *software* esteja sem erros e defeitos é um dos desafios da Engenharia de *Software*, mas não é possível rever todas as combinações de testes que garantam a total cobertura de testes em uma aplicação. Portanto, sempre haverá algo a se fazer, seja testar, seja corrigir um defeito. Contudo, de todas as formas de verificação e validação da qualidade, o processo de testes é a forma mais eficaz de se aproximar do erro zero e, aliado ao trabalho de equipes de testes cada vez mais especializadas e independentes, os resultados tendem a ficar cada vez melhores, e os objetivos de aumento da qualidade e redução dos custos podem ser alcançados com esforços cada vez menores.

6.2 Ciclo de vida de testes

O ciclo de vida do processo de testes deve seguir rigorosamente o processo de desenvolvimento adotado para a construção do *software*.

O mais utilizado e conhecido processo de testes é o Modelo V, visto a seguir, o qual prevê que os testes devam começar o mais cedo possível, a partir de uma especificação inicial, devendo ser realizados durante todo o processo de construção. Porém, a falta dessa especificação ou a existência de especificações incompletas e ambíguas podem levar a resultados insatisfatórios nos testes e demonstrar

falsas conclusões a respeito da qualidade do *software*. A definição de um oráculo correto e validado pelos usuários é essencial para o sucesso dos testes.

Observação

Oráculo é uma referência que define os resultados esperados para serem comparados com os resultados reais durante os testes de um *software*.

Segundo Rios e Moreira (2013), para que o processo de testes seja eficiente, é necessário:

- entender o sistema em todos os seus detalhes;
- dominar as técnicas de testes;
- ter habilidade para aplicar essas técnicas.

6.2.1 O Modelo V

O Modelo V foi desenvolvido a partir do processo de desenvolvimento cascata, no início da década de 1980, para incluir o processo de testes no ciclo de desenvolvimento do *software*. O foco é desenvolver os testes desde o início do ciclo de vida do *software*, para permitir a identificação de defeitos o mais corretamente possível, por meio de atividades de verificação e validação e da criação de casos de testes e roteiro de testes. A Figura 35 ilustra o Modelo V de testes e de desenvolvimento.

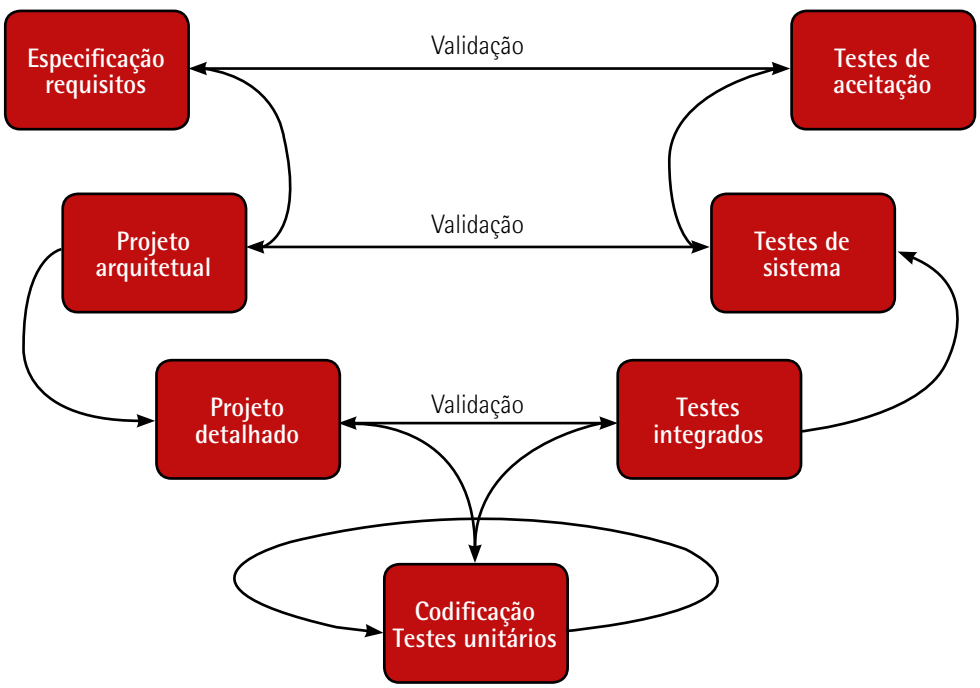


Figura 35 – O Modelo V do processo de testes

Como dito anteriormente, o processo de testes deve acontecer em conjunto com o processo de desenvolvimento de *software*, seguindo suas fases e avaliando os produtos à medida que ficam prontos. Assim, mesmo que o Modelo V esteja desenhado a partir do processo cascata, todos os demais processos de desenvolvimento possuem em suas características pequenos ciclos desse processo denominados de **minicascatas**, seja o processo de desenvolvimento utilizado o espiral, o incremental ou o iterativo, por exemplo.

Após o levantamento de requisitos pela equipe de desenvolvimento, a equipe de testes entra no processo a fim de obter as necessidades do cliente e começar a identificar os casos de testes com base na especificação elaborada e no protótipo de telas produzido e validado pelo usuário. A identificação dos casos de testes deve ser feita pela equipe de testes e validada com os usuários. Essa lista de casos de testes deve ser acrescida das situações de testes fornecidas pelos usuários.

Uma vez que os casos de testes são avaliados e aprovados, inicia-se a elaboração do roteiro de testes que descreve em detalhes o passo a passo para a realização dos testes, especificando o que deve ser feito e qual o resultado esperado. O nível de detalhe deve ser suficientemente claro para ser executado por pessoa externa ao processo. Esse roteiro deve ser submetido à inspeção e à aprovação do usuário antes de ser utilizado e deve cobrir todas as situações que os usuários utilizam na fase de aceite do sistema. Esse processo é ilustrado na Figura 36.

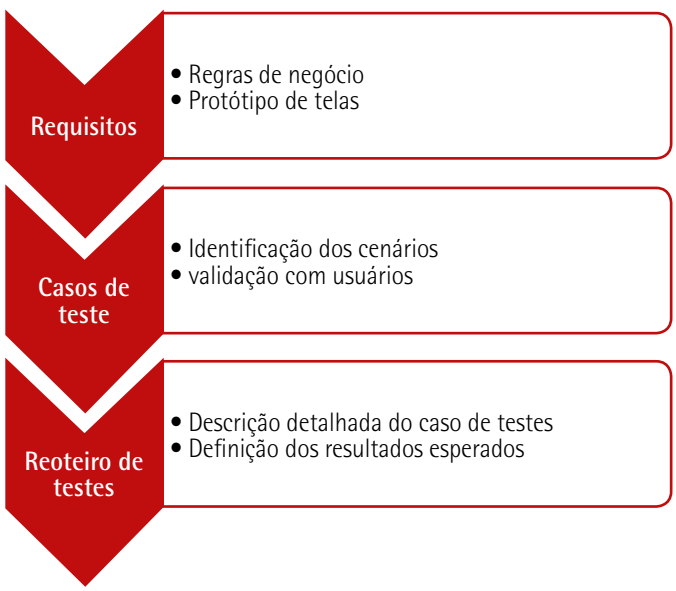


Figura 36 – Processo de criação de roteiro de testes

Durante a fase de codificação os desenvolvedores realizam testes unitários e integrados que devem garantir que os programas construídos funcionam corretamente. Os roteiros podem ser utilizados nessa fase.

Passado pelos testes iniciais, o *software* deve ser avaliado pela equipe de testes que, de posse do roteiro de testes, executa-os para garantir que o *software* esteja de acordo com as especificações dos usuários. Nessa fase devem ser gerados os registros e as evidências de que os testes foram realizados com sucesso.

Na fase final, a homologação é feita pelos usuários que fazem os testes de aceitação, normalmente, sem utilizar o roteiro, para garantir que o *software* esteja de acordo com as necessidades e execute corretamente as funções que lhe são atribuídas.

O Quadro 19 apresenta a coerência entre as fases de desenvolvimento e a fase de testes, as atividades que são realizadas nesta fase, as ações de qualidade que devem ser executadas e os membros das equipes envolvidos em cada caso.

Quadro 19 – Atividades de testes dentro do ciclo de desenvolvimento

Fase de desenvolvimento	Fase de testes	Atividade	Ações de qualidade	Participantes
Especificação	Planejamento	Estratégia e preparação do ambiente de testes	Revisão	Equipe de testes
Projeto de arquitetura	Análise	Identificação dos casos de testes	Revisão e inspeção	Equipe de projeto e de testes
Projeto detalhado	Especificação	Elaboração do roteiro de testes	Inspeção	Equipe de testes e usuários
Construção	Execução	Localização de defeitos e realização da correção	Testes unitários e integrados	Equipe de projeto e de testes
Implantação	Homologação	Correção de defeitos	Testes de sistema e aceitação	Equipe de projeto, de testes e usuários

6.2.1.1 Níveis de testes do Modelo V

Os testes são divididos em níveis para facilitar o entendimento de sua abrangência e cobertura dentro do ciclo de vida de um *software*. Esses níveis são detalhados a seguir.

6.2.1.1.1 Testes de unidade ou testes unitários

São testes realizados pelos desenvolvedores de *software* com o objetivo de garantir o funcionamento adequado do programa, do módulo, da função ou da classe que foi construída. Podem ser automatizados ou realizados por meio de ferramentas. A Figura 37 ilustra as unidades de um teste.

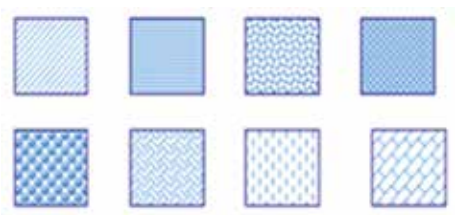


Figura 37 – Representação de unidades de testes

6.2.1.1.2 Testes de integração

São testes realizados após o teste unitário para garantir que os elementos que compõem a aplicação funcionam de forma integrada com sucesso. Feitos pelos analistas de sistemas, envolvem testes de subsistemas ou incrementos do *software*. A Figura 38 ilustra o agrupamento dessas unidades.

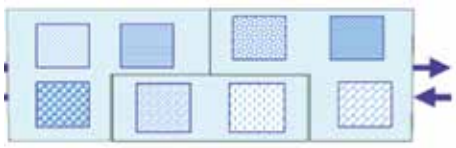


Figura 38 – Representação da integração das unidades de testes

6.2.1.1.3 Testes de validação/sistemas

Realizados após os testes de integração, visam verificar se a aplicação desenvolvida está de acordo com a especificação inicial do sistema. São realizados pelos analistas de testes.

Para melhor avaliação dos testes de validação é necessário que o ambiente seja o mais semelhante possível ao ambiente de produção e que exista um roteiro de testes, a ser seguido pelos envolvidos no teste, a fim de que os resultados sejam avaliados, registrados e evidenciados pelo sistema de controle da qualidade.

6.2.1.1.4 Testes de aceitação

São testes realizados pelos usuários finais e analistas de testes que visam garantir que todos os requisitos solicitados foram incluídos e funcionam corretamente no produto entregue. São feitos utilizando os critérios estabelecidos pelos usuários e sem roteiro preestabelecido, pois o usuário testa como se estivesse utilizando o *software* no seu dia a dia. Sua duração está condicionada à aceitação, pelos usuários, dos defeitos e pendências encontrados durante os testes, bem como aos riscos associados à liberação do *software*.

6.2.2 Testes na fase de manutenção do sistema

O processo de testes também deve ser aplicado à fase de manutenção do *software*, que consiste na inclusão ou na retirada de alguma funcionalidade do sistema ou na alteração de alguma característica preexistente. A manutenção nos *softwares* ocorre com muito mais frequência do que o desenvolvimento de novos sistemas, portanto a importância da realização de testes é tão grande quanto em novos sistemas.

Os roteiros de testes elaborados para o desenvolvimento do *software* devem ser alterados para atender às novas necessidades incluídas ou excluídas pelo processo de manutenção do sistema. Porém, não basta executar a parte do sistema que sofreu alterações: toda a aplicação deve ser testada para garantir que nenhuma funcionalidade preexistente tenha sido afetada pela mudança. Esse processo é chamado de **teste de regressão**.

Segundo Rios e Moreira (2013), é evidente que, quanto melhores forem os testes feitos durante o desenvolvimento, menores serão os custos durante a fase de manutenção.

6.3 Tipos de testes

Normalmente, a preocupação dos desenvolvedores está em realizar testes que garantam que o *software* atende às necessidades dos usuários. Porém, os testes não se limitam aos requisitos funcionais.

Os **requisitos funcionais** envolvem o correto funcionamento do *software*, sua integração com outros sistemas, suas interfaces e a garantia de que qualquer alteração feita não afete a aplicação.

Os **requisitos não funcionais** surgem principalmente na fase de projeto arquitetural e/ou detalhado, estão ligados às características comportamentais do *software* e não são solicitados pelos usuários. Esses requisitos devem ser avaliados pela equipe de desenvolvimento, validados junto aos usuários e definidos aqueles que devem ser aplicados a cada tipo de sistema.

Neste tópico serão apresentados e descritos os diversos tipos de testes mais utilizados pelo mercado de Tecnologia da Informação, tanto para os requisitos funcionais quanto para os requisitos não funcionais, e que devem ser parte integrante do *checklist* de qualidade de desenvolvimento de *software*, principalmente, para a fase de projeto.

6.3.1 Amplitude dos tipos de testes

Existem diversos tipos de testes para serem realizados. Podem ter caráter funcional ou não funcional e, no desenvolvimento ou na manutenção de um sistema, podem ser total ou parcialmente executados, dependendo das características da aplicação e do ambiente em que serão executados. Esses tipos de testes são ilustrados na Figura 39.

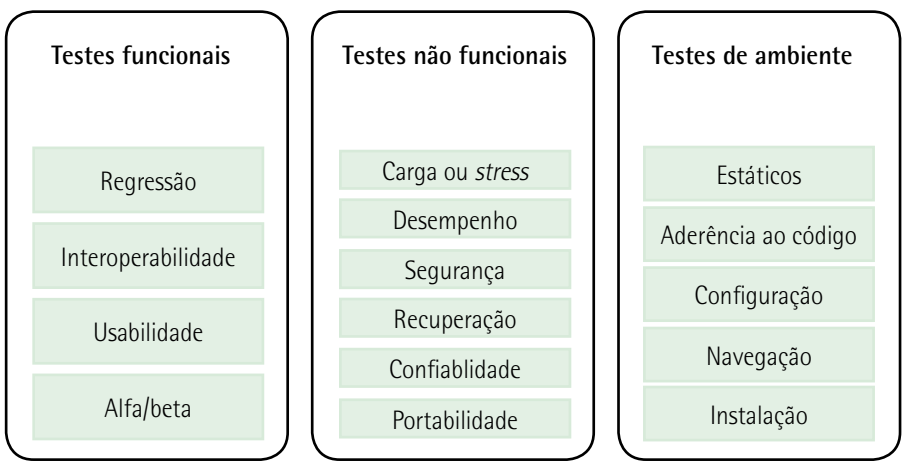


Figura 39 – Principais tipos de testes

6.3.1.1 Tipos de testes para requisitos funcionais

- Testes de regressão: têm como objetivo garantir que, mesmo após mudanças realizadas nas fases de desenvolvimento ou manutenção, a aplicação continue funcionando adequadamente e produzindo os resultados esperados.

- Testes de interoperabilidade: visam avaliar e garantir que a comunicação entre os sistemas envolvidos e entre os ambientes computacionais da aplicação estejam funcionando adequadamente. Por exemplo: ao acessar um sistema de validação de CEP, este retorna valores válidos.

- Testes alfa e beta: são testes executados após a aplicação estar pronta e feita pelos usuários finais da aplicação. Os roteiros de testes não são seguidos, e o objetivo é identificar o maior número de defeitos possível antes de liberar a aplicação para a produção.

- Testes de usabilidade: avaliam a facilidade de uso da aplicação quanto à visão dos dados na tela, às cores, à disposição dos componentes de interação dos usuários, à facilidade de interpretação das mensagens e ao conteúdo e à clareza das informações nas telas.

6.3.1.2 Tipos de testes para requisitos não funcionais

- Testes de carga ou *stress*: são testes que têm como objetivo avaliar o comportamento da aplicação sob condições extremas de acessos simultâneos ou de requisições ao servidor para verificar se suporta o volume esperado. Em razão do volume de requisições que precisam ser geradas, esses testes são realizados por ferramentas específicas para esse fim. Exemplo: a aplicação deve suportar até mil usuários realizando *login* ao mesmo tempo ou responder a até trezentas requisições por minuto.

- Testes de desempenho ou *performance*: têm por objetivo verificar se a aplicação responde às requisições dos usuários dentro do tempo esperado. Também são testes automatizados. Por exemplo: a cada solicitação do usuário, o sistema deve responder em, no máximo, 5 segundos.

Saiba mais

Existem diversas ferramentas para testes de carga e testes de desempenho. Um dos mais utilizados é o JMeter, que pode encontrado no *site*: <http://jakarta.apache.org/jmeter>.

- Testes de segurança: verificam a capacidade da aplicação de lidar com as tentativas não autorizadas de acesso aos perfis dos usuários autorizados.

- Testes de recuperação ou *disaster recovery*: o objetivo desses testes é validar como a aplicação se comporta em situações extremas de falta de energia, queda da rede, falha de comunicação, dentre outras.

- Testes de portabilidade: visam avaliar a instalação e o funcionamento da aplicação em ambientes operacionais diferentes, previstos no planejamento.

- Testes de confiabilidade ou disponibilidade: visam avaliar o comportamento da aplicação quando submetida à falha de algum item do seu ambiente operacional, como servidor de aplicação, servidor de banco de dados, dentre outros. Nessa situação, o *software* deverá manter-se funcionando parcialmente e/ou tratar as falhas adequadamente por meio de mensagens aos usuários.

6.3.1.3 Tipos de testes para ambiente

- Testes estáticos: validam se as especificações, os diagramas de casos de uso, o diagrama de classes, o modelo de dados, dentre outros estão coerentes com a aplicação desenvolvida. São realizados por meio de revisões técnicas formais ou inspeções.

- Testes de aderência de código: visam avaliar se o código-fonte da aplicação está de acordo com os padrões e as melhores práticas previstas no processo de qualidade. São executados por meio de ferramentas ou de inspeções.

- Testes de configuração: avaliam se o *software* se comporta adequadamente de acordo com as especificações de *hardware* e *software* planejadas. Por exemplo: o *software* funciona nas versões x e y do *browser* Z?

- Testes de navegação: avaliam o comportamento da aplicação no que tange à navegação entre telas, aos *links* para outros sistemas, dentre outros.

- Testes de instalação: verificam se o plano de instalação do sistema está correto e se a aplicação funciona após a instalação em um novo ambiente.

6.4 Técnicas de testes

Como visto até aqui, os testes são essenciais no processo de desenvolvimento e devem ser aplicados nas várias fases do ciclo de vida do *software*. As técnicas de testes mais utilizadas no mercado definem basicamente duas estratégias: uma baseada na especificação funcional da aplicação e outra focada em avaliar a qualidade do código produzido. A primeira é mais direcionada aos testes de sistema e de aceite, enquanto a segunda é utilizada nos testes de unidade e de integração.

A definição de qual técnica deve ser utilizada depende de vários fatores, mas o mais determinante é o tempo disponível para a aplicação desses testes dentro do prazo do projeto.

Entrando em maiores detalhes, as principais técnicas de testes estão ilustradas na Figura 40.

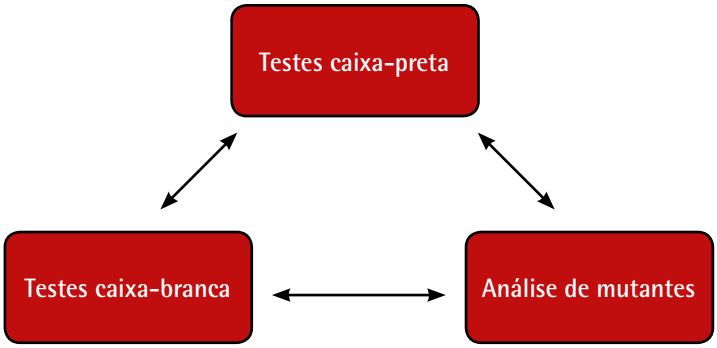


Figura 40 – Principais tipos de testes

6.4.1 Técnica de testes funcionais ou caixa-preta

Baseada na especificação inicial do sistema, elaborada na fase de levantamento de requisitos, como as especificações de casos de uso e o protótipo de telas, são extraídas as situações de sucesso e insucesso na execução de determinadas funcionalidades que são chamadas de casos de teste.

- Técnica de teste estrutural ou teste caixa-branca: realizada sobre o código escrito pelos desenvolvedores para garantir a qualidade do código produzido.

- Técnica de teste baseada em erros: feita com base nos erros típicos cometidos durante o desenvolvimento de *software*. Fundamenta-se em duas abordagens básicas. Uma delas é relativa à semeadura de defeitos, em que erros mais comuns de programação são inseridos no código, sem o conhecimento do testador, para que sejam encontrados. A outra abordagem é a chamada análise de mutantes, em que cópias do programa a ser testado são criadas e condições de erro são incluídas, para verificação. São mais utilizadas na fase de testes unitários e testes de integração, mas seu uso é bastante reduzido no mercado e não são detalhadas neste livro.

6.4.2 Técnicas de teste estrutural ou caixa-branca

O teste estrutural ou caixa-branca é focado em avaliar a qualidade do código produzido pelos desenvolvedores, garantindo que toda linha de código escrita seja executada pelo menos uma vez. Para isso, são identificadas todas as condições de controle do programa (*if*, *while*, *repeat*, *case*, *switch*, dentre outras) e gerada a massa de testes necessária para a verificação do código-fonte em cada uma das situações mapeadas.

Observação

Massa de testes refere-se ao conjunto de dados de entrada em um teste que produz um resultado esperado pelo testador.

Como os testes estruturais são focados na codificação, sua aplicação pode fazer parte das inspeções de código, em que, além de verificar a aderência do código aos padrões e às boas práticas de programação, devem-se avaliar as diversas condições previstas no programa.

Para auxiliar na identificação dos caminhos de programação exclusivos e necessários para testar cada linha de código, existe uma técnica de mapeamento chamada **grafo de fluxo de controle**.

6.4.2.1 Grafo de fluxo de controle

Trata-se de uma representação gráfica que permite visualizar a complexidade dos caminhos de um programa, independentemente da linguagem de programação utilizada. O grafo é formado por nós que são ligados por arestas mediante setas que mostram sua direção. Os nós representam blocos de comandos. O bloco de comando é um conjunto de instruções no qual, ao executar o primeiro comando, todos os comandos seguintes também são executados. As arestas informam a direção e a sequência em que os nós são chamados.

Além das estruturas básicas de nós e arestas, o grafo de controle possui os chamados **nós predicados**, que são aqueles de onde partem pelo menos duas arestas.

6.4.2.2 Representação gráfica das estruturas básicas de controle

A seguir são apresentadas as estruturas básicas do grafo de controle.

- Sequência: são comandos procedurais. Representação gráfica na Figura 41.

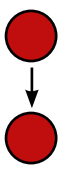


Figura 41 – Grafo de controle: sequência

- Seleção: comandos *if... then... else*; *case*; *switch*. Representação gráfica na Figura 42.

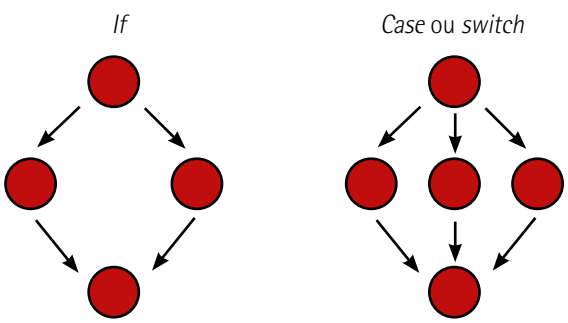


Figura 42 – Grafo de controle: seleção

- Repetição: comandos *while*...; *for*...; *do*... *until*. Representação gráfica na Figura 43.

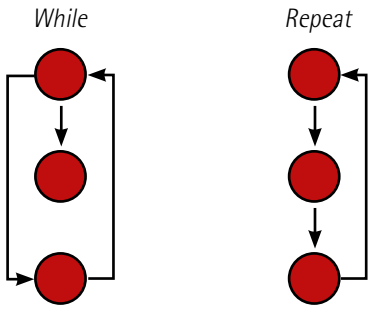


Figura 43 – Grafo de controle: repetição

Essas estruturas básicas de controle permitem a construção da lógica do programa que será avaliado nos testes de caixa-branca.

No exemplo da Figura 44, tem-se a representação de um grafo de controle para um programa com dois caminhos retirados a partir de um comando *if*. Cada número no programa representa um nó no grafo, e cada caminho possível é uma aresta.

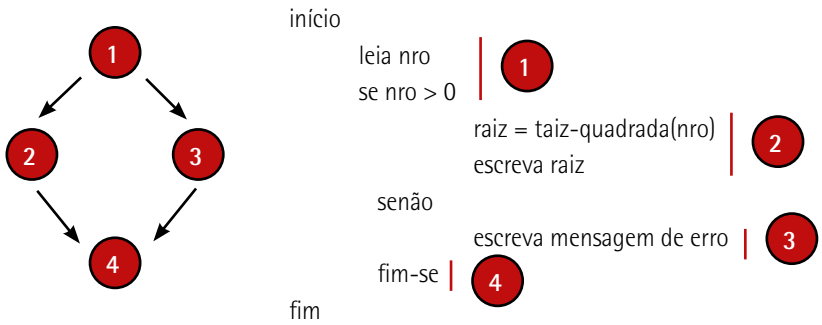


Figura 44 – Exemplo 1 de um grafo de controle

No exemplo da Figura 45, tem-se um programa um pouco mais complexo que inclui, além do comando *if*, um comando *while*. No grafo, a bolha 1 representa um comando sequencial e as bolhas 2 e 7 são da repetição, enquanto as bolhas 3, 4 e 5 e 6 representam o *if*.

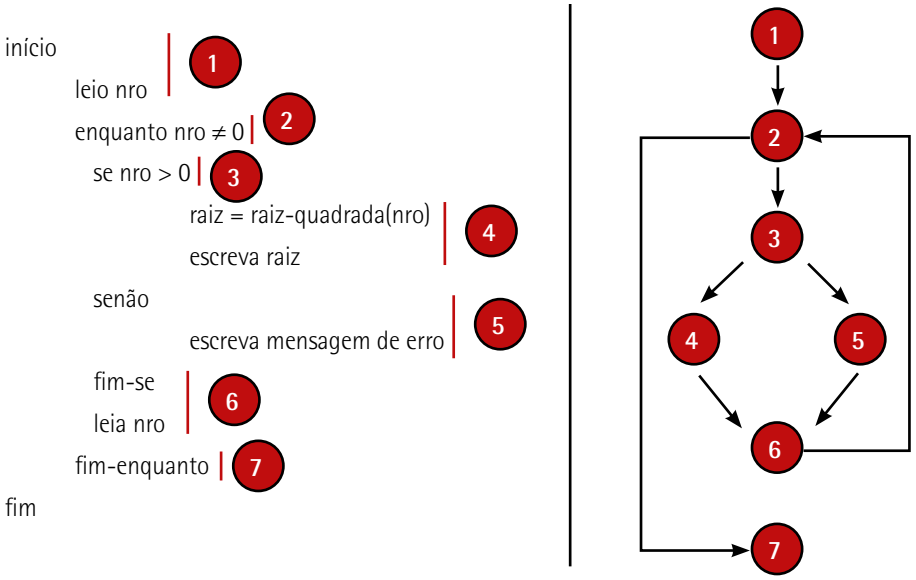


Figura 45 – Exemplo 2 de um grafo de controle

6.4.2.3 Testes de caminhos independentes

Com a construção do grafo de controle é possível determinar os caminhos de testes básicos do programa, chamados de **caminhos independentes**.

Caminho independente é aquele que contém pelo menos uma nova aresta no grafo de controle e garante que todo comando será executado pelo menos uma vez. No exemplo da Figura 46, cada caminho possui ao menos um nó exclusivo.

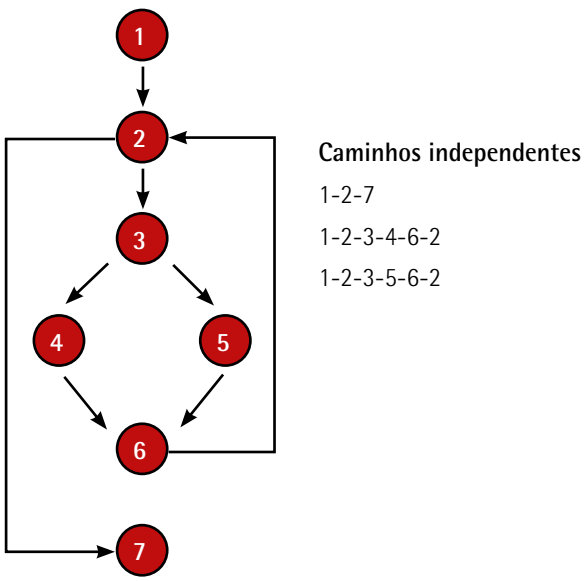


Figura 46 – Identificação de caminhos independentes

6.4.2.4 Complexidade ciclomática – V(G)

Desenvolvida por Tom McCabe em 1976, baseada no grafo de controle, tem como objetivo medir quantitativamente a complexidade lógica de um programa e fornecer o limite superior para o número de caminhos independentes, que determina a quantidade de testes necessários para garantir que todas as linhas de código sejam executadas pelo menos uma vez.

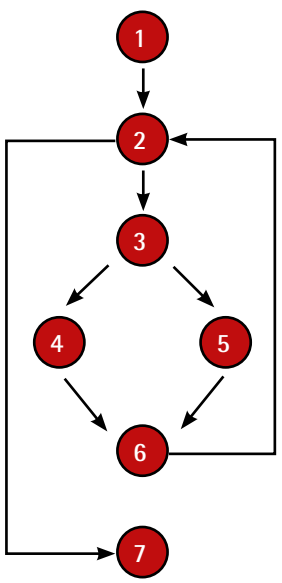
Existem duas formas para se calcular a complexidade ciclomática:

- **V(G) = (E – N)+2**, onde:
  E é o número de arestas.
  N é o número de nós do grafo.

- **V(G) = P + 1**, onde:
  P é o número de nós predicados do grafo.

Para o grafo de controle ilustrado na Figura 47, existem sete nós, oito arestas e dois nós predicados, que são os nós 2 e 3, de onde sai mais de uma aresta. Nesse caso, tem-se a determinação de três caminhos independentes para testar, conforme apresentado na Figura 46.

Ainda segundo McCabe (1976), a partir do cálculo da complexidade ciclomática, é possível determinar o grau de risco de se testar um programa, ou seja, se o código ultrapassar a V(G) de 20, significará que existe um alto grau de chance de que não seja possível testá-lo adequadamente e de que ele venha a falhar em determinado momento.



Complexidade ciclomática
V(G) = (E – N) + 2
V(G) = (8 – 7) + 2 => **V(G) = 3**
Ou
V(G) = P + 1
V(G) = 2 + 1 => **V(G) = 3**

Figura 47 – Cálculo da complexidade ciclomática

Atualmente, com a programação orientada a objetos, os programas estão muito reduzidos em linhas de código, limitando-se a métodos de uma classe. Com esse conceito, a complexidade

ciclomática tende a ser baixa, facilitando os testes caixa-branca, mas aumentando a complexidade dos testes caixa-preta.

A Tabela 6 mostra a classificação de riscos de acordo com a complexidade ciclomática de um programa.

Tabela 6 – Classificação de riscos de acordo com a V(G)

Complexidade	Avaliação de risco
1 – 10	Programa simples, sem risco
11 – 20	Programa médio, risco moderado
21 – 50	Programa complexo, alto risco
> 50	Programa não testável

Fonte: McCabe (1976).

6.4.2.5 Definindo os casos de testes caixa-branca

Elaborado o grafo de controle e calculados a V(G) e o número de caminhos independentes, resta determinar os casos de testes necessários para verificar o código do programa em questão. Para isso, seguem-se os quatro passos básicos:

1. Desenhar o grafo de fluxo correspondente ao programa (Figura 48).

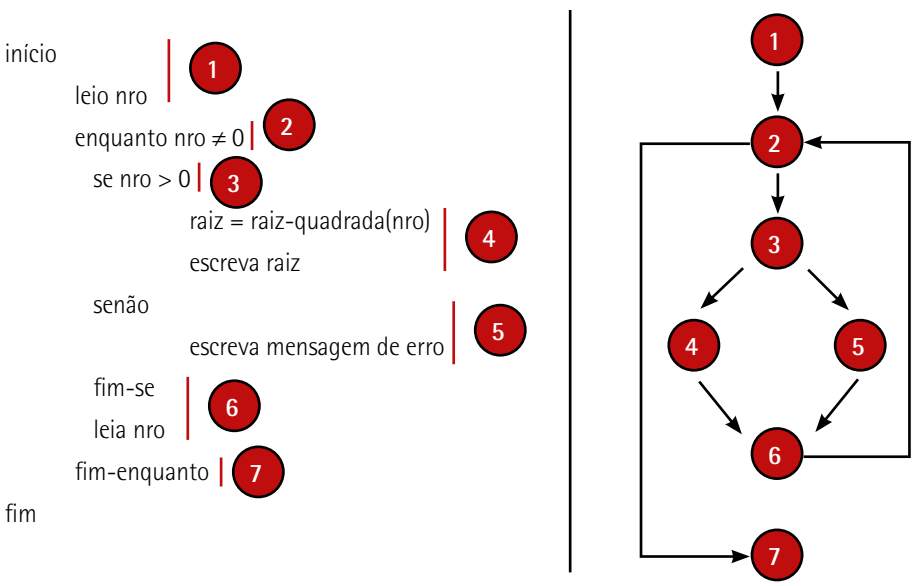


Figura 48 – Passo 1: desenhar o grafo de controle

2. Determinar a V(G) do grafo de fluxo correspondente:

V(G) = (E – N) + 2 ou V(G) = P + 1

V(G) = (8 – 7) + 2 => **V(G) = 3** V(G) = 2 + 1 => **V(G) = 3**

3. Determinar os caminhos independentes. São eles:

1-2-7

1-2-3-4-6-2

1-2-3-5-6-2

4. Preparar os casos de teste para executar cada caminho:

Tabela 7 – Preparação dos casos de teste

Caminho	Dado de entrada	Resultado esperado
1-2-7	Zero	Sai do programa sem executar nada
1-2-3-4-6-2	2 e 4	Calcula a raiz quadrada de 2 e depois de 4
1-2-3-5-6-2	4 e–20	Calcula a raiz quadrada de 4 e escreve mensagem de erro para–20

O teste estrutural conforme apresentado aqui é pouco utilizado no mercado, em função da falta de conhecimento da técnica pela maioria dos desenvolvedores e, principalmente, pela falta de processo de qualidade voltado para o código produzido.

6.4.3 Técnica de teste funcional ou caixa-preta

Os testes funcionais são os mais amplamente utilizados no desenvolvimento de *software*. Focadas nas necessidades ditadas pelos usuários e transformadas em requisitos pelos analistas de sistemas, as situações de testes criadas devem atestar que o *software* faz exatamente o que foi solicitado e que funciona corretamente.

A única desvantagem para os testes funcionais é que não é possível garantir que a especificação esteja 100% correta, mesmo com as validações dos usuários. Quando isso ocorre, podem aparecer surpresas durante os testes de aceite realizados pelos usuários, em que surgem novos comportamentos esperados ou novas funcionalidades não previstas pela aplicação e que são necessárias para esta. Em tal situação, essas novas necessidades também não são mapeadas pelos testes funcionais.

Para garantir a elaboração de bons testes funcionais, devem ser obtidos, no mínimo, os seguintes artefatos validados pelos usuários:

- a especificação dos requisitos, como uma especificação de casos de uso ou um documento de requisitos;

- um protótipo de telas visual com as respectivas especificações detalhadas de seu comportamento.

De posse desses artefatos, a equipe de testes pode iniciar suas atividades de planejamento, preparação e execução dos testes e definir o oráculo de testes, ou seja, constituir a referência para dizer se os resultados produzidos pela aplicação estão corretos ou não. A Figura 49 apresenta um processo de elaboração dos testes funcionais.

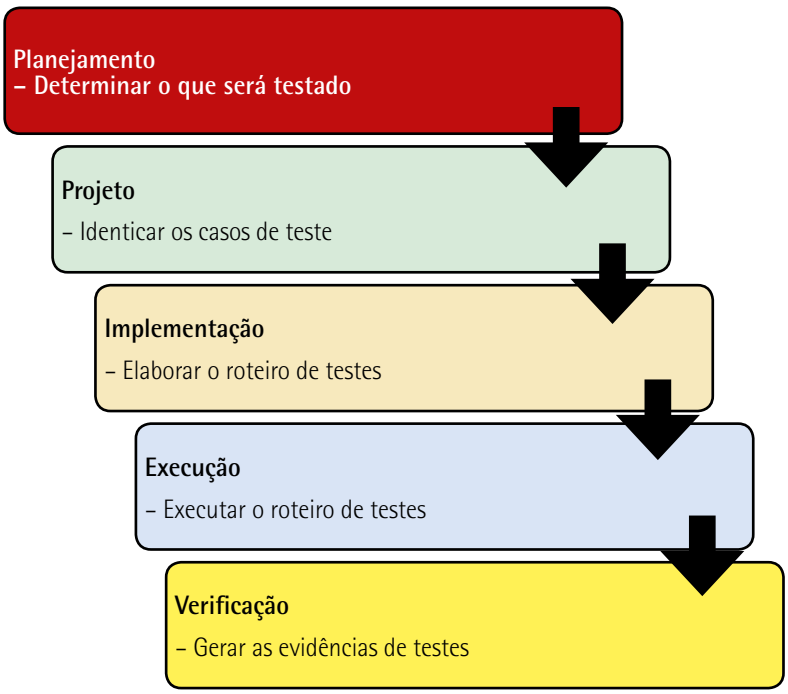


Figura 49 – Passos para a elaboração dos testes funcionais

Para a elaboração dos testes funcionais, duas atividades devem ser desenvolvidas e devidamente validadas com os usuários do sistema:

- especificar os casos de testes ou cenários de testes;

- elaborar o roteiro de testes.

6.4.3.1 Casos de testes ou cenários de testes

Um caso de teste ou um cenário de teste é uma situação que o sistema apresenta para a qual, dada uma informação de entrada, será gerada uma saída esperada pelo usuário. Por exemplo, você quer testar um *software* para uma calculadora simples com quatro operações. Todas as operações precisam ser testadas, e, para cada uma, um conjunto de valores precisa ser verificado para constatar se está fazendo o cálculo correto. Nessa situação, os casos de testes para operação de soma seriam:

- 2 + 2 = 4 – soma de dois números inteiros positivos.
- -2 + 2 = 0 – soma de número inteiro positivo e negativo.
- -2 – 2 =–4 – soma de dois números.
- 2 + 0,5 = 2,5 – soma de um número inteiro com decimal positivo.
- 2 +–0,5 = 1,5 – soma de número inteiro com decimal negativo.
- - 2 +–0,5 =–2,5 – soma de número inteiro negativo com decimal positivo.

E assim sucessivamente até que todas as situações possíveis possam ser verificadas e validadas com as combinações envolvendo dois números e com aquelas envolvendo três números, quando todas as possíveis operações estiverem dentro da chamada cobertura dos testes. Quanto maior a cobertura dos testes, melhores os resultados da qualidade.

Outro exemplo, agora relacionado a uma aplicação transacional: imagine que você queira testar o cadastro de um cliente para uma locadora de *games* (o sistema deve incluir, alterar, excluir e consultar clientes). As regras informadas pelos usuários dizem que o CPF deve ser válido, o endereço deve ser obtido por meio do CEP e o cliente não pode possuir pendências de dívidas no órgão XPTO. Para a situação descrita, os seguintes casos de testes devem ser identificados:

- incluir cliente com sucesso;
- alterar cliente com sucesso;
- excluir cliente com sucesso;
- consultar cliente com sucesso;
- consultar cliente por CPF inexistente;
- incluir cliente com CPF inválido;
- alterar cliente com CPF inválido;
- incluir cliente com CEP inválido;
- alterar cliente com CEP inválido;
- incluir cliente com pendência no órgão XPTO.

Contudo, de onde vamos obter as informações para criar esses casos de testes? Normalmente, usam-se a história do cenário, como descrito nos exemplos, a especificação de requisitos do sistema, a especificação de casos de uso e o protótipo de telas. A partir desses artefatos, identificam-se os cenários de sucesso, o que significa realizar os testes sem erros, e as diversas situações de exceção que podem ocorrer durante o uso da aplicação, conforme o exemplo do cadastro de cliente e os cenários de CPF inválido, CEP inválido e pendência no órgão. Vale ressaltar que o objetivo dos casos de testes é identificar e criar cenários de teste relacionados às regras de negócio da aplicação.

Lembrete

Se utilizar a especificação de casos de uso para criação dos casos de testes, o cenário de sucesso será o fluxo básico, e os cenários de insucesso serão os fluxos alternativos. Além disso, criar casos de testes para validar as precondições de cada caso de uso.

A elaboração dos casos de testes pode ser feita pela equipe de testes de forma independente, e esses devem ser claros e objetivos. Após a elaboração da lista de casos de testes, essa precisa ser avaliada e validada pelos usuários do sistema para se ter a maior cobertura possível das situações de testes.

Com esses casos de testes prontos, pode-se avançar para a elaboração do roteiro de testes.

6.4.3.2 Roteiro de testes

O roteiro de testes é uma descrição detalhada do passo a passo para a execução do sistema, a fim de verificar cada caso de teste identificado na fase anterior. Para cada caso de teste deve haver um roteiro contendo as seguintes informações:

- nome do caso de testes;
- procedimento inicial para determinar onde começa o teste;
- descrição detalhada contendo:
  - passos para a execução;
  - dados de entrada;
  - resultado esperado;
  - situação (sucesso ou não);
  - data da realização;

– usuário que realizou o teste.

- evidência de realização do teste.

Observe que nos dois exemplos são informados os dados de entrada para que a operação seja verificada por meio do dado de saída. O que são esses dados de entrada e o resultado esperado?

- Procedimento inicial: descreve detalhadamente as atividades que precisam ser feitas antes de iniciar a execução dos testes.

- Dado de entrada: é a informação que deve ser inserida no sistema para satisfazer o passo descrito e gerar a saída esperada.

- Resultado esperado: é o dado de saída gerado pelo sistema após a execução do passo e que serve de avaliação para indicar o sucesso da realização do passo.

- Evidência de teste: é a informação que pode mostrar ao usuário que o caso de teste foi realmente executado. Pode ser uma "foto" da tela, um arquivo, dentre outros.

Ao elaborar o roteiro de testes, deve-se observar a integração do roteiro com a tela associada à funcionalidade e ao uso cotidiano pelos usuários para que sua execução seja intuitiva e objetiva. Os passos devem ser descritos com detalhes suficientes para que não causem dúvidas ao testador e este possa fazer a análise correta do resultado do teste. Além disso, é importante que os casos de testes tenham poucos passos, para serem de fácil compreensão.

O roteiro, assim como as especificações, não é um documento estático do projeto. As evoluções e mudanças que ocorrem durante o desenvolvimento devem ser informadas à equipe de testes para a manutenção dos roteiros de testes alinhados com as especificações de requisitos e para evitar divergências e conflitos internos.

As informações descritas sobre a situação em que o passo se encontra (sucesso ou insucesso), a data do teste, o usuário que o realizou e a evidência de testes são atualizados quando da execução efetiva do roteiro de teste, não fazendo parte da elaboração desse roteiro.

A Tabela 8 ilustra o plano de um roteiro de testes para o caso de teste Consultar cliente com sucesso descrito no exemplo.

Tabela 8 – Exemplo de um roteiro de testes

Caso de teste: consultar cliente com sucesso			
Procedimento inicial: acessar a URL <xxxxx> como usuário administrador, acessar o menu Cadastros, acessar a opção Cliente e clicar em Consultar			
ID	Passo para execução	Dado de entrada	Resultado esperado
1	Sistema exibe tela para pesquisa por nome ou CPF	–	Dados exibidos: campos, nome e CPF
2	Usuário informa CPF válido e clica em Buscar	111.111.111-11	Sistema exibe dados do cliente: nome: xxxx, endereço: xxxxx, cidade: xxxx, estado: xx e país: xxxxx
3	Usuário clica em Nova Busca	–	Tela é limpa, e retorna a tela de pesquisa

Durante a execução do roteiro de testes, os defeitos podem e devem ser encontrados na aplicação. Esses defeitos precisam ser registrados e controlados pela equipe de testes e, após o término do ciclo de testes, a equipe de desenvolvimento analisa e verifica se os apontamentos são procedentes. Em caso afirmativo, faz as correções e encaminha a aplicação para novo ciclo de testes. Esse ciclo se repete até que o roteiro de testes esteja 100% executado, e os resultados, corretos.

Saiba mais

São boas ferramentas para gerenciamento dos defeitos encontrados durante os testes funcionais, o Mantis – <http://www.mantisbt.org> e o Jira – <http://www.atlassian.com/software/jira>.

6.4.3.3 Testes funcionais de interface

Além dos casos de testes relacionados às regras de negócio abordadas até aqui, existem os casos de testes relativos ao comportamento técnico das telas ou interfaces. Esses casos de testes são importantes para garantir que a interface faça as verificações necessárias para tornar o *software* mais robusto e confiável com os dados de entrada.

Esses casos de testes devem ser identificados a partir da própria interface ou da especificação de interface gerada para a aplicação e devem ser executados desde a fase dos testes de unidade até os testes de sistema. A participação dos usuários na definição desses testes é essencial para evitar solicitações de mudanças na interface no momento dos testes de aceite, e a realização desses testes pelos desenvolvedores reduz substancialmente os apontamentos de defeitos nos testes de sistema.

Os principais casos de testes de interface estão relacionados a:

- reconhecer os atributos de cada campo;
- identificar e obter as regras de validação de cada campo;

- validar a navegação;
- validar as mensagens que serão exibidas.

Para criar os casos de testes de interface, obedeça aos seguintes passos:

- identifique os campos e componentes da interface;
- para cada campo identificado:
  - coloque o nome, o tipo do campo, o tamanho, o formato, as validações e se é obrigatório ou não.
- identifique os eventos que podem ser disparados pelos componentes (*links*, botões, caixas de texto, listas, dentre outros);
- valide as ações que serão realizadas para cada evento;
- defina e valide as mensagens de advertência criadas.

Por exemplo, para o cenário de testes de Consultar cliente com sucesso exposto anteriormente, temos a seguinte especificação da interface:

Tabela 9 – Especificações da interface

Elemento	Descrição	Tipo/tamanho	Formato	Validação
Campo	Nome	Alfa (40)	Alinhado à esquerda	Pode estar em branco
Campo	CPF	Numérico (9)	111.111.111-11	CPF deve ser válido
Campo	Endereço	Alfa (40)	Alinhado à esquerda	–
Campo	Cidade	Alfa (20)	Alinhado à esquerda	–
Campo	Estado	Alfa (2)	–	UFs válidas do Brasil
Campo	País	Alfa (10)	Fixo	"Brasil"
Botão	Buscar	-	–	Obter dados
Botão	Nova Busca	–	–	Limpa tela
Botão	Voltar	–	–	Retorna ao menu

Temos também a seguinte especificação de mensagem:

Quadro 20 – Especificações da mensagem a ser exibida

Elemento	Descrição	Situação	Mensagem a ser exibida
Botão	Buscar	Cliente não encontrado	"Cliente não encontrado"

Com essa especificação criada para cada interface do *software* em desenvolvimento, os testes funcionais devem ser realizados seguindo as abordagens de testes a seguir:

- ortografia: verificar a escrita correta de todo o texto da tela, inclusive das mensagens;
- tamanho dos campos: verificar se os campos permitem a entrada da quantidade de dados até o limite especificado;
- formato: validar se a máscara do campo corresponde ao formato da especificação, por exemplo, o campo valor dever ser apresentado na forma 9,999,999.99, ou o CPF, no formato 111.111.111-11;
- campos obrigatórios: validar se os campos com indicação de obrigatoriedade estão sendo bloqueados;
- caracteres especiais: testar se a aplicação permite o preenchimento dos campos com &, *,%, @, :, ;, dentre outros;
- valores de domínio: testar se as regras de validação estão funcionando conforme especificado e se os valores mínimo e máximo de cada campo estão sendo respeitados;
- ordem alfabética: testar se os objetos do tipo lista e as caixas de texto estão em ordem alfabética;
- navegação: acionar todos os *links* e botões para verificar o correto funcionamento, de acordo com a especificação;
- massa de dados: testar a aplicação em relação à massa de dados para verificar o comportamento do *software* quanto a tabela vazia, tabela indisponível, falha de comunicação com o banco de dados, tabela carregada com volume similar à produção, consultas que não retornam dados, dentre outras.

Podemos observar neste tópico que os testes funcionais são variados e amplos e não são executados de forma trivial e não planejada. Sem dúvida alguma, a correta execução dos testes funcionais garante a qualidade do *software*, mas exige esforço, dedicação e comprometimento de todos os envolvidos e não apenas da equipe de testes.

6.5 Testes em processos ágeis

Antes de abordar os testes em processos ágeis, é importante entender quando surgiu e o que é um processo ágil. É importante salientar que ágil não quer dizer "desenvolver rápido e de qualquer jeito", mas sim desenvolver um *software* com foco no produto, criando o que é estritamente necessário para o entendimento da equipe e do usuário e utilizando técnicas e ferramentas para auxiliar nesse processo.

Os processos ágeis começaram a tomar forma a partir de 2001, quando Kent Beck e outros pesquisadores do desenvolvimento de *software* se reuniram para discutir alternativas aos problemas

mais comuns em um processo de desenvolvimento tradicional e propuseram quatro pilares básicos para o desenvolvimento ágil:

- foco nas pessoas e não em processos;
- *software* funcionando, em vez de documentação abrangente;
- colaboração do cliente, em vez de negociação de contratos;
- resposta às modificações, em vez de seguir um plano.

À primeira vista, esses pilares parecem direcionar para o caos, e parece que tudo o que foi pesquisado até à época não tem fundamento, mas não é bem assim.

O que esses pilares querem dizer é que a equipe de projeto é uma só; gerente, desenvolvedores, clientes, usuários, todos estão juntos para atingir um objetivo comum, que é o *software* funcionando. De nada adianta criar documentos infinitos que não são utilizados agora nem depois do *software* pronto. Cria-se o que é necessário para o entendimento e a comunicação da equipe, e as temidas mudanças de escopo existem e são bem-vindas; não haverá por que bloqueá-las ou deixá-las para depois, se forem essenciais para o cliente. Em outras palavras, para manter tudo isso orquestrado, um processo ágil é até mais organizado do que um processo tradicional.

Nessa mesma época, além desses pilares, foi criado e assinado pela equipe reunida e por Kent Beck o chamado Manifesto Ágil, o qual descreve doze princípios que, se seguidos, tornam o desenvolvimento do *software* mais rápido que a forma tradicional, com mais qualidade e satisfazendo às necessidades dos usuários. Esses doze princípios estão listados a seguir, em ordem de importância para o processo ágil:

- clientes, usuários e desenvolvedores trabalham juntos;
- O levantamento e a validação dos requisitos são feitos face a face;
- entregas contínuas de *software* funcionando, não só ao final;
- satisfação do cliente por meio das entregas contínuas;
- a evolução do projeto é feita por meio de funcionalidades entregues;
- mudanças de requisitos são bem-vindas;
- equipes autogerenciadas: todos sabem o que fazer;
- equipes multidisciplinares e capacitadas tecnicamente;
- faça o essencial, a simplicidade é o caminho;

- equipe motivada e comprometida com o processo e o produto;
- ritmo constante de desenvolvimento, não "picos" de entrega;
- a equipe busca tornar-se mais efetiva sempre; melhoria contínua.

Embora sejam princípios de um processo ágil, nada impede que esses fundamentos sejam utilizados como parte de qualquer processo de desenvolvimento tradicional, com o objetivo de melhorar os resultados e aumentar a qualidade do *software*.

Saiba mais

Mais conceitos e informações sobre os doze princípios ágeis podem ser encontrados em: <http://agilemanifesto.org>.

6.5.1 Testes ágeis

Contudo, o que tudo isso tem a ver com o processo de testes ágeis? A relação está no fato de que muda radicalmente a forma de atuação e interação do pessoal de testes com os desenvolvedores e com os usuários. Como visto no tópico anterior, o pessoal de testes, no processo tradicional, é separado da equipe de desenvolvimento do *software*; as interações ocorrem de forma pontual no ciclo de vida e em maior grau quando a aplicação está pronta e precisa ser realizado o teste de sistema, com o objetivo de encontrar e corrigir erros.

No processo ágil, o teste é um compromisso e uma responsabilidade de toda a equipe. Todos devem possuir habilidade para testar e todos trabalham em conjunto, com interação constante durante todo o ciclo de desenvolvimento do *software*. Os testadores atuam de forma que entendam o processo de negócio, criando os casos de testes e participando efetivamente do projeto desde a fase de requisitos, passando pelos testes de unidade, testes integrados, testes de sistema e testes de aceitação. O objetivo nos testes ágeis é colaborar com a solução dos defeitos e não apenas apontá-los, por meio da identificação das causas desses defeitos, para que não voltem a acontecer.

No processo ágil, não há uma fase de testes específica. Os testes são realizados à medida que a codificação termina, e o *feedback* é imediato, ou seja, o defeito é apontado e corrigido na hora. Além disso, os testadores devem acompanhar a evolução das entregas de *software* com foco na qualidade e automatizar os testes, sempre que possível.

Em alguns casos, as equipes de testes podem atuar de forma separada das equipes de projeto que utilizam um processo ágil. Essas situações podem estar relacionadas ao desenvolvimento de sistemas grandes e complexos, para aumentar a automação de testes, pela necessidade de fazer testes finais de uma versão antes de entrar em produção ou para que a equipe de testes possa ser utilizada em mais de um projeto.

6.5.2 Roteiro de testes ágeis

A elaboração do roteiro de testes para processos ágeis começa pela definição de quais tipos de testes devem ser executados, se devem ser automatizados ou manuais e os responsáveis por cada um. Para os testes automatizados, são definidas as ferramentas que devem ser utilizadas.

Definido o conjunto de testes necessários, a equipe elabora, a partir das histórias e do protótipo, os casos para os testes funcionais e para os requisitos não funcionais. A identificação dos casos de testes, obrigatoriamente, deve ser realizada pela equipe e pelos usuários, para obter a maior cobertura possível, não sendo necessária a elaboração do roteiro com o passo a passo, em razão de os testes serem realizados pela própria equipe envolvida no projeto.

Com o roteiro elaborado, os testes são executados com uma visão colaborativa entre desenvolvedores e usuários até que a aplicação esteja pronta, ou seja, até que os defeitos tenham sido removidos e a aplicação funcione de acordo com as expectativas dos usuários.

6.5.3 Processo de testes ágeis

O principal processo de testes ágil está baseado nos quadrantes de testes definidos por Crispin e Gregory (2009), que agrupam os testes em visões e auxiliam na definição do que deve ser feito em cada nível de testes dentro do ciclo de desenvolvimento de *software*.

A Figura 50 apresenta esses quadrantes, cujas características são descritas a seguir.

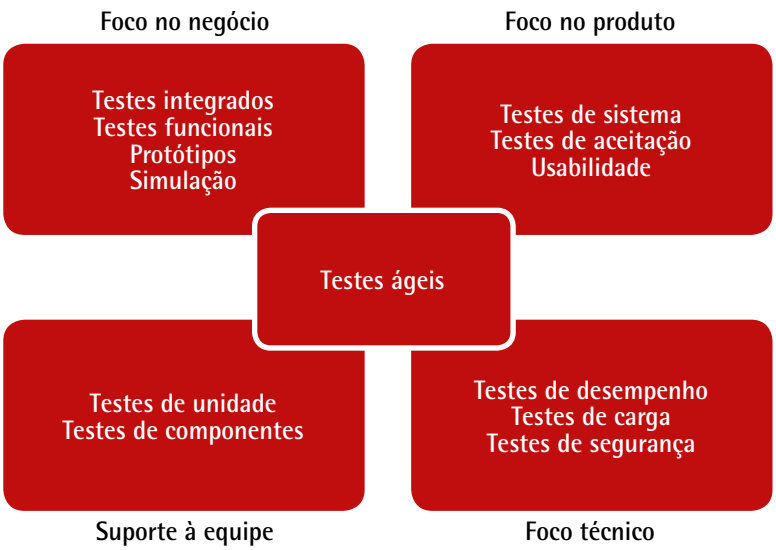


Figura 50 – Quadrantes de testes ágeis

6.5.3.1 Quadrante 1

São os testes focados no código produzido. Devem verificar a aderência aos padrões de codificação, bem como as boas práticas de programação e as de testes caixa-branca. O papel dos testes nesse

quadrante é apoiar os desenvolvedores para encontrar e corrigir os defeitos. Os testes de unidade e de componente são a menor unidade testável do *software* e devem abranger toda a ligação da arquitetura da aplicação, como camada visual, camada de negócio, acesso ao banco de dados, integração com outros sistemas, comunicação, dentre outros. Quanto mais automatizados, mas rápidos e efetivos. Podem utilizar o *Test Driven Development* (TDD) para auxiliar na elaboração.

6.5.3.2 Quadrante 2

São os testes focados no negócio. O objetivo é garantir que as regras de negócio especificadas tenham sido construídas e funcionem corretamente. Envolve os testes de integração, em que essas regras são verificadas por meio dos casos de testes criados, e os testes são executados pelo desenvolvedor, pelo testador e pelos usuários. O papel dos testes nesse quadrante é melhorar o alinhamento a respeito do negócio entre os desenvolvedores e os usuários, por meio de testes simulando situações reais de utilização da aplicação. Aqui os testes podem ser parcialmente automatizados, mas a maior parte é manual. Também podem utilizar o *Behavior Driven Development* (BDD) para auxiliar na elaboração dos testes seguintes, uma estrutura de linguagem natural que facilita o entendimento de todos e funciona como um caso de teste dentro do processo ágil de desenvolvimento.

Para elaborar um bom BDD, o testador do processo ágil deve seguir esta estrutura básica para cada história descrita no cenário de testes considerado:

- Teste a ser executado:
  - **funcionalidade**: <descrição da funcionalidade>;
  - **como um** <usuário/ator>;
  - **eu quero** <meta a ser alcançada>;
  - **de modo que** <a razão para alcançar a meta>.
- Caso de teste:
  - **cenário**: <descrição do teste>;
  - **dado** <um estado conhecido>;
  - **quando** <um determinado evento ocorre>;
  - **então** <isso deve ocorrer>.

Por exemplo, para o cenário de testes do cadastro de clientes relativo à consulta de cliente com sucesso por CPF:

- Teste a ser executado:
  - **funcionalidade**: consultar clientes;
  - **como um** funcionário;
  - **eu quero** consultar um cliente cadastrado com sucesso por CPF;
  - **de modo que** eu possa garantir que a consulta está correta.
- Caso de teste:
  - **cenário**: consultar cliente com sucesso por CPF;
  - **dado**: ao selecionar a opção Buscar;
  - **quando**: digitado o CPF 111.111.111-11;
  - **então:** deve listar nome, CPF e data de nascimento do cliente.

6.5.3.3 Quadrante 3

São os testes com foco no produto final da aplicação, os mais próximos dos testes realizados no processo tradicional. A ideia é criar situações nos produtos para permitir os testes de sistema e os testes de aceitação. Também são apoiados pelos casos de testes do roteiro elaborado, mas podem ser aleatórios. Esses testes simulam a visão do usuário final e são realizados após uma versão estar fechada para ser entregue ao usuário. Os testes desse quadrante devem ser executados em todo o sistema, principalmente, nas áreas mais cruciais para o negócio. Devem-se avaliar a usabilidade do *software* e os testes de navegação para garantir que tudo esteja funcionando adequadamente, conforme o esperado pelos usuários.

O papel dos testes nessa etapa é ganhar a confiança do cliente e envolvê-lo na entrega, conseguindo a sua avaliação prévia e conjunta a respeito da qualidade do *software* e, principalmente, corrigir defeitos que possam causar danos ao usuário quando em produção. Esses testes são manuais e devem ser realizados por pessoal experiente e com conhecimento do negócio.

6.5.3.4 Quadrante 4

São os testes com foco nos requisitos técnicos, ditos requisitos não funcionais. Abrangem os testes de carga, segurança, desempenho, recuperação, confiabilidade, dentre outros. Os padrões de qualidade esperados para esses testes precisam ser levantados, e sua execução é dependente do uso de ferramentas, pois não são executados de forma manual. O papel dos testes é apontar deficiências técnicas do produto, e requerem especialistas para sua execução. Para obter melhores resultados com esses testes, é preciso definir claramente a necessidade do tipo de teste, montar o ambiente adequado

para simular as situações necessárias, criar a massa de testes para a replicação e definir as ferramentas que devem ser utilizadas.

6.5.4 Conceitos sobre *Test Driven Development* (TDD)

O TDD é uma técnica utilizada para a realização de testes unitários e de integração que começa pela identificação dos casos de teste. Em seguida, faz-se a escrita do código necessário para atender ao caso de testes, e, se necessário, a refatoração do código para acomodar eventuais mudanças que possam ocorrer. É um método para construir *software* direcionado ao programador, com o objetivo de melhorar a qualidade do código produzido. Vem do conceito ágil da *Extreme Programming* de "testar primeiro que programar". Basicamente, o processo TDD funciona como ilustrado na Figura 51.

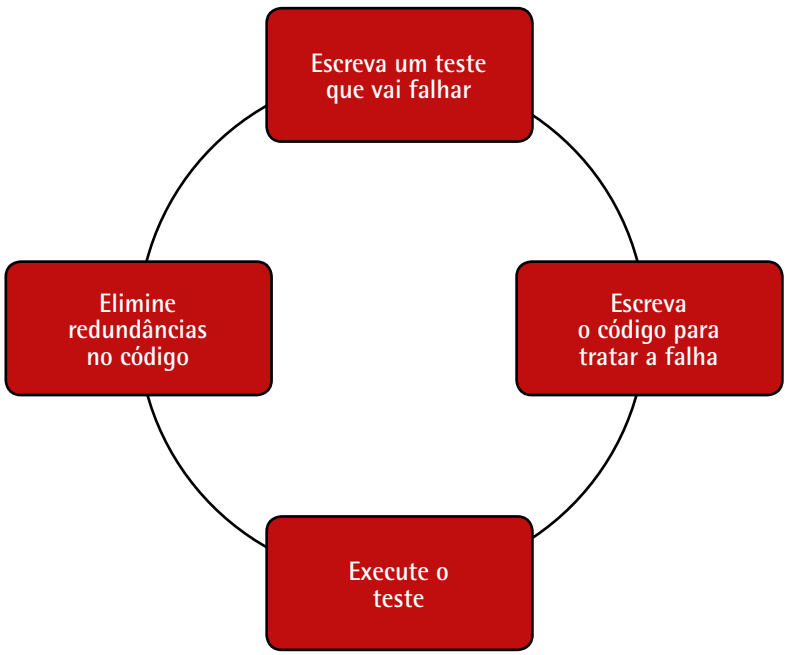


Figura 51 – Processo TDD

Os principais benefícios que podemos obter com a utilização do TDD no processo de desenvolvimento são:

- melhorar a qualidade do código;
- garantir a criação dos testes unitários;
- permitir testes contínuos sempre que o programa for alterado.

Saiba mais

Mais detalhes de como utilizar o TDD podem ser encontrados no livro:

BECK, K. *TDD*: desenvolvimento guiado por testes. São Paulo: Bookman, 2010.

6.5.5 Quando terminar os testes?

Um dos principais desafios é determinar quando os testes são suficientes para garantir que o *software* atende às necessidades dos usuários. A maior parte dos testes termina em virtude da pressão de tempo que os usuários impõem à equipe de desenvolvimento, muitas vezes, causando entregas desastrosas, em razão da falta de qualidade do produto final e também em virtude dos custos da realização de todos os testes necessários para garantir a qualidade. Como visto anteriormente, não são poucos e demandam tempo, dinheiro e especialistas no assunto.

Considerando um cenário em que essas duas variáveis não afetem o término dos testes, pode-se dizer que os testes terminam quando:

- o número de defeitos está dentro de uma margem aceitável;
- a frequência em que os defeitos são encontrados é pequena;
- houver a garantia de que os testes-caixa foram realizados;
- todos os casos de testes tiverem sido realizados com sucesso.

Enfim, determinar o fim dos testes não é uma tarefa isolada da equipe de desenvolvimento. Cabe uma decisão conjunta entre a equipe e o cliente para determinar se as condições atuais do *software* são aceitáveis e os riscos desse aceite para a operação.

Resumo

Nesta unidade foram abordadas as principais técnicas de verificação e validação de um produto de *software*, com o objetivo de garantir e melhorar a qualidade do *software* em desenvolvimento.

Abordamos que as técnicas de verificação têm por objetivo avaliar os artefatos produzidos à medida que são construídos, visando reduzir o retrabalho em caso de ajustes e reduzir o impacto de corrigir o que já está pronto. Vimos que as atividades de verificação podem ser reuniões técnicas

formais ou não, revisão por pares ou inspeções e são essenciais para a garantia da qualidade de todos os artefatos produzidos durante o ciclo de desenvolvimento e não só ao seu final.

Aprendemos ainda que as técnicas de validação visam avaliar os artefatos produzidos contra uma especificação inicial que serve de referência para validar se o que foi produzido atende àquelas necessidades especificadas. Normalmente são utilizados quando o produto de *software* está pronto. As atividades de validação podem ser inspeções, sala limpa, testes caixa-branca e testes caixa-preta. Lembrando que a especificação inicial deve ser validada pelo usuário antes da utilização pela equipe do projeto, caso contrário poderá tornar-se uma referência incorreta e não aceita pelos usuários, causando erros no processo de validação.

Vimos também que as revisões por pares ou passeios (*walkthroughs*) são a técnica mais simples para verificar a qualidade de um produto. Trata-se de uma técnica essencial para atender a um princípio básico da qualidade, que é: "Quem faz não revisa". Com a revisão por pares, os artefatos sofrem críticas em tempo de construção, permitindo a rápida correção de eventuais desvios encontrados. A revisão por pares pode ser formal ou informal, sendo esta última a mais utilizada pelas equipes de desenvolvimento. O objetivo é encontrar erros e defeitos nos artefatos.

Em seguida, abordamos que as revisões técnicas formais também são técnicas de verificação e envolvem cenários de planejamento e preparação antes de uma sessão de revisão. Envolvem pelo menos três participantes (autor, revisor e mediador), com o objetivo de avaliar os artefatos quanto ao formato (capítulo, título, subtítulo, ortografia etc.) e quanto ao conteúdo técnico do documento. Devem ser planejadas e organizadas com antecedência e não devem durar mais de duas horas.

Vimos também que a inspeção é uma técnica que pode ser usada tanto para a verificação como para a validação. Esta é a mais formal das reuniões de avaliação dos artefatos e envolve, pelo menos, cinco participantes: o autor, o inspetor, o leitor, o escritor e o moderador. Os artefatos que são objeto da inspeção e a especificação de referência devem ser enviados aos inspetores com antecedência, para a preparação da reunião. Todos os participantes têm o papel de inspetor e devem ser treinados e capacitados tecnicamente no assunto envolvido na inspeção. O processo de inspeção é iniciado pelo leitor, que faz a leitura do artefato por partes, para que os inspetores façam os questionamentos e apontamentos. O autor responde às questões, e o escritor registra as informações da inspeção. O moderador tem o papel de resolver conflitos entres as partes, e o *checklist* é um excelente apoio durante a sessão.

Abordamos que a técnica sala limpa tem origem no processo de produção de componentes eletrônicos, e sua fundamentação é matemática e estatística. O processo sala limpa é indicado para sistemas que podem trazer riscos a vidas humanas, como controle de aviões, trens, usinas, dentre outros, e sua especificação deve ser precisa e sem ambiguidades, com formato matemático.

Aprendemos que a verificação se dá por meio de inspeções rigorosas, provas matemáticas e testes estatísticos para determinar a confiabilidade do sistema e que isso a torna diferente de outras técnicas. Porém, seu uso é bastante limitado, em razão dos custos, da capacitação técnica exigida, do fundamento matemático, que afasta os desenvolvedores, e da falta de maturidade das empresas de desenvolvimento de *software* para uma utilização efetiva.

Vimos ainda que, das técnicas de validação, a mais amplamente utilizada são os testes de *software* que envolvem a execução do código da aplicação para que possam ser realizados. A execução desses testes tem o mesmo fundamento: encontrar defeitos no código quando comparado com a especificação de referência.

Abordamos que existem diversos tipos de testes, com abordagem funcional (usabilidade, caixa-preta, dentre outros) e não funcional (desempenho, segurança, carga, configuração, confiabilidade, dentre outros), e ambos devem ser executados durante o processo de desenvolvimento do *software*. Além disso, existem os níveis de testes que começam com os testes unitários, com foco no código e realizados pelos desenvolvedores; os testes de integração, também realizados pelos desenvolvedores, mas com foco na verificação das regras de negócio; os testes de sistema, realizados pelos testadores e focados em validar o produto de *software*; e os testes de aceitação, realizados pelos usuários do sistema e acompanhados pelos testadores, com o objetivo de dar o aceite final ao *software* construído.

Vimos que o processo de testes mais conhecido no mercado é o Modelo V, baseado no processo de desenvolvimento, desde o seu início até a entrega final ao usuário. Este considera a equipe de testes separada do time de construção. Preconiza que a atividade de testes deva começar junto com a fase de levantamento de requisitos, para que os analistas de testes tenham conhecimento do negócio e possam gerar bons casos de testes, a partir das especificações e do protótipo de telas. A partir dos casos de teste são elaborados os roteiros de testes, que devem ser executados pelas equipes de testes durante os testes de sistema e de aceitação. Esses testes são chamados funcionais ou caixa-preta.

Aprendemos ainda que os testes estruturais ou caixa-branca são aqueles focados no código-fonte produzido e têm por objetivo garantir que todas as linhas de código sejam executadas pelo menos uma vez. Requerem que o código esteja pronto para serem executados. Utilizam uma técnica chamada de grafos de controle, que identifica os diversos caminhos independentes que o programa pode percorrer e, de posse desses caminhos, cria as massas de testes necessárias para garantir a aderência do código às situações previstas nos caminhos mapeados. São chamados de testes unitários e de testes de integração, caracterizando o ponto de partida para a qualidade de um *software*. Porém, são muito pouco utilizados, em razão das constantes pressões de prazos nos projetos e da resistência natural dos desenvolvedores a executar testes mais estruturados nessa fase.

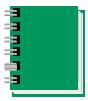
Vimos também os testes utilizados em processos ágeis, cujo objetivo principal é fazer mais rápido e com alta qualidade, com a participação de todos: desenvolvedores, usuários e clientes, para a entrega de um *software* que atenda às expectativas de todos. Os testes ágeis têm os mesmos níveis daqueles do processo tradicional e são divididos em quatro quadrantes: testes para suporte ao desenvolvedor, testes com foco no negócio, testes com foco no produto e testes com foco na tecnologia. Porém, diferem daqueles do processo tradicional pela participação efetiva do usuário na criação e na validação dos testes, utilizando o conceito "meu sucesso, seu sucesso", e na automação dos testes de unidade e de integração que proporciona excelentes resultados na qualidade final do produto.

Aprendemos que no processo ágil não há uma fase de testes específica, e a equipe não é separada do time de desenvolvimento. Os testes são realizados à medida que a codificação termina, e o *feedback* é imediato, ou seja, o defeito é apontado e corrigido na hora. Além disso, os testadores devem acompanhar a evolução das entregas de *software*, realizar os testes de aceitação junto com os usuários e garantir a qualidade dos testes técnicos, tais como os de carga, desempenho, segurança, dentre outros.

Abordamos as principais técnicas de testes ágeis. Um deles é o *Test Driven Development* (TDD), cujo objetivo é apoiar os testes unitários do desenvolvedor e facilitar a automação desses testes para facilitar mudanças posteriores. Sua estrutura básica é especificar um teste, construir o código, executar o teste e melhorar o código por meio de *refactoring*. O outro teste é o *Behavior Driven Development* (BDD), cujo objetivo é apoiar o desenvolvedor nos testes integrados com foco nas regras de negócio por meio da elaboração de casos de testes, que podem ser executados na fase de testes de sistema.

Finalmente, aprendemos que o processo de teste é a técnica mais utilizada no mercado para verificação e validação da qualidade, porém é sabido que não

podemos identificar todos os defeitos existentes somente com a realização de testes. É necessário um processo preventivo de revisão, inspeção, execução de *checklists* e definição de padrões e boas práticas de programação para que a qualidade seja parte integrante do processo de desenvolvimento do *software* e não requerer que, com a realização dos testes, se atinja a qualidade esperada somente ao final do processo de desenvolvimento.



Exercícios

Questão 1. As técnicas de verificação e validação são essenciais para manter a qualidade no processo de desenvolvimento de *software* e são chamadas popularmente de técnicas de V&V. Existem diversas técnicas, no entanto, das estudadas apenas uma utiliza como metodologia a revisão técnica informal denominada revisão por pares. O revisor pode ser um técnico, um cliente ou uma pessoa externa ao projeto, porém com amplo domínio no assunto.

Das alternativas apresentadas a seguir, assinale qual é a técnica informal que se ajusta à metodologia de revisão por pares.

A) Passeio (*walkthrough*).

B) Inspeção.

C) Sala limpa.

D) Processo de especificação.

E) RTF.

Resposta correta: alternativa A.

Análise das alternativas

A) Alternativa correta.

Justificativa: é uma técnica informal que utiliza revisão por pares, mas é possível ter até três participantes: autor, revisor e moderador. É uma técnica muito mais simples de executar quando comparada ao RTF. O alto grau de informalidade faz com que seja uma técnica bastante aplicada e facilmente aceita pela maioria das equipes de projeto.

B) Alternativa incorreta.

Justificativa: não é uma técnica informal, e sim formal, por meio da qual os envolvidos examinam os artefatos produzidos e os confrontam com uma especificação inicial com o objetivo de encontrar incoerências, inconsistências e possíveis erros.

C) Alternativa incorreta.

Justificativa: a técnica de sala limpa é baseada em especificações formais matemáticas e destinada ao desenvolvimento de *software* com alto nível de confiabilidade.

D) Alternativa incorreta.

Justificativa: não é uma técnica informal ou formal, é apenas um dos quatro processos nos quais está estruturada a técnica de sala limpa.

E) Alternativa incorreta.

Justificativa: não é uma técnica informal. O RTF é uma técnica de verificação formal.

Questão 2. A atividade de teste de *software* sempre foi considerada como um gasto de tempo desnecessário, uma atividade de segunda classe, uma vez que a programação é a atividade principal no desenvolvimento de *software*. Atualmente são muitos os desenvolvedores que ainda confundem o processo de testes com o processo de depuração de um programa. Considerando o exposto, leia as afirmativas a seguir:

I – Os testes são processos de depuração, mais conhecida como *debug*. Trata-se de um processo de busca de erros de forma não estruturada durante a execução de um programa, através de uma ferramenta de apoio ao desenvolvimento: uma vez encontrado, o erro deve ser corrigido.

II – Os testes são um processo que ocorre normalmente durante a atividade de programação e antes da execução da segunda etapa de testes propriamente dita.

III – A execução de testes é uma busca estruturada para encontrar erros em um programa pronto.

IV – Os testes devem ser realizados pelos desenvolvedores – por meio dos testes unitários; pelos testadores – por meio dos testes integrados guiados por roteiro de testes elaborados a partir da especificação inicial; e pelos usuários – por meio dos testes de aceitação, que verificam se o *software* atende às suas necessidades.

É correto apenas o que se destaca em:

A) I, II.

B) I, II, III, IV.

C) I, III.

D) III, IV.

E) I, IV.

Resposta desta questão na plataforma.