13 Teste

Este capítulo discute os *fundamentos* (Seção 13.1) da área de teste de software. Na sequência são apresentados os *níveis de teste de funcionalidade* (Seção 13.2) e os *testes suplementares* (Seção 13.3) mais frequentemente usados. Depois, são apresentadas técnicas específicas de teste, iniciando pelo *teste estrutural* (Seção 13.4) e seguido da técnica de *teste funcional* (Seção 13.5). É apresentado o modelo de processo da técnica de *desenvolvimento dirigido por testes* (*TDD*) (Seção 13.6). São discutidas métricas (Seção 13.7) a serem usadas nas atividades de teste. A atividade de depuração (Seção 13.8) e a prova de correção de programas (Seção 13.9) são brevemente comentadas ao final do capítulo.

Por melhores que sejam as técnicas de modelagem e especificação de software, por mais disciplinada e experiente que seja a equipe de desenvolvimento, sempre haverá um fator que faz com que o teste de software seja necessário: o *erro humano*. É um mito pensar que bons desenvolvedores bem concentrados com boas ferramentas serão capazes de desenvolver software sem erros (Beizer, 1990).

A Lei de Murphy (Bloch, 1977) em vários de seus enunciados, parece falar diretamente para a indústria de software. Por exemplo:

a) Se alguma coisa pode sair errado, sairá (no pior momento possível).
b) Se tudo parece estar indo bem é porque você não olhou direito.
c) A natureza sempre está a favor da falha oculta.

Durante muitos anos a tarefa de teste de software foi considerada como um castigo para os programadores. O teste era considerado como uma tarefa ingrata porque se esperava justamente que os desenvolvedores construíssem software de boa qualidade. A necessidade de testes declarava justamente esta incapacidade que era indesejada.

A área de teste podia ser caracterizada, então, por situações caricatas como as seguintes:

a) “depois eu testo”;
b) “na minha máquina funcionou...”;
c) “vamos deixar os testes para a próxima fase”;
d) “temos que entregar o produto na semana que vem” etc.

Porém, as coisas mudaram. Conforme visto em alguns modelos de ciclo de vida, a disciplina de teste passou a ser considerada extremamente importante. Hoje ela é parte integrante do processo de desenvolvimento de software. Os métodos ágeis também incorporaram o teste de software como uma atividade crítica, assumindo inclusive que os casos de teste deveriam passar a ser escritos antes das unidades de software que iriam testar.

Além disso, grandes empresas desenvolvedoras de software passaram a contratar o teste de software de forma independente (fábricas de teste). Ou seja, os responsáveis pelo teste não são mais apenas os desenvolvedores, mas equipes especialmente preparadas para executar esta tarefa.

13.1 Fundamentos

A tarefa de testar software, porém, não é simples. Em algumas situações, pode ser mais difícil elaborar bons casos de teste do que produzir o próprio software. Assim, muita sistematização e controle são necessários para que a atividade de teste de software deixe de ser uma tarefa totalmente *ad-hoc* e ingênua, para se tornar uma atividade de engenharia com resultados efetivos e previsíveis.

Esta seção apresenta os fundamentos da área de teste de software, iniciando com uma fundamentação que caracteriza os termos mais comuns para evitar possíveis confusões, sendo seguida de uma apresentação dos diferentes níveis e objetivos dos testes, bem como das principais técnicas de teste que podem ser imediatamente aplicadas, seja manualmente, seja pelo uso de ferramentas automatizadas.

13.1.1 Erro, Defeito e Falha

Inicialmente convém definir alguns termos que, caso contrário, poderiam ser considerados sinônimos, mas que na literatura de teste tem significados bastante precisos:

a) Um *erro* (*error*) é uma diferença detectada entre o resultado de uma computação e o resultado correto ou esperado.
b) Um *defeito* (*fault*) é uma linha de código, bloco ou conjunto de dados incorretos, que provocam um erro observado.
c) Uma *falha* (*failure*) é um não funcionamento do software, possivelmente provocada por um defeito, mas também com outras causas possíveis.
d) Um *engano* (*mistake*), ou *erro humano*, é a ação que produz ou produziu um defeito no software.

Cabe observar aqui que o termo *fault* (defeito) algumas vezes é traduzido como *falha*. Mas a falha em si (*failure*) é a observação de que o software não funciona adequadamente. Existem falhas que são provocadas por defeitos no software, mas outras que são provocadas por dados incorretos ou problemas tecnológicos (como falha de leitura, segurança ou comunicação).

Assim, nem todas as falhas são provocadas por defeitos. A área de teste de software ocupa-se principalmente das falhas provocadas por defeitos, para que os defeitos sejam corrigidos e assim essas falhas nunca mais ocorram.

Já as falhas provocadas por causas externas ao software usualmente são assunto da área de *tolerância a falhas* (Linden, 1976). Os requisitos suplementares de tolerância a falhas de um sistema é que vão estabelecer, por exemplo, como o software se comporta no caso de uma falha de comunicação. Note-se que o requisito vai especificar o comportamento do software frente a uma situação de falha provocada externamente ao software. Assim, mesmo que a falha ocorra, caso o software se comporte de acordo com sua especificação, pode-se dizer que esta falha não é provocada por um defeito no software e que, pelo menos em relação a esta situação, ele *parece estar* livre de defeito.

No parágrafo anterior evitou-se afirmar categoricamente que o software está livre de defeito porque, de fato, tal afirmação dificilmente poderá ser feita para sistemas que não sejam triviais. A maioria dos sistemas de médio e grande porte podem conter defeitos ocultos

porque a quantidade de testes necessária para garantir que estejam livres de defeitos pode ser virtualmente infinita.

Assim, a área de teste de software ocupa-se em definir conjuntos finitos e exequíveis de teste que, mesmo não *garantindo* que o software esteja livre de defeitos, consigam localizar os mais prováveis, permitindo, assim, sua eliminação.

Uma máxima da área de teste de software afirma que o teste não consegue provar que o software está livre de defeitos, ele apenas consegue provar que o software *possui* algum defeito.

13.1.2 Verificação, Validação e Teste

Outra distinção que convém ser feita é entre verificação, validação e teste:

a) *Verificação* consiste em analisar o software para ver se ele está sendo construído de acordo com o que foi especificado.
b) *Validação* consiste em analisar o software construído para ver se ele atende às verdadeiras necessidades dos interessados.
c) *Teste* é uma atividade que permite realizar a verificação e a validação do software.

Assim, a pergunta-chave para a validação é “estamos fazendo a coisa certa?”, enquanto a pergunta chave para a verificação é “estamos fazendo certo a coisa?”. No caso da validação, trata-se de saber se os requisitos incorporados no software refletem efetivamente as necessidades dos interessados (cliente, usuário etc.). No caso da verificação, trata-se de saber se o produto criado atende aos requisitos da forma mais adequada possível, ou seja, que esteja livre de defeitos e possua outras características de qualidade já definidas (Seção 11.1).

Os níveis de teste vão estabelecer diferentes objetivos para as atividades de teste. A maioria dos objetivos de teste é relacionado à verificação. Usualmente apenas o teste de aceitação é efetuado com vistas à validação do software.

13.1.3 Teste e Depuração

Enquanto a atividade de teste consiste em sistematicamente executar o software para encontrar falhas desconhecidas, a *depuração* é a atividade que consiste em buscar a causa da falha, ou seja, o defeito oculto que a está causando.

O fato de se saber que o software não funciona não significa que necessariamente se saiba qual ou quais são as linhas de código que provocam essa falha. A depuração pode ser uma atividade dispendiosa. Por isso, os processos mais modernos recomendam que a integração dos sistemas seja feita de forma incremental, e que sejam integradas partes pequenas de código de cada vez, pois se um sistema funcionava antes da integração e passou a falhar depois da integração, o defeito provavelmente está nos componentes que acabaram de ser integrados e não nos componentes que já funcionavam antes.

13.1.4 *Stubs* e *Drivers*

Frequentemente, partes do software precisam ser testadas isoladamente. Mas essas partes normalmente se comunicam com outras partes.

Quando um componente *A* que vai ser testado chama operações de outro componente *B* que ainda não foi implementado, pode-se criar uma implementação simplificada de *B*, chamada *stub*, que será utilizada no lugar de *B*. Essa implementação simplificada pode ser, por exemplo, uma função que ao invés de realizar um cálculo retorna um valor predeterminado, mas que será útil para testar o componente *A*.

Suponha que *A* é uma classe que precisa usar um gerador de números primos *B*. A implementação completa de *B* capaz de gerar o n-ésimo número primo através da função `primo(n):integer` pode ainda não ter sido implementada. Mas, possivelmente, para efeito de testar *A* não será necessário gerar mais do que alguns poucos números primos, por exemplo, os cinco primeiros. Então *B* poderia ser substituído por uma função *stub* que gera apenas os 5 primeiros números primos, implementada da seguinte forma:

```
Função primo(n):integer
  Caso n
    1: retorna 2;
    2: retorna 3;
    3: retorna 5;
    4: retorna 7;
    5: retorna 11
  Fim
```

Dessa forma, a classe *A* pode ser testada sem que *B* tenha sido efetivamente implementada.

Por outro lado, muitas vezes é o módulo *B* que já está implementado, mas o módulo *A* que chama as funções de *B* ainda não foi implementado. Neste caso, deverá ser implementada uma simulação do módulo *A*, denominada *driver*. Essa simulação deverá chamar as funções do módulo *B* e, de preferência, executar todos os casos de teste necessários para testar *B*, de acordo com a técnica de teste adotada.

A diferença entre o *stub* e o *driver* reside na direção da relação de dependência entre os componentes. Se o componente testado depende (ou seja, chama operações) do componente simulado, então o componente simulado é um *stub*. Inversamente, se o componente simulado é que chama as operações do componente testado, então o componente simulado é um *driver*.

Assim, um *stub* e um *driver* são implementações simplificadas ou simuladas de componentes de sistema, construídas especificamente para atividades de teste. São, normalmente, código do tipo *throw-away*, embora, muito provavelmente, os *drivers* deverão ser guardados como patrimônio de teste para execução de futuros testes de regressão (Seção 13.2.6). Isso porque os *drivers* terão sido construídos para sistematicamente testar as combinações de entradas mais importantes para as classes ou módulos aos quais eles estão associados, conforme será visto adiante neste capítulo.

13.2 Níveis de Teste de Funcionalidade

Os objetivos dos testes podem variar bastante, e vão desde verificar se as funções mais básicas do software estão bem implementadas até validar os requisitos junto ao cliente. A classificação de testes mais empregada considera que existem níveis de testes de funcionalidades (unidade, integração, sistema, ciclo de negócio, aceitação, operação, etc.), que

serão detalhados nesta seção, e testes suplementares (performance, segurança, tolerância a falhas, etc.).

Os testes de funcionalidade têm como objetivo basicamente verificar e validar se as funções implementadas estão corretas nos seus diversos níveis.

13.2.1 Teste de Unidade

Os testes de unidade são os mais básicos e usualmente consistem em verificar se um componente individual do software (unidade) foi implementado corretamente. Esse componente pode ser um método ou procedimento, uma classe completa ou ainda um pacote de funções ou classes de tamanho pequeno a moderado. Usualmente, essa unidade ainda estará isolada do sistema do qual fará parte.

Os testes de unidade são usualmente realizados pelo próprio programador e não pela equipe de teste. A técnica de desenvolvimento orientado a testes (Beck, 2003) recomenda inclusive que antes de o programador desenvolver uma unidade de software ele deve desenvolver o seu *driver* de teste, que, conforme foi visto, é um programa que obtém ou gera um conjunto de dados que serão usados para testar sistematicamente o componente que ainda vai ser desenvolvido.

O teste de unidade pode se valer bem da técnica de teste estrutural (Seção 13.4), que vai garantir pelo menos que todos os comandos e decisões do componente implementado tenham sido exercitados para verificar se têm defeitos. Esta técnica ainda poderá ser complementada pelo teste funcional (Seção 13.5), que vai verificar o componente relativamente à sua especificação.

Um exemplo de teste de unidade consiste em verificar se um método foi corretamente implementado em uma classe. Suponha a classe `Livro` e o método `setIsbn(umIsbn:string)`. A especificação do método estabelece que ele deve trocar o valor do atributo `isbn` do livro para o parâmetro passado, mas, antes disso, deve verificar se o valor do ISBN passado não corresponde a um ISBN já cadastrado (porque uma regra de negócio estabelece que dois livros não podem ter o mesmo ISBN). Assim, independentemente de que se tenha implementado ou não o método `setIsbn`, um caso de teste de unidade poderia ser escrito para ele da seguinte forma:

a) Inserir um ISBN que ainda não consta na base e verificar se o atributo do livro foi adequadamente modificado.
b) Inserir um ISBN que já existe na base e verificar que a exceção foi sinalizada, como esperado.

O *driver* para testar a operação `setIsbn` poderia, então, ser escrito assim:

```
PROGRAMA DriverParaSetIsbn
  (* 1 – entrada correta *)
  REPITA
    isbn := geraIsbnAutomaticamente()
  ATÉ NÃO existeLivronaBaseComIsbn(isbn)
  umLivro := obtemUmLivroDaBase()
  umLivro.setIsbn(isbn)
```



```
SE umLivro.getIsbn() = isbn ENTÃO
  Escreva('teste 1 correto')
SENAO
  Escreva('falha encontrada no teste 1')
FIM SE
(* 2 - entrada incorreta *)
umLivro := obtemUmLivrodaBase()
REPITA
  outroLivro := obtemUmLivroDaBase()
ATÉ umLivro <> outroLivro
TENTE
  umLivro.setIsbn(outroLivro.getIsbn())
  Escreva('falha encontrada no teste 2 - não ocorreu exceção')
FIM TESTE
CAPTURE EXCEÇÃO
  SE EXCEÇÃO 'isbnInvalido' ENTÃO escreva ('teste 2 correto')
FIM CAPTURE
```

Escrever este tipo de código para cada um dos milhares de *set* e *get* de um programa de porte médio pode ser altamente tedioso e consumidor de tempo. Mas é necessário que tal código exista porque os testes de unidade possivelmente terão que ser repetidos inúmeras vezes ao longo do processo de desenvolvimento do software e, possivelmente, muitas outras vezes, durante sua operação, para realizar testes de regressão (seção 13.2.6).

Por outro lado, estes testes de unidade podem ser extremamente reduzidos caso um gerador de código automatizado seja usado para gerar as operações padrão *get* e *set* sobre atributos ou associações. Neste caso, assume-se que este código estará correto, pois o gerador não comete enganos. Assim, apenas outros métodos (métodos delegados) teriam que ser testados na fase de teste de unidade. Felizmente, em sistemas de informação, *set* e *get* acabam sendo a maioria dos métodos (em quantidade, não em esforço) e assim, a economia de tempo que se pode conseguir com geradores automáticos de código, mesmo que implementados apenas para essas funções mais simples, é muito significativa.

Uma ferramenta para automatização de testes de unidade desenvolvida para Java, mas também adaptada para outras linguagens é o *JUnit*[178]. Trata-se de um *framework* gratuitamente distribuído. O *framework* permite inserir comandos específicos de verificação no programa que acusarão os erros caso venham a ser encontrados.

Outra ferramenta é o *OCL Query-Based Debugger* (Hobatr & Malloy, 2001), que é uma ferramenta para depurar programas em C++ usando consultas formuladas em OCL (Object Constraint Language[179]).

Uma lista de *frameworks* de teste para mais de 70 linguagens de programação pode ser encontrada da Wikipédia[180].

[178] junit.wikidot.com/
[179] www.omg.org/spec/OCL/2.0/
[180] en.wikipedia.org/wiki/List_of_unit_testing_*frameworks*

13.2.2 Teste de Integração

Testes de integração são feitos quando unidades, como classes, por exemplo, estão prontas e testadas isoladamente e precisam ser integradas em um *build* para gerar uma nova versão de um sistema.

Dentre as estratégias de integração, usualmente são citadas:

a) *Integração big bang*: consiste em construir as diferentes classes ou componentes separadamente e depois integrar tudo junto no final. É uma técnica não incremental, utilizada no ciclo de vida Cascata com Sub-Projetos. Tem como vantagem o alto grau de paralelismo que se pode obter durante o desenvolvimento e o fato de não precisar de *drivers* e *stubs*, mas como desvantagem tem o fato de não ser incremental (portanto, inadequada para o Processo Unificado e métodos ágeis). Além disso, a integração de muitos componentes ao mesmo tempo pode dificultar bastante a localização dos defeitos, pois estes poderão estar em qualquer um dos componentes.
b) *Integração bottom up*: consiste em integrar inicialmente os módulos de mais baixo nível, ou seja, aqueles que não dependem de nenhum outro, e depois ir integrando os módulos de nível imediatamente mais alto. Assim, um módulo só é integrado quando todos os módulos dos quais ele depende já foram integrados e testados. Dessa forma não é necessário escrever *stubs*, mas em compensação as funcionalidades de mais alto nível do sistema somente serão testadas tarde, quando os módulos de nível superior forem finalmente integrados.
c) *Integração top-down*: consiste em integrar inicialmente os módulos de nível mais alto, deixando os mais básicos para o fim. A vantagem está em verificar inicialmente os comportamentos mais importantes do sistema onde repousam as maiores decisões. Mas como desvantagem está o fato de que muitos *stubs* são necessários, e que o teste, para ser efetivo, precisa de bons *stubs*, caso contrário, ao se integrarem os módulos de nível mais baixo poderão ocorrer problemas inesperados.
d) *Integração sandwich*: consiste em integrar os módulos de nível mais alto da forma *top-down* e os de nível mais baixo da forma *bottom-up*. Esta técnica reduz um pouco os problemas das duas estratégias anteriores, mas seu planejamento é mais complexo.

A principal desvantagem da maioria das técnicas (exceto *big-bang*) é o fato de que as classes ou componentes individuais que vão ser testados necessitam comunicar-se com outros componentes ou classes que ainda não foram testados ou sequer escritos. Neste caso, as interfaces que ainda não existem são supridas pelos *stubs* que são implementações incompletas e simplistas e que se tornam descartáveis depois que o código real for produzido.

O problema com *stubs*, então, é que se perde tempo desenvolvendo software que não vai ser efetivamente entregue e também nem sempre é possível saber se a simulação produzida pelo *stub* será suficientemente adequada para os testes.

No caso do desenvolvimento iterativo, especialmente no caso dos métodos ágeis, nem sempre se sabe em que ordem os componentes serão integrados, porque vários desenvolvedores estarão trabalhando em paralelo em componentes de diferentes níveis da hierarquia de dependência. Assim, a cada integração deverá ser avaliado se o componente já possui no

sistema os componentes dos quais ele depende e os componentes que dependem dele. Na falta destes, usa-se *drivers* ou *stubs*, como nos testes de unidade.

É necessário estar ainda atento às versões dos componentes do sistema, de forma que a integração sempre deixe claro quais versões de quais componentes foram efetivamente testadas juntas (Capítulo 10).

No caso de desenvolvimento utilizando o Processo Unificado e as técnicas de projeto baseadas em MVC (Wazlawick, 2011, p. 98), o teste de integração deverá testar, sempre que houver integração de algum componente (novo ou atualizado), as operações e consultas de sistema (isto é, implementadas na controladora), que utilizam funções deste componente (classe, pacote ou método). Mais adiante a Seção 13.5 vai apresentar técnicas para de testes de operações e consultas de sistema frente às suas especificações funcionais (contratos).

13.2.3 Teste de Sistema

O *teste de sistema* visa verificar se a versão corrente do sistema permite executar processos ou casos de uso completos do ponto de vista do usuário que executa uma série de operações de sistema em uma interface (não necessariamente gráfica) e sendo capaz de obter os resultados esperados.

O teste de sistema pode ser encarado então como o teste de execução dos fluxos de um caso de uso expandido. Se cada uma das operações de sistema (passos do caso de uso) estiverem já testadas e integradas corretamente, então deve-se verificar se o fluxo principal do caso de uso pode ser executado corretamente obtendo os resultados desejados, bem como os fluxos alternativos (Wazlawick, 2011, p. 56).

É possível programar tais testes automaticamente, utilizando um módulo de programa que faz as chamadas diretamente na controladora de sistema e testa, desta forma, todas as condições de sucesso e fracasso dos passos do caso de uso, ou ainda utilizar a interface do sistema para executar tais operações manualmente.

Considere-se, a título de exemplo, um teste de sistema a ser conduzido para avaliar o seguinte caso de uso:

Caso de Uso: Comprar livros

1. [IN] O comprador informa sua identificação.
2. [OUT] O sistema informa os livros disponíveis para venda (título, capa e preço) e o conteúdo atual do carrinho de compras (título, capa, preço e quantidade).
3. [IN] O comprador seleciona os livros que deseja comprar.
4. O comprador decide se finaliza a compra ou se guarda o carrinho:
    - 4.1 Variante: Finalizar a compra.
    - 4.2 Variante: Guardar carrinho.

Variante 4.1: Finalizar a compra

4.1.1. [OUT] O sistema informa o valor total dos livros e apresenta as opções de endereço cadastradas.
4.1.2. [IN] O comprador seleciona um endereço para entrega.
4.1.3. [OUT] O sistema informa o valor do frete e total geral, bem como a lista de cartões de crédito já cadastrados para pagamento.
4.1.4. [IN] O comprador seleciona um cartão de crédito.

4.1.5. [OUT] O sistema envia os dados do cartão e valor da venda para a operadora.
4.1.6. [IN] A operadora informa o código de autorização.
4.1.7. [OUT] O sistema informa o prazo de entrega.

Variante 4.2: Guardar carrinho

4.2.1 [OUT] O sistema informa o prazo (dias) em que o carrinho será mantido.

Exceção 1a: Comprador não cadastrado

1a.1 [IN] O comprador informa seu CPF, nome, endereço e telefone.
Retorna ao passo 1.

Exceção 4.1.2a: Endereço consta como inválido.

4.1.2a.1 [IN] O comprador atualiza o endereço.
Avança para o passo 4.1.2.

Exceção 4.1.6a: A operadora não autoriza a venda.

4.1.6a.1 [OUT] O sistema apresenta outras opções de cartão ao comprador.
4.1.6a.2 [IN] O comprador seleciona outro cartão.
Retorna ao passo 4.1.5.

Este caso de uso apresenta então duas situações onde o processo termina sem percalços que são indicadas pelas duas variantes: *guardar o carrinho de compras* e *finalizar a compra*. O processo completo que vai do passo 1 até o final de cada uma destas variantes (passos 4.1.7 e 4.2.1) deve ser, portanto, executado como caso de teste de sucesso.

Além das duas situações normais, os fluxos de exceção devem ser testados. A exceção 1*a* pode ocorrer tanto com a variante 4.1 quanto com a variante 4.2, mas normalmente não precisaria ser testada com ambas as variantes. O testador poderia escolher uma delas para testar o processo completo com a exceção 1*a*. No exemplo acima, poder-se-ia escolher a variante 4.2, por ser mais curta.

Já as outras exceções só ocorrem dentro da variante 4.1 e, portanto, serão testadas juntamente com ela.

A técnica de elaboração dos casos de teste para teste de sistema de um caso de uso consiste então em selecionar todos os caminhos possíveis e passar pelo menos uma vez por cada um dos fluxos alternativos (variantes e exceções). Essa técnica é bastante semelhante à técnica de caminhos independentes do teste estrutural (Seção 13.4), mas é mais simples de aplicar porque a estrutura lógica de um caso de uso normalmente é bem mais simples do que a de um programa.

O plano de teste de sistema para este caso de uso poderia, então, ser definido como na Figura 13-1

Objetivo	Caminho	Como testar	Resultados
Fluxo principal com variante 4.1	1, 2, 3, 4, 4.1, 4.1.1, 4.1.2, 4.1.3, 4.1.4, 4.1.5, 4.1.6, 4.1.7	Um cliente cadastrado informa livros válidos, indica um endereço e cartão válidos e a operadora (possivelmente um *stub*) autoriza a compra.	Compra efetuada.
Fluxo principal com variante 4.2	1, 2, 3, 4, 4.2, 4.2.1	Um cliente cadastrado informa livros válidos, e guarda o carrinho.	Carrinho guardado.
Exceção 1a	1, 1a,1a.1, 1, 2, 3, 4, 4.2, 4.2.1	Um cliente não cadastrado informa livros válidos e guarda o carrinho.	Cliente é cadastrado e o carrinho guardado.

Exceção 4.1.2a	1, 2, 3, 4, 4.1, 4.1.1, 4.1.2, 4.1.2a, 4.1.2a.1, 4.1.2, 4.1.3, 4.1.4, 4.1.5, 4.1.6, 4.1.7	Um cliente cadastrado informa livros válidos, indica um endereço inválido, e depois um endereço válido; indica um cartão válido e a operadora (possivelmente um *stub*) autoriza a compra.	O endereço inválido é atualizado e a compra efetuada.
Exceção 4.1.6a	1, 2, 3, 4, 4.1, 4.1.1, 4.1.2, 4.1.3, 4.1.4, 4.1.5, 4.1.6, 4.1.6a, 4.1.6a.1, 4.1.6a.2, 4.1.5, 4.1.6, 4.1.7	Um cliente cadastrado informa livros válidos, indica um endereço e cartão válidos e a operadora (possivelmente um *stub*) não autoriza a compra. O cliente informa outro cartão válido e a operadora autoriza a compra.	Compra efetuada.

Figura 13-1: Exemplo de plano de teste de sistema para um caso de uso.

O teste de sistema, normalmente, é realizado com o uso da técnica funcional, ou seja, não se examina a estrutura interna do código, mas apenas a forma como o sistema se comporta em relação a sua especificação.

Considera-se que o teste de sistema somente é executado em uma versão (*build*) do sistema onde todas as unidades já foram testadas e os componentes recém integrados também já foram testados. Caso muitos erros sejam encontrados durante o teste de sistema, o usual é abortar o processo e refazer, ou mesmo replanejar, os testes de unidade e integração, para que uma versão suficientemente estável do sistema seja produzida e possa assim ser testada.

13.2.4 Teste de Aceitação

O *teste de aceitação* é usualmente realizado pelo usuário ou cliente, usando a interface final do sistema. Ele pode ser planejado e executado exatamente como o teste de sistema, mas a diferença é que é realizado pelo usuário final ou cliente e não pela equipe de desenvolvimento.

Em outras palavras, enquanto o teste de sistema faz a *verificação* do sistema, o teste de aceitação faz a *validação* do sistema. O teste de aceitação tem como objetivo principal, então, a validação do software frente aos requisitos, e não a verificação de defeitos. Ao final do teste de aceitação o cliente poderá aprovar a versão do sistema testada ou solicitar modificações.

O teste de aceitação ainda pode ter mais duas variantes assim conhecidas:

a) *Teste alfa*: teste efetuado pelo cliente ou seu representante de forma livre, sem o planejamento e formalidade do teste de sistema. O usuário vai livremente utilizar o sistema e suas funções e, por isso, este teste também é chamado de teste de aceitação informal, em oposição ao *teste de aceitação formal* que deveria seguir o mesmo planejamento utilizado pelo teste de sistema (Figura 13-1).
b) *Teste beta*: este teste é ainda menos controlado pela equipe de desenvolvimento. No teste beta, versões operacionais do software são disponibilizadas para vários usuários que, sem acompanhamento direto nem controle por parte da empresa desenvolvedora, vão explorar o sistema e suas funcionalidades. Normalmente versões beta de sistemas expiram após um período pré-determinado, quando então os usuários são convidados a fazer uma avaliação do sistema.

Usualmente sistemas feitos sob medida (*tailored*) deveriam ser testados pelo teste de aceitação formal. Já sistemas de prateleira (*off-the-shelf*) são normalmente testados por testes alfa e beta.

13.2.5 Teste de Ciclo de Negócio

O *teste de ciclo de negócio* é uma abordagem possível tanto no teste de sistema quanto no teste de aceitação formal, e consiste em testar uma sequência casos de uso que corresponde a um possível ciclo de negócio da empresa.

Assim, ao invés de testar os casos de uso isoladamente, o analista ou cliente vai testá-los no contexto de um ciclo de negócio. Por exemplo, pode-se testar a sequência de compra de um item do fornecedor, seu registro de chegada em estoque, sua venda, entrega e respectivo pagamento. São vários casos de uso relacionados no tempo, pois representam o ciclo de vida em um item em relação à empresa.

Os testes de ciclo de negócio podem ser planejados a partir de especificações de modelo de negócio (se existirem). Essas especificações usualmente estarão feitas em diagramas de atividade (Wazlawick, 2011, pp. 9-19) ou *BPMN* (*Business Process Modeling Notation*)[181].

13.2.6 Teste de Regressão

O teste de regressão é executado sempre que um sistema em operação sofre manutenção. O problema é que a correção de um defeito no software, ou modificação de alguma de suas funções pode ter gerado novos defeitos. Neste caso, devem ser executados novamente todos os testes de unidade das unidades alteradas, bem como os testes de integração e sistema sobre as partes afetadas.

No caso de manutenção adaptativa (Seção 14.2.2) pode ainda ser necessário executar testes de aceitação, com participação do cliente. Mas no caso da manutenção corretiva (Seção 14.2.1) tais testes não são necessários visto que a funcionalidade do software não muda.

O teste de regressão tem esse nome porque se ao se aplicarem testes a uma nova versão na qual versões anteriores passaram, e essa nova versão não passar, então se considera que o sistema *regrediu*.

Como é um teste muito trabalhoso e aplicado muitas vezes ao longo do tempo de vida do software, o ideal é que o teste de regressão seja sempre automatizado (Dustin, 2002).

Algumas ferramentas que podem ser usadas para testes de regressão são o *Rational Funcional Tester*[182], o *Mercury Quick Test Professional* [183] e o *JUnit*[184].

13.3 Testes Suplementares

Ao contrário dos testes de funcionalidade, que verificam e validam as funções do sistema, ou seja, os processos que podem ser efetivamente realizados, os testes suplementares verificam características normalmente associadas aos requisitos suplementares, como performance, interface, tolerância a falhas etc.

---

[181] www.bpmn.org/

[182] www-01.ibm.com/software/awdtools/tester/functional/

[183] https://h10078.www1.hp.com/cda/hpms/di*SPL*ay/main/hpms_content.jsp?zn=bto&cp=1-11-15-24^1322_4000_100__

[184] junit.wikidot.com/

13.3.1 Teste de Interface com Usuário

O teste de interface com usuário tem como objetivo verificar se a interface permite realizar as atividades previstas nos casos de uso de forma eficaz e eficiente. Mesmo que as funções estejam corretamente implementadas, o que já teria sido visto no teste de sistema, isso não quer dizer que a interface o esteja. Então, em geral, é necessário testar a interface de forma objetiva e específica.

O teste de interface com usuário pode ainda verificar a conformidade das interfaces com normas ergonômicas que se apliquem[185].

No Processo Unificado, a base para a realização de testes de interface serão também os casos de uso, de forma análoga ao que acontece no teste de sistema. Porém, agora, ao invés de simplesmente verificar se o sistema permite a execução das funcionalidades e obtenção dos resultados corretos, o teste deverá verificar se a interface está organizada da melhor forma possível para atender o usuário com eficiência.

Outra vantagem do uso de casos de uso como base para o teste de interface com usuário reside no fato de que o caso de uso apresenta a estrutura de fluxo principal de uma interação do usuário com o sistema. Assim, a interface projetada deverá procurar otimizar a sequência de ações do usuário neste fluxo principal. Ao mesmo tempo as ações necessárias para realizar os fluxos alternativos deverão estar disponíveis no momento em que forem necessárias, ou seja, quando o usuário estiver executando os passos do caso de uso que poderiam provocar esta ou aquela exceção.

Em outras palavras, a sequência de operações do caso de uso deve aparecer claramente na interface em uma ordem lógica, evitando-se, por exemplo, que o usuário precise focar sua atenção em diferentes áreas da interface para seguir uma simples sequência lógica. Algumas interfaces exigem que o usuário faça muitos movimentos de braço para levar o mouse para cima e para baixo, para a direita e para a esquerda, enquanto executa uma atividade sequencial. Esse tipo de coisa deveria ser evitado.

Da mesma forma, ao ocorrerem exceções que precisam ser tratadas pelo usuário, a mensagem deve aparecer em local visível e próximo do foco de atenção do usuário (são comuns sistemas que colocam mensagens de erro que não são vistas pelo usuário) e os controles de interface necessários para iniciar as ações corretivas também devem estar próximos à mensagem e ao foco de atenção do usuário sempre que possível (e não três ou quatro janelas adiante).

13.3.2 Teste de Performance (Carga, *Stress* e Resistência)

Tenha o sistema requisitos de desempenho ou não, o teste de performance pode ser importante, especialmente nas operações que serão realizadas com muita frequência ou de forma iterativa. O teste consiste em executar a operação e mensurar seu tempo, avaliando se está dentro dos padrões definidos.

O teste de performance também pode ser usado para avaliar a confiabilidade e a estabilidade de um sistema. Existem assim, diversos tipos de teste de performance:

[185] Por exemplo, a NBR ISO 9241-11:2011: www.abntcatalogo.com.br/norma.aspx?ID=86090.

a) *Teste de carga*. É a forma mais simples de teste de performance. Normalmente o teste de carga é feito para uma determinada quantidade de dados ou transações, que se espera sejam típicos para um sistema, e avalia o comportamento do sistema em termos de tempo para estes dados ou transações. Desta forma, pode-se verificar se o sistema atende aos requisitos de performance estabelecidos e também pode-se verificar se existem gargalos de performance para serem tratados.
b) *Teste de stress*. É um caso extremo de teste de carga. Procura-se levar o sistema ao seu limite máximo esperado de funcionamento para verificar como se comporta. Este tipo de teste é feito para verificar se o sistema é suficientemente robusto frente a situações anormais de carga de trabalho. O teste também ajuda a verificar quais seriam os problemas encontrados caso a carga do sistema ficasse acima do limite máximo estabelecido.
c) *Teste de resistência*. É feito para verificar se o sistema consegue manter suas características de performance durante um longo período de tempo com uma carga nominal de trabalho. Os testes de resistência devem verificar basicamente o uso da memória ao longo do tempo para garantir que não existam perdas acumulativas de memória em função de lixo não recolhido, e também deverão verificar se não existe degradação de performance após um substancial período de tempo em que o sistema opera com carga nominal ou acima desta.

Pode-se verificar também, no caso de sistemas concorrentes, o que acontece quando um número de usuários próximo ou acima do máximo gerenciável tenta utilizar o sistema. Algumas vezes, efeitos colaterais indesejados podem surgir nessas situações, pelo que tais testes são considerados altamente desejáveis.

13.3.3 Teste de Segurança

Os testes de segurança usualmente são considerados parte da área de *segurança computacional*.

Basicamente os seis tipos de segurança devem ser testados quando os requisitos de sistema assim o exigirem:

a) *Integridade*: é uma forma de garantir ao receptor que a informação que ele recebeu é correta e completa.
b) *Autenticação*: é a garantia de que um usuário realmente é quem ele diz ser e que os documentos, programas e sites realmente sejam aqueles que se espera que sejam.
c) *Autorização*: é o processo de verificar se alguma pessoa ou sistema pode ou não acessar determinada informação ou sistema.
d) *Confidencialidade*: segurança de que quem não tem direito à informação não possa obtê-la.
e) *Disponibilidade*: é a segurança de quem tem direito à informação consiga obtê-la quando necessário.
f) *Não repúdio*: é uma forma de garantir que o emissor e receptor de uma mensagem não possam posteriormente alegar não ter enviado ou recebido a mensagem.

13.3.4 Teste de Recuperação de Falha

Quando um sistema tem requisitos suplementares referentes a tolerância ou recuperação de falhas, estes devem ser testados separadamente. Basicamente, busca-se verificar se o sistema de fato atende aos requisitos especificados relacionados a esta questão.

Normalmente trata-se de situações referentes a:

a) Queda de energia no cliente ou no servidor.
b) Discos corrompidos.
c) Problemas de comunicação.
d) Quaisquer outras condições que possam potencialmente provocar a terminação anormal do programa ou a interrupção temporária de seu funcionamento.

13.3.5 Teste de Instalação

Basicamente busca-se no teste de instalação verificar se o software não entra em conflito com outros sistemas eventualmente instalados em uma máquina, bem como se todas as informações e produtos para instalação estão disponíveis para os usuários instaladores.

O teste de instalação também é associado com o teste de compatibilidade, onde se busca verificar se o sistema é compatível com diferentes sistemas operacionais, fabricantes de máquinas, *browsers* etc.

13.4 Teste Estrutural

Em relação às técnicas de teste, existem duas grandes famílias:

a) *Testes estruturais ou caixa-branca*: são testes que são executados com conhecimento do código implementado, ou seja, eles testam a estrutura do programa em si.
b) *Testes funcionais* ou *caixa-preta* (Seção 13.5): são testes executados sobre as entradas e saídas do programa sem que se tenha necessariamente conhecimento do seu código fonte.

Estas duas famílias de teste incluem ainda várias técnicas, algumas das quais serão definidas a seguir. Procurou-se aqui propositalmente não apresentar um tratamento muito formal e extenso a tais técnicas, embora tal tratamento exista (Delamaro, Maldonado, & Jino, 2007). Pode-se a partir deste capítulo compreender o espírito das técnicas, sua aplicabilidade e limitações, ou seja, no nível que um engenheiro de software deve conhecer tais técnicas.

Os primeiros testes aos quais um sistema é normalmente submetido são os testes de unidade, que tem como objetivo verificar se as funções mais simples do sistema estão corretamente implementadas. Usualmente esse tipo de teste pode ser conduzido como um teste do tipo caixa-branca, ou teste estrutural, pois o que se deseja é analisar exaustivamente a estrutura interna do código implementado.

Um dos problemas com testes de programas é que é impossível definir um procedimento algorítmico que certifique que um programa qualquer está livre de defeitos. Este problema é computacionalmente *indecidível* (Lucchesi, Simon, Simon, Simon, & Kowaltowski, 1979). Então os testes precisam ser feitos, de certa forma, por amostragem.

Mas não se trata de amostragem aleatória. As técnicas de teste indicam como os programas devem ser testados para que se possa com um número razoável de tentativas avaliar se um programa contém erros.

Assim, o teste estrutural não vai testar *todas* as possibilidades de funcionamento de um programa, mas aquelas mais representativas. Em especial, o que se busca com o teste estrutural é exercitar todos os comandos do programa pelo menos uma vez, e todas as condições pelo menos uma vez, para cada um de seus valores possíveis (verdadeiro ou falso).

Uma primeira técnica de teste estrutural pode ser definida assim: cada estrutura de seleção ("*if*" ou "*case*") ou repetição ("*while*" ou "*repeat*") deve ser testada em pelo menos duas situações: quando a condição é verdadeira e quando a condição é falsa. No caso do "*for*" deve-se testar os casos limites, isto é, quando a variável que limita o número de repetições assume um valor mínimo e quando ela assume um valor máximo. Se esse valor máximo for indefinido, então se pode testar com um número arbitrariamente grande.

Assim, o teste estrutural é capaz de detectar uma quantidade substancial de possíveis erros, pela garantia de ter executado pelo menos uma vez todos os comandos e condições do programa.

13.4.1 Complexidade Ciclomática

Uma medida de complexidade de programa bastante utilizada é a *complexidade ciclomática* (McCabe, 1972), a qual pode ser definida, de forma simplificada, assim: se *n* é o número de estruturas de seleção e repetição no programa, então a complexidade ciclomática do programa é *n*+1. Como estruturas conta-se:

a) IF-THEN: 1 ponto.
b) IF-THEN-ELSE: 1 ponto.
c) CASE: 1 ponto para cada opção, exceto OTHERWISE.
d) FOR: 1 ponto.
e) REPEAT: 1 ponto.
f) OR ou AND na condição de qualquer das estruturas acima: acrescenta-se 1 ponto para cada OR ou AND (ou qualquer outro operador lógico binário, se a linguagem suportar, como XOR ou IMPLIES).
g) NOT: não conta.
h) Chamada de sub-rotina (inclusive recursiva): não conta.
i) Estruturas de seleção e repetição em sub-rotinas ou programas chamados: não conta.

Assume-se, usualmente, que programas com complexidade ciclomática menor ou igual a 10 são simples e fáceis de ser testados, com complexidade ciclomática de 11 a 20 são de médio risco em relação ao teste, de 21 a 50 são de alto risco e que programas com complexidade acima de 50 não são testáveis[186]. Porém, deve-se tomar cuidado aqui com a interpretação do termo "programa". Um programa, neste sentido é um bloco de código único. Se ele fizer chamadas a outros blocos de código, as estruturas destes outros blocos não são contabilizadas na complexidade ciclomática do programa em questão.

[186] Fonte: SEI – Software Engineering Institute. www.sei.cmu.edu

A Figura 13-2 apresenta um programa Pascal com duas estruturas de seleção e uma estrutura de repetição, portanto com complexidade ciclomática 4 (o "else" não é contado como uma nova estrutura, pois faz parte do "if").

```
01.  if num = 0 then
02.    fib := 0
03.  else
04.    if num = 1 then
05.      fib := 1
06.    else
07.      begin
08.        ultimoFib := 1;
09.        cont := 1;
10.        repeat
11.          penultimoFib := ultimoFib;
12.          ultimoFib := fib;
13.          fib := penultimoFib + ultimoFib;
14.          cont := cont + 1;
15.        until cont = num;
16.      end
17.    ;
18.  ;
```

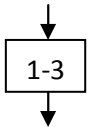
Figura 13-2: Exemplo de programa com complexidade ciclomática 4.

Na sessão seguinte é definido o grafo de fluxo de um programa, a partir do qual pode-se também calcular a complexidade ciclomática. Em caso de dúvida na contagem de estruturas de seleção e repetição, pode-se calcular essa complexidade contando-se o número de regiões do grafo de fluxo ou ainda contando o número de nós e arestas e calculando: 2 + nós - arestas. Nos três casos, o valor deve ser o mesmo.

13.4.2 Grafo de Fluxo

O *grafo de fluxo* de um programa é obtido colocando-se todos os comandos em nós e os fluxos de controle em arestas. Comandos em sequência podem ser colocados em um único nó, e estruturas de seleção e repetição devem ser representadas através de nós distintos com arestas que indicam a decisão e a repetição, quando for o caso.

As regras da Figura 13-3 podem ser seguidas para a criação do grafo de fluxo.

Regra	Código	Grafo
Comandos em sequência (dois ou mais)	01. <…>; 02. <…>; 03. <…>;	

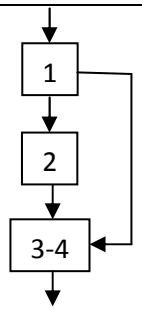
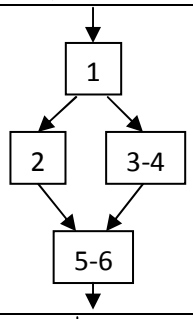
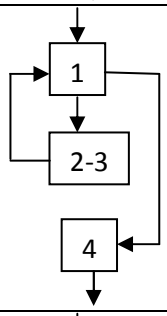
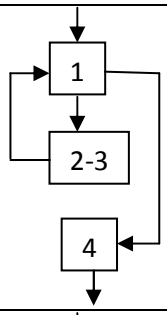
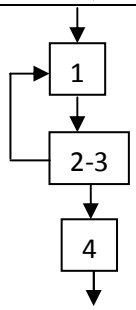
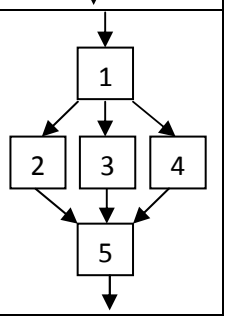
If-Then	01. IF condição THEN 02. <…>; 03. ENDIF; 04. <…>;	
If-Then-Else	01. IF condição THEN 02. <…>; 03. ELSE 04. <…>; 05. ENDIF; 06. <…>;	
For	01. FOR condição DO 02. <…>; 03. ENDFOR; 04. <…>;	
While	01. WHILE condição DO 02. <…>; 03. ENDWHILE; 04. <…>;	
Repeat	01. REPEAT 02. <…>; 03. UNTIL condição; 04. <…>;	
Case (para qualquer quantidade de casos)	01. CASE X OF 02. a: <…>; 03. b: <…>; 04. OTHERWISE <…>; 05. <…>;	

Figura 13-3: Regras para criação de grafos de fluxo.

Expressões booleanas compostas com OR podem ser tratadas como instruções à parte, como na Figura 13-4.

```
01. IF cond1 OR cond2 THEN
02.  <…>;
03. ELSE
04.   <…>;
05. ENDIF;
06. <…>;
```

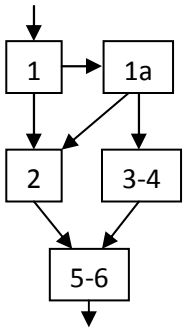


Figura 13-4: Regra para criação de grafo de fluxo para estruturas com condição OR.

No passo 1, se `cond1` for falsa então o fluxo vai para 1a, onde `cond2` será testada. Se no passo 1 `cond1` for verdadeira então o fluxo vai diretamente para o passo 2. Já no nodo 1a, se `cond2` for verdadeira, o fluxo vai para o passo 2, senão vai para o passo 3-4.

Simetricamente, uma condição com AND deve ser tratada como na Figura 13-5.

```
01. IF cond1 AND cond2 THEN
02.   c1;
03. ELSE
04.   c2;
05. ENDIF;
06. c3;
```

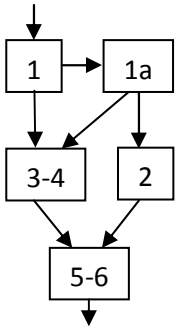


Figura 13-5: Regra para criação de grafo de fluxo para estruturas com condição AND.

No passo 1, se `cond1` for verdadeira então o fluxo vai para 1a, onde `cond2` será testada. Se no passo 1 `cond1` for falsa então o fluxo vai diretamente para o passo 3-4. Já no nodo 1a, se `cond2` for verdadeira, o fluxo vai para o passo 2, senão vai para o passo 3-4.

Note-se que, no caso de estruturas de seleção ou repetição com OR ou AND, a complexidade ciclomática resultante é maior do que em estruturas que não tenham esses operadores binários. Assim, cada OR ou AND deve ser efetivamente contabilizado na contagem de predicados para cálculo da complexidade ciclomática.

Assim, aplicando as regras da Figura 13-3 ao programa da Figura 13-2, será obtido um grafo de fluxo semelhante ao da Figura 13-6.

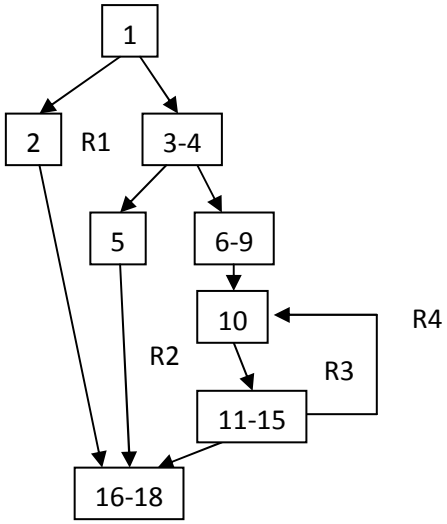


Figura 13-6: Grafo de fluxo do programa da Figura 13-2.

A figura também mostra as quatro regiões do grafo de fluxo: *R*1, *R*2, *R*3 e *R*4, cuja quantidade corresponde ao valor da complexidade ciclomática.

Outra forma ainda de calcular essa complexidade, caso se tenha dúvida com as duas anteriores, é calcular o número de arestas do grafo, subtrair o número de nodos e somar 2. No caso da Figura 13-6, o grafo possui 10 arestas e 8 nodos. Portanto, sua complexidade ciclomática é 10-8+2 = 4.

13.4.3 Caminhos Independentes

O valor da complexidade ciclomática indica número *máximo* de execuções *necessárias* para exercitar todos os comandos do programa. No caso do exemplo da seção anterior seriam 4. Porém, a execução de todos os comandos (nós) não vai necessariamente testar os valores verdadeiro e falso de todas as condições possíveis. Por exemplo, é possível passar por todos os nós sem nunca passar pela aresta que vai dos nós 11-15 ao nó 10. Isso acontecerá se em todas as vezes que o REPEAT da linha 10 for executado ele sair para a linha 16 logo após a primeira repetição.

Assim, uma abordagem mais adequada para o teste seria exercitar não apenas todos os nodos, mas todas as *arestas* do grafo de fluxo. Isso é feito pela determinação dos *caminhos independentes* do grafo, que são possíveis navegações do início ao fim do grafo (do nodo 1 ao nodo 16-18).

O conjunto dos caminhos independentes pode ser definido da seguinte forma:

a) Inicialize o conjunto dos caminhos independentes com um caminho qualquer do início ao fim do grafo (usualmente pode ser o caminho mais curto).
b) Enquanto for possível adicione ao conjunto dos caminhos independentes outros caminhos que passem por pelo menos uma aresta na qual nenhum dos caminhos anteriores ainda passou.

Dessa forma, o conjunto dos caminhos independentes do grafo da Figura 13-6, pode ser definido como:

a) O caminho c_1 = <1, 2, 16-18>.
b) O caminho c_2 = <1, 3-4, 5, 16-18>.
c) O caminho c_3 = <1, 3-4, 6-9, 10, 11-15, 16-18>.
d) O caminho c_4 = <1, 3-4, 6-9, 10, 11-15, 10, 11-15, 16-18>.

A complexidade ciclomática define o número *máximo* de testes necessários para passar por todas as arestas. Com 4 testes, como mostrado acima, é possível exercitar todos os comandos e todas as possíveis condições das estruturas de seleção e repetição. Apesar disso, nada impede que mais testes sejam feitos se assim for desejado.

Além disso, esse número define a quantidade *máxima* de testes necessários. Isso significa que possivelmente um conjunto menor de testes poderia passar por todas as arestas. Por exemplo, no caso acima, o caminho c_4 exercita os mesmos comandos e condições do caminho c_3, pois ao final da repetição a condição do "until" será verdadeira (e isso é testado no caminho c_3). Mas convém manter o caminho c_3 visto que ele trata uma condição limite da estrutura de repetição, quando ela é executada apenas uma única vez.

13.4.4 Casos de Teste

Resta ainda definir os *casos de teste* (Myers, Sandler, Badgett, & Thomas, 2004), ou seja, quais dados de entrada levam o programa a executar cada um dos caminhos independentes e qual a saída esperada do programa para cada um destes casos.

Considerando novamente o programa da Figura 13-2, a única entrada deste programa é a variável `num`, da qual se deseja obter o número de Fibonacci correspondente. Pode haver mais de uma possibilidade, mas de forma geral deve acontecer o seguinte:

a) Para exercitar o caminho c_1: `num`=0. Dará verdadeiro na condição da linha 1.
b) Para exercitar o caminho c_2: `num`=1. Dará falso na linha 1 e verdadeiro na linha 4.
c) Para exercitar o caminho c_3: `num`=2. Dará falso nas linhas 1 e 4 e verdadeiro na primeira vez que passar pela linha 15.
d) Para exercitar o caminho c_4: `num`>2. Dará falso nas linhas 1, 4 e, pelo menos, na primeira vez que passar pela linha 15. Depois em algum momento dará verdadeiro na linha 15.

Um caso de teste, então poderia ser definido assim como na Tabela 13-1.

Tabela 13-1: Casos de teste.

Caminho	Entrada	Saída esperada
c_1	`num=0`	`fib=0`
c_2	`num=1`	`fib=1`
c_3	`num=2`	`fib=1`
c_4	`num=3`	`fib=2`

Poderiam ser definidos outros testes para valores de "`num`" superiores a 3, mas estariam exercitando os mesmos nodos e arestas do caminho c_4, ou seja, a mesma lógica.

13.4.5 Múltiplas Condições

O critério de cobertura de todas as arestas, conforme visto anteriormente, precisa cobrir também os desvios provocados pelo uso de operadores lógicos binários em estruturas de seleção e repetição, como por exemplo, o programa da Figura 13-7.

```
01.  var anos : array [1..6] of integer;
02.      coluna : integer;
03.  begin
04.    coluna := 1;
05.    while (anos[coluna] <> 1996) and (coluna < 6) do
06.      coluna := coluna + 1;
07.  end
```

Figura 13-7: Um programa com uma decisão baseada em múltiplas condições.

O programa em questão deve verificar se o valor 1996 pertence a um vetor de 6 posições. Se pertencer, o programa deve retornar a coluna onde o valor ocorre, caso contrário, deve retornar o valor 7. Este programa, propositalmente possui um erro de lógica muito comum embutido.

O grafo de fluxo do programa é mostrado na Figura 13-8.

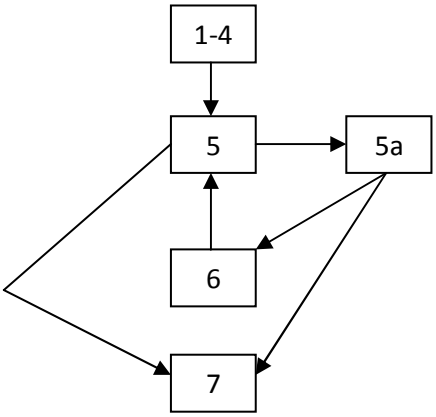


Figura 13-8: Grafo de fluxo do programa da Figura 13-7.

Neste caso, então, o grafo de fluxo têm três regiões, e, a princípio, três testes será o máximo necessário para verificar todas as condições e comandos.

O conjunto de caminhos independentes deste grafo poderia ser definido assim:

a) c_1: 1-4, 5, 7.
b) c_2: 1-4, 5, 5a, 7.
c) c_3: 1-4, 5, 5a, 6, 5, 7.

Para executar o primeiro caminho independente é necessário que a condição `(anos[coluna] <> 1996)` seja falsa na primeira vez. Como coluna será necessariamente 1 neste ponto, então será necessário ter `anos[1]=1996`. Assim o `while` não será executado nenhuma vez e o primeiro caminho independente estará concluído.

Para executar o segundo caminho independente é necessário que a condição `(anos[coluna] <> 1996)` seja verdadeira e a condição `(coluna < 6)` falsa na primeira vez que o `while` for executado. Esse caminho é impossível pela lógica do programa, pois na primeira vez que o `while` for executado, `coluna` terá o valor 1 e a segunda condição não poderá ser falsa. Porém, a escolha do caminho independente foi feita de forma arbitrária, isto é, sem considerar as restrições lógicas do programa. Em um caso como este, o testador deverá tentar divisar uma maneira de executar o caminho mesmo que outras arestas tenham que ser adicionadas. No caso em tela, o objetivo é testar o `while` quando a primeira condição for verdadeira e a segunda condição falsa. Isso pode ser obtido, considerando a lógica do programa fazendo-se com que o *loop* seja executado até que `coluna` seja igual a 6. Para que a primeira condição seja verdadeira até este ponto, é necessário que todos os valores do vetor `anos` sejam diferentes de 1996. Assim, uma entrada que poderia exercitar este caminho seria `anos = <0,0,0,0,0,0>`. O caminho, por sua vez, vai incluir uma sequência de repetições do padrão 5, 5a, 6. Assim c_2 será 1-4, 5, 5a, 6, 5, 5a, 6, 5, 5a, 6, ..., 5, 5a, 7.

Embora o segundo caminho independente faça várias repetições do *loop*, em nenhuma delas ele repete a sequência 5, 7 esperada pelo terceiro caminho independente. Assim, este terceiro caminho precisa efetivamente ser executado de forma independente. Para exercitar esse caminho é necessário que as duas condições sejam verdadeiras na primeira passada pelo `while` e a primeira condição seja falsa na segunda passada pelo `while`. Para obter este efeito é necessário um vetor que tenha um valor diferente de 1996 na primeira posição e igual a 1996 na segunda posição.

Assim, o conjunto de casos de teste para este programa poderia ser definido conforme a Tabela 13-2.

Tabela 13-2: Casos de teste para o programa da Figura 13-7.

Caminho independente	Entrada	Saída esperada
c_1	`anos=<1996,0,0,0,0,0>`	`coluna=1`
c_2	`anos=<0,0,0,0,0,0>`	`coluna=7`
c_3	`anos=<0,1996,0,0,0,0>`	`coluna=2`

Pode-se observar que a execução do programa com as entradas definidas na tabela acima vai exercitar todas as combinações possíveis dos valores verdade das condições pelo menos uma vez.

A execução dos testes com os valores de entrada definidos vai mostrar que o programa apresenta um defeito, porque o teste com o caminho c_2 vai dar como resultado `coluna=6` e não `coluna=7`, como esperado.

Observa-se então, claramente, que o teste apenas aponta que o programa contém um defeito. Mas o teste não diz onde está o defeito, nem como consertá-lo. Se o programa parasse com um erro de divisão por zero, por exemplo, até seria mais fácil saber onde está o defeito, mas o programa executa até o final. Apenas o resultado é que não é o esperado.

Porém, uma pista é dada pelo teste estrutural, porque entre vários caminhos independentes possíveis, apenas o caminho c_2 não produziu o resultado esperado. Então o processo de depuração deverá analisar os comandos que compõe esse caminho, bem como as condições com os respectivos valores verdade testados.

13.4.6 Caminhos Impossíveis

Uma possível correção do programa da Figura 13-7 pode levar a outra situação à qual o testador deve estar atento. Algumas vezes, certos caminhos do grafo de fluxo simplesmente são impossíveis de testar porque a lógica do programa torna impossível passar por eles.

```
01.  var anos : array [1..6] of integer;
02.      coluna : integer;
03.  begin
04.    coluna := 1;
05.    while (anos[coluna] <> 1996) and (coluna < 6) do
06.      coluna := coluna + 1
07.    ;
08.    if (coluna = 6) and (anos[coluna]<>1996) then
09.      coluna := 7;
10.  end
```

Figura 13-9: Um programa com caminhos impossíveis de testar.

Esse programa tem duas estruturas de controle em sequência: o WHILE e o IF. Mas o IF está estruturado de forma a depender do WHILE, ou seja, o IF só poderá ser verdadeiro se o WHILE terminar em uma determinada condição. Assim, o comando da linha 09 só poderá ser executado quando o WHILE sair na sexta repetição sem encontrar o valor 1996. O comando 09 é, portanto, incompatível com os caminhos c_1 e c_3. Assim, embora o grafo de fluxo deste programa seja como mostrado na Figura 13-9, nem todos os caminhos são possíveis.

Um caminho como, por exemplo, 1-4, 5, 7-8, 9, 10, embora permitido pelo grafo, seria impossível de executar devido à lógica do programa.

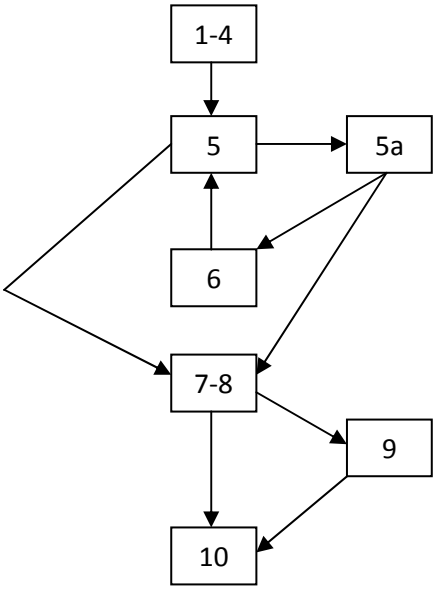


Figura 13-10: Grafo de fluxo do programa da Figura 13-9.

Embora este grafo tenha complexidade ciclomática 4, apenas três caminhos independentes serão necessários para testá-lo (lembrando que a complexidade ciclomática é um limite máximo):

a) c_1: 1-4, 5, 7-8, 10.
b) c_2: 1-4, 5, 5a, 6, 5, 5a, 6, ..., 5, 5a, 7-8, 9, 10.
c) c_3: 1-4, 5, 5a, 6, 5, 7-8, 10.

Este tipo de problema deixa claro porque programas com complexidade ciclomática acima de 10 começam a apresentar maior risco em relação ao teste. É cada vez mais difícil construir bons casos de teste.

13.4.7 Limitações

O teste estrutural usualmente é aplicado para verificar defeitos no código. Mas, este tipo de teste não é capaz de identificar, por exemplo, se o programa foi bem especificado. Ou seja, o teste vai verificar se o programa se comporta de acordo com a especificação dada, mas não necessariamente se ele se comporta de acordo com a especificação *esperada*. Então, essa técnica não pode ser usada para validar sistemas, apenas verificar defeitos.

Outra limitação deste tipo de teste é que ele não necessariamente cobre potenciais problemas com estruturas de dados como *arrays* e listas. Além disso, ele não necessariamente trata situações típicas de programas orientados a objetos, especialmente quando se usa herança e polimorfismo. Para essas situações existem técnicas de teste específicas (Delamaro, Maldonado, & Jino, 2007).

Outras limitações conhecidas do teste estrutural são o fato de que funcionalidades ausentes não são testadas, pois a técnica preconiza apenas o teste daquilo que existe no programa. Ou seja, se algum comando que *deveria* ter sido incluído no código, mas não o foi, o teste estrutural não será capaz de identificar isso. Por exemplo, o programa de cálculo de número de Fibonacci da Figura 13-2 não testa entradas formadas por números negativos. Uma entrada negativa neste programa fará com que ele entre em *loop* infinito. Mas isso não é percebido no teste porque apenas os caminhos efetivamente definidos no programa são testados, e não os caminhos ausentes.

Além disso, algumas vezes o programa pode estar produzindo o resultado correto para uma entrada por mera coincidência, não significando que esteja correto.

Uma última limitação também relaciona-se ao fato de que o teste estrutural só pode ser produzido *depois* que o código está escrito, o que não habilita seu uso com a técnica de desenvolvimento dirigido pelo teste, preconizada pelos métodos ágeis, que sugere que os casos de teste sejam definidos antes de se escrever o código.

Apesar dessas limitações a técnica é relevante e seu uso é importante para a detecção de vários tipos de defeitos no software, especialmente nas suas unidades mais básicas, pois ele é usado para garantir que todos os comandos e condições lógicas sejam executados pelo menos uma vez.

13.5 Teste Funcional

O teste estrutural é adequado quando se pretende verificar a estrutura de um programa. Mas em várias situações a necessidade consiste em verificar a funcionalidade de um programa independentemente de sua estrutura interna. Um programa pode ter uma especificação, ou comportamento esperado, usualmente elaborado em um contrato de operação de sistema e o que se deseja saber é se ele efetivamente cumpre este contrato ou especificação. O teste estrutural é adequado para uma verificação em primeira mão de unidades (métodos, procedimentos ou algoritmos), ou seja, os elementos mais básicos do software.

Em orientação a objetos, em especial, o teste estrutural pode ser aplicado às operações básicas das classes que criam e destroem instâncias, adicionam ou removem associações ou alteram o valor de atributos. Tais testes também são adequados para as consultas básicas que retornam o valor de atributos normais ou derivados, bem como de associações normais ou derivadas (Wazlawick, 2011, p. 108).

Porém, em um nível mais alto de operações, tal verificação estrutural pode ser difícil de realizar, pois métodos podem chamar métodos de outras classes, delegando responsabilidades de um objeto a outro. Assim, no nível de integração de funções mais básicas do software será mais adequado realizar *testes funcionais*, que avaliam o comportamento de uma operação mais abrangente do que as operações elementares, porque essa técnica pode avaliar comportamentos que estão distribuídos em várias classes (coisa que a técnica estrutural não faz diretamente).

O padrão MVC (*Model View Control*), bastante empregado em sistemas de informação, sugere que um sistema deva ser dividido em pelo menos três camadas, sendo que a superior (*View*) corresponde à interface com usuário, e a intermediária (*Control*) é a que efetivamente se responsabiliza pela execução de processamento lógico sobre a informação. Com o uso dessa técnica, todas as transformações da informação vão ocorrer encapsuladas pela controladora (uma classe com a função específica de encapsular estes comportamentos), e serão acessíveis pela interface através de chamadas a métodos dessa classe. Quando os métodos fazem alteração de dados, são chamados de *operações de sistema* e quando fazem consulta a dados são chamados de *consultas de sistema*.

Técnicas de análise de sistemas (Wazlawick, 2011, pp. 153-192) (Larman, 2001), recomendam que essas operações e consultas sejam especificadas por contratos bem definidos com parâmetros tipados, pré-condições, pós-condições e exceções. O teste funcional então consistirá em verificar se em situação de normalidade (pré-condições atendidas) as pós-condições desejadas são realmente obtidas, e se em situações de anormalidade as exceções são efetivamente levantadas.

13.5.1 Particionamento de Equivalência

Um dos princípios do teste funcional é a identificação de situações equivalentes. Por exemplo, se um programa aceita um conjunto de dados (normalidade) e rejeita outro conjunto (exceção) então se pode dizer que existem duas *classes de equivalência* para os dados de entrada do programa, os dados aceitos e os dados rejeitados. Pode ser impossível testar todos

os elementos de cada conjunto[187], até porque esses conjuntos podem ser infinitos. Então, o particionamento de equivalência vai determinar que pelo menos um elemento de cada conjunto seja testado.

Classicamente, a técnica de particionamento de equivalência considera a divisão das entradas possíveis da seguinte forma (Myers, Sandler, Badgett, & Thomas, 2004):

a) Se as entradas válidas são especificadas como um *intervalo de valores* (por exemplo, de 10 a 20), então é definido um conjunto válido (10 a 20) e dois inválidos (menor do que 10 e maior do que 20).
b) Se as entradas válidas são especificadas como uma *quantidade de valores* (por exemplo, uma lista com cinco elementos), então é definido um conjunto válido (lista com 5 elementos) e dois inválidos (lista com menos de 5 elementos e lista com mais de 5 elementos).
c) Se as entradas válidas são especificadas como um *conjunto de valores aceitáveis* que podem ser tratados de forma diferente (por exemplo, os *strings* "masculino" e "feminino"), então é definido um conjunto válido para cada uma das formas de tratamento e um conjunto inválido para outros valores quaisquer.
d) Se as entradas válidas são especificadas como uma condição do tipo "deve ser de tal forma" (por exemplo, uma restrição sobre os dados de entrada como "a data final deve ser posterior a data inicial"), então deve ser definido um conjunto válido (quando a condição é verdadeira) e um inválido (quando a condição é falsa).

Os conjuntos de valores válidos devem ser definidos não só em termos de restrições sobre as entradas, mas também em função dos resultados a serem produzidos. Se a operação a ser testada puder ter dois comportamentos possíveis em função do valor de um dos parâmetros, então vão existir dois conjuntos distintos de valores válidos para aquele parâmetro, um para cada comportamento possível.

Por exemplo, considere um contrato de operação de sistema "`adicionaLivro(idLivro, idCompra, quant)`" que toma da interface três valores: um identificador de livro (`idLivro`), um identificador de uma compra (`idCompra`) e uma quantidade (`quant`). O objetivo da operação consiste em adicionar na compra indicada um item que associe o livro indicado e uma quantidade. A Figura 13-11 apresenta o modelo conceitual de referência para este exemplo. Apenas os atributos relevantes para o exemplo foram usados, embora as classes originais pudessem ter vários outros.

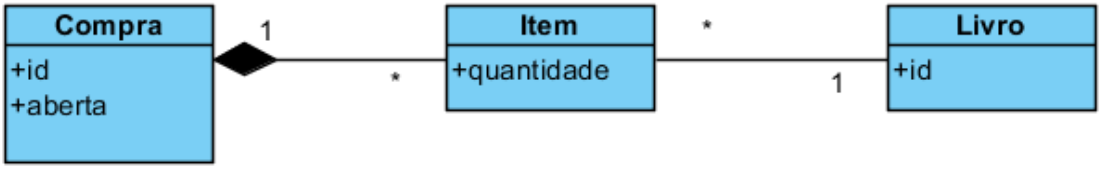


Figura 13-11: Modelo conceitual de referência para exemplo.

[187] A literatura usualmente fala em *classes* de elementos aceitos e rejeitados, que é um conceito matemático. Aqui será usada a palavra *conjunto* para que não haja confusão com o conceito de *classe* de orientação a objetos.

Em função das entradas, os seguintes conjuntos de valores válidos e inválidos podem ser definidos:

a) Para `idLivro`:
    a. Válido: se existe um livro com este id.
    b. Inválido: se tal livro não existe.
b) Para `idCompra`:
    a. Válido: se existe uma compra com este id e ela está aberta
    b. Inválido: caso a compra não exista.
    c. Inválido: se a compra já está fechada.
c) Para `quantidade`:
    a. Válido: se for um valor maior ou igual a 1.
    b. Inválido: para zero.

Em relação aos resultados da operação, pode-se ainda considerar que, se o livro já consta na compra, então ao invés de criar um novo item, deve-se somar a quantidade solicitada com a quantidade que já consta na compra. Ou seja, existem dois comportamentos possíveis para um parâmetro de entrada, dependendo do valor deste e do estado interno dos objetos. Neste caso, pode-se subdividir o conjunto de valores válidos relacionados com `idLivro` em dois conjuntos válidos: um quando o `idLivro` passado não consta na compra e outro quando o `idLivro` passado já consta na compra. Isso resulta na seguinte subdivisão:

a) Para `idLivro`:
    a. Válido: se existe um livro com este id *e ele não consta na compra*.
    b. Válido: se existe um livro com este id *e ele já consta na compra*.
    c. Inválido: se tal livro não existe.

Assim, os casos de teste poderiam ser definidos em termos desses conjuntos válidos e inválidos conforme a Tabela 13-3.

Tabela 13-3: Casos de teste funcional para uma operação de sistema.

Tipo	`idLivro`	`idCompra`	`quant`	Resultado esperado
Sucesso	um id já cadastrado que não consta na compra	um id já cadastrado de compra aberta	qualquer valor acima de 0	um item é criado e associado à compra e ao livro, com atributo quantidade = quant
Sucesso	um id já cadastrado que consta na compra	um id já cadastrado de compra aberta	qualquer valor acima de 0	o item do livro que já consta na compra tem seu valor incrementado com quant
Exceção	um id não cadastrado	um id já cadastrado de compra aberta	qualquer valor acima de 0	exceção: livro não cadastrado
Exceção	um id já cadastrado	um id não cadastrado	qualquer valor acima de 0	exceção: compra não cadastrada
Exceção	um id já cadastrado	um id já cadastrado de compra fechada	qualquer valor acima de 0	exceção: compra fechada
Exceção	um id já cadastrado	um id já cadastrado de compra aberta	0	exceção: quantidade não pode ser zero

Usualmente, nem todas as combinações de conjuntos válidos e inválidos para todos os parâmetros são testadas. Normalmente, testa-se todas as combinações de conjuntos válidos, como nas duas primeiras linhas da Tabela 13-3, e acrescenta-se uma possível classe inválida de cada vez, correspondendo às demais linhas da tabela.

Assim, usualmente, não se testam condições de erro que ocorrem combinadas, ou seja, duas ou mais condições de erro de cada vez. Isso ocorre porque a combinação de todos os conjuntos válidos e inválidos cresce exponencialmente em relação à quantidade deles. Ou seja, cada novo conjunto válido ou inválido identificado tecnicamente multiplica o número de testes necessários. No exemplo acima, a condição (a) revisada tem 3 conjuntos, a condição (b) tem também 3 conjuntos e a condição (c) dois conjuntos, resultando portanto em 3x3x2=18 combinações possíveis, caso todas fossem ser testadas. Se houvesse apenas mais um conjunto na condição (c), haveria 3x3x3=27 combinações possíveis.

Então, embora não se testem todas as combinações, é necessário testar pelo menos:

a) Todas as combinações de conjuntos válidos.
b) Todas as combinações de conjuntos válidos com cada um dos conjuntos inválidos.

Assim, o testador é obrigado a incluir cada conjunto inválido pelo menos em um teste. Mas, se por algum motivo, quiser combinar conjuntos inválidos também, isso não é proibido, apenas poderá ser mais trabalhoso.

13.5.2 Análise de Valor Limite

Um dos ditados em teste de software diz que os *bugs* (*insetos*, em inglês) costumam se esconder nas *frestas*. Em função dessa realidade, a técnica de partição de equivalência normalmente é usada em conjunto com o critério de *análise de valor limite*.

A *análise de valor limite* consiste em considerar não apenas um valor qualquer para teste dentro de uma classe de equivalência, mas um ou mais valores fronteiriços com outras classes de equivalência quando isso puder ser determinado.

Em domínios ordenados (números inteiros, por exemplo), esse critério pode ser aplicado. Por exemplo, se um programa exige uma entrada que para ser válida deve estar no intervalo [*n*..*m*], então existem três classes de equivalência:

a) Inválida para qualquer $x < n$.
b) Válida para qualquer $x >= n$ e $x <= m$.
c) Inválida para qualquer $x > m$.

Então a análise de valor limite sugere que possíveis erros de lógica do programa não vão ocorrer em pontos arbitrários dentro desses intervalos, mas nos pontos onde um intervalo se encontra com outro. Então:

a) Para a primeira classe inválida, deve-se testar para o valor *n*-1.
b) Para a classe válida, deve-se testar os valores *n* e *m*.
c) Para a segunda classe inválida, deve-se testar para o valor *m*+1.

Assim, se existir um erro no programa para alguma dessas classes é muito mais provável que ele seja capturado dessa forma do que se fosse selecionado um valor qualquer dentro de cada um destes três intervalos.

13.6 TDD – Desenvolvimento Orientado a Testes

TDD (*Test Driven Development*) ou *Desenvolvimento Orientado a Testes* (Beck, 2003) é uma técnica ou filosofia de programação que incorpora o teste no processo de produção de código da seguinte forma:

a) Primeiramente o desenvolvedor, que recebe a especificação de uma nova funcionalidade a ser implementada, deve programar um conjunto de testes automatizados para testar o código que ainda não existe.
b) Esse conjunto de testes deve ser executado e falhar. Isso é feito para mostrar, que os testes efetivamente tem algum poder de teste e que não terão sucesso a não ser que o código específico seja desenvolvido. Se o código passar em algum destes testes é possível que, ou o teste não seja efetivo, ou a característica a ser implementada já existe no sistema.
c) Em seguida, o código deve ser desenvolvido da forma mais minimalista possível, isto é, ser desenvolvido apenas para passar nos testes.
d) Depois que o código passar em todos os testes ele deve ser refatorado para atender padrões de qualidade interna e testado novamente até que passe em todos os testes novamente.

A Figura 13-12 apresenta o diagrama de atividades para este tipo de processo.

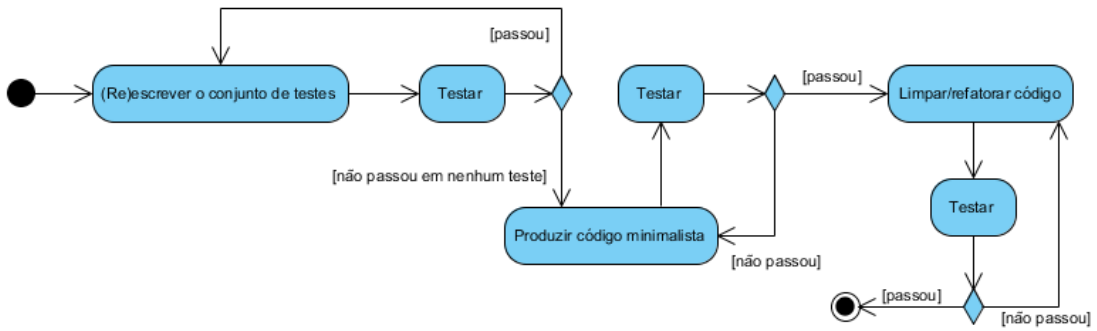


Figura 13-12: Processo de desenvolvimento orientado a testes.

Um dos pontos chave neste processo é que, uma vez que os testes foram estabelecidos, o programador não deve implementar nenhuma funcionalidade além daquelas para as quais o teste foi criado. Isso evita que os programadores percam tempo criando estruturas que efetivamente não eram necessárias desde o início. Isso satisfaz o princípio *KISS* (*Keep it Simple, Stupid*!)[188], que indica que seria estupidez fazer algo mais do que o mais simples necessário em um projeto, e também o princípio *YAGNI* (*You Ain't Gonna Need It*)[189]

Para escrever um conjunto de testes efetivo, é necessário que o programador primeiramente compreenda perfeitamente os requisitos e a especificação do componente que ele vai desenvolver. Neste sentido, os contratos de operação de sistema e o correspondente teste funcional podem ajudar muito.

[188] "Mantenha isso simples, tolo!" en.wikipedia.org/wiki/KISS_principle
[189] "Você não vai precisar disso" en.wikipedia.org/wiki/You_ain%27t_gonna_need_it

A técnica é bastante exigente em relação ao processo a ser seguido. Um programador, por exemplo, que descubra a necessidade de adicionar um ELSE a um IF em um código já aprovado nos testes, deverá, antes de escrever código, escrever um teste para este ELSE, certificar-se que o teste falha para o programa atual, e somente depois adicionar o ELSE no código. Pular etapas não é encorajado pela técnica

Um desenvolvimento extra de TDD pode ser atingido quando se consegue automatizar os testes de aceitação do usuário. Essa abordagem, conhecida com *ATDD* (*Acceptance Test-Driven Development*) (Koskela, 2007) disponibiliza aos desenvolvedores a garantia de que os requisitos estão sendo atendidos, mesmo que o usuário não esteja presente todo o tempo.

13.7 Medição em Teste

Uma informação importante a ser relatada por equipes de teste é o *status* da atividade de teste (Crowther, 2012), o que deve ser feito de forma precisa e compreensiva. Então, neste caso aplica-se também o conceito de métrica para avaliação objetiva de *status*. Pode-se falar em dois grandes conjuntos de métricas de teste:

a) *Métricas do processo de teste*. As medidas relacionadas reportam a quantidade de testes requeridos, planejados, executados, etc., mas não o estado do produto em si.
b) *Métricas de teste do produto*. As medidas relacionadas reportam o estado do produto em relação à atividade de teste, como por exemplo, quantos defeitos foram encontrados, quantos estão em revisão, quantos foram resolvidos, etc.

As seguintes medições relacionadas às métricas de teste poderiam ser apresentadas durante a preparação e execução dos testes:

a) Número de testes previstos (sua necessidade já foi identificada).
b) Número de testes planejados (casos de teste definidos).
c) Número de testes executados, incluindo (a) testes que falharam e (b) testes que foram aprovados.

Estas métricas podem ser comparadas ou relativizadas com outras métricas usualmente tomadas sobre o processo, como, por exemplo, o tempo investido nas atividades, o número de pontos de função, etc.

Exemplos de métricas de teste relativas ao produto seriam:

a) Número de defeitos descobertos.
b) Número de defeitos corrigidos.
c) Distribuição dos defeitos por grau de severidade.
d) Distribuição dos defeitos por módulo.

As medições relacionadas a essas métricas podem ser apresentados em gráficos, que permitem visualizar o desempenho das equipes de desenvolvimento e testes, onde se espera que o número de defeitos descobertos seja relativamente constante no tempo, e que as atividades de correção atinjam seus objetivos também de forma constante, até que o produto esteja suficientemente livre de defeitos para ser implantado (Figura 13-13).

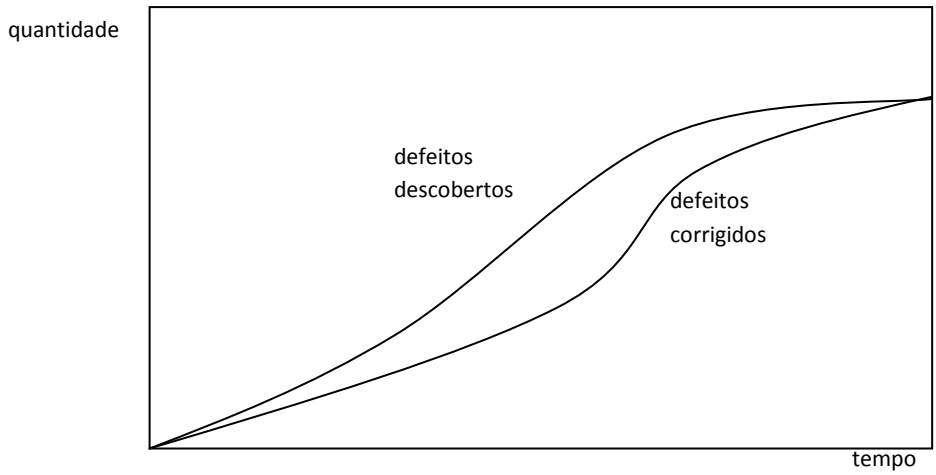


Figura 13-13: Evolução esperada das medidas de defeitos de produto ao longo de um projeto.

Outras medidas relacionadas a teste são o tempo gasto com as atividades de teste (isto é, o custo dos testes) e a cobertura dos testes, isto é, quantas funcionalidades do sistema tiveram seus testes efetivamente planejados, executados e aprovados.

O site *Software Testing Stuff*[190] apresenta várias métricas compostas, como, por exemplo:

a) *Custo para encontrar um defeito* = esforço total com atividades de teste / número de defeitos encontrados.
b) *Adequação dos casos de teste* = número de casos de teste real / número de casos de teste estimado.
c) *Efetividade dos casos de teste* = número de defeitos encontrados através de testes / número total de defeitos encontrados.

13.8 Depuração

Depuração é o processo de remover defeitos de um programa. Ela normalmente inicia com atividades de teste, que identificam o problema, ou a partir de inspeções de código, ou ainda, a partir de relatos de usuários. No último caso, a equipe deve, no início do processo de depuração, tentar reproduzir o defeito relatado pelo usuário, o que nem sempre é possível ou fácil.

A depuração é uma atividade ainda artesanal, onde a quantidade e variedade de casos dificulta a elaboração de um processo padrão. Porém, a adoção de boas técnicas de engenharia de software, como, integrações frequentes, controle de versões, desenvolvimento orientado a testes, etc., podem facilitar bastante o processo de descoberta e correção de defeitos.

A tarefa de depuração pode ser tão simples quanto corrigir uma *string* com erro ortográfico ou tão complexa a ponto de exigir um trabalho de pesquisa, coleta de dados e formulação de hipóteses digno dos melhores detetives.

De forma geral, ferramentas, conhecidas como *debuggers* ajudam no trabalho de depuração, pois permitem executar programas passo a passo, observando o valor das variáveis, parar,

[190] www.softwaretestingstuff.com/2007/10/software-testing-metrics.html

retornar, etc. A Wikipédia[191] apresenta uma extensa lista de tais ferramentas para várias linguagens.

13.9 Prova de Correção de Programas

O teste de software, como foi visto neste capítulo ajuda a encontrar defeitos no software, mas não *garante* que o software esteja livre de defeitos. Uma abordagem que forneça este tipo de garantia, em relação a esta especificação, é possível, mas tem alto custo. Por isso, usualmente, só é aplicada em sistema e componentes onde o erro é absolutamente não aceitável.

Uma das técnicas correntemente usadas para prova de correção de programas é o *Calculo de Hoare* (Hoare, 1969)[192], que consiste em uma técnica de definição de semântica axiomática para programas.

A idéia principal do cálculo é colocar precondições antes e poscondições após um comando ou fragmento de programa, da seguinte forma:

{*P*} *C* {*Q*}

Onde *P* são as precondições, *Q* são as poscondições e *C* é o comando a ser testado.

Neste caso, afirma-se que o comando *C* está correto quando os valores de entrada passam por *P*, resultando em verdadeiro, e depois de *C* ser executado, os valores de saída passam por *Q*, também resultando em verdadeiro.

O cálculo de Hoare utiliza um conjunto de regras baseadas em técnicas de análise de algoritmos:

a) *Comando Skip* (`;`). Um fragmento de programa que não executa nada tem pré e poscondições idênticas. Portanto {*P*} `;` {*P*} é um axioma do cálculo.>>>
b) *Comando de Atribuição.* Se uma variável *x* recebe uma atribuição com uma expressão *f*, então qualquer propriedade que fosse verdadeira para *f* antes da atribuição será verdadeira para *x* após a atribuição (mas não mais necessariamente verdadeira para *f*, já que a expressão *f* pode conter ocorrências de *x*). Assim, {*P*(*f*)} `x:=f` {*P*(*x*)} será um esquema de axioma, que define um conjunto infinito de axiomas do cálculo para todas as variáveis e expressões possíveis.
c) *Comando if-then-else.* No caso de comandos if-then-else, há dois caminhos e a regra de inferência indica que cada um dos caminhos deve manter as pós-condições esperadas. Ou seja, considerando a expressão {*P*} `if C then X else Y endif` {*Q*}, essa sequência de precondição, comando e poscondição somente será verdadeira se antes for verdadeiro {*P* AND *C*} `X` {*Q*} e se também for verdadeiro {*Q* AND NOT *C*} `Y` {*Q*}.
d) *Comando if-then.* A regra de if-then é idêntica à de if-then-else, apenas que se substitui o comando que estaria no else por um comando *skip*. Então {*P*} `if C then X endif` {*Q*} será verdadeiro se antes {*P* AND *C*} `X` {*Q*} for verdadeiro e {*Q* AND NOT *C*} `;` {*Q*} for verdadeiro.

[191] en.wikipedia.org/wiki/Comparison_of_debuggers
[192] sunnyday.mit.edu/16.355/Hoare-CACM-69.pdf

e) *Comando while.* Um comando como `WHILE C DO X ENDWHILE` será executado até que a condição *C* seja falsa. Então, se {*P* AND *C*} `X` {*P*} for verdadeira, {*P*} `WHILE C DO X ENDWHILE` {*P* AND NOT *C*} também deverá ser verdadeiro, onde *P* é uma invariante do *loop* (Meyer, 1997), ou seja, uma expressão que deve ser verdadeira ao longo de toda a execução do laço.
f) *Enfraquecimento da poscondição.* Se {*P*} `X` {*Q*} for verdadeiro e *Q* implica *R*, então {*Q*} `X` {*R*} também é verdadeiro.
g) *Enfraquecimento da precondição*. Se *P* implica *Q* e {*Q*} *X* {*R*} for verdadeiro, então {*P*} *X* {*R*} também será.

Imperial (2003)[193] apresenta exemplos de aplicação destas regras para a prova formal de propriedades de programas. Leite (2000) apresenta um exemplo de aplicação da regra do comando *while*. Deseja-se provar que se *x* for inicialmente maior ou igual a zero, o programa abaixo fará com que ele seja igual a zero ao final:

```
while x>0 do
      x := x-1
end while
```

Neste caso, a precondição *P* é {*x*>=0}, a condição *C* é {*x*>0}, e assim, {*P* AND *C*} fica: {*x*>=0 AND *x*>0}. O comando *X* é `x := x-1`, e a pós condição {*P* AND NOT *C*} será {*x*>=0 AND NOT *x*>0}, ou seja, {*x*=0}. Aplicando-se a regra do while, tem-se que:

{*x*>=0 AND *x*>0} `x:=x-1` {*x*>=0} implica {*x*>=0} `while x>0 do x:=x-1 end while` {*x*>=0 AND NOT *x*>0}

Assim, como o lado esquerdo da implicação é verdadeiro, pois se x é maior que zero, subtrair 1 de x fará com que continue sendo maior que zero ou igual a zero, a seguinte conclusão pode ser obtida:

{*x*>=0} `while x>0 do x:=x-1 end while` {*x*=0}

Dessa forma, comprova-se, pela aplicação do axioma do *while* que o programa satisfaz sua especificação.

Como no caso do método *cleanroom* (Seção 11.4.3), existe certa dificuldade para se encontrar desenvolvedores capacitados a aplicar especificação formal a software e o alto custo da realização destas provas faz com que acabem sendo utilizadas mais fortemente apenas em aspectos de segurança crítica de sistemas.

[193] www2.dbd.puc-rio.br/pergamum/tesesabertas/0124811_03_cap_03.pdf

