CAPÍTULO

3

Teste de Software

3.1 Considerações Iniciais

Neste capítulo, os conceitos relacionados ao teste de software são apresentados. São considerados fundamentos, técnicas, critérios e formas de avaliação de abordagens de teste de software. Este capítulo fornece embasamento teórico necessário para os Capítulos 4, 5, 6 e para a proposta de trabalho apresentada no Capítulo 7.

Este capítulo está organizado da seguinte forma. Na Seção 3.2, os principais conceitos sobre teste de software são apresentados. Na Seção 3.3 são apresentados os principais critérios das técnicas funcional, estrutural e baseada em defeitos. Na Seção 3.4, mecanismos para a geração automática de casos de teste são apresentados. Na Seção 3.5 são apresentadas formas de avaliação considerando estudos teóricos e experimentais sobre teste de software.

3.2 Fundamentos do Teste de Software

O processo de desenvolvimento de software envolve uma série de atividades nas quais, apesar das técnicas, métodos e ferramentas empregados, erros no produto ainda podem ocorrer. Atividades agregadas sob o nome de Garantia de Qualidade de Software têm sido introduzidas ao longo de todo o processo de desenvolvimento. Entre elas estão as atividades de Verificação e Validação (V&V), com o objetivo de minimizar a ocorrência de erros e riscos associados. Atividades de verificação permitem ao desenvolvedor identificar se o desenvolvimento de software está sendo feito da maneira correta. As atividades de validação possibilitam avaliar a adequação do software

produzido em relação aos requisitos do cliente. A atividade de teste é um processo em que um programa é executado com a intenção de encontrar defeitos. Teste é um elemento crítico de garantia de qualidade de software e representa a revisão final da especificação, projeto e código (Myers et al., 2004; Pressman, 2005). A atividade de teste é uma das mais utilizadas, constituindo-se em um dos elementos para fornecer evidências da confiabilidade do software em complemento a outras atividades, como por exemplo, o uso de revisões e de técnicas formais e rigorosas de especificação e de verificação (Maldonado, 1991).

No contexto de teste de software, o padrão IEEE 610.12 (IEEE, 1990) diferencia os seguintes termos: defeito (*fault*) – passo, processo ou definição de dados incorreta (instrução ou comando incorreto); engano (*mistake*) – ação humana que produz um resultado incorreto (uma ação incorreta tomada pelo programador); erro (*error*) – a diferença entre o valor obtido e o valor esperado, ou seja, qualquer estado intermediário incorreto ou resultado inesperado na execução; e falha (*failure*) – produção de uma saída incorreta em relação à especificação. O padrão também define o conceito de caso de teste como um conjunto de entradas de teste, condições de execução e resultados esperados desenvolvido para um objetivo particular como exercitar um determinado caminho no programa ou verificar um determinado requisito. Uma série de casos de teste é denominada conjunto de teste.

A atividade de teste de software pode ser dividida em quatro etapas (Maldonado, 1991):

Planejamento de Testes: nessa etapa, é criado um plano de teste que é uma parte essencial para o gerenciamento do processo de teste. Segundo Myers et al. (2004), um plano de teste deve possuir: objetivos; critérios de finalização que julgam quando uma fase dos testes foi completada; cronogramas; distribuição de responsabilidades; padrões para casos de teste; identificações das ferramentas; tempo de uso de computadores para realizar os testes; configuração de hardware; estratégia de integração; rastreamento do progresso; procedimentos para depuração (localizar e corrigir defeitos); e teste de regressão (reexecução de algum subconjunto dos testes que já foram conduzidos, para garantir que mudanças não tenham inseridos novos defeitos (Pressman, 2005)).

Projeto dos Casos de Teste: nessa etapa, são utilizadas as técnicas que são apresentadas mais adiante. Como o teste não garante inexistência de defeitos, é muito importante a criação de casos de teste efetivos, ou seja, capazes ou com alta probabilidade de revelar defeitos. Myers et al. (2004) recomendam que casos de teste sejam desenvolvidos usando a técnica de teste funcional e complementados com a técnica de teste estrutural.

Execução do Teste: nessa etapa, são aplicados os procedimentos especificados no plano de teste.

Coleta e avaliação dos resultados dos testes: nessa etapa, os resultados dos testes realizados são registrados, organizados e apresentados na forma de relatórios. Os resultados obtidos são comparados com os resultados esperados. Os resultados esperados são obtidos por meio de

um oráculo, que é o testador ou outro mecanismo, que decide se os valores de saída são corretos.

Os testes são aplicados em três fases: teste de unidade, teste de integração e teste do sistema.

Teste de unidade: concentra-se na menor unidade do projeto de software implementada no código-fonte. Essa unidade pode ser chamada de componente ou módulo do software, que pode ser uma função no paradigma procedimental. No paradigma da Orientação a Objetos (OO), a unidade pode ser um método (Vincenzi, 2004) ou uma classe (Binder, 1999). O teste de unidade limita-se a lógica interna e estruturas de dados dentro dos limites da unidade.

Teste de integração: acontece em paralelo com a fase de integração do software, conduzindo testes para revelar defeitos associados às interfaces. Segundo Sommerville (2003), as estratégias de teste *top-down* e *bottom-up* refletem abordagens diferentes de integração do sistema. Na integração *top-down*, os componentes de alto nível de um sistema são integrados e testados antes que seus projetos e sua implementação tenham sido completados. Na integração *bottom-up*, os componentes de nível inferior são integrados e testados antes que os componentes de nível superior tenham sido desenvolvidos. Como o teste de integração envolve a realização de testes em partes do software que ainda estão incompletas, *drivers* e *stubs* precisam ser utilizados (Ammann e Offutt, 2008). O *driver* é um módulo que emula a chamada para a unidade testada e o *stub* é um módulo que simula um comportamento de uma unidade chamada.

Teste de sistema: visa a exercitar o sistema como um todo por meio de vários tipos de teste como teste de recuperação, segurança, estresse e desempenho (Pressman, 2005). O teste de recuperação força a falha do sistema de várias formas, verificando se a recuperação é executada corretamente. O teste de segurança verifica os mecanismos de proteção do sistema, na qual o testador tenta invadir o sistema. O teste de estresse verifica o comportamento do sistema com uma alta demanda de recursos em quantidade, frequência ou volume. O teste de desempenho é projetado para testar o desempenho do software em tempo de execução.

3.3 Técnicas de Teste

Durante a atividade de teste, é necessário saber se o teste efetuado é realmente de boa qualidade. Um critério de teste define quais propriedades ou requisitos que precisam ser testados para avaliar a qualidade dos testes (Zhu et al., 1997). Dados um programa P que está sendo testado, um conjunto de teste T (que contém um subconjunto das entradas de P) e um critério de teste C, diz-se que o conjunto de casos de teste T é C-adequado para o teste de P se T satisfazer os requisitos de teste estabelecidos pelo critério C.

Com base na fonte de informação utilizada para derivar os casos de teste, os critérios podem ser classificados a partir das técnicas: funcional, estrutural e baseada em defeitos. Na técnica funcional

(ou caixa-preta), os critérios e requisitos de teste são constituídos a partir da especificação do software; na técnica estrutural (ou caixa-branca), os casos de teste são derivados essencialmente a partir das características internas da implementação em teste; e, na técnica baseada em defeitos, os critérios e os requisitos são oriundos do conhecimento sobre defeitos típicos no processo de desenvolvimento (Maldonado, 1991).

3.3.1 Técnica de Teste Funcional

O teste funcional, também conhecido com teste caixa-preta, é uma técnica em que o testador considera o programa como uma caixa-preta. O testador não possui conhecimento sobre o comportamento interno e estrutura do programa (Myers et al., 2004). Para a geração dos casos de teste são consideradas apenas as funcionalidades do software, baseando-se na especificação. A seguir, são apresentados os critérios: particionamento em classes de equivalência, análise de valor limite, grafo causa-efeito, método de partição-categoria e teste funcional sistemático.

Particionamento em Classes de Equivalência: divide o domínio de entrada do programa em um número finito de classes de equivalência, por meio das quais são derivados os casos de teste. A divisão é feita em classes de equivalência válidas e inválidas, que respectivamente, agrupam entradas válidas e inválidas. Casos de teste são selecionados para cobrir o máximo de classes válidas. Por outro lado, é selecionado um único caso de teste para cobrir cada classe inválida (Myers et al., 2004). A justificativa para isso é que testar um valor representativo de cada classe é equivalente a testar qualquer outro valor dentro dessa mesma classe.

Análise de Valor Limite: é geralmente utilizado em conjunto com o critério particionamento em classes de equivalência, dando maior ênfase aos limites associados às condições de entrada (Mathur, 2008). Os casos de teste são selecionados nas fronteiras das classes, pois nesses pontos acontece o maior número de defeitos (Pressman, 2005). O mesmo é feito para o domínio de saída, que é particionado em classes e são necessários casos de teste que produzam resultados nos limites dessas classes (Myers et al., 2004).

Grafo Causa-Efeito: explora uma fraqueza dos critérios anteriores que não exploram combinações das condições de entrada (Myers et al., 2004). O critério grafo causa-efeito estabelece requisitos de teste baseado nas possíveis condições de entrada. Primeiramente, são levantadas as possíveis condições de entrada (causas) e as possíveis ações (efeitos) do programa. Em seguida, é construído um grafo relacionando as causas e efeitos levantados. Esse grafo é convertido em uma tabela de decisão por meio da qual são derivados os casos de teste.

Método de Partição-Categoria: instrui a criação de casos de teste funcionais pela decomposição de requisitos funcionais em especificações de teste para as principais funções do software (Ostrand e Balcer, 1988). Essas especificações de teste consistem de categorias correspondentes às entradas do programa (Mathur, 2008). Cada categoria é particionada em escolhas

(*choices*) que correspondem a um ou mais valores de uma entrada. Restrições nas escolhas podem ser definidas para que sejam selecionados somente conjuntos de teste válidos e razoáveis. No trabalho de Offutt e Irvine (1995), esse método é aplicado em programas OO para demonstrar que técnicas de teste existentes podem ser efetivas em teste de programas OO.

Teste Funcional Sistemático: é uma tentativa de combinar os critérios particionamento em classes de equivalência e análise de valor limite, seguindo um conjunto bem definido de diretrizes (Linkman et al., 2003). Assim que o particionamento ocorre, o teste funcional sistemático exige no mínimo dois casos de teste de cada partição. Além disso, o critério possui uma série de diretrizes para construção dos casos de teste, relacionados a casos especiais, valores ilegais, *arrays*, valores reais, entre outros.

3.3.2 Técnica de Teste Estrutural

O teste estrutural, também chamado de teste caixa-branca, é uma técnica em que o testador deriva os casos de teste por meio da lógica interna do programa (Myers et al., 2004). Em geral, a maioria dos critérios dessa técnica utiliza uma representação de programa conhecida com grafo de fluxo de controle ou grafo de programa. Um programa P pode ser decomposto em um conjunto de blocos disjuntos de comandos tal que a execução do primeiro comando de um bloco acarreta a execução de todos os outros comandos desse bloco.

A representação de um programa P como um grafo de fluxo de controle consiste em estabelecer uma correspondência entre nós e blocos e em indicar possíveis fluxos de controle entre blocos por meio dos arcos. Portanto, um grafo de fluxo de controle (GFC) é um grafo orientado, com um único nó de entrada e um único nó de saída, no qual cada vértice representa um bloco indivisível de comandos e cada aresta representa um possível desvio de um bloco para outro. A partir do GFC podem ser escolhidos os componentes que devem ser executados, caracterizando assim os critérios da técnica de teste estrutural.

No trabalho de Rapps e Weyuker (1985) é proposto o Grafo Def-Uso (*Def-Use Graph*) que consiste em uma extensão do GFC. Nele são adicionadas informações a respeito do fluxo de dados do programa, caracterizando associações entre pontos do programa onde é atribuído um valor a uma variável (definição de variável) e pontos onde esse valor é utilizado (referência ou uso de variável). Os requisitos de teste são determinados com base em tais associações. Dois tipos de usos são distinguidos: c-uso e p-uso. O c-uso afeta diretamente uma computação sendo realizada ou permite que o resultado de uma definição anterior possa ser observado; e o p-uso afeta diretamente o fluxo de controle do programa.

Os critérios que consideram somente características do controle da execução, baseando-se principalmente no GFC, são chamados de critérios baseados no fluxo de controle. Os critérios mais conhecidos dessa classe são:

Todos-Nós: esse critério exige que os casos de teste executem ao menos uma vez cada vértice do GFC, ou seja, cada comando do programa seja executado ao menos uma vez.

Todos-Arcos: esse critério requer que os casos de teste executem ao menos uma vez cada aresta do GFC, ou seja, cada desvio de fluxo de controle do programa.

Todos-Caminhos: esse critério requer que todos os caminhos possíveis sejam executados no GFC.

Os critérios baseados no fluxo de dados utilizam informações do fluxo de dados para determinar os requisitos de teste. Esses critérios exploram definições de variáveis e referências e seus respectivos usos no programa. Segundo Rapps e Weyuker (1985), esses critérios têm origem na intuição de que, assim como não se deve considerar suficientemente testado um programa se todos seus comandos não forem exercitados, não se deve considerar suficientemente testado um programa se todos os resultados computacionais não tiverem sido usados ao menos uma vez. Os critérios baseados em fluxo de dados mais conhecidos, propostos por Rapps e Weyuker (1985), são:

Todas-Definições (*all-defs*): requer que cada definição de variável seja exercitada pelo menos uma vez, por p-uso ou c-uso.

Todos-Usos (*all-uses*): requer que todas as associações entre uma definição de variável e seus subsequentes usos sejam exercitadas pelos casos de teste, através de um caminho livre de definição, ou seja, um caminho onde a variável em questão não é redefinida.

Todos-Du-Caminhos (*all-du-paths*): requer que todos os caminhos livres de definição para toda associação entre uma definição de variável e seus subsequentes usos sejam exercitados pelos casos de teste.

3.3.3 Técnica de Teste Baseado em Defeitos

O teste baseado em defeitos utiliza o conhecimento sobre defeitos mais comuns no processo de desenvolvimento de software para derivar os requisitos de teste. Dois exemplos dessa técnica são a Semeadura de Erros (*Error Seeding*) e a Análise de Mutantes (*Mutation Analysis*).

A **Semeadura de Erros** insere uma quantidade conhecida de defeitos no programa. Após a execução dos testes, por meio do total de defeitos encontrados verificam-se quais são naturais e quais são artificiais. Usando estimativas de probabilidade, o número de defeitos naturais ainda existentes no programa pode ser estimado (Budd, 1981; Ramamoorthy e Bastani, 1982). A semeadura de erros também pode ser usada para medir a eficácia de um critério de teste (Ramamoorthy e Bastani, 1982). Diferentes tipos de defeitos podem ser semeados em um programa e, ao aplicar um conjunto de teste derivado de um determinado critério, os resultados indicam a eficácia do critério e para quais tipos de defeitos o critério é mais adequado.

A **Análise de Mutantes** (AM) (DeMillo, 1978) é um critério de teste que avalia a adequação de um conjunto de casos de teste em revelar defeitos específicos. Para isso, utiliza-se um conjunto de operadores de mutação, os quais geram programas semelhantes ao programa que está sendo testado, mas com algum defeito. Esses programas gerados são chamados de mutantes. Os mutantes são executados com os casos de teste e os resultados são comparados com os resultados do original. O nível de confiança da adequação dos casos de teste é medido pelo escore de mutação, que relaciona o número de mutantes mortos (mutantes que apresentaram resultados diferentes do programa original) com o número de mutantes gerados. Foi demonstrada a eficácia do critério AM para detectar defeitos (Wong et al., 1995; Souza, 1996) e sua utilização para a geração de conjuntos de casos de teste adequados por construção (DeMillo e Offutt, 1991; Simão e Maldonado, 2000). O critério AM vem sendo aplicado em outros contextos como MEFs (Fabbri et al., 1995), Statecharts (Sugeta, 1999), Redes de Petri (Simão, 2000, 2004), especificações WSDL (Siblini e Mansour, 2005) e Programação orientada a aspectos (Ferrari et al., 2008).

3.4 Geração Automática de Casos de Teste

Os critérios de teste podem ser utilizados tanto para definir se a fase de teste está completa quanto para guiar a construção do conjunto de teste (Frankl e Weyuker, 2000). É importante que o testador tenha uma medida que indique se o software já foi suficientemente testado. Critérios de teste podem ser utilizados para essa tarefa, caracterizando os *critérios de adequação de teste*. Os critérios também podem auxiliar o processo de geração, guiando o testador no momento da seleção dos casos de teste. Quando os critérios são utilizados nessa tarefa, eles são referenciados como *critérios de seleção de teste*.

Um dos objetivos do teste de software é automatizar o máximo possível e, assim, reduzir o custo de aplicação, minimizar os erros humanos e facilitar o teste de regressão (Ammann e Offutt, 2008). Durante a geração de casos de teste, é possível implementar critérios de seleção para automatizar o processo. Pesquisas estão sendo desenvolvidas na geração automática de casos de teste. No entanto, existem limitações próprias da atividade de teste que dificultam a geração de casos de teste como, por exemplo, decidir se dois programas são equivalentes ou se um determinado caminho de um programa é executável ou não.

Uma forma de abordar o problema da geração automática é, dado um conjunto de critérios de teste, identificar quais as entradas de teste são necessárias para satisfazer esses critérios. Nesse contexto, os paradigmas mais utilizados são (Michael et al., 2001):

Aleatório: consiste basicamente em gerar dados de entrada aleatoriamente até que alguma entrada útil seja encontrada. O principal problema desse paradigma é que, com programas e critérios de teste complexos, a chance de encontrar uma entrada adequada é muito baixa. No trabalho de Korel (1996), a geração aleatória de teste foi inferior a outros métodos mesmo usando critérios e programas simples.

Simbólico: consiste na execução simbólica do programa para encontrar as entradas adequadas para um determinado critério de teste. A execução simbólica atribui valores simbólicos às variáveis para criar uma caracterização abstrata e matemática do que o programa faz. Assim, idealmente, a geração de teste é reduzida ao problema de resolução de expressões algébricas. O paradigma simbólico apresenta algumas limitações práticas, tais como o tratamento de iterações e *arrays* (Clarke, 1976).

Dinâmico: consiste na ideia de que, se algum requisito de teste não foi satisfeito, dados coletados durante a execução podem ser utilizados para determinar quais testes estão mais próximos de satisfazer o requisito. Com o auxílio desses dados, entradas de teste são incrementalmente modificadas até que uma delas satisfaça o requisito de teste. Métodos de minimização de função e algoritmos genéticos vêm sendo utilizados no paradigma dinâmico (Korel, 1990; Michael et al., 2001).

Além dos trabalhos apresentados, uma abordagem, chamada teste baseado em modelos, vem sendo utilizada recentemente para apoiar a geração automática de casos de teste. Por ser o foco deste trabalho, o teste baseado em modelos é descrito em detalhes no Capítulo 4.

3.5 Avaliação em Teste de Software

Embora muitos fatores afetem o desenvolvimento de um software confiável, teste é o método primário que a indústria utiliza para avaliar um software em desenvolvimento (Ammann e Offutt, 2008). No entanto, a escolha da abordagem mais adequada para testar uma determinada característica do sistema permanece uma questão em aberto (Bertolino, 2004). A existência de estudos teóricos e experimentais que realizam avaliações de critérios de teste pode auxiliar na escolha da abordagem de teste mais adequada.

3.5.1 Estudos Teóricos

Esforços da comunidade vêm sendo gastos no estudo da fundamentação teórica do teste (Goodenough e Gerhart, 1975; Gaudel, 1995; Hierons et al., 2009). Esses estudos buscam fornecer propriedades que deem embasamento teórico para a adoção de um determinado critério de teste. Goodenough e Gerhart (1975) formalizam a noção de um critério de teste ser válido e confiável. Um critério de teste é válido se para todo programa com defeito, o critério de teste é capaz de produzir um conjunto de teste que mostre que o programa contém defeitos. Um critério de teste é confiável se todo conjunto de teste que pode ser produzido usando o critério leve ao mesmo veredito. Hierons et al. (2009) afirmam que o equilíbrio entre os dois extremos, o teste exaustivo e a prova completa do sistema, pode ser alcançado pela definição de suposições sobre o artefato sendo testado que apóiem a seleção de conjuntos de teste menores e eventualmente finitos. Essas

suposições são referenciadas como *hipóteses de seleção de teste*. Gaudel (1995) identificou dois tipos de hipótese: uniformes e regulares. Uma hipótese uniforme está relacionada à uniformidade do comportamento do programa para determinados intervalos de dados. Uma hipótese é regular quando está relacionada à regularidade do comportamento do programa a medida que o tamanho dos dados aumenta.

A comparação teórica entre critérios tem sido apoiada principalmente por uma relação de inclusão e pelo estudo da complexidade dos critérios (Weyuker, 1984; Rapps e Weyuker, 1985; Ntafos, 1988). A relação de inclusão estabelece uma ordem parcial entre os critérios, caracterizando uma hierarquia entre eles. Diz-se que um critério $C1$ inclui outro critério $C2$ se e somente se todo conjunto de teste $C1$-adequado seja também $C2$-adequado. A complexidade é definida como o número máximo de casos de teste requeridos por um critério, no pior caso. No caso dos critérios baseados em fluxo de dados, esses têm complexidade exponencial, o que motiva a condução de estudos experimentais para determinar o custo de aplicação desses critérios do ponto de vista prático (Maldonado et al., 2004).

Os trabalhos de Frankl e Weyuker (1993); Zhu (1996) abordaram do ponto de vista teórico a questão de eficácia de critérios de teste, e definiram outras relações, que captam a capacidade de revelar defeitos dos critérios de teste. Segundo Frankl e Weyuker (1993), o fato de um critério $C1$ incluir um critério $C2$ não garante que $C1$ é melhor em revelar defeitos do que $C2$. Zhu (1996) contesta os resultados de Frankl e Weyuker (1993), argumentando que esses resultados são válidos apenas em um cenário em que os casos de teste são construídos com base no critério utilizado. Considerando um segundo cenário, em que os casos de teste são construídos sem o conhecimento do critério, mostrou-se que a relação de inclusão implica também em maior capacidade em descobrir defeitos (Zhu, 1996).

3.5.2 Estudos Experimentais

Disciplinas de engenharia são caracterizadas pelo uso de conhecimento maduro por meio do qual resultados previstos possam ser alcançados. O tipo de conhecimento usado em teste e, de forma mais ampla, em Engenharia de Software, pode ser considerado relativamente de baixa maturidade (Juristo et al., 2004). Nesse contexto, a realização de estudos experimentais e a pesquisa em como realizá-los em Engenharia de Software vêm sendo uma constante preocupação e foco de pesquisa nos últimos anos (Kitchenham et al., 2002).

Evidências sobre uma determinada hipótese podem ser obtidas por meio da elaboração, execução e análise de estudos experimentais. Os dados que são coletados durante um estudo podem ser quantitativos ou qualitativos (Runeson e Höst, 2009). Os dados quantitativos envolvem números e classes e são analisados usando estatísticas, enquanto que os dados qualitativos envolvem descrições, figuras ou diagramas e são analisados usando categorização e ordenação. Esses dados são coletados durante a realização de alguma estratégia de pesquisa. Wohlin et al. (2000) classificam as estratégias de pesquisa em três tipos principais:

***Survey*:** é a coleção de informações padronizadas de uma população específica (ou uma amostragem dela) realizada, em geral, por meio de questionários ou entrevistas.

Estudo de caso: é um estudo puramente observacional que objetiva investigar fenômenos em seus contextos. Estudos de caso são usados para monitorar projetos, atividades ou atribuições, coletando dados para um propósito específico durante o estudo.

Experimento controlado: é caracterizado pela medida dos efeitos de manipular uma variável em relação a outra. Nos experimentos controlados, os assuntos são atribuídos aos tratamentos de forma aleatória. Os *quase experimentos* são similares aos experimentos com a diferença de que os assuntos não são atribuídos aleatoriamente (Runeson e Höst, 2009).

No contexto de teste, três aspectos costumam ser avaliados nos estudos experimentais: custo, eficácia e *strength* (ou dificuldade de satisfação) (Weyuker et al., 1991; Wong e Mathur, 1995). O custo reflete o esforço necessário para que o critério seja utilizado; a eficácia refere-se à capacidade que um critério possui de detectar a presença de defeitos; e o *strength* refere-se à probabilidade de satisfazer-se um critério tendo sido satisfeito um outro critério (Mathur e Wong, 1994). O custo e a eficácia são os aspectos mais presentes em estudos experimentais em teste de software principalmente pela necessidade de medir a relação entre ambos (Briand, 2007).

O **custo** é em geral medido pelo número de casos de teste, embora nem sempre seja uma medida adequada. O custo de teste envolve pelo menos duas dimensões: o esforço humano e custo computacional, embora o primeiro seja mais crucial (Briand, 2007). Outras medidas devem ser consideradas como as relacionadas a tempo, tais como o tempo de projeto e de execução do conjunto de teste. A geração e a codificação de toda a infraestrutura necessária para executar os testes também devem ser consideradas.

A **eficácia** é em geral medida pelo número de defeitos encontrados pelos testes (Briand, 2007). A eficácia pode ser medida relacionando o número de defeitos encontrados com o número total de defeitos. Nesse caso, o número total de defeitos é conhecido, pois esses foram introduzidos nos programas. Na realização dos estudos, os defeitos podem ser obtidos de uma base histórica (defeitos reais) ou inseridos propositadamente (defeitos semeados) (Andrews et al., 2005; Briand, 2007). Os defeitos reais são mais complicados de serem obtidos e são encontrados em menor número. Esse fato limita a análise estatística de um estudo realizado com defeitos reais, mas possui maior representatividade do mundo real. Os defeitos semeados vêm sendo constantemente adotados pela dificuldade de obter defeitos reais e em uma quantidade razoável. Esses defeitos podem ser inseridos de forma manual ou automatizada. Na forma manual, engenheiros de software inserem (semeiam) manualmente defeitos nos programas que serão testados. Na forma automatizada, a análise de mutantes é adotada não como critério de adequação, mas sim como mecanismo para gerar inúmeras versões com defeito (mutantes). O uso de teste de mutação pode ser uma estratégia interessante, tendo em vista que Andrews et al. (2005) mostraram por meio de um experimento

que os mutantes são similares a defeitos reais. Outro resultado obtido é que defeitos manualmente semeados são mais difíceis de detectar do que defeitos reais e mutantes.

No contexto de teste de software, os estudos experimentais mostram-se mais importantes, já que obter uma prova teórica da eficácia é uma tarefa muito difícil ou, em alguns casos, impossível. Pesquisas que relatam e analisam o estado atual da experimentação em teste de software vêm sendo desenvolvidas (Juristo et al., 2004; Do et al., 2005). Juristo et al. (2004) apresentam uma análise do nível de conhecimento na área de teste de software baseada nos estudos experimentais encontrados na literatura. Os estudos foram agrupados de acordo com a técnica de teste avaliada e se a comparação foi feita entre critérios da mesma técnica ou de técnicas diferentes. Para cada grupo, os trabalhos foram sumarizados apresentando os critérios comparados, uma recomendação prática, o nível de maturidade dos resultados e quais conhecimentos precisam de mais avaliação. A principal conclusão desse trabalho é que o conhecimento na área de teste é limitado, e que até o momento da realização do estudo, não existia conhecimento formalmente testado e mais da metade do conhecimento existente é baseado em impressões e percepções.

Do et al. (2005) analisam 107 artigos sobre estudos experimentais na área de teste selecionados nas principais conferências e revistas de Engenharia de Software. Os autores identificaram um pequeno número de experimentos controlados, além de uma pequena porcentagem de estudos que utilizam vários programas, versões, dados sobre defeitos e compartilhamento limitado de artefatos. No entanto, foi notado um crescente interesse por parte dos pesquisadores na realização de experimentos controlados.

3.6 Considerações Finais

Neste capítulo foi apresentada uma visão geral sobre teste de software. Os principais fundamentos de teste de software foram apresentados inicialmente. Os critérios de teste das técnicas funcional, estrutural e baseada em defeitos foram apresentados. É também abordada a geração automática de casos de teste. Por fim, foram apresentados estudos teóricos e experimentais sobre como avaliar e comparar critérios.

O teste de software apresenta-se como uma atividade fundamental para a garantia de qualidade aplicada durante todo o processo de desenvolvimento. Existem inúmeros critérios de teste propostos dentro de diferentes técnicas de teste. Nesse contexto, abordagens estão sendo desenvolvidas de forma a utilizar modelos formais do software para derivar casos de teste de forma automatizada. No próximo capítulo, essa abordagem também conhecida como teste baseado em modelos é apresentada em detalhes.