Orientações para Desenho e Implementação de Testes de Software

A Disciplina de Testes é crítica para o desenvolvimento de software. Embora as revisões técnicas sejam mais eficazes na detecção e remoção de defeitos, os testes são importantes para complementar as revisões e aferir a qualidade alcançada no produto. A implementação de testes é, quase sempre, limitada por restrições de cronograma e orçamento; eles determinam quantos testes será possível executar. É importante que os testes sejam bem planejados e desenhados, para se conseguir o melhor proveito possível dos recursos alocados para eles.

Deve-se maximizar a cobertura dos testes, ou seja, a quantidade potencial de defeitos que podem ser detectados por meio dos testes, de forma a se conseguir detectar a maior quantidade possível de defeitos que não foram apanhados pelas revisões, considerando-se os custos e prazos previstos.

Para serem eficazes, os testes devem ser cuidadosamente planejados e desenhados. Testes irreproduzíveis e improvisados são quase inúteis e devem ser evitados. Os resultados de cada teste devem ser minunciosamente inspecionados, comparando-se resultados previstos e obtidos, pois nem sempre é óbvio quando um teste detectou um defeito.

Os testes automatizados devem ter implementação mais simples possível para ser fácil de entender e alterar, porque nesse tipo de teste é software testando software, e, portanto, podem inserir tantos erros no produto quanto a própria implementação do código.

Os desenvolvedores não são as pessoas mais adequadas para testar seus próprios produtos e quase sempre são maus testadores. Assim como nas revisões, os autores dos artefatos têm maior dificuldade de enxergar os problemas, se comparados com pessoas que não participaram do desenho e da implementação desses artefatos. Torna-se importante, portanto, ter testadores no time, isto é, pessoas cuja habilidade primária é testar – e não necessariamente ter a função exclusiva de testar. Se o time está usando TDD (*Test-Driven Development*), o testador poderia fazer par com o desenvolvedor enquanto eles estão escrevendo códigos de teste, uma vez que bons testadores normalmente adicionam tipos diferentes de testes aos de um bom desenvolvedor.

Existem basicamente duas maneiras de se construir testes:

- **Método caixa branca**: tem por objetivo determinar defeitos na estrutura interna do produto, por meio do desenho de testes que exercitem suficientemente os possíveis caminhos de execução.
- **Método caixa preta**: tem por objetivo determinar se os requisitos foram totalmente satisfeitos pelo produto por meio da verificação dos resultados produzidos.

Testes de Aceitação

Testes de Aceitação têm por objetivo validar o produto, ou seja, verificar se ele atende aos requisitos especificados, tanto os funcionais quantos os não-funcionais.

Testes funcionais

Os **testes funcionais** são desenhados para verificar a consistência entre o produto implementado e os respectivos requisitos funcionais. Por isso, a completeza e a precisão da especificação de requisitos são fundamentais para a qualidade desses testes. Serão tratados neste guia das três principais técnicas de desenho de testes funcionais: partição de equivalência, análise do valor limite e os testes de comparação.

Partição de Equivalência

A partição de equivalência é um método que divide o domínio de entrada em categorias de dados. Cada categoria revela uma classe de erros, permitindo que casos de teste na mesma categoria sejam eliminados sem que se prejudique a cobertura dos testes.

Para cada entrada do sistema, são identificados os conjuntos de valores válidos e inválidos associados que definem as classes de equivalência para essa entrada. As classes de equivalência podem ser derivadas, como se vê na tabela abaixo:

Definição da entrada	Classes de equivalência
Intervalo válido	Uma válida, para os valores pertencentes ao intervalo; duas inválidas, para os valores menores e maiores que os limites inferior e superior, respectivamente.
Lista de valores válidos	Uma válida, para os valores incluídos na lista; uma inválida, para todos os outros valores.
Valor específico	Uma válida, que inclui o valor; duas inválidas, para os valores maiores e menores.
Lógica	Uma válida e uma inválida.

Análise de valor limite

Em geral, erros nas fronteiras do domínio da entrada são mais frequentes do que nas regiões centrais. A análise do valor limite é uma técnica para a seleção de casos de teste que exercitam os limites. O emprego dessa técnica deve ser complementar ao emprego da partição de equivalência. Assim, em vez de se selecionar um elemento aleatório de cada classe de equivalência, selecionam-se os casos de teste nas extremidades de cada classe.

A seleção dos casos de teste deve ser feita de acordo com as recomendações da tabela abaixo:

Entradas válidas	Casos de teste
Intervalo delimitado pelos valores *a* e *b*	Valores imediatamente abaixo de *a*; *a*; *b*; e imediatamente acima de *b*.
Série de valores	Valor imediatamente abaixo do mínimo, para o mínimo, para o máximo e imediatamente acima do máximo.
Intervalo ou série de valores	Saídas máxima e mínima definidas.
Estruturas de dados	Caso que exercite a estrutura em suas fronteiras.

Testes de comparação

Existem situações em que é necessário comparar as saídas de diferentes versões de um sistema quando submetidas às mesmas entradas. Esses testes se aplicam a situações como:

- Uso de sistemas redundantes para aplicações críticas;
- Comparação de resultados de produtos em evolução.

Quando produtos são substituídos por versões mais novas que incluem mais funcionalidades, devem-se comparar os resultados das características que fazem parte de ambas as versões. Essas características já foram testadas pelos usuários com dados reais e tendem a estar mais estabilizadas. Se as saídas forem consistentes, presume-se que todas as versões testadas estejam corretas. Caso contrário, deve-se investigar em qual ou quais das versões se encontra o defeito. A comparação das saídas pode ser feita com o auxílio de uma ferramenta automatizada.

Testes não-funcionais

Os **testes não-funcionais** procuram detectar se o comportamento do sistema está consistente com a respectiva especificação de requisitos quanto aos aspectos não-funcionais. Esses testes cobrem, por exemplo, os aspectos mostrados na tabela abaixo:

Tipos de requisitos	Tipos de testes
Desempenho	Número de usuários simultâneos. Volume de informação que deve ser tratado.
Persistência	Frequência de uso Restrições de acesso Restrições de integridade Requisitos de auditoria

	Requisitos de cópia de segurança e restauração de dados
Restrição ao desenho	Formatos de impressão, responsividade, acessibilidade
Atributos de qualidade	Funcionalidade Confiabilidade Usabilidade Manutenibilidade Portabilidade
Aspectos de sistema	Instalabilidade Integridade de bancos de dados Recursos de suporte

Testes de Integração

Testes de Integração têm por objetivo verificar as interfaces entre as partes de uma arquitetura de produto, se as unidades implementadas em cada iteração funcionam corretamente em conjunto com as unidades já implementadas em iterações anteriores, realizando corretamente os casos de uso e histórias de usuário que se quer entregar nessa iteração.

De maneira geral, os testes de integração são subconjuntos dos testes de aceitação, uma vez que verificam se as versões parciais do produto entregues em cada Sprint satisfazem os respectivos requisitos funcionais, e talvez a alguns requisitos não-funcionais. Em alguns casos, há necessidade de utilizar cotos (*stubs*) de classes, possivelmente aproveitando os mesmos componentes usados nos testes de unidades, se em uma Sprint não forem implementadas de maneira completa todas as classes que fazem parte da iteração.

Normalmente, a preparação dos testes se inicia pelos de mais alto nível, ou seja, os testes de aceitação, baseados principalmente na especificação de requisitos. Desses testes, do planejamento das *sprints* e do desenho arquitetônico do sistema, são derivados os testes de integração. O desenho detalhado das unidades, realizado durante cada Sprint, serve de base para o desenho dos testes de unidade.

Testes de Unidade

Testes de Unidade têm por objetivo verificar um elemento que possa ser logicamente tratado como uma unidade de implementação, tipicamente uma classe ou um grupo de classes correlatas. Por convenção, são testadas por baterias de testes apenas unidades especialmente críticas, cuja falha pode acarretar consequências graves, como risco de vida ou perdas materiais vultuosas.

O teste de unidade é do tipo caixa branca; ele exercita detalhadamente uma unidade de código. Um dos pontos fracos desse tipo de teste é que ele geralmente é desenhado pelos próprios desenvolvedores, já que, na maioria das vezes, são os únicos que conhecem bem o desenho e o código das unidades a serem testadas. Entretanto, se forem bem planejadas e executados, os testes de unidade poderão detectar cerca de 70% dos defeitos que seriam encontrados pelos usuários (McConnell, 1996). O Modelo da Solução (diagramas UML, casos de uso, histórias de usuário, critérios de aceitação e regras de negócio) e o próprio código da unidade que está sendo testada devem ser as principais referências do desenho dos testes de unidade.

Teste de Caminhos Básicos

Os testes de caminhos básicos têm por objetivo verificar os caminhos independentes e os caminhos de tratamento de erros. Para verificar os caminhos independentes, todas as instruções da unidade sob teste devem ser executadas pelo menos uma vez. É difícil selecionar um conjunto de caminhos que ofereça uma cobertura adequada dos possíveis defeitos. Entretanto, as diretrizes seguintes são geralmente úteis:

- Selecionar os caminhos que ligam o início ao fim da execução de cada procedimento ou método.
- Selecionar caminhos que fazem pequenas variações, utilizando como referência o detalhamento dos elementos do Modelo da Solução.
- Selecionar caminhos sem significado funcional, para verificar se não existem combinações de entradas que possam provocar a ativação desses caminhos.

Teste de Condições

Os testes de condições têm por objetivo detectar erros nas condições contidas na unidade que está sendo testada e assegurar que ela opera adequadamente nos limites estabelecidos, incluindo, entre outros aspectos, número de entradas e limites das entradas e saídas válidas. Os casos de teste são desenhados para que as condições assumam valores que exercitem o máximo de caminhos possíveis, ou pelo menos os mais relevantes.

Teste de Laços

Os testes de laços verificam os laços da unidade sob teste. Esse tipo de teste verifica também as condições de limite da unidade sob teste: um defeito comum nos laços que percorrem um conjunto é errar, para cima ou para baixo, o número de elementos do conjunto.

Para um laço simples cujo número máximo de iterações possíveis é *n*, casos de teste para as situações enumeradas a seguir devem ser desenhados:

- Não executar o laço;
- Executar apenas uma iteração do laço;
- Executar m iterações pelo laço, em que m < n;
- Executar n-1, n+1 iterações.

Testes de Regressão

O código de um produto sofre alterações ao longo de um projeto, para correção dos defeitos detectados, ou por causa da evolução do desenho ao longo das *sprints*, e, principalmente, durante as fases de manutenção. Alterações em certas unidades podem causar modificações do comportamento de outras unidades já testadas, por causa dos efeitos colaterais; um bom desenho diminuirá a probabilidade de que isso aconteça, mas dificilmente eliminará completamente os efeitos colaterais. Os **testes de regressão** verificam novamente as unidades já testadas, para checar se eles continuam funcionando de maneira apropriada após as alterações.

Uma bateria de testes de regressão consiste em um conjunto selecionado de testes de boa cobertura, que é executado periodicamente, de forma automatizada. Caso essa bateria seja muito grande, pode-se utilizar uma bateria menor com testes selecionados, para uso mais frequente, testando-se a bateria completa em intervalos maiores. Embora não haja um modelo predefinido para este produto de trabalho e as ferramentas de teste afetem como o produto de trabalho é manipulado, devem ser abordados os seguintes problemas:

1. Configurar ambiente de testes
2. Povoar banco de dados com dados para os testes
3. Exercitar roteiro de testes
4. Analisar os resultados dos testes conforme os critérios de aceitação
5. Limpar banco de dados de testes