PUC Minas

Resenha

Projeto de Software

Gustavo Delfino Guimarães

1489062@sga.pucminas.br

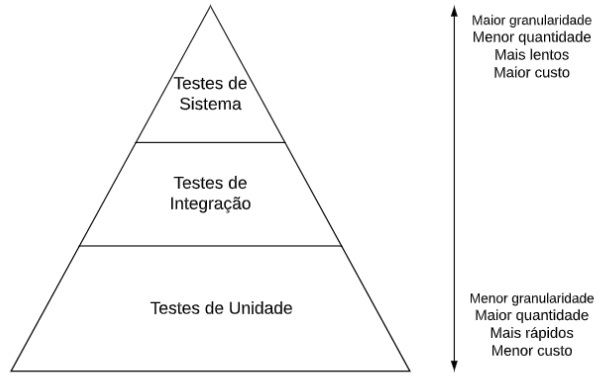
Matrícula: 836079

Belo Horizonte, MG – Fevereiro de 2025

Engenharia de Software Moderna

Capítulo 8: Testes

O software é uma das construções mais complexas da humanidade e, por isso, está sujeito a erros e inconsistências. Para evitar prejuízos, os testes de software tornaram-se uma prática essencial no desenvolvimento moderno. No modelo em cascata, os testes eram realizados manualmente em uma fase separada e por equipes especializadas, com o principal objetivo de detectar bugs antes da produção. Com a adoção dos métodos ágeis, essa abordagem mudou radicalmente: os testes passaram a ser automatizados, sendo desenvolvidos pelos próprios programadores, muitas vezes antes da implementação das classes. Além de detectar erros, os testes agora garantem a estabilidade do código e servem como documentação.



Pirâmide de testes

Uma das formas mais utilizadas para classificar testes automatizados é a pirâmide de testes, que os divide em três níveis. Na base, os testes de unidade verificam pequenas partes do código e são os mais comuns, devido à sua rapidez e facilidade de implementação. No nível intermediário, os testes de integração validam a interação entre diferentes módulos do sistema, incluindo componentes externos, e exigem mais esforço. No topo da pirâmide, os testes de sistema simulam o uso real do software por um usuário, sendo os mais custosos e frágeis. A recomendação geral é que 70% dos testes sejam de unidade, 20% de integração e 10% de sistema. Esse capítulo foca nos testes automatizados, destacando sua importância na manutenção da qualidade e confiabilidade do software.

Testes unitários são uma prática fundamental no desenvolvimento de software, permitindo a verificação automatizada de pequenas unidades de código, normalmente classes isoladas do restante do sistema. Estes testes garantem que métodos de uma classe retornem os resultados esperados, dividindo o código do sistema entre classes funcionais e seus respectivos testes. Implementados com frameworks como JUnit, os testes unitários oferecem rapidez e facilidade de execução, além de possibilitar a detecção imediata de falhas durante o desenvolvimento. O JUnit, baseado na arquitetura xUnit, permite que desenvolvedores utilizem a mesma linguagem do sistema para escrever testes, tornando a adoção dessa prática mais acessível.

A estrutura dos testes unitários segue um padrão: criar um contexto de teste (instanciar objetos), chamar métodos da classe testada e verificar os resultados usando asserções. O JUnit executa cada teste em uma instância separada da classe de teste, utilizando a anotação @Before para preparar o ambiente antes da execução de cada @Test. Além

disso, a ferramenta permite testar exceções esperadas, como no caso de uma pilha vazia levantando uma EmptyStackException. Essa abordagem automatizada reduz retrabalho, melhora a qualidade do código e facilita a manutenção ao longo do ciclo de vida do software.

Estes testes unitários são muito importantes e começamos a ter contato com ele desde o primeiro semestre da faculdade em AEDS e continua aprimorando em Programação Modular, para assim melhorar as funções e classes.
Os testes unitários podem ser escritos após a implementação da funcionalidade ou antes, sendo antes um desenvolvimento orientado a testes. Deixar para fazer testes ao final da implementação é considerado uma atitude ruim no desenvolvimento de software.

O principal benefício dos testes de unidade é a detecção precoce de bugs, evitando que erros cheguem à produção. Além disso, eles atuam como proteção contra regressões, garantindo que alterações no código não introduzem novos problemas.

Os Princípios e Smells abordam anti-padrões e princípios para a implementação de testes de unidade, visando sua qualidade e manutenibilidade.

Os testes unitários devem seguir o princípio FIRST.

- **Fast:** Execução rápida para ter um feedback imediato sobre bugs e regressões.
- **Independentes:** Não deve haver sequência de execução de testes, eles devem ser independentes e sem influência mútua.
- **Repeatable:** Os testes devem retornar sempre o mesmo resultado, evitando testes flaky.
- **Self-checking:** O resultado deve ser facilmente verificável e dependendo da IDE deve ser claro e visual.
- **Timely:** Os testes devem ser escritos o quanto antes, até mesmo antes da escrita do código.

Test Smells são características indesejáveis no código de testes:

- **Teste Obscuro:** Testes longos e complexos que dificultam a compreensão.
- **Teste com Lógica Condicional:** Testes que incluem comandos condicionais, prejudicando a linearidade e a clareza.
- **Duplicação de Código em Testes:** Quando há repetição de código em vários métodos de teste.

Esses smells não devem ser evitados a todo custo, mas sim encarados como alertas para melhorar a simplicidade e a clareza dos testes. A refatoração regular do código de testes é recomendada.

Cobertura de Testes é uma metrificação que quantifica a porcentagem de comandos de um programa que são executados durante os testes. Cobertura de testes = número de comandos executados pelos testes / total de comandos do programa. Algumas ferramentas de cobertura de testes destacam as áreas de código cobertas por testes, evidenciando a eficiência do mesmo.

Não há um valor absoluto ideal de cobertura de testes; o percentual varia conforme o projeto e sua complexidade. Embora 100% de cobertura seja desejável, não é sempre

necessário, especialmente para métodos simples. O importante é monitorar a cobertura ao longo do tempo e focar em áreas críticas que não são testadas.

Valores de cobertura próximos de 70% são comuns em equipes que priorizam testes, enquanto abaixo de 50% é preocupante. Estatísticas de empresas, como o Google, mostraram que a média de cobertura é de cerca de 78%, com uma recomendação de 85%.

Testabilidade se refere à facilidade de implementar testes em um sistema. Um código bem projetado, que respeita princípios SOLID, tende a ser mais testável.

Mocks são objetos que imitam o comportamento de outros objetos, facilitando testes de unidade sem a necessidade de acessar serviços externos, como bancos de dados ou APIs remotas. Eles ajudam a manter os testes rápidos e focados em uma única unidade de código.

Alguns autores distinguem mocks e stubs com base na funcionalidade. Mocks podem verificar tanto o estado quanto o comportamento, enquanto stubs são usados apenas para fornecer dados de retorno. No entanto, esta obra opta por não fazer essa distinção.

Apesar de sua utilidade, os mocks podem aumentar o acoplamento entre testes e implementações, tornando os testes mais frágeis. Mudanças na implementação podem exigir ajustes nos testes que utilizam mocks. Além disso, certas construções, como classes e métodos finais, métodos estáticos e construtores, não podem ser mockadas.

Os testes de integração, ou testes de serviços, estão em um nível intermediário na pirâmide de testes. Diferente dos testes unitários, que verificam pequenas partes do código isoladamente, os testes de integração avaliam funcionalidades maiores, envolvendo múltiplas classes, dependências e serviços reais, como bancos de dados e APIs externas. Por essa razão, não faz sentido utilizar mocks ou stubs nesses testes. Como são mais complexos e demorados, são executados com menor frequência.

No topo da pirâmide, os **testes de sistema** simulam o uso real de um sistema, sendo também chamados de testes ponta-a-ponta (end-to-end). Eles garantem que todas as partes do software funcionam juntas corretamente, mas são mais caros e demorados para executar.

O framework Selenium permite automatizar testes em aplicações web, simulando interações como preenchimento de formulários e cliques em botões. Um exemplo prático envolve um robô que pesquisa "software" no Google e valida o título da página de resultados. No entanto, testes de interface são mais frágeis, pois mudanças visuais na aplicação podem quebrá-los.

Os testes de sistema para compiladores são mais diretos, pois envolvem apenas entrada e saída de arquivos. O teste consiste em compilar programas em uma linguagem específica, executar os binários gerados e comparar a saída com os resultados esperados. No entanto, localizar a causa de falhas nesses testes pode ser desafiador, pois envolve todo o compilador.

Em resumo, os testes de integração garantem a interação correta entre componentes, enquanto os testes de sistema validam a experiência completa do usuário.

Além dos testes citados existem outros testes não menos importantes, também já havia certo conhecimento sobre eles pois já foram tratados em algumas aulas de Introdução a Engenharia de Software, AEDS e Programação modular

Caixa-preta: Os testes são baseados apenas na interface do sistema, sem conhecimento da sua implementação interna. São chamados de testes funcionais.

Caixa-branca: Os testes consideram a estrutura interna do código, explorando sua lógica e cobertura. São chamados de testes estruturais.

Desempenho: Simula alta carga para avaliar a resposta do sistema (exemplo: Black Friday).

Usabilidade: Analisa a interface e a experiência do usuário.

Testes de Falha: Simulam falhas de componentes ou serviços críticos para verificar a resiliência do sistema.

Também existem técnicas para a seleção de dados de teste, já que testes exaustivos são inviáveis na prática na grande maioria dos códigos.

Duas técnicas auxiliam na escolha das entradas para os testes:

- **Partição via Classe de Equivalência:** Divide os valores possíveis em grupos (classes de equivalência) com comportamento semelhante, testando apenas um valor de cada classe.
- **Análise de Valor Limite:** Testa valores próximos aos limites das classes de equivalência, pois erros costumam ocorrer nesses pontos.

Ao final, e mais importante na minha opinião é o **teste de aceitação por parte do cliente**, onde o cliente irá testar o sistema e avaliar se ele atende os requisitos e suas necessidades.