

Gestão de Software

UC: Gestão e Qualidade de Software

Prof. Eliane Faveron Maciel

UNIFACS
Ecosistema ânima

4 de abril de 2024

Overview

1. Boas práticas no git

2. Revisões

3. Testes de Software

3.1 Técnicas de testes

3.2 Níveis de Testes

4. Testes de Unidade

5. Exemplos de teste

6. Cobertura de testes

7. Prática

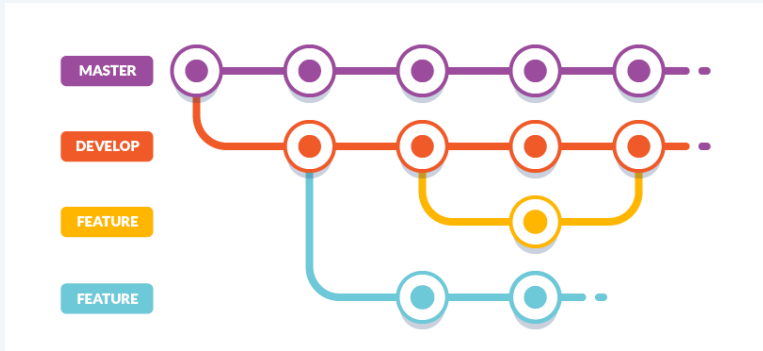
8. Referências

Git e GitFlow

Empresas utilizam ferramentas de versionamento de código como GitHub, GitLab, BitBucket, essas por sua vez possibilitam que o código seja revisado antes que seja incorporado para o core do sistema.

1. Branch (ramo) - significa que você diverge da linha principal de desenvolvimento e continua a trabalhar sem alterar essa linha principal.
2. Commits - registra alterações em um ou mais arquivos no seu branch
3. Pull - é usado para buscar e baixar conteúdo de repositórios remotos e fazer a atualização imediata ao repositório local para que os conteúdos sejam iguais.
4. Push - é usado para gravar em um repositório remoto.
5. Merge - é o jeito do Git de unificar um histórico bifurcado

GitFlow



Boas práticas no git

Dica: Nomeando as branches

- Letras minúsculas e separadas por hífen: Por exemplo, feature/new-loginou bugfix/header-styling.
- Caracteres alfanuméricos: use apenas caracteres alfanuméricos (az, 0-9) e hífen.
- Sem hífen contínuos: não use hífen contínuos.
- Sem hífen no final: não termine o nome da sua filial com um hífen.
- Descritivo: O nome deve ser descritivo e conciso, refletindo idealmente o trabalho realizado na filial.

Dica: Nomeando as branches

- **Features:** são usadas para desenvolver novos recursos. Use o prefixo **feature/**. Por exemplo, **feature/login-system**.
- **Bugs:** são usados para corrigir *bugs* no código. Use o prefixo **bugfix/**. Por exemplo, **bugfix/header-styling**.
- **Hotfix:** São feitas diretamente a partir da branch de produção para corrigir *bugs* críticos. Use o prefixo **hotfix/**. Por exemplo, **hotfix/critical-security-issue**.
- **Release:** são usadas para preparar uma nova versão. Use o prefixo **release/**.
- **Documentação:** Esses ramos são usados para escrever, atualizar ou corrigir documentação. Use o prefixo **docs/**. Por exemplo, **docs/api-endpoints**.

Dica: Nomeando as branches

Além dessas dicas é comum empresas utilizarem o código da tarefa que está relacionada a esse código.

exemplo

<prefixo>/<id da tarefa>-<descrição>

feature/sq-63071-daken-alterar-cadastro-de-usuario-na-api-daken

Dica: Conventional commits

Um padrão para criar mensagens de commit com uma estrutura e formatação específicas. Utiliza vários padrões para classificar e identificar alterações. Isso facilita a compreensão de como uma alteração impacta o projeto. São vários padrões, como *"feat"*, *"fix"*, *"docs"*, *"style"*, *"refactor"*, *"perf"*, *"test"*, *"build"* e *"ci"*.

exemplo

feat:<id da tarefa> - <descrição>

feat:(JIRA-ID): Adiciona navegação por gestos

Dica: Convencional commits

- feat: para introduzir uma nova funcionalidade
- fix: para corrigir um bug
- docs: para documentação
- style: para atualização de estilos CSS
- refactor: para refatoração de código já existente
- test: para adicionar, remover, refatorar, melhorar e corrigir testes
- build: para alterações no processo de compilação
- perf: para alteração do projeto melhorando o desempenho e otimizando uso de recurso computacional
- ci: para alterações no processo de integração contínua

Revisões

Revisões Informais

“As revisões informais incluem um teste de mesa simples de um artefato de engenharia de software (com um colega), uma reunião informal (envolvendo mais de duas pessoas) com a finalidade de revisar um artefato [Pressman, 2021].”

1. Teste de mesa;
2. Programação em pares;

Revisão Técnica Formal

O objetivo de um revisão técnica formal (RTF) é encontrar erros antes que eles ocorram.

1. descobrir erros na função, na lógica ou na implementação, para qualquer representação do software;
2. verificar se o software sob revisão satisfaz seus requisitos;
3. garantir que o software tenha sido representado de acordo com padrões predefinidos;
4. conseguir software que seja desenvolvido de modo uniforme;
5. tornar os projetos mais administráveis.

O papel de revisor

”Todo código desenvolvido por um desenvolvedor tem que ser, em seguida, analisado por pelo menos um outro desenvolvedor, chamado de **revisor**. O **revisor** pode adicionar comentários no código sob revisão, procurando esclarecer dúvidas, sugerindo melhorias, indicando bugs, etc.”

1. Encontro de defeitos – para achar bugs;
2. Melhoria de código – para melhorar a consistência do código, legibilidade, etc.;
3. Soluções alternativas – para encontrar uma melhor implementação;
4. Transferência de conhecimento – para fins de aprendizagem;
5. Conscientização e transparência da equipe – para tornar a equipe ciente da evolução do código e tornar as alterações do código transparentes.
6. Compartilhamento de posse do código – relacionado a “Conscientização e Transparência da Equipe”, mas com conotação de colaboração.

Recomendações aos revisores

1. Revisores sempre devem lembrar que o objetivo da revisão é detectar problemas inequívocos no código submetido.
2. Evite comentários subjetivos e relacionados a estilos pessoais.
3. Sempre restrinja seus comentários ao código que foi submetido e evite tratar de assuntos pessoais.
4. Sempre que possível, use emojis, pois eles deixam a linguagem mais coloquial e amigável.
5. Não deixe de elogiar o código submetido.
6. Sempre que for razoável, use o pronome nós ou a expressão a gente, em vez de usar o pronome você.
7. Se você tiver uma divergência muito forte em relação ao código submetido, tente agendar uma reunião com o autor para expor sua visão e tentarem chegar a um consenso.

Recomendações aos autores


1. Revisão de código não é um exame de proficiência. Ou seja, como autor, não leve a revisão para o lado pessoal e nunca imagine que o revisor está julgando sua competência.
2. Submeta PRs pequenos, caso queiram obter uma resposta rápida e mais proveitosa dos revisores. Por exemplo, os autores do livro *Software Engineering at Google*, recomendam que um PR deve ter no máximo 200 linhas de código.
3. Se PRs forem muito grandes, existe também a chance de a qualidade da revisão cair muito.

Pull Requests


É uma proposta para mesclar as alterações de um branch em outro. Em uma pull request, os colaboradores podem revisar e discutir o conjunto de alterações proposto antes de integrá-las à base de código principal.

estacionamento.java Outdated


```
2 +  
3 + public class Estacionamento {  
4 +  
5 +     public Hashtable<String, String> veiculos;
```

 **mtov** 5 days ago 😊 ...


Sugestão: "ocultar" essa tabela hash dos clientes da classe, tornando-a privada. E, também, criar um método "estaciona" que deve ser chamado toda vez que um novo cliente quiser estacionar um veículo.

 **alinedtorres** 3 days ago Author Owner 😊 ...

Agradeço a sugestão! Acabei de realizar as alterações sugeridas.

 **mtov** now 😊 ...

Obrigado, LGTM!



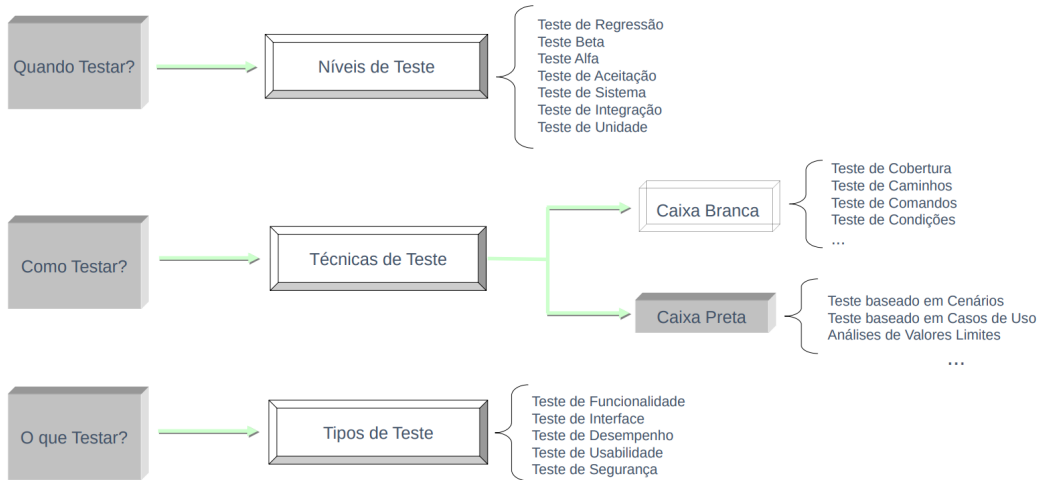
Automatizando a review

1. CodeFactor
2. Codacy
3. CodeClimate
4. SonarLint
5. Code Rabbit

Testes de Software

O que é?

O software é testado para revelar erros cometidos inadvertidamente quando ele foi projetado e construído. Uma estratégia de teste de componentes de software considera o teste de componentes individuais e a sua integração a um sistema em funcionamento.



Caixa branca vs Caixa preta

O teste **caixa-branca**, também chamado de teste da caixa-de-vidro ou teste estrutural, é uma loso a de projeto de casos de teste que usa a estrutura de controle descrita como parte do projeto no nível de componentes para derivar casos de teste.

O teste **caixa-preta** concentra-se nos testes com uma perspectiva de usuário. Não precisando entender todo o funcionamento do sistema.

Qual o objetivo do teste?

- Validar se um programa está em conformidade com os requisitos.
- Medir se a entrega possui qualidade.
- Localizar erros significativos dentro do prazo definido e verifica se as correções aplicadas estão em conformidade com os requisitos.
- Poder melhorar a qualidade e satisfação do cliente.

Níveis de Testes

1. Testes de Unidade
2. Testes de Integração
3. Testes de Sistema
4. Testes de Aceitação

Estratégia para o sucesso do teste de software

No livro de [Pressman, 2021], cita que o teste de software terá sucesso se seguir os itens a seguir:

1. Especificar os requisitos do produto de uma maneira quantificável;
2. Definir os objetivos do teste;
3. Entender os usuários do software e desenvolverem um perfil para cada categoria de usuário;
4. Desenvolver um plano de teste;
5. Criar software “robusto” que seja projetado para testar-se a si próprio;
6. Usar revisões técnicas como filtro antes do teste;
7. Realizar revisões técnicas para avaliar a estratégia de teste e os próprios casos de teste;
8. Desenvolver abordagem de melhoria contínua para o processo de teste

Testes de Unidade

Teste de Unidade

”O teste de unidade focaliza o esforço de verificação na menor unidade de projeto do software [Pressman, 2021].”

O propósito principal dos testes é ajudar os desenvolvedores a descobrir defeitos antes desconhecidos. É importante elaborar casos de teste que exercitam as capacidades de manipulação de erros do componente.

Casos de teste negativos: Para descobrir novos defeitos, também é importante produzir casos de teste que testem que o componente não faz algo que não deveria fazer.

Características do teste unitário

- É necessário ter uma plataforma e um ambiente de teste ou depuração para executar o código.
- Ferramentas de cobertura de código podem ajudar a verificar se todos os caminhos foram percorridos.
- Os caminhos de erro mais complexos costumam ser negligenciados, o que pode ter consequências dispendiosas mais tarde no ciclo de desenvolvimento.

Figura: Enter Caption

Testes de unidade

Testes de unidade são implementados usando-se **frameworks** construídos especificamente para esse fim. [Valente, 2020].

- Python: Pytest ou unittest
- JavaScript: JestJS
- Java: JUnit

(Os exemplos geralmente serão em python ou javascript, podendo ter algum retirado de livros em java.)

Conceitos importante

- **Teste**: método que implementa um teste.
- **Fixture**: estado do sistema que será testado por um ou mais métodos de teste, incluindo dados, objetos, etc.
- **Casos de Teste (Test Case)**: classe com os métodos de teste.
- **Suíte de Testes (Test Suite)**: conjunto de casos de teste, os quais são executados pelo framework de testes de unidade.
- **Sistema sob Teste (System Under Test, SUT)**: sistema que está sendo testado.

Propriedade FIRST

- **Rápidos (Fast):** desenvolvedores devem executar testes de unidades frequentemente, para obter feedback rápido sobre bugs e regressões no código.
- **Independentes:** a ordem de execução dos testes de unidade não é importante. Para quaisquer testes T1 e T2, a execução de T1 seguida de T2 deve ter o mesmo resultado da execução de T2 e depois T1.
- **Determinísticos (Repeatable):** testes de unidade devem ter sempre o mesmo resultado.
- **Auto-verificáveis (Self-checking):** O resultado de um teste de unidades deve ser facilmente verificável. Adicionalmente, quando um teste falha, deve ser possível identificar essa falha de forma rápida, incluindo a localização do comando **assert** que falhou.
- **Escritos o quanto antes (Timely),** se possível antes mesmo do código que vai ser testado.

Exemplos de teste

Exemplos

```
1 # content of test_sample.py
2 def funcao(x):
3     return x + 1
4
5 # Testes
6 def test_answer():
7     assert funcao(3) == 4
8
9 def test_answer_two():
10    assert funcao(3) != 6
```

```
1 import pytest
2
3 class MyClass:
4     def __init__(self, name, age):
5         self.name = name
6         self.age = age
7
8 @pytest.fixture
9 def my_class():
10     return MyClass(name="pavol", age=39)
11
12 def test_name(my_class):
13     assert my_class.name == "pavol"
14
15 def test_age(my_class):
16     assert my_class.age == 39
```

Cobertura de testes

Cobertura de Testes

Um sistema com alta cobertura de código significa que foi mais exaustivamente testado e tem uma menor chance de conter erros, ao contrário de um sistema com baixa cobertura de código.

A cobertura de código, sendo uma métrica quantitativa, visa medir quanto (%) do software é coberto/exercitado ao executar um determinado conjunto de casos de testes.

Cobertura de Testes – Métricas

-
- Branchs: percentual de branches de um programa que são executados por testes; um comando if sempre gera dois branches: quando a condição é verdadeira e quando ela é falsa). Cobertura de comandos e de branches são também chamadas de Cobertura C0 e Cobertura C1, respectivamente.
- Statement: verifica quantas instruções do código são executadas;
- Conditional:

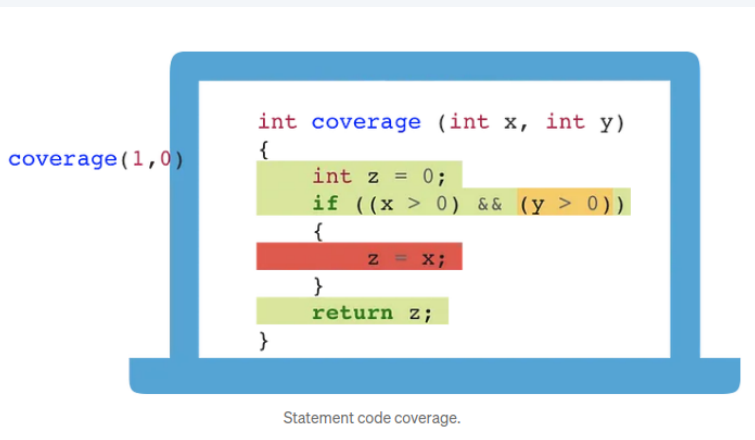
Cobertura de Testes: Funções

Funções: percentual de funções que são executadas por um teste

```
int coverage (int x, int y)
{
    int z = 0;
    if ((x > 0) && (y > 0))
    {
        z = x;
    }
    return z;
}
```

Cobertura de Testes: Statement

Verifica quantas instruções do código são executadas;



Cobertura de Testes: Branch

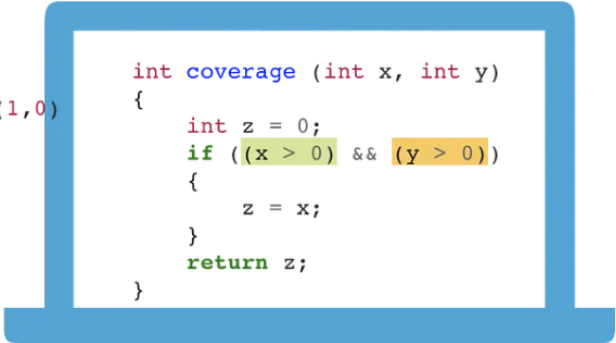
Branchs: percentual de branches de um programa que são executados por testes; um comando if sempre gera dois branches: quando a condição é verdadeira e quando ela é falsa). Cobertura de comandos e de branches são também chamadas de Cobertura C0 e Cobertura C1, respectivamente.

coverage(1,1)

```
int coverage (int x, int y)
{
    int z = 0;
    if ((x > 0) && (y > 0))
    {
        z = x;
    }
    return z;
}
```


Cobertura de Testes

Verifica se cada sub-expressão booleana são avaliadas ambas como verdadeiras e falsas;



The diagram shows a blue laptop frame containing a white screen with C code. To the left of the screen, the text `coverage(1,0)` is written in blue. The code on the screen is as follows:

```
int coverage (int x, int y)
{
    int z = 0;
    if ((x > 0) && (y > 0))
    {
        z = x;
    }
    return z;
}
```

In the code, the sub-expression `(x > 0)` is highlighted with a light green background, and the sub-expression `(y > 0)` is highlighted with a light orange background.

Material Auxiliar

<https://www.youtube.com/watch?v=4bubIRBCLVQ>

<https://medium.com/liferay-engineering-brazil/um-pouco-sobre-cobertura-de-c>

<https://youtu.be/bLdEypr2e-8?si=hkwo5cwAe66xFeDg>

Prática

Prática - Coding Dojo

O que é?

Um *Coding Dojo* é um encontro onde um grupo de programadores se reúne para treinar técnicas e metodologias de desenvolvimento de software através da solução de um pequeno desafio de programação.

Como funciona?

O objetivo é que todos os participantes sejam capazes de reproduzir a solução alcançando o mesmo resultado. Nesse formato há a **participação de todos**. É proposto um problema a ser resolvido e a programação é realizada em apenas uma máquina, **por pares**. Para esse formato é imprescindível a utilização de **TDD** e passos de bebê.

A pessoa que está codificando é o **piloto**, e seu par o co-piloto. A cada cinco minutos o piloto volta para a plateia e o co-piloto assume a condição de piloto. Uma pessoa da plateia passa a assumir a posição de co-piloto.

Prática – Coding Dojo

1. Formem grupos de até 8 integrantes;
2. Utilizem a linguagem de programação de sua preferência;
3. Seleccionem um problema do site: Dojo Puzzles
4. Escolher um integrantes para as funções: piloto, co-piloto, controle de tempo
5. A rodada terá duração de 2 minutos, cronometrados. Após cada rodada o co-piloto assume a posição de piloto e outro integrante a de co-piloto.
6. Enquanto o tempo estiver rodando somente piloto e co-piloto poderão interagir, os demais ficarão em silêncio.

Referências

Referências



Pressman, Roger (2021)

Engenharia de Software: Uma abordagem Profissional
AMGH Editora Ltda – 9. ed.



Sommerville, Ian (2011)

Engenharia de Software
Pearson Prentice Hall – 9. ed.



Marco Tulio Valente (2020)

Engenharia de Software Moderna: Princípios e Práticas para Desenvolvimento de Software com Produtividade
Editora: Independente



Obrigada

Prof. Eliane Faveron Maciel

UNIFACS
Ecosistema ânima

4 de abril de 2024