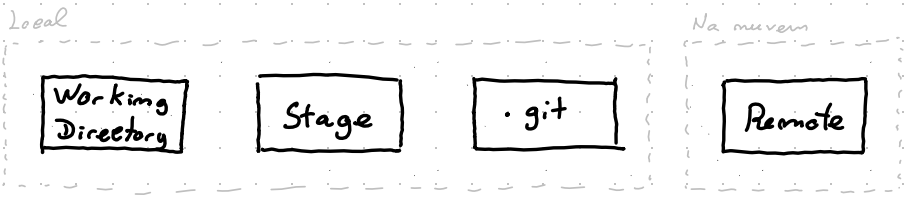
L.EIC
#2.1

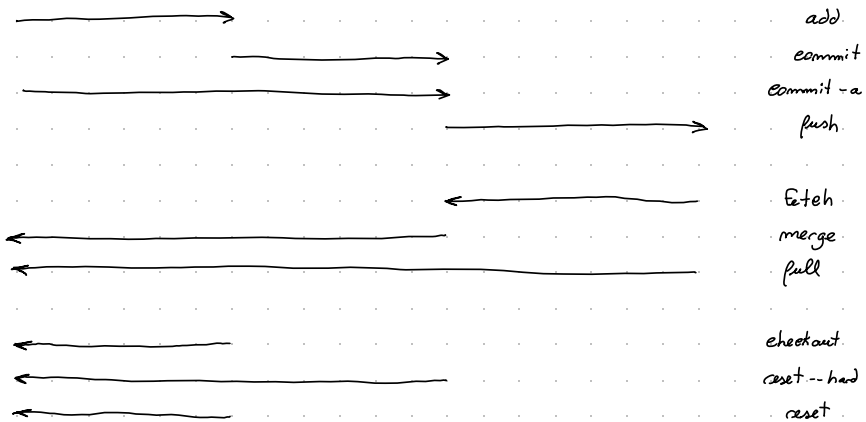
Laboratório de Desenho e Teste de Software

Resumos

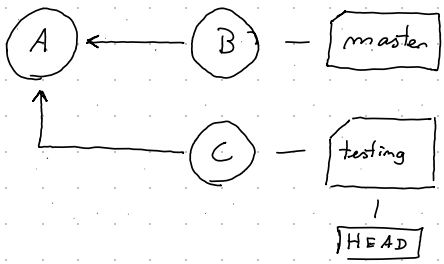
1. Git
   - Java
2. Testes Unitários
3. Design Patterns
4. UML

1. Git





Branches



- **branch** (nome)
  | Criar novo branch
- **branch -d** (branch)
  | Apagar o "pointer" do branch
- **checkout** (branch)
  | Mover "HEAD" e reverter ficheiros
- **merge** (branch)
  | Junta conteúdos entre "branches"

Novo repositório

- init | Criar repositório vazio
- clone | Copiar remoto já existente

Controlo

- status
- log
- diff
- (Ficheiro .gitignore)

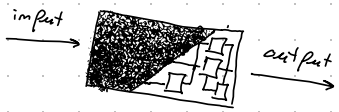
2. Testes Unitários

Estrutura (JUnit)

Arrange → **Act** → **Assert** → **(Limpar)**

- Arrange: `@BeforeAll` `@BeforeEach`
- Act: `@Test`
- Assert: (Assertions class)
- (Limpar): `@AfterAll` `@AfterEach`

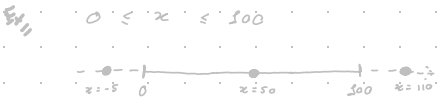
Estratégias de testes



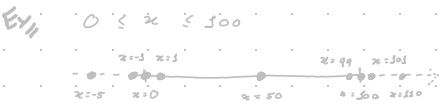
"Blackbox"

Baseado na especificação.

- **"Equivalence Partitioning"**
  Um teste por cada conjunto de possibilidades distinto e com o mesmo significado.



- **"Boundary value analysis"**
  Imelui testes dentro e fora da especificação, bem como na fronteira.



"White box"

Baseado na implementação.

- **"Statement coverage"**
  Executar uma vez por declaração.
  Assumir condições verdadeiras, p. e.
- **"Branch coverage"**
  Executar duas vezes por condição, um verdadeiro e outro falso.
- **"Path coverage"**
  Cobrir todos os caminhos independentes.
  Uma condição gera dois caminhos, p. e.
- **"Condition coverage"**
  Ter em consideração todas as variáveis booleanas.

Nomenclaturas

- **Erro humano** | Ação que gera falhas
- **"Fault"** | Defeito no software
- **"Error"** | Mudança inesperada no sistema
- **"Failure"** | Erro visível

- **Verificação** (mais Formal) (usa provas matemáticas)
  Projeto bem feito com base na especificação.
- **Validação** (mais empírico) (usa testes unitários)
  Projeto bem feito com base nas necessidades do cliente.

Test Doubles

Útil para isolar testes e reduzir dependências

- **Stub** (State Testing)
  Simula respostas pré-programadas no teste para evitar chamar a dependência
  Ex// Num serviço de email, substituir a função de enviar para apenas notificar que enviou, mas sem ter feito nada nos servidores.

<

- **Mock** (Behavioral Testing)
  Usado para verificar que as interações do código cumprem expetativas.
  Ex// Num serviço de email, conta quantas vezes a função de enviar foi chamada e com os parâmetros que se esperam

Mutation Testing

id 'info.solidsoft.pitest'

- Cria mutações no código e verifica se as mesmas passam aos testes
- Útil para avaliar a qualidade dos testes unitários.

Test-driven Development

- Fazer primeiro os testes e depois a implementação
- Útil, pois obriga a pensar na especificação antes de implementar, bem como dá mais liberdade para modificar o código com confiança.

Coverage

- Analisa as linhas de código que cada teste executa.
- Útil para encontrar testes em falta e funções por testar.

Property-based Testing

testImplementation 'net.jqwik:jqwik'

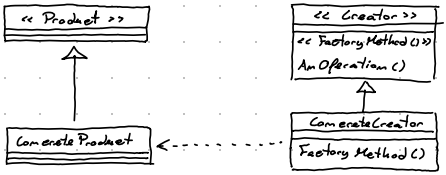
- Especificar restrições do resultado e gerar testes automaticamente
  → Cada teste representa uma propriedade da função Ex// sum(a, b) = sum(b, a)
  → Jqwik usa estatísticas para enviesar os testes nos pontos mais propícios a erro e permite reprodutibilidade e encolhimento dos testes para um debug mais simples.

- Útil para criar testes que normalmente não escreveríamos sozinhos.
  Ex// Questões de integer overflow

3. Padrões de design

Creational

- **Factory Method**
  Criar uma interface para criar um objeto, mas deixar as detalhes para as subclasses



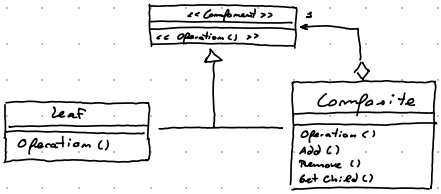
- **Abstract - Factory**
  Dar uma interface para criar uma família de objetos dependentes sem especificar classes concretas.

- **Singleton**
  O objeto só é instanciado uma única vez.

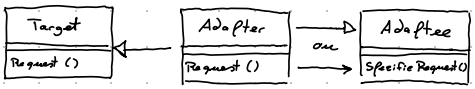
- Builder
- Prototype

Structural

- **Composite** (Relação parte - todo)
  Permite tratar composições e objetos individuais da mesma forma.



- **Adapter**
  Converter interfaces de forma a se conseguirem comunicar por protocolos diferentes.



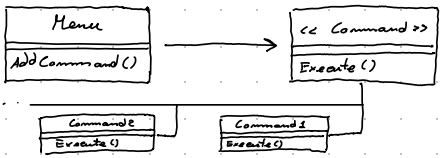
- **Decorator**
  Adicionar mais funcionalidades a objetos e funções.

Ex:
@Test
@Property

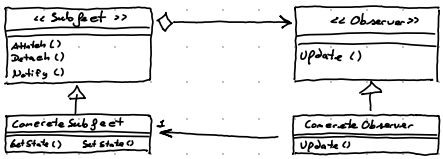
- Bridge
- Facade
- Flyweight
- Proxy

Behavioral

- **Command**
  Encapsular pedidos. Suporta registros, "undo", ...

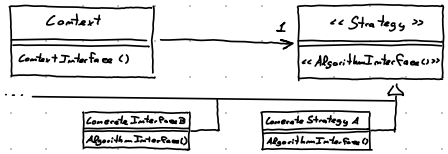


- **Observer**
  Quando um objeto muda, todos os dependentes são notificados.

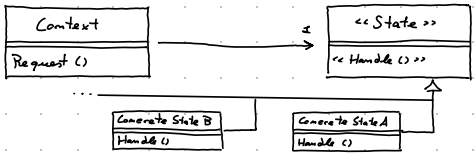


LDTS

- **Strategy**
  Uma família de algoritmos que podem ser trocados em tempo real para o mesmo contexto.



- **State**
  Permite alterar o comportamento de um objeto em tempo real, ao mudar o estado interno.

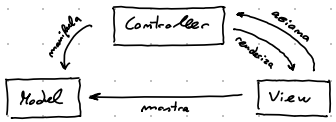


- Chain of Responsibility
- Interpreter
- Iterator
- Mediator
- Memento
- Template Method
- Visitor

Padrões de Arquitetura

- Pipe - Filter
- Broker

- **Model - View - Controller**
  Útil em GUIs.
  → Model | Representa os dados
  → View | Mostra os dados para o ecrã
  → Controller | Interpreta as ações do utilizador



SOLID

S ingle Responsibility Principle | Cada módulo só tem uma razão para mudar
O pen - Closed Principle | Módulo aberto a extensão, mas fechado a modificação
L iskov Substitution Principle | Sub classes devem ser substituíveis pelas classes base.
I nterface Segregation Principle | Várias interfaces específicas > Uma interface geral
D ependency Inversion Principle | Detalhes dependem da abstração, e não o contrário

Package Architecture

- **Release Reuse Equivalency**
  Não copiar e colar código. Reutilizar.

- **Common Closure**
  Se uma alteração envolve 2 objetos, deveriam estar juntos.

- **Common Reuse**
  Todas as dependências da classe no mesmo pacote.

Package Coupling

- **Acyclic Dependencies**
  Dependências entre pacotes não podem formar ciclos.

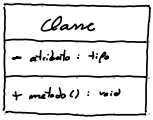
- **Stable Dependencies**
  Dependências devem ser difíceis de mudar.

- **Stable Abstractions**
  - Pacotes abstratos para tirar dependências.

4. UML

Diagrama de Classes

- Classe

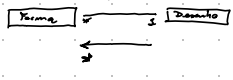


- Herança

- Classe / Método abstrato

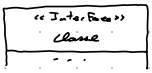
(Escrever em itálico, ou <<a>>)

- Associações

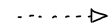


- Classes de associação

- Interface



- Implementa



- Agregação

- Composição

- Dependência

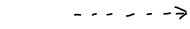
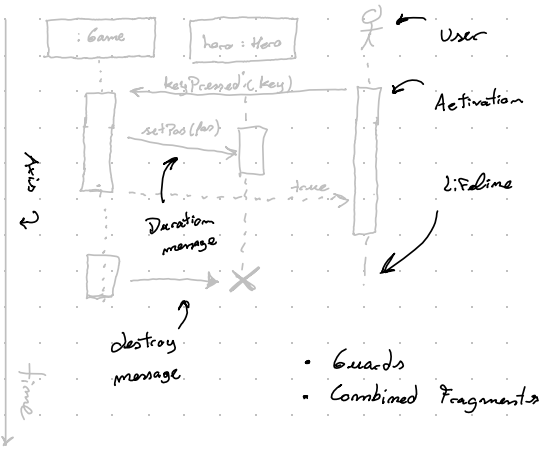


Diagrama de sequência



- Guards
- Combined Fragments

Mensagens:

- Síncrona
- Assíncrona

Return Value

Self Message

Create Message

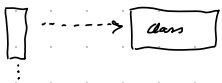
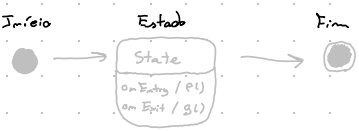


Diagrama de estados



- Transições

Trigger [Guard] / Effect

- Composite State