Resumos LPOO

MIEIC

10 de fevereiro de 2021

Conteúdo

1 Introdução **2**

2 Git **2**

3 Java **5**

4 Unit Testing **7**

5 SOLID and other OO principles **8**

6 UML - Class Diagrams **9**

7 Design Patterns **11**
7.1 Factory Method 11
7.2 Composite 11
7.3 Command 12
7.4 Observer 12
7.5 Strategy 12
7.6 State 13
7.7 Adapter 13
7.8 Decorator 13
7.9 Singleton 14
7.10 Abstract-Factory 14

8 Refactoring **14**

9 Java Generics **15**

10 Swing **16**

11 UML - Sequence Diagrams **17**

12 UML - Communication Diagrams **18**

1 Introdução

Este documento contém resumos dos conteúdos lecionados em LPOO no ano letivo 2018/2019. Este documento contém apenas resumos dos conteúdos abordados na unidade curricular não substitui o estudo mais aprofundado destes conteúdos.

2 Git

Guarda as diferentes versões de um ficheiro como snapshots. Se não forem alterados, guarda um link para o snapshot anterior.

Tudo no git é guardado como um SHA-1.

Cada versão (commit) tem um snapshot da versão dos ficheiros (ou links caso não haja alterações). Todos os objetos (ficheiros, commits...) têm uma hash identificadora.

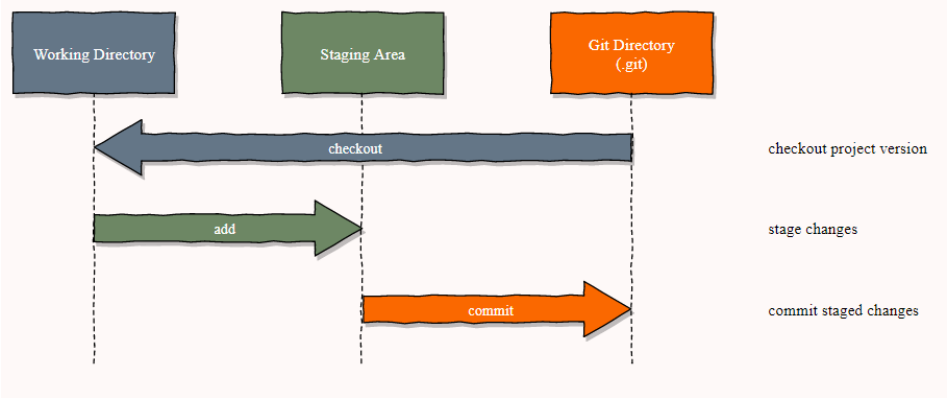


Figura 1: Git Areas

git init : cria um subdiretório .git/ escondido

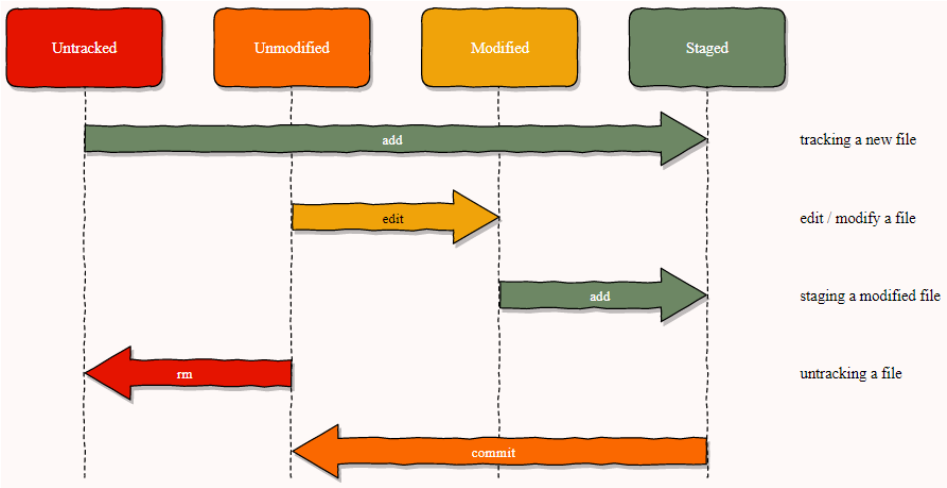


Figura 2: Estados dos Ficheiros

git add ...

- Dá track e stage a um ficheiro untracked pelo Git
- Dá stage a um ficheiro que foi modificado

git commit: grava um novo snapshot do repositório

git status: mostra o estado dos ficheiros

git log: mostra o histórico de commits do repositório

Um branch é um apontador para um commit. O branch inicial é o master. HEAD é um apontador especial que aponta sempre para o branch atual.

git branch test: cria um branch chamado *test*

git branch: mostra os branches atuais

git checkout test: alternar para o branch *test*

git merge test: junta ao branch atual as modificações feitas no branch *test*

Existem duas estratégias de merge:

- Fast-forward merge
  - Quando não existe trabalho divergente
  - Move-se o apontador para a frente
- Three-way merge
  - Quando existe trabalho divergente
  - Utiliza-se os snapshots dos dois branches e o antepassado comum e cria-se um novo commit

git branch −d test: apaga o branch *test*

Ficheiros *.gitignore* especificam ficheiros untracked que o git deve ignorar.

git clone url: clonar/criar repositório remoto (adiciona o origin)

- git remote −v: lista todos os repositórios remotos
- git remove add nome url: adicionar outro remoto

git fetch: download dos ficheiros do remoto

git pull: git fetch + git merge

git push: upload das modificações realizadas

git reset: reverter modificações

- git revert <commit_id>: caso já tenha sido realizado um push

COMMIT^: commit anterior a COMMIT

COMMIT~2: refere-se a 2 commits atrás de COMMIT, ou seja, ao commit anterior a COMMIT^

3 Java

Não é uma linguagem OOP pura, porque as variáveis podem ter valores primitivos ou ser referenciadas como objetos.

Não há apontadores, mas variáveis primitivas são guardadas como valores e objetos são guardados como referências.

Literals: são representações sintáticas de variáveis (Boolean, Character, String, Integer, ...).

Operador == compara tipos primitivos pelo seu valor, mas compara objetos pela sua referência.

Input/Output:

- System.out.println (...);
- Scanner scanner = new Scanner(System.in);
  String line = scanner.nextLine();

Strings de Java são imutáveis. Para as comparar usa-se o método equals().

OOP: providencia uma abstração onde os elementos do problema são objetos no espaço solução; permite descrever o problema em termos do problema.

Pilares da Orientação por Objetos (A PIE)

- Data **A**bstraction: separação entre interface pública de um tipo de dados e a sua implementação
- **P**olymorphism: um único símbolo pode representar muitos tipos
- **I**nheritance: objetos podem herdar propriedades e comportamentos de outros objetos
- **E**ncapsulation: acesso restrito a algumas componentes de um objeto

Visibilidade:

- Classes:
  - public
  - protected (mesmo package)
  - private (dentro de outras classes)
- Variáveis e Métodos:
  - public
  - protected
  - private
  - package

Para criar um novo objeto é necessário usar new:

```
Light light = new Light();
Light another = light;
```

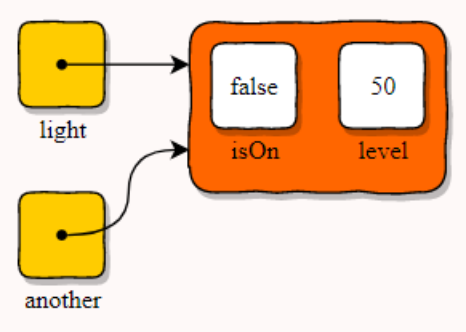


Figura 3: Criação de novo objeto

Para termos duas instâncias do mesmo objeto, a classe tem que implementar a interface Cloneable e o seu método clone();

final: variável que não pode ser alterada.

Herança deve ser usada para estabelecer uma relação de is-a (é um/a). Só é possível estender uma classe.

Métodos final, static e private não podem ser reescritos.

Deve-se reescrever o método equals() para todas as classes que vão ser comparadas, o método hashCode() quando usamos HashSet e o método toString().

O bloco finally executa sempre que se sai de um bloco try. Assegura que este código é sempre corrido mesmo que ocorra uma exceção.

```
try {
    ...
} catch {
    ...
} finally { ... }
```

Coleções: Set, List, Queue, Deque, Map

- São parameterizadas
- Exemplo: List<Animal> list = new ArrayList<>();
  - Princípio "Retorna o tipo mais específico, aceito o tipo mais genérico"

4 Unit Testing

Testes black-box: estrutura interna é desconhecida ou não considerada.

Testes white-box: o design é baseado na estrutura interna para que o número máximo de "caminhos"do código sejam testados.

Unit Testing:

- testar unidades individuais de software
- código necessita de ser modular, o que o faz ser reutilizável

Princípios FIRST:

- **F**ast: devem ser rápidos.
- **I**solated/**I**ndependent: só se testa uma unidade de cada vez. Ordem não interessa.
- **R**epeatable: não devem depender do ambiente (time, random values, ...).
- **S**elf-validating: não é necessário verificar à mão.
- **T**horough/**T**imely: devem cobrir todos os casos de uso.

Três A's nos quais os testes devem ser divididos:

- **A**rrange: inicialização.
- **A**ct: método a testar é invocado.
- **A**ssert: é usado um assert para testar o resultado.

Stubs: providenciam respostas para as chamadas realizadas.

Mocks: pré-programados com expectativas que geram uma especificação das chamadas que se espera receber.

State Testing: testar o estado após invocação do método.

Behaviour Testing: testar o comportamento do método.

5 SOLID and other OO principles

SOLID:

- **S** - The **S**ingle Responsibility Principle (SRP)
  - Cada módulo do software só deve ter uma e uma só razão para mudar.
- **O** - The **O**pen-Closed Principle (OCP)
  - Um módulo deve ser aberto para extensão, mas fechado para modificação
- **L** - The **L**iskov Substitution Principle (LSP)
  - Subclasses devem ser subtituíveis pelas suas classes-base
- **I** - The **I**nterface Segregation Principle (ISP)
  - Muitas interfaces específicas de clientes são melhores que uma interface de propósito geral
- **D** - The **D**ependency Inversion Principle (DIP)
  - Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações.
  - Abstrações não devem depender de detalhes. Detalhes devem depender das abstrações.

Princípios da Arquitetura em Pacotes (packages):

- The Release Reuse Equivalency Principle
  - Código não deve ser reutilizável através da cópia de uma classe para outra
  - Só componentes lançadas através de um sistema de tracking podem ser reutilizáveis
- The Common Closure Principle (CCP)
  - Classes que mudam juntas, devem estar juntas
- The Common Reused Principle (CRP)
  - Classes que não são reutilizáveis em conjunto não devem estar no mesmo grupo (package)

Princípios do Agrupamento de Pacotes (packages):

- The Acyclic Dependencies Principle (ADP)
  - As dependências entre pacotes não podem formar ciclos
- The Stable Dependencies Principle (SDP)
  - Depender da direção da estabilidade. Precisa-se de packages "fáceis de mudar"(instáveis) para alterar facilmente o software
- The Stable Abstractions Principle (SAP)
  - Pacotes estáveis devem ser abstratos

6 UML - Class Diagrams

Diagrama de classes: diagrama estrutural.

Mostram as classes do sistema, as suas operações, atributos e relações.

Atributos (secção do meio): nome:tipo_do_atributo=valor_por_defeito (valor por defeito é opcional)

Operações (secção de baixo): nome(list_par[U+FFFD]metros):tipo_retornado

- Cada parâmetro da lista: nome:tipo
- Os parâmetros podem ter uma marca "in"ou "out"indicando se é um input ou output

Herança: aponta da subclasse para a superclass

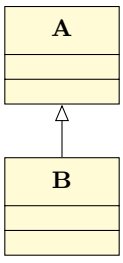


Figura 4: Exemplo Herança

Classes e métodos abstratos são representados em itálico.

Associações bidirecionais: cada lado tem uma multiplicidade.

Figura 5: Exemplo Associação Bidirecional

Associações unidirecionais: só uma classe sabe que a relação existe - aquela de onde parte a seta.

Figura 6: Exemplo Associação Unidirecional

Classe de associação: inclui informação sobre a relação (ligada à associação com —)

Interfaces: classe com «interface» antes do nome.
Implementação de uma interface:

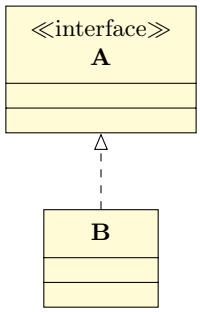


Figura 7: Exemplo Implementação Interface

Agregação: modela um todo para as partes.

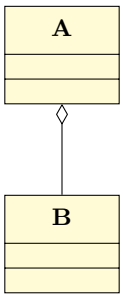


Figura 8: Exemplo Agregação

Composição: forma mais forte de agregação. O todo só existe com as suas partes.

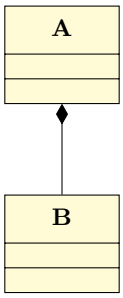


Figura 9: Exemplo Composição

7 Design Patterns

7.1 Factory Method

Define uma interface para criar um objeto, mas deixa as subclasses decidir qual classe será instanciada.

Usa-se quando:

- uma classe não consegue antecipar a classe dos objetos a serem criados.
- uma classe pretende que as subclasses especifiquem os objetos a criar.

Elimina a necessidade de ligar classes específicas ao código.

7.2 Composite

Compôr objetos em estruturas de árvores para representar hierarquias parte-todo. Permite que os clientes tratem objetos e composições uniformemente.

Usa-se quando:

- se pretende representar hierarquias parte-todo de objetos.
- se pretende que os clientes ignorem a diferença entre composições e objetos individuais.

Tipos primitivos podem ser compostos em objetos complexos.

7.3 Command

Encapsular um pedido como um objeto permitindo parametrizar clientes com diferentes pedidos, filas ou registo de pedidos e suportar operações reversíveis.

Usa-se quando:

- se especifica, enfileira e executa pedidos em momentos diferentes.
- se suporta operações de undo/redo.
- se estrutura um sistema envolvido em operações de alto nível baseadas em operações primitivas.

Comandos podem ser estendidos e manipulados como qualquer outro objeto.
É fácil adicionar comandos.
Tipos primitivos podem ser compostos em objetos complexos.

7.4 Observer

Define uma dependência um-para-muitos entre objetos para que quando um objeto muda o seu estado todos os dependentes são notificados e atualizados.

Usa-se quando:

- uma abstração tem dois aspetos, um dependente do outro.
- a mudança de um objeto implica a mudança de outro.
- um objeto deve notificar outros sem assumir quem são.

Emparelhamento entre sujeito e observador.
Suporte para comunicação.
Atualizações inesperadas.

7.5 Strategy

Define-se uma família de algoritmos, encapsula-se cada um e fazem-se permutáveis. Permite que o algoritmo varie dependendo do cliente que o usa.

Usa-se quando:

- muitas classes relacionadas só diferem no seu comportamento.
- se precisa de variantes diferentes de um algoritmo.
- um algoritmo usa dados que um cliente não deve conhecer.

- uma classe define muitos comportamentos que aparecem em múltiplas declarações condicionais.

Alternativa a ter subclasses.
Elimina condições e providencia implementações diferentes.

7.6 State

Permite que um objeto mude o seu comportamento quando os seus estados internos mudam. O objeto aparecerá para mudar a sua classe.

Usa-se quando:

- o comportamento de um objeto depende do seu estado e precisa de o mudar em run-time.
- as operações têm grandes condições que dependem de uma ou mais enumerações.

Faz com que as transições entre estados sejam explícitas.

7.7 Adapter

Converte a interface de uma classe noutra interface esperada pelo cliente. Permite que classes trabalhem juntas e não o poderiam fazer noutro caso devido à incompatibilidade de interfaces.

Usa-se quando:

- se quer utilizar uma classe existente e a sua interface não combina com a que se precisa.
- se quer criar uma classe reutilizável que trabalhe com classes imprevistas.

7.8 Decorator

Anexar responsabilidades adicionais a um objeto dinamicamente. Providencia uma alternativa flexível a criar subclasses por extensão.

Usa-se quando:

- se quer adicionar responsabilidades a objetos individuais dinamicamente.
- se quer retirar responsabilidades.
- estender por criação de subclasses é impraticável.

Mais flexível que herança estática. Evita classes com muitas funções/responsabilidades.

7.9 Singleton

Assegurar que uma classe só tem uma instância e providenciar uma forma global para a aceder.

Usa-se quando:

- tem que haver uma e uma só instância da classe.
- a única instância tem que ser estensível para subclasse.

É um considerado um **anti-pattern**:

- difícil de testar.
- difícil de implementar com multi-threading.

O que fazer em vez de usar o padrão? Criar uma instância da classe e propagá-la para o sítio onde será utilizada.

7.10 Abstract-Factory

Providenciar uma interface para a criação de famílias de objetos relacionados ou dependentes sem especificar a classe concreta.

Usa-se quando:

- um sistema deve ser independente de como os seus produtos são criados, compostos e representados.
- um sistema deve ser configurável com mais do que uma família de produtos.

Isola classes concretas.
Promove a consistência entre os produtos.
É difícil suportar novos tipos de produtos.
subsectionMode-View-Controller (MVC)
Divide o sistema em 3 partes:

- o modela representa os dados.
- a vista mostra o modelo e envia ações do utilizador ao controlador.
- o controlador providencia o modelo à vista e interpreta as ações do utilizador.

8 Refactoring

Um code smell nem sempre indica um problema, não é um problema.

Alguns code smells:

- **Long Method**: método com muitas linhas.

- **Large Class**: classe contém muitos atributos/métodos/linhas de código.
- **Long Parameter List**: mais de 3 ou 4 parâmetros por método.
- **Data Clumps**: partes diferentes do código contêm o mesmo grupo de variáveis.
- **Switch Statements**: operações if/switch complexas.
- **Refused Biquest**: subclasse só usa alguns métodos/atributos herdados da superclasse.
- **Alternative Classes with Different Interfaces**: duas classes têm funções idênticas, mas nomes de métodos diferentes.
- **Divergent Change**: mudar muitos métodos não relacionados quando se faz alterações numa classe.
- **Lazy Class**: classes que não fazem muita coisa.
- **Data Class**: classe que apenas contém atributos e métodos para lhes aceder.
- **Feature Envy**: método que acessa mais aos dados de outro objeto que aos seus dados.

Técnicas de Refactoring:

- Extract Method / Class
- Inline Method / Class
- Extract Variable
- Split Temporary Variable
- Decompose Conditional
- Consolidate Duplicate Conditional Fragments
- Replace Nested Conditional with Guard Clauses
- Introduce Null Object

9 Java Generics

Genéricos providenciam Type-safety.

Type Variable: identificador não qualificado (ex: <T>).

- Nomes:
  - **E** - Elemento
  - **K** - Chave (Key)
  - **N** - Número
  - **T** - Tipo (S, U, V.. -> 2º, 3º e 4º tipos)
  - **V** - Valor

Métodos genéricos: public <T> void arrayToList(T[] a, List<T> l)

Classes genéricas: pubic class Box<T>
É possível criar classes que estendem tipos genéricos.

Variância:

- **Covariância**: preserva a ordem dos tipos (aceita subtipos) -> tipos de Java.
- **Contravariância**: troca a ordem dos tipos (aceita supertipos).
- **Invariância**: só aceita o tipo específico -> Java Generics.

Wildcards:

- em vez de <T> é possível utilizar <?> (tipo desconhecido).
- possível utilizar <? extends Animal> (variante) ou <? super Animal> (contravariante).

10 Swing

Componentes:

- JButton, JCheckBox, JRadioButton
- JLabel, JTextField, JPasswordField, JTextArea (múltiplas linhas)
- JComboBox (drop-down list + textbox)
- JList

Layouts:

- BoxLayout: componentes numa linha ou coluna
- BorderLayout: componentes em 5 posições (cima, baixo, esquerda, direita, centro)
- GridBagLayout: componentes dispostos numa grelha com número de linhas e colunas diferentes
- Panels: contentor genérico

Key Listener: notifica caso haja uma tecla premida.

11 UML - Sequence Diagrams

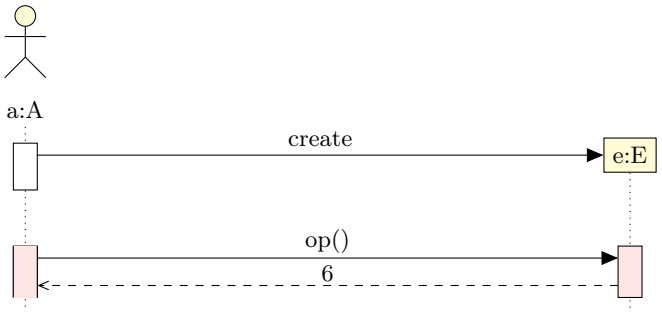
Diagrama de sequência: diagrama comportamental.

Mostra a interação entre os objetos numa ordem sequencial.

Lifeline: elemento com nome que representa um indivíduo na interação.

Linha vertical representa o tempo (de cima para baixo).

Actor: sistema ou pessoa que está fora do escopo do sistema.



Mensagens: seta a partir do objeto que envia para o que recebe. Nome do método por cima da seta. Retorno é opcional e representa-se com uma seta a tracejado.

12 UML - Communication Diagrams

Diagrama de Comunicação: diagrama comportamental.

Versão simplificada dos diagramas de sequência.

Ordem da sequência: inteiro representa a ordem sequencial da mensagem; letras correspondem a chamadas concorrentes.

Loops: asterisco seguido da condição dentro de parêntesis retos (exemplo: *[elements]).

13 Inheritance vs. Composition

Interfaces de Java permitem a criação de variáveis public, static, final.

Interfaces também podem ter implementações de métodos default, mas não são uma boa ideia porque os atributos têm que ser public, static e final.

Composição em vez de herança é o princípio de que as classes devem alcançar um comportamento polimórfico e a sua composição deve reutilizar código em vez de herança.