



Programação Orientada a Objetos



Prof. Isaac



Convenções de Código e Código Legível

Boas Práticas de Programação

Por que convenções são importantes?

- 80% do custo para manter um software durante sua vida vai para manutenção.
- Normalmente, a manutenção de softwares não é feita pelo autor original.
- Convenções são importantes pois melhoram a legibilidade do software.
- Se você for usar seu código-fonte como um produto, você precisa ter certeza de que ele é tão bem embalado e limpo como qualquer outro produto.

Convenções / boas práticas
de programação servem
para todas as linguagens!

Convenções do Java

- **Classes, Arquivos e Projetos:** nomes de devem ser escritos com primeira letra MAIÚSCULA.
 - Exemplo: Principal, Entrada, Banco.
- **variáveis, métodos e pacotes:** nomes devem ser escritos com primeira letra minúscula.
 - Exemplos: numero, somar(), formularios.

Convenções do Java

- Quando os nomes necessitarem mais de uma palavra, para facilitar o entendimento, o início das palavras seguintes devem ser escritos com letra maiúscula.
 - Exemplos: ContaConjunta, segundoNome, encontrarMaiorValor(), ProjAula1.
- Escolha nomes que implicitamente já sugiram sua utilização (nomes significativos). Use nomes completos em vez de abreviações confusas.

Indentação

- **Indentação** (recuo) é um neologismo derivado da palavra em inglês *indentation*.
- Indentação é um termo aplicado ao código fonte de um programa para ressaltar ou definir a estrutura do algoritmo.
- O código **com indentação é mais legível**.

Código em C com indentação:

```
if (unlikely(prev->policy == SCHED_RR))
    if (!prev->counter)
    {
        prev->counter = NICE_TO_TICKS(prev->nice);
        move_last_runqueue(prev);
    }
switch (prev->state)
{
    case TASK_INTERRUPTIBLE:
        if (signal_pending(prev))
        {
            prev->state = TASK_RUNNING;
            break;
        }
    default:
        del_from_runqueue(prev);
}
prev->need_resched = 0;
```

O mesmo código, sem indentação:

```
if (unlikely(prev->policy == SCHED_RR))
if (!prev->counter)
{
    prev->counter = NICE_TO_TICKS(prev->nice);
    move_last_runqueue(prev);
}
switch (prev->state)
{
    case TASK_INTERRUPTIBLE:
        if (signal_pending(prev))
        {
            prev->state = TASK_RUNNING;
            break;
        }
    default:
        del_from_runqueue(prev);
}
prev->need_resched = 0;
```


Indentação

- 4 espaços devem ser utilizados
- Evite linhas com mais de 80 caracteres
- Quando uma expressão não couber em uma única linha, quebre utilizando os seguintes princípios:
 - Quebre depois de uma vírgula
 - Quebre antes de um operador
 - Alinhe a nova linha com o começo da expressão de mesmo nível da linha anterior.

```
function(longExpression1, longExpression2, longExpression3,  
        longExpression4, longExpression5);  
  
var = function1(longExpression1,  
                function2(longExpression2,  
                          longExpression3));
```

Documentação



Documentação

- É um texto escrito que acompanha o **software** e geralmente explica como utilizá-lo.
- Os textos podem também trazer diagramas que explicam o funcionamento e o relacionamento existente entre as funções, as classes ou partes específicas do código.

Documentação

- Comentários Iniciais:

- Todo código-fonte deve ser iniciado com um comentário que liste: o nome do programador, a data, a licença, uma breve descrição do propósito do programa e mais alguma informação que o programador achar relevante, ex: versão, projeto etc.
- No Java o comentário inicial segue o mesmo estilo adotado no c, ex:

```
/*  
 * Classname  
 *  
 * Version info  
 *  
 * Copyright notice  
 */
```

O que documentar

- **Pacote e Classe:** colocar um comentário inicial com informações sobre conteúdo, autor e data de codificação antes do nome da classe
- **Método:** escrever antes do método um comentário descrevendo o objetivo do método, seus parâmetros de entrada e o seu retorno (um exemplo de uso também pode ajudar)
- **Estruturas e Fórmulas:** caso não sejam óbvias, explique o que fazem e até porque está sendo feito

Documentação

- Comentários gerais:
 - Servem para explicar os programas para outros programadores (ou para você mesmo)
 - São ignorados pelo compilador / interpretador
 - No Java segue o seguinte padrão:
 - Comentário em Bloco - mesmo padrão do comentário inicial
 - Comentário de Linha Única: `/* Comment */`
 - Comentário de fim de linha: `// Comment`

javadoc



- Cria a documentação dos códigos Java em HTML:
 - No terminal:
 - *javadoc *.java*
 - Nas IDEs:
 - O método de chamar o **javadoc** varia, mas sempre é possível e, normalmente, simples

Exemplo de código com o padrão de comentário esperado

Considera as relações entre as classes e os comentários, desde que estejam dentro do padrão esperado pelo javadoc:

```
1 /**..  
2 * Passaro.java - a simple class for demonstrating the use of javadoc comments..  
3 * @author Danilo Perico  
4 * @version 1.0  
5 * @see Animal  
6 */  
7  
8 public class Passaro extends Animal {  
9     public void move() {  
10         System.out.println("Voa");  
11     }  
12 }
```


Algumas tags muito úteis

@author	This tag lets you put the name of the code author into the documentation.
@parameter	This tag is used to define parameters that are passed into a method.
@return	This tag defines values that are returned from methods.
@see	This tag creates "See Also:" output. You'll normally use this tag to refer to related classes.
@version	This tag lets you define the version of the Java code you're developing. For instance, the code displayed in Listing 1 is considered to version 1.0.

Alguns parâmetros

- `javadoc *.java -public`
 - Show only public classes and members
- `-protected`
 - Show protected/public classes and members (default)
- `-package`
 - Show package/protected/public classes and members
- `-private`
 - Show all classes and members
- `-d <directory>`
 - Destination directory for output files
- `-version`
 - Include @version paragraphs
- `-author`
 - Include @author paragraphs

Exemplo de Documentação

All Classes

Animal
MainAnimalPoli
Passaro
Peixe
Sapo

PACKAGE CLASS TREE DEPRECATED INDEX HELP

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHODDETAIL: FIELD | CONSTR | METHOD

Class Animal

java.lang.Object
Animal

Direct Known Subclasses:
Passaro, Peixe, Sapo

```
public class Animal  
extends java.lang.Object
```

Constructor Summary

Constructors

Constructor and Description

Animal()

Method Summary

All MethodsInstance MethodsConcrete Methods

Modifier and Type	Method and Description
void	move()

Exemplo usando algumas tags

```
/**
 * Soma.java - a simple class for demostranting the use of javadoc comments.
 * @author ISAACJESUSDASILVA
 * @version 1.0
 */
public class Soma extends OperacaoMatematica{

    /**
     * Método que realiza o cálculo da soma
     * @param x operando utilizado na soma
     * @param y operando utilizado na soma
     * @return Retorna a soma de x + y
     */
    @Override
    public double calcular(double x, double y) {
        return x + y;
    }
}
```

Exemplo usando algumas tags

All Classes

OperacaoMatematica

Principal

Soma

Subtracao

PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

exemplo02polimorfismo

Class Soma

java.lang.Object
 exemplo02polimorfismo.OperacaoMatematica
 exemplo02polimorfismo.Soma

public class Soma
extends OperacaoMatematica

Soma.java - a simple class for demostranting the use of javadoc comments.

Constructor Summary

Constructors

Constructor and Description

Soma()

Method Summary

Exemplo usando algumas tags

The screenshot shows a Java IDE interface. On the left, a sidebar titled 'All Classes' lists the following items: 'OperacaoMatematica', 'Principal', 'Soma' (highlighted in orange), and 'Subtracao'. The main area displays the 'Method Detail' for the 'calcular' method. The method signature is 'public double calcular(double x, double y)'. Below the signature, there is a description in Portuguese: 'Método que realiza o cálculo da soma'. This is followed by 'Specified by: calcular in class OperacaoMatematica'. Then, under 'Parameters:', it lists 'x - operando utilizado na soma' and 'y - operando utilizado na soma'. Finally, under 'Returns:', it states 'Retorna a soma de x + y'.

All Classes

- OperacaoMatematica
- Principal
- Soma**
- Subtracao

Method Detail

calcular

```
public double calcular(double x,  
                        double y)
```

Método que realiza o cálculo da soma

Specified by:
calcular in class OperacaoMatematica

Parameters:

- x - operando utilizado na soma
- y - operando utilizado na soma

Returns:

Retorna a soma de x + y

Doxygen



“Doxygen is a standard tool for generating documentation from annotated C++ sources, but it also supports other popular programming languages such as C, Objective-C, C#, PHP, Java, Python, IDL (Corba, Microsoft, and UNO/OpenOffice flavors), Fortran, VHDL, Tcl, and to some extent D.”

A documentação pode ser gerada em HTML e PDF.

Usando Doxygen pelo terminal

- Primeiro, é necessário gerar um arquivo de configuração:

```
doxygen -g my_proj.conf
```

- Depois, o arquivo de configuração é executado:

```
doxygen my_proj.conf
```

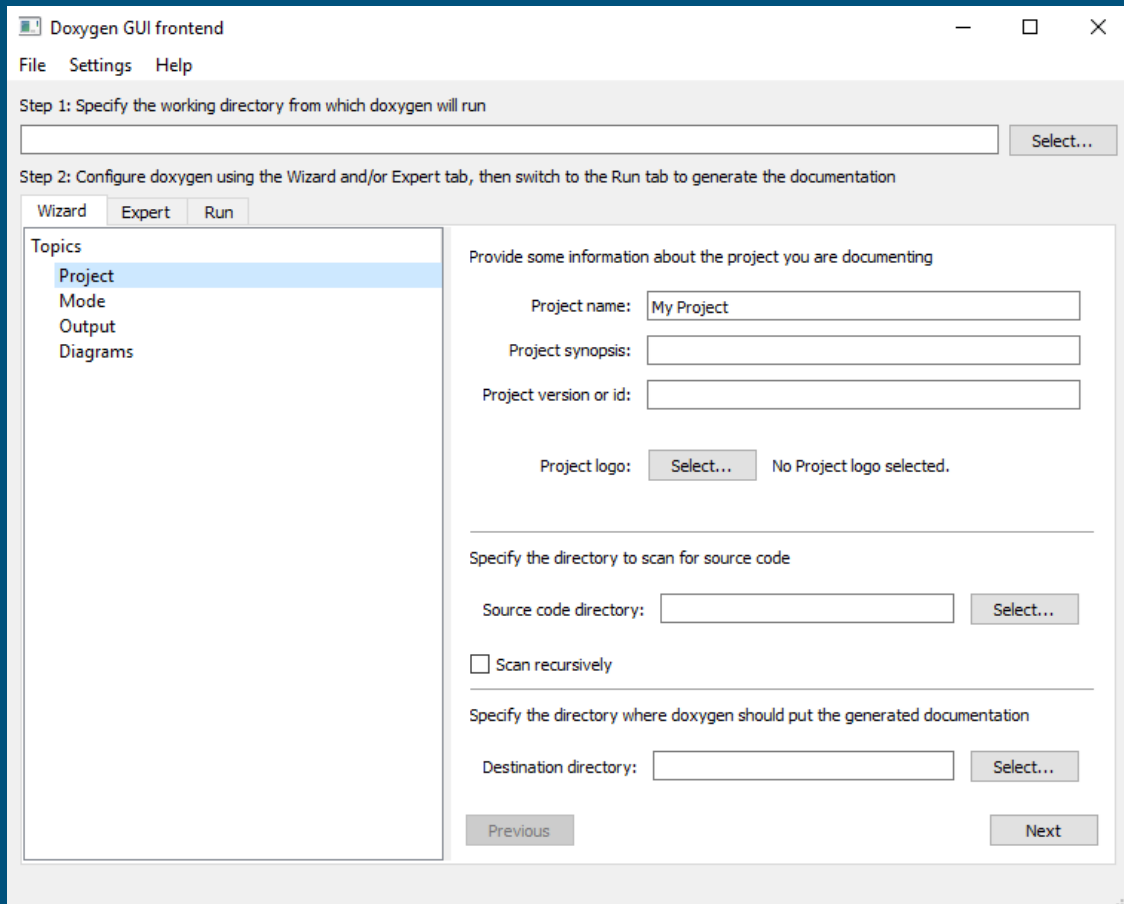

Comentários

```
/**  
 * ... text ...  
 */
```

```
///  
/// ... text ...  
///
```

```
int var; ///  
         ///  
         Detailed description after the member  
         ///  
         Detailed description after the member
```

Usando o Doxygen no Windows



The screenshot shows the 'Doxygen GUI frontend' window. It has a menu bar with 'File', 'Settings', and 'Help'. Below the menu bar, there are two steps: 'Step 1: Specify the working directory from which doxygen will run' and 'Step 2: Configure doxygen using the Wizard and/or Expert tab, then switch to the Run tab to generate the documentation'. Step 1 includes a text input field and a 'Select...' button. Step 2 includes three tabs: 'Wizard', 'Expert', and 'Run'. The 'Wizard' tab is active, showing a 'Topics' list on the left with 'Project', 'Mode', 'Output', and 'Diagrams'. The 'Project' topic is selected. The main area of the Wizard tab contains several form fields: 'Project name' (filled with 'My Project'), 'Project synopsis', 'Project version or id', 'Project logo' (with a 'Select...' button and the text 'No Project logo selected.'), 'Source code directory' (with a 'Select...' button), 'Scan recursively' (unchecked checkbox), 'Destination directory' (with a 'Select...' button'), 'Previous' button, and 'Next' button.

Doxygen GUI frontend

File Settings Help

Step 1: Specify the working directory from which doxygen will run

Select...

Step 2: Configure doxygen using the Wizard and/or Expert tab, then switch to the Run tab to generate the documentation

Wizard Expert Run

Topics

- Project
- Mode
- Output
- Diagrams

Provide some information about the project you are documenting

Project name: My Project

Project synopsis:

Project version or id:

Project logo: Select... No Project logo selected.

Specify the directory to scan for source code

Source code directory: Select...

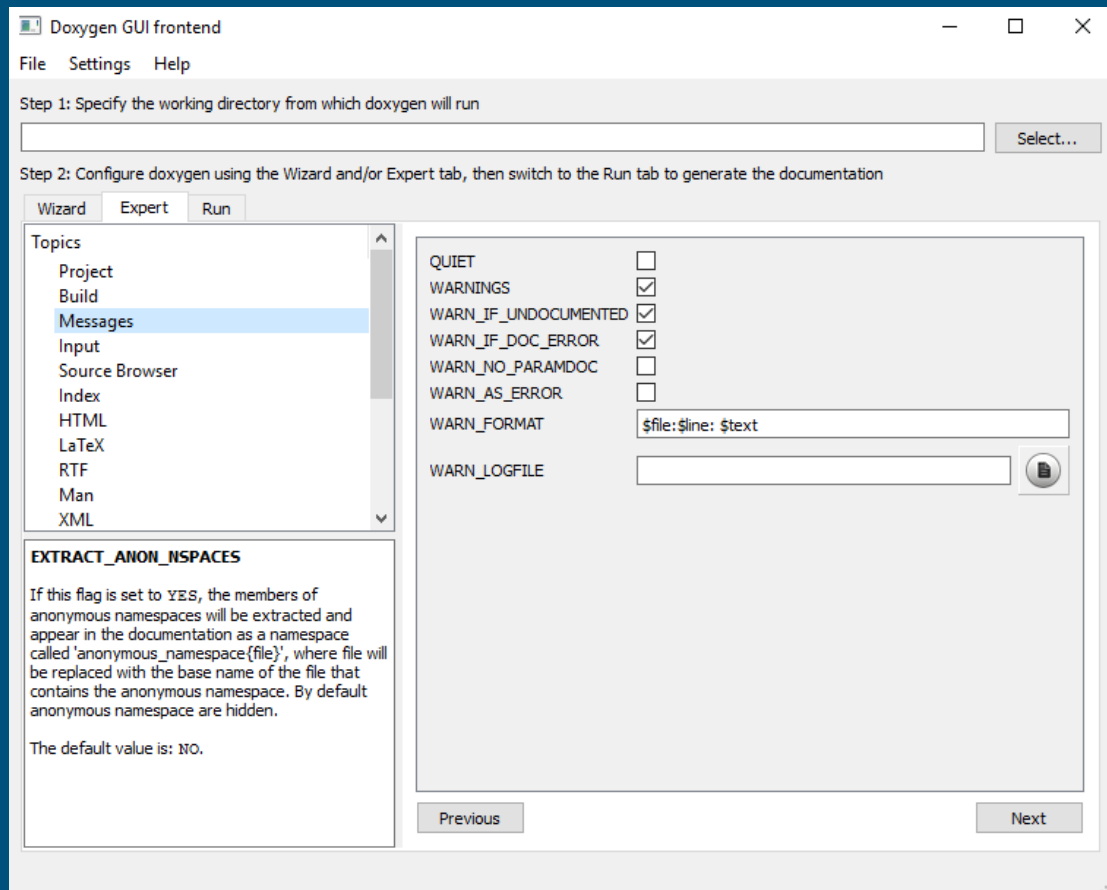
☐ Scan recursively

Specify the directory where doxygen should put the generated documentation

Destination directory: Select...

Previous Next

Usando o Doxygen no Windows



Exemplo de Arquivo em PDF

Chapter 1

Hierarchical Index

1.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Shape	4
Circunferencia	14
Cilindro	20
Paralelogramo	6
TresDimensiones	18
Cilindro	20
Paralelepipedo	24
Triangulo	9

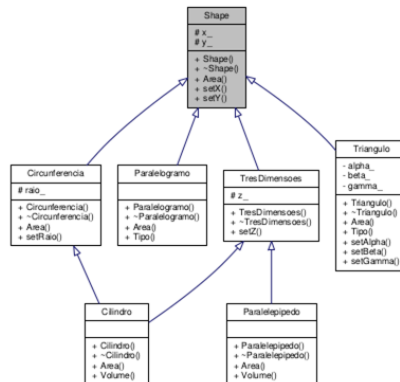
Chapter 4

Class Documentation

4.1 Shape Class Reference

```
#include <Shapes.h>
```

Inheritance diagram for Shape:



Shape
x_ # y_
+ Shape() + ~Shape() + Area() + setX() + setY()

Public Member Functions

- Shape()
- ~Shape()
- virtual float Area() const
- void setX(float x)
- void setY(float y)

Protected Attributes

- float x_
- float y_

4.1.1 Constructor & Destructor Documentation

4.1.1.1 Shape:Shape() [inline]

Constructor: initializes x_ with 2.0 and y_ with 2.0.

4.1.1.2 Shape::~Shape() [inline]

Destructor.

4.1.2 Member Function Documentation

4.1.2.1 virtual float Shape::Area() const [pure virtual]

Implemented in Cilindro, Paralelepipedo, Circunferencia, Paralelogramo, and Triangulo.

4.1.2.2 void Shape::setX(float x)

Function member responsible for setting the x_ value.

Exemplo de Arquivo em HTML

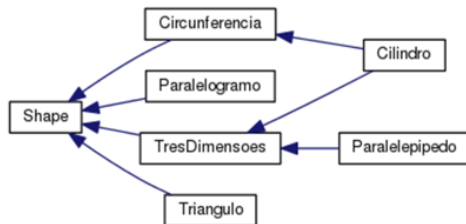
My Project

Main Page **Classes** Files Examples

Class List Class Index **Class Hierarchy** Class Members

Class Hierarchy

Go to the textual class hierarchy



My Project

Main Page **Classes** Files Examples

Class List Class Index Class Hierarchy Class Members

Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

C Cilindro	Derived class Cilindro indicating inheritance relationship with class TresDimensoes and Circunferencia . Multiple inheritance
C Circunferencia	Derived class Circunferencia indicating inheritance relationship with class Shape
C Paralelepipedo	Derived class Paralelepipedo indicating inheritance relationship with class TresDimensoes
C Paralelogramo	Derived class Paralelogramo indicating inheritance relationship with class Shape
C Shape	Base-class
C TresDimensoes	Derived class TresDimensoes indicating inheritance relationship with class Shape
C Triangulo	Derived class Triangulo indicating inheritance relationship with class Shape

Será que existe outra maneira de
organizar o código?

Talvez uma arquitetura de software?

MVC - Model-View-Controller

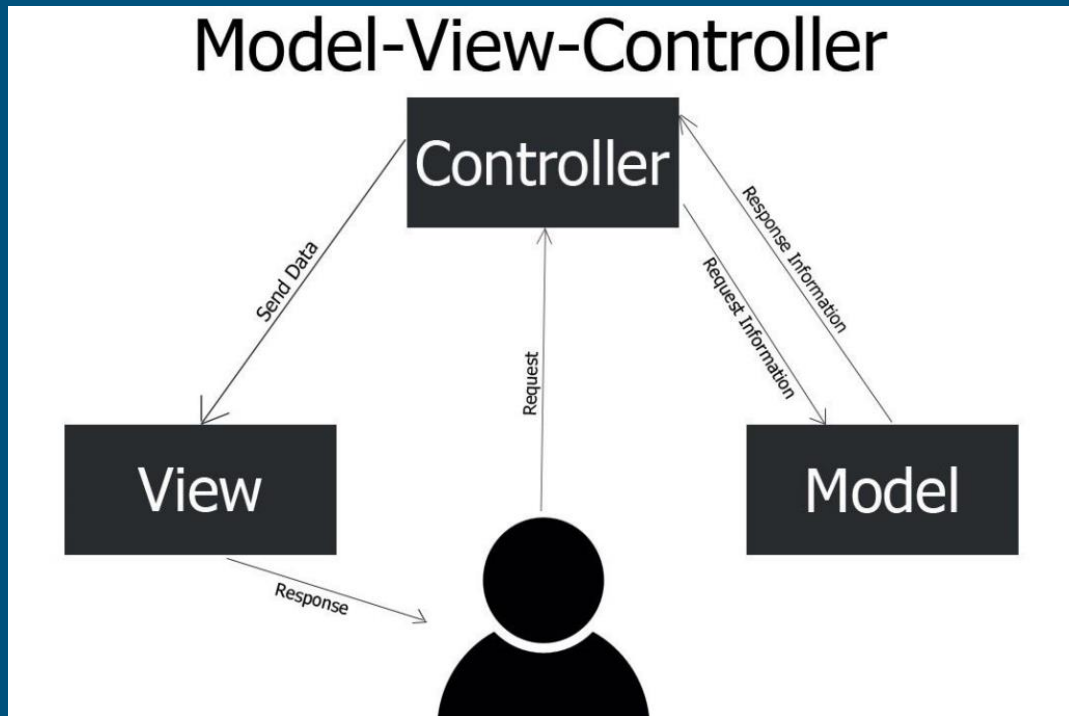
- A arquitetura MVC é utilizada para separar as funcionalidades e a apresentação de uma aplicação
- Divide um dado aplicativo em três partes interconectadas:
 - **Model** (Modelo)
 - **View** (Visualização)
 - **Controller** (Controle)

MVC - Model-View-Controller

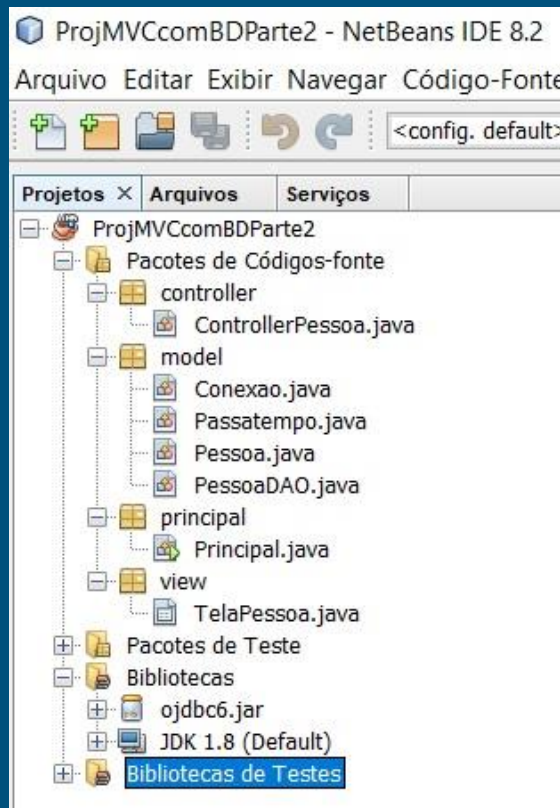
- MVC permite a reutilização eficiente do código e o desenvolvimento paralelo;
- Tradicionalmente usado para interfaces gráficas de usuário (GUIs);
- Também para:
 - Web
 - Mobile

MVC - Model-View-Controller

- **Model** (Modelo): lógica, regra do negócio
- **View** (Visualização): interface com o usuário
- **Controller** (Controle): intermediação entre View e Model



Conceito MVC na implementação Java



- Cada parte do MVC em um pacote:
 - controller
 - model
 - View
- Um pacote (principal) com a classe Principal que inicializa o primeiro formulário e o primeiro controle

Será que existem padrões para
escrever o código?



Design Patterns

Design Patterns

- São soluções reutilizáveis para problemas recorrentes;
- São criados a partir da similaridade entre problemas;
- São descritos através de um formato padrão.

Um **pattern** é independente de qualquer linguagem de programação

Design Patterns - Vantagens

- Transmissão de conhecimentos valiosos
- São importantes para discussão entre programadores
- Pode-se combiná-los para formar outros padrões

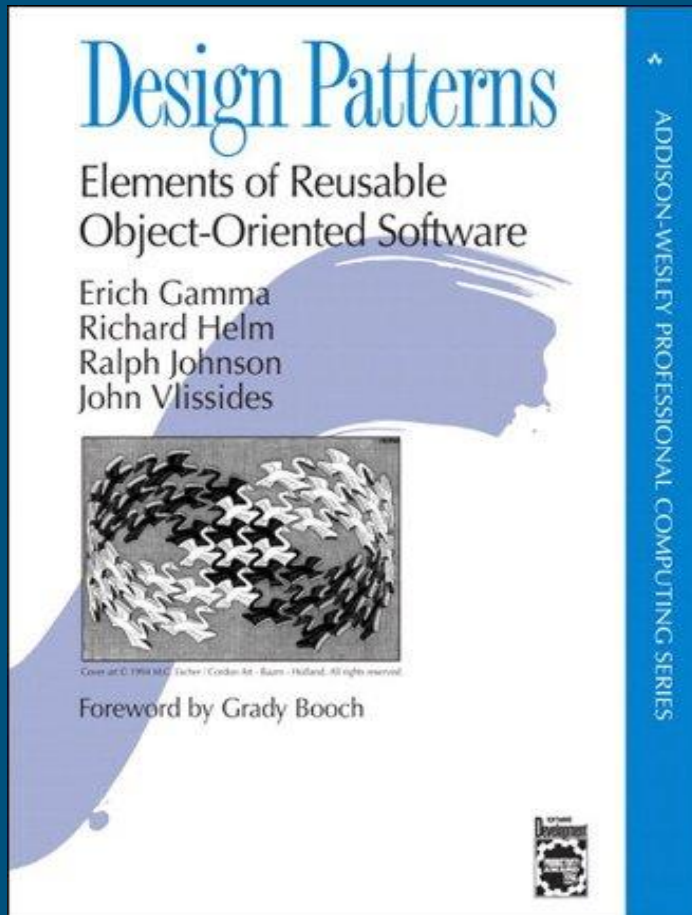
Design Patterns - Desvantagens

- Cada pattern é adaptado para um tipo de problema
- Pode ser problemático se não aplicado corretamente
- Não há patterns suficientes para resolver todos os problemas
- Excesso de patterns pode não ser bom para um software.

"Gang of Four (GoF)"

O movimento ao redor de padrões de projeto ganhou popularidade com o livro **Design Patterns: Elements of Reusable Object-Oriented Software**, publicado em 1995.

Os autores deste livro, Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, são conhecidos como a "Gangue dos Quatro" (Gang of Four) ou simplesmente "GoF".



Classificação dos *Patterns*

- Os *patterns* podem ser classificados em diversas categorias.
- As principais categorias são:
 - **Padrões de Criação**: relacionados à criação de objetos.
 - **Padrões Estruturais**: relacionados às associações entre classes e objetos.
 - **Padrões Comportamentais**: relacionados com a divisão de responsabilidades de cada objeto/classe.

Padrões de Criação (*Creational Patterns*)

Name	Description	In <i>Design Patterns</i>
Abstract factory	Provide an interface for creating <i>families</i> of related or dependent objects without specifying their concrete classes.	Yes
Builder	Separate the construction of a complex object from its representation, allowing the same construction process to create various representations.	Yes
Dependency Injection	A class accepts the objects it requires from an injector instead of creating the objects directly.	No
Factory method	Define an interface for creating a <i>single</i> object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.	Yes
Lazy initialization	Tactic of delaying the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed. This pattern appears in the GoF catalog as "virtual proxy", an implementation strategy for the Proxy pattern.	No
Multiton	Ensure a class has only named instances, and provide a global point of access to them.	No
Object pool	Avoid expensive acquisition and release of resources by recycling objects that are no longer in use. Can be considered a generalisation of connection pool and thread pool patterns.	No
Prototype	Specify the kinds of objects to create using a prototypical instance, and create new objects from the 'skeleton' of an existing object, thus boosting performance and keeping memory footprints to a minimum.	Yes
Resource acquisition is initialization (RAII)	Ensure that resources are properly released by tying them to the lifespan of suitable objects.	No
Singleton	Ensure a class has only one instance, and provide a global point of access to it.	Yes

Padrões Estruturais (*Structural patterns*)

Name	Description	In <i>Design Patterns</i>
Adapter, Wrapper, or Translator	Convert the interface of a class into another interface clients expect. An adapter lets classes work together that could not otherwise because of incompatible interfaces. The enterprise integration pattern equivalent is the translator.	Yes
Bridge	Decouple an abstraction from its implementation allowing the two to vary independently.	Yes
Composite	Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.	Yes
Decorator	Attach additional responsibilities to an object dynamically keeping the same interface. Decorators provide a flexible alternative to subclassing for extending functionality.	Yes
Extension object	Adding functionality to a hierarchy without changing the hierarchy.	No
Facade	Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.	Yes
Flyweight	Use sharing to support large numbers of similar objects efficiently.	Yes
Front controller	The pattern relates to the design of Web applications. It provides a centralized entry point for handling requests.	No
Marker	Empty interface to associate metadata with a class.	No
Module	Group several related elements, such as classes, singletons, methods, globally used, into a single conceptual entity.	No
Proxy	Provide a surrogate or placeholder for another object to control access to it.	Yes
Twin ^[20]	Twin allows modeling of multiple inheritance in programming languages that do not support this feature.	No

https://en.wikipedia.org/wiki/Software_design_pattern

Padrão Comportamental (Behavioral Pattern)

Name	Description	In <i>Design Patterns</i>
Blackboard	Artificial intelligence pattern for combining disparate sources of data (see blackboard system)	No
Chain of responsibility	Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.	Yes
Command	Encapsulate a request as an object, thereby allowing for the parameterization of clients with different requests, and the queuing or logging of requests. It also allows for the support of undoable operations.	Yes
Interpreter	Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.	Yes
Iterator	Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.	Yes
Mediator	Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it allows their interaction to vary independently.	Yes
Memento	Without violating encapsulation, capture and externalize an object's internal state allowing the object to be restored to this state later.	Yes
Null object	Avoid null references by providing a default object.	No
Observer or Publish/subscribe	Define a one-to-many dependency between objects where a state change in one object results in all its dependents being notified and updated automatically.	Yes
Servant	Define common functionality for a group of classes. The servant pattern is also frequently called helper class or utility class implementation for a given set of classes. The helper classes generally have no objects hence they have all static methods that act upon different kinds of class objects.	No
Specification	Recombinable business logic in a Boolean fashion.	No
State	Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.	Yes
Strategy	Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.	Yes
Template method	Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.	Yes
Visitor	Represent an operation to be performed on the elements of an object structure. Visitor lets a new operation be defined without changing the classes of the elements on which it operates.	Yes

https://en.wikipedia.org/wiki/Software_design_pattern

Fim

