



CCM310

Arquitetura de Software e

Programação Orientada a Objetos

Profa. Dra. Gabriela Biondi

Prof. Dr. Isaac Jesus

Prof. Dr. Luciano Rossi

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

Vantagens

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

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 da *patterns* pode não ser bom para um software.

Um pouco de história...

- Christopher Alexander em seus livros* (1977/1979), estabelece que um padrão deve ter, idealmente, algumas características, como, por exemplo:
 - Encapsulamento: um padrão encapsula um problema ou solução bem definida.
 - Generalidade: todo padrão deve permitir a construção de outras realizações a partir deste padrão.
 - Abertura: um padrão deve permitir a sua extensão para níveis mais baixos de detalhe.

*Notes on the Synthesis of Form, The Timeless Way of Building e A Pattern Language

Formato de um *pattern*

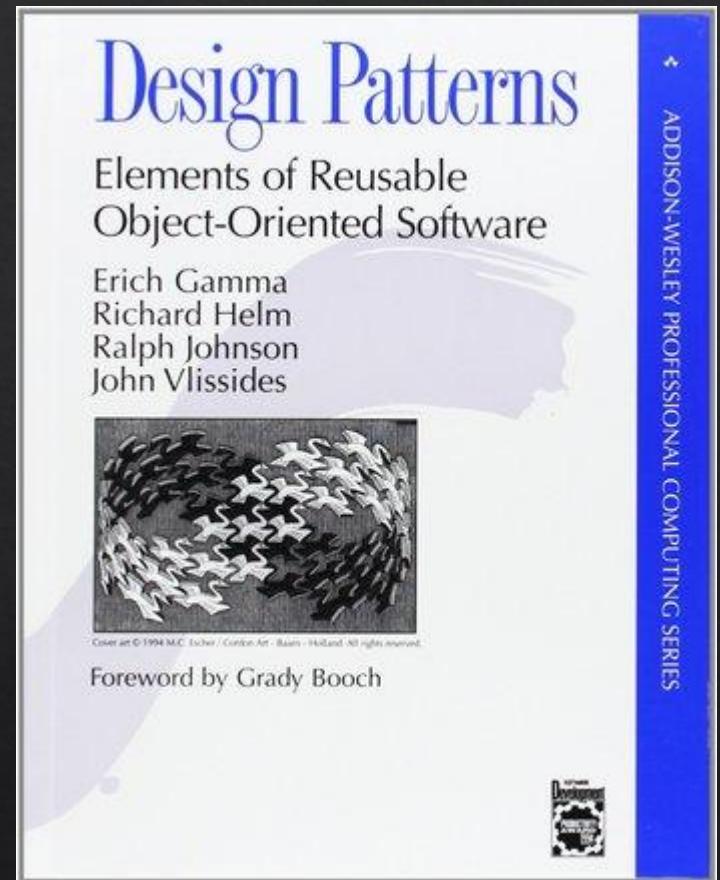
O formato dos *patterns*, também definido por Alexander, é normalmente descrito em cinco partes:

- **Nome:** uma descrição da solução
- **Exemplo:** figuras, diagramas ou descrições
- **Contexto:** a descrição das situações sob as quais o padrão se aplica.
- **Problema:** uma descrição das forças e restrições envolvidos
- **Solução:** relacionamentos estáticos e regras dinâmicas descrevendo como construir artefatos de acordo com o padrão. Inclui referências a outras soluções e o relacionamento com outros padrões de nível mais baixo ou mais alto

“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 Design Patterns
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
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 (dependency injection ^[19]).	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.	Yes
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 Design Patterns
Adapter or 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 [23]	Twin allows modeling of multiple inheritance in programming languages that do not support this feature.	No

Padrão Comportamental (*Behavioral Pattern*)

Name	Description	In Design Patterns
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.	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

Padrões de criação

Factory

- É um dos *patterns* mais utilizados
- De forma geral todos os padrões Factory (*Simple Factory*, *Factory Method*, *Abstract Factory*) encapsulam a criação de objetos.

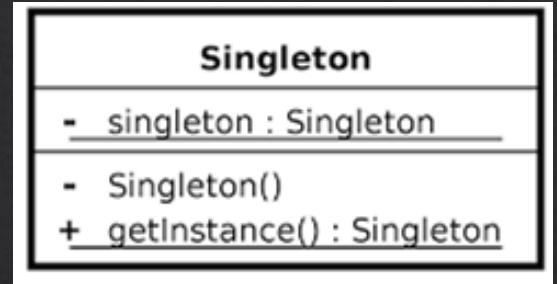


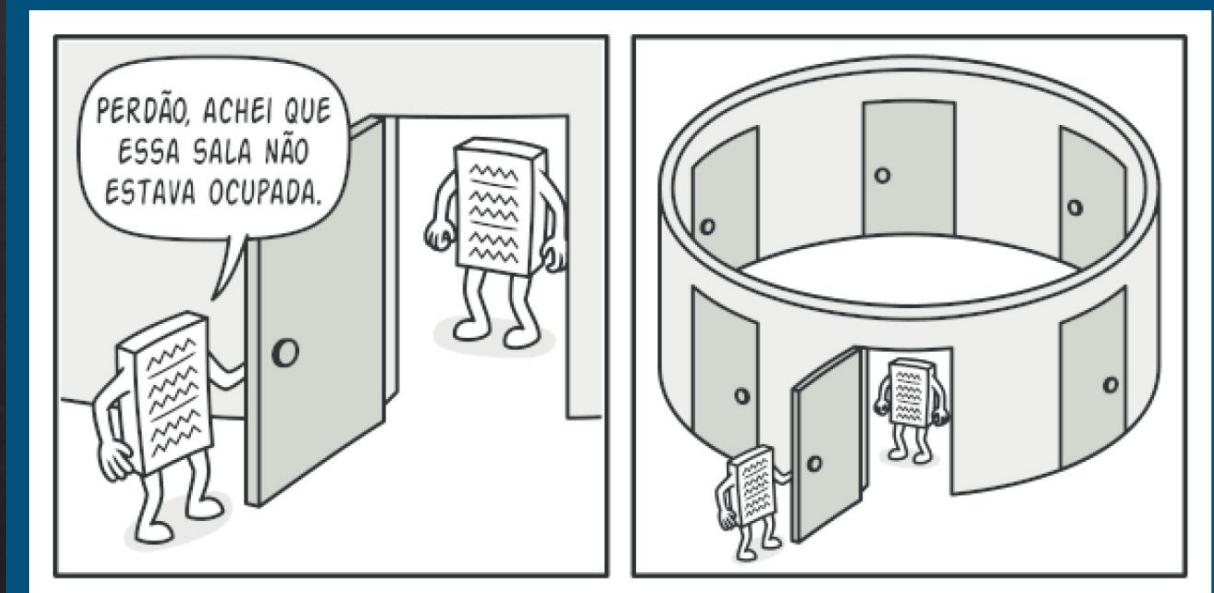
Simple Factory - Exemplo

Padrões de criação

Singleton

- O padrão Singleton é um padrão que restringe a instanciação de uma classe a uma instância "única".
- Isso é útil quando exatamente um objeto é necessário para coordenar ações no sistema.
 - Exemplo: Um objeto que cuida do acesso ao banco de dados





Singleton - Exemplo

Padrões comportamentais

Strategy

- Padrão bastante popular e de simples entendimento
- O comportamento da classe ou de seu algoritmo podem mudar em tempo de execução.
- Utilizada quando uma classe define muitos comportamentos selecionados por diversos *IFs*.



Várias estratégias para se chegar ao aeroporto.

Strategy - Exemplo

Obrigada pela sua
participação, nos vemos na
próxima aula! :)