PADRÕES DE PROJETOS

Alexandre Mendonça Fava
alexandre.fava@udesc.br

Universidade do Estado de Santa Catarina – UDESC
Programa de Pós-graduação em Computação Aplicada – PPGCA

Roteiro

- Definição
- História
- Catálogo
- Resumo
- Referências

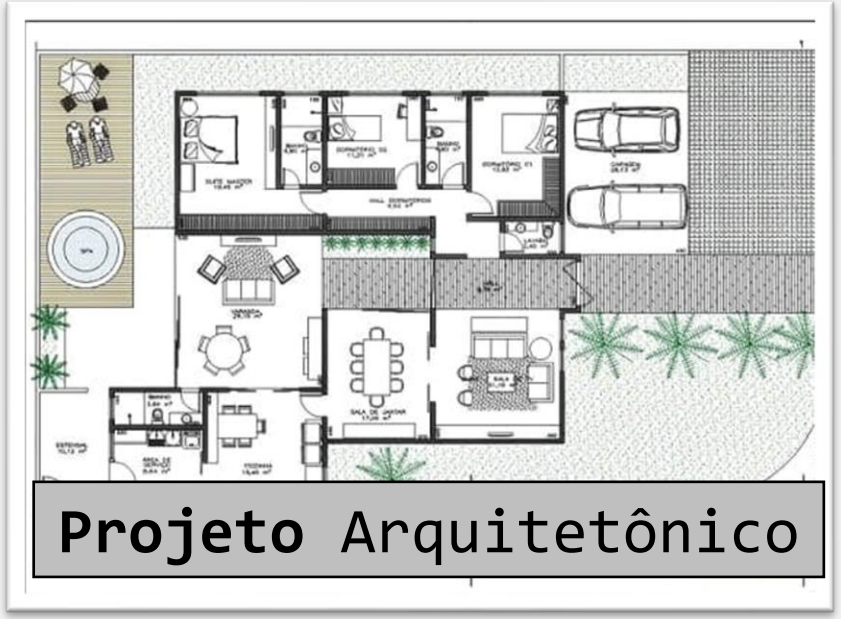
Definição

Padrão de Projeto

Consenso definido para a elaboração de algo

Padrão de tomada brasileiro

Planejamento para o desenvolvimento de algo



Projeto Arquitetônico

Definição

Minhas biblioteca são soluções para problemas.

Não é bem assim...

Padrão de Projeto

Métodos **planejados** para solucionar problemas de desenvolvimento de *software* de forma **consensual**

VALENTE, Marco. **Engenharia de Software Moderna.**

Cada padrão descreve um problema que sempre ocorre em nosso contexto e uma solução para ele, de forma que possamos usá-la um milhão de vezes.

Definição

Biblioteca

São trechos de códigos já prontos para solução de problemas

PROBLEMA:

```
Imprimir valores
Ler valores
```

stdio.h

Padrão de Projeto

São soluções genéricas para problemas recorrentes

PROBLEMA:

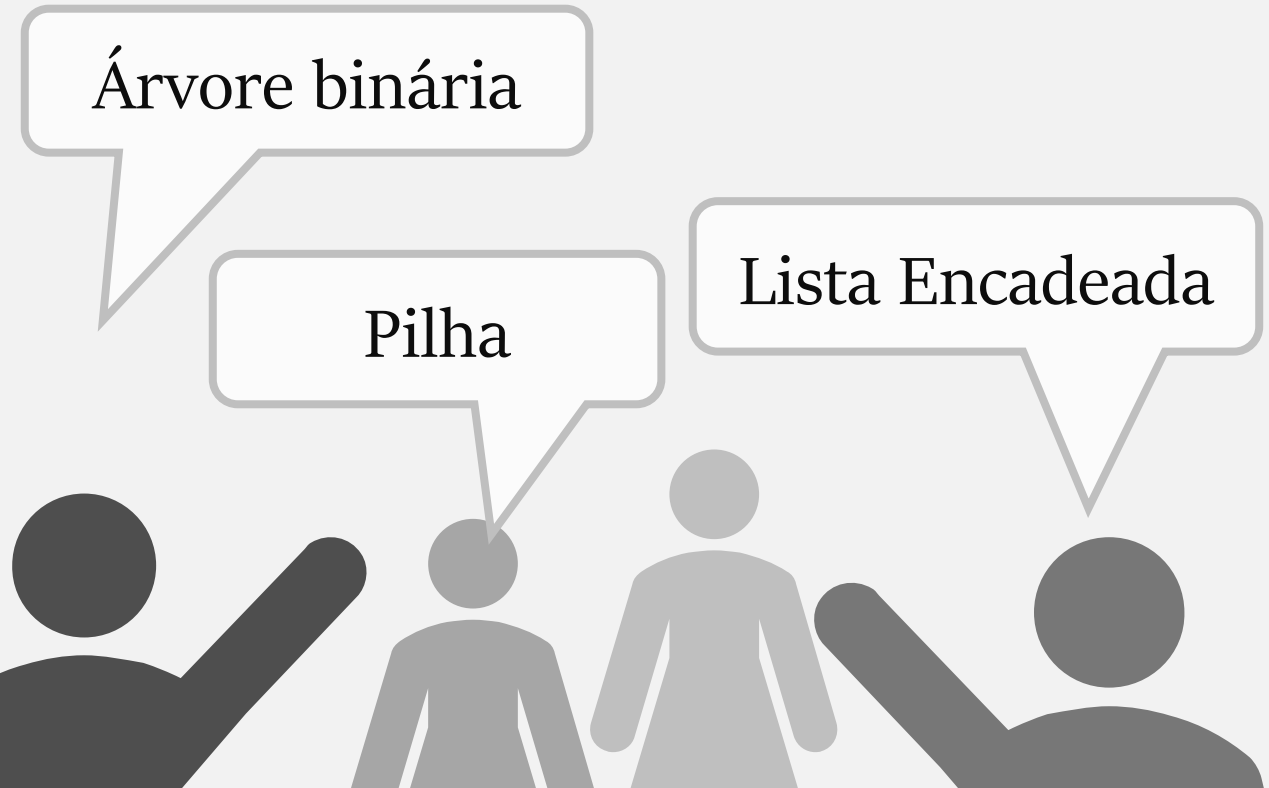
```
Garantir uma única instância
```

singleton

História

Antes

ESTRUTURA DE DADOS

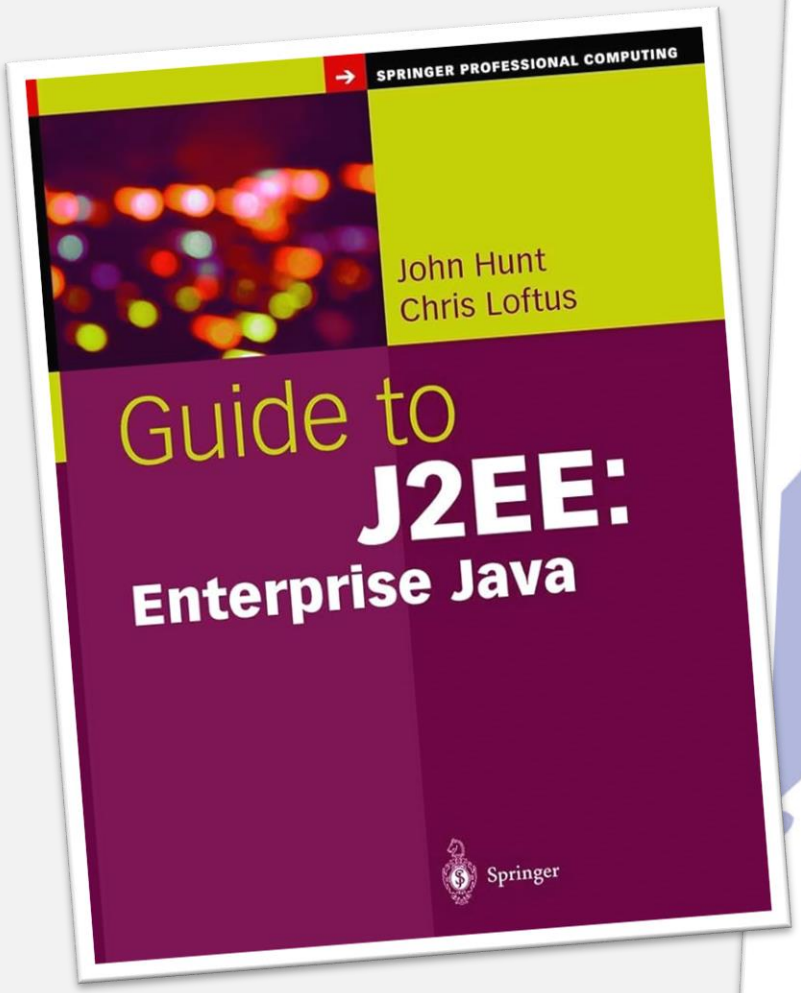


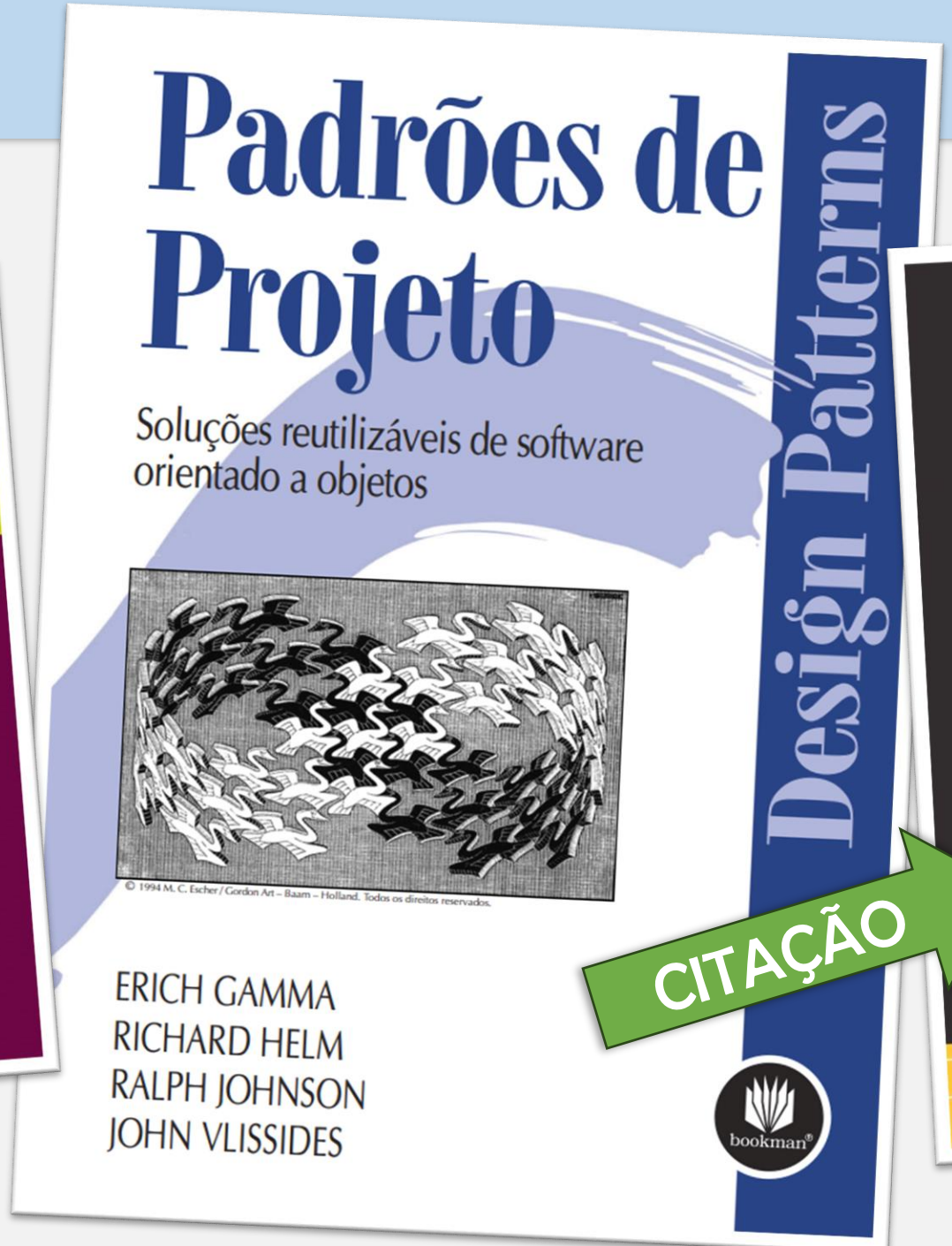
Depois

ORIENTAÇÃO A OBJETOS

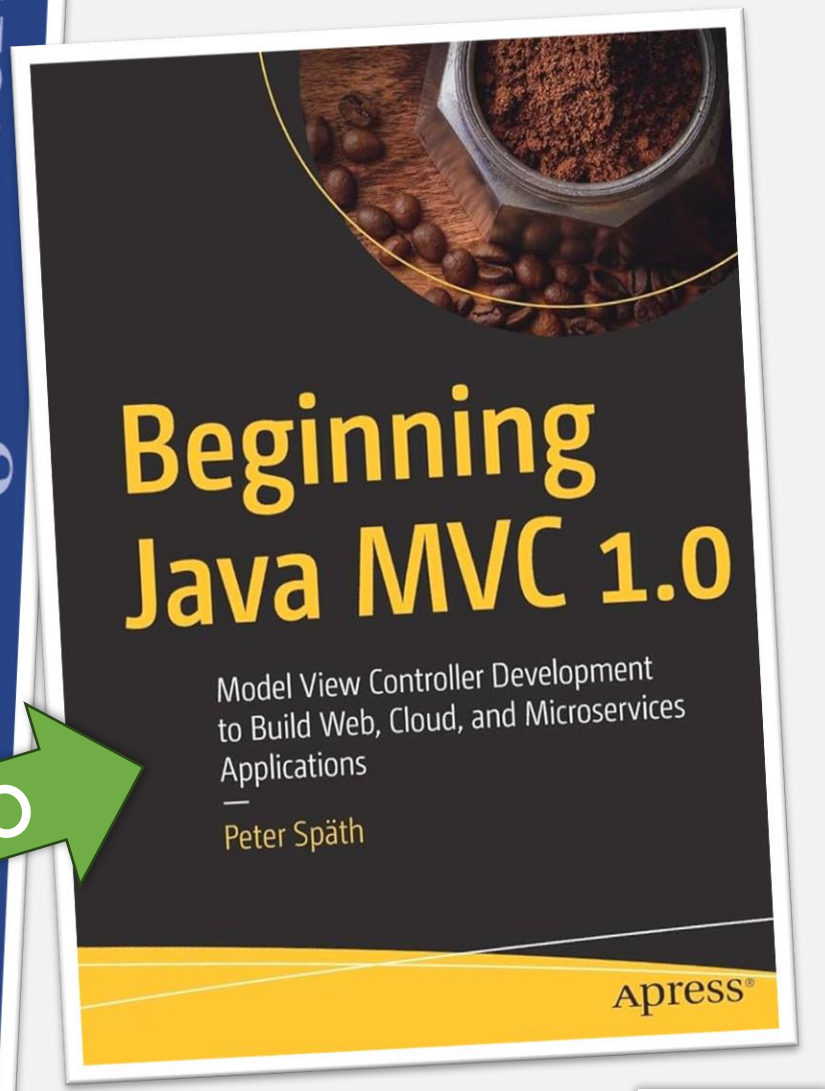
A Gangue dos Quatros (GoF)

História





CITAÇÃO



História

GAMMA, Erich, et. al. **Padrões de Projeto.**

Projetar software orientado a objetos é difícil, mas projetar software *reutilizável* orientado a objetos é ainda mais complicado.

Uma coisa que os melhores projetistas sabem que não devem fazer é resolver cada problema a partir de princípios elementares ou do zero. Em vez disso, eles reutilizam soluções que funcionaram no passado.

Esses padrões resolvem problemas específicos de projetos e tornam os projetos orientados a objetos mais flexíveis e, em última instância, reutilizáveis. Eles ajudam os projetistas a reutilizar projetos bem-sucedidos ao basear os novos projetos na experiência anterior. Um projetista que está familiarizado com tais padrões pode aplicá-los imediatamente a diferentes problemas de projeto, sem necessidade de redescobri-los.

História

GAMMA, Erich, et. al. **Padrões de Projeto.**

Neste livro concentramos-nos sobre os padrões que estão em um certo nível de abstração. Padrões de projeto não são projetos, como listas encadeadas e tabelas de acesso aleatório, que podem ser codificadas em classes e ser reutilizadas tais como estão. Tampouco são projetos complexos, de domínio específico, para uma aplicação inteira ou subsistema. Padrões de projeto, neste livro, são *descrições de objetos e classes comunicantes que precisam ser personalizadas para resolver um problema geral de projeto num contexto particular.*

Padrões de Projeto

O catálogo proposto apresenta 23 soluções/padrões para o desenvolvimento de *software*

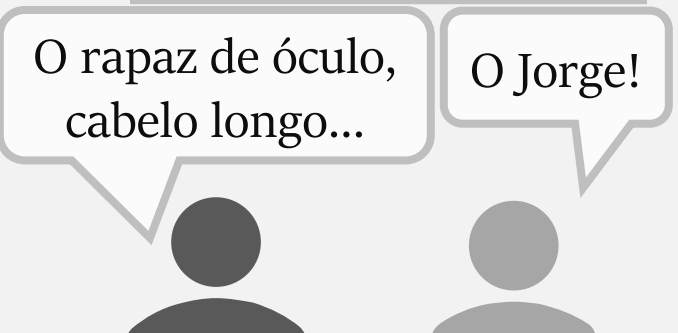
GAMMA, Erich, et. al. Padrões de Projeto.

		Propósito		
		De criação	**Estrutural**	**Comportamental**
Escopo	**Classe**	Factory Method (112)	Adapter (class) (140)	Interpreter (231) Template Method (301)
	Objeto	Abstract Factory (95) Builder (104) Prototype (121) Singleton (130)	Adapter (object) (140) Bridge (151) Composite (160) Decorator (170) Façade (179) Flyweight (187) Proxy (198)	Chain of Responsibility (212) Command (222) Iterator (244) Mediator (257) Memento (266) Observer (274) State (284) Strategy (292) Visitor (305)

Padrões de Projeto

UM PADRÃO DEVE POSSUIR QUATRO ELEMENTOS ESSENCIAIS

Nome	Problema	Solução	Consequência
O nome torna mais fácil pensar sobre projetos e a comunicá-los	Descreve em que situação aplicar o padrão	Descreve os relacionamentos, as responsabilidades e colaborações do padrão	Vantagens e desvantagens em termos de flexibilidade, extensibilidade ou a portabilidade



Padrões de Projeto

Consequência

FLEXIBILIDADE	EXTENSIBILIDADE	PORTABILIDADE
Capacidade de um programa de se adaptar a mudanças sem exigir uma grande reescrita do código	Capacidade de estender um programa com novas funcionalidades sem afetar as existentes	Capacidade de um programa de poder executar nos mais variados ambientes, cenários e tecnologias

Classificação dos padrões de projeto

CRIACIONAL	ESTRUTURAL	COMPORTAMENTAL
Relacionados à criação de objetos	Relacionados com as associações entre classes e objetos	Relacionados com as interações e divisões de responsabilidade entre classes e objetos

Abstract Factory

Propósito: Promove uma interface para criação de famílias de objetos relacionados sem especificar sua classe concreta

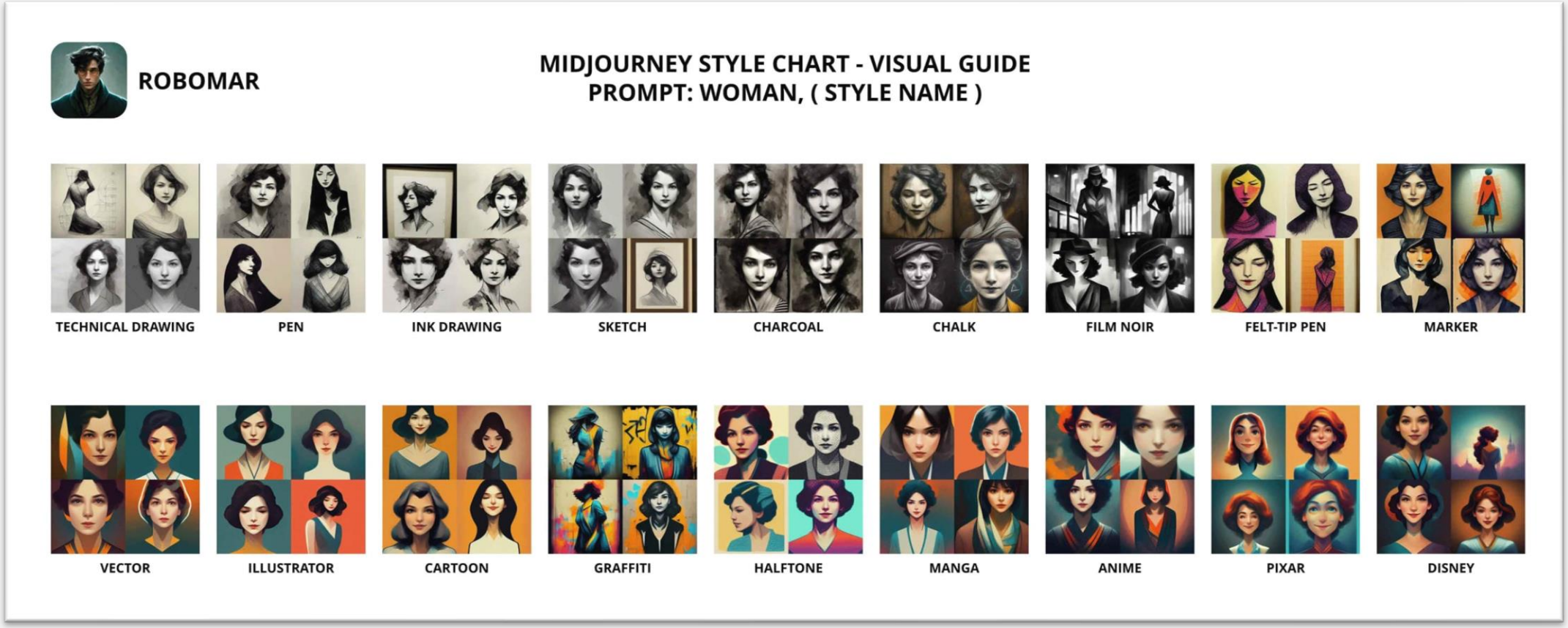
Padrão **criacional** para a instanciação de **objetos** que compartilhem um "tema" em comum

Exemplo: Uma fábrica cria vários móveis (mesas, cadeiras). Esses móveis podem ter estilos diferentes (Barroco).

O programa não precisa ter as classes concretas para criar mesa e cadeira, basta ter uma "fábrica abstrata".

Abstract Factory

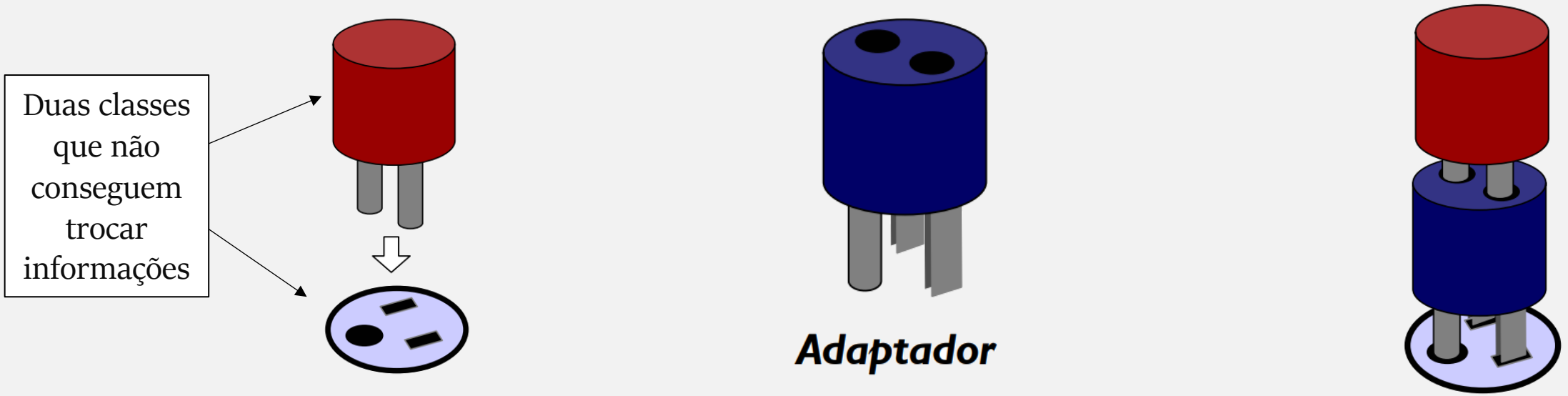
A geração de imagens por IA, onde uma mesma imagem pode ser gerada com diversos estilos diferentes, pode ser considerada um exemplo do padrão *Abstract Factory*



Adapter

Propósito: Converte a interface de uma classe em outra interface esperada pelos clientes (herança múltipla)

Padrão **estrutural** para a definição de **classes** permitindo a comunicação entre classes que não poderiam trabalhar juntas devido à incompatibilidade de suas interfaces



Bridge

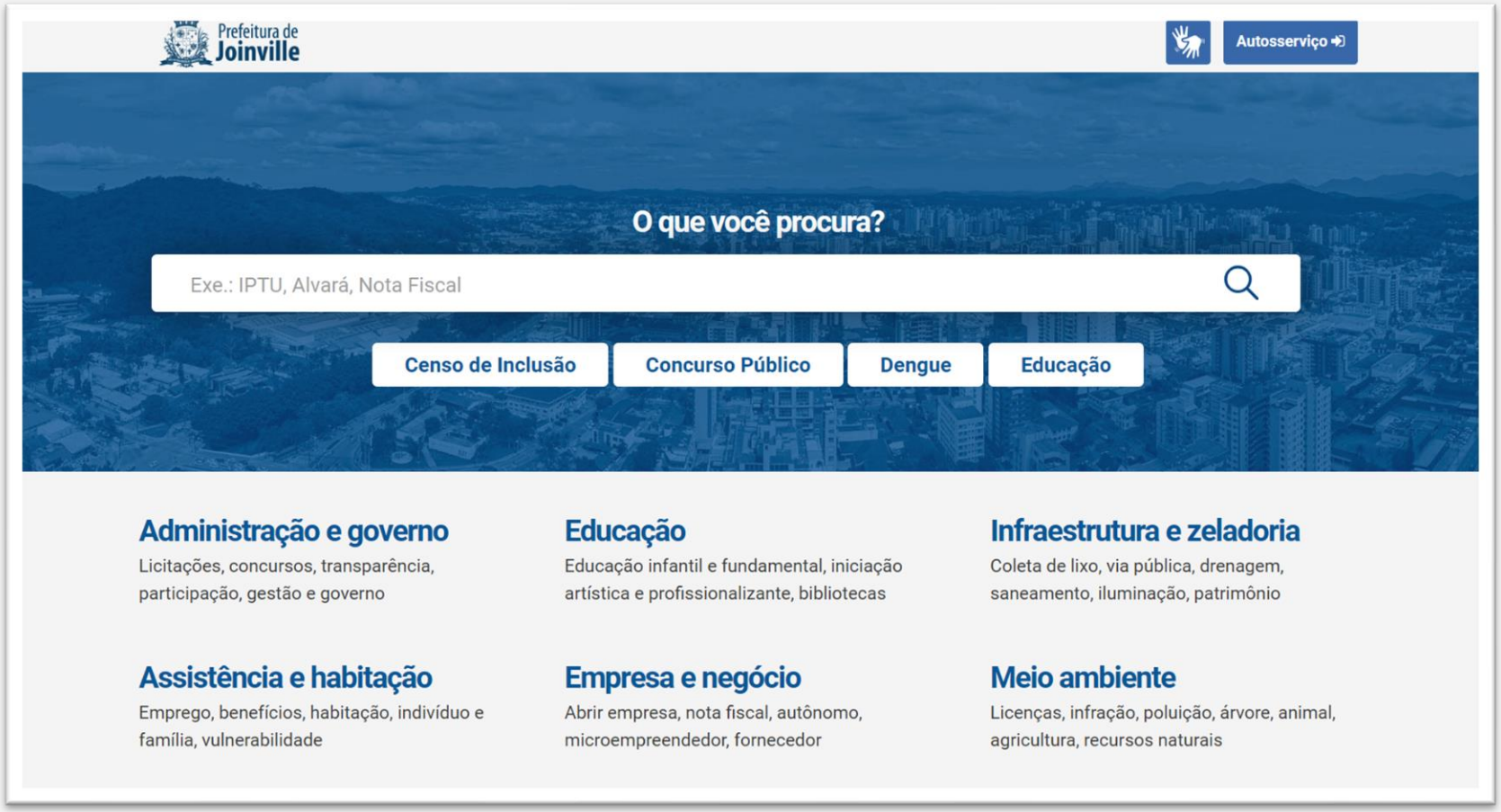
Propósito: Separa uma abstração da sua implementação, de modo que as duas possam variar independentemente

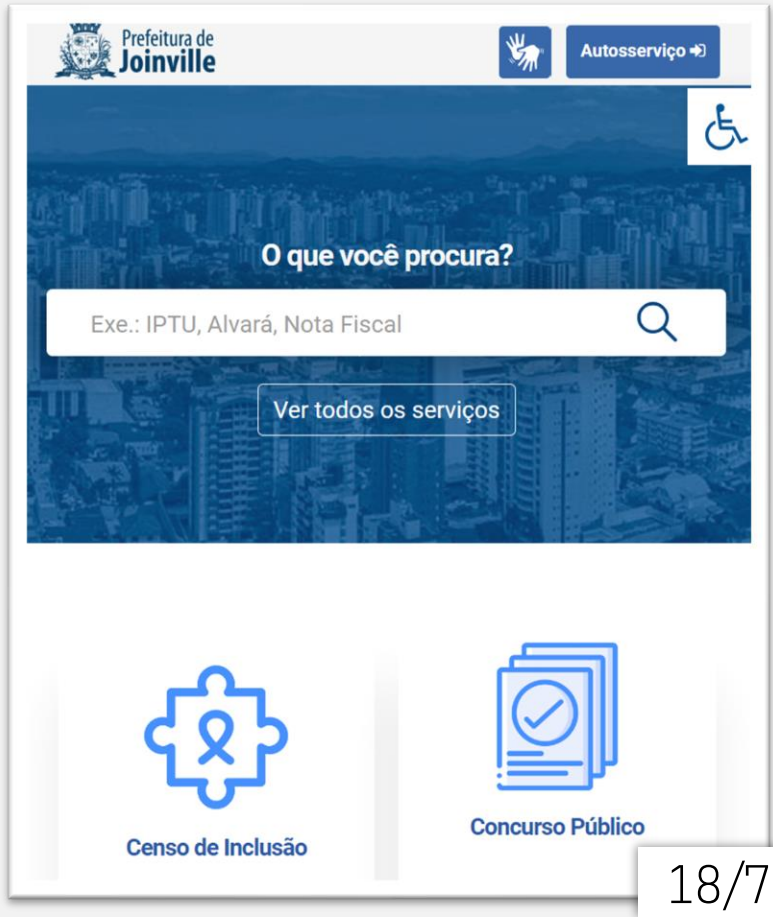
Padrão **estrutural** para a instanciação de **objetos**
que possam variar de forma independente (desacoplamento)

Exemplo: Um *site* pode ser portável tanto para dispositivos móveis quanto para computadores de mesa.
Os elementos do *site* aparecem e podem até ter execuções distintas dependendo do dispositivo aberto.

Bridge

Os elementos do site aparecem e podem até ter execuções distintas dependendo do dispositivo aberto.





Builder

Propósito: Separa a construção de um objeto complexo da sua representação, de modo que o mesmo processo de construção possa criar diferentes representações

Padrão **criacional** que facilita a instanciação de **objetos** que têm muitos atributos

Exemplo: Na hora de construir um objeto do tipo CARRO, pode ser requisitado uma série de especificidades.

Para facilitar o usuário na criação de carro, basta implementar um **BUILDER** que servirá de auxílio.

Builder

Uma alternativa ao uso de um **Builder** seria a implementação de vários métodos construtores.

Desta forma, o usuário podem utilizar o método construtor (**com assinaturas diferentes**) que melhor lhe agradar.

Código (Carro.java)

```
1  public class Carro{
2
3   Carro(){
4    cor = 'B';
5    tanque = 100;
6    velocidade = 0;
7   }
8
9   Carro(char colorir){
10   cor = colorir;
11   tanque = 100;
12   velocidade = 0;
13  }
14
```

Chain of Responsibility

Propósito: Encadeia os objetos receptores e passa a solicitação ao longo da cadeia até que um objeto a trate

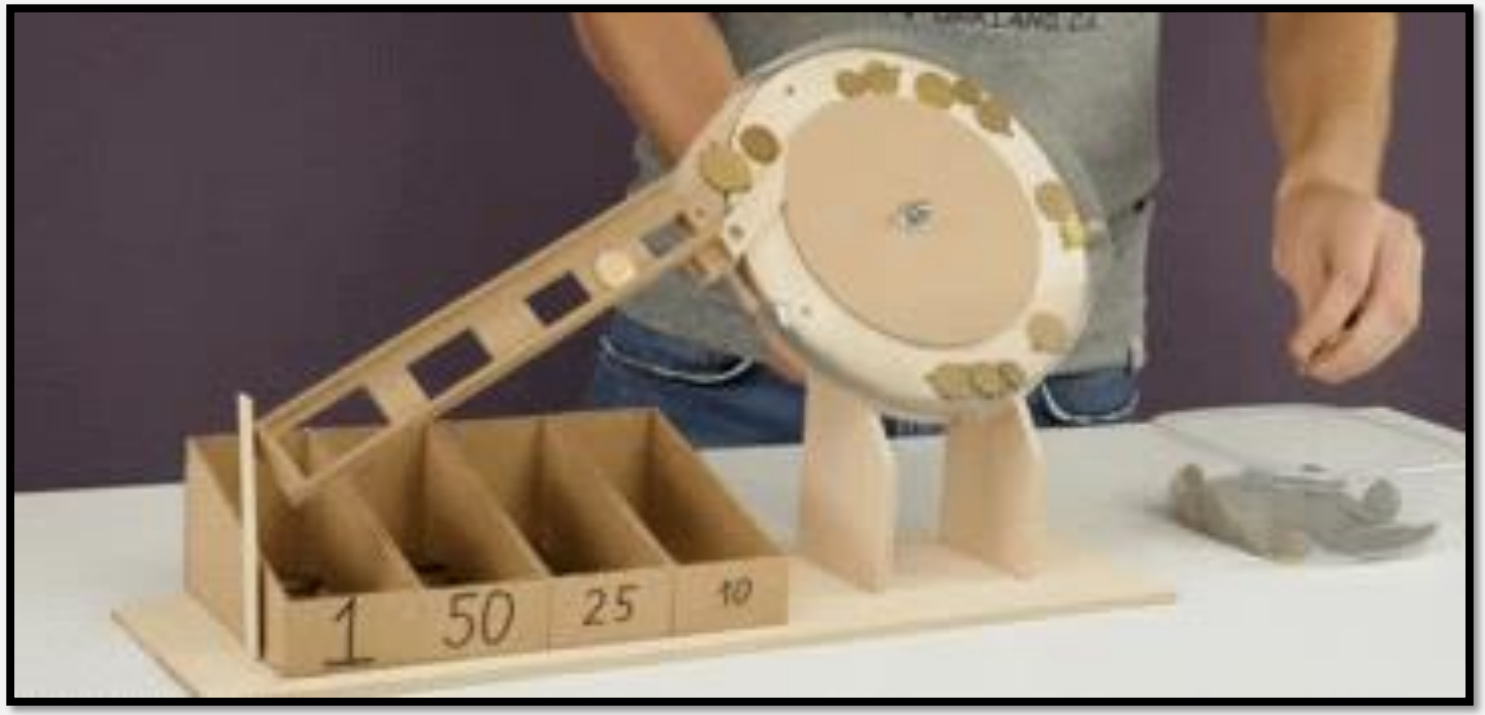
Padrão **comportamental** que permite que uma requisição passe por uma corrente de **objetos** até encontrar um que a processe

Exemplo: Dividir para conquistar. Mais fácil que resolver um problema complexo é resolver uma parte dele.

Os objetos serão criados para resolver uma parte do problema (possível apenas em problemas que podem ser decompostos, *e.g.* matrizes)

Chain of Responsibility

Realiza a divisão de responsabilidade de uma tarefa de forma transparente



Todos os objetos, `caixa1`, `caixa50`, `caixa25` e `caixa10`, possuem o mesmo método `receberdinheiro()`, porém executam ele de forma distinta para as mesma entradas. A execução é em cadeia.

Command

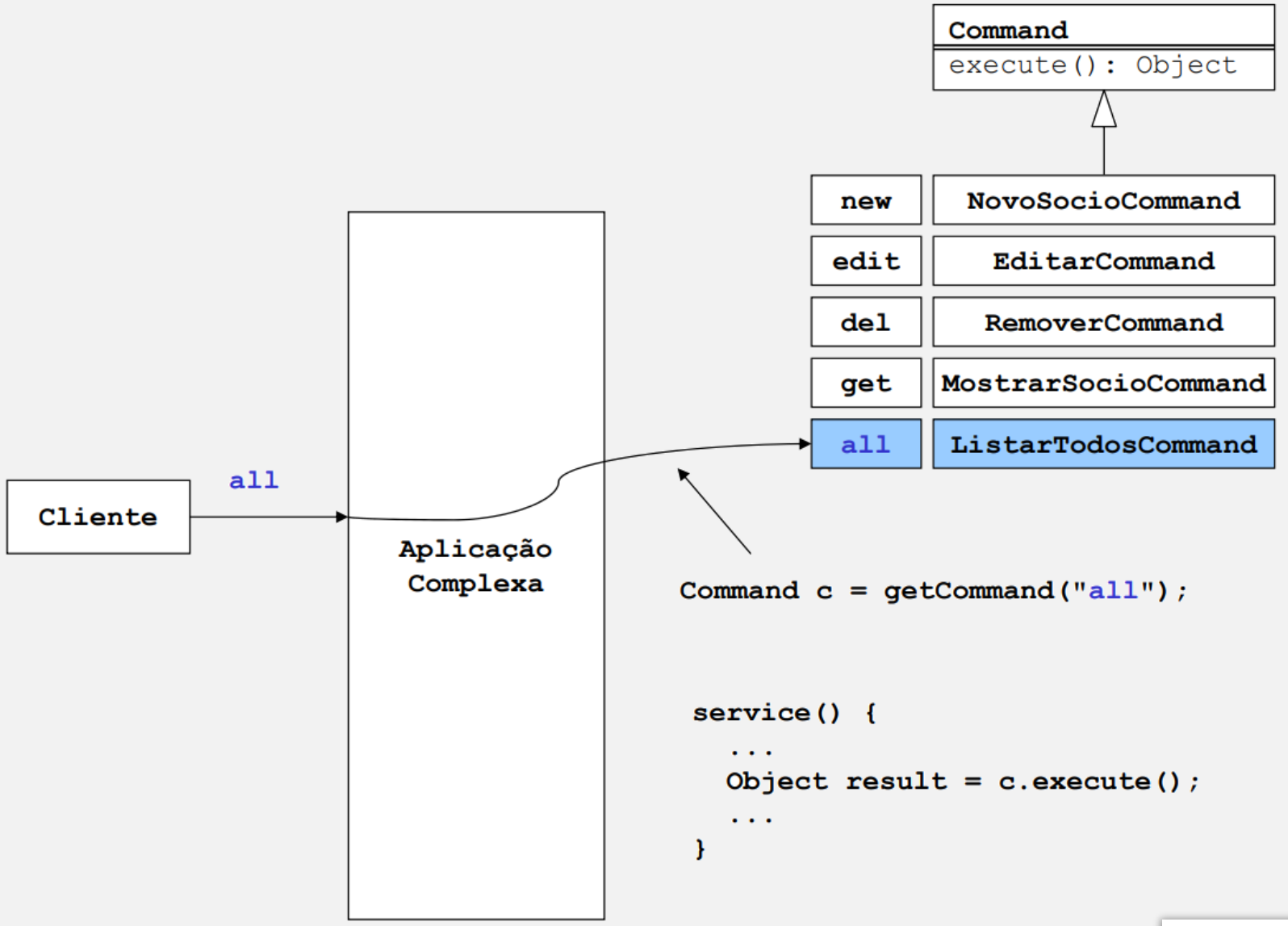
Propósito: Transforma as solicitações em objeto, permitindo que clientes parametrizem diferentes requisições

Padrão **comportamental** que encapsula uma instrução em um **objeto**

Exemplo: Eu quero acessar as funcionalidades de todos os objetos que eu instanciei (seus métodos).
Ao invés de criar um método eu posso transformar o comando em objeto para ser usado por outros objetos.

Command

Ao invés de criar um método em cada uma das minhas classes para listar todos os métodos disponíveis (comandos disponíveis). Eu posso criar um objeto que executa o comando pedido em qualquer classe conforme a requisição do usuário.



Composite

Eu amor recursividade! 

Propósito: Descreve como construir uma hierarquia de classes composta para dois tipos de objetos: primitivos e compostos

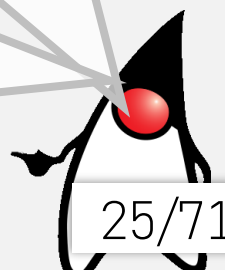
Padrão **estrutural** que elabora uma interface padrão para **objetos** e **composições de objetos**

Recursividade? Isso não pode dar problema?

Dá nada, só se o relacionamento for cíclico.

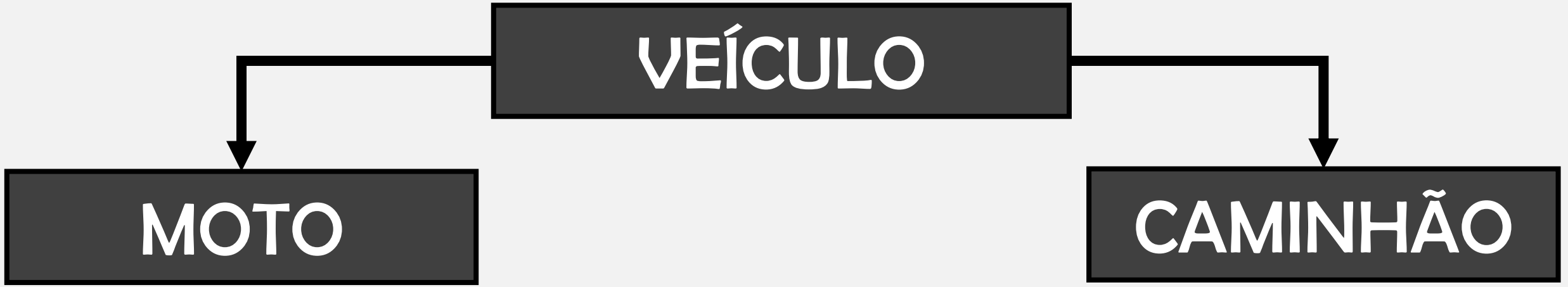
Outra coisa... Um dos seus pilares já não é a **composição**?

A **composição** é utilizada na construção de objetos compondo um dentro de outro. Já o **composite** é um jeito de tratar objetos e agregações de maneira uniforme.



Composite

É possível ter inúmeros objetos instanciados e grupos de objetos. Passar instruções para cada objeto pode demandar muito trabalho. Com o `composite` é possível enviar todos as `motos` para o lava-jato.



Código (Moto.java)

```
1  public class Moto
   extends Veiculo{
2   int inclinacao;
```

Código (Veiculo.java)

```
1  public class Veiculo{
2   private char cor;
3   private int ano;
```

Código (Caminhao.java)

```
1  public class Caminhao
   extends Veiculo{
2   int capacidade;
```

Decorator

Propósito: Fornecem uma alternativa flexível a subclasses para extensão da funcionalidades

É uma forma de herança onde novos métodos são adicionados ao objeto, onde a responsabilidade recaí ao objeto e não à classe.

VALENTE, Marco. **Engenharia de Software Moderna.**

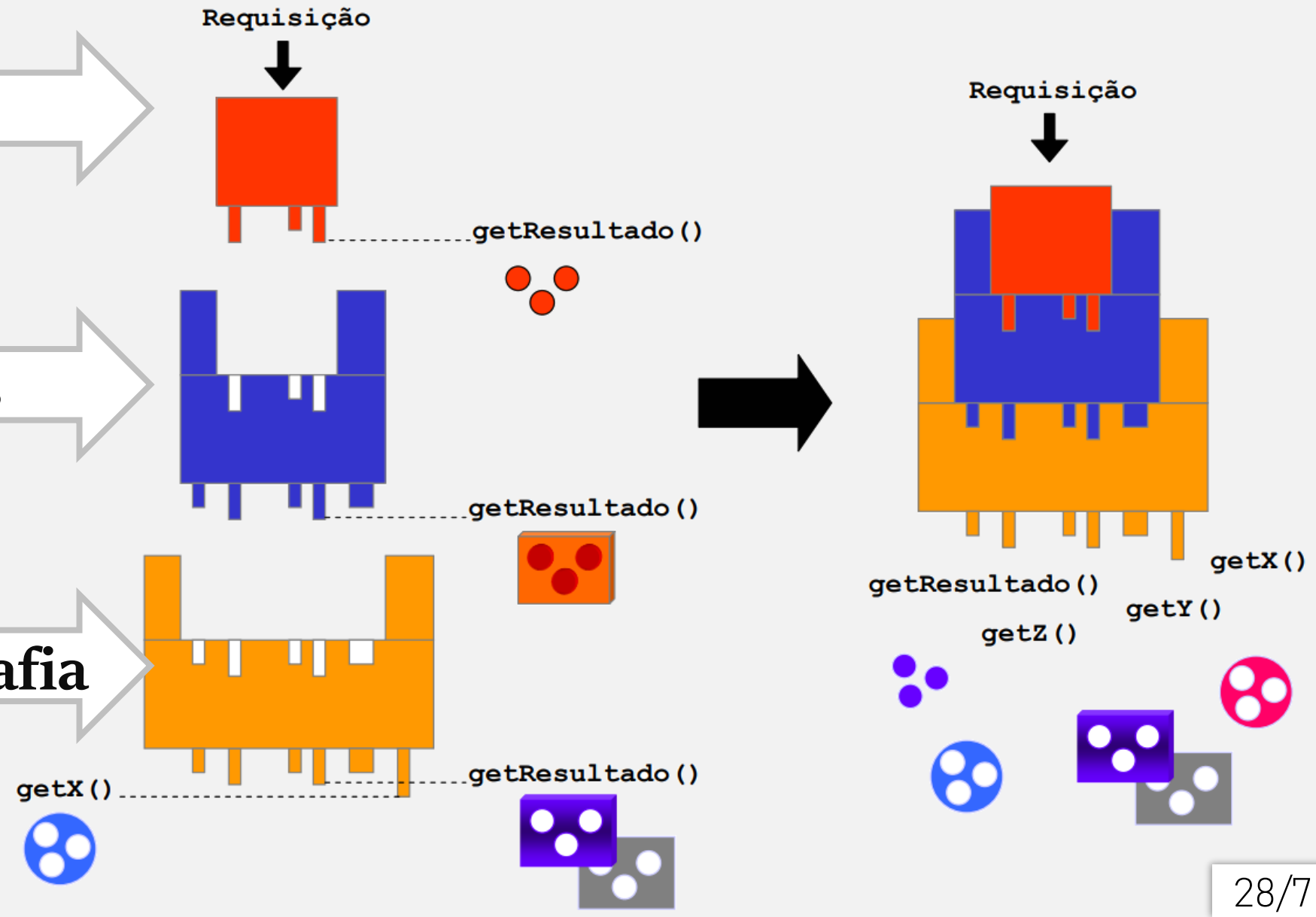
Solução: O Padrão Decorador representa uma alternativa a herança quando se precisa adicionar novas funcionalidades em uma classe base. Em vez de usar herança, usa-se composição para adicionar tais funcionalidades dinamicamente nas classes base. Portanto, Decorador é um exemplo de aplicação do princípio de projeto "Prefira Composição a Herança", que estudamos no capítulo anterior.

Decorator

Classe para **escrever**

Classe para **arquivos**

Classe para **criptografia**



Façade

Propósito: Fornece uma interface unificada para um conjunto de interfaces em um subsistema (uma fachada).

Padrão **estrutural** que elabora uma interface padrão de comunicação entre **objetos** de diferentes classes

Exemplo: Usuários tem dificuldade usar um sistema pois suas classes internas são demasiadamente complexas.
Ao invés do usuário ter que saber usar classes desse sistema ele pode usar o sistema por uma **Fachada**.

Façade

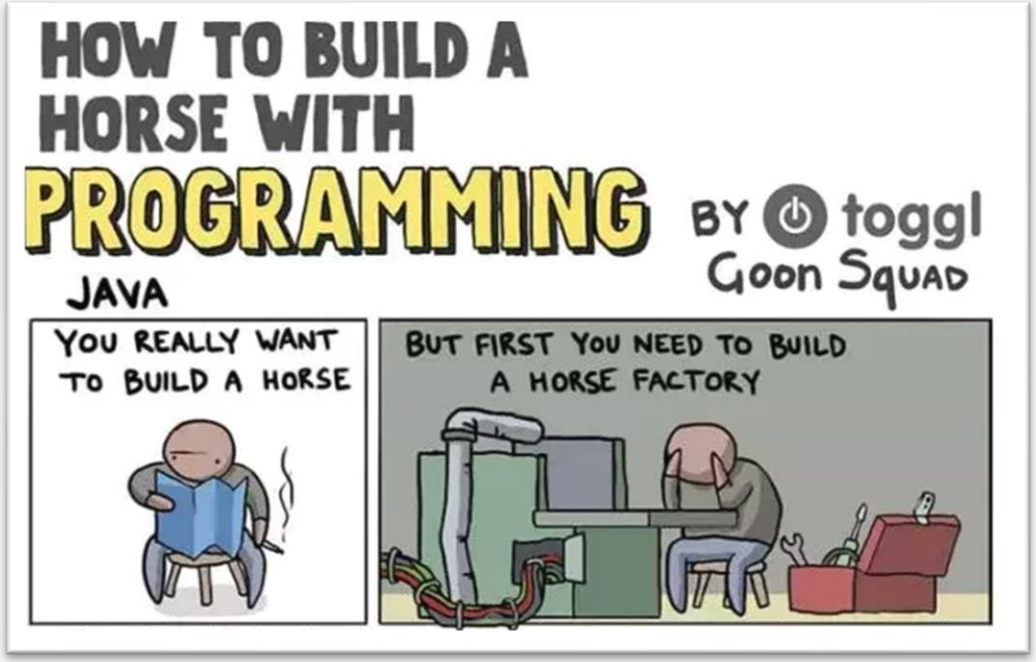
Complicado demais, onde está o botão de desligar que eu não acho?

Muito melhor e muito mais fácil de achar tudo que eu preciso. Todo o resto é só usar o comando de voz agora.

Factory Method

Propósito: Define uma interface para criar um objeto, mas deixa as subclasses decidirem qual classe a ser instanciada, postergando a instanciação.

Padrão **criacional** que adia a decisão sobre qual **classe** concreta instancia



Factory Method

Um usuário pode criar novos objetos sem ter que saber os detalhes de como eles são criados, ou quais são suas dependências. O usuário só precisa fornecer o atributos (parâmetros) que deseja.

Exemplo:

Quando uma empresa precisa contratar funcionários, ela não quer fazer isso, a empresa delega essa ação a uma agência de contratação.

Quem é responsável pela forma como o funcionário é contratado é a agência, ou seja “Construtor Virtual”.

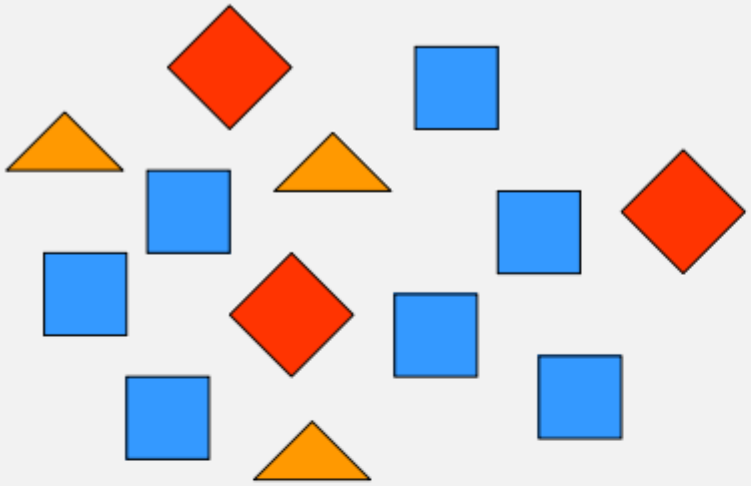
Flyweight

Propósito: Presente no gerenciamento de vários objetos que possuem informações repetidas e que devem ser manipulados em memória

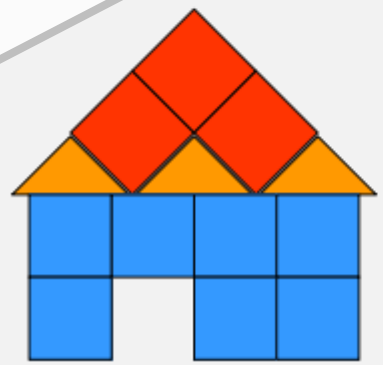
Padrão **estrutural** que centraliza a responsabilidade em **objetos** compartilhados de alta granularidade

Exemplo: Em um editor de texto, cada caractere tem atributos como tamanho, fonte, cor, etc. Guardar essa informação de cada caractere excederia a memória RAM em um texto grande.
Como várias informações são repetidas é só fazer o objeto guardar referência para outro objeto com suas respectivas propriedades (usando flyweight).

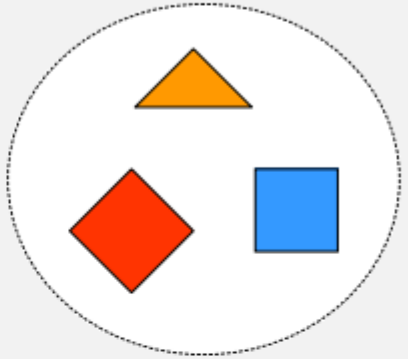
Flyweight



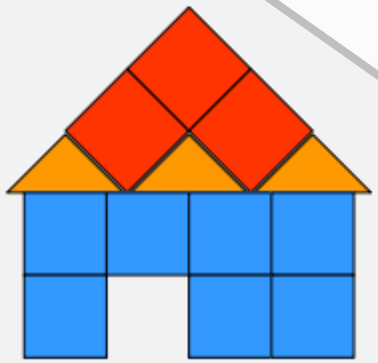
Um, dois, três, quatro... Ih, não vou ter memória para tudo isso.



Pool de objetos imutáveis compartilhados



Três objetos, tenho memória para instanciar todos eles.



Interpreter

Propósito: Dada uma linguagem, definir uma forma gramatical junto com um interpretador que usa a forma para interpretar sentenças na linguagem

Padrão **comportamental** usa uma **classe** para representar cada regra de uma gramática

Exemplo:

Há uma aplicação que necessita ler dados de diferentes tipos de arquivos, como XML, JSON e CSV.

Então surge o intérprete que visa converte a expressão "linguística" em uma estrutura de dados que pode ser manipulada e avaliada pela aplicação.

Iterator

Propósito: Fornece uma maneira de acessar sequencialmente os elementos de uma agregação de objetos sem expor sua representação subjacente

Padrão **comportamental** voltado ao acesso de **objetos** em uma **coleção de objetos**

Arranjos

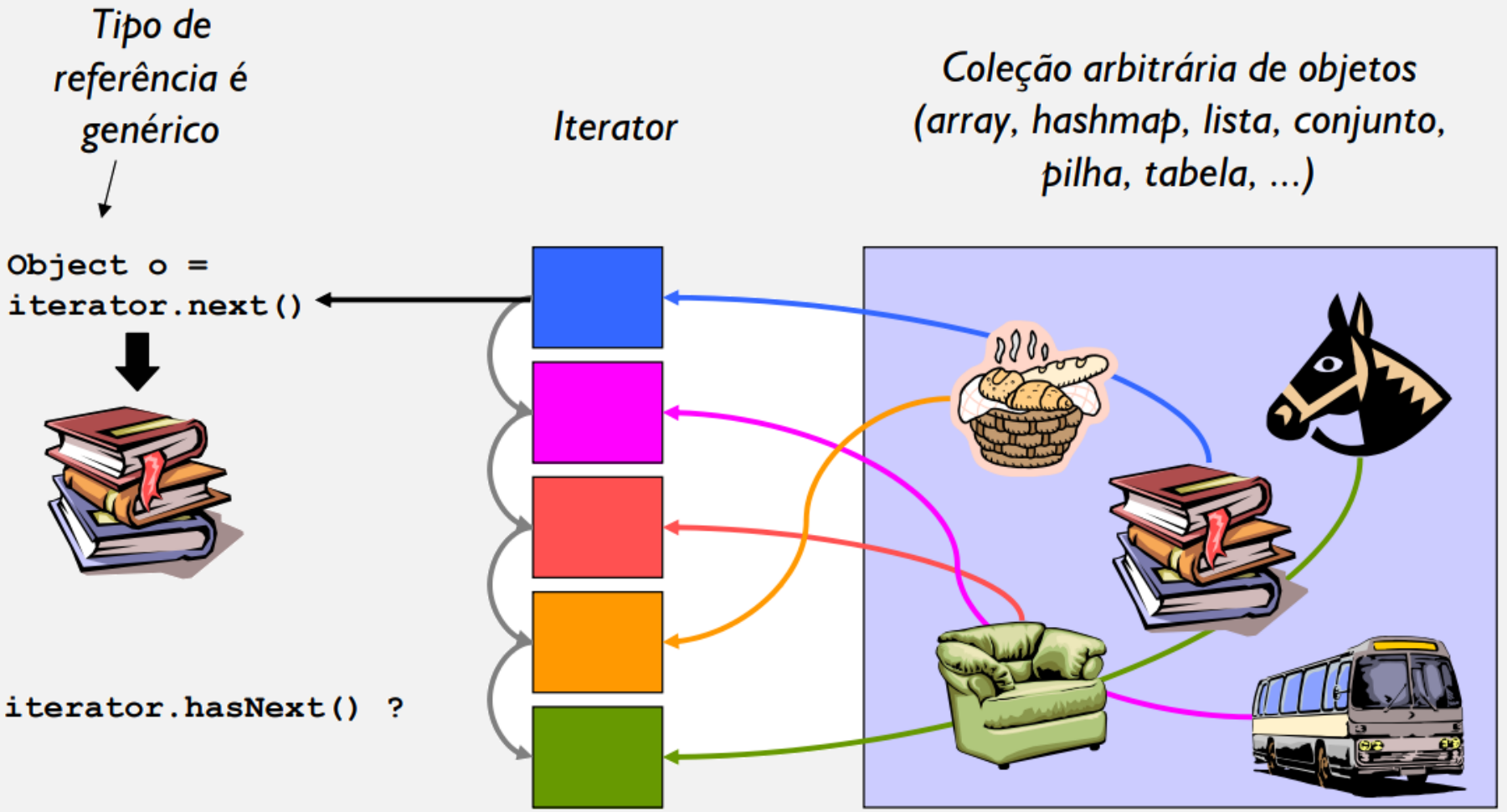
for

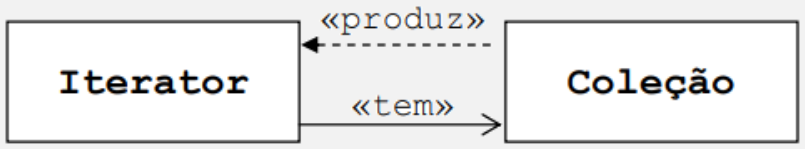
Estrutura de repetição que executa um determinando bloco de código com base nos **limites** e nas **iterações**

forEach

Estrutura de repetição que percorre todos os **elementos** de um determinado **arranjo** ou **coleção**

Iterator


Tipo de referência é genérico
Iterator
Coleção arbitrária de objetos (array, hashmap, lista, conjunto, pilha, tabela, ...)
Object o = iterator.next()
iterator.hasNext() ?


«produz»
Iterator
«tem»
Coleção

Mediator

Propósito: Definir um objeto que encapsula a forma como um conjunto de objetos interage.

Padrão **comportamental** responsável pela comunicação entre os **objetos**

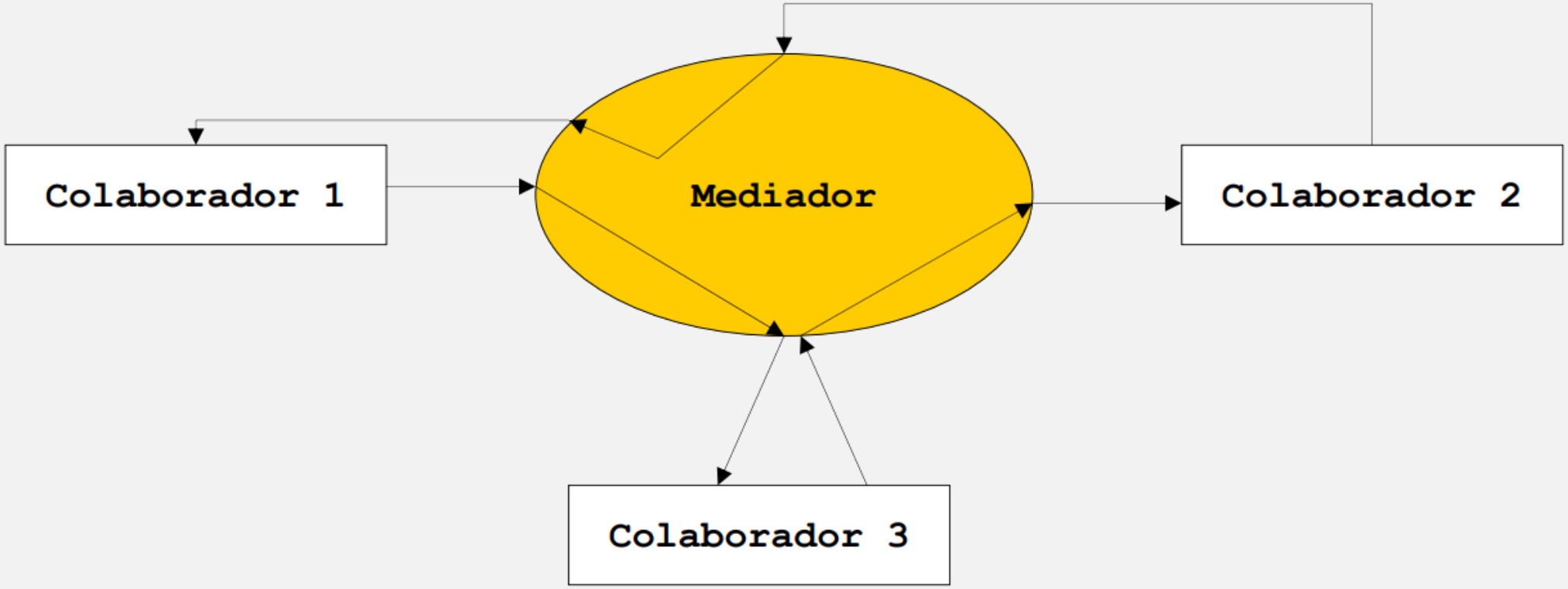
Exemplo:

Dois aviões são incapazes de comunicar entre si.

A torre de controle é responsável por gerenciar as informações e mensagens de todos os aviões e comunicar o que for estritamente necessário.

Mediator

Objetos podem se comunicar sem se conhecer, promovendo assim o acoplamento fraco



Memento

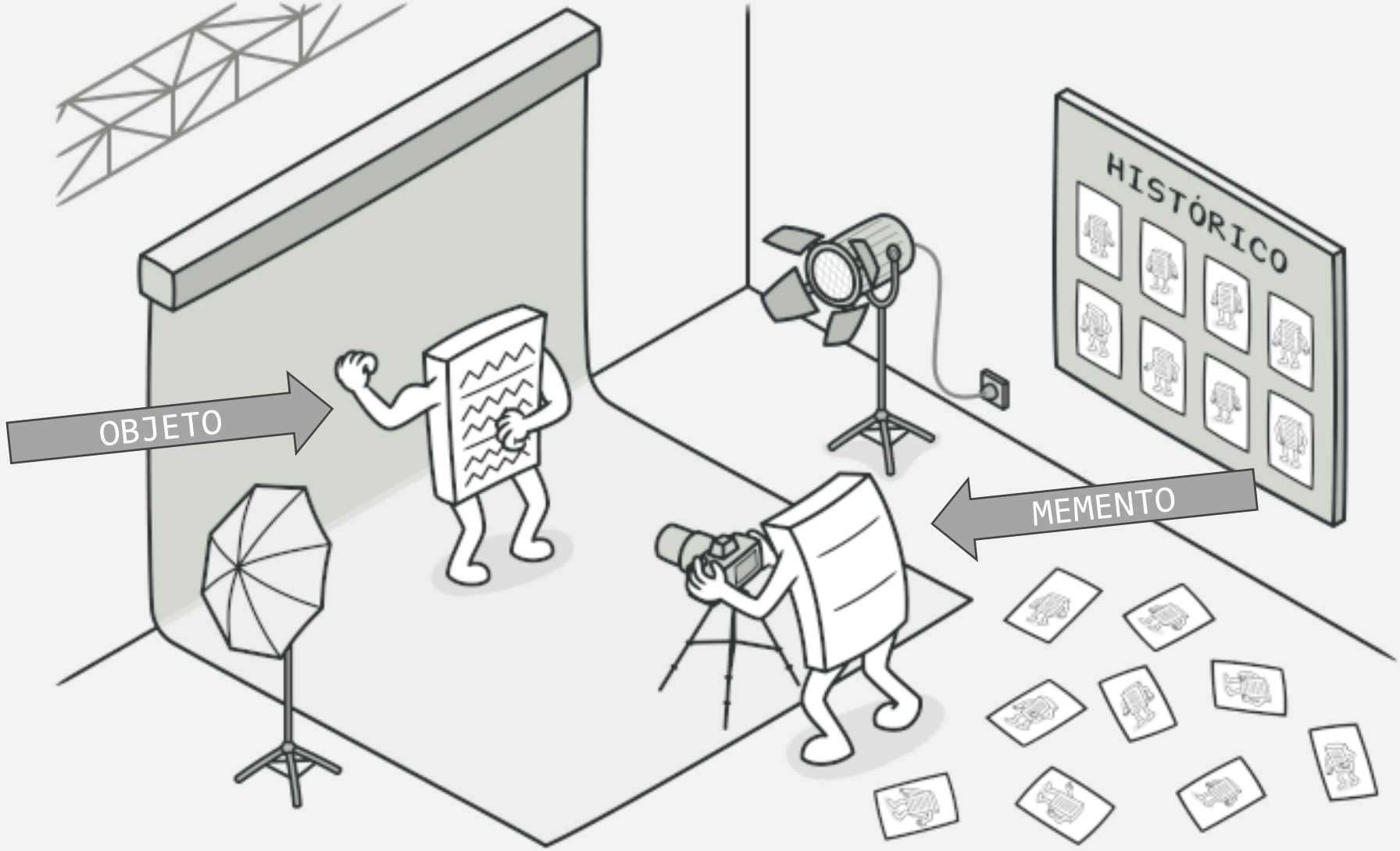
Propósito: Captura e externaliza um estado interno de um objeto, de modo que o mesmo possa posteriormente ser restaurado para este estado

Padrão **comportamental** responsável por guardar e restaurar o estado de **objetos**

Exemplo: Eu quero que todos os objetos que eu instanciei possam voltar a estados anteriores (CTRL + Z).

Com o memento é criado um pequeno repositório para guardar estado dos objetos.

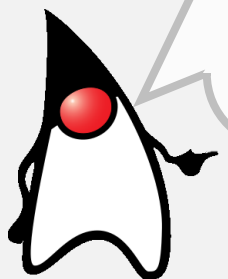
Memento



Observer

Propósito: Realiza a observação de objetos e quando um objeto muda de estado, todos os seus dependentes são notificados e atualizados

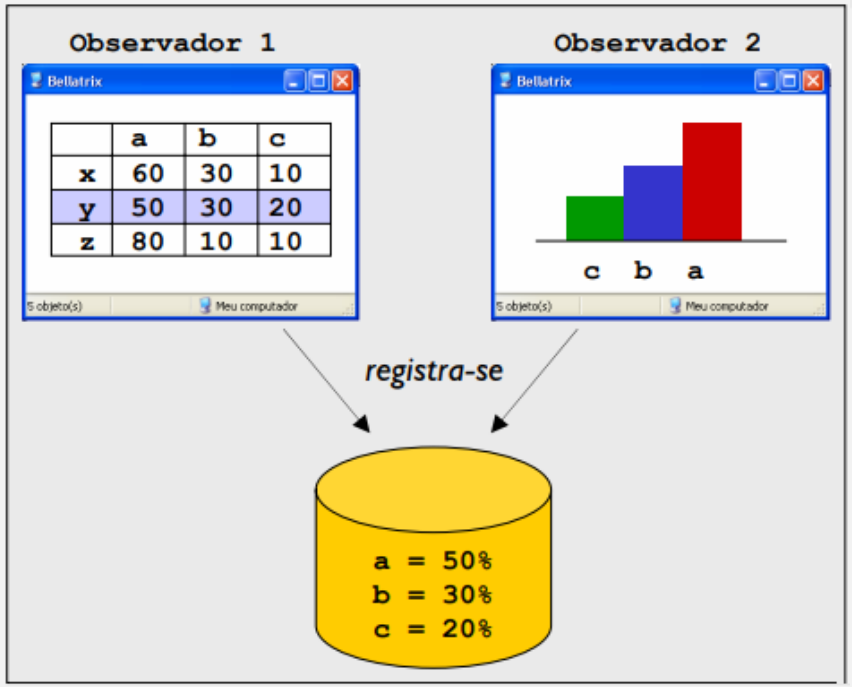
Padrão **comportamental** responsável por observar e avisar sobre mudanças no estado de um **objetos**

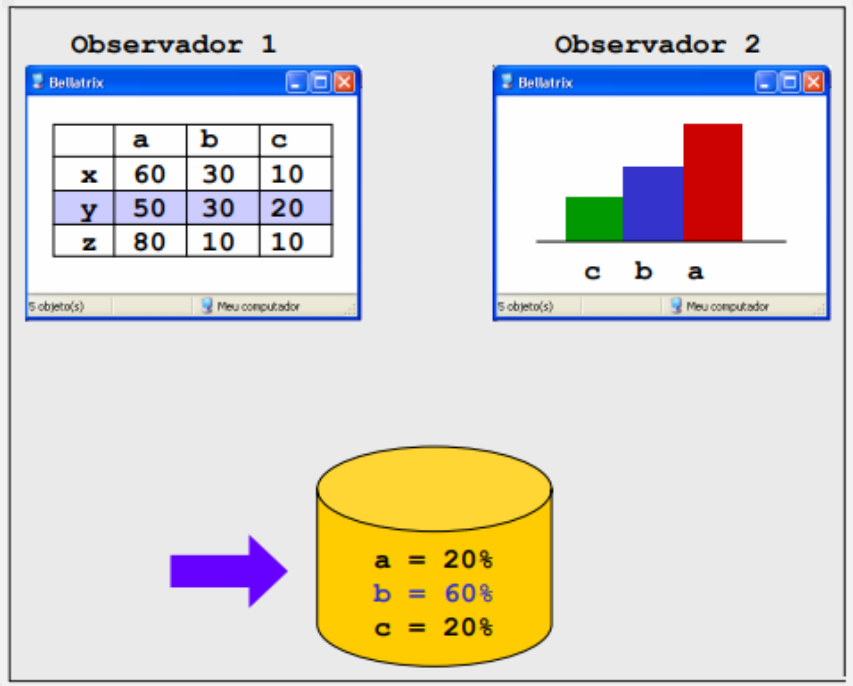


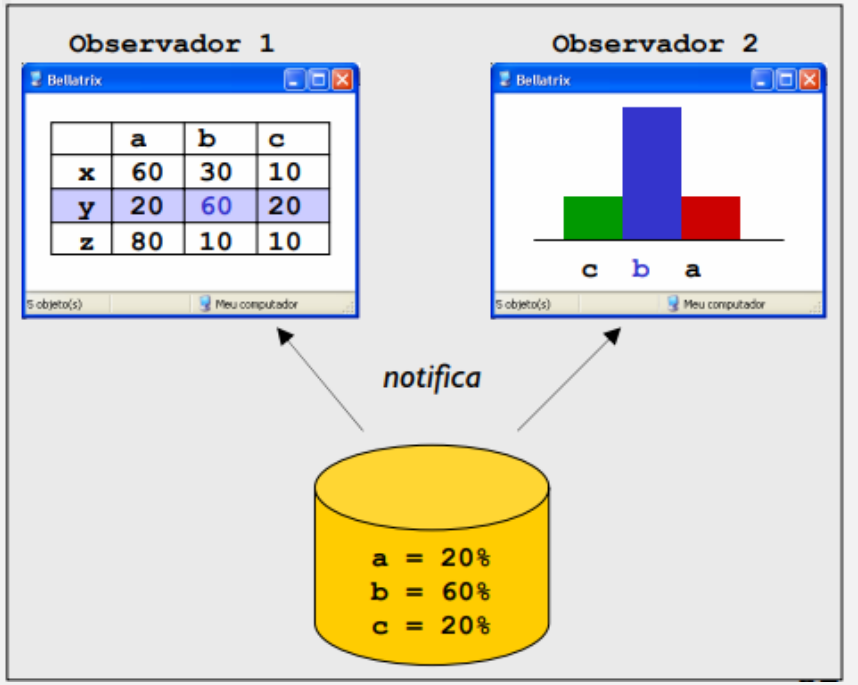
O **observer** é quase a mesma coisa dos **eventos** utilizados em aplicações gráficas como os botões de uma calculadora.

Tô de olho, só observando....

Observer







Porém o **observer** costuma mais ser utilizado entre objetos que se "inscreveram" para receber as notificações de outro objeto

Ó lá, mudou que eu vi!

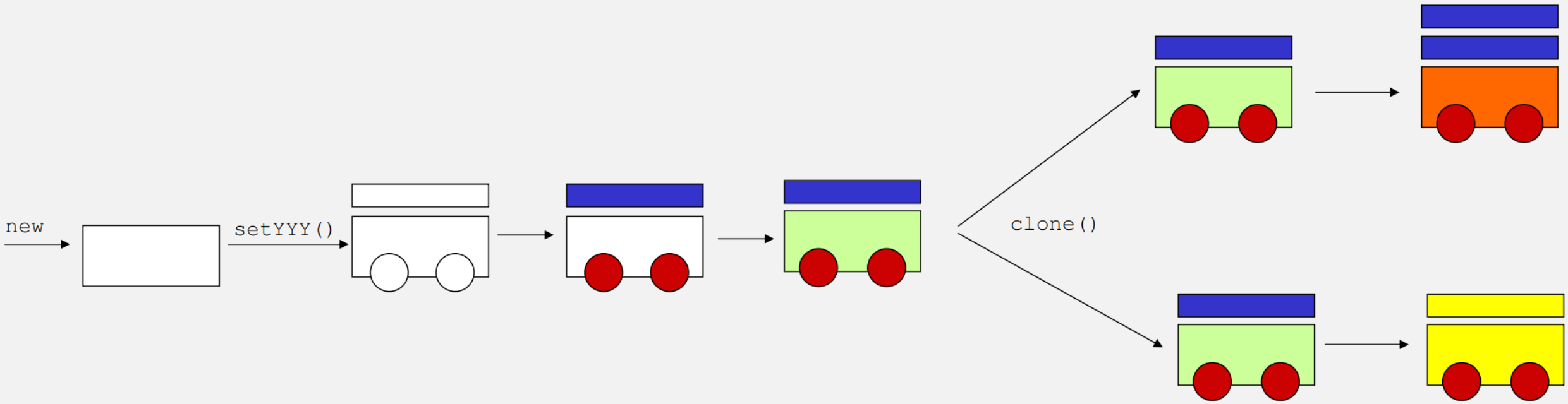
Prototype

Propósito: Especifica tipos de objetos a serem criados usando uma instância prototípica e criar novos objetos copiando esse protótipo

Padrão **criacional** responsável instanciar novos **objetos** a partir de estados já existente de outros objetos

Exemplo: Em um programa um carro foi instanciado e passou por várias alterações nos seus atributos.
Se você precisar de outro carro igual, não precisará ter todo esse trabalho, basta clonar ele.

Prototype



Caramba. Você é preguiçoso mesmo.

Preguiça não! Reuso!
É diferente.

Proxy

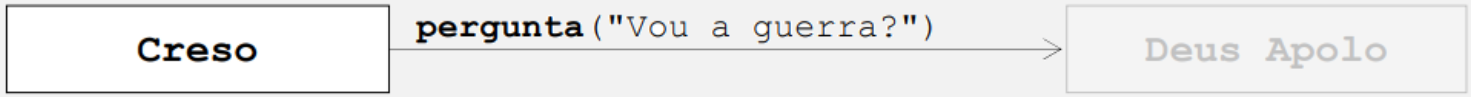
Propósito: Prover um substituto ou ponto através do qual um objeto possa controlar o acesso a outro

Padrão **estrutural** que assume a responsabilidade de outro **objeto**

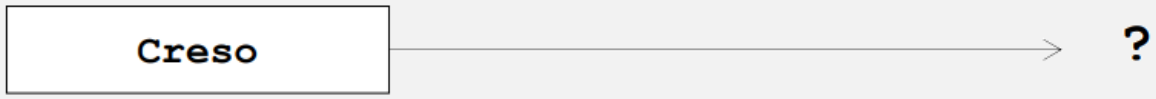
Exemplo: Usuário quer acessar as informações de um objeto, mas esse objeto está indisponível.
A solução é utilizar um intermediário (proxy) que contém uma referência para o objeto e repassa requisições

Proxy

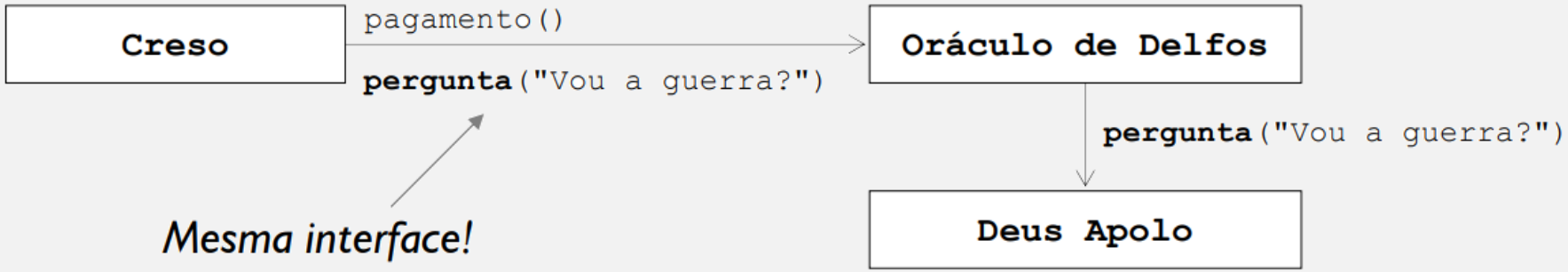
Sistema quer utilizar objeto real...



Mas ele não está disponível (remoto, inaccessível, ...)



*Solução: arranjar um **intermediário** que saiba se comunicar com ele eficientemente*



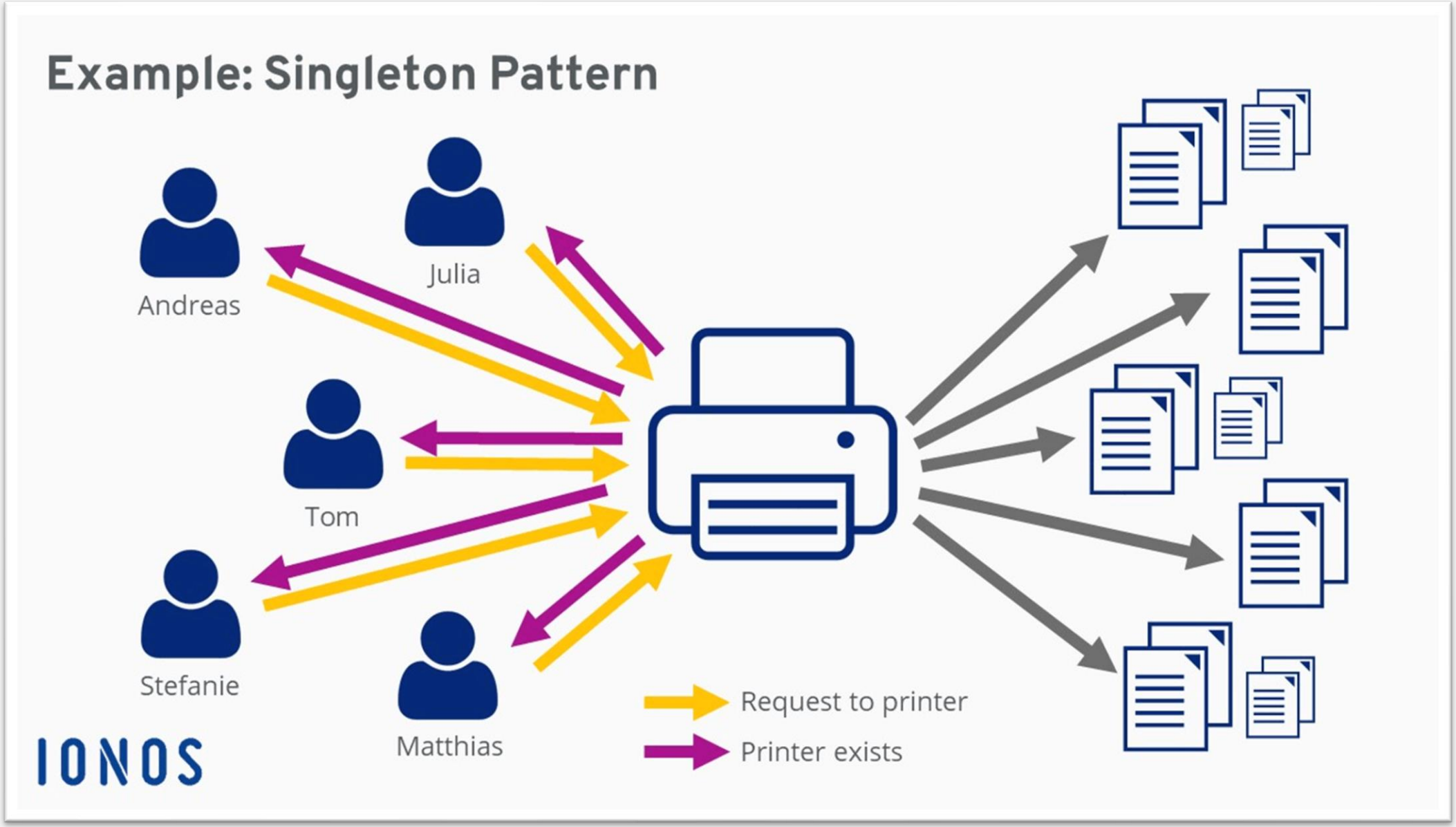
Singleton

Propósito: Garante que uma classe tenha somente uma instância e fornece um ponto global de acesso para ela

Padrão **criacional** para a criação de um único **objeto**

Exemplo: Alguns programas necessitam que algumas classes tenham apenas uma instância.
O singleton garantirá que nenhuma outra instância possa ser criada.

Singleton


Example: Singleton Pattern
Andreas
Julia
Tom
Stefanie
Matthias
Request to printer
Printer exists
IONOS

State

Propósito: Permite que um objeto altere seu comportamento quando seu estado interno muda. O objeto parecerá ter mudado de classe

Padrão **comportamental** que permite alteração de estado de um **objeto**

Exemplo: Em uma empresa de carros, o carro pode assumir inúmeros estados: alugado, disponível, batido, etc.
Ao invés de trabalhar com vários “if” verificando o estado do carro, é possível utilizar o state para isso.

Strategy

Propósito: Separa a implementação de algoritmos das classes que o usam

Padrão **comportamental** permite definir novas operações sem alterar as classes dos opera **objetos** sobre os quais opera

Exemplo: Seu programa faz a ordenação de elementos utilizando o método BubbleSort. Mas o usuário quer poder alterar o método.

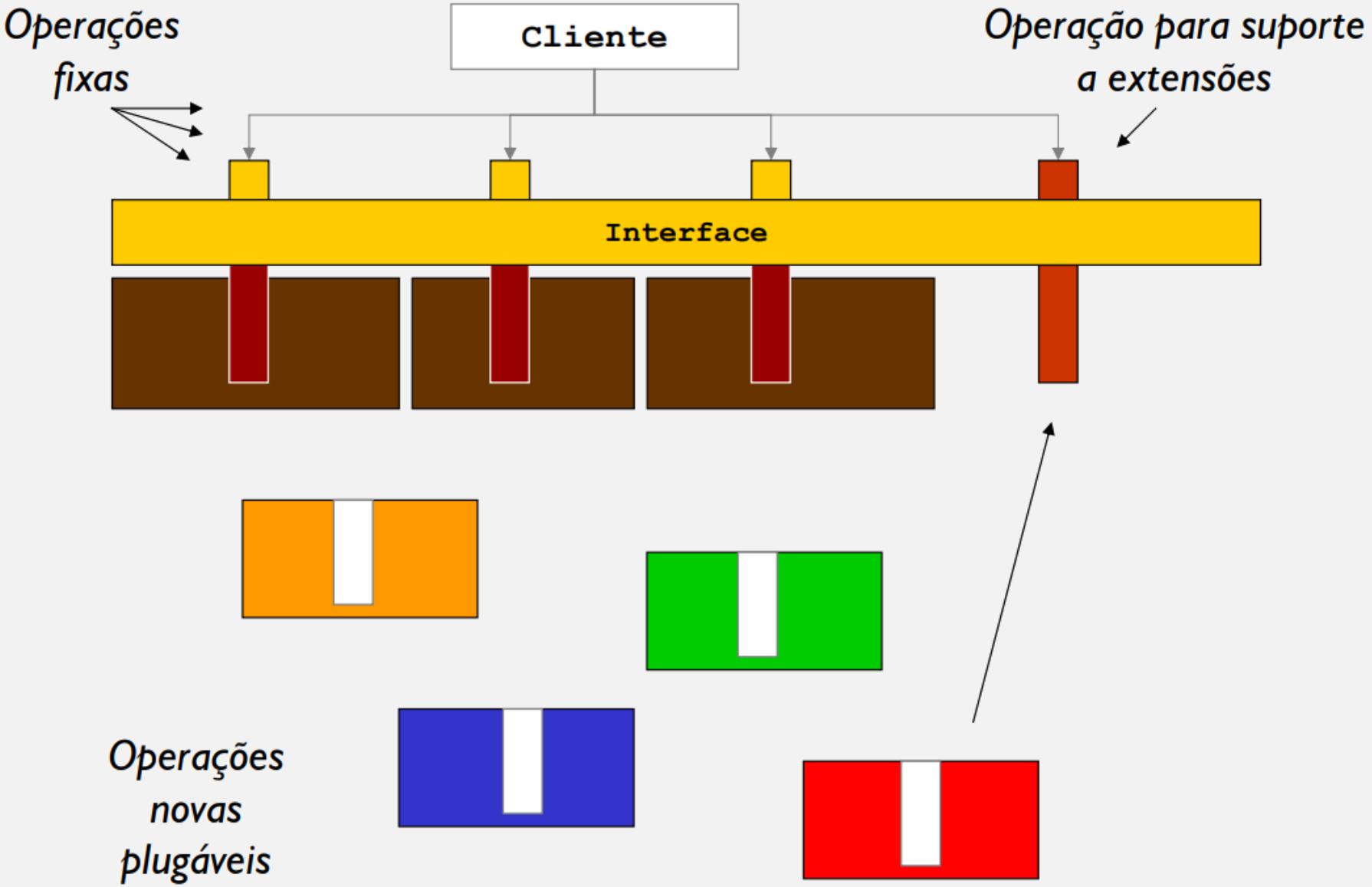
Com o strategy é possível o usuário informar o tipo de algoritmo e a forma como deseja ordenar seus dados.

Visitor

Propósito: Permite definir uma nova operação sem mudar as classes dos elementos sobre os quais opera

Padrão **comportamental** que permite adição de novas funcionalidades a um **objeto** sem a necessidade de modifica-lo.

Visitor


Operações
fixas
Cliente
Operação para suporte
a extensões
Interface
Operações
novas
plugáveis

Quando não Usar Padrões de Projeto

Em muitos sistemas observa-se um uso exagerado de padrões de projeto, em situações nas quais os ganhos de flexibilidade e extensibilidade são questionáveis.

VALENTE, Marco. Engenharia de Software Moderna.

Existe até um termo para se referir a essa situação: **paternite**, isto é, uma "inflamação" associada ao uso precipitado de padrões de projeto.

John Ousterhout tem um comentário relacionado a essa "doença":

> "
>
> *"O maior risco de padrões de projetos é a sua super-aplicação (over-application). Nem todo problema precisa ser resolvido por meio dos padrões de projeto; por isso, não tente forçar um problema a caber em um padrão de projeto quando uma abordagem tradicional funcionar melhor. O uso de padrões*

Referências

SCHILDT, Herbert. **Java para iniciantes: crie, compile e execute**. 6. ed. Porto Alegre: Bookman, 2015. Disponível em: <https://app.minhabiblioteca.com.br/reader/books/9788582603376>.

SILVA, Fabricio; LEITE, Márcia; OLIVEIRA, Diego. **Paradigmas de Programação**. Porto Alegre: SAGAH, 2019. Disponível em: <https://app.minhabiblioteca.com.br/reader/books/9788533500426>.

DEITEL, Paul. **Java: como programar**. 4. ed. Porto Alegre: Bookman, 2003.

DEITEL, Paul. **Java: como programar**. 10. ed. São Paulo: Pearson Education do Brasil, 2017.

PADRÕES DE PROJETOS

Alexandre Mendonça Fava
alexandre.fava@udesc.br

Universidade do Estado de Santa Catarina – UDESC
Programa de Pós-graduação em Computação Aplicada – PPGCA

Exercícios

(FGV, 2024) Assinale a opção que indica o padrão responsável pela criação de objetos de várias classes graças à especialização de sua classe principal.

a) Abstract Factory

b) Decorator

c) Strategy

d) Builder

e) Adapter

Exercícios

(UNEMAT, 2024) Uma equipe de sistemas de uma empresa recebeu uma demanda para incluir, em um sistema, um tratamento que permita observar se um determinado conjunto de dados está sendo atualizado. Caso haja alguma atualização, um conjunto de assinantes que desejem ser informados sobre alguma mudança nesses dados deverá ser notificado. Para atender à demanda descrita, qual padrão de projeto pode ser usado?

a) Memento

b) Observer

c) Strategy

d) Abstract Factory

e) Template Method

Exercícios

(UNEMAT, 2024) J participa de uma equipe de desenvolvimento e recebeu uma demanda de codificar uma estrutura de árvore no sistema em que está trabalhando. O padrão de projeto estrutural que poderá ajudar nessa tarefa de implementação é o

a) Visitor

b) Façade

c) Singleton

d) Composite

e) Factory Method

Exercícios

(UFG, 2024) Os padrões de projeto, também conhecidos como Design Patterns, referem-se a soluções reutilizáveis para problemas comuns no design de software. Eles representam as melhores práticas usadas por desenvolvedores experientes para resolver problemas específicos de design durante o desenvolvimento de software. Os padrões de projeto observer, factory method e proxy, são classificados, respectivamente, como padrões de projeto do tipo

a) comportamental, criacional e estrutural.

b) criacional, estrutural e comportamental.

c) criacional, comportamental e estrutural.

d) estrutural, comportamental e criacional.