

# Padrões GoF – Strategy, Observer, Singleton, Abstract Factory e outros...



---

Análise e Projeto Orientados a Objetos

Profa Dra Rosana T. V. Braga



# Mais Padrões GoF

---

- Strategy
- Abstract Factory
  
- OUTROS



# Padrão de Projeto: Estratégia

---

- utilizado quando:
  - várias classes relacionadas diferem apenas no comportamento ou
  - são necessárias diversas versões de um algoritmo ou
  - as aplicações-cliente não precisam saber detalhes específicos de estruturas de dados de cada algoritmo.



# Padrão de Projeto: Estratégia

---

- ***Problema***

- Como permitir que diferentes algoritmos alternativos sejam implementados e usados em tempo de execução?

- ***Forças***

- O fato de um algoritmo diferente poder ser selecionado para realizar uma determinada tarefa, dependendo da aplicação-cliente, pode ser solucionado com uma estrutura case. Mas isso leva a projetos difíceis de manter e código redundante.
- Usar herança é uma alternativa, mas também tem seus problemas: várias classes relacionadas são criadas, cuja única diferença é o algoritmo que empregam.



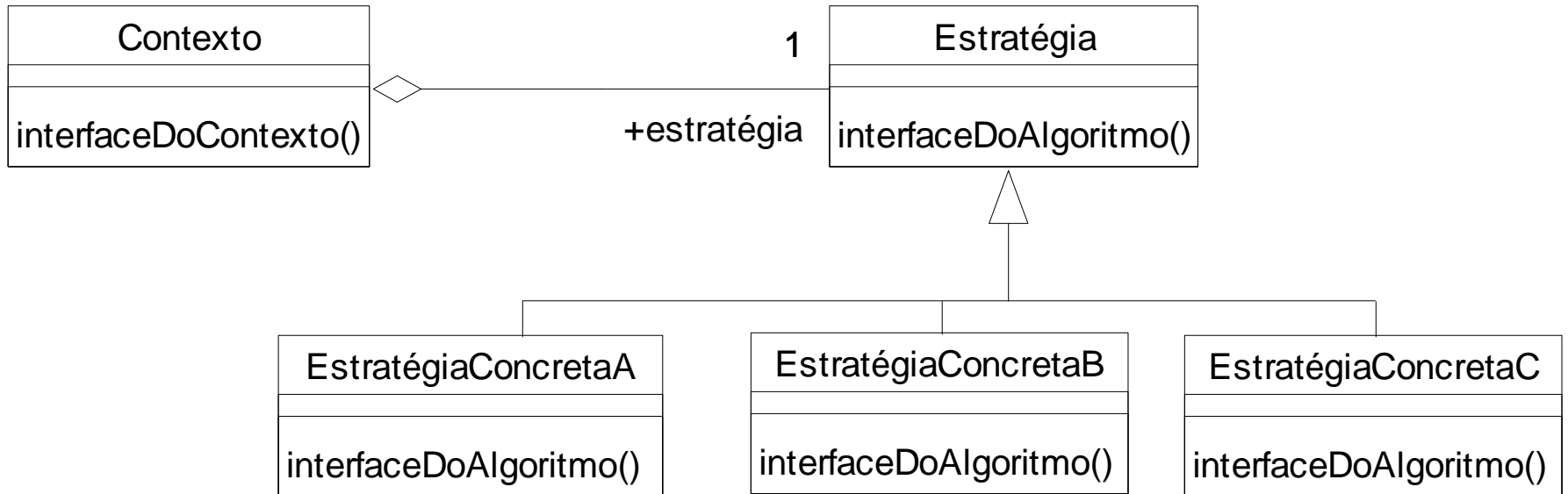
# Padrão de Projeto: Estratégia

---

## ■ *Solução*

- Criar uma classe abstrata para a Estratégia empregada pelo algoritmo, bem como subclasses especializando cada um dos algoritmos.
- O Contexto mantém uma referência para o objeto Estratégia e pode definir uma interface para permitir que a Estratégia acesse seus dados. A Estratégia define uma interface comum a todos os algoritmos disponíveis. O Contexto delega as solicitações recebidas das aplicações-cliente para sua estratégia.

# Padrão de Projeto: Estratégia



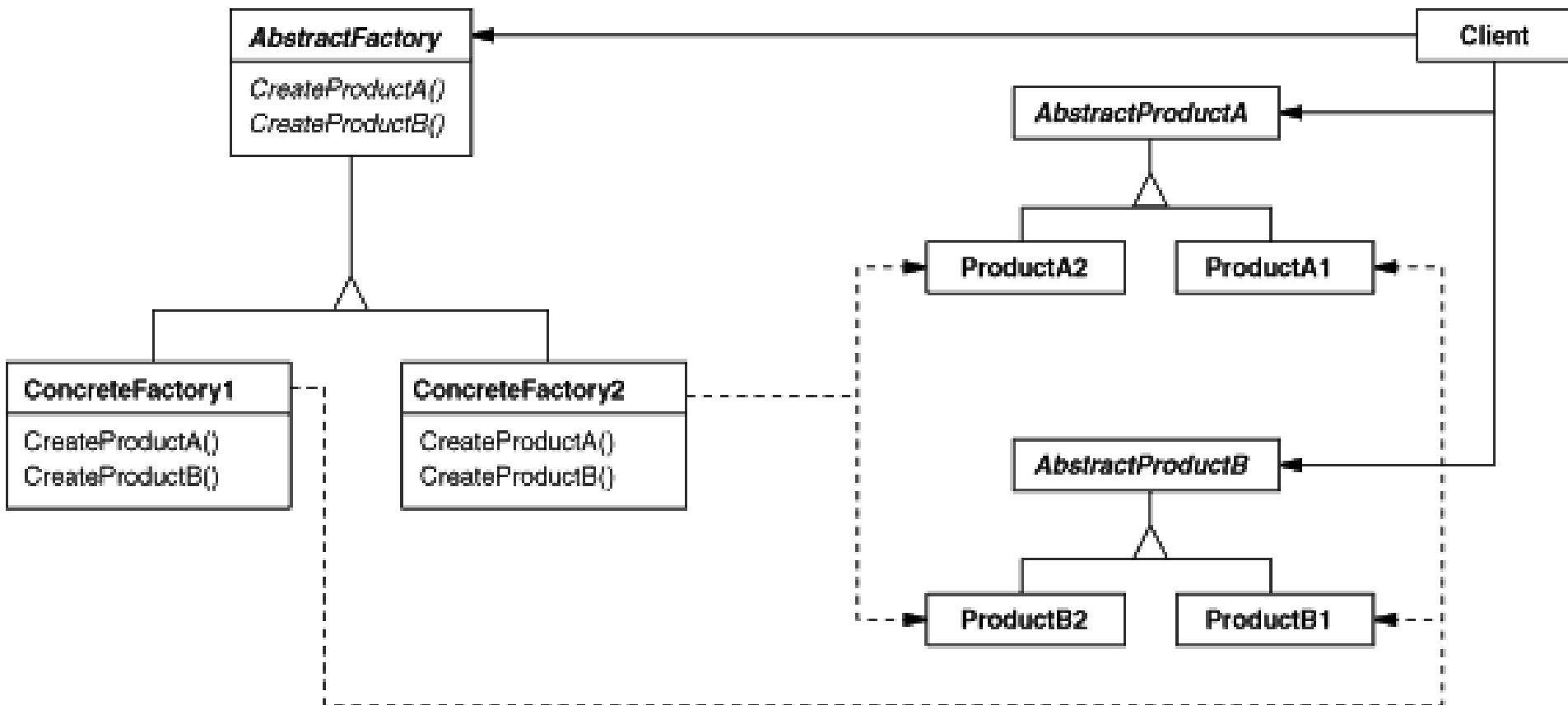


# Abstract Factory- Fábrica Abstrata

---

- **Intenção:** Fornece uma interface para criar famílias de objetos relacionados ou dependentes sem a necessidade de especificar suas classes concretas
- **Aplicabilidade:** Use o padrão Abstract Factory quando
  - Um sistema deveria ser independente de como seus produtos são criados, compostos ou representados.
  - Um sistema deve ser configurado com uma de múltiplas famílias de produtos.
  - Uma família de objetos de produtos relacionados é projetado para ser usado em conjunto.
  - É desejada uma biblioteca de classes de produtos, e deseja-se revelar apenas suas interfaces e não a implementação.

# Abstract Factory





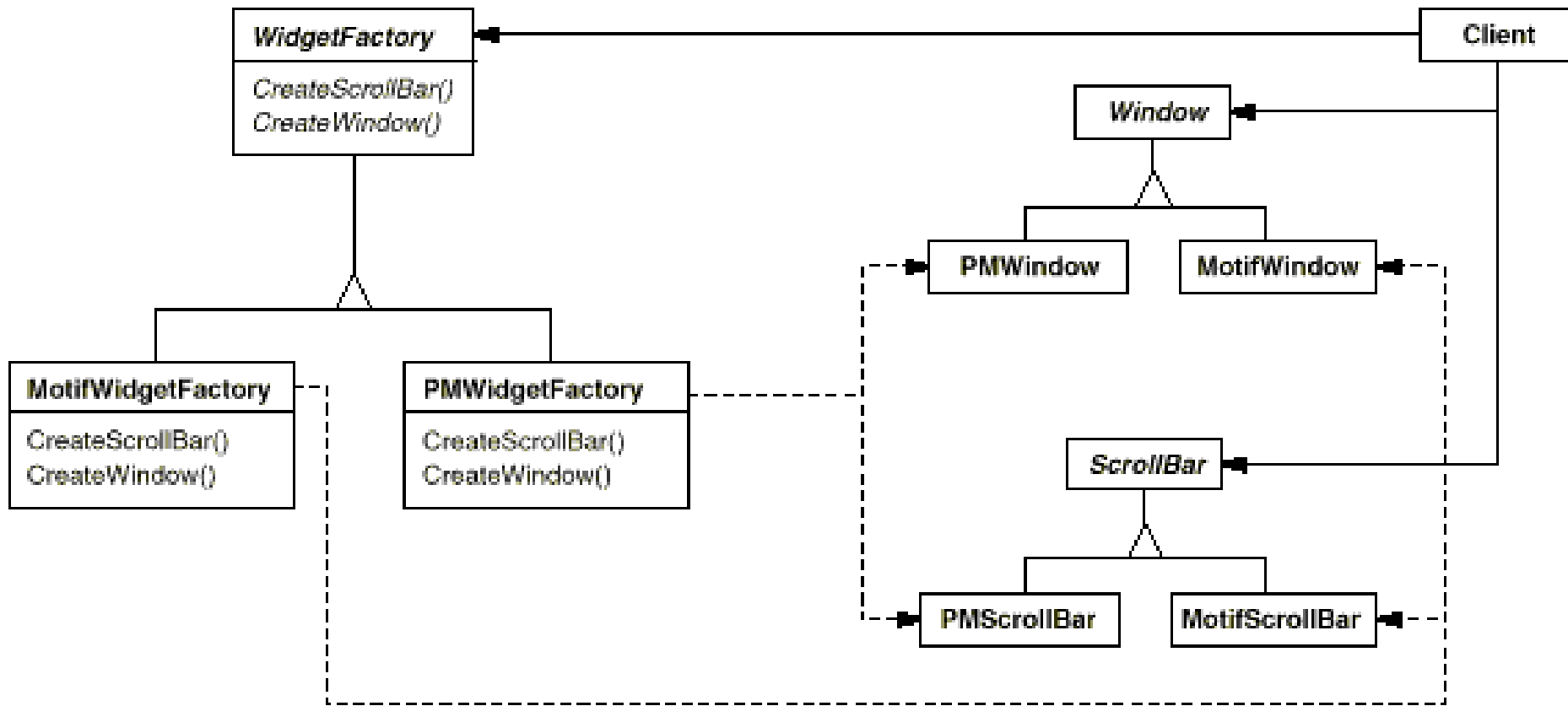
# Abstract Factory



---

- É uma fábrica de objetos que retorna uma das várias fábricas.
- Uma aplicação clássica do Abstract Factory é o caso onde o seu sistema precisa de suporte a múltiplos tipos de interfaces gráficas, como Windows, Motif ou MacIntosh.
- A fábrica abstrata retorna uma outra fábrica de GUI que retorna objetos relativos ao ambiente de janelas do SO desejado.

# Exemplo: Abstract Factory

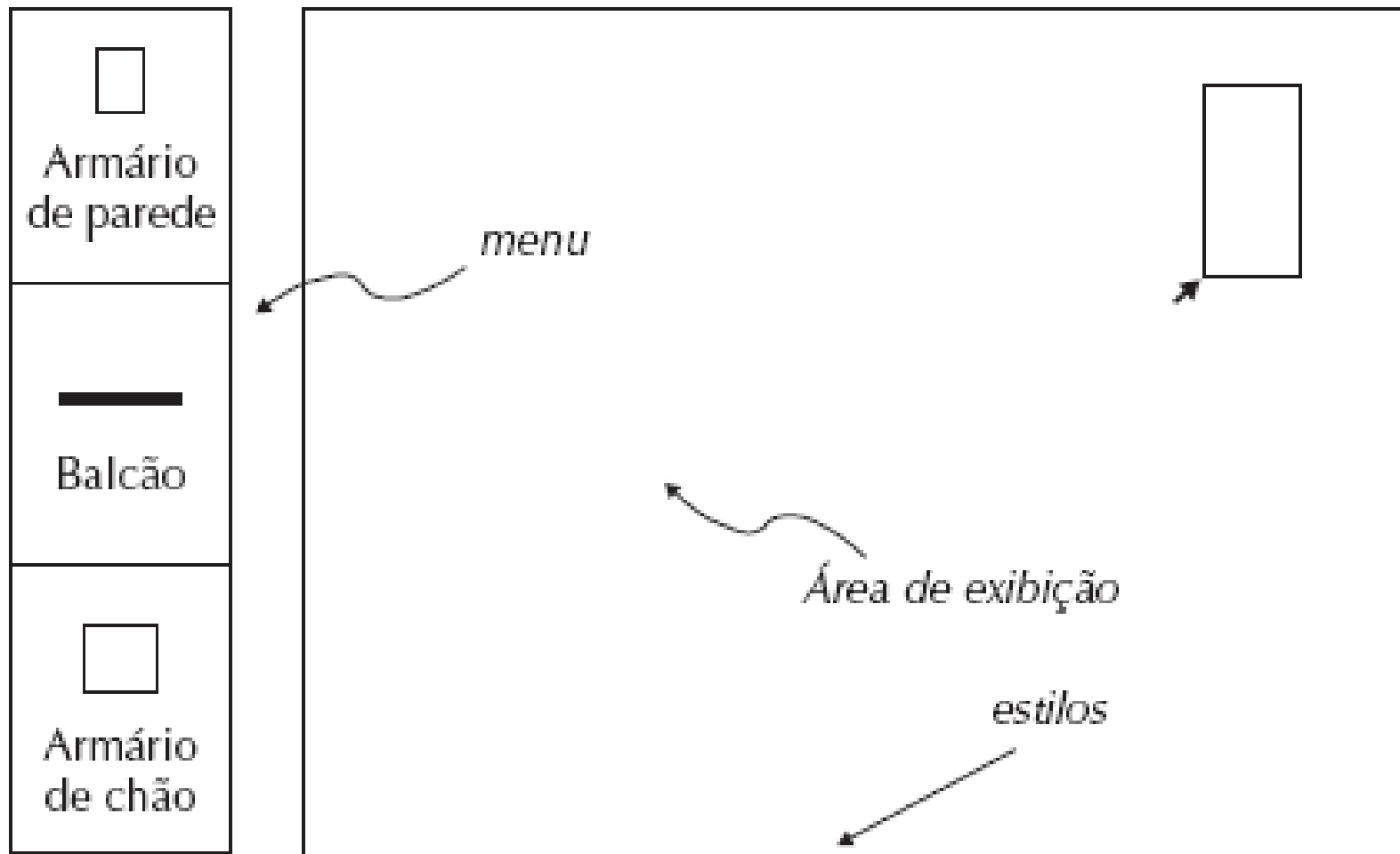




# Exemplo: Aplicação para construir armários de cozinha

---

- Proprietários de residências sempre têm a intenção de modernizar suas cozinhas, freqüentemente utilizando um software para visualizar as possibilidades.
- *VisualizadorDeCozinhas*: aplicação que permite que o usuário crie o layout das partes de uma cozinha, sem comprometer-se com um estilo.



Moderno

Clássico

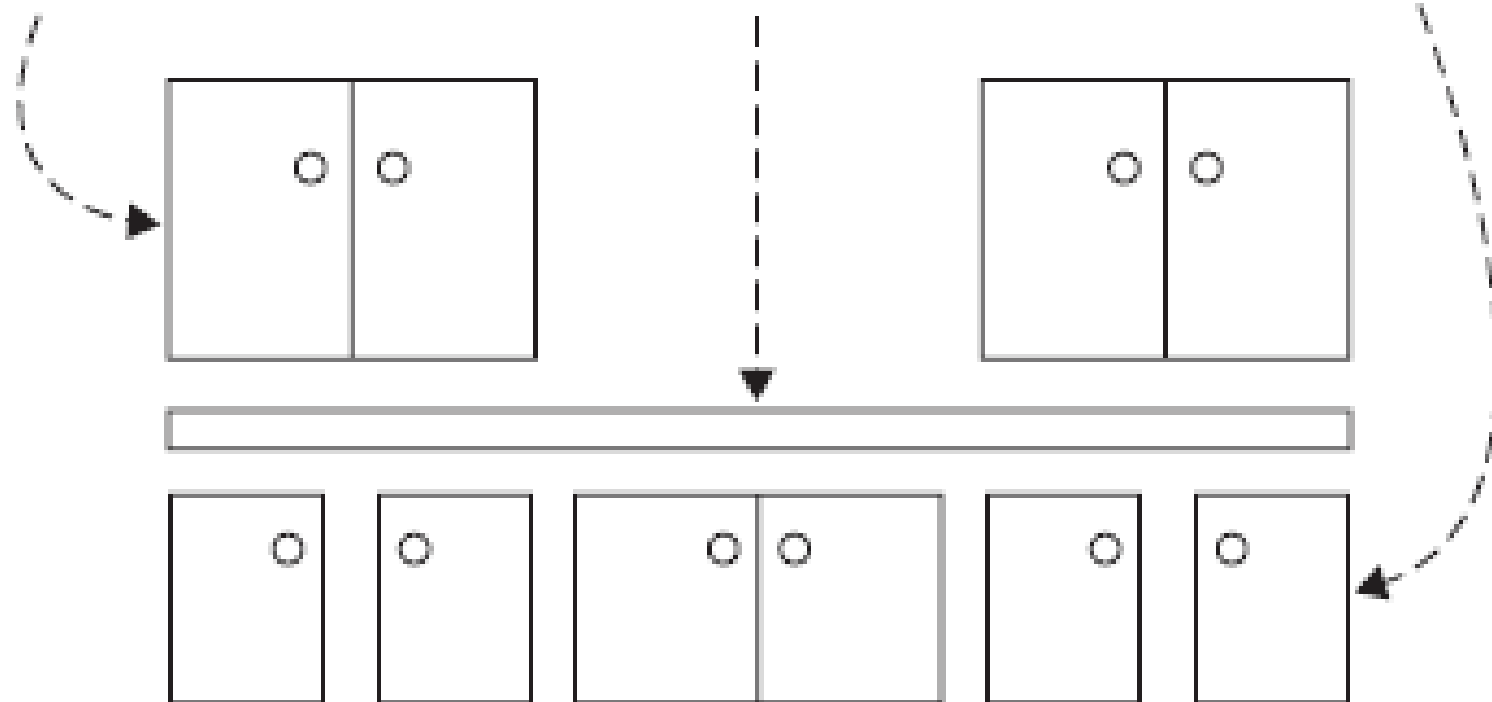
Antigo

Artesanal

*Armários de parede*

*Balcão*

*Armários de chão*

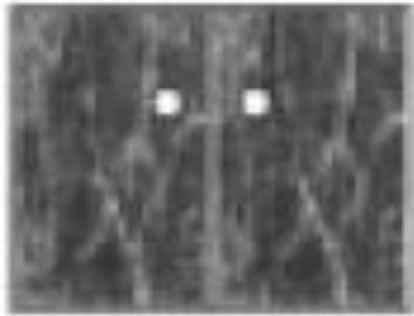


Moderno

Clássico

Antigo

Artesanal



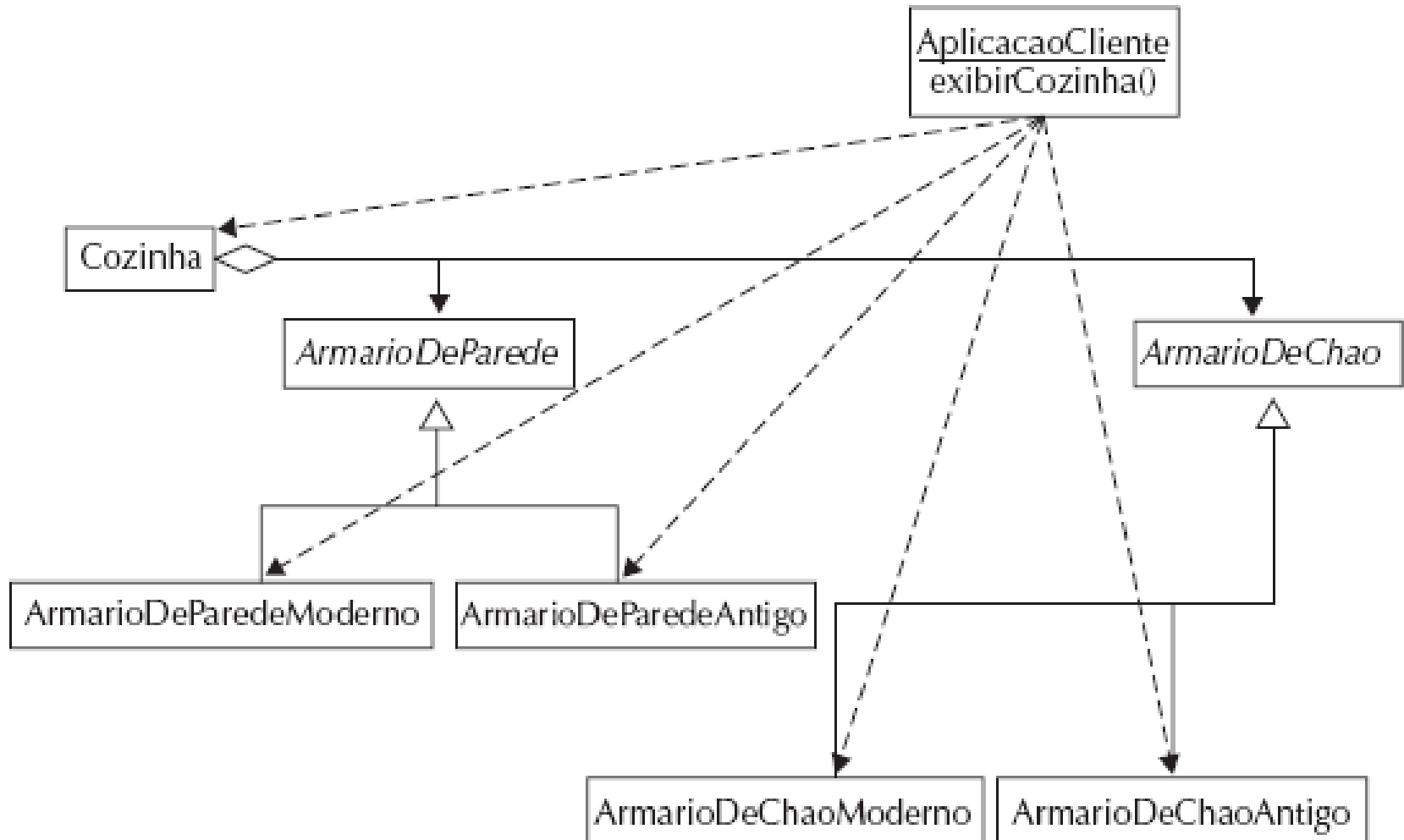
Moderno

Clássico

Antigo

Artesanal

# Versão sem padrões de projeto



```
// VERSÃO QUE IGNORA NOSSOS PROPÓSITOS DE PROJETO
```

```
// Determina o estilo
```

```
... // Instrução case?
```

```
// Assume que o estilo antigo foi selecionado.
```

```
// Cria os armários de parede com o estilo antigo
```

```
ArmarioDeParedeAntigo armarioDeParedeAntigo1 = new ArmarioDeParedeAntigo ();
```

```
ArmarioDeParedeAntigo armarioDeParedeAntigo2 = new ArmarioDeParedeAntigo ();
```

```
...
```

```
// Cria os armários de chão com o estilo antigo
```

```
ArmarioDeChaoAntigo armarioDeChaoAntigo1 = new ArmarioDeChaoAntigo ();
```

```
ArmarioDeChaoAntigo armarioDeChaoAntigo2 = new ArmarioDeChaoAntigo ();
```

```
...
```

```
// Cria o objeto cozinha, assumindo a existência de métodos adicionar()
```

```
Cozinha cozinhaAntiga = new Cozinha();
```

```
cozinhaAntiga.adicionar( armarioDeParedeAntigo1, ... ); // demais parâmetros  
especificam a localização
```

```
cozinhaAntiga.adicionar( armarioDeParedeAntigo2, ... );
```

```
...
```

```
cozinhaAntiga.adicionar( armarioDeChaoAntigo1, ... );
```

```
cozinhaAntiga.adicionar( armarioDeChaoAntigo2, ... );
```

```
...
```

```
// exhibe cozinhaAntiga
```



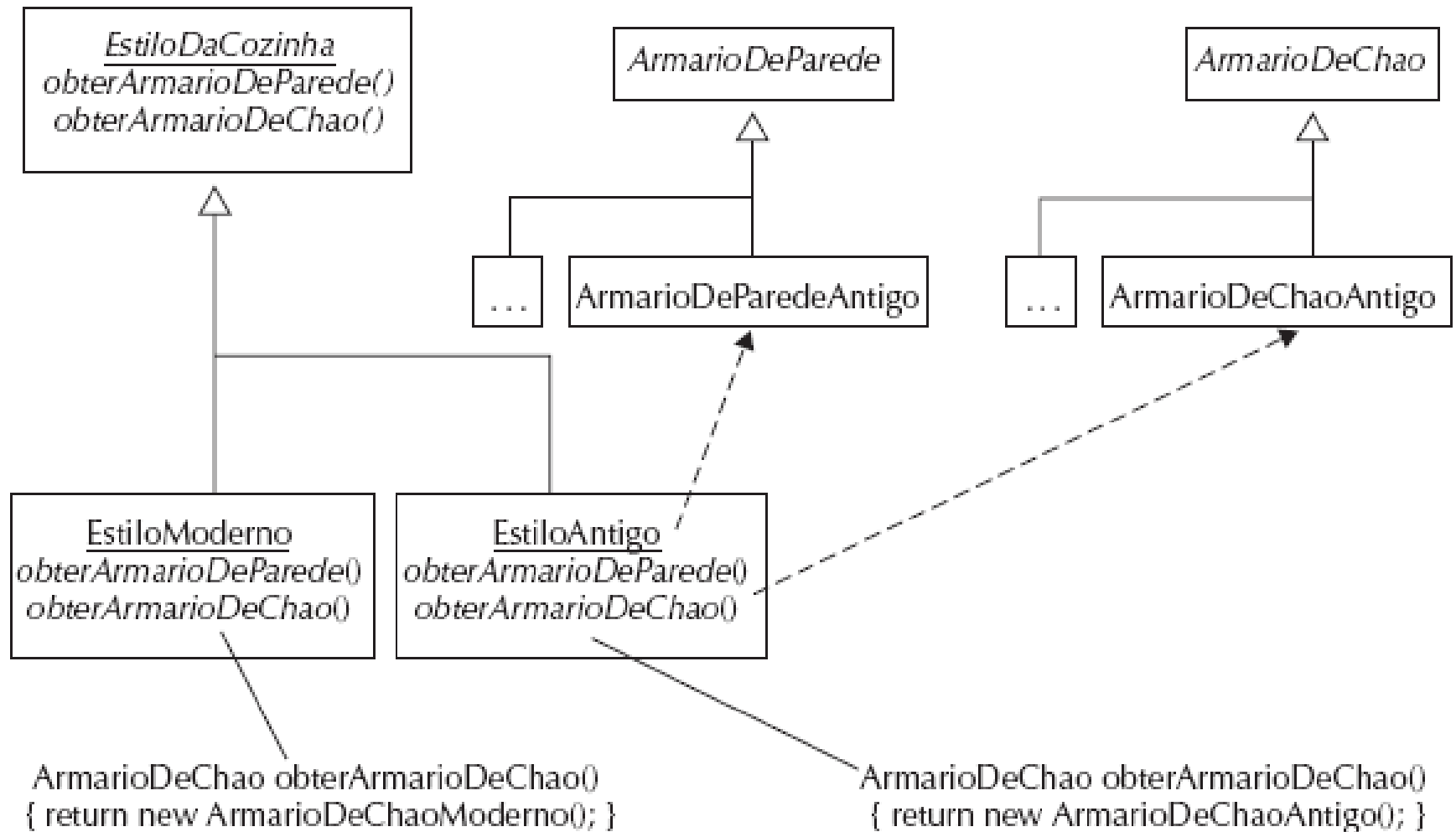


# Esboço da Solução

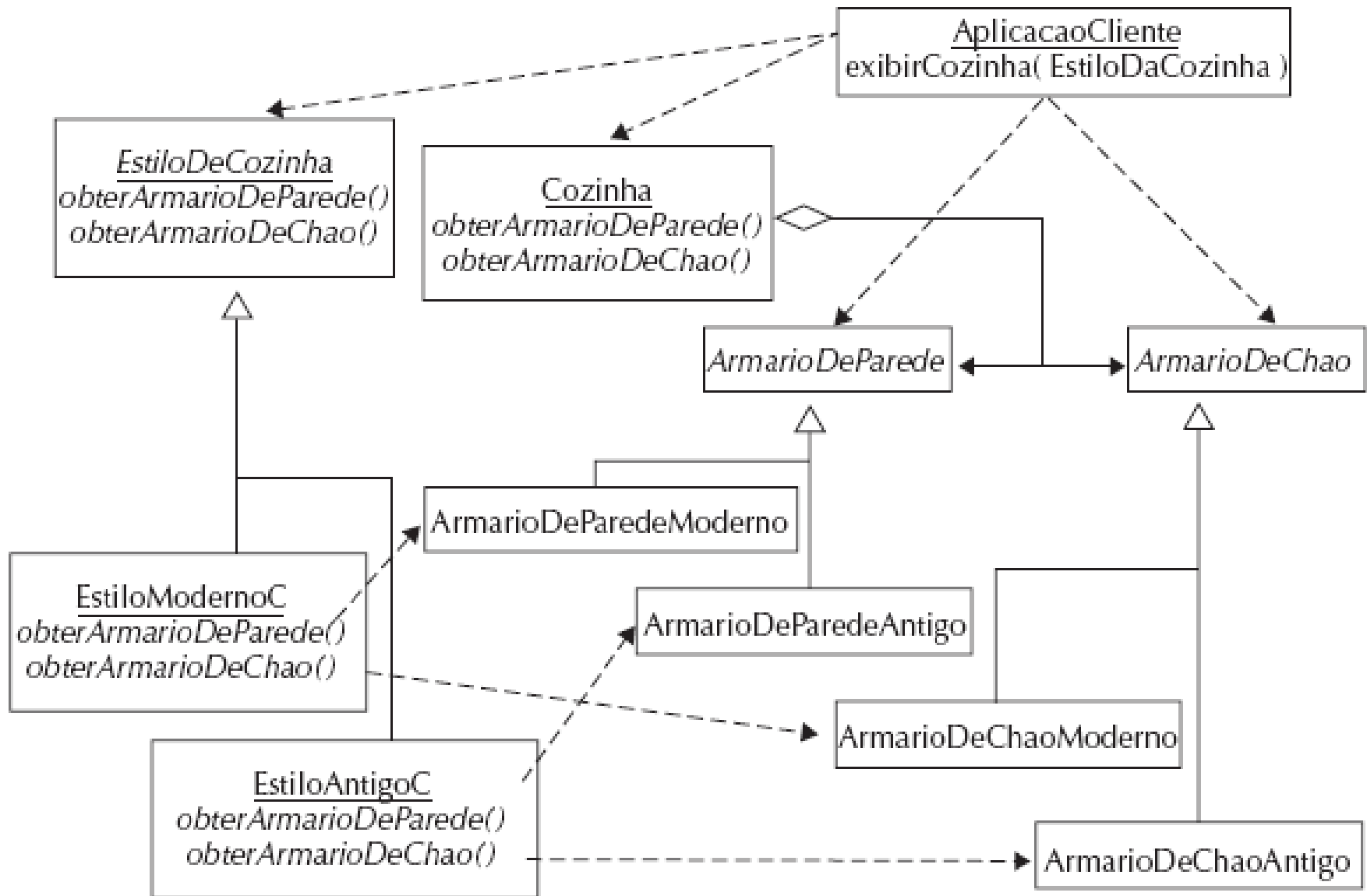
---

- Em vez de criar os objetos *ArmarioDeParedeAntigo*, *ArmarioDeChaoAntigo*, etc diretamente, criar uma versão parametrizada de *exibirCozinha()* que delega a criação desses objetos, substituindo frases como
  - . . . *new ArmarioDeParedeAntigo();*  
por versões delegadas a um parâmetro de estilo:
  - . . . *meuEstilo.obterArmarioDeParede();*
- Em tempo de execução, a classe de *meuEstilo* determina a versão de *obterArmarioDeParede()* executada, produzindo assim o tipo apropriado de armário de parede

# A idéia do Abstract Factory



# Versão usando Abstract Factory



```
// VERSÃO CONSIDERANDO OS PROPÓSITOS DE PROJETO
```

```
//Determina o estilo instanciando meuEstilo
```

```
EstiloAntigoC meuEstilo = new EstiloAntigoC;
```

```
// Cria os armários de parede: Tipo determinado pela classe de meuEstilo
```

```
ArmarioDeParede ArmarioDeParede1 = meuEstilo.obterArmarioDeParede();
```

```
ArmarioDeParede ArmarioDeParede2 = meuEstilo.obterArmarioDeParede();
```

```
...
```

```
// Cria os armários de chão: Tipo determinado pela classe de meuEstilo
```

```
// Cria o objeto cozinha (no estilo requerido)
```

```
ArmarioDeChao armarioDeChao1 = meuEstilo.obterArmarioDeChao();
```

```
ArmarioDeChao armarioDeChao2 = meuEstilo.obterArmarioDeChao();
```

```
...
```

```
Cozinha cozinha = new Cozinha();
```

```
Cozinha.adicionar( armarioDeParede1, ... );
```

```
Cozinha.adicionar( armarioDeParede2, ... );
```

```
...
```

```
Cozinha.adicionar( armarioDeChao1 ... );
```

```
Cozinha.adicionar( armarioDeChao2 ... );
```

```
...
```

```
// VERSÃO CONSIDERANDO OS PROPÓSITOS DE PROJETO
//Determina o estilo instanciando meuEstilo
EstiloModernoC meuEstilo = new EstiloModernoC;
// Cria os armários de parede: Tipo determinado pela classe de meuEstilo
ArmarioDeParede ArmarioDeParede1 = meuEstilo .obterArmarioDeParede();
ArmarioDeParede ArmarioDeParede2 = meuEstilo .obterArmarioDeParede();
...
// Cria os armários de chão: Tipo determinado pela classe de meuEstilo
// Cria o objeto cozinha (no estilo requerido)
ArmarioDeChao armarioDeChao1 = meuEstilo.obterArmarioDeChao();
ArmarioDeChao armarioDeChao2 = meuEstilo.obterArmarioDeChao();
...
Cozinha cozinha = new Cozinha();
Cozinha.adicionar( armarioDeParede1, ... );
Cozinha.adicionar( armarioDeParede2, ... );
...
Cozinha.adicionar( armarioDeChao1 ... );
Cozinha.adicionar( armarioDeChao2 ... );
...
```



# OUTROS PADRÕES

---

- Método-gabarito
- Método-fábrica
- Proxy



# Outros Padrões GoF

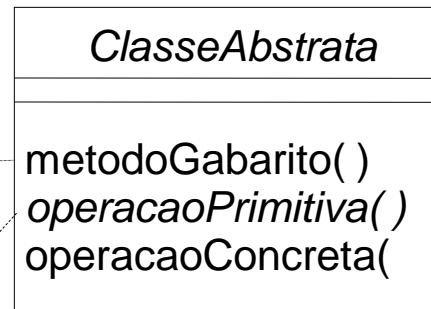
---

- Método-gabarito (*Template Method*)
  - A idéia do Método-gabarito é definir um método gabarito em uma superclasse, que contenha o esqueleto do algoritmo, com suas partes variáveis e invariáveis. Esse método invoca outros métodos, alguns dos quais são operações que podem ser definidas em uma subclasse.
  - Assim, as subclasses podem redefinir os métodos que variam, acrescentando comportamento específico dos pontos variáveis.

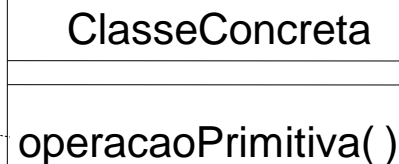
# Padrão Método-gabarito

```
metodoGabarito()  
{  
  ...  
  operacaoPrimitiva()  
  ...  
  operacaoConcreta()  
  ...  
}
```

operacao Primitiva abstrata  
- parte variante  
- redefinida na subclasse



operação concreta  
comportamento por  
omissão  
se puderem ser  
redefinidas ==>  
método gancho (hook  
method)





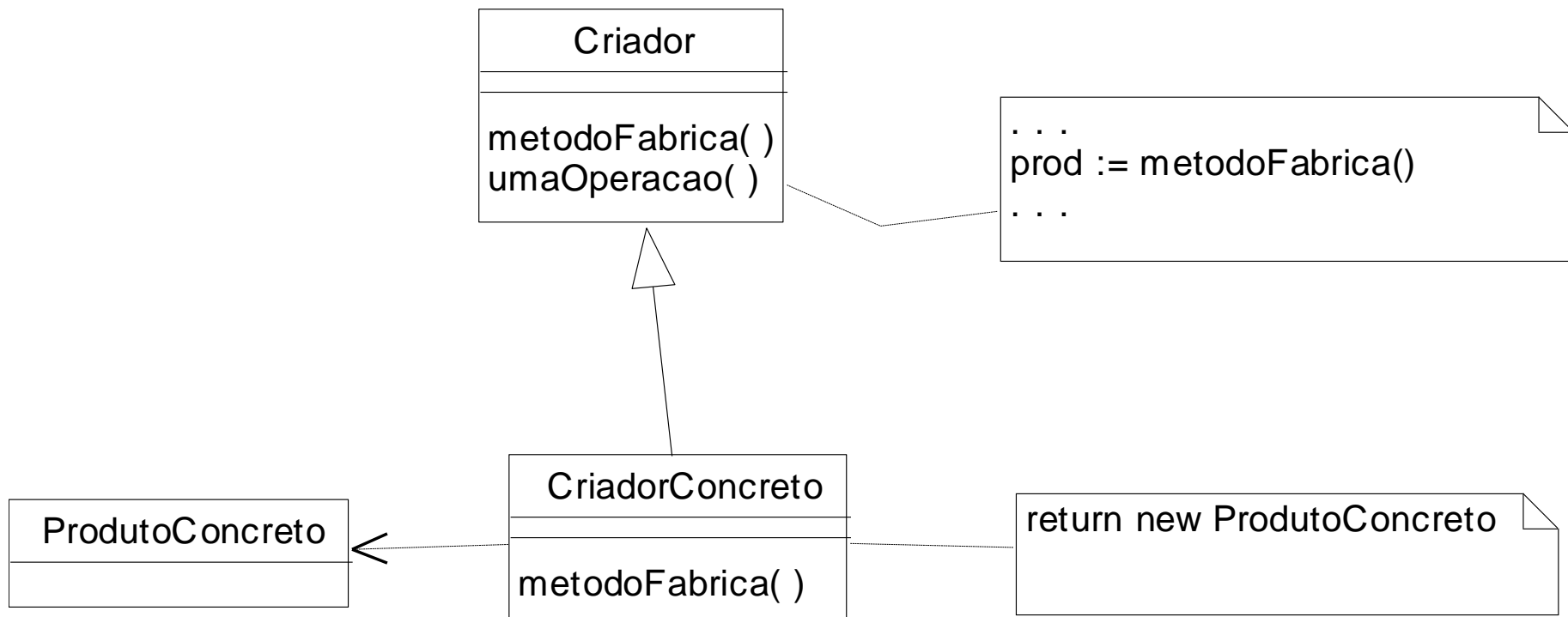


# Outros Padrões GoF

---

- Método-fábrica (Factory Method)
  - Às vezes, uma aplicação não pode antecipar a classe da qual criará um certo objeto
    - Ela sabe que precisará instanciar um objeto, mas não sabe de que tipo
    - Ela pode desejar que suas subclasses especifiquem os objetos a serem criados
  - A idéia do método-fábrica é definir uma interface para a criação de objetos, mas deixar as subclasses decidirem qual classe irão instanciar.
  - Permite que uma classe transfira para as subclasses a responsabilidade pela criação de novas instâncias.

# Padrão Método-fábrica





# Padrão de Projeto: Procurador (*Proxy*)

---

- utilizado nos casos em que uma referência simples a um objeto não é eficiente.
  - Exemplos:
    - quando é necessário fazer referência a um objeto em um espaço de endereçamento diferente,
    - quando se deseja criar objetos sob demanda (adiar a materialização do objeto até que seja realmente necessária),
    - quando se deseja manter um controle maior sobre o objeto original (controle de acesso por exemplo),
    - ou quando se deseja manter uma referência “esperta” (*smart reference*) ao objeto (o ponteiro para o objeto realiza algum comportamento adicional sempre que o objeto é acessado).



# Padrão de Projeto: Procurador

---

- *Problema*

- Usar ponteiros simples para objetos pode não ser eficiente em certos casos

- *Forças*

- Criar um objeto pode ser uma tarefa custosa em certos casos. Por exemplo, se o objeto possui várias ligações com outros objetos ou coleções, então ao criá-lo esses outros objetos também precisam ser criados.
- Quando um objeto possui restrições de acesso por diferentes aplicações-cliente, uma simples referência a esse objeto faz com que a responsabilidade de controle de acesso fique a cargo do objeto, o que o torna mais complexo e menos coeso.
- O mesmo vale para objetos que, quando acessados, precisam realizar algum comportamento adicional. Como esse comportamento é algo em geral não relacionado ao objeto, isso acaba tornando o objeto mais complexo desnecessariamente.



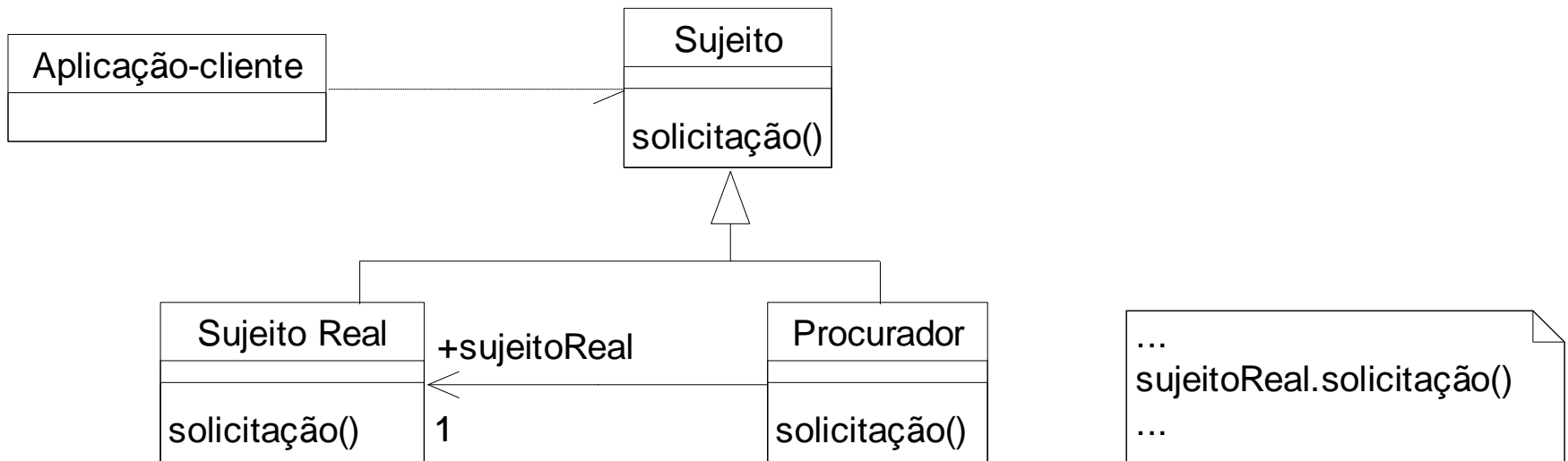
# Padrão de Projeto: Procurador

---

## ■ *Solução*

- Forneça um substituto, ou procurador, ou ainda um placeholder para o objeto, com a finalidade específica de controlar o acesso ao objeto.
- O Procurador mantém referência ao Sujeito Real e possui a mesma interface para que possa ser substituído pelo Sujeito Real.
- Em geral é o Procurador quem cria o sujeito Real, quando necessário.
- O Procurador delega solicitações ao Sujeito Real somente quando necessário. Isso depende do tipo de Procurador, sendo que vários tipos estão descritos na versão original do padrão PROCURADOR (Gamma et al. 1995).

# Padrão de Projeto: Procurador





# Exemplo de aplicação do padrão Procurador

---

- **Problema:** materialização e desmaterialização de objetos
  - Materialização: trazer um registro de uma tabela na forma de um objeto
  - Desmaterialização: persistir um objeto como um registro em uma tabela



# Exemplo de aplicação de padrões de projeto

---

- Exemplo: em um sistema de biblioteca, ao materializar um livro, podemos no mesmo momento materializar os objetos referentes a todos os empréstimos do livro.
  - No entanto, os objetos empréstimo talvez sejam materializados inutilmente, pois o sistema provavelmente não utilizará informações sobre esses objetos, desperdiçando tempo e espaço.
- Solução: materialização sob demanda





# Exemplo de aplicação do padrão Procurador

---

- O padrão Procurador Virtual pode ser usado para solucionar este problema
- Quais objetos associados ao objeto X devem ser recuperados toda vez que um objeto X é recuperado?
  - Para os objetos que não precisarão ser recuperados, cria-se um procurador.



# Exemplo de aplicação do padrão Procurador

---

- Exemplo: ao materializar um Livro, deve-se também materializar seus empréstimos associados?
  - Resposta: não

# Exemplo de aplicação do padrão Procurador

