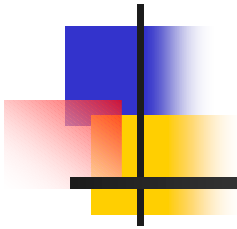


Padrões GoF – Strategy, Observer, Singleton, Abstract Factory



SCE 220 – Análise e Projeto
Orientados a Objetos

Profa Dra Rosana T. V. Braga



Mais Padrões GoF

- Strategy
- Observer
- Singleton
- Abstract Factory



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.

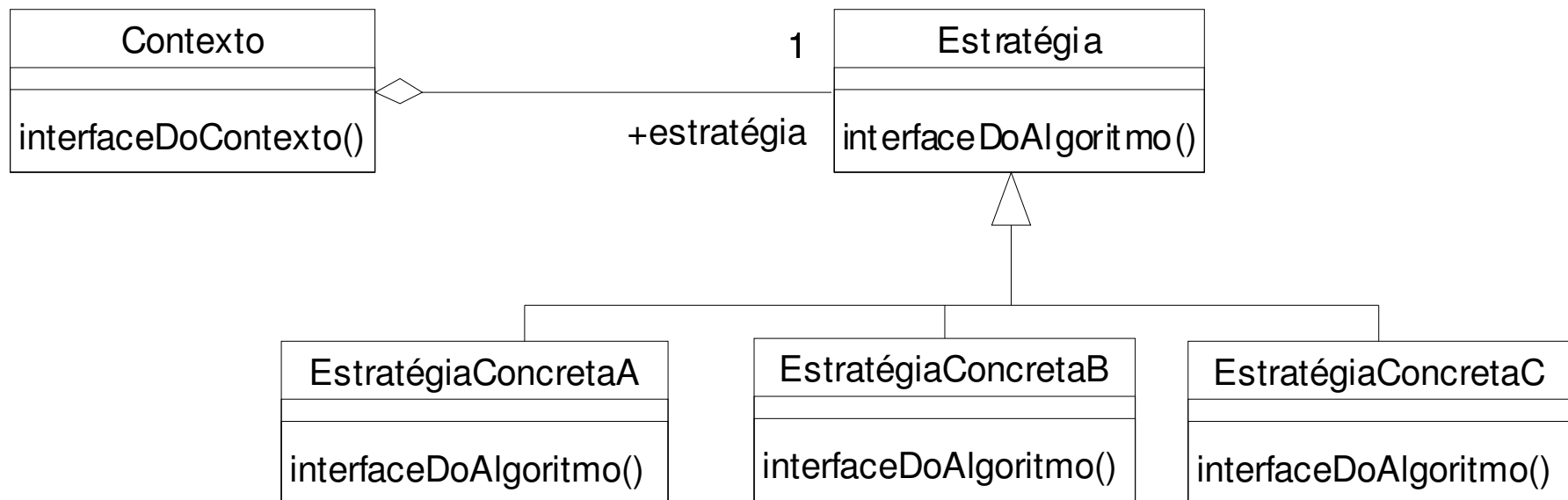


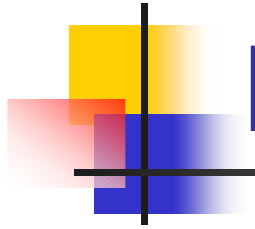
Padrão de Projeto: Estratégia

■ *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





Padrão Observador

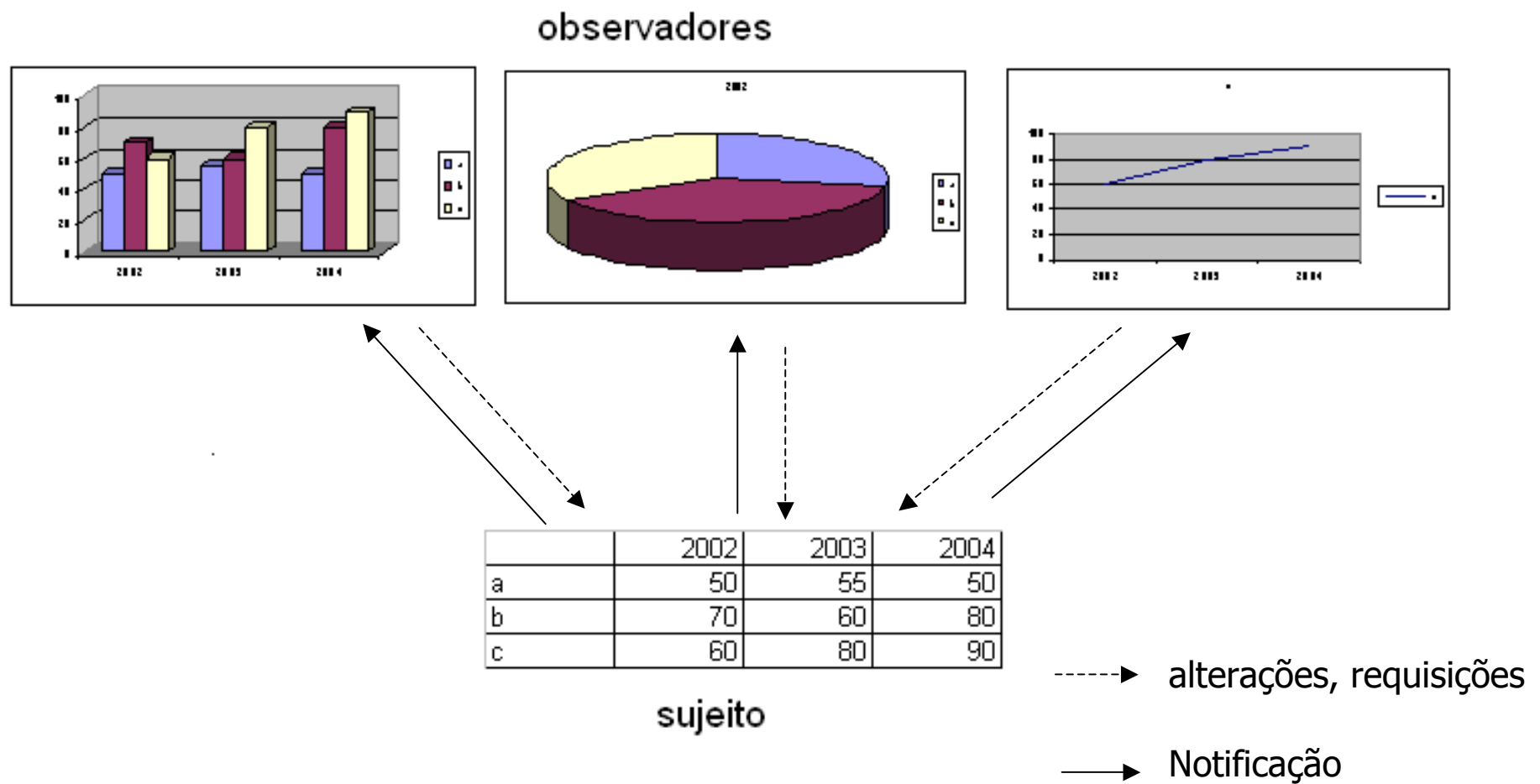
- Intenção: Definir uma dependência de um-para-muitos entre objetos, de forma que quando um objeto muda de estado, todos os seus dependentes são notificados e atualizados automaticamente.



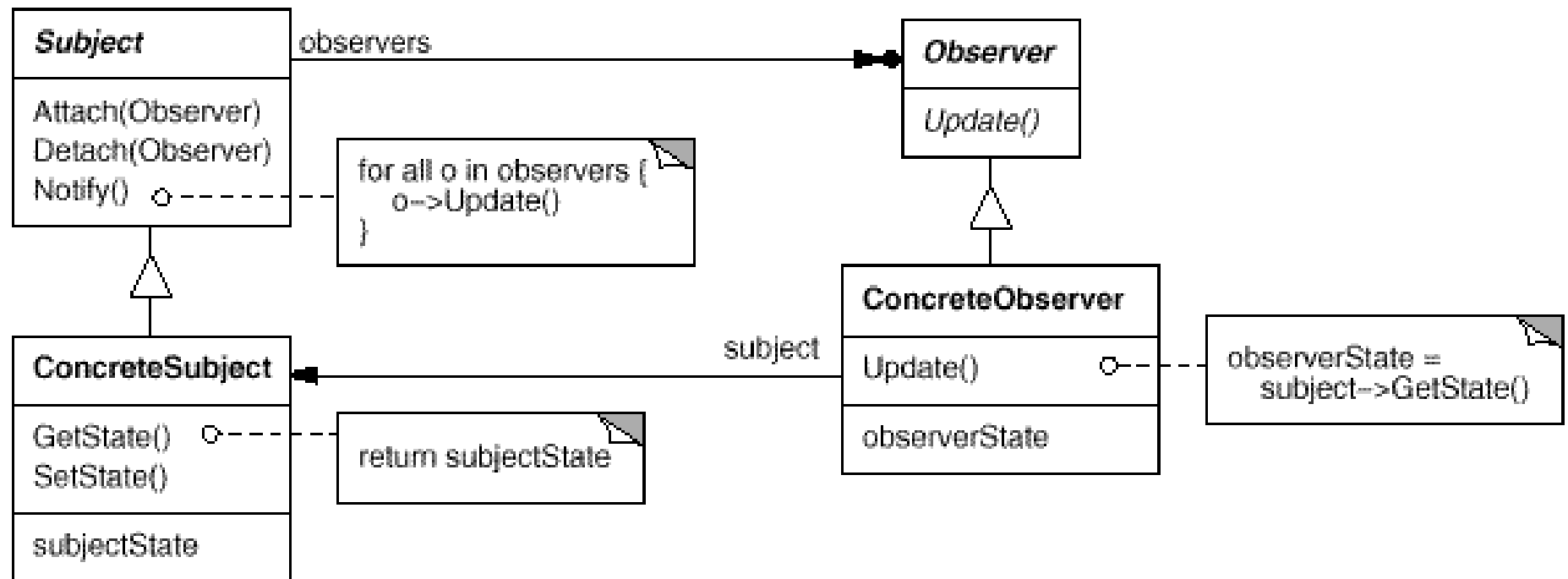
Padrão Observer

- Aplicabilidade: Use o padrão Observer em quaisquer das seguintes situações:
 - Quando uma abstração tem dois aspectos, um dependente do outro. Encapsular esses aspectos em objetos separados permite variar e reutiliza-los independentemente.
 - Quando uma mudança em um objeto requer mudar outros, e não se sabe quantos objetos devem ser mudados.
 - Quando um objeto deve ser capaz de notificar outros objetos sem assumir quem são esses objetos. Em outras palavras, não é desejável que esses objetos estejam fortemente acoplados.

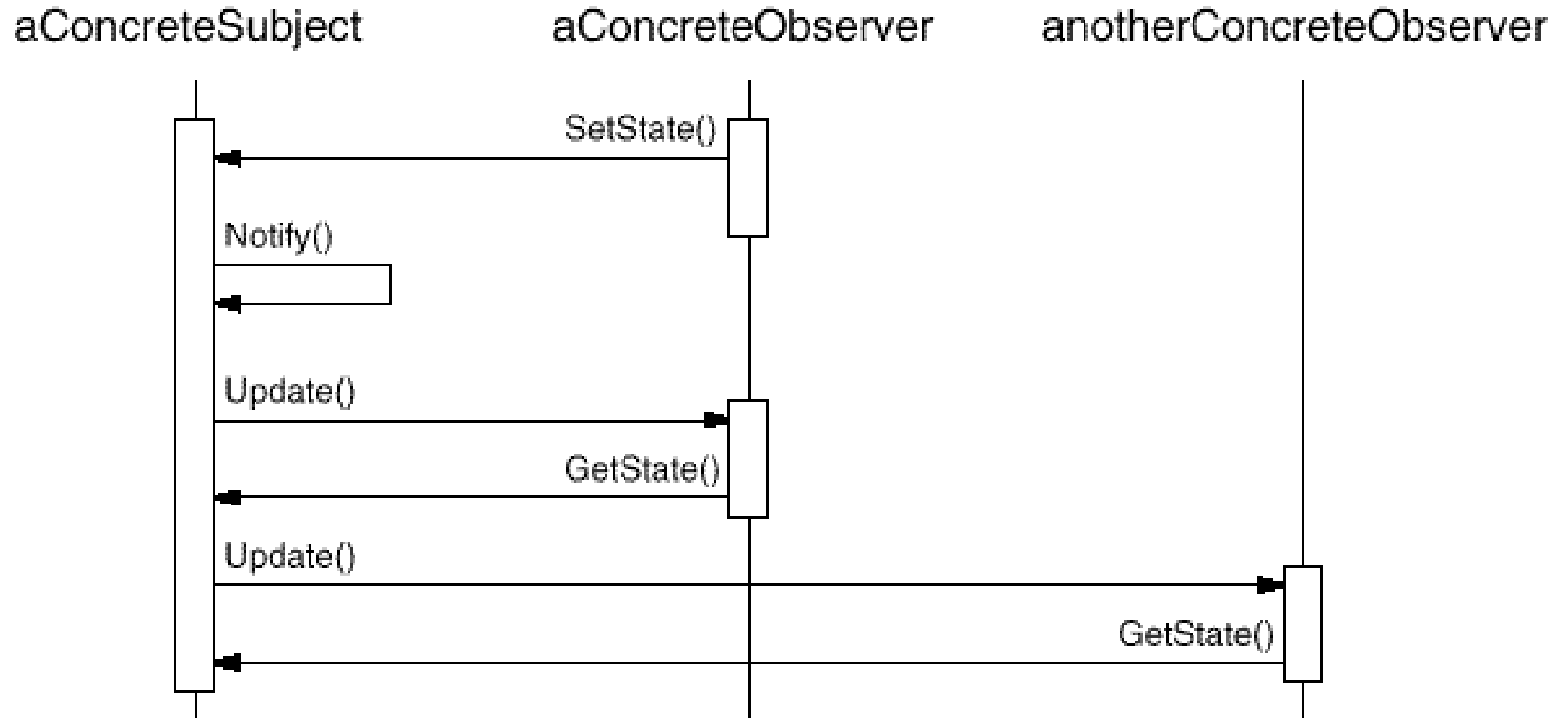
Padrão Observer



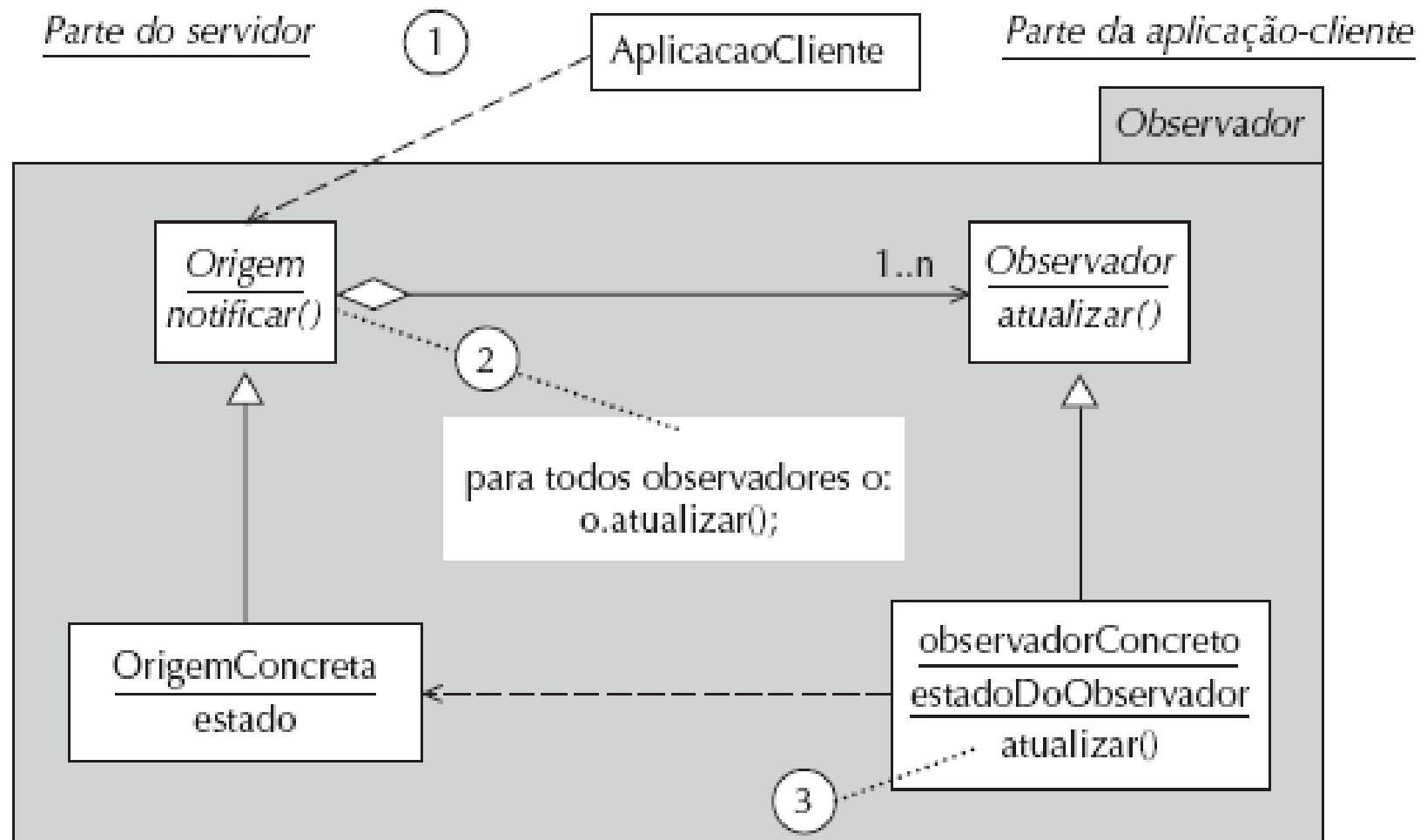
Padrão Observer



Padrão Observer



Como funciona o Observer?

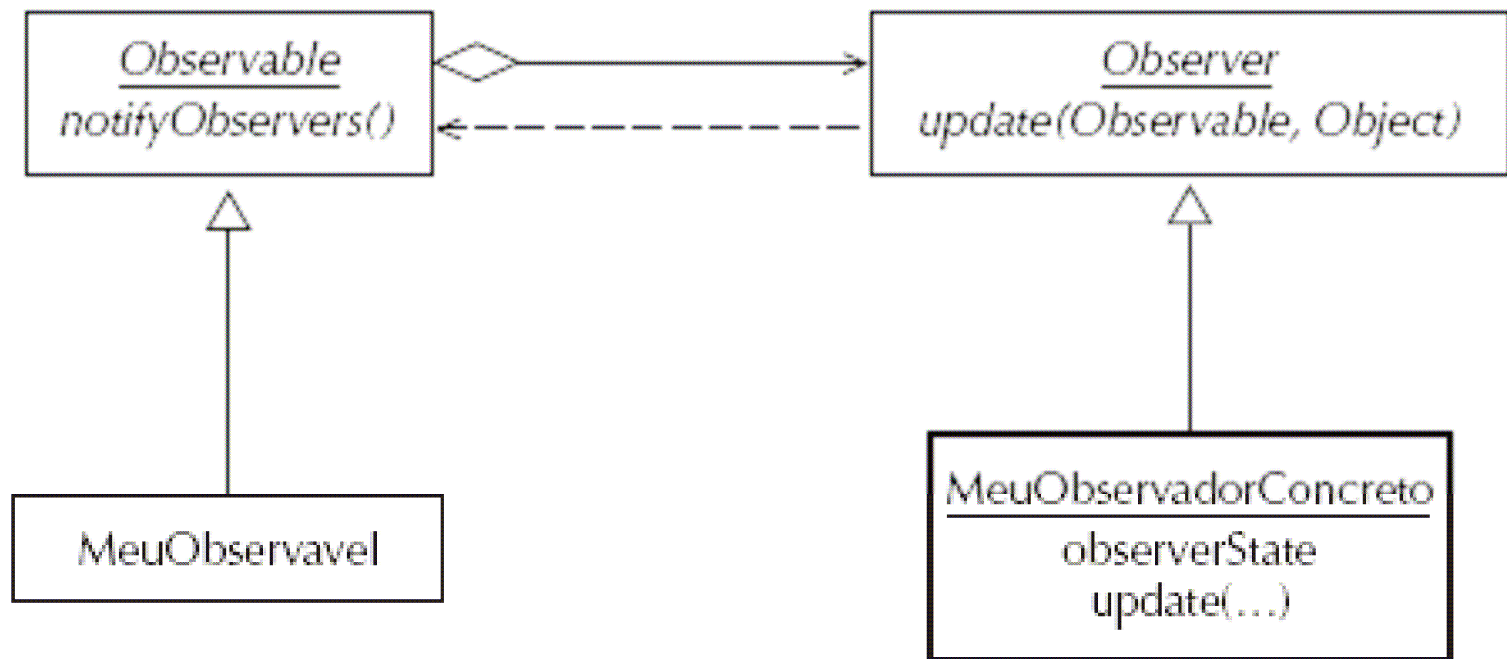




Como funciona o Observer?

- (Passo 1) A aplicação cliente referencia um objeto de interface conhecido, solicitando que os observadores sejam notificados.
 - Por exemplo, a aplicação cliente poderia ser um processo programado para alertar sobre uma alteração de dados. No modelo, isso é mostrado como um objeto `AplicacaoCliente`, que informa ao objeto `Origem` para executar sua função `notificar()`.
- (Passo 2) O método `notificar()` chama a função `atualizar()` em cada objeto `Observador` que ele agrega.
- (Passo 3) A implementação de `atualizar()` depende do `ObservadorConcreto` particular a que pertence.
 - Normalmente, `atualizar()` compara o estado do objeto `ObservadorConcreto` (valores de variáveis) com aquele da origem de dados central, para então decidir se deve ou não alterar seus valores de variáveis da mesma forma.

Observer na API Java



Legenda:

Classe API Java

Classe do Desenvolvedor



Observer na API Java

- A API Java utiliza praticamente os mesmos termos de Gamma et al. [Ga].
- Observe que `update(..)` é um método retroativo (callback), porque fornece aos objetos `Observer` uma referência a sua origem, desta forma permitindo que eles comparem seus dados etc. com o objeto `Observable` na execução de `update()`.
- Como a atualização (`update`) é implementada de modo retroativo, não há necessidade das classes concretas `Observer` manterem referências ao objeto `Observable`.



Padrão de Projeto: Objeto Unitário (Singleton)

- utilizado quando é necessário garantir que uma classe possui apenas uma instância, que fica disponível às aplicações-cliente de alguma forma.
 - Por exemplo, uma base de dados é compartilhada por vários usuários, mas apenas um objeto deve existir para informar o estado da base de dados em um dado momento.



Padrão de Projeto: Objeto Unitário

- ***Problema***

- Como garantir que uma classe possui apenas uma instância e que ela é facilmente acessível?

- ***Forças***

- Uma variável global poderia ser uma forma de tornar um objeto acessível, embora isso não garanta que apenas uma instância seja criada.

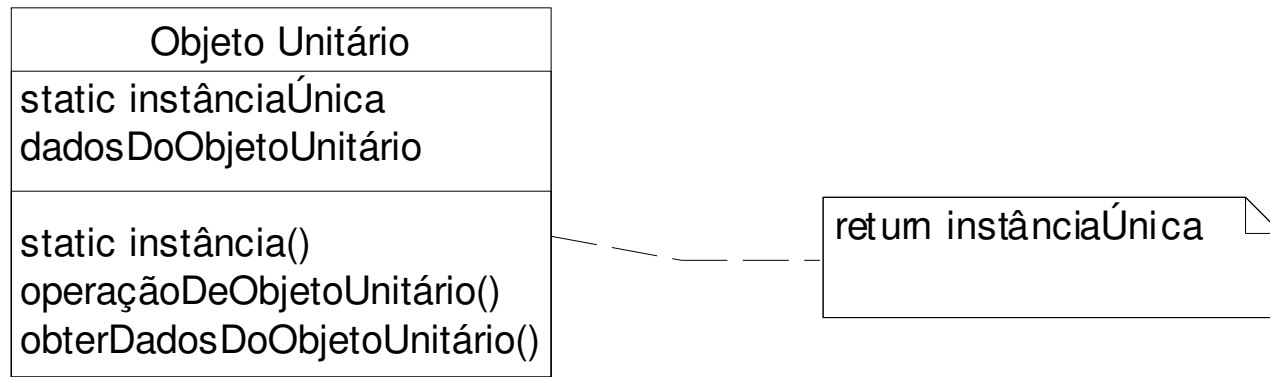


Padrão de Projeto: Objeto Unitário

■ *Solução*

- Fazer a própria classe responsável de controlar a criação de uma única instância e de fornecer um meio para acessar essa instância.
- A classe Objeto Unitário define um método instância para acesso à instância única, que verifica se já existe a instância, criando-a se for necessário.
 - Na verdade esse é um método da classe (static em C++ e Java), ao invés de método do objeto.
 - A única forma da aplicação-cliente acessar a instância única é por meio desse método.
 - Construtor é privado e instância das classes só poderão ser obtidas por meio da operação pública e estática getInstance().

Padrão de Projeto: Objeto Unitário





Exemplo de Singleton

- Aplicado na camada de persistência do Sistema Passe Livre para se obter um único ponto de acesso a um pool de conexões com a base de dados MySQL.

```

// final -> evita que seja feita uma herança
public final class ConnectionPool {

    private static final String DATA_SOURCE_MYSQL = "java:comp/env/jdbc/passeLivre";
    private DataSource dataSource;           // pool de conexão com a base de dados
    private static ConnectionPool mySelf;    // referência para uma única instância dessa classe

    // construtor privado
    private ConnectionPool( DataSource dataSource ) {

        this.dataSource = dataSource;
    }
    // synchronized para evitar que mais de uma instância seja criada num sistema multithread
    public static synchronized ConnectionPool getInstance() {

        try {
            // verifica se ainda não foi criada uma única instância
            if( mySelf == null ) {

                // pega o contexto da aplicação
                Context contexto = new InitialContext();
                // pega o pool de conexões com a base
                DataSource dataSource = ( DataSource )contexto.lookup( DATA_SOURCE_MYSQL );
                // cria a única instância dessa classe
                mySelf = new ConnectionPool( dataSource );
            }

        } catch( NamingException e ) {

            System.err.println( e.getMessage() );
        }

        return mySelf;
    }

    public Connection getConnection() throws SQLException {

        return dataSource.getConnection();
    }
}

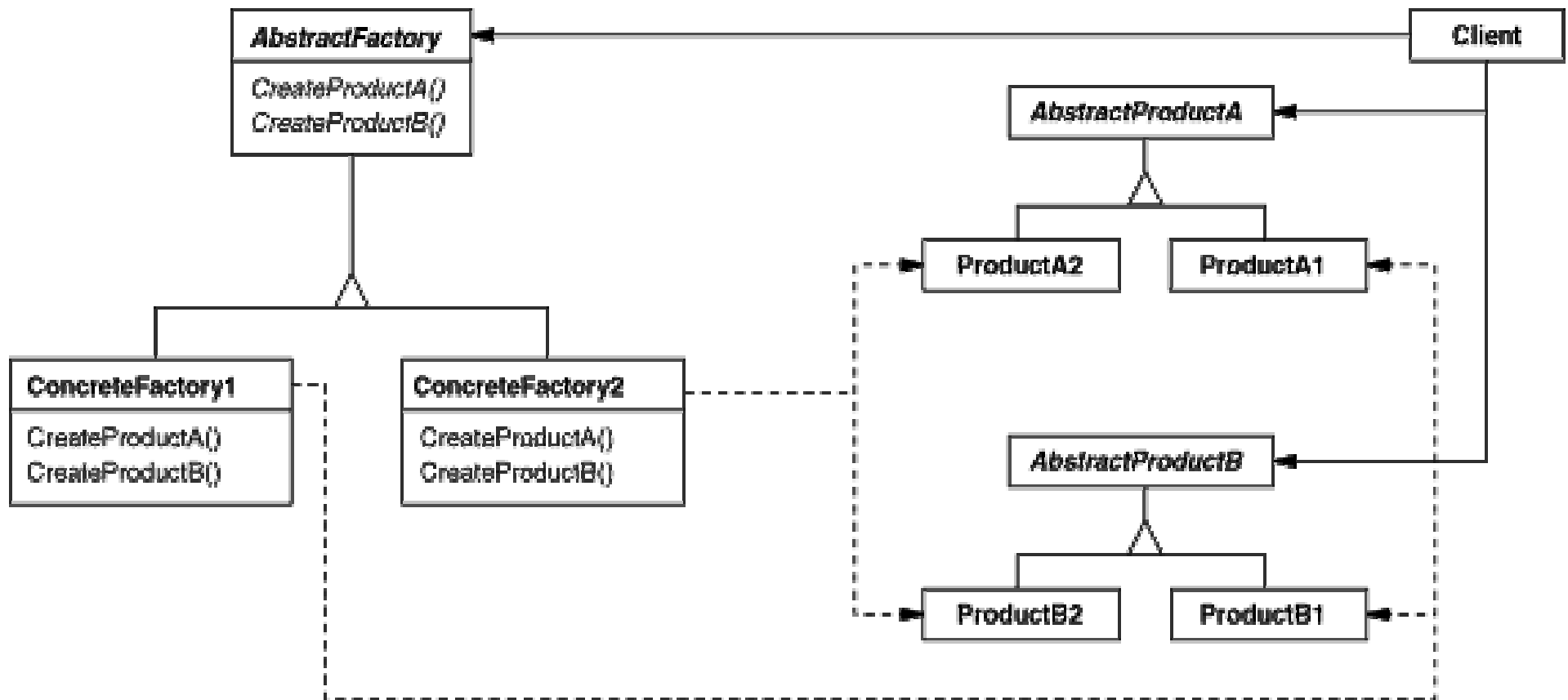
```



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

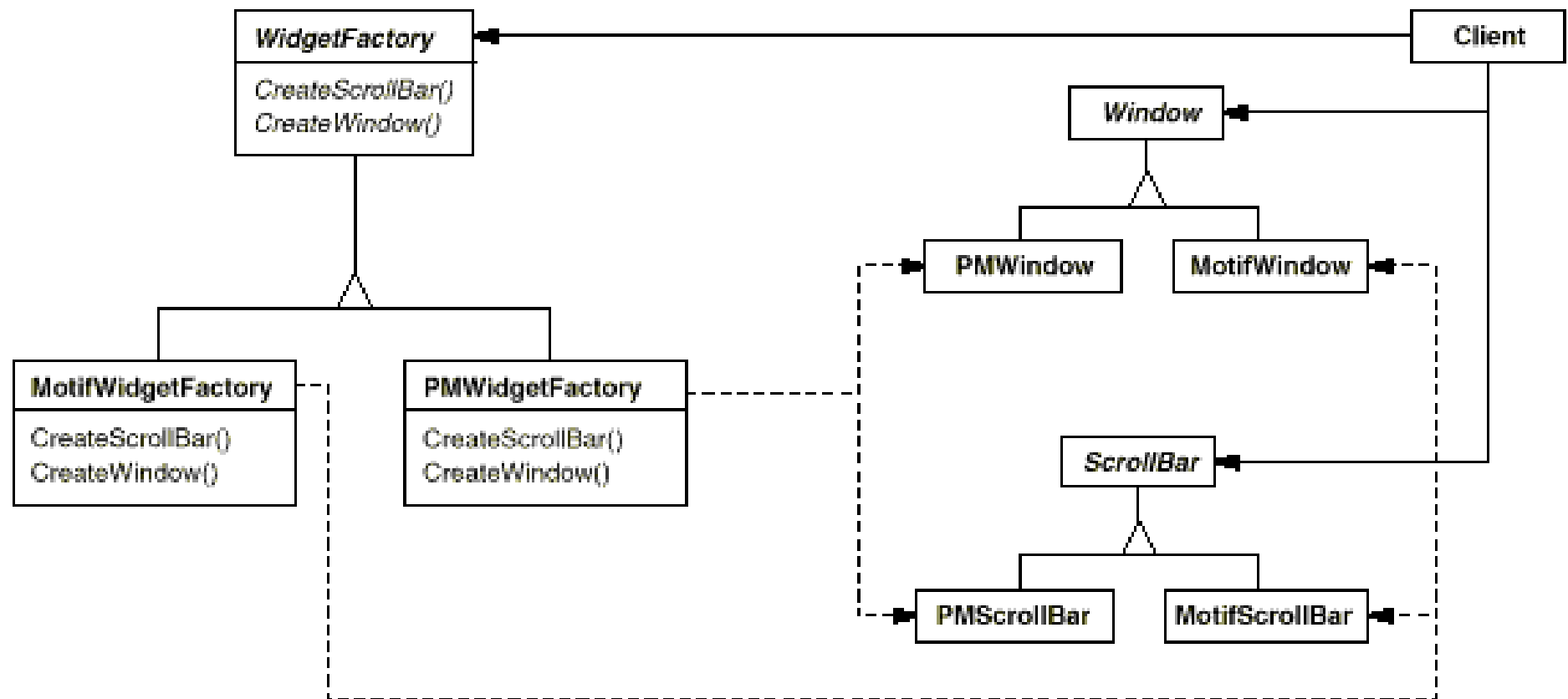


Abstract Factory



- 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.

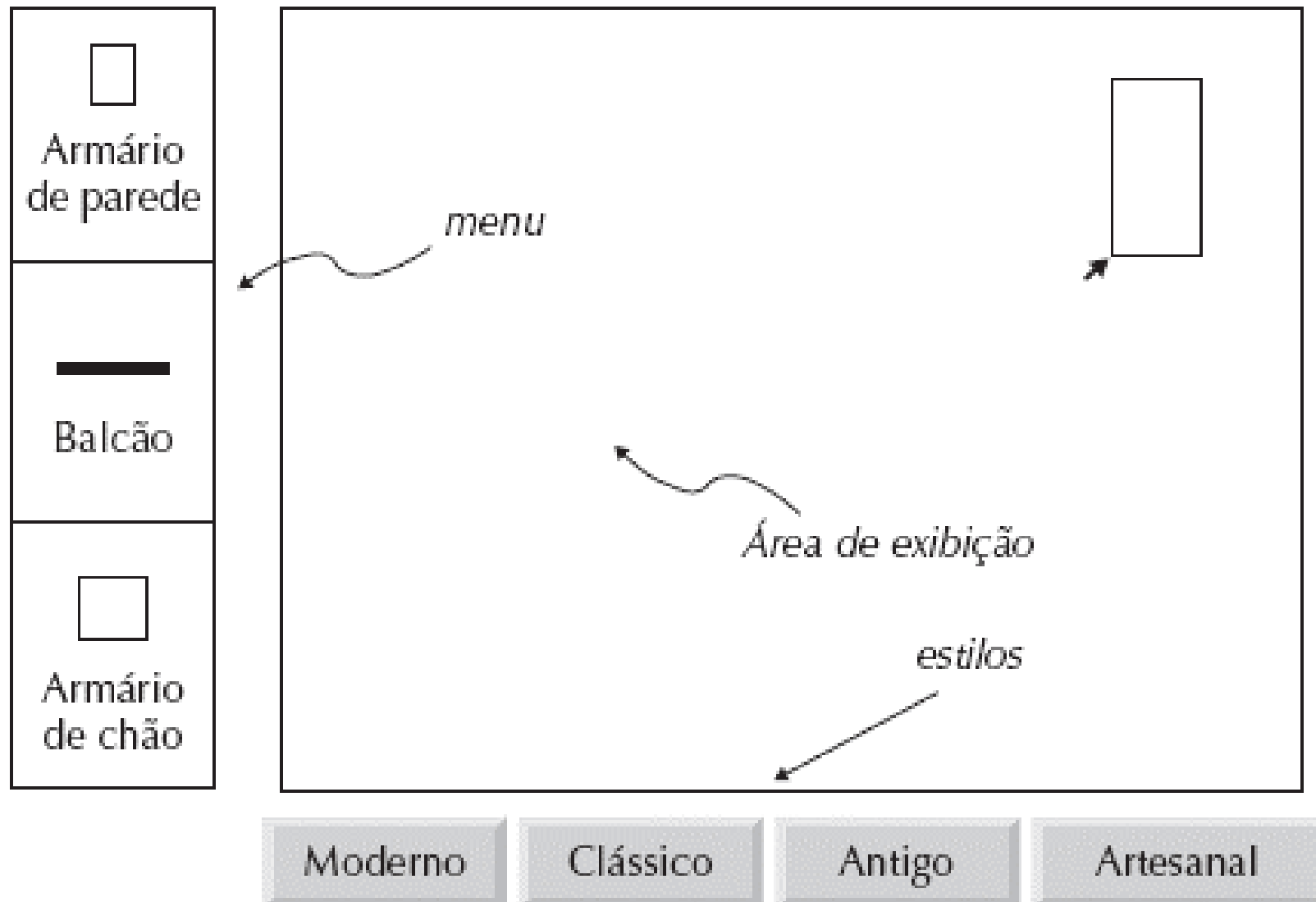
Exemplo: Abstract Factory





Exemplo: Aplicação para construir armários de cozinha

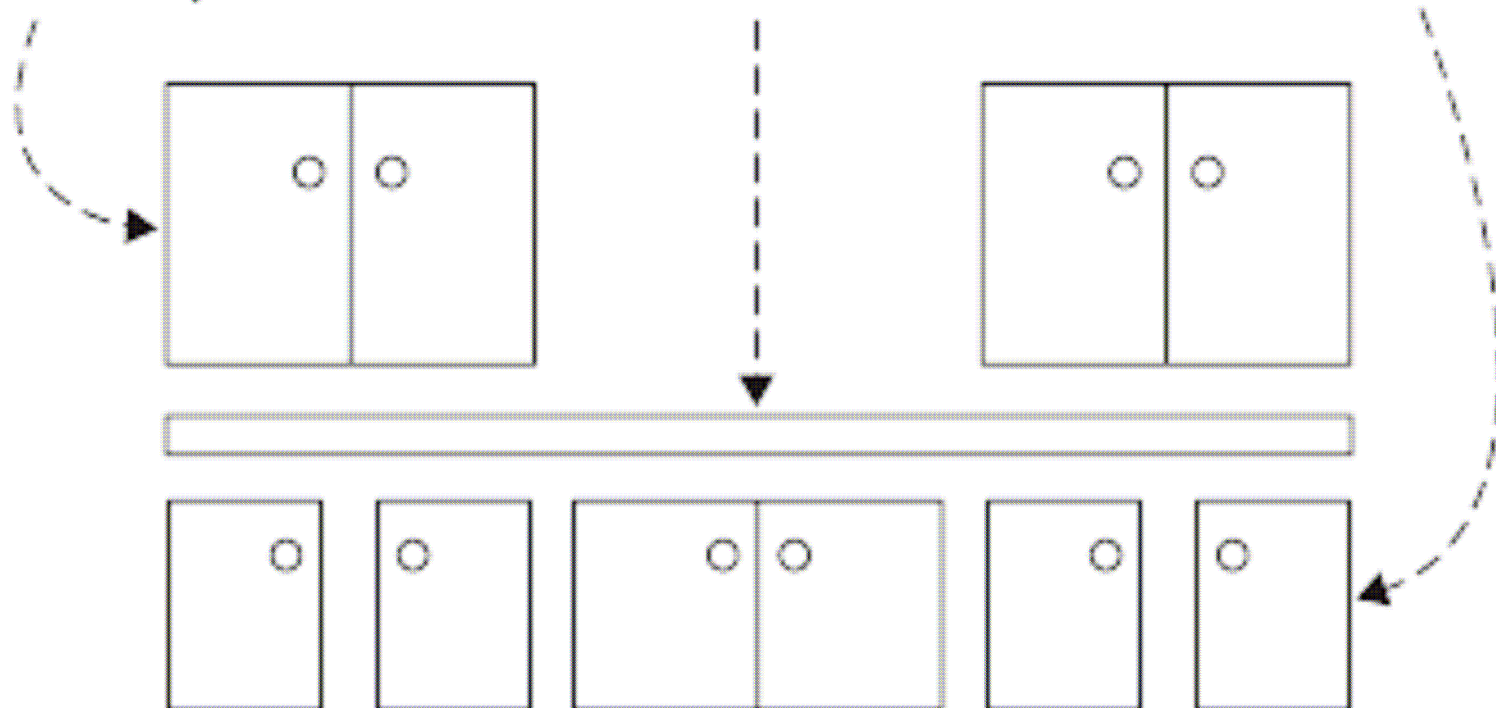
- Proprietários de residências sempre têm a intenção de modernizar suas cozinhas, freqüentemente utilizando um software para visualizar as possibilidades.
- *VisualizadorDeCozinhas*: aplicação que permite que o usuário crie o layout das partes de uma cozinha, sem comprometer-se com um estilo.



Armários de parede

Balcão

Armários de chão

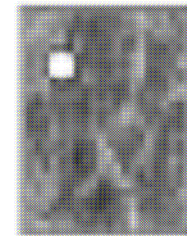
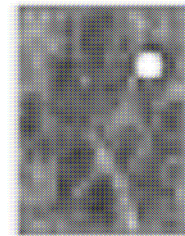
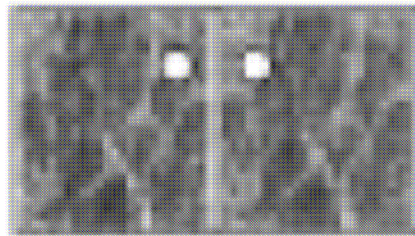
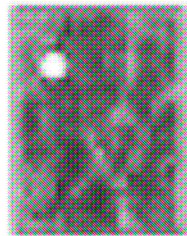
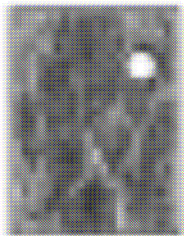
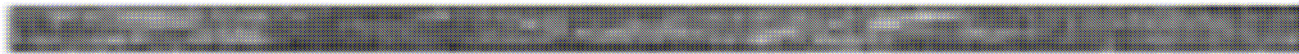
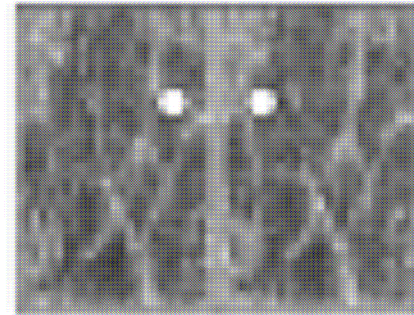
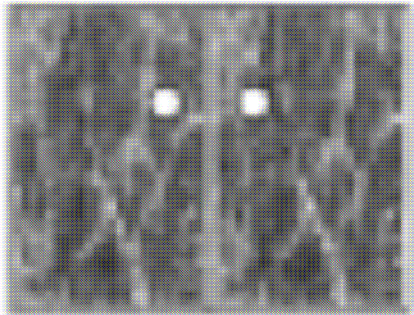


Moderno

Clássico

Antigo

Artesanal



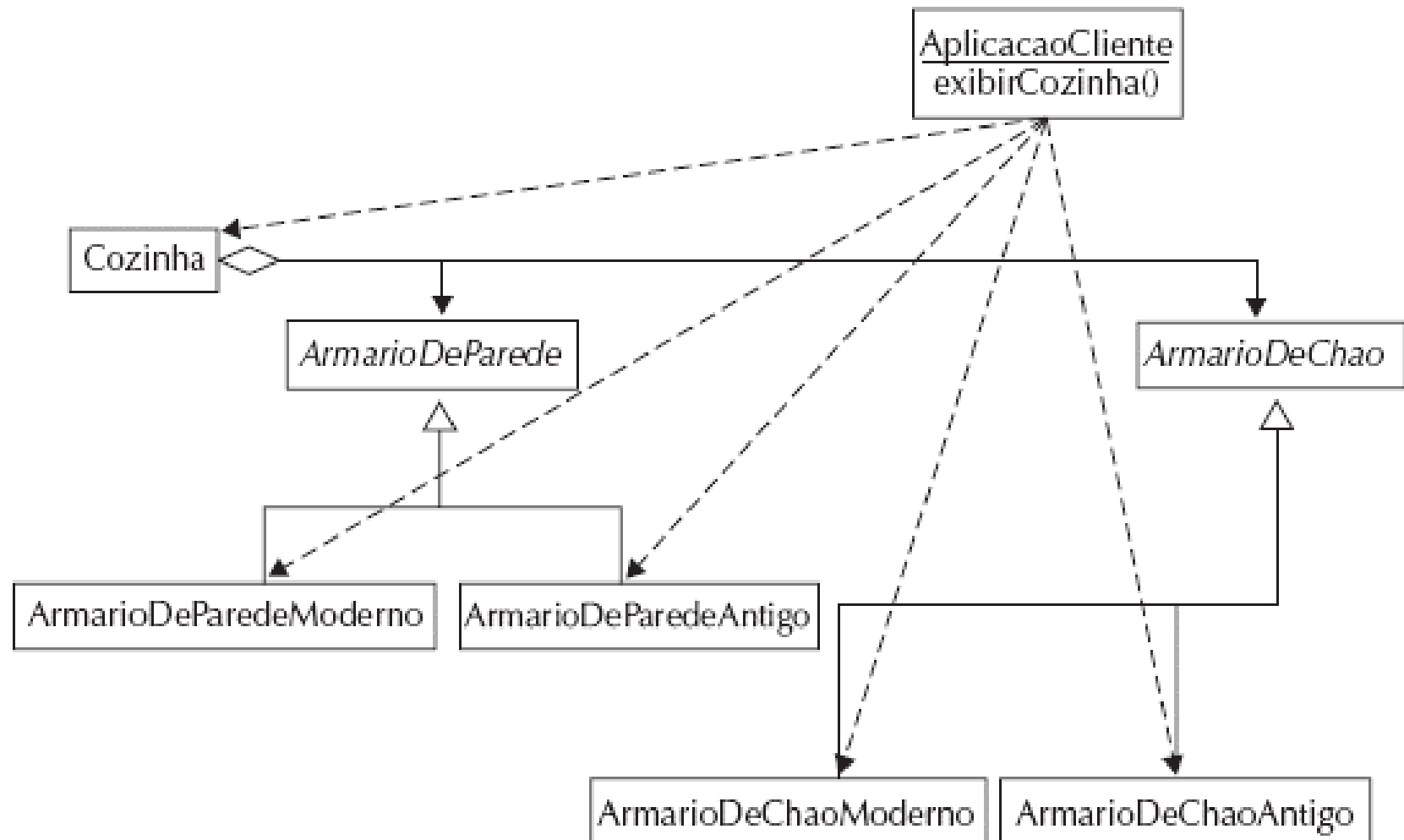
Moderno

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, ... );
```

```
...
```

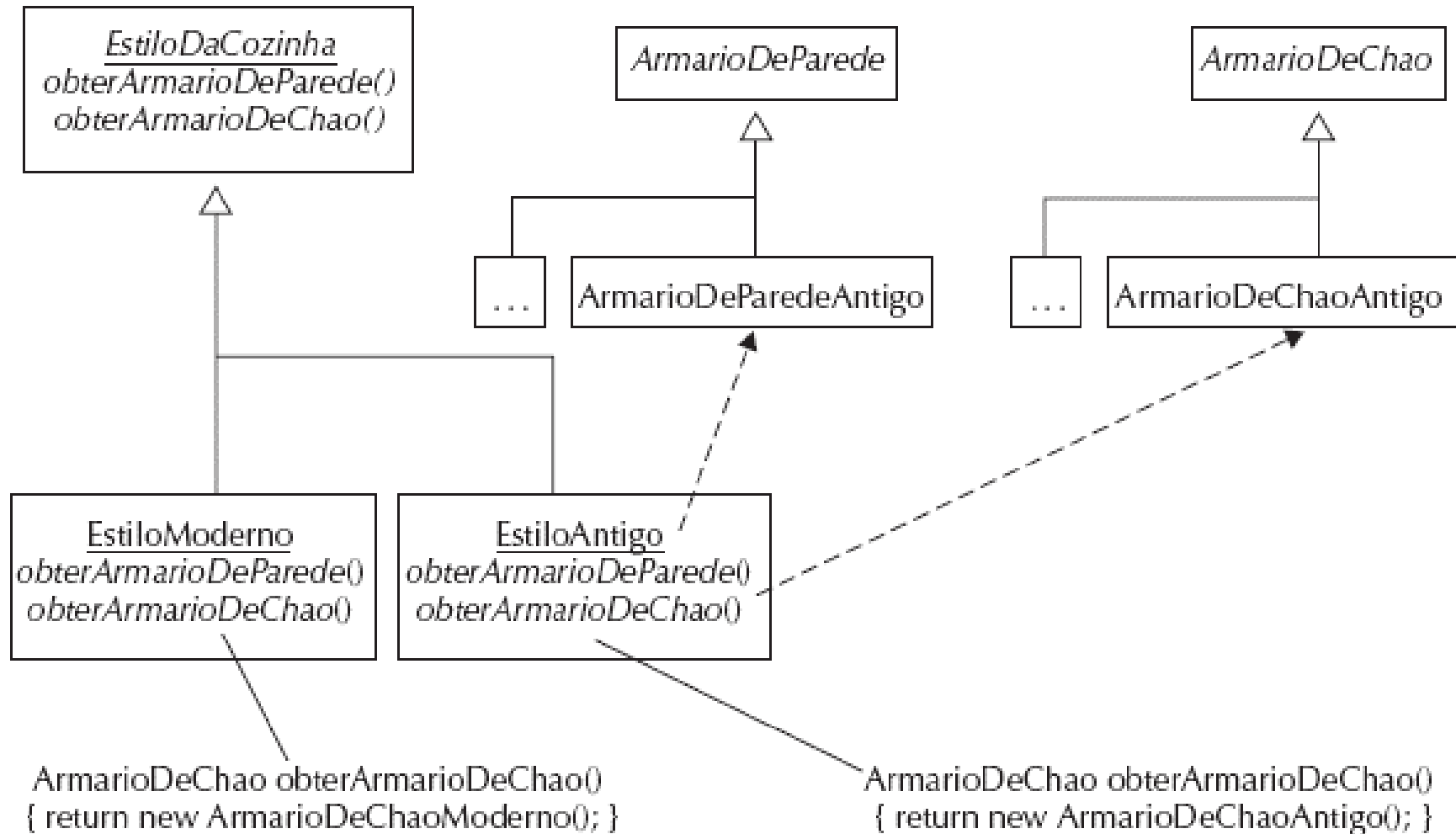
```
// exhibe cozinhaAntiga
```



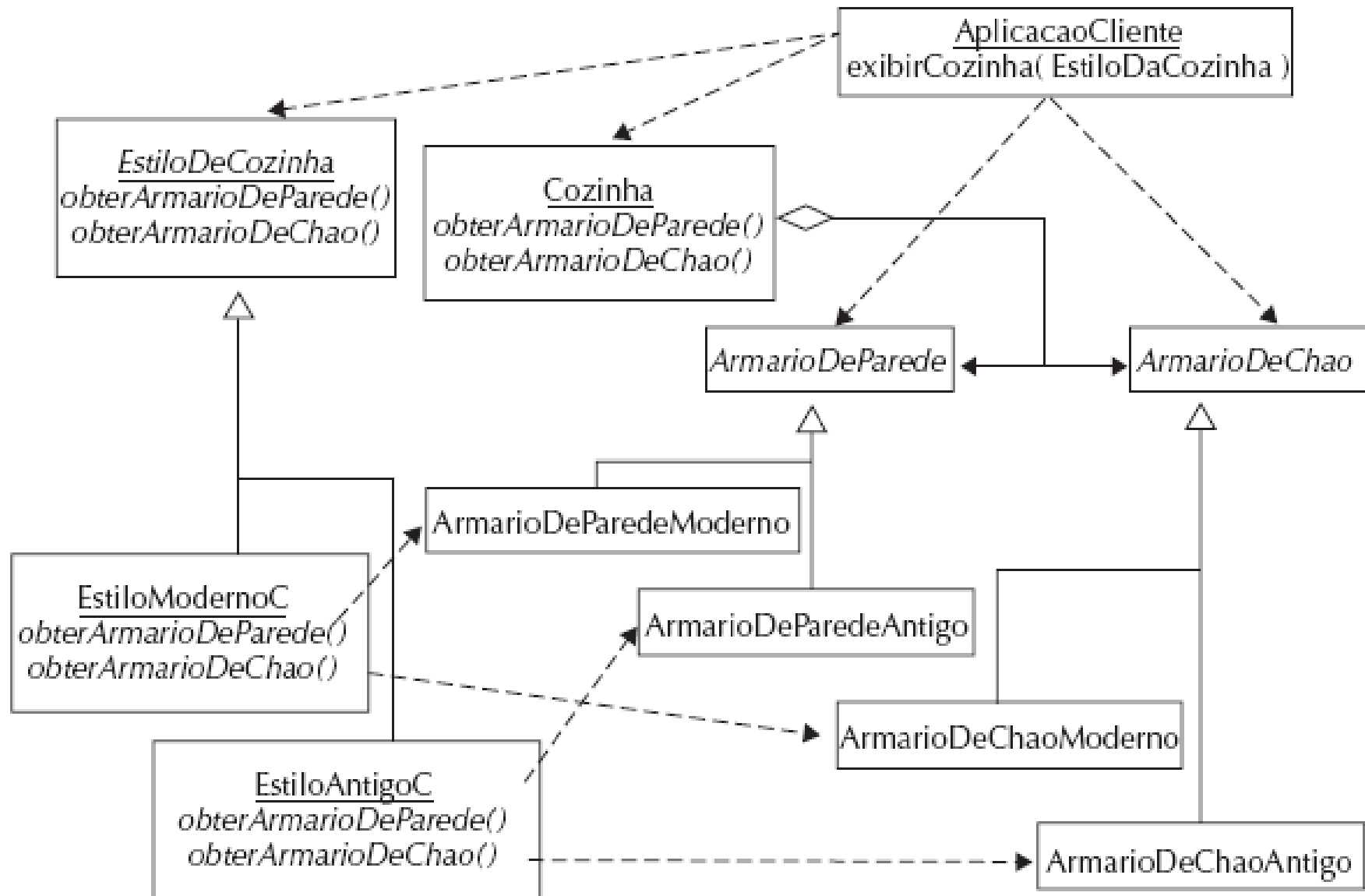
Esboço da Solução

- 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 ...);

...