Padrões de Software

Profa. Rosana T. Vaccare Braga

Laboratório de Engenharia de Software

Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo
São Carlos – SP

Sumário

- Motivação: Reuso de Software
- Introdução a Padrões de Software
- Padrões de Projeto
- Discussão e considerações finais

Reuso de Software

- replicação de qualquer tipo de conhecimento sobre um sistema em outros sistemas similares, com o objetivo de reduzir o esforço de desenvolvimento e a manutenção nesses novos sistemas *(Biggerstaff e Perlis, 1989)*.

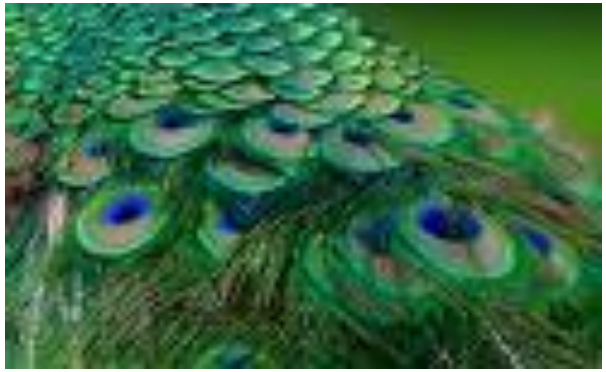
Introdução

- **Reuso de Software:**
  - **Anos 70: módulos e sub-rotinas**
  - **Anos 80: classes e geradores de aplicação**
  - **Anos 90: análise de domínio, componentes, padrões e frameworks;**

Benefícios da reutilização

- Melhores índices de produtividade
- Produtos de melhor qualidade, mais confiáveis, consistentes e padronizados
- Redução dos custos e tempo envolvidos no desenvolvimento de software
- Maior flexibilidade na estrutura do software produzido, facilitando sua manutenção e evolução

Introdução aos Padrões



Introdução aos Padrões

Introdução aos Padrões

- **Origem dos padrões:**
  - **Christopher Alexander (1977, 1979)**
    - **"Cada padrão descreve um problema que ocorre repetidas vezes em nosso ambiente, e então descreve o núcleo da solução para esse problema, de forma que você possa utilizar essa solução milhões de vezes sem usá-la do mesmo modo duas vezes"**
    - **Proposta de padrões extraídos a partir de estruturas de edifícios e cidades de diversas culturas, com o intuito de ajudar as pessoas a construir suas comunidades com a melhor qualidade de vida possível**

Introdução aos Padrões

- **Origem dos Padroes em software:**
  - **Beck e Cunningham: 1987**
    - **pequena linguagem de padrões para guiar programadores inexperientes em Smalltalk**
  - **Peter Coad: 1992**
    - **padrões de análise descobertos na modelagem de sistemas de informação**
  - **James Coplien – 1992**
    - **catálogo de vários estilos (“idioms”), com o intuito de padronizar a escrita de código em C+**
  - **Gamma et al – 1995**
    - **padrões de projeto derivados a partir de experiência prática com desenvolvimento de software orientado a objetos**

Padrões de Software

- **Por que Padrões?**
  - **Desenvolvedores acumulam soluções para os problemas que resolvem com freqüência**
  - **Essas soluções são difíceis de serem elaboradas e podem aumentar a produtividade, qualidade e uniformidade do software**
  - **Como documentar essas soluções de forma que outros desenvolvedores, menos experientes, possam utilizá-las?**

Padrões de Software

- **Padrões de Software:**
  - –Descrevem soluções para problemas que ocorrem com freqüência no desenvolvimento de software *(Gamma 95)*
- **Um bom padrão:**
  - –resolve um problema
  - –é um conceito aprovado (não apenas teoria)
  - –a solução não é óbvia
  - –descreve um relacionamento (estruturas e mecanismos)
  - –tem um componente humano significativo

Padrões de Software

- **Vantagens de Padrões?**
  - **Aumento de produtividade**
  - **Uniformidade na estrutura do software**
  - **Aplicação imediata por outros desenvolvedores**
  - **Redução da complexidade: blocos construtivos**

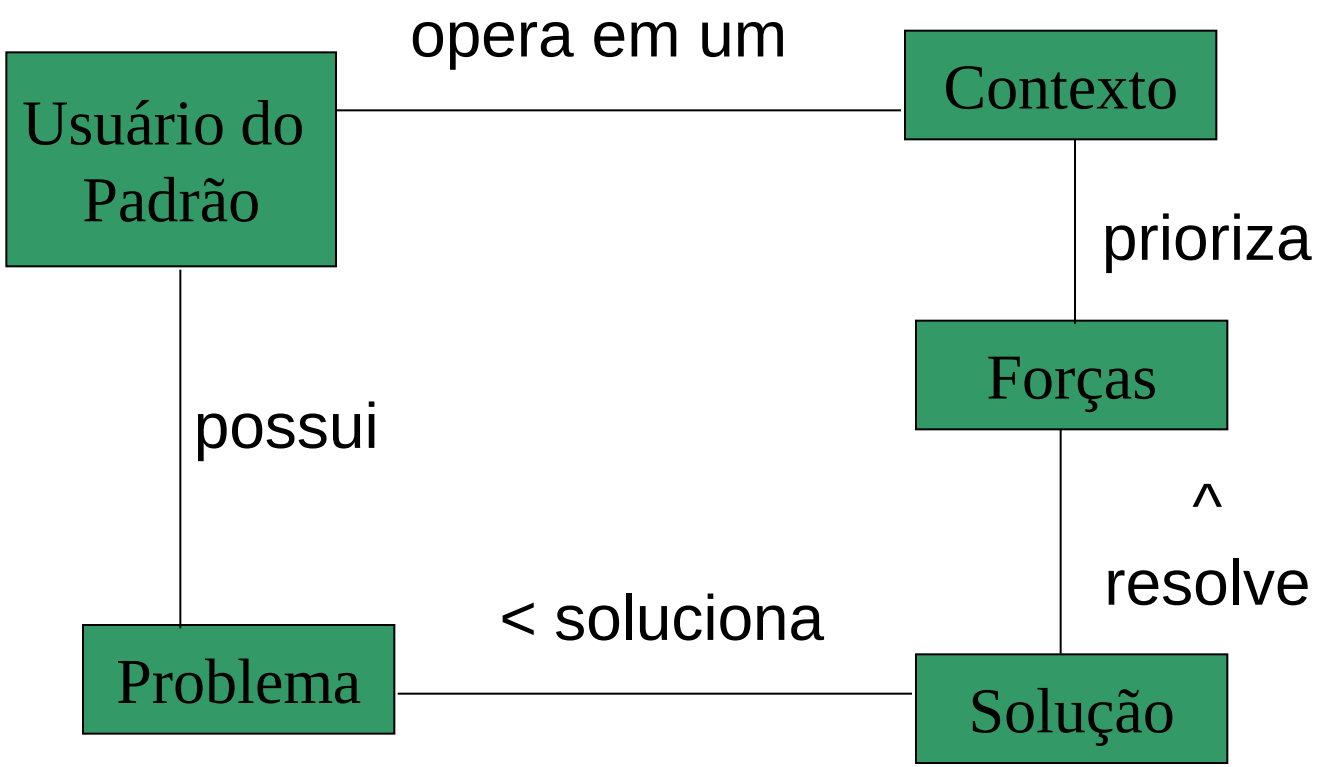
Elementos de um Padrão de Software

- Um padrão tem elementos essenciais e outros opcionais
- Os seguintes elementos são essenciais em um padrão:
  - Nome
  - Problema
  - Solução
  - Conseqüências

Elementos Essenciais de um padrão

- **Nome**: descreve o padrão de maneira muito sucinta (uma ou duas palavras), servindo como referência para comunicação entre os desenvolvedores;
- **Problema**: descreve o problema e o contexto em que ele ocorre. Pode conter um conjunto de condições que devem ser satisfeitas para a aplicação do padrão;
- **Solução**: descreve os elementos necessários para a solução do problema, seus relacionamentos, responsabilidades e colaborações. Essa solução não é restrita a um caso em particular, e sim genérica o suficiente para ser aplicada a diferentes situações;
- **Conseqüências**: são os resultados, positivos e negativos, de se utilizar o padrão. As conseqüências permitem analisar os custos e benefícios da aplicação de um determinado padrão.

Relacionamento entre Elementos do Padrão (Meszaros & Doble)



Elementos Opcionais em um padrão

- **Intenção**: resume o que o padrão faz, bem sucintamente, para ajudar na escolha entre um padrão ou outro;
- **Contexto**: descreve o ambiente em que o problema ocorre e no qual o padrão pode ser aplicável.
  - Trata-se de uma alternativa em que o elemento "problema" da estrutura básica é desmembrado, originando esta seção;
- **Motivação**: mostra um cenário, por meio de um exemplo prático, em que há um problema de projeto e como estruturas de classes e objetos no padrão resolvem o problema;

Elementos Opcionais em um padrão

- **Forças**: apresentam restrições ou pontos positivos que devem ser levados em consideração ao tentar solucionar o problema.
  - Por um lado há forças mais técnicas, como por exemplo, questões de desempenho ou memória, e por outro há forças mais humanas, como legibilidade e compreensibilidade.
- **Participantes**: descreve em detalhes cada um dos elementos que compõem a solução;
- **Colaborações ou Dinâmica**: descreve como os participantes colaboram entre si para cuidarem de suas responsabilidades;

Elementos Opcionais em um padrão

- **Figura (ou Sketch): descreve o funcionamento do padrão por meio de desenhos, modelos ou diagramas.**
  - **Pode ser um elemento à parte ou ser embutido na Solução;**
- **Aliases ou Também Conhecido Por: cita outros nomes usados para o padrão, se existirem.**
  - **Muitas vezes esses outros nomes são variantes ou foram escritos por outros autores com outro enfoque;**

Elementos Opcionais em um padrão

- **Contexto Resultante**: descreve o estado do sistema após a aplicação do padrão, mostrando:
  - quais forças foram resolvidas
  - que novos problemas podem surgir decorrentes de sua aplicação e
  - que padrões podem ou devem ser aplicados depois dele *[Coplien, 1996]*.
  - pode também ser embutido no elemento "Conseqüências";

Elementos Opcionais em um padrão

- **Padrões Relacionados ou Veja Também**: mostra o relacionamento do padrão com outros padrões que se referem ao mesmo problema, podendo citar outros padrões usados em conjunto, outras soluções para o mesmo problema ou variações do padrão;
- **Raciocínio** (ou Rationale): descreve porque a solução é a mais apropriada para o problema dentro do contexto existente e detalha como e porquê a solução funciona.
  - Alternativamente, pode contar a história que deu origem ao padrão, funcionando como uma fonte de aprendizagem;

Elementos Opcionais em um padrão

- **Implementação**: para padrões em níveis de abstração mais baixos, como os padrões de projeto, este elemento ajuda a descrever técnicas, dicas ou questões específicas de linguagem necessárias para implementar o padrão;
- **Exemplos**: apresenta exemplos de como aplicar o padrão, auxiliando o entendimento da solução quando ela é muito abstrata;
- **Usos Conhecidos**: mostra exemplos da aplicação do padrão em sistemas existentes.
  - Recomenda-se incluir pelo menos três usos conhecidos (regra de 3).

Formato de um padrão

- **Ao escrever um padrão, pode-se combinar os elementos essenciais e os elementos opcionais.**
- **A princípio, não deve se prender a nenhum formato de escrita e não há obrigatoriedade de uso de todos os elementos em todos os padrões.**
- **Na medida em que o padrão vai sendo reusado, pode-se mudar seu formato para facilitar seu entendimento e reuso**
- **Existem algumas sugestões de gabaritos já testados e aprovados : *Gamma et al. (1995), Coplien (1996) e Buschmann et al. (1996b)***

Formato de um padrão de Software - exemplos

Gamma	**Coplien**	**Buschmann**
Nome	Nome	Nome
Classificação	Contexto	Também Conhecido Como
Intenção	Problema	Exemplo
Também Conhecido Como	Forças	Contexto
Motivação	Solução	Problema
Aplicabilidade	Esquema	Solução
Estrutura	Contexto	Estrutura
Participantes	Resultante	Dinâmica
Colaborações	Raciocínio	Implementação
Implementação		Exemplo Resolvido
Código Exemplo		Variantes
Conseqüências		Usos Conhecidos
Usos Conhecidos		Conseqüências
Padrões Relacionados		Veja Também

Tipos ou Categorias de Padrões

- Padrões podem ser
  - Organizacionais
  - Arquiteturais
  - De Análise
  - De Projeto
    - Gamma et al
    - Outros : p.ex J2EE
  - De programação (estilos)
  - De usabilidade
  - Educacionais
  - Etc.

Padrões de Projeto - GoF

- **Catálogo de Padrões de Projeto [Gamma95]**
  - **Dois critérios de classificação**
    - **Propósito - reflete o que o padrão faz**
      - **De Criação: trata da criação de objetos**
      - **Estrutural: cuida da composição de classes e objetos**
      - **Comportamental: caracteriza o modo como as classes e objetos interagem e distribuem responsabilidades**
    - **Escopo**
      - **Classe: trata do relacionamento entre classes e subclasses (herança - relacionamento estático)**
      - **Objetos: lida com a manipulação de objetos (podem ser modificados em tempo de execução)**

GoF: Gang of Four – apelido dado aos quatro autores do livro

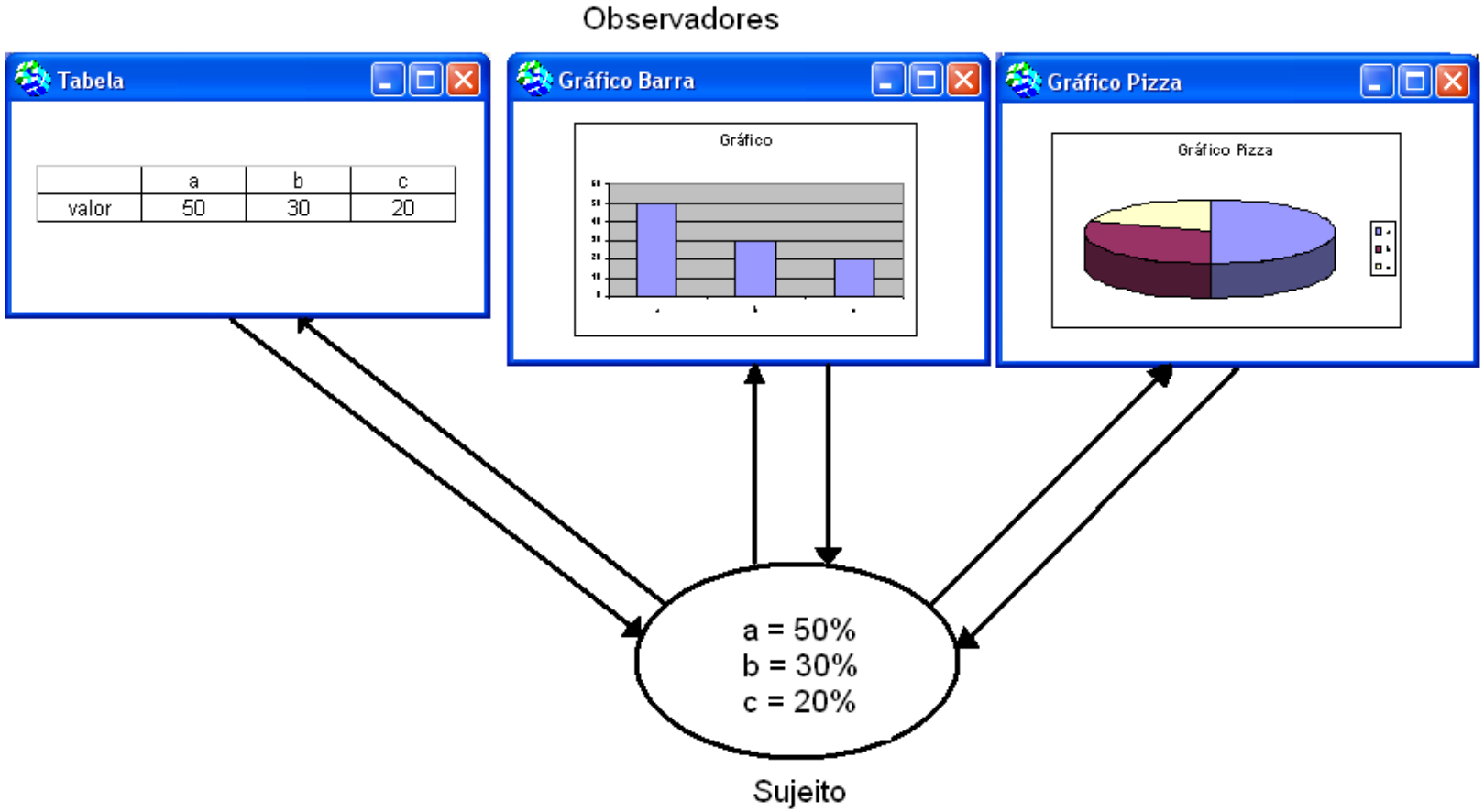
Padrões de Projeto - GoF

Escopo	Propósito: De Criação	Propósito: Estrutural	Propósito: Comportamental
Classe	Método-fábrica	Adaptador	Interpretador Método Gabarito
Objeto	Fábrica Abstrata Construtor Protótipo Objeto Unitário	Adaptador Ponte Composto Decorador Fachada Peso-pena Procurador	Cadeia de Responsabilidade Comando Iterador Mediador Memento Observador Estado Estratégia Visitador

Padrão de Projeto: Observador (*Observer*)

- **Aplicação: sistemas cujas informações são preferencialmente centralizadas em uma única fonte (sujeito), e existem diversas visões (observadores) que devem ser atualizadas sempre que a informação for modificada**

Padrão de Projeto: Observador



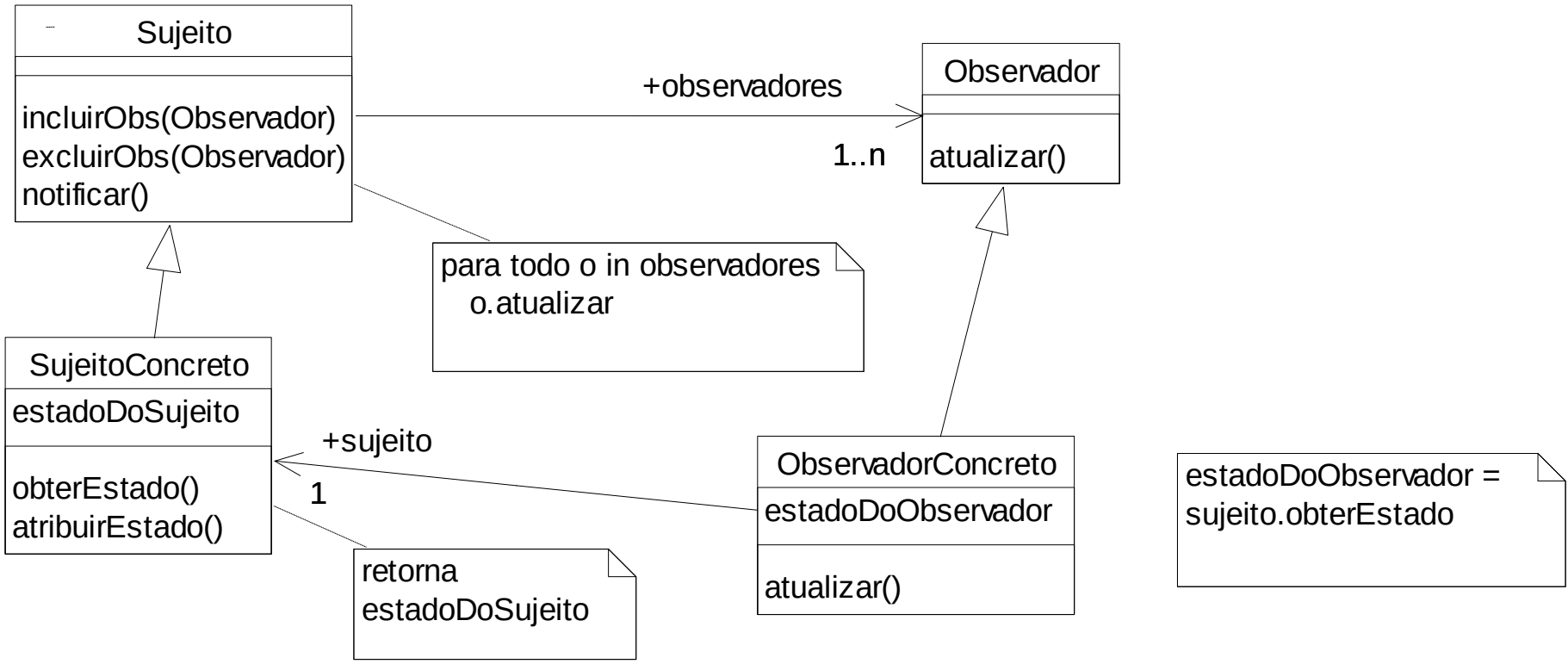
Padrão de Projeto: Observador

- ***Problema***
  - **Quando se utiliza orientação a objetos, o sistema é dividido em um conjunto de classes que interagem, mas é difícil manter a consistência entre objetos relacionados.**
- ***Forças***
  - **Qualquer tentativa de manter a consistência aumenta o acoplamento entre as classes.**
    - **Por exemplo, em uma ferramenta integrada, pode-se separar os dados em si das diversas visões sobre esses dados, de forma que mudança em uma das visões seja refletida nas demais, mesmo que as visões não tenham conhecimento umas das outras.**
  - **Essa separação pode levar a um projeto mais fácil de entender, estender, manter e reusar cada objeto separado.**

Padrão de Projeto: Observador

- ***Solução***
  - **Separar em objetos diferentes as informações e suas visões.**
  - **Definir uma dependência de um-para-muitos entre os objetos, de forma que quando um objeto muda de estado, todos os seus dependentes são notificados e atualizados automaticamente.**

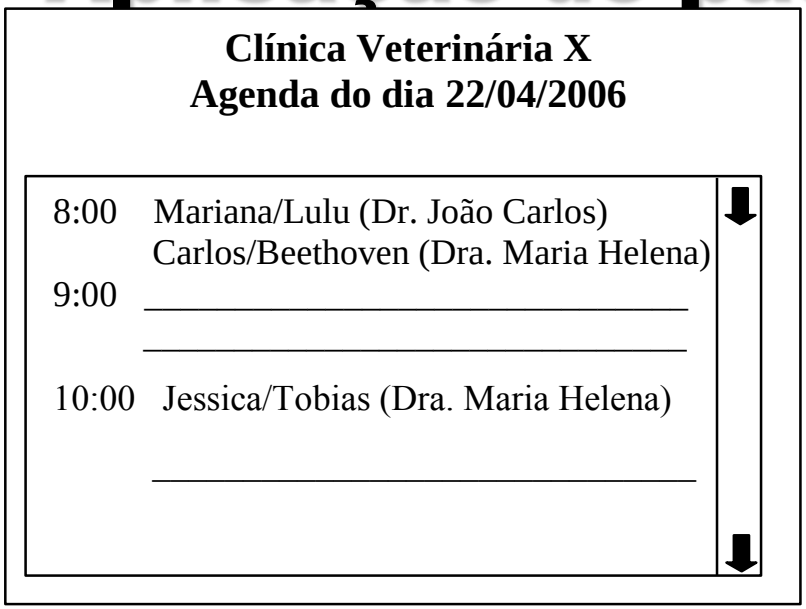
Padrão de Projeto: Observador

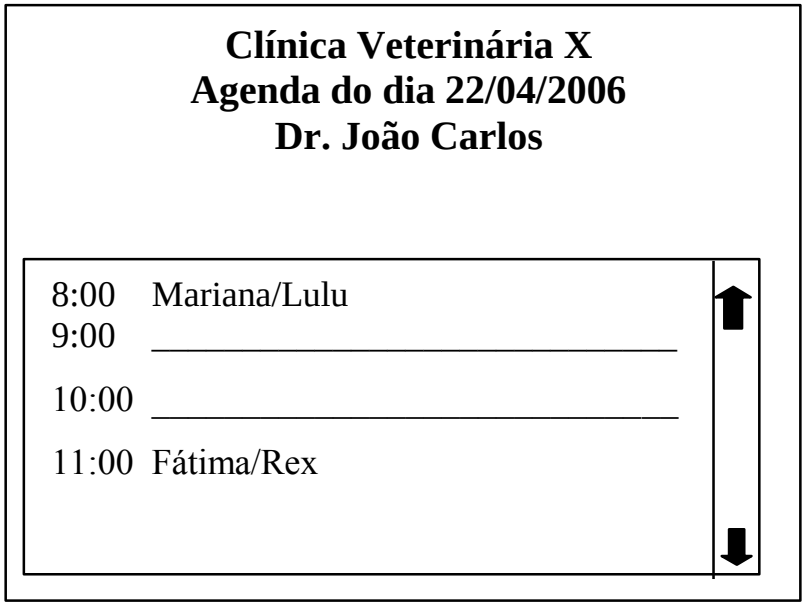


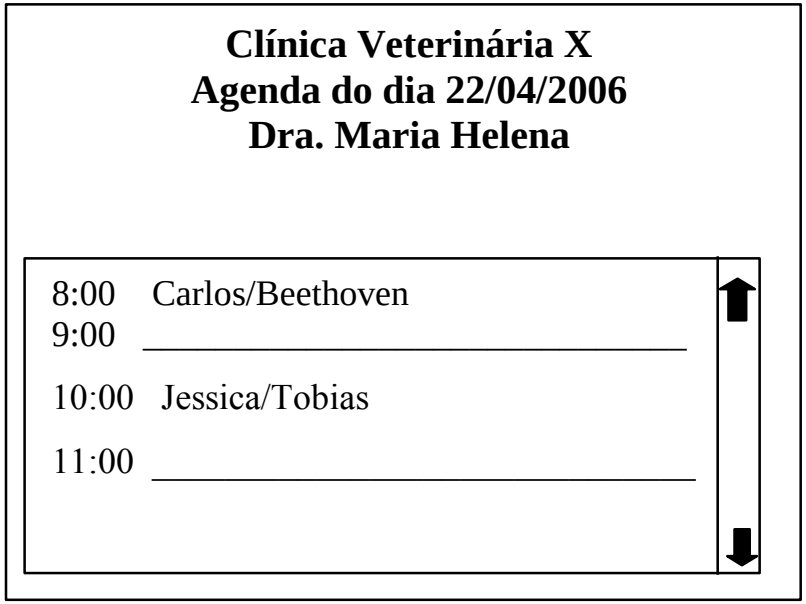
Exemplo de aplicação de padrões de projeto

- **Exemplo: padrão de projeto Observador usado para projetar a interface gráfica com o usuário para os agendamentos atribuídos aos diversos veterinários que trabalham em uma clínica.**

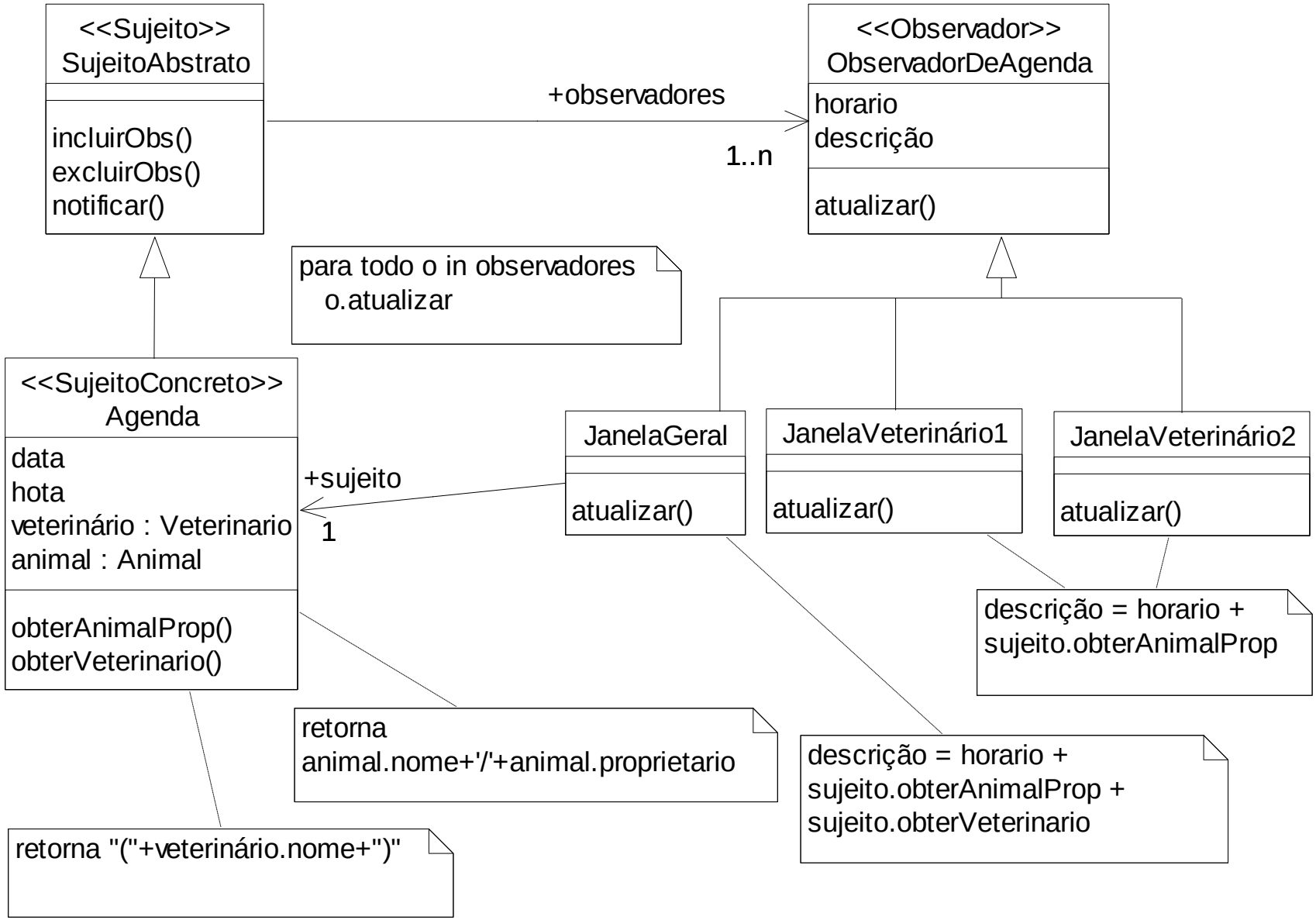
Aplicação do padrão Observador







Aplicação do padrão Observador



Padrão de Projeto: Estado

- aplicado quando um objeto apresentar comportamentos diferentes, dependendo de seu estado atual, mas o uso de estruturas case para implementar isso pode favorecer erros, principalmente quando o sistema evolui e novos estados surgem.

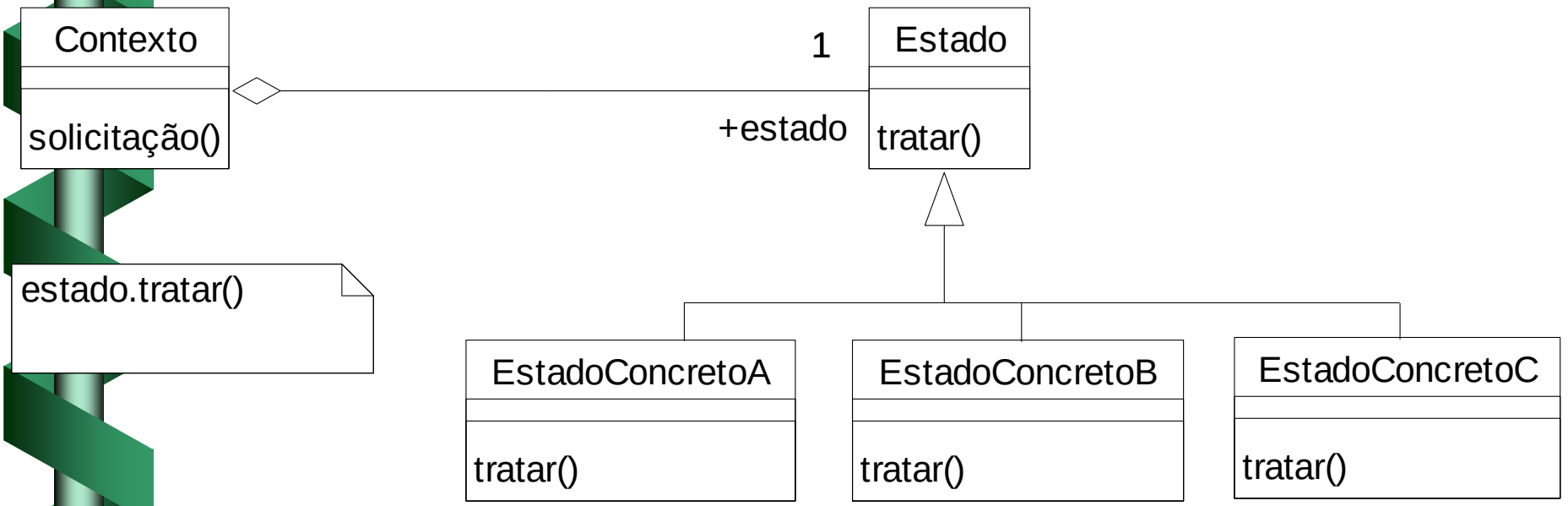
Padrão de Projeto: Estado

- ***Problema***
  - **Como evitar que comportamentos dependentes de estado fiquem espalhados pelas classes em diversos locais no código?**
- ***Forças***
  - **Tratar o comportamento de acordo com cada estado usando estruturas condicionais soluciona o problema, mas torna o programa difícil de manter e sujeito a erros humanos quando surgem novos estados, além de causar redundância de código.**
  - **Se for usada herança simplesmente, quando o estado do objeto mudar em tempo de execução, o objeto terá que mudar de classe, o que torna o código ainda mais complexo.**

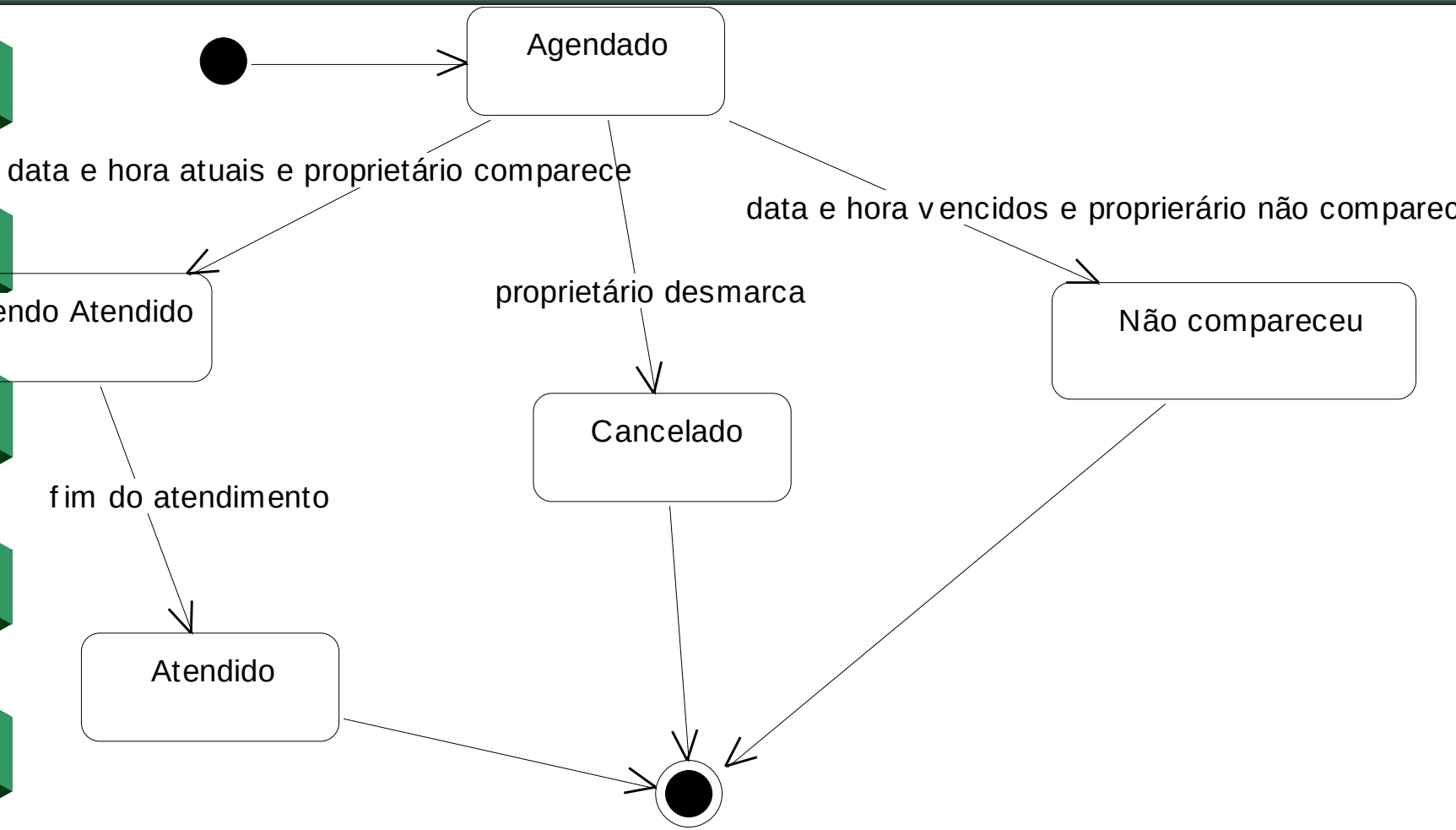
Padrão de Projeto: Estado

- ***Solução***
  - **Criar uma classe para representar o objeto em si e outra para representar seu estado**
  - **O estado pode variar independentemente de outros objetos.**

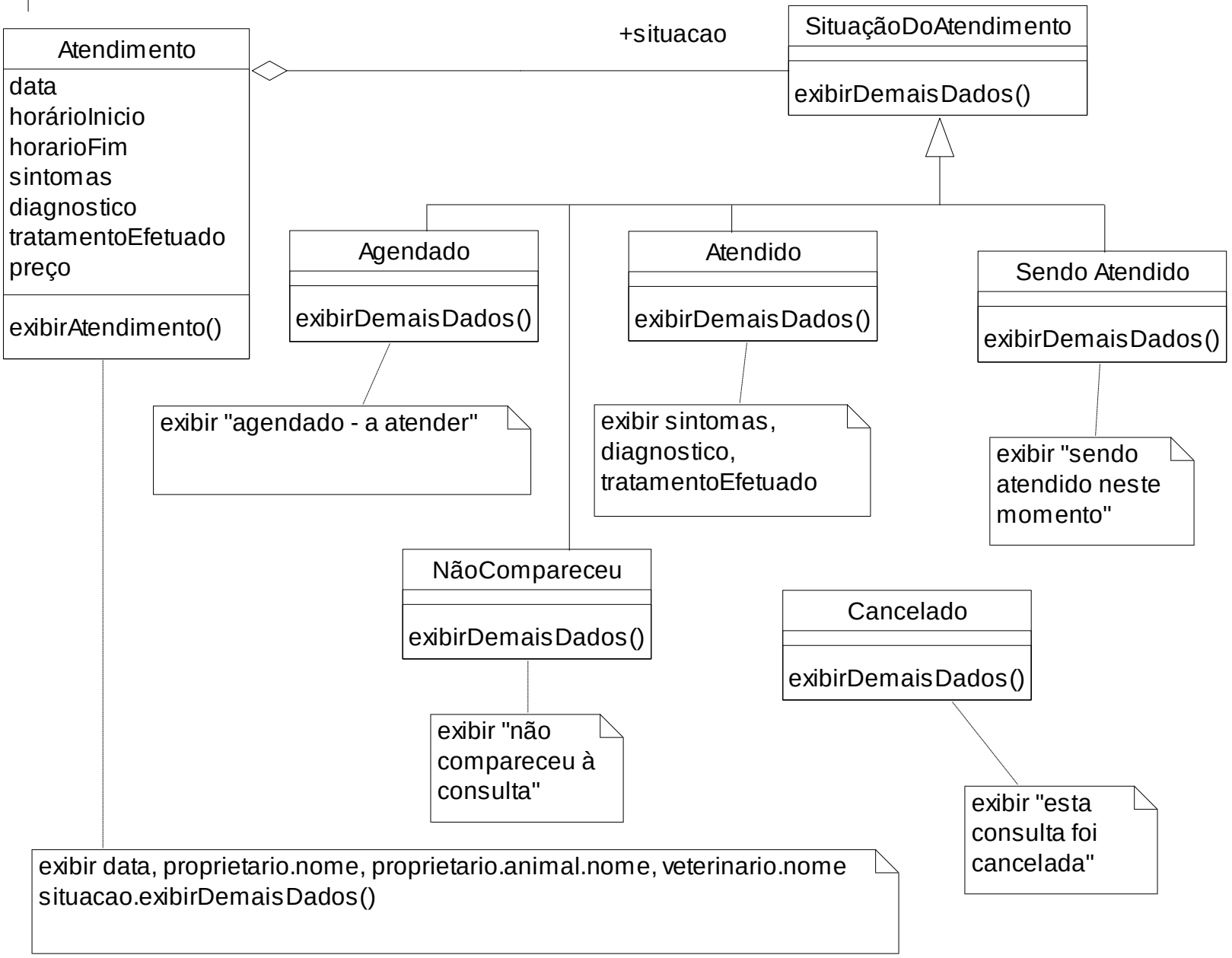
Padrão de Projeto: Estado



Exemplo de aplicação do padrão Estado



Exemplo de aplicação do padrão Estado



Padrão de Projeto: Iterador

- **Sua função básica é unificar a iteração por quaisquer tipos de coleções, de forma que o código fique padronizado em relação às operações básicas necessárias para percorrer e acessar coleções.**
- **Utilizado na maioria dos projetos OO e, por isso, já possui implementação disponível em diversas linguagens (Java, C#, etc.).**

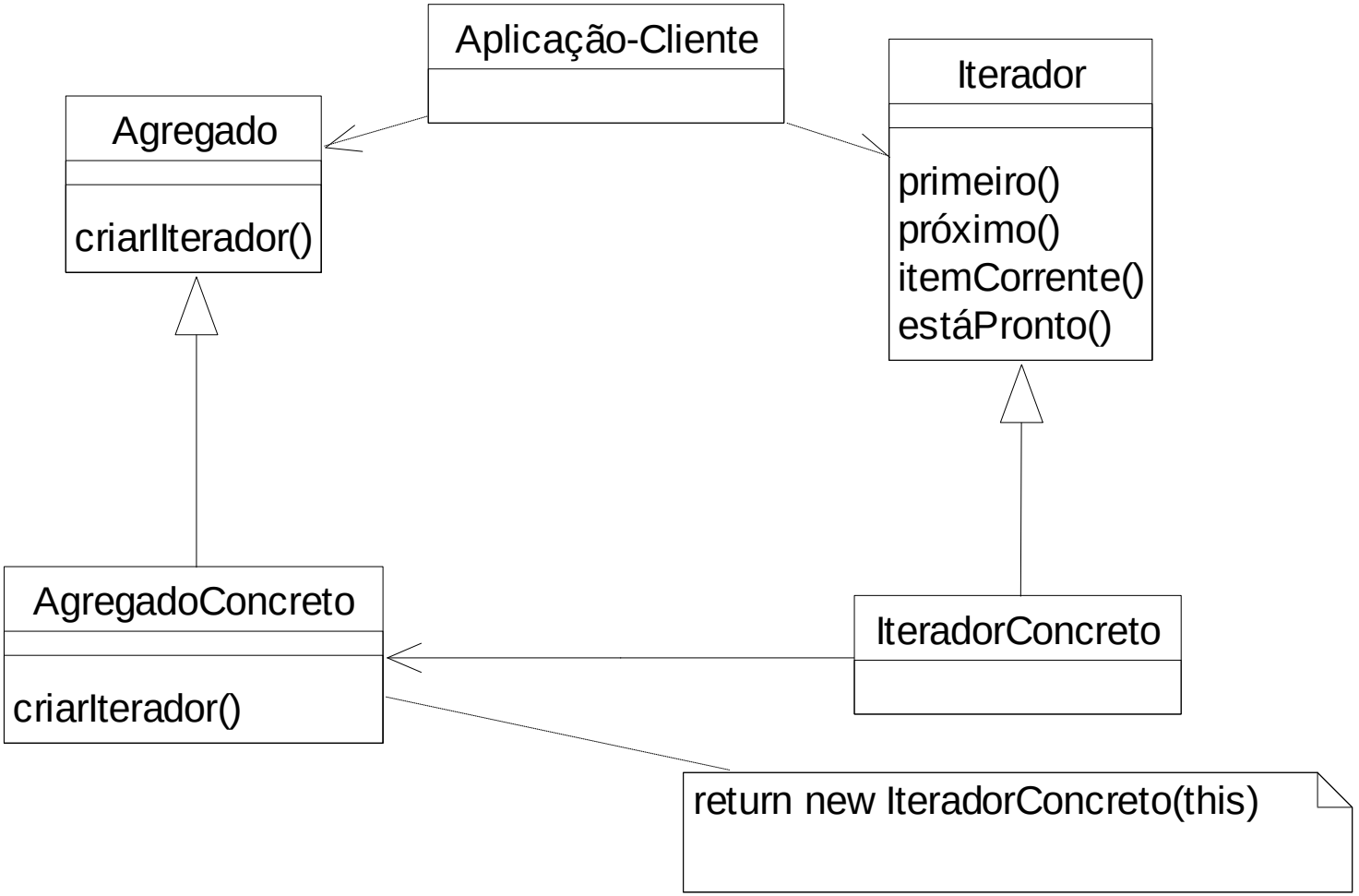
Padrão de Projeto: Iterador

- ***Problema***
  - **Existe a necessidade de percorrer agregados quaisquer, em diversas ordens, mas como fazer isso sem precisar saber detalhes de sua representação subjacente?**
- ***Forças***
  - **Incluir muitos métodos para cuidar da iteração sobre coleções pode poluir a interface.**
  - **Se for desejável percorrer o agregado de diferentes formas, ainda mais métodos serão necessários, além de variáveis para controlar cada trajeto.**

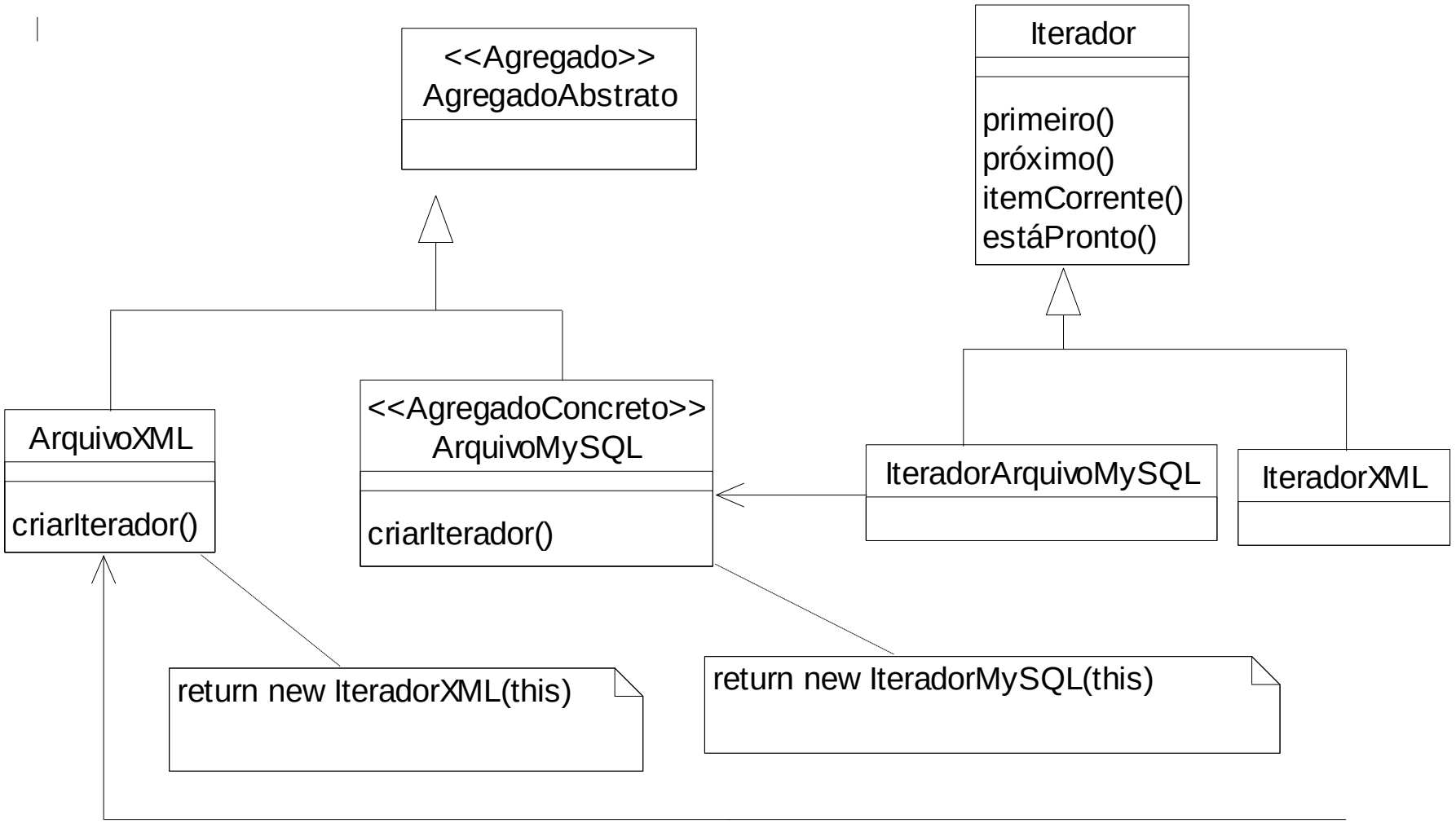
Padrão de Projeto: Iterador

- ***Solução***
  - **criar uma classe abstrata Iterador, que terá a interface de comunicação com a aplicação-cliente.**
  - **para cada agregado concreto, criar uma subclasse de Iterador, contendo a implementação dos métodos necessários para percorrer o agregado.**
  - **operações básicas de iteração: primeiro, próximo, itemCorrente e estáPronto.**
  - **O IteradorConcreto é o iterador que realiza efetivamente o trajeto no agregado real.**

Padrão de Projeto: Iterador



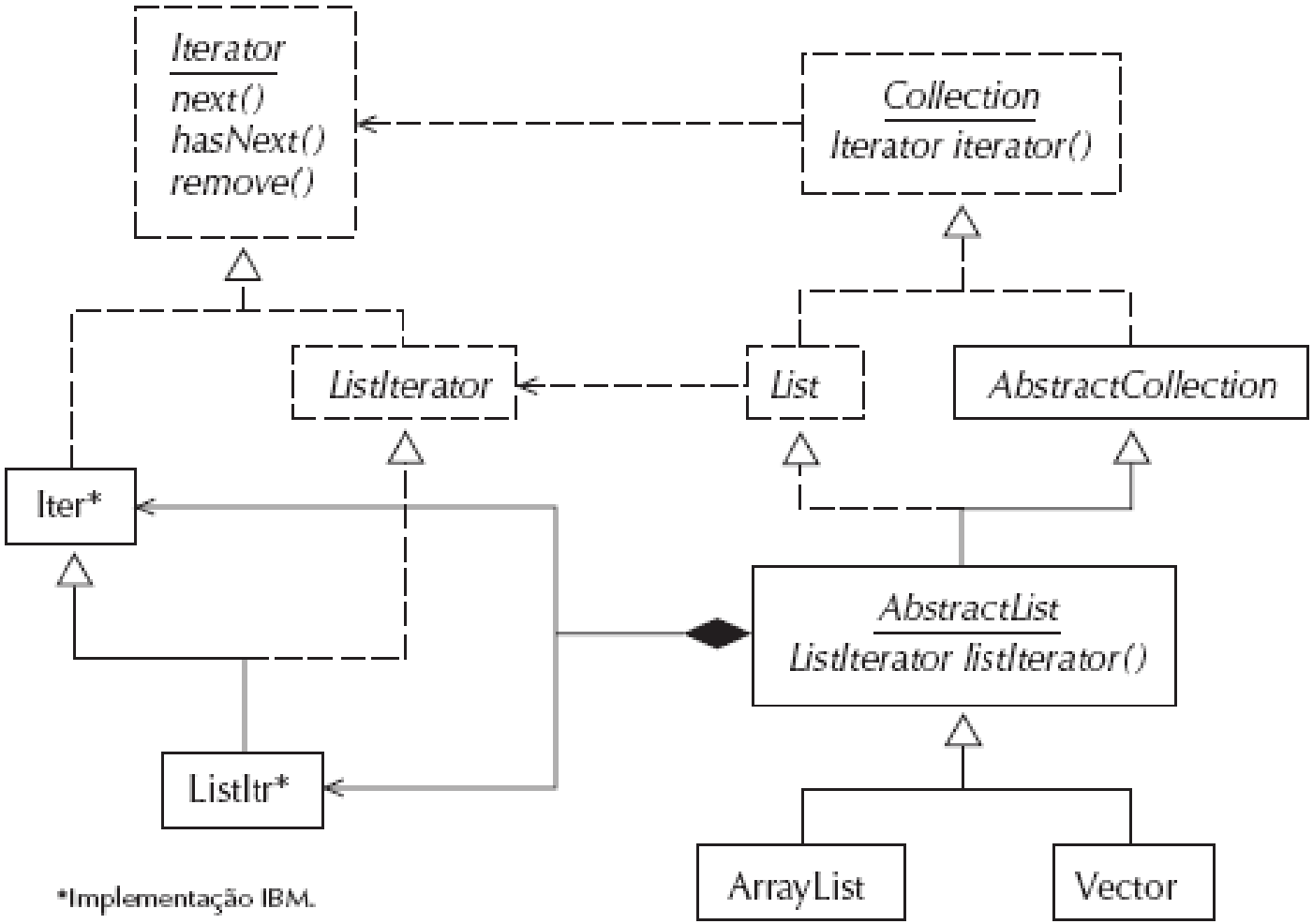
Aplicação do padrão Iterador



Iterador na API Java

- O pacote *java.util* contém uma interface *Iterator* com os métodos:*next(), hasNext()* e *remove().*
- O método *next()* é uma combinação de *proximo()* e *itemCorrente()*
- A subinterface *ListIterator* especifica esses métodos para operarem nos objetos *List* e adiciona os métodos apropriados aos objetos *List*.
- A interface *List* tem um método *listIterator(),* que retorna um objeto *ListIterator* que itera sobre ele. Isso permite ao programador mover-se e iterar sobre objetos *List*.

Iterador na API Java



Outros Padrões GoF: Composto (*Composite*)

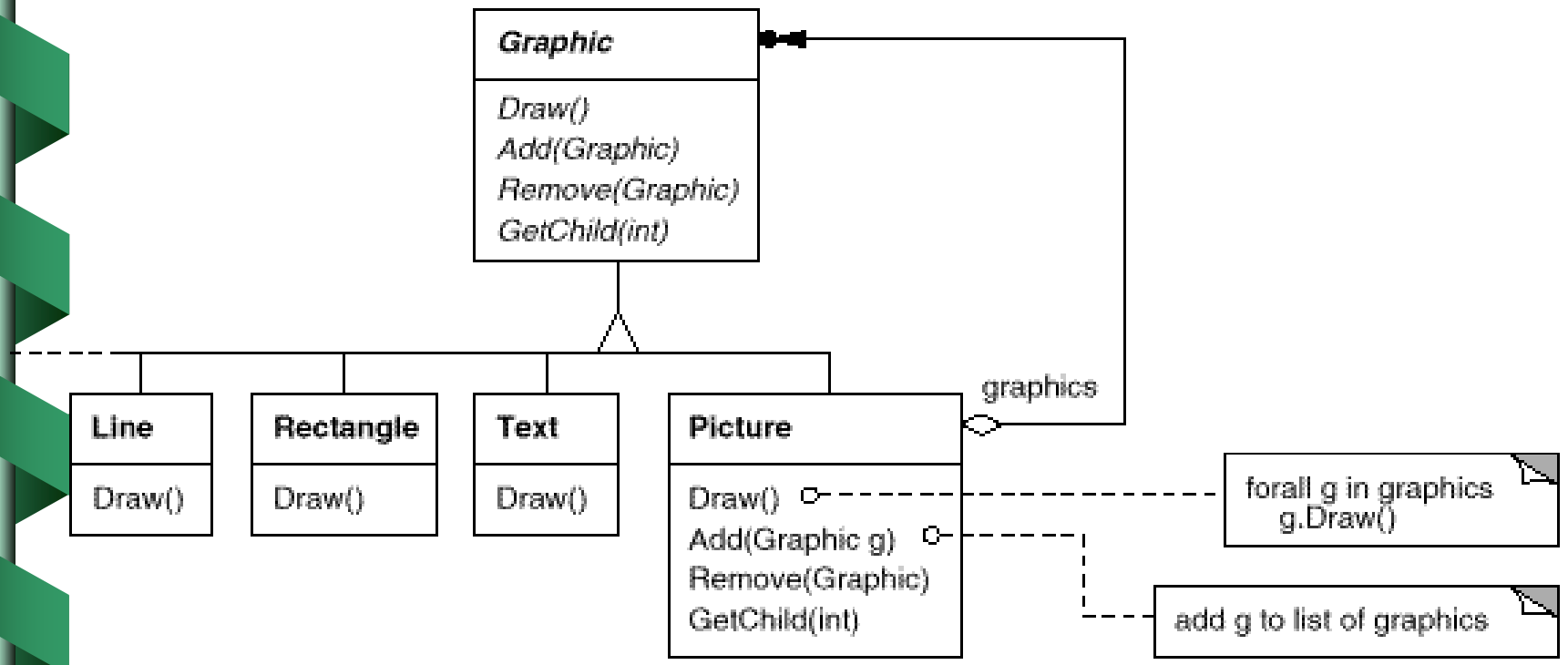
- **Composto (Objeto Estrutural)**
  - **Intenção (*Intent*)**
    - **compõe objetos em estruturas de árvore para representar hierarquias *part-whole*. Composto deixa o cliente tratar objetos individuais e composição de objetos uniformemente.**

Padrões de Projeto: Composto

- **Motivação (*Motivation*)**
  - **Editores gráficos permitem aos usuários construir diagramas complexos, agrupando componentes simples**
    - **Implementação simples: definir uma classe para primitivas gráficas tais como Texto, Linhas e outras classes que agem como depósitos (*containers*) para essas primitivas**
    - **Problema: Código que usa essas classes deve tratar primitivas e objetos do depósito diferentemente, tornando a aplicação mais complexa**

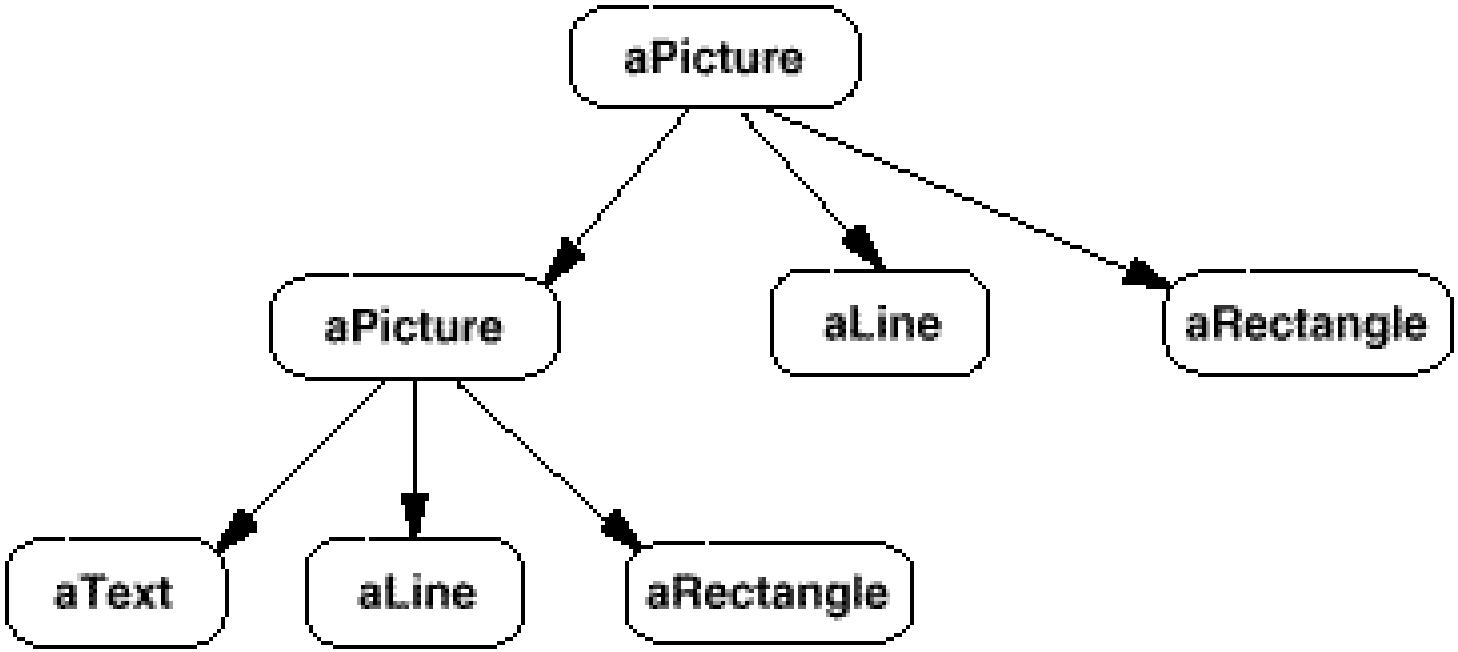
Padrões de Projeto: Composto

- **O Composto cria uma classe abstrata que representa primitivas e seus depósitos.**



Padrões de Projeto: Composto

- **Exemplo de composição recursiva de objetos**

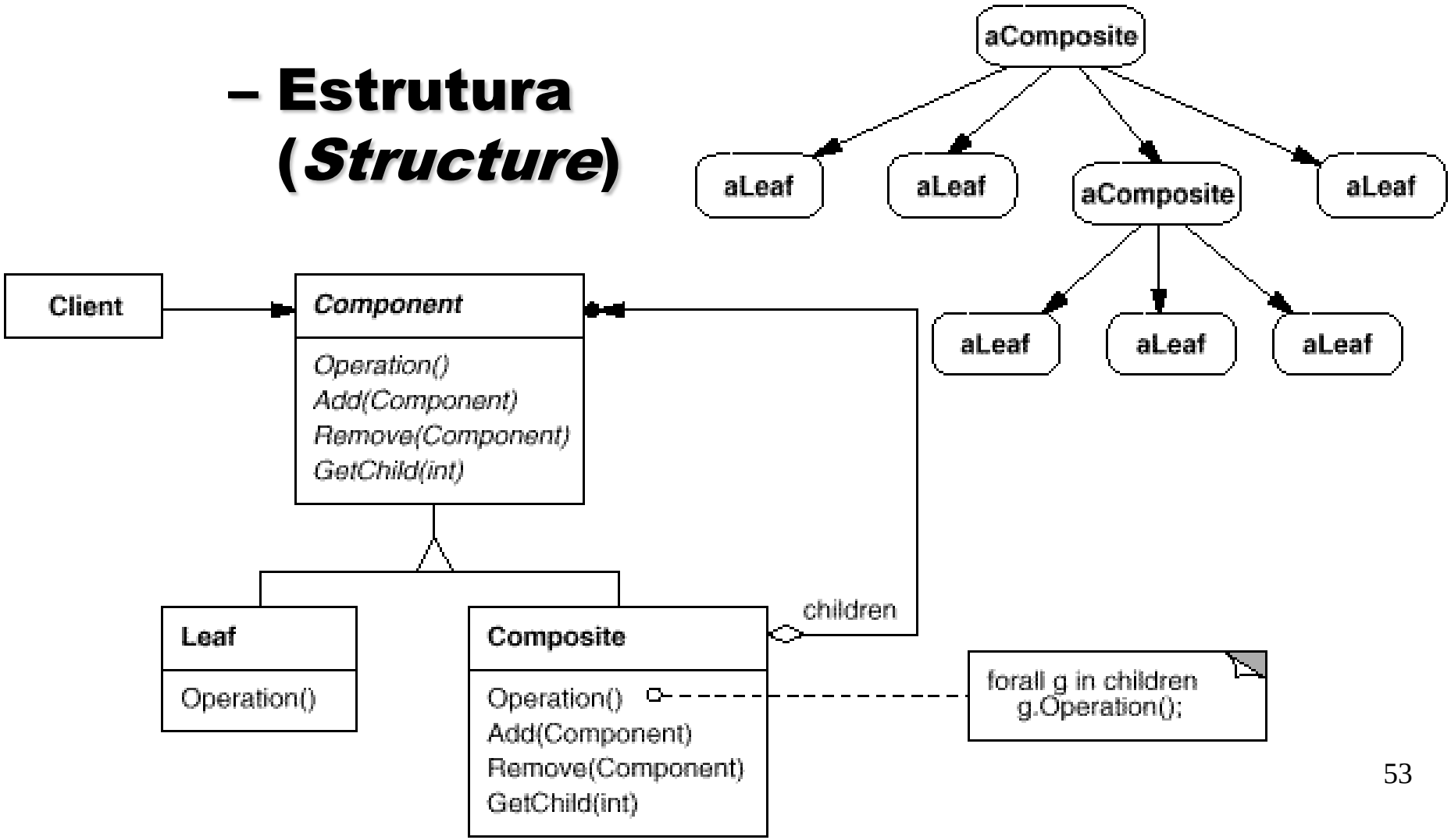


Padrões de Projeto: Composto

- **Aplicabilidade (*Applicability*)**
  - **representar hierarquias de objetos *part-whole***
  - **permitir aos usuários ignorar a diferença entre composições de objetos e objetos individuais. Todos os objetos na estrutura são tratados uniformemente**

Padrões de Projeto: Composto

- **Estrutura (*Structure*)**



Padrões de Projeto: Composto

- Participantes (*Participants*)
  - Component (Grafic)
    - declara a interface para os objetos na composição
    - implementa o comportamento padrão para a interface comum de todas as classes, quando apropriado
    - declara uma interface para acessar e gerenciar os componentes filho
    - define uma interface para acessar o pai de um componente na estrutura recursiva, implementado-o se for apropriado

Padrões de Projeto: Composto

- **Leaf (Rectangle, Line, Text, etc.)**
  - **representa objetos “folha” na composição. Uma folha não tem filhos**
  - **define o comportamento para objetos primitivos na composição**
- **Composite (Picture)**
  - **define o comportamento para componentes que têm filhos**
  - **armazena componentes filho**
  - **implementa operações relacionadas aos filhos na interface Component**
- **Client**
  - **manipula objetos na composição pelo através da interface Component**

Padrões de Projeto: Composto

- Colaboradores (*Collaborations*)
  - Clients usam a interface Component para interagir com objetos na estrutura composta.
  - Se o receptor é uma folha então o pedido é manipulado diretamente
  - Se o receptor é um Composite então os pedidos são enviados para seus componentes filhos

Padrões de Projeto: Composto

- Conseqüências (*Consequences*)
  - define hierarquias de classes que consistem de objetos primitivos e compostos
  - simplifica o cliente. Clientes podem tratar estruturas compostas e objetos individuais de maneira uniforme
  - facilita a adição de novos componentes

Padrões de Projeto: Composto

– Exemplo de Código (*Sample Code*)

```
class Equipment {
   public:
      virtual ~Equipment();

      const char* Name() { return _name; }

      virtual Watt Power();
      virtual Currency NetPrice();
      virtual Currency DiscountPrice();

      virtual void Add(Equipment*);
      virtual void Remove(Equipment*);
      virtual Iterator* CreateIterator();
   protected:
      Equipment(const char*);
   private:
      const char* _name;
   };
```

```
class CompositeEquipment : public Equipment {
   public:
      virtual ~CompositeEquipment();

      virtual Watt Power();
      virtual Currency NetPrice();
      virtual Currency DiscountPrice();

      virtual void Add(Equipment*);
      virtual void Remove(Equipment*);
      virtual Iterator* CreateIterator();

   protected:
      CompositeEquipment(const char*);
   private:
      List _equipment;
   };
```

Padrões de Projeto: Composto

```
class FloppyDisk : public Equipment {
   public:
      FloppyDisk(const char*);
      virtual ~FloppyDisk();
      virtual Watt Power();
      virtual Currency NetPrice();
      virtual Currency DiscountPrice();
   };
```

```
class Chassis : public CompositeEquipment {
   public:
      Chassis(const char*);
      virtual ~Chassis();
      virtual Watt Power();
      virtual Currency NetPrice();
      virtual Currency DiscountPrice();
   };
```

Nós folha

```
Currency CompositeEquipment::NetPrice () {
      Iterator* i = CreateIterator();
      Currency total = 0;
      for (i->First(); !i->IsDone(); i->Next()) {
         total += i->CurrentItem()->NetPrice();
      }
      delete i;
      return total;
   }
```

Método do composto

Padrões de Projeto: Composto

```
Cabinet* cabinet = new Cabinet("PC Cabinet");
    Chassis* chassis = new Chassis("PC Chassis");
    cabinet->Add(chassis);
    Bus* bus = new Bus("MCA Bus");
    bus->Add(new Card("16Mbs Token Ring"));

    chassis->Add(bus);
    chassis->Add(new FloppyDisk("3.5in Floppy"));

    cout << "The net price is " << chassis->NetPrice() << endl;
```

Programa principal (cliente)

Padrões de Projeto: Composto

- Usos Conhecidos (*Known Uses*)
  - Presente em quase todos os sistemas OO
  - A classe original View do MVC
  - RTL Smalltalk *compiler framework*
  - Etc.

Padrões de Projeto: Composto

- Padrões Relacionados (*Related Patterns*)
  - Chain of Responsibility
  - Decorator
  - Flyweight
  - Iterator
  - Visitor

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).**

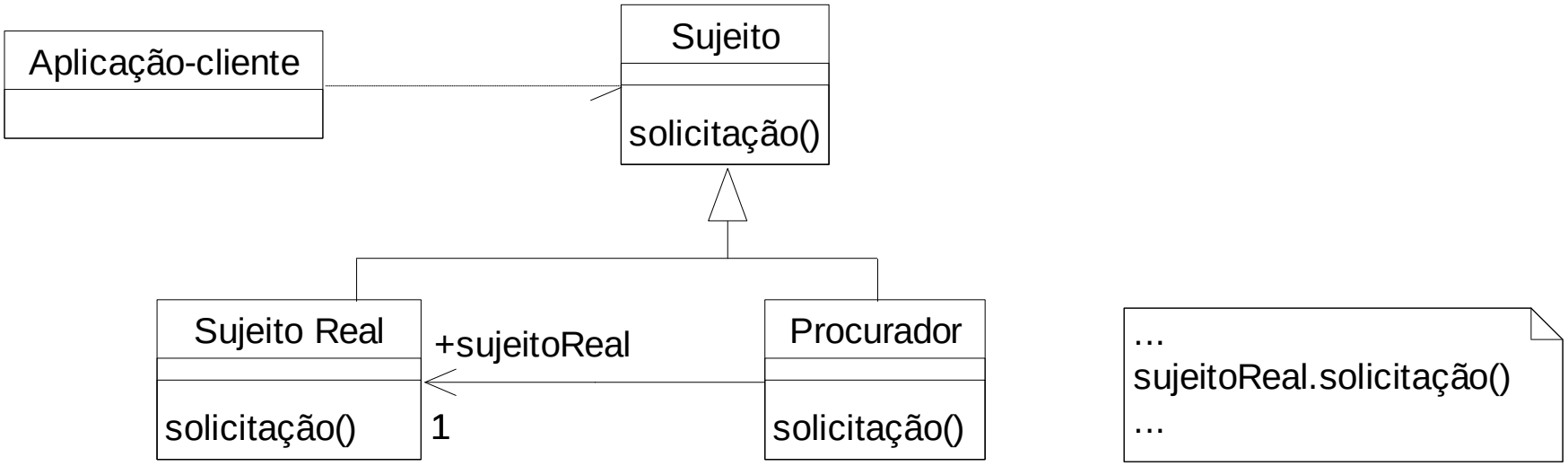
Padrão de Projeto: Procurador

- ***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**

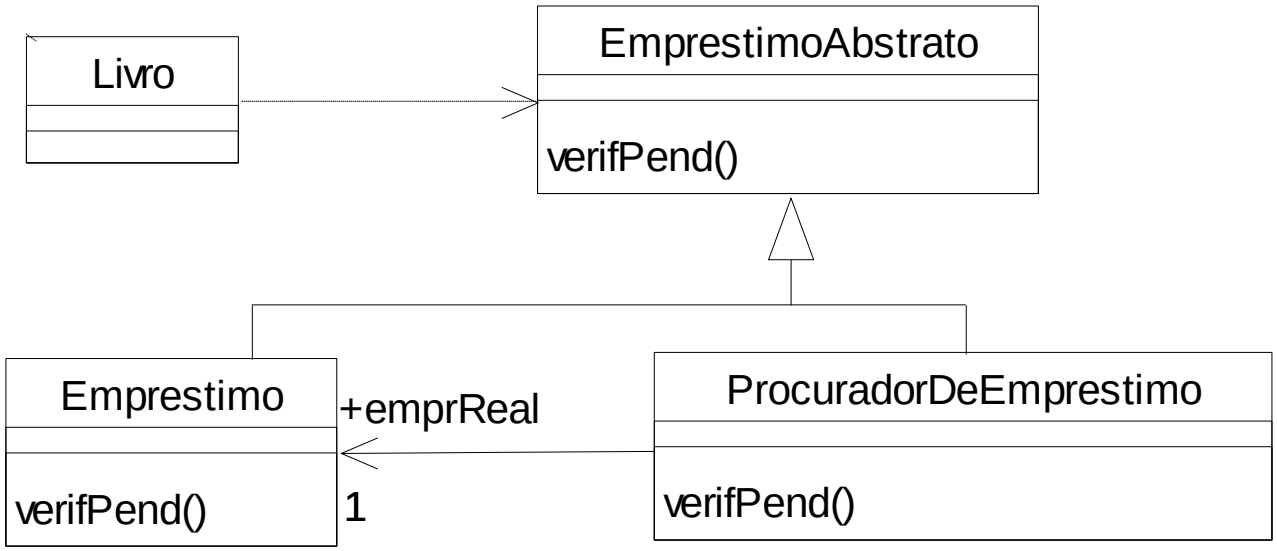
Exemplo de aplicação do padrão Procurador

- 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




```
...
emprReal.verifPend()
...
```

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.**

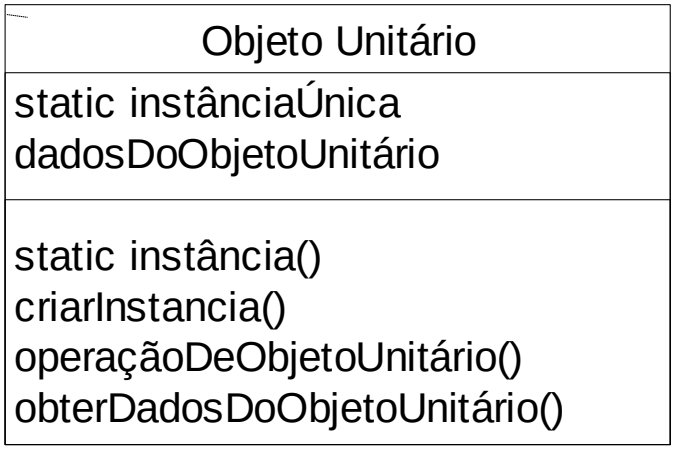
Padrão de Projeto: Objeto Unitário

- ***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.**

Padrão de Projeto: Objeto Unitário




```
if instancia == null
    criarInstancia;
return instancia
```

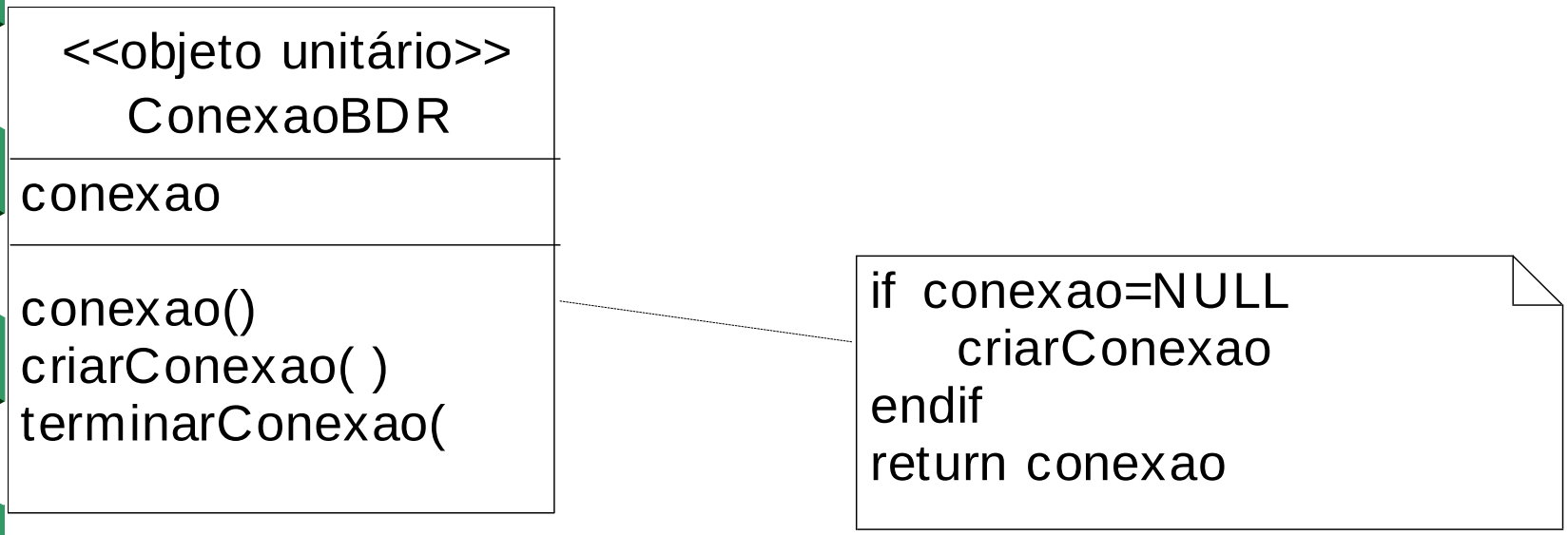
Exemplo de aplicação de padrões de projeto

- **Problema: Ao iniciar uma aplicação, será necessário estabelecer uma conexão com a base de dados, para ter um canal de comunicação aberto durante a execução da aplicação.**
  - **Em geral, isso é implementado por meio de uma classe Conexão (várias interfaces de BDRs existentes fornecem uma classe Connection especialmente para este fim).**
  - **Esta conexão deve posteriormente ser fechada, no momento do término da aplicação.**

Exemplo de aplicação do padrão Objeto Unitário

- A conexão com o BDR deve preferencialmente ser única, pois se cada vez que um objeto for criado uma nova conexão for criada, inúmeros objetos existirão na memória, desnecessariamente.
- Assim, pode-se aplicar o padrão Objeto Unitário para garantir que, para cada aplicação sendo executada, uma única conexão válida com o BDR existe.

Exemplo de aplicação do padrão Objeto Unitário



Exemplo

- **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.**

```java
// 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 ums 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();
    }
}
```

Conferências sobre Padrões - PLoP

- **Submissão de artigos: processo de "shepherding"**
- **Conferência: Sessões de "workshop do autor – *writers' workshop*", intercaladas por "jogos"**
- **Grupos com 6 a 8 autores e 6 a 8 não autores, 1 líder e 1 secretário.**

Conferências sobre Padrões - PLoP

- **Workshop:**
  - Líder pede a alguém para fazer um resumo do padrão
  - Autor tem 5 minutos para ler trechos do artigo para os demais participantes, que já leram de antemão.
  - Autor “congela”
  - 15 minutos: demais participantes levantam pontos fortes do padrão
  - 15 minutos: demais participantes apresentam sugestões para melhoria
  - Autor “descongela” e tem mais 10 minutos para perguntas e respostas.

Introdução aos Padrões

- Conferências sobre padrões
  - Primeira conferência PLoP – 1994 - EUA
  - Outras conferências:
    - EuroPLoP – Europa
    - ChiliPLoP – EUA (Arizona)
    - KoalaPLoP – Austrália
    - MensorePLoP – Japão
    - VikingPLoP - Escandinávia
  - No Brasil: SugarloafPLoP!!!

Allerton House – Monticello – Illinois - EUA

PLoP

EuroPLoP - Alemanha

2001: 1st Latin-American Conference on Pattern Languages of Programs

Rio de Janeiro

2005: 5th Latin-American Conference on Pattern Languages of Programs

Campos do Jordão – São Paulo

2007: 6th Latin-American Conference on Pattern Languages of Programs

Porto de Galinhas – PE

2008: 7th Latin-American Conference on Pattern Languages of Programs

Fortaleza – CE

2010: 8th Latin-American Conference on Pattern Languages of Programs

Salvador – BA

2012: 9th Latin-American Conference on Pattern Languages of Programs

Natal – RN

Discussão

- **Padrões documentam soluções para problemas que ocorrem com freqüência durante o desenvolvimento de software**
  - **aumento da produtividade e qualidade no desenvolvimento.**
- **A proliferação e divulgação de padrões, bem como de ferramentas que auxiliam o desenvolvimento com padrões, fará com que em pouco tempo o uso de padrões fique inerente ao processo de desenvolvimento.**

Discussão

- **O uso de padrões têm crescido visivelmente na indústria de software.**
  - **cada vez mais empresas fornecem aos funcionários recém-contratados cursos sobre padrões de projeto, padrões J2EE, padrões de persistência, padrões arquiteturais, etc.**
  - **Frameworks de apoio ao uso de padrões também estão se popularizando. Os próprios ambientes de desenvolvimento, tais como ambientes para Java [Core J2EE, 2001; Java 2002] e ferramentas CASE, como a Rational Rose [Rose, 2005; Forbrig et al., 2001], já disponibilizam formas de reusar padrões durante o desenvolvimento.**

Discussão

- **Pesquisas sobre padrões**
  - **Apresentação de novos padrões**
  - **Uso de padrões na indústria de software, explorando principalmente comparações relativas à produtividade e qualidade decorrentes do uso de padrões**
  - **Ferramentas para facilitar a recuperação e uso de padrões. Por exemplo**
    - **Repositório de Padrões**
    - **GREN-Wizard**

Mais informações

- www.hillside.net

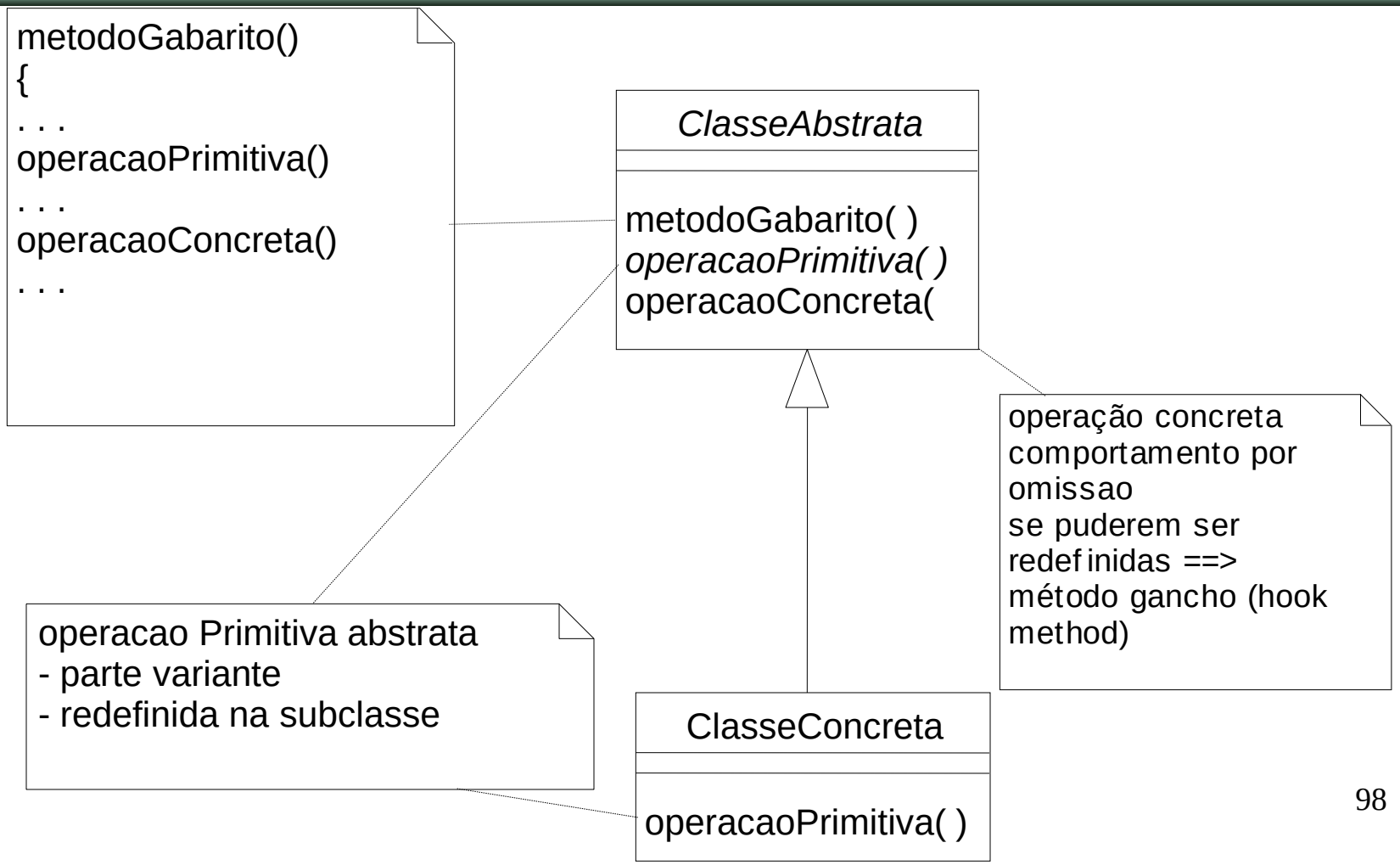
Referências

- Alexander, Christopher et. al. (1977) A Pattern Language. Oxford University Press.
- Alexander, Christopher (1979) The Timeless Way of Building. Oxford University Press.
- Beck, Kent; Cunningham, Ward (1987) Using Pattern Languages for Object-Oriented Programs, Technical Report nº CR-87-43,
- Coplien, J. O. (1996) Software Patterns. SIGS books and Multimedia, Junho.
- Cunningham, W. (1996) The CHECKS Pattern Language of Information Integrity, in Coplien & Schmidt (1996), p. 145-155.
- Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. (1995) Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley.
- MESZAROS, G.; DOBLE, J. A pattern language for pattern writing, cap. 29 in J. Coplien; D. Schmidt. Pattern Languages of Program Design, Reading-MA, Addison-Wesley, p. 529–574, 1998.
- (LER E ENTREGAR RESUMO PARA A PRÓXIMA AULA)
- BRAGA, Rosana T. V.; GERMANO, Fernão S. R.; MASIERO, Paulo C. A Pattern Language for Business Resource Management. In: 6th Annual Conference on Pattern Languages of Programs (PLoP 99), Monticello – Illinois, EUA, 1999. v. 7, p. 1-33.
- (LER PÁGINAS 1 a 5 + seções 2.2.1, 2.31 e 3.0)

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.**

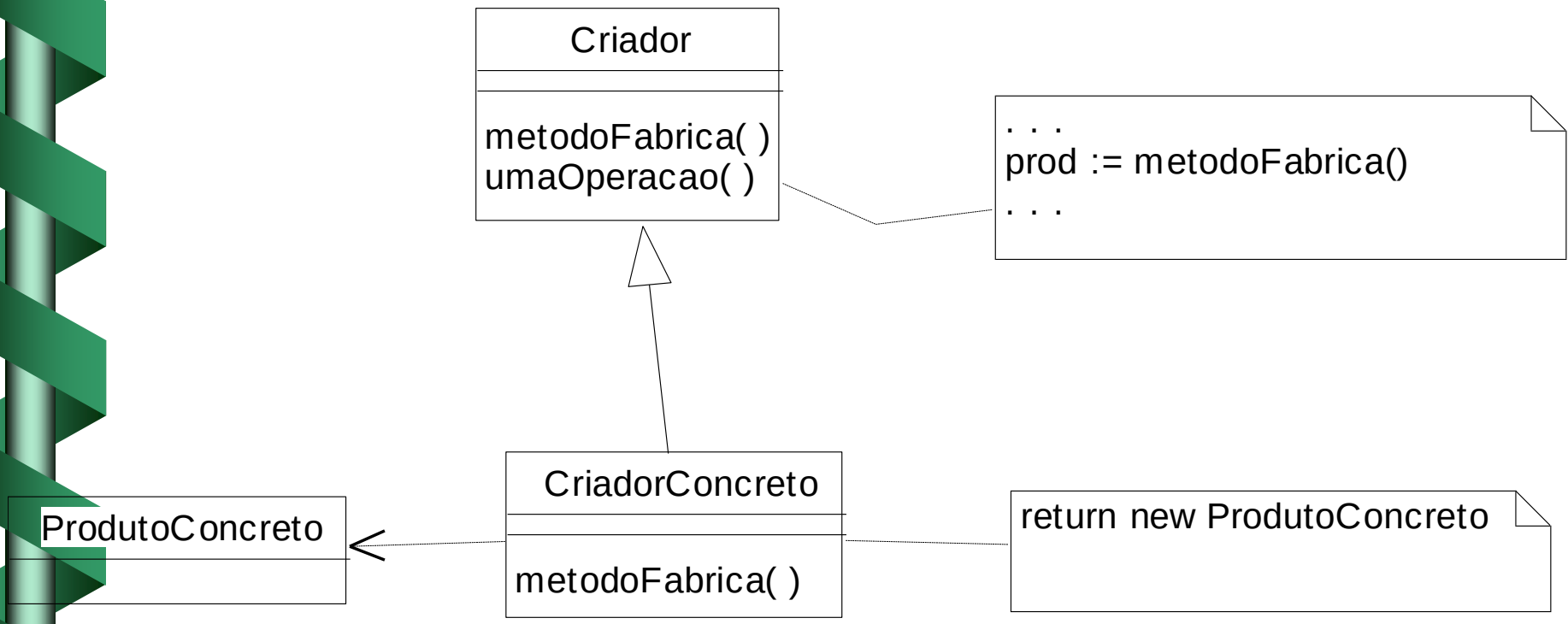
Padrão Método-gabarito



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: 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.

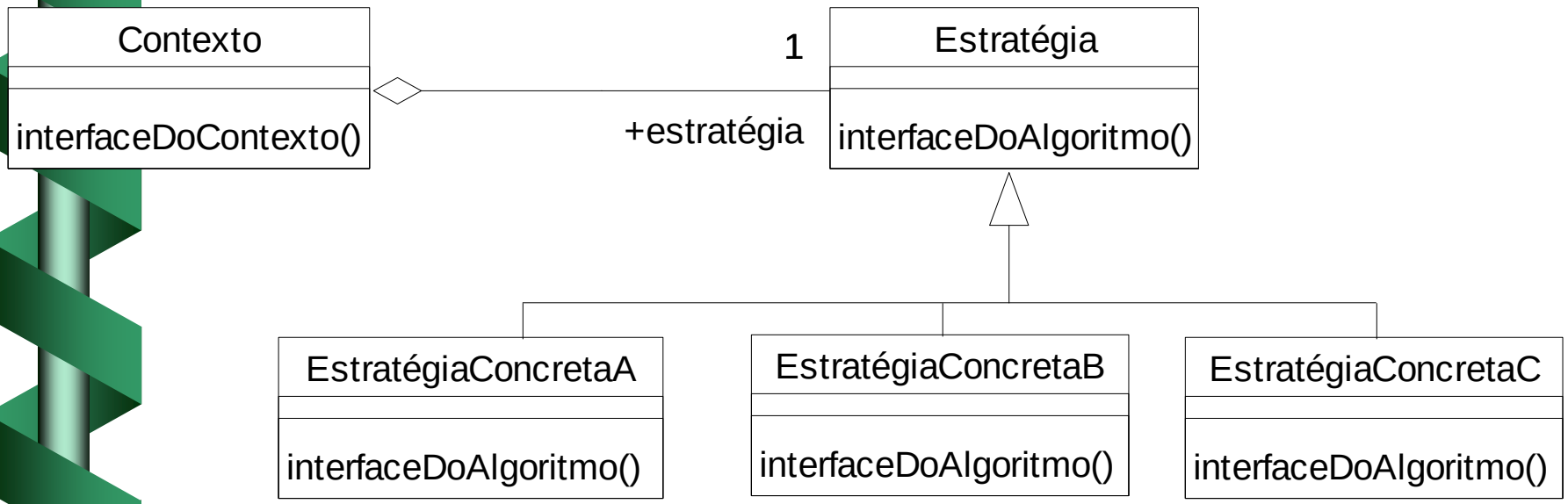
Padrão de Projeto: Estratégia

- ***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.**

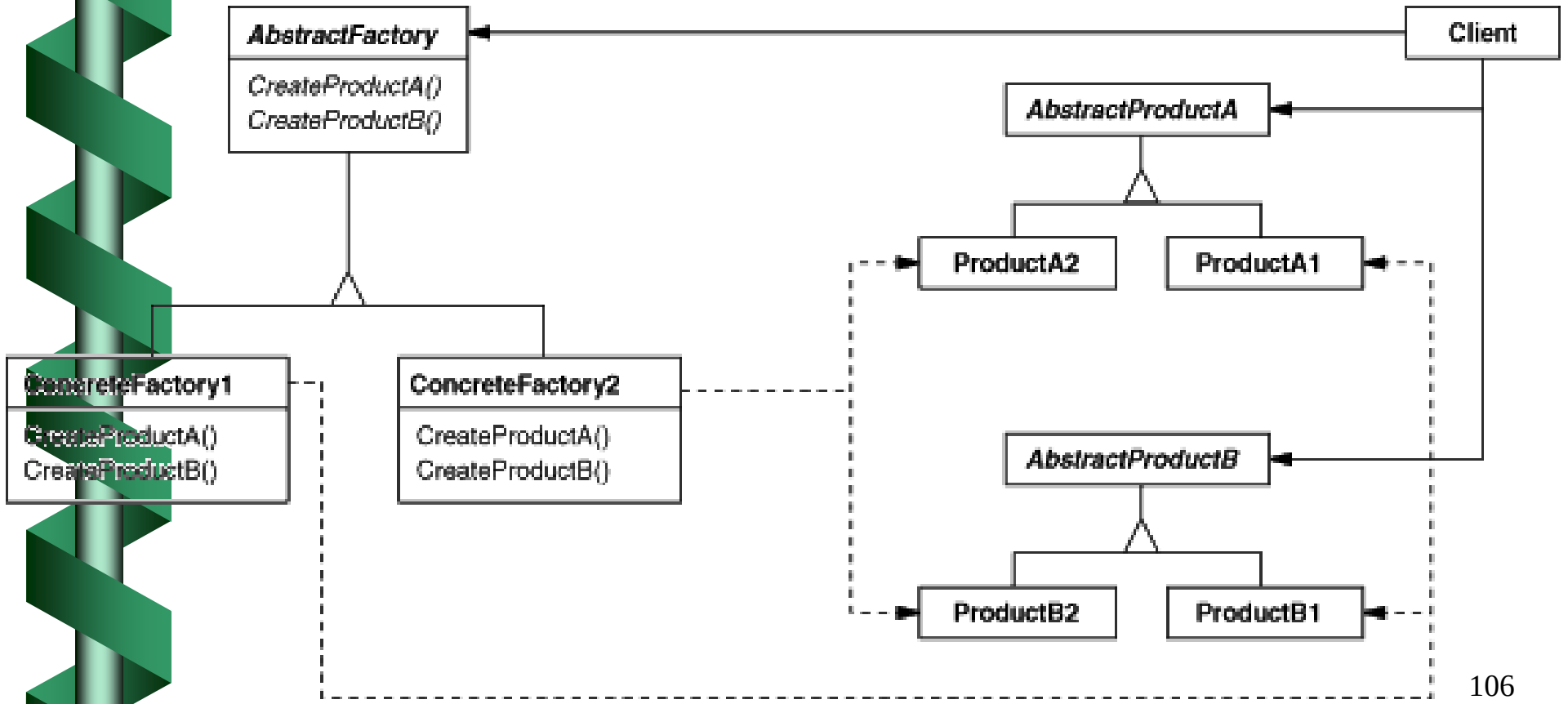
Padrão de Projeto: Estratégia



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

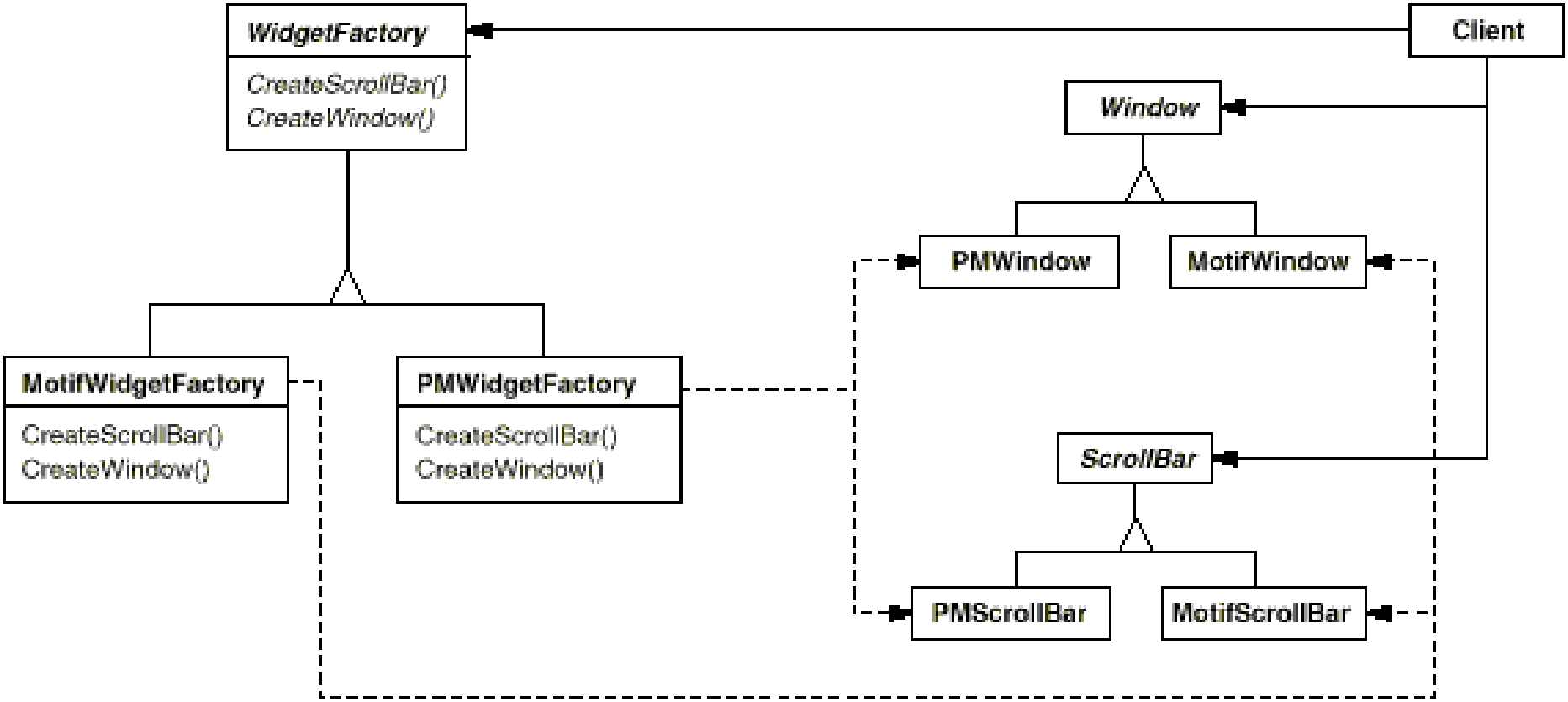
Abstract Factory



Abstract Factory

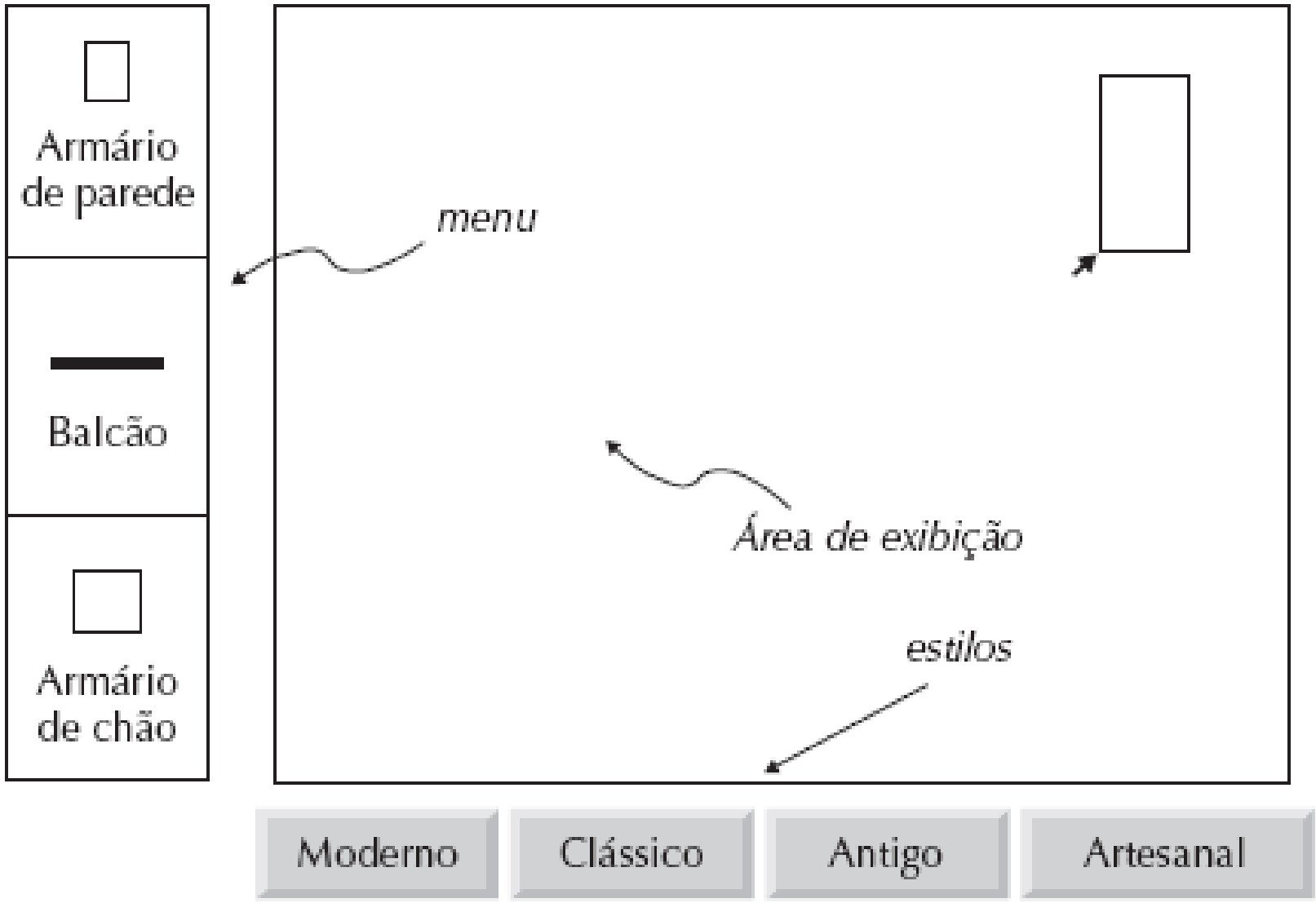
- É uma fábrica de objetos que retorna uma das várias fábricas.
- Uma aplicação clássica do Abstract Factory é o caso onde o seu sistema precisa de suporte a múltiplos tipos de interfaces gráficas, como Windows, Motif ou MacIntosh.
- A fábrica abstrata retorna uma outra fábrica de GUI que retorna objetos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

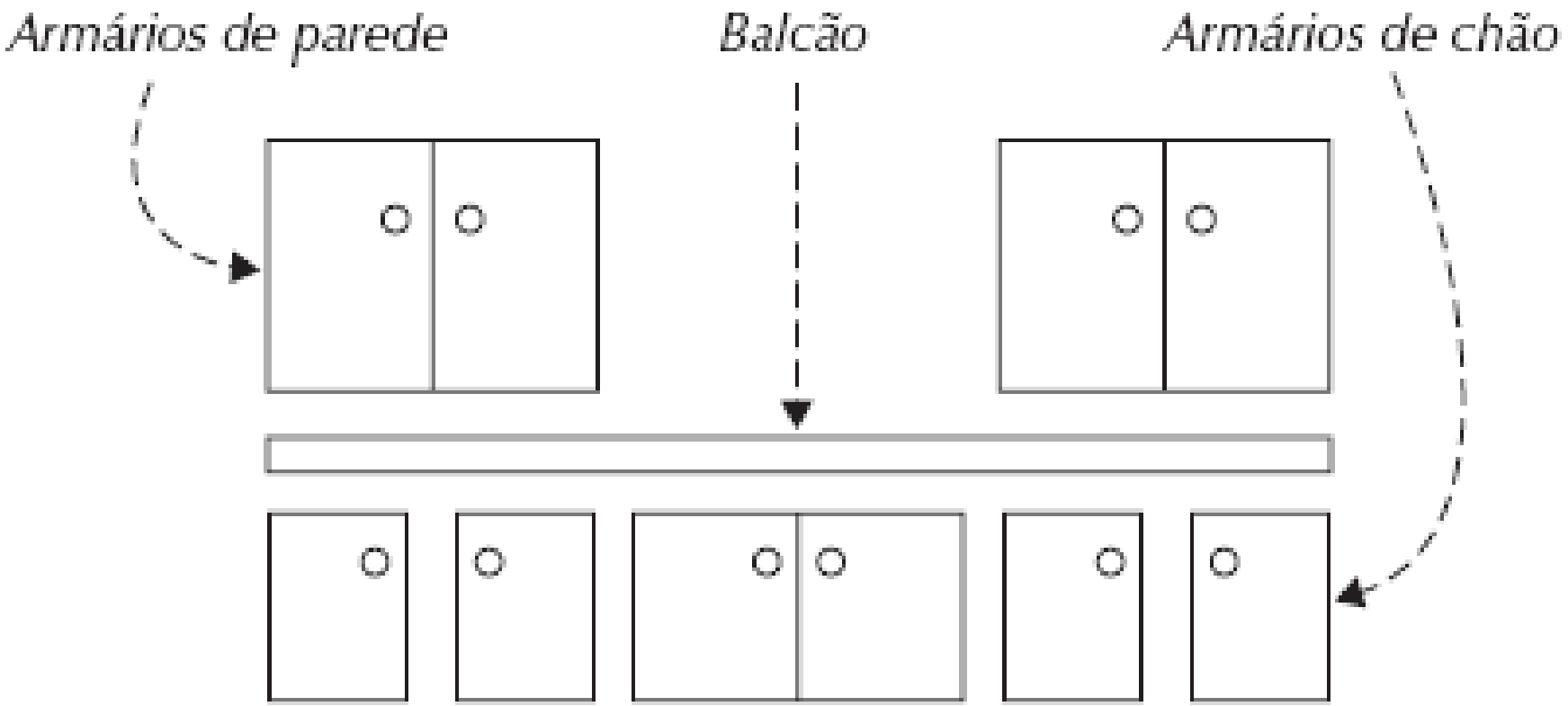
Exemplo: Abstract Factory



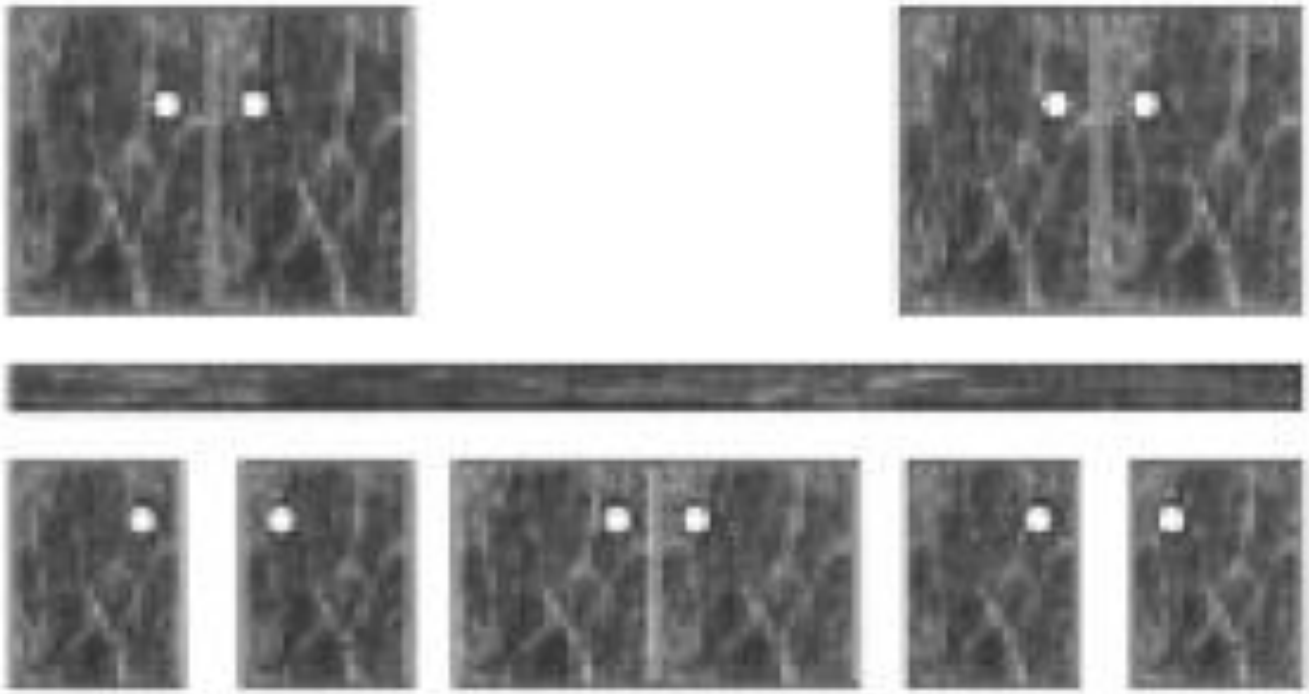
Exemplo: Aplicação para construir armários de cozinha

- **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.**


Armário de parede
Balcão
Armário de chão
menu
Área de exibição
estilos
Moderno
Clássico
Antigo
Artesanal

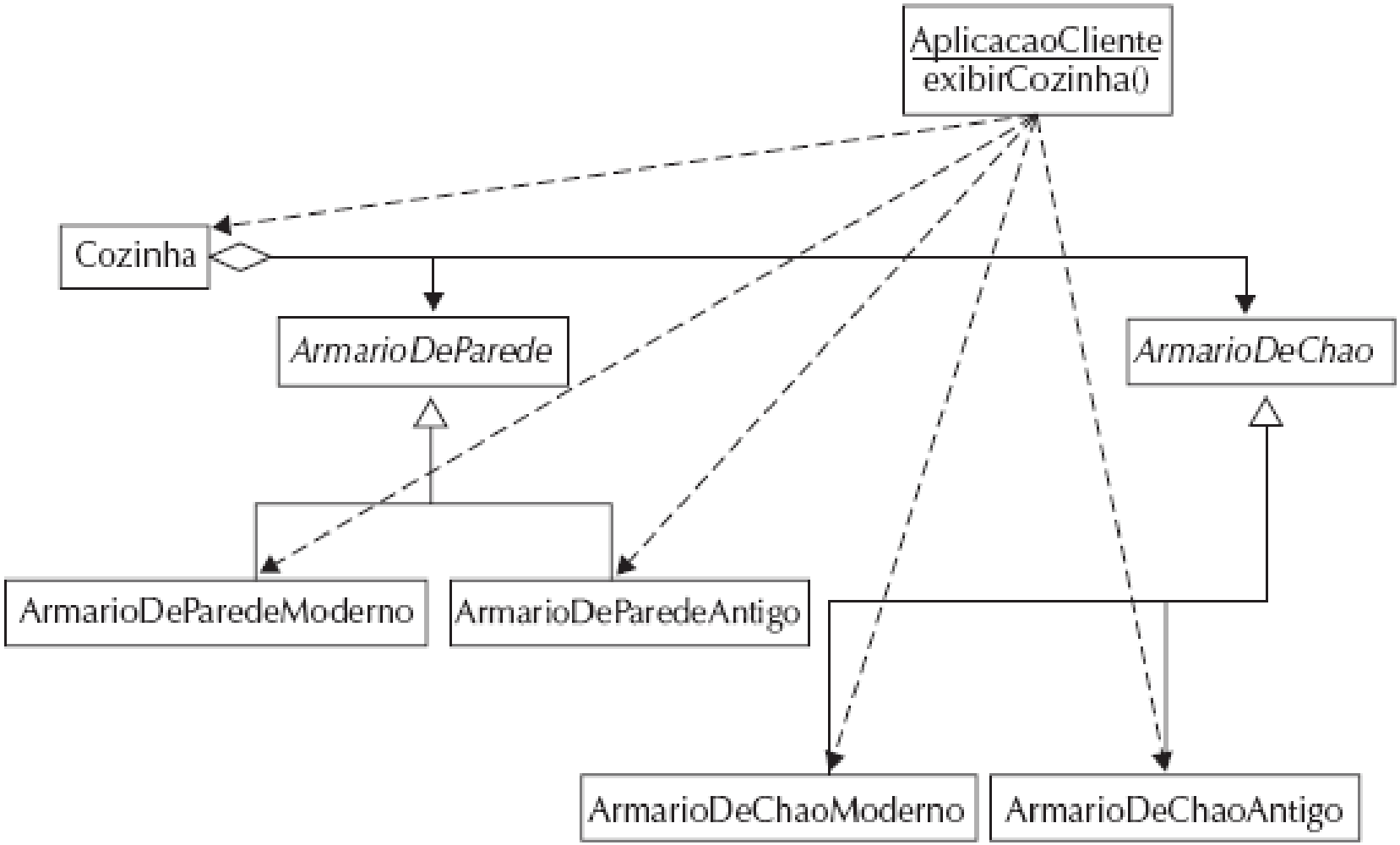


Moderno | Clássico | Antigo | Artesanal



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, . . . );
. . .
// exibe 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**

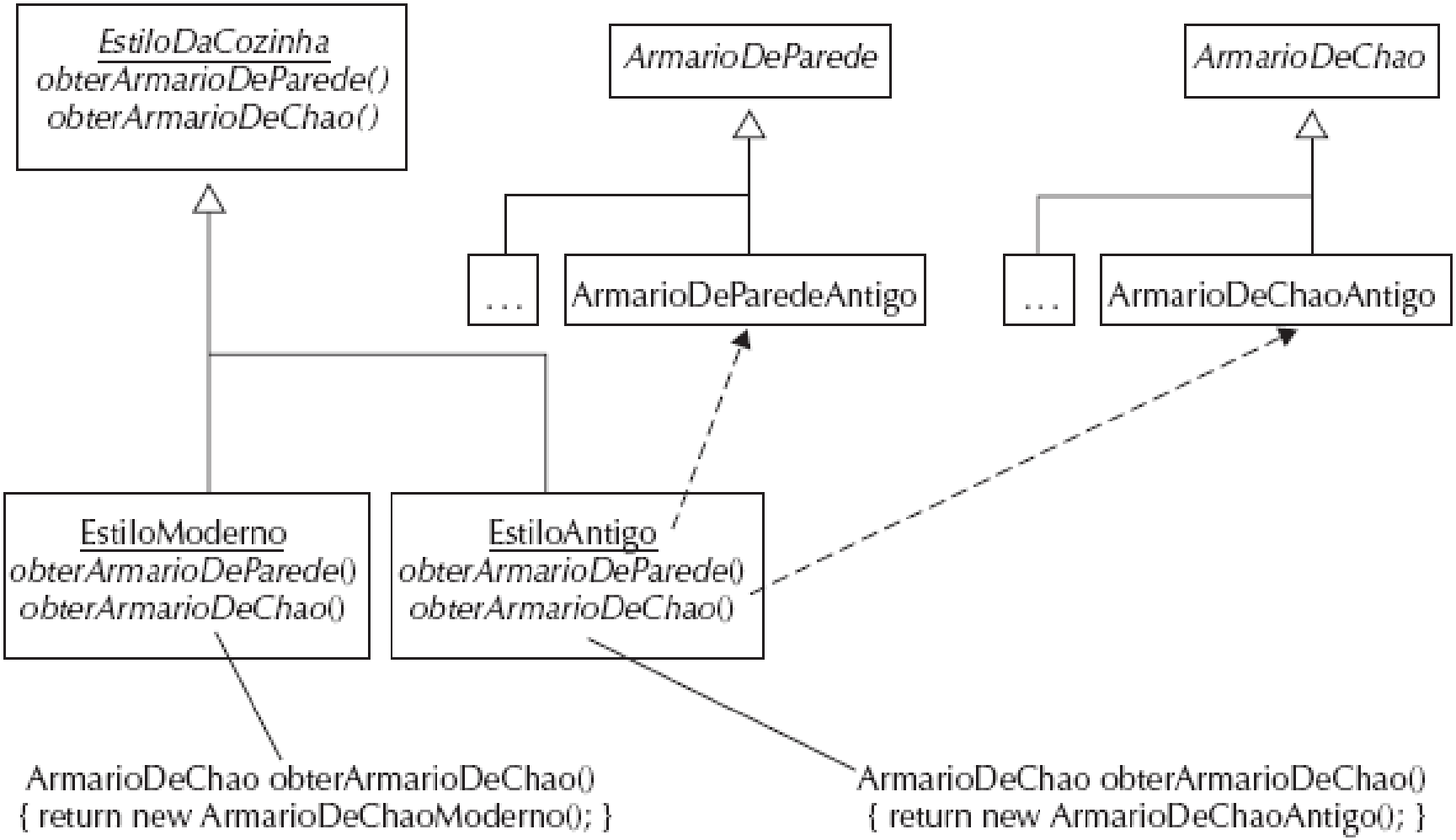
. . . new ArmarioDeParedeAntigo();

por versões delegadas a um parâmetro de estilo:

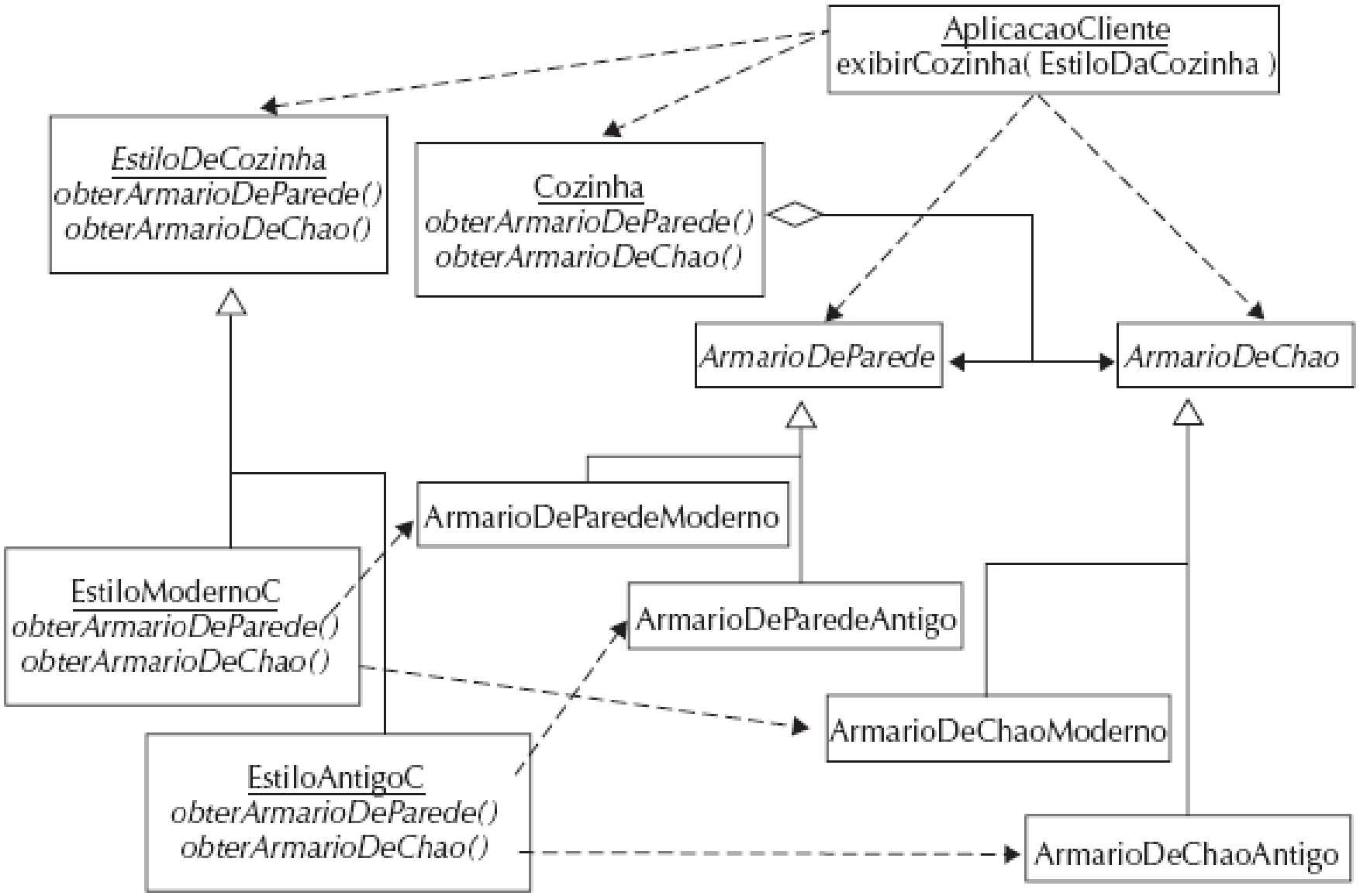
. . . meuEstilo.obterArmarioDeParede();

- **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 . . . );
. . .
```