

Padrões GoF – Iterator, State, Singleton, Observer e Composite



Análise e Projeto Orientados a Objetos

Profa Dra Rosana T. V. Braga



Alguns Padrões GoF

- Singleton
- Iterator
- State
- Observer
- Composite



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 **idades** 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 frequência
- 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 frequência no desenvolvimento de software (*Gamma 95*)



Padrões de Software

■ Vantagens de Padrões?

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



Exemplo: 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

		Propósito		
		De Criação	Estrutural	Comportamental
Escopo	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: 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.

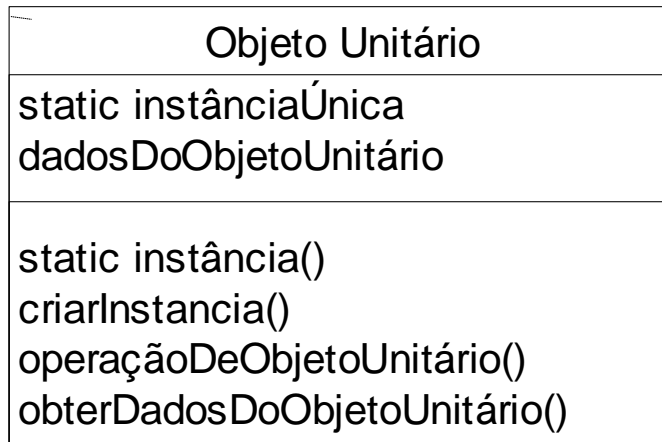


Padrão de Projeto: Objeto Unitário

■ *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



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



Exemplo de aplicação de padrões de projeto

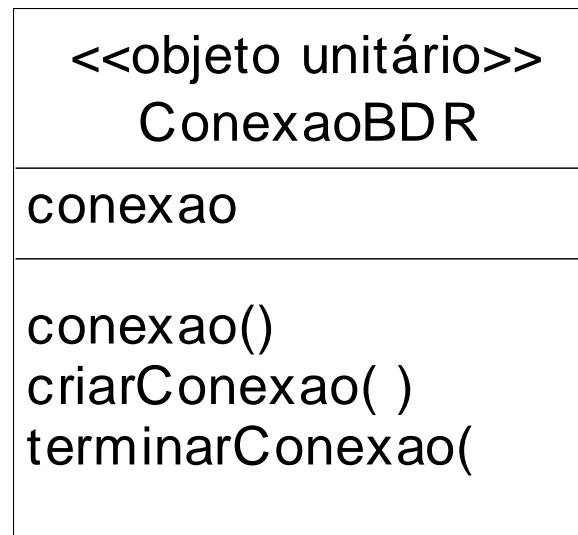
- **Problema:** Ao iniciar uma aplicação, será necessário estabelecer uma conexão com a base de dados, para ter um canal de comunicação aberto durante a execução da aplicação.
 - Em geral, isso é implementado por meio de uma classe Conexão (várias interfaces de BDRs existentes fornecem uma classe Connection 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



```
if conexao=NULL
    criarConexao
endif
return conexao
```




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.

```
// 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();
    }
}
```



Padrão Iterator

- Nome: **Iterator** (Gamma 94)
- Objetivo:
 - Existe a necessidade de percorrer agregados quaisquer, em diversas ordens, sem conhecer sua representação subjacente
- Motivação:
 - além de acessar os elementos sem conhecer a estrutura interna do agregado, pode ser desejável percorrer o agregado de diferentes formas, sem poluir a interface com inúmeros métodos para isso e mantendo controle do percurso.



Padrão Iterator

- Solução: Criar uma classe abstrata Iterator, que terá a interface de comunicação com o cliente. Para cada agregado concreto, criar uma subclasse de Iterator, contendo a implementação dos métodos necessários



Padrão Iterator

- Aplicabilidade
- Use o padrão Iterator
 - Para acessar o conteúdo de um objeto agregado sem expor sua representação interna.
 - Para apoiar diferentes trajetos de objetos agregados.
 - Para conseguir uma interface uniforme para percorrer diferentes estruturas de agregados (isto é, para apoiar a iteração polimórfica).

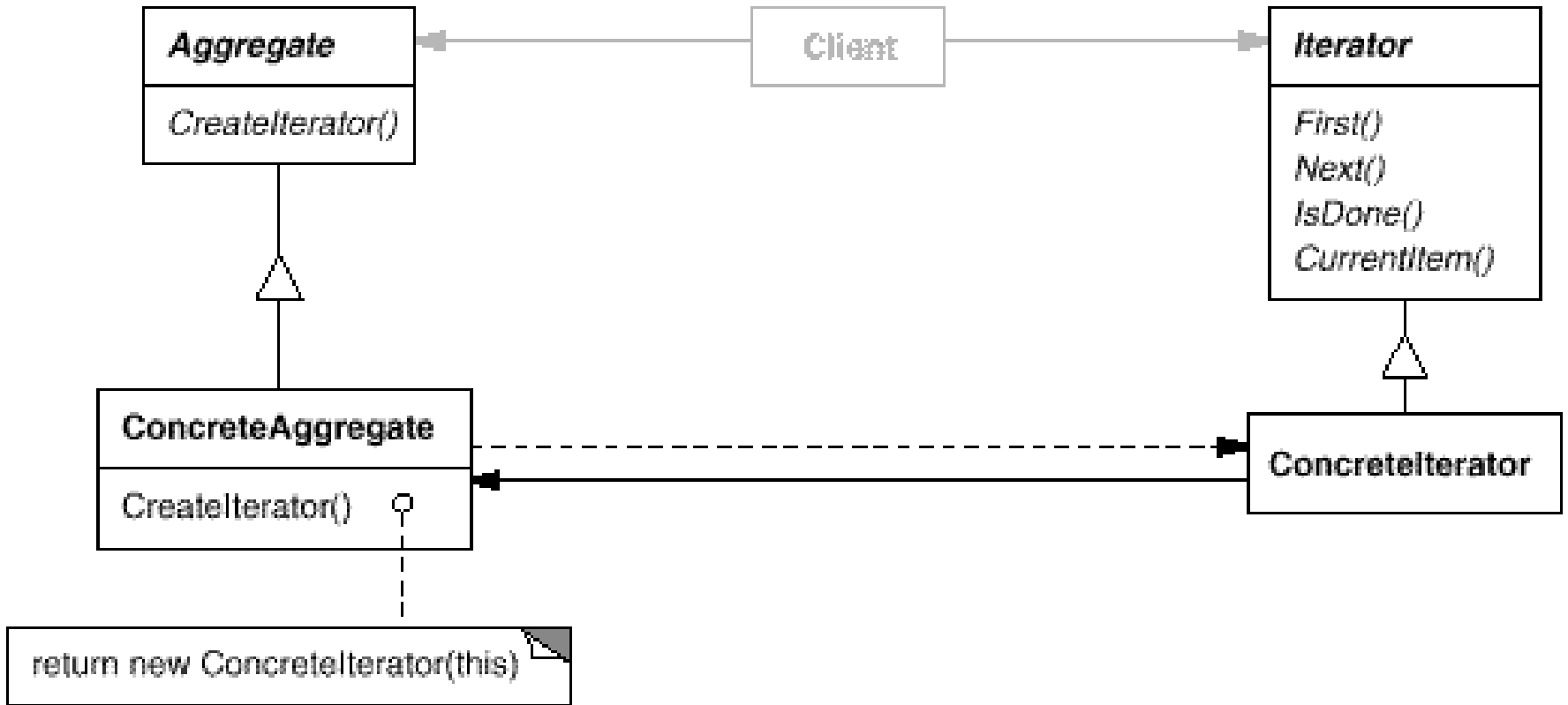


Funções do Iterador

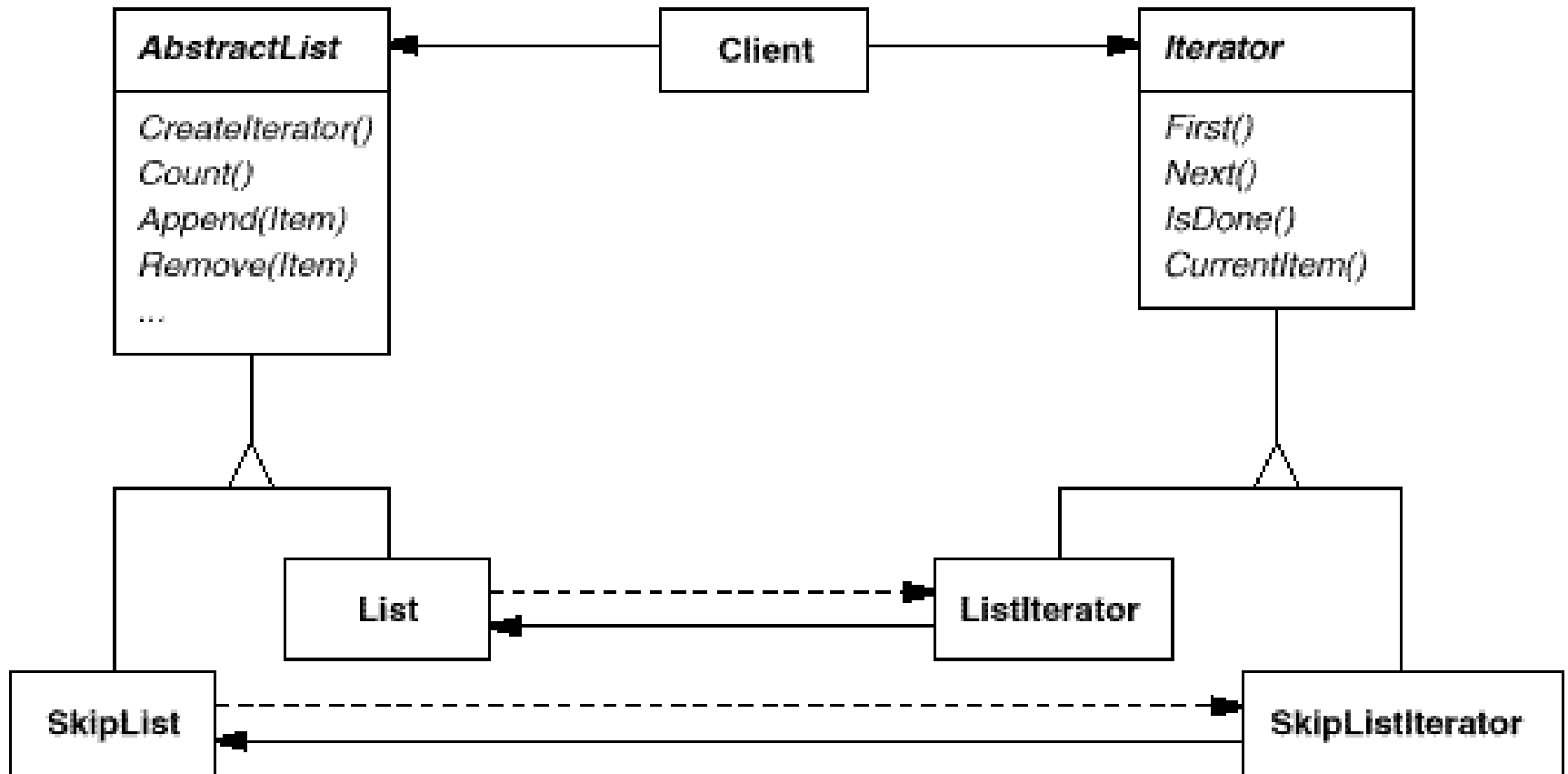
- Responsabilidades do Iterador
 - First: Para acessar o primeiro elemento do agregado
 - Next: Para ir para o próximo elemento do agregado
 - IsDone: booleano que representa o fim do agregado
 - CurrentItem: retorna o item do agregado referenciado no momento

Padrão de Projeto: **Iterator**

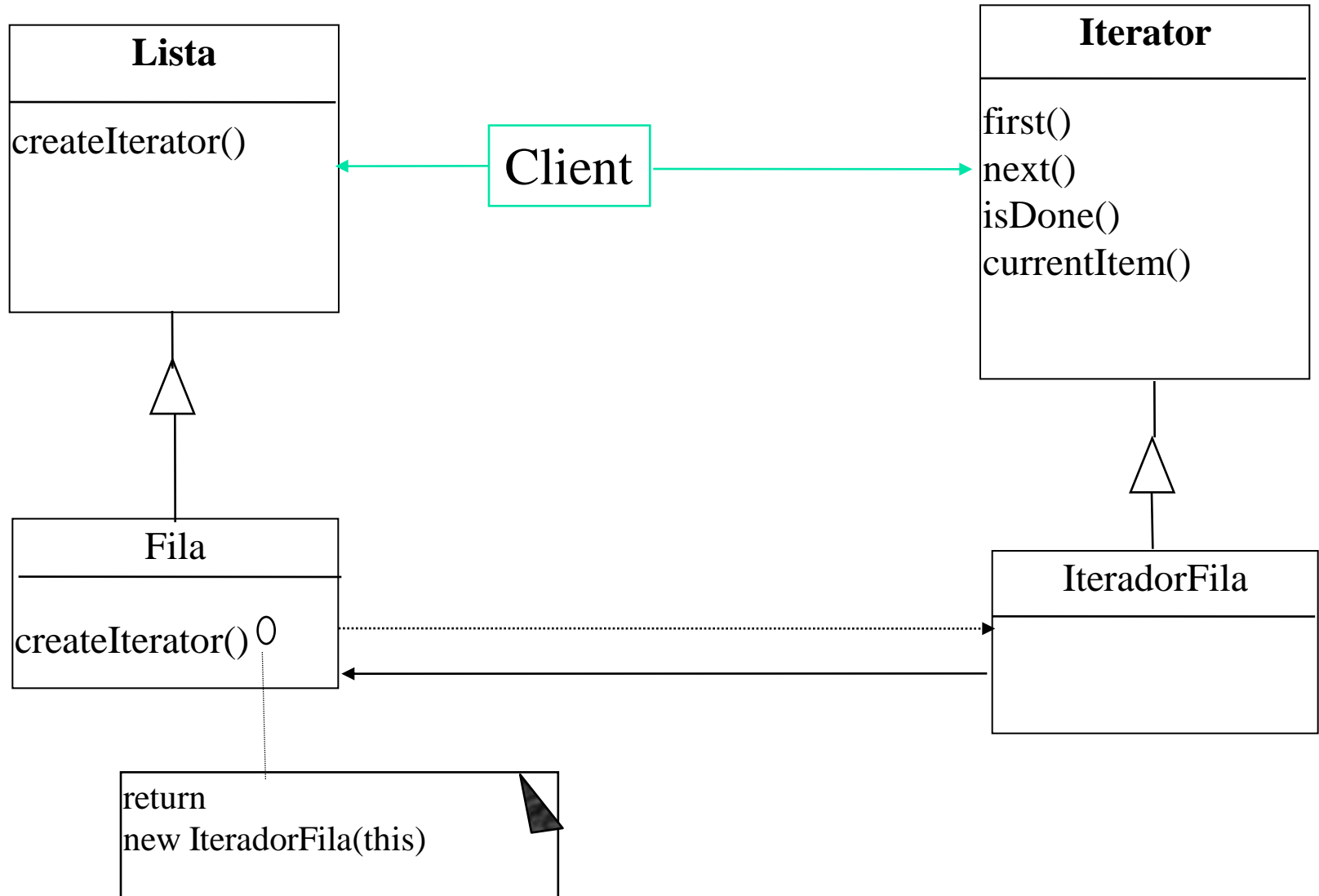
Iterator



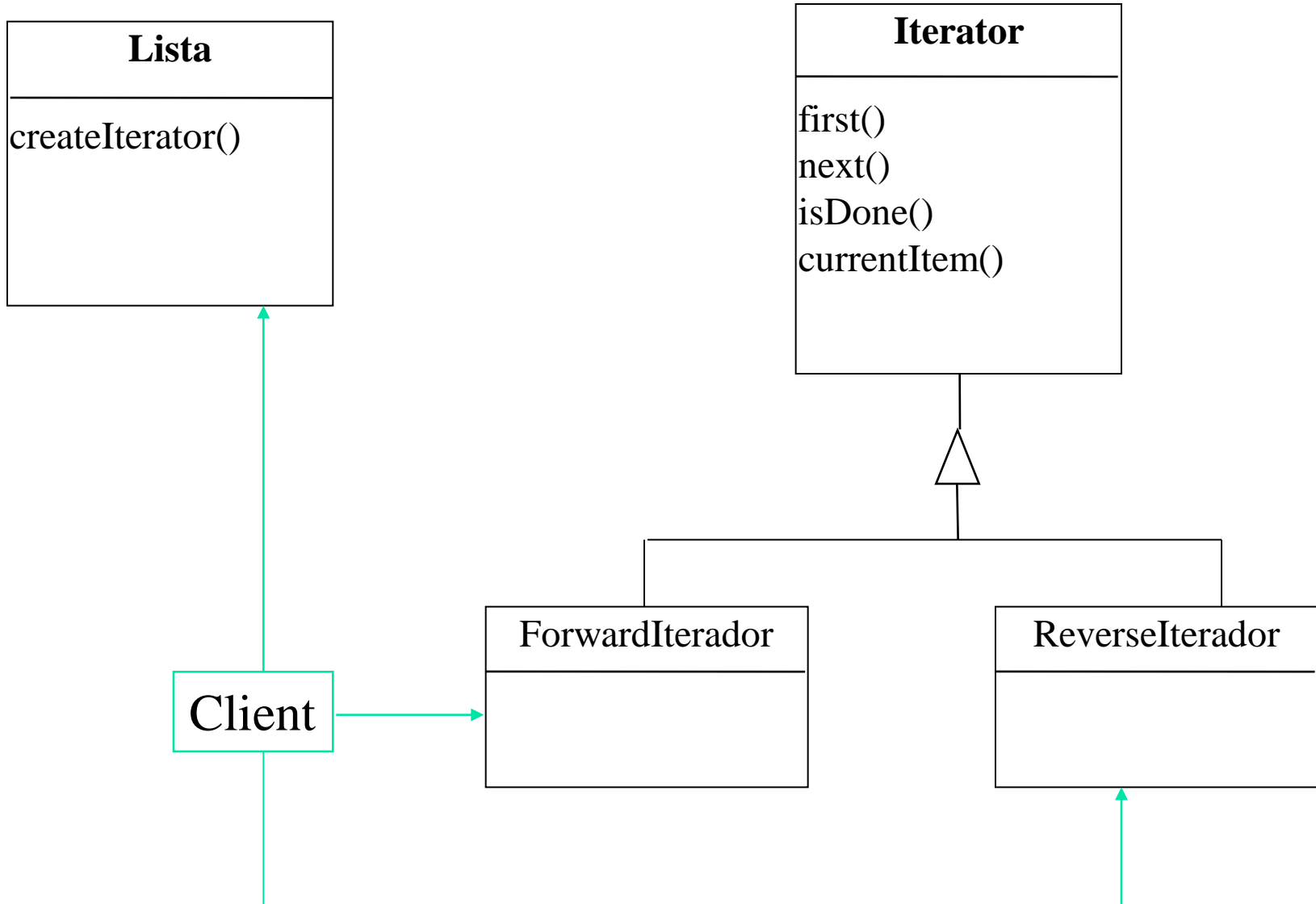
Exemplo 1



Exemplo 2



Exemplo 3

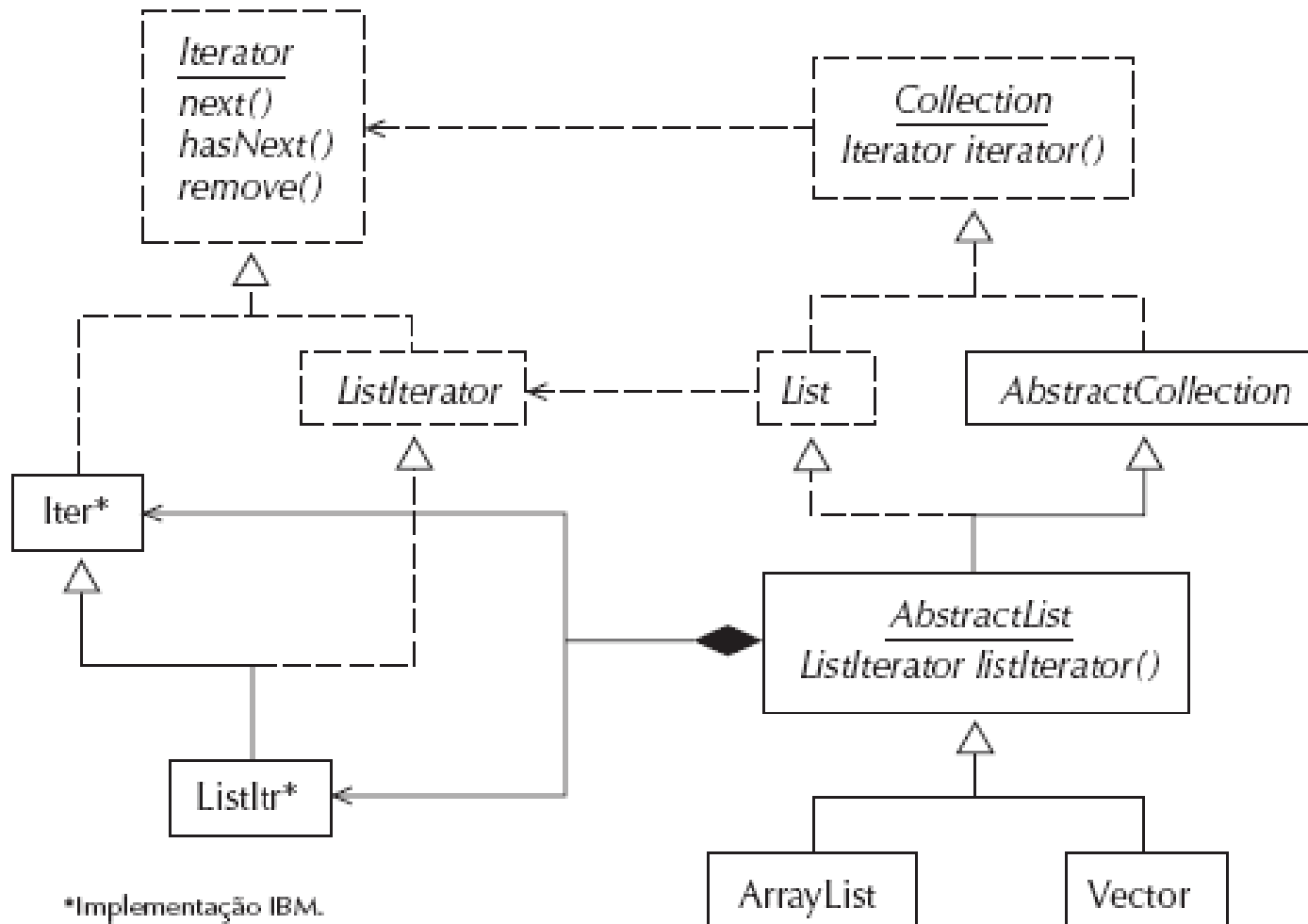




Iterador na API Java

- O pacote *java.util* contém uma interface *Iterator* com os métodos: *next()*, *hasNext()* e *remove()*.
- O método *next()* é uma combinação de *next()* e *currentItem()*
- 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



Uso do Iterator da API Java

```
public void devolverCopia(int codCopia)
{
    Iterator i = linhas.iterator();
    Date dataDeHoje = new Date();
    while (i.hasNext()) {
        LinhaDeEmprestimo linha = (LinhaDeEmprestimo) i.next();
        cc=linha.codigoCopia();
        if (cc==codCopia)
            linha.atualizaDataDev(dataDeHoje);
    }
}
```



Padrão State

- *Intenção*

- Permite que um objeto altere seu comportamento de acordo com mudança interna de estado.
- Dará a impressão que o objeto mudou de classe.

- Propósito:

- fazer com que um objeto comporte-se de uma forma determinada por seu estado.
- Agregar um objeto Estado e delegar comportamento a ele.



Padrão State

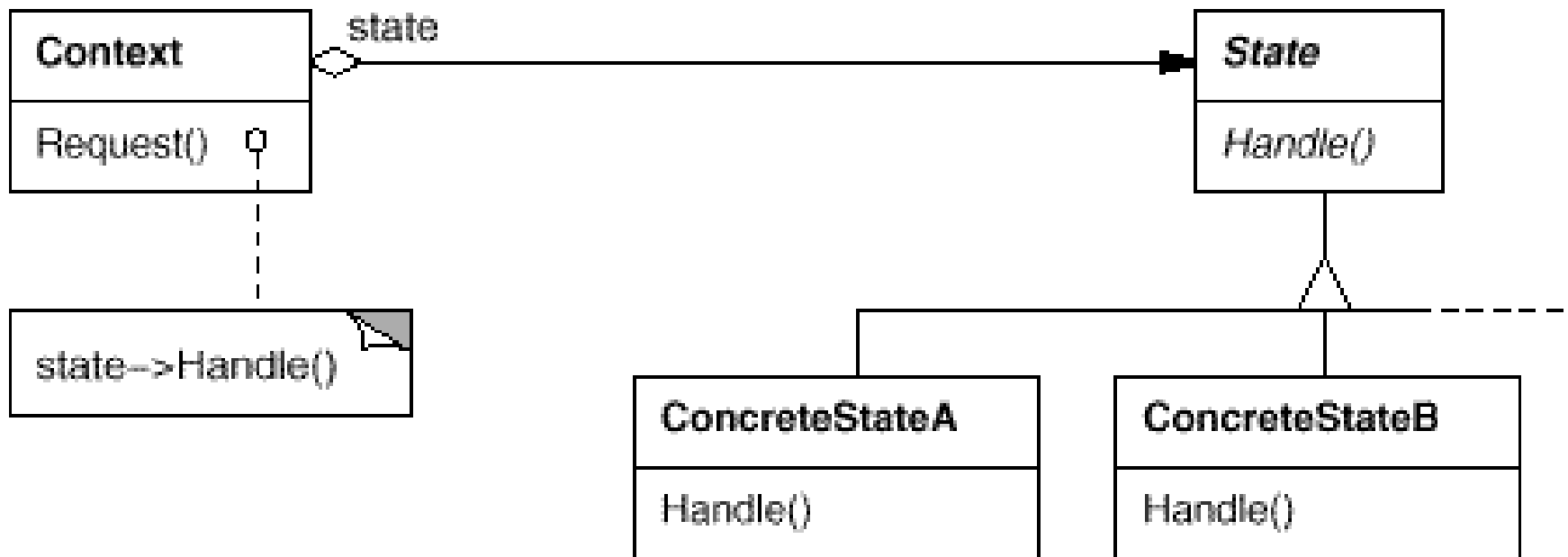
- Aplicabilidade: Use o padrão State em quaisquer das seguintes situações:
 - O comportamento de um objeto depende de seu estado e ele precisa mudar de comportamento em tempo de execução, dependendo de tal estado.
 - Operações possuem comandos grandes, de múltiplas partes condicionais, que dependem do estado do objeto. Em geral esse estado é representado por uma ou mais constantes enumeradas. Frequentemente será necessário repetir essa estrutura condicional em diversas operações.



Padrão State

- Aplicabilidade: Use o padrão State em quaisquer das seguintes situações:
 - O padrão State faz com que cada ramo da condicional fique em uma classe separada, permitindo que o estado do objeto seja tratado como um objeto em si e que esse estado possa variar independentemente de outros objetos.

Padrão State





Colaborações

- **O Contexto delega solicitações específicas de estado ao objeto ConcreteState.**
- **O Contexto pode passar a si próprio como argumento ao objeto State que está tratando a solicitação.**
- **O Contexto é a interface principal para os clientes, que podem configurá-lo com objetos State, de forma que não precisem lidar com os objetos State diretamente.**
- **Tanto as subclasses de Contexto quanto de ConcreteState podem decidir qual estado sucede outro e em que circunstâncias (transições).**



Questões de implementação

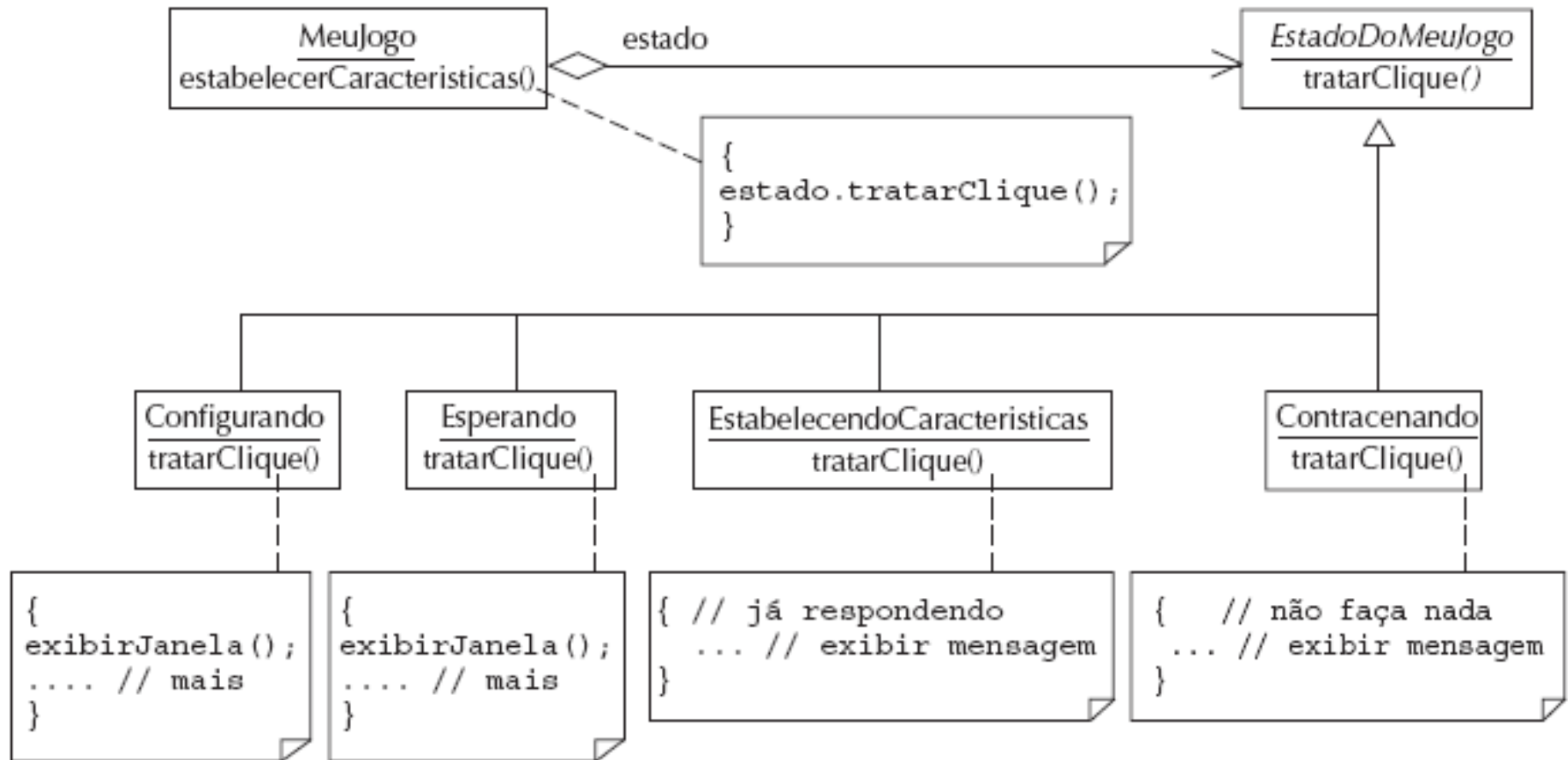
- Quem define as transições de estado: contexto ou estado?
 - Se há um critério fixo para mudança de estado: contexto
 - Mudanças constantes: estado
 - Necessária interface para que seja possível que objetos estados modifiquem explicitamente o estado atual do contexto
- Tabela para mapear transições
- Criar e destruir objetos Estado
 - Criar todos os objetos a priori ou criar/destruir sob demanda?

Exemplo: jogo de RPG



Cortesia de Tom VanCourt e Corel.

Exemplo





Padrões de Projeto

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



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.

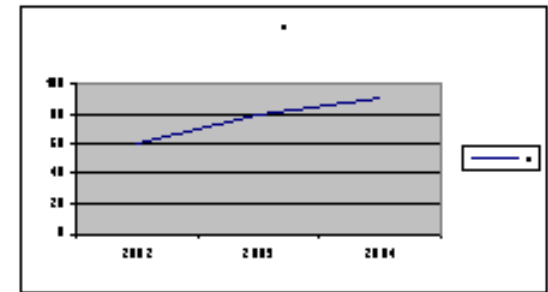
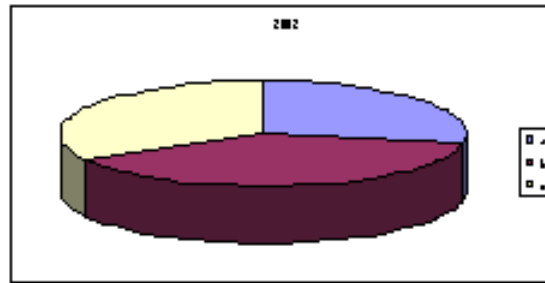
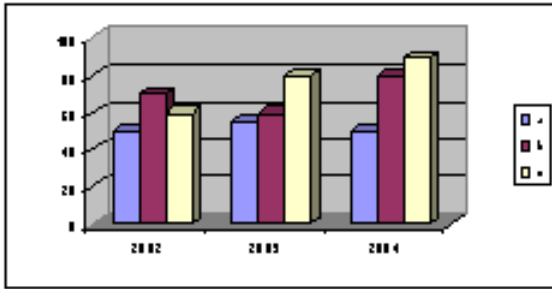


Padrão Observer

- 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

observadores



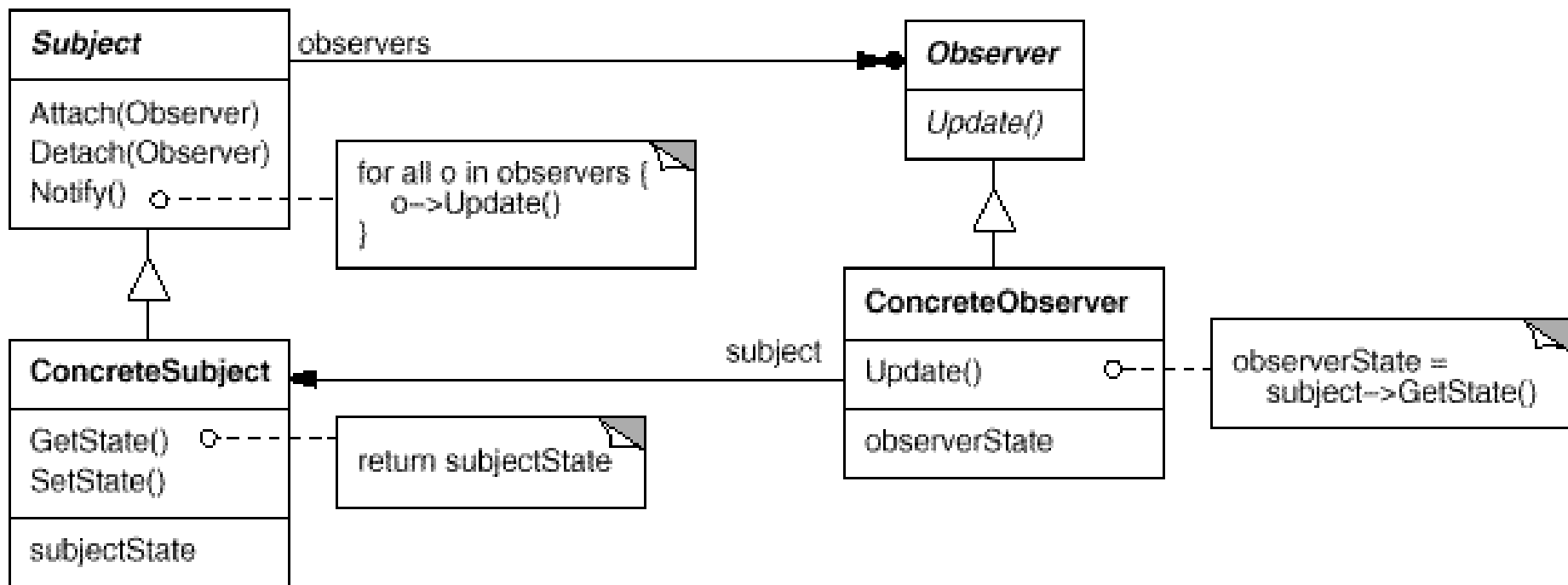
	2002	2003	2004
a	50	55	50
b	70	60	80
c	60	80	90

sujeito

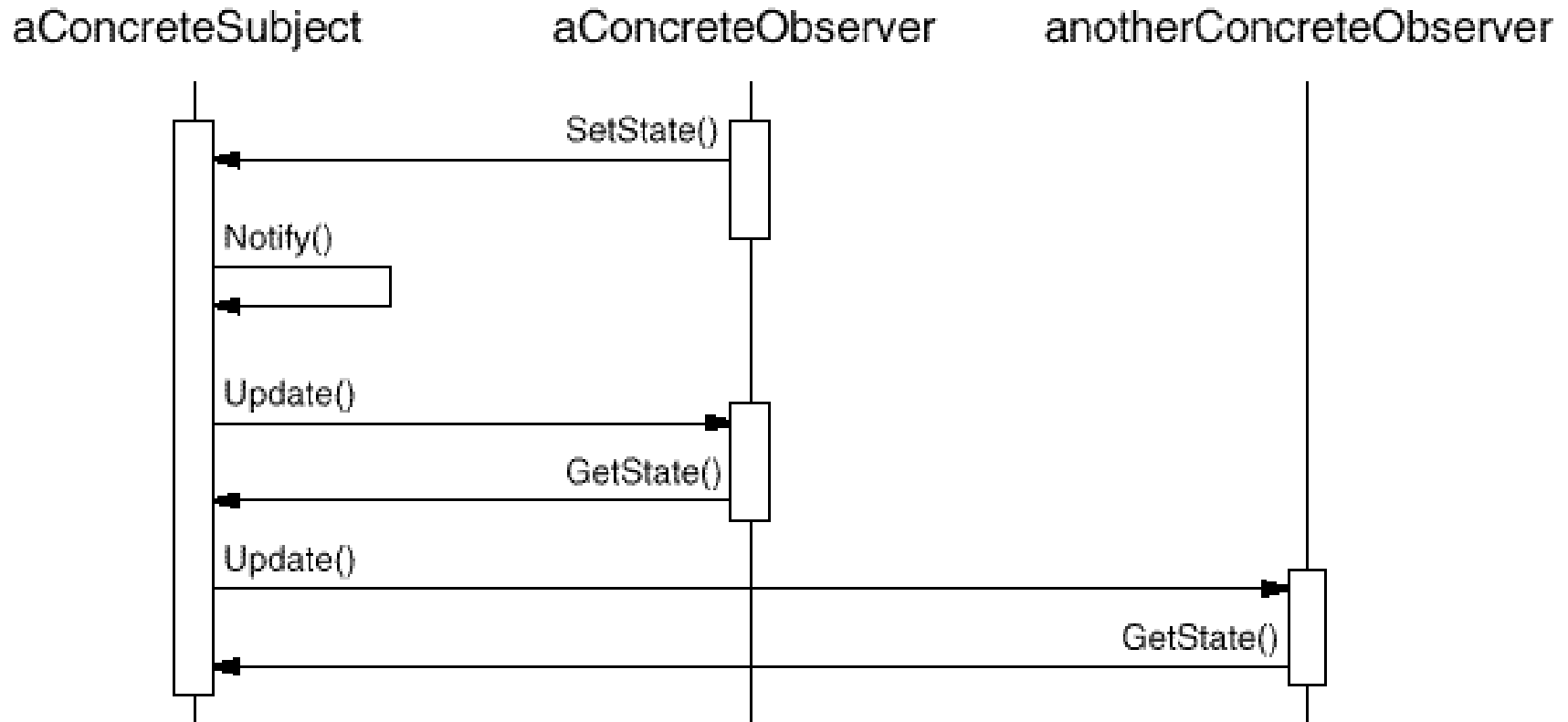
-----> alterações, requisições

-----> Notificação

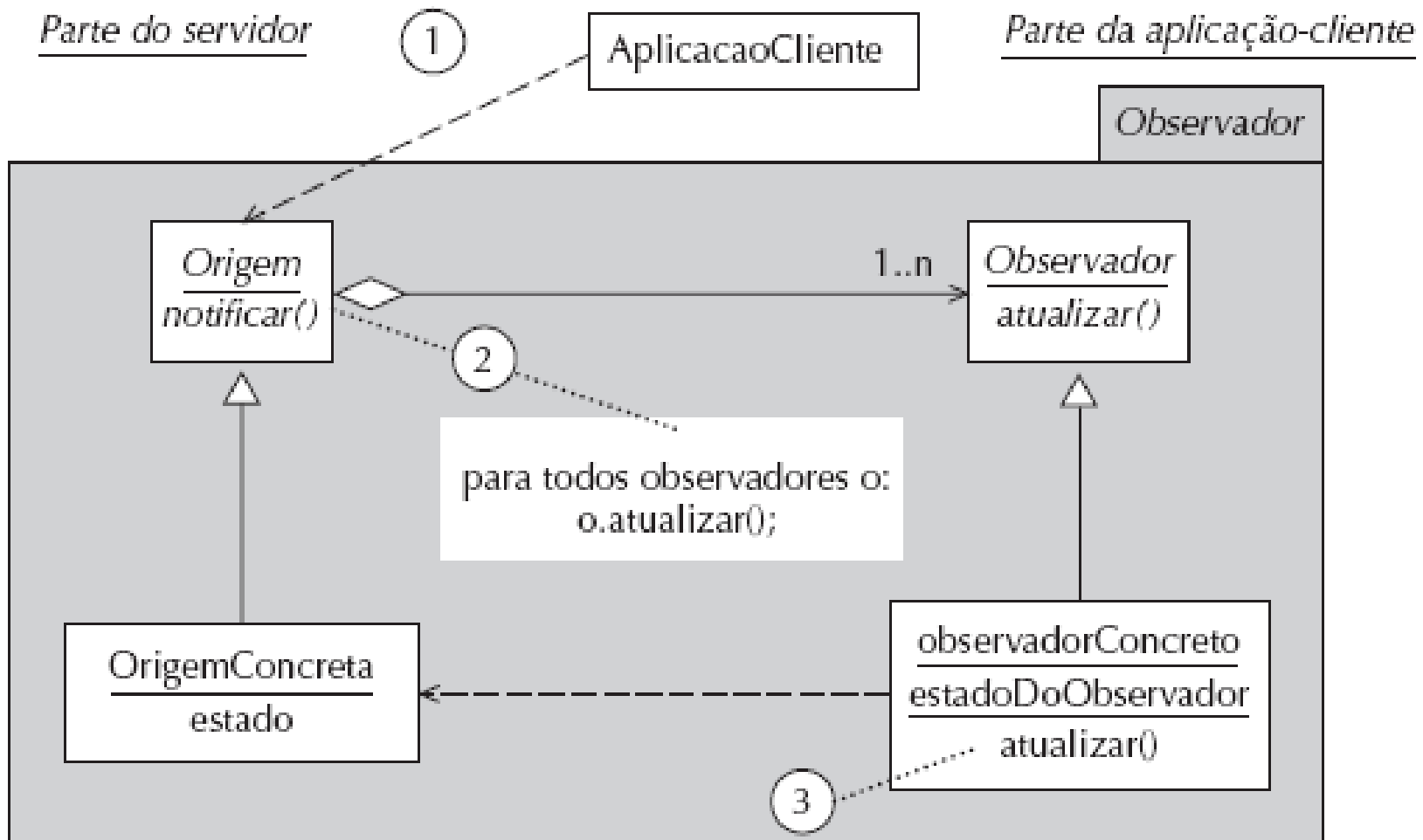
Padrão Observer



Padrão Observer



Como funciona o Observer?

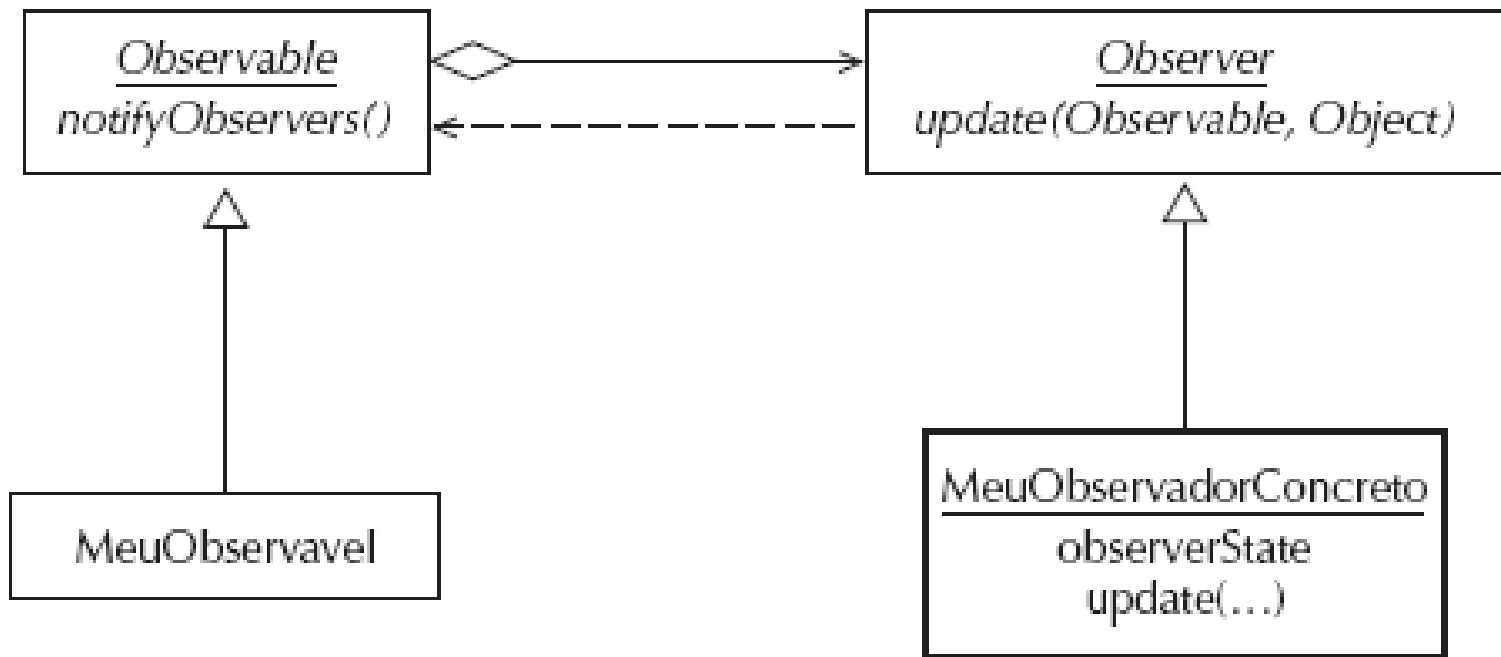




Como funciona o Observer?

- (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 Composite

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



Padrões de Projeto

- 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

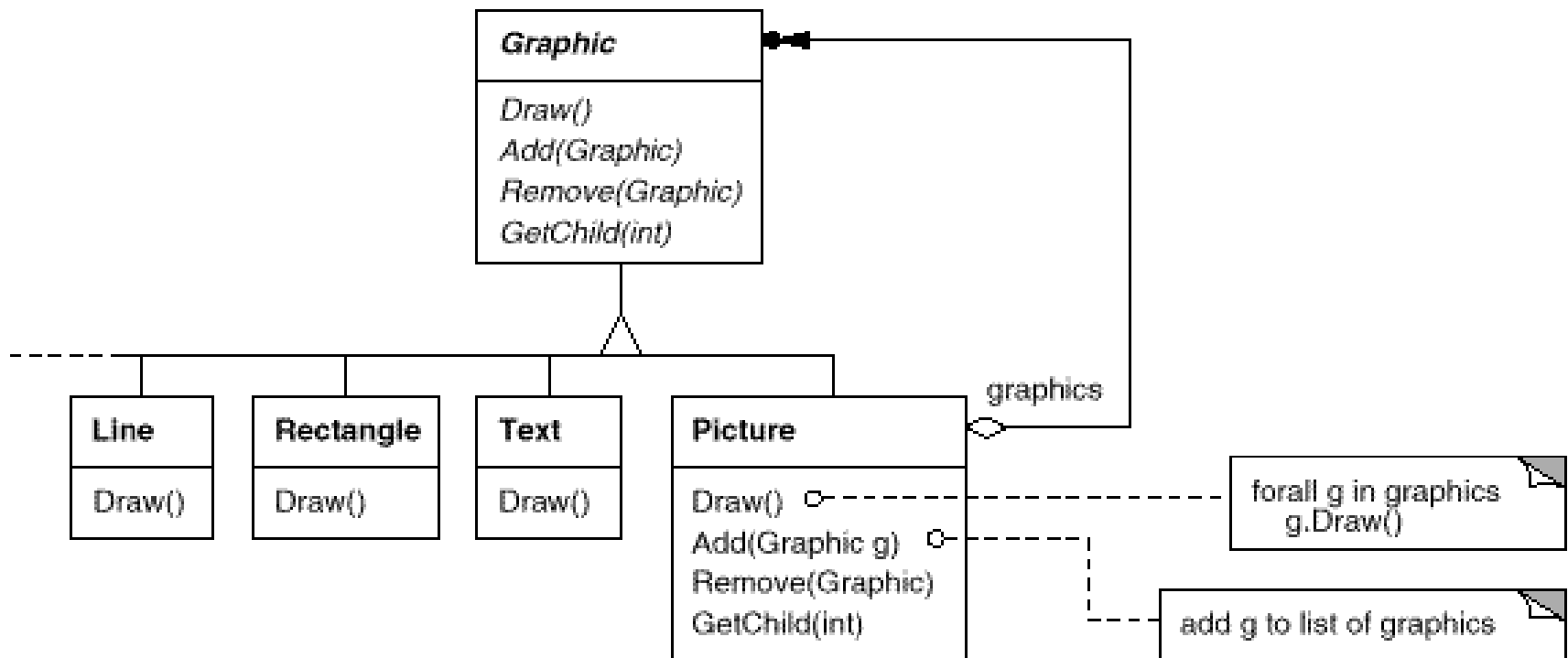


Padrões de Projeto

- 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

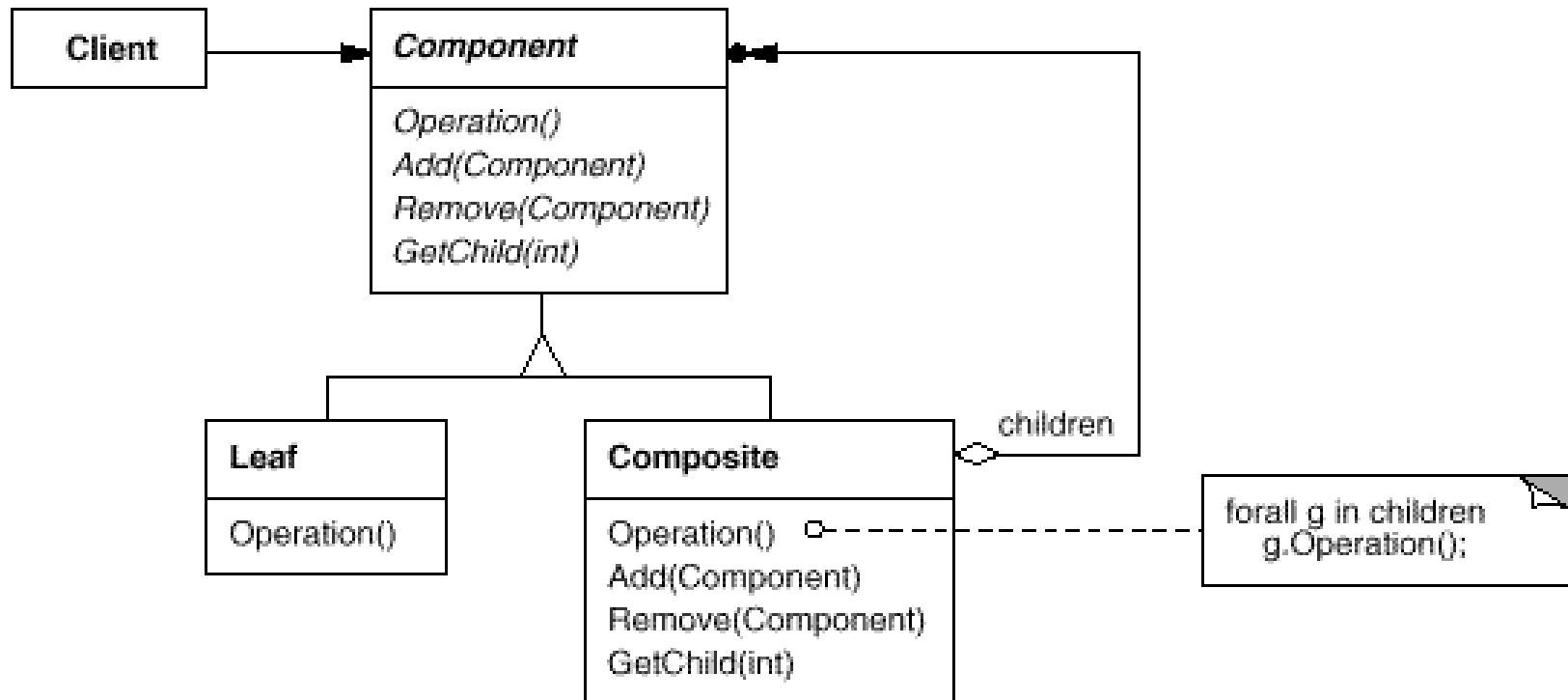
Padrões de Projeto

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



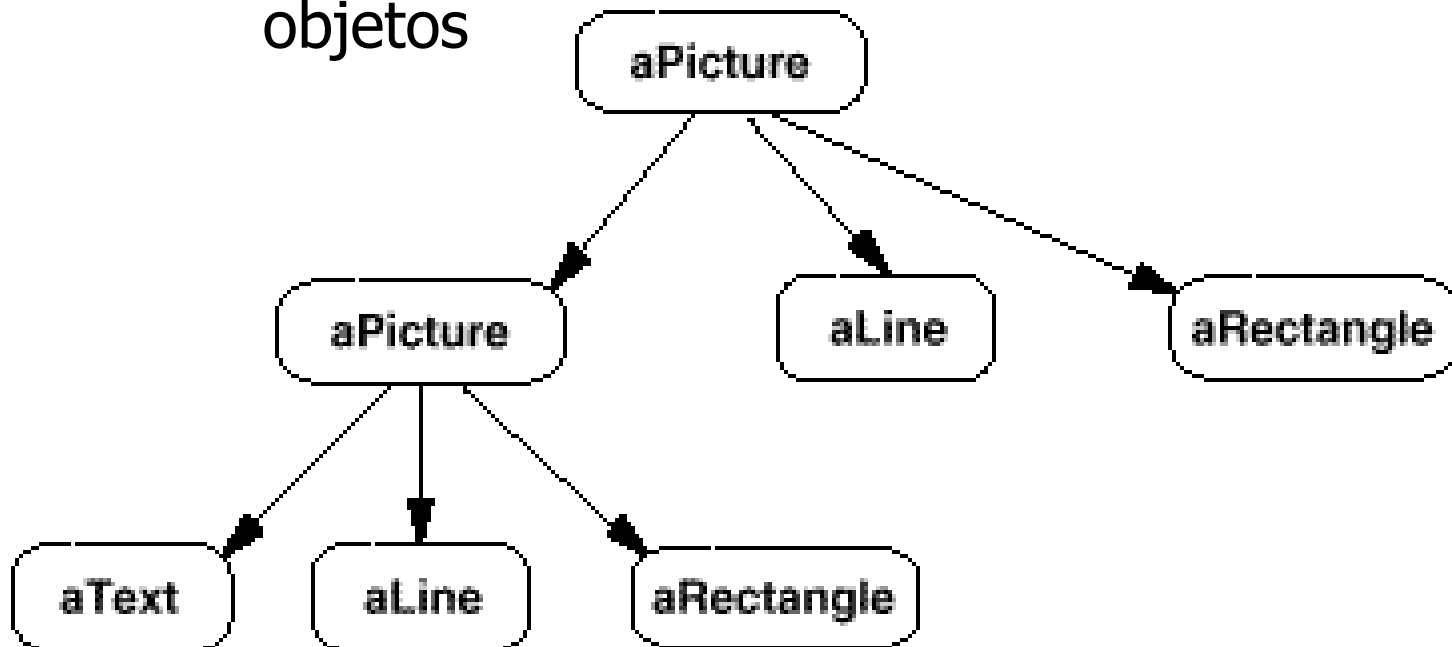
Padrões de Projeto

■ Estrutura (*Structure*)



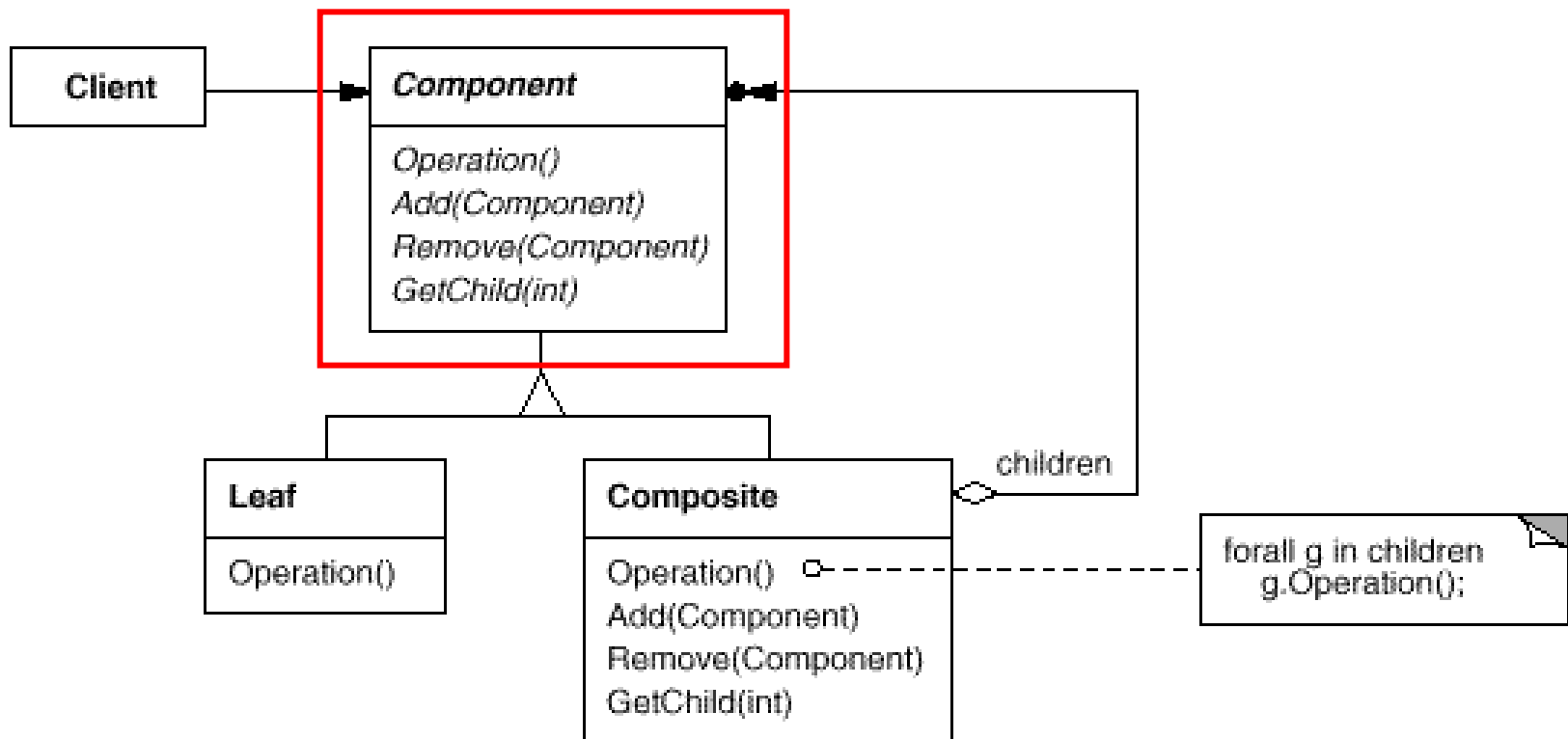
Padrões de Projeto

- Exemplo de composição recursiva de objetos



Padrões de Projeto

■ Participantes (*Participants*)



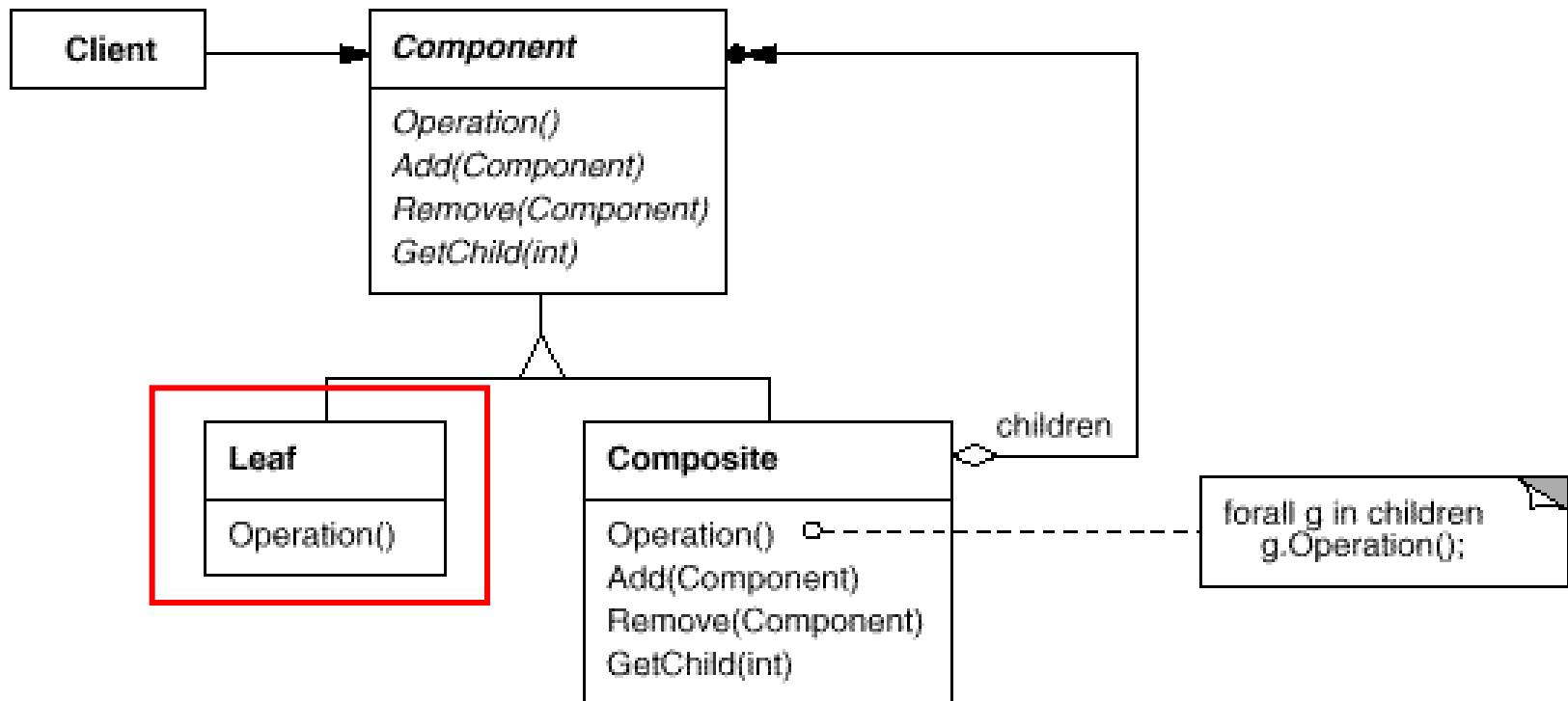


Padrões de Projeto

- Participantes (*Participants*)
 - Component (Gráfico)
 - 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

■ Participantes (*Participants*)



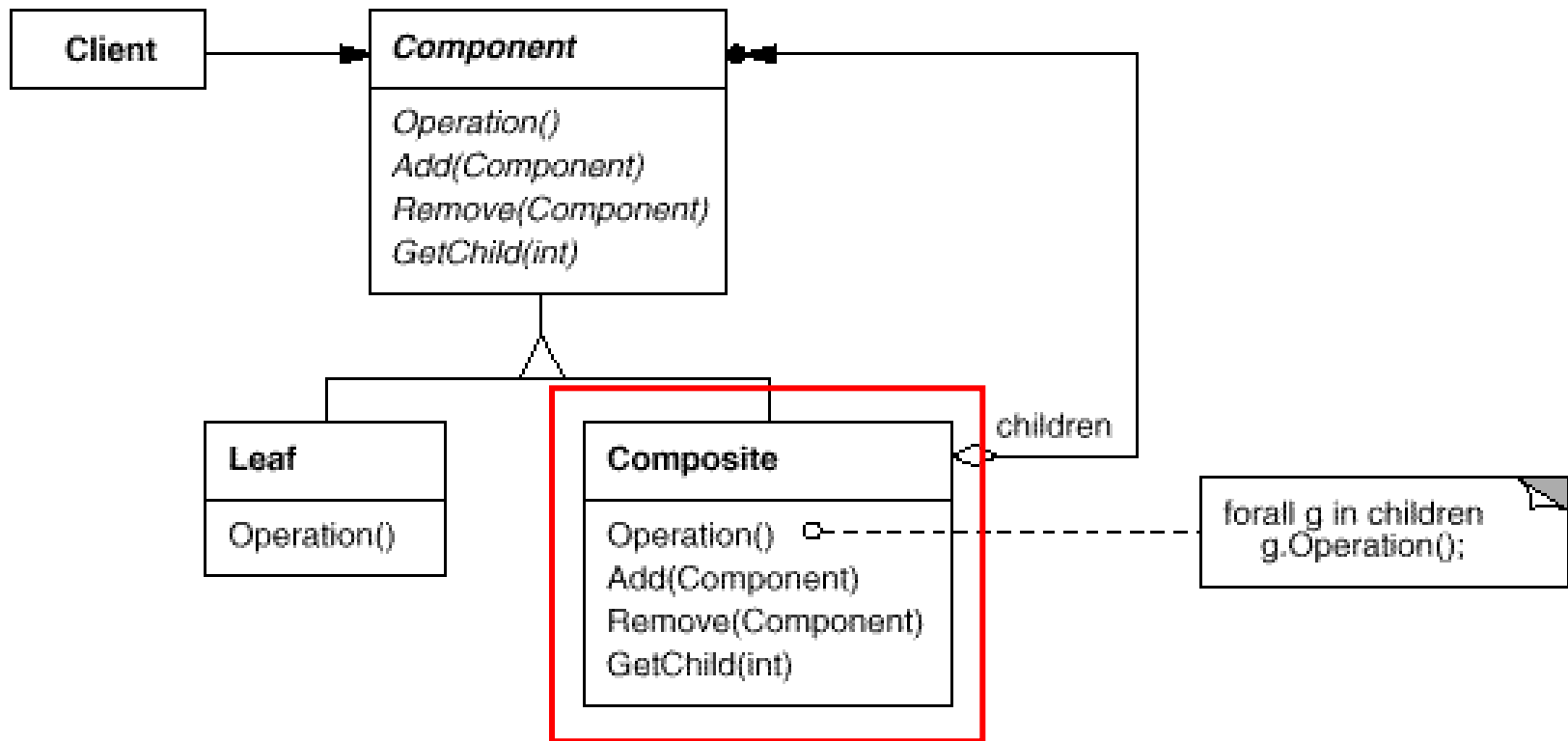


Padrões de Projeto

- Participantes (*Participants*)
 - Leaf (Rectangle, Line, Text, etc.)
 - representa objetos “folha” na composição. Uma folha não tem filhos
 - define o comportamento para objetos primitivos na composição

Padrões de Projeto

■ Participantes (*Participants*)



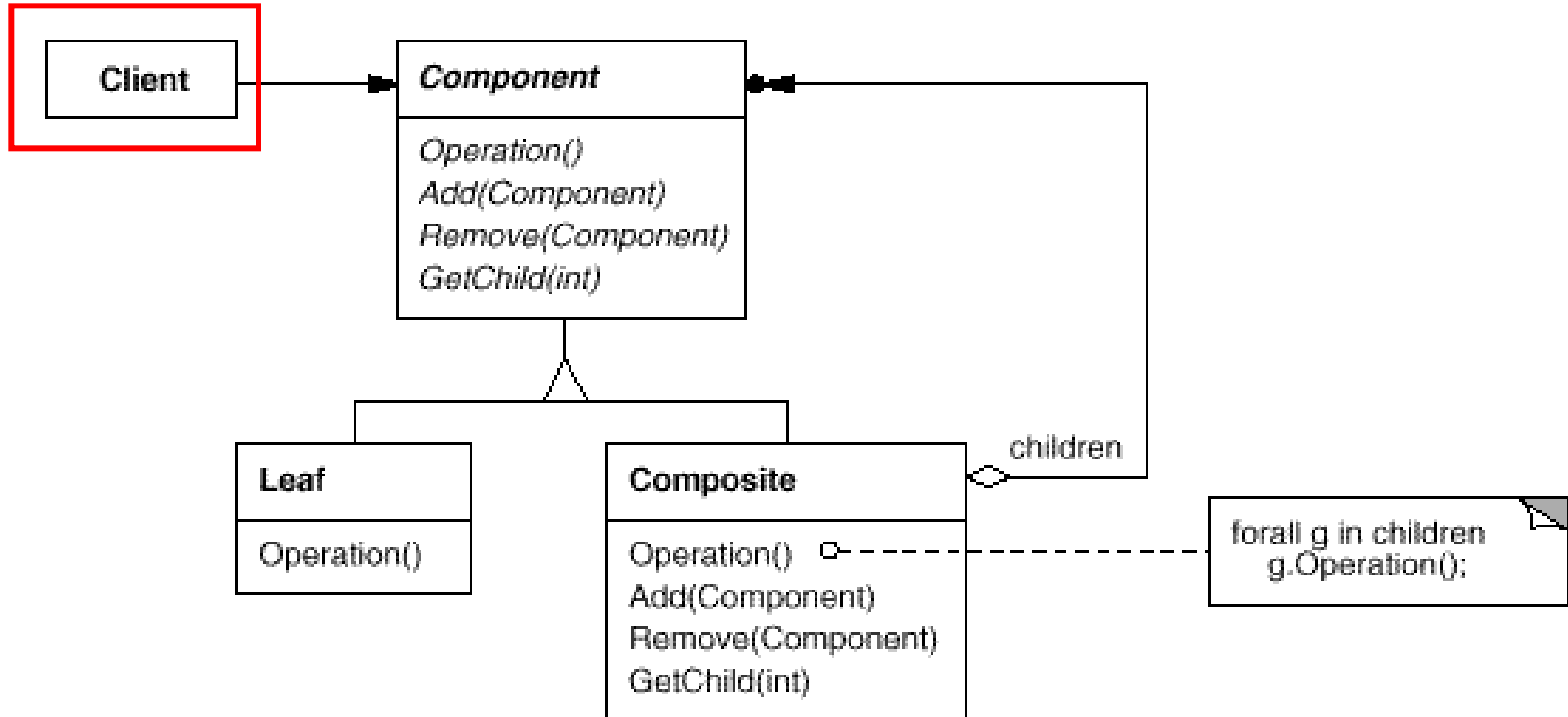


Padrões de Projeto

- Participantes (*Participants*)
 - Composite (Picture)
 - define o comportamento para componentes que têm filhos
 - armazena componentes filho
 - implementa operações relacionadas aos filhos na interface Component

Padrões de Projeto

■ Participantes (*Participants*)





Padrões de Projeto

- Participantes (*Participants*)
 - Client
 - manipula objetos na composição pelo através da interface Component



Padrões de Projeto

- 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

- 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

■ 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

```
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();  
};
```

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

Nós folha

Método do composto

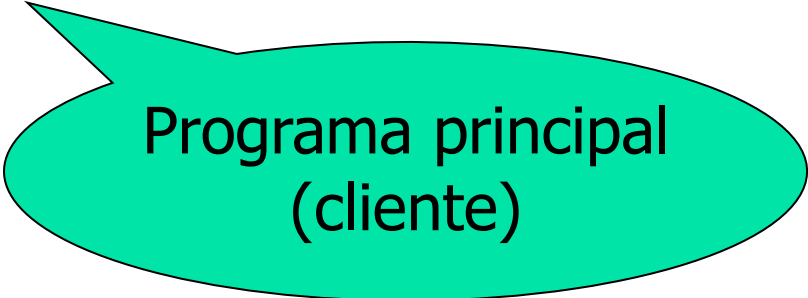


Padrões de Projeto

```
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

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