Design Patterns

Histórico e Fundamentos:

- *Padrões do GoF*
- *Criacionais*
- *Estruturais*
- *Comportamentais*

Fernando Anselmo
fernando.anselmo@x25.com.br

25 anos na área de Desenvolvimento e Coordenação
13 Livros e diversos artigos publicados
Coordenador do DFJUG
Cargo Atual: Coordenador Técnico da X25 Informática

Site: http://fernandoans.site50.net

Principais Benefícios

- Natural
- Confiável
- Reutilizável
- Manutenível
- Extensível
- Oportuna

1ª – Pensar na POO como linguagem

2ª – Medo da reutilização

3ª – Pensar como solução “sucesso”

4ª – Programação Egoísta

POO x Técnicas Tradicionais

- Modular
- Métodos
- Atributos
- Mensagens
- Classes
- Herança
- Controle do Usuário

- Seqüêncial
- Procedimentos e Funções
- Dados
- Chamadas
- Tipos de Dados
- Cópia de arquivos
- Linear

- **Abstração**

Isolar os aspectos que sejam importantes para algum propósito e suprimir os que não forem.

- **Encapsulamento**

Definição da OO que não é preciso conhecer o todo para saber o funcionamento da classe.

- **Herança**

Uma classe pode gerar novas classes que sejam suas cópias perfeitas e a partir destas e possível readaptá-las ao meio em que vivem.

- **Por segurança o acesso aos atributos e objetos são definidos por "private":**

```
private double salario;
```

- **Para acessá-los define-se métodos modificadores**

- **Método "set" - Entrada**

```
public void setSalario(double val) { salario = val;}
```

- **Método "get" - Saída**

```
public double getSalario() { return salario; }
```

Modificadores Importantes

- **final – classes, métodos e variáveis, não poderão mais serem modificados;**

- **abstract – classes e métodos, não são instanciados, serão utilizados por herança, declarado sem corpo;**

- **static – métodos e variáveis, não ocorre a subscrição dos métodos (utilizada em Serializable);**

- **Constructor**

É um método público, sem retorno e com o mesmo nome da classe, utilizado no momento da instância da classe por um objeto:

```
class Teste { public Teste() { } }
```

- **Utilização por herança**

Usa-se a palavra chave "super" para se referir a utilização de um "constructor pai"

```
class A { public A(int a) { } }
class B extends A { public B(int b) { super(b); } }
```

- # Métodos Overload

São métodos com nomes iguais mas que recebem parâmetros diferentes, ocorrendo dentro da mesma classe:

```
public int calcCPF(long num) { }
public boolean calcCPF(long num, byte num) { }
```

- # Métodos Override

Ocorrem por herança, para modificar ou complementar determinado método, neste caso todo o cabeçalho deve ser idêntico:

```
class A { public void show() { System.out.println("A"); } }
class B extends A { public void show() { super.show(); } }
```

- Projetar software OO é difícil, e projetar software OO reusável é mais difícil ainda.

- É preciso encontrar objetos pertinentes, fatorá-los em classes na granularidade certa, definir interfaces de classes e hierarquias de herança, e estabelecer relacionamentos chave entre elas.

- Soluções individuais aparecem. Desenvolvedor Heróico é boa solução?
- Boas soluções aumentam a produtividade, qualidade e a uniformidade. Como aproveitá-las?
- Como documentar as soluções de modo a reutilizá-las sempre que preciso?

Como resolver estes problemas?

- Capturam soluções desenvolvidas e evoluídas com o passar do tempo.
- Boas soluções que já funcionaram devem ser reutilizadas (reuso de experiências anteriores).
- Cada padrão descreve um problema que ocorre freqüentemente em seu ambiente, e então descreve a solução para aquele problema,de um modo tal que pode-se usar essa solução.

Idéia Inicial

- Chistopher Alexander do livro: The Timeless Way of Building.

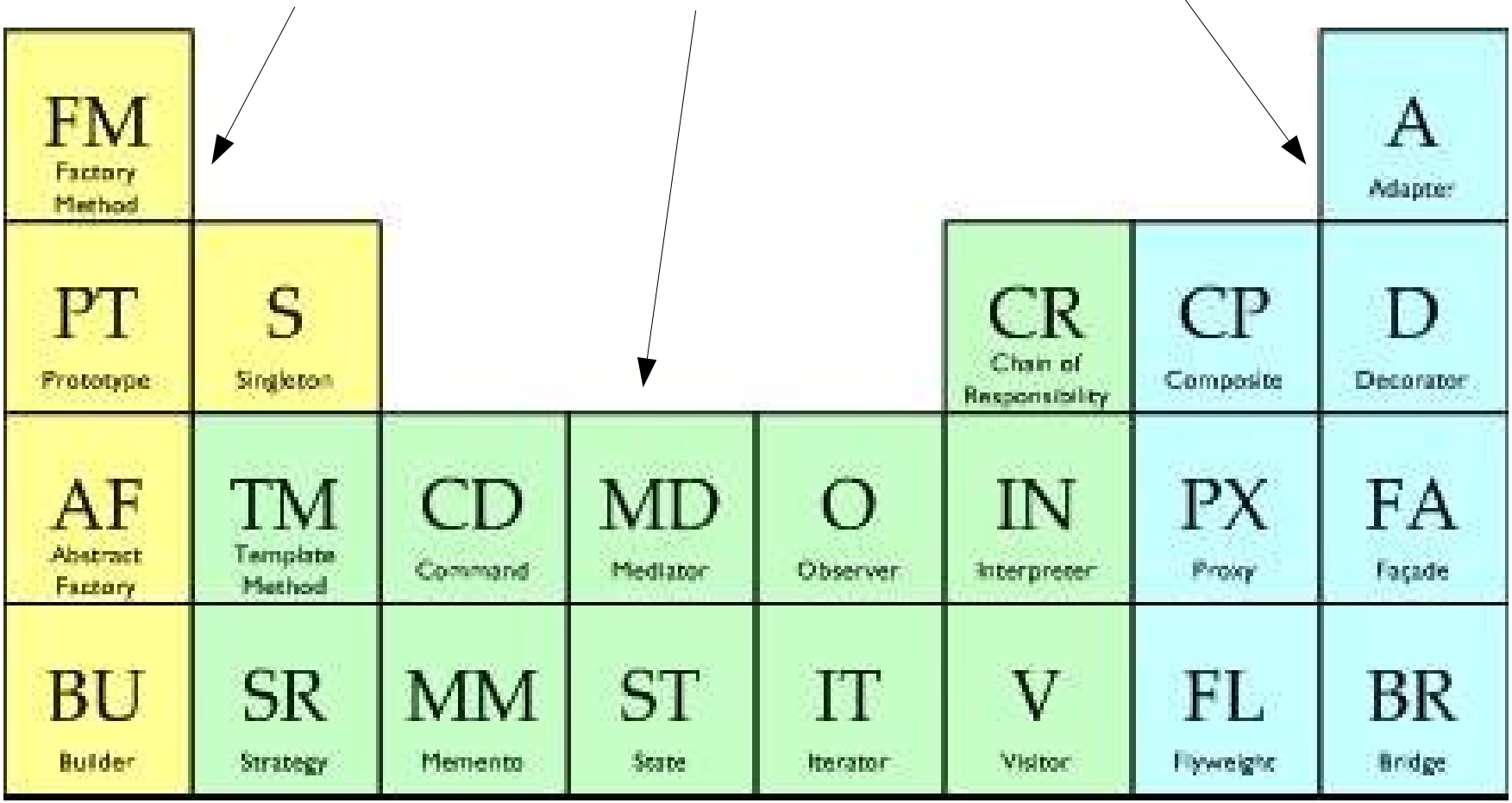
- “Se padrões são úteis, conjuntos de padrões podem ser usados juntos na construção de arquiteturas completas, são ainda mais úteis” (Brugali, 2000)

- 1994 – Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides - GoF

O que é um Design Pattern

- Soluções exaustivamente refinadas: Resultados de um longo processo de projeto, re-projeto, teste e reflexão sobre o que torna um sistema mais flexível, re-usável, modulado e compreensível.
- Soluções compartilhadas: Construídas em grupo utilizando um vocabulário comum.
- Soluções sucintas e de fácil aplicação: Capturam, de forma sucinta e facilmente aplicável, soluções do projeto que foram desenvolvidas e evoluíram com o passar do tempo.

- **Criacionais, Comportamentais e Estruturais**



Criacionais

- Todos os patterns criacionais tratam da melhor maneira como instanciar objetos

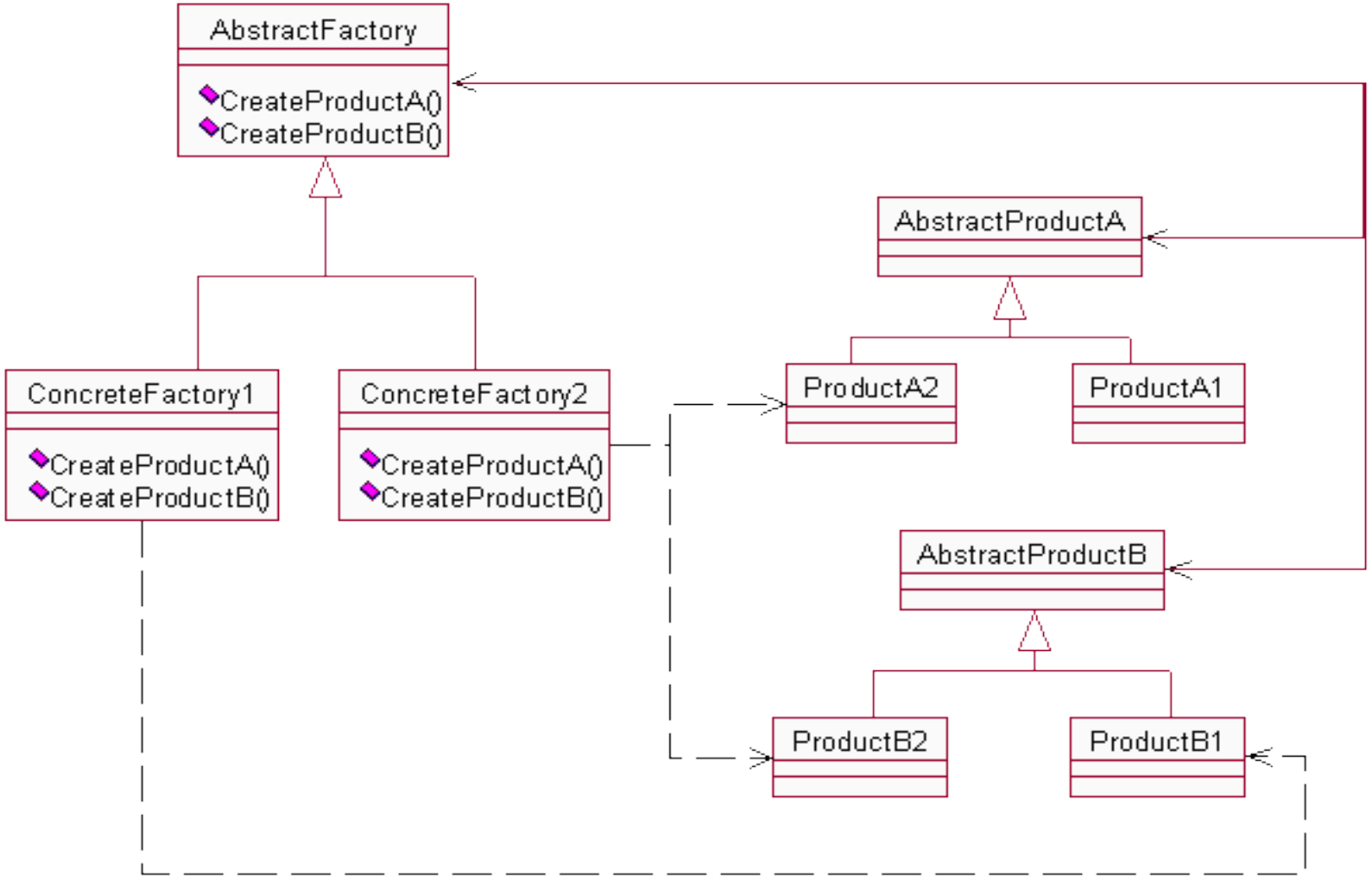
- É importante porque o programa não depende de como os objetos são criados ou organizados

- Em muitos casos a natureza exata de um objeto que é criado pode variar de acordo com as necessidades do programa e abstrair o processo de criação em uma "Classe Criadora" especial pode tornar o programa mais flexível e geral.

- **Provê uma interface para criação de famílias de objetos relacionados ou dependentes sem especificar suas classes concretas;**

- **É usado quando:**
  - Um sistema deve ser independente de como os seus elementos são criados, compostos e representados;
  - Um sistema deve ser configurado para trabalhar com uma única família dentre múltiplas famílias de produtos;
  - Uma família de produtos relacionados é projetada para ser usada em conjunto e há necessidade de reforçar essa restrição;
  - Se quer criar uma biblioteca de classes de produtos, revelando apenas suas interfaces e não suas implementações.

Pattern Abstract Factory - Estrutura



- **Conseqüências:**
  - Isola as classes concretas;
  - Facilita a troca de famílias de produtos;
  - Facilita o suporte a novos tipos de produtos.
- **Patterns Relacionados:**
  - As classes Abstract Factory normalmente são implementadas com Factory Methods, podendo também ser implementadas com Prototype;
  - Uma classe do tipo Concrete Factory freqüentemente é um Singleton.

Pattern Abstract Factory - Problema

- Planeje um modelo de plano de jardins. Podem ser anuais, vegetais ou perenes. Porém, não importa que tipo de jardim, sempre temos os seguintes tipos de plantas:

1. Quais são boas para as bordas?
2. Quais são boas para o centro?
3. Quais são boas nas áreas sombreadas?

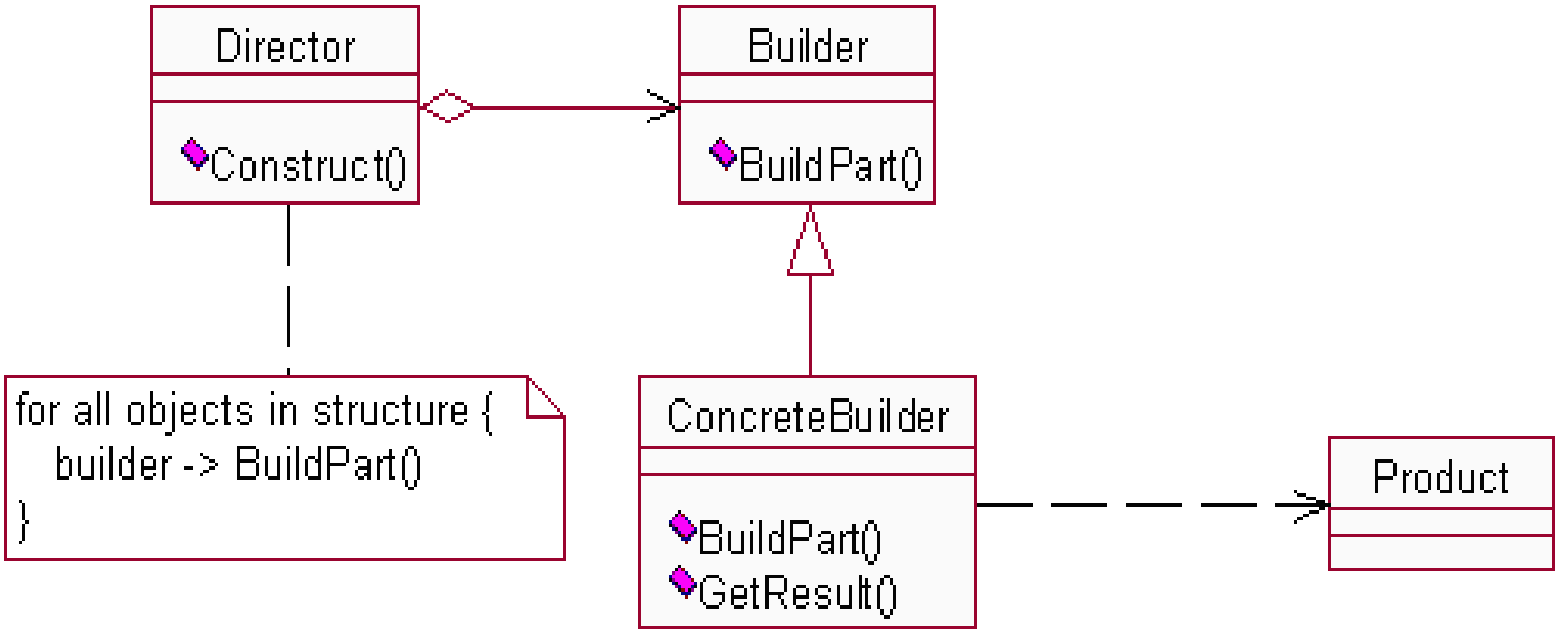
Resumidamente:

	Sombra	Centro	Borda
Anual	Bromélias	Jatobá	Bamboo
Perene	Roseira	Ipê	Pinheiro
Vegetal	Brócolis	Milho	Ervilhas

- Separa construção de um objeto complexo de sua representação, de modo que o mesmo processo de construção possa criar diferentes representações;

- É usado quando:
  - O algoritmo para a criação de um objeto complexo deve ser independente das partes que o compõem e de como estas são conectadas entre si;
  - O processo de construção deve permitir a criação de diferentes representações do objeto construído.

Pattern Builder - Estrutura



- **Conseqüencias:**
  - Possibilita variar a representação interna de um produto;
  - Isola o código de construção e representação aumentando a modularidade;
  - Possibilita um grande controle sobre o processo de construção.

- **Patterns Relacionados:**
  - O Abstract Factory é similar ao Builder pelo fato de, com ele, também se poder construir objetos complexos. A Principal diferença é que o Builder objetiva a construção de objetos complexos passo a passo. A ênfase do Abstract Factory é em famílias de objetos produto. O Builder retorna o produto como um passo final, mas no que diz respeito ao Abstract Factory, o produto final é retornado imediatamente.

Pattern Builder - Problema

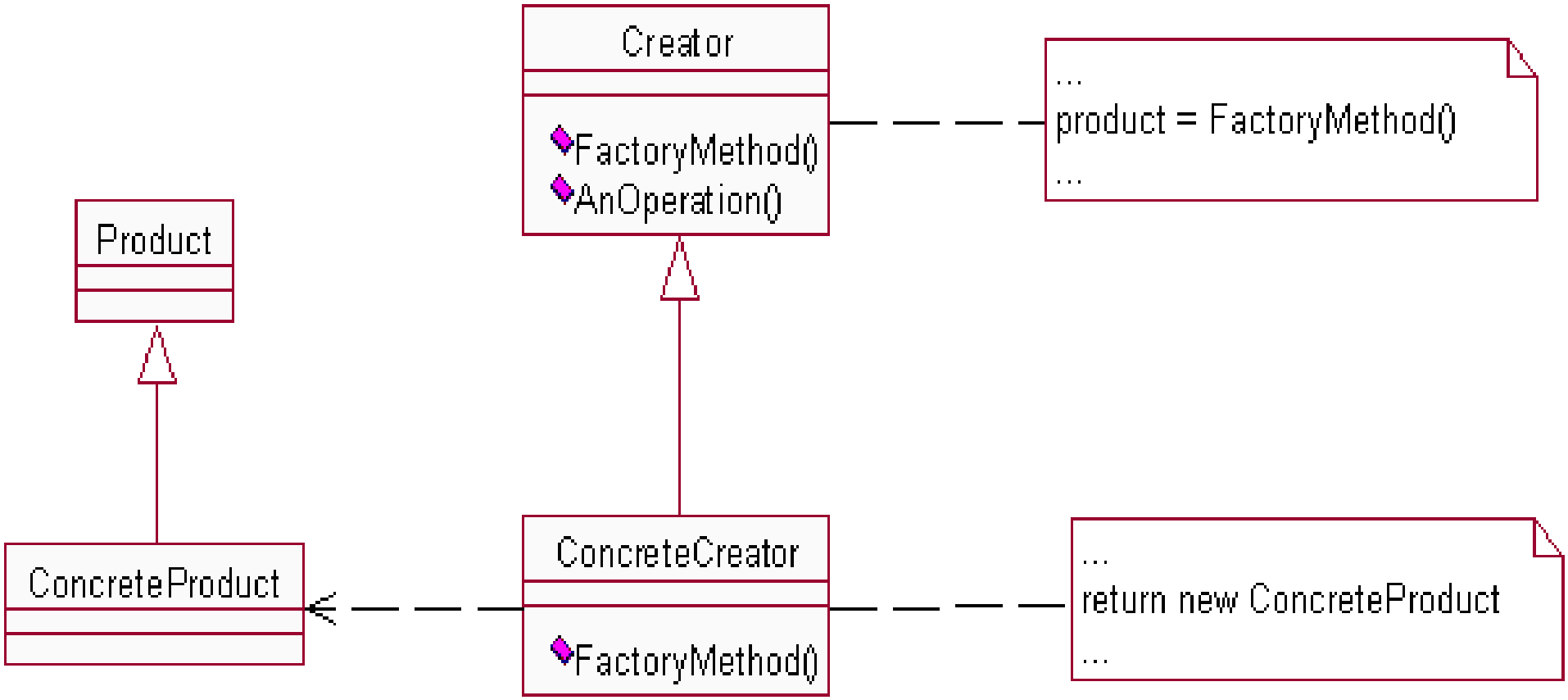
- Um **Cliente** pode necessitar aleatoriamente de construir uma casa ou um prédio o que pode ser realizado em várias etapas. O **Cliente** repassa estas informações para seu **Diretor** que deve ordenar a construção desta utilizando uma interface que permita o acréscimo de novas construções e de novas etapas.

- Define uma interface para a criação de um objeto, mas deixa as sub-classes definirem qual classe deve ser instanciada

- Permite a uma classe delegar a instanciação às sub-classes.

- **É usado quando:**
  - Uma classe não pode antecipar a classe de objeto que deve ser criada;
  - Uma classe quer que suas sub-classes especifiquem os objetos que ela cria;
  - Classes delegam responsabilidades para uma dentre várias sub-classes auxiliares, e se deseja localizar o conhecimento de qual sub-classe auxiliar implementa a delegação.

Pattern Factory Method – Estrutura



- **Conseqüências:**
  - Provê ganchos para subclasses;
  - Conecta hierarquia de classes paralelas quando há delegação.

- **Patterns Relacionados:**
  - Abstract Factory é frequentemente implementado com Factory Methods;
  - Factory Methods são usualmente chamados de dentro de Template Methods.

Problema

- Como acessar quaisquer objetos das classes descritas, sem que o cliente saiba qual objeto concreto estará acessando?

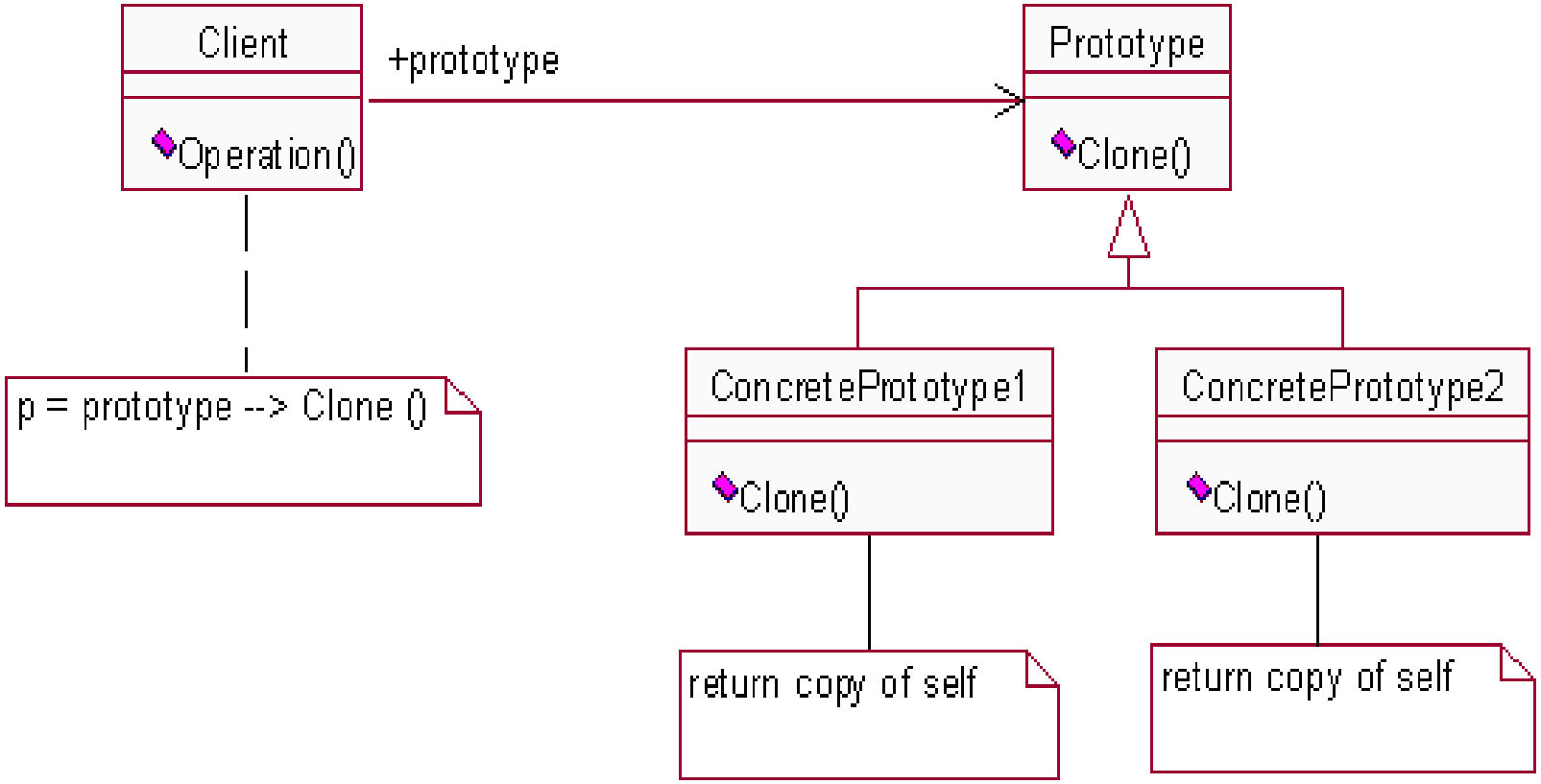
```
public interface Shape {
    public void draw();
}
public class Retangulo implements Shape {
    public void draw() { System.out.println("Retângulo"); }
}
public class Circulo implements Shape {
    public void draw() {  System.out.println("Círculo"); }
}
public class Cliente {
    public static void main(String [] args) {
        Shape obj = new QualquerObjeto();
        obj.draw();
    }
}
```

- Especifica os tipos de objetos a criar usando uma instância protótica, e cria os novos objetos através da cópia deste protótipo.

- **É usado quando:**
  - O sistema deve ser independente de como os seus produtos são criados, compostos e representados;
  - As classes a instanciar são especificadas em tempo de execução;
  - Queremos evitar a construção de uma hierarquia de classes de fabricação paralela a uma hierarquia de classes de produtos por elas fabricados;

- É usado quando:
  - Instâncias de uma classe podem ter uma de algumas poucas combinações de estados diferentes. Pode ser mais conveniente instalar um número correspondente de protótipos e cloná-los, ao invés de instanciar as classes no estado apropriado manualmente, toda vez que for necessário.

Pattern Prototype - Estrutura



- **Conseqüências:**
  - Adicionar e remover produtos em tempo de execução;
  - Especificar novos objetos pela variação de valores;
  - Especificar novos objetos pela variação de estruturas: utilizar objetos compostos como protótipos;
  - Redução do número de subclasses (hierarquia);
  - Configurar uma aplicação com classes dinamicamente.

- **Patterns Relacionados:**
  - Abstract Factory" e "Prototype" são patterns que podem vir a competir em algumas situações, porém também podem ser usados juntos. Um "Abstract Factory" poderia armazenar um conjunto de "Prototypes" que seriam clonados para retornar os objetos produto;
  - Projetos que fazem grande uso dos patterns "Composite" e "Decorator" freqüentemente também podem ser beneficiados com o uso do "Prototype".

Problema

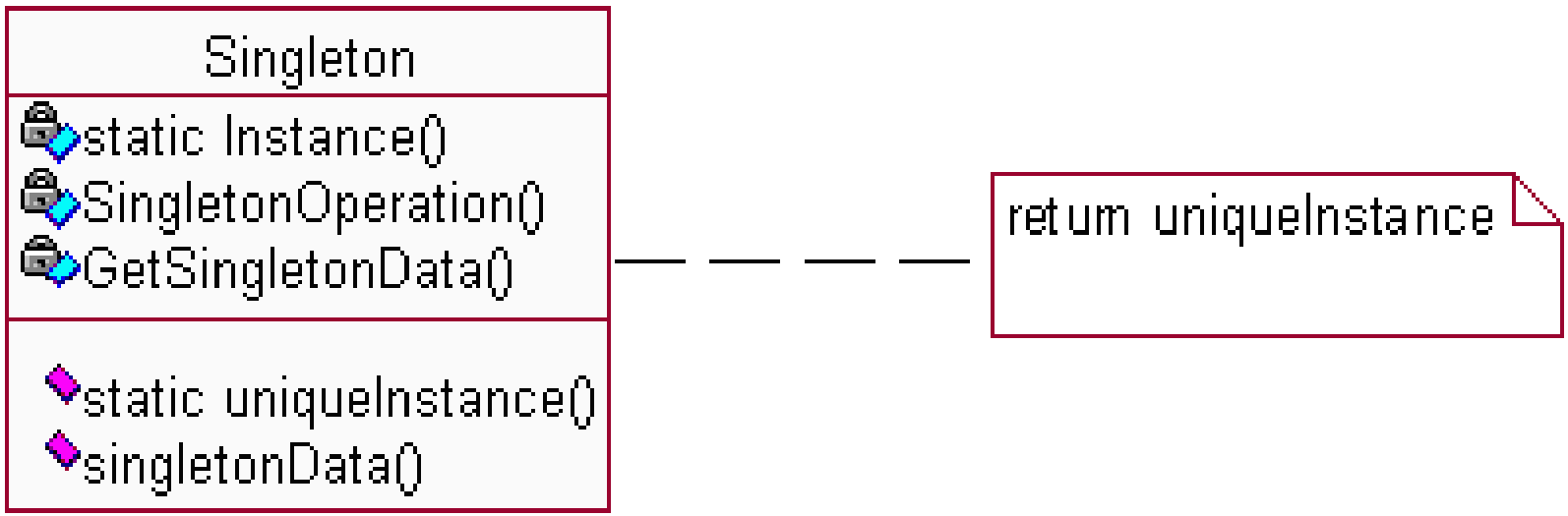
- Promova uma maneira de poder gerar novos objetos da classe Circulo sendo estes cópias perfeitas do estado do original:

```
class Ponto {
    public final int x, y;
    public Ponto(int x, int y) { this.x = x; this.y = y; }
    public String toString() { return " " + x + y; }}
class Circulo {
    private Ponto origem; private double raio;
    public Circulo(int x, int y, double raio) {
        origem = new Ponto(x, y); this.raio = raio;
    }
    public void show() {
      System.out.println(origem + " raio = " + raio); }}
public class Cliente {
    public static void main(String [] args) {
        Circulo c = new Circulo(4, 5, 6); c.show();
        Circulo copia = c;
        System.out.println((c == copia)?"Iguais":"Difer");
}}
```

- Garantir que uma classe tem apenas uma instância, e prover um ponto de acesso global a ela;

- **É usado quando:**
  - Deve haver exatamente uma única instância de uma classe, e ela deve estar disponível a todos os clientes de um ponto de acesso bem definido;
  - Desejar que a única instância possa ser estendida por herança, e os clientes serem capazes de utilizar essa instância estendida sem terem de modificar o seu código.

Pattern Singleton - Estrutura



- **Conseqüências:**
  - Acesso controlado à única instância;
  - Espaço de nomes reduzido;
  - Permite refinamento de operações e representação via especialização;
  - Permite um número variável de instâncias;
  - Maior flexibilidade do que em operações de classes.

- **Patterns Relacionados:**
  - Muito patterns, como o Abstract Factory, o Builder e o Prototype podem ser implementados usando o pattern Singleton.

Problema

- **Escreva uma classe que permita resolver as seguintes questões:**

- Controlar (contar) o número de instâncias da classe?
- Armazenar a(s) instância(s)?
- Controlar ou impedir a construção normal? Se for possível usar new e um construtor para criar o objeto, há como limitar instâncias?
- Definir o acesso à um número limitado de instâncias (no caso, uma apenas)?
- Garantir que o sistema continuará funcionando se a classe participar de uma hierarquia de classes?

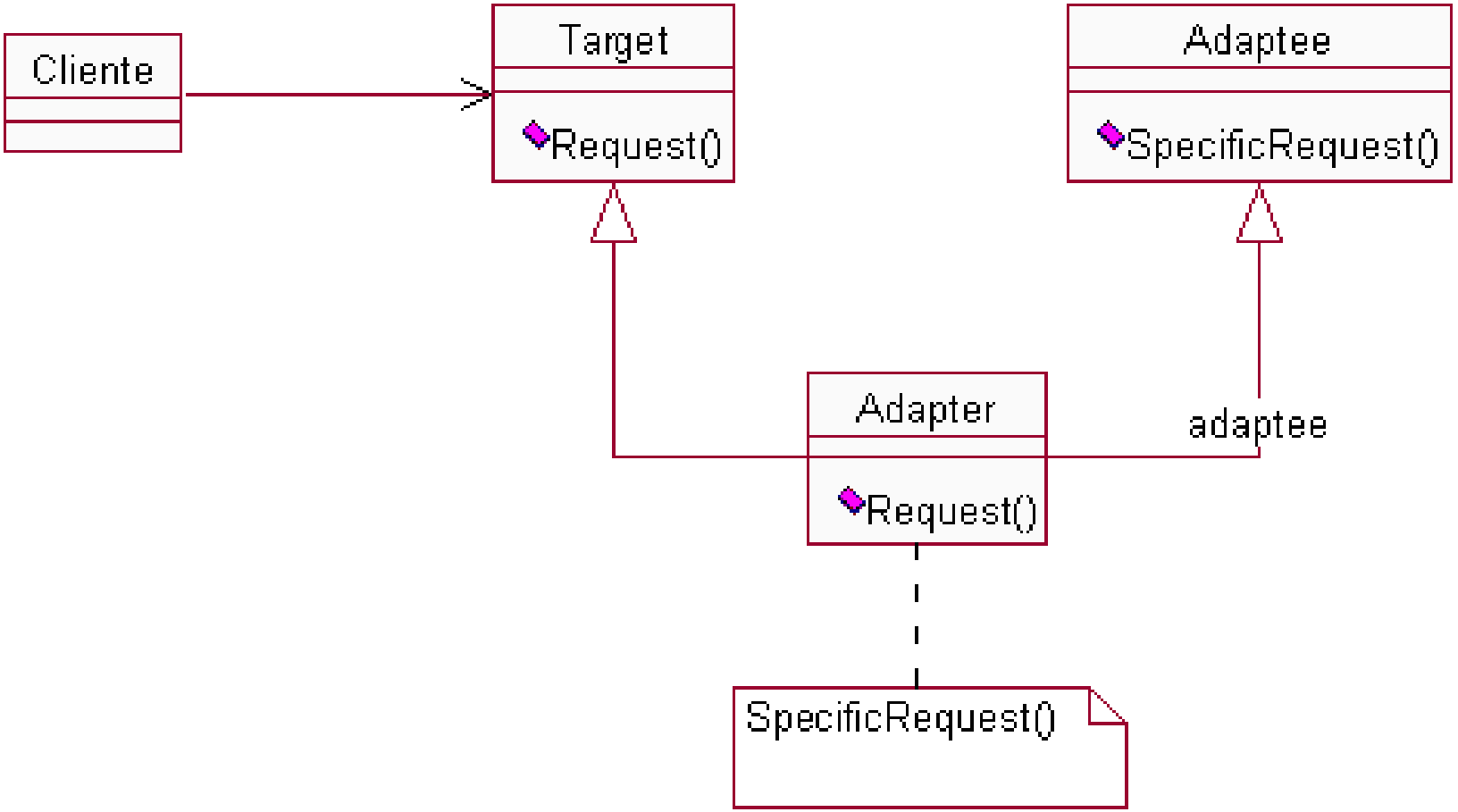
Estruturais

- Descrevem como classes e objetos podem ser combinados para formar grandes estruturas;
- Classes patterns descrevem como a herança pode ser usada para prover interfaces de programas mais comuns. Objetos patterns descrevem como objetos podem ser compostos em grandes estruturas utilizando composição ou inclusão de objetos com outros objetos.

- Converte a interface da classe em outra interface esperada pelo cliente;
- Permite que classes que não poderiam interagir devido a incompatibilidades nas interfaces possam trabalhar em conjunto.

- **É usado quando:**
  - Se quer usar uma classe já existente e sua interface não combina com a esperada pelo cliente;
  - Se quer criar uma classe reutilizável que coopera com classes não relacionadas ou não previstas, isto é, classes que não necessariamente tenham interfaces compatíveis;
  - Se necessita usar várias subclasses existentes, mas é impraticável adaptar suas interfaces fazendo um Subclassing de cada uma.

Pattern Adapter - Estrutura



- **Conseqüências:**
  - Adapta a classe Adaptee à Target pelo comprometimento com a classe concreta Adapter. Como uma consequência, a classe Adapter não funcionará quando se quiser adaptar uma classe e todas as suas subclasses;
  - Permite que a classe Adapter reimplemente alguns comportamentos da classe Adaptee, já que a classe Adapter é uma subclasse da classe Adaptee;

- **Conseqüências:**
  - Introduz apenas um objeto, e não será necessário o uso de ponteiros adicionais para chegar a classe Adaptee;
  - Dificulta reimplementar o comportamento da classe Adaptee. Será necessário criar subclasses Adaptee para que o objeto Adapter as referencie ao invés da própria classe Adaptee;

- **Conseqüências:**
  - Deixa um único objeto Adapter trabalhar com várias classes Adaptee, quer dizer, a própria classe Adaptee e todas as suas subclasses (se houver). O objeto Adapter também pode adicionar funcionalidades a todas as classes Adaptee de uma só vez.

- **Patterns Relacionados:**
  - O pattern Bridge tem uma estrutura similar a de um objeto Adapter, mas possui um propósito diferente: separar uma interface de sua implementação de forma que ambas possam variar de modo fácil e independente, enquanto que um Adapter tem a intenção de compatibilizar a interface de um objeto previamente existente;

- **Patterns Relacionados:**
  - Decorator agrega funcionalidades a outro objeto sem alterar sua interface. Por isso, é mais transparente para a aplicação do que um Adapter. Como a interface se mantém, o Decorator suporta composição recursiva, algo que não se aplica a objetos Adapter;
  - Proxy define uma representação ou um substituto para outro objeto e não modifica a sua interface.

Pattern Adapter - Problema

- Escreva uma classe que permita que o cliente VectorDraw, que já usa a classe Shape, use a RasterBox (e Coords) para obter os mesmos dados.

```
public class VectorDraw {
    ...
    Shape s;
    int x = s.getX(), heigth = s.getHeight();
    ... }
public class Shape {
    protected int x, y, heigth, width;
    public int getX() { return x; }
    public int getY() { return y; }
    public int getHeight() { return height; }
    public int getWidth() { return width; }}
public class RasterBox {
    private Coords topLeft, bottomRight;
    public Coords getTopLeft() { return topLeft; }
    public Coords getBottomRight() { return bottomRight; }}
public class Coords { public int x, y; }
```

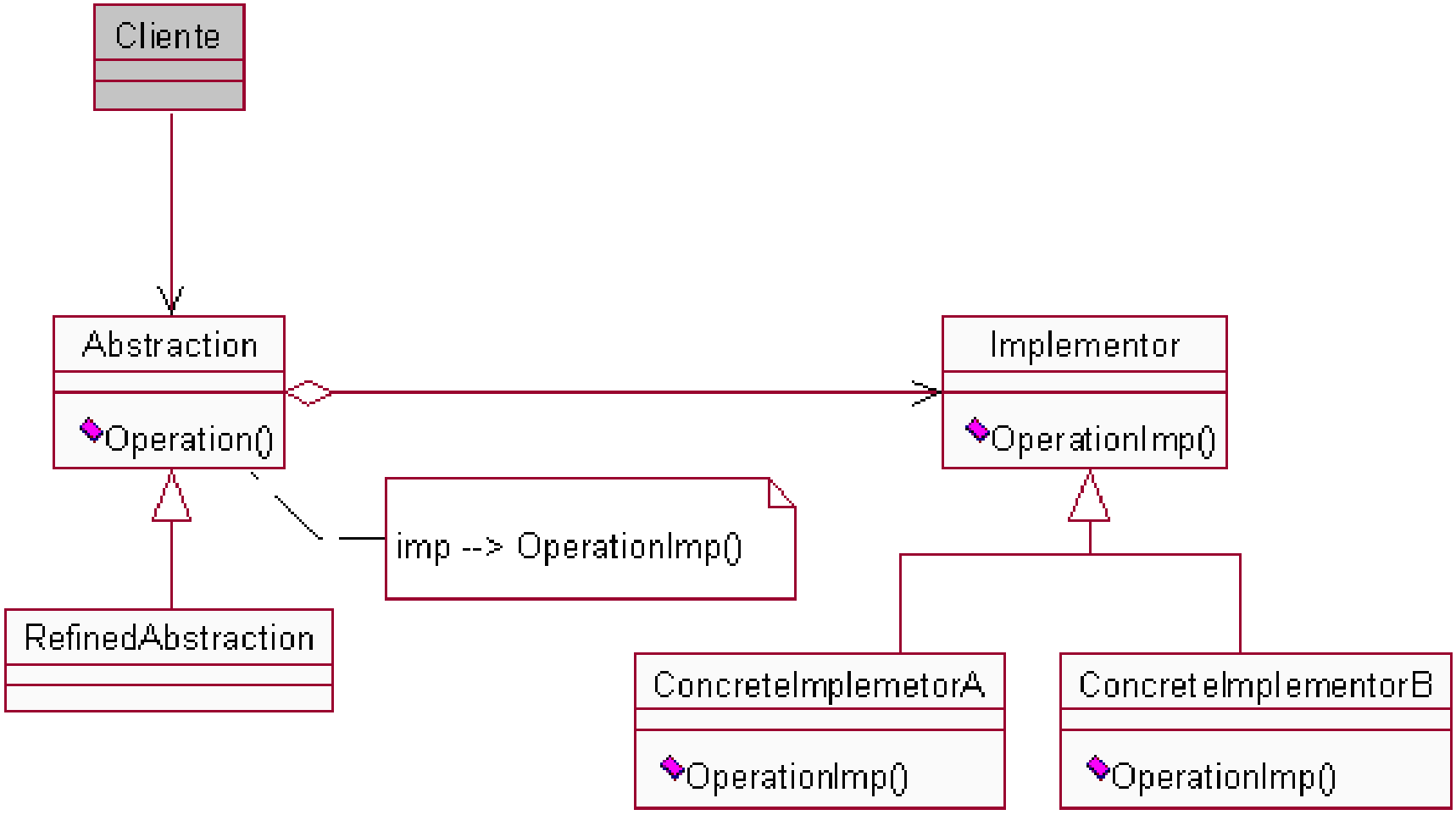
- **Desacopla uma abstração de sua implementação de tal modo que ambos possam variar independentemente.**

- **É usado quando:**
  - Se quer evitar uma ligação permanente entre uma abstração e sua implementação. O que ocorre, por exemplo, quando a implementação deve ser selecionada ou trocada em tempo de execução;
  - Tanto a abstração quanto sua implementação devem ser extensíveis por especialização. Neste caso, o pattern Bridge deixa você combinar estas diferentes abstrações e implementações e estendê-las independentemente;

- É usado quando:
  - Mudanças na implementação de uma abstração não devem ter impacto nos clientes, isto é, seu código não deve ter que ser recompilado;
  - Se quer esconder completamente a implementação dos clientes, evitando que a representação de uma classe seja feita através da interface dessa classe;

- É usado quando:
  - Há um tipo de hierarquia de classes onde ocorrem generalizações aninhadas, indicando a necessidade de se dividir um objeto em duas partes;
  - Se quer compartilhar uma implementação entre vários objetos, e este fato deva estar oculto para os clientes.

- **Estrutura:**


Cliente
Abstraction
Operation()
Implementor
OperationImp()
imp --> OperationImp()
RefinedAbstraction
ConcreteImplemetorA
OperationImp()
ConcreteImplementorB
OperationImp()

- **Conseqüências:**
  - Desacopla interface e implementação;
  - Melhora a extensibilidade;
  - Oculta detalhes de implementação do cliente.

Pattern Bridge - Problema

- Deseja-se ligar a estrutura abstrata com real, de modo que no cliente final consiga implementar um novo livro sem ser obrigado a implementar todos os métodos das classes abstratas:

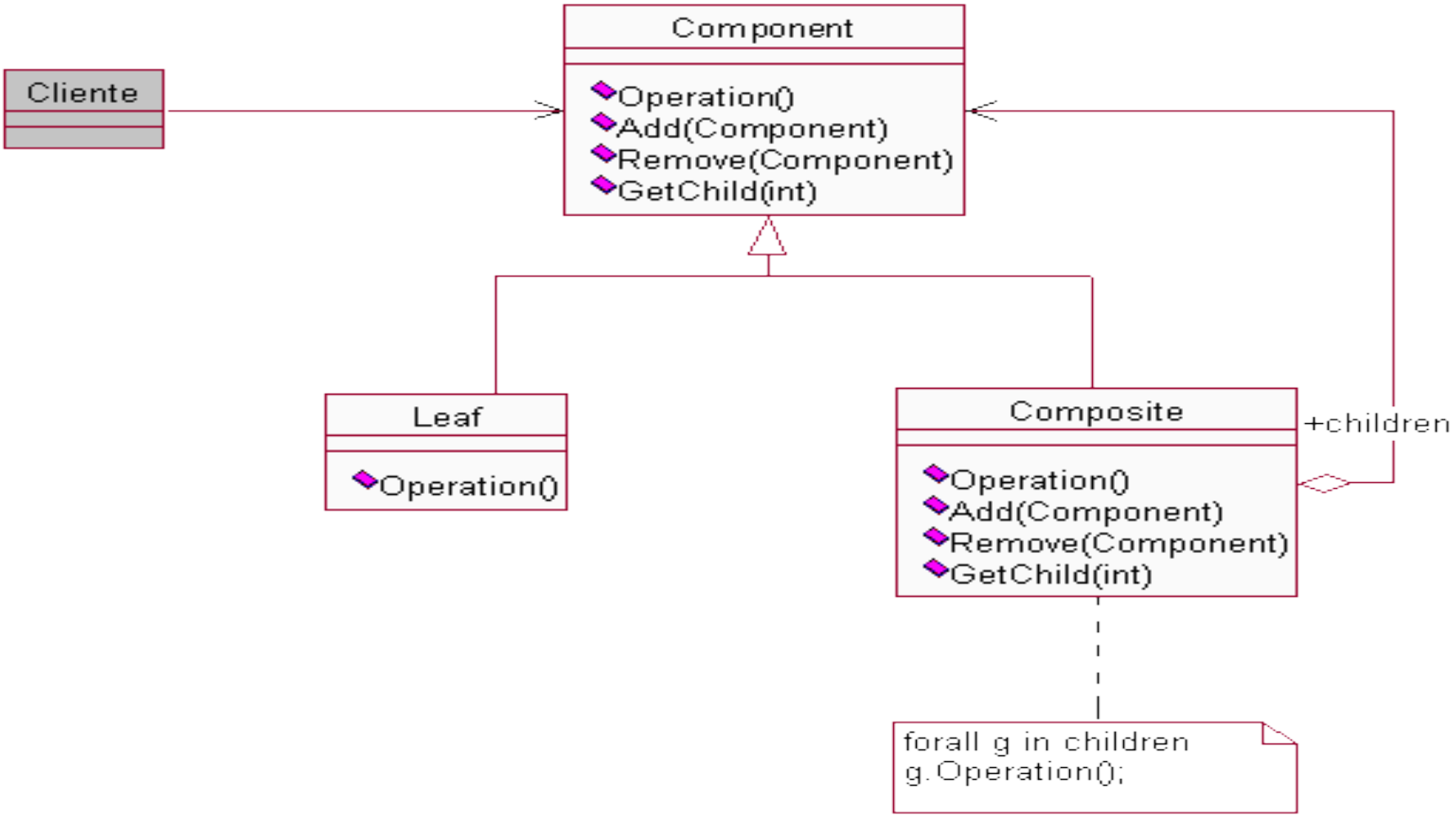
```
// Estrutura abstrata
abstract class Publicacao {
    métodos abstract get/setTitulo()  { ... }
    métodos abstract get/setAutor() { ... }}
abstract class Livro extends Publicacao {
    métodos abstract get/setISBN() { ... }}

// Estrutura real
public class LivroImpl extends Livro {
    métodos get/setTitulo() { ... }  // obrigado
    métodos get/setAutor(id) { ... } // obrigado
    métodos get/setISBN() { ... }    // obrigado
    outros métodos
}
```

- **Compõe objetos em estruturas de árvore para representar hierarquias do tipo todo-parte;**
- **Deixa os clientes tratarem objetos individuais e composições de objetos do mesmo modo.**

- **É usado quando:**
  - **Se quer representar hierarquias de objetos do tipo todo-parte;**
  - **Se quer que os clientes ignorem a diferença entre composições de objetos e objetos individuais.**

Pattern Composite - Estrutura



- **Conseqüências:**
  - **Define uma hierarquia de classes que consiste de objetos primitivos (Leaf) e objetos compostos (Composite).**
  - **Torna o cliente simples. O cliente pode tratar estruturas compostas e objetos individuais uniformemente. Os clientes normalmente não sabem (e nem devem se preocupar) se eles estão tratando comum componente individual ou composto.**

- **Consequências:**
  - **Se torna mais fácil adicionar novos tipos de componentes. Os clientes não precisam ser modificados devido a criação de novas classes “Component”.**
  - **Pode tornar o projeto, como um todo, mais geral. A desvantagem de tornar mais fácil adicionar novos componentes, é que isso torna mais difícil restringir que tipo de componentes podem fazer parte de uma composição.**

Pattern Composite - Problema

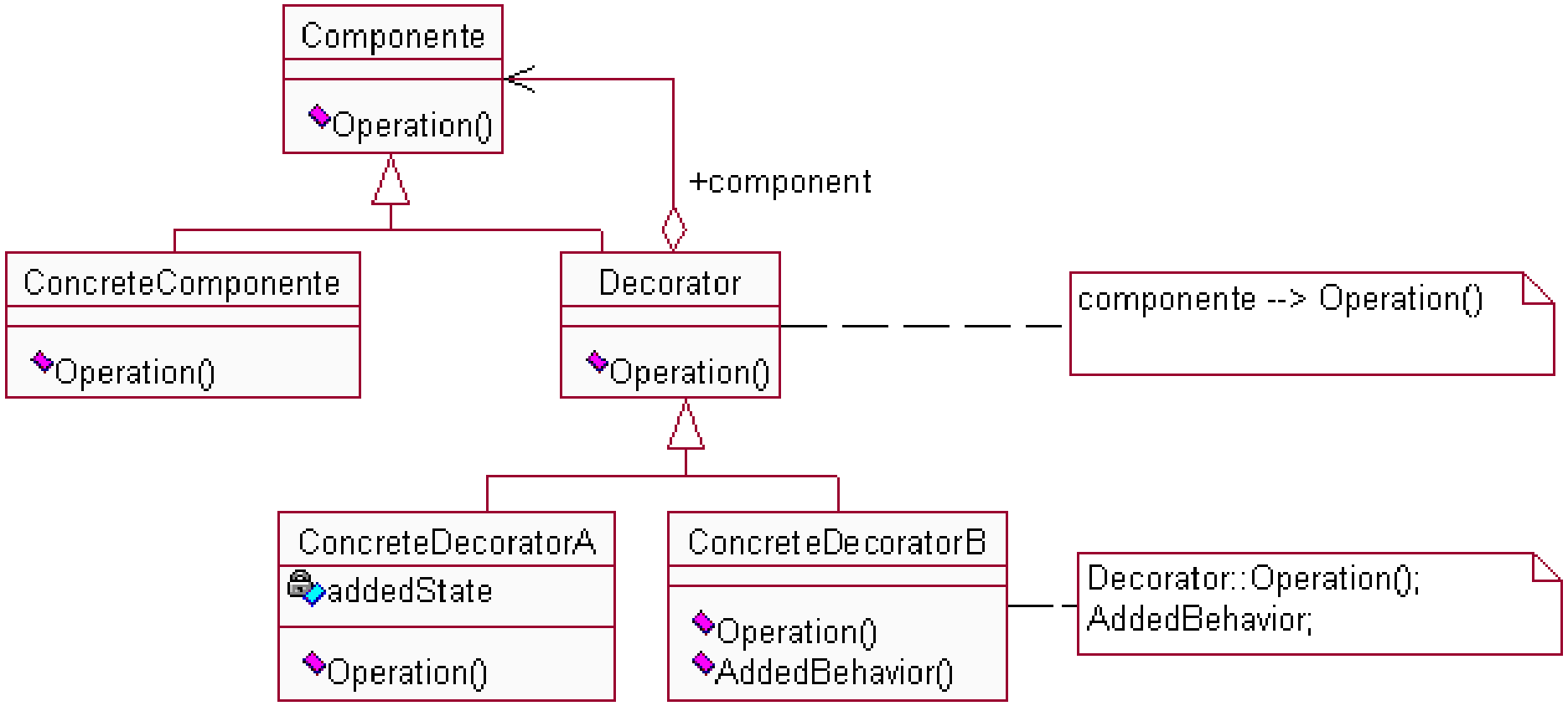
- Escreva uma classe que permita uma agregação de diversos componentes de máquina, tendo como base na seguinte classe:

```
public class MachineComponent {
    private String nome;
    public String getNome() {
        return this.nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
}
```

- **Agrega responsabilidades adicionais a um objeto dinamicamente;**
- **Provê uma alternativa flexível à criação de subclasses para estender funcionalidades (troca herança por delegação).**

- **É usado quando:**
  - **Queremos adicionar responsabilidades a objetos individuais de forma dinâmica e transparente, isto é, sem afetar outros objetos;**
  - **Queremos retirar responsabilidades;**
  - **Extensões através de subclasses são impraticáveis.**

Pattern Decorator - Estrutura



- **Consequências:**
  - **Maior flexibilidade do que herança estática;**
  - **Evita sobrecarga de características em classes de alto nível da hierarquia;**
  - **Um Decorator e seu componente são idênticos, não confie na identidade de objetos ao usá-lo;**
  - **Multidão de pequenos objetos: fácil de configurar, difícil para compreender e debugar.**

Pattern Decorator - Problema

- A partir da seguinte estrutra de classes, obtenha o mesmo resultado de modo racional

```
public class A { public void doIt() { System.out.print('A'); }}
public class AwithX extends A {
    public  void doIt() { super.doIt(); doX(); }
    private void doX()  { System.out.print('X'); }}
public class AwithY extends A {
    public void doIt() { super.doIt();  doY(); }
    public void doY()  { System.out.print('Y'); }}
public class AwithZ extends A {
    public void doIt() { super.doIt(); doZ(); }
    public void doZ()  { System.out.print('Z'); }}
public class AwithXY extends AwithX {
    private AwithY obj = new AwithY();
    public void doIt() { super.doIt(); obj.doY(); }}
public class AwithXYZ extends AwithX {
    private AwithY obj1 = new AwithY();
    private AwithZ obj2 = new AwithZ();
    public void doIt() { super.doIt(); obj1.doY(); obj2.doZ(); }}
```

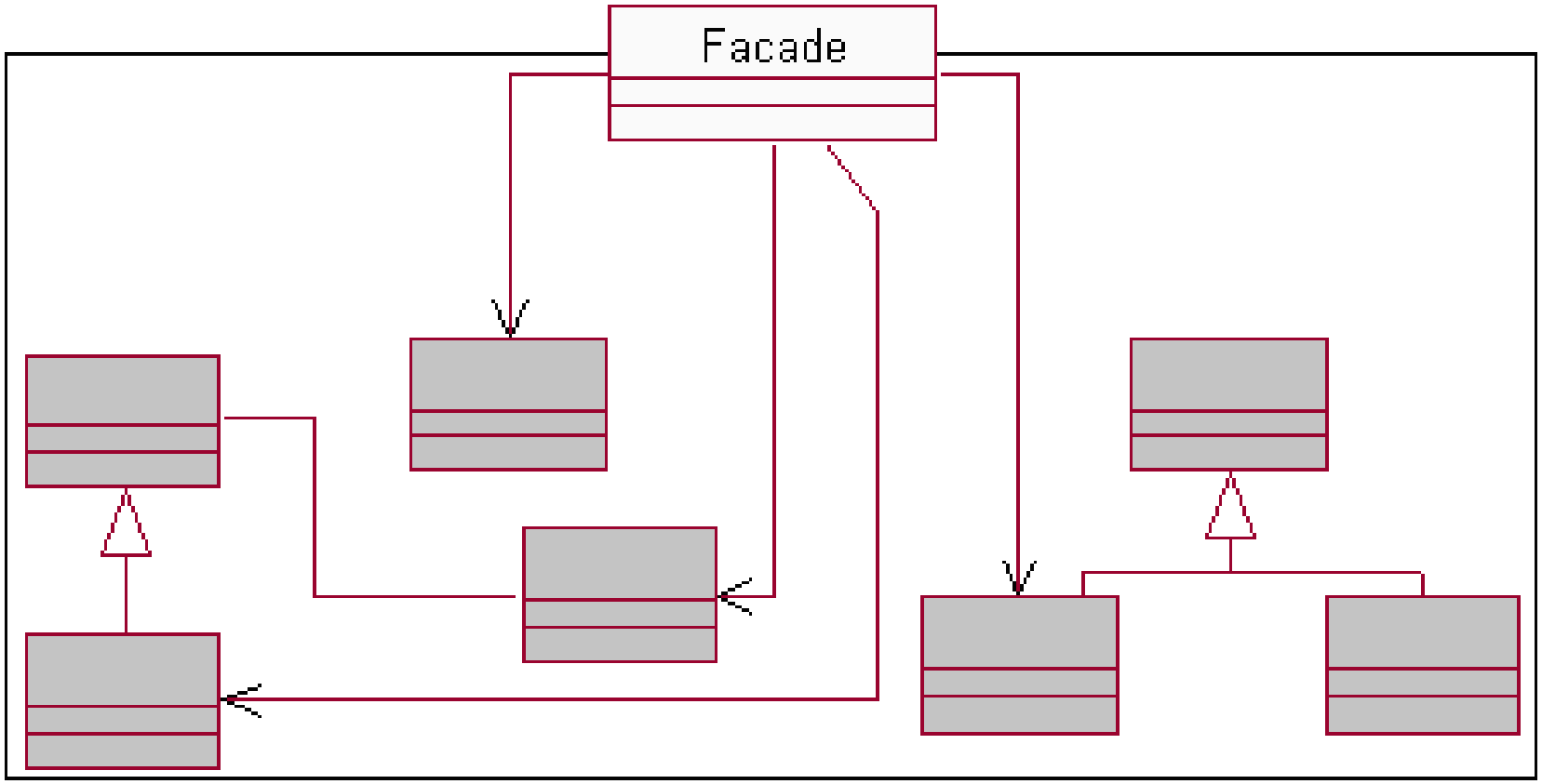
Pattern Façade

- **Provê uma interface unificada para um conjunto de interfaces em um subsistema;**
- **Define uma interface de mais alto nível que torna mais fácil o uso do subsistema.**

- **É usado quando:**
  - **Existem muitas dependências entre clientes e as classes de implementação de uma abstração. A introdução de um Façade irá desacoplar o subsistema dos clientes e dos outros subsistemas.**
  - **Quando se deseja obter subsistemas em camadas. Use um Façade para definir um ponto de entrada para cada nível do subsistema.**

- **É usado quando:**
  - **Se quer oferecer uma interface simples para um subsistema complexo.**
  - **Um Façade pode disponibilizar uma vista padrão simples do subsistema que é boa o suficiente para a maioria dos clientes. Apenas clientes que precisem de maiores peculiaridades é que necessitariam olhar além do Façade.**

Pattern Façade - Estrutura



- **Conseqüências:**
  - **Isola os clientes dos componentes do subsistema, reduzindo, desse modo, o número de objetos com que o cliente interage e fazendo com que o subsistema seja muito mais fácil de se usar;**
  - **Ajuda a estruturar em camadas não só o sistema, assim como as dependências entre os objetos;**

- **Conseqüências:**
  - **Promove um fraco acoplamento entre o subsistema e seus clientes. Um baixo acoplamento permite que se varie os componentes de um subsistema sem afetar seus clientes;**
  - **Podem eliminar referências complexas ou circulares entre objetos, o que é fundamental quando o cliente e subsistema são implementados separadamente.**

- **Conseqüências:**
  - **Não impede as aplicações de usarem um subsistema de classes se elas precisarem. Assim pode-se escolher entre facilidade de uso e generalidade.**

Pattern Façade - Problema

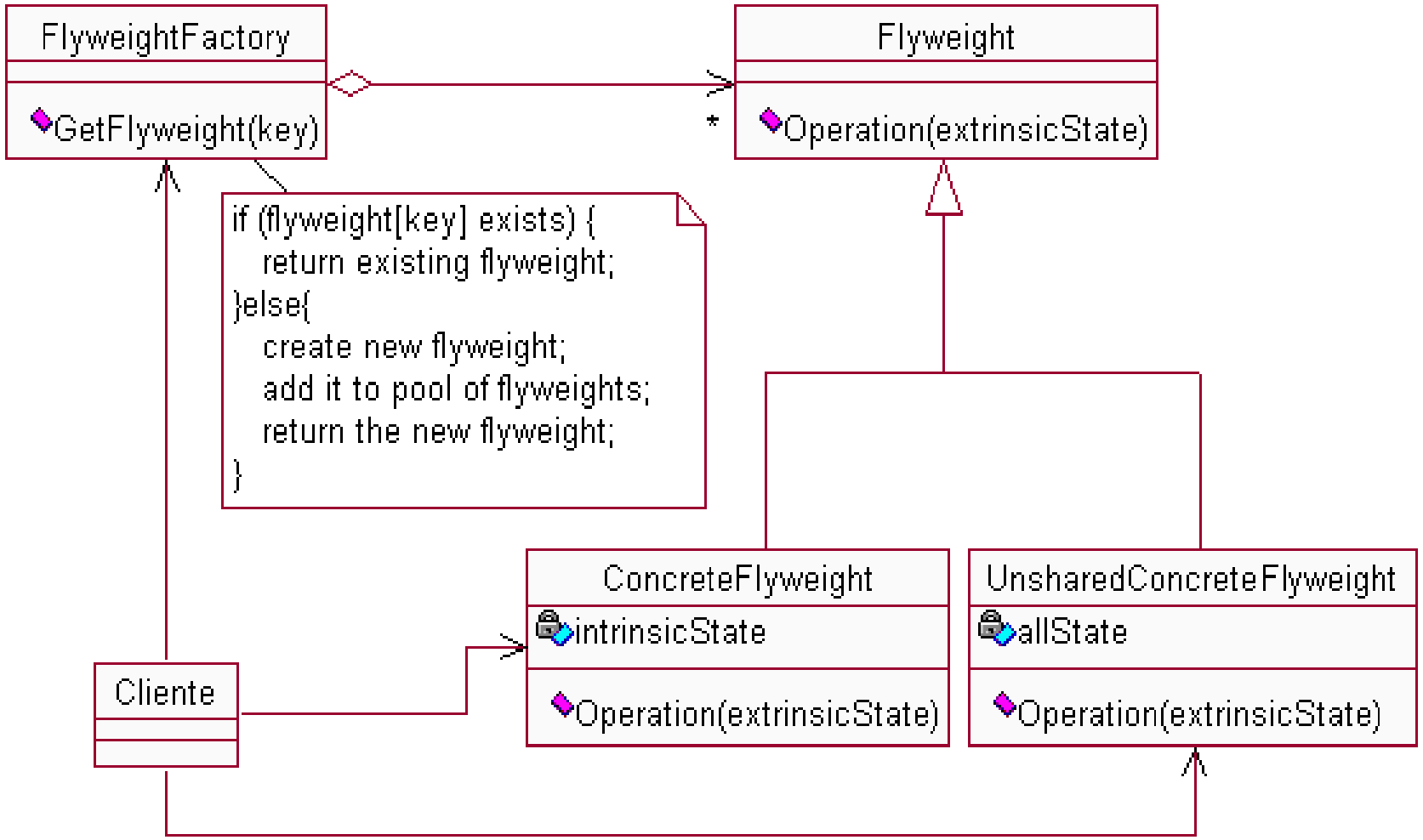
- Em um comércio foi definido as seguintes classes, e as aplicações precisam fazer ações tais como, registrar novos clientes, comprar produtos e fechar compra:

```
public class Carrinho {
    void adicionar(Produto p) {...}
    double getTotal() {...}}
public class Cliente {
    Cliente(String nome, int id) {...}
    void adicionarCarrinho(Carrinho c) {...}
    Carrinho getCarrinho() {...}}
public class Produto {
    Produto(String nome, int id,double preco) {...}
    double getPreco() {...}}
public class BancoDeDados {
    Produto selecionarProduto(int id) {...}
    void processarPagamento() {...}}
```

- **Usa compartilhamento para suportar eficientemente um grande número de objetos de fina granularidade.**

- **É usado quando:**
  - Uma aplicação usa um grande número de objetos;
  - Custos de armazenamento são grandes por causa da grande quantidade de objetos;
  - A maior parte do estado dos objetos podem se tornar extrínsecos;
  - A aplicação não depende da identidade do objeto.

Pattern Flyweight - Estrutura



- **Conseqüências:**
  - Pode introduzir alguns custos em tempo de execução associados a transferência, procura estados extrínsecos, especialmente se eles forem formalmente armazenados como estados intrínsecos.
  - A economia de armazenamento é função de diversos fatores: Redução do número total de instâncias que se obtém com o compartilhamento, a quantidade de estado intrínseco por objeto, se o estado extrínseco é computado ou armazenado.

Pattern Flyweight - Problema

- Como gerar uma matriz de 6x10, se a necessidade de 60 objetos da classe “Posicao”?

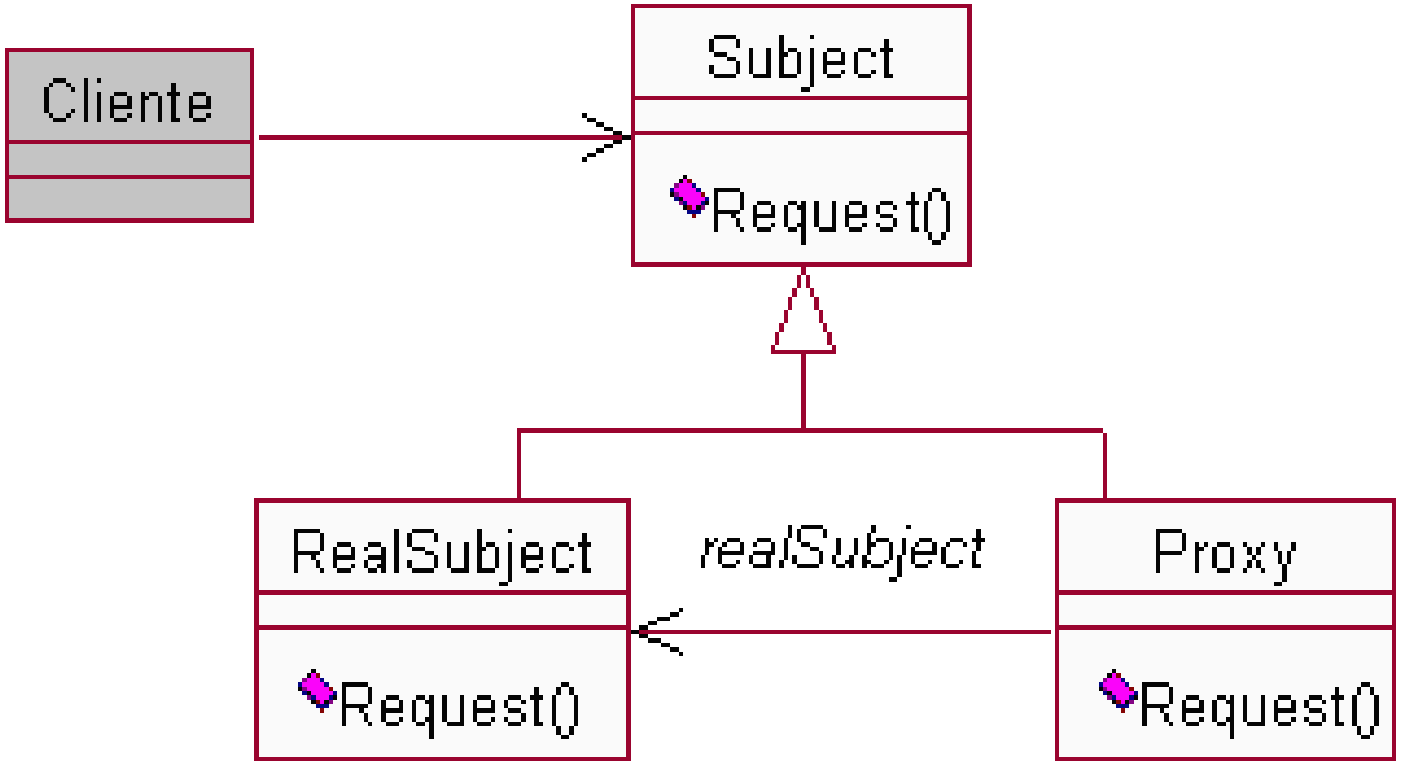
```
public class Posicao {
    private static int num = 0;
    private int row, col;
    public Posicao(int maxPerRow) {
        row = num / maxPerRow; col = num % maxPerRow; num++;
        System.out.println("Criado Objeto " + row + col); }
    public void report() { System.out.print( " " + row + col );}
}
public class Demo {
    public static final int ROWS = 6, COLS = 10;
    public static void main(String [] args) {
        Posicao[][] matrix = new Posicao[ROWS][COLS];
        for (int i=0; i < ROWS; i++) for (int j=0; j < COLS; j++)
                matrix[i][j] = new Posicao(COLS);
        for (int i=0; i < ROWS; i++) {
            for (int j=0; j < COLS; j++) matrix[i][j].report();
            System.out.println();
}}}
```

- **Provê um substituto ou um receptáculo para um objeto, com o objetivo de controlar como esse objeto é acessado.**

- **É usado quando:**
  - **Um Remote Proxy provê um representante local para um objeto em um espaço de endereçamento diferente;**
  - **Um Virtual Proxy cria objetos custosos, por consumirem grande número ou quantidade de recursos, sob demanda.**

- **É usado quando:**
  - Um Protection Proxy controla o uso do objeto original. Muito útil quando clientes devem ser diferentes permissões de acesso ao objeto;
  - Um Smart Reference é uma substituição por simples ponteiro que faz ações adicionais quando um objeto é acessado.

Pattern Proxy - Estrutura



Pattern Proxy - Conseqüências

- **O pattern Proxy introduz um nível de intermediação para acessar um objeto. Essas intermediações adicionais tem vários usos, dependendo do tipo do Proxy: O Remote Proxy pode esconder o fato de que um objeto reside em um espaço de endereçamento diferente, o Virtual Proxy pode fazer otimizações, tais como a criação de objetos sob demanda, tanto o Protection Proxy quanto o Smart Proxy permitem que sejam feitas tarefas adicionais quando um objeto é acessado.**

Pattern Proxy - Problema

- Um determinado sistema precisa acessar um objeto Real:

```
public class Crente {
    public void perguntar(String pergunta) {
        SerSupremo.responder(pergunta);
    }
}
```

- Entretanto o objeto não está disponível (remoto, inacessível...)

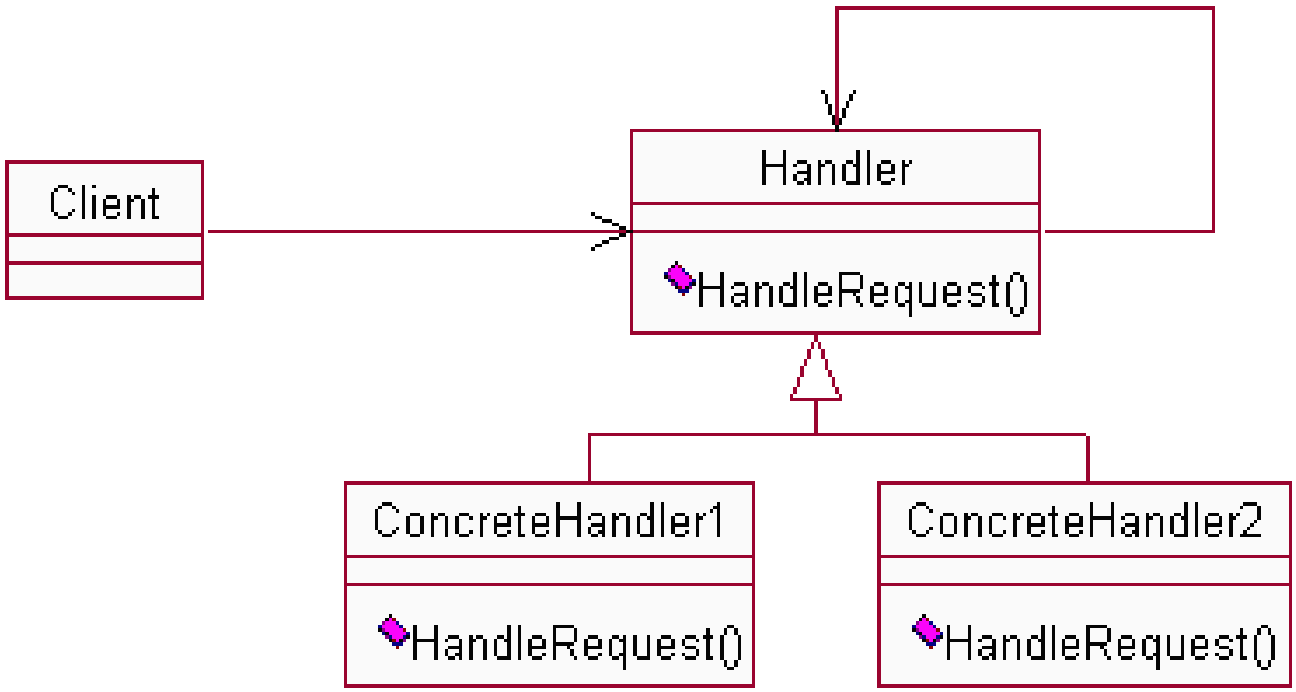
- São patterns que estão mais especificamente voltados para a comunicação entre objetos.

Pattern Chain of Responsability

- Evitar o acoplamento de um transmissor com uma requisição aos seus receptores, fazendo com que mais de um projeto, tenha a chance de manipular esta requisição.
- Encadeia os objetos receptores e passa a requisição ao longo dessa cadeia até que um objeto possa manipulá-lo.

- **É usado quando:**
  - Mais de um objeto pode tratar uma requisição, e estes não são conhecidos a priori. O Objeto a tratar a requisição deve ser definido automaticamente;
  - Queremos emitir uma requisição para um dos vários objetos envolvidos, sem especificá-lo explicitamente;
  - O conjunto de objetos que pode manipular uma requisição deve ser especificado dinamicamente.

Pattern Chain of Responsability - Estrutura



- **Conseqüências:**
  - Redução do acoplamento;
  - Maior flexibilidade na associação de responsabilidades aos objetos;
  - A recepção e tratamento da requisição não é garantida.

- **Patterns Relacionados:**
  - É normalmente utilizado em Composite, onde o pai do componente atua como o seu sucessor.

Pattern Chain of Responsability - Problema

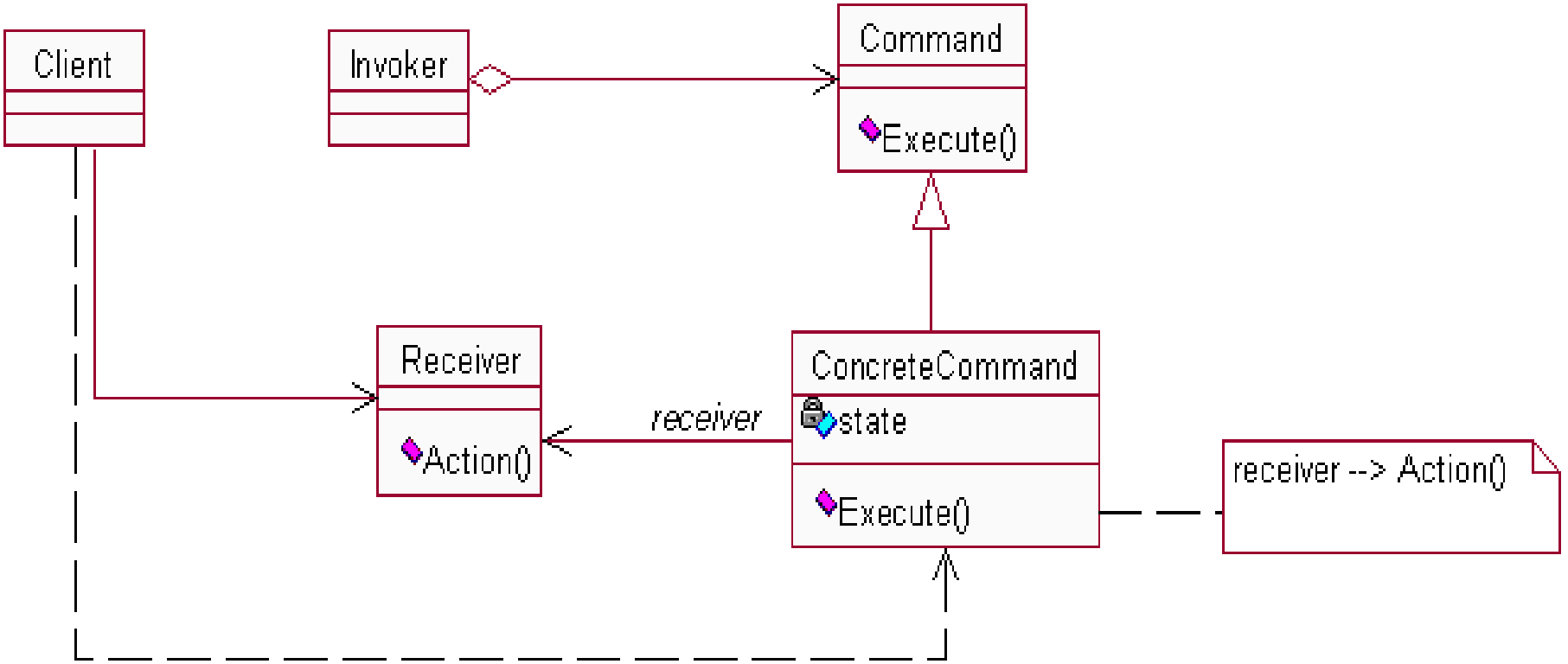
- **Nas classes abaixo exige muito esforço para o cliente administrar uma lista de Processadores:**

```
interface Image { String processo(); }
class P1 implements Image { public String processo() { return "P1"; }}
class P2 implements Image { public String processo() { return "P2"; }}
class Processador {
    private java.util.Random rn = new java.util.Random();
    private int nextId = 1; private int id = nextId++;
    public boolean handle(Image img) {
        if (rn.nextInt(2) != 0) {
            System.out.println("Processador: " + id + " está ocupado");
            return false; }
        System.out.println("Processador " + id + " - " + img.processo());
        return true; }}
public class Cliente {
    public static void main( String[] args ) {
        Image[] input = { new P1(), new P1(), new P2(), new P1(), new P2()};
        Processador[] procs = {
            new Processador(), new Processador(), new Processador()};
        for (int i=0, j; i < input.length; i++) { j = 0;
            while ( ! procs[j].handle( input[i] )) j = (j+1) % procs.length;
}}}
```

- Encapsula uma requisição como um objeto, desse modo podemos parametrizar os clientes com diferente pedidos, enfileirando ou fazendo um log de pedidos.
- Suporta operações de “undo”.

- **É usado quando:**
  - Queremos parametrizar objetos por uma ação a ser executada;
  - Queremos especificar, enfileirar ou executar requisições em diferentes momentos. Um objeto Command tem um tempo de vida independente da requisição original;

Pattern Command - Estrutura



- **Conseqüências:**
  - Desacopla o objeto que invoca a operação daquele que sabe como executá-la;
  - Podem ser manipulados e extendidos como qualquer outro objeto, pois são objetos de primeira classe.
  - Podem ser utilizados para construir comandos compostos, que em geral são uma instância de um Composite;
  - Facilita adicionar novos comandos, porque não é necessário modificar as classes existentes.

- **Patterns relacionados:**
  - Um Memento pode manter o estado requerido, para que o Command desfaça seus efeitos;
  - Comandos que precisam ser copiados antes de serem colocados numa lista de histórico atua como o Prototype.

Problema

- Um cliente precisa padronizar as ações de incl. e excl. em um banco de dados para isto conta com as seguintes classes:

```
public interface Receiver { void action(); }
public class IncluirReceiver implements Receiver {
    public void action() { System.out.println("Incluindo..."); }
}
public class ExcluirReceiver implements Receiver {
    public void action() { System.out.println("Excluindo..."); }
}
```

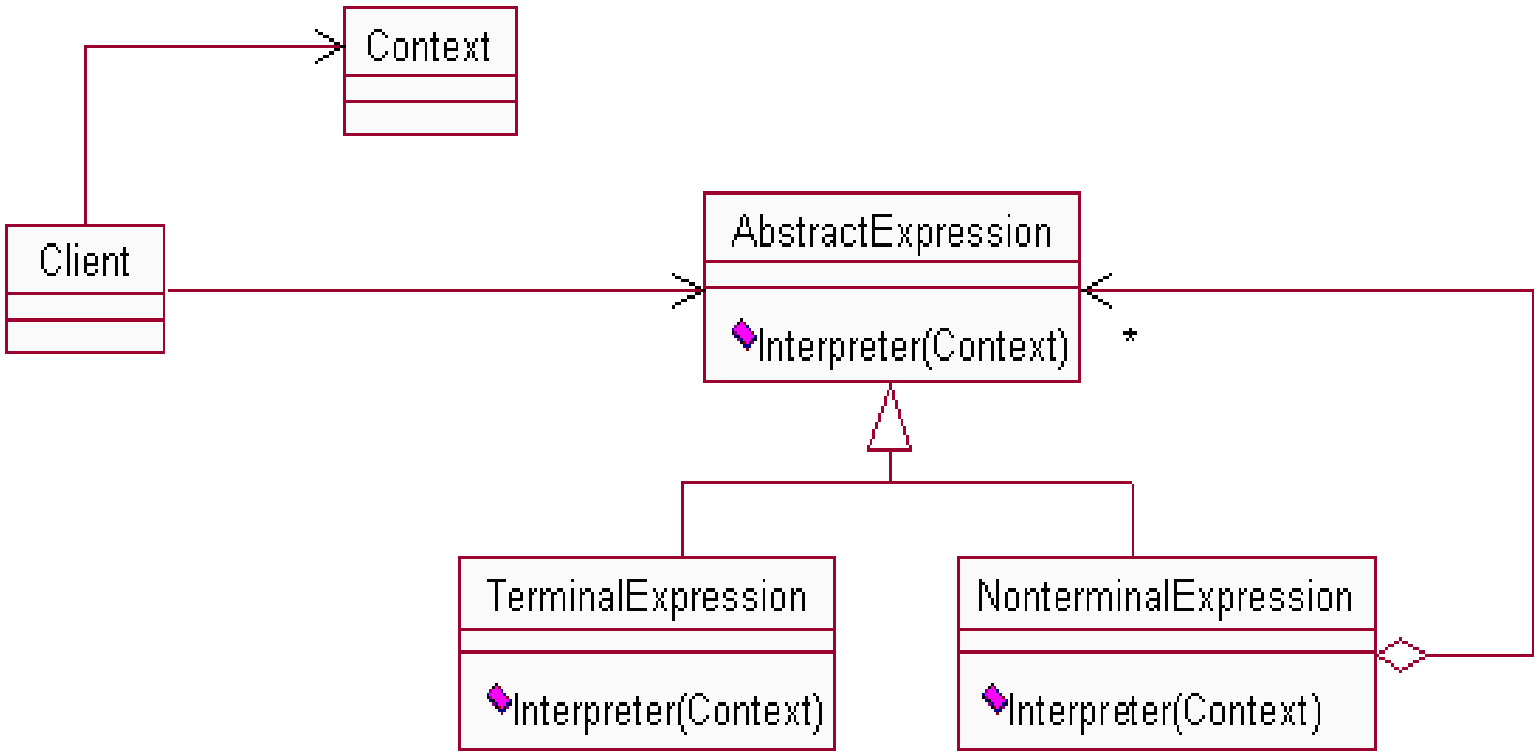
- Como fazer para que este cliente possa executar estas ações a partir da seguinte interface:

```
public interface Command {
    void setReceiver(Receiver receiver);
    Receiver getReceiver();
    void execute();
}
```

- Dada uma linguagem, define uma representação para sua gramática, juntamente com um interpretador que usa essa representação para interpretar sentenças da gramática.

- É usado quando:
  - A gramática é simples;
  - A eficiência não é o ponto crítico.

Pattern Interpreter - Estrutura



- **Conseqüências:**
  - É fácil para mudar, implementar e extender uma gramática;
  - Adiciona novos caminhos para interpretar expressões.

- **Patterns Relacionados:**
  - Uma árvore de sintaxe abstrata é uma instância do pattern Composite;
  - Flyweight mostra como compartilhar símbolos terminais dentro da árvore de sintaxe abstrata;
  - Pode usar um Iterator para percorrer a estrutura;
  - O comportamento em cada nó na árvore pode ser mantido pelo Visitor.

- **Implementar as regras nas classes descritas:**

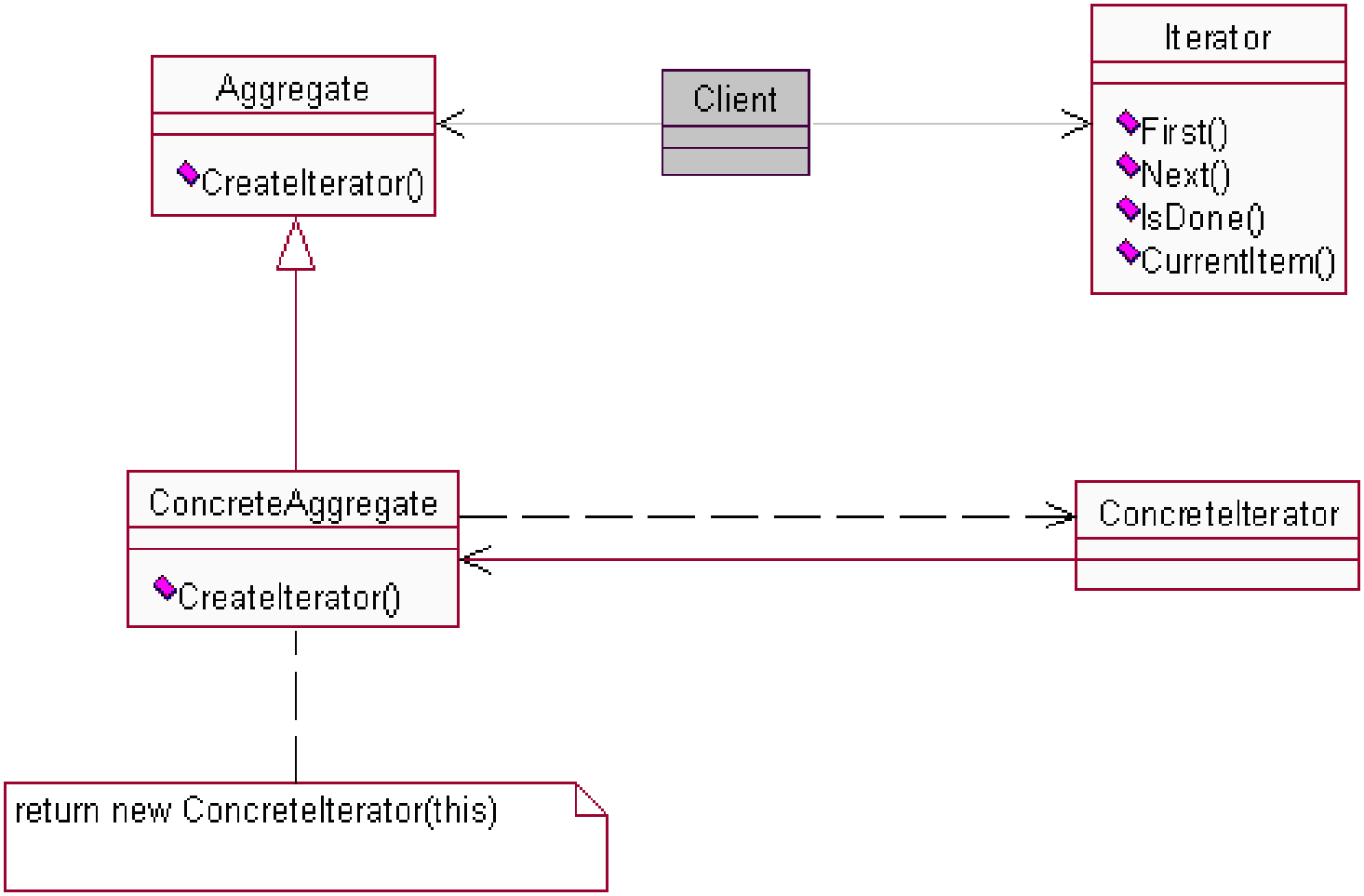
```
constant ::= '0'|'1'|...|'9'| {'0'|...|'9'}*|
{'0'|...|'9'}*'.'{'0'|...|'9'}*
variable ::= 'x'
add ::= expression '+' expression


public interface AbstractExpression { }
public class Constant implements AbstractExpression {
    private double value;
    public Constant(double value) { this.value = value; }}
public class Variable implements AbstractExpression {
    public Variable() { }}
public class Addition implements AbstractExpression {
    private AbstractExpression left, right;
    public Addition(AbstractExpression left,
        AbstractExpression right) {
        this.left = left; this.right = rigth; }}
```

- **Provê um modo de acessar seqüencialmente elementos de um objeto agregado sem expor sua representação básica.**

- É usado para:
  - Acessar o conteúdo de um objeto agregado sem expor sua representação interna;
  - Suportar múltiplas varreduras de objetos agregados;
  - Prover uma interface uniforme para varrer diferentes estruturas agregadas.

Pattern Iterator - Estrutura



- **Conseqüências:**
  - Suporta variações no percurso de um agregado;
  - Simplifica a interface do Aggregate;
  - Mais que um caminho pode estar pendente em um agregado.

- **Patterns Relacionados:**
  - Iterators são freqüentemente aplicados a estruturas recursivas tais como o Composite;
  - Iterators polimórficos delegam aos Factory Methods a instanciação da subclasse do Iterator apropriado;
  - Pode usar um Memento para capturar o estado da iteração. O Iterator armazena um Memento internamente.

Pattern Iterator - Problema

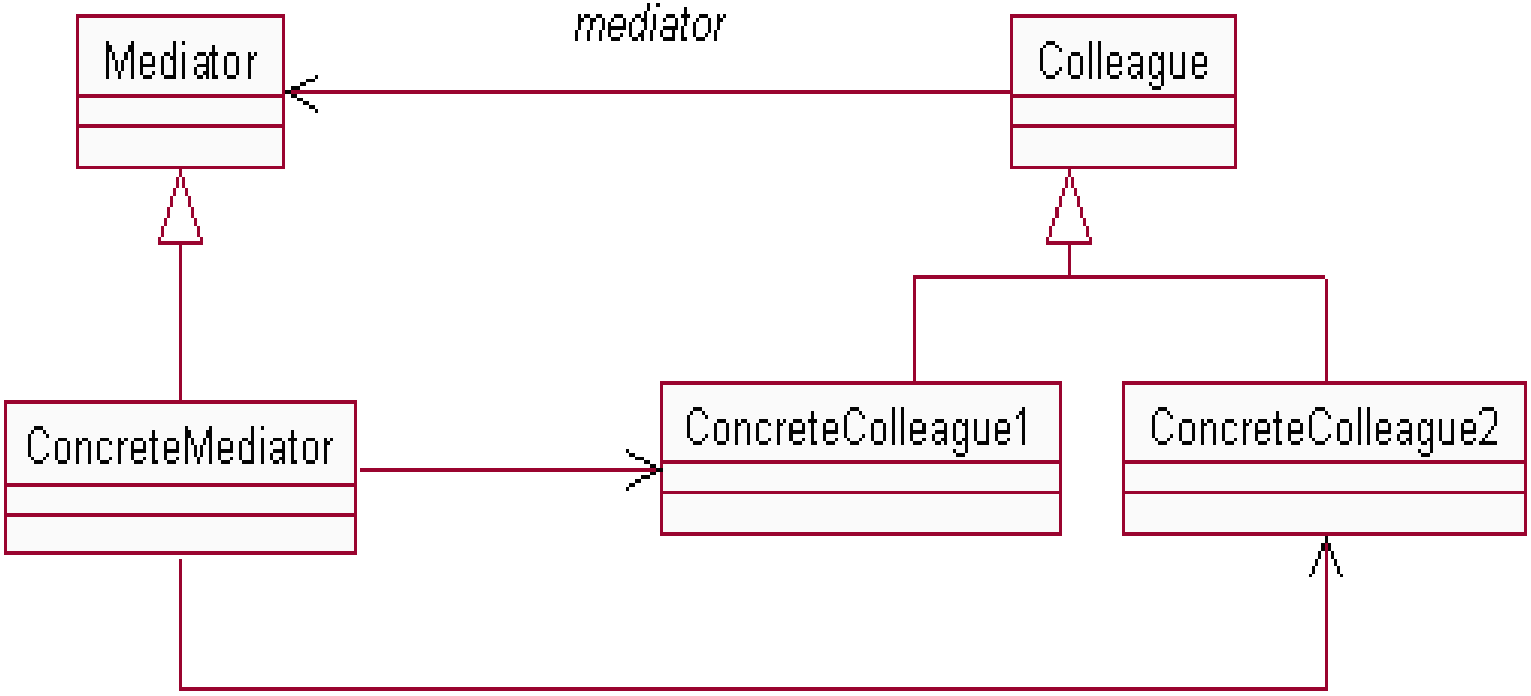
- Dada a seguinte classe Funcionário com os dados em um Array de Objetos desejamos navegar padronizadamente por esta estrutura, evitando o laço descrito abaixo:

```
public class Funcionario {
    private String nome; private double salario;
    public Funcionario(String nome, double salario) {
        this.setNome(nome); this.setSalario(salario);}
    public String getNome() { return this.nome; }
    public double getSalario() { return this.salario; }}
public class Cliente {
    public static void main(String [] args) {
        Funcionario [] funcs = new Funcionario[2];
        funcs[0] = new Funcionario("Frank Sinatra", 3000.00);
        funcs[1] = new Funcionario("Samis David Jr", 800.00);
        for (int i = 0; i < funcs.length; i++)
          System.out.println(
          funcs[i].getNome() + " - " + funcs[i].getSalario());
}}
```

- Define um objeto que encapsula a maneira pela qual um conjunto de objetos interagem;
- Promove um baixo acoplamento por fazer com que os objetos não façam referências explícitas a outros;
- Permite que a interação entre objetos seja variada independentemente.

- **É usado quando:**
  - Um conjunto de objetos se comunicam de uma maneira bem definida porém complexa, resultando em interdependências desestruturadas e difíceis de entender;
  - O reuso de um objeto é difícil por causa de suas referências e comunicação com outros objetos;
  - Um comportamento que é distribuído entre várias classes deve ser customizado sem um conjunto de subclasses.

Pattern Mediator - Estrutura



- **Conseqüências:**
  - Limita Subclassing;
  - Provê baixo acoplamento entre Colleagues;
  - Simplifica protocolos de objetos;
  - Abstrai a maneira como os objetos cooperam;
  - Centraliza o controle.

- **Patterns Relacionados:**
  - O Facade difere do Mediator porque o primeiro abstrai um subsistema de objetos para prover uma interface mais conveniente. Este protocolo é unidirecional, isto é, os objetos Facade fazem requisições de um subsistema de classes mas o contrário não é verdade. De outro lado, o Mediator possibilita comportamentos cooperativos que Colleague objetos não fazem ou não podem prover, e o protocolo é multidirecional;

- **Patterns Relacionados:**
  - Colleagues podem se comunicar com o Mediator usando o pattern Observer.

Pattern Mediator - Problema

- Permitir que o seguinte grupo de objetos se comunique entre si sem que haja acoplamento entre eles?
- Remover o forte acoplamento presente em relacionamentos muitos para muitos?
- Permitir que novos participantes sejam ligados ao grupo facilmente?

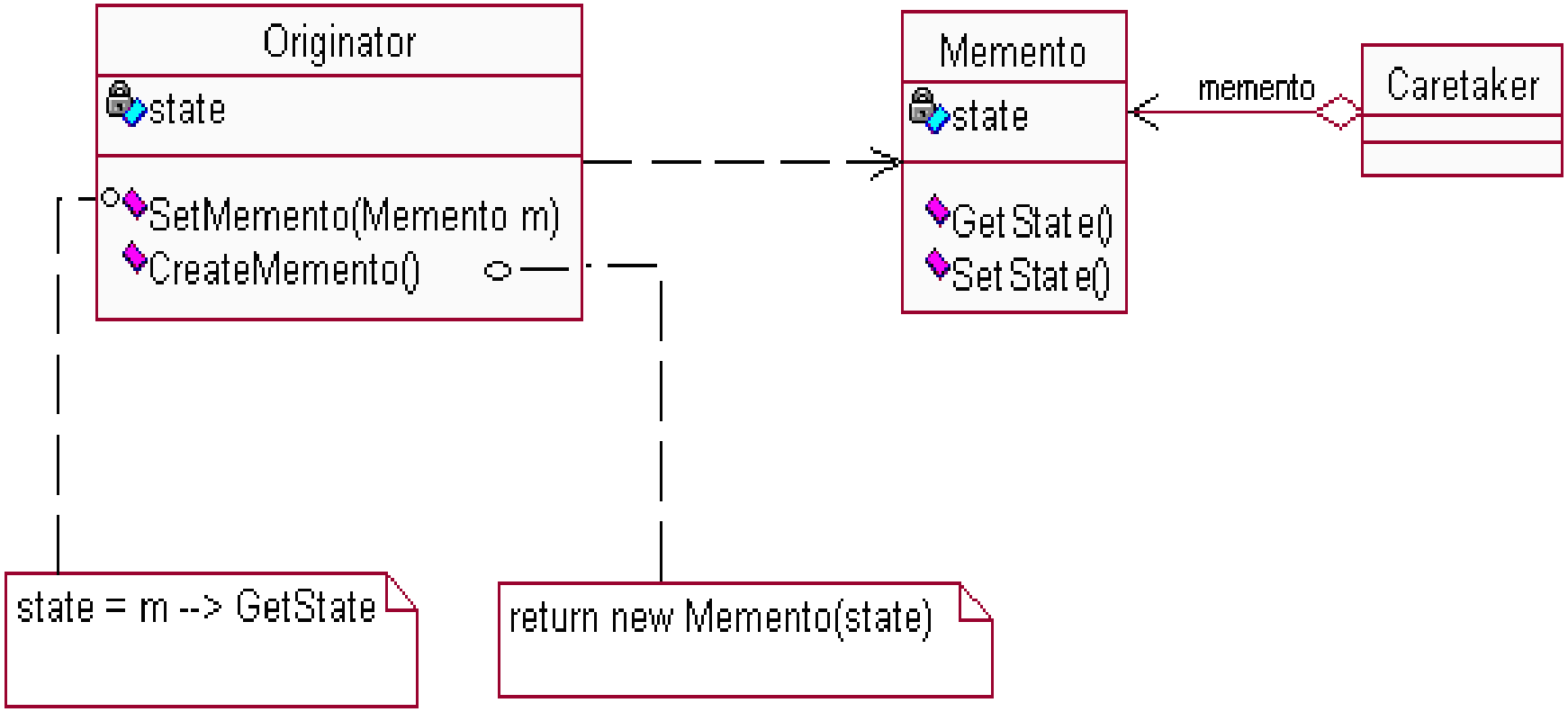
```
public class Rato {
    public void escaparGato() {
        ...
    }
}
public class Gato {
    public void perseguirRato() {
        ...
    }
}
```

Pattern Memento

- **Sem violar o encapsulamento, captura e externaliza o estado interno de um objeto até que ele possa ser restaurado para este estado mais tarde.**

- **É usado quando:**
  - O estado instantâneo de um objeto deve ser salvo até que ele possa ser restaurado para aquele estado mais tarde;
  - Um acesso direto a interface para obter o estado do objeto fizesse com que fossem expostos detalhes de implementação, quebrando assim o encapsulamento do objeto.

Pattern Memento - Estrutura



- **Conseqüências:**
  - Preserva o encapsulamento;
  - Simplifica o Originator;
  - Seu uso é caro;
  - Esconde os custos no Carataker para o memento.

- **Patterns Relacionados:**
  - Commands podem ser usados para manter estados para operações de “undo”;
  - O memento pode ser usado para interação.

Pattern Memento - Problema

- Baseado na descrição da classe Funcionario, crie uma modo de restaurar o salário ao seu valor original:

```
public class Funcionario {
    private String nome;
    private double salario;
    public Funcionario(String nome, double salario) {
        this.setNome(nome);
        this.setSalario(salario);
    }
    public String getNome() { return this.nome; }
    public double getSalario() { return this.salario; }
    public void setNome(String nome) { this.nome = nome; }
    public void setSalario(double salario) {
        this.salario = salario;
    }
    public String toString() { return nome + " " + salario; }
}
```

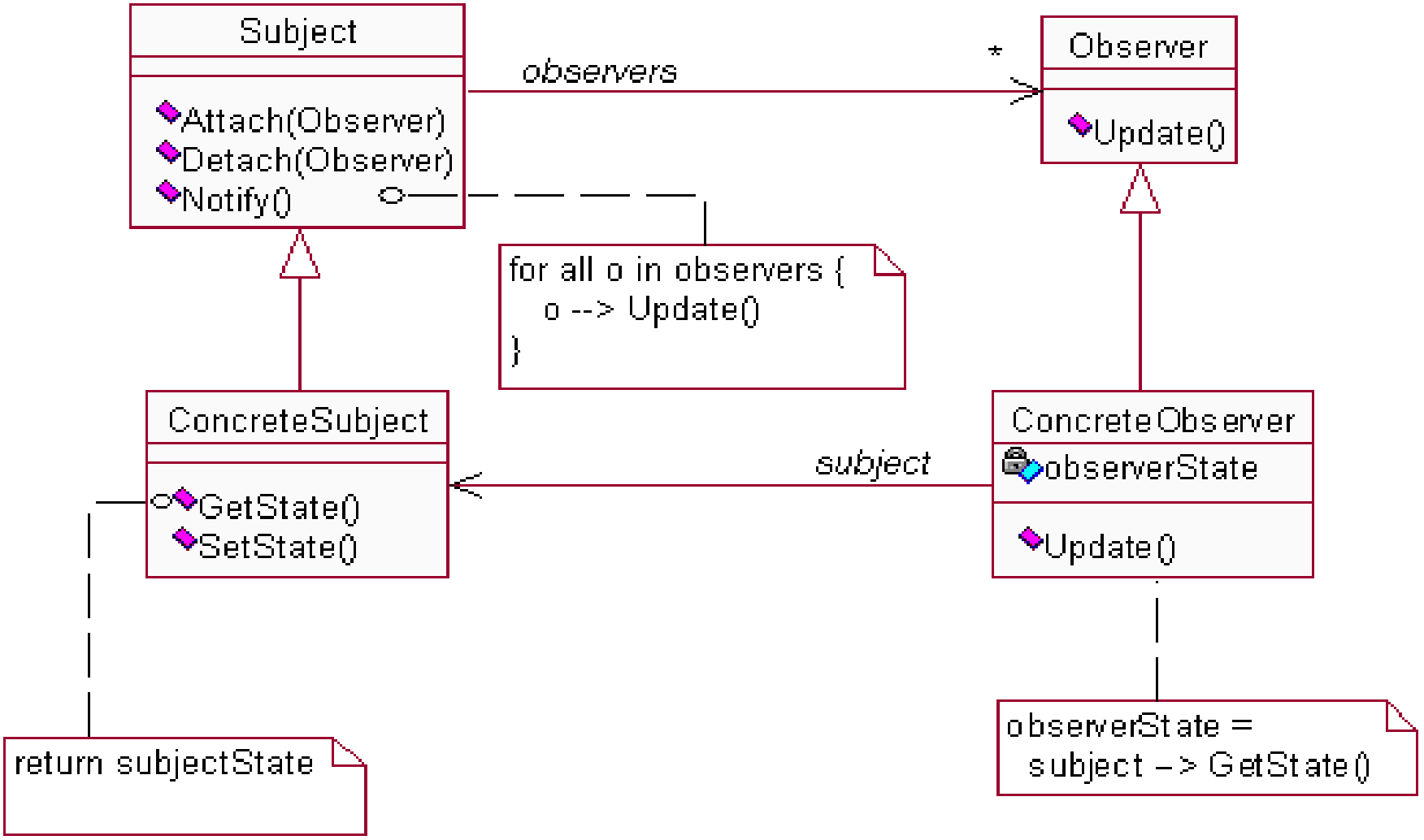
Pattern Observer

- Define uma dependência de um-pra-muitos entre objetos, dessa forma, quando um objeto muda de estado, todas as suas dependências são notificadas e atualizadas automaticamente.

- **É usado quando:**
  - Uma abstração tem dois aspectos, um dependente do outro. Encapsulando-se esses aspectos em objetos separados fará com que se possa variá-los e reusá-los independentemente;
  - Uma mudança em um objeto requer uma mudança em outros, e não se sabe como esses outros objetos efetivamente fazem suas mudanças;

- **É usado quando:**
  - Um objeto deve poder notificar outros objetos sem assumir nada sobre eles. Dessa forma evita-se que os objetos envolvidos fiquem fortemente acoplados.

Pattern Observer - Estrutura



- **Conseqüências:**
  - Permite o acoplamento entre Subject e Observer;
  - Suporta comunicação broadcast;
  - Atualizações inesperadas.

- **Patterns Relacionados:**
  - O ChangeMnager pode usar o patterns Singleton para fazer com que seu acesso seja único e global;
  - Quando o relacionamento de dependência entre Subjects e Observers é particularmente complexo, é necessário um objeto que mantenha este relacionamento. Este objeto é chamado ChangeManager. Ele propõe minimizar o trabalho requerido para fazer com que os observers reflitam as mudanças ocorridas nos Subjects. Nesse caso o ChangeManager atua como um Mediator entre Subjects e Observers.

Pattern Observer - Problema

- Como garantir que varios objetos da classe “Observador” que dependem de um objeto da classe “Sujeito” fiquem em dia com mudanças:

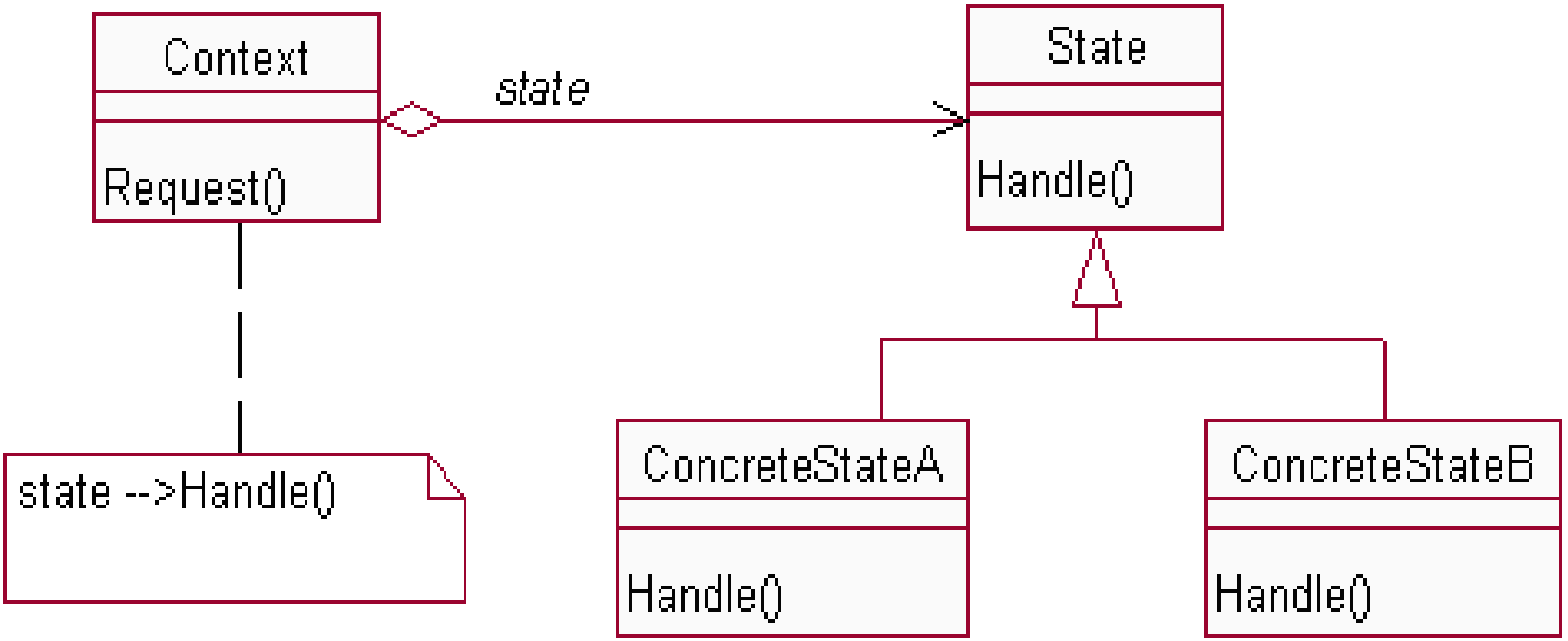
```
class Sujeito{
    private Observador observador;
    private int dado = 2;
    public int getDado() { return this.dado; }
    public void setObservador(Observador observador) {
        this.Observador = Observador;
    }
}
class Observador {
    public void atualizar(Sujeito o) {
        ... // deve atualizar o  Sujeito
    }
}
```

- Permite que um objeto altere seu comportamento quando seu estado interno mudar.

- É usado quando:
  - O comportamento de um objeto depende de seu estado, e este deve ser mudado em tempo de execução conforme as mudanças ocorridas em seu estado;

- **É usado quando:**
  - Operações que possuem comandos condicionais muito grandes, que dependem do estado do objeto. Este estado é usualmente representado por uma ou mais constantes enumeradas. Freqüentemente, muitas operações irão conter a mesma estrutura condicional. O pattern State coloca cada ramo dessa estrutura em uma classe separada. Dessa maneira, o estado do objeto pode ser tratado como um objeto com seus próprios direitos que podem variar independentemente dos outros objetos.

Pattern State - Estrutura



- **Conseqüências:**
  - Localiza o comportamento de um estado específico e particiona o comportamento por diferentes estados. O pattern State coloca todo comportamento associado a um estado em particular em um objeto. Assim, todo código de um estado específico fica em uma subclasse da classe State, podendo ser adicionados novos estados e transições;

- **Conseqüências:**
  - Faz a transição de estados explicitamente;
  - Objetos State podem ser compartilhados.

- **Patterns Relacionados:**
  - O pattern Flyweight explica como e quando objetos State podem ser compartilhados. Objetos State freqüentemente são Singletons.

Pattern State - Problema

- Como eliminar a estrutura de Decisão (if ou switch) da classe Chain, preservando o mesmo resultado:

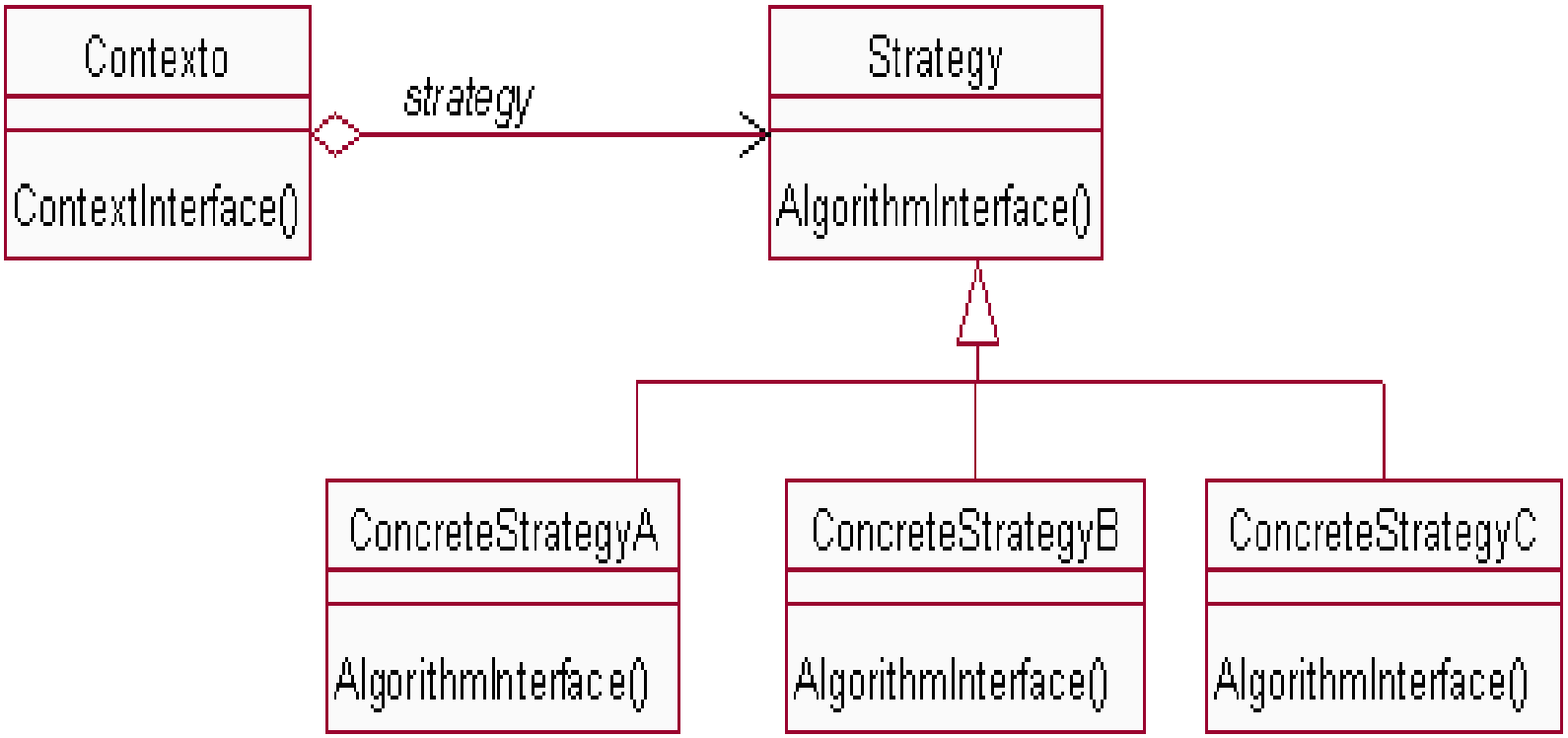
```
public class Chain {
    private int state;
    public Chain()      { state = 0; }
    public void pull() {
        if (state == 0) {
            state = 1; System.out.println( "Baixa" );
        } else if (state == 1) {
            state = 2; System.out.println( "Média" );
        } else if (state == 2) {
            state = 0; System.out.println( "Alta" );
        }
    }}
public class Cliente {
    public static void main( String[] args ) {
        Chain chain = new Chain();
        for (int i = 0; i < 6; i++) chain.pull();
}}
```

- Define uma família de algoritmos, encapsula cada um, e faz com que eles possam ser permutáveis. Strategy permite que o algoritmo varie independentemente dos clientes que o usam.

- **É usado quando:**
  - Muitas classes relacionadas diferem apenas em seus comportamentos. Strategies provêem uma maneira de configurar uma classe com um entre vários comportamentos possíveis;
  - É necessário diferentes variações de um algoritmo. Strategies podem ser usados quando essas variações são implementadas como uma classe hierárquica de algoritmos;

- **É usado quando:**
  - Um algoritmo utiliza dados dos quais o cliente não pode ter conhecimento. Strategy é usado para evitar a exposição de complexas estruturas de dados de algoritmos específicos;
  - Uma classe define muitos comportamentos, e isso faz com que haja muitas condições de estado em suas operações. Ao invés de muitas condições, move-se as condições relacionadas em ramos dentro de sua própria classe Strategy.

Pattern Strategy - Estrutura



- **Conseqüências:**
  - Famílias de algoritmos relacionados;
  - Uma alternativa ao subclassing;
  - Strategies eliminam comandos condicionais;
  - Uma escolha de implementações;
  - Os clientes devem estar cientes dos diferentes Strategies;
  - Overhead de comunicação entre Strategy e Context;
  - Aumento do número de objetos.

- **Patterns Relacionados:**
  - Objetos Strategy normalmente são bons Flyweights.

Pattern Strategy - Problema

- Você precisa elaborar classes de estratégias para “Jogos de Guerra”, baseado na classe Inimigo

```
class Inimigo {
    private int exercito;
    private boolean nuclear;
    public Inimigo(int exercito, boolean nuclear) {
        this.exercito = exercito; this.nuclear = nuclear; }
    public int getExercito() { return this.exercito; }
    public boolean isNuclear() { return this.nuclear; }
}
```

- As regras para adotar uma estratégia são:

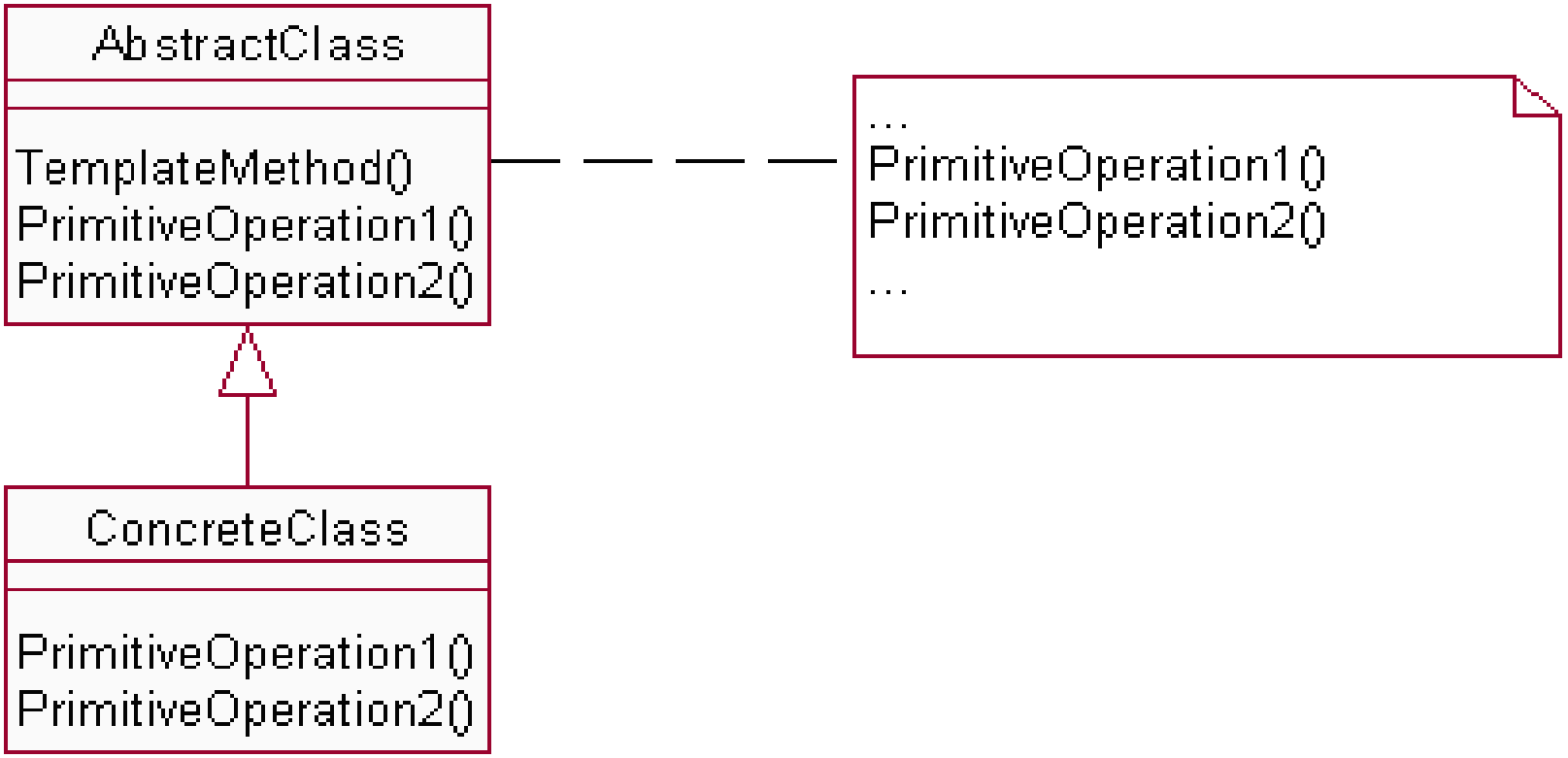
a. Caso o inimigo possua armas nucleares devemos usar de diplomacia;
b. Caso o inimigo tenha um exército maior de 10.000 elementos devemos fazer alianças com nossos amigos.
c. Caso o inimigo seja pequeno e sem armas nucleares devemos atacar sozinho.

Pattern Template Method

- Define a estrutura de um algoritmo em operação, delegando alguns passos às subclasses. Template Method permite que as subclasses redefinam certos passos de um algoritmo sem mudar a estrutura do mesmo.

- **É usado quando:**
  - Queremos implementar partes invariáveis de um algoritmo e deixar que as subclasses implementem os comportamentos variáveis;
  - Comportamentos comuns entre subclasses devem ser fatorados e localizados em uma classe comum evitando duplicação de código;
  - Queremos controlar a extensão das subclasses. Pode-se definir um Template Method que chama “hook operations” em pontos específicos, permitindo desse modo extensões apenas nesses pontos.

Pattern Template Method - Estrutura



- **Conseqüências:**
  - Chama os seguintes tipos de operações: Concrete Operations, Concrete Abstract Class Operation, Primitive Operations, Factory Methods e Hook Operations, as quais fornecem um comportamento padrão que as subclasses podem extender se necessário. Uma Operation Hook normalmente não faz nada.
  - São técnicas fundamentais de reuso de código. São importante em bibliotecas de classes, porque eles são uma maneira para fatorar o comportamento comum fora do código em bibliotecas de classes;

- **Patterns Relacionados:**
  - Factory Methods são freqüentemente chamados de Template Methods;
  - Template Methods usam herança para variar partes de um algoritmo. Strategies usam delegação para variar todo o algoritmo.

Pattern Template Method - Problema

- **Simplifique as seguintes estruturas de Classes:**

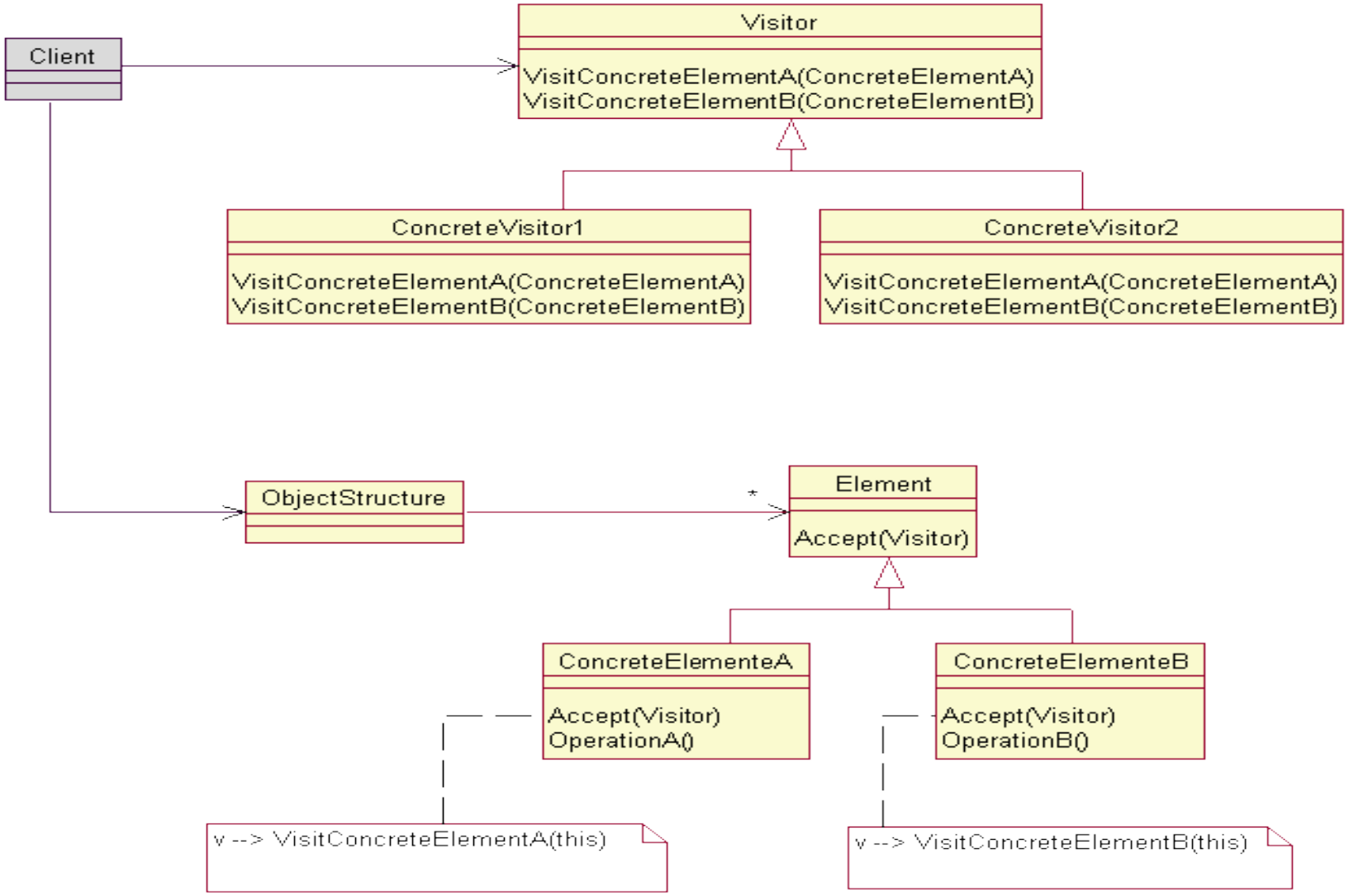
```
public class XMLData {
    public String link(String texto, String url) {
        return "<endereco xlink:href='"+url+"'>"+texto+"</endereco>"; }
    public String transform(String texto) { return texto; }
    public String show() {
        String msg = "Endereço: " + link("Empresa", "http://www.empresa.com");
        return transform(msg); }}
public class HTMLData {
    public String link(String texto, String url) {
        return "<a href='"+url+"'>"+texto+"</a>"; }
    public String transform(String texto) { return texto.toLowerCase(); }
    public String show() {
        String msg = "Endereço: " + link("Empresa", "http://www.empresa.com");
        return transform(msg); }}
public class Cliente {
    public static void main(String [] args) {
        XMLData xml = new XMLData();
        System.out.println(xml.show());
        HTMLData html = new HTMLData();
        System.out.println(html.show());
    }
}
```

- Representa uma operação a ser feita nos elementos de uma estrutura de objetos. O Visitor permite que se defina uma nova operação sem mudar a classe dos elementos nos quais ela irá operar.

- **É usado quando:**
  - Uma estrutra de objetos contém várias classes de objetos com diferentes interfaces, e se quer fazer operações nesse objetos que dependem de suas classes concretas;
  - Muitas operações distintas e não relacionadas precisam ser feitas em objetos numa mesma estrutura de objetos, e se quer evitar uma “poluição” dessas classes com essas operações. Visitor permite que se guarde operações relacionadas juntas em uma classe. Quando uma estrutura de objetos é compartilhada por várias aplicações, usa-se o Visitor para pegar operações apenas para aquelas aplicações que necessitem delas;

- **É usado quando:**
  - As classes definidas em uma estrutura de objetos raramente mudam, mas frequentemente se quer definir novas operações sobre essa estrutura. Mudando a estrutura de objetos as classes irão requerer que a interface seja redefinida para todos os Visitors, o que é potencialmente custoso. Se as classes da estrutura de objetos mudam com frequência, então é melhor que se defina operações naquelas classes.

Pattern Visitor - Estrutura



- **Conseqüências:**
  - Visitor faz com que a adição de novas operações seja bastante fácil;
  - Um Visitor reúne operações relacionadas e separa as não relacionadas;
  - A adição de novas classes ConcreteElement é difícil;
  - A visitação através de hierarquias de classes.

- **Patterns Relacionados:**
  - Visitors podem ser usados para aplicar uma operação sobre uma estrutura de objetos definida pelo pattern Composite;
  - Visitor pode ser aplicado para fazer visitação.

Pattern Visitor - Problema

- Necessitamos implementar nova funcionalidade para configurar o modem para trabalhar com os Sistemas UNIX, MAC, entre outros sistemas, criando determinados tipos de Modem para determinados Sistemas, bloqueando modems incompatíveis.

```
public interface Modem {
    public void dial(); public void send();
    public void receive();
}
public class USRobotics implements Modem {
    public void dial() { System.out.println("Discando USRobotics"); }
    public void send() { System.out.println("Enviando USRobotics"); }
    public void receive() { System.out.println("Recebendo USRobotics"); }
}
public class ThreeCOM implements Modem {
    // Mesmos métodos de USRobotics apontados para ThreeCOM
}
public class AZTech implements Modem {
    // Mesmos métodos de USRobotics apontados para AZTech
}
```

Dúvidas? Agradecimentos

Home Page
http://fernandoans.site50.net

Blog
http://fernandoanselmo.blogspot.com

X25 Home Page
http://www.x25.com.br

Fernando Anselmo

fernando.anselmo@x25.com.br