

Padrões de Projeto

1 - Padrões de Criação

1.1 Factory Method

- Este é um tipo bem comum de padrão utilizado nos programas orientados a objeto. O padrão *Factory Method* é caracterizado por retornar uma instância dentre muitas possíveis, dependendo dos dados providos a ele. Geralmente, todas as classes que ele retorna têm uma classe pai e métodos em comum, mas cada um executa tarefas diferentes e é otimizado para diferentes tipos de dados.



Padrões de Projeto

- Como já visto a definição do padrão *Factroy Method* é a seguinte:

"Define uma interface para criar um objeto, mas deixa as subclasses decidirem qual classe instanciar. O padrão *Factory Method* deixa uma classe repassar a responsabilidade de instanciação para subclasses."



Padrões de Projeto

Intenção: Definir uma interface para a criação de um objeto, deixando as subclasses decidirem que classe instanciar. O *Factory Method* delega a instanciação para as subclasses.

Motivação: Em muitas situações, uma aplicação necessita criar objetos cujas classes fazem parte de uma hierarquia de classes, mas não necessita ou não tem como definir qual a subclasse a ser instanciada. O *Factory Method* é usado nesses casos e decide com base do 'contexto', qual das subclasses ativar. Um exemplo simples: leitura de objetos serializados num arquivo.



Padrões de Projeto

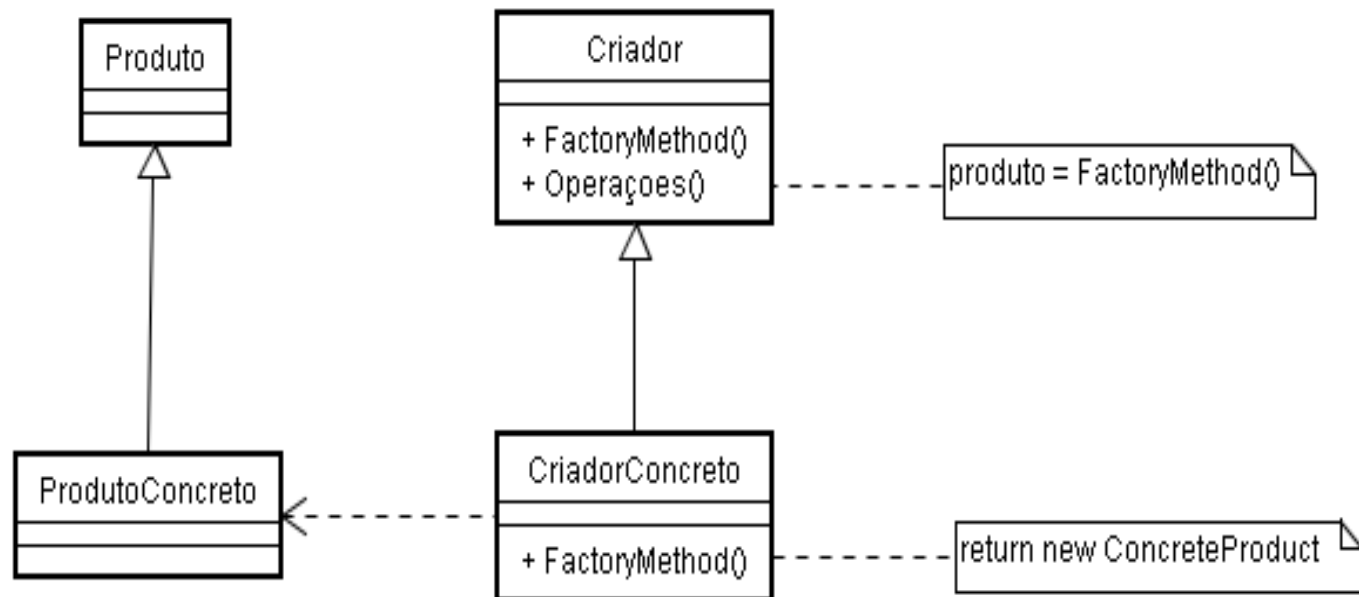
Aplicabilidade: Use esse padrão quando:

- O cliente não consegue antecipar a classe de objetos que deve criar;
- Uma classe quer que suas subclasses especifiquem os objetos que criam.



Padrões de Projeto

Estrutura



Padrões de Projeto

Participantes

Produto:

- Define a interface dos objetos criados pelo Factory Method

ProdutoConcreto:

- Implementa a interface Produto



Padrões de Projeto

Criador:

- Declara o Factory Method que retorna um objeto do tipo Produto. Criador pode também definir uma implementação por omissão do método factory que retorna por omissão um objeto ProdutoConcreto.
- Pode chamar o Factory Method para criar um produto do tipo Produto

CriadorConcreto:

- Faz override do Factory Method para retornar uma instância de ProdutoConcreto



Padrões de Projeto

Colaborações

- Criador depende das suas subclasses para definir o método de construção necessário para retornar a instância do produto concreto apropriado.

Consequências

- Fornece ganchos para as subclasses.
- Conecta hierarquias de classe paralelas.



Padrões de Projeto

1.2 – Abstract Factory

- O padrão *Abstract Factory* está relacionado com o padrão *Factory Method*, porém está a um nível de abstração maior. O padrão *Factory* é útil para se construir objetos individuais, para um propósito específico, sem que a construção requeira conhecimento das classes específicas sendo instanciadas. Se você precisar criar um grupo combinado de tais objetos, então o **padrão *Abstract Factory*** é o mais apropriado.



Padrões de Projeto

- Você pode usar este padrão quando desejar retornar uma das muitas classes de objetos relacionadas. Cada qual pode retornar diferentes objetos se requisitado. Em outras palavras, o **padrão *Abstract Factory*** é uma fábrica de objetos que retorna uma das várias fábricas.
- A definição do padrão *Abstract Factory* é a seguinte:

“Provê uma interface para criar famílias de objetos relacionados ou inter-dependentes sem especificar suas classes concretas.”



Padrões de Projeto

Intenção: fornecer uma interface para a criação de famílias de objetos relacionados ou dependentes sem especificar suas classes completas.

Motivação: em muitas situações uma aplicação cliente precisa criar determinados objetos cuja construção efetiva só é definida em tempo de execução. A aplicação cliente não deve se preocupar com a criação dos objetos.



Padrões de Projeto

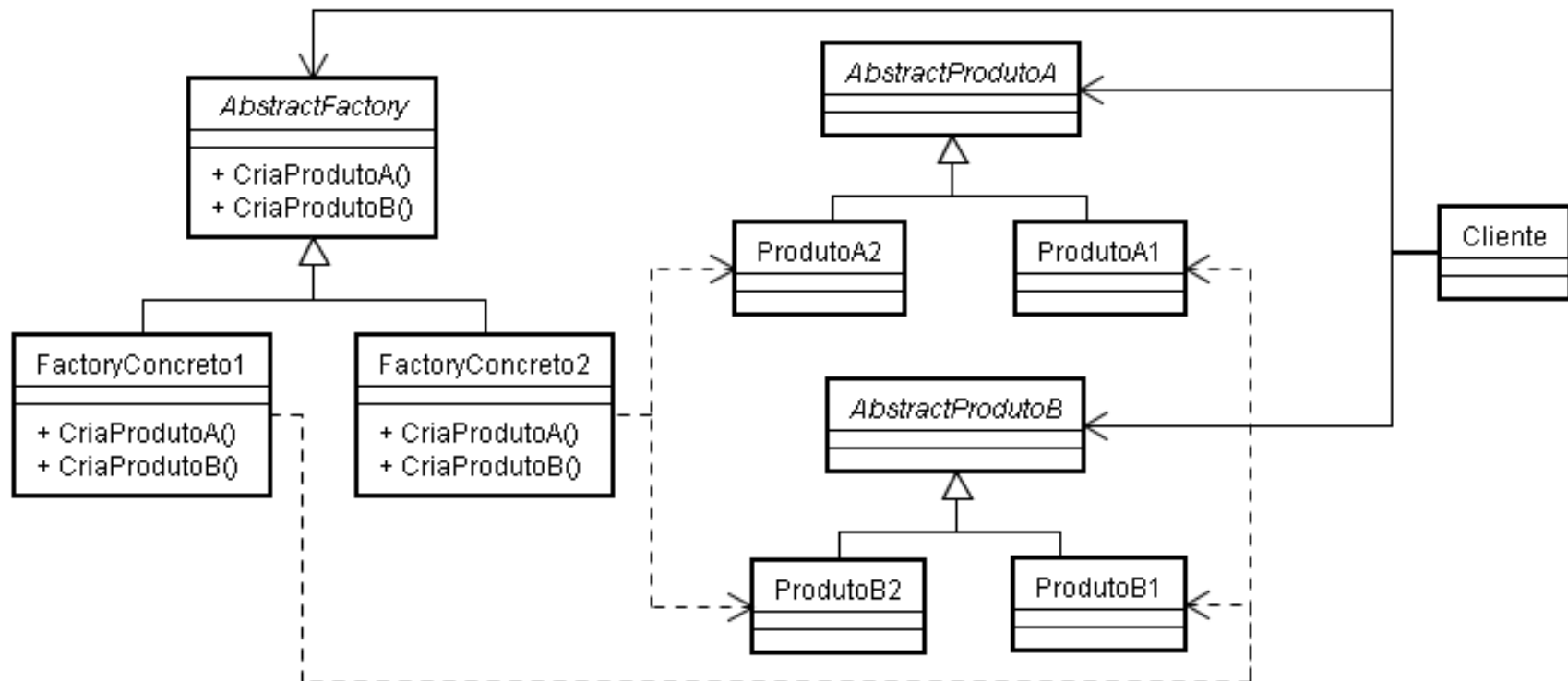
Aplicabilidade: Use esse padrão quando:

- O sistema deve ser independente de como seus produtos são criados, compostos ou representados;
- O sistema deve ser configurado como um produto de uma família de múltiplos produtos;
- A família de objetos produto é projetada para ser usada em conjunto;
- Deseja-se revelar apenas a interface da biblioteca de classes produto e não a sua implementação.



Padrões de Projeto

Estrutura



Padrões de Projeto

Participantes

AbstractFactory:

- Declara uma interface para operações que criam objetos para produto abstrato.

FactoryConcreto:

- Implementa as operações para criar objetos para produtos concretos.

AbstractProduto:

- Define uma interface para objetos de um tipo de objeto para produto.



Padrões de Projeto

Produto:

- Define um objeto produto a ser criado pela FactoryConcreta correspondente.

Cliente:

- Usa apenas interfaces definidas por AbstractFactory e AbstractProduto.



Padrões de Projeto

Colaborações

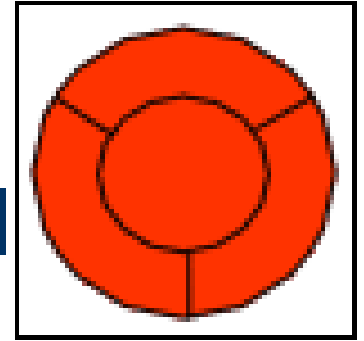
- Em tempo de execução, normalmente é criada uma única instância da classe FactoryConcreto. Ela será a responsável pela criação dos produtos concretos.
- AbstractFactory delega a criação dos objetos-produto para suas subclasses concretas (FactoryConcreto).

Consequências

- Isola as classes concretas
- Facilita a troca de famílias de produtos
- É difícil suportar novos tipos de produtos



Padrões de Projeto



1.3 – Singleton

- “Garante que uma classe tenha uma única instância e provê um ponto global de acesso à instância.”
- Acima está a descrição do **padrão Singleton** que assegura que apenas uma única instância daquela classe vai existir. Por exemplo, seu sistema pode ter apenas um gerenciador de janelas, ou gerenciador de impressão, ou então **um único ponto de acesso** ao banco de dados.
- A maneira mais fácil de fazer uma classe que possua uma única instância dela mesma é utilizar **uma variável estática** na classe, onde será guardada a referência para a instância corrente.



Padrões de Projeto

- No caso do *Singleton*, a classe dever ter um **construtor private**, ou seja, ela não poderá ser instanciada diretamente, mas sim fornecer um método comum para que a instância única da classe seja retornada. Cada vez que esse **método for chamado**, ele deve checar se já existe uma instância da classe e retorná-la, caso contrário ele deve instanciar a classe, guardar a referência ao objeto no atributo estático da classe e então retorná-lo.



Padrões de Projeto

Intenção: Garantir que uma classe tenha somente uma instância e fornecer um ponto global de acesso à mesma

Motivação: Em muitas situações é necessário garantir que algumas classes tenham uma e somente uma instância. Exemplo: o gerenciador de arquivos num sistema deve ser único.

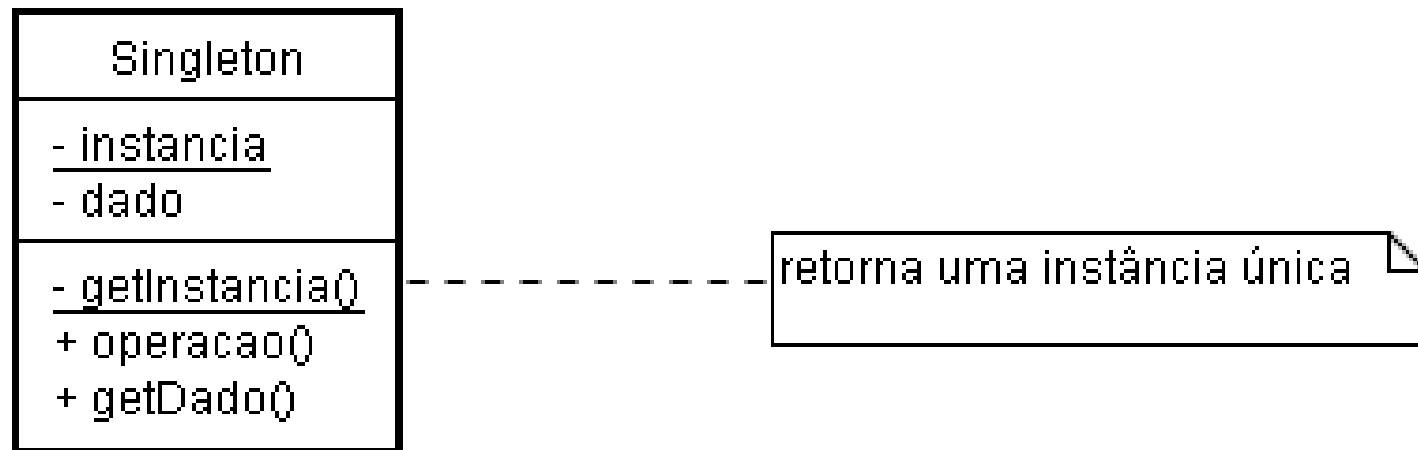
Aplicabilidade: Use esse padrão quando:

- Quando deva existir apenas uma instância de uma classe e essa instância deve dar acesso aos clientes através de um ponto bem conhecido.



Padrões de Projeto

Estrutura



Padrões de Projeto

Participante

Singleton:

- Define uma operação `getInstancia()` que permite que clientes acessem sua instância única. É um método estático e pode ser responsável pela criação de sua instância única



Padrões de Projeto

Colaborações

- Os clientes acessam uma única instância do *Singleton* pela operação `getInstancia()`

Consequências

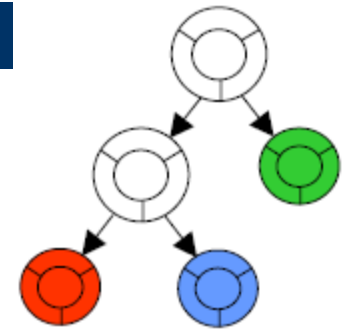
- Acesso controlado à instância única;
- Espaço de nomes reduzido;
- Permite o refinamento de operações e da representação;
- Permite um número variável de instâncias;
- Mais flexível que operações de classe (métodos estáticos não são polimórficos - não permitiriam número variável de instâncias).



Padrões de Projeto

2 – Padrões Estruturais

2.1 – Composite



- O padrão *Composite* é utilizado para representar um objeto que é constituído pela composição de objetos similares a ele. Neste padrão, o objeto composto possui um conjunto de outros objetos que estão na mesma hierarquia de classes a que ele pertence.



Padrões de Projeto

- O **padrão Composite** é normalmente utilizado para representar listas recorrentes - ou recursivas - de elementos. Além disso, esta forma de representar elementos compostos em uma hierarquia de classes permite que os elementos contidos em um objeto composto **sejam tratados como se fossem um único objeto**. Desta forma, todos os métodos comuns às classes que representam objetos atômicos da hierarquia poderão ser aplicáveis também ao conjunto de objetos agrupados no objeto composto.



Padrões de Projeto

Intenção: Compor objetos em estruturas que permitam aos clientes tratarem de maneira uniforme objetos individuais e composição de objetos.

Motivação: algumas aplicações exigem que o mesmo tratamento seja dado tanto a objetos simples como estruturas formadas por vários objetos. O padrão Composite descreve como usar a composição de forma que os clientes não precisem distinguir objetos simples de estruturas complexas.



Padrões de Projeto

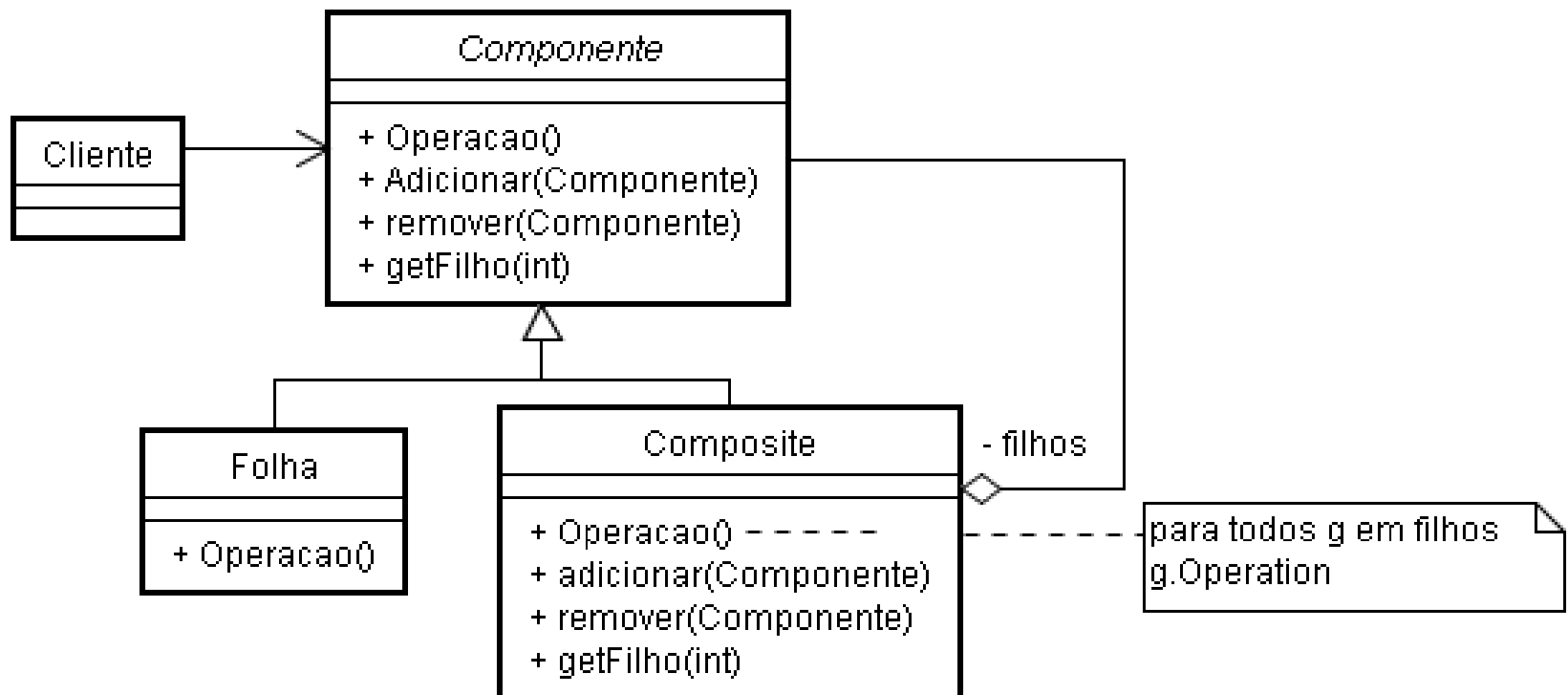
Aplicabilidade: Use esse padrão quando:

- Representação de hierarquias todo-parte de objetos.
- Clientes devem ser capazes de ignorar a diferença entre composições de objetos e objetos individuais.



Padrões de Projeto

Estrutura



Padrões de Projeto

Participantes

Componente:

- Declara a interface para objetos na composição;
- Implementa comportamento default para interface comum a todas as classes, como apropriado;
- Declara uma interface para acessar ou gerenciar seus Componentes filhos;

Folha:

- Representa objetos folhas na composição. Uma folha não tem filhos;



Padrões de Projeto

- Define comportamento para objetos primitivos na composição.

Composite:

- Define comportamento para Componentes que têm filhos;
- Armazena Componentes filhos;
- Implementa operações relacionadas com filhos na interface do Componente.

Cliente:

- Manipula objetos na composição através da interface Componente.



Padrões de Projeto

Colaborações

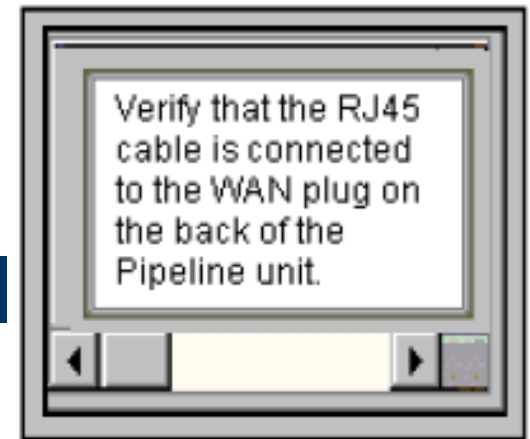
- Os clientes usam a interface da classe Componente para interagir com os objetos na estrutura composta.
- Se o receptor pertence à classe Folha então a solicitação é tratada diretamente.
- Se o receptor é um Composite, então ele normalmente repassa as solicitações para os seus componentes filhos.

Consequências


- Referências explícitas aos pais.
- Compartilhamento de componentes
- Maximização da interface de Componente



Padrões de Projeto



2.2 – Decorator

- O padrão *Decorator* é muito útil quando o assunto é variação de objetos e dinamismo de criação. Com este padrão, podemos “envolver” um determinado objeto com determinado comportamento distinto em tempo de execução. Ele **agrega dinamicamente responsabilidades** adicionais a objetos individuais, e não a toda uma classe.
- “Adiciona **responsabilidades a um objeto dinamicamente**. Decoradores provêem uma alternativa flexível à herança para estender funcionalidade” é a descrição dada pelo autor a esse padrão. 

Padrões de Projeto

- **Intenção :** Agregar dinamicamente responsabilidades adicionais a um objeto.
- **Motivação:** Existem situações nas quais se deseja acrescentar responsabilidades a objetos individuais e não a toda uma classe. O uso de herança nesses casos é inflexível porque as novas propriedades precisam estar definidas em tempo de compilação.



Padrões de Projeto

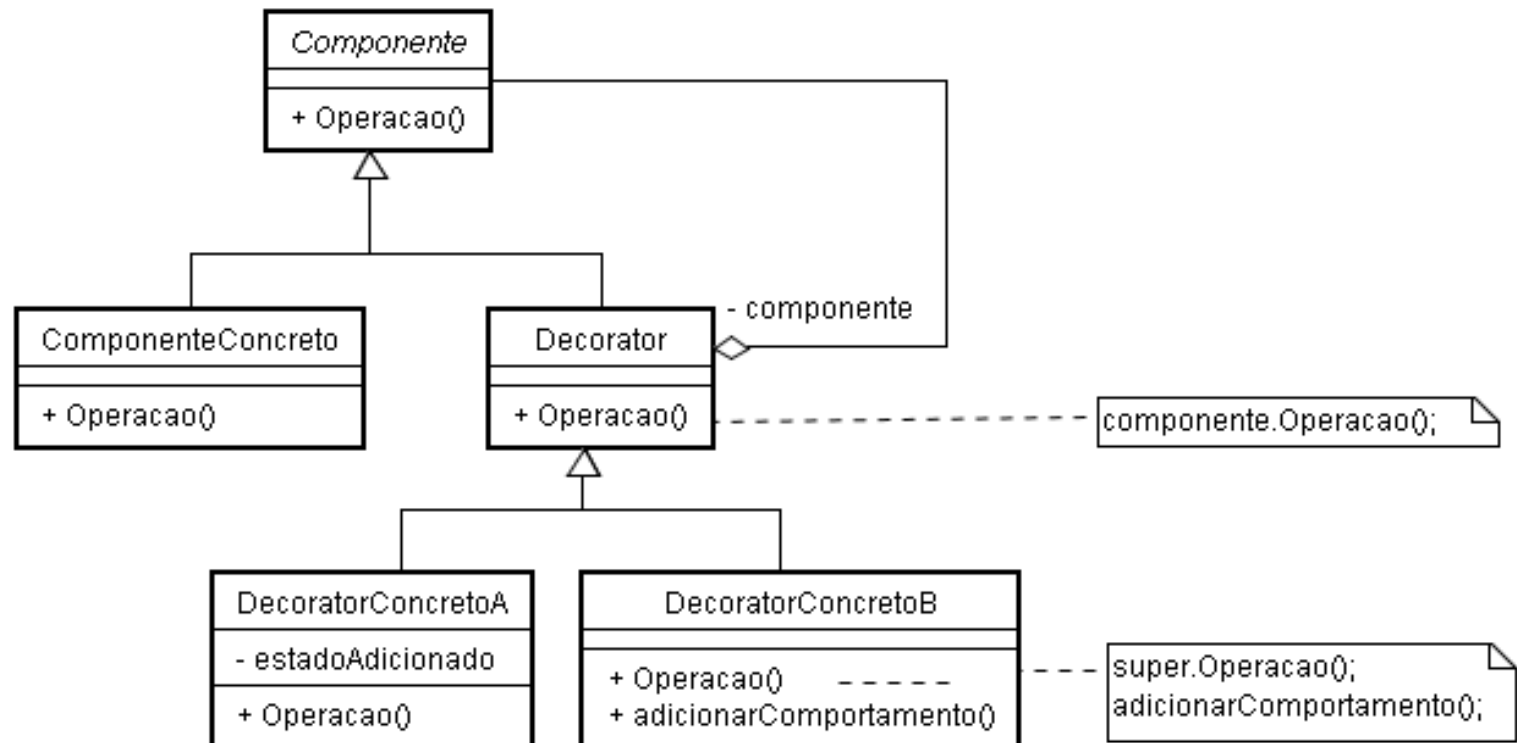
Aplicabilidade: Use esse padrão quando:

- Deseja-se acrescentar responsabilidades a objetos de forma dinâmica e transparente (ou seja, sem afetar outros objetos);
- Responsabilidades podem ser removidas ou alteradas;
- Quando a extensão através de herança não é prática ou não é possível.



Padrões de Projeto

Estrutura



Padrões de Projeto

Participantes

Componente:

- Define a interface para objetos que podem ter responsabilidades acrescentadas a eles dinamicamente.

ComponenteConcreto:

- Define um objeto para o qual responsabilidades adicionais podem ser atribuídas.

Decorator:

- Mantém uma referência para um objeto Componente. Define uma interface que segue a interface de Componente.



Padrões de Projeto

DecoratorConcreto:

- Acrescenta responsabilidades ao componente.



Padrões de Projeto

Colaborações

- *Decorator* repassa as solicitações para o seu objeto Componente. Opcionalmente pode executar operações adicionais antes e depois de repassar a solicitação.

Consequências

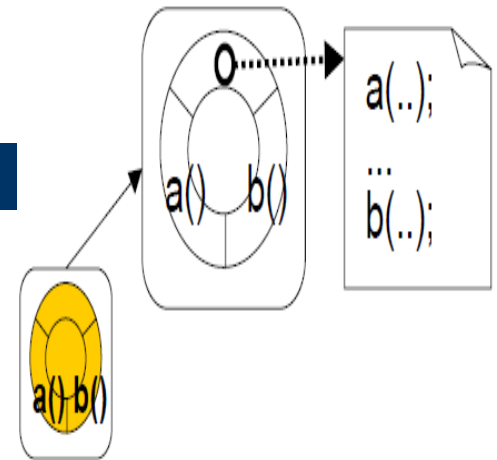
- Maior flexibilidade que herança estática;
- Evita classes sobrecarregadas de métodos e atributos na raiz da hierarquia;
- Um *Decorator* e seu Componente não são idênticos;
- Grande quantidade de pequenos objetos.



Padrões de Projeto

3 – Padrões Comportamentais

3.1 – Template Method



- O *Template Method* define um esqueleto de algum algoritmo em um método, postergando a implementação dos passos deste algoritmo para as subclasses. Os métodos que são criados nesta classe são chamados de operações Primitivas.
- Este padrão permite que subclasses redefinam certos passos de um algoritmo sem mudar a estrutura do mesmo.



Padrões de Projeto

Intenção: Definir o esqueleto de um algoritmo, delegando determinados passos para as subclasses. Subclasses redefinem passos do algoritmo, sem alterar a estrutura geral do mesmo.

Motivação: Famílias de aplicações baseadas num framework podem necessitar de algoritmos genéricos para determinadas funções, sendo que os detalhes de execução do mesmo dependem da aplicação específica.



Padrões de Projeto

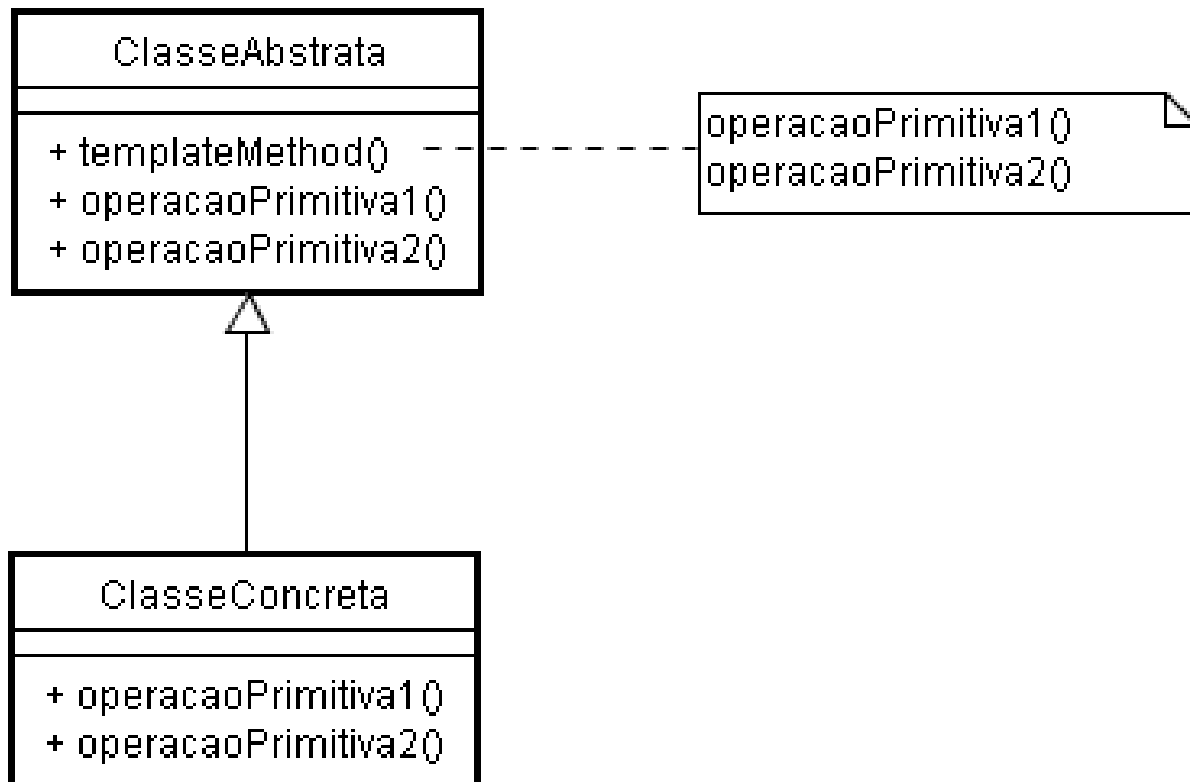
Aplicabilidade: Use esse padrão quando:

- Para implementar as partes invariantes de um algoritmo uma só vez e deixar as subclasses a implementação do comportamento que pode variar.
- Quando comportamento comum entre subclasses deveria ser **fatorado e localizado** numa classe comum para evitar duplicação
- Para controlar extensões de subclasses. Pode ser definido um método template que chama operações gancho em pontos específicos, permitindo extensões somente nestes pontos.



Padrões de Projeto

Estrutura



Padrões de Projeto

Participantes

Classe Abstrata

- Define operações abstratas que subclasses concretas definem para implementar certas etapas do algoritmo.
- Implementa um Template Method definindo o esqueleto de um algoritmo. O **Template Method** chama várias operações, entre as quais as operações abstratas da classe.

Classe Concreta

- Implementa as operações abstratas para desempenhar as etapas do algoritmo que tenham comportamento específico a esta subclasse.



Padrões de Projeto

Colaborações

- ClasseConcreta depende da ClasseAbstrata para implementar os passos invariantes do algoritmo.

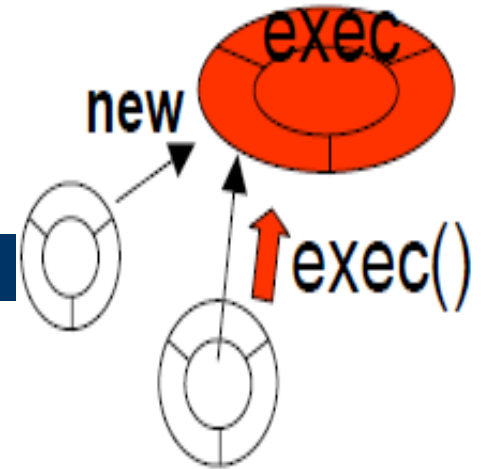
Consequências

- Técnica fundamental para reuso de código.
- “Princípio de Hollywood”: não nos chame, nós chamaremos você.



Padrões de Projeto

3.2 – Command



- A principal função do **padrão Command** é encapsular invocações a objetos. Quando invocar objetos torna-se uma tarefa complexa há a necessidade de se encapsular essas invocações para se simplificar o código. Com isso são abstraídas, para o cliente, coisas do tipo: onde, quando ou qual componente vai realizar uma determinada tarefa.



Padrões de Projeto

- Esse padrão guarda um objeto comando para cada objeto a ser invocado. Para **invocar o objeto destino**, invoca-se sempre o mesmo **método do “objeto comando”** e esse método se encarrega de chamar os métodos adequados para realizar as ações requeridas.
- O **Command** gera uma maior flexibilidade na arquitetura do sistema, uma vez que permite a adição de novas funcionalidades (comandos) sem a necessidade de mudar o design da arquitetura.



Padrões de Projeto

Intenção: Encapsular uma solicitação como um objeto, permitindo parametrizar clientes com diferentes solicitações, enfileirar ou fazer registro (log) de solicitações e suportar operações que podem ser desfeitas (undo).

Motivação: Existem situações nas quais é necessário emitir solicitações para objetos sem que se conheça nada a respeito da operação ou do receptor da mesma.



Padrões de Projeto

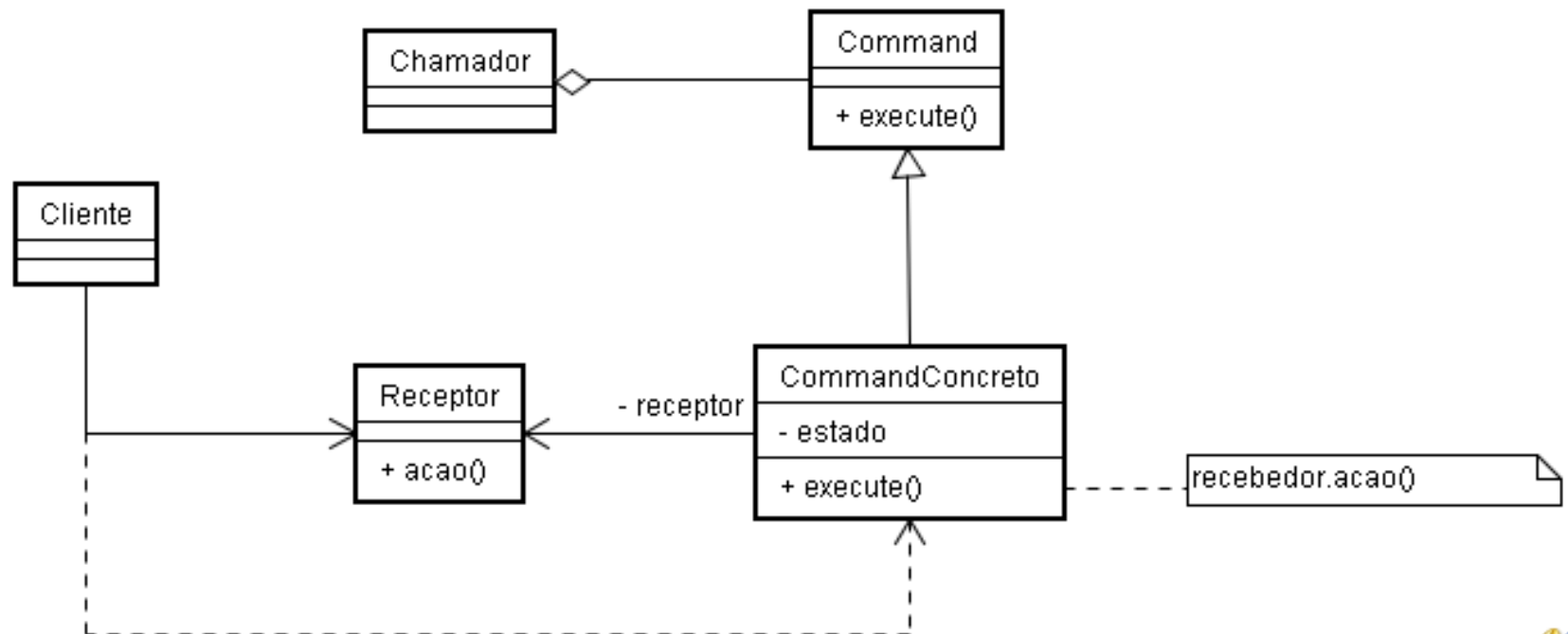
Aplicabilidade: Use esse padrão quando:

- Parametrizar as ações a serem executadas pelos objetos (ao estilo “callback” em linguagem procedurais).
- Especificar, enfileirar e executar solicitações em tempos diferentes.
- Registrar e eventualmente desfazer operações realizadas
- Estruturar um sistema com base em operações de alto nível **construídas sobre** operações básicas.



Padrões de Projeto

Estrutura



Padrões de Projeto

Participantes

Command

- Declara uma interface para a execução de uma operação.

CommandConcreto

- Define uma vinculação entre um objeto Receptor e uma ação.
- Implementa execute() através da invocação da(s) correspondente(s) operação(ções) no Receptor.



Padrões de Projeto

Cliente

- Cria um objeto CommandConcreto e estabelece o seu receptor.

Chamador

- Solicita ao Command a execução da solicitação.

Receptor

- Sabe como executar as operações associadas a uma solicitação.
- Qualquer classe pode funcionar como um Receptor.



Padrões de Projeto

Colaborações

- O cliente cria um objeto `CommandConcreto` e especifica o seu receptor
- Um objeto `Chamador` armazena o objeto `CommandConcreto`
- O `Chamador` emite uma solicitação chamando `execute` no `Command`. Caso os comandos precisem ser desfeitos, `CommandConcreto` armazena estados para desfazer o comando antes de invocar o método `execute()`.
- O objeto `CommandConcreto` invoca operações no seu `Receptor` para executar a solicitação.



Padrões de Projeto

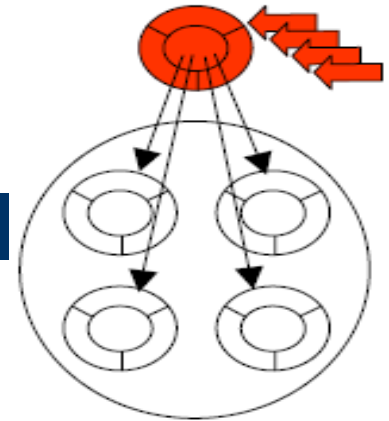
Consequências

- Command desacopla o objeto que invoca a operação daquele que sabe como executá-la.
- Commands são objetos que podem ser manipulados e estendidos como qualquer outro objeto.
- É possível juntar **comandos formando um “comando composto”** (podendo-se usar o padrão *Composite*).
- É fácil acrescentar novos Commands porque não é necessário mudar classes existentes.



Padrões de Projeto

3.3 – Iterator



- O padrão *Iterator* é utilizado para prover acesso sequencial a elementos de uma agregação sem expor a implementação que **está “por baixo”**. A idéia deste padrão é garantir **toda a responsabilidade de varredura** sobre um objeto que contém uma lista de elementos a um objeto *iterator*. Este deve prover uma interface de acesso a todos os objetos, incluindo métodos que facilitem o controle sequencial da varredura.



Padrões de Projeto

- O padrão *Iterator* como já descrito “Provê uma forma de acessar os elementos de uma coleção de objetos sequencialmente sem expor sua representação subjacente”.
- Esse padrão também é conhecido como **cursor**, pois mantém qualquer informação de estado necessária para saber até onde a iteração já foi para acessar os elementos de uma lista, ou seja, mantém o cursor (índice) do item da lista.



Padrões de Projeto

Intenção: Fornecer um meio de acessar sequencialmente os elementos de um objeto agregado, sem expor sua representação subjacente.

Motivação: Um agregado de objetos, assim como uma lista deve fornecer um meio de acessar seus elementos sem necessariamente expor sua estrutura interna. – Pode ser necessário **percorrer um agregado de objetos** de mais de uma maneira diferente. – Eventualmente é necessário manter mais de um percurso pendente em um dado agregado de objetos.



Padrões de Projeto

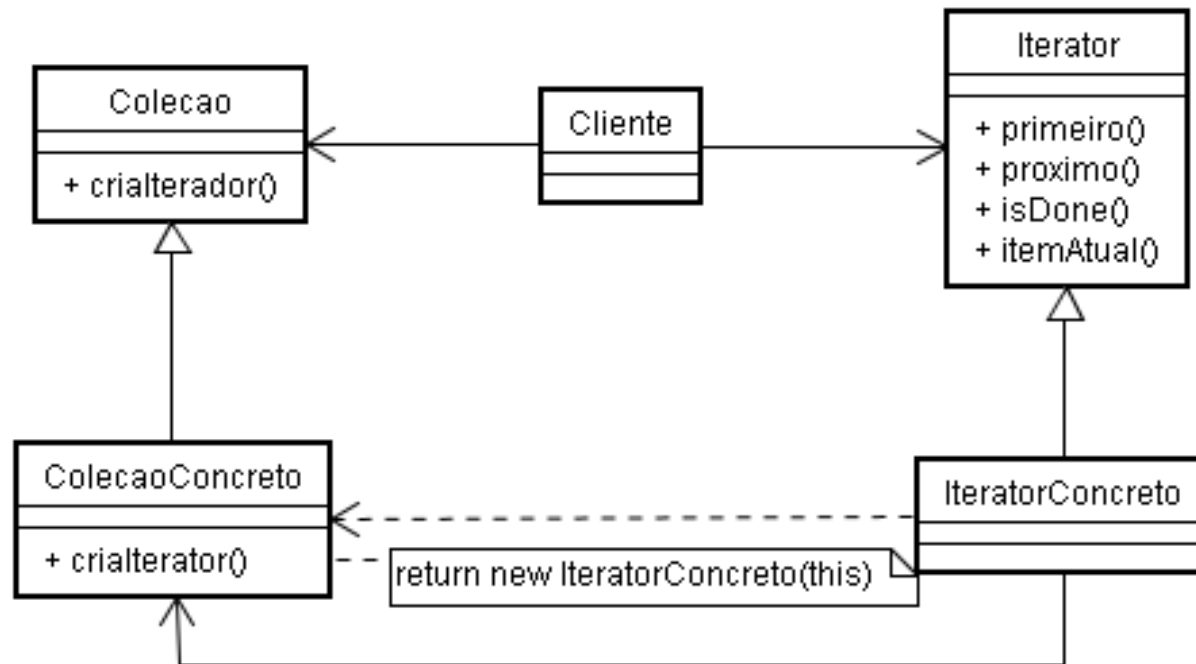
Aplicabilidade: Use esse padrão quando:

- Para acessar os conteúdos de um agregado de objetos sem expor a sua representação interna;
- Para suportar múltiplos percursos de agregados de objetos;
- Para fornecer uma interface uniforme que percorra diferentes **estruturas agregadas** (suportando iteração polimórfica).



Padrões de Projeto

Estrutura



Padrões de Projeto

Participantes

Iterador:

- Define a interface para acessar e percorrer os elementos

IteradorConcreto :

- Implementa a interface Iterator.
- Mantém o controle da posição corrente no percurso da coleção.



Padrões de Projeto

Coleção

- Define uma interface para a criação de um objeto Iterator.

ColecaoConcreto

- Implementa a interface de criação do Iterator para retornar uma instância do IteratorConcreto.



Padrões de Projeto

Colaborações

- Um objeto `IteratorConcreto` mantém o controle do objeto corrente no agregado de objeto e consegue definir o seu sucessor durante o percurso.

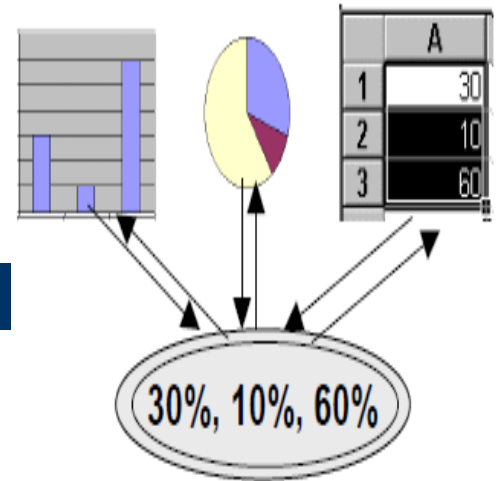
Consequências

- Suporta variações no percurso de um agregado.
- Iteradores simplificam a interface do agregado.
- Mais de um percurso pode estar em curso num mesmo agregado.



Padrões de Projeto

3.4 – Observer



- O cerne desse padrão está na possibilidade de uma classe poder “avisar” um conjunto de classes associadas a ela que o seu estado foi **alterado por algum motivo**. Como já visto a definição do padrão *observer* é a seguinte:
- "Definir uma dependência **um-para-muitos** entre objetos para que quando um objeto mudar de estado, todos os seus dependentes sejam notificados e atualizados automaticamente."



Padrões de Projeto

- Esse aviso pode ser feito de outras maneiras, por exemplo, utilizando eventos **através de chamadas a métodos de cada instância**. Porém dessa maneira você estará criando um “monstro”, ou seja, um sistema com alto acoplamento e de difícil manutenção.
- Utilizando o padrão *Observer* conseguimos reduzir o uso do **relacionamento bidirecional** por meio de interfaces e classes abstratas, separando a abstração para ter um alto nível de coesão e baixo acoplamento.



Padrões de Projeto

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

Motivação: Ao se particionar um sistema em uma coleção de classes cooperantes, muitas vezes é necessário manter a consistência entre objetos relacionados. Isso deve ser feito preservando-se o acoplamento fraco para não comprometer a reusabilidade.



Padrões de Projeto

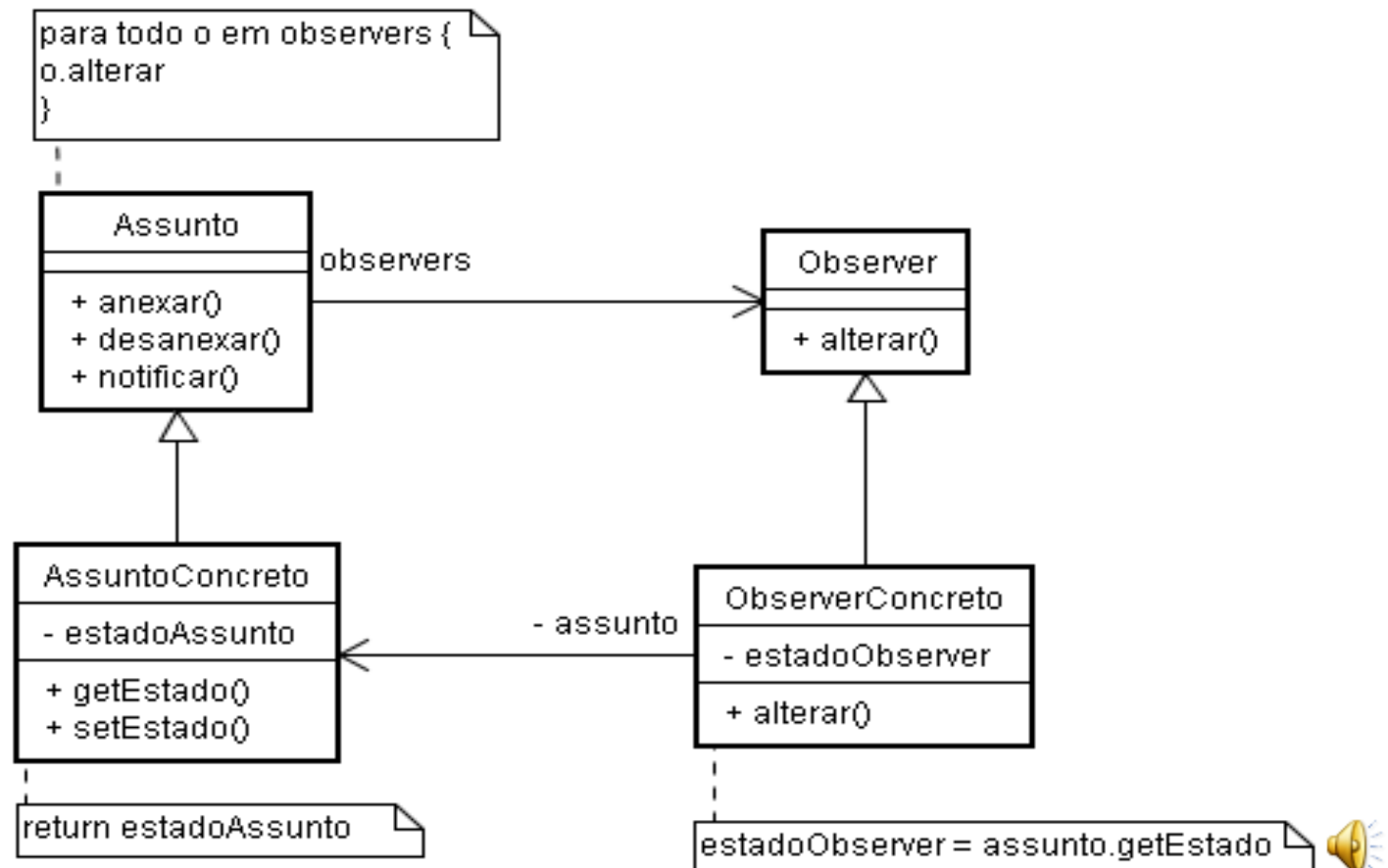
Aplicabilidade: Use esse padrão quando:

- Uma abstração pode ser “apresentada” de várias formas.
- A mudança de estado de um objeto implica em mudanças em outros objetos.
- Um objeto deve ser capaz de notificar a outros objetos, sem necessariamente saber quem são esses objetos.



Padrões de Projeto

Estrutura



Padrões de Projeto

Participantes

Assunto

- Conhece seus objetos Observer. Quaisquer números de objetos Observer podem observar um Assunto.
- Provê uma interface para acoplar e desacoplar objetos Observer.

AssuntoConcreto

- Guarda o estado de interesse para ObserverConcreto
- Envia uma notificação para seu Observer quando seu estado muda.



Padrões de Projeto

Observer

- Define uma interface de atualização para objetos que devem ser notificados sobre mudanças em um Assunto.

ObserverConcreto

- Mantém uma referência para um objeto AssuntoConcreto
- Guarda o estado que deve ficar consistente com o de Assunto
- Implementa o Observer atualizando a interface para manter seu estado consistente com o de Assunto.



Padrões de Projeto

Colaborações

- O AssuntoConcreto notifica seus observadores sempre que ocorrer uma mudança que poderia tornar inconsistente o estado deles com o seu próprio;
- Ao ser informado de uma mudança pelo AssuntoConcreto um objeto Observer pode consultá-lo para obter as informações necessárias para atualizar o seu estado.



Padrões de Projeto

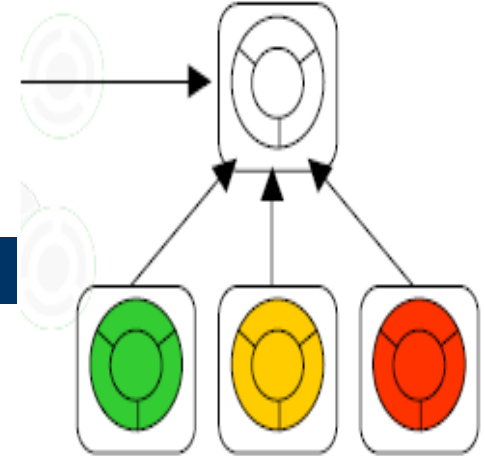
Consequências

- Acoplamento abstrato entre Assunto e Observer.
- Suporte a broadcast.
- Atualizações inesperadas.



Padrões de Projeto

3.5 – Strategy



- O padrão *Strategy* define uma família de algoritmos intercambiáveis e encapsula cada um deles fazendo com que eles possam ser permutáveis. O **padrão *Strategy* permite que os algoritmos variem independentemente** dos clientes que os utilizam. Este padrão visa à reutilização, organização e flexibilidade de seus algoritmos.



Padrões de Projeto

- Ele é usado quando muitas classes relacionadas diferem apenas em **alguns de seus comportamentos**. Provê uma maneira de configurar uma classe com um entre vários comportamentos possíveis.



Padrões de Projeto

Intenção: Definir uma família de algoritmos, encapsular cada uma delas e torná-las intercambiáveis. O **padrão Strategy** permite que o algoritmo varie independentemente dos clientes que os utilizam.

Motivação: Em uma mesma aplicação, para tratar certos tipos de problemas, diferentes algoritmos são apropriados em diferentes situações. Novos algoritmos podem ser criados e incorporados a aplicação sem que esta tenha que sofrer alterações estruturais.



Padrões de Projeto

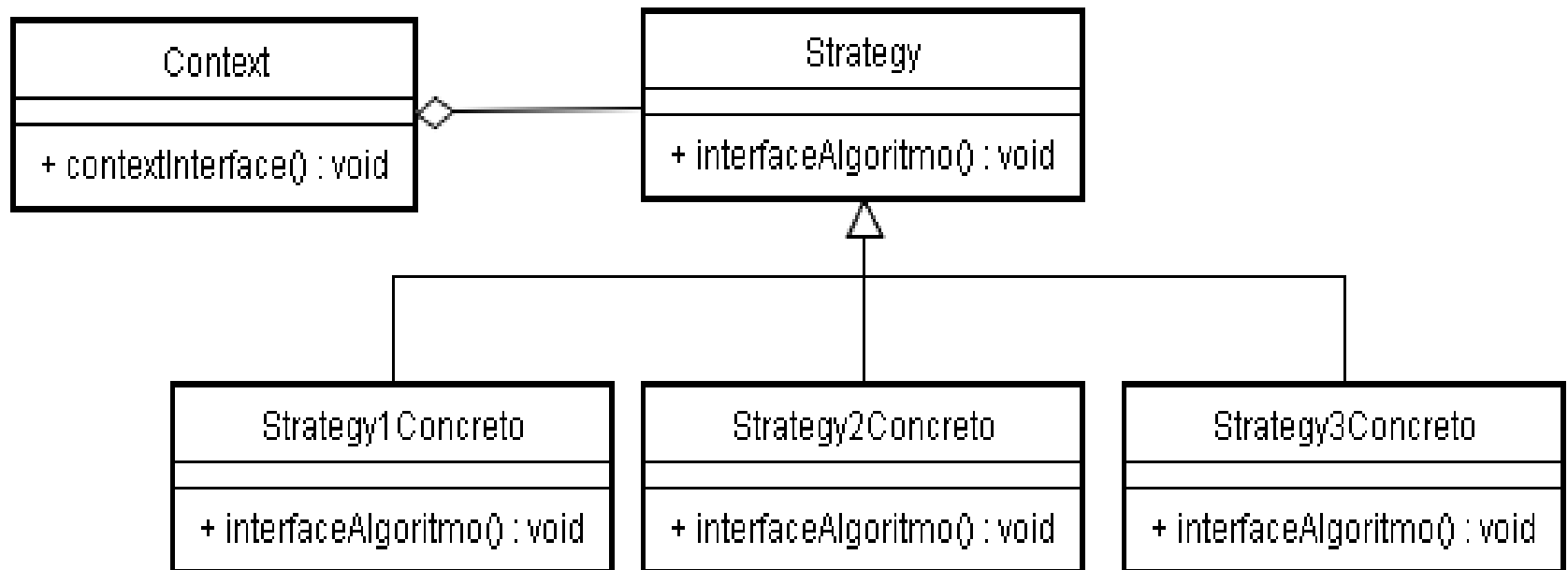
Aplicabilidade: Use esse padrão quando:

- Muitas classes relacionadas diferem apenas em seus comportamentos;
- É necessária diferente variação de um algoritmo;
- Um algoritmo utiliza dados dos quais o cliente não pode ter conhecimento.



Padrões de Projeto

Estrutura



Padrões de Projeto

Participantes

Strategy

- Define uma interface comum para todos os algoritmos suportados.

StrategyConcreto

- Implementa o algoritmo usando a interface de Strategy.

Context

- É configurado com um objeto StrategyConcreto.
- Mantém uma referência para um objeto StrategyConcreto.
- Pode definir uma interface que permite a Strategy acessar seus dados.



Padrões de Projeto

Colaborações

- Strategy e Context interagem para implementar o algoritmo escolhido.
- Context repassa solicitações dos seus clientes para o Strategy. Os clientes em geral passam um objeto StrategyConcreto para o Context.

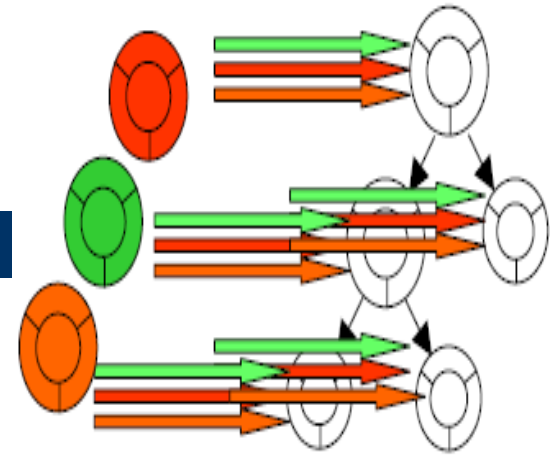
Consequências

- Famílias de algoritmos relacionados.
- Alternativa para o uso de subclasses.
- Eliminam comandos condicionais ao se codificar os métodos.



Padrões de Projeto

3.6 – Visitor



- O padrão *Visitor* como já definido anteriormente representa uma **operação a ser realizada nos elementos** de uma estrutura de objetos permitindo que se defina uma nova operação sem alterar as classes dos elementos nos quais a operação age.
- Esse padrão é uma solução para separar o algoritmo da estrutura. Uma de **suas vantagens é a habilidade** de adicionar novas operações a uma estrutura já existente.



Padrões de Projeto

Intenção: Representar uma operação a ser implementada nos elementos de um agregado de objetos. O padrão *Visitor* permite implementar uma nova operação sem mudar as classes dos elementos que constituem os agregados.

Motivação: Em algumas situações é necessário percorrer um agregado de objetos realizando operações sobre seus elementos e **dependendo do caso**, o conjunto de operações independe das classes dos objetos que constituem o agregado.



Padrões de Projeto

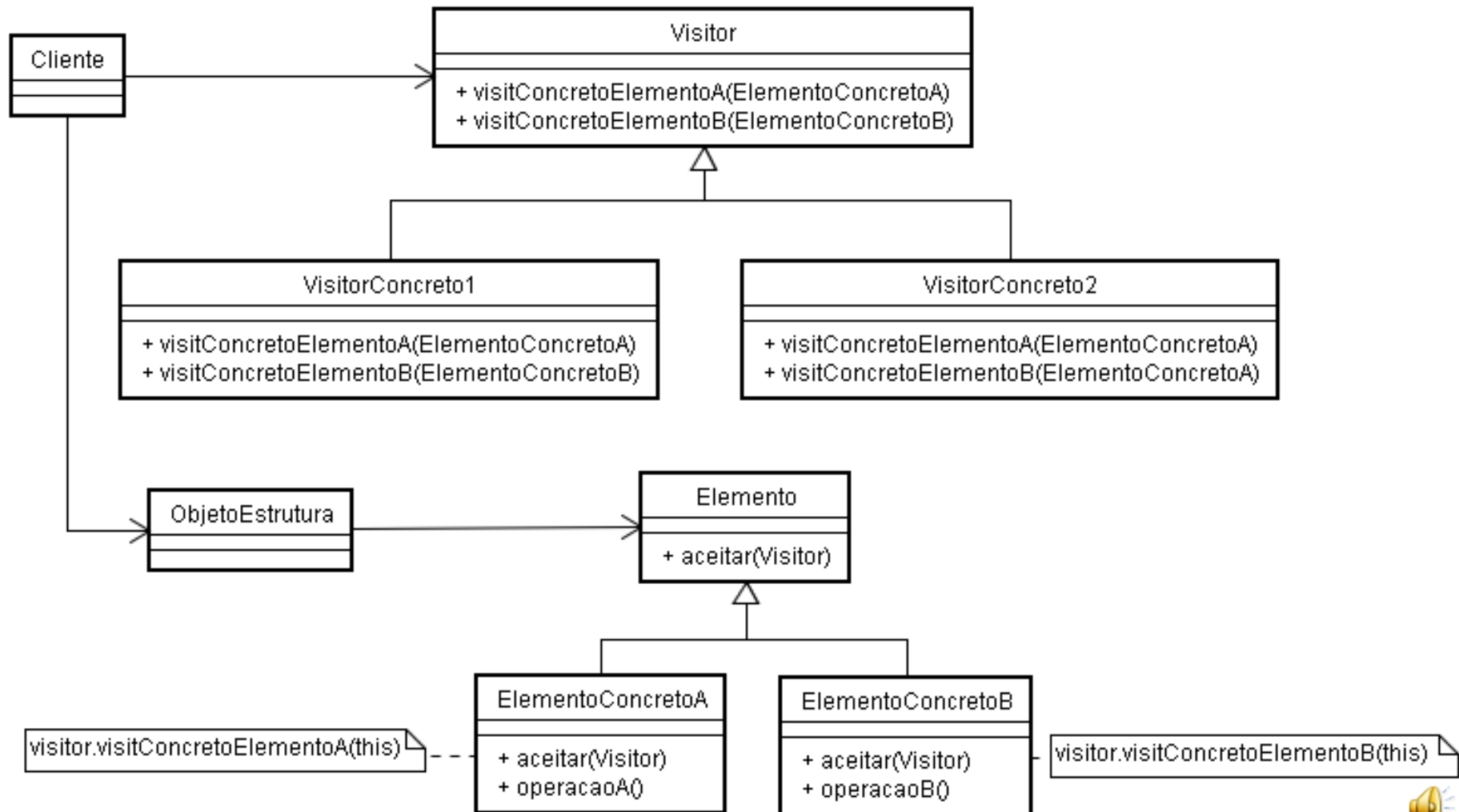
Aplicabilidade: Use esse padrão quando:

- Quando se têm muitas classes de objetos com interfaces distintas e quer-se realizar operações nesses objetos que dependam das suas classes concretas;
- Quando a estrutura de objetos é utilizada por diversas aplicações;
- Quando se quer evitar a poluição da classe com operações não relacionadas e que são utilizadas em vários objetos;
- Quando a estrutura que define os objetos é praticamente estática e as **operações realizadas neles estão** em constante mudança



Padrões de Projeto

- Estrutura



Padrões de Projeto

Participantes

Visitor

- Declara uma operação de visita para cada classe a ser visitada (ElementoConcreto);
- Identifica a classe ElementoConcreto através do nome e assinatura da operação visit.

VisitorConcreto

- Implementa cada operação de visita declarada pelo Visitor;
- Cada operação implementará um algoritmo que receberá o seu contexto e estado de atuação .



Padrões de Projeto

Elemento

- Define a operação aceitar que recebe um Visitor como argumento

ElementoConcreto

- Implementa a operação aceitar que recebe um Visitor como argumento

ObjectoEstructura

- Pode enumerar diversos Elementos;
- Pode prover uma interface que permita o Visitor visitar os seus Elementos associados;
- Pode ser um composite ou uma coleção (lista ou conjunto). 🔊

Padrões de Projeto

Colaborações

- Um cliente que usa o padrão *Visitor* deve criar um objeto *ConcreteVisitor* e percorrer a estrutura do objeto, visitando cada elemento através desse *Visitor*.

Consequências

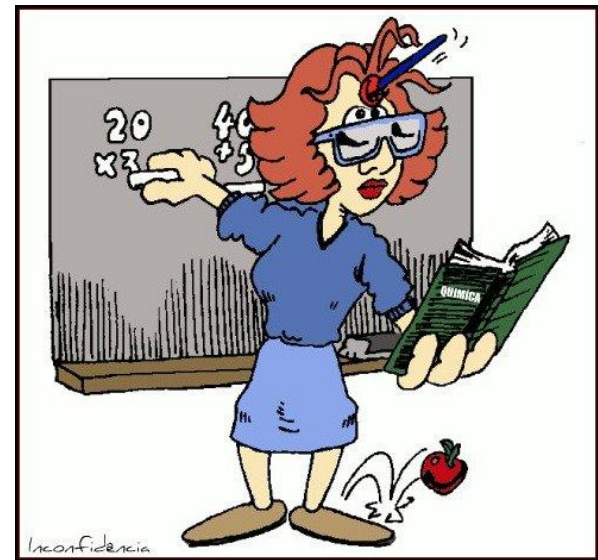
- Torna fácil a adição de novas operações.
- Um visitante reúne operações relacionadas e separa as operações não relacionadas.
- É difícil acrescentar novas classes *ConcreteElement*.
- Compromete o encapsulamento.



Padrões de Projeto



**Não durma
no ponto.**



**Exercício de
fixação.**

