Programação Orientada a Objetos
Padrões de Projeto
(*design patterns*)

Fernando Vanini
IC - UNICAMP

Padrões de Projeto (*design patterns*)

- Apresentação do conceito de design pattern
- Classificação dos design patterns
- Catálogo
- Uso dos design patterns

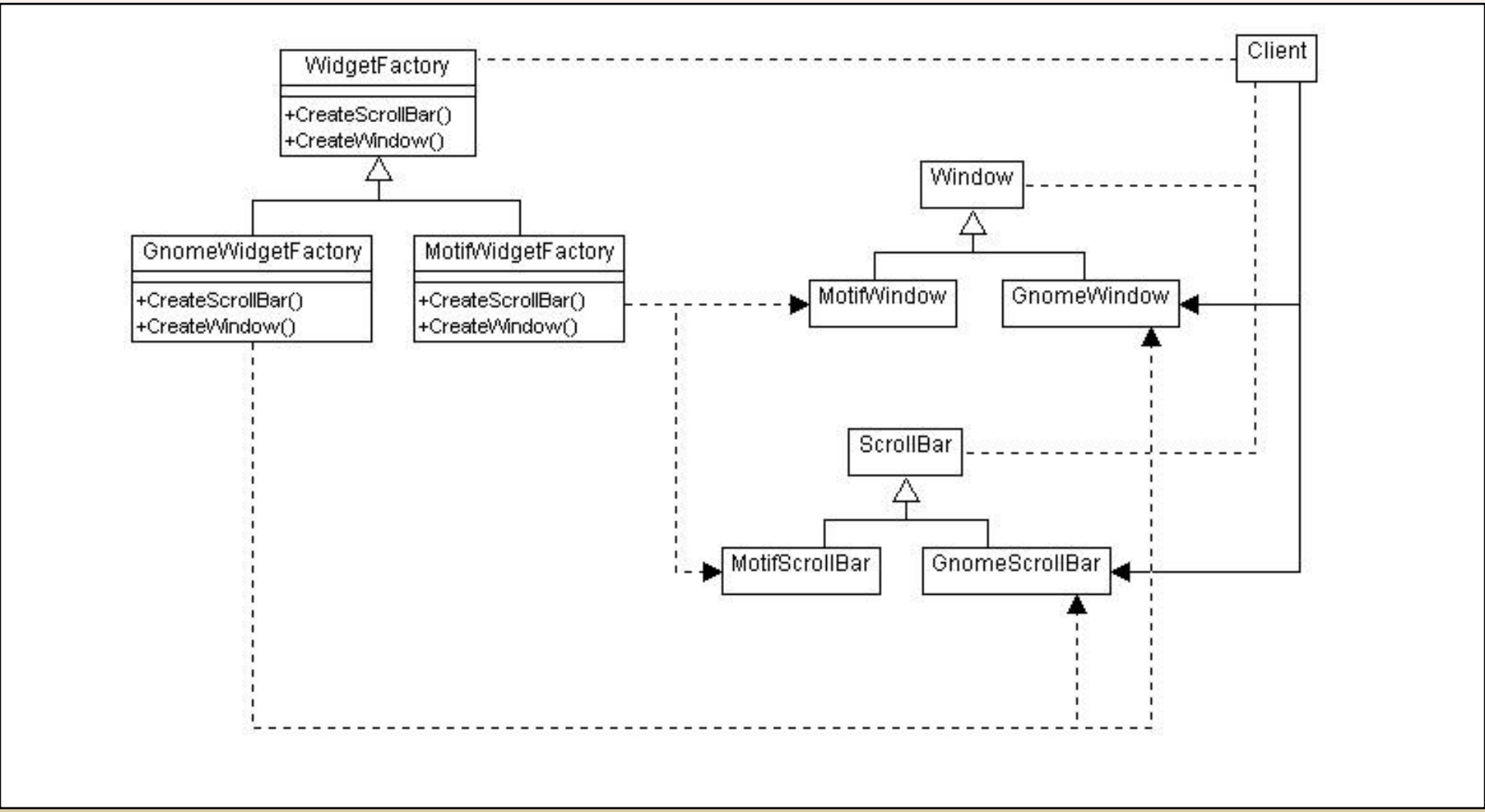
Design Patterns

- À medida que se acumula experiência em projetos usando objetos, observa-se que determinadas situações de colaboração entre objetos se repetem, independentemente da tecnologia ou linguagem de programação utilizada.
- Alguns autores (Gamma et. al.) catalogaram um conjunto de soluções de projeto (em ingles *design patterns*) que consideraram representativas e desde então, essas soluções tem sido uma referência importante para empresas e programadores em geral.

Um exemplo

- Uma aplicação com interface gráfica precisa ser implementada de forma portável para diversas plataformas gráficas, como por exemplo Motif e Gnome.
- O padrão *Abstract Factory* é aplicável a esse tipo de situação

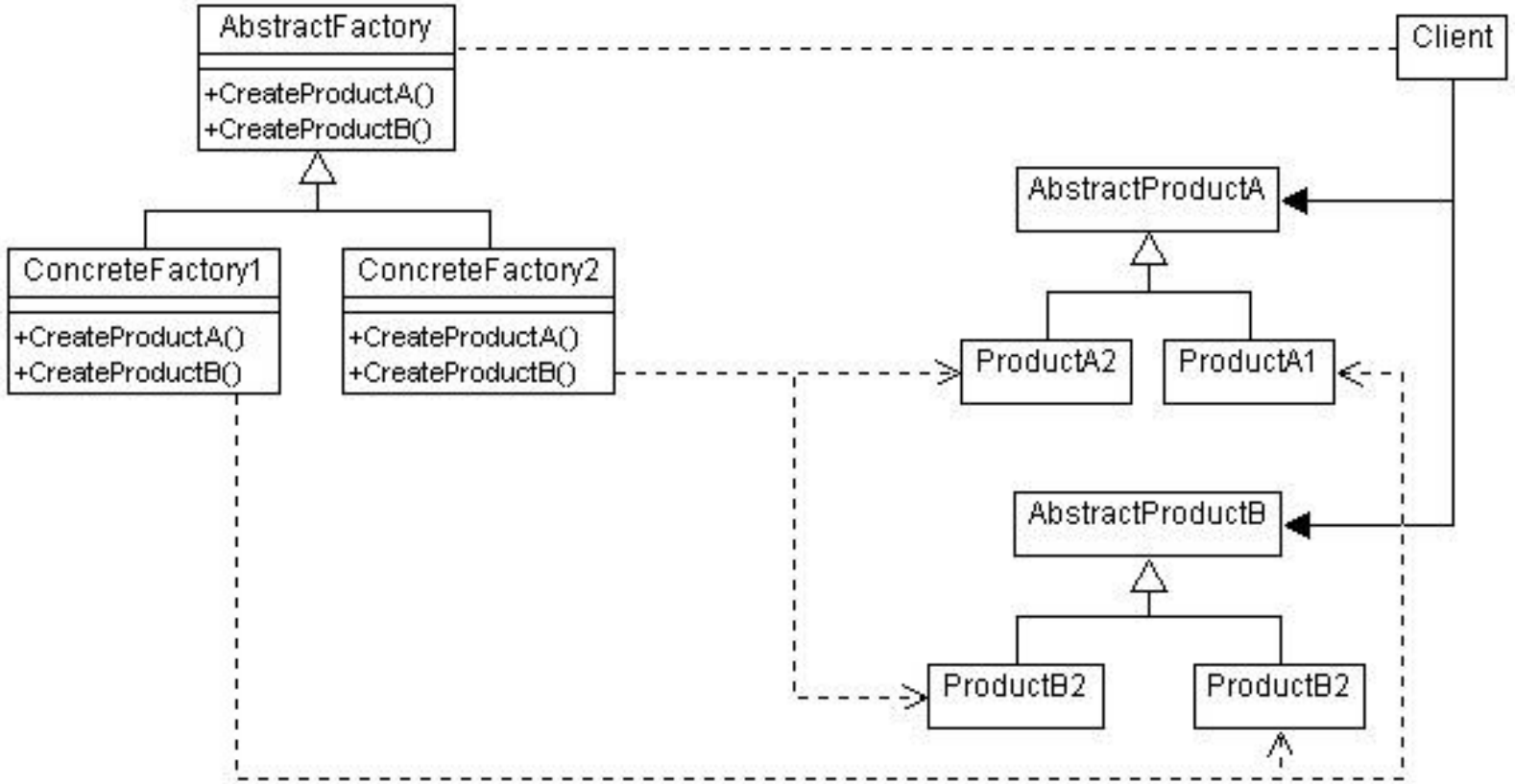
Um exemplo – Abstract Factory



Abstract Factory

- No padrão *Abstract Factory*, ou Fábrica Abstrata, a aplicação cliente interage com uma 'fábrica genérica de objetos‘
- os objetos serão gerados efetivamente pela fábrica concreta que estiver sendo utilizada no momento
- a aplicação cliente não precisa ser configurada para interagir com cada uma das fábricas concretas
- novas fábricas concretas podem ser agregadas, alteradas ou retiradas do sistema sem necessidade de alterações na aplicação cliente

Abstract Factory



Design Patterns: reuso da solução

- Ao se comparar o exemplo com a descrição genérica apresentada, nota-se que a cada nova situação, um novo código deverá ser escrito, apesar da estrutura ser basicamente a mesma.
- A principal idéia por trás dos *design patterns* é o *reuso da solução* e não necessariamente o reuso do código, como acontece no caso de bibliotecas.

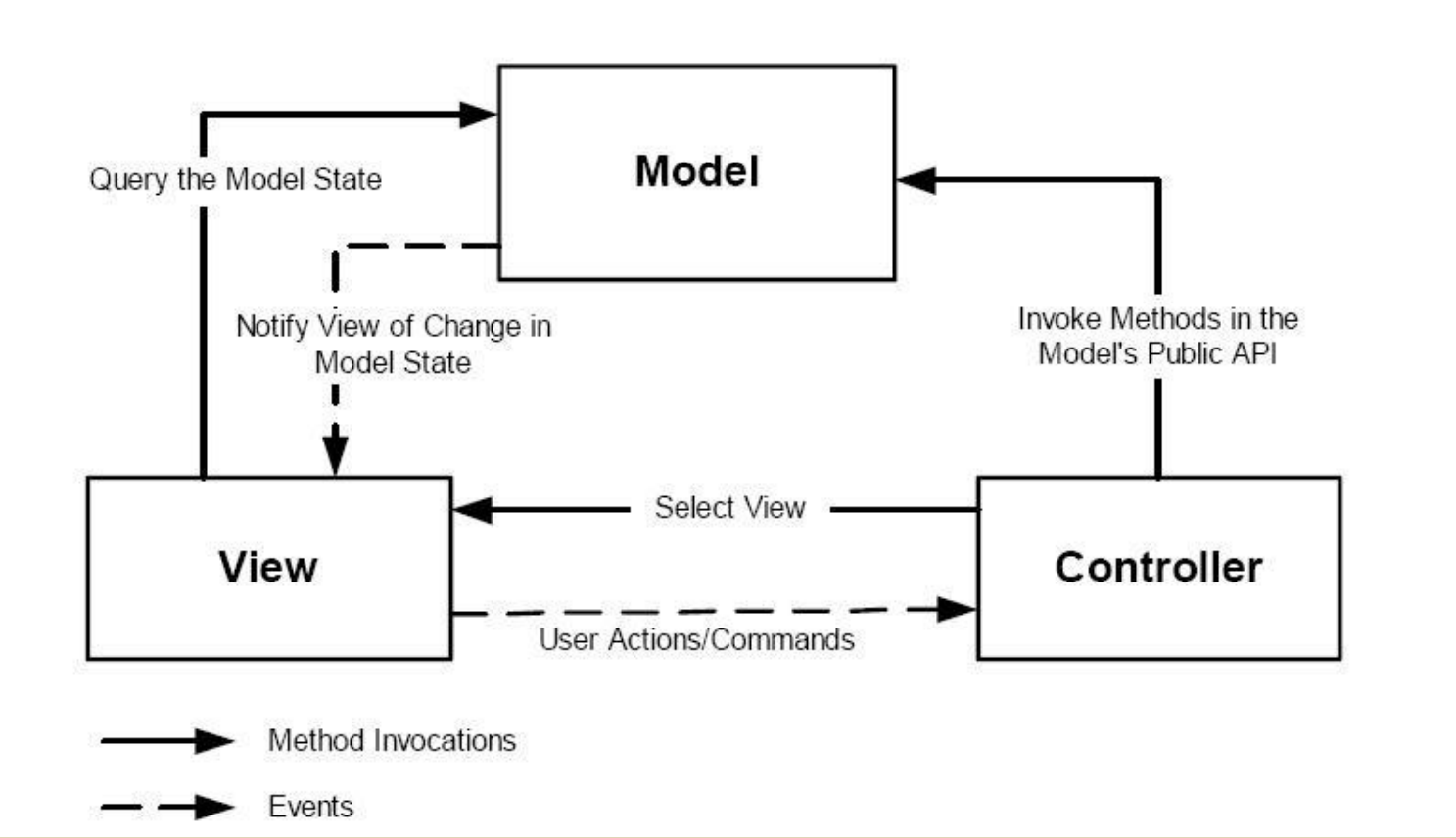
Design Patterns: a origem

- A idéia de ‘padrões de projeto’ como forma de reusar soluções recorrentes surgiu de um abordagem semelhante na área de arquitetura (Cristopher Alexander).
- Um dos primeiros padrões identificados na área de software foi o modelo MVC (*model, view, controller*), que foi usado no projeto da interface de usuário da linguagem Smalltalk (Xerox).

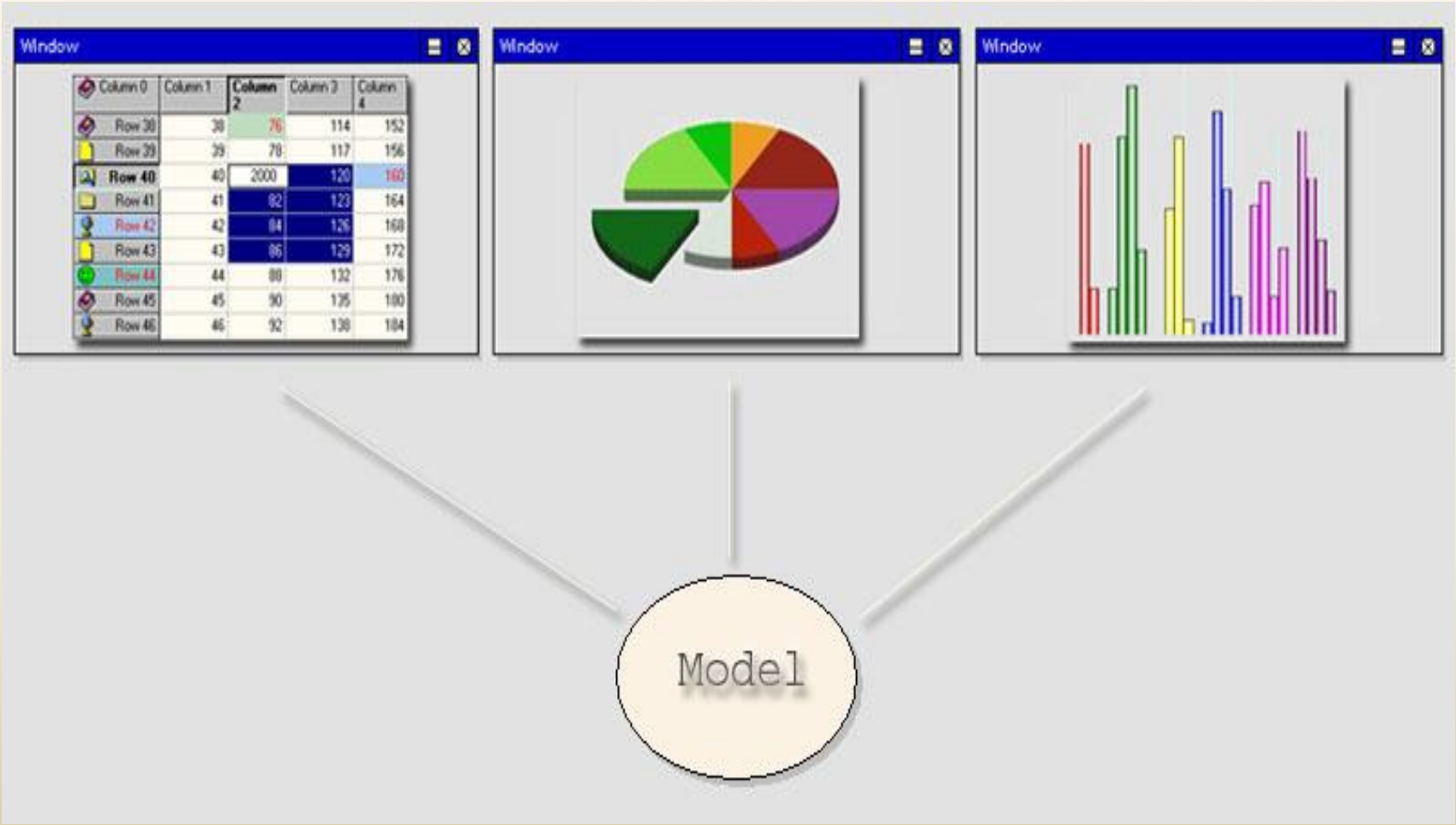
O modelo MVC

- No modelo MVC de arquitetura, o software é organizado em três camadas:
  - *model* – responsável pelo armazenamento e manutenção dos dados utilizados pela aplicação.
  - *view* – é a camada responsável pela interface com o usuário
  - *controller* – é a camada responsável pelo tratamento de eventos e implementação das regras de negócio (que normalmente implicam em mudanças nos dados através dos serviços de *model*)

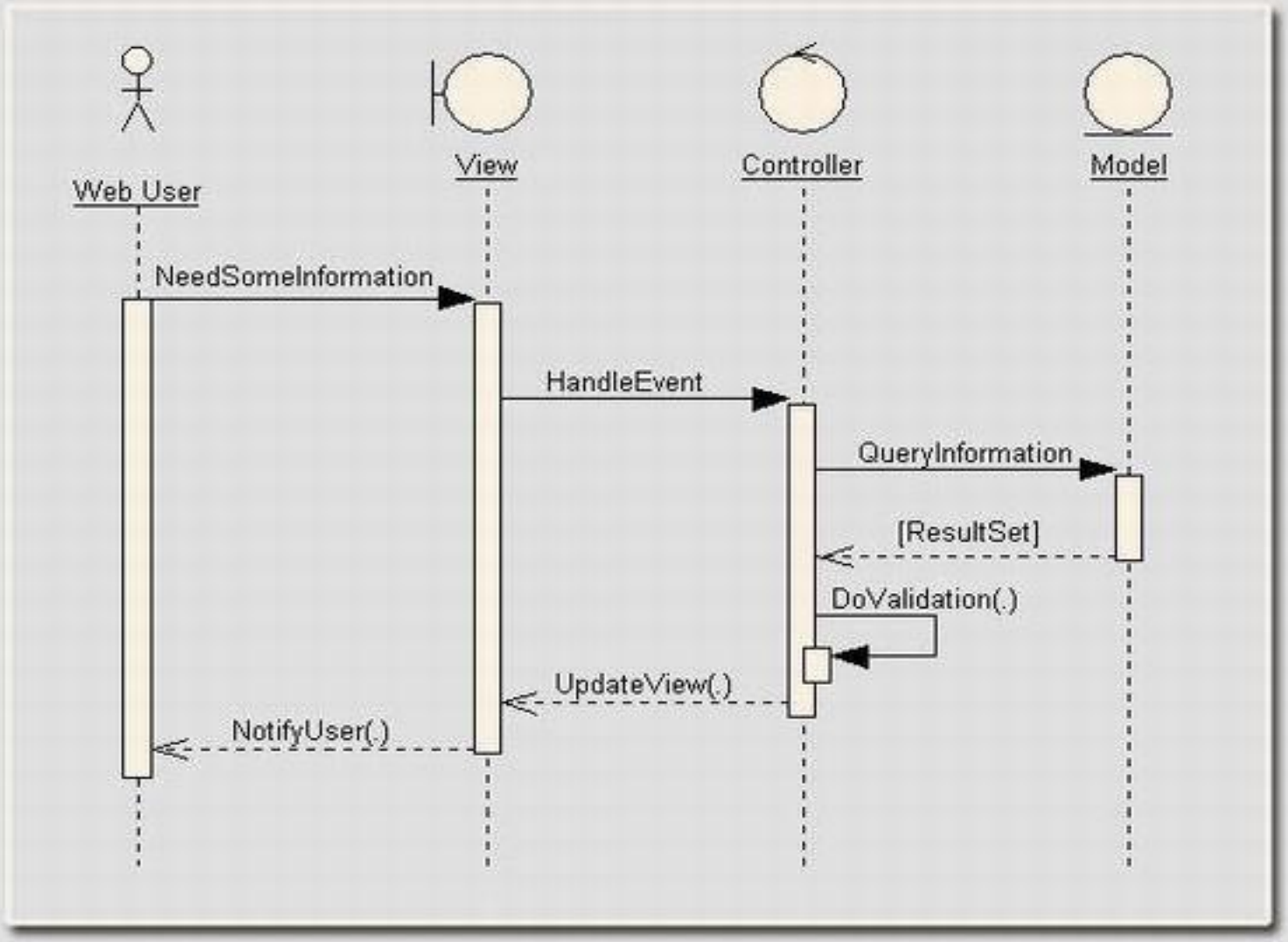
O modelo MVC


Model
Query the Model State
Notify View of Change in Model State
Invoke Methods in the Model's Public API
View
Select View
Controller
User Actions/Commands
Method Invocations
Events

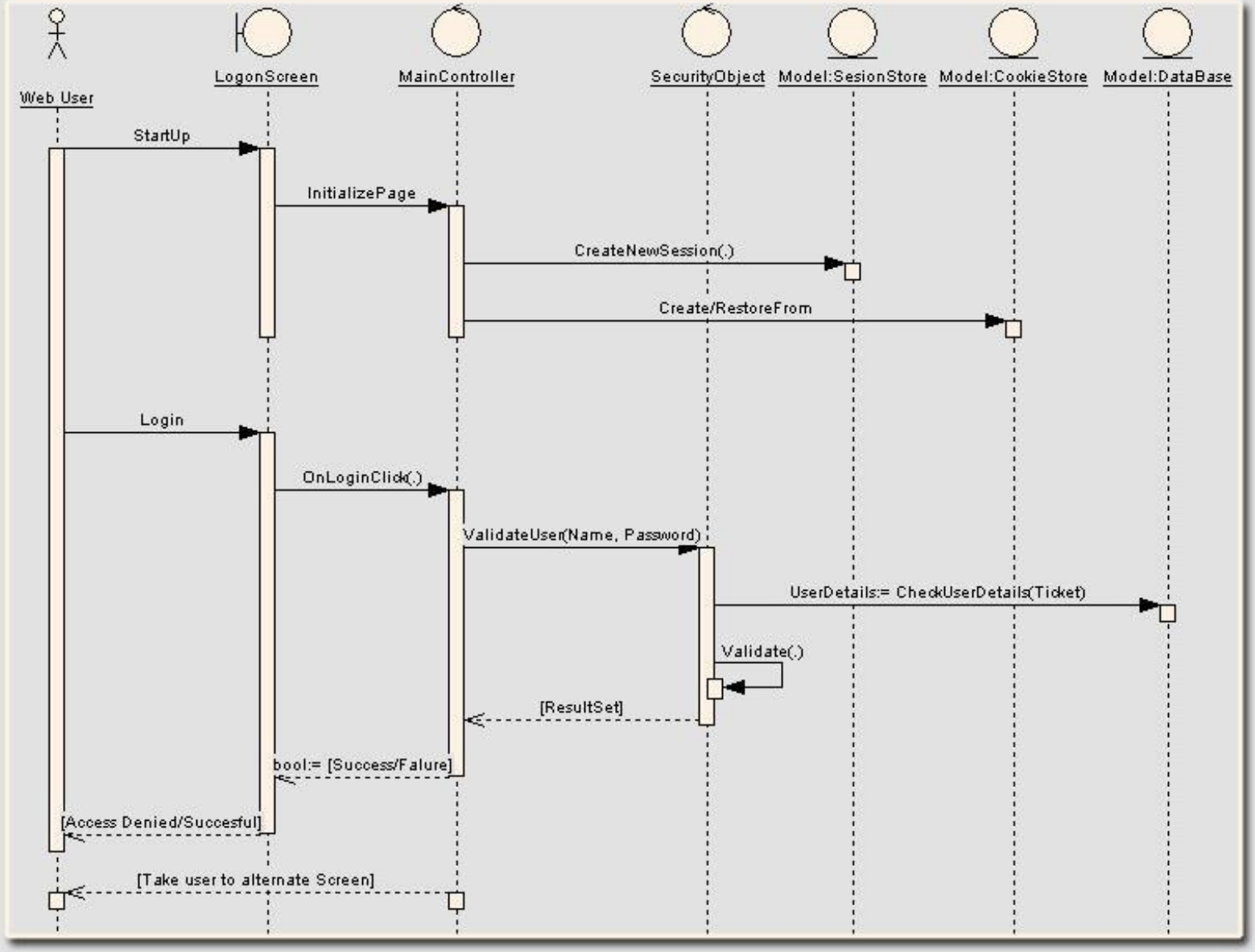
MVC – motivação inicial



MVC numa aplicação web



MVC numa aplicação web



MVC - conseqüências

- Vantagens
  - O mesmo modelo pode ser usado com diferentes visões (ou aplicações diferentes).
  - Novos tipos de clientes podem ser agregados à aplicação, sem nenhum impacto para o modelo.
  - Clareza e modularidade de projeto.
- Desvantagens
  - Complexidade adicional, só justificável em aplicações de médio e grande porte.

Design Patterns: classificação

- para facilitar o estudo e uso dos *design patterns*, os autores os classificaram segundo dois critérios: *finalidade* e *escopo*
- A *finalidade* diz respeito ao que a solução faz ou se propõe a fazer.
- O *escopo* de um padrão especifica se o padrão se aplica primariamente a classes ou a objetos.

Design Patterns: finalidade

- De acordo com a sua finalidade, os padrões são classificados em
  - Padrões de Criação(*creational*) - se preocupam com o processo de criação de objetos.
  - Estruturais (*structural*) - lidam com a composição de classes ou de objetos.
  - Comportamentais (*behavioral*) - tratam da forma pela qual classes ou objetos interagem e distribuem responsabilidades

Design Patterns: escopo

- De acordo com a seu escopo, os padrões são classificados em
  - *padrões para classes (class patterns)*: lidam com os relacionamentos entre classes e suas subclasses, estabelecidos através de herança, sendo portanto estáticos (fixados em tempo de compilação).
  - *padrões para objetos (object patterns)* tratam de relacionamentos entre objetos que podem se alterar em tempo de execução, portanto dinâmicos.
  - No 'catálogo oficial' alguns padrões são apresentados em duas versões, uma aplicável a classes e outra aplicável a objetos.

O Catálogo

O catálogo dos design patterns é apresentado no seguinte formato:

- nome e classificação do padrão
- Intenção e objetivo
- Também conhecido como
- Motivação
- Aplicabilidade
- Estrutura
- Participantes
- Colaborações
- Conseqüências
- Implementação
- Exemplos de Código
- Usos conhecidos
- Padrões Relacionados

Padrões de Criação

- Abstraem o processo de instanciação dos objetos, contribuindo para tornar o sistema independente de como seus objetos são criados, compostos e representados.
- Um padrão de criação de classe usa herança para variar a classe sendo instanciada.
- Um padrão de criação de objeto delega a instanciação a outro objeto.

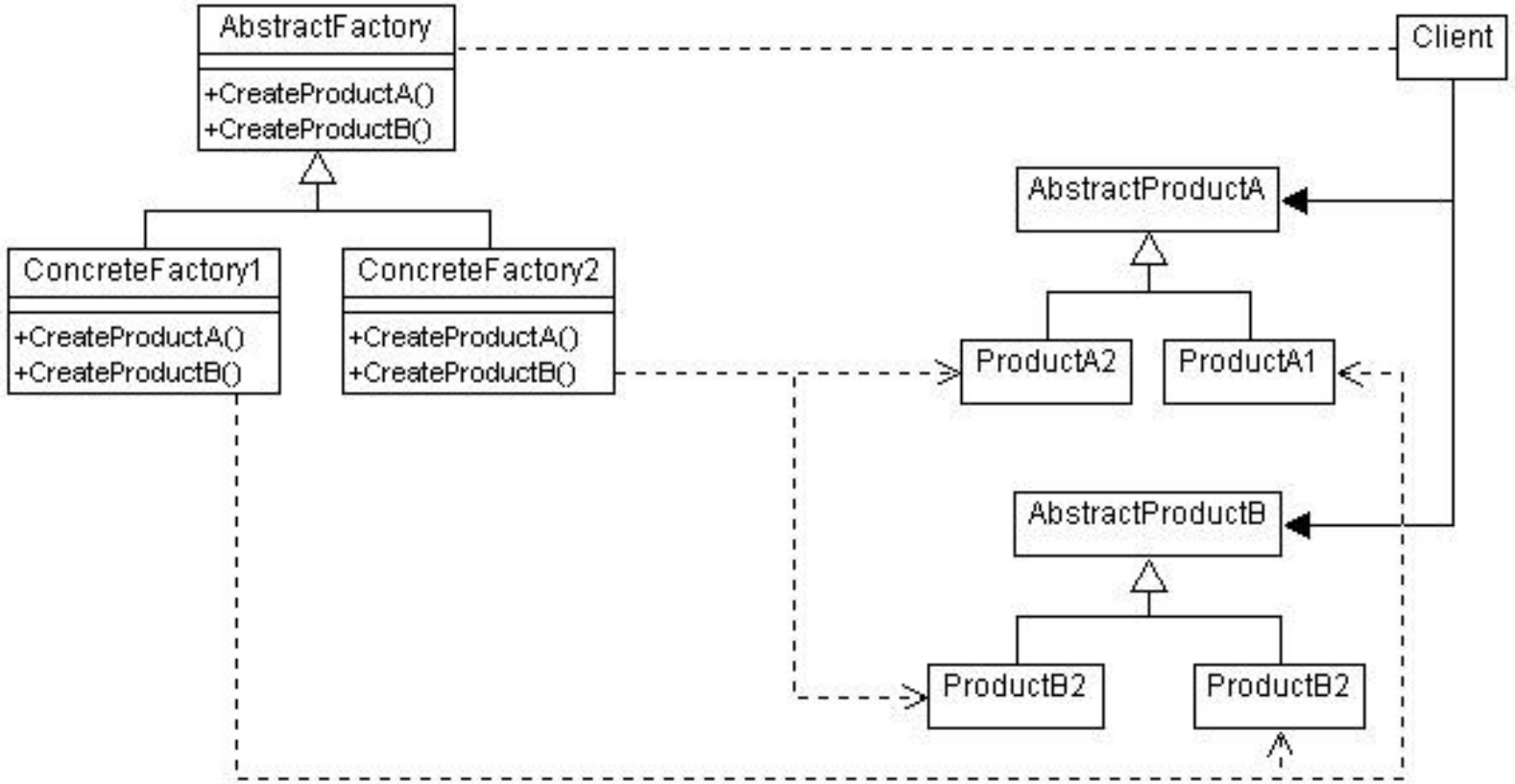
Abstract Factory

- **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.

Abstract Factory

- Aplicabilidade: aplicável a situações nas quais
  - 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

Abstract Factory - estrutura



Abstract Factory

- Colaborações
  - Em tempo de execução, normalmente é criada uma única instância da classe ConcreteFactory. Ela será a responsável pela criação dos produtos concretos.
  - AbstractFactory delega a criação dos objetos-produto para suas subclasses concretas (ConcreteFactory).
- Conseqüências
  - Isola as classes concretas
  - Facilita a troca de famílias de produtos
  - É difícil suportar novos tipos de produtos
- Padrões Correlatos
  - *FactoryMethod, Prototype*

Builder

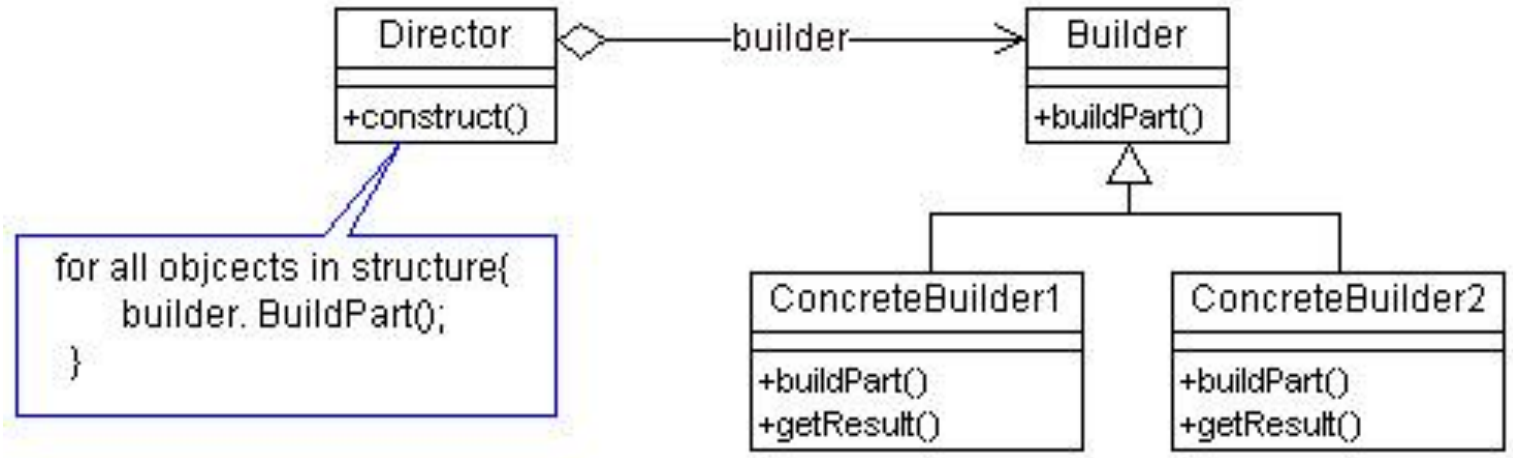
- Intenção: Separar a construção de um objeto complexo de sua representação de modo que o mesmo processo de construção possa criar diferentes representações.

- Motivação: Um exemplo pode ser um *leitor* de documento RTF capaz de converter o texto lido para vários formatos diferentes, ficando aberto o número de conversões possíveis. Deve ser portanto fácil acrescentar uma nova conversão sem modificar o *leitor*.

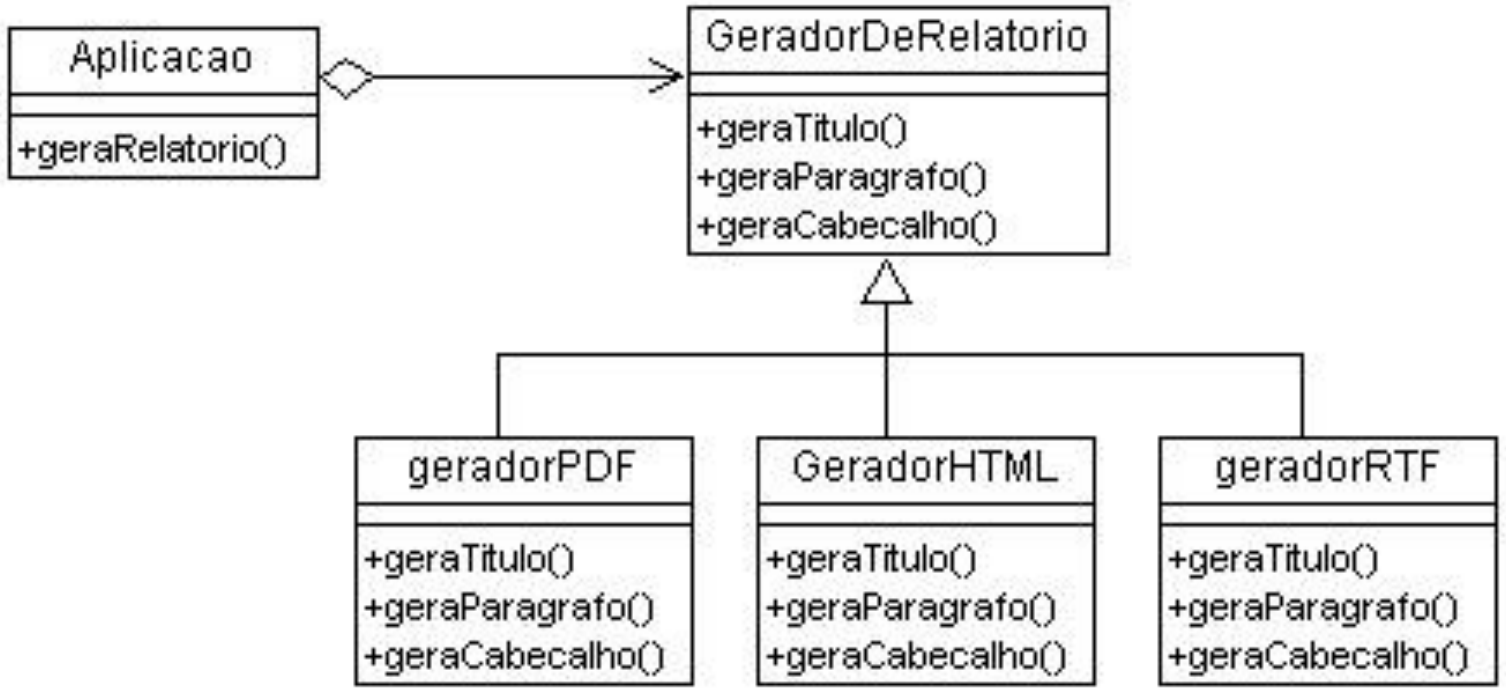
Builder

- Aplicabilidade: aplicável a situações onde
  - O algoritmo para a criação de um objeto complexo deve ser independente das partes que compõem o objeto e de como elas devem ser montadas.
  - O processo de construção deve permitir diferentes representações para o objeto que é construído.

Builder - Estrutura



Builder – Um exemplo



Builder

- Colaborações
  - O cliente cria o objeto Director e o configura com o Builder desejado.
  - O cliente notifica o Builder sempre que uma parte do produto deve ser construída
  - Builder trata solicitações e acrescenta partes ao produto
  - O Cliente recupera o produto do construtor

Builder

- Conseqüências
  - Permite variar a representação interna de um produto
  - Isola o código para a construção e representação
  - Oferece um controle mais fino sobre o processo de construção
- Padrões Correlatos
  - Abstract Factory
  - Composite

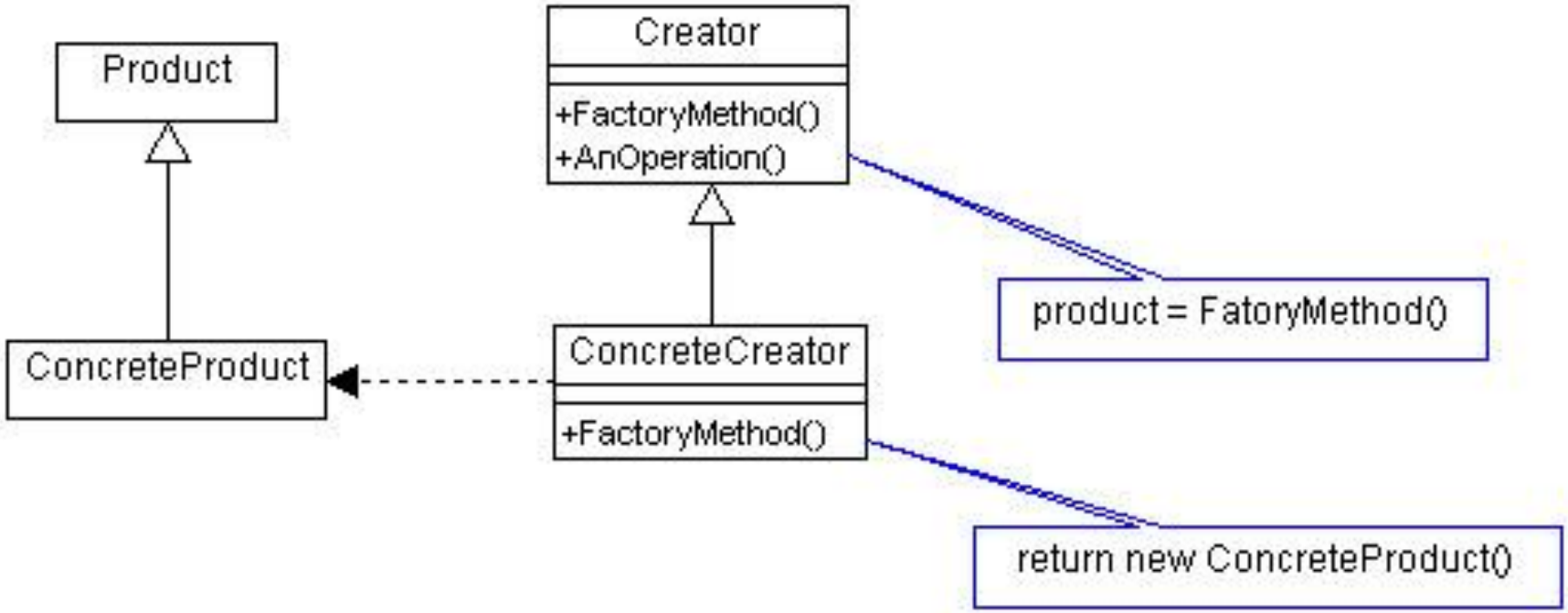
Factory Method

- Intenção: Definir uma interface para a criação de um objeto, deixando as subclasses decidirem que classe instanciar. O FactoryMethod 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 FactoryMethod é usado nesses casos e decide com base do 'contexto', qual das subclasses ativar. Um exemplo simples: leitura de objetos serializados num arquivo.

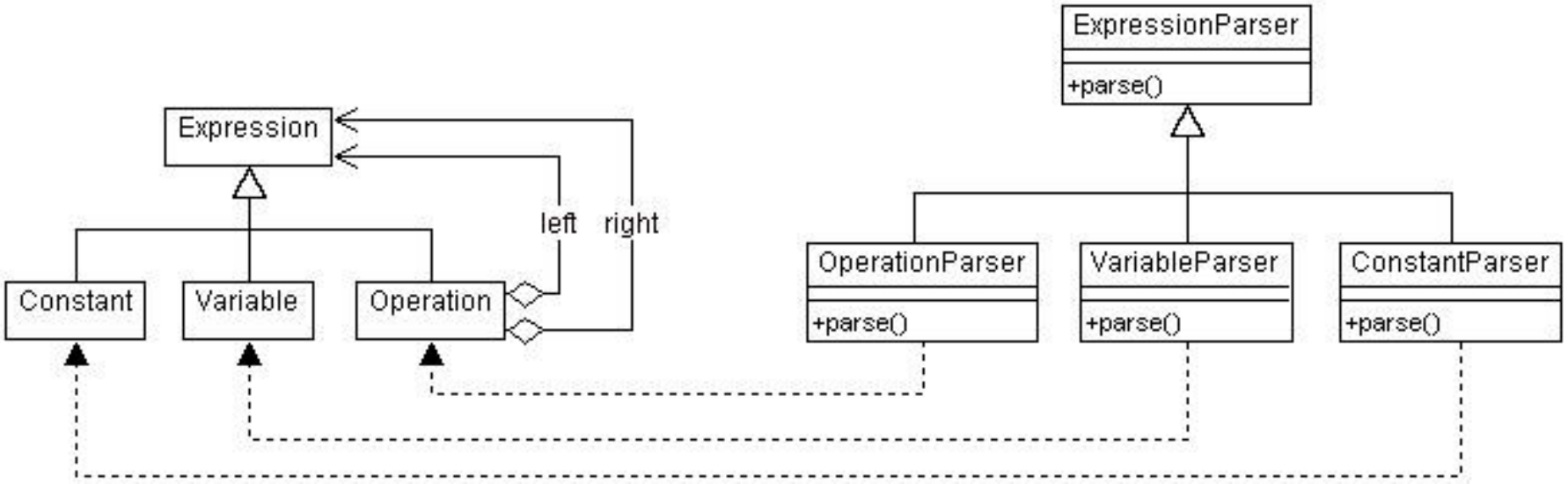
Factory Method

- Aplicabilidade: em casos tais que
  - O cliente não consegue antecipar a classe de objetos que deve criar.
  - Uma classe quer que suas subclasses especifiquem os objetos que criam
- Colaborações
  - Creator depende das suas subclasses para definir o método de construção necessário à retornar a instância do produto concreto apropriado.
- Conseqüências
  - Fornece ‘ganchos’ para as subclasses.
  - Conecta hierarquias de classe paralelas.
- Padrões Correlatos
  - Abstract Factory, Template Method

Factory Method - Estrutura



Factory Method – Um exemplo



Factory Method

- Colaborações
  - Creator depende das suas subclasses para definir o ‘FactoryMethod’ apropriado.
- Conseqüências
  - Fornece ‘ganchos’ para a subclasses
  - Conecta hierarquia de classes paralelas
- Padrões Correlatos
  - Abstract Factory
  - Composite
  - TemplateMethod

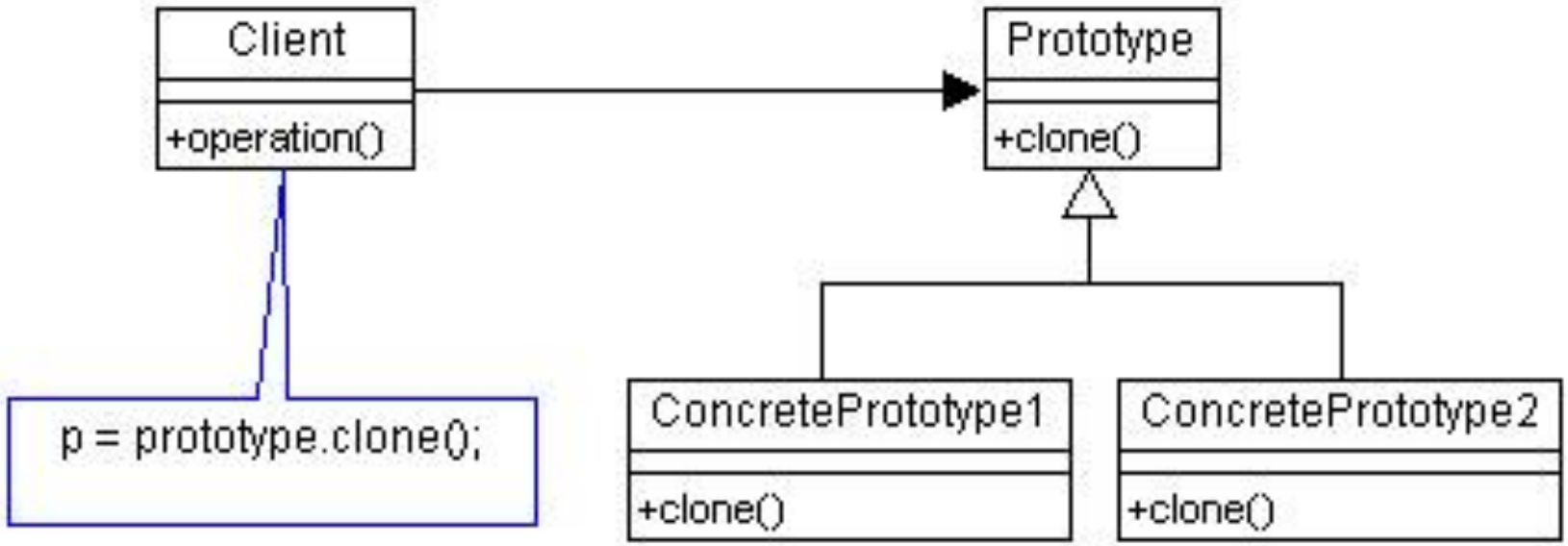
Prototype

- Intenção
  - Especificar os tipos de objetos a serem criados usando uma instância protótipo e criar novos objetos copiando o protótipo.
- Motivação
  - Uma ferramenta CAD em geral uma paleta de ferramentas e entre elas aquelas necessárias à inserção de figuras no projeto sendo editado. As figuras em geral são complexas e a sua replicação no projeto exige uma seqüência de operações que é particular de cada figura. Ao invés de se ter métodos de construção específico para cada figura, tem-se protótipos das figuras possíveis, cada um oferecendo um método clone() para a sua replicação.

Prototype

- Aplicabilidade
  - Quando as classes a instanciar são especificads em tempo de execução ou
  - Deseja-se evitar uma hierarquia de classes de fábricas paralelas à hierarquia ‘principal’ ou
  - As instâncias de classe podem ter uma entre poucas combinações diferentes de estados.

Prototype - Estrutura



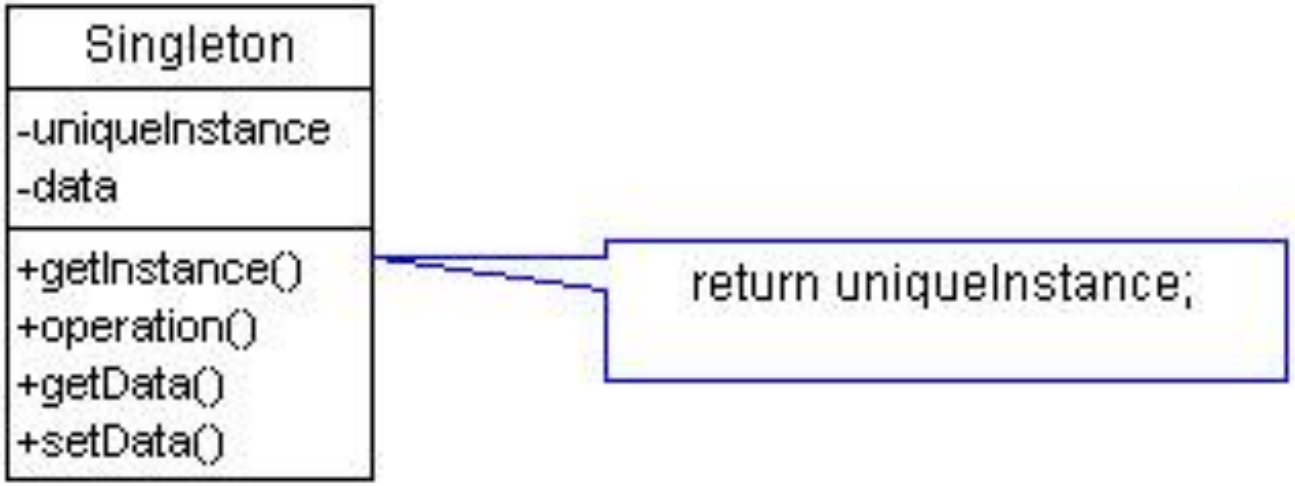
Prototype

- Colaborações
  - Um cliente solicita um protótipo e este clona a si próprio. Este será o responsável pela criação dos produtos concretos.
- Conseqüências
  - Permite acrescentar e remover produtos em tempo de execução
  - Especifica novos objetos pela variação de valores ou pela variação de estrutura
  - Reduz o número de subclasses
  - Configura dinamicamente as classes ou objetos da aplicação
  - Cada subclasse deve implementar a operação clone(), o que pode ser difícil em alguns casos
- Padrões Correlatos
  - Abstract Factory, Composite

Singleton

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

Singleton - Estrutura



Singleton

- Colaborações
  - Os clientes acessam uma única instância do Singleton pela operação Instance()
- Conseqüências
  - 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 Correlatos
  - AbstractFactory, Prototype, Builder

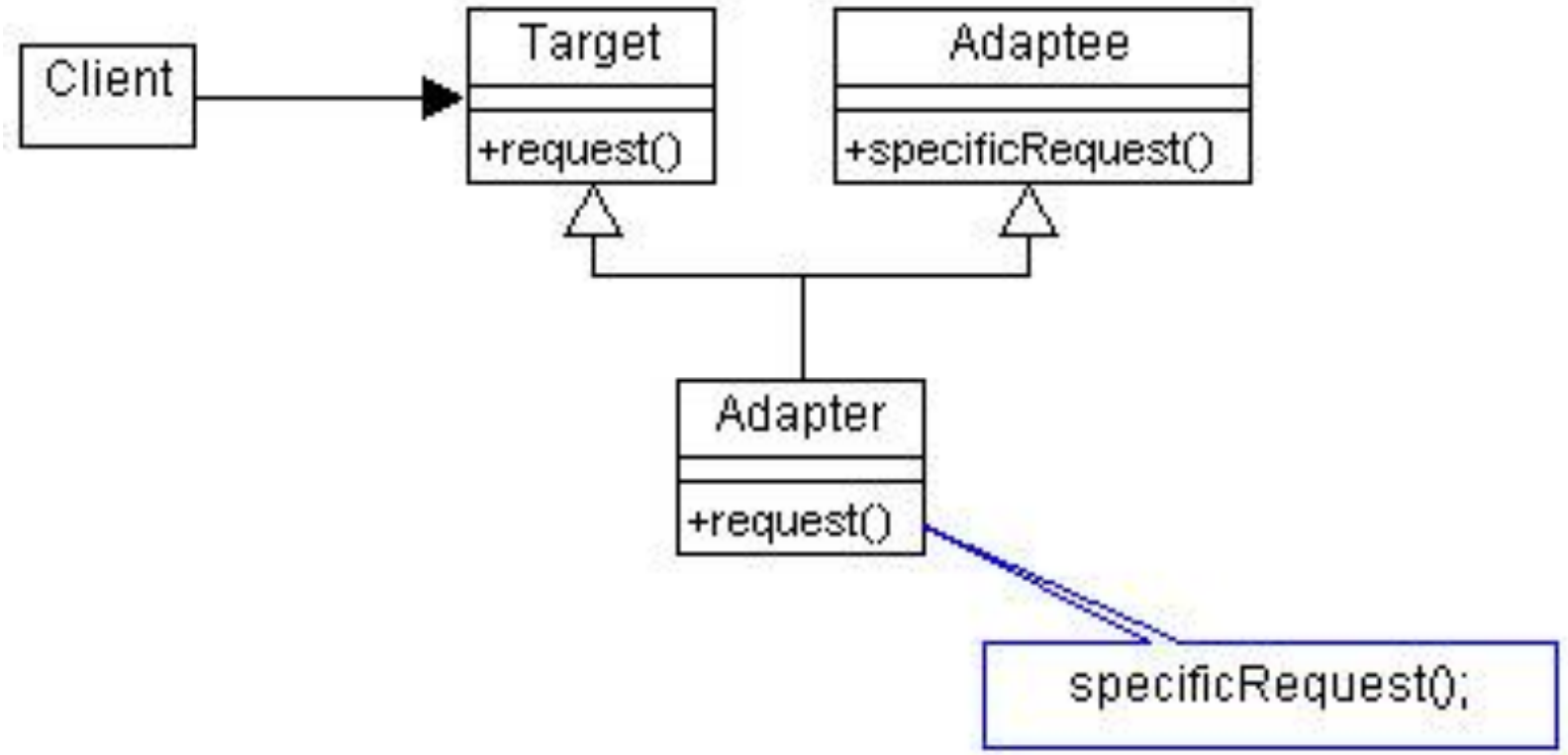
Padrões Estruturais

- Preocupam-se com a forma como classes e objetos são compostos para formar estruturas mariores.
- Utilizam herança pra compor interfaces ou implementações.
- Descrevem formas de compor objetos para obter novas funcionalidades.
- Possibilidade de mudar a composição em tempo de execução, o que é impossível com uma hierarquia de classes.

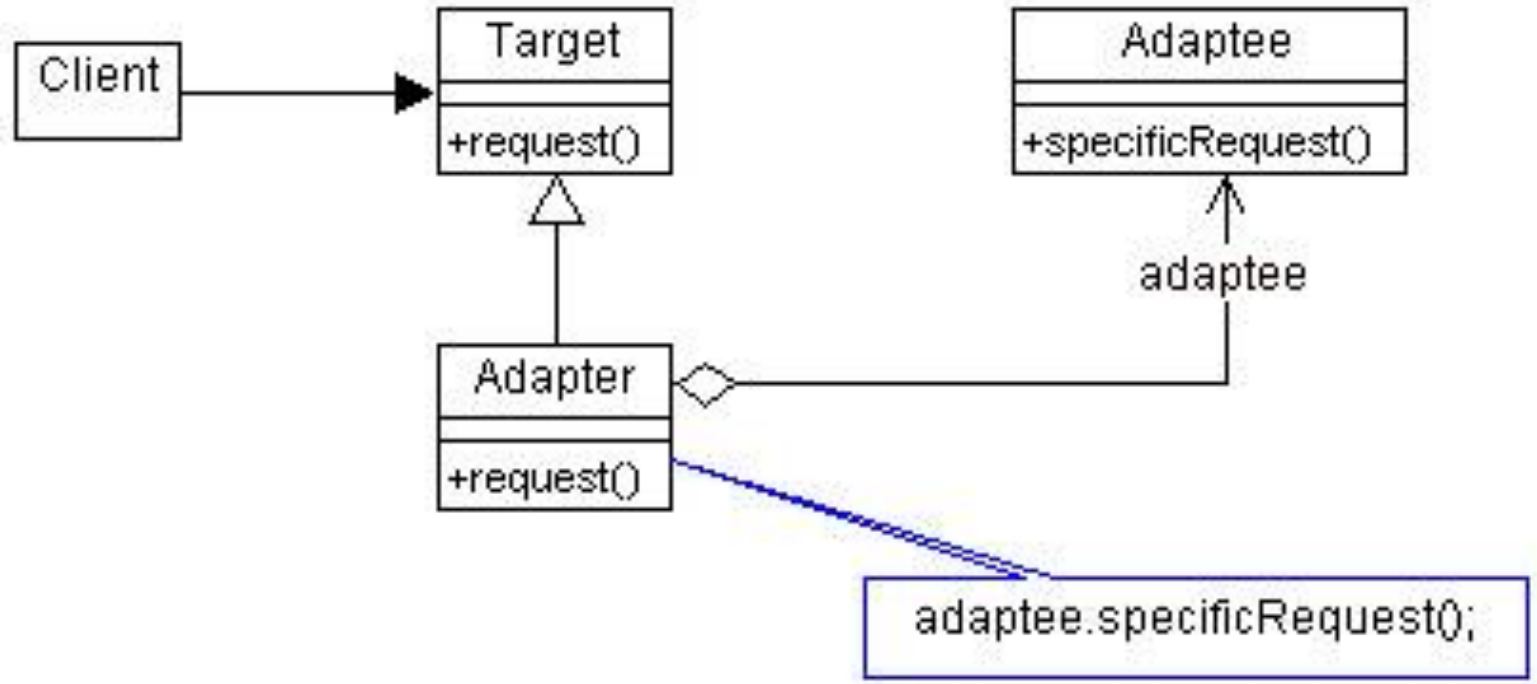
Adapter

- Intenção
  - converter a interface de uma classe na interface esperada pelos clientes. O Adapter permite que classes com interfaces incompatíveis trabalhem em conjunto.
- Motivação
  - Em algumas situações, a interface oferecida por um toolkit, projetada para ser reutilizada não pode ser usada numa aplicação porque sua interface não corresponde à interface específica.
- Aplicabilidade
  - situações nas quais as classes que devem interagir não têm interfaces compatíveis
  - adaptador de objetos é aplicável nos casos em que não é possível adaptar as classes existentes através de subclasses.

Adapter – Estrutura (1)



Adapter – Estrutura (2)



Adapter

- Colaborações
  - Os clientes chamam operações de uma instância de Adapter e este por sua vez chama as operações de Adaptee que executam a solicitação.
- Conseqüências (adaptador de classe)
  - um adaptador de classe nao funciona se quisermos adaptar uma dada classe e todas as suas subclasses.
  - é possível substituir algum comportamento do Adaptee, uma vez que Adapter é uma subclasse de Adaptee.
  - introduz somente um objeto intermediário, não sendo necessário endereçamento indireto adicional até se chegar ao Adaptee.

Adapter

- Conseqüências (adaptador de objetos)
  - permite a um único Adapter trabalhar com muitos Ataptees.
  - é difícil redefinir o comportamento de um Adaptee. Para isso é necessário a criação de subclasses.
- Pontos a considerar
  - o 'grau de adaptação' varia muito de uma situação para outra.
  - adaptadores plugáveis.
  - adaptadores nos dois sentidos para fornecer transparência
- Padrões Correlatos
  - Bridge, Decorator, Proxy

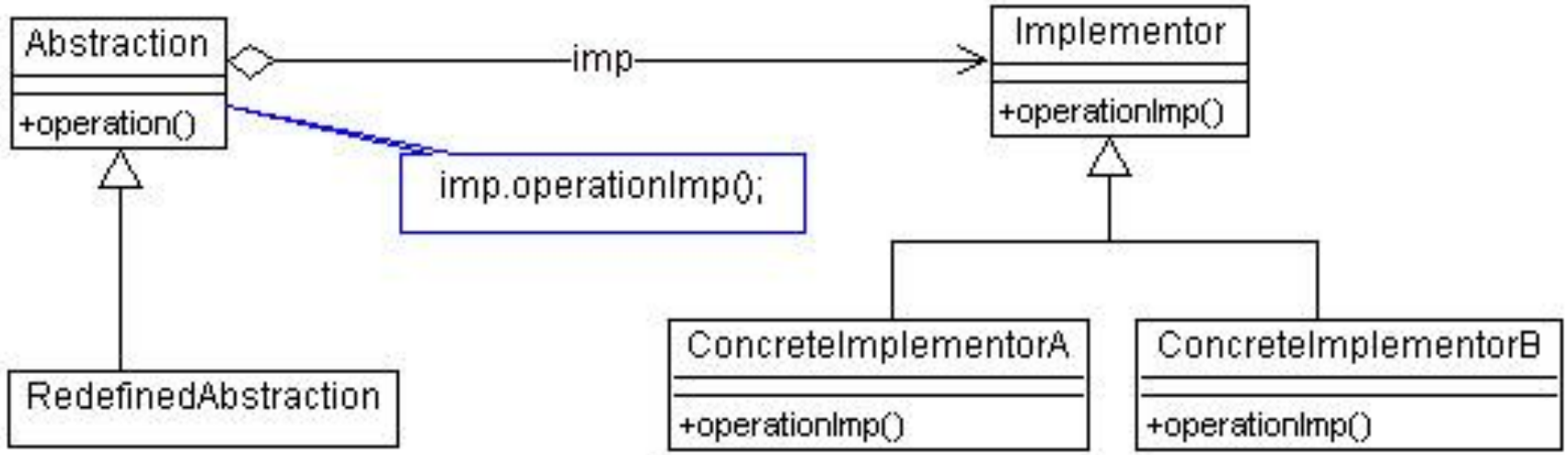
Bridge

- Intenção
  - Desacoplar uma abstração de sua implementação, de modo que as duas possam variar independentemente.
- Motivação
  - Em alguns casos, uma abstração pode ter mais de uma implementação possível e herança não é suficientemente flexível porque liga de forma permanente a abstração da implementação.

Bridge

- Aplicabilidade
  - deseja-se evitar o vínculo permanente da abstração e sua implementação. Isso pode ser necessário, por exemplo, quando a implementação deve ser definida ou alterada em tempo de execução.
  - tanto as abstrações como suas implementações devem ser extensíveis por meio de subclasses. Através do padrão Bridge é possível combinar as diferentes abstrações e implementações e extende-las independentemente.
  - deseja-se ocultar completamente a implementação dos clientes.
  - é necessário compartilhar uma implementação entre múltiplos objetos.

Bridge - Estrutura



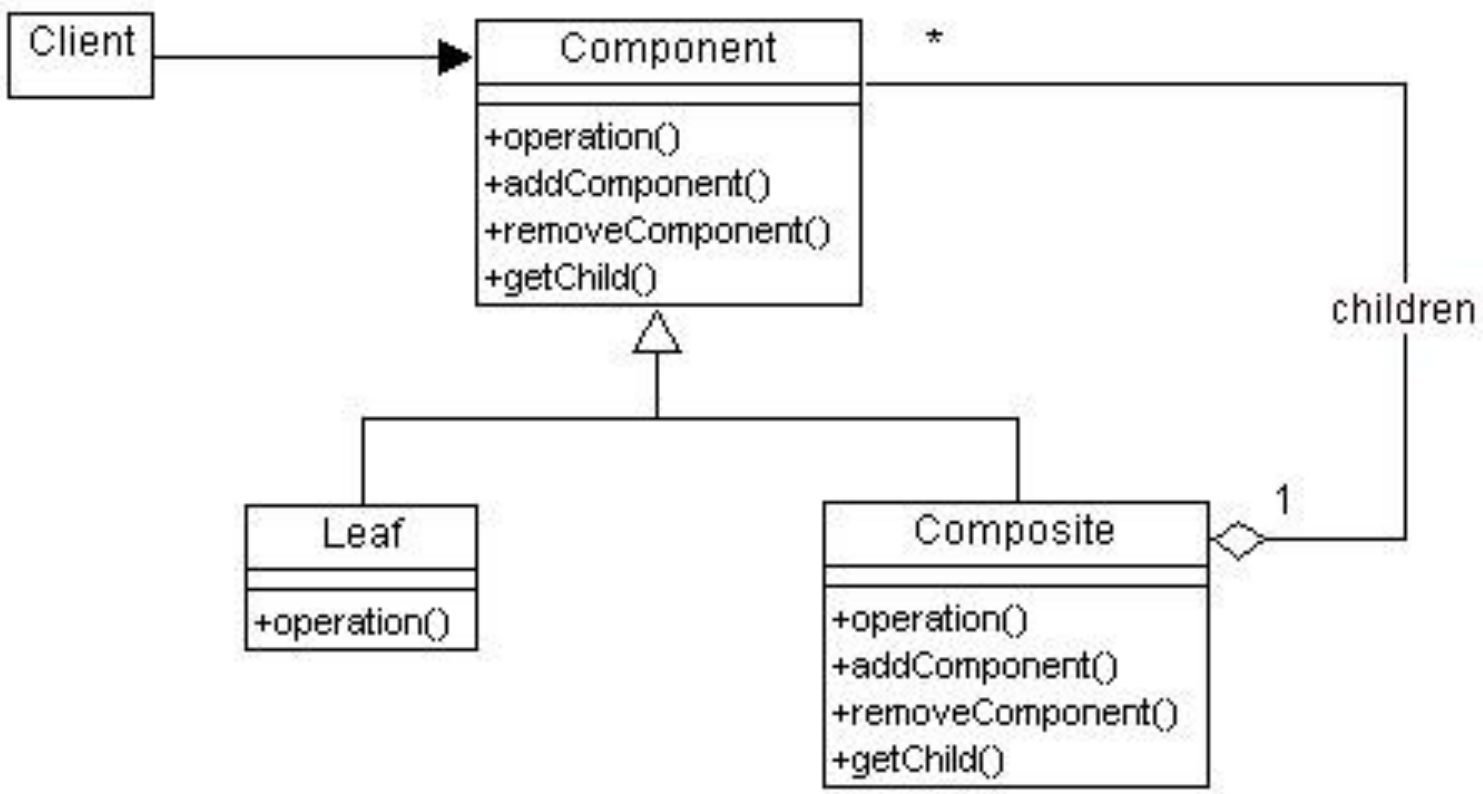
Bridge

- Colaborações
  - Abstraction repassa as solicitações dos clientes para o seu objeto Implementor
- Conseqüências
  - Desacopla a interface da implementação
  - É possível estender as hierarquias de Abstraction e Implementor de forma independente
  - É capaz de ocultar detalhes de implementação dos clientes
- Padrões Correlatos
  - AbstractFactory, Adapter

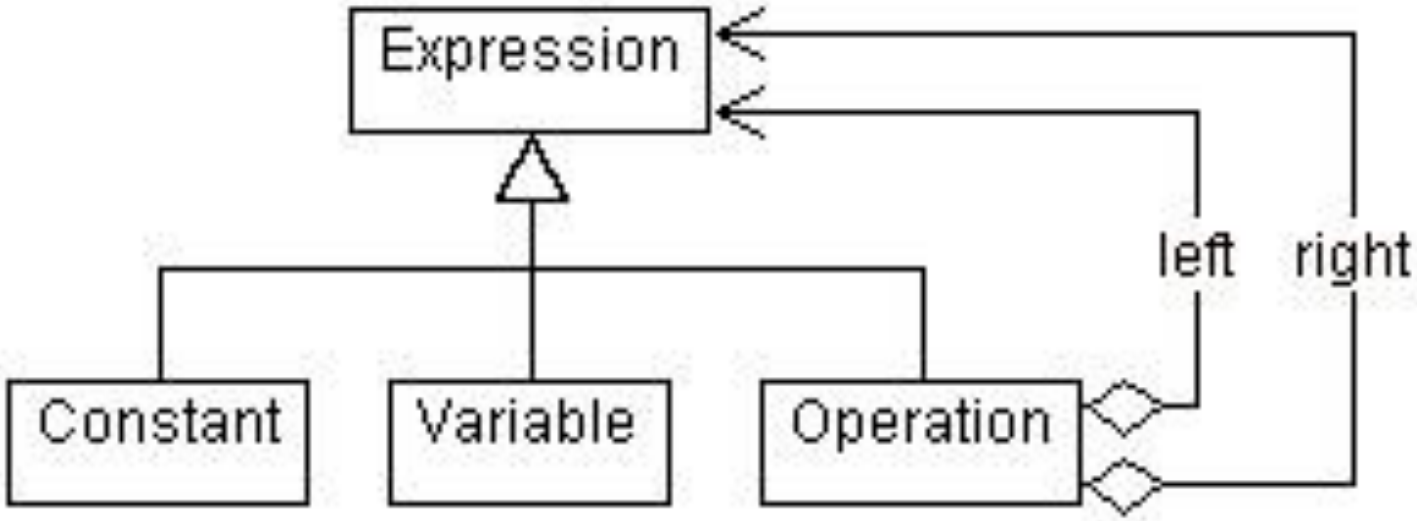
Composite

- 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.
- Aplicabilidade
  - representação de hierarquias todo-parte de objetos.
  - clientes devem ser capazes de ignorar a diferença entre composições de objetos e objetos individuais.

Composite - Estrutura



Composite - Exemplo



Composite

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

Composite

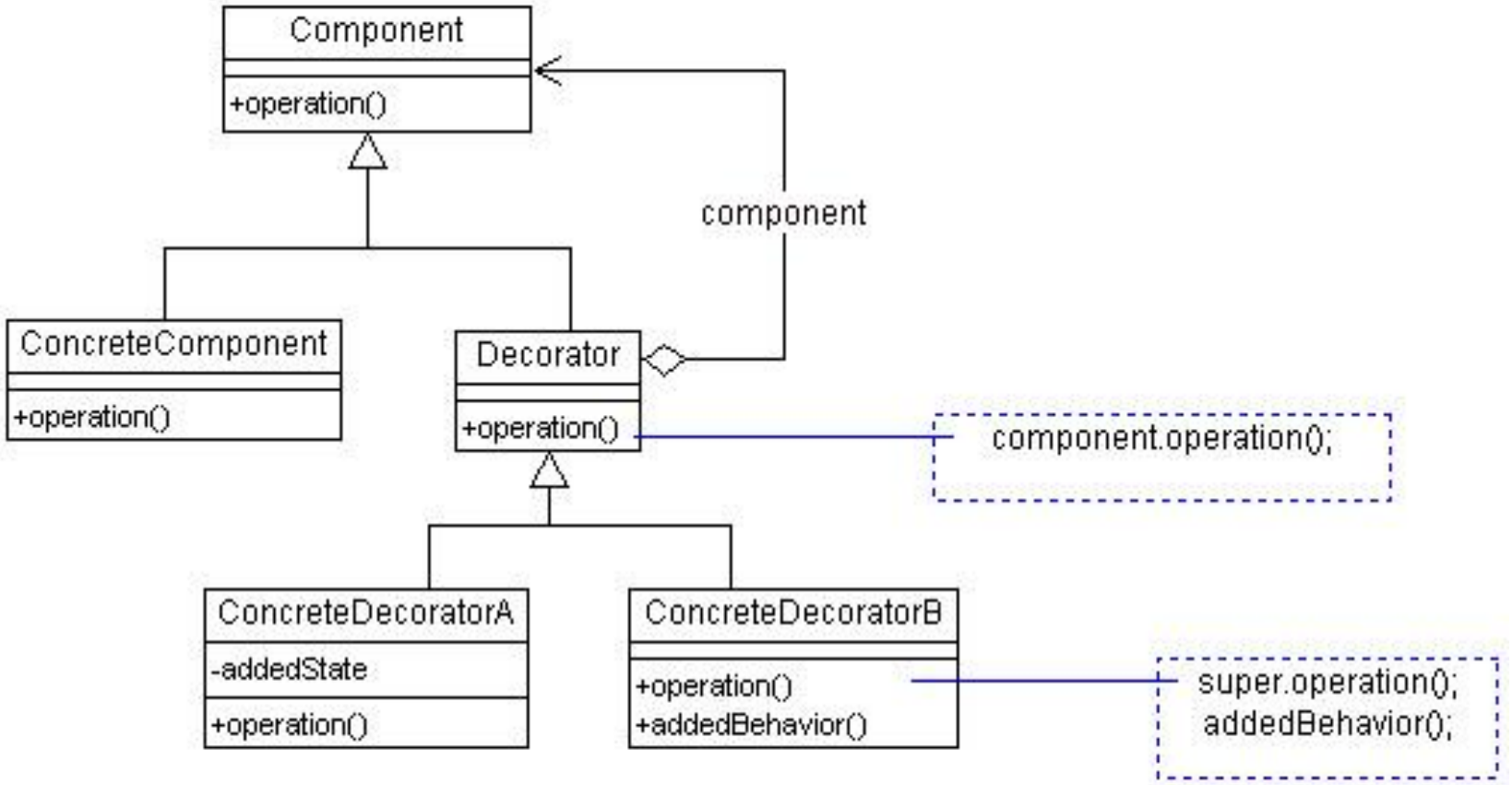
- Conseqüências
  - Referências explícitas aos pais.
  - Compartilhamento de componentes
  - Maximização da interface de Component

- Padrões Correlatos
  - Chain of Responsibility
  - Decorator
  - Flyweight
  - Iterator
  - Visitor

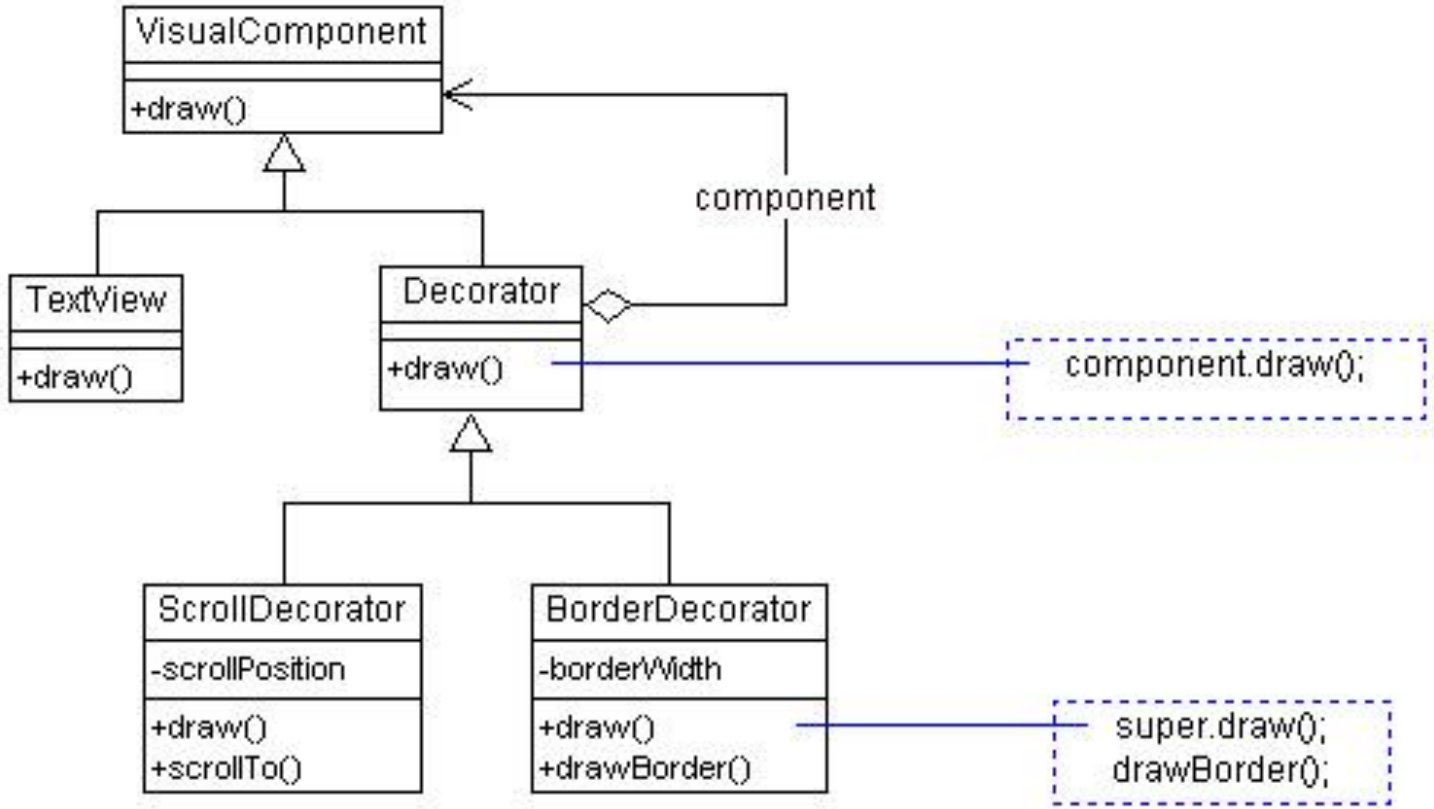
Decorator

- 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.
  - Exemplo: um toolkit para a construção de interfaces gráficas deve permitir adicionar propriedades como bordas ou barras de rolagem.
  - O uso de herança nesses casos é inflexível porque as novas propriedades precisam estar definidas em tempo de compilação.
- Aplicabilidade
  - 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.

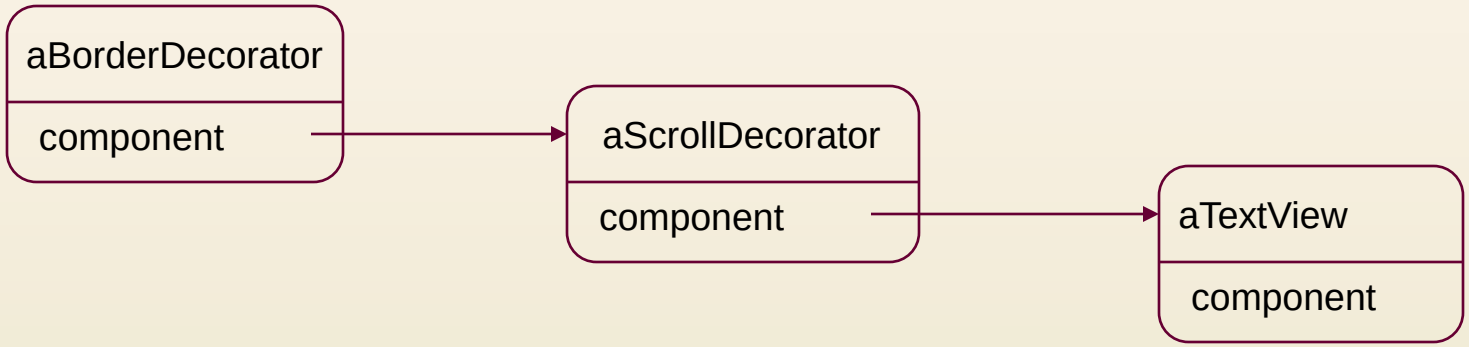
Decorator - Estrutura



Decorator – Exemplo (1)



Decorator – Exemplo (2)



Decorator

- Colaborações
  - Decorator repassa as solicitações para o seu objeto Component. Opcionalmente pode executar operações adicionais antes e depois de repassar a solicitação.
- Conseqüências
  - Maior flexibilidade que herança estática
  - Evita classes sobrecarregadas de métodos e atributos na raiz da hierarquia
  - Um Decorator e seu Component não são idênticos.
  - Grande quantidade de pequenos objetos
- Padrões Correlatos
  - Adapter, Composite, Strategy

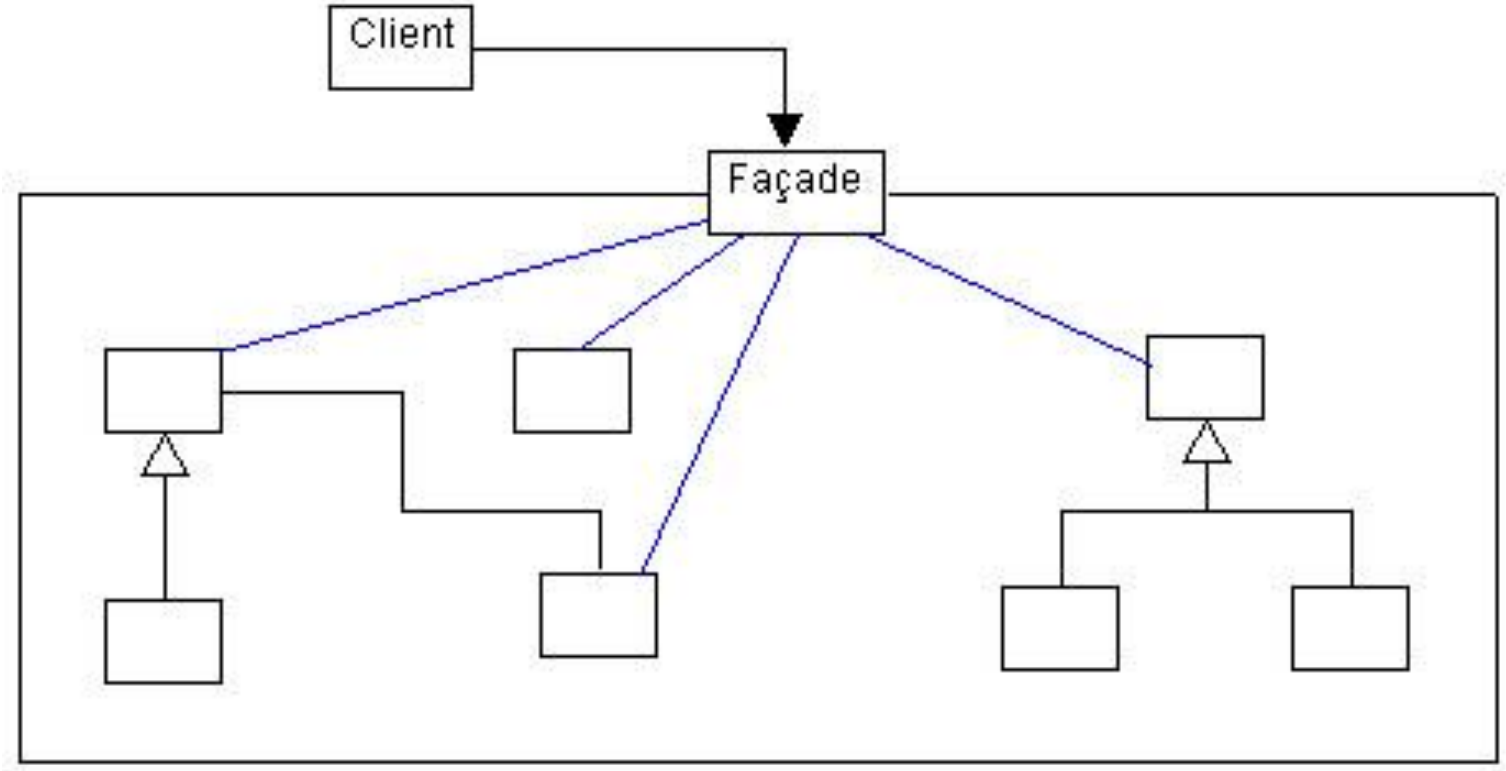
Façade

- Intenção
  - Fornecer uma interface unificada para um conjunto de interfaces de um subsistema.
  - Definir uma interface de nível mais alto que torna os subsistemas mais fáceis de serem utilizados.
- Motivação
  - Necessidade de estruturar um sistema em subsistema, facilitando o acesso e minimizando a comunicação e dependências entre os subsistemas.

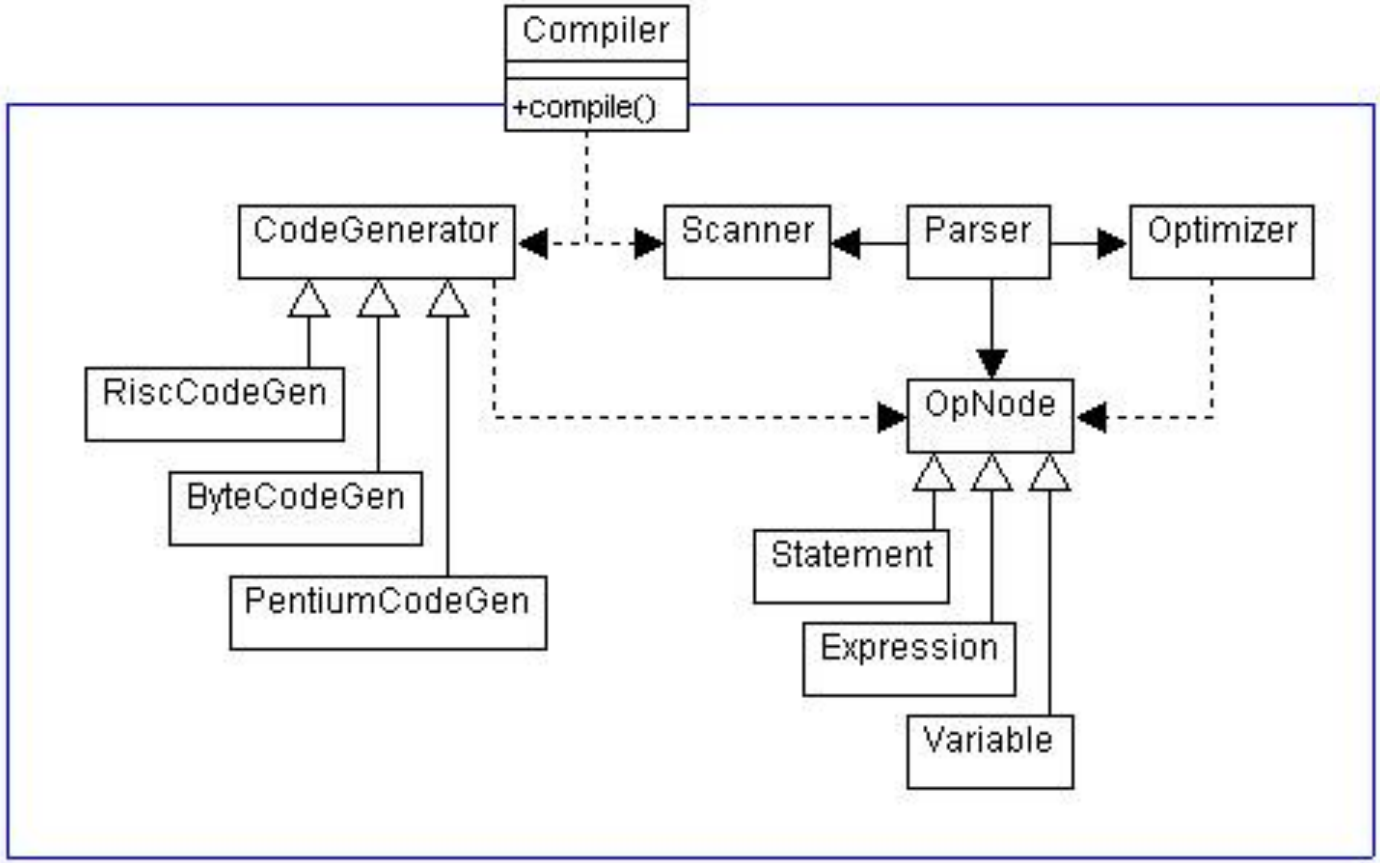
Façade

- Aplicabilidade
  - Deseja-se fornecer uma interface simples e unificada para um sistema complexo.
  - Deseja-se desacoplar os sub-sistemas dos clientes, promovendo-se a independência e portabilidade dos subsistemas.
  - Deseja-se estruturar o sistema em camadas.

Façade - Estrutura



Façade - Exemplo



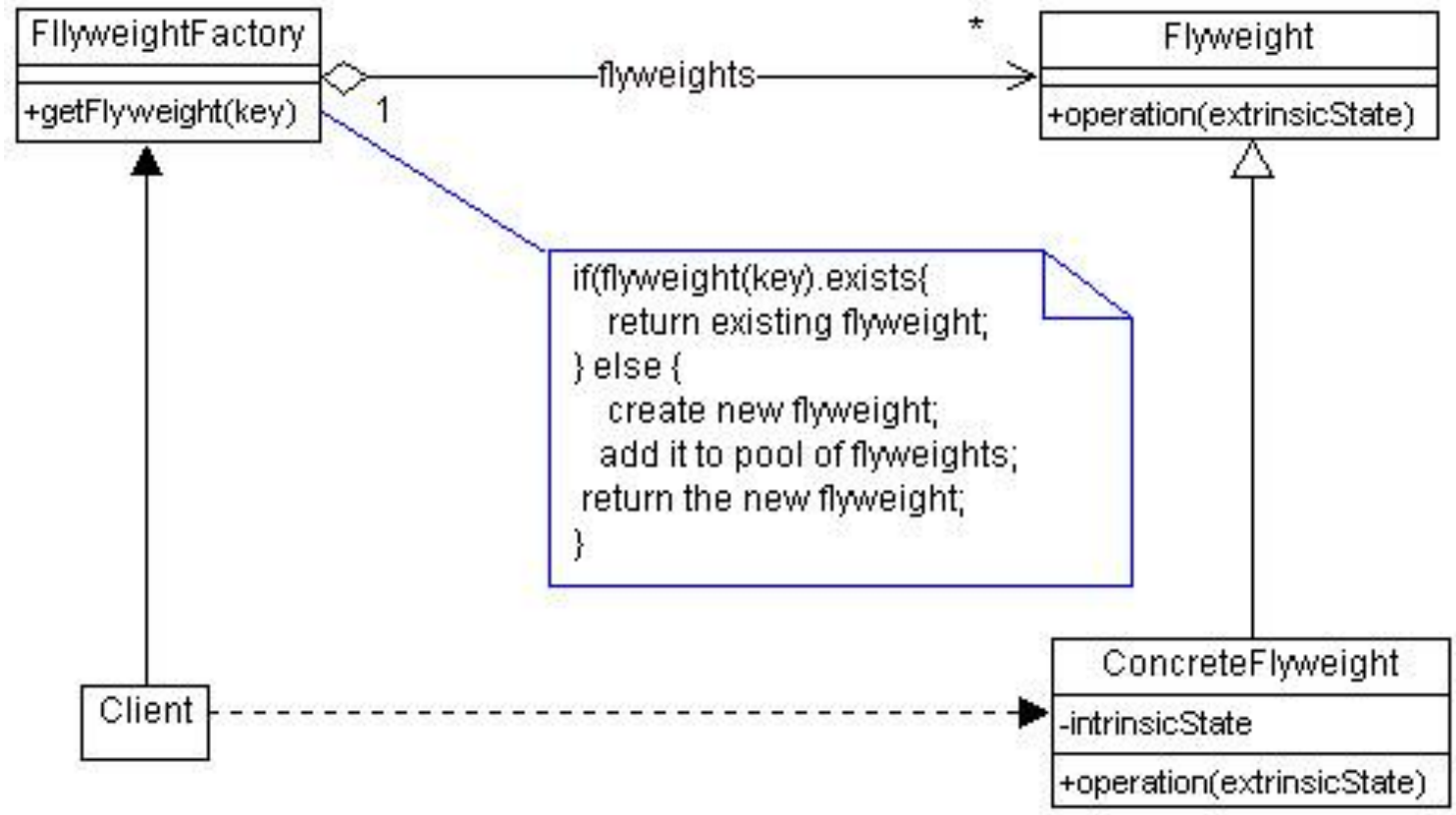
Façade

- Colaborações
  - Os clientes se comunicam com os subsistemas através de solicitações para Façade e este as repassa aos objetos apropriados.
  - Os clientes que usam o sistema através de Façade não precisam acessar os objetos do subsistema diretamente.
- Conseqüências
  - Isola os clientes dos subsistemas, tornando o sistema mais fácil de usar.
  - Promove o acoplamento fraco entre o subsistema e seus clientes.
  - Impede as aplicações de usar diretamente as classes dos subsistemas.
- Padrões Correlatos
  - Abstract Factory, Mediator, Singleton

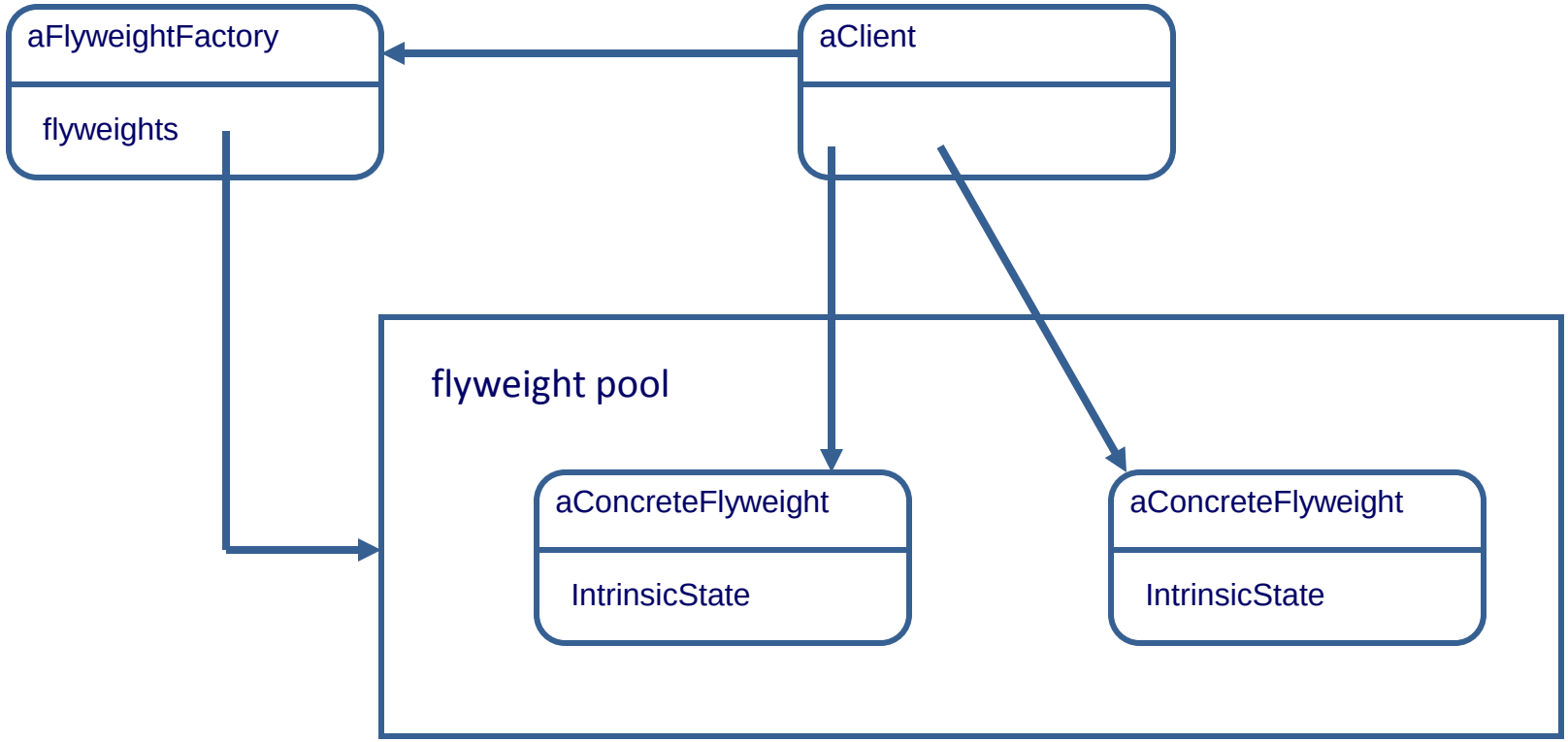
Flyweight

- Intenção:
  - Usar compartilhamento para suportar eficientemente grandes quantidades de objeto de granularidade fina.
- Motivação:
  - Existem situações nas quais objetos se repetem em grandes quantidades, sendo que as diferenças de configuração entre eles são muito pequenas. É uma situação comum em editores de texto.
- Aplicabilidade
  - A aplicação utiliza um grande número de objetos
  - Os custos de armazenamento são altos
  - Os estados do objeto podem ser extrínsecos
  - Muitos grupos de objetos podem ser substituídos por relativamente poucos objetos compartilhados
  - A aplicação não depende da quantidade de objetos.

Flyweight - Estrutura



Flyweight - Estrutura

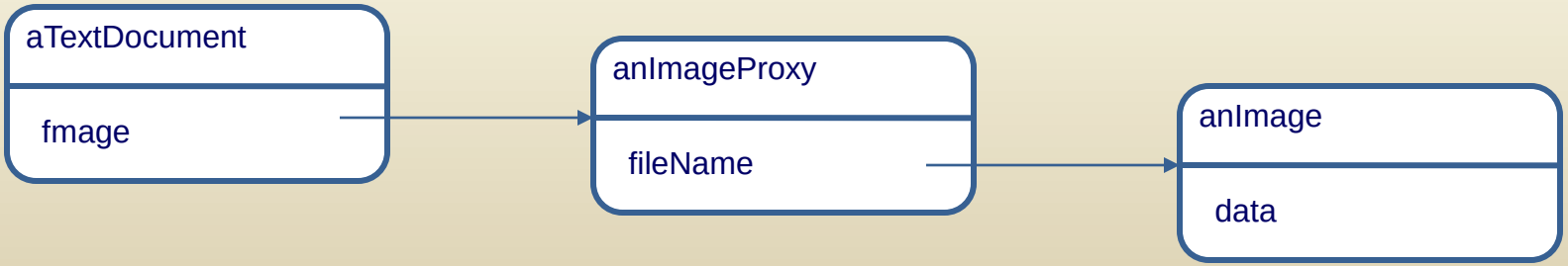


Flyweight

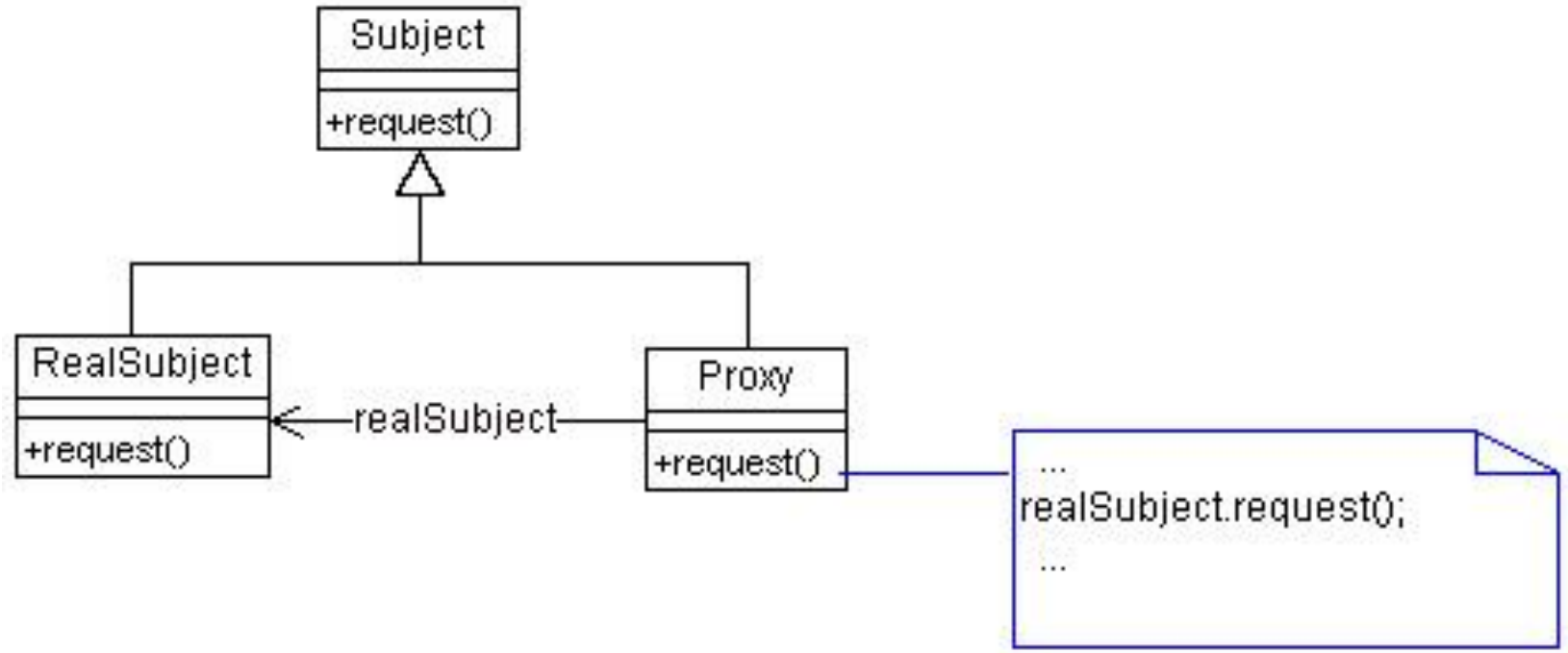
- Colaborações
  - O estado que um flyweight necessita para funcionar deve ser caracterizado com o intrínseco ou extrínseco
    - Intrínseco: armazenado no objeto ConcreteFlyweight
    - Extrínseco: armazenado o cliente e usado quando este invoca as operações
  - Os clientes devem obter os objetos ConcreteFlyweight através do objeto FlyweightFactory para garantir que sejam compartilhados.
- Consequências
  - Redução do número de instâncias (compartilhamento)
  - Redução do número de estados intrínsecos por objeto
- Padrões Correlatos
  - Composite, State e Strategy

Proxy

- Intenção
  - Fornecer um substituto ou marcador da localização de outro objeto para controlar o acesso ao mesmo.
- Motivação
  - Deseja-se adiar o custo da criação do 'objeto completo' para quando ele seja realmente necessário.



Proxy - Estrutura



Proxy

- Colaborações
  - Proxy repassa as solicitações para o RealSubject quando necessário.
- Consequências
  - Um Proxy remoto pode ocultar o fato de que um objeto reside num espaço de endereçamento diferente.
  - Um proxy virtual pode executar otimizações como por exemplo a criação de objetos sob demanda.
  - Proxies de proteção e ‘smart references’.
- Padrões relacionados
  - Adapter, Decorator

Padrões Estruturais - discussão

- Adapter x Bridge
  - Intenções
    - Adapter: focaliza a compatibilização das interfaces
    - Bridge: estabelece a ponte entre a abstração e sua implementação.
  - Aplicáveis a pontos diferentes do ciclo de vida do software
    - Adapter: necessário quando se deteta que é necessário compatibilizar interfaces diferentes, já implementadas.
    - Bridge: no início do projeto se descobre que uma abstração pode ter implementações diferentes.
- Adapter x Façade
  - Façade define uma nova interface
  - Adapter adapta interfaces já existentes

Padrões Estruturais - discussão

- Composite x Decorator x Proxy
  - Composite e Decorator apresentam estruturas semelhantes.
  - Composite: composição recursiva
  - Decorator permite acrescentar responsabilidades a objetos sem usar subclasses.
  - Composite estrutura as classes de forma que ‘objetos compostos’ possam ser tratados de maneira uniforme.
  - Composite e Decorator são frequentemente usados em conjunto.
  - Decorator e Proxy têm estruturas similares: ambos descrevem uma forma de ‘endereçamento indireto’ para os objetos.
  - Decorator e Proxy têm finalidades diferentes: Proxy não se preocupa em incluir novas funcionalidades.

Padrões Comportamentais

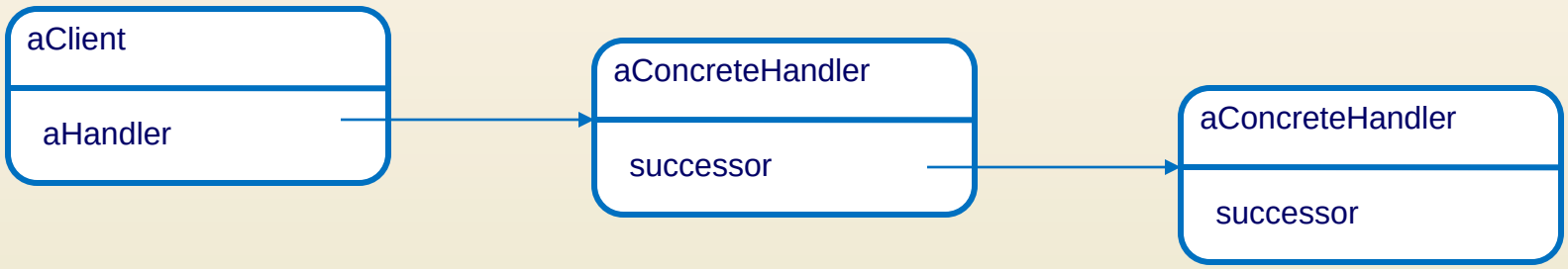
- Preocupam-se com algoritmos e atribuição de responsabilidades entre objetos.
- Descrevem tanto padrões de *objetos* e *classes* e também *padrões de comunicação* entre eles.
- Padrões comportamentais *de classes* utilizam herança para distribuir o comportamento entre *classes*.
- Padrões comportamentais *de objetos* utilizam composição em vez de herança para distribuir o comportamento entre *objetos*.

Chain of Responsability

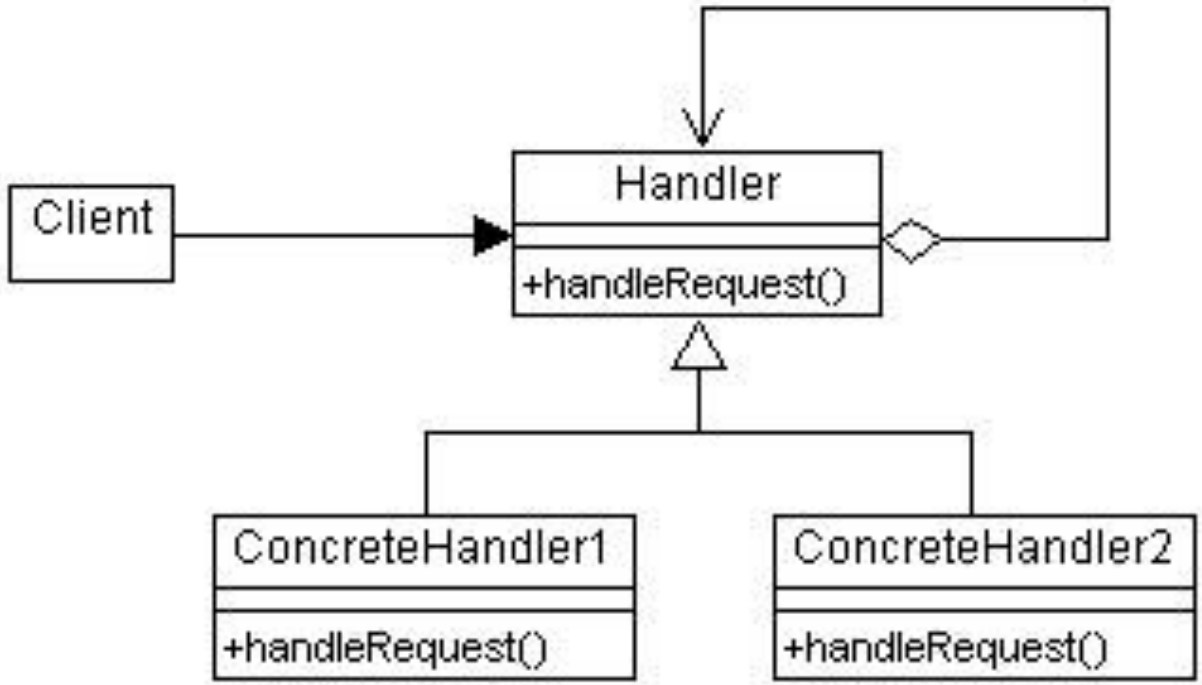
- Intenção
  - Evitar o acoplamento do remetente de uma solicitação ao seu receptor, dando a mais de um objeto a oportunidade de tratar uma solicitação.
  - Encadear os objetos receptores passado a solicitação ao longo da cadeia até que um objeto a trate.
- Motivação
  - Situações nas quais uma solicitação deve ser tratada por uma sequência de receptores que só é definida em tempo de execução.
- Aplicabilidade
  - Mais de um objeto pode tratar uma solicitação e o tratador não é conhecido *a priori*.
  - O conjunto de objetos que pode tratar a solicitação é definido dinamicamente.

Chain of Responsability

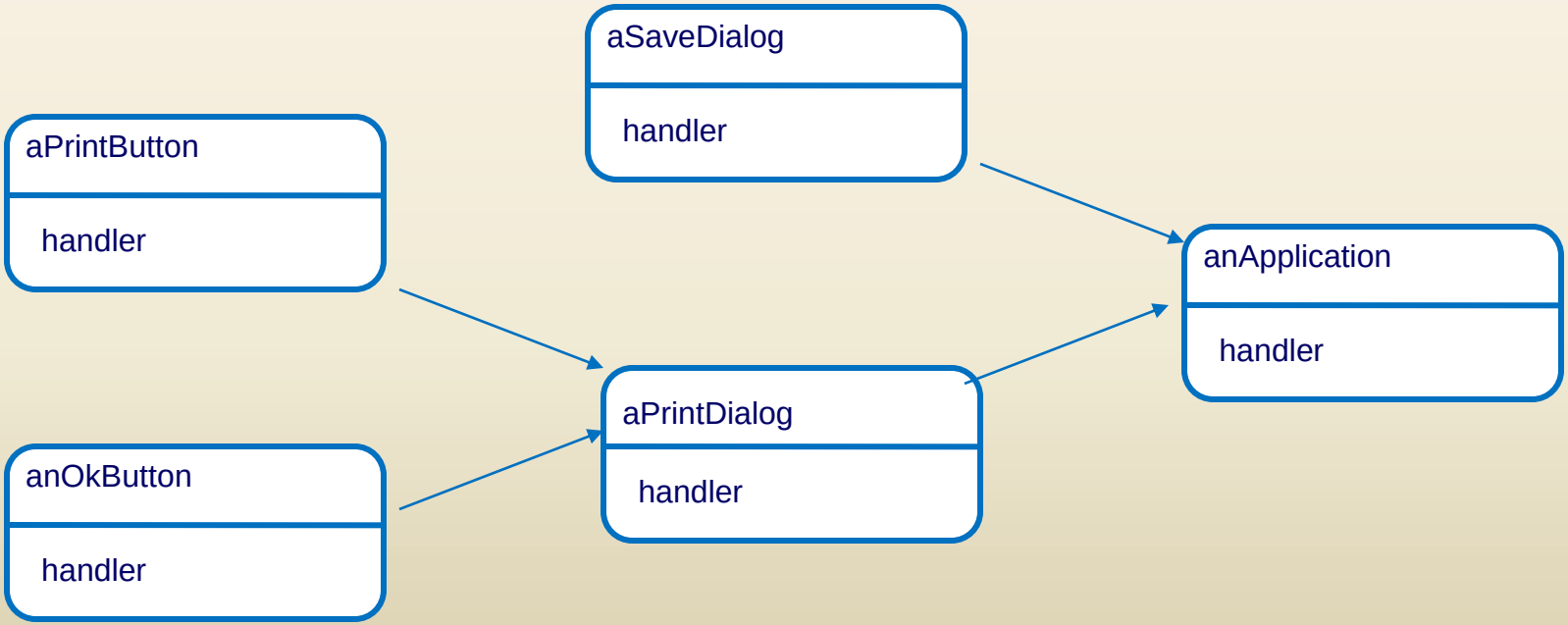
Estrutura



Chain of Responsability
Estrutura



Chain of Responsability
Exemplo



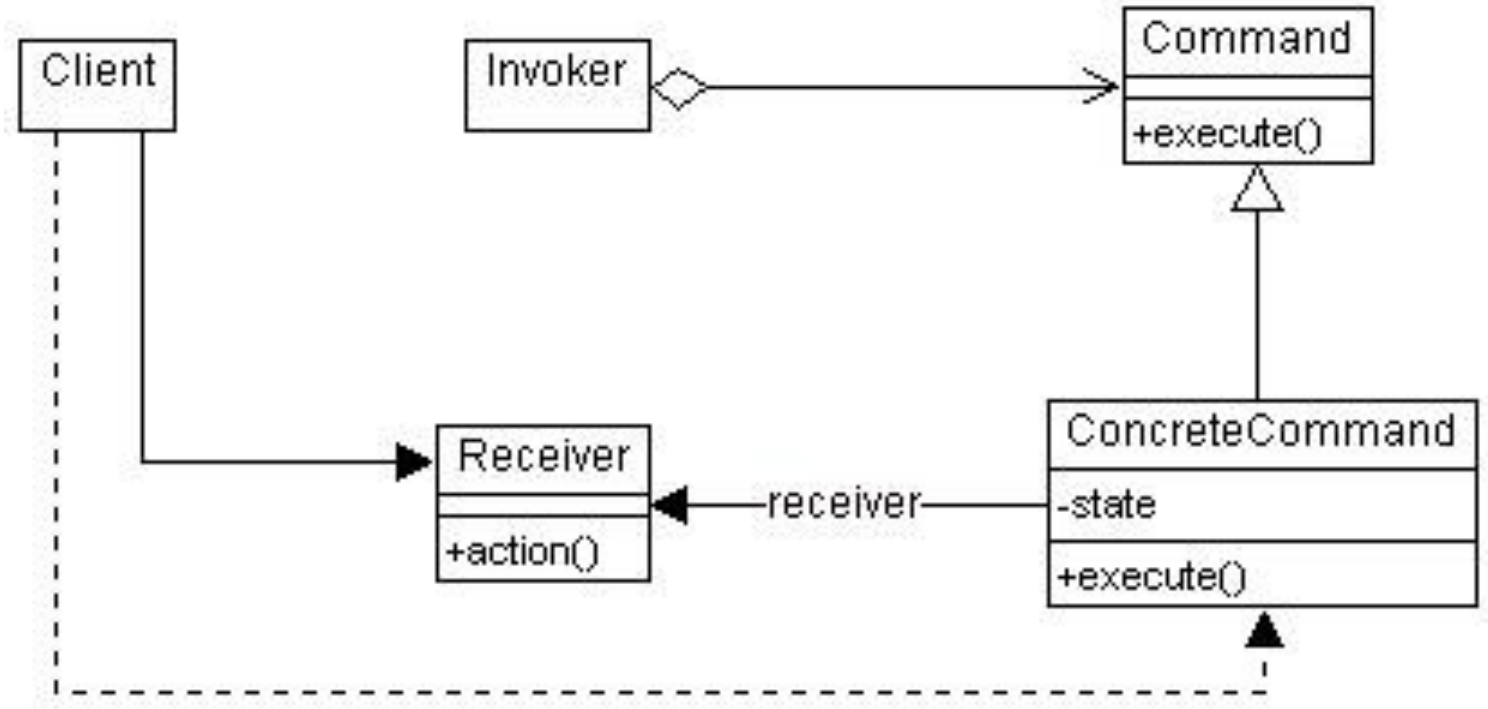
Chain of Responsability

- Colaborações
  - Quando um cliente emite uma solicitação, esta se propaga ao longo da cadeia até que um objeto ConcreteHandler assuma a responsabilidade de tratá-lo.
- Conseqüências
  - Acoplamento reduzido.
  - Flexibilidade na atribuição de responsabilidades.
  - A recepção não é garantida.
- Padrões Correlatos
  - Composite

Command

- 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.
- Aplicabilidade: situações em que deseja-se
  - parametrizar as ações a serem execuadas 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 operaçõe básicas.

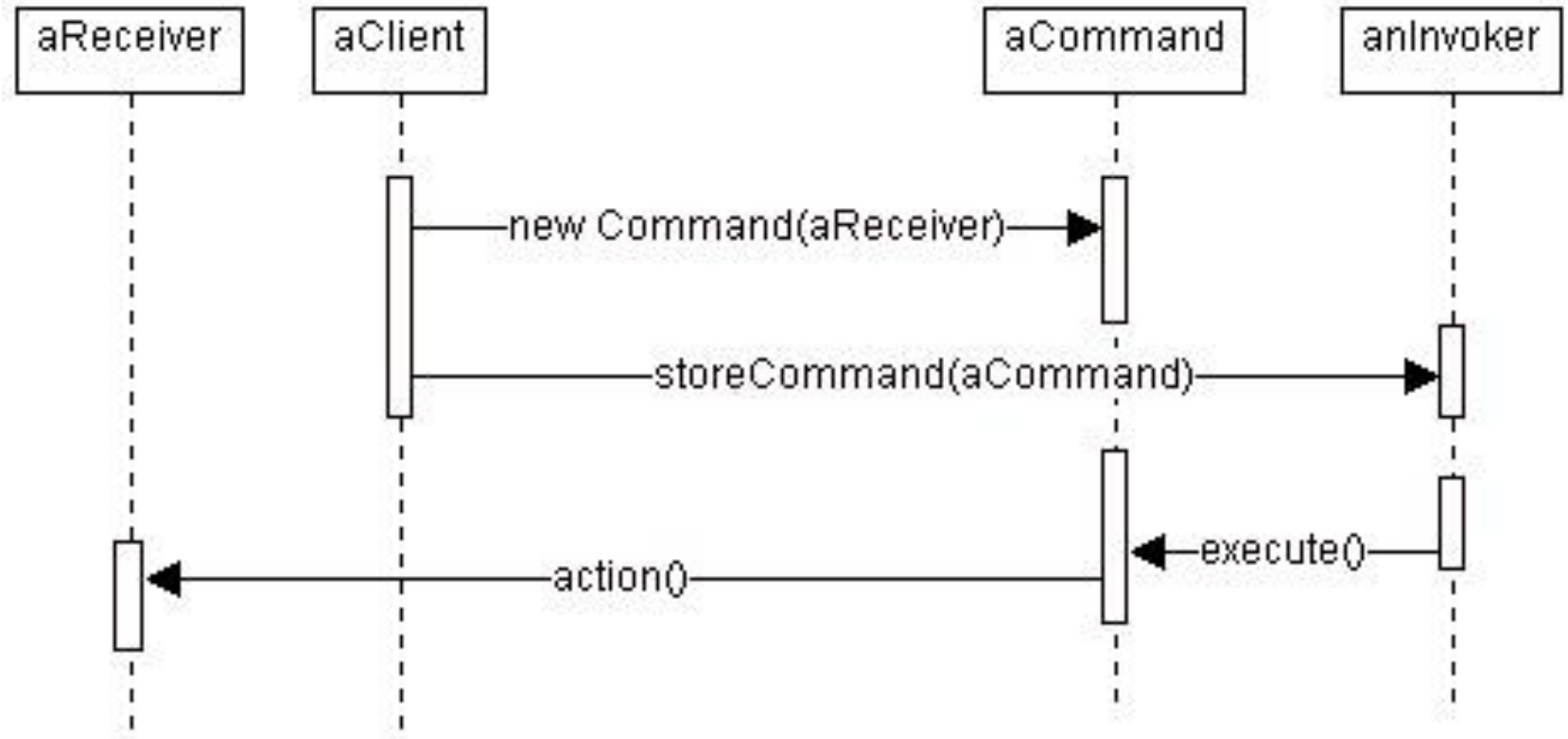
Command - Estrutura



Command

- Colaborações
  - O cliente cria um objeto ConcreteCommand e especifica o seu receptor
  - Um objeto Invoker armazena o objeto ConcreteCommand
  - O Invoker emite uma solicitação chamando Execute no Command. Caso os comandos preciser ser desfeitos, ConcreteCommand armazena estados para desfazer o comando antes de invocar o método *execute().*
  - O objeto ConcreteCommand invoca operações no seu Receiver para executar a solicitação.

Command


aReceiver
aClient
aCommand
anInvoker
new Command(aReceiver)
storeCommand(aCommand)
execute()
action()

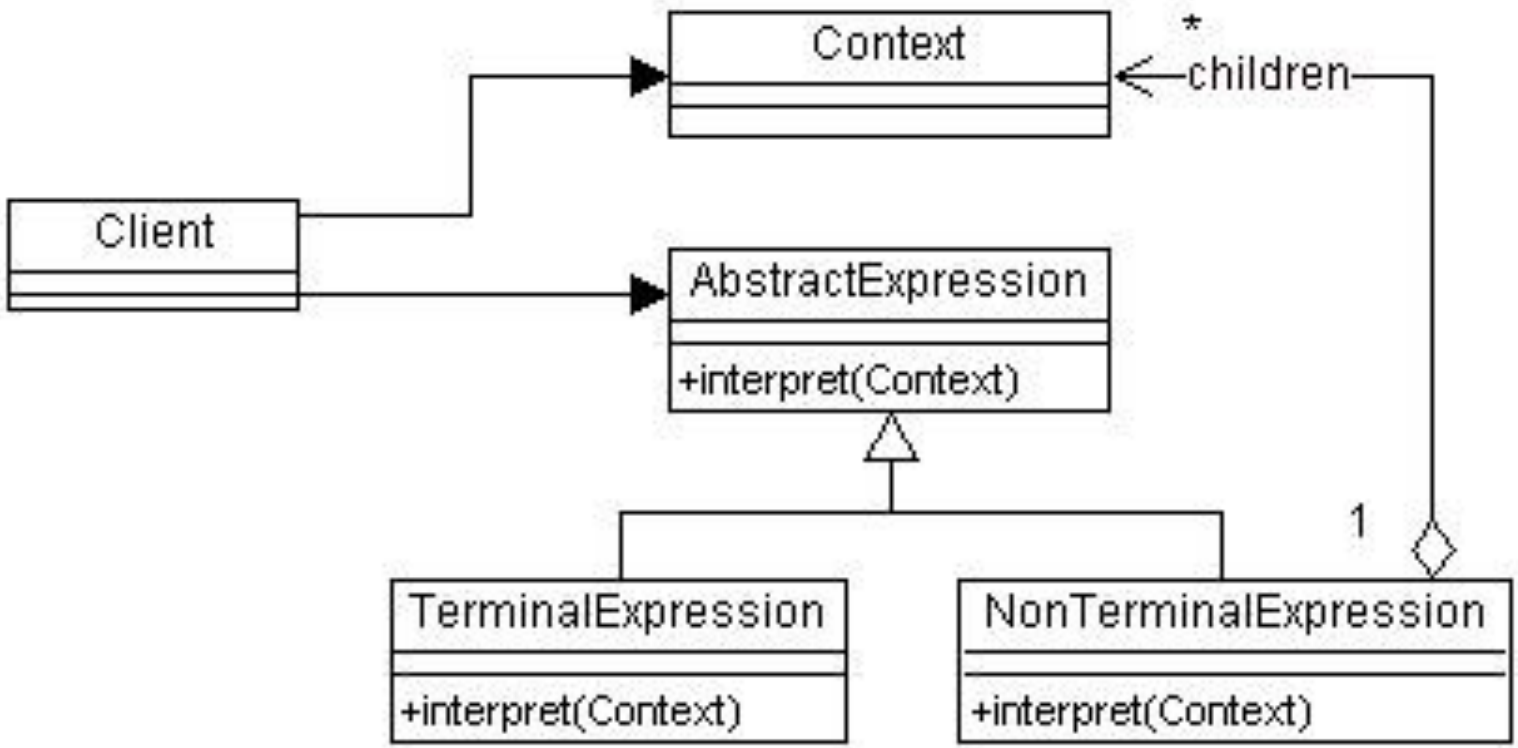
Command

- Conseqüências
  - 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 Correlatos
  - Composite, Memento, Prototype

Interpreter

- Intenção
  - Dada uma linguagem, interpretar sentenças nessas linguagens.
- Motivação
  - Existem problemas que ocorrem com freqüência e para os quais é possível expressar suas instâncias como sentenças de uma linguagem. Exemplos: pesquisa de padrões em cadeias de caracteres (*pattern matching*) que podem ser descritos por *expressões regulares*.
- Aplicabilidade: situações tais que
  - Gramáticas simples
  - Eficiência não é crítica.

Interpreter - Estrutura



Interpreter

- Colaborações
  - O cliente invoca a operação *interpret()* para uma árvore sintática formada por instâncias das subclasses de AbstractExpression.
  - Os nós da árvore são interpretados ‘recursivamente’ (cada nó do tipo NonTerminalExpression invoca *interpret()* para os seus filhos).
  - As operações *interpret()* em cada nó utilizam o contexto definido pelo cliente para armazenar e acessar o estado do interpretador.

Interpreter

- Conseqüências
  - É fácil implementar, alterar e estender gramáticas simples.
  - Gramáticas complexas são difíceis de manter.
  - Pode se usar diferentes formas de interpretar as expressões
- Padrões correlatos
  - Composite
  - Flyweight
  - Iterator
  - Visitor

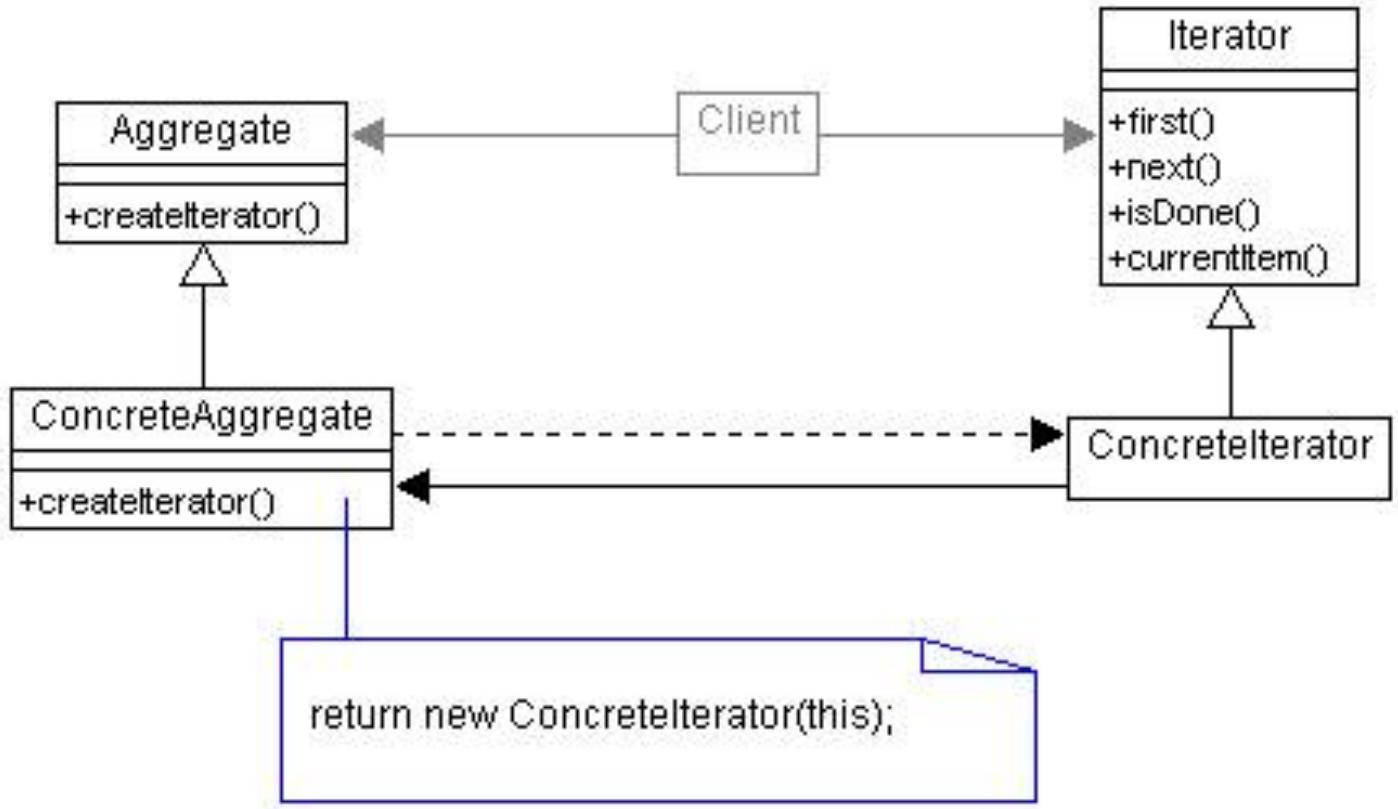
Iterator

- 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.

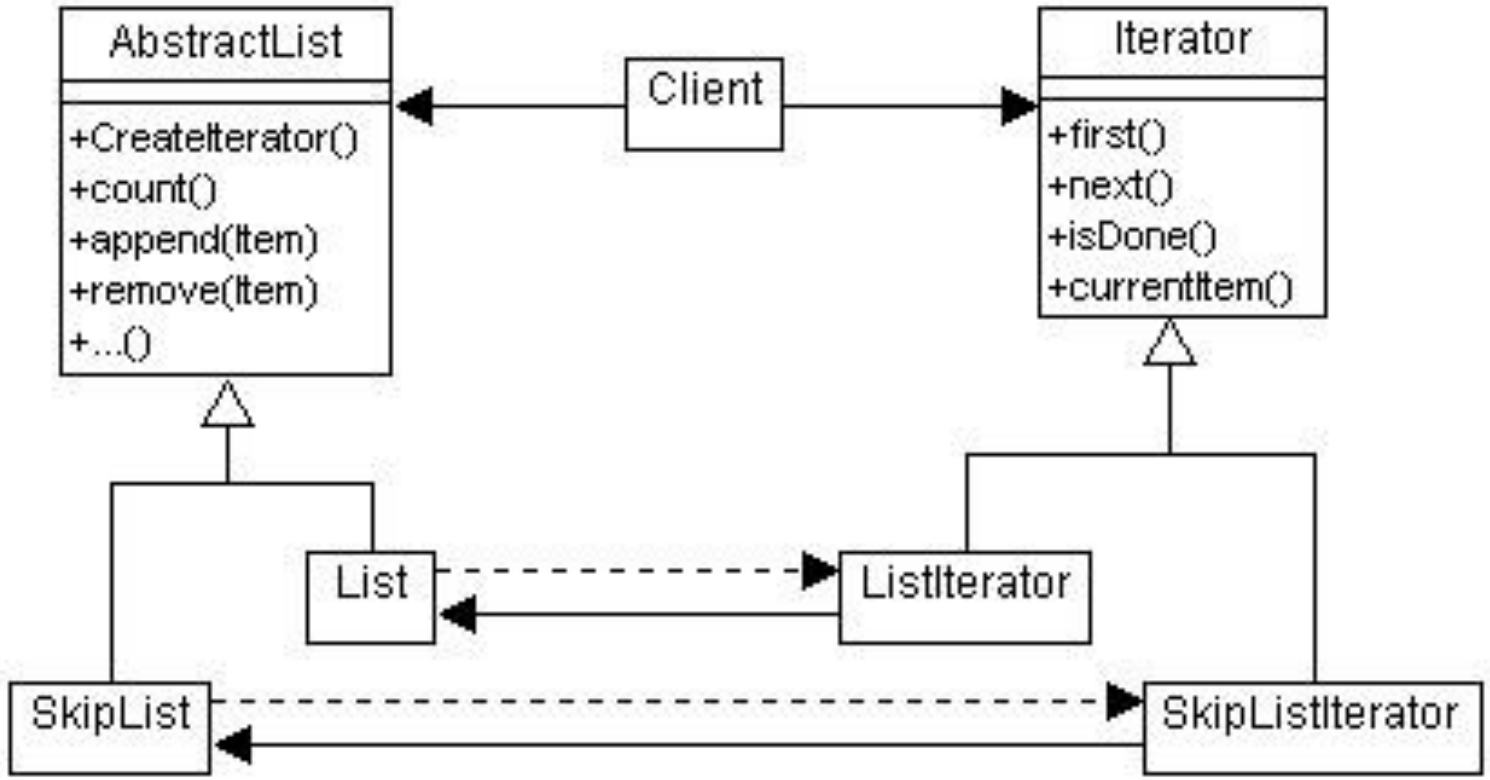
Iterator

- Aplicabilidade
  - Para acessar os conteúdo 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’).

Iterator - Estrutura



Iterator - Exemplo



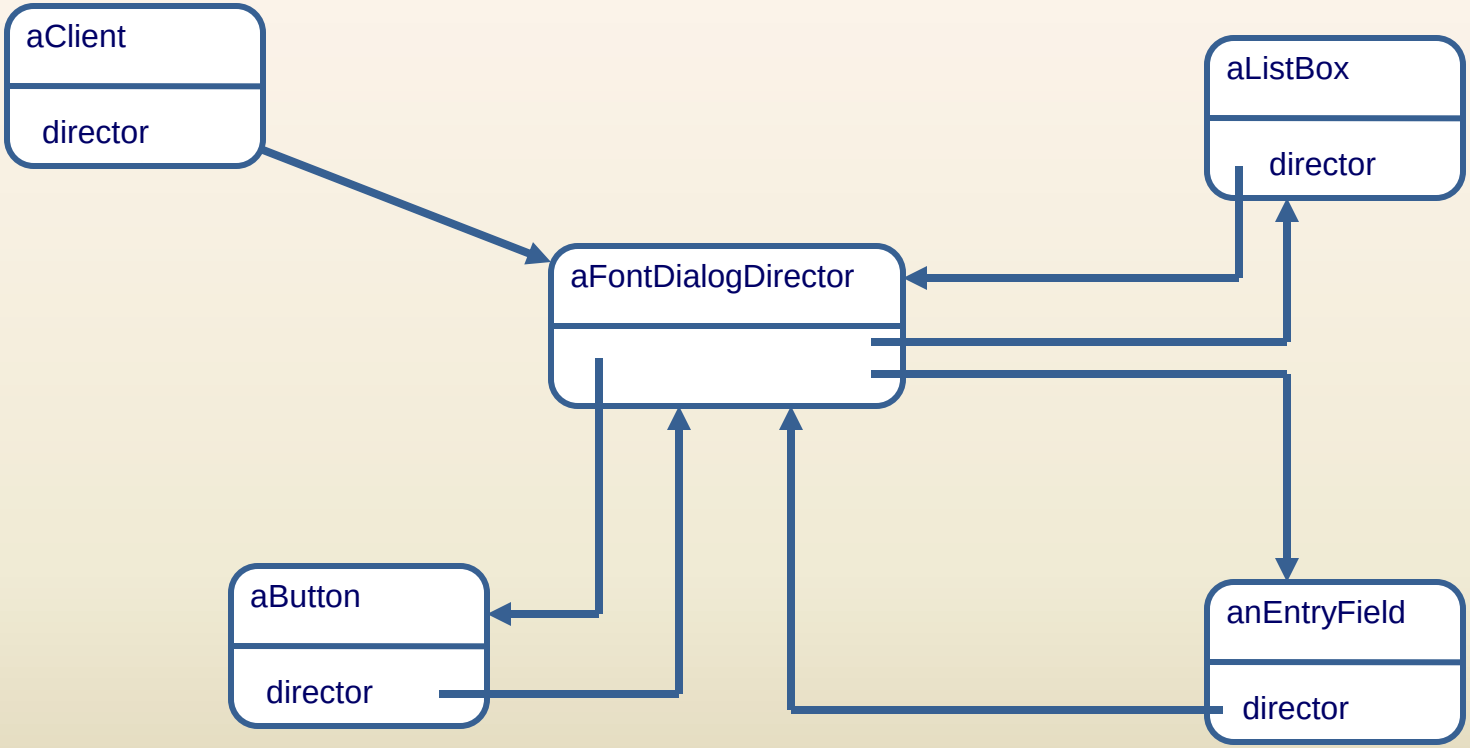
Iterator

- Colaborações
  - Um objeto ConcreteIterator mantém o controle do objeto corrente no agregado de objeto e consegue definir o seu sucessor durante o percurso.
- Conseqüências
  - 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 correlatos
  - Composite, FactoryMethod, Memento.

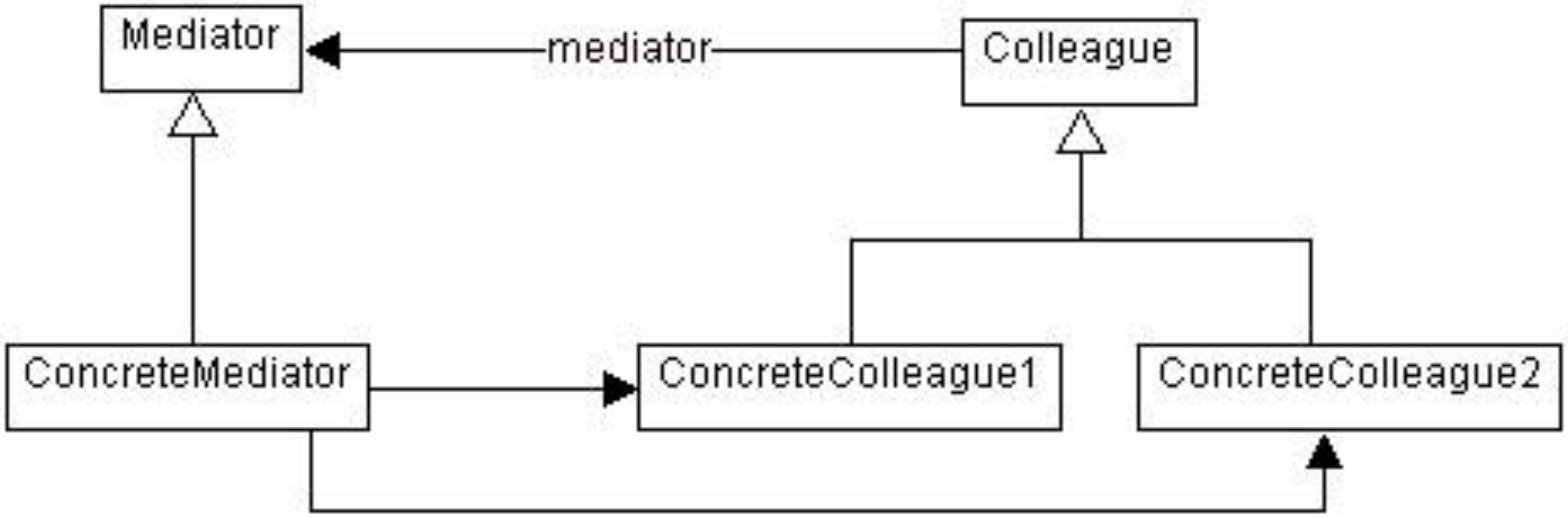
Mediator

- Intenção
  - Definir um objeto que encapsula a forma como um conjunto de objetos interage.
  - Promove o acoplamento fraco entre os objetos ao evitar que os objetos explicitamente se refiram uns aos outros, permitindo que se varie independentemente as interações.
- Motivação
  - Em projetos orientados a objetos, é normal distribuir o comportamento entre várias classes. Isso pode resultar numa estrutura com muitas conexões entre os objetos e gera a necessidade de que cada objeto conheça os demais.

Mediator


aClient
director
aListBox
director
aFontDialogDirector
aButton
director
anEntryField
director

Mediator


Mediator
mediator
Colleague
ConcreteMediator
ConcreteColleague1
ConcreteColleague2

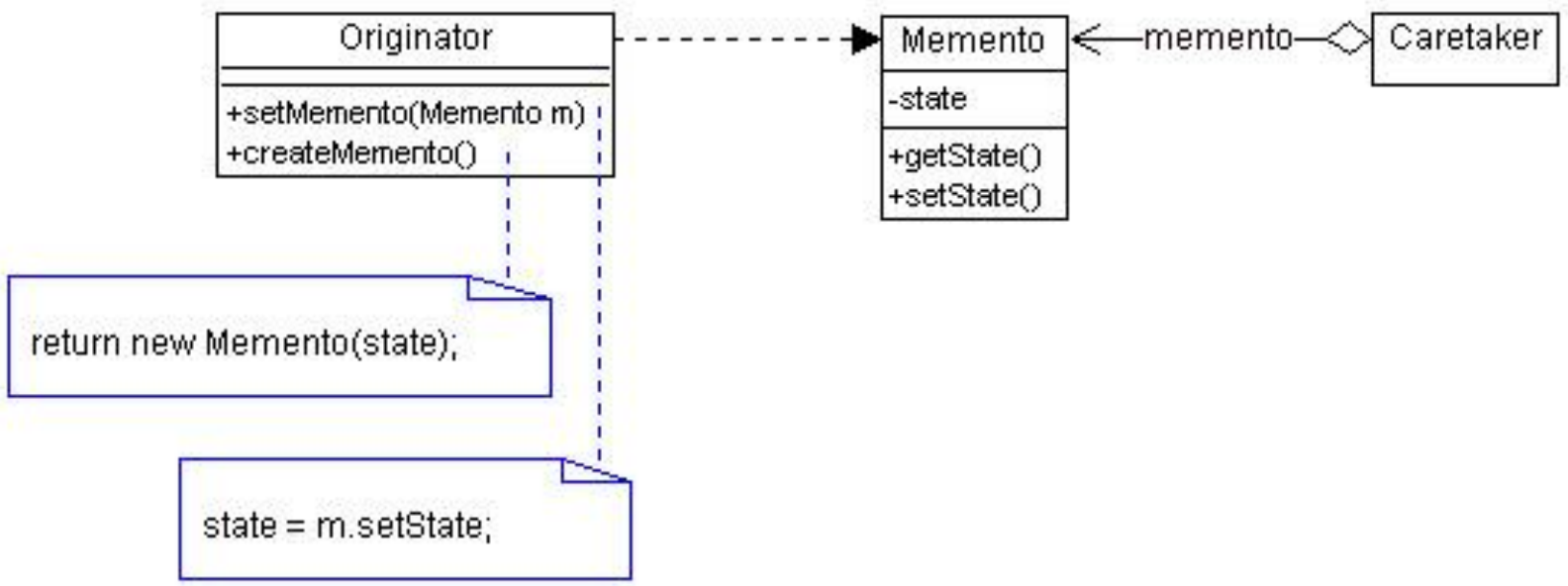
Mediator

- Colaborações
  - Colegas enviam e recebem solicitações do objeto Mediator.
  - O Mediator implementa o comportamento cooperativo pelo redirecionamento das solicitações para os colegas apropriados.
- Conseqüencias
  - Limita o uso de subclasses
  - Desacopla os colegas.
  - Simplifica o protocolo dos objetos
  - Abstrai a maneira como os objetos cooperam
  - Centraliza o controle
- Padrões Correlatos
  - Façade, Observer

Memento

- Intenção
  - Sem violar o encapsulamento, capturar e externalizar o estado interno de um objeto, de forma que este possa ser restaurado mais tarde.
- Motivação
  - Algumas vezes é necessário registrar o estado interno de um objeto (*checkpoints*, *undo*).
- Aplicabilidade
  - Um instantâneo do estado de um objeto deve ser salvo para que possa ser restaurado mais tarde.
  - Uma interface direta para acesso ao estado exporia detalhes de implementação do objeto, violando o encapsulamento.

Memento - Estrutura



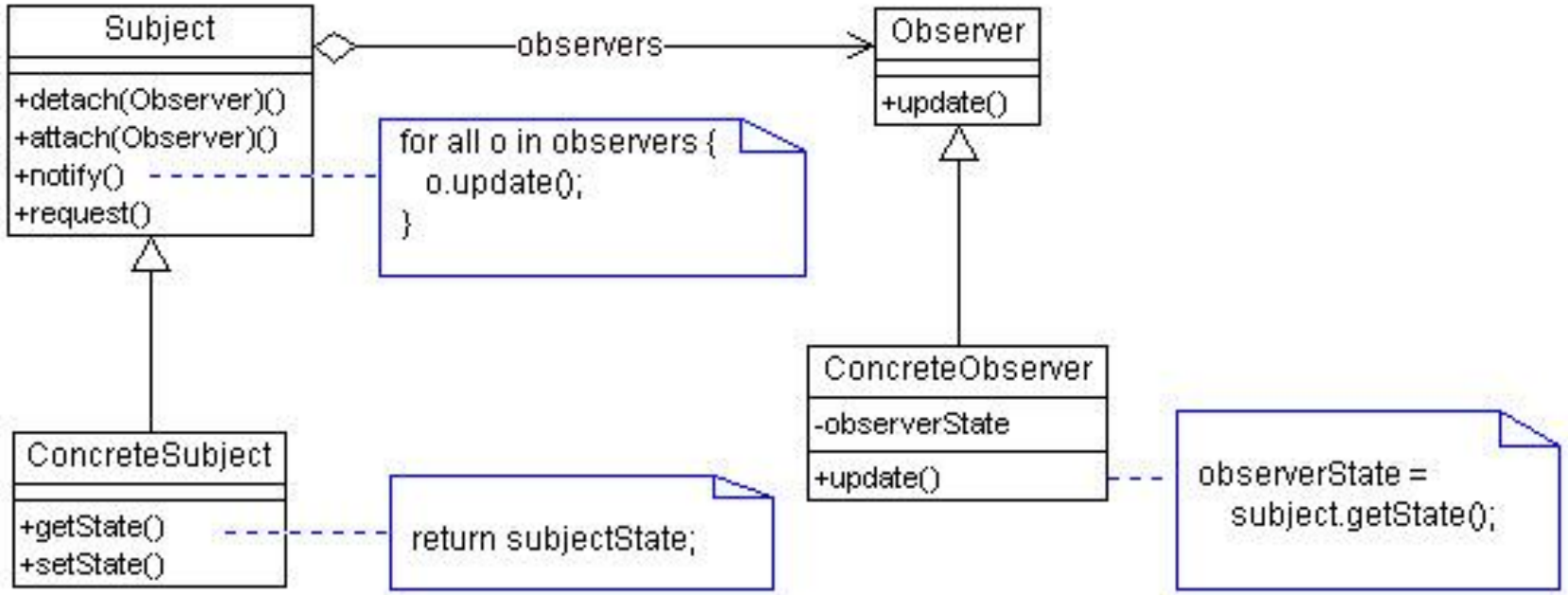
Memento

- Colaborações
  - Um Caretaker (curador) solicita um memento de um originador, mantém o mesmo durante um tempo e quando necessário, o devolve ao originador.
  - Mementos são passivos. Somente o originador que o criou irá atribuir ou recuperar o seu estado.
- Conseqüências
  - Preserva o encapsulamento.
  - Simplifica o originador.
  - Pode ser computacionalmente caro.
  - Interfaces podem ser *estreitas* ou *largas*.
  - Custos ocultos na custódia dos mementos.
- Padrões Correlatos
  - Command, Iterator

Observer

- 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 particiopar 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.
- Aplicabilidade: em situações tais que
  - Uma abstração pode ser ‘apresentada’ de várias formas.
  - A mudança de estado de um objeto ímplica em mudanças em outros objetos.
  - Um objeto deve ser capaz de notificar a outros objetos, sem necesáriamente saber quem são esses objetos.

Observer - Estrutura



Observer

- Colaborações
  - O ConcreteSubject 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 ConcreteSubject um objeto Observer pode consultá-lo para obter as informações necessárias para atualizar o seu estado.

Observer

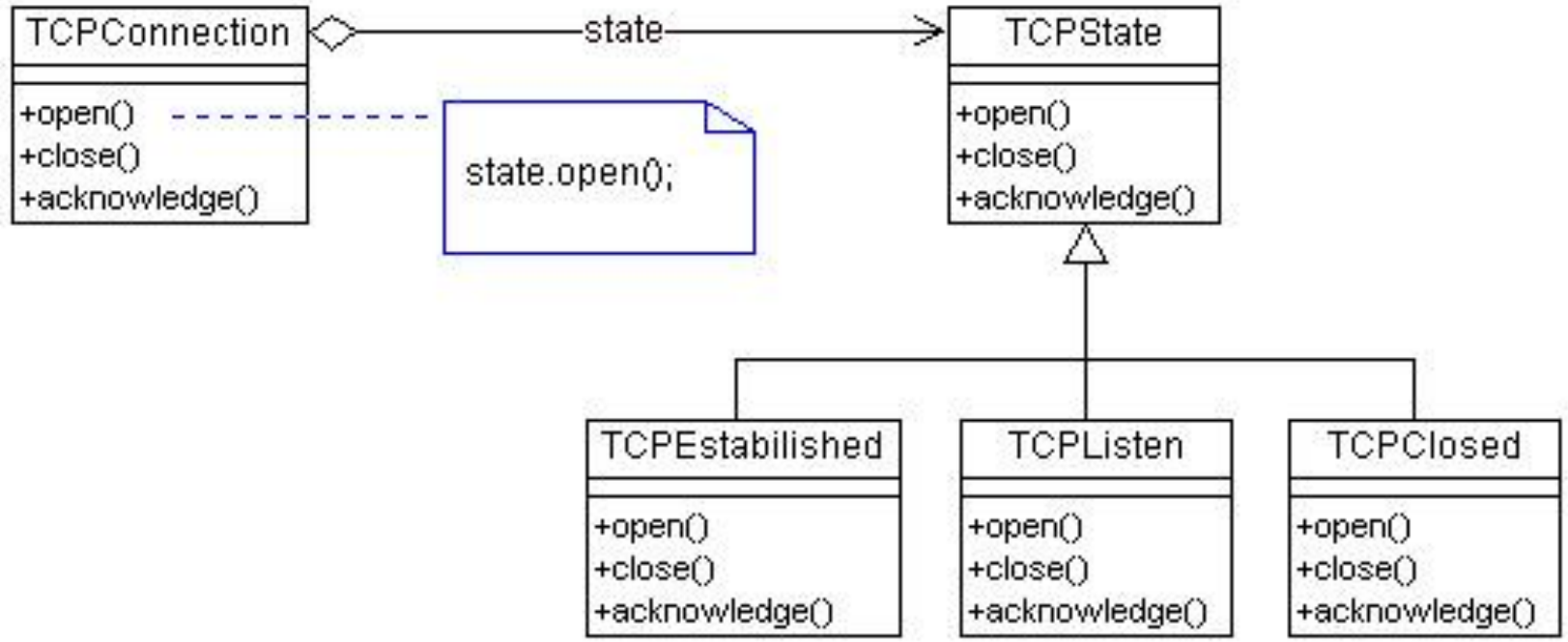
- Conseqüências
  - Acoplamento abstrato entre *Subject* e *Observer.*
  - Suporte a *broadcast*.
  - Atualizações inesperadas.

- Padrões Correlatos
  - Mediator
  - Singleton

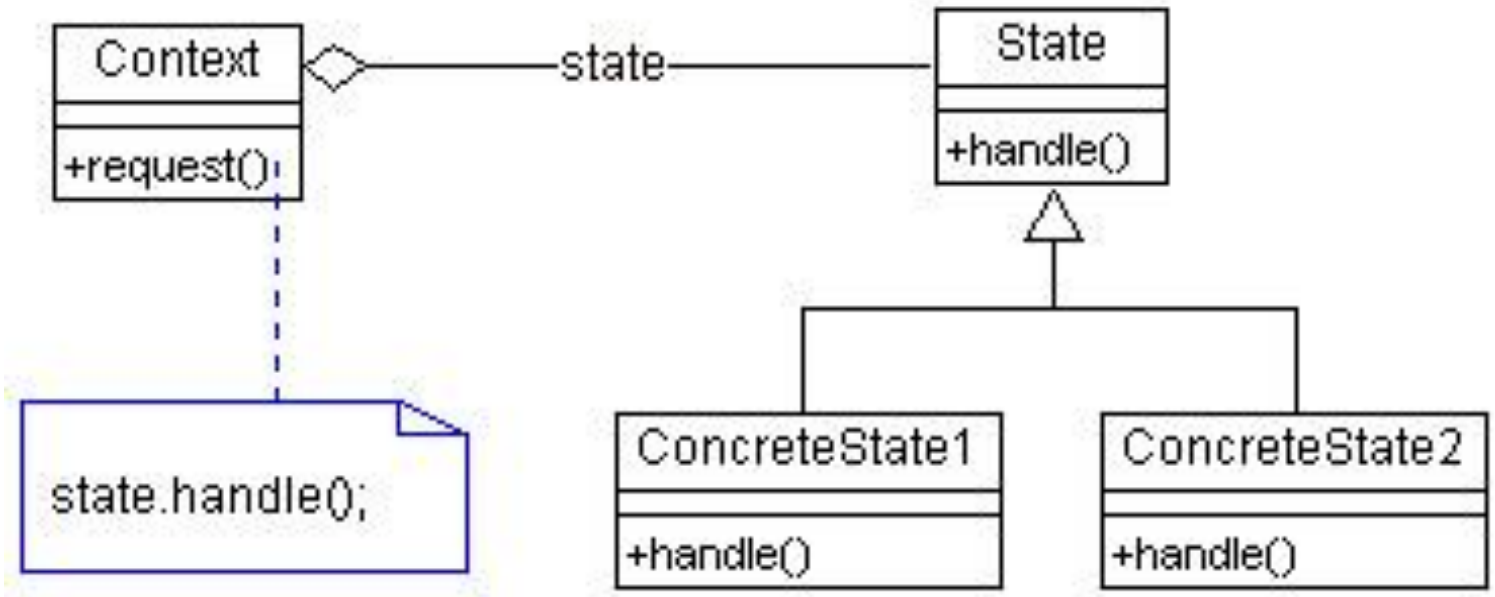
State

- Intenção
  - Permite a um objeto alterar o seu comportamento em função de alterações no seu estado interno.
- Motivação
  - Em muitas situações o comportamento de um objeto deve mudar em função de alterações no seu estado.
- Aplicabilidade
  - O comportamento de um objeto depende do seu estado e pode mudar em tempo de execução.
  - Operações têm comandos condicionais grandes, com várias alternativas que dependem do estado do objeto.

State – um exemplo



State – Estrutura



State

- Colaborações
  - O objeto Context delega solicitações específicas de estados para o objeto ConcreteState corrente.
  - Um objeto Context pode passar uma referência a si próprio como um argumento para o objeto State acessar o seu contexto, se necessário.
  - Context é a interface primária para os clientes. Clientes não necessitam tratar os objetos State diretamente.
  - Tanto Context como as subclasses ConcreteState podem decidir a seqüência de estados.

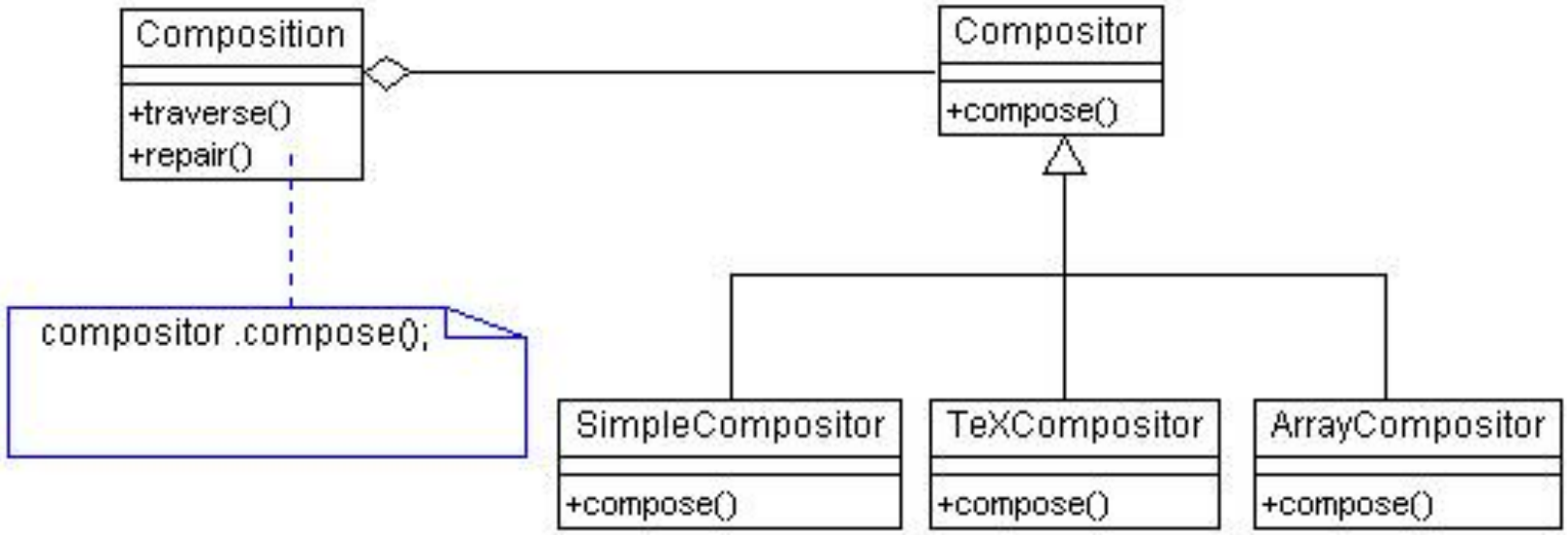
State

- Conseqüências
  - Confina comportamento específico de estados e particiona o comportamento específico para estados diferentes.
  - Torna explícitas as transições de estado.
  - Objetos State podem ser compartilhados, se não possuirem variáveis de instância. Nesse caso eles acabam implementando o padrão Flyweight sem estado intrínseco.
- Padrões Relacionados
  - Flyweight, Singleton

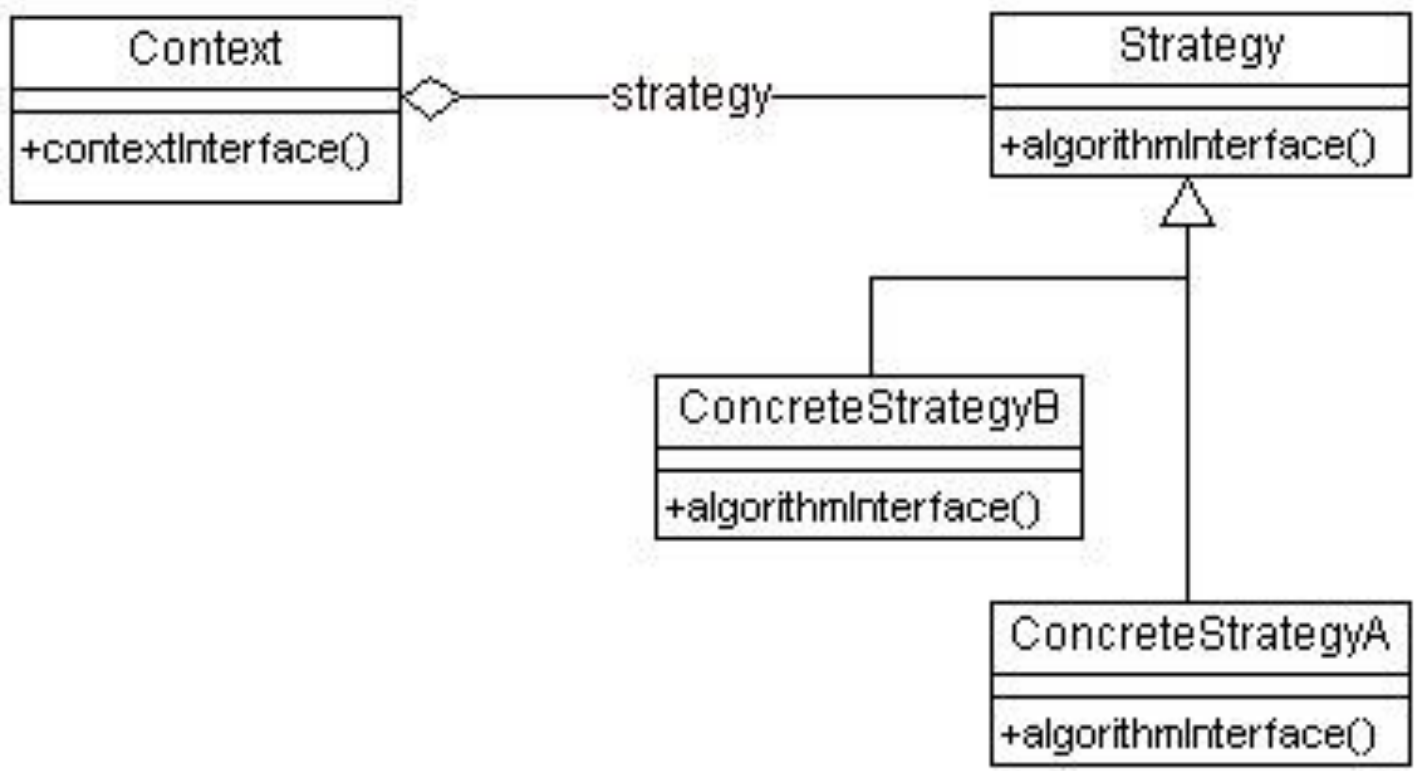
Strategy

- 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
  - Numa 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.

Strategy - exemplo



Strategy - Estrutura



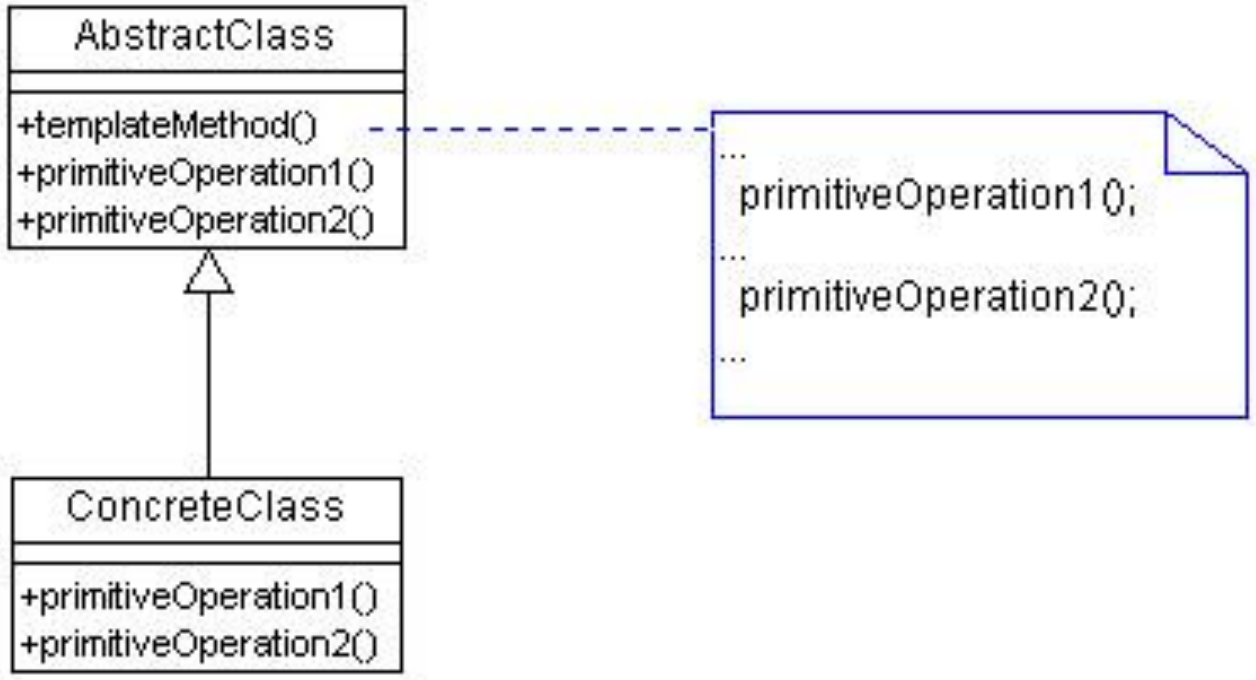
Strategy

- Colaborações
  - Strategy e Context interagem para implementar o algoritmo escolhido.
  - Context repassa solicitações dos seus clientes para a estratégia. Os clientes em geral passam um objeto ConcreteStrategy para o contexto.
- Conseqüências
  - Famílias de algoritmos relacionados
  - Alternativa para o uso de subclasses
  - Eliminam comandos condicionais ao se codificar os métodos
- Padrões Correlatos
  - Flyweight

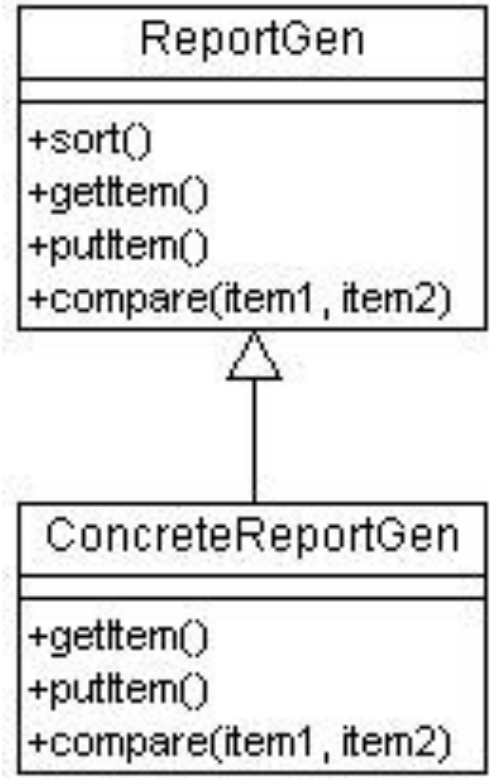
Template Method

- 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 / Aplicabilidade
  - 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.

Template Method - Estrutura



Template Method - exemplo



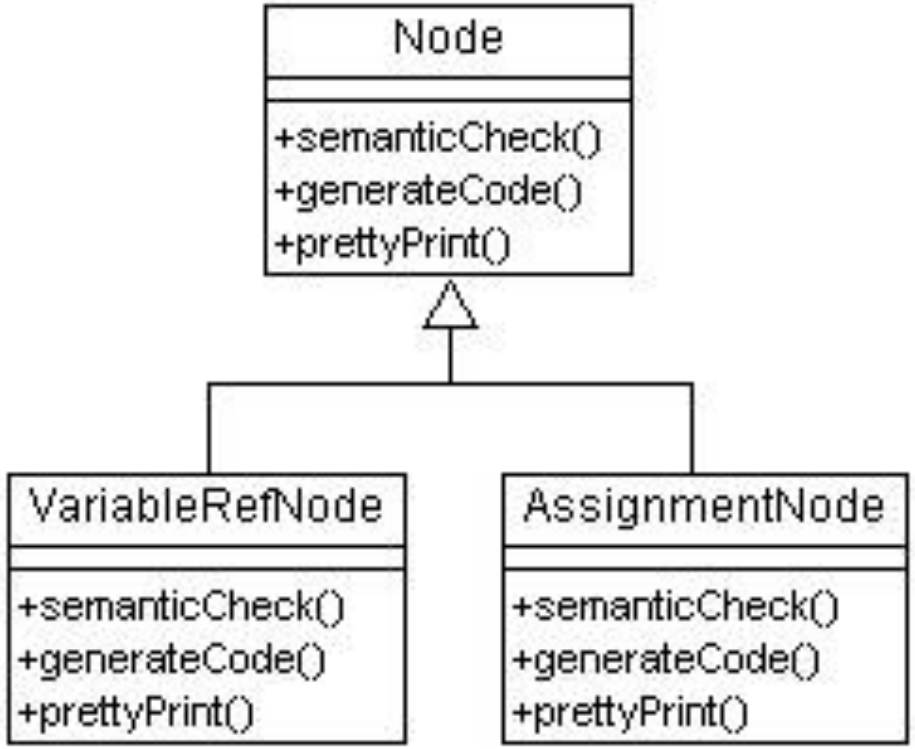
Template Method

- Colaborações
  - ConcreteClass depende de AbstractClass para implementar os passos invariantes do algoritmo.
- Conseqüências
  - Técnica fundamental para reuso de código.
  - “Princípio de Hollywod”: não nos chame, nós chamaremos você.
- Padrões Correlatos
  - FactoryMethods, Strategy

Visitor

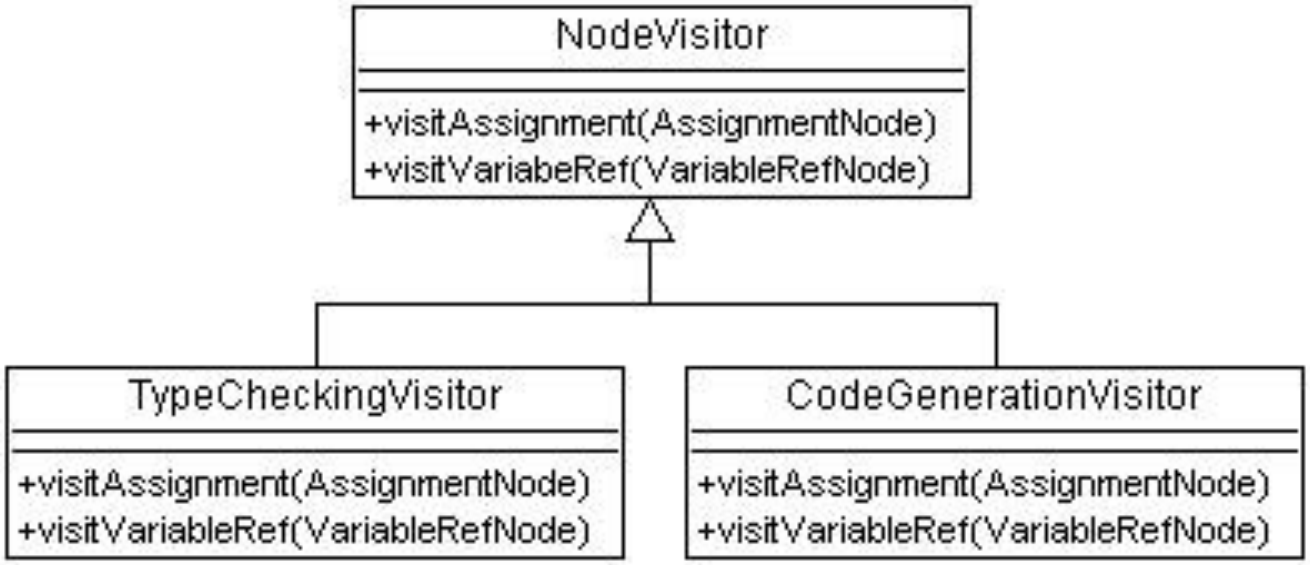
- Intenção
  - Representar uma operação a ser implementada nos elementos de um agregado de objetos. Visitor permite implementar uma nova operação sem mudar as classes dos elementos que constituem os agregados.
- Motivação / Aplicabilidade
  - Em algumas situações é necessário percorrer um agregado de objetos realizando operações sobre seus elementos e dependendo do caso, o conjunto de operações independe das classes dos objetos que constituem o agregado.

Visitor - exemplo

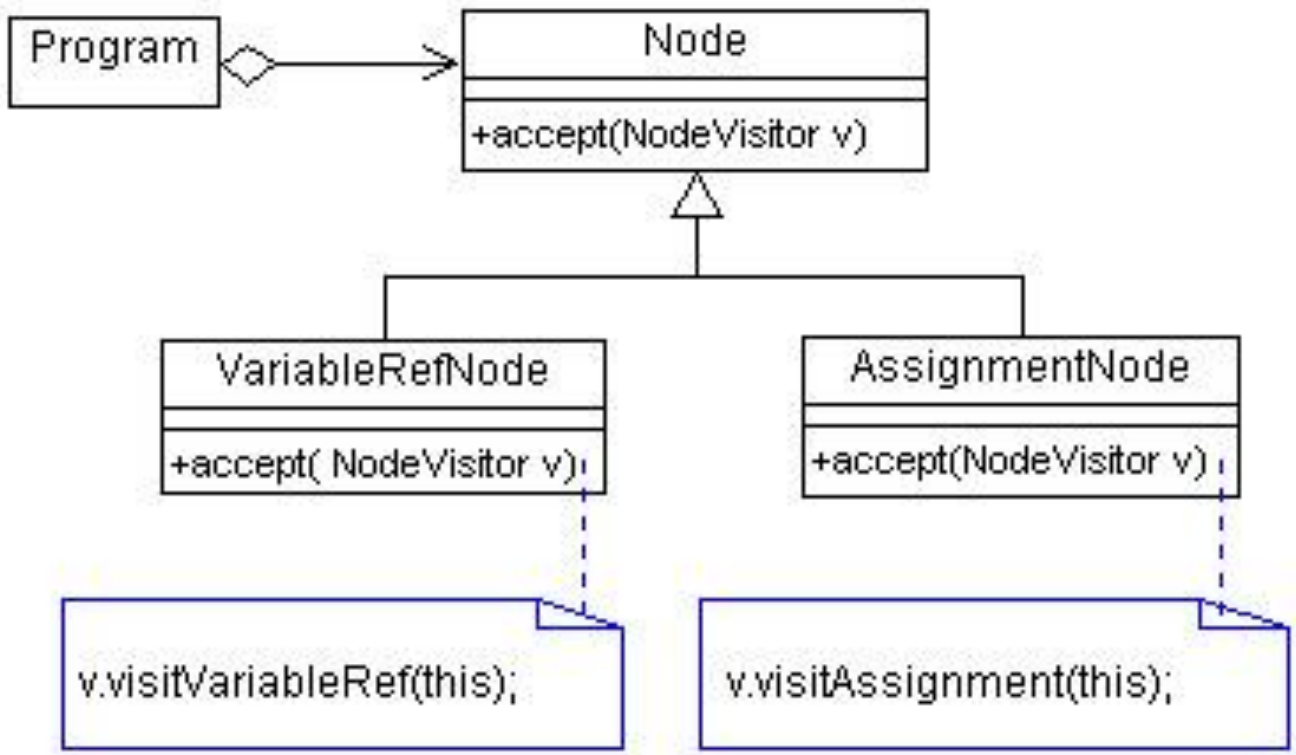


Se for necessário gerar código para outros processadores, será necessário um método generateCode() para cada um deles.

Visitor – Estrutura(1)



Visitor – Estrutura(2)



Visitor

- Colaborações
  - Um cliente que usa o padrão Visitor deve criar um objeto ConcreteVisitor e percorrer a esrutura do objeto, visitando cada elemento através desse Visitor.
- Conseqüências
  - 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 Correlatos
  - Composite, Interpreter

Padrões de Projeto Conclusões

- Vocabulário comum de projeto
  - Para comunicar, documentar e explorar alternativas de projeto.
  - Tornam um sistema menos complexo ao permitir que se fale sobre ele num nível mais alto de abstração.
  - Elevam o nível no qual se projeta e se discute o projeto.
- Auxílio para a documentação e aprendizado
  - Facilita a compreensão de sistemas existentes.
  - Ao fornecer soluções para problemas comuns, acelera o processo de aprendizado.

Padrões de Projeto Conclusões

- Complementam o processo de desenvolvimento
  - Mostram como usar técnicas básicas e também como parametrizar um sistema.
  - Auxiliam a evoluir do modelo de análise para o modelo de implementação.
- Apoio na refatoração
  - Ajudam a definir como reorganizar um projeto.
  - Reduzem o esforço de uma futura refatoração do projeto.