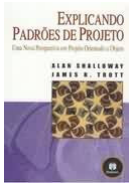
# Padrões de Projeto (*Design Patterns*)

Marco Antônio Pereira Araújo, M.Sc.
maraujo@acessa.com

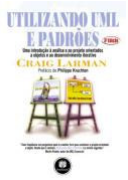
## Agenda

- Conceituação
- Estrutura
- Padrão MVC
- Padrões de Criação
- Padrões Estruturais
- Padrões Comportamentais
- Exemplos de Aplicação

# Bibliografia Específica


- GAMMA, Erich, et al. Padrões de Projeto – Soluções Reutilizáveis de Software Orientado a Objetos. Bookman, 2005.
- LARMAN, Craig. Utilizando UML e Padrões. 3ª. ed. Bookman, 2007.
- SHALLOWAY, Allan; TROTT, James. Explicando Padrões de Projeto. Bookman, 2004.
- FREEMAN, Eric. Use a Cabeça! Padrões de Projeto. Alta Books, 2005.







# Padrões de Projeto

- "Cada Pattern descreve um problema que ocorre várias e várias vezes no seu ambiente, e então descreve o núcleo de solução para aquele problema, num caminho em que você possa usar esta solução um milhão de vezes mais, sem ter que fazer o mesmo caminho duas vezes"
- "Patterns são uma forma de capturar a experiência de projeto de forma que outras pessoas possam usar efetivamente"

# Padrões de Projeto

- Ao desenvolver sistemas, sempre encontramos soluções semelhantes para problemas semelhantes...
- Padrões de projeto são um conjunto de idéias que solucionam problemas tentando fazer um projeto OO ficar mais flexível e reutilizável
- Não reutilizamos código, mas a idéia que está por trás da solução de um problema



# Padrões de Projeto
## Porque usar?

- Formam um vocabulário comum que permite uma melhor comunicação entre os desenvolvedores, uma documentação mais completa e uma melhor exploração das alternativas de projeto;
- Funcionam como peças na construção de projetos de software mais complexos; podem ser considerados micro-arquiteturas que contribuem para a arquitetura geral do sistema;
- Reduzem o tempo de aprendizado de uma determinada biblioteca de classes;
- Quanto mais cedo são usados, menor será o re-trabalho em etapas mais avançadas do projeto.

# Padrões de Projeto
# Origens

- Os *Design Patterns* originam-se no final dos anos 80 quando Ward Cunningham e Kent Beck desenvolveram um conjunto de padrões para serem aplicados no desenvolvimento de interfaces do usuário elegantes em Smalltalk
- No mesmo período, Jim Coplien estava desenvolvendo um catálogo de padrões C++ chamados idiomas
- Enquanto isso, Erich Gamma estava trabalhando em sua tese de doutorado sobre desenvolvimento de software orientado a objeto, e reconheceu a importância de acumular explicitamente as estruturas de projetos que se repetiam com frequência
- Viraram “moda” quando a "Gang of Four" (GoF) publicou um catálogo contendo 23 padrões, em 1995

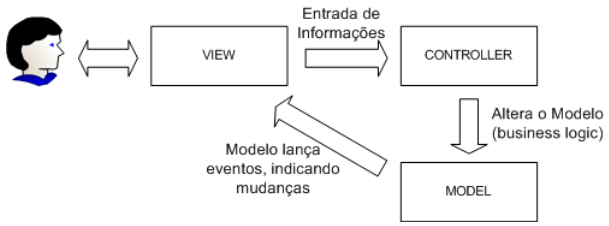


# Padrões de Projeto
# Estrutura

- Um padrão de projeto é composto basicamente de quatro elementos:
  - O nome do padrão
    - Usado na comunicação entre os projetistas OO;
  - O problema
    - Descreve quando aplicar o padrão, explica o problema e seu contexto;
  - A solução
    - Descreve os elementos que compõem o projeto, seus relacionamentos, responsabilidades e colaborações. É um gabarito que deve ser usado em problemas semelhantes
  - As conseqüências
    - Resultados e análises das vantagens e desvantagens da aplicação do padrão. Mostra os custos e benefícios da aplicação de um padrão.

# Padrão MVC

- Modelo *Model-View-Controller*
- Modelo clássico de desenvolvimento OO de aplicações






# Padrão MVC

- Principais características:
  - As classes de interface conhece apenas as classes de controle
  - As classes de controle conhecem apenas as classes de domínio
  - As classes de domínio não devem conhecer nada sobre as classes de interface e controle

# Padrão MVC

- Principais vantagens:
  - A classe de domínio é independente da interface do sistema
  - A interface pode ser alterada sem afetar a classe de domínio
  - A classe de domínio pode ser reutilizada em outras aplicações
  - A seqüência de operações é encapsulada na classe de controle
  - As dependências entre as classes do sistema são reduzidas
  - As classes de domínio informam mudanças de seu estado para as classes de interface através do padrão Observer



# Padrões de Projeto
# Padrões de Criação

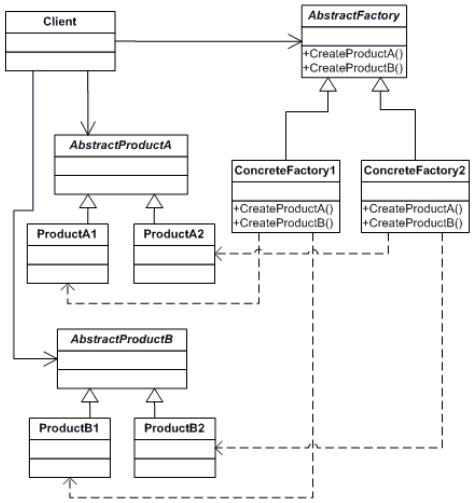
- Abstraem o processo de instanciação, ajudam a tornar um sistema independente de como seus objetos são criados
  - Abstract Factory
  - Builder
  - Factory Method
  - Prototype
  - Singleton

# Padrão Abstract Factory

- Objetivo
  - Fornecer uma interface para criação de famílias de objetos relacionados ou dependentes sem especificar suas classes concretas
- Motivação
  - Um sistema deve ser independente de como seus produtos são criados, compostos ou representados
  - Um sistema deve ser configurado como um produto de uma família de múltiplos produtos
  - Uma família de objetos-produto for projetada para ser usada em conjunto, e necessita-se garantir esta restrição
  - Construção de interfaces de usuários que suporte múltiplos estilos de interação, por exemplo



# Padrão Abstract Factory

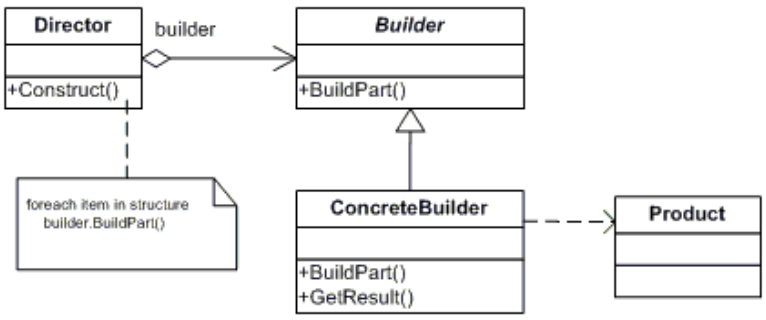


# Padrão Builder

- Objetivo
  - Separar a construção de um objeto complexo da sua representação de modo que o mesmo processo de construção possa criar diferentes representações
- Motivação
  - O algoritmo para a criação de um objeto complexo deve ser independente das partes que compõem o objeto e de como elas são montadas
  - O processo de construção deve permitir diferentes representações para o objeto que é construído
  - Converter um arquivo RTF em diferentes formatos é um exemplo



# Padrão Builder

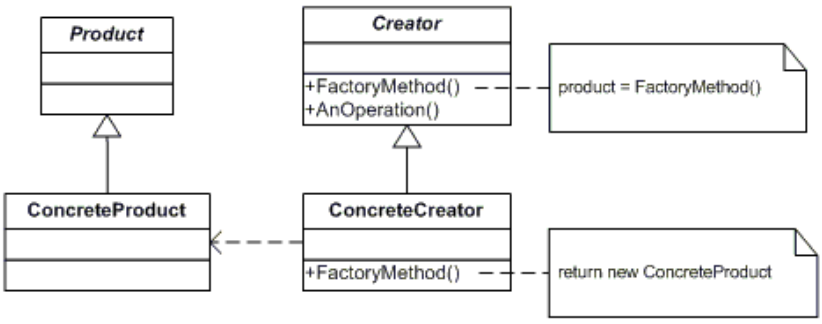


# Padrão Factory Method

- Objetivo
  - Definir uma interface para criar um objeto, mas deixar as subclasses decidirem que classe instanciar
  - Permite adiar a instanciação para subclasses
- Motivação
  - Uma classe não pode antecipar a classe de objetos que deve criar
  - Uma classe quer que suas subclasses especifiquem os objetos que criam



# Padrão Factory Method

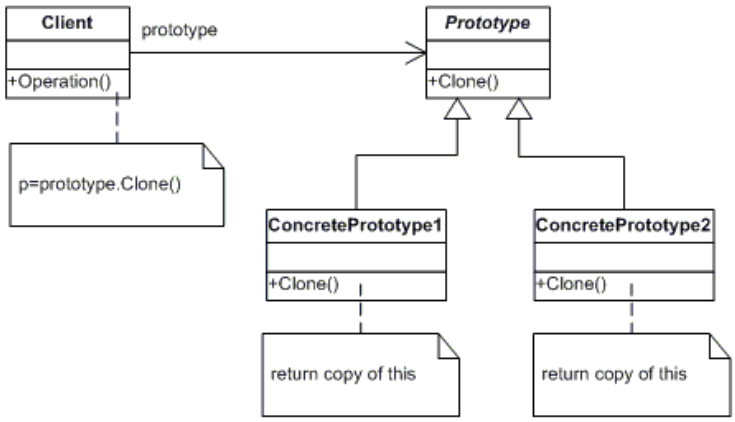


# Padrão Prototype

- Objetivo
  - Especificar os tipos de objetos a serem criados usando uma instância protótipo e criar novos objetos pela cópia deste protótipo
- Motivação
  - Quando as classes a instanciar são especificadas em tempo de execução
  - Evitar a construção de uma hierarquia de classes de fábricas paralela à hierarquia de classes do produto
  - Quando as instâncias de uma classe puderem ter uma dentre poucas combinações diferentes de estado. Pode ser mais conveniente instalar um número correspondente de protótipos e cloná-los ao invés de instanciar a classe manualmente, cada vez com um estado apropriado
  - A construção de um editor de partituras musicais customizando um *framework* geral para editores gráficos, acrescentando novos objetos que representam notas, pausas e pentagramas pode ser um exemplo



# Padrão Prototype

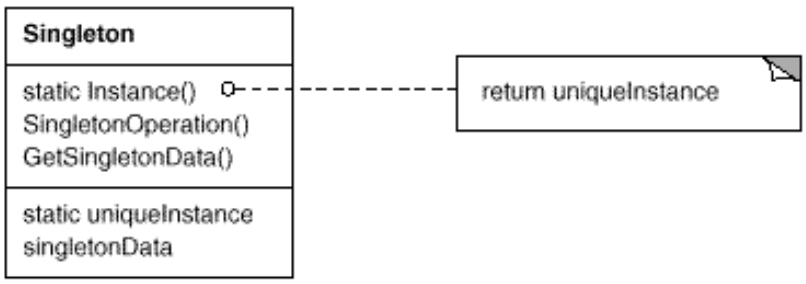


# Padrão Singleton

- Objetivo
  - Garantir que uma classe tenha somente uma instância e fornecer um ponto global de acesso para a mesma;
- Motivação
  - É importante manter apenas uma instância para determinadas classes, como exemplo: acesso ao sistema de arquivos; spool de impressão; um objeto de configuração de um sistema, etc.



# Padrão Singleton



# Padrões de Projeto
# Padrões Estruturais

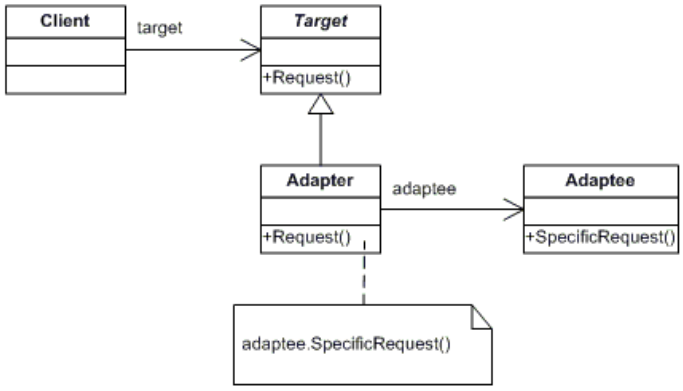
- Preocupam-se com a forma como as classes e objetos são compostos para formar estruturas maiores
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Façade
  - Flyweight
  - Proxy



# Padrão Adapter

- Objetivo
  - Converter a interface de uma classe em outra interface, esperada pelos clientes
  - Permite que classes com interfaces incompatíveis trabalhem em conjunto
- Motivação
  - Algumas vezes, classes projetadas para serem reutilizadas não o são por não corresponderem à interface específica requerida por uma aplicação

# Padrão Adapter

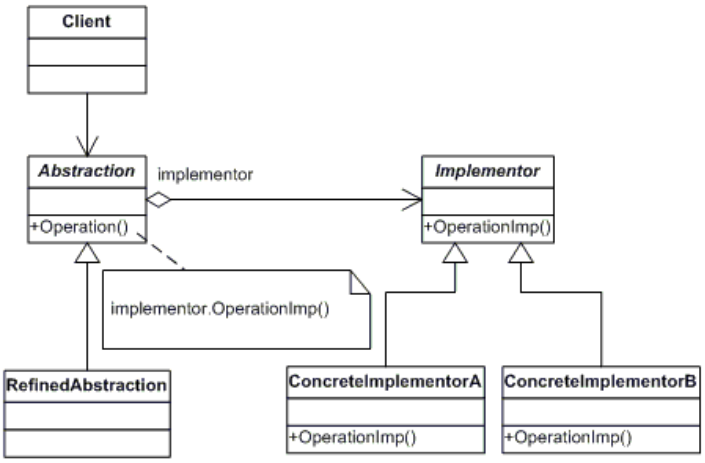





# Padrão Bridge

- Objetivo
  - Desacoplar uma abstração de sua implementação, de modo que as duas possam variar independentemente
- Motivação
  - Quando uma abstração pode ter uma entre várias implementações possíveis, a maneira usual de acomodá-las é a herança
  - Esta abordagem nem sempre é suficientemente flexível
  - A herança liga uma implementação à abstração permanentemente, o que torna difícil a modificação, aumento e reutilização de abstração e implementações independentemente

# Padrão Bridge

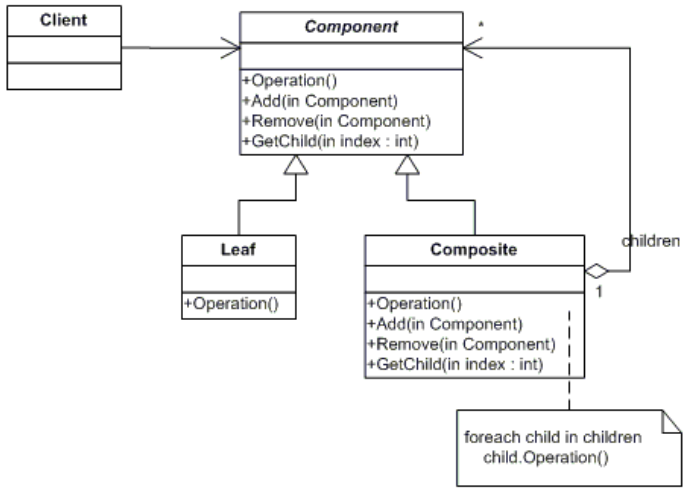





# Padrão Composite

- Objetivo
  - Compor objetos em estruturas de árvores para representarem estruturas todo-parte
  - Permite aos clientes tratarem de maneira uniforme objetos individuais e composições
- Motivação
  - Aplicações gráficas costumam permitir a construção de diagramas e interfaces complexas a partir de componentes simples
  - Componentes simples podem ser agrupados para formar componentes maiores

# Padrão Composite

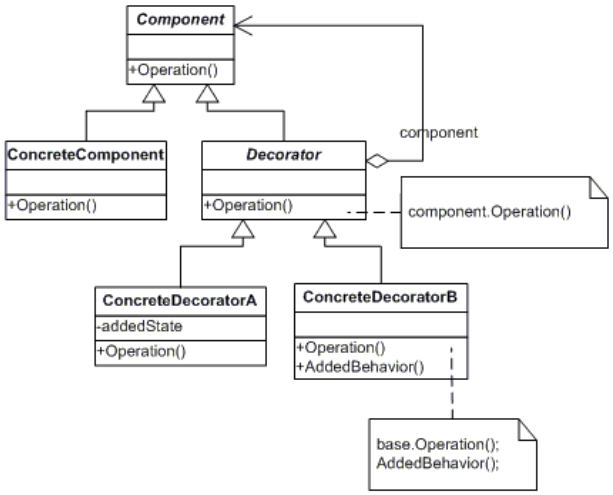





# Padrão Decorator

- Objetivo
  - Agregar dinamicamente responsabilidades adicionais a um objeto
  - Fornecem uma alternativa flexível ao uso de subclasses para extensão de funcionalidades
- Motivação
  - Acrescentar responsabilidades a objetos individuais, não à classe
  - Ao invés de herdar, pode-se embutir o objeto em outro (o decorator) que adicione a responsabilidade

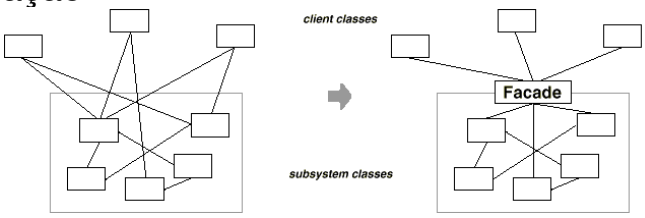
# Padrão Decorator



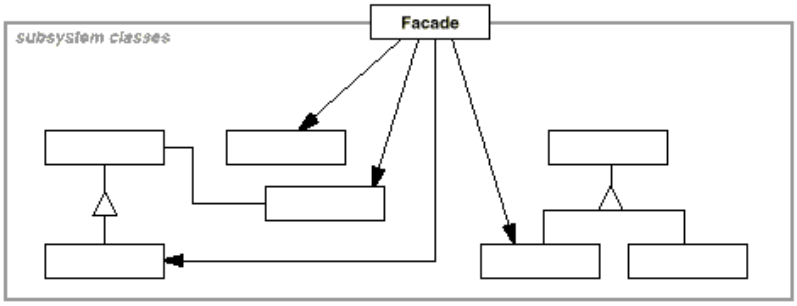



# Padrão Façade

- Objetivo
  - Fornecer uma interface unificada para um conjunto de interfaces em um subsistema. Esta interface torna o subsistema mais fácil de ser usado
- Motivação



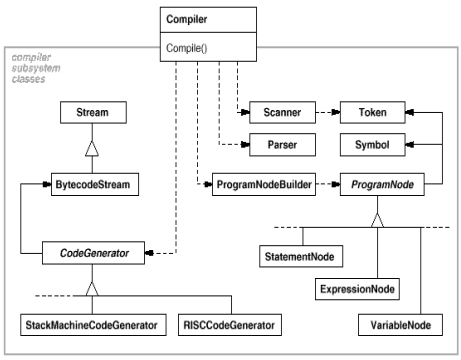
# Padrão Façade






# Padrão Façade

- Em um compilador, temos inúmeras etapas de compilação: Análise sintática, semântica, geração de código, etc.
- No entanto, os sistemas usuários de um compilador estão interessados apenas em compilar o código, independente de como isso é feito.

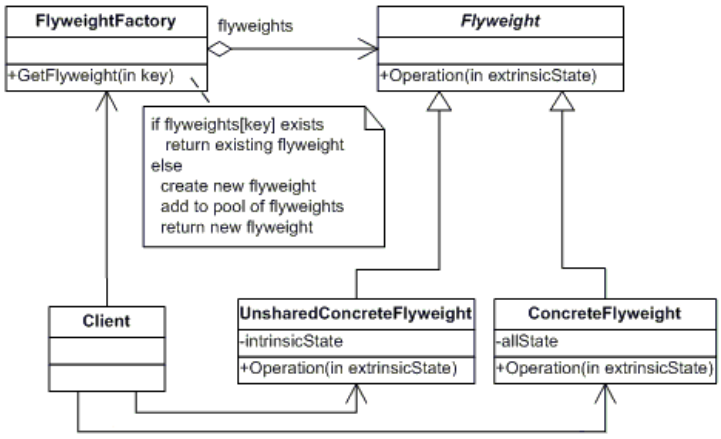


# Padrão Flyweight

- Objetivo
  - Usar compartilhamento para suportar de forma eficiente grandes quantidades de objetos de granularidade fina
- Motivação
  - Representa objetos independentes, não importando quantas vezes sejam referenciados
  - Outros objetos são responsáveis por indicar a seqüência destes objetos



# Padrão Flyweight

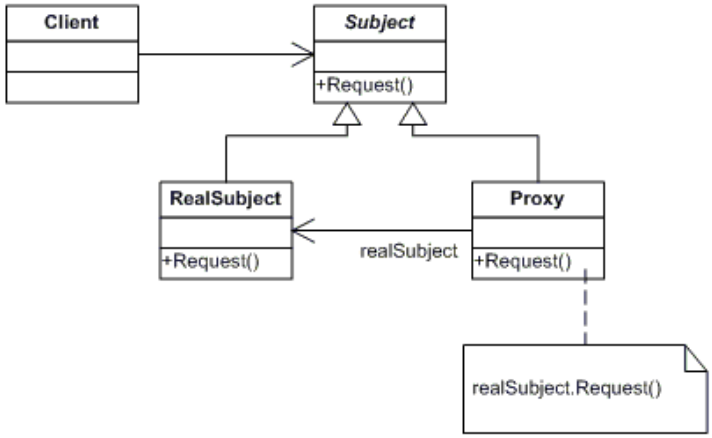


# Padrão Proxy

- Objetivo
  - Fornece um substituto ou marcador da localização de outro objeto para controlar o acesso ao mesmo
- Motivação
  - Uma razão para controlar o acesso a um objeto é adiar o custo integral de sua criação e inicialização até o momento de usá-lo
  - Um editor de documentos pode embutir objetos gráficos, que podem ser caros para serem criados
  - A abertura de documentos deveria ser rápida, evitando a criação, de uma só vez, de todos os objetos caros



# Padrão Proxy



# Padrões de Projeto
# Padrões Comportamentais

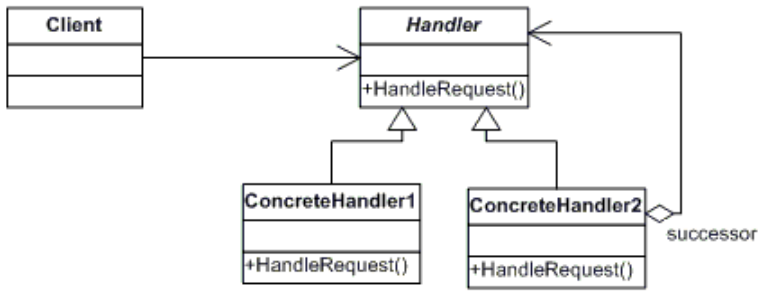
- Preocupam-se com algoritmos e a atribuição de responsabilidades entre objetos. Descrevem padrões de comunicação entre os objetos
  - Chain of Responsability
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Template Method
  - Visitor



# Padrão Chain of Responsability

- Objetivo
  - Evitar o acoplamento do remetente de uma solicitação ao seu receptor, ao dar a mais de um objeto a oportunidade de tratar a solicitação
  - Encadear os objetos receptores passando a solicitação ao longo da cadeia, até que um objeto a trate
- Motivação
  - Um sistema de help sensitivo ao contexto, onde a ajuda é fornecida dependendo da parte da interface que é selecionada e do seu contexto
  - Natural organizar do mais específico para o mais genérico: se não existe uma ajuda específica, exibir a caixa de ajuda geral
  - O objeto que solicita a ajuda não conhece o objeto que fornecerá a ajuda

# Padrão Chain of Responsability

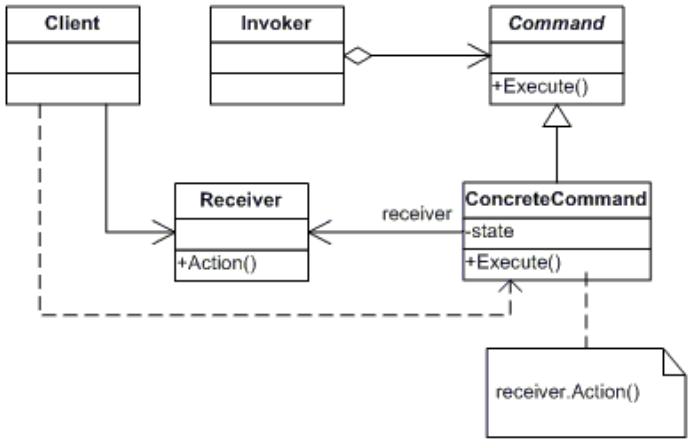





# Padrão Command

- Objetivo
  - Encapsular uma solicitação como um objeto, permitindo parametrizar clientes com diferentes solicitações, enfileirar ou fazer o registro (log) de solicitações e suportar operações que podem ser desfeitas
- Motivação
  - Desacopla o objeto que invoca a operação daquele que tem o conhecimento para executá-la
  - A construção de um componente genérico de menu pode utilizar este padrão

# Padrão Command

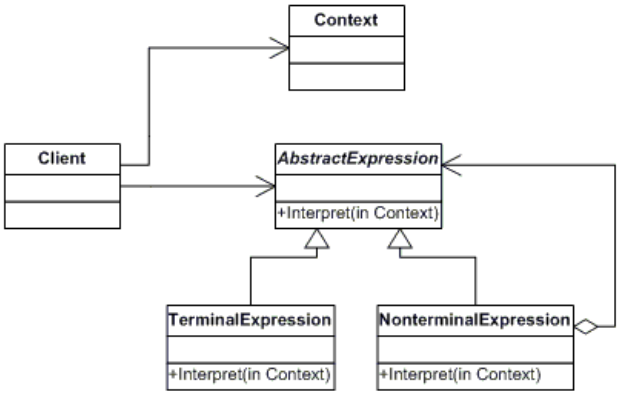





# Padrão Interpreter

- Objetivo
  - Dada uma linguagem, definir uma representação para a sua gramática juntamente com um interpretador que usa representação para interpretar sentenças da linguagem
- Motivação
  - Calcular o resultado de expressões boleanas é um exemplo de utilização deste padrão

# Padrão Interpreter

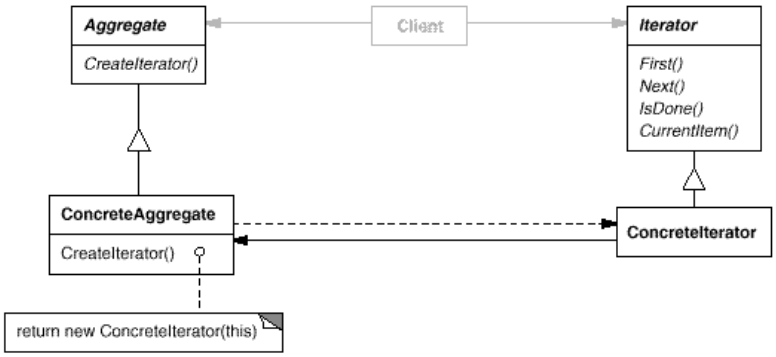





# Padrão Iterator

- Objetivo
  - Prover uma forma de, seqüencialmente, acessar os elementos de uma coleção sem expor sua representação interna
- Motivação
  - Queremos navegar em uma estrutura de dados independente de sua representação interna, de modo a poder mudá-la sem afetar quem a usa

# Padrão Iterator

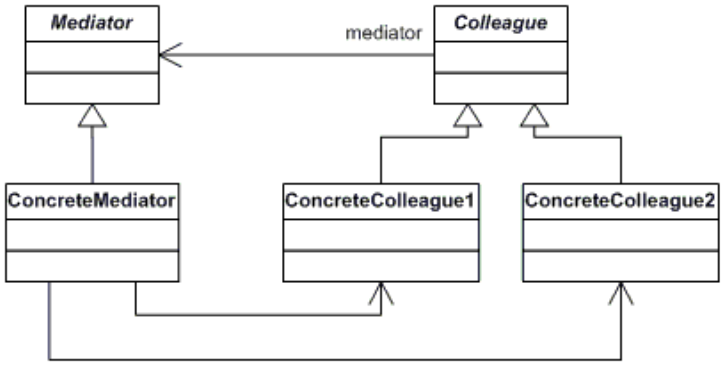





# Padrão Mediator

- Objetivo
  - Definir um objeto que encapsula a forma como um conjunto de objetos interage
  - Promove o acoplamento fraco ao evitar que os objetos se refiram uns aos outros explicitamente e permite variar suas interações independentemente
- Motivação
  - Em caixas de diálogo, as dependências entre os componentes podem ser delegadas a um mediador
  - Os componentes conhecem apenas o mediador, que é responsável pelo controle e coordenação das interações de um grupo de objetos

# Padrão Mediator

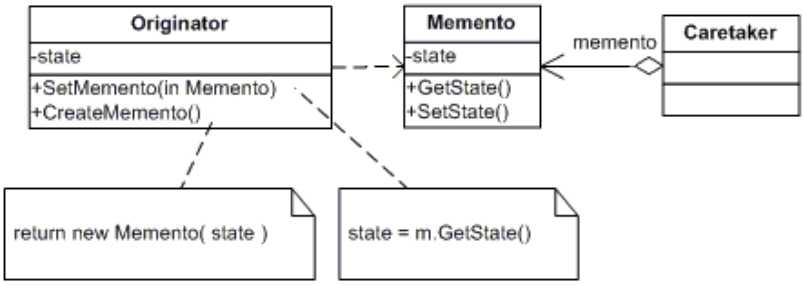





# Padrão Memento

- Objetivo
  - Sem violar o encapsulamento, capturar e externalizar um estado interno de um objeto, de maneira que este possa ser restaurado para este estado mais tarde
- Motivação
  - Mecanismos de *checkpoint* e *undo*

# Padrão Memento

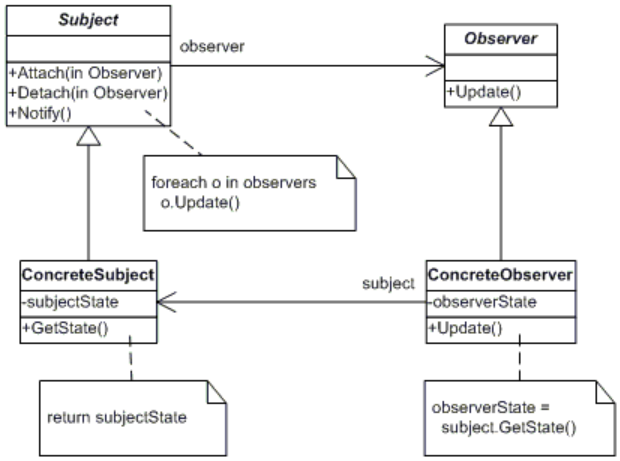





# Padrão Observer

- Objetivo
  - Definir uma dependência um-para-muitos entre objetos, de maneira que quando um objeto muda seu estado, todos os seus dependentes são notificados e a atualizados automaticamente
- Motivação
  - Manter a consistência entre objetos relacionados, sem torná-los fortemente acoplados
  - Construção de interfaces de apresentação de dados separadas da manipulação dos dados da aplicação, podendo ser reutilizadas de forma independente

# Padrão Observer

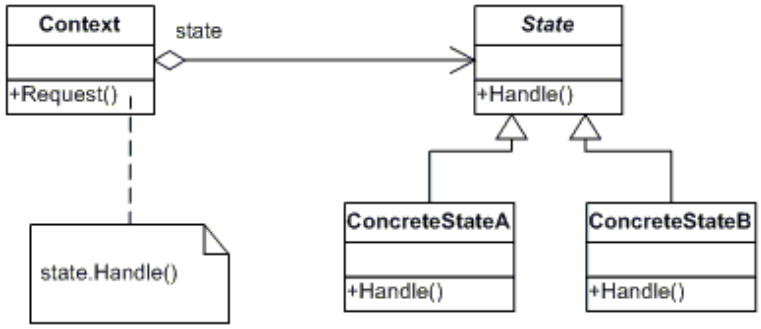





# Padrão State

- Objetivo
  - Permite ao objeto alterar seu comportamento quando o seu estado interno muda. O objeto parecerá ter mudado de classe
- Motivação
  - O comportamento de um objeto depende do seu estado e ele pode mudar seu comportamento em tempo de execução, dependendo desse estado

# Padrão State

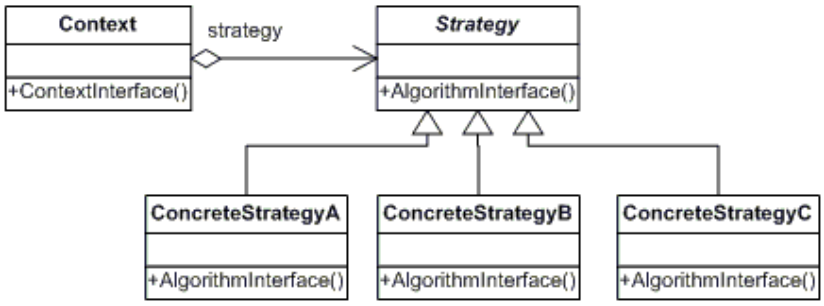





# Padrão Strategy

- Objetivo
  - Definir uma família de algoritmos, encapsular cada uma delas e torná-las intercambiáveis
  - Permite que o algoritmo varie independentemente dos clientes que o utilizam
- Motivação
  - Necessidade de variantes de um mesmo algoritmo
  - Pode ser utilizado, por exemplo, em validações de campos ou algoritmos de quebras de linha

# Padrão Strategy

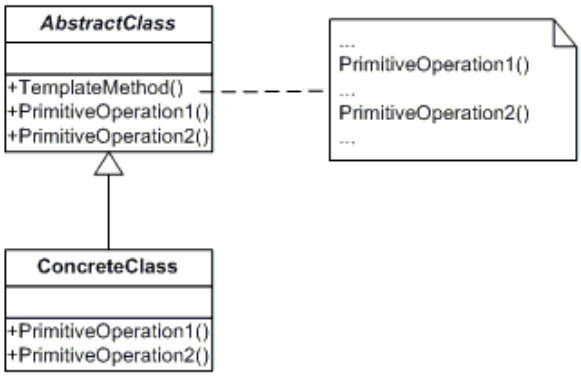





# Padrão Template Method

- Objetivo
  - Definir o esqueleto de um algoritmo em uma operação, postergando alguns passos para subclasses
  - Permite que subclasses redefinam certos passos de um algoritmo sem mudar a estrutura do mesmo
- Motivação
  - Implementar as partes invariantes de um algoritmo uma só vez e deixar para as subclasses a implementação do comportamento que pode variar
  - Muito utilizados em classes abstratas

# Padrão Template Method






# Padrão Visitor

- Objetivo
  - Representar uma operação a ser executada nos elementos de uma estrutura de objetos
  - Permite definir uma nova operação sem mudar as classes dos elementos sobre os quais opera
- Motivação
  - Percorrer os elementos de uma árvore abstrata, criando um caminho para o visitante

# Padrão Visitor