Sistemas Distribuidos e Mobile

Prof. Otaviano SIlvério

GoF Introdução

Arquitetura de Software

O Problema (como já dito)

A boa notícia é que bons projetos existem:

- na verdade, eles apresentam características recorrentes,
- mas eles quase nunca são idênticos.

Perspectiva de engenharia: os projetos podem ser descritos, codificados ou padronizados?

- isto iria diminuir a fase de tentativa e erro.
- sofwares melhores seriam produzidos mais rápidos.

Quantos Padrões de Projeto (PP) existem?

Muitos. Um site relata ao menos 250 PP. E muito mais são criados a cada dia.

PP não são idiomas, algortimos ou componentes.

HISTÓRIA

1977
CHRISTOPHER ALEXANDER

1987
KENT BECK E WARD CUNNINGHAM

1995
ERICH GAMMA, RICHARD HELM, RALPH JOHNSON E JOHN VLISSIDES (GANG OF FOUR)

2005
CRAIG LARMAN

Arquitetura de Software

Qual a relação entre esses padrões?

Para criar um sistema, podem ser necessários vários padrões.

Diferentes designers podem usar diferentes padrões para resolver o mesmo problema.

Geralmente:

- Alguns padrões se 'encaixam' naturalmente
- Um padrão pode levar a outro
- Alguns padrões são similares e alternativos
- Padrões podem ser descobertos e documentados
- Padrões não são métodos ou frameworks

Arquitetura de Software

O que faz um padrão?

Um padrão encerra o conhecimento de uma pessoa muito experiente em um determinado assunto de uma forma que este conhecimento pode ser transmitido para outras pessoas menos experientes.

Outras ciências (p.ex. química) e engenharias possuem catálogos de soluções.

Desde 1995, o desenvolvimento de software passou a ter o seu primeiro catálogo de soluções para projeto de software: o livro GoF.

Arquitetura de Software

Gang of Four

Passamos a ter um vocabulário comum para conversar sobre projetos de software.

Soluções que não tinham nome passam a ter nome.

Ao invés de discutirmos um sistema em termos de pilhas, filas, árvores e listas ligadas, passamos a falar de coisas de muito mais alto nível como Fábricas, Fachadas,

Gang of Four

A maioria dos autores eram entusiastas de Smalltalk, principalmente o Ralph Johnson.

Mas acabaram baseando o livro em C++ para que o impacto junto à comunidade de CC fosse maior. E o impacto foi enorme, o livro vendeu centenas de milhares de cópias.

O Formato de um Padrão

Todo padrão inclui

- Nome
- Problema
- Solução
- Conseqüências / Forças

O Formato de um Padrão no GoF

1. Nome (inclui número da página)
   - um bom nome é essencial para que o padrão caia na boca do povo
2. Objetivo / Intenção
3. *Também Conhecido Como*
4. Motivação
   - um cenário mostrando o problema e a necessidade da solução
5. Aplicabilidade
   - como reconhecer as situações nas quais o padrão é aplicável

O Formato de um Padrão no GoF

6. Estrutura
   - uma representação gráfica da estrutura de classes do padrão
     (usando OMT91) em, às vezes, diagramas de interação (Booch 94)
7. Participantes
   - as classes e objetos que participam e quais são suas responsabilidades
8. Colaborações
   - como os participantes colaboram para exercer as suas responsabilidades

O Formato de um Padrão no GoF

9. Conseqüências
   - vantagens e desvantagens, *trade-offs*
10. Implementação
    - com quais detalhes devemos nos preocupar quando implementamos o padrão
    - aspectos específicos de cada linguagem
11. Exemplo de Código

O Formato de um Padrão no GoF

12. Usos Conhecidos
    - exemplos de sistemas reais de domínios diferentes onde o padrão é utilizado
13. Padrões Relacionados
    - quais outros padrões devem ser usados em conjunto com esse
    - quais padrões são similares a este, quais são as **diferenças**

Tipos de Padrões de Projeto

Categorias de Padrões do GoF

- Padrões de Criação
- Padrões Estruturais
- Padrões Comportamentais

Os 23 Pãdrões do GoF

Criação

- **Singleton**: assegura que somente um objeto de uma determinada classe seja criado em todo o projeto;
- **Abstract Factory**: permite que um cliente crie famílias de objetos sem especificar suas classes concretas;
- **Builder**: encapsular a construção de um produto e permitir que ele seja construído em etapas;
- **Prototype**: permite você criar novas instancias simplesmente copiando instancias existentes;
- **Factory Method**: as subclasses decidem quais classes concretas serão criadas.

Os 23 Pãdrões do GoF

Estrutural

- **Decorator**: envelopa um objeto para fornecer novos comportamentos;
- **Proxy**: envelopa um objeto para controlar o acesso a ele;
- **FlyWeigth**: uma instancia de uma classe pode ser usada para fornecer muitas “instancias virtuais”;
- **Facade**: simplifica a interface de um conjunto de classes;
- **Composite**: Os clientes tratam as coleções de objetos e os objetos individuais de maneira uniforme;
- **Bridge**: permite criar uma ponte para variar não apenas a sua implementação, como também as suas abstrações;
- **Adapter**: envelopa um objeto e fornece a ele uma interface diferente;

Os 23 Pãdrões do GoF

Comportamental

- **Template Method**: As subclasses decidem como implementar os passos de um algoritimo;
- **Visitor**: permite acrescentar novos recursos a um composto de objetos e o encapsulamento não é importante;
- **Command**: encapsula uma solicitação como um objeto;
- **Strategy**: encapsula comportamentos intercambiáveis e usa a delegação para decidir qual deles será usado;
- **Chair of Responsability**: permite dar a mais de um objeto a oportunidade de processar uma solicitação;

Os 23 Pãdrões do GoF

Comportamental

- **Iterator**: fornece uma maneira de acessar seqüencialmente uma coleção de objetos sem expor a sua implementação;
- **Mediator**: centraliza operações complexas de comunicação e controle entre objetos relacionados;
- **Memento**: permite restaurar um objeto a um dos seus estados prévios, por exemplo, quando o usuário seleciona um “desfazer”;
- **Interpreter**: permite construir um intérprete para uma linguagem;
- **State**: encapsula comportamentos baseados em estados e usa a delegação para alternar comportamentos;
- **Observer**: permite notificar outros objetos quando ocorre

Os 23 Pãdrões do GoF

Organização do Catálogo

		Propósito		
		Criação	**Estrutural**	**Comportamental**
Escopo	**Classe**	Factory Method (121)	Adapter (157)	Interpreter (274) Template Method (360)
	Objeto	Abstract Factory (99) Builder (110) Prototype (133) Singleton (144)	Adapter (157) Bridge (171) Composite (183) Decorator (196) Facade (208) Flyweight (218) Proxy (233)	Chain of Responsibility (251) Command (263) Iterator (289) Mediator (305) Memento (316) Observer (326) State (338) Strategy (349) Visitor (366)

Outros Padrões

- Há vários catálogos de padrões em software
  - Muitos são específicos a uma determinada área (padrões J2EE, padrões de implementação em Java, em C#, padrões para concorrência, sistemas distribuídos, etc.)
  - Os padrões apresentados aqui são aplicáveis em Java e outras linguagens

Outros Padrões

- Dois outros padrões são muito populares atualmente
  - Dependency Injection: um caso particular de um dos padrões GRASP – **General responsibility assignment software patterns** – (Indirection) bastante popular no momento (também conhecido como Inversão de Controle)
  - Aspectos: uma extensão ao paradigma orientado a objetos que ajuda a lidar com limitações dos sistemas OO

Padrão de projeto MVVM

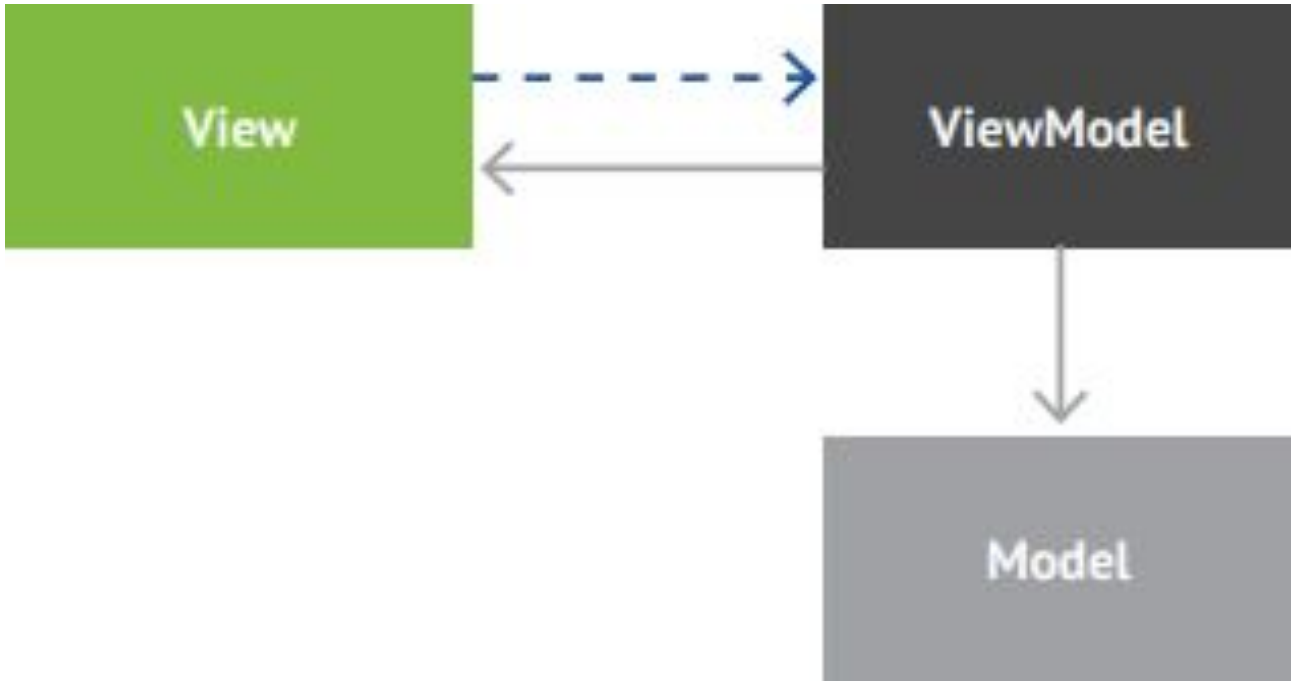
Arquitetura de Software

Padrão de projeto MVVM

O MVVM (Model-View-ViewModel) foi divulgado pela primeira vez em 2005 por John Gossman, na época arquiteto da plataforma Silverlight da Microsoft em seu blog.

O mesmo se baseou fortemente no padrão de projeto Presentation Model (muito semelhante ao MVP – Model-View-Presenter), divulgado por Martin Fowler em 2004.

Padrão de projeto MVVM



Padrão de projeto MVVM

Um lembrete sobre o MVVM

O padrão MVVM é uma variação de outro padrão de separação bem conhecido chamado Model-View-Controller, ou MVC.

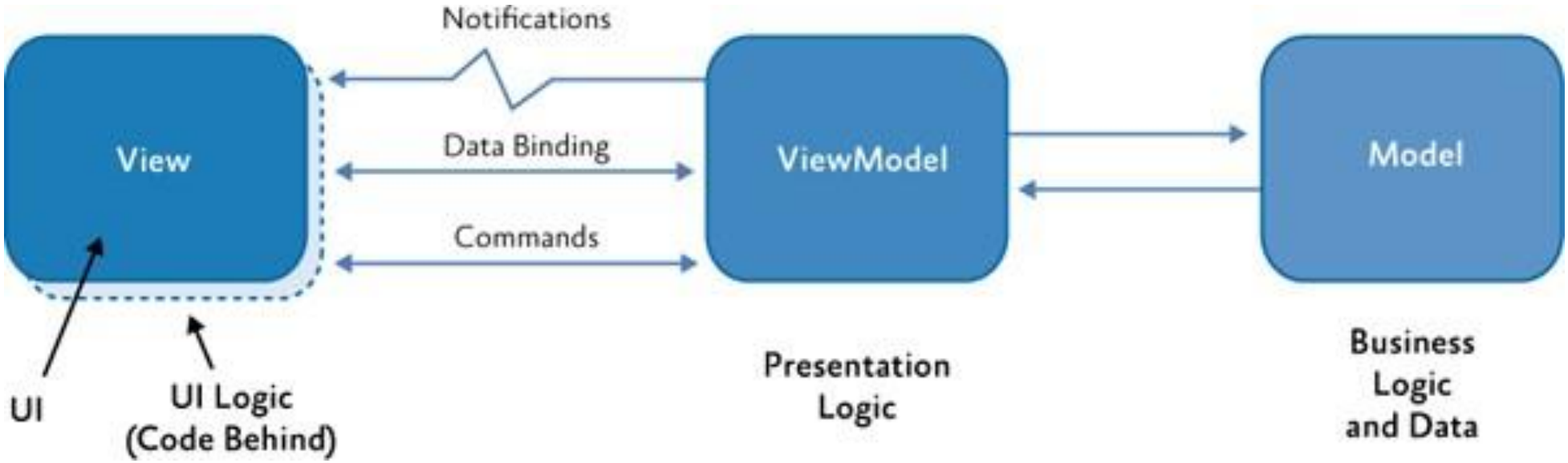
Esse padrão é usado em uma infinidade de estruturas, notavelmente a estrutura de aplicativo Web amplamente usada Ruby on Rails, bem como o ASP.NET MVC by Microsoft.

Ele é usado não só em aplicativos Web, mas também amplamente de aplicativos de desktop a aplicativos móveis (em iOS, por exemplo).

Arquitetura de Software

Padrão de projeto MVVM

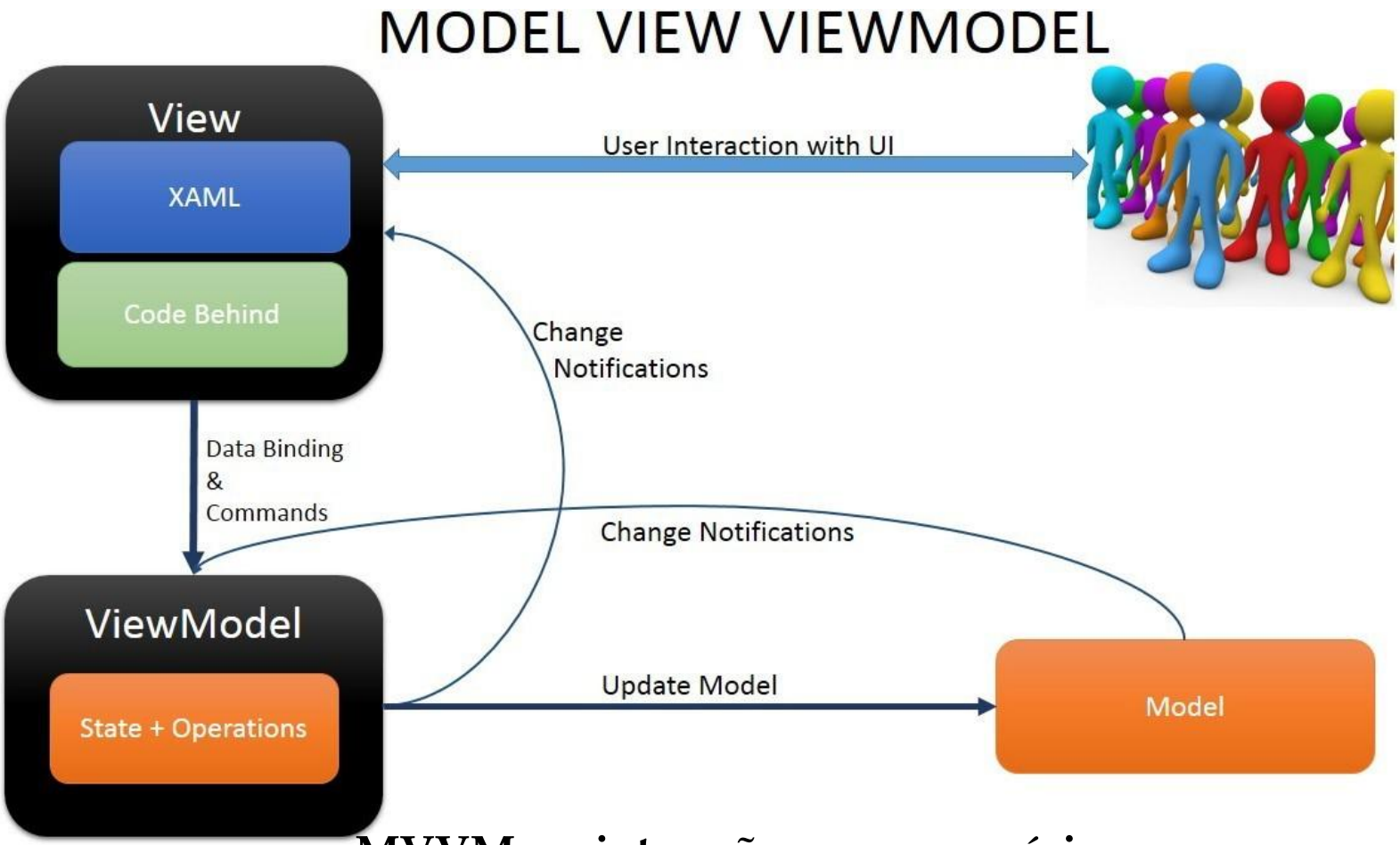
Um lembrete sobre o MVVM



MVVM

Arquitetura de Software

Padrão de projeto MVVM



MVVM e a interação com os usuários

Arquitetura de Software

Padrão de projeto MVVM

Um lembrete sobre o MVVM

A principal vantagem de um padrão de separação é que ele atribui responsabilidades claramente definidas para cada uma das camadas.

Com essas responsabilidades claramente definidas, os membros da equipe de desenvolvimento podem se concentrar em cada parte sem pisar um no calo do outro.

Da mesma forma, um designer de interação não precisa se preocupar com a criação ou a persistência dos dados e assim por diante.

Arquitetura de Software

Padrão de projeto MVVM

MVP

O padrão Model-View-Presenter (MVP) é uma variação do padrão Model-View-Controller,.

O que você vê na tela é o modo de exibição, os dados que ele exibe são o modelo e o apresentador conecta os dois juntos.

O modo de exibição se baseia em um apresentador para preenchê-la com dados de modelo, reagir a entrada do usuário, fornecer validação de entrada (por exemplo, delegando para o modelo) e outras tarefas.

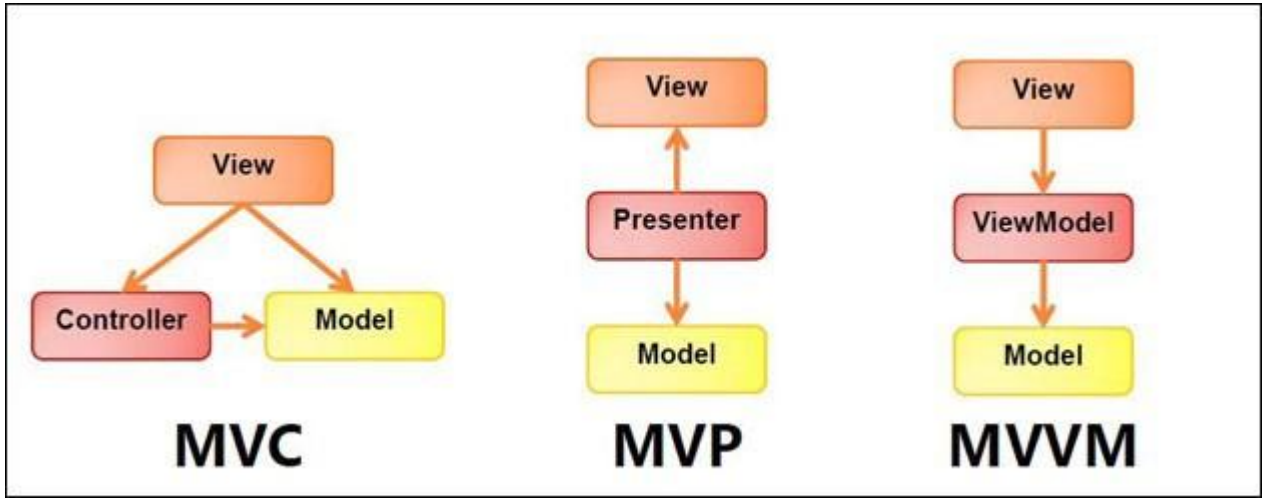
Arquitetura de Software

Padrão de projeto MVVM

De MVC para MVVM

Embora o padrão MVC tenha vantagens claras para muitas estruturas, ele não é o mais adequado para estruturas baseadas em XAML devido (ou graças) ao sistema de associação de dados.

Essa poderosa infra-estrutura pode associar as propriedades de objetos diferentes e mantê-las sincronizadas.



MVC x MVP x MVVM

Arquitetura de Software

Padrão de projeto MVVM

De MVC para MVVM

Para facilitar a associação de dados e evitar objetos muito grandes, o controlador é dividido em objetos menores, mais finos, chamados ViewModels, ou modelos de apresentação.

Um uso comum é combinar uma exibição com um ViewModel, definindo o ViewModel como DataContext da exibição.

Na prática, no entanto, esse não é sempre o caso; não é incomum ter várias exibições associadas a determinado ViewModel, ou ter uma exibição complexa dividida em vários ViewModels.

Arquitetura de Software

Padrão de projeto MVVM

De MVC para MVVM

Graças ao uso do XAML para criar a interface do usuário, a associação de dados pode ser expressa de maneira declarada, diretamente no corpo do documento XAML.

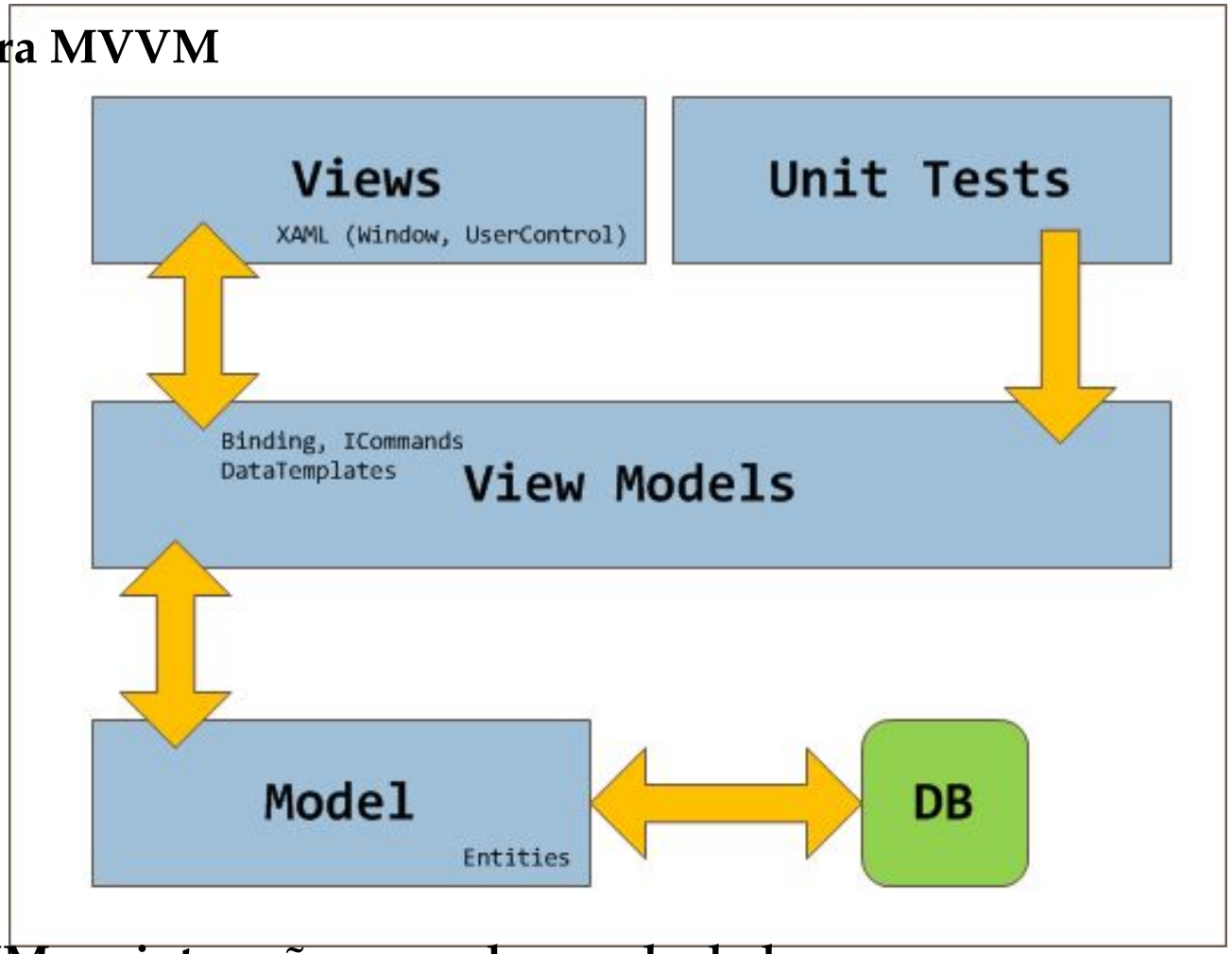
Isso permite que um processo dissociado, onde o desenvolvedor se preocupa com o modelo e o ViewModel, enquanto um designer de interação assume a criação da experiência do usuário ao mesmo tempo, ou até mesmo em um momento posterior.

Quando configurado corretamente, o MVVM permite visualizar dados de tempo de design na tela, o que permite que o designer da experiência do usuário trabalhe sem ter de executar o aplicativo.

Arquitetura de Software

Padrão de projeto MVVM

De MVC para MVVM



MVVM e a interação com o banco de dados

Arquitetura de Software

Padrão de projeto MVVM

MVVM - Camada VIEW

Responsabilidades e características das classes

A responsabilidade da View é definir a aparência ou estrutura que o usuário vê na tela.

O ideal é que o codebehind da view, contenha apenas a chamada ao método InitializeComponent dentro do construtor, ou em alguns casos, código que manipule os controles visuais, ou crie animações; algo que é mais difícil de fazer em XAML.

A View se liga ao ViewModel, através da propriedade **DataContext** que é setada para a classe ViewModel correspondente à aquela View.

Arquitetura de Software

Padrão de projeto MVVM

MVVM - Camada VIEW

Características comuns

- A View é um elemento visual, como um objeto Window, Page, UserControl ou DataTemplate.

-A View referencia a ViewModel através da propriedade **DataContext**. Os controles da View são preenchidos com propriedades ou comando, expostos pela ViewModel.

-O codebehind da view, define comportamentos visuais (Behaviors) difíceis de expressar em XAML.

Arquitetura de Software

Padrão de projeto MVVM

MVVM - Camada VIEWMODEL

A responsabilidade da ViewModel no contexto do MVVM, é disponibilizar para a View uma lógica de apresentação.

A View Model não tem nenhum conhecimento específico sobre a view, ou como ela implementada, nem o seu tipo.

A ViewModel implementa propriedades e comandos, para que a View possa preencher seus controles e notifica a mesma, caso haja alteração de estado; seja através de eventos ou notificação de alteração.

Arquitetura de Software

Padrão de projeto MVVM

MVVM - Camada VIEWMODEL

A ViewModel é peça fundamental no MVVM, por que é ela quem vai coordenar as iterações da View com o Model, haja vista, ambos não terem conhecimento um do outro.

A ViewModel, pode implementar a lógica de validação, para garantir a consistência dos dados.

Arquitetura de Software

Padrão de projeto MVVM

MVVM - Camada VIEWMODEL

Caracteristicas comuns

-A ViewModel é uma classe não visual, que expões para a View uma lógica de apresentação.

-A ViewModel é testável, independentemente da View ou Model.

-A ViewModel coordena as iteações entre a View e o Model.

-A ViewModel não referencia a View, na verdade não tem nenhum conhec imento sobre a mesma.

Arquitetura de Software

Padrão de projeto MVVM

MVVM - Camada VIEWMODEL

Caracteristicas comuns

-A ViewModel implementa as interfaces INotifyPropertyChanged

-A ViewModel expões propriedade e comando, para que a View possa utilizar para preencher seus controles; e notifica a View quando o estado de uma determinada propriedade muda, via implementação da inteface **INotifyPropertyChanged** ou **INotifyCollectionChanged.**

- A ViewModel pode conter a lógica de validação, atravéz da implementação da intefaces **IDataErrorInfo** ou **INotifyDataErrorInfo.**

Arquitetura de Software

Padrão de projeto MVVM

MVVM - Camada MODEL

O Model no MVVM, encapsula a lógica de negócios e os dados.

O Modelo nada mais é do que o Modelo de domínio de uma aplicação, ou seja, as classes de negócio que serão utilizadas em uma determinada aplicação.

O Modelo também contém os papéis e também a validação dos dados de acordo com o negócio, cuja aplicação em questão visa atender.

Arquitetura de Software

Padrão de projeto MVVM

MVVM - Camada MODEL

Características comuns

-O Modelo são classes que encapsulam a lógica de negócios e os dados.

-O Modelo não referencia diretamente a View ou ViewModel.

-O Modelo provê eventos de notificação de mudança de estado, através das interfaces **INotifyPropertyChanged** and **INotifyCollectionChanged.** Isto facilita o preenchimento de dados na View.

-O Modelo de dados contém validação de dados e reporta os erros através da interface INotifyDataErrorInfo.

- O Modelo de dados geralmente é utilizado, com um repositório (pode ser o Repository Pattern) ou serviço.

Padrões Comportamentais

1. Chain of Responsability
2. Command
3. Iterator
4. Mediator
5. Memento
6. State

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.

Padrões Comportamentais

Chain of Responsability

Chain of Responsability

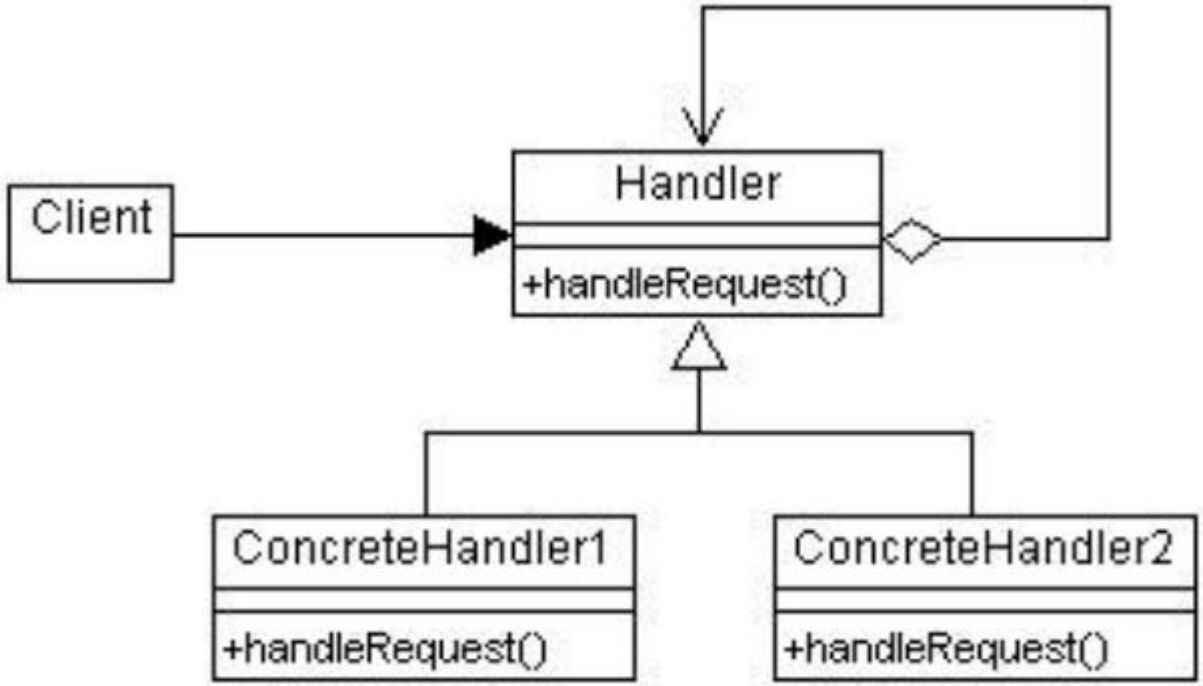
- **Intenção** – Evitar o acoplamento do remetente de uma solicitação ao seu receptor, dando a mais de um objeto a oportunidade de tratar uma solicitação. – Encadear os objetos receptores passado a solicitação ao longo da cadeia até que um objeto a trate.

- **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.

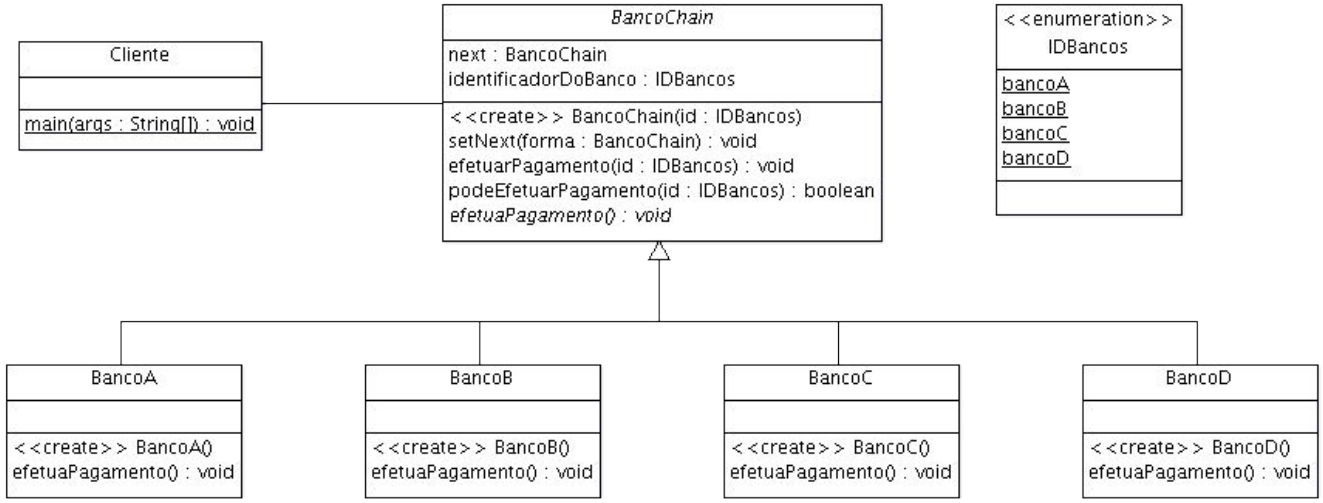
Chain of Responsability

Estrutura



Chain of Responsability

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 trata-lo.

• **Conseqüências** – Acoplamento reduzido. – Flexibilidade na atribuição de responsabilidades. – A recepção não é garantida.

• Padrões Correlatos – Composite

Padrões Comportamentais

Command

Padrões Comportamentais

Command

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).

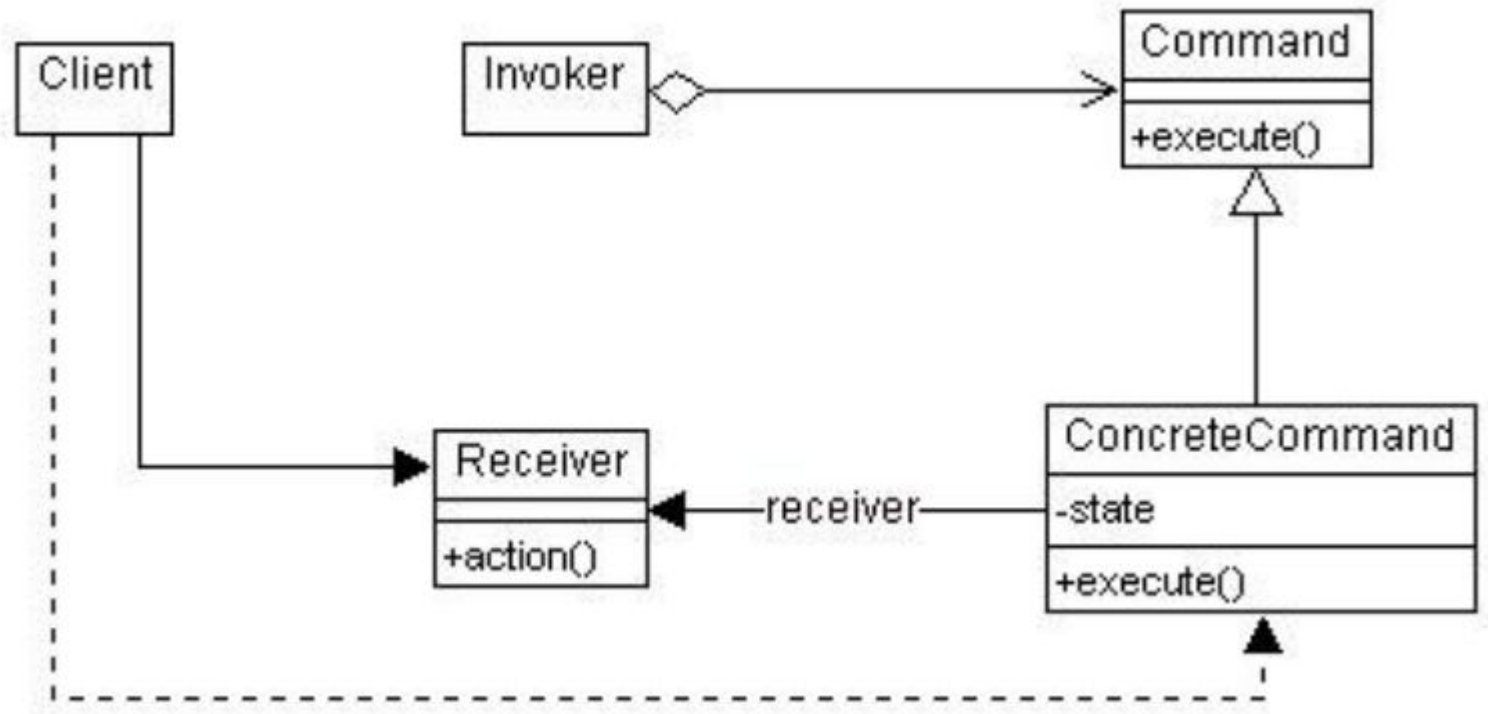
• **Motivação –** Existem situações nas quais é necessário emitir solicitações para objetos sem que se conheça nada a respeito da operação ou do receptor da mesma.

• **Aplicabilidade:** situações em que deseja-se:

– parametrizar as ações a serem 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.

Command

Estrutura



Command

Colaborações

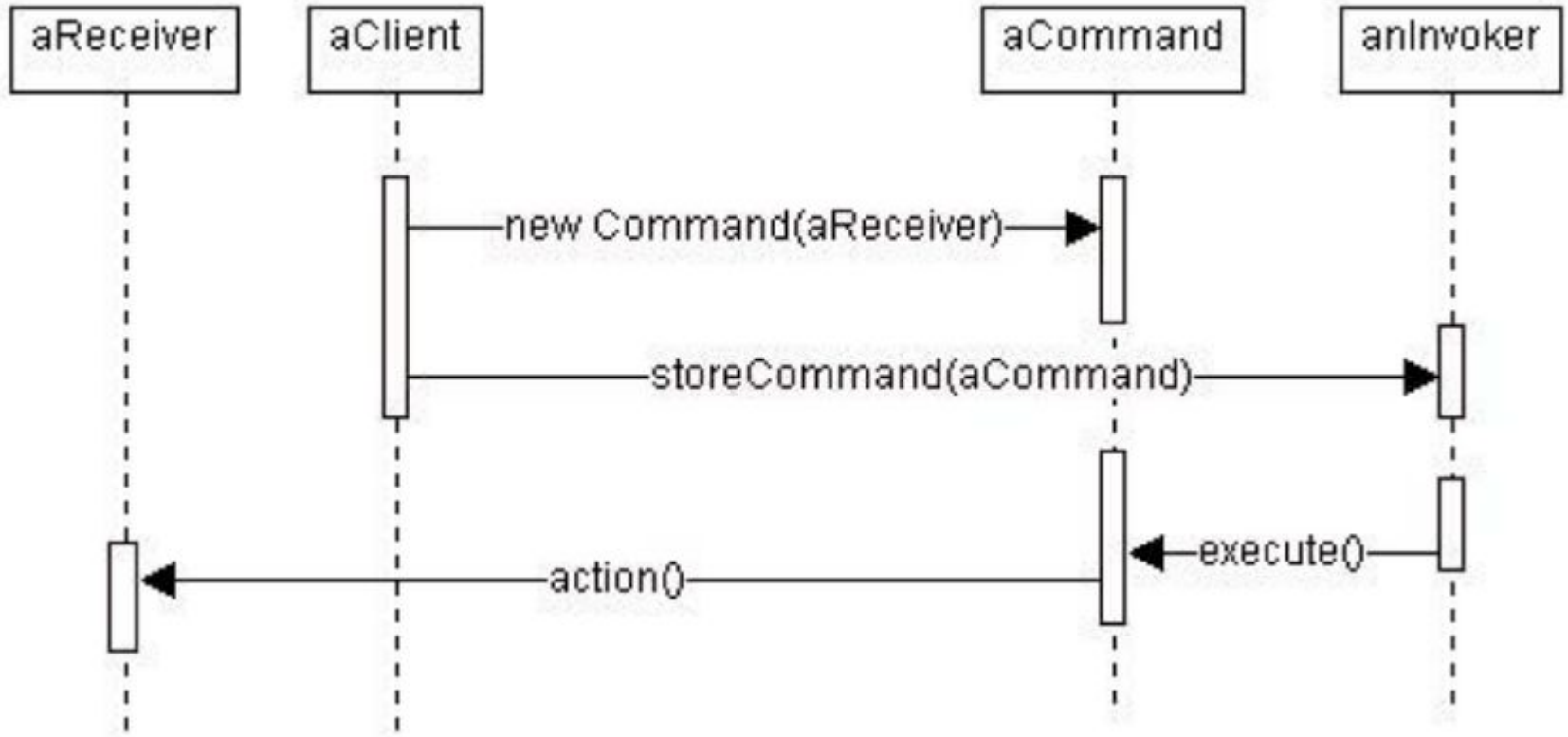
– O cliente cria um objeto **ConcreteCommand** e especifica o seu receptor

– 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().

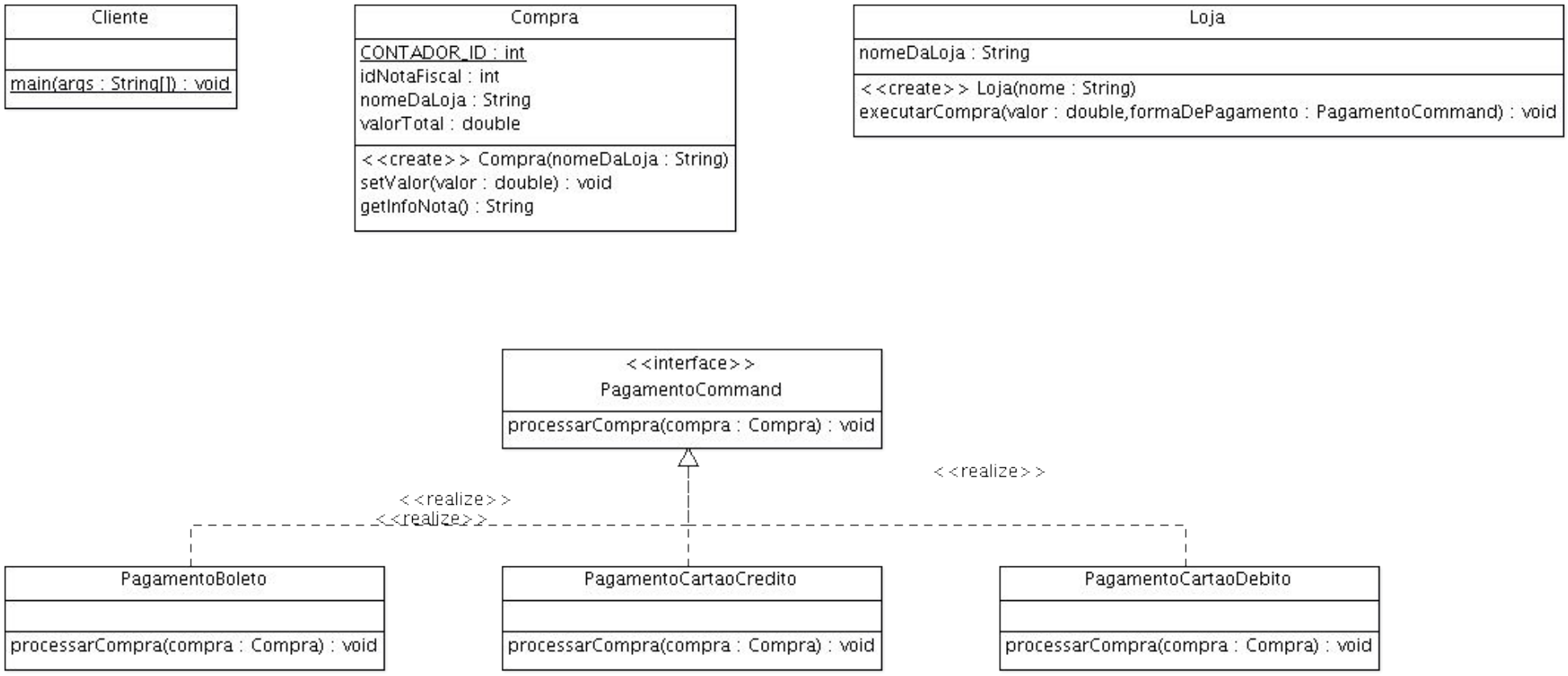
– O objeto ConcreteCommand invoca operações no seu Receiver para executar a solicitação

Command


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

Command

Exemplo



- **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.

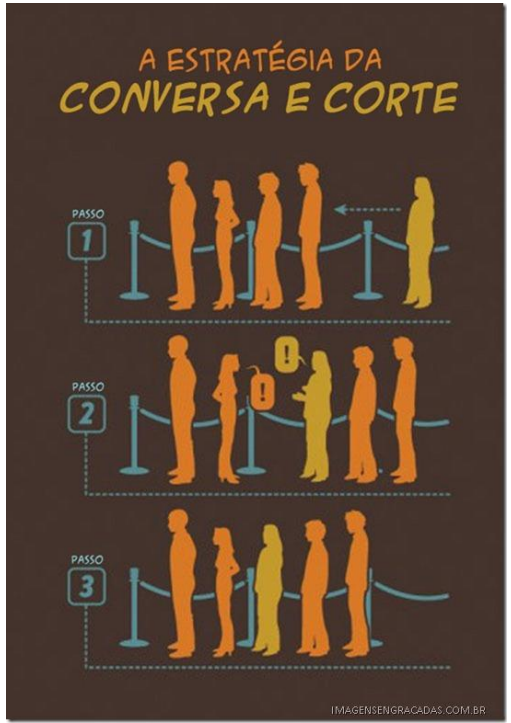
– É possível juntar comandos formando um 'comando composto' (podendo-se usar o padrão Composite).

– É fácil acrescentar novos Commands porque não é necessário mudar classes existentes.

- **Padrões Correlatos** – Composite, Memento, Prototype

Padrões Comportamentais

Iterator



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

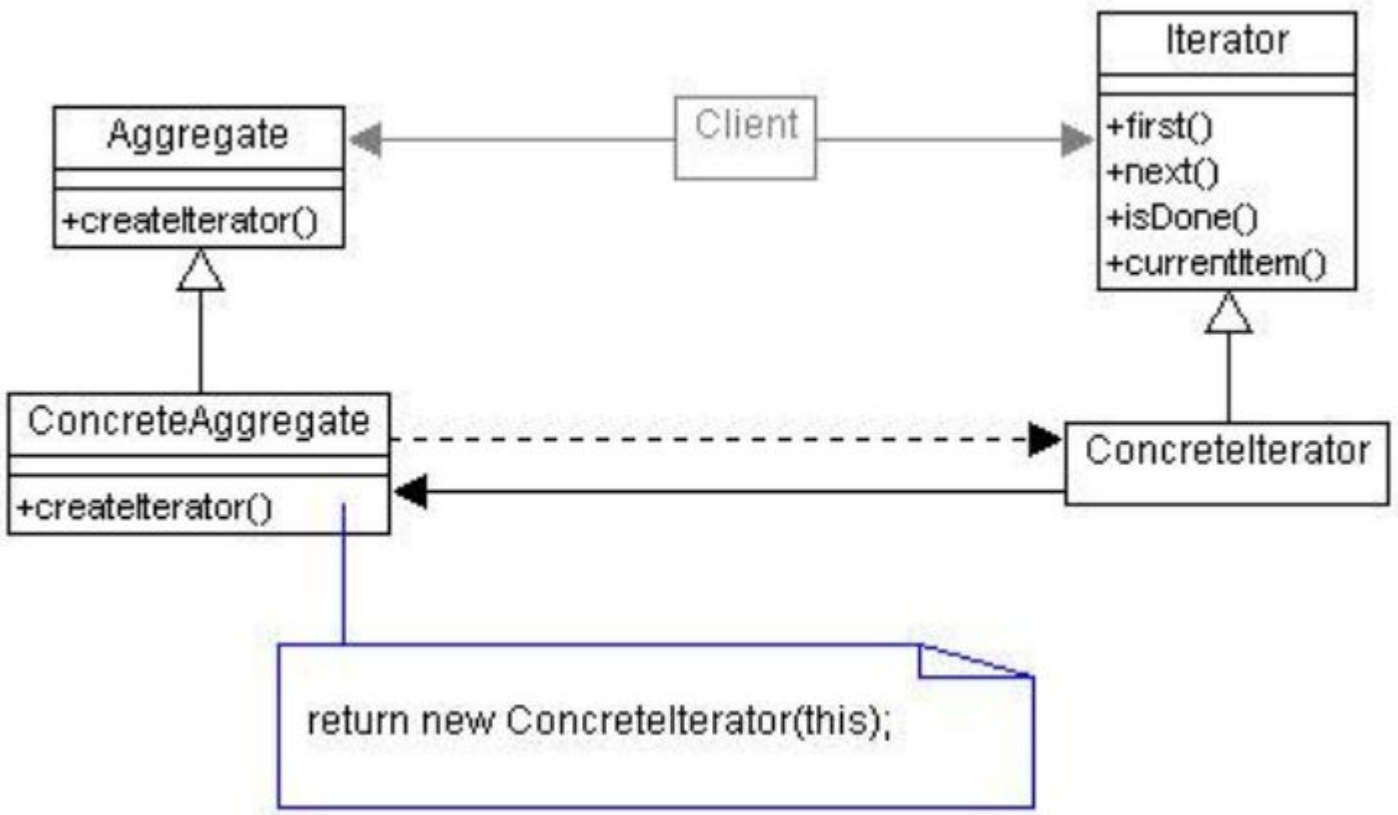
Aplicabilidade

– 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


Iterator
+first()
+next()
+isDone()
+currentItem()
Aggregate
+createIterator()
Client
ConcreteAggregate
+createIterator()
ConcreteIterator
return new ConcreteIterator(this);

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.

Padrões Comportamentais

Mediator

Padrões Comportamentais

Mediator

Mediator

- **Intenção**

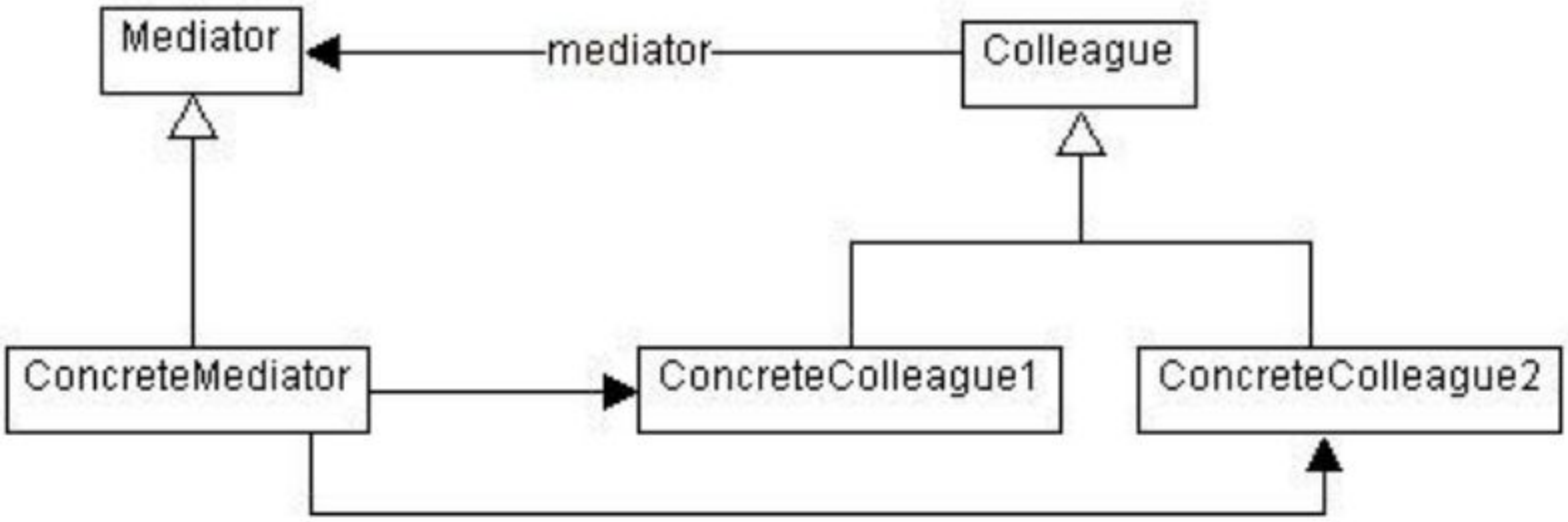
– Definir um objeto que encapsula a forma como um conjunto de objetos interage.

– 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**

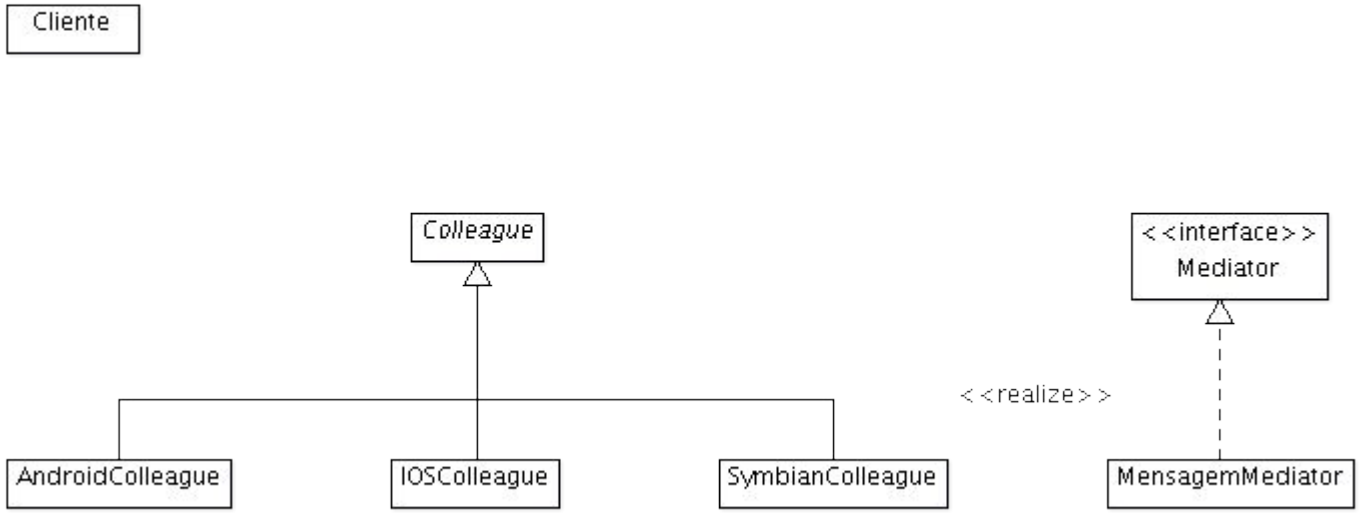
– Em projetos orientados a objetos, é normal distribuir o comportamento entre várias classes. Isso pode resultar numa estrutura com muitas conexões entre os objetos e gera a necessidade de que cada objeto conheça os demais.

Mediator


Mediator
mediator
Colleague
ConcreteMediator
ConcreteColleague1
ConcreteColleague2

Mediator

Exemplo



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**

– Facade, Observer

Padrões Comportamentais

Memento

Padrões Comportamentais

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).

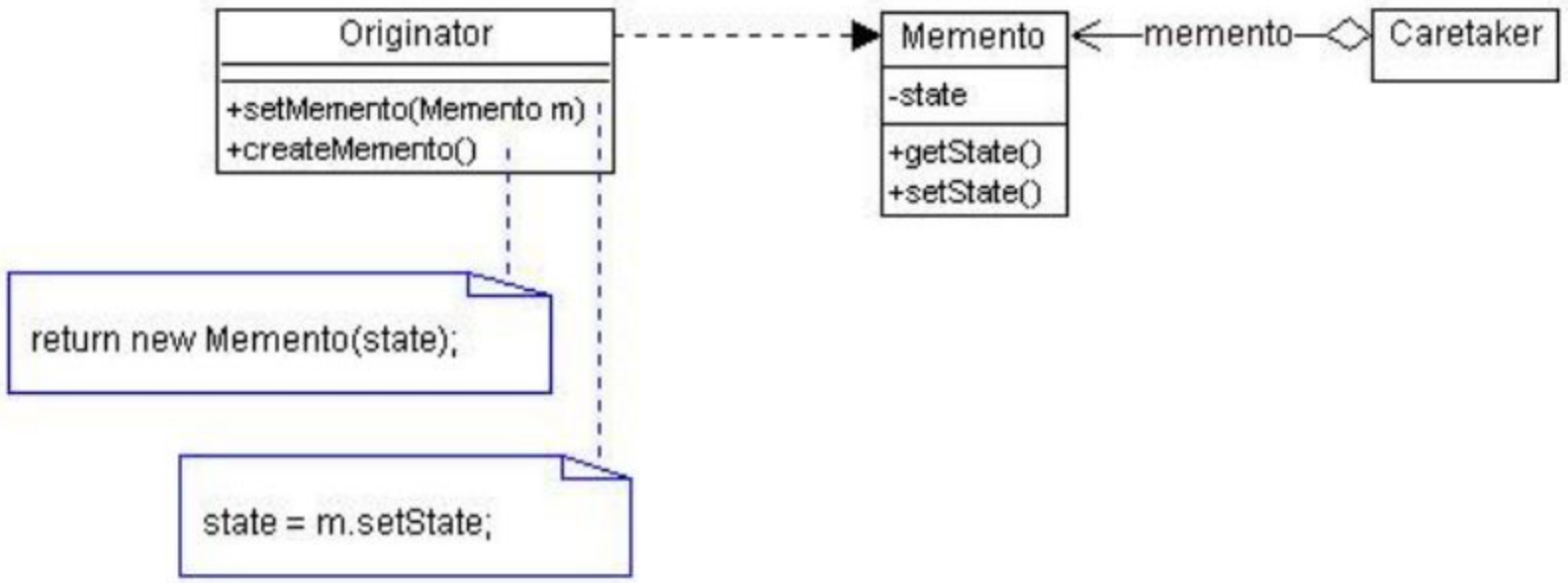
- **Aplicabilidade**

– 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.

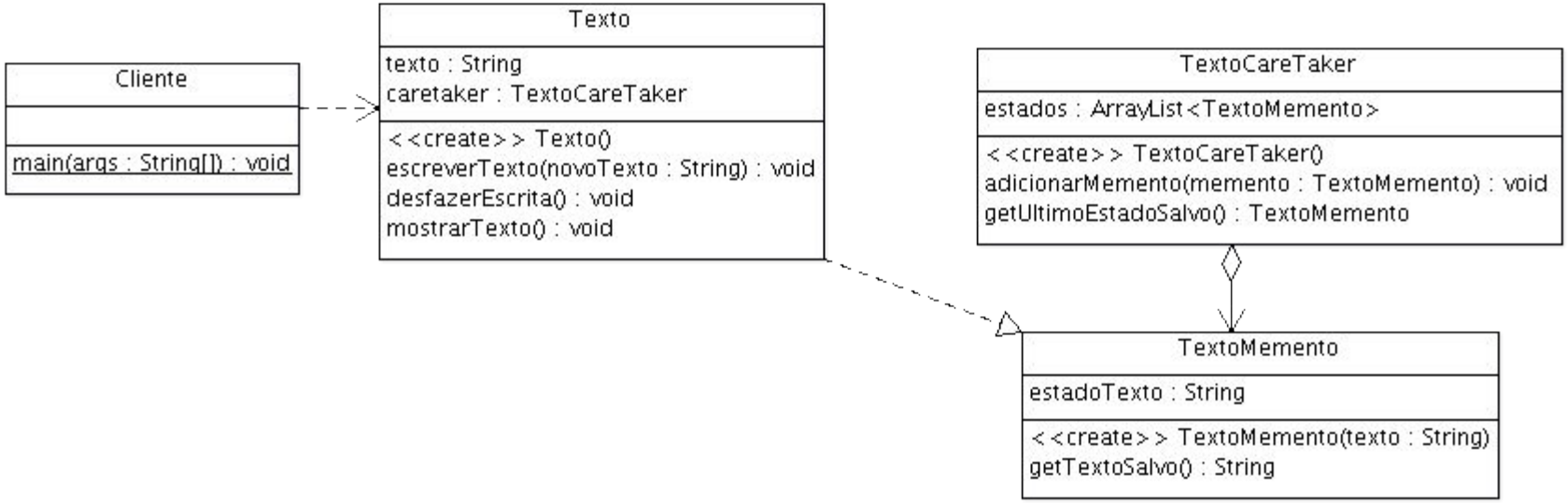
Memento

Estrutura



Memento

Exemplo



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

Padrões Comportamentais

State

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.

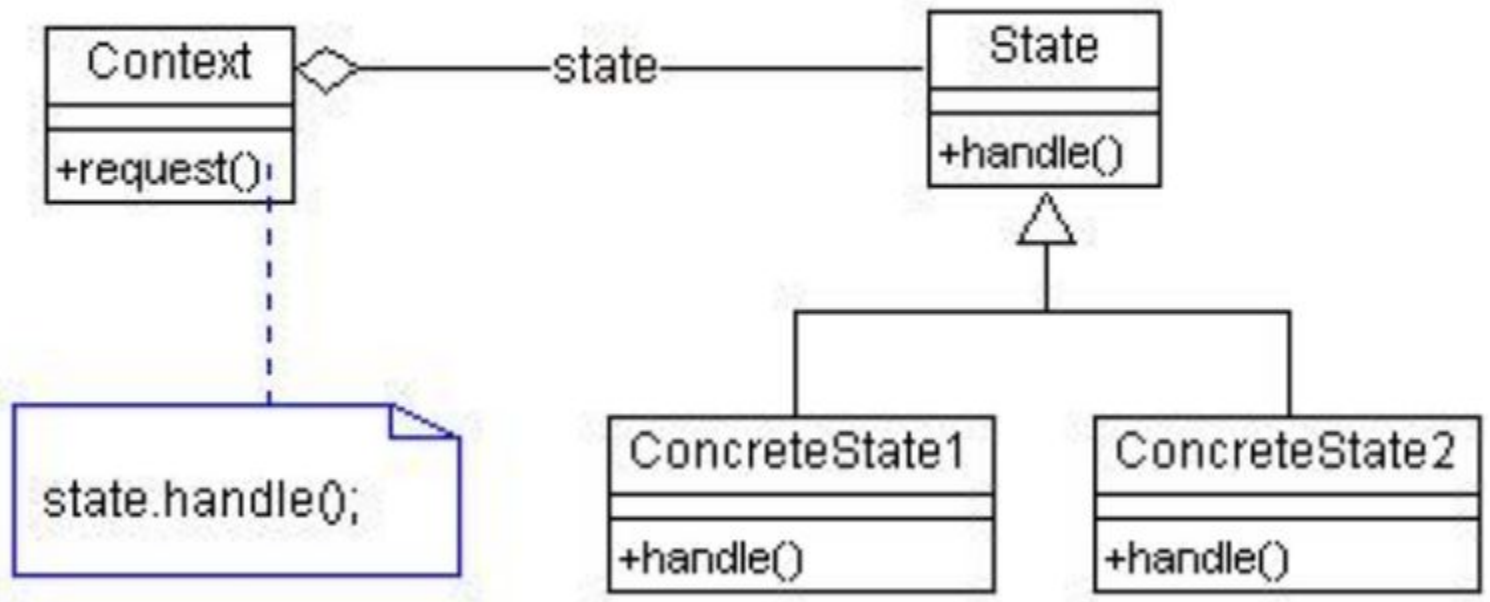
- **Aplicabilidade**

– 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.

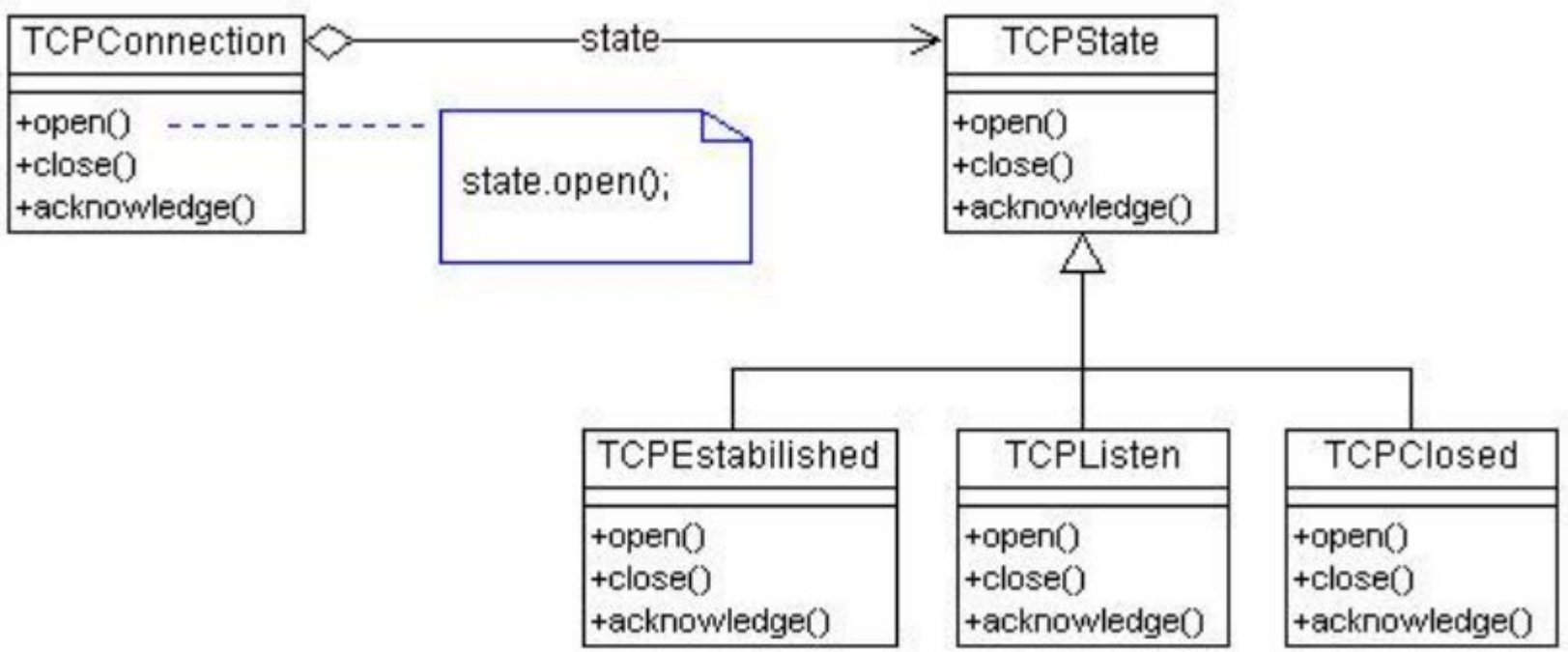
State

Estrutura



State

Exemplo



- **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 sequência de estados.

- **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 possuírem variáveis de instância. Nesse caso eles acabam implementando o padrão Flyweight sem estado intrínseco.

- **Padrões Relacionados**

– Flyweight, Singleton