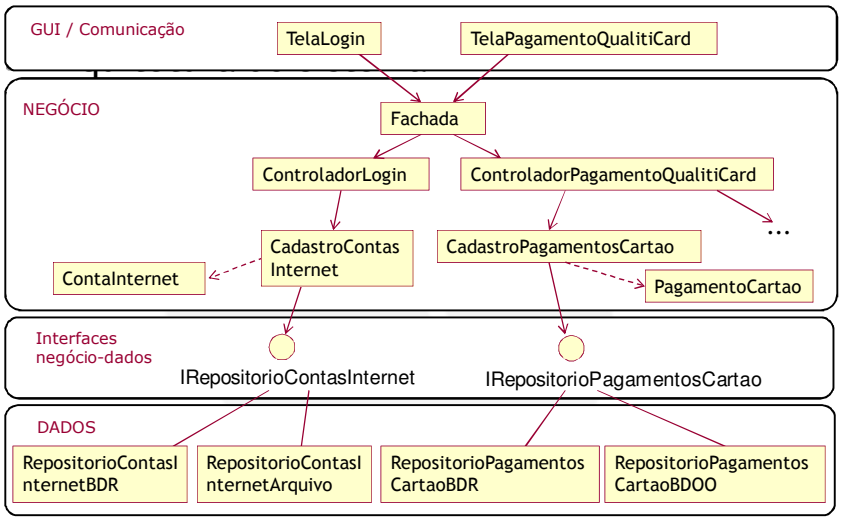
Introdução a Padrões

Contexto



Evolução de Software

"Grandes sistemas de software nunca são completados; eles simplesmente continuam evoluindo" [Lehman]

- Motivos e conseqüências (algumas Leis de Lehman)
  - Mudança Contínua e Crescimento Contínuo
    - Mudanças são inevitáveis para que o software continue útil
    - Adição de funcionalidades para manter a satisfação do usuário
  - **Complexidade Crescente e Qualidade Decrescente**
    - Mudanças aumentam a complexidade do software
    - Qualidade diminui a não ser que haja adaptações

Como devemos proceder?

Pensar antes de agir!

Algumas ações:

- Analisar impactos
- Modelar, antes de codificar
- Utilizar soluções e técnicas validadas
  - Não reinventar a roda!

Padrões em outras Engenharias

"Cada padrão descreve um problema que ocorre repetidas vezes em nosso ambiente, e então descreve o núcleo da solução para aquele problema, de tal maneira que pode-se usar essa solução milhões de vezes sem nunca fazê-la da mesma forma duas vezes"

Christopher Alexander, sobre padrões em Arquitetura

Padrões de Projeto/ *Design Patterns*

- Soluções (validadas) para alcançar objetivos na engenharia de software usando linguagens O-O
  - Registram o conhecimento existente de uma forma que possa ser reaplicados em vários contextos

"Os padrões de projeto são descrições de objetos que se comunicam e classes que são customizadas para resolver um problema genérico de design em um contexto específico"

Gamma, Helm, Vlissides & Johnson, sobre padrões em software

Por que aprender padrões?

- Aprender com a experiência dos outros
  - identificar problemas comuns em engenharia de software
  - utilizar soluções testadas e bem documentadas
- Aprender boas práticas de programação
  - Padrões utilizam eficientemente Herança, composição, modularidade, polimorfirmo e abstração para construir código reutilizável, eficiente, de alta coesão e baixo acoplamento
- Ter um caminho e um alvo para refatorações
- Facilita a comunicação, compreensão e documentação
  - Vocabulário comum
  - Ajuda a entender a papel de classes do sistemas.

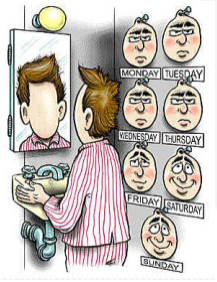
Elementos de um padrão

- Nome
- Problema
  - Quando aplicar o padrão, em que condições?
- Solução
  - Descrição abstrata de um problema e como usar os elementos disponíveis (classes e objetos) para solucioná-lo
- Conseqüências
  - Custos e benefícios de se aplicar o padrão
  - Impacto na flexibilidade, extensibilidade, portabilidade e eficiência do sistema

Metáforas no mundo real

State

Pessoas reagem de formas diferente de acordo com o humor.

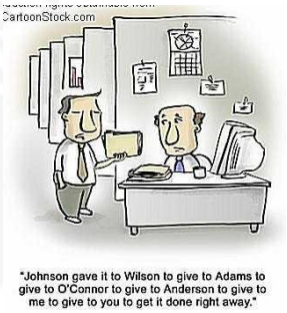


Observer

Jornais são entregues aos seus assinantes .

Chain of Responsability

Chefe delega responsabilidade a um subordinado, que delega a outro, que ...



Padrões são formados por vários objetos.
Cada objeto tem um **papel** (**responsabilidade)** na solução.

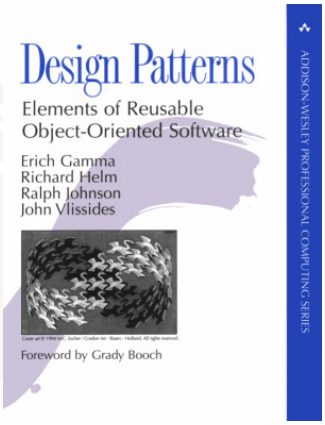
Padrões de Projeto GoF

Padrões de Projeto/Design

Introduzidos no livro em 1994

- Descreve 23 padrões
- soluções genéricas para os problemas mais comuns do desenvolvimento de software orientado a objetos;
- obtidas através de experiências de sucesso na indústria de software

- Sobre o livro
  - Seus quatro autores são conhecidos como "The **G**ang **o**f **F**our" (GoF).
  - O livro tornou-se um clássico na literatura orientada a objetos e continua atual.



Mais padrões?

- Há vários catálogos de padrões para software
  - Muitos são específicos a uma determinada área
    - padrões J2EE
    - padrões de implementação Java ou C#
    - padrões para concorrência
    - padrões para distribuição,
    - ...
  - Os padrões apresentados aqui são aplicáveis em Java e outras linguagens O-O

Padrões de Projeto

Formas de classificação

Há várias formas de classificar os padrões

- Os padrões GoF são tradicionalmente classificados pelo propósito:
  1. criação de classes e objetos;
  2. alteração da estrutura de um programa,
  3. controle do seu comportamento
- Metsker os classifica em 5 grupos, por intenção
  1. oferecer uma interface,
  2. atribuir uma responsabilidade,
  3. realizar a construção de classes ou objetos
  4. controlar formas de operação
  5. implementar uma extensão para a aplicação

23 Padrões GoF Classificação por Propósito

		Propósito		
		Criação	Estrutura	Comportamento
Escopo	Classe	Factory Method	Class Adapter	Interpreter Template Method
	Objeto	Builder Abstract Factory Prototype Singlenton	Object Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

*** Classificação segundo GoF.*

23 Padrões GoF
Classificação por Intenção

Intenção	Padrões
1. Interfaces	Adapter, Facade, Composite, Bridge
3. Construção	Factory Method, Abstract Factory, Builder, Prototype, Memento
2. Responsabilidade	Singleton, Observer, Chain of Responsibility, Mediator, Proxy, Flyweight
4. Operações	Template Method, State, Command, Strategy, Interpreter
5. Extensões	Iterator, Decorator, Visitor

Vistos em Sala Vistos Anteriormente Praticados nos próximos exercícios

*** Classificação segundo Steven Metsker, Design Patterns Java Workbook, 2002.*

PADRÕES RELACIONADOS A INTERFACES

Interface

- Interface:
  - coleção de métodos e constantes que uma classe permite que objetos de outras classes acessem
- Exigem que a classe que implementa a interface ofereça implementação para seus métodos
- Não garante que métodos terão implementação que efetue alguma computação: *stubs*

Padrões relacionados a Interfaces

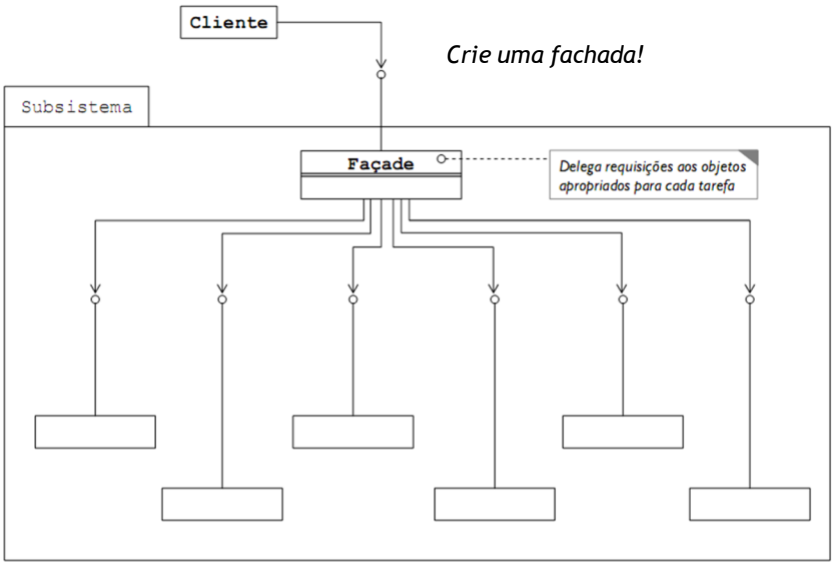
- Adapter:
  - para adaptar a interface de uma classe para outra que o cliente espera
- Composite:
  - definir uma interface comum para objetos individuais e composições de objetos
- Bridge:
  - desacoplar uma abstração de sua implementação para que ambos possam variar independentemente
- Façade:
  - oferecer uma interface simples para uma coleção de classes

Visto anteriormente

Fachada

- Objetivo segundo o GoF:
  - "oferecer uma interface única para um conjunto de interfaces de um subsistema."
  - "Façade define uma interface de nível mais elevado que torna o subsistema mais fácil de usar."

Relembrando o Padrão Façade

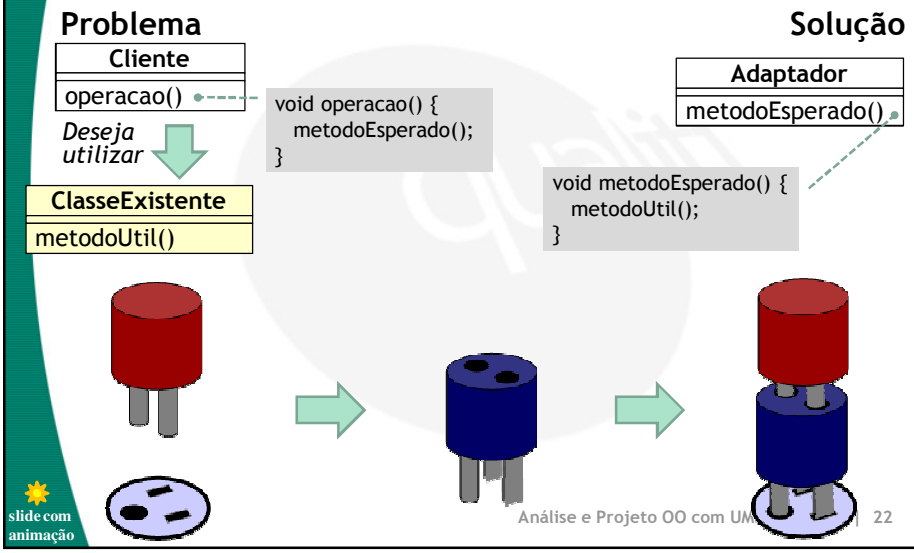


slide com animação

Adapter

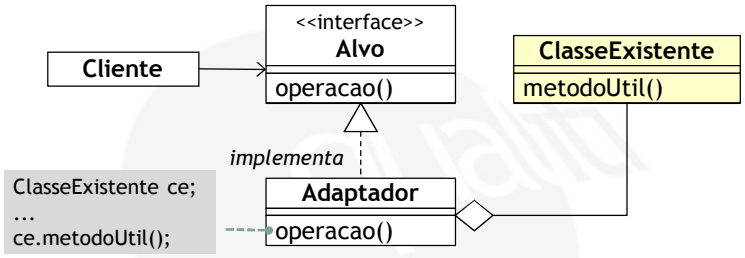
- Objetivo segundo o GoF:
  - "converter a interface de uma classe em outra interface esperada pelos clientes."
  - "Adapter permite a comunicação entre classes que não poderiam trabalhar juntas devido à incompatibilidade de suas interfaces."

Adapter
Problema e Solução



slide com animação

Object Adapter



- Cliente: aplicação que colabora com objetos que implementam Alvo
- Alvo: define a interface requerida pelo Cliente
- ClasseExistente: classe dos objetos que requerem adaptação
- Adaptador: adapta a interface do Recurso à interface Alvo

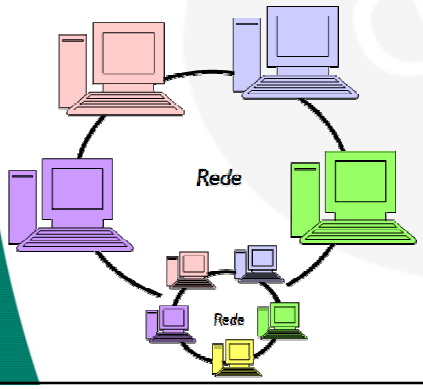
Composite

- Objetivo segundo o GoF:
  - "Compor objetos em estruturas de árvore para representar hierarquias todo-parte."
  - "Composite permite que clientes tratem objetos individuais e composições de objetos de maneira uniforme."

Composite
Problema e Solução

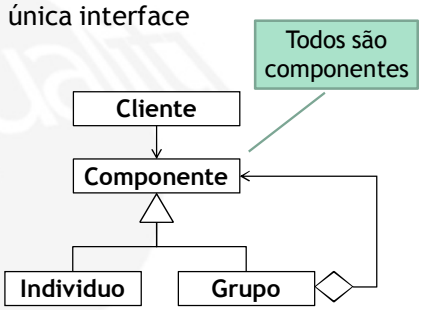
Problema

- Cliente precisa tratar de maneira uniforme objetos individuais e composições desses objetos

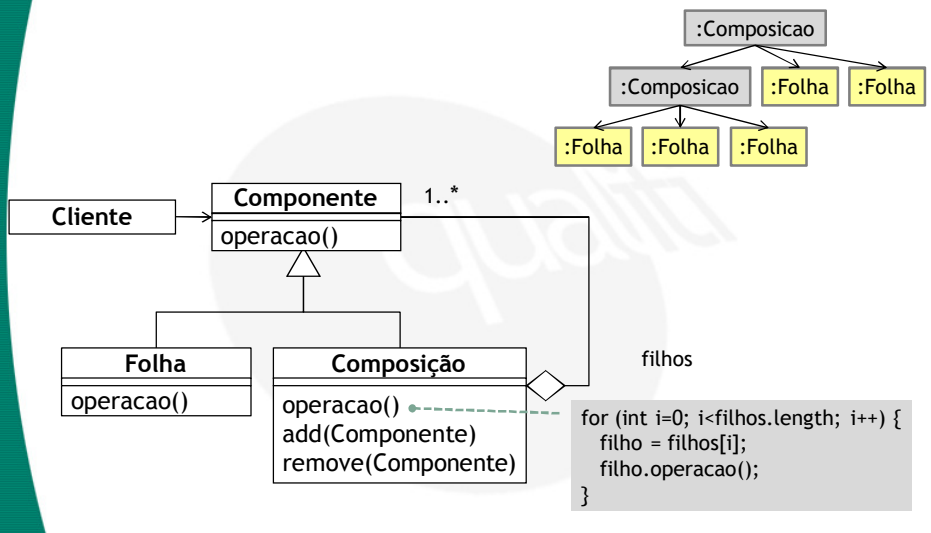


Solução

- Tratar grupos e indivíduos diferentes através de uma única interface



Estrutura do Composite



Discussão

1. Você consegue citar exemplos de Composite e Adaptor na API JAVA ou em outros frameworks?
2. Você consegue visualizar outra forma de criar adaptadores?
   - Quais as vantagens de sua abordagem?

Resumo: quando usar?

- Façade
  - **Simplificar** o uso de uma coleção de objetos
- Adapter
  - **Adaptar uma interface** existente para um cliente
- Bridge
  - Implementar um design que permita total **desacoplamento entre interface e implementação**
- Composite
  - Tratar **composições** e unidades uniformemente

PADRÕES QUE OFERECEM ALTERNATIVAS À CONSTRUÇÃO DE OBJETOS

Construtores

- Alguns problemas em depender de construtores
  - Cliente pode não ter todos os dados necessários para instanciar um objeto
  - Cliente fica acoplado a uma implementação concreta (precisa saber a classe concreta para usar new com o construtor)
  - Objeto complexo pode necessitar da criação de objetos menores previamente, com certo controle difícil de implementar com construtores
  - Não há como limitar o número de instâncias criadas

Padrões que oferecem alternativas à construção de objetos

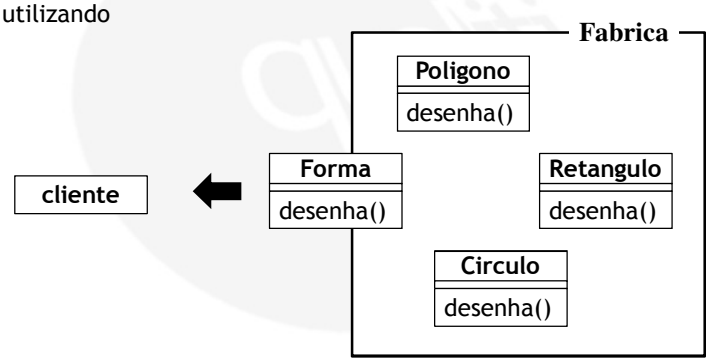
- Factory Method: adia a decisão sobre qual classe concreta instanciar
- Abstract Factory: constuir uma família de objetos que compartilham um "tema" em comum
- Prototype: especificar a criação de um objeto a partir de um exemplo fornecido
- Memento: reconstruir um objeto a partir de uma versão que contém apenas seu estado interno
- Builder: obtém informação necessária em passos antes de requisitar a construção de um objeto

Factory Method

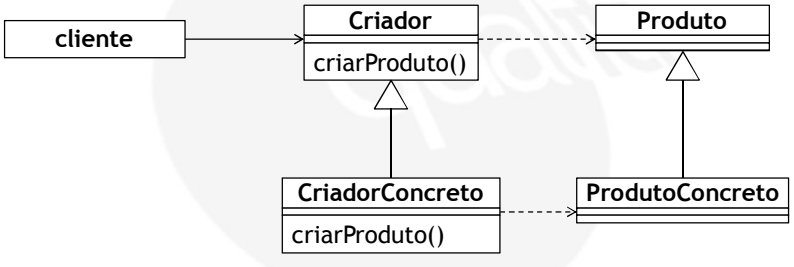
- Objetivo segundo o GoF:
  - "Definir uma interface para criar um objeto mas deixar que subclasses decidam que classe instanciar."
  - Factory Method permite que uma classe delegue a responsabilidade de instanciação às subclasses."

Funcionamento de uma Fábrica

- Problema:
  - O acesso a um objeto concreto será feito através da interface conhecida através de sua superclasse, mas o cliente também não quer (ou não pode) saber qual implementação concreta estará utilizando



Estrutura do Factory Method

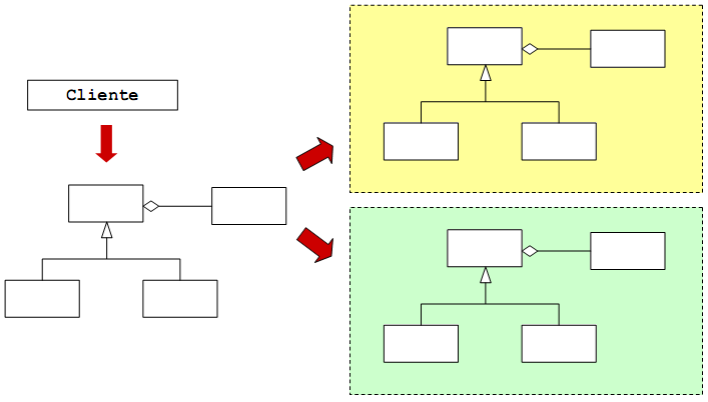


Abstract Factory

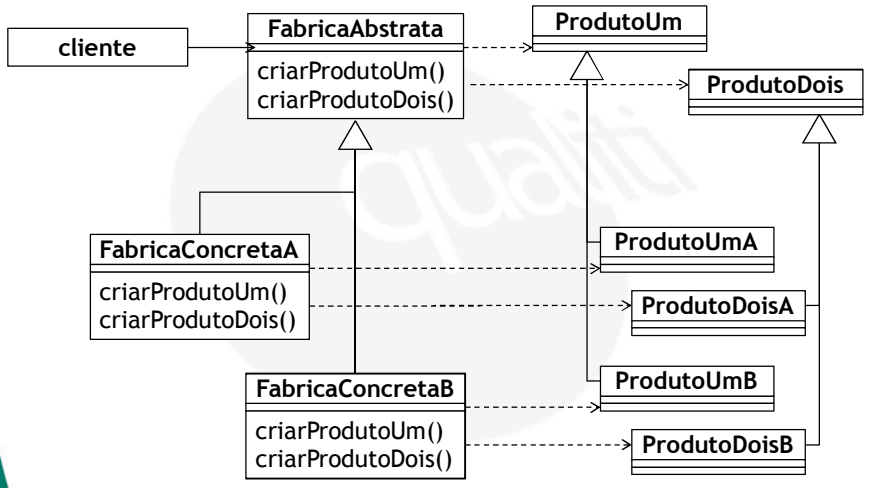
- Objetivo segundo o GoF:
  - "Prover uma interface para criar famílias de objetos relacionados ou dependentes sem especificar suas classes concretas."

Problema

- Criar uma família de objetos relacionados sem conhecer suas classes concretas



Estrutura



Discussão

1. Qual a diferença entre Factory Method e Abstract Factory?
2. É possível o cliente solicitar a criação de um objeto sem ter conhecimento algum de sua classe concreta?
   - Que possíveis formas você enxerga para implementar isto?

Quando usar?

- Factory Method
  - Para isolar a classe concreta do produto criado da interface usada pelo cliente
- Abstract Factory
  - Para criar famílias inteiras de objetos que têm algo em comum sem especificar suas interfaces.
- Prototype
  - Para criar objetos usando outro como base
- Memento
  - Para armazenar o estado de um objeto sem quebrar o encapsulamento. O uso típico deste padrão é na implementação de operações de Undo.
- Builder
  - Para construir objetos complexos em várias etapas e/ou que possuem representações diferentes

Exercício - Qualiti Internet Banking

- Dado:
  - A arquitetura do sistema modelada no exercício anterior
- Refatorar o Projeto para:
  - Permitir flexibilidade na mudança do mecanismo de persistência (BDR ou XML)
  - Uniformizar as diversas interfaces de operadoras de cartão (visa, master, etc) no subsistema OperadoraCartão

Dica: utilize os padrões Adapter e Abstract Factory.

PADRÕES ASSOCIADOS A DISTRIBUIÇÃO DE RESPONSABILIDADES

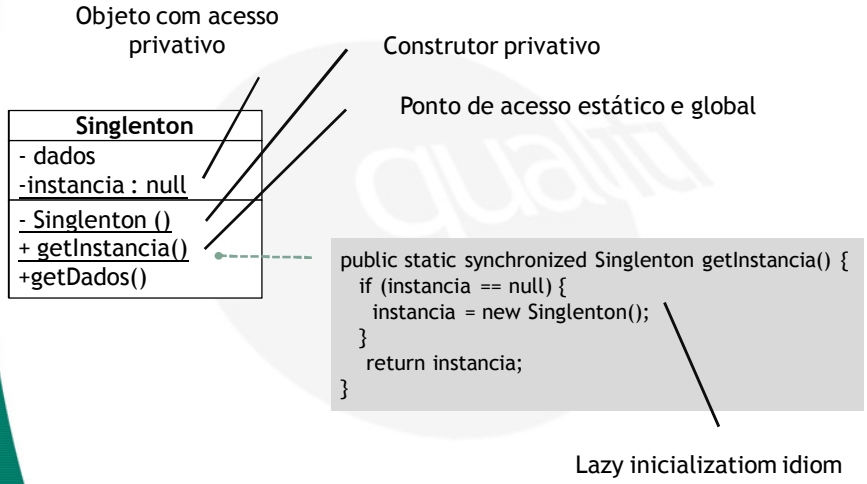
Padrões associados a Distribuição de responsabilidades

- Singleton: centraliza a responsabilidade em uma única instância de uma classe
- Observer: desacopla um objeto do conhecimento de que outros objetos dependem dele
- Mediator: centraliza a responsabilidade em uma classe que determina como outros objetos interagem
- Proxy: assume a responsabilidade de outro objeto (intercepta)
- Chain of Responsibility: permite que uma requisição passe por uma corrente de objetos até encontrar um que a processe
- Flyweight: centraliza a responsabilidade em objetos compartilhados de alta granularidade (blocos de montagem)

Singlenton

- Objetivo segundo o GoF:
  - "Garantir que uma classe só tenha uma única instância, e prover um ponto de acesso global a ela."
    - independente do número de requisições que receber para criá-lo
- Aplicações
  - Um único banco de dados
  - Um único acesso ao arquivo de log
  - Um único objeto que representa um vídeo
  - Uma única fachada (Façade pattern)

Estrutura do Singlenton

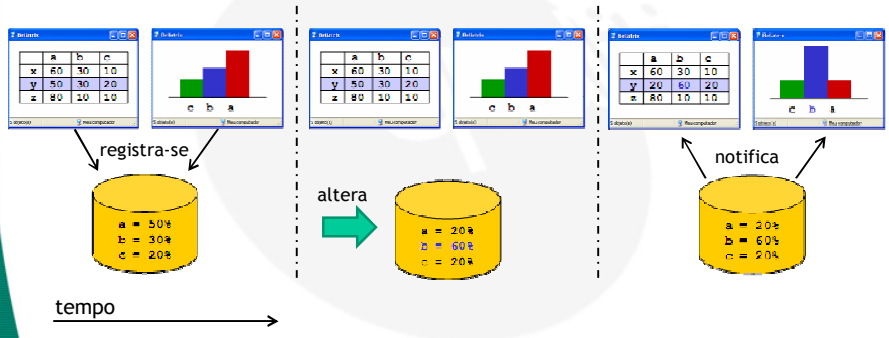


Observer

- Objetivo segundo o GoF:
  - "Definir uma dependência um-para-muitos entre objetos para que quando um objeto mudar de estado, todos os seus dependentes sejam notificados e atualizados automaticamente."

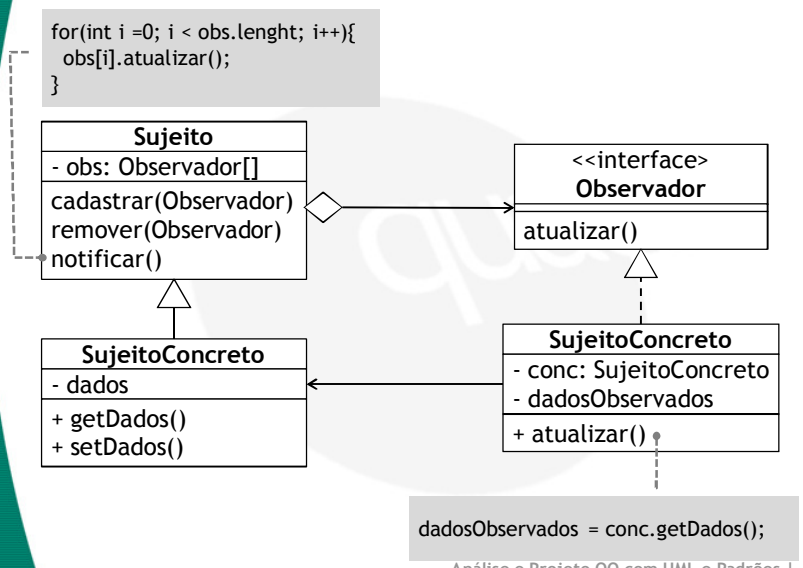
Observer

Problema:



slide com animação

Estrutura de Observer



Chain of Responsibility

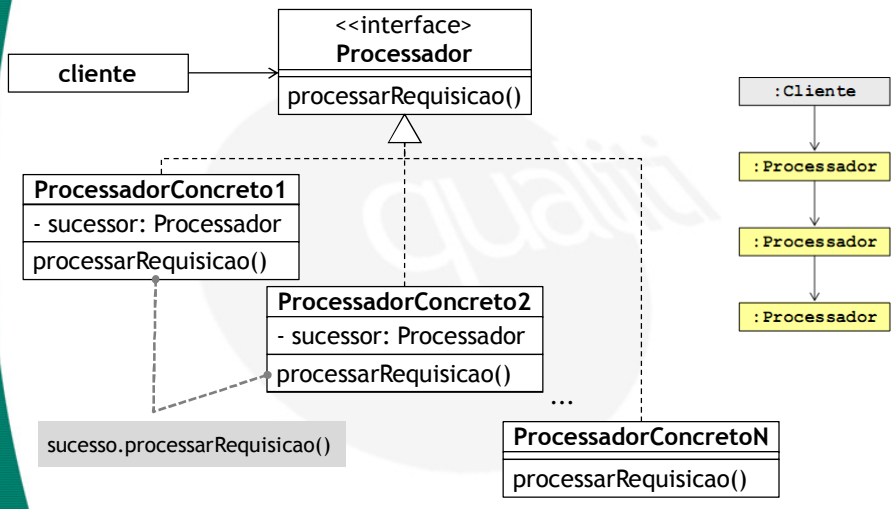
- Objetivo segundo o GoF:
  - "Evita acoplar o remetente de uma requisição ao seu destinatário ao dar a mais de um objeto a chance de servir a requisição."
  - "Compõe os objetos em cascata e passa a requisição pela corrente até que um objeto a sirva."

Chain of Responsability

- Problema
  - Permitir que vários objetos possam servir a uma requisição ou repassá-la
  - Permitir divisão de responsabilidades de forma transparente

Filme: O Grande Dave, Estrelado por Eddie Murphy

Estrutura do Chain of Responsability



Discussão

1. Qual a diferença entre:
   - Singleton e Façade?
   - Chain of Responsibility e Adapter?
2. Na sua opnião:
   - Todos os objetos deveriam ser Singlentons?
   - Singlentons estariam indiretamente introduzindo variáveis globais? Isto é bom ou ruim?
   - Que diferentes maneiras você conseguiria visualizar para indicar o próximo processo em uma cadeia de responsabilidades? Qual a diferença entre cada uma delas?

Quando usar?

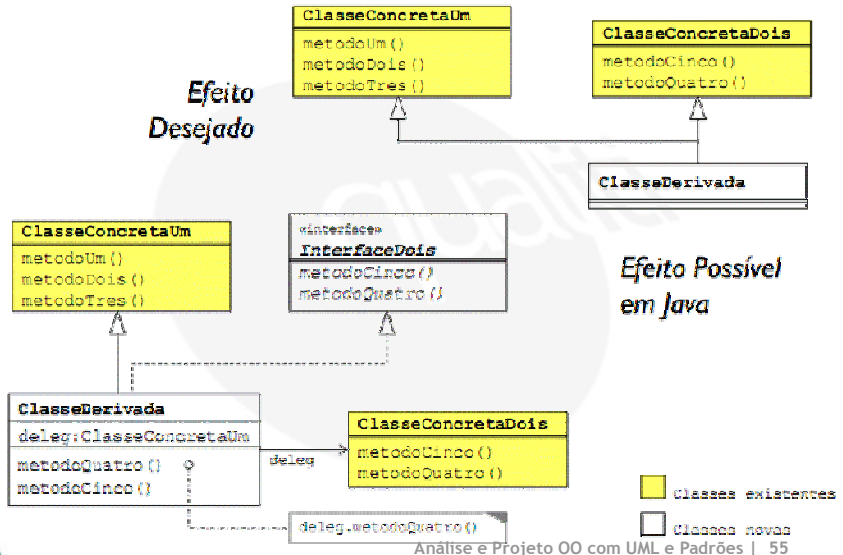
- Singleton: quando apenas uma instância for permitida
- Observer: quando houver necessidade de notificação automática
- Chain of Responsibility: quando uma requisição puder ou precisar ser tratada por um ou mais entre vários objetos
- Mediator: para controlar a interação entre dois objetos independentes
- Proxy: quando for preciso um intermediário para o objeto real
- Flyweight: quando for necessário reutilizar objetos visando performance (cuidado com o efeito oposto!)

PADRÕES QUE PERMITEM ACRESCENTAR COMPORTAMENTOS

Extensões

- Extensão é a adição de uma classe, interface ou método a uma base de código existente [2]
- Formas de extensão
  - Herança (criação de novas classes)
  - Delegação (para herdar de duas classes, pode-se estender uma classe e usar delegação para "herdar" o comportamento da outra classe)
- Desenvolvimento em Java é sempre uma forma de extensão
  - Extensão começa onde o reuso termina

Exemplo de Extensão por Delegação



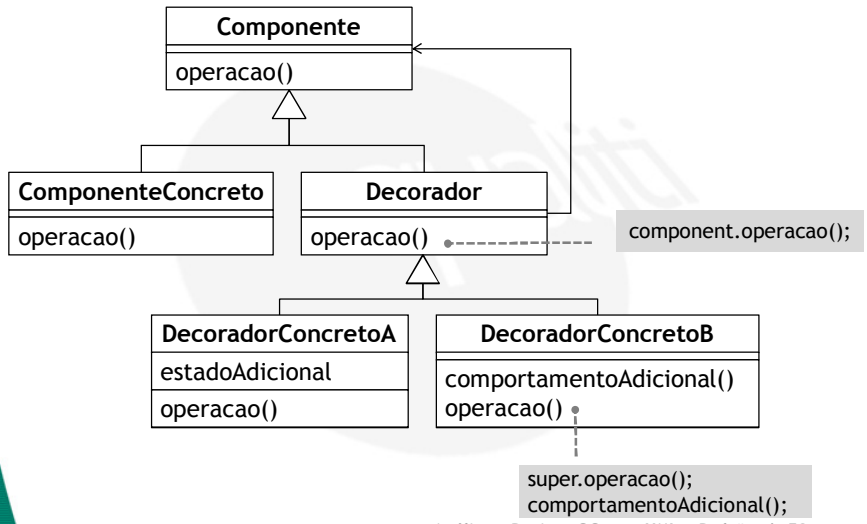
Padrões que permitem acrescentar comportamentos

- Padrões que permitem acrescentar comportamentos em um objeto sem mudar sua classe
  - Command (capítulo anterior)
  - Template Method (capítulo anterior)
  - Decorator: adiciona responsabilidades a um objeto dinamicamente.
  - Iterator: oferece uma maneira de acessar uma coleção de instâncias de uma classe carregada.
  - Visitor: permite a adição de novas operações a uma classe sem mudar a classe.

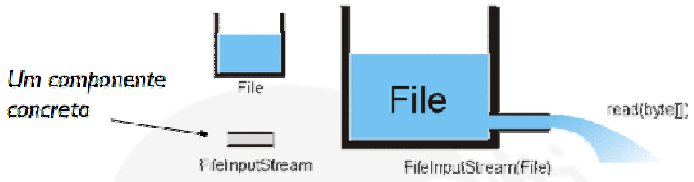
Decorator

- Objetivo segundo GoF:
  - "Anexar responsabilidades adicionais a um objeto dinamicamente."
  - "Decorators oferecem uma alternativa flexível ao uso de herança para estender uma funcionalidade."

Estrutura



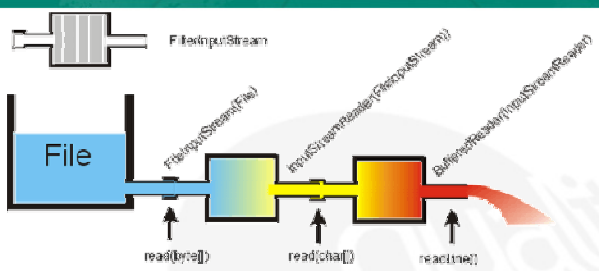
Exemplo: I/O Streams




```
// objeto do tipo File
File tanque = new File("agua.txt");

// componente FileInputStream
// cano conectado no tanque
FileInputStream cano = new FileInputStream(tanque);

// read() lê um byte a partir do cano
byte octeto = cano.read();
```


```
// partindo do cano (componente concreto)
FileInputStream cano = new FileInputStream(tanque);
// decorador leitor conectado no componente
InputStreamReader leitor = new InputStreamReader(cano);
// pode-se ler um char a partir de chf (mas isto impede que
// o char chegue ao fim da linha: há um vazamento no cano!)
char letra = leitor.read();

// decorador br conectado no decorador chf
BufferedReader leitorbf = new BufferedReader (leitor);
// lê linha de texto a de br
String linha = leitorbf.readLine();
```

Concatenação de decorador

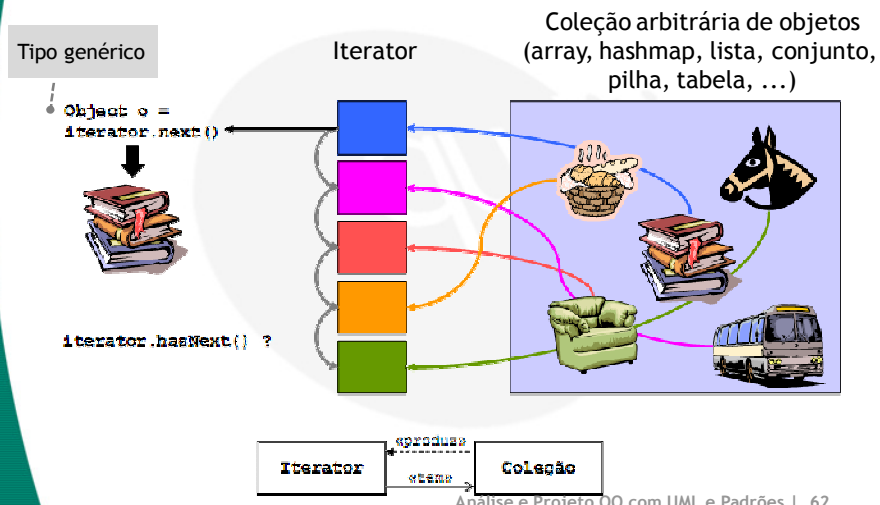
Uso de método com comportamento alterado

Comportamento adicional

Iterator

- Objetivo de acordo com o GoF:
  - "Prover uma maneira de acessar os elementos de um objeto agregado seqüencialmente sem expor sua representação interna."

Problema



Iterator

- Iterators servem para acessar o conteúdo de um agregado (coleções, arrays, etc) sem expor sua representação interna
- Oferece uma interface uniforme para atravessar diferentes estruturas agregadas
- Em Java, Iterators são implementados nas coleções do Java. É obtido através do método iterator() de Collection, que devolve uma instância de java.util.Iterator.
- Interface java.util.Iterator:
  - package java.util;
  - public interface Iterator<E> {
  - boolean hasNext();
  - Object next();
  - void remove();
  - }
- iterator() é um exemplo de Factory Method

quando usar?

- Decorator
  - Para acrescentar recursos e comportamento a um objeto existente, receber sua entrada e podermanipular sua saída.
- Iterator
  - Para navegar em uma coleção elemento por elemento
- Visitor
  - Para estender uma aplicação com novas operações sem que seja necessário mexer na interface existente.

Exercício - Qualiti Internet Banking

- Dado:
  - A arquitetura do sistema modelada no exercício anterior
- Modelar o Caso de Uso Configurar Conta:
  - Produzir diagramas de Análise
  - Introduzir caso de uso à arquitetura

Dica: utilize os padrões Composite, Observer e Decorator.

PADRÕES QUE LIDAM COM FORMAS DE IMPLEMENTAR OPERAÇÕES E ALGORITMOS

Operações
Algumas Definições

- Operação: especificação de um serviço que pode ser requisitado por uma instância de uma classe.
  - Exemplo: operação toString() é implementada em todas as classes.
- Método: implementação de uma operação. Um método tem uma assinatura.
  - Exemplo: cada classe implementa toString() diferentemente
- Assinatura: descreve uma operação com um nome, parâmetros e tipo de retorno.
  - Exemplo: public String toString()
- Algoritmo: uma seqüência de instruções que aceita entradas e produz saída. Pode ser um método, parte de um método ou pode consistir de vários métodos.

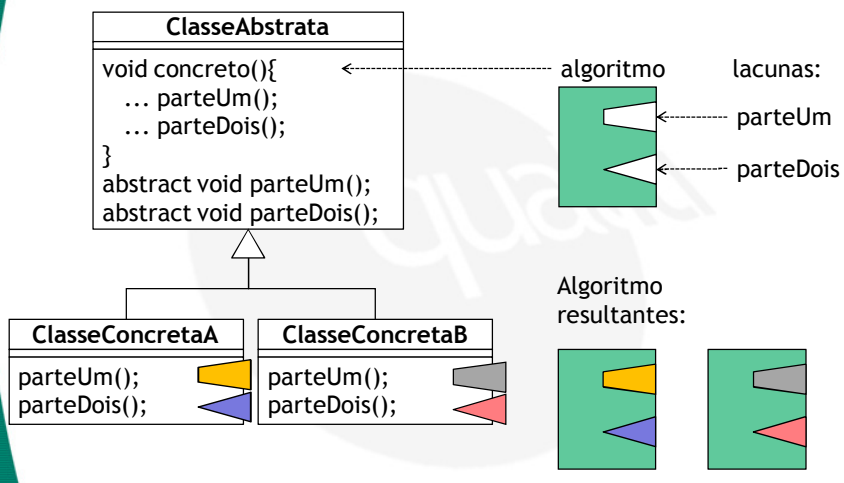
Padrões que lidam com formas de implementar operações e algoritmos

- Template Method:
  - implementa um algoritmo em um método adiando a definição de alguns passos do algoritmo para que subclasses possam defini-los
- State:
  - distribui uma operação para que cada classe represente um estado diferente; encapsula um estado
- Strategy:
  - encapsula um algoritmo fazendo com que as implementações sejam intercambiáveis
- Command:
  - encapsula uma instrução em um objeto
- Interpreter:
  - distribui uma operação de tal forma que cada implementação se aplique a um tipo de composição diferente

Template Method

- Objetivo segundo o GoF:
  - "Definir o esqueleto de um algoritmo dentro de uma operação, deixando alguns passos a serem preenchidos pelas subclasses."
  - "Template Method permite que suas subclasses redefinam certos passos de um algoritmo sem mudar sua estrutura."

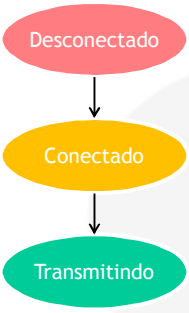
Problema/Solução



State

- Objetivo segundo o GoF:
  - "Permitir a um objeto alterar o seu comportamento quanto o seu estado interno mudar."
  - "O objeto irá aparentar mudar de classe."

Problema



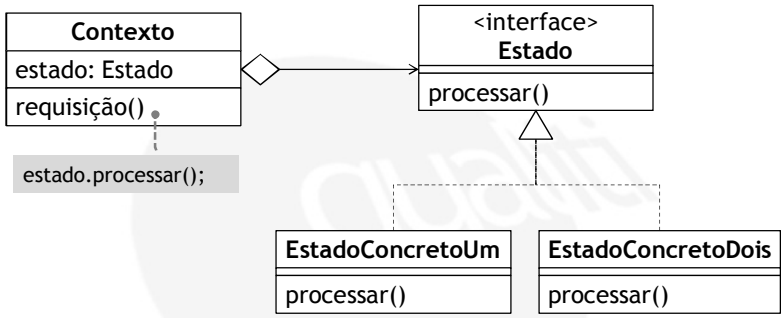
- Cenário Atual:
  - :Objeto
    - Operação

```
If(estado == desconectado){
façaIsto();
} else if (estado == conectado){
falaAquilo();
} else {
  faça();
}
```

- Desejado:
  - estado.faça();

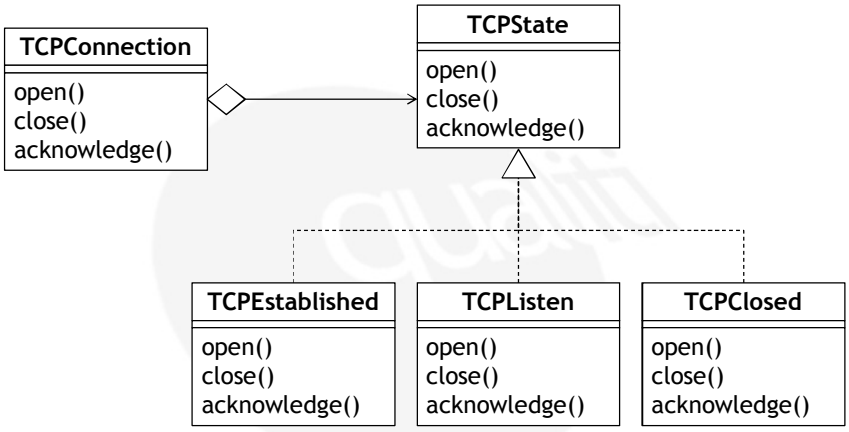
Desejamos usar objetos para representar estados e polimorfismo para tornar a execução de tarefas dependentes de estado transparentes

Estrutura



- Contexto: define uma interface de interesse ao cliente, delegando suas requisições ao estado corrente.
- Estado: define uma interface para encapsular o comportamento de todos os estados.
- EstadoConcreto: implementa o comportamento associado ao estado do contexto.

Exemplo

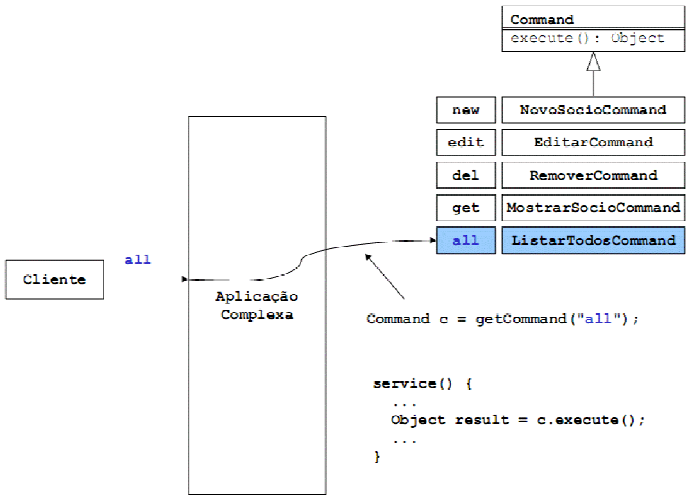


Sempre que a aplicação mudar de estado, o objeto TCPConnection muda o objeto TCPState que está usando

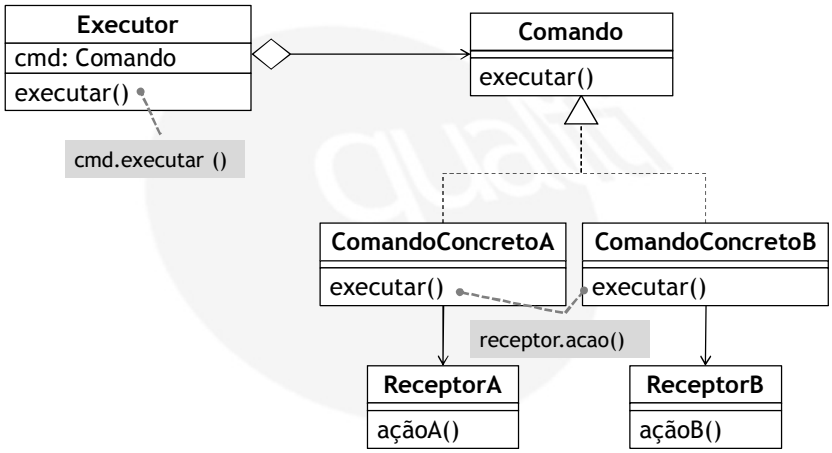
Command

- Objetivo segundo GoF:
  - "Encapsular uma requisição como um objeto, permitindo que clientes parametrizem diferentes requisições, filas ou equisições de log, e suportar operações reversíveis."

Problema



Estrutura



Discussão

1. Qual a diferença entre State e Command?
2. Sobre State, você consegue imagina alguma implementação alternativa que:
   - Evite que cada estado concreto tenha que implementar métodos não suportados neste estado;
   - O contexto não esta ciente das transições entre estados.

quando usar?

- Template Method
  - Para **compor um algoritmo** feito por métodos abstratos que podem ser completados em subclasses
- State
  - Para representar o **estado** de um objeto
- Command
  - Para representar um **comando** (ação imperativa do cliente)
- Strategy
  - Para representar um algoritmo (**comportamento**)
- Interpreter
  - Para realizar composição com comandos e **desenvolver uma linguagem** de programação usando objetos

Exercício - Qualiti Internet Banking

- Dado:
  - A arquitetura do sistema modelada no exercício anterior
- Modelar o Caso de Uso de Agendamento:
  - Produzir diagramas de Análise
  - Introduzir caso de uso à arquitetura

Dica: utilize o padrão Command em sua resposta