



Padrões de Projeto

1

Introdução



- Projetar software orientado a objetos é diferente de software **reutilizável** orientado a objetos
- Um programa deve ser específico para o problema a ser resolvido, mas também genérico o suficiente para atender futuros problemas e requisitos
- Projetistas orientados a objetos experientes dizem que um projeto reutilizável e flexível é difícil de se obter da primeira vez, e que novatos frequentemente tendem a recair em técnicas não-orientadas a objetos. O que é que os experientes sabem e que os novatos ainda não sabem?

2

2



Introdução

- Uma coisa que os projetistas avançados sabem que não devem fazer é resolver cada problema do zero
- Ao invés disso, eles reutilizam soluções que funcionaram no passado. Quando encontram uma boa solução, eles a utilizam repetidamente. Consequentemente, criam-se **padrões**
- Esses padrões resolvem problemas específicos de projetos e tornam os projetos orientados a objetos mais flexíveis e reutilizáveis. Alguém que está familiarizado com tais padrões pode aplicá-los imediatamente a problemas de projeto, sem a necessidade de redescobri-los

3

3



Introdução

- Analogia: Romanistas e roteiristas raramente projetam suas tramas do zero. Ao invés disso, seguem padrões, como “o herói tragicamente problemático”, ou “a novela romântica”.
- Do mesmo modo, projetistas orientados a objetos seguem padrões como “represente estados como objetos” e “decore objetos de maneira que possa facilmente acrescentar/remover características”. Uma vez que você conhece o padrão, uma grande quantidade de decisões de projeto decorre automaticamente

4

4



Introdução

•Costuma ocorrer aquela sensação de "déjà vu" quando se está fazendo um projeto e você já resolveu um problema parecido antes. Mas normalmente não fazemos um bom trabalho em registrar as soluções que usamos no passado

•Os **padrões de projetos**(design patterns) são apenas isso, soluções reutilizáveis de softwares orientados a objetos já registradas, documentadas e testadas

•Eles ajudam a escolher alternativas de projeto que tornam um sistema reutilizável. Melhoram também a documentação e a manutenção de sistemas, e permitem a comunicação com outros projetistas em nível de projeto, e não de "linhas de código"

5

5



O que são Padrões de Projeto

•Já usados a muito tempo pelas engenharias, mas apenas recentemente incorporados de modo formal à engenharia de software(1995)

•“Cada padrão descreve um problema do nosso ambiente e o núcleo de sua solução, de tal forma que você possa usar esta solução mais de um milhão de vezes, sem nunca fazê-lo da mesma maneira”

•Possuem quatro elementos essenciais:

- Nome do padrão
- Problema resolvido
- Solução
- Consequências da aplicação

6

6



O que são Padrões de Projeto

• Mas na literatura, um padrão é composto por muito mais:

- Nome do padrão
- Classificação
- Intenção
- Motivação
- Aplicabilidade
- Estrutura
- Participantes
- Colaborações
- Consequências da aplicação
- Implementação
- Amostras de código
- Usos conhecidos
- Padrões relacionados

7

7



O que são Padrões de Projeto

• São parecidos com padrões de arquitetura, mas são aplicados num nível mais baixo; os padrões de arquitetura de software determinam um padrão que é seguido pelo sistema inteiro

• Padrões de projeto não apenas tornam a vida do programador mais fácil, uma vez que existe um repositório com boas soluções para se resolver uma série de problemas, mas eles também independem de linguagem, podendo ser usados com qualquer linguagem orientada a objetos

• Será apresentado um breve histórico sobre padrões de projeto, e na sequência, veremos os padrões mais utilizados

8

8



Histórico

- Em 1987, os programadores Kent Beck e Ward Cunningham propuseram os primeiros padrões de projeto para a área da ciência da computação, em um trabalho para a conferência OOPSLA
- Eles apresentaram alguns padrões arquiteturais para o desenvolvimento na linguagem Smalltalk
- Esse trabalho alcançou certa notoriedade, mas ainda faltava muito para a ideia de padrões se desenvolver entre os profissionais de computação

9

9



Histórico

- Começaram a ganhar popularidade com o livro *Design Patterns: Elements of Reusable Object-Oriented Software*, de 1995, que focava em C++. Foram apresentados padrões influencias, que são usados até hoje
- Os autores do livro foram Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, conhecidos como a "Gangue dos Quatro" (Gang of Four) ou simplesmente "GoF"
- No mesmo ano foi criado o Portland Pattern Repository, cuja função é guardar o "estado-da-arte" em padrões de projeto. Esse repositório está acompanhado da WikiWikiWeb, que foi a primeira wiki do mundo

10

10



Histórico

- Em 2005, foi publicado o livro *Applying UML and Patterns – An Introduction to Object-Oriented Analysis and Design and Iterative Development*, de Craig Larman
- O livro apresenta os padrões GRASP (General Responsibility Assignment Software Patterns), que hoje são mais considerados “boas práticas de engenharia de software orientada a objetos” do que padrões que podem ou não ser aplicados
- Entre esses padrões, temos Baixo Acoplamento, Alta Coesão, Polimorfismo e Controlador (usado, por exemplo, na arquitetura MVC, Model-View-Controller, das interfaces gráficas em Smalltalk)

11

11



Histórico

- Em 2009, mais de 30 pessoas colaboraram com Thomas Erl em seu livro, *SOA Design Patterns*. O objetivo desse livro era estabelecer o catálogo absoluto para padrões de projeto em arquiteturas orientadas à serviço
- Mais de 200 profissionais de TI no mundo fizeram revisões e avaliações sobre o livro de Erl e sobre seus padrões. Esses mesmos padrões foram também publicados e discutidos no site de pesquisa comunitária soapatterns.org

12

12

Padrões GoF

•São 23 padrões de projeto, divididos em três famílias de padrões:

- Criacionais - relacionados à criação de objetos;
- Estruturais - tratam das associações entre classes e objetos;
- Comportamentais – tratam das interações e divisões de responsabilidades entre as classes ou objetos.

•Veremos uma breve explicação de cada um, e mais tarde serão vistos exemplos de aplicação de alguns deles em situações reais, em código Java, para facilitar o entendimento

13

13

Padrões GoF

		PROPÓSITO		
		Criação	Estrutura	Comportamento
ESCOPO	Classe	Factory Method	Adapter (classe)	Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Adapter (objeto) Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

14

14



Padrões GoF - Criacionais

- **Abstract Factory:** Fornece uma interface para criação de famílias de objetos relacionados ou dependentes sem especificar suas classes concretas.
- **Builder:** Separa a construção de um objeto complexo da sua representação, de modo que o mesmo processo de construção possa criar diferentes representações.
- **Factory Method:** Define uma interface para criar um objeto, mas deixa as subclasses decidirem qual a classe a ser instanciada. O Factory Method permite postergar a instanciação às subclasses.

15

15



Padrões GoF - Criacionais

- **Prototype:** Especifica os tipos de objetos a serem criados usando uma instância prototípica e cria novos objetos copiando este protótipo.
- **Singleton:** Garante que uma classe tenha somente uma instância e fornece um ponto de acesso global para ela.

16

16



Padrões GoF - Estruturais

- Adapter: Converte a interface de uma classe em outra interface esperada pelos clientes. O Adapter permite que certas classes trabalhem em conjunto, pois de outra forma seria impossível por causa de suas interfaces incompatíveis.
- Bridge: Separa uma abstração da sua implementação, de modo que as duas possam variar independentemente.
- Composite: Compõe objetos em estrutura de árvore para representar hierarquias do tipo partes-todo. O Composite permite que os clientes tratem objetos individuais e composições de objetos de maneira uniforme.

17

17



Padrões GoF - Estruturais

- Decorator: Atribui responsabilidades adicionais a um objeto dinamicamente. Os Decorators fornecem uma alternativa flexível a subclasses para extensão da funcionalidade.
- Façade: Fornece uma interface unificada para um conjunto de interfaces em um subsistema. O Façade define uma interface de nível mais alto que torna o subsistema mais fácil de usar.
- Flyweight: Usa compartilhamento para suportar grandes quantidades de objetos, de granularidade fina, de maneira eficiente.

18

18



Padrões GoF - Estruturais

- Proxy: Fornece um objeto representante(surrogate), ou um marcador de outro objeto, para controlar o acesso ao mesmo.

19

19




Padrões GoF - Comportamentais

- Chain of Responsibility: Evita o acoplamento do remetente de uma solicitação ao seu destinatário, dando a mais de um objeto a chance de tratar a solicitação. Encadeia os objetos receptores e passa a solicitação ao longo da cadeia até que um objeto a trate.
- Command: Encapsula uma solicitação como um objeto, desta forma permitindo que você parametrize clientes com diferentes solicitações, enfileire ou registre(log) solicitações e suporte operações que possam ser desfeitas.
- Interpreter: Dada uma linguagem, define uma representação para sua gramática juntamente com um interpretador que usa a representação para interpretar sentenças nesta linguagem.

20

20




Padrões GoF - Comportamentais

- Iterator: Fornece uma maneira para acessar sequencialmente os elementos de um objeto agregado sem expor sua representação subjacente.
- Mediator: Define um objeto que encapsula como um conjunto de objetos interage. O Mediator promove o acoplamento fraco ao evitar que os objetos se refiram explicitamente uns aos outros, permitindo que você varie suas interações independentemente.
- Memento: Sem violar o encapsulamento, captura e externaliza o estado interno de um objeto, de modo que o mesmo possa posteriormente ser restaurado para este estado.

21

21




Padrões GoF - Comportamentais

- Observer: Define uma dependência um-para-muitos entre objetos, de modo que, quando um objeto muda de estado, todos os seus dependentes são automaticamente notificados e atualizados.
- State: Permite que um objeto altere seu comportamento quando seu estado interno muda. O objeto parecerá ter mudado sua classe.
- Strategy: Define uma família de algoritmos, encapsula cada um deles e torna-os intercambiáveis. O Startegy permite que o algoritmo varie independentemente dos clientes que o utilizam.

22

22



Padrões GoF - Comportamentais

- Template Method: Define o esqueleto de um algoritmo em uma operação, postergando a definição de alguns passos para subclasses. O Template Method permite que as subclasses redefinam certos passos de um algoritmo sem mudar sua estrutura.
- Visitor: Representa uma operação a ser executada sobre os elementos da estrutura de um objeto. O visitor permite que você defina uma nova operação sem mudar as classes dos elementos sobre os quais opera.

23

23



Padrões J2EE

- Introduzidos no livro “Core J2EE Patterns: Best Practices and Design Strategies”, de Deepak Alur, John Crupi e Dan Malks
- Feitos especialmente para Java Enterprise Edition (JEE), buscando produzir aplicações robustas web voltadas para empresas, mas alguns são universalmente aplicáveis
- Ajudam a preencher o grande vácuo entre as abstrações e os serviços providenciadas pelo JEE e a aplicação final que a equipe deve construir. Aplicando esses padrões, escreve-se menos software, e por isso corre-se menos riscos de software.
- “(Esse livro) é como ter uma equipe de especialistas sentando ao seu lado.”* Grady Booch, chefe cientista da Rational

24

24



Padrões J2EE

- Composite View: Forma uma “view”(camada de visão) composta de múltiplas “subviews” atômicas. Cada componente pode ser incluído dinamicamente e o layout geral pode ser gerenciado independentemente do conteúdo.
- Business Delegate: Esconde os detalhes da implementação, como detalhes de consulta e de acesso da arquitetura, dos serviços de negócio. Isso reduz o acoplamento entre clientes de apresentação e serviços de negócio.
- Transfer Object Assembler: Usado para construir o modelo desejado, após ter usado Transfer Objects para recuperar dados das várias partes do sistema.

25

25



Padrões J2EE

- Composite Entity: Representa um grafo de objetos, permitindo que se modele, represente e gerencie um conjunto de objetos persistentes não-relacionados.
- Transfer Object (TO): Serve para encapsular dados de negócio, contidos em um enterprise bean, para que uma única chamada de método possa enviar ou buscar esses dados.
- Data Access Object (DAO): Tem como função abstrair e encapsular todo o acesso à fonte de dados. O DAO gerencia a conexão com a fonte de dados para obter e armazenar dados.

26

26



Padrões J2EE

- **Intercepting Filter:** Cria filtros conectáveis para se processar serviços comuns de forma padrão, sem precisar mudar o código de processamento. Os filtros interceptam requisições e respostas, permitindo pré e pós-processamento.
- **Front Controller:** Usa um controlador como ponto inicial para tratamento de requisições, revogando serviços de segurança como autenticação, delegando processamento de negócio, gerenciando a escolha de uma view apropriada, tratando erros, e mais.
- **View Helper:** Classes auxiliares, normalmente JavaBeans, para as quais uma view delega suas responsabilidades de processamento. Também guardam o modelo de dados da view. 27

27



Padrões J2EE

- **Dispatcher View:** Combina um Front Controller e um “Dispatcher”, com views e view helpers, para tratar requisições de cliente e preparar uma apresentação dinâmica como resposta. Um “Dispatcher” é responsável por gerenciar views e pela navegação no sistema.
- **Service to Worker:** Similar à Dispatcher View, mas o controlador dessa vez delega a recuperação de conteúdo de negócio aos Helpers.
- **Session Facade:** Usa um bean de sessão como Facade para encapsular a complexidade das interações entre os objetos de negócio participantes, providenciando uma interface uniforme de serviços aos clientes.

28

28



Padrões J2EE

- Service Locator: Permite abstrair o uso de JNDI. Múltiplos clientes podem reusá-lo para reduzir a complexidade do código, prover um ponto único de controle, e aumentar performance pelo uso de cache.
- Value List Handler: Controla a busca, cacheia os resultados, e provê os resultados ao cliente num "result set" cujo tamanho e método de iteração atendem aos requisitos do cliente.
- Service Activator: Receba mensagens assíncronas de clientes. Recebendo uma mensagem, o Service Activator localiza e invoca os métodos de negócio necessários para preencher a requisição assincronamente.

29

29




Outros Padrões

- Mencionados e aplicados em várias literaturas, e em especial, bastante evidenciados no livro "Expert One-on-One: JEE Design e Development", de Rod Johnson
- O livro elabora sobre o Spring Framework, um framework open source Java, não intrusivo, fortemente baseado em padrões de projeto. O Spring permite o uso de arquivos XML para configurar as dependências entre classes de um software
- Os padrões de projeto orientado a objetos que serviram de base ao Spring são bastante gerais e úteis

30

30




Outros Padrões

- **Dependency Injection:** É um padrão utilizado quando é necessário manter baixo o nível de acoplamento em um sistema. Com a DI, as dependências entre as classes são definidas pela configuração de uma infraestrutura de software (container), que é responsável por "injetar" em cada classe suas dependências declaradas.
- **Inversion of Control:** A sequência de chamadas dos métodos é invertida em relação à programação tradicional, ou seja, não é definida pelo programador. Este controle é delegado a uma infraestrutura de software (muitas vezes chamada de container) ou a qualquer outro componente que possa tomar controle sobre a execução.

31

31



Outros Padrões

- **MVC:** O MVC (Model-View-Controller) é um padrão arquitetural de alto nível, que visa separar a lógica de negócio da lógica de apresentação, permitindo o desenvolvimento, teste e manutenção isolado de ambos. Combina os padrões de projeto Observer, Strategy e Composite. O MVC é utilizado desde 1979, quando foi implementado na linguagem orientada a objetos Smalltalk.

32

32



Padrão Singleton

- Imagine um aplicativo que simula tráfego urbano, usando Threads. Cada objeto Rua tem um nome e uma Rua destino, e a cada X segundos cria um objeto Carro e o transfere para a Rua destino. Dessa forma, os carros passam a impressão de que estão circulando pelo sistema.
- Mas se nós criarmos uma instância de uma determinada Rua para podermos simulá-la, e também criarmos uma instância dessa mesma Rua como destino de outra, os carros que forem adicionados nesta rua serão refletidos em TODAS as instâncias dessa mesma Rua?

33

33



Padrão Singleton

```
public class RuaAntonioDoAmaral extends Rua {
    public RuaAntonioDoAmaral(){
        super();
        nome = "Rua Antonio do Amaral";
        destino = new AvenidaBrasil();
        taxaCarrosPorSegundo = (float)0.5;
    }
}

public class TesteCarros {
    public static void main(String[] args){
        Rua antonio = new RuaAntonioDoAmaral();
        Rua brasil = new AvenidaBrasil();
        Rua gaivotas = new RuaDasGaivotas();
    }
}
```

- Não. Como cada instância de uma classe é diferente da outra, um carro adicionado numa instância da Rua Avenida Brasil não seria adicionado na outra.

34

34



Padrão Singleton

```
public class AvenidaBrasil extends Rua {
    private static AvenidaBrasil avenida = null;
    private AvenidaBrasil() {
        nome = "Avenida Brasil";
        destino = new AvenidaMarginalLeste();
        taxaCarrosPorSegundo = 4;
    }
    public synchronized static AvenidaBrasil getInstance() {
        if(avenida == null) avenida = new AvenidaBrasil();
        return avenida;
    }
}
```

```
public class TesteCarros {
    public static void main(String[] args) {
        Thread[] ruas = new Thread[3];
        Rua antonio = new RuaAntonioDoAmaral();
        Rua brasil = AvenidaBrasil.getInstance();
        Rua gaivotas = new RuaDasGaivotas();
    }
}
```

•Ao invés de instanciar o objeto usando new, deve-se obter uma instância do objeto através do método estático getInstance(). O construtor deve ser privado, e esse método, sincronizado. 35

35



Padrão Singleton

```
public class ConexaoBD {
    private static Connection conexao = null;
    private static String url = "jdbc:postgresql://localhost:5432/db";
    private static String username = "usuario";
    private static String password = "senha";

    private ConexaoBD() {} // Construtor privado

    public static synchronized Connection getConexao() {
        if (conexao == null) {
            try {
                Class.forName("org.postgresql.Driver");
                conexao = DriverManager.getConnection(url, username, password);
            } catch (Exception e) {
                System.err.println("Não foi possível conectar.");
                return null;
            }
        }
        return conexao;
    }
}
```

•Singletons são bem úteis também quando falamos de banco de dados, para evitar ter de se abrir uma conexão cada vez que for feita uma consulta ou inserção. 36

36



Padrão Façade

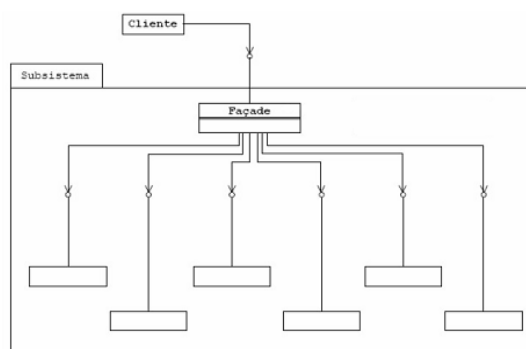
- Lidar com projetos de software grandes sem utilizar a noção de subsistemas é muito difícil. Frequentemente nos deparamos com situações em que devemos manipular um grande número de objetos diferentes, e é inconveniente guardar várias referências de objetos na classe principal somente para poder executar uma funcionalidade específica.
- O padrão Façade tem como finalidade prover uma interface de alto nível para facilitar a utilização de um subsistema. Este padrão é extremamente útil, pois a aplicação não precisa saber a implementação de seus subsistemas, apenas saber como fazer chamadas para as funcionalidades dos mesmos. Veremos alguns exemplos na sequência.

37

37



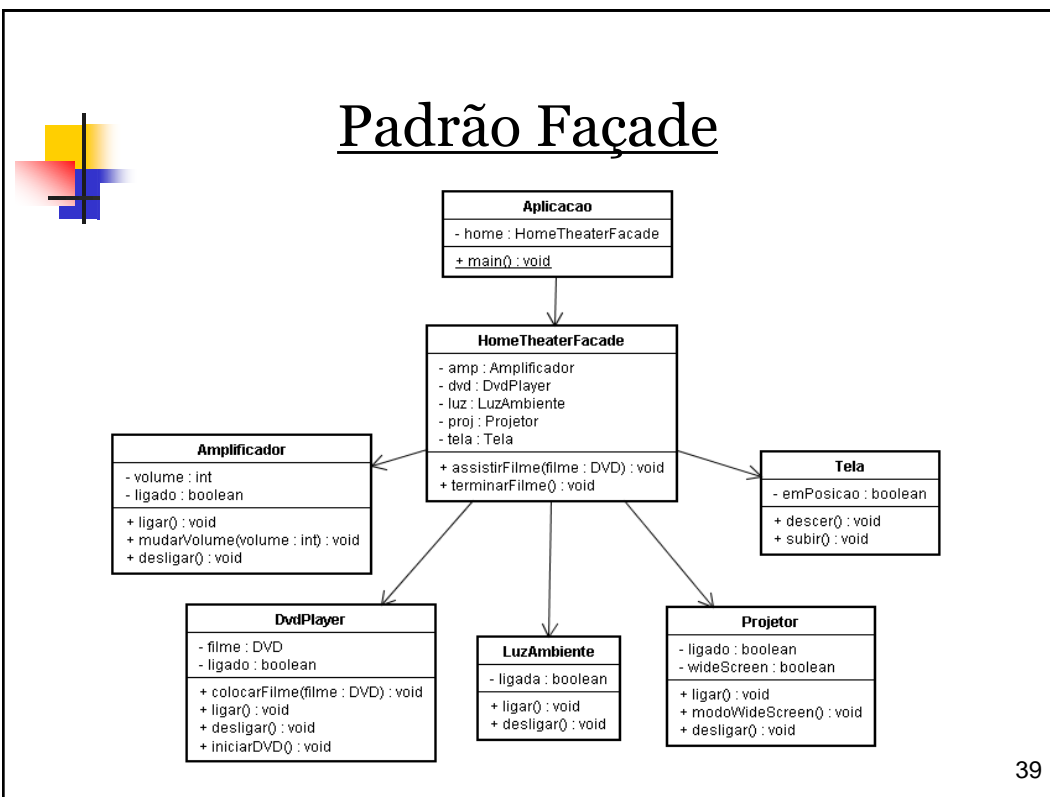
Padrão Façade



Um Façade diminui o acoplamento e torna o projeto mais organizado e flexível

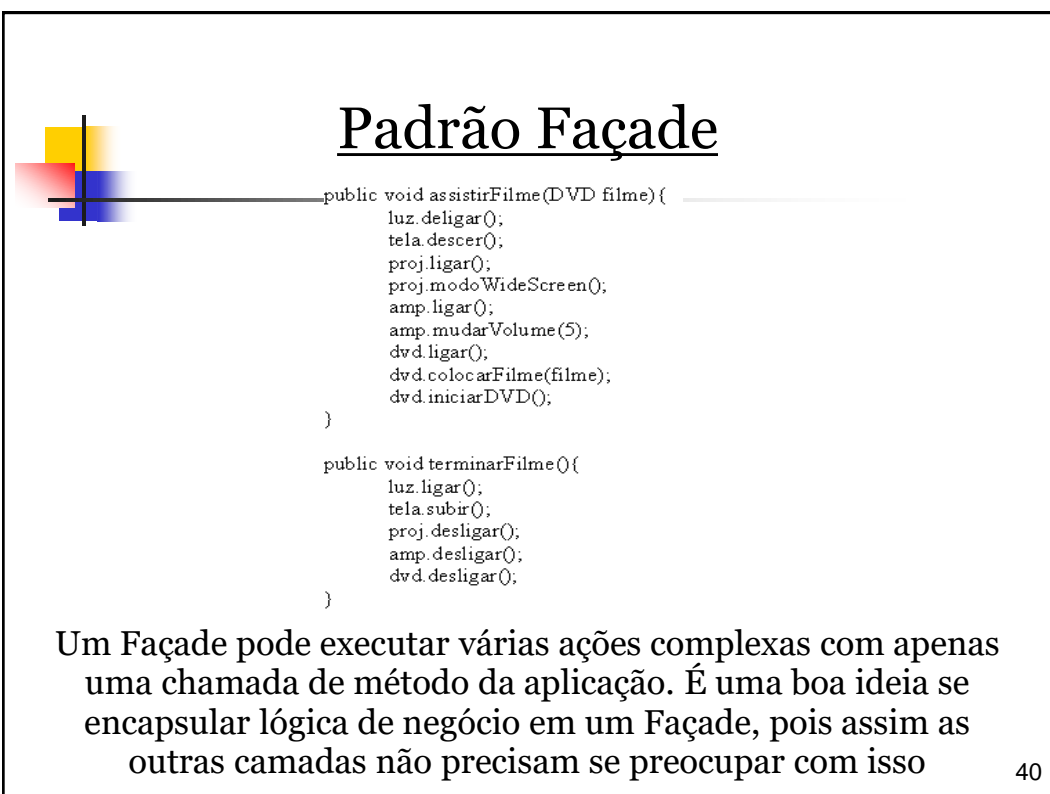
38

38



39

39



40

40



Padrão Iterator

- Acesso sequencial em estruturas de dados é uma operação trivial e usada frequentemente, mas um projetista de software muitas vezes não deve se preocupar com o exato tipo de coleção sendo usada
- Um Iterator é um objeto que permite o acesso a uma estrutura de dados sem expor sua estrutura interna, ou seja, através de métodos padronizados `hasNext()` e `next()`, permite iterar sobre uma coleção de maneira uniforme
- Iterators já são implementados no Java

41

41

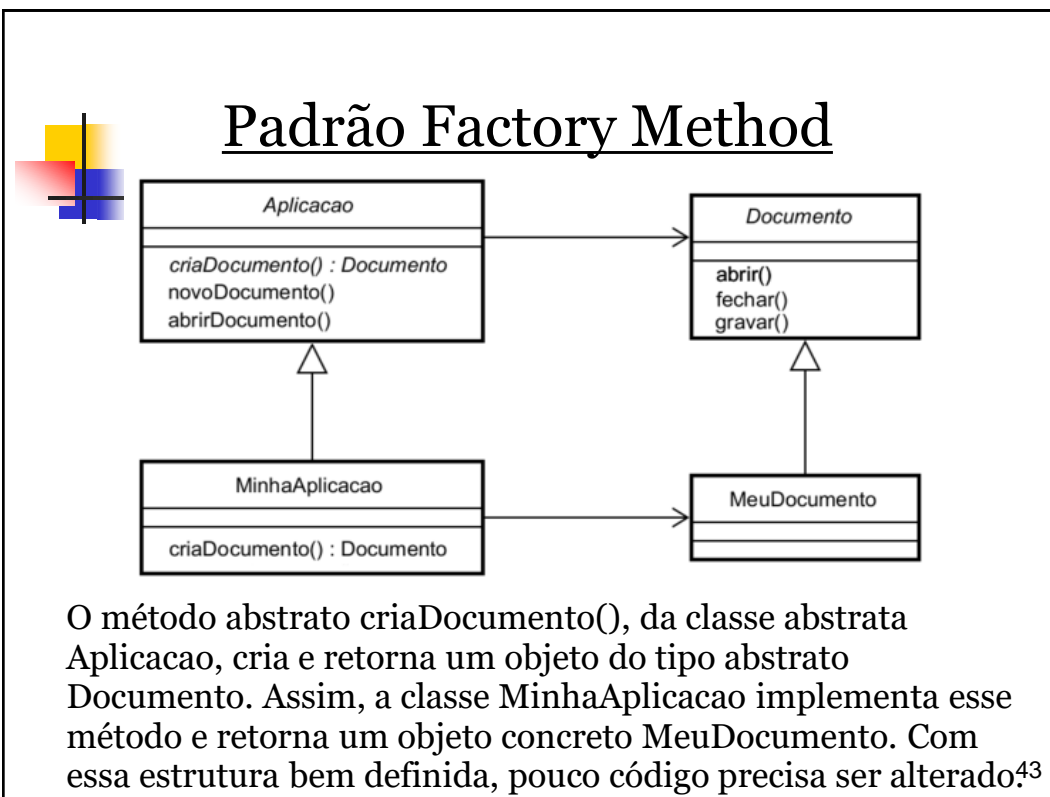


Padrão Factory Method

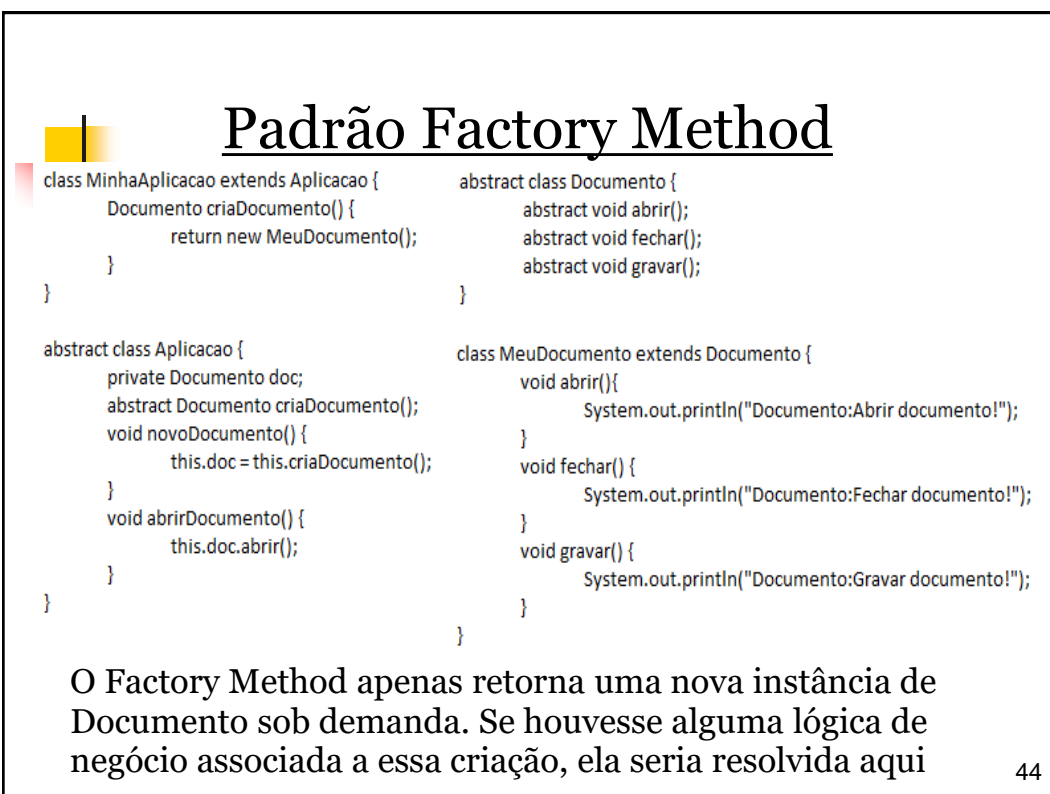
- Imagine o desenvolvimento de uma aplicação que permita a criação de documentos de texto. Durante a engenharia deste projeto, o formato do documento mudou constantemente devido a diversos fatores, como maior facilidade de se gerar arquivos de um formato específico. Isso significaria que boa parte do código teria que ser reescrito para acomodar essa nova escolha de formato?
- Se ela for criada usando Factory, não. Assim, a aplicação seria construída através de um framework baseado neste padrão, suportando a criação de documentos do tipo `MeuDocumento`. O framework é constituído pelas classes abstratas `Aplicacao` e `Documento`, a aplicação disponibiliza as classes concretas `MinhaAplicacao` e `MeuDocumento`.

42

42



43



44

44



Padrão Abstract Factory

• A grande diferença entre Factory Methods e Abstract Factories é que numa Factory o programador escolhe o tipo de objeto a ser instanciado; numa AbstractFactory, só se define a superclasse do objeto a se obter, e não a classe concreta.

• Um uso para esse padrão seria na implementação de um toolkit que disponibilize widgets (componentes gráficos) que funcionem em diferentes interfaces gráficas, tal como Motif ou Qt. Estas GUIs possuem diferentes padrões de controles visuais, e por isso é interessante que se defina interfaces comuns de acesso, independentemente da GUI utilizada. O aplicativo, ou "cliente", interage com o toolkit através das classes abstratas, sem ter conhecimento da implementação das classes concretas.

45

45



Padrão Abstract Factory

```
abstract class WidgetFactory{
    public static WidgetFactory obterFactory(){
        if(Configuracao.obterInterfaceGraficaAtual() == Configuracao.MOTIFWIDGET){
            return new MotifWidgetFactory();
        } else {
            return new QtWidgetFactory();
        }
    }
    public abstract Botao criarBotao();
}

class MotifWidgetFactory extends WidgetFactory{
    public Botao criarBotao(){
        return new BotaoMotif();
    }
}

class QtWidgetFactory extends WidgetFactory{
    public Botao criarBotao() {
        return new BotaoQt();
    }
}

public class Cliente{
    public static void main(String[] args){
        WidgetFactory factory = WidgetFactory.obterFactory();
        Botao botao = factory.criarBotao();
        botao.desenhar();
    }
}
```

A Abstract Factory WidgetFactory retorna uma Factory diferente para cada interface gráfica, e a aplicação simplesmente chama o método desenhar() da superclasse

46

46

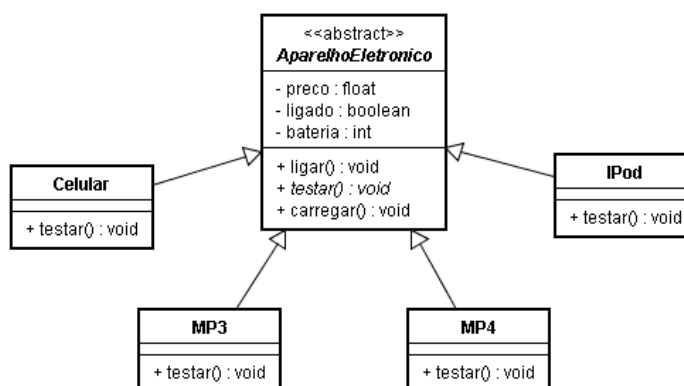
Padrão Strategy

- Suponha que uma loja inovadora de aparelhos eletrônicos permite seus clientes entrem, liguem, testem e carreguem os aparelhos livremente, e só depois decidam se vão comprar
- Para garantir que esse modelo de loja onde um cliente “pode tudo” funcione, os funcionários usam um simulador digital para analisar o possível comportamento dos clientes e do sistema de atendimento dessa loja. Esse simulador foi feito em Java com orientação a objetos
- Agora, o que aconteceria se a loja passasse a disponibilizar também aparelhos eletrônicos com touchscreen? Ora, o simulador deveria também permitir que os clientes virtuais experimentem a tela ao toque. Logo...

47

47

Padrão Strategy

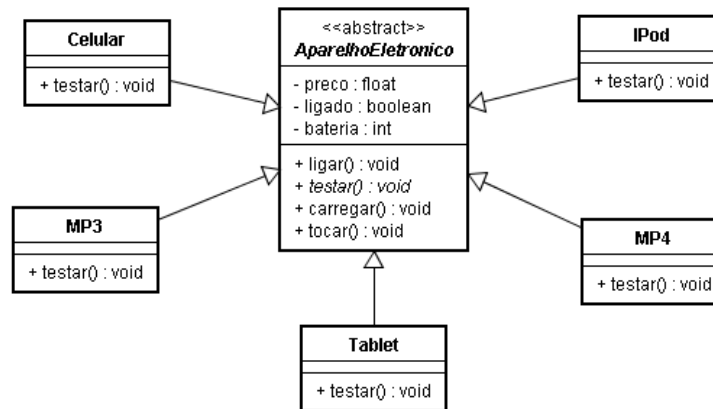


Modelo antigo

48

48

Padrão Strategy



Modelo novo

49

49

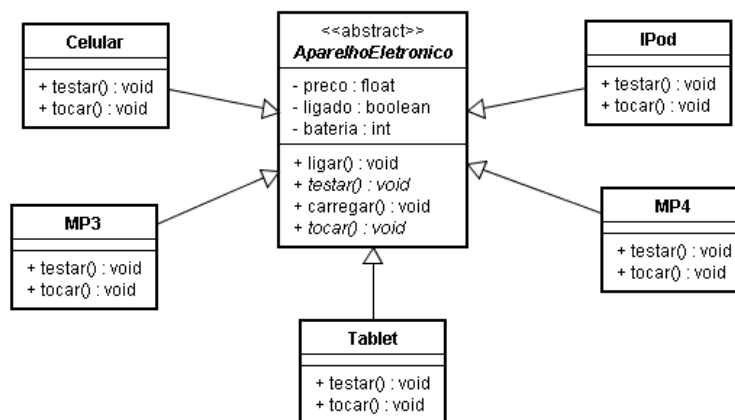
Padrão Strategy

- Mas essa solução realmente funciona?
- Sim, ela resolve o problema, mas cria outro. Agora que a superclasse abstrata **AparelhoEletronico** possui o método `tocar()` implementado, TODAS as especializações de **AparelhoEletronico** agora respondem a comandos de tela ao toque. Isso pode ser até aceitável para alguns tipos de **Celular** ou **Ipod**, mas e para um **MP3**?
- Outra solução seria deixar a implementação para as subclasses. Como ficaria o diagrama de classes se fosse feito dessa forma?

50

50

Padrão Strategy



Esta solução é boa?

51

51

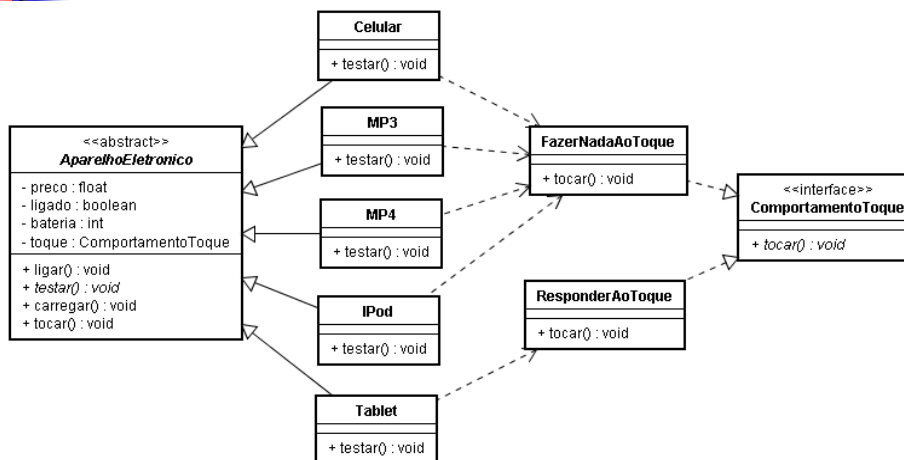
Padrão Strategy

- O método `tocar()` nos aparelhos sem tela ao toque tem implementação vazia, ou seja, não faz nada, enquanto que no Tablet, opera o aparelho
- Esta solução não é prática se existirem 50 ou mais subclasses de **AparelhoEletronico**. O método `tocar()` terá que ser implementado para todas essas classes individualmente!
- Então o que fazer? É aí que entra o padrão Strategy, que define uma família de algoritmos para determinar o comportamento de um objeto, e permite mudá-lo dinamicamente

52

52

Padrão Strategy



Solução com Strategy. Agora veremos isso em código, para mostrar que não é tão difícil quanto possa parecer

53

53

Padrão Strategy

```

public interface ComportamentoToque {
    public void tocar();
}

public class FazerNadaAoToque implements ComportamentoToque {
    public void tocar() {
        //Não faz nada
    }
}

public class ResponderAoToque implements ComportamentoToque {
    public void tocar() {
        System.out.println("Algo está acontecendo!"); //Faz algo
    }
}

public class Celular extends AparelhoEletronico {
    public Celular() {
        toque = new FazerNadaAoToque();
    }
    public void testar() {
        System.out.println("Testando celular...");
    }
}

public class Tablet extends AparelhoEletronico {
    public Tablet() {
        toque = new ResponderAoToque();
    }
    public void testar() {
        System.out.println("Testando tablet...");
    }
}
  
```

Desta forma, o comportamento de um **AparelhoEletronico** quando ele lida com um toque é determinado pela instância de **ComportamentoToque** dele. O código de um comportamento específico só é escrito uma vez, é reutilizável, e é só atribuir uma nova instância ao atributo `toque` para mudá-lo

54

54



Padrão Observer

• Imagine desenvolver um software que gerencia informações meteorológicas vindas de um instituto. O instituto lhe fornece uma classe WeatherData, que possui um método medicaoMudaram() que é chamado sempre que as medições climáticas mudam, e deve enviar as novas medições para diversas interfaces de dispositivos diferentes, cada um com uma função (previsão do tempo, condições climáticas atuais, etc...). Parece trabalhoso?

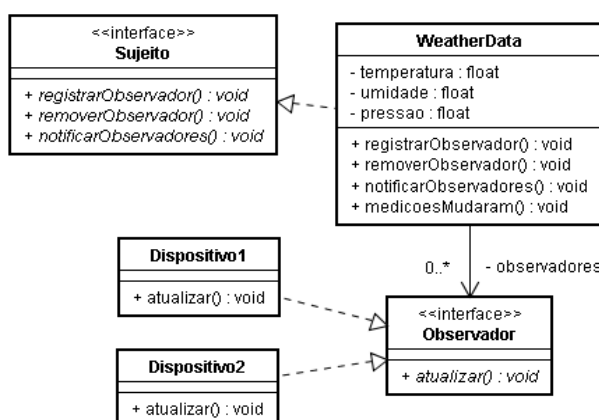
• O padrão Observer serve para situações assim. Um objeto, que está sendo observado, possui informações que podem mudar, e outros objetos, observadores dele, são notificados assim que a mudança ocorrer.

55

55



Padrão Observer

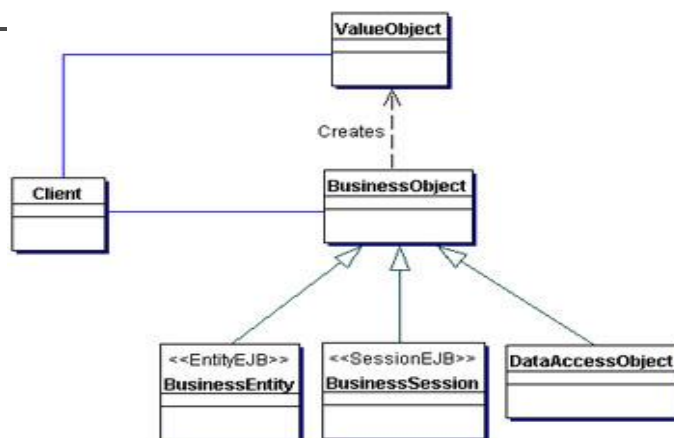


O sujeito(no caso, o WeatherData) se encarrega de notificar os observadores(no caso, os dispositivos) sobre quaisquer mudanças. O nosso problema está resolvido, e o uso de interfaces torna o código mais fácil de se fazer manutenção

56

56

Padrão TO



Um Transfer Object é um objeto que encapsula dados da fonte de dados em um determinado instante, ou seja, é uma cópia direta dos dados, enviada à aplicação no lugar dos originais

57

57

Padrão TO

```

public class ProjetoTO implements java.io.Serializable {
    public String idProjeto;
    public String nomeProjeto;
    public String idGerente;
    public String idCliente;
    public Date dataInicio;
    public Date dataFim;
    public boolean comecou;
    public boolean completado;
    public boolean aceito;
    public Date dataAceitacao;
    public String descricaoProjeto;
    public String statusProjeto;

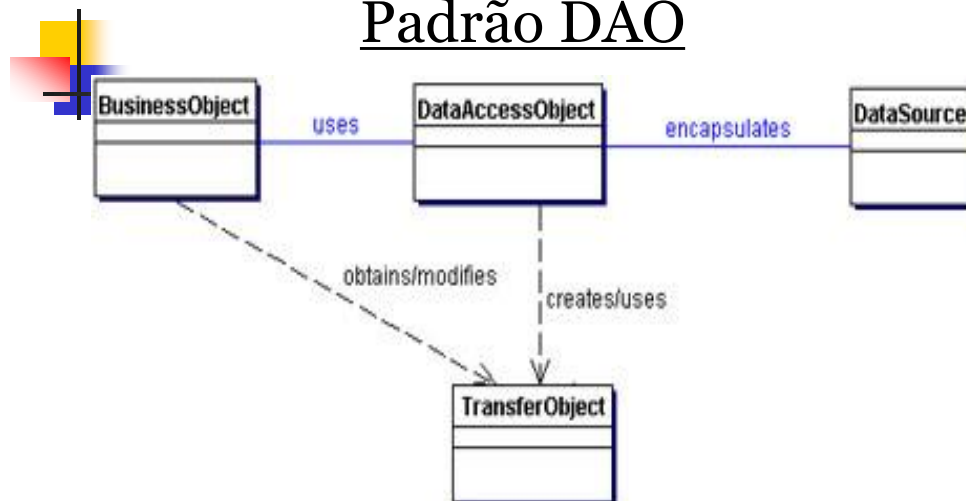
    // Construtores do TO aqui...
}
  
```

TOs podem implementar seus métodos de consulta na base de dados ou outras funcionalidades, mas o principal é servir como cópia dos dados, para que eles sejam enviados à aplicação

58

58

Padrão DAO



Um Data Access Object lida com a obtenção dos dados da base de dados. Ou seja, DAOs servem para obter, encapsular essas informações e retornar Transfer Objects que as contenham

59

59

Padrão DAO


```

import java.sql.*;
public class ClienteDAO {
    public ClienteDAO() {
        // inicializacao
    }
    public int insereCliente(...) {
        // Implementar inserção do cliente
    }
    public boolean removeCliente(...) {
        // Implementar remoção do cliente
    }
    public Cliente consultaCliente(...) {
        // Implementar consulta do cliente
    }
    public boolean atualizaCliente(...) {
        // Implementar update do cliente
    }
    public RowSet selectClientesRS(...) {
        // Implementar consulta com parâmetros supridos e retorna o RowSet
    }
    public Collection selectClientesTO(...) {
        // Implementar consulta com parâmetros supridos e retorna uma coleção de TOs
    }
}
  
```

DAOs são extremamente úteis quando lidamos com aplicações de banco de dados

60

60



FIM

61