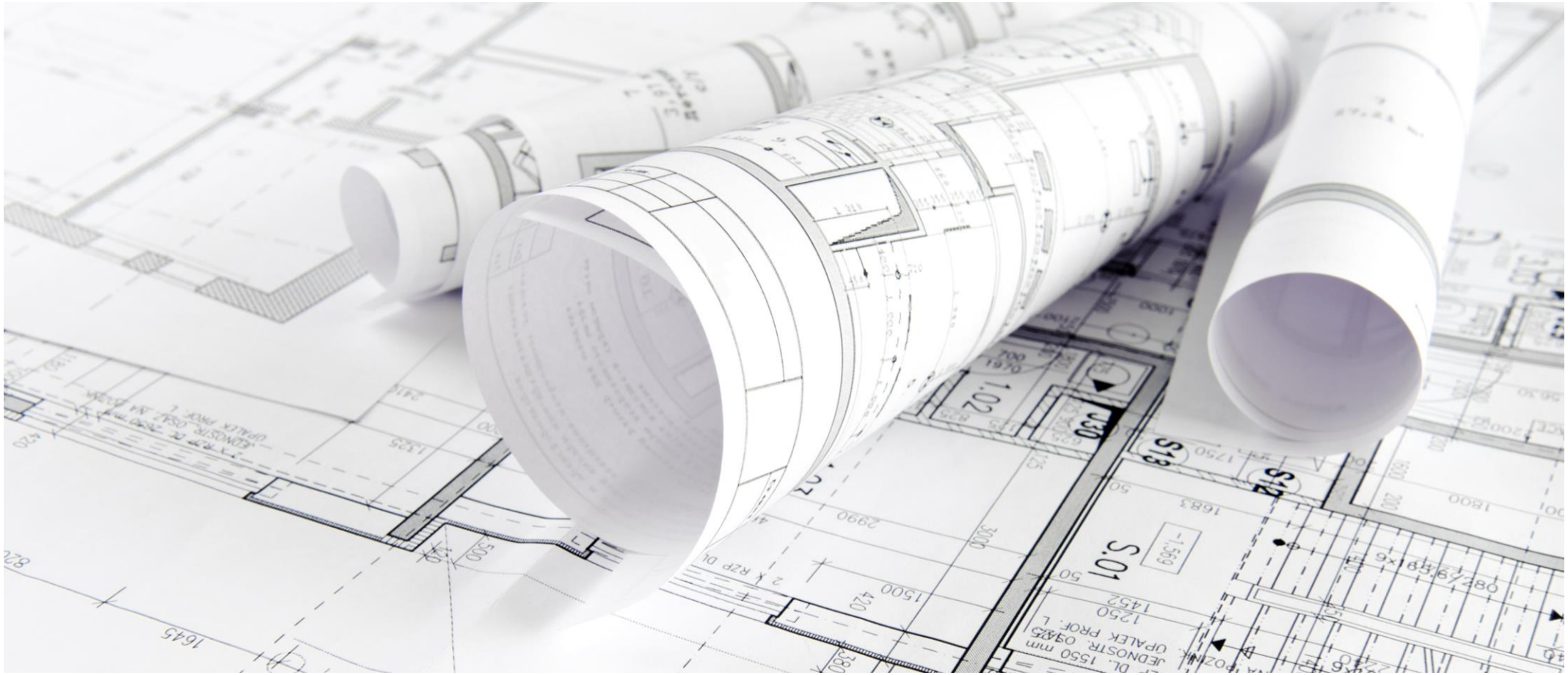
DOMINE OS 23 PADRÕES DE SOFTWARE

EXERCÍCIOS PRÁTICOS, EXEMPLOS REAIS



SOBRE MIM

Sou o **José Cruz**!

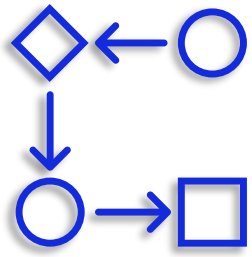
Comecei como programador, team leader, arquiteto, gestor de projeto e atualmente arquiteto empresarial.

Gosto de IT!

Desde código, a craftsmanship, qualidade, gestão, arquitetura, DevOps, Inteligência Artificial, gestão de dados, fiz e faço de tudo um pouco.

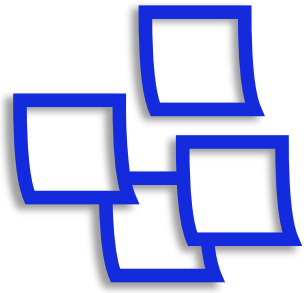
Podem-me encontrar facilmente no LinkedIn e quem sabe em alguns canais no Youtube.

REQUISITOS



- Conhecimentos de Java
- O que é o **maven**, e como adicionar dependências
- Conhecimentos de **JDBC**, **SQL**, como executar queries e criar tabelas.
- Saber o que são chamadas **Http** com um **GET**.
- Saber usar bibliotecas como o **HttpClient** (Java 11/apache) é um plus
- O que é **JSON** e **XML** poderá ajudar em alguns exercícios
- Java 17 (**records**, e um pouco de text Strings **"""…"""**)

MATERIAL



- IDE utilizado (**IntelliJ** – a versão gratuita é suficiente)
- Todos os exercícios têm uma resolução anexada (zip).
- Documento(s) PDF com tutoriais simples sobre:
  - **jdbc**, **h2**, **sqlite**, **mailpit**, **jsoup**, bibliotecas de excel e pdf (**poi**, **openpdf**), **jackson** para **json** e **xml** e **httpclient** (apache)
- Há Quiz em alguns dos módulos!
- Cábula (pdf) com um resumo dos 23 padrões.

AGENDA

Creacionais	Estruturais	Comportamento
• Factory Method • Abstract Factory • Singleton • Builder • Prototype	• Adapter • Bridge • Proxy • Facade • Composite • Decorator • Flyweight	• Chain of Responsibility • Command • Interpreter • Iterator • Mediator • Memento • Observer • State • Strategy • Template Method • Visitor

Também designados Gang of Four

PADRÕES

São soluções reutilizáveis para problemas comuns que surgem durante o desenvolvimento.

São como modelos que podem ser aplicados em diversas situações para resolver problemas semelhantes.

```
// Contador com ciclo
for(int contador = 0; contador < 10; contador++){
    // usa o contador
}

// contador com while
int contador = 0;
while(contador < 10){
    // usa o contador
    contador++;
}
```

```
// Acumulador, faz uso de um contador
int soma = 0;
for(int contador = 1; contador <= 10; contador++){
    soma += contador;
}

// Acumulador com while
int contador = 0;
int soma = 0;
while(contador <= 10){
    soma += contador;
    contador++;
}
```

PADRÕES CREACIONAIS

INTRODUÇÃO

PADRÕES CREACIONAIS

FACTORY METHOD

ABSTRACT METHOD

SINGLETON

BUILDER

PROTOTYPE

RESUMO

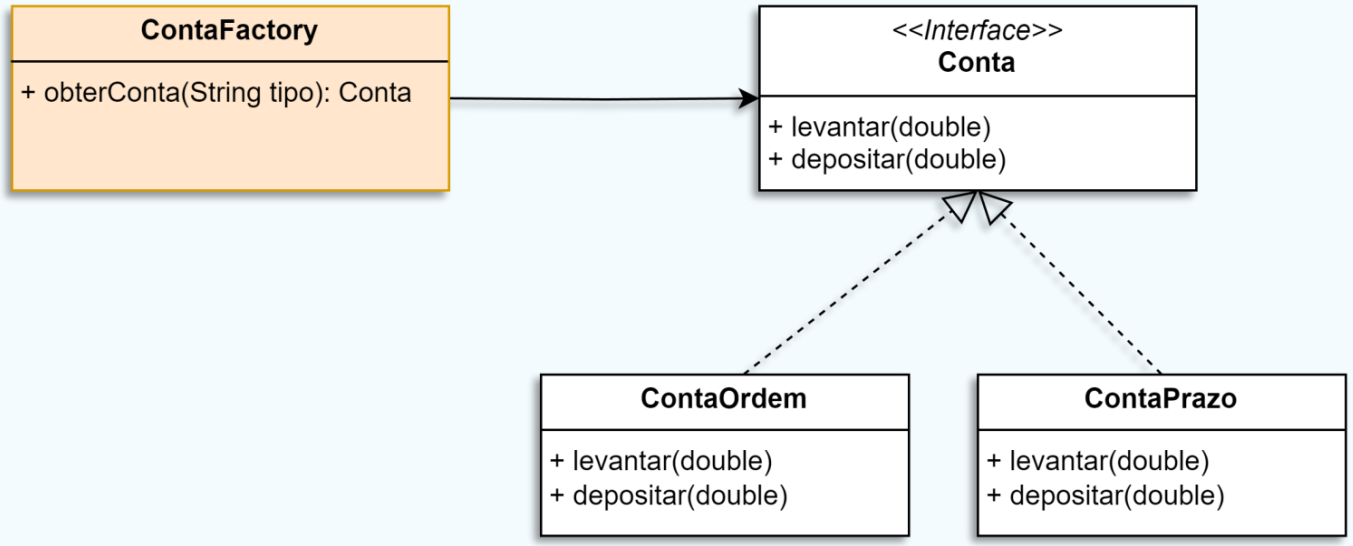
PADRÕES CRIACIONAIS

São padrões de design que lidam com mecanismos de **criação de objetos**.

- **Encapsulamento**: Reduz a complexidade e promovem a reutilização
- **Flexibilidade**: Facilitam a adição de novos tipos de objetos
- **Desaclopamento**: Reduzem a dependência entre classes e permitem isolar mudanças
- **Manutenção e testabilidade**: A criação dos objetos está centralizada, facilitando a manutenção e o desenvolvimento dos testes
- **Consistência**: Os objectos são criados da mesma forma em todo o sistema, promovendo, por exemplo, standarização entre a equipa de desenvolvimento
- **Eficiência**: Padrões como o Singleton podem ajudar a otimizar recursos e a reduzir potênciais bugs.

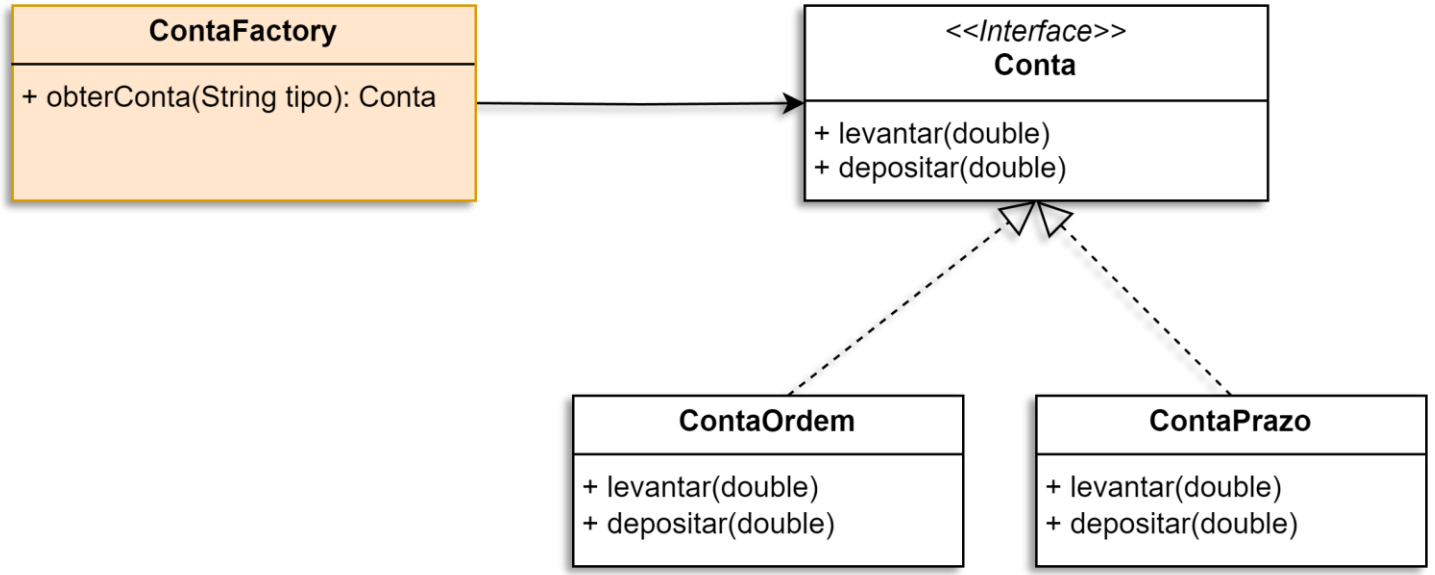
FACTORY METHOD

Um padrão que fornece uma interface para criar objetos, a partir de uma superclasse/interface, e permite que subclasses alterem o tipo dos objetos a criar.

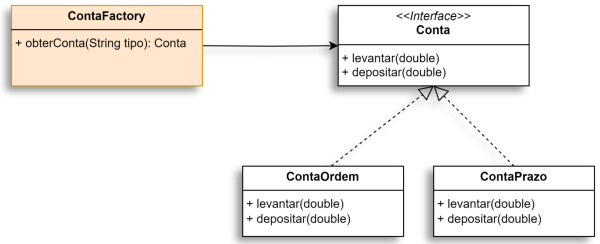


FACTORY METHOD

```java
public static void main(String[] args) {
    Conta contaOrdem = ContasFactory.obterConta("ordem", new Titular("N001", "Joaquim"));
    System.out.println(contaOrdem);
    contaOrdem.depositar(50.0);
    System.out.println(contaOrdem);

    Conta contaPrazo = ContasFactory.obterConta("prazo", new Titular("P002", "Maria"));
    System.out.println(contaPrazo);
    contaPrazo.depositar(100.0);
    System.out.println(contaPrazo);
    contaPrazo.levantar(50.0);
    System.out.println(contaPrazo);
}
```



FACTORY METHOD



Conceitos

- **Método Factory**: um método na superclasse responsável por criar os objetos.
- **Produto**: o objeto criado pelo método factory.
- **Criador**: a classe que contém o método factory. Pode ser uma classe abstracta ou uma interface, ou uma classe que devolve um objeto default.

Benefícios

- **Encapsulação**: encapsula o processo de criação do objeto.
- **Flexibilidade**: permite criar objetos sem conhecer os detalhes da classe.
- **Decoupling**: o cliente fica decoupled das classes instanciadas.
- **Single Responsibility Principle**: cada item tem a sua responsabilidade bem definida.
- **Open/Closed Principle**: é possivel introduzir novos objetos sem alterar a estrutura do código.

Desvantagens/Desafios

- **Complexidade**: projetos simples podem ficar mais complexos-ish com a adição do padrão(ões).
- **Subclasses**: cada novo objeto requer uma nova subclasse, aumentado a complexidade de código, por vezes de forma desnecessária
- **Manutenção**: quando o número de classes não funcionais aumenta, o código torna-se mais complicado de gerir.
- **Leitura**: os padrões exigem sempre um conhecimento mais aprofundado por parte dos programadores.

Casos de uso

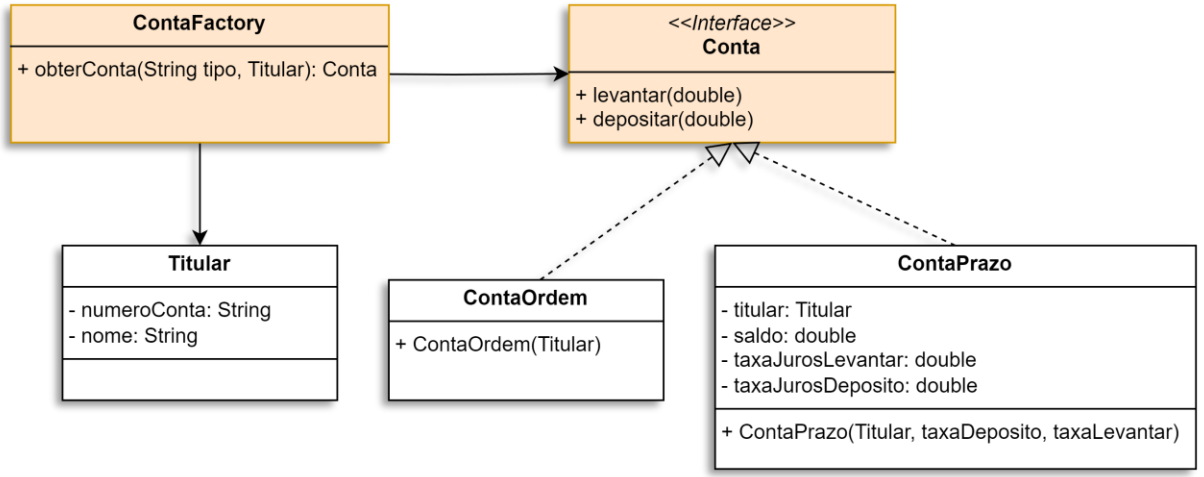
- **Database Connection Factories**
- **Document Factories**
- **Logger Factories**
- **Cache Factories**
- **Configuration Factories**
- **Image Factories**
- **Notification Factories**
- **Authentication Factories**
- **Payment Factories**
- **Data Source Factories**

FACTORY METHOD

Actividade: Contas Bancárias (fácil)

Criar uma aplicação que use o factory method para criar contas à ordem e contas a prazo

- A conta à ordem recebe e disponibiliza dinheiro
  - Não permite a conta ficar negativa
- A conta a prazo
  - Adiciona 10% por cada depósito
  - Penaliza com 2% cada levantamento.
- O constructor recebe dados sobre um **Titular**
- Simule as chamadas de deposito e levantamento de ambas as Contas



FACTORY METHOD

Actividade: Conexões a Base de Dados (médio)

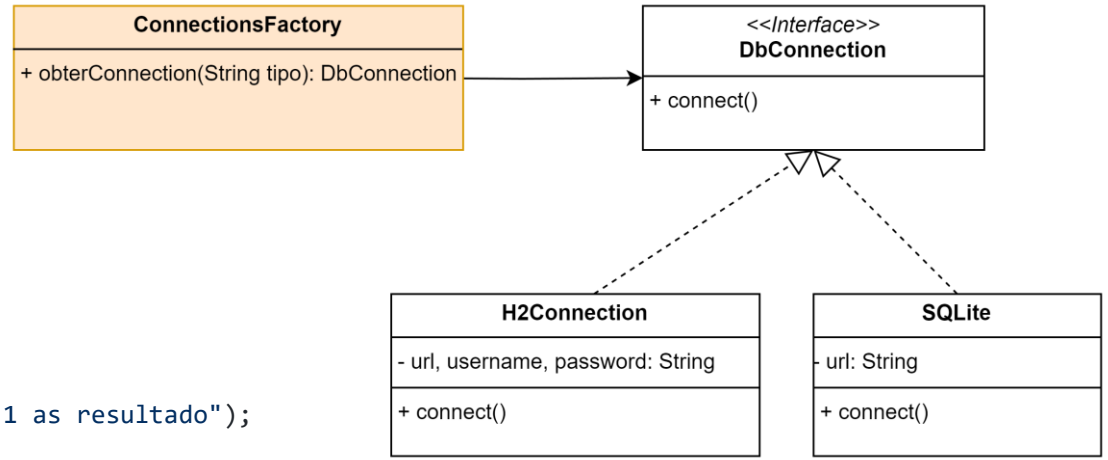
Criar uma aplicação que use o factory method para obter conexões a diferentes bases de dados usando uma **DbConnection** criada por nós.

- Usar o **Connection** do jdbc, dentro do **DbConnection**
- Usar as bases de dados: **H2** e **SQLite** (maven)

Não é necessário executar o "connect". Basta criar o url, no factory, de cada base de dados, conforme os requisitos.

Assumir que os urls são parâmetros da JVM

- *-Dh2.url=jdbc:h2:mem:testdb*
- *-Dsqlite.url=jdbc:sqlite::memory:*



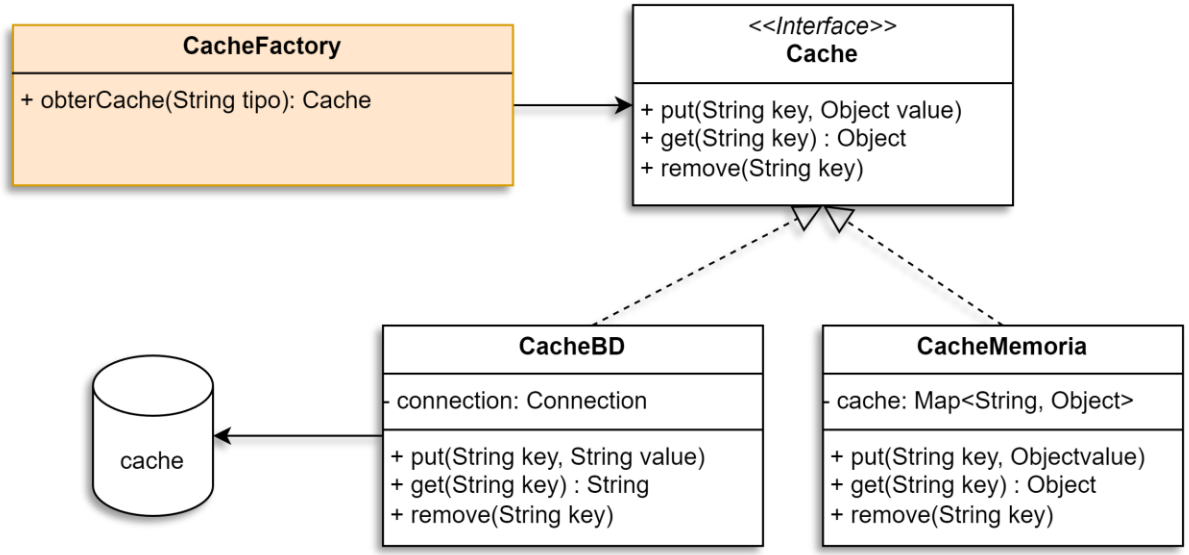

```
var res = h2Connection.connect()
        .createStatement()
        .executeQuery("select 1 + 1 as resultado");
```

FACTORY METHOD

Actividade: Multiplas Caches (médio)

Criar uma aplicação que use o factory method para gerir múltiplas caches

- Uma cache em memória, usando um ***HashMap<String, Object>***
- Uma cache numa base de dados (H2), usando uma tabela ***cache*** com as colunas (***chave***, ***valor***)
- criar 3 métodos no objeto cache:
  - *put(chave, valor)*
  - *get(chave) : valor*
  - *remove(chave)*



NO JDK

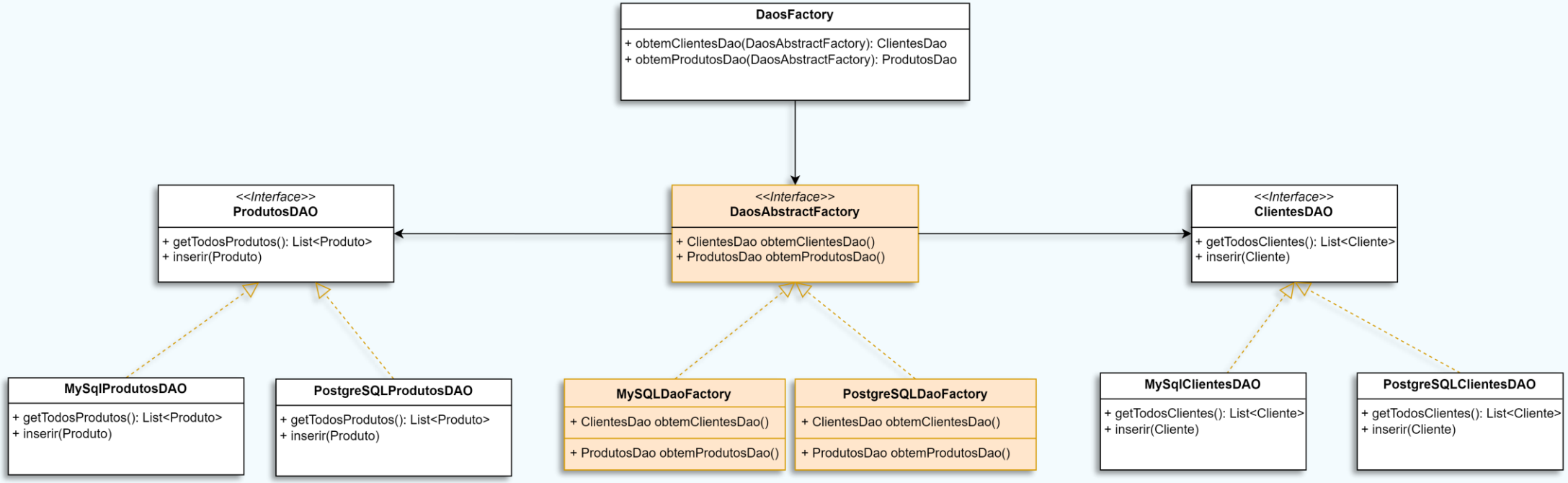
```
// java.util.Calendar
Calendar calendar = Calendar.getInstance();

// java.text.NumberFormat
NumberFormat numberFormat = NumberFormat.getInstance();
NumberFormat currencyFormat = NumberFormat.getCurrencyInstance();
NumberFormat percentFormat = NumberFormat.getPercentInstance();

// java.sql.DriverManager
Connection connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb", "username", "password");

// java.util.ResourceBundle
ResourceBundle resourceBundle = ResourceBundle.getBundle("bundle");
```

ANTI PATTERNS

- **Subclasses excessivas**: demasiados subtipos pode levar a um código difícil de se ler e manter.
- **Demasiada complexidade**: usar o padrão onde não é necessário é adicionar complexidade ao código desnecessariamente
- **Abstrações reveladoras**: Se o método factory revelar muito acerca do processo de criação, pode criar coupling do código com o padrão.
- **Criação inconsistente**: a ideia do padrão é criar consistência, se houver demasiadas formas diferentes de criar os objetos, o padrão deixa de fazer sentido (talvez o Builder!)
- **Sobre generalização**: se houver a necessidade de tornar o método muito genérico, vai dificultar o propósito do padrão, principalmente na adição de novos tipos.
- **Demasiado coupling**: se o método começa a ficar demasiadamente dependente das classes que pretende criar, vai reduzir a flexibilidade e reutilização do código.

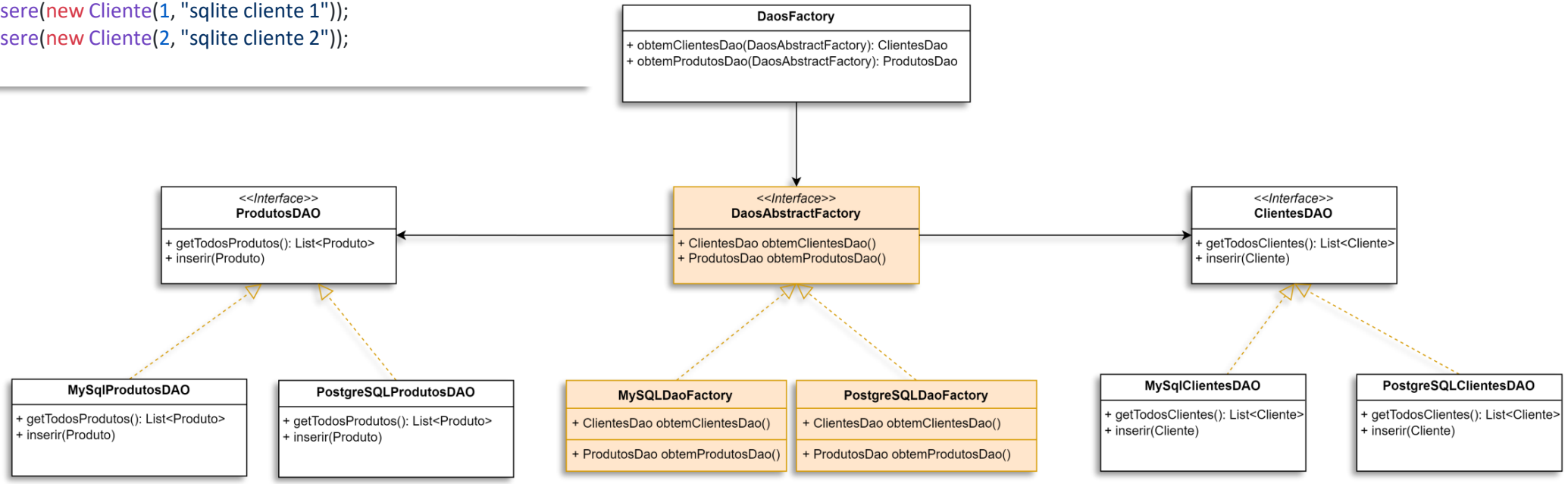
ABSTRACT FACTORY

Um padrão que fornece uma interface para criar uma família de objetos num determinado domínio.

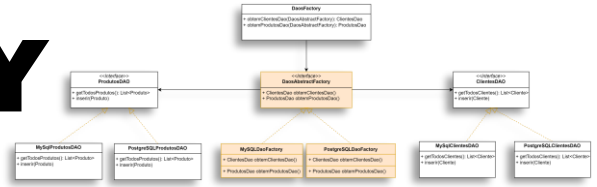


ABSTRACT FACTORY

```
public static void main(String[] args) {

    // h2 database
    H2DaosFactory h2DaosFactory = new H2DaosFactory(h2con);
    ProdutosDao h2produtosDao = DaosFactory.obtemProdutosDao(h2DaosFactory);
    h2produtosDao.insere(new Produto(1, "h2 produto 1"));
    h2produtosDao.insere(new Produto(2, "h2 produto 2"));

    // sqlite database
    SQLiteDaosFactory sqliteDaosFactory = new SQLiteDaosFactory(sqlitecon);

    ClientesDao sqliteclientesDao = DaosFactory.obtemClientesDao(sqliteDaosFactory);
    sqliteclientesDao.insere(new Cliente(1, "sqlite cliente 1"));
    sqliteclientesDao.insere(new Cliente(2, "sqlite cliente 2"));
}
```



ABSTRACT FACTORY



Componentes

- **Abstract Factory**: uma interface ou classe abstracta com os métodos dos tipos de objetos a criar
- **Concrete Factory**: A implementação da Factory abstracta
- **Abstract Product**: a interface ou classe abstracta para cada família de objetos
- **Concrete Product**: a implementação do Abstract Product.
- **Client**: o utilizador das factories e objetos

Benefícios

- **Isolamento de código**: O código de criação esta separado do código que os usa
- **Facilidade na troca de família de objetos**
- **Consistência**: Os objetos que fazem parte de uma família são "compativeis" entre si.
- **Escalabilidade**: Facilita a adição de novas famílias de objetos.
- **Manutenção**: As alterações nas implementações podem ser efetuadas sem impacto directo no código que as usa.

Desvantagens/Desafios

- **Complexidade**: O código pode começar a ficar complicado. Demasiada abstração por vezes esconde a funcionalidade.
- **Extensão**: Adicionar famílias pode ser complicado, do ponto de vista do coupling com as factories.
- **Sobrecarga do código**: Este tipo de abstrações adicionam sempre código não functional que pode ficar fora de controlo rapidamente.
- **Coupling**: Há um coupling entre as factories

Casos de uso

- **Sistemas de Interface Gráfica (GUI)**
- **Sistemas de Jogos**
- **Aplicações de Banco de Dados**
- **Aplicações de Documentos**
- **Sistemas de Veículos**
- **Sistemas de Comércio Eletrônico**
- **Sistemas de Fabricação**
- **Sistemas de Comunicação**
- **Sistemas de Segurança**
- **Sistemas de Relatórios**

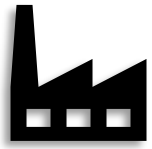
ABSTRACT FACTORY

Actividade: Acesso a base de dados (médio)

Criar uma aplicação que use o padrão abstract factory para usar classes DAO no acesso a duas bases de dados

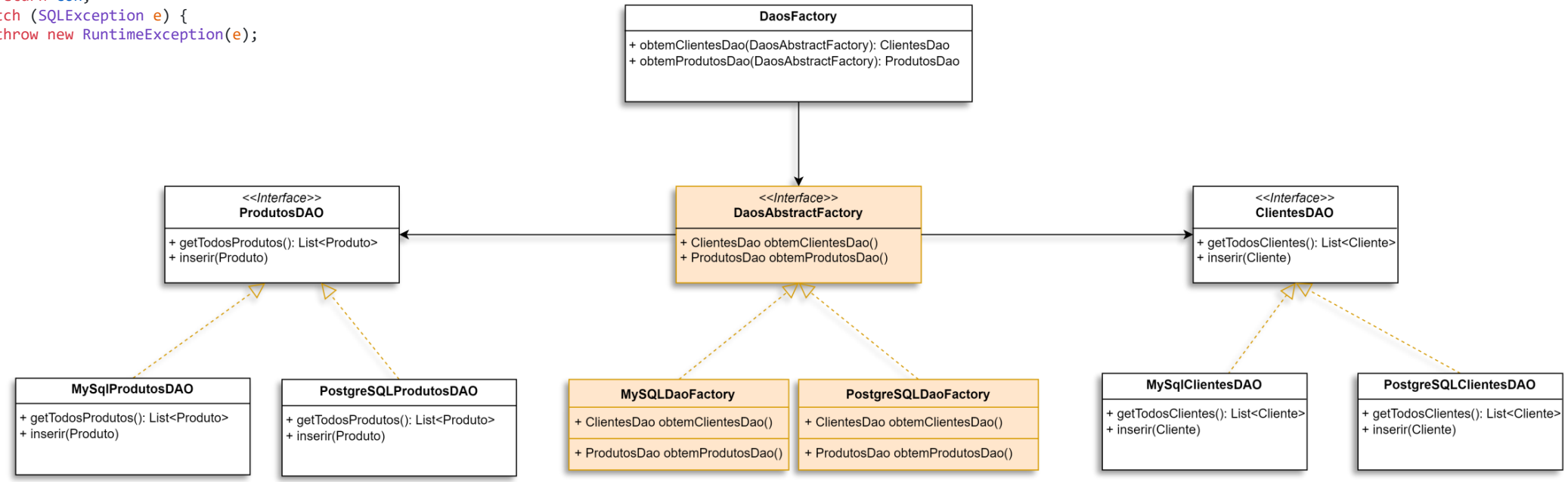
- Bases de dados: **H2** e **SQLite**
- Inicializar ambas as bases de dados com 2 tabelas: **produtos**, **clientes** com as colunas (*id integer, nome varchar(50)*)
- inicializar as tabelas com dados, nas primeiras linhas do **main**.
- 2 DAOs: **ProdutosDao** e **ClientesDao**
- Cada DAO tem pelo menos um método que faz um select à tabela associada e devolve uma lista de valores.
- O método main deverá usar as factories para criar os DAOs e aceder às tabelas e imprimir o resultado.

ABSTRACT FACTORY



Actividade: Acesso a base de dados

```java
public class H2Database {

    public static Connection inicializa() {
        try {
            Connection con = DriverManager.getConnection("jdbc:h2:mem:loja", "sa", "");

            Statement statement = con.createStatement();
            statement.execute("CREATE TABLE IF NOT EXISTS clientes (id INTEGER PRIMARY KEY, nome VARCHAR(255))");
            statement.execute("CREATE TABLE IF NOT EXISTS produtos (id INTEGER PRIMARY KEY, nome VARCHAR(255))");
            return con;
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }

}
```



ABSTRACT FACTORY

Actividade: Relatórios (avançado)

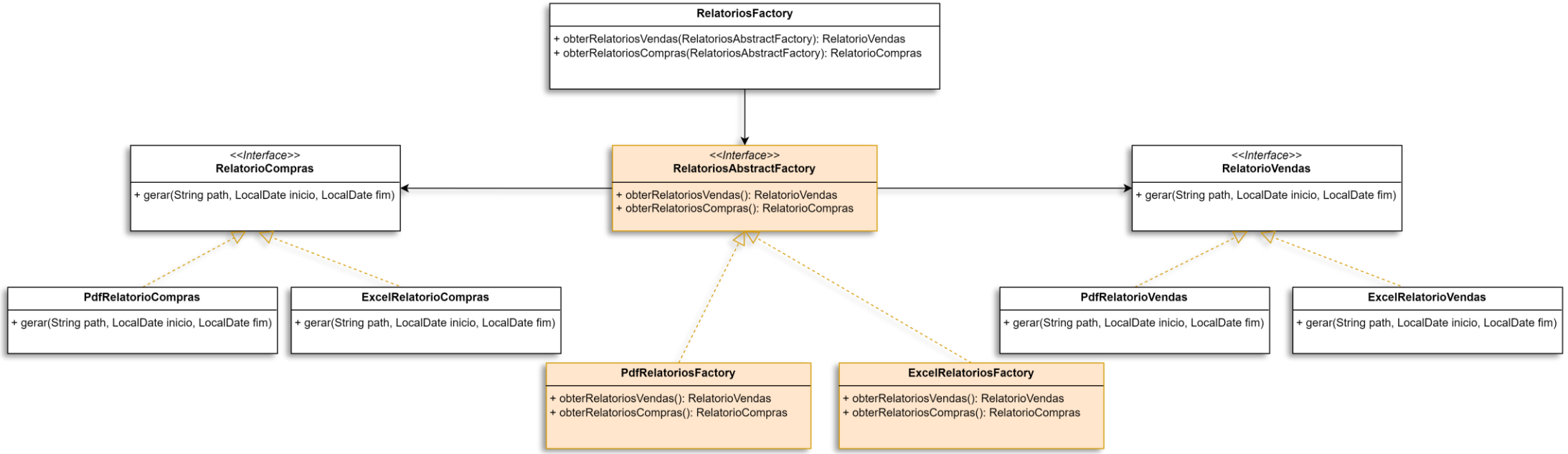
Criar uma aplicação que use o padrão abstract factory criar dois tipos de relatorios: *Vendas* e *Compras*, ambos no formato PDF e Excel

- Não é mandatório gerar os ficheiros, mas opcionalmente pode usar as bibliotecas: **openpdf** e **poi.ooxml** (pdf e excel respectivamente)
- Os relatórios terão, pelo menos, um método: *gerar(String path, LocalDate inicio, LocalDate fim)*
- As factories deverão ter pelos menos dois métodos:
  - obterRelatóriosVendas()
  - obterRelatóriosCompras()

NOTA: Os dados gerados não são relevantes. Qualquer texto como *"Olá sou um relatório PDF de Compras"* é suficiente!

ABSTRACT FACTORY

Actividade: Relatórios



ABSTRACT FACTORY

Actividade: Autenticação (médio)

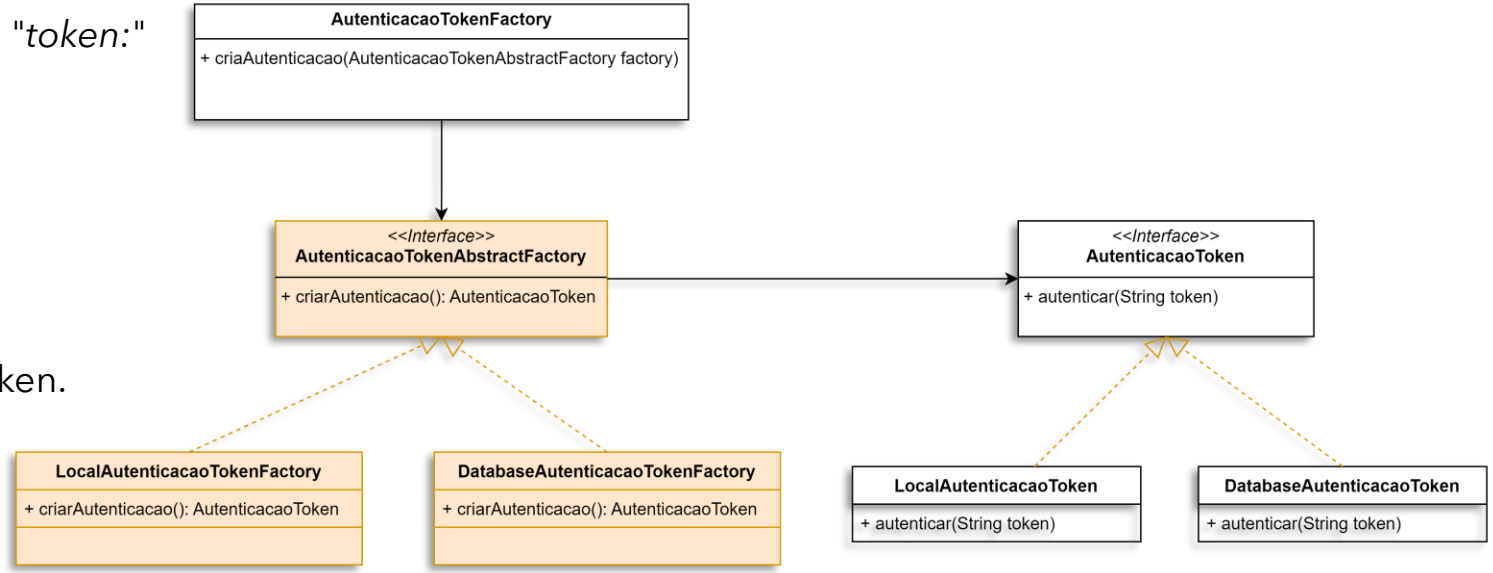
Criar uma aplicação que use o padrão abstract factory para gerir dois tipos de autenticação baseada em tokens, uma local e outra na base de dados.

Autenticação Local:

- verifica se um token começa como a string: "*token:*"

Autenticação Base de dados:

- usar a base de dados H2,
    - tabela: *tokens (token varchar(100))*
- verifica se a tabela tem um determinado token.



NO JDK

```
// javax.xml.validation.SchemaFactory
SchemaFactory factory = SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
Schema schema = factory.newSchema(new File("schema.xsd"));

// javax.xml.transform.TransformerFactory
TransformerFactory factory = TransformerFactory.newInstance();
Transformer transformer = factory.newTransformer();

// javax.xml.parsers.DocumentBuilderFactory
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
factory.setNamespaceAware(true);
DocumentBuilder builder = factory.newDocumentBuilder();
Document document = builder.parse(new File("input.xml"));
```

ANTI PATTERNS

Complexidade desnecessária: implementar o padrão abstract factory quando o factory method seria suficiente

Excesso de abstração: facilmente se abstrai demasiado tornando o código difícil de se ler.

Fábricas infladas: quando o número de tipos de objetos a criar cresce, pode começar a ficar difícil de gerir o Código

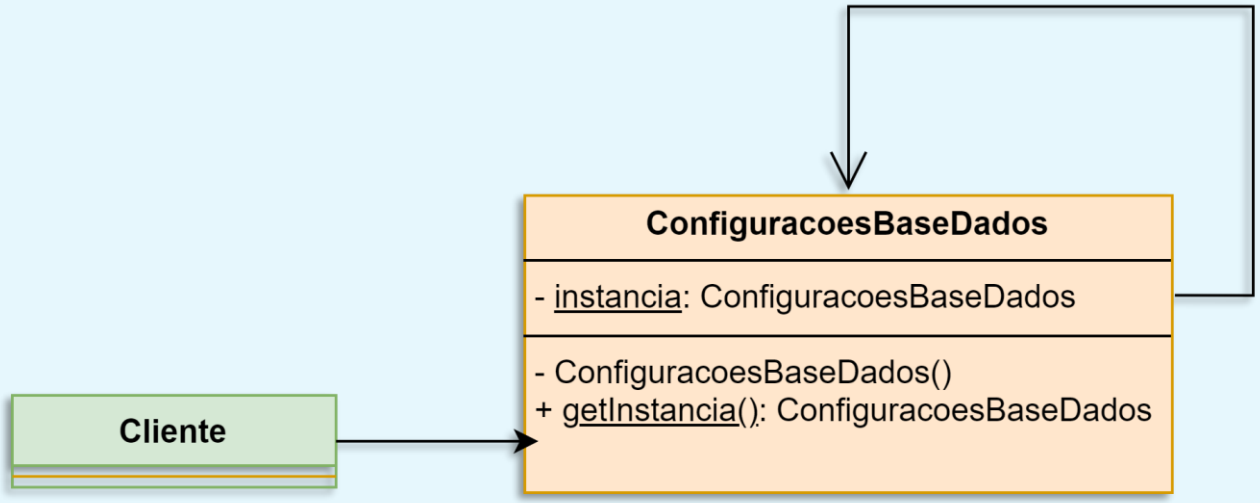
Rigidez: um uso excessivo do uso de padrão, pode impeder dinamismo no código, onde tudo passa a ser visto como tem de estar no padrão

Testes: fábricas de fábricas complica o código dos testes.

Dependência circular: facilmente se criam dependências circulares se o padrão não for bem planeado.

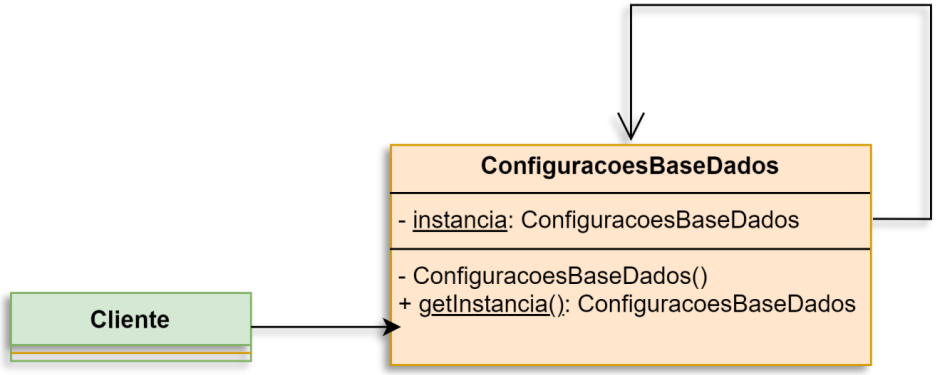
SINGLETON

Um padrão criacional que garante que uma classe tenha apenas uma única instância durante todo o ciclo de vida da aplicação.

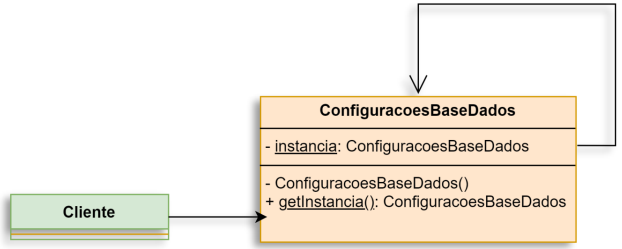


SINGLETON

```
public class ConfiguracoesBaseDados {
    private static ConfiguracoesBaseDados instance;

    private ConfiguracoesBaseDados() {
    }

    public static ConfiguracoesBaseDados getInstance() {
        if (instance == null) {
            instance = new ConfiguracoesBaseDados();
        }
        return instance;
    }
}
```



SINGLETON



Características

- **Instância única**: Só uma única instância da classe pode existir
- **Acesso global**: Fornece um ponto de acesso à instância
- **Controlo de instanciação**: A classe controla a sua própria instanciação

Benefícios

- **Controlo de recursos**: Útil para controlar o acesso a recursos que têm de ser únicos, como configurações, acesso a ficheiros, e semelhantes.
- **Consistência**: Garante que toda a aplicação tenha um local consistente para as configurações ou acessos a recursos.

Desafios

- **Testabilidade**: Classes que envolvem métodos estáticos e constructores privados são sempre um desafio para os testes
- **Concorrência**: Ambientes multithread têm sempre os seus desafios nestas situações.
- **Coupling**: Um ponto de acesso global, cria um coupling muito forte com todos os components da aplicação, criando pontos de falha com grande impacto.

Cenários

- **Gestão de configurações**
- **Sistemas de Log**: usam recursos externos com configurações transversais, que requerem varios "singleton" na aplicação.
- **Conexões** a bases de dados ou outros recursos externos "únicos" na aplicação.

SINGLETON

Formas de criar um Singleton em Java

- Lazy Initialization
- Eager Initialization
- Static initializer
- Enum
- Double-Checked Locking
- Bill Pugh Singleton

SINGLETON

Formas de criar um Singleton em Java

- **Lazy Initialization**
- Eager Initialization
- Static initializer
- Enum
- Double-Checked Locking
- Bill Pugh Singleton

```java
public class LazySingleton {
    private static LazySingleton instance;

    private LazySingleton() {
        // Construtor privado
    }

    public static LazySingleton getInstance() {
        if (instance == null) {
            instance = new LazySingleton();
        }
        return instance;
    }
}
```

SINGLETON

Formas de criar um Singleton em Java

- Lazy Initialization
- ➢ **Eager Initialization**
- Static initializer
- Enum
- Double-Checked Locking
- Bill Pugh Singleton

```java
public class EagerSingleton {
    private static final EagerSingleton instance =
                    new EagerSingleton();

    private EagerSingleton() {
        // Construtor privado
    }

    public static EagerSingleton getInstance() {
        return instance;
    }
}
```

SINGLETON

Formas de criar um Singleton em Java

- Lazy Initialization
- Eager Initialization
- ➢ **Static initializer**
- Enum
- Double-Checked Locking
- Bill Pugh Singleton

```
public class StaticBlockSingleton {
    private static StaticBlockSingleton instance;

    static {
        try {
            instance = new StaticBlockSingleton();
        } catch (Exception e) {
            throw new
RuntimeException("Erro ao criar a instância singleton");
        }
    }

    private StaticBlockSingleton() {
        // Construtor privado
    }

    public static StaticBlockSingleton getInstance() {
        return instance;
    }
}
```

SINGLETON

Formas de criar um Singleton em Java

- Lazy Initialization
- Eager Initialization
- Static initializer
- ➢ **Enum**
- Double-Checked Locking
- Bill Pugh Singleton

```java
public enum EnumSingleton {
    INSTANCE;

    private String info = "Informação inicial";

    public String getInfo() {
        return info;
    }

    public void setInfo(String info) {
        this.info = info;
    }
}
```

SINGLETON

Formas de criar um Singleton em Java

- Lazy Initialization
- Eager Initialization
- Static initializer
- Enum
- ➢ **Double-Checked Locking**
- Bill Pugh Singleton

```java
public class DoubleCheckedLockingSingleton {
    private static volatile DoubleCheckedLockingSingleton instance;

    private DoubleCheckedLockingSingleton() {
        // Construtor privado
    }

    public static DoubleCheckedLockingSingleton getInstance() {
        if (instance == null) {
            synchronized (DoubleCheckedLockingSingleton.class) {
                if (instance == null) {
                    instance = new DoubleCheckedLockingSingleton();
                }
            }
        }
        return instance;
    }
}
```

SINGLETON

Formas de criar um Singleton em Java

- Lazy Initialization
- Eager Initialization
- Static initializer
- Enum
- Double-Checked Locking
- ➢ **Bill Pugh Singleton**

```
public class BillPughSingleton {
    private BillPughSingleton() {
    }

    private static class SingletonHelper {
        private static final BillPughSingleton INSTANCE =
           new BillPughSingleton();
    }

    public static BillPughSingleton getInstance() {
        return SingletonHelper.INSTANCE;
    }
}
```

SINGLETON - SERIALIZABLE

```java
public class Singleton implements Serializable {
    private static final long serialVersionUID = 1L;
    private static final Singleton INSTANCE = new Singleton();

    private Singleton() {
    }

    public static Singleton getInstance() {
        return INSTANCE;
    }

    // Assegura que a propriedade singleton é mantida durante a serialização
    protected Object readResolve() throws ObjectStreamException {
        return INSTANCE;
    }
}
```

SINGLETON - REFLECTION

```
public class ReflectionBreakSingleton {
    public static void main(String[] args) {
        Singleton instance1 = Singleton.getInstance();
        Singleton instance2 = null;

        try {
            Constructor[] constructors = Singleton.class.getDeclaredConstructors();
            for (Constructor constructor : constructors) {
                constructor.setAccessible(true);
                instance2 = (Singleton) constructor.newInstance();
                break;
            }
        } catch (Exception e) {
            e.printStackTrace();
        }

        System.out.println("Instance 1 hash:" + instance1.hashCode());
        System.out.println("Instance 2 hash:" + instance2.hashCode());
    }
}
```

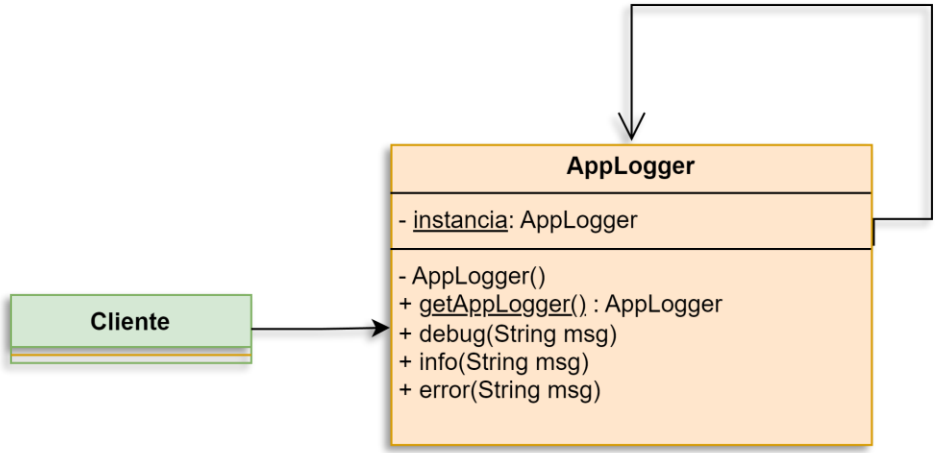
Apenas o ENUM garante proteção no singleton

SINGLETON

Actividade: Logger (fácil)

Criar uma aplicação que use o padrão singleton para implementar um Logger simples

- Nome da classe: **AppLogger**
- deverá suportar os métodos:
  - debug(String msg) – output: <DEBUG: msg>
  - info(String msg) – output: <INFO: msg>
  - error(String msg) – output: <ERROR: msg>

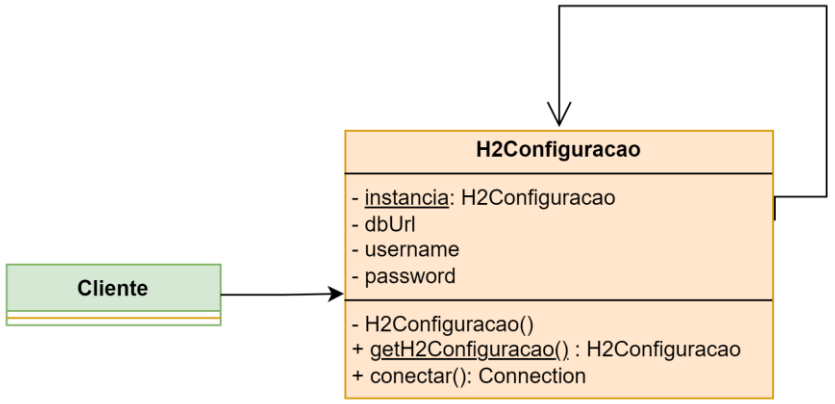


SINGLETON

Actividade: Configuração de bases de dados (médio)

Criar uma aplicação que use o padrão singleton para gerir o acesso a uma base de dados (ex: h2)

- Nome: **H2Configuracao**
- deverá conter as propriedades: **dbUrl**, **username**, **password**
- as propriedades poderão ser obtidas de propriedades de sistema (*-Ddb.url=, -Ddb.username=, -Ddb.password=*)
- deverá disponibilizar uma **Connection** quando pedida

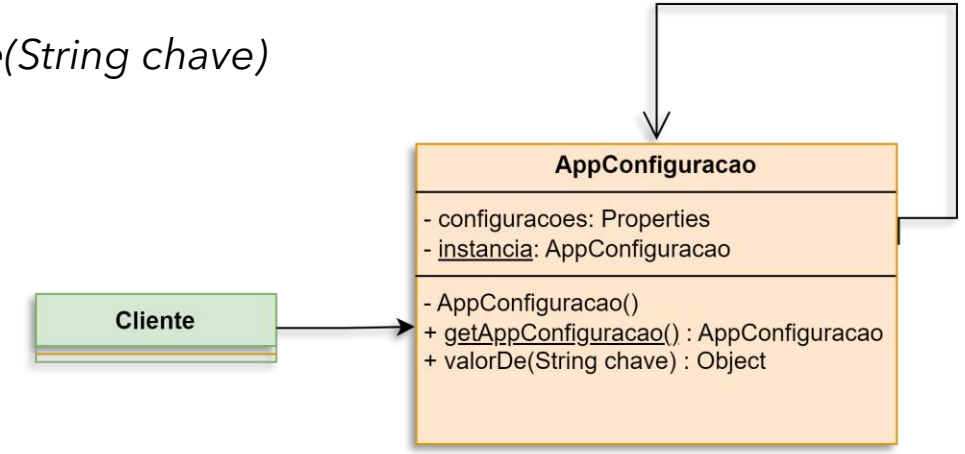


SINGLETON

Actividade: Configurações de Aplicações (médio)

Criar uma aplicação que use o padrão singleton para gerir um ficheiro de configurações no formato **configuracao.properties**

- A classe singleton poder-se-á chamar **AppConfiguracao**
- Deverá ler o ficheiro com o nome **configuracao.properties** (estará no classpath - resources)
- Deverá disponibilizar um método chamado: *Object valorDe(String chave)*



NO JDK

```
Runtime runtime = Runtime.getRuntime();
System.out.println("Processadores: " + runtime.availableProcessors());

Properties properties = System.getProperties();
properties.setProperty("user.name", "Pedro");
System.out.println("username: " + properties.getProperty("user.name"));

ClassLoader systemClassLoader = ClassLoader.getSystemClassLoader();
System.out.println("System class loader: " + systemClassLoader);

Logger logger = Logger.getLogger(MySingleton.class.getName());
logger.info("uma mensagem de log");
```

ANTI PATTERNS

Dificuldade em testar: O uso de static e o seu conceito de “global” torna difícil de isolar dependências no singleton.

Estado global oculto: pode ser difícil de se entender porque um sistema não está a funcionar como pretendido, devido ao que pode estar a ser escondido num singleton.

Concorrência e sincronização: statics são sempre inimigos de concorrência e sincronização.

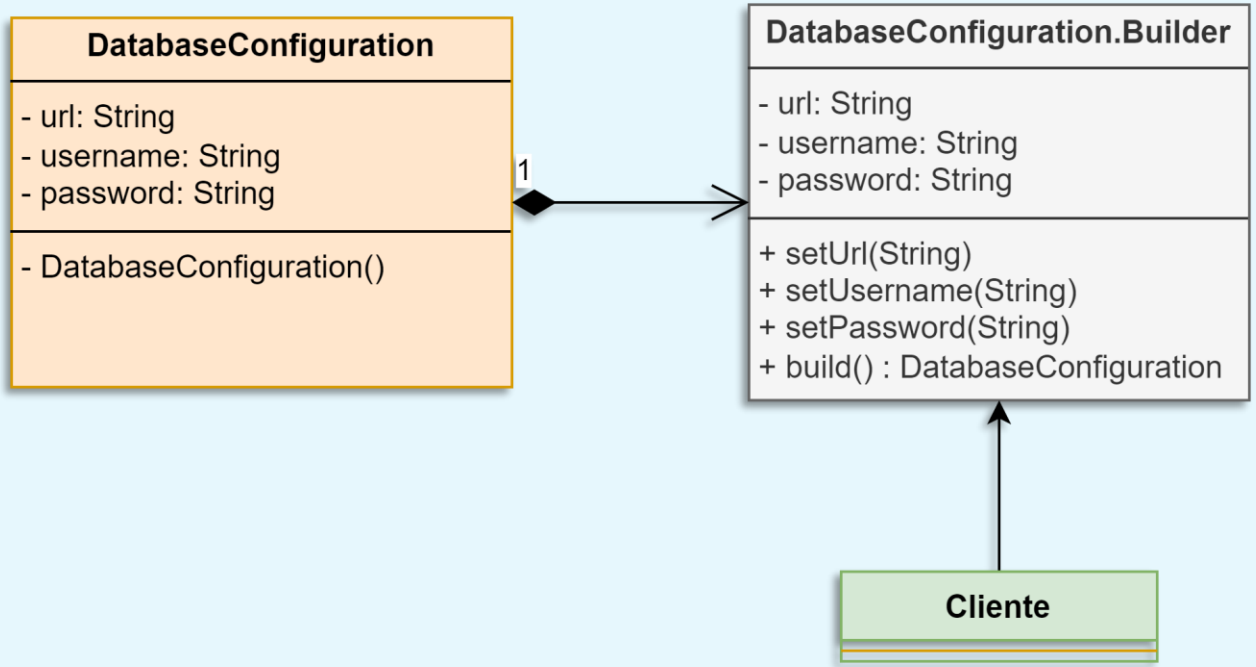
Coupling forte: objetos únicos têm tendência a conter informação útil para muitos outros objetos. Pequenas alterações no singleton tem forte impacto em outras partes do sistema.

Herança: não permitem uma herança harmoniosa.

Falsa sensação de segurança: O singleton por vezes parece ser “óbvio”, mas traz riscos. Por vezes há outras soluções melhores e mais flexíveis.

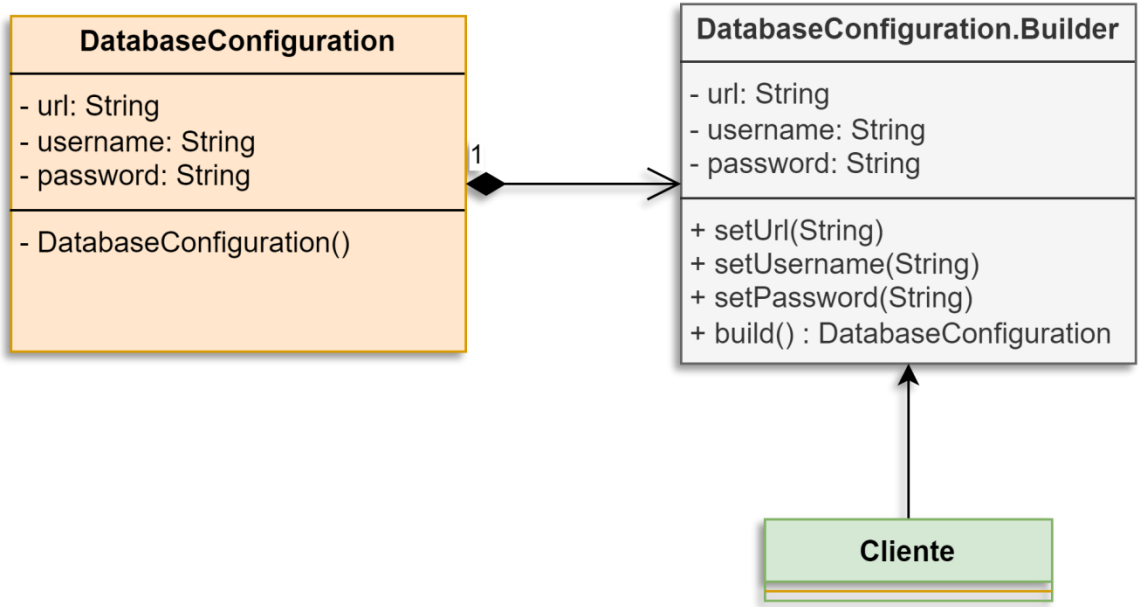
BUILDER

Um padrão criacional que permite a construção de objetos complexos passo a passo.

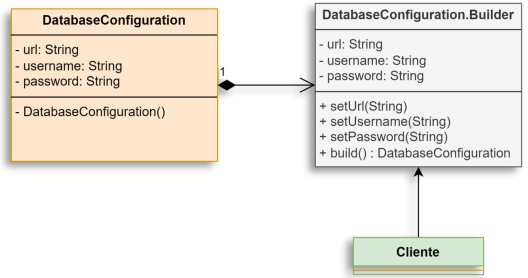


BUILDER

```java
public class DatabaseConfiguration {
    private String url;
    private String username;
    private String password;

    private DatabaseConfiguration() {
    }

    public static class Builder {
        private String url;
        private String username;
        private String password;

        public Builder setUrl(String url) {
            this.url = url;
            return this;
        }

        public DatabaseConfiguration build() {
            DatabaseConfiguration config = new DatabaseConfiguration();
            config.url = this.url;
            config.username = this.username;
            config.password = this.password;
            return config;
        }
    }
}

public class MainApp {
  public static void main(String[] args) {
        DatabaseConfiguration config = new DatabaseConfiguration.Builder()
            .setUrl("jdbc:mysql://localhost:3306/mydb")
            .setUsername("root")
            .setPassword("password")
            .build();
      }
}
```



BUILDER



Características

- **Builder**:Interface ou classe abstracta que define os métodos para construir as diferentes partes do objeto
- **ConcreteBuilder**: Constroi e monta as partes do produto. Fornece um método para recuperar o produto final.
- **Director (opcional)**: A classe que controla o processo de construção.
- **Produto**: O objeto que esta a ser construido.

Vantagens

- **Facilita a construção** de objetos complexos
- **Construtores**: Evita o uso de múltiplos constructores para múltiplas configurações.
- **Flexibilidade**: permite criar diversos objetos com configurações diferentes, usando o mesmo código.
- **Separação de preocupações**: separa a lógica de construção da lógica de representação.
- **Reutilização** de código: o mesmo builder para diversas formas do objeto.
- **Manutenção**: alterações no código não afetam o código do cliente.

Desafios

- **Complexidade**: objetos simples, com poucas configurações necessitam sempre de alguma quantidade de código base.
- **Sobrecarga de código**: padrões com varios components exigem sempre algum código de suporte.

Cenários

- **Construção de objetos complexos** com diversas formas
- **Construção a partir de configurações exteriores** como ficheiros (xml, json, etc)
- **Interfaces gráficas**
- **Construção de documentos**

BUILDER

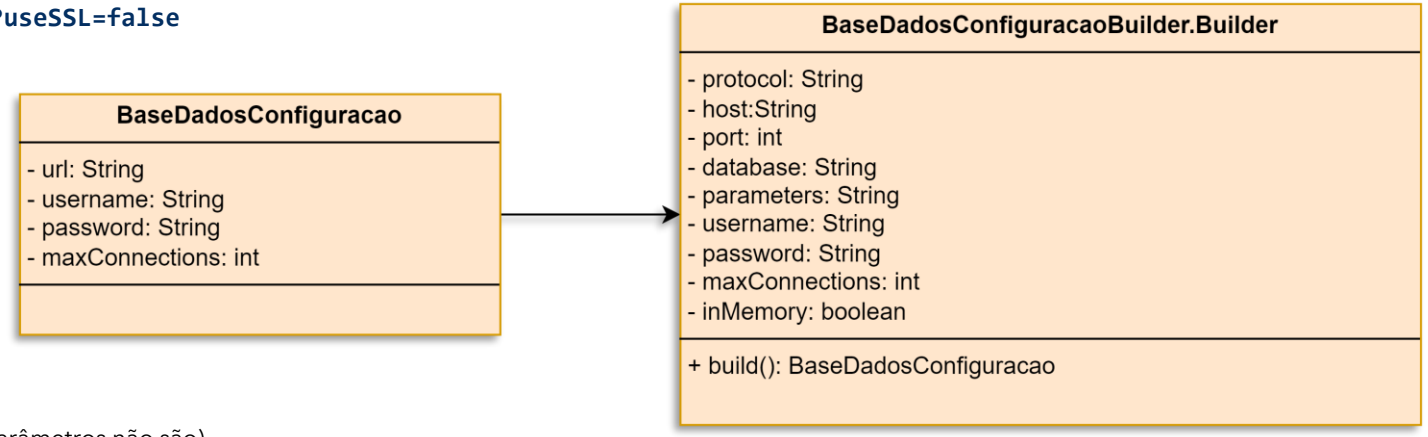
Actividade: Configuração base de dados (fácil)

Criar um programa usando o padrão Builder para acesso a base de dados:

- `Ex do url: jdbc:mysql://localhost:3306/mydb?useSSL=false`

Para a construção do URL, usar os atributos:

- **protocol**: (jdbc:mysql, jdbc:oracle, jdbc:postgresql)
- **host**: (//localhost)
- **port** (:3306)
- **Database Name** (/mydatabase)
- **Parameters** (ex: ?useSSL=false&serverTimezone=UTC)
- **NOTA**: jdbc:mysql://localhost:3306/mydb (obrigatório, os parâmetros não são)



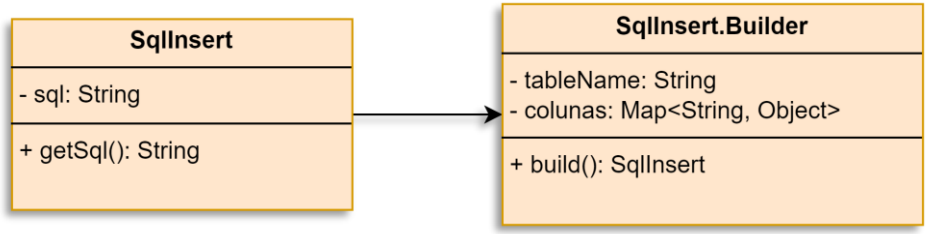
Campos a apresentar: ***url***, ***username***, ***password***, ***maxConnections***

BUILDER

Actividade: SQL Insert Builder (médio)

Criar um programa usando o padrão Builder para construir um **insert** para uma base de dados

- Classe de nome: **SqlInsert**
- atributos: *String nomeTabela, Map<String, Object> colunas*
- o build constroi o SQL



NOTAS:

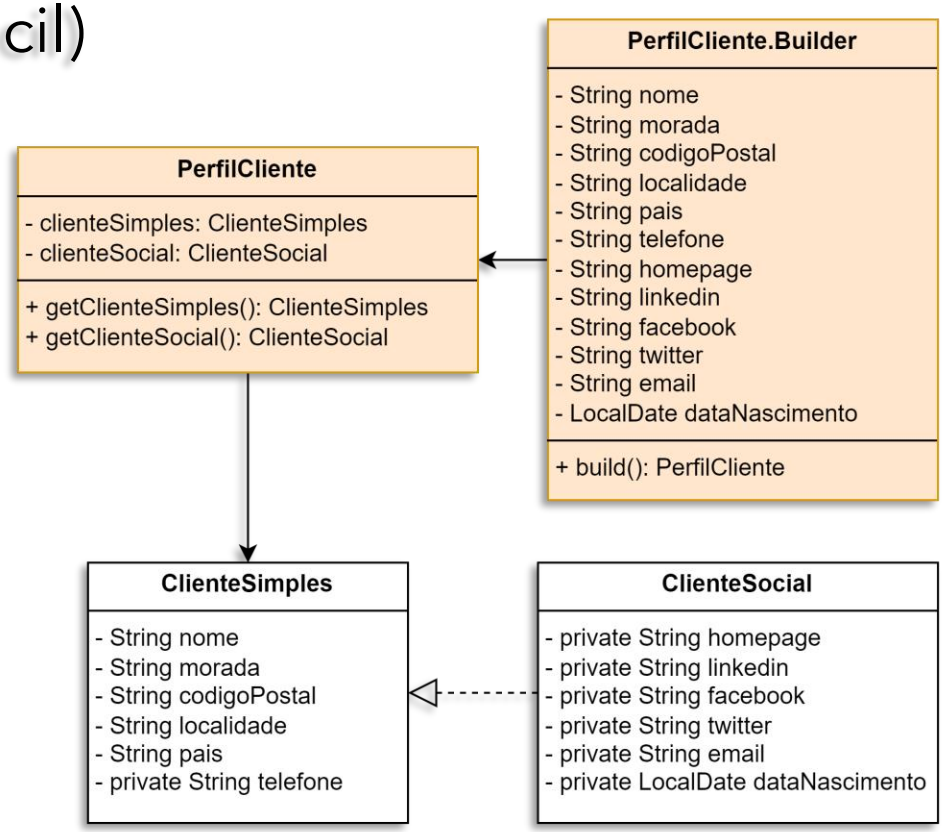
- Se possível considerar que há valores no insert que usam pelicas (só os inteiros não usam).
- Nas colunas, poderá ser necessário usar *Map<Coluna, Object>* ao invés do Map<String, Object>, devido ao tipo de dados, afim de se decidir o uso de pelicas ou não. Ex: *insert into (id, nome) values (2, 'um nome')*

BUILDER

Actividade: Perfil de um cliente (fácil)

Criar um programa usando o padrão Builder para construir 2 tipos de clientes

- Todos os atributos possiveis para os clientes:
  - nome, morada, codigoPostal, localidade, pais, telephone, homepage, linkedin, facebook, twitter, email, dataNascimento
- Cliente simples (**ClienteSimples**):
  - Atributos: nome, morada, codigoPostal, localidade, país, telephone
- Cliente social (**ClienteSocial**)
  - Atributos: "cliente simples" + homepage, linked, facebook, twitter, email, dataNascimento
- O builder constroi os dois objetos que podem ser obtidos com um simples **getXXX**
- As duas classes para os clientes podem ser separadas, ou uma herdar da outra **(+)**.



NO JDK

```
StringBuilder sb = new StringBuilder();
    sb.append("Olá, ");
    sb.append("mundo!");
    String resultado = sb.toString();

// Locale -> build
Locale locale = new Locale.Builder()
    .setLanguage("pt")
    .setRegion("BR")
    .build();

// HttpRequest -> build
HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("https://api.teste.me"))
    .header("Content-Type", "application/json")
    .GET()
    .build();
```

ANTI PATTERNS

Uso excessivo: uso em POJOs ou Value Objects, quando 1 ou 2 construtores seriam a solução.

Falta de valores default: Quando há atributos não escolhidos e não têm um valor default adequado.

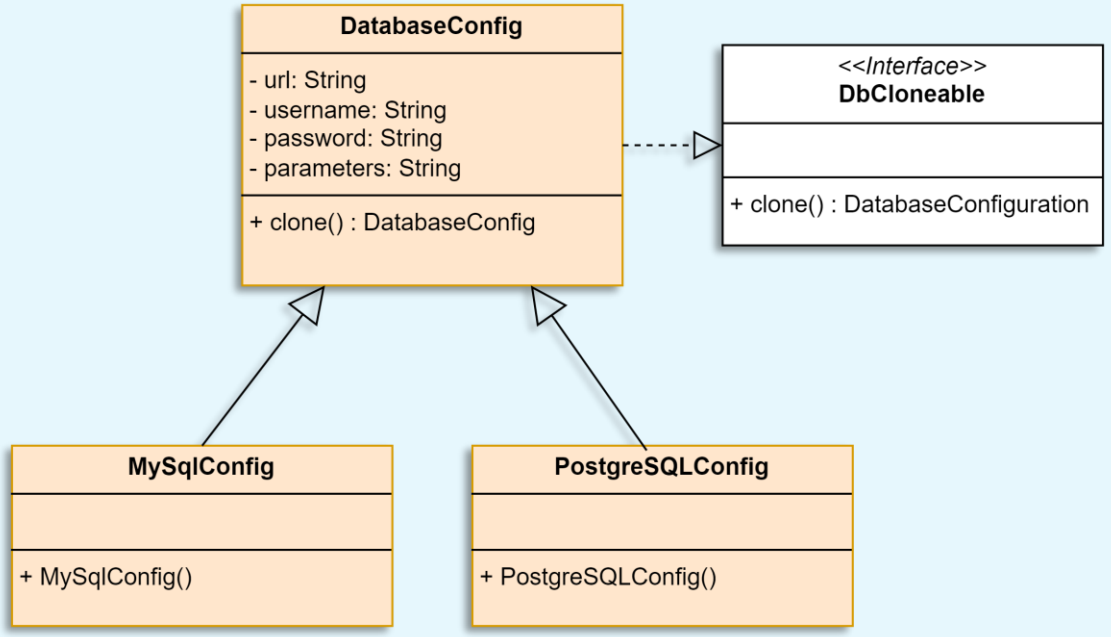
Validação durante a execução: efetuar validações nos campos durante o Build ao invés de assegurar em tempo de compilação.

Complexidade desnecessária: se um objeto é simples não se usa o Builder

Inconsistência: mau uso do Builder, permitindo que um objeto seja criado numa determinada forma de diversas maneiras.

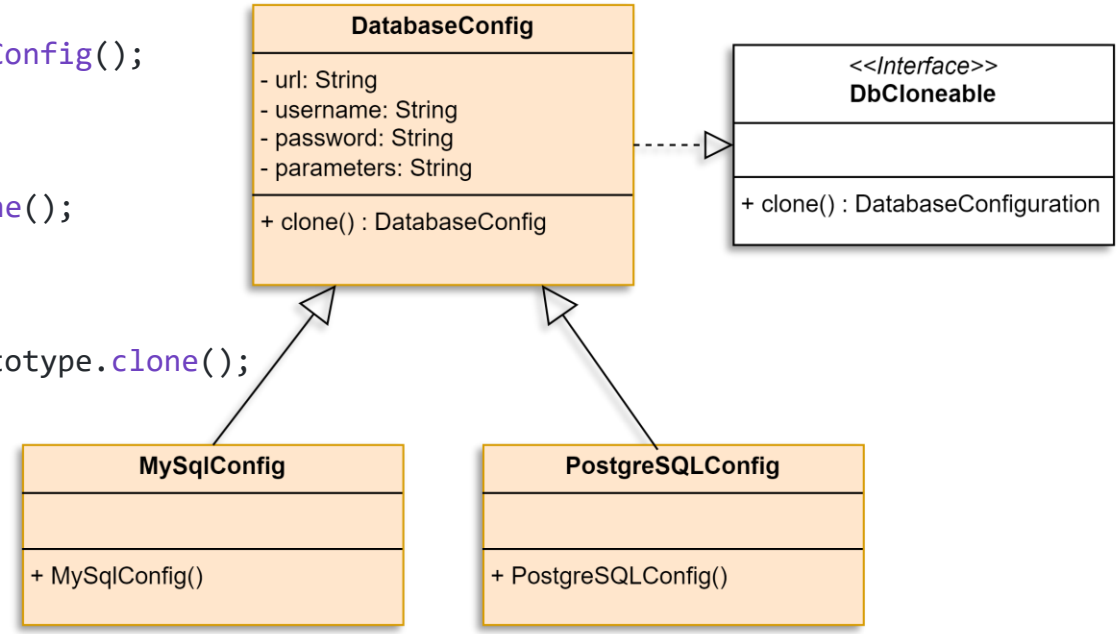
PROTOTYPE

Um padrão criacional que permite a criação de novos objetos a partir de um modelo original ou protótipo que é clonado.

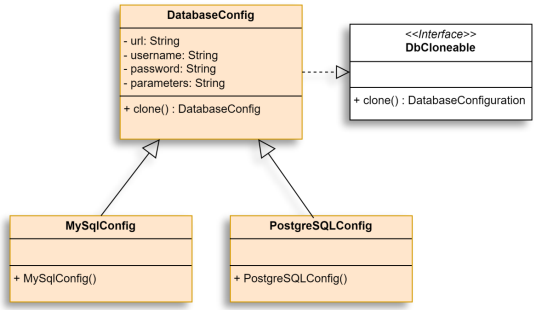


PROTOTYPE

```java
public class Main {
    public static void main(String[] args) {
        // Protótipos
        DatabaseConfig mySQLPrototype = new MySQLConfig();
        DatabaseConfig postgreSQLPrototype = new PostgreSQLConfig();

        // Clona
        for (int i = 0; i < 3; i++) {
            DatabaseConfig mySQLConfig = mySQLPrototype.clone();
            mySQLConfig.setUser("usuario" + i);
            mySQLConfig.setPassword("senha" + i);

            DatabaseConfig postgreSQLConfig = postgreSQLPrototype.clone();
            postgreSQLConfig.setUser("usuario" + i);
            postgreSQLConfig.setPassword("senha" + i);
        }
    }
}
```



PROTOTYPE



Características

- **Prototype**: A interface que declara o método para a clonagem
- **ConcretePrototype**: implementação da interface "Prototype"
- **Client**: Classe que utiliza o método de clonagem para criar novo objetos a partir do protótipo.

Vantagens

- **Redução de custos**: Evita o custo da criação de objetos a partir do zero, especialmente quando são complexos
- **Flexibilidade**: Permite a criação de objetos em tempo de execução, sem necessidade de usar subclasses.
- **Simplicidade**: simplifica a criação de objetos complexos, reutilizando um protótipo existente.

Desafios

- **Clonagem profunda**: quando um objeto contem sub-objetos, que por sua vezes têm outros objetos, a clonagem pode-se tornar complexa de implementar.

Cenários

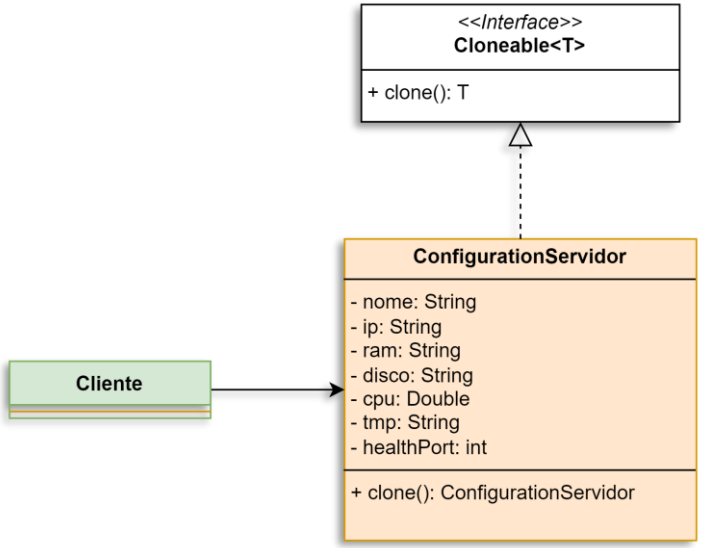
- **Jogos**: os objetos usados nos jogos baseiam-se em clonagem de objetos existentes com pequenas variações, que poderão acontecer em runtime.
- **Interfaces gráficas**: a operação duplicar existe em todos estes tipos de interfaces
- **Configurações**: diferentes configurações a partir de um modelo base

PROTOTYPE

Actividade: Configuração de servidores (fácil)

Criar um programa usando o padrão Prototype que lê configuracoes de uma aplicação para correr num servidor

- Configurações da aplicação
  - Nome, ip, ramSize, hdSize, cpus, tmpFolder, healthPort
- Crie uma configuração de servidor, duplique e mude o IP
- Implemente o toString para verificar as Mudanças



NOTA: Por vezes é necessário fazer start a outra instância com a mesma configuração, mas mudando o IP ou outra configuração.

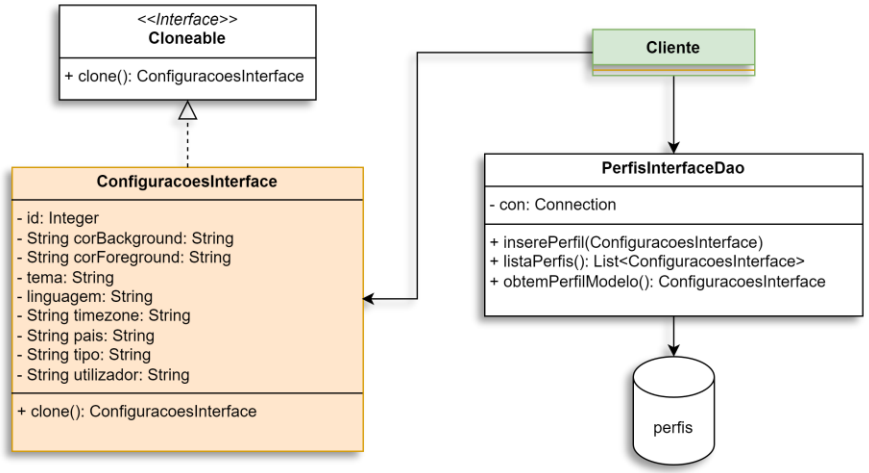
PROTOTYPE

Actividade: Configurações de interface preferênciais (médio)

Criar um programa usando o padrão Prototype que permite clonar configurações de um utilizador modelo para novos utilizadores

classe: **ConfiguracoesInterface**

- corBackground: String
- corForeground: String
- tema: String (dark, light)
- linguagem: String (ex: pt ou en, ou outro)
- timezone: String (ex: GMT+1)
- País: String
- tipo: String (modelo, regular) – um utilizador *modelo* não existe, serve como base de cópia para novos utilizadores
- utilizador: String



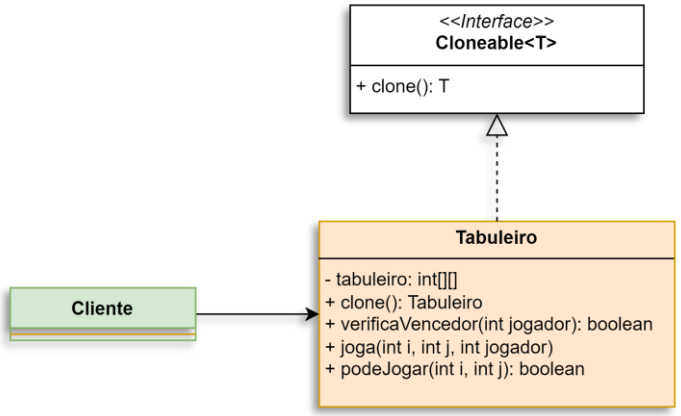
- Criar uma tabela com **perfis_interface**, com colunas a corresponder aos atributos, com uma chave primária *id:int* adicional.
- A tabela deverá ser inicializada com o utilizador modelo (base de dados H2 deverá ser suficiente para o código)
- Quando um novo utilizador é criado, deve ser feita uma cópia do "modelo" e permitir a alteração. Depois é inserido na tabela

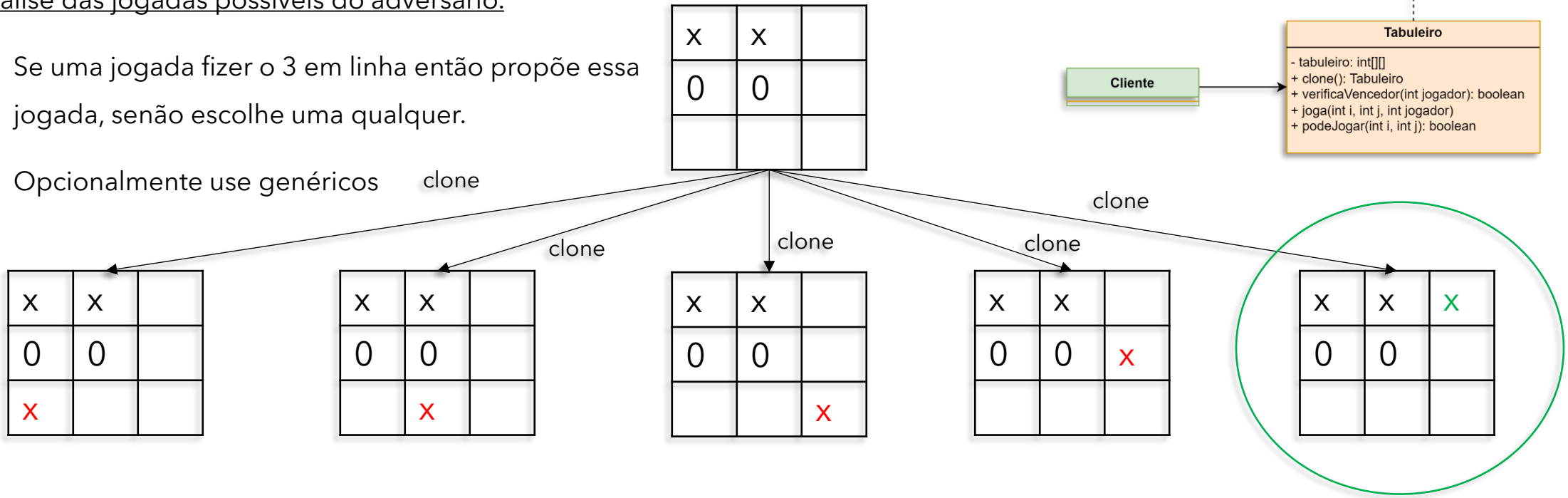
PROTOTYPE

Actividade: Jogo 3 em linha - próxima jogada (avançado)

Criar um programa usando o padrão Prototype que dado um **Tabuleiro** do jogo 3 em linha (3x3), faça uma análise das jogadas possiveis do adversário.

- Se uma jogada fizer o 3 em linha então propõe essa jogada, senão escolhe uma qualquer.
- Opcionalmente use genéricos





NO JDK

```
// Clone arraylist in java
ArrayList<String> list = new ArrayList<>();
list.add("item1");
list.add("item2");
ArrayList<String> listClone = (ArrayList<String>) list.clone();

// Clone date in java
java.util.Date date = new java.util.Date();
java.util.Date dateClone = (java.util.Date) date.clone();

// Clone object in java
Object obj = new Object();
Object objClone = obj.clone();

// Clone localdatet java
java.time.LocalDate localDate = java.time.LocalDate.now();
java.time.LocalDate localDateClone = localDate.clone();
```

ANTI PATTERNS

Clonagem excessiva: se houver muitos níveis de profundidade pode complicar o código e o consumo de recursos.

Falta de controlo de versões: o que fazer quando há necessidade de sincronizar os clones com o seu protótipo.

Dependência do estado global: se houver dependências com singletons, ou outro de valores globais, há grande possibilidade de bugs difíceis de capturer.

Clonagem de objetos mutáveis: se o protótipo for mutável, os clones poderão estar inconsistentes com o expectável.

Complexidade desnecessária: uso do prototype quando simples “new” com construtores serve o propósito.

Falta de limpeza de recursos: se a clonagem envolve recursos externos, como garantir o control do acesso e a respectiva gestão, como limpeza ou fecho de ligações, etc?

RESUMO

OS PADRÕES CRIACIONAIS, SÃO O QUE O NOME INDICA, PARA CRIAR OBJETOS DE DIVERSAS FORMAS.

CADA PADRÃO TEM AS SUAS FORÇAS E FRAQUEZAS.

PERCEBENDO A SEMÂNTICA DE CADA UM, É O SEGREDO PARA A ESCOLHA ACERTADA.

RUMOR: AS FRAMEWORKS MODERNAS USAM COM FREQUÊNCIA O PADRÃO BUILDER.

PADRÕES ESTRUTURAIS

INTRODUÇÃO

PADRÕES ESTRUTURAIS

ADAPTER

BRIDGE

PROXY

FACADE

COMPOSITE

DECORATOR

FLYWEIGHT

RESUMO

PADRÕES ESTRUTURAIS

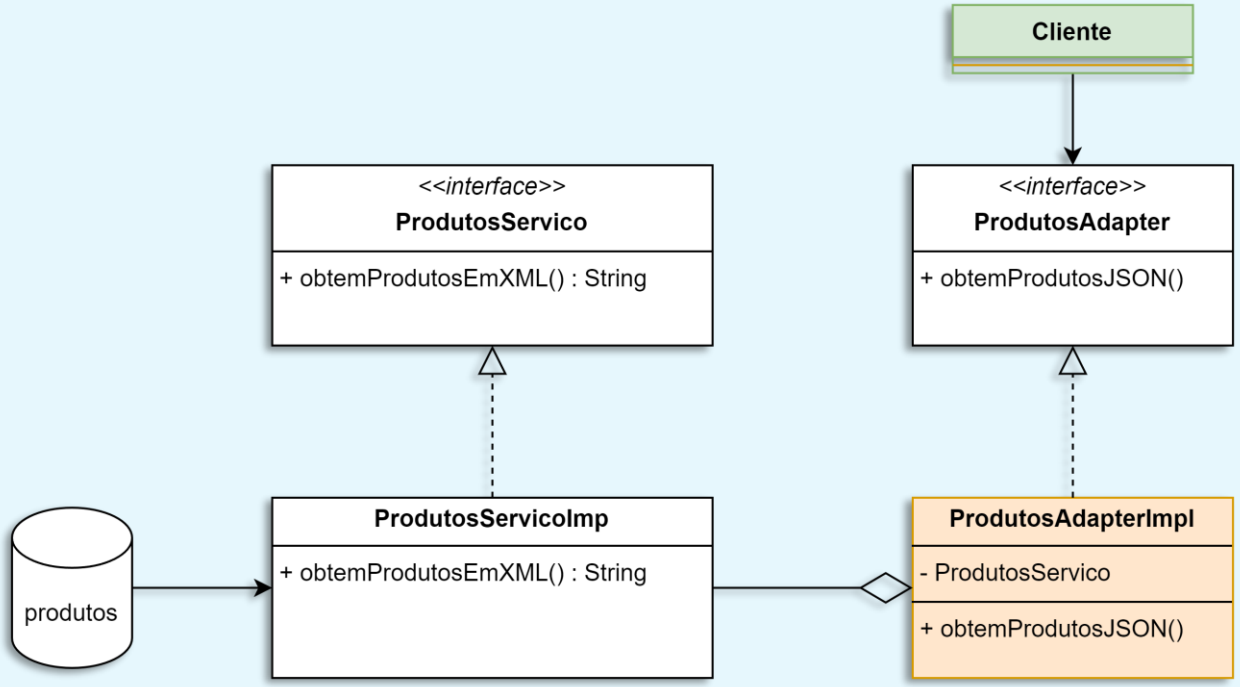
São padrões de software com soluções pré-definidas para problemas comuns de design em programação. Ajudam a determinar a forma como as classes e objetos são compostos, permitindo alcançar outros níveis de abstração.

- **Organização**: permitem cria uma melhor relacionamento entre classes.
- **Flexibilidade**: as estruturas ficam flexíveis, abertas para manutenção e evolução do sistema
- **Complexidade**: reduzem a complexidade, criando uma estrutura mais clara e compreensivel
- **Coupling**: reduz o coupling entre classes

ADAPTER

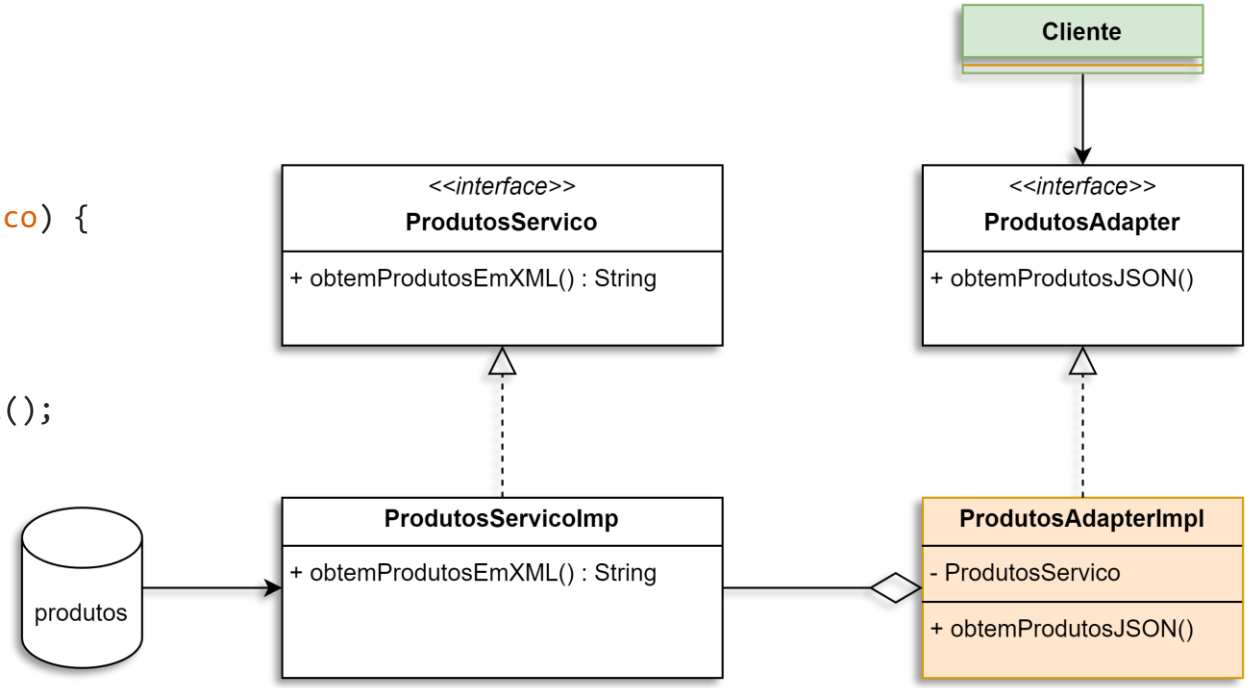
Um padrão que permite que objetos com interfaces diferentes colaborem entre si.

(ex: adaptadores eléctricos entre países diferentes)

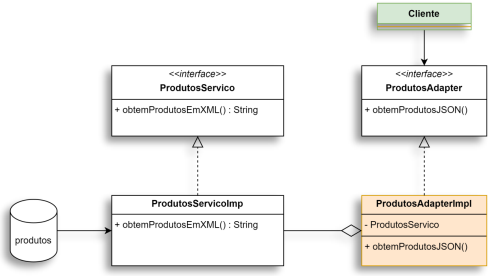


ADAPTER

```java
public class ProdutosAdapter {
    private ProdutosServico produtosServico;

    public ProductAdapter(ProdutosServico produtosServico) {
        this.produtosServico = produtosServico;
    }

    public String obtemProdutosJSON() {
        String xml = produtosServico.obtemProdutosEmXML();
        JSONObject jsonObject = XML.toJSONObject(xml);
        return jsonObject.toString();
    }
}
```



ADAPTER



Componentes

- **Target**: A interface que o cliente espera.
- **Client**: Colabora com objetos que seguem a interface Target.
- **Adaptee**: Uma classe existente que precisa de ser adaptada para a interface Target.
- **Adapter**: Converte a interface do Adaptee para a interface Target.

Benefícios

- **Reutilização de código**: permite reutilizar classes existentes mesmo que as suas interfaces não sejam compativeis.
- **Flexibilidade**: facilita a integração de novas bibliotecas ou classes sem modificar o código existente.
- **Manutenção simplificada**: se a interface do Adaptee mudar, apenas o Adapter precisa ser alterado

Desvantagens

- **Complexidade**: se o número de adapter for elevado, o código pode começar a ficar difícil de seguir
- **Sobrecarga de código**:o uso de padrões traz sempre código intermédio que ajuda no desenho para o código fica mais complex de se entender por todos os membros

Casos de uso

- **integração** com bibliotecas e classes
- adaptação com **APIs** existentes
- importante em código **legacy**

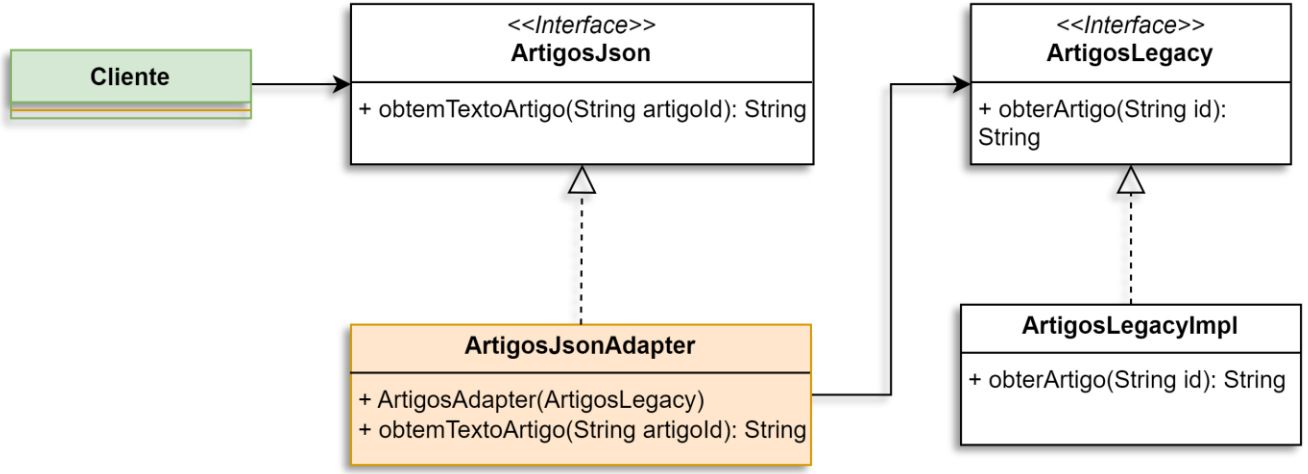
ADAPTER

Actividade: Artigos JSON (fácil)

Criar um programa usando o padrão Adapter, que dado um sistema existente que devolve artigos em texto, é necessário adaptá-lo ao nosso sistema moderno em JSON.

O texto é transformado na estrutura:

```
{
    "texto": "texto capturado"
}
```



ADAPTER

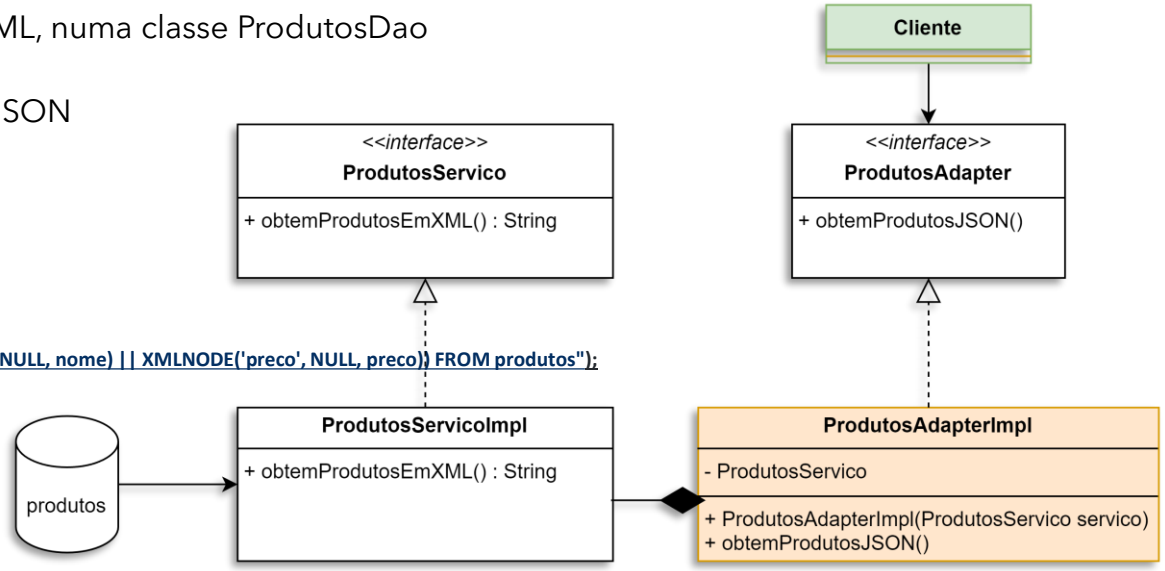
Actividade: Produtos de XML para JSON (médio)

```xml
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.13.0</version>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.dataformat</groupId>
    <artifactId>jackson-dataformat-xml</artifactId>
    <version>2.13.0</version>
</dependency>
```

Criar um programa usando o padrão Adapter, seguindo a lógica apresentada no diagrama.

- Utilizar a base de dados **H2**
- Criar uma tabela de **produtos** com os atributos: *id, nome, preco*
- Fazer uma query à tabela dos produtos que devolve o resultado em XML, numa classe ProdutosDao
- Criar um Produtosadapter para transformar o resultado em XML para JSON
- Apresentar os dados em JSON no método *main*

```java
public String obtemProdutosAsXml() {
    try {
        String xml = "<produtos>";
        Statement stat = con.createStatement();
        ResultSet rs = stat.executeQuery("SELECT XMLNODE('produto', NULL, XMLNODE('id', NULL, id) || XMLNODE('nome', NULL, nome) || XMLNODE('preco', NULL, preco)) FROM produtos");
        while(rs.next()) {
            xml += rs.getString(1);
        }
        xml += "</produtos>";
        return xml;
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```



ADAPTER

Actividade: Ligar a REST Api (médio)

Criar um programa usando o padrão Adapter, que permite que um cliente chame uma api que obtem um facto sobre gatos

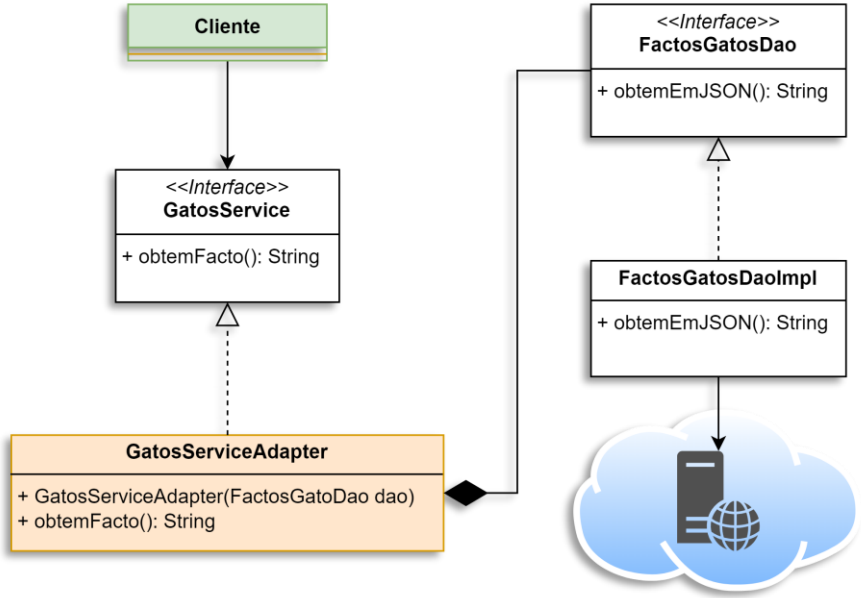
- Factos acerca de gatos pode ser obtida aqui: *https://catfact.ninja/fact*
- O cliente usa o adapter para transformar o JSON para String
    - *String gatosService() // o adapter*

NOTAS:

- Para aceder à REST API pode-se implementar um DAO: **FactosGatosDao**
- Uma biblioteca para fazer as chamadas HTTP: **HttpClient** do Java 11

```
<dependency>
  <groupId>org.json</groupId>
  <artifactId>json</artifactId>
  <version>20240303</version>
</dependency>
```

```
{
    "fact": "Cats control the outer ear using 32 muscles; humans use 6",
    "length": 57
}
```



NO JDK

```
// Converte um array para List
String[] array = {"a", "b", "c"};
List<String> list = Arrays.asList(array);

// Stream para Reader (parece um Decorator mas não é)
InputStream inputStream = new FileInputStream("file.txt");
Reader reader = new InputStreamReader(inputStream);

// Stream para Writer
OutputStream outputStream = new FileOutputStream("file.txt");
Writer writer = new OutputStreamWriter(outputStream);

// Lista para Enumeration
List<String> list = Arrays.asList("a", "b", "c");
Enumeration<String> enumeration = Collections.enumeration(list);

// As streams são adapters
int[] numbers = {1, 2, 3, 4, 5};
IntStream intStream = Arrays.stream(numbers);
```

ANTI PATTERNS

Adapters para tudo: usar em todas as incompatibilidades entre interfaces

Adapter inflado: demasiada funcionalidade

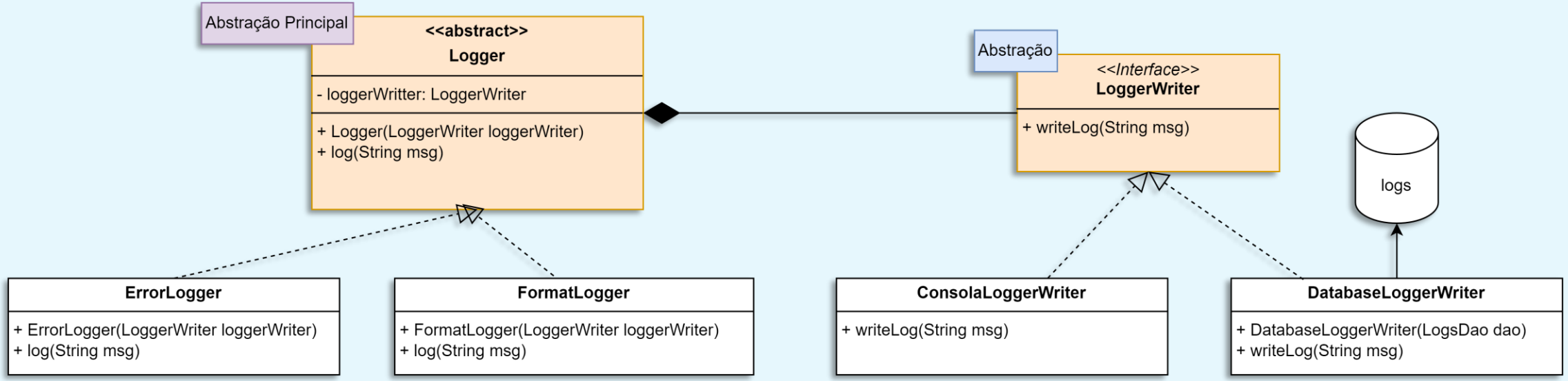
Falta de coesão: uso para interfaces com pouca relação entre elas

Dependências circulares: quando o adapter sabe demasiado do adaptee e vice versa.

Solução temporária: usar para resolver um problema de integração e nunca mais fazer a evolução ou refactor do sistema

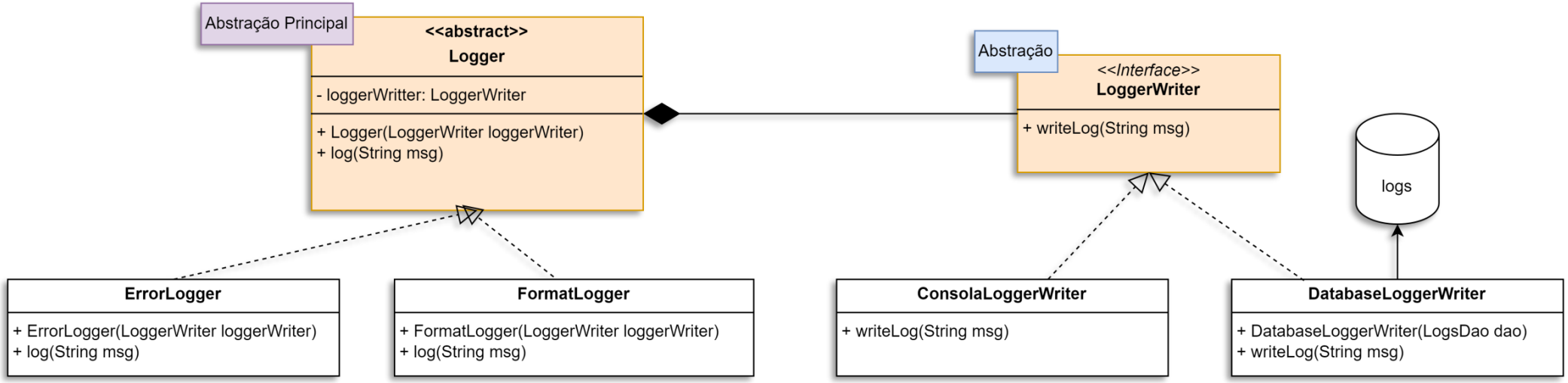
BRIDGE

É um padrão de design estrutural que tem como objetivo desacoplar uma abstração de sua implementação, permitindo que ambas possam variar independentemente.

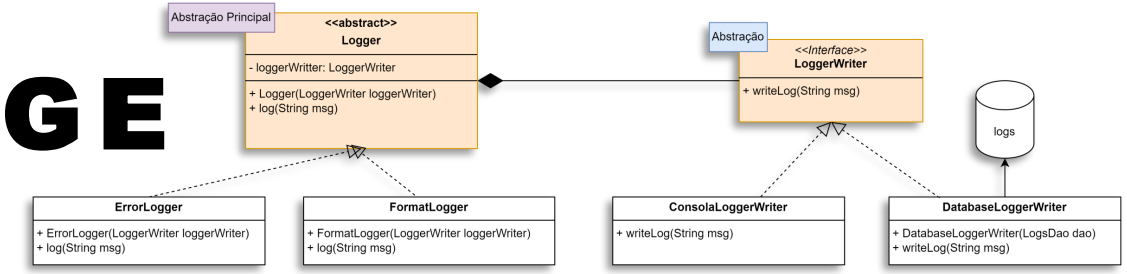


BRIDGE

```
// Consola
LoggerWriter clw = new ConsolaLoggerWriter();
Logger errorLogger = new ErrorLogger(clw);
errorLogger.log("log com consola e erro");

Logger formatLogger = new FormatLogger(clw, "formatado: %s");
formatLogger.log("log com consola e formatado");

// database
LoggerWriter dbw = new DatabaseLoggerWriter(logsDao);
Logger errorLoggerDb = new ErrorLogger(dbw);
errorLoggerDb.log("log com database e erro");

Logger formatLoggerDb = new FormatLogger(dbw, "formatado: %s");
formatLoggerDb.log("log com database e formatado");
```



BRIDGE



Componentes

- **Abstraction**: a interface com os métodos que devem ser implementados.
- **RefinedAbstraction**: extende a interface Abstraction e implementa métodos adicionais
- **Implementor**: define a interface para as classes de implementação
- **ConcreteImplementor**: implementa a interface Implementor.

Benefícios

- **Desacoplamento**: As abstrações e implementações podem crescer de forma independente.
- **Facilidade de testes**: o isolamento dos componentes facilita os testes

Desvantagens

- **Complexidade adicional**: introdução de múltiplas hierarquias de classes aumenta sempre a complexidade.
- **Sobrecarga de código**: os padrões requerem sempre algum código não functional, adicional.
- **Desempenho**: abstrações criam camadas intermédias.
- **Manutenção complexa**: quando se usa padrões, o código fica de alguma forma mais coupled ao padrão.

Casos de uso

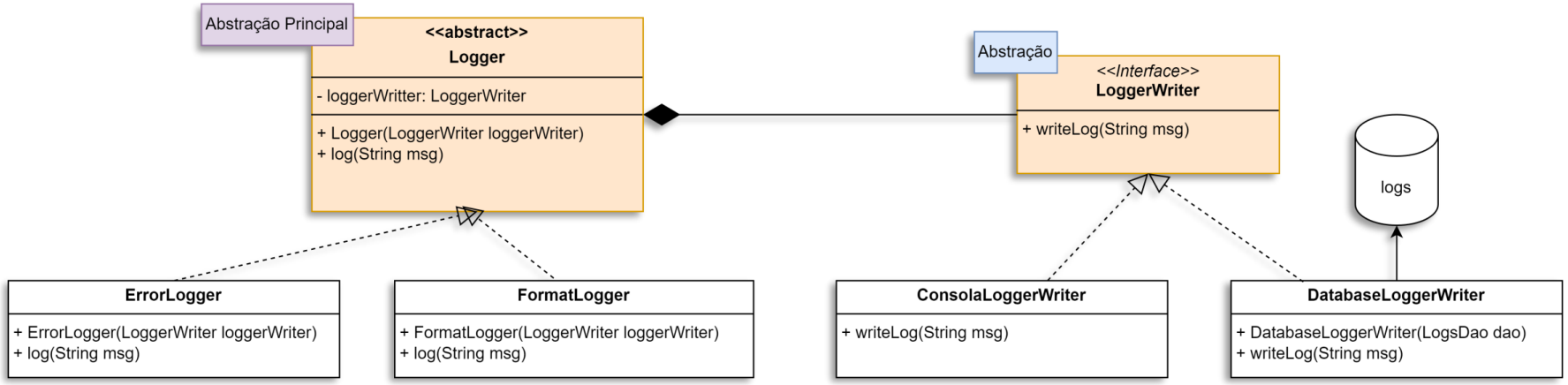
- **Multiplataforma**: quando a interface do utilizador deve estar separada da lógica da Plataforma.
- **Drivers**: separação de diferentes tipos de dispositivos do hardware
- **APIs de serviços**: quando se pretende trabalhar com diversos fornecedores de cloud.
- **Sistemas de base de dados**: separação da lógica de negócio e de acesso aos dados.

BRIDGE

Actividade: Logger (fácil)

Criar um programa usando o padrão Bridge, que cria uma gestão de logs.

- Um **LoggerWriter** pode escrever para a consola ou para uma base de dados **H2** → tabela **logs(id, mensagem, data)**
- Um **Logger** tem duas subclasses:
  - **ErrorLogger**: escreve *System.out.println("ERROR: " + msg)*
  - **FormatLogger:** permite formatar a mensagem: `String format = "LOG: %s";`



BRIDGE

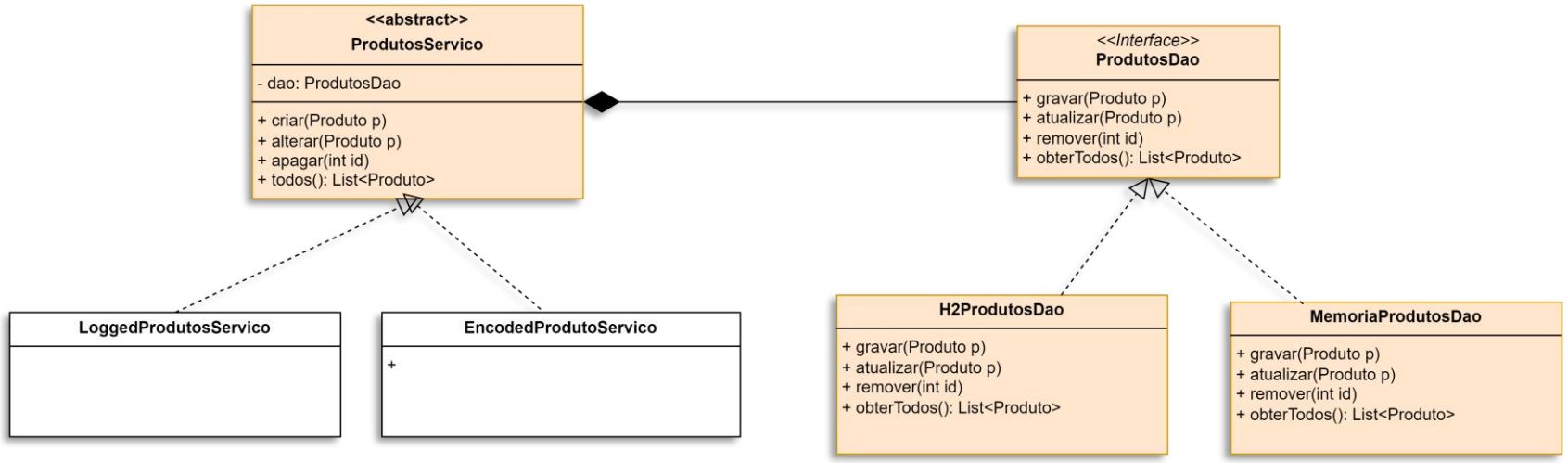
Actividade:Base de dados Memória e H2 (médio)

Criar um programa usando o padrão Bridge, que permite criar duas hierarquias: (ver diagrama)

1. **ProdutosDao**, com duas subclasses: uma implementação **H2** (tabela **produtos**) e outra para **Memória** (**List** produtos)

2. **ProdutosServico**, com duas subclasses: uma que faz *log* (System.out.println) e outra que faz um encode em *Base64* do nome do produto.

- Criar uma classe **Produto (id, nome, preco)**, e uma tabela na base de dados H2



BRIDGE

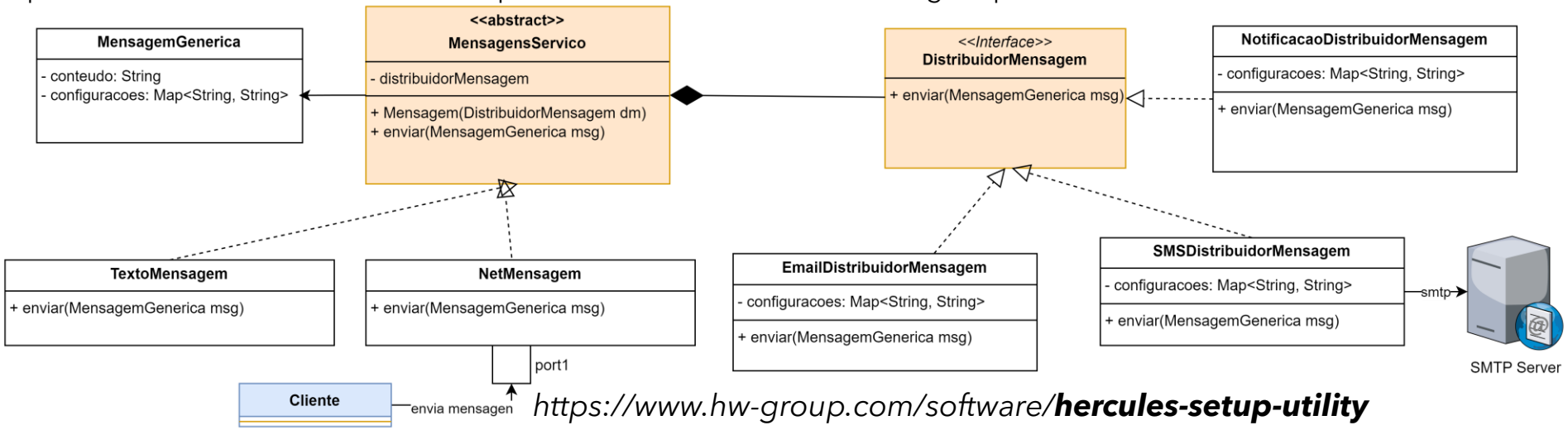
Actividade: Mensagens e Distribuidores (avançado)

Criar um programa usando o padrão Bridge, que dado varios tipos de mensagens, como texto, possam ser enviadas por SMS, Email ou Notificação (Distribuidores)

- Não é necessário implementar o envio por sms (*System.out*) ou notificações, mas poderá valer a pena enviar o email, usando um serviço de smtp! (**mailpit**).

- Opcional: Criar uma pequeno servidor a escuta num socket no port 10000 e redirecionar as mensagens para um dos distribuidores.

```
<dependency>
    <groupId>com.sun.mail</groupId>
    <artifactId>javax.mail</artifactId>
    <version>1.6.2</version>
</dependency>
```



*https://www.hw-group.com/software/**hercules-setup-utility***

NO JDK

java.awt.Component e suas subclasses:

A classe **Component** usa o padrão Bridge para separar a abstração da implementação específica da plataforma.

ANTI PATTERNS

Uso desnecessário: quando a abstração e implementação não precisam de crescer em separado

Hierarquias infladas: quando as hierarquias crescem muito

Falta de coesão: a abstração e a implementação não estão coesas

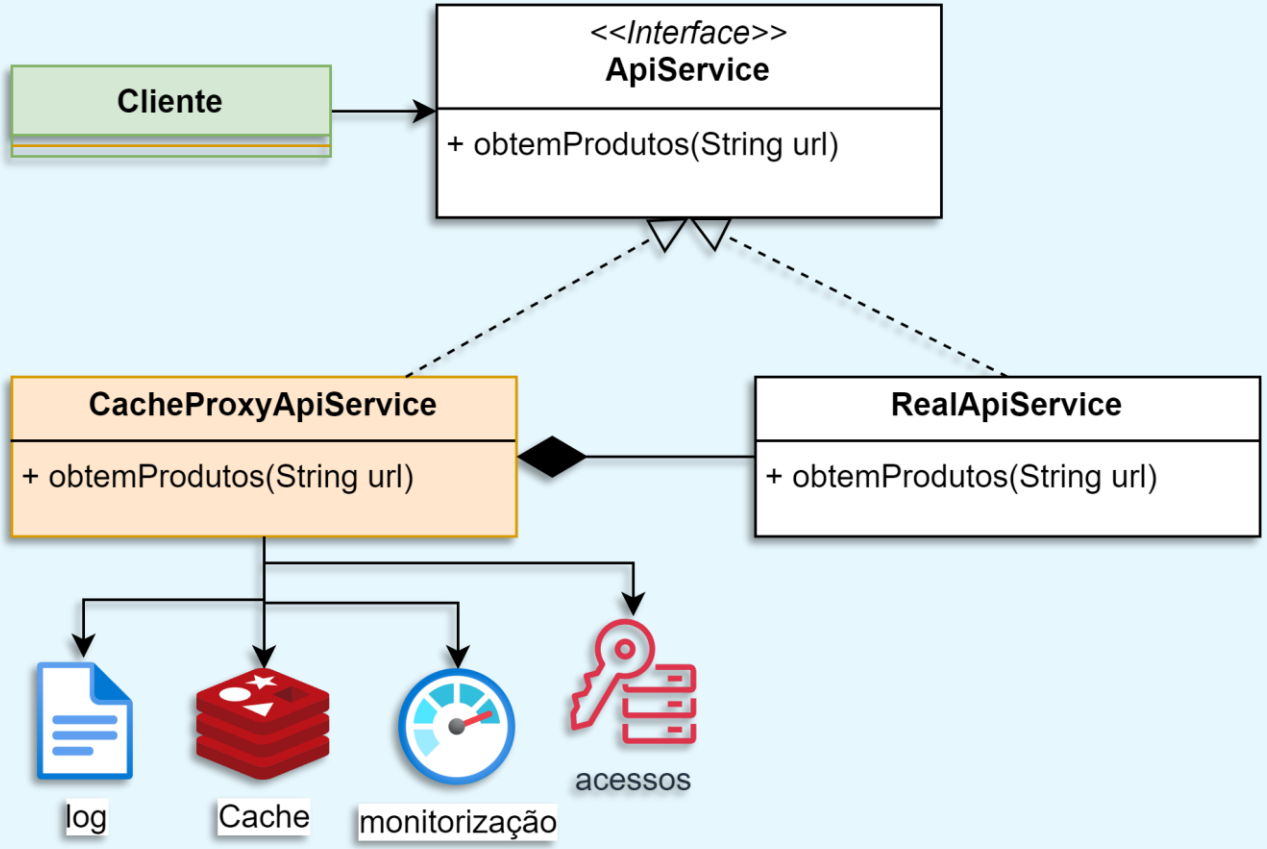
Acoplamento excessivo: mesmo após o uso do Bridge a abstração e a implementação continuam com um coupling forte.

Uso do Bridge ao invés do Adapter: por vezes o problema é mais simples do que parece, e o Adapter seria suficiente

Inflexibilidade: quando a implementação do padrão não permite a adição de novas abstrações e implementações.

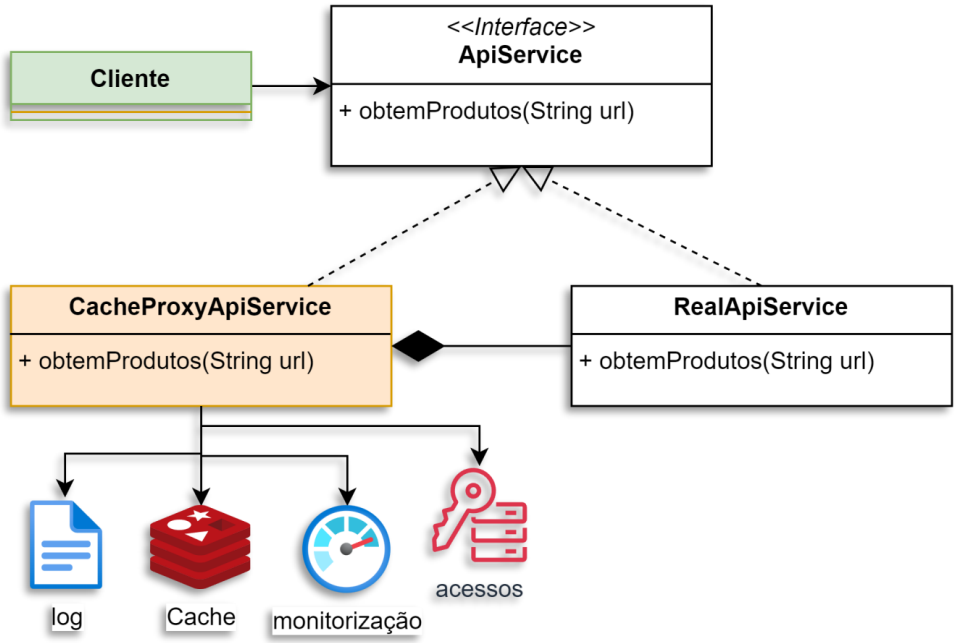
PROXY

O padrão Proxy é um padrão de design estrutural que atua como um intermediário entre um cliente e um objeto real.

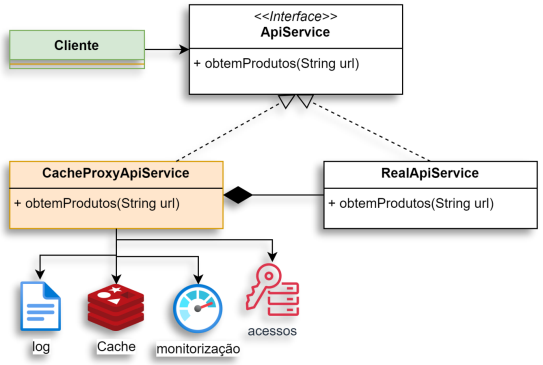


PROXY

```
public static void main(String[] args) {
    ApiService apiService = new CachedApiServiceProxy();

    System.out.println(apiService.obtemProdutos("/produtos"));
    System.out.println(apiService.obtemProdutos("/produtos/tech"));
    System.out.println(apiService.obtemProdutos("/produtos/books"));
}
```



PROXY



Componentes

- **Subject**: a interface para o RealSubject e para o Proxy. Assim o Proxy pode ser usado como se fosse o RealSubject.
- **RealSubject**: a classe real, a classe destino, ou sistema.
- **Proxy**: a classe que controla o acesso ao RealSubject

Benefícios

- **Controlo de acessos**: permite a implementação de controlo de acessos a um recurso
- **Lazy Loading**: o objeto real só é chamado quando for realmente necessário
- **Cache**: um proxy pode ter ou aceder a sistemas de cache, evitando tráfego e reduzindo a latência.
- **Registo e monitorização**: logging, monitorização, audit, etc.
- **Proxies remotos**: em sistemas distribuídos abstrai acesso a objetos remotos, em múltiplas clouds, data centers, etc.

Desvantagens

- **Desempenho**: o proxy é um objeto intermédio que pode estar a "esconder" processamento.
- **Latência**: se o proxy estiver a executar diversas ações sobre os pedidos.
- **Configuração**: pode-se tornar complexo em sistemas distribuídos, que usam urls e outros valores que devem estar sincronizados.
- **Compatibilidade**: alterações no objeto real não contempladas no proxy

Casos de uso

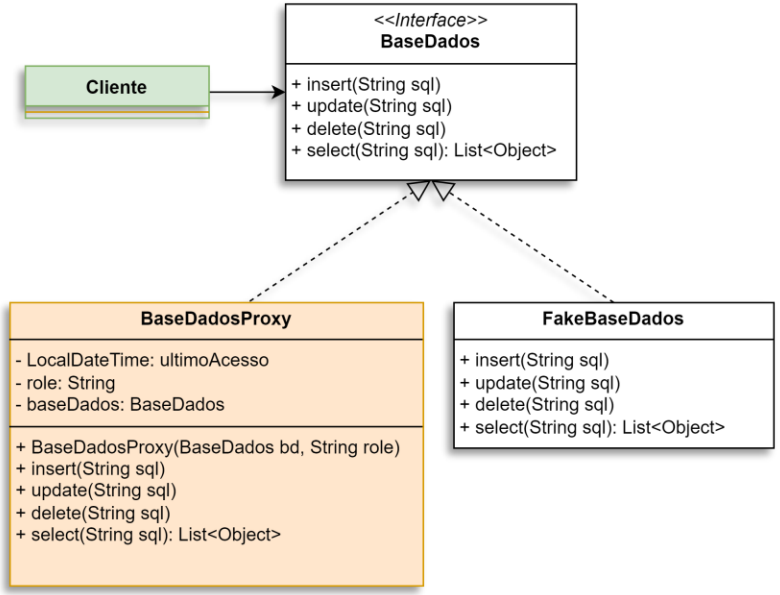
- Lazy Loading
- Cache
- Controlo de acessos
- Monitorização, logging e auditoria
- Facilitar acesso a recursos externos

PROXY

Actividade: base de dados com permissoes e rate (fácil)

Criar um programa usando o padrão Proxy, que dada uma base de dados existente (na forma de uma classe de nome **BaseDados**), permite a execução das queries na **BaseDados,** mas sujeito a permissões e controlo do número de chamadas num intervalo de tempo

- Classe **FakeBaseDados**, com os seguintes métodos. (A implementação pode ser apenas *"System.out"*, e return *List.of();*
  - *void insert(String query)*
  - *void update(String query)*
  - *void delete(String query)*
  - *List<Object> select(String query)*
- Classe **BaseDadosProxy**. Deve ter a mesma assinatura dos métodos, mas:
  - operações de alteração só se o utilizador for admin
    - O role pode ser enviado no constructor do Proxy, ex: "admin"
  - o controlo de chamadas, só permite que qualquer operação tenha um interval de 1 segundo!
    - usar o Thread.sleep() para similar a espera no cliente (main)

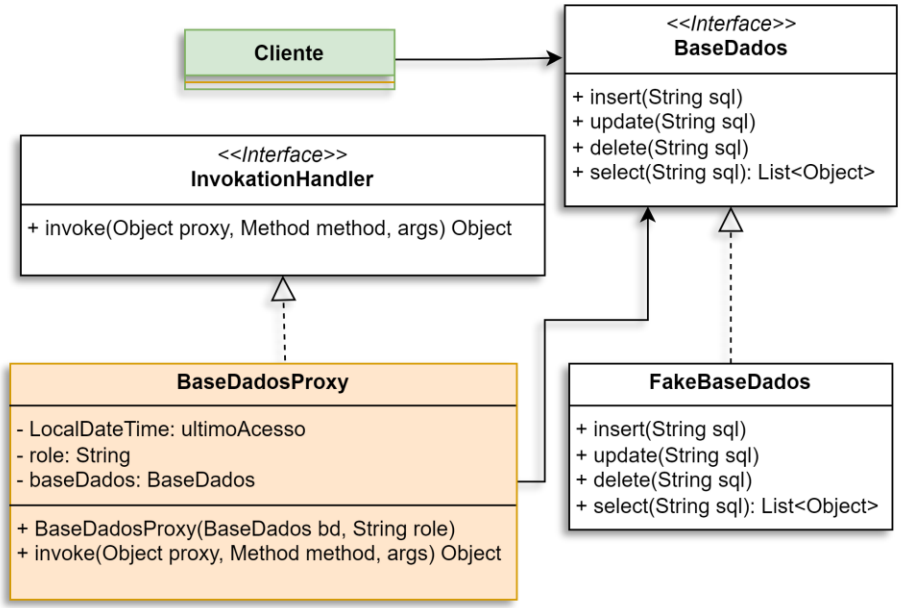


PROXY

Actividade: Usar o "Dynamic Proxy" do Java como Proxy (médio)

Criar um programa usando o padrão Proxy, que dada uma base de dados existente (na forma de uma classe de nome **BaseDados**), permite a execução das queries na **BaseDados,** mas sujeito a permissões e controlo do número de chamadas num intervalo de tempo

- Classe **BaseDados**, com os seguintes métodos. (A implementação podem ser apenas *"System.out"*, e return *List.of();*
  - *void insert(String query)*
  - *void update(String query)*
  - *void delete(String query)*
  - *List<Object> select(String query)*
- Classe **BaseDadosProxy** deve usr o **DynamicProxy** do java. (Cria Proxies!)
  - operações de alteração só se o utilizador for admin
    - A permissão pode ser enviada no constructor do Proxy, ex: "admin"
  - o controlo de chamadas, só permite que qualquer operação tenha um interval de 1 segundo!
    - usar o Thread.sleep() para similar a espera no cliente (main)



DYNAMIC PROXY - JAVA

```java
// Interface
public interface Servico {
    void executar(String tarefa);
}

// Implementação concreta
public class ServicoImpl implements Servico {
    @Override
    public void executar(String tarefa) {
        System.out.println("Executando tarefa: " + tarefa);
    }
}

public class ServicoInvocationHandler implements InvocationHandler {
    private final Servico servico;

    public ServicoInvocationHandler(Servico servico) {
        this.servico = servico;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        System.out.println("Antes de executar o método: " + method.getName());
        Object result = method.invoke(servico, args);
        System.out.println("Depois de executar o método: " + method.getName());
        return result;
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Servico servicoReal = new ServicoImpl();
        ServicoInvocationHandler handler = new
ServicoInvocationHandler(servicoReal);

        Servico proxy = (Servico) Proxy.newProxyInstance(
                Servico.class.getClassLoader(),
                new Class[]{Servico.class},
                handler
        );

        proxy.executar("Tarefa 1");
    }
}
```

PROXY

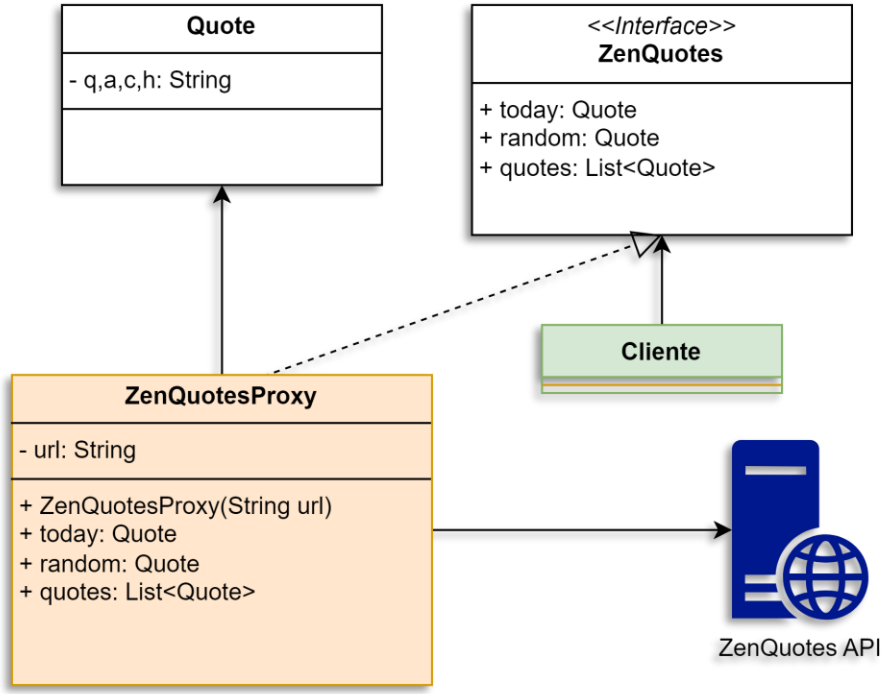
Actividade: Usar API externa – zenquotes (avançado)

Criar um programa usando o padrão Proxy, que dado uma API externa, crie uma classe que faça a mediação dos pedidos

- Api: https://zenquotes.io/
  - https://zenquotes.io/api/quotes - Gera um JSON com 50 quotes
  - https://zenquotes.io/api/today - Gera a "quote" do dia
  - https://zenquotes.io/api/random - Gera uma "quote" aleatória
- Class proxy, deverá ter uma método para cada API
  - *List<Quote> quotes()*
  - *Quote today()*
  - *Quote random()*


```
[
    {
        "q": "We are always complaining that our days are few,
        "a": "Seneca",
        "h": "<blockquote>&ldquo;We are always complaining that
    }
]
```


```
<dependency>
    <groupId>org.json</groupId>
    <artifactId>json</artifactId>
    <version>20210307</version>
</dependency>
<dependency>
    <groupId>org.apache.httpcomponents</groupId>
    <artifactId>httpclient</artifactId>
    <version>4.5.13</version>
</dependency>
```



NO JDK

```java
class ServiceInvocationHandler implements InvocationHandler {
    private final Object realService;

    public ServiceInvocationHandler(Object realService) {
        this.realService = realService;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
        System.out.println("Antes da chamada do metodo");
        Object result = method.invoke(realService, args);
        System.out.println("Depois da chamada");
        return result;
    }
}

public class DynamicProxyExample {
    public static void main(String[] args) {
        Service realService = new RealService();
        Service proxyInstance = (Service) Proxy.newProxyInstance(
                realService.getClass().getClassLoader(),
                realService.getClass().getInterfaces(),
                new ServiceInvocationHandler(realService)
        );

        proxyInstance.perform();
    }
}
```

java.lang.reflect.Proxy

```java
public static void main(String[] args) {
    try {
        Proxy proxy = new Proxy(Proxy.Type.HTTP,
                                new InetSocketAddress("example.com", 8080));
        URL url = new URL("http://www.example.com");
        URLConnection connection = url.openConnection(proxy);
        connection.connect();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

java.net.Proxy

```java
public static void main(String[] args) {
    try {
        MyRemoteService service = new MyRemoteServiceImpl();
        Registry registry = LocateRegistry.createRegistry(1099);
        registry.bind("MyRemoteService", service);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

java.rmi.server.Proxy

ANTI PATTERNS

Proxy Inflado: Adicionar muitas responsabilidades ao proxy além de simplesmente delegar chamadas. Viola o princípio da responsabilidade única

Proxy Desnecessário: Usar proxies quando não há necessidade real de controle adicional ou funcionalidades extras.

Proxy Transparente: O proxy não adiciona valor significativo e apenas passa as chamadas diretamente para o objeto real.

Proxy com Estado: Manter estado no proxy que deveria estar no objeto real. Pode levar a inconsistências e dificuldades na sincronização do estado entre o proxy e o objeto real.

Proxy com Lógica de Negócio: Implementar lógica de negócio no proxy em vez de no objeto real.

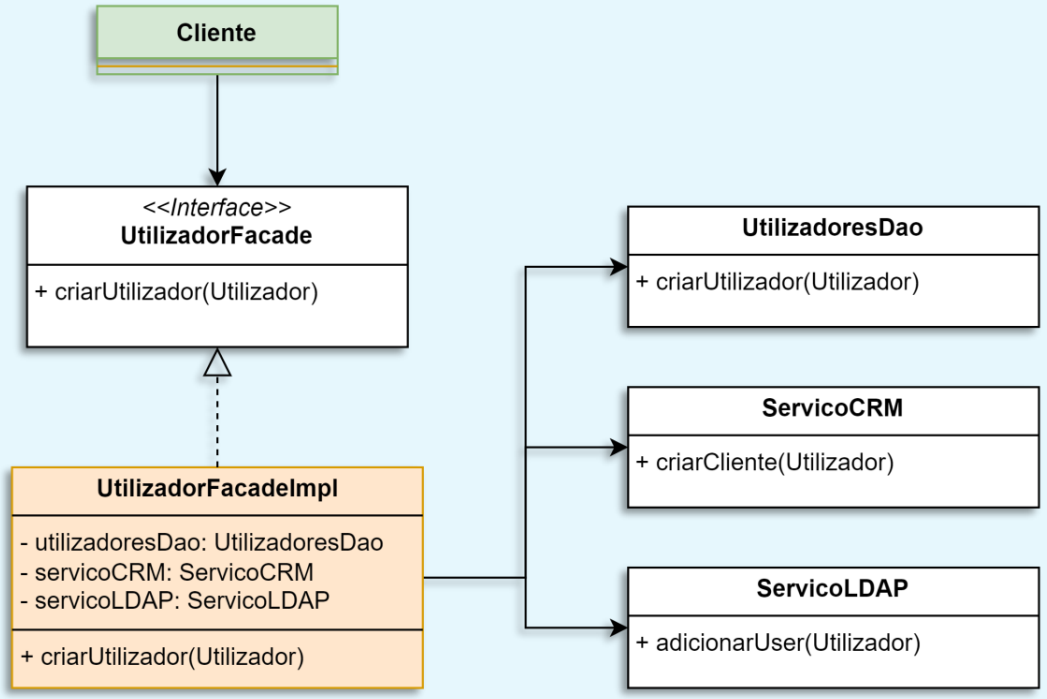
Proxy Circular: Criar uma situação onde o proxy depende do objeto real e vice-versa.

Proxy com Dependências Fortes: O proxy depende fortemente de detalhes específicos da implementação do objeto real.

Proxy com Performance Degradada: O proxy adiciona uma sobrecarga significativa, degradando a performance do sistema.

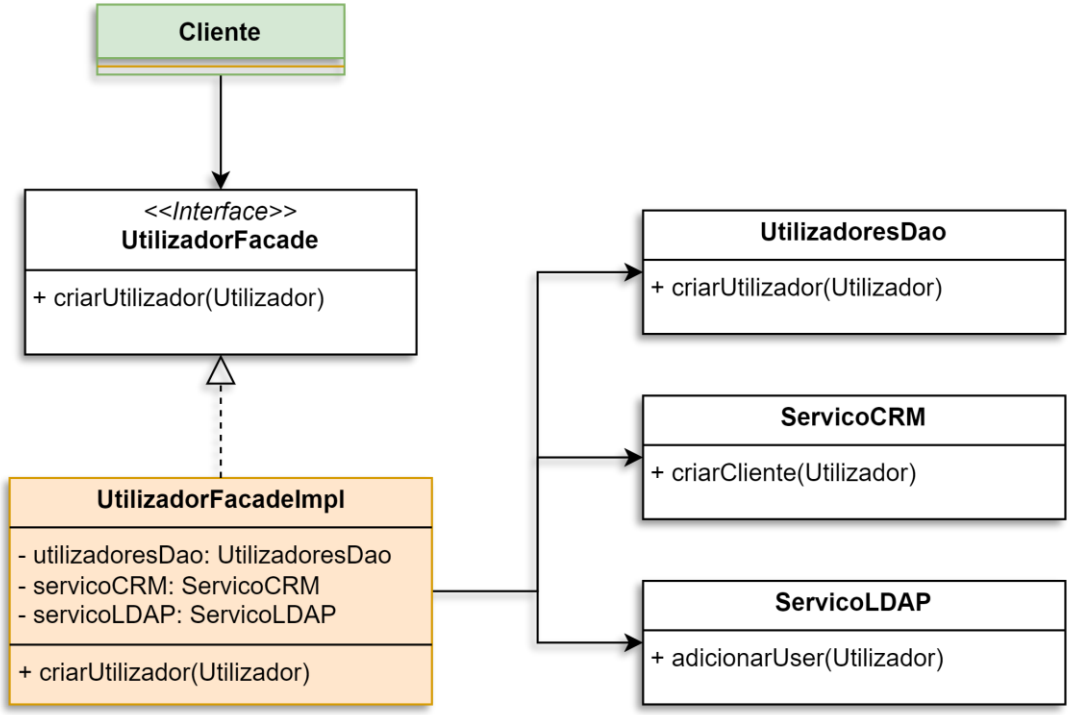
FACADE

O padrão Facade (ou Fachada) é um padrão de design estrutural que fornece uma interface simplificada para um conjunto de interfaces em um subsistema complexo.

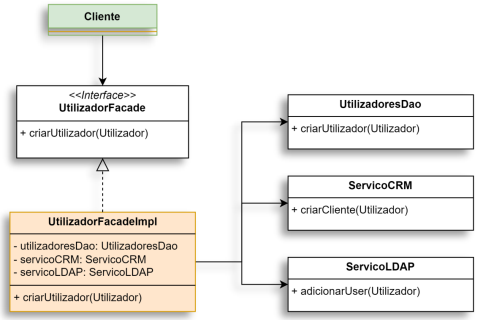


FACADE

```
public static void main(String[] args) {
    UtilizadorFacade facade = new UtilizadorFacadeImpl();
    Utilizador utilizador = new Utilizador("joao", "senha123", "joao@example.com");
    facade.createUser(utilizador);
}
```



FACADE



Componentes

- **Facade**: Uma classe com um interface simples para os clientes.
- **Subsistema**: Onde está o trabalho real, a complexidade.

Benefícios

- **Simplificação**: O interface fica simples para os clientes, escondendo os detalhes e sequência da funcionalidade.
- **Redução de dependências**: Os clientes ficam menos "coupled" com os subsistemas.
- **Facilidade**: Simples acesso, só com o que interessa.
- **Manutenção**: Os subsistemas podem ser alterados sem impacto nos clientes.
- **Organização do código**: O código fica mais claro nos clientes.

Desvantagens

- **Encapsulamento**: O Facade pode esconder demasiado acerca das funcionalidades, não deixando espaço para customizações do lado do cliente.
- **Flexibilidade**: o cliente deixa de ter acesso a chamadas de baixo nível
- **Dependência**: O Facade pode-se tornar um ponto único de falha.
- **Manutenção**: alterações nos subsistemas do Facade pode trazer complicações, devido a impactos "desconhecidos" nas funcionalidades de alto nível.

Casos de uso

- **Sistemas complexos**
- **Biblioticas e frameworks**
- **Aplicações multicamadas**: no negócio, por vezes, existe uma Facade antes do negócio, para abstrair múltiplas chamadas ao negócio para concretizar uma funcionalidade.
- **Integração de Sistemas**
- **Legacy**
- **Automação de processos**

FACADE

Actividade: Criar utilizador (fácil)

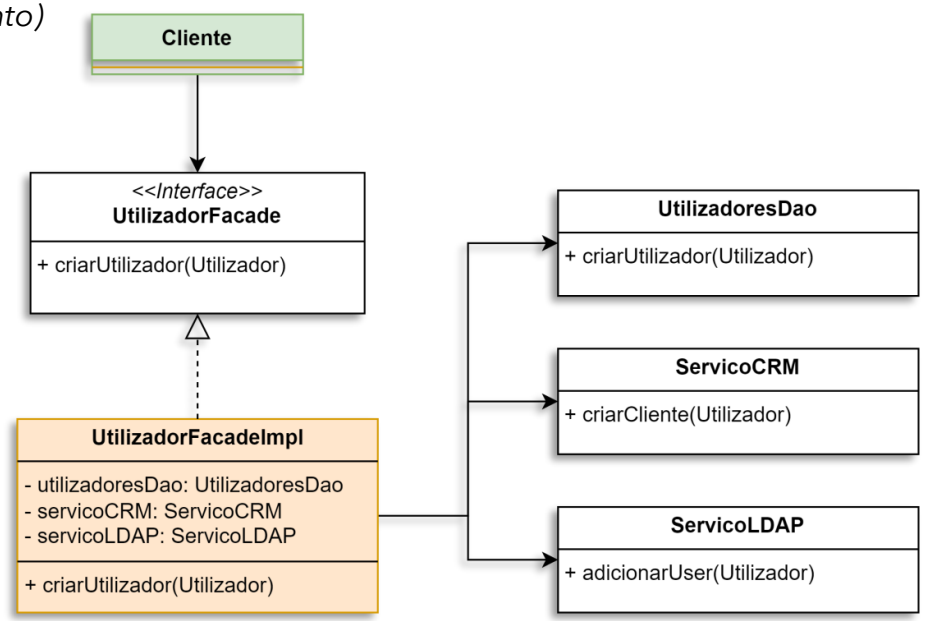
Criar um programa usando o padrão Facade, que realiza a criação de um utilizador em 3 Sistemas (ver diagrama)

Criar 3 classes que representam 3 subsistemas (todas criam um utilizador/cliente)

- **UtilizadoresDao** (base de dados, tabela *utilizadores (username, nome, dataNascimento)*
- **ServicoCRM** (um subsistemas de CRM. A classe não tem de ter funcionalidade)
- **ServicoLDAP** (um subsistema de LDAP. A classe não tem de ter funcionalidade)

Criar a interface e classe Facade

- Método *criarUtilizador(Utilizador)* que chama todos os 3 subsistemas.

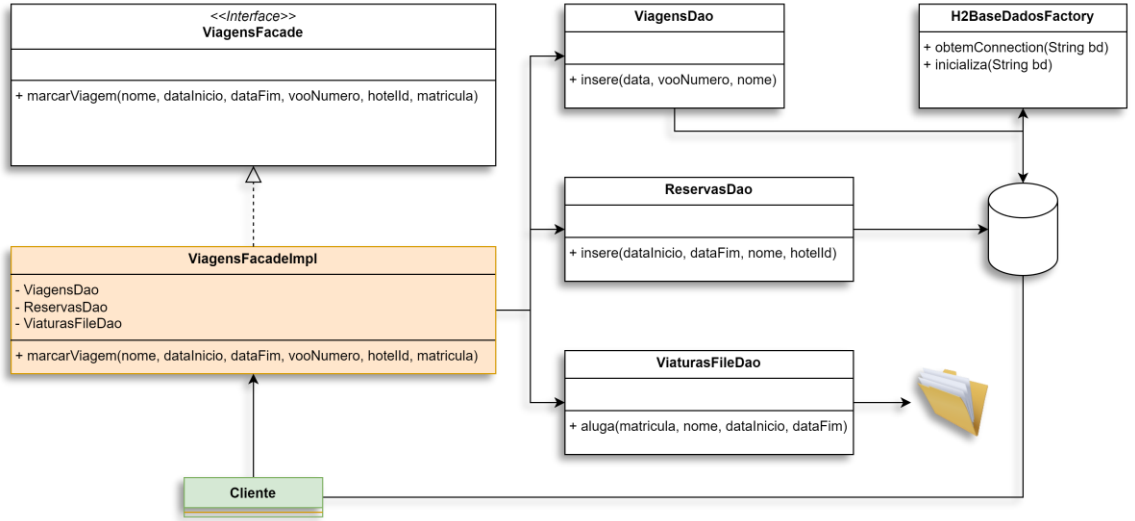


FACADE

Actividade: Marcar viagem (médio)

Criar um programa usando o padrão Facade, que permite marcar uma viagem tendo em conta, avião, hotel e aluguer de uma viaturas

- Os dados dos avião ficarão numa tabela numa base de dados (avioesdb) H2: **viagens** (id, data, vooNumero, nome)
- Os dados para o Hotel ficarão numa outra base de dados (hoteisdb) H2: **reservas**(id, dataInicio, dataFim, nome, hotelId)
- O aluguer de uma viatura deve ser criado num ficheiro **viaturas-yyyy-mm-dd.txt** (matricula, nome, dataInicio, dataFim)



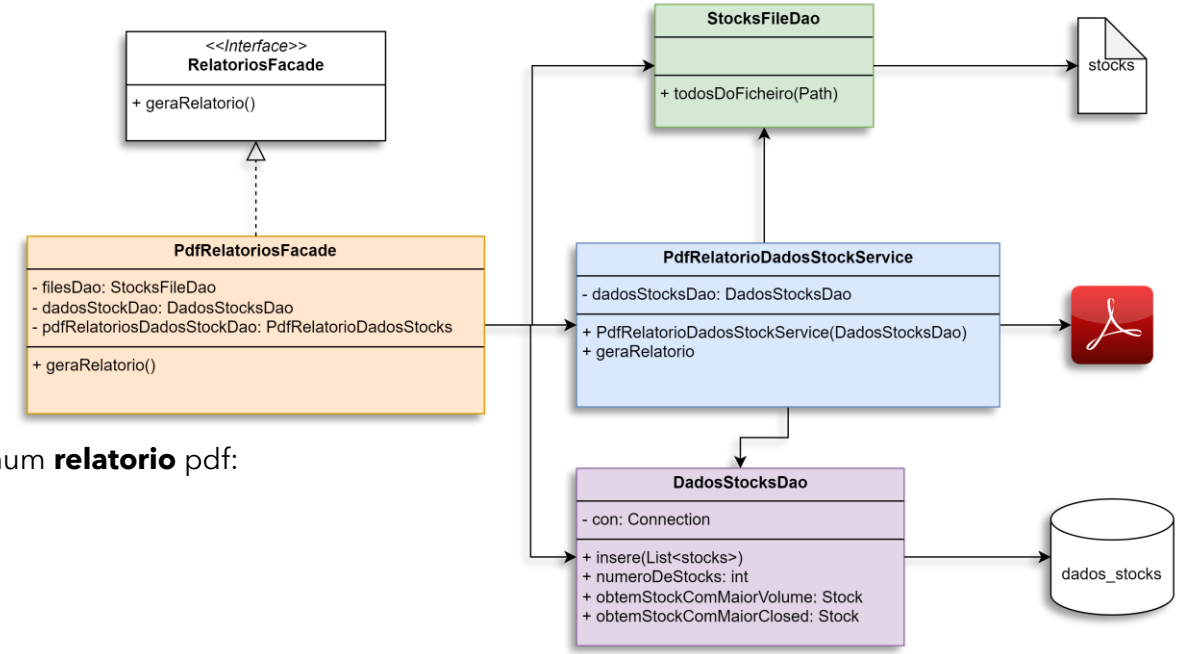
FACADE

Actividade: geração de relatórios (avançado)

Criar um programa usando o padrão Facade, que acede a um ficheiro com informação de stocks, importa os dados para uma tabela, calcula algumas estatisticas e produz um relatório em pdf.

A solução tém 3 subsistemas

- Um sistema que disponibiliza acesso a **ficheiros** (stocksDao)
    - usar o ficheiro em anexo **dados-stocks.csv** e colocar na pasta /**resources**
- Um sistema que insere e obtem stocks de uma tabela
    - tabela **base dados** H2
    - tabela: **dados_stocks** *(id, symbol,date,open,high,low,close,volume)*
- Um sistema que lê a tabela **dados_stocks** e produz as seguintes estatísticas num **relatorio** pdf:
    - Quantas ações/stocks diferentes existem.
    - Qual a ação/stock e em que data apresentou o maior volume
    - Qual a ação/stock que fechou com o valor mais elevado de sempre



NOTA: Uma biblioteca de pdf possível de usar: **openpdf**

NO JDK

```
javax.faces.context.FacesContext

public class FacesContextExample {
    public void exampleMethod() {
        FacesContext context = FacesContext.getCurrentInstance();
        HttpServletRequest request = (HttpServletRequest) context.getExternalContext().getRequest();
        String param = request.getParameter("numero");
        System.out.println("Valor do parametro: " + param);
    }
}
```

```
java.nio.file.Files

public static void main(String[] args) {
    Path path = Paths.get("ficheiro.txt");

    try {
        List<String> lines = Files.readAllLines(path);
        lines.forEach(System.out::println);

        Files.write(path, "Novo conteudo".getBytes(), StandardOpenOption.APPEND);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

ANTI PATTERNS

Facade Inflado: Adicionar muitas responsabilidades ao Facade

Facade Transparente: O Facade não adiciona valor significativo e apenas passa as chamadas diretamente para os subsistemas.

Dependência Forte no Facade: Tornar o código cliente fortemente dependente do Facade.

Facade como Solução Temporária: Usar o Facade como uma solução temporária para esconder a complexidade, mas nunca refatorar o código subjacente.

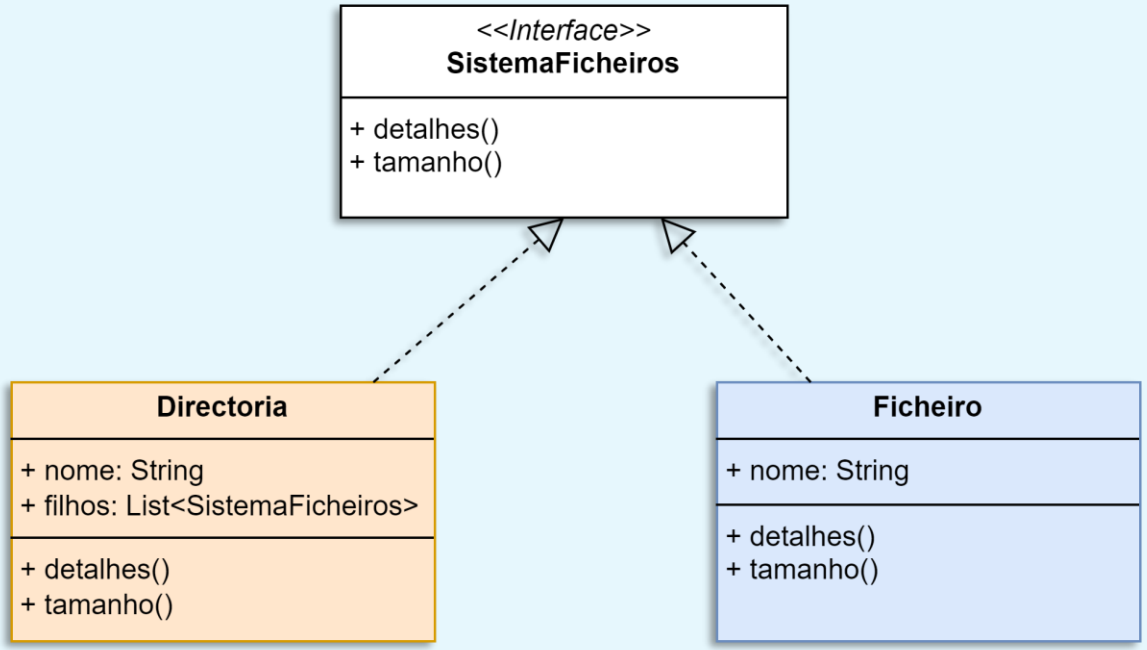
Falta de Encapsulamento: O Facade expõe detalhes internos dos subsistemas em vez de encapsulá-los.

Facade com Lógica de Negócio: Implementar lógica de negócio no Facade em vez de nos subsistemas apropriados.

Facade com Estado: Manter estado no Facade que deveria estar nos subsistemas.

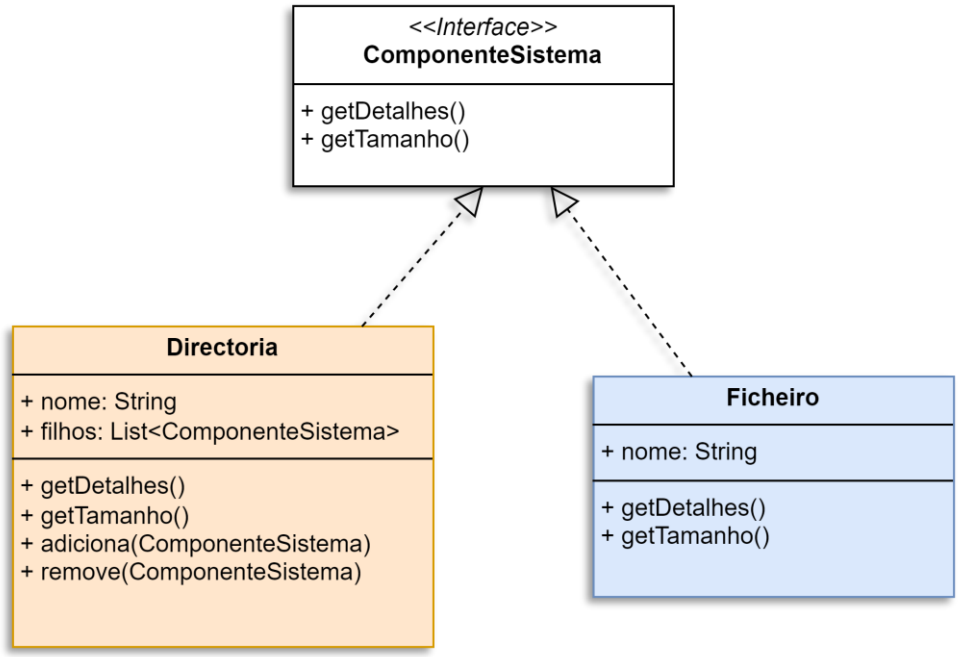
COMPOSITE

O padrão Composite é um padrão de design estrutural que permite tratar objetos individuais e composições de objetos de maneira uniforme, formando uma estrutura hierárquica.

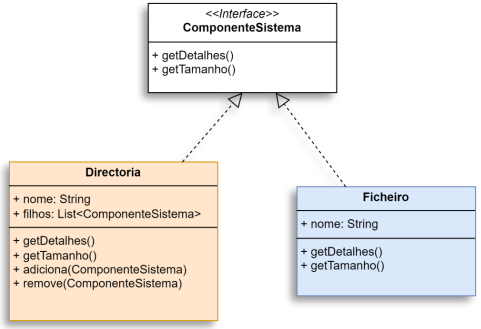


COMPOSITE

```java
public static void main(String[] args) {
    ComponenteSistema file1 = new Ficheiro("doc1.txt");
    ComponenteSistema file2 = new Ficheiro("doc2.txt");
    ComponenteSistema file3 = new Ficheiro("foto.jpg");

    Directoria dir1 = new Directoria("documentos");
    Directoria dir12 = new Directoria("imagens");

    dir1.adiciona(file1);
    dir2.adiciona(file2);
    dir12.adiciona(file3);

    Directoria raiz = new Directoria("root");
    raiz.adiciona(dir1);
    raiz.adiciona(dir12);

    raiz.getDetalhes();
}
```



COMPOSITE



Componentes

- **Componente**: A interface com a definição das operações a serem aplicadas aos objetos da composição.
- **Folha**: Os objetos "folha"/terminais que não têm filhos
- **Composto**: Um objeto que pode ter filhos.

Benefícios

- **Recursividade**: Permite lidar com sistemas hierarquicos, como sistemas de ficheiros, organizações, documentos, etc.
- **Simplicidade**: Uniformiza a forma de lidar com as hierarquias
- **Reutilização de código**: uma vantagem dos padrões é a sua reutilização. Muitas linguagens de programação aceitam genéricos, ou tipos de dados parameterizados.
- **Flexibilidade**: torna simples a adição de composições ou objetos simples.

Desvantagens

- **Complexidade**: Uso de hierarquias traz sempre os seus desafios.
- **Desempenho**: hierarquias longas e largas podem trazer impactos na execução dos algoritmos e memória.
- **Manutenção e troubleshooting**: a gestão de hirarquias pode começar a tornar-se difícil de gerir.

Casos de uso

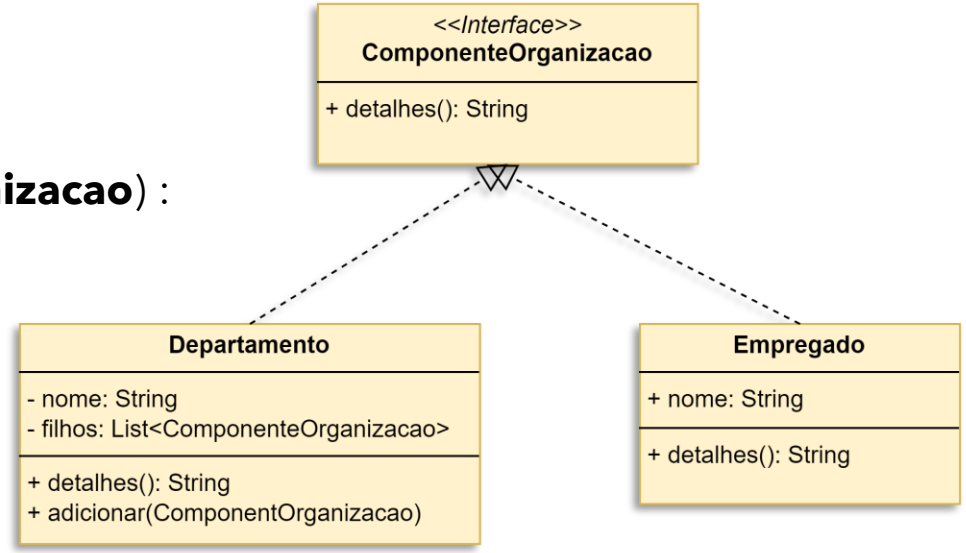
- **Interfaces de utilizador**
- **Sistemas de ficheiros**
- **Estrutura de documentos**
- **Gráficos e desenhos**
- **Compiladores, interpretadores**

COMPOSITE

Actividade: Empregados e departamentos (fácil)

Objetivo: Implementar um sistema simples que use o conceito de departamento e empregados usando o padrão Composite.

- Criar a interface de **ComponenteOrganizacao**
  - deve conter pelo menos o método: "*detalhes()*".
- Criar as classes (implementam o interface **ComponenteOrganizacao**) :
  - **Departamento** (contem *nome* e uma lista de componentes)
  - **Empregado** (contem *nome)*



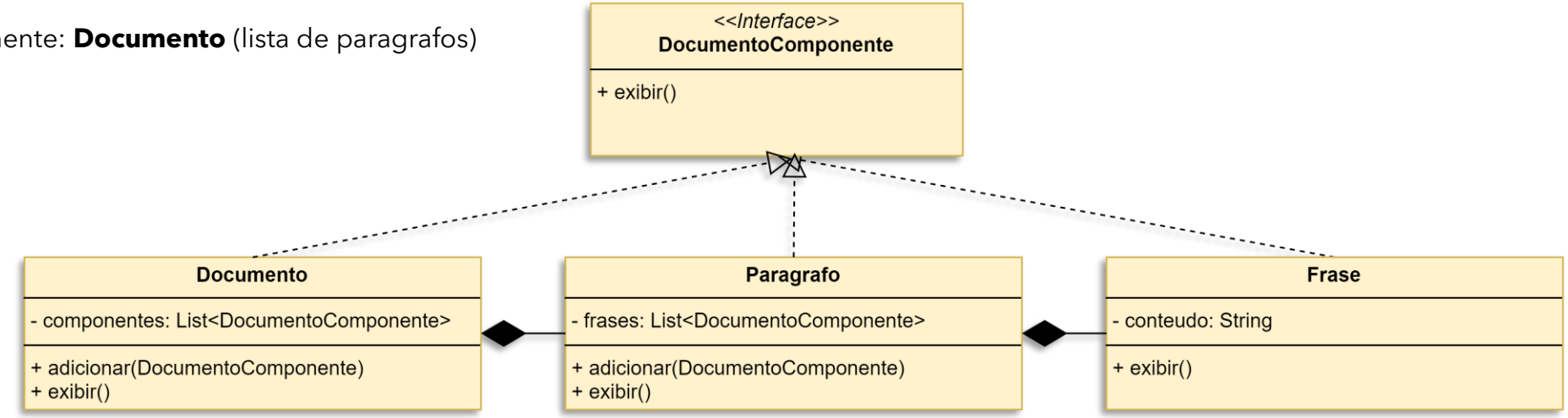
COMPOSITE

Actividade: Documento (fácil)

Objetivo: Implementar um modução simples de um Documento usando o padrão Composite.

Um documento é constituido por componentes (**DocumentoComponente)**

- Componente: **Frase** (string sem <enter> no fim)
- Componente: **Paragrafo** (lista de frases + <enter> no fim)
- Componente: **Documento** (lista de paragrafos)



COMPOSITE

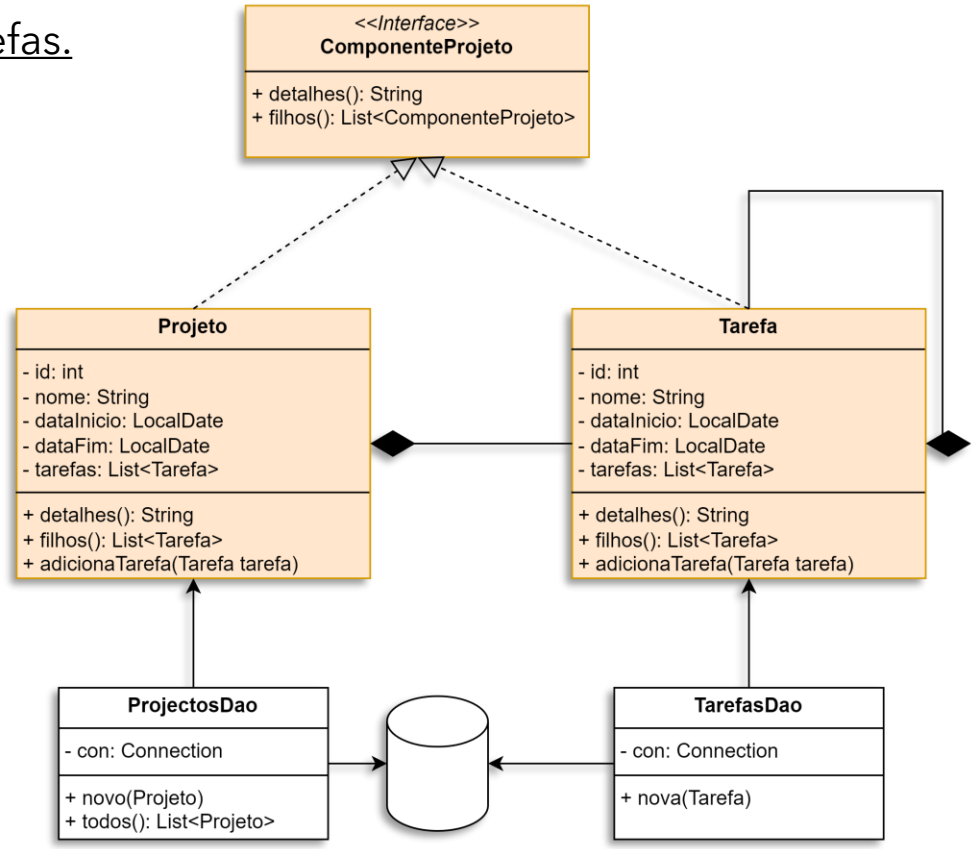
Actividade: Projectos, tarefas e subtarefas (avançado)

Objetivo: Implementar um sistema de gestão de projectos com tarefas.

- Um **Projeto** tem *(id, nome, dataInicio, dataFim, lista de tarefas)*
- Uma **Tarefa** tem *(id, nome, dataInicio, duracao, lista de tarefas)*
- Uma base de dados H2, tem duas tabelas
  - **projetos**
  - **tarefas**

Funcionalidades:

- Dado o id de um projeto exibir a arvore das tarefas
- Criar e listar os projetos
- Criar tarefas em projetos ou em tarefas



NO JDK

```java
// java.util.List
List<String> list = new ArrayList<>();
list.add("Element 1");
list.add("Element 2");
list.add("Element 3");
```

```java
// javax.faces.component.UIComponent
HtmlPanelGroup panelGroup = new HtmlPanelGroup();

UIOutput output1 = new UIOutput();
output1.setValue("Output 1");

UIOutput output2 = new UIOutput();
output2.setValue("Output 2");

panelGroup.getChildren().add(output1);
panelGroup.getChildren().add(output2);
```

```java
// java.awt.Container/JComponent
JFrame frame = new JFrame("Composite Pattern Example");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setSize(300, 200);

JPanel panel = new JPanel();
panel.setLayout(new FlowLayout());

JButton button1 = new JButton("Button 1");
JButton button2 = new JButton("Button 2");

panel.add(button1);
panel.add(button2);

frame.add(panel);
frame.setVisible(true);
```

ANTI PATTERNS

Objeto God: Quando um único objeto assume muitas responsabilidades, tornando-se um ponto central de controle, o que vai contra a ideia de distribuir responsabilidades.

Dependência cíclica: Quando os componentes do Composite têm dependências cíclicas, dificultando a manutenção e a compreensão do código.

Demasiada abstração: Tornar a interface do Composite muito genérica pode levar a uma complexidade desnecessária e a dificuldades na implementação de funcionalidades específicas.

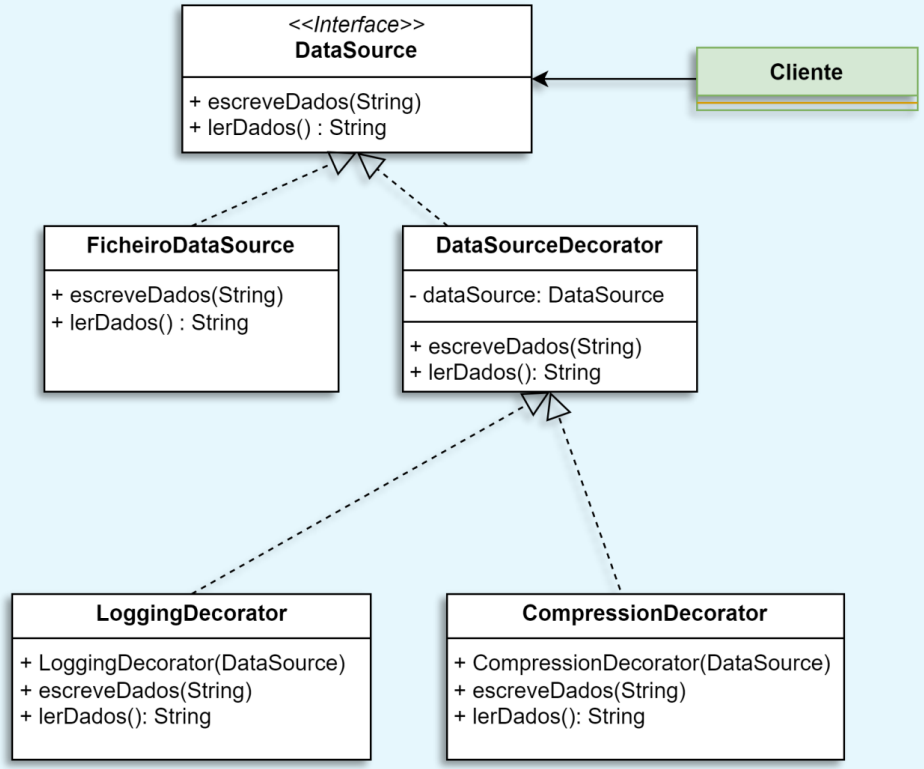
Abstração inconsistente: Quando os métodos na interface do Composite não são aplicáveis a todos os componentes, resultando em implementações vazias ou lançando exceções.

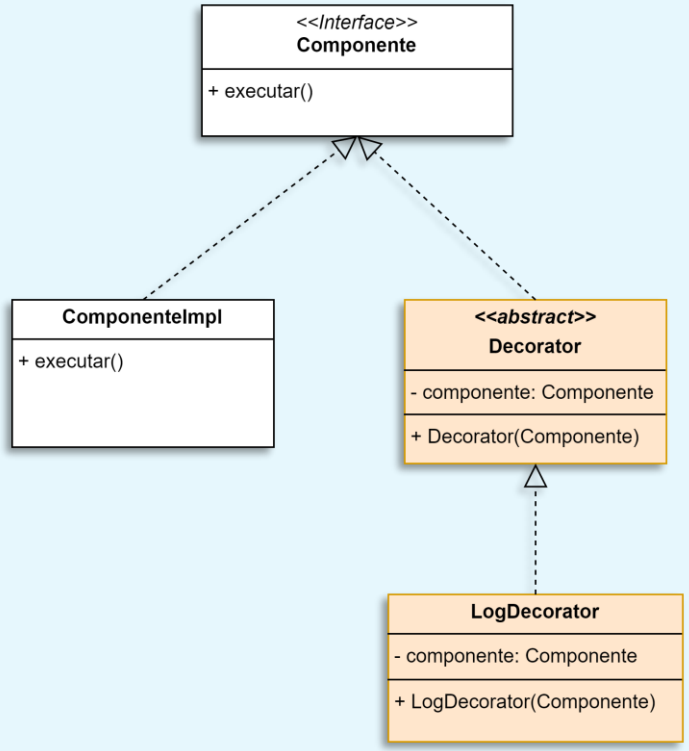
Uso errado do Composite: Utilizar o padrão Composite em situações onde uma estrutura hierárquica não é necessária pode complicar o design sem benefícios claros.

Falta de emcapsulamento: Quando detalhes internos do Composite vazam para os clientes, quebrando o encapsulamento e aumentando o acoplamento.

DECORATOR

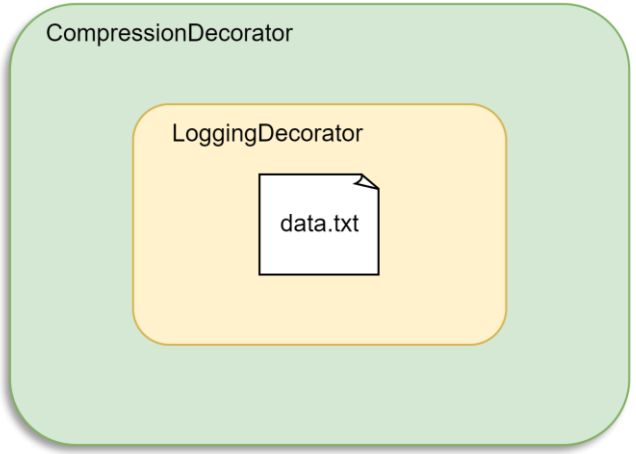
O padrão Decorator é um padrão de design estrutural que permite adicionar comportamentos ou responsabilidades a objetos de forma dinâmica, sem alterar a estrutura original da classe.

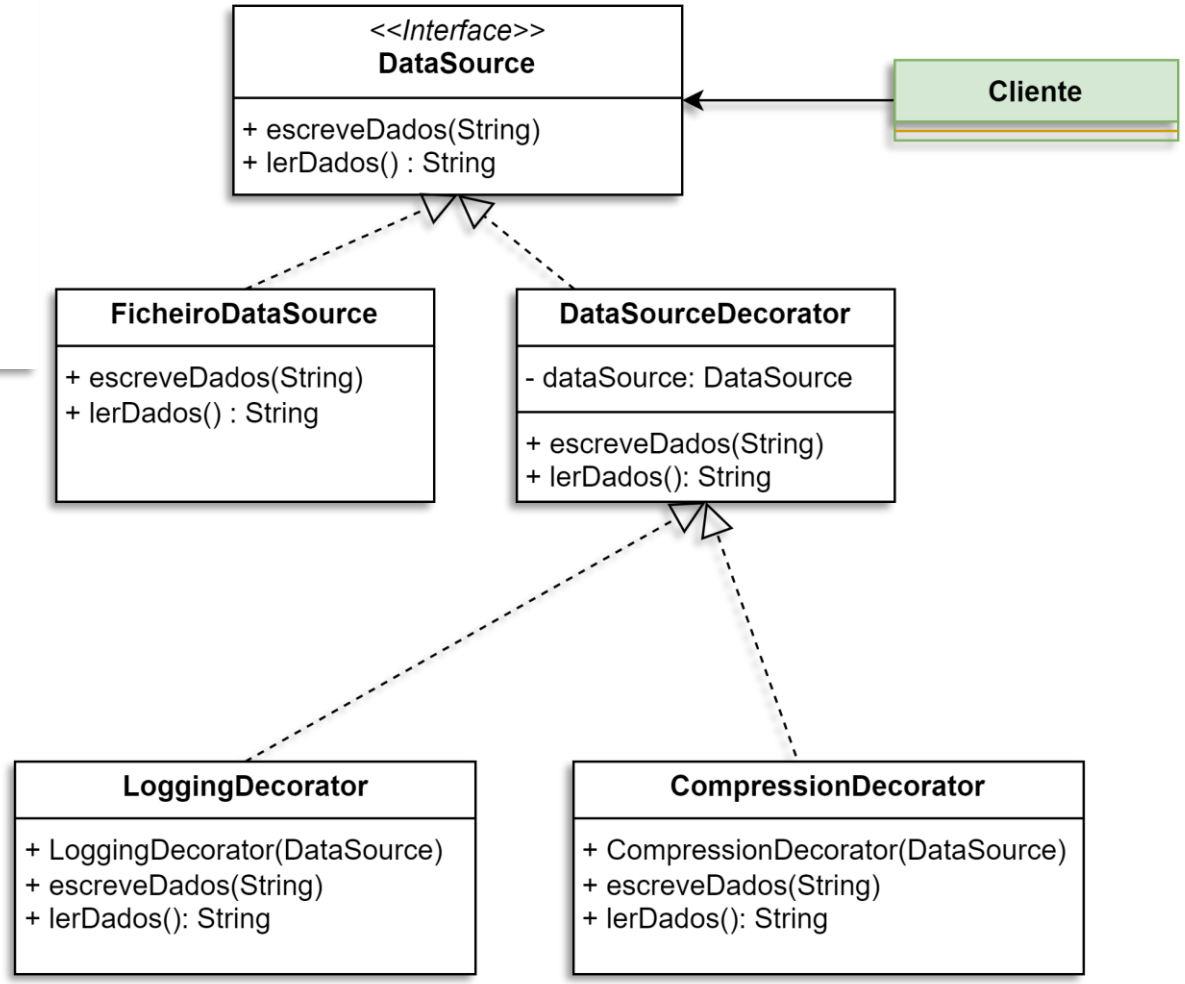




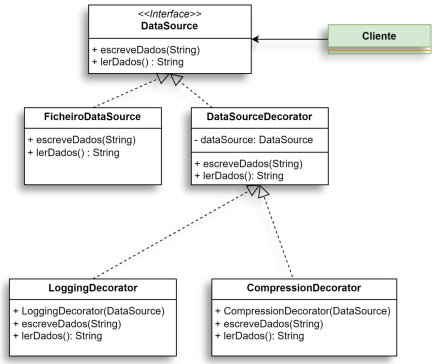
DECORATOR

```java
public static void main(String[] args) {
    DataSource source = new FileDataSource("data.txt");
    source = new LoggingDecorator(source);
    source = new CompressionDecorator(source);

    source.writeData("Olá, Mundo!");
    System.out.println(source.readData());
}
```





DECORATOR



Componentes

- **Componente**: A interface para os objetos que podem ter responsabilidades adicionadas
- **Componentes (concreto)**: A implementação da interface Componente.
- **Decorador Abstrato**: Define uma interface semelhante à interface do componente, mas serve de interface para a criação de operações adicionais
- **Decoradores (concretos)**: Extendem ou implementa o Decorador Abstracto, adicionando funcionalidades.

Benefícios

- **Adição de comportamento**: permite adicionar funcionalidades a um objeto em tempo de execução, sem modificar a classe do objetos existente.
- **Flexibilidade**: para adicionar funcionalidades apenas é necessário criar outra classe e embrulhar a original.
- **Transparência**: Os decoradores implementam a mesma interface que os objetos que estão a decorar.
- **Composição de comportamentos**: pode-se decorar decoradores, indefinidamente.

Desvantagens

- **Complexidade**: se houver varios decoradores de decoradores, pode começar a tornar a leitura dificil.
- **Muitas classes**: muitas classes pequenas com propósitos muito reduzidos.
- **Debug**: o aninhamento das classes pode complicar o troubleshooting.
- **Sobrecarga de performance**: adição de camadas tem impacto da execução do código e na criação de objetos, por vezes de forma “invisivel”.

Casos de uso

- **Adição de comportamentos:** útil em cenários de legacy. Muito usado pelo Java para adicionar funcionalidades em streams como leitura e escrita de ficheiros, compressao, encriptação, rede, etc.
- **Cross-cutting concerns:** ajuda na adição de logs, autenticação, compressão e outros tipos de não funcionalidades.
- **Legacy:** usado em operações de refactor para sistemas legacy

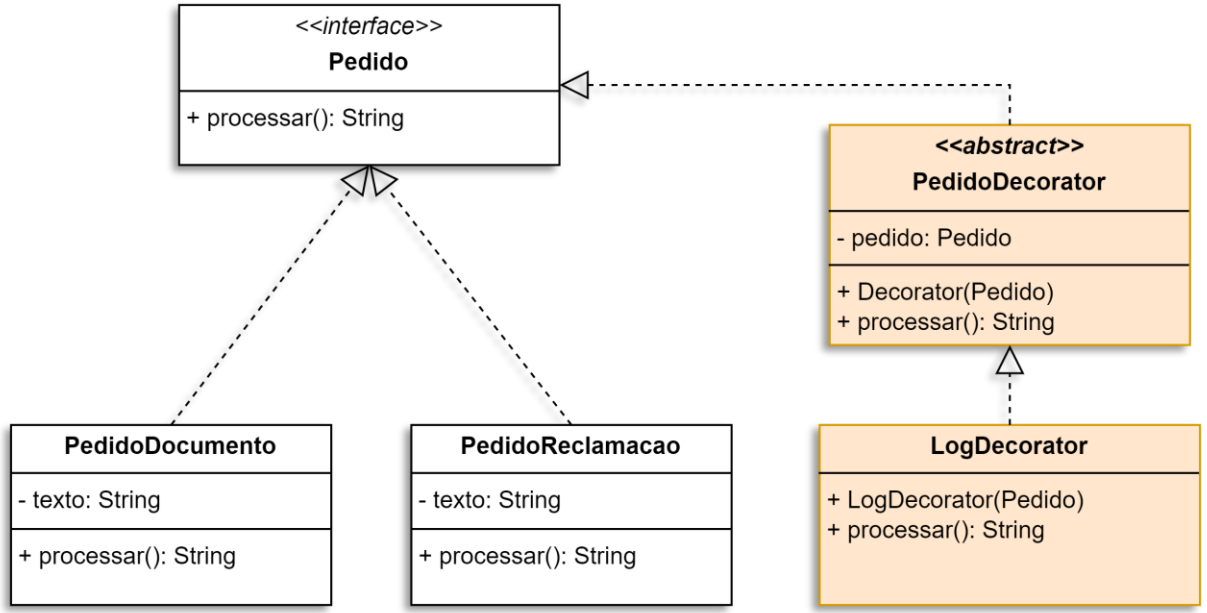
DECORATOR

Actividade: Log de pedidos (fácil)

Objetivo: Implementar um projeto usando o padrão Decorator para efetuar Log de pedidos

- O projeto tem **Pedido**s
  - texto: String
- O Decorator LogDecorator imprime o **Pedido**

NOTA: opcionalmente usar o Logger do java.



DECORATOR

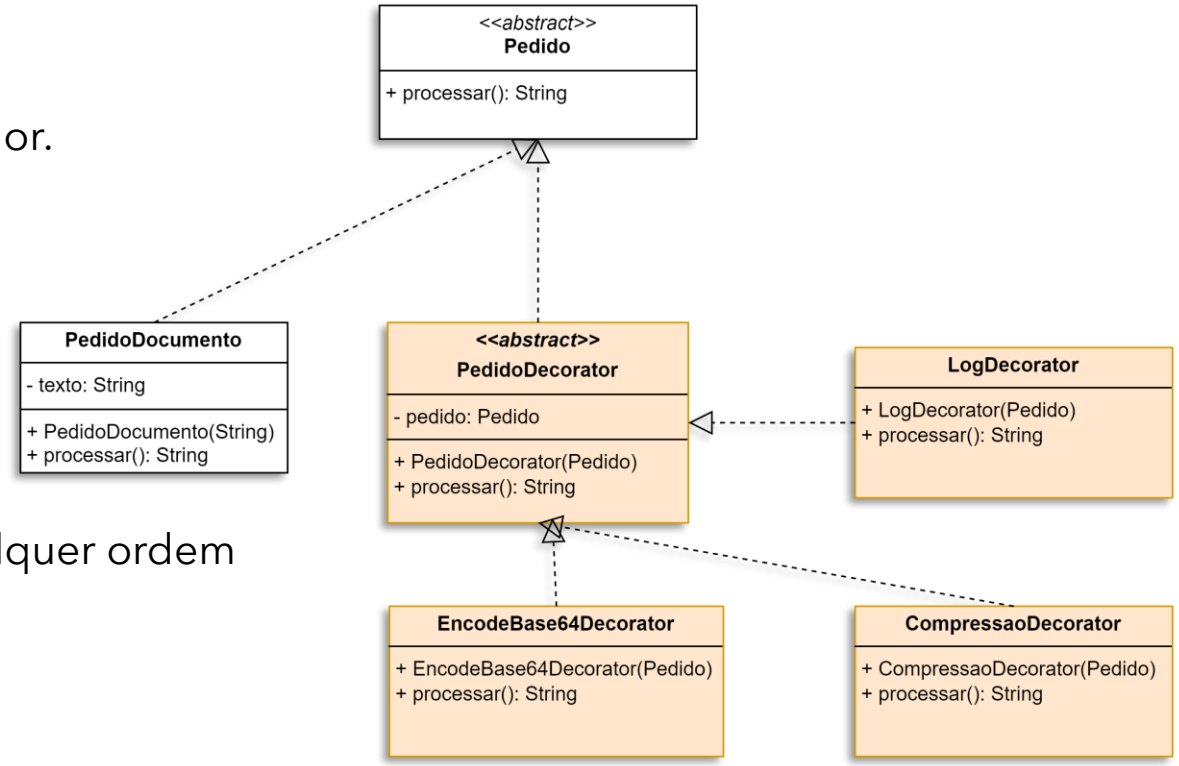
Actividade: Múltiplos decorators em pedidos (médio)

Objetivo: Implementar um projeto usando o padrão Decorator para efetuar Log, Encode em Base64 e compressão dos campos do Pedido

- Pedido: (texto: String) – o processar() devolve o valor.
- LogDecorator: imprime o resultado
- CompressaoDecorator: comprime o resultado
- EncodeBase64: codifica em base 64 o resultado

Os decorators devem poder ser encadeados em qualquer ordem

- DICA: log → encode → log → compressão → log

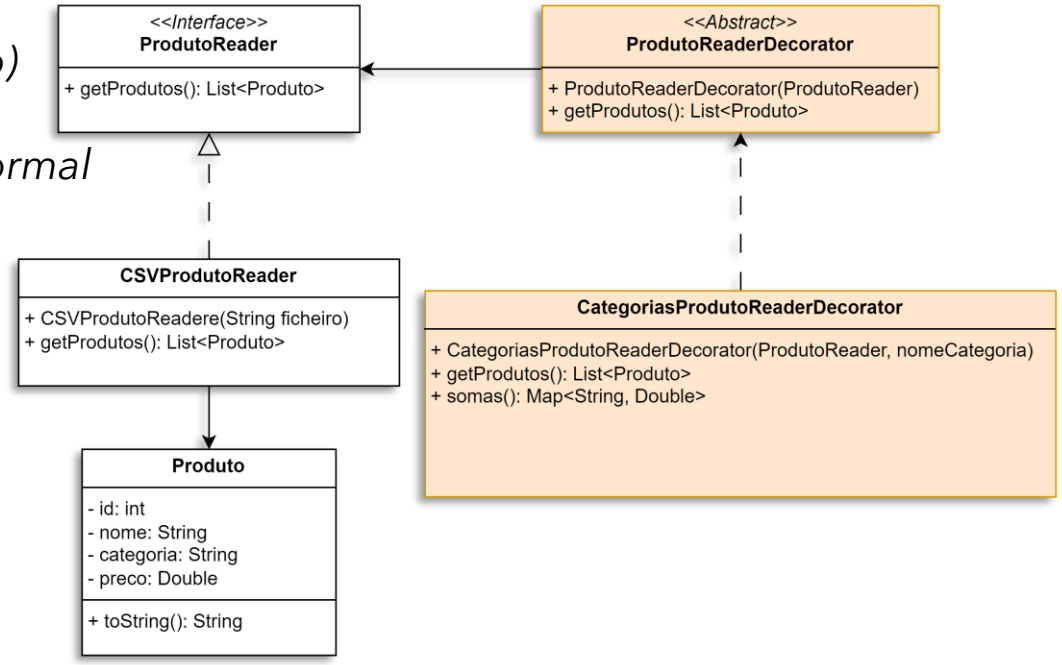


DECORATOR

Actividade: Ler ficheiros de produtos (médio)

Objetivo: Implementar uma solução com o padrão decorator, que permita ler um ficheiro de produtos

- usar o ficheiro **produtos.csv** *(id, nome, categoria, preco)*
- *A classe CSVProdutoReader le os produtos de formal normal*
- *O decorator CategoriasProdutoReaderDecorator*
  - *Le os produtos de uma categoria*
  - *Permite devolver um Map<String, Double>*
    - *As somas dos preços por categoria*



NO JDK

```java
// java.io
try (BufferedReader reader = new BufferedReader(new
FileReader("exemplo.txt"))) {
    String line;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

```java
// java.util.*
List<String> list = new ArrayList<>();

List<String> synchronizedList = Collections.synchronizedList(list);
List<String> unmodifiableList = Collections.unmodifiableList(list);
```

```java
// java.nio.channels.Channels
try (InputStream inputStream =
Channels.newInputStream(Channels.newChannel(System.in))) {
    int data = inputStream.read();
    while (data != -1) {
        System.out.print((char) data);
        data = inputStream.read();
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

ANTI PATTERNS

Excesso de camadas: adicionar muitos decoradores pode tornar o código difícil de entender e depurar, criando uma cadeia de chamadas complexa e difícil de seguir.

Abstração com fugas: quando os detalhes internos dos decoradores vazam para os clientes, quebrando o encapsulamento e aumentando o acoplamento entre componentes.

Estado inconsistente: decoradores que mantêm estado próprio podem levar a inconsistências, especialmente se múltiplos decoradores modificarem o estado de maneiras conflitantes.

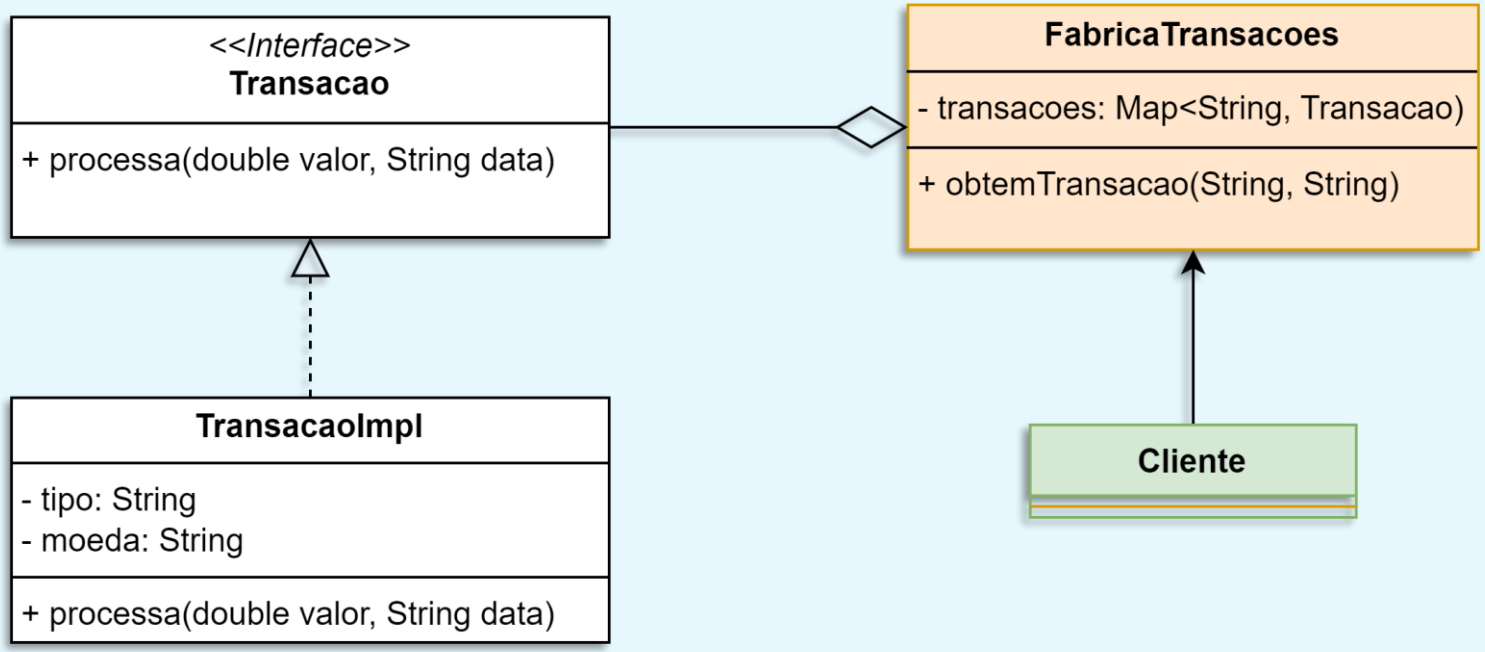
Abuso: utilizar decoradores para adicionar funcionalidades que poderiam ser implementadas de maneira mais simples e direta pode complicar desnecessariamente o design.

Performance: cada decorador adiciona uma camada extra de chamada de método, o que pode impactar negativamente o desempenho se muitos decoradores forem usados.

Violação do Single Responsibility Principle: decoradores que adicionam múltiplas responsabilidades podem violar o princípio da responsabilidade única, tornando o código mais difícil de manter e entender.

FLYWEIGHT

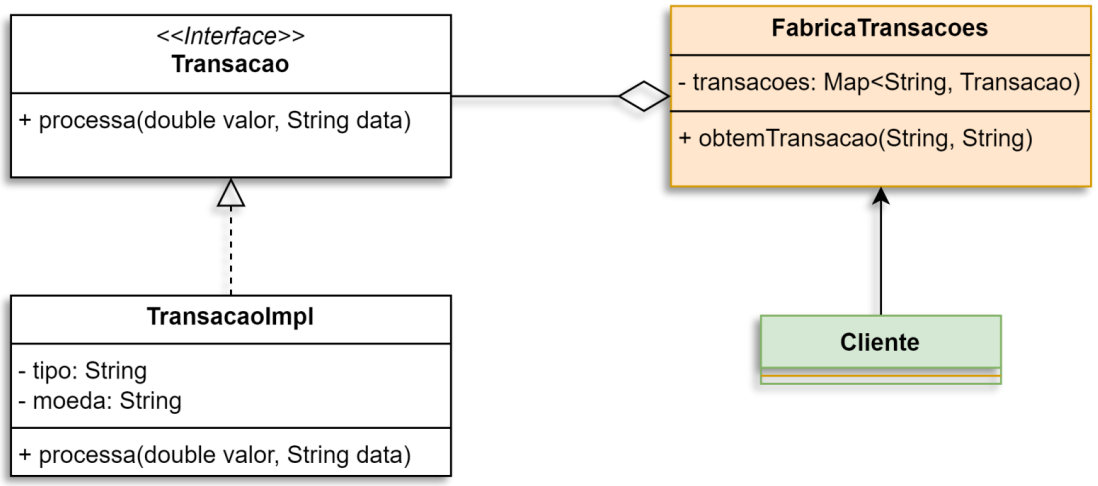
O padrão Flyweight é um padrão de design estrutural que visa minimizar o uso de memória e melhorar o desempenho ao compartilhar o máximo possível de dados entre objetos semelhantes.



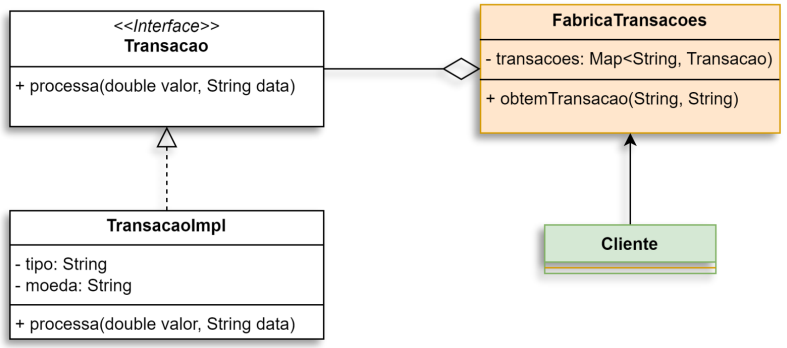
FLYWEIGHT

```java
public class FabricaTransacoes {
    private static final Map<String, Transacao> transacoes = new HashMap<>();

    public static Transacao obtemTransacao(String tipo, String moeda) {
        String chave = tipo + "-" + moeda;
        if (!transacoes.containsKey(chave)) {
            transacoes.put(chave, new TransacaoImpl(tipo, moeda));
        }
        return transacoes.get(chave);
    }
}
```

```java
public static void main(String[] args) {
    Transacao compra = FabricaTransacoes.obtemTransacao("compra", "dolar");
    compra.processa(100.0, "2023-05-01");

    Transacao venda = FabricaTransacoes.obtemTransacao("venda", "real");
    venda.processa(500.0, "2023-05-02");
}
```



FLYWEIGHT



Componentes

- **Flyweight interface**: operações associados aos objetos a usar
- **Concrete Flyweight**: a implementação da interface.
- **Flyweight Factory**: gere uma pool de objetos.

- NOTA: É recomendado que sejam imutáveis.

Benefícios

- **Memória**: redução do uso, pois beneficia de uma pool.
- **Desempenho**: redução de instanciações e configurações.
- **Cache de objetos**: os benefícios normais de uma cache.
- **Facilidade**: o Flyweight Factory facilita o pedido de objetos.
- **Escalabilidade**: a necessidade de um grande número de objetos pode beneficiar deste padrão.

Desafios/Desvantagens

- **Manutenção de código**: se existirem muitos grupos de objetos nas pools, pode-se tornar complicado intervir no código.
- **Limitações**: se os objetos tiverem muitos estados possiveis, o padrão pode prejudicar mais do que ajudar.
- **Sincronização**: em ambientes multi-thread pode constituir um desafio a implementação da Factory.

Casos de uso

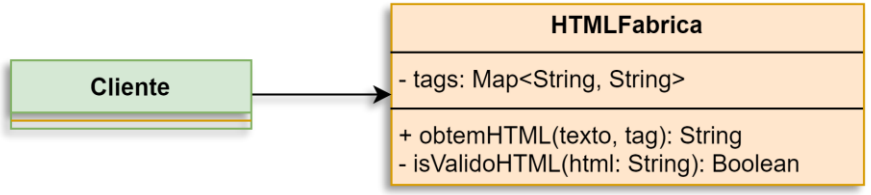
- **Aplicações gráficas e jogos**: há muitos objetos semelhantes
- **GUIs:** botões, caixas, linhas, icons, são muito semelhantes
- **Em IDEs ou editors de documentos:** dados de formatação associados a objetos
- **Relatórios e gráficos**: dados comuns em relatórios, headers, footers, formatações, snippets de texto, etc.
- **Redes e protocolos:** muitos campos em protocolos mudam pouco.

FLYWEIGHT

Actividade: HTML tags para texto (fácil)

Objetivo: Usar o padrão Flyweight para, dada uma string e um nome de uma tag HTML, devolve a tag HTML para a string pretendida.

- Ex: *String obtemHtml(String txt, String nomeTag)*
  - **"ola", "strong"** → ola
  - A ideia está em guardar o *String.format("**%s**")*
- O uso do *String.format* ao invés de toda a String optimiza a performance.
- Dependendo do uso, pode-se dar o caso de ao invés do "string.*format*" usar toda a string
  - bom para tags mais genéricas como H1, H2, etc, Title, header, style, e semelhantes.
- Pode-se combinar as soluções
- OPCIONAL: Validar se o html é válido: procurar a biblioteca: **JSoup**

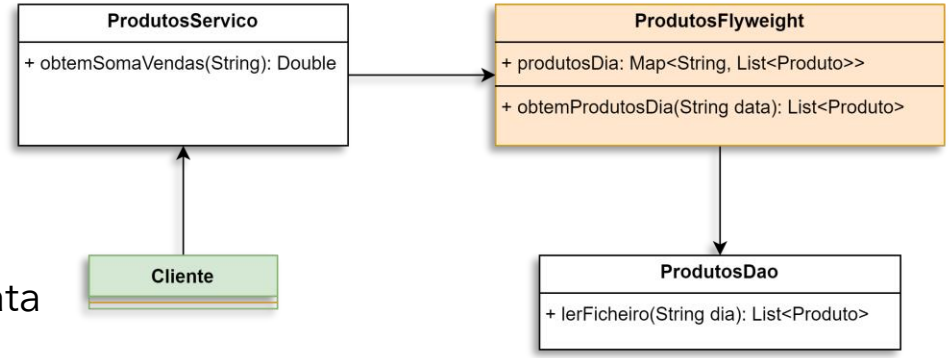



```
<dependency>
  <groupId>org.jsoup</groupId>
  <artifactId>jsoup</artifactId>
  <version>1.15.3</version>
</dependency>
```

FLYWEIGHT

Actividade: Sistema de cache de ficheiros (médio)

Objetivo: Dados 7 ficheiros, pode ser pedido aleatoriamente o conteúdo de qualquer um deles. Usar o flyweight para evitar constantes leituras dos ficheiros.



- Os ficheiros estão na pasta resources
- Os ficheiros contêm produtos com preços
- É necessário calcular a soma dos preços dos produtos numa data
- A classe "Flyweight" contem o nome dos ficheiros com o conteúdo.
    - ex: produtos-2024-01-01.csv (sao sete)
- A solução pode ir guardando os ficheiros em memória ou alternativamente, visto que a ideia e a soma dos preços, guardar a data e o total, após o primeiro cálculo! Qualquer uma das soluções é possível.

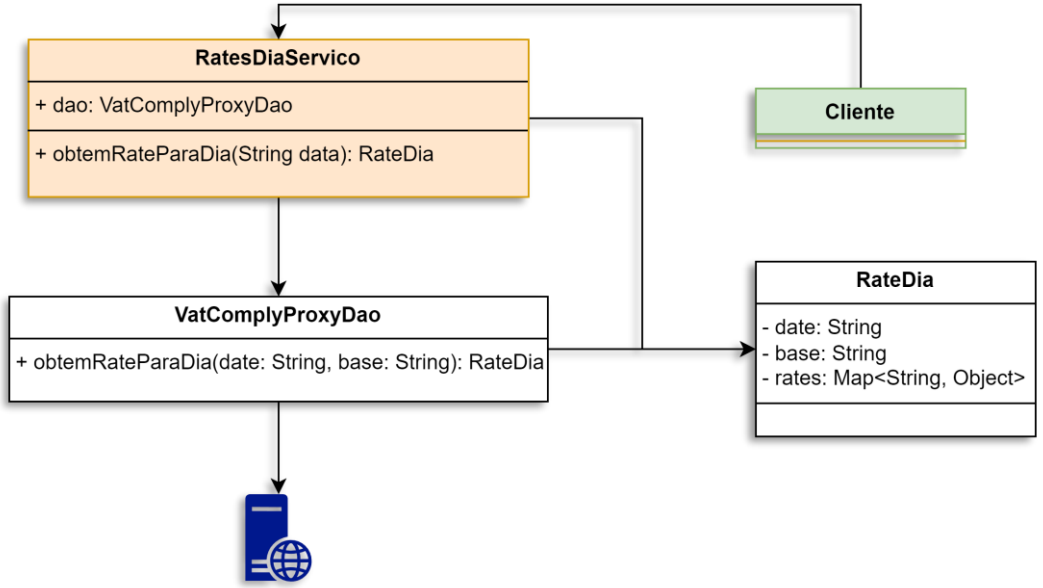
FLYWEIGHT

Actividade: VATComply API – rates por datas (avançado)

Objetivo: Usar o padrão flyweight para chamar uma das APIs existentes no site https://www.vatcomply.com/documentation, relativamente a rates por datas.

- A classe Flyweight deverá gerir instâncias por MOEDA e Data
  - Ex: "2024-01-01:EUR" **date:base**
  - https://api.vatcomply.com/rates?**date**=2000-04-05&**base**=EUR


```
{
    "date": "2000-04-05",
    "base": "EUR",
    "rates": {
        "EUR": 1,
        "USD": 0.9673,
        "JPY": 101.71,
        "CYP": 0.57521,
        "CZK": 36.283,
        "DKK": 7.4483,
        "EEK": 15.6466,
        "GBP": 0.6088,
        "HUF": 258.91,
        "LTL": 3.8665,
```

NO JDK

```java
// java.lang.String
String s1 = "Hello";
String s2 = "Hello";

// Verifica se s1 e s2 referenciam o mesmo objeto
System.out.println(s1 == s2); // true
```

```java
// java.lang.Integer (-128 a 127)
Integer i1 = Integer.valueOf(127);
Integer i2 = Integer.valueOf(127);

// Verifica se i1 e i2 referenciam o mesmo objeto
System.out.println(i1 == i2); // true

Integer i3 = Integer.valueOf(128);
Integer i4 = Integer.valueOf(128);

// Verifica se i3 e i4 referenciam o mesmo objeto
System.out.println(i3 == i4); // false
```

```java
// java.util.Locale
Locale locale1 = Locale.getDefault();
Locale locale2 = Locale.getDefault();

// Verifica se locale1 e locale2 referenciam o mesmo objeto
System.out.println(locale1 == locale2); // true
```

ANTI PATTERNS

Optimização prematura: Implementar o padrão Flyweight antes de identificar um problema real de desempenho pode levar a um design desnecessariamente complexo.

Complexidada desnecessária: O uso excessivo do Flyweight pode tornar o código mais difícil de entender e manter, especialmente se a separação entre estado intrínseco e extrínseco não for clara.

Abstração com fugas: Quando os detalhes internos do Flyweight vazam para os clientes, quebrando o encapsulamento e aumentando o acoplamento entre componentes.

Gestão do estado: Gerenciar o estado extrínseco pode ser complicado e propenso a erros, especialmente em sistemas concorrentes onde múltiplos threads podem acessar e modificar o estado.

Demasiados objetos: A criação e o gerenciamento de um grande número de objetos Flyweight podem introduzir um overhead significativo, especialmente se a fábrica de Flyweights não for implementada de maneira eficiente

RESUMO

OS PADRÕES ESTRUTURAIS FOCAM-SE NA COMPOSIÇÃO DE CLASSES E OBJETOS PARA FORMAR ESTRUTURAS MAIORES E MAIS COMPLEXAS.

COM ESTAS ESTRUTURAS PODEM-SE CRIAR COMPOSIÇÕES OU ADICIONAR COMPORTAMENTOS A OBJETOS, SEM ALTERAR O SEU CÓDIGO ORIGINAL.

RUMOR:O DECORATOR É MUITO USADO EM CÓDIGO LEGACY, A FACADE EM APIS, O PROXY MUITO POPULAR.

BEHAVIORAL PATTERNS

INTRODUÇÃO

PADRÕES COMPORTAMENTO

CHAIN OF RESPONSIBILITY

COMMAND

INTERPRETER

ITERATOR

MEDIATOR

MEMENTO

OBSERVER

STATE

STRATEGY

TEMPLATE METHOD

VISITOR

RESUMO

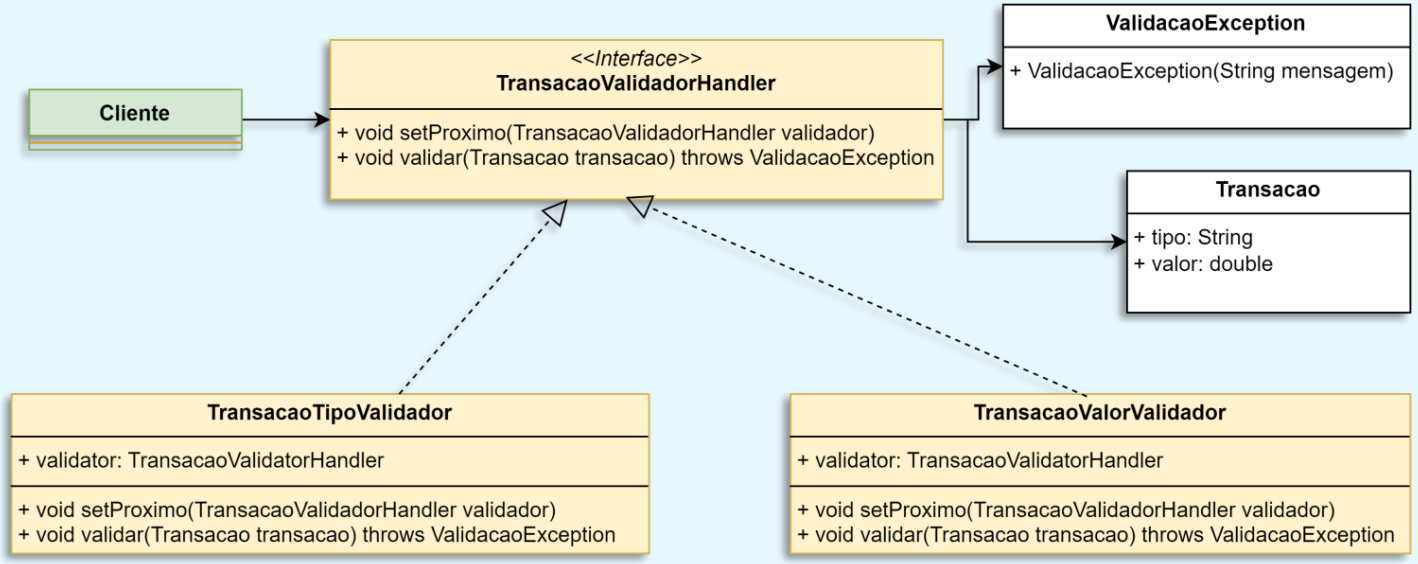
PADRÕES COMPORTAMENTAIS

Os padrões de comportamento são soluções recorrentes para problemas comuns relacionados com a interação e comunicação entre objetos. Ajudam a definir como os objetos colaboram e distribuem responsabilidades, promovendo a flexibilidade e a reutilização do código

- **Comunicação e interação:** definem como os objetos comunicam entre si
- **Distribuição de responsabilidades**: as responsabilidades de cada objeto ficam claras e organizadas
- **Reutilização de código**: padrões facilitam o desenvolvimento modular melhorando a gestão do código.
- **Interoperabilidade**:permitem adição de novo código com poucas refatorizações.

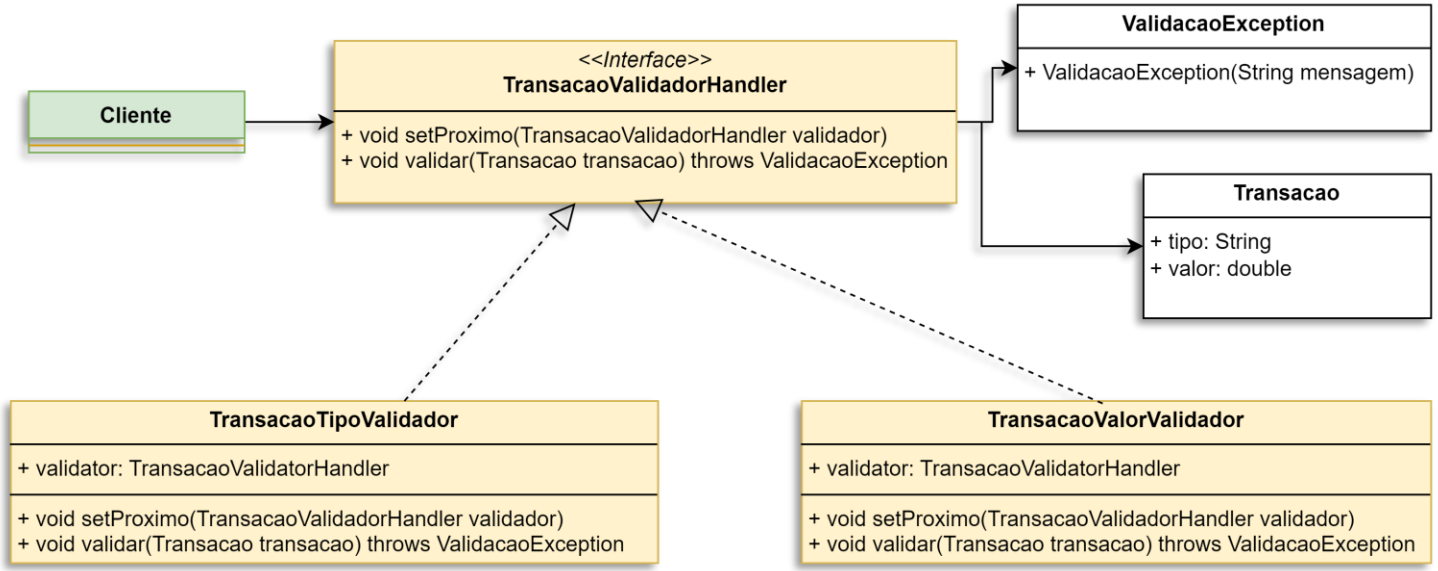
CHAIN OF RESPONSIBILITY

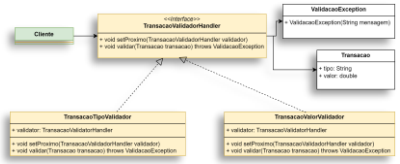
O padrão Chain of Responsibility é um padrão comportamental que permite que um pedido seja passado por uma cadeia de objetos receptores até que um deles o trate.



CHAIN OF RESPONSIBILITY

```java
public static void main(String[] args) {
    Transacao transacao = new Transacao("bancaria", 1200.00);

    TransacaoValidadorHandler validaTipo = new TransacaoTipoValidador();
    TransacaoValidadorHandler validaValor = new TransacaoValorValidador();

    validadaTipo.setProximo(validaValor);

    // a implementação do validador chama o próximo
    validaTipo.validar(transacao);
}
```





CHAIN OF RESPONSIBILITY

Componentes

- **Handler**: A interface com os métodos para o processamento.
- **Implementação do Handler**: A implementação do Handler.
- **Cliente**: quem efectua o pedido

Benefícios

- **Flexibilidade**: permite a adição de handlers de forma simples
- **Single Responsibility Principle**: Cada Handler tem uma responsabilidade específica
- **Simplicidade**: Cada Handler contém a aua propria lógica, não precisa de saber nada acerda dos outros Handlers.

Desvantagens/Desafios

- **Debug e troubleshooting**: Se o número de handlers crescer pode-se tornar complicado fazer o debug, principalmente se eles tiverem uma lógica muito divergente entre eles.
- **Complexidade**: em hierarquias pode ser díficil a gestão do código.
- **Desempenho**: Cada handler tem a sua lógica, seja ela de acesso a memória interna ou externa. Isto pode ter um impacto, por vezes escondido, no desempenho.
- **Planeamento**: embora uma das vantagens dos padrões é a adaptação, o padrão chain of responsibility, em alguns casos pode exigir planeamento, afim de garantir que determinados handlers sejam sempre usados.

Casos de uso

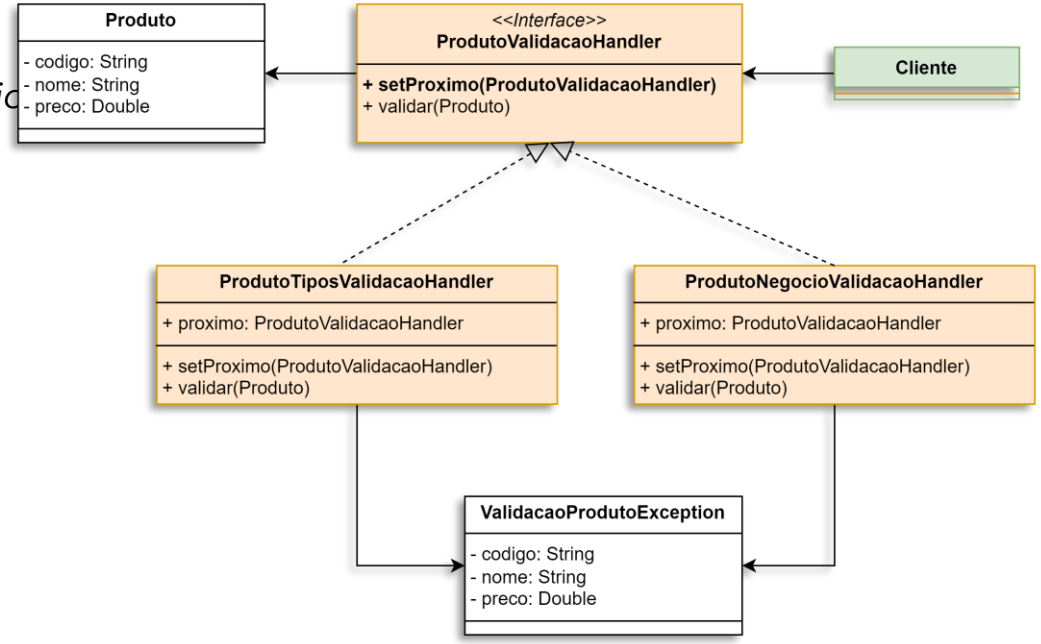
- **Sistemas de log:** os níveis de log, em sequência granular (trace, debug, ...)
- **Autenticação e Autorização:** a autenticação em alguns cenários usa eventos, em etapas, e conforme o que se recebe, o handler associado toma conta do pedido.
- **Processamento de pedidos em servidores:** pedidos em http é usual usarem uma sequência de handlers no processamento de pedidos, afim de o transformarem, validarem, auditarem, etc.
- **Manipulação de Eventos:** eventos em interfaces gráfica costumam usar uma hierarquia de handlers. Ex: um click pode ser trado directamente por uma área, ou uma imagem dentro dessa area, e assim sucessivamente.
- **Validação de dados:** regras de validação de dados pode estar em sequência conforma o detalhe pretendido.

CHAIN OF RESPONSIBILITY

Actividade: níveis de validações de campos (fácil)

Objetivo: usando o padrão chain of responsibility, implemente uma solução que contém dois níveis de validações aos campos de um Produto

- No primeiro passo (handler) de validações, testa-se por **null** e *vazio*
- No segundo passo (handler), valida-se regras de negócio como:
  - os valores são maiores que zero
  - os valores são menores que 1000, etc.
- Implementar a validação de um **Produto**
  - campos: *id, nome, preço*
  - o preço é sempre **>= 0** e **<= 1000**
- A implementação pode lançar uma **ValidacaoException** quando falha.



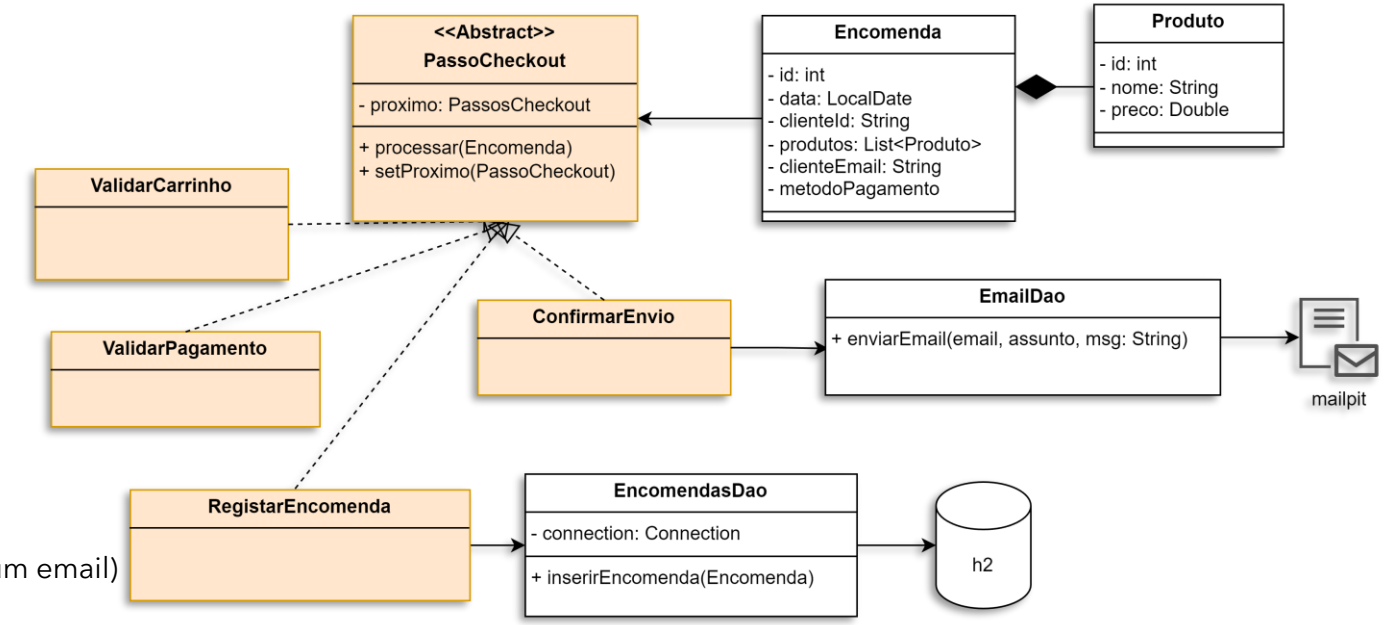
CHAIN OF RESPONSIBILITY

Actividade: checkout (médio-avançado)

Objetivo: usando o padrão chain of responsibility, implemente um sistema simples que simula os passos de um checkout

Fases do checkout:

- **ValidacaoCarrinho**:
  - tem pelo menos 1 produto
- **ValidaPagamento**: um método de pagamento foi escolhido
  - um atributo com uma String com um dos seguintes valores:
    - "visa", "mastercard", "paypal"
- **RegistaEncomenda** numa tabela H2
  - *Encomendas (id, data, clienteId)*
  - *EncomandasItems (id, nome, preco, encomendaId)*
- **Confirma** envio (imprime o objeto como se estivesse a enviar um email)
  - opcionalmente pode enviar email usando: **mailpit**

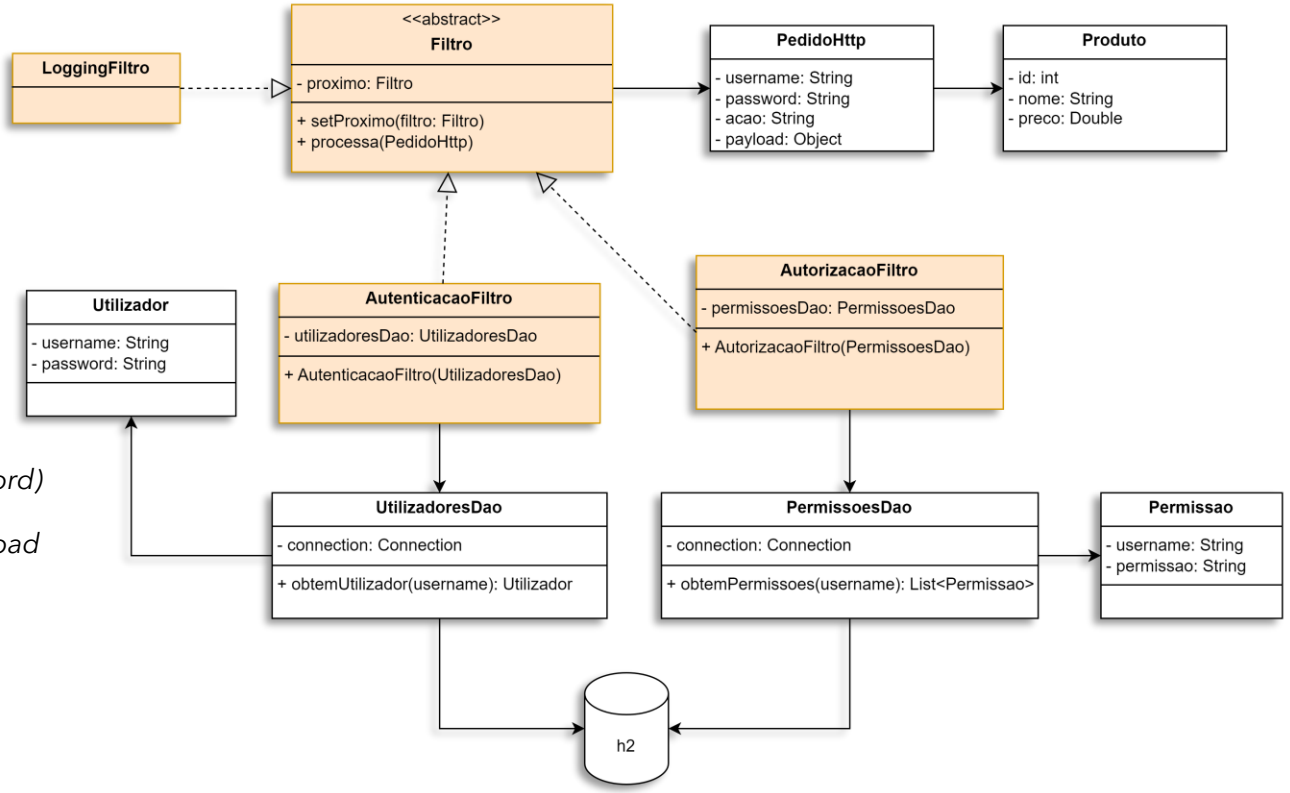


CHAIN OF RESPONSIBILITY

Actividade: simula pedido internet com filtros (avançado)

Objetivo: usando o padrão chain of responsibility, implemente um sistema idêntico aos filtros num pedido HTTP.

- Um classe **PedidoHttp** contém os campos:
  - *username, password, acao, payload (um objeto)*
- Uma aplicação cria um **PedidoHttp** com um **Produto**
  - *id, nome e preco*
- Os seguintes filtros devem ser criados
  - **autenticação**: tabela em H2 com *utilizadores (username, password)*
  - **logging**: Logger do java para escrever o *username*, *acao* e *payload*
  - **autorização**: tabela *permissões (username, permissao)*
    - inserir o *admin:"inserir"*, todos os outros so *"ler"*
- A ordem dos filtros: logging → autenticação → autorizacao



NO JDK

```java
// java.servlet.Filter
public class AuthenticationFilter implements Filter {
    public void doFilter(ServletRequest request,
                         ServletResponse response,
                         FilterChain chain)
            throws IOException, ServletException {
        // Lógica de autenticação
        // Passa para o próximo filtro
        chain.doFilter(request, response);     }
}
```

```java
// Eventos
public class MyButton extends Button {
    public boolean handleEvent(Event evt) {
        if (evt.id == Event.ACTION_EVENT) {
            // Lógica de tratamento do evento
            return true;
        }
        // Passa para o próximo manipulador
        return super.handleEvent(evt);
    }
}
```

```java
// java.util.logging.Logger
Logger logger = Logger.getLogger(MyClass.class.getName());
logger.setLevel(Level.INFO);
logger.addHandler(new ConsoleHandler());
logger.addHandler(new FileHandler("mylog.txt"));

logger.info("mensagem");
```

ANTI PATTERNS

Complexidade desnecessária: Implementar uma cadeia de responsabilidade quando uma simples estrutura condicional (como `if-else`) seria suficiente pode adicionar complexidade desnecessária ao código.

Número de elos incontrolavel: Permitir que a cadeia de responsabilidade cresça sem controle pode resultar em uma longa cadeia de handlers, tornando difícil rastrear e depurar o fluxo de processamento.

Tight Coupling: Se os handlers na cadeia dependem fortemente uns dos outros, isso pode levar a um acoplamento apertado, contrariando o objetivo de promover um design flexível e desacoplado.

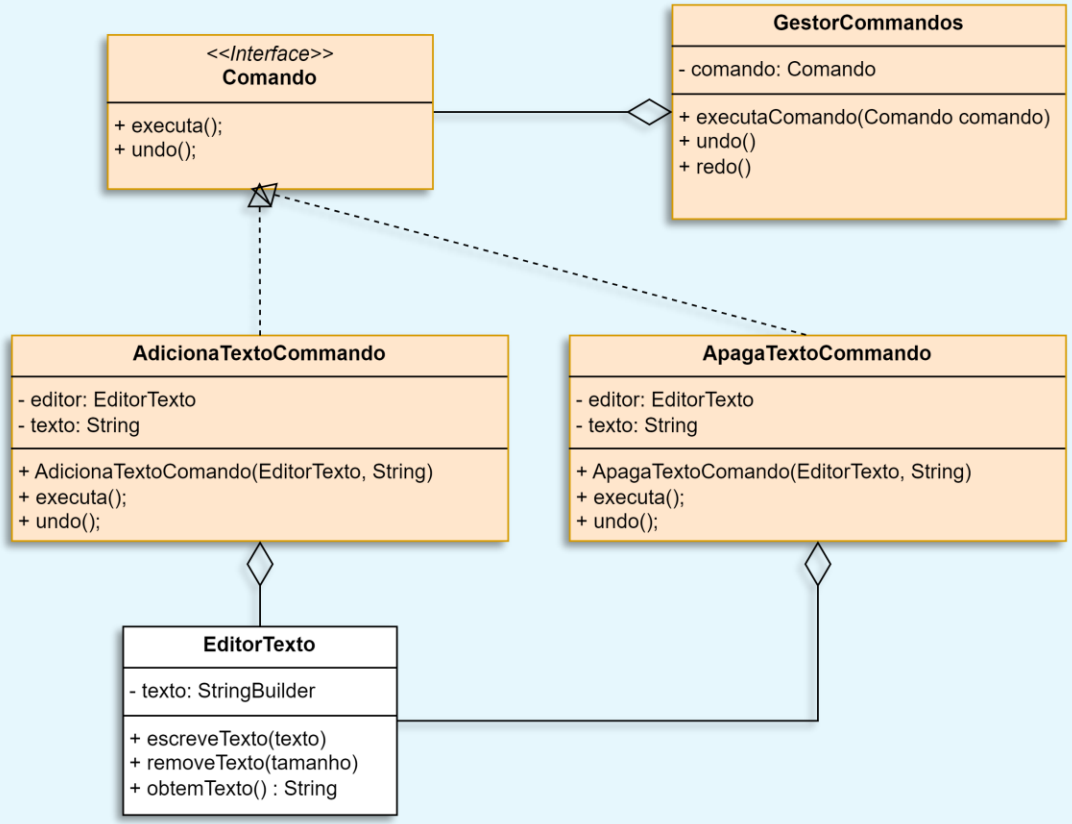
Termino indefenido: Não garantir que a cadeia de responsabilidade tenha um ponto de terminação claro pode resultar em loops infinitos ou em handlers que nunca processam a solicitação.

Handling inconsistente: Quando diferentes handlers na cadeia processam a mesma solicitação de maneiras inconsistentes, isso pode levar a comportamentos imprevisíveis e bugs difíceis de identificar.

Performance: Passar uma solicitação por uma longa cadeia de handlers pode introduzir um overhead de desempenho significativo, especialmente se cada handler realizar operações complexas.

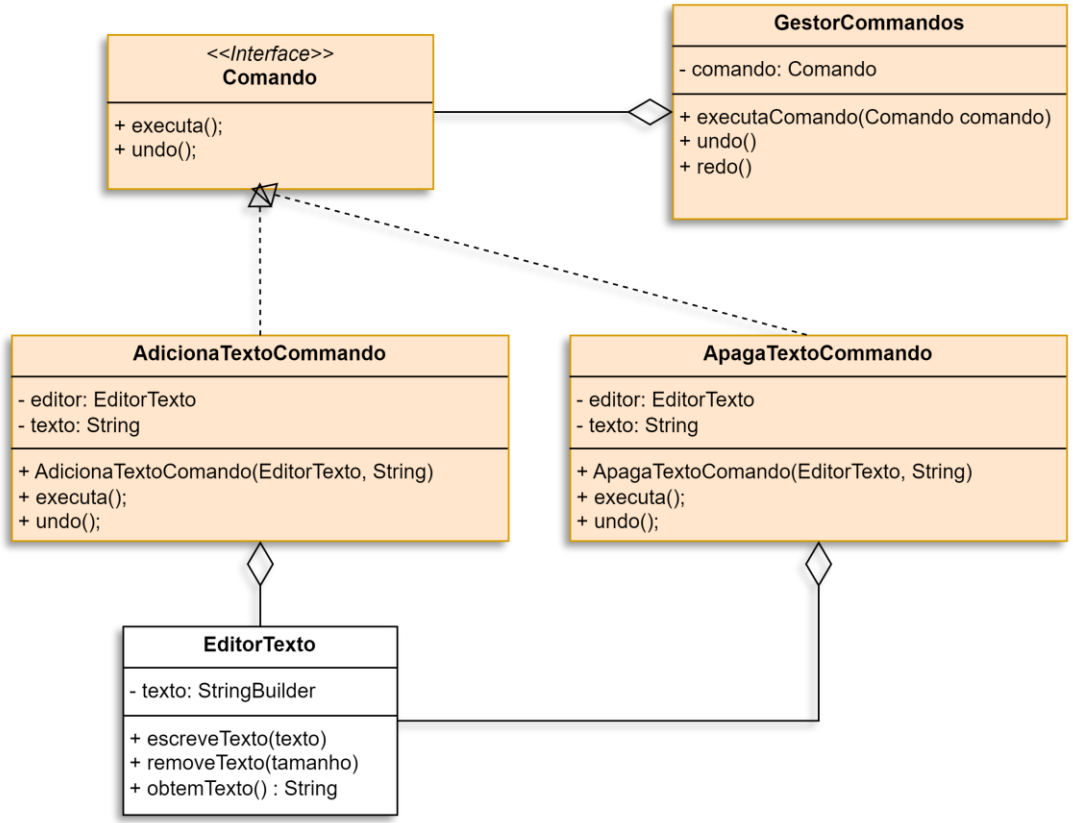
COMMAND

É um padrão comportamental que transforma um pedido em um objeto independente que contém toda a informação necessária para executar a ação.

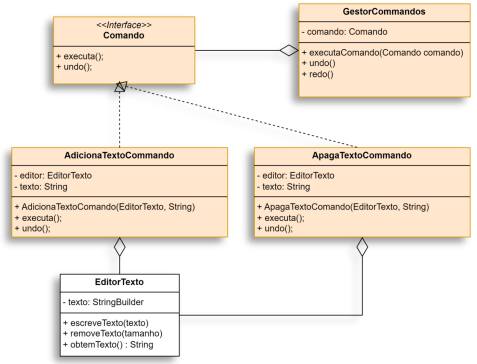


COMMAND

```java
public static void main(String[] args) {

    EditorTexto editor = new EditorTexto();

    Commando adicionaTexto = new AdicionaTextoComando(editor, "Ola!");
    Commando adicionaTexto2 = new AdicionaTextoComando(editor, "Bem vindo ao curso de padrõeses!");
    Commando apagaTexto = new ApagarTextoCommando(editor, "es!");
    Commando adicionaTexto3 = new AdicionaTextoComando(editor, "!");

    GestorComandos invoca = new GestorComandos();

    invoca.setCommando(adicionaTexto);
    invoca.executa();

    System.out.println("-".repeat(40));
    System.out.println(editor.obtemTexto());

    invoca.setCommando(adicionaTexto2);
    invoca.executa();
    System.out.println("-".repeat(40));
    System.out.println(editor.obtemTexto());

    invoca.setCommando(apagaTexto);
    invoca.executa();
```



COMMAND



Componentes

- **Command**: interface que declara o método que permite executar o comando.
- **Implementação do Command**: a implementação da lógica associada ao comando e define a ligação entre o Receiver e a ação.
- **Receiver**: a classe que realiza a ação real quando o comando é executado
- **Invoker**: a classe que pede a execução do comando.
- **Client**: quem configura e inicia os commndos.

Benefícios

- **Desacoplamento**: o objeto que pede a ação fica separado do objeto que a executa
- **Reversibilidade**: Facilita operações de *undo* e *redo*
- **Extensibilidade**: facilidade de adição de novos comandos
- **Flexibilidade**: permite a criação de filas de comandos

Desvantagens/Desafios

- **Complexidade**: como todos os padrões, há sempre adição de código não funcional para alavancar as vantagens
- **Sobrecarga**: mais objetos no código
- **Manutenção**: exige conhecimento sobre padrões, tornando mais exigente a manutenção do código
- **Desempenho**: em sistemas com muitos comandos, dá-se muito redireção.

Casos de uso

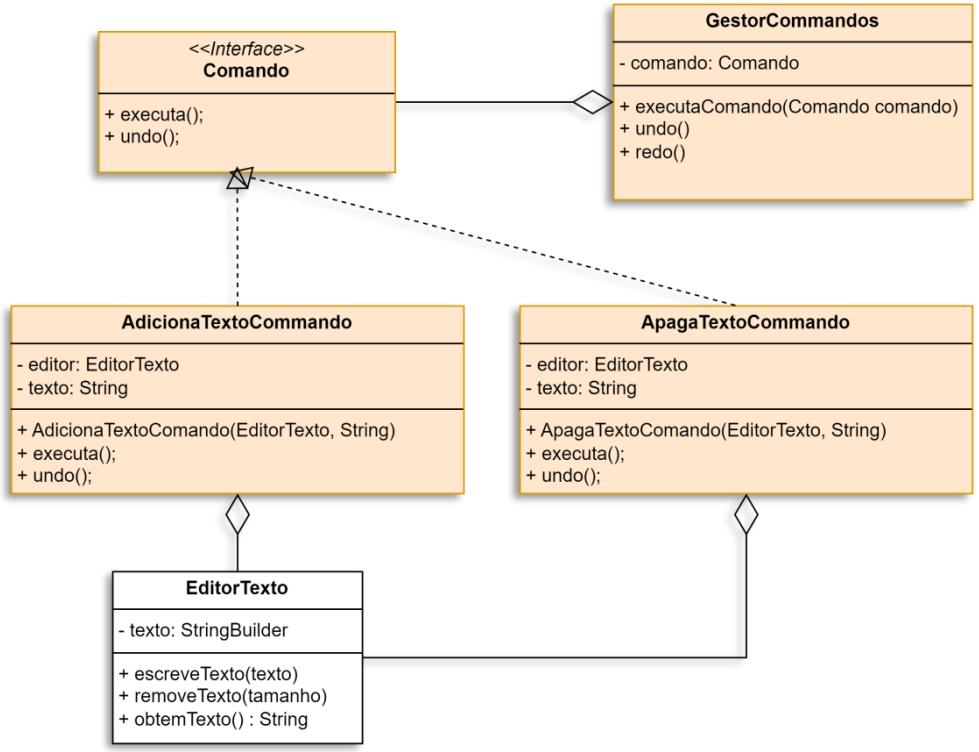
- **Sistemas redo/undo**: uso de pilhas para permitir os típicos undo/redo
- **Filas de processamento:** quando sequências de commandos são necessárias, tipicamente para execuções assincronas.
- **Menus e botões em GUIs:** ações de menus e botões podem ser encapsuladas em Command.
- **Macros:** as macros são tipicamente sequencies de commandos.
- **Frameworks web:** junto com o FrontController para associar pedidos HTTP a ações no servidor.

COMMAND

Actividade: Editor de Texto (fácil)

Objetivo: usando o padrão Command, escreva um pequeno editor de textos (com strings ☺)

- Dois comandos:
  - AdicionaTextoComando: acrescenta texto
  - ApagaTextoComando: remove texto (do fim)
- O **Comando** deverá ter dois métodos: *executa()*, e *undo()*
  - *O undo() (anula a operação anterior) executado logo a seguir*
    - *existe outro padrão para este tipo de situações*
- O **EditorTexto** permite
  - adicionar texto e
  - apagar

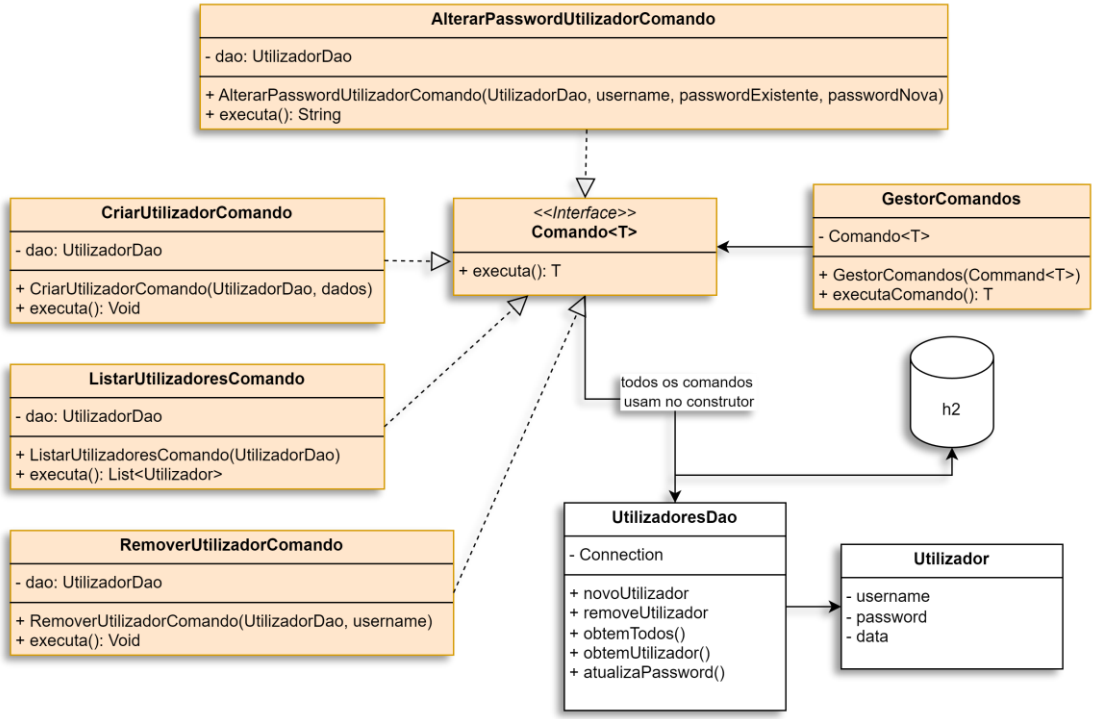


COMMAND

Actividade: Gestor utilizadores, CLI (médio)

Objetivo: usando o padrão Comando, escreva uma aplicação de Command Line que aceita pedidos para utilizadores. Ex: ***> GestorUtilizadores criar admin,password,2024-01-01***

- As ações possíveis são:
  - **criar**: cria um utilizador
    - A password deverá ser transformada usando MD5
  - **remover**: remove um utilizador pelo *username*
  - **listar**: lista os utilizadores existentes
  - **alterar-password**: altera a password
    - O utilizador tem de inserir a password atual e a nova
    - Ex: *GestorUtilizadores alterar-password admin password pass2*
- Base de dados H2, com uma tabela
  - *utilizadores (username, password, data)*
- NOTA: cuidado com os espaços na linha de comando!
- DICA: como é uma operação CLI convém criar um utilizador inicial



COMMAND

Actividade: Gestor de utilizadores com reflection (avançado)

Objetivo: Crie de raíz ou adapte o exercício anterior (Gestor utilizadores) para configurar num ficheiro de propriedades qual a classe Comando a ser executada

- Ficheiro: ***comandos.properties*** *(/resources)*
- Ex: "**comando.*criar=com.josedacruz.patterns.comandos.CriarComando***"
- A estrutura das classes é a mesma (do exercício anterior), no entanto:
  - O gestor de comandos deverá usar Reflection para carregar a classe do comando, inicializar com o **UtilizadoresDao** e os argumentos da aplicação, ou seja, todos os Comandos usam o mesmo construtor.
  - A interface Comando deverá ser uma classe abstracta.

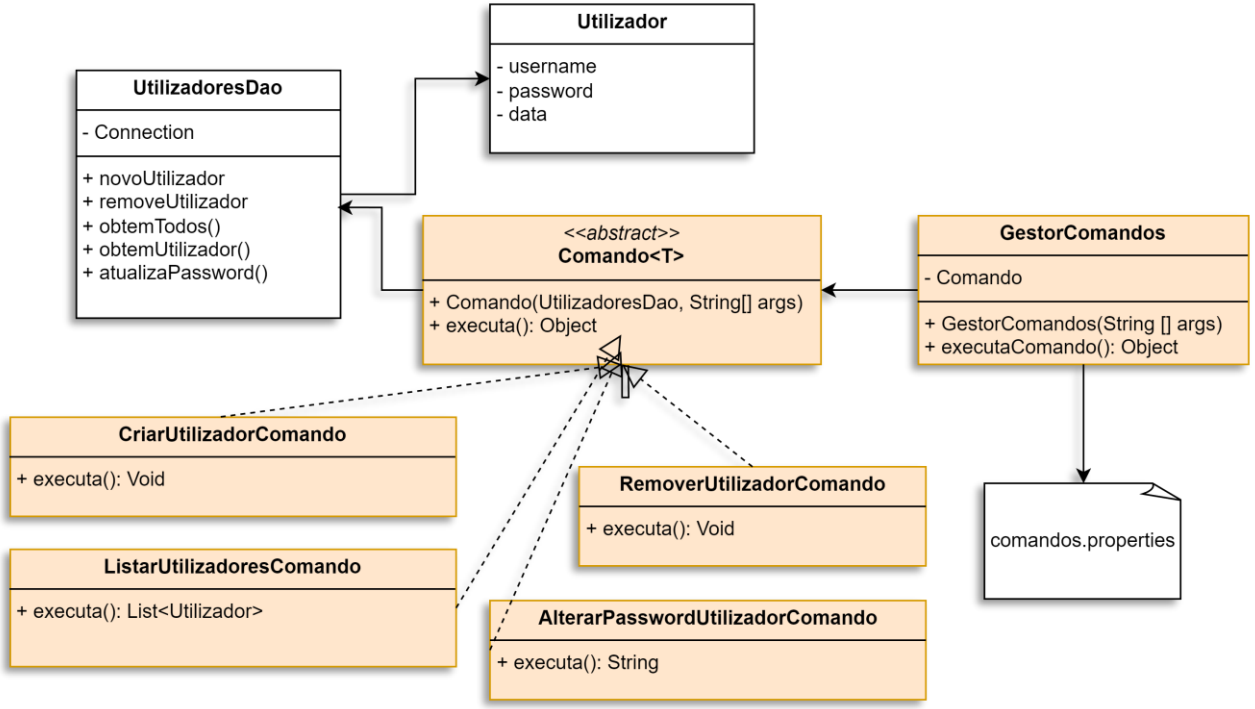
```
comando.criar=com.josedacruz.patterns.comandos.CriarUtilizadorComando
comando.alterar-password=com.josedacruz.patterns.comandos.AlterarPasswordUtilizadorComando
```

```java
public abstract class  Comando {

    protected UtilizadoresDao utilizadoresDao;
    protected String[] parametros;
    protected Comando(UtilizadoresDao dao, String[] parametros)
{
        this.utilizadoresDao = dao;
        this.parametros = parametros;
    }

    public abstract Object executa();
}
```

```java
private Comando carregarComando(String[] args) {

    try {
        Properties parametros = ParametrosAplicacao.obtemParametros();
        String nomeComando = parametros.getProperty("comando." + args[0]);
        Class comandoClass = Class.forName(nomeComando);
        Constructor constructor =
comandoClass.getConstructors()[0]; Comando cmd = Comando)constructor.newInstance(
                new UtilizadoresDao(H2BaseDados.obtemConnection()), args);
        return cmd;
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

COMMAND

Actividade: Gestor de utilizadores com reflection (avançado)




```
comando.criar=com.josedacruz.patterns.comandos.CriarUtilizadorComando
comando.alterar-password=com.josedacruz.patterns.comandos.AlterarPasswordUtilizadorComando
```

```java
public abstract class  Comando {

    protected UtilizadoresDao utilizadoresDao;
    protected String[] parametros;
    protected Comando(UtilizadoresDao dao, String[] parametros)
{
        this.utilizadoresDao = dao;
        this.parametros = parametros;
    }

    public abstract Object executa();
}
```

```java
private Comando carregarComando(String[] args) {

    try {
        Properties parametros = ParametrosAplicacao.obtemParametros();
        String nomeComando = parametros.getProperty("comando." + args[0]);
        Class comandoClass = Class.forName(nomeComando);
        Constructor constructor = comandoClass.getConstructors()[0]; Comando cmd
= (Comando)constructor.newInstance(
                new UtilizadoresDao(H2BaseDados.obtemConnection()), args);
        return cmd;
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

NO JDK

```java
// Runnable
Runnable command = new Runnable() {
    @Override
    public void run() {
        System.out.println("Comando executado!");
    }
};
new Thread(command).start();
```

```java
// ActionListener - Swing
JButton button = new JButton("Clique aqui");
button.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Botão clicado!");
    }
});
```

```java
// Action - Swing
Action saveAction = new AbstractAction("Salvar") {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Ação de salvar executada!");
    }
};
JButton saveButton = new JButton(saveAction);
```

ANTI PATTERNS

Excessivamente complexos: Comandos que encapsulam muitas ações e lógica interna, tornando-os difíceis de entender, testar e reutilizar.

Muitos parâmetros: Comandos que exigem um grande número de parâmetros para sua execução, dificultando a sua utilização e aumentando a possibilidade de erros.

Excesso de lógica de negócio: Comandos que contêm grande parte da lógica de negócio da aplicação, violando o princípio da responsabilidade única.

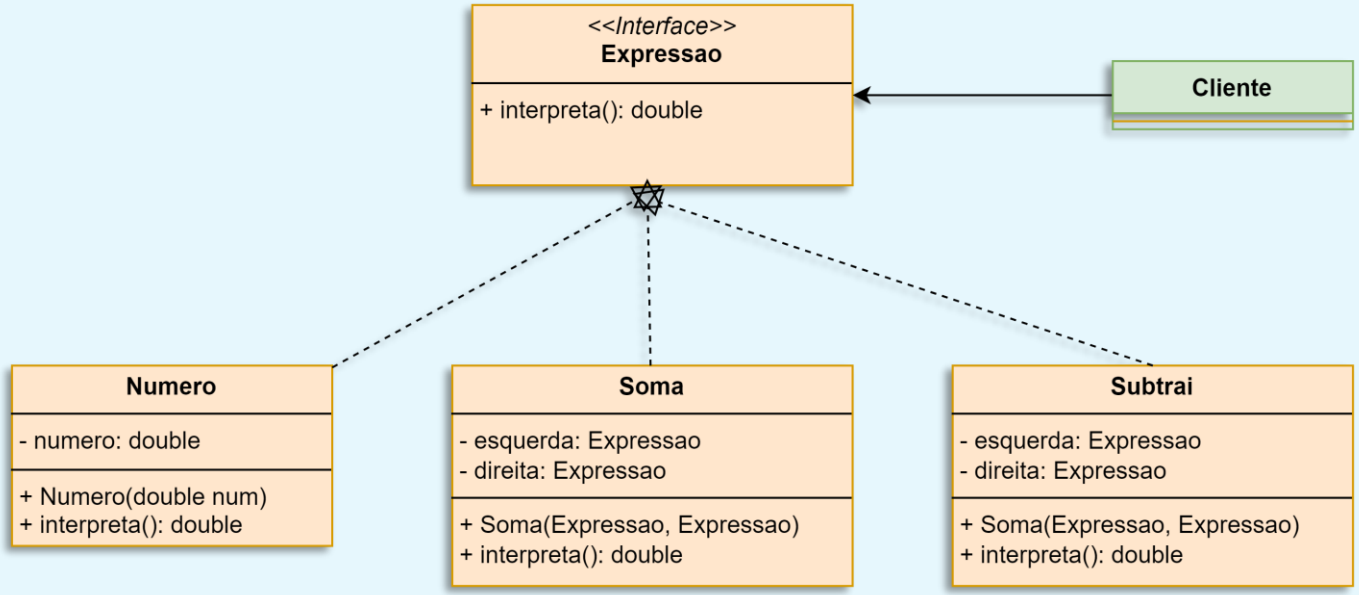
Efeitos colaterais inesperados: Comandos que modificam o estado de outros objetos de forma imprevista, dificultando a depuração e o rastreamento de erros.

Muita lógica condicional: Comandos que contêm muitas instruções condicionais, tornando-os complexos e difíceis de entender.

Demasiadas exceptions: Comandos que utilizam excessivamente exceções para controlar o fluxo de execução, dificultando a leitura do código.

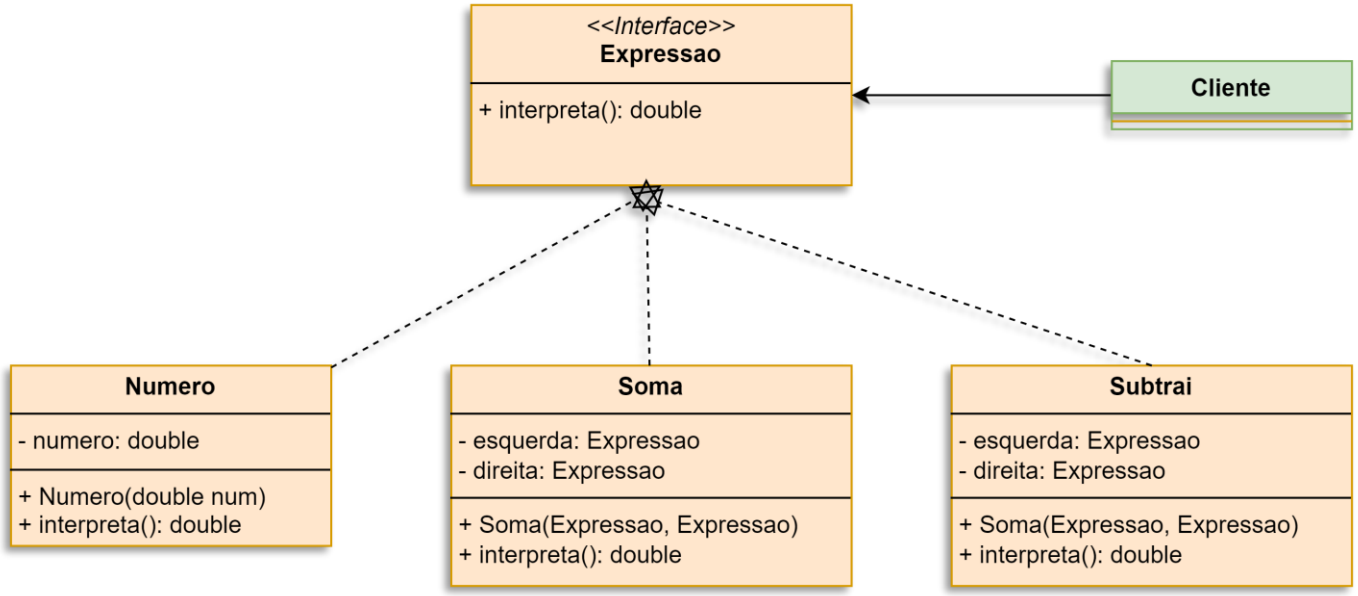
INTERPRETER

Um padrão que define uma gramática de uma linguagem e um interpretador para ela.



INTERPRETER

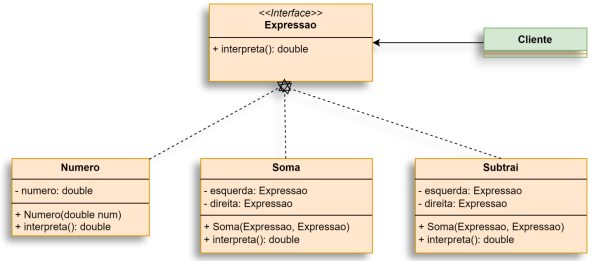
```java
class Numero implements Expressao {
    private double numero;

    public Numero(double number) {
        this.number = number;
    }

    @Override
    public double interpreta() {
        return this.numero;
    }
}
```




```java
public static void main(String[] args) {
    // (5 + 10) - (2 + 3)
    Expressao expressao = new Subtrair(
        new Soma(new Numero(5), new Numero(10)),
        new Soma(new Numeror(2), new Numero(3))
    );

    double total = expressao.interpreta();
    System.out.println("Resultado: " + total);
}
```

```java
@Override
public double interpreta() {
    return this.expEsquerda.interpreta() + this.expDireita.interpreta();
}
```

INTERPRETER



Componentes

- **Expressão abstracta:** a interface para a interpretação do contexto
- **Expressão terminal:** implementa a interpretação para as instruções terminais
- **Expressão não terminal:** implementa a interpretação para um grupo de instruções
- **Contexto:** as informações necessárias para a interpretação
- **Cliente:** constroi a árvore de interpretação

Benefícios

- **Extensão:** facilidade em adicionar novas regras gramaticais
- **Clareza:** representa directamente um gramática de uma linguagem
- **Flexibilidade:** permite diversas interpretações para uma mesma gramática
- **Reutilização:** devido à sua natureza, um linguagem permite diversas construções para diversos cenários
- **Simplicidade:** facilita a criação de linguagens para sistemas de configuração ou consultas de dados.

Desvantagens/Desafios

- **Eficiência e complexidade:** se a linguagem for complexa, pode ser dificil de manter. A árvore da gramática pode começar a ser dificil de gerir devido a uma grande quantidade de nós.
- **Frequência de uso:** não é muito usado, e tem um objetivo muito concreto Muitas vezes já existem soluções alternativas.
- **Debug:** hierarquias têm sempre os seus desafios no que respeita a debug e troubleshooting.

Casos de uso

- **Consultas de base de dados:** SQL e linguagens semelhantes
- **Formatos usados em Protocolos:** XML e JSON
- **Expressões regulares**
- **Conversão:** ex: tradução de números romanos para decimal.
- **Interpretação de formatos de dados:** conversões de dados, números, e outros formatos

INTERPRETER

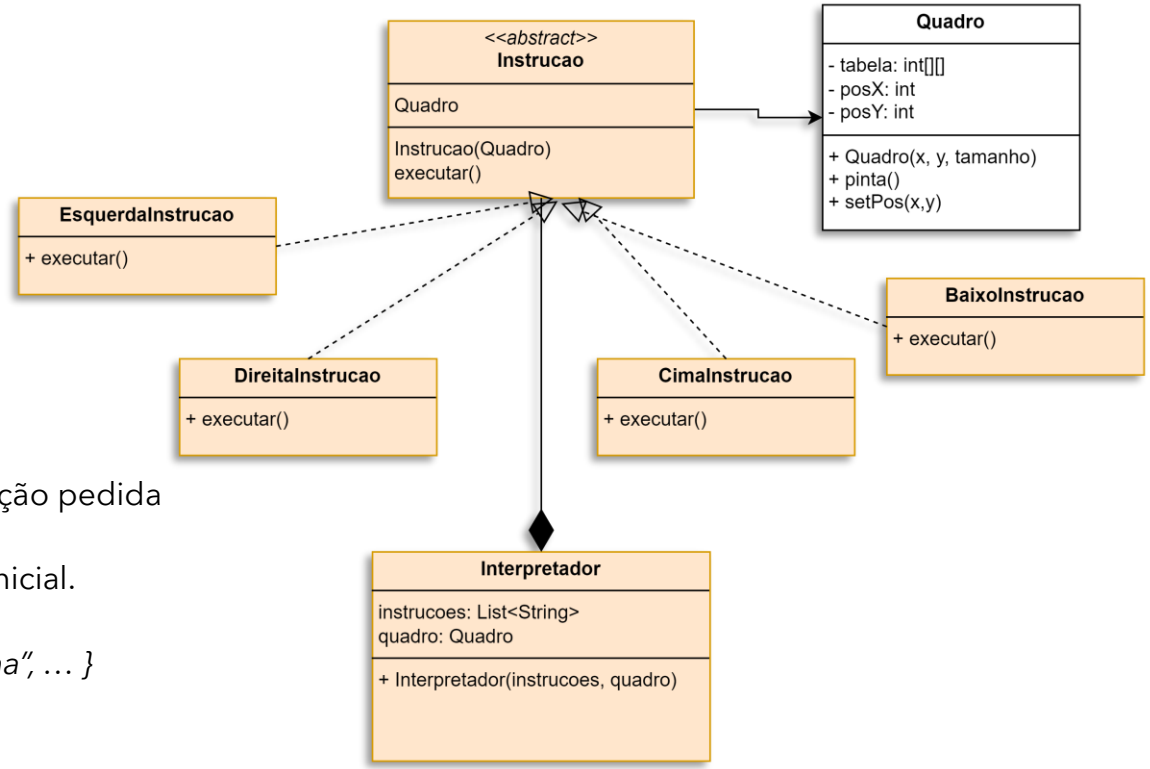
Actividade: Desenha num quadro (fácil)

Objetivo: usando o padrão interpreter, desenvolva uma forma de desenhar num quadro.

- Crie um array NxN: *int[] quadro;*
- Uso os comandos:
  - **esquerda**
  - **direita**
  - **cima**
  - **baixo**

	0	1	2	3
0				
1		1		
2		1		
3		1	1	
4			1	
5			1	
6		1	1	
7				
8				

- O comando faz "desenhar" um ponto na posição atual e move-se na direção pedida
- Cria uma classe **Quadro**, no constructor colocar o tamanho e a posição inicial.
- As instruções são dadas por uma lista de *Strings: {"esquerda", "cima", "cima", … }*



INTERPRETER

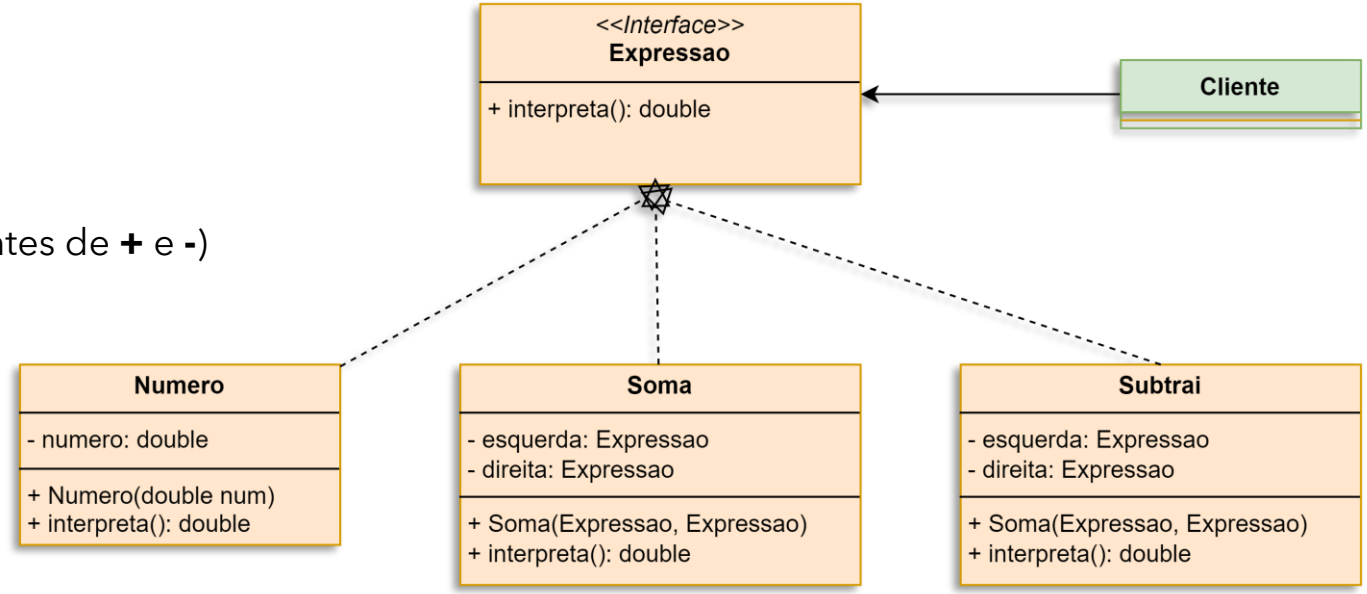
Actividade: Calculadora básica (fácil-médio)

Objetivo: usando o padrão Interpreter desenvolva uma forma de avaliar expressões matemáticas para os 4 operadores: +, -, /, *

- A expressão será uma String: “25 + 67” com dois termos
- Não há parêntesis nem precedência

OPCIONAL-1:

- Incluir precedência e múltiplos termos (***** e **/** antes de **+** e **-**)



INTERPRETER

Actividade: MeuRedis – Cache (médio)

Objetivo: usando o padrão interpreter, implementar um sistema de cache idêntico-ish ☺ ao Redis

INTEIROS	(OPCIONAL) TABELA PONTUAÇÃO
CREATE_I <var> <valor> - cria	**CREATET** <tabela>
INCRI <var> <valor> - incrementa	**ADDT** <tabela> <jogador>
DECRI <var> <valor> - decrementa	**SETT** <tabela> <jogador> <valor>
CLEARI <var> - zera a variavel	**INCRT** <tabela> <jogador> <valor>
GETI <var> - devolve o valor	**DECRT** <tabela> <jogador> <valor>
	CLEART <tabela>
	GETT <tabela>

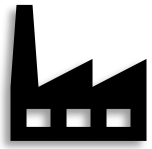
--| *Exemplo INTEIROS:*

CREATEI acessos_serv1 0
INCRI acessos_serv1 1
INCRI acessos_serv1 1
INCRI acessos_serv1 1
INCRI acessos_serv1 1
CREATEI acessos_serv2 0
INCRI acessos_serv1 2
GETI acessos_serv1

--| Exemplo TABELA
CREATET voltas
ADDT voltas joao
ADDT voltas elpepe
INCRT voltas joao 1
INCRT voltas joao 1
INCRT voltas elpepe
GETT voltas

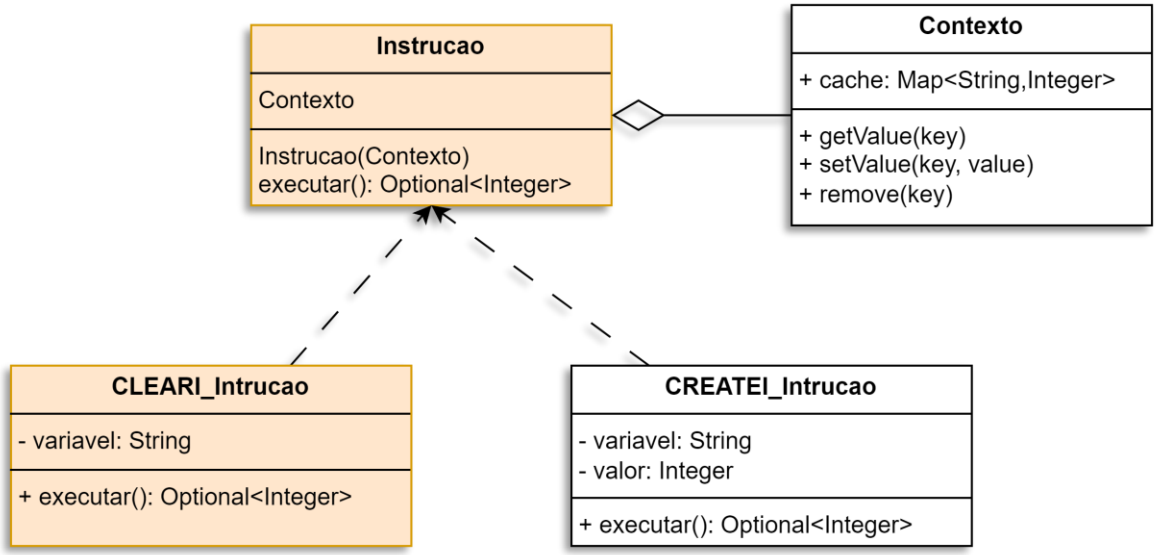
Voltas:
joao | 2
elpepe | 1

INTERPRETER



Actividade: MeuRedis – Cache (médio)

```
List<String> instrucoes = List.of(
        "CREATEI acessos-serv1 0",
        "INCRI acessos-serv1 1",
        "INCRI acessos-serv1 1",
        "INCRI acessos-serv1 1",
        "INCRI acessos-serv1 1",
        "DECRI acessos-serv1 1",
        "INCRI acessos-serv1 1",
        "CREATEI acessos-serv2 0",
        "INCRI acessos-serv2 2",
        "GETI acessos-serv1",
        "GETI acessos-serv2"
);

Contexto contexto = new Contexto();
for (String instrucao : instrucoes) {
    String[] items = instrucao.split(" ");
    switch (items[0].trim()) {
        case "CREATEI":
            new CREATEI_Instrucao(contexto, items[1], Integer.parseInt(items[2])).executar();
            break;
```



NO JDK

```java
// java.util.regex
public static void main(String[] args) {
    String regex = "\\d+";
    String input = "123 abc 456";

    Pattern pattern = Pattern.compile(regex);
    Matcher matcher = pattern.matcher(input);

    while (matcher.find()) {
        System.out.println("Encontrado: " + matcher.group());
    }
}
```

```java
// java.text.DateFormat
public static void main(String[] args) {
    DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
    Date date = new Date();
    String formattedDate = dateFormat.format(date);

    System.out.println("Data formatada: " + formattedDate);
}
```

```java
// java.nio.CharSet
public static void main(String[] args) {
    String input = "Olá, mundo!";
    byte[] bytes = input.getBytes(StandardCharsets.UTF_8);
    String output = new String(bytes, Charset.forName("UTF-8"));

    System.out.println("Texto original: " + input);
    System.out.println("Texto decodificado: " + output);
}
```

ANTI PATTERNS

Gramática Complexa: Criar uma gramática excessivamente complexa que torna o interpretador difícil de entender e manter.

Falta de Otimização: Não otimizar o interpretador para desempenho, resultando em uma execução lenta das expressões.

Interpretação Incompleta: Implementar um interpretador que não cobre todos os casos de uso necessários, deixando lacunas na funcionalidade.

Dependência Excessiva de Subclasses: Criar muitas subclasses para diferentes tipos de expressões, o que pode tornar o código difícil de gerenciar.

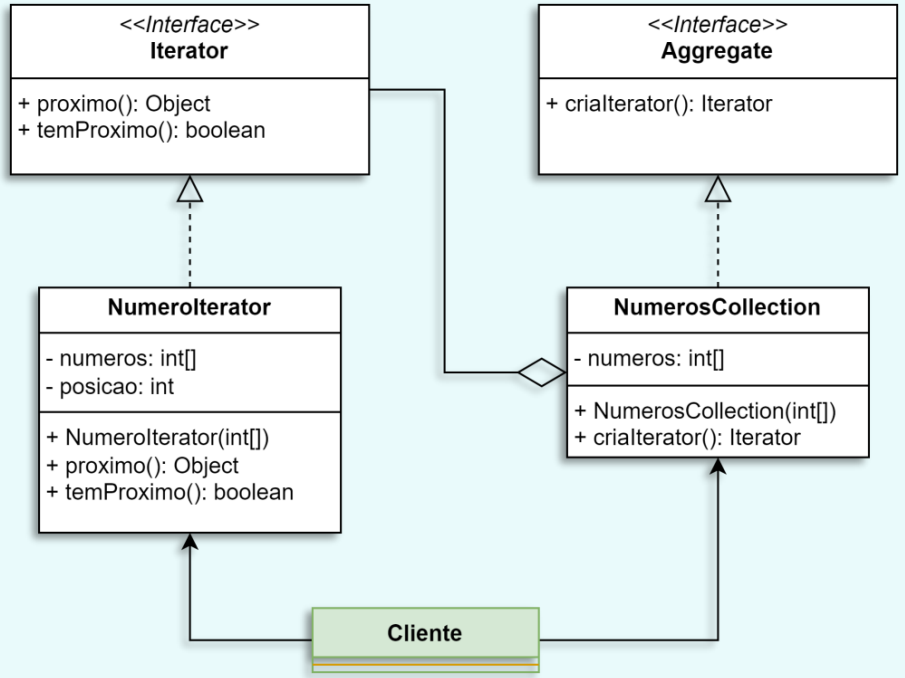
Falta de Documentação: Não documentar adequadamente a gramática e o funcionamento do interpretador, dificultando a manutenção e a compreensão do código.

Interpretação Inline: Realizar a interpretação diretamente no código do cliente, em vez de encapsular a lógica de interpretação em classes dedicadas.

Acoplamento Forte: Manter um acoplamento forte entre o interpretador e outras partes do sistema, dificultando a reutilização e a modificação do interpretador.

ITERATOR

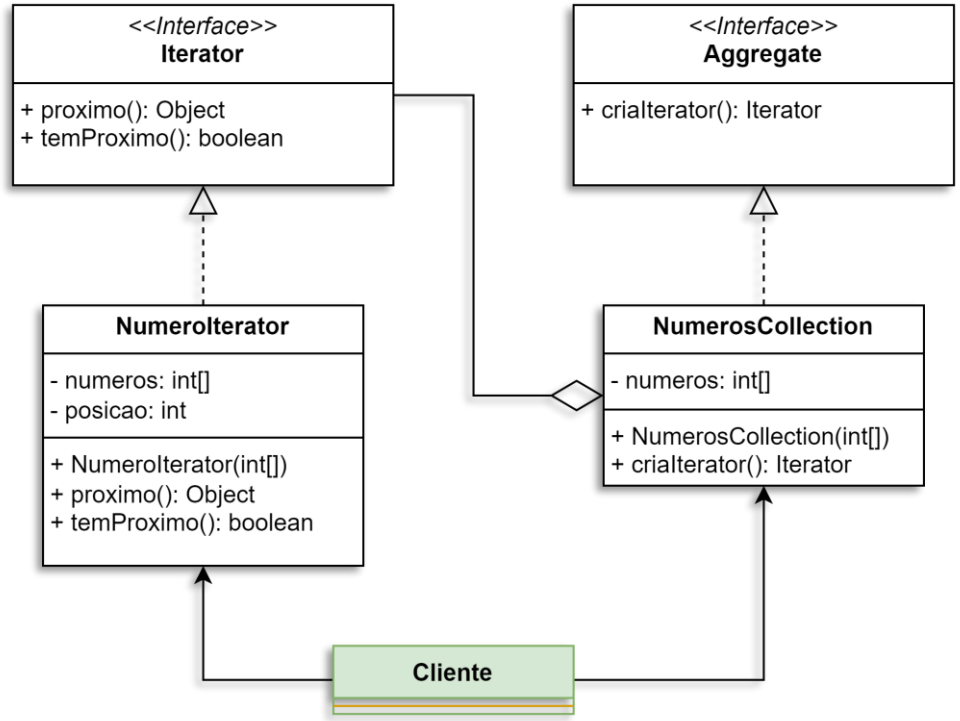
É um padrão que permite percorrer elementos de uma coleção de objetos de forma sequential, sem expor a representação subjacente da coleção.



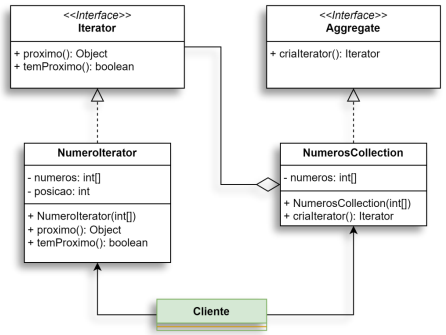
ITERATOR

```java
class NumeroIterator implements Iterator {
    private int[] numeros;
    private int posicao = 0;

    public NumeroIterator(int[] numeros) {
        this.numeros = numeros;
    }

    @Override
    public boolean temProximo() {
        return posicao < numeros.length;
    }

    @Override
    public Object proximo() {
        if (this.temProximo()) {
            return numeros[posicao++];
        }
        return null;
    }
}
```

```java
public static void main(String[] args) {
    int[] numeros = {1, 2, 3, 4, 5};
    NumeroCollection numeros = new NumerosCollection(numeros);
    Iterator iterator = numeros.criaIterator();

    while (iterator.temProximo()) {
        System.out.println(iterator.proximo());
    }
}
```



ITERATOR



Componentes

- **Iterator**: interface que declara os métodos necessários para iterar sobre uma coleção (ex: proximo, temProximo)
- **Implementação do interator**: a implementação da interface.
- **Aggregate**: uma interface que declara um método para criar um iterator
- **Implementação do aggregate**

Benefícios

- **Encapsulamento**: permite percorrer uma coleção sem expor a sua representação subjacente, nem a sua estrutura de dados.
- **Facilidade**: os clientes têm uma forma simples de percorrer coleções.
- **Reutilização**: normalmente a implementação é genérica o suficiente para ser reutilizada em múltiplas coleções.

Desvantagens/Desafios

- **Desempenho**: um iterator pode ser menos eficiente do que estruturas onde é possivel aceder diretamente aos elementos usando endereços de memória.
- **Flexibilidade**: os iterators têm uma forma de percorrer as coleções rígida, que pode não ser adequada para alguns tipos de estruturas.

Casos de uso

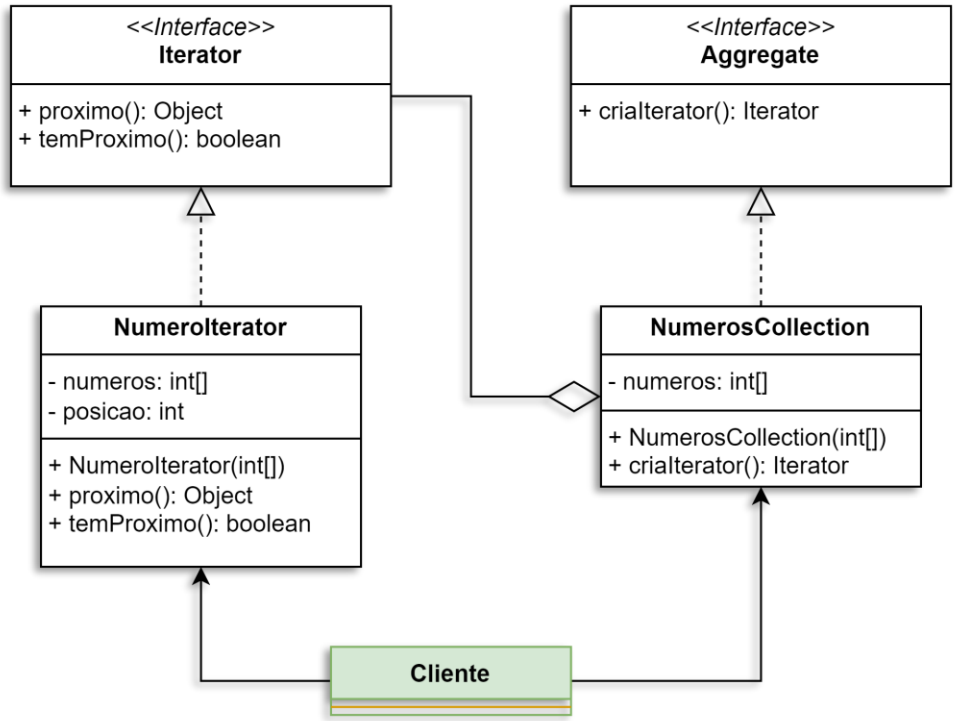
- **Percorrer coleções:** O seu maior uso
- **Em APIs:** para efectuar chamadas em sequência para obter dados
- **Frameworks de query:** ex: linq do .net, quer permite queries em objetos
- **Interfaces GUI:** para percorrer os componentes/objetos de uma interface (Lists, components num container)
- **Pipelines de dados:** garantir a execução em sequência nos componentes de um pipeline.
- **Árvores e Grafos:** quando há um ordem a estabeler, é uma das formas de "linear" estas estruturas.

ITERATOR

Actividade: Array de int[] (fácil)

Objetivo: usando o padrão Iterator, crie uma forma de percorrer qualquer Array de int[].

- Dado um array de *int* desenvolver um iterator
- OPCIONAL:
  - Transformar num iterator de Array de qualquer tipo
  - (usar genéricos)



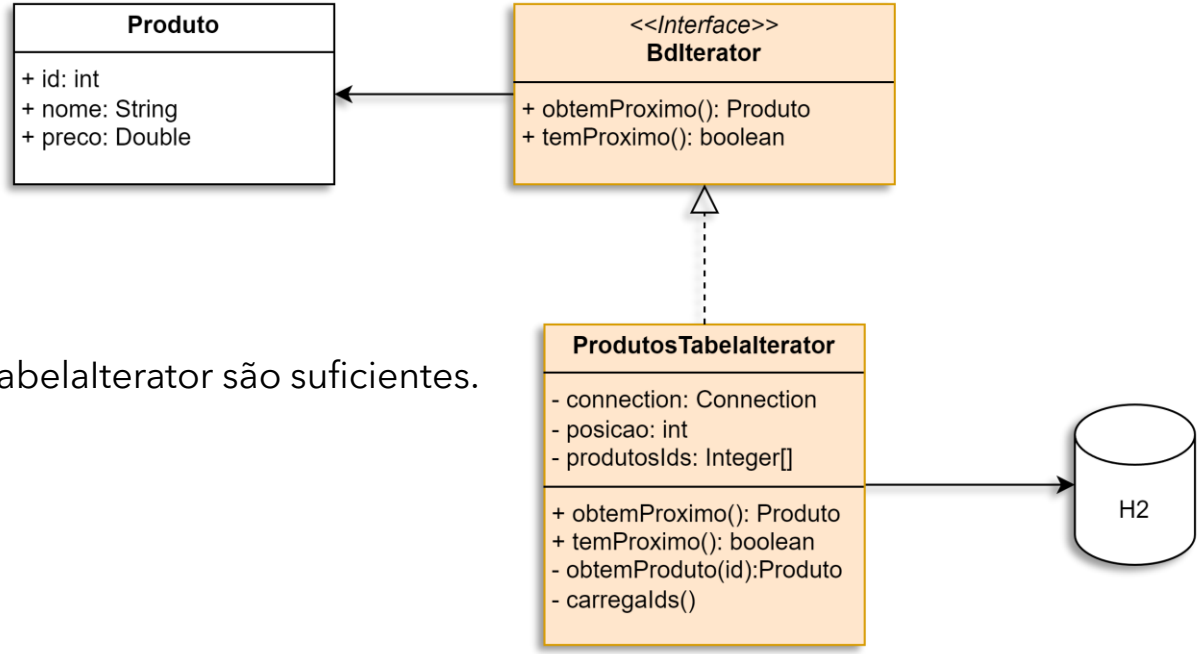
ITERATOR

Actividade: Iterator sobre uma tabela (médio)

Objetivo: usando o padrão Iterator, desenvolva uma forma de obter os registos de uma tabela de produtos, um a um.

- Usar a a base de dados **H2**
- Criar uma tabela ***Produtos***
  - *id int,*
  - *nome varchar(100),*
  - *preco decimal(10.2))*

NOTA: Neste exemplo as classes para BdIterator e ProdutosTabelaIterator são suficientes.

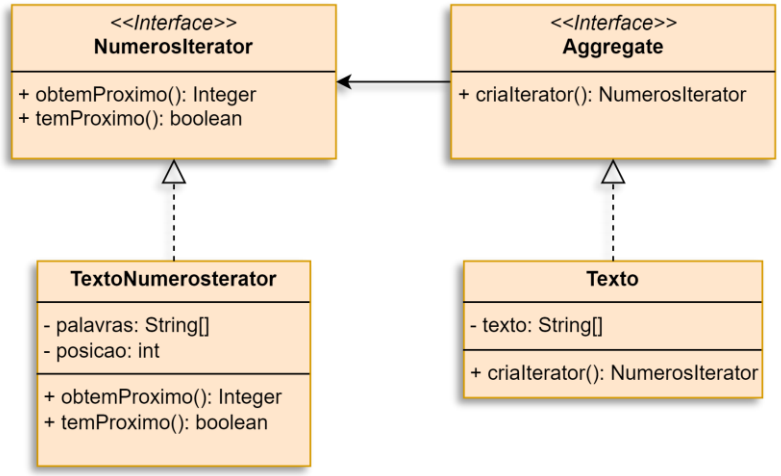


ITERATOR

Actividade: subconjuntos de números (médio)

Objetivo: usando o padrão iterator e um texto com palavras e números, crie um iterator que vai devolvendo apenas os números da frase.

- **Texto**: *"O João tem 3 gatos e 2 cachorros. Ele mora na casa número 45 da Rua das Flores. Em 2023, ele planeja adotar mais 1 gato e 2 pássaros."*
- NOTA: Pode criar um array a partir do Texto: *String[]* ***palavrasENumeros*** *= texto.split("[\\s.,;!?]+");*



NO JDK

```java
// ArrayList
ArrayList<String> list = new ArrayList<>();
list.add("Apple");
list.add("Banana");
list.add("Cherry");

Iterator<String> iterator = list.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next());
}
```

```java
// HashSet
HashSet<Integer> set = new HashSet<>();
set.add(1);
set.add(2);
set.add(3);

Iterator<Integer> iterator = set.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next());
}
```

ANTI PATTERNS

Iteradores Expostos: Expor iteradores diretamente ao cliente, permitindo que o cliente manipule o estado interno da coleção, o que pode levar a inconsistências.

Falta de Encapsulamento: Não encapsular a lógica de iteração dentro do iterador, resultando em código duplicado e difícil de manter.

Iteradores Dependentes: Criar iteradores que dependem fortemente de detalhes internos da coleção, quebrando o princípio de encapsulamento.

Iteradores Não Reentrantes: Implementar iteradores que não suportam reentrância, ou seja, não podem ser usados de forma segura em contextos concorrentes.

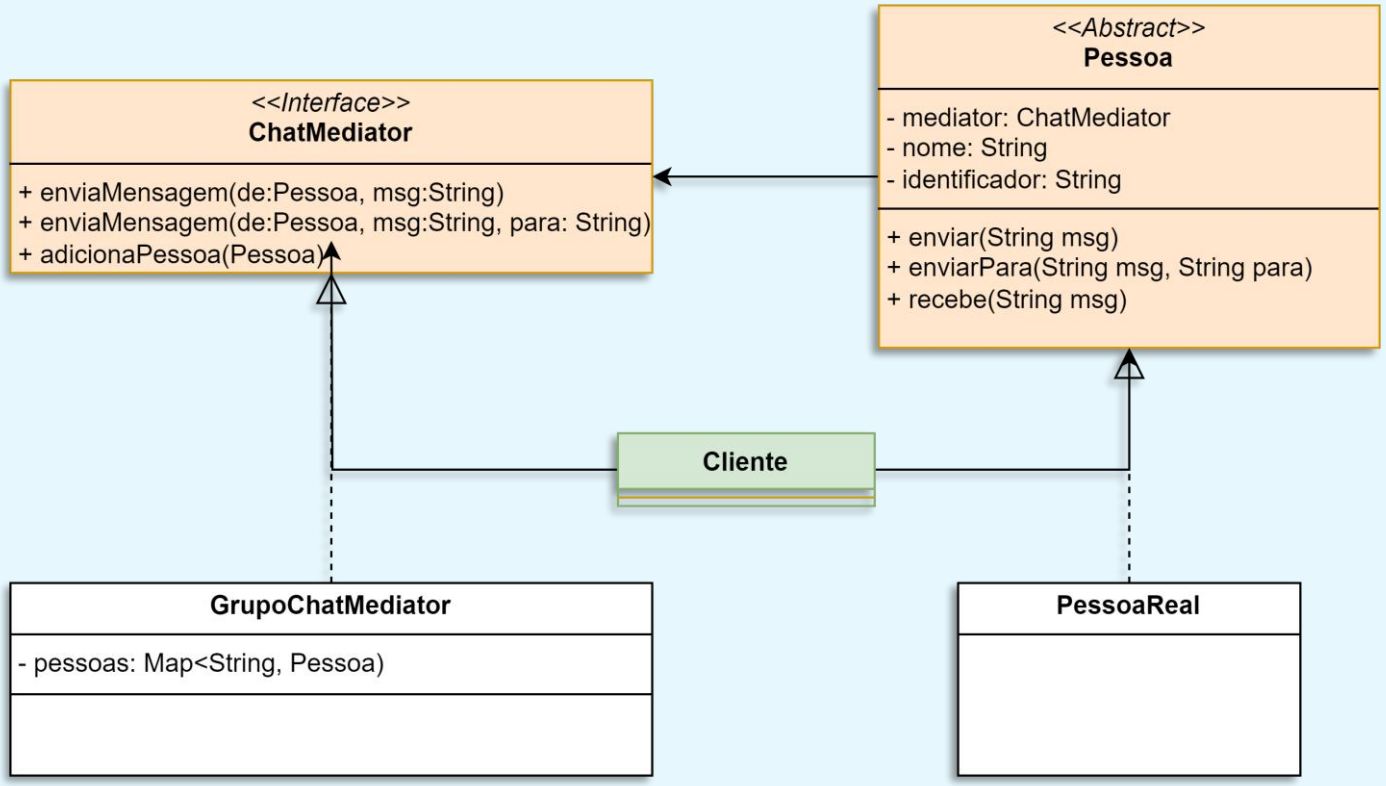
Falta de Suporte a Remoção: Não implementar métodos para remover elementos durante a iteração, limitando a funcionalidade do iterador.

Iteradores Complexos: Criar iteradores com lógica complexa demais, tornando-os difíceis de entender e manter.

Iteradores Sem Estado: Implementar iteradores que não mantêm estado suficiente para realizar a iteração corretamente, resultando em comportamento inesperado.

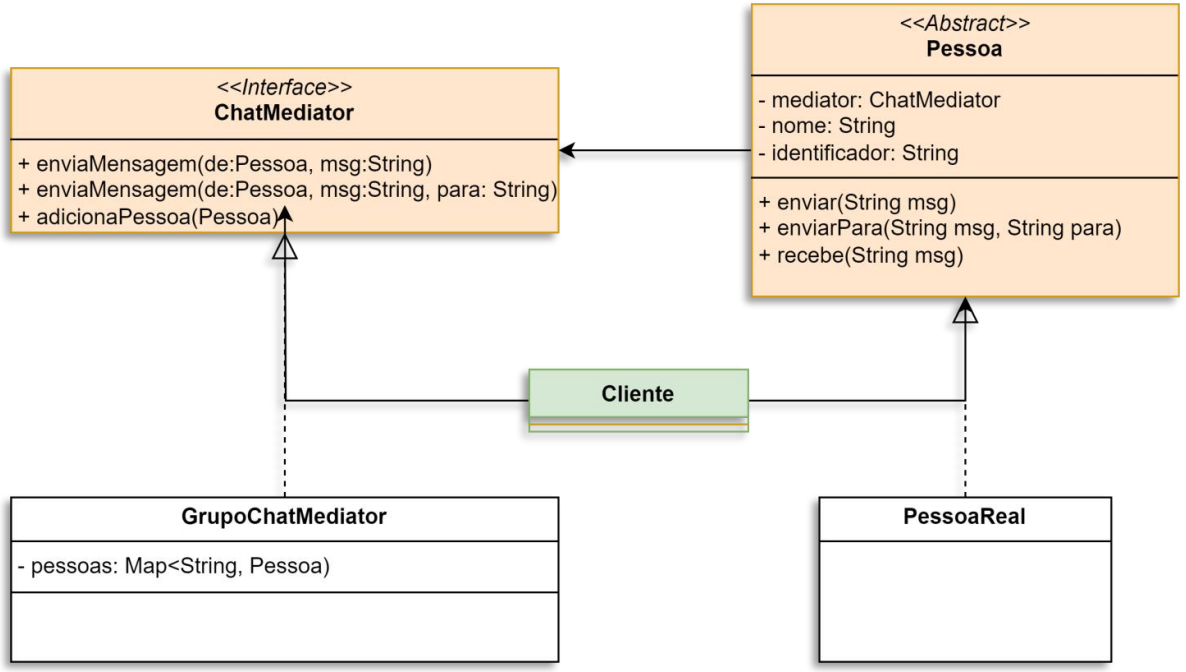
MEDIATOR

É um padrão que facilita a comunicação entre objetos em um sistema, agindo como um intermediário para centralizar e gerenciar a comunicação.

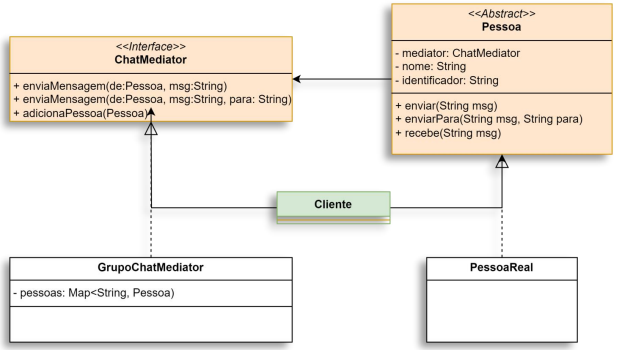


MEDIATOR

```java
ChatMediator mediator = new GrupoChatMediator();

Pessoa joao = new PessoaReal("1", "joão", mediator);
Pessoa ana = new PessoaReal("2", "ana", mediator);
Pessoa sandra = new PessoaReal("3", "sandra", mediator);
Pessoa pedro = new PessoaReal("4", "pedro", mediator);
Pessoa maria = new PessoaReal("5", "maria", mediator);

mediator.adicionaPessoa(joao);
mediator.adicionaPessoa(ana);
mediator.adicionaPessoa(sandra);
mediator.adicionaPessoa(pedro);
mediator.adicionaPessoa(maria);

// vamos conversar
joao.enviar("Olá pessoal!");
pedro.enviarPara("Olá maria", "5");
```



MEDIATOR



Componentes

- **Mediator**: interface que define os métodos para a comunicação entre os objetos.
- **ConcreteMediator**: a implementação dos métodos do Mediator
- **Colleague**: a interface que define os métodos que os objetos participantes devem implementar.
- **ConcreteColleague**: as implementações da interface dos objetos participantes.

Benefícios

- **Redução do coupling**: os objetos participantes só precisam de conhecer a interface do Mediator.
- **Manutenção**: alterações em objetos não afectam os outros
- **Flexibilidade e Modularidade**: a substituição de objetos é mais simples
- **Escalabilidade**: favorece a escalabilidade, visto que a adição de novos objetos não aumenta a complexidade das comunicações.

Desvantagens/Desafios

- **Centralização**: tudo depende do Mediator, o que pode tornar a gestão do mesmo complexo
- **Bottleneck**: pode-se tornar um ponto crítico do Sistema, tudo passa por ele
- **Debug e troubleshooting**: testes, debug e troubleshooting pode ser complicado devido ao coupling com todos os outros objetos.

Casos de uso

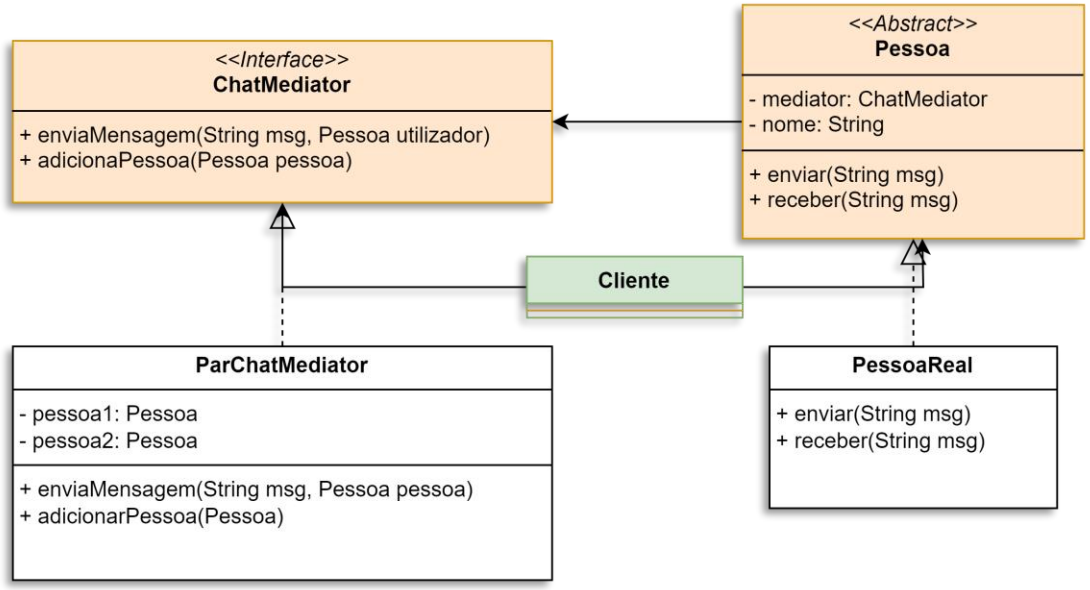
- **Interfaces Gráficas de Utilizador (GUI):** o mediator pode facilitar a comunicação entre os diversos components gráficos
- **Sistemas de Workflow:** pode servir de coordenador entre os componentes do workflow, garantido o fluxo pre-estabelecido
- **Sistemas de Chat:** serve de encaminhador das mensagens entre os participantes
- **Simulações:** para gerir as interações entre os components, entidades, etc.
- **Aplicações de Comércio Eletrónico:** pode ajudar na interação entre carrinho, pagamentos, inventários, etc.

MEDIATOR

Actividade: conversa a dois (fácil)

Objetivo: usando o padrão Mediator, crie um sistema para uma conversa a dois.

- A classe Mediator define dois intervenientes: ex: pessoa1, pessoa2: Pessoa
- Quando a pessoa1 fala, a pessoa2 recebe a mensagem, e vice-versa.

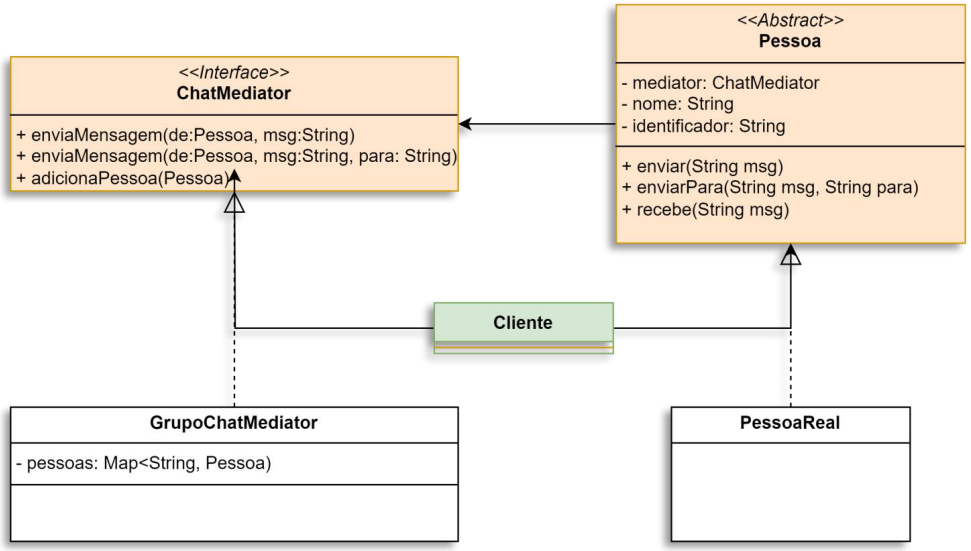


MEDIATOR

Actividade: Chat (médio)

Objetivo: usando o padrão Mediator implemente um sistema de chat entre Pessoas.

- Indique visualmente quem está a enviar a mensagem, Ex: *(joão): olá a todos*
- A pessoa pode enviar a mensagem para todos ou especificar uma pessoa em concreto
  - Usar um número para identificar a pessoa (ex: telefone ☺), mas aceitar também um nome para o chat.

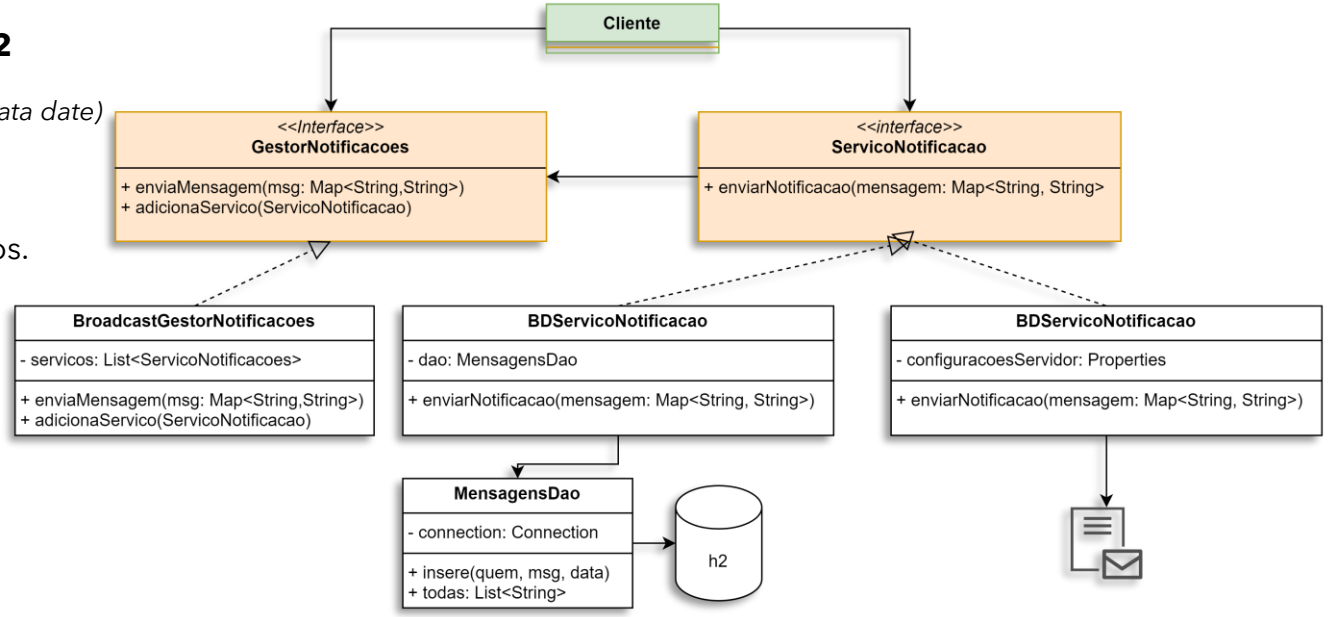


MEDIATOR

Actividade: Sistema de notificações Email e BD (avançado)

Objetivo: usando o padrão Mediator, implemente um sistema de envio de notificações, que permite enviar mensagens para serviços registados como Email e Base de dados.

- NOTA: é idêntico a um sistema de Chat, mas as "pessoas" são os serviços.
- Usar uma tabela de mensagens numa base de dados **H2**
  - ***mensagens*** *(fonte varchar(100), mensagem varchar(500), data date)*
- Opcionalmente usar o sistema de email **Mailpit**
- Podemos assumir que a mensagem é enviado para todos.




```
Map<String, String> mensagem = Map.of(
        "mensagem", "Olá, tudo bem? Espero que sim!",
        "subject", "hello",
        "from", "joao@teste.com",
        "to", "to@someone.com",
        "quem", "joão");
```

NO JDK

```java
// java.util.Timer
Timer timer = new Timer();
TimerTask task = new TimerTask() {
    @Override
    public void run() {
        System.out.println("Tarefa executada!");
    }
};
timer.schedule(task, 1000);
```

```java
// java.util.concurrent.Executor
ExecutorService executor = Executors.newFixedThreadPool(2);
Runnable task1 = () -> System.out.println("Tarefa 1 executada");
Runnable task2 = () -> System.out.println("Tarefa 2 executada");

executor.execute(task1);
executor.execute(task2);
executor.shutdown();
```

ANTI PATTERNS

Mediator God Object: Criar um mediador que centraliza demasiada lógica, tornando-se um “objeto deus” que conhece e controla tudo, o que pode levar a um código difícil de manter e testar.

Falta de Coesão: Adicionar demasiadas responsabilidades ao mediador, resultando em uma falta de coesão e violação do princípio de responsabilidade única.

Comunicação Direta: Permitir que os colegas (colleagues) se comuniquem diretamente entre si, em vez de através do mediador, quebrando o propósito do padrão.

Mediador Inflado: Continuar adicionando funcionalidades ao mediador sem refatorar, o que pode levar a um mediador inflado e complexo.

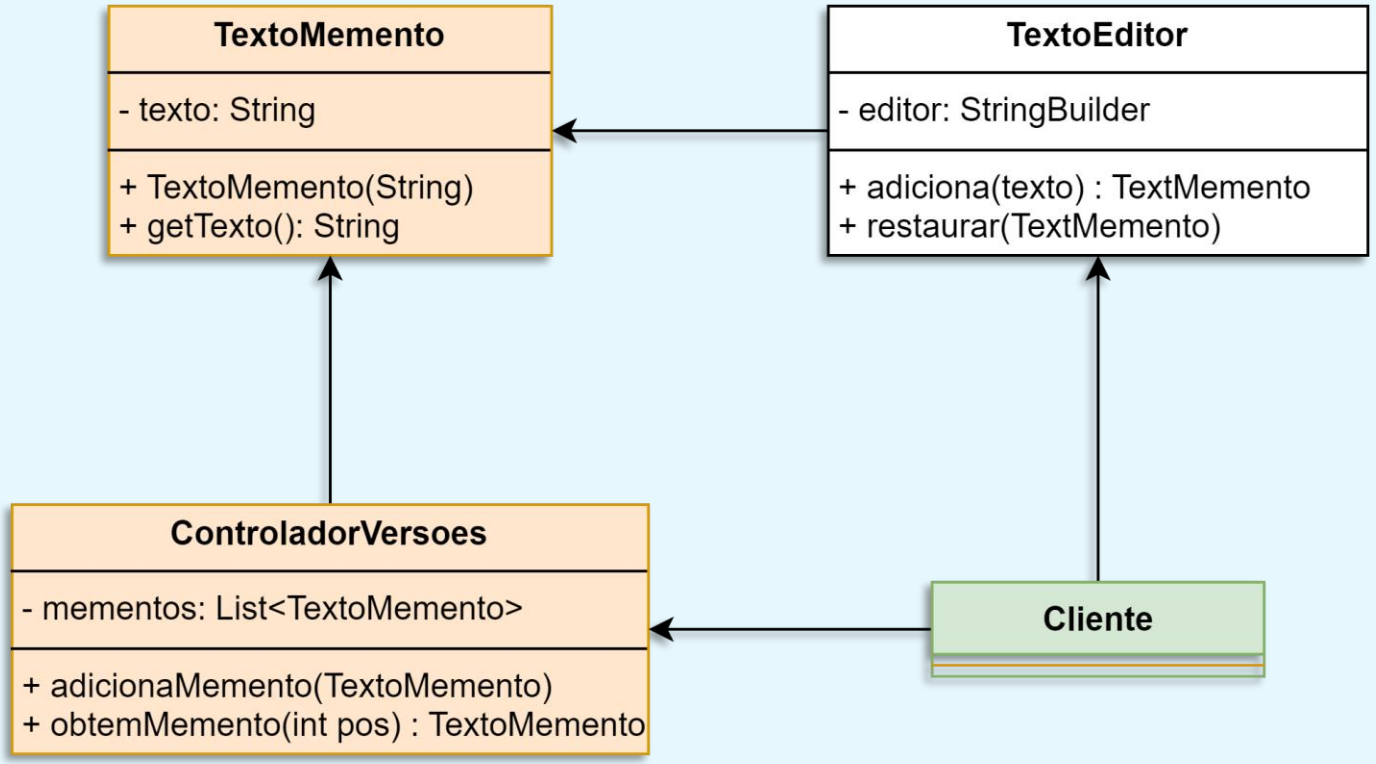
Dependência Circular: Criar dependências circulares entre o mediador e os colegas, o que pode complicar o fluxo de controle e dificultar a depuração.

Falta de Documentação: Não documentar adequadamente as interações mediadas, tornando difícil para outros desenvolvedores entenderem e manterem o código.

Uso Inadequado: Usar o padrão Mediator quando a comunicação direta entre objetos seria mais simples e eficiente, resultando em uma complexidade desnecessária.

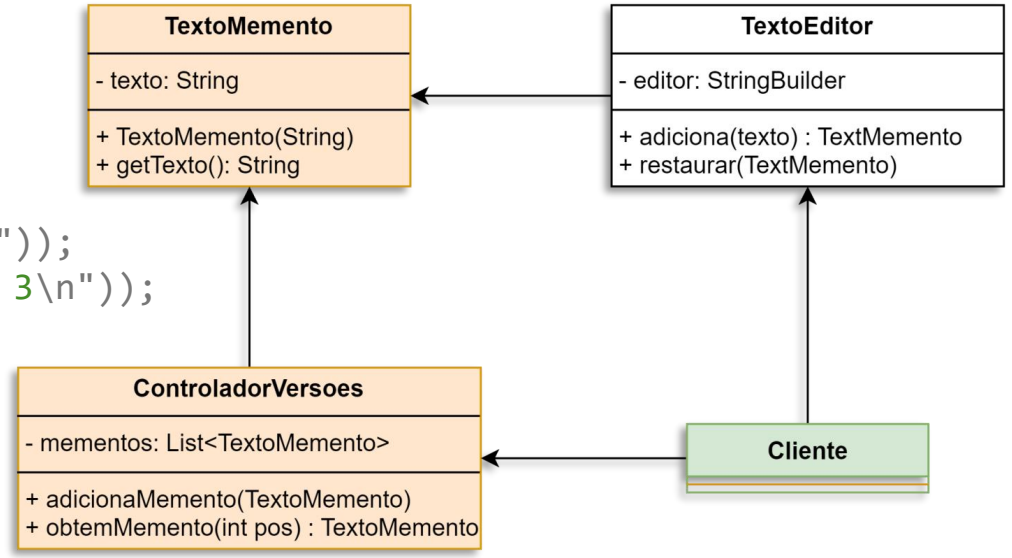
MEMENTO

Um padrão que permite capturar e armazenar o estado interno de um objeto sem violar o encapsulamento. Isso possibilita que o objeto seja restaurado para um estado anterior, se necessário.

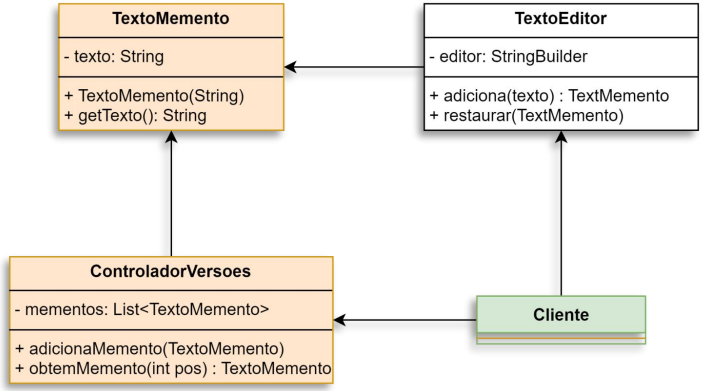


MEMENTO

```java
public class Memento {
    public static void main(String[] args) {

        ControladorVersoes cv = new ControladorVersoes();
        TextoEditor editor = new TextoEditor();

        cv.adicionaMemento(editor.adiciona("Olá tudo bem?\n"));
        cv.adicionaMemento(editor.adiciona("Este é o memento 2\n"));
        cv.adicionaMemento(editor.adiciona("e é este é o memento 3\n"));

        System.out.println(editor.texto());

        editor.restaurar(cv.obtemMemento(-1));

        System.out.println(editor.texto());
    }
}
```



MEMENTO



Componentes

- **Originator**: o objeto que pode necessitar de guardar o seu estado interno para poder ser restaurado.
- **Memento**: armazena o estado interno do Originador. Guarda uma cópia do estado ou uma forma de voltar ao estado pretendido.
- **Caretaker**: o gestor dos Mementos.

Benefícios

- **Escapsulamento**: permite guardar e restaurar o estado de um objeto sem expor os detalhes internos
- **Facilidade undo/redo**: a cópia dos estados facilita a gestão dos undo/redo.
- **Histórico**: guarda o histórico dos estados dos objetos
- **Redução de erros**: reduz erros e inconsistências, pois permite restaurar o sistema para um estado anteriores
- **Flexibilidade**: pode ser usado em editors, de texto, versionamento, etc.

Desvantagens/Desafios

- **Memória**: o aumento de Mementos incrementa o uso de memória
- **Ciclo de vida**: o componente Caretaker tem de gerir os Mementos, nomeadamente aqueles que deixam de fazer sentido
- **Complexidade**: como todos os padrões, adiciona código não funcional

Casos de uso

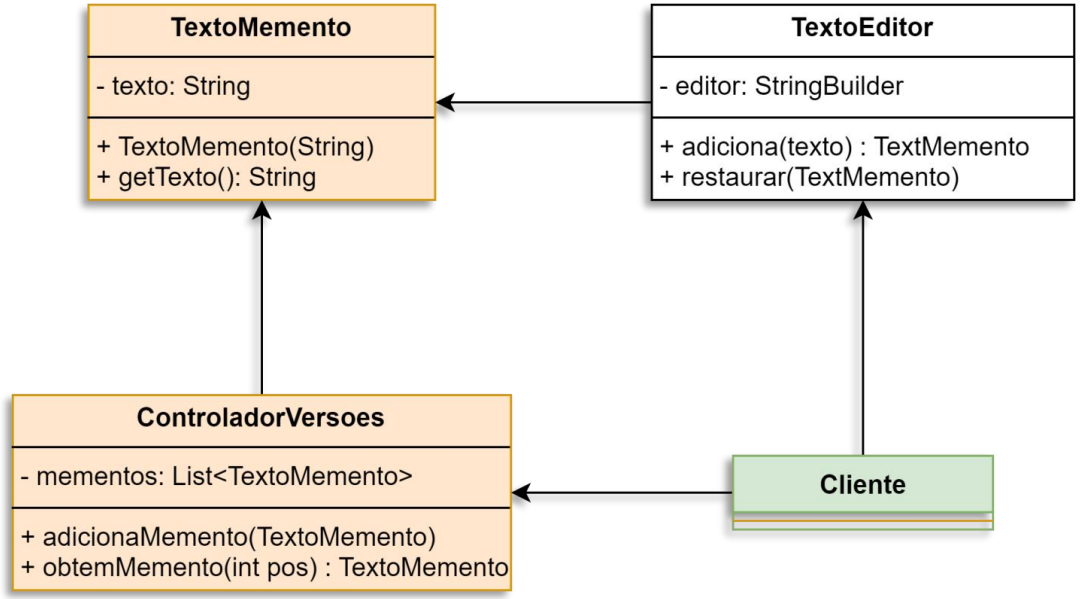
- **Aplicações com undo/redo:** a natureza do padrão
- **Backups e restores:** para a criação de pontos de recuperações, savepoints e semelhantes.
- **Controlo de versões:** versões podem ser vistas como undo/redo.
- **Simulações e Jogos:** para guardar estados e voltar a eles quando necessário
- **Transações bancárias:** commits e rollbacks.

MEMENTO

Actividade: Editor com undo (fácil)

Objetivo: usando o padrão Memento crie um editor de texto simples que permite o undo.

- O editor deve permitir adicionar texto e fazer undo n vezes

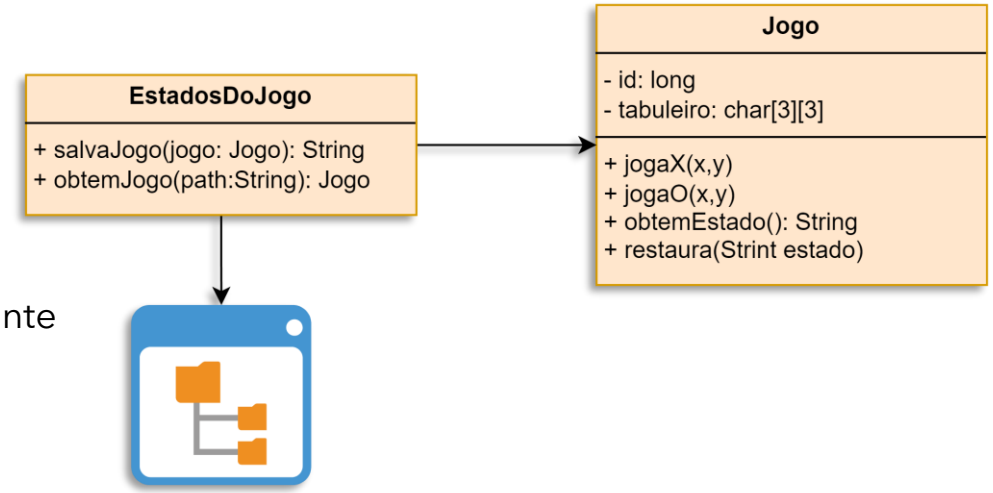


MEMENTO

Actividade: jogo do 3 em linha (fácil)

Objetivo: usando o padrão memento, construa uma tabela 3x3 que permite a gravação para um ficheiro do estado da tabela

- A tabela 3x3 (:char) contem ‘x’ e ‘o’
- Cada jogo tem um id único
- A qualquer momento pode-se pedir para gravar
  - Ex: 2024-01-01-456.txt
- O Jogo deverá permitir carregar um estado salvado anteriormente



NOTA: Não é necessário implementar o jogo, apenas a tabela

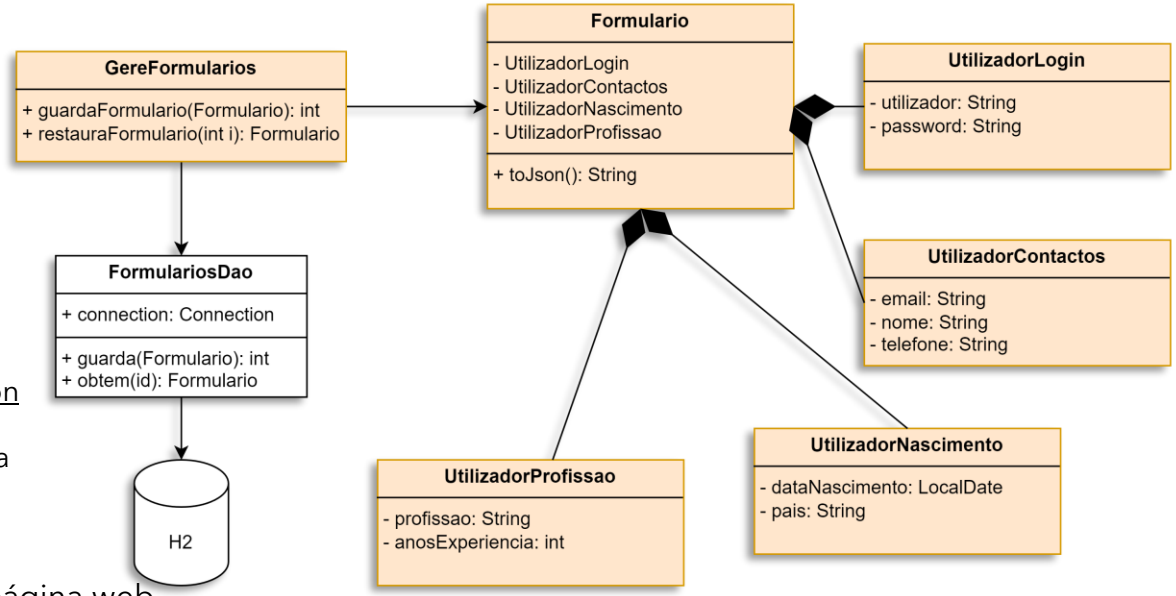
Exemplo do estado: “X0.X…..”

MEMENTO

Actividade: formulário (médio)

Objetivo: usando o padrão Memento, simula o preenchimento de um formulário que pode ter até 4 etapas.

- **Etapa 1**: utilizador e password
- **Etapa 2**: email, nome e telefone
- **Etapa 3**: data de nascimento e país
- **Etapa 4**: profissao e anos de experiencia
- Os dados são gravados numa tabela de base de dados
  - EX: *formularios (id, conteúdo, data)* – serializar o objecto em json
  - Apos cada gravação um id é devolvido para recuperação futura
- Pode-se sempre regressar a uma etapa anterior baseado no id
- NOTA: Imaginemos que os dados provêm de um wizard numa página web.



NOTA: Trate como se fosse uma transação com possibilidade de pontos de salvaguarda

NO JDK

```java
// Criar um UndoManager
UndoManager undoManager = new UndoManager();
textArea.getDocument().addUndoableEditListener(new UndoableEditListener() {
    @Override
    public void undoableEditHappened(UndoableEditEvent e) {
        undoManager.addEdit(e.getEdit());
    }
});

// Cria ações de Undo e Redo
Action undoAction = new AbstractAction("Undo") {
    @Override
    public void actionPerformed(ActionEvent e) {
        if (undoManager.canUndo()) {
            undoManager.undo();
        }
    }
};

Action redoAction = new AbstractAction("Redo") {
    @Override
    public void actionPerformed(ActionEvent e) {
        if (undoManager.canRedo()) {
            undoManager.redo();
        }
    }
};
```

```java
// Lista inicial de números
List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

// Checkpoint 1: Filtrando números pares
List<Integer> checkpoint1 = numbers.stream()
                                   .filter(n -> n % 2 == 0)
                                   .collect(Collectors.toList());
System.out.println("Checkpoint 1 (Números Pares): " + checkpoint1);

// Checkpoint 2: Multiplicando por 2
List<Integer> checkpoint2 = checkpoint1.stream()
                                       .map(n -> n * 2)
                                       .collect(Collectors.toList());
System.out.println("Checkpoint 2 (Multiplicado por 2): " + checkpoint2);

// Checkpoint 3: Somando 10
List<Integer> checkpoint3 = checkpoint2.stream()
                                       .map(n -> n + 10)
                                       .collect(Collectors.toList());
System.out.println("Checkpoint 3 (Somado 10): " + checkpoint3);

// Restaurando e continuando o processamento a partir do Checkpoint 2
List<Integer> restoredCheckpoint2 = checkpoint2.stream()
                                               .map(n -> n + 5)
                                               .collect(Collectors.toList());
System.out.println("Restaurado Checkpoint 2 e Somado 5: " + restoredCheckpoint2);
```

ANTI PATTERNS

Memento Inflado: Armazenar demasiada informação no memento, o que pode levar a um consumo excessivo de memória e a um desempenho degradado.

Falta de Encapsulamento: Expor detalhes internos do objeto ao memento, violando o princípio de encapsulamento.

Uso Excessivo de Mementos: Criar mementos para cada pequena mudança de estado, resultando em um grande número de mementos e consumo de recursos.

Mementos Não Versionados: Não versionar os mementos, dificultando a restauração de estados específicos em sistemas complexos.

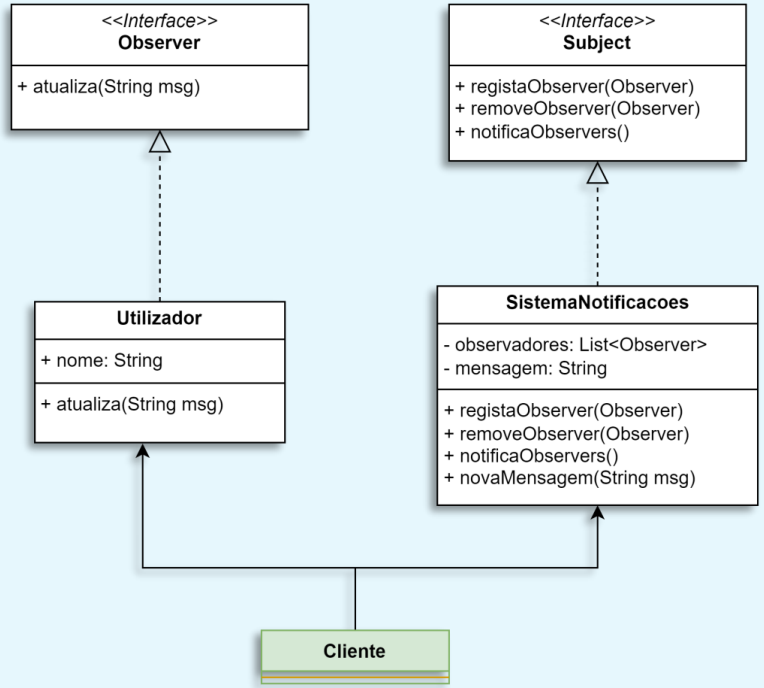
Falta de Limpeza: Não limpar mementos antigos, levando a um acúmulo de mementos desnecessários e ao desperdício de memória.

Dependência Circular: Criar dependências circulares entre o objeto e o memento, complicando o fluxo de controle e a manutenção do código.

Mementos Incompletos: Armazenar estados incompletos ou inconsistentes no memento, resultando em restaurações incorretas.

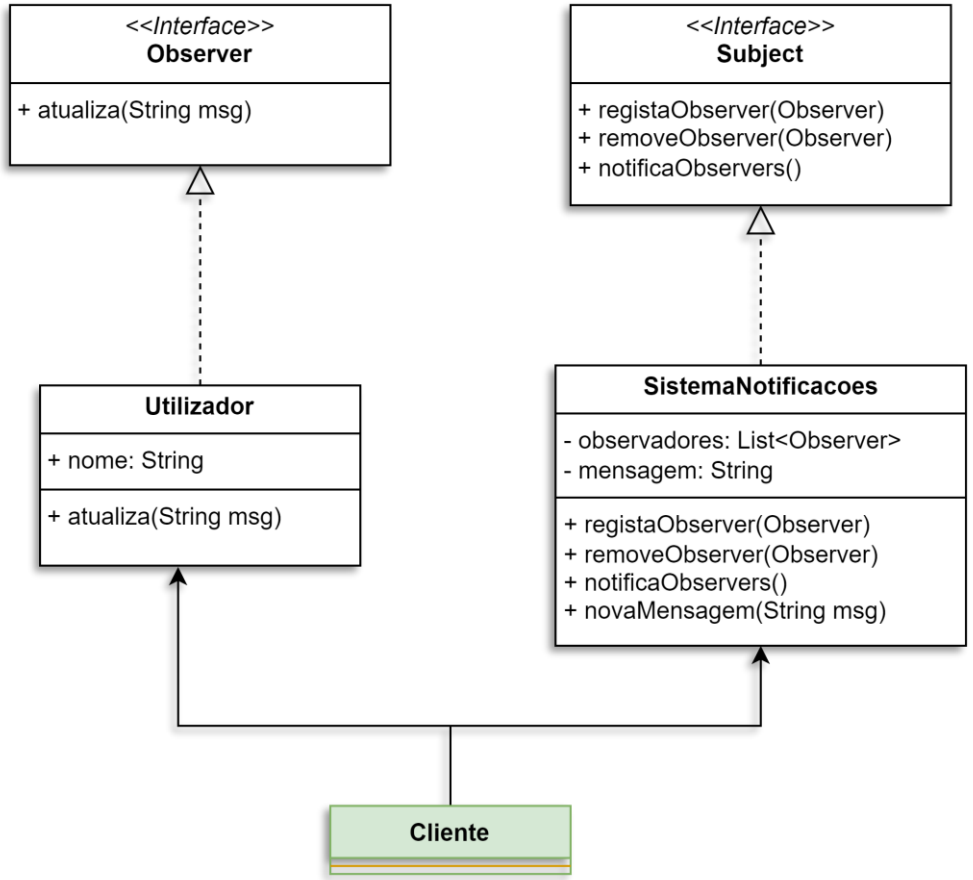
OBSERVER

é um padrão comportamental que define uma dependência um-para-muitos entre objetos. Quando um objeto (o sujeito ou observado) muda de estado, todos os seus dependentes (os observadores) são notificados e atualizados automaticamente.

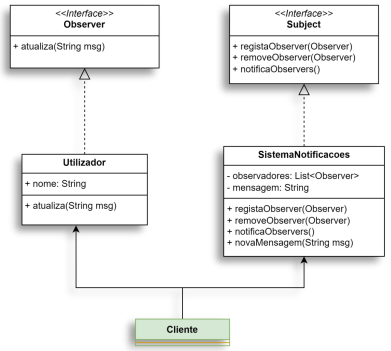


OBSERVER

```java
public static void main(String[] args) {
    SistemaNotificacoes SistemaNotificacoes = new SistemaNotificacoes();

    User user1 = new User("Alice");
    User user2 = new User("Pedro");
    User user3 = new User("Carlos");

    SistemaNotificacoes.registaObserver(user1);
    SistemaNotificacoes.registaObserver(user2);
    SistemaNotificacoes.registaObserver(user3);

    SistemaNotificacoes.newMessage("Nova mensagem disponível!");

    SistemaNotificacoes.removeObserver(user2);

    SistemaNotificacoes.newMessage("Outra mensagem importante!");
}
```



OBSERVER



Componentes

- **Subject**: mantém uma lista de observadores e fornece métodos para adicionar, remover e notifica-los.
- **Observer**: a interface com o método de receção de notificações
- **ConcreteSubject**: implementa o Subject. Notifica os observadores quando o estado muda.
- **ConcreteObserver**: implementa a interface do Observer.

Benefícios

- **Desacoplamento**: cria uma separação entre o subject e os observers
- **Flexibilidade**: é simples adicionar novos observadores
- **Atualização automática**: todos os observadores são notificados da mudança do estado do subject.
- **Escalabilidade**: permite que o número arbitrário de observadores seja associado a um subject.

Desvantagens/Desafios

- **Desempenho**: o crescimento do número de observadores reduz o desempenho do Sistema.
- **Ordem nas notificações**: se a ordem é importante, pode ser complicada a implementação.
- **Dependências circulares**: há risco de criar ciclos nas dependências.
- **Debug e troubleshooting**: o desacoplamento acarreta dificuldades em debug e troubleshooting.

Casos de uso

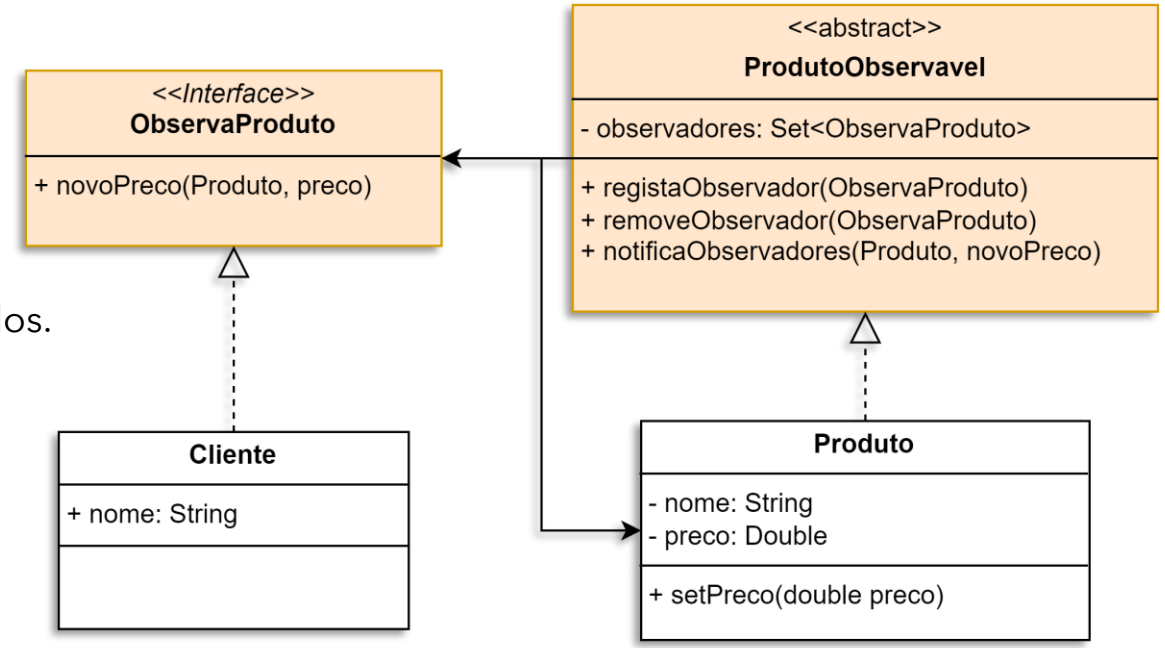
- **Sistemas de eventos**: quando há components que necessitam de ser notificados quando há alterações de estados
- **Interfaces gráficas (GUI)**
- **Sistemas de notificações:** alertas de email, push notifications, e semelhantes
- **Model-View-Controller:** para notificar o View para se atualizar quando o modelo muda.
- **Sistemas publish/subscriber:** assinantes têm de ser notificados, quando subscrevem determinados eventos
- **Monitorização:** alertas na saúde, alertas em sistemas, etc.

OBSERVER

Actividade: Alerta de preços (fácil)

Objetivo: usando o padrão Observer, implemente um sistema que monitoriza preço de produtos. Quando o preço desce notifica os clientes interessados.

- Criar Produto com preço.
- Criar Clientes
- Os clientes observam produtos (preços)
- Quando um preço desce notifica os clientes interessados.

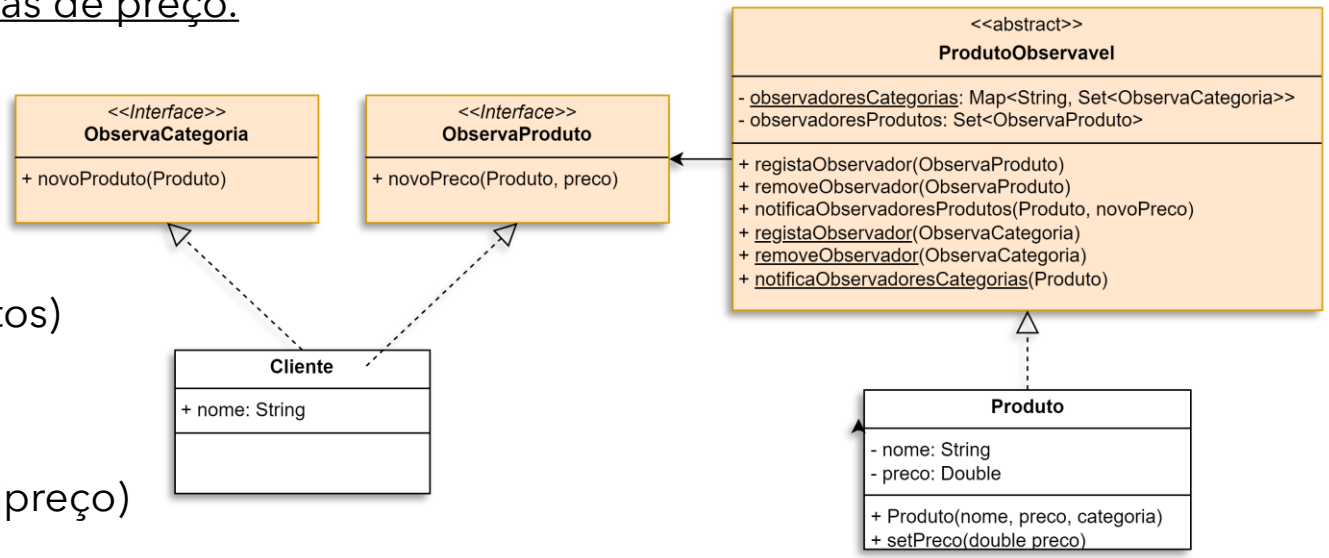


OBSERVER

Actividade: Categorias de produtos e variações (médio)

Objetivo: usando o padrão Observer, implemente um sistema que permite um cliente estar interessado em novos produtos de uma categoria, e em descidas de preço.

- Criar produtos: nome, categoria, preco
- Criar clientes (nome)
- Clientes observam categorias (novos produtos)
  - Um produto novo na categoria notifica
- Clientes observam determinados produtos (preço)
  - Um preço de um produto específico desce, notifica



OBSERVER

Actividade: utilizadores monitorizados (avançado)

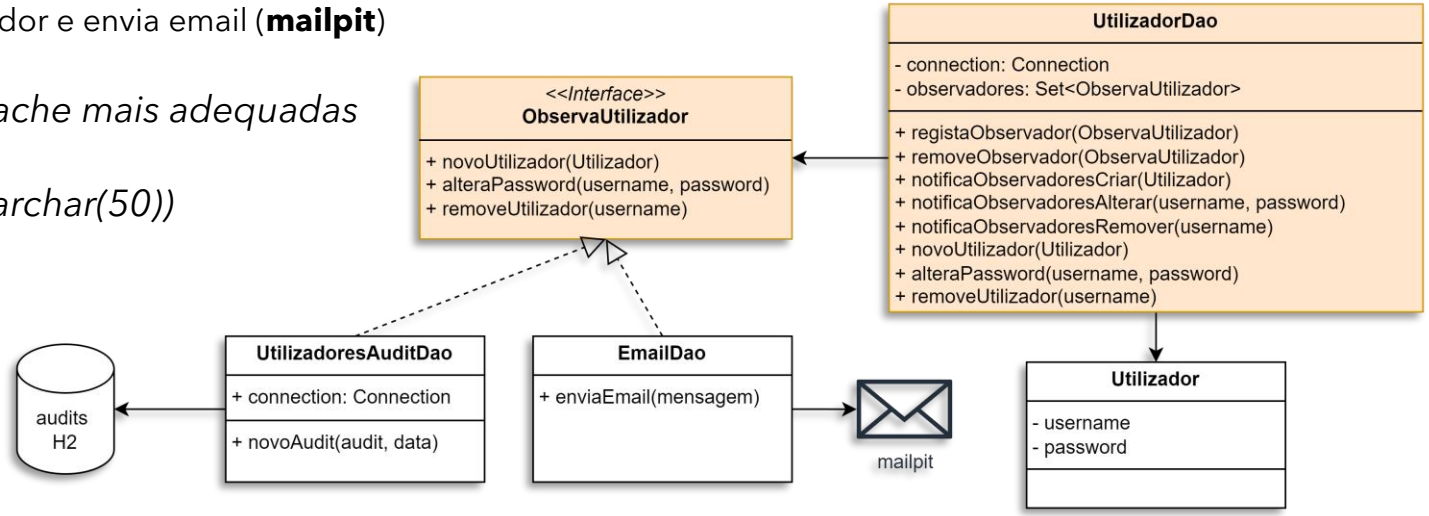
Objetivo: usando o padrão Observer, implemente um sistema de criação de utilizadores na base de dados monitorizados.

- Quando um **Utilizador** é criado na base de dados usando um **UtilizadoresDao**
  - O observador **UtilizadoresAuditDao** recebe o **Utilizador** e envia para uma tabela de **audits**.
  - O observador **EmailDao**, recebe o Utilizador e envia email (**mailpit**)

As tabelas podem ser criadas com as colunas que ache mais adequadas

Ex: utilizadores (username varchar(50), password varchar(50))

Ex: audits (audit varchar(100), data date)



NO JDK

```java
import java.util.Observable;
import java.util.Observer;

class AgenciaNoticias extends Observable {
    private String noticias;

    public void setNoticias(String noticias) {
        this.noticias = noticias;
        setChanged();
        notifyObservers(noticias);
    }
}

class CanalNoticias implements Observer {
    private String noticias;

    @Override
    public void update(Observable o, Object arg) {
        this.noticias = (String) arg;
    }

    public String getNoticias() {
        return noticias;
    }
}
```

```java
public class Exemplo {
    public static void main(String[] args) {
        AgenciaNoticias observable = new AgenciaNoticias();
        CanalNoticias observer = new CanalNoticias();

        observable.addObserver(observer);
        observable.setNoticias("Novo curso de padrões!");

        System.out.println(observer.getNoticias());
    }
}
```

ANTI PATTERNS

Notificações Excessivas: Enviar notificações com muita frequência, o que pode sobrecarregar os observadores e levar a problemas de desempenho.

Falta de Desacoplamento: Manter uma forte dependência entre o sujeito (subject) e os observadores, o que pode dificultar a manutenção e a escalabilidade do sistema.

Observadores Inativos: Não remover observadores que não estão mais interessados nas notificações, resultando em desperdício de recursos.

Notificações Não Filtradas: Enviar notificações para todos os observadores, mesmo quando apenas alguns estão interessados em determinadas mudanças, o que pode ser ineficiente.

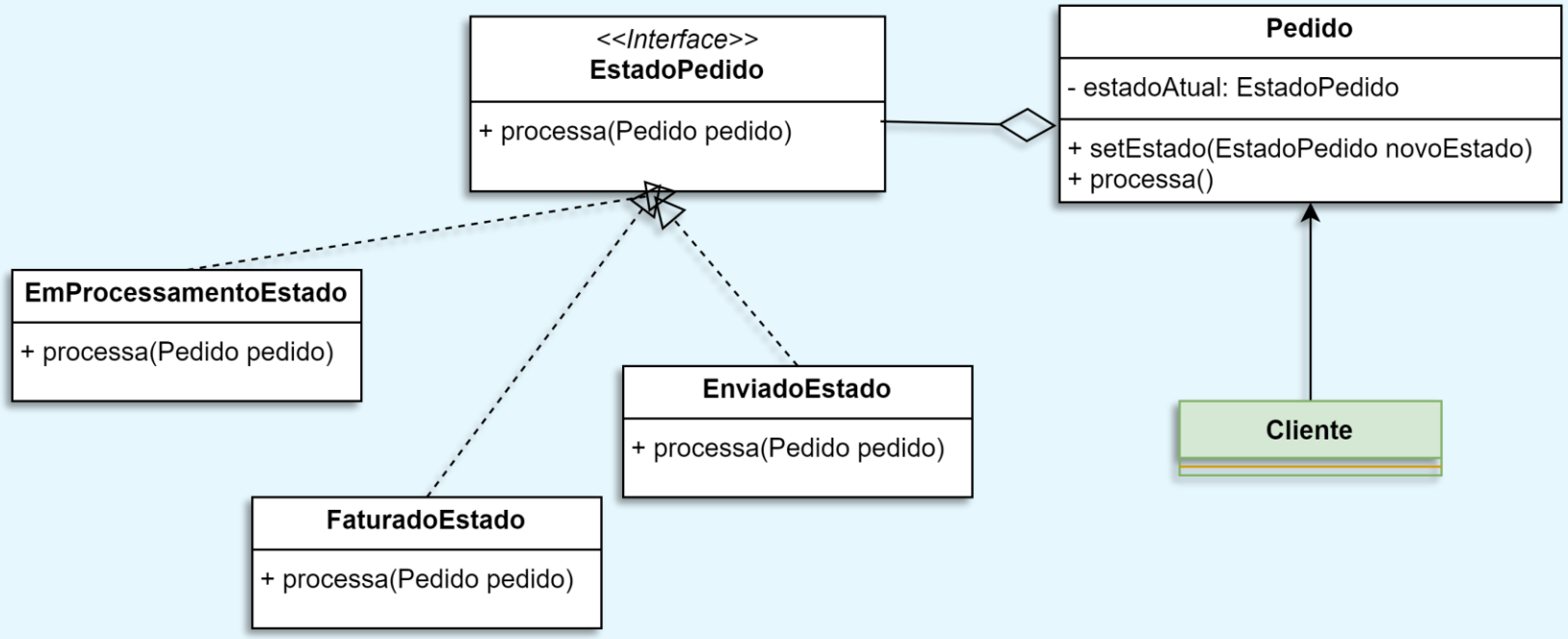
Falta de Sincronização: Não garantir a sincronização adequada ao notificar observadores em um ambiente multithread, o que pode levar a condições de corrida e inconsistências.

Cascata de Notificações: Criar uma cadeia de notificações onde a atualização de um observador desencadeia notificações adicionais, levando a um efeito cascata que pode ser difícil de controlar.

Falta de Documentação: Não documentar adequadamente as interações entre o sujeito e os observadores, dificultando a compreensão e a manutenção do código.

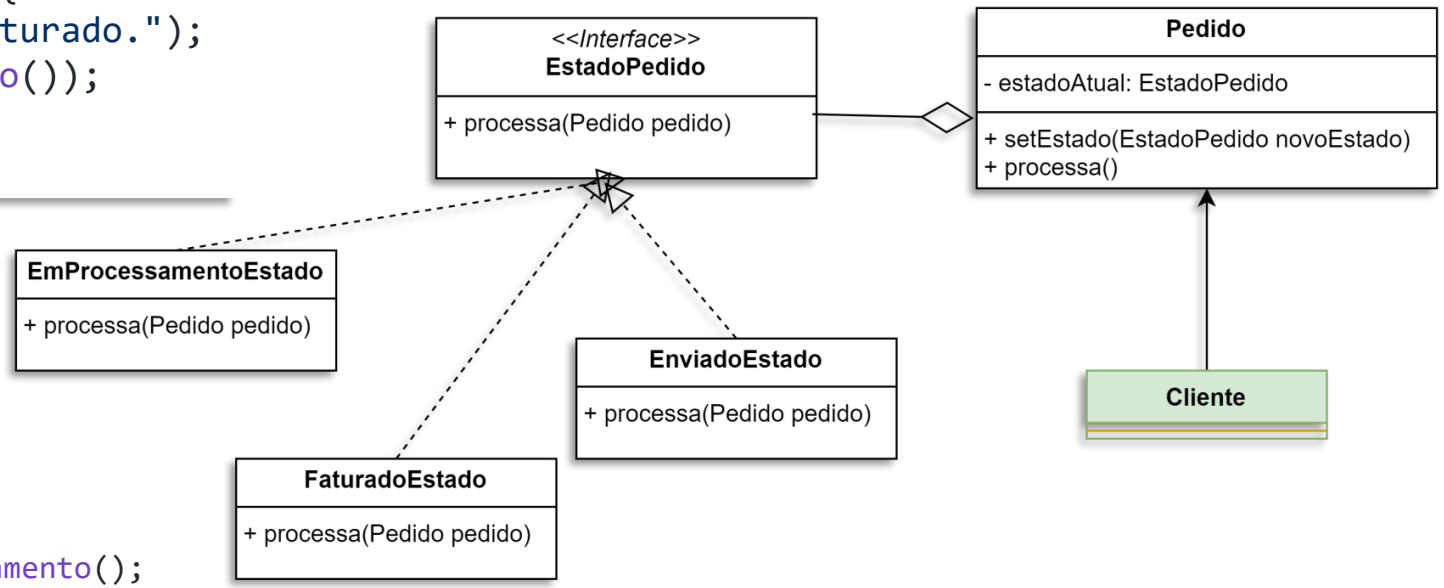
STATE

É um padrão de design de software que permite que um objeto altere seu comportamento quando seu estado interno muda.



STATE

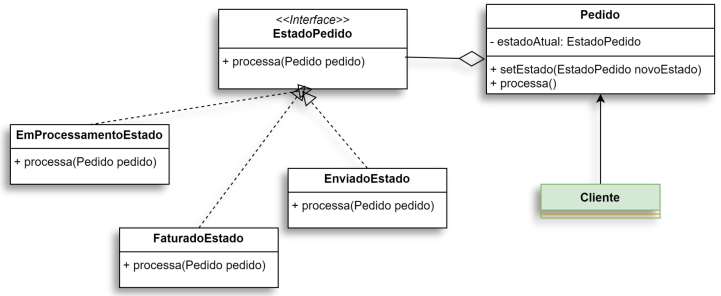
```java
class EstadoFaturado implements EstadoPedido {
    public void processar(Pedido pedido) {
        System.out.println("Pedido foi faturado.");
        pedido.setEstado(new EstadoEnviado());
    }
}
```




```java
class Pedido {
    private EstadoPedido estadoAtual;

    public Pedido() {
        this.estadoAtual = new EstadoEmProcessamento();
    }

    public void setEstado(EstadoPedido novoEstado) {
        this.estadoAtual = novoEstado;
    }

    public void processar() {
        estadoAtual.processar(this);
    }
}
```

```java
public static void main(String[] args) {
    Pedido pedido = new Pedido();
    pedido.processar();
    pedido.processar();
    pedido.processar();
}
```

STATE



Componentes

- **Context**: a classe que pode mudar o seu estado interno
- **State**: a interface que define os métodos para os estados
- **Concrete States**: as classes que são os estados, ao implementar a interface State.

Benefícios

- **Redução de condições**: evita complicações de código com IF, SWITCH.
- **Organização**: O código fica mais organizado e simples de ler
- **Manutenção**: adição e redução de estados torna-se mais simples
- **Encapsulamento**: promove o Single Responsibility Principle. Cada classe tem o seu requisito.

Desvantagens/Desafios

- **Complexidade**: cada estado necessita de uma classe. O aumento de estados e lógica de fluxo pode complicar o código.
- **Manutenção**: o aumento de classes traz desafios de organização das classes.
- **Debug e troubleshooting**: a lógica depende do estado atual e do contexto para decidir o próximo estado, o que pode tornar mais dificil o debug.
- **Memória**: aumento de classes obriga a um consumo de memória.
- **Acoplamento**: várias classes ficam associadas à classe do contexto.

Casos de uso

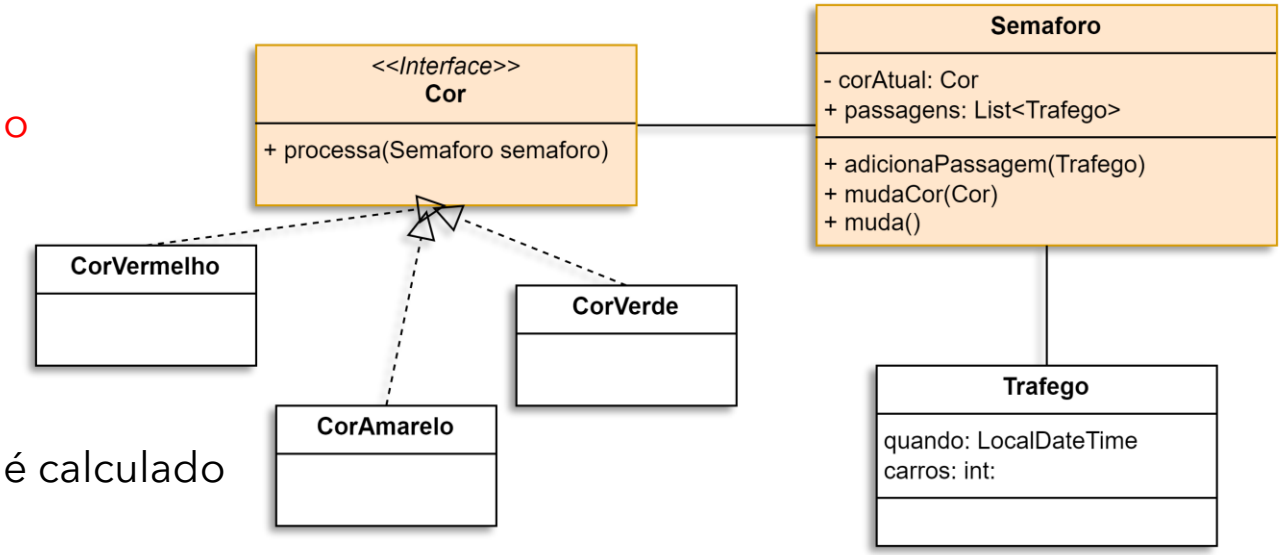
- **Interfaces de utilizador:** quando se necessita de transições entre ecrans, wizards e semelhantes.
- **Processos de workflow:** todo o sistema de workflow baseia-se em estados
- **Protocolos:** para gerir os estados em que ligações estão como: "conectando, conectado, rejeitado"
- **Jogos:** personagens e objetos que podem estar em estados diferentes como: "atacando, defendendo, morto".
- **Controlo de dispositivos:** ligado, desligado, "em espera".
- **Máquinas de estados finitas:** máquinas de vendas, semáforos, jogos, e soluções que dependem de eventos.

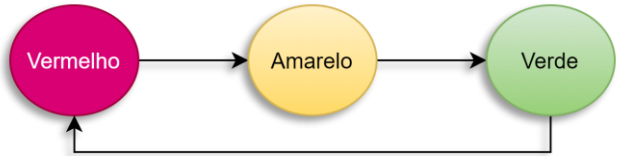
STATE

Actividade: Semáforos (fácil)

Objetivo: usando o padrão State, implemente uma solução que simule os semáforos que se encontram normalmente na rua.

- Existem 3 estados: verde, amarelo e vermelho
  - O estado vermelho demora 1 segundo
  - O estado amarelo 500ms
  - O estado verde 2 segundos.
- Durante o estado verde um numero de 1-10 é calculado
  - Representa o número de carros que passam
  - O número calculado deve ser guardado numa Lista com a data e hora
- Faça a simulação durante 10 iterações e mostre os resultados na consola



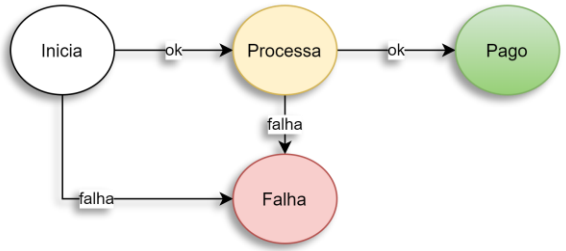


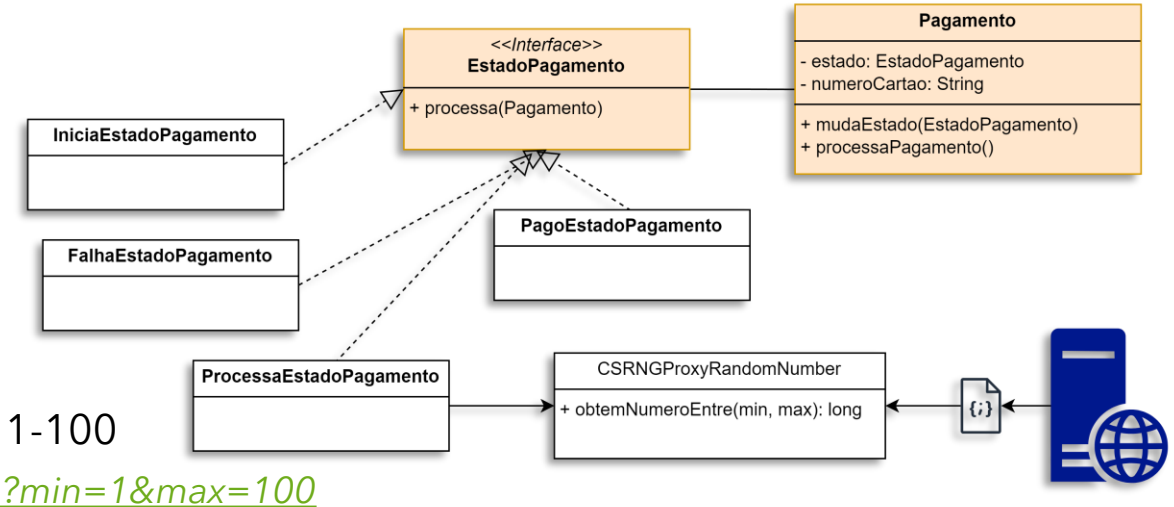
STATE

Actividade: Pagamentos (médio)

Objetivo: usando o padrão State, implemente um checkout de uma encomenda no que respeita ao pagamento.

Os estados são:





Inicia: Pode seguir ou falhar com 50% probabilidade.

Processa: Chama uma API para gerar um número entre 1-100

Se for > 50 tem sucesso: https://csrng.net/csrng/csrng.php?min=1&max=100

Ou falha

Cada estado envia uma mensagem para output com o resultado.

STATE

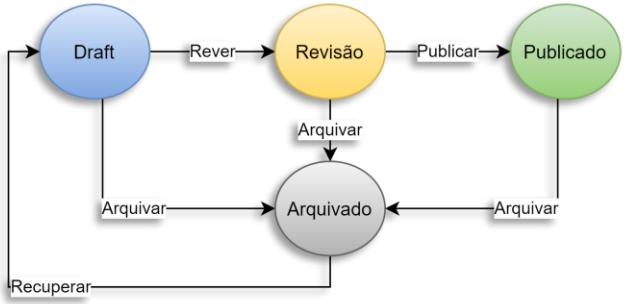
Actividade: Posts e blogs (médio)

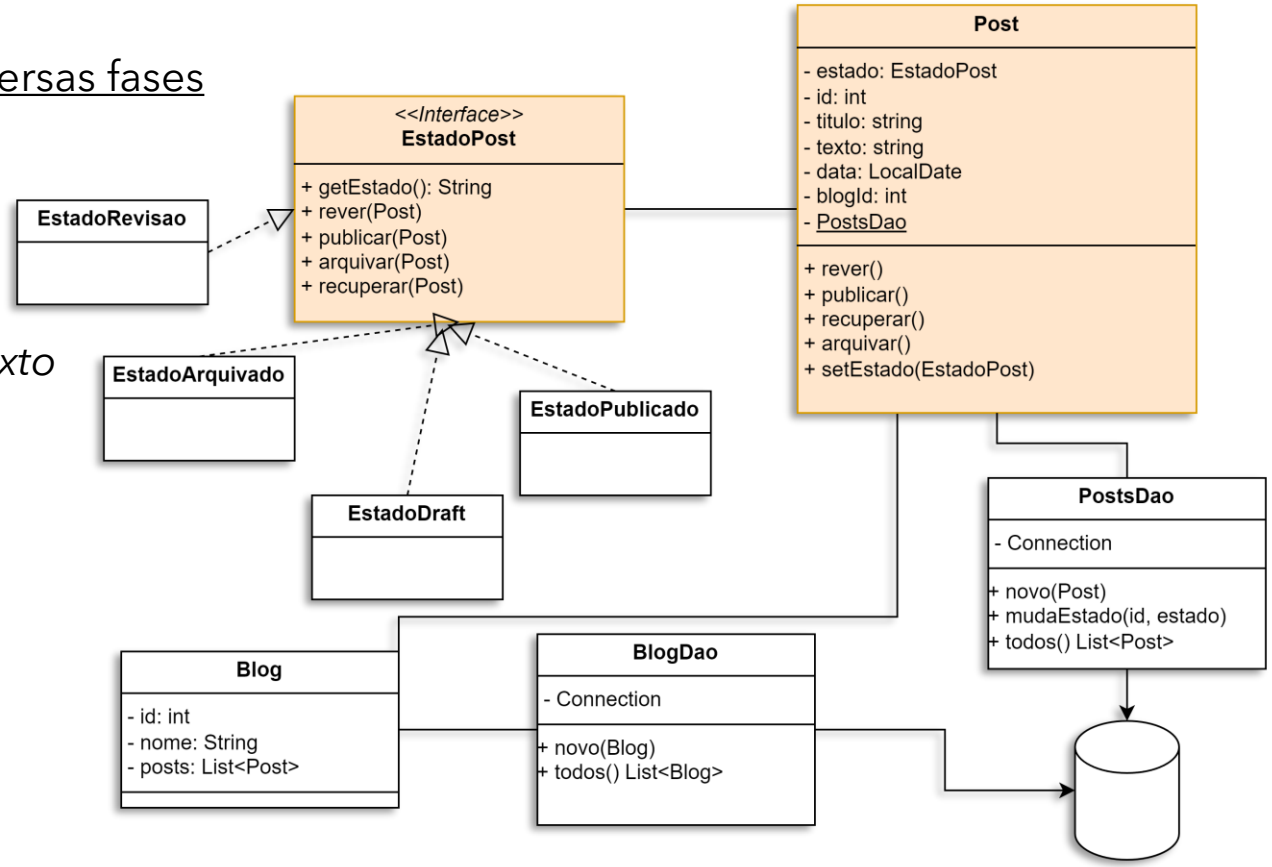
Objetivo: usando o padrão State, implemente as diversas fases para publicar um Post num blog

Use uma tabela de *blogs(id int, nome varchar(100))*

Use uma tabela de *posts(id int, titulo varchar(100), texto varchar(1000), data date, blogId int, estado int)*

Um post pode estar em 4 estados:

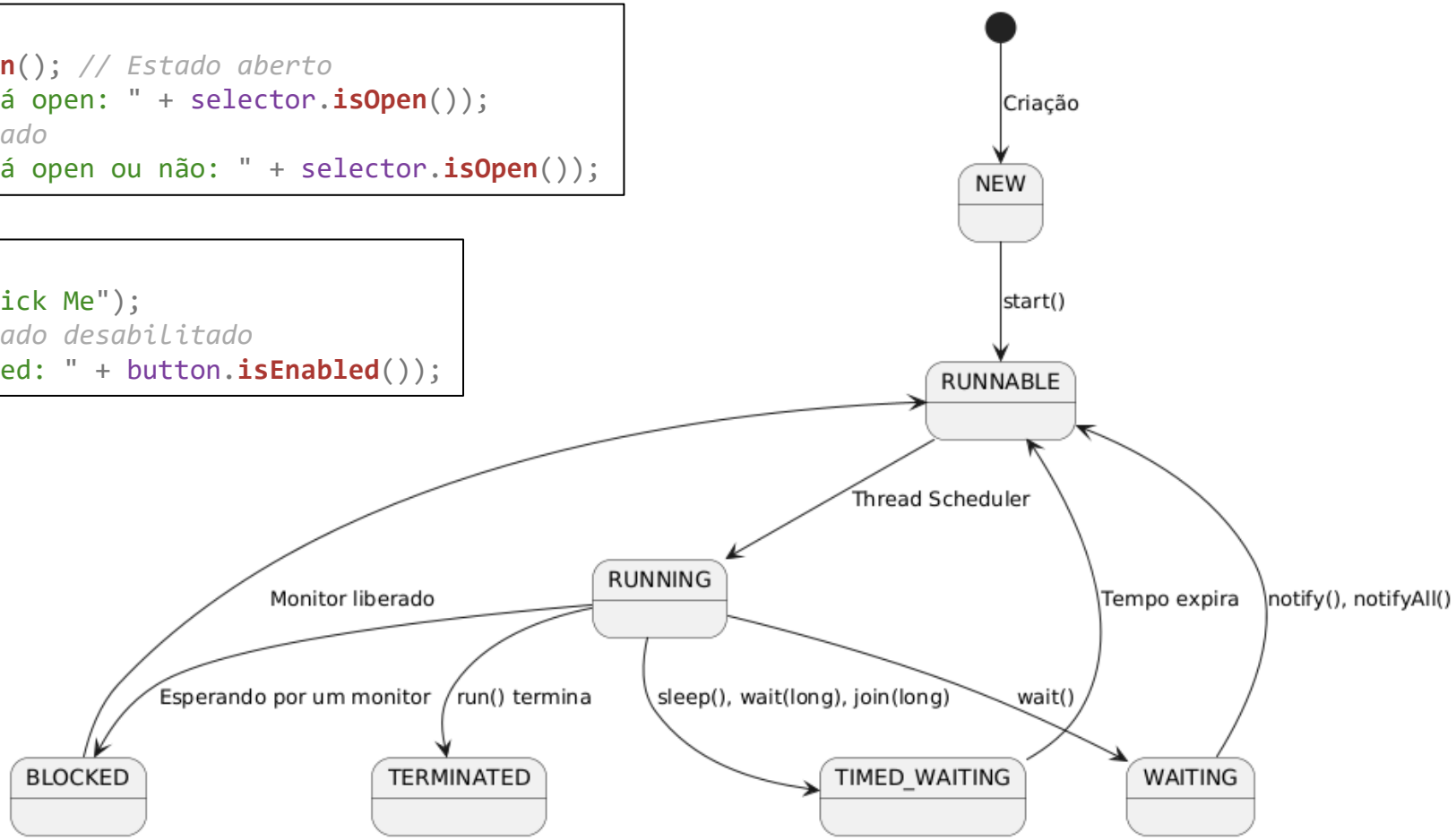




NO JDK

```java
// java.nio.channels
Selector selector = Selector.open(); // Estado aberto
System.out.println("Selector está open: " + selector.isOpen());
selector.close(); // Estado fechado
System.out.println("Selector está open ou não: " + selector.isOpen());
```

```java
// javax.swing
JButton button = new JButton("Click Me");
button.setEnabled(false); // Estado desabilitado
System.out.println("Button enabled: " + button.isEnabled());
```



ANTI PATTERNS

Estado Inflado: Criar estados que realizam muitas tarefas, tornando-os difíceis de entender e manter.

Falta de Encapsulamento: Expor detalhes internos do estado, violando o princípio de encapsulamento.

Estados Dependentes: Criar estados que dependem fortemente de outros estados, quebrando o princípio de responsabilidade única.

Transições Complexas: Implementar transições de estado complexas demais, tornando o fluxo de controle difícil de seguir e depurar.

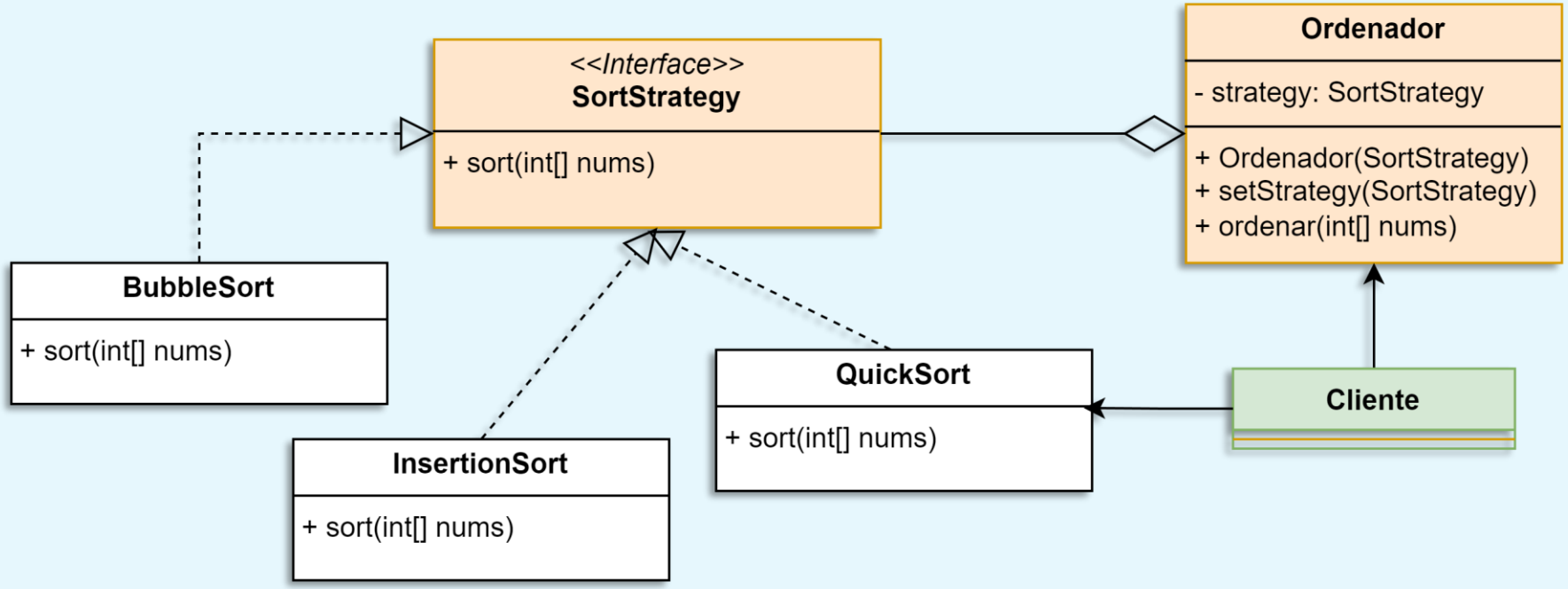
Falta de Documentação: Não documentar adequadamente os estados e suas transições, dificultando a compreensão e a manutenção do código.

Estados Incompletos: Implementar estados que não cobrem todas as operações necessárias, resultando em comportamento inesperado.

Uso Inadequado: Usar o padrão State quando uma simples estrutura condicional seria mais eficiente e fácil de manter.

STRATEGY

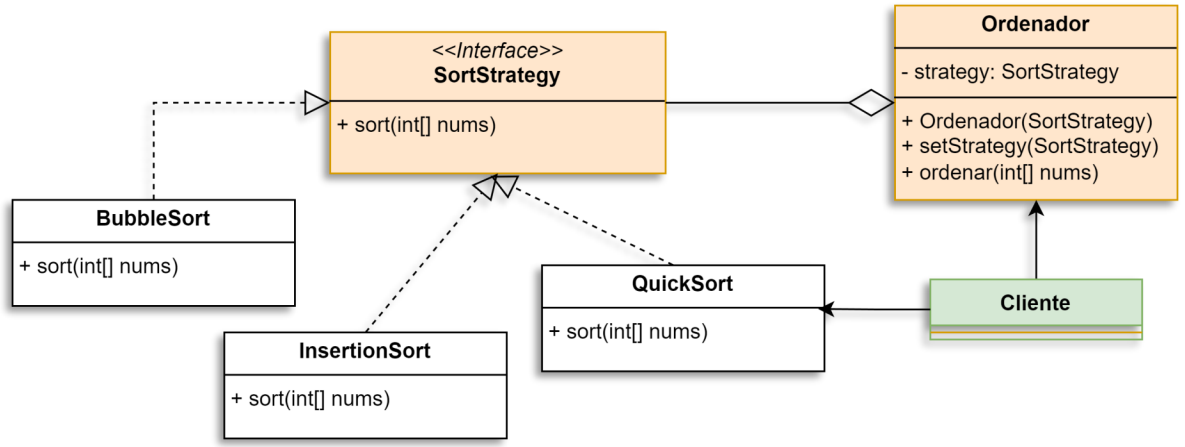
É um padrão quer permite trocar algoritmos sem alterar o objeto que os usa.



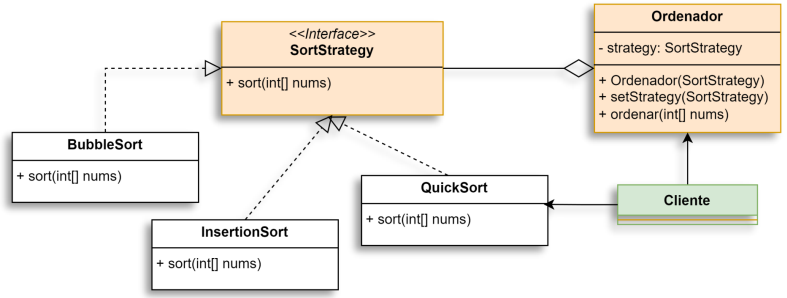
STRATEGY

```java
public static void main(String[] args) {
    int[] array = {5, 2, 9, 1, 5, 6};

    Ordenador ordenador = new Ordenador(new BubbleSort());
    ordenador.ordenar(array);
    System.out.println(Arrays.toString(array));

    ordenador.setStrategy(new InsertionSort());
    ordenador.ordenar(array);
    System.out.println(Arrays.toString(array));

    ordenador.setStrategy(new QuickSort());
    ordenador.ordenar(array);
    System.out.println(Arrays.toString(array));
}
```

```java
    private SortStrategy strategy;

    public Ordenador(SortStrategy strategy) {
        this.strategy = strategy;
    }

    public void setStrategy(SortStrategy strategy) {
        this.strategy = strategy;
    }

    public void ordenar(int[] array) {
        strategy.sort(array);
    }
}
```



STRATEGY



Componentes

- **Context**: a classe que usa os algoritmos
- **Strategy**: a interface que define os métodos para a definição dos algoritumos
- **Concrete Strategy**: as classes que implementam os algoritmos

Benefícios

- **Flexibilidade e extensibilidade**: facilita a alteração dos algoritmos em tempo de execução.
- **Reutilização**: cada classe tem o seu algoritmo, promovendo a partilha de lógica
- **Princípio Aberto/Fechado**: as classes são abertas para extensão e fechadas para modificação. Pode-se adicionar novos algoritmos sem alterar o código existente.
- **Redução de condições**: IF e SWITCH são reduzidos.
- **Isolamento**: cada algoritmo está na sua classe

Desvantagens/Desafios

- **Sobrecarga de comunicação**: algumas estratégias podem não necessitar de informações que outras necessitam, criando algum código redundante
- **Aumento de código**: cada estratégia tem pelo menos uma classe associada e a sua gestão no projeto.
- Exige programadores com algum **conhecimento de engenharia** de código.

Casos de uso

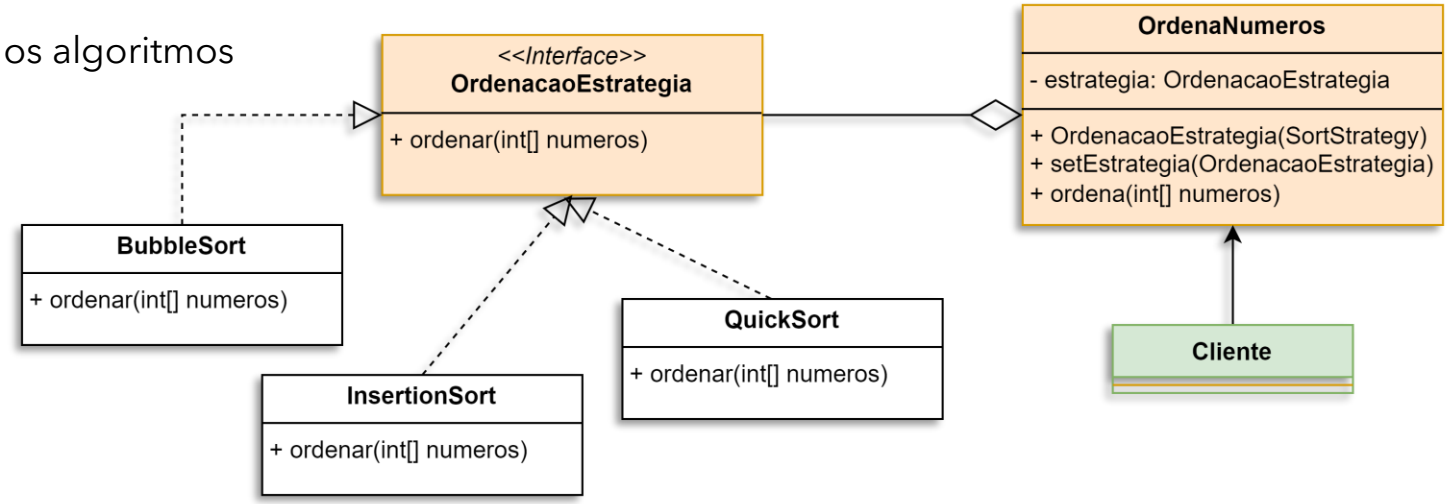
- **Cálculos de preços:** varios tipos de cálculos conforme produtos, clientes, e outros critérios
- **Ordenações e pesquisas:** escolha dos algoritmos adequados a cada situação
- **Formatação de dados:** xml, csv, yaml, muito populares em APIs, configurações, etc.
- **Autenticação e autorização:** possibilidade de escolher o método adequado
- **Algoritmos de compressão:** escolha do algoritumo
- **Inteligência artificial e ML**
- **Jogos:** estratégias para personagens ou objetos diferentes
- **Processamento de pagamentos:** cartão, dinheiro, paypal, etc.

STRATEGY

Actividade: algoritmos de ordenação (fácil-médio)

Objetivo: usando o padrão Strategy, crie uma aplicação que permite escolher entre 3 algoritmos de ordenação de inteiros.

- BubbleSort, InsertSort, QuickSort
  - Fácil, ou médio se tentar implementar os algoritmos
- Crie um array de 30000 números (*ints*)
- Imprima o tempo de execução



STRATEGY

Actividade: descontos encomendas (fácil)

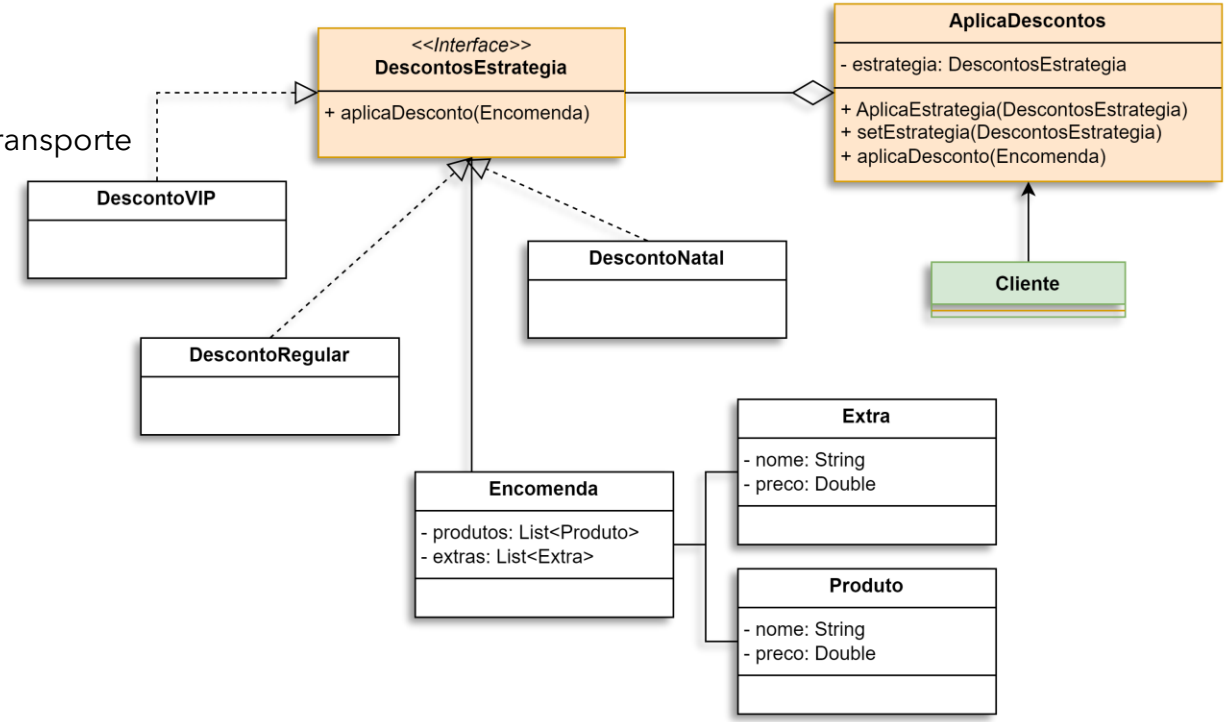
Objetivo: usando o padrão Strategy, aplique 3 tipos de descontos conforme os critérios:

VIP: Não pagam transporte + 5% desconto total

Regular: Não tem descontos, para 10€ transporte

Natal: Mais de 2 produtos, paga menos 10% no mais barato, 10€ transporte

- Usar uma classe **Encomenda**
  - Lista de produtos.
  - Item de ***extras*** *(nome, valor)*
    - *desconto*
    - *transporte*
- Usar class **Produto** (nome, preco)



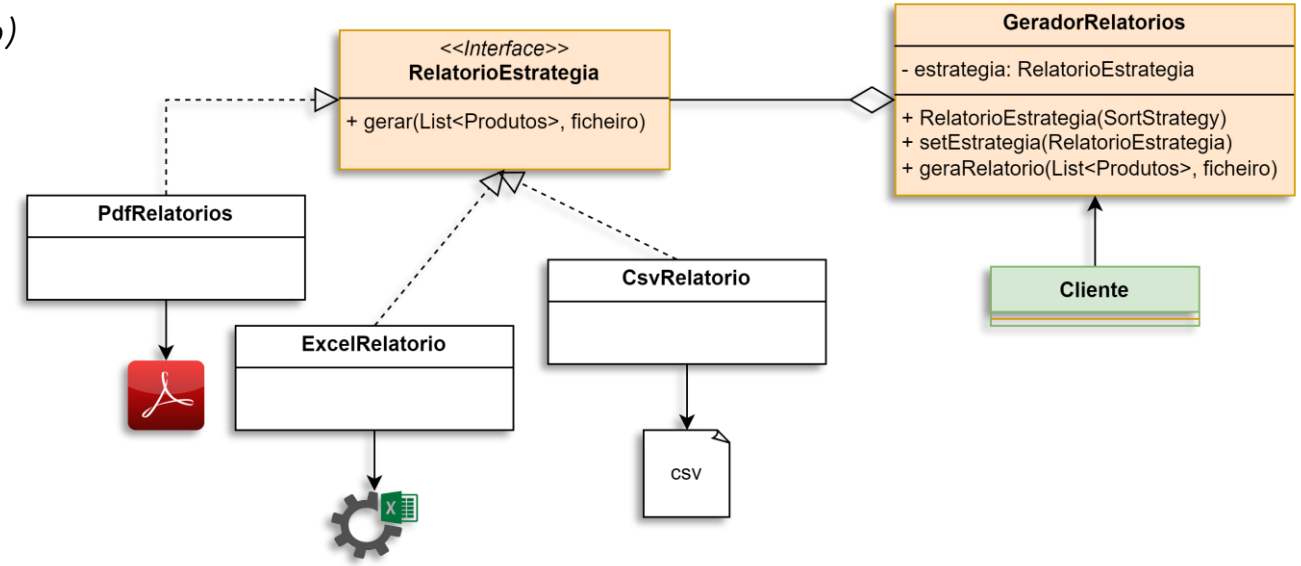
STRATEGY

Actividade: geração de relatórios (avançado)

Objetivo: usando o padrão Strategy, a partir de uma lista de produtos, gerar um ficheiro num dos seguintes formatos: pdf, excel e csv

Bibliotecas: *openpdf, poi-poxml, apache-commons-csv (precisa do commons-io)*

Produtos *(id, nome, preco)*



NO JDK

```java
// java.util.Comparator
// Estratégia de comparação por ordem alfabética
Collections.sort(names, new Comparator<String>() {
    @Override
    public int compare(String o1, String o2) {
        return o1.compareTo(o2);
    }
});
System.out.println("Ordem alfabética: " + names);

// Estratégia de comparação por ordem inversa
Collections.sort(names, new Comparator<String>() {
    @Override
    public int compare(String o1, String o2) {
        return o2.compareTo(o1);
    }
});
System.out.println("Ordem inversa: " + names);
```

```java
// javax.crypto
KeyGenerator keyGen = KeyGenerator.getInstance("AES");
SecretKey secretKey = keyGen.generateKey();

Cipher cipher = Cipher.getInstance("AES");
cipher.init(Cipher.ENCRYPT_MODE, secretKey);

String plainText = "Hello, World!";
byte[] encryptedText = cipher.doFinal(plainText.getBytes());
System.out.println("Texto criptografado: " + new String(encryptedText));
```

```java
// java.util.zip
String inputString = "Hello, World!";
byte[] input = inputString.getBytes("UTF-8");

// Compressão
byte[] output = new byte[100];
Deflater deflater = new Deflater();
deflater.setInput(input);
deflater.finish();
int compressedDataLength = deflater.deflate(output);
System.out.println("Dados comprimidos: " + new String(output, 0,
compressedDataLength));

// Descompressão
Inflater inflater = new Inflater();
inflater.setInput(output, 0, compressedDataLength);
byte[] result = new byte[100];
int resultLength = inflater.inflate(result);
inflater.end();
System.out.println("Dados descomprimidos: " +
            new String(result, 0, resultLength, "UTF-8"));
```

ANTI PATTERNS

Estratégias Infladas: Criar estratégias que realizam muitas tarefas, tornando-as difíceis de entender e manter.

Falta de Encapsulamento: Expor detalhes internos das estratégias, violando o princípio de encapsulamento.

Estratégias Dependentes: Criar estratégias que dependem fortemente de outras estratégias ou do contexto, quebrando o princípio de responsabilidade única.

Uso Excessivo de Estratégias: Criar estratégias para tarefas triviais, aumentando a complexidade desnecessariamente.

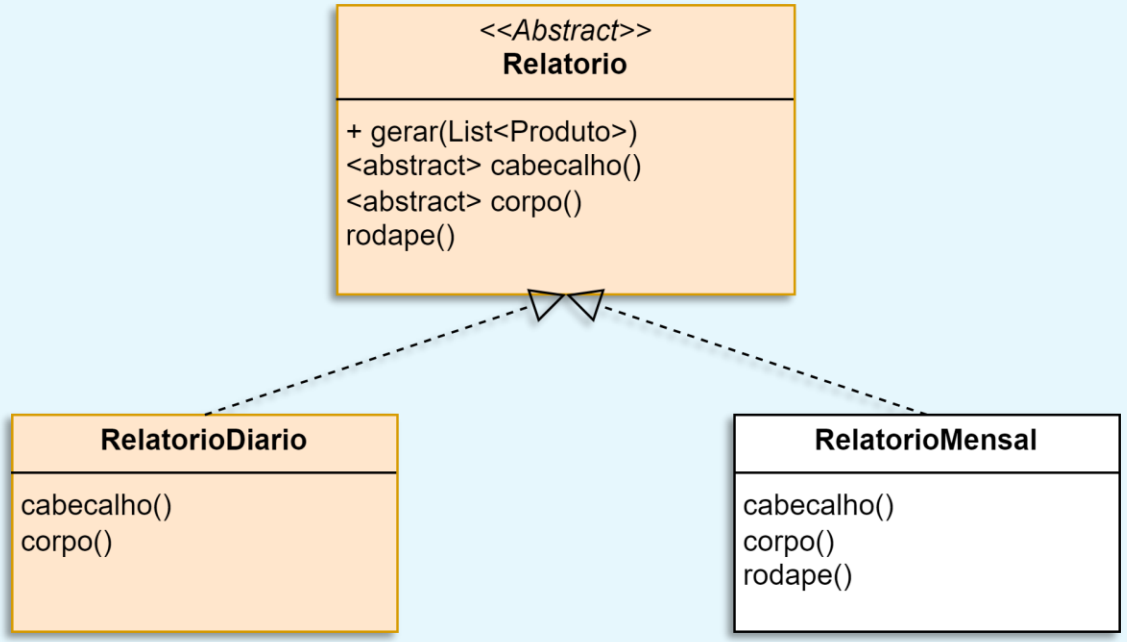
Falta de Documentação: Não documentar adequadamente as estratégias e seus usos, dificultando a compreensão e a manutenção do código.

Estratégias Incompletas: Implementar estratégias que não cobrem todos os casos de uso necessários, resultando em comportamento inesperado.

Falta de Flexibilidade: Não permitir a troca dinâmica de estratégias, limitando a flexibilidade do sistema.

TEMPLATE METHOD

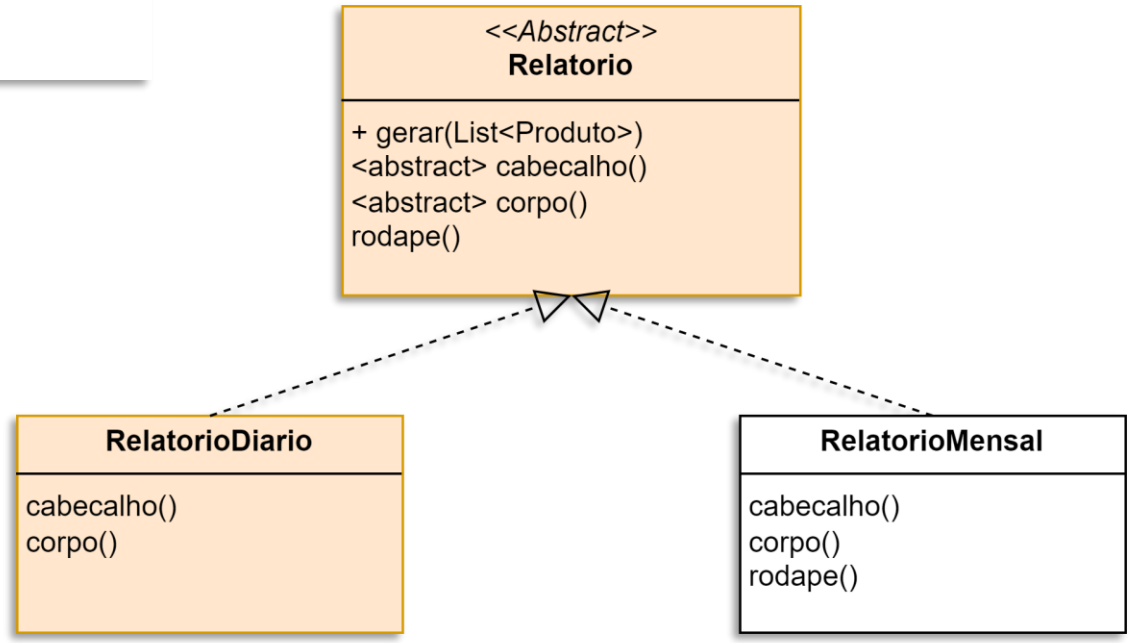
É um padrão que define o esqueleto de um algoritmo em uma operação, deixando alguns passos para serem implementados pelas subclasses.



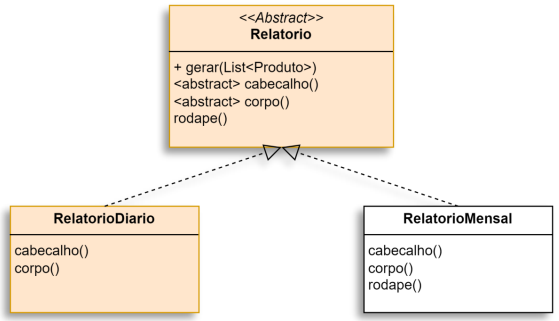
TEMPLATE METHOD

```
public static void main(String[] args) {
    Relatorio relatorioDiario = new RelatorioDiario();
    relatorioDiario.gerar();

    Relatorio relatorioMensal = new RelatorioMensal();
    relatorioMensal.gerar();
}
```

```
public String gerar() {

    geraHeaderHTML(builder);

    cabecalho(builder);
    corpo(builder);
    rodape(builder);

    geraFooterHTML(builder);

    return builder.toString();
}
```



TEMPLATE METHOD



Componentes

- **Classe abstrata**: define o template, onde esta o esqueleto do algoritmo
- **Métodos abstratos**: métodos na classe abstracta mas que serão implementados em subclasses
- **Subclasses concretas**: implementações dos métodos abstratos, que formam o esqueleto do algoritmo

Benefícios

- **Consistência**: a estrutura do algoritmo permanece sempre intacta, independente da implementação das subclasses.
- **Extensibilidade**: permite adicionar ou alterar comportamentos, implementando subclasses (override)
- **Controle sobre a sequência de execução**: a classe abstracta define o fluxo, os passos do algoritmo a implementar.

Desvantagens/Desafios

- **Rigidez**: se houver necessidade de variantes ao algoritumo tem de se criar outra class abstracta sem reaproveitamento das subclasses, ou alternativamente "complicar" o algoritumo trazendo mais **complexidade**.
- **Dependência da classe base**: coupling muito forte com as classes base, no que respeita a funcionalidades.
- **Heranças longas**: hoje em dia prefere-se composição sobre herança. As heranças longas tornam a gestão do código mais dificil.

Casos de uso

- **Geração de relatórios:** quando os relatórios têm uma estrutura fixa, mas os detalhes podem variar. ex: dados de fontes diferentes, ou reaproveitamento de um modelo para outro tipo de relatórios.
- **Sistemas de workflow:** quando implementado em código, com um algoritmo bem definido acerca dos passos a seguir
- **Construção de objetos complexos:** quando a construção de objetos complexos necessitam de um conjunto de passos, e o padrão Builder não é suficiente.

TEMPLATE METHOD

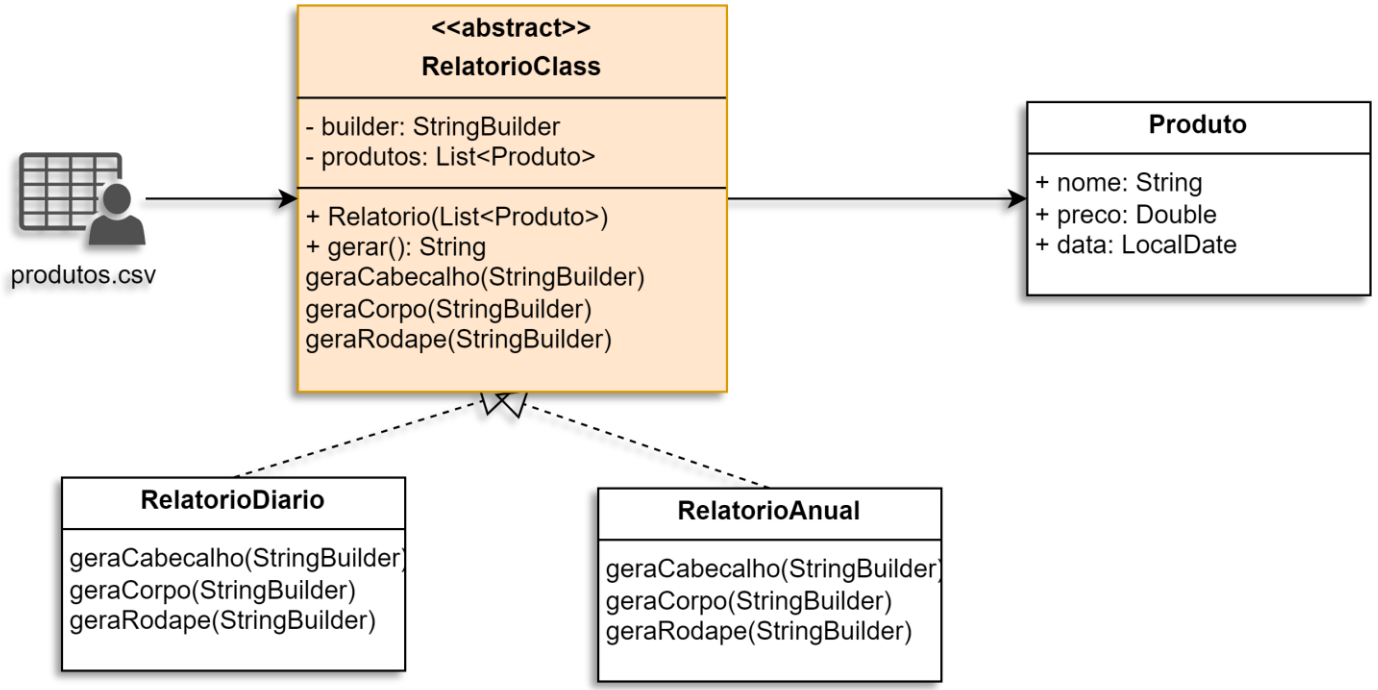
Actividade: Relatórios produtos (fácil-médio)

Objetivo: usando o padrão Template Method, implemente uma forma de produzir 2 tipos de relatórios para uma lista de produtos fornecida.

- Os relatórios têm todos: cabeçalho, corpo e rodapé
  - O cabeçalho tem o título
  - O corpo tem os dados estatísticos ou listagem
  - O rodapé tem o total
- Relatório diário (2024-01-01)
  - Corpo: Lista os produtos com a data do dia
  - Rodapé: O total dos preços
- Relatório Anual
  - Corpo: as somas por mês
  - Rodapé: O total dos meses

Fácil: usar System.out para gerar os relatórios

Médio: Usar HTML como output

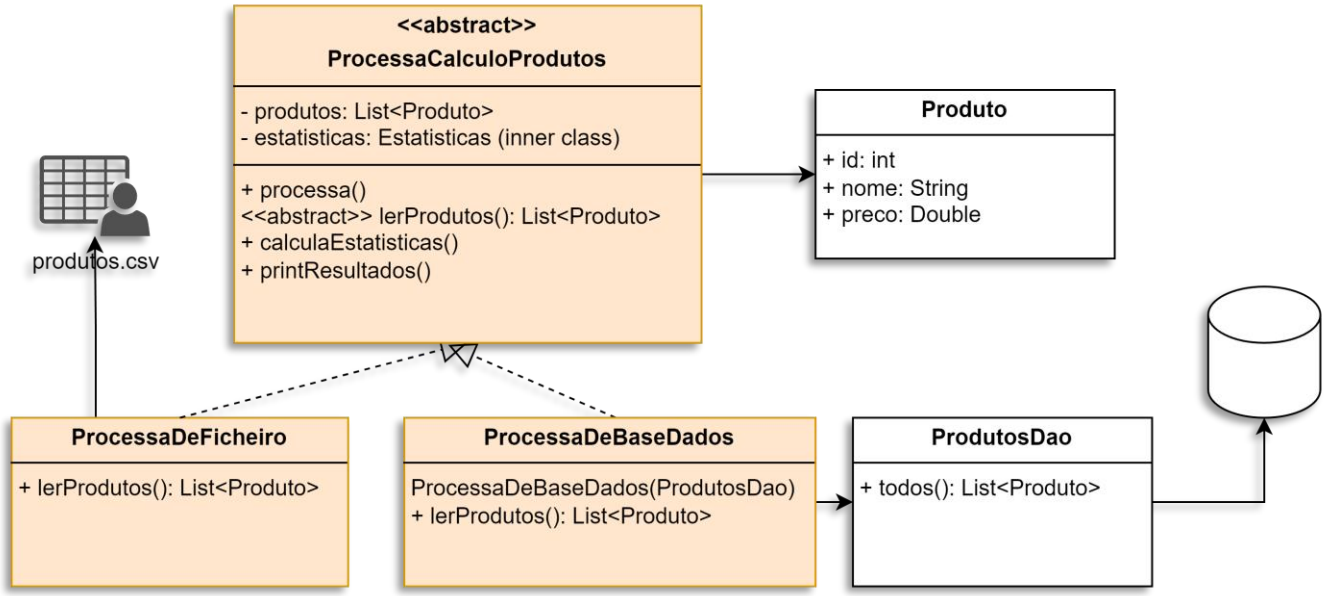


TEMPLATE METHOD

Actividade: Calcular estatísticas (médio)

Objetivo: usando o padrão Template Method, calcule estatísticas dos preços de produtos que podem ser lidos de um ficheiro CSV e de uma tabela "produtos".

- *Produto (id, nome, preco)*
- Os passos que devem ser seguidos são:
  1. Ler Produtos (*abstract*) – certamente override
  2. Calcula Estatisticas - *soma|media|mínimo|máximo*
  3. Escrever resultados para consola

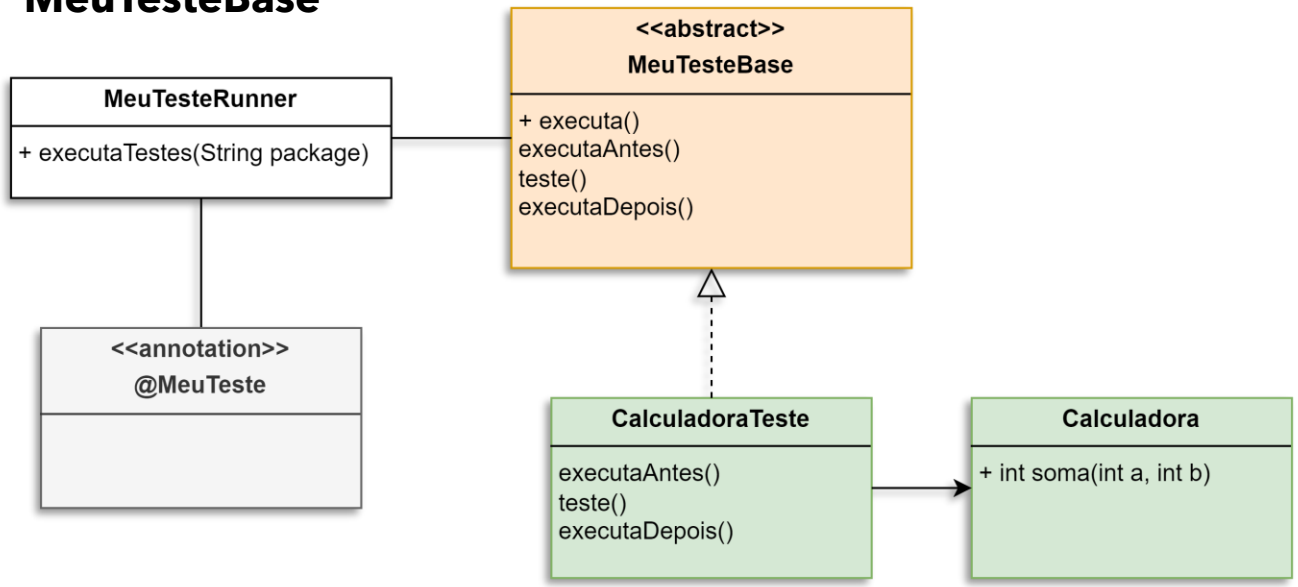


TEMPLATE METHOD

Actividade: Framework Testes (avançado)

Objetivo: usando o padrão Template Method, implemente uma classe que pode ser usada para o inicio de um desenvolvimento de uma framework de testes

- Cada teste é uma classe que deverá herdar do "**MeuTesteBase**"
- Um teste pode conter 2 métodos:
  - um método que é executado antes
  - Um método que é executado depois
- É necessário desenvolver um **MeuTesteRunner**
  - Procura classes que sejam testes (anotação: **@MeuTeste**)
  - Executa o método "executa" (*antes()→teste()→depois()*)
- DICA1: Requer criar a anotação **@MeuTeste**
- DICA2: ***Classgraph*** pode ser uma solução para procurar as classes anotadas (https://github.com/classgraph/classgraph)



NO JDK

```java
// java.util.AbstractList - add, remove e get são oferecidos
public class MinhaLista extends AbstractList<String> {
    private String[] elementos;

    public MinhaLista(String[] elementos) {
        this.elementos = elementos;
    }

    @Override
    public String get(int index) {
        return elementos[index];
    }

    @Override
    public int size() {
        return elementos.length;
    }

    public static void main(String[] args) {
        String[] dados = {"A", "B", "C"};
        List<String> lista = new MinhaLista(dados);
    }
}
```

```java
// java.io.InputStream - read tem lógica mais não define onde lê
public class MeuInputStream extends InputStream {
    private ByteArrayInputStream byteArrayInputStream;

    public MeuInputStream(byte[] dados) {
        this.byteArrayInputStream = new ByteArrayInputStream(dados);
    }

    @Override
    public int read() throws IOException {
        return byteArrayInputStream.read();
    }

    public static void main(String[] args) throws IOException {
        byte[] dados = {65, 66, 67}; // ASCII para A, B, C
        MeuInputStream meuInputStream = new MeuInputStream(dados);
        int dado;
        while ((dado = meuInputStream.read()) != -1) {
            System.out.print((char) dado); // Output: ABC
        }
    }
}
```

ANTI PATTERNS

Métodos Template Rígidos: Criar métodos template que são difíceis de modificar ou estender, limitando a flexibilidade do código.

Falta de Flexibilidade: Não permitir que subclasses substituam métodos hook, resultando em uma implementação rígida e difícil de personalizar.

Violação do Princípio de Responsabilidade Única: Colocar demasiada lógica no método template, violando o princípio de responsabilidade única e tornando o código difícil de manter.

Dependência de Subclasses: Forçar subclasses a implementar muitos métodos abstratos, o que pode levar a uma implementação complexa e difícil de gerenciar.

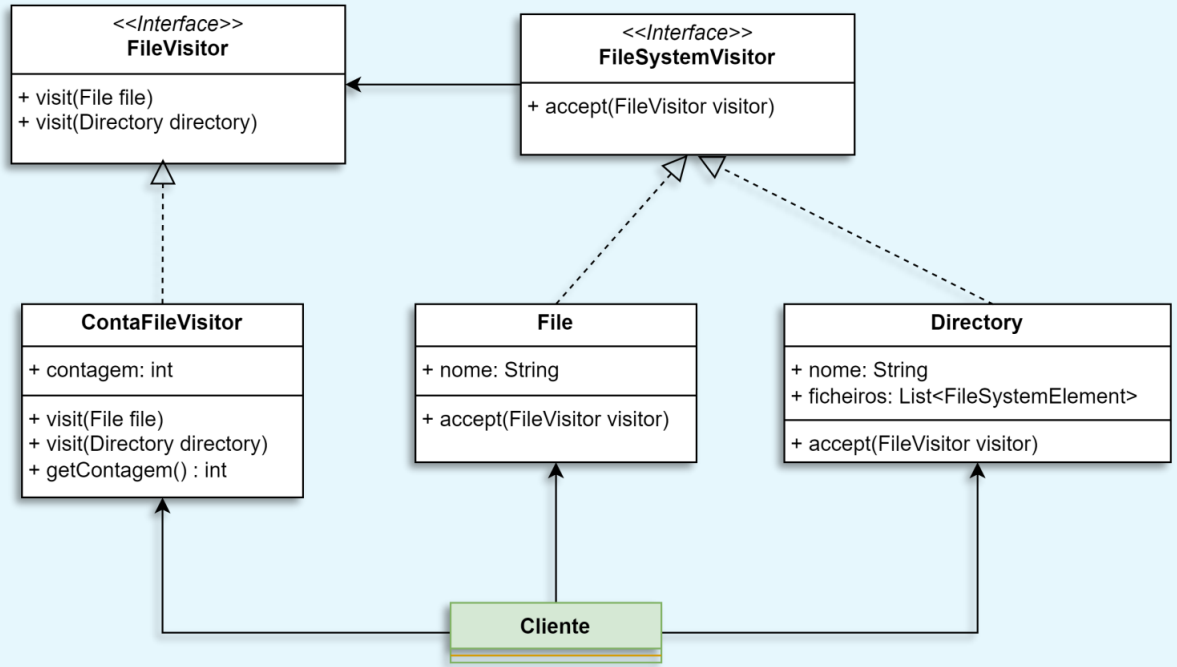
Métodos Template Monolíticos: Criar métodos template que realizam muitas tarefas diferentes, tornando-os difíceis de entender e manter.

Falta de Documentação: Não documentar adequadamente os métodos template e os métodos hook, dificultando a compreensão e a manutenção do código.

Uso Inadequado: Usar o padrão Template Method quando uma simples herança ou composição seria mais adequada, resultando em uma complexidade desnecessária.

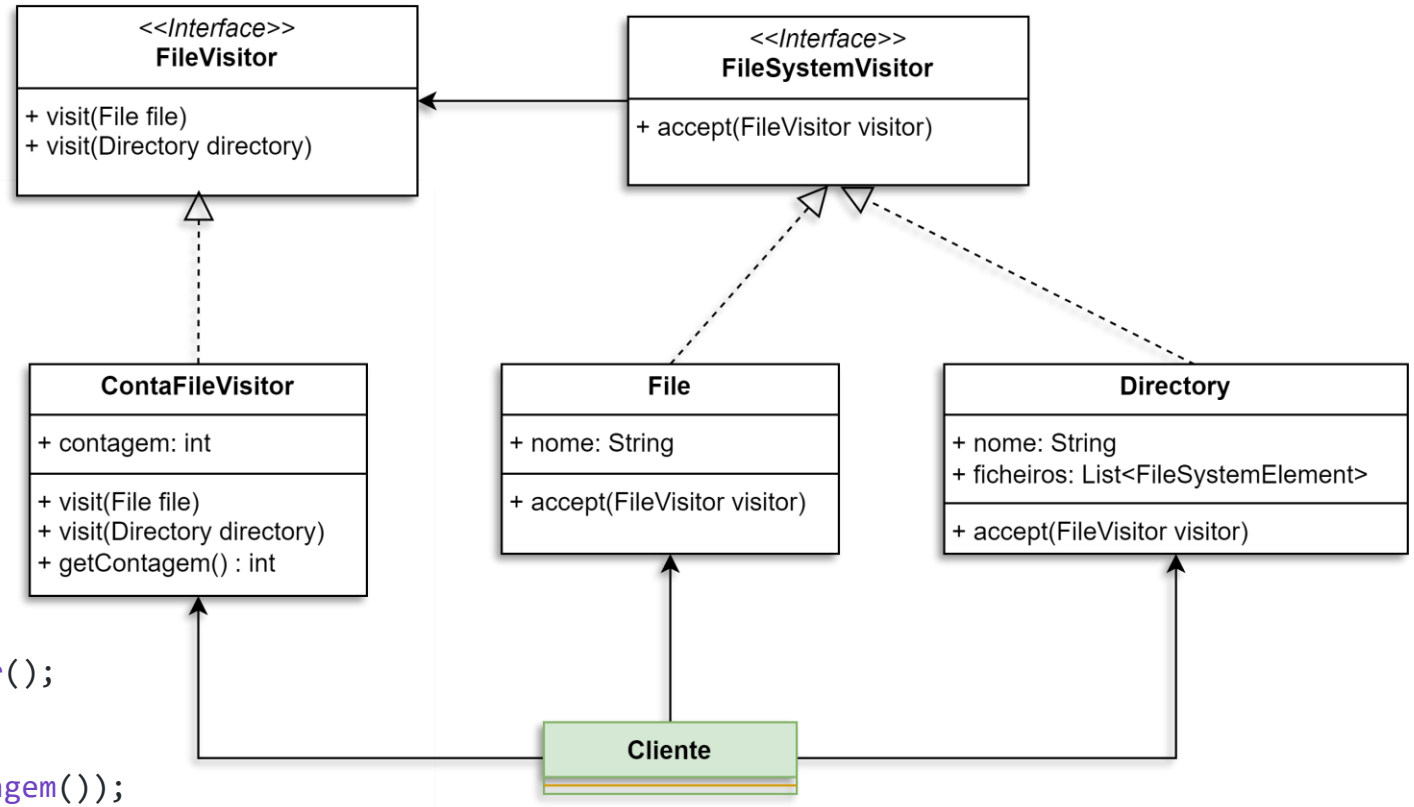
VISITOR

É um padrão que permite adicionar novas operações a uma estrutura de objetos sem modificar as classes dos objetos sobre os quais opera. Ele separa o algoritmo da estrutura do objeto, facilitando a adição de novas funcionalidades para serem implementados pelas subclasses.

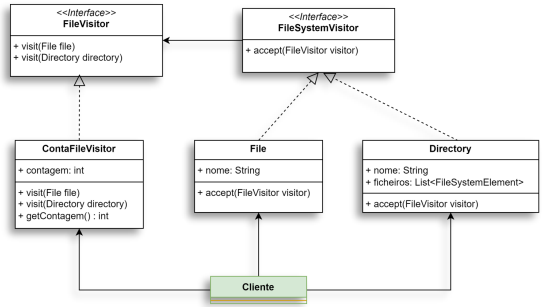


VISITOR

```java
public static void main(String[] args) {
        Directory root = new Directory("root");
        File file1 = new File("file1.txt");
        File file2 = new File("file2.txt");
        Directory subDir = new Directory("subDir");
        File file3 = new File("file3.txt");

        root.addElement(file1);
        root.addElement(file2);
        root.addElement(subDir);
        subDir.addElement(file3);

        ContaFileVisitor conta = new ContaFileVisitor();
        root.accept(conta);

        System.out.println("Total: " + conta.getContagem());
    }
```



VISITOR



Componentes

- **Visitor**: uma interface ou classe abstracta que define os métodos de visita para cada tipo de Element.
- **ConcreteVisitor**: a implementação do Visitor
- **Element**: uma interface ou classe abstrata que contém o método **accept**, que aceita um visitante.
- **ConcreteElement**: implementa o método **accept**, que chama o método do visitante.

Benefícios

- **Extensabilidade**: permite adicionar operações sem alterar as classes.
- **Princípio Aberto/Fechado**: classes abertas para extensão, e fechadas para modificação.
- **Princípio Single Responsibility Principle**: cada classe visitada fornece a informação que é necessária.
- **Acumulação**: permite acumular informações de múltiplas fontes (visitados)

Desvantagens/Desafios

- **Complexidade**: o código fica mais complexo, quando o número de elementos cresce.
- **Acesso**: os visitantes podem não ter acesso a membros privados dos elementos, obrigando por vezes a "abrir" um pouco o encapsulamento.
- **Dependência**: Mudanças na estrutura podem exigir alterações significativas nos visitantes.
- **Implementação**: exige um perfil um pouco mais avançado que um junior.

Casos de uso

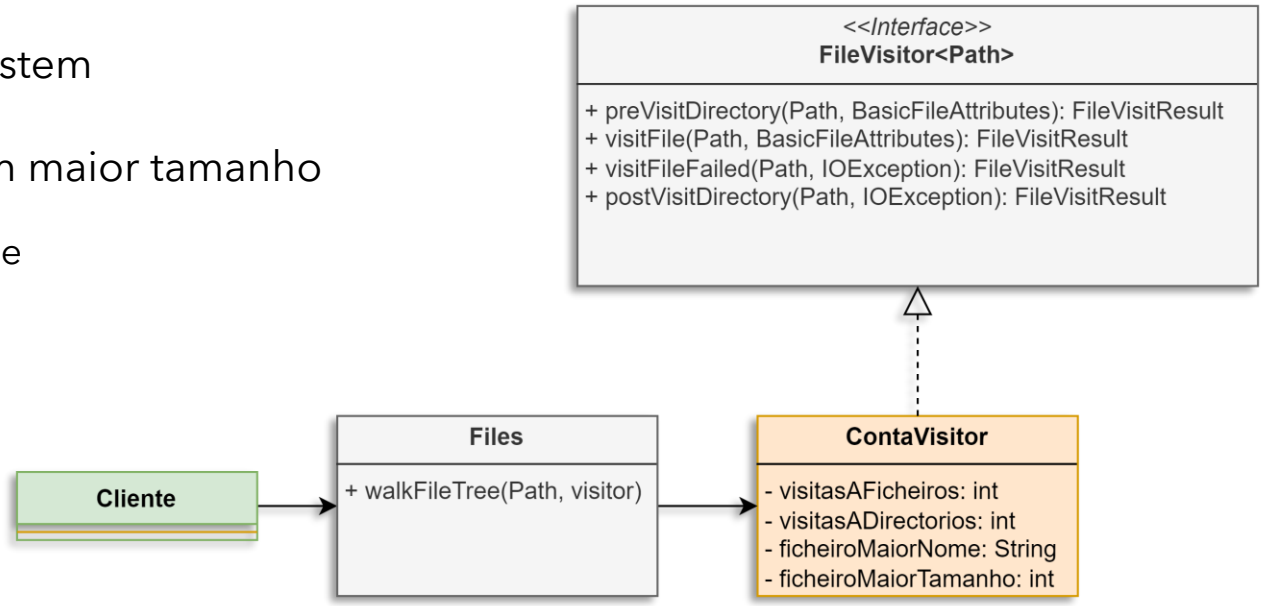
- **Compiladores:** usado para realizar operações nas árvores criadas.
- **Sistemas de arquivos:** percorrer estruturas de ficheiros, e processar informações sobre ficheiros, como contas, somar, procurar, etc.
- **Processamento de documentos:** permite formatar, exportar, transformar relatórios, documentos, e outros documentos com estruturas em árvore.
- **Análise de dados:** ajuda a processar dados em formatos de grafos e árvores.

VISITOR

Actividade: Usar o FileVisitor do Java (fácil)

Objetivo: usando o padrão Visitor, conte o número de Pastas e Ficheiros no disco ou pasta à escolha.

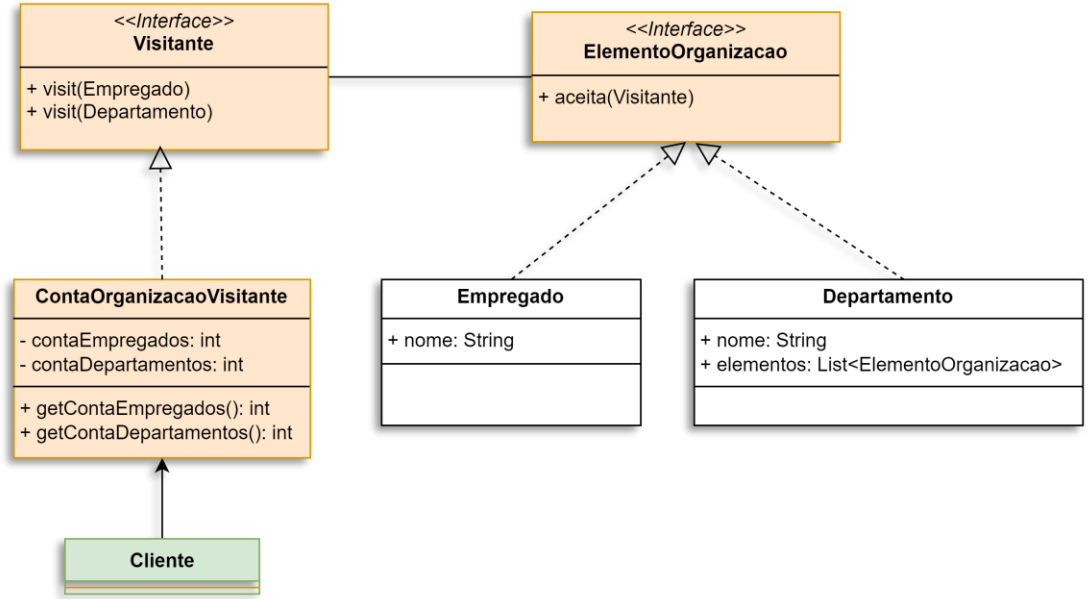
- Quantos ficheiros existem
- Quantas pastas existem
- Qual o ficheiro com maior tamanho
  - Qual o seu nome



VISITOR

Actividade: Organigrama e empregados (médio)

Objetivo: usando o padrão Visitor crie um organigrama com departamentos e empregados, e conte o número total de empregados na organização e o número de departamentos.

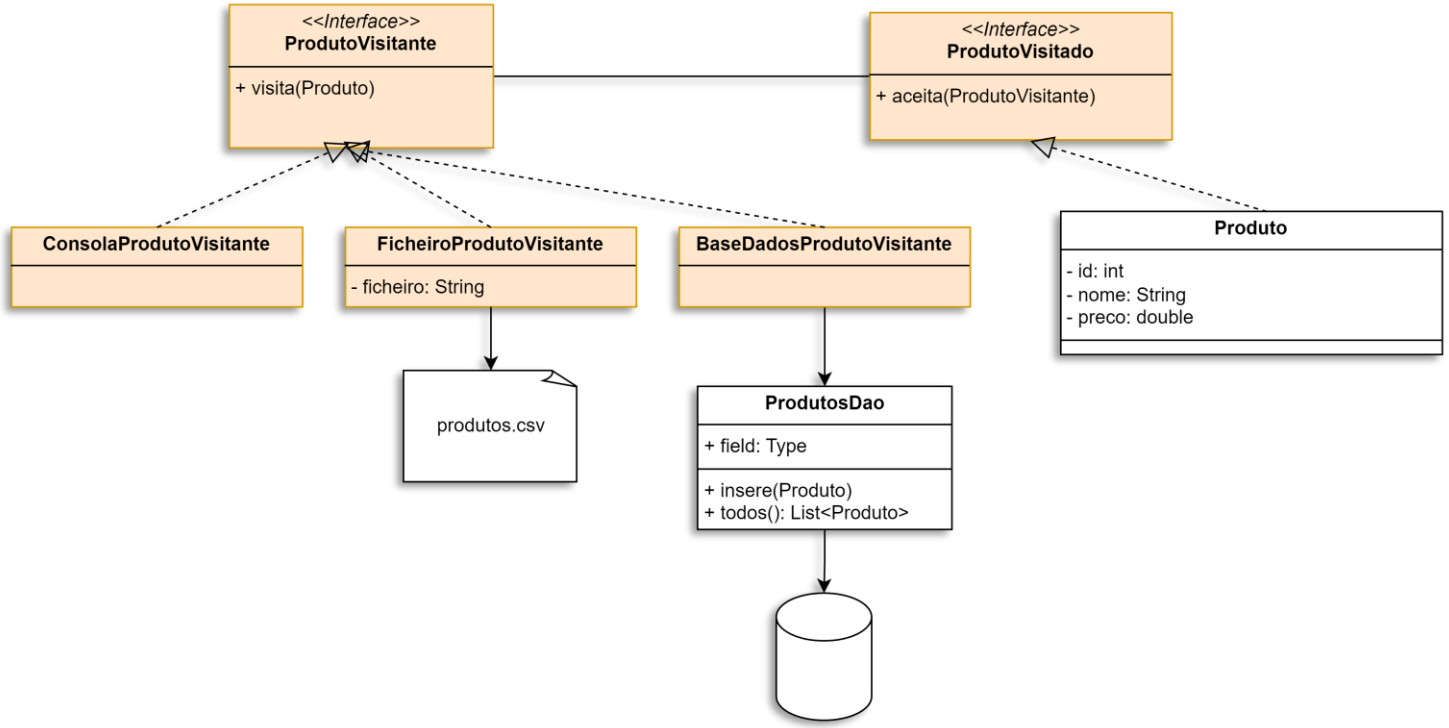


VISITOR

Actividade: escreve para múltiplos outputs (avançado)

Objetivo: usando o padrão Visitor, implemente um sistema que permite enviar produtos para base de dados, ficheiro e consola.

- Produto (id, nome, preco)
- Ficheiro:
  - **produtos.csv**
- Base de dados (H2)
  - Tabela: *produtos (id, nome, preco)*
- Consola
  - Imprimir



NO JDK

```java
public class FileVisitorExample {
    public static void main(String[] args) {
        Path startPath = Paths.get("caminho/do/diretorio");

        try {
            Files.walkFileTree(startPath, new SimpleFileVisitor<Path>() {
                @Override
                public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs) {
                    System.out.println("Visitando diretório: " + dir);
                    return FileVisitResult.CONTINUE;
                }

                @Override
                public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) {
                    System.out.println("Visitando arquivo: " + file);
                    return FileVisitResult.CONTINUE;
                }

                @Override
                public FileVisitResult visitFileFailed(Path file, IOException exc) {
                    System.err.println("Falha ao visitar arquivo: " + file + " (" + exc + ")");
                    return FileVisitResult.CONTINUE;
                }

                @Override
                public FileVisitResult postVisitDirectory(Path dir, IOException exc) {
                    if (exc == null) {
                        System.out.println("Finalizando visita ao diretório: " + dir);
                        return FileVisitResult.CONTINUE;
                    } else {
                        throw new RuntimeException(exc);
                    }
                }
            });
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

ANTI PATTERNS

Visitor Inflado: Criar visitantes que realizam muitas operações diferentes, tornando-os difíceis de entender e manter.

Falta de Encapsulamento: Expor detalhes internos dos elementos visitados ao visitante, violando o princípio de encapsulamento.

Dependência Circular: Criar dependências circulares entre o visitante e os elementos, complicando o fluxo de controle e a manutenção do código.

Falta de Flexibilidade: Não permitir a adição de novos tipos de elementos sem modificar o visitante, limitando a extensibilidade do sistema.

Visitantes Monolíticos: Implementar visitantes que lidam com muitos tipos de elementos diferentes, resultando em um código monolítico e difícil de gerenciar.

Falta de Documentação: Não documentar adequadamente as interações entre visitantes e elementos, dificultando a compreensão e a manutenção do código.

Uso Inadequado: Usar o padrão Visitor quando uma simples herança ou composição seria mais adequada, resultando em uma complexidade desnecessária.

RESUMO

OS PADRÕES DE COMPORTAMENTO PERMITEM ADICIONAR FUNCIONALIDADES A CLASSES, SEM TER DE AS ALTERAR, ASSIM COMO GERIR OS SEUS ESTADOS INTERNOS, SEJA COMO VERSÕES, QUER NO SUPORTE A UNDO/REDO.

RUMOR: O PADRÃO VISITOR COSTUMA SER O QUE OFERECE MAIS DESAFIO DE IMPLEMENTAR DE RAIZ

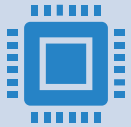
CÁBULA - CRIAÇÃO

Factory Method: Define uma interface para criar um objeto, mas deixa as subclasses decidirem qual classe instanciar. **Quando** você precisa de uma família de objetos relacionados e não sabe de antemão qual será necessário.

Abstract Factory: Fornece uma interface para criar famílias de objetos relacionados ou dependentes sem especificar suas classes concretas. **Quando** você precisa criar famílias de objetos relacionados, como interfaces de usuário para diferentes plataformas.

Builder: Separa a construção de um objeto complexo de sua representação, permitindo que o mesmo processo de construção crie diferentes representações. **Quando** a construção de um objeto é complexa e envolve muitos passos.

CÁBULA - CRIAÇÃO



Prototype: Especifica os tipos de objetos a serem criados usando uma instância protótipo e cria novos objetos copiando este protótipo. **Quando** a criação de um novo objeto é cara e você quer reduzir o tempo de criação.

Singleton: Garante que uma classe tenha apenas uma instância e fornece um ponto de acesso global a ela. **Quando** você precisa de exatamente um objeto de uma determinada classe, como um gerenciador de logs ou um objeto de configuração.

CÁBULA - ESTRUTURA

Adapter: Converte a interface de uma classe para outra interface que os clientes esperam. **Quando** você precisa usar uma classe existente que não possui a interface correta.

Bridge: Separa uma abstração de sua implementação, de modo que as duas possam variar independentemente. **Quando** você precisa desacoplar uma abstração de sua implementação.

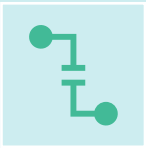
Composite: Compõe objetos em estruturas de árvore para representar hierarquias parte-todo. **Quando** você precisa representar hierarquias de objetos, como um sistema de arquivos ou uma estrutura de menus.

Decorator: Adiciona responsabilidades a um objeto dinamicamente. **Quando** você precisa adicionar funcionalidades a um objeto sem alterar sua classe.

CÁBULA - ESTRUTURA

Facade: Fornece uma interface unificada para um conjunto de interfaces em um subsistema. **Quando** você precisa simplificar a interface de um subsistema.

Flyweight: Usa compartilhamento para suportar eficientemente um grande número de objetos de granularidade fina. **Quando** você precisa representar um grande número de objetos similares.



Proxy: Fornece um substituto para outro objeto para controlar o acesso a ele. **Quando** você precisa controlar o acesso a um objeto, como um proxy de rede ou um proxy virtual.

CÁBULA - COMPORTAMENTO

Chain of Responsibility: Permite que várias classes manipulem uma solicitação, encadeando os objetos e passando a solicitação ao longo da cadeia até que um objeto a trate. **Quando** você precisa implementar diferentes comportamentos para uma mesma solicitação.

Command: Encapsula uma solicitação como um objeto, permitindo que você parametrize clientes com diferentes solicitações, coloque solicitações em fila ou registre o histórico de solicitações. **Quando** você precisa implementar operações undo/redo ou quando você precisa parametrizar um objeto com uma ação a ser realizada.

Interpreter: Dada uma gramática, constrói uma árvore de representação para uma sentença nessa gramática e define uma interpretação para essa árvore. **Quando** você precisa implementar uma linguagem simples.

Iterator: Fornece uma maneira de acessar os elementos de um objeto agregado sequencialmente sem expor sua representação interna. **Quando** você precisa percorrer os elementos de uma coleção de objetos.

CÁBULA - COMPORTAMENTO

Mediator: Define um objeto que encapsula como um conjunto de objetos interage. O mediador promove o acoplamento fraco, centralizando toda a comunicação. **Quando** você precisa reduzir o acoplamento entre muitos objetos.

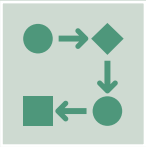
Memento: Captura e externaliza o estado interno de um objeto, permitindo que o objeto seja restaurado a esse estado posteriormente. **Quando** você precisa implementar a funcionalidade undo/redo.

Observer: Define uma dependência um-para-muitos entre objetos, de modo que quando um objeto muda de estado, todos os seus dependentes são notificados e atualizados automaticamente. **Quando** você precisa implementar um mecanismo de notificação.

State: Permite que um objeto altere seu comportamento quando seu estado interno muda. **Quando** o comportamento de um objeto depende de seu estado interno.

CÁBULA - COMPORTAMENTO

Strategy: Define uma família de algoritmos, encapsula cada um deles e os torna intercambiáveis. **Quando** você precisa escolher entre diferentes algoritmos em tempo de execução.



Template Method: Define o esqueleto de um algoritmo em um método, diferindo alguns passos para as subclasses. **Quando** você precisa implementar a estrutura de um algoritmo, mas permitir que as subclasses forneçam a implementação de alguns passos.

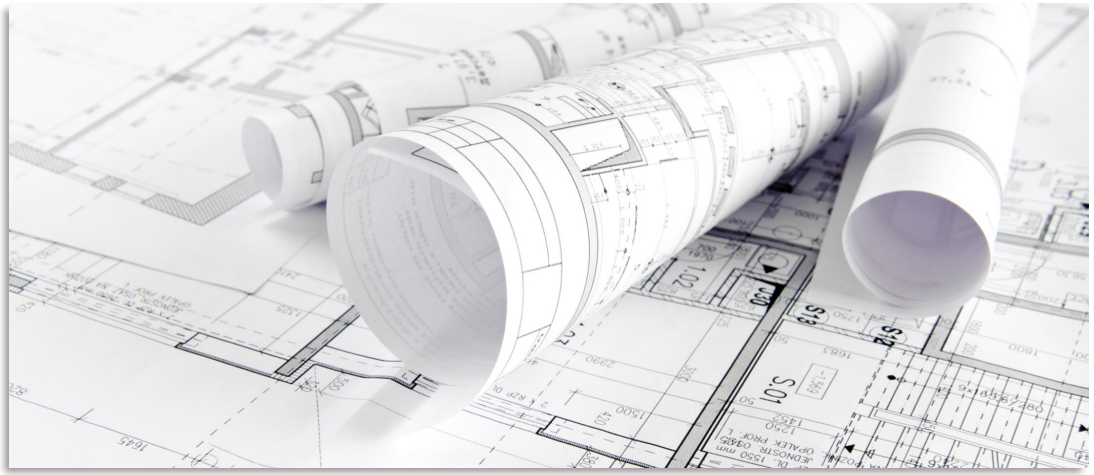
Visitor: Representa uma operação a ser executada sobre os elementos de uma estrutura de objeto. Permite que você defina novas operações sem mudar as classes dos elementos sobre os quais operam. **Quando** você precisa adicionar novas operações a uma estrutura de objetos existente.

OBRIGADO

Obrigado pela participação e dedicação ao longo deste exigente e ambicioso curso!

Padrões

- Creacionais
- Estruturais
- Comportamentais



Há sempre mais a aprender! … mas depois de tanto esforço…

mais de 100 exercícios! … um descanso é merecido!

Podes sempre encontrar-me por aqui, ou nas redes sociais como o LinkedIn: https://www.linkedin.com/in/joselopescruz/