COTI Informática

Java Design Patterns

Apostila

COTI Informática - Av. Rio Branco, 185 - Sala 904
Rio de Janeiro – RJ Tel. 21-2262-9043
www.cotiinformatica.com.br

Design Patterns - Introdução

Os padrões de projeto buscam descrever uma solução geral reutilizável para um problema recorrente no desenvolvimento de sistemas de *software* orientados a objetos.

Não é um código final, é uma descrição ou modelo de como resolver o problema do qual trata, que pode ser usada em muitas situações diferentes.

Os Padrões de Projeto normalmente definem as relações e interações entre as classes ou objetos, sem especificar os detalhes das classes ou objetos envolvidos, ou seja, estão num nível de generalidade mais alto.

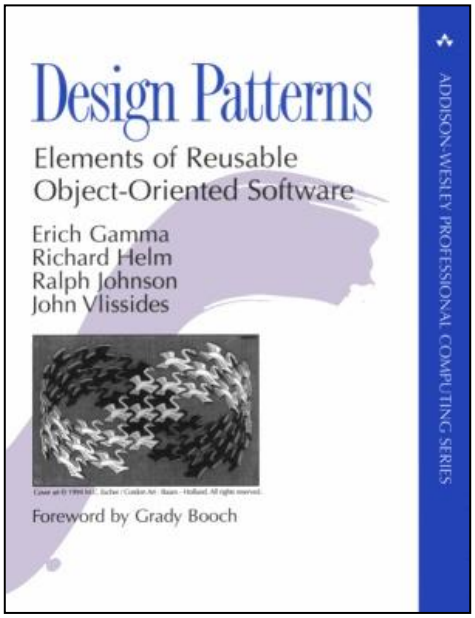
Os padrões de projeto:

- Visam facilitar a reutilização de soluções de desenho - isto é, soluções na fase de projeto do software.
- Estabelecem um vocabulário comum de desenho, facilitando comunicação, documentação e aprendizado dos sistemas de*software*.

Os design patterns são organizados em três famílias:

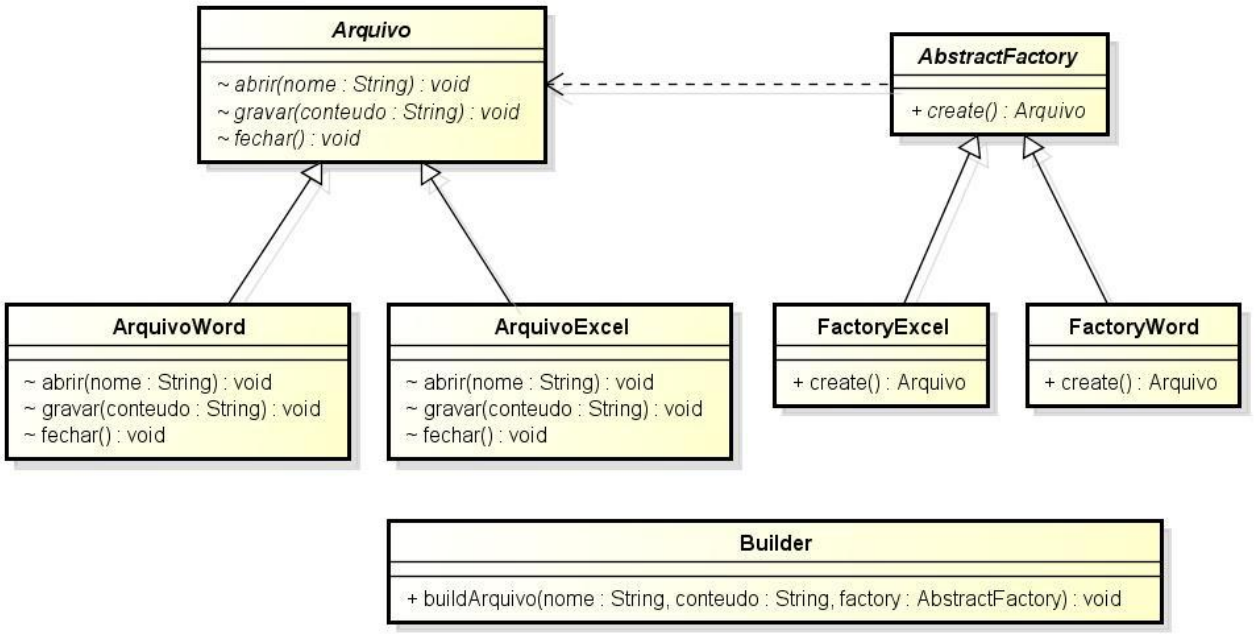
- **Padrões de criação**: relacionados à criação de objetos
- **Padrões estruturais**: tratam das associações entre classes e objetos.
- **Padrões comportamentais**: tratam das interações e divisões de responsabilidades entre as classes ou objetos.

Bibliografia recomendada:



Abstract Factory

Este padrão permite a criação de famílias de objetos relacionados ou dependentes por meio de uma única interface e sem que a classe concreta seja especificada.




```
package abstractfactory;

public abstract class AbstractFactory {

	public abstract Arquivo create();

}
```

```
package abstractfactory;

import java.io.FileWriter;

public abstract class Arquivo {

	protected FileWriter arquivo;

	abstract void abrir(String nome) throws Exception;

	abstract void gravar(String conteudo) throws Exception;

	abstract void fechar() throws Exception;

}
```

```
package abstractfactory;

import java.io.File;
import java.io.FileWriter;
```

```
public class ArquivoExcel extends Arquivo {

	@Override
	void abrir(String nome) throws Exception {
		arquivo = new FileWriter(new File("d:\\" + nome + ".csv"));
	}

	@Override
	void gravar(String conteudo) throws Exception {
		arquivo.write(conteudo);
	}

	@Override
	void fechar() throws Exception {
		arquivo.close();
	}
}
```

```
package abstractfactory;

import java.io.File;
import java.io.FileWriter;

public class ArquivoWord extends Arquivo {

	@Override
	void abrir(String nome) throws Exception {
		arquivo = new FileWriter(new File("d:\\" + nome + ".doc"));
	}

	@Override
	void gravar(String conteudo) throws Exception {
		arquivo.write(conteudo);
	}

	@Override
	void fechar() throws Exception {
		arquivo.close();
	}
}
```

```
package abstractfactory;

public class Builder {

	public void buildArquivo(String nome, String conteudo,
				AbstractFactory factory) throws Exception{

		Arquivo a = factory.create();

		a.abrir(nome);
		a.gravar(conteudo);
		a.fechar();
	}
}
```

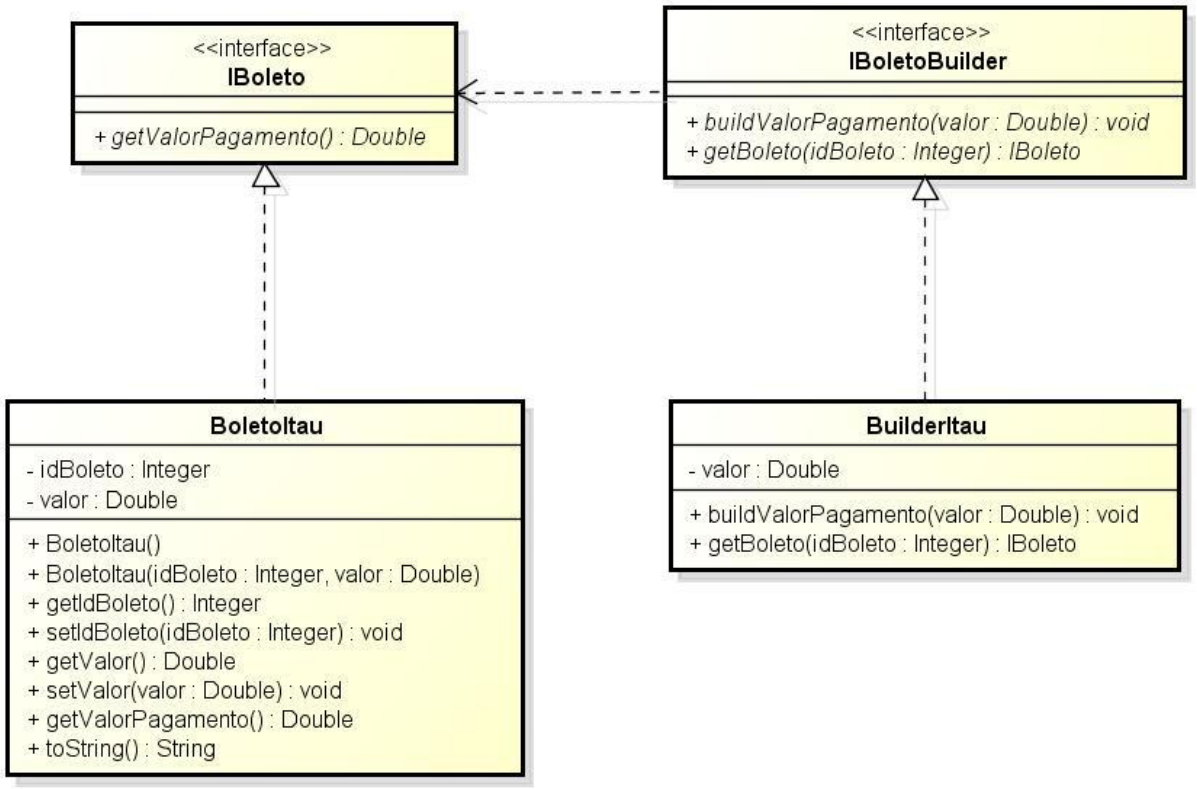
```
package abstractfactory;

public class FactoryExcel extends AbstractFactory {

	@Override
	public Arquivo create() {
		return new ArquivoExcel();
	}

}
```

```
package abstractfactory;

public class FactoryWord extends AbstractFactory {

	@Override
	public Arquivo create() {
		return new ArquivoWord();
	}

}
```

```
package abstractfactory;

import java.util.Scanner;

public class Main {

	public static void main(String[] args) {

		Builder builder = new Builder();
		AbstractFactory factory = null;

		try {

			System.out.println("Tipo: ");
			String opcao = new Scanner(System.in).nextLine();

			if (opcao.equalsIgnoreCase("word")) {
				factory = new FactoryWord();

			} else if (opcao.equalsIgnoreCase("excel")) {
				factory = new FactoryExcel();
			}

			builder.buildArquivo("teste", "Testando Patterns", factory);

			System.out.println("Dados gravados");

		} catch (Exception e) {
			e.printStackTrace();
		}
	}
}
```

Builder

Permite a separação da construção de um objeto complexo da sua representação, de forma que o mesmo processo de construção possa criar diferentes representações.




```
package builder;

public class BoletoItau implements IBoleto {

	private Integer idBoleto;
	private Double valor;

	public BoletoItau() {

	}

	public BoletoItau(Integer idBoleto, Double valor) {
		super();
		this.idBoleto = idBoleto;
		this.valor = valor;
	}

	public Integer getIdBoleto() {
		return idBoleto;
	}

	public void setIdBoleto(Integer idBoleto) {
		this.idBoleto = idBoleto;
	}
```

```
	public Double getValor() {
		return valor;
	}

	public void setValor(Double valor) {
		this.valor = valor;
	}

	@Override
	public Double getValorPagamento() {
		return valor * 1.12;
	}

	@Override
	public String toString() {
		return "BoletoItau [idBoleto=" + idBoleto + ", valor=" + valor
				+ ", Valor Pagamento =" + getValorPagamento() + "]";
	}

}
```

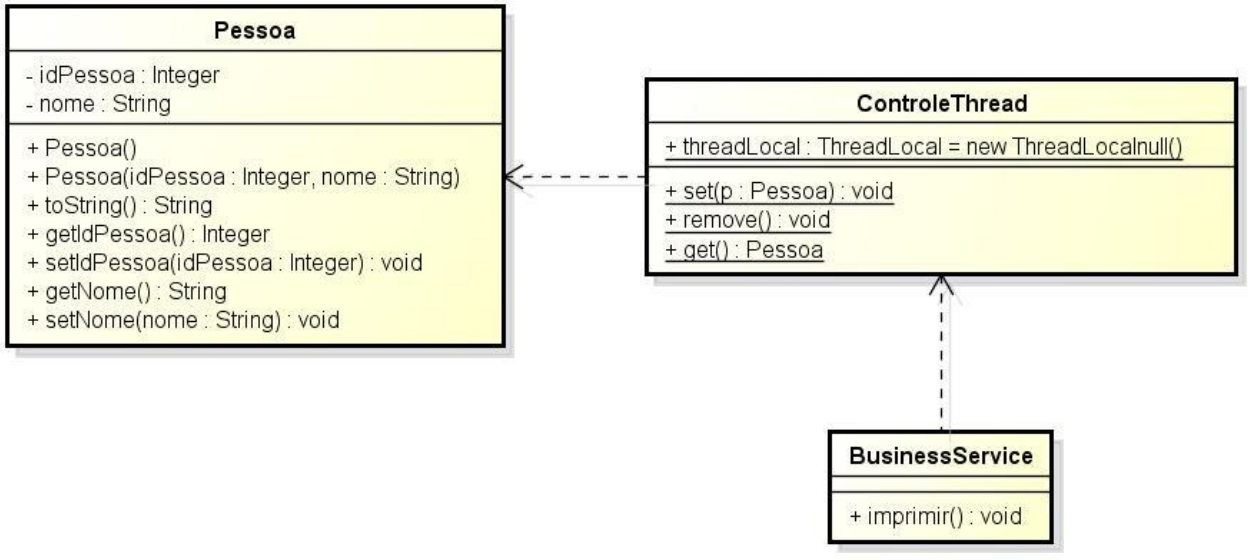
```
package builder;

public class BuilderItau implements IBoletoBuilder {

	private Double valor;

	@Override
	public void buildValorPagamento(Double valor) {
		this.valor = valor;
	}

	@Override
	public IBoleto getBoleto(Integer idBoleto) {
		return new BoletoItau(idBoleto, valor);
	}

}
```

```
package builder;

public interface IBoleto {

	Double getValorPagamento();

}
```

```
package builder;

public interface IBoletoBuilder {

	void buildValorPagamento(Double valor);
	IBoleto getBoleto(Integer idBoleto);
}
```

```
package builder;

public class Main {

	IBoletoBuilder builder;

	public Main(IBoletoBuilder builder) {
		this.builder = builder;
	}

	public static void main(String[] args) {

		Main m = new Main(new BuilderItau());

		m.builder.buildValorPagamento(1000.0);

		System.out.println(m.builder.getBoleto(1));

	}
}
```

Business Service




```
package business;

public class BusinessService {

	public void imprimir() {

		Pessoa p = ControleThread.get();
		System.out.println(p);
	}
}
```

```
package business;

@SuppressWarnings("unchecked")
public class ControleThread {

	@SuppressWarnings("rawtypes")
	public static final ThreadLocal threadLocal = new ThreadLocal();

	public static void set(Pessoa p) {
		threadLocal.set(p);
	}

	public static void remove() {
		threadLocal.remove();
	}

	public static Pessoa get() {
		return (Pessoa) threadLocal.get();
	}

}
```

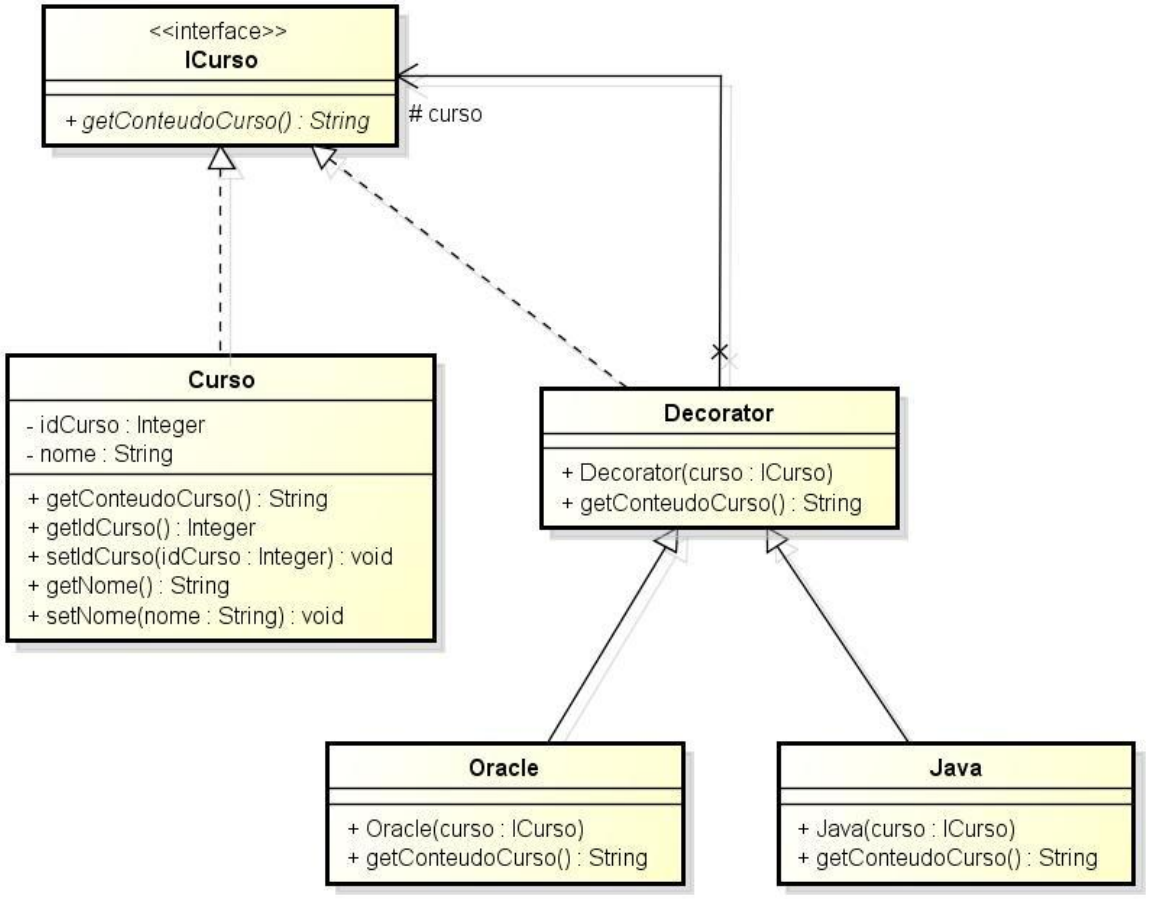
```
package business;

public class Pessoa {

	private Integer idPessoa;
	private String nome;

	public Pessoa() {

	}

	public Pessoa(Integer idPessoa, String nome) {
		super();
		this.idPessoa = idPessoa;
		this.nome = nome;
	}

	@Override
	public String toString() {
		return "Pessoa [idPessoa=" + idPessoa + ", nome=" + nome + "]";
	}

	public Integer getIdPessoa() {
		return idPessoa;
	}

	public void setIdPessoa(Integer idPessoa) {
		this.idPessoa = idPessoa;
	}
```

```
	public String getNome() {
		return nome;
	}

	public void setNome(String nome) {
		this.nome = nome;
	}

}
```

```
package business;

public class Main extends Thread {

	public static void main(String args[]) {

		Thread threadUm   = new Main();
		Thread threadDois = new Main();

		threadUm.start();
		threadDois.start();
	}

	@Override
	public void run() {

		for(int i = 1; i < 10; i++){

			Pessoa p = new Pessoa(i, "Belem -> " + i);
			ControleThread.set(p);

			new BusinessService().imprimir();
			ControleThread.remove();

		}
	}
}
```

Decorator

O padrão de projeto (Design Pattern) Decorator tem como principal objetivo a decoração de classes em tempo de execução, isto é, adicionar novos produtos e/ou novas responsabilidades à objetos dinamicamente sem alterar o código das classes existentes.




```
package decorator;

public class Curso implements ICurso {

	private Integer idCurso;
	private String nome;

	@Override
	public String getConteudoCurso() {
		return idCurso + ", " + nome;
	}

	public Integer getIdCurso() {
		return idCurso;
	}

	public void setIdCurso(Integer idCurso) {
		this.idCurso = idCurso;
	}
```

```
	public String getNome() {
		return nome;
	}

	public void setNome(String nome) {
		this.nome = nome;
	}

}
```

```
package decorator;

public class Decorator implements ICurso {

	protected ICurso curso;

	public Decorator(ICurso curso) {
		this.curso = curso;
	}

	@Override
	public String getConteudoCurso() {
		return curso.getConteudoCurso();
	}
}
```

```
package decorator;

public interface ICurso {

	String getConteudoCurso();

}
```

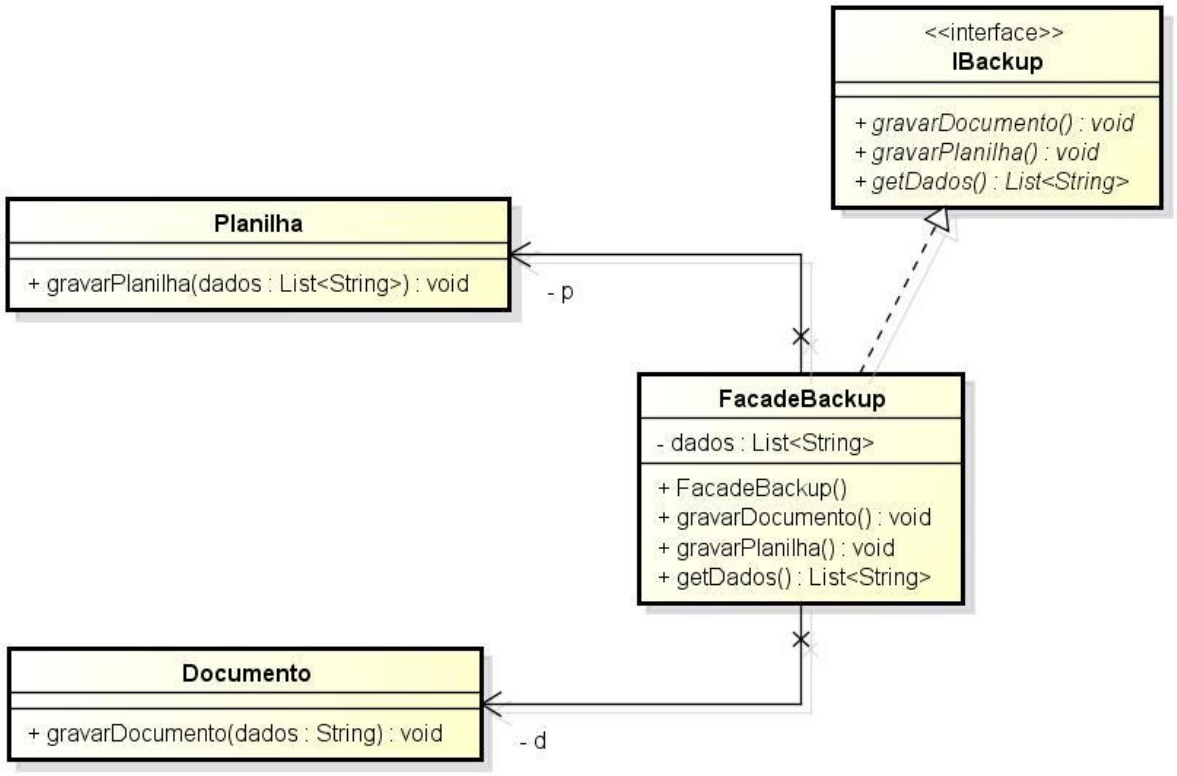
```
package decorator;

public class Java extends Decorator {

	public Java(ICurso curso) {
		super(curso);
	}

	@Override
	public String getConteudoCurso() {
		return super.getConteudoCurso() + ", Java Orientado a Objetos";
	}

}
```

```
package decorator;

public class Oracle extends Decorator {

	public Oracle(ICurso curso) {
		super(curso);
	}

	@Override
	public String getConteudoCurso() {
		return super.getConteudoCurso() + ", Oracle DBA";
	}

}
```

```
package decorator;

public class Main {

	public static void main(String[] args) {

		Curso c = new Curso();
		c.setIdCurso(1);
		c.setNome("Intensivo");

		ICurso curso = new Java(new Oracle(c));
		System.out.println(curso.getConteudoCurso());

	}
}
```

Facade

Um façade (*fachada*) é um objeto que disponibiliza uma interface simplificada para uma das funcionalidades de uma API, por exemplo.




```
package facade;

import java.io.File;
import java.io.FileWriter;

public class Documento {

	public void gravarDocumento(String dados) throws Exception{

		FileWriter w = new FileWriter(new File("d:\\facade.doc"), true);
		w.write(dados);
		w.close();
	}

}
```

```
package facade;

import java.util.ArrayList;
import java.util.List;

public class FacadeBackup implements IBackup {

	private List<String> dados;
```

```
	private Documento d;
	private Planilha p;

	public FacadeBackup() {
		dados = new ArrayList<String>();
		d = new Documento();
		p = new Planilha();
	}

	@Override
	public void gravarDocumento() throws Exception {
		d.gravarDocumento(dados.toString());
	}

	@Override
	public void gravarPlanilha() throws Exception {
		p.gravarPlanilha(dados);
	}

	public List<String> getDados() {
		return dados;
	}

}
```

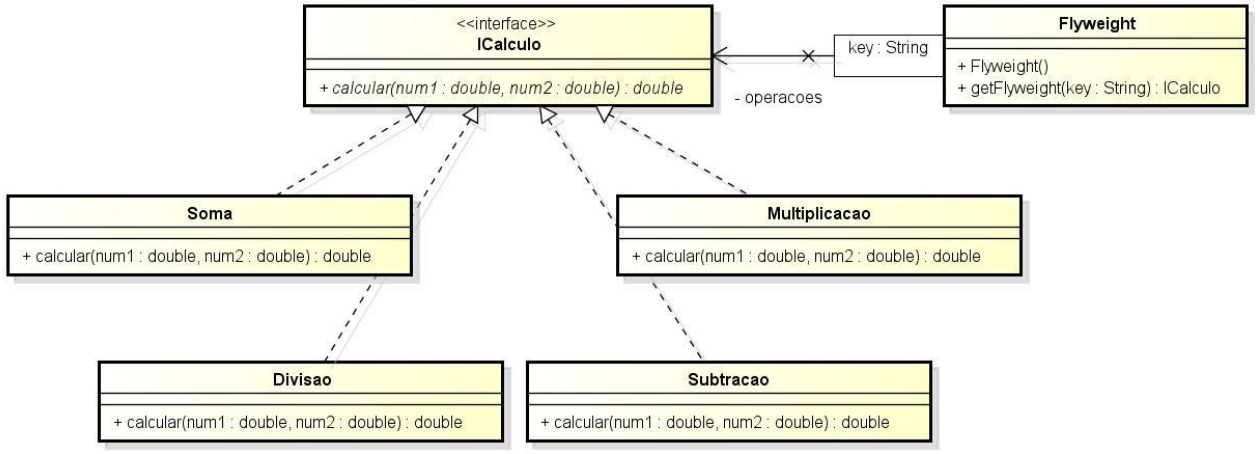
```
package facade;

import java.util.List;

public interface IBackup {

	void gravarDocumento() throws Exception;

	void gravarPlanilha() throws Exception;

	List<String> getDados();
}
```

```
package facade;

import java.io.File;
import java.io.FileWriter;
import java.util.List;

public class Planilha {


	public void gravarPlanilha(List<String> dados) throws Exception {

		FileWriter w = new FileWriter(new File("d:\\facade.csv"), true);
```

```
            for (String linha : dados) {


                  w.write(linha);
                  w.write("\n");
            }
            w.close();
      }

}



package facade;

public class Main {

      public static void main(String[] args) {

            IBackup b = new FacadeBackup();

            b.getDados().add("Aula de Java");
            b.getDados().add("Design Patterns");
            b.getDados().add("Facade");

            try{
                  b.gravarDocumento();
                  b.gravarPlanilha();

                  System.out.println("Backup realizado");
            }
            catch(Exception e){
                  System.out.println(e.getMessage());
            }
      }
}
```

Flyweight

Flyweight é um padrão de projeto de software apropriado quando vários objetos devem ser manipulados, e esses não suportam dados adicionais. No padrão *flyweight* não existem ponteiros para os métodos do dado, pois isto consome muita memória. Em contrapartida são chamadas sub-rotinas diretamente para acessar o dado.




```
package flyweight;

public class Divisao implements ICalculo {

	@Override
	public double calcular(double num1, double num2) {
		try {
			return num1 / num2;
		}
		catch (ArithmeticException e) {
			e.printStackTrace();
			return 0.;
		}
	}
}
```

```
package flyweight;

import java.util.HashMap;
import java.util.Map;

public class Flyweight {

	private Map<String, ICalculo> operacoes;

	public Flyweight() {
		operacoes = new HashMap<String, ICalculo>();
	}
```

```
	public ICalculo getFlyweight(String key) {

		if ( ! operacoes.containsKey(key)) {

			if ("soma".equals(key)) {
				operacoes.put(key, new Soma());
			}
			else if("sub".equals(key)){
				operacoes.put(key, new Subtracao());
			}
			else if("mult".equals(key)){
				operacoes.put(key, new Multiplicacao());
			}
			else if("div".equals(key)){
				operacoes.put(key, new Divisao());
			}
		}

		return operacoes.get(key);
	}

}
```

```
package flyweight;

public interface ICalculo {

	double calcular(double num1, double num2);

}
```

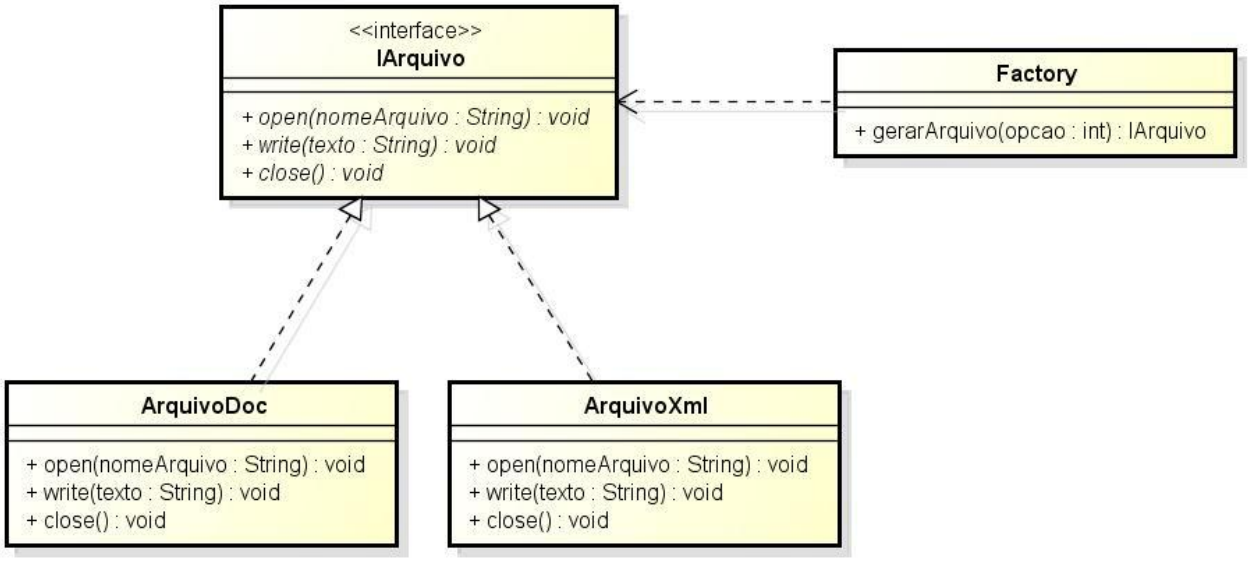
```
package flyweight;

public class Multiplicacao implements ICalculo {

	@Override
	public double calcular(double num1, double num2) {
		return num1 * num2;
	}
}
```

```
package flyweight;

public class Soma implements ICalculo {

	@Override
	public double calcular(double num1, double num2) {
		return num1 + num2;
	}
}
```

```
package flyweight;

public class Subtracao implements ICalculo {

	@Override
	public double calcular(double num1, double num2) {
		return num1 - num2;
	}
}



package flyweight;

public class Main {

	public static void main(String[] args) {

		Flyweight f = new Flyweight();

		for (int i = 0; i < 5; i++) {

			ICalculo soma = f.getFlyweight("soma");
			System.out.println("Somando: " + soma.calcular(i, i));

			ICalculo multiplicacao = f.getFlyweight("mult");
			System.out.println("Multiplicando: " +
					multiplicacao.calcular(i, i));
		}

	}

}
```

Factory Method

Define uma interface para criação de um objeto, mas permite que as subclasses escolham quais classes instanciar. O factory method permite delegar a instanciação para as subclasses.




```
package factory;

import java.io.File;
import java.io.FileWriter;

public class ArquivoDoc implements IArquivo {

	private FileWriter fw;

	@Override
	public void open(String nomeArquivo) throws Exception {
		fw = new FileWriter(new File("d:\\" + nomeArquivo + ".doc"));
	}

	@Override
	public void write(String texto) throws Exception {
		fw.write(texto);
	}

	@Override
	public void close() throws Exception {
		fw.close();
	}

}
```

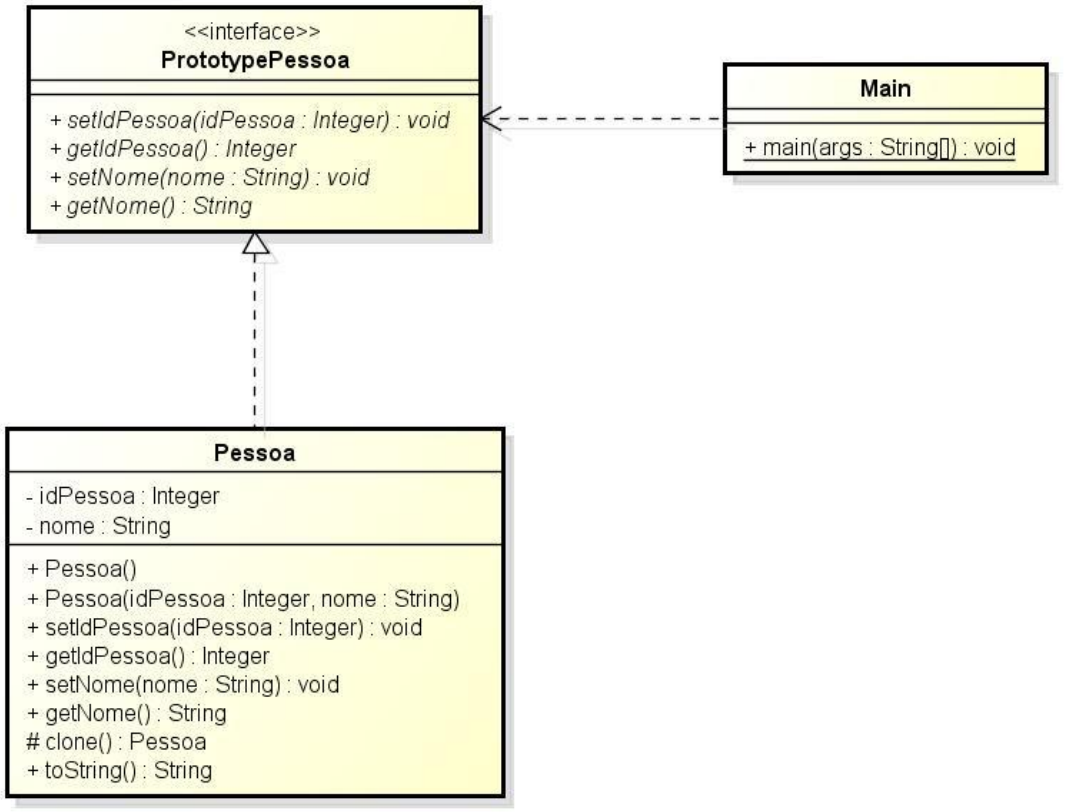
```
package factory;

public class Factory {

	public IArquivo gerarArquivo(int opcao) {

		switch (opcao) {
		case 1:
			return new ArquivoDoc();
		case 2:
			return new ArquivoXml();
		}

		return null;
	}

}
```

```
package factory;

public interface IArquivo {

	void open(String nomeArquivo) throws Exception;

	void write(String texto) throws Exception;

	void close() throws Exception;

}
```

```
package factory;

public class Main {

	public static void main(String[] args) {

		IArquivo a = new Factory().gerarArquivo(2);

		try{

			a.open("dados");
			a.write("Aula de Design Pattern");
			a.close();

			System.out.println("Dados gravados com sucesso");
		}
		catch(Exception e){
			System.out.println(e.getMessage());
		}
	}
}
```

Prototype

É um padrão de projeto de software (*design pattern*, em inglês) que permite a criação de objetos a partir de um modelo original, ou protótipo.




```
package prototype;

public class Pessoa implements PrototypePessoa, Cloneable {

	private Integer idPessoa;
	private String nome;

	public Pessoa() {
	}

	public Pessoa(Integer idPessoa, String nome) {
		super();
		this.idPessoa = idPessoa;
		this.nome = nome;
	}

	@Override
	public void setIdPessoa(Integer idPessoa) {
		this.idPessoa = idPessoa;
	}

	@Override
	public Integer getIdPessoa() {
		return idPessoa;
	}
```

```
	@Override
	public void setNome(String nome) {
		this.nome = nome;
	}

	@Override
	public String getNome() {
		return nome;
	}

	@Override
	protected Pessoa clone() throws CloneNotSupportedException {
		return (Pessoa) super.clone();
	}

	@Override
	public String toString() {
		return "Pessoa [idPessoa=" + idPessoa + ", nome=" + nome + "]";
	}

}
```

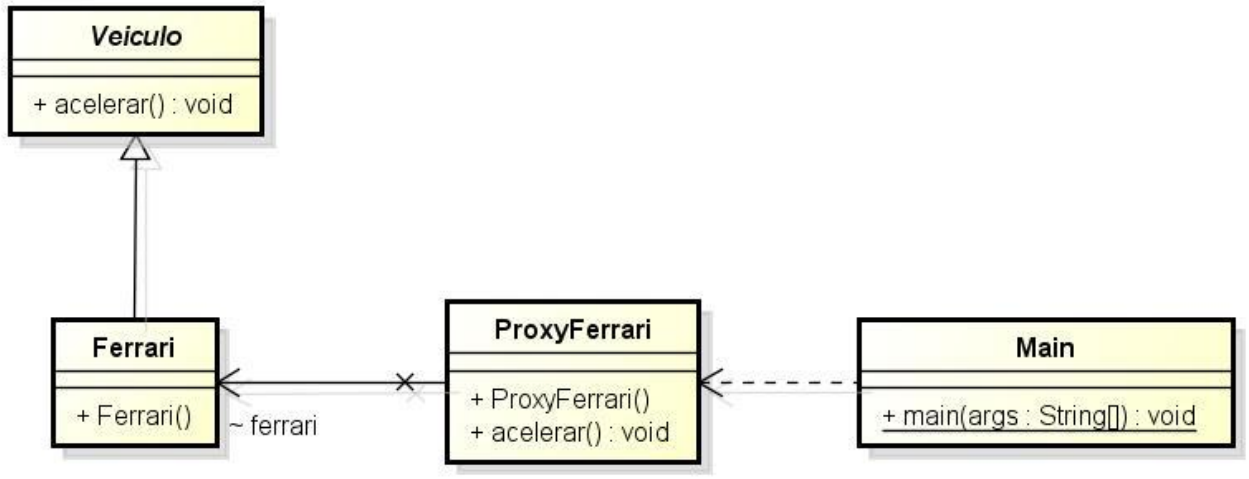
```
package prototype;

public interface PrototypePessoa {

	void setIdPessoa(Integer idPessoa);
	Integer getIdPessoa();

	void setNome(String nome);
	String getNome();

}
```

```
package prototype;

public class Main {

	public static void main(String[] args) {

		try {

			Pessoa p = new Pessoa();

			for (int i = 1; i <= 5; i++) {

				PrototypePessoa prototype = p.clone();

				prototype.setIdPessoa(i);
				prototype.setNome("Belem " + i);
```

```
                        System.out.println(prototype);
                  }

            } catch (Exception e) {
                  System.out.println(e.getMessage());
            }
      }
}
```

Proxy

Um proxy, em sua forma mais geral, é uma classe que funciona como uma interface para outra classe. A classe proxy poderia conectar-se a qualquer coisa: uma conexão de rede, um objeto grande em memória, um arquivo, ou algum recurso que é difícil ou impossível de ser duplicado.




```
package proxy;

public class Ferrari extends Veiculo {

      public Ferrari() {

            try {
                  Thread.sleep(5000);
            }
            catch (Exception e) {
                  e.printStackTrace();
            }

      }

}
```

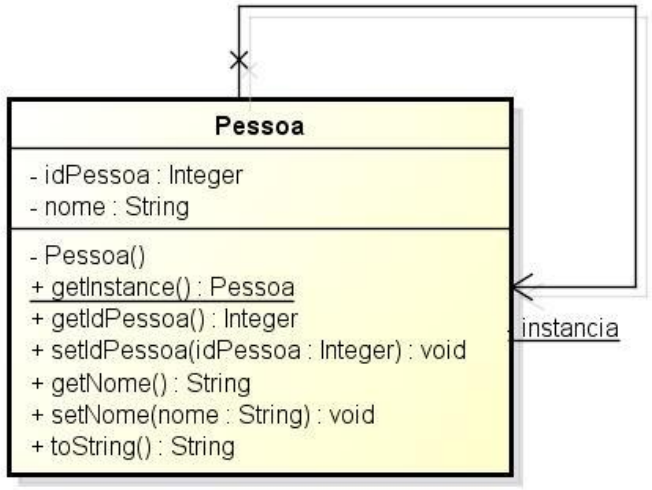
```
package proxy;

import java.text.SimpleDateFormat;
import java.util.Date;

public class ProxyFerrari {

	Ferrari ferrari;

	public ProxyFerrari() {

		String tempo = new SimpleDateFormat("HH:mm:ss").
					format(new Date());

		System.out.println("Ferrari parada em: " + tempo);
	}

	public void acelerar(){

		if(ferrari == null){
			ferrari = new Ferrari();
		}

		ferrari.acelerar();
	}
}
```

```
package proxy;

import java.text.SimpleDateFormat;
import java.util.Date;

public abstract class Veiculo {

	public void acelerar(){

		String tempo = new SimpleDateFormat("HH:mm:ss").
				format(new Date());

		System.out.println(this.getClass().getSimpleName() +
			", chegou a 100km/h em " + tempo);
	}
}
```

```
package proxy;

public class Main {

	public static void main(String[] args) {

		ProxyFerrari proxy = new ProxyFerrari();
		proxy.acelerar();
	}
}
```

Singleton

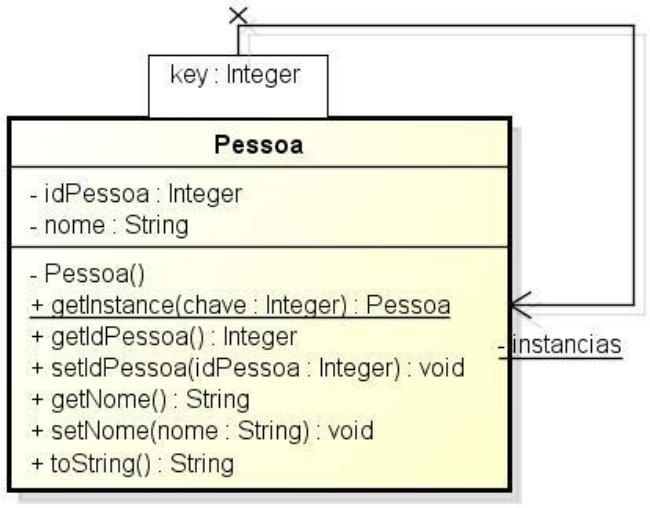
Este padrão garante a existência de apenas uma instância de uma classe, mantendo um ponto global de acesso ao seu objeto.




```
package singleton;

public final class Pessoa {

	private Integer idPessoa;
	private String nome;
	private static Pessoa instancia;

	private Pessoa() {

	}

	public static Pessoa getInstance() {
		if (instancia == null)
			instancia = new Pessoa();
		return instancia;
	}

	public Integer getIdPessoa() {
		return idPessoa;
	}

	public void setIdPessoa(Integer idPessoa) {
		this.idPessoa = idPessoa;
	}

	public String getNome() {
		return nome;
	}

	public void setNome(String nome) {
		this.nome = nome;
	}
```

```
	@Override
	public String toString() {
		return "Pessoa [idPessoa=" + idPessoa + ", nome=" + nome + "]";
	}

}
```

```
package singleton;

public class Main {

	public static void main(String[] args) {

		Pessoa p1 = Pessoa.getInstance();
		Pessoa p2 = Pessoa.getInstance();

		p1.setIdPessoa(1);
		p1.setNome("Belem");

		p2.setIdPessoa(2);
		p2.setNome("NetCat");

		System.out.println(p1);
		System.out.println(p2);
	}

}
```



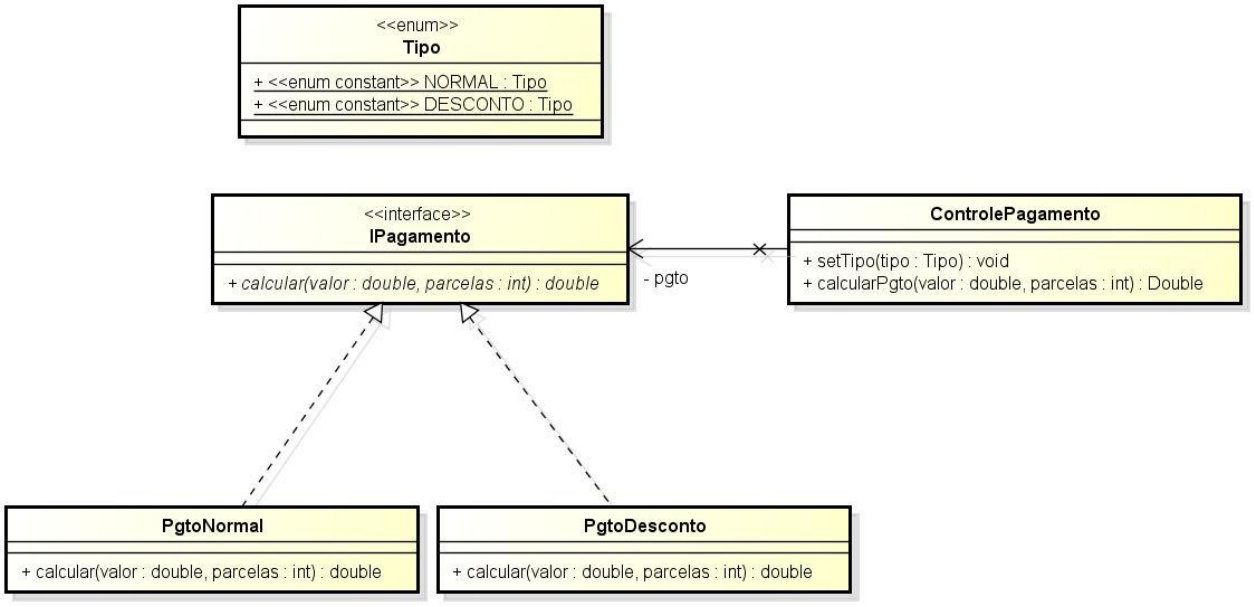

```
package singletonmapa;

import java.util.HashMap;
import java.util.Map;

public class Pessoa {

	private static final Map<Integer, Pessoa> instancias
			= new HashMap<>();
```

```
	private Integer idPessoa;
	private String nome;

	private Pessoa() {
	}

	public static synchronized Pessoa getInstance(Integer chave) {

		Pessoa instancia = instancias.get(chave);

		if (instancia == null) {
			instancia = new Pessoa();

			instancias.put(chave, instancia);
		}

		return instancia;
	}

	public Integer getIdPessoa() {
		return idPessoa;
	}

	public void setIdPessoa(Integer idPessoa) {
		this.idPessoa = idPessoa;
	}

	public String getNome() {
		return nome;
	}

	public void setNome(String nome) {
		this.nome = nome;
	}

	@Override
	public String toString() {
		String saida = "Pessoas: ";
		for (Pessoa p : instancias.values()) {
			saida += "\n" + p.getIdPessoa() + ", " + p.getNome();
		}

		return saida;
	}
}


package singletonmapa;

public class Main {

	public static void main(String[] args) {

		Pessoa p1 = Pessoa.getInstance(1);
		p1.setIdPessoa(1);
		p1.setNome("Sergio");
```

```
            Pessoa p2 = Pessoa.getInstance(2);
            p2.setIdPessoa(2);
            p2.setNome("Pedro");

            Pessoa p3 = Pessoa.getInstance(2);
            p3.setIdPessoa(3);
            p3.setNome("Belem");

            System.out.println(p1);
            System.out.println(p2);
      }

}
```

State

State é um padrão de projeto de software usado quando o comportamento de um objeto muda, dependendo do seu estado.




```
package state;

public class ControlePagamento {

      private IPagamento pgto;

      public void setTipo(Tipo tipo) {

            switch (tipo) {

            case NORMAL:
                  pgto = new PgtoNormal();
                  break;
```

```
            case DESCONTO:
                  pgto = new PgtoDesconto();
                  break;
            }
      }

      public Double calcularPgto(double valor, int parcelas) {
            return pgto.calcular(valor, parcelas);
      }
}
```

```
package state;

public interface IPagamento {

      double calcular(double valor, int parcelas);

}
```

```
package state;

public class PgtoDesconto implements IPagamento {

      @Override
      public double calcular(double valor, int parcelas) {
            return parcelas * (valor * 0.9);
      }

}
```

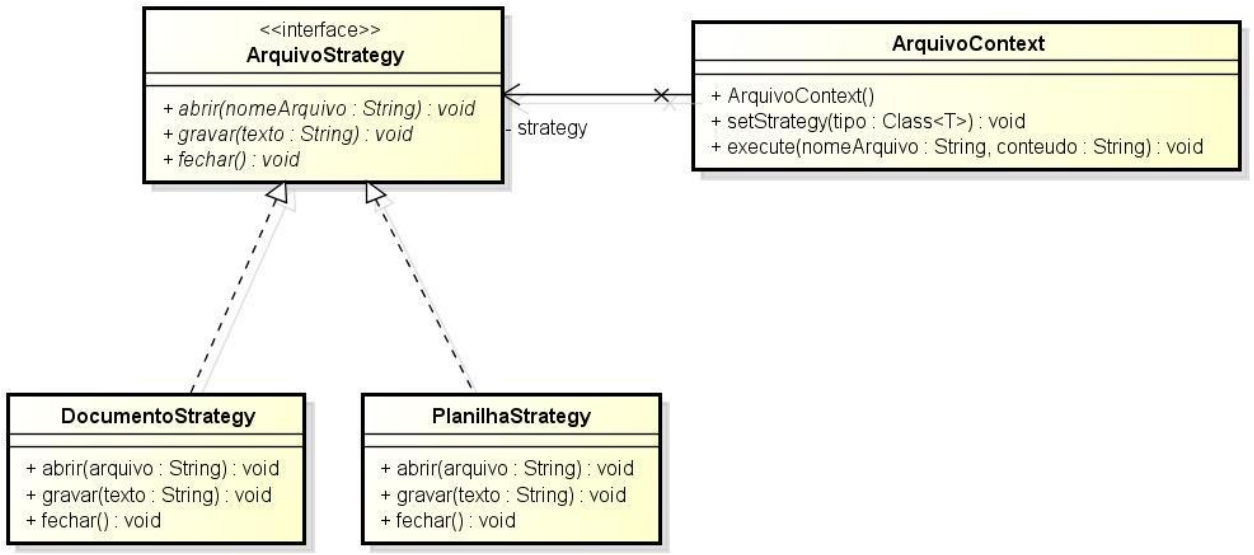
```
package state;

public class PgtoNormal implements IPagamento {

      @Override
      public double calcular(double valor, int parcelas) {
            return parcelas * valor;
      }

}
```

```
package state;

public enum Tipo {

      NORMAL, DESCONTO

}
```

```
package state;

public class Main {

	public static void main(String[] args) {

		ControlePagamento c = new ControlePagamento();

		c.setTipo(Tipo.NORMAL);
		System.out.println("Pgto: " + c.calcularPgto(100., 10));

		c.setTipo(Tipo.DESCONTO);
		System.out.println("Pgto: " + c.calcularPgto(100., 10));

	}

}
```

Strategy

O objetivo é representar uma operação a ser realizada sobre os elementos de uma estrutura de objetos. O padrão Strategy permite definir novas operações sem alterar as classes dos elementos sobre os quais opera. Definir uma família de algoritmos e encapsular cada algoritmo como uma classe, permitindo assim que elas possam ter trocados entre si. Este padrão permite que o algoritmo possa variar independentemente dos clientes que o utilizam.




```
package strategy;

public class ArquivoContext {

	private ArquivoStrategy strategy;

	public ArquivoContext() {

	}
```

```
	public <T extends ArquivoStrategy> void setStrategy(Class<T> tipo) {
		try {
			this.strategy = tipo.newInstance();
		} catch (Exception e) {
			e.printStackTrace();
		}
	}

	public void execute(String nomeArquivo, String conteudo)
		throws Exception {

		strategy.abrir(nomeArquivo);
		strategy.gravar(conteudo);
		strategy.fechar();

	}
}
```

```
package strategy;

public interface ArquivoStrategy {

	void abrir(String nomeArquivo) throws Exception;

	void gravar(String texto) throws Exception;

	void fechar() throws Exception;

}
```

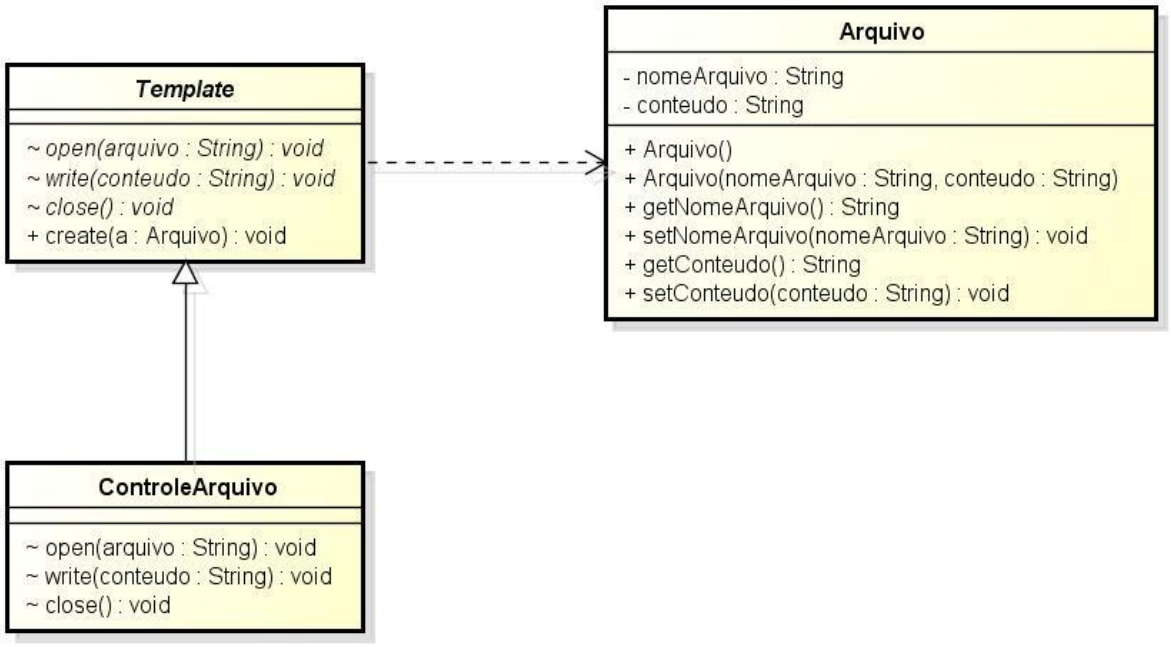
```
package strategy;

import java.io.File;
import java.io.FileWriter;

public class DocumentoStrategy implements ArquivoStrategy {

	private FileWriter writer;

	@Override
	public void abrir(String arquivo) throws Exception {
		writer = new FileWriter(new File("d:\\" + arquivo + ".doc"));
	}

	@Override
	public void gravar(String texto) throws Exception {
		writer.write(texto + "\n");
	}

	@Override
	public void fechar() throws Exception {
		writer.close();
	}
}
```

```
package strategy;

import java.io.File;
import java.io.FileWriter;

public class PlanilhaStrategy implements ArquivoStrategy {

	private FileWriter writer;

	@Override
	public void abrir(String arquivo) throws Exception {
		writer = new FileWriter(new File("d:\\" + arquivo + ".csv"));
	}

	@Override
	public void gravar(String texto) throws Exception {
		writer.write(texto + "\n");
	}

	@Override
	public void fechar() throws Exception {
		writer.close();
	}

}
```

```
package strategy;

public class Main {

	public static void main(String[] args) {

		ArquivoContext c = new ArquivoContext();
		c.setStrategy(DocumentoStrategy.class);

		try {

			c.execute("aula", "testando design patterns");

			System.out.println("Dados gravados.");

		} catch (Exception e) {
			System.out.println(e.getMessage());
		}

	}
}
```

Template Method

Um Template Method auxilia na definição de um algoritmo com partes do mesmo definidos por Método abstratos. As subclasses devem se responsabilizar por estas partes abstratas, deste algoritmo, que serão implementadas, possivelmente de várias formas, ou seja, cada subclasse irá implementar à sua necessidade e oferecer um comportamento concreto construindo todo o algoritmo.




```
package template;

public class Arquivo {

	private String nomeArquivo;
	private String conteudo;

	public Arquivo() {
	}

	public Arquivo(String nomeArquivo, String conteudo) {
		super();
		this.nomeArquivo = nomeArquivo;
		this.conteudo = conteudo;
	}

	public String getNomeArquivo() {
		return nomeArquivo;
	}

	public void setNomeArquivo(String nomeArquivo) {
		this.nomeArquivo = nomeArquivo;
	}
```

```
	public String getConteudo() {
		return conteudo;
	}

	public void setConteudo(String conteudo) {
		this.conteudo = conteudo;
	}

}
```

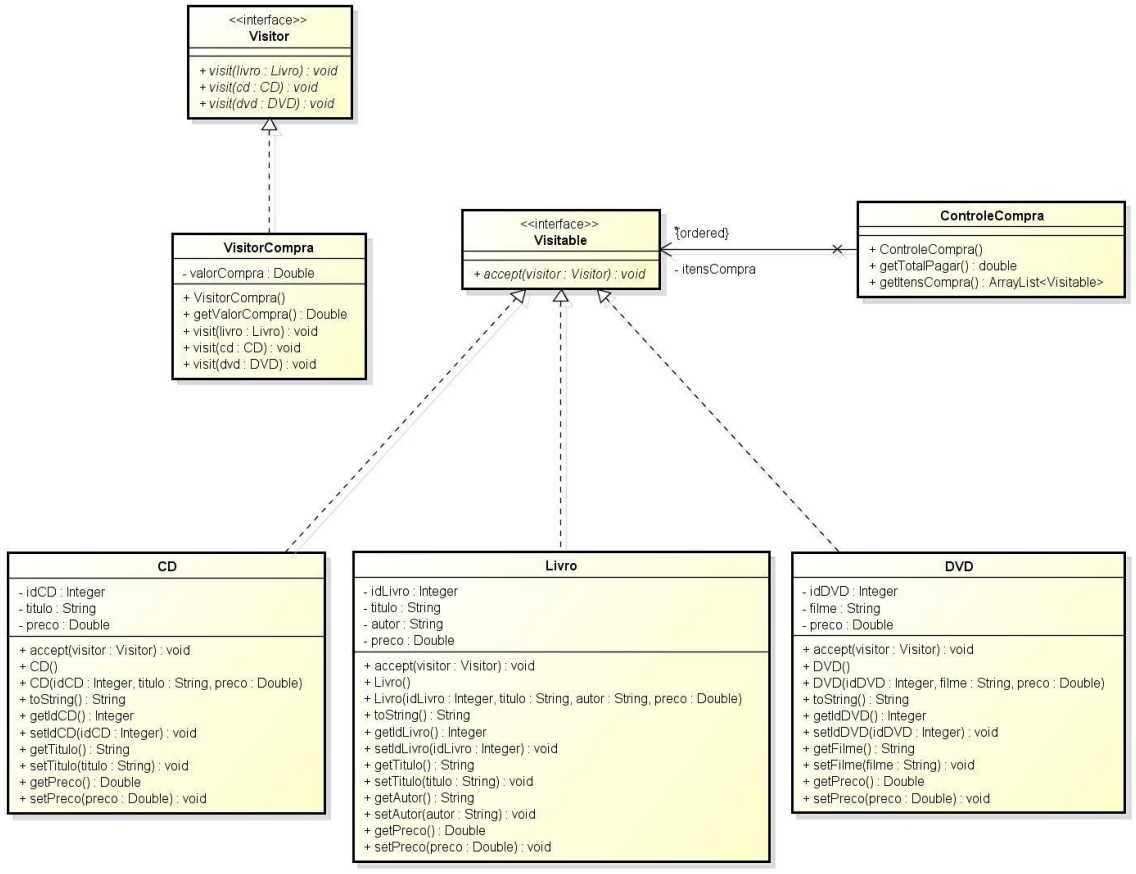
```
package template;

import java.io.File;
import java.io.FileWriter;

public class ControleArquivo extends Template {

	private FileWriter fw;

	@Override
	void open(String arquivo) throws Exception {
		fw = new FileWriter(new File("d://" + arquivo), true);
	}

	@Override
	void write(String conteudo) throws Exception {
		fw.write(conteudo);
	}

	@Override
	void close() throws Exception {
		fw.close();
	}

}
```

```
package template;

public abstract class Template {

	abstract void open(String arquivo) throws Exception;

	abstract void write(String conteudo) throws Exception;

	abstract void close() throws Exception;

	public final void create(Arquivo a) throws Exception {
		open(a.getNomeArquivo());
		write(a.getConteudo());
		close();
	}
}
```

```
package template;

public class Main {

	public static void main(String[] args) {

		Arquivo a = new Arquivo("texto.txt", "Aula de Java");
		ControleArquivo c = new ControleArquivo();

		try{
			c.create(a);
			System.out.println("Dados gravados");
		}
		catch(Exception e){
			System.out.println(e.getMessage());
		}
	}
}
```

Visitor

Em programação orientada a objetos e engenharia de software, o *visitor pattern* é um padrão de projeto comportamental. Representa uma operação a ser realizada sobre elementos da estrutura de um objeto. O Visitor permite que se crie uma nova operação sem que se mude a classe dos elementos sobre as quais ela opera. É uma maneira de separar um algoritmo da estrutura de um objeto. Um resultado prático é a habilidade de adicionar novas funcionalidades a estruturas de um objeto pré-existente sem a necessidade de modificá-las.



```
package visitor;

public class CD implements Visitable {

	private Integer idCD;
	private String titulo;
	private Double preco;

	@Override
	public void accept(Visitor visitor) {
		visitor.visit(this);
	}

	public CD() {

	}

	public CD(Integer idCD, String titulo, Double preco) {
		super();
		this.idCD = idCD;
		this.titulo = titulo;
		this.preco = preco;
	}

	@Override
	public String toString() {
		return "CD [idCD=" + idCD + ", titulo=" + titulo
				+ ", preco=" + preço + "]";
	}

	public Integer getIdCD() {
		return idCD;
	}

	public void setIdCD(Integer idCD) {
		this.idCD = idCD;
	}

	public String getTitulo() {
		return titulo;
	}

	public void setTitulo(String titulo) {
		this.titulo = titulo;
	}

	public Double getPreco() {
		return preco;
	}

	public void setPreco(Double preco) {
		this.preco = preco;
	}

}
```

```
package visitor;

import java.util.ArrayList;

public class ControleCompra {

	private ArrayList<Visitable> itensCompra;

	public ControleCompra() {
		itensCompra = new ArrayList<Visitable>();
	}

	public double getTotalPagar() {

		VisitorCompra visitor = new VisitorCompra();

		for (Visitable item : itensCompra) {
			item.accept(visitor);
		}

		return visitor.getValorCompra();
	}

	public ArrayList<Visitable> getItensCompra() {
		return itensCompra;
	}

}



package visitor;

public class DVD implements Visitable {

	private Integer idDVD;
	private String filme;
	private Double preco;

	@Override
	public void accept(Visitor visitor) {
		visitor.visit(this);
	}

	public DVD() {
	}

	public DVD(Integer idDVD, String filme, Double preco) {
		super();
		this.idDVD = idDVD;
		this.filme = filme;
		this.preco = preco;
	}
```

```
	@Override
	public String toString() {
		return "DVD [idDVD=" + idDVD + ", filme=" + filme
				+ ", preco=" + preço + "]";
	}

	public Integer getIdDVD() {
		return idDVD;
	}

	public void setIdDVD(Integer idDVD) {
		this.idDVD = idDVD;
	}

	public String getFilme() {
		return filme;
	}

	public void setFilme(String filme) {
		this.filme = filme;
	}

	public Double getPreco() {
		return preco;
	}

	public void setPreco(Double preco) {
		this.preco = preco;
	}

}


package visitor;

public class Livro implements Visitable {

	private Integer idLivro;
	private String titulo;
	private String autor;
	private Double preco;

	@Override
	public void accept(Visitor visitor) {
		visitor.visit(this);
	}

	public Livro() {

	}

	public Livro(Integer idLivro, String titulo, String autor,
			Double preco) {
		super();
		this.idLivro = idLivro;
		this.titulo = titulo;
		this.autor = autor;
```

```
			this.preco = preco;
		}

		@Override
		public String toString() {
			return "Livro [idLivro=" + idLivro + ", titulo="
				+ titulo + ", autor=" + autor + ", preco=" + preco + "]";
		}

		public Integer getIdLivro() {
			return idLivro;
		}

		public void setIdLivro(Integer idLivro) {
			this.idLivro = idLivro;
		}

		public String getTitulo() {
			return titulo;
		}

		public void setTitulo(String titulo) {
			this.titulo = titulo;
		}

		public String getAutor() {
			return autor;
		}

		public void setAutor(String autor) {
			this.autor = autor;
		}

		public Double getPreco() {
			return preco;
		}

		public void setPreco(Double preco) {
			this.preco = preco;
		}

	}



	package visitor;

	public interface Visitable {

		public void accept(Visitor visitor);

	}
```

```
package visitor;

public interface Visitor {

	public void visit(Livro livro);

	public void visit(CD cd);

	public void visit(DVD dvd);
}
```

```
package visitor;

public class VisitorCompra implements Visitor {

	private transient Double valorCompra;


	public VisitorCompra() {
		valorCompra = 0.;
	}


	public Double getValorCompra() {
		return valorCompra;
	}


	@Override
	public void visit(Livro livro) {
		// Livro tem desconto de 10%
		valorCompra += livro.getPreco() - (livro.getPreco() * 0.10);
	}

	@Override
	public void visit(CD cd) {
		// CD tem desconto de 5%
		valorCompra += cd.getPreco() - (cd.getPreco() * 0.05);
	}

	@Override
	public void visit(DVD dvd) {
		// Filme não tem desconto
		valorCompra += dvd.getPreco();
	}

}
```

```
package visitor;

public class Main {

	public static void main(String[] args) {

		ControleCompra c = new ControleCompra();

		c.getItensCompra().add(new Livro(1, "Senhor dos Aneis",
				"Tolkien", 100.0));
		c.getItensCompra().add(new Livro(2, "Codigo daVinci",
				"Dan Brown", 50.0));
		c.getItensCompra().add(new CD(3, "Legião Urbana", 40.0));
		c.getItensCompra().add(new DVD(4, "Se Beber não case", 20.0));
		c.getItensCompra().add(new CD(5, "Depois da Terra", 30.0));

		System.out.println("Itens: " + c.getItensCompra());
		System.out.println("Total a pagar: " + c.getTotalPagar());
	}

}
```