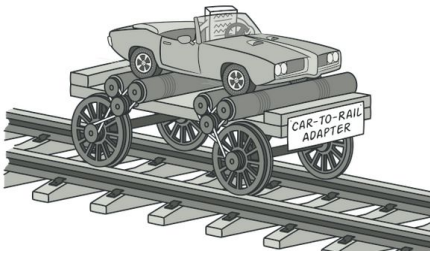
Padrões de Estruturais

Exemplos Práticos

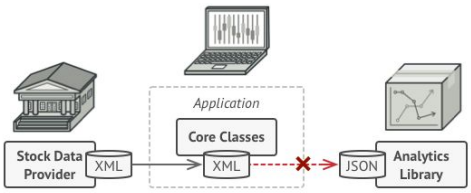
Slides + https://refactoring.guru/design-patterns

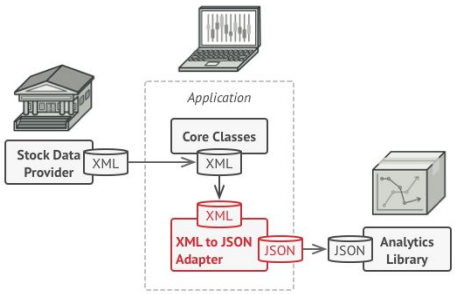
Carina Neves 90451

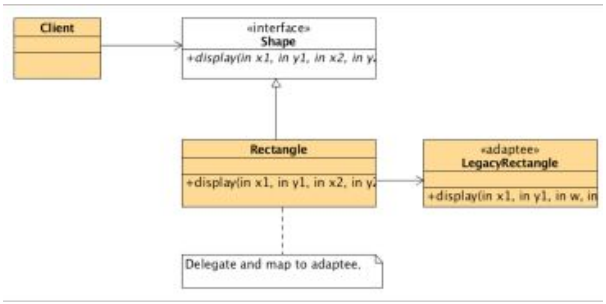
Adapter:



Permite que objetos com interfaces incompatíveis colaborem.



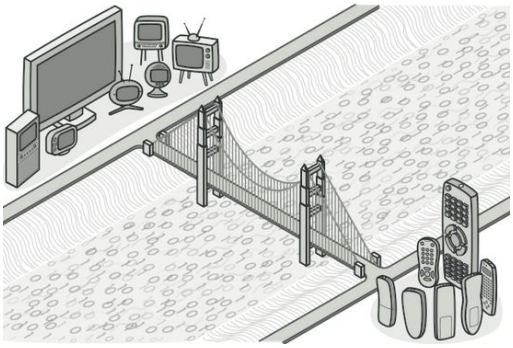




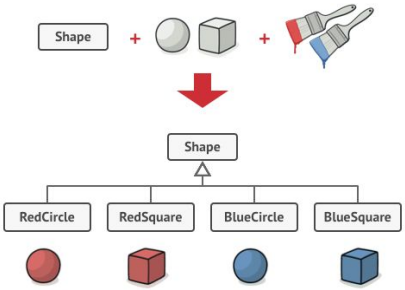

```
interface Shape {
        void draw(int x1, int y1, int x2, int y2);
}

class Rectangle implements Shape {
        public void draw(int x1, int y1, int x2, int
y2) {
        System.out.println("rectangle from (" +
x1 + ',' + y1 + ") to (" + x2 + ',' + y2 + ')');
        }
}

class LegacyRectangle {
        public void draw(int x, int y, int w, int h) {
                System.out.println("old format rectangle at (" + x + ',' + y + ") with width " + w + "
        and height " + h);
        }
}

class OldRectangle implements Shape {
        private LegacyRectangle adaptee = new LegacyRectangle();
        public void draw(int x1, int y1, int x2, int y2) {
                adapter.draw(Math.min(x1, x2), Math.min(y1, y2), Math.abs(x2 - x1),
                        Math.abs(y2 - y1));
        }
}
```

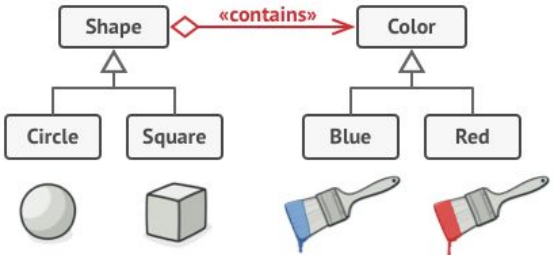
```
public class AdapterDemo2 {
	public static void main(String[] args) {
		Shape[] shapes = { new Rectangle(), new OldRectangle() };
		// A begin and end point from a graphical editor
		int x1 = 10, y1 = 20;
		int x2 = 30, y2 = 60;
		for (int i = 0; i < shapes.length; ++i)
			shapes[i].draw(x1, y1, x2, y2);
	}
}
```

Bridge:

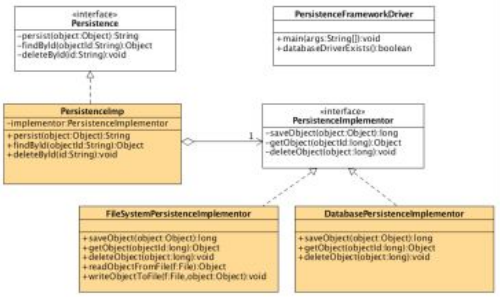


Permite dividir uma classe grande ou um conjunto de classes relacionadas em duas hierarquias separadas - abstração e implementação - que podem ser desenvolvidas independentemente umas das outras.



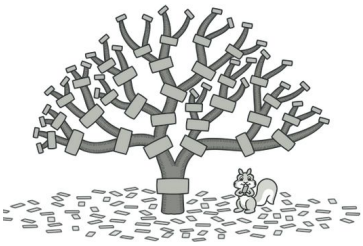


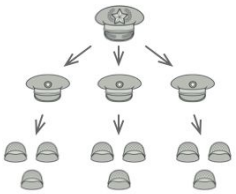

```
public class PersistenceFrameworkDriver {
	public static void main(String[] args) {
		PersistenceImplementor implementor = null;
		if(databaseDriverExists()) {
			implementor = new
		DabatasePersistenceImplementor();
		} else {
			implementor = new
		FileSystemPersistenceImplementor();
		}
		Persistence persistenceAPI = new PersistenceImp(implementor);
		Object o = persistenceAPI.findById("12343755");
```



```
                // do changes to the object ... then persist
                persistenceAPI.persist(o);
                // can also change implementor
                persistenceAPI = new PersistenceImp(
                new DabatasePersistenceImplementor());
                persistenceAPI.deleteById("2323");
        }
}

public class BridgeDemo {
        public static void main(String[] args) {
                Vehicle vehicle = new BigBus(new SmallEngine());
                vehicle.drive();
                vehicle.setEngine(new BigEngine());
                vehicle.drive();
                vehicle = new SmallCar(new SmallEngine());
                vehicle.drive();
                vehicle.setEngine(new BigEngine());
                vehicle.drive();
        }
}
```

Composite:



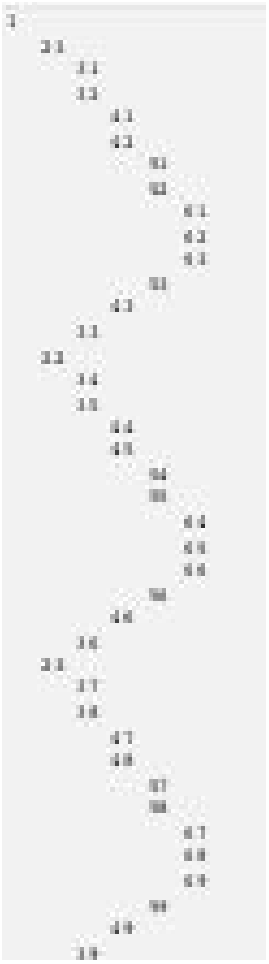


Permite compor objetos em estruturas de árvores e depois trabalhar com essas estruturas como se fossem objetos individuais.

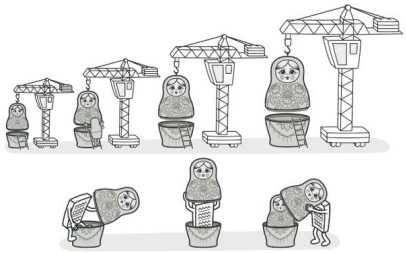
Entity/Product/Box:

```
abstract class Entity {
        protected static StringBuffer indent = new StringBuffer();
        public abstract void traverse();
}

class Product extends Entity {
        private int value;
        public Product(int val) {
                value = val;
        }
        public void traverse() {
                System.out.println(indent.toString() + value);
```

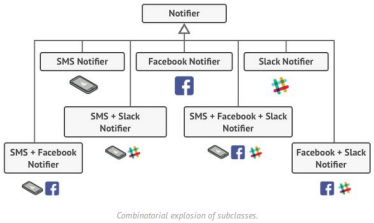
```
        }
}

class Box extends Entity {
        private List<Entity> children = new ArrayList<>();
        private int value;
        public Box(int val) {
                value = val;
        }
        public void add(Entity c) {
                children.add(c);
        }
        public void traverse() {
                System.out.println(indent.toString() + value);
                indent.append(" ");
                for (int i = 0; i < children.size(); i++)
                        children.get(i).traverse();
                indent.setLength(indent.length() - 3);
        }
}

public class CompositeLevels {
        public static void main(String[] args) {
                Box root = initialize();
                root.traverse();
        }
        private static Box initialize() {
                Box[] nodes = new Box[7];
                nodes[1] = new Box(1);
                int[] s = { 1, 4, 7 };
                for (int i = 0; i < 3; i++) {
                        nodes[2] = new Box(21 + i);
                        nodes[1].add(nodes[2]);
                        int lev = 3;
                        for (int j = 0; j < 4; j++) {
                                nodes[lev - 1].add(new Product(lev * 10 + s[i]));
                                nodes[lev] = new Box(lev * 10 + s[i] + 1);
                                nodes[lev - 1].add(nodes[lev]);
                                nodes[lev - 1].add(new Product(lev * 10 + s[i] + 2));
                                Lev++;
                        }
                }
                return nodes[1];
        }
}
```

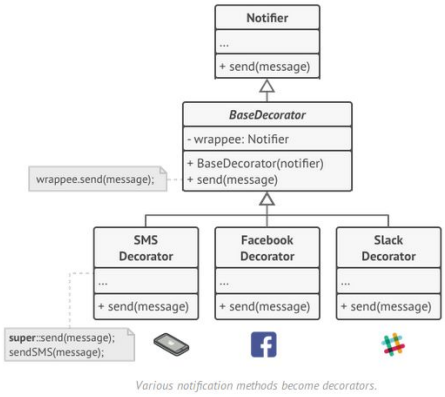


Decorator:



Permite anexar novos comportamentos a objetos colocando esses objetos dentro de objetos especiais que contêm os comportamentos.



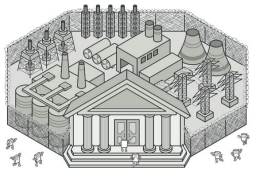


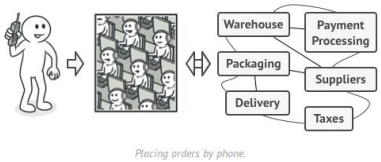

```
interface JogadorInterface {
	void joga();
}

class Jogador implements JogadorInterface {
	private String name;
	Jogador(String n) { name = n; }
	@Override public void joga() {
		System.out.print("\n"+name+" joga ");
	}
}

abstract class JogDecorator implements JogadorInterface {
	protected JogadorInterface j;
	JogDecorator(JogadorInterface j) { this.j = j; }
	public void joga() { j.joga(); }
}

class Futebolista extends JogDecorator {
	Futebolista(JogadorInterface j) { super(j); }
	@Override public void joga() {
		j.joga();
		System.out.print("futebol ");
	}
```

```
        public void remata() { System.out.println("-- Remata!"); }
}

class Xadrezista extends JogDecorator {
        Xadrezista(JogadorInterface j) { super(j); }
        @Override public void joga() {
                j.joga();
                System.out.print("xadrez ");
        }
}

class Tenista extends JogDecorator {
        Tenista(JogadorInterface j) { super(j); }
        @Override public void joga() {
                j.joga();
                System.out.print("tenis ");
        }
        public void serve() { System.out.println("-- Serve!"); }
}

public class PlayTest{
        public static void main(String args[]) {
                JogadorInterface j1 = new Jogador("Rui");
                Futebolista f1 = new Futebolista(new Jogador("Luis"));
                Xadrezista x1 = new Xadrezista(new Jogador("Ana"));
                Xadrezista x2 = new Xadrezista(j1);
                Xadrezista x3 = new Xadrezista(f1);
                Tenista t1 = new Tenista(j1);
                Tenista t2 = new Tenista(
        new Xadrezista( new Futebolista( new Jogador("Bruna"))));

        JogadorInterface lista[] = { j1, f1, x1, x2, x3, t1, t2 };
        for (JogadorInterface ji: lista)
                ji.joga();
        }
}
```

```
Rui joga
Luis joga futebol
Ana joga xadrez
Rui joga xadrez
Luis joga futebol xadrez
Rui joga ténis
Bruna joga futebol xadrez ténis
```

Facade:

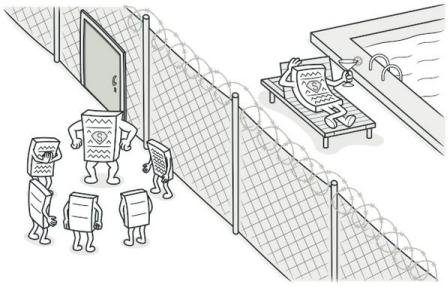




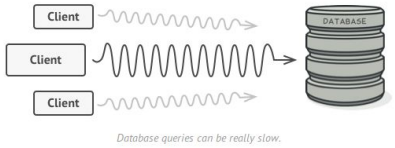
Fornece uma interface simplificada para uma biblioteca, uma estrutura ou qualquer outro conjunto complexo de classes

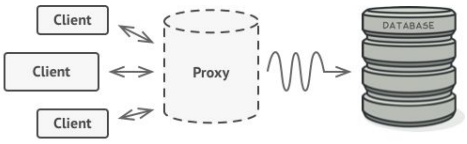
```
// ...
class TravelFacade {
        private HotelBooker hotelBooker;
        private FlightBooker flightBooker;
        private LocalTourBooker tourBooker;
        public void getFlightsAndHotels(City dest, Date from, Data to) {
                List<Flight> flights = flightBooker.getFlightsFor(dest, from, to);
                List<Hotel> hotels = hotelBooker.getHotelsFor(dest, from, to);
                List<Tour> tours = tourBooker.getToursFor(dest, from, to);
                // process and return
        }
}

public class FacadeDemo {
        public static void main(String[] args) {
                TravelFacade facade = new TravelFacade();
                facade.getFlightsAndHotels(destination, from, to);
        }
}
```

Proxy:



Permite fornecer um substituto ou espaço reservado para outro objeto. Um proxy controla o acesso ao objeto original, permitindo que você execute algo antes ou depois da solicitação chegar ao objeto original.





Resumo das funcionalidades:

Adapter- Corresponder interfaces de diferentes classes
Bridge- Separa a interface de um objeto de sua implementação
Composite- Uma estrutura de árvore de objetos simples e compostos

Decorador- Adicionar responsabilidades aos objetos dinamicamente
Façade Uma única classe que representa um subsistema inteiro
Flyweight- Uma instância refinada usada para compartilhamento eficiente
Proxy- Um objeto representando outro objeto