Most (Bridge)

Most jest to rodzaj strukturalnego wzorca projektowego który konwertuje jeden rodzaj interfejsu na inny.
Został zaprojektowany żeby odseparować interfejs klasy od jej implementacji. Zaletą tego jest to, iż można zmienić implementację bez potrzeby wprowadzania zmian w kodzie, który korzysta z klasy.

```
public interface Cook {
        public void cookDinner();
}

class CookImpl {
        public void cookDinner() {
        // implementacja metody
        }
 }

public interface Dish() {
        public void prepare();
}

class Spaghetti implements Dish {
        private Cook cook;

        public Spaghetti( Cook cook) {
                this.cook = cook;
        }

        public void prepare() { cook.cookDinner();]
}

class BridgePattern {
        public static void main(String[] args) {
        Dish[] dish = new Dish[2];
        dish[0] = new Spaghetti( new Cook());
        dish[0].prepare();
        }
}
```

Składowe wzorca:

- abstrakcja, która definiuje interfejs klasy mostu
- implementator, który definiuje interfejs klasy implementującej
- implementatory, którymi są klasy implementujące
- abstrakcje pierwotne (poprzez dziedziczenie lub kompozycję)

Zalety stosowania wzorca Bridge:

1. Można w bardzo łatwy sposób ukryć szczegóły implementacyjne
2. Można niezależnie rozszerzać klasy implementujące i klasy mostu
3. Wzorzec pozwala zachować stały interfejs dla programu klienta podczas wprowadzania zmian

Dekorator (Decorator)

Dekorator umożliwia zmianę właściwości indywidualnych obiektów bez potrzeby tworzenia nowej klasy pochodnej.

Wzorzec Dekorator stosujemy gdy chcemy dodać w sposób jawny nowe zachowanie do obiektu

```
public interface Signature {
        public boolean addSignature();
}

class SignatureImpl {
        public boolean addSignature() {
        // implementacja metody
        }
}

class SignatureDecorator implements Signature {
        private SignatureImpl delegate;

        public boolean addSignature() {
        // tu możemy dodać rozszerzoną implementacyjne
                signature.addCsrfProtection();
                return delegate.addSignature();
        }
}
```

Zalety stosowania wzorca Dekorator:

1. Umożliwia elastyczny sposób dodawania nowej funkcjonalności do klasy. Pozwala wprowadzać modyfikacje bez potrzeby tworzenia rozbudowanej hierarchii dziedziczenia

Wady stosowania wzorca Dekorator:

1. Trudniejsza kontrola typu obiektów
2. Wzorzec może wprowadzać w system wiele małych obiektów, które w późniejszym czasie mogą stać się trudne do utrzymania przez programistę

Kompozyt (Composite)

Kompozyt jest kolekcją obiektów, z których każdy może być kompozytem lub pojedynczym obiektem. Można go użyć do tworzenia hierarchii związków lub drzewiastych struktur danych. Używa się go w celu traktowania całej hierarchii (struktury) jako jednego obiektu.

W założeniu, wzorzec kompozyt ma uniemożliwiać tworzenie złożonych drzew z różnych, powiązanych ze sobą klas

Kompozyt składa się z następujących części :

- Component - czyli klasa abstrakcyjna lub interfejs, który reprezentuje pojedynczy obiekt (*leaf*)
- Leaf - typ prosty
- Composite - przechowuje obiekty proste

```
public interface Component {
	public void action();
}
class Leaf implements Component {
	public void action() {
	// implementacja
	}
}
class Composite implements Component{
	private List<Component> components = new ArrayList<Component>();

	public void action() {
	// implementacja
	}

	public boolean addComponent(Component component) {
		return components.add(component);
	}

	public boolean removeComponent(Component component) {
		return components.remove(component);
	}
 }
class Client {
	public static void main(String[] args) {
		Leaf leaf = new leaf();
		Composite composite = new Composite();
		composite.addComponent(leaf);
	}
}
```

Zalety stosowania wzorca Kompozyt:

1. Wzorzec pozwala w łatwy sposób dodawać nowe rodzaje obiektów
2. Pozwala zdefiniować klasę zawierającą hierarchię prostych i bardziej złożonych obiektów w taki sposób, że będą one miały taki sam interfejs.
3. Upraszcza kod klientów