Agnieszka Iżuk
Numer albumu: 43120
Kierunek: Magento PHP Developer
Link GitHub: https://github.com/student43120/Wzorce-projektowe.git

Spis Treści

- BARRIER (BARIERA)
- POLECENIE (COMMAND)
- STRATEGIA (STRATEGY)
- STAN (STATE)
- WZÓR(TEMPLATE)

Wzorce projektowe odnoszą się do wielu aspektów życia tj. sfery służbowej lub prywatnej, czego najprostszym przykładem jest osobowość Christophera Alexander ’a. Architekt jako pierwszy zbadał wzorce występujące w budynkach i środowiskach oraz opracował „język wzorców" do ich generowania. Struktura każdego ze wzorców sprowadza się do określenia problemu, następnie zebrania informacji w jaki sposób można go rozwiązać oraz ostatecznej decyzja wyboru najkorzystniejszego i postepowania według przyjętego schematu. W poniżej pracy skupimy się na metodach, które wykorzystywane są w programowaniu, czyli mówiąc szerzej w obszarze IT.

Programowanie obiektowe opiera się na operacjach i procedurach, które przeprowadzane są na obiektach. Obiekt uruchamia metodę, kiedy otrzyma żądanie (lub komunikat) od klienta. Zgłoszenie żądania to jedyny sposób na zmuszenie obiektu do uruchomienia operacji. Z kolei jej wywołanie to jedyny sposób na zmodyfikowanie wewnętrznych danych obiektu. Tak “zakapsułkowany” obiekt zostaje

zabezpieczony i nie można bez-pośrednio uzyskać dostępu do stanu obiektu, a jego reprezentacja jest niewidoczna poza nim. W projektowaniu obiektowym trudność sprawia podział systemu na obiekty, ponieważ musimy brać pod uwagę m.in: kapsułkowanie, szczegóło-wość, zależności, elastyczność, wydajność, możliwość powtórnego wykorzystania. Wszystkie te aspekty wpływają na podział systemu i często proponowane rozwiązania są opozycyjne względem siebie. Dodatkową komplikacją jest fakt, że obiekty mogą znacznie różnić się między sobą pod względem wielkości i liczby.

Interfejsy są podstawowym elementem systemów obiektowych, dlatego nie można zażądać od obiektu wy-konania operacji z pominięciem interfejsu, nie określa on jednak implementacji obiektu. Dodatkowym mankamentem jest fakt, że na róż-nych obiektach żądania mogą być realizowane w inny sposób. Oznacza to, że dwa obiekty o zupełnie innej implementacji mogą mieć identyczny interfejs, dlatego wzorce projektowe określają też relacje między interfejsami.

Przykłady użycia wzorców projektowych:

- **Typ: Behawioralny**

Wzorzec strategia bywa użyteczny, gdy klasa może zachowywać się różnie w zależności od potrzeb. Gdy wykonanie metody wymaga użycia różnych algorytmów, możemy zapisać je w postaci klas i używać dokładnie jednego z nich w danym czasie.

- **Typ: Strukturalny**

Wzorzec dekorator stosowany jest tam, gdzie dziedziczenie nie jest optymalnym sposobem rozszerzania funkcjonalności klasy. Polega na dołączeniu nowych atrybutów poprzez “opakowanie” obiektu bazowego obiektem zwanym dekoratorem.

- **Typ: Konstrukcyjny**

Celem wzorcu “Fabryka” jest dostarczenie interfejsu dla klas odpowiedzialnych za tworzenie konkretnego typu obiektów.

Główne zadania spełniane przez wzorce to m.in: przedstawianie zarysu powiązań pomiędzy klasami i obiektami, ułatwianie tworzenia, utrzymania i edycji kodu źródłowego, zmniejszanie kosztów utrzymania i rozwoju projektu oraz umożliwiają wizualizację rozwiązania problemu przed implementacją .

Podsumowując wzorców nie należy używać bez zastano-wienia. Często pozwalają one uzyskać elastyczność, możliwość wprowadzania zmian oraz zwiększają prawdopodobieństwo reużywalności wytworzonego rozwiązania. Musimy mieć, jednak na uwadze, że istnieje ryzyko skomplikowania projekt lub pogorszenia wydajności. Wzorce projektowe należy stosować tylko wtedy, kiedy większa elastyczność jest naprawdę potrzebna lub w przypadku braku doświadczenia developerów tj. team składający się z juniorów.

Opis kodu na przykładzie konkretnych wzorców

Adapter

Wspiera multitasking obiektów o niezgodnych interfejsach. Opakowuje pewną funkcjonalność spełniając konkretny interfejs.

```
public class Main {

  public static void main(String[] args) {

    Walkable minotaur = new Minotaur(
        "Wojowniku! Na Twojej drodzę staje Minotaur!",
        "Minotaur",100,50);

    // Brak możliwości wywołania metody walk dla centaura,
    // klient może wywołać jedynie metode galop dla tego obiektu, trzeba zmienić kod klienta
    Centaur centaur = new Centaur();

    klientWywolujePrzeciwnikaNaPlansze(minotaur);
    klientWywolujePrzeciwnikaNaPlansze(centaur);
  }

  private static void klientWywolujePrzeciwnikaNaPlansze(Object creature) {
    if (creature instanceof Walkable){
    System.out.println(((Walkable) creature).walk());}
    else {
      System.out.println(((Centaur) creature).gallop());
    }
  }
}
```

Po zmianie kodu mamy już dostęp do dwóch metod :

```
public class Main {
```

```
  public static void main(String[] args) {
    Walkable minotaur = new Minotaur(
        "Wojowniku! Na Twojej drodzę staje Minotaur!",
        "Minotaur",100,50);
    Walkable centaur = new Adapter(new Centaur());
    klientWywolujePrzeciwnikaNaPlansze(minotaur);
    klientWywolujePrzeciwnikaNaPlansze(centaur);
  }
  private static void klientWywolujePrzeciwnikaNaPlansze(Walkable
creature) {
    System.out.println(creature.walk());
  }
}
```

Reprezentacja dwóch niezgodnych interfejsów są odmienne mnistyczne stworzy, czyli minotaur i centaur.

Builder

Upraszcza budowanie drzewa obiektów-kompozytów oraz do zachowania niezmienności procesu wytwarzania obiektu z zachowaniem jego różnorodności. Seria tworzenia nowych obiektów i umieszczania ich w innych staje się łatwiejsza dzięki temu wzorcowi, ponieważ ukrywa on budowanie kompozytu za interfejsem. Drugim zastosowaniem wzorca, który przedstawiłam w kodzie jest budowanie niezmiennych obiektów, eliminując niebezpieczne dla wątków settery. Typowa implementacja budowniczego to statyczna wewnętrzna klasa.

Poniżej tworzymy elementry, z których pwstaje zabawka dla dzieci:

```
TeddyCat firstTeddyCat = new TeddyCat();
    firstTeddyCat.setTeddyCatHead("Fluffy Head");
    firstTeddyCat.setTeddyCatTorso("Fluffy Toros");
    firstTeddyCat.setTeddyCatTail("Fluffy Tail");
    firstTeddyCat.setTeddyCatPaws("Fluffy Paws");
    firstTeddyCat.setTeddyCatFur("Fluffy Fur");

 System.out.println("TeddyCat Make");
    System.out.println("TeddyCat Head Type: " +
firstTeddyCat.getTeddyCatHead());
    System.out.println("TeddyCat Torso Type: " +
firstTeddyCat.getTeddyCatTorso());
    System.out.println("TeddyCat Tail Type: " +
firstTeddyCat.getTeddyCatTail());
    System.out.println("TeddyCat Paws Type: " +
firstTeddyCat.getTeddyCatPaws());
    System.out.println("TeddyCat Fur Type: " +
firstTeddyCat.getTeddyCatFur());
```

Aktualnie wytwarzamy tylko jeden rodzaj kota ustawiając jego parametry ręcznie.Klient wie wszystko o specyfikacji obiektu - zmiana obiektu wymaga zmiany w kodzie klienta, aby poprawić powyższy proces proponuje stworzenie “FluffyTeddyCatBuilder”

```
public static void main(String[] args) {
    TeddyCatBuilder fluffyStyleTeddyCat = new
FluffyTeddyCatBuilder();
    TeddyCatMaker teddyCatMaker = new
TeddyCatMaker(fluffyStyleTeddyCat);
    teddyCatMaker.makeTeddyCat();

    TeddyCat firstTeddyCat = teddyCatMaker.getTeddyCat();
    System.out.println("TeddyCat Make");
    System.out.println("TeddyCat Head Type: " +
firstTeddyCat.getTeddyCatHead());
```

```
        System.out.println("TeddyCat Torso Type: " +
firstTeddyCat.getTeddyCatTorso());
        System.out.println("TeddyCat Tail Type: " +
firstTeddyCat.getTeddyCatTail());
        System.out.println("TeddyCat Paws Type: " +
firstTeddyCat.getTeddyCatPaws());
        System.out.println("TeddyCat Fur Type: " +
firstTeddyCat.getTeddyCatFur());

public class FluffyTeddyCatBuilder implements TeddyCatBuilder {

  private final TeddyCat teddyCat;

  public FluffyTeddyCatBuilder() {
    this.teddyCat = new TeddyCat();
  }

  @Override
  public void buildTeddyCatHead() {

    teddyCat.setTeddyCatHead("Fluffy Head");

  }

  @Override
  public void buildTeddyCatTorso() {

    teddyCat.setTeddyCatTorso("Fluffy Torso");

  }

  @Override
  public void buildTeddyCatTail() {

    teddyCat.setTeddyCatTail("Fluffy Tail");

  }

  @Override
  public void buildTeddyCatPaws() {
```

```
        teddyCat.setTeddyCatPaws("Fluffy Paws");
    }

    @Override
    public void buildTeddyCatFur() {
        teddyCat.setTeddyCatFur("Fluffy Fur");
    }

    @Override
    public TeddyCat getTeddyCat() {
        return this.teddyCat;
    }
}
```

Chain of responsibility

Połączenie wywołań konstruktorów w łańcuch,w celu nie powtarzania fragmentów kodu. Stworzenie konstruktora zbierającego, który jest wywoływany przez inne konstruktory. Każdy kolejny obiekt w danej sekwencji obsłużyć wyowłanie oraz decyduje, czy przekazać je kolejnemu w łańcuchu, czy może je przerwać.

W naszym przypadku mamy do czynienia z wytowrzeniem doktora, jako robota sprawdzającego stan pacjetna.

Sprawdzamy stan poszczególnych organów w tym brzucha i głowy. Wydajemy werdykt odnośnie stanu zdrowia pacjęta.
W przypadku, gdy nie wynikła potrzeba sprawdzenia głowy lub klatki nie podejmujemy próby jej dodatkowego obejrzenia.

```
class Head extends healthChecker,
{
```

```
public function check(patientWellBeing $patient)
{
if (!$patient->patientHeadPain) {
echo "Head wasn't checked \n"; ---> nie było potrzeby
}

$this->next($patient);
}
}
public function next(patientWellBeing $patient),
{
if ($this->doctor) {
$this->doctor->check($patient);    --> sprawdzamy pacjenta
}
}
}

ublic function check(patientWellBeing $patient),
{
if (!$patient->patientChestpain) {
echo "in Chest we don't see any changes \n";   --> klatka piersiowa jest w normie
}

$this->next($patient);
}
```

Command

Wykorzystywane w celu eliminacji długich instrukcji warunkowych w tzw. Dispatcherach tj. ukrycie logiki obsługi poleceń w obiektach poleceń.

Znajdujemy się na głosowaniu, w którym mozemy podjąć decyzje, czy głosowanie przebiegło zgodnie ze sztuką lub nie:

1.Głosowanie za

```
public function countAdvocates(): int,
{
$sums = array_count_values($this->reactionStatistics);

return $sums[self::ADVOCATE_VOICE] ?? 0;
}
```

2.Głosowanie przeciw

```
public function countAgainsts(): int
{
$sums = array_count_values($this->reactionStatistics);

return $sums[self::AGAINST_VOICE] ?? 0;
}
```

3.Podjęcie dezycji o stanie głosowania:

```
private function shouldUndo(string $electoralId, string $reactionVoice): bool
{
return !empty($this->reactionStatistics[$electoralId])
&& ($this->reactionStatistics[$electoralId] === $reactionVoice);
}
```

Composite

Drzewo obiektów jako pojedynczy obiekt. Przykładem kompozytu jest struktura elementów pliku XML lub klauzula WHERE dla bazy danych zapisana w systemie obiektowym.

Tworzymy w oparciu o klase category :

```
class CompositeCategory extends Category,
{
private $categories = [];
```

```
public function __construct(string $name, array $categories)
{
parent::__construct($name);
$this->categories = $categories;
}

public function getCategories()
{
return $this->categories;
}
}
$categories = [,
new CompositeCategory('Konsola', [
new Category('Xbox'),
new Category('Playstation'),
new Category('Sony')
]),
new CompositeCategory('Konsola Przenośna', [
new Category('PSP'),
new Category('Nintendo')
]),
new CompositeCategory('Game Pass', [
new CompositeCategory('Game Pass', [new Category('XBOX Game Pass'), new Category('PlayStation Plus')])
]),
];
```

Decorator

Dodanie dodatkowej funkcjonalności do klasy bez jej modyfikacji. Klasa rozszerzana o nową funkcjonalność nie jest świadoma tego, podobnie i użytkownicy tej klasy. Wzorzec jest zgodny z zasadą single responsibility principle.

Do rustynowej kontroli laptopa w serwisie, podczas wykrycia wady oferujemy jej naprawe:

```
class Overview implements laptopService,
{
public function getCost()
{
return 20;
}

public function getListtoDo()
{
return 'Laptop health check';
}
}

$service = new batteryChange();,
$service = new keyboardChange($service);
```

Funkcją publiczną wyliczamy koszt usługi : public function getCost()

```
class batteryChange implements laptopService,
{
protected $laptopService;

public function __construct(laptopService $laptopService)
{
$this->laptopService = $laptopService;
}


public function getCost()
{
return 5 + $this->laptopService->getCost();
}
```

Facade

Ukrycie długiego lub nieprzejrzystego podsystemu za interfejsem uporządkowanym i/lub uproszczonym.

```
class PlatformaSteam,
{
function __construct(){
}
public static function PlatformaSteam()
{
$local_this = new PlatformaSteam();
return $local_this;
}
public static function main(&$args)
{
echo "Steam","\n";
echo "Terminal obslugi Steam","\n";
}
```

Obszerna ilość kodu :

```
class SteamFacade,
{
function __construct(){
$this->numerKonta = 0;
$this->identyfikator = 0;
$this->kontoId = NULL;
$this->identyfikatorCheck = NULL;
$this->przyznajGre = NULL;
$this->platformaSteam = NULL;
}
private $numerKonta;
private $identyfikator;
public $kontoId;
public $identyfikatorCheck;
public $przyznajGre;
```

```
public $platformaSteam;
public static function SteamFacade($newNumerKonta, $newIdentyfikator)
{
$local_this = new SteamFacade();
$this->numerKonta = $newNumerKonta;
$this->identyfikator = $newIdentyfikator;
$this->platformaSteam = PlatformaSteam::PlatformaSteam();
$this->kontoId = KontoId::KontoId();
$this->identyfikatorCheck = IdentyfikatorCheck::IdentyfikatorCheck();
$this->przyznajGre = PrzyznajGre::PrzyznajGre();
return $local_this;
}
```

Factory

Sposób na wyeliminowanie logiki tworzenia obiektów rozrzuconej pomiędzy dwie klasy.

Fabryka do przygotowywania kanapek:

```
class KanapkaFactory implements Factory {,

public static function zrobKanapka (KanapkaItem $kanapka): Kanapka {

switch ($itemKanapka->getTyp()) {
case KanapkaItem::TYP_WEGE:
return new WegeKanapka($itemKanapka);
break;
case KanapkaItem::TYP_MIESO:
return new MiesoKanapka($itemKanapka);
break;
default:
throw new InvalidArgumentException("Wrong file type provided: {$itemKanapka->getTyp()}");
```

```
break;
}
}
}
```

Interpreter

Zapisanie prostego języka w postaci obiektów-kompozytów. Idealny dla prostych zapytań do bazy danych, gdy nie ma z góry zdefiniowanych, stałych zapytań.

```
interface WordInterface
{
   public function runRequest(termOfRequest $termOfRequest);
}


class termOfRequest
{
   private $request;
   private $index  = 0;


   public function __construct($command)
   {
      $this->request = explode(' ', trim($command));
   }


   public function next()
   {
      $this->index++;
      return $this;
   }
```

```
public function getRequest()
{
    if (!array_key_exists($this->index, $this->request)) {
        return null;
    }
    return $this->request[$this->index];
}
```

Observer

Powiadamianie obserwatorów o zajściu zdarzenia. Ma na celu wyeliminowanie zależności pomiędzy klasą powiadamiającą i powiadamianą.

```
interface webInterface,
{
public function attach($observable);

public function detach($observer);

public function notify(); -> powiadomienie o zajściu
}


},

public function detach($index)
{
unset($this->observers[$index]);
}

public function notify()
```

```
{
foreach ($this->observers as $observer) {
$observer->catch();
}
}

public function upload()
{
$this->notify();
```

Singleton

Zapewnienie istnienia tylko jednego obiektu danej klasy, gdy tworzenie wielu egzemplarzy byłoby nieefektywne czasowo i pamięciowo.

```
class Counter,
{
function construct(){
$this->currentValue = 0;
}
private $currentValue;
public static function Counter() -->nasza funkcja licznikowa
{
$local_this = new Counter();
return $local_this;
}
public function getCurrentValue()
{
return $this->currentValue;
}
public function add()
{
```

```
$this->currentValue = $this->currentValue + 1;  ->przyrost o 1
}
}
Main::main($argv);
```

State

Wykorzystanie prostych obiektów stanu zamiast wyrażeń warunkowych w celu uproszczenia logiki zmian stanu oraz poprawie czytelności.

```
interface CarStateInterface,
{
public function doorOpen(): doorOpen;  -> otweranie drzwi od auta
public function doorClose(): doorClose;  ->zamykanie drzwi od auta
public function drive(): Drive;  ->jazda
public function stop(): Stop;   -->zatrzymanie
}

class Car,
{
protected $state;

public function getState(): Carstate   -->stan pjazdu
{
return $this->state;
}
```

Strategy

Wzorzec strategii jest przydatny, gdy istnieje zestaw powiązanych algorytmów,a obiekt musi mieć możliwość dynamicznego wybierania algorytmu, który odpowiada jego bieżącym potrzebom. Wzorzec stan

jest implementowany jako interface lub klasa abstrakcyjna. Zawsze przed zastosowaniem dziedziczenia warto pomyśleć, czy wzorzec strategii nie będzie lepszym rozwiązaniem.

Poniżej kod przedstawia proces rejestracji do trzech różnych destynacji serwisu, bazy danych oraz strony internetowej :

```
Interface Register,
{
public function registration($data);
}

class registrationToservice implements Register
{

public function registration($data)
{
echo "{$data}: registration to the service \n";
}
}

class registrationToDatabase implements Register
{
public function registration($data)
{
echo "{$data}: registration to the database \n";
}
}

class registrationToWebService implements Register
{
public function registration($data)
{
echo "{$data}: registration to the Web service \n";
}
}
```

```
class App
{
public function registration($data, register $register)
{
$register->registration($data);
}
}

$app = new App();
$app->registration('Info', new registrationToservice());
```

Template Method

Umieszczenie części wspólnych kilku klas w abstrakcyjnej klasie bazowej i pozostawienie części rozłącznych w klasach ją rozszerzających. Uwaga - jednym z niebezpieczeństw tego wzorca jest problem ułomnej klasy bazowej. Zajrzyj do książki Holub on patterns lub innych publikacji Allena Holuba.

Poniżej jako przykład przedstawiamy proces wydruku obrazka, na którym jest świeczka

- koloru zielonego.

```
public function getColor()
{
return "Candel is Green";
}
```

- koloru różowego

```
public function getColor(),
{
return "Candel is Pink";
}
```

- koloru czerwonego

```
public function getColor(),
{
return "Candel is Red";
}
```

```
abstract class candlePhoto,
{
public abstract function getProcess();
public abstract function getColor();
public abstract function getHeight();
public abstract function getWidht();
public abstract function loadPhoto();
public abstract function getProcessEnd();
```

Wzorce projektowe nie są wynajdywane, ale odkrywane i wdrażane w rzeczywistość.Świetny sposób rozwiązania jakiegoś problemu nie jest wzorcem projektowym.Wzorzec musi być zastosowany wielokrotnie, niejako pojawić się w codziennej praktyce projektowej, potwierdzić swoje znaczenie. W IT jest dostrzegany i właśnie odkrywany ponownie. Elastyczny schemat, choć trwały.