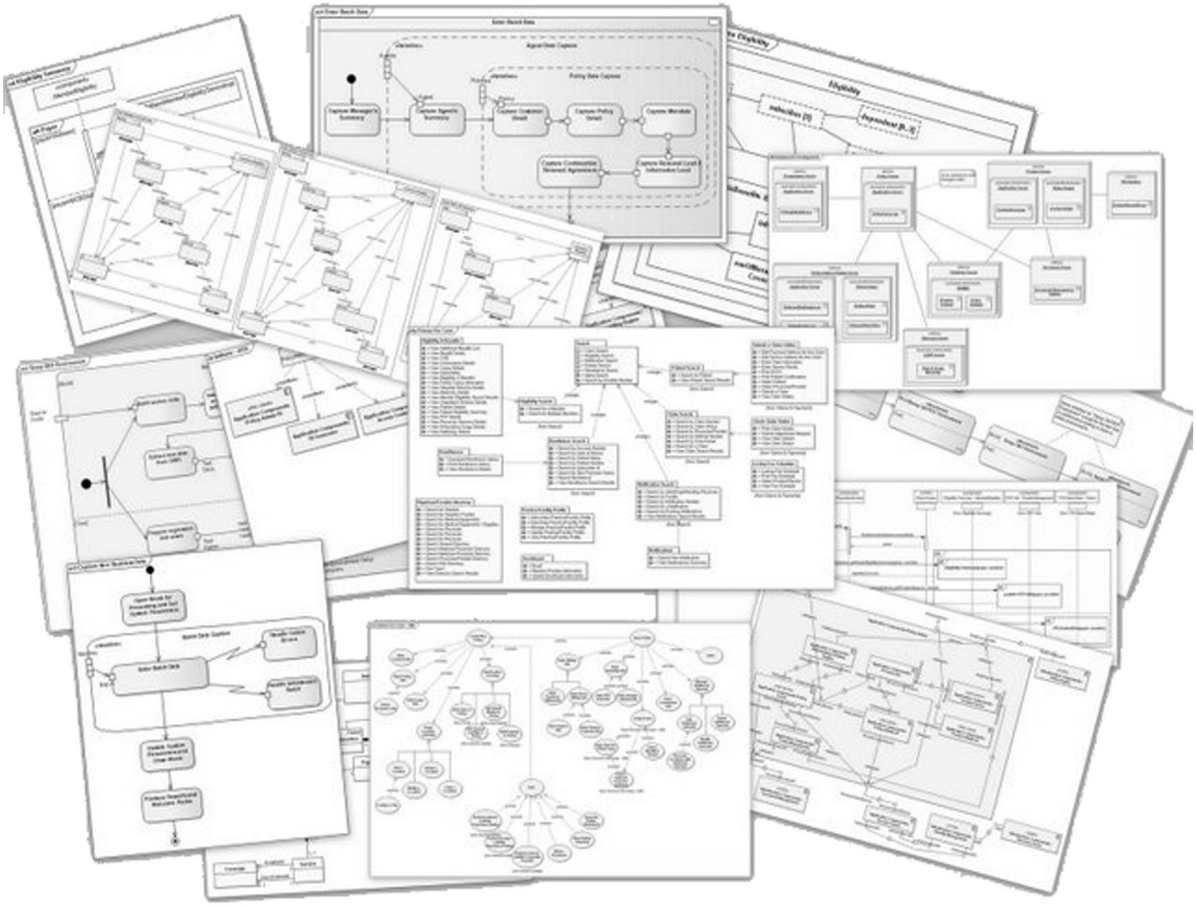
< ŁUKASZSOCHA />
WEB DEVELOPER

Wzorce projektowe
w aplikacjach webowych



opracował: Łukasz Socha

Spis treści

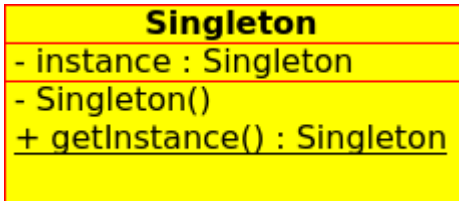
Wstęp

Publikacja ta powstała na podstawie cyklu artykułów o wzorcach projektowych publikowanych na www.lukasz-socha.pl. Moim celem było omówienie najpopularniejszych i najbardziej przydatnych wzorców projektowych przy tworzeniu aplikacji webowych. Starałem się zaprezentować jak najprostsze, a zarazem praktycznie, przykłady wykorzystania danego wzorca. Zakładam, że czytelnik zna podstawy PHP i rozumie ideę programowania obiektowego ;)

Grafika wykorzystana na stronie tytułowej pochodzi z Wikipedii:
http://pl.wikipedia.org/w/index.php?title=Plik:UML_Diagrams.jpg&filetimestamp=20090205143517

1. Singleton

Singleton jest jednym z najprostszych wzorców projektowych. Jego celem jest ograniczenie możliwości tworzenia obiektów danej klasy do jednej instancji oraz zapewnienie globalnego dostępu do stworzonego obiektu – jest to obiektowa alternatywa dla zmiennych globalnych.



Listing 1.1: Diagram klasy wzorca Singleton

Singleton implementuje się poprzez stworzenie klasy, która posiada statyczną metodę ***getInstance()***. Metoda ta sprawdza, czy istnieje już instancja tej klasy, jeżeli nie – tworzy ją i przechowuje jej referencję w prywatnym polu. Aby uniemożliwić tworzenie dodatkowych instancji, konstruktor klasy deklaruje się jako prywatny lub chroniony.

1.2 Przykładowa implementacja

Listing 1.2:

```
<?php
class Singleton {
    private static $instance;
    private function __construct() {}
    private function __clone() {}
    public static function getInstance() {
        if(self::$instance === null) {
            self::$instance = new Singleton();
        }
        return self::$instance;
    }
}
$singleton = Singleton::getInstance();
?>
```

1.3 Przykład z życia wzięty

Rozważmy klasę zawierającą konfigurację aplikacji.

Listing 1.3:

```
<?php
class Config {
    private static $instance;
    private $config = array(
        "login"     =>  "mojlogin",
        "password"  =>  "haslo",
        "language"  =>  "pl"
        );
    private function __construct() {}
    private function __clone() {}
    public static function getInstance() {
        if(self::$instance === null) {
            self::$instance = new Config();
        }
        return self::$instance;
    }
    public function setLanguage($lang) {
        $this->config["language"] = $lang;
    }
    public function getLanguage() {
        return $this->config["language"];
```

```
    }
}
// testy
$conf1 = Config::getInstance();
echo $conf1->getLanguage(); // wyswietla "pl"
$conf2 = Config::getInstance();
$conf2->setLanguage("en");
echo $conf1->getLanguage(); // wyswietla "en"
?>
```

Klasa ***Config*** zawiera tablicę z podstawowymi ustawieniami aplikacji. Dzięki zastosowaniu ***Singletona*** zmiana ustawień w jednym miejscu (np. zmiana języka na stronie przez użytkownika) jest „widoczna” w każdym miejscu aplikacji. Ponadto można łatwo zaprojektować klasę z konfiguracją, tak by było można łatwo dodawać kolejne ustawienia w miarę potrzeb.

1.4 Zalety i wady

Zalety:

- Pobieranie instancji klasy jest niewidoczne dla użytkownika. Nie musi on wiedzieć, czy w chwili wywołania metody instancja istnieje czy dopiero jest tworzona.
- Tworzenie nowej instancji zachodzi dopiero przy pierwszej próbie użycia.
- Klasa zaimplementowana z użyciem wzorca singleton może samodzielnie kontrolować liczbę swoich instancji istniejących w aplikacji.

Wady:

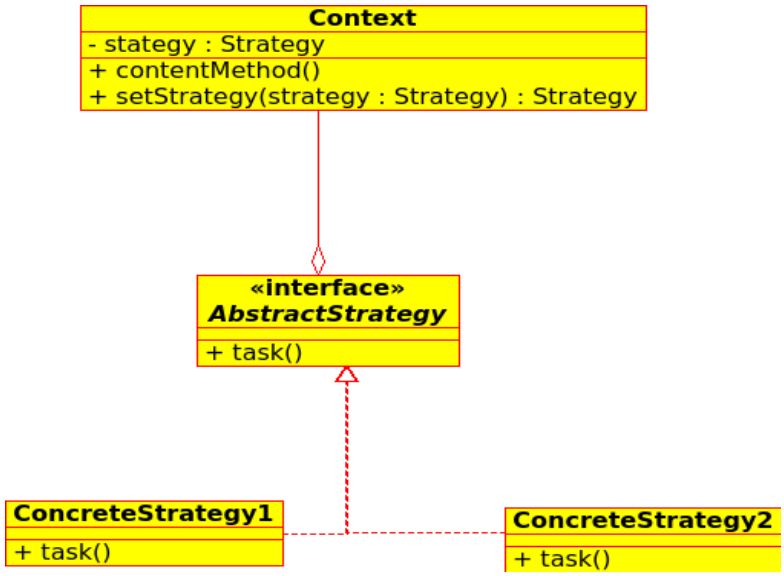
- Brak elastyczności, ponieważ już na poziomie kodu, na „sztywno” określana jest liczba instancji klasy.
- Utrudnia testowanie i usuwanie błędów w aplikacji.

1.5 Zastosowanie

Programując w PHP używa się wzorca Singleton do przechowywania konfiguracji aplikacji oraz utrzymania połączenia z bazą danych. Jednak, warto pamiętać o wadach tego wzorca i korzystać z niego rozważnie. Zbyt częste stosowanie wzorca Singleton pogarsza przejrzystość kodu.

2. Strategy

Strategia jest wzorcem projektowym, który definiuje rodzinę wymiennych algorytmów i kapsułkuje je w postaci klas. Dzięki temu umożliwia wymienne stosowanie każdego z nich w trakcie działania programu.



Listing 2.1: Diagram klas wzorca Strategy

2.2 Przykładowa implementacja

Listing 2.2:

```
<?php

interface AbstractStrategy{

    function task();

}

class ConcreteStrategy1 implements AbstractStrategy{
```

```
    public function task() {
        echo "Strategy 1";
    }
}
class ConcreteStrategy2 implements AbstractStrategy{
    public function task() {
        echo "Strategy 2";
    }
}
class Context{
    private $strategy;

    public function setStrategy(AbstractStrategy $obj) {
        $this->strategy=$obj;
    }
    public function getStrategy() {
        return $this->strategy;
    }
}
// testy
$obj = new Context();
$obj->setStrategy(new ConcreteStrategy1);
$obj->getStrategy()->task(); // wyswietla „Strategy 1”
$obj->setStrategy(new ConcreteStrategy2);
$obj->getStrategy()->task(); // wyswietla „Strategy 2”
```

```
?>
```

We wzorcu tym definiujemy wspólny interfejs dla wszystkich strategii. Następnie, w poszczególnych klasach implementujemy metody z konkretnymi już algorytmami. Za pomocą klasy ***Context*** możemy łatwo zmieniać używaną strategię w trakcie działania aplikacji.

2.3 Przykład z życia wzięty

Przypuśćmy, że tworzymy sklep internetowy oferujący swoje usługi w kilku państwach. Jak wiadomo prawo podatkowe znacząco różni się w poszczególnych krajach. Powstaje problem naliczenia odpowiedniego podatku dla klientów pochodzących z odmiennych państw. Jak to rozwiązać? Wybrać odpowiednią stawkę za pomocą licznych instrukcji warunkowych? Nie, do tego świetnie nadaje się wzorzec ***strategii***.

Listing 2.3:

```
<?php


interface Tax{
    public function count($net);
}


class TaxPL implements Tax{
    public function count($net) {
        return 0.23*$net;
    }
}


class TaxEN implements Tax{
    public function count($net) {
```

```
        return 0.15*$net;
    }
}

class TaxDE implements Tax{
    public function count($net) {
        return 0.3*$net;
    }
}

class Context{
    private $strategy;

    public function setCountry($country) {
        switch ($country) {
            case "PL":
                $this->strategy = new TaxPL();
                break;
            case "DE":
                $this->strategy = new TaxDE();
                break;
            case "EN":
                $this->strategy = new TaxEN();
                break;
        }
```

```
    }
    public function getTax() {
        return $this->strategy;
    }
}

// testy
$tax = new Context();
$tax->setCountry("PL");
echo $tax->getTax()->count(100); // wyswietla "23"
$tax->setCountry("EN");
echo $tax->getTax()->count(100); // wyswietla "15"
$tax->setCountry("DE");
echo $tax->getTax()->count(100); // wyswietla "30"

?>
```

Za pomocą metody ***setCountry()*** możemy w bardzo prosty sposób wybierać odpowiednią strategię obliczania podatku dostosowaną do państwa, z którego pochodzi nasz klient. Jeżeli sklep zdecyduje się na poszerzenie działalności o kolejny kraj, wystarczy tylko dopisać odpowiednią klasę implementującą ***interfejs Tax*** oraz dodać do ***switch'a*** odpowiedni warunek.

2.4 Zalety i wady

Zalety:

- Eliminacja instrukcji warunkowych – kod jest bardziej przejrzysty.
- Umożliwia wybór implementacji – algorytmy mogą rozwiązywać ten sam problem, lecz różnić się uzyskiwanymi korzyściami.
- Łatwość dołączania kolejnych strategii.
- Łatwiejsze testowanie programu – można debugować każdą strategię z osobna.

Wady:

- Dodatkowy koszt komunikacji między klientem, a strategią (wywołania metod, przekazywanie danych).
- „Rozmycie" kodu na kilka klas.

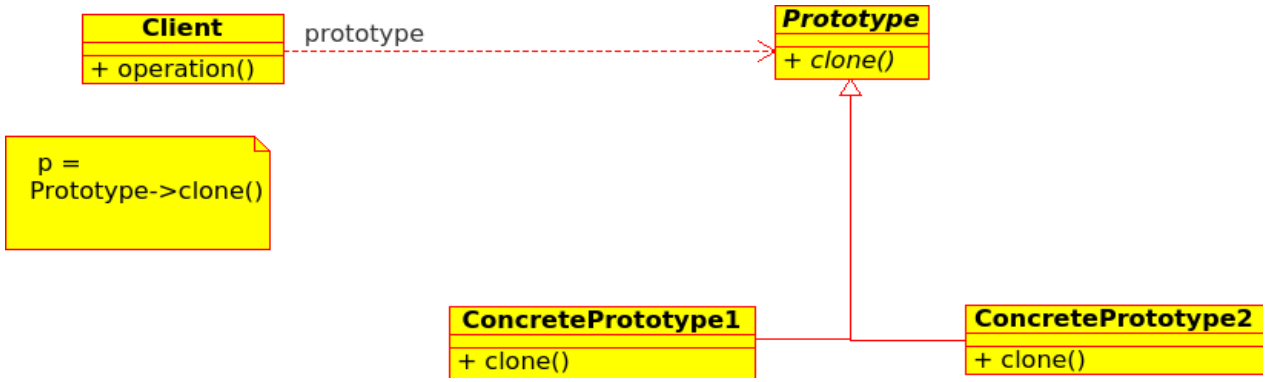
2.5 Zastosowanie

Wszędzie tam, gdzie istnieje potrzeba rozwiązania danego problemu na kilka różnych sposobów.

Oficjalna wirtualna maszyna ***Javy HotSpot*** wykorzystuje wzorzec strategii w wewnętrznej implementacji mechanizmu odśmiecania pamięci, oferując do wyboru kilka algorytmów różniących się właściwościami. Sam programista wybiera strategię odśmiecania najlepiej dopasowaną do profilu jego aplikacji.

3. Prototype

Prototyp jest wzorcem, opisującym mechanizm tworzenie nowych obiektów poprzez klonowanie jednego obiektu macierzystego. Mechanizm klonowania wykorzystywany jest wówczas, gdy należy wykreować dużą liczbę obiektów tego samego typu lub istnieje potrzeba tworzenia zbioru obiektów o bardzo podobnych właściwościach.



Listing 3.1: Diagram klas wzorca Prototype

Implementując ten wzorzec deklaruje się klasę ***Prototype*** z abstrakcyjną operacją klonującą ***clone()***. Operacja ta jest implementowana w klasach dziedziczonych po ***Prototype***. Klient chcąc stworzyć nowy obiekt wywołuje metodę ***clone()*** pośrednio, za pomocą zdefiniowanej przez siebie operacji z parametrem określającym wymaganą docelową klasę realizującą abstrakcję ***Prototype***.

3.2 Przykładowa implementacja

Listing 3.2:

```
<?php

abstract class Prototype{

    protected $name;


    public function __construct($name) {

        $this->name=$name;

    }
```

```
     abstract function __clone();
    public function getName() {
        return $this->name;
    }
}

class ConcretePrototype extends Prototype{

    public function __construct($name) {
        parent::__construct($name);
    }
    public function __clone() {}
}

// testy
$prototype = new ConcretePrototype("nazwa");
echo  $prototype->getName(); // wyswietli "nazwa"
$prototype2 = clone $prototype;
echo  $prototype2->getName(); // wyswietli "nazwa"
?>
```

3.3 Przykład z życia wzięty

Przypuśćmy, że tworzymy księgarnię internetową. Istnieje potrzeba stworzenia wielu obiektów książek o podobnych właściwościach (wspólna kategoria, autor itp.). Zamiast ustawiać pola dla każdego obiektu oddzielnie możemy wykonać klony i zmieniać tylko elementy unikalne.

Listing 3.3:

```
<?php
abstract class Book {
    protected $title;
    protected $topic;
    abstract function __clone();
    public function getTitle() {
        return $this->title;
    }
    public function setTitle($title) {
        $this->title = $title;
    }
    public function getTopic() {
        return $this->topic;
    }
}

class PHPBook extends Book {
    public function __construct() {
        $this->topic = 'PHP';
    }
    function __clone() {
    }
```

```
}

class JAVABook extends Book {
    public function __construct() {
        $this->topic = 'JAVA';
    }
    function __clone() {
    }
}

//testy
$phpbook1 = new PHPBook();
$phpbook1->setTitle("Ksiazka1");
$phpbook2 = clone $phpbook1;
$phpbook2->setTitle("Ksiazka2");

$javabook1 = new JAVABook();
$javabook1->setTitle("Ksiazka1");
$javabook2 = clone $javabook1;
$javabook2->setTitle("Ksiazka2");

echo "Kategoria: ".$phpbook1->getTopic()." Tytul: ".$phpbook1-
>getTitle()."<br />";
echo "Kategoria: ".$phpbook2->getTopic()." Tytul: ".$phpbook2-
>getTitle()."<br />";
```

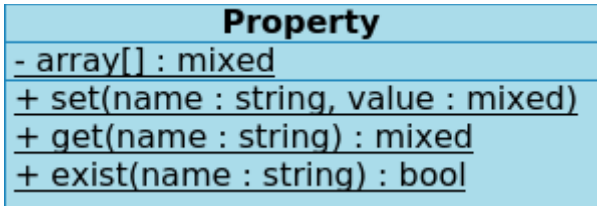
```
echo "Kategoria: ".$javabook1->getTopic()." Tytul: ".$javabook1-
>getTitle()."<br />";

echo "Kategoria: ".$javabook2->getTopic()." Tytul: ".$javabook2-
>getTitle()."<br />";

?>
```

3.4 Zastosowanie

Wzorzec ***Prototype*** można stosować w sytuacjach, gdy tworzona jest duża liczba obiektów tego samego typu. Stosuje się go głównie w celach optymalizacji, gdyż klonowanie obiektu jest szybsze niż jego stworzenie.

4. Property

Property to wzorzec projektowy, którego zadanie, jest przechowywać i udostępniać dane w obrębie aplikacji. Implementacja wzorca zastępuje globalne zmienne jakże nielubiane w dobie programowania obiektowego. Brzmi znajomo? Wzorzec ten ma zbliżone zastosowania do Singletona. Jednak tu nie tworzymy obiektu – wszystkie metody są statyczne.



Listing 4.1: Diagram klasy wzorca Property

Wszystkie dane trzymamy w tablicy ***array***. Za pomocą metod ***set()*** i ***get()*** możemy ustawiać/pobierać dane. Poza tym możemy zaimplementować inne metody kontrolujące np. dostęp do zmiennych.

4.2 Przykładowa implementacja

Listing 4.2:

```
<?php
class Property{
    private static $array = array();

    public static function set($name, $value) {
        self::$array[$name]=$value;
    }
    public static function get($name) {
        return self::$array[$name];
    }
    public static function exist($name) {
        return isset(self::$array[$name]);
    }
```

```
}

//testy
Property::set("nazwa1", 1);
Property::set("nazwa2", "wartosc");
echo Property::get("nazwa2"); // wyswietli "wartosc"
echo Property::exist("nazwa1"); // wyswietli "true"
echo Property::exist("nazwa3"); // wyswietli "false"
?>
```

4.3 Przykład z życia wzięty

Rozważmy klasę zawierającą konfigurację aplikacji.

Listing 4.3:

```
<?php
class Config{
    private static $conf = array();

    public static function set($name, $value) {
        self::$conf[$name]=$value;
    }
    public static function get($name) {
        return self::$conf[$name];
    }
    public static function exist($name) {
        return isset(self::$conf[$name]);
    }
}
```

```
//testy
Config::set("language", "pl");
Config::set("path", "/jakas_sciezka/");
echo Config::get("language"); // wyswietli "pl"
echo Config::get("path"); // wyswietli "/jakas_sciezka/"
echo Config::exist("language"); // wyswietli "true"
?>
```

Klasa ***Config*** zawiera tablicę ***conf*** z podstawowymi ustawieniami aplikacji. Dzięki zastosowaniu pola statycznego zmiana ustawień w jednym miejscu (np. zmiana języka na stronie przez użytkownika) jest „widoczna" w każdym miejscu aplikacji.

4.3 Zalety i wady

Zalety:

- Globalny dostęp do zmiennych wraz z dobrodziejstwami jakie daje programowanie obiektowe.
- Wygodne API do ustawiania/pobierania danych, które muszą mieć dostęp globalny.

Wady:

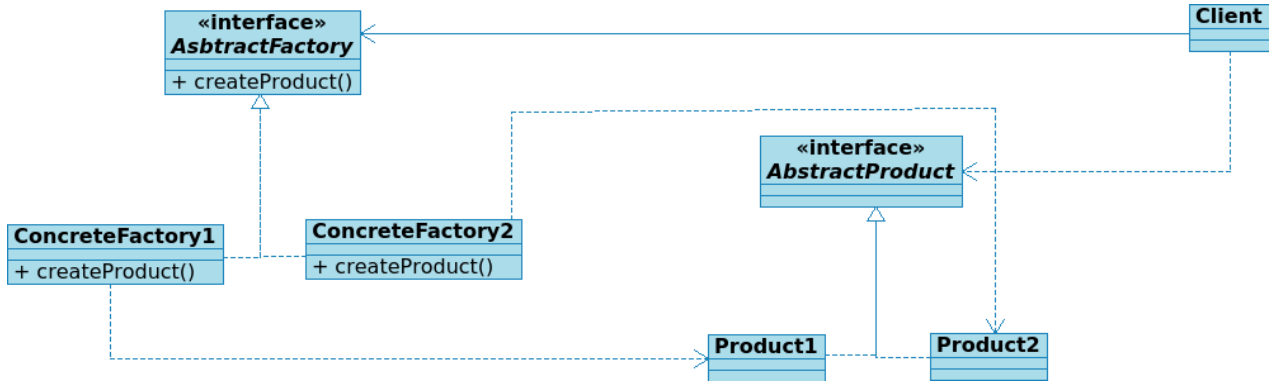
- Nie znam żadnej :)

4.5 Zastosowanie

Property można wykorzystać do przechowywania konfiguracji aplikacji.

5. Abstract factory

Fabryka abstrakcyjna jest wzorcem projektowym, którego zadaniem jest określenie interfejsu do tworzenia różnych obiektów należących do tego samego typu (rodziny). Interfejs ten definiuje grupę metod, za pomocą których tworzone są obiekty.



Listing 5.1: Diagram klasy wzorca Abstract factory

5.2 Przykładowa implementacja

Listing 5.2:

```
<?php

interface AbstractFactory {

    function createProduct();

}


class ConcreteFactory1 implements AbstractFactory {

    public function createProduct() {

        return new Product("Produkt 1");

    }

}
```

```
class ConcreteFactory2 implements AbstractFactory {
    public function createProduct() {
        return new Product("Produkt 2");
    }
}

interface AbstractProduct {
    function getName();
}

class Product implements AbstractProduct {
    private $name;

    public function __construct($name) {
        $this->name = $name;
    }
    public function getName() {
        return $this->name;
    }
}

class Factory {
    const FACTORY1="Factory 1";
    const FACTORY2="Factory 2";
```

```
    public static function chooseFactory($name) {
        switch($name) {
            case self::FACTORY1:
                return new ConcreteFactory1();
                break;
            case self::FACTORY2:
                return new ConcreteFactory2();
                break;
        }
    }
}

// test
$factory1=Factory::chooseFactory("Factory 1");
$factory2=Factory::chooseFactory("Factory 2");
$product1=$factory1->createProduct();
$product2=$factory2->createProduct();
echo $product1->getName();
echo $product2->getName();
?>
```

Najczęściej fabrykę abstrakcyjną buduje się w postaci interfejsu. Po stronie klienta tworzone są konkretne implementacje fabryki. Konkretne obiekty tworzone są poprzez wywołanie metod interfejsu. Fabryka pozwala na tworzenie zestawów obiektów dopasowanych do konkretnych zastosowań (np. różnych funkcjonalności, platform, itp.). Każda z konkretnych fabryk realizuje odmienny zestaw klas, ale zawsze posiadają one pewien zdefiniowany zespół interfejsów.

5.3 Przykład z życia wzięty

Rozważmy fragment aplikacji odpowiedzialny za wyświetlanie treści.

Listing 5.3:

```
<?php


// Produkty
interface Document {
    function generate();
}


class PDF implements Document {
    public function generate() {
        return 'Dokument PDF';
    }

    public function setColor($color) {
        echo "// Ustawiam kolor: ".$color;
    }

}


class HTML implements Document {
    public function generate() {
        return 'Dokument HTML';
```

```
    }

    public function set_color($color) {
        echo "// Ustawiam kolor: ".$color;
    }

}

// Fabryki
interface DocumentFactory {
    function create();
    function setColor($color);
}

class PDFFactory implements DocumentFactory {
    private $color;

    public function create() {
        $doc = new PDF();
        $doc->setColor($this->color);
        return new $doc;
    }

    public function setColor($color) {
        $this->color = $color;
```

```
    }

}

class HTMLFactory implements DocumentFactory {
    private $color;

    public function create() {
        $doc = new HTML();
        $doc->set_color($this->color);
        return new $doc;
    }

    public function setColor($color) {
        $this->color = $color;
    }
}

class Page {
    private $documentFactory;

    public function __construct(DocumentFactory $factory) {
        $this->documentFactory = $factory;
    }
    public function render() {
```

```
        $document = $this->documentFactory->create();
        echo $document->generate();
    }

}

// testy
$pdf = new PDFFactory();
$pdf->setColor("#000000");
$html = new HTMLFactory();
$html->setColor("#ffffff");
$page1 = new Page($pdf);
$page1->render(); // wyswietli "Dokument PDF"
$page2 = new Page($html);
$page2->render(); // wyswietli "Dokument HTML"

?>
```

Klasy ***HTML*** i ***PDF*** zawierają konkretne dane na temat różnych typów dokumentów (w tym wypadku są to produkty fabryki). Z kolei ***HTMLFactory*** i ***PDFFactory*** tworzą obiekty odpowiednich klas. Dzięki zastosowaniu wspólnego interfejsu dla fabryk możemy bardzo łatwo zmieniać typ generowanego dokumentu.

5.4 Zalety i wady

Zalety:

- Odseparowanie klas konkretnych - klienci nie wiedzą jakich typów konkretnych używają, posługują się interfejsami abstrakcyjnymi.
- Łatwa wymiana rodziny produktów.
- Spójność produktów - w sytuacji, gdy pożądane jest, aby klasy produkty były z określonej rodziny, fabryka bardzo dobrze to zapewnia.

Wady:

- Trudność w dołączaniu nowego produktu do rodzin produktów, spowodowana koniecznością rozszerzania interfejsów fabryki.

5.5 Zastosowanie

Wzorzec fabryki abstrakcyjnej można wykorzystać między innymi do stworzenia uniwersalnego sterownika do obsługi różnych typów baz danych (MySQL, Oracle SQL itp.).

6. Dependency Injection

Dependency Injection jest chyba jednym z najprostszych wzorców projektowych, więc zapewne wiele osób używało go nieświadomie. Warto jednak wiedzieć, że dane rozwiązanie ma szerokie zastosowanie i jakąś nazwę... ;)

Tym razem wyjątkowo nie będzie opisu teoretycznego, bo i nie za bardzo jest o czym mówić. Zasadę najłatwiej będzie zrozumieć na praktycznym przykładzie.

6.1 Przykład z życia wzięty

Przeanalizujmy fragment klasy do obsługi użytkownika.

Listing 6.1:

```
<?php

class Session {

    public function __construct($name = 'PHP_SESSSION') {

        session_name($name);

        session_start();

    }

    public function set($key, $value) {

        $_SESSION[$key] = $value;

    }

    public function get($key) {

        return $_SESSION[$key];

    }

}

class User {

    protected $session;

    public function __construct($session) {
```

```
        $this->session = $session;
    }
    public function setName($name) {
        $this->session->set('name', $name);
    }
    public function getName() {
        return $this->session->get('name');
    }
}
$session = new Session("MOJA_SESJA");
$user = new User($session);
$user->setName("moj_login");
echo $user->getName();
?>
```

Klasa ***Session*** udostępnia obiektowe API do obsługi sesji. Obiekt tej klasy jest przekazywany w konstruktorze obiektu klasy ***User***, gdzie jest dalej wykorzystywany. Jest to właśnie ***wstrzykiwanie zależności***, czyli nie tworzymy instancji obiektu w danej klasie, tylko przekazujemy go przez konstruktor/metodę.

Dlaczego takie rozwiązanie jest lepsze niż:

```
$this->session = new Sessioon($parametrKonstuktora);
```

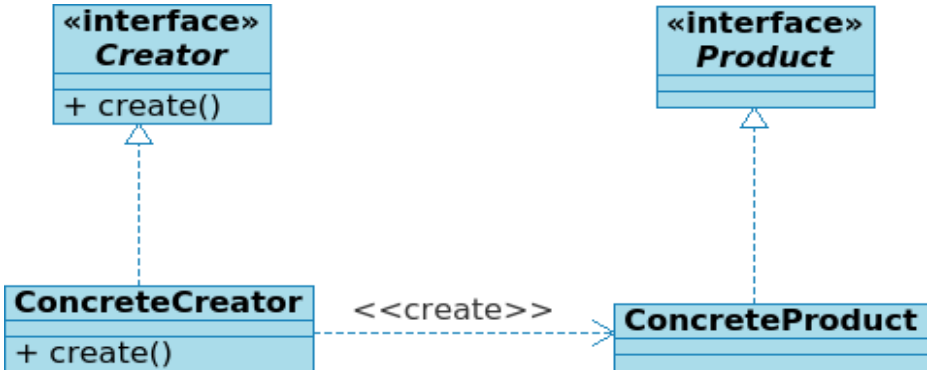
Parametr ten na dobrą sprawę nie dotyczy obiektu klasy ***User*** – jest „sztucznie” dodany. Zdecydowanie przejrzyściej i wygodniej jest zdefiniować obiekt poza klasą i przekazać już gotowy, w pełni skonfigurowany przez nasze parametry. Poza tym przy zmianach parametrów konstruktora musielibyśmy za każdym razem modyfikować wszystkie klasy inicjujące obiekt danej klasy.

6.2 Zastosowanie

Wszędzie tam, gdzie występują zależności między obiektami.

7. Factory method

Wzorzec ***metody wytwórczej*** dostarcza abstrakcji do tworzenia obiektów nieokreślonych, ale powiązanych typów. Umożliwia także dziedziczącym klasom decydowanie jakiego typu ma to być obiekt.



Listing 7.1: Diagram klasy wzorca Factory method

Wzorzec składa się z dwóch ról: produktu ***Product*** definiującego typ zasobów oraz kreatora ***Creator*** definiującego sposób ich tworzenia. Wszystkie typy produktów (***ConreteProduct1, ConreteProduct2*** itp.) muszą implementować interfejs ***Product***. Z kolei ***ConcreteCrator*** dostarcza mechanizm umożliwiający stworzenie obiektu produktu danego typu.

7.2 Przykładowa implementacja

Listing 7.2:

```
<?php


// Produkty

interface Product{

    public function getName();

}

class ConceteProduct1 implements Product{

    public function getName() {

        return "Produkt 1";
```

```
    }
}
class ConceteProduct2 implements Product{
    public function getName() {
        return "Produkt 2";
    }
}

// Kreator tworzacy obiekt produktu
interface Creator{
    public function create($type);
}
class ConcreteCreator implements Creator{
    public function create($type) {
        switch($type) {
            case 'Product 1':
                return new ConceteProduct1();
                break;
            case 'Product 2':
                return new ConceteProduct2();
                break;
        }
    }
}
```

```
// testy

$creator = new ConcreteCreator();

$prod1 = $creator->create("Product 1");

$prod2 = $creator->create("Product 2");

echo $prod1->getName(); // wyswietli "Produkt 1"

echo $prod2->getName(); // wyswietli "Produkt 2"


?>
```

7.3 Przykład z życia wzięty

Tworzymy system zamówień dla pizzerii. W ofercie są różne typy pizz. Podstawowym pytaniem jest: jak stworzyć wydajny mechanizm do tworzenia obiektów różnych rodzai pizz? Posłużmy się metodą wytwórczą...

Listing 7.3:

```
<?php


// Produkty

interface Pizza{

    public function getName();

}

class HawaiianPizza implements Pizza{

    public function getName() {

        return "Hawalian pizza";

    }

}

class DeluxePizza implements Pizza{
```

```
    public function getName() {
        return "Deluxe pizza";
    }
}

// Kreator tworzacy obiekt produktu
interface Creator{
    public function create($type);
}
class ConcreteCreator implements Creator{
    public function create($type) {
        switch($type) {
            case 'Hawalian':
                return new HawaiianPizza();
                break;
            case 'Deluxe':
                return new DeluxePizza();
                break;
        }
    }
}

// testy
$creator = new ConcreteCreator();
$prod1 = $creator->create("Hawalian");
```

```
$prod2 = $creator->create("Deluxe");

echo $prod1->getName(); // wyswietli "Hawalian pizza"

echo $prod2->getName(); // wyswietli "Deluxe pizza"


?>
```

Dzięki zastosowaniu factory method możemy w łatwy sposób dołączać kolejne pizze. Zamiast używania konstrukcji ***switch*** (korzystam z tego, gdyż nie chcę komplikować przykładu) warto byłoby stworzyć bardziej abstrakcyjny mechanizm.

7.4 Zalety i wady

Zalety:

- Niezależność od konkretnych implementacji zasobów oraz procesu ich tworzenia.
- Wzorzec hermetyzuje proces tworzenia obiektów, zamykając go za ściśle zdefiniowanym interfejsem.
- Spójność produktów - w sytuacji, gdy pożądane jest, aby klasy produkty były z określonej rodziny.

Wady:

- Nie znam...

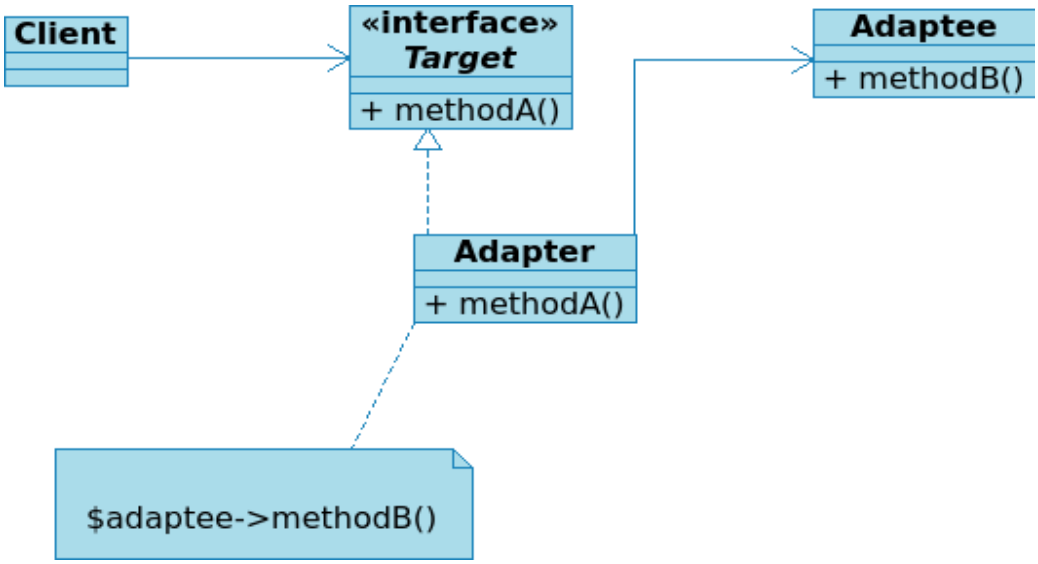
7.5 Zastosowanie

Wzorzec metody wytwórczej można wykorzystać między innymi przy tworzeniu systemów zamówień, gdzie oferta może się zmieniać, ale składa z jednakowego typu produktów.

Innym zastosowaniem może być system pluginów. Dzięki zastosowaniu metody wytwórczej możemy łatwo rozbudowywać nasz skrypt o kolejne funkcjonalności (np. o obsługę kolejnych formatów plików).

8. Adapter

Wzorzec ***adapter*** (znany także pod nazwą ***wrapper***) służy do przystosowania interfejsów obiektowych, tak aby możliwa była współpraca obiektów o niezgodnych interfejsach. Szczególnie przydaje się przypadku wykorzystania gotowych bibliotek o interfejsach niezgodnych ze stosowanymi w aplikacji. W świecie rzeczywistym adapter to przejściówka, np. przejściówka do wtyczki gniazdka angielskiego na polskie.



Listing 8.1: Diagram klas wzorca Adapter

Struktura wzorca składa się z elementów takich jak: ***Target, Adaptee, Adapter oraz Client***. ***Target*** jest abstrakcją (zazwyczaj interfejsem), jakiej oczekuje klient. Elementem dostarczającym żądanej przez klienta funkcjonalności jest ***Adaptee*** (np. zewnętrzną biblioteką). Rolą ***adaptera***, który implementuje interfejs ***Target***, jest „przetłumaczenie" wywołania metod należących do interfejsu ***Target*** poprzez wykonanie innych, specyficznych metod z klasy ***Adaptee***.

8.2 Przykładowa implementacja

Listing 8.2:

```
<?php

interface Target {

    public function  methodA();

}

class Adaptee {

    public function methodB() {

        echo "Metoda B";

    }

}

class Adapter implements Target {

    public function methodA() {

        $adaptee = new Adaptee();

        $adaptee->methodB();

    }

}


//test

$client = new Adapter();

$client->methodA(); // wyswietli "metoda B"

?>
```

8.3 Przykład z życia wzięty

Przejmujemy administrację nad jakimś większym projektem i istnieje potrzeba wymiany starej biblioteki XML na nową. Oczywiście nazwy metod różnią się diametralnie. Zamiast poprawiać nazwy metod w całym projekcie możemy napisać adapter.

Listing 8.3:

```
<?php

interface OldXML {

    public function  writeXml();

}

class NewXML {

    public function xml() {

        echo "Kod XML";

    }

}

class XML implements OldXML {

    public function writeXml() {

        $adaptee = new NewXML();

        $adaptee->xml();

    }

}


//test

$client = new XML();

$client->writeXml(); // wyswietli "Kod XML"

?>
```

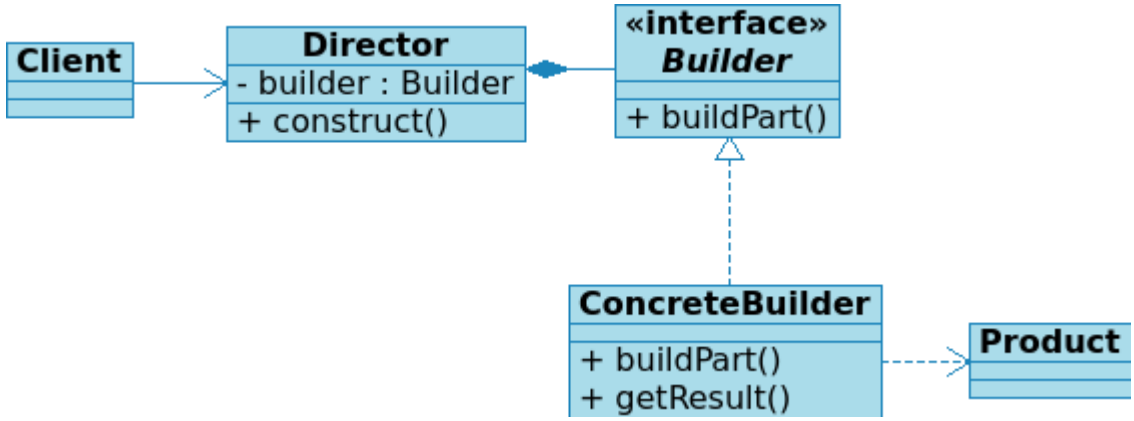
Interfejs ***OldXML*** zawiera zestaw metod starej biblioteki. ***NewXML*** jest klasą nowej biblioteki. Natomiasr klasa ***XML*** jest swoistą przejściówką pomiędzy dwoma wersjami klas.

8.4 Zastosowanie

Wzorzec adaptera stosowany jest najczęściej w przypadku, gdy wykorzystanie istniejącej klasy jest niemożliwe ze względu na jej niekompatybilny interfejs. Drugim powodem użycia może być chęć stworzenia klasy, która będzie współpracowała z klasami o nieokreślonych interfejsach.

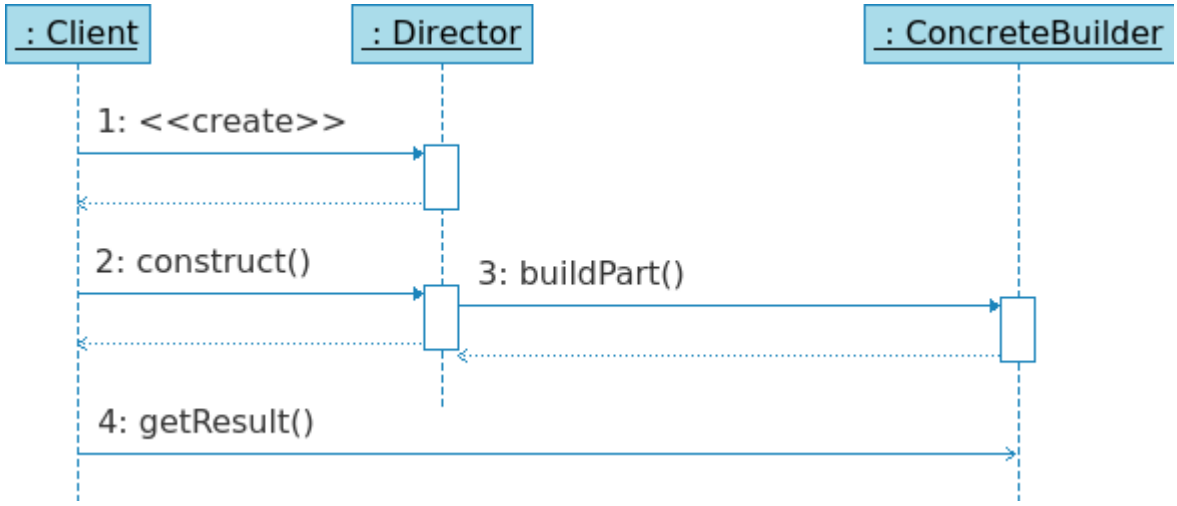
9. Builder

Budowniczy jest wzorcem, gdzie proces tworzenia obiektu podzielony jest na kilka mniejszych etapów, a każdy z nich może być implementowany na wiele sposobów. Dzięki takiemu rozwiązaniu możliwe jest tworzenie różnych reprezentacji obiektów w tym samym procesie konstrukcyjnym.



Listing 9.1: Diagram klas wzorca Builder

Standardowo wzorzec składa się z dwóch podstawowych elementów. Pierwszy z nich oznaczony jest jako ***Builder*** – jego celem jest dostarczenie interfejsu do tworzenia obiektów nazywanych produktami (***product***). Drugim elementem jest obiekt oznaczony jako ***ConcreteBuilder***, a jego celem jest tworzenie konkretnych reprezentacji produktów przy pomocy zaimplementowanego interfejsu ***Builder***. W ***ConcreteBuilder*** zawarte są procedury odpowiedzialne za konstrukcje i inicjalizację obiektu. Strukturę wzorca uzupełnia obiekt ***Director*** (zwany także czasem kierownikiem, nadzorcą – tak jak na budowie ;)), który zleca konstrukcję produktów poprzez obiekt ***Builder*** dbając o to, aby proces budowy przebiegał w odpowiedniej kolejności.



Listing 9.2: Diagram sekwencji wzorca Builder

9.2 Przykładowa implementacja

Listing 9.3:

```
<?php


class Product {


    private $part1;

    private $part2;


    public function setPart1($part1) {

        $this->part1 = $part1;

    }

    public function getPart1() {

        return $this->part1;

    }

    public function setPart2($part2) {
```

```
        $this->part2 = $part2;
    }
    public function getPart2() {
        return $this->part2;
    }

}

interface Builder {
    public function buildPart1();
    public function buildPart2();
}

class ConcreteBuilder implements Builder {
    private $product;

    public function __construct() {
        $this->product = new Product();
    }
    public function buildPart1() {
        $this->product->setPart1("1. faza budowy");
    }
    public function buildPart2() {
        $this->product->setPart2("2. faza budowy");
    }
```

```
    public function getProduct() {
        return $this->product;
    }

}

class Director{
    private $builder;

    public function __construct() {
        $this->builder = new ConcreteBuilder();
    }
    public function construct() {
        $this->builder->buildPart1();
        $this->builder->buildPart2();
    }
    public function getResult() {
        return $this->builder->getProduct();
    }
}

// testy
$director = new Director();
$director->construct();
$product = $director->getResult();
```

```
echo $product->getPart1(); // wyswetli "1. faza budowy"

echo $product->getPart2(); // wyswieli "2. faza budowy"


?>
```

9.3 Przykład z życia wzięty

Tworząc serwis udostępniający filmy wideo chcemy mieć możliwość wyboru technologii do generowania odtwarzacza wideo. Możemy użyć do tego budowniczych.

Listing 9.4:

```
<?php
class Player {
    private $player;


    public function setplayer($player) {
        $this->player = $player;
    }
    public function render() {
        return $this->player;
    }
}


interface Builder {
    public function buildPlayer();
    public function getPlayer();
}
```

```
class FlashBuilder implements Builder {
    private $player;

    public function __construct() {
        $this->player = new Player();
    }
    public function buildPlayer() {
        $this->player->setPlayer("Player w Flash");
    }
    public function getPlayer() {
        return $this->player;
    }
}

class HTMLBuilder implements Builder {
    private $player;

    public function __construct() {
        $this->player = new Player();
    }
    public function buildPlayer() {
        $this->player->setPlayer("Player w HTML5");
    }
    public function getPlayer() {
```

```
        return $this->player;
    }
}

class Director{
    private $builder;

    public function __construct(Builder $builder) {
        $this->builder = $builder;
    }
    public function construct() {
        $this->builder->buildPlayer();
    }
    public function getResult() {
        return $this->builder->getPlayer();
    }
}

// testy
$html = new HTMLBuilder();
$flash = new FlashBuilder();
$director = new Director($flash);
$director->construct();
$player = $director->getResult();
echo $player->render(); // wyswietli "player w flash"
```

```
$director2 = new Director($html);

$director2->construct();

$player2 = $director2->getResult();

echo $player2->render(); // wyswietli "player w HTML5"


?>
```

Dwaj budowniczy ***FlashBuilder*** i ***HTMLBuilder*** mają za zadanie stworzyć obiekt ***Video*** z uwzględnieniem wykorzystanej technologii. Z kolei ***Director*** „pilnuje" poprawnej kolejności wywoływania metod. Dzięki wykorzystaniu Dependency Injection możemy łatwo dodawać kolejnych budowniczych.

9.4 Zalety i wady

Zalety:

- Duża możliwość zróżnicowania wewnętrznych struktur klas.
- Duża skalowalność (dodawanie nowych reprezentacji obiektów jest uproszczone).
- Większa możliwość kontrolowania tego, w jaki sposób tworzony jest obiekt (proces konstrukcyjny jest niezależny od elementów, z których składa się tworzony obiekt.

Wady:

- Duża liczba obiektów reprezentujących konkretne produkty.
- Nieumiejętne używanie wzorca może spowodować nieczytelność kodu (jeden produkt może być tworzony przez zbyt wielu budowniczych).

9.5 Zastosowanie

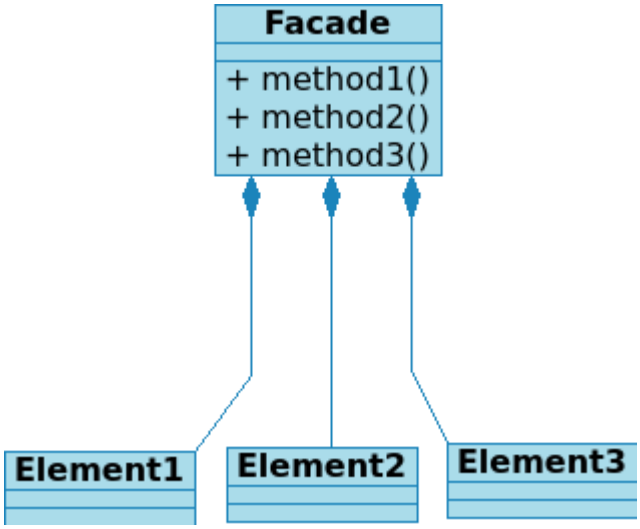
Wzorzec budowniczego stosowany jest do oddzielenia sposobu tworzenia obiektów od tego jak te obiekty mają wyglądać. Przykładem jest oprogramowanie konwertujące tekst z jednego formatu na drugi. Algorytm odczytujący i interpretujący dane wejściowe jest oddzielony od algorytmu

tworzącego dane wyjściowe. Dzięki takiemu rozwiązaniu możliwe jest zastosowanie jednego obiektu odczytującego dane wejściowe oraz wielu obiektów konwertujących odczytane dane do różnych formatów (ASCII, HTML, RTF, itp.), co zwiększa skalowalność rozwiązania.

Innym zastosowaniem może być stworzenie narzędzia, które będzie miało różną implementację zależnie od użytej technologii – tak jak w omawianym przykładzie odtwarzacz wideo.

10. Facade

Fasada służy do ujednolicenia dostępu do złożonego systemu poprzez udostępnienie uproszczonego i uporządkowanego interfejsu programistycznego. Fasada zwykle implementowana jest w bardzo prosty sposób - w postaci jednej klasy powiązanej z klasami reprezentującymi system, do którego klient chce uzyskać dostęp.



Listing 10.1: Diagram klas wzorca Facade

10.2 Przykładowa implementacja

Listing 10.2:

```
<?php


class Element1{

    public function method() {
```

```
        echo "metoda klasy Element1";
    }
}

class Element2{
    public function method() {
        echo "metoda klasy Element2";
    }
}

class Element3{
    public function method() {
        echo "metoda klasy Element3";
    }
}

class Facade{
    private $objects = array();

    public function __construct() {
        $this->objects[0] = new Element1();
        $this->objects[1] = new Element2();
        $this->objects[2] = new Element3();
    }
```

```
    public function method1() {
        $this->objects[0]->method();
    }

    public function method2() {
        $this->objects[1]->method();
    }

    public function method3() {
        $this->objects[2]->method();
    }

    public function apiElement1() {
        return $this->objects[0];
    }
}

// testy
$api = new Facade();
$api->method1(); // wyswietli "metoda klasy Element1"
$api->method2(); // wyswietli "metoda klasy Element2"
$api->method3(); // wyswietli "metoda klasy Element3"
var_dump($api->apiElement1());

?>
```

Klasa ***Facade*** zawiera obiekty klas, które stanowią elementy systemu. Wywołując metodę (***method1(), method2(), method3()***) tej klasy, tak naprawdę wywołujemy metody poszczególnych obiektów składowych systemu. Metoda ***apiElement1()*** jest pewnym „rozszerzeniem" fasady, która zwraca cały obiekt klasy ***Elemnt1*** – rozwiązanie takie zezwala w łatwy sposób udostępniać całą bibliotekę wymaganą w specyficznych sytuacjach.

10.3 Przykład z życia wzięty

Mamy za zadanie stworzyć sklep internetowy, który ma umożliwić łatwą integrację z innymi serwisami. Do realizacji tego zadania musimy stworzyć API udostępniane partnerom. Najłatwiej użyć do tego wzorca fasady.

Listing 10.3:

```
<?php


class User{
    public function login() {
        echo "Logowanie do systemu\n";
    }
    public function register() {
        echo "Rejestracja\n";
    }
}


class Cart{
    public function getItems() {
        echo "Zawartość koszyka\n";
    }
}
```

```
class Product{
    public function getAll() {
        echo "Lista produktów\n";
    }

    public function get($id) {
        echo "Produkt o ID ".$id."\n";
    }
}

class API{
    private $user;
    private $cart;
    private $product;

    public function __construct() {
        $this->user = new User();
        $this->cart = new Cart();
        $this->product = new Product();
    }

    public function login() {
        $this->user->login();
    }
```

```
    public function register() {
        $this->user->register();
    }

    public function getBuyProducts() {
        $this->cart->getItems();
    }

    public function getProducts() {
        $this->product->getAll();
    }

    public function getProduct($id) {
        $this->product->get($id);
    }
}

// testy
$client = new API();
$client->register();
$client->login();
$client->getProducts();
$client->getProduct(5);
$client->getBuyProducts();
```

```
?>
```

Prawda, że dużo łatwiej korzystać z ***API*** niż odwoływać się bezpośrednio do składowych systemu? ;) Nie wspominając o kwestiach bezpieczeństwa...

10.4 Zalety i wady

Zalety:

- Zmniejszenie liczby zależności między klientem, a złożonym systemem — jeśli klient nie korzysta bezpośrednio z żadnych elementów ukrytego za fasadą systemu, całość jest łatwiejsza w konserwacji i utrzymaniu.
- Wprowadzenie podziału aplikacji na warstwy, który ułatwia niezależny rozwój elementów klienta i złożonego systemu.
- Możliwość zablokowania klientowi drogi do bezpośredniego korzystania ze złożonego systemu, jeśli jest to konieczne.

Wady:

- Są jakieś? ;)

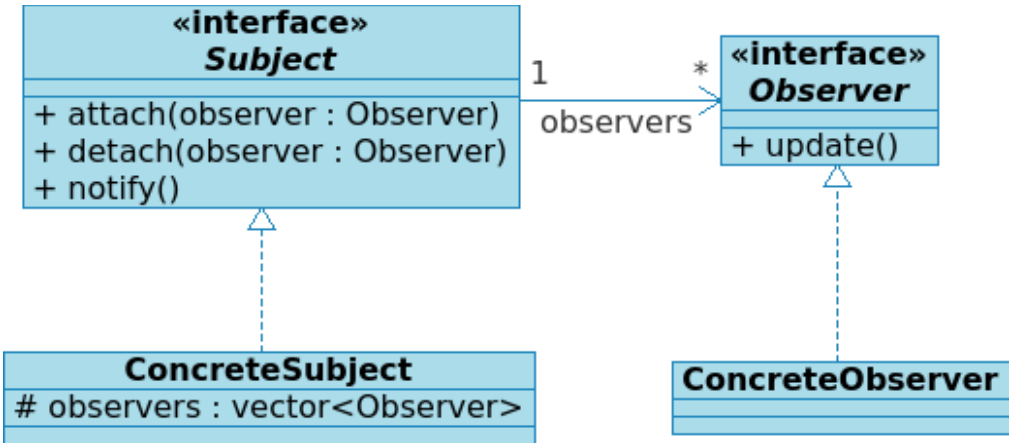
10.5 Zastosowanie

Przykładem użycia wzorca fasady może być zbudowanie API, które zezwoli zewnętrznym serwisom i aplikacjom łączyć się w prosty sposób z naszą aplikacją.

Innym praktycznym przykładem jest system bankowy. Logując się na internetowe konto mamy dostęp tylko do pojedynczych składowych całego systemu (autoryzacja, przelewy, saldo konta itp).

11. Observer

Głównym obszarem wykorzystania wzorca ***Obserwator*** jest stworzenie relacji typu jeden-do-wielu łączącej grupę obiektów. Dzięki zastosowaniu wzorca zmiana stanu (czyli zmiana aktualnych wartości pól) obiektu obserwowanego umożliwi automatyczne powiadomienie o niej wszystkich innych dołączanych elementów (obserwatorów).



Listing 11.1: Diagram klas wzorca Observer

Interfejs ***Subject*** definiuje operacje ***attach()*** i ***detach()*** pozwalające odpowiednio na dołączanie i odłączanie obserwatorów. Ponadto zdefiniowana jest też operacja ***notify()***, służąca do powiadamiania wszystkich zarejestrowanych obserwatorów o zmianie stanu obiekt obserwowanego poprzez wywołanie w pętli metody ***update()***, która jest zadeklarowana w interfejsie ***Observer***. Operacja ta jest implementowana w klasie realizującej ten interfejs i służy do powiadamiania konkretnego obserwatora o zmianie stanu obiektu obserwowanego.

11.2 Przykładowa implementacja

Listing 11.2:

```
<?php


class Observer1 implements SplObserver {


    public function update(SplSubject $subject) {

        echo 'Aktualizacja Observer1';
```

```
    }

}

class Observer2 implements SplObserver {

    public function update(SplSubject $subject) {
        echo 'Aktualizacja Observer2';
    }

}

class Subject implements SplSubject {

    protected $observers = array();

    public function attach(SplObserver $observer) {
        $this->observers[spl_object_hash($observer)] = $observer;
    }

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

    public function notify() {
```

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

}

$subject = new Subject();
$observer1 = new Observer1();
$observer2 = new Observer2();

$subject->attach($observer1);
$subject->attach($observer2);
$subject->notify();

?>
```

11.3 Przykład z życia wzięty

No i tu zaczynają się małe schody … Ze względu na specyfikę PHP (aplikacja „działa” raptem ułamki sekund) użyteczność tego wzorca jest nieco ogranicza w porównaniu z innymi językami (Java, C++ itp.). Jednak, mimo tego da się wykorzystać dobrodziejstwa ***Obserwatora*** w PHP.

Listing 11.3:

```
<?php

class CacheObserver implements SplObserver {

    public function update(SplSubject $subject) {
        echo "Odswieza cache";
    }

}

class RSSObserver implements SplObserver {

    public function update(SplSubject $subject) {

        echo "Odswieza RSS";
    }

}

class NewsletterObserver implements SplObserver {

    public function update(SplSubject $subject) {

        echo "Wysylam maile z nowym newsem";
```

```
    }

}

class News implements SplSubject {

    private $observers = array();

    public function attach(SplObserver $observer) {
        $this->observers[spl_object_hash($observer)] = $observer;
    }

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

    public function notify() {
        foreach ($this->observers as $observer) {
            $observer->update($this);
        }
    }

    public function add($data) {
        echo 'Dodaje news do bazy';
        $this->notify();
```

```
    }

}

$news = new News();

$news->attach(new RSSObserver());
$news->attach(new CacheObserver());
$news->attach(new NewsletterObserver());

$news->add(array(
    'title' => 'Tytuł',
    'content' => 'blablabla'
));

?>
```

Do obiektu klasy ***News***, który jest obserwowany dodajemy trzech obserwatorów. Dzięki temu zabiegowi przy dodawaniu nowego artykułu elementy systemu zostaną „poinformowane" i odświeżą dane w cache/RSS oraz wyślą odpowiednie maile do czytelników.

11.4 Zalety i wady

Zalety:

- Luźna zależność między obiektem obserwującym i obserwowanym. Ponieważ nie wiedzą one wiele o sobie nawzajem, mogą być niezależnie rozszerzane i rozbudowywane bez wpływu na drugą stronę.
- Relacja między obiektem obserwowanym a obserwatorem tworzona jest podczas

wykonywania programu i może być dynamicznie zmieniana.

- Możliwość zablokowania klientowi drogi do bezpośredniego korzystania ze złożonego systemu, jeśli jest to konieczne.

Wady:

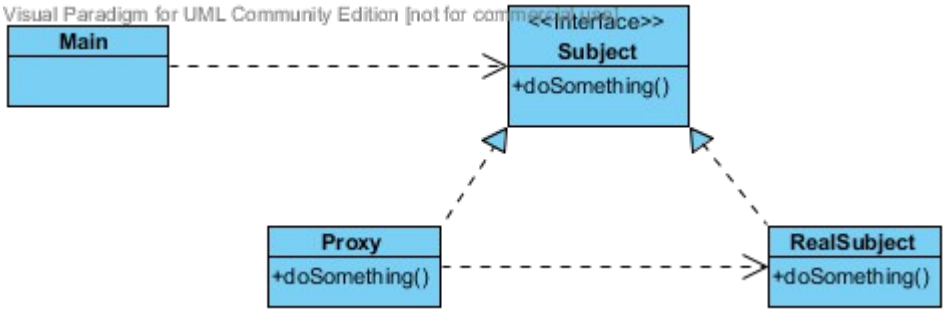
- Obserwatorzy nie znają innych obserwatorów, co w pewnych sytuacjach może wywołać trudne do znalezienia skutki uboczne.

11.5 Zastosowanie

Wzorzec Obserwatora sprawdza się wszędzie tam, gdzie stan jednego obiektu uzależniony jest od stanu drugiego obiektu.

12. Proxy

Pełnomocnik jest wzorcem projektowym, którego celem jest utworzenie obiektu zastępującego inny obiekt. Stosowany jest on w celu kontrolowanego tworzenia na żądanie kosztownych obiektów oraz kontroli dostępu do nich.



Listing 12.1: Diagram klas wzorca Proxy

Interfejs ***Subcject*** definiuje pewną metodę. Klasy ***Proxy*** i ***RealSubject*** realizują powyższy interfejs na dwa różne sposoby. W klasie ***Proxy*** zostaje wywołana metoda ***doSmething()*** obiektu klasy ***RealSubject***. Przed wywołaniem tej metody można oczywiście wykonać inne operacje. Obrazuje to poniższy listing.

12.2 Przykładowa implementacja

Listing 12.2:

```
<?php


interface Subject {


    function doSomething();
}


class RealSubject implements Subject {
```

```
    public function doSomething() {
        echo 'I do something...';
    }

}

class Proxy implements Subject {

    private $realSubject;

    public function doSomething() {
        if ($this->realSubject == null) {
            echo 'Proxy';
            $this->realSubject = new RealSubject();
        }
        $this->realSubject->doSomething();
    }

}

$obj = new Proxy();
$obj->doSomething(); // wyswietli najpierw "Proxy", a nastepnie "I do something..."
$obj->doSomething(); // wyswietl "I do something..."
?>
```

12.3 Przykład z życia wzięty

Zapewne zastanawiasz się jaki jest sens stosowania klasy pośredniczącej. Przeanalizujmy przykład, w którym wykorzystamy wzorzec ***Proxy*** dla biblioteki generującej obrazki w wysokiej rozdzielczości. Dodatkowo chcemy zabezpieczyć dostęp hasłem.

Listing 12.3:

```
<?php
interface HighResolutionImage{
    function display();
}
class Image implements HighResolutionImage{
    private $options;
    public function __construct($options) {
        $this->options=$options;
        // generate image
    }
    public function display() {
        echo 'display image';
    }
}
class ProxyImage implements HighResolutionImage{
    private $options;
    private $passwrd;
    private $image = null;
    public function __construct($options, $password) {
        $this->options=$options;
```

```
        $this->passwrd=$password;
    }
    public function display() {
        if($this->passwrd=='tajne') {
        if($this->image==null) {
            $this->image=new Image($this->options);
        }
        $this->image->display();
        } else {
            echo 'Access denied';
        }
    }
}

$image = new ProxyImage(array('width' => 3800, 'height' => 2000),
'tajne');
$image2 = new ProxyImage(array('width' => 3800, 'height' => 2000),
'tajne2');
$image->display(); // wyswietli "display image"
$image2->display(); // wyswietli "Access denied"
?>
```

Dzięki wykorzystaniu ***ProxyImage*** obrazek zostanie wygenerowany tylko w przypadku podania prawidłowego hasła oraz w sytuacji, gdy naprawdę istnieje potrzeba jego wyświetlenia.

12.4 Zastosowanie

Wzorzec ***Proxy*** możemy wykorzystać w sytuacji, gdy koszt utworzenia obiektu jest wysoki, a wykorzystanie danego obiektu uzależnione jest od spełnienia pewnych warunków.

13. Podsumowanie

Na zakończenie dołączam jeszcze kilka praktycznych przykładów wykorzystania wzorców w budowie sklepu internetowego.

13.2 Przykład 1 – strategia

Tworzymy sklep internetowy i w specyfikacji mamy określane, że wyświetlana cena ma zależeć od typu klienta (nowy klient, stały klient, hurtownik itp.). Ponadto nie mamy gwarancji, że sposób wyliczania ceny nie zmieni się w przyszłości. Jak odpowiednio zaprogramować ten mechanizm? Najlepiej użyć wzorca strategii. Spójrzmy na przykład:

Listing 13.1:

```
<?php


interface Price {


    public function count($value);
}


class PriceForNewClient implements Price {


    public function count($value) {
        return 1.20 * $value;
    }


}


class PriceForRegularClient implements Price {
```

```
    public function count($value) {
        return 1.15 * $value;
    }

}

class PriceForWholesaler implements Price {

    public function count($value) {
        return 1.10 * $value;
    }

}

class Product {

    private $name;
    private $basicPrice;
    private $strategyPrice;

    public function __construct($name, $basicPrice, $strategy) {
        $this->name=$name;
        $this->basicPrice=$basicPrice;
        $this->strategyPrice = new $strategy();
```

```
    }
    public function getPrice() {
        return $this->strategyPrice->count($this->basicPrice);
    }
    public function getName() {
        return $this->name;
    }
    public function setStrategy($strategy) {
        $this->strategyPrice = new $strategy();
    }

}

// testy
$product = new Product("produkt 1", 100, "PriceForNewClient");
echo "Nazwa produktu: ".$product->getName().", cena produktu: ".
$product->getPrice()."<br />";
$product->setStrategy("PriceForRegularClient");
echo "Nazwa produktu: ".$product->getName().", cena produktu: ".
$product->getPrice()."<br />";
$product->setStrategy("PriceForWholesaler");
echo "Nazwa produktu: ".$product->getName().", cena produktu: ".
$product->getPrice()."<br />";

?>
```

W powyższym przykładzie mamy trzy strategie naliczania ceny zależne od typu klienta. Żeby zmienić sposób wyliczania ceny wystarczy tylko utworzyć nowy obiekt strategii za pomocą ***setStrategy()***. Dzięki zastosowaniu wzorca późniejsze modyfikacje będą znacznie prostsze.

13.3 Przykład 2 – metoda wytwórcza

Kolejnym problemem w naszym sklepie jest sposób generowania różnych widoków. Dla różnego rodzaju porównywarek cenowych listę produktów musimy wygenerować zwykle jako dokument XML, natomiast dla użytkownika potrzebujemy zwykłą stronę HTML. Problem możemy rozwiązać za pomocą fabryki abstrakcyjnej lub metody wytwórczej. Wybierzemy metodę wytwórczą, ponieważ nie potrzebujemy dwóch różnych fabryk. Naszym wymogiem są tylko dwa różne produkty (XML i HTML). Przykład:

Listing 13.2:

```
<?php


interface IView {


    function render();
}


class HtmlView implements IView {


    function render() {
        // generowanie widoku html
    }


}
```

```
class XmlView implements IView {

    function render() {
        // generowanie widoku xml
    }

}

class PdfView implements IView {

    function render() {
        // generowanie widoku pdf
    }

}

class View {

    static function factory($fileName) {
        switch (end(explode('.', $fileName))) {
            case 'html' :
                return new HtmlView();
            case 'xml' :
                return new XmlView();
            case 'pdf' :
```

```
                return new PdfView();
            default :
                throw new Exception('nieznany typ pliku');
        }
    }

}

$html = View::factory("strona.html");
var_dump($html);
$xml = View::factory("strona.xml");
var_dump($xml);
$pdf = View::factory("strona.pdf");
var_dump($pdf);

?>
```

Na podstawie rozszerzenia pliku metoda ***factory()*** stworzy obiekt odpowiedniego typu. Tworząc interfejs zapewniamy sobie jednakowe API dla różnych typów dokumentów. Rozwiązanie takie znacznie ułatwia dalszą pracę programisty – nie musi wdawać się w szczegóły techniki generowania danego typu dokumentu, ma do tego zapewnione abstrakcyjne API.

13.4 Przykład 3 – fasada

Nasz sklep internetowy ma być naprawdę rozbudowany ;). Kolejnym problemem jaki możemy napotkać jest stworzenie API, które ułatwi integrację innym serwisom z naszym sklepem. Takie API na pewno musi zawierać metody do logowania, pobierania produktów i kupowania. Dobrze napisana aplikacja funkcje te ma zawarte w różnych klasach (modelach). A więc powstaje pytanie: jak je scalić? Z pomocą przybywa wzorzec fasady...

Listing 13.3:

```
<?php


class User{

    public function login() {

        echo "Logowanie do systemu\n";

    }

    public function register() {

        echo "Rejestracja\n";

    }

}


class Cart{

    public function getItems() {

        echo "Zawartość koszyka\n";

    }

}


class Product{

    public function getAll() {

        echo "Lista produktów\n";

    }


    public function get($id) {

        echo "Produkt o ID ".$id."\n";
```

```
    }
}

class API{
    private $user;
    private $cart;
    private $product;

    public function __construct() {
        $this->user = new User();
        $this->cart = new Cart();
        $this->product = new Product();
    }

    public function login() {
        $this->user->login();
    }

    public function register() {
        $this->user->register();
    }

    public function getBuyProducts() {
        $this->cart->getItems();
    }
```

```
    public function getProducts() {
        $this->product->getAll();
    }

    public function getProduct($id) {
        $this->product->get($id);
    }
}

// testy
$client = new API();
$client->register();
$client->login();
$client->getProducts();
$client->getProduct(5);
$client->getBuyProducts();

?>
```

Chyba nie muszę tego fragmentu tłumaczyć :)

13.5 Przykład 4 – obserwator

W projektowanym przez nas sklepie mamy zaimplementowany mechanizm cache'owania. Dodatkowo chcemy informować klientów o dodaniu nowego produktu (np. z wybranej przez nich kategorii). Oczywiście wszystko ma się dziać automatycznie. Jak sobie można z tym poradzić? Używając wzorca obserwatora...

Listing 13.4:

```
<?php

class CacheObserver implements SplObserver {

    public function update(SplSubject $subject) {
        echo "Odswieza cache\n";
    }

}

class RSSObserver implements SplObserver {

    public function update(SplSubject $subject) {

        echo "Odswieza RSS\n";
    }

}

class NewsletterObserver implements SplObserver {

    public function update(SplSubject $subject) {

        echo "Wysylam maile z nowym produktem\n";
```

```
    }

}

class Product implements SplSubject {

    private $observers = array();

    public function attach(SplObserver $observer) {
        $this->observers[spl_object_hash($observer)] = $observer;
    }

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

    public function notify() {
        foreach ($this->observers as $observer) {
            $observer->update($this);
        }
    }

    public function add($data) {
        echo 'Dodaje prodykt do bazy';
        $this->notify();
```

```
    }

}

$product = new Product();

$product->attach(new RSSObserver());
$product->attach(new CacheObserver());
$product->attach(new NewsletterObserver());

$product->add(array(
    'name' => 'Nazwa',
    'description' => 'blablabla'
));

?>
```

W powyższym przykładzie obiekt modelu produktu obserwowany jest przez trzech obserwatorów. W przypadku wywołania metody ***add()***, która zapisuje dane do bazy, zostaną o tym zdarzeniu poinformowani wszyscy obserwatorzy – z kolei oni wykonają odpowiednie czynności takie jak aktualizacja cache, wysłanie maili itp.