

Projektowanie Obiektowe

Wykład

Piotr Marek Oramus

Zakład Technologii Informatycznych
Wydział Fizyki, Astronomii i Informatyki Stosowanej
Uniwersytet Jagielloński w Krakowie

6 maja 2023

Historia kursu

- Jeszcze parę lat temu takiego kursu nie było, były kursy typu "Programowanie sieciowe klient-serwer" w C czy "Sieci komputerowe LAN".
- Potem pojawił się kurs języka Java. Był prowadzony tylko przez jeden rok (kolizja nazw)
- Następnie, pojawił się kurs PO prowadzony w semestrze zimowym (przed zajęciami z PRiR, które były w semestrze letnim) i zawierał kurs programowania w języku Java i projektowania obiektowego. Wykład realizowany był przez 22+2 godziny.
- Od roku akademickiego 2019/20 kurs realizowany jest w semestrze letnim i zawiera tematykę maksymalnie zbliżoną¹ do realizowanej na kursie dla studiów dziennych.

¹Mamy 18 godzin na wykład

Cele kursu

- Nauka programowania i projektowania obiektowego
- Zapoznanie się ze wzorcami projektowymi i zasadami programowania obiektowego
- Omówienie spraw związanych z implementacją (np. refaktoryzacja)
- Wskazanie praktycznych aspektów przedstawianych rozwiązań

Przy okazji: ponieważ przykłady będą prezentowane w języku Java, więc kurs pozwoli na lepsze poznanie tego języka.

Podstawowe zasady zaliczenia kursu

- Zaliczenie kursu odbywa się wyłącznie na podstawie wyniku egzaminu. Ocen nie przepisywałem i nie przepisuję.
- Egzamin zostanie zrealizowany stacjonarnie.
- Forma egzaminu - test wielokrotnego wyboru. W pytaniach mogą pojawić się fragmenty kodu.
- **Egzamin mogą zdawać tylko osoby, które wcześniej zaliczyły ćwiczenia.**
- Ocena (pozytywna) z ćwiczeń nie ma wpływu na ocenę z egzaminu.
- Będą dwa terminy egzaminu.
- Brak udziału w egzaminie (również z powodu niezaliczenia ćwiczeń) - NZAL.
- Slajdy są udostępnione.
- Te same zasady rozliczania obowiązują wszystkich uczestników kursu.

Terminy egzaminów

- Egzamin: zgodnie z harmonogramem zajęć
- Egzamin poprawkowy: niedziela 3 września 2023. Godzina do ustalenia.
Egzamin będzie wpisany w harmonogram zajęć.
UWAGA: egzamin w dniu 3 września oznacza, że rozliczenie ćwiczeń w sesji poprawkowej będzie zrealizowane (jeśli będzie to możliwe to zdalnie) w dniu 2 września.

Dlaczego Programowanie Obiektowe?

- Obiekt to struktura zawierająca dane (pozwalają one przechowywać stan) i metody (coś co pozwala na wykonywanie operacji na danych).
- Program obiektowy to zbiór obiektów, które mogą (i powinny) się ze sobą komunikować. **Zdecydowanie nacisk kładziony jest na komunikację pomiędzy obiektyami.** Można napisać program, który będzie używać klas tylko po to, aby "upychać" w nich niezwiązane ze sobą metody - nie będzie to jednak obiektowe podejście do programowania.
- Obiektowy sposób kodowania ma ułatwić ponowne użycie kodu oraz jego pisanie i konserwację.
- Składanie programu z obiektów odpowiadających istniejącym w rzeczywistości bytom jest dla nas bardziej naturalne niż operowanie na liczbach.

Dlaczego Projektowanie?

- Projektowanie służy do ustalenia zbioru obiektu potrzebnych do wykonania określonego zadania, powiązań i sposobu komunikacji.
- Oprogramowanie powinno być tak zaprojektowane aby **redukować koszty**
- Wbrew pozorom programy są częściej czytane niż pisane. Więcej czasu zajmie modyfikowanie kodu niż początkowe jego napisanie. I nie mam tu na myśli poprawiania błędów!
- Koszt to nie tylko wytwarzanie, ale i utrzymanie oprogramowania. Utrzymanie może kosztować więcej niż wytwarzanie!
- Koszt utrzymania to suma kosztów zrozumienia, modyfikacji, testowania, wdrożenia...
- Kodu nie daje się napisać "raz, a dobrze". Zawsze będą zmiany. "Nie ma czegoś takiego jak «napisany» program".
- I to pomimo, że takie rozwiązanie sugerują zasady ekonomii: czasowa wartość pieniądza ($\text{inflacja} > 0$) oraz niepewność jutra.

Dlaczego skoro jest tak dobrze, to jest tak źle?

- Dlaczego kolejny raz rozwiązujeemy ten sam problem?
- Czy nie można po prostu kontynuować wcześniejszych prac?
- Czy winę należy przypisać:
 - Brakom w dokumentacji?
 - Wyborowi języka programowania (np. C)?
 - Łamaniu zasad programowanie obiektowego?
 - Tworzeniu kodu typu "wyłącznie do zapisu"?
 - Silnej optymalizacji ze względu na czas wykonania?

Demotywator

"Widziałem już zbyt wiele razy, że pisząc kiepski kod, i tak można zarobić duże pieniądze, aby uwierzyć, że jakość kodu jest niezbędna lub nawet wystarczająca do odniesienia komercyjnego sukcesu i zdobycia dużej popularności."

Kent Beck

Mimo wszystko będziemy się starać programować lepiej niż do tej pory...

Programując warto brać pod uwagę to, że ludzie jednak istnieją i że kod nie jest tworzony wyłącznie dla komputera.

Literatura

- Eric Freeman, Elisabeth Freeman, Kathy Sierra, Bert Bates. *Head First Design Patterns.*
- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Wzorce projektowe*
- Robert C. Martin (wujek Bob). *Czysty kod*
- Joshua Bloch. *Java Efektywne Programowanie*
- Kent Beck. *Wzorce implementacyjne*

Cechy obiektowego języka programowania - wg. wikipedii

Abstrakcja

obiekt - model abstrakcyjnego "wykonawcy" - obiekty mogą opisywać i zmieniać swój stan. Mogą komunikować się z innymi obiektami. Nie muszą ujawniać jak zaimplementowane są ich cechy.

Hermetyzacja

Stan wewnętrzny nie może się zmienić w nieoczekiwany sposób. Obiekt prezentuje innym interfejs - w nim są zawarte dopuszczalne metody współpracy.

Polimorfizm

Rozszerzamy pojęcie zgodności typu. Pewne typy są ze sobą zgodne, mogą więc występować w tym samym kontekście. Jeśli zachowanie będzie zależeć od tego jaki konkretnie obiekt się pojawi to mamy polimorfizm.

Dziedziczenie

Klasy bardziej szczegółowe mogą być budowane na podstawie bardziej ogólnych, a już istniejących.

Obiektowe co i jak

Klasa

W niej są metody i pola, czasami inne klasy. Elementy są publiczne lub nie.

Co

Metody dostępne dla użytkownika określają **co** można z klasą zrobić - jest to pewien rodzaj kontraktu pomiędzy twórcą klasy a jej użytkownikiem. Kontrakt określa **co** się stanie po wywołaniu określonej metody.

Jak

Implementacja metod pozwala na wykonanie obiecanej w kontrakcie pracy. Część metod może być ukryta przed użytkownikiem. **Jak** to działa może wiedzieć tylko autor kodu klasy.

Instancja

Instancja

Instancja (z łac. *instantia*) to w programowaniu obiektowym wystąpienie, początek bytu obiektu określonej klasy. Instancja oznacza konkretny obiekt istniejący w pamięci operacyjnej.

Świadome tworzenie obiektu (explicite)

Wiemy, że powstał obiekt, bo użyliśmy stosownego słowa kluczonego.

```
Czlowiek adam = new Pracownik();
```

```
String nazwisko = new String("Kowalski");
```

Tworzenie obiektu implicite

Obiekt powstał w sposób dla nas niezauważalny. Da się tak?

```
String nazwisko = "Nowak" + "-" + "Kowalska";
```

```
Long i = 0;
```

```
i += 1;
```

Elementy publiczne a prywatne

Z punktu widzenia naszego kursu:

Klasa składa się z:

publicznego interfejsu i prywatnej implementacji.

Interfejs

Zawiera dopuszczalne metody współpracy. Są tu wszystkie metody, które mogą wywoływać użytkownicy naszego kodu.

Hermetyzacja/Enkapsulacja

Kontrolujemy dostęp. Ukrywamy implementację. Dzięki temu nikt nie zmieni stanu naszego obiektu bez naszej wiedzy. Łatwiej wykrywać błędy - sami zmieniamy stan obiektu poprzez nasze metody.

Rozbudowa klas

Dziedziczenie a kompozycja

Kompozycja

składową klasy jest inna klasa. Relacja na zasadzie: *całość – część*.

Kompozycja – przykład

w klasie **Auto** mamy pola np. klasy **Silnik**, klasy **Koło**, klasy **szybaPrzednia**.

Dziedziczenie

nowa klasa specjalizuje istniejącą, bardziej ogólną. Relacja na zasadzie:
generalizacja – specjalizacja.

Dziedziczenie – przykład

na potrzeby programu kadrowego ogólna klasa **Człowiek** została wyspecjalizowana w klasie **Pracownik**.

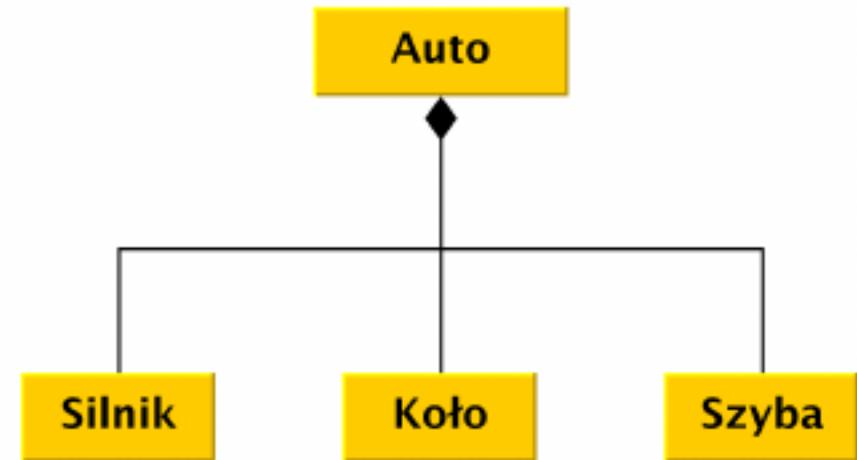
Obrazki UML

Nadkласа, superklаса, klasа bazowa



Podkласа, kласа походна

Rysunek: Dziedziczenie



Rysunek: Kompozycja

Agregacja całkowita (kompozycja) i częściowa różnią się tym kto kontroluje czas życia obiektów składowych.

Agregacja całkowita a częściowa

Kwestia techniczna

Potrzebujemy kontenera, który będzie przechowywać obiekty. Postanawiamy użyć kompozycji i gotowego rozwiązania w postaci jednej z typowych implementacji kolekcji (lista, tablica). W zależności od stosunku odczytów do zapisów, dane rozwiązanie będzie lepsze lub gorsze. Kto decyduje, która z implementacji będzie stosowana?

W przypadku agregacji całkowitej decyduje sama klasa naszego kontenera i sama może podjąć decyzję o zmianie sposoby przechowywania danych.

W przypadku agregacji częściowej element przechowujący dane musi zostać dostarczony z zewnątrz - nasz kontener jest tylko jego użytkownikiem. Dodatkowy problem: nasz kontener może nie być jedynym użytkownikiem elementu przechowującego.

Kwestia zarządzania pamięcią

Kto może usunąć obiekt z pamięci? W Java mamy Garbage Collector - on usunie obiekt za nas. Jeśli sami musimy usunąć obiekt, to trzeba śledzić czas jego życia.

Dziedziczenie - przykład

Program: Dziedziczenie

```
1 class Czlowiek {  
2     protected String dataUrodzenia = "1970-01-01";  
3     protected String nazwisko = "Nowak";  
4  
5     public String toString() {  
6         return "Jestem " + nazwisko + " urodzony " + dataUrodzenia;  
7     }  
8 }  
9 class Pracownik extends Czlowiek {  
10    private String stanowisko = "barman";  
11  
12    public String toString() {  
13        return "Jestem " + nazwisko + " urodzony " + dataUrodzenia +  
14            "; Stanowisko pracy " + stanowisko;  
15    }  
16 }  
17  
18 class Start {  
19     public static void main( String[] argv ) {  
20         System.out.println( new Czlowiek() );  
21         System.out.println( new Pracownik() );  
22     }  
23 }
```

Wynik pracy programu:

Jestem Nowak urodzony 1970-01-01

Jestem Nowak urodzony 1970-01-01; Stanowisko pracy barman

Polimorfizm

Polimorfizm (wielopostaciowość)

Obiekt klasy potomnej może zostać wysłany tam, gdzie oczekiwany (zadeklarowany) jest typ klasy macierzystej. Przy pewnych warunkach, wykonany zostanie kod metody zdefiniowanej w klasie potomnej, a nie wersja tej samej metody wg. klasy bazowej.

Przykład rozszerzonego pojmowania zgodności typów

Człowiek adam = new Pracownik();

Zadeklarowano (oczekiwano) na obiekt typu **Człowiek** i pojawia się **Pracownik**. Jeśli klasa **Pracownik** dziedziczy po **Człowiek** wszystko jest w porządku!

Klasa potomna może jeszcze nie istnieć

a my już możemy pisać program bazujący na własnościach (kontraktie) nadklasy. Kompilator zagwarantuje, że podklasy kontraktu klasy bazowej nie złamią.

Polimorfizm - przykład

Program: Polimorfizm

```
1 class Czlowiek {  
2     private String dataUrodzenia = "1970-01-01", nazwisko = "Nowak";  
3  
4     protected String getString() {  
5         return "Jestem " + nazwisko + " urodzony " + dataUrodzenia;  
6     }  
7     public String toString() { return getString(); }  
8 }  
9 class Pracownik extends Czlowiek {  
10    private String stanowisko = "barman";  
11  
12    public String toString() {  
13        return getString() + "; Stanowisko pracy " + stanowisko;  
14    }  
15 }  
16  
17 class Start {  
18    public static void show( Czlowiek czl ) {  
19        System.out.println( czl );  
20    }  
21    public static void main( String[] argv ) {  
22        show( new Pracownik() );  
23    }  
24 }
```

Wynik pracy programu:

Jestem Nowak urodzony 1970-01-01; Stanowisko pracy barman

Interfejsy

Interfejs

Interfejsy

Wygodna forma projektowania obiektowego. Interfejs używany w Java do niedawna nie miał prawa zawierać definicji metod, a tylko ich deklaracje.

Instancje interfejsu

Nie można utworzyć instancji interfejsu.

Implementacja interfejsu

Klasy mogą implementować metody jednego interfejsu lub z wielu interfejsów. Interfejs jest wtedy super-typem, a klasa go implementująca jest pod-typem.

Interfejs to typ!

Interfejs jest typem. Nazwy interfejsu można używać tak jak nazwy bazowej klasy. Do referencji zadeklarowanej wg. typu interfejsu można przesłać referencję do obiektu klasy implementującej ten interfejs.

Interfejsy a ponowne użycie kodu

Implementacja

Klasa musi implementować wszystkie metody zadeklarowane w interfejsie albo być klasą abstrakcyjną.

Gotowa implementacja

Wraz z interfejsem można otrzymać gotową implementację w postaci klasy. Jak jej użyć gdy nie możemy jej odziedziczyć?

Rozwiązanie

Należy użyć kompozycji. Referencja do obiektu klasy implementującej interfejs musi należeć do nowej klasy jako jej pole.

Interfejsy a ponowne użycie kodu – przykład

Program: Interfejsy a ponowne użycie kodu

```
1 interface WaznyInterface {  
2     void showBlaBlaBla();  
3 }  
4  
5 class KlasaImplementujaca implements WaznyInterface {  
6     public void showBlaBlaBla() {  
7         System.out.println( "Bla Bla Bla.. " );  
8     }  
9 }  
10  
11 class Start implements WaznyInterface {  
12     final private WaznyInterface wi = new KlasaImplementujaca();  
13  
14     public void showBlaBlaBla() {  
15         wi.showBlaBlaBla();  
16     }  
17  
18     public static void main( String[] argv ) {  
19         ( new Start() ).showBlaBlaBla();  
20     }  
21 }
```

Klasy abstrakcyjne a interfejsy

Porównanie klas abstrakcyjnych i interfejsów w Java

<i>cecha</i>	<i>klasa abstrakcyjna</i>	<i>interfejs</i>
dziedziczenie	jednokrotne	wielokrotne
implementacja	możliwa, nieobowiązkowa	od Java 8 tak
public	tak	tylko takie
protected	tak	nie
private	tak	od Java 9 tak
pola	tak	tylko stałe !!!
metody finalne	tak	nie (nawet z private)

Dobry projekt klasy

Główne klasy projektu, te które będą rozszerzane, powinny być implementacjami interfejsu. Nawet jeśli będą to klasy abstrakcyjne!

Wszystkie metody abstrakcyjne

Klasa abstrakcyjna, której wszystkie metody są abstrakcyjne powinna się pojawić raczej jako interfejs.

Analiza i projektowanie obiektowe

Object-oriented analysis and design

Po co używać OOA&D?

Z punktu widzenia klienta istotne jest gdy tworzone dla nich aplikacje:

- *działają* - czyli program robi to, czego oczekuje od niego klient. Klient ma zawsze rację i trudno oczekiwany, że będzie zadowolony z aplikacji, która nie spełnia jego oczekiwania.
- *działają dłujo* - działają poprawnie nawet jeśli będą używane w niespodziewany sposób. W trakcie projektowania należy przewidzieć również sytuacje niezwykłe. Nie zawsze wszystko działa zgodnie z optymistycznym scenariuszem zdarzeń. Np. dokument wysłany do druku jednak nie został wydrukowany.
- *można je uaktualniać* - w rozsądny czasie i za rozsądne pieniądze.

Po co używać OOA&D?

Z punktu widzenia programisty istotne jest gdy tworzony przez niego kod:

- *można wielokrotnie używać* - zasady projektowania obiektowego pozwalają na stworzenie kodu, który po niewielkich modyfikacjach będzie można użyć w nowym projekcie i ponownie sprzedać. Pomaga np. takie pisanie kodu, w którym klasy nie są od siebie uzależnione i zbyt mocno ze sobą powiązane.
- *jest elastyczny* - łatwo wprowadzać w nim zmiany, jest prosty i tani w utrzymaniu

Dla kogo jest ważne OOA&D

Okazuje się, że tak naprawdę analiza i projektowanie obiektowe potrzebne jest bezpośrednio tylko programiście (dostawcy programu)!

Wymaganie

Wymaganie wg. wikipedii

Wymaganie w inżynierii, jest pojedynczą, udokumentowaną potrzebą określonego produktu czy usługi, albo sposobu ich działania. Jest to stwierdzenie identyfikujące potrzebne cechy, możliwości, charakterystyki lub jakość systemu, aby był on wartościowy i pożyteczny dla użytkownika. W klasycznej inżynierii, zbiór wymagań jest wykorzystywany w fazie projektowania nowego produktu. Wymagania pokazują, jakie elementy i funkcje są niezbędne w konkretnym projekcie.

Wymaganie

Jest to pewna szczególna i szczegółowo określona potrzeba, którą system musi wykonywać aby można było uzać, że działa on poprawnie.

Zbiór wymagań

definiuje charakterystyki lub cechy oczekiwanejego systemu. 'Dobra' lista wymagań generalnie nie mówi jak wymagania są implementowane, pozostawiając to projektantowi systemu.

Gromadzenie wymagań

Klient

trzeba pozwolić mu mówić i spisać (zrozumieć) o co mu chodzi... Klient może nie być świadomy wszystkich wymagań!

Wymagania powinny być jasne

Wymagania powinny być łatwe do zrozumienia zarówno przez użytkowników jak i projektantów. Dokumentacja wymagań pozwala na stwierdzenie co system będzie robił.

Podejście typu "to nie mój problem..."

Za problemy odpowiada twórca systemu - jeśli o coś nie zapyta, nie przewidzi konsekwencji jakiejś czynności, to on będzie odpowiadać za złą pracę aplikacji, a nie klient.

Zmiany

jedyna pewna rzecz to zmiany... OOA&D ma pozwolić je łatwo wprowadzić.

Gromadzenie wymagań

Więcej!

Użytkownik oczekuje, że system będzie działać poprawnie mimo pojawienia się problemów. Oznacza to konieczność dodania wymagań, które umożliwią rozwiążanie problemu. Wymagań jest więc zazwyczaj więcej niż podaje (uświadamia sobie) klient.

Zrozumienie co system ma robić!

"Im więcej rozumiesz tym mniej musisz zapamiętać". Projektant systemu musi rozumieć co system ma robić.

Przypadek użycia

Przypadek użycia (use case)

przedstawia interakcję pomiędzy aktorem (użytkownikiem systemu), który inicjuje zdarzenie oraz samym systemem jako sekwencję prostych kroków. To lista kroków, jakie muszą zostać wykonane aby coś się stało.

Przypadek użycia opisuje

co system musi zrobić aby konkretne wymaganie klienta (cel) zostało zaspokojone.

Uwagi:

- Jeśli mamy wiele wymagań to i wiele przypadków użycia.
- P.U. powinien być pozbawiony szczegółów dotyczących implementacji oraz interfejsu użytkownika
- Opisywać system na właściwym poziomie szczegółowości
- P.U. musi posiadać oczywiste znaczenie dla systemu i stosowną nazwę.
- Musi posiadać jasno określony punkt rozpoczęcia i zakończenia
- P.U. inicjuje zewnętrzny czynnik - coś co istnieje poza systemem.

Przypadek użycia - scenariusze

Scenariusze

Każdy przypadek użycia składa się z co najmniej jednego scenariusza przedstawiającego interakcję użytkownik–system lub system–system. W wyniku ma zostać zrealizowany jakiś konkretny cel. Scenariusz to kompletna ścieżka prowadząca od początku do końca przypadku użycia.

Ścieżka główna/optymalna

inaczej szczęśliwa ścieżka wydarzeń (basic flow / happy flow) przedstawia podstawowy przebieg operacji

Ścieżki alternatywne

opis sytuacji, gdy ścieżka optymalna nie zachodzi czyli obsługa problemów w działaniu systemu.

Przypadek użycia - przykład

Domofon

Cel: wpuścić wybrane osoby bez konieczności wychodzenia z domu.

Przypadek użycia:

- 1 Ktoś przychodzi
- 2 Przycisk guzik dzwonka
- 3 Za pomocą rozmowy ustalamy kto to jest
- 4 Jeśli ta osoba powinna wejść, to
 - 4.1 Naciskamy przycisk otwierający drzwi furtki
 - 4.2 Drzwi się otwierają
 - 4.3 Uprowadzona osoba wchodzi do środka

Czy to wszystko?

Przypadek użycia - przykład

Domofon

Cel: wpuścić wybrane osoby bez konieczności wychodzenia z domu.

Przypadek użycia:

- 1 Ktoś przychodzi
- 2 Przyciski guzik dzwonka
- 3 Za pomocą rozmowy ustalamy kto to jest
- 4 Jeśli ta osoba powinna wejść, to
 - 4.1 Naciskamy przycisk otwierający drzwi furtki
 - 4.2 Drzwi się otwierają
 - 4.3 Uprawniona osoba wchodzi do środka
 - 4.4 Automat blokuje drzwi furtki
 - 4.4.1 Jeśli automat zablokuje drzwi furtki przed wejściem uprawnionej osoby to idz do 2.

Lista wymagań - przykład

Wymagania:

- 1 Przycisk dzwonka powinien być łatwo dostępny
- 2 Naciśnięcie przycisku dzwonka powoduje, że w domu słyszać dzwonek
- 3 System pozwala na przeprowadzenie rozmowy dom-furtka
- 4 Naciśnięcie stosownego przycisku otwiera furtkę
- 5 System automatycznie blokuje drzwi furtki

Warto porównać listę wymagań z przypadkami użycia. Przy każdym z punktów przypadku użycia można wpisać numer wymagania i sprawdzić, że wszystkie wzięto pod uwagę.

Przypadek użycia - zyski

Zyski z posiadana przypadku użycia:

- Wiemy jak system ma działać
- Rosną szanse, że system będzie działać w praktyce (w rzeczywistym kontekście)
- Istnieje możliwość przeprowadzenia analizy tekstowej

Analiza tekstowa przypadku użycia

polega na poszukiwaniu w opisie rzeczowników i czasowników. Rzeczowniki wskazują klasy, czasowniki zaś metody.

Przypadek użycia - analiza tekstowa

- 1 Ktoś przychodzi
- 2 Przyciska guzik dzwonka
- 3 Za pomocą rozmowy ustalamy kto to jest
- 4 Jeśli ta osoba powinna wejść, to
 - 4.1 Naciskamy przycisk otwierający drzwi furtki
 - 4.2 Drzwi się otwierają
 - 4.3 Uprowadiona osoba wchodzi do środka
 - 4.4 Automat blokuje drzwi furtki
 - 4.4.1 Jeśli automat zablokuje drzwi furtki przed wejściem uprowadionej osoby to idz do 2.

Przykładowe klasy: Dzwonek, Furtka, Automat (coś odmierzającego czas)

Metody: dzwonek.nacisnac(), furtka.otworzyc(), furtka.zablokowac(),

Uwaga: jak widać mamy pewne braki w dokumentacji. Np.: jak ma się odbywać rozmowa? Brak słowa domofon i informacji jak ma być użyty.

Przypadek użycia - analiza tekstowa

Analiza tekstowa podaje *sugestie*. Nie wszystkie rzeczowniki wygenerują klasy, bo nie wszystkie klasy są nam potrzebne.

Trzeba zwracać uwagę na powtórzenia. Ktoś == Osoba, Furtka == DrzwiFurtki.

W przypadku użycia pojawiają się elementy, które nie należą do naszego systemu, więc klasy im odpowiadające nie są w implementacji systemu potrzebne.
Przykładem takiej klasy jest Osoba.

Karty CRC

Karta CRC (od ang. Class, Responsibilities and Collaboration)

karta (kartka papieru), na której zapisujemy nazwę klasy (u góry) oraz z kim (po prawej) i w czym (po lewej) współpracuje. Wykrywamy z kim i jakie dane nasza klasa wymienia.

Symulacja systemu z użyciem kart CRC

Grupa osób odgrywa role klas. Jedna osoba może być opiekunem jednej lub wielu klas. Jest to alternatywa do rysowania diagramu sekwencji UML.

Wbrew pozorom taka "scenka" pozwala na wykrycie braków w interfejsach i np. stwierdzenie, że większość pracy ma wykonywać pojedyncza (tzw. boska) klasa. Największy problem — przełamanie się osób uczestniczących w symulacji, bo uważają, że jest to *infantylne*...

Podstawowe zasady projektowania obiektowego

Zasady projektowania obiektowego to techniki, które zastosowane w trakcie projektowania ułatwiają jego późniejsze utrzymanie, rozbudowę i zwiększą jego elastyczność.

Zmiany

Zmiany

Zmian nie sposób uniknąć. Tylko dobry projekt pozwala na wprowadzanie ich w łatwy sposób. Dobra projektujemy dla siebie samych !!!

Następstwa zmian

Nawet niewielka zmiana specyfikacji może doprowadzić do poważnych problemów z systemem. Jeśli zmiana pociąga za sobą konieczność modyfikacji kodu w wielu miejscach, to łatwo jest coś przeoczyć.

Jeden powód do zmiany

Każda klasa powinna posiadać tylko jeden powód do zmiany. Poprzez podział i kompozycję zmniejszamy liczbę czynników, które mogą wymusić zmianę klasy. Klasa nie powinna realizować za dużo zadań.

Klasy abstrakcyjne

Wzór

Klasa abstrakcyjna stanowi wzór do tworzenia implementacji. Może dostarczać części metod, które już na tym etapie mogą zostać napisane.

Definicja zachowania

Klasa abstrakcyjna definiuje zachowanie, które zostanie zaimplementowane przez jej dziedziców.

Wspólne zachowania

Zawsze gdy zachowanie się powtarza pojawia się możliwość wydrębnienia go i umieszczenia w osobnej (abstrakcyjnej) klasie.

Programowanie z użyciem interfejsu

Java a "interfejs"

Interfejs oznacza tu (nad)typ. Klasa abstrakcyjna i *interface* to typowe nadtypy. W Java nie mamy wielodziedziczenia, ale możemy implementować wiele interfejsów.

Użycie

Zamiast używania w deklaracjach referencji nazw konkretnych klas należy, jeśli tylko to możliwe, używać nadtypu.

Program, który bazuje wyłącznie na typach bazowych jest łatwiejszy do rozszerzenia i bardziej elastyczny - w przyszłości, bez zmian, będzie mógł używać innych klas, które implementują czy rozszerzają użyte w nim typy bazowe.

Rozszerzenie pojęcia hermetyzacji

Hermetyzacja zazwyczaj rozumiana jest jako tworzenie w klasie pól i metod, które nie są publicznie dostępne.

Hermetyzacja zachowania.

Jeśli jest podejrzenie, że zachowanie może ulec zmianie, należy je oddzielić od tych części aplikacji, które nie będą zbyt często modyfikowane. Poprawia się elastyczność aplikacji. Łatwiej potem wprowadzać zamiany.

Usuwanie powielonego kodu

Powielony kod można zamknąć w osobnej klasie - to także forma hermetyzacji.

Spójność

Spójność/kohezja/zwartość/spoistość

Terminu tego używa się w odniesieniu do klasy lub modułu na oznaczenie wzajemnego zintegrowania procedur, metod składowych architektury, itp. *Duża kohezja oznacza silną interakcję wewnętrz modułu lub klasy i relatywnie słabszą interakcję z otoczeniem. Klasy powinna cechować duża kohezja.*

Spójna klasa

naprawdę dobrze realizuje jedną operację. Nie robi tego czym nie jest. Jest dobrze zdefiniowana.

Jeden powód do zmiany

Jedna klasa — jeden powód do zmiany!

Mały przykład

Problem

Tworzymy program, w którym gracz niszczy wirtualne cele za pomocą wirtualnego działa. Musimy stworzyć implementację działa, która na podstawie parametrów otoczenia i ustaw działa odpowie gdzie upada pocisk. Jakie klasy mogą się nam przydać? Co mają zrobić?

Oto lista przewidywanych zmian:

- Działo zmienia parametry - predkość wylotową pocisku, kaliber, minimalny i maksymalny kąt uniesienia lufy
- Aby gra była bardziej realistyczna uwzględniamy opór, ciśnienie, wilgotność i temperaturę powietrza
- Chcemy uwzględnić czas przeładowania działa
- Chcemy uwzględnić ograniczoną celność działa
- Chcemy zmienić rozkład rozrzutu pocisków
- Chcemy uwzględnić kształt pocisku

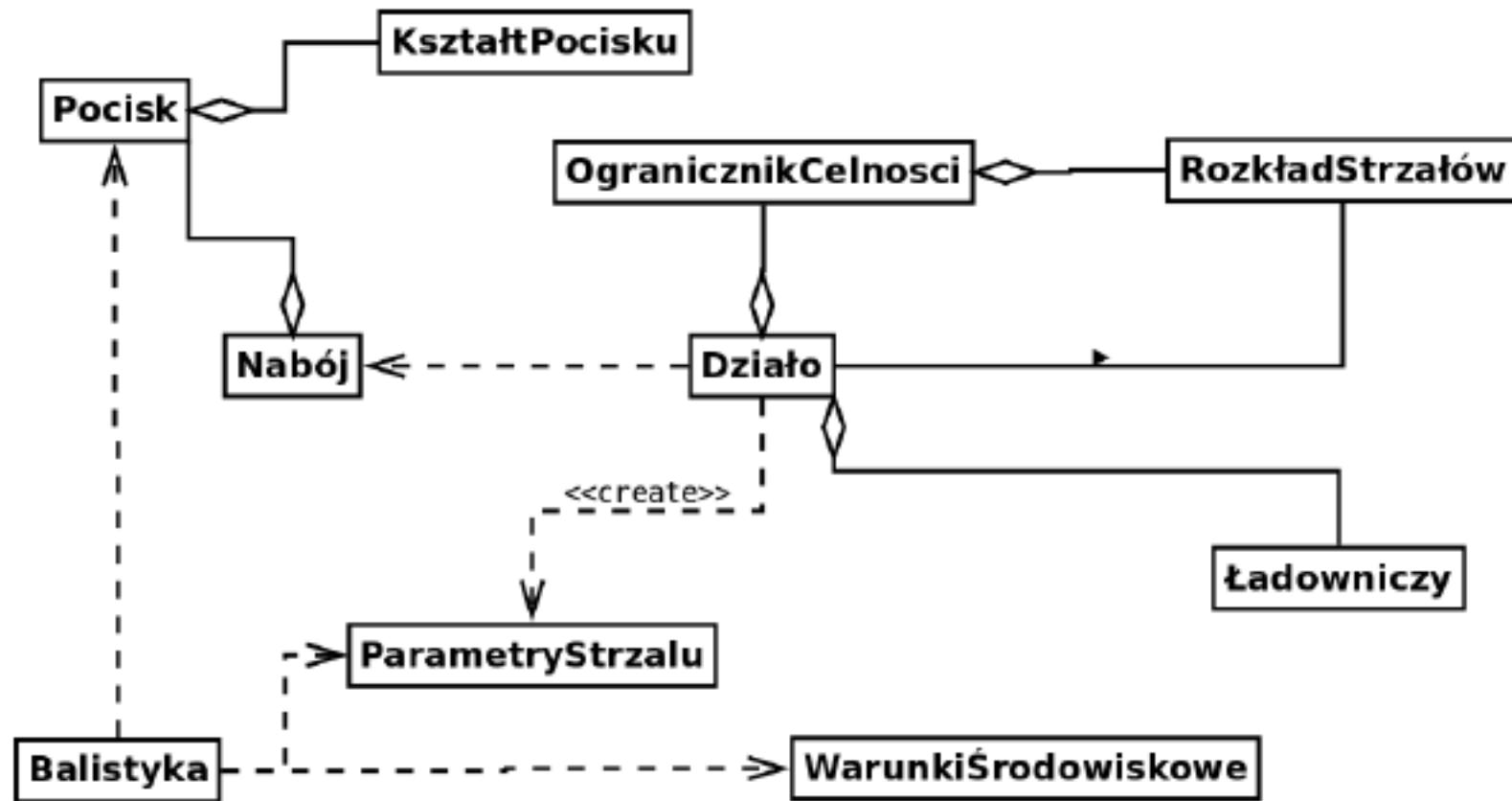
Mały przykład - klocki

Fundamentalne pytanie

Problem to zmiana kodu źródłowego, a nie utworzenie nowego obiektu z innymi parametrami.

- KształtPocisku
- Pocisk
- Nabój
- Działo
- OgranicznikCelności
- RozkładStrzałów
- Ładowniczy
- ParametryStrzalu
- Balistyka
- WarunkiŚrodowiskowe

Mały przykład - klocki



Rysunek: Połącznie klocków

Zasada otwarte-zamknięte (Open-Close Principle)

Klasy powinny być otwarte na rozbudowę i zamknięte na modyfikację

Czyli?

Klasa powinna dać się rozbudować, ale bez konieczności zmiany kodu, który już istnieje!

Czyli?

Pozwalamy na zmianę zachowania klasy ale np. poprzez dziedziczenie czyli tworzenie klas potomnych. Dzięki polimorfizmowi uzyskamy zmianę zachowania pomimo pozostawienia oryginalnego kodu w pierwotnej postaci.

Metody prywatne a OCP

Gdy klasa pozwala na wywoływanie swoich metod prywatnych za pomocą odpowiednich metod nieprywatnych można rozszerzać zachowanie metod prywatnych jednocześnie niczego w nich nie zmieniając.

SRP

Zasada jednej odpowiedzialności (Single Responsibility Principle)

Jedna klasa - jedno zadanie. Klasa ma być stworzona do rozwiązywania jednego problemu. Jeśli coś trzeba zmienić to dzięki tej zasadzie wiadomo dokładnie gdzie należy tych zmian dokonać.

To działa w dwie strony

Jedna klasa - jedno zadanie. Jedno zadanie powinno być realizowane przez jedną klasę czyli nie powielamy tej samej funkcjonalności.

SRP a DRY

Są ze sobą powiązane.

SRP

Powoduje, że klasy rosną — funkcjonalność nie jest rozbijana na wiele małych klas.

SRP — test

Wykrywanie wielu funkcjonalności

Zapisujemy zdania typu: "NazwaKlasy nazwaMetody się/sobie" i je czytamy. Czy mają one sens? Zadania powinny pasować do nazwy klas. Jeśli w obiektach danej klasy atrybuty często mają wartość null to może wskazywać, że klasa wykonuje za dużo zadań.

Poprawne nazwy klas i metod

Bez poprawnego, jasnego nazywania klas i ich metod, tego typu sposoby nic nie dadzą.

A przypadek użycia?

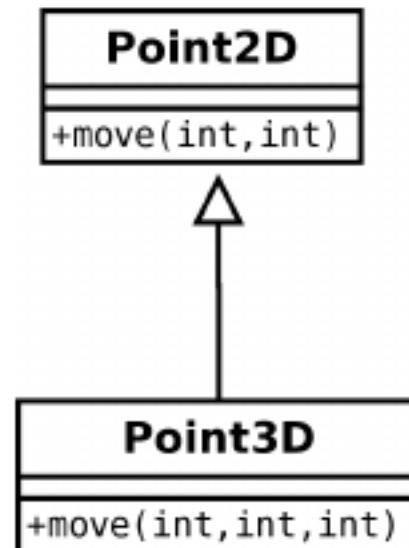
Jeśli zadbaliśmy o posiadanie poprawnego przypadku użycia to analiza tekstowa już była przeprowadzana - powinna ona pomóc w znalezieniu dobrych kandydatów na klasy.

SRP — przykład

- Mrówka.przemiesć() się
- Mrówka.wyświetlOkno()
- Okno.wyświetl() się
- Rower.wymieńŁańcuch()
- Długopis.zróbZakupy()
- Piekarnik.podajTemperature()

Zasada podstawiania Liskov (Liskov Substitution Principle)

Typy pochodne muszą dać się podstawić w miejsce ich typów bazowych nie tylko od strony technicznej (zgodność refencji), ale i użytkowej (wyzyskanie zalet polimorfizmu). Ta zasada pozwala prawidłowo zaprojektować klasy z użyciem dziedziczenia.



Rysunek: Przykład negatywny dla LSP

Prosty test: czy Punkt3D **jest** Punkt(em)2D?

Zasada segregacji interfejsów (Interface segregation principle)

Klienci nie powinni zależeć od interfejsów, których nie używają.

Wyjaśnienie

W tej zasadzie chodzi o definiowanie spójnych i zwięzłych interfejsów. Interfejsy powinny deklarować możliwie najmniejszą liczbę metod. Metody zaś powinny być ze sobą powiązane. Zaleca się tworzenie mniejszych kontraktów. Zwięzłość interfejsów zwiększa szanę na ponowne użycie kodu.

Kiedy sprawy idą źle?

Gdy okazuje się, że wiele metod, które musi zaimplementować klasa aby być zgodna z pewnym interfejsem, nie zawiera wcale kodu (lub jedyne co robią, to zgłaszają wyjątek informujący o braku implementacji) to na pewno mamy problem z przestrzeganiem ISP. Przy okazji z LSP także.

SOLID

Poznane zasady łączą się razem. Zostały one przedstawione przez Roberta C. Martina w "Design Principles and Design Patterns". Sam mnemonik/akronim SOLID zaproponował Michael Feathers.

- S - Single responsibility principle (Zasada jednej odpowiedzialności)
- O - Open/closed principle (Zasada otwarte-zamknięte)
- L - Liskov substitution principle (Zasada podstawienia Liskov)
- I - Interface segregation principle (Zasada segregacji interfejsów)
- D - Dependency inversion principle (Zasada odwrócenia zależności)

DIP

Wysokopoziomowe moduły, nie powinny zależeć od modułów niskopoziomowych.
Tyle w tej chwili. Więcej w przyszłości...

Zasada nie powtarzaj się (Don't Repeat Yourself)

Wspólne fragmenty kodu powinny zostać wyodrębnione i umieszczone w jednym miejscu.

Inaczej

Każda informacja czy zachowanie ma być umieszczone w jednym miejscu. Jedno wymaganie, jedna możliwość, jedno miejsce. Po prostu: nie duplikować kodu!

Jeszcze inaczej

Duplicate is evil

Zasada "Keep it simple, stupid" lub "Keep it stupid simple".

Polski odpowiednik to BUZI czyli "Bez Udziwnień Zapisu, Idioto".

Regułka pochodzi z 1960 z U.S. Navy od Kelly Johnson-a - chodziło o takie projektowanie samolotów, aby można je było naprawić w warunkach polowych przez średnio uzdolniony personel używający do tego celu ograniczonego zestawu narzędzi.

Bjarne Stroustrup "Make Simple Tasks Simple!"

Generalnie chodzi o to, aby do rozwiązania prostego problemu nie stosować skomplikowanych metod.

Nie poluje się pancernikami na pijawki, choć może to być widowiskowe, ale jest przede wszystkim bardzo kosztowne!

Zasada "You ain't gonna need it")

Nie będziesz tego potrzebować.

Nie należy tworzyć kodu (a także funkcjonalności), który nie jest potrzebny. Nie tworzymy kodu "na zapas".

Kod, którego nie potrzebujemy powinien być bezwzględnie usuwany. Po to są systemy kontroli wersji aby, jeśli będzie taka potrzeba, do usuniętego kodu można było powrócić.

Inne zasady

SoC - Separation of Concern - Separacja zagadnień

Program powinien zostać podzielony na moduły o rozłącznych (a przynajmniej o najmniej pokrywających się) funkcjonalnościach. Dzięki temu poprawia się czytelność i reużywalność. Dodatkowo, praca grupowa jest uproszczona.

MoSCoW (M – Must have it, S – Should have it, C – Could have if not affecting other things, W – Won't have this time).

Zasada MoSCoW dzieli wymagania klienta na kilka grup o różnych priorytetach. M - muszą być, S - powinny być, C - mogą być, o ile nie szkodzą innym funkcjonalnościom, W - nieosiągalne w podanym terminie.

Dziedziczenie?

Alternatywy dla dziedziczenia

Delegacja

Przekazujemy odpowiedzialności za wykonanie zadania innej klasie lub metodzie. Używamy, gdy funkcjonalności oferowanej przez inną klasę chcemy użyć bez jej modyfikacji.

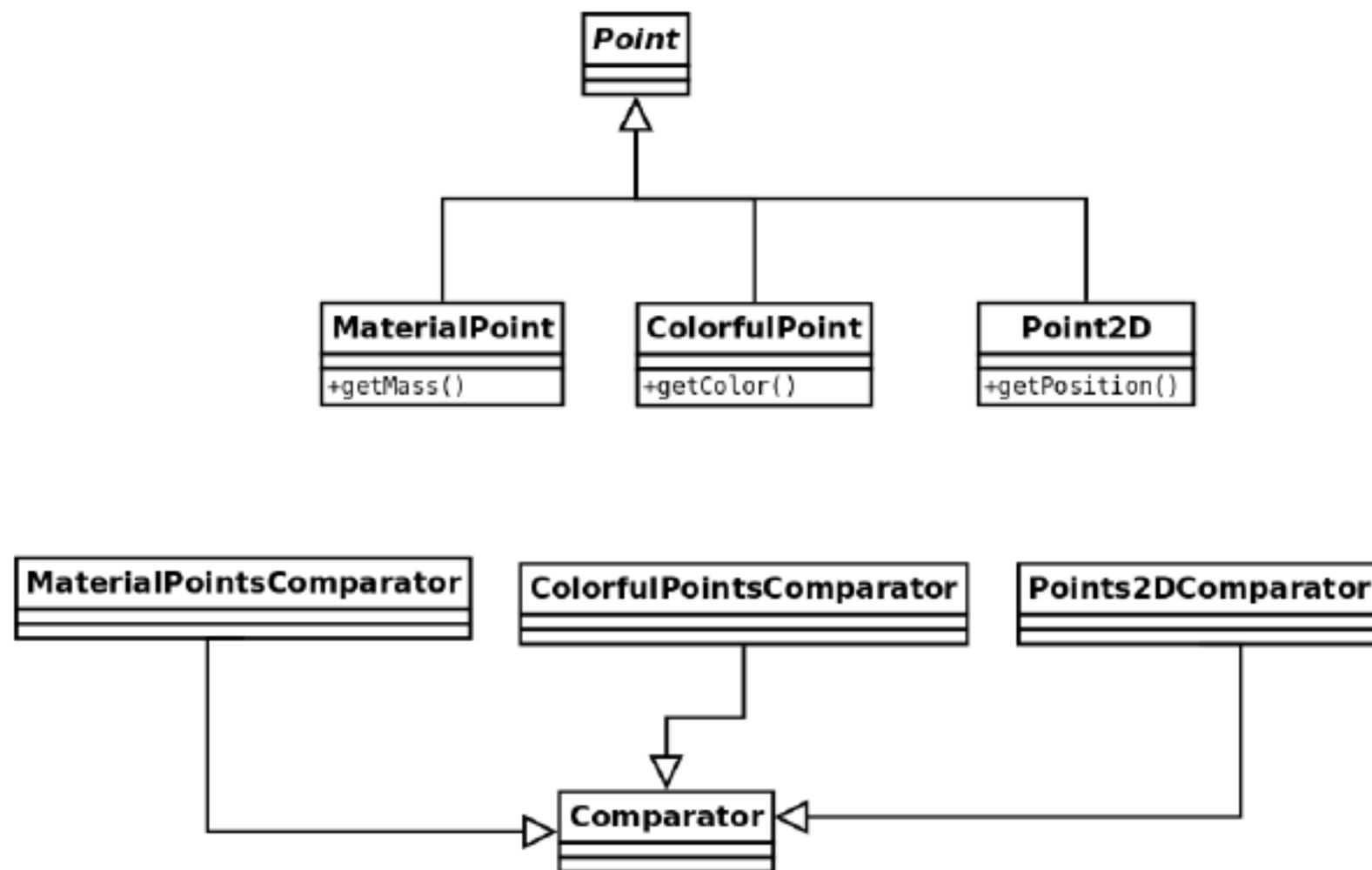
Kompozycja

Używamy zachowania oferowanego przez rodzinę różnych klas. Zachowanie to możemy zmieniać w trakcie pracy programu. To nasza klasa decyduje jakie z zachowań jest optymalne w danej chwili - stąd mówimy o kompozycji, bo to nasza klasa odpowiada za czas życia obiektów

Agregacja

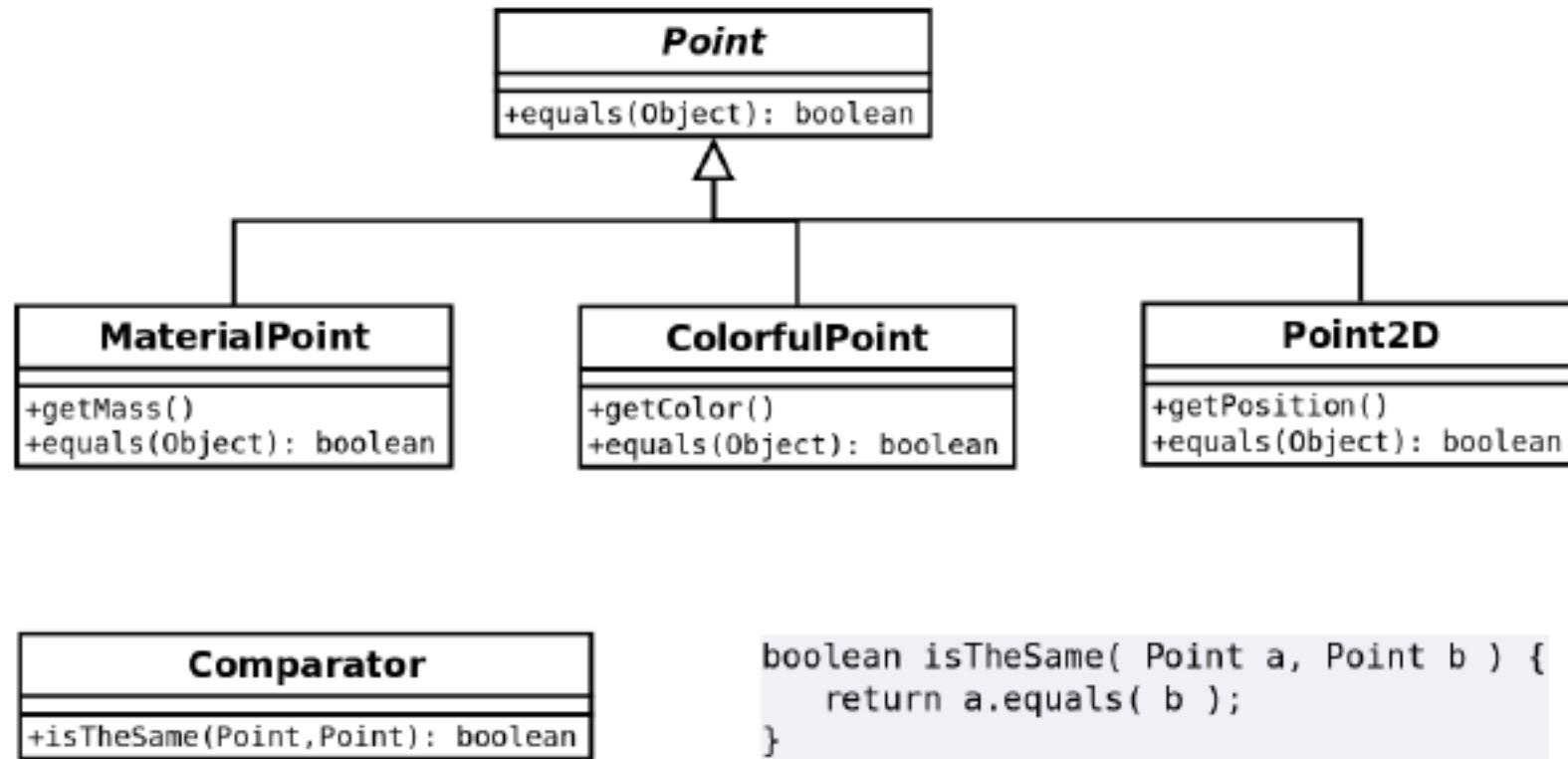
Podobne do kompozycji, ale obiekty klasy, której używamy mogą istnieć także poza naszą klasą.

Delegacja - najpierw problem



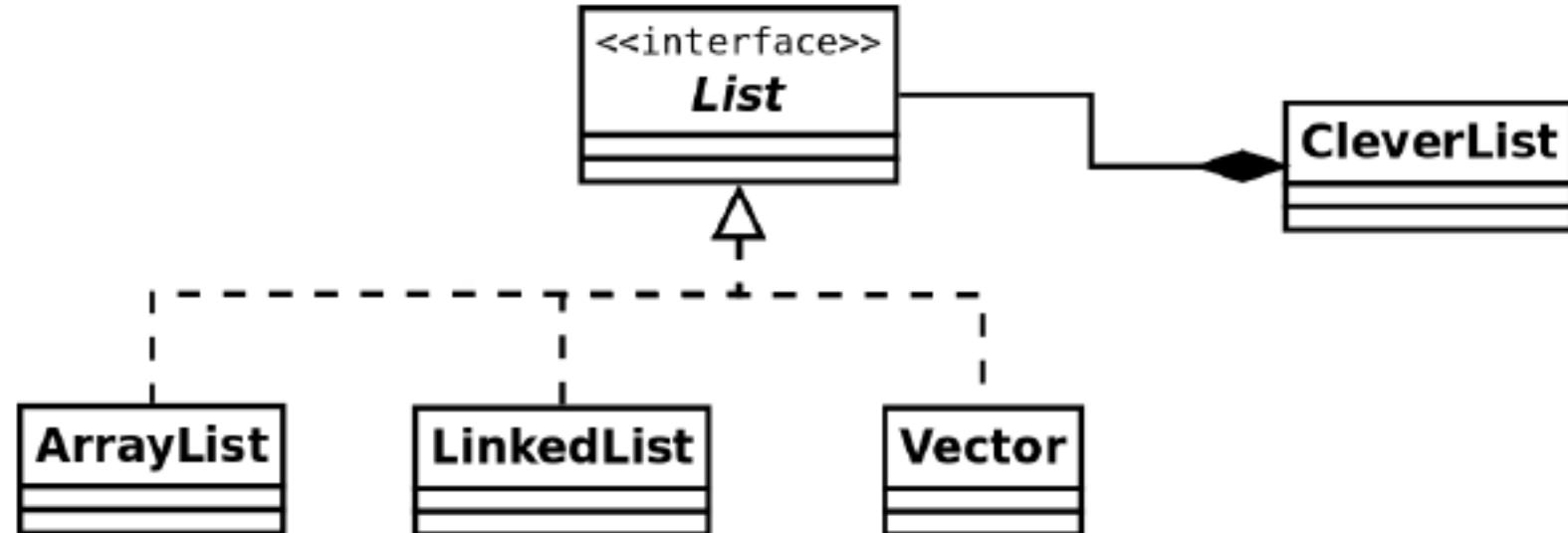
Rysunek: Złe użycie dziedziczenia. Nie działa polimorfizm, bo klasy potomne wprowadzają nowe metody.

Delegacja - rozwiązanie problemu



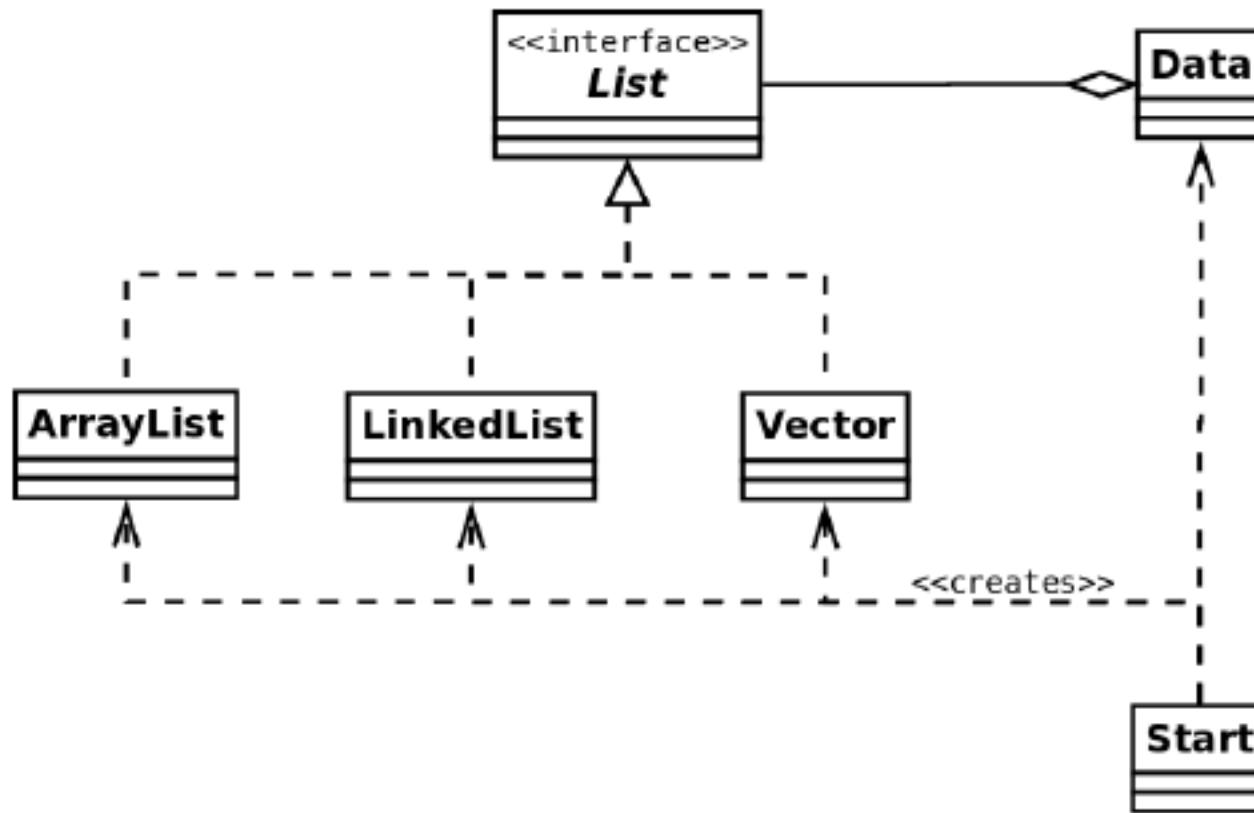
Rysunek: Rozwiążanie problemu za pomocą delegacji.

Kompozycja



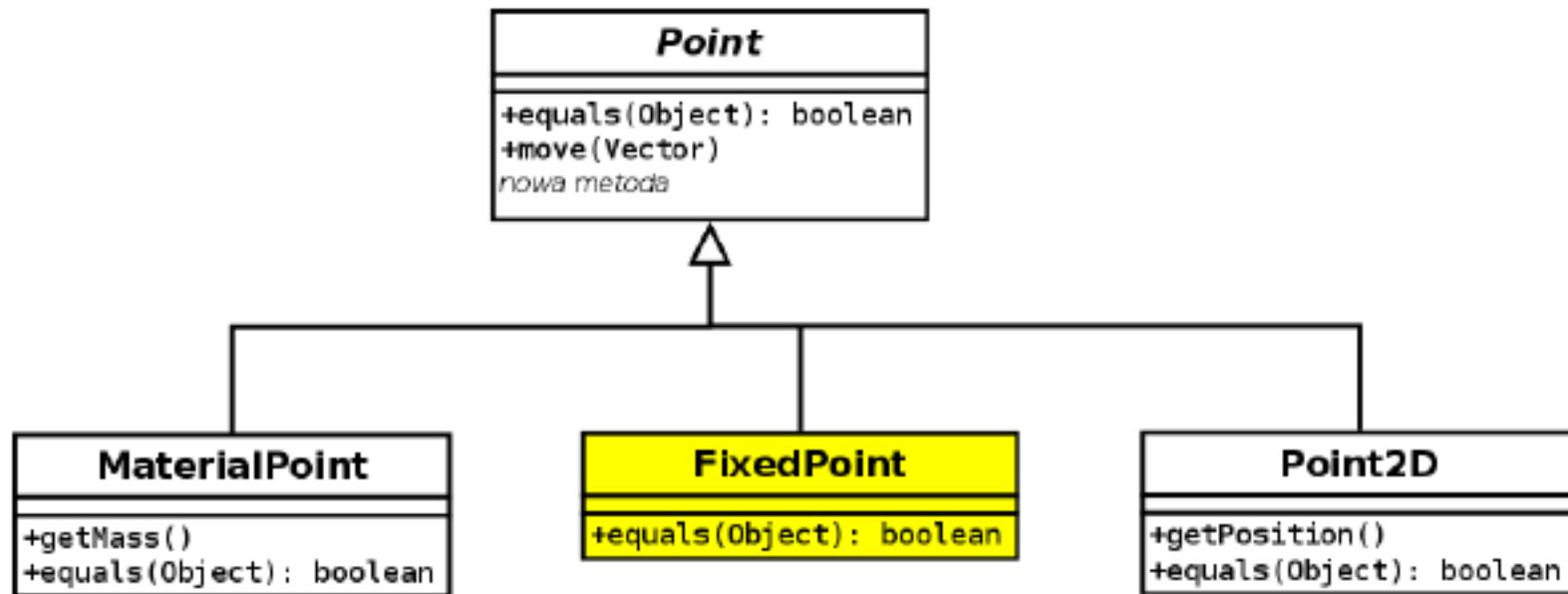
Rysunek: Inteligentna lista.

Agregacja



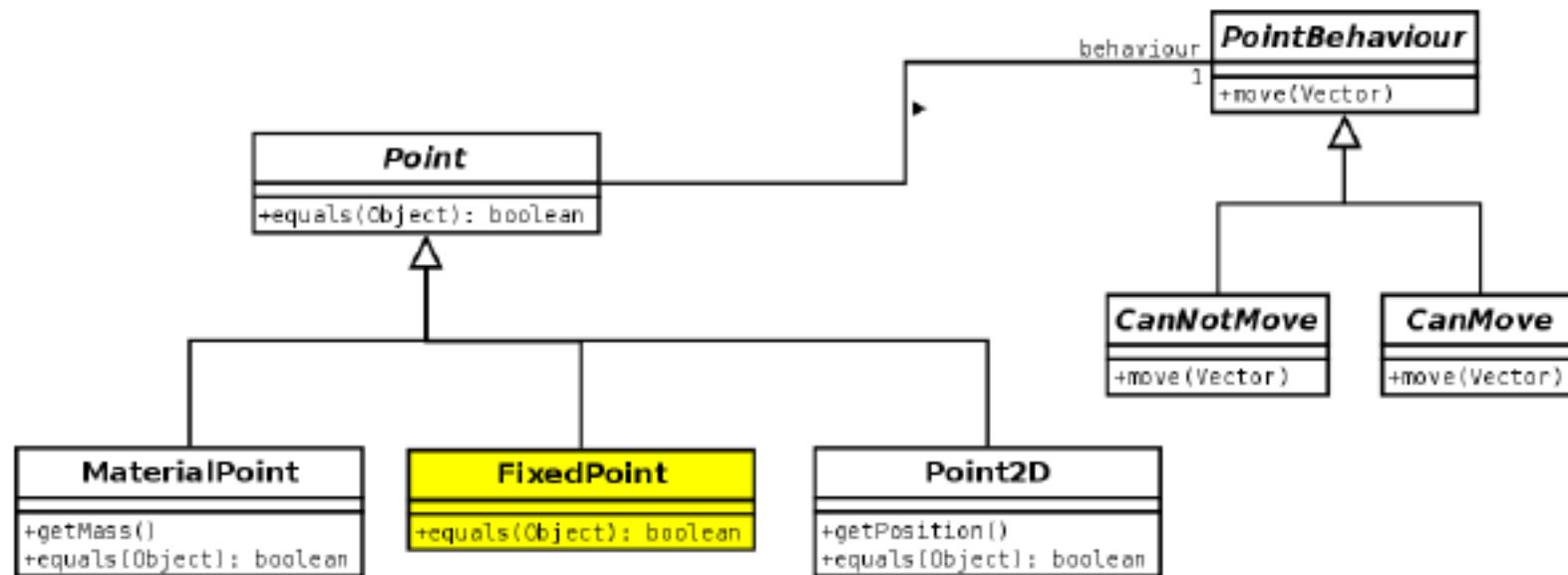
Rysunek: Dane gromadzone w postaci listy. Klasa gromadząca dane nie wie jaka implementacja listy będzie optymalna.

Kolejny problem z dziedziczeniem



Rysunek: Właśnie dodano nową metodę (`move`) do klasy bazowej. Metoda ta znacząco zmienia zachowanie potomków. Klasa **FixedPoint** do tej pory reprezentowała punkt o ustalonych współrzędnych, teraz już nie.

Kolejny problem z dziedziczeniem - pomaga hermetyzacja tego co może ulec zmianie



Rysunek: Hermetyzacja tego co zmienne

Kolejna zasada projektowania

Lepsza jest relacja "ma" niż "jest". Preferuj kompozycję nad dziedziczeniem. To pozwala zmienić zachowanie w momencie uruchomienia programu.

Wzorce projektowe

Wzorzec projektowy

Wzorzec projektowy wg. wikipedii

Uniwersalne, sprawdzone w praktyce rozwiązanie często pojawiających się, powtarzalnych problemów projektowych. Pokazuje powiązania i zależności pomiędzy klasami oraz obiektami i ułatwia tworzenie, modyfikację oraz pielęgnację kodu źródłowego. Jest opisem rozwiązania, a nie jego implementacją. Wzorce projektowe stosowane są w projektach wykorzystujących programowanie obiektowe.

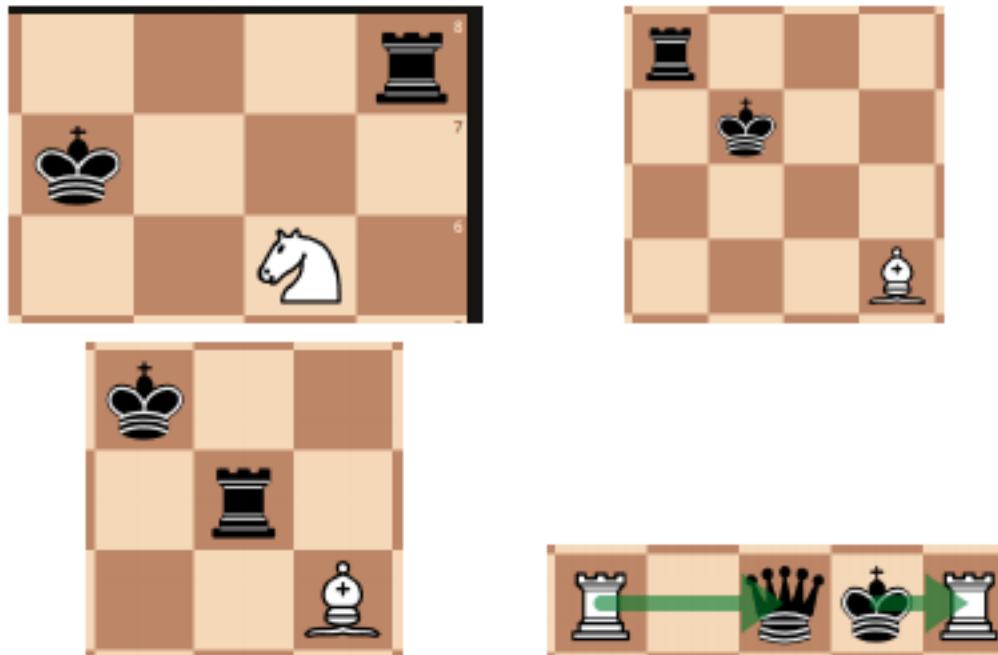
Algorytm

nie jest wzorcem projektowym - to przepis na rozwiązanie problemu obliczeniowego, a nie projektowego.

Wzorce projektowe

stanowią abstrakcyjny opis zależności pomiędzy klasami, co w efekcie wprowadza pewną standaryzację kodu oraz zwiększa jego zrozumiałość, wydajność i niezawodność. To sposób na ponowne użycie nie tyle kodu, co doświadczenia.

Wzorzec projektowy - dygresja szachowa



Czy znając te taktyki (zawsze) wygramy?

Niestety, ale nie. Ale zwiększą one nasze szanse!

Wzorzec projektowy - zyski

Wspólny język

Zamiast szczegółowo opisywać rozwiązanie wystarczy podać jaki wzorzec został użyty. Można zrozumieć innych!

Sposób myślenia

Z myślenia w kategorii poziomu klas czy obiektów przechodzimy do myślenia na poziomie bardziej abstrakcyjnym, czyli wzorców. Używanie wzorców pozwala na wydłużenie pracy na poziomie projektowania - nie przechodzimy zbyt szybko do problemu implementacji.

Zmiany

Wzorce pozwalają na zmianę jednej części kodu niezależnie od innych.

Wzorzec projektowy - eksperyment

Odkrywanie wzorców

Wzorce są odkrywane jak prawa natury. To są rozwiązania, które *odkryto doświadczalnie* przy pracy nad dużymi projektami. Programiści upublicznili swoje rozwiązania dla typowych problemów.

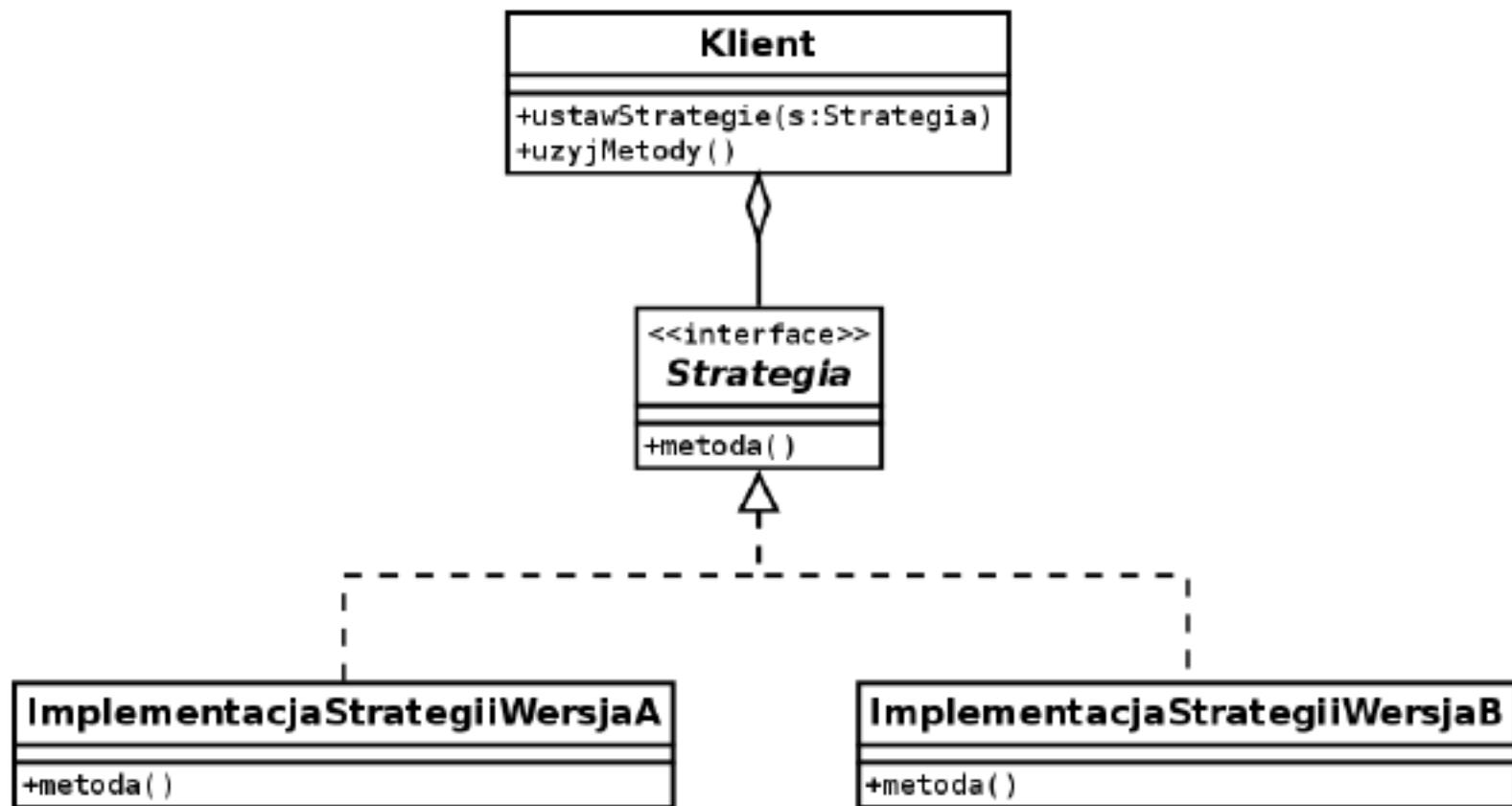
Wzorzec projektowy STRATEGIA

Wzorzec projektowy STRATEGIA wg. wikipedii

to jeden z czynnościowych wzorców projektowych (opisuje zachowanie i odpowiedzialność współpracujących ze sobą obiektów), który definiuje rodzinę wymiennych algorytmów i kapsułkuje je w postaci klas. Umożliwia wymienne stosowanie każdego z nich w trakcie działania aplikacji niezależnie od korzystających z nich klientów.

[http://pl.wikipedia.org/wiki/Strategia_\(wzorzec_projektowy\)](http://pl.wikipedia.org/wiki/Strategia_(wzorzec_projektowy))

Wzorzec projektowy STRATEGIA



Rysunek: Przykładowy diagram klas

Przykład użycia tego wzorca to implementacja odśmiecacza pamięci w wirtualnej maszynie Java HotSpot.

Wzorzec projektowy STRATEGIA

Zalety

- Kod jest bardziej zwięzły i przejrzysty. Nie ma potrzeby stosowania instrukcji warunkowych.
- Implementacje tej samej strategii mogą nie być identyczne - każda może być optymalizowana ze względu na inny aspekt pracy programu.
- Każda ze strategii może być testowana niezależnie, łatwiej jest odszukać błąd.
- W każdej chwili możemy dopisać kolejną strategię.

Wady

- To co robiła jedna klasa teraz wymaga wielu.
- Narzut na komunikację pomiędzy użytkownikiem a strategią.

Dziedziczenie a kompozycja - kolejne uwagi

Dziedziczenie

Klasa dziedzicząca ma zwiększoną widoczność wewnętrznych elementów klasy dziedziczonej. Tworzenie podklas określone jest z tego powodu jako **otwarte powtórne wykorzystanie** (white-box reuse).

Kompozycja

Użycie kompozycji klas w celu uzyskania bardziej złożonego działania prowadzi do użycia klas składowych przez ich interfejsy. Żadne wewnętrzne mechanizmy klas komponentów nie są widoczne, więc taki sposób działania nazywany jest jako **zamknięte powtórne wykorzystanie** (black-box reuse).

Poważne wady dziedziczenia

Dziedziczenie to operacja statyczna definiowana na etapie komplikacji.

Dziedziczenie narusza zasady kapsułkowania, bo podklasa może uzyskać dostęp do implementacji ukrytej przed innymi.

Wzorce projektowe - kolejne uwagi

Jak działają profesjonalisci?

Nie rozwiązujeją każdego problemu od podstaw. Używają wiedzy o rozwiązaniach projektowych, które wcześniej okazały się dobre. Budują *nowe* rozwiązania na wcześniejszych doświadczeniach.

"Każdy wzorzec opisuje problem powtarzający się w danym środowisku i istotę rozwiązania tego problemu w taki sposób, że można wykonać określone rozwiązanie milion razy i nigdy nie zrobić tego tak samo".

Christopher Alexander

Wzorce projektowe - kryteria podziału

Kryteria podziału wzorców: rodzaj wzorca

Rodzaj wzorca określa sposób działania. Mamy wzorce konstrukcyjne (zajmują się procesem tworzenia obiektów), wzorce strukturalne (związane są ze sposobem składania obiektów lub klas) i wzorce operacyjne [pomagają przy problemach związanych z podziałem pracy pomiędzy klasy (czy obiekty) i ich współdziałaniem].

Kryteria podziału wzorców: zasięg wzorca

Zasięg wzorca to informacja czy wzorzec dotyczy przede wszystkim obiektów (i tych jest większość) czy klas. Wzorce "klasowe" generują statyczne związki pomiędzy elementami kodu, bo ustalane są na etapie kompilacji kodu i wynikają z relacji pomiędzy nad- i pod-typami, które powstają przez dziedziczenie czy implementację interfejsu. Wzorce "obiektowe" prowadzą do bardziej dynamicznych związków pomiędzy obiektami, gdyż te mogą się zmieniać w trakcie pracy programu.

Wzorce konstrukcyjne.

Wzorce konstrukcyjne

Pozwalają na ukrycie procesu połączenia interfejsu i implementacji (klasy), czyli dokonują enkapsulacji informacji o tym z jakich konkretnych klas konkretnych się korzysta. Ponadto, ukrywany jest proces kompozycji (składania) obiektów. Sam proces tworzenia obiektu jest abstrakcyjny. To one pozwalają pisać system w kategorii interfejsów, zachować niezależność od: sposobu tworzenia, składania i reprezentowania obiektów.

Programowanie pod kątem interfejsu

Jakie są zalety używania obiektów tylko za pomocą interfejsu?

1. Klient nie musi znać typu obiektu, którego używa. Wystarczy, że obiekt jest zgodny z oczekiwanym przez niego interfejsem.
 2. Klient nie musi znać klas, które implementują interfejs. Wystarczy, że ktoś taki obiekt mu dostarczy.
- Obie te cechy zmniejszają zależności implementacyjne pomiędzy podsystemami.

Wskazówka praktyczna

Nie deklarować zmiennych podając jako ich typ nazwy klas konkretnych.

Projektowanie pod kątem zmian

Minimalizacja kosztów

Co jest kosztowne? Zmiany. Co jest bardzo kosztowne? Zmiany *nieprzewidziane*. W projektowaniu chodzi o to, aby przewidywać co może ulec zmianie w przyszłości i tak zaprojektować aplikację, aby mogła ewoluować.

Lista typowych problemów i lekarstw w postaci wzorców.

- Wskazanie nazwy klasy przy tworzeniu obiektu. Fabryka abstrakcyjna, Metoda wytwórcza, Prototyp.
- Zależność od specyficznych operacji. Łańcuch zobowiązań, Polecenie.
- Zależność od platformy sprzętowej lub programowej. Należy minimalizować zależność od platformy. Fabryka abstrakcyjna, Most.
- Zależność od implementacji lub reprezentacji obiektu. Fabryka abstrakcyjna, Most, Pamiątka, Pełnomocnik.

Projektowanie pod kątem zmian - cd.

Lista typowych problemów i lekarstw w postaci wzorców.

- Zależność od algorytmów. Budowniczy, Iterator, Strategia, Metoda szablonowa, Odwiedzający.
- Ścisłe powiązanie. Fabryka abstrakcyjna, Most, Łańcuch zobowiązań, Polecenie, Fasada, Mediator, Obserwator.
- Rozszerzanie możliwości oprogramowania poprzez tworzenie podklas. Most, Łańcuch zobowiązań, Kompozyt, Dekorator, Obserwator, Strategia.
- Niewygodne modyfikowanie klas. Adapter, Dekorator, Odwiedzający.

Wzorce projektowe a projektowanie obiektowe

Jak podzielić system?

W programie będzie współdziałać ze sobą wiele obiektów. Dzieląc system trzeba uwzględnić: enkapsulację, poziom szczegółowości, zależności, elastyczność, wydajność, możliwość wprowadzania zmian i ponownego użycia kodu. Problem w tym, że każde kryterium może sugerować inne i sprzeczne z pozostałymi rozwiązanie!

Jak tworzymy projekt?

Poznaliśmy analizę leksykalną. Można system projektować analizując jego zadania, można tworzyć niemal wierną symulację świata rzeczywistego - ale czy system będzie prawidłowo odzwierciedlał przyszły stan?

Kluczem do elastyczności jest abstrakcja!

Okazuje się, że w projekcie i tak pojawiają się (nisko- lub wysokopoziomowe) klasy, których w świecie rzeczywistym brak (np. zbiór danych). Dodatkowe, i to mniej oczywiste, abstrakcje i powiązane z nimi obiekty pozwalają wykryć wzorce. Np. poznana już Strategia.

Wzorce projektowe - katalog

Tabela: Skrócony katalog wzorców projektowych

Zasięg	Konstrukcyjne	Rodzaj Strukturalne	Operacyjne
Klasa	Metoda wytwarzca (str.103)	Adapter klasowy (str.152)	Metoda szablonowa (str.163)
Obiekt	Fabryka abstrakcyjna (str.112) Singleton (str.121) Budowniczy (str.99) Prototyp (str.116)	Dekorator (str.87) Adapter obiektowy (str.152) Fasada (str.156) Pełnomocnik (str.177) Pyłek (str.180) Kompozyt (str.184) Most (str.204)	Strategia (str.75) Obserwator (str.134) Polecenie (str.142) Stan (str.171) Łańcuch zobowiązań (str.188) Odwiedzający (str.192) Pamiątka (str.197) Mediator (str.200)

Wzorzec projektowy DEKORATOR

Wzorzec projektowy DEKORATOR wg. wikipedii

jeden ze wzorców projektowych należący do grupy wzorców strukturalnych. Pozwala na dodanie nowej funkcjonalności do istniejących klas dynamicznie podczas działania programu.

[http://pl.wikipedia.org/wiki/Dekorator_\(wzorzec_projektowy\)](http://pl.wikipedia.org/wiki/Dekorator_(wzorzec_projektowy))

DEKORATOR

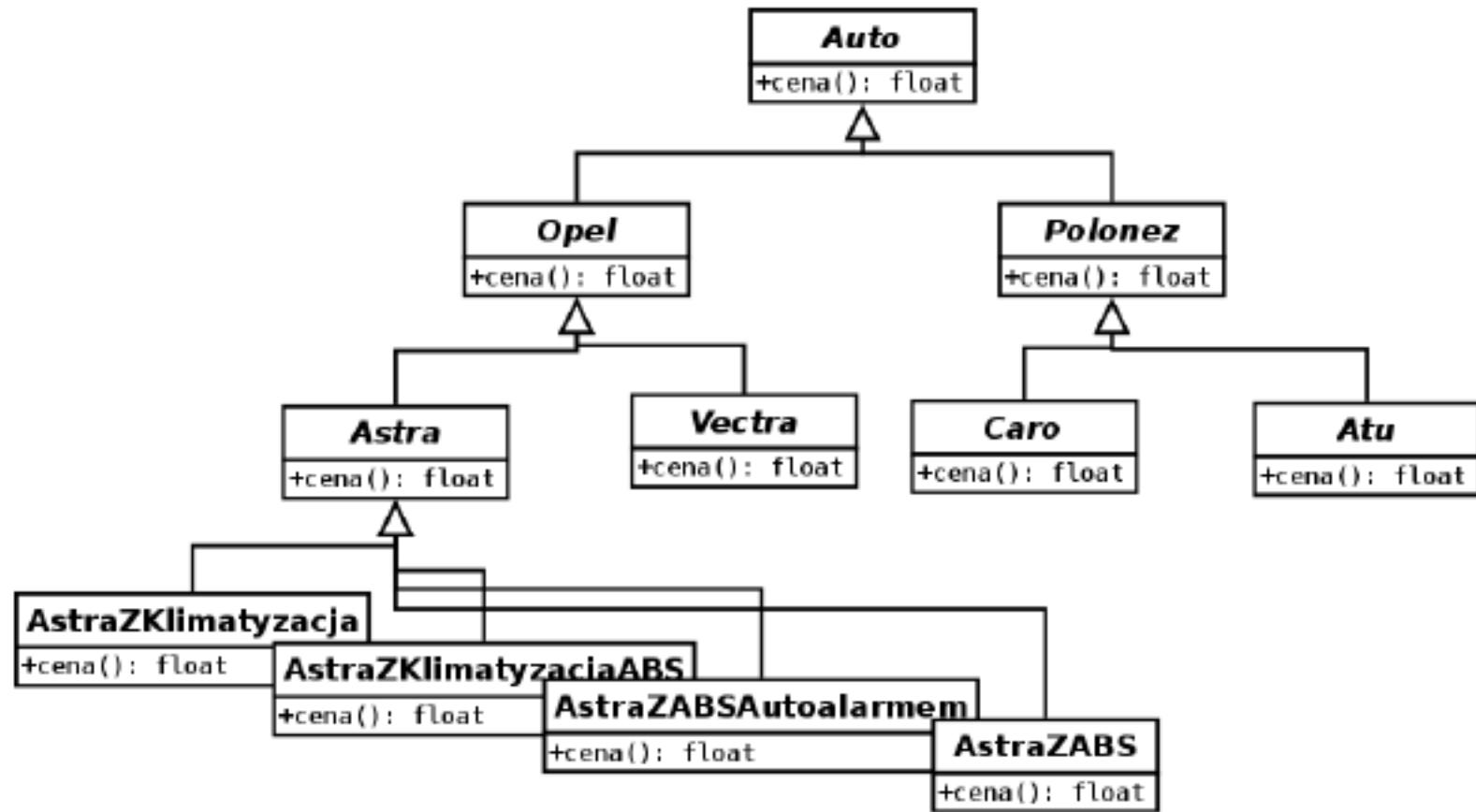
Klasy opakowują inne klasy. Metody klasy opakowującej wywołują metody klasy opakowywanej zmieniając jednak ich oryginalne zachowanie.

Alternatywa dla dziedziczenia

Dziedziczenie wymaga kompilacji kodu, DEKORATOR pozwala na rozszerzanie klas w trakcie ich działania.

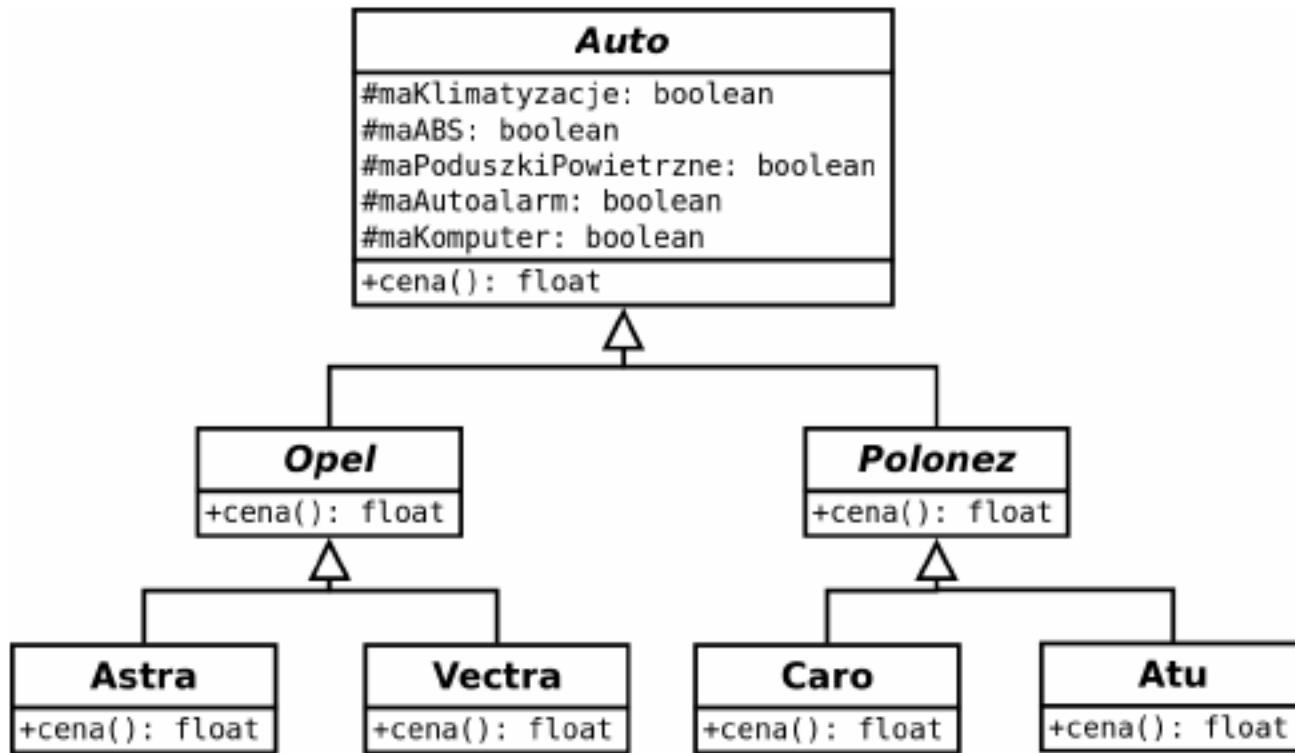
Za dużo klas

Problem: wyliczyć cenę auta wraz z opcjami (ABS, klimatyzacja, poduszki powietrzne, autoalarm, komputer, nawigacja itd)



Rysunek: Diagram klas dla programu wycenяj±cego auta

Za dużo klas - próba ratunku



Rysunek: Diagram klas dla programu wyceniającego auta

Problem 1: klient chce auto z alufelgami, antyradarem i pancernymi szybami.
Problem 2: czy wszystkie auta rzeczywiście muszą mieć wszystkie opcje, i co się stanie, gdy cena ulegnie zmianie?

Przypomnienie zasady otwarte-zamknięte

Dążymy do tego aby istniała możliwość modyfikacji klas czyli dodawania czy zmiany ich zachowania, ale jednocześnie nie chcemy zmieniać kodu, który już istnieje.

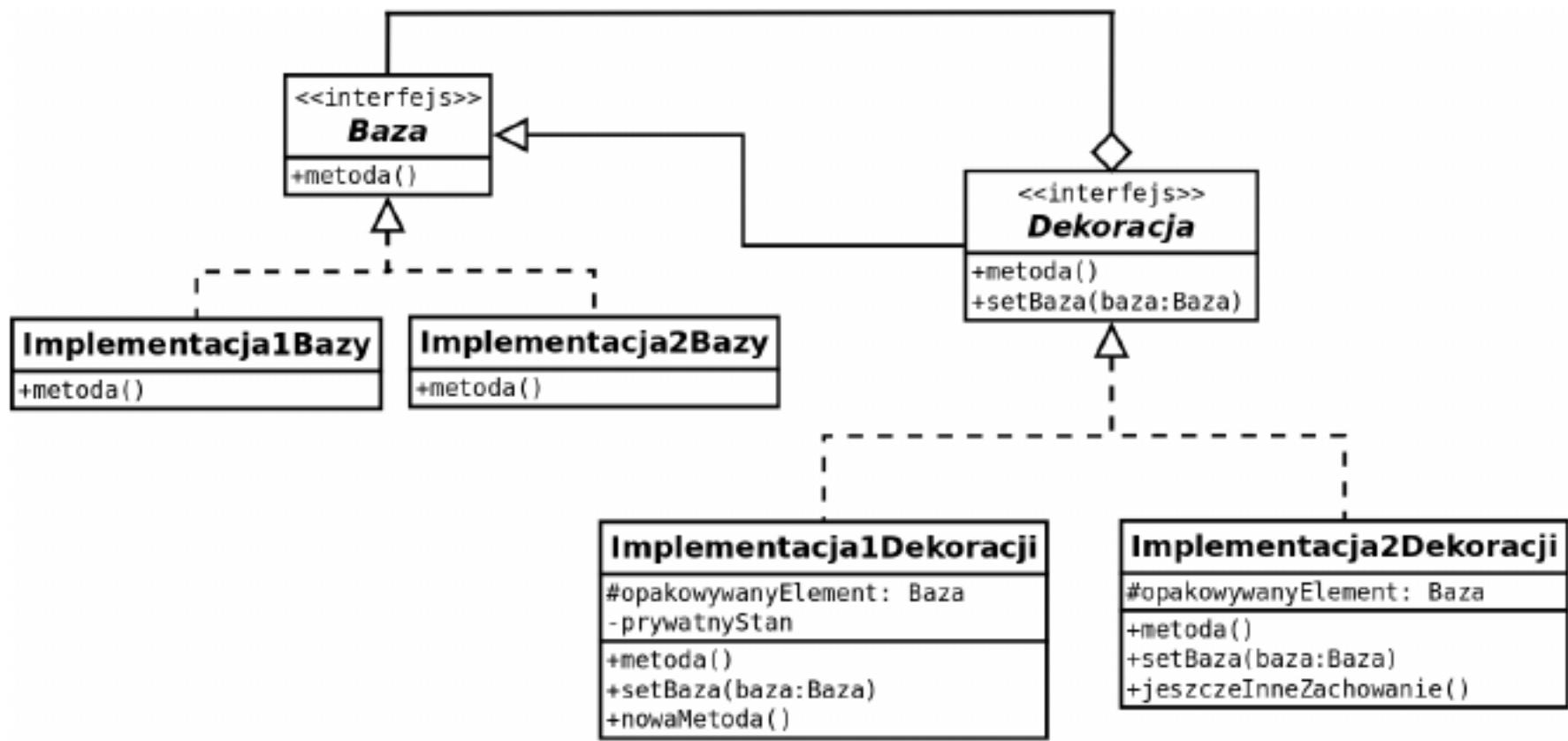
"Nowa wersja usunęła stare błędy i dodała nowe" ...

Zasada otwarte-zamknięte (Open–Close Principle)

Klasy powinny być otwarte na rozbudowę i zamknięte na modyfikację

Uwaga: jak zawsze trzeba myśleć. Używanie tej zasady do wszystkich fragmentów aplikacji może doprowadzić do złożonego i trudnego do zrozumienia kodu.

Wzorzec projektowy DEKORATOR



Rysunek: Przykładowy diagram klas

DEKORATOR – przykład

Program Dekorator.java - auta

```
1 abstract class Auto {  
2     protected float cena;  
3     public float cena() {  
4         return cena;  
5     }  
6 }  
7  
8 abstract class Polonez extends Auto {}  
9  
10 class Atu extends Polonez {  
11     public Atu() {  
12         cena = 20000;  
13     }  
14 }  
15  
16 class Caro extends Polonez {  
17     public Caro() {  
18         cena = 17000;  
19     }  
20 }
```

DEKORATOR – przykład

Program Dekorator.java - dodatki

```
1 abstract class Dodatki extends Auto {  
2  
3     protected Auto komponent;  
4     protected int czasInstalacji;  
5  
6     public int czasInstalacji() {  
7         if ( komponent instanceof Dodatki ) {  
8             return czasInstalacji +  
9                 ((Dodatki)komponent).czasInstalacji();  
10        } else {  
11            return czasInstalacji;  
12        }  
13    }  
14  
15    public float cena() {  
16        return komponent.cena() + cena;  
17    }  
18}
```

DEKORATOR – przykład

Program Dekorator.java - dodatki cd.

```
1 class ZKlimatyzacja extends Dodatki {  
2     public ZKlimatyzacja( Auto a ) {  
3         komponent = a;  
4         czasInstalacji = 7;  
5         cena = 3000;  
6     }  
7  
8     public float cena() {  
9         if ( komponent instanceof Caro )  
10            return komponent.cena() + 3 * cena;  
11         return super.cena();  
12     }  
13 }  
14  
15 class ZAutoalarmem extends Dodatki {  
16     public ZAutoalarmem( Auto a ) {  
17         komponent = a;  
18         czasInstalacji = 1;  
19         cena = 1000;  
20     }  
21 }
```

DEKORATOR – przykład

Program Dekorator.java - Start

```
1 class Start {  
2  
3     private static void rozliczenie( Auto a ) {  
4         System.out.println( "To auto kosztuje : " + a.cena() );  
5         System.out.println( " + dodajemy klimatyzacje" );  
6         a = new ZKlimatyzacja( a );  
7         System.out.println( "To samo auto z klimatyzacją kosztuje : "  
8                             + a.cena() );  
9         System.out.println( "           czas instalacji komponentów : "  
10                            + ((Dodatki)a).czasInstalacji() );  
11  
12        System.out.println( " + Dodajemy autoalarm: " );  
13        a = new ZAutoalarmem( a );  
14  
15        System.out.println( "To samo auto z klimatyzacją i autoalarmem [...]"  
16                                + a.cena() );  
17        System.out.println( "           czas instalacji [...]"  
18                                + ((Dodatki)a).czasInstalacji() );  
19    }  
20  
21    public static void main( String[] argv ) {  
22        System.out.println( "ATU" );  
23        rozliczenie( new Atu() );  
24        System.out.println( "CARO" );  
25        rozliczenie( new Caro() );  
26    }  
27 }
```

DEKORATOR – przykład

Wynik pracy programu:

ATU

To auto kosztuje : 20000.0

+ dodajemy klimatyzacje

To samo auto z klimatyzacją kosztuje : 23000.0

 czas instalacji komponentów : 7

+ Dodajemy autoalarm:

To samo auto z klimatyzacją i autoalarmem kosztuje : 24000.0

 czas instalacji komponentów : 8

CARO

To auto kosztuje : 17000.0

+ dodajemy klimatyzacje

To samo auto z klimatyzacją kosztuje : 26000.0

 czas instalacji komponentów : 7

+ Dodajemy autoalarm:

To samo auto z klimatyzacją i autoalarmem kosztuje : 27000.0

 czas instalacji komponentów : 8

DEKORATOR – przykład

W Java obsługa plików zrealizowana jest za pomocą wzorca dekorator. Dekoracja polega na dodawaniu funkcjonalności takich jak:

- Kompresja w locie
- Konwersja bajty - znaki
- Buforowanie
- Filtrowanie danych
- Zliczanie wierszy

```
1 BufferedReader in = new BufferedReader(
2                     new InputStreamReader(
3                     new GZIPInputStream(
4                     new FileInputStream( "plik.gz" ) ) ) );
5 String c;
6 while ((c = in.readLine()) != null) {
7     System.out.println( c );
8 }
```

Wzorzec projektowy DEKORATOR - wady i zalety

Zalety

Zwiększoną elastyczność w porównaniu ze statycznym dziedziczeniem. Dodatkowe funkcjonalności można w dowolnej chwili dodawać i usuwać. Ta sama dekoracja może zostać dodana wielokrotnie np. do okna można dodać ramkę i jeszcze raz ramkę.

Ogranicza liczbę metod w klasach bazowych (na wysokich stopniach hierarchii). Nie ma potrzeby deklarowania dużej liczby metod aby obsłużyć aktualnie potrzebne i przewidywane na przyszłość funkcjonalności. Dekorator pozwala dodawać funkcjonalności dopiero gdy będą potrzebne.

Wady

Problem identyczności. Obiekt X i opakowany obiekt X to różne (nie identyczne) obiekty.

Dużo małych i podobnych obiektów. Różnice są tylko w sposobie połączenia.

Wzorzec projektowy BUDOWNICZY

Wzorzec projektowy BUDOWNICZY

(ang. builder) oddziela tworzenie złożonego obiektu (produkту) od jego reprezentacji. Ten sam proces konstrukcji może doprowadzić do powstania różnych reprezentacji. Proces tworzenia, konfiguracji i inicjacji obiektu podzielony jest na mniejsze części-etypy. Każdy z etapów może być implementowany na różne sposoby. Sposób tworzenia obiektów zamknięty jest w tzw. konkretnych budowniczych. Produkt nie jest gotowy natychmiast - trzeba poczekać na zakończenie wszystkich działań Budowniczego.

Konkretny budowniczy

- tworzy i łączy składniki produktu
- odpowiada za stan wykonywanego produktu
- udostępnia metodę pobrania gotowego produktu

[https://pl.wikipedia.org/wiki/Budowniczy_\(wzorzec_projektowy\)](https://pl.wikipedia.org/wiki/Budowniczy_(wzorzec_projektowy))

Wzorzec projektowy BUDOWNICZY - przykład

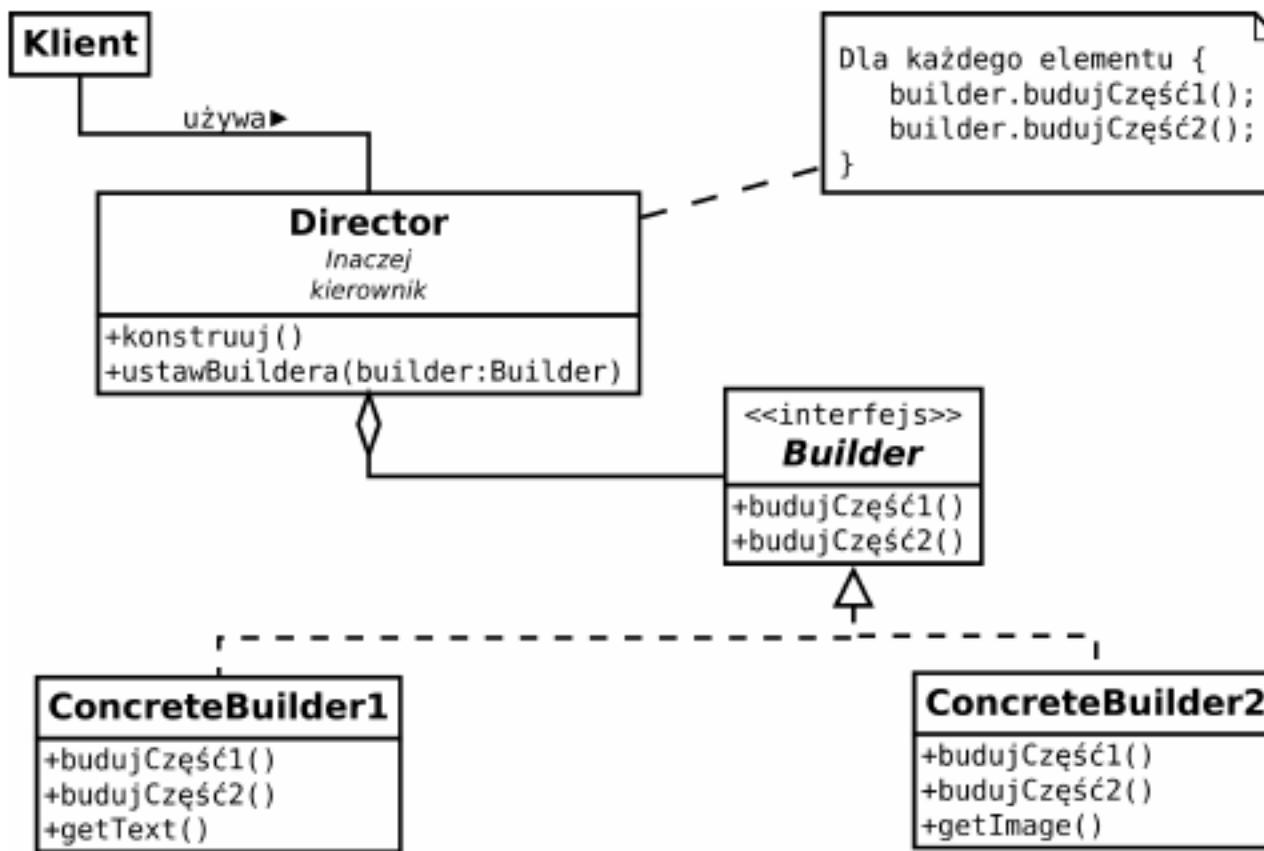
Typowy przykład: odczyt danych z pliku ASCII.

Kod odpowiedzialny za odczyt pliku może być niezależny od kodu, który dane interpretuje i buduje z nich produkt. Produkt budowany jest tak samo bez względu na to co tak naprawdę powstaje.

Produktem może być np.:

- zwykły tekst
- tekst sformatowany
- tabelka
- statystyka występowania słów
- obrazek (w tekście mogą być komendy języka Logo)

Wzorzec projektowy BUDOWNICZY - elementy



Rysunek: Przykładowy diagram klas

Wzorzec projektowy BUDOWNICZY - współdziałanie

- Klient tworzy obiekt Director-a i ConcreteBuilder-a
- Klient przekazuje do obiektu Directora obiekt ConcreteBuildera
- Klient wywołuje metodę Director.konstruj
- Director buduje złożony obiekt, choć nie wie jaki.
Budowa odbywa się poprzez metody budujCzęść1 i budujCzęść2
- Klient pobiera produkt z obiektu ConcreteBuilder-a

Uwaga: pobieranie produktu nie jest "ustandardyzowane" - wynika, to z tego, że produkty są tak różne, że wprowadzenie wspólnej **i użytecznej** klasy dla produktów nie jest zazwyczaj możliwe, a to prowadzi do braku możliwości ustalenia typu zwracanego rezultatu-produktu.

Metody Buildera mogą posiadać implementację domyślną. Dzięki temu tworząc klasę ConcreteBuilder nie trzeba implementować/przesłaniać wszystkich.

Wzorzec projektowy METODA WYTWÓRCZA

Wzorzec projektowy METODA WYTWÓRCZA wg. wikipedii

(ang. factory method) to jeden z kreacyjnych wzorców projektowych (klasowy), którego celem jest dostarczenie interfejsu do tworzenia obiektów. Tworzeniem instancji zajmują się podklasy.

Kreacyjny wzorzec projektowy

opisuje proces tworzenia nowych obiektów; jego zadaniem jest tworzenie, inicjacja oraz konfiguracja obiektów, klas oraz innych typów danych.

Wzorzec klasowy

oznacza, że opisuje on związki pomiędzy klasami.

[http://pl.wikipedia.org/wiki/Metoda_wytwórcza_\(wzorzec_projektowy\)](http://pl.wikipedia.org/wiki/Metoda_wytwórcza_(wzorzec_projektowy))

Wzorzec projektowy METODA WYTWÓRCZA

Problem

Powinniśmy programować z użyciem typów bazowych. Użycie operatora `new` wymaga podania konkretnej klasy. Często klas pasujących do typu bazowego jest wiele.

Klient chce kupić konkretne auto i pomalować je w wybrany wzorek

```
1 Auto a;
2 if ( firma == "Opel" ) {
3     if ( model == "Omega" ) a = new Omega();
4     if ( model == "Vectra" ) a = new Vectra();
5     if ( model == "Sintra" ) a = new Sintra();
6 }
7
8 malowanie( a, wzorMalowania );
```

Pojawienie się nowego modelu oznacza również konieczność modyfikacji tej części aplikacji. Kod nie jest zamknięty na modyfikację. Ponadto ten sam model może być produkowany w wersji dla ruchu lewo- i prawo-stronnego...

Wzorzec projektowy METODA WYTÓRCZA

Krok pierwszy

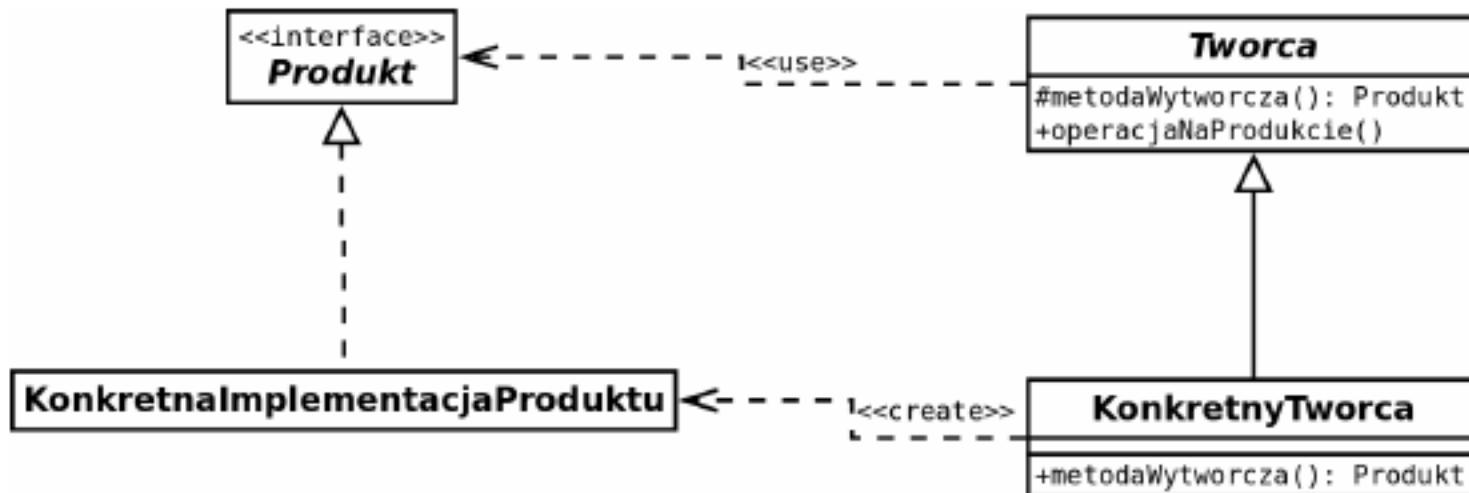
Kod, który może ulec zmianie hermetyzujemy w osobnej metodzie.

Klient chce kupić konkretne auto i pomalować je w wybrany wzorek

```
1 protected Auto zbudujAuto( firma, model ) {  
2     if ( firma == "Opel" ) {  
3         if ( model == "Omega" ) return new Omega();  
4         if ( model == "Vectra" ) return new Vectra();  
5         if ( model == "Sintra" ) return new Sintra();  
6     }  
7 }  
8  
9 Auto a = zbudujAuto( wybranaFirma, wybranyModel );  
10 malowanie( a, wzorMalowania );
```

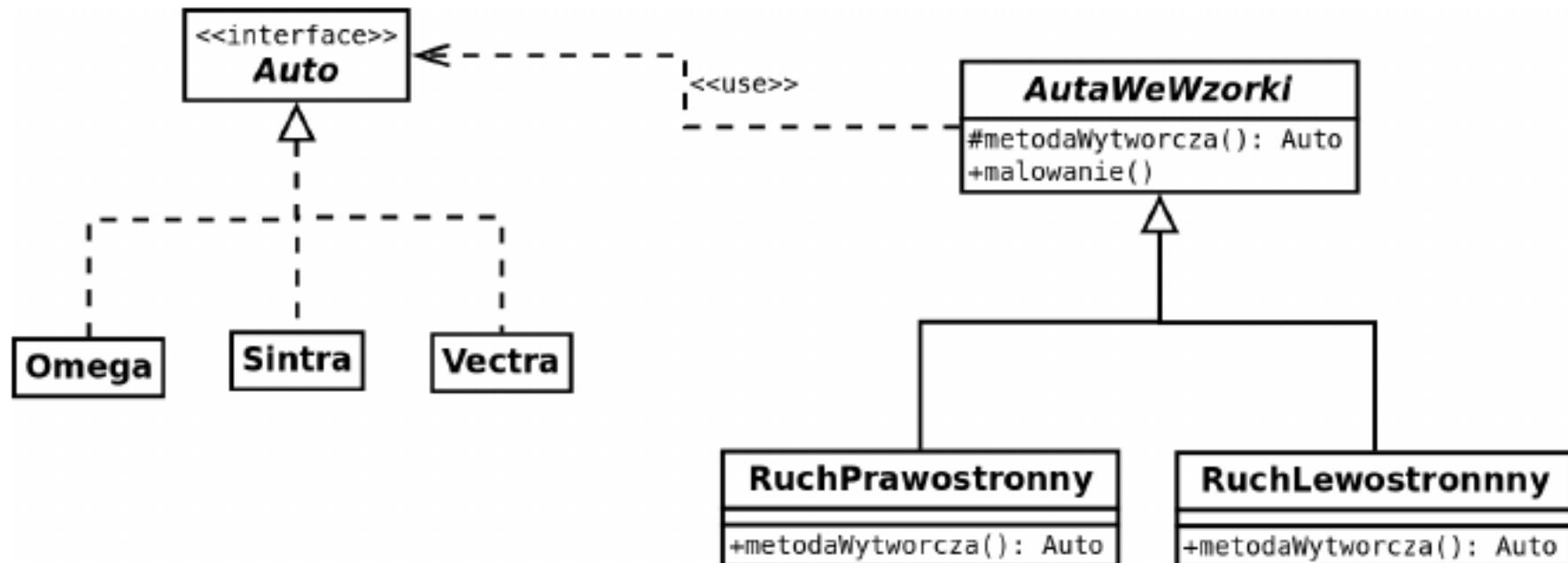
Dodawanie kolejnych wymogów obsługiwanych przez jedna metodę oznacza coraz dłuższy blok warunków. Kod staje się nieczytelny i łatwo się pomylić.
Klasa sama tworzy obiekty. Do tego potrzebuje konkretnych klas. Oznacza to, że jest ona silnie zależna od konkretnych implementacji.

Wzorzec projektowy METODA WYTWÓRCZA



Rysunek: Przykładowy diagram klas

Wzorzec projektowy METODA WYTWÓRCZA - przykład



Rysunek: Przykładowy diagram klas

Wzorzec projektowy METODA WYTWÓRCZA - uwagi

Wspieranie abstrakcji

Metoda wytwórcza pozwala na używaniu w kodzie klasy Tworca tylko abstrakcji w postaci klasy **Auto**.

Wada wzorca

Aby obsłużyć nowy konkretny produkt musi pojawić się podklasa klasy Tworca.

Łączenie równoległych hierarchii klas

Mamy dwie hierarchie klas. Obiekty jednej z hierarchii używają określonych obiektów drugiej. Metoda fabryczna dodana do pierwszej z hierarchii pozwala jej klasom na tworzenie odpowiednich, pasujących obiektów z klas hierarchii drugiej.

Wzmocnienie zasady programowania z użyciem interfejsu

Zasada odwrócenia zależności (Dependency Inversion Principle - DIP)

Zależeć od abstrakcji. Nie zależeć od konkretnych klas.

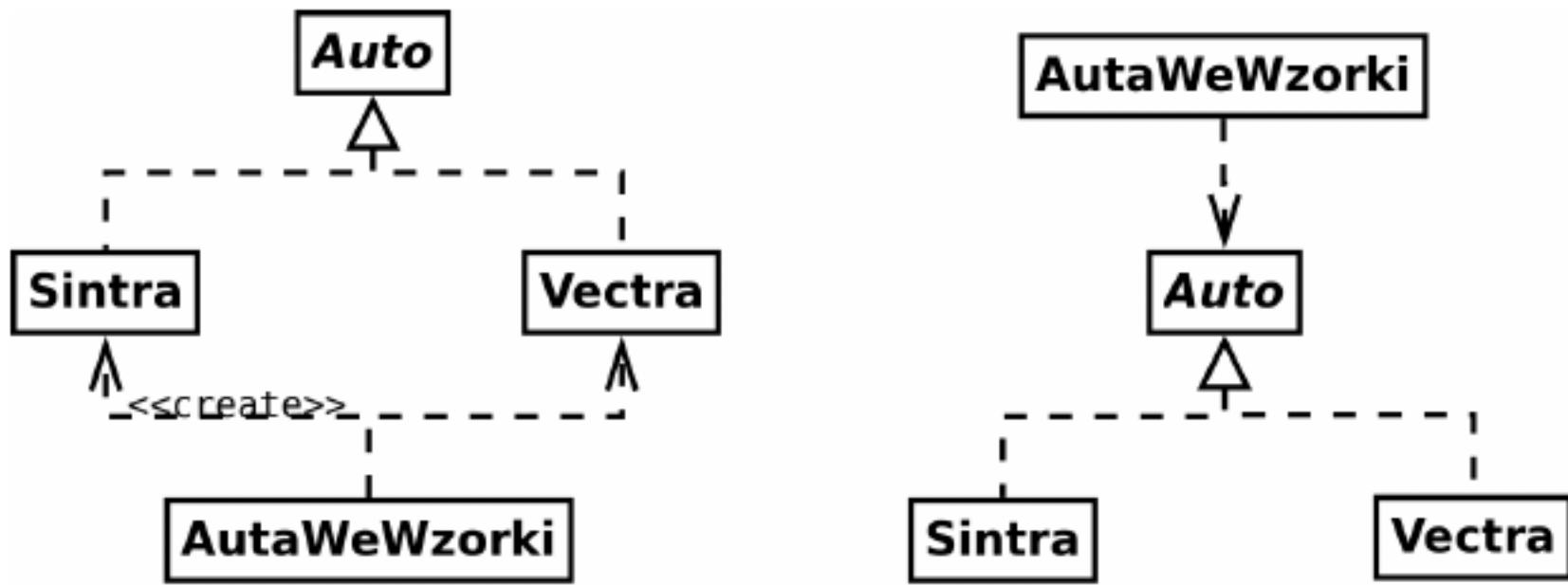
Czyli?

- Moduły wysokopoziomowe (definiowane w oparciu o inne moduły) nie powinny zależeć od modułów niskopoziomowych - oba typy modułów powinny zależeć od abstrakcji.
- Abstrakcje nie powinny zależeć od szczegółowych rozwiązań. To szczegółowe rozwiązania powinny zależeć od abstrakcji.

Przypomnienie: jeśli w jednej klasie używamy abstrakcji (**Auto**) i jednocześnie tworzymy w niej obiekty konkretnych klas (**Vectra**, **Omega**) to zysk z abstrakcją słabnie, bo i tak zależymy od klas niskopoziomowych.

Użycie wzorca METODA WYTÓRCZA powoduje, że klasa **AutaWeWzorki** zależy tylko od abstrakcji (**Auto**), jednocześnie klasy konkretne (**Omega**, **Sintra**, **Vectra**) także zależą od tej samej abstrakcji (**Auto**).

Zastosowanie DIP



AutaWeWzorki same tworzą konkretne Auta

Tu działa DIP

Rysunek: Przykładowy diagram klas

Jak nie naruszać DIP?

Wytyczne:

- Żadna ze zmiennych nie powinna być typu klasy konkretnej
- Żadna klasa nie powinna pochodzić od klasy konkretnej
- Zaimplementowane metody nie powinny być przesłaniane / przeddefiniowane. Jeśli metoda zadeklarowana jest w klasie abstrakcyjnej to po to, aby była ona współdzielona przez potomków.

To są tylko wytyczne. Trzeba traktować je z rozsądkiem — mają one pomagać w oddzielaniu tych części kodu, które mogą podlegać zmianom.

Np. obiekty klasy `String` używane są legalnie za pomocą referencji typu `String`, ale klasa `String` nie może mieć podklas.

Wzorzec projektowy FABRYKA ABSTRAKCYJNA

Wzorzec projektowy FABRYKA ABSTRAKCYJNA wg. wikipedii

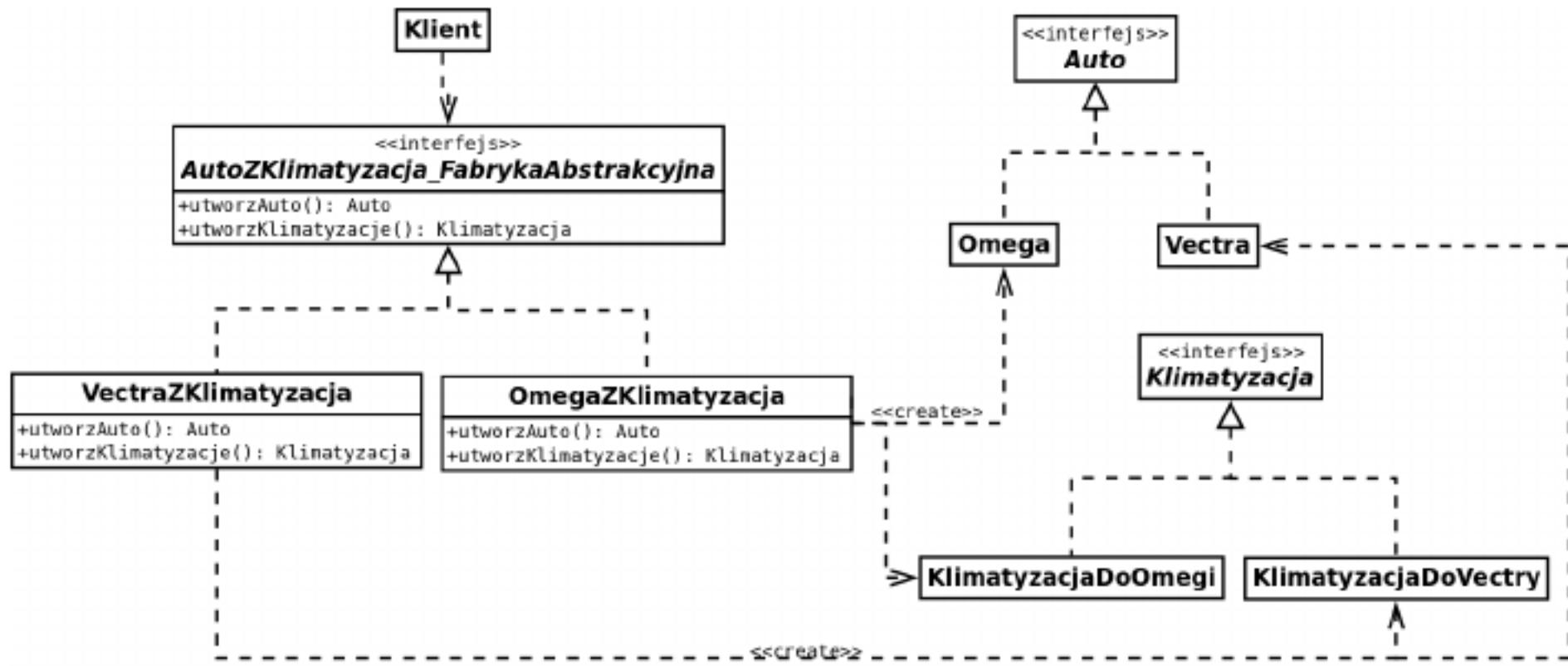
(ang. Abstract Factory) jest to jeden z kreacyjnych wzorców projektowych (obiektowy), którego celem jest dostarczenie interfejsu do tworzenia różnych obiektów jednego typu (tej samej rodziny) bez specyfikowania ich konkretnych klas. Umożliwia jednemu obiekowi tworzenie różnych, powiązanych ze sobą, reprezentacji pod-obiektów określając ich typy podczas działania programu.

[http://pl.wikipedia.org/wiki/Fabryka_abstrakcyjna_\(wzorzec_projektowy\)](http://pl.wikipedia.org/wiki/Fabryka_abstrakcyjna_(wzorzec_projektowy))

Wzorzec projektowy FABRYKA ABSTRAKCYJNA wg. E. Gamma "Wzorce Projektowe"

Udostępnia interfejs do tworzenia rodzin powiązanych ze sobą lub zależnych od siebie obiektów bez określania ich klas konkretnych.

Wzorzec projektowy FABRYKA ABSTRAKCYJNA - przykład



Rysunek: Przykład użycia wzorca FABRYKA ABSTRAKCYJNA do tworzenia par powiązanych ze sobą obiektów auta i pasującej do niego klimatyzacji

FABRYKA ABSTRAKCYJNA a METODA WYTWÓRCZA

FABRYKA ABSTRAKCYJNA a METODA WYTWÓRCZA

Oba wzorce są powiązane. Fabryka abstrakcyjna dostarcza interfejsu do tworzenia **zbioru** obiektów. Jedna metoda tworzy jeden obiekt. Metody fabryki abstrakcyjnej są implementowane w podklasach fabryki abstrakcyjnej i tu mogą pojawić się poznane wcześniej metody twórcze.

Wzorzec projektowy FABRYKA ABSTRAKCYJNA

Zalety

Implementacja może zostać ukryta przed klientem. Klient dysponuje wyłącznie interfejsem, przez który uzyskuje referencje do typów bazowych — nie ma możliwości sprawdzenia jakie konkretne klasy są używane przez fabrykę. Ponadto, wzorzec ten wymusza spójność obiektów.

Wady

Problem z rozszerzaniem — dodanie nowych podklas wymusza modyfikację kodu fabryki. Np. chcemy dodać podkласę **KlimatyzacjaStrefowa** - trzeba zmienić interfejs i do każdego z typów aut dodać tworzenie odpowiednich obiektów.

Wzorzec projektowy PROTOTYP

Wzorzec projektowy PROTOTYP

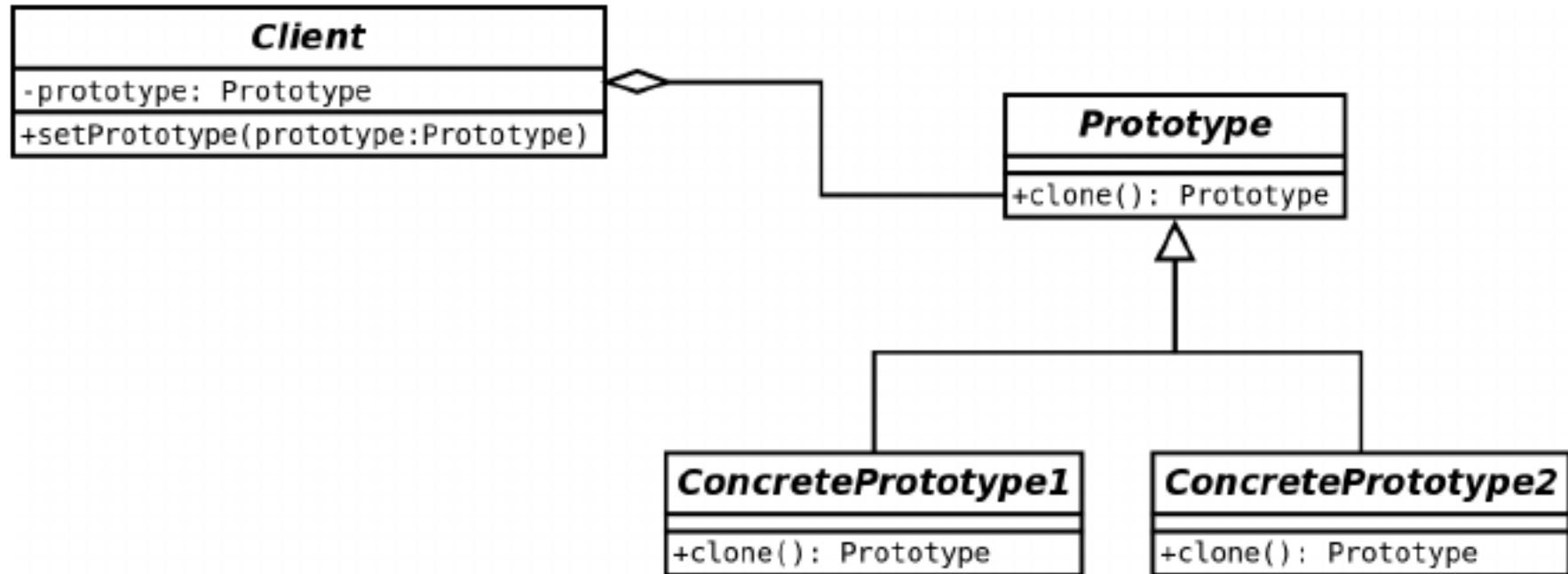
(ang. Prototype) umożliwia tworzenie obiektu/obiektów poprzez kopiowanie istniejącego obiektu, który nazywany jest prototypem.

Wzorzec jest stosowany, gdy:

- system ma być niezależny od sposobu tworzenia, składania i reprezentowania produktów oraz, gdy:
 - klasy tworzonych egzemplarzy są określane w trakcie pracy programu *LUB*
 - chcemy uniknąć tworzenia hierarchii klas-fabryk dla hierarchii klas-produktów *LUB*
 - obiekty posiadają małą liczbę stanów, zamiast tworzenia obiektów i ich inicjacji łatwiejsze może być klonowanie posiadanych prototypów.

[https://pl.wikipedia.org/wiki/Prototyp_\(wzorzec_projektowy\)](https://pl.wikipedia.org/wiki/Prototyp_(wzorzec_projektowy))

Wzorzec projektowy PROTOTYP - schemat



Rysunek: Struktura wzorca Prototyp

Wzorzec projektowy PROTOTYP - zalety stosowania

- Duża elastyczność pracy - aby dodać nowy produkt wystarczy pokazać prototyp
- Prototyp wystarczy przekazać przez wartość - nie trzeba programować
- Prototypy nie muszą być trywialnie - jeśli klonowanie jest dogłębne, to można utworzyć kopię złożonego układu - np. klonowanie zgrupowanych obiektów
- Zmniejsza liczbę podklas. Nie trzeba pisać klas do tworzenia różnych produktów. Wystarczy sam produkt.
- Można utworzyć katalog prototypów, którym zarządza pewien manager prototypów. Może on wczytywać klasy, których pierwotnie w aplikacji nie było.

Część tych zalet blednie w Java, gdyż w Java mamy refleksję. Klasy są reprezentowane przez obiekty i z nich można bezpośrednio tworzyć instancje. Klasy nie muszą być znane w trakcie komplikacji kodu.

Wzorzec ten nabiera na wartości w językach ze statyczną kontrolą typu, gdzie w trakcie pracy programu praktycznie nie ma już informacji o typie.

Wzorzec projektowy PROTOTYP - wady

- Trzeba zaimplementować metodę odpowiedzialną za klonowanie. Może to być bardzo trudne (np. obiekt A ma referencję do B, a B do obiektu A).
- Klonowanie nie zawsze będzie możliwe np. brak dostępu do kodu używanej nadklasy.
- Konieczne jest podejmowanie decyzji o tym, które dane są prywatne, a które współdzielone pomiędzy obiektami.
- Nie zawsze stan prototypu jest stanem odpowiednim dla jego kлона. Z kolei przekazywanie parametrów (wielu!) do metody klonującej nie jest zgodne z ideą jednolitego interfejsu klonowania.

Wzorzec projektowy SINGLETON

Wzorzec projektowy SINGLETON wg. wikipedii

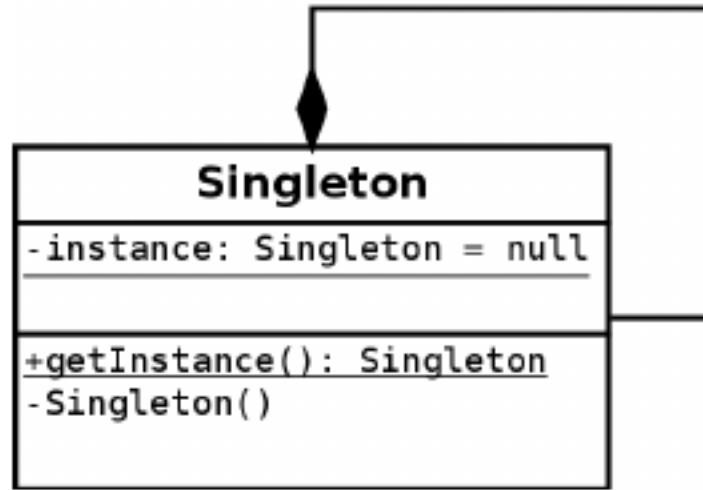
jeden z kreacyjnych, obiektowych wzorców projektowych, którego celem jest ograniczenie możliwości tworzenia obiektów danej klasy do jednej instancji oraz zapewnienie globalnego dostępu do stworzonego obiektu. Niekiedy wzorzec uogólnia się do przypadku wprowadzenia pewnej maksymalnej liczby obiektów, jakie mogą istnieć w systemie.

Typowe zastosowanie

obiektowy zamiennik zmiennej globalnej - jedna zmienna dostępna dla wielu fragmentów kodu - zmiana w jednym z nich jest natychmiast widoczna w pozostałych

[http://pl.wikipedia.org/wiki/Singleton_\(wzorzec_projektowy\)](http://pl.wikipedia.org/wiki/Singleton_(wzorzec_projektowy))

Wzorzec projektowy SINGLETON



Rysunek: Tak za pomocą UML można przedstawić wzorzec Singleton

Warte zauważenia:

- Konstruktor jest prywatny
- Metoda `getInstance()` jest statyczna

Wzorzec projektowy SINGLETON - zarys implementacji

Singleton, który (prawie) nic nie robi

```
1 class Singleton {  
2     private static Singleton instance;  
3  
4     private Singleton() {}  
5  
6     public static Singleton getInstance() {  
7         if ( instance == null ) {  
8             instance = new Singleton();  
9         }  
10    return instance;  
11 }  
12 }
```

Współbieżność

Powyższy kod nie jest zabezpieczony na wypadek użycia w programie współbieżnym!!! Rozwiązaniem tego problemu zajmiemy się w ramach kursu "Programowanie Rozproszone i Równoległe".

Zalety i wady wzorca SINGLETON

Zalety:

- Istnieje ograniczona i zarządzana przez wzorzec liczba obiektów
- Tworzenie instancji jest niewidoczne dla użytkownika
- Możliwa jest implementacja "leniwa" - nie pobiera zasobów do chwili pierwszego użycia

Wady:

- Ustalona na sztywno liczba instancji
- Utrudnia testowanie aplikacji - jej części są ze sobą mocno powiązane poprzez istnienie "globalnego stanu"
- Łamie zasadę jednej odpowiedzialności, bo Singleton robi kilka różnych rzeczy
- Łamie zasadę otwarte-zamknięte, bo Singletona nie można rozszerzać

Tworzenie obiektów - uwagi

Statyczne metody fabryczne

Statyczna metoda fabryczna

Chodzi o użycie metody, która jest statyczna i zwraca referencję/wskaźnik do obiektu. Metoda taka powinna być nie prywatna. Nie jest to odpowiednik wzorca "Metoda fabryczna".

Jakie ma zalety:

- Ma nazwę! Nazwa może coś znaczyć. Pozwala na powtórzenie listy typów parametrów.
- Wywołanie metody nie wymusza pojawienia się nowej instancji. Zwrócenie istniejącego obiektu może poprawić wydajność aplikacji. Metoda statyczna pozwala zapanować nad utworzonymi obiektów i to nie tylko w kwestii ich liczby (tak działająca klasa nazywana jest "kontrolowaną przez instancje").
- Metoda nie musi zwracać obiektu typu, który został podany jako typ rezultatu. Może zwrócić podtyp.
- Faktycznie zwracany typ może być uzależniony od parametrów.
- Może zwracać obiekty klas, które nie istniały w trakcie pracy nad klasą z metodą statyczną. Np. JDBC.

Statyczne metody fabryczne - wady

Wady zalety:

- Prywatny konstruktor ogranicza możliwości dziedziczenia.
- W dokumentacji są traktowane na równi z innymi metodami - można je przeoczyć. Przy okazji konwencja nazewnicza:
 - `from` - konwersja typu
 - `of lub valueOf` - agregacja
 - `instance lub instanceOf` - zwraca obiekt (jeśli istnieje)
 - `create lub newInstance` - tworzy nowy obiekt
 - `getType lub newType lub type` - Type wskazuje typ zwracanego obiektu.
Nazwy używane, gdy statyczna metoda fabryczna znajduje się w osobnej klasie.

Konstruktor z wieloma parametrami (opcjonalnymi)

Oczekiwanie

Chcemy obiektów wyłącznie w stanie niespójnym. Spójności żądamy od chwili utworzenia obiektu. Problem próbujemy rozwiązać za pomocą konstruktora. Wszystkie dane potrzebne do zainicjowania obiektu muszą zostać do konstruktora dostarczone.

Problemy

Lista parametrów może być dłuża. Ich typy mogą być zbliżone. Nie wszystkie parametry są zawsze wymagane - część z nich to parametry opcjonalne.

Konstruktor teleskopowy

Tworzymy konstruktor obsługujący wszystkie wymagane parametry. Następny z jednym parametrem opcjonalnym. Kolejny z dwoma itd. Jeśli język programowania na to pozwala, to można wywoływać jedne konstruktory z innych. Kod robi się nieczytelny, a to prowadzi do błędów. Z uwagi na ograniczoną liczbę sygnatur użytkownik może być zmuszony do używania konstruktora, którego część argumentów wywołania będzie sztucznie wypełniona np. przez zero.

Wzorzec JavaBeans.

JavaBeans zamiast wieloparametrowego konstruktora

Obiekt tworzymy za pomocą konstruktora bezparametrowego. Wszystkie potrzebne dane przekazujemy do obiektu poprzez metody set.

Problemy

Kod robi się długi, choć czytelny.

W trakcie przygotowywania obiektu do pracy jest on w stanie niespójnym. Klasa nie może wymusić spójności! Możliwe użycie obiektu w stanie niespójnym z trudnymi do przewidzenia konsekwencjami.

Nie dla klas niezmiennych!

Wzorzec JavaBeans praktycznie uniemożliwia utworzenie klasy niezmiennej. Ratunkiem może być sztuczne zablokowanie możliwości użycia obiektu do czasu zakończenia inicjalizacji.

Odmiana wzorca Builder.

Obiekt budowniczego

Konstruktor klasy jest prywatny. Zamiast konstruktora klasy używany jest konstruktor klasy Bulder. Użytkownik przekazuje mu wszystkie **obowiązkowe** parametry i otrzymuje obiekt budowniczego. Obiekt budowniczego używany jest do ustawienia parametrów opcjonalnych itworzenia obiektu oczekiwanej klasy. Budowniczy może przed wytworzeniem obiektu sprawdzić poprawność przekazanych do niego parametrów.

Kiedy używać budowniczego?

Gdy do utworzenia obiektu potrzeba wielu parametrów lub gdy oczekujemy, że liczba parametrów może w przyszłości wzrosnąć. W szczególności, kiedy część parametrów jest opcjonalna.

Uwaga

Należy unikać łączenie w jednej klasie rozwiązań typu konstruktory, statyczne metody fabryczne i klasa budowniczego.

Celem jest utworzenie obiektu klasy MainClass.

```
1 class MainClass {  
2     private int required1;  
3     private int required2;  
4  
5     private int optional1;  
6     private int optional2;  
7  
8     public static class Builder {  
9         private int required1;  
10        private int required2;  
11  
12        private int optional1 = 1; // domyslna inicializacja  
13        private int optional2 = 2;  
14  
15        public Builder( int required1, int required2 ) {  
16            this.required1 = required1;  
17            this.required2 = required2;  
18        }  
19  
20        public Builder setOptional1( int value ) {  
21            optional1 = value;  
22            return this;  
23        }  
24  
25        public Builder setOptional2( int value ) {  
26            optional2 = value;  
27            return this;  
28        }  
}
```

Celem jest utworzenie obiektu klasy MainClass.

```
1     public MainClass build() {
2         return new MainClass( this );
3     }
4 } // koniec klasy Builder
5
6 // prywatny konstruktor klasy MainClass
7 private MainClass( Builder builder ) {
8     required1 = builder.required1;
9     required2 = builder.required2;
10    optional1 = builder.optional1;
11    optional2 = builder.optional2;
12 }
13
14 } // koniec kodu klasy MainClass
15
16 class User {
17     public static void main(String[] args) {
18         MainClass mc = new MainClass.Builder(10,20) // utworzenie Buildera
19                         .setOptional1(123) // ustalenie parametrow
20                         .setOptional2(321) // opcjonalnych
21                         .build(); // utworzenie MainClass
22     }
23 }
```

Singleton raz jeszcze.

Kiedy nie stosować wzorca Singleton

Wzorzec Singleton nie nadaje się do implementacji klas, które zależą od zasobów zmieniających ich działanie, a same takiego zasobu nie tworzą.

Wstrzykiwanie zależności

Rozwiążanie polega na przekazaniu do konstruktora (lub budowniczego) zwykłej klasy (nie Singletonu) potrzebnego zasobu lub fabryki, która potrafi go utworzyć. Jest to jedna z postaci wstrzykiwania zależności.

Singleton i raz jeszcze.

Testowanie

Trudno testować kod, w którym używany jest Singleton np. dlatego, że nie można użyć imitacji implementacji Singletonu. Rozwiązanie polega na użyciu interfejsu jako typu dla klasy implementowanej jako Singleton.

Singleton a Serializacja

UWAGA: Bez specjalnych zabiegów operacje (de)serializacji tworzą nowe obiekty!

Singleton utworzony z pomocą typu wyliczeniowego!

W języku Java ciekawą (i najlepszą!) metodą dla implementacji Singletonu jest użycie typu wyliczeniowego z **jednym** elementem. Sposób zapewnia obsługę serializacji i gwarantuje posiadanie jednego obiektu nawet w przypadku ataku z użyciem refleksji.

Wzorzec projektowy OBSERWATOR

Wzorzec projektowy OBSERWATOR wg. wikipedii

wzorzec należący do grupy wzorców czynnościowych. Używany jest do powiadamiania zazwyczaj **wielu** zainteresowanych obiektów o zmianie stanu pewnego innego obiektu. Wszystkie zależne obiekty powiadamiane są automatycznie.

[http://pl.wikipedia.org/wiki/Obserwator_\(wzorzec_projektowy\)](http://pl.wikipedia.org/wiki/Obserwator_(wzorzec_projektowy))

Co nam daje?

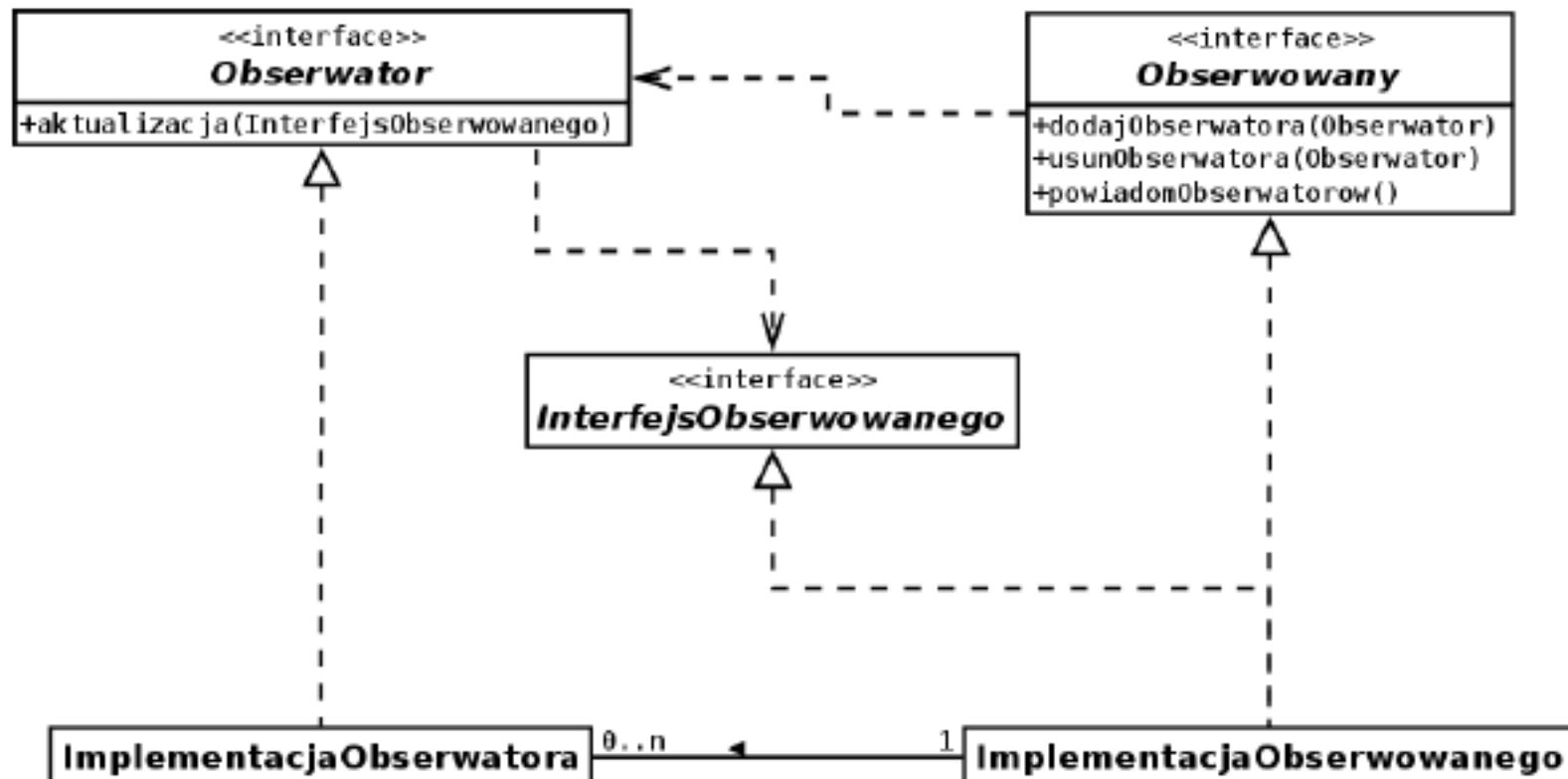
efektywność pracy (unikamy ciągłego odpytywania o stan), luźniejszą relację/powiązanie pomiędzy obiektami, relacja ta może ulegać zmianie w czasie wykonywania programu, informacja rozsyłana jest do wszystkich zainteresowanych - to oni muszą dokonać stosownej subskrypcji, relacja może być typu jeden-do-wielu (wiele obiektów informowanych o zmianie stanu jednego)

Wzorzec projektowy OBSERWATOR

Przykład: licytacja na internetowym portalu

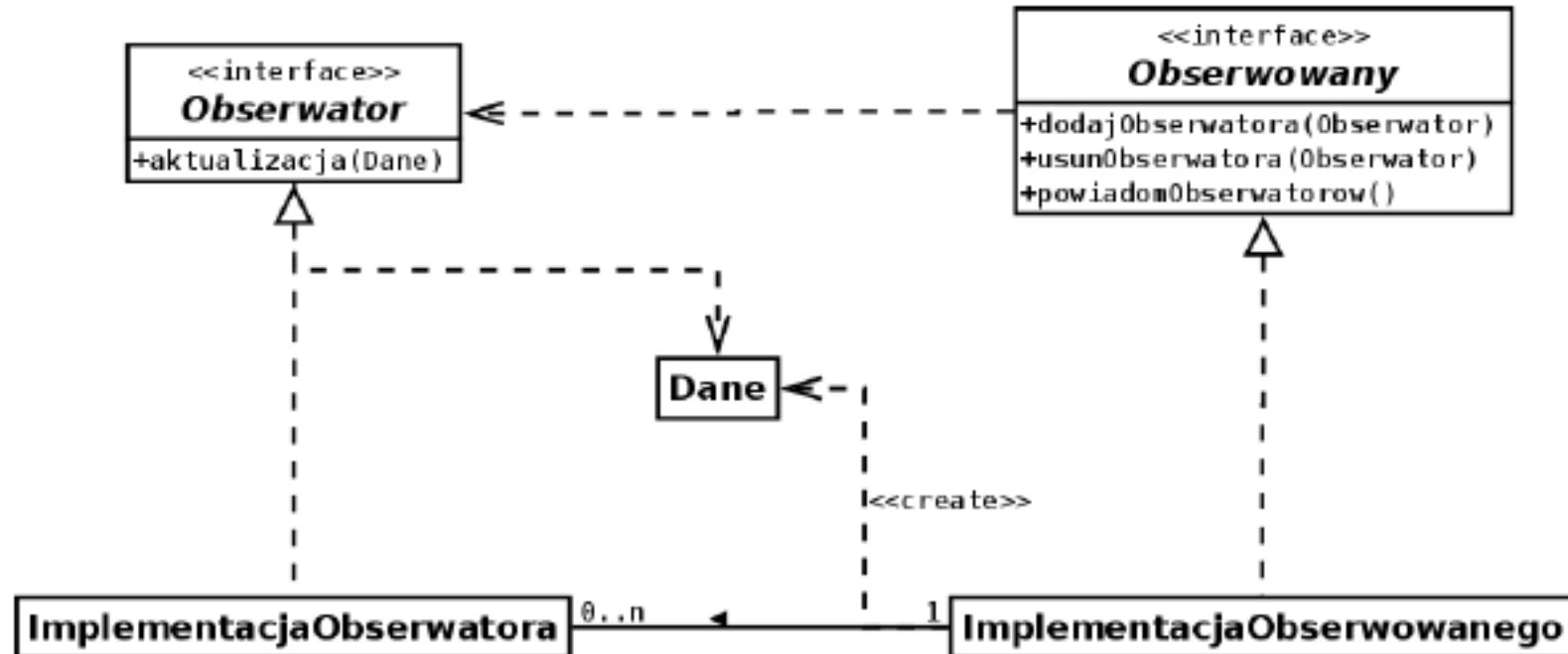
- ➊ Osoby zainteresowane przedmiotem aukcji rejestrują się jako obserwatorzy
- ➋ Gdy poprzednia kwota zostanie przebita do wszystkich zainteresowanych wysyłana jest wiadomość e-mail
- ➌ Jeśli kwota jest za duża obserwator może wyrejestrowywać się i od tej chwili wiadomości o stanie aukcji nie są mu przesyłane.

Wzorzec projektowy OBSERWATOR - strategia wyciągania



Rysunek: Przykładowy diagram klas

Wzorzec projektowy OBSERWATOR - strategia wypychania



Rysunek: Przykładowy diagram klas

Zalety luźnego powiązania pomiędzy klasami

Luźne powiązanie pomiędzy obiektami

oznacza, że mogą one współdziałać dysponując tylko niewielką wiedzą o sobie. Luźne powiązanie daje elastyczny kod, łatwo wprowadzać zmiany, gdy nie mają one wpływu na pracę innych części aplikacji

Nowa zasada projektowa

Należy dążyć do luźnego powiązania pomiędzy obiektami, które współdziałyają.

Zalety i wady OBSERWATORA

Zalety:

- Luźne powiązanie pomiędzy klasami
- Jedyną rzeczą, którą musi znać Obserwator o Obserwowanym jest interfejs.
- Nowy obserwator może zostać dodany/usunięty w każdej chwili - dynamika!
- Nie trzeba zmieniać Obserwowanego aby wspierać nowych Obserwatorów
- Nie trzeba zmieniać Obserwatorów aby wspierać nowych Obserwowanych
- Obserwatorzy i Obserwowani mogą być stosowani niezależnie.

Wady:

- Obserwatorzy nie znają się wzajemnie, może to doprowadzać do wygenerowania trudnych do odnalezienia skutków ubocznych.
- Możemy nie wiedzieć w jakiej kolejności Obserwatorzy są o jakimś fakcie informowani.

Przykład użycia wzorca OBSERWATOR

Graficzny Interfejs Użytkownika

Interface ActionListener i klasa JButton (addActionListener, removeActionListener, getActionListeners)

Przyciski w Java

```
1 przycisk1 = new JButton( "Przycisk 1" );
2 przycisk1.addActionListener( new ActionListener() {
3     public void actionPerformed(ActionEvent e) {
4         etykieta.setText( "Kliknieto przycisk 1" );
5     }
6 } );
7
8 przycisk2 = new JButton( "Przycisk 2" );
9 przycisk2.addActionListener( new ActionListener() {
10    public void actionPerformed(ActionEvent e) {
11        etykieta.setText( "Kliknieto przycisk 2" );
12    }
13 } );
```

OBSERWATOR a Java

Gotowe narzędzia

W Java mamy do dyspozycji publiczną klasę `java.util.Observable`, która implementuje funkcjonalność obserwowanego oraz interfejs `java.util.Observer`, który zawiera jedną metodę `update(Observable o, Object arg)`, która jest wywoływana przez obserwowanego, gdy dochodzi do jego zmiany.

Uwaga!

Aby obserwowany faktycznie powiadomił obserwatorów o swej zmianie konieczne jest wykonanie metody `setChanged()` przed `notifyObservers()`.

Wzorzec projektowy POLECENIE

Wzorzec projektowy POLECENIE wg. wikipedii

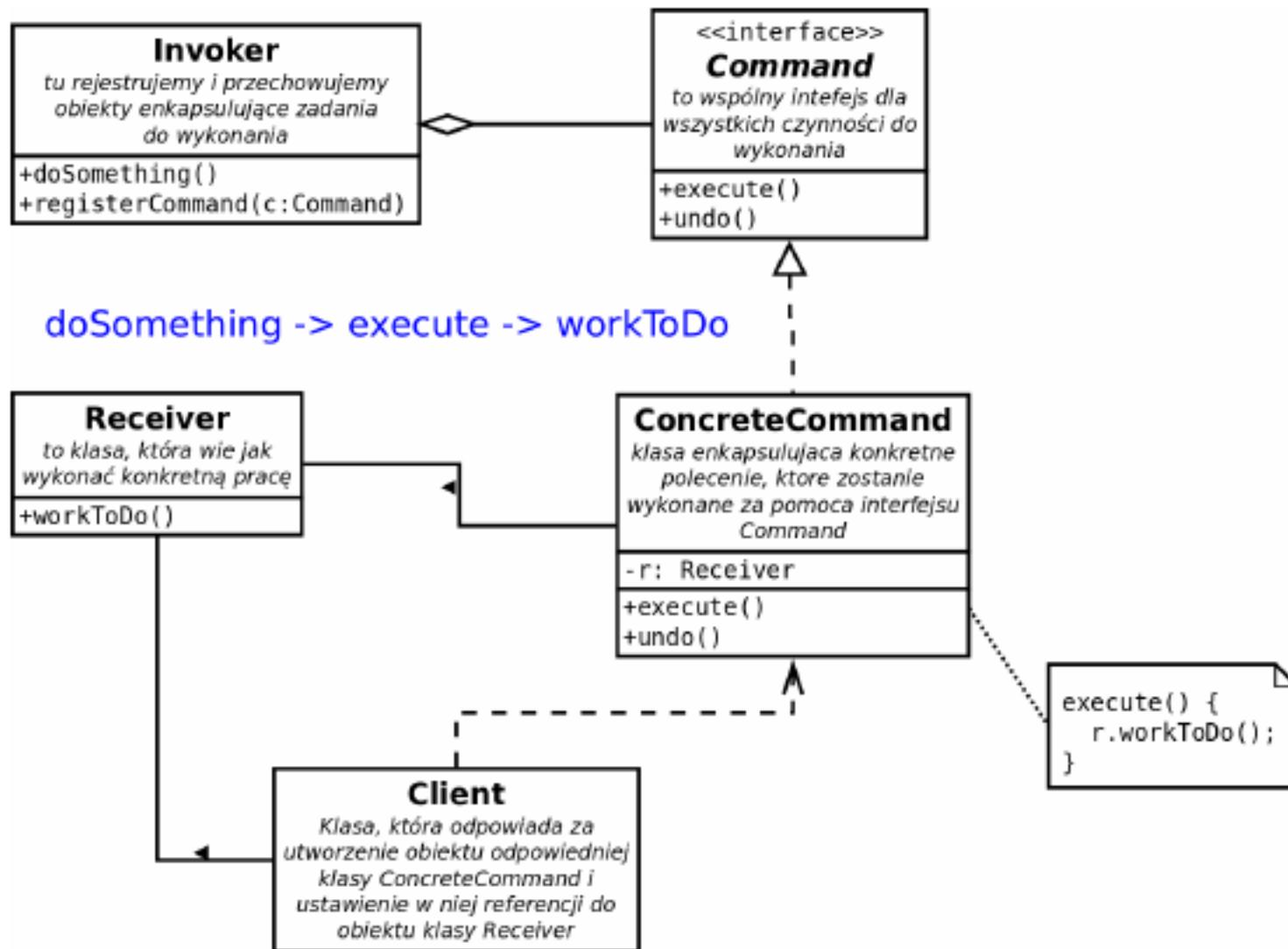
(ang. Command, komenda) w informatyce to jeden z czynnościowych wzorców projektowych traktujący żądanie wykonania określonej czynności jako obiekt, dzięki czemu mogą być one parametryzowane w zależności od rodzaju odbiorcy, a także umieszczane w kolejkach i dziennikach.

[http://pl.wikipedia.org/wiki/Polecenie_\(wzorzec_projektowy\)](http://pl.wikipedia.org/wiki/Polecenie_(wzorzec_projektowy))

Wzorzec czynnościowy

opisuje zachowanie i odpowiedzialność współpracujących ze sobą obiektów.

Wzorzec projektowy POLECENIE



Rysunek: Przykładowy diagram klas

POLECENIE – przykład

Program Polecenie.java

```
1 interface Command {  
2     public void execute();  
3 }  
4  
5  
6 class Wlacznik {  
7     java.util.List<Command> list =  
8         new java.util.ArrayList<Command>();  
9  
10    public void add( Command c ) {  
11        list.add( c );  
12    }  
13  
14    public void wlaczamy() {  
15        for ( Command c : list )  
16            c.execute();  
17    }  
18 }
```

POLECENIE – przykład

Program Polecenie.java

```
1 class Lampka {  
2     public void zapalamy() {  
3         System.out.println("I stala sie swiatlosc...");  
4     }  
5 }  
6  
7 class LampkaCommand implements Command {  
8     private Lampka s;  
9  
10    public LampkaCommand( Lampka s ) {  
11        this.s = s;  
12    }  
13  
14    public void execute() {  
15        s.zapalamy();  
16    }  
17 }
```

POLECENIE – przykład

Program Polecenie.java

```
1 class Silnik {  
2     public void odpalamy() {  
3         System.out.println( "Brum..." );  
4     }  
5 }  
6  
7 class SilnikCommand implements Command {  
8     private Silnik s;  
9  
10    public SilnikCommand( Silnik s ) {  
11        this.s = s;  
12    }  
13  
14    public void execute() {  
15        s.odpalamy();  
16    }  
17 }
```

POLECENIE – przykład

Program Polecenie.java

```
1 class Klient {  
2     public static void main( String[] argv ) {  
3         Wlacznik w = new Wlacznik();  
4  
5         w.add( new SilnikCommand( new Silnik() ) );  
6         w.add( new LampkaCommand( new Lampka() ) );  
7  
8         w.wlaczamy();  
9     }  
10 }
```

Wynik pracy programu:

Brum...

I stala sie swiatlosc...

Wzorzec projektowy POLECENIE - zalety i wady

Zalety:

- oddzielenie operacji od obiektów, na których jest ona wykonywana
- polecenia są reprezentowane jako standardowe obiekty
- możliwość łączenia elementarnych poleceń w polecenia złożone
- łatwość dodawania nowych rodzajów poleceń
- eliminacja długich instrukcji warunkowych

Wady:

- konieczność użycia dodatkowych zasobów

Wzorzec projektowy ADAPTER

Wzorzec projektowy ADAPTER wg. wikipedii

(inne nazwy: Opakowanie, ang. Wrapper) jeden ze strukturalnych wzorców projektowych (zarówno klasowy, jak i obiektowy), którego celem jest umożliwienie współpracy dwóm klasom o niekompatybilnych interfejsach. Adapter przekształca interfejs jednej z klas na interfejs drugiej klasy. Innym zadaniem omawianego wzorca jest opakowanie istniejącego interfejsu w nowy.

[http://pl.wikipedia.org/wiki/Adapter_\(wzorzec_projektowy\)](http://pl.wikipedia.org/wiki/Adapter_(wzorzec_projektowy))

Wzorzec strukturalny

opisuje struktury powiązanych ze sobą obiektów

Wzorzec projektowy ADAPTER

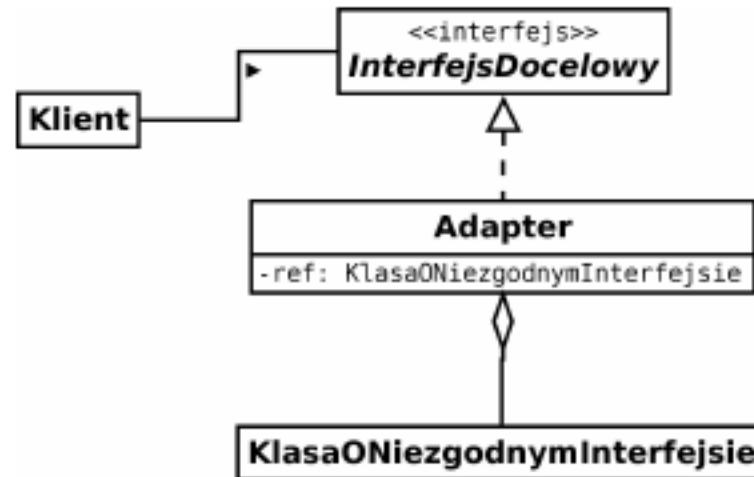
Cel użycia

Nie możemy użyć gotowego kodu klasy, bo jej interfejs nie jest zgodny z naszym oprogramowaniem. Adapter rozwiązuje ten problem dostosowując zastany interfejs do oczekiwanej przez użytkownika.

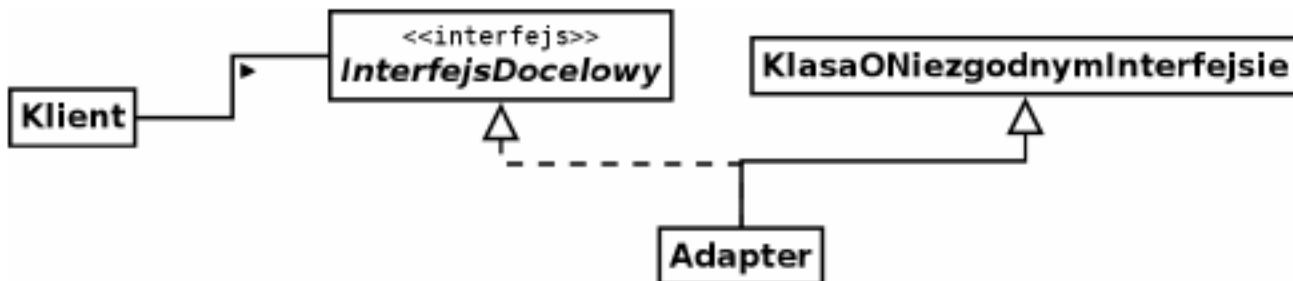
Przy okazji

Mogliśmy stworzyć klasę, która w przyszłości będzie współpracować z innymi klasami, choć jeszcze nie znamy ich interfejsów.

Wzorzec projektowy ADAPTER



Rysunek: Adapter obiektowy używa *agregacji*



Rysunek: Adapter klasowy używa (wielokrotnego) *dziedziczenia*

Wzorzec projektowy ADAPTER - wady i zalety

Adapter klasowy:

- Dziedziczymy, więc adaptujemy tylko jedna klasę — jej podklas już nie
- Dziedziczymy, możemy więc użyć przeładowywania metod
- Tworzony jest tylko jeden obiekt - obiekt opakowania

Adapter obiektowy:

- Do referencji typu klasy adaptowanej można przypisać referencję do jej podklas, czyli nasz adapter działa dla adaptowanej klasy i wszystkich jej podklas.
- Nie ma dziedziczenia — nie ma możliwości przeładowania metod
- Nie ma dziedziczenia — nie wiąże nas kontrakt klasy adaptowanej. Możemy swobodnie używać nazw metod, możemy dodawać nowe funkcjonalności.

Wada obu rozwiązań: niezerowy narzut na wykonywanie operacji za pomocą pośrednika.

Wzorzec projektowy ADAPTER - wersja dwukierunkowa

Adapter dwukierunkowy

Jednocześnie adaptuje interfejs klienta oraz klasy adaptowanej. Może więc jednocześnie działać jako klient i klasa adaptowana.

Wzorzec projektowy ADAPTER - sztuczka

Czy wszystko można zaadaptować?

Oczywiście nie! Nie wszystkie operacje można przeprowadzić na każdej z klas. Lepiej więc ostrzec użytkownika adaptera przed skutkami domniemania, że jakaś metoda jednak działa.

Program Polecenie.java

```
1 public void metodaKtorejNieDaSieZaimplementowac() {  
2     throw new UnsupportedOperationException();  
3 }
```

Wzorce projektowe ADAPTER, MOST, DEKORATOR, PEŁNOMOCNIK i FASADA

Różnica?

Most odpowiada za rozdzielenie interfejsu od implementacji. Adapter zmienia interfejs istniejącej już klasy. Dekorator wzbogaca obiekty ale nie zmienia ich bazowego interfejsu. Pełnomocnik nie zmienia interfejsu tworzy zaś klasę zastępującą oryginalną.

A wzorzec FASADA?

Generalnie, upraszcza interfejs.

Wzorzec projektowy FASADA

Wzorzec projektowy FASADA wg. wikipedii

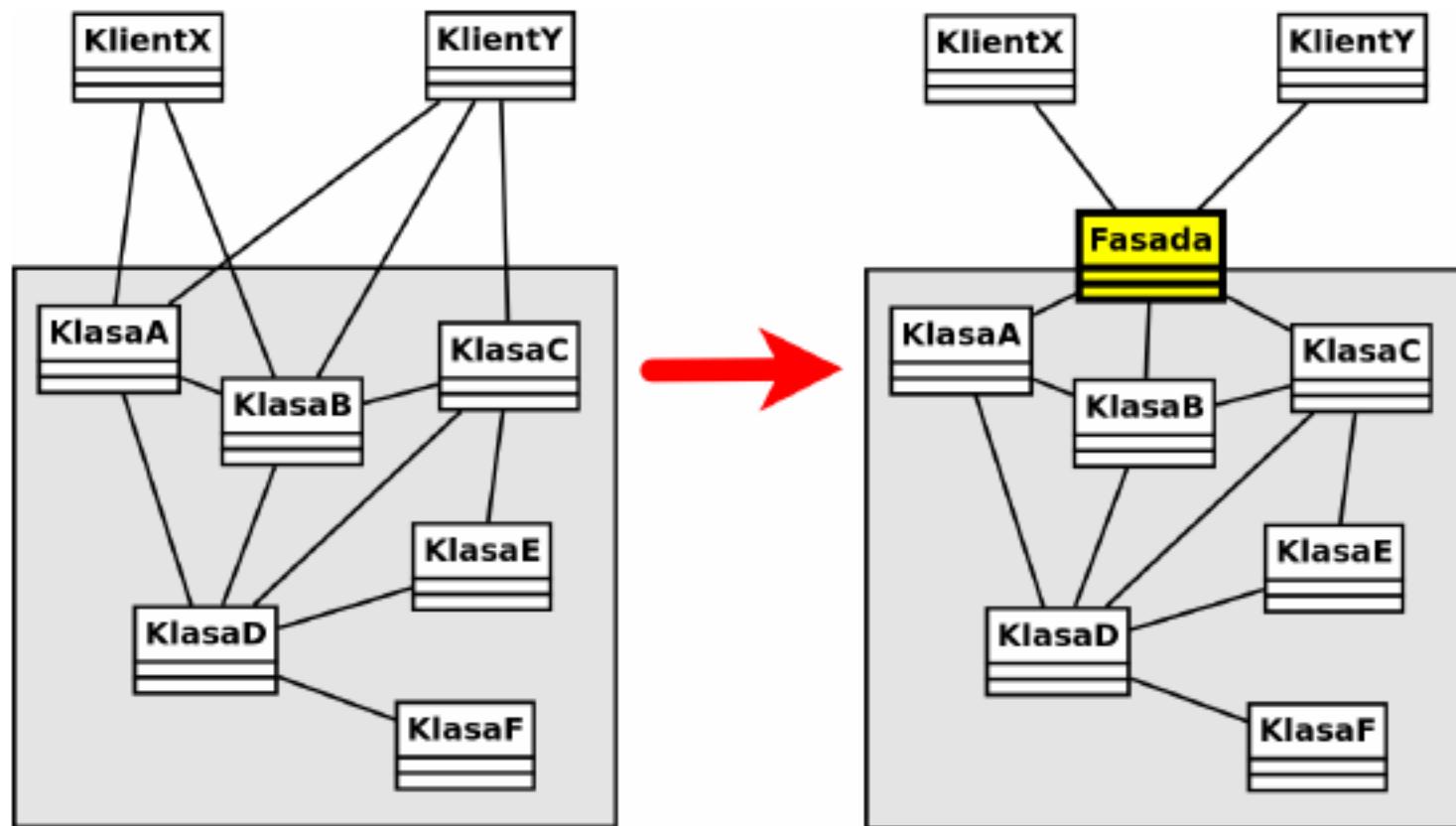
jeden z wzorców projektowych należący do grupy wzorców strukturalnych. Służy do ujednolicenia dostępu do złożonego systemu poprzez wystawienie uproszczonego, uporządkowanego interfejsu programistycznego, który ułatwia jego użycie.

[http://pl.wikipedia.org/wiki/Fasada_\(wzorzec_projektowy\)](http://pl.wikipedia.org/wiki/Fasada_(wzorzec_projektowy))

Fasada

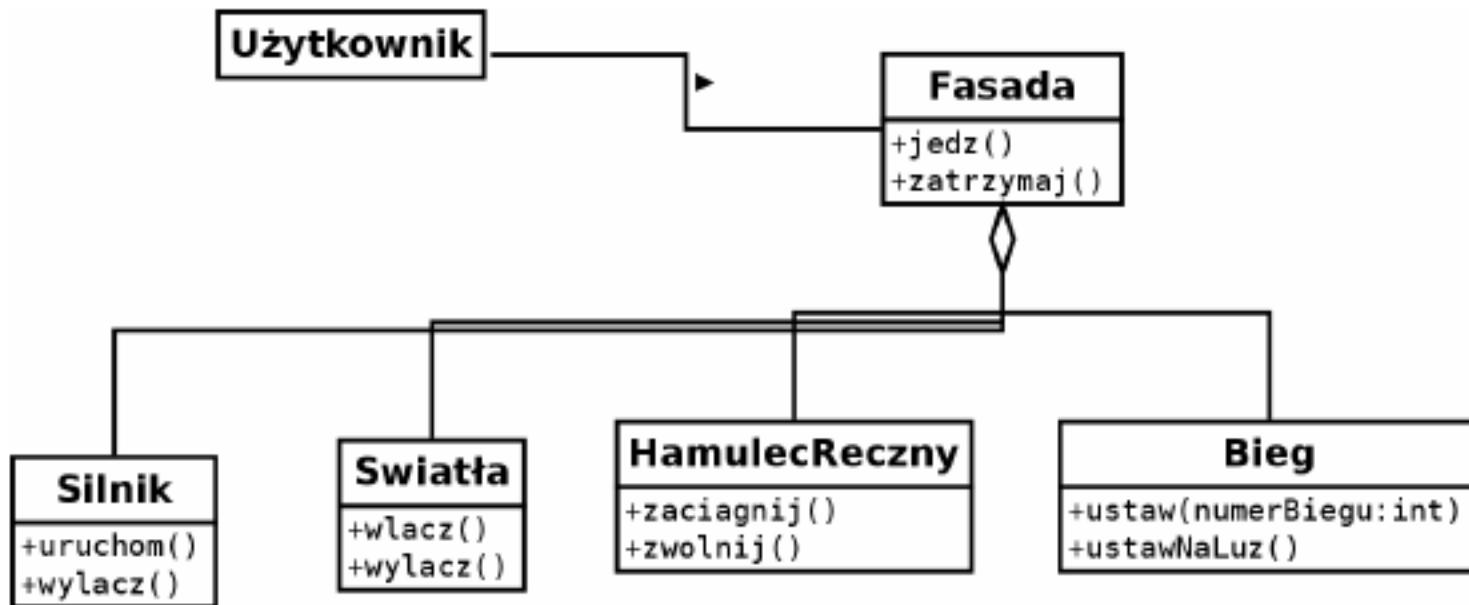
Fasada definiuje interfejs wyższego poziomu. Ale to nie tylko uproszczenie interfejsu, to także możliwość oddzielenia klienta od podsystemu złożonego z wielu elementów o różnych interfejsach.

Wzorzec projektowy FASADA - idea



Rysunek: Idea zastosowania wzorca FASADA.

Wzorzec projektowy FASADA



Rysunek: Przykładowy diagram klas dla wzorca fasada

Prawo Demeter

Prawo Demeter

zasada projektowania oprogramowania (w szczególności projektowania w językach obiektowych), która w skrócony i nieformalny sposób jest ujmowana: "rozmawiaj tylko z (bliskimi) przyjaciółmi".

Inaczej

ang. Law of Demeter (LoD), inaczej Zasada minimalnej wiedzy lub Reguła ograniczania interakcji - ang. Principle of Least Knowledge

W pełnej postaci: Metoda danego obiektu może odwoływać się jedynie do metod należących do:

- metod tego samego obiektu,
- metod dowolnego parametru przekazanego do niej,
- dowolnego obiektu przez nią stworzonego,
- dowolnego składnika klasy, do której należy dana metoda.

Prawo Demeter

W prawie tym chodzi o to, aby nie tworzyć projektu, w którym wiele klas powiązanych jest ze sobą zależnościami. W przeciwnym wypadku, zmiany w jednej części aplikacji przeniosą się na pozostałe.

Mam u unikać tworzenia takiego oto kodu:

Łamanie prawa Demeter

```
1 public void uruchamiamySilnik() {  
2     auto.getSilnik().getRozrusznik().wlacz();  
3 }
```

W efekcie zmiany interfejsu `Rozrusznik` trzeba będzie zmienić również kod wszystkich tych metod, w których jest on używany.

Prawo Demeter

Zalety:

- zmniejszenie zależności
- użytkownik nie musi znać szczegółów (przypominam: Fasada upraszcza interfejs)
- zwiększa się łatwość utrzymania kodu

Wady:

- Zbyt ścisłe trzymanie się prawa Demeter prowadzi do tworzenia wielu metod zajmujących się wyłącznie delegowaniem wykonania pracy
- Może prowadzić do rozrastania się interfejsu, który oferuje operacje (np.: auto.ruszPowoli(), auto.ruszSzybko(), auto.ruszPodGóre() itd.)
- Może skutkować spadkiem wydajności

Wzorzec projektowy FASADA

Zalety używania tego wzorca:

- zmniejszenie zależności pomiędzy klientem, a złożonym systemem,
- wprowadzenie podziału na warstwy — daje to możliwość ich niezależnego rozwoju
- możliwość zablokowania klientowi drogi do bezpośredniego używania złożonego systemu
- kod jest czytelniejszy, czyli łatwiejszy w utrzymaniu i konserwacji

Wzorzec projektowy METODA SZABLONOWA

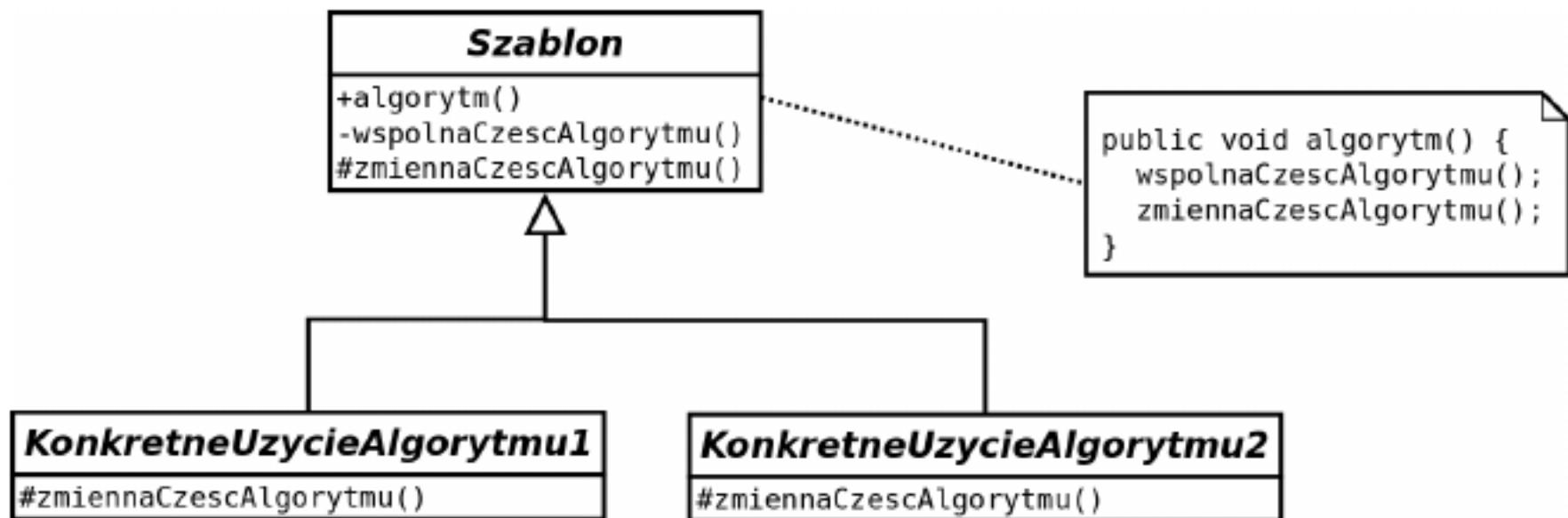
Wzorzec projektowy METODA SZABLONOWA wg. wikipedii

(ang. Template method) w inżynierii oprogramowania jeden z czynnościowych wzorców projektowych. Jego zadaniem jest zdefiniowanie metody będącej szkieletem algorytmu. Niezmienna część algorytmu zostaje opisana w metodzie szablonowej, której klient nie może nadpisać. W metodzie szablonowej wywoływane są inne metody, reprezentujące zmienne kroki algorytmu. Mogą one być abstrakcyjne lub definiować domyślne zachowania. Klient, który chce skorzystać z algorytmu, może wykorzystać domyślną implementację bądź może utworzyć klasę pochodną i nadpisać metody opisujące zmienne fragmenty algorytmu

[http://pl.wikipedia.org/wiki/Metoda_szablonowa_\(wzorzec_projektowy\)](http://pl.wikipedia.org/wiki/Metoda_szablonowa_(wzorzec_projektowy))

Wzorca używamy gdy udaje się wykryć powtarzające się części algorytmu. Choć konkretna realizacja poszczególnych kroków może być inna, to algorytm pozostaje taki sam. *Wzorzec ten pozwala na enkapsulację algorytmu.*

Wzorzec projektowy METODA SZABLONOWA



Rysunek: Przykładowy diagram klas dla wzorca metoda szablonowa

Metoda zaczepowa, hak

Metoda zaczepowa ang. hook method - metoda zadeklarowana w klasie abstrakcyjnej mająca pustą albo domyślną implementację. Klient może ją przedefiniować, ale nie musi — wtedy algorytm pracuje w wersji domyślnej.

Metoda zaczepowa

```
1 abstract class SzablonPrania {  
2     public void pierz() {  
3         wlozPranie();  
4         wsypProszek();  
5         ustawProgram();  
6         nacisnijStart();  
7     }  
8     [...]  
9  
10    protected void ustawProgram() { // tu HAK  
11        program = "Bawelna";  
12    }  
13}  
14}
```

Jeśli program "Bawelna" nam odpowiada nic nie robimy. Jeśli chcemy uprać coś innego, musimy przedefiniować metodę `ustawProgram`.

Jeszcze jeden przykład

Kod z pewnego zadania z kursu Java - chodziło w nim o proste operacje matematyczne na wektorach. Metoda `doIt` realizuje się w różny sposób w zależności od sposobu implementacji metod z niej wywoływanych (`calc` czy `normalize`). Implementacja dostarczana była w klasach potomnych.

Operacje na wektorach

```
1 private void showNormalizeShow( double[] v ) {  
2     show(v);  
3     normalize(v);  
4     show(v);  
5 }  
6  
7 public void doIt( double[] a, double[] b, double[] c ) {  
8     showNormalizeShow( a );  
9     showNormalizeShow( b );  
10    calc( a, b, c );  
11    showNormalizeShow( c );  
12 }
```

Reguła Hollywood

Reguła Hollywood (ang. The Hollywood Principle)

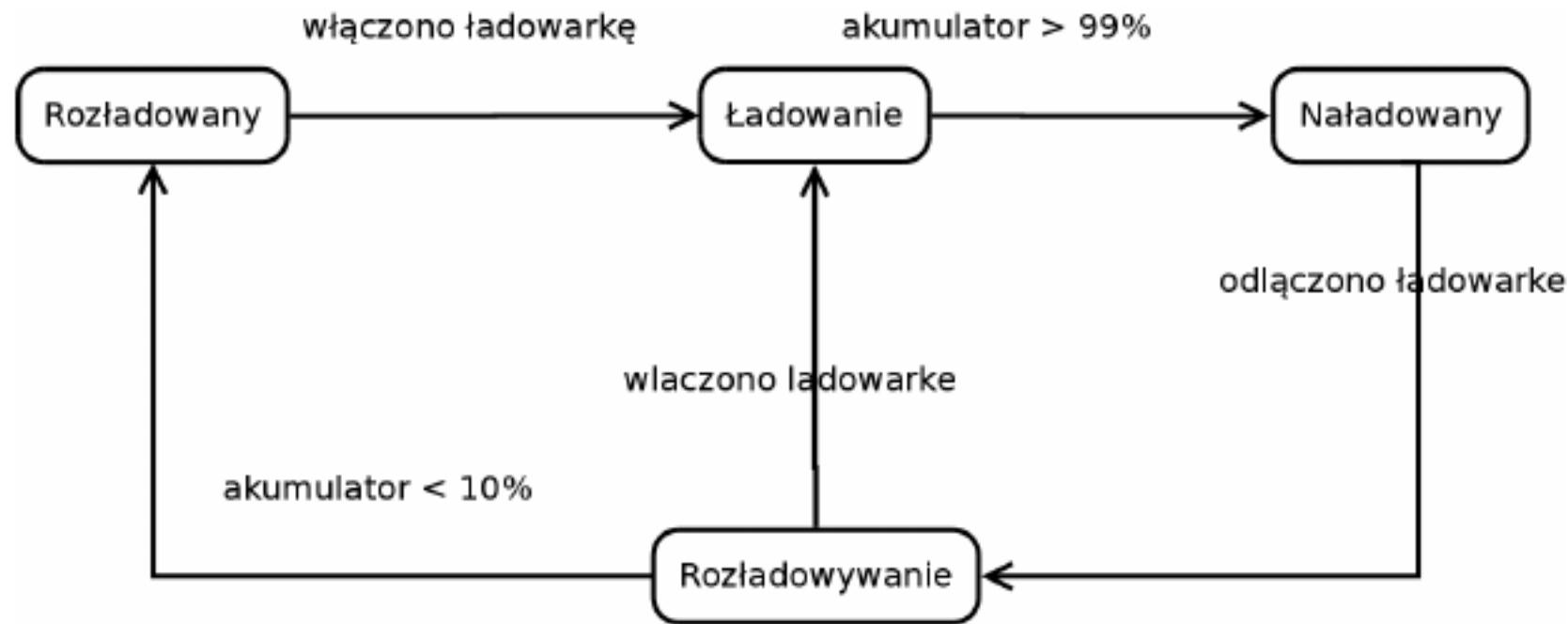
Nie dzwoń do nas, my zadzwonimy do Ciebie

W regule tej chodzi o to, aby komponenty wysokiego poziomu nie zależały od komponentów niskopoziomowych. Dązymy do tego by to klasa bazowa wywoływała metody klas potomnych, a nie odwrotnie. W przypadku wzorca METODA SZABLONOWA algorytm jest określony na poziomie klasy bazowej i to ona wywołuje implementacje metod (umieszczone we własnej klasie i potomnych) potrzebnych do jego konkretnej realizacji.

Uwagi o implementacji METODY SZABLONOWEJ

- Warto używać kontroli dostępu i ograniczać możliwość wywołania metod używanych przez metodę szablonową. Najwygodniej użyć metod typu `protected`.
- Tworząc metodę szablonową warto zminimalizować liczbę metod, które muszą zostać zaimplementowane/przesłonięte w podklasach.
- Odpowiednie nazwy metod mogą być użyteczne i wskazywać, które z nich przeznaczone są do przeddefiniowania.

Przygotowanie do omówienia wzorca STAN



Rysunek: Stany akumulatora

Uwaga: diagram nie jest kompletny.
Jak zapisać program, który odda takie zamiany stanu?

Implementacja czegoś co zmienia stan

Idea:

- To co jest w środku pudełek to stany. Zdefiniujemy stałe (może nawet typ wyliczeniowy), które będą odpowiadać tym stanom.
- Strzałki oznaczają zmianę stanu — to kandydaci na metody. Ktoś wykonuje na obiekcie naszej klasy metodę np. `wlaczonoLadowarke()` i stan ulegnie zmianie (lub nie – przecież ładowarka może być już podłączona)
- Implementacja samego diagramu: trochę warunków `if` albo `switch` i jakoś będzie...

Problemy: jak się nie pomylić, oraz jak zwykle zmiany...

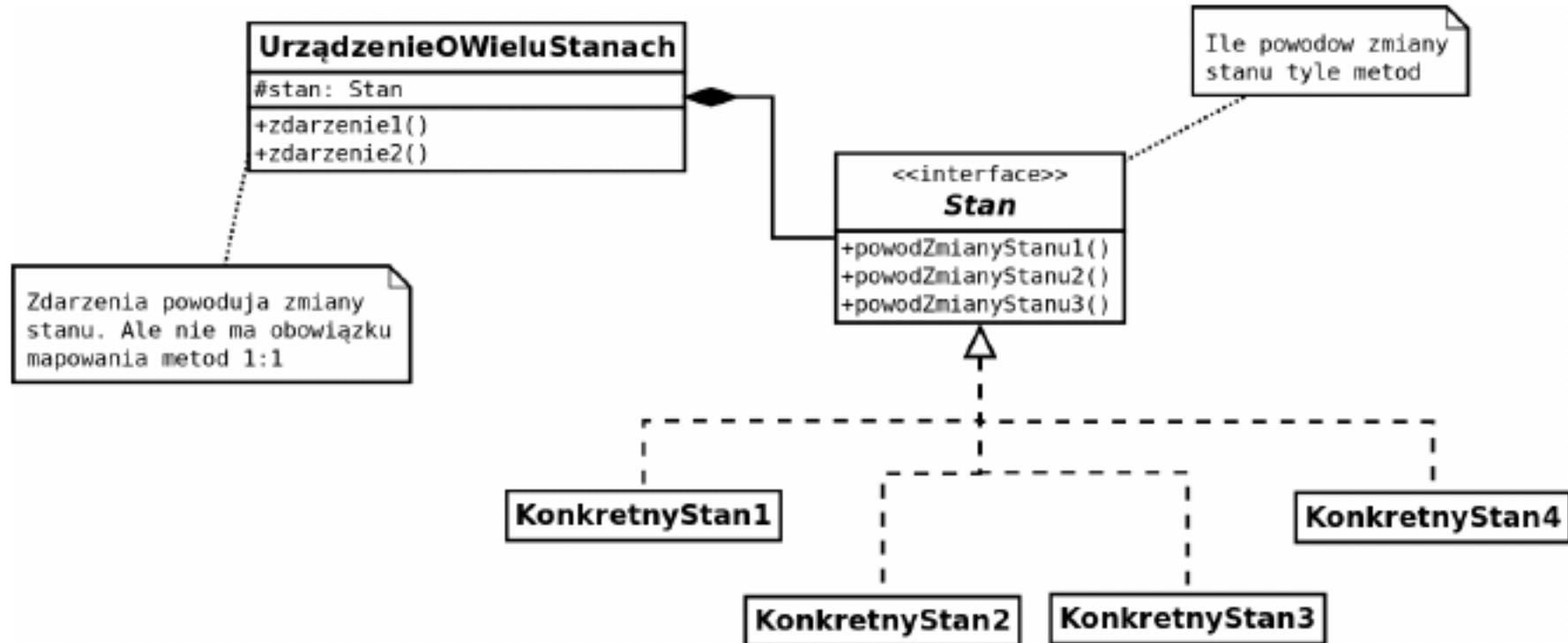
Wzorzec projektowy STAN

Wzorzec projektowy STAN wg. wikipedii

jest to jeden z czynnościowych wzorców projektowych (obiektowy), który umożliwia zmianę zachowania obiektu poprzez zmianę jego stanu wewnętrznego. Innymi słowy — uzależnia sposób działania obiektu od stanu w jakim się aktualnie znajduje

[http://pl.wikipedia.org/wiki/Stan_\(wzorzec_projektowy\)](http://pl.wikipedia.org/wiki/Stan_(wzorzec_projektowy))

Wzorzec projektowy STAN



Rysunek: Przykładowy diagram klas dla wzorca stan

Wzorzec projektowy STAN - przykład

Interfejs zawierający wszystkie powody zmiany stanu

```
1 interface StanAkumulatora {  
2     void wlaczonoLadowarke();  
3     void odlaczonoLadowarke();  
4     void poziomNaladowania( float procent );  
5 }
```

Wzorzec projektowy STAN - przykład

Klasa Akumulator

```
1 class Akumulator {
2     private StanAkumulatora stan;
3     StanAkumulatora ladowanie;
4     StanAkumulatora rozladowywanie;
5     StanAkumulatora naladowany;
6     StanAkumulatora rozladowany;
7
8     protected void ustawStan( StanAkumulatora nowyStan ) {
9         stan = nowyStan;
10    }
11
12    public void wlaczonoLadowarke() {
13        stan.wlaczonoLadowarke();
14    }
15
16    public void odlaczonoLadowarke() {
17        stan.odlaczonoLadowarke();
18    }
19
20    public void poziomNaladowania( float procent ) {
21        stan.poziomNaladowania( procent );
22    }
23
24    public Akumulator() {
25        ladowanie = new StanLadowania( this );
26        rozladowywanie = new StanRozladowywania( this );
27        naladowany = new StanNaladowania( this );
28        rozladowany = new StanRozladowywania( this );
29        stan = rozladowany;
30    }
}
```

Wzorzec projektowy STAN - przykład

Klasa Akumulator cd.

```
1  public StanAkumulatora getStanRozladowywania() {  
2      return rozladowywanie;  
3  }  
4  
5  public StanAkumulatora getStanLadowania() {  
6      return ladowanie;  
7  }  
8  
9  public StanAkumulatora getStanNaladowania() {  
10     return naladowany;  
11  }  
12  
13  public StanAkumulatora getStanRozladowania() {  
14      return rozladowany;  
15  }
```

Wzorzec projektowy STAN - przykład

Klasa - implementacja jednego stanu

```
1 class StanLadowania implements StanAkumulatora {
2     private Akumulator akumulator;
3
4     public void wlaczonoLadowarke() {
5         System.out.println( "Co robisz ? Ladowarka juz jest podlaczona!" );
6     }
7
8     public void odlaczonoLadowarke() {
9         System.out.println( "Przechodzimy do stanu rozladowywania" );
10        akumulator.ustawStan( akumulator.getStanRozladowywania() );
11    }
12
13    public void poziomNaladowania( float procent ) {
14        if ( procent > 99.0 ) {
15            System.out.println( "Przechodzimy do stanu naladowany" );
16            akumulator.ustawStan( akumulator.getStanNaladowania() );
17        }
18    }
19
20    public StanLadowania( Akumulator akumulator ) {
21        this.akumulator = akumulator;
22    }
23 }
```

Nie wszystkie sytuacje umieszczone są na diagramie stanów.

Wzorzec projektowy PEŁNOMOCNIK

Wzorzec projektowy PEŁNOMOCNIK wg. wikipedii

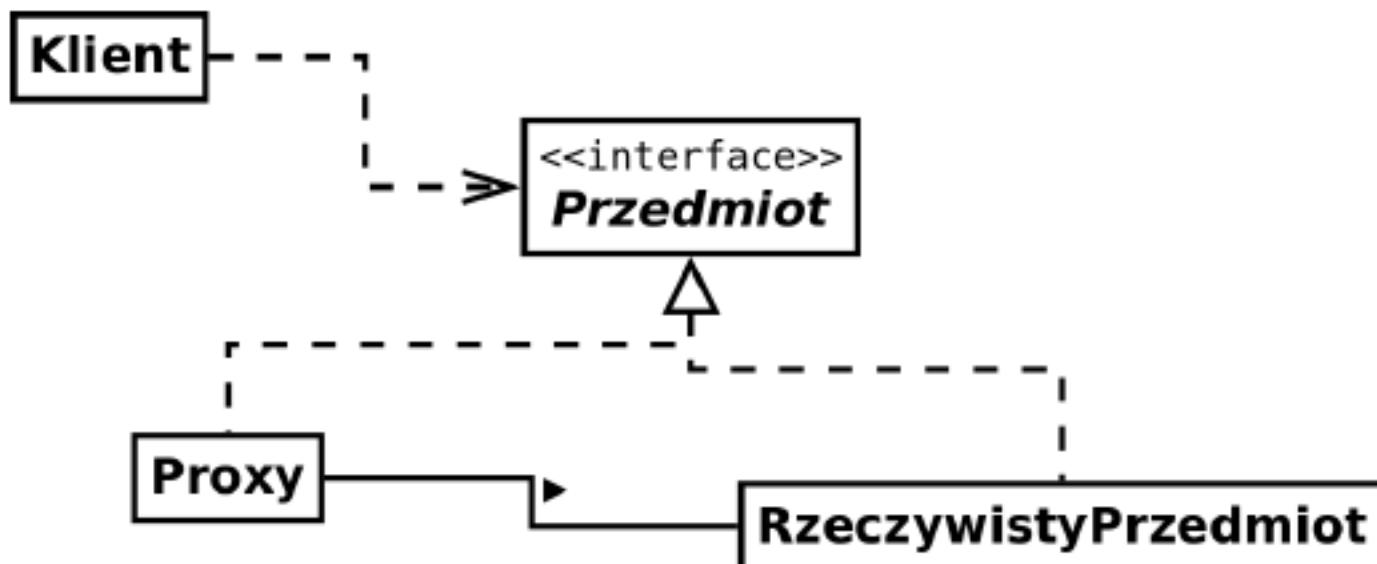
(ang. proxy) jest to jeden ze strukturalnych wzorców projektowych (obiektowy), którego celem jest utworzenie obiektu zastępującego inny obiekt.

[http://pl.wikipedia.org/wiki/Pelnomocnik_\(wzorzec_projektowy\)](http://pl.wikipedia.org/wiki/Pelnomocnik_(wzorzec_projektowy))

Rodzaje

- wirtualny - przechowuje obiekty, których utworzenie jest kosztowne; tworzy je na żądanie; umożliwia optymalizację
- ochraniający – kontroluje dostęp do obiektu sprawdzając, czy obiekt wywołujący ma odpowiednie prawa do obiektu wywoływanego
- zdalny – czasami nazywany ambasadorem; reprezentuje obiekty znajdujące się w innej przestrzeni adresowej
- sprytne odwołanie – czasami nazywany sprytnym wskaźnikiem; pozwala na wykonanie dodatkowych akcji podczas dostępu do obiektu, takich jak: zliczanie referencji do obiektu czy ładowanie obiektu do pamięci

Wzorzec projektowy PEŁNOMOCNIK



Rysunek: Przykładowy diagram klas dla wzorca pełnomocnik

Wzorzec projektowy PEŁNOMOCNIK - przykład

Przykład użycia wzorca PEŁNOMOCNIK

```
1 class ProxyShow {
2     static class MyInvocationHandler implements InvocationHandler {
3         private Object o;
4
5         MyInvocationHandler(Object o) {
6             this.o = o;
7         }
8
9         public Object invoke(Object proxy, Method method, Object[] args)
10            throws Throwable {
11             System.out.println(method + " args " + Arrays.toString(args));
12             return method.invoke(o, args);
13         }
14     }
15
16     private static void user( List<String> lista ) {
17         lista.add("Pomidor");
18         lista.remove("Pomidor");
19     }
20
21     public static void main(String[] args) throws Exception {
22         List<String> lista = (List<String>)Proxy.newProxyInstance(
23             Proxy.class.getClassLoader(), new Class<?>[] { List.class },
24             new MyInvocationHandler( new ArrayList<>() ) );
25         user( lista );
26     }
27 }
```

Wzorzec projektowy PYŁEK

Wzorzec projektowy PYŁEK

(ang. flyweight) Wykorzystuje współdzielenie w celu uzyskania wydajnej obsługi dużej liczby małych obiektów. Podstawowe cel użycia to zmniejszenie wykorzystania pamięci. Wzorzec pokazuje jak współużytkować obiekty bez nadmiernych kosztów.

Typ

Pyłek to wzorzec obiektowy, strukturalny.

Przykład problemu: edytor tekstu, w którym **każdy** znak to osobny obiekt! Naiwna implementacja wygeneruje tyle obiektów ile jest znaków w dokumencie. Kod źródłowy tego wykładu (prezentacja tworzona jest w Latex-ie) liczy w tej chwili około 220000 znaków...

PYŁEK - idea

Zamiast tworzyć dużą liczbę obiektów zastanawiamy się nad tym jak podzielić własności obiektów na takie, które opisują stan wewnętrzny i zewnętrzny obiektu.

Stan wewnętrzny jest zapisywany w pyłku. Pyłek jest współdzielony i ten sam jeden obiekt występuje w wielu kontekstach, jednak w każdym z nich pyłek ma zachowywać jak niezależny obiekt. Aby to osiągnąć informacje ważne dla kontekstu użycia (czyli stan zewnętrzny) nie mogą być współdzielone i to klient musi je dostarczyć w chwili użycia pyłku.

W przykładzie z edytorem tekstu stanem wewnętrznym będzie znak reprezentowany przez pyłek. Miejsce i sposób prezentacji (czcionka) to informacje ważne dla kontekstu użycia. W sumie wystarczy około 100-150 pyłków-znaków.

PYŁEK - implementacja

Do utworzenia implementacji pyłku można zastosować następujące składniki:

- Interfejs, za pomocą którego pyłek otrzyma informację o stanie zewnętrznym a klient uzyska potrzebny efekt.
- Klasa konkretna implementująca powyższy interfejs. W niej zostanie zapisany stan wewnętrzny pyłku. Stan ten nie może zależeć od kontekstu działania.
- Fabryka pyłków. Fabryka zarządza produkcją obiektów-pyłków. Jeśli obiekt spełniający oczekiwania klienta (np. pyłek-litera-A) już istnieje, to jest zwracany. Jeśli go nie ma, to jest tworzony i mieszczany w repozytorium pyłków.
- Klient - uzyskuje obiekty-pyłki. Przechowuje i dostarcza w odpowiednim momencie informację o stanie zewnętrznym.

Klienci nie mogą samodzielnie tworzyć pyłków. Muszą posiłkować się pracą fabryki pyłków, bo tylko ona gwarantuje prawidłowe współużytkowanie obiektów (fabryka zajmuje repozytorium).

PYŁEK - konsekwencje użycia

Wadą jest wzrost obciążenia procesora w szczególności, gdy:

- stan zewnętrzny trzeba każdorazowo obliczać
- pyłki trzeba często wyszukiwać w repozytorium

Zaletą jest obniżenie zapotrzebowania na pamięć. Efekt jest tym większy, gdy:

- stanu zewnętrznego nie trzeba pamiętać lecz można policzyć
- współużytkowane stany wewnętrzne pyłków są duże
- wiele pyłków jest współużytkowanych

W sumie jest to sposób zamiany zapotrzebowania na pamięć w zapotrzebowanie na czas procesora.

Wzorzec projektowy KOMPOZYT

Wzorzec projektowy KOMPOZYT

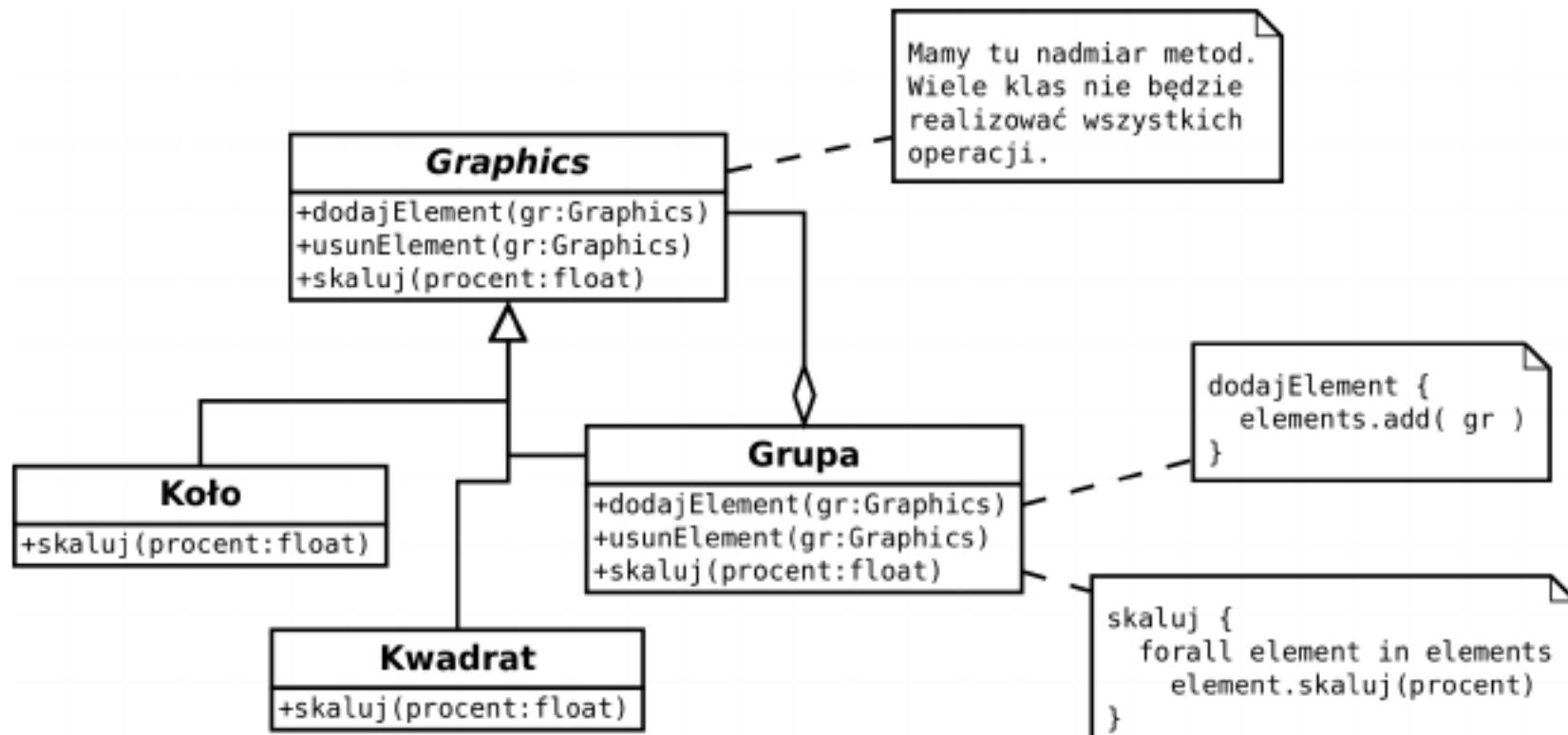
(ang. composite) Wzorzec składa obiekty w struktury typu drzewo - w ten sposób oddawana jest kompozycja (całość-część). Klient może traktować obiekty-składowe i obiekty-kompozyty w ten sam sposób.

Typ

Kompozyt to wzorzec obiektowy, strukturalny.

Wzorzec pozwala na używanie w identyczny sposób obiektu jak i grupy obiektów. Typowy przykład to grupowanie obiektów w programie graficznym. W ten sam sposób użytkownik może skalować pojedynczy obiekt jak i ich grupę.

Wzorzec projektowy KOMPOZYT



Rysunek: Przykładowy diagram klas dla wzorca Kompozyt

Wzorzec projektowy KOMPOZYT - współdziałanie

Tabela: Współdziałanie obiektów

Wykonawca	Metoda wykonująca pracę	Metoda zmieniająca strukturę
Obiekt prosty	wykonuje pracę	jest ignorowana
Kompozyt	zleca pracę komponentom	dodaje/usuwa elementy podległe

Typ bazowy - definiuje metody używane do wykonywania pracy (np. skalowanie), metody do operowania na strukturze (składanie obiektów) oraz metody udostępniające podległe elementy składowe. Metody mogą posiadać domyślną implementację na poziomie typu bazowego.

Interfejs typu bazowego jest maksymalizowany. Dąży się do umieszczenia w nim możliwie jak największej liczby wspólnych metod.

Wzorzec projektowy KOMPOZYT - wady/zalety

Zalety:

- Możliwość definiowania hierarchii składających się z elementów prostych i złożonych.
- Uproszczenia kodu klienta, bo klient tak samo używa obiektu prostego jak i kompozytu.
- Ułatwienie dla dodawania nowych rodzajów komponentów. Kod klienta nie musi być z tej okazji modyfikowany.

Wady:

- Projekt może stać się zbyt ogólny. Przy małej liczbie metod interfejsu może się zdarzyć, że dołączone zostaną do komponentu obiekty, których nikt się tam nie spodziewa.
- Maksymalizacja interfejsu jest sprzeczna z zasadą poprawnego projektowania hierarchii klas - powinny być deklarowane tylko operacje sensowne w podklasach.
- Konieczność zapewnienia równowagi pomiędzy bezpieczeństwem (możliwość realizacji operacji strukturalnych na obiektach typów prostych) a przeźroczystością (brak konieczności rozróżniania typu obiektów)

Wzorzec projektowy ŁAŃCUCH ZOBOWIĄZAŃ

Wzorzec projektowy ŁAŃCUCH ZOBOWIĄZAŃ

(ang. chain of responsibility) Pozwala uniknąć powiązania nadawcy żądania z obiektem, które je zrealizuje. Umożliwia wprowadzenie niejawnego odbiorcy. Łączy w łańcuch obiekty odbiorcze i przekazuje pomiędzy nimi żądanie aż do obiektu, który je obsłuży.

Typ

Łańcuch zaobowiązań to wzorzec obiektowy, operacyjny.

Wzorzec projektowy ŁAŃCUCH ZOBOWIĄZAŃ

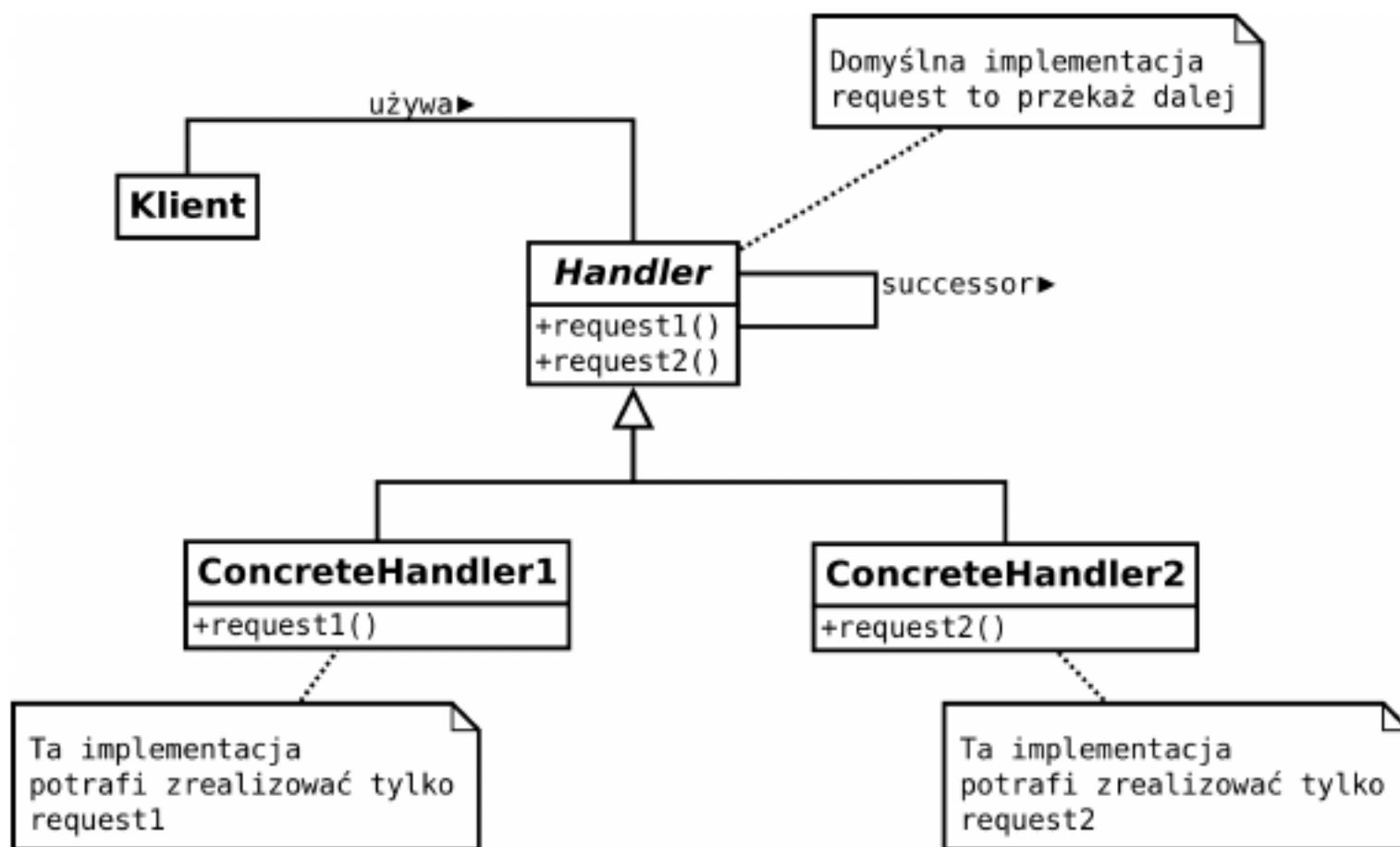
Niejawny odbiorca

Pierwszy z obiektów łańcucha zobowiązań odbiera żądanie. Realizuje je lub przekazuje je do kolejnego obiektu. Kolejny obiekt realizuje żądanie albo przekierowuje do kolejnego obiektu. Itd. W efekcie nie wiemy, który z obiektów ostatecznie zrealizował żądanie. Obiekt inicjujący żądanie nie zawsze dysponuje referencją do obiektu, który zrealizuje żądanie (nie zawsze, bo żądanie zrealizować może obiekt pierwszy z łańcucha).

Interfejs

Aby odbiorca pozostawał niewidzialny - każdy z obiektów musi dysponować identycznym interfejsem obsługi żądania i przekazywania żądań dalej w łańcuchu. Przekazywanie żądania dalej może być implementacją domyślną. Klasa, która potrafi żądanie obsłużyć metodę tą przeddefiniowuje.

Wzorzec projektowy ŁAŃCUCH ZOBOWIAZAŃ



Rysunek: Przykładowy diagram klas dla wzorca Łańcuch Zobowiązań

Wzorzec projektowy ŁAŃCUCH ZOBOWIĄZAŃ - wady/zalety

Zalety:

- Ogranicza powiązania obiektów. Obiekty nie muszą wiedzieć kto realizuje żądanie. Obiekt realizujący żądanie nie wie kto je zlecił. Zlecający nie musi przechowywać referencji do wszystkich potencjalnych odbiorców.
- Zwiększenie elastyczności. Podział zadań pomiędzy obiekty może być modyfikowany w trakcie pracy programu.
- Żądanie może być przekazane do wielu odbiorców.

Wady:

- Brak gwarancji, że któryś z obiektów zadanie obsłuży
- Łańcuch może zostać niepoprawnie zbudowany (np. można sobie wyobrazić działanie, o którego powodzeniu zależy kolejność wykonania żądań).

Wzorzec projektowy ODWIEDZAJĄCY

Wzorzec projektowy ODWIEDZAJĄCY

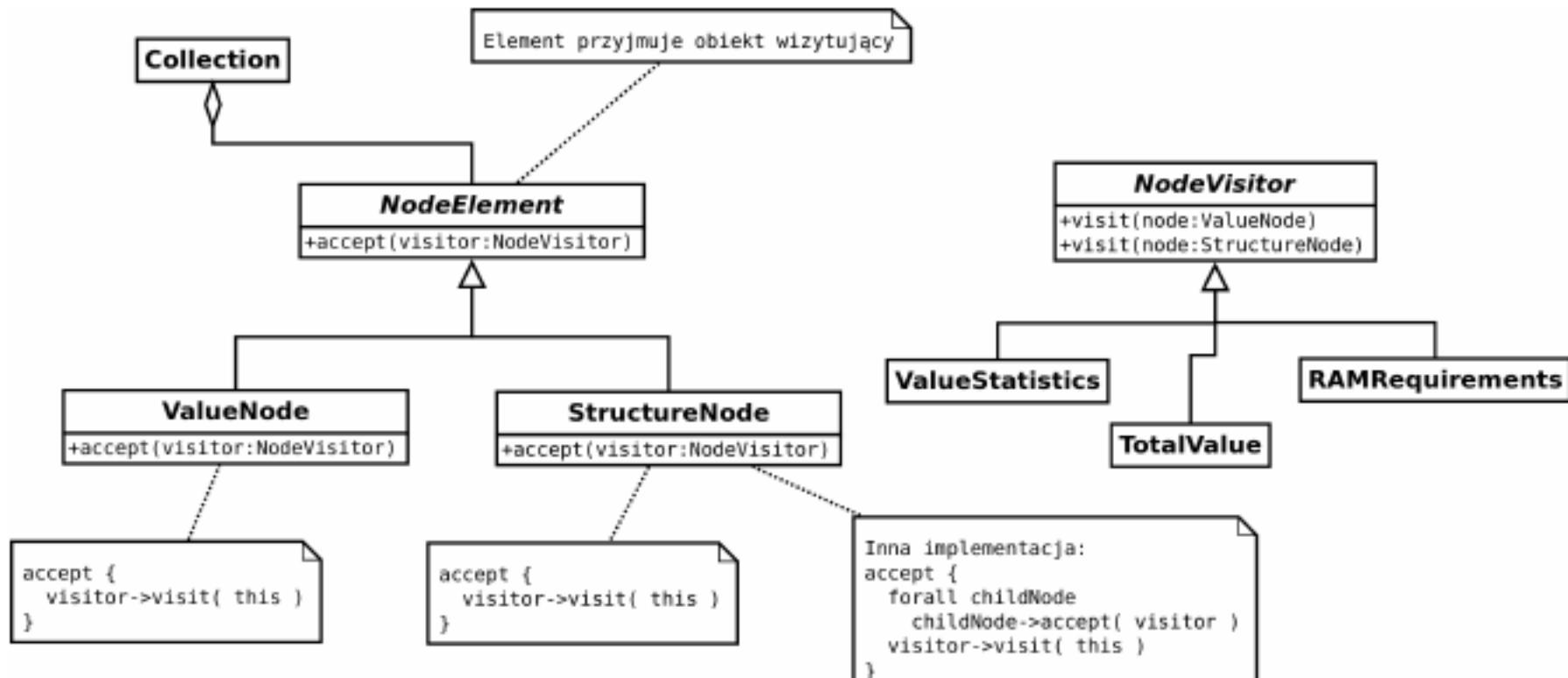
(ang. visitor) Umożliwia dodanie nowej operacji bez konieczności zmiany klas elementów, na których działa. Odseparowuje algorytm od struktury obiektów.

Typ

Odwiedzający (inaczej wizytator) to wzorzec obiektowy, operacyjny.

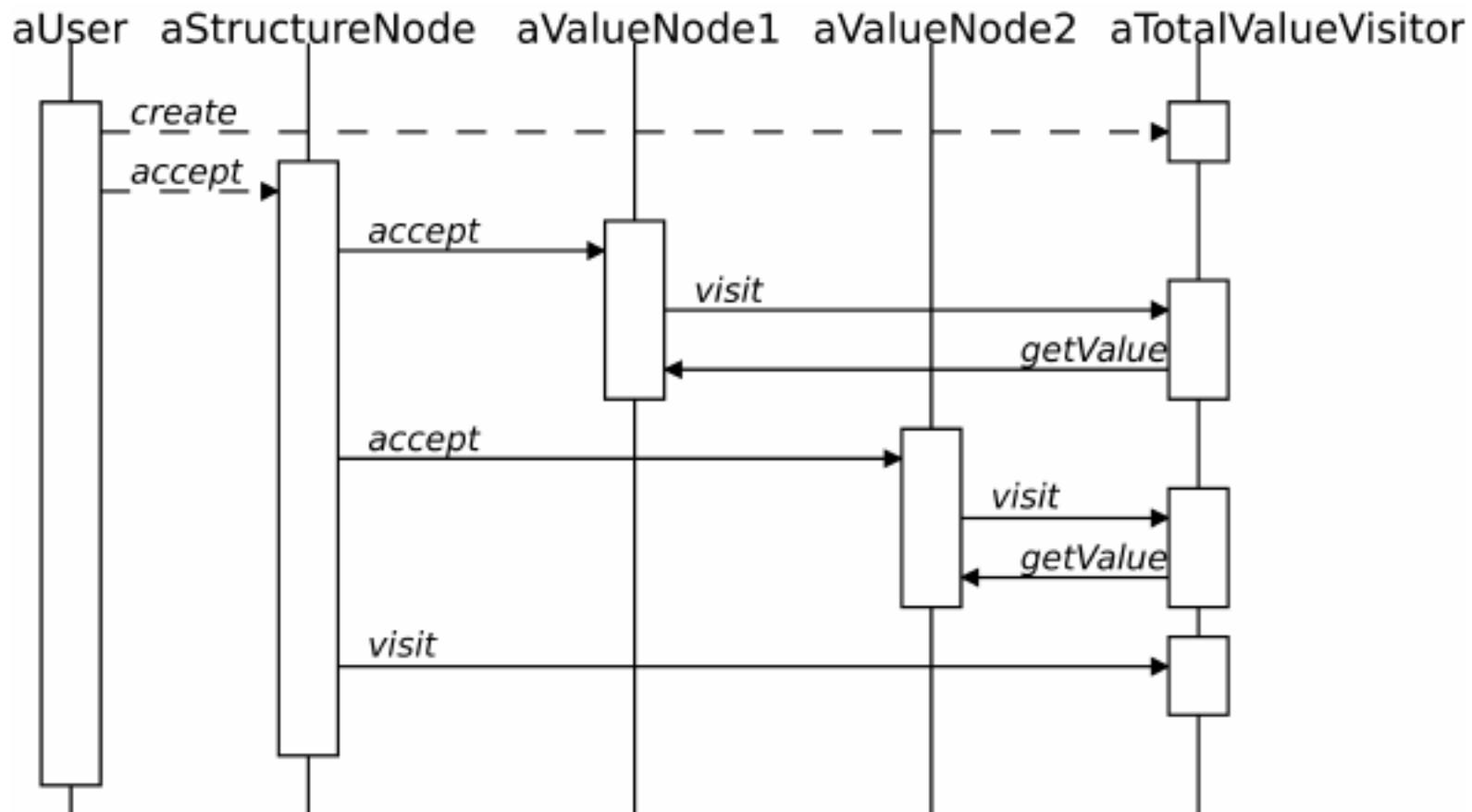
Przykład zastosowania: mamy pewną strukturę klas, która pozwala na utworzenie obiektów i powiązanie ich np. w postaci drzewa. Są klasy, które pozwalają na pobranie wartości przechowywanej w obiekcie, są i takie, które umożliwiają poznanie ilości używanej pamięci (np. elementu budujące strukturę, same wartości nie mają, ale łączą inne i potrzebują na to pamięci). Chcemy dodać nowe operacje np. wyznaczenie statystyki wystąpienia danej wartości, zliczenie wartości elementów w drzewie, zliczenia obiektów, których zapotrzebowanie na pamięć jest większe niż podany limit. Nie chcemy jednak zmieniać kodu samych klas tworzących strukturę.

Wzorzec projektowy ODWIEDZAJĄCY



Rysunek: Przykładowy diagram klas dla wzorca Odwiedzający

Wzorzec projektowy ODWIEDZAJĄCY



Rysunek: Diagram interakcji dla wzorca Odwiedzający

ODWIEDZAJĄCY - kiedy stosować

Warunki zastosowania wzorca odwiedzający.

- Gdy zestaw klas definiujących strukturę jest statyczny (nie ulega częstym zamianom), ale często pojawiają się potrzeby tworzenia nowych operacji.
- Gdy struktura obiektów zawiera klasy o różnych interfejsach i jest potrzeba wykonywania operacji zależnych od konkretnych klas.
- Nowych operacji nie chcemy umieszczać bezpośrednio w klasach tworzących strukturę.
- Jeśli różne aplikacje używają tej samej struktury obiektów ale wymagają różnych na niej operacji.

ODWIEDZAJĄCY - zalety/wady

Zalety:

- Wzorzec ułatwia dodawanie nowych operacji
- Ułatwia połączenie powiązanych operacji i wyodrębnienie niepowiązanych. Zachowania mogą być wyodrębnione jako podklasy Visitor.
- Możliwe jest odwiedzanie klas, które nie są powiązane wspólnym interfejsem (metody klasy Visitor mogą mieć argumenty dowolnego typu).
- Akumulowanie stanu jest łatwe. Stan zapisany jest w obiekcie odwiedzającym.

Wady:

- Utrudnia dodawanie nowych elementów składowych struktury odwiedzanych obiektów. Nowy typ to nowa metoda w Visitor i nowa implementacja klasy odwiedzającej.
- Utrudnienie enkapsulacji - odwiedzający muszą mieć dostęp do stanu elementu.

Wzorzec projektowy PAMIĄTKA

Wzorzec projektowy PAMIĄTKA

(ang. memento) Wzorzec pozwala zapisać wewnętrzny stan obiektu bez naruszania hermetyzacji. Zapisany stan może zostać użyty do późniejszego przywrócenia stanu obiektu.

Typ

Pamiątka to wzorzec obiektowy, operacyjny.

Przykład zastosowania: operacja UNDO cofająca stan systemu od wcześniejszego.

PAMIĄTKA - implementacja

- Idea implementacji sprowadza się do przechowywania stanu w obiekcie-pamiątce. Dane do obiektu-pamiątki wprowadza ten obiekt, którego stan chcemy zachować.
- Obiekt-pamiątka powinien posiadać dwa interfejsy - tajny do zapisania w nim stanu i zawężony ale publiczny do zarządzania obiektami pamiętek.
- W miarę możliwości języka, w którym wzorzec jest implementowany dostęp do danych zapisanych w pamiętce powinno mieć tylko jej źródło. C++ - klasa zaprzyjaźniona. Java - klasa wewnętrzna.
- Pamiątki są pasywne - tylko źródło zapisuje stan i ewentualne go odczytuje.

PAMIĄTKA - zalety/wady

Zalety:

- Możliwe jest zachowanie enkapsulacji.
- Klasa, która jest źródłem pamiętki (to jej stan jest zapisywany) nie musi zajmować się przechowywaniem i zarządzaniem pamięatkami.

Wady:

- Kopiowanie informacji do pamiętek i utrzymywanie pewnej ich liczby może być kosztowne.
- Nie zawsze jest możliwe ukrycie interfejsu do wprowadzenia do pamiętki danych
- Zarządzanie pamięatkami jest trudne, bo nie wiadomo jak zapisany jest stan i jak kosztowne będzie jego utrzymanie i odtworzenie.

Wzorzec projektowy MEDIATOR

Wzorzec projektowy MEDIATOR

(ang. mediator) Pozwala na ograniczenie bezpośredniego oddziaływanie na siebie obiektów. Kapsułkuje informacje o interakcji pomiędzy obiektami z pewnego zbioru. Ułatwia utrzymanie luźnych powiązań pomiędzy klasami.

Typ

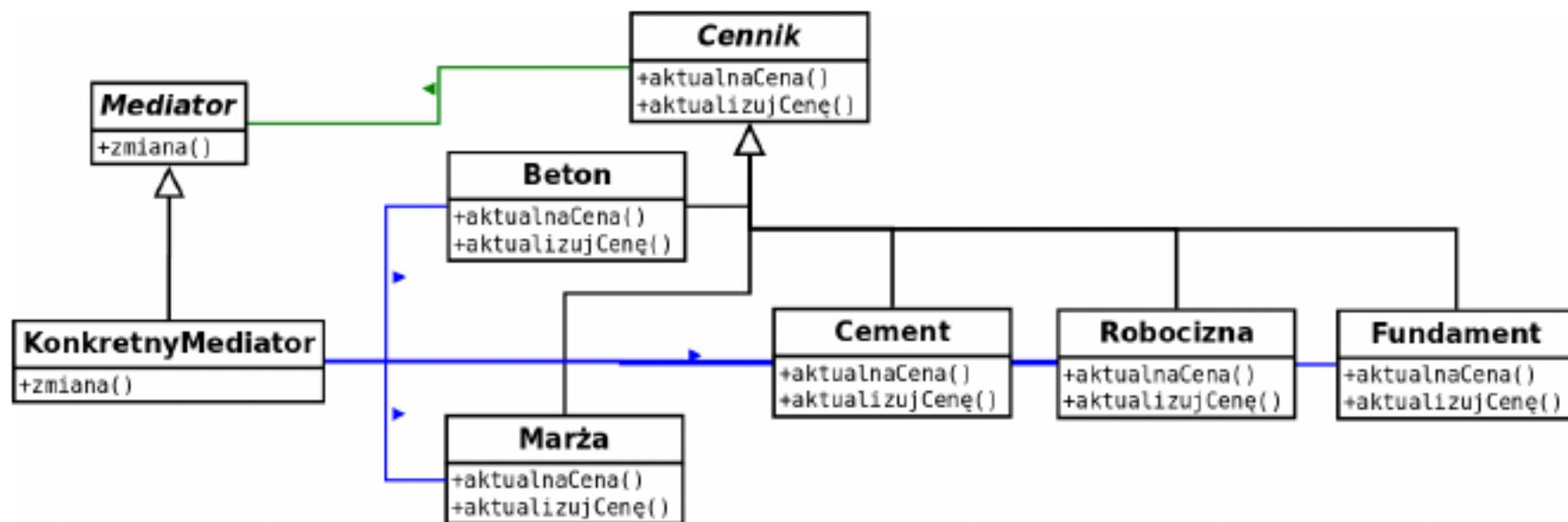
Mediator to wzorzec obiektowy, operacyjny.

Przykład: jedna operacja wymaga współpracy wielu obiektów, które odpowiadają za różne zachowania (wynik projektowania obiektowego i użycia STRATEGII). Obiekt zlecający operację musi znać wszystkie obiekty, które konieczne są do jej przeprowadzenia. Wraz ze wzrostem liczby powiązań spada szansa na ponowne użycie, a system działa jak monolit. Zamiany wprowadza się trudno. Ponadto, różne akcje mogą mieć różnych inicjatorów.

Wzorzec projektowy MEDIATOR - idea

Idea

Wspólne zachowanie jest enkapsulowane w odrębnym obiekcie-mediatorze. Mediator przejmuje na siebie zadanie organizacji komunikacji między obiektami, dzięki temu obiekty nie muszą już komunikować się bezpośrednio ze sobą. Wystarczy aby obiekty znały wyłącznie mediatora.



Rysunek: Prosty diagram przedstawiający użycie wzorca MEDIATOR

Wzorzec projektowy MEDIATOR - stosowanie i implementacja

Kiedy stosować wzorzec MEDIATOR?

- Zestaw obiektów komunikuje się z dobrze zdefiniowanym, ale skomplikowanym sposobem.
- Powtórne użycie jest trudne, bo obiekt komunikuje się z wieloma innymi obiektami.
- Nie trzeba tworzyć wielu podklas aby dostosować zachowania rozproszone w wielu klasach.

Implementacja wzorca polega na np.:

- Utworzeniu nadtypu, który reprezentować będzie wspólny dla różnych Mediatorów interfejs użytkownika.
- Utworzenie do rozwiązywania konkretnych problemów stosownych podklas Konkretnych-Mediatorów.
- Zaimportowanie Mediatorów z zastosowaniem wzorca Obserwator aby umożliwić przekazywanie informacji o zmianach w stanie współpracujących obiektów.
- Zlecaniu wykonania operacji do Konkretnego-Mediatora. Konkretny-Mediator zna metody innych klas i rozsyła polecenia do poszczególnych obiektów.

MEDIATOR - zalety/wady

Zalety:

- Ogranicza tworzenie podklas. Mediator pozwala umieścić w jednym miejscu zachowanie.
- Pozwala na luźne powiązanie pomiędzy współpracującymi obiektami. Klasy Mediatorów i klasy wykonujące działania mogą być rozwijane niezależnie.
- Mediator upraszcza interfejsy - zamiast interakcji wiele-do-wielu jest jeden-do-wielu.
- Hernetyzuje informacje o współdziałaniu obiektów.

Wady:

- Mediator zmienia złożoność interakcji pomiędzy obiektami w złożoność samego mediatora. W efekcie to mediator może stać się trudnym do konserwacji monolitem.

Wzorzec projektowy MOST

Wzorzec projektowy MOST

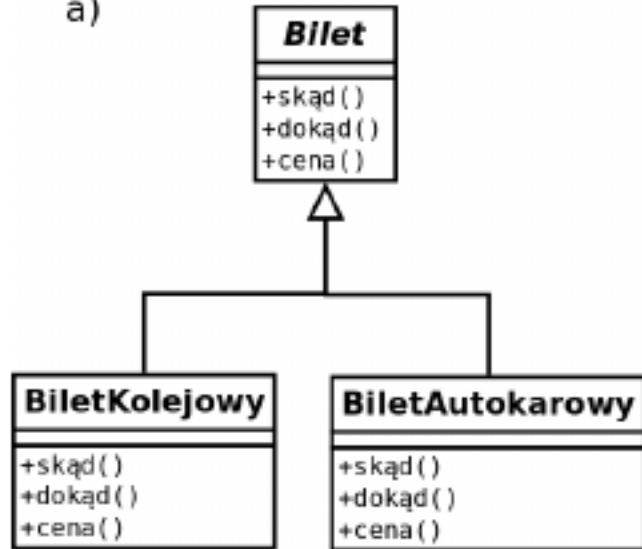
(ang. bridge) Pozwala niezależnie modyfikować abstrakcję i implementację.

Typ

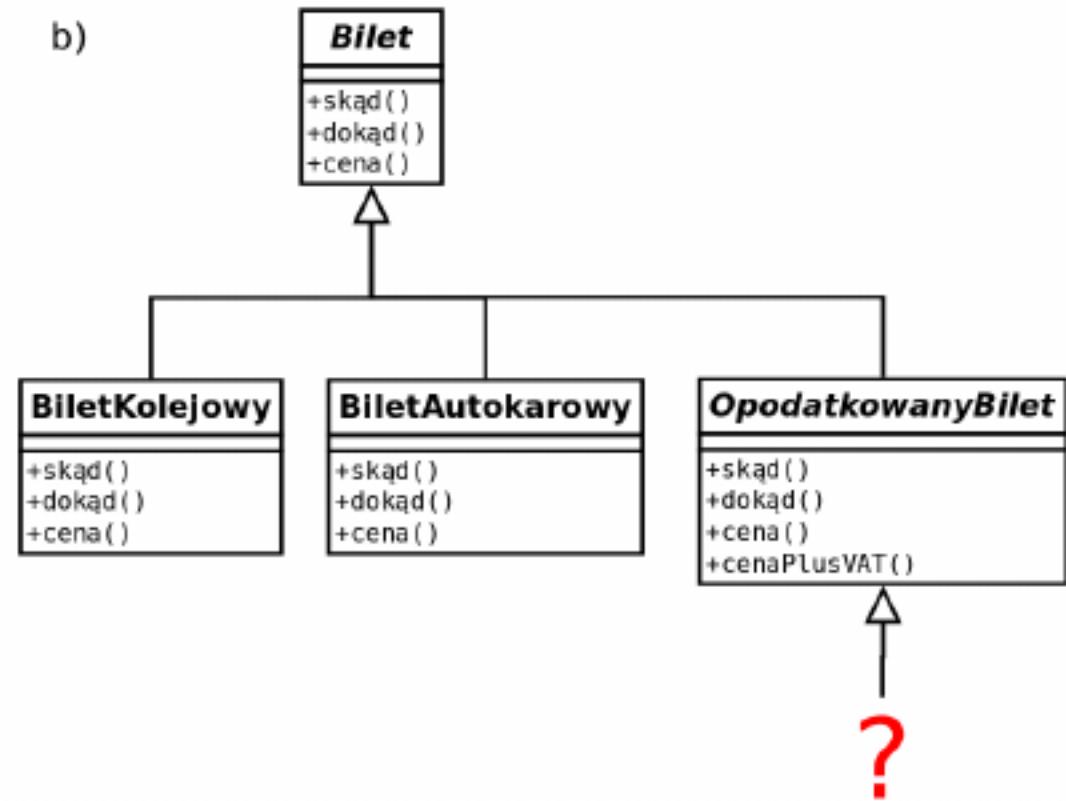
Mediator to wzorzec obiektowy, strukturalny.

Wzorzec projektowy MOST - problem

a)



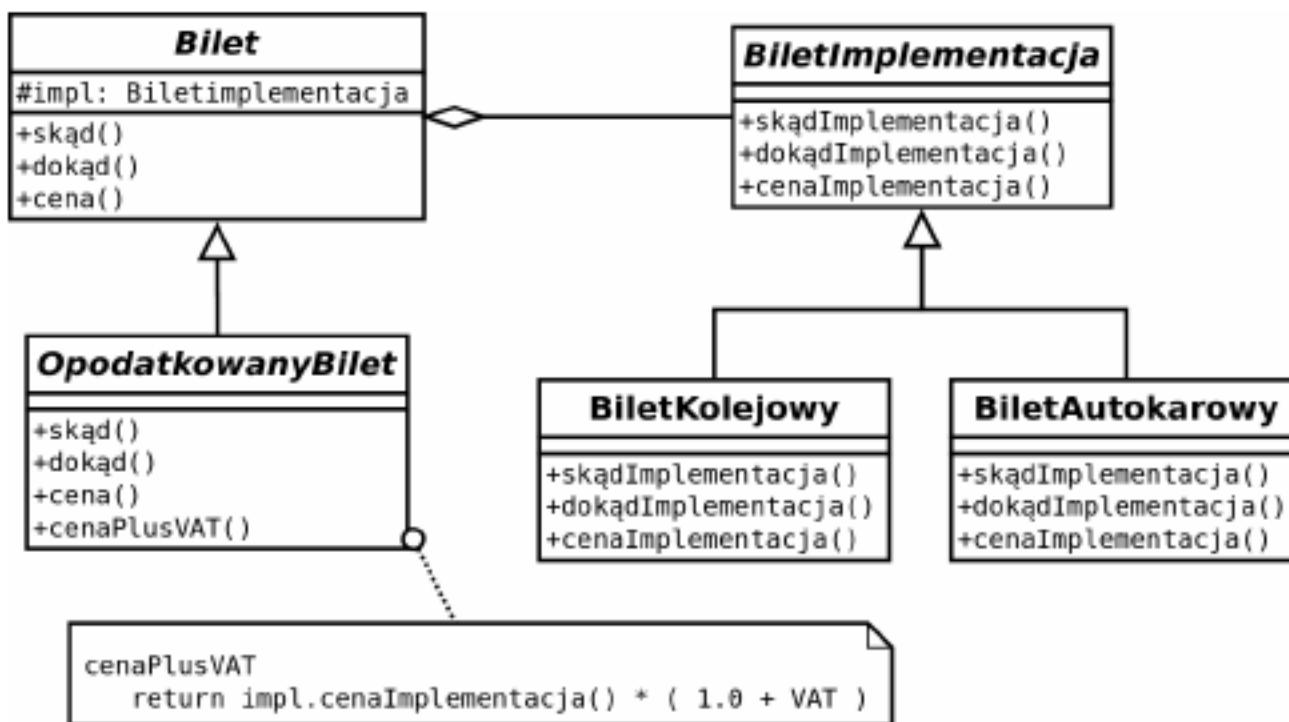
b)



Rysunek: Jak rozwijać projekt?

Wzorzec projektowy MOST - rozwiązanie

Rozwiązaniem są dwie hierarchie - jedna pozwala rozwijać abstrakcje a druga implementacje.



Rysunek: Przykład użycia wzorca MOST.

Wzorzec projektowy MOST - zalety

- Interfejs jest rozdzielony od implementacji
- MOST pozwala na łatwiejszy i niezależny rozwój.
- Możliwość podziału na warstwy co pozwala poprawić strukturę projektu
- Szczegóły implementacji mogą zostać ukryte przed użytkownikami

Klasy, obiekty, mapy... dygresja

W naszej aplikacji pojawia się wiele praktycznie identycznych rzeczy (niekoniecznie są identycznego typu. Np. kolejne owoce w sklepie, kolejne gatunki kwiatów w kwiaciarni). Jak pozwolić aplikacji rosnąć bez generowania tym nadmiernych kosztów?

- Klasy, podklasy - pozwalają używać polimorfizmu, ale trzeba je napisać - kod źródłowy nie generuje się sam
- Obiekty jednej klasy - są tego samego typu, ale dzięki STRATEGII ich zachowanie może być różne. Dobrze przygotowany zestaw zachowań może doprowadzić do składania nowych elementów ze starych klocków.
- Mapy - liczba atrybutów może rosnąć bez jakiegokolwiek przebudowy kodu źródłowego. Rozsądnie napisane metody nadal potrafią sortować, sprawdzać identyczność obiektów...

Wybranie rozwiązania z mapą może okazać się bardzo elastyczne.

Rozpoczynamy pisanie kodu

Kod dla klienta

Sposób na dobry kod

odkładać pisanie kodu na później. Czas wykorzystać na projektowanie aplikacji.

Dobry projekt

nie jest **bezpośrednio** ważny dla klienta. Dla niego ważna jest działająca zgodnie z jego oczekiwaniami aplikacja.

Od ogółu do szczegółu

Najpierw sprawy ogólne, a potem szczegóły. Na każdym z etapów należy kierować się zasadami analizy i projektowania obiektowego. Gdy okazuje się, że na jakimś etapie nie wiemy co zrobić należy cofnąć się do spraw bardziej ogólnych - może nie do końca rozumiemy jak ma działać aplikacja.

Dobre oprogramowanie tworzy się iteracyjnie! Kolejne powtórki pozwalają lepiej zrozumieć zagadnienie. Praca wykonana wcześniej owocuje.

Ku szczegółom

Programowanie w oparciu o możliwości aplikacji.

Wybieramy jedną z konkretnych możliwości aplikacji i poprzez analizę, projekt dochodzimy do jej pełnej implementacji. Konkretna możliwość to fragment funkcjonalności aplikacji, która jest istotna dla przyszłego użytkownika.

Programowanie w oparciu o przypadek użycia.

Uwagę skupiamy na konkretnym sposobie użycia (przejścia) aplikacji. Mamy ścieżkę prowadzącą od pewnego stanu początkowego do konkretnego stanu końcowego - zgodnego z oczekiwaniami użytkownika. Doprowadzamy do implementacji pełnej ścieżki.

Wymagania

Oba sposoby są równie dobre. Oba opierają się na zgłoszonych wymaganiach dla systemu, czyli w efekcie dają to co jest istotne dla klienta. Ponadto, prowadzą do pojawianie się kodu, który można pokazać przyszłemu użytkownikowi, jako dowód na rozwój systemu.

Programowanie w oparciu o możliwości aplikacji.

- Jest bardziej szczegółowe, bo poszczególne możliwości mogą być niewielkie, a ich samych dużo.
- Metoda przydatna, gdy aplikacja ma posiadać wiele *niezależnych* możliwości (a przynajmniej niezbyt ścisłe powiązanych).
- Koncentruje się na funkcjonalnościach systemu.
- Pozwala na szybkie wyprodukowanie kodu, który może zostać zademonstrowany klientowi.

Programowanie w oparciu o przypadek użycia.

- Dobra dla aplikacji, gdy aplikacja ma być używana na wiele sposobów. Mamy więc wiele scenariuszy użycia przy ograniczonej liczbie niezależnych możliwości.
- Koncentruje się na użytkowniku - na tym jak będzie używać programu
- Szczególnie dobrze działa ta metoda w projektach składających się z długich i skomplikowanych procesów (np. systemy transakcyjne).
- Kod, który można pokazać klientowi jest generalnie większy niż w poprzednim przypadku - to kompletny proces realizowany przez aplikację.

Scenariusz testowy

Coś co ma sens dla klienta

Sama implementacja fragmentu aplikacji zazwyczaj nie spełni oczekiwania klienta - klient potrzebuje zobaczyć aplikację "w akcji".

Testy - scenariusze testowe

nie muszą być skomplikowane. Ich cel to pokazać, że kod realizuje poprawnie pewną funkcjonalność. Skomplikowane testy sprawdzające zbyt wiele rzeczy jednocześnie powodują, że trudno jest odszukać prawdziwą przyczynę niepowodzenia.

Odporność na błędy

nie należy testować wyłącznie szczęśliwej ścieżki. Warto sprawdzić jak kod reaguje na niestandardowe zachowania użytkownika.

Zmiany

Projekt może podlegać zmianom

nie tylko dlatego, że wymagania się zmieniły, ale także dlatego, że można znaleźć lepsze rozwiązanie problemu.

Paraliż analityczny

ciągnące się dyskusje nad propozycjami rozwiązań blokujące powstanie implementacji. Lepiej wybrać jakieś rozwiązanie i konsekwentnie je stosować, chyba, że szczegółowa analiza mniejszych fragmentów aplikacji pokaże, że nie można go dłużej utrzymać. Nie można blokować zmian w projekcie, jeśli widać, że nie zdaje on egzaminu.

Programowanie wg. kontraktu vs. defensywne

Programowanie wg. kontraktu

Umawiamy się jakie wartości będą zwracać metody i jak będą się zachowywać. Obie strony *ufają sobie wzajemnie* wierząc np., że druga potrafi prawidłowo obsługiwać np. odczyt nieistniejącej wartości (null). Kod jest szybszy i krótszy!

Programowanie defensywne

Jeśli istnieje uzasadnione przypuszczenie, że kod może być używany w sposób nieprawidłowy, to należy posługiwać się weryfikowalnymi wyjątkami - użytkownik naszego kodu, będzie musiał stworzyć kod odpowiedzialny za obsługę takiej sytuacji. Staramy się nie być odpowiedzialni za ewentualne przyszłe problemy z aplikacją. W programowaniu defensywnym najważniejsze jest to, że tu mamy do czynienia z jednostronną decyzją o kształcie interfejsu - decyzja wynika z braku zaufania do użytkownika. Samo używanie wyjątków może być częścią zwykłego kontraktu.

Programowanie wg. kontraktu vs RuntimeException

Wyjątki nieweryfikowalne

nie muszą być obsługiwane po stronie wywołującego nasz kod. Możemy zgłosić taki wyjątek gdy kontrakt jest łamany - niby użytkownikowi naszego kodu wierzymy, ale w przypadku problemów błąd jednak zdarzy się ewidentnie "z jego winy".

Brak zmian po stronie użytkownika

Dodanie kodu zgłaszającego wyjątek nieweryfikowalny nie wymusza zmiany kodu po stronie użytkownika.

Proces projektowania i analizy obiektowej

- 1 Tworzymy listę możliwości - co aplikacja ma robić
- 2 Rysujemy przypadki użycia - określamy procesy realizowane w ramach aplikacji i czynniki zewnętrzne, jakie są związane z aplikacją i jej działaniem
- 3 Dzielimy aplikację na moduły funkcjonalności
- 4 Określamy kolejność opracowywania modułów
- 5 Określamy wymagania dla modułu
- 6 Ustalamy odwzorowanie przypadków użycia na obiekty używane w aplikacji
- 7 Określamy związki pomiędzy obiektami, stosujemy zasady projektowania obiektowego i poznane wzorce projektowe
- 8 Implementujemy i testujemy kod
- 9 Jeśli pozostały jeszcze jakieś niezaimplementowane zachowania, możliwości czy przypadki użycia to idź do punktu 5
- 10 Koniec

Podział problemu a wymagania

Podział problemu

na moduły funkcjonalne jest po naszej stronie. Dotyczy naszego kodu.

Wymagania

to krok bardzo ściśle powiązany z użytkownikiem - ze sposobem w jaki chce używać naszego programu.

A pomiędzy podziałem a wymaganiami

jest **zrozumienie** problemu!!! Jeśli problem nie został zrozumiany to trudno napisać poprawny zestaw wymagań.

Ochrona własnych klas

Dostęp do klas

Użytkownik powinien mieć dostęp tylko do tych klas, które są mu **niezbędne**. Klasy, których klient nie używa można zmieniać do woli. Chronimy jeden fragment kodu przez zmianami w innej części projektu.

Czysty kod czyli kod kodowi nierówny.

Na podstawie: Robert C. Martin, "Clean Code: A Handbook of Agile Software Craftsmanship"

Dlaczego tworzymy zły kod?

Brodzenie

Postęp pracy spowalnia zły kod, w którym brodzimy.

Powody produkowania złego kod:

- Pośpiech, presja terminów
- "Przecież działa"
- Posprząta się później

Efektywność zespołu i szansa na dotrzymanie terminu są odwrotnie proporcjonalne do bałaganu w kodzie.

Dlaczego chcemy czystego kodu?

Czytanie kodu vs pisanie kodu

10:1

Warunkiem łatwego pisania nowego kodu jest łatwe przeczytanie kodu istniejącego.

Nazwy

Intencje

Warto używać nazw, które przedstawiają zamiary programisty.

Dobre nazwy

- nie potrzebują komentarzy
- zdradzają powód swojego istnienia
- nie dezinformują
- nie są nadmiarowe (np. człon "table" dla nazwy tablicy)
- dają się wypowiedzieć (wymówić)
- dają się wyszukać w tekście
- unikają odwzorowania mentalnego na inne nazwy
- dla klas to rzeczowniki, dla metod to czasowniki
- powinny pochodzić ze spójnego leksykonu
- nie są kalamburami
- pochodzą z dziedziny zagadnienia/problemu (np. `WatchdogObserver`).

Funkcje/metody

Rozmiar

Funkcje powinny być, przede wszystkim, małe!

Porady dotyczące pisania funkcji/metod

- bloki instrukcji typu `if`, `while` itd. powinny mieć po jednym wierszu, czyli wywoływać funkcję
- poziom wcięć nie powinien być większy niż 2
- funkcje powinny (dobrze) wykonywać jedną czynność
- instrukcje powinny dotyczyć tego samego poziomu abstrakcji
- funkcja nie powinna posiadać więcej niż 2 argumentów - ideał brak argumentów!
- wynikiem działania powinna być zwracana wartość a nie modyfikacja argumentu
- funkcje, które nie zwracają wyniku (tzw. zdarzenia) zmieniają stan - użytkownik musi być tego świadomy
- należy unikać efektów ubocznych

Funkcje/metody cd.

Porady dotyczące pisania funkcji/metod

- Funkcje nie powinny jednocześnie wykonywać operacji i odpowiadać na pytania.
- Bloki `try-catch` warto przenosić do osobnych funkcji
- warto zadbać o unikanie powtórzeń kodu

Komentarze

Komentarze

Dobre komentarze są rzeczą pomocną i cenną!

Ale...

- bywają niejasne
- stare i opisują inną wersję kodu (rzeczywistości)
- zawsze sygnalizują problemy programisty, który nie potrafił sobie poradzić bez nich
- lepszy jest, po prostu, czysty (czytelny) kod, który komentarza nie wymaga

Użyteczne komentarze

Kiedy warto poświęcić czas na dodanie komentarza?

- Komentarze prawne np. z informacją o prawach autorskich
- Komentarze informacyjne - np. krótka informacja o przeznaczeniu metody
- Wyjaśnianie zamierzenia programisty - komentarz wyjaśnia dlaczego takie rozwiązanie zostało wprowadzone.
- Wyjaśnianie użycia kodu, którego nie można zmienić, aby jego metody miały lepsze nazwy.
- Ostrzeżenia - wyjaśniają konsekwencje wykonania operacji np. duże zapotrzebowanie na RAM czy czas CPU.
- Notatki TODO - czyli, co ma być wykonane w przyszłości.
- Wzmocnienie - podkreślenie, że dana, nieoczywista operacja ma faktycznie sens.
- Javadoc w publicznym API

Złe komentarze

Kiedy tracimy czas pisząc (czytając) komentarze

- Bełkot - komentarz, który powstał tylko jako wypełniacz miejsca
- Powtórka - komentarz dokładnie równoważny komentowanemu kodowi
- Mylący komentarz - nieprecyzyjny, czy wręcz błędny, komentarz
- Komentarze wymagane - tworzone tylko dlatego, aby spełnić pewien wymóg.
- Changelog - komentarz opisujący zmiany w kolejnych wersjach
- Szum informacyjny - opis rzeczy oczywistych
- Przerażający szum - niepotrzebne komentarze tworzone do wielu elementów kodu metodą skopiuj-wklej.
- Znacznik pozycji - znacznik miejsca w kodzie.
- Dopiski - informacje o wprowadzanych zmianach.
- Kod wyłączony komentarzem
- Komentarz nielokalny - odnosi się do innej części systemu
- Nieoczywiste połączenie - nie wiadomo co dokładnie jest komentowane

Formatowanie kodu

Formatowanie kodu źródłowego

Kod powinien być ładnie sformatowany, bo jest to ważne. Ważne, bo poprawia komunikację.

Przedkładamy "zapewnienie komunikacji" nad "zapewnieniem działania".

Formatowanie

- pionowe - pliki nie powinny być zbyt duże, układ typu artykułu w gazecie. Odstępy pomiędzy segmentami poprawiają czytelność. Deklaracje zmiennych - możliwe blisko miejsca użycia. Pola klasy - na górze kodu. Funkcja wywołująca powyżej wywoływanej. Jeśli fragmenty kodu są powiązane - powinny być umieszczone blisko siebie.
- poziome - dawniej obowiązywało ograniczenie związane z możliwościami terminali (80 znaków na wiersz). Preferowane są wiersze krótkie. Nie należy doprowadzać do konieczności przewijania tekstu w edytorze w prawo!

Ukrywanie danych

Dane prywatne

Dane prywatne a przez akcesory dostęp do nich publiczny. Gdzie tu sens?

Ukrywanie implementacji

Polega na tworzeniu abstrakcji! Dzięki abstrakcji użytkownik nie wie w jakiej formie przechowywane są dane.

Prawo Demeter a struktury danych

Wraki pociągów

`directory.getPath().getDisk().getAvailableBlocks()`

taki kod nazywany jest wrakiem pociągu, bo wygląda jak katastrofa kolejowa. Jest to naruszenie prawa Demeter!

A czy tu łamiemy prawo Demeter?

`directory.path.disk.getAvailableBlocks`

Zamiast obiektów używane są struktury danych (klasy bez metod), naruszenie więc nie ma, bo struktury danych w sposób naturalny ujawniają implementację.

Obsługa błędów

Kiedy obsługa błędów to зло?

Obsługa błędów, sama w sobie, jest rzeczą dobrą, ale jeśli jest zrealizowana w taki sposób, że cierpi na tym logika kodu staje się rzeczą niewłaściwą.

Jak bronić się przed tym złem?

- Używamy wyjątków a nie kodów powrotu
- Bez względu na sposób zakończenia bloku `try` program musi pozostać w stanie spójnym.
- Metoda wywoływana może zostać zmodyfikowana tak, aby nie używała wyjątków - przywraca to normalny przebieg programu.
- Wyjątek powinien zawierać dodatkowe informacje, np. o przeznaczeniu nieudanej operacji.

Null

Nie zwracać null

Metody zwracające null powodują problemy i przysparzają dodatkowej pracy w celu wykrycia tej sytuacji. Zamiast zwrócić null lepiej użyć wyjątku. Możliwe jest także zwrócenie obiektu specjalnego przypadku.

Nie przekazywać null

Wywołanie metody z argumentem w postaci null jest zdecydowanie odradzane. Nie ma eleganckiego rozwiązania tego problemu. Oczywiście, wywoływana metoda może mieć kod wykrywający takie wywołanie i zgłaszać `InvalidArgumentException`, ale to pociąga za sobą konieczność ujęcia jej w blok `try-catch`.

Organizacja kodu klasy

W kodzie źródłowym klasy umieszczamy kolejno:

- publiczne stałe statyczne
- publiczne stałe prywatne
- prywatne pola
- metody publiczne
- metody prywatne

Kolejność metod od metod wyższego poziomu (wywołują inne metody) do niskopoziomowych (to one są przez inne metody wywoływane).

Dehermetyzacja

Testy

Testy mogą wymusić ograniczenie hermetyzacji, choć takiego rozwiązania należy użyć w ostateczności.

Rozmiar klasy

Jak mierzyć rozmiar klasy?

Zamiast liczby linii kodu zliczane są *odpowiedzialności*.

Wykrywanie zbyt wielu odpowiedzialności

Czy można utworzyć spójną nazwę klasy?

Czy w nazwie występują słowa typu "Manager", "Super"?

Czy krótki opis działania klasy wymaga użycia słów "oraz", "ale", "jeżeli"?

Zasada pojedynczej odpowiedzialności

SRP

Klasy powinny mieć **jedną** odpowiedzialność a przez to jeden powód do zmiany.

Identyfikowanie odpowiedzialności (powodów do zmiany klasy) pozwala na stworzenie lepszego kodu i zmniejszanie kodu klas.

Spójność związana z użyciem pól

Pola niestatyczne

Zalecane są klasy o niewielkiej liczbie pól niestatycznych.

Maksymalna spójność klasy

Maksymalna spójność klasy uzyskiwana jest, gdy każda metoda używa wszystkich pól niestatycznych.

Maksymalizacja spójności klasy

Zaleca się tworzenie klas o wysokiej spójności.

Długie metody z wieloma zmiennymi lokalnymi

Długie metody z wieloma zmiennymi lokalnymi

Nie lubimy długich metod. Chcemy podzielić je na mniejsze. Musimy tylko zadbać o przekazanie pomiędzy metodami odpowiednich informacji.

Długie listy parametów

Nie lubimy metod z długimi listami parametrów. Aby ich uniknąć tworzymy niestatyczne pola klasy. W ten sposób zmienne lokalne zamieniają się w pola. Metody komunikują się z użyciem tych pól.

Spójność

Z nowo powstałych pól korzysta część metod. Nie lubimy, gdy z niestatycznych pól korzysta tylko część metod klasy, bo obniża to spójność. Aby poprawić spójność tworzymy nową klasę.

Refaktoryzacja

Utrzymanie kodu

Łatwiej napisać niż utrzymać

Koszty utrzymania (konserwacji) kodu są wyższe od kosztów jego napisania. Ponad 80% nakładów ponoszonych jest na działania nienaprawcze (czyli nie chodzi o usuwanie błędów).

Prawa Lehmana

W 1969 roku Meira Lehman formułuje pojęcia konserwacji oprogramowania i ewolucji systemów. Wynik badań: utrzymanie to ewolucja systemu. Programy rosną i stają się coraz bardziej skomplikowane. Aby przeciwdziałać temu trendowi należy prowadzić ciągłą restrukturyzację, której celem jest uproszczenie kodu.

Refaktoryzacja

Refaktoryzacja kodu

proces restrukturyzacji istniejącego kodu źródłowego, bez zmiany widocznego na zewnątrz zachowania ^a.

^a Jest problem z określeniem co to tak naprawdę oznacza!

Refaktoryzacja kodu

zmiana wewnętrznej struktury programu aby uczynić go łatwiejszym do **zrozumienia** i tańszym w modyfikacji, ale bez zmiany obserwowanego zachowania
(Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts-Refactoring - Improving the Design of Existing Code)

Refaktoryzacja

poprawia niefunkcjonalne elementy programu, kod jest łatwiejszy do przeczytania, zmniejsza złożoność, ułatwia utrzymanie i rozszerzanie.

Uwaga: optymalizacja to nie refaktoryzacja!

Testy; zatracanie pierwotnego projektu

Refaktoryzacja

nie generuje nowych testów. Testy pojawiają się z okazji dodawania nowych funkcjonalności.

Dlaczego program zatracza pierwotny kształt?

Ze względu na konieczność szybkiego wprowadzania zmian i brak zrozumienia oryginalnego projektu. Refaktoryzacja sprząta kod np. poprzez przenoszenie fragmentów, które można umieścić w lepszym miejscu. Bez okresowej refaktoryzacji aplikacja zatracza pierwotny kształt - tym samym jego utrzymianie (przywrócenie) z czasem staje się coraz bardziej skomplikowane.

Czytelność kodu

Czytelność a efektywność

Pytanie co jest ważniejsze czas procesora, który musi wykonać kilka operacji więcej czy czas programisty, który będzie w przyszłości musiał zmodyfikować kod. Często tym programistą jesteśmy sami...

Czytelność a struktura projektu

Czytelniejszy kod ułatwia poprawę struktury projektu, ułatwia zrozumienie. To z kolei pozwala szybciej wykrywać błędy w kodzie.

Kiedy dokonywać refaktoryzacji?

Zasada trzech

Jesli fragment kodu powtarza się po raz trzeci — czas na refaktoryzację!

Gdy dodawana jest nowa właściwość

Refaktoryzacja ułatwia dodanie nowej własności, poprawia zrozumienie kodu, który już istnieje.

Gdy trzeba naprawić błąd

Z jednej strony poprawiamy możliwość zrozumienia kodu, z drugiej, skoro pojawił się błąd to oznacza, że kod nie był wystarczająco jasny by błąd wcześniej dostrzec.

Z okazji okresowego przeglądu kodu

Wiedza jest przekazywana w zespole, osoby bardziej doświadczone mogą dzielić się doświadczeniem.

Kiedy nie dokonywać refaktoryzacji?

Bazy danych

Kod jest powiązany ze schematem bazy - trudno go zmienić, bo migracja danych może być trudna.

Zmiany *upublicznionego* interfejsu

Nie cały kod zależny od klasy może być dostępny w celu zmiany np. nazwy metod. Najprawdopodobniej w okresie przejściowym konieczne będzie wspieranie dwóch interfejsów: oryginalnego i nowego. Stąd wynika porada: upublicznić możliwie jak najmniej.

Inne niesprzyjające okoliczności:

Kod nadaje się do przepisania od nowa.

Kod nie działa poprawnie.

Brak czasu (zblża się deadline).

Refaktoryzacja a wydajność

Wydajność

faktycznie może być gorsza, ale z drugiej strony, dzięki refaktoryzacji kod jest wygodniejszy do "tuningu". Okazuje się, że najważniejszy jest dobry kod, który można łatwo modyfikować aż do uzyskania wymaganej wydajności aplikacji.

Zbyt wczesna optymalizacja wydajności

zwalnia proces rozwoju oprogramowania. Często chodzi o lokalne poprawki, które nie uwzględniają specyfiki działania całego programu. Okazuje się, że przy jednakowej optymalizacji całości kodu, około 90% prac wykonywana jest niepotrzebnie, bo czas wykonywania tego kodu jest znacznie mniejszy w porównaniu z czasem pracy całej aplikacji.

Obserwacja:

Większość czasu pracy aplikacji realizuje zdecydowana mniejszość kodu. Kod optymalizować należy po analizie wydajności aplikacji za pomocą profilera. Kod zrefaktoryzowany łatwiej zoptymalizować, bo łatwo go zrozumieć i poprawić.

Zapachy kodu i heurystyki

Zapachy kodu (code smell)

Zapachy kodu

pewne cechy kodu źródłowego sugerujące złą implementację i oznaczające konieczność refaktoryzacji

Zapachy kodu

Nie ukrywajmy, zazwyczaj nie chodzi tu o wykwintne zapachy - bardzo często mamy do czynienia ze smrodem rozchodzącym się od złego kodu.

Zapachy kodu i heurystyki - komentarze

Niewłaściwe informacje Metadane nie powinny występować w komentarzach.

Komentarzem nie powinny być dane, które przechowywane są w dedykowanych systemach typu system kontroli wersji, śledzenia błędów.

Przestarzałe komentarze to już było. Generalnie, najlepiej nie pisać komentarzy, które mogą się przedawnić.

Nadmiarowe komentarze nie opisywać tego, co jest absolutnie jasne

Źle napisane komentarze zły dobór słów, chaos

Zakomentowany kod to już było

Zapachy kodu i heurystyki - środowisko

Budowanie wymaga więcej niż jednego kroku Zbudowanie kodu wynikowego powinno być możliwe za pomocą jednej operacji

Testy wymagają więcej niż jednego kroku Wszystkie testy powinno dać się uruchomić za pomocą jednego polecenia

Zapachy kodu i heurystyki - funkcje

Nadmiar argumentów Im więcej argumentów tym gorzej. Sytuacja idealna to funkcja bezparametrowa.

Argumenty wyjściowe są sprzeczne z intuicją. Nie stosować!

Argumenty znacznikowe to argumenty typu logicznego. Wyraźnie wskazują na to, że funkcja wykonuje więcej niż jedną operację. Eliminować!

Martwe funkcje to funkcje, których nikt nie wywołuje. Usuwać!

Zapachy kodu i heurystyki - ogólne 1/7

Wiele języków w jednym pliku źródłowym W jednym pliku mamy kod Java, HTML, YAML... Sytuacja idealna: jeden plik źródłowy jeden język. Realia: im mniej języków w jednym pliku tym lepiej.

Oczywiste działanie jest nieimplementowane Kod powinien działać zgodnie z "zasadą najmniejszego zaskoczenia".

Niewłaściwe działanie w warunkach granicznych kod powinien działać w oparciu o wykonane testy a nie intuicję autora. Testy powinny obejmować także warunki skrajne.

Zdjęte zabezpieczenia np. blokowanie ostrzeżeń kompilatora czy wyłączenie części testów. Nie używać!

Powtórzenia są złe, więc DRY, albo wg. Kenta Becka "raz i tylko raz". Zlokalizować i zniszczyć!

Kod na nieodpowiednim poziomie abstrakcji Koncepcje wysokiego poziomu powinny trafiać do klas bazowych a niskopoziomowe do klas pochodnych, nie należy ich mieszać.

Zapachy kodu i heurystyki - ogólne 2/7

Klasy bazowe zależne od swoich klas pochodnych Klasy bazowne nie powinny wiedzieć o potomnych.

Za dużo informacji Ukrywajmy nasze dane, twórzmy małe interfejsy.

Martwy kod to kod, który nigdy się nie wykonuje. Nadaje się tylko do usunięcia.

Separacja pionowa Definicja zmiennych czy funkcji blisko miejsca użycia. Zmienne lokalne powinny mieć najmniejszy zakres pionowy. Metody prywatne poniżej pierwszego ich użycia.

Niespójność czyli zaskoczenie. Należy się zdecydować jak nazywana jest dana czynność, do czego służy zmienna o danej nazwie i **być konsekwentnym**.

Zaciemnianie kod, który nie ma żadnego znaczenia dla pracy programu powinien zostać usunięty.

Sztuczne sprzężenia to moduły połączone ze sobą choć nie są od siebie zależne. Trzeba myśleć gdzie umieszcza się kod.

Zapachy kodu i heurystyki - ogólne 3/7

Zazdrość o kod (feature envy) Metoda zbyt intensywnie korzysta z danych należących do *innej* klasy. Należy rozważyć migrację metody.

Argumenty wybierające to argumenty zmieniające zachowanie funkcji np. wartość liczbową, wyliczenie, wartość logiczna. Metodę należy podzielić.

Zaciemnianie intencji metoda napisana tak, że nie wiadomo co autor miał na myśli.

Źle rozmieszczona odpowiedzialność Kod (metody, stałej) powinien być tam, gdzie zaskakuje to najmniej. Np. stała E czy PI w Math a nie np. w List.

Niewłaściwe metody statyczne to metody, które blokują możliwość użycia polimorfizmu, bo są zadeklarowane jako statyczne.

Zapachy kodu i heurystyki - ogólne 4/7

Użycie opisowych **zmiennych** to dobra rzecz. Warto dzielić pracę na małe kroki, których wyniki trafiają do zmiennych o dobrych nazwach.

Nazwy funkcji powinny informować o tym, co realizują . Jeśli ustalenie co robi funkcja wymaga szukania tego w dokumentacji, to znaczy, że jej nazwa jest zła.

Zrozumienie algorytmu - Pisząc funkcję nie wystarczy doprowadzić ją do stanu, w którym "działa" tj. przechodzi testy. Pisząc funkcję trzeba być pewnym, że rozumie się jak działa algorytm, który ma ona realizować. Test: liczba instrukcji **if** - duża ich liczba sugeruje, że autor nie rozumiał algorytmu i w ten sposób ratował kod.

Zamiana zależności logicznych na fizyczne. Zależność logiczna to założenie. Zależność fizyczna to wywołanie metody i test jej wyniku. Należy sprawdzać a nie coś zakładać.

Zapachy kodu i heurystyki - ogólne 5/7

Zastosowanie polimorfizmu zamiast instrukcji if-else lub switch-case. Alternatywą dla switch jest polimorfizm.

Wykorzystanie standardowych konwencji ułatwia czytanie kodu, bo wiadomo gdzie się można danego elementu (np. stałych) spodziewać.

Zamiana magicznych liczb na stałe nazwane. Tak po prostu. Uwagę można rozszerzyć również na ciągi znaków ("Ala ma kota" umieszczone gdzieś w kodzie co oznacza i czym jest?)

Precyzja jest ważna w kodowaniu. Gdy stajemy przed alternatywnymi rozwiązaniami należy precyzyjnie wybrać lepsze.

Struktura przed konwencją. Nawet najlepsze nazwy funkcji utworzone w zgodzie z pewną konwencją i użyte w konstrukcji switch-case nie będą lepsze od rozwiązania używającego klas i podklas.

Zapachy kodu i heurystyki - ogólne 6/7

Hermetyzacja warunków. Warunki należy ukryć w funkcji/metodzie, dzięki temu pętle czy operacje warunkowe będą łatwiejsze do analizy.

Unikanie warunków negatywnych, są trudniejsze do zrozumienia od warunków pozytywnych.

Funkcje powinny wykonywać jedną operację. Jeśli wykonują wiele, to trzeba je podzielić.

Ukryte sprzężenia czasowe to funkcje, które muszą być wykonane w określonej kolejności aby doprowadzić do poprawnego wyniku, ale ich postać na tą kolejność nie wskazuje ani jej nie wymusza.

Unikanie dowolnych działań. Chodzi tu o strukturę kodu, która nie ma żadnego uzasadnienia. Kod wygląda jak powstały na skutek jakiś niezrozumiałych, losowych decyzji.

Zapachy kodu i heurystyki - ogólne 7/7

Hermetyzacja warunków granicznych. Warunki graniczne powinny zostać hermetyzowane w stosownych zmiennych.

Funkcje powinny zagłębiać się na jeden poziom abstrakcji. Funkcja powinna wywoływać wyłącznie funkcje o jeden poziom abstrakcji niżej niż ona sama.

Przechowywanie danych konfigurowalnych na wysokim poziomie. Dane oczekiwane na wysokim poziomie powinny na nim być przechowywane i przekazywane funkcjom niższego poziomu w postaci argumentów. Nigdy odwrotnie (funkcja wysokiego poziomu szuka danych na poziomie niskim).

Unikanie nawigacji przechodnich - prawo Demeter lub inaczej pisanie "wstydliwego kodu". Moduł ma znać tylko bezpośrednich współpracowników.

Zapachy kodu i heurystyki - Java

Unikanie długich list importu przez użycie znaków wieloznacznych Import konkretnych klas tworzy silną zależność, bo importowana klasa musi istnieć. Użycie importu ze znakiem wieloznaczny m zaledwia nie wytwarza, ponadto skraca listę importów.

Nie dziedziczymy stałych. Chodzi o absurdalne użycie dziedziczenia tylko po to aby nazwy stałej nie trzeba było poprzedzać nazwą klasy, w której się znajduje.

Stałe kontra typy wyliczeniowe. W Java `enum` wprowadzono w wersji 1.5 w 2004 roku. Warto używać typów wyliczeniowych zamiast wymyślonych przez nas stałych liczbowych.

Zapachy kodu i heurystyki - Nazwy

Wybór opisowych nazw. Nazwy decydują o czytelności kodu.

Wybór nazw na odpowiednich poziomach abstrakcji. Nazwy powinny pasować do poziomu abstrakcji.

Korzystanie ze standardowej nomenklatury tam, gdzie jest to możliwe. Nazwy zawierające słowa np. Decorator, Listener, Observer są łatwiejsze do zrozumienia, bo używają słów, które coś znaczą.

Jednoznaczne nazwy ułatwiają życie. Jeśli dzięki nazwie nie ma wątpliwości co funkcja robi, to szybciej analizuje się kod.

Użycie długich nazw dla długich zakresów. Mały zakres (np. kilka linijek pętli `for`) lepiej czyta się z krótkimi nazwami. Długie pozwalają lepiej śledzić kod na większym obszarze.

Unikanie kodowania. Nie należy w nazwie ukrywać informacji np. o type.

Nazwy powinny opisywać efekty uboczne. Nie należy ukrywać faktu, że funkcja coś robi.

Zapachy kodu i heurystyki - Testy

Niewystarczające testy. Jak długo testy nie uwzględniają wszystkich możliwości, jest ich za mało.

Użycie narzędzi kontroli pokrycia. Użyte wskażą co nie jest testowane.

Nie pomijaj prostych testów.

Ignorowany test jest wskazaniem niejednoznaczności. Niejasne wymagania to niepewność jaki ma być wynik.

Warunki graniczne. Często środkowa część działa, a problemy są na krańcach.

Dokładne testowanie pobliskich błędów. "Nieszczęścia chodzą parami". Jeśli metoda nie przechodzi jednego testu, to warto ją dokładniej zbadać, bo jest spora szansa na to, że nie jest to jedyny błąd.

Wzorce błędów wiele ujawniają. Kompletne przypadki testowe pozwalają ujawnić pewne wzorce (prawidłowości) w pojawianiu się błędów.

Wzorce pokrycia testami wiele ujawniają. Warto sprawdzić co (nie) było testowane - może to pomóc w odnalezieniu błędu w kodzie.

Testy powinny być szybkie. Wolne testy często nie będą wykonywane.

O testowaniu więcej w przyszłości...

Zapachy kodu i heurystyki - inne

długa metoda (long method) - trudno zrozumieć (czy choćby przeczytać) taki kod
duża klasa (large class) - klasa ma za dużo odpowiedzialności - było SRP!

długa lista parametrów (long parameter list) - dziedzictwo programowania proceduralnego, za dużo parametrów utrudnia zrozumienie i testy.

shotgun surgery - gdy jedna zmiana to zmiana małych rzeczy w wielu, wielu klasach - łatwo coś wtedy przeoczyć

zbyt mała intymność (inappropriate intimacy) - naruszono zasady hermetyzacji, bo jedne klasy w swym działaniu zależą od **implementacji** innych (wiedzą o nich za dużo).

Strona, którą warto odwiedzić

<http://www.refactoring.com/catalog/>

Brak dbania o kod - a czym to grozi?

Może być tak, że koszty rozbudowy oprogramowania o nowe funkcjonalności oraz naprawiania błędów staną się porównywalne (a nawet większe) od przychodu. Oznacza to często śmierć projektu.

Pisząc zły kod zaciągamy pewien kredyt. Mamy do spłacenia pewien dług, **dług technologiczny (tech debt)**. Kolejne zmiany wprowadzane na szybko, bez refaktoryzacji, to kolejne zobowiązania. Dług rośnie. Każda zła zmiana kodu utrudni kolejną. Odsetki od dłużu rosną. Naprawa będzie coraz dłuższa i droższa (nikt nie zrobi jej za darmo).

Co jest powodem powstania dłużu technologicznego?

- Lenistwo (no przecież działa!)
- Pośpiech
- Brak świadomości skutków złego kodu
- Złe zarządzanie projektem
- Cięcia finansowe (brak refaktoryzacji, testów automatycznych)
- Wybór rozwiązania, które utraciło wsparcie

Dbanie o kod - refaktoryzacja

Co robimy refaktoryzując?

- usuwamy duplikacje
- upraszczamy kod
- wprowadzamy/zmieniamy wzorce
- wykonujemy często (najlepiej ciągle) niewielkie zmiany, jedna po drugiej

Niewielkie zmiany pozwalają łatwiej wykryć błąd. Bazą do refaktoryzacji są testy. Trzeba przygotować testy automatyczne dla funkcjonalności, która ma być refaktoryzowana.

Antywzorce

Co to jest antywzorzec?

Antywzorzec?

Skoro istnieją wzorce, które są przepisem na sukces, równowaga wymaga, aby istniały przepisy na katastrofę... a przynajmniej na uzyskanie złego rozwiązania. Innymi słowy "antywzorzec" to "kiepski pomysł".

Po co o nic mówimy?

aby ich unikać...

Opis antywzorca

Dobry opis zawiera informację dlaczego dany antywzorzec wygląda atrakcyjnie.

Gdzie znaleźć antywzorzec?

Antywzorce występują w:

- kodzie/budowie aplikacji
- architekturze aplikacji
- procesie zarządzania projektem

Elementy antywzorca

Elementy antywzorca:

- problematyczne rozwiązanie, to rozwiązanie, które ma negatywne konsekwencje
- rozwiązanie zrefaktoryzowane, to metoda dzięki której problem wywołany przez pierwszy element może zostać rozwiązywany i przetworzony w formę bardziej korzystną.

Antywzorce kodu

Blob (The blob)

Inne nazwy

Winnebago lub Boska klasa (The God Class) lub Boski obiekt

Problem

Pojedynczy obiekt realizuje większość obowiązków. Pozostałe zajmują się prostymi operacjami czy przechowywaniem informacji.

Rozwiązanie

Refaktoryzacja projektu w celu zapewnienia bardziej równomiernego rozkładu obowiązków, oraz izolacji zmian. Chodzi o zmniejszenie złożoności obiektu Blob przez wytworzenie obiektów, które zyskają dodatkowe zdolności.

Blob, choć może być zrealizowany obiektowo, reprezentuje programowanie w stylu proceduralnym. Przetwarzanie i dane są odseparowane. Blob rośnie np., gdy w wyniku kolejnych iteracji, pewien moduł obrasta w kolejne funkcjonalności, bo nie jest wykonywany podział obowiązków. Moduł ten staje się dominujący.

Blob (The blob)

Symptomy i konsekwencje:

- Pojedyncza klasa posiadająca dużą liczbę pól i/lub metod.
- Niepowiązane ze sobą atrybuty / metody, brak spójności
- Pojedyncza klasa z powiązanymi prostymi klasami do reprezentowania danych
- Brak projektowania obiektowego
- Główna pętla programu wewnętrz klasy Blob powiązana z pasywnymi obiektami innych klas
- Migracja starego projektu bez refaktoryzacji do projektu zorientowanego obiektowo
- Klasa, którą trudno przetestować i użyć ponownie
- Klasa wymagająca dużo zasobów nawet dla wykonania prostych operacji

Blob (The blob)

Typowe przyczyny:

- Brak architektury obiektowej wynikający np. z braku zrozumienia jej zasad
- Brak (jakiejkolwiek) architektury, brak definicji komponentów systemu, interakcji.
- Brak egzekwowania architektury - nawet jeśli została poprawnie zaprojektowania.
- W kolejnych iteracjach klasy zyskują kolejne funkcjonalności. Zamiast tworzyć nowe klasy, funkcjonalności dodawane są do istniejących.
- "Katastrofa wg. ustaleń" - np. wymagania dyktują rozwiązanie proceduralne, którego skutkiem będzie Blob.

Strumień lawy (Lava flow)

Inne nazwy

Martwy kod

Problem

"Nie wiem co ten kod robi, napisał go ktoś zanim tu przyszedłem do pracy". Okazuje się, że bardzo dużo kodu nie jest zrozumiałą dla osób, które aktualnie pracują w firmie. Kod pozostaje w systemie (choć, często nic nie robi), ale obecni pracownicy nie wiedzą, czy można go usunąć, bo nie znają konsekwencji takiego działania. Ten antywzorzec pojawia się, gdy wątpliwej jakości (np. eksperymentalny, prototypowy) kod jest włączany do produkcyjnej wersji systemu; to źle, bo nikt nie wie w jakiej fazie poprzednicy pozostawili taki moduł.

Rozwiązanie

Unikać zmian architektury podczas aktywnego rozwoju aplikacji. Przed tworzeniem kodu należy poczekać na zdefiniowanie architektury. Interfejsy muszą być stabilne i jasno udokumentowane. Używać systemów kontroli kodu źródłowego (Source Code Control System).

Strumień lawy (Lava flow)

Symptomy i konsekwencje

- Częste, zupełnie nieuzasadnione zmienne i fragmenty kodu.
- Nieudokumentowane, złożone i wyglądające na ważne segmenty kodu, które nie są jasno powiązane z architekturą systemu
- Bardzo luźna architektura systemu
- Bloki wykomentowanego kodu, bez wyjaśnienia czy dokumentacji
- Dużo kodu przeznaczonego do zastąpienia
- Nieużywany, martwy kod.
- Nieużywane, przestarzałe interfejsy znajdujące się w plikach nagłówkowych
- Jeśli kod nie zostanie poprawiony, lava będzie płynąć dalej i rozprzestrzeniać się, gdy dojdzie do ponownego użycia modułu, w którym problem występuje.
- Możliwy jest wzrost wykładniczy
- Przygotowanie dokumentacji szybko staje się niewykonalne. Szanse na wprowadzenie ulepszeń także maleją, bo zrozumienie kodu staje się coraz mniejsze.

Strumień lawy (Lava flow)

Typowe przyczyny:

- Kod R&D podniesiony do rangi produkcyjnego się bez zastanowienia nad zarządzaniem konfiguracją.
- Realizacja wielu próbnych podejść w celu implementacji funkcjonalności i niekontrolowana dystrybucja niedokończonego kodu.
- Kod napisany przez jednego programistę (samotny wilk).
- Brak zarządzania konfiguracją
- Brak architektury
- Cele projektu oprogramowania są niejasne lub wielokrotnie zmieniane. Brak czasu powoduje przełożenie spraw dokumentacji i refaktoryzacji na czas nieokreślony.
- Blizny architektoniczne - pewne rozwiązania są implementowane, ale, gdy okazują się źle działać rzadko są usuwane (są rozproszone w wielu plikach) pozostając w systemie.

Dekompozycja funkcjonalna (Functional Decomposition)

Inne nazwy

No Object-Oriented lub No OO

Problem

Programiści, którzy dobrze programują funkcyjnie są zmuszeni do napisania aplikacji w języku obiektowym tworzą kod, który przypomina język strukturalny ubrany w strukturę klas. Mają tendencję do tworzenia osobnych klas dla każdej z metod. W efekcie powstaje kod ignorujący ideę programowania obiektowego i przy okazji jest bardzo skomplikowany.

Rozwiązanie

Szkolenia z programowania zorientowanego obiektowo albo zatrudnienie odpowiednich programistów.

Dekompozycja funkcjonalna (Functional Decomposition)

Symptomy i konsekwencje

- Nazwy klas przypominają nazwy funkcji.
- Klasy z pojedynczą metodą.
- Architektura ignoruje aspekt programowania obiektowego
- Brak użycia dziedziczenia i polimorfizmu
- Jeśli rozwiązanie zadziałało może okazać się niezwykle drogie w utrzymaniu
- Bez szans na udokumentowanie (a nawet wyjaśnienie) jak to działa.
- Model klas nie ma sensu
- Brak nadziei na ponowne użycie kodu
- Frustracja testerów.

Dekompozycja funkcjonalna (Functional Decomposition)

Typowe przyczyny:

- Brak zrozumienia zasad programowania obiektowego
- Brak nadzoru. Nawet jeśli architektura została zaprojektowana potrzebny jest nadzór, aby programiści jej przestrzegali.
- "Katastrofa wg. ustaleń" tj. problem na poziomie specyfikacji.

Rady:

- Potrzebny jest model projektowy, który zawiera elementy istniejącego systemu. Model powinien uzasadniać istnienie modułów.
- Dla klas, które nie wpisują się model:
 - Jeśli klasa ma pojedynczą metodę to powinna zostać dodana do klasy, która z tej metody korzysta.
 - Można połączyć kilka klas w jedną, która jest zgodna z celem projektu.

Duchy (Poltergeists)

Inne nazwy

Gypsy, Proliferation of Classes, and Big Dolt Controller Class

Problem

W kodzie istnieją klasy, które mają bardzo ograniczoną odpowiedzialność i nie odgrywają żadnej istotnej roli w systemie. Zaśmiecają one projekt, tworzą niepotrzebne abstrakcje. Są one nadmiernie złożone przez co trudne do zrozumienia i utrzymania. Czas ich życia jest dość krótki - są używane do wywołania metod innych klas. W sumie są niepotrzebne, nieefektywne i ich użycie to marnotrawstwo zasobów. Utrudniają prawidłowe projektowanie.

Rozwiążanie

Usunięcie klas stanowiących problem i uzupełnienie brakującego kodu z uwzględnieniem żądanej architektury.

Duchy (Poltergeists)

Symptomy i konsekwencje

- Nadmiarowe ścieżki nawigacji
- Klasy bezstanowe
- Tymczasowe, krótkotrwałe obiekty
- Klasy wykonujące jedną operację, istniejące tylko do wywołania innych przez chwilowe asocjacje
- Klasy o nazwach związanych z kontrolą.

Duchy (Poltergeists)

Typowe przyczyny:

- Brak architektury zorientowanej obiektowo
- Niewłaściwe narzędzie do pracy. Programowanie zorientowane obiektowo nie musi być zawsze dobrym rozwiązaniem.
- Ponownie "Katastrofa wg. ustaleń"

Złoty młotek (Golden Hammer)

Inne nazwy

Głowa w piasku (Head-in-the sand)

Problem

Zespół przyzwyczaja się do pewnego rozwiązania np. pewnej biblioteki czy implementacji bazy danych (ta rzecz to Złoty Młotek). Każde następne przedsięwzięcie jest postrzegane jako rozwiązywalne tylko za pomocą naszego Złotego Młotka. Ogranicza się wysiłki w celu znalezienia innego rozwiązania/technologii. Złoty Młotek używany jest na siłę nawet jeśli prowadzi to do nieprawidłowości.

Rozwiązanie

Eksploracja nowych technologii, konferencje branżowe, szkolenia. Generalnie, rozwój zawodowy programistów. Ponadto, strategia rozwoju umożliwiająca migrację do innych technologii. Odpowiedni podział systemu/granice. Interfejsy pozwalające na izolowanie zastrzeżonych rozwiązań w modułach, które mogą zostać wymienione na inne.

Złoty młotek

Symptomy i konsekwencje

- Identyczne narzędzia używane w różnych koncepcyjnie produktach
- Rozwiązania odznaczają się gorszą skalowalnością i wydajnością niż w przypadku rozwiązań konkurencyjnych.
- Promowane są wymagania zgodne z tymi, które dobrze współpracują ze Złotym Młotkiem. Inne są spychane na drugi plan.
- Brak wiedzy i doświadczenia deweloperów z innymi technologiami niż Złoty Młotek.
- Istniejące produkty decydują o projekcie i architekturze systemu.
- Bazowanie na produktach lub technologii określonego dostawcy.

Złoty młotek

Typowe przyczyny:

- Rozwiążanie oferowane przez Złotym Młotek kilkukrotnie się sprawdziło.
- Duże inwestycje w Złoty Młotek
- Grupa odizolowana od konkurencji
- Bazowanie na rozwiązańach zastrzeżonych (własnościowych), które nie są łatwo dostępne w innych produktach.

Spaghetti Code

Problem

Istnieje od początków programowania. Najbardziej uwidacznia się w językach, które nie są zorientowane obiektowo. W tym anty wzorcu chodzi o kod, w którym brak struktury; kod tak zły, że sam autor się w nim gubi. W przypadku użycia języka obiektowego będzie się obserwować mało obiektów i wielkie metody, które wywołują pojedynczy, wieloetapowy przebieg programu. Wywoływanie metod są przewidywalne, bo pomiędzy obiektami brak interakcji. Kod jest trudny w rozbudowie i utrzymaniu. Nie ma szans na ponowne użycie.

Rozwiązanie

Refaktoryzacja kodu czyli inwestowanie w lepszy kod. Najlepszym sposobem na unikanie kodu typu Spaghetti jest profilaktyka: przed implementacją przemyślany plan działania. Jeśli kod już osiągnął stan kodu Spaghetti, a nie ma szans na jego wymianę, z okazji kolejnych zmian dokonywać stopniowej refaktoryzacji do kodu łatwiejszego w utrzymaniu.

Spaghetti Code

Symptomy i konsekwencje

- Szanse na ponowne użycie kodu są marne
- Obiekt ustala kolejność operacji a nie użytkownicy obiektu
- Brak relacji pomiędzy obiektami
- Metody bez parametrów używające zmiennych globalnych lub ich odpowiedników.
- Sposób użycia obiektów bardzo przewidywalny
- Kod trudny do ponownego użycia, jeśli już to raczej poprzez jego skłonowanie
- Brak dziedziczenia, polimorfizmu. Wszystkie potencjalne korzyści z użycia języka obiektowego utracone.
- Zyski maleją
- Koszt utrzymania kodu jest większy od wytworzenia go od nowa

Spaghetti Code

Typowe przyczyny:

- Brak doświadczenia w programowaniu obiektowym.
- Nieskuteczne przeglądy kodu
- Brak projektu przed implementacją
- Wynik pracy programistów pracujący w izolacji

Programowanie typu skopuj-wklej albo wytnij-wklej (Cut-and-Paste Programming)

Inne nazwy

Clipboard Coding, Software Cloning, Software Propagation

Problem

Zdegenerowana forma idei ponownego użycia kodu. Zamiast pisania kodu od podstaw, istniejące, stworzone przez kogoś innego, rozwiązanie jest modyfikowane. Głównym problemem jest nadużywanie tej idei.

W projekcie uczestniczy wielu programistów, którzy tworzą swoje moduły modyfikując istniejący kod. W efekcie pojawiają się kolejne duplikaty oryginalnego kodu.

Na krótką metę rozwiązań kopuj-wklej jest ciekawe, bo pozwala na szybką implementację nowych funkcjonalności.

Programowanie typu skopuj-wklej albo wytnij-wklej (Cut-and-Paste Programming)

Rozwiązanie

Modyfikacja kodu w celu podniesienia znaczenia ponownego użycia kodu wg. modelu czarnej skrzynki.

Eksploracja kodu w celu wykrycia powtarzających się segmentów kodu.

Refaktoryzacja kodu do bibliotek nadających się do ponownego użycia.

Zastosowanie zarządzania konfiguracją w celu zapobieżenia używania techniki kopiuj i wklej w przyszłości - wymaga to monitorowania rozwoju aplikacji przez kontrole/przegląd kodu.

Ponadto, działania edukacyjne i zapewnienie odpowiedniego finansowania.

Programowanie typu skopiuj-wklej

Symptomy i konsekwencje

- Ten sam błąd pojawia się w aplikacji pomimo wielu lokalnych poprawek.
- Trudno odszukać wszystkie miejsca, gdzie dany błąd występuje
- Ilość kodu rośnie, możliwości aplikacji nie bardzo
- Kod jest uważany za samo-dokumentujący.
- Prowadzi do nadmiernych kosztów utrzymania.
- Kod przeznaczony do ponownego użycia nie jest w formie, która na to łatwo pozwoli i dodatkowo byłaby przyzwycięstwie udokumentowana
- Błędy naprawiane wieloma unikalnymi poprawkami bez szansy na uogólnienie
- Pomimo ponownego użycia kodu, spodziewanych oszczędności w kosztach utrzymania nie widać.

Programowanie typu skopiuj-wklej

Typowe przyczyny:

- Tworzenie dobrego kodu przeznaczonego do ponownego użycia wymaga wysiłku. Firma może nie być zainteresowana taką inwestycją wybierając działania krótkoterminowe.
- Szybkość pracy jest kluczowym czynnikiem oceny
- Brak abstrakcji, brak zrozumienia typowych strategii rozwoju aplikacji
- Kod gotowy do ponownego użycia nie jest łatwo dostępny i/lub udokumentowany
- Brak perspektywicznego myślenia
- Brak znajomości nowych technologii czy narzędzi prowadzi do bazowania na przykładowym kodzie i próbach modyfikacji.

Mini antywzorzec: Ciągłe starzenie (Continuous Obsolescence)

Problem

Szybkie zamiany technologii, wersji bibliotek. Problem znalezienia zestawu kodu, który będzie chciał razem współpracować. Programiści nie nadążają za zmianami. Dokumentacja/literatura nie nadąża za zmianami w kolejnych wydaniach. Uwaga na produkty zawierające rok wydania w nazwie. Zalety przejścia do nowej technologii, która szybko przestaje być aktualna, są chwilowe.

Rozwiązanie

Polegać na interfejsach, które są stabilne lub pod kontrolą. Odpowiednio dobierać technologię i ciągle aktualizować aplikację.

Mini antywzorzec: Niejednoznaczny punkt widzenia (Ambiguous Viewpoint)

Problem

Analiza i projektowanie obiektowe (Object-oriented analysis and design) dostarcza modelu, ale często bez jasnego określenia z jakiego punktu widzenia on powstał. Przemieszane punkty widzenia nie pozwalają na separację interfejsów i szczegółów implementacyjnych.

Mini antywzorzec: Niejednoznaczny punkt widzenia (Ambiguous Viewpoint)

Rozwiązanie

W analizie i projektowaniu obiektowym używane są 3 punkty widzenia: biznesowy, specyfikacji i implementacji.

Biznesowy punkt widzenia określa model informacji i procesów użytkownika - to model stabilny i warty utrzymania.

Ponieważ perspektywa, z której patrzymy na zagadnienie ma znaczenie dla projektu, musi być jasno określona. Wtedy model staje się użyteczny.

Określenie punktu widzenia dla specyfikacji pozwala prawidłowo zdefiniować abstrakcję i zachowania obiektów.

Aby model związany z implementacją był użyteczny, musi być ciągle uzgadniany ze zmianami jakie pojawiają się w trakcie pisania kodu.

Mini antywzorzec: Kotwica (Boat Anchor)

Problem

Kotwica to fragment kodu lub element sprzętowy, który niewiele (dobrego) wnosi do aplikacji, ale z pewnych przyczyn musi być użyty. Zjawisko nasila się jeśli zakup kotwicy był kosztowny i/lub kupił ją ktoś ważny, przekonany o jej użyteczności.

Rozwiązanie

Ustalenie rozwiązań alternatywnych. Prototypowanie z zastosowaniem licencji demonstracyjno-ewaluacyjnych.

Mini antywzorzec: Ślepy zaułek (Dead End)

Problem

Używany jest komponent, który przestaje być wspierany przez swego dostawcę. Modyfikacje konieczne do wykonania, aby można użyć kolejnej wersji komponentu (o ile będzie dostępna) mogą okazać się zbyt kosztowne lub niewykonalne np. z powodu rotacji pracowników.

Rozwiązanie

Brać pod uwagę harmonogram wydań dostawcy i w odpowiednim momencie migrować do nowszych wersji. Używać izolowanych warstw do odseparowania głównej części aplikacji od rozwiązań własnościowych.
Polegać na sprawdzonych dostawcach.

Mini antywzorzec: Prowizorka na wejściu (Input Kludge)

Problem

Użytkownik końcowy może momentalnie doprowadzić do problemów z programem tylko przez używanie klawiatury. Dzieje się tak wtedy, gdy doraźnie napisane algorytmy odpowiadają za obsługę wejścia programu.

Problem pojawia się, bo programiści znają kombinacje klawiszy, których należy unikać i... ich unikają, lecz tym samym nie w pełni testują zachowanie programu. Czy zawsze sprawdzą co się dzieje, gdy w czasie zapisu danych użytkownik naciśnie np. [Alt]+[F4]?

Rozwiązanie

Jeśli program osiąga fazę produkcyjną to i algorytmy wejściowe muszą być jakości produkcyjnej. Programy do analizy leksykalnej są dostępne.

Mini antywzorzec: Pole minowe (Walking through a Mine Field)

Problem

W kodzie programu są błędy. Jest ich wiele. Użytkownicy ich doświadczają. Systemy operacyjne zawierają błędy, które narażają je na atak. Wraz z rozpowszechnianiem się i wszechobecnością systemów komputerowych skutki błędów stają się coraz bardziej poważne (kosztowne i tragiczne). Używanie aplikacji, lecz także naprawa błędów, jest jak chodzenie po polu minowym - nie wiadomo kiedy program przestanie działać.

Rozwiązanie

Odpowiednie inwestycje w testy.

Mini antywzorzec: Programowanie na ślepo (Blind Development)

Problem

Wymagania użytkowników przekazywane są pośrednio do programistów. Zakłada się, że obie strony (użytkownicy i projektanci systemu) rozumieją wymagania i że są one stałe. W rzeczywistości wymagania ulegają częstym zmianom, lecz o tym, że system nie spełnia oczekiwaniań nic nie wiadomo aż do dostarczenia systemu i problemów z jego akceptacją. Ponadto, użytkownicy końcowi rzadko rozumieją co wynika z zapisanych wymagań. Programiści, którzy nie rozumieją wymagań mają problemy z prawidłowym ustaleniem interakcji pomiędzy komponentami i przygotowaniem odpowiednich interfejsów.

Praca przebiega w niepewności, dokumentacja zazwyczaj nie jest wystarczająco szczegółowa. Nie ma jak jej doprecyzować. Programiści muszą bazować na założeniach, a to prowadzi do pseudo-analizy.

Dochodzi również do całkowitego pominięcia kroków analizy na rzecz bezpośredniego przejścia od wymagań do kodowania.

Mini antywzorzec: Programowanie na ślepo (Blind Development)

Rozwiązanie

Dopuszczenie użytkowników do głosu. Zmiana interfejsu/pojawienie się nowych funkcjonalności to iteracja zewnętrzna z praktycznymi doświadczeniami z udziałem użytkowników. Zmniejsza to ryzyko odrzucenia aplikacji.

Antywzorce architektury

Stovepipe Enterprise

Problem

Wiele systemów w przedsiębiorstwie zaprojektowano niezależnie na każdym z poziomów. Brak kompatybilności blokuje możliwość ponownego użycia i podnosi koszty.

Rozwiązanie

Koordynacja technologii na kilku poziomach. Zdefiniowanie szczegółowych konwencji dotyczących współpracy systemów.

Stovepipe Enterprise

Symptomy i konsekwencje:

- Systemy o niezgodnej terminologii, podejściu, technologii
- Brak dokumentacji architektury systemów
- Brak możliwości rozbudowy systemów w celu obsługi potrzeb biznesowych
- Nieprawidłowe użycie standardów
- Brak ponownego wykorzystania oprogramowania pomiędzy systemami korporacyjnymi
- Brak możliwości współpracy systemów nawet przy użyciu tych samych standardów
- Nadmierne koszty utrzymania spowodowane zmieniającymi się wymaganiami biznesowymi
- Rotacja pracowników, która powoduje problemy z utrzymaniem.

Stovepipe Enterprise

Typowe przyczyny:

- Brak strategii technologicznej
- Brak zachęty do współpracy w zakresie rozwoju systemów
- Brak komunikacji między projektami rozwoju systemów
- Brak wiedzy na temat stosowanego standardu technologicznego

Stovepipe System

Problem

Odpowiednik Stovepipe Enterprise dla pojedynczego systemu. Brak wspólnych abstrakcji dla podsystemów. Podsystemy są integrowane w sposób ad hoc przy użyciu wielu różnych strategii i mechanizmów. Integracja podsystemów jest typu punkt-punkt, czyli każdy niezależnie z każdym innym. Implementacja systemu jest krucha z powodu wielu ukrytych zależności (konfiguracja, sposób instalacji, stan). System jest trudny do rozbudowy, bo każdy dodany podsystem generuje kolejne połączenia. Konserwacja także staje się coraz bardziej trudna.

Rozwiązanie

Architektura, która pozwala na elastyczne zastępowanie modułów. Podsystemy modelowane w sposób abstrakcyjny, ograniczenie liczby udostępnionych interfejsów. Interfejsy powinny poprzez abstrakcję ukrywać wszystkie szczegóły niższego poziomu.

Stovepipe System

Symptomy i konsekwencje:

- Duża rozbieżność pomiędzy dokumentacją architektury a implementacją
- Projekt przekroczył budżet i bez wyraźnego powodu przesunął się harmonogram wydania
- Zmiany wymagań są kosztowne do wdrożenia
- Utrzymanie systemu generuje zaskakujące koszty
- System może spełnić udokumentowane wymagania, ale nie spełnia oczekiwania użytkowników
- Użytkownicy opracowują obejścia aby poradzić sobie z ograniczeniami systemu
- Problem z automatyzacją procesu instalacji
- Współdziałanie z innymi systemami nie jest możliwe
- Wprowadzenia zmiany jest coraz trudniejsze
- Modyfikacje systemu wprowadzają nowe, poważne błędy

Stovepipe System

Typowe przyczyny:

- Brak wspólnego mechanizmu integracji podsystemów utrudnia opisanie i modyfikację architektury
- Brak abstrakcji; interfejsy są unikalne dla każdego z podsystemów
- Niewystarczające wykorzystanie metadanych
- Ścisłe powiązanie pomiędzy zaimplementowanymi klasami
- Brak wizji architektonicznej

Uzależnienie (Vendor Lock-In)

Inne nazwy

Product-Dependent Architecture, Bondage and Submission, Connector Conspiracy

Problem

Zbyt duże uzależnienie od jednego dostawcy. Przenoszenie danych na serwery usługodawcy - nagle może nie być do nich dostępu lub, co gorsze, za ich dalsze przetrzymywanie zaczną być naliczane opłaty.

Wymiana informacji oparta o pewien ustalony format własnościowy. Format może ulegać zmianom wraz z kolejnymi wydaniami aplikacji. Konieczna jest ciągła konserwacja aby utrzymać całość w działaniu.

Rozwiązanie

Użycie warstwy służącej do izolacji aplikacji od rozwiązań własnościowych. Używana jest abstrakcja, która ukrywa leżącą poniżej infrastrukturę czy interfejsy zależne od konkretnego produktu. Dzięki temu aplikacja jest niezależna od interfejsów produktów.

Uzależnienie (Vendor Lock-In)

Symptomy i konsekwencje:

- Aktualizacje aplikacji zbiegają się z nowymi wersjami innego (komercyjnego) produktu.
- Opóźnienia w dostarczaniu nowych funkcjonalności pewnego produktu powodują opóźnienia w wydaniach własnego.
- Jeśli zewnętrzny produkt nie jest aktualizowany konieczny jest zakup innego i integracja z nowym rozwiązaniem.

Uzależnienie (Vendor Lock-In)

Typowe przyczyny:

- Produkt jest wybierany wyłącznie na podstawie informacji marketingowych bez testów
- Brak prac nad odizolowaniem aplikacji od bezpośrednich zależności z produktem
- Produkt przewyższa swoją złożonością potrzeby aplikacji.

Architektura przez domniemanie (Architecture by Implication)

Inne nazwy

Wherfore art thou architecture?

Problem

Brak specyfikacji architektury dla tworzonego systemu. Osoby odpowiedzialne za architekturę systemu miały już wcześniej doświadczenie z tworzeniem takiego systemu, wydaje im się więc, że dokumentacji nie potrzebują. Taka nieuzasadniona pewność siebie prowadzi do znaczącego wzrostu ryzyka w obszarach kluczowych dla odniesienia przez system sukcesu.

Architektura przez domniemanie (Architecture by Implication)

Czego może brakować w definiowaniu architektury?

- Architektury oprogramowania i specyfikacji zawierającej wybór języka, bibliotek, standardów kodowania, zarządzania pamięcią itd.
- Architektury sprzętowej w tym konfiguracji klientów i serwisów
- Architektury komunikacyjnej w tym protokołów sieciowych i urządzeń
- Architektury przechowywania danych zawierającej informacje o bazach danych czy obsługie plików.
- Architektury bezpieczeństwa aplikacji zawierającej model użycia współbieżności
- Architektury zarządzania systemami.

Rozwiązanie

Zorganizowane podejście do definicji architektury systemów, które opiera się na wielu widokach. Dokumentacja powinna być łatwa do zrozumienia i niedroga w utrzymaniu. Jeśli architektura systemu ma być rozumiana muszą ją przygotowywać osoby, które w pełni rozumieją używane technologie. Rozwijanie architektury od zera powinno być realizowane iteracyjnie.

Architektura przez domniemanie (Architecture by Implication)

Symptomy i konsekwencje:

- Nieznajomość nowych technologii
- Brak planów awaryjnych,
- Brak rozwiązań alternatywnych dla stosowanych technologii
- Brak planowania architektury; niewystarczające określenie architektury dla oprogramowania, sprzętu, komunikacji, przechowywania danych, bezpieczeństwa
- Zbliżające się problemy z powodu nieodpowiedniej wydajności czy nadmiernej złożoności.

Architektura przez domniemanie (Architecture by Implication)

Typowe przyczyny:

- Nadmierna pewność siebie managerów i programistów
- Poleganie na poprzednio zdobytych doświadczeniach
- Domniemane i nierozwiążane problemy z architekturą spowodowane lukami w inżynierii systemów.
- Brak zarządzania ryzykiem

Projekt komisyjny (Design by Committee)

Inne nazwy

Gold Plating, Standards Disease, Make Everybody Happy, Political Party

Problem

Projekt jest złożony. Posiada wiele funkcji i wariacji. Realizacja specyfikacji nie jest możliwa w rozsądny czasie. Testowanie jest praktycznie niemożliwe z powodu nadmiernej złożoności, dwuznaczności, nadmiernych ograniczeń. Ponieważ projekt jest wynikiem pracy wielu osób nie jest przejrzysty koncepcyjnie - każdy *wtrącił swoje trzy grosze*.

Rozwiązanie

Głęboka reforma procesu spotkań zespołu. Zegar na ścianie, który daje świadomość upływu czasu. Podsumowanie i dodawanie szczegółów tylko na żądanie. Publikowanie celów spotkania. Plan spotkania. Przypisanie wyraźnych ролей w procesie tworzenia oprogramowania.

Projekt komisyjny (Design by Committee)

Symptomy i konsekwencje:

- Dokumentacja projektowa jest zbyt złożona, nieczytelna, niespójna...
- Dokumentacja projektowa jest obszerna (setki lub tysiące stron)
- Brakuje zbieżności i stabilności w wymaganiach i projekcie.
- Na spotkaniach kwestie merytoryczne omawiane są rzadko.
- Postęp pracy jest niezwykle powolny
- Rozmowy i praca wykonywane są sekwencyjnie czyli rozważana jest w danym czasie tylko jedna kwestia, większość osób jest nieproduktywna przez większość czasu
- Niewiele kwestii można rozstrzygnąć poza spotkaniami
- Podejmowanie decyzji na czas się nie zdarza
- Brak priorytetyzacji, brak kolejności wdrażania funkcjonalności
- Architekci i deweloperzy mają sprzeczne interpretacje projektu
- Zajmowanie się każdą specyfikacją zaprojektowaną przez komisję staje się pełnoetatową pracą.

Projekt komisyjny (Design by Committee)

Typowe przyczyny:

- Brak wyznaczonego architekta projektu
- Dozwolenie na spotkaniach aby najgłośniejsi wygrywali
- Próba uszczęśliwienia wszystkich poprzez uwzględnianie ich pomysłów
- Próba projektowania podczas spotkań więcej niż pięciu osób
- Brak wyraźnych priorytetów
- Brak podziału zagadnień
- Dodawanie funkcjonalności ze względu na czyjś interes.

Odkrywanie koła na nowo (Reinvent the Wheel)

Inne nazwy

Design in a Vacuum, Greenfield System

Problem

Ignorowanie faktu istnienia dużej ilości oprogramowania (darmowego czy komercyjnego), które nadaje się do ponownego użycia. Systemy budowane od podstaw pomimo istnienia innych z nakładającymi się funkcjami.

Analiza "od ogółu do szczegółu" prowadzi do zaprojektowania nowej architektury i niestandardowego oprogramowania, czyli problemów z ponowym użyciem kodu.

Rozwiązańe

Poszukiwania! Zarówno kodu, który można użyć jak i sposobu na stworzenie architektury, która pozwoli na ponowne użycie. Użycie istniejącego rozwiązania może być tańsze i mniej ryzykowne od tworzenia kodu od początku.

Odkrywanie koła na nowo (Reinvent the Wheel)

Symptomy i konsekwencje:

- Systemy budowane bez możliwości ponownego użycia
- Replikacja funkcji oprogramowania komercyjnego
- Niedojrzałe i niestabilne architektury oraz wymagania.
- Wydłużone cykle rozwojowe obejmujące nieudane i ślepe prototypy.
- Niewłaściwe zarządzanie ryzykiem i kosztami, prowadzące do przekroczenia harmonogramu i budżetu.
- Niemożność dostarczenia pożądanych funkcji dla użytkownika końcowego
- Wkładanie dużego wysiłku w celu odtworzenia funkcjonalności działającej w istniejących systemach.

Odkrywanie koła na nowo (Reinvent the Wheel)

Typowe przyczyny:

- Brak komunikacji i transferu technologii
- Z założenia system budowany od podstaw
- Brak odpowiedniego zarządzania
- Każdy system używający innych interfejsów
- Brak odpowiedniej wiedzy

Mini antywzorzec: Stożki migracyjne

Problem

System zaprojektowany do pracy lokalnej zmieniane są w systemy rozproszone. Oryginalne interfejsy mogą używać informacji, która w systemie rozproszonym nie ma sensu (np. lokalizacja pliku na dysku czy stan połączeń sieciowych). Wymiana informacji działająca poprawnie lokalnie (np. przesyłanie dużych ilości małych obiektów - drobnoziarniste operacje transferu) okazuje się źle (nieefektywnie) działać w systemach rozproszonych. W tej chwili panuje zwyczaj przenoszenia aplikacji "do chmury". Przebudowa aplikacji do pracy w nowym środowisku może być kosztowna.

Rozwiązanie

Przeprojektowanie interfejsów. Powiększyć "ziarna" w celu zwiększenie efektywności komunikacji. Generalnie refaktoryzacja.

Mini antywzorzec: bezładna mieszanina/zbieranina (jumble)

Problem

Zmieszanie poziomych i pionowych elementów projektu daje niestabilną architekturę. Wymieszanie poziomych i pionowych elementów ogranicza możliwość ponownego użycia oraz solidność architektury i komponentów oprogramowania systemowego.

Rozwiązanie

Po pierwsze zidentyfikować poziome elementy projektu i przekazać je do osobnej warstwy architektury. Następnie użyć elementów pionowych do uchwycenia wspólnych funkcjonalności.

Właściwa równowaga elementów poziomych, pionowych i metadanych w architekturze prowadzi do dobrze ustrukturyzowanego, rozszerzalnego oprogramowania wielokrotnego użytku.

Mini antywzorzec: Papierkowa robota

Problem

Jeśli dokumentacja jest podstawą dla projektu, często okazuje się, że zawiera ona mniej niż przydatne wymagania i specyfikacje, ponieważ autorzy unikają podejmowania ważnych decyzji. W celu uniknięcia błędu opracowywane są kolejne alternatywy. Dokumenty są obszerne i niezrozumiałe.

Gdy nie ma ustalonych priorytetów dokumenty mają ograniczoną wartość. Setki stron wymagań, które wszystkie są jednakowo ważne i/lub obowiązkowe nie mają sensu. Deweloperzy nie wiedzą co wdrożyć i w jakiej kolejności.

Rozwiązanie

Przydatny projekt architektury to mały zestaw diagramów i tabel, który przedstawia architekturę operacyjną, techniczną i systemową teraźniejszą i przyszłą systemu informatycznego. Typowy plan obejmować powinien na tyle mało diagramów i tabel aby mogły być przedstawione w ciągu godziny. Plany powinny charakteryzować zarówno istniejące systemy, jak i planowane rozszerzenia. Ponieważ plany architektury pozwalają na przedstawienie technologii wielu projektów, zwiększa się możliwości interoperacyjności i ponownego wykorzystania.

Mini antywzorzec: Wilczy bilet (Wolf Ticket)

Problem

Wilczy bilet to produkt, który twierdzi, że jest otwarty i zgodny ze standardami ale te standardy nie dają się wyegzekwować. Produkty są dostarczane z zastrzeżonymi interfejsami, które mogą się znacznie różnić od opublikowanego standardu.

Problemem jest to, że konsumenci technologii często zakładają, że otwartość przynosi pewne korzyści. W rzeczywistości najważniejsze jest utrzymanie standardów, bo normy zmniejszają koszty migracji i poprawiają stabilność technologii. Różnice we wdrażaniu standardów często negują zakładane korzyści. Wiele specyfikacji norm jest zbyt elastyczna aby zapewnić interoperacyjność i przenośność. Inne standardy są nadmiernie skomplikowane i są wdrażane w sposób niekompletny i niekonsekwentny.

Rozwiązanie

Ostrożne korzystanie z rozwiązań otwartych z jednoczesnym domaganiem się gwarancji przydatności.

Mini antywzorzec: Ciepła posadka (Seat Warmers)

Problem

Aby projekt odniósł sukces potrzebni są wykwalifikowani programiści. Tylko nieliczni pracują efektywnie, pozostały to przeciętni pracownicy. Z uwagi na ciągle zmieniające się wymagania i specyfikacje liczba osób w zespole lubi rosnąć.

Rozwiązanie

Eliminacja ciepłych posadek. Mniejsze zespoły z jasno ustaloną odpowiedzialnością indywidualnych pracowników dają większą szansę na powodzenie. Trzymanie się harmonogramu prac wymusza opracowywanie działających rozwiązań.

Mini antywzorzec: Szwajcarski scyzoryk (Swiss Army Knife)

Problem

Szwajcarski scyzoryk to nadmiernie złożony interfejs klasy. Projektant stara się zapewnić wszystkie możliwe zastosowania danej klasy poprzez dodawanie dużej liczby metod. Projektant nie potrafi ustalić wyraźnej abstrakcji lub celu dla tworzonej klasy. Skomplikowany interfejs jest trudny do zrozumienia dla innych programistów. Preferowany sposób użycia klasy jest ukryty. Ponadto, utrudniona jest dokumentacja, konserwacja i wykrywanie błędów.

Rozwiązanie

Złożone interfejsy muszą być pod kontrolną. Pomaga opracowanie dokumentu przedstawiającego stosowne konwencje użycia technologii.

Mini antywzorzec: Abstrakcjonisi kontra Implementatorzy (Abstractionists versus Implementationists)

Problem

Umiejętność programowania nie oznacza umiejętności definiowania abstrakcji. "Abstrakcjonisi" potrafią omawiać koncepcje projektowania oprogramowania bez zagłębiania się w detale implementacyjne; potrafią definiować dobre abstrakcje. "Implementatorzy" potrzebują często przykładu aby zrozumieć ideę abstrakcji, sami dobrych abstrakcji nie potrafią stworzyć. Niestety, "Implementatorów" jest zdecydowanie więcej niż "Abstrakcjonistów". Jeśli decyzje projektowe podejmowane są przez głosowanie, "Abstrakcjonisi" przegrywają. Główną konsekwencją są projekty oprogramowania cechujące się nadmierną złożonością, która utrudnia rozwój, modyfikację, rozszerzanie systemu, dokumentację i testy.

Mini antywzorzec: Abstrakcjonenci kontra Implementatorzy (Abstractionists versus Implementationists)

Rozwiązanie

Trzeba zdać sobie sprawę z tego, że różne osoby mają do odegrania różne role. "Abstrakcjonenci" to typowo osoby ze znacznym doświadczeniem w większości kluczowych technologii. Ich zadanie to ułatwiać komunikację między użytkownikami końcowymi a programistami; mają odpowiadać za zarządzanie złożonością i zapewniać możliwość dostosowania systemów do potrzeb. "Abstrakcjonistom" trzeba pozwolić na zarządzanie interfejsami programistycznymi.

Antywzorce zarządzania projektem

Paraliż Analityczny (Analysis Paralysis)

Inne nazwy

Waterfall, Process Mismatch

Problem

Analiza obiektowa koncentruje się na dekompozycji problemu na części składowe, ale nie ma oczywistej metody identyfikacji odpowiedniego poziomu szczegółowości niezbędnego do zaprojektowania systemu.

Powodem Paraliżu Analitycznego jest przekonanie, że „projekty nigdy nie zawodzą, tylko implementacje”. Przedłużając fazę analizy i projektowania, unika się ryzyka odpowiedzialności za wyniki. Ale to jest strategia przegrywająca, bo rozpoczęcia (i zakończenia) implementacji nie sposób przecież uniknąć.

Paraliż Analityczny występuje, gdy celem (samym w sobie) jest osiągnięcie doskonałości i kompletności fazy analizy. Tworzone są bardzo szczegółowe modele, które okazują się nieprzydatne w kolejnych fazach.

Kluczowym wskaźnikiem wystąpienia Paraliżu Analitycznego jest to, że dokumenty analizy nie mają już sensu dla ekspertów z danej dziedziny.

Paraliż Analityczny

Założenia, które przyczyniają się do Paraliżu Analitycznego:

- Szczegółową analizę można pomyślnie ukończyć przed kodowaniem.
- Wszystko na temat problemu jest z góry znane.
- Modele nie zostaną rozszerzone ani ponownie sprawdzone podczas opracowywania oprogramowania.

Rozwiązańe

Kluczową kwestią jest rozwój przyrostowy. Nie zakłada on całościowej znajomości problemu lecz rozpoznawanie szczegółów w trakcie rozwoju oprogramowania.

Paraliz Analityczny

Symptomy i konsekwencje:

- Projekt uruchamiany wielokrotnie i wiele przeróbek modeli z uwagi na zmiany osobowe oraz zmiany w samym projekcie.
- Zagadnienia projektowe i implementacyjne wielokrotnie powracają w trakcie fazy analizy
- Koszt analizy przekracza oczekiwania a czas jej zakończenia jest nieprzewidywalny.
- Faza analizy nie obejmuje już interakcji z użytkownikami. Większość przeprowadzonych analiz ma charakter czysto spekulacyjny.
- Złożoność modeli fazy analizy skutkuje zawiłymi implementacjami, w efekcie czego system jest trudny do opracowania, udokumentowania i przetestowania.

Paraliż Analityczny

Typowe przyczyny:

- Zakłada się, że proces analizy jest kaskadowy, w rzeczywistości lepiej sprawdzi się praca iteracyjna
- Zarząd nalega na zakończenie wszystkich analiz przed rozpoczęciem fazy projektowania.
- Cele w fazie analizy nie są dobrze określone
- Kierownictwo nie chce podejmować zdecydowanych decyzji dotyczących tego, kiedy części domeny są wystarczająco dobrze opisane.
- Wizja projektu jest rozproszona
- Analiza wychodzi poza dostarczenie znaczącej wartości (zamiast zakończyć analizę, jest ona na siłę, nadal kontynuowana).

Śmierć przez planowanie (Death By Planning)

Inne nazwy

Glass Case Plan, Detailitis Plan

Problem

W wielu przypadkach szczegółowy plan jest rzeczą dobrą, ale niekoniecznie gdy w grę wchodzi projekt informatyczny. Tworzenie oprogramowania musi zakładać ciągłe zmiany i sporo chaosu wynikającego z samej natury problemu. Śmierć przez planowanie występuje, gdy szczegółowe plany projektu oprogramowania są traktowane zbyt poważnie. Wiele projektów kończy się niepowodzeniem z powodu nadmiernego planowania. Gdy zbyt wiele wysiłków pochłania stworzenie planów, na wytworzenie kodu zaczyna brakować i czasu i funduszy.

Rozwiążanie

Plan projektu wskazujący głównie elementy takie jak produkty i komponenty. Plan powinien zawierać "kamienie milowe" dla każdego komponentu i produktu - dzięki temu można będzie poznać czy projekt mieści się w ramach czasowych, czy już są w nim opóźnienia.

Śmierć przez planowanie (Death By Planning)

Symptomy i konsekwencje:

- Niezdolność do planowania na racjonalnym poziomie
- Koncentracja bardziej na kosztach niż na dostarczeniu wyniku
- Spędzanie więcej czasu na planowaniu, opisywaniu postępów i przebudowywaniu niż na dostarczaniu oprogramowanie
- Celem nie jest już dostarczanie oprogramowania lecz planów
- Ciągłe zmiany planów
- Zmiany jednego planu powodują kolejne zmiany w innych planach
- Dalsze koszty, status projektu rozmywa się, plany przestają mieć znaczenia, kontrola nad czasem dostawy zmniejsza się.
- Projekt nie jest dostarczany na czas, może zostać anulowany
- Utrata kluczowego personelu

Śmierć przez planowanie (Death By Planning)

Typowe przyczyny:

- Brak racjonalnego podejścia do planowania, harmonogramów i rejestrowania postępów.
- Brak aktualnego planu projektu z opisem stanu komponentów i czasu dostarczenia
- Nieznajomość podstawowych zasad zarządzania projektem.
- Nadgorliwe wstępne (i ciągłe) planowanie mające na celu wymuszenie absolutnej kontroli rozwoju.
- Planowanie jako podstawowa działalność projektu.

Corncob

Inne nazwy

Corporate Shark, Loose Cannon

Problem

Ze względu na rygorystyczne harmonogramy i presję budżetową rozwój oprogramowania może się stać stresujący. Trudne osoby tylko sytuację pogarszają poprzez destrukcyjne zachowania w zespole programistycznym lub, co gorsza, w całym przedsiębiorstwie.

Dzieje się tak dlatego, że są one zwykle ekspertami w manipulowaniu polityką na poziomie osobistym i/lub organizacyjnym.

Rozwiążanie

Osoba powodująca problemy nie może być w tym wspierana przez zarząd - tylko w takiej sytuacji interes zespołu programistycznego może przeważyć. W sytuacjach skrajnych trudna osoba musi zostać odseparowana od zespołu łącznie z zakończeniem współpracy.

Symptomy i konsekwencje:

- Zespół programistyczny lub projekt nie jest w stanie zrobić postępu, ponieważ ktoś się nie zgadza z ich kluczowymi celami lub niezbędnymi procesami i ciągle stara się je zmieniać.
- Pod pozorem troski jedna osoba nieustannie podnosi zarzuty dotyczące wydajność, niezawodność, udziału w rynku technologii itd.
- Niszczycielskie zachowanie jest dobrze znane wielu osobom w przedsiębiorstwie, ale jest tolerowane
- "Siły polityczne" tworzą środowisko, w którym trudno jest ograniczyć dyskusje do tematów technicznych
- Częste zmiany zakresu lub wymagań systemu spowodowane przez "siły polityczne".
- Często osobą niszczącą jest kierownik, który pozostaje bez nadzoru.
- Firma nie ma zdefiniowanego procesu decyzyjnego, który umożliwiałby rozwiązywanie problemów. Pozwala to menadżerom ingerować w sprawy spoza zakresu odpowiedzialności.

Typowe przyczyny:

- Kierownictwo wspiera destrukcyjne zachowanie, ponieważ nie zauważa/reaguje na zachowanie danej osoby. Często wręcz to ta osoba przedstawia sytuację kierownictwu.
- Taka trudna osoba ma ukryty plan, który jest sprzeczny z celami zespołu.
- Istnieje zasadnicza różnica zdań między członkami zespołu
- Kierownictwo nie izoluje grupy od sił wewnętrznych i zewnętrznych
- Źle przydzielone role pracownikom, są tacy, którzy nadużywają ich do własnych celów
- Brak przydziału pracownikom odpowiedzialności

Mini antywzorzec: Zlot chwalipieł (Blowhard Jamboree)

Problem

W różnych mediach pojawiają się informacje krytykujące poszczególne technologie. Wiele z tych informacji nie ma żadnego oparcia w doświadczeniach czy badaniach - wynikają z niewiedzy czy stronniczości piszącego. Programiści tracą czas na rozwiewanie obaw decydentów, których takie informacje zaniepokoiły.

Rozwiązanie

Zatrudniony ekspert/eksperci znający się na kluczowych technologiach. Można także wyznaczyć pracownika do śledzenia określonych technologii. Zamiast komunikatów prasowych bazować na informacjach ekspertów.

Mini antywzorzec: Viewgraph Engineering

Problem

Zdarza się, że programiści zamiast pisać kod przygotowują raporty. Sfrustrowani sytuacją i wiedząc, że ta praca nie przynosi żadnych korzyści, produkują je byle jak.

Rozwiązanie

Przekierować pracowników do wykonywania użytecznej pracy np. tworzenia prototypów.

Mini antywzorzec: Strach przed sukcesem (Fear of Success)

Problem

Będąc na krawędzi sukcesu niektóre osoby zaczynają się obsesyjnie martwić o to, co może pójść źle. Mogą swoimi wątpliwościami zarazić inne osoby z zespołu, który zaczyna podejmować irracjonalne, szalone decyzje. To z kolei powoduje zmianę sposobu postrzegania rezultatu i może ostatecznie mieć destrukcyjny wpływ na wynik projektu.

Ponieważ zakończenie projektu może doprowadzić do rozwiązania grupy, problemy nasilają się w miarę zbliżania się do końca pracy.

Rozwiązanie

Oświadczenie kierownictwa, które oficjalnie potwierdzi pomyślny wynik projektu. Jest ono potrzebne, nawet jeśli faktyczny wynik jest niejednoznaczny.

Zadeklarowanie sukcesu pomaga złagodzić problemy z zakończeniem pracy i utrzymać zaangażowanie zespołu.

Profesjonalne uznanie jest tanie i jednocześnie bardzo wysoko cenione przez odbiorców.

Mini antywzorzec: Przemoc intelektualna (Intellectual Violence)

Problem

Osoba znająca dobrze teorię, technologię czy nowinki wykorzystuje tą wiedzę do (nieświadomego) zastraszania innych w trakcie spotkań. Dochodzi do załamania komunikacji, bo część osób nie rozumie pewnych koncepcji. Może na tym ucierpieć postęp prac.

Kiedy przemoc intelektualna jest wszechobecna, powstaje kultura obronna, która zmniejsza wydajność. Ludzie kontrolują i ukrywają informacje zamiast je udostępniać. Mogą także nie chcieć o pewnych sprawach dyskutować.

Rozwiązanie

Mentoring. Uznaje się, że każdy ma wyjątkowe talenty i wartościową wiedzę, niezależnie od pozycji w hierarchii organizacyjnej. Ludzie są zachęcani do dzielenia się wiedzą.

Zachęcanie ludzi do dzielenia się informacjami to skuteczny sposób na wykorzystanie zasobów wiedzy i ograniczenie przypadków ponownego rozwiązywania rozwiązań już wcześniej problemów.

Mini antywzorzec: Ćwiczenia przeciwpożarowe (Fire Drill)

Problem

Projekt rozpoczęto, ale przez długi okres czasu działania projektowe i samo tworzenie kodu nie postępują. Powodem jest hamowanie działania poprzez kierownictwo, które każe pracownikom czekać lub udziela niepewnych i sprzecznych wskazówek. Dochodzi do kolejnych zmian związanych z projektem. Po pewnym czasie, z racji tego, że czas do dostarczenia produktu zaczyna się kończyć, faza tworzenia programu zaczynana jest nagle. Management żąda dostarczenia produktu w nierealnym czasie. Ponieważ najważniejszym czynnikiem staje się czas chętnie dochodzi się do kompromisów w kwestii jakości i testowania oprogramowania. Ponadto, im mniej czasu pozostaje, tym gorszej jakości kod (i dokumentacja) jest akceptowany przez kierownictwo.

Rozwiązanie

Ćwiczenia przeciwpożarowe zachować tylko dla sytuacji kryzysowych. Dla normalnych prac związanych z dostarczaniem oprogramowania lepsze są dłuższe ramy czasowe. Ponadto, trzeba ustalić, że oprogramowanie jest rozwijane pomimo istniejących i nierozwiązywanych problemów szczebla wyższego.

Mini antywzorzec: Waść (The Feud)

Problem

Konflikt osobowości pomiędzy menadżerami. Pracownicy cierpią z powodu nieporozumień wśród kierownictwa. Ponadto obserwowany jest brak produktywnej komunikacji. Ogólny brak współpracy hamuje jakąkolwiek formę użytecznego transferu technologii. Waść może negatywnie wpływać na produktywność i wizerunek firmy.

Rozwiązanie

"Nie ma problemu, którego przyjęcie z pizzą nie może rozwiązać". Generalnie - osoby skonfliktowane muszą ze sobą zacząć rozmawiać.

Testowanie

Definicje podstawowe

Testowanie weryfikacja czy zachowanie systemu zgodne jest z przewidywanym

Test uruchamianie określonej funkcjonalności w zadanych warunkach i parametrach wejściowych w celu testowania.

Błąd zachowanie aplikacji, które nie jest zgodne z zapisem w wymaganiach. Jak błąd uznaje się również zachowanie sprzeczne czy niepełne.

Błąd krytyczny niepożądane zachowanie systemu, które wpływa na funkcjonowanie procesów biznesowych.

Błąd blokujący sytuacja, która uniemożliwia kontynuowanie testów danego obszaru. Błąd o bardzo dużym priorytecie.

Przypadek testowy zestaw warunków początkowych i końcowych dla wykonania pewnej ścieżki w programie, realizacja tej ścieżki i zapis wyniku.

Scenariusz testów czynności, które trzeba zrealizować aby przeprowadzić test.

Wymaganie zestaw warunków i spodziewanych wyników działania

Techniki testowania

Metoda czarnej skrzynki

W tej metodzie kod traktowany jest jako nieznany - tester używa wyłącznie interfejsu lub programu klienckiego. Testowanie sprowadza się do uruchamiania funkcjonalności dla zadanych parametrów wejściowych (powinny odpowiadać warunkom rzeczywistej obsługi systemu) i sprawdzania poprawności odpowiedzi. Zaleta: nie trzeba umieć analizować kodu. Wada: nie wiadomo jaką część kodu została przetestowana.

Metoda białej skrzynki

W tej metodzie kod jest dostępny i jest analizowany. Tester musi znać dany język oraz zastosowaną technologię. Testy b.s. powinny być używane do znalezienia luk w pokryciu kodu przez scenariusze testowe. Zaleta: możliwość obliczenia pokrycia kodu testami, możliwość lokalizacji błędu. Wada: testy mogą być tendencyjnie ułożone tak, aby znany testerowi kod ich nie zaliczył, choć warunki testowania odbiegać będą od wymagań.

Miara jakości oprogramowania

Proporcja błędów

Ważna jest proporcja błędów ujawnionych w trakcie testowania w fazie produkcji do liczby błędów, które ujawniły się u użytkowników (w środowisku zewnętrznym). Rodzaj błędu nie jest uwzględniany.

Waga błędów

Błądem będzie zarówno błąd krytyczny jak i ortograficzny czy literówka. Mają one jednak różne wagi - warto to uwzględnić.

Opinia klienta

Liczby mówią, że jest dobrze, a klient, że źle... Wniosek: należy zmienić zasady pomiaru.

Kiedy błąd występuje

Warto zestawić powagę błędów (skutek) z warunkami koniecznymi do jego powtórzenia. Bardzo źle o produkcie świadczy to, że łatwo doprowadzić do poważnego błędu.

Poziomy wykonywania testów

Poziomy od najniższego do najwyższego

- testy modułowe (jednostkowe),
- testy integracyjne wewnętrzne,
- testy systemowe,
- testy integracyjne zewnętrzne,
- testy akceptacyjne (odbiorcze).

Typy testów

Podział wg. celu wykonania testu

- Testy funkcjonalne
- Testy niefunkcjonalne
 - Testy wydajności
 - Testy wydajnościowe (ang. performance testing)
 - Testy obciążeniowe (ang. load testing)
 - Testy przeciążeniowe (ang. stress testing)
 - Testy bezpieczeństwa
 - Testy ergonomii
 - Testy przenośności kodu (instalacji)

Testy regresywne

Zmiany

W kodzie dochodzi do zmian. Jakie funkcjonalności mogą zostać potencjalnie uszkodzone na skutek zmiany kodu?

Testy regresywne

T.r. mają ustalić, czy zmiana w kodzie nie uszkodziła istniejących funkcjonalności. Polegają na ponownym wykonaniu testu wcześniej sprawdzonego już kodu (nawet całości aplikacji), aby mieć pewność, że zmiany nie zaszkodziły.

How Much is Enough?

Test całości kodu jest kosztowny. Test części niepewny. Nie ma złotej reguły, pozostaje tylko doświadczenie i znajomość testowanego kodu.

Projektowanie testów

Po co projektować?

Minimalizacja kosztów i maksymalizacja jakości aplikacji.

Projektowanie:

- ① Identyfikacja warunków
- ② Wyspecyfikowanie przypadków testowych
- ③ Ustalenie kolejności użycia przypadków testowych

Harmonogram

Testy wykonywane są aby odszukać błędy. Zakładanie, że błędów nie będzie jest nieuzasadnione. Należy w harmonogramie prac zaplanować czas na iteracje i poprawki.

Automatyzacja testów

Po co automatyzować?

Minimalizacja kosztów powtarzalnych i stałych testów. Skrócenie czasu wykonania. Możliwość innego użycia zasobów ludzkich.

Niebezpieczeństwo

Kiepskie testy automatyczne przy jednoczesnym zaniechaniu wykonywania testów manualnych. Przecena korzyści z automatyzacji.

Co automatyzować?

- Testy wydajnościowe
- Testy regresyjne
- Testy dające powtarzane wyniki
- Testy mało podatne na zmiany GUI
- Testy dla prostego GUI i rozbudowanej logiki biznesowej

KONIEC

KONIEC