Dobre zasady tworzenia oprogramowania w kropelce

Niniejszy artykuł będzie próbą skondensowania poprawnych zasad tworzenia oprogramowania, głównie w paradygmacie obiektowym. Niemniej jednak w innych paradygmatach wiele z tych zasad można również uznać za poprawne i godne naśladowania.

Publikacja jest zbiorem zasad, który pozwoli odpowiedzieć programiście na pytanie, jak projektować hierarchię obiektów oraz organizować współpracę pomiędzy nimi. Może też odpowiadać na pytanie, co zrobiłem źle w moim systemie, że tak trudno wprowadza się w nim zmiany.

Artykuł jest skierowany do programistów, którzy znają już dosyć dobrze paradygmat obiektowy. Jest on podsumowaniem tej metodyki tworzenia systemów.

Poznasz:

- Manifest Agile a praca programisty
- Zasady GRASP
- Zasady SOLID
- Programowanie poprzez kontrakt
- Programowanie przez dziedziczenie versus programowanie przez kompozycję
- Kilka słów o refaktoryzacji
- Metazasady dobrego programowania
- Wzorce projektowe w praktycznych zastosowaniach

Powinieneś:

- mieć doświadczenie w programowaniu w języku obiektowym
- znać obiektowy paradygmat programowania

Agile a praca programisty

W artykule wspominam o manifeście Agile ze względu na jego wpływ na metodykę tworzenia systemów. Manifest Agile składa się z czterech zasad postępowania:

- Ludzie i ich wzajemne interakcje (współdziałanie) ponad procedury i narzędzia.
- Działające oprogramowanie ponad wyczerpującą dokumentację.
- Współpraca z klientem ponad negocjację umów.
- Reagowanie na zmiany ponad realizowanie planu.

Dla samego procesu implementowania systemu czwarty punkt ma najsilniejszy wydźwięk.

Manifest Agile pokazał, że podobnie jak w życiu, w tworzeniu oprogramowania sztuką jest umiejętność reagowania na zmiany. W kontekście tworzenia oprogramowania reagowanie na zmiany rozumiem jako możliwość i łatwość wprowadzania zmian w kodzie źródłowym w momencie zmiany wymagań funkcjonalnych. Trzeba więc tworzyć systemy, które pozwalają na łatwe wprowadzanie zmian w przyszłości. Cały ten artykuł ma pokazać zestaw reguł, dzięki którym będzie to możliwe.

Nurt Agile i techniki w nim stosowane zostały opisane w przejrzysty sposób w książce Alana S.Kocha „Agile Software Development Evaluating the Methods for Your Organization”.

GRASP czyli General Responsibility Assignment Software Patterns

GRASP składa się z kilku podstawowych zasad mówiących jaką odpowiedzialność powinno się przypisywać określonym obiektom w systemie.
Każda z zasad odpowiada na inne pytanie dotyczące ustalania odpowiedzialności wybranego obiektu:

- Creator – Kiedy obiekt powinien tworzyć inny obiekt?
- Information Expert - Jak ustalać zakres odpowiedzialności obiektu?
- Controller – Jak część systemu odpowiedzialna za logikę biznesową programu powinna komunikować się z interfejsem użytkownika?
- Low Coupling – Jak ograniczyć zakres zmian w systemie w momencie zmiany fragmentu systemu?
- High Cohesion – Jak ograniczać zakres odpowiedzialności obiektu?
- Polymorphism – Co zrobić, gdy odpowiedzialność różni się w zależności od typu?
- Pure Fabrication – Komu delegować zadania, gdy nie można zidentyfikować do kogo należy podany zbiór operacji?
- Indirection – Jak zorganizować komunikację między obiektami, aby mocno ich nie wiązać?
- Protected Variations – Jak przypisywać odpowiedzialność, aby zmiana w jednej części systemu nie powodowała niestabilności innych elementów?

Creator

Określa kiedy podany obiekt powinien tworzyć inny obiekt.
Dwa obiekty pozostają w relacji ze sobą w momencie, gdy jeden jest tworzony przez drugi obiekt.
Obiekt B powinien tworzyć obiekt A, jeśli: B agreguje A, B operuje na danych obiektu A, B używa bezpośrednio A, B dostarcza informacji niezbędnej do utworzenia A.

Information Expert

Programista powinien delegować nową odpowiedzialność (nowe funkcje) do obiektów, które zawierają najwięcej informacji/danych pozwalających zrealizować nową funkcjonalność. Istotne jest więc określenie, jakie dane są niezbędne do wypełnienia danej odpowiedzialności. Dopiero w tym momencie można powiedzieć, który obiekt powinien mieć przypisaną nową funkcjonalność.

Controller

Programista powinien delegować zadania z interfejsu użytkownika zawsze do kontrolera. Zadaniem kontrolera jest: odbieranie informacji od UI, wykonywanie operacji oraz zwracanie ich wyników do UI. Kontroler powinien delegować zadania w głąb systemu.

Low Coupling

Klasy powinny być od siebie jak najbardziej niezależne. Zmiana w jednej klasie powinna prowadzić do jak najmniejszej ilości zmian w innych klasach. Deleguj odpowiedzialność tak, aby zachowywać jak najmniejsze powiązania klas.

High Cohesion

Obiekt powinien skupiać się na jednej odpowiedzialności. Jego odpowiedzialność nie powinna być rozmyta, gdyż nie wiadomo wtedy dlaczego obiekt istnieje w systemie. Deleguj odpowiedzialność tak, aby obiekt cały czas skupiał się na jednej funkcjonalności.

Polymorphism

Pewien zbiór obiektów może mieć podzbiór zachowań wspólnych oraz podzbiór zachowań odmiennych. Różnicowanie zachowań powinno mieć miejsce na poziomie klas pochodnych z wykorzystaniem mechanizmu metod polimorficznych. Często zamiast takiego różnicowania zachowań stosuje się konstrukcje

if-else co prowadzi do powstawania kodu trudnego w utrzymaniu.

Pure Fabrication

W systemie mogą istnieć operacje, których nie da się przypisać do żadnego obiektu dziedziny. Prowadzi to do powstawania w systemie obiektów, które nie reprezentują żadnego obiektu dziedziny, a kondensują funkcje udostępniane na rzecz innych obiektów. Zapewniają one pewien zbiór usług na rzecz innych obiektów, np.: dostęp do repozytorium. Jeśli nie wiesz jaki obiekt domenowy powinien zawierać wybraną odpowiedzialność, być może zachodzi potrzeba utworzenia takiego sztucznego bytu.

Indirection

W systemie obiekty powinny być jak najmniej ze sobą powiązane. Aby zapewnić *low coupling* często zachodzi potrzeba dodania mediatora w komunikacji między obiektami. Jego zadaniem jest jedynie wymiana informacji między obiektami. Taki obiekt deleguje zadania z jednego obiektu na rzecz drugiego. Dodatkowo projektowanie systemu z użyciem mediatora pozwala respektować prawo Demeter czyli filozoficznie ujmując konieczność rozmawiania tylko z najbliższymi przyjaciółmi. Ogranicza więc znacząco łańcuchowe wywoływanie getterów, aby dostać się do funkcji, którą programista potrzebuje. To wpływa na poprawę hermetyzacji elementów systemu. Model MVC jest dobrym tego przykładem zastosowania tej zasady. Kontroler jest mediatorem, co izoluje interfejs użytkownika od modelu.

Protected Variations

Zasada mówiąca o zakresie modyfikacji w systemie wymaganym przez określoną zmianę.
W systemie powinno się identyfikować punkty niestabilności i budować interfejsy wokół tych punktów. To ograniczy zakres zmian w przypadku, gdyby okazało się, że podany punkt niestabilności systemu wymaga zmian. Programista będzie musiał wtedy tylko dostarczyć inną implementację interfejsu.

GRASP został szczegółowo omówiony w książce Craiga Larman'a „UML i wzorce projektowe Analiza i projektowanie obiektowe oraz iteracyjny model wytwarzania aplikacji”.

SOLID czyli Single responsibility, Open-closed, Liskov substitution, Interface segregation, Dependency inversion

SOLID określa pięć podstawowych zasad programowania obiektowego. Stosowanie zasad SOLID pozwala tworzyć kod, w którym w prosty sposób będzie można wprowadzać zmiany i reagować na zmianę wymagań.

Single Responsibility Principle

Klasa powinna mieć pojedynczy powód do zmiany. Jeśli jest więcej takich powodów powinniśmy przenieść funkcjonalności do kolejnych klas. Klasa musi mieć więc pojedynczą odpowiedzialność, aby wymagała zmian w tylko jednym konkretnym przypadku. Wyobraźmy sobie klasę reprezentującą fakturę. Faktura ma pewną strukturę oraz pewien format wydruku. Może się zmienić dopuszczalna struktura faktury, dodatkowo może zmienić się format wydruku faktury. Widać więc, że tak zamodelowany obiekt ma dwa powody do zmiany, powinien zostać on więc rozłożony na dwie niezależne klasy.

Open Close Principle

Klasy powinny być otwarte na zmiany i zamknięte na modyfikacje. Pisząc klasę programista musi to robić w taki sposób, aby dodając kolejne funkcje nie musiał jej w przyszłości przebudowywać, a jedynie rozszerzać. Ta sama zasada dotyczy bibliotek, modułów.
Programista powinien więc tworzyć abstrakcyjne klasy, a konkretne zachowania umieszczać w klasach konkretnych.

Liskov Substitution Principle

Typ bazowy powinien być zastępowalny w pełni przez typ pochodny. Prosty test czy

wymieniona zasada jest spełniona – odpowiadamy na pytanie czy w każdym miejscu aplikacji gdzie mamy typ bazowy możemy podstawić typ pochodny bez żadnych konsekwencji dla działania aplikacji i bez konieczności jakichkolwiek zmian w kodzie. Jeśli tak zasada Liskov jest spełniona.

Zasada ta jest często łamana, gdyż jest ona nierozumiana. Zgodnie z tą zasadą kwadrat nie jest prostokątem w kontekście ustawiania długości boków i liczenia powierzchni. Wielu programistów uznaje, że kwadrat jest szczególnym typem prostokąta zgodnie z wiedzą wyniesioną z lekcji matematyki. Niestety w kontekście pewnego zbioru odpowiedzialności (funkcji), tak nie jest. Przykład na złamanie zasady Liskov znajduje się poniżej:

```
class Rectangle {
  protected int width;
  protected int height;

  public void setWidth(int width) {
    this.width = width;
  }

  public void setHeight(int height) {
    this.height = height;
  }


  public int getWidth() {
    return this.width;
  }

  public int getHeight() {
    return this.height;
  }

  public int getArea() {
    return this.width * this.height;
  }
}


class Square extends Rectangle {

  public void setWidth(int width) {
    this.width = width;
    this.height = width;
  }

  public void setHeight(int height) {
    this.width = height;
    this.height = height;
  }

}

class LiskovTester {

  private static Rectangle getRectangle() {
    return new Square();
  }

  public static void main(String[] args) {
    Rectangle rec = getRectangle();
    rec.setWidth(5);
    rec.setHeight(10);
    // w tym miejscu programista może nie
    // wiedzieć, że rec to referencja
    // do kwadratu, więc spodziewa się
    // zachowania zgodnego z zachowaniem
    // prostokąta
    // (wartość pola 50,  tymczasem otrzyma pole
    // równe 100)
    System.out.println(r.getArea());
  }
}
```

Interface Segregation Principle

Programista nie powinien być zmuszony do korzystania z interfejsów/ klas, które nie są mu potrzebne. Programista powinien dodawać do interfejsów tylko metody, które ten interfejs rzeczywiście powinien zawierać. Dodawanie nadmiarowych metod prowadzi do późniejszej konieczności implementacji tych metod w klasach konkretnych. Programista powinien tworzyć interfejsy dedykowane do poszczególnych funkcjonalności i zawierające funkcje rzeczywiście wymagane przez zakres odpowiedzialności reprezentowany przez budowany interfejs.

Dependency Inversion Principle

Abstrakcja nie powinna zależeć od szczegółów implementacji. Szczegółowa implementacja powinna zależeć od abstrakcji. Interfejs, klasa abstrakcyjna nie powinna operować na klasie konkretnej. Powinna operować na klasach abstrakcyjnych oraz innych interfejsach. Co więcej klasa konkretna sama również powinna operować na interfejsach oraz ewentualnie innych klasach abstrakcyjnych.

Zasady SOLID zostały szczegółowo opisane w książce Roberta C.Martin'a "Agile Principles, Patterns, and Practices in C#" .

Programowanie poprzez kontrakt (Design by Contract)

Programowanie poprzez kontrakt jest metodyką tworzenia systemów. Każdy komponent, klasa powinna mieć dobrze zdefiniowaną specyfikację interfejsu. Interfejs jest rozumiany jako zbiór

operacji dostępnych dla danej części systemu (w szczególności klasy). Specyfikacja metod interfejsu powinna zawierać: typy danych wejściowych, typy wartości zwracanych (to gwarantuje nam język statycznie typowany), typy rzucanych wyjątków (to gwarantują nam obiektowe języki programowania). Dodatkowo każda operacja interfejsu powinna mieć określone *preconditions*, *postconditions* i *invariants*. Taka specyfikacja interfejsu jest kontraktem. Kontrakt pozwala na ustalenie warunków współpracy obiektu z pozostałymi elementami systemu.

Warunki początkowe (*preconditions*)
Warunki jakie musi spełniać obiekt będący argumentem wywołania metody, aby wykonanie tej metody się powiodło.

Warunki końcowe (*postconditions*)
Warunki jakie musi spełniać obiekt zwracany przez metodę na wyjściu z jej zakresu, jeśli podane warunki początkowe (*preconditions*) będą spełnione.

Niezmienniki (*invariants*) dla obiektów
Dla obiektów są to warunki jakie spełniają właściwości danego obiektu przed i po wywoływaniu dowolnej metody publicznej. Niezmienniki mogą być niespełnione w momencie wywoływania metod prywatnych.

Weryfikacja *preconditions*, *postconditions* oraz *invariants* jest główną zasadą programowania defensywnego, pozwala jak najszybciej odizolować elementy systemu, które zawierają błędne dane lub działają niepoprawnie. Programowanie przez kontrakt zawiera się w nurcie programowania defensywnego na poziomie interfejsów systemu.
W aspekcie programowania w języku obiektowym programowanie poprzez kontrakt możemy rozumieć jako ustalanie wymienionych powyżej ograniczeń dla klas o szerokim zakresie widzialności w systemie, a w szczególności dla interfejsów.

W Javie występuje dodatkowo mechanizm asercji, który wspiera programowanie poprzez kontrakt. Można je wykorzystywać do weryfikacji parametrów wejściowych metod, niemniej jednak chciałbym tutaj przytoczyć zasadę stosowalności asercji:
Asercje stosujemy jedynie do weryfikacji parametrów metod prywatnych, gdyż można je wyłączyć, dodatkowo specyfikacja metod nic nie mówi o istnieniu w niej asercji. Do weryfikacji parametrów metod publicznych powinniśmy stosować i rzucać wyjątek java.lang.IllegalArgumentException lub java.lang.IllegalStateException.

Testowanie jednostkowe mocno wspomaga programowanie przez kontrakt, gdyż pozwala zweryfikować *preconditions*, *postconditions* oraz *invariants*. Ale to jest tylko narzędzie do weryfikacji tych warunków.

Dodatkowo w kontekście projektowania metod dla klas, programowanie kontraktowe wymaga od nas dostarczania metod skupionych na wykonaniu pojedynczej czynności. Jest to zasada separacji zapytania o dane od aktualizacji danych czyli *Command-query separation*.

Command-query separation
Każda metoda powinna wykonywać tylko pojedynczą czynność, więc albo zwraca jakieś dane, albo modyfikuje dane, nigdy nie powinna wykonywać obu tych czynności naraz. Filozoficznie można by powiedzieć, że zadawanie pytań, nie powinno wpływać na udzielaną odpowiedź. Zadanie dwóch tych samych pytań, powinno dać tą samą odpowiedź.

Rozwinięcie tego tematu znajduje się w książce Bertranda Meyer'a „Object-Oriented Software Construction”.

Programowanie przez dziedziczenie versus programowanie przez kompozycję

Dziedziczenie pozwala na korzystanie z

dobrodziejstw polimorfizmu oraz dynamicznego wiązania w czasie wykonania. Dziedziczenie jest dobrym rozwiązaniem, jeśli istnieje konieczność modyfikacji zachowania obiektu. Z dziedziczeniem związane są jednak istotne problemy:

• gdy programista zmienia interfejs nadklasy, wtedy we wszystkich podklasach musi zaimplementować nową metodę interfejsu.

• słaba enkapsulacja, gdyż często obiekty klas pochodnych mają dostęp do danych nadklasy.

Ze względu na dwa powyższe problemy obecnie często zastępuje się dziedziczenie mechanizmem kompozycji. Przy kompozycji łatwiej zmienić interfejs klasy będącej delegatem. Co więcej kompozycja pozwala odsunąć konieczność utworzenia obiektu delegata.

Kilka zasad kiedy stosować kompozycję, a kiedy dziedziczenie:

Dziedziczenie

• Jeśli obiekt A posiada wszystkie zachowania obiektu B, to łączymy je relacją dziedziczenia.

• Jeśli zmieniają się zachowania obiektu, to rozszerzamy obiekt.

• Jeśli relacja pomiędzy obiektami znana jest na poziomie czasu kompilacji, to używamy relacji dziedziczenia.

Kompozycja

• Jeśli obiekt A używa tylko części zachowań obiektu B, to powinniśmy te obiekty połączyć relacją kompozycji.

• Jeśli zmieniają się dane obiektu, to konfigurujemy je, czyli używamy relacji kompozycji.

• Jeśli relacja pomiędzy obiektami znana jest dopiero w czasie wykonania, to używamy relacji kompozycji.

Najważniejsza reguła czy stosować kompozycję czy dziedziczenie wynika z zasady Liskov i brzmi: programista nie powinien używać dziedziczenia, jeśli nie wszystkie metody w klasie pochodnej mają sens funkcjonalny, co oznacza, że obiekty mają różne zbiory zachowań.

Obecnie zapanowała moda na stosowanie kompozycji przy budowaniu systemów obiektowych ze względu na słynną frazę: „Favor Composition Over Inheritance”, która była przedstawiona przez Bandę Czworga. W wyszukiwarce google nie można znaleźć wyników dla frazy: „Inheritance over composition”, natomiast fraza „Composition over inheritance” ma szczególnie dużo wyników wyszukiwania.

Być może wynika to z mody związanej z wzorcami projektowymi oraz stosowaniem kompozycji w ich implementacji.

Niemniej jednak oba podejścia mają sens w szczególnym kontekście tworzenia systemów.

Jeśli chodzi o tworzenie systemów to najczęściej dziedziczenie jest wykorzystywane do definiowania klas abstrakcyjnych oraz interfejsów, co jest prawidłowym podejściem.

Klasy konkretne powinny dostarczać szczegółowych zachowań zdefiniowanych w klasach abstrakcyjnych. Dziedziczenie klas konkretnych powinno być mniej popularne.

Warto również wspomnieć o kompozycji i dziedziczeniu w kontekście testowalności takich relacji. Łatwiej się testuje relację kompozycji dwóch obiektów niż dziedziczenia, ponieważ nie powinno się mock'ować zachowań testowanego obiektu, to zmusza do implementacji wszystkich metod w relacji dziedziczenia.

Doskonały przykład pokazujący kiedy nie stosować dziedziczenia, choć na pierwszy rzut oka może wydawać się to słuszne, zaczerpnąłem ze strony http://pettermahlen.com/2010/08/20/composition-vs-inheritance .

```
public class Animal {}

public class Primate extends Animal {}

public class Human extends Primate {}

public class Chimpanzee extends Primate {}
```

```
public class Kangaroo extends Animal {}

public class Tiger extends Animal {}
```

Modelujemy informację o kończynach oraz funkcje chodzenia dla zwierząt. Podana powyżej hierarchia obiektów wydaje się słusznym podejściem do modelowania relacji pomiędzy gatunkami zwierząt. Niestety ma ona swoje wady. W podanym przykładzie ludzie i szympansy mają dwie nogi i dwie ręce, tygrysy natomiast mają cztery nogi.
Możemy dodać informację o dwóch nogach i rękach na poziomie Primate, a o czterech nogach tygrysa na poziomie Tiger. Niestety dodanie do hierarchii Kangura, powoduje problem, gdyż kangury również mają dwie nogi i dwie ręce. Wtedy dochodzi do duplikacji informacji na poziomie Tiger oraz Kangaroo. Jeśli chodzi o czynność chodzenia, to tutaj również pojawia się problem, gdyż szympansy chodzą na czterech kończynach, a ludzie na dwóch, więc gdzie umieścić logikę chodzenia na dwóch kończynach i również potem na czterech, żeby nie doszło do duplikacji informacji.

Może więc słuszniejszym podejściem będzie stworzenie następującej hierarchii obiektów modelującej powyższy problem.

```
public interface BodyType {}

public class TwoArmsTwoLegs implements BodyType
{}

public class FourLegs implements BodyType {}

public interface Locomotion<B extends BodyType>
{
  void walk(B body);
}

public class BipedWalk implements
Locomotion<TwoArmsTwoLegs> {
  public void walk(TwoArmsTwoLegs body) {}
}

public class Slither implements
Locomotion<NoLimbs> {
  public void walk(NoLimbs body) {}
}

public class Animal {
   BodyType body;
   Locomotion locomotion;
}

Animal human = new Animal(new TwoArmsTwoLegs(),
new BipedWalk());
```

Zdefiniowany powyżej model pozwala w dowolny sposób konfigurować liczbę kończyn oraz sposób chodzenia bez duplikacji informacji. Powyższy przykład pokazuje, że należy być ostrożnym w modelowaniu relacji obiektów i wiązać ją z kontekstem. W podanym przykładzie kontekstem jest modelowanie informacji o kończynach oraz funkcji chodzenia. W innych kontekstach być może relacja dziedziczenia będzie bardziej sensowna dla modelowanego zbioru obiektów.

Kilka słów o refaktoryzacji

Refaktoryzacja zalicza się również do kanonu dobrych praktyk programistycznych, dlatego o niej wspominam w tej publikacji. Polega na poprawianiu czytelności/struktury kodu. Nie będę omawiał tutaj jednak poszczególnych technik refaktoryzacji, a wymienię tylko metaregułę refaktoryzacji.
Refaktoryzacja powinna być prowadzona *bottom-up* czyli: najpierw programista powinien poprawiać nazwy zmiennych/ funkcji, potem wydzielać funkcje w klasie i dopiero w ostatecznej fazie wydzielać nowe klasy i zmieniać współpracę klas. Refaktoryzacja musi być wspierana przez testowanie jednostkowe, tak aby nie wprowadzać błędów do systemu. Dodatkowo refaktoryzacja musi być prowadzona permanentnie w całym czasie życia systemu.

Do kanonu lektur związanych z refaktoryzacją należy książka Martina Fowlera "Refaktoryzacja ulepszanie struktury istniejącego kodu" . W niej programista znajdzie opis technik refaktoryzacji oraz opis niepoprawnych struktur kodu, których występowanie powinno wymuszać stosowanie refaktoryzacji.

Metazasady dobrego programowania

Są one synergią wymienionych wcześniej zasad. Nie są związane z żadnym paradygmatem programowania.

Kod jest pisany/optymalizowany pod kątem łatwego czytania/zrozumienia

To człowiek jest odbiorcą kodu, nie maszyna. Maszyna potrafi poradzić sobie z najbardziej zagmatwanym kodem. Dobry kod jest napisany tak, żeby ktoś, kto nie jest jego autorem, mógł go zrozumieć. Chodzi o to, aby programista koncentrował się na abstrakcji dziedziny, a nie rozszyfrowywał kod czy niuanse języka, które stosuje autor.

Jak w prosty sposób podnieść czytelność naszego kodu:

- stosować proste konstrukcje języka w którym programujemy, najlepiej interjęzykowe, które w każdym języku zachowują się podobnie, aby programista, dla którego podany język jest nowy, również je rozumiał
- stosować wymowne nazwy zmiennych, również lokalnych, klas i interfejsów
- stosować wymowne nazwy funkcji oraz pamiętać aby funkcje wykonywały pojedynczą operację
- komentarze powinny raczej opisywać dlaczego coś zrealizowałem w podany sposób (decyzje implementacyjne)
- utrzymywanie zasady tego samego poziomu abstrakcji dla operacji składowych danej funkcji

Piękno tkwi w prostocie

- jak najmniej wyrażeń warunkowych,
- zamykanie wyrażeń warunkowych w osobne funkcje o wymownej nazwie
- usuwanie martwego kodu

Rozwinięciem tej zasady jest zasada KISS (*Keep it simple stupid*) wprowadzona przez twórców systemu Unix. W projektowaniu interfejsów powyższą zasadę można nazwać zasadą najmniejszego zaskoczenia, czyli fragment kodu powinien robić dokładnie to co ma robić (tworzymy funkcje removeA..., createA..., putA..., setA..., które odpowiadają odpowiednio za usuwanie, dodawanie elementów i nie odpowiadają za żadne inne operacje).

Czasem programista musi wybrać, czy dany fragment kodu napisać z wykorzystaniem wzorca projektowego czy prostej konstrukcji, zgodnie z tą zasadą nie powinien kombinować, niech będą to proste konstrukcje w kodzie źródłowym.

Nie powtarzaj się (*Don't Repeat Yourself*)

Jedno miejsce w systemie, na pojedynczą informację, bo to ułatwia późniejsze zmiany. Jest to według mnie metazasada programowania zorientowanego obiektowo, z niej wywodzi się cały zbiór zasad SOLID. Inaczej zwana również *Single Source Of Truth* (SSOT), każda informacja w systemie powinna być przechowywana dokładnie raz, bo ułatwia to jej modyfikację. Jest ona również aktualna w projektowaniu systemów baz danych, przechowywaniu danych w repozytoriach.

Najważniejsze są projekty interfejsów w systemie

Powinny mieć dużą siłę wyrazu, posiadać operacje o odpowiednim poziomie abstrakcji, nie powinny ujawniać informacji niezwiązanych z danym interfejsem, pozwalać na późniejsze refaktoryzacje w odpowiednio hermetycznym kodzie, a nie w całym systemie. Refaktoryzacja interfejsów jest najbardziej kosztowna, bo wymaga zmian w całym systemie, dlatego programista powinien skupić się w szczególności na ich modelowaniu, aby zmian ich dotyczących było w przyszłości jak najmniej.

Nigdy nie siadaj od razu do kodowania (*Design before Implement*)

Gdy programista dostaje projekt/zadanie, powinien: zrobić krótką refleksję jak je najlepiej zrealizować, jakie wzorce użyć, czy ktoś nie zrobił już czegoś podobnego, poprosić o sugestię ze strony kolegi jak on by to zrealizował. Powinien opisywać słownie funkcję lub klasę, a potem załączać odpowiednie pola i metody, zostawiając opisy w postaci komentarzy, celem poprawienia czytelności kodu.

Zostawiaj kod lepszym niż go zastałeś
Naprawiaj "wybite okna" małymi kroczkami. Robiąc zmianę funkcjonalną widzisz, że w innej części systemu jest coś nieczytelne, poprawiaj to ad hoc. Czasem nawet prosta zmiana nazwy zmiennej poprawia czytelność kodu. Tutaj programista musi być rozważny, najważniejsza jest zmiana funkcjonalna, rób tylko małe refaktoryzacje w tym momencie na poziomie metody, klasy. Nie przebudowuj całej hierarchii klas.

Ważniejsza jest skuteczność niż wydajność w tworzeniu oprogramowania
Staraj się ograniczać dług technologiczny i rób coś najlepiej jak potrafisz w danym momencie biorąc pod uwagę swoje kompetencje, zastosowaną technologię oraz ograniczenia czasowe.

Każda implementacja wymaga uwzględniania kontekstu implementacyjnego
Jeśli tworzę mały system, to nie korzystam z wzorców projektowych, bo one mogą spowodować, że system przestaje być czytelny.

Implementuj system tak, aby spełniał wymagania funkcjonalne na dany dzień
Nie dodawaj specjalnych zachować dla klas/metod, by uczynić je bardziej elastycznym dla przyszłych zmian, bo nie znasz kierunku tych zmian, a koszty tworzenia tak zaprojektowanej metody/klasy są niewspółmierne do przyszłych korzyści. To jest jedna z zasad popularnej obecnie metodyki Scrum.

Najpierw rozwiąż szczegółowy przypadek w najprostszy sposób
Jest to według mnie istotna zasada wynikająca z ewolucyjnego rozwoju systemu.
Jest to rozwinięcie zasady „Implementuj tak system, aby spełniał wymagania funkcjonalne na dany dzień". Przy implementacji systemu nie rozpoczynaj od zbytniego uogólniania swojego zadania. Implementuj najprostsze rozwiązanie, jeśli się okaże, że twoje rozwiązanie wymaga zastosowania w kilku miejscach kodu i w kilku przypadkach użycia, wtedy zaczniesz uogólniać swoje rozwiązanie. Dodatkowo widząc inne przypadki użycia dla swojego najprostszego rozwiązania, poznasz kierunek zmian. Będziesz mógł uogólnić swoje rozwiązanie dla tego konkretnego kierunku zmian.
Ewolucyjny rozwój systemu prowadzi do powstania systemów czytelniejszych i nie zawierających funkcjonalności redundantnych, nie prowadzi do powstawania aplikacji typu *golden hammer,* czyli mającej zastosowanie w niezliczonej liczbie przypadków użycia tylko tak naprawdę w żadnym konkretnym nie spełnia wymagań dobrze.

Nigdy nie optymalizuj systemu pod kątem wydajności do momentu zakończenia prac
Najpierw zmierz „wąskie gardła programu", dopiero potem optymalizuj. Optymalizacja jest niekorzystna prawie zawsze z punktu widzenia czytelności kodu, a czasem zwyczajnie nie jest konieczna. Używaj JProfilera (lub innego narzędzia) do mierzenia rzeczywistych wąskich gardeł systemu, a nie gdybaj, w którym miejscu systemu jest wąskie gardło. Jprofiler pokaże Ci dokładnie wywołanie funkcji trwającej najdłużej i wtedy będziesz mógł ją zoptymalizować.

Twórz konwencje tam, gdzie brakuje czegoś językowi w którym programujesz
Jeśli czujesz, że coś powinno być rozwiązywane w określony sposób przy tworzeniu danego systemu, podziel się tą informacją z innymi.

Programowanie obiektowe jest abstrakcją pozwalająca zapanować nad złożonością
Jest to najważniejsza z wszystkich zasad. Istotą dobrych zasad programowania jest panowanie nad złożonością. Jeśli któraś zasada uniemożliwia nam panowanie nad złożonością, to jest ona barierą i należy ją wyeliminować. Nie zawsze trzymaj się sztywno paradygmatów programowania (czasem lepiej coś zaczerpnąć z innej metodyki)

Nie wymieniłem w publikacji popularnych obecnie podejść TDD, DDD ze względu na to, że stanowią one tylko narzędzie i pokrywają którąś z bardziej ogólnych zasad w aspekcie tworzenia oprogramowania i są one tylko zastosowaniem tych zasad, które niekoniecznie muszą być osiągnięte w ten sposób. Pamiętajmy TDD i DDD są narzędziami.

Wzorce projektowe w praktycznych zastosowaniach

Wzorce projektowe są pewnymi konwencjami tworzenia oprogramowania. Można tworzyć oprogramowanie bez używania wzorców. Wzorce powstały z dwóch powodów: identyfikacji podobnych zbiorów abstrakcji w systemach oraz braków w wyrażaniu danego języka programowania, w szczególności Javy. Pierwszymi wzorcami w oprogramowaniu były wzorce GoF. Potem oczywiście pojawiły się kolejne wzorce JEE oraz mniej popularne wzorce ajaxowe opisane szczegółowo na stronie http://ajaxpatterns.org/. Wzorce powstają w każdej technologii, mogą być one specyficzne dla jakieś technologii, wtedy rozwiązują bolączki najczęściej konkretnej technologii/ metodyki.

Przy omawianiu zasad dobrego programowania nie sposób nie wspomnieć o wzorcach projektowych, które są w nurcie dobrego programowania obiektowego. W publikacji zaprezentuję kilka szczególnie użytecznych wzorców, z którymi spotkałem się we własnej pracy zawodowej. Najpierw prezentuję problem przed jakim może stanąć programista, a potem rozwiązanie tego problemu z wykorzystaniem wzorca. Czasem pokazuję kilka rozwiązań, gdyż jedne można użyć korzystając w modelowaniu z dziedziczenia, a inne przy modelowaniu hierarchii obiektów z wykorzystaniem kompozycji.

Nie znam dokładnie zachowania obiektu, ale znam pewien schemat. Zachowanie szczegółowe dostarczą inni programiści.

Gdybyśmy stosowali dziedziczenie powyższe zachowanie szczegółowe dostarczylibyśmy w klasie pochodnej, tak jak poniżej:

```
public abstract class Counter {

  public int count(int x) {
    return precount() + 10*x + postcount();
  }

  public abstract int precount(int x);

  public abstract int postcount(int x);

}


public class NormalCounter extends Counter {

  public int precount(int x) {
    return x+10;
  }

  public int postcount(int x) {
    return x+20;
  }
}
```

Powyższy wzorzec jest wzorcem metody szablonującej (*template method*).

Gdybyśmy stosowali kompozycję problem można rozwiązać tak jak poniżej:

```
public class Counter {

  private CountingStrategy countingStrategy;

  public Counter(CountingStrategy
countingStrategy) {
    this.countingStrategy = countingStrategy;

  }

  public int count(int x) {
    return countingStrategy.precount() + 10*x +
countingStrategy.postcount();
  }

}

interface CountingStrategy {

  int precount(int x);

  int postcount(int x);

}

public class SpecialPreAndPostCounting
implements CountingStrategy {

  public int precount(int x) {
    return x+10;
  }

  public int postcount(int x) {
    return x+20;
  }
}
```

Powyższe rozwiązanie jest wzorcem strategii (*strategy*).

Mam dwie metody o prawie identycznej konstrukcji poza ograniczoną liczbą wierszy. Złamałem zasadę DRY, bo kod skopiowałem do drugiej metody.

Pierwotna sytuacja została zaprezentowana poniżej:

```
public class MethodTester {

   public void method1() {
     for (Item item : list) {
        method3();
        method4();
        method5();
        method8();
     }
     method6();
   }

   public void method2() {
     for (Item item : list) {
       method3();
       method4();
       method5();
       method7();
     }
     method6();
   }

}
```

Wprowadzona modyfikacja wygląda następująco:

```
public interface Closure {
   public void execute(Item item);
}

public class Closure1 implements Closure {
   public void execute(Item item) {
        method8();
   }
}

public class Closure2 implements Closure {
   public void execute(Item item) {
        method7();
  }

}

public class MethodTester {

   public void method(Closure closure) {
       for (Item item : list) {
            method3();
            method4();
            method5();
            closure.execute(item);
       }
       method6();
   }

  public static void main(String [] args) {
     new MethodTester().method(new Closure1());
     new MethodTester().method(new Closure2());
  }

}
```

Przykład jest prymitywny, możemy w nim wstawić wyrażenie if i zrobić jedną metodę, ale często jest wiele miejsc metody, które możemy chcieć zmienić. Wzorzec ten jest szczególnie często wykorzystywany w bibliotekach apache-commons (CollectionUtils, ListUtils). Obiekty typu Transformator, Predicate to nic innego jak polecenie (*command*). Istnienie wzorca polecenia wynika z braku w języku Java domknięć znanych z języków funkcyjnych. Tutaj czynność jest wstrzykiwana jako obiekt.

Potrzebuję wielodziedziczenia w Javie.

```
public class ClassB {
   public void methodB() {
    // do something
   }
}

public class ClassC {
   public void methodC() {
    // do something
  }

}

public class ClassA extends Class B {

   private ClassC classC;

   public ClassA(ClassC classC) {
     this.classC = classC;
   }

   public void methodB() {
       // do something
   }

  public method() {
    classC.methodC();
  }
}
```

Zaprezentowany powyżej sposób rozwiązania bazuje na połączeniu dziedziczenia oraz kompozycji. Jest to wzorzec most (*bridge*) w możliwe najprostszej postaci.

Mam jakąś strukturę danych i potrzebuję nawigować po tej strukturze, wykonując operacje na niej.

Najprostsze rozwiązanie jakie przychodzi do głowy jest następujące:

```
public class Node {

  private Node parent;
  private Node left;
  private Node right;
  private int value;
```

```
  public void printAllNodes() {
       // print node info
  }

  public void sumAllNodesValues() {
     // sum all nodes values
  }
}
```

Powyższy projekt ma swoje wady w momencie, gdy programista potrzebuje dodać nową operację na strukturze, musi dodawać kolejną metodę do tej struktury. Można stworzyć rozwiązanie bardziej generyczne:

```
interface NodeVisitor {

  void visit(Node node);

}

public class PrintNodeVisitor implements
NodeVisitor {

  public void visit(Node node) {
    System.out.println(node.getValue());
  }
}

public class SumNodeVisitor implements
NodeVisitor {

  private int sum;

  public void visit(Node node) {
    sum = sum + node.getValue();
  }
}

public class Node {
  private Node parent;
  private Node left;
  private Node right;
  private int value;

  // move through structure
  public void walkThrough(NodeVisitor visitor) {
     visitor.visit(value);
     if (left != null) {
       left.walkThrough(visitor);
     }
     if (right != null) {
       right.walkThrough(visitor);
     }
  }

  public getValue() {
    return this.value;
  }
}

class Tester {

  public void main(String[] args) {
    Node node = intializeNode(); // in this
method structure is created
    node.walkThrough(new PrintNodeVisitor());
    node.walkThrough(new SumNodeVisitor());
  }
}
```

Zaprezentowane rozwiązanie izoluje strukturę od operacji, które jej dotyczą i nosi nazwę odwiedzającego (*visitor*).

Potrzebuję zmodyfikować funkcjonalność istniejącej metody dodając pewien zbiór operacji do obecnej funkcjonalności.

Można to zrobić używając dziedziczenia:

```
public class SimpleClass {

  public void method() {
    // do something
  }
}

public class SimpleClassWithLogging extends
SimpleClass {

  public void method() {
     super.method();
     log();
  }

  public void log () {
    // logging here
  }
}
```

Możemy to zrobić używając kompozycji:

```
public class SimpleClass {

  public void method() {
    // do something
  }
}

public class SimpleClassWithLogging {

  private SimpleClass simpleClass;

  public SimpleClassWithLogging(SimpleClass
simpleClass) {
    this.simpleClass = simpleClass;
  }

  public void method() {
     simpleClass.method();
     log();
  }

  public void log () {
    // logging here
  }
}
```

Zastosowane powyżej rozwiązanie to nic innego jak wzorzec dekoratora (*decorator*) w najprostszej postaci.

Określony interfejs (np.: w bibliotece zależnej) muszę zamienić na inny wymagany przez program.

```
interface OldInterface {
  String method2();
```

```
}

public class OldClass implements OldInterface {

  public String methodOld() {
     return "methodOld";
  }
}

interface NewInterface {
  String methodNew();
}

public class NewClass implements NewInterface {

  private OldClass oldClass;

  public NewClass(OldClass oldClass) {
    this.oldClass = oldClass;
  }

  public String methodNew() {
    return oldClass.methodOld();
  }
}
```

Zaproponowana powyżej transformacja interfejsów jest wzorcem projektowym adapter (*adapter*).

Potrzebuję złożonej struktury danych.

Zarówno klasa agregująca jak i klasa reprezentująca pojedynczy obiekt mają wspólny interfejs. Dlatego można je traktować tak samo. Klasa agregująca może agregować pojedyncze obiekty, jak również inne klasy agregujące. Prezentowany tutaj wzorzec jest kompozytem (*composite*).

```
interface Component {
  String method1();
}

public class Leaf implements Component {

  public String method1() {
     return "method1";
  }
}

public class Compound implements Component {

  private Component component;

  public Compound(Component component) {
    this.component = component;
  }

  public String method1() {
    System.out.println("Compound method1");
    return component.method1();
  }
}
```

Potrzebuję *fluent interface'u* reprezentującego zbiór operacji z danej dziedziny

Czym jest fluent interface:

- reprezentuje zbiór operacji z jakiejś dziedziny
- pozwala tworzyć mikrojęzyki wyrażane w składni języka w kontekście którego jest tworzony
- pozwala na *method chaining* (wartością zwracaną musi być referencja na ten obiekt)

```
interface FormFieldBuilder {
  FormFieldBuilder required();
  FormFieldBuilder maxLength();
  FormFieldBuilder validator(Validator
validator);
  FormFieldBuilder defaultValue(String value);
  FormField build();
}

public class TextFieldBuilder extends
FormFieldBuilder {

  FormFieldBuilder required() {
    // field is set required
    return this;
  }

  FormFieldBuilder maxLength() {
    // max length is set
    return this;
  }

  FormFieldBuilder validator(Validator
validator) {
    // add validator to field
    return this;
  }

  FormFieldBuilder defaultValue(String value) {
    // add default value
    return this;
  }

  FormField build() {
    // build field finally
  }

}

public class FormBuilder {

  public add(FormFieldBuilder FieldBuilder) {
    // add field builder to form
  }

  public void buildForm() {
    // build form
  }

  public abstract build();
}

public class MyFormBuilder extends FormBuilder {

  public void build() {
      add(new TextFieldBuilder()
        .required()
        .maxLength(100)
        .defaultValue("aaa"));
  }
}
```

Zaprezentowany powyżej sposób tworzenia obiektu jest wzorcem budowniczego (*builder*).

Nie chcę modelować przepływu sterowania w aplikacji z użyciem wielu skomplikowanych instrukcji warunkowych.

Rozwiązaniem jest maszyna stanowa, gdzie zachowania systemu są zależne od stanu systemu.

```
interface State {
  State transit();
}

public class StateA implements State {

  public void compute() {
    System.out.println("compute StateA");
  }

  public State transit() {
     compute();
     // choose and return next state nextState
     return nextState;
  }

}

public class StateB implements State {

  public void compute() {
    System.out.println("compute StateB");
  }

  public State transit() {
     compute();
     // choose and return next state nextState
     return nextState;
  }

}

public class StateMachine {

  public static StateA a = new StateA();
  public static StateB b = new StateB();

  private State currentState;

  public StateMachine() {
    this.currentState = a;
  }

  public void doSomething() {
    currentState = currentState.transit();
  }
}
```

Wzorzec stanu (*state*) albo bardziej maszyny stanowej wbrew pozorom można wykorzystywać w wielu aplikacjach, również aplikacjach internetowych. W modelu MVC, w którym najczęściej programista buduje aplikację www, kontroler możemy zaimplementować jako maszynę stanową i zamodelować przejścia pomiędzy ekranami za pomocą przepływu sterowania. Podejście takie stosuje Spring webFlow. Jest ono o tyle wygodne, że kontroler zawiera zbiór operacji niezbędnych do przygotowania danych wyjściowych (baza danych) na podstawie danych wejściowych (żądanie http, sesja), natomiast możliwe przejścia pomiędzy ekranami (logikę dopuszczalnych przejść) modeluje przepływ sterowania.

Na zakończenie chciałbym wspomnieć krótko o dwóch wzorcach: MVC oraz IoC.

MVC

MVC jest wzorcem, który pozwala odizolować logikę od operacji związanych z interfejsem użytkownika. Wzorzec ten jest powszechny w aplikacjach www, większość szkieletów do budowy aplikacji również wykorzystuje to podejście. Model jest reprezentacją dziedziny biznesowej lub problemu jaki rozwiązuje nasz program. Widok jest odpowiedzialny za prezentację danych. Kontroler odpowiada za odbiór danych z interfejsu użytkownika, przetwarzanie i delegację zadań do modelu.

IoC

Ostatnio modnym i użytecznym wzorcem stały się kontenery Inversion of Control i związane z nim wstrzykiwanie zależności. Programista pobiera gotowe instancje obiektów z takiego kontenera, nie martwiąc się o cały proces tworzenia obiektu i jego zależności.
Stosowanie IoC wyeliminowało praktycznie stosowanie wzorca singleton, gdyż dla aplikacji www tworzenie komponentów aplikacyjnych pozwala na utworzenie tylko pojedynczej instancji obiektu w całym systemie. IoC wyeliminował również praktycznie stosowanie wzorca fabryki oraz metody fabrykującej. Wiele framework'ów korzysta z automatycznego wstrzykiwania zależności dla przykładu Spring. Innym popularnym kontener IoC jest picocontainer (www.picocontainer.org).

W publikacji nie zostały omówione następujące wzorce równie często wymieniane w literaturze:

- prototyp (*prototype*) (operacja clone w Javie)
- fasada (*fasade*)

- front controller
- pyłek (*flyweight*)
- proxy
- łańcuch odpowiedzialności (*chain of reponsisbility*)
- iterator
- memento
- mediator
- null object
- specyfikacja (*specification*)
- interpreter
- obserwator (*publish/subscribe* inaczej *observer*)
- DAO i DTO

Do podstawowej literatury poruszającej tematykę wzorców projektowych należą:

„Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku" napisana przez Gang of Four oraz "Agile Principles, Patterns, and Practices in C#" napisana przez Roberta C. Martin'a. Wzorce JEE zostały dokładnie opisane w książce Martina Fowler'a „Patterns of Enterprise Application Architecture".

W internecie również znajduje się wiele stron opisujących wzorce projektowe:

- http://en.wikipedia.org/wiki/Design_pattern_%28computer_science%29
- http://sourcemaking.com/design_patterns
- http://www.oodesign.com/design-principles.html
- http://ajaxpatterns.org/Patterns

Podsumowanie

Programowanie jest jedną z najbardziej abstrakcyjnych i najtrudniejszych czynności jaką wykonuje człowiek. Wymienione tutaj zasady mają pomóc programiście tworzyć dobre systemy. Z punktu widzenia tworzenia systemów dwie zasady są ważne: czytelność kodu dla innych programistów oraz łatwość wprowadzania zmian do kodu w przyszłości. Reguły tutaj przytaczane mają pomóc tworzyć takie właśnie systemy.

Wiele tych reguł jest ogólnych, dla początkujących programistów mogą one być nieczytelne. Jednak publikacja jest podsumowaniem paradygmatu obiektowego i ma pomóc doświadczonym programistom weryfikować czy kierunek implementacji systemu jest właściwy.

Nie zapominajmy, że istnieją inne paradygmaty: programowanie funkcyjne i programowanie w logice. Nie lekceważmy ich, nie zamykajmy się w jednym świecie.

Na zakończenie chciałbym przedstawić literaturę opisującą dobre praktyki programistyczne, z którą każdy programista powinien się zapoznać, gdyż w nich są zawarte rozwinięcia zasad zawartych w tym artykule. Część z nich została już wymieniona w poszczególnych sekcjach. Do tego zbioru należą:

- "Czysty kod" Robert C.Martin
- „Kod doskonały" Steve MCConnell
- "Agile Principles, Patterns, and Practices in C#" Robert C.Martin
- "Refaktoryzacja ulepszanie struktury istniejącego kodu" M.Fowler
- "Pragmatyczny programista" A.Hunt
- „Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku" Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides
- „Patterns of Enterprise Application Architecture" Martin Fowler
- „UML i wzorce projektowe Analiza i projektowanie obiektowe oraz iteracyjny model wytwarzania aplikacji" Craig Larman
- „Test Driven Development: By Example" Kent Beck

Łukasz Baran
Inżynier oprogramowania w firmie e-point S.A., gdzie zajmuje się projektowaniem i implementacją systemów internetowych w technologii JEE. Absolwent Wydziału Elektroniki i Technik Informacyjnych Politechniki Warszawskiej oraz Szkoły Głównej Handlowej. Interesuje się platformą Java SE, Java EE oraz dobrymi praktykami programistycznymi.
Kontakt: lukasz.baran@yahoo.com