Jarosław Dzikowski 273233 Wrocław, 07.04.16r.

Lista 3

WAŻNE: Do zadań, do których należy zapewnić kod znajdują się pliki ZadanieX.cs zawierające kod. Do zadań, do których należy zapewnić diagram znajdują się pliki ZadanieX.pdf zawierające diagramy. Opracowanie zadań oraz odpowiedzi na różne pytania znajdują się w tym pliku.

Zadanie 1

W zadaniu należy zobrazować 5 wybranych zasad GRASP działającym kodem. W moim kodzie mamy do czynienia z płaszczyzną, na której istnieją różnego rodzaju figury. Płaszczyzna jest reprezentowana przez klasę Surface. Płaszczyzna zawiera fabrykę figur ShapeFactory tworzącą nowe figury typu Shape. Shape jest klasą bazową, z której dziedziczą klasy Circle oraz Rectangle.

Chcemy umieć obliczać pola oraz obwody figur. Do tego implementujemy wirtualne metody klasy Shape odpowiednio ComputeArea() oraz ComputePerimeter(), które następnie nadpisywane są przez metody klas potomnych klasy bazowej Shape. Całość służy temu, by wyliczać sumaryczny pola i sumaryczny obwód wszystkich figur na płaszczyźnie.

Użyte zasady GRASP:

1. Creator: Mamy fabrykę ShapeFactory przechowującą oraz tworzącą na zamówienie odpowiednie obiekty typu Shape.
2. Low Coupling: W celu jak największego wykorzystania powtórnego kodu, fabryka figur zawiera factory workerów realizujących interfejs IFactoryWorker. Każdy taki pracownik fabryki wie, które figury jest w stanie tworzyć oraz posiada kod tworzący w odpowiedni sposób dane figury. Pozwala to na wielką elastyczność fabryki, dodawanie i usuwanie pracowników fabryki (Czyli de facto modyfikowanie stanu konkretnych obiektów, które fabryka może produkować). Fabryka odwołuje się do metod swoich pracowników.
3. Polymorphism: Dla różnych figur sposób liczenia pola lub obwodu jest różny. Dlatego każda figura dziedziczy z klasy bazowej Shape i nadpisuje metody klasy bazowej ComputeArea() oraz ComputePerimeter().
4. Information Expert: Przydzielamy zobowiązania obiektom, które posiadają dane konieczne do realizacji zobowiązań. Figury same liczą swoje pola i obwody, bo mają wymagane do tego informacje. Analogicznie klasa Surface reprezentująca płaszczyznę liczy sumaryczne pole oraz sumaryczny obwód, ponieważ posiada listę figur znajdujących się na płaszczyźnie.
5. Protected Variations: Ta zasada GRASP rozwiązuje problem projektowania obiektów, tak by ich zmienność nie wywierała szkodliwego wpływu na inne elementy. Posłużmy się przykładem z mojego kodu: mamy fabrykę ShapeFactory produkującą różne figury na różne sposoby. Gdybyśmy pisali fabrykę w sposób "naiwny" (Tj. fabryka poproszona o utworzenie jakiegoś przedmiotu sprawdza instrukcją switch oraz paroma if'ami czy i w jaki sposób może utworzyć obiekt figury), to gdybyśmy chcieli dodać/usunąć figurę do/z fabryki lub dodać nowy sposób tworzenia jakiejś figury, mielibyśmy problem. Musielibyśmy modyfikować cały kod. Podejście do problemu fabryki z implementacją pracowników jest dokładnie tym, o czym mówi zasada Protected Variations: znalezioną zmienność, która nas "boli" otaczamy solidnym i niezmiennym interfejsem (IFactoryWorker).

Zadanie 2

Starą hierarchię rozbijemy na trzy klasy:

1. Klasę Document przechowującą dane dotyczące dokumentu i umożliwiającą podgląd danych (GetData()).
2. Klasę DocumentFormatter odpowiedzialną za formatowanie dokumentu.
3. Klasę ReportPrinter odpowiedzialną za wydruk dokumentu.

Dlaczego akurat podział na tyle klas? Przede wszystkim nalezy zaważyć, że formatowanie dokumentu, to zadanie na tyle skomplikowane, że zasługuje na osobną klasę. Oczywiście musimy mieć jakąś osobną klasę, która będzie wiedziałą jak wydrukować dokument. Musimy gdzieś przechowywać dane dotyczące dokumentu. Przechowywanie ich w klasie odpowiedzialnej na formatowanie lub drukowanie jest oczywiście bez sensu. Stąd wynika konieczność utorzenia osobnej klasy odpowiedzialnej za przechowywanie danych dokumentu.

Oczywiście zasada Single Responsibility nie oznacza, że zawsze musimy tworzyć osobną klasę dla każdej metody. Rozważmy przykład klasy Samochód z wykładu zawierającej m.in. metody takie jak:

- rusza()
- zatrzymuje()
- pobieraOlej()
- kieruje()
- sprawdzaOlej()
- zmieniaOpony()
- myje()

Widać, że jest tutaj wiele metod nie pasujących do samochodu: samochód na pewno nie kieruje się sam, na pewno sam nie sprawdza oleju, nie zmienia opon samodzielnie oraz nie myje się sam. Te metody należy rozdzielić do osobnych klas. Natomiast niewymienione wcześniej metody (rusza(), zatrzumuje(), pobieraOlej()), to funkcjonalności, za które z pewnością odpowiada samochód. Oczywiście jest ich więcej niż jedna i nie powinny one trafić do osobnych klas.

Zadanie 3

Kod z treści zadania można zmodyfikować w sposób następujący: do TaxCalculatora dodajemy pole Tax typu Decimal wraz z odpowiednimi getterami i setterami. Ponieważ CashRegister komponuje się z TaxCalculatora oraz chcemy być w stanie modyfikować stawkę podatku, musimy dodać getter i setter do pola Tax naszego TaxCalculatora.

Aby być w stanie wypisywać przedmioty na paragonie w dowolnych kolejnościach, dodajemy kolejny parametr do metody CashRegister::PrintBill(). Tym nowym parametrem jest itemSorter typu IitemSorter. IitemSorter to interfejs dodany przeze mnie zawierający tylko jedną funkcję: SortItems sortującą tablicę przedmiotów.

W celach demonstracyjnych utworzyłem dwie klasy realizujące interfejs IItemSorter: AlphabeticalSorter sortujący przedmioty względem nazwy rosnąco oraz PriceSorter sortujący przedmioty względem ceny rosnąco.

Taki kod jest otwarty na rozbudowę (Możemy zmieniać stawkę podatku oraz wymyślać dowolne sposoby porządkowania przedmiotów na paragonie) oraz zamknięty z punktu widzenia klienta.

Zadanie 4

Z pewnością kod podany w treści zadania łamie zasadę Liskov Substitution Principle. Warunkiem wyjścia setterów klasy Rectangle jest to, że długość drugiego boku prostokąta jest taka sama jak przed zmianą długości wybranego boku. Oczywiście warunek ten w przypadku klasy Square nie jest spełniany, co łamie zasadę LSP. Mamy do czynienia z pozornym dziedziczeniem. Z pewnością geometrycznie kwadraty są prostokątami, ale już z punktu widzenia projektowania obiektowego nie.

W załączonym kodzie zostało opisane rozwiązanie problemu przedstawionego w zadaniu. Problem ten można było rozwiązać na dwa sposoby:

1. Zarówno prostokąt jak i kwadrat są wielokątami, więc tworzymy klasę Polygon, z której dziedziczą Rectangle oraz Square. Settery klasy Rectangle pozostają bez zmian, natomiast klasa Square ma tylko jedno pole, którym jest rozmiar boku (Size). Do klasy AreaCalculator dodajemy metodę CalculateArea pobierającą argument typu Square obliczającą pole kwadratu oraz metodę liczenia pola dla dowolnego Polygon'a w jakiś sposób wspólny dla wielokątów (np. poprzez triangulację wielokąta).
2. Usunąć z hierarchii klas klasę Square i jako kwadraty używać tylko prostokątów posiadających pola Width i Height o takich samych wartościach.

Wybrałem pierwszy sposób rozwiązania problemu. Widać, że możemy zastępować obiekty klasy bazowej obiektami klasy potomnej, zatem zasada LSP jest zachowana.

Zadanie 5

W tym zadaniu chcemy znaleźć przykład z biblioteki standardowej jakiegoś języka programowania łamiący zasadę Interface Segregation Principle. Wyjaśnienie zasady znajduje się w następnym zadaniu.

Jako mój przykład wybrałem klasę ReadOnlyCollection z biblioteki standardowej języka C#. Nazwa klasy jasno tłumaczy, czym ona jest. Spełnia ona interfejs ICollection, co zmusza klasę do implementacji metod, których nigdy nie będzie używała (Add(), Remove(), Clear() - Skoro kolekcja jest tylko do odczytu, to nie będziemy modyfikowali jej zawartości.). To już z definicji łamie zasadę ISP.

Zadanie 6

Na początku sformułujmy zasadę Single Responsibility Principle: "Klasa ma tylko jedną odpowiedzialność". Jest to całkiem proste do zrozumienia: jeśli klasa robi zbyt wiele rzeczy, to być może powinniśmy rozbić tę klasę na kilka prostszych klas, z których każda będzie odpowiedzialna za jedną rzecz. Na przykładzie klasy Samochód opisanej w zadaniu 2 widzimy, że samochód powinien być odpowiedzialny za ruszanie, zatrzymywanie oraz pobieranie oleju, a resztę funkcjonalności powinniśmy rozdzielić na osobne klasy (Mechanik powinien sprawdzać stan oleju oraz zmieniać opony, Kierowca powinien kierować samochodem, a Myjnia powinna myć samochód.). Równoważna definicja SRP mówi, że "Klasa powinna być modyfikowana tylko z jednego powodu".

Zasada Interface Segregation Principle mówi, że "Klient nie powinien być zmuszany do zależności od metod, których nie używa". Chodzi o to, że jeśli chcemy utworzyć klasę spełniającą metodę A i B jakiegoś interfejsu, to nie powinniśmy być zmuszeni także do implementacji wszystkich metod od A do Z tegoż interfejsu. Bardziej przyziemnie: jeśli chcemy użyć młotka, to nie powinniśmy musieć zbudować najpierw całej Castoramy ze wszystkimi możliwymi narzędziami

budowlanymi.

Widać, że obie zasady są podobne. W szczególności moglibyśmy interpretować zasadę ISP w ten sposób, że powinniśmy rozbijać duże interfejsy na mniejsze, w którym każdy jest odpowiedzialny za jedną funkcjonalność. Jednakże, różnica między zasadą SRP a zasadą ISP tkwi w tym, że pierwsza dotyczy projektowania kodu od strony "wewnętrznej" (projektanta modułu), natomiast druga dotyczy punktu widzenia klienta korzystającego z kodu.

Podsumowując, obie zasady reprezentują te same idee, lecz z dwóch przeciwnych stron (strona projektanta modułu i strona klienta korzystającego z modułu).