UML. Inżynieria oprogramowania. Wydanie II

Autor: Perdita Stevens
Tłumaczenie: Ireneusz Jakóbik
ISBN: 978-83-246-0805-8
Tytuł oryginału: Using UML: Software Engineering with Objects and Components (2nd Edition)
Format: B5, stron: 304

Praktyczny podręcznik do nauki języka UML

- Jak zaprojektować dobry system?
- Jak poprawnie tworzyć i odczytywać modele?
- Jak w praktyce stosować UML i poprawić jakość projektowanych produktów?

W świecie informatyki dobry projekt to często więcej niż połowa sukcesu, a wraz ze wzrostem popularności obiektowych języków programowania UML – ujednolicony język modelowania przeznaczony do reprezentacji elementów w analizie obiektowej i programowaniu obiektowym – stał się podstawowym narzędziem do tworzenia modeli. Dlatego też trudno wyobrazić sobie dobrego informatyka, który nie potrafi przygotować poprawnego projektu w tym języku lub odczytać modelu utworzonego przez kogoś innego.

„UML. Inżynieria oprogramowania. Wydanie II" to przystępny podręcznik dla studentów i informatyków pragnących nie tylko poznać ujednolicony język modelowania, ale przede wszystkim nauczyć się korzystać z niego w kontekście inżynierii oprogramowania. Czytając go, dowiesz się, jak powinien wyglądać dobry system, poznasz składnię i funkcje języka UML, a przedstawione studia przypadku pozwolą Ci zobaczyć, jak używać go do projektowania praktycznych rozwiązań.

- Projektowanie systemów bazujących na komponentach
- Wprowadzenie do obiektowości
- Efektywny proces projektowania
- Opracowywanie modeli klas
- Przygotowywanie modeli przypadków użycia
- Tworzenie diagramów interakcji, stanów i aktywności
- Przygotowywanie diagramów struktury i wdrożeń
- Stosowanie komponentów i wzorców
- Dbanie o jakość procesu i produktu
- Praktyczne przykłady projektowania systemów

Jeśli chcesz tworzyć oprogramowanie najwyższej jakości, zacznij od dobrego projektu

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

rozdział 2

Koncepcje obiektów

W tym rozdziale będziemy kontynuować udzielanie odpowiedzi na pytanie „jakie są dobre systemy?", opisując zorientowany obiektowo paradygmat, który zawdzięcza swoją popularność częściowo temu, że wspiera on projektowanie dobrych systemów. Korzystanie jednak ze zorientowania obiektowego nie jest ani niezbędne, ani wystarczające do budowania dobrych systemów. Udzielimy odpowiedzi na następujące pytania:

- Czym jest obiekt?
- W jaki sposób porozumiewają się obiekty?
- Jak jest zdefiniowany interfejs obiektu?
- Co obiekty mają wspólnego z komponentami?

Pod koniec omówimy dziedziczenie, polimorfizm i wiązanie dynamiczne.

Treści zawarte w tym rozdziale odnoszą się do standardowych, nowoczesnych, opartych na klasach, obiektowo zorientowanych języków, takich jak Java, C# i C++. Większość rozdziału można odnieść też do języków Smalltalk, Eiffel, CLOS, Perl5 i innych obiektowo zorientowanych języków, z którymi można się spotkać, jednak dokładne ich porównywanie wykracza poza zakres tej książki.

W niniejszym rozdziale są zawarte pytania na temat sposobów, w jaki języki programowania używane przez *Czytelnika* udostępniają omawiane obiektowo zorientowane funkcje. Jeśli Czytelnik zna już jakiś język programowania, powinien być w stanie udzielić odpowiedzi na te pytania. Jeśli nie, warto aby przeczytał ten rozdział do końca, poznał podstawy wybranego języka, a następnie powrócił do pytań w celu sprawdzenia swojej wiedzy na temat sposobów realizowania przez język zorientowania obiektowego. W pytaniach język programowania używany przez Czytelnika będziemy w skrócie nazywać „Twoim językiem".

2.1. Czym jest obiekt?

Pojęciowo **obiekt** jest elementem, z którym można wchodzić w interakcje: można wysyłać do niego różne **komunikaty**, a on będzie reagował. **Zachowania** obiektu zależą od jego bieżącego **stanu** wewnętrznego, który może ulegać zmianie, na przykład na skutek reakcji na

otrzymany komunikat. Ważne jest, z którym obiektem zachodzi interakcja, i dlatego odniesienie do obiektu następuje za pośrednictwem jego nazwy. Obiekt ma zatem swoją **tożsamość**, która odróżnia go od wszystkich innych obiektów. Obiekt jest więc elementem posiadającym swoje zachowanie, stan i tożsamość; w taki sposób charakteryzuje go Grady Booch [7]. Rozważymy te aspekty bardziej szczegółowo.

Przedmiot

Pojęcie przedmiotu jest bardziej interesujące, niż może się to początkowo wydawać. Obiekt nie jest tylko elementem systemu; reprezentuje on w systemie koncepcję przedmiotu. Obiekt może być na przykład reprezentacją prawdziwego przedmiotu, takiego jak zegar, tablica rozdzielcza albo klient. W rzeczywistości obiekty pojawiły się jawnie po raz pierwszy w języku Simula, w którym reprezentowały obiekty z realnego świata — te, które były symulowane. Oczywiście, aby systemy programistyczne ogólnego przeznaczenia mogły być tworzone w oparciu o obiekty, obiekty nie muszą reprezentować fizycznie istniejących rzeczy. Do kwestii sposobu rozpoznawania obiektów powrócimy w rozdziale 5.

P: Które z elementów następującej listy mogą być obiektami: młotek, kula, proces chemiczny, miłość, mgła, rzeka, złość, kot, szarość, szary kot?

Stan

Stan obiektu to wszystkie dane aktualnie w nim zapisane. Zwykle obiekt ma wiele nazwanych **atrybutów** (**zmiennych obiektu**, **danych składowych**), z których każdy ma określoną wartość. Niektóre z atrybutów mogą podlegać zmianom, to znaczy ich wartości mogą się zmieniać. Obiekt reprezentujący na przykład klienta może mieć atrybut o nazwie **adres**, którego wartość ulegnie modyfikacji po przeprowadzce klienta. Inne atrybuty mogą być niezmienne, czyli stałe. Obiekt reprezentujący klienta może mieć atrybut, którego wartością będzie numer identyfikacyjny klienta, pozostający bez zmian przez cały czas trwania życia obiektu. W większości współczesnych obiektowo zorientowanych języków programowania zestaw atrybutów przypisanych obiektowi nie może ulec zmianie podczas trwania życia tego obiektu, chociaż wartości atrybutów mogą się zmieniać.

Zachowanie

Zachowanie oznacza sposób, w jaki obiekt działa i reaguje w sensie zmian swojego stanu i przekazywania komunikatów. Obiekt **rozumie** niektóre **komunikaty**, co znaczy, że może je odbierać i zgodnie z nimi postępować. Zestaw komunikatów rozumianych przez obiekt, podobnie jak i zestaw atrybutów, jest zwykle stały. Sposób, w jaki obiekt reaguje na komunikat, może zależeć od aktualnych wartości jego atrybutów. Dzięki temu, nawet jeśli świat zewnętrzny nie ma zagwarantowanego bezpośredniego dostępu do atrybutów (a zazwyczaj właśnie tak jest), może pośrednio wpływać na ich wartości. W taki sposób wartości atrybutów decydują o **stanie** obiektu.

Tożsamość

To zagadnienie może być trochę trudne. Idea jest taka, że obiekty są definiowane nie tylko przez bieżące wartości swoich atrybutów. Istnienie obiektu jest ciągłe. Wartości atrybutów mogą na przykład ulec zmianie, zwykle w odpowiedzi na komunikat, ale to będzie ciągle ten sam obiekt. Odniesienie do obiektu następuje zazwyczaj przez jego **nazwę** (wartość zmiennej w programie-kliencie, na przykład `Klient`), ale nazwa obiektu to nie to samo, co sam obiekt, ponieważ obiektowi można nadać kilka różnych nazw (co może być zaskakujące dla programistów funkcyjnych, a niektórzy z nich twierdzą, że zorientowanie obiektowe można zrealizować w czysto funkcyjny sposób; jednak tego zagadnienia nie będziemy poruszać w niniejszej książce).

2.1.1. Przykład

Rozważmy obiekt o nazwie `mojZegar`, który rozumie komunikaty `podajCzas` i `ustawCzasNa(07:43)`, i w ogólności `ustawCzasNa(nowyCzas)` dla sensownych wartości parametru `nowyCzas`. Można to zrealizować, tworząc interfejs, który ogłosi, że przyjmuje komunikaty w postaci `podajCzas` i `ustawCzasNa(nowyCzas: Czas)`, gdzie `Czas` jest typem[1], którego elementy są sensownymi wartościami wspomnianego parametru `nowyCzas`.

W jaki sposób zaimplementować taką funkcję? Świat zewnętrzny nie musi tego wiedzieć (informacja powinna być ukryta), ale być może obiekt ma atrybut `Czas`, którego wartość zwraca w odpowiedzi na komunikat `podajCzas` i który jest ustawiany na wartość `nowyCzas` po otrzymaniu przez obiekt komunikatu `ustawCzasNa(nowyCzas)`. Jest też możliwe, że komunikat jest przekazywany do innego obiektu, znanego obiektowi `mójZegar`, i że to właśnie ten inny obiekt pracuje z komunikatami.

2.1.2. Komunikaty

Z poprzedniego przykładu można wysnuć kilka wniosków na temat komunikatów. W skład komunikatu wchodzi słowo kluczowe nazywane **selektorem**. W przykładzie selektorami są `podajCzas` i `ustawCzasNa`. Komunikat może, ale nie musi, zawierać jeden lub więcej argumentów, które są wartościami przekazywanymi obiektowi, podobnie jak są przekazywane wartości w przypadku wywoływania funkcji. W powyższym przykładzie `07:43` jest argumentem, stanowiącym część komunikatu `ustawCzasNa(07:43)`. Zazwyczaj dla danego selektora istnieje pojedyncza „prawidłowa" liczba argumentów, które powinny znajdować się w komunikacie rozpoczynającym się tym od tego selektora. Nie można oczekiwać, że obiekt `mojZegar` zrozumie komunikat `ustawCzasNa` bez żadnego argumentu lub w postaci `ustawCzasNa(4, 12:30)` i tak dalej. Akceptowalne wartości argumentów również są ustalane w interfejsie obiektu[2].

---

[1] Albo klasą; klasy omówimy w podrozdziale 2.1.4.

[2] Podając klasę (patrz podrozdział 2.1.4) lub niekiedy tylko interfejs, do których powinien należeć obiekt przekazywany jako argument, lub jego typ, jeśli należy do typu podstawowego, takiego jak `int` albo `bool` — szczegóły zależą od konkretnego języka.

P: Jaka jest składnia przesyłania komunikatów w Twoim języku?

Warto zauważyć, że wartości, na przykład wartości atrybutów obiektu albo wartości argumentów przesyłane w komunikatach, nie muszą należeć do typu podstawowego (takiego jak znaki, liczby całkowite i tak dalej. Rodzaje typów podstawowych zależą od konkretnego języka). Mogą to być również obiekty.

P: Jakie są podstawowe typy w Twoim języku?

Jedną z czynności, którą może wykonać obiekt w odpowiedzi na komunikat, jest wysłanie komunikatu do innego obiektu. Jeśli tak ma się stać, to powinien on w jakiś sposób poznać nazwę tego obiektu. Prawdopodobnie wartością jednego z jego atrybutów mógłby być na przykład obiekt, którego nazwa byłaby wykorzystywana podczas wysyłania komunikatu.

P: Niech `o` będzie obiektem. Do jakich innych obiektów może wysyłać komunikaty obiekt `o`, oprócz obiektów, dla których może być wartością ich atrybutów?

Warto zwrócić uwagę na to, że podczas wysyłania komunikatu do obiektu nie wiadomo, jaki kod zostanie w wyniku wykonany, gdyż informacja o tym jest hermetyzowana. Będzie to istotne pod koniec niniejszego rozdziału, kiedy omówimy wiązanie dynamiczne.

2.1.3. Interfejsy

Publiczny interfejs definiuje komunikaty, które będą przyjmowanie przez obiekt niezależnie od miejsca ich pochodzenia. Zazwyczaj interfejs zapisuje selektory tych komunikatów, łącznie z niektórymi informacjami na temat wymaganych argumentów oraz wartości, która ewentualnie mogłaby być zwrócona. Jak wspomnieliśmy w rozdziale 1., korzystne byłoby, gdyby interfejs zawierał jakiś rodzaj specyfikacji dotyczącej skutków wysłania komunikatu do obiektu. Informacje tego typu są jednak podawane jedynie w komentarzach i dokumentacji. W większości języków programowania atrybuty mogą być zawarte również w publicznym interfejsie obiektu. Umieszczenie atrybutu `x` w interfejsie jest równoważne z zadeklarowaniem, że fragment danych tego obiektu, `x`, może być widziany i zmieniany z zewnątrz.

Zagadnienie do przedyskutowania 9

Jeśli korzystasz z języka, który nie zezwala na umieszczanie atrybutów w interfejsie publicznym, to w jaki sposób możesz osiągnąć taki sam efekt, używając komunikatów w celu uzyskania dostępu do danych? Czy są jakieś zalety stosowania takiego rozwiązania nawet w językach, które dopuszczają używanie atrybutów w interfejsie? A wady?

Często jest tak, że nie każdy element systemu powinien mieć możliwość wysyłania do obiektu wszystkich komunikatów, które dany obiekt mógłby zrozumieć. Dlatego też obiekt może zwykle zrozumieć pewne komunikaty, które nie są częścią jego publicznego interfejsu. Jak już na przykład wyjaśniliśmy w rozdziale 1., struktura danych używana przez moduł powinna być hermetyzowana, co jest zgodne z założeniem, że interfejs publiczny obiektu nie powinien zazwyczaj zawierać atrybutów.

Obiekt zawsze może wysłać **sam do siebie** komunikat, który będzie w stanie zrozumieć. Wysyłanie komunikatów samemu sobie może wydawać się dziwne lub zbyt skomplikowane,

jednak powód takiego postępowanie wyjaśnimy w dalszej części niniejszego rozdziału przy okazji omawiania wiązania dynamicznego.

Zazwyczaj obiekt ma przynajmniej dwa interfejsy: publiczny, z którego może korzystać dowolny element systemu, oraz obszerniejszy interfejs prywatny, z którego może korzystać zarówno sam obiekt, jak i inne uprzywilejowane elementy systemu. Czasami może być jeszcze przydatny interfejs pośredni, udostępniający więcej możliwości niż interfejs publiczny, ale mniej niż interfejs prywatny. Czasami w danym kontekście jest potrzebna tylko część publicznego interfejsu obiektu, co powoduje wzrost liczby obiektów, które potencjalnie mogłyby być używane zastępczo. Może być zatem przydatne zapisywanie naprawdę potrzebnych funkcji w mniejszym interfejsie, zamiast umieszczania ich w całym interfejsie publicznym obiektu. Z tego względu obiekt może mieć (lub **realizować**) więcej niż tylko jeden interfejs. I odwrotnie — wiele różnych obiektów może realizować taki sam interfejs.

W tej książce określenie „interfejs" zawsze będzie oznaczać interfejs publiczny. Jeśli będzie inaczej, zostanie to wyraźnie określone.

2.1.4. Klasy

Do tej pory omawialiśmy obiekty, jak gdyby każdy z nich był osobno definiowany, każdy miał swój własny interfejs i sposób sprawdzania, który inny obiekt może wysyłać do niego komunikaty. Oczywiście taki sposób tworzenia typowego systemu nie jest rozsądny, ponieważ obiekty mają ze sobą wiele wspólnego. Jeśli przedsiębiorstwo ma 10 000 klientów i potrzebne jest zbudowanie systemu z obiektami reprezentującymi każdą z tych osób, to wszystkie obiekty reprezentujące klientów będą posiadały wiele wspólnych cech. Ich zachowania powinny być spójne, aby programiści i osoby opiekujące się systemem mogły je zrozumieć. Potrzebna będzie **klasa** obiektów reprezentująca klientów. Klasa opisuje zestaw obiektów pełniących w systemie takie same funkcje.

W opartych na klasach, zorientowanych obiektowo językach programowania każdy obiekt należy do klasy, a klasa obiektu determinuje jego interfejs[3]. Obiekt `mojZegar` może należeć do klasy `Zegar`, której interfejs publiczny określa, że każdy obiekt klasy `Zegar` będzie udostępniał operację `ustawCzasNa(nowyCzas: Czas)`. Znaczy to, że będzie rozumiał komunikaty z selektorem `ustawCzasNa` i pojedynczym argumentem, który jest obiektem klasy (typu) `Czas`.

W rzeczywistości nawet sposób, w jaki obiekt reaguje na komunikaty, jest zdeterminowany przez jego klasę, łącznie z wartościami atrybutów obiektu. **Metoda** jest określonym fragmentem kodu, w którym są zaimplementowane operacje. W interfejsie obiektu widoczne jest jedynie to, że obiekt wykonuje operację; metoda, w której jest zaimplementowana funkcja, jest ukryta.

Podobnie zestaw atrybutów posiadanych przez obiekt jest zdeterminowany przez klasę, chociaż oczywiście wartości przyjmowane przez atrybuty obiektu nie są określone w klasie i mogą ulegać zmianom. Być może na przykład obiekty należące do klasy `Zegar` mają prywatny atrybut o nazwie `czas`. Jeśli taki sam komunikat zostanie wysłany do dwóch obiektów tej samej klasy, w obu przypadkach zostanie wykonana ta sama metoda, chociaż efekt jej wykonania może być znacząco różny, gdy atrybuty obiektów będą mieć inne wartości.

---

[3] W rzeczywistości obiekt może należeć do więcej niż jednej klasy, z których klasa **najbardziej szczególna** decyduje o jego interfejsie.

Podsumowując, obiekty tej samej klasy mają takie same interfejsy. Publiczne i prywatne interfejsy klasy `Zegar` mogłyby być opisane następująco:

```
- czas : Czas
+ podajCzas() : Czas
+ ustawCzasNa (nowyCzas: Czas)
```

W skład interfejsu publicznego wchodzą wiersze oznaczone znakiem +, a w skład interfejsu prywatnego także wiersz ze znakiem –.

P: W jaki sposób definiuje się klasę, taką jak `Zegar` w Twoim języku? Czy interfejs i implementacja są definiowane oddzielnie? Jak jest nazywany interfejs publiczny, a jak prywatny (może właśnie: publiczny i prywatny)? Czy są może jeszcze jakieś inne możliwości? Co one oznaczają?

Proces tworzenia nowego obiektu należącego do klasy `Zegar` jest nazywany **tworzeniem egzemplarza** klasy `Zegar`, a powstały w wyniku obiekt **egzemplarzem** klasy `Zegar`. To właśnie dlatego zmienne, których wartości należą do obiektu i które mogą ulegać zmianie w czasie życia obiektu, są nazywane zmiennymi egzemplarza. Utworzenie obiektu oznacza utworzenie nowego egzemplarza klasy posiadającego własny stan i tożsamość, które będą pozostawać spójne w obrębie wszystkich obiektów klasy. Proces tworzenia zazwyczaj obejmuje także nadanie wartości atrybutom. Wartości mogą zostać określone w zleceniu utworzenia lub nadane przez klasę obiektu jako domyślne wartości początkowe. W językach takich jak C++ i Java klasa odgrywa bezpośrednią rolę podczas tworzenia nowych obiektów — w rzeczywistości wykonuje całą pracę, nie ograniczając się jedynie do roli szablonu. Dlatego klasa może być często uważana za fabrykę **obiektów**.

P: W jaki sposób jest tworzony w Twoim języku nowy obiekt danej klasy?

W wielu językach klasy mogą zachowywać się, jak gdyby same były obiektami. **Klasa** `Klient` mogłaby mieć atrybut `liczbaEgzemplarzy`, który byłby liczbą całkowitą zwiększaną za każdym razem, gdy tworzony jest nowy **obiekt** typu `Klient`. Takie rozwiązanie konsekwentnie przyjęto w języku Smalltalk, a w językach C++ i Java można odnaleźć jego ślady. W Smalltalk na przykład nowy obiekt typu `Klient` zostałby utworzony przez wysłanie komunikatu (przykładowo `new`) do **klasy** `Klient`. Reakcją klasy byłoby utworzenie nowego obiektu i zwrócenie go do metody wywołującej.

Zagadnienie do przedyskutowania 10

Zastanów się, co by było, gdyby konsekwentnie stosować takie rozwiązanie. Stwierdziliśmy już, że każdy obiekt ma swoją klasę. Jeśli więc klasa `k` zostanie uznana za obiekt, to co będzie klasą dla `k`? I co będzie klasą tej klasy?

P: Czy Twój język posiada takie funkcje? Jakie dokładnie?

Dygresja: po co istnieją klasy?

Dlaczego nie wystarczą obiekty, które posiadają wymagany stan, zachowanie i tożsamość?

Klasy w zorientowanych obiektowo językach programowania służą dwóm celom. Po pierwsze, są wygodnym sposobem na opisanie zbioru (klasy) obiektów posiadających takie same właściwości. W językach zorientowanych obiektowo klasy są używane przede wszystkim dla

wygody. Nie ma na przykład potrzeby przechowywania kopii kodu opisującego zachowania obiektu w każdym obiekcie, chociaż można wyobrażać sobie, że w każdym obiekcie znajduje się jego kod. Zamiast tego programista może jednorazowo napisać definicję klasy, a kompilator utworzy pojedynczą reprezentację klasy i pozwoli obiektom na uzyskiwanie dostępu do reprezentacji klasy w celu pobierania potrzebnego kodu.

Należy pamiętać, że taka wygoda wcale nie jest czymś błahym. Zasadniczym celem jest przecież uczynienie systemów łatwiejszymi do zrozumienia, aby ich projektowanie i pielęgnacja były jak najbardziej tanie, łatwe i niezawodne. Sprawienie, że podobne obiekty dzielą ten sam projekt oraz implementację, jest ważnym krokiem w tym kierunku.

W taki sposób przejawia się zasada[4], która jest tak ważna w inżynierii oprogramowania — nie tylko w odniesieniu do kodu — że ją wykrzyczymy:

PISZ TYLKO RAZ!

Oznacza to, że jeśli dwie kopie jakiegoś produktu inżynierii oprogramowania, na przykład fragment kodu, fragment diagramu czy fragment tekstu pochodzący z dokumentu, powinny być spójne, to nie powinny **istnieć** w dwóch egzemplarzach. Za każdym razem, gdy ta sama informacja jest zapisywana w więcej niż jednym miejscu, nie tylko marnuje się wysiłek na powtórne robienie tego samego, zamiast zrobienia czegoś nowego, ale też naraża się na kłopoty związane z pielęgnacją, ponieważ dwie wersje nigdy nie będą zsynchronizowane bez żadnych starań. Sztuka stosowania tej zasady polega na stwierdzeniu, czy dwie kopie informacji są rzeczywiście takie same, czy też zdarzyło się, że potencjalnie różne informacje są akurat w danym momencie identyczne. W tym drugim przypadku skopiowanie i wklejenie odpowiedniego fragmentu jest jak najbardziej prawidłowe. W pierwszym przypadku niechęć do powielania informacji w więcej niż jednym miejscu jest zaletą. Należy tak postąpić wyłącznie wtedy, gdy istnieją ku temu dobre powody.

Zagadnienie do przedyskutowania 11

Czy ta zasada oznacza, że powielanie danych w rozproszonych bazach danych jest złym pomysłem? Dlaczego?

Zauważmy, że istnieją inne sposoby na tworzenie wielu podobnych obiektów, pisząc ich kod tylko raz. Można na przykład zdefiniować prototyp obiektu, a następnie definiować kolejne obiekty jako będące „takie jak tamten, ale z następującymi różnicami". W takim przypadku również wystarczyłoby przechowywanie tylko jednej kopii kodu. Mniej więcej w taki sposób postępuje się, programując w eleganckim, opartym na prototypach, języku Self. Języki programowania oparte na klasach zdominowały jednak obecnie zorientowanie obiektowe i zapewne tak już pozostanie.

Po drugie, w większości zorientowanych obiektowo językach programowania klasy są używane w taki sam sposób, jak w wielu innych językach są wykorzystywane typy — w celu określenia, jakie wartości są dopuszczalne w określonych kontekstach, umożliwiając tym samym kompilatorowi (oraz, co jest równie ważne, osobie czytającej kod) zrozumienie intencji programisty i sprawdzenie, czy pewne rodzaje błędów nie wystąpią w programie.

[4] Na którą kładzie nacisk zwłaszcza Kent Beck.

Wiele osób uważa, że klasy i typy oznaczają tę samą rzecz. Rzeczywiście, jest to wygodne i często nie prowadzi do żadnych nieporozumień, niemniej taki sposób myślenia jest błędny. Należy pamiętać, że klasa definiuje nie tylko to, jakie komunikaty zrozumie obiekt, co tak naprawdę wystarcza, aby sprawdzać, czy jest on akceptowalny w danym kontekście. Określa również czynności obiektu wykonywane w odpowiedzi na komunikaty.

2.2. Jaki to ma związek z celami poprzedniego rozdziału?

Szukamy modułów, najlepiej nadających się do wielokrotnego używania wymiennych modułów, które mogłyby być dobrymi komponentami. Gdzie można je znaleźć w systemie obiektowo zorientowanym?

Poszczególne obiekty mogłyby być uważane za moduły, ale nie byłyby one dobre. W systemie znajduje się bowiem zazwyczaj wiele różnych modułów, które są blisko spokrewnione pojęciowo. Gdyby każdy obiekt był uważany za odrębny moduł, to albo istniałyby wprowadzające w błąd nieścisłości między pojęciowo powiązanymi modułami, albo potrzeba zachowania spójności powodowałaby tworzenie silnych skojarzeń.

Klasy powinny być słabo skojarzonymi, wysoce spójnymi modułami. Można zatem mieć nadzieję (być może na próżno, co przedyskutujemy w następnym podrozdziale), że będzie realne uzyskanie korzyści przedstawionych w rozdziale 1., a mianowicie niskiego kosztu i krótkiego czasu opracowywania, łatwości pielęgnacji i wysokiej niezawodności. Zalety te nie są cechą charakterystyczną zorientowania obiektowego, chociaż jest ono obecnie najlepiej poznaną ścieżką do nich prowadzącą.

Kolejna uznana zaleta zorientowania obiektowego należy do innej kategorii. Naturalne jest postrzeganie otaczającego świata w kategoriach obiektów. Warto przypomnieć sobie, że głównym wymogiem stawianym przed dobrym systemem jest spełnianie oczekiwań użytkowników i że prawidłowe uchwycenie tych oczekiwań oraz śledzenie ich zmian należą do najtrudniejszych zadań związanych z tworzeniem systemów. Ważne jest, aby model systemu (model domeny problemu i zachodzących procesów) był zgodny z modelem użytkownika. Wydaje się, że obiekty z domeny problemu rzadziej ulegają zmianom, a same zmiany są mniej dramatyczne, niż jest to w przypadku działania systemu wymaganego przez użytkownika. Jeśli zatem system jest zbudowany w oparciu o te obiekty, to zmiany w systemie prawdopodobnie będą mniej rewolucyjne, niż gdyby system powstał w oparciu o bardziej ulotne aspekty funkcjonalne. Podsumowując, należy mieć nadzieję, że system będzie zgodny z modelem świata użytkownika, dzięki czemu będzie możliwe:

- łatwiejsze i dokładniejsze uchwycenie wymagań;
- lepsze nadążanie za zmianami wymagań użytkownika;
- tworzenie systemów, które prowadzą interakcje w bardziej naturalny sposób. Łatwiejsze jest zaimplementowanie na przykład interfejsu użytkownika, który pozwala użytkownikowi na interakcję z plikiem, zegarem albo drukarką jako całością (co ma sens z punktu widzenia użytkownika, ponieważ te przedmioty są podobne do obiektów z rzeczywistego świata), jeśli są one spójnymi obiektami także z punktu widzenia systemu. Z tego powodu interfejsy użytkownika były jednymi z pierwszych głównych obszarów, w których zorientowanie obiektowe stało się popularne.

Należy jednak zaznaczyć, że te argumenty nie są bezdyskusyjne. W niektórych przypadkach, takich jak na przykład systemy obsługi zamówień, podejście zorientowane obiektowo wydaje się bardzo naturalne, w innych (na przykład kompilatory) taka argumentacja jest mniej przekonująca.

W każdym jednak przypadku gdy zorientowanie obiektowe osiąga swój cel, zwykle **nie** jest tak z powodu charakterystycznych cech zorientowania obiektowego. Jest tak, ponieważ:

- podstawami zorientowania obiektowego są modularność, hermetyzacja i abstrakcja,
- zorientowane obiektowo języki programowania sprawiają, że osiąganie celów staje się względnie proste, ponieważ istnieje uzasadnione prawdopodobieństwo, że oczywisty sposób zrobienia czegoś jest również dobrym sposobem na zrobienie tego.

Zorientowanie obiektowe jest religią: to **ludzie** zostają zorientowani w stronę obiektów.

Zagadnienie do przedyskutowania 12

Istnienie jakich konsekwencji, dobrych i złych, może sugerować stwierdzenie, że „zorientowanie obiektowe jest religią"?

2.2.1. Co wspólnego z komponentami mają obiekty?

Euforia towarzysząca zorientowaniu obiektowemu mogłaby czasami sugerować, że każda klasa jest automatycznie komponentem, nadającym się do wielokrotnego używania. Oczywiście nie jest to prawdą. W rozdziale 1. omówiliśmy powody, dla których możliwość wielokrotnego używania komponentu nie jest po prostu faktem o komponencie jako takim, ale zależy od kontekstu projektu i proponowanego sposobu jego powtórnego użycia. Innym ważnym czynnikiem jest to, że struktura klasy może być zbyt szczegółowa, aby mogła być efektywnie wielokrotnie wykorzystywana. Aby na przykład można było wielokrotnie korzystać z pojedynczej klasy w sposób praktyczny i łatwy, należałoby pisać programy w tym samym języku i sięgać do zgodnej architektury. Nie zawsze jest to niewykonalne. Istnieją powszechnie używane biblioteki klas, które odniosły sukces, i ich biblioteki mogą być uważane za komponenty do wielokrotnego używania. Często jednak określenie komponentu odnosi się do grupy powiązanych klas. Im więcej wysiłku włoży się w dostosowanie komponentu do kontekstu, tym więcej korzyści powinien przynieść dany komponent, jeśli ten wysiłek ma być opłacalny.

Celowo rozmyliśmy kwestię tego, czy komponenty są zbudowane z obiektów, czy z klas; nie dokonaliśmy jeszcze wyraźnego rozróżnienia między typem a egzemplarzem. Powód jest taki, że wziąwszy pod uwagę szeroką definicję komponentu, oba te przypadki mogą być prawdziwe. Klasa albo zbiór klas może być komponentem na poziomie kodu źródłowego. Kontekst, w którym jest używany komponent, korzysta z udogodnień danego języka w celu tworzenia obiektów klasy. W innych przypadkach komponent może realizować dostęp do określonego, gotowego obiektu jakiejś klasy, realizując również dostęp do funkcji związanych z tworzeniem kolejnych obiektów tej klasy. Należy pamiętać, że sam komponent ma dobrze zdefiniowany interfejs. Jeśli komponent składa się z obiektów i (lub) klas, to interfejs komponentu zwykle nie pozwala na dostęp do wszystkiego, co znajduje się w interfejsach obiektu i wewnątrz klas. Komponent sam stara się być hermetyzowaną abstrakcją i prawdopodobnie jest właściwe, że ukrywa część lub całość swojej wewnętrznej struktury — na przykład czy i jak składa się z obiektów i klas.

Zagadnienie do przedyskutowania 13

Skorzystaj z sieci Web i innych dostępnych zasobów w celu odszukania popularnych komponentów architektury, takich jak JavaBeans, CORBA albo .NET. Co to jest architektura? Jaką wewnętrzną strukturę może mieć komponent i jak jest zdefiniowany interfejs komponentu?

2.3. Dziedziczenie

Do tej pory omówiliśmy jedynie pojęciowe podstawy tego, co czasami jest nazywane **projektem opartym na obiektach**. W tym podrozdziale weźmiemy pod uwagę **dziedziczenie**, które będąc pojęciowym elementem leżącym u podstaw obiektowo **zorientowanego** projektu, jest niczym lukier dla pączka. Jak wskazuje metafora, pojęcie to jest uważane za atrakcyjne i ważne, ale tak naprawdę jest mniej treściwe niż to, co do tej pory opisaliśmy w książce.

W niniejszym podrozdziale rozpatrzymy dziedziczenie jako techniczną właściwość języków zorientowanych obiektowo, użyteczną (w ograniczonym zakresie) podczas niskopoziomowego wielokrotnego wykorzystywania. W rozdziale 5. omówimy korzystanie z dziedziczenia w modelowaniu.

Poniższy przykład został zaczerpnięty ze studium przypadku z rozdziału 15. W rozważanym systemie istnieje klasa, której obiekty reprezentują wykładowców, oraz klasa, której obiekty reprezentują opiekunów studiów. Opiekun studiów jest kimś w rodzaju wykładowcy, a dokładniej jest wykładowcą, który oprócz pełnienia zwykłych obowiązków jest odpowiedzialny także za nadzorowanie postępów poszczególnych studentów. W systemie `OpiekunStudiow` powinien rozumieć takie same komunikaty jak `Wykladowca` i dodatkowo powinien być w stanie odpowiedzieć na komunikat `podopieczni`, zwracając zbiór obiektów typu `Student`.

Załóżmy, że klasa `Wykladowca` została już zaimplementowana i teraz należy zaprojektować klasę `OpiekunStudiow`. Załóżmy też (wyłącznie dla skonkretyzowania przykładu), że używany jest język podobny do C++, w którym interfejs klasy i jego implementacja są przechowywane w oddzielnych plikach. Co jest potrzebne do zaimplementowania nowej klasy tak szybko jak to tylko możliwe? Można skopiować i wkleić kod definiujący interfejs klasy `Wykladowca` do nowego pliku z interfejsem i dodać u dołu opis dodatkowego komunikatu. Podobnie można skopiować do pliku z implementacją nowej klasy kod, w którym zaimplementowano reakcję obiektów klasy `Wykladowca` na komunikat, a następnie dokonać niezbędnych zmian. Jak już jednak wspomnieliśmy, wielokrotne używanie kodu przez jego kopiowanie ma wady. Najważniejsza z nich jest taka, że w przypadku konieczności dokonania zmian w zduplikowanym kodzie (na przykład wykryto błąd w programie) należy wprowadzić zmiany w dwóch miejscach zamiast w jednym. Jest to uciążliwe i prawdopodobnie doprowadzi do błędów.

Zamiast tego obiektowo zorientowane języki programowania dopuszczają utworzenie nowej klasy `OpiekunStudiow` na podstawie istniejącej klasy `Wykladowca`. Wystarczy zadeklarować, że klasa `OpiekunStudiow` jest **podklasą** klasy `Wykladowca`, a następnie dopisać jedynie to, co ma związek z dodatkowymi atrybutami lub operacjami klasy `OpiekunStudiow`[5].

Co więcej, ponieważ w zorientowaniu obiektowym wiedza na temat kodu, jaki zostanie wykonany przez obiekt po tym, gdy otrzyma on komunikat, jest hermetyzowana, istnieje

[5] W niektórych językach, a zwłaszcza w języku Eiffel, można w podklasach usuwać atrybuty i operacje nadklas, jednak takie postępowanie nie jest zazwyczaj uważane za dobry zwyczaj.

możliwość, aby w klasie `OpiekunStudiow` zaimplementować w odpowiedzi na komunikat inne zachowania niż te, które są zaimplementowane w klasie `Wykladowca`. Oznacza to, że w klasie `OpiekunStudiow` można **przesłonić** niektóre z metod klasy `Wykladowca`. Oprócz dziedziczenia metod po klasie `Wykladowca`, klasa `OpiekunStudiow` może zawierać nową metodę implementującą taką samą operację, dla której klasa `Wykladowca` miała już metodę. Kiedy obiekt klasy `OpiekunStudiow` otrzyma komunikat z żądaniem wykonania operacji, wywołana metoda będzie wyspecjalizowaną wersją metody udostępnianej przez klasę `OpiekunStudiow`.

> Podklasa jest rozszerzoną, wyspecjalizowaną wersją swojej nadklasy. Zawiera operacje oraz atrybuty nadklasy i możliwie jeszcze inne, dodatkowe elementy.

P: W jaki sposób jest definiowana podklasa w Twoim języku? Czy klasa może być bezpośrednią podklasą kilku klas (dziedziczenie wielokrotne), czy nie?

Terminologia

Mówi się, że:

- klasa `OpiekunStudiow` dziedziczy po klasie `Wykladowca`,
- klasa `OpiekunStudiow` jest **podklasą** (lub **klasą pochodną**) klasy `Wykladowca`,
- klasa `OpiekunStudiow` jest **specjalizacją** klasy `Wykladowca`,
- klasa `OpiekunStudiow` jest **bardziej wyspecjalizowana** niż klasa `Wykladowca`,
- klasa `Wykladowca` jest **nadklasą** (lub **klasą bazową**) klasy `OpiekunStudiow`,
- klasa `Wykladowca` jest **generalizacją** klasy `OpiekunStudiow`.

Wszystkie powyższe określenia mają niemal to samo znaczenie. Często pojęcia podklasy i nadklasy są używane przy opisie konkretnych relacji pomiędzy pojęciami reprezentowanymi w klasach. Zazwyczaj zachodzi między nimi związek. Związki pojęciowe jednak niekoniecznie muszą być widoczne w strukturze klasy. Przeciwnie, niektórzy programiści używają dziedziczenia w strukturze klasy w przypadkach, w których nie ma pojęciowego związku dziedziczenia pomiędzy klasami (nie zawsze można polecać takie rozwiązanie. Może być ono wygodne w krótszym okresie czasu, jednak niemal zawsze spowoduje nieporozumienia po upływie dłuższego czasu).

Czy można powiedzieć, że klasa `OpiekunStudiow` należy do klasy `Wykladowca`? Można, jeśli myśli się o klasach jak o zbiorach obiektów. Gdy mówi się o obiektach klasy `Wykladowca`, z pewnością chce się uwzględnić także obiekty klasy `OpiekunStudiow` — po prostu ciągłe pisanie o „klasie `Wykladowca` i dowolnych podklasach" jest zbyt niewygodne. Co więcej, zasada, że obiekt podklasy powinien być **zastępowalny** obiektem nadklasy, jest fundamentalna. Za każdym razem więc, gdy będzie mowa o obiektach należących do klasy, będzie to oznaczać także obiekty należące do podklas. Z drugiej strony bardzo wygodna byłaby możliwość mówienia o „klasie tego obiektu", jak gdyby należał on wyłącznie do jednej klasy. Przez określenie „klasa" obiektu będziemy więc rozumieć **najbardziej wyspecjalizowaną** klasę, do której należy dany obiekt. Oznacza to, że twierdzenie, iż zachowanie obiektu jest określone przez jego klasę, pozostaje prawdziwe.

2.4. Polimorfizm i wiązanie dynamiczne

Te dwa określenia są często w środowisku zorientowanym obiektowo używane zamiennie. W rzeczywistości oznaczają one całkiem różne koncepcje. Ogólnie w językach programowania jedno pojęcie może istnieć bez drugiego, chociaż w językach zorientowanych obiektowo są one spokrewnione i mają związek z dziedziczeniem.

Przede wszystkim należy zauważyć, że sam fakt, iż jedną klasę można definiować na podstawie innej klasy, korzystając ponownie z jej kodu, nie musi mieć żadnego wpływu na to, w jaki sposób kod-klient będzie traktował obiekty tych klas. Możliwy byłby język, w którym dziedziczenie byłoby **tylko** sposobem na wielokrotne wykorzystywanie kodu. Można by zdefiniować klasę `OpiekunStudiow` na podstawie klasy `Wykladowca`, lecz cały kod klienta (a także kompilator) musiałby dokładnie znać klasę każdego obiektu, z którym by wchodził w interakcję. Technicznie rzecz biorąc oznacza to, że można by utworzyć język z takim rodzajem dziedziczenia, w którym występowałyby wyłącznie typy **monomorficzne**. Języki zorientowane obiektowo radzą sobie jednak lepiej.

Polimorfizm

Określenie to, wywodzące się z języka greckiego, oznacza **mający wiele kształtów**. W językach obiektowo zorientowanych odnosi się do sytuacji, w której obiekt może należeć do jednego z kilku typów. W obiektowo zorientowanej sytuacji zmienna polimorficzna mogłaby odnosić się (w różnych przypadkach) do obiektów kilku różnych klas. Funkcja polimorficzna może pobierać argumenty różnych typów. Powiedzieliśmy już, że obiekt podklasy powinien być stosowany w każdej sytuacji, w której mógłby zostać użyty obiekt nadklasy. Powinno to oznaczać w szczególności, że obiekt podklasy powinien być akceptowany jako wartość zmiennej, jeśli jest akceptowany jako obiekt klasy bazowej. Innymi słowy, w obiektowo zorientowanym języku **każda** zmienna jest polimorficzna — przynajmniej w tym ograniczonym znaczeniu. Podobnie, jeśli jakaś funkcja (na przykład metoda) może pobierać argument klasy B, powinna być w stanie pobrać argument dowolnej klasy C, która jest podklasą klasy B; tak więc również każda funkcja jest polimorficzna w tym ograniczonym sensie. Funkcja, która na przykład oczekuje jako argumentu obiektu `Wykladowca`, nie powinna zgłosić błędu, jeśli jako argument otrzyma obiekt `OpiekunStudiow`.

Zagadnienie do przedyskutowania 14

Czy w Twoim języku (w C albo w typologicznie funkcjonalnym języku programowania) jest realizowany polimorfizm? Czy na polimorfizm są nałożone większe, czy mniejsze ograniczenia niż w polimorfizmie obiektowo zorientowanym, który został tu opisany? Czy jest on przydatny? W jaki sposób?

Polimorfizm pozwala na znaczne ograniczenie duplikacji kodu. Swoje prawdziwe możliwości pokazuje jednak w powiązaniu z związaniem dynamicznym.

Wiązanie dynamiczne

Słowo „wiązanie" w określeniu **wiązanie dynamiczne** (albo też późne wiązanie — oba te określenia są używane zamiennie) odnosi się do procesu identyfikowania kodu, który powinien zostać wykonany na skutek otrzymanego komunikatu. Zrozumienie tego będzie łatwiejsze po rozważeniu kilku przykładów.

Załóżmy, że interfejs obiektu `Wykladowca` zawiera funkcję `mozeWykonac (obowiazek: Obowiazek)`, która jako argument pobiera obiekt reprezentujący obowiązek administracyjny, a w wyniku zwraca wartość logiczną, której wartością jest `prawda`, jeśli wykładowca może wypełnić dany obowiązek. Przyjmijmy też założenie (trochę nierealne), że `OpiekunStudiow` jest w stanie zrobić wszystko, a tym samym że zawsze powinien zwracać wartość `prawda`. Znaczy to, że podklasa `OpiekunStudiow` przeimplementowała (przesłoniła) funkcję `mozeWykonac`, prawdopodobnie zastępując skomplikowany kod sprawdzający informacje na temat zainteresowań lub doświadczenia wykładowcy prostym kodem, który zawsze zwraca wartość `prawda`.

Załóżmy teraz, że `wykladowcy` są zmienną odnoszącą się do zbioru obiektów klasy `Wykladowca`. W tym zbiorze mogą być oczywiście zawarte także obiekty typu `OpiekunStudiow`, ponieważ jeśli do zmiennej `wykladowcy` można dodawać obiekty klasy `Wykladowca`, to można do niej dodawać również obiekty klasy `OpiekunStudiow`. Przypuśćmy, że klient-zadanie próbujące znaleźć kogoś do zorganizowania seminarium wykonuje kod w rodzaju poniższego pseudokodu:

```
dla kazdy o w wykladowca
 o.mozeWykonac(organizacjaSeminarium)
```

Warto zwrócić uwagę na zapis z kropką w wierszu z wysyłanym komunikatem, który jest wspólny dla wielu języków programowania. Wyrażenie `o.mozeWykonac(organizacjaSeminarium)` oznacza wysłanie komunikatu `mozeWykonac(organizacjaSeminarium)` do obiektu `o`. Może ono być również używane w celu reprezentowania wartości, którą obiekt `o` zwróci klientowi po przetworzeniu komunikatu.

Przyglądając się kodowi można przypuszczać, że obiekt `o` należy do klasy `Wykladowca`. Warto przypomnieć ideę, że w obiekcie hermetyzowane jest jego zachowanie, a klasy wprowadzono jedynie z potrzeby zwięzłości zapisu. Jeśli obiekt `o` znajduje się wyłącznie w klasie `Wykladowca`, to powinien zostać wykonany kod obiektu `Wykladowca` dla komunikatu `mozeWykonac`, czyli jego zachowanie w reakcji na komunikat. Co powinno się stać, gdy pętla `dla` osiągnie obiekt `o`, który w rzeczywistości należy do klasy `OpiekunStudiow`? Oczywiście w takim przypadku powinien wykonać się kod klasy `OpiekunStudiow` (gdyby obiekt `o` rzeczywiście zabierał ze sobą swój kod, działoby się tak automatycznie; fakt, że tak nie jest, wynika po prostu z optymalizacji). Znaczy to, że ten sam fragment składni powinien spowodować wykonanie dwóch różnych fragmentów kodu w różnych sytuacjach. Wysyłany komunikat jest **dynamicznie wiązany** z odpowiednim kodem[6].

Na koniec rozważmy inny bardzo prosty przykład wiązania dynamicznego, który ilustruje także przypadek wysyłania przez obiekt komunikatu do samego siebie.

[6] W najprostszy sposób można to zrobić, sprawdzając typ obiektu podczas wykonywania się programu. W nowoczesnych językach jednak większość pracy bierze na siebie kompilator, co pozwala na większą wydajność.

Przypuśćmy, że interfejs klasy `Wykladowca` zawiera komunikat `mozliwosci`; w odpowiedzi na niego obiekt klasy `Wykladowca` powinien zwrócić zbiór czynności, które wykładowca może wykonać. Być może kod klasy `Wykladowca` jest zaimplementowany w podobny sposób jak w poniższym pseudokodzie (w którym instrukcja `sam.mozeWykonac(obowiazek)` oznacza wynik zwracany po wysłaniu do tego obiektu komunikatu z selektorem `mozeWykonac` i argumentem `obowiazek`):

```
coMogeWykonac := pustyZbior
dla kazdy obowiazek w listaWszystkichObowiazkow
jesli (sam.mozeWykonac(obowiazek)) wtedy dodaj obowiazek do coMogeWykonac
inaczej zrob nic
zwroc coMogeWykonac
```

Oczywiście tę operację można przesłonić w nowej klasie `OpiekunStudiow`, tylko po co tak robić (wydajność raczej nie będzie ważna)? Załóżmy, że kod pozostanie na swoim miejscu, dzięki czemu zostanie bez zmian odziedziczony przez obiekty klasy `OpiekunStudiow`. Przypomnijmy jednak, że w klasie `OpiekunStudiow` została przesłonięta operacja `mozeWykonac`. Co się stanie, gdy obiekt klasy `OpiekunStudiow` otrzyma komunikat `mozliwosci`? Zostanie wykonany powyższy kod — w czym nie ma nic szczególnego — co spowoduje wysłanie komunikatu `mozeWykonac` do siebie samego. Jest to obiekt klasy `OpiekunStudiow`, tak więc po otrzymaniu komunikatu `mozeWykonac` zostanie wykonana wyspecjalizowana wersja kodu zdefiniowana w klasie `OpiekunStudiow`.

W przypadku używania czystego języka obiektowo zorientowanego takie zachowanie szybko powinno stać się drugą naturą. Ostrożność należy zachować w przypadku języka C++: takie zachowanie wykazują tylko metody wirtualne. Niektóre osoby, łącznie ze mną[7], są głęboko przekonane, że zaczynając programowanie obiektowo zorientowane w języku C++ powinno się początkowo tworzyć wyłącznie metody wirtualne[8], a dopiero potem nauczyć się tworzenia metod niewirtualnych.

PODSUMOWANIE

Ten rozdział był krótkim wprowadzeniem do zorientowania obiektowego. Opisaliśmy, czym jest obiekt i jak między obiektami są przesyłane komunikaty. Pokazaliśmy, że obiekty w systemie można z pożytkiem podzielić na klasy, zamiast rozpatrywać je indywidualnie, oraz rozpoczęliśmy rozpatrywanie powiązań między obiektami i komponentami. Omówiliśmy, jak za pomocą dziedziczenia klasa może zostać zdefiniowana z wykorzystaniem definicji innej klasy. Na koniec przedstawiliśmy inną cechę dziedziczenia w językach zorientowanych obiektowo — możliwość udostępniania polimorfizmu i wiązania dynamicznego.

W następnym rozdziale dokonamy wprowadzenia do głównego tematu tej książki — Ujednoliconego Języka Modelowania, który może być używany do określania specyfikacji i projektowania systemów korzystających z przewagi, jaką zapewnia zorientowanie obiektowe.

[7] Stevens.

[8] Łącznie (zwłaszcza) z destruktorami, ale nigdy z konstruktorami. Dlaczego?