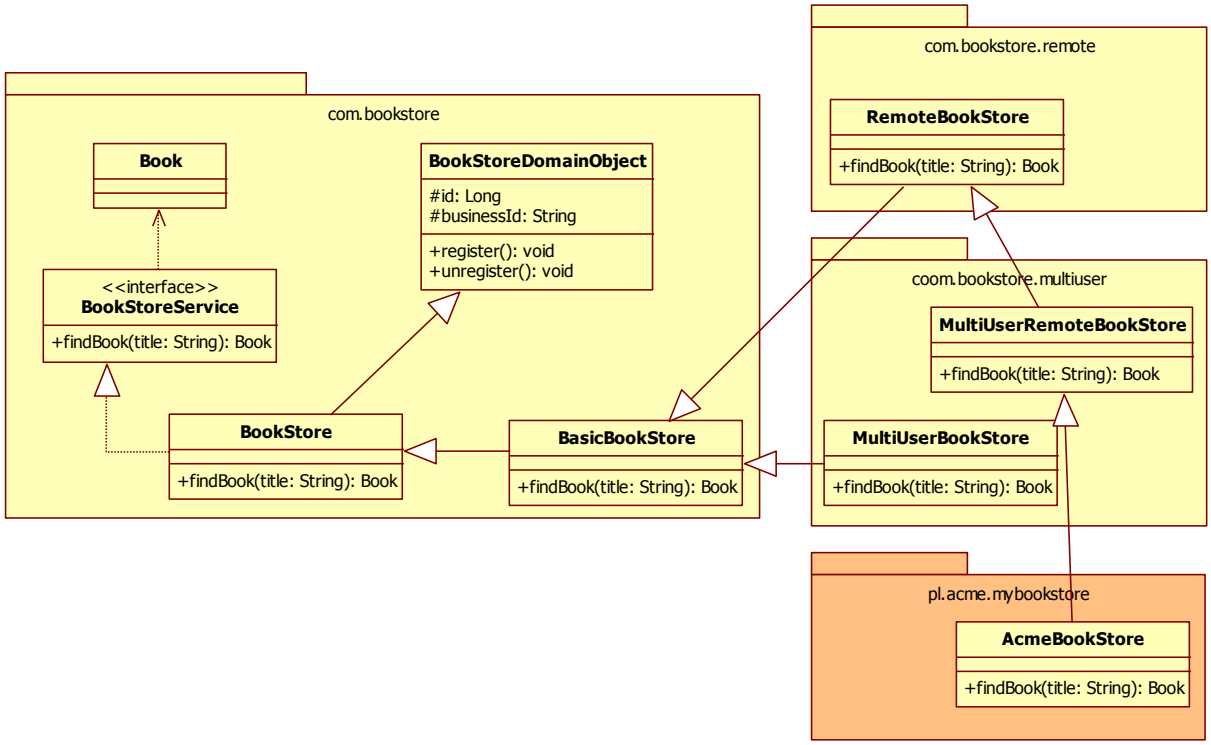
Ostatnio wygrzebałem w polskiej blogsferze namiar na Aristotle's Error or Agile Smile. Wynika z niego, że nasza wspaniała obiektowość jest dziełem przypadku (a co nie jest?), a ideologię o modelowaniu świata rzeczywistego dorobili spece od marketingu. Jak było w rzeczywistości nie dociekałem. Skoro już obiektowość, a wraz z nią sztandarowe dziedziczenie

DZIEDZICZENIE

Na Rysunek 1 jest fragment jakiegoś tam systemu. Pewnie każdy z nas widział podobne rzeczy choćby w Java API, które roi się od podobnych lub o wiele bardziej skomplikowanych konstrukcji.



RYSUNEK 1

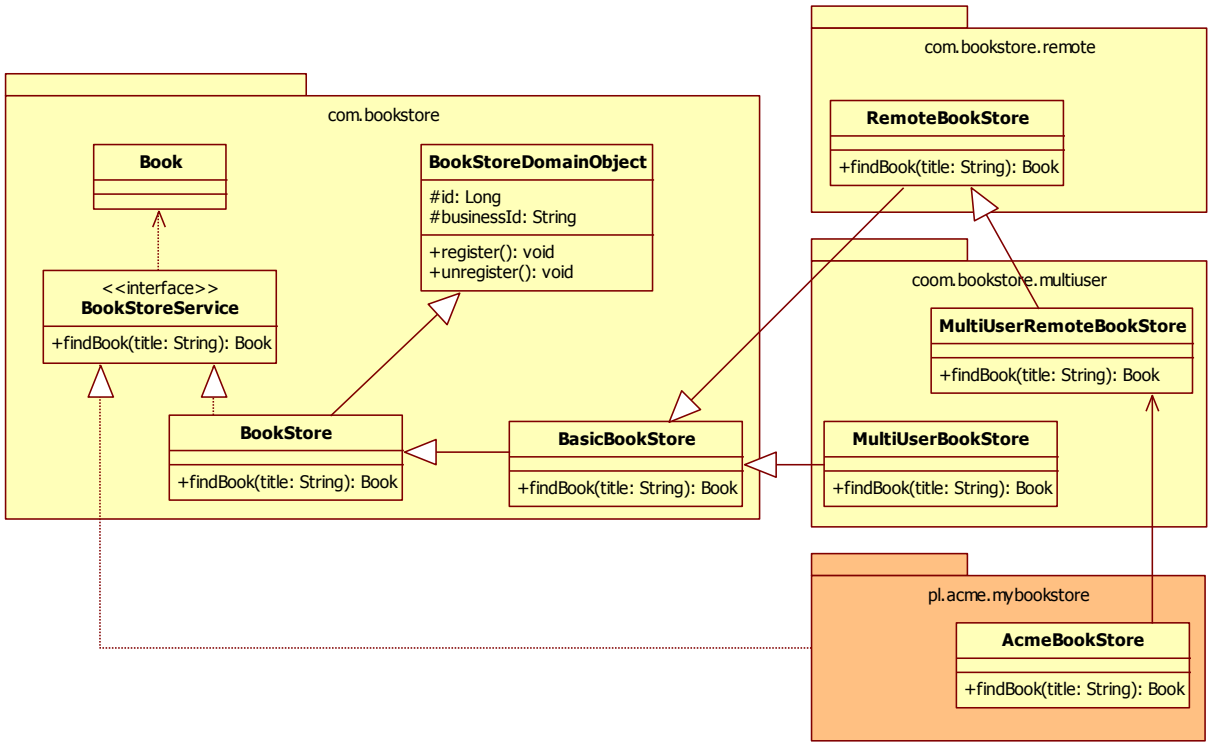
Choć programiście całkiem nieźle używa się klas zaprojektowanych w ten sposób, to gdy programista chciałby na bazie takich konstrukcji rozwijać swoją aplikację (czytaj: dalej nieograniczenie dziedziczyć), to pojawia się kilka problemów:

- Aby zrozumieć działanie metody `MultiUserRemoteBookStore.findBook()` zapoznać się z całą hierarchią dziedziczenia,
- Faktycznie uruchamiany kod metody tej metody jest rozsiany pomiędzy wiele klas w hierarchii,
- Wymusza się na nas użycie konstruktorów (parametrowych) z nadklas, których wcale nie mieliśmy zamiaru używać,

- Dodatkowo nie wiadomo co robią te kostruktory; a nóż przy tworzeniu mojego obiektu coś wybuchnie…
- Trudno przetestować jednostkowo nową klasę. No, bo jak tu zamokować kod wywoływany w testowanej metodzie poprzez `super.findBook()`?

KOMPOZYCJA

Większość problemów wynikających ze swobodnego dziedziczenia rozwiązuje *kompozycja*. Zamiast wyprowadzać nową klasę `AcmeBookStore` z `MultiUserBookStore` implementujemy główny interfejs `BookStoreService` a do potrzebnych metod odwołujemy się poprzez delegację.



RYSUNEK 2

I kawałek kodu:

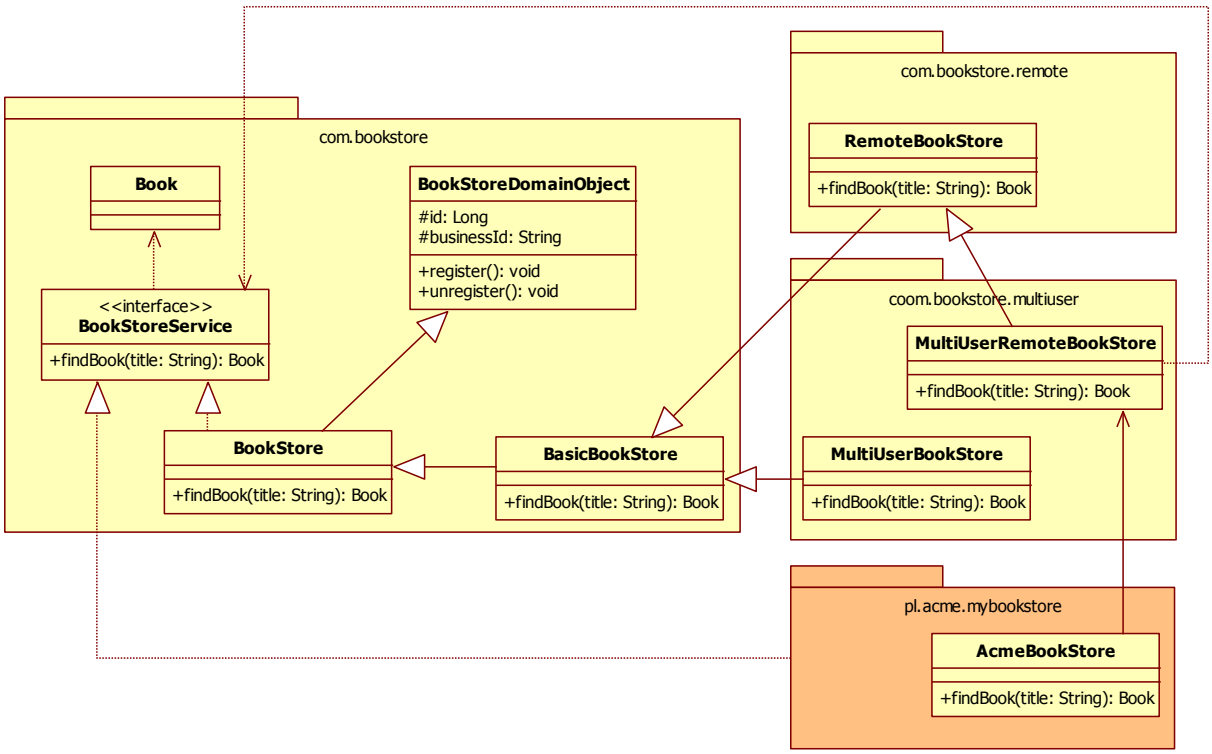
```
public class AcmeBookStore implements BookStoreService {

	private MultiUserBookStore multiUserBookStore;

	public Book findBook( String title ) {
		//...
		multiUserBookStore.findBook( title );
		//...
	}

}
```

(Na marginesie warto zauważyć, że w ten sposób stworzyliśmy implementację Dekoratora.)

To rozwiązanie jest zdecydowanie bardziej czytelne i *unit-testing-friendly.* Kłopot pojawia się gdy nie w architekturze, z którą pracujemy nie istnieje odpowiednik interfejsu `BookStoreService`. Wtedy już, chcąc nie chcąc, zazwyczaj godzimy się godzimy się na dziedziczenie.

PROGRAMOWANIE POPRZEZ INTERFEJSY

Mając na uwadze w/w mogę zastanawiać się: w jaki sposób mogę projektować architekturę mojego kodu tak, aby nie generował problemów z dziedziczeniem. Pierwszą rzeczą, która przychodzi mi na myśl jest *programowanie poprzez interfejsy*. Nie oznacza to bynajmniej, że każda nowa klasa `UserManager` ma swój interfejs UserManagerService, albo (w wersji hardcore) z każdą nową klasą pojawiają się trzy nowe byty w systemie (`UserManagerService`, `UserManagerImpl`, `AbstractUserManager`). Zerknijmy na rysunek



RYSUNEK 3

W tej architekturze centralnym punktem systemu (podsystemu, biblioteki) jest interfejs – kontrakt, który mają realizować poszczególne implementacje. Specyfika implementacji np. `RemoteBookStore` uzyskiwana jest nie poprzez odpowiednie klasy narzędziowe np. `RemotingUtilty`. Dzięki temu można tworzyć kolejne implementacje specjalizujące się w coraz to nowych rzeczach, bez konieczności dziedziczenia.

PODSUMOWUJĄC

Ujawniły nam się następujące rzeczy:

- Preferowanie kompozycji ponad dziedziczenie,
- Programowanie poprzez interfejsy.

Są to jedne z kluczowych paradygmatów programowania obiektowego. Mamy z nich następujące korzyści:

- Kod jest czytelny;
- Kod jest otwarty na testowanie.

Ostatecznie pozostaje jedno pytanie: czy dziedziczenie jest złe? Nie, jest bardzo dobre... w pewnych kontekstach… Złe jest jego nadużywanie. Jak więc rozpoznawać, które konteksty są odpowiedni dla dziedziczenia? Pewnie można wykombinować jakieś obiektywne kryteria, ale ja proponuję znać się na intuicję: czy dziedziczenie uprościło czy zagmatwała architekturę? Czy łatwiej testować, czy trudniej? Czy przyjemniej się pisze – przeciwnie – czy chce Ci się….

O AUTORZE

Michał Bartyzel

Konsultant i trener w firmie szkoleniowo-doradczej **BNS IT**. W pracy zawodowej zajmuje się doskonaleniem programistów i zespołów programistycznych, wdrażaniem metodyk pracy oraz rozwijaniem kompetencji pracowników branży IT. Jest współzałożycielem łódzkiego oddziału Java User Group.

http://www.linkedin.com/in/mbartyzel

O BNS IT

BNS IT jest firmą szkoleniowo-doradczą zajmującą się **rozwijaniem i doskonaleniem zespołów projektowych z branży IT**.

Nasze usługi skierowane są do firm zatrudniających programistów. Pracujemy dla trzech grup klientów:

- firm zajmujących się wytwarzaniem oprogramowania
- firm, które posiadają zespoły programistyczne, lecz działają na innych rynkach niż IT
- firm integrujących systemy informatyczne

http://www.bnsit.pl