

IDŹ DO

PRZYKŁADOWY ROZDZIAŁ



SPIS TREŚCI

KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK

DODAJ DO KOSZYKA

CENNIK I INFORMACJE

ZAMÓW INFORMACJE
O NOWOŚCIACH

ZAMÓW CENNIK

CZYTELNIA

FRAGMENTY KSIĄŻEK ONLINE

Sztuka pisania oprogramowania. Wybór i redakcja Joel Spolsky



Autor: Joel Spolsky

Tłumaczenie: Andrzej Grażyński

ISBN: 83-246-0464-2

Tytuł oryginału: [The Best Software Writing I: Selected and Introduced by Joel Spolsky](#)

Format: B5, stron: 320

Programowanie to nie tylko wiedza – to także sztuka. Aplikacja jest narzędziem, które powinno być przede wszystkim użyteczne i ergonomiczne. Niestety, wielu twórców oprogramowania zapomina o tym, pisząc swoje programy. Powodów jest wiele: zbyt mało czasu, źle sformułowane założenia, nieprawidłowa komunikacja między członkami zespołu projektowego czy też niestosowanie się do konwencji kodowania i testowania. Niezależnie od przyczyn, konsekwencją jest oprogramowanie, które nie spełnia swojej podstawowej funkcji, jaką jest użyteczność.

Po przeczytaniu książki „Sztuka pisania oprogramowania. Wybór i redakcja Joel Spolsky” spojrzysz na proces tworzenia aplikacji w inny sposób. Czytając zawarte w niej artykuły autorstwa doświadczonych programistów i menedżerów, dowiesz się, jak prowadzić projekty informatyczne, czego unikać i jakie techniki stosować. Znajdziesz w niej opracowania dotyczące stylu kodowania, zarządzania projektem, testowania, korzystania z nieudokumentowanych właściwości systemu oraz zatrudniania programistów. Nieważne, czy jesteś programistą, czy kierownikiem projektu, dowiesz się z niej wielu interesujących rzeczy.

W książce poruszono między innymi:

- Styl kodowania
- Projektowanie interfejsów użytkownika
- Pułapki outsourcingu
- Właściwe procedury testowania
- Unikanie nadmiernie rozbudowanych procesów decyzyjnych
- Zasady pracy z zespołem projektowym
- Dobór odpowiednich pracowników

**Jeśli chcesz podnieść jakość tworzonego oprogramowania,
koniecznie przeczytaj tę książkę**



SPIS TREŚCI

	O redaktorze	7
	O autorach	9
	Wprowadzenie	17
<i>Ken Arnold</i>	Styl jest istotny	21
<i>Ken Bambrick</i>	Nominacja do nagrody za najgłupszy interfejs użytkownika: narzędzie wyszukiwania Windows	29
<i>Michael Bean</i>	Pułapki programistycznego outsourcingu. Dlaczego niektóre firmy programistyczne mylą pudełko z czekoladkami?	31
<i>Rory Blyth</i>	Excel jako baza danych	39
<i>Adam Bosworth</i>	Prosto, zwyczajnie, po ludzku	45
<i>Danah Boyd</i>	Autystyczne oprogramowanie społeczne	57
<i>Raymond Chen</i>	Dlaczego nie blokować programów wykorzystujących nieudokumentowane mechanizmy?	69
<i>Kevin Cheng i Tom Chi</i>	Kopanie lamy	73

<i>Cory Doctorow</i>	Boże, zachowaj kanadyjski internet od WIPO ..	75
<i>ea_spouse</i>	EA: ludzka historia	81
<i>Bruce Eckel</i>	Rygorystyczna kontrola typów kontra rygorystyczne testowanie	89
<i>Paul Ford</i>	Processing	101
<i>Paul Graham</i>	Wielcy hakerzy	117
<i>John Gruber</i>	Gdy pole URL staje się wierszem poleceń ...	133
<i>Gregor Hohpe</i>	Dlaczego w Starbuck® nie korzysta się z potwierdzania dwufazowego?	141
<i>John Jeffries</i>	Pasja	147
<i>Eric Johnson</i>	C++ — zapomniany koń trojański	151
<i>Eric Lippert</i>	Ilu pracowników Microsoftu potrzeba do wymiany żarówki?	157
<i>Michael „Rands” Lopp</i>	Co robić, gdy zostaniesz wkręcony? 5 scenariuszy dla pracowitych dyrektorów technicznych	161
<i>Larry Osterman</i>	Reguła tworzenia oprogramowania nr 2: pomiar wydajności testerów za pomocą metryk ilościowych nie zdaje egzaminu	173
<i>Mary Poppendieck</i>	Kompensacja zespołowa	179
<i>Rick Schaut</i>	Mac Word 6.0	193
<i>Clay Shirky</i>	Grupa sama dla siebie jest największym wrogiem	203
<i>Clay Shirky</i>	Grupa jako użytkownik: <i>flamewars</i> a projektowanie oprogramowania społecznego	229
<i>Eric Sink</i>	Zamykanie luki, część 1.	241
<i>Eric Sink</i>	Zamykanie luki, część 2.	251
<i>Eric Sink</i>	Loteria zatrudniania	265
<i>Aaron Swartz</i>	PowerPoint — remiks	279
<i>why the lucky stiff</i>	Krótką, ilustrowaną i (mam nadzieję) bezsstresową wycieczką po języku Ruby	285
	Skorowidz	311

Rozdział 11

Bruce Eckel

RYGORYSTYCZNA KONTROLA TYPÓW KONTRA RYGORYSTYCZNE TESTOWANIE¹

Od redakcji: Pamiętam, że gdy pracowaliśmy w Microsoftzie nad VBA, toczyliśmy niekończące się dyskusje na temat statycznej i dynamicznej kontroli typów w programach.

Ze *stacyną* kontrolą typów mamy do czynienia wówczas, gdy weryfikacja poprawności typów wszystkich zmiennych przeprowadzana jest na etapie kompilacji programu. Jeżeli na przykład w programie znajduje się funkcja `log()`, która wymaga liczby rzeczywistej jako jedyne parametru, wywołanie tej funkcji w postaci `log("foo")` spowoduje sygnalizację błędu w rodzaju „tu oczekuje się liczby rzeczywistej” lub podobnego. Konsekwencją użycia

¹ Bruce Eckel, „Strong Typing vs. Strong Testing”, *Thinking About Computing*, artykuły Bruce Eckela na *MindView.net* (<http://www.MindView.net>), 2 maja 2003. Patrz <http://mindview.net/WebLog/log-0025>.

niewłaściwego typu — łańcucha zamiast liczby — jest niemożność skompilowania programu.

Dla odróżnienia, w ramach *dynamicznej* kontroli typów weryfikacja tychże odbywa się w czasie wykonania programu. Konstrukcja `log("foo")` zostanie skomplikowana, a jej niepoprawność okaże się faktem dopiero w trakcie wykonania programu. Podejście to ma tę oczywistą wadę, iż fakt ten może stać się wiadomy dopiero po kilku miesiącach czy latach eksploatacji programu, zwłaszcza jeżeli wzmiankowane wywołanie znajduje się wewnątrz funkcji wywoływanej bardzo rzadko.

Jako że projektowany VBA miał być z założenia językiem skryptowym dla Excela, osobiście optowałem za kontrolą dynamiczną. Jest ona koncepcyjnie prostsza dla użytkowników niebędących programistami, którzy mogą mieć trudności ze zrozumieniem pojęcia zmiennej, nie mówiąc już o pojęciu typu.

Miałem wówczas po swej stronie wielu użytkowników Smalltalka argumentujących (raczej ogólnikowo): „Wciąż szukasz przyczyny problemu, więc lepiej byłoby, żebyś poznał ją w ciągu kilku sekund”. Często okazuje się to prawdą, jednak nie zawsze.

Ostatecznie, po długich debatach, udało mi się przekonać innych do moich racji i tak narodził się typ `Variant` — struktura zdolna przechowywać wartości dowolnego typu — jako element VBA i COM oraz jako jedyny dopuszczalny typ późniejszego języka skryptowego VBS.

Nie zapominam oczywiście, że rygorystyczna kontrola typów dokonywana w czasie kompilacji jest wspaniałą rzeczą, umożliwia bowiem wczesne wykrycie wielu błędów, co dla mnie, jako użytkownika C++, jest sprawą bezdyskusyjną. Jeżeli na przykład tworzysz oprogramowanie dla przedsiębiorstwa, w którym jedynie menedżerowie uprawnieni są do otrzymywania nagród, możesz zdefiniować dwie różne struktury danych — dla pracowników i menedżerów — i tylko drugą z tych struktur wyposażyć w metodę `PayBonus()`. Wywołanie tej metody na rzecz rekordu reprezentującego szarego pracownika, nie szacownego menedżera, będzie niemożliwe, bo nie pozwoli na to kompilator.

Problem w tym, iż tworzenie typów danych — jako samoistnych bytów — tylko po to, by częściowe testowanie programu przeprowadzone zostało już na etapie kompilacji, jest pomysłem po trosze niefortunnym. Owo „częściowe testowanie” może bowiem obejmować jedynie testy o charakterze ogólnym, w rodzaju „czy mogę zrobić z tym obiektem to a to”, nie zaś testy bardziej szczegółowe w rodzaju „czy ta funkcja zwróci wartość 2,12 jeśli parametry jej wywołania będą miały postać 1, 32 i 'aardvark'”.

W efekcie rygorystyczna kontrola typów — jako jeden z mechanizmów weryfikujących pewien tylko aspekt poprawności programu — może okazywać się

dla programisty w pewnym stopniu kłopotliwa. Staje się oczywiste, że w celu dogłębnej weryfikacji tej poprawności powinniśmy posłużyć się narzędziem bardziej funkcjonalnym i bezpośrednim — testami modułów. Prezentowany przez autora pogląd, iż powinny one stać się zastępnikiem rygorystycznej kontroli typów na etapie kompilacji, jest ideą samą w sobie intrygującą.

Zanim jednak oddam głos Bruce'owi, muszę zwrócić uwagę na jedną istotną rzecz. Otóż dynamiczna kontrola typów — czyli weryfikacja ich poprawności w czasie wykonywania programu — wiąże się ze spadkiem efektywności wykonywania programu. Wielkość tego spadku jest oczywiście różna dla różnych programów, generalnie jednak języki stosujące kontrolę dynamiczną okazują się wolniejsze od swych odpowiedników opartych na statycznej kontroli. Wykorzystuję na co dzień program do odfiltrowywania spamu, napisany w języku Python i niekiedy „oflagowanie” pojedynczej wiadomości wymaga kilku sekund, tak że zakwalifikowanie jako spam 10 czy 20 wiadomości może już oznaczać oczekiwanie przez kilka minut. Taka jest cena dynamicznej kontroli typów, w przypadku dużej „farmy” serwerowej przybierająca konkretne rozmiary perspektywy pięcio- lub dziesięciokrotnego zwiększenia liczby serwerów w celu utrzymania wydajności systemu na akceptowalnym poziomie.

Trudno więc jednoznacznie wskazać wyraźną przewagę któregoś z opisanych sposobów — statycznego czy dynamicznego — kontroli typów danych w programie. Nawet bowiem jeśli testy modułów dają możliwość wszechstronnego weryfikowania poprawności programów, to jednak nie można wpadać w przesadę, optując na rzecz całkowitego zaniechania kontroli typów przez kompilatory.



Od pewnego czasu szczególnym przedmiotem mego zainteresowania jest produktywność programistów. Czas programisty jest drogi, czas komputera — niemal darmowy, nie ma więc żadnego powodu, by płacić tym pierwszym za ten drugi.

Jak możemy maksymalnie ułatwić sobie rozwiązanie konkretnego problemu? Gdy tylko pojawia się nowe narzędzie (głównie — nowy język programowania), dostarcza ono pewnego poziomu abstrakcji, który może, lecz nie musi, skrywać przed programistą pewne nieistotne szczegóły. Osiągnięcie tej abstrakcji wymaga jednak niekiedy tyle wysiłku, iż programista często gotów byłby sprzymierzyć się (niczym doktor Faust) z samym diabłem, by tylko mieć ten wysiłek już za sobą. Koronnym przykładem takiego języka jest Perl: jego

bezpośredniość uwalnia co prawda programistę od dużej dozy „programistycznej biurokracji”, lecz nieczytelna składnia (wzorowana na uniksowych narzędziach w stylu `awk`, `sed` i `grep`) z punktu widzenia produktywności programisty okazuje się fatalna.

W ciągu kilku ostatnich lat wspomniany „faustowski targ” zdaje się przybierać postać bardziej namacalną — orientację tradycyjnych języków w kierunku statycznej kontroli typów. Zdecydowało to o mojej dwumiesięcznej przygodzie miłosnej z Perlem, który oznaczał całkowity zwrot w kategoriach mojej własnej produktywności jako programisty (płomienna miłość skończyła się równie szybko za sprawą karygodnego traktowania przez Perl referencji i klas; problemy ze składnią dały znać o sobie nieco później). Kwestię wyboru statycznej albo dynamicznej kontroli typów trudno w kontekście Perla rozważać, jako że nie sposób budować w nim projektów dostatecznie dużych na to, by wynikające z tej kwestii problemy mogły naprawdę dawać znać o sobie.

Po przesiadce na język Python (do pobrania za darmo z witryny *www.Python.org*) — język, w którym można tworzyć ogromne, skomplikowane systemy — coraz wyraźniej począłem konstatować, że mimo ewidentnej beztroski w kategoriach kontroli typów tworzone w tym języku programy funkcjonują całkiem nieźle, a ich tworzenie wymaga niewielkiego wysiłku i wolne jest od tych wszystkich problemów, jakich moglibyśmy spodziewać się po języku pozbawionym statycznej kontroli typów. Kontroli, którą wielu zwykło uważać za jedyne właściwy sposób rozwiązywania problemów programistycznych.

Skoro jednak statyczna kontrola typów jest mechanizmem tak istotnym i tak nieodzownym, dlaczego ludzie w ogóle decydują się na używanie języka Python do tworzenia dużych, skomplikowanych systemów (w dodatku szybciej i przy znacznie mniejszym wysiłku niż w przypadku języków „statycznych”), wolnych od katastrofalnych zachowań, jakie (rzekomo) niechybnie powinny wystąpić?

Powyzsza kwestia zachwiała mym niewzruszonym dotąd przekonaniem do statycznej kontroli typów (nabytym w czasie przesiadki z wczesnych wersji języka C na C++, gdzie usprawnienie wielu mechanizmów było dramatyczne) i to do tego stopnia, że kwestionować począłem nawet istnienie weryfikowanych wyjątków (*checked exceptions*) w języku Java² — co wywołało dyskusję tak burzliwą³, jakby moja obrona wyjątków nieweryfikowanych spowodowała miała co najmniej zagładę cywilizacji ludzkiej. W książce *Thinking in Java*

² Wyjątki weryfikowane są cechą języka polegającą na tym, że kompilator (w czasie kompilacji) dokonuje sprawdzenia, czy każda funkcja, mogąca potencjalnie generować konkretne wyjątki, posiada kod niezbędny do obsłużenia tychże lub przynajmniej zabezpieczający przed wydostaniem się ich na zewnątrz — *przyp. red.*

³ Patrz <http://www.mindview.net/Etc/Discussions/CheckedExceptions>.

(Prentice Hall PTR, wyd. trzecie, 2000)⁴ posunąłem się jeszcze dalej i zdemontowałem użycie wyjątku `RuntimeException` jako klasy-otoczki „eliminującej” wyjątki weryfikowane. Każdorazowo, gdy używam tego mechanizmu, wydaje się on działać prawidłowo (gwoźli sprawiedliwości winien jestem Czytelnikom informację, iż Martin Fowler wpadł na ten sam pomysł mniej więcej w tym samym czasie co ja), mimo to wciąż otrzymuję listy upominające mnie, że szargam w ten sposób prawdę i uświęcone zasady i powinienem być ścigany na mocy USA Patriot⁵ (hej, chłopcy z FBI — witam w moim blogu).

Stwierdzenie, że weryfikowane wyjątki niewarte są kłopotów, jakie mogą powodować (chodzi oczywiście o samo weryfikowanie, nie o wyjątki w ogólności — jestem przekonany, że spójny, jednolity mechanizm raportowania błędów jest dla języka niezbędny), nie daje odpowiedzi na pytanie: „Dlaczego Python spisuje się tak dobrze, wszak zgodnie z naszymi konwencjonalnymi przekonaniami tworzone w nim programy powinny objawiać istne lawiny błędów?”. Python i języki jemu podobne podchodzą do kontroli typów w sposób zgoła leniwy: zamiast narzucać możliwie najwcześniej, możliwie najbardziej rygorystyczne ograniczenia na typy obiektów (jak czyni to Java), języki takie jak Ruby, Smalltalk i oczywiście Python maksymalnie tę kontrolę *rozluźniają*, przywiązując wagę do typu obiektów dopiero wtedy, kiedy naprawdę jest to konieczne.

Koncepcja ta, zwana typowaniem spowolnionym (*latent typing*) lub typowaniem strukturalnym (*structural typing*), określana bywa także potocznym mianem „kaczego typowania” (*duck typing*) — jest niemrawa niczym kaczy chód i jak kwakanie niewyraźna. Typowanie to oznacza w praktyce tyle, że można przesłać do obiektu dowolny komunikat, bez względu na jego typ. Gdybyśmy na przykład chcieli zaprogramować w Javie dialog dwóch zwierzątek, mógłby on wyglądać mniej więcej tak:

```
// zwierzęca konwersacja – wersja w Javie
interface Pet {
    void speak();
}

class Cat implements Pet {
    public void speak() { System.out.println("miau!"); }
}
```

⁴ Dostępne jest wydanie polskie, szczegóły pod adresem <http://helion.pl/ksiazki/thija3.htm> — *przyp. tłum.*

⁵ *USA Patriot Act* — prawo amerykańskie ustanowione 26 października 2001 w wyniku zamachu na WTC. Na mocy tego prawa wolno przetrzymywać bez sądu, przez czas nieokreślony, obywateli nieamerykańskich, uznanych za zagrożenie dla bezpieczeństwa narodowego. Patrz http://pl.wikipedia.org/wiki/USA_Patriot_Act. — *przyp. tłum.*

```

class Dog implements Pet {
    public void speak() { System.out.println("hau!"); }
}

public class PetSpeak {
    static void command(Pet p) { p.speak(); }
    public static void main(String[] args) {
        Pet[] pets = { new Cat(), new Dog() };
        for(int i = 0; i < pets.length; i++)
            command(pets[i]);
    }
}

```

Zwróćmy uwagę na ważny fakt, że w charakterze wywołania funkcji `command()` dopuszczalny jest jedynie obiekt określonego typu — `Pet` — reprezentującego dowolne (abstrakcyjne) zwierzę. Aby zatem zaprogramować zachowanie się konkretnych zwierząt (psa i kota), musimy z klasy `Pet` wyprowadzić reprezentujące je klasy (`Dog` i `Cat`) i w każdej z nich zdefiniować metodę `speak()`, wywołującą się oryginalnie z typu `Pet`.

Przez dłuższy czas uważałem, że przedstawione dziedziczenie metod jest przyrodzonym elementem programowania obiektowego (a odmienne zdanie użytkowników `Smalltalka` w tej kwestii bardzo mnie irytowało). Kiedy jednak zacząłem używać Pythona, zauważyłem bardzo ciekawą rzecz. Przetłumaczmy mianowicie prezentowany kod na język Python:

```

# zwierzęca konwersacja – wersja w Pythonie
class Pet:
    def speak(self): pass

class Cat(Pet):
    def speak(self):
        print "miau!"

class Dog(Pet):
    def speak(self):
        print "hau!"

def command(pet):
    pet.speak()

pets = [ Cat(), Dog() ]

for pet in pets:
    command(pet)

```

Jeśli nie widziałeś wcześniej Pythona, z pewnością stwierdzisz, iż definiuje on na nowo pojęcie języka związłego, w bardzo pozytywnym znaczeniu. Czy uważasz C i C++ za języki związłe? Wyrzucmy z nich nawiasy klamrowe — czytelne dla człowieka akapitowanie jest w Pythonie jednocześnie środkiem

do zaznaczania granic bloków. Typy argumentów i wyników funkcji? Zostawmy je samemu językowi. W instrukcjach tworzących klasy nazwy klas bazowych ujęte są w nawiasy. `def` oznacza definicję funkcji lub metody. Nie ma domyślnego parametru `this` reprezentującego obiekt wywołujący metodę, parametr ten trzeba specyfikować w sposób jawny i zgodnie z przyjętą konwencją nadaje mu się nazwę `self`.

Słowo kluczowe `pass` oznacza odłożenie definicji na później i jako takie może być uważane za analogię słowa kluczowego `abstract`.

Zwróćmy uwagę, że w wywołaniu `command(pet)` parametrem wywołania jest *jakiś* obiekt o nazwie `pet`, lecz nie sposób wywnioskować (na podstawie definicji funkcji `command`) niczego na temat typu tego obiektu. Jedyną rzeczą, jakiej się od tego obiektu wymaga, jest możliwość wywołania na jego rzecz metody `speak()`. Tak właśnie wygląda spowolnione typowanie, którego wybranymi aspektami zajmiemy się za chwilę.

Kolejną rzeczą wartą wzmianki jest fakt, że funkcja `command()` jest *zwykłą funkcją*, nie metodą. W języku Python jest to rzecz naturalna, nie wszystko bowiem musi się w nim odbywać na modłę obiektową.

Listy i słowniki (nazywane także mapami i tablicami skojarzeniowymi), tak ważne przy tworzeniu wielu programów, w języku Python uczynione zostały jego nierozzerwalną częścią, ich używanie nie wymaga więc importowania jakichś specjalnych, dodatkowych bibliotek. Poniższa lista

```
pets = [ Cat(), Dog() ]
```

składa się z dwóch nowo utworzonych obiektów typu `Cat` i `Dog`. W celu ich utworzenia wywoływane są oczywiście odpowiednie konstruktory, lecz nie trzeba zaznaczać tego faktu *explicite* za pomocą słowa kluczowego `new` (w języku Java też nie jest ono tak naprawdę potrzebne, pozostało w nim jednak jako jeden z elementów schedy po C++).

Rodzimym elementem Pythona jest także inna równie istotna operacja – iterowanie po sekwencji elementów: instrukcja

```
for pet i pets:
```

podstawia kolejne elementy listy `pets` pod zmienną `pet`. Rozwiązanie bardziej eleganckie niż w Javie, nawet w porównaniu z instrukcją `foreach` w wersji J2SE5.

Wynik działania powyższego fragmentu jest identyczny z przedstawioną wcześniej wersją w Javie. Łatwo teraz zrozumieć, dlaczego Python jest często nazywany „wykonywalnym pseudokodem”. Nie tylko jest on wystarczająco prosty, by można go było używać jak pseudokodu, lecz ma tę wspaniałą cechę, iż tworzone fragmenty kodu można natychmiast wykonywać. W praktyce oznacza to, że używając języka Python można błyskawicznie wypróbować nowe

pomysły i koncepcje, a gdy się już dostatecznie wykrystalizują, można kodować je w Javie, C, C++ lub innym języku „z wyboru”. No dobrze, ale skoro właśnie rozwiązaliśmy konkretny problem w języku Python, po cóż jeszcze kłopotać się „przepisywaniem” rozwiązania na inny język? Na prowadzonych przeze mnie seminariach wielokrotnie używałem Pythona do kodowania przykładowych ćwiczeń, ponieważ pozwoliło mi to na pokazanie w sposób wyraźny drogi, jaką sam dochodziłem do rozwiązania, a poza tym poprawność tworzonego sukcesywnie pseudokodu można było weryfikować na bieżąco, przez jego wykonywanie.

Najbardziej jednak istotnym spostrzeżeniem w stosunku do prezentowanego przed chwilą kodu jest to, że skoro funkcja `command()` nie troszczy się o typ swego parametru, można w ogóle *zrezygnować z budowania hierarchii klas* i z klasy bazowej `Pet` w szczególności:

```
# zwierzęca konwersacja – wersja w Pythonie, bez klasy bazowej
class Cat:
    def speak(self):
        print "miau!"

class Dog:
    def speak(self):
        print "hau!"

class Bob:
    def bow(self):
        print "dziękuję serdecznie!"
    def speak(self):
        print "witam w moich skromnych progach!"
    def drive(self):
        print "piiiiiiiiiiiiip!"

    def command(pet):
        pet.speak()

pets = [ Cat(), Dog(), Bob() ]

for pet in pets:
    command(pet)
```

Ponieważ jedyną cechą obiektu `pet`, istotną z punktu widzenia wywołania `command(pet)`, jest możliwość wysłania do tego obiektu komunikatu `speak()`, mogliśmy usunąć z programu klasę bazową `Pet`, a nawet dodać do niego klasę `Bob`, której z klasy `Pet` wyprowadzić niepodobna, ale która posiada metodę `speak()` i której obiekty z tego względu *mogą* być używane w charakterze parametrów wywołania funkcji `command()`.

Na widok powyższego zwoleńcy statycznej kontroli typów zapewne dostaną apopleksji i z jeszcze większą stanowczością twierdzić będą, że takie

rozluźnienie reguł niechybnie doprowadzić musi do katastrofy. Korzyści wynikające z bardziej klarownego wyrażania koncepcji nie są warte ponoszonego ryzyka, nawet jeśli oznaczać mogą w praktyce pięcio- lub dziesięciokrotne zwiększenie produktywności programistów w porównaniu z Javą czy C++.

Czy rzeczywiście? Czy przekazanie obiektu tam, gdzie nie powinien się on znaleźć, może istotnie powodować aż tak poważne problemy? Otóż w Pythonie wszelkie błędy raportowane są w postaci wyjątków, podobnie jak powinny być raportowane w Javie, C# i C++. Błędne użycie obiektu *zostanie* wykryte, tyle tylko, że stanie się to dopiero podczas wykonywania kodu zawierającego ów błąd. To kolejna woda na młyn zwolenników typowania statycznego, którzy twierdzą, że wobec tego nie można zagwarantować poprawności programu *przed* jego wykonaniem, bo kompilator nie wykonuje niezbędnej kontroli typów.

Gdy pisałem książkę *Thinking in C++* (Prentice Hall PTR, wyd. pierwsze, 1998)⁶ zastosowałem bardzo prostą metodę weryfikacji poprawności zamieszczonych przykładów: napisałem program, który automatycznie wyluskuje przykładowy kod z treści książki (na podstawie odpowiednich markerów-komentarzy na początku i na końcu każdego fragmentu), po czym tworzy odpowiedni skrypt dla programu MAKE, umożliwiający skompilowanie wszystkich przykładów. W ten sposób mogłem zagwarantować, że cały przykładowy kod zamieszczony w książce jest akceptowany przez kompilator i że jako taki jest (tak wtedy myślałem) w pełni poprawny. Dręczące mnie wątpliwości, że przecież bezbłędne skompilowanie kodu nie oznacza jeszcze jego poprawnego działania, dziwnym trafem ignorowałem, dumny z faktu, że udało mi się zautomatyzować wielki krok na początku drogi do weryfikacji kodu (Czytelnicy książki o programowaniu nadto dobrze zdają sobie sprawę z faktu, że wielu autorów po prostu nie przejmuje się błędami w prezentowanym przez siebie kodzie). Gdy jednak w ciągu następnych lat okazywało się, że niektóre z przykładów działają niezgodnie z oczekiwaniami, nie mogłem dłużej ignorować konieczności testowania skompilowanych programów. Dałem temu zresztą wyraz w trzecim wydaniu *Thinking in Java*, pisząc

Jeśli coś nie jest przetestowane, jest niepoprawne

Jeśli więc program napisany w języku o statycznej kontroli typów kompiluje się bezbłędnie, oznacza to tylko tyle, że jest składniowo poprawny (kompilator Pythona także sprawdza składnię, tyle że jej reguły są znacznie mniej rygorystyczne). Składniowa poprawność programu to tylko jeden z aspektów

⁶ Dostępne jest wydanie polskie, szczegóły pod adresem <http://helion.pl/ksiazki/thicpp.htm> — *przyp. tłum.*

poprawności rozumianej całościowo — jeśli kod *zdaje się wyglądać* na funkcjonujący prawidłowo, wcale nie oznacza to, że w istocie będzie prawidłowo funkcjonował.

Jedyną gwarancją poprawności — i to niezależnie od tego, czy kontrola typów ma charakter statyczny, czy dynamiczny — może dać tylko pomyślnie zaliczenie wszystkich testów *definiujących kryteria poprawności programu*. Mowa tu o testach modułów, testach akceptacyjnych itd. W książce *Thinking in Java*, wyd. trzecie, znaleźć można mnóstwo takich testów, których trud tworzenia solennie się opłacił. Gdy tylko programista wyrobi w sobie odruch testowania każdego napisanego kodu (zostanie „zarażony testowaniem”, jak się popularnie mówi), nigdy już nie będzie w stanie zmienić swego podejścia do programowania.

Przypominać to może przesiadkę z wczesnych wersji języka C na C++ — kompilator tego ostatniego wykonywał zdecydowanie więcej testów i tworzył kod bardziej efektywny. Na tym jednak rola każdego kompilatora musi się skończyć, bo nie ma on żadnej informacji odnośnie *spodziewanego zachowania się programu*; dalszą weryfikację poprawności mogą zapewnić jedynie wyczerpujące testy niezależnie od tego, w jakim języku tworzony jest program. Istnienie zestawu takich testów umożliwia weryfikowanie na bieżąco poprawności wszelkich zmian w kodzie (w ramach refaktoryzacji lub zmiany projektu), podobnie jak wykonanie kompilacji umożliwia weryfikację ich poprawności składniowej.

Tak więc niezależnie od wariantu typowania — statycznego albo dynamicznego — w obliczu braku wspomnianych testów poprawność składniowa programu może stwarzać co najwyżej iluzję jego poprawnego funkcjonowania (o czym wielu z nas zdążyło się już przekonać osobiście). Tym wobec tego, co z punktu widzenia poprawności programów rzeczywiście niezbędne, jest

Rygorystyczne testowanie, nie rygorystyczna kontrola typów

To właśnie przesądza o tym, że w Pythonie można tworzyć poprawne systemy. W C++ większość testów przeprowadzana jest w czasie kompilacji (z nielicznymi wyjątkami); w języku Java niektóre testy przeprowadzane są w czasie kompilacji, inne zaś (jak kontrola poprawności indeksów tablic) w czasie wykonania programu. W Pythonie większość testów przeprowadzana jest w czasie wykonania programu, a więc w istocie są one wykonywane, nieważne na jakim etapie. W dodatku uruchomienie programu w języku Python odbywa się znacznie szybciej niż uruchomienie podobnego programu w C++, Javie czy C#, a więc samo przeprowadzanie *rzeczywistych* testów — testów modułów,

testów weryfikujących hipotezy, testów sprawdzających alternatywne podejścia itp. — odbywa się efektywniej. Napisany w Pythonie program, który zaliczył wszystkie wspomniane testy, może być uważany za tak samo solidny jak jego odpowiednik w C++, Javie czy C#, w dodatku w środowisku Pythona można testy te tworzyć i uruchamiać znacznie szybciej.

Robert Martin jest długoletnim członkiem społeczności C++, autorem książek i artykułów, konsultantem i nauczycielem. No i oczywiście zatwardziałym zwolennikiem statycznego typowania. Tak myślałem o nim do czasu, gdy po lekturze jego blogu⁷ uświadomiłem sobie, iż to tylko część prawdy: *w istocie* należy on do programistów „zarażonych testowaniem” i doskonale zdaje sobie sprawę z fragmentarycznego charakteru kontroli wykonywanej przez kompilator. A także z tego, że języki z typowaniem dynamicznym pozwalają na tworzenie programów równie solidnych, jak ich statyczne odpowiedniki, jednakże w sposób znacznie bardziej produktywny.

Oczywiście Martin także był adresatem zwyczajowych komentarzy w rodzaju: „Jak możesz myśleć w ten sposób?” — komentarzy, które mnie samego skłoniły do zastanawiania się nad wyższością jednego rodzaju typowania nad drugim. Obydwoje zaczęliśmy jako zagorzali zwolennicy typowania statycznego i interesujące jest to, iż potrzeba doświadczeń na miarę trzęsienia ziemi, by człowiek skłonny był przewartościować swą filozofię.

⁷ <http://www.artima.com/weblogs/viewpost.jsp?thread=4639> — przyp. red.