

O'REILLY®



Python

w uczeniu maszynowym

PODEJŚCIE STEROWANE TESTAMI

Matthew Kirk

Python w uczeniu maszynowym

Podejście sterowane testami

Matthew Kirk

przekład: Jakub Niedźwiedź

Python w uczeniu maszynowym

© 2018 APN PROMISE SA

Authorized translation of English edition of
Thoughtful Machine Learning with Python
ISBN 978-1-491-92413-6

Copyright © 2017 Matthew Kirk. All rights reserved.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls of all rights to publish and sell the same.

APN PROMISE SA, ul. Domaniewska 44a, 02-672 Warszawa
tel. +48 22 35 51 600, fax +48 22 35 51 699
e-mail: mspress@promise.pl

Wszystkie prawa zastrzeżone. Żadna część niniejszej książki nie może być powielana ani rozpowszechniana w jakiegokolwiek formie i w jakikolwiek sposób (elektroniczny, mechaniczny), włącznie z fotokopiowaniem, nagrywaniem na taśmy lub przy użyciu innych systemów bez pisemnej zgody wydawcy.

Logo O'Reilly jest zarejestrowanym znakiem towarowym O'Reilly Media, Inc. Ilustracja z okładki i powiązane elementy są znakami towarowymi O'Reilly Media, Inc.

Wszystkie inne nazwy handlowe i towarowe występujące w niniejszej publikacji mogą być znakami towarowymi zastrzeżonymi lub nazwami zastrzeżonymi odpowiednich firm odnośnych właścicieli.

Przykłady firm, produktów, osób i wydarzeń opisane w niniejszej książce są fikcyjne i nie odnoszą się do żadnych konkretnych firm, produktów, osób i wydarzeń. Ewentualne podobieństwo do jakiegokolwiek rzeczywistej firmy, organizacji, produktu, nazwy domeny, adresu poczty elektronicznej, logo, osoby, miejsca lub zdarzenia jest przypadkowe i niezamierzone.

APN PROMISE SA dołożyła wszelkich starań, aby zapewnić najwyższą jakość tej publikacji. Jednakże nikomu nie udziela się rękojmi ani gwarancji.

APN PROMISE SA nie jest w żadnym wypadku odpowiedzialna za jakiegokolwiek szkody będące następstwem korzystania z informacji zawartych w niniejszej publikacji, nawet jeśli APN PROMISE została powiadomiona o możliwości wystąpienia szkód.

ISBN: 978-83-7541-357-1

Projekt okładki: Randy Comer
Ilustracje: Rebecca Demarest

Przekład: Jakub Niedźwiedź
Redakcja: Marek Włodarz
Korekta: Ewa Swędrowska
Skład i łamanie: MAWart Marek Włodarz

Spis treści

Wstęp	ix
Podziękowania.....	xi
1. W przybliżeniu prawdopodobnie poprawne oprogramowanie	1
Prawidłowe pisanie oprogramowania.....	2
SOLID	2
Testowanie albo TDD.....	4
Refaktoring	6
Pisanie prawidłowego oprogramowania	7
Pisanie odpowiedniego oprogramowania przy zastosowaniu uczenia	
maszynowego	7
Czym dokładnie jest uczenie maszynowe?	8
Wysoko oprocentowany dług uczenia maszynowego	8
Zastosowanie zasad SOLID w uczeniu maszynowym.....	9
Kod uczenia maszynowego jest skomplikowany	13
TDD: metoda naukowa 2.0	13
Refaktoring wiedzy.....	13
Plan tej książki	14
2. Szybkie wprowadzenie do uczenia maszynowego	15
Czym jest uczenie maszynowe?.....	15
Uczenie nadzorowane.....	16
Uczenie nienadzorowane	17
Uczenie wzmacniane.....	17
Co może osiągnąć uczenie maszynowe?	17
Notacja matematyczna używana w tej książce.....	19
Podsumowanie.....	20
3. K najbliższych sąsiadów	21
Jak ustalić, czy chcemy kupić dom?	21
Ile wart jest dany dom?	22
Regresja hedonistyczna.....	22
Czym jest sąsiedztwo?.....	23
K najbliższych sąsiadów	24

Najbliższe sąsiedztwo	24
Odległości	25
Nierówność trójkąta	25
Odległość geometryczna	26
Odległości obliczeniowe	27
Odległości statystyczne	30
Przekleństwo wymiarowości	31
Jak wybrać K?	32
Zgadywanie K	33
Heurystyka wyboru K	33
Wycenianie domów w Seattle	36
Informacje o danych	36
Ogólna strategia	37
Projekt kodowania i testowania	37
Konstrukcja regresora dla algorytmu K najbliższych sąsiadów	38
Testowanie algorytmu K najbliższych sąsiadów	40
Podsumowanie	43
4. Naiwna klasyfikacja bayesowska	45
Wykorzystanie twierdzenia Bayesa do znajdowania oszukańczych zamówień ..	45
Prawdopodobieństwa warunkowe	46
Symbole prawdopodobieństwa	46
Odwrócone prawdopodobieństwo warunkowe (czyli twierdzenie Bayesa)	48
Naiwny klasyfikator bayesowski	49
Reguła łańcuchowa	49
Naiwność w rozumowaniu bayesowskim	49
Pseudozliczanie	51
Filtr spamu	52
Uwagi przygotowawcze	52
Projekt kodowania i testowania	52
Źródło danych	53
EmailObject	53
Analiza leksykalna i kontekst	58
SpamTrainer	60
Minimalizacja błędów przez sprawdzanie krzyżowe	67
Podsumowanie	70
5. Drzewa decyzyjne i losowe lasy decyzyjne	71
Niuanse dotyczące grzybów	72
Klasyfikowanie grzybów przy wykorzystaniu wiedzy ludowej	73
Znajdowanie optymalnego punktu zwrotnego	74
Zysk informacyjny	75

Niejednorodność Giniego	76
Redukcja wariancji	77
Przycinanie drzew	77
Uczenie zespołowe	77
Pisanie klasyfikatora grzybów	79
Podsumowanie	87
6. Ukryte modele Markowa	89
Śledzenie zachowania użytkownika przy użyciu automatów skończonych.	89
Emisje/obserwacje stanów	91
Uproszczenie poprzez założenie Markowa	93
Wykorzystanie łańcuchów Markowa zamiast automatu skończonego	93
Ukryty model Markowa	94
Ocena: algorytm Naprzód-Wstecz	94
Matematyczne przedstawienie algorytmu Naprzód-Wstecz	94
Wykorzystanie zachowania użytkownika	96
Problem dekodowania poprzez algorytm Viterbiego	98
Problem uczenia	99
Oznaczanie części mowy z wykorzystaniem korpusu Browna	100
Uwagi przygotowawcze	100
Projekt kodowania i testowania	100
Podstawa naszego narzędzia do oznaczania części mowy: CorpusParser ...	101
Pisanie narzędzia do oznaczania części mowy	103
Sprawdzanie krzyżowe w celu potwierdzenia poprawności modelu.	110
Jak ulepszyć ten model.	111
Podsumowanie	112
7. Maszyny wektorów nośnych	113
Zadowolenie klientów jako funkcja tego, co mówią.	113
Klasyfikacja nastrojów przy użyciu maszyn wektorów nośnych	114
Teoria stojąca za maszynami wektorów nośnych	115
Granica decyzyjna	117
Maksymalizowanie granic.	117
Sztuczka jądrowa: transformacja cech.	118
Optymalizacja przez poluzowanie	120
Analizator nastrojów	121
Uwagi przygotowawcze	121
Projekt kodowania i testowania	121
Strategie testowania maszyny wektorów nośnych	122
Klasa Corpus	122
Klasa CorpusSet	125
Sprawdzanie poprawności modelu i klasyfikator nastrojów	128

Agregowanie nastrojów	132
Wykładnicza ważona średnia ruchoma	133
Mapowanie nastroju do wyniku finansowego	134
Podsumowanie	134
8. Sieci neuronowe	135
Czym jest sieć neuronowa?	135
Historia sieci neuronowych	136
Logika boolowska	136
Perceptrony	137
Jak konstruować sieci neuronowe ze sprzężeniem w przód	137
Warstwa wejściowa	138
Warstwy ukryte	140
Neurony	141
Funkcje aktywacyjne	142
Warstwa wyjściowa	147
Algorytmy uczące	147
Zasada delty	148
Propagacja wsteczna	149
QuickProp	149
RProp	150
Budowanie sieci neuronowych	151
Ile ukrytych warstw?	151
Ile neuronów dla każdej warstwy?	152
Tolerancja błędów i maksymalna liczba epok	152
Wykorzystanie sieci neuronowej do klasyfikowania języków	153
Uwagi przygotowawcze	153
Projekt kodowania i testowania	154
Dane	154
Pisanie testu podstawowego dla języka	154
Przejście do klasy Network	157
Dostrajanie sieci neuronowej	161
Precyzja i czułość w sieciach neuronowych	161
Podsumowanie przykładu	161
Podsumowanie	161
9. Grupowanie	163
Badanie danych bez żadnego błędu systematycznego	163
Kohorty użytkowników	164
Testowanie mapowań do grup	166
Zdatność grupy	166
Współczynnik zarysu	166

Porównywanie wyników z prawdą bazową.	167
Grupowanie K-średnich.	167
Algorytm K-średnich.	168
Słabe strony grupowania K-średnich.	169
Grupowanie przez maksymalizację wartości oczekiwanej	169
Algorytm	170
Twierdzenie o niemożności	171
Przykład: kategoryzowanie muzyki.	172
Uwagi przygotowawcze	172
Zbieranie danych	173
Projekt kodowania	173
Analizowanie danych przez algorytm K-średnich.	174
Grupowanie naszych danych	175
Wyniki z grupowania danych dotyczących muzyki jazzowej z użyciem maksymalizowania wartości oczekiwanej.	180
Podsumowanie.	182
10. Poprawianie modeli i wydobywania danych.	183
Klub dyskusyjny.	183
Wybieranie lepszych danych	184
Wybieranie cech	184
Wyczerpujące wyszukiwanie.	187
Losowe wybieranie cech	188
Lepszy algorytm wybierania cech	189
Wybieranie cech przez minimalizowanie redundancji i maksymalizowanie istotności.	190
Transformacja cech i rozkład macierzy	191
Analiza głównych składowych	191
Analiza niezależnych składowych	193
Uczenie zespołowe.	195
Grupowanie typu bootstrap	195
Boosting.	195
Podsumowanie.	197
11. Łączenie wszystkiego razem: Podsumowanie.	199
Przypomnienie algorytmów uczenia maszynowego.	199
Jak wykorzystywać te informacje do rozwiązywania problemów.	201
Co dalej?	202
Indeks	203
O autorze	209
Kolofon	209

Wstęp

Pierwsze wydanie książki *Uczenie maszynowe w praktyce* napisałem sfrustrowany brakiem dyscypliny swoich współpracowników. W roku 2009 pracowałem nad wieloma projektami związanymi z uczeniem maszynowym i zauważyłem, że gdy tylko wprowadzaliśmy pojęcie maszyn wektorów nośnych, sieci neuronowych itp., nagle podstawowe praktyki kodowania po prostu znikwały.

Moją odpowiedzią była książka *Uczenie maszynowe w praktyce*. W tym czasie pisałem 100% swojego kodu w języku Ruby i napisałem tę książkę z wykorzystaniem tego języka. Cieszę się, że mogę przedstawić nowe wydanie tej książki, przepisane z wykorzystaniem języka Python, co stanowiło nie lada wyzwanie. Przejrzałem większość rozdziałów, pozmieniałem przykłady i zaktualizowałem treść, aby była bardziej przydatna dla osób, które będą pisać kod związany z uczeniem maszynowym. Mam nadzieję, że spodoba się to czytelnikom.

Jak wspominałem w pierwszym wydaniu, moje drzwi zawsze stoją otworem. Jeśli ktoś chciałby ze mną porozmawiać, może się ze mną skontaktować pod adresem matt@mathewkirk.com. A jeśli ktoś trafi kiedyś do Seattle, możemy umówić się na spotkanie przy kawie.

Konwencje wykorzystywane w tej książce

Następujące konwencje typograficzne są używane w tej książce:

Kursywa

Wskazuje nowe pojęcia, adresy URL, adresy e-mail, nazwy plików i rozszerzenia plików.

Czcionka stałopozycyjna

Używana w przykładach programów, a także w treści akapitów przy odwołaniach do elementów programów, takich jak nazwy zmiennych lub funkcji, bazy danych, typy danych, zmienne środowiskowe, instrukcje i słowa kluczowe.

Czcionka stałopozycyjna z pogrubieniem

Pokazuje polecenia lub inny tekst, który powinien być dosłownie wpisany przez użytkownika.

Czcionka stałopozycyjna z kursywą

Pokazuje tekst, który powinien być zastąpiony wartościami podanymi przez użytkownika lub wynikającymi z kontekstu.



Ten element symbolizuje uwagę ogólną.

Korzystanie z przykładów kodu

Materiały dodatkowe (przykłady kodu, ćwiczenia, itd.) są dostępne do pobrania pod adresem <http://github.com/thoughtfulml/examples-in-python>.

Ta książka ma pomóc czytelnikom w realizacji własnych projektów. Jeśli jakiś kod przykładowy jest oferowany w tej książce, można z niego korzystać w swoich programach i dokumentacji. Nie trzeba uzyskiwać od nas pozwolenia, o ile nie kopiuje się znacznej ilości kodu. Na przykład napisanie programu, który wykorzystuje kilka fragmentów kodu z tej książki, nie wymaga zezwolenia. Sprzedawanie lub dystrybucja dysku CD-ROM z przykładami kodu z książek wydawnictwa O'Reilly wymaga pozwolenia. Cytowanie tej książki i przykładów kodu w odpowiedzi na czyjeś pytanie nie wymaga pozwolenia. Włączenie znaczącej ilości kodu przykładowego z tej książki do dokumentacji swojego produktu wymaga pozwolenia.

Doceniamy powołanie się na źródło, ale tego nie wymagamy. Zwykle zawiera ono tytuł, autora, wydawcę i numer ISBN książki.

Jeśli ktoś ma wątpliwości, czy użycie przykładów kodu z tej książki wymaga dodatkowego zezwolenia, może się z nami skontaktować pod adresem permissions@oreilly.com.

Jak się z nami skontaktować

Komentarze i pytania dotyczące tej książki należy kierować na adres wydawcy:

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (w USA lub Kanadzie)

707-829-0515 (połączenie międzynarodowe lub lokalne)

707-829-0104 (fax)

Przygotowaliśmy stronę WWW dla tej książki, na której umieszczamy erratę, przykłady i wszelkie dodatkowe informacje. Strona ta jest dostępna pod adresem <http://bit.ly/thoughtful-machine-learning-with-python>.

Aby przesłać komentarz lub zadać pytanie techniczne dotyczące tej książki, należy wysłać wiadomość pod adres bookquestions@oreilly.com.

Więcej informacji na temat naszych książek, kursów i konferencji można znaleźć w naszej witrynie WWW pod adresem <http://www.oreilly.com>.

Można nas znaleźć na Facebooku pod adresem: <http://facebook.com/oreilly>.

Można nas śledzić w serwisie Twitter: <http://twitter.com/oreillymedia>.

Można nas oglądać na YouTube: <http://www.youtube.com/oreillymedia>.

Podziękowania

Musiałem poczekać ponad rok na ukończenie tej książki. Zdiagnozowany u mnie rak jądra i nagła śmierć mojego ojca zmusiły mnie do przerwy i refleksji, zanim mogłem ponownie zabrać się za pisanie. Choć zajęło to dłużej niż wcześniej szacowałem, jestem całkiem zadowolony z wyniku.

Jestem wdzięczny wszystkim w wydawnictwie O'Reilly za wsparcie, jakie otrzymałem podczas pisania tej książki. Chciałbym szczególnie wyróżnić swoją redaktorkę Shannon Cut, która była dla mnie podporą i wsparciem; Liz Rush, która jako recenzent techniczny wytrwała przez cały proces pisania tej książki; Stephena Elstona, który dał mi wiele pomocnych uwag; Mike'a Loukidesa, za wsparcie mojego pomysłu i pomoc w przekształceniu go w dwie wydane książki; Alexeya Porotnikova, który pomógł mi opracować przykłady kodu w języku Python.

Chciałbym też specjalnie podziękować Alexeyowi Porotnikowowi (<https://github.com/alpo>) za pracowitą pomoc w przekonwertowaniu wszystkich tych przykładów z języka Ruby do Python, a także z wersji Python 2 do Python 3. Naprawdę dziękuję!

Jestem wdzięczny swoim przyjaciołom, szczególnie Curtis Fanta. Znamy się od małego. Dziękuję za poświęcony mi czas.

Chciałbym też podziękować swojej rodzinie. Moim bratankom Zoe i Darby za ciekawość i zdumienie. Bratu Jake'owi za zabawianie mnie nową muzyką i filmami. Mamie Carol za to, że pozwalała mi odkrywać świat i radziła zająć się fizyką (czego nie posłuchałem). Wszyscy wiele dla mnie znaczą.

Rodzinie Le za traktowanie mnie jak swojego. Dziękuję Lilianie za spotkania przy klockach Lego oraz Sayone i Alyssie za rozpromienianie mojego życia. Martinowi i Hanowi za ciągłe wsparcie i miłość. Thanh (tacie) i Kim (mamie) za karmienie mnie ponad miarę oraz za podarowanie mi mierników i książek na temat wzmacniaczy operacyjnych. Dziękuję za bycie częścią mojego życia.

Babci, która ciągle pytała, kiedy zobaczy okładkę. Zawsze pobudzała mnie do działania, czy to w harcerstwie, czy we własnym biznesie. Dziękuję za bycie zawsze przy mnie.

Dziękuję Sophii, mojej żonie. Rok temu byliśmy w szpitalnej sali, gdzie wpompowano we mnie mnóstwo środków przeciwbólowych...i przeżyliśmy. Jesteś stałą podporą mojego dorosłego życia. Gdy podejmuję się jakiegoś śmiałego wyzwania (jak napisanie książki), zawsze odkładasz swoje potrzeby na bok, aby się mną zająć. Jesteś dla mnie całym światem.

Wreszcie dziękuję mojemu tacie. Brakuje mi Twoich odwiedzin i naszych wypraw pod namiot do lasu. Chciałbym dzielić się z Tobą tym osiągnięciem, ale doceniam czas, jaki spędziliśmy razem. Ta książka jest dla Ciebie.

W przybliżeniu prawdopodobnie poprawne oprogramowanie

Lecąc samolotem, korzystamy z jednej z najbezpieczniejszych form podróżowania na świecie. Szanse na utratę życia podczas lotu samolotem wynoszą 1 na 29,4 milionów¹, co oznacza, że decydując się na zawód pilota pasażerskiego, można nigdy nie uczestniczyć w wypadku podczas całej 40-letniej kariery. Te szanse są szokujące, biorąc pod uwagę, jak skomplikowanymi urządzeniami są samoloty. Nie zawsze jednak tak było.

Rok 2014 był zły dla lotnictwa; odnotowano wtedy 824 zgony związane z lotnictwem², w tym zaginięcie samolotu linii Malaysia Air. W roku 1929 było 257 ofiar³ wypadków lotniczych. Wygląda to, jakby bezpieczeństwo lotów uległo pogorszeniu, dopóki nie zdamy sobie sprawy, że w samych Stanach Zjednoczonych odbywa się obecnie ponad 10 milionów lotów rocznie, natomiast w roku 1929 było ich znacznie mniej – mniej więcej od 50000 do 100000. Oznacza to, że ogólne prawdopodobieństwo śmierci w katastrofie lotniczej zmalało od 1929 do 2014 roku z 0,25% do 0,00824%.

Podróże lotnicze zmieniły się przez lata, tak samo jak tworzenie oprogramowania. Podczas gdy w roku 1929 tworzenie oprogramowania w postaci znanej obecnie nie istniało, to przez ostatnie lata zbudowaliśmy i zawaliliśmy wiele projektów programistycznych.

Do niedawnych przykładów należą takie projekty programistyczne, jak uruchomienie amerykańskiego serwisu healthcare.gov, który okazał się finansową katastrofą, kosztującą około 634 miliony dolarów⁴. Jeszcze gorsze są projekty programistyczne, które mają katastrofalne błędy. W roku 2013 giełda NASDAQ została zamknięta z powodu błędu oprogramowania i została ukarana grzywną w wysokości 10 milionów USD⁵. W roku 2014 byliśmy świadkami rozprzestrzeniania się błędu Heartbleed, który naraził na niebezpieczeństwo

1 <http://www.statisticbrain.com/airplane-crash-statistics/>

2 <http://bit.ly/2014-aviation>

3 <http://bit.ly/casualties-1929>

4 <http://bit.ly/cost-healthcaregov>

5 <http://reut.rs/2i2HfgS>

wiele witryn korzystających z SSL. W konsekwencji firma CloudFlare musiała odwołać ponad 100000 certyfikatów SSL, co kosztowało ich wiele milionów⁶.

Oprogramowanie i samoloty mają jedną wspólną cechę: są skomplikowane i skutki ewentualnej awarii są katastrofalne i widoczne publicznie. Linie lotnicze były w stanie zapewnić bezpieczeństwo podróży i zmniejszyć prawdopodobieństwo katastrof lotniczych o ponad 96%. Niestety nie możemy powiedzieć tego samego o oprogramowaniu, które staje się jeszcze bardziej skomplikowane. Katastrofalne błędy uderzają w nas regularnie, powodując miliardowe straty.

Dlaczego podróże lotnicze stały się tak bezpieczne, a oprogramowanie tak wadliwe?

Prawidłowe pisanie oprogramowania

Między rokiem 1929 a 2014 samoloty stały się bardziej skomplikowane, większe i szybsze. Jednak z tym wzrostem wiązały się też większe regulacje ze strony FAA oraz instytucji międzynarodowych, a także kultura stosowania list kontrolnych przez pilotów.

Choć technologie i sprzęt komputerowy zmieniały się bardzo szybko, to uruchamianie na nich oprogramowanie zmieniało się znacznie wolniej. Nadal korzystamy głównie z kodu proceduralnego i zorientowanego obiektowo, który nie wykorzystuje w pełni możliwości obliczeń równoległych. Programiści podjęli jednak kroki w kierunku opracowania wytycznych dotyczących pisania oprogramowania oraz stworzenia kultury testowania kodu. Doprowadziło to do przyjęcia zasad SOLID i metodologii TDD. SOLID jest zestawem zasad, które prowadzą nas do pisania lepszego kodu, a TDD oznacza projektowanie sterowane testami (test-driven design) lub programowanie sterowane testami (test-driven development). Omówimy te dwa modele i ich związki z prawidłowym pisanem oprogramowania oraz porozmawiamy o refaktoringu oprogramowania.

SOLID

Zasady SOLID stanowią platformę, która pomaga w projektowaniu lepszego kodu zorientowanego obiektowo. W taki sam sposób, jak FAA (Federal Aviation Administration – Federalna Administracja Lotnictwa) definiuje, jak *powinny* działać linie lotnicze lub samoloty, tak samo zasady SOLID mówią nam, jak *powinniśmy* tworzyć oprogramowanie. Naruszenia regulacji FAA czasem się zdarzają i mogą mieć różne skutki, od katastrofalnych do drobnych. To samo odnosi się do zasad SOLID. Zasady te mogą mieć czasem ogromne znaczenie, ale najczęściej są po prostu wskazówkami. Zasady SOLID wprowadził Robert Martin jako pięć zasad tworzenia oprogramowania. Motywem do ich opracowania było pisanie lepszego kodu, który będzie łatwy w utrzymaniu, zrozumiały i stabilny. Michael Feathers wymyślił mnemotechniczny skrót *SOLID* (od pierwszych liter nazw poszczególnych zasad) do łatwiejszego ich zapamiętania.

⁶ <http://bit.ly/cost-heartbleed>

SOLID oznacza:

- SRP – Single Responsibility Principle (zasada pojedynczej odpowiedzialności)
- OCP – Open/Closed Principle (zasada otwarte-zamknięte)
- LSP – Liskov Substitution Principle (zasada podstawienia Liskov)
- ISP – Interface Segregation Principle (zasada rozdzielania interfejsów)
- DIP – Dependency Inversion Principle (zasada odwrócenia zależności)

Zasada pojedynczej odpowiedzialności

Zasada pojedynczej odpowiedzialności (SRP) stała się jednym z bardziej rozpowszechnionych elementów pisania dobrego kodu zorientowanego obiektowo. Pojedyncza odpowiedzialność powoduje definiowanie prostych klas lub obiektów. Tę samą mentalność można stosować w programowaniu funkcjonalnym, tworząc prostsze funkcje. Cała koncepcja opiera się na prostocie. Dany element oprogramowania powinien zajmować się jedną i tylko jedną kwestią. Dobrym przykładem naruszenia zasady SRP jest szczerpek wielofunkcyjny (rysunek 1-1). Może zrobić niemal wszystko, ale użyteczność poszczególnych elementów jest ograniczona.



Rysunek 1-1 Szczerpek wielofunkcyjny ma zbyt wiele odpowiedzialności.

Zasada otwarte-zamknięte

Zasada OCP, określana też czasem mianem hermetyzacji, mówi, że obiekty powinny być otwarte na rozszerzanie, ale nie na modyfikowanie. Można to pokazać na przykładzie obiektu zliczającego, z którym jest związany wewnętrzny licznik. Obiekt ten ma metody `increment` i `decrement`. Obiekt ten nie powinien zezwalać innym elementom na zmienianie wewnętrznego licznika poza zdefiniowanym interfejsem API, ale może być rozszerzany (np., aby powiadamiać kogoś o zmianie licznika przez obiekt typu `Notifier`).

Zasada podstawienia Liskov

Zasada LSP określa, że dowolny podtyp powinien być łatwo zastępowalny elementem występującym niżej w drzewie obiektów, bez efektów ubocznych. Na przykład model samochodu można podstawić zamiast rzeczywistego samochodu.

Zasada rozdzielania interfejsów

Zasada ISP stwierdza, że występowanie wielu interfejsów specyficznych dla klientów jest lepsze niż ogólny interfejs dla wszystkich klientów. Ta zasada dotyczy upraszczania wymiany danych pomiędzy różnymi elementami. Dobrym przykładem jest segregowanie śmieci na organiczne, do ponownego przetworzenia i pozostałe odpady. Zamiast jednego dużego kubła, mamy trzy dla poszczególnych typów śmieci.

Zasada odwrócenia zależności

Zasada DIP radzi nam polegać na abstrakcjach, a nie na elementach konkretnych. Mówi nam, że powinniśmy budować drzewo dziedziczenia obiektów. Przykład użyty w oryginalnej pracy⁷ Roberta Martina wyjaśnia, że powinniśmy utworzyć klasę `KeyboardReader` dziedziczącą po ogólnej klasie `Reader`, zamiast umieszczać wszystko w jednej klasie. Pasuje to też dobrze do rady Arthura Riela dotyczącej unikania *boskich klas*, pochodzącej z jego pracy *Heurystyka projektowania zorientowanego obiektowo*. Choć moglibyśmy bezpośrednio przylutować przewód prowadzący z gitary do wzmacniacza, byłoby to pewnie niewydolne i nie brzmiałoby zbyt dobrze.



Zestaw zasad SOLID przeszedł próbę czasu i został przedstawiony w wielu książkach autorstwa Martina i Feathersa, a także pojawił się w książce Sandi Metz *Praktyczne projektowanie zorientowane obiektowo w języku Ruby*⁸. Zestaw ten ma służyć jako wskazówki i przypominać nam o prostych sprawach przy programowaniu, aby tworzony przez nas kod był jak najlepszy. Wskazówki te pomagają pisać poprawne architektonicznie oprogramowanie.

Testowanie albo TDD

We wczesnych dniach lotnictwa piloci nie używali list kontrolnych do sprawdzania, czy samolot jest gotowy do startu. W książce *The Right Stuff* Tom Wolfe pisał, że większość wczesnych pilotów testowych takich jak Chuck Yeager kierowało się wyczuciem i swoim

⁷ Robert Martin, „The Dependency Inversion Principle”, <http://bit.ly/the-DIP>.

⁸ <http://poodr.info/>

talentem przy opanowywaniu złożoności samolotu. To prowadziło również do tego, że jedna czwarta pilotów testowych zginęła w akcji.⁹

Obecnie sprawy mają się inaczej. Przed startem piloci przeprowadzają mnóstwo weryfikacji. Niektóre z nich mogą wydawać się niepotrzebnie uciążliwe, jak na przykład przedstawienie się imieniem innym członkom załogi. Wyobraźmy sobie jednak, że samolot wpadnie w korkociąg i trzeba natychmiast poinformować kogoś o problemie. Nie znając imienia, ciężko byłoby się komunikować.

To samo można odnieść wobec dobrego oprogramowania. Regularne uruchamianie zestawu systematycznych sprawdzeń w celu przetestowania, czy nasze oprogramowanie działa poprawnie, czy nie, pozwala na jego spójne działanie.

We wczesnych dniach tworzenia oprogramowania większość testów była przeprowadzana po napisaniu właściwego oprogramowania (jest to tak zwany model kaskadowy¹⁰, wykorzystywany przez NASA i inne organizacje do projektowania oprogramowania i testowania go). Współdziałało to dobrze z powszechnym wówczas stylem zarządzania projektami. Podobnie do sposobu budowania samolotów, w przypadku oprogramowania najpierw powstawał jego projekt, następnie pisano kod zgodnie ze specyfikacjami, a potem testowano program przed dostarczeniem go klientowi. Ta metoda testowania mogła zajmować miesiące a nawet lata. Doprowadziło to do sformułowania *Manifestu programowania zwinnego*¹¹, a także do powstania kultury testowania i programowania sterowanego testami (TDD), zainicjowanego przez Kenta Becka, Warda Cunninghama i wielu innych.

Koncepcja programowania sterowanego testami jest prosta: najpierw piszemy test sprawdzający, co chcemy osiągnąć; wykonujemy go, potwierdzając, że daje wynik negatywny; piszemy kod naprawiający test; a następnie poprawiamy kod, aby dopasować go do wskázówek SOLID. Choć wiele osób zwraca uwagę, że wydłuża to cykl tworzenia oprogramowania, to znacząco ogranicza liczbę błędów w kodzie i poprawia stabilność działania oprogramowania.¹²

Samoloty, które mają niską tolerancję na błędy, często działają w ten sam sposób. Zanim pilot będzie mógł zasiąść za sterami Boeinga 787, spędza X godzin w symulatorze lotów, żeby zrozumieć i sprawdzić swoją znajomość samolotu. Samoloty są testowane przed startem, a później również w trakcie lotu. Z nowoczesnym tworzeniem oprogramowania jest bardzo podobnie. Testujemy swoją wiedzę, pisząc testy przed wdrożeniem oprogramowania, a także monitorując już wdrożone oprogramowanie.

Nadal pozostaje jednak jeden problem: w rzeczywistości, w której nic nie pozostaje takie samo, napisanie testu nie sprawia, że kod będzie dobry. David Heinemer Hanson w swojej popularnej prezentacji na temat szkód wynikających z podejścia sterowanego

9 Atul Gawande, *The Checklist Manifesto* (New York: Metropolitan Books), str. 161.

10 https://en.wikipedia.org/wiki/Waterfall_model

11 <http://agilemanifesto.org/>

12 Nachiappan Nagappan i in., „Realizing Quality Improvement through Test Driven Development: Results and Experience of Four Industrial Teams”, *Empirical Software Engineering* 13, no. 3 (2008): 289 – 302, <http://bit.ly/Nagappanetal>.

testami¹³ pokazał, jak ślepe oddanie technikom TDD i SOLID może prowadzić do skomplikowanego kodu. Większość jego uwag jest związanych z niepotrzebnymi komplikacjami wynikającymi z wyciągania drobnych fragmentów kodu do osobnych klas lub pisania kodu, który łatwo testować, ale ciężko czytać. Tutaj pojawia się kolejny czynnik związany z prawidłowym pisaniem oprogramowania: refaktoring.

Refaktoring

Refaktoring jest jedną z najtrudniejszych do wytłumaczenia osobom nieprogramującym praktyk programistycznych. Podczas lotu samolotem widzimy jedynie 20% z tego, co sprawia, że samolot leci. Pod warstwami aluminium i tytanu znajdują się złożone systemy elektryczne, które zasilają oświetlenie awaryjne, hydraulikę, konstrukcję nośną, która ma być lekka, a przy tym wytrzymała – zbyt wiele, aby wszystko tutaj wymienić. Wyjaśnianie komuś, co składa się na konstrukcję samolotu, jest jak wyjaśnianie komuś, że pod zlewem są rury.

Refaktoring bazuje na istniejącej strukturze i ją ulepsza. Polega na czyszczeniu istniejącego kodu, aby przeglądając go, łatwo było zrozumieć, co się dzieje. Samoloty są projektowane bardzo rygorystycznie, ale oprogramowanie nie. Oprogramowanie zmienia się bardzo szybko. Wiele firm nieustannie wdraża oprogramowanie w środowisku produkcyjnym. Całe to projektowanie nowych funkcji może czasem powodować tzw. „dług techniczny”.

Dług techniczny, nazywany też *długiem projektowym* lub *długiem kodu*, jest metaforą dla słabego zaprojektowania systemu, który ulega degradacji w czasie życia projektów programistycznych. Problemem długu technicznego jest nawarstwianie się „odsetek”, które w końcu blokują przyszły rozwój oprogramowania.

Jeśli ktoś pracował długo nad jakimś projektem, to zna to uczucie, gdy mamy szybkie nowe wersje w początkowym okresie rozwoju i stagnację pod koniec. Dług techniczny w wielu przypadkach wynika z braku testów albo niestosowania zasad SOLID.

Dług techniczny niekoniecznie jest zły – czasami trzeba projekty przepchnąć wcześniej, aby wspomóc rozwój biznesu – ale „niespłacanie” go sprawi, że narosną tak duże „odsetki”, że projekt zostanie zniszczony. Sposobem na poradzenie sobie z tym problemem jest refaktoring kodu.

Poprzez refaktoring przybliżamy nasz kod do wskazówek SOLID i bazy kodu opartej na TDD. Stosując się do poniższych wskazówek, możemy czyścić istniejący kod i ułatwiać nowym programistom dołączanie do pracy nad jego dalszym rozwojem:

1. Stosowanie wskazówek SOLID
 - a. Zasada pojedynczej odpowiedzialności
 - b. Zasada otwarte-zamknięte
 - c. Zasada podstawienia Liskov

¹³ <http://bit.ly/test-induced-damage>

- d. Zasada rozdzielania interfejsów
 - e. Zasada odwrócenia zależności
2. Implementowanie TDD (programowania/projektowania sterowanego testami)
 3. Refaktoring kodu w celu uniknięcia narastania długu technicznego

Co tak właściwie sprawia, że oprogramowanie jest prawidłowe?

Pisanie prawidłowego oprogramowania

Pisanie prawidłowego oprogramowania jest dużo trudniejsze niż prawidłowe pisanie oprogramowania. W swojej książce *Specification by Example* Gojko Adzic twierdzi, że najlepszym podejściem do pisania oprogramowania jest utworzenie najpierw specyfikacji, a następnie bezpośrednia praca z użytkownikami tego oprogramowania. Dopiero po ukończeniu specyfikacji można pisać kod odpowiadający tej specyfikacji. W praktyce nie zawsze się to sprawdza – czasami świat nie jest taki, jak nam się wydaje. Nasz początkowy model może stać się nieaktualny.

Na przykład firmie Webvan nie udało się zbudować biznesu opartego na sprzedaży artykułów spożywczych przez sieć. Mieli niemal 400 milionów \$ kapitału inwestycyjnego i szybko zbudowaną infrastrukturę do wspierania (jak sądzili) dobrze rozwijającego się biznesu. Niestety wszystko okazało się kląpą ze względu na koszt dostarczania jedzenia i przeszacowany rynek klientów chcących kupować artykuły spożywcze przez Internet. Ze wszech miar udało im się napisać oprogramowanie i zbudować biznes, ale rynek nie był jeszcze na to gotowy i szybko zbankrutowali. Obecnie znaczna część zbudowanej wtedy infrastruktury jest wykorzystywana przez Amazon.com do wspierania AmazonFresh.

W teorii praktyka i teoria są tym samym. W praktyce nie są.

– *Albert Einstein*

Obecnie jesteśmy w momencie, gdy teoretycznie możemy poprawnie pisać oprogramowanie i będzie ono działać, ale pisanie odpowiedniego oprogramowania jest znacznie bardziej skomplikowanym problemem. Tutaj przydaje się uczenie maszynowe.

Pisanie odpowiedniego oprogramowania przy zastosowaniu uczenia maszynowego

W książce *The Knowledge-Creating Company* Nonaka i Takeuchi zarysowali, co sprawiło, że japońskie firmy odnosiły takie sukcesy w latach 80-ych XX wieku. Zamiast stosować podejście „od ogółu do szczegółu” przy rozwiązywaniu problemów, uczyły się i zdobywały wiedzę. Pochodzący z tej książki przykład z zagniataniem ciasta na chleb i opracowaniem

urządzenia do wypieku chleba jest doskonałym przykładem iteracji, który można łatwo zastosować w opracowywaniu oprogramowania.

Możemy pójść jeszcze dalej, korzystając z uczenia maszynowego.

Czym dokładnie jest uczenie maszynowe?

Według większości definicji uczenie maszynowe jest zbiorem algorytmów, technik i branżowych sztuczek, które pozwalają maszynom uczyć się na podstawie danych – czyli czegoś, co można przestawić w formie liczbowej (macierze, wektory, itd.).

Aby jednak lepiej zrozumieć uczenie maszynowe, przyjrzyjmy się, jak powstało. W latach 50-ych XX wieku prowadzono szeroko zakrojone badania nad grą w warcaby. Wiele spośród tych modeli skupiało się na lepszej grze i wymyśleniu optymalnych strategii. Moglibyśmy pewnie obecnie stworzyć prosty program grający w warcaby, cofając się od sytuacji wygrywającej, rozrysowując drzewo decyzyjne i optymalizując posunięcia.

Jest to jednak dość zawężony i dedukcyjny sposób rozumowania. Trzeba by w efekcie zaprogramować agenta podejmującego decyzje. W większości tych wczesnych programów nie było zaprogramowanego kontekstu albo irracjonalnych zachowań.

Niemal 30 lat później uczenie maszynowe zaczynało się gwałtownie rozwijać. Wiele tegich umysłów zaczęło pracować nad problemami obejmującymi filtrowanie spamu, klasyfikowanie i ogólną analizę danych.

Ważną zmianą jest tutaj przeniesienie nacisku z komputerowej dedukcji na komputerową indukcję. Jak w przypadku Sherlocka Holmesa, dedukcja obejmuje zastosowanie skomplikowanych modeli logicznych do uzyskania wniosków. Z kolei indukcja obejmuje przyjęcie danych za prawdziwe i próbę dopasowania modelu do tych danych. Zmiana ta przyczyniła się do wielu ważnych postępów w znajdowaniu dobrych rozwiązań dla typowych problemów.

Problemem przy rozumowaniu indukcyjnym jest jednak to, że możemy algorytmowi dostarczać jedynie dane, które *znamy*. Ilościowe określanie niektórych kwestii jest niezwykle trudne. W jaki sposób na przykład wyrazić liczbowo, jak milutko wygląda kotek na zdjęciu?

W ostatnich 10 latach jesteśmy świadkami renesansu dogłębnego uczenia, które pomaga złączyć te problemy. Zamiast polegać na danych zakodowanych przez ludzi, algorytmy są w stanie znajdować punkty danych, których wcześniej nie byliśmy w stanie określać liczbowo.

Brzmi to świetnie, ale możliwości te wiążą się z niezwykle wysokimi kosztami i odpowiedzialnością.

Wysoko oprocentowany dług uczenia maszynowego

W artykule opublikowanym niedawno przez Google i zatytułowanym „Machine Learning: The High Interest Credit Card of Technical Debt” (Uczenie maszynowe: dług techniczny

z wysoko oprocentowanej karty kredytowej)¹⁴, Sculley i in. wyjaśniają, że projekty uczenia maszynowego również cierpią na problemy związane z długiem technicznym (tabela 1-1).

Zauważyli, że projekty uczenia maszynowego są z natury skomplikowane, mają niejasno określone granice, polegają w dużym stopniu na zależnościach od danych, cierpią z powodu mieszanego kodu na poziomie systemowym i mogą się znacznie zmieniać ze względu na zmiany w świecie zewnętrznym. Wymieniają problemy, które są szczególnie związane z projektami uczenia maszynowego.

Zamiast omawiać poszczególne problemy, pomyślałem, że ciekawe będzie powiązanie naszego wcześniejszego przeglądu zasad SOLID i TDD oraz refaktoringu i zobaczenie, jak odnosi się to do kodu uczenia maszynowego.

Tabela 1-1 Wysoko oprocentowany dług uczenia maszynowego

Problem uczenia maszynowego	Przejawia się jako	Naruszenie zasady SOLID
Spłątanie	Zmiana jednego czynnika zmienia wszystko	SRP
Ukryte pętle sprzężenia zwrotnego	Wbudowane, ukryte funkcje w modelu	OCP
Niezadeklarowany dług klientów/ widoczności		ISP
Niestabilne zależności między danymi	Ulotne dane	ISP
Nie w pełni wykorzystywane zależności między danymi	Niewykorzystywane wymiary	LSP
Kaskada poprawek		*
Kod spajający	Pisanie kodu, który zajmuje się wszystkim	SRP
Gęszcz potoków	Przesyłanie danych przez złożone przepływy zadań	DIP
Ścieżki eksperymentalne	Ślepe ścieżki, które nigdzie nie prowadzą	DIP
Dług konfiguracji	Wykorzystywanie starych konfiguracji dla nowych danych	*
Stałe pułapy w dynamicznym świecie	Mała elastyczność wobec zmian w korelacjach	*
Zmiany korelacji	Modelowanie korelacji zamiast przyczynowości	Specyficzne dla uczenia maszynowego

Zastosowanie zasad SOLID w uczeniu maszynowym

Zasady SOLID, jak pamiętamy, są jedynie wskazówkami przypominającymi nam o pewnych celach przy pisaniu kodu zorientowanego obiektowo. Wiele algorytmów uczenia maszynowego z zasady nie jest obiektowo zorientowane. Są one funkcjonalne, matematyczne i wykorzystują statystykę, ale nie zawsze tak jest. Zamiast myśleć o sprawach w czysto

¹⁴ <http://research.google.com/pubs/pub43146.html>