THE ART OF CLEAN CODE

JAK ELIMINOWAĆ ZŁOŻONOŚĆ
I PISAĆ CZYSTY KOD

CHRISTIAN MAYER

Helion
no starch press

Tytuł oryginału: The Art of Clean Code: Best Practices to Eliminate Complexity and Simplify Your Life

Tłumaczenie: Robert Górczyński

ISBN: 978-83-8322-064-2

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/theart
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/theart.zip

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

1

Negatywny wpływ złożoności na produktywność

W tym rozdziale zamierzam wyjaśnić, jak ważny i niedoceniany jest temat złożoności. Czym dokładnie jest *złożoność*? Kiedy mamy z nią do czynienia? Dlaczego może negatywnie wpływać na produktywność? Złożoność jest przeciwieństwem prostej i efektywnej organizacji i jednostki, więc warto dokładniej przyjrzeć się wszystkim obszarom, na których może się pojawiać, i poznać przybierane przez nią formy. W tym rozdziale skoncentruję się na problemie — złożoności — natomiast w pozostałych przedstawię efektywne metody walki z nią przez przekierowanie zwolnionych zasobów pochłanianych wcześniej przez złożoność.

Rozpocznę od krótkiego wyjaśnienia, gdzie złożoność może być przytłaczająca dla nowego programisty:

- wybór języka programowania;
- wybór projektu, nad którym będzie prowadzona praca — spośród tysięcy projektów open source i niezliczonych problemów;
- wybór biblioteki (np. scikit-learn kontra NumPy kontra TensorFlow);
- wybór technologii (aplikacje Alexa, aplikacje dla smartfonów, aplikacje oparte na interfejsie przeglądarki WWW, aplikacje zintegrowane z serwisami FaceBook lub WeChat, aplikacje rzeczywistości wirtualnej);

- wybór edytora do tworzenia kodu źródłowego (np. PyCharm, IDLE lub Atom).
- Biorąc pod uwagę zamieszanie powodowane przez te źródła złożoności, nie powinno być zaskoczeniem, że *„Od czego mam zacząć?”* to jedno z najczęściej pojawiających się pytań w głowach początkujących programistów.

Najlepszym sposobem *nie* będzie wybór książki dotyczącej programowania i zapoznanie się ze wszystkimi syntaktycznymi cechami języka programowania. Niejeden ambitny student kupuje na zachętę książkę dotyczącą programowania, a następnie poznanie języka programowania umieszcza na liście rzeczy do zrobienia — skoro zostały wydane pieniądze na książkę, to lepiej ją przeczytać, ponieważ w przeciwnym razie te pieniądze będą zmarnowane. Jednak podobnie jak z wieloma innymi punktami na liście rzeczy do zrobienia, także *przeczytanie książki dotyczącej programowania* rzadko zostaje ukończone.

Najlepszym sposobem na rozpoczęcie będzie wybór projektu programistycznego — prostego w przypadku początkujących — i jego ukończenie. Zanim projekt zostanie w pełni ukończony, nie czytaj książek o programowaniu ani samouczków przypadkowo znalezionych w internecie. Nie przeglądaj w nieskończoność serwisu StackOverflow. Po prostu utwórz projekt i rozpocznij pisanie kodu źródłowego, wykorzystując swoje ograniczone możliwości i zdrowy rozsądek. Jedna z moich studentek chciała opracować aplikację finansową oferującą panel graficzny i sprawdzającą wcześniejsze stopy zwrotu wielu inwestycji, aby umożliwić odpowiedzi na pytania w stylu „Jaki był maksymalny roczny spadek portfela składającego się w połowie z akcji i w połowie z obligacji skarbowych”? Na początku nie wiedziała, jak zabrać się za ten projekt, ale wkrótce natrafiła na framework Python Dash, przeznaczony do tworzenia aplikacji internetowych opartych na danych. Dowiedziała się, jak skonfigurować serwer, oraz poznała technologie HTML (ang. *hypertext markup language*) i CSS (ang. *cascading style sheets*) w stopniu pozwalającym kontynuować pracę. Obecnie jej aplikacja działa i pomaga tysiącom użytkowników w podjęciu właściwych decyzji dotyczących inwestycji. Co ważniejsze, dołączyła do zespołu programistów rozwijających framework Python Dash, a nawet jest współautorką wydanej przez No Starch Press książki o tym narzędziu. Tego wszystkiego dokonała w ciągu roku — skoro ona mogła, Ty również możesz. Jest całkiem zrozumiałe, jeśli na początku nie do końca rozumiesz to, co robisz, i będziesz powoli poszerzać wiedzę. Czytaj artykuły tylko po to, aby dokonać pewnego postępu w trakcie pracy nad projektem. Podczas pracy nad swoim pierwszym projektem spotkasz się z wieloma problemami, m.in.:

Którego edytora kodu źródłowego używać?

Jak zainstalować język programowania używany w tym projekcie?

W jaki sposób odczytywać dane wejściowe z pliku?

W jaki sposób przechowywać dane wejściowe w programie, aby można było później ich używać?

W jaki sposób przetwarzać dane wejściowe, aby otrzymać żądany wynik?

Udzielając odpowiedzi na te pytania, stopniowo nabędziesz wszechstronnych umiejętności. Wraz z upływem czasu nauczysz się lepiej i łatwiej odpowiadać na tego rodzaju pytania. Zyskasz umiejętność rozwiązywania znacznie większych problemów oraz przygotujesz wewnętrzną bazę danych wzorców programistycznych i koncepcyjnych. Nawet zaawansowani programiści nieustannie się uczą i rozwijają, wykorzystując podany proces — tyle że tworzone przez nich projekty są znacznie większe i dużo bardziej skomplikowane.

Dzięki takiemu podejściu opartemu na projekcie prawdopodobnie okaże się, że zmagasz się ze złożonością na obszarach takich jak wyszukiwanie błędów w nieustannie powiększających się bazach kodu, zrozumienie komponentów kodu i zachodzących między nimi interakcji, wybór odpowiedniej funkcjonalności, która powinna być zaimplementowana jako następna, zrozumienie matematycznych i koncepcyjnych podstaw kodu źródłowego.

Złożoność jest wszędzie i pojawia się na każdym etapie projektu. Ukrytym kosztem złożoności jest to, że początkujący programiści poddają się, a ich projekty nigdy nie ujrzą światła dziennego. Dlatego rodzi się pytanie: jak poradzić sobie z problemem złożoności?

Odpowiedź na nie jest oczywista: chodzi o *prostotę*. Poszukaj prostoty i skupiaj się na każdym etapie tworzenia kodu źródłowego. Jeżeli z tej książki możesz wynieść tylko jedno, niech będzie to następujące zdanie: na każdym etapie programowania przyjmij radykalnie minimalistyczne podejście. W książce omówię następujące metody:

- Uporządkuj swój dzień, wykonuj mniej zadań i skup się na tych, które mają znaczenie. Na przykład zamiast rozpoczynać równocześnie 10 nowych interesujących projektów programistycznych, starannie wybierz tylko jeden, a następnie cały wysiłek skoncentruj na jego ukończeniu. W rozdziale 2. znacznie dokładniej poznasz zastosowanie reguły 80/20 w programowaniu.
- Po wybraniu jednego projektu programistycznego pozbądź się z niego wszystkich niepotrzebnych funkcjonalności i skup na opracowaniu produktu o minimalnej niezbędnej funkcjonalności (zobacz rozdział 3.), dokończ jego budowanie i tym samym szybko i efektywnie sprawdź poprawność przyjętych hipotez.
- Gdzie tylko możesz, twórz prosty i zwięzły kod. W rozdziale 4. przedstawię wiele praktycznych podpowiedzi związanych z tą regułą.
- Ogranicz nakład czasu i pracy poświęcony na przedwczesną optymalizację. Niepotrzebna optymalizacja kodu źródłowego to jedna z przyczyn niepotrzebnej złożoności (więcej informacji na ten temat znajdziesz w rozdziale 5.).
- Zmniejsz ilość czasu potrzebnego na przestawianie się między zadaniami. W tym celu zarezerwuj większe bloki czasu na programowanie, aby osiągnąć w ten sposób stan *przepływu* — jest to pojęcie psychologiczne opisujące stan skupienia, dzięki któremu można zwiększyć swoje zaangażowanie, skupienie i produktywność. Stanowi przepływu poświęciłem rozdział 6.

- Zastosuj filozofię systemu UNIX i twórz kody funkcji w taki sposób, aby koncentrowały się na dobrym wykonaniu tylko jednego zadania. W rozdziale 7. znajdziesz szczegółowe informacje o filozofii system UNIX oraz przykłady jej zastosowania w kodzie pisanym w Pythonie.
- Stosuj prostotę w projektach, aby w ten sposób tworzyć piękne, przejrzyste i odpowiednie interfejsy użytkownika, które będą intuicyjne i łatwe w użyciu (zobacz rozdział 8.).
- Techniki skupienia stosuj podczas planowania kariery, następnego projektu, dnia pracy lub obszaru doświadczenia (zobacz rozdział 9.)

Zagłębimy się teraz z koncepcję złożoności, aby dokładnie poznać jednego z największych wrogów produktywności podczas tworzenia kodu źródłowego.

Czym jest złożoność?

Na różnych polach pojęcie *złożoności* ma odmienne znaczenie. Czasami jest ściśle zdefiniowane, np. *złożoność obliczeniowa* programu komputerowego pozwalającego analizować kod danej funkcji pod kątem różnych danych wejściowych. Z kolei w innych sytuacjach złożoność jest luźno zdefiniowana jako pewna wielkość lub struktura interakcji między komponentami systemu. W tej książce będę używał pojęcia w bardziej ogólnym znaczeniu.

Złożoność będzie zdefiniowana następująco:

Złożoność — pewna całość składająca się z elementów, które są trudne do przeanalizowania, zrozumienia lub wyjaśnienia.

Złożoność dotyczy całego systemu bądź encji. Ponieważ złożoność utrudnia wyjaśnienie systemu, prowadzi do niepotrzebnych zmagań i dezorientacji. Systemy pozostające w codziennym użyciu są zagmatwane, więc ze złożonością można spotkać się dosłownie wszędzie: na giełdzie papierów wartościowych, w trendach mediów społecznościowych, w punktach widzenia polityków, w ogromnych programach komputerowych składających się z setek tysięcy wierszy kodu — przykładem może być tutaj system operacyjny Windows.

Jeżeli zajmujesz się programowaniem, masz szczególną podatność na wszechobecną złożoność, np. pochodzącą z następujących źródeł, które omówię w tym rozdziale:

złożoność cyklu życiowego projektu,

złożoność w teorii oprogramowania i algorytmów,

złożoność w nauce,

złożoność w procesach,

złożoność w serwisach społecznościowych,

złożoność w życiu codziennym.

Złożoność cyklu życiowego projektu

Przeanalizujmy różne etapy cyklu życiowego projektu: planowanie, definiowanie, projektowanie, kompilowanie, testowanie i wdrożenie (zobacz rysunek 1.1).

Rysunek 1.1. Sześć faz koncepcyjnych projektu oprogramowania, na podstawie oficjalnego standardu IEEE (Institute of Electrical and Electronics Engineers) inżynierii oprogramowania

Nawet jeśli pracujesz nad bardzo małym projektem oprogramowania, prawdopodobnie i tak przechodzisz przez te wszystkie sześć faz cyklu życiowego. Pamiętaj, że przejście przez daną fazę niekoniecznie będzie odbywało się tylko jednokrotnie — ogólnie rzecz biorąc, w nowoczesnych metodach tworzenia oprogramowania preferowane jest podejście iteracyjne, w którym poszczególne fazy odbywają się wielokrotnie. W kolejnych punktach wyjaśnię, jaki wpływ na każdą z tych faz ma złożoność.

Planowanie

Pierwszą fazą cyklu życiowego podczas tworzenia oprogramowania jest planowanie, które w literaturze specjalistycznej czasami określane jest jako **analiza wymagań**. Celem tej fazy jest określenie, jak powinien przedstawiać się produkt. Zakończona sukcesem faza planowania prowadzi do ściśle zdefiniowanego zbioru funkcjonalności, które trzeba dostarczyć użytkownikowi końcowemu.

Niezależnie od tego, czy tylko w pojedynkę pracujesz nad projektem hobbystycznym, czy też odpowiadasz za zarządzanie i koordynowanie współpracy między wieloma zespołami programistów, musisz określić optymalny zbiór funkcjonalności tworzonego oprogramowania. Pod uwagę trzeba wziąć wiele czynników: koszt opracowania funkcjonalności, ryzyko związane z brakiem możliwości pełnego zaimplementowania funkcjonalności, oczekiwaną wartość dla użytkownika końcowego, implikacje marketingowe i sprzedażowe, kwestie związane z późniejszą obsługą, skalowalność, ograniczenia prawne itd.

Ta faza ma znaczenie krytyczne, ponieważ później może chronić przed zmarnowaniem ogromnej ilości energii. Skutkiem błędnego planowania mogą być straty

zasobów liczone nawet w milionach. Natomiast staranne planowanie może zapewnić ogromny sukces w kolejnych latach. W trakcie fazy planowania należy wykorzystać nowo zdobytą umiejętność, czyli regułę 80/20 (zobacz rozdział 2.).

Prawidłowe przeprowadzenie fazy planowania jest również trudne ze względu na nieodłącznie związaną z nią złożoność. Kilka dodatkowych kwestii powoduje dalsze zwiększenie złożoności: poprawna ocena ryzyka z wyprzedzeniem, określenie strategicznego kierunku organizacji, odgadywanie odpowiedzi udzielanych przez klientów, ocena pozytywnego wpływu różnych kandydatów na funkcjonalności, określenie implikacji prawnych związanych z daną funkcjonalnością oprogramowania itd. Podsumowując: już sama złożoność tego wielowymiarowego problemu może być zabójcza.

Definiowanie

Faza definiowania polega na przekształceniu wyników fazy planowania na prawidłowo wyrażone wymagania oprogramowania. Innymi słowy formalizuje dane wyjściowe poprzedniej fazy w celu uzyskania akceptacji klienta lub informacji zwrotnych od klienta bądź użytkowników końcowych, którzy później będą korzystali z tego produktu.

Jeżeli poświęcisz sporo czasu na etapie planowania i określania wymagań projektu, ale nie potrafisz ich prawidłowo zakomunikować, później doprowadzi to do poważnych problemów i trudności. Błędnie nakreślone wymagania, które mają pomóc projektowi, są tak samo niebezpieczne jak prawidłowo sformułowane wymagania, które nie pomagają projektowi. Efektywna komunikacja i precyzyjna specyfikacja mają znaczenie krytyczne dla uniknięcia niejasności i nieporozumień. W przypadku komunikacji międzyludzkiej przekazywanie informacji jest niezwykle złożonym przedsięwzięciem ze względu na „przekleństwo wiedzy" oraz inne psychologiczne uprzedzenia, przeważające nad znaczeniem osobistych doświadczeń. Zachowaj ostrożność, gdy próbujesz podzielić się z kimś ideami (lub wymaganiami projektu) — dopadnie Cię złożoność.

Projektowanie

Celem fazy projektowania jest naszkicowanie architektury systemu, wybranie modułów i komponentów dostarczających zdefiniowaną funkcjonalność, a także opracowanie interfejsu użytkownika — jednocześnie trzeba pamiętać o wymaganiach zdefiniowanych w trakcie dwóch poprzednich faz. Standardem podczas fazy projektowania jest wyraźne określenie ostatecznego wyglądu i sposobu utworzenia produktu. To dotyczy wszystkich metod inżynierii oprogramowania. Podejście oparte na metodach zwinnych oznacza po prostu znacznie większe tempo iteracji przez te fazy.

Jednak diabeł tkwi w szczegółach! Świetny projektant systemu musi znać wady i zalety ogromnej liczby narzędzi, które mogą być używane podczas tworzenia systemu. Na przykład niektóre biblioteki są łatwe w użyciu dla programistów, ale charakteryzują się niską wydajnością działania. Samodzielne opracowanie biblioteki jest trudniejszym zadaniem dla programisty, choć skutkiem może być znacznie większa szybkość działania kodu, a tym samym poprawiona użyteczność

ostatecznego produktu oprogramowania. W trakcie fazy projektowania trzeba uwzględnić te aspekty i wybrać rozwiązanie zapewniające najlepszy stosunek korzyści do kosztów.

Budowanie

Faza budowania jest tą, w której wielu programistów chce spędzić najwięcej czasu. W jej trakcie odbywa się transformacja szkicu architekturalnego na produkt oprogramowania. Twoje idee zmieniają się w namacalne wyniki.

Jeżeli poprzednie fazy zostały poprawnie przeprowadzone, to wiele złożoności udało się już wyeliminować. W idealnej sytuacji będziesz wiedzieć, które funkcjonalności powinny być zaimplementowane, w jaki sposób będą działać oraz jakie narzędzia mają zostać użyte do ich implementacji. W trakcie fazy budowania zawsze pojawia się mnóstwo nowych problemów. Nieoczekiwane zdarzenia — takie jak błędy w bibliotekach zewnętrznych, kwestie związane z wydajnością działania, uszkodzenie danych i błędy popełniane przez człowieka — mogą spowolnić proces. Zbudowanie produktu oprogramowania to bardzo skomplikowane przedsięwzięcie. Nawet niewielki błąd może podkopać użyteczność całego produktu.

Testowanie

Gratulacje! Udało Ci się zaimplementować całą niezbędną funkcjonalność i wydaje się, że program działa. Jednak na tym nie koniec pracy. Nadal trzeba przetestować sposób działania tego produktu dla różnych danych wejściowych i wzorców użycia. Ta faza jest często najważniejsza ze wszystkich — jest tak ważna, że wiele osób stało się zwolennikami tzw. **programowania sterowanego testami** (ang. *test-driven development*, TDD). W tym podejściu nawet nie rozpoczyna się implementacji (fazy budowania) bez wcześniejszego przygotowania wszystkich testów. Wprawdzie możesz się nie zgadzać z takim punktem widzenia, ale mimo to dobrym pomysłem jest poświęcenie nieco czasu na przetestowanie produktu. W tym celu należy zdefiniować zestawy testów i sprawdzić, czy w trakcie ich wykonywania oprogramowanie dostarcza poprawne wyniki.

Na przykład załóżmy, że implementujesz autonomiczny samochód. Musisz zdefiniować *testy jednostkowe* sprawdzające, czy każda z funkcji (*jednostek*) w kodzie generuje żądane dane wyjściowe dla konkretnych danych wejściowych. Testy jednostkowe zwykle są tworzone dla funkcji działających błędnie po otrzymaniu określonych (ekstremalnych) danych wejściowych. Na przykład rozważ przedstawioną tutaj funkcję Pythona, której zadaniem jest obliczenie średniej wartości składowych koloru — RGB, czyli czerwony (ang. *red*), niebieski (ang. *blue*) i zielony (ang. *green*) — obrazu, prawdopodobnie w celu ustalenia, czy samochód porusza się w mieście czy na przykład w lesie:

```
def average_rgb(pixels):
    r = [x[0] for x in pixels]
    g = [x[1] for x in pixels]
    b = [x[2] for x in pixels]
```

```
    n = len(r)
    return (sum(r)/n, sum(g)/n, sum(b)/n)
```

Przedstawiona tutaj lista pikseli spowoduje wygenerowanie wartości średnich odpowiednio 96.0, 64.0 i 11.0:

```
print(average_rgb([(0, 0, 0),
                   (256, 128, 0),
                   (32, 64, 33)]))
```

Oto dane wyjściowe po wykonaniu kodu:

```
(96.0, 64.0, 11.0)
```

Wprawdzie ta funkcja wydaje się bardzo prosta, ale w praktyce może się w niej zdarzyć wiele złego. Co się stanie w przypadku uszkodzenia listy krotek i na przykład niektóre z nich będą zawierały tylko dwa elementy zamiast trzech? Co będzie w sytuacji, gdy jedna z wartości jest innego typu niż liczba całkowita? A co w przypadku, kiedy dane wyjściowe muszą mieć postać krotki liczb całkowitych, aby uniknąć błędu związanego z liczbami zmiennoprzecinkowymi, który jest nierozerwalnie związany z wszelkimi obliczeniami przeprowadzanymi na liczbach zmiennoprzecinkowych?

Test jednostkowy może sprawdzić te wszystkie warunki, co pozwala ustalić, czy w izolacji funkcja działa prawidłowo.

Oto dwa proste testy jednostkowe. Pierwszy z nich sprawdza, czy funkcja działa w przypadku dostarczenia jej zer jako danych wejściowych. Natomiast zadaniem drugiego jest sprawdzenie, czy wartością zwrotną funkcji jest krotka liczb całkowitych.

```
def unit_test_avg():
    print('Sprawdzenie średniej...')
    print(average_rgb([(0, 0, 0)]) == average_rgb([(0, 0, 0), (0, 0, 0)]))

def unit_test_type():
    print('Sprawdzenie typu...')
    for i in range(3):
        print(type(average_rgb([(1, 2, 3), (4, 5, 6)])[i]) == int)

unit_test_avg()
unit_test_type()
```

Wygenerowane przez nie wyniki pokazują, że sprawdzenie typu kończy się niepowodzeniem — funkcja nie zwraca poprawnego typu, którym powinna być krotka liczb całkowitych:

```
Sprawdzenie średniej...
True
Sprawdzenie typu...
False
False
False
```

W bardziej rzeczywistym projekcie będą zdefiniowane setki testów jednostkowych przeznaczonych do sprawdzania funkcji względem wszelkiego rodzaju danych wejściowych — i ustalenia, czy te funkcje generują oczekiwane dane wyjściowe. Tylko po upewnieniu się za pomocą testu jednostkowego o poprawnym działaniu funkcji można przejść do testowania funkcji wyższego poziomu w aplikacji.

Nawet jeśli wszystkie testy jednostkowe zostaną zaliczone, nie oznacza to zakończenia fazy testów projektu. Konieczne jest sprawdzenie poprawności interakcji testów jednostkowych, ponieważ tworzą one większą całość. Należy opracować rzeczywiste testy sprawdzające prowadzenie samochodu przez tysiące bądź dziesiątki tysięcy kilometrów i dopiero to pozwoli odkryć nieoczekiwane wzorce zachowania w dziwnych i nieprzewidywalnych sytuacjach. Jak samochód będzie zachowywał się na wąskich drogach pozbawionych znaków? Co w sytuacji, gdy jadący przez nim pojazd nagle się zatrzyma? Co się zdarzy, kiedy kierowca nagle przejmie kierowanie samochodem w ruchu ulicznym?

Istnieje ogromna liczba testów do rozważenia. Złożoność osiąga taki poziom, że wiele osób po prostu się poddaje. To, co wyglądało dobrze w teorii, a nawet na początkowym etapie implementacji, w praktyce często okazuje się porażką po zastosowaniu różnych poziomów testów oprogramowania, takich jak testy jednostkowe bądź testy symulujące rzeczywiste użycie produktu.

Wdrażanie

Oprogramowanie przeszło rygorystyczną fazę testowania. Najwyższa pora na jego wdrożenie. Faza wdrażania może przyjąć wiele postaci. Aplikacja może zostać opublikowana w sklepie z oprogramowaniem, pakiety mogą być dostępne w repozytoriach, a ważniejsze (lub mniej ważne) wydania mogą być publicznie udostępniane. W bardziej iteracyjnym i zwinnym podejściu do tworzenia oprogramowania faza wdrażania jest powtarzana wielokrotnie z wykorzystaniem tzw. **ciągłego wdrażania** (ang. *continuous deployment*). W zależności od konkretnego projektu ta faza wymaga uruchomienia produktu, opracowania kampanii marketingowych, komunikacji z pierwszymi użytkownikami produktu, usunięcia nowych błędów, które na pewno pojawią się po udostępnieniu aplikacji użytkownikom, koordynacji wdrożenia oprogramowania w różnych systemach operacyjnych, zapewnienia pomocy technicznej i możliwości rozwiązywania różnego rodzaju problemów, a także obsługi bazy kodu w celu jego usprawniania na przestrzeni lat. Dość szybko ta faza staje się trudna, biorąc pod uwagę złożoność i współzależności między różnymi rozwiązaniami projektowymi, które zostały wybrane i zaimplementowane we wcześniejszych fazach. W kolejnych rozdziałach zasugeruję taktykę pomocną w przezwyciężeniu tego bałaganu.

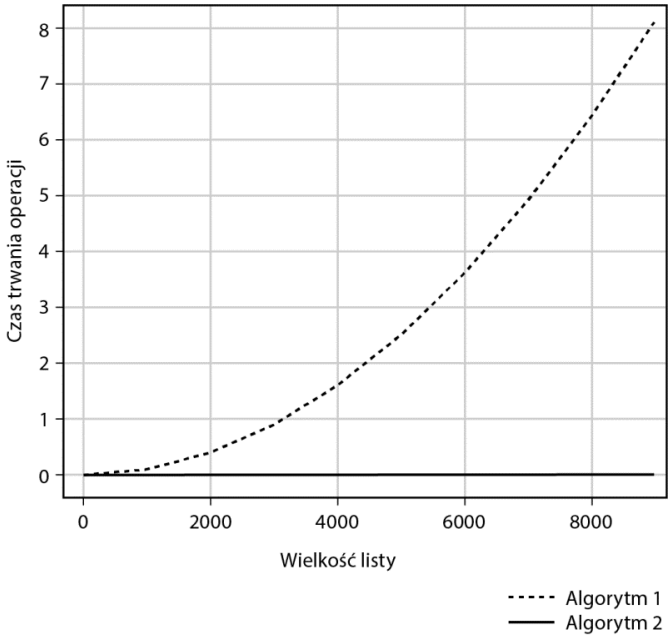
Złożoność w teorii oprogramowania i algorytmów

W oprogramowaniu może być tyle złożoności, ile w procesie związanym z tworzeniem tego oprogramowania. W inżynierii oprogramowania wiele wskaźników zostało przeznaczonych do pomiaru złożoności w oficjalny sposób.

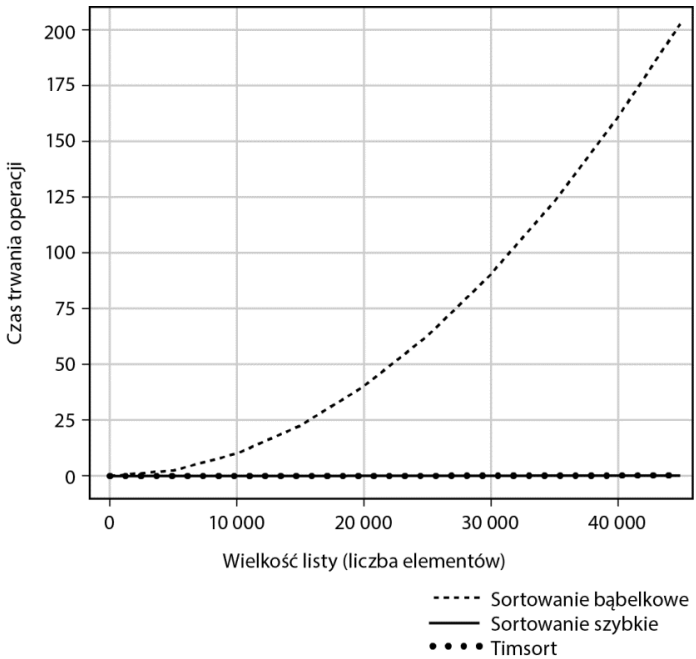
Zaczniemy od zapoznania się ze **złożonością algorytmiczną**, związaną z wymaganiami zasobów przez różne algorytmy. Używając złożoności algorytmicznej, można porównywać różne algorytmy przeznaczone do rozwiązywania tego samego problemu. Na przykład załóżmy, że została zaimplementowana gra oferująca tabelę najlepszych wyników. W takim przypadku gracze z najwyższą liczbą punktów powinni pojawiać się na górze listy, a ci z gorszym wynikiem w dolnej części listy. Innymi słowy konieczne jest *sortowanie* listy. Opracowano tysiące algorytmów przeznaczonych do sortowania list, a sama ta operacja jest pod względem obliczeniowym w przypadku miliona graczy znacznie bardziej wymagająca niż na przykład dla stu graczy. Niektóre algorytmy skalują się lepiej wraz ze wzrostem wielkości listy danych wejściowych, podczas gdy inne skalują się gorzej. Kiedy aplikacja obsługuje kilkuset użytkowników, wybór konkretnego algorytmu nie ma znaczenia. Natomiast wraz ze wzrostem wielkości bazy użytkowników złożoność związana z obsługą listy w środowisku uruchomieniowym rośnie w postępie geometrycznym. Wkrótce użytkownicy będą musieli czekać coraz dłużej na posortowanie listy, zaczną narzekać i konieczne będzie zastosowanie znacznie lepszych algorytmów.

Na rysunku 1.2 pokazałem złożoność algorytmiczną dwóch algorytmów schematycznych. Oś X to wielkość listy przeznaczonej do sortowania. Z kolei oś Y to czas działania algorytmu wyrażony w jednostkach czasu. Algorytm 1 jest znacznie wolniejszy niż algorytm 2. W rzeczywistości niewydolność algorytmu 1 staje się coraz bardziej widoczna wraz ze wzrostem liczby elementów listy do sortowania. W przypadku użycia algorytmu 1 wraz z każdym kolejnym graczem aplikacja będzie coraz wolniejsza.

Zobaczmy, jak to wygląda w przypadku rzeczywistych algorytmów sortowania w Pythonie. Na rysunku 1.3 pokazałem porównanie wydajności działania trzech popularnych algorytmów sortowania: bąbelkowego, szybkiego i Timsort. Sortowanie bąbelkowe ma największą złożoność algorytmiczną, natomiast dwa pozostałe rodzaje sortowania charakteryzują się podobną złożonością algorytmiczną. Mimo to algorytm Timsort jest znacznie szybszy — i dlatego w Pythonie jest domyślnym algorytmem sortowania. Czas potrzebny na przeprowadzenie operacji sortowania bąbelkowego gwałtownie rośnie razem z wielkością listy.

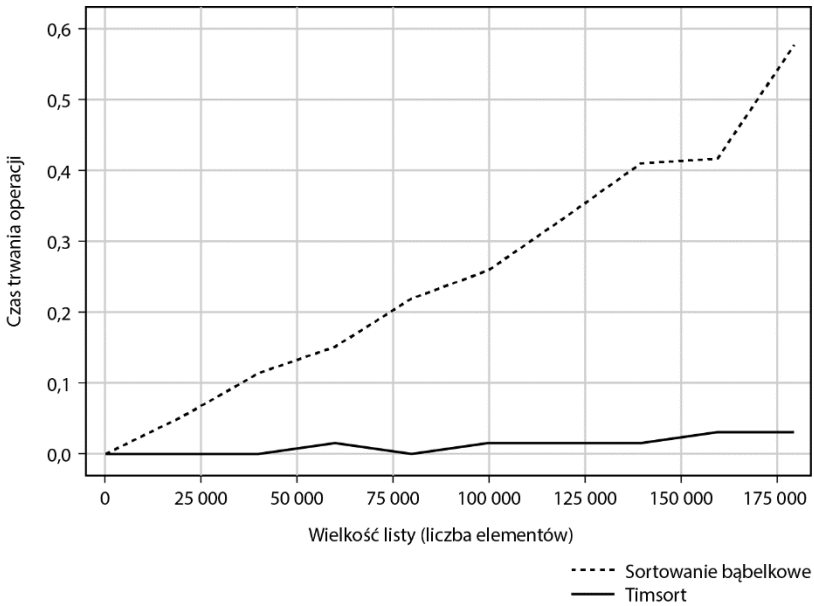


Rysunek 1.2. Złożoność algorytmiczna dwóch różnych algorytmów sortowania



Rysunek 1.3. Złożoność algorytmiczna sortowania bąbelkowego, szybkiego i Timsort

Z kolei na rysunku 1.4 pokazałem wynik powtórzenia tego eksperymentu, ale jedynie dla algorytmów sortowania bąbelkowego i Timsort. Także tym razem mamy znaczną różnicę w złożoności algorytmicznej: sortowanie Timsort skaluje się lepiej i działa szybciej w przypadku zwiększającej się listy. Teraz już wiesz, dlaczego domyślny algorytm sortowania w Pythonie nie zmienił się od długiego czasu.



Rysunek 1.4. Złożoność algorytmiczna sortowania bąbelkowego i Timsort

Jeżeli chcesz samodzielnie odtworzyć ten eksperyment, skorzystaj z kodu zamieszczonego na listingu 1.1. Zalecam użycie mniejszej wartości `n`, ponieważ na moim komputerze wykonanie kodu w przedstawionej postaci trwało dość długo.

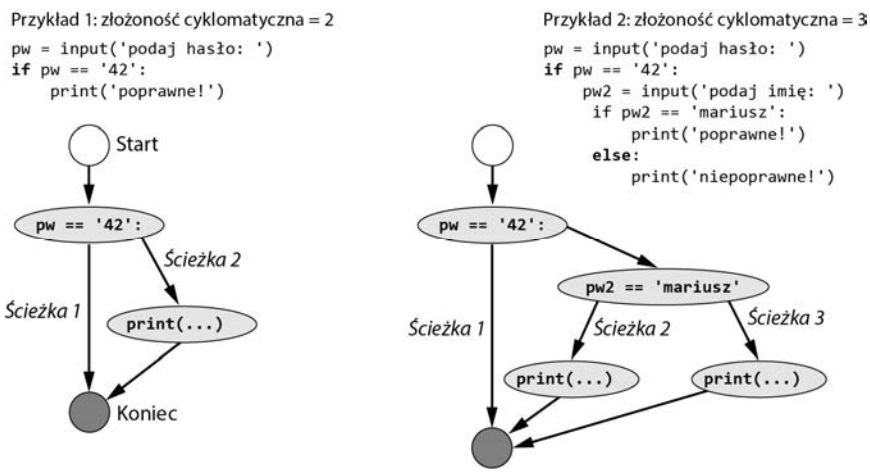
Listing 1.1. Pomiar czasu działania trzech popularnych algorytmów sortowania

```
import matplotlib.pyplot as plt
import math
import time
import random


def bubblesort(l):
    # src: https://blog.finxter.com/daily-python-puzzle-bubble-sort/
    lst = l[:]  # Praca z kopią, nie należy modyfikować oryginału
    for passesLeft in range(len(lst)-1, 0, -1):
        for i in range(passesLeft):
            if lst[i] > lst[i + 1]:
                lst[i], lst[i + 1] = lst[i + 1], lst[i]
    return lst
```

```
def qsort(lst):
    # Wyjaśnienie: https://blog.finxter.com/python-one-line-quicksort/
    q = lambda lst: q([x for x in lst[1:] if x <= lst[0]])
                    + [lst[0]]
                    + q([x for x in lst if x > lst[0]]) if lst else []
     return q(lst)

def timsort(l):
    # Metoda sorted() wewnętrznie używa algorytmu Timsort
    return sorted(l)

def create_random_list(n):
    return random.sample(range(n), n)


n = 50000
xs = list(range(1,n,n//10))
y_bubble = []
y_qsort = []
y_tim = []

for x in xs:
    # Utworzenie listy
    lst = create_random_list(x)

    # Pomiar czasu trwania operacji sortowania bąbelkowego
    start = time.time()
    bubblesort(lst)
    y_bubble.append(time.time()-start)

    # Pomiar czasu trwania operacji sortowania szybkiego
    start = time.time()
    qsort(lst)
    y_qsort.append(time.time()-start)

    # Pomiar czasu trwania operacji sortowania Timsort
    start = time.time()
    timsort(lst)
    y_tim.append(time.time()-start)

plt.plot(xs, y_bubble, '-x', label='Sortowanie bąbelkowe')
plt.plot(xs, y_qsort, '-o', label='Sortowanie szybkie')
plt.plot(xs, y_tim, '--.', label='Sortowanie Timsort')

plt.grid()
plt.xlabel('Wielkość listy (liczba elementów)')
plt.ylabel('Czas trwania operacji (s)')
plt.legend()
plt.savefig('alg_complexity_new.pdf')
plt.savefig('alg_complexity_new.jpg')
plt.show()
```

Złożoność algorytmiczna jest przedmiotem dokładnych badań. Według mnie usprawnione algorytmy opracowane na podstawie wyników tych badań są najcenniejszym zasobem technologicznym dla ludzkości, ponieważ pozwalają na rozwiązywanie tych samych problemów z wykorzystaniem mniejszej ilości zasobów. Naprawdę stoimy na ramionach olbrzymów.

Poza złożonością algorytmiczną złożoność kodu źródłowego można również mierzyć za pomocą tzw. **złożoności cyklomatycznej**. Ten opracowany w 1976 roku przez Thomasa McCabe'a wskaźnik opisuje liczbę *liniowo niezależnych ścieżek* w kodzie, inaczej liczbę ścieżek, w których przynajmniej jedna krawędź nie znajduje się w innej ścieżce. Na przykład kod zawierający konstrukcję `if` będzie miał dwie niezależne ścieżki działania, a tym samym charakteryzuje się większą złożonością cyklomatyczną niż kod pozbawiony wszelkich rozgałęzień typu polecenie `if`. Na rysunku 1.5 pokazałem złożoność cyklomatyczną dwóch programów w Pythonie przetwarzających dane wejściowe użytkownika i odpowiednio na nie reagujących. Pierwszy program zawiera tylko jedną konstrukcję warunkową, którą można uznać za rozwidlenie na drodze. Można wybrać jedno odgałęzienie, ale nie oba. Dlatego też złożoność cyklomatyczna wynosi 2, ponieważ mamy dwie liniowo niezależne ścieżki wykonywania kodu. Z kolei drugi program zawiera dwie konstrukcje warunkowe prowadzące do istnienia trzech liniowo niezależnych ścieżek, więc złożoność cyklomatyczna wynosi 3. Każda dodatkowa konstrukcja `if` zwiększa złożoność cyklomatyczną.



Rysunek 1.5. Złożoność cyklomatyczna dwóch programów Pythona

Złożoność cyklomatyczna to solidny wskaźnik przeznaczony do sprawdzenia trudnej do zmierzenia **złożoności kognitywnej**, która określa, jak trudno jest zrozumieć daną bazę kodu. Jednak złożoność cyklomatyczna ignoruje złożoność kognitywną wynikającą na przykład z wielu zagnieżdżonych pętli `for` w porównaniu z płaską pętlą `for`. Dlatego inne wskaźniki, takie jak złożoność `NPath`, poprawiają złożoność cyklomatyczną. Podsumowując, złożoność kodu źródłowego nie tylko

jest ważnym przedmiotem teorii algorytmicznej, ale również ma duże znaczenie podczas implementowania kodu — także podczas tworzenia łatwego do zrozumienia, czytelnego i niezawodnego kodu. Zarówno teoria algorytmiczna, jak i złożoność programistyczna były przez dekady przedmiotami szczegółowych badań. Podstawowym celem tych wysiłków jest *zmniejszenie złożoności obliczeniowej i nieobliczeniowej* w celu złagodzenia negatywnego wpływu na produktywność i poziom wykorzystania zasobów ludzkich i sprzętowych.

Złożoność w nauce

Fakty nie istnieją w próżni, ale są ze sobą powiązane. Rozważ dwa następujące fakty:

Walt Disney urodził się w 1901 roku.

Louis Armstrong urodził się w 1901 roku.

Jeżeli te fakty zostaną przekazane programowi, będzie mógł on udzielać odpowiedzi na pytania typu *„W którym roku urodził się Walt Disney?"* i *„Kto urodził się w 1901 roku?"*. Aby udzielić odpowiedzi na to drugie pytanie, program musi ustalić wzajemne zależności między różnymi faktami. Informacje mogą być modelowane na przykład w takiej postaci:

```
(Walt Disney, rok urodzenia, 1901)
(Louis Armstrong, rok urodzenia, 1901)
```

Aby pobrać wszystkie osoby urodzone w 1901 roku, można użyć zapytania typu `(*, rok urodzenia, 1901)` albo w jakikolwiek inny sposób powiązać fakty i je grupować.

W 2012 roku firma Google udostępniła nową funkcjonalność wyszukiwarki internetowej — wyświetlanie ramek z informacjami na stronie wyników wyszukiwania. Te oparte na faktach ramki są wypełniane za pomocą struktury danych nazywanej **grafem wiedzy**, czyli ogromnej bazy danych miliardów powiązanych ze sobą faktów przedstawiających informacje w strukturze przypominającej sieć. Zamiast przechowywać obiektywne i niezależne fakty baza ta zawiera dane o wzajemnych powiązaniach między różnymi faktami i innymi skrawkami informacji. Silnik wyszukiwarki Google wykorzystuje ten graf wiedzy do wzbogacenia wyników wyszukiwania wiedzą wysokiego poziomu i automatycznego tworzenia odpowiedzi.

Przykład możesz zobaczyć na rysunku 1.6. Jeden z węzłów grafu wiedzy może dotyczyć sławnego informatyka Alana Turinga. W tym grafie wiedzy koncepcja `Alan Turing` jest powiązana z różnymi fragmentami informacji, np. rokiem jego urodzenia (`1912`), obszarami jego zainteresowań (`informatyka`, `filozofia`, `lingwistyka`), promotorem jego pracy doktorskiej (`Alonzo Church`). Każdy z tych fragmentów informacji jest również połączony z innymi faktami (obszarem zainteresowań `Alonzo Church` również była `informatyka`), co prowadzi do powstania ogromnej sieci powiązanych ze sobą faktów. Tę sieć można wykorzystać do pobierania nowych informacji i automatycznego udzielania odpowiedzi na zapytania. Na przykład

zapytanie dotyczące `zainteresowań promotora Turinga` powinno wygenerować odpowiedź `informatyka`. Wprawdzie to może wydawać się banalne i oczywiste, ale możliwość generowania nowych danych takich jak te doprowadziła do przełomu w zakresie pobierania informacji i efektywności pracy silnika wyszukiwania. Prawdopodobnie zgodzisz się z tym, że nauka przez skojarzenia jest znacznie efektywniejsza od zapamiętywania niepowiązanych ze sobą faktów.

Oto wybrane dane użyte w omawianym grafie wiedzy:

```
("Alan Turing", "promotor pracy doktorskiej", "Alonzo Church")
("Alan Turing", "obszar zainteresowań", "filozofia")
("Alan Turing", "obszar zainteresowań", "lingwistyka")
```

Rysunek 1.6. Przykład prostego grafu wiedzy

Każdy obszar zainteresowań skupia się jedynie na niewielkim skrawku grafu, a każdy skrawek składa się z niezliczonych powiązanych ze sobą faktów. Dany obszar można zrozumieć jedynie biorąc pod uwagę powiązane ze sobą fakty. Aby naprawdę dobrze poznać Alana Turinga, trzeba zapoznać się z jego przekonaniami, filozofią i cechami promotora jego pracy doktorskiej. Aby w pełni zrozumieć Churcha, trzeba poznać jego powiązanie z Turingiem. Oczywiście w grafie wiedzy istnieje zbyt wiele powiązanych ze sobą zależności i faktów, więc trudno oczekiwać, że uda się zrozumieć wszystko. Złożoność tych powiązań nakłada największe ograniczenia na Twoje ambicje związane ze zdobywaniem wiedzy. Nauka i złożoność to dwie strony medalu: złożoność znajduje się na granicy wiedzy, którą już masz. Jeżeli chcesz nauczyć się więcej, najpierw musisz nauczyć się kontrolować złożoność.

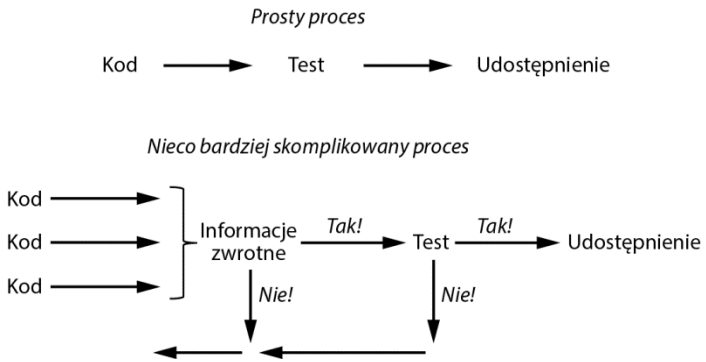
To dość abstrakcyjne, więc posłużę się przykładem. Załóżmy, że chcesz utworzyć bota, którego zadaniem jest kupno i sprzedaż aktywów na podstawie zbioru zaawansowanych reguł. Przed rozpoczęciem pracy nad projektem można spróbować zdobyć użyteczną wiedzę: podstawy programowania, systemy rozproszone, bazy danych, interfejs programowania aplikacji (ang. *application programming interface*, API), usługi sieciowe, uczenie maszynowe, nauka i powiązana z nią matematyka. Można również spróbować poznać praktyczne narzędzia, takie jak Python,

NumPy, scikit-larn, ccxt, TensorFlow, Flask itd. Ponadto można spróbować poznać strategie dotyczące handlu i filozofii rynku papierów wartościowych. W celu rozwiązania problemu wiele osób zastosuje właśnie takie podejście i nigdy nie poczuje gotowości do rozpoczęcia pracy nad projektem. Problem polega na tym, że im więcej się uczysz, tym większe masz poczucie, że mniej wiesz. Nigdy nie poczujesz się wystarczająco pewnie w tych wszystkich dziedzinach, aby czuć się w pełni przygotowanym do pracy. Czując przytłoczenie złożonością całego przedsięwzięcia, łatwiej zdecydować się na jego porzucenie. Tym samym złożoność zbiera kolejną ofiarę: Ciebie.

Na szczęście dzięki tej książce zdobędziesz umiejętności pozwalające stawić czoło złożoności: skupienie, uproszczenie, skalowanie w dół, redukcja i minimalizm. Postaram się nauczyć Cię tych umiejętności.

Złożoność w procesach

Proces to seria działań podejmowanych w celu osiągnięcia zdefiniowanego wyniku. Złożoność procesu jest ustalana na podstawie liczby działań, uczestników lub gałęzi. Ogólnie rzecz biorąc, im więcej działań (i uczestników), tym bardziej skomplikowany staje się proces (zobacz rysunek 1.7).



Rysunek 1.7. Dwa przykładowe procesy: programowanie jednoosobowe i programowanie zespołowe

Wiele firm tworzących oprogramowanie stosuje modele procesów dla różnych aspektów działalności w celu uproszczenia procesów. Spójrz na kilka przykładów:

Tworzenie oprogramowania może odbywać się z użyciem metod zwinnych lub scrum.

Nawiązywanie relacji z klientem może wykorzystywać system CRM (ang. *customer relationship management*) i skrypty sprzedaży.

Tworzenie modelu biznesowego i nowego produktu może wykorzystywać płótno modelu biznesowego.

Gdy organizacja zgromadzi zbyt wiele procesów, złożoność zaczyna blokować system. Na przykład zanim na rynku pojawił się Uber, proces podróżowania z miejsca A do miejsca B często wiązał się z wieloma krokami: znalezienie numeru telefonu do korporacji taksówkarskich, porównanie cen, przygotowanie różnych opcji płatności, zaplanowanie podróży. W przypadku wielu klientów Uber usprawnił proces podróży z miejsca A do miejsca B, integrując cały proces planowania do postaci łatwej w obsłudze aplikacji mobilnej. To znaczne uproszczenie wprowadzone przez Ubera oznacza, że podróżowanie stało się znacznie wygodniejsze dla klientów. Ponadto wiąże się ze skróceniem czasu planowania i niższymi kosztami w porównaniu z korzystaniem z usług typowych korporacji taksówkarskich.

W przesadnie skomplikowanych organizacjach innowacja znajduje mniej możliwości dla zmian, ponieważ trudno jest przebić się przez tę złożoność. Zasoby są marnowane z powodu przeprowadzania zbędnych działań w procesach. Próbując uzdrowić tę sytuację, menedżerowie inwestują energię w opracowanie nowych procesów i działań, złe cykle zaś rozpoczynają niszczenie biznesu bądź organizacji.

Złożoność jest wrogiem efektywności. Rozwiązaniem jest tutaj minimalizm: w celu zapewnienia efektywności procesom trzeba zdecydowanie pozbyć się niepotrzebnych kroków i działań. Istnieje znikome prawdopodobieństwo, że proces będzie można uznać za *zbyt uproszczony*.

Złożoność w życiu codziennym, czyli kara tysiąca cięć

Celem tej książki jest zwiększenie Twojej produktywności podczas programowania. Postęp może być zakłócony przez codzienne zwyczaje i rutynowe zadania. Musisz zmierzyć się z odrywającą Cię od zadań codziennością i nieustanną rywalizacją o Twój cenny czas. Profesor informatyki Cal Newport porusza ten temat w swojej doskonałej książce *Deep Work: Rules for Focused Success in a Distracted World* (Grand Central Publishing, 2016). Przekonuje w niej, że istnieje zarówno *stale zwiększające się* zapotrzebowanie na pracę wymagającą głębokiego zastanowienia się — przykładem mogą być tutaj programowanie, praca badawcza, leczenie i pisanie — jak i *coraz mniej* czasu na wykonanie tej pracy ze względu na szybkie rozpowszechnianie się urządzeń służących do komunikacji i systemów rozrywkowych. Jeżeli mamy coraz większy popyt (tutaj na pracę) i coraz mniejszą podaż (tutaj czasu), zgodnie z teorią ekonomii nastąpi wzrost cen. Jeżeli masz możliwość pracy w stanie głębokiej koncentracji, Twoja wartość ekonomiczna wzrośnie. Nigdy nie było lepszego czasu dla programistów, którzy chcą pracować w takim właśnie stanie.

A teraz łyżka dziegciu w beczce miodu: praktycznie niemożliwa jest praca w stanie głębokiej koncentracji, o ile nie będzie bezwzględnie wymuszony system nadawania priorytetów zadaniom. Otaczający nas świat non stop próbuje nas rozpraszać. Twoi koledzy pojawiają się w biurze. Twój telefon co 20 minut domaga się uwagi.

Przez cały dzień w Twojej skrzynce odbiorczej poczty elektronicznej pojawiają się nowe wiadomości — to wszystko wymaga poświęcenia chwili Twojego cennego czasu.

Efektem pracy w stanie głębokiej koncentracji jest późniejsza satysfakcja: po tygodniach pracy nad programem komputerowym można odczuwać pełną satysfakcję, widząc jego poprawne działanie. Jednak w większości przypadków *oczekuje się* natychmiastowej satysfakcji. Twoja podświadomość często szuka pretekstów do porzucenia wysiłku związanego z pracą w stanie głębokiej koncentracji. Nawet drobne sukcesy bardzo wzmagają produkcję endorfin: sprawdzanie poczty elektronicznej, niezobowiązujący czat, zajrzenie do Netflixa. Wizja późniejszej satysfakcji staje się coraz mniej atrakcyjna w porównaniu z zadowoleniem, radością i pełnym życia światem natychmiastowej satysfakcji.

Twoje wysiłki zmierzające do zachowania skupienia i produktywności zostają skazane na śmierć na skutek kary tysiąca cięć. Oczywiście możesz raz wyłączyć smartfona i oprzeć się pokusie sprawdzenia profilu w serwisie społecznościowym lub obejrzenia ulubionego programu, ale czy jesteś w stanie robić to dzień w dzień? Tutaj odpowiedź również tkwi w zastosowaniu radykalnego minimalizmu względem źródła problemu — *odinstalowanie* aplikacji serwisów społecznościowych zamiast podejmowania prób zarządzania czasem ich używania, *zmniejszenie* liczby realizowanych projektów i zadań zamiast podejmowania próby zrobienia więcej przez wydłużenie czasu pracy, a także *dokładniejsze* poznanie języka programowania zamiast poświęcania wiele czasu na przeskakiwanie między różnymi językami programowania.

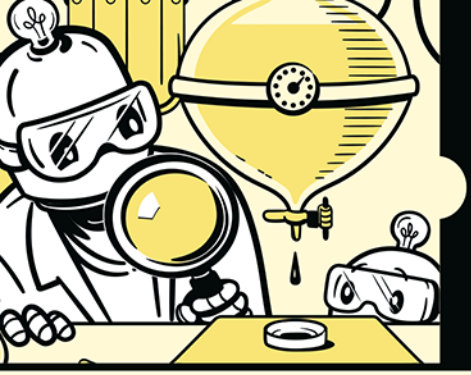
Podsumowanie

W tym momencie prawdopodobnie masz motywację do pokonania złożoności. Jeżeli chcesz nieco dokładniej poznać temat złożoności i sposobów na jej pokonanie, zachęcam do sięgnięcia po książkę Cala Newporta zatytułowaną *Deep Work*.

Złożoność szkodzi produktywności i zmniejsza poziom skupienia. Jeżeli wcześnie nie przejmiesz kontroli nad złożonością, ona bardzo szybko wykorzysta większość Twojego najcenniejszego zasobu, jakim jest czas. U schyłku życia nie będziesz go oceniać przez pryzmat liczby przetworzonych wiadomości e-mail, godzin poświęconych na gry komputerowe ani liczbę rozwiązanych sudoku. Jeżeli nauczysz się zarządzać złożonością przez zachowanie prostoty, będziesz w stanie z nią walczyć i zapewnisz sobie potężną przewagę.

W rozdziale 2. poznasz zasadę 80/20: skupienie na kilku najważniejszych kwestiach i ignorowanie wielu błahych.


PROGRAM
PARTNERSKI
GRUPY HELION
1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ
Zmień swoją stronę WWW w działający bankomat!
Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl
GRUPA
Helion



OTO DZIEWIĘĆ ZASAD TWORZENIA IDEALNEGO KODU — POZNAJ I STOSUJ!

Wielu adeptów kodowania ulega złudnemu przekonaniu, że opanowanie jakiegoś języka programowania wystarczy, aby być programistą. Nader często w pośpiechu piszą nieuporządkowany kod, który zawiera mnóstwo powtórzeń i jest kompletnie nieczytelny. Tymczasem prawdziwi mistrzowie programowania pracują inaczej: w pełni skupiają się na jednym aspekcie swojej pracy, efektywnie wykorzystują czas i tworzą kod o niewielkiej objętości, a przy tym czytelny, elegancki i łatwy w utrzymaniu.

Dzięki tej książce dowiesz się, w jaki sposób pisać czysty i w pełni funkcjonalny kod. Nauczysz się przy mniejszym nakładzie pracy uzyskiwać lepsze rezultaty, co pozwoli Ci przeznaczyć zaoszczędzony czas na dopracowanie najistotniejszych elementów programu. Przekonasz się, że przemyślany minimalizm świetnie wspiera produktywność i znakomicie sprawdza się w praktyce. Dowiesz się, jak wykrywać źródła zbędnej złożoności i je eliminować, wyrobisz w sobie nawyk koncentrowania się na najważniejszych aspektach programu, a także docenisz zalety niezwykle prostych interfejsów użytkownika. Zrozumiesz, że optymalizacja nie musi oznaczać ograniczania użycia cykli procesora za wszelką cenę, a minimalizm i prostota świetnie się sprawdzają przy tworzeniu strategii projektu i wrażeń użytkownika.

Dowiedz się, jak:

- skoncentrować się na najważniejszych 20% kodu
- unikać samotnej pracy
- eliminować zbędną złożoność
- ustrzec się przedwczesnej optymalizacji
- osiągać produktywny stan przepływu
- w jednym czasie skupiać się na jednym zadaniu
- projektować proste i funkcjonalne interfejsy użytkownika

Dr Christian Mayer jest pasjonatem Pythona, założył i prowadzi popularną stronę poświęconą temu językowi programowania (https://blog.finxter.com/). Słynie z wybitnych umiejętności przekazywania wiedzy, co doceniły już tysiące adeptów Pythona. Autor książek z serii „Coffee Break Python".

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej! ▶

ISBN 978-83-8322-064-2

Cena: 59,00 zł

no starch press