Street Coder - Zasady do złamania i jak je złamać

tłumaczenie czatem GuPeTe

https://www.manning.com/books/street-coder

1 Na ulicy

Ten rozdział obejmuje

- Rzeczywistość ulic
- Kim jest programista uliczny?
- Problemy współczesnego rozwoju oprogramowania
- Jak rozwiązać swoje problemy za pomocą wiedzy ulicznej

Mam szczęście. Napisałem swoją pierwszą pętlę w latach 80. Wystarczyło mi tylko włączyć komputer, co zajęło mniej niż sekundę, napisać 2 linie kodu, wpisać RUN, i voila! Ekran nagle zapełnił się moim imieniem. Byłem od razu pod wrażeniem możliwości. Jeśli mogłem to zrobić za pomocą 2 linii, wyobraź sobie, co mogłem zrobić z 6 liniami, czy nawet z 20 liniami! Moje dziewięcioletnie mózgowe było zalane tyle dopaminy, że od razu uzależniłem się od programowania.

Dziś rozwój oprogramowania jest ogromnie bardziej skomplikowany. Nie ma już tej prostoty lat 80., gdy interakcje użytkownika ograniczały się do "naciśnij dowolny klawisz, aby kontynuować", chociaż czasami użytkownicy mieli problem ze znalezieniem "dowolnego" klawisza na swojej klawiaturze. Nie było okien, myszy, stron internetowych, elementów interfejsu użytkownika, bibliotek, frameworków, maszyn wirtualnych, urządzeń mobilnych. Miałeś tylko zestaw poleceń i statyczną konfigurację sprzętu.

Istnieje powód dla każdego poziomu abstrakcji, który teraz posiadamy, i nie jest tak, że jesteśmy masochistami, z wyjątkiem programistów Haskell1. Te abstrakcje są wprowadzane, ponieważ są jedynym sposobem na nadążanie za obecnymi standardami oprogramowania. Programowanie już nie polega na wypełnianiu ekranu swoim imieniem. Twoje imię musi być we właściwej czcionce i musi znajdować się w oknie, które możesz przeciągać i zmieniać jego rozmiar. Twój program musi wyglądać dobrze. Powinien obsługiwać kopiowanie i wklejanie. Musi obsługiwać różne nazwy dla konfigurowalności. Być może powinien przechowywać nazwy w bazie danych, nawet w chmurze. Wypełnianie ekranu swoim imieniem już nie jest takie zabawne.

Na szczęście mamy zasoby, które pomagają nam poradzić sobie z tą złożonością: uniwersytety, hackathony, obozy programistyczne, kursy online i kaczki gumowe.

PORADA

Kaczka gumowa debugging to ezoteryczna metoda znajdowania rozwiązań problemów programistycznych. Polega na rozmawianiu z żółtym plastikowym ptakiem. Opowiem ci więcej o tym w rozdziale o debugowaniu.

1.1 Co ma znaczenie na ulicach

Świat zawodowego rozwoju oprogramowania jest dość tajemniczy. Niektórzy klienci przysięgają, że zapłacą Ci w ciągu kilku dni za każdym razem, gdy do nich dzwonisz przez miesiące. Niektórzy pracodawcy w ogóle nie płacą Ci wynagrodzenia, ale upierają się, że zapłacą Ci "kiedy już zarobią pieniądze". Chaotyczna przypadkowość wszechświata decyduje, kto dostaje biurowe okno. Niektóre błędy znikają, gdy używasz debugera. Niektóre zespoły w ogóle nie korzystają z kontroli wersji. Tak, to przerażające. Ale musisz zmierzyć się z rzeczywistością.

Jedno jest jasne na ulicach: najważniejsza jest wydajność. Nikogo nie obchodzi twój elegancki projekt, twoja wiedza o algorytmach czy wysokiej jakości kod. Wszystko, co ich obchodzi, to ile możesz dostarczyć w określonym czasie. Wbrew intuicji, dobry projekt, dobre wykorzystanie algorytmów i wysokiej jakości kod mogą znacząco wpływać na twoją wydajność, a tego wielu programistów nie rozumie. Takie sprawy są zazwyczaj postrzegane jako przeszkody, tarcia między programistą a terminem. Taki sposób myślenia może uczynić z ciebie zombi z kulą u nogi.

W rzeczywistości niektórym ludziom zależy na jakości twojego kodu: twoim kolegom. Nie chcą pilnować twojego kodu. Chcą, żeby twój kod działał, był łatwo zrozumiały i możliwy do utrzymania. To jest coś, co im zawdzięczasz, ponieważ kiedy raz zatwierdzisz swój kod w repozytorium, staje się on kodem każdego. W zespole wydajność zespołu jest ważniejsza niż wydajność każdego z jego członków. Jeśli piszesz zły kod, spowalniasz swoich kolegów. Brak jakości twojego kodu szkodzi zespołowi, a spowolniony zespół szkodzi produktowi, a niezrealizowany produkt szkodzi twojej karierze.

Najłatwiejszą rzeczą, którą możesz napisać od zera, jest pomysł, a następną najłatwiejszą rzeczą jest projekt. Dlatego dobre projektowanie ma znaczenie. Dobre projektowanie to nie coś, co dobrze wygląda na papierze. Możesz mieć projekt w głowie, który działa. Natkniesz się na ludzi, którzy nie wierzą w projektowanie i improwizują kod. Ci ludzie nie cenią swojego czasu.

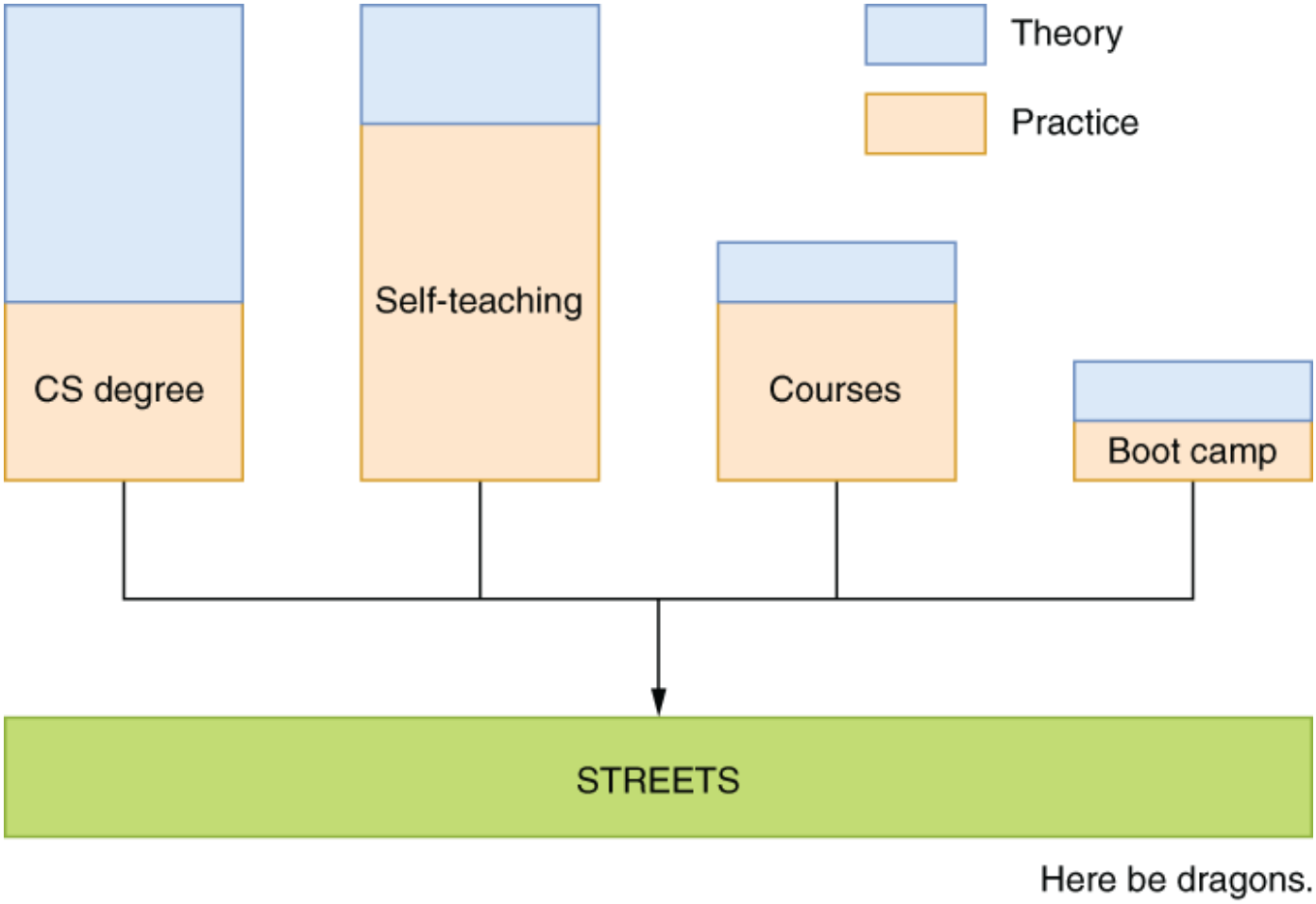
Podobnie dobry wzorzec projektowy czy algorytm może zwiększyć twoją wydajność. Jeśli nie pomaga w twojej wydajności, nie jest użyteczny. Ponieważ prawie wszystko można przyporządkować wartość pieniężną, wszystko, co robisz, można zmierzyć pod względem wydajności.

Możesz mieć wysoką wydajność przy złym kodzie, ale tylko w pierwszej iteracji. W chwili, gdy klient prosi o zmianę, zostajesz z utrzymaniem okropnego kodu. Przez całą tę książkę będę mówił o przypadkach, w których możesz zdać sobie sprawę, że wpadasz w pułapkę i wydostać się z niej, zanim stracisz zmysły.

1.2 Kim jest programista uliczny?

Microsoft rozważa dwie różne kategorie kandydatów podczas rekrutacji: absolwentów wydziałów informatyki i ekspertów branżowych, którzy posiadają znaczne doświadczenie w rozwoju oprogramowania.

Nie ważne, czy jesteś samoukiem czy ktoś, kto studiował informatykę, na początku swojej kariery brakuje ci wspólnego elementu: wiedzy ulicznej, czyli umiejętności wiedzenia, co jest najważniejsze. Samouk programista ma za sobą wiele prób i błędów, ale może brakować mu wiedzy na temat formalnej teorii i tego, jak można ją zastosować w codziennym programowaniu. Absolwent uczelni z kolei wie wiele o teorii, ale brakuje mu praktyczności i czasami podejścia krytycznego do tego, co się nauczył. Patrz rysunek 1.1.



Korpus, który zdobywasz w szkole, nie ma z nim związanej priorytetowości. Uczysz się według ścieżki edukacyjnej, a nie według ważności. Nie masz pojęcia, jakie pewne przedmioty mogą być użyteczne na ulicach, gdzie konkurencja jest bezlitosna. Terminy są nierealistyczne. Kawa jest zimna. Najlepszy framework na świecie ma ten jeden błąd, który sprawia, że tydzień twojej pracy idzie na marne. Twoja doskonale zaprojektowana abstrakcja rozpada się pod naciskiem klienta, który nieustannie zmienia swoje wymagania. Udaje ci się szybko zrefaktoryzować swój kod za pomocą kopiuj-wklej, ale teraz musisz edytować 15 różnych miejsc tylko po to, aby zmienić jedną wartość konfiguracji.

Przez lata zdobywasz nowe umiejętności, aby radzić sobie z niejasnością i złożonością. Samoucy programiści uczą się pewnych algorytmów, które im pomagają, a absolwenci uczelni w końcu rozumieją, że najlepsza teoria nie zawsze jest najbardziej praktyczna.

Programista uliczny to każdy z doświadczeniem w rozwoju oprogramowania w przemyśle, którego przekonania i teorie zostały ukształtowane przez rzeczywistość nierealistycznego szefa, który chciał, żeby tydzień pracy został zrobiony rano. Nauczyli się oni robić kopie zapasowe wszystkiego na wielu nośnikach po tym, jak stracili tysiące linii kodu i musieli to wszystko pisać od nowa. Widzieli migoczące światło C-beam w pomieszczeniu serwerowym spowodowane palącymi się dyskami twardymi i walczyli z administratorem

systemu na drzwiach pomieszczenia serwerowego, tylko po to, aby uzyskać dostęp do produkcji, ponieważ ktoś właśnie wdrożył nieprzetestowany kawałek kodu. Testowali swój kod kompresji oprogramowania na własnym kodzie źródłowym, tylko po to, aby odkryć, że wszystko zostało skompresowane do jednego bajtu, a wartość tego bajtu wynosi 255. Alorytm dekompresji musi być jeszcze wynaleziony.

Właśnie skończyłeś studia i szukasz pracy, albo fascynujesz się programowaniem, ale nie masz pojęcia, co cię czeka. Wyszedłeś z obozu programistycznego i szukasz możliwości zatrudnienia, ale nie jesteś pewien, co brakuje ci w umiejętnościach. Nauczyłeś się samodzielnie jakiegoś języka programowania, ale nie wiesz, czego brakuje ci w swoim zestawie narzędzi umiejętności. Witaj na ulicach.

1.3 Wspaniali programiści uliczni

Oprócz reputacji na ulicy, honoru i lojalności, programista uliczny powinien idealnie posiadać te cechy:

1.3.1 Podejście Krytyczne

Osoba rozmawiająca z samą sobą jest uważana za nietypową, zwłaszcza jeśli nie ma odpowiedzi na pytania, które sobie zadaje. Jednakże bycie osobą krytyczną, zadawanie pytań sobie, kwestionowanie najbardziej powszechnie akceptowanych pojęć i ich dekonstrukcja mogą oczyścić twoją wizję.

Wiele książek, ekspertów od oprogramowania i Slavoj Žižek podkreśla wagę bycia krytycznym i dociekliwym, ale niewielu z nich dostarcza ci narzędzi do pracy. W tej książce znajdziesz przykłady bardzo znanych technik i najlepszych praktyk oraz jak mogą być mniej skuteczne, niż twierdzą.

Krytyka techniki nie oznacza, że jest ona bezużyteczna. Jednakże poszerzy twoje horyzonty, dzięki czemu będziesz w stanie zidentyfikować pewne przypadki użycia, w których alternatywna technika może być lepsza.

Celem tej książki nie jest omówienie każdej techniki programowania od podstaw do końca, ale przedstawienie ci perspektywy, jak traktować najlepsze praktyki, jak je priorytetyzować na podstawie wartości, i jak możesz rozważyć za i przeciw alternatywnych podejść.

1.3.2 Skoncentrowany na Wynikach

Możesz być najlepszym programistą na świecie, znającym wszystkie niuanse rozwoju oprogramowania, potrafiącym wymyślić najlepszy projekt dla własnego kodu, ale to nic nie znaczy, jeśli nie dostarczasz, jeśli nie wypuszczasz produktu.

Według paradoksu Zenona, aby osiągnąć cel końcowy, musisz najpierw osiągnąć punkt w połowie drogi. To paradoks, ponieważ aby osiągnąć punkt w połowie drogi, musisz osiągnąć punkt w ćwiartce drogi, i tak dalej, co sprawia, że nie możesz osiągnąć żadnego miejsca. Zenon miał rację: aby mieć produkt końcowy, musisz spełniać terminy i osiągać po drodze cele pośrednie. W przeciwnym razie niemożliwe jest osiągnięcie celu końcowego. Bycie skoncentrowanym na wynikach oznacza również koncentrację na celach pośrednich, na postępach.

„Jak projekt ma się stać rok po terminie? ... Dzień po dniu."

—Fred Brooks, The Mythical Man Month

Osiąganie wyników może oznaczać poświęcenie jakości kodu, elegancji i doskonałości technicznej. Ważne jest, aby mieć takie spojrzenie na rzeczywistość i zachować kontrolę nad tym, co robisz i dla kogo.

Poświęcanie jakości kodu nie oznacza poświęcania jakości produktu. Jeśli masz dobre testy, jeśli są dobre wymagania pisemne, możesz nawet pisać wszystko w PHP. Może to jednak oznaczać, że w przyszłości będziesz musiał ponieść pewne koszty, ponieważ kod niskiej jakości w końcu cię ugryzie. Nazywa się to karmą kodu.

Niektóre z technik, które przyswoisz z tej książki, pomogą ci podejmować decyzje w celu osiągnięcia wyników.

1.3.3 Wysoka Wydajność

Największymi czynnikami wpływającymi na szybkość twego rozwoju są doświadczenie, dobre i jasne specyfikacje oraz mechaniczne klawiatury. Tylko żartuję - wbrew powszechnemu przekonaniu, mechaniczne klawiatury wcale nie przyspieszają twojej pracy. Po prostu wyglądają fajnie i świetnie irytują twojego partnera. W rzeczywistości nie sądzę, że prędkość pisania ma cokolwiek wspólnego z szybkością rozwoju. Twoja pewność w prędkości pisania może nawet skłonić cię do pisania bardziej rozbudowanego kodu, niż jest to konieczne.

Część wiedzy można zdobyć, ucząc się na cudzych błędach i desperacji. W tej książce znajdziesz przykłady takich przypadków. Techniki i wiedza, które przyswoisz, sprawią, że napiszesz mniej kodu, podejmiesz szybsze decyzje i pozwoli ci mieć jak najmniejsze zadłużenie techniczne, aby nie spędzać dni na rozplątywaniu kodu, który napisałeś zaledwie pół roku temu.

1.3.4 Przyjmowanie Złożoności i Niejasności

Złożoność jest przerażająca, a niejasność jeszcze bardziej, ponieważ nie wiesz nawet, jak bardzo powinieneś się bać, co jeszcze bardziej cię przeraża.

Radzenie sobie z niejasnością to jedna z kluczowych umiejętności, o które pytają rekruterzy Microsoftu podczas rozmów kwalifikacyjnych. Zwykle obejmuje to hipotetyczne pytania, takie jak "Ile jest warsztatów naprawy skrzypiec w Nowym Jorku?", "Ile jest stacji benzynowych w Los Angeles?" lub "Ile agentów tajnej służby ma prezydent i jaki mają grafik zmianowy? Podaj ich imiona, a najlepiej pokaż ich ścieżki chodzenia na tym planie Białego Domu."

Sztuczka polega na rozjaśnianiu wszystkiego, co wiesz o problemie i opracowaniu przybliżenia na podstawie tych faktów. Na przykład możesz zacząć od ludności Nowego Jorku i od ilości osób, które mogą tam grać na skrzypcach. To da ci pojęcie o wielkości rynku i o tym, ile konkurencji może obsłużyć ten rynek.

Podobnie, gdy stajesz przed problemem z nieznanymi parametrami, takim jak oszacowanie czasu potrzebnego do opracowania funkcji, zawsze możesz zawęzić okno przybliżenia na podstawie tego, co wiesz. Możesz wykorzystać swoją wiedzę na swoją korzyść i jak najbardziej ją wykorzystać, co może zredukować niejasność do minimum.

Co ciekawe, radzenie sobie z złożonością jest podobne. Coś, co wydaje się niezwykle skomplikowane, można podzielić na części, które są znacznie bardziej zarządzalne, mniej złożone i ostatecznie prostsze.

Im więcej rzeczy wyjaśnisz, tym więcej będziesz w stanie uporać się z nieznanym. Techniki, które przyswoisz z tej książki, rozjaśnią niektóre z tych rzeczy i sprawią, że będziesz pewniejszy w radzeniu sobie z niejasnością i złożonością.

1.4 Problemy nowoczesnego rozwoju oprogramowania

Oprócz wzrastającej złożoności, licznych warstw abstrakcji i moderacji na Stack Overflow, nowoczesny rozwój oprogramowania ma również inne problemy:

1. Jest zbyt wiele technologii: zbyt wiele języków programowania, zbyt wiele frameworków, i z pewnością zbyt wiele bibliotek, biorąc pod uwagę, że npm (menedżer pakietów dla frameworka Node.js) miał bibliotekę o nazwie "left-pad" służącą tylko do dodawania spacji na końcu ciągu znaków.
2. Jest napędzany paradygmatami, a zatem konserwatywny. Wielu programistów uważa języki programowania, najlepsze praktyki, wzorce projektowe, algorytmy i struktury danych za relikty starożytnej obcej rasy i nie ma pojęcia, jak one działają.
3. Technologia staje się bardziej nieprzejrzysta, podobnie jak samochody. Ludzie kiedyś potrafili samodzielnie naprawiać swoje samochody. Teraz, gdy silniki stają się coraz bardziej zaawansowane, pod maską widzimy tylko metalową pokrywę, podobną do tej na grobowcu faraona, która uwolni przeklęte dusze na każdego, kto ją otworzy. Technologie w rozwoju oprogramowania są podobne. Mimo że prawie wszystko jest teraz open source, uważam, że nowe technologie są bardziej niejasne niż dekompilowane kody binarne z lat 90., ze względu na ogromnie wzrosłą złożoność oprogramowania.
4. Ludzie nie przejmują się nadmiernym obciążeniem swojego kodu, ponieważ mamy dostęp do rzeczywistości w skali dziesiątek tysięcy. Napisałeś nową prostą aplikację do czatu? Dlaczego by jej nie spakować z pełnoprawną przeglądarką internetową, bo wiesz, że to po prostu oszczędza ci czas, i nikt nie kiwnie powieką, kiedy i tak używasz gigabajtów pamięci?
5. Programiści skupiają się na swoim stosie technologicznym i ignorują, jak reszta działa, i słusznie: muszą zapewnić jedzenie na stole, i nie ma czasu na naukę. Nazywam to "problemem kuchennych programistów". Wiele rzeczy, które wpływają na jakość ich produktu, pozostaje niezauważonych ze względu na ograniczenia, które mają. Programista webowy zazwyczaj nie ma pojęcia, jak działają protokoły sieciowe pod warstwą internetu. Akceptują opóźnienia podczas ładowania strony takie, jakie są, i uczą się żyć z nimi, ponieważ nie wiedzą, że drobny techniczny szczegół, jak zbyt długa łańcuch certyfikatów, może spowolnić ładowanie strony internetowej.
6. Ze względu na nauczane paradygmaty, istnieje stigma przeciwko monotonnej pracy, takiej jak powtarzanie się czy kopiuj-wklej. Oczekuje się, że znajdziesz rozwiązanie DRY (Don't Repeat Yourself). Tego rodzaju kultura sprawia, że zaczynasz wątpić w siebie i swoje umiejętności, co szkodzi twojej produktywności.

> **HISTORIA NPM I LEFT-PAD**
>
> npm stało się w ostatniej dekadzie de facto ekosystemem pakietów JavaScript. Ludzie mogli przyczyniać się do tego ekosystemu swoimi własnymi pakietami, a inne pakiety mogły z nich korzystać, ułatwiając rozwijanie dużych projektów. Azer Koçulu był jednym z tych programistów. Left-pad był tylko jednym z pakietów spośród 250, które przyczynił się do ekosystemu npm. Miał tylko jedną funkcję: dodawać spacje do ciągu znaków, aby zawsze miał stały rozmiar, co jest dość trywialne.
>
> Pewnego dnia otrzymał e-mail od npm, w którym poinformowano go, że usunięto jeden z jego pakietów o nazwie "Kik", ponieważ firma o tej samej nazwie złożyła skargę. npm zdecydowało się usunąć pakiet Azer'a i przyznać nazwę innej firmie. To sprawiło, że Azer był tak zły, że usunął wszystkie swoje pakietu, w tym left-pad. Problem polegał na tym, że były setki projektów o dużych rozmiarach na całym świecie, które bezpośrednio lub pośrednio używały tego pakietu. Jego działania spowodowały zatrzymanie wszystkich tych projektów. To było dość katastrofalne i dobre nauczanie zaufania, jakie

> mamy do platform.
>
> Morał z tej historii jest taki, że życie na ulicy jest pełne niepożądanych niespodzianek.

W tej książce proponuję rozwiązania tych problemów, w tym przejście przez niektóre podstawowe koncepcje, które mogą ci się wydać nudne, priorytetyzowanie praktyczności, prostoty, odrzucanie niektórych długo utrzymywanych niepodważalnych przekonań i, co najważniejsze, kwestionowanie wszystkiego, co robimy. Wartość polega na zadawaniu pytań najpierw.

1.4.1 Zbyt wiele technologii

Nasze nieustanne poszukiwanie najlepszej technologii wynika z fałszywego przekonania o istnieniu srebrnej kuli. Myślimy, że istnieje technologia, która może zwiększyć naszą produktywność o rzędy wielkości. Nie istnieje. Na przykład Python7 to język interpretowany. Nie musisz kompilować kodu Pythona - działa od razu. Co więcej, nie musisz określać typów zmiennych, które deklarujesz, co sprawia, że jesteś jeszcze szybszy, więc Python musi być lepszą technologią niż C#, prawda? Niekoniecznie.

Ponieważ nie poświęcasz czasu na adnotowanie swojego kodu typami i kompilację, pomijasz popełniane błędy. Oznacza to, że możesz je odkryć tylko podczas testowania lub w produkcji, co jest znacznie droższe niż po prostu kompilacja kodu. Większość technologii to kompromisy, a nie zwiększacze produktywności. To, co zwiększa twoją produktywność, to jak biegły jesteś w tej technologii i jakie masz techniki, a nie to, jakie technologie używasz. Tak, są lepsze technologie, ale rzadko sprawiają one różnicę rzędu wielkości.

Kiedy chciałem stworzyć moją pierwszą interaktywną stronę internetową w 1999 roku, absolutnie nie miałem pojęcia, jak zacząć pisać aplikację internetową. Gdybym próbował najpierw znaleźć najlepszą technologię, musiałbym nauczyć się VBScript lub Perl. Zamiast tego użyłem tego, co najlepiej znałem wtedy: Pascal.8 Był to jeden z najmniej odpowiednich języków do tego celu, ale zadziałał. Oczywiście były z nim problemy. Za każdym razem, gdy się zawieszał, proces pozostawał aktywny w pamięci na losowym serwerze w Kanadzie, i użytkownik musiał dzwonić do dostawcy usług za każdym razem i prosić o ponowne uruchomienie fizycznego serwera. Mimo to Pascal pozwolił mi szybko osiągnąć prototyp, ponieważ czułem się z nim swobodnie. Zamiast uruchomić stronę internetową, którą sobie wyobrażałem po miesiącach pracy i nauki, napisałem i opublikowałem kod w trzy godziny.

Z niecierpliwością czekam, aby pokazać ci sposoby, dzięki którym możesz być bardziej efektywny, korzystając z istniejącego zestawu narzędzi, które masz pod ręką.

1.4.2 Paralotniarstwo na paradoksach

Najwcześniejszym paradygmatem programowania, z którym się spotkałem, był programowanie strukturalne w latach 80. Programowanie strukturalne polegało głównie na pisaniu kodu w strukturalnych blokach, takich jak funkcje i pętle, zamiast numerów linii, instrukcji GOTO czy wysiłku i łez. Sprawiało to, że kod był łatwiejszy do czytania i utrzymania, nie tracąc przy tym wydajności. Programowanie strukturalne zainteresowało mnie językami programowania takimi jak Pascal i C.

Następny paradygmat, z którym się zetknąłem, pojawił się co najmniej pół dekady po tym, jak nauczyłem się o programowaniu strukturalnym: programowanie obiektowe, czyli OOP (Object-Oriented Programming). Pamiętam, że wtedy magazyny komputerowe nie mogły się go doczekać. Była to kolejna wielka rzecz, która pozwoliła nam pisać jeszcze lepsze programy niż w przypadku programowania strukturalnego.

Po OOP spodziewałem się, że będę zetknął się z nowym paradygmatem co pięć lat lub coś w tym stylu. Jednakże zaczęły się one pojawiać częściej. W latach 90. poznaliśmy JIT-compiled9 zarządzane języki programowania dzięki nadejściu Javy, skrypty internetowe z użyciem JavaScript oraz programowanie funkcyjne, które powoli zaczęło przesuwać się do mainstreamu pod koniec lat 90.

Potem przyszły lata 2000. W następnych dekadach zaczęliśmy używać coraz częściej terminu aplikacje wielowarstwowe N-tier. Grube klienty. Cienkie klienty. Generyki. MVC, MVVM i MVP. Programowanie asynchroniczne zaczęło się rozpowszechniać dzięki obietnicom (promises), przyszłościom (futures) i ostatecznie programowaniu reaktywnemu. Mikroserwisy. Więcej koncepcji związanych z programowaniem funkcyjnym, takich jak LINQ, dopasowywanie wzorców (pattern matching) i niemutowalność, znalazło się w językach mainstreamowych. To jest tornado słów kluczowych.

Nawet nie wspomniałem jeszcze o wzorcach projektowych czy najlepszych praktykach. Mamy niezliczone najlepsze praktyki, porady i sztuczki dotyczące prawie każdego tematu. Napisano manifesty na temat tego, czy powinniśmy używać tabulacji czy spacji do wcięć w kodzie źródłowym, mimo że oczywista odpowiedź brzmi: spacje.10

Zakładamy, że nasze problemy można rozwiązać, stosując paradygmat, wzorzec, framework lub bibliotekę. Biorąc pod uwagę złożoność problemów, które teraz mamy do rozwiązania, to nie jest bezpodstawne. Jednak ślepe przyjmowanie tych narzędzi może przysporzyć więcej problemów w przyszłości: mogą spowolnić pracę, wprowadzając nową wiedzę dziedzinową do nauki oraz własne zestawy błędów. Mogą nawet zmusić cię do zmiany projektu. Ta książka sprawi, że będziesz bardziej pewny, że używasz wzorców poprawnie, podchodząc do nich bardziej dociekliwie, i będziesz gromadzić dobre argumenty do użycia podczas przeglądów kodu.

1.4.3 Czarne skrzynki technologii

Framework lub biblioteka to pakiet. Programiści instalują go, czytają jego dokumentację i używają. Ale zwykle nie wiedzą, jak działa. Tak samo podchodzą do algorytmów i struktur danych. Używają słownika, ponieważ wygodnie jest przechowywać klucze i wartości. Nie znają konsekwencji.

Niewzruszone zaufanie do ekosystemów pakietów i frameworków jest podatne na poważne błędy. Może nas to kosztować dni debugowania, ponieważ po prostu nie wiedzieliśmy, że dodawanie elementów do słownika o tym samym kluczu będzie różnić się od listy pod względem wydajności wyszukiwania. Używamy generatorów w C#, gdy wystarczyłby prosty tablica, i doznajemy znacznego pogorszenia wydajności, nie wiedząc dlaczego.

Pewnego dnia w 1993 roku przyjaciel podał mi kartę dźwiękową i poprosił, żebym zainstalował ją w moim komputerze PC. Tak, kiedyś potrzebowaliśmy dodatkowych kart, żeby uzyskać porządny dźwięk z PC, bo inaczej słyszeliśmy tylko beep. W każdym razie nigdy wcześniej nie otwierałem obudowy swojego komputera, i bałem się, że coś zniszczę. Powiedziałem mu: "Czy nie możesz tego zrobić za mnie?" Mój przyjaciel powiedział mi: "Musisz to otworzyć, żeby zobaczyć, jak to działa."

To rezonowało we mnie, ponieważ zrozumiałem, że moje lęki wynikały z mojej niewiedzy, a nie z mojej niezdolności. Otwarcie obudowy i zobaczenie wnętrza mojego własnego komputera mnie uspokoiło. Zawierał tylko kilka płyt. Karta dźwiękowa wchodziła do jednego z gniazd. To dla mnie już nie było tajemnicze pudełko. Później użyłem tej samej techniki, ucząc studentów szkoły artystycznej podstaw komputerów. Otworzyłem myszkę i pokazałem im jej kulkę. Myszy miały kule wtedy. No cóż, to było niestety dwuznaczne. Otworzyłem obudowę komputera. "Widzicie, to nie jest straszne, to jest płyta główna i kilka

gniazd."

To później stało się moim mottem w radzeniu sobie z czymś nowym i skomplikowanym. Przestałem się bać otwierać pudełko i zwykle robiłem to jako pierwszą rzecz, żeby móc zmierzyć się z pełnym zakresem złożoności, która zawsze była mniejsza, niż się jej obawiałem.

Podobnie szczegóły tego, jak działa biblioteka, framework czy komputer, mogą mieć ogromny wpływ na twoje zrozumienie tego, co jest na nich zbudowane. Otwarcie pudełka i przyjrzenie się częściom może pomóc ci w prawidłowym użyciu pudełka. Nie musisz od razu czytać kodu od początku do końca ani przechodzić przez tysiącstronicową książkę teorii, ale przynajmniej powinieneś wiedzieć, która część idzie gdzie i jak może wpływać na twoje przypadki użycia.

Dlatego niektóre z tematów, o których będę mówił, są fundamentalne lub niskopoziomowe. Chodzi o otwarcie pudełka i zobaczenie, jak to działa, żebyśmy mogli podejmować lepsze decyzje w programowaniu na wysokim poziomie.

1.4.4 Zbyt lekceważony nadmiar

Cieszę się, że każdego dnia widzimy coraz więcej aplikacji opartych na chmurze. Są one nie tylko opłacalne, ale także rzeczywistym sprawdzianem dla zrozumienia rzeczywistych kosztów naszego kodu. Kiedy zaczynasz płacić dodatkowego centa za każdą złą decyzję w kodzie, nadmiar nagle staje się problemem.

Frameworki i biblioteki zwykle pomagają nam unikać nadmiaru, co czyni je użytecznymi abstrakcjami. Niemniej jednak nie możemy przekazywać całego naszego procesu podejmowania decyzji frameworkom. Czasami musimy sami podejmować decyzje i musimy uwzględnić nadmiar. W przypadku aplikacji o dużym zasięgu nadmiar staje się jeszcze bardziej istotny. Każdy millisekundy, który oszczędzisz, może pomóc odzyskać cenne zasoby.

Priorytetem programisty nie powinno być eliminowanie nadmiaru. Niemniej jednak wiedza o tym, jak unikać nadmiaru w określonych sytuacjach i perspektywa jako narzędzie w twoim arsenale pomogą ci zaoszczędzić czas, zarówno dla siebie, jak i dla użytkownika, który czeka na ten kręcący się spinner na twojej stronie internetowej.

W trakcie lektury książki znajdziesz scenariusze i przykłady, jak można łatwo unikać nadmiaru, nie robiąc z tego swojego najważniejszego celu.

1.4.5 To nie moja praca

Jednym ze sposobów radzenia sobie z złożonością jest skupienie się wyłącznie na swoich obowiązkach: na komponencie, który posiadasz, na kodzie, który piszesz, na błędach, które popełniłeś, i czasami na rozsadzanym lasagne w mikrofalówce w biurze. Może się to wydawać najbardziej efektywnym sposobem wykonywania pracy, ale jak wszelkie istoty, tak i kod jest ze sobą powiązany.

Poznanie, jak działa określona technologia, jak biblioteka wykonuje swoją pracę i jakie są zależności oraz jak są ze sobą połączone, pozwala nam podejmować lepsze decyzje, gdy piszemy kod. Przykłady w tej książce dadzą ci perspektywę skupienia się nie tylko na swojej dziedzinie, ale także na jej zależnościach i problemach, które wykraczają poza twoją strefę komfortu, ponieważ odkryjesz, że one przewidują los twojego kodu.

1.4.6 Powszednie jest genialne

Wszystkie zasady nauczane w rozwoju oprogramowania sprowadzają się do jednego napomnienia: spędzaj mniej czasu wykonując swoją pracę. Unikaj powtarzających się, bezmyślnych zadań, takich jak kopiowanie i wklejanie oraz pisanie tego samego kodu od zera z drobnymi zmianami. Po pierwsze, zajmują one więcej czasu, a po drugie, jest niezwykle trudno je utrzymać.

Nie wszystkie zadania powszednie są złe. Nawet kopiowanie i wklejanie nie są złe. Istnieje silne piętno wobec nich, ale są sposoby, aby uczynić je bardziej wydajnymi niż niektóre najlepsze praktyki, które cię nauczono.

Ponadto nie cały kod, który piszesz, działa jako kod rzeczywistego produktu. Niektóry kod, który piszesz, będzie używany do opracowywania prototypu, niektóry będzie służył do testów, a niektóry będzie cię przygotowywać do właściwego zadania. Omówię niektóre z tych scenariuszy i jak możesz wykorzystać te zadania na swoją korzyść.

1.5 Czym książka nie jest

Ta książka nie jest kompleksowym przewodnikiem po programowaniu, algorytmach ani żadnym innym zagadnieniu. Nie uważam się za eksperta w konkretnych tematach, ale mam wystarczającą wiedzę na temat rozwoju oprogramowania. Książka składa się głównie z informacji, które nie są oczywiste w popularnych i znakomitych książkach dostępnych na rynku. To zdecydowanie nie jest przewodnikem do nauki programowania.

Doświadczeni programiści mogą znaleźć niewielkie korzyści z tej książki, ponieważ zdobyli już wystarczającą wiedzę i stali się już programistami znającymi ulice. Niemniej jednak mogą być zaskoczeni niektórymi spostrzeżeniami zawartymi w tej książce.

Ta książka to także eksperyment w dziedzinie tego, jak książki programistyczne mogą być zabawne w czytaniu. Chciałbym przede wszystkim przedstawić programowanie jako zabawę. Książka nie traktuje siebie zbyt poważnie, więc ty też nie powinieneś. Jeśli po przeczytaniu książki poczujesz się lepszym programistą i będziesz się dobrze bawić, uznaję siebie za spełnionego.

1.6 Tematy

Pewne tematy będą się powtarzać w całej książce:

1. Minimalna wiedza podstawowa, która wystarcza, aby poradzić sobie na ulicach. Te tematy nie będą wyczerpujące, ale mogą zainteresować cię, jeśli wcześniej wydawały ci się nudne. To zazwyczaj kluczowa wiedza, która pomaga podjąć decyzje.
2. Powszechnie znane lub akceptowane najlepsze praktyki lub techniki, które ja proponuję jako antywzorce, które w pewnych przypadkach mogą być bardziej efektywne. Im więcej przeczytasz na ich temat, tym bardziej zostanie wyostrzony twój szósty zmysł do krytycznego myślenia o praktykach programistycznych.
3. Pozornie nieistotne techniki programowania, takie jak triki optymalizacji na poziomie CPU, które mogą wpływać na twoje decyzje i pisanie kodu na wyższym poziomie. Znajomość wewnętrznych mechanizmów, "otwieranie pudełka", ma ogromną wartość, nawet jeśli nie korzystasz bezpośrednio z tej wiedzy.

4. Techniki, które uważam za przydatne w mojej codziennej pracy programisty, które mogą ci pomóc zwiększyć swoją produktywność, w tym obgryzanie paznokci i stawanie się niewidzialnym dla swojego szefa.

Te tematy będą podkreślać nową perspektywę, kiedy będziesz przyglądać się zagadnieniom programistycznym, zmieniać twoje zrozumienie pewnych "nudnych" tematów i być może zmieniać twoje podejście do pewnych dogmatów. Pomogą ci czerpać radość z pracy.

Podsumowanie

Surowa rzeczywistość "ulic", czyli świata profesjonalnego rozwoju oprogramowania, wymaga umiejętności, które nie są nauczane lub nie są priorytetem w formalnym kształceniu, a czasem zupełnie pomijane podczas samodzielnej nauki.

Nowi programiści często albo zwracają uwagę na teorię, albo zupełnie ją ignorują. Ostatecznie znajdziesz złoty środek, ale można go przyspieszyć pewną perspektywą.

Współczesny rozwój oprogramowania jest znacznie bardziej złożony niż kilka dekad temu. Aby stworzyć prostą działającą aplikację, wymaga się ogromnej wiedzy na wielu poziomach.

Programiści stoją przed dylematem między tworzeniem oprogramowania a nauką. Można to przezwyciężyć, zmieniając sposób postrzegania tematów w sposób bardziej pragmatyczny.

Brak jasności co do tego, nad czym pracujesz, sprawia, że programowanie staje się nudnym zadaniem, co w rezultacie obniża twoją rzeczywistą produktywność. Lepsze zrozumienie tego, czym się zajmujesz, przyniesie ci więcej radości.

1. Haskell to ezoteryczny język programowania, który został stworzony jako wyzwanie, aby zmieścić jak najwięcej prac naukowych w jednym języku programowania.
2. Linus Torvalds stworzył system operacyjny Linux i oprogramowanie do kontroli wersji Git oraz poparł przekonanie, że przeklinanie wolontariuszy pracujących nad projektem jest w porządku, jeśli są technicznie w błędzie.
3. Slavoj Žižek to współczesny filozof, który cierpi na schorzenie, które zmusza go do krytykowania wszystkiego na świecie, bez wyjątków.
4. Zenon z Elei był starożytnym Grekiem, który żył tysiące lat temu i nie mógł przestać zadawać frustrujących pytań. Naturalnie żadne z jego pism nie przetrwały. Wszystkie Zenki to fajne chłopaki (oprócz Zenka Martyniuka).
5. PHP był kiedyś językiem programowania, który stanowił przykład tego, jak nie projektować języka programowania. O ile wiem, PHP przeszedł długą drogę od czasów, gdy był obiektem żartów programistycznych, i teraz jest fantastycznym językiem programowania. Niemniej jednak wciąż ma pewne problemy z wizerunkiem marki do rozwiązania.
6. DRY. Nie powtarzaj się. Przesąd, że jeśli ktoś powtarza linię kodu zamiast owijać ją w funkcję, natychmiast zostanie przekształcony w żabę.
7. Python to kolektywna próba promowania białych znaków, udająca praktyczny język programowania.
8. Wczesny kod źródłowy Ekşi Sözlük jest dostępny na GitHubie: https://github.com/ssg/sozluk-cgi.
9. JIT, kompilacja "just-in-time". Mityczne pojęcie stworzone przez firmę Sun Microsystems, twórcę języka Java, że jeśli skompilujesz kod podczas jego działania, stanie się szybszy, ponieważ optymalizator będzie miał więcej danych zebranych podczas działania. To nadal mit.

10. Napisałem o debacie dotyczącej używania tabulatorów czy spacji z pragmatycznego punktu widzenia: https://medium.com/@ssg/tabs-vs-spaces-towards-a-better-bike-shed-686e111a5cce.
11. Spinnery są współczesnymi klepsydrami w świecie komputerów. W czasach starożytnych, komputery używały klepsydr do sprawienia, żebyś czekał przez nieokreślony czas. Spinner to nowoczesny odpowiednik tej animacji. Zazwyczaj jest to nieskończony obracający się łuk. To tylko dystrakcja, która pozwala utrzymać frustrację użytkownika pod kontrolą.

Praktyczna teoria

Rozdział ten obejmuje:

- **Dlaczego teoria informatyki jest istotna dla twojego przetrwania**
- **Jak wykorzystać typy danych na swoją korzyść**
- **Zrozumienie cech algorytmów**
- **Struktury danych i ich dziwaczne cechy, o których twoi rodzice zapomnieli ci powiedzieć**

Wbrew powszechnie przyjętemu przekonaniu, programiści są ludźmi. Mają te same błędy poznawcze co inni ludzie w praktyce tworzenia oprogramowania. Szeroko przeceniają korzyści płynące z pomijania typów danych, niezwracania uwagi na poprawne struktury danych czy zakładania, że algorytmy są ważne jedynie dla autorów bibliotek.

Ty nie jesteś wyjątkiem. Oczekuje się od ciebie, że dostarczysz produkt na czas, z dobrą jakością, i z uśmiechem na twarzy. Jak mówi przysłowie, programista to efektywny organizm, który otrzymuje kawę jako wejście i tworzy oprogramowanie jako wyjście. Możesz równie dobrze pisać wszystko na najgorszy możliwy sposób, używać kopiuj-wklej, korzystać z kodu znalezionego na Stack Overflow, używać plików tekstowych do przechowywania danych, albo nawet zawierać pakt z demonem, jeśli twoja dusza jeszcze nie podpisała umowy o poufności.1 Tylko twoi rówieśnicy naprawdę przejmują się tym, jak wykonujesz swoją pracę – wszyscy inni chcą po prostu, aby produkt działał i był dobry.

Teoria może być przytłaczająca i niezwiązana z rzeczywistością. Algorytmy, struktury danych, teoria typów, notacja Big-O i złożoność wielomianowa mogą wydawać się skomplikowane i nieistotne w kontekście tworzenia oprogramowania. Istniejące biblioteki i frameworki już zajmują się tym w zoptymalizowany i przetestowany sposób. Zazwyczaj jest zalecane, aby nigdy nie implementować algorytmu od zera, zwłaszcza w kontekście bezpieczeństwa informacji lub napiętych terminów.

To dlaczego powinieneś interesować się teorią? Ponieważ wiedza z dziedziny informatyki pozwala ci nie tylko rozwijać algorytmy i struktury danych od podstaw, ale także właściwie określać, kiedy należy zastosować daną technikę. Pomaga zrozumieć koszty podejmowanych decyzji kompromisowych. Pomaga zrozumieć charakterystyki skalowalności kodu, który piszesz. Pozwala patrzeć w przyszłość. Prawdopodobnie nigdy nie będziesz implementować struktury danych ani algorytmu od podstaw, ale wiedza na temat ich działania uczyni cię efektywnym programistą. Poprawi twoje szanse na przetrwanie na "ulicach".

Ta książka omówi tylko pewne kluczowe aspekty teorii, które mogłeś przeoczyć w czasie nauki - niektóre mniej znane aspekty typów danych, zrozumienie złożoności algorytmów i sposób działania pewnych struktur danych. Jeśli wcześniej nie uczyłeś się o typach danych, algorytmach lub strukturach danych, ten rozdział dostarczy ci wskazówek, które mogą zainteresować cię tym tematem.

2.1 Krótki kurs na temat algorytmów

Algorytm to zestaw reguł i kroków mających na celu rozwiązanie problemu. Dziękuję za udział w moim wykładzie TED. Spodziewałeś się bardziej skomplikowanej definicji, prawda? Na przykład przeglądanie elementów tablicy w celu sprawdzenia, czy zawiera określoną liczbę, jest algorytmem, nawet jeśli prostym:

```
public static bool Contains(int[] array, int lookFor) {
   for (int n = 0; n < array.Length; n++) {
       if (array[n] == lookFor) {
           return true;
       }
   }
   return false;
}
```

Moglibyśmy nazwać to "Algorytmem Sedata", gdybym to ja był osobą, która go wynalazła, ale prawdopodobnie był to jeden z pierwszych algorytmów, które kiedykolwiek powstały. Nie jest w żaden sposób wymyślny, ale działa i ma sens. To jeden z istotnych punktów dotyczących algorytmów: muszą one działać zgodnie z twoimi potrzebami. Niekoniecznie muszą zdziałać cuda. Kiedy wkładasz naczynia do zmywarki i ją uruchamiasz, również stosujesz algorytm. Istnienie algorytmu nie oznacza, że jest on inteligentny.

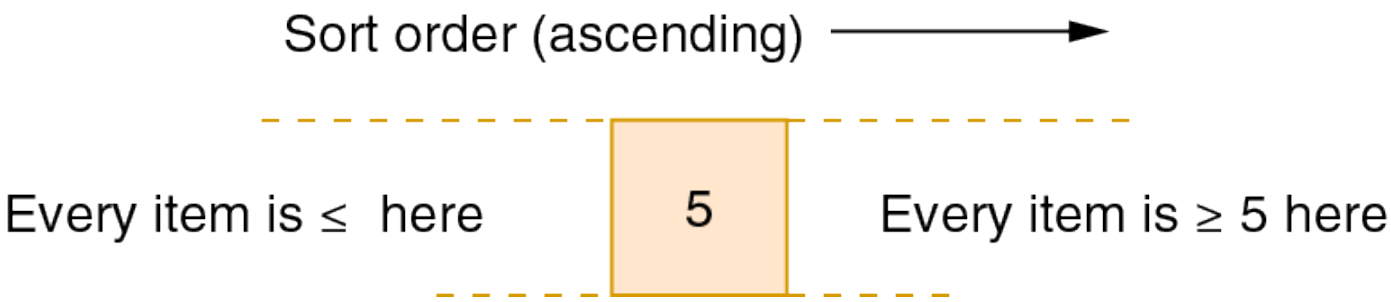
Mając powiedziane, mogą istnieć bardziej zaawansowane algorytmy, w zależności od twoich potrzeb. W poprzednim przykładzie kodu, jeśli wiesz, że lista zawiera tylko liczby całkowite, możesz dodać specjalne obsługi dla liczb niebędących dodatnimi:

```
public static bool Contains(int[] array, int lookFor) {
   if (lookFor < 1) {
       return false;
   }
   for (int n = 0; n < array.Length; n++) {
       if (array[n] == lookFor) {
           return true;
       }
   }
   return false;
}
```

To może znacznie przyspieszyć działanie algorytmu, w zależności od tego, jak często jest wywoływany z liczbą ujemną. W najlepszym przypadku twoja funkcja zawsze byłaby wywoływana z liczbami ujemnymi lub zerami, zwracając od razu, nawet jeśli tablica miała miliardy liczb całkowitych. W najgorszym przypadku twoja funkcja byłaby zawsze wywoływana z liczbami dodatnimi, a ty miałbyś tylko jedno dodatkowe, zbędne sprawdzenie. Typy danych mogą ci tu pomóc, ponieważ w języku C# istnieją bezznakowe wersje liczb całkowitych, nazywane uint. Dzięki nim możesz zawsze otrzymywać liczby dodatnie, a kompilator będzie sprawdzał to za ciebie, nie wprowadzając żadnych problemów z wydajnością:

```
public static bool Contains(uint[] array, uint lookFor) {
   for (int n = 0; n < array.Length; n++) {
       if (array[n] == lookFor) {
           return true;
       }
   }
   return false;
}
```

Naprawiliśmy wymaganie dotyczące liczb dodatnich, stosując restrykcje typów, zamiast zmieniać nasz algorytm, ale nadal może być on szybszy w zależności od kształtu danych. Czy posiadamy więcej informacji o danych? Czy tablica jest posortowana? Jeśli tak, możemy przypuszczać więcej o miejscu, w którym może znajdować się nasza liczba. Porównując naszą liczbę z dowolnym elementem w tablicy, możemy łatwo wykluczyć ogromną ilość elementów (zobacz rysunek 2.1).



Jeśli nasza liczba wynosi na przykład 3, i porównujemy ją z 5, możemy być pewni, że nasza liczba nie znajdzie się po prawej stronie od 5. To oznacza, że możemy natychmiast wyeliminować wszystkie elementy znajdujące się po prawej stronie listy.

Zatem, jeśli wybierzemy element ze środka listy, możemy zagwarantować, że po porównaniu możemy natychmiast wyeliminować co najmniej połowę listy. Możemy zastosować tę samą logikę do pozostałej części, wybierając środkowy punkt i kontynuując. Oznacza to, że potrzebujemy co najwyżej 3 porównań do posortowanej tablicy z 8 elementami, aby stwierdzić, czy dany element istnieje w niej. Co ważniejsze, zajmie to maksymalnie około 10 przeszukań, aby stwierdzić, czy dany element istnieje w tablicy z 1000 elementami. To jest moc, którą uzyskujesz, dzieląc na pół. Twoja implementacja mogłaby wyglądać jak w przykładzie 2.1. W zasadzie ciągle znajdujemy środkowe miejsce i eliminujemy pozostałą połowę, zależnie od tego, jak wartość, której szukamy, wpasowuje się w nią. Zapisujemy formułę w dłuższej, bardziej rozbudowanej formie, chociaż odpowiada ona (start + koniec) / 2. To dlatego, że start + koniec może przekroczyć wartość dla dużych wartości startu i końca i doprowadzić do znalezienia nieprawidłowego środka. Jeśli zapiszesz wyrażenie tak, jak w poniższym przykładzie, unikniesz tego przypadku przepełnienia.

Przykład 2.1 Wyszukiwanie w posortowanej tablicy za pomocą wyszukiwania binarnego

```
public static bool Contains(uint[] array, uint lookFor) {
  int start = 0;
  int end = array.Length - 1;
  while (start <= end) {
    int middle = start + ((end - start) / 2);
    uint value = array[middle];
    if (lookFor == value) {
      return true;
    }
```

```
    if (lookFor > value) {
      start = middle + 1;
    } else {
      end = middle - 1;
    }
  }
  return false;
}
```

Tutaj zaimplementowaliśmy wyszukiwanie binarne, znacznie szybszy algorytm niż Algorytm Sedata. Teraz, gdy możemy sobie wyobrazić, jak wyszukiwanie binarne może być szybsze niż zwykła iteracja, możemy zacząć myśleć o szanowanej notacji Big-O.

2.1.1 Big-O musi być dobrze zrozumiane

Zrozumienie wzrostu to doskonała umiejętność dla programisty. Bez względu na to, czy chodzi o rozmiar czy liczbę, kiedy wiesz, jak szybko coś rośnie, możesz zobaczyć przyszłość i ocenić, w jakiego rodzaju kłopoty się wpakowujesz, zanim na to zbyt dużo czasu poświęcisz. Jest to szczególnie przydatne, gdy światło na końcu tunelu rośnie, choć ty się nie poruszasz.

Notacja Big-O, jak sama nazwa wskazuje, jest po prostu sposobem wyjaśnienia wzrostu, ale również podlega nieporozumieniom. Kiedy pierwszy raz zobaczyłem O(N), myślałem, że to zwykła funkcja, która powinna zwracać liczbę. Ale nie jest. To sposób, w jaki matematycy opisują wzrost. Daje nam podstawowy pomysł na to, jak skalowalny jest algorytm. Przechodzenie przez każdy element sekwencyjnie (zwane także Algorytmem Sedata) wymaga liczby operacji liniowo proporcjonalnej do liczby elementów w tablicy. Oznaczamy to, pisząc O(N), gdzie N oznacza liczbę elementów. Nadal nie możemy dokładnie określić, ile kroków algorytm wykona, patrząc tylko na O(N), ale wiemy, że wzrasta to liniowo. Pozwala nam to dokonywać założeń na temat charakterystyki wydajności algorytmu w zależności od wielkości danych. Możemy przewidzieć, w którym momencie może stać się to problemem, patrząc na to.

Wyszukiwanie binarne, które zaimplementowaliśmy, ma złożoność O(log2n). Jeśli nie jesteś zaznajomiony z logarytmami, jest to przeciwieństwo funkcji wykładniczej, więc złożoność logarytmiczna jest naprawdę wspaniała, chyba że chodzi o pieniądze. W tym przykładzie, jeśli nasz algorytm sortowania magicznie miałby złożoność logarytmiczną, potrzebowałby tylko 18 porównań, aby posortować tablicę z 500 000 elementów. Nasza implementacja wyszukiwania binarnego jest więc świetna.

Notacja Big-O nie służy tylko do pomiaru wzrostu kroków obliczeniowych, czyli złożoności czasowej, ale również do pomiaru wzrostu zużycia pamięci, co nazywane jest złożonością przestrzeniową. Algorytm może być szybki, ale może mieć wzrost wielomianowy w pamięci, jak w naszym przykładzie sortowania. Powinniśmy zrozumieć tę różnicę.

> PORADA
>
> Wbrew powszechnemu przekonaniu, O(Nx) nie oznacza złożoności wykładniczej. Oznacza to złożoność wielomianową, która, choć dosyć zła, nie jest tak straszna jak złożoność wykładnicza, która jest oznaczana jako O(xn). Przy zaledwie 100 elementach O(N2) wykona 10 000 iteracji, podczas gdy O(2n) wykona jakąś nieprawdopodobną liczbę iteracji z 30 cyframi—nawet nie jestem w stanie tego wymówić. Istnieje także złożoność silniowa, która jest jeszcze gorsza niż wykładnicza, ale nie widziałem żadnych algorytmów, które by ją używały, poza obliczaniem permutacji lub kombinacji,

> prawdopodobnie dlatego, że nikt nie był w stanie jej wynaleźć do końca.

Algorytm wyszukiwania	**Złożoność**	**Czas znalezienia rekordu wśród 60 wierszy**
Domowy kwantowy komputer, który wujek Lisy ma w swoim garażu	O(1)	1 sekunda
Wyszukiwanie binarne	O(log N)	6 sekund
Wyszukiwanie liniowe (ponieważ szef poprosił cię o to godzinę przed prezentacją)	O(N)	60 sekund
Praktykant przypadkowo dodał zagnieżdżone dwie pętle.	O(N^2)	1 godzina
Jakiś kod wklejony przypadkowo ze Stack Overflow, który znajduje również rozwiązanie pewnego problemu szachowego podczas wyszukiwania, ale programista nie zadał sobie trudu, aby to usunąć	O(2^N)	36,5 miliarda lat
Zamiast znalezienia rzeczywistego rekordu, algorytm próbuje znaleźć układ rekordów, które ułożone w pewien sposób tworzą szukany rekord. Dobra wiadomość polega na tym, że ten programista już tu nie pracuje.	O(N!)	Koniec wszechświata, ale nadal przed tym, zanim małpy skończą swoje tak zwane "Shakespeare'y"

Musisz być zaznajomiony z tym, jak notacja Big-O opisuje wzrost prędkości wykonania algorytmu i zużycia pamięci, abyś mógł podejmować świadome decyzje podczas wyboru struktury danych i algorytmu do użycia. Zapoznaj się z notacją Big-O, nawet jeśli nie musisz implementować algorytmu. Uważaj na złożoność.

2.2 Struktury danych od środka

Na początku była pustka. Kiedy pierwsze sygnały elektryczne trafiły do pierwszego bitu w pamięci, zrodziły się dane. Dane były początkowo wolno unoszącymi się bajtami. Te bajty zeszły się razem i stworzyły strukturę.

— Początek 0:1

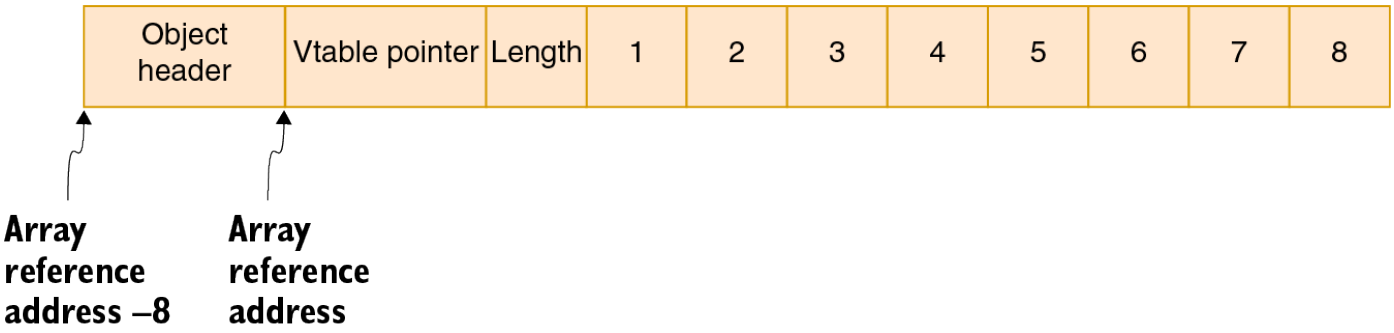
Struktury danych dotyczą tego, jak dane są rozmieszczone. Ludzie odkryli, że gdy dane są rozmieszczone w określony sposób, mogą być bardziej przydatne. Lista zakupów na kartce jest łatwiejsza do czytania, jeśli każdy przedmiot znajduje się w osobnej linii. Tabela mnożenia jest bardziej użyteczna, jeśli jest ułożona w siatkę. Zrozumienie, jak działa określona struktura danych, jest kluczowe dla twojego rozwoju jako programisty. To zrozumienie zaczyna się od zerknięcia pod maskę i przyjrzenia się, jak struktura działa.

Przyjrzyjmy się na przykład tablicom. Tablica w programowaniu to jedna z najprostszych struktur danych, a jest rozmieszczana jako ciągłe elementy w pamięci. Załóżmy, że masz taką tablicę:

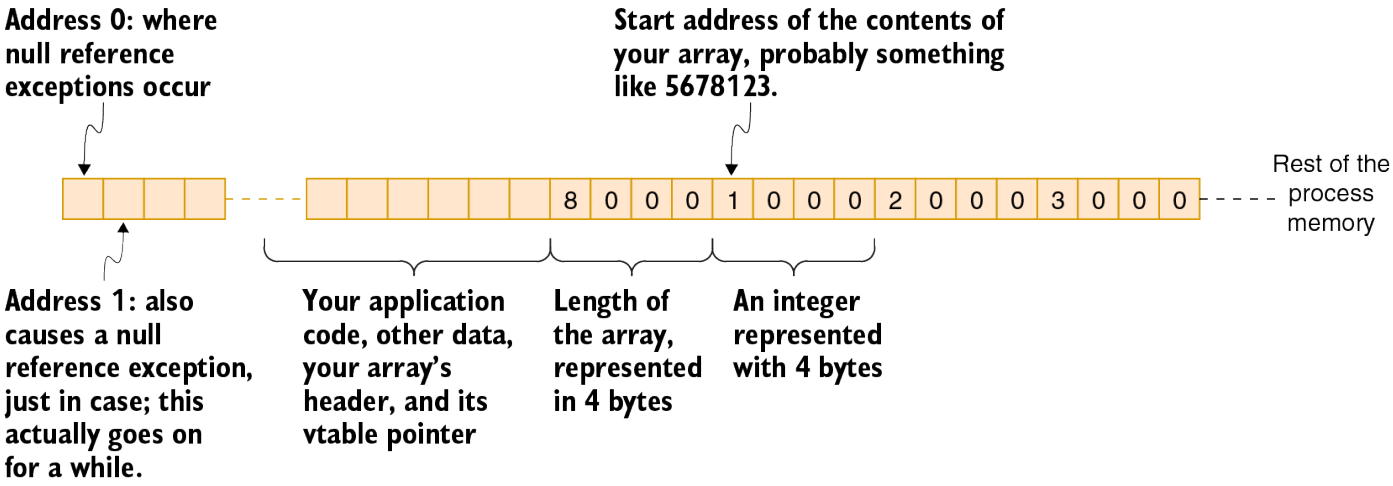
```
1 var wartosci = new int[] { 1, 2, 3, 4, 5, 6, 7, 8 };
```

Możesz sobie wyobrazić, jak wyglądałaby w pamięci, jak na rysunku 2.2.

Tak naprawdę nie wyglądałoby to w pamięci w taki sposób, ponieważ każdy obiekt w .NET ma określony nagłówek, wskaźnik do tablicy wskaźników do tabeli wirtualnych metod oraz informacje o długości, jak pokazano na rysunku 2.3.



Staje się to jeszcze bardziej realistyczne, gdy spojrzysz na to, jak jest umieszczone w pamięci RAM, ponieważ RAM nie jest zbudowany z pojedynczych liczb całkowitych, jak pokazano na rysunku 2.4. Dzielę się tymi informacjami, ponieważ chcę, abyś nie bał się tych niskopoziomowych koncepcji. Ich zrozumienie pomoże ci na wszystkich poziomach programowania.



To nie jest wygląd twojej rzeczywistej pamięci RAM, ponieważ każdy proces ma swój własny fragment pamięci do niego dedykowany, zgodnie z tym, jak działają nowoczesne systemy operacyjne. Ale ten układ będzie zawsze mieć z nim do czynienia, chyba że zaczniesz tworzyć swój własny system operacyjny lub własne sterowniki urządzeń.

Ogólnie rzecz biorąc, sposób, w jaki dane są rozmieszczone, może sprawić, że rzeczy będą działać szybciej lub bardziej efektywnie, lub wręcz przeciwnie. Ważne jest, aby znać podstawowe struktury danych i jak działają wewnętrznie.

2.2.1 Ciągi znaków

Ciągi znaków mogą być najbardziej przyjaznym typem danych w świecie programowania. Reprezentują tekst i zazwyczaj mogą być czytelne dla ludzi. Nie powinieneś używać ciągów znaków, gdy inny typ lepiej się nadaje, ale są one nieuniknione i wygodne. Korzystając z ciągów znaków, musisz znać kilka podstawowych faktów na ich temat, które nie są oczywiste od razu.

Mimo że są podobne do tablic pod względem użycia i struktury, ciągi znaków w .NET są niemodyfikowalne. Niemodyfikowalność oznacza, że zawartość struktury danych nie może być zmieniana po jej zainicjowaniu. Załóżmy, że chcemy połączyć imiona osób, aby utworzyć pojedynczy ciąg znaków oddzielony przecinkami i że wróciliśmy dwie dekady wstecz w czasie, więc nie ma lepszego sposobu na wykonanie tego zadania:

```
public static string JoinNames(string[] names) {
    string result = String.Empty;
    int lastIndex = names.Length - 1;
    for (int i = 0; i < lastIndex; i++) {
        result += names[i] + ", ";
    }
    result += names[lastIndex];
    return result;
}
```

Na pierwszy rzut oka może się wydawać, że mamy ciąg znaków o nazwie result i modyfikujemy ten sam ciąg znaków podczas wykonywania operacji, ale tak nie jest. Za każdym razem, gdy przypisujemy zmiennej result nową wartość, tworzony jest nowy ciąg znaków w pamięci. .NET musi określić długość nowego ciągu znaków, zaalokować nową pamięć na jego potrzeby, skopiować zawartość innych ciągów znaków do nowo zbudowanej pamięci i zwrócić ci go. To dość kosztowna operacja, a koszt rośnie w miarę wydłużania się ciągu znaków oraz śladu śmieci do zebrania.

W frameworku są narzędzia, które pozwalają unikać tego problemu. Nawet jeśli nie zależy ci na wydajności, te narzędzia są darmowe, więc nie musisz zmieniać swojej logiki ani starać się osiągnąć lepszą wydajność. Jednym z nich jest StringBuilder, z którym możesz pracować, aby budować ostateczny ciąg znaków i uzyskać go za pomocą wywołania ToString w jednym kroku:

```
public static string JoinNames(string[] names) {
    var builder = new StringBuilder();
    int lastIndex = names.Length - 1;
    for (int i = 0; i < lastIndex; i++) {
        builder.Append(names[i]);
        builder.Append(", ");
    }
    builder.Append(names[lastIndex]);
    return builder.ToString();
}
```

StringBuilder wewnętrznie używa kolejnych bloków pamięci zamiast realokować i kopiować za każdym razem, gdy musi zwiększyć rozmiar ciągu znaków. Dlatego zazwyczaj jest bardziej wydajny niż budowanie ciągu znaków od zera.

Oczywiście od dłuższego czasu dostępne jest idiomatyczne i znacznie krótsze rozwiązanie, ale twoje przypadki użycia nie zawsze muszą pokrywać się z tymi przypadkami:

```
String.Join(", ", names);
```

Łączenie ciągów znaków jest zazwyczaj akceptowalne podczas inicjalizacji ciągu, ponieważ obejmuje tylko jedną alokację bufora po obliczeniach wymaganej całkowitej długości. Na przykład jeśli masz funkcję, która łączy imię, drugie imię i nazwisko, oddzielając je spacją za pomocą operatora dodawania, tworzysz tylko jeden nowy ciąg znaków w jednym momencie, a nie w wielu krokach:

```
public string ConcatName(string firstName, string middleName, string lastName) {
    return firstName + " " + middleName + " " + lastName;
}
```

Może to wydawać się niewłaściwe, gdybyśmy zakładali, że `firstName + " "` stworzyłoby najpierw nowy ciąg znaków, a następnie tworzył nowy ciąg znaków z middleName i tak dalej, ale kompilator faktycznie zamienia to na pojedyncze wywołanie funkcji `String.Concat()`, która alokuje nowy bufor o długości sumy długości wszystkich ciągów znaków i zwraca go w jednym kroku. Dlatego jest to nadal szybkie. Ale gdy łączysz ciągi znaków w wielu krokach z pomiędzy klauzul if lub pętli, kompilator nie może tego zoptymalizować. Musisz wiedzieć, kiedy można łączyć ciągi znaków, a kiedy nie.

Mówiąc o tym, niemutowalność nie jest świętą pieczęcią, której nie można złamać. Istnieją sposoby na modyfikowanie ciągów znaków w miejscu lub innych niemodyfikowalnych strukturach, które głównie obejmują niebezpieczny kod i istoty astralne, ale zazwyczaj nie są one zalecane, ponieważ ciągi znaków są deduplikowane przez środowisko wykonawcze .NET, a niektóre z ich właściwości, takie jak hasze, są buforowane. Wewnętrzna implementacja silnie polega na właściwości niemutowalności.

Funkcje dla ciągów znaków domyślnie działają w kontekście bieżącej kultury (culture), co może być bolesne doświadczenie, gdy twoja aplikacja przestaje działać w innym kraju.

UWAGA

Kultura, znana również jako lokalizacja w niektórych językach programowania, to zestaw reguł dotyczących wykonywania operacji związanych z danym regionem, takich jak sortowanie ciągów znaków, wyświetlanie daty/czasu we właściwym formacie, układanie sztućców na stole i tak dalej. Bieżąca kultura to zazwyczaj to, co system operacyjny uznaje za aktualnie używaną.

Zrozumienie kultur może uczynić twoje operacje na ciągach znaków bezpieczniejszymi i szybszymi. Na przykład rozważ kod, w którym sprawdzamy, czy podana nazwa pliku ma rozszerzenie .gif:

```
public bool CzyGif(string nazwaPliku) {
   return nazwaPliku.ToLower().EndsWith(".gif");
}
```

Jesteśmy sprytni, widzisz: zamieniamy ciąg znaków na małe litery, więc radzimy sobie z przypadkiem, gdy rozszerzenie może być .GIF lub .Gif lub jakimkolwiek innym połączeniem wielkich i małych liter. Problem polega na tym, że nie wszystkie języki mają takie same zasady małych liter. W języku tureckim na przykład mała litera "I" to nie "i," ale "ı," znane jako kropkowe "i". Kod w tym przykładzie nie zadziałałby w Turcji i być może w niektórych innych krajach, takich jak Azerbejdżan. Zamieniając ciąg znaków na małe litery, faktycznie tworzymy nowy ciąg znaków, co, jak się nauczyliśmy, jest niewydajne.

.NET dostarcza wersje metod dla ciągów znaków, które są niezależne od kultury, takie jak ToLowerInvariant. Oferuje również przeciążenia tych samych metod, które przyjmują wartość StringComparison, która ma alternatywy w postaci invariant (niezależnego od kultury) i ordinal (porównanie znaków według kodów ASCII). Dlatego możemy napisać tę samą metodę w bezpieczniejszy i szybszy sposób:

```
public bool CzyGif(string nazwaPliku) {
   return nazwaPliku.EndsWith(".gif",
       StringComparison.OrdinalIgnoreCase);
}
```

Korzystając z tej metody, unikamy tworzenia nowego ciągu znaków, a także używamy szybszej i bezpiecznej metody porównywania ciągów znaków, która nie zależy od naszej bieżącej kultury i jej skomplikowanych zasad. Moglibyśmy użyć StringComparison.InvariantCultureIgnoreCase, ale w przeciwieństwie do porównania ordinal, dodaje ono kilka dodatkowych reguł tłumaczenia, takich jak traktowanie niemieckich znaków diakrytycznych lub grafemów ich odpowiednikami w alfabecie łacińskim (ß versus ss), co może powodować problemy z nazwami plików lub innymi identyfikatorami zasobów. Porównanie ordinal porównuje wartości znaków bezpośrednio, bez jakiejkolwiek translacji.

2.2.2 Tablica (Array)

Przejrzeliśmy już, jak wygląda tablica w pamięci. Tablice są praktyczne do przechowywania wielu elementów, których liczba nie będzie przekraczać rozmiaru tablicy. Są to struktury statyczne. Nie mogą rosnąć ani zmieniać swojego rozmiaru. Jeśli potrzebujesz większej tablicy, musisz stworzyć nową i skopiować zawartość starej do nowej. Istnieje kilka rzeczy, które musisz wiedzieć na temat tablic.

Tablice, w przeciwieństwie do ciągów znaków, są mutowalne. O to właśnie w nich chodzi. Możesz dowolnie manipulować ich zawartością. Tak naprawdę trudno jest uczynić je niemodyfikowalnymi, co czyni je słabymi kandydatami do interfejsów. Przyjrzyjmy się tej właściwości:

```
public string[] NazwyUżytkowników { get; }
```

Mimo że właściwość nie ma settera, jej typ wciąż jest tablicą, co sprawia, że jest mutowalna. Nic nie powstrzymuje cię przed wykonaniem poniższego kodu:

```
NazwyUżytkowników[0] = "root";
```

Co może komplikować sytuację, nawet gdy to tylko ty używasz tej klasy. Nie powinieneś pozwalać sobie na dokonywanie zmian w stanie, chyba że jest to absolutnie konieczne. Stan jest korzeniem wszelkiego zła, nie null. Im mniej stanów ma twoja aplikacja, tym mniej problemów będziesz mieć.

Stań na stanowisku, które ma najmniejszą funkcjonalność odpowiednią dla twojego celu. Jeśli potrzebujesz tylko przejść przez elementy sekwencyjnie, użyj `IEnumerable<T>`. Jeśli potrzebujesz także powtarzalnie dostępnego licznika, użyj `ICollection<T>`. Zauważ, że metoda rozszerzenia LINQ `.Count()` ma specjalny kod obsługi dla typów, które obsługują `IReadOnlyCollection<T>`, więc nawet jeśli używasz jej na `IEnumerable`, istnieje szansa, że może zwrócić wartość z pamięci podręcznej.

Tablice najlepiej sprawdzają się do użycia w lokalnym zakresie funkcji. Do każdego innego celu istnieje lepiej dostosowany typ lub interfejs do użycia obok `IEnumerable<T>`, takie jak `IReadOnlyCollection<T>`, `IReadOnlyList<T>` lub `ISet<T>`.

2.2.3 Lista (List)

Lista zachowuje się jak tablica, która może nieznacznie rosnąć, podobnie jak działa StringBuilder. Możliwe jest stosowanie list zamiast tablic prawie wszędzie, ale to spowoduje niepotrzebne obciążenie wydajności, ponieważ dostępy indeksowane są wywołaniami wirtualnymi w liście, podczas gdy tablica używa bezpośredniego dostępu.

Wiesz, programowanie obiektowe pozwala na używanie przyjemnej funkcji zwanej polimorfizmem, co oznacza, że obiekt może zachowywać się zgodnie z jego podstawową implementacją, bez zmiany jego interfejsu. Jeśli masz, na przykład, zmienną `a` z typem interfejsu `IOpenable`, `a.Open()` może otworzyć plik lub połączenie sieciowe, w zależności od typu obiektu przypisanego do niej. Osiąga się to przez przechowywanie odwołań do tabeli, która mapuje funkcje wirtualne do wywołania dla typu na początku obiektu, nazywanego tabelą metod wirtualnych lub vtable. W ten sposób, chociaż `Open` mapuje do tego samego wpisu w tabeli w każdym obiekcie tego samego typu, nie będziesz wiedzieć, dokąd prowadzi, dopóki nie sprawdzisz rzeczywistej wartości w tabeli.

Ponieważ nie wiemy, co dokładnie wywołujemy, są to wywołania wirtualne. Wywołanie wirtualne obejmuje dodatkowe wyszukiwanie w tabeli metod wirtualnych, więc jest nieco wolniejsze niż zwykłe wywołania funkcji. To może nie być problemem w przypadku kilku wywołań funkcji, ale gdy dzieje się to wewnątrz algorytmu, jego nadmiar może wzrastać w sposób wykładniczy. W rezultacie, jeśli twoja lista nie będzie rosła po inicjalizacji, możesz chcieć użyć tablicy zamiast listy w lokalnym zakresie.

Zwykle nie powinieneś zbytnio zastanawiać się nad tymi szczegółami. Ale gdy znasz różnicę, są sytuacje, w których tablica może być lepsza od listy.

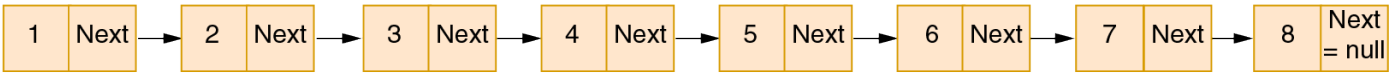
Listy są podobne do StringBuilder. Obydwa są strukturami danych o dynamicznie rosnącej liczbie elementów, ale listy są mniej wydajne w mechanice wzrostu. Kiedy lista stwierdza, że potrzebuje wzrosnąć, alokuje nową tablicę o większym rozmiarze i kopiuje do niej istniejącą zawartość. Z drugiej strony StringBuilder trzyma ze sobą połączone fragmenty pamięci, co nie wymaga operacji kopiowania. Obszar bufora dla list rośnie, gdy osiągnięty zostanie limit bufora, ale rozmiar nowego bufora podwaja się za każdym razem, co oznacza, że potrzeba wzrostu zmniejsza się w miarę czasu. To jest przykład, w którym użycie konkretnej klasy do danego zadania jest bardziej wydajne niż użycie klasy ogólnej.

Można także uzyskać doskonałą wydajność z list, określając ich pojemność. Jeśli nie określisz pojemności dla listy, zacznie ona od pustej tablicy, następnie zwiększy swoją pojemność do kilku elementów. Po osiągnięciu limitu, podwaja swoją pojemność, gdy zostanie zapełniona. Jeśli ustawisz pojemność podczas tworzenia listy, unikniesz niepotrzebnych operacji wzrostu i kopiowania. Pamiętaj o tym, gdy już wiesz, ile maksymalnie elementów będzie mieć lista.

Mając to na uwadze, nie rób nawyku określania pojemności listy bez znanego powodu. Może to spowodować zbędny nadmiar pamięci, który może się akumulować. Używaj tych funkcji świadomie.

2.2.4 Lista dwukierunkowa (Linked list)

Listy dwukierunkowe to listy, w których elementy nie są ułożone kolejno w pamięci, ale każdy element wskazuje na adres następnego elementu. Są użyteczne ze względu na ich wydajność przy operacjach wstawiania i usuwania, które mają złożoność O(1). Nie można uzyskać dostępu do pojedynczych elementów za pomocą indeksu, ponieważ mogą być one przechowywane w dowolnym miejscu w pamięci, co nie jest możliwe do obliczenia. Jednak jeśli głównie uzyskujesz dostęp do początku lub końca listy, lub jeśli potrzebujesz tylko przeglądać elementy, może być równie szybka. W przeciwnym razie sprawdzanie, czy element istnieje w liście dwukierunkowej, ma złożoność O(N), podobnie jak w przypadku tablic i list. Rysunek 2.5 przedstawia przykładowy układ listy dwukierunkowej.



To nie oznacza, że lista dwukierunkowa zawsze jest szybsza niż zwykła lista. Indywidualne przydzielanie pamięci dla każdego elementu zamiast alokacji całego bloku pamięci na raz oraz dodatkowe odwołania do referencji mogą wpływać negatywnie na wydajność.

Lista dwukierunkowa może być potrzebna, gdy potrzebujesz struktury kolejki lub stosu, ale .NET ma to już wbudowane. Idealnie rzecz biorąc, chyba że interesujesz się programowaniem systemowym, nie powinieneś potrzebować używać listy dwukierunkowej w swojej codziennej pracy, z wyjątkiem sytuacji podczas rozmów kwalifikacyjnych o pracę. Niestety, osoby przeprowadzające rozmowy kwalifikacyjne uwielbiają zagadki związane z listami dwukierunkowymi, dlatego warto z nimi się zapoznać.

NIE, NIE BĘDZIESZ ODWRACAĆ LISTY DWUKIERUNKOWEJ

Odpowiadanie na pytania związane z kodowaniem podczas rozmów kwalifikacyjnych to rytuał dla stanowisk związanych z rozwojem oprogramowania. Większość pytań dotyczy również pewnych struktur danych i algorytmów. Listy dwukierunkowe są częścią tego zestawu, więc istnieje szansa, że ktoś poprosi cię o odwrócenie listy dwukierunkowej lub odwrócenie drzewa binarnego.

Prawdopodobnie nigdy nie będziesz wykonywać tych zadań w swojej rzeczywistej pracy, ale aby oddać sprawiedliwość osobie przeprowadzającej rozmowę kwalifikacyjną, testują one twoją wiedzę z zakresu struktur danych i algorytmów, aby po prostu upewnić się, że wiesz, co robisz. Chcą sprawdzić, czy jesteś w stanie podjąć odpowiednie decyzje, gdy pojawi się potrzeba użycia odpowiedniej struktury danych we właściwym miejscu. Testują również twoją umiejętność myślenia analitycznego i rozwiązywania problemów, dlatego ważne jest, abyś myślał na głos i dzielił się swoim sposobem myślenia z osobą przeprowadzającą rozmowę.

Nie zawsze musisz rozwiązywać dane pytanie. Osoba przeprowadzająca rozmowę zwykle szuka kogoś, kto jest pasjonatem i ma wiedzę na temat pewnych podstawowych koncepcji oraz potrafi się odnaleźć, nawet jeśli czasem się zagubi.

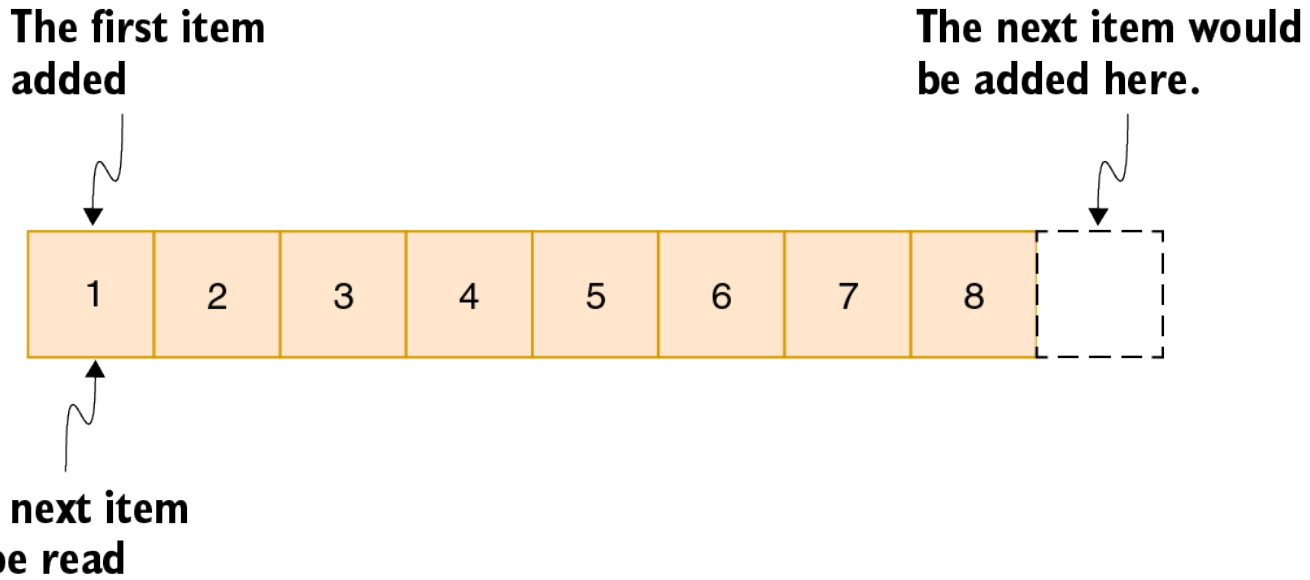
Na przykład ja zwykle dodawałem dodatkowy etap do moich pytań podczas rozmów kwalifikacyjnych w Microsoft, w którym kandydat musiał znaleźć błędy w swoim kodzie. To sprawiało, że czuli się lepiej, ponieważ mieli świadomość, że błędy są oczekiwane i nie byli oceniani na podstawie tego, jak bezbłędny jest ich kod, ale na podstawie tego, jak potrafią zidentyfikować błędy.

Rozmowy kwalifikacyjne nie polegają tylko na znalezieniu odpowiedniej osoby, ale także na znalezieniu kogoś, z kim przyjemnie będzie się pracować. Ważne jest, abyś był ciekawski, pełen pasji, wytrwały i łatwy w obyciu, kto naprawdę może pomóc im w wykonywaniu ich zadań.

Listy dwukierunkowe były bardziej popularne w czasach prehistorycznych programowania, ponieważ priorytetem było efektywne zarządzanie pamięcią. Nie mogliśmy pozwolić sobie na alokację kilobajtów pamięci tylko dlatego, że nasza lista potrzebowała rosnąć. Musieliśmy trzymać się szczelnej gospodarki pamięcią. Lista dwukierunkowa była idealną strukturą danych dla tego celu. Są nadal często używane w jądrach systemów operacyjnych ze względu na ich nieodpartą charakterystykę O(1) dla operacji wstawiania i usuwania.

2.2.5 Queue

Kolejka to struktura danych, która reprezentuje najbardziej podstawową formę cywilizacji. Pozwala odczytywać elementy z listy w kolejności ich dodawania. Kolejka może być po prostu tablicą, pod warunkiem, że trzymasz osobne miejsca na odczytanie następnego elementu i wstawienie nowego. Jeśli dodalibyśmy rosnące liczby do kolejki, wyglądałaby to mniej więcej tak jak na rysunku 2.6.



Bufor klawiatury na komputerach PC w czasach MS-DOS używał prostego tablicy bajtów do przechowywania naciśnięć klawiszy. Bufor zapobiegał utracie naciśniętych klawiszy ze względu na wolne lub niewydolne oprogramowanie. Gdy bufor był pełny, BIOS wydawał sygnał dźwiękowy, abyśmy wiedzieli, że nasze naciśnięcia klawiszy nie są już rejestrowane. Na szczęście w .NET istnieje gotowa klasa Queue, którą możemy używać bez konieczności martwienia się o szczegóły implementacji i wydajność.

2.2.6 Directory

Słowniki, nazywane również hashmapami lub czasem strukturami klucz/wartość, należą do najbardziej przydatnych i najczęściej używanych struktur danych. Zwykle nie zastanawiamy się nad ich zdolnościami, traktujemy je jak coś oczywistego. Słownik to pojemnik, który może przechowywać klucz i wartość. Później może odnaleźć wartość związana z kluczem w czasie stałym, czyli O(1). Oznacza to, że są one niezwykle szybkie w pobieraniu danych. Dlaczego są tak szybkie? Gdzie tkwi magia?

Magia tkwi w słowie "hash" (hasz). Haszowanie to proces generowania pojedynczej liczby na podstawie dowolnych danych. Wygenerowana liczba musi być deterministyczna, co oznacza, że dla tych samych danych zawsze będzie generować tę samą liczbę, ale nie musi być unikalna. Istnieje wiele różnych sposobów obliczania wartości hasza. Logika haszowania obiektu znajduje się w implementacji GetHashCode.

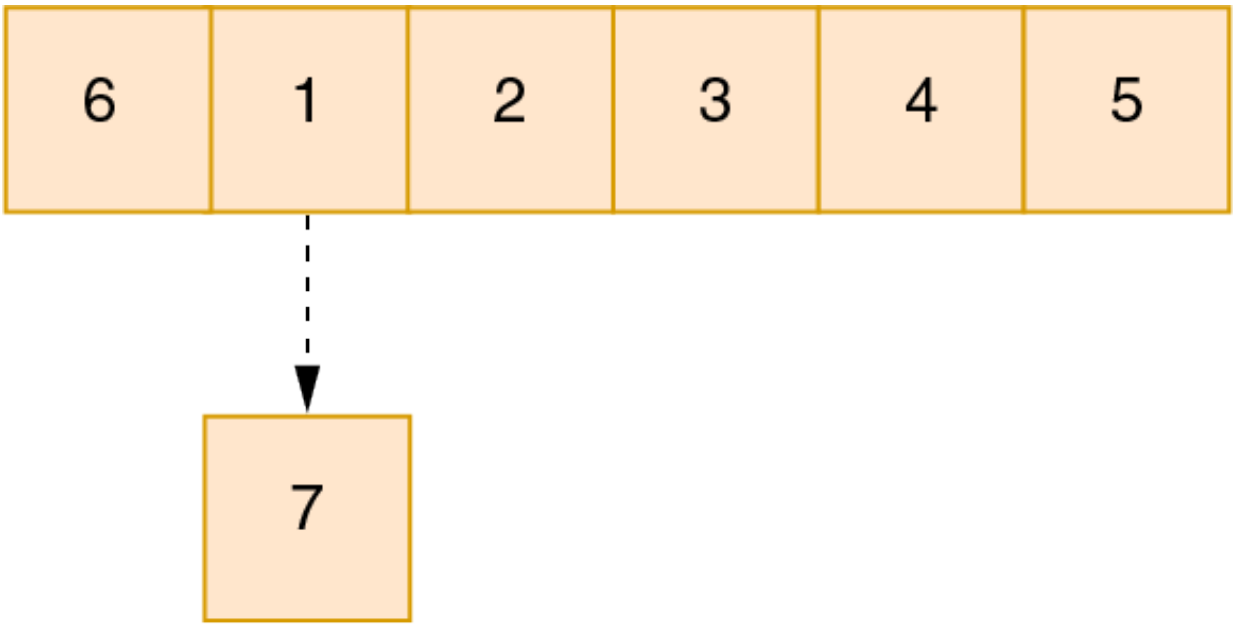
Hasze są fajne, ponieważ za każdym razem dostajesz tę samą wartość, co pozwala na ich używanie do przeszukiwania. Na przykład mając tablicę wszystkich możliwych wartości hasz, można je przeszukać, używając indeksów tablicy. Ale taka tablica zajęłaby około 16 gigabajtów dla każdego utworzonego słownika, ponieważ każda liczba całkowita zajmuje cztery bajty i może mieć około czterech miliardów możliwych wartości.

Słowniki przydzielają znacznie mniejszą tablicę i polegają na równomiernym rozłożeniu wartości hasza. Zamiast przeszukiwania wartości hasza, szukają "wartość hasza modulo długość tablicy". Załóżmy, że słownik z kluczami całkowitymi przydziela tablicę sześciu elementów, aby je indeksować, a metoda GetHashCode() dla liczby całkowitej zwraca po prostu jej wartość. To oznacza, że nasza formuła do określenia, gdzie element zostanie zmapowany, byłaby po prostu wartość % 6, ponieważ indeksy tablicy zaczynają się od zera. Tablica liczb od 1 do 6 zostałaby rozłożona, jak pokazano na rysunku 2.7.

Co się dzieje, gdy mamy więcej elementów niż pojemność słownika? Na pewno będą się nakładać, więc słowniki przechowują te nakładające się elementy w dynamicznie rosnącej liście. Jeśli przechowujemy elementy z kluczami od 1 do 7, tablica wyglądałaby tak, jak to pokazano na rysunku 2.8.

Rys 2.7

Rys 2.8

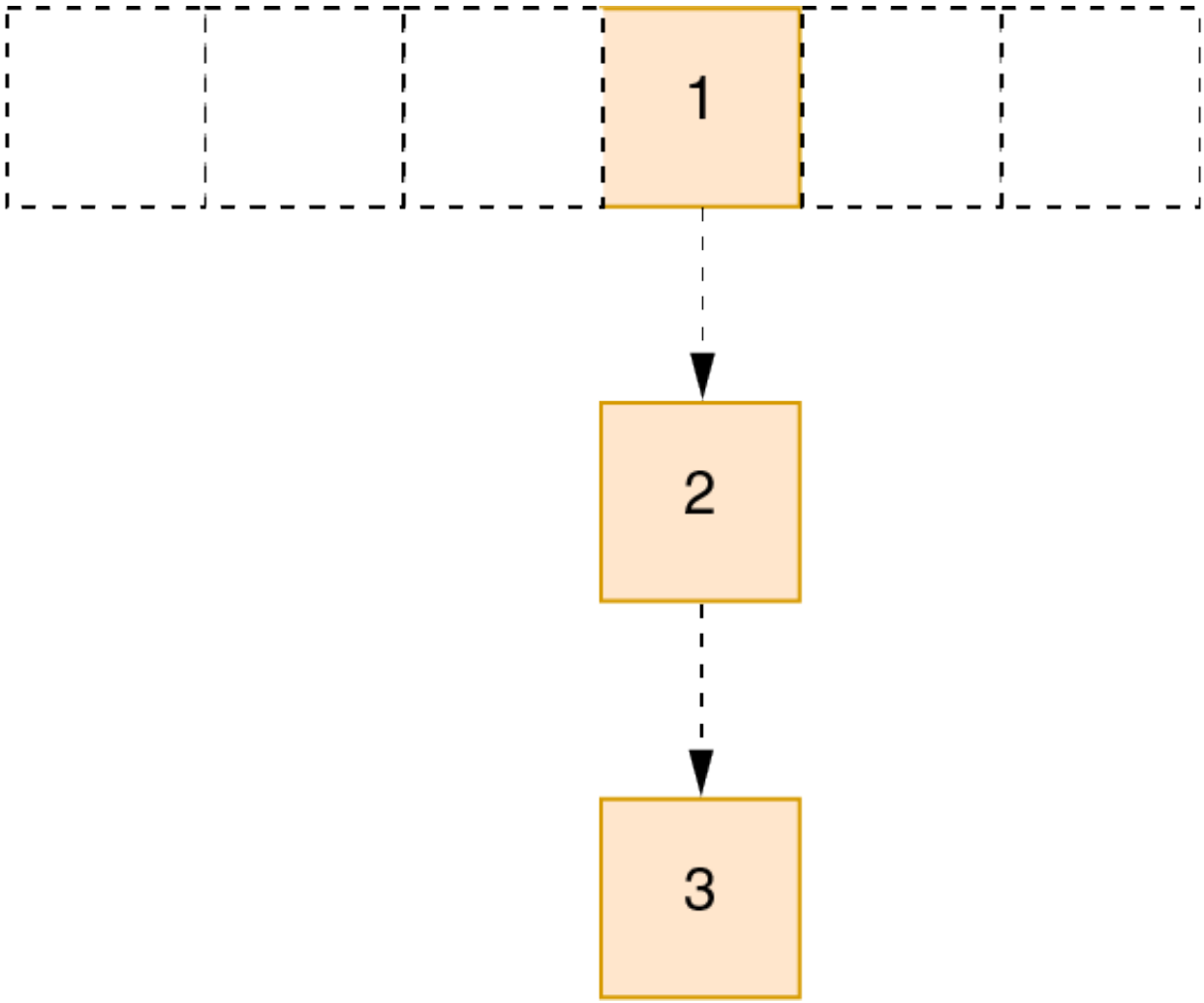


Dlaczego omawiam ten temat? Ponieważ wydajność wyszukiwania klucza w słowniku wynosi zazwyczaj O(1), podczas gdy nadmiar wyszukiwania w liście połączonej wynosi O(N). Oznacza to, że w miarę wzrostu liczby nakładających się elementów, wydajność wyszukiwania będzie się pogarszać. Jeśli na przykład miałbyś funkcję GetHashCode, która zawsze zwracałaby 4:

```
public override int GetHashCode() {
   return 4; // wybrane za pomocą uczciwego rzutu kością
}
```

Oznacza to, że wewnętrzna struktura słownika przy dodawaniu do niego elementów przypominałaby schemat przedstawiony na rysunku 2.9.

Rys 2.9



Słownik nie jest lepszy od listy połączonej, jeśli masz złe wartości funkcji skrótu (GetHashCode). Może nawet działać gorzej ze względu na dodatkowe działania, jakie musi wykonać słownik, aby obsłużyć te elementy. To przynosi nas do najważniejszego punktu: Twoja funkcja GetHashCode musi być jak najbardziej unikalna. Jeśli masz wiele nakładających się elementów, Twoje słowniki będą cierpieć, cierpiący słownik sprawi, że Twoja aplikacja będzie cierpieć, a cierpiąca aplikacja sprawi, że cała firma będzie cierpieć. W końcu to Ty będziesz cierpieć. Przez brak gwoździa, cała królestwo zostało utracone.

Czasami musisz połączyć wartości z wielu właściwości w klasie, aby obliczyć unikalną wartość skrótu. Na przykład nazwy repozytoriów na GitHubie są unikalne dla każdego użytkownika. Oznacza to, że dowolny użytkownik może mieć repozytorium o tej samej nazwie, a sama nazwa repozytorium nie jest wystarczająca, aby je uczynić unikalnym. Jeśli użyjesz tylko samej nazwy, spowoduje to większe kolizje. Oznacza to, że musisz połączyć wartości skrótu. Podobnie, jeśli nasza witryna ma unikalne wartości dla różnych tematów, mielibyśmy ten sam problem.

Aby skutecznie łączyć wartości skrótu, musisz znać ich zakresy i zajmować się ich reprezentacją bitową. Jeśli po prostu używasz operatora jak dodawanie lub proste operacje OR/XOR, nadal możesz otrzymać znacznie więcej kolizji, niż się tego spodziewasz. Musisz również używać przesunięć bitowych. Prawidłowa funkcja GetHashCode będzie używała operacji bitowych, aby uzyskać równomierne rozłożenie na pełnych 32 bitach liczby całkowitej.

Kod takiej operacji może wyglądać jak sceny hakerskie z tandetnego filmu o hakerach. Jest enigmatyczny i trudny do zrozumienia nawet dla kogoś, kto jest zaznajomiony z tym pojęciem. W zasadzie obracamy jedną z 32-bitowych liczb całkowitych o 16 bitów, więc jej najniższe bajty są przesuwane ku środkowi i XORujemy ("^") tę wartość z inną 32-bitową liczbą całkowitą, co znacznie zmniejsza szanse na kolizje. Wygląda to tak— przerażająco:

```
public override int GetHashCode() {
   return (int)(((TopicId & 0xFFFF) << 16)
       ^ (TopicId & 0xFFFF0000 >> 16)
       ^ PostId);
}
```

Na szczęście, wraz z pojawieniem się .NET Core i .NET 5, łączenie wartości skrótu w sposób minimalizujący kolizje zostało zasłonięte za pomocą klasy HashCode. Aby połączyć dwie wartości, wystarczy zrobić tak:

```
public override int GetHashCode() {
   return HashCode.Combine(TopicId, PostId);
}
```

Kody skrótu są używane nie tylko jako klucze słowników, ale także w innych strukturach danych, takich jak zbiory. Ponieważ jest znacznie łatwiej napisać odpowiednią funkcję GetHashCode za pomocą funkcji pomocniczych, nie masz wymówki, aby tego nie zrobić. Bądź czujny w tym zakresie.

Kiedy nie powinieneś używać słownika? Jeśli potrzebujesz przejść po parach klucz-wartość sekwencyjnie, słownik nie oferuje żadnych korzyści. W rzeczywistości może nawet zaszkodzić wydajności. Warto wówczas rozważyć użycie List<KeyValuePair<K, V>>, aby uniknąć niepotrzebnego narzutu.

Przepraszam za moje niedoprecyzowanie. Oto tłumaczenie nagłówka:

2.2.7 HashSet

HashSet (Zbiór) to struktura danych podobna do tablicy lub listy, z tą różnicą, że może zawierać tylko unikalne wartości. Jej przewagą nad tablicami lub listami jest to, że ma stały czas dostępu O(1) podobnie jak klucze w słowniku, dzięki mapom opartym na funkcjach skrótu, o których właśnie rozmawialiśmy. Oznacza to, że jeśli musisz często sprawdzać, czy dana tablica lub lista zawiera określony element, używanie zbioru może być szybsze. W .NET jest to nazywane HashSet i jest dostępne za darmo.

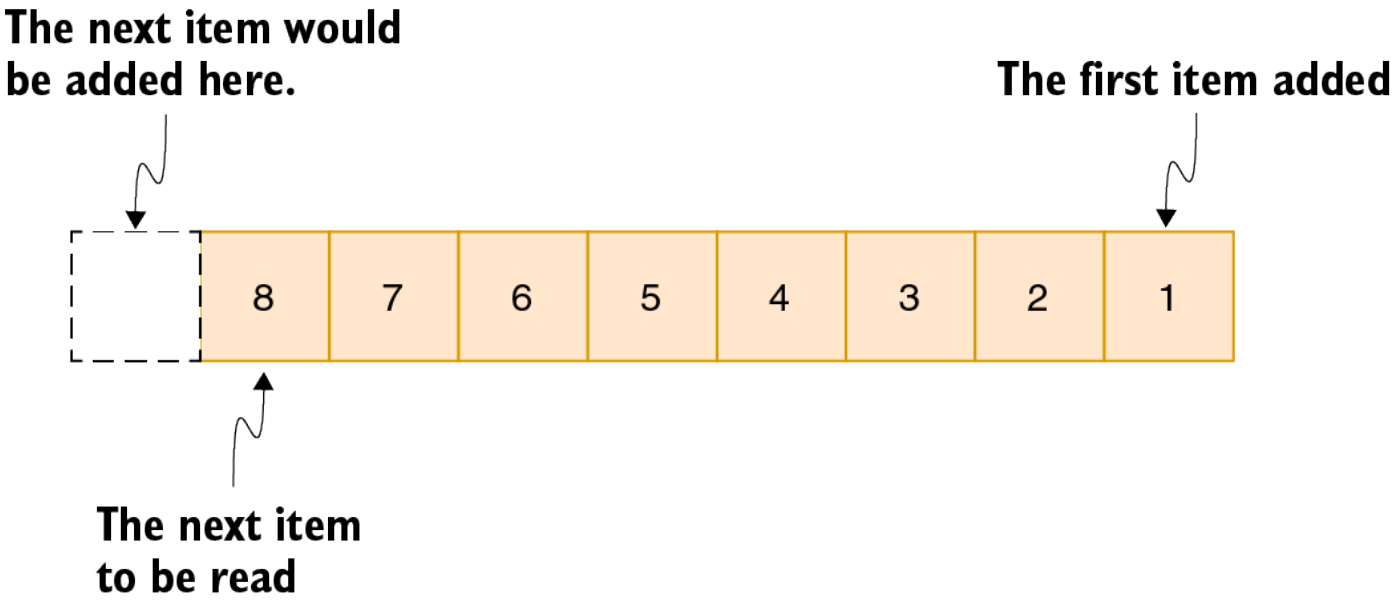
Ponieważ HashSet jest szybki podczas wyszukiwania i wstawiania, nadaje się również do operacji na przecięciach i sumach zbiorów. Nawet dostarcza metody, które umożliwiają korzystanie z tych funkcji. Aby czerpać korzyści z tych możliwości, musisz zwrócić uwagę na implementacje metody GetHashCode().

2.2.8 Stos

Stosy to kolejki LIFO (Last In First Out). Są użyteczne, gdy chcesz zapisać stan i przywrócić go w odwrotnej kolejności, w jakiej został zapisany. Kiedy odwiedzasz urząd Departamentu Komunikacji Samochodowej (DMV) w rzeczywistości, czasami musisz korzystać ze stosu. Najpierw podchodzisz do stanowiska 5, a pracownik przy stanowisku sprawdza twoje dokumenty i widzi, że brakuje ci płatności, więc odsyła cię do stanowiska 13. Pracownik przy stanowisku 13 widzi, że brakuje ci zdjęcia w dokumentach i odsyła cię do kolejnego stanowiska, tym razem do stanowiska 47, aby zrobić zdjęcie. Następnie musisz wrócić do stanowiska 13, gdzie odbierasz potwierdzenie płatności, a następnie wrócić do stanowiska 5, aby odebrać prawo jazdy. Lista stanowisk i sposób ich przetwarzania w kolejności (LIFO) to operacja przypominająca stos, która zazwyczaj jest bardziej wydajna niż w przypadku DMV.

Stos można reprezentować za pomocą tablicy. Różnica polega na tym, gdzie umieszczasz nowe elementy i skąd odczytujesz następny element. Gdybyśmy zbudowali stos, dodając liczby w kolejności rosnącej, wyglądałby jak na rysunku 2.10.

Rys 2.10



Dodawanie do stosu jest zwykle nazywane "pushing", a odczytywanie następnej wartości ze stosu nazywane jest "popping". Stosy są przydatne do cofania się po wykonanych krokach. Być może już jesteś zaznajomiony ze stosu wywołań, ponieważ pokazuje on nie tylko miejsce wystąpienia wyjątku, ale także ścieżkę wykonania, którą podążył program. Funkcje wiedzą, gdzie wrócić po zakończeniu wykonania, używając stosu. Przed wywołaniem funkcji, adres powrotu jest dodawany na stos. Gdy funkcja chce wrócić do swojego wywołującego, odczytywany jest ostatni adres dodany na stos i CPU kontynuuje wykonywanie kodu od tego adresu.

2.2.9 Stos wywołań

Stos wywołań to struktura danych, w której funkcje przechowują adresy powrotu, dzięki czemu wywołane funkcje wiedzą, gdzie wrócić po zakończeniu działania. Istnieje jeden stos wywołań na wątek.

Każda aplikacja działa w jednym lub więcej osobnych procesach. Procesy pozwalają na izolację pamięci i zasobów. Każdy proces ma jeden lub więcej wątków. Wątek jest jednostką wykonawczą. Wszystkie wątki działają równolegle na systemie operacyjnym, stąd nazwa wielowątkowość. Nawet jeśli masz procesor czterordzeniowy, system operacyjny może równocześnie uruchamiać tysiące wątków. Możliwe jest to, ponieważ większość wątków większość czasu czeka na zakończenie jakiegoś procesu, więc można zapełnić ich miejsce innym wątkiem, co daje wrażenie działania wszystkich wątków równocześnie. Dzięki temu nawet na pojedynczym procesorze możliwe jest wielozadaniowość.

Kiedyś proces był zarówno kontenerem dla zasobów aplikacji, jak i jednostką wykonawczą w starszych systemach UNIX. Chociaż podejście to było proste i eleganckie, powodowało problemy, takie jak procesy zombie. Wątki są bardziej lekkie i nie mają takiego problemu, ponieważ są związane z czasem życia wykonania.

Każdy wątek ma swój własny stos wywołań: stałą ilość pamięci. Tradycyjnie stos rośnie od góry do dołu w przestrzeni pamięci procesu, gdzie góra oznacza koniec przestrzeni pamięci, a dół oznacza nasz słynny wskaźnik zerowy: adres zero. Dodanie elementu na stos wywołań oznacza umieszczenie go tam i zmniejszenie wskaźnika stosu.

Jak każda dobra rzecz, stos ma swój koniec. Ma stały rozmiar, więc gdy przekroczy tę wielkość, CPU generuje wyjątek StackOverflowException, z którym będziesz się spotykać w swojej karierze, gdy przypadkowo wywołasz funkcję od siebie samej. Stos jest dość duży, więc zazwyczaj nie martwisz się o osiągnięcie limitu w normalnych przypadkach.

Stos wywołań przechowuje nie tylko adresy powrotu, ale także parametry funkcji i zmienne lokalne. Ponieważ zmienne lokalne zajmują tak mało pamięci, stos jest bardzo efektywny dla nich, ponieważ nie wymaga dodatkowych kroków zarządzania pamięcią, takich jak alokacja i dealokacja. Stos jest szybki, ale ma stały rozmiar i ma taki sam czas życia jak funkcja go używająca. Po zwróceniu się z funkcji, przestrzeń stosu jest oddawana. Dlatego idealne jest przechowywanie w nim małej ilości danych lokalnych. W związku z tym zarządzane środowiska wykonawcze, takie jak C# czy Java, nie przechowują danych klas w stosie, a jedynie ich odwołania.

To jest kolejny powód, dla którego typy wartości mogą mieć lepszą wydajność w pewnych przypadkach w porównaniu do typów referencyjnych. Typy wartości istnieją na stosie tylko wtedy, gdy są deklarowane lokalnie, chociaż są przekazywane przez kopiowanie.

2.3 O co chodzi z typami danych?

Programiści często biorą typy danych za pewnik. Niektórzy nawet twierdzą, że programiści są szybsi w językach o typach dynamicznych, takich jak JavaScript czy Python, ponieważ nie muszą przejmować się skomplikowanymi detalami, jak na przykład określanie typu każdej zmiennej.

UWAGA

Typ dynamiczny oznacza, że typy danych zmiennych lub elementów klas w języku programowania mogą się zmieniać podczas działania programu. W języku JavaScript możesz przypisać zmienną najpierw jako string, a później jako liczbę całkowitą, ponieważ jest to język o typach dynamicznych. Języki o typach statycznych, takie jak C# czy Swift, nie pozwoliłyby na takie działanie. Omówimy to dokładniej później.

Tak, określanie typów dla każdej zmiennej, każdego parametru i każdego elementu w kodzie może być uciążliwe, ale musisz podchodzić do tego w sposób całościowy, jeśli chcesz być szybszy. Bycie szybkim nie polega tylko na pisaniu kodu, ale także na jego utrzymaniu. Mogą być przypadki, kiedy nie musisz martwić się o utrzymanie, na przykład gdy właśnie zostałeś zwolniony i nie obchodzi cię to. Poza takimi sytuacjami, rozwijanie oprogramowania to maraton, a nie sprint.

Szybkie wykrywanie błędów to jedna z najlepszych praktyk w programowaniu. Typy danych stanowią jedną z najwcześniejszych obron przed tarciem w trakcie tworzenia kodu. Typy pozwalają ci na szybkie wykrywanie błędów i naprawę ich, zanim staną się problemem. Oprócz oczywistej korzyści z unikania przypadkowego mylenia stringa z liczbą całkowitą, typy mogą działać na twoją korzyść także w inny sposób.

2.3.1 Silne związki z typami

Większość języków programowania posiada typy danych. Nawet najprostsze języki programowania, takie jak BASIC, miały typy danych: stringi i liczby całkowite. Niektóre ich dialekty miały nawet liczby rzeczywiste. Istnieją także języki określane jako "beztroskie związki" (typeless), takie jak Tcl, REXX, Forth, i tak dalej. W tych językach operacje są wykonywane tylko na jednym typie danych, zwykle na stringach lub liczbach całkowitych. Brak konieczności myślenia o typach danych sprawia, że programowanie jest wygodniejsze, ale prowadzi do wolniejszego i bardziej podatnego na błędy kodu.

Typy danych to podstawowe narzędzie do zapewnienia poprawności kodu, więc zrozumienie systemu typów może znacznie pomóc w staniu się produktywnym programistą. Sposób, w jaki języki programowania implementują typy danych, jest silnie powiązany z tym, czy są one interpretowane czy kompilowane:

- Interpretowane języki programowania, takie jak Python czy JavaScript, pozwalają na natychmiastowe uruchamianie kodu z pliku tekstowego bez konieczności etapu kompilacji. Ze względu na swoją natychmiastową naturę, zmienne mają zwykle elastyczne typy: możesz przypisać string do zmiennej, która wcześniej przechowywała liczbę całkowitą, a nawet dodawać do siebie stringi i liczby. Są one zwykle określane jako języki o typach dynamicznych ze względu na sposób, w jaki implementują typy danych. W językach interpretowanych kod można pisać znacznie szybciej, ponieważ nie trzeba deklarować typów.
- Kompilowane języki programowania są zwykle bardziej restrykcyjne. Jak restrykcyjne są, zależy od tego, jak wiele trudności chce ci sprawić projektant języka. Na przykład język Rust może być uważany za niemieckie inżynieria wśród języków programowania: bardzo rygorystyczny, perfekcjonistyczny i dlatego wolny od błędów. Język C także można porównać do niemieckiej inżynierii, ale bardziej jak Volkswagen: pozwala ci łamać zasady i później płacić za to cenę. Oba języki są statycznie typowane. Po zadeklarowaniu zmiennej jej typ nie może się zmienić, ale Rust jest uważany za silnie typowany, podobnie jak C#, podczas gdy C uznawany jest za słabo typowany.

"Silnie typowane" i "słabo typowane" oznaczają, jak elastycznie język pozwala na przypisywanie różnych typów zmiennych do siebie nawzajem. W tym sensie C jest bardziej elastyczny: możesz przypisać wskaźnik do liczby całkowitej i odwrotnie bez problemów. Z drugiej strony C# jest bardziej restrykcyjny: wskaźniki/referencje i liczby całkowite to niezgodne typy. Tabela 2.2 pokazuje, do jakich kategorii należą różne języki programowania.

	Statycznie Typowane	Dynamicznie Typowane
	Typ zmiennej nie może się zmieniać w trakcie działania programu.	Typ zmiennej może się zmieniać w trakcie działania programu.
Silnie Typowane		
Różne typy nie mogą być zamienione nawzajem.	C#, Java, Rust, Swift, Kotlin,TypeScript, C++	Python, Ruby, Lisp
Słabo Typowane		
Różne typy mogą byćzamienione nawzajem.	Visual Basic, C	JavaScript, VBScript

Ścisłe języki programowania mogą być frustrujące. Języki takie jak Rust mogą nawet sprawić, że zaczniemy zastanawiać się nad sensem życia i istnieniem we wszechświecie. Deklarowanie typów i jawne ich konwertowanie, gdy jest to potrzebne, może wydawać się uciążliwe i pełne biurokracji. Przykładowo, w języku JavaScript nie musisz deklarować typów każdej zmiennej, argumentu czy elementu klasy. Dlaczego więc obciążamy się jawnymi deklaracjami typów, skoro wiele języków programowania może działać bez nich?

Odpowiedź jest prosta: typy mogą pomóc nam pisać kod, który jest bezpieczniejszy, szybszy i łatwiejszy do utrzymania. Dzięki nim odzyskujemy czas, który straciliśmy na deklarowaniu typów zmiennych, adnotacji naszych klas. Zyskujemy go dzięki mniejszej liczbie błędów do debugowania i mniejszej ilości problemów z wydajnością.

Oprócz oczywistych korzyści wynikających z korzystania z typów, mają one również subtelne zalety. Przeanalizujmy je.

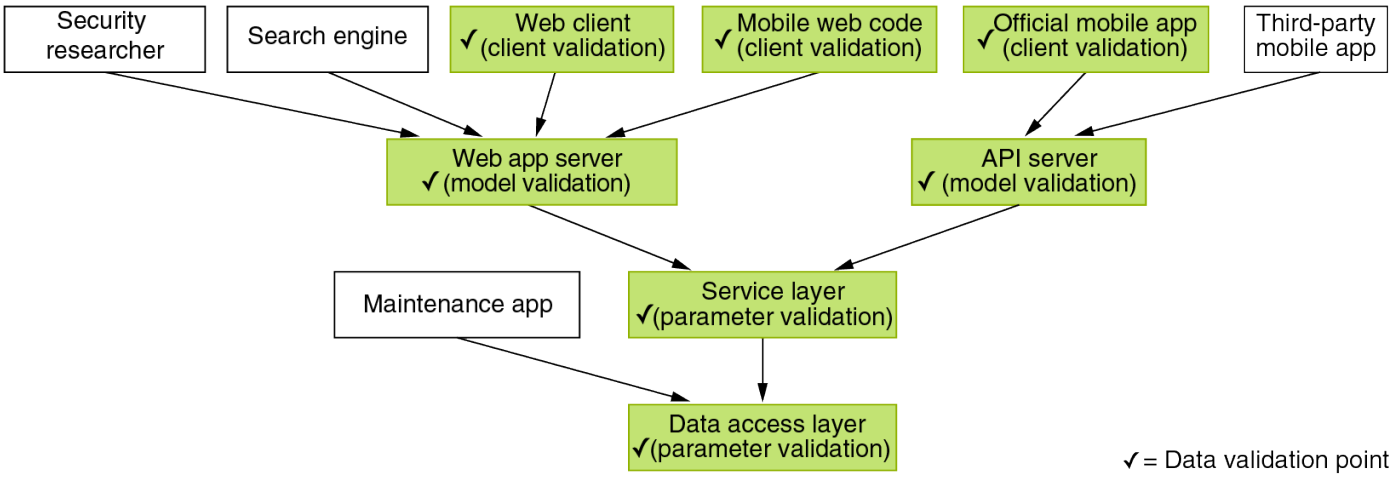
2.3.2 Dowód poprawności

Dowód poprawności to jedna z mniej znanych korzyści płynących z posiadania predefiniowanych typów. Załóżmy, że tworzysz platformę mikroblogowania, która pozwala na określoną liczbę znaków w każdym poście, i w zamian nie jesteś oceniany za lenistwo w pisaniu czegoś dłuższego niż zdanie. Na tej hipotetycznej platformie mikroblogowania możesz wspominać innych użytkowników w poście za pomocą prefiksu @ oraz oznaczać inne posty za pomocą prefiksu #, po którym następuje identyfikator posta. Możesz nawet odnaleźć post, wpisując jego identyfikator w polu wyszukiwania. Jeśli wpiszesz nazwę użytkownika z prefiksem @ w polu wyszukiwania, pojawi się profil tego użytkownika.

Wejście użytkownika wiąże się z nowym zestawem problemów z walidacją. Co się stanie, jeśli użytkownik wpisze litery po prefiksie #? Co jeśli wpiszą dłuższą liczbę, niż jest dozwolona? Może się wydawać, że te scenariusze same się rozwiążą, ale zwykle aplikacja się zawiesi, ponieważ w jakimś miejscu ścieżki kodu coś, co nie oczekuje na nieprawidłowe dane wejściowe, spowoduje wystąpienie wyjątku. To jest najgorsze możliwe doświadczenie dla użytkownika: nie wiedzą, co poszło nie tak, i nie wiedzą nawet, co mają zrobić

dalej. Może to nawet stać się problemem związanym z bezpieczeństwem, jeśli wyświetlasz to wejście bez jego oczyszczenia.

Walidacja danych nie zapewnia dowodu poprawności w całym kodzie. Możesz zwalidować dane wejściowe po stronie klienta, ale ktoś, na przykład aplikacja stron trzecich, może wysłać żądanie bez walidacji. Możesz zwalidować kod obsługujący żądania internetowe, ale inna twoja aplikacja, na przykład kod API, może wywołać kod usługi bez koniecznej walidacji. Podobnie, kod bazy danych może otrzymywać żądania z różnych źródeł, takich jak warstwa usługi i zadanie konserwacyjne, dlatego musisz upewnić się, że wstawiasz odpowiednie rekordy do bazy danych. Rysunek 2.11 przedstawia punkty, w których aplikacja może potrzebować zwalidować dane wejściowe.



To w rezultacie może doprowadzić do tego, że będziesz musiał zweryfikować dane wejściowe w wielu miejscach w kodzie, a ponadto musisz być konsekwentny w walidacji. Nie chcesz mieć posta z identyfikatorem -1 lub profilu użytkownika o nazwie ' OR 1=1— (co jest podstawowym atakiem SQL injection, który omówimy w rozdziale dotyczącym bezpieczeństwa).

Typy mogą zapewnić dowód poprawności. Zamiast przekazywać liczby całkowite jako identyfikatory postów na blogu lub ciągi znaków jako nazwy użytkowników, możesz używać klas lub struktur, które zwalidują swoje dane wejściowe podczas konstrukcji, co sprawia, że niemożliwe jest ich zawarcie nieprawidłowej wartości. Jest to proste, ale potężne narzędzie. Każda funkcja, która otrzymuje identyfikator posta jako parametr, prosi o klasę PostId zamiast liczby całkowitej. Pozwala to przenieść dowód poprawności po pierwszej walidacji w konstruktorze. Jeśli jest to liczba całkowita, wymaga walidacji; jeśli jest to PostId, został już zweryfikowany. Nie ma potrzeby sprawdzania jego zawartości, ponieważ nie ma możliwości jego utworzenia bez walidacji, jak widać w poniższym fragmencie kodu. Jedynym sposobem na utworzenie instancji PostId w fragmencie kodu jest wywołanie jego konstruktora, który zwaliduje jego wartość i w razie niepowodzenia zgłosi wyjątek. Oznacza to, że niemożliwe jest posiadanie niewłaściwej instancji PostId:

```
1  public class PostId
2  {
3      public int Value { get; private set; }
4      public PostId(int id) {
5          if (id <= 0)  {
6              throw new ArgumentOutOfRangeException(nameof(id));
7          }
8          Value = id;
9      }
10 }
```

STYL KODU W PRZYKŁADACH

Rozmieszczenie nawiasów klamrowych to drugi najbardziej kontrowersyjny temat w programowaniu, który jeszcze nie został rozstrzygnięty jednomyślnie, zaraz po kwestii używania tabulatorów czy spacji. Osobiście preferuję styl Allman dla większości języków podobnych do C, zwłaszcza dla C# i Swift. W stylu Allman każdy znak nawiasu klamrowego znajduje się w swojej własnej linii. Oficjalnie Swift zaleca stosowanie stylu 1TBS (One True Brace Style), zwanej także ulepszonym stylem K&R, gdzie nawias otwierający jest w tej samej linii co deklaracja. Niemniej jednak ludzie nadal czują potrzebę dodawania dodatkowych pustych linii po każdej deklaracji bloku, ponieważ styl 1TBS jest zbyt ciasny. Gdy dodasz puste linie, efektywnie zamienia się to w styl Allman, ale ludzie nie potrafią się do tego przyznać.

W C# domyślnym stylem jest Allman, gdzie każdy nawias klamrowy znajduje się w swojej własnej linii. Uważam, że jest to znacznie czytelniejsze niż 1TBS lub K&R. Java używa stylu 1TBS, swoją drogą.

Musiałem sformatować kod w stylu 1TBS ze względu na ograniczenia składania wydawnictwa, ale sugeruję, abyś rozważył użycie stylu Allman, zwłaszcza w przypadku C#, nie tylko dlatego, że jest bardziej czytelny, ale także dlatego, że jest najczęściej stosowanym stylem dla C#.

Jednakże, gdy zdecydujesz się na ten styl, nie jest to takie proste jak w przykładzie, który pokazałem. Na przykład porównanie dwóch różnych obiektów PostId o takiej samej wartości nie zadziałałoby tak, jak oczekujesz, ponieważ domyślnie porównanie porównuje jedynie referencje, a nie zawartość klas (mówię o referencjach kontra wartości później w tym rozdziale). Musisz dodać pełną strukturę wokół tego, aby działało bez problemów. Oto krótka lista kontrolna:

1. Musisz przynajmniej zaimplementować nadpisanie metody Equals, ponieważ niektóre funkcje frameworka i biblioteki mogą od niej zależeć, aby porównać dwie instancje twojej klasy.
2. Jeśli planujesz porównywać wartości samodzielnie, używając operatorów równości (== i !=), musisz zaimplementować ich przeciążenia operatorów w klasie.
3. Jeśli planujesz używać klasy w Dictionary<K,V> jako klucza, musisz nadpisać metodę GetHashCode. Wyjaśnię później w tym rozdziale, jak związane są haszowanie i słowniki.

Funkcje formatowania ciągów znaków, takie jak String.Format, używają metody ToString do uzyskania odpowiedniej reprezentacji ciągu znaków klasy do drukowania.

NIE UŻYWAJ PRZECIĄŻANIA OPERATORÓW, ODMIENIAJ JE TYLKO WTEDY, GDY JEST TO KONIECZNE

Przeciążanie operatorów to sposób zmiany zachowania operatorów, takich jak ==, !=, + i -, w języku programowania. Programiści, którzy uczą się przeciążania operatorów, czasem przesadzają i zaczynają tworzyć własne, dziwaczne zachowanie dla nieistotnych klas, na przykład przeciążając operator +=, aby wstawić rekord do tabeli za pomocą składni takiej jak db += rekord. Prawie niemożliwe jest zrozumienie intencji takiego kodu. Jest również niemożliwe do zrozumienia, chyba że przeczytasz dokumentację. Nie ma funkcji w środowisku IDE do odkrywania, które operatory zostały przeciążone dla danego typu. Nie bądź osobą, która bez potrzeby stosuje przeciążanie operatorów. Nawet sam zapomnisz, co dany operator robi, i będziesz się za to obwiniał. Przeciążaj operatory tylko wtedy, gdy konieczne jest dostarczenie alternatywnych operatorów równości i rzutowania typów. Nie trać czasu na ich implementację, jeśli nie będą potrzebne.

W niektórych przykładach będziemy korzystać z przeciążania operatorów, ponieważ jest to wymagane, aby klasy były semantycznie równoważne wartościom, które reprezentują. Oczekujesz, że klasa będzie działać z operatorem == w taki sam sposób jak liczba, którą reprezentuje.

Listing 2.2 pokazuje klasę PostId z wszelkimi niezbędnymi konstrukcjami, aby upewnić się, że działa we wszystkich przypadkach równości. Przeciążyliśmy ToString(), więc nasza klasa staje się zgodna z formatowaniem ciągów znaków i łatwiejsza do zbadania jej wartości podczas debugowania. Przeciążyliśmy GetHashCode(), więc zwraca ona Value bezpośrednio, ponieważ sama wartość idealnie mieści się w typie int. Przeciążyliśmy metodę Equals(), aby sprawdzenia równości w kolekcjach tej klasy działały poprawnie, jeśli potrzebujemy unikalnych wartości lub chcemy przeszukać tę wartość. Wreszcie przeciążyliśmy operatory == i !=, abyśmy mogli porównywać wartości PostId bezpośrednio, nie uzyskując dostępu do ich wartości.

UWAGA

Niemodyfikowalna klasa służąca wyłącznie do reprezentowania wartości w ulicznej terminologii nazywana jest typem wartości ValueType. Dobrze jest znać potoczne nazwy, ale nie skupiaj się na nich. Skup się na ich użyteczności.

```
public class PostId
{
    public int Value { get; private set; }
    public PostId(int id)
    {
        if (id <= 0)
        {
            throw new ArgumentOutOfRangeException(nameof(id));
        }
        Value = id;
    }
    public override string ToString() => Value.ToString();
    public override int GetHashCode() => Value;
    public override bool Equals(object obj)
    {
        return obj is PostId other && other.Value == Value;
    }
    public static bool operator ==(PostId a, PostId b)
    {
        return a.Equals(b);
    }
    public static bool operator !=(PostId a, PostId b)
    {
        return !a.Equals(b);
    }
}
```

Składnia strzałki została wprowadzona do języka C# w wersji 6.0 i jest równoważna zwykłej składni metody z pojedynczym poleceniem return. Możesz wybrać składnię strzałki, jeśli kod staje się dzięki temu łatwiejszy do czytania. Nie ma jednoznacznej odpowiedzi na to, czy należy używać składni strzałki czy tradycyjnej—poprawny kod to czytelny kod, a nieczytelny kod jest błędny.

Metoda

```
public int Suma(int a, int b) {
   return a + b;
}
```

jest równoważna zapisowi

```
public int Suma(int a, int b) => a + b;
```

To zazwyczaj nie jest konieczne, ale jeśli twoja klasa musi być przechowywana w kontenerze, który jest sortowany lub porównywany, musisz zaimplementować dwie dodatkowe funkcje:

1. Musisz zapewnić możliwość sortowania, implementując interfejs IComparable, ponieważ samo porównanie równości nie jest wystarczające do określenia kolejności. Nie użyliśmy tego w przykładzie z listingu 2.1, ponieważ identyfikatory nie są oceniane.
2. Jeśli planujesz porównywać wartości za pomocą operatorów mniejsze niż (<) lub większe niż (>), musisz także zaimplementować odpowiednie przeciążenia tych operatorów (<=, >=).

To może wydawać się dużo pracy, zwłaszcza gdy można po prostu przekazywać liczbę całkowitą, ale w dużych projektach, zwłaszcza w pracy zespołowej, ta praktyka przynosi korzyści. Więcej zalet zobaczysz w kolejnych sekcjach.

Nie zawsze musisz tworzyć nowe typy, aby korzystać z kontekstu walidacji. Możesz użyć dziedziczenia, aby tworzyć podstawowe typy, które zawierają określone typy podstawowe z wspólnymi regułami. Na przykład możesz mieć ogólny typ identyfikatora, który można dostosować do innych klas. Po prostu możesz zmienić nazwę klasy PostId z listingu 2.1 na DbId i dziedziczyć wszystkie typy po nim.

Zawsze, gdy potrzebujesz nowego typu, jak PostId, UserId lub TopicId, możesz odziedziczyć go po DbId i rozszerzyć według potrzeb. W ten sposób możemy mieć w pełni funkcjonalne odmiany tego samego typu identyfikatora, aby lepiej je odróżnić od innych typów. Możesz również dodać więcej kodu do klas, aby je specjalizować w własny sposób:

```
public class PostId: DbId {
    public PostId(int id): base(id) { }
}

public class TopicId: DbId {
    public TopicId(int id) : base(id) { }
}

public class UserId: DbId {
    public UserId(int id): base(id) { }
}
```

Posiadanie osobnych typów dla elementów projektowych ułatwia semantyczną kategoryzację różnych zastosowań naszego typu DbId, jeśli używasz ich razem i często. Dodatkowo chroni cię przed przekazywaniem niewłaściwego typu identyfikatora do funkcji.

UWAGA

Zawsze, gdy widzisz rozwiązanie problemu, upewnij się, że wiesz, kiedy go nie używać. Ten scenariusz dotyczący ponownego wykorzystywania nie jest wyjątkiem. Możesz nie potrzebować tak zaawansowanego podejścia do swojego prostego prototypu – być może nie będziesz nawet potrzebować niestandardowej klasy. Kiedy zauważysz, że często przekazujesz tę samą wartość do funkcji i zapominasz, czy wymagała ona walidacji, może być korzystne, aby objąć ją klasą i przekazywać ją jako obiekt.

Niestandardowe typy danych są potężne, ponieważ mogą lepiej wyjaśnić twoje projekty niż typy podstawowe i pomóc w unikaniu powtarzającej się walidacji, a co za tym idzie, błędów. Mogą być warte trudu implementacji. Ponadto, używany przez ciebie framework może już dostarczać odpowiednich typów, których potrzebujesz.

2.3.3 Nie unikaj frameworku, korzystaj z niego mądrze

.NET, podobnie jak wiele innych frameworków, dostarcza zestawu przydatnych abstrakcji dla pewnych typów danych, które zazwyczaj są mało znane lub ignorowane. Niestandardowe wartości tekstowe, takie jak adresy URL, adresy IP, nazwy plików czy nawet daty, są zazwyczaj przechowywane jako ciągi znaków. Przyjrzymy się kilku z tych gotowych typów i jak możemy je wykorzystać.

Możliwe, że już znasz klasy w .NET dla tych typów danych, ale nadal możesz preferować użycie ciągu znaków, ponieważ jest łatwiejszy w obsłudze. Problem z ciągami znaków polega na braku dowodu walidacji; twoje funkcje nie wiedzą, czy dany ciąg znaków został już zweryfikowany, czy nie, co może prowadzić do przypadkowych błędów lub niepotrzebnego kodu ponownej walidacji, co z kolei spowalnia pracę. Korzystanie z gotowej klasy dla określonego typu danych jest lepszym wyborem w takich przypadkach.

Kiedy jedynym narzędziem, jakie masz, jest młotek, każdy problem wydaje się gwoździem. To samo dotyczy ciągów znaków. Ciągi znaków są świetnym ogólnym narzędziem do przechowywania treści i łatwo je analizować, dzielić, łączyć czy manipulować nimi. Są bardzo kuszące. Ale ta pewność co do ciągów znaków sprawia, że skłonny jesteś czasami wynajdować koło na nowo. Kiedy zaczynasz obsługiwać rzeczy za pomocą ciągu znaków, masz tendencję do wykonywania wszystkiego za pomocą funkcji przetwarzania ciągów znaków, chociaż może to być zupełnie zbędne.

Rozważ ten przykład: masz za zadanie napisać usługę wyszukiwania dla firmy skracającej adresy URL o nazwie Supercalifragilisticexpialidocious, która ma problemy finansowe z nieznanych powodów, i jesteś ich jedyną nadzieją, czyli Obi-wanem. Ich usługa działa w ten sposób:

Użytkownik podaje długi adres URL, na przykład:

https://llanfair.com/pwllgw/yngyll/gogerych/wyrndrobwll/llan/tysilio/gogo/goch.html

Usługa tworzy krótki kod dla tego URL-a oraz nowy krótki URL, na przykład:

https://su.pa/mK61

Za każdym razem, gdy użytkownik przechodzi do skróconego URL-a ze swojej przeglądarki internetowej, zostaje przekierowany pod podany w długim URL-u adres.

Funkcję, którą musisz zaimplementować, należy wykorzystać, aby wyodrębnić krótki kod ze skróconego URL-a. Podejście oparte na ciągach znaków wyglądałoby tak:

```
public string GetShortCode(string url)
{
    const string urlValidationPattern =
        @"^https?://([\w-]+.)+[\w-]+(/[\w- ./?%&=])?$";
    if (!Regex.IsMatch(url, urlValidationPattern)) {
        return null;
    }
    // weź część po ostatnim ukośniku
    string[] parts = url.Split('/');
    string lastPart = parts[^1];
    return lastPart;
}
```

Ten kod może wyglądać na prawidłowy na pierwszy rzut oka, ale już teraz zawiera błędy, oparte na naszej hipotetycznej specyfikacji. Wzorzec walidacji dla adresu URL jest niekompletny i pozwala na nieprawidłowe URL-e. Nie uwzględnia możliwości występowania wielu ukośników w ścieżce URL. Dodatkowo niepotrzebnie tworzy tablicę ciągów znaków tylko po to, aby uzyskać końcową część URL-a.

WAŻNE

Błąd może istnieć tylko w kontekście specyfikacji. Jeśli nie masz żadnej specyfikacji, nie możesz twierdzić, że coś jest błędem. Dzięki temu firmy unikają skandali PR-owych, odrzucając błędy stwierdzeniem, że „To jest cecha”. Specyfikacja nie musi być udokumentowana na piśmie – może istnieć tylko w twojej głowie, o ile możesz odpowiedzieć na pytanie, „Czy tak ma działać ta funkcja?”

Co ważniejsze, logika nie jest widoczna w kodzie. Lepsze podejście może wykorzystać klasę Uri z frameworka .NET i wyglądać jak w tym przykładzie:

```
public string GetShortCode(Uri url)
{
    string path = url.AbsolutePath;
    if (path.Contains('/')) {
        return null;
    }
    return path;
}
```

Tym razem nie mamy do czynienia z analizą ciągów znaków samodzielnie. To zostało już obsłużone przed wywołaniem naszej funkcji. Nasz kod jest bardziej opisowy i łatwiej jest go napisać, ponieważ użyliśmy Uri zamiast string. Ponieważ analiza i walidacja zachodzą wcześniej w kodzie, jest to również łatwiejsze do debugowania. Ta książka ma cały rozdział poświęcony debugowaniu, ale najlepszym rozwiązaniem jest nie musieć debugować od samego początku.

Oprócz podstawowych typów danych, takich jak int, string, float itp., .NET udostępnia wiele innych przydatnych typów danych do użycia w naszym kodzie. IPAddress jest lepszą alternatywą dla string do przechowywania adresów IP, nie tylko dlatego, że zawiera walidację, ale także dlatego, że obsługuje IPv6, które jest obecnie używane. To niewiarygodne, wiem. Klasa ta ma także skrócone metody do definiowania lokalnego adresu:

```
var testAddress = IPAddress.Loopback;
```

W ten sposób unikasz pisania 127.0.0.1 za każdym razem, gdy potrzebujesz adresu pętli zwrotnej, co sprawia, że pracujesz szybciej. Jeśli popełnisz błąd w adresie IP, złapiesz go wcześniej niż w przypadku stringa.

Inny taki typ to TimeSpan. Jak sama nazwa wskazuje, reprezentuje okres czasu. Okresy czasu są używane prawie wszędzie w projektach oprogramowania, zwłaszcza w mechanizmach pamięci podręcznej lub wygaszania. Zazwyczaj definiujemy okresy czasu jako stałe kompilacji. Najgorszy możliwy sposób to ten:

```
const int cacheExpiration = 5; // minuty
```

Nie jest od razu jasne, że jednostką wygaśnięcia pamięci podręcznej są minuty. Bez przeglądania kodu źródłowego niemożliwe jest ustalenie jednostki. Lepszym pomysłem jest przynajmniej jej uwzględnienie w nazwie, więc Twoi koledzy, a nawet Ty sam w przyszłości, będą wiedzieć, jakiego typu jest ta wartość, nie zaglądając do kodu źródłowego:

```
public const int cacheExpirationMinutes = 5;
```

To jest lepsze, ale gdy będziesz musiał używać tego samego okresu czasu dla innej funkcji, która przyjmuje inną jednostkę, będziesz musiał go przeliczyć:

```
cache.Add(key, value, cacheExpirationMinutes * 60);
```

To dodatkowa praca. Musisz pamiętać, żeby to zrobić. Jest to także podatne na błędy. Możesz źle wpisać 60 i otrzymać nieprawidłową wartość na końcu, co może prowadzić do wielogodzinnego debugowania lub niepotrzebnej optymalizacji wydajności z powodu takiej prostej pomyłki.

TimeSpan jest niesamowity pod tym względem. Nie ma powodu, aby reprezentować jakikolwiek okres czasu w inny sposób niż za pomocą TimeSpan, nawet gdy funkcja, którą wywołujesz, nie akceptuje TimeSpan jako parametru:

```
public static readonly TimeSpan cacheExpiration = TimeSpan.FromMinutes(5);
```

Patrzcie na tę piękną konstrukcję! Już wiesz, że to jest okres czasu, i jest zadeklarowany. Co lepsze, nie musisz znać jego jednostki nigdzie indziej. Dla każdej funkcji, która przyjmuje TimeSpan, po prostu go przekazujesz. Jeśli funkcja przyjmuje określoną jednostkę, na przykład minuty, jako liczbę całkowitą, możesz ją wywołać w ten sposób:

```
cache.Add(key, value, cacheExpiration.TotalMinutes);
```

I zostanie przekształcone na minuty. Genialne!

Wiele innych typów jest podobnie użytecznych, na przykład DateTimeOffset, który reprezentuje określoną datę i czas jak DateTime, ale zawiera informacje o strefie czasowej, dzięki czemu nie tracisz danych, gdy nagle zmieniają się informacje o strefie czasowej twojego komputera lub serwera. W rzeczywistości zawsze powinieneś próbować używać DateTimeOffset zamiast DateTime, ponieważ jest również łatwo

konwertowalny na/from DateTime. Możesz nawet używać operatorów arytmetycznych z TimeSpan i DateTimeOffset dzięki przeciążaniu operatorów:

```
var teraz = DateTimeOffset.Now;
var dataUrodzenia = new DateTimeOffset(1976, 12, 21, 02, 00, 00,
TimeSpan.FromHours(2));
TimeSpan czasOdUrodzenia = teraz - dataUrodzenia;
Console.WriteLine($"Minęło {czasOdUrodzenia.TotalSeconds} sekund od mojego
urodzenia!");
```

UWAGA

Obsługa daty i czasu to bardzo delikatny temat i łatwo go zepsuć, zwłaszcza w globalnych projektach. Dlatego istnieją oddzielne zewnętrzne biblioteki, które pokrywają brakujące przypadki użycia, takie jak Noda Time autorstwa Jona Skeeta.

.NET jest jak kupa złota, do której Sknerus McKwacz skacze i płynie w niej. Pełen jest świetnych narzędzi, które ułatwiają nam życie. Może się wydawać, że ich nauka jest marnotrawstwem czasu lub nudna, ale jest znacznie szybsza niż próba używania ciągów tekstowych lub wymyślanie własnych prowizorycznych implementacji.

2.3.4 Typy zamiast literówek

Pisanie komentarzy w kodzie może być uciążliwe, a w dalszej części książki przedstawiam argumenty przeciwko robieniu tego, chociaż zalecam poczekanie z oceną tego stanowiska, zanim zaczniesz rzucać w moim kierunku klawiaturą. Nawet bez komentarzy, twój kod nie musi być pozbawiony opisu. Typy mogą pomóc ci w wyjaśnieniu swojego kodu.

Wyobraź sobie, że trafiasz na ten fragment w labiryntach kodu twojego projektu:

```
public int Move(int from, int to) {
   // ... dość sporo kodu tutaj
   return 0;
}
```

Co robi ta funkcja? Co przemieszcza? Jakiego rodzaju parametry przyjmuje? Jakiego rodzaju wynik zwraca? Odpowiedzi na te pytania są niejasne bez typów. Możesz próbować zrozumieć kod lub sprawdzić klasę obejmującą, ale to zajmie czas. Twoje doświadczenie mogłoby być znacznie lepsze, gdyby zostało nazwane lepiej:

```
public int MoveContents(int fromTopicId, int toTopicId) {
   // ... dość sporo kodu tutaj
   return 0;
}
```

Teraz jest znacznie lepiej, ale nadal nie wiesz, jakiego rodzaju wynik zwraca ta funkcja. Czy to kod błędu, czy liczba przemieszczonych elementów, czy nowy identyfikator tematu wynikający z konfliktów w operacji przenoszenia? Jak możesz przekazać tę informację bez polegania na komentarzach w kodzie? Oczywiście, za pomocą typów. Zastanów się nad tym fragmentem kodu zamiast tego:

```
public MoveResult MoveContents(int fromTopicId, int toTopicId) {
   // ... wciąż dość sporo kodu tutaj
   return MoveResult.Success;
}
```

To nieco jasniejsze. Nie dodaje wiele, ponieważ już wiedzieliśmy, że typ `int` był wynikiem funkcji `move`, ale istnieje różnica – teraz możemy zgłębić, co znajduje się w typie `MoveResult`, aby zobaczyć, co właściwie robi ta funkcja, po prostu naciskając klawisz F12 w programach Visual Studio i VS Code:

```
public enum MoveResult
{
   Success,
   Unauthorized,
   AlreadyMoved
}
```

Mamy teraz znacznie lepszy pomysł. Zmiany nie tylko poprawiają zrozumienie API metody, ale także sam kod w funkcji, ponieważ zamiast stałych lub, co gorsza, hardcoded wartości całkowitych, widzisz czytelne `MoveResult.Success`. W przeciwieństwie do stałych w klasie, enumeracje ograniczają możliwe wartości, które można przekazywać, i posiadają swoją własną nazwę typu, co pozwala lepiej opisać intencję.

Ponieważ funkcja przyjmuje liczby całkowite jako parametry, musi zawierać pewną walidację, ponieważ jest to publicznie dostępne API. Można powiedzieć, że może być nawet potrzebne w kodzie wewnętrznym lub prywatnym ze względu na powszechność walidacji. To wyglądałoby lepiej, gdyby była tam logika walidacji w oryginalnym kodzie:

```
public MoveResult MoveContents(TopicId from, TopicId to) {
   // ... nadal dość sporo kodu tutaj
   return MoveResult.Success;
}
```

Jak widać, typy mogą działać na twoją korzyść, przenosząc kod w odpowiednie miejsce i ułatwiając zrozumienie. Ponieważ kompilator sprawdza, czy poprawnie napisałeś nazwę typu, chronią cię przed literówkami.

2.3.5 Być nullable czy nie być nullable

W dłuższej perspektywie każdy programista napotka na wyjątek NullReferenceException. Choć Tony Hoare, potocznie znany jako wynalazca null, nazywa swoje wprowadzenie go do programowania „błędem za miliard dolarów”, nie wszystko jest stracone.

KILKA SŁÓW O NULL

Null, zwane także nil w niektórych językach, to wartość symbolizująca brak wartości lub apatię programisty. Zwykle jest synonimem wartości zero. Ponieważ adres pamięci o wartości zero oznacza nieprawidłowy obszar w pamięci, nowoczesne procesory potrafią wykryć ten nieprawidłowy dostęp i zamienić go na przyjazną wiadomość o wyjątku. W średniowiecznej erze komputingu, gdy dostępy do nulli nie były sprawdzane, komputery zamrażały się, ulegały korupcji lub po prostu się restartowały.

Problemem nie jest dokładnie sama wartość null - w naszym kodzie musimy i tak opisać brakującą wartość. Ona istnieje z określonym celem. Problem polega na tym, że wszystkim zmiennym można przypisać nulla domyślnie i nigdy nie jest sprawdzane, czy nie zostały one nieoczekiwanie przypisane do nulli, co powoduje, że są one przypisywane do nulli w najbardziej nieoczekiwanych miejscach i w końcu program się zawiesza.

JavaScript, jakby nie miał już dość problemów z systemem typów, posiada dwie różne wartości null: null i undefined. Null symbolizuje brakującą wartość, podczas gdy undefined symbolizuje brak przypisania. Wiem, to boli. Musisz zaakceptować JavaScript takim, jaki jest.

C# 8.0 wprowadził nową funkcję, zwaną nullable references (nullable references). Wydaje się to być prostą zmianą: referencje nie mogą być domyślnie przypisywane do null. To wszystko. Nullable references są prawdopodobnie najważniejszą zmianą w języku C# od wprowadzenia generyków. Każda inna funkcja związana z nullable references jest powiązana z tą podstawową zmianą.

Mylącą nazwą tej funkcji jest, że referencje były już nullable przed C# 8.0. Powinno to być nazwane non-nullable references (non-nullable references), aby dać programistom lepszy pomysł na to, co oznacza. Rozumiem logikę w nazwaniu tego ze względu na wprowadzenie nullable value types, ale wielu programistów może uważać, że nie wprowadza to niczego nowego. Kiedy wszystkie referencje były nullable, wszystkie funkcje, które przyjmowały referencje, mogły otrzymać dwie różne wartości: prawidłową referencję i null. Każda funkcja, która nie spodziewała się wartości null, powodowała awarię, gdy próbowała odnieść się do wartości.

Robienie referencji non-nullable jako domyślne zachowanie zmieniło to. Funkcje nigdy nie otrzymają już wartości null, o ile kod wywołujący istnieje w tym samym projekcie. Przyjrzyj się poniższemu kodowi:

```
public MoveResult MoveContents(TopicId from, TopicId to) {
    if (from is null) {
        throw new ArgumentNullException(nameof(from));
    }
    if (to is null) {
        throw new ArgumentNullException(nameof(to));
    }
    // .. tutaj jest właściwy kod
    return MoveResult.Success;
}
```

Wskazówka:

Składnia `is null` w powyższym kodzie może wydać ci się obca. Ostatnio zacząłem używać jej zamiast `x == null`, gdy dowiedziałem się o niej w dyskusji na Twitterze prowadzonej przez starszych inżynierów Microsoftu. Wygląda na to, że operator `is` nie może być przeciążony, więc zawsze gwarantuje poprawny wynik. Podobnie możesz używać składni `x is object`, zamiast `x != null`. Sprawdzenie non-nullable eliminuje potrzebę sprawdzania nulli w kodzie, ale zewnętrzny kod nadal może wywoływać twój kod z nullami, na przykład, jeśli publikujesz bibliotekę. W takim przypadku nadal możesz potrzebować jawnie

sprawdzać nulli.

DLACZEGO SPRAWDZAMY NULL, SKORO I TAK KOD ZAWIESI SIĘ?

Jeśli nie sprawdzisz argumentów na null na początku funkcji, funkcja będzie nadal działać, aż odniesie się do tej wartości null. Oznacza to, że może zatrzymać się w niepożądanym stanie, takim jak połowicznie zapisany rekord, lub może nie zatrzymać się w ogóle, ale wykonać nieprawidłową operację, o której nie zauważysz. Jak najszybsze wykrycie błędów i unikanie niewłaściwych stanów zawsze są dobrymi pomysłami. Nie ma powodu, by bać się awarii: to szansa dla ciebie, aby znaleźć błędy.

Jeśli zawiśnie wcześnie, ślad stosu dla wyjątku będzie wyglądać czyściej. Będziesz dokładnie wiedzieć, który parametr spowodował awarię funkcji.

Nie wszystkie wartości null muszą być sprawdzane. Możesz otrzymywać wartość opcjonalną, a null jest najprostszym sposobem wyrażenia tego zamiaru. Rozdział dotyczący obsługi błędów będzie omawiał ten temat bardziej szczegółowo.

Można włączyć sprawdzanie nulli na poziomie projektu lub pliku. Zawsze polecam włączanie go na poziomie projektu dla nowych projektów, ponieważ zachęca to do pisania poprawnego kodu od samego początku, dzięki czemu mniej czasu trzeba poświęcać na naprawę błędów. Aby włączyć to na poziomie pliku, dodajesz linię #nullable enable na początku pliku.

Rada eksperta:

Zawsze zakończ dyrektywę kompilatora enable/disable odpowiednikiem restore, a nie przeciwnością enable/disable. W ten sposób nie wpływasz na ustawienia globalne. Pomaga to, gdy eksperymentujesz ze zbiorczymi ustawieniami projektu. W przeciwnym razie możesz przegapić cenne informacje zwrotne.

Po włączeniu sprawdzania nulli, twój kod wygląda tak:

```
#nullable enable
public MoveResult MoveContents (TopicId from, TopicId to) {
    // .. tutaj jest właściwy kod
    return MoveResult.Success;
}
#nullable restore
```

Gdy próbujesz wywołać funkcję MoveResult z wartością null lub wartością nullable, otrzymasz natychmiastowe ostrzeżenie kompilatora, zamiast błędu w losowym momencie w produkcji. Zidentyfikujesz błąd jeszcze przed próbą uruchomienia kodu. Możesz zdecydować się zignorować ostrzeżenia i kontynuować, ale nigdy nie powinieneś tego robić.

Nullable references mogą być początkowo uciążliwe. Nie możesz łatwo deklarować klas tak, jak kiedyś to robiłeś. Załóżmy, że tworzymy stronę rejestracyjną na konferencję, która otrzymuje imię i adres e-mail uczestnika i zapisuje wyniki do bazy danych. Nasza klasa ma pole źródło kampanii, które jest dowolnym ciągiem znaków przekazywanym z sieci reklamowej. Jeśli ciąg nie ma wartości, oznacza to, że strona jest dostępna bezpośrednio, a nie po kliknięciu w reklamę. Załóżmy, że mamy klasę taką jak ta:

```
#nullable enable
class ConferenceRegistration
{
    public string CampaignSource { get; set; }
    public string FirstName { get; set; }
    public string? MiddleName { get; set; }
    public string LastName { get; set; }
    public string Email { get; set; }
    public DateTimeOffset CreatedOn { get; set; }
}
#nullable restore
```

Kiedy próbujesz skompilować klasę ze snippetu, otrzymasz ostrzeżenie kompilatora dla wszystkich niemodyfikowalnych ciągów znaków, czyli wszystkich właściwości oprócz MiddleName i CreatedOn:

```
Non-nullable property '...' is uninitialized. Consider declaring the property
as nullable.
```

Drugie imię jest opcjonalne, dlatego zadeklarowaliśmy MiddleName jako nullable. Dlatego nie otrzymałeś błędu kompilatora.

UWAGA

Nigdy nie używaj pustych ciągów znaków do oznaczania opcjonalności. Do tego celu używaj null. Niemożliwe jest dla twoich kolegów zrozumienie twojego zamiaru za pomocą pustego ciągu znaków. Czy puste ciągi znaków są prawidłowymi wartościami, czy wskazują opcjonalność? Niemożliwe do ustalenia. Null jest jednoznaczny.

NA TEMAT PUSTYCH CIĄGÓW ZNAKÓW

Przez całą swoją karierę będziesz musiał deklarować puste ciągi znaków w celach innych niż opcjonalność. Kiedy zrobisz to, unikaj stosowania notacji "" do oznaczania pustych ciągów znaków. Ze względu na wiele różnych środowisk, w których kod może być wyświetlany, takich jak twój edytor tekstu, okno wyjściowe testowego uruchomienia lub strona ciągłej integracji, łatwo jest je pomylić z ciągiem znaków zawierającym pojedynczą spację (" "). Wyraźnie deklaruj puste ciągi znaków za pomocą String.Empty, aby wykorzystać istniejące typy. Możesz go również używać z małej litery, jako string.Empty, zgodnie z zasadami konwencji w twoim kodzie. Pozwól, że to kod przekazywać będzie twoje zamiary.

Z kolei CreatedOn to struktura, więc kompilator wypełnia ją zerami. Dlatego nie generuje on błędu kompilacji, ale nadal może być to coś, czego chcemy uniknąć.

Pierwszą reakcją programisty na naprawę błędu kompilacji jest zastosowanie sugerowanej przez kompilator propozycji. W poprzednim przykładzie oznaczałoby to deklarację właściwości jako nullable, co jednak zmienia nasze zrozumienie. Nagle robimy właściwości dotyczące imienia i nazwiska opcjonalnymi, co nie jest naszym celem. Musimy zastanowić się, jak chcemy zastosować semantykę opcjonalności.

Jeśli chcesz, aby właściwość nie była wartością null, musisz sobie zadać kilka pytań. Po pierwsze, "Czy właściwość ma wartość domyślną?"

Jeśli tak, możesz przypisać wartość domyślną podczas konstrukcji. To pozwoli ci lepiej zrozumieć zachowanie klasy, gdy przeglądasz kod. Jeśli pole źródła kampanii ma wartość domyślną, można to wyrazić w ten sposób:

```
public string CampaignSource { get; set; } = "organic";
public DateTimeOffset CreatedOn { get; set; } = DateTimeOffset.Now;
```

To usunie ostrzeżenie kompilatora i przekazywać będzie twoje intencje osobie czytającej twój kod.

Imiona i nazwiska nie mogą być opcjonalne ani mieć wartości domyślnych. Nie próbuj ustawić wartości domyślnych na "John" i "Doe". Zadaj sobie pytanie: „Jak chcę zainicjalizować tę klasę?”

Jeśli chcesz, aby twoja klasa była inicjalizowana za pomocą niestandardowego konstruktora, aby nie zezwalała na wartości nieprawidłowe, możesz przypisać wartości właściwości w konstruktorze i oznaczyć je jako private set, dzięki czemu będzie niemożliwe ich zmienienie. Omówimy to bardziej szczegółowo w sekcjach dotyczących niemutowalności. Opcjonalność można oznaczyć w konstruktorze za pomocą parametru opcjonalnego o wartości domyślnej null, jak pokazano poniżej.

Listing 2.3 Przykładowa niemodyfikowalna klasa

```
class ConferenceRegistration
{
    public string CampaignSource { get; private set; }
    public string FirstName { get; private set; }
    public string? MiddleName { get; private set; }
    public string LastName { get; private set; }
    public string Email { get; private set; }
    public DateTimeOffset CreatedOn { get; private set; } = DateTime.Now;

    public ConferenceRegistration(
        string firstName,
        string? middleName,
        string lastName,
        string email,
        string? campaignSource = null) {
        FirstName = firstName;
        MiddleName = middleName;
        LastName = lastName;
        Email = email;
        CampaignSource = campaignSource ?? "organic";
    }
}
```

Słyszę, jak marudzisz: „Ale to za dużo pracy”. Zgadzam się. Tworzenie niemodyfikowalnej klasy nie powinno być takie trudne. Na szczęście zespół C# wprowadził nową konstrukcję o nazwie rekordy w C# 9.0, aby ułatwić to zadanie, ale jeśli nie możesz używać C# 9.0, musisz podjąć decyzję: czy chcesz mieć mniej błędów, czy chcesz po prostu jak najszybciej zakończyć pracę?

REKORDY RATUJĄ

C# 9.0 wprowadził rekordy, które sprawiają, że tworzenie niemodyfikowalnych klas jest niezwykle łatwe. Klasę z listingu 2.3 można wyrazić za pomocą takiego kodu:

```
public record ConferenceRegistration(
    string CampaignSource,
    string FirstName,
    string? MiddleName,
    string LastName,
    string Email,
    DateTimeOffset CreatedOn);
```

Rekord automatycznie tworzy właściwości o tych samych nazwach, co argumenty podane w liście parametrów, i czyni te właściwości niemodyfikowalnymi, więc kod rekordu będzie działał dokładnie tak samo jak klasa pokazana w listingu 2.3. Możesz również dodawać metody i dodatkowe konstruktory w bloku rekordu, tak jak w zwykłej klasie, zamiast kończyć deklarację średnikiem. To fenomenalne ułatwienie. Oszczędza mnóstwo czasu.

To trudna decyzja, ponieważ ludzie są dość kiepscy w szacowaniu kosztów przyszłych wydarzeń i zwykle skupiają się tylko na bliskiej przyszłości. To właśnie dlatego udało mi się napisać tę książkę teraz – przestrzegam nakazu pozostawania w miejscu zamieszkania w San Francisco z powodu pandemii COVID-19, ponieważ ludzkość nie przewidziała przyszłych kosztów małego wybuchu w Wuhanie, w Chinach. Jesteśmy bardzo słabymi oszacowującymi. Przyjmijmy to do wiadomości.

Rozważmy to: masz szansę wyeliminować całą klasę błędów spowodowanych brakiem kontroli wartości null i niepoprawnym stanem, po prostu mając taki konstruktor, albo możesz pozostawić to tak, jak jest, i zmagać się z konsekwencjami dla każdego zgłoszonego błędu: raportami o błędach, śledzeniem problemów, omawianiem tego z menedżerem projektu, sortowaniem i naprawianiem odpowiedniego błędu, tylko po to, by napotkać kolejny błąd tej samej klasy, dopóki nie zdecydujesz: "Ok, dość tego, zrobię to tak, jak Sedat mi powiedział." Którą ścieżkę chcesz wybrać?

Jak już wcześniej powiedziałem, wymaga to pewnego rodzaju intuicji dotyczącej tego, ile błędów przewidujesz w jakiejś części kodu. Nie powinieneś ślepo stosować sugestii. Powinieneś mieć pewien obraz przyszłych zmian, czyli ilość zmian w danym fragmencie kodu. Im więcej kod będzie się zmieniać w przyszłości, tym bardziej będzie podatny na błędy.

Ale powiedzmy, że już to zrobiłeś i zdecydowałeś: "Nie, to będzie w porządku, nie warto sobie tym zawracać głowy." Możesz nadal uzyskać pewien poziom bezpieczeństwa przed wartościami null, zachowując kontrole wartości null, ale inicjalizując pola wcześniej, na przykład tak:

```
class ConferenceRegistration
{
    public string CampaignSource { get; set; } = "organic";
    public string FirstName { get; set; } = null!;
    public string? MiddleName { get; set; }
    public string LastName { get; set; } = null!;
    public string Email { get; set; } = null!;
    public DateTimeOffset CreatedOn { get; set; }
}
```

Operator wykrzyknika (!) precyzyjnie informuje kompilator: "Wiem, co robię": w tym przypadku oznacza to, że "Będę się upewniać, że zainicjalizuję te właściwości zaraz po utworzeniu tej klasy. Jeśli tego nie zrobię, akceptuję, że sprawdzenia wartości null nie będą działać dla mnie w ogóle." W zasadzie nadal zachowujesz pewność co do wartości null, jeśli dotrzymasz obietnicy i zainicjujesz te właściwości od razu po utworzeniu obiektu.

To jednak delikatne pole do działania, ponieważ może być trudno przekonać cały zespół do tego podejścia, a niektórzy mogą nadal inicjować te właściwości później. Jeśli uważasz, że możesz zapanować nad ryzykiem, możesz się trzymać tego podejścia. Może to być nawet nieuniknione w przypadku niektórych bibliotek, takich jak Entity Framework, które wymagają domyślnego konstruktora i właściwości, które można ustawiać.

MAYBE UMARŁO, NIECH ŻYJE NULLABLE!

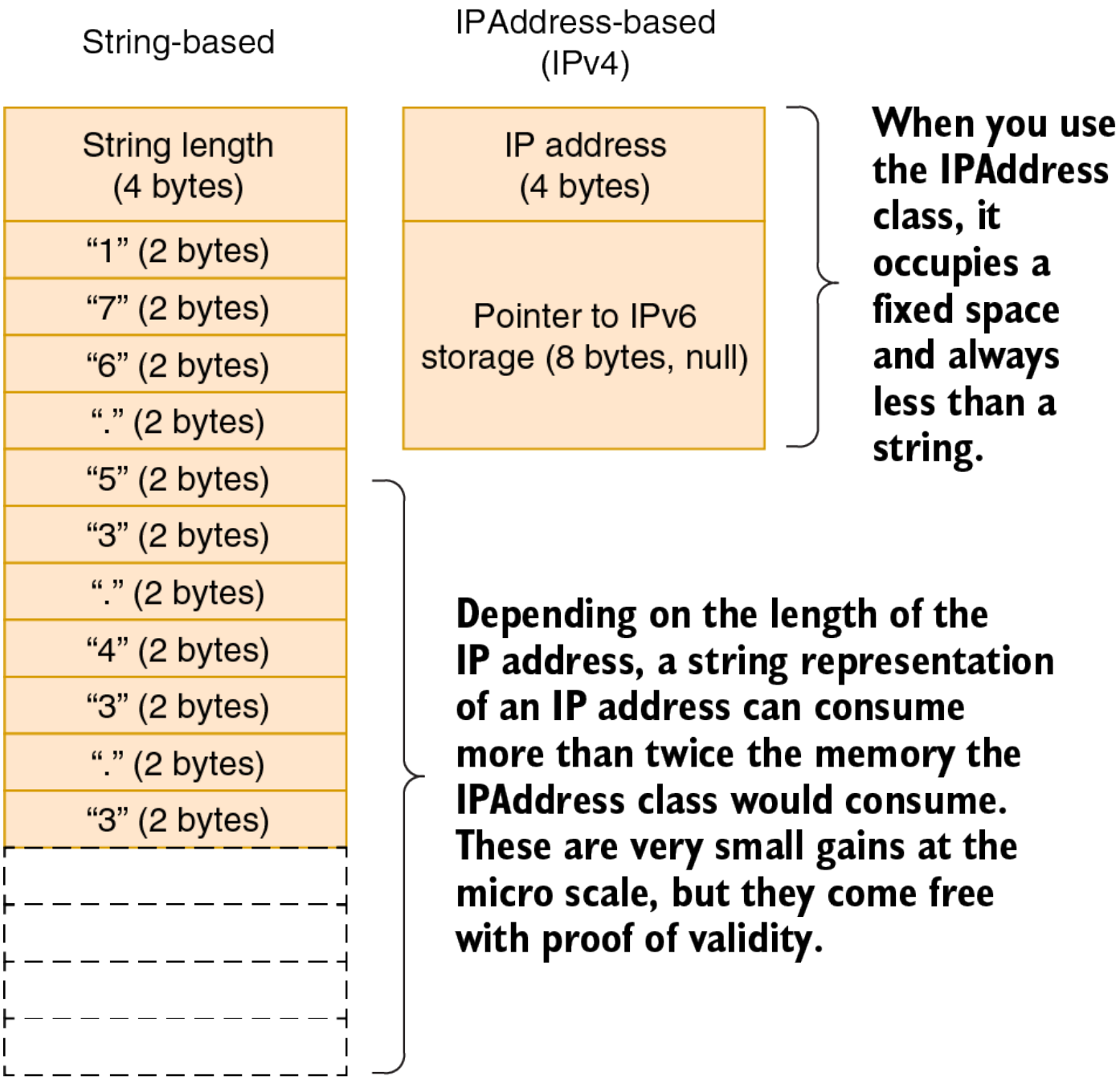
Ponieważ typy nullable w C# wcześniej nie miały wsparcia kompilatora do egzekwowania ich poprawności, a błąd mógłby spowodować awarię całego programu, były one historycznie uważane za gorsze rozwiązanie do oznaczania opcjonalności. Ze względu na to ludzie implementowali własne typy opcjonalne, nazywane Maybe lub Option, aby uniknąć ryzyka wystąpienia wyjątku null reference. C# 8.0 wprowadza kontrolę poprawności wartości null przez kompilator jako pierwszorzędową funkcję, więc era tworzenia własnych typów opcjonalnych oficjalnie dobiegła końca. Kompilator może lepiej sprawdzać i optymalizować typy nullable niż ad hoc implementacje. Dodatkowo język oferuje wsparcie składniowe, takie jak operatory i dopasowanie wzorców. Niech żyje Nullable!

Sprawdzenia wartości null pomagają zrozumieć twoje intencje dotyczące pisania kodu. Będziesz miał jasniejszy obraz, czy ta wartość jest naprawdę opcjonalna, czy może wcale nie musi być opcjonalna. To zmniejszy liczbę błędów i uczyni cię lepszym programistą.

2.3.6 Lepsza wydajność za darmo

Wydajność nie powinna być twoim głównym zmartwieniem, kiedy piszesz prototyp, ale ogólne zrozumienie charakterystyk wydajności typów, struktur danych i algorytmów może sprawić, że twoja droga będzie szybsza. Możesz napisać szybszy kod, nie zdając sobie z tego sprawy. Używanie odpowiedniego typu do konkretnej pracy zamiast bardziej ogólnych może pomóc ci za kulisami.

Istniejące typy mogą korzystać z bardziej wydajnych metod przechowywania za darmo. Na przykład prawidłowy ciąg znaków IPv6 może mieć nawet 65 znaków. Adres IPv4 ma co najmniej siedem znaków. Oznacza to, że przechowywanie oparte na ciągach znaków zajmie od 14 do 130 bajtów, a gdy doliczymy nagłówki obiektów, to między 30 a 160 bajtów. Typ IPAddress z kolei przechowuje adres IP jako serię bajtów i używa od 20 do 44 bajtów. Rysunek 2.12 pokazuje różnicę w układzie pamięci między przechowywaniem opartym na ciągach znaków a bardziej "rodzimą" strukturą danych.



To może wydawać się niewiele, ale pamiętaj, że to wszystko jest darmowe. Im dłuższy staje się adres IP, tym więcej oszczędzasz miejsca. Zapewnia również weryfikację, dzięki czemu możesz bezpiecznie zaufać, że przekazany obiekt zawiera poprawny adres IP przez cały kod. Twój kod staje się łatwiejszy do czytania, ponieważ typy także opisują intencję za danymi.

Z drugiej strony wszyscy wiemy, że nie ma darmowego obiadu. Jaka jest tutaj haczyk? Kiedy nie powinieneś tego używać? Cóż, istnieje niewielkie obciążenie analizą ciągu znaków, aby go rozebrać na bajty. Pewien fragment kodu analizuje ciąg znaków, aby określić, czy to adres IPv4, czy IPv6, i odpowiednio parsuje go za pomocą zoptymalizowanego kodu. Z drugiej jednak strony, ponieważ ciąg zostanie zweryfikowany po analizie, to w zasadzie eliminuje potrzebę walidacji w reszcie kodu, rekompensując niewielkie obciążenie spowodowane parsowaniem. Użycie właściwego typu od samego początku pozwala uniknąć obciążenia związanego z upewnianiem się, że przekazane argumenty są prawidłowego typu. Ostatecznie, preferowanie odpowiedniego typu może również wykorzystywać typy wartości w niektórych przypadkach, gdzie są one korzystne. Więcej na ten temat dowiemy się w następnym rozdziale.

Wydajność i skalowalność to nie jednowymiarowe koncepcje. Na przykład optymalizacja przechowywania danych może czasem prowadzić do gorszej wydajności, jak to wyjaśnię w rozdziale 7. Ale z uwagi na wszystkie zalety stosowania odpowiedniego typu do zadania, używanie specjalistycznego typu danych jest większość czasu oczywistym wyborem.

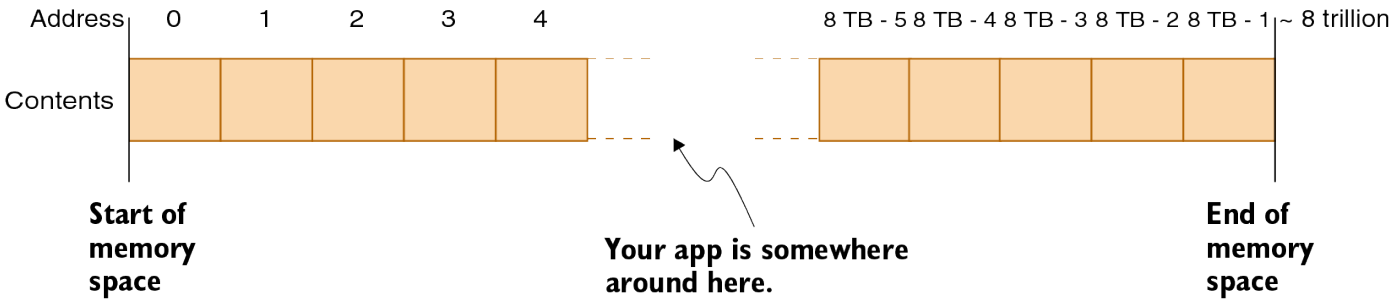
2.3.7 Typy referencyjne vs. typy wartości

Różnica między typami referencyjnymi a typami wartości polega głównie na tym, jak są przechowywane w pamięci. W prostych słowach, zawartość typów wartości jest przechowywana na stosie wywołań, podczas gdy typy referencyjne są przechowywane na stercie, a na stosie jest przechowywany jedynie odnośnik do ich zawartości. Oto prosty przykład, jak to wygląda w kodzie:

```
int result = 5;
var builder = new StringBuilder();
var date = new DateTime(1984, 10, 9);
string formula = "2 + 2 = ";
builder.Append(formula);
builder.Append(result);
builder.Append(date.ToString());
Console.WriteLine(builder.ToString());
```

Java nie posiada typów wartości oprócz podstawowych, takich jak int. C# pozwala dodatkowo definiować własne typy wartości. Znajomość różnicy między typami referencyjnymi a typami wartości pozwala ci być bardziej efektywnym programistą, ponieważ pozwala używać odpowiedniego typu dla konkretnej sytuacji. To nie jest trudne do nauczenia się.

Referencja jest analogiczna do zarządzanego wskaźnika. Wskaźnik to adres w pamięci. Zwykle wyobrażam sobie pamięć jako wyjątkowo długi ciąg bajtów, jak pokazano na rysunku 2.13.



To nie cała twoja pamięć RAM; to tylko układ pamięci jednego procesu. Zawartość twojej fizycznej pamięci RAM wygląda znacznie bardziej skomplikowanie, ale systemy operacyjne ukrywają fakt, że pamięć RAM jest chaotyczna, prezentując ci schludny, czysty, ciągły obszar pamięci dla każdego procesu, który może nawet nie istnieć w twojej pamięci RAM. Dlatego nazywa się to pamięcią wirtualną. Do roku 2020 nikt nie posiadał nawet blisko 8 TB pamięci RAM w swoich komputerach, ale na 64-bitowym systemie operacyjnym można uzyskać dostęp do 8 TB pamięci. Jestem pewien, że ktoś w przyszłości będzie na to patrzył i śmiał się, podobnie jak ja śmieję się z mojego starego komputera, który miał 1 MB pamięci w latach 90.

DLACZEGO 8 TB? MYŚLAŁEM, ŻE 64-BITOWE PROCESORY MOGĄ OBSŁUGIWAĆ 16 EXABAJTÓW!

Mogą. Ograniczenie przestrzeni użytkownika ma głównie praktyczne uzasadnienie. Tworzenie tabel mapowania pamięci wirtualnej dla mniejszego zakresu pamięci zużywa mniej zasobów i jest szybsze dla systemu operacyjnego. Na przykład przełączanie się między procesami wymaga pełnego przemapowania pamięci, a większa przestrzeń adresowa sprawiłaby, że proces ten byłby wolniejszy. W przyszłości będzie możliwe zwiększenie zakresu przestrzeni użytkownika, gdy 8 TB RAM stanie się powszechnie dostępnym towarem, ale do tego czasu 8 TB stanowi nasz horyzont.

Wskaźnik to po prostu liczba wskazująca na adres w pamięci. Korzyścią z używania wskaźników zamiast rzeczywistych danych jest unikanie niepotrzebnego kopiowania, co może być dość kosztowne. Możemy przekazywać gigabajty danych z funkcji do funkcji, przekazując jedynie adres, czyli wskaźnik. W przeciwnym razie musielibyśmy kopiować gigabajty pamięci przy każdym wywołaniu funkcji. W tym przypadku kopiujemy tylko liczbę.

Oczywiście nie ma sensu używać wskaźników dla wartości mniejszych niż rozmiar wskaźnika. 32-bitowa liczba całkowita (int w C#) jest tylko w połowie rozmiaru wskaźnika na systemie 64-bitowym. Dlatego typy podstawowe, takie jak int, long, bool i byte, są uważane za typy wartości. Oznacza to, że zamiast wskaźnika na ich adres, przekazywana jest tylko ich wartość do funkcji.

Referencja jest synonimem wskaźnika, z tą różnicą, że dostęp do jej zawartości jest zarządzany przez środowisko wykonawcze .NET. Nie możesz poznać wartości referencji. Dzięki temu system GC (Garbage Collector) może przenosić pamięć wskazywaną przez referencje w dowolne miejsce, bez twojej wiedzy na ten temat. W C# możesz również używać wskaźników, ale jest to możliwe tylko w kontekście niesafe.

GARBAGE COLLECTION

Programista musi śledzić alokację pamięci i zwolnić (dealokować) przydzieloną pamięć, gdy już nie jest ona potrzebna. W przeciwnym razie zużycie pamięci w aplikacji stale rośnie, co jest nazywane wyciekiem pamięciowym. Ręczna alokacja i zwalnianie pamięci jest podatna na błędy. Programista może zapomnieć zwolnić pamięć lub, co gorsza, próbować zwolnić już zwolnioną pamięć, co jest źródłem wielu błędów bezpieczeństwa.

Jednym z pierwszych proponowanych rozwiązań dla problemów z manualnym zarządzaniem pamięcią było zliczanie referencji. Jest to prymitywna forma garbage collection. Zamiast pozostawiać inicjatywę zwalniania pamięci programiście, środowisko uruchomieniowe przechowuje ukryty licznik dla każdego przydzielonego obiektu. Każda referencja do danego obiektu zwiększa ten licznik, a za każdym razem, gdy zmienna wskazująca na obiekt wychodzi poza zakres, licznik jest zmniejszany. Gdy licznik osiągnie zero, oznacza to, że nie ma zmiennych wskazujących na obiekt, więc zostaje zwolniony.

Zliczanie referencji działa dobrze w wielu przypadkach, ale ma kilka wad: jest wolne, ponieważ za każdym razem, gdy referencja wychodzi poza zakres, wykonuje się dealokacja, która zwykle jest mniej wydajna niż zwalnianie odpowiednich bloków pamięci razem. Tworzy także problem z cyklicznymi referencjami, który wymaga dodatkowej pracy i staranności ze strony programisty, aby ich uniknąć.

Następnie jest garbage collection, a dokładniej garbage collection typu mark and sweep, ponieważ zliczanie referencji jest również formą garbage collection. Garbage collection stanowi kompromis między zliczaniem referencji a manualnym zarządzaniem pamięcią. W garbage collection nie przechowuje się osobnych liczników referencji. Zamiast tego osobny proces przechodzi przez całe drzewo obiektów, aby znaleźć obiekty, które nie są już referencjonowane, i oznacza je jako śmieci. Śmieci są przechowywane przez pewien czas, a gdy ich liczba przekroczy określony próg, garbage collector przychodzi i zwalnia nieużywaną pamięć

jednym przebiegiem. To redukuje nakład operacji dealokacji pamięci i fragmentację pamięci spowodowaną częstymi małymi dealokacjami. Brak konieczności przechowywania liczników sprawia również, że kod działa szybciej.

Język programowania Rust wprowadził innowacyjne podejście do zarządzania pamięcią zwane borrow checker. Kompilator śledzi dokładny moment, kiedy przydzielona pamięć przestaje być potrzebna. Oznacza to, że alokacja pamięci nie ma dodatkowego kosztu w czasie wykonania, pod warunkiem, że kod jest napisany w określony sposób i obejmuje rozwiązywanie błędów kompilatora, aż do znalezienia odpowiedniego sposobu postępowania.

W języku C# programiści mają możliwość korzystania z bardziej złożonych typów wartości zwanych strukturami (structs). Struktura jest podobna do klasy pod względem definicji, ale w przeciwieństwie do klasy jest zawsze przekazywana przez wartość. Gdy struktura jest przekazywana do funkcji, tworzony jest jej egzemplarz, a gdy ta funkcja przekazuje ją do innej funkcji, tworzony jest kolejny egzemplarz. Struktury są zawsze kopiowane. Poniższy przykład ilustruje ten proces.

Listing 2.4 Przykład niemodyfikowalności

```
struct Point
{
   public int X;
   public int Y;
   public override string ToString() => $"X:{X},Y:{Y}";
}

static void Main(string[] args) {
   var a = new Point() {
       X = 5,
       Y = 5,
   };
   var b = a;
   b.X = 100;
   b.Y = 200;
   Console.WriteLine(b);
   Console.WriteLine(a);
}
```

Co myślisz, co ten program wypisze na konsoli? Kiedy przypisujesz `a` do `b`, środowisko uruchomieniowe tworzy nową kopię `a`. Oznacza to, że gdy modyfikujesz `b`, zmieniasz nową strukturę o wartościach `a`, a nie samą `a`. Co by było, gdyby `Point` był klasą? Wtedy `b` miałoby ten sam odnośnik co `a`, i zmiana zawartości `a` oznaczałaby jednoczesną zmianę `b`.

Typy wartości istnieją, ponieważ istnieją przypadki, w których są one bardziej efektywne niż typy referencyjne, zarówno pod względem przechowywania, jak i wydajności. Omówiliśmy już, jak typ o rozmiarze referencji lub mniejszym może być bardziej efektywnie przekazywany przez wartość. Typy referencyjne wymagają także jednego poziomu pośrednictwa. Za każdym razem, gdy musisz uzyskać dostęp do pola typu referencyjnego, środowisko uruchomieniowe .NET musi najpierw odczytać wartość referencji, przejść pod adres wskazywany przez referencję, a następnie odczytać rzeczywistą wartość. Dla typu wartości, środowisko uruchomieniowe odczytuje wartość bezpośrednio, co sprawia, że dostęp jest szybszy.

Podsumowanie

- Teoria informatyki może być nudna, ale znajomość pewnych koncepcji może uczynić cię lepszym programistą.
- Typy są często postrzegane jako zbędny kod w językach silnie typowanych, ale mogą także pomóc w redukcji ilości kodu.
- .NET oferuje bardziej wydajne struktury danych dla określonych typów danych, co może znacznie przyspieszyć i ulepszyć twój kod.
- Używanie typów może sprawić, że twój kod będzie bardziej zrozumiały, co oznacza mniej potrzeby komentowania kodu.
- Wprowadzona w C# 8.0 funkcja nullable references może sprawić, że twój kod będzie bardziej niezawodny i pozwoli ci zaoszczędzić czas potrzebny na debugowanie aplikacji.
- Różnica między typami wartościowymi a typami referencyjnymi jest istotna, a zrozumienie tej różnicy uczyni cię bardziej efektywnym programistą.
- Stringi są bardziej użyteczne i wydajne, jeśli znasz ich wewnętrzną strukturę.
- Tablice są szybkie i wygodne, ale mogą nie być najlepszym wyborem dla publicznego interfejsu API.
- Listy są świetne do dynamicznego powiększania się, ale tablice są bardziej wydajne, jeśli nie planujesz dynamicznego zmieniania ich rozmiaru.
- Lista dwukierunkowa to specyficzna struktura danych, ale zrozumienie jej cech może pomóc w zrozumieniu kompromisów związanych z korzystaniem z słowników.
- Słowniki są doskonałe do szybkich wyszukiwań kluczy, ale ich wydajność zależy w dużej mierze od prawidłowej implementacji funkcji GetHashCode().
- Listę unikalnych wartości można reprezentować za pomocą HashSet, co zapewnia świetną wydajność podczas wyszukiwania.
- Stosy są doskonałymi strukturami danych do cofania się wstecz. Stos wywołań jest ograniczony.
- Zrozumienie działania stosu wywołań uzupełnia również aspekty wydajnościowe związane z typami wartościowymi i referencyjnymi.

Przypisy

1. NDA (Non-disclosure agreement) to umowa, która uniemożliwia pracownikom mówienie o swojej pracy, chyba że rozpoczną rozmowę od słów „Nie słyszałeś tego ode mnie, ale...”
2. Inspiracja do tego tekstu pochodziła z doskonałego komiksu xkcd o liczbach losowych: https://xkcd.com/221.

3 Przydatne Antywzorce

Rozdział ten obejmuje:

- Znane złe praktyki, które można wykorzystać w dobry sposób
- Antywzorce, które w rzeczywistości są przydatne
- Rozpoznawanie, kiedy stosować najlepsze praktyki, a kiedy ich złe odpowiedniki

Literatura programistyczna obfituje w najlepsze praktyki i wzorce projektowe. Niektóre z nich wydają się być niezaprzeczalne i mogą wywoływać zdziwienie, gdy ktoś zaczyna podważać ich skuteczność. Z czasem stają się one dogmatami, rzadko poddawanymi w wątpliwość. Okazjonalnie ktoś napisze na ich temat artykuł na blogu, a jeśli zdobędzie akceptację społeczności, na przykład na Hacker News, można uznać go za uzasadnioną krytykę i otworzyć drzwi dla nowych pomysłów. W przeciwnym razie nie można nawet ich omawiać. Gdybym miał przekazać jedną wiadomość świecie programowania, byłoby to wezwanie do kwestionowania wszystkich rzeczy, które są ci przekazywane - ich przydatności, sensu, korzyści i kosztów.

Dogmaty, czyli niezmienne prawa, tworzą obszary naszej ślepoty, a im dłużej się ich trzymamy, tym większe stają się te obszary. Mogą one przysłonić nam pewne użyteczne techniki, które w określonych przypadkach mogą być nawet bardziej przydatne.

Antywzorce, czyli złe praktyki, zasługują na swoją złą sławę, ale to nie znaczy, że powinniśmy ich unikać jak substancji radioaktywnych. W tekście zostaną przedstawione niektóre z tych wzorców, które mogą okazać się bardziej pomocne niż ich najlepsze praktyki. W ten sposób będziesz również korzystać z najlepszych praktyk i doskonałych wzorców projektowych, lepiej rozumiejąc, w jaki sposób one pomagają i kiedy nie są pomocne. Zobaczysz, co jest poza twoim polem widzenia i jakie skarby tam się kryją.

3.1 Jak to nie jest zepsute, to zepsuj

Jedną z pierwszych rzeczy, które nauczyłem się w firmach, w których pracowałem - oprócz lokalizacji toalet - było unikanie zmian w kodzie, nazywane również "code churn", za wszelką cenę. Każda zmiana, którą wprowadzasz, niesie ze sobą ryzyko stworzenia regresji, czyli błędu, który psuje już działający scenariusz. Błędy są już kosztowne, a ich naprawa zajmuje czas, zwłaszcza gdy stanowią część nowej funkcji. Jeśli chodzi o regresję, to jest to sytuacja gorsza niż wydanie nowej funkcji z błędami - to krok wstecz. Spudłowanie rzutu w koszykówce to błąd. Samodzielne zdobycie bramki w swojej bramce, efektywnie zdobycie punktu dla przeciwnika, to regresja. Czas jest najważniejszym zasobem w rozwoju oprogramowania, a utrata czasu ma najpoważniejsze konsekwencje. Regresje zabierają najwięcej czasu. Ma sens unikać regresji i unikać psucia kodu.

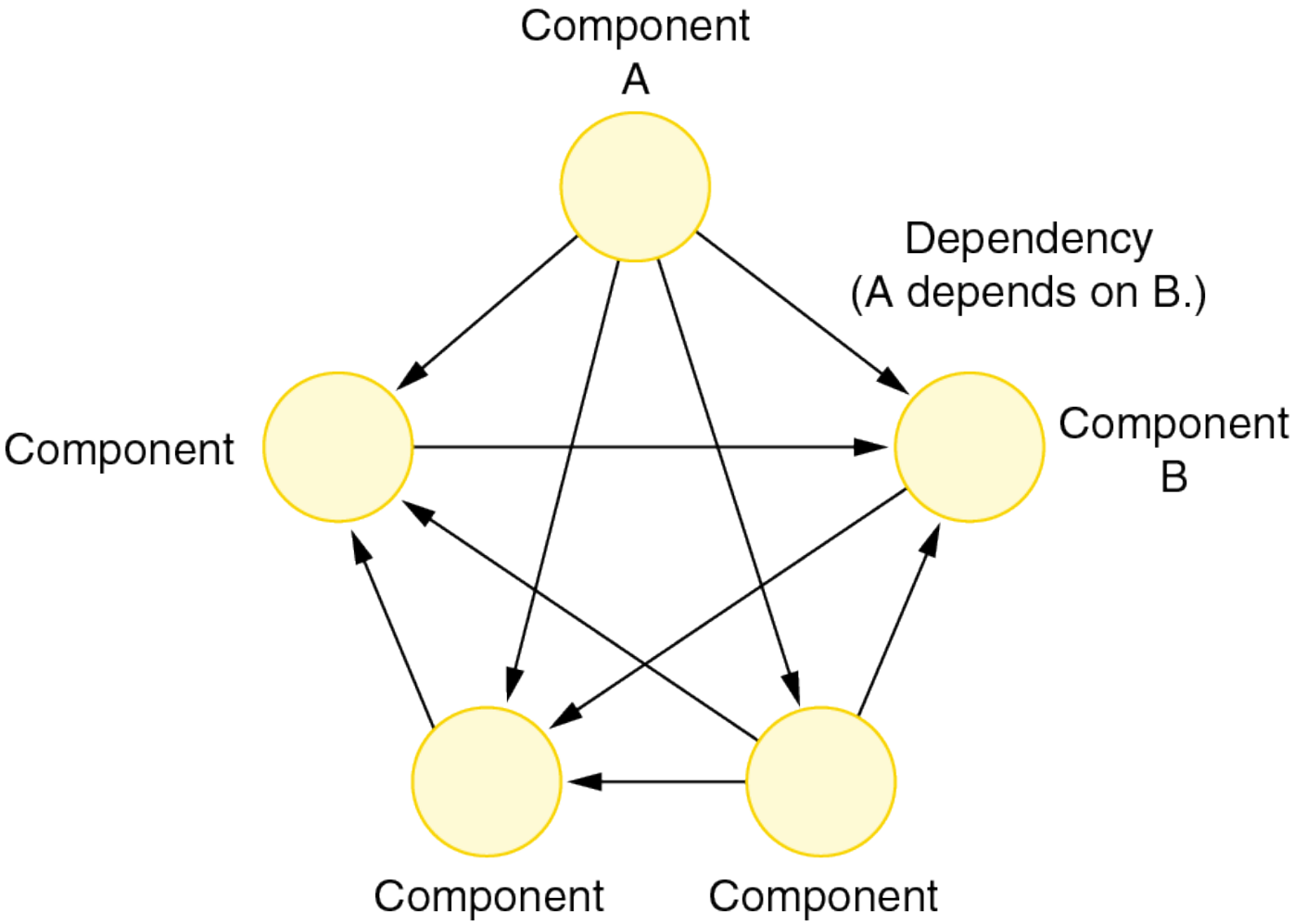
Unikanie zmian może jednak prowadzić do dylematu, ponieważ jeśli nowa funkcja wymaga, żeby coś było zepsute i ponownie naprawione, może to spotkać się z oporem podczas jej rozwoju. Możesz przyzwyczaić się do chodzenia na palcach wokół istniejącego kodu i próby dodawania wszystkiego w nowym kodzie, bez dotykania istniejącego kodu. Twoje starania o pozostawienie kodu nietkniętego mogą zmusić cię do tworzenia większej ilości kodu, co tylko zwiększa ilość kodu do utrzymania.

Jeśli musisz zmienić istniejący kod, to jest to większy problem. Tym razem nie ma możliwości skradania się. Modyfikacja istniejącego kodu może być bardzo trudna, ponieważ jest on ściśle związany z określonym sposobem działania, a jego zmiana wymusi zmiany w wielu innych miejscach. Odporność istniejącego kodu na zmiany nazywa się sztywnością kodu. Oznacza to, że im bardziej sztywny staje się kod, tym więcej kodu trzeba zepsuć, aby go manipulować.

3.1.1 Stawianie czoła sztywności kodu

Sztywność kodu opiera się na wielu czynnikach, a jednym z nich jest zbyt wiele zależności w kodzie. Zależność może odnosić się do różnych rzeczy: może dotyczyć zestawu narzędziowego, zewnętrznej biblioteki lub innej jednostki w twoim własnym kodzie. Wszelkiego rodzaju zależności mogą sprawiać problemy, jeśli twój kod się w nich zaplącze. Zależność może być zarówno błogosławieństwem, jak i przekleństwem. Rysunek 3.1 przedstawia fragment oprogramowania z okropnym grafem zależności. Narusza on granice zainteresowań, a każde zakłócenie w jednej z komponentów wymagałoby zmian w

prawie całym kodzie.



Dlaczego zależności powodują problemy? Kiedy rozważasz dodawanie zależności, rozważ także każdy komponent jako innego klienta lub każdą warstwę jako inny segment rynku o różnych potrzebach. Obsługa wielu segmentów klientów to większa odpowiedzialność niż obsługa tylko jednego rodzaju klienta. Klienci mają różne potrzeby, co może zmusić cię do niepotrzebnego zaspokajania różnych wymagań. Zastanów się nad tymi relacjami, gdy decydujesz się na łańcuchy zależności. Najlepiej starać się obsługiwać jak najmniejszą liczbę rodzajów klientów. To klucz do utrzymania twojego komponentu lub całej warstwy jak najprostszym.

Nie możemy unikać zależności. Są one niezbędne do ponownego użytku kodu. Ponowne użycie kodu to dwustronny kontrakt. Jeśli komponent A zależy od komponentu B, pierwsza klauzula brzmi: „B będzie świadczył usługi dla A”. Istnieje także druga klauzula, którą często pomija się: „A będzie przechodzić przez konserwację, kiedy B wprowadzi zmianę łamiącą zgodność”. Zależności wynikające z ponownego użycia kodu są akceptowalne, o ile możesz zachować uporządkowaną i skompartmentalizowaną strukturę łańcucha zależności.

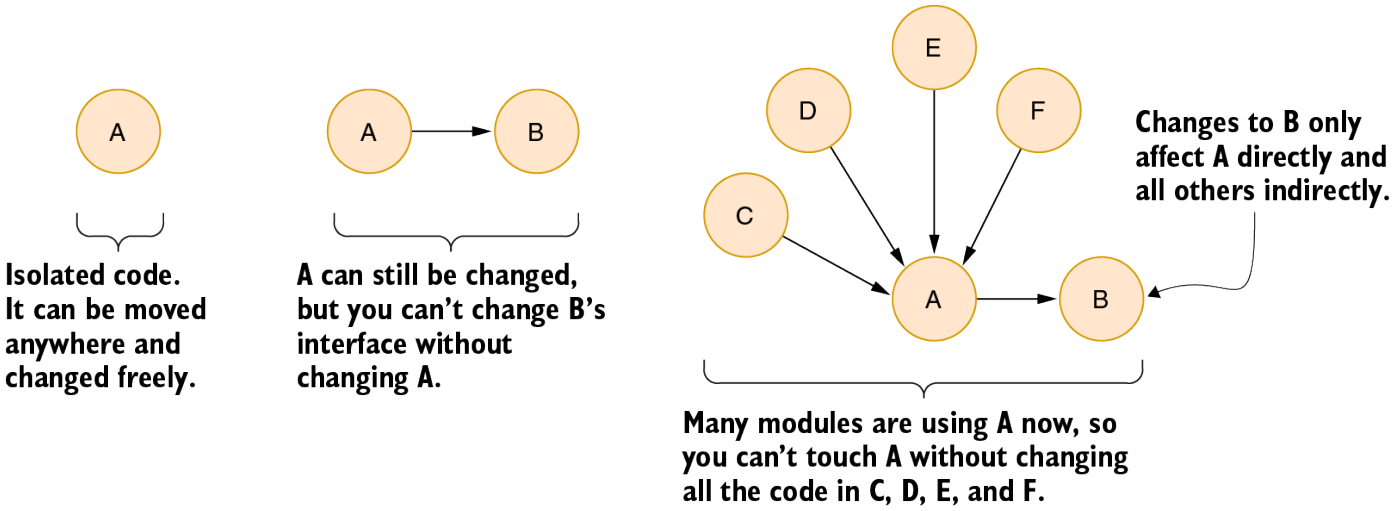
Dlaczego musisz zepsuć ten kod, czyli sprawić, że nie kompiluje się lub nie przechodzi testów? Ponieważ wzajemne zależności powodują sztywność w kodzie, która sprawia, że staje się on odporny na zmiany. To strome wzniesienie sprawi, że będziesz zwalniać z czasem, a ostatecznie zatrzymasz się. Łatwiej jest radzić sobie z problemami na początku, dlatego musisz zidentyfikować te problemy i zepsuć swój kod, nawet gdy

działa. Możesz zobaczyć, jak zależności wymuszają pewne działania na rysunku 3.2.

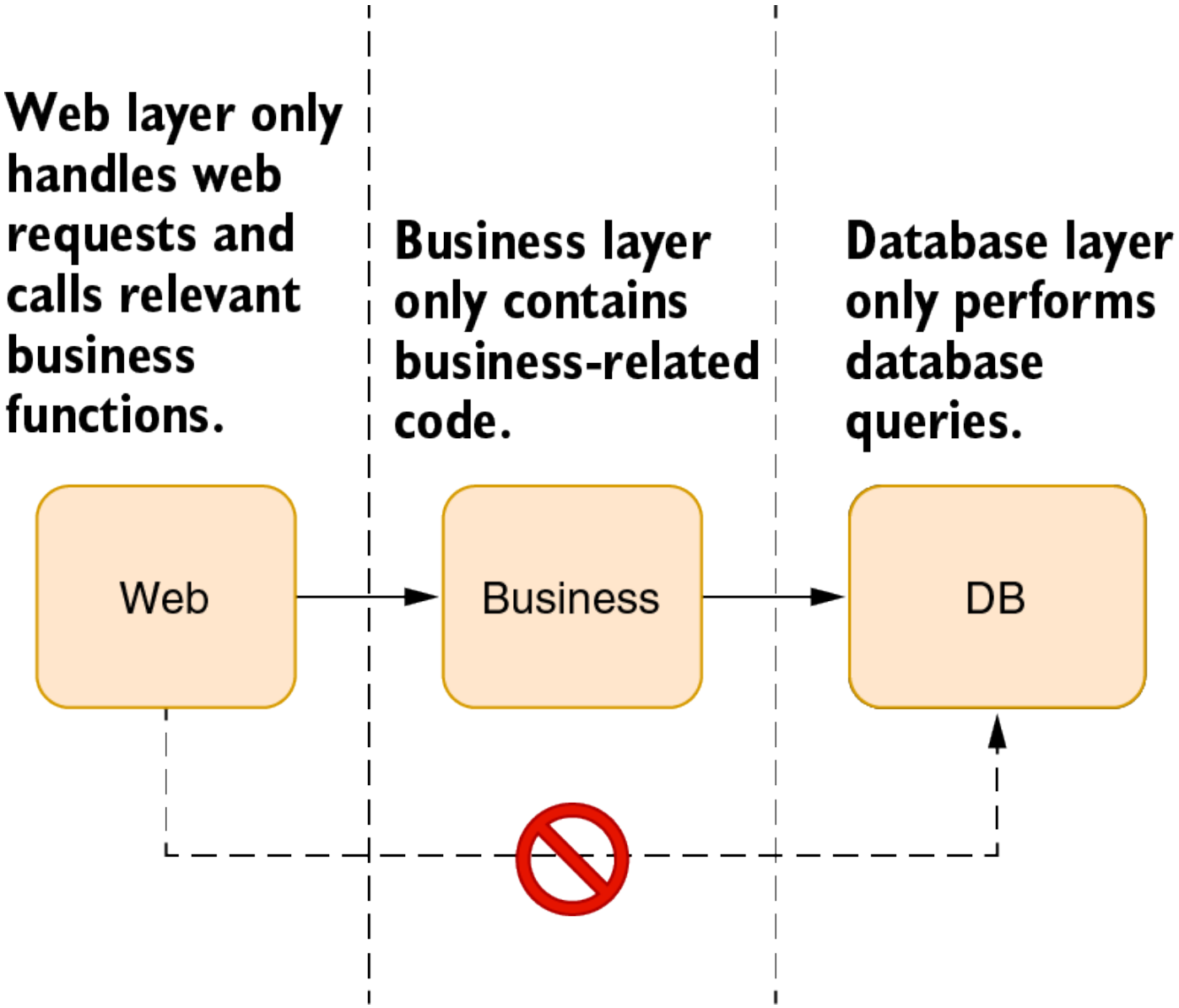
Komponent bez żadnych zależności jest najłatwiejszy do zmiany. Niemożliwe jest zepsucie czegokolwiek innego. Jeśli twój komponent zależy od jednego z twoich innych komponentów, powstaje pewna sztywność, ponieważ zależność oznacza umowę.

Jeśli zmienisz interfejs na B, oznacza to, że musisz zmienić także A. Jeśli zmienisz implementację B bez zmiany interfejsu, nadal możesz zepsuć A, ponieważ psujesz B. To staje się większym problemem, gdy masz wiele komponentów, które zależą od jednego komponentu.

Zmiana A staje się trudniejsza, ponieważ wymaga zmiany zależnego komponentu i niesie ryzyko zepsucia któregoś z nich. Programiści często zakładają, że im więcej kodu ponownie użyją, tym więcej czasu zaoszczędzą. Ale za jaką cenę? Musisz to wziąć pod uwagę.



Pierwszym nawykiem, który musisz przyjąć, jest unikanie naruszania granic abstrakcji dla zależności. Granicą abstrakcji jest logiczny obszar, którym otaczasz warstwy swojego kodu, zbiór zadań danej warstwy. Na przykład możesz mieć w swoim kodzie warstwy odpowiedzialne za interfejs sieciowy, logikę biznesową i bazę danych jako abstrakcje. Gdy układasz kod w warstwy, warstwa odpowiedzialna za bazę danych nie powinna wiedzieć o warstwie interfejsu sieciowego czy warstwie biznesowej, a warstwa interfejsu sieciowego nie powinna być świadoma warstwy bazy danych, jak pokazano na rysunku 3.3.



Dlaczego przekraczanie granic jest złą praktyką? Ponieważ eliminuje korzyści płynące z abstrakcji. Gdy przenosisz złożoność niższych warstw do wyższych, stajesz się odpowiedzialny za utrzymanie zmian wszędzie, także w niższych warstwach. Wyobraź sobie zespół, którego członkowie są odpowiedzialni za własne warstwy kodu. Nagle deweloper odpowiedzialny za warstwę interfejsu sieciowego musi nauczyć się SQL-a. Ponadto zmiany w warstwie bazy danych muszą być teraz komunikowane z większą liczbą osób, niż to jest konieczne. Obciąża to dewelopera zbędnymi obowiązkami. Czas potrzebny na osiągnięcie porozumienia wśród osób, które trzeba przekonać, wzrasta wykładniczo. Tracisz czas i tracisz wartość abstrakcji.

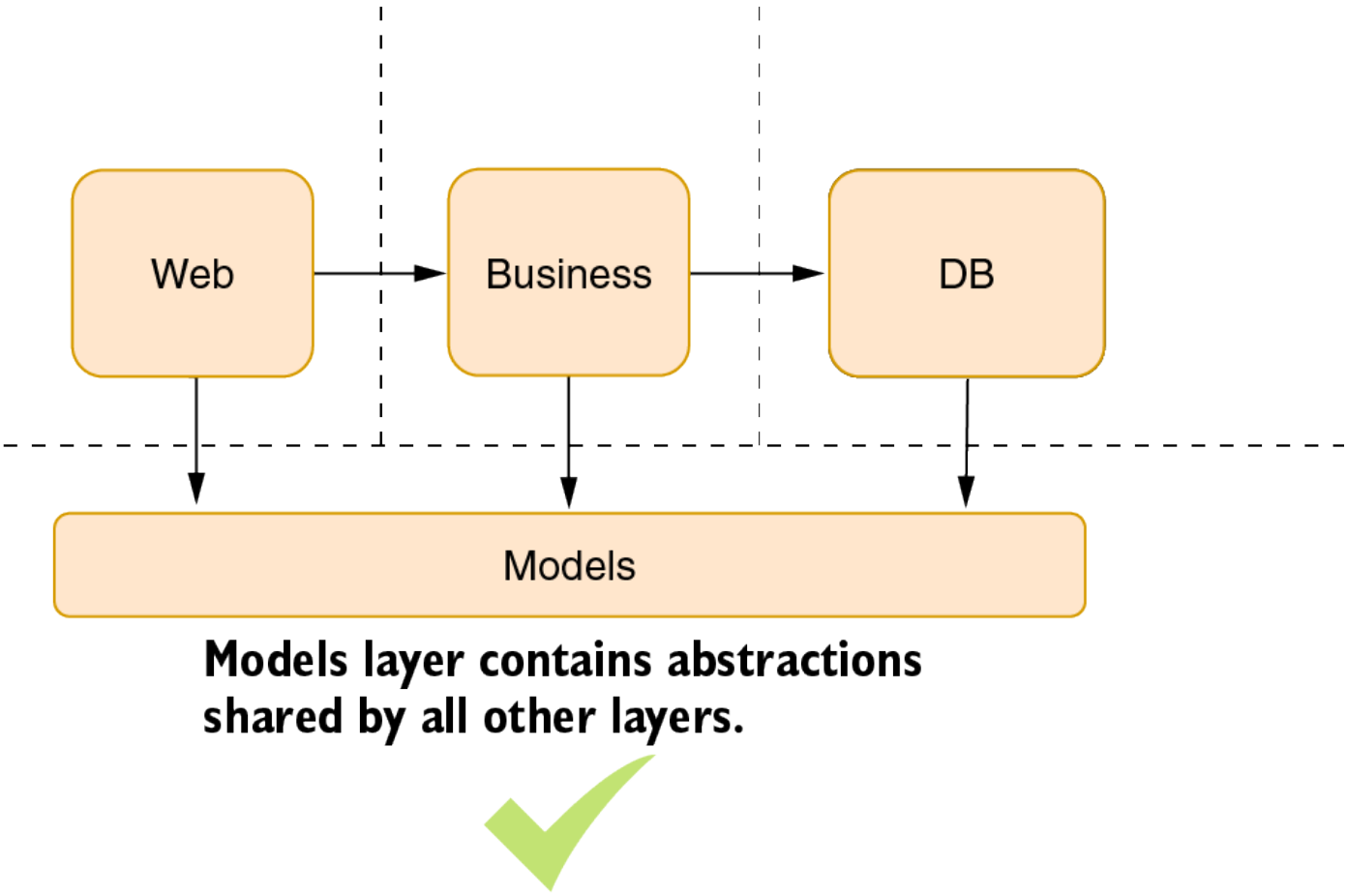
Jeśli napotkasz takie problemy z granicami, przejdź przez kod, czyli zdemontuj go tak, aby mógł przestać działać, usuń naruszenie, przeprowadź refaktoring kodu i zajmij się konsekwencjami. Napraw inne części kodu, które od niego zależą. Musisz być czujny na takie problemy i natychmiast je odciąć, nawet jeśli grozi to zepsuciem kodu. Jeśli kod sprawia, że boisz się go zepsuć, to jest źle zaprojektowany kod. To nie oznacza, że dobry kod nigdy się nie psuje, ale gdy tak się dzieje, łatwiej jest skleić jego fragmenty z powrotem.

Ważność testów

Musisz być w stanie ocenić, czy zmiana w kodzie może spowodować awarię scenariusza. Możesz polegać na własnym zrozumieniu kodu, ale Twoja skuteczność będzie maleć, gdy kod stanie się bardziej złożony w miarę upływu czasu.

Pod tym względem testy są prostsze. Testy mogą być listą instrukcji na kartce papieru lub mogą być w pełni zautomatyzowane. Testy zautomatyzowane są zazwyczaj preferowane, ponieważ piszesz je tylko raz i nie marnujesz czasu na ich ręczne wykonywanie. Dzięki frameworkom do testowania, ich napisanie jest również dość proste. Zagłębimy się głębiej w ten temat w rozdziale poświęconym testowaniu.

Czy oznacza to, że warstwa internetowa na rysunku 3.3 nigdy nie może mieć wspólnej funkcjonalności z bazą danych? Oczywiście, że może. Ale takie przypadki wskazują na potrzebę osobnego komponentu. Na przykład obie warstwy mogą polegać na wspólnych klasach modelu. W takim przypadku diagram związków wyglądałby jak ten pokazany na rysunku 3.4.



Przerabianie kodu może spowodować, że proces kompilacji się załamie lub że testy zawiodą, teoretycznie więc tego nigdy nie powinieneś robić. Ja jednak uważam takie naruszenia za ukryte awarie. Wymagają one natychmiastowej uwagi, a jeśli powodują większe uszkodzenia i więcej błędów w procesie, nie oznacza to, że zepsułeś działający kod: to oznacza, że błąd, który już istniał, teraz ujawnił się w sposób łatwiejszy do zrozumienia.

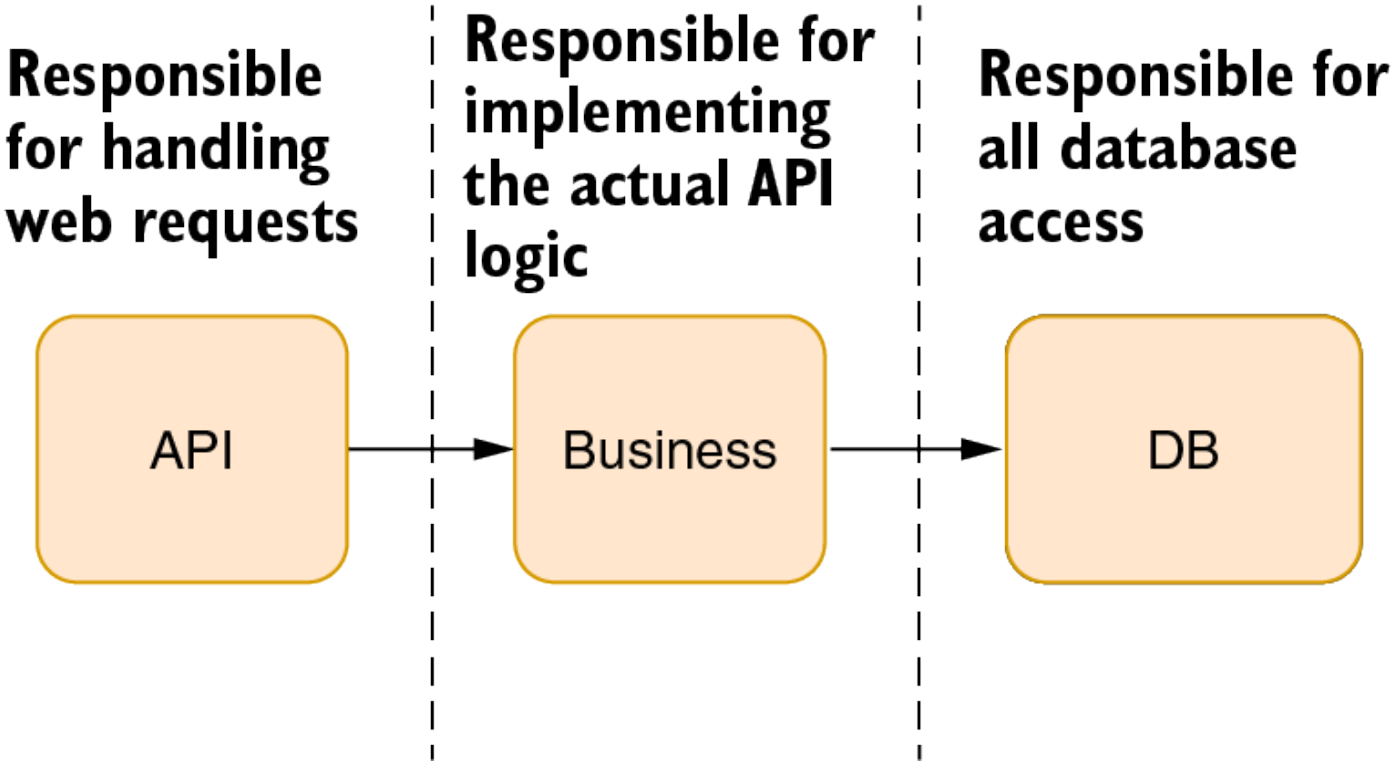
Przyjrzyjmy się przykładowi. Załóżmy, że piszesz interfejs API dla aplikacji czatowej, w której można komunikować się tylko za pomocą emotikon. Tak, brzmi okropnie, ale kiedyś istniała aplikacja czatowa, w której można było wysyłać tylko "Yo" jako wiadomość.[2] Nasza aplikacja jest przynajmniej nieco lepsza niż ta.

Projektujemy aplikację z warstwą internetową, która przyjmuje żądania od urządzeń mobilnych i wywołuje warstwę biznesową (nazywaną również warstwą logiczną), która wykonuje rzeczywiste operacje. Taki podział pozwala nam testować warstwę biznesową bez warstwy internetowej. Możemy również później użyć tej samej logiki biznesowej na innych platformach, takich jak strona mobilna. Dlatego rozdzielenie logiki biznesowej ma sens.

UWAGA

Słowo "biznes" w logice biznesowej lub warstwie biznesowej niekoniecznie oznacza coś związanego z biznesem, ale bardziej odnosi się do głównej logiki aplikacji z abstrakcyjnymi modelami. Można argumentować, że czytanie kodu warstwy biznesowej powinno dać ci pojęcie o tym, jak aplikacja działa w wyższym sensie.

Warstwa biznesowa nie ma pojęcia o bazach danych ani o technikach przechowywania danych. Do tego celu korzysta z warstwy bazy danych. Warstwa bazy danych kapsułkuje funkcjonalność bazy danych w sposób niezależny od konkretnej bazy danych. Taki podział odpowiedzialności ułatwia testowanie logiki biznesowej, ponieważ możemy łatwo podłączyć zmockowaną implementację warstwy przechowywania danych do warstwy biznesowej. Co ważniejsze, taka architektura pozwala nam zmieniać bazę danych za kulisami, nie zmieniając przy tym ani jednej linijki kodu w warstwie biznesowej, czy nawet w warstwie internetowej. Możesz zobaczyć, jak taki podział warstw wygląda na rysunku 3.5.



Wadą tego podejścia jest to, że za każdym razem, gdy dodajesz nową funkcję do interfejsu API, musisz utworzyć nową klasę lub metodę warstwy biznesowej oraz odpowiednią klasę i metody warstwy bazy danych. Wydaje się to być dużą pracą, zwłaszcza gdy mamy napięte terminy i funkcja jest stosunkowo prosta. "Dlaczego mam się bawić w to wszystko dla prostej zapytania SQL?" możesz pomyśleć. Niech więc spełni się fantazja wielu programistów i naruszmy istniejące abstrakcje.

3.1.5 Przykładowa strona web

Przyjmijmy, że otrzymałeś żądanie od swojego menedżera o dodanie nowej funkcji, nowej karty statystyk, która pokazuje, ile użytkownik wysłał i otrzymał wiadomości ogółem. To są tylko dwa proste zapytania SQL do wykonania po stronie serwera:

```
SELECT COUNT(*) as Sent FROM Messages WHERE FromId=@userId
SELECT COUNT(*) as Received FROM Messages WHERE ToId=@userId
```

Możesz uruchomić te zapytania w warstwie API. Nawet jeśli nie jesteś zaznajomiony z ASP.NET Core, rozwojem stron internetowych czy nawet samym SQL, nie powinieneś mieć problemów z zrozumieniem głównej idei kodu przedstawionego w przykładzie 3.1. Ten kod definiuje model do zwrócenia do aplikacji mobilnej. Model ten jest automatycznie serializowany do formatu JSON. Pobieramy ciąg połączenia z naszej bazy danych na serwerze SQL. Używamy tego ciągu do otwarcia połączenia, wykonania naszych zapytań do bazy danych i zwrócenia wyników.

Klasa StatsController przedstawiona w przykładzie 3.1 stanowi abstrakcję nad obsługą sieciową, w której otrzymane parametry zapytań są argumentami funkcji, a adres URL jest określany przez nazwę kontrolera, a wynik jest zwracany jako obiekt. Dlatego dostaniesz się do tego kodu za pomocą adresu URL takiego jak https://twojadomena/Stats/Get?userId=123, a infrastruktura MVC automatycznie mapuje parametry zapytania na parametry funkcji oraz zwrócony obiekt na rezultat JSON. Ułatwia to pisanie kodu obsługi sieci, ponieważ nie musisz faktycznie zajmować się adresami URL, ciągami zapytań, nagłówkami HTTP ani serializacją do formatu JSON.

Listing 3.1 przedstawia implementację funkcji poprzez naruszenie abstrakcji:

```
public class UserStats {
  public int Received { get; set; }
  public int Sent { get; set; }
}

public class StatsController: ControllerBase {
  public UserStats Get(int userId) {
    var result = new UserStats();
    string connectionString = config.GetConnectionString("DB");
    using (var conn = new SqlConnection(connectionString)) {
      conn.Open();
      var cmd = conn.CreateCommand();
      cmd.CommandText =
        "SELECT COUNT(*) FROM Messages WHERE FromId={0}";
      cmd.Parameters.Add(userId);
      result.Sent = (int)cmd.ExecuteScalar();
      cmd.CommandText =
        "SELECT COUNT(*) FROM Messages WHERE ToId={0}";
      result.Received = (int)cmd.ExecuteScalar();
    }
    return result;
  }
}
```

Prawdopodobnie zajęło mi to pięć minut napisanie tej implementacji. Wydaje się być prosta. Dlaczego więc mamy się przejmować abstrakcjami? Po prostu umieśćmy wszystko w warstwie API, prawda?

Takie rozwiązania mogą być akceptowalne podczas pracy nad prototypami, które nie wymagają doskonałego projektu. Ale w systemie produkcyjnym musisz być ostrożny, podejmując takie decyzje. Czy masz zgodę na wprowadzenie zmian w produkcji? Czy jest w porządku, jeśli strona zostanie wyłączona na kilka minut? Jeśli tak, to możesz śmiało używać tego podejścia. A co z twoim zespołem? Czy osoba odpowiedzialna za warstwę API jest zadowolona z tego, że takie zapytania SQL są rozproszone w różnych miejscach? A co z testowaniem? Jak przetestować ten kod i upewnić się, że działa poprawnie? Co z dodaniem

nowych pól? Spróbuj sobie wyobrazić, jak będą cię traktować ludzie w biurze następnego dnia. Czy widzisz ich ściskających cię w ramionach? Kibicujących ci? Czy może znajdziesz swoje biurko i krzesło ozdobione pinezkami?

Dodajesz zależność od fizycznej struktury bazy danych. Jeśli będziesz musiał zmienić układ tabeli Messages lub technologię bazy danych, będziesz musiał przejść przez cały kod i upewnić się, że wszystko działa z nową bazą danych lub nowym układem tabeli.

3.1.6 Nie pozostawiaj zadłużenia

My, programiści, nie jesteśmy dobrzy w przewidywaniu przyszłych wydarzeń i ich kosztów. Kiedy podejmujemy niekorzystne decyzje tylko po to, aby spełnić termin, utrudniamy spełnienie kolejnego ze względu na bałagan, który stworzyliśmy. Programiści często nazywają to długiem technicznym.

Długi techniczne są świadomymi decyzjami. Te nieświadome nazywane są nieudolnością techniczną. Są one nazywane długami, ponieważ albo spłacisz je później, albo kod przyjdzie do ciebie w nieprzewidzianej przyszłości i złamie ci nogi żelaznym kołem.

Istnieje wiele sposobów, w jakie dług techniczny może się nagromadzić. Może wydawać się łatwiej przekazać arbitralną wartość zamiast tworzyć dla niej stałą. "Ciąg znaków wydaje się działać tam dobrze", "Nie będzie szkody z tego, że skrócę nazwę", "Pozwól mi po prostu skopiować wszystko i zmienić niektóre jego części", "Wiem, użyję wyrażeń regularnych." Każda mała zła decyzja doda sekundy do twojej i twojego zespołu wydajności. Twój przepływ pracy będzie degradować się kumulacyjnie w czasie. Będziesz stawać się coraz wolniejszy, otrzymując mniej satysfakcji z pracy i mniej pozytywnych opinii od zarządzania. Będąc niewłaściwie leniwym, skazujesz się na porażkę. Bądź właściwie leniw: pracuj na przyszłe lenistwo.

Najlepszym sposobem na radzenie sobie z długiem technicznym jest odkładanie go na później. Czeka cię większe zadanie? Użyj tego jako okazji do rozruszania się. To może zepsuć kod. To dobrze – użyj tego jako szansy na zidentyfikowanie sztywnych części kodu, spraw, by stały się bardziej elastyczne. Spróbuj się zająć tym, zmień to, a następnie, jeśli uznasz, że nie działa wystarczająco dobrze, cofnij wszystkie swoje zmiany.

3.2 Zaczynaj od nowa

Jeśli zmiana kodu jest ryzykowna, napisanie go od nowa musi być o rząd wielkości bardziej ryzykowne. Oznacza to, że każdy nieprzetestowany scenariusz może być uszkodzony. Nie tylko oznacza to, że wszystko musi być napisane od nowa, ale także naprawiane od nowa, wszystkie błędy. Uważane jest za bardzo nieefektywną metodę naprawy wad projektu.

Jednak jest to prawdziwe tylko dla kodu, który już działa. Dla kodu, nad którym już pracowano, rozpoczęcie od nowa może być błogosławieństwem. Jak to możliwe, zapytasz? Wszystko to jest związane ze spiralą desperacji podczas pisania nowego kodu. To wygląda mniej więcej tak:

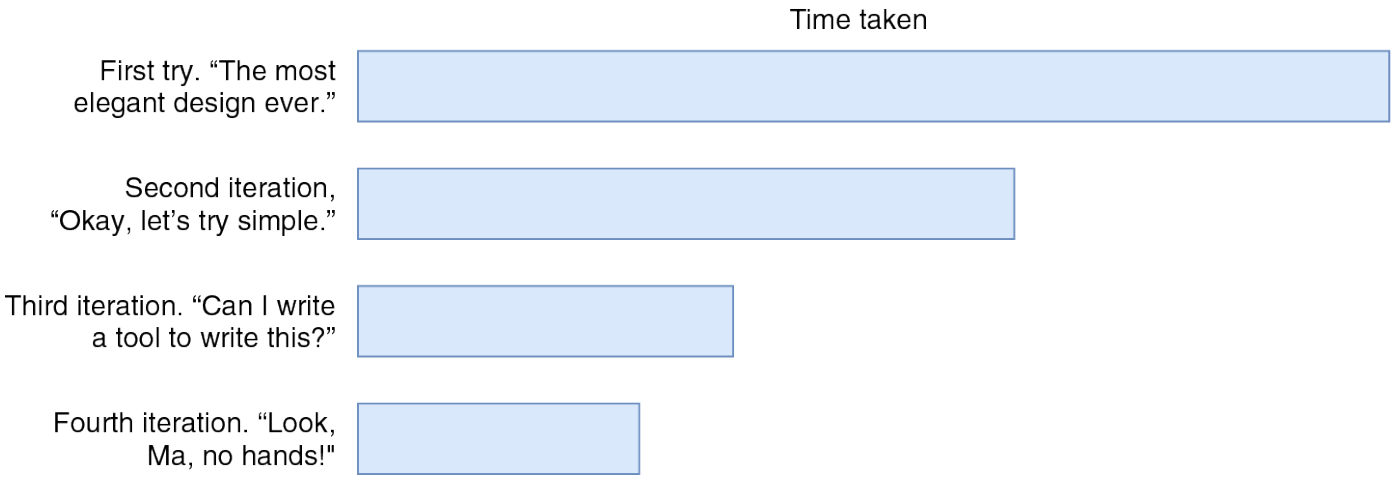
1. Zaczynasz od prostego i eleganckiego projektu.
2. Zaczniesz pisać kod.
3. Następnie pojawią się pewne przypadki brzegowe, o których nie pomyślałeś.
4. Zaczynasz przemyślać swój projekt.
5. Potem zauważasz, że aktualny projekt nie spełnia wymagań.
6. Zaczynasz ponownie dostosowywać projekt, ale unikasz jego ponownego napisania, ponieważ spowodowałoby to zbyt wiele zmian. Każda linijka gnębi cie coraz bardziej.

7. Twój projekt staje się teraz monstrum Frankenstein'a, w którym pomysły i kod są ze sobą zmieszane. Elegancja jest utracona, prostota jest utracona, a wszelka nadzieja jest utracona.

W tym momencie wpadasz w pętlę błędnego przekonania o koszcie. Czas, który już poświęciłeś swojemu istniejącemu kodowi, sprawia, że boisz się go zaczynać od nowa. Ale ponieważ nie może rozwiązać głównych problemów, poświęcasz dni na przekonywanie samego siebie, że projekt może działać. Być może uda ci się go naprawić w pewnym momencie, ale może cię to kosztować tygodnie, tylko dlatego, że zaszpachlowałeś się w dziurze.

3.2.1 Skasuj i przepisz

Ja mówię: zacznij od nowa, przepisz to. Wyrzuć wszystko, co już zrobiłeś, i napisz każdy fragment od nowa. Nie potrafisz sobie wyobrazić, jak odświeżające i szybkie może to być. Możesz myśleć, że przepisanie wszystkiego od nowa byłoby ogromnie nieefektywne i zajęłoby ci podwójnie więcej czasu, ale to nieprawda, ponieważ już to raz zrobiłeś. Już znasz rozwiązanie problemu. Korzyści z ponownego wykonania zadania przypominają coś takiego, jak to pokazano na rysunku 3.6.



Trudno przecenić zyski związane z prędkością, gdy robisz coś po raz drugi. W przeciwieństwie do przedstawianych w filmach hakerów, większość czasu spędzasz patrząc na ekran: nie piszesz rzeczy, ale myślisz o rzeczach, zastanawiasz się nad właściwym sposobem ich zrobienia. Programowanie to nie tyle tworzenie rzeczy, co poruszanie się po labiryncie skomplikowanego drzewa decyzyjnego. Kiedy zaczynasz od nowa, znasz już możliwe potknięcia, znasz pułapki, których należy unikać, oraz pewne projekty, do których doszedłeś w poprzedniej próbie.

Jeśli czujesz się zablokowany, próbując coś nowego, napisz to od nowa. Ja bym powiedział, że nawet nie zapisuj poprzedniej wersji swojej pracy, ale może chcesz to zrobić, jeśli nie jesteś pewien, czy zdołasz szybko to zrobić ponownie. No dobrze, zapisz kopię gdzieś, ale zapewniam cię, że większość czasu nie będziesz nawet musiał patrzeć na swoją poprzednią pracę. Jest już w twojej głowie, prowadząc cię znacznie szybciej, i bez wpadania w ten sam spiralny stan desperacji.

Co ważniejsze, kiedy zaczynasz od nowa, znasz wcześniej błędny kierunek w procesie niż wcześniej. Twoje radar pułapek zostanie zainstalowany już tym razem. Zdobędziesz wrodzony zmysł dla tego, jak rozwijać daną funkcję w odpowiedni sposób. Programowanie w ten sposób przypomina grę w gry konsolowe takie jak Marvel's Spider-Man czy The Last of Us. Nieustannie umierasz i zaczynasz tę sekwencję od nowa. Umierasz, odradzasz się. Z każdym powtórzeniem stajesz się lepszy, im więcej powtarzasz, tym lepiej stajesz

się w programowaniu. Pisanie od nowa poprawia twoje umiejętności rozwijania tej konkretnej funkcji, tak, ale także ulepsza twoje ogólne umiejętności programowania dla wszystkich przyszłych kodów, które będziesz pisać.

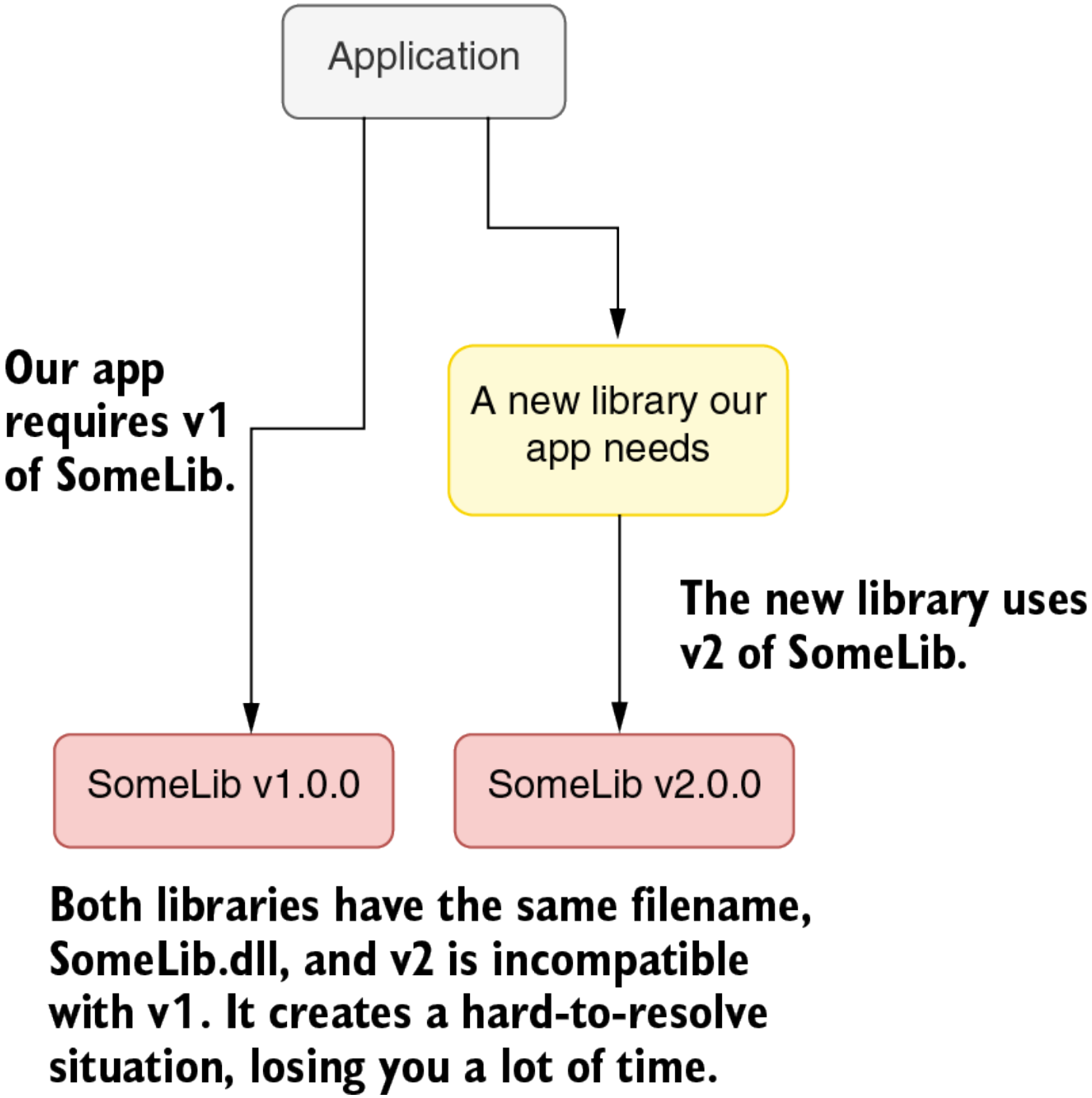
Nie wahaj się porzucić swojej pracy i zacznij pisać od nowa. Nie daj się zwieść błędowi kosztów utopionych.

3.3 Naprawiaj, nawet jeśli nie jest zepsute

Istnieją sposoby na radzenie sobie z sztywnością kodu, a jednym z nich jest regularne wprowadzanie zmian, aby nie stał się on zbyt sztywny—o ile chodzi o tę analogię. Dobry kod powinien być łatwy do zmiany i nie powinien wymagać zmiany w tysiącach miejsc, aby wprowadzić potrzebną zmianę. Pewne zmiany można przeprowadzić w kodzie, które nie są konieczne, ale mogą pomóc w dłuższej perspektywie. Możesz uczynić z tego regularny nawyk, dbając o to, aby utrzymywać aktualność swoich zależności, sprawiając, że Twoja aplikacja pozostaje płynna, oraz identyfikować najbardziej sztywne części, które są trudne do zmiany. Możesz również poprawiać kod jako swoistą aktywność ogrodniczą, regularnie dbając o drobne problemy w kodzie.

3.3.1 Wyścig w kierunku przyszłości

Niezwykle prawdopodobne jest, że będziesz używać jednego lub więcej pakietów z ekosystemu pakietów, które pozostawisz takimi, jakie są, ponieważ nadal działają dla Ciebie. Problem w tym, że gdy będziesz musiał użyć innego pakietu, który wymaga nowszej wersji Twojego pakietu, proces aktualizacji może być znacznie bardziej bolesny niż stopniowe aktualizowanie pakietów i pozostawanie na bieżąco. Taki konflikt możesz zobaczyć na rysunku 3.7.



Większość czasu, twórcy pakietów myślą tylko o scenariuszach aktualizacji między dwiema głównymi wersjami, a nie o wielu pośrednich wersjach. Na przykład popularna biblioteka Elasticsearch do wyszukiwania wymaga, aby aktualizacje wersji głównej były wykonywane pojedynczo; nie obsługuje bezpośredniej aktualizacji z jednej wersji na drugą.

.NET obsługuje przekierowania powiązań (binding redirects), aby uniknąć problemu wielu wersji tego samego pakietu, w pewnym stopniu. Przekierowanie powiązań to dyrektywa w konfiguracji aplikacji, która powoduje, że .NET przekierowuje wywołania do starszej wersji zestawu do jego nowszej wersji, lub odwrotnie. Oczywiście działa to tylko wtedy, gdy obie wersje pakietów są ze sobą kompatybilne. Zazwyczaj nie musisz samodzielnie zajmować się przekierowaniami powiązań, ponieważ Visual Studio może to zrobić za Ciebie, jeśli już zaznaczyłeś opcję Automatycznie generuj przekierowania powiązań w oknie właściwości projektu.

Okresowe aktualizowanie swoich pakietów będzie miało dwa ważne korzyści. Po pierwsze, rozłożysz wysiłek związany z aktualizacją do bieżącej wersji na okres utrzymania. Każdy krok będzie mniej bolesny. Po drugie, co ważniejsze, każda drobna aktualizacja może psuć Twój kod lub projekt w mały lub subtelny sposób, który będziesz musiał naprawić, aby przejść do przyszłości. Może to brzmieć niepożądanie, ale zmusi Cię do poprawy kodu i projektu stopniowo, pod warunkiem, że masz już testy na swoim miejscu.

Możesz mieć aplikację internetową, która używa Elasticsearch do operacji wyszukiwania i Newtonsoft.Json do analizy i generowania JSON. Są to jedne z najbardziej powszechnie używanych bibliotek. Problem pojawia się, gdy musisz zaktualizować pakiet Newtonsoft.Json, aby używać nowej funkcji, ale Elasticsearch używa starej wersji. Ale aby zaktualizować Elasticsearch, musisz także zmienić kod obsługujący Elasticsearch. Co zrobić?

Większość pakietów obsługuje tylko pojedyncze aktualizacje wersji. Elasticsearch, na przykład, oczekuje od Ciebie, że przeprowadzisz aktualizację z wersji 5 do 6, i ma wytyczne, jak to zrobić. Nie ma wytycznych dotyczących aktualizacji z wersji 5 do 7. Musisz stosować każdy poszczególny krok aktualizacji oddzielnie. Niektóre aktualizacje wymagają również znaczących zmian w kodzie. Elasticsearch 7 prawie sprawia, że musisz napisać kod od nowa.

Możesz pozostać w starszych wersjach, gdzie Twój kod pozostaje bez zmian, ale nie tylko wsparcie dla starszych wersji kończy się w pewnym momencie, ale także dokumentacja i przykłady kodu nie zostają na zawsze. Stack Overflow zapełnia się odpowiedziami dotyczącymi nowszych wersji, ponieważ ludzie używają najnowszej wersji, gdy zaczynają nowy projekt. Twoja sieć wsparcia dla starszej wersji zanika z czasem. To sprawia, że aktualizacja staje się trudniejsza z każdym kolejnym rokiem, co popycha Cię w dół spiralę desperacji.

Moje rozwiązanie tego problemu polega na dołączeniu do wyścigu ku przyszłości. Trzymaj biblioteki aktualne. Stań się nawykiem regularnie aktualizować biblioteki. To spowoduje okresowe problemy w twoim kodzie, a dzięki temu dowiesz się, który fragment kodu jest bardziej kruchy, i będziesz mógł zwiększyć pokrycie testami.

Kluczowym pomysłem jest to, że aktualizacje mogą powodować awarie w twoim kodzie, ale pozwalając im na mikropauzy, zapobiegniesz ogromnym przeszkodom, które stają się naprawdę trudne do pokonania. Inwestujesz nie tylko w fikcyjne przyszłe korzyści, ale także w elastyczność zależności twojej aplikacji, pozwalając jej się psuć i naprawiać tak, aby nie łamała się tak łatwo przy kolejnej zmianie, niezależnie od aktualizacji pakietów. Im mniej oporny jest twój kod na zmiany, tym lepiej pod względem projektowania i łatwości utrzymania.

3.3.2 Czystość bliska kodowaniu

To, co najbardziej polubiłem w komputerach, to ich determinizm. To, co napisałeś, zawsze będzie działo się tak samo, gwarantowane. Kod, który działa, zawsze będzie działał. W tym znajdowałem pocieszenie. Ale jak naiwny byłem. W mojej karierze widziałem wiele przypadków błędów, które można było zaobserwować tylko okazjonalnie, w zależności od prędkości procesora lub pory dnia. Pierwszą prawdą uliczną jest: "Wszystko się zmienia". Twój kod się zmieni. Wymagania się zmienią. Dokumentacja się zmieni. Środowisko się zmieni. Niemożliwe jest utrzymanie stabilnego kodu, nie ruszając go.

Skoro już to przekuliśmy, możemy się zrelaksować i powiedzieć, że nic nie szkodzi ruszyć kod. Nie powinniśmy bać się zmian, ponieważ i tak się zdarzą. Oznacza to, że nie powinieneś wahać się nad poprawą działającego kodu. Poprawki mogą być niewielkie: dodanie niezbędnych komentarzy, usunięcie zbędnych, lepsze nazewnictwo rzeczy. Utrzymuj kod przy życiu. Im więcej zmian wprowadzisz w kodzie, tym mniej oporny stanie się on na przyszłe zmiany. To dlatego, że zmiany spowodują awarie, a awarie pozwolą ci zidentyfikować słabe punkty i uczynić je bardziej zarządzalnymi. Powinieneś zrozumieć, jak i gdzie twój kod może ulec awarii. Ostatecznie zyskasz wrodzoną zdolność oceny, jaki rodzaj zmiany będzie najmniej ryzykowny.

Można nazwać ten rodzaj aktywności poprawiającej kod "ogrodnictwem". Niekoniecznie dodajesz funkcji ani nie naprawiasz błędów, ale kod powinien być lekko ulepszony po zakończeniu pracy nad nim. Taka zmiana może pozwolić następnemu programiście, który odwiedzi kod, lepiej go zrozumieć lub poprawić pokrycie testowe kodu, jakby ktoś zostawił prezenty na noc lub jakby bonsai w biurze tajemniczo ożyło.

Dlaczego więc miałbyś trudzić się przy zadaniu, które nigdy nie będzie docenione przez nikogo w twojej karierze? Idealnie byłoby, gdyby było to doceniane i wynagradzane, ale nie zawsze jest to możliwe. Nawet możesz spotkać się z krytyką ze strony swoich kolegów, ponieważ może im się nie podobać zmiana, którą wprowadziłeś. Nawet możesz zakłócić ich pracę, nie łamiąc kodu. Możesz przekształcić go w gorszy projekt niż pierwotnie zamierzał to zrobić oryginalny programista, podczas gdy próbujesz go ulepszyć.

Tak, i to jest oczekiwane. Jedynym sposobem na dojrzałość w kwestii radzenia sobie z kodem jest jego wielokrotne zmienianie. Upewnij się, że twoje zmiany są łatwo odwracalne, więc w przypadku zaniepokojenia kogoś, możesz cofnąć swoje zmiany. Nauczysz się również, jak komunikować się z kolegami na temat zmian, które mogą ich dotyczyć. Dobra komunikacja to najważniejsza umiejętność, którą możesz rozwijać w programowaniu.

Największą korzyścią z drobnych poprawek kodu jest to, że szybko wkraczasz w tryb programowania. Duże zadania są najcięższymi mentalnymi ciężarami. Zazwyczaj nie wiesz, od czego zacząć i jak się zająć tak dużą zmianą. Pesymizm typu "O, to będzie takie trudne, że będę musiał to tylko znosić" sprawia, że odkładasz rozpoczęcie projektu. Im bardziej to odkładasz, tym bardziej będziesz obawiać się kodowania.

Drobne ulepszenia kodu są sztuczką, aby szybko wkręcić swoje umysłowe koła, dzięki czemu możesz się wystarczająco rozgrzać, aby poradzić sobie z większym problemem. Ponieważ już kodujesz, twoje myśli przeciwstawiają się przełączaniu się na inne tryby mniej niż w przypadku próby przejścia od przeglądania mediów społecznościowych do kodowania. Odpowiednie części twojego mózgu zostały już uruchomione i są gotowe na większy projekt.

Jeśli nie możesz znaleźć nic do poprawy, możesz skorzystać z analizatorów kodu. To świetne narzędzia do znajdowania drobnych problemów w kodzie. Upewnij się, że dostosowujesz opcje używanego przez ciebie analizatora kodu, aby unikać jak największych kontrowersji. Porozmawiaj z kolegami, co o tym myślą. Jeśli uważają, że nie chce im się naprawiać problemów, obiecaj im, że naprawisz pierwszą partię sam i wykorzystaj to jako okazję do rozgrzewki. W przeciwnym razie możesz użyć alternatywy w wierszu poleceń lub własnych funkcji analizy kodu w programie Visual Studio do uruchamiania analizy kodu, nie łamiąc wytycznych zespołu dotyczących kodowania.

Nawet nie musisz stosować wprowadzonych zmian, ponieważ służą one tylko do rozgrzewki przed kodowaniem. Na przykład możesz być niepewny, czy możesz zastosować określoną poprawkę, może wydawać się ryzykowna, ale zrobiłeś już tak wiele. Ale jak już się nauczyłeś, po prostu to odrzuć. Zawsze możesz zacząć od nowa i zrobić to jeszcze raz. Nie martw się zbytnio o odrzucanie swojej pracy. Jeśli jesteś bardzo zainteresowany, zrób kopię zapasową, ale naprawdę bym się tym nie martwił.

Jeśli wiesz, że twoja drużyna będzie zadowolona z wprowadzonych zmian, opublikuj je. Satysfakcja z poprawy, choćby najmniejszej, może cię zmotywować do wprowadzenia większych zmian.

3.4 Nie powtarzaj się

Powielanie i programowanie metodą kopiuj-wklej to koncepcje, które są patrzone z goryczą w środowisku rozwoju oprogramowania. Jak każda zdrowa zasada, stały się one ostatecznie jakaś formą religii, co powoduje cierpienie ludzi.

Teoria jest taka: napisujesz kawałek kodu. Potrzebujesz tego samego kawałka kodu w innym miejscu w programie. Zachętą początkującego programisty byłoby po prostu skopiowanie i wklejenie tego samego kodu i użycie go. Na razie wszystko jest w porządku. Następnie znajdujesz błąd w skopiowanym kodzie. Teraz musisz zmienić kod w dwóch oddzielnych miejscach. Musisz trzymać je w synchronizacji. To stworzy więcej pracy i sprawi, że nie dotrzymasz terminów.

Brzmi logicznie, prawda? Zazwyczaj rozwiązaniem tego problemu jest umieszczenie kodu w wspólnym pliku klasy lub module i używanie go w obu częściach programu. Więc, kiedy zmieniasz wspólny kod, zmieniasz go magicznie wszedzie tam, gdzie jest używany, co pozwala zaoszczędzić wiele czasu.

Na razie wszystko jest w porządku, ale to nie trwa wiecznie. Problemy zaczynają się pojawiać, kiedy zaczynasz stosować tą zasadę do wszystkiego, co możliwe, i to w niewłączony sposób. Jedna drobna rzecz, którą pomijasz, próbując przerobić kod na ponownie używalne klasy, polega na tym, że tworzysz nowe zależności, a zależności wpływają na twój projekt. Czasami nawet mogą cię zmusić do określonego podejścia.

Największy problem z wspólnymi zależnościami polega na tym, że części oprogramowania, które używają wspólnego kodu, mogą się różnić w swoich wymaganiach. Kiedy to się zdarzy, odruchem programisty jest dostosowanie się do różnych potrzeb przy użyciu tego samego kodu. Oznacza to dodawanie opcjonalnych parametrów, logiki warunkowej, aby upewnić się, że wspólny kod może obsługiwać dwa różne wymagania. To sprawia, że właściwy kod staje się bardziej skomplikowany, co ostatecznie powoduje więcej problemów, niż rozwiązuje. W pewnym momencie zaczynasz myśleć o bardziej skomplikowanym projekcie niż kod kopiowany i wklejany.

Rozważmy przykład, w którym masz za zadanie napisać interfejs API dla sklepu internetowego. Klient chce zmienić adres dostawy dla klienta, który jest reprezentowany przez klasę o nazwie PostalAddress, wyglądającą tak:

```
public class PostalAddress {
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Address1 { get; set; }
    public string Address2 { get; set; }
    public string City { get; set; }
    public string ZipCode { get; set; }
    public string Notes { get; set; }
}
```

Musisz zastosować pewne normalizacje do pól, takie jak kapitalizacja, aby wyglądały one odpowiednio nawet wtedy, gdy użytkownik nie poda prawidłowych danych wejściowych. Funkcja aktualizacji mogłaby wyglądać jak sekwencja operacji normalizacji i aktualizacji w bazie danych:

```
public void SetShippingAddress(Guid customerId, PostalAddress newAddress) {
    normalizeFields(newAddress);
    db.UpdateShippingAddress(customerId, newAddress);
}

private void normalizeFields(PostalAddress address) {
    address.FirstName = TextHelper.Capitalize(address.FirstName);
    address.LastName = TextHelper.Capitalize(address.LastName);
    address.Notes = TextHelper.Capitalize(address.Notes);
}
```

Nasza metoda `Capitalize` działałaby przez zamienianie pierwszego znaku na wielką literę, a resztę ciągu na małe litery.

Teraz wydaje się, że działa to dla notatek i imion: "gunyuz" staje się "Gunyuz", a "PLEASE LEAVE IT AT THE DOOR" staje się "Proszę zostawić to przy drzwiach", co oszczędza nieco nerwów dostawcy. Po uruchomieniu aplikacji na pewien czas chcesz również znormalizować nazwy miast. Dodajesz to do funkcji `normalizeFields`:

```
address.City = TextHelper.Capitalize(address.City);
```

Jak dotąd wszystko jest w porządku, ale gdy zaczynasz otrzymywać zamówienia z San Francisco, zauważasz, że są one znormalizowane jako "San francisco." Teraz musisz zmienić logikę funkcji kapitalizującej, aby kapitalizować każde słowo, więc nazwa miasta staje się "San Francisco." To pomoże również w przypadku imion dzieci Elona Muska. Ale potem zauważasz, że notatka dostawy staje się "Please Leave It At The Door." To jest lepsze niż całe zdanie w dużych literach, ale szef chce, żeby było idealnie. Co zrobić?

Najłatwiejszą zmianą, która wprowadza najmniejsze zmiany w kodzie, wydaje się być zmiana funkcji `Capitalize`, aby przyjmowała dodatkowy parametr dotyczący zachowania. Kod w poniższym listingu 3.2 przyjmuje dodatkowy parametr o nazwie `everyWord`, który określa, czy ma kapitalizować każde słowo czy tylko pierwsze słowo. Proszę zauważyć, że nie nazwałeś tego parametru `isCity` lub czegoś w tym stylu, ponieważ to, do czego go używasz, nie jest problemem funkcji `Capitalize`. Nazwy powinny wyjaśniać rzeczy w kontekście, w którym są używane, a nie wywołującego. W każdym razie dzielisz tekst na słowa, jeśli `everyWord` jest prawdziwe, i kapitalizujesz każde słowo osobno, wywołując siebie dla każdego słowa, a następnie łączysz słowa z powrotem w nowy ciąg znaków.

```
public static string Capitalize(string text,
    bool everyWord = false) {
    if (text.Length < 2) {
      return text;
    }
    if (!everyWord) {
      return Char.ToUpper(text[0]) + text.Substring(1).ToLower();
```

```
    }
    string[] words = text.Split(' ');
    for (int i = 0; i < words.Length; i++) {
      words[i] = Capitalize(words[i]);
    }
    return String.Join(" ", words);
  }
```

Już teraz zaczyna to wyglądać na skomplikowane, ale proszę wytrzymaj ze mną - naprawdę chcę, abyś się przekonał o tym. Zmiana zachowania funkcji wydaje się być najprostszym rozwiązaniem. Po prostu dodajesz parametr i stosujesz tu i ówdzie instrukcje warunkowe, i gotowe. To tworzy złą nawyk, niemal odruch, aby każdą małą zmianę traktować w ten sposób, co może stworzyć ogromną ilość złożoności.

Załóżmy, że potrzebujesz również kapitalizacji dla nazw plików do pobrania w swojej aplikacji, i już masz funkcję, która poprawia wielkość liter, więc potrzebujesz tylko kapitalizacji nazw plików i ich oddzielenia podkreśleniem. Na przykład, jeśli API otrzyma raport faktury, powinien stać się Invoice_Report. Ponieważ już masz funkcję kapitalizacji, twoim pierwszym instynktem będzie znowu nieznaczna modyfikacja jej zachowania. Dodajesz nowy parametr o nazwie `filename`, ponieważ dodawane zachowanie nie ma bardziej ogólnej nazwy, i sprawdzasz ten parametr w miejscach, gdzie ma to znaczenie. Przy konwersji na wielkie i małe litery musisz używać funkcji ToUpper i ToLower w wersjach z invariant culture, aby nazwy plików na tureckich komputerach nagle nie stały się İnvoice_Report. Zauważ kropkę nad literą "I" w İnvoice_Report? Nasza implementacja teraz wyglądałaby tak, jak pokazano poniżej.

```
public static string Capitalize(string text,
    bool everyWord = false, bool filename = false) {
    if (text.Length < 2) {
      return text;
    }
    if (!everyWord) {
      if (filename) {
        return Char.ToUpperInvariant(text[0])
          + text.Substring(1).ToLowerInvariant();
      }
      return Char.ToUpper(text[0]) + text.Substring(1).ToLower();
    }
    string[] words = text.Split(' ');
    for (int i = 0; i < words.Length; i++) {
      words[i] = Capitalize(words[i]);
    }
    string separator = " ";
    if (filename) {
      separator = "_";
    }
    return String.Join(separator, words);
  }
```

Spójrz, jakie monstrum stworzyłeś. Naruszyłeś swoją zasadę dotyczącą krzyżujących się zagadnień i sprawiłeś, że twoja funkcja `Capitalize` stała się świadoma twoich konwencji dotyczących nazw plików. Nagle stała się częścią konkretnej logiki biznesowej, zamiast pozostać ogólna. Tak, maksymalnie wykorzystujesz kod wielokrotnego użytku, ale utrudniłeś sobie pracę w przyszłości.

Zauważ, że stworzyłeś nowy przypadek, który nawet nie był uwzględniony w twoim projekcie: nowy format nazwy pliku, w którym nie wszystkie słowa są zapisane wielkimi literami. Jest to widoczne poprzez warunek, gdzie `everyWord` jest fałszywe, a `filename` jest prawdziwe. Nie zamierzałeś tego, ale teraz to masz. Inny programista może polegać na tym zachowaniu, i to właśnie sprawia, że twój kod staje się z czasem chaosem.

Proponuję czystsze podejście: nie bój się powtarzać. Zamiast próbować scalenia każdego drobiazgu logicznego w ten sam kod, staraj się mieć osobne funkcje, które być może mają nieco powtarzający się kod. Możesz mieć oddzielne funkcje dla każdego przypadku użycia. Możesz mieć jedną, która kapitalizuje tylko pierwszą literę, inną, która kapitalizuje każde słowo, i jeszcze inną, która faktycznie formatuje nazwę pliku. Nawet nie muszą one być obok siebie - kod dotyczący nazwy pliku może pozostać bliżej logiki biznesowej, dla której jest wymagany. Zamiast tego masz te trzy funkcje, które lepiej przekazują swoje intencje. Pierwsza nosi nazwę `CapitalizeFirstLetter`, więc jej funkcja jest jasna. Druga to `CapitalizeEveryWord`, co też lepiej wyjaśnia, co robi. Wywołuje `CapitalizeFirstLetter` dla każdego słowa, co jest znacznie łatwiejsze do zrozumienia niż próba zastanawiania się nad rekurencją. Na koniec masz `FormatFilename`, która ma zupełnie inną nazwę, ponieważ kapitalizacja to nie jedyna rzecz, którą robi. Zawiera całą logikę kapitalizacji zaimplementowaną od podstaw. Pozwala to swobodnie modyfikować funkcję, gdy twoje zasady formatowania nazw plików się zmieniają, bez konieczności zastanawiania się, jak wpłynie to na pracę z wielkością liter, jak pokazano poniżej.

```
public static string CapitalizeFirstLetter(string text) {
   if (text.Length < 2) {
     return text.ToUpper();
   }
   return Char.ToUpper(text[0]) + text.Substring(1).ToLower();
 }

 public static string CapitalizeEveryWord(string text) {
   var words = text.Split(' ');
   for (int n = 0; n < words.Length; n++) {
     words[n] = CapitalizeFirstLetter(words[n]);
   }
   return String.Join(" ", words);
 }

 public static string FormatFilename(string filename) {
   var words = filename.Split(' ');
   for (int n = 0; n < words.Length; n++) {
     string word = words[n];
     if (word.Length < 2) {
       words[n] = word.ToUpperInvariant();
     } else {
       words[n] = Char.ToUpperInvariant(word[0]) +
         word.Substring(1).ToLowerInvariant();
     }
   }
```

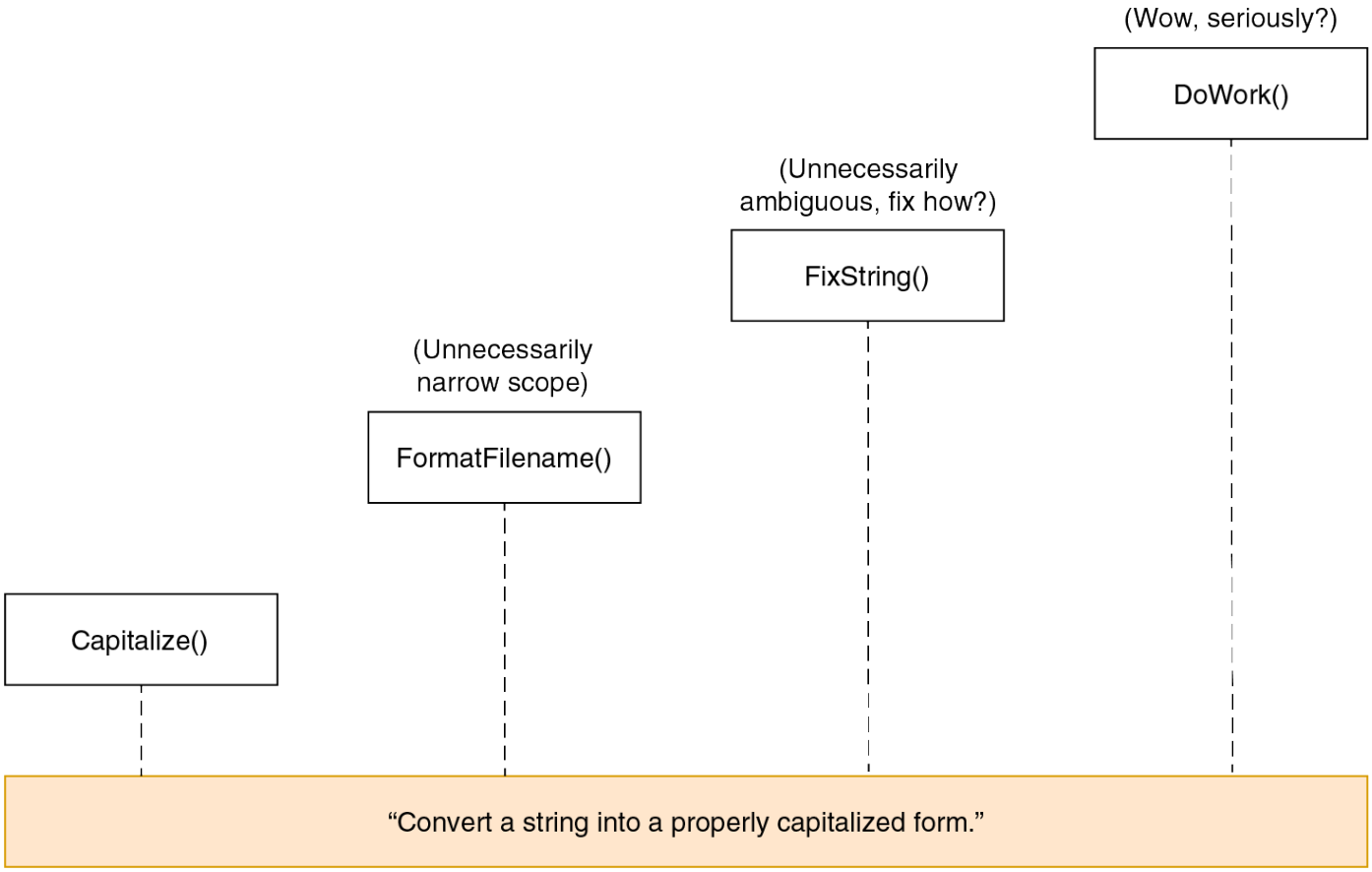
```
27      return String.Join("_", words);
28  }
```

W ten sposób nie będziesz musiał wciskać każdego możliwego fragmentu logiki do jednej funkcji. To staje się szczególnie istotne, gdy wymagania różnią się między użytkownikami.

3.4.1 Powtórne użycie czy kopiowanie?

Jak decydujesz między ponownym użyciem kodu a jego replikacją w innym miejscu? Największym czynnikiem jest to, jak opisujesz obawy użytkownika, czyli opisanie wymagań użytkownika tak, jak są one naprawdę. Gdy opisujesz wymagania funkcji, w której nazwa pliku musi być sformatowana, jesteś stronniczy ze względu na istnienie funkcji, która jest dość bliska temu, co chcesz zrobić (kapitalizacja), co natychmiast sygnalizuje twojemu umysłowi, aby użyć tej istniejącej funkcji. Jeśli nazwa pliku miałaby być kapitalizowana dokładnie tak samo, może to mieć sens, ale różnice w wymaganiach powinny być sygnałem alarmowym.

W informatyce trzy rzeczy są trudne: nieprawidłowe nazywanie, nazywanie rzeczy i błędy o jeden.3 Prawidłowe nazywanie rzeczy jest jednym z najważniejszych czynników podczas rozumienia konfliktujących wymagań w ponownym użyciu kodu. Nazwa `Capitalize` prawidłowo określa funkcję. Mogliśmy nazwać ją `NormalizeName`, gdy ją pierwszy raz tworzyliśmy, ale uniemożliwiłoby nam to ponowne użycie jej w innych polach. To, co zrobiliśmy, to nazwanie rzeczy tak blisko, jak to tylko możliwe, ich rzeczywistej funkcjonalności. W ten sposób nasza funkcja może obsługiwać różne cele bez wprowadzania zamieszania, a co ważniejsze, lepiej wyjaśnia swoją pracę, gdziekolwiek jest używana. Możesz zobaczyć, jak różne podejścia do nazywania wpływają na opis rzeczywistego zachowania na rysunku 3.8.



Moglibyśmy zagłębić się w rzeczywistą funkcjonalność, taką jak "Ta funkcja zamienia pierwsze litery każdego słowa w ciągu na wielkie litery i pozostałe litery zamienia na małe litery", ale to trudno zmieścić w nazwie. Nazwy powinny być jak najkrótsze i jak najbardziej jednoznaczne. Nazwa `Capitalize` spełnia te wymagania.

Świadomość obaw związanych z danym fragmentem kodu to ważna umiejętność. Zazwyczaj przypisuję funkcjom i klasom osobowości, aby sklasyfikować ich obawy. Mówię na przykład: "Ta funkcja się tym nie przejmuje", jakby to była osoba. W ten sposób możesz zrozumieć obawy danego fragmentu kodu. Dlatego nazwaliśmy parametr do kapitalizacji każdego słowa `everyWord`, a nie `isCity`, ponieważ funkcja po prostu nie przejmuje się tym, czy to miasto, czy nie. To nie jest troska funkcji.

Kiedy nazywasz rzeczy bliżej ich kręgu zainteresowań, ich wzorce użycia stają się bardziej widoczne. Dlaczego więc nazwaliśmy funkcję formatującą nazwę pliku `FormatFilename`? Czy nie powinniśmy nazwać jej `CapitalizeInvariantAndSeparateWithUnderscores`? Nie. Funkcje mogą wykonywać wiele rzeczy, ale wykonują tylko jedno zadanie, i powinny być nazwane zgodnie z tym zadaniem. Jeśli czujesz potrzebę użycia spójników "i" lub "lub" w nazwie funkcji, albo źle ją nazwałeś, albo nadajesz jej zbyt wiele odpowiedzialności.

Nazwa to tylko jeden aspekt obaw związanych z kodem. Gdzie kod się znajduje, jego moduł, jego klasa, także może być wskazaniem, jak zdecydować, czy go ponownie użyć.

3.5 Wymyśl to sam

Istnieje powszechne tureckie wyrażenie, które dosłownie tłumaczy się na "Nie wymyślaj teraz wynalazku". Oznacza to: "Nie sprawiaj nam kłopotów próbując teraz czegoś nowego, nie mamy na to czasu". Wynajdywanie koła na nowo jest problematyczne. Ta patologia ma nawet swoją nazwę w środowisku informatycznym: Zespół "Nie Wynaleźliśmy Tego Tu". Odnosi się ona do osób, które nie potrafią spać spokojnie, jeśli sami nie wynalazą już istniejącego produktu.

Oczywiście, tworzenie czegoś od podstaw, kiedy istnieje znane i działające alternatywne rozwiązanie, to dużo pracy. Jest to również podatne na błędy. Problem pojawia się, gdy ponowne używanie istniejących rozwiązań staje się normą, a tworzenie czegoś nowego staje się nieosiągalne. Zaognienie tego punktu widzenia ostatecznie zamienia się w motto "nigdy niczego nie wynajduj". Nie powinieneś bać się wynajdywania rzeczy.

Po pierwsze, wynalazca ma umysł pytający. Jeśli będziesz ciągle zadawać pytania, nieuchronnie stanie się z ciebie wynalazca. Kiedy wyraźnie uniemożliwiasz sobie zadawanie pytań, zaczynasz stawać się nudny i zamieniasz się w pracownika umysłowego. Powinieneś unikać tego podejścia, ponieważ osobie pozbawionej pytającego umysłu niemożliwe jest zoptymalizowanie swojej pracy.

Po drugie, nie wszystkie wynalazki mają alternatywy. Twoje własne abstrakcje to także wynalazki - twoje klasy, twoje projekty, pomocnicze funkcje, które wymyślasz. Wszystkie one są ułatwieniem pracy, ale wymagają wynalezienia.

Zawsze chciałem napisać stronę internetową, która dostarcza raporty ze statystyk Twittera dotyczące moich obserwujących oraz osób, które obserwuję. Problem polega na tym, że nie chcę uczyć się, jak działa Twitter API. Wiem, że istnieją biblioteki, które się tym zajmują, ale również nie chcę uczyć się, jak one działają, a co ważniejsze, nie chcę, aby ich implementacja wpływała na mój projekt. Jeśli użyję konkretnej biblioteki, zwiąże mnie to z API tej biblioteki, i jeśli zechcę ją zmienić, będę musiał przepisać kod wszędzie.

Sposób radzenia sobie z tymi problemami polega na wynalezieniu czegoś własnego. Tworzymy nasze wymarzone interfejsy i umieszczamy je jako abstrakcje przed biblioteką, którą używamy. W ten sposób unikamy związania się z konkretnym projektem API. Jeśli chcemy zmienić używaną bibliotekę, po prostu zmieniamy naszą abstrakcję, a nie cały kod w naszym projekcie. Obecnie nie mam pojęcia, jak działa Twitterowe API internetowe, ale wyobrażam sobie, że jest to zwykły żądanie sieciowe z jakimś elementem identyfikującym autoryzację dostępu do API Twittera. Oznacza to uzyskanie informacji od Twittera.

Pierwszy odruch programisty to znalezienie pakietu i sprawdzenie dokumentacji, jak go zintegrować z własnym kodem. Zamiast tego wynalazłem własne API i używam go, które ostatecznie wywołuje bibliotekę, którą używam pod spodem. Moje API powinno być jak najprostsze dla moich wymagań. Stań się swoim własnym klientem.

Po pierwsze, przejrzyj wymagania dotyczące API. API oparte na stronie internetowej zapewnia interfejs użytkownika w sieci, umożliwiający udzielenie uprawnień dla aplikacji. Otwiera stronę na Twitterze, która prosi o pozwolenie i przekierowuje z powrotem do aplikacji, jeśli użytkownik zatwierdzi. Oznacza to, że musimy wiedzieć, którego URL-a użyć do autoryzacji i którego URL-a przekierować z powrotem. Następnie możemy użyć danych ze strony przekierowanej do wykonania dodatkowych wywołań API później.

Nie powinniśmy potrzebować niczego więcej po autoryzacji. Wyobrażam sobie API do tego celu w następujący sposób.

```
public class Twitter {
  public static Uri GetAuthorizationUrl(Uri callbackUrl) {
    string redirectUrl = "";
    // ... do something here to build the redirect url
    return new Uri(redirectUrl);
  }

  public static TwitterAccessToken GetAccessToken(
    TwitterCallbackInfo callbackData) {
    // we should be getting something like this
    return new TwitterAccessToken();
  }

  public Twitter(TwitterAccessToken accessToken) {
    // we should store this somewhere
  }

  public IEnumerable<TwitterUserId> GetListOfFollowers(
    TwitterUserId userId) {
    // no idea how this will work
  }
}

public class TwitterUserId {
  // who knows how twitter defines user ids
}

public class TwitterAccessToken {
  // no idea what this will be
}

public class TwitterCallbackInfo {
  // this neither
}
```

Wynaleźliśmy coś od podstaw, nowe API Twittera, chociaż niewiele wiemy o tym, jak naprawdę działa Twitter API. Nasze API może nie być najlepsze dla ogólnego użytku, ale naszymi klientami jesteśmy sami, więc mamy luksus zaprojektowania go tak, aby odpowiadał naszym potrzebom. Na przykład nie sądzę, że będę musiał radzić sobie z tym, jak dane są przesyłane porcjami z oryginalnego API, i nie obchodzi mnie, czy muszę czekać i blokować działający kod, co może nie być pożądane w bardziej ogólnym API.

UWAGA

To podejście do posiadania własnych wygodnych interfejsów, które działają jako adapter, jest, jak nietrudno się domyślić, nazywane wzorcem adaptera na ulicach. Unikam podkreślania nazw nad rzeczywistą użyteczność, ale w przypadku, gdy ktoś zapyta, teraz już wiesz.

Później możemy wyodrębnić interfejs z klas, które zdefiniowaliśmy, aby nie musieć polegać na konkretnych implementacjach, co ułatwia testowanie. Nawet nie wiemy, czy używana przez nas biblioteka Twittera obsługuje łatwe zastępowanie swojej implementacji. Czasami możesz napotkać przypadki, gdy twój wymarzony projekt naprawdę nie pasuje do projektu rzeczywistego produktu. W takim przypadku musisz dostosować swój projekt, ale to dobry znak - oznacza to, że twój projekt również odzwierciedla twoje zrozumienie podstawowej technologii.

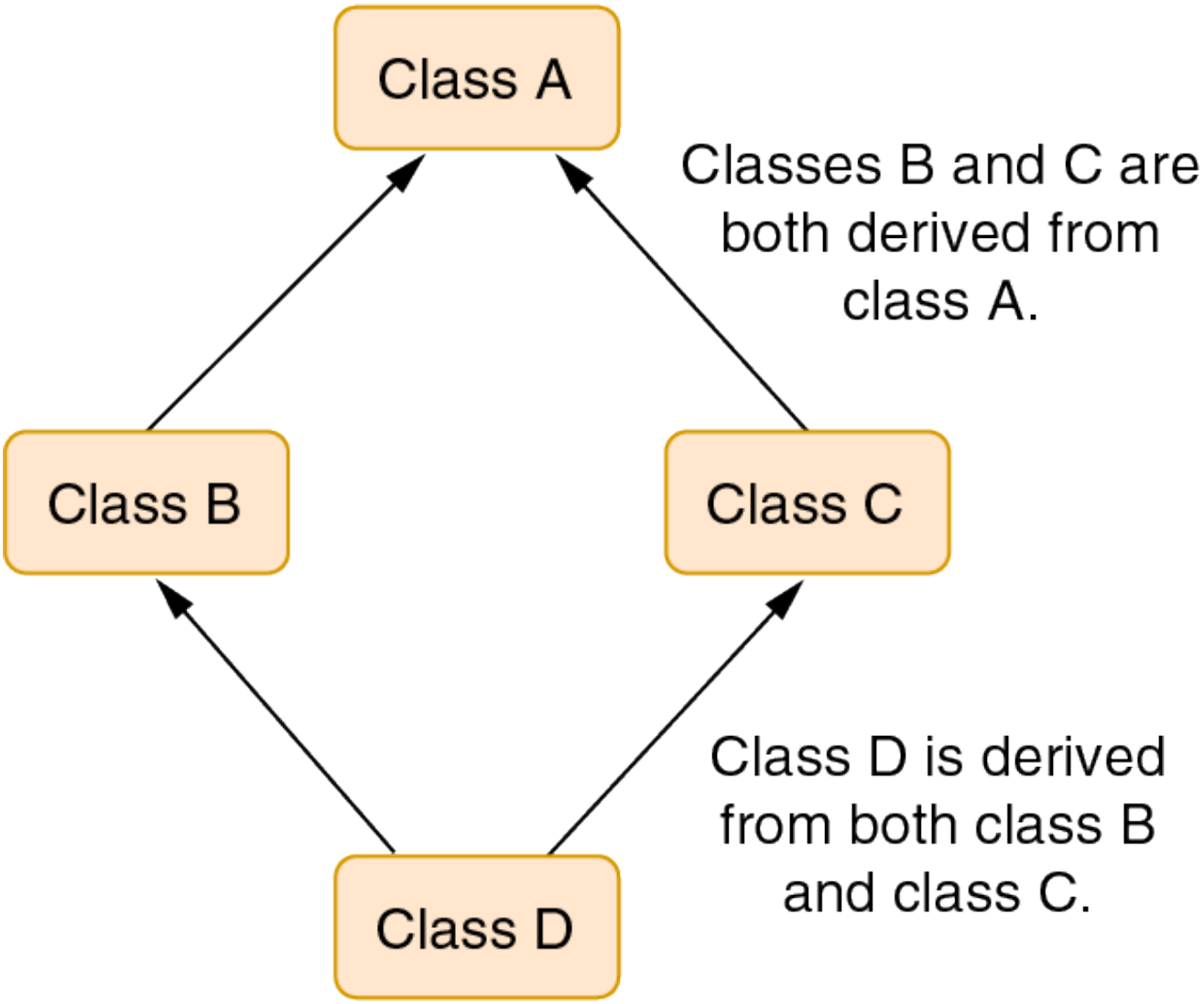
Więc może trochę skłamałem. Nie pisz biblioteki Twittera od zera. Ale nie odbiegaj od myślenia wynalazcy. Idą one w parze i powinieneś trzymać się obu.

3.6 Nie używaj dziedziczenia

Programowanie obiektowe (OOP) spadło na świat programowania jak kowadło w latach 90. XX wieku, powodując zmianę paradygmatu z programowania strukturalnego. Uważano je za rewolucyjne. Wiekowe problemy związane z ponownym użyciem kodu zostały wreszcie rozwiązane.

Najbardziej podkreślaną cechą OOP była dziedziczenie. Można było definiować ponowne użycie kodu jako zestaw dziedziczonych zależności. Dzięki temu możliwe było nie tylko prostsze ponowne użycie kodu, ale także prostsza modyfikacja kodu. Aby stworzyć nowy kod o nieco innych zachowaniach, nie musiałeś myśleć o zmianie oryginalnego kodu. Po prostu dziedziczyłeś go i przesłaniałeś odpowiednią część, aby uzyskać zmodyfikowane zachowanie.

Dziedziczenie przysporzyło jednak więcej problemów, niż rozwiązało w dłuższej perspektywie. Jednym z pierwszych problemów było wielokrotne dziedziczenie. Co by było, gdybyś musiał użyć kodu z wielu klas, a wszystkie miały metodę o tej samej nazwie, być może o tej samej sygnaturze? Jak to miałoby działać? A co z problemem diamentowej zależności, przedstawionym na rysunku 3.9? Byłoby to naprawdę skomplikowane, dlatego bardzo mało języków programowania zdecydowało się na jego implementację.



Oprócz wielokrotnego dziedziczenia, większym problemem z dziedziczeniem jest silna zależność, znana również jako ścisłe powiązanie. Jak już wspomniałem, zależności są korzeniem wszelkiego zła. Ze względu na swoją naturę dziedziczenie łączy cię z konkretną implementacją, co jest uważane za naruszenie jednej z uznawanych zasad programowania obiektowego, czyli zasady odwrócenia zależności. Ta zasada mówi, że kod nie powinien zależeć od konkretnej implementacji, lecz od abstrakcji.

Dlaczego istnieje taka zasada? Ponieważ gdy jesteś związany z konkretną implementacją, twój kod staje się sztywny i niezmienny. Jak już widzieliśmy, sztywny kod jest bardzo trudny do przetestowania lub zmodyfikowania.

Jak więc można ponownie użyć kodu? Jak można odziedziczyć swoją klasę po abstrakcji? To proste - nazywa się to kompozycją. Zamiast dziedziczyć po klasie, otrzymujesz jej abstrakcję jako parametr w swoim konstruktorze. Wyobraź sobie swoje komponenty jako klocki Lego, które wzajemnie się wspierają, zamiast jako hierarchię obiektów.

W przypadku zwykłego dziedziczenia, związek między wspólnym kodem a jego wariantami jest wyrażany modelem przodka/potomka. W przeciwieństwie do tego, kompozycja traktuje wspólną funkcję jako osobny komponent.

ZASADY SOLID

Istnieje słynny skrótowiec SOLID, który oznacza pięć zasad programowania obiektowego. Problem polega na tym, że SOLID wydaje się być stworzony po to, aby utworzyć sensowne słowo, a nie po to, aby uczynić nas lepszymi programistami. Nie uważam, że wszystkie jego zasady mają takie samo znaczenie, a niektóre mogą w ogóle nie mieć znaczenia. Zdecydowanie sprzeciwiam się przyjmowaniu zestawu zasad bez przekonania się o ich wartości.

Zasada pojedynczej odpowiedzialności, litera S w SOLID, mówi, że klasa powinna być odpowiedzialna za jedną rzecz, a nie za wiele, jak w przypadku tzw. klas boskich. Jest to dość niejasne, ponieważ to my definiujemy, co oznacza jedna rzecz. Czy możemy powiedzieć, że klasa z dwoma metodami nadal jest odpowiedzialna za jedną rzecz? Nawet klasa boska jest odpowiedzialna za jedną rzecz na pewnym poziomie: bycie klasą boską. Zamiast tego proponuję zasadę klarownej nazwy: nazwa klasy powinna jak najmniej niejasno wyjaśniać jej funkcję. Jeśli nazwa jest zbyt długa lub niejasna, klasę należy podzielić na kilka mniejszych klas.

Zasada otwarte/zamknięte, litera O w SOLID, mówi, że klasa powinna być otwarta na rozszerzenie, ale zamknięta na modyfikację. Oznacza to, że powinniśmy projektować klasy tak, aby ich zachowanie można było modyfikować z zewnątrz. Jest to ponownie bardzo niejasne i może być czasochłonne. Rozszerzalność to decyzja projektowa, która nie zawsze jest pożądana, praktyczna lub nawet bezpieczna. Wydaje się to być porównywalne do rady „użyj opon wyścigowych" w programowaniu. Zamiast tego powiedziałbym „Potraktuj rozszerzalność jako funkcję".

Zasada podstawienia Liskov, wymyślona przez Barbarę Liskov, mówi, że zachowanie programu nie powinno się zmieniać, jeśli jedną z używanych klas zostanie zastąpiona klasą pochodną. Chociaż rada jest słuszna, nie sądzę, że ma znaczenie w codziennej pracy programisty. Wydaje się to być porównywalne do rady „Nie popełniaj błędów". Jeśli łamiesz kontrakt interfejsu, program będzie miał błędy. Jeśli projektujesz zły interfejs, program również będzie miał błędy. To naturalny porządek rzeczy. Może to być ujęte w prostsze i bardziej praktyczne rady, takie jak „Przestrzegaj kontraktu".

Zasada segregacji interfejsów faworyzuje mniejsze i konkretne interfejsy nad ogólnymi i szeroko zakrojonymi interfejsami. Jest to zbyt skomplikowana i niejasna rada, a czasem po prostu błędna. Mogą wystąpić sytuacje, w których szeroko zakrojone interfejsy są bardziej odpowiednie dla zadania, a nadmiernie rozbudowane interfejsy mogą generować zbyt duże obciążenie. Podział interfejsów powinien wynikać z rzeczywistych wymagań projektowych, a nie arbitralnych kryteriów dotyczących ich zakresu. Jeśli pojedynczy interfejs nie spełnia wymagań, można go podzielić, ale nie dla spełnienia określonych kryteriów granulacji.

Zasada odwrócenia zależności jest ostatnią zasadą. Ponownie, nie jest to zbyt dobra nazwa. Po prostu nazywajmy to zależnością od abstrakcji. Tak, uzależnienie od konkretnych implementacji prowadzi do silnego powiązania, a widzieliśmy już jego niepożądane skutki. Ale to nie oznacza, że powinieneś zaczynać tworzyć interfejsy dla każdej zależności. Wręcz przeciwnie: warto uzależniać się od abstrakcji, gdy zależy nam na elastyczności i widzimy w tym wartość, a od konkretnej implementacji w przypadkach, gdzie to nie ma znaczenia. Twój kod powinien dostosowywać się do twojego projektu, a nie na odwrót. Śmiało eksperymentuj z różnymi modelami.

Kompozycja jest bardziej podobna do relacji klient-serwer niż do relacji rodzic-dziecko. Wywołujesz ponownie używany kod za jego referencją, zamiast dziedziczyć jego metody w swoim zakresie. Możesz konstruować klasę, od której zależysz, w swoim konstruktorze, a jeszcze lepiej, możesz ją otrzymać jako parametr, co pozwoliłoby ci używać jej jako zewnętrznego źródła. Pozwala to na bardziej konfigurowalne i elastyczne zarządzanie tą relacją.

Otrzymywanie jej jako parametru ma dodatkową zaletę ułatwiającą testowanie jednostkowe obiektu przez wstrzykiwanie atrap (ang. mock) konkretnych implementacji. Omówię wstrzykiwanie zależności bardziej szczegółowo w rozdziale 5.

Użycie kompozycji zamiast dziedziczenia może wymagać napisania znacznie więcej kodu, ponieważ możesz potrzebować definiować zależności za pomocą interfejsów zamiast konkretnych odwołań, ale jednocześnie pozwoli to na uwolnienie kodu od zależności. Nadal musisz rozważyć za i przeciw kompozycji, zanim zdecydujesz się jej użyć.

3.7 Nie używaj klas

Żeby było jasne – klasy są świetne. Wykonują swoją pracę, a potem ustępują miejsca. Jak omawiałem w rozdziale 2, niosą ze sobą niewielkie obciążenie odwołań do referencji i zajmują nieco więcej przestrzeni w stosunku do typów wartości. Te kwestie nie będą miały znaczenia większość czasu, ale ważne jest, abyś znał ich zalety i wady, aby zrozumieć kod oraz jak możesz wpływać na niego podejmując złe decyzje.

Typy wartości mogą być, cóż, wartościowe. Podstawowe typy wartości dostępne w języku C#, takie jak int, long i double, już są typami wartości. Możesz również tworzyć własne typy wartości za pomocą konstrukcji takich jak enum i struct.

3.7.1 Enumeracje są cacy!

Enumeracje są świetne do przechowywania dyskretnych wartości porządkowych. Klasy również mogą być używane do definiowania dyskretnych wartości, ale brakuje im pewnych udogodnień, których enums posiadają. Oczywiście, klasa zawsze jest lepsza od zakodowania wartości na stałe.

Jeśli piszesz kod obsługujący odpowiedź na żądanie sieciowe, które wysyłasz w swojej aplikacji, możesz być zmuszony do radzenia sobie z różnymi numerycznymi kodami odpowiedzi. Załóżmy, że odpytujesz serwis pogodowy Narodowej Służby Meteorologicznej o informacje pogodowe dla podanej lokalizacji użytkownika i piszesz funkcję do pobierania potrzebnych danych. W kodzie przedstawionym w listingu 3.6 używamy RestSharp do żądań API oraz Newtonsoft.JSON do analizy odpowiedzi, sprawdzając, czy kod statusu HTTP jest udany, korzystając z hardcoded wartości (200) w warunku if. Następnie używamy biblioteki Json.NET do analizy odpowiedzi na dynamiczny obiekt, aby wydobyć potrzebne informacje.

```
static double? getTemperature(double latitude,
  double longitude) {
  const string apiUrl = "https://api.weather.gov";
  string coordinates = $"{latitude},{longitude}";
  string requestPath = $"/points/{coordinates}/forecast/hourly";
  var client = new RestClient(apiUrl);
  var request = new RestRequest(requestPath);
  var response = client.Get(request);
  if (response.StatusCode == 200) {
    dynamic obj = JObject.Parse(response.Content);
    var period = obj.properties.periods[0];
    return (double)period.temperature;
```

```
    }
    return null;
}
```

Największym problemem z zakodowanymi na stałe wartościami jest niezdolność ludzi do zapamiętania liczb. Nie jesteśmy w tym dobry. Nie jesteśmy w stanie zrozumieć ich od razu, z wyjątkiem liczby zer na naszych wypłatach. Są trudniejsze do wpisania niż proste nazwy, ponieważ trudno jest skojarzyć liczby z mnemonikami, a jednak łatwo popełnić literówkę. Drugim problemem z zakodowanymi wartościami jest to, że wartości mogą się zmieniać. Jeśli używasz tej samej wartości wszędzie indziej, oznacza to, że trzeba zmienić wszystko inne tylko po to, aby zmienić jedną wartość.

Drugi problem z liczbami polega na tym, że brakuje im intencji. Wartość numeryczna, tak jak 200, może oznaczać cokolwiek. Nie wiemy, co to jest. Dlatego unikaj zakodowywania wartości na stałe.

Klasy są jednym ze sposobów na hermetyzowanie wartości. Możesz hermetyzować kody statusu HTTP w klasie, na przykład tak:

```
class HttpStatusCode {
    public const int OK = 200;
    public const int NotFound = 404;
    public const int ServerError = 500;
    // ... i tak dalej
}
```

W ten sposób możesz zmienić linię, która sprawdza, czy żądanie HTTP zostało pomyślnie wykonane, używając kodu podobnego do tego:

```
if (response.StatusCode == HttpStatusCode.OK) {
    // ...
}
```

Ta wersja jest znacznie bardziej opisowa. Od razu rozumiemy kontekst, co wartość oznacza i jakie ma znaczenie w danym kontekście. Jest to absolutnie opisowe.

A więc po co są enumy? Czy nie możemy użyć klas do tego? Weźmy pod uwagę, że mamy inną klasę do przechowywania wartości:

```
class ImageWidths {
    public const int Small = 50;
    public const int Medium = 100;
    public const int Large = 200;
}
```

Teraz ten kod skompiluje się i co ważniejsze, zwróci prawdę:

```
return HttpStatusCode.OK == ImageWidths.Large;
```

To coś, czego prawdopodobnie nie chcesz. Załóżmy, że napisalibyśmy to za pomocą enuma:

```
enum HttpStatusCode {
    OK = 200,
    NotFound = 404,
    ServerError = 500,
}
```

To o wiele łatwiejsze do napisania, prawda? Jego użycie byłoby takie samo w naszym przykładzie. Co ważniejsze, każdy zdefiniowany typ enum jest odrębny, co sprawia, że wartości są bezpieczne pod względem typów, w przeciwieństwie do naszego przykładu z klasami i constami. Enumy są błogosławieństwem w naszym przypadku. Gdybyśmy próbowali porównać dwie różne typy enum, kompilator wygenerowałby błąd:

```
error CS0019: Operator '==' cannot be applied to operands of type
'HttpStatusCode' and 'ImageWidths'
```

Świetnie! Enumy oszczędzają nam czasu, nie pozwalając nam porównywać jabłek do pomarańczy podczas kompilacji. Przekazują też intencję, podobnie jak klasy zawierające wartości. Enumy są również typami wartości, co oznacza, że są tak szybkie jak przekazywanie wartości liczbowej.

3.7.2 Struktury są świetne!

Jak wskazano w rozdziale 2, klasy posiadają niewielki narzut na pamięć. Każda klasa musi przechowywać nagłówek obiektu oraz tablicę metod wirtualnych, gdy jest instancjonowana. Dodatkowo klasy są alokowane na stercie (heap) i są ewidencjonowane przez mechanizm zbierania śmieci.

Oznacza to, że .NET musi śledzić każdą zinstancjonowaną klasę i usuwać je z pamięci, gdy nie są już potrzebne. To bardzo wydajny proces - większość czasu nawet nie zauważasz, że tam jest. Jest to magia. Nie wymaga ręcznego zarządzania pamięcią. Więc nie, nie musisz się obawiać używania klas.

Ale jak widzieliśmy, warto wiedzieć, kiedy możesz skorzystać z darmowej korzyści, gdy jest dostępna. Struktury są podobne do klas. Możesz w nich definiować właściwości, pola i metody. Struktury mogą również implementować interfejsy. Jednak struktura nie może być dziedziczona i także nie może dziedziczyć po innej strukturze ani klasie. To dlatego, że struktury nie mają tablicy metod wirtualnych ani nagłówka obiektu. Nie są zbierane przez mechanizm zbierania śmieci, ponieważ są alokowane na stosie wywołań (call stack).

Jak już wspomniałem w rozdziale 2, stos wywołań to po prostu ciągły blok pamięci, którego jedyną zmienną jest wskaźnik do jego górnej części, który porusza się w górę i w dół w miarę wywoływania i kończenia funkcji. To sprawia, że stos jest bardzo efektywnym mechanizmem przechowywania, ponieważ oczyszczanie jest szybkie i automatyczne. Nie ma możliwości fragmentacji, ponieważ zawsze jest to LIFO (Last In, First Out).

Skoro stos jest tak szybki, dlaczego nie używamy go do wszystkiego? Dlaczego istnieje sterta (heap) i mechanizm zbierania śmieci? To dlatego, że stos może istnieć tylko przez czas życia funkcji. Gdy twoja funkcja zwróci wartość, wszystko na stosie ramki funkcji jest usuwane, więc inne funkcje mogą używać tego samego miejsca na stosie. Potrzebujemy sterty dla obiektów, które przetrwają funkcje.

Warto również wiedzieć, że stos ma ograniczony rozmiar. To właśnie z tego powodu istnieje cała strona internetowa o nazwie "Stack Overflow": dlatego, że twoja aplikacja ulegnie awarii, jeśli przekroczysz limit stosu. Respektuj stos - poznaj jego ograniczenia.

Mimo że struktury same w sobie są typami wartości, mogą nadal zawierać typy referencyjne. Jeśli na przykład struktura zawiera ciąg znaków, nadal jest to typ referencyjny wewnątrz typu wartości, podobnie jak możesz mieć typy wartości wewnątrz typu referencyjnego. Pokażę to na ilustracjach w tej sekcji.

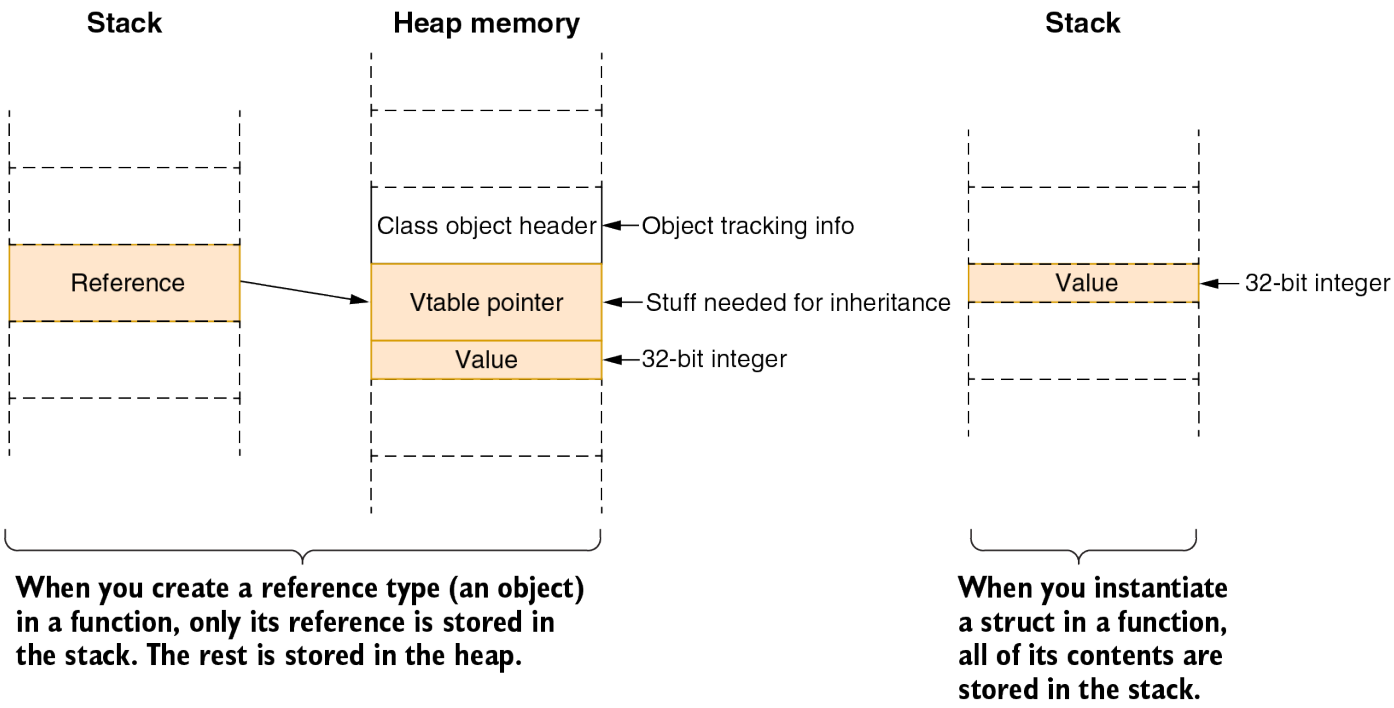
Jeśli masz strukturę, która zawiera tylko wartość całkowitą, to zajmuje ona ogólnie mniej miejsca w pamięci niż referencja do klasy zawierającej wartość całkowitą, jak pokazano na rysunku 3.10. Nasze struktury i warianty klas służą do przechowywania identyfikatorów, o których rozmawiałem w rozdziale 2. Dwa rodzaje tej samej struktury wyglądałyby tak, jak pokazano w poniższym przykładzie.

```
public class Id {
 public int Value { get; private set; }

 public Id (int value) {
   this.Value = value;
 }
}

public struct Id {
 public int Value { get; private set; }

 public Id (int value) {
   this.Value = value;
 }
}
```

Jedyna różnica w kodzie polega na słowach kluczowych `struct` i `class`, ale zauważ, jak różnią się one w sposobie przechowywania, gdy są tworzone w funkcji, np. takiej jak ta:

```
var a = new Id(123);
```

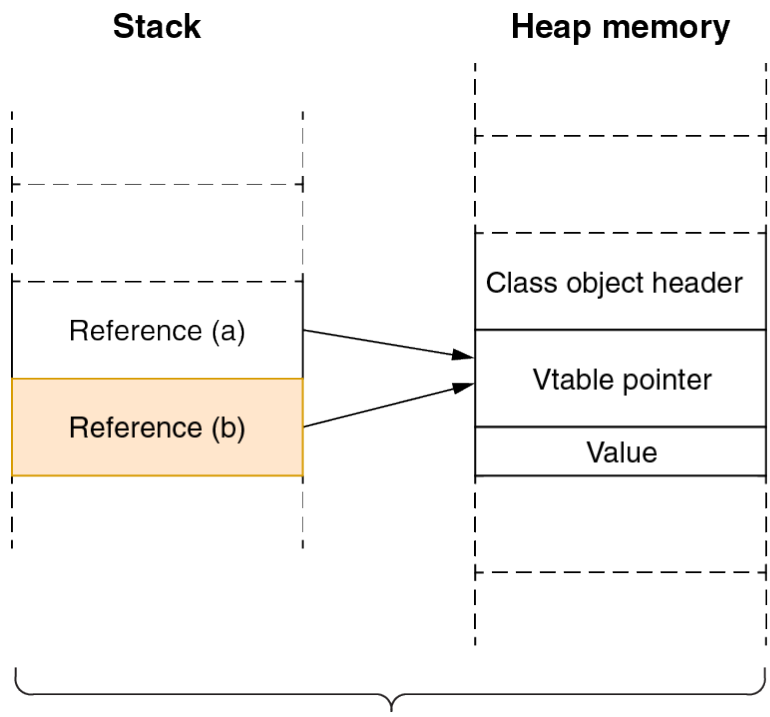
Rysunek 3.10 pokazuje, jak są one rozmieszczone.



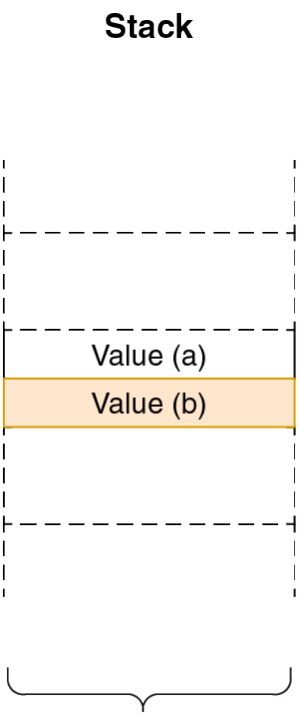
Ponieważ struktury są typami wartości, przypisanie jednej struktury do drugiej tworzy również kolejną kopię całej zawartości struktury, zamiast tworzyć tylko kolejną kopię referencji:

```
var a = new Id(123);
var b = a;
```

W tym przypadku rysunek 3.11 pokazuje, jak struktury mogą być efektywne w przechowywaniu małych typów.



When you assign a reference to another variable, only the reference is copied, not the contents of the class.



When you assign a created struct to another variable, all of its contents are copied.

Chociaż przechowywanie danych na stosie jest tymczasowe podczas wykonywania funkcji, jest ono znacznie mniejsze w porównaniu do sterty. Stos ma rozmiar 1 megabajta w .NET, podczas gdy sterta może pomieścić terabajty danych. Stos jest szybki, ale jeśli wypełnisz go dużymi strukturami, może łatwo się zapełnić. Ponadto kopiowanie dużych struktur jest również wolniejsze niż kopiowanie tylko referencji. Załóżmy, że chcemy przechowywać pewne informacje o użytkowniku razem z naszymi identyfikatorami. Nasza implementacja wyglądałaby jak w poniższym przykładzie.

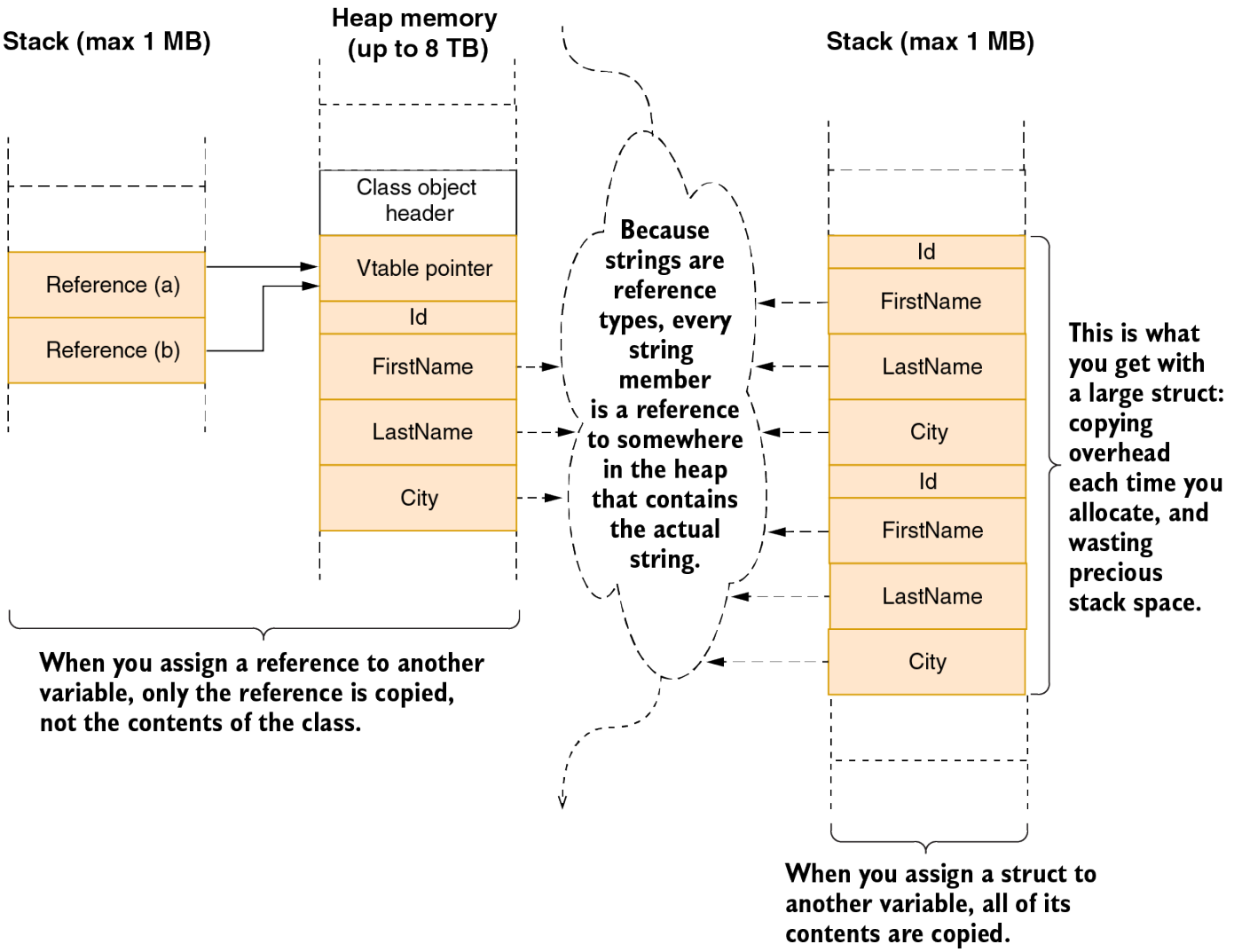
Listing 3.8 Definicja większej klasy lub struktury

```
public class Person {
  public int Id { get; private set; }
  public string FirstName { get; private set; }
  public string LastName { get; private set; }
  public string City { get; private set; }

  public Person(int id, string firstName, string lastName,
    string city) {
    Id = id;
    FirstName = firstName;
    LastName = lastName;
    City = city;
  }
}
```

Jedyna różnica między tymi dwoma definicjami to słowa kluczowe struct i class. Niemniej jednak tworzenie i przypisywanie jednego do drugiego ma głęboki wpływ na to, jak działają rzeczy za kulisami. Rozważ ten prosty kod, gdzie Person może być zarówno strukturą, jak i klasą:

```
var a = new Person(42, "Sedat", "Kapanoglu", "San Francisco");
var b = a;
```

Po przypisaniu a do b, różnice w układach pamięci wynikowych są pokazane na rysunku 3.12.



Stos wywołań może być niezwykle szybki i efektywny w przechowywaniu danych. Są świetne do pracy z małymi wartościami o mniejszym nakładzie pracy, ponieważ nie podlegają one gromadzeniu odpadów (garbage collection). Ponieważ nie są typami referencyjnymi, nie mogą mieć wartości null, co oznacza, że nie jest możliwe wystąpienie wyjątku null reference exception w przypadku struktur.

Nie można używać struktur do wszystkiego, co jest oczywiste ze względu na sposób ich przechowywania: nie można udostępniać im wspólnego odniesienia, co oznacza, że nie można zmieniać wspólnej instancji z różnych odniesień. To jest coś, co często robimy nieświadomie i o czym nigdy nie myślimy. Rozważmy, że chcemy, aby struktura była mutowalna i używamy modyfikatorów get; set; zamiast get; private set;. Oznacza to, że możemy modyfikować strukturę na bieżąco. Spójrzmy na poniższy przykład.

```
public struct Person {
```

```
  public int Id { get; set; }
  public string FirstName { get; set; }
  public string LastName { get; set; }
  public string City { get; set; }

  public Person(int id, string firstName, string lastName,
    string city) {
    Id = id;
    FirstName = firstName;
    LastName = lastName;
    City = city;
  }
}
```

Przyjrzyjmy się temu kawałkowi kodu z mutowalną strukturą:

```
var a = new Person(42, "Sedat", "Kapanoglu", "San Francisco");
var b = a;
b.City = "Eskisehir";
Console.WriteLine(a.City);
Console.WriteLine(b.City);
```

Jak myślisz, jaki będzie wynik? Gdyby to była klasa, obie linie pokazałyby "Eskisehir" jako nowe miasto. Ale ponieważ mamy dwie osobne kopie, wydrukuje "San Francisco" i "Eskisehir". Z tego powodu zawsze warto, aby struktury były prawie niezmienne, dzięki czemu nie można ich przypadkowo zmieniać później i powodować błędów.

Chociaż powinieneś preferować kompozycję nad dziedziczeniem dla ponownego użycia kodu, dziedziczenie może być również przydatne, gdy dana zależność jest zawarta. Klasy mogą zapewnić większą elastyczność niż struktury w tych przypadkach.

Klasy mogą zapewnić bardziej efektywne przechowywanie, gdy są większe rozmiarowo, ponieważ w przypadku przypisania będą kopiowane tylko ich odniesienia. Biorąc pod uwagę wszystko to, śmiało używaj struktur do małych, niezmienialnych typów wartości, które nie wymagają dziedziczenia.

3.8. Napisz zły kod

Dobre praktyki wynikają z złego kodu, ale zarazem zły kod może również powstawać w wyniku ślepego stosowania dobrych praktyk. Programowanie strukturalne, obiektowe, a nawet funkcyjne są opracowane po to, aby programiści pisali lepszy kod. Podczas gdy uczymy się dobrych praktyk, niektóre złe praktyki są też wytykane jako "zło" i całkowicie wykluczone. Przejrzyjmy niektóre z nich.

3.8.1 Nie używaj instrukcji If/Else

Instrukcja If/Else jest jednym z pierwszych elementów, które uczysz się w programowaniu. Jest to wyraz jednej z podstawowych części komputerów: logiki. Kochamy If/Else. Pozwala nam wyrażać logikę naszego programu w sposób przypominający schemat przepływu. Jednak tego rodzaju wyrażenie może także sprawić, że kod staje się mniej czytelny.

Podobnie jak wiele konstrukcji programistycznych, bloki If/Else powodują, że kod w warunkach jest wcięty. Załóżmy, że chcemy dodać pewną funkcjonalność do naszej klasy Person z ostatniej sekcji, aby przetworzyć rekord w bazie danych. Chcemy sprawdzić, czy właściwość City klasy Person została zmieniona i zmienić ją również w bazie danych, jeśli klasa Person wskazuje na ważny rekord. Jest to dość przesunięta implementacja. Są lepsze sposoby na realizację tych rzeczy, ale chcę ci pokazać, jak kod może się skończyć, a nie jego rzeczywistą funkcjonalność. Przedstawiam ci kształt w poniższym kodzie.

```
public UpdateResult UpdateCityIfChanged() {
 if (Id > 0) {
   bool isActive = db.IsPersonActive(Id);
   if (isActive) {
     if (FirstName != null && LastName != null) {
       string normalizedFirstName = FirstName.ToUpper();
       string normalizedLastName = LastName.ToUpper();
       string currentCity = db.GetCurrentCityByName(
         normalizedFirstName, normalizedLastName);
       if (currentCity != City) {
         bool success = db.UpdateCurrentCity(Id, City);
         if (success) {
           return UpdateResult.Success;
         } else {
           return UpdateResult.UpdateFailed;
         }
       } else {
         return UpdateResult.CityDidNotChange;
       }
     } else {
       return UpdateResult.InvalidName;
     }
   } else {
     return UpdateResult.PersonInactive;
   }
 } else {
   return UpdateResult.InvalidId;
 }
}
```

Nawet jeśli wytłumaczyłem, co funkcja robi krok po kroku, po pięciu minutach wrócenie do tej funkcji może ponownie wywołać zamieszanie. Jednym z powodów tego zamieszania jest zbyt dużo wcięć. Ludzie nie są przyzwyczajeni do czytania rzeczy w formacie wciętym, z małym wyjątkiem użytkowników Reddit. Trudno określić, do jakiego bloku należy dany wiersz, jaki jest kontekst. Ciężko jest zrozumieć logikę.

Ogólna zasada unikania niepotrzebnych wcięć polega na jak najszybszym opuszczaniu funkcji oraz unikaniu stosowania else, gdy przepływ już implikuje else. Na listingu 3.11 pokazane jest, jak instrukcje return już same wskazują koniec przepływu kodu, eliminując potrzebę stosowania else.

```
public UpdateResult UpdateCityIfChanged() {
  if (Id <= 0) {
    return UpdateResult.InvalidId;
  }
```

```
  bool isActive = db.IsPersonActive(Id);
  if (!isActive) {
    return UpdateResult.PersonInactive;
  }
  if (FirstName is null || LastName is null) {
    return UpdateResult.InvalidName;
  }
  string normalizedFirstName = FirstName.ToUpper();
  string normalizedLastName = LastName.ToUpper();
  string currentCity = db.GetCurrentCityByName(
    normalizedFirstName, normalizedLastName);
  if (currentCity == City) {
    return UpdateResult.CityDidNotChange;
  }
  bool success = db.UpdateCurrentCity(Id, City);
  if (!success) {
    return UpdateResult.UpdateFailed;
  }
  return UpdateResult.Success;
}
```

Zastosowana tutaj technika nazywana jest podążaniem ścieżką optymistyczną (happy path). Ścieżka optymistyczna w kodzie to fragment kodu, który jest wykonywany, jeśli nic innego nie pójdzie źle. To, co idealnie powinno zdarzyć się podczas wykonania. Ponieważ ścieżka optymistyczna podsumowuje główną pracę funkcji, musi być najłatwiejsza do odczytania. Przekształcając kod w instrukcje return zamiast używać else, pozwalamy czytelnikowi łatwiej zidentyfikować ścieżkę optymistyczną, niż gdybyśmy mieli matryoszkowe układy instrukcji warunkowych.

Waliduj wcześnie, i zwracaj jak najwcześniej. Umieszczaj przypadki wyjątkowe wewnątrz instrukcji warunkowych, a staramy się umieścić ścieżkę optymistyczną poza blokami. Zapoznaj się z tymi dwoma kształtami, aby uczynić swój kod bardziej czytelnym i łatwiejszym w utrzymaniu.

3.8.2 Użyj goto

Cała teoria programowania można streścić jako pamięć, podstawowa arytmetyka oraz instrukcje warunkowe if i goto. Instrukcja goto przekierowuje wykonanie programu bezpośrednio do arbitralnego punktu docelowego. Są trudne do śledzenia, i ich stosowanie było zniechęcane od czasu, gdy Edsger Dijkstra napisał artykuł zatytułowany „Go to statement is considered harmful” (https://dl.acm.org/doi/10.1145/362929.362947). Istnieje wiele nieporozumień dotyczących tego artykułu Dijkstry, przede wszystkim co do jego tytułu. Dijkstra zatytułował swoją pracę „A case against the GO TO statement”, ale jego redaktor, również wynalazca języka Pascal, Niklaus Wirth, zmienił tytuł, co sprawiło, że stanowisko Dijkstry stało się bardziej agresywne, a walka przeciwko goto zamieniła się w polowanie na wiedźmy.

Wszystko to wydarzyło się przed latami 80. Języki programowania miały dużo czasu, aby stworzyć nowe konstrukcje rozwiązujące funkcje instrukcji goto. Pętle for/while, instrukcje return/break/continue, a nawet wyjątki zostały stworzone, aby obsłużyć konkretne scenariusze, które wcześniej były możliwe tylko za pomocą goto. Byli programiści BASIC zapamiętają słynną instrukcję obsługi błędów ON ERROR GOTO, która była prymitywnym mechanizmem obsługi wyjątków.

Chociaż wiele nowoczesnych języków nie ma już odpowiedników instrukcji goto, C# ma, i działa świetnie w jednym konkretnym scenariuszu: eliminuje nadmiarowe punkty wyjścia z funkcji. Możliwe jest użycie instrukcji goto w sposób łatwy do zrozumienia i sprawia, że twój kod jest mniej podatny na błędy, oszczędzając jednocześnie czas. To jak trzykrotne uderzenie w grze Mortal Kombat.

Punkt wyjścia to każda instrukcja w funkcji, która powoduje jej powrót do wywołującego ją kodu. W języku C# każda instrukcja return stanowi punkt wyjścia. Eliminowanie punktów wyjścia w starszych językach programowania było ważniejsze niż obecnie, ponieważ ręczne sprzątanie było bardziej powszechną częścią codziennego życia programisty. Musiałeś pamiętać, co alokowałeś i co musisz posprzątać przed powrotem.

C# dostarcza doskonałe narzędzia do strukturalnego sprzątania, takie jak bloki try/finally i instrukcje using. Mogą wystąpić przypadki, w których żadne z nich nie jest odpowiednie dla twojego scenariusza, i możesz użyć instrukcji goto również do sprzątania, ale naprawdę świetnie się sprawdza w eliminowaniu nadmiarowości. Załóżmy, że rozwijamy formularz wprowadzania adresu dostawy dla strony internetowej z zakupami online. Formularze internetowe są świetne do demonstracji wielopoziomowej walidacji, która w nich zachodzi. Załóżmy, że chcielibyśmy używać ASP.NET Core do tego celu. Oznacza to, że potrzebujemy akcji przesyłania dla naszego formularza. Kod dla niej może wyglądać tak, jak w poniższym przykładzie. Mamy walidację modelu, która odbywa się po stronie klienta, ale jednocześnie potrzebujemy pewnej walidacji po stronie serwera z naszym formularzem, abyśmy mogli sprawdzić, czy adres jest naprawdę poprawny, korzystając z API USPS. Po sprawdzeniu możemy próbować zapisać informacje do bazy danych, a jeśli to się powiedzie, przekierowujemy użytkownika do strony z informacjami o płatności. W przeciwnym razie musimy ponownie wyświetlić formularz adresu dostawy.

```
[HttpPost]
public IActionResult Submit(ShipmentAddress form) {
  if (!ModelState.IsValid) {
    return RedirectToAction("Index", "ShippingForm", form);
  }
  var validationResult = service.ValidateShippingForm(form);
  if (validationResult != ShippingFormValidationResult.Valid) {
    return RedirectToAction("Index", "ShippingForm", form);
  }
  bool success = service.SaveShippingInfo(form);
  if (!success) {
    ModelState.AddModelError("", "Problem occurred while " +
      "saving your information, please try again");
    return RedirectToAction("Index", "ShipingForm", form);
  }
  return RedirectToAction("Index", "BillingForm");
}
```

Już omówiłem kilka problemów z kopiowaniem i wklejaniem, ale wiele punktów wyjścia w przykładzie 3.12 stanowi kolejny problem. Zauważyłeś literówkę w trzecim poleceniu return? Przez pomyłkę usunęliśmy znak bez zauważenia, a ponieważ znajduje się on w ciągu znaków, ten błąd jest niemożliwy do wykrycia, chyba że napotkamy problem podczas zapisywania formularza w produkcji lub zbudujemy rozbudowane testy dla naszych kontrolerów. Duplikacja może powodować problemy w tych przypadkach. Instrukcja goto może pomóc w scaleniu poleceń return pod jedną etykietą goto, jak pokazano w przykładzie 3.13. Tworzymy nową etykietę dla przypadku błędu w naszej ścieżce głównej i wielokrotnie jej używamy w naszej funkcji,

korzystając z instrukcji goto.

```
[HttpPost]
public IActionResult Submit2(ShipmentAddress form) {
  if (!ModelState.IsValid) {
    goto Error;
  }
  var validationResult = service.ValidateShippingForm(form);
  if (validationResult != ShippingFormValidationResult.Valid) {
    goto Error;
  }
  bool success = service.SaveShippingInfo(form);
  if (!success) {
    ModelState.AddModelError("", "Problem occurred while " +
      "saving your shipment information, please try again");
    goto Error;
  }
  return RedirectToAction("Index", "BillingForm");
Error:
  return RedirectToAction("Index", "ShippingForm", form);
}
```

To, co jest świetne w tym rodzaju konsolidacji, polega na tym, że jeśli kiedykolwiek chcesz dodać więcej do wspólnego kodu wyjścia, musisz to zrobić tylko w jednym miejscu. Powiedzmy, że chcesz zapisać plik cookie na kliencie, gdy wystąpi błąd. Wystarczy, że dodasz go po etykiecie "Error", jak pokazano poniżej.

Listing 3.14 Łatwość dodawania dodatkowego kodu do wspólnego kodu wyjścia

```
[HttpPost]
public IActionResult Submit3(ShipmentAddress form) {
  if (!ModelState.IsValid) {
    goto Error;
  }
  var validationResult = service.ValidateShippingForm(form);
  if (validationResult != ShippingFormValidationResult.Valid) {
    goto Error;
  }
  bool success = service.SaveShippingInfo(form);
  if (!success) {
    ModelState.AddModelError("", "Problem occurred while " +
      "saving your information, please try again");
    goto Error;
  }
  return RedirectToAction("Index", "BillingForm");
Error:
  Response.Cookies.Append("shipping_error", "1");
  return RedirectToAction("Index", "ShippingForm", form);
}
```

Korzystając z instrukcji goto, zachowaliśmy czytelniejszy styl kodu, z mniejszą ilością wcięć, zaoszczędziliśmy czas i ułatwiliśmy wprowadzanie zmian w przyszłości, ponieważ musimy to zrobić tylko raz.

Instrukcja goto nadal może wprowadzić w zakłopotanie kolegę, który nie jest przyzwyczajony do tego składni. Na szczęście w C# 7.0 wprowadzono lokalne funkcje, które mogą być używane do wykonania tego samego zadania, być może w sposób łatwiejszy do zrozumienia. Deklarujemy lokalną funkcję o nazwie "error", która wykonuje wspólne operacje zwracania błędów i zwraca jej wynik, zamiast korzystać z instrukcji goto. Możesz zobaczyć to w akcji w poniższym przykładzie.

```
[HttpPost]
public IActionResult Submit4(ShipmentAddress form) {
  IActionResult error() {
    Response.Cookies.Append("shipping_error", "1");
    return RedirectToAction("Index", "ShippingForm", form);
  }
  if (!ModelState.IsValid) {
    return error();
  }
  var validationResult = service.ValidateShippingForm(form);
  if (validationResult != ShippingFormValidationResult.Valid) {
    return error();
  }
  bool success = service.SaveShippingInfo(form);
  if (!success) {
    ModelState.AddModelError("", "Problem occurred while " +
      "saving your information, please try again");
    return error();
  }
  return RedirectToAction("Index", "BillingForm");
}
```

Korzystanie z lokalnych funkcji pozwala nam również na deklarację obsługi błędów na początku funkcji, co jest normą w nowoczesnych językach programowania, takich jak Go, za pomocą instrukcji takich jak defer, chociaż w naszym przypadku musimy jawnie wywołać funkcję error(), aby ją wykonać.

3.9 Nie pisz komentarzy w kodzie

Turecki architekt o imieniu Sinan żył w XVI wieku. Zbudował słynną meczet Sulejmaniye w Stambule oraz wiele innych budowli. Istnieje opowieść o jego umiejętnościach w architekturze. Według tej historii, wiele lat po śmierci Sinana, grupa architektów rozpoczęła prace restauracyjne na jednym z jego budynków. W jednym z łuków był kluczowy kamień, który musieli wymienić. Ostrożnie usunęli kamień i znaleźli małą fiolkę z zamieszczoną wewnątrz notatką. Notatka brzmiała: „Ten kamień kluczowy przetrwał tylko trzysta lat. Jeśli czytasz tę notatkę, musi się on już zepsuć lub próbujesz go naprawić. Istnieje tylko jeden właściwy sposób, aby właściwie umieścić nowy kamień kluczowy.” Notatka zawierała techniczne szczegóły dotyczące właściwej wymiany kamienia kluczowego.

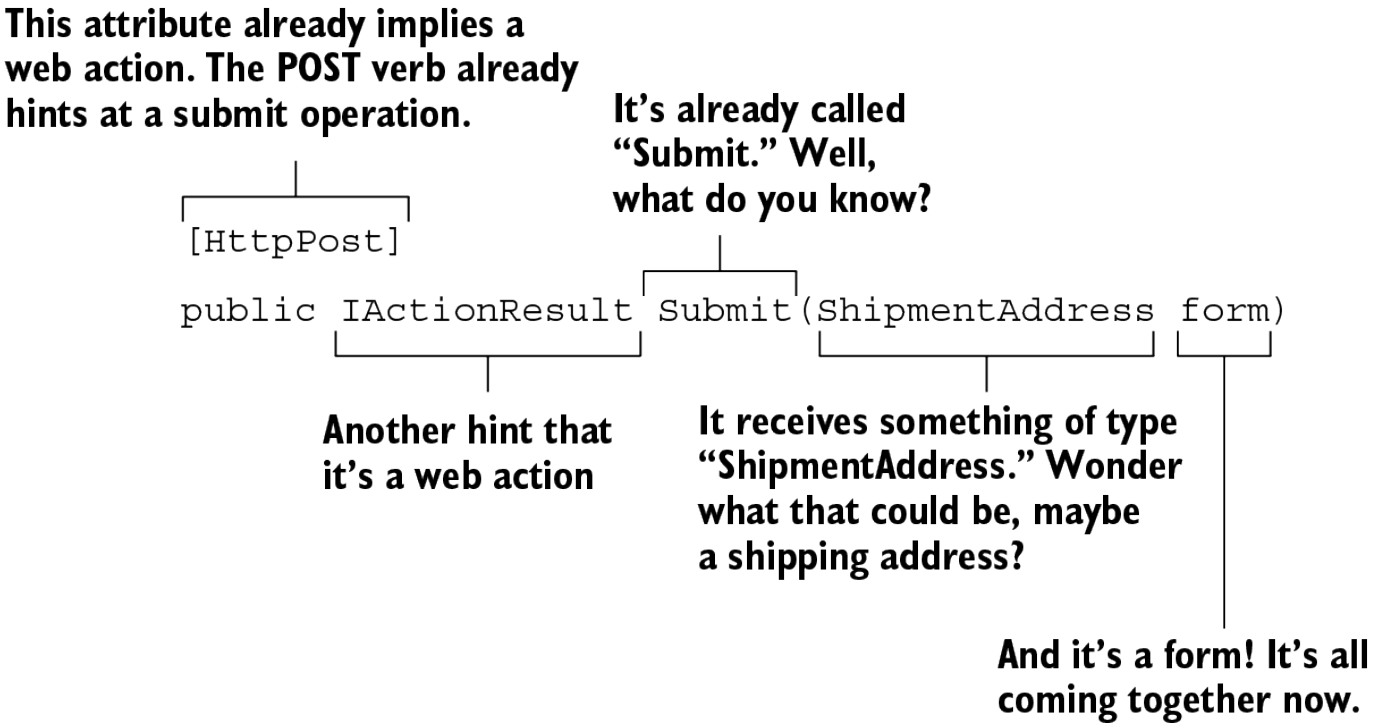
Sinan architekt mógł być pierwszą osobą w historii, która użyła komentarzy w kodzie prawidłowo. Rozważmy przeciwny przypadek, gdyby budynek był pokryty wszędzie napisami. Drzwi miałyby napis „To jest drzwi". Okna miałyby napis „Okno" nad nimi. Pomiędzy każdą cegłą znajdowała się fiolka z notatką w środku mówiącą: „To są cegły".

Nie musisz pisać komentarzy w kodzie, jeśli twój kod jest wystarczająco samowyjaśniający się. Z drugiej strony, nadmiernymi komentarzami możesz zaszkodzić czytelności kodu. Nie pisz komentarzy w kodzie tylko dla samego pisania komentarzy. Używaj ich mądrze i tylko wtedy, gdy jest to konieczne.

Rozważ przykład w poniższym spisie. Gdybyśmy przesadzili z komentarzami w kodzie, mogłoby to wyglądać tak.

```
[HttpPost]
public IActionResult Submit(ShipmentAddress form) {
    // Funkcja obsługująca błędy, która zapisuje plik cookie
    // i przekierowuje użytkownika z powrotem do formularza
    // wprowadzania informacji o przesyłce.
    IActionResult obslugaBledow() {
        Response.Cookies.Append("shipping_error", "1");
        return RedirectToAction("Index", "ShippingForm", form);
    }

    // Sprawdź, czy stan modelu jest prawidłowy.
    if (!ModelState.IsValid) {
        return obslugaBledow();
    }

    // Sprawdź poprawność formularza z wykorzystaniem logiki
    // walidacji po stronie serwera.
    var wynikWalidacji = service.ValidateShippingForm(form);
    // Czy walidacja zakończyła się sukcesem?
    if (wynikWalidacji != ShippingFormValidationResult.Valid) {
        return obslugaBledow();
    }

    // Zapisz informacje o przesyłce.
    bool sukces = service.SaveShippingInfo(form);
    if (!sukces) {
        // Nie udało się zapisać. Zgłoś błąd użytkownikowi.
        ModelState.AddModelError("", "Wystąpił problem podczas " +
            "zapisywania informacji, spróbuj ponownie");
        return obslugaBledow();
    }

    // Przejdź do formularza płatności.
    return RedirectToAction("Index", "BillingForm");
}
```

Kod, który czytamy, opowiada nam historię nawet bez komentarzy. Przejdźmy przez ten sam kod bez komentarzy i odkryjmy ukryte wskazówki w nim (rysunek 3.13).



To może wyglądać na dużo pracy. Próbujesz połączyć różne elementy, aby zrozumieć, co robi kod. Z czasem będzie łatwiej. Im lepiej się w tym znajdziesz, tym mniej wysiłku będziesz musiał wkładać. Istnieją jednak pewne rzeczy, które możesz zrobić, aby ułatwić życie biednej duszy, która czyta twój kod, a nawet sobie samemu, za pół roku, ponieważ po pół roku może to być równie dobrze kod kogoś innego.

3.9.1 Wybieraj świetne nazwy

Dotknąłem znaczenia dobrych nazw na początku tego rozdziału, o tym, jak nasze nazwy powinny jak najdokładniej odzwierciedlać lub podsumowywać funkcjonalność. Funkcje nie powinny mieć dwuznacznych nazw jak Process, DoWork, Make, i tak dalej, chyba że kontekst jest absolutnie jasny. Czasami może to wymagać od Ciebie wpisania dłuższych nazw niż zwykle, ale zazwyczaj można tworzyć dobre nazwy, zachowując jednocześnie zwięzłość.

To samo dotyczy nazw zmiennych. Rezerwuj jednoliterowe nazwy zmiennych tylko dla zmiennych pętli (i, j, n) oraz współrzędnych takich jak x, y i z, gdzie są oczywiste. W przeciwnym razie zawsze wybieraj opisową nazwę i unikaj skrótów. W porządku jest korzystanie z dobrze znanych skrótów takich jak HTTP i JSON lub dobrze znanych skrótów takich jak ID i DB, ale nie skracaj słów. Nazwę zmiennej i tak wpisujesz tylko raz. Później za resztę może zadbać automatyczne uzupełnianie kodu. Korzyści płynące z opisowych nazw są ogromne. Co najważniejsze, oszczędzają Ci czas. Gdy wybierasz opisową nazwę, nie musisz pisać komentarza w pełnym zdaniu, aby wyjaśnić ją tam, gdzie jest używana. Przejrzyj dokumentację dotyczącą konwencji języka programowania, którego używasz. Przykładowo, wytyczne Microsoftu dotyczące konwencji nazewnictwa w .NET są doskonałym punktem wyjścia dla C#: https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/naming-guidelines.

3.9.2 Wykorzystuj funkcje

Małe funkcje są łatwiejsze do zrozumienia. Postaraj się, aby funkcja była na tyle mała, aby zmieścić się na ekranie programisty. Przewijanie w górę i w dół jest okropne dla zrozumienia tego, co robi kod. Powinieneś być w stanie zobaczyć wszystko, co funkcja robi, bez przewijania ekranu.

Jak skrócić funkcję? Początkujący mogą być skłonni umieścić jak najwięcej na jednej linii, aby funkcja była bardziej skompresowana. Nie! Nigdy nie umieszczaj wielu instrukcji w jednej linii. Zawsze miej co najmniej jedną linię na jedną instrukcję. Możesz nawet dodawać puste linie w funkcji, aby grupować ze sobą odpowiednie instrukcje. W związku z tym przyjrzyjmy się naszej funkcji w następnym przykładzie.

```
[HttpPost]
public IActionResult Submit(ShipmentAddress form) {
  IActionResult error() {
    Response.Cookies.Append("shipping_error", "1");
    return RedirectToAction("Index", "ShippingForm", form);
  }
  if (!ModelState.IsValid) {
    return error();
  }
  var validationResult = service.ValidateShippingForm(form);
  if (validationResult != ShippingFormValidationResult.Valid) {
    return error();
  }
  bool success = service.SaveShippingInfo(form);
  if (!success) {
    ModelState.AddModelError("", "Problem occurred while " +
      "saving your information, please try again");
    return error();
  }
  return RedirectToAction("Index", "BillingForm");
}
```

Rzeczywista funkcja jest tak prosta, że prawie czyta się jak zdanie w języku angielskim—no cóż, może jak hybryda angielskiego i tureckiego, ale nadal bardzo czytelne. Osiągnęliśmy bardzo opisowy kod, nie pisząc ani jednej linii komentarza, i to jest klucz, który powinieneś mieć na uwadze, jeśli zastanawiasz się, czy to jest zbyt dużo pracy. To mniej pracy niż pisanie akapitów komentarza. Będziesz również wdzięczny sobie samemu, potrząsając lewą ręką prawą ręką, kiedy dowiesz się, że nie musisz utrzymywać komentarzy i kodu zgodnych, aby komentarze pozostały przydatne przez cały okres trwania projektu. To o wiele lepsze rozwiązanie.

Wyodrębnianie funkcji może wydawać się żmudne, ale w rzeczywistości jest łatwe dzięki środowiskom programistycznym, takim jak Visual Studio. Wystarczy zaznaczyć fragment kodu, który chcesz wyodrębnić, i nacisnąć Ctrl-. (kropka) lub wybrać ikonę żarówki pojawiającą się obok kodu i wybrać opcję Wyodrębnij Metodę. Wszystko, co musisz zrobić, to nadać jej nazwę.

Gdy wyodrębniasz te fragmenty, otwierasz także możliwość ponownego wykorzystania ich w tym samym pliku, co może zaoszczędzić ci czas, gdy piszesz formularz do faktur, jeśli semantyka obsługi błędów nie różni się.

To wszystko może brzmieć, jakbym był przeciwny komentarzom w kodzie. To dokładnie przeciwnie. Unikanie niepotrzebnych komentarzy sprawia, że użyteczne komentarze błyszczą jak klejnoty. To jedyny sposób, aby komentarze były użyteczne. Kiedy piszesz komentarze, myśl jak Sinan: "Czy ktoś będzie potrzebował wyjaśnienia dla tego?" Jeśli to wymaga wyjaśnienia, bądź jak najbardziej jasny, być może rozwlekły, nawet rysuj diagramy ASCII, jeśli to konieczne. Napisz tyle akapitów, ile potrzebujesz, tak aby inni programiści pracujący nad tym samym kodem nie musieli przychodzić do twojego biurka i pytać, co ten fragment kodu robi lub nie naprawiać go błędnie, dlatego że zapomniałeś się wyjaśnić. Gdy produkcja pada, to od ciebie zależy, aby poprawnie naprawić kod. Jesteś to winien zarówno sobie, jak i wszystkim innym.

Są przypadki, w których musisz pisać komentarze, czy są one użyteczne czy nie, takie jak publiczne interfejsy API, ponieważ użytkownicy mogą nie mieć dostępu do kodu. Ale to także nie oznacza, że napisanie komentarzy sprawia, że twój kod staje się łatwy do zrozumienia. Nadal musisz pisać czytelny kod z małymi, łatwymi do strawienia fragmentami.

Podsumowanie:

- Unikaj tworzenia sztywnego kodu, przestrzegając granic logicznych zależności.
- Nie obawiaj się rozpoczęcia pracy od podstaw, ponieważ za każdym razem będzie to szło znacznie szybciej.
- Rozbijaj kod, gdy istnieją zależności, które w przyszłości mogą sprawić problemy, a następnie je napraw.
- Unikaj pogłębiania się w dołku dziedzictwa, regularnie aktualizuj kod i rozwiązuj problemy, które powoduje.
- Unikaj powtarzania kodu, zamiast tego przemyślaj jego ponowne użycie, aby unikać naruszania logicznych odpowiedzialności.
- Twórz inteligentne abstrakcje, aby przyszły kod wymagał mniej czasu. Używaj abstrakcji jako inwestycji.
- Nie pozwól, aby zewnętrzne biblioteki, które używasz, dyktowały twoje projekty.
- Preferuj kompozycję nad dziedziczeniem, aby uniknąć wiązania kodu z określoną hierarchią.
- Staraj się zachować styl kodu, który jest łatwy do czytania od góry w dół.
- Zakończ funkcje wcześniej i unikaj używania instrukcji else.
- Używaj goto lub, jeszcze lepiej, lokalnych funkcji, aby trzymać wspólny kod w jednym miejscu.
- Unikaj bezcelowych, zbędnych komentarzy w kodzie, które sprawiają, że trudno odróżnić drzewo od lasu.
- Pisz samoopisujący się kod, wykorzystując dobre nazewnictwo zmiennych i funkcji.
- Dziel funkcje na łatwe do przyswajania podfunkcje, aby zachować kod jak najbardziej opisowy.
- Dodawaj komentarze w kodzie, gdy są one przydatne.

1. Hacker News to platforma udostępniająca wiadomości związane z technologią, gdzie każdy jest ekspertem we wszystkim: https://news.ycombinator.com.
2. Aplikacja czatowa o nazwie Yo, w której można było wysyłać tylko tekst zawierający "Yo", została kiedyś wyceniona na 10 milionów dolarów. Firma została zamknięta w 2016 roku: https://en.wikipedia.org/wiki/Yo_(app).

3. To doskonała wariacja na słynne słowa Phila Karltona stworzona przez Leona Bambricka (https://twitter.com/secretGeek/status/7269997868). Phil Karlton oryginalnie wypowiedział te słowa bez części dotyczącej "błędów o jeden": [brak linku].