O'REILLY®

C++

Projektowanie oprogramowania

Zasady i wzorce projektowe

Helion

Klaus Iglberger

Tytuł oryginału: C++ Software Design: Design Principles and Patterns for High-Quality Software

Tłumaczenie: Piotr Rajca

ISBN: 978-83-8322-720-7

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

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

Printed in Poland.

Spis treści

ROZDZIAŁ 1

Sztuka projektowania oprogramowania

Czym jest projekt oprogramowania? I dlaczego należy zwracać na niego uwagę? W tym rozdziale przygotuję podstawę dla dalszej części niniejszej książki poświęconej projektowaniu oprogramowania. Opiszę w nim ogólnie, czym jest projektowanie oprogramowania, oraz wyjaśnię, dlaczego ma ono kluczowe znaczenie dla pomyślnej realizacji projektu i dlaczego jest czymś, co należy zrobić dobrze. Jednak czytając ten rozdział, przekonasz się także, że projektowanie oprogramowania jest trudne i złożone. Bardzo złożone. W rzeczywistości jest ono najbardziej złożonym aspektem wytwarzania oprogramowania. Dlatego w tym rozdziale wyjaśnię także kilka zasad projektowania oprogramowania, które ułatwią Ci podążanie w odpowiednim kierunku.

W podrozdziale pt. „Wytyczna 1.: Znaczenie projektu oprogramowania" skoncentruję się na zagadnieniach ogólnych i wyjaśnię, dlaczego należy się spodziewać, że w oprogramowaniu będą zachodzić zmiany. W efekcie oprogramowanie powinno zapewniać możliwość wprowadzania tych zmian. Jednak znacznie łatwiej to powiedzieć, niż zrobić, gdyż w rzeczywistości powiązania i zależności bardzo mocno utrudniają programistom życie. Właśnie tym problemem zajmuje się projektowanie oprogramowania. Przedstawię je zatem jako sztukę zarządzania zależnościami i abstrakcjami — jeden z kluczowych elementów inżynierii oprogramowania.

W podrozdziale pt. „Wytyczna 2.: Projektuj pod kątem zmian" skoncentruję się na powiązaniach i zależnościach i postaram się pomóc Ci zrozumieć, jak należy projektować pod kątem zmian i jak sprawić, by tworzone oprogramowanie było bardziej elastyczne. W tym celu przedstawię zasadę jednej odpowiedzialności (ang. *single responsibility principle*; określaną skrótowo jako *SRP*) oraz zasadę „nie powtarzaj się" (ang. *don't repeat yourself*, w skrócie *DRY*), które mogą Ci ułatwić osiągnięcie tego celu.

W podrozdziale pt. „Wytyczna 3.: Separuj interfejsy w celu unikania sztucznych powiązań" rozszerzę rozważania dotyczące powiązań, a konkretnie: skoncentruję się na powiązaniach wynikających ze stosowania interfejsów. Wprowadzę w nim także zasadę segregacji interfejsów (ang. *interface segregation principle*, w skrócie *ISP*), pozwalającą na ograniczanie sztucznych powiązań generowanych przez interfejsy.

W podrozdziale pt. „Wytyczna 4.: Projektuj pod kątem łatwości testowania" skoncentruję się na problemach utrudniających testowanie i stanowiących efekt sztucznych powiązań. W szczególności zastanowię się nad tym, w jaki sposób należy testować prywatną funkcję składową, i zademonstruję, że jedynym prawdziwym rozwiązaniem jest konsekwentne stosowanie separacji zagadnień.

W podrozdziale pt. „Wytyczna 5.: Projektuj pod kątem rozszerzania” zajmiemy się ważnym rodzajem zmian: rozszerzaniem. Kod powinien zapewniać możliwość nie tylko łatwego wprowadzania zmian, lecz także łatwego rozszerzania. W tym podrozdziale pokażę, w jaki sposób można osiągnąć ten cel, i zaprezentuję zalety kolejnej zasady projektowej: zasady otwarte-zamknięte (ang. *open-close principle*, w skrócie *OCP*).

Wytyczna 1.: Znaczenie projektu oprogramowania

Gdybym Cię zapytał, jakie właściwości kodu są dla Ciebie najważniejsze, to prawdopodobnie po pewnym zastanowieniu odpowiedziałbyś, że będzie to przejrzystość kodu, łatwość testowania go, utrzymania, rozszerzania, możliwość wielokrotnego stosowania oraz skalowalność. I w zupełności bym się z taką odpowiedzią zgodził. Jednak gdybym następnie zapytał Cię, w jaki sposób osiągnąć ten cel, to bardzo możliwe, że zacząłbyś wymieniać takie cechy C++ jak wzorzec RAII, algorytmy, wyrażenia lambda, moduły itd.

Cechy to nie projekt oprogramowania

Owszem, język C++ udostępnia wiele możliwości. Bardzo wiele! Około połowy z 2 tysięcy stron, jakie zajmuje wydrukowany opis standardu języka C++, jest poświęcone wyjaśnieniu mechaniki języka oraz jego możliwości[1]. A w momencie udostępnienia wersji C++ 11 obiecano, że tych możliwości będzie więcej: co trzy lata komitet standaryzacyjny zajmujący się rozwojem C++ przekazuje nam nowy standard języka, obejmujący dodatkowe, zupełnie nowe możliwości. Dlatego nie jest wielkim zaskoczeniem, że społeczność programistów C++ przykłada bardzo dużą wagę do możliwości oraz mechaniki języka. Większość książek, prelekcji oraz blogów koncentruje się na możliwościach, nowych bibliotekach oraz szczegółach języka[2].

Niemal czuję, jakby możliwości były jednym z najważniejszych aspektów programowania w C++ i miały kluczowe znaczenie dla pomyślności projektów pisanych w tym języku. Jednak szczerze mówiąc, wcale tak nie jest. Ani znajomość wszystkich możliwości języka, ani wybór standardu języka nie zagwarantują pomyślności projektu. Nie — nie powinieneś oczekiwać, że cechy i możliwości języka uchronią Twój projekt od katastrofy. Wprost przeciwnie: projekt może się zakończyć ogromnym sukcesem, nawet jeśli będzie używał starszych wersji standardu C++, i to nawet w przypadkach, kiedy będzie w nim stosowany tylko fragment wszystkich dostępnych możliwości. Pomijając całkowicie ludzki aspekt programowania, znacznie większy wpływ na pomyślność lub porażkę projektu ma ogólna *struktura* tworzonego oprogramowania. To właśnie ona jest odpowiedzialna za możliwości utrzymania projektu — za to, jak łatwo będzie można zmieniać kod,

[1] Oczywiście nikt z nas nigdy nie spróbowałby wydrukować standardu C++. Skorzystalibyśmy bądź to z dokumentu PDF zawierającego oficjalny standard (*https://isocpp.org/std/the-standard*), bądź też z jego bieżącego projektu roboczego (*http://eel.is/c++draft/*). Jednak podczas zwyczajnej, codziennej pracy najlepiej byłoby zapewne skorzystać z witryny poświęconej językowi C++ — *https://en.cppreference.com/w*.

[2] Niestety nie jestem w stanie podać tu żadnych liczb, gdyż nie mogę z czystym sumieniem stwierdzić, że posiadłem kompletną znajomość ogromnego królestwa języka C++. Wprost przeciwnie! Jest całkiem możliwe, że nie w pełni znam nawet te źródła, o których istnieniu wiem! Dlatego te informacje należy uznać jedynie za moje wrażenia oraz przejaw sposobu, w jaki postrzegam społeczność C++. Być może Twoja opinia będzie zupełnie inna.

rozszerzać go oraz testować. Bez możliwości łatwego modyfikowania kodu, dodawania do niego nowych funkcjonalności oraz wiary w poprawność jego działania, zapewnianą przez testy, projekt znajdzie się na końcu swojego cyklu istnienia. Struktura decyduje także o skalowalności projektu programistycznego: Jak bardzo może się on rozrosnąć, nim zapadnie się pod własnym ciężarem? Ile osób może pracować nad urzeczywistnieniem wizji projektu, nim wzajemnie zaczną sobie przeszkadzać?

Ta ogólna struktura jest określana jako projekt oprogramowania. Ten projekt odgrywa znacznie ważniejszą rolę w sukcesie przedsięwzięcia — projektu programistycznego. Kiedy mowa o dobrym oprogramowaniu, nie chodzi przede wszystkim o odpowiednie zastosowanie jakichkolwiek możliwości, ale raczej o dobrą architekturę i dobry projekt oprogramowania. Dobry projekt oprogramowania może wyeliminować negatywne skutki niektórych błędnych decyzji implementacyjnych, jednak skutków kiepskiej jakości projektu nie można zrównoważyć nawet najdoskonalszym wykorzystaniem samych możliwości języka (zarówno tych starych, jak i najnowszych).

Projektowanie oprogramowania: Sztuka zarządzania zależnościami i abstrakcjami

Dlaczego projekt oprogramowania ma tak kluczowe znaczenie dla jakości projektu programistycznego? No cóż, zakładając, że obecnie wszystko działa idealnie, to dopóki w oprogramowaniu nic się nie zmieni i nic nie trzeba będzie do niego dodać, będziemy bezpieczni. Jednak najprawdopodobniej to nie potrwa zbyt długo. Z dużą dozą pewności można założyć, że coś się zmieni. W końcu zmiany są jedynym pewnikiem wytwarzania oprogramowania. Zmiany są podstawową przyczyną wszystkich naszych problemów (kryją się także za większością opracowywanych rozwiązań). To właśnie dlatego w języku angielskim oprogramowanie jest określane jako *software* — gdyż w odróżnieniu od sprzętu (ang. *hardware*) jest czymś niestałym i podatnym na zmiany. I owszem — oprogramowanie ma się adaptować i zapewniać możliwość dostosowywania do nieustannie zmieniających się wymagań. Jednak jak zapewne wiesz, w rzeczywistości takie oczekiwania nie zawsze są zgodne z prawdą.

Aby to zilustrować, wyobraźmy sobie, że z systemu do zarządzania problemami wybraliśmy zgłoszenie, którego poziom trudności zespół ocenił na 2. Niezależnie od tego, co ta liczba oznacza w naszym projekcie, bez wątpienia nie wygląda to na wielki problem, więc możemy mieć pewność, że zadanie zrealizujemy stosunkowo szybko. Z pozytywnym nastawieniem poświęcamy nieco czasu na zrozumienie zadania, a następnie zaczynamy wprowadzać zmiany w jakimś fragmencie kodu A. Dzięki natychmiastowym informacjom zwrotnym zapewnianym przez testy (jakie to szczęście, że dysponujemy testami!) szybko się dowiadujemy, że musimy także rozwiązać problem, który pojawił się we fragmencie kodu B. To nas zaskakuje. Nie sądziliśmy, że wprowadzane modyfikacje wywołają jakieś efekty w tym fragmencie kodu. Niemniej jednak modyfikujemy go i dostosowujemy do nowych wymagań. A później, podczas nocnej próby zbudowania aplikacji, okazuje się, że przestały działać fragmenty kodu C i D. Teraz, nim zabierzemy się do pracy, nieco dokładniej badamy problem — okazuje się, że jego pierwotne przyczyny obejmują znacznie większe części bazy kodu. To niewielkie, początkowo wyglądające na łatwe zadanie przekształciło się w dużą,

potencjalnie ryzykowną modyfikację[3]. Nasza pewność i wiara w rozwiązanie zadania szybko znika… podobnie jak plany, które robiliśmy na dalszą część tygodnia.

Być może ta opowieść wyda Ci się znajoma. Być może będziesz nawet w stanie przytoczyć kilka podobnych przykładów, zaczerpniętych z własnych doświadczeń. W rzeczywistości większość programistów zna podobne historie. A przyczyna większości z nich jest taka sama. Zazwyczaj problem ten można opisać jednym słowem: *zależności*. Jak napisał Kent Beck w swojej książce poświęconej programowaniu opartemu na testach[4]:

> Zależności są kluczowym problemem wytwarzania oprogramowania, niezależnie od skali.

Zależności są zmorą każdego twórcy oprogramowania. Mógłbyś się spierać, że występowanie zależności to oczywistość, że zawsze będą występować, w końcu gdyby ich nie było, to nie byłoby możliwe współdziałanie różnych elementów kodu. I oczywiście miałbyś rację. Różne fragmenty kodu muszą ze sobą współpracować, a te iteracje zawsze będą powodować powstawanie jakichś powiązań. Jednak oprócz koniecznych, nieuniknionych zależności pojawiają się także zależności sztuczne, które wprowadzamy w wyniku braku znajomości rozwiązywanego problemu albo szerszego, bardziej ogólnego spojrzenia na ten problem czy też po prostu przez nieuwagę. Nie trzeba wspominać, że te sztuczne zależności są niekorzystne. Utrudniają zrozumienie oprogramowania, wprowadzanie w nim zmian, dodawanie nowych możliwości i pisanie testów. Dlatego jednym z głównych zadań twórcy oprogramowania, a może nawet *najważniejszym* z nich, jest maksymalne zredukowanie liczby sztucznych zależności.

Ta minimalizacja zależności jest celem architektury i projektowania oprogramowania. Robert C. Martin[5] wyraził to w następujący sposób:

> Celem architektury oprogramowania jest minimalizacja zasobów ludzkich niezbędnych do stworzenia i utrzymania wymaganego systemu.

Architektura i projekt oprogramowania są narzędziami potrzebnymi do zminimalizowania nakładu pracy w każdym projekcie. Odnoszą się one do zależności i redukują złożoność, wykorzystując do tego celu abstrakcje. Ja opisuję to w następujący sposób[6]:

> Projektowanie oprogramowania to sztuka zarządzania wzajemnymi zależnościami pomiędzy komponentami tworzonego oprogramowania. Jego celem jest minimalizacja sztucznych (technicznych) zależności i wprowadzanie niezbędnych abstrakcji oraz kompromisów.

---

[3] To, czy modyfikacja jest ryzykowna, czy nie, w znacznej mierze będzie zależeć od stopnia pokrycia kodu testami. Jeśli to pokrycie będzie duże, to testy mogą zminimalizować zgubne efekty złego projektu oprogramowania.

[4] Kent Beck, *TDD. Sztuka tworzenia dobrego kodu*, Helion, Gliwice 2014.

[5] Robert C. Martin, *Czysta architektura. Struktura i design oprogramowania. Przewodnik dla profesjonalistów*. Helion, Gliwice 2018.

[6] To faktycznie są moje własne słowa, gdyż nie istnieje jedna, wspólna definicja projektowania oprogramowania. W efekcie możesz sformułować własną definicję tego, czym jest projektowanie oprogramowania, i nie będzie w tym nic złego. Pamiętaj jednak, że niniejsza książka, jak również zaprezentowane w niej opisy wzorców projektowych opierają się na powyższej, mojej definicji.

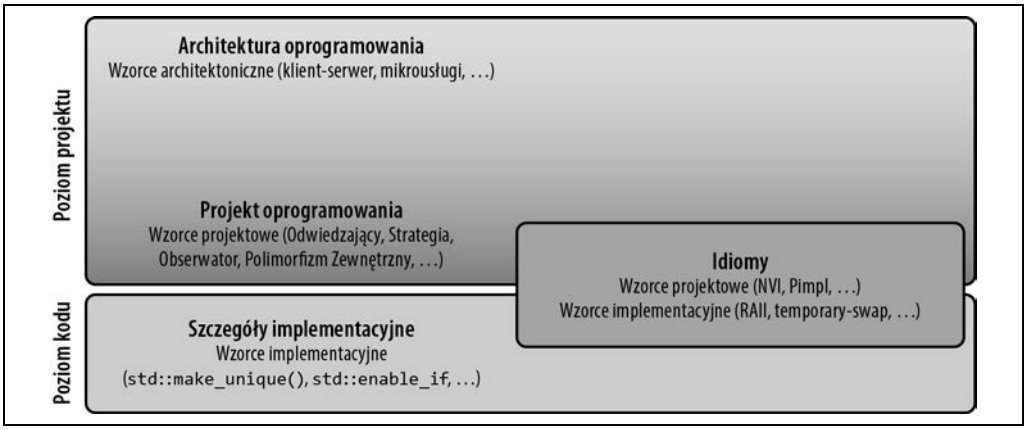
Owszem, projektowanie oprogramowania to sztuka. Nie jest to nauka i nie można podać żadnego zestawu prostych i jasnych odpowiedzi[7]. Bardzo często ogólnie pojęte projektowanie oprogramowania zwodzi nas, onieśmiela złożonością wzajemnych powiązań pomiędzy poszczególnymi elementami tworzonego rozwiązania. Jednak staramy się radzić sobie z tą złożonością i redukować ją przez wprowadzanie odpowiednich abstrakcji. W ten sposób utrzymujemy szczegółowość na rozsądnym poziomie. Niemniej jednak bardzo często zdarza się, że poszczególni programiści wchodzący w skład zespołu mają różne opinie na temat architektury oraz projektowania oprogramowania. Może się okazać, że nie będziemy w stanie wcielić w życie własnej wizji projektu oraz że zapewnienie postępu prac będzie wymagało kompromisów.

Termin **abstrakcja** może być używany w różnych kontekstach. Stosuje się go w odniesieniu do organizowania funkcjonalności i poszczególnych informacji w formie typów danych i funkcji. Jednak można go także używać do opisywania modelowania często występujących zachowań oraz reprezentacji zbiorów wymagań i oczekiwań. W niniejszej książce, poświęconej projektowaniu oprogramowania, będę go używał głównie w tym drugim znaczeniu (dotyczy to w szczególności rozdziału 2.).

Zwróć uwagę, że w powyższych cytatach słowa *architektura* oraz *projekt oprogramowania* mogą być stosowane zamiennie, gdyż mają zbliżony sens i te same cele. Jednak nie są tym samym. Ich podobieństwa, lecz także różnice pomiędzy nimi, staną się jasne, kiedy przyjrzymy się trzem poziomom projektowania oprogramowania.

Trzy poziomy projektowania oprogramowania

Architektura oprogramowania oraz **projekt oprogramowania** to dwa spośród trzech poziomów procesu wytwarzania oprogramowania. Ich dopełnieniem jest trzeci poziom — **szczegółów implementacyjnych**. Wszystkie te trzy poziomy przedstawiłem na rysunku 1.1.



Rysunek 1.1. Trzy poziomy procesu wytwarzania oprogramowania: architektura oprogramowania, projekt oprogramowania oraz szczegóły implementacyjne. Idiomami mogą być wzorce projektowe lub implementacyjne

[7] Dla jasności: informatyka jest nauką. Jednak *wytwarzanie* oprogramowania wydaje się hybrydą — połączeniem nauki, rzemiosła i sztuki. A *projektowanie* oprogramowania jest jednym z elementów tej ostatniej — sztuki.

Aby łatwiej Ci było wyobrazić sobie te trzy poziomy, zacznijmy od przedstawienia rzeczywistego przykładu związków pomiędzy architekturą, projektem oprogramowania oraz szczegółami implementacyjnymi. Wyobraź sobie, że pełnisz funkcję architekta. I nie, nie wyobrażaj sobie, że siedzisz w wygodnym fotelu przed komputerem z kubkiem kawy w ręce, a zamiast tego pomyśl, że jesteś gdzieś na terenie budowy. Tak, mam na myśli architekta zajmującego się projektowaniem budynków[8]. Jako taki architekt byłbyś odpowiedzialny za wszelkie istotne właściwości budynku: jego integrację z otoczeniem, jego integralność strukturalną, rozmieszczenie pomieszczeń, kanalizację itd. Byłbyś także odpowiedzialny za atrakcyjny wygląd i funkcjonalność — zapewne pomyślałbyś o dużym salonie, łatwym dostępie pomiędzy kuchnią i jadalnią itd. Innymi słowy, musiałbyś zadbać o ogólne cechy architektoniczne — rzeczy, które trudno byłoby zmienić później — jak również o drobne szczegóły projektu tworzonego budynku. Jednak wskazanie różnic pomiędzy tymi dwiema kategoriami jest dość trudne: granica pomiędzy architekturą a projektem wydaje się płynna i określona w mało precyzyjny sposób.

Jednak te decyzje stanowiłyby koniec Twojego zakresu obowiązków. Jako architekt nie przejmowałbyś się tym, gdzie ustawić lodówkę, telewizor oraz poszczególne meble. Nie zajmowałbyś się takimi szczegółami jak to, w jakich miejscach należy powiesić obrazy czy też gdzie umieścić inne dekoracje. Innymi słowy, nie zajmowałbyś się szczegółami; zadbałbyś jedynie o to, by właściciel domu otrzymał odpowiednią strukturę, w której mógłby żyć.

Meble oraz wszelkie inne „szczegóły" zastosowane w powyższej metaforze odpowiadają najniższemu i najbardziej konkretnemu poziomowi wytwarzania oprogramowania — szczegółom implementacyjnym. Ten poziom zajmuje się tym, w jaki sposób rozwiązanie zostanie zaimplementowane. To my musimy wybrać niezbędny (i dostępny) standard C++ bądź też jego określony podzbiór, jak również odpowiednie możliwości języka, słowa kluczowe, których trzeba będzie używać, a także zająć się innymi aspektami, takimi jak przydzielanie pamięci, obsługa wyjątków, wydajność itd. To także poziom, na którym są stosowane **wzorce implementacyjne**, takie jak `std::make_unique()`, używany jako **funkcja wytwórcza**, `std::enabled_if`, stosowany jako sposób pozwalający na bezpośrednie korzystanie z mechanizmu SFINAE, itd.[9]

W przypadku projektowania oprogramowania początkowo koncentrujemy się na obrazie ogólnym. Na tym poziomie najważniejsze są zagadnienia związane z łatwością utrzymania, możliwościami wprowadzania zmian, testowania i skalowalnością. Projektowanie oprogramowania w głównej mierze dotyczy interakcji pomiędzy elementami oprogramowania, które w odniesieniu do przedstawionej wcześniej metafory można by porównać z pomieszczeniami, drzwiami, rurami oraz

[8] Używając tej metafory, nie staram się sugerować, że architekt zajmujący się projektowaniem budynków spędza całe dnie na budowie. Jest wysoce prawdopodobne, że taki architekt spędza dziennie równie dużo czasu w wygodnym fotelu przed ekranem komputera, co ja czy Ty. Jednak sądzę, że zrozumiałeś o co mi chodzi.

[9] SFINAE, co jest skrótem od angielskiego terminu *Substitution Failure Is Not An Error* (niepowodzenie podstawienia nie jest błędem), to podstawowy mechanizm szablonów, powszechnie wykorzystywany do wprowadzania w nich ograniczeń, będący zamiennikiem konceptów C++20. Wszelkich informacji dotyczących mechanizmu SFINEA oraz `std::enabled_if` powinieneś poszukać w swojej ulubionej książce poświęconej szablonom w C++. Jeśli takiej nie masz, to doskonałym wyborem będzie biblia szablonów C++ — książka autorstwa Davida Vandevoorde, Nicolaia Josuttisa i Douglasa Gregora pt. *C++ Templates: The Complete Guide*, wydana przez wydawnictwo Addison-Wesley.

kablami. Na tym poziomie zajmujemy się fizycznymi, logicznymi zależnościami komponentów (klas, funkcji itp.)[10]. To poziom, na którym uwzględnia się wzorce projektowe, takie jak **Odwiedzający**, **Strategia** czy **Dekorator**, definiujące strukturę zależności pomiędzy elementami oprogramowania — zagadnienia opisane dokładniej w rozdziale 3. Wzorce te, które zazwyczaj można stosować w różnych językach programowania, pomagają dzielić bardziej złożone rozwiązania na prostsze fragmenty.

Spośród wszystkich trzech poziomów *architektura oprogramowania* jest tym, który jest najtrudniej opisać słowami i precyzyjnie zdefiniować. Dzieje się tak, gdyż nie istnieje żadna ogólnie przyjęta definicja architektury oprogramowania. Choć można wskazać wiele różnych poglądów na temat tego, czym ona jest, to istnieje tylko jeden jej aspekt, co do którego wszyscy wydają się zgadzać: architektura zazwyczaj wiąże się z decyzjami o dużej skali, tymi aspektami tworzonego oprogramowania, które w przyszłości będzie najtrudniej zmienić:

> Architektura to decyzje, które chcielibyśmy podjąć na wczesnych etapach prac nad projektem, a które najprawdopodobniej nie będą bardziej poprawne niż wszelkie pozostałe[11].
>
> — Ralph Johnson

Na poziomie architektury oprogramowania używamy wzorców architektonicznych takich jak **klient-serwer**, **mikrousługi** itd.[12] Wzorce te stanowią także odpowiedź na pytanie, jak projektować systemy, w których można będzie zmienić jakiś element bez wpływu na wszystkie pozostałe. Podobnie do **wzorców projektowych oprogramowania**, także wzorce architektoniczne określają strukturę i wzajemne zależności pomiędzy elementami. Jednak w odróżnieniu od wzorców projektowych, wzorce architektoniczne zazwyczaj operują na kluczowych graczach — dużych elementach tworzonego oprogramowania (takich jak moduły i komponenty, a nie klasy i funkcje).

Patrząc z tej perspektywy, architektura oprogramowania reprezentuje ogólną strategię naszego podejścia do wytwarzanego systemu, natomiast projekt oprogramowania można by porównać do taktyki zapewniającej działanie tej strategii. Jednak problem z takim punktem widzenia polega na tym, że nie istnieje definicja słowa „duże". Zwłaszcza w kontekście rosnącej popularności mikrousług wyznaczenie granicy pomiędzy dużymi a małymi elementami wytwarzanego oprogramowania staje się coraz trudniejsze[13].

Dlatego architekturę często się opisuje jako to, co doświadczeni programiści zajmujący się projektem postrzegają jako kluczowe decyzje.

Czynnikiem, który dodatkowo utrudnia rozgraniczenie architektury, projektu oprogramowania oraz szczegółów, jest pojęcie **idiomu**. **Idiom** oznacza powszechnie używane, lecz zależne od

[10] Znacznie więcej informacji na temat zarządzania zależnościami fizycznymi i logicznymi można znaleźć w książce Johna Lakosa pt. *Large-Scale C++ Software Development: Process and Architecture*, wydanej przez wydawnictwo Addison-Wesley, z charakterystyczną tamą na okładce.

[11] Martin Fowler, *Who Needs an Architect?*, IEEE Software, 20, nr 5 (2003), 11 – 13, *https://ieeexplore.ieee.org/document/1231144*.

[12] Doskonałym wprowadzeniem do zagadnień mikrousług jest książka Sama Newmana pt. *Budowanie mikrousług: Projektowanie drobnoziarnistych systemów. Wydanie II*, wydana przez wydawnictwo Helion.

[13] Mark Richards i Neal Ford, *Fundamentals of Software Architecture: An Engineering Approach*, O'Reilly, 2020.

konkretnego języka, rozwiązanie często występującego problemu. Jako taki idiom także stanowi wzorzec, może on być jednak bądź to **wzorcem implementacyjnym**, bądź **wzorcem projektowym**[14]. W nieco bardziej potocznym ujęciu idiomy C++ są najlepszymi praktykami projektowania lub implementacji, opracowanymi przez społeczność programistów C++. Na przykład istnieje **idiom „kopiuj i zamień"** (ang. *copy-and-swap*, *https://en.wikibooks.org/wiki/More_C++_Idioms/Copy-and-swap*), który być może znasz z implementacji kopiującego operatora przypisania, oraz **idiom RAII** (*https://en.cppreference.com/w/cpp/language/raii*; ang. *resource acquisition is initialization*; ten bezwzględnie powinieneś znać, a jeśli nie znasz, to koniecznie zajrzyj do drugiej ze swoich ulubionych książek o C++[15]). Żaden z tych idiomów nie wprowadza abstrakcji, jak również żaden z nich nie wspomaga separacji. Niemniej jednak są one nieodzowne do pisania dobrego kodu w języku C++.

Już słyszę, jak pytasz: „Czy mógłbyś być nieco bardziej konkretny? Czy RAII nie zapewnia także pewnej formy separacji? Czy nie oddziela zarządzania zasobami od logiki biznesowej?". Masz rację: mechanizm RAII separuje zarządzanie zasobami od logiki biznesowej. Jednak nie uzyskuje tego efektu przez separację, czyli abstrakcję, lecz dzięki użyciu hermetyzacji. Zarówno abstrakcja, jak i hermetyzacja ułatwiają pisanie złożonych systemów tak, by były one łatwiejsze do zrozumienia i modyfikacji, jednak w odróżnieniu od abstrakcji, która rozwiązuje pojawiające się problemy na poziomie projektu oprogramowania, hermetyzacja rozwiązuje je na poziomie szczegółów implementacyjnych. Cytując za Wikipedią (*https://en.wikipedia.org/wiki/Resource_acquisition_is_initialization*):

> Zaletami RAII jako techniki zarządzania zasobami jest to, że: zapewnia hermetyzację, obsługę wyjątków […] oraz lokalność […]. Hermetyzacja jest zapewniana ze względu na to, że logika zarządzania zasobami jest definiowana jeden raz w klasie, a nie w miejscu każdego wywołania.

Choć większość idiomów zalicza się do kategorii szczegółów implementacyjnych, to jednak można wskazać także idiomy zaliczające się do kategorii projektu oprogramowania. Dwoma przykładami takich idiomów mogą być: **idiom Non-Virtual Interface** (w skrócie NVI, interfejs niewirtualny) oraz **idiom Pimpl**. Oba opierają się na klasycznych wzorcach projektowych, odpowiednio: **Metodzie szablonowej** oraz **Most**[16].

---

[14] Termin **wzorce implementacyjne** został po raz pierwszy użyty w książce Kenta Becka pt. *Wzorce implementacyjne* (wydanej przez wydawnictwo Helion). W niniejszej książce używam go po to, by wyraźnie odróżnić to pojęcie od pojęcia **wzorców projektowych**, gdyż termin **idiom** może się odnosić do wzorców stosowanych zarówno na poziomie projektu oprogramowania, jak też na poziomie szczegółów implementacyjnych. Będę konsekwentnie używał tego terminu, odnosząc się do rozwiązań powszechnie stosowanych na poziomie szczegółów implementacyjnych.

[15] Drugiej z ulubionych, oczywiście po niniejszej. Jeśli ta jest Twoją ulubioną, to na przykład możesz zajrzeć do uznawanej za klasykę książki *Effective C++: 55 Specific Ways to Improve Your Programs and Designs, 3rd ed.*, napisanej przez Scotta Mayersa, a wydanej przez wydawnictwo Addison-Wesley.

[16] **Metoda szablonowa** oraz **Most** to dwa spośród 23 klasycznych wzorców projektowych opisanych w książce Bandy Czworga (ang. *Gang of Four*, w skrócie GoF) pt. *Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku*, napisanej przez Ericha Gammę, Richarda Helma, Ralpha Johnsona i Johna Vlissidesa. W tej książce nie będę szczegółowo opisywał wzorca **Metoda szablonowa**, jednak doskonałe informacje na jego temat można znaleźć w wielu publikacjach, w tym we wspomnianej wcześniej książce. Opiszę natomiast wzorzec projektowy **Most**; informacje na jego temat znajdują się w rozdziale 7., w podrozdziale pt. „Wytyczna 28.: Tworzenie mostów w celu wyeliminowania fizycznych zależności".

Zwracanie uwagi na możliwości

Skoro architektura oprogramowania oraz projekt oprogramowania są tak ważne, to dlaczego jesteśmy w społeczności programistów C++, w której tak wiele uwagi zwraca się na możliwości? Dlaczego tworzymy iluzję, że standardy C++, mechanika języka oraz jego możliwości mają kluczowe znaczenie dla projektu? Uważam, że są trzy istotne powody. Przede wszystkim tych możliwości jest tak wiele, a niektóre z nich mają tak bardzo złożone szczegóły, że musimy poświęcać wiele czasu na rozważania o tym, jak należy ich właściwie używać. Musimy stworzyć powszechnie dostępną wiedzę na temat tego, które zastosowania są dobre, a które złe. To my, jako społeczność, musimy wykształcić poczucie idiomatycznego C++.

Drugim powodem jest to, że być może mamy błędne oczekiwania co do możliwości. W ramach przykładu weźmy moduły C++20. Bez wdawania się w szczegóły — tę możliwość faktycznie można uznać za jedną z największych technicznych rewolucji wprowadzonych od momentu opracowania języka C++. Moduły mogą przynajmniej położyć kres wysoce wątpliwej i niezgrabnej praktyce dołączania plików nagłówkowych do kodów źródłowych.

Ze względu na ten potencjał oczekiwania co do tej możliwości są ogromne. Są nawet tacy, którzy sądzą, że moduły mogą ocalić ich projekty przez rozwiązanie problemów z ich strukturą. Jednak zaspokojenie tych oczekiwań wcale nie będzie łatwe: moduły wcale nie poprawiają ani struktury, ani projektu tworzonego kodu — mogą jedynie reprezentować jego bieżącą strukturę i projekt. Moduły nie naprawią naszych błędów projektowych, jednak mogą sprawić, że problemy staną się wyraźniej widoczne. Innymi słowy, moduły nie mogą ocalić naszych projektów. Jak widać, faktycznie możemy mieć względem możliwości zbyt duże lub błędne oczekiwania.

I w końcu trzecim powodem jest to, że niezależnie do wielkiej liczby możliwości oraz stopnia ich złożoności są one i tak niewielkie w porównaniu ze złożonością tworzonego oprogramowania. Znacznie łatwiej jest wyjaśnić konkretny zbiór reguł odnoszących się do konkretnych możliwości, i to niezależnie od tego, jak wiele przypadków szczególnych obejmują, niż wyjaśnić optymalny sposób oddzielania od siebie dwóch elementów oprogramowania.

Choć zazwyczaj można podać dobre odpowiedzi na wszystkie pytania dotyczące możliwości, to przeważnie powszechną odpowiedzią na pytania związane z projektowaniem oprogramowania jest „to zależy". Co więcej, taka odpowiedź nawet nie musi świadczyć o braku doświadczenia, ale raczej o świadomości tego, że najlepszy sposób zapewnienia, by kod był łatwiejszy do utrzymania, modyfikowania i testowania oraz bardziej skalowalny, jest w ogromnym stopniu zależny od licznych czynników związanych z danym projektem. Wzajemne odseparowanie od siebie wielu wzajemnie współdziałających elementów oprogramowania faktycznie może być jednym z największych wyzwań stojących przed ludzkością:

> Projektowanie i programowanie są czynnościami wykonywanymi przez ludzi; jeśli o tym zapomnimy, wszystko zostanie stracone[17].

Według mnie to właśnie kombinacja tych trzech czynników sprawia, że tak dużą uwagę zwracamy na możliwości. Ale proszę, nie zrozum mnie źle. Bynajmniej nie chcę powiedzieć, że możliwości

[17] Bjarne Stroustrup, *Język C++. Kompendium wiedzy. Wydanie IV*, wydawnictwo Helion, Gliwice 2014.

nie są ważne. Wprost przeciwnie — *są*. I owszem: trzeba rozmawiać na ich temat, podobnie jak nauczyć się właściwych sposobów ich używania; niemniej jednak koniecznie trzeba powtórzyć, że same możliwości nie ocalą naszego projektu.

Zwracanie uwagi na projekt oprogramowania oraz zasady projektowe

Choć możliwości są istotne i oczywiście należy o nich rozmawiać, to jednak projekt oprogramowania ma większe znaczenie. Projekt oprogramowania jest kluczowy. Stwierdziłbym nawet, że jest on podstawą sukcesu każdego projektu programistycznego. Dlatego w niniejszej książce spróbuję się skoncentrować właśnie na zagadnieniach związanych z projektem oprogramowania oraz zasadami projektowymi, a nie na możliwościach. Oczywiście wciąż będę przedstawiał dobry i nowoczesny kod C++, niemniej jednak wcale nie będę się koncentrował na stosowaniu najnowszych i najwspanialszych dodatków do języka[18]. Oczywiście *będę* używał wybranych nowych możliwości języka, takich jak *koncepty* (ang. *concepts*) C++20, kiedy będzie to uzasadnione i korzystne, jednak nigdzie *nie będę* zwracał uwagi na stosowanie `noexcept` czy też `constexpr`[19]. Zamiast tego zajmę się trudniejszym aspektem oprogramowania. Postaram się, w większości przypadków, koncentrować na projekcie oprogramowania, uzasadnianiu podejmowanych decyzji projektowych, zasadach projektowych, zarządzaniu zależnościami oraz radzeniu sobie z abstrakcjami.

Podsumowując, projekt to część procesu wytwarzania oprogramowania, która ma kluczowe znaczenie. Programiści powinni doskonale rozumieć zasady projektowania, by móc pisać dobre i łatwe w utrzymaniu programy. Gdyż, w efekcie, dobre oprogramowanie jest tanie, a złe oprogramowanie jest kosztowne.

Wytyczna 1.: Znaczenie projektu oprogramowania

- Projekt oprogramowania należy traktować jako bardzo ważny etap tworzenia oprogramowania.
- Należy w mniejszym stopniu koncentrować się na możliwościach C++, a w większym na projekcie oprogramowania.
- Należy unikać niepotrzebnych powiązań i zależności, by oprogramowanie mogło łatwiej adaptować się do częstych zmian.
- Projekt oprogramowania należy pojmować jako sztukę zarządzania zależnościami i abstrakcjami.
- Warto przyjąć, że granica pomiędzy projektem oprogramowania a architekturą oprogramowania jest płynna.

[18] Z wyrazami uznania dla Johna Lakosa, który przedstawił analogiczną argumentację i w swojej książce *C++. Projektowanie systemów informatycznych. Vademecum profesjonalisty* (wydanej przez wydawnictwo Helion) zdecydował się na zastosowanie języka C++98.

[19] Benie i Jasonie, tak, oczy was nie mylą: nie będę do WSZYSTKIEGO dodawał słowa kluczowego `constexpr`. Patrz wystąpienie Bena Deane'a oraz Jasona Turnera pt. *constexpr ALL the things* na konferencji CppCon 2017 (*https://www.youtube.com/watch?v=PJwd4JLYJJY*).

Wytyczna 2.: Projektuj pod kątem zmian

Jednym z bardzo ważnych oczekiwań co do dobrego oprogramowania jest możliwość łatwego wprowadzania w nim zmian. To oczekiwanie jest nawet jawnie wyrażone przez angielski termin **software**, oznaczający właśnie oprogramowanie. *Software*, w odróżnieniu od komponentów sprzętowych komputera, określanych angielskim terminem **hardware**, powinien zapewniać możliwość łatwej adaptacji do zmieniających się wymagań (patrz podrozdział pt.: „Wytyczna 1.: Znaczenie projektu oprogramowania"). Jednak na podstawie własnych doświadczeń zapewne będziesz mógł stwierdzić, że wprowadzanie zmian w kodzie nie zawsze jest łatwe. Wprost przeciwnie: czasami pozornie prosta zmiana okazuje się przedsięwzięciem, którego przeprowadzenie może zająć kilka długich dni.

Separacja zagadnień

Jednym z najlepszych i doskonale sprawdzonych rozwiązań pozwalających na ograniczanie liczby sztucznych powiązań i upraszczanie wprowadzania zmian w oprogramowaniu jest separacja zagadnień (ang. *separation of concerns*). Podstawą tej koncepcji jest podział, segregacja oraz wyodrębnianie fragmentów funkcjonalności[20]:

> Systemy podzielone na małe, dobrze nazwane i zrozumiałe fragmenty zapewniają szybszą pracę.

Celem separacji zagadnień jest lepsze zrozumienie złożoności i zarządzanie nią, a przez to projektowanie bardziej modularnego oprogramowania. Idea jest zapewne równie stara jak samo oprogramowanie i dlatego nadano jej już wiele różnych nazw. Na przykład w książce *Pragmatic Programmer* określono ją mianem **ortogonalności**[21]. Jej autorzy zalecają separowanie „ortogonalnych" aspektów oprogramowania. Z kolei Tom DeMarco tą samą ideę określa jako **spójność**[22]:

> Spójność jest miarą siły powiązań pomiędzy elementami wewnątrz modułu. Moduł o wysokiej spójności jest kolekcją instrukcji i danych, które należy traktować jako całość, gdyż ich wzajemne związki są tak silne. Każda próba ich rozdzielenia doprowadziłaby jedynie do zwiększenia stopnia powiązań i zmniejszenia czytelności.

W przypadku *SOLID*[23], jednego z najpopularniejszych zestawów zasad projektowych, ta sama idea jest nazywana zasadą jednej odpowiedzialności (ang. *single-responsibility principle*, w skrócie *SRP*):

> Dla danej klasy powinien istnieć tylko jeden powód wprowadzania w niej zmian[24].

---

[20] Michael Feathers, *Working Effectively with Legacy Code*, Addison-Wesley, 2013.

[21] David Thomas i Andrew Hunt, *Pragmatic Programmer: Your Journey to Mastery*, 20th Anniversary Edition, Addison-Wesley, 2019.

[22] Tom DeMarco, *Structured Analysis and System Specification*, Prentice Hall, 1979.

[23] SOLID to akronim utworzony z pięciu innych, połączonych akronimów — SRP, OCP, LSP, ISP oraz DIP — które zostały opisane w kilku dalszych wytycznych.

[24] Pierwszą książkę poświęconą wytycznym SOLID napisał Robert C. Martin, który zatytułował ją *Zwinne wytwarzanie oprogramowania. Najlepsze zasady, wzorce i praktyki*, w Polsce wydaną przez wydawnictwo Helion. Alternatywnie możesz sięgnąć po nowszą i zarazem znacznie tańszą książkę tego samego autora pt. *Czysta architektura. Struktura i design oprogramowania. Przewodnik dla profesjonalistów*, także wydaną przez Helion.

Choć sam pomysł jest stary i powszechnie znany pod wieloma nazwami, to jednak wiele prób wyjaśnienia, czym jest separacja zagadnień, rodzi więcej pytań, niż daje odpowiedzi. W szczególności dotyczy to zasady jednej odpowiedzialności. Już sama jej nazwa wywołuje wątpliwości: Czym bowiem jest odpowiedzialność? I czym jest *jedna* odpowiedzialność? Jedną z często spotykanych prób wyjaśnienia tych niejasności jest stwierdzenie:

> Wszystko powinno wykonywać tylko jedną czynność.

Niestety to wyjaśnienie trudno pobić pod względem niejasności. Choć sam termin **odpowiedzialność** nie ma zbyt wielu znaczeń, to fragment *tylko jedną czynność* wcale nie ułatwia zrozumienia, o jaką odpowiedzialność chodzi.

Jednak bez względu na nazwę idea jest zawsze ta sama: należy grupować tylko te rzeczy, które faktycznie są ze sobą powiązane, a separować te, których nic ze sobą nie łączy. Albo innymi słowy: należy oddzielać od siebie te rzeczy, które zmieniają się z różnych powodów. Dzięki temu można zredukować sztuczne powiązania pomiędzy różnymi aspektami kodu, jak również sprawić, że oprogramowanie będzie można łatwiej adaptować do zachodzących zmian. W najlepszym razie będziesz mógł zmienić wybrany aspekt swojego oprogramowania tylko w jednym miejscu.

Przykład sztucznych powiązań

Rzućmy nieco światła na zagadnienie separacji zagadnień w formie prostego przykładu. Przykład, który mogę przedstawić, jest naprawdę świetny. Oto abstrakcyjna klasa `Document`:

```
//#include <some_json_library.h> // Potencjana zależność fizyczna

class Document
{
 public:
   // ...
   virtual ~Document() = default;

   virtual void exportToJSON( /*...*/ ) const = 0;    ❶
   virtual void serialize( ByteStream&, /*...*/ ) const = 0;  ❷
   // ...
};
```

Można by sądzić, że to bardzo użyteczna klasa bazowa dla wszelkiego rodzaju dokumentów, nieprawdaż? W pierwszej kolejności deklaruje funkcję `exportToJSON()` ❶. Wszystkie jej klasy pochodne będą musiały implementować funkcję `exportToJSON()`, aby móc zapisać dokument w pliku w formacie JSON (*https://www.json.org/json-en.html*). To może się okazać bardzo wygodne: bez znajomości konkretnego typu dokumentu (a można sobie wyobrazić, że po pewnym czasie będziemy dysponowali dokumentami PDF, dokumentami programu Word i wieloma innymi) zawsze będziemy w stanie wyeksportować go w formacie JSON. Świetna sprawa! Druga z deklarowanych funkcji to `serialize()` ❷. Pozwala ona przekazać dokument w formie bajtów z wykorzystaniem strumienia `ByteStream`. Te bajty możemy następnie zapisać w jakimś systemie, który trwale je przechowa, na przykład w pliku lub bazie danych. No i oczywiście możemy przypuszczać, że istnieje wiele innych użytecznych funkcji, które pozwolą nam używać dokumentu w zasadzie do wszystkiego.

Jednak już widzę grymas na Twojej twarzy. Nie — wcale nie jesteś przekonany, że projekt przedstawionego kodu jest dobry. Być może wynika to z faktu, że podchodzisz do tego przykładu bardzo

podejrzliwie (on po prostu wygląda zbyt dobrze). Jednak być może już kiedyś boleśnie się przekonałeś, że taki projekt kodu może przysparzać problemów. Być może na podstawie własnych doświadczeń wiesz, że stosowanie zasad projektowania obiektowego do łączenia danych oraz funkcji, które na tych danych operują, może prowadzić do powstawania niefortunnych powiązań. W pełni się z tym zgadzam: choć taka klasa bazowa wygląda jak świetny, wszechstronny pakiet, a nawet sprawia wrażenie, jakby miała wszystko, czego w przyszłości możemy potrzebować, to jednak jej projekt już niebawem może nam przysporzyć problemów.

Przedstawiony kod jest przykładem złego projektu, gdyż zawiera wiele zależności. Rzecz jasna występują w nim oczywiste, bezpośrednie zależności, na przykład zależność od klasy `ByteStream`. Jednak projekt tej klasy daje także możliwość wprowadzenia sztucznych zależności, które utrudnią późniejsze wprowadzanie w niej zmian. W omawianym przykładzie istnieją trzy rodzaje takich sztucznych zależności. Dwie z nich wprowadza funkcja `exportToJSON()`, a trzecią funkcja `serialize()`.

Przede wszystkim funkcję `exportToJSON()` trzeba zaimplementować w klasach pochodnych. Oczywiście nie ma innej możliwości, gdyż mamy tu do czynienia z funkcją czysto wirtualną (*https://en.cppreference.com/w/cpp/language/abstract_class*, co jasno pokazuje sekwencja `= 0`). Ponieważ można sądzić, że klasy pochodne nie będą chciały samodzielnie zajmować się generowaniem danych w formacie JSON, zatem najprawdopodobniej będą w tym celu korzystać z jakichś innych bibliotek obsługujących ten format: *json* (*https://github.com/nlohmann/json*), *rapidjson* (*https://github.com/Tencent/rapidjson/*) czy też *simdjson* (*https://github.com/simdjson/simdjson*). Niezależnie do tego, której biblioteki zdecydujemy się użyć, nagle się okaże, że ze względu na funkcję składową `exportToJSON()` będą od niej zależeć także typy dokumentów dziedziczące po naszej abstrakcyjnej klasie bazowej. Co więcej, ze względu na chęć zapewnienia spójności od wybranej biblioteki najprawdopodobniej będą zależeć wszystkie klasy pochodne. Oznacza to, że klasy pochodne nie są wcale niezależne — są one w sztuczny sposób powiązane z konkretnymi decyzjami projektowymi[25]. Co więcej, zależność od konkretnej biblioteki JSON znacząco ograniczy możliwości wielokrotnego wykorzystania hierarchii klas, ponieważ nie będzie już można uznać jej za „lekką”. A decyzja o zastosowaniu innej biblioteki stanowiłaby bardzo poważną zmianę, gdyż wywołałaby konieczność modyfikacji wszystkich klas pochodnych[26].

Oczywiście druga funkcja, `serialize()`, także wprowadza sztuczną zależność dokładnie tego samego typu. Jest wysoce prawdopodobne, że także ona zostanie zaimplementowana z wykorzystaniem jakiejś zewnętrznej biblioteki, takiej jak *protobuf* (*https://github.com/protocolbuffers/protobuf*) lub *Boost.serialization* (*https://www.boost.org/doc/libs/1_78_0/libs/serialization/doc/index.html*). To znacząco pogarsza sytuację z zależnościami, gdyż wprowadza powiązanie pomiędzy dwoma ortogonalnymi, niepowiązanymi ze sobą aspektami projektowymi (czyli eksportem danych do formatu JSON oraz serializacją). Zmiana jednego z tych aspektów może wywoływać zmiany w drugim z nich.

[25] Nie zapominaj, że decyzje projektowe podejmowane podczas tworzenia takiej biblioteki zewnętrznej także mogą mieć wpływ na Twoje projekty, co w oczywisty sposób przyczyni się do powstania jeszcze silniejszych powiązań.

[26] W tym klas napisanych przez innych programistów, czyli klas, nad którymi nie będziemy mieć żadnej kontroli. Bez wątpienia ci inni programiści wcale nie będą z tego powodu szczęśliwi. Zatem taka zmiana może być *naprawdę* trudna.

W najgorszym razie funkcja `exportToJSON()` może wprowadzać także drugą zależność. Otóż przekazywane do niej argumenty mogą niezamierzenie odzwierciedlać pewne szczegóły implementacyjne wybranej biblioteki JSON. W takim przypadku decyzja o zmianie używanej biblioteki może powodować zmianę sygnatury funkcji `exportToJSON()`, co w efekcie wymusiłoby modyfikację kodu, który ją wywołuje. Oznacza to, że zupełnie niezamierzenie zależność od wybranej biblioteki JSON może mieć konsekwencje o znacznie większym zakresie.

Trzeci rodzaj zależności wprowadza druga z funkcji składowych naszej przykładowej klasy: `serialize()`. Funkcja ta sprawia, że klasy dziedziczące po klasie `Document` będą zależeć od naszych globalnych decyzji związanych ze sposobem serializacji dokumentów. Jakiego formatu użyjemy? Jaką kolejność zapisu bajtów zastosujemy? Czy konieczne będzie dodanie informacji o tym, że bajty reprezentują plik PDF lub dokument programu Word? A jeśli taka informacja będzie potrzebna (a zakładam, że to całkiem prawdopodobne), to jak będziemy reprezentować taki dokument? Przy użyciu liczby całkowitej? Na przykład moglibyśmy w tym celu użyć typu wyliczeniowego[27]:

```
enum class DocumentType
{
   pdf,
   word,
   // ... Zapewne znacznie więcej innych typów dokumentów
};
```

Takie rozwiązanie powszechnie się stosuje w przypadku serializacji. Jeśli jednak ta niskopoziomowa reprezentacja dokumentu będzie używana w implementacjach klas pochodnych klasy `Document`, to niezamierzenie powiążemy ze sobą wszystkie różne rodzaje dokumentów. Każda z klas pochodnych będzie niejawnie wiedzieć o wszystkich innych typach dokumentów. W efekcie dodanie nowego typu dokumentu będzie mieć bezpośredni wpływ na wszystkie pozostałe, już istniejące typy dokumentów. To byłby bardzo poważny błąd projektowy, gdyż znacznie utrudniałby przyszłe wprowadzanie zmian.

Niestety klasa `Document` stwarza okazję do wprowadzania wielu różnych rodzajów powiązań. A zatem nie można powiedzieć, że stanowi ona świetny przykład dobrego projektu, przede wszystkim dlatego, że nie można jej łatwo zmieniać. Wprost przeciwnie, utrudnia ona wprowadzanie zmian i z tego powodu jest znakomitym przykładem naruszenia zasady jednej odpowiedzialności: dziedziczące po niej klasy oraz klasy, które jej używają, mogą się zmieniać z wielu różnych powodów, gdyż klasa `Document` wprowadza silne powiązania pomiędzy kilkoma ortogonalnymi aspektami kodu. Podsumowując, klasy dziedziczące po `Document` oraz klasy, których kod będzie używał klasy `Document`, mogą się zmieniać z następujących powodów:

- Szczegóły implementacyjne funkcji `exportToJSON()` mogą się zmieniać ze względu na swoją bezpośrednią zależność od biblioteki zewnętrznej obsługującej format JSON.
- Sygnatura funkcji `exportToJSON()` może się zmieniać z powodu zmian szczegółów implementacyjnych biblioteki.
- Klasa `Document` oraz funkcja `serialize()` mogą się zmieniać ze względu na bezpośrednią zależność od klasy `ByteStream`.

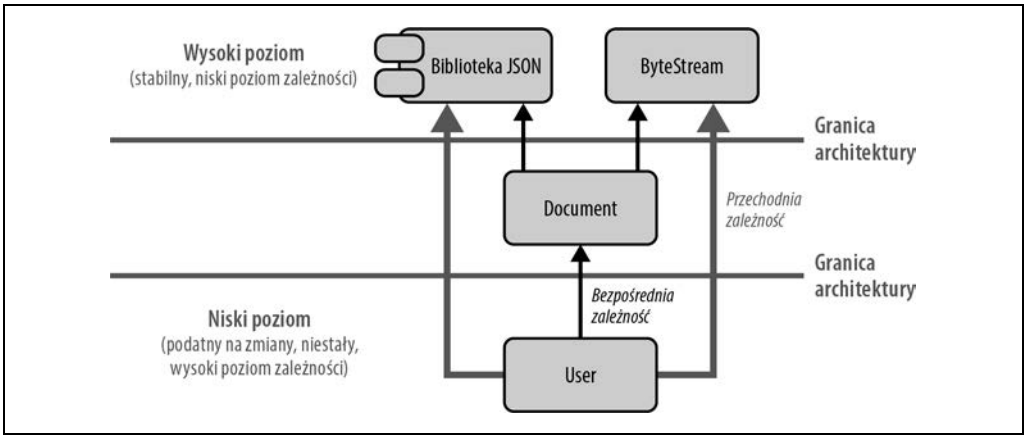
[27] Użycie typu wyliczeniowego wydaje się naturalnym rozwiązaniem, choć oczywiście istnieją także inne. W gruncie rzeczy potrzebujemy zaakceptowanego zbioru wartości reprezentujących różne formaty dokumentów zapisane w formie sekwencji bajtów.

- Szczegóły implementacyjne funkcji `serialize()` mogą się zmieniać ze względu na bezpośrednią zależność od szczegółów implementacyjnych klasy `ByteStream`.
- Wszystkie typy dokumentów mogą się zmieniać ze względu na bezpośrednią zależność od typu wyliczeniowego `DocumentType`.

Oczywiście ten projekt daje okazję do wprowadzenia większej liczby zmian, a każda z takich zmian byłaby coraz trudniejsza. I oczywiście w ogólnym przypadku istnieje niebezpieczeństwo wprowadzenia sztucznych powiązań pomiędzy ortogonalnymi aspektami kodu w klasach dokumentów, co jeszcze bardziej utrudniłoby wprowadzanie zmian. Co więcej, niektóre z tych zmian bez wątpienia nie będą się ograniczać tylko do jednego miejsca w kodzie. Zwłaszcza zmiany szczegółów implementacyjnych funkcji `exportToJSON()` oraz `serialize()` nie ograniczałyby się do jednej klasy, lecz najprawdopodobniej obejmowały wiele różnych typów dokumentów (PDF, Word itd.). Z tego względu zmiana obejmowałaby swym zasięgiem wiele miejsc w całej bazie kodu, co z kolei zwiększałoby ryzyko związane z jej utrzymaniem.

Powiązania logiczne oraz fizyczne

Powiązania nie ograniczają się jedynie do tych logicznych — występują także powiązania fizyczne. Ten rodzaj powiązań jest przedstawiony na rysunku 1.2. Załóżmy, że gdzieś na niskim poziomie naszej architektury istnieje klasa `User` korzystająca z dokumentów, których klasy znajdują się na wysokim poziomie architektury. Oczywiście klasa `User` bezpośrednio zależy od klasy `Document`. Jest to zależność konieczna — stanowi naturalny element danego problemu. Dlatego też nie powinna nas zajmować. Niemniej jednak (potencjalna) fizyczna zależność klasy `Document` od wybranej biblioteki JSON oraz bezpośrednia zależność od klasy `ByteStream` sprawiają, że klasa `User` staje się w bezpośredni i przechodni sposób zależna zarówno od biblioteki JSON, jak i klasy `ByteStream`; co więcej, zależności te występują na wysokim poziomie naszej architektury. W najgorszym razie oznacza to, że zmiany wprowadzone w bibliotece JSON lub w klasie `ByteStream` mogą mieć wpływ na klasę `User`. Na szczęście łatwo zauważyć, że jest to zależność sztuczna, a nie zamierzona: klasa `User` nie powinna w żaden sposób zależeć ani od biblioteki JSON, ani od serializacji.

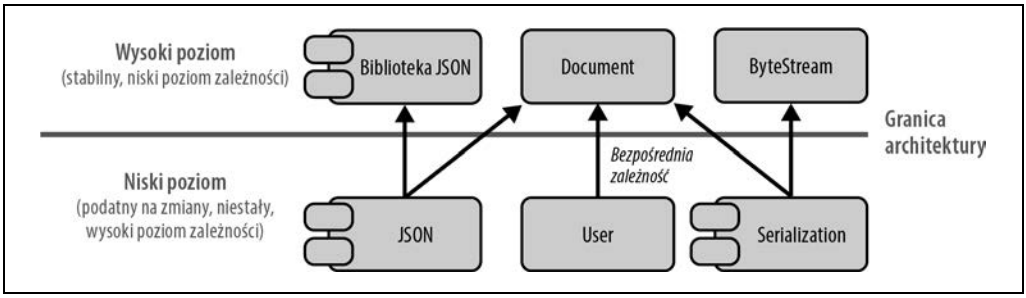


Rysunek 1.2. Silne, przechodnie, fizyczne powiązanie pomiędzy klasą User a ortogonalnymi aspektami kodu, takimi jak biblioteka JSON oraz serializacja

Powinienem jasno i wyraźnie zaznaczyć, że istnieje ryzyko wystąpienia *potencjalnej* fizycznej zależności pomiędzy klasą `Document` a biblioteką JSON. Jeśli plik nagłówkowy `<Document.h>` zawiera jakikolwiek nagłówek z wybranej biblioteki JSON (co oznaczyłem komentarzem umieszczonym na początku przykładu przedstawionego wcześniej, w punkcie pt. „Przykład sztucznych powiązań"), na przykład dlatego, że funkcja `exportToJSON()` oczekuje jakichś argumentów bazujących na tej bibliotece, to będzie to świadczyło o występowaniu wyraźnej zależności od tej biblioteki. Niemniej jednak, jeśli interfejs może się obyć bez tych szczegółów, a plik nagłówkowy `<Document.h>` nie musi zawierać niczego z wybranej biblioteki JSON, to może uda się uniknąć fizycznej zależności. A zatem wszystko zależy od tego, w jakim stopniu można wyodrębnić i odseparować abstrakcje.

Pewnie myślisz sobie z wyrzutem: „Wysoki poziom, niski poziom — gubię się w tym". Zdaję sobie sprawę, że te dwa terminy często przysparzają problemów. Zatem nim przejdziemy do dalszych rozważań, spróbujmy ustalić terminologię i wyjaśnić, co oznacza wysoki oraz niski poziom. Oba terminy pochodzą od sposobu, w jaki są rysowane diagramy UML (ang. *Unified Modeling Language*; *https://pl.wikipedia.org/wiki/Unified_Modeling_Language*): funkcjonalności, które uważamy za stabilne, umieszczamy na górze, na wysokim poziomie. Z drugiej strony funkcjonalności, które zmieniają się częściej i z tego powodu są traktowane jako mniej trwałe i bardziej podatne na zmiany, umieszcza się niżej — na niskim poziomie. Niestety w graficznej prezentacji często staramy się pokazać, w jaki sposób jedne rzeczy bazują na innych, więc te najbardziej stabilne są umieszczane na dole architektury. A to oczywiście powoduje zamieszanie i wątpliwości. Jednak abstrahując od graficznej reprezentacji, powinieneś zapamiętać te dwa określenia i ich znaczenie: *wysoki poziom* odnosi się do stabilnych elementów architektury, a *niski poziom* do tych aspektów, które zmieniają się częściej lub w których przypadku prawdopodobieństwo zmiany jest większe.

A teraz wróćmy do problemu: zasada jednej odpowiedzialności zaleca, by separować zagadnienia oraz rzeczy, które tak naprawdę nie są ze sobą powiązane — czyli które są niespójne. Innymi słowy, zaleca ona oddzielanie od siebie rzeczy, które zmieniają się z różnych powodów, jako **punktów zmienności**. Rysunek 1.3 przedstawia wzajemne powiązania w przypadku, gdy bibliotekę JSON oraz serializację wyodrębnimy jako oddzielne zagadnienia.



Rysunek 1.3. Zgodność z zasadą jednej odpowiedzialności eliminuje sztuczne powiązanie pomiędzy klasą User i sposobem serializacji a biblioteką JSON

Opierając się na tej zasadzie, możemy zmodyfikować klasę `Document` w następujący sposób:

```
class Document
{
 public:
   // ...
   virtual ~Document() = default;

   // Klasa nie zawiera już funkcji 'exportToJSON()' oraz 'serialization()'.
   // Zawiera jedynie bardzo proste operacje na dokumentach, które nie
   // powodują występowania silnych powiązań
   // ...
};
```

Eksport do formatu JSON ani serializacja to nie są już podstawowe elementy funkcjonalności klasy `Document`. Klasa ta powinna reprezentować jedynie bardzo proste operacje typowe dla różnych rodzajów dokumentów. Należy z niej wyodrębnić wszystkie aspekty ortogonalne. W ten sposób wprowadzanie zmian w bazie kodu stanie się znacznie łatwiejsze. Na przykład wyodrębnienie eksportu do formatu JSON do innego punktu zmienności oraz do nowego komponentu `JSON` albo zmiana obecnie używanej biblioteki JSON na inną będzie mieć wpływ wyłącznie na jeden komponent. Taką zmianę trzeba by wprowadzić tylko w jednym miejscu i byłaby ona odseparowana od innych ortogonalnych aspektów. Co więcej, taka zmiana ułatwiłaby także obsługę formatu JSON przy użyciu kilku różnych bibliotek. Oprócz tego wszelkie zmiany związane ze sposobem serializacji dokumentów trzeba by wprowadzać tylko w jednym komponencie: `Serialization`. Także ten komponent `Serialization` działałby jako punkt zmienności, umożliwiający łatwe wprowadzanie zmian. To byłoby optymalne rozwiązanie.

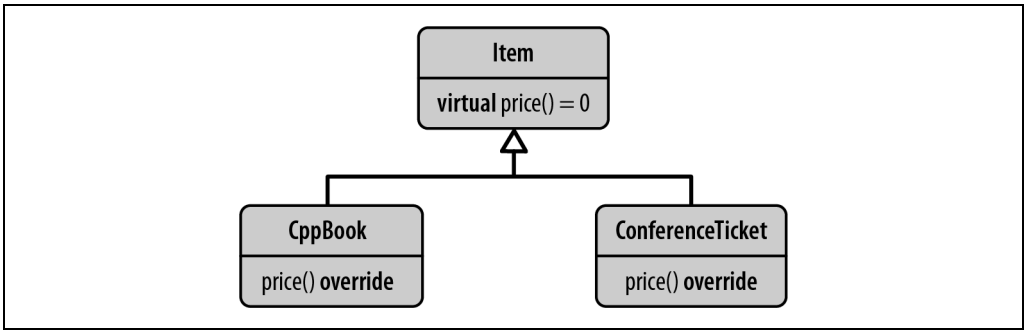
Już widzę, jak po początkowym rozczarowaniu klasą `Document` znowu wyglądasz na bardziej zadowolonego. Być może nawet uśmiechnąłeś się pod nosem, mówiąc „Wiedziałem!”. Jednak pomimo to nie jesteś jeszcze w pełni usatysfakcjonowany: „Owszem, zgadzam się z ogólną ideą separacji zagadnień. Ale w jaki sposób mam określać strukturę swojego oprogramowania, żeby oddzielać te zagadnienia? W jaki sposób mam sprawić, żeby ta idea zdała egzamin w praktyce?”. To świetne pytanie, lecz jednocześnie pytanie o wielu odpowiedziach, które opiszę w kilku następnych rozdziałach. Jednak pierwszym i najważniejszym zagadnieniem jest identyfikacja punktu zmienności, czyli jakiegoś aspektu kodu, który przypuszczalnie się zmieni. Te punkty zmienności należy wyodrębnić, wyizolować i opakować, tak by pozostały kod nie był już od nich w żaden sposób zależny. To w ostatecznym rozrachunku ułatwi wprowadzanie zmian.

Już słyszę, jak mówisz: „Ale to tylko powierzchowna rada”. I masz rację. Dlatego, że w tym przypadku nie można podać tylko jednej odpowiedzi ani odpowiedzi, która byłaby prosta. Wszystko bowiem — zależy. Jednak na pewno w kilku dalszych rozdziałach przedstawię wiele konkretnych odpowiedzi na pytanie, jak należy separować zagadnienia. W końcu niniejsza książka jest poświęcona projektowaniu oprogramowania, czyli zarządzaniu zależnościami. Oto drobna zapowiedź: w rozdziale 3. przedstawię ogólne i praktyczne podejście do rozwiązywania tego problemu — wzorce projektowe. Pamiętając o tej ogólnej koncepcji, pokażę, w jaki sposób można separować zagadnienia, wykorzystując do tego celu różne wzorce projektowe. Na szybko przychodzą mi do głowy następujące wzorce: **Odwiedzający**, **Strategia** i **Polimorfizm zewnętrzny**. Każdy z tych wzorców ma swoje mocne i słabe strony, jednak wszystkie mają tę cechę, że wprowadzają pewnego rodzaju abstrakcję, która ułatwia zmniejszenie stopnia zależności. Co więcej, obiecuję, że dokładniej pokażę, w jaki sposób implementować te wzorce projektowe w nowoczesnym C++.

Wzorzec **Odwiedzający** przedstawię w rozdziale 4., w podrozdziale pt. „Wytyczna 16.: Stosuj wzorzec Obserwator do rozszerzania operacji”, a wzorzec **Strategia** w rozdziale 5., w podrozdziale pt. „Wytyczna 19.: Stosuj wzorzec Strategia do określania sposobu wykonywania operacji”. Z kolei wzorzec projektowy **Polimorfizm zewnętrzny** opiszę w rozdziale 7., w podrozdziale pt. „Wytyczna 31.: Stosuj wzorzec Polimorfizm zewnętrzny, by tworzyć nieintruzyjny polimorfizm czasu wykonywania”.

Nie powtarzaj się

Zmienność ma także inny ważny aspekt. Aby go wyjaśnić, przedstawię kolejny przykład: hierarchię przedmiotów. Przykład takiej hierarchii jest zilustrowany na rysunku 1.4.



Rysunek 1.4. Hierarchia klasy Item

Na samej górze hierarchii znajduje się klasa bazowa `Item`:

```
//---- <Money.h> ----------------

class Money { /*...*/ };

Money operator*( Money money, double factor );
Money operator+( Money lhs, Money rhs );

//---- <Item.h> ----------------
#include <Money.h>
class Item
{
 public:
   virtual ~Item() = default;
   virtual Money price() const = 0;
};
```

Klasa bazowa `Item` reprezentuje abstrakcję wszelkich przedmiotów, które mogą mieć jakąś cenę (która z kolei jest reprezentowana przez klasę `Money`). Tę cenę można sprawdzić przy użyciu funkcji `price()`. Oczywiście może istnieć wiele różnych przedmiotów, choć my, aby to zademonstrować, ograniczymy się tylko do dwóch, klas `CppBook` oraz `ConferenceTicket`:

```
//---- <CppBook.h> ----------------

#include <Item.h>
#include <Money.h>
```

```
#include <string>

class CppBook : public Item
{
 public:
   explicit CppBook( std::string title, std::string author, Money price )  ❸
      : title_( std::move(title) )
      , author_( std::move(author) )
      , priceWithTax_( price * 1.15 ) // 15% podatku
   {}

   std::string const& title() const { return title_; }      ❹
   std::string const& author() const { return author_; }    ❺

   Money price() const override { return priceWithTax_; }   ❻

 private:
   std::string title_;
   std::string author_;
   Money priceWithTax_;
};
```

Konstruktor klasy `CppBook` oczekuje przekazania tytułu i autora, w formie łańcuchów znaków, jak również ceny, w formie obiektu klasy `Money` ❸[28]. Oprócz konstruktora przedstawiona klasa udostępnia jedynie funkcje `title()`, `author()` oraz `price()` ❹, ❺, ❻, pozwalające na pobieranie, odpowiednio, tytułu książki, jej autora oraz ceny. Z tym że funkcja `price()` jest nieco szczególna: oczywiście cena książki musi zawierać podatek. Dlatego też pierwotna cena książki musi zostać zmodyfikowana o określoną wartość podatku. W naszym przykładzie zakładam, że podatek ten wynosi 15%.

Drugą klasą pochodną klasy `Item` jest `ConferenceTicket`:

```
//---- <ConferenceTicket.h> ----------------
#include <Item.h>
#include <Money.h>
#include <string>

class ConferenceTicket : public Item
{
 public:
   explicit ConferenceTicket( std::string name, Money price )    ❼
      : name_( std::move(name) )
      , priceWithTax_( price * 1.15 ) // 15% podatku
   {}

   std::string const& name() const { return name_; }
```

[28] Możesz się zastanawiać nad powodem jawnego użycia słowa kluczowego `explicit` w tym konstruktorze. Powinieneś pamiętać o wytycznej podstawowej C.46 (*http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c46-by-default-declare-single-argument-constructors-explicit*), która zaleca domyślne stosowanie słowa kluczowego `explicit` w konstruktorach jednoargumentowych. To naprawdę dobra i godna polecenia sugestia, gdyż zapobiega niepożądanym i potencjalnie niebezpiecznym konwersjom. Można ją stosować także we wszelkich innych konstruktorach, choć wtedy jej przydatność nie będzie już aż tak duża; nie dotyczy to oczywiście konstruktorów kopiujących, które nie wykonują konwersji. W końcu stosowanie tej rady w niczym nie szkodzi.

```
    Money price() const override { return priceWithTax_; }

  private:
    std::string name_;
    Money priceWithTax_;
};
```

Klasa `ConferenceTicket` jest bardzo podobna do `CppBook`, jednak jej konstruktor oczekuje przekazania jedynie nazwy konferencji oraz ceny biletu ❼. Oczywiście dostęp do tych informacji jest możliwy za pośrednictwem funkcji, odpowiednio, `name()` oraz `price()`. Jednak znacznie ważniejsze jest to, że także cena biletu na konferencję o języku C++ podlega opodatkowaniu. Dlatego również tym razem modyfikujemy pierwotną cenę: mnożymy ją przez nasz przykładowy podatek wysokości 15%.

Dysponując już tymi trzema klasami, możemy przejść do napisania programu, w którego funkcji `main()` utworzymy kilka obiektów `Item`:

```
#include <CppBook.h>
#include <ConferenceTicket.h>
#include <algorithm>
#include <cstdlib>
#include <memory>
#include <vector>

int main()
{
   std::vector<std::unique_ptr<Item>> items{};

   items.emplace_back( std::make_unique<CppBook>("Język C++. Kompendium wiedzy", 89.40) );
   items.emplace_back( std::make_unique<CppBook>("Wprowadzenie do C++", 119.40) );
   items.emplace_back( std::make_unique<ConferenceTicket>("code::dive", 499.0) );
   items.emplace_back( std::make_unique<ConferenceTicket>("Polska Konferencja C++", 399.0)
);
   items.emplace_back( std::make_unique<ConferenceTicket>("Warsaw C++ Conference", 499.0) );

   Money const total_price =
      std::accumulate( begin(items), end(items), Money{},
         []( Money accu, auto const& item ){
            return accu + item->price();
         } );
   // ...

   return EXIT_SUCCESS;
}
```

W funkcji `main()` tworzymy kilka przedmiotów (a konkretnie: dwie książki i trzy konferencje), po czym obliczamy ich sumaryczną cenę. Ta sumaryczna cena będzie oczywiście uwzględniać 15% podatku.

Przedstawione rozwiązanie wydaje się dobrym projektem. Rozdzieliliśmy różne rodzaje przedmiotów i mamy możliwość określania sposobu obliczania ceny niezależnie dla każdego z tych rodzajów. Wydaje się, że postąpiliśmy zgodnie z zasadą jednej odpowiedzialności i wyodrębnili oraz odizolowali punkty zmienności. No i oczywiście przedmiotów jest więcej. O wiele więcej. A każdy z nich dba o to, by prawidłowo uwzględniać odpowiedni podatek. No i świetnie! Choć

taka hierarchia klasy `Item` może nas usatysfakcjonować na pewien czas, to jednak jej projekt ma pewną poważną wadę. Być może obecnie jeszcze nie potrafimy uświadomić sobie jej istnienia, lecz bezustannie gdzieś na horyzoncie można zauważyć jej cień, praprzyczynę wszystkich problemów programistycznych: zmianę.

Co się stanie, jeśli z jakiegoś powodu zmieni się stawka podatku? Co, jeśli podatek zostanie zmniejszony z 15% do 12%? Albo podniesiony do 16%? Wciąż mam w pamięci argumenty przedstawiane w dniu, gdy początkowy projekt był dodawany do bazy kodu: „To się nigdy nie zdarzy!". No cóż, nawet najbardziej nieprawdopodobne rzeczy kiedyś mogą się przydarzyć. Na przykład w 2021 roku w Niemczech stawkę podatkową zmniejszono z 19 do 16%. A to oczywiście oznaczałoby, że musimy zmienić stawkę podatkową w swojej bazie kodu. Gdzie należałoby wprowadzić taką zmianę? W obecnej sytuacji taka zmiana objęłaby swoim zasięgiem właściwie wszystkie klasy dziedziczące po klasie `Item`. A zatem zmiana wywarłaby wpływ na całą bazę kodu!

Chociaż zasada jednej odpowiedzialności zaleca separację punktów zmienności, to powinniśmy zadbać także o to, by nie powielać informacji w bazie kodu. Oprócz tego, że wszystko w bazie kodu powinno mieć tylko jedną odpowiedzialność (jeden powód do zmiany), każda z tych odpowiedzialności powinna istnieć wyłącznie w jednym miejscu systemu. Tę ideę często się określa jako zasadę *nie powtarzaj się*. Zaleca ona, by nie powtarzać tej samej kluczowej informacji w wielu miejscach, a zamiast tego zaprojektować system w taki sposób, żeby zmianę można było wprowadzić tylko w jednym miejscu kodu. W optymalnym przypadku stawka (lub stawki) podatku powinna być określona wyłącznie w jednym miejscu, co pozwoliłoby na jej łatwe modyfikowanie.

Zazwyczaj zasady jednej odpowiedzialności i „nie powtarzaj się" współgrają ze sobą bardzo dobrze. Zachowywanie zgodności z pierwszą z nich sprawia, że zachowujemy zgodność także z drugą i na odwrót. Jednak czasami zdarza się, że zachowanie zgodności z obydwiema będzie wymagać podjęcia dodatkowych kroków. Zdaję sobie sprawę z tego, że chciałbyś się już teraz dowiedzieć, co to za kroki, jednak na obecnym etapie wystarczy, że przedstawię ogólne koncepcje zasad jednej odpowiedzialności i „nie powtarzaj się". Obiecuję, że w dalszej części książki wrócę nich i pokażę, jak rozwiązywać problemy tego typu (patrz podrozdział pt. „Wytyczna 35.: Stosuj dekoratory, aby dodawać dostosowania hierarchicznie").

Unikaj zbyt wczesnego separowania zagadnień

Mam nadzieję, że udało mi się już Cię przekonać, jak świetnym i odpowiedzialnym pomysłem jest stosowanie się do zasad jednej odpowiedzialności oraz „nie powtarzaj się". Być może te pomysły spodobały Ci się tak bardzo, że postanowiłeś oddzielać od siebie wszystko — wszystkie klasy i funkcje — tworząc w ten sposób najmniejsze z możliwych jednostek funkcjonalności. Bo przecież właśnie o to chodzi, prawda? Jeśli teraz właśnie to chodzi Ci po głowie, to proszę: nie idź tą drogą! Weź głęboki oddech… i jeszcze jeden… a następnie wsłuchaj się w słowa mądrości Kateriny Trajczewskiej[29]:

> Nie staraj się dążyć do zasad SOLID, używaj ich, by zapewnić możliwości utrzymania kodu.

[29] Katerina Trajczewska: *Becoming a Better Developer by Using the SOLID Design Patterns* (*https://www.youtube.com/watch?v=rtmFCcjEgEw*), Laracon EU, 30 – 31 sierpnia 2018 r.

Obie zasady, jednej odpowiedzialności oraz „nie powtarzaj się", są naszymi narzędziami do poprawy możliwości utrzymania kodu oraz ułatwienia wprowadzania w nim zmian. Same w sobie nie są jednak naszym celem. Choć obie są niezwykle ważne w dłuższej perspektywie, to jednak przeprowadzanie separacji elementów bez dobrego zrozumienia tego, jakie rodzaje zmian mogą się pojawiać w przyszłości, może bardzo mocno zmniejszać produktywność. Projektowanie pod kątem zmian zazwyczaj faworyzuje jeden konkretny rodzaj zmiany, co zarazem może znacząco utrudniać wprowadzanie zmian innych rodzajów. Ta filozofia jest elementem zasady ogólnie określanej jako *nie będziesz tego potrzebował* (ang. *You Aren't Gonna Need It*, w skrócie *YAGNI*), która ostrzega przed uwzględnianiem w projekcie zbyt wielu szczegółów i funkcjonalności (patrz także podrozdział pt. „Wytyczna 5.: Projektuj pod kątem rozszerzania" w dalszej części tego rozdziału). Jeśli mamy przejrzysty plan, jeśli wiemy, jakiej zmiany należy się spodziewać, to powinniśmy skorzystać z zasad jednej odpowiedzialności i „nie powtarzaj się", by ułatwić wprowadzanie tej zmiany. Jeśli jednak nie mamy pojęcia, jakiego rodzaju zmiany należy oczekiwać, to nie próbujmy jej zgadywać — po prostu poczekajmy. Poczekajmy, aż zyskamy jasny obraz tego, jakiego rodzaju zmiany należy oczekiwać, i dopiero gdy to nastąpi, zmodyfikujmy kod w taki sposób, by jak najbardziej ułatwić jej wprowadzanie.

Nie można przy tym zapominać, że jednym z aspektów łatwego wprowadzania zmian jest przygotowanie testów jednostkowych, które pozwalają się dowiedzieć, czy wprowadzona zmiana nie wywołała problemów w działaniu oczekiwanych zachowań.

Podsumowując: występowanie zmian w *oprogramowaniu* jest czymś oczekiwanym i dlatego kluczowe znaczenie ma projektowanie oprogramowania pod kątem zmian. Należy separować zagadnienia i minimalizować powtórzenia, by zapewnić możliwość łatwego wprowadzania zmian bez obaw przed wywołaniem problemów w innych ortogonalnych aspektach kodu.

Wytyczna 2.: Projektuj pod kątem zmian

- Oczekuj, że w *oprogramowaniu* będą zachodzić zmiany.
- Projektuj pod kątem zmian i staraj się, by tworzone oprogramowanie zapewniało łatwość adaptacji.
- Unikaj łączenia niepowiązanych ze sobą, ortogonalnych aspektów kodu, by eliminować powiązania.
- Zapamiętaj: powiązania zwiększają prawdopodobieństwo, że zmiana utrudni wprowadzanie modyfikacji.
- Stosuj zasadę jednej odpowiedzialności (SRP), by separować zagadnienia.
- Stosuj zasadę „nie powtarzaj się" (DRY), by minimalizować powielanie.
- Unikaj przedwczesnego wyodrębniania, jeśli nie masz pewności, jakiej zmiany należy się spodziewać.

Wytyczna 3.: Separuj interfejsy w celu unikania sztucznych powiązań

Wróćmy do przykładu z klasą `Document` przedstawionego wcześniej w podrozdziale pt. „Wytyczna 2.: Projektuj pod kątem zmian”. Jak przypuszczam, obecnie czujesz, że widziałeś już dostatecznie dużo dokumentów, ale uwierz mi: jeszcze z nimi nie skończyliśmy. W klasie `Document` wciąż występują powiązania pewnego ważnego rodzaju, którymi musimy się zająć. Tym razem nie będziemy koncentrować się na żadnej konkretnej funkcji klasy `Document`, lecz na jej interfejsie jako całości:

```
class Document
{
 public:
   //...
   virtual ~Document() = default;

   virtual void exportToJSON( /*...*/ ) const = 0;
   virtual void serialize( ByteStream& bs, /*...*/ ) const = 0;
   //...
};
```

Segregacja interfejsów w celu separacji zagadnień

Klasa `Document` wymaga, by zarówno eksport dokumentów do formatu JSON, jak i ich serializacja były implementowane w klasach pochodnych. Choć z punktu widzenia dokumentu takie rozwiązanie można uznać za rozsądne (w końcu *wszystkie* dokumenty powinny zapewniać możliwość eksportu do formatu JSON oraz serializacji), to jednak przyczynia się ono do wystąpienia powiązań innego rodzaju. Wyobraźmy sobie następujący kod:

```
void exportDocument( Document const& doc )
{
   //...
   doc.exportToJSON( /* Niezbędne argumenty */ );
   //...
}
```

Funkcja `exportDocument()` zajmuje się jedynie wyeksportowaniem danego dokumentu do formatu JSON. Innymi słowy, *nie* obchodzi jej ani serializacja dokumentu, ani jakikolwiek inny aspekt dokumentów. Jednak pomimo to, ze względu na postać definicji interfejsu klasy `Document` oraz powiązań pomiędzy wieloma ortogonalnymi aspektami, funkcja `exportDocument()` będzie zależna nie tylko od sposobu eksportu do formatu JSON, lecz także wielu innych czynników. Wszystkie te zależności są sztuczne i niepotrzebne. Zmiana którejkolwiek z nich — na przykład klasy `ByteStream` lub sygnatury funkcji `serialize()` — będzie mieć wpływ na *wszystkich* użytkowników klasy `Document`, i to nawet takich, którzy wcale nie potrzebują serializacji. Dowolna zmiana sprawi, że konieczne będzie ponowne skompilowanie, przetestowanie, a w najgorszym razie także wdrożenie *wszystkich* użytkowników klasy `Document` (na przykład jeśli są oni udostępniani w ramach odrębnej biblioteki). Dokładnie to samo się stanie, gdy klasa `Document` zostanie rozszerzona o nową funkcję, na przykład eksportującą dane do dokumentu innego typu. Ten problem staje się tym większy, im więcej ortogonalnych funkcjonalności zostanie połączonych

w klasie `Document`: każda zmiana niesie ze sobą ryzyko wywołania efektu lawinowego w całej bazie kodu. Co jest bardzo przykre, gdyż interfejsy powinny pomagać w separacji, a nie wprowadzać dodatkowe powiązania.

Przyczyną tego powiązania jest naruszenie zasady segregacji interfejsów (która odpowiada literze *I* w akronimie *SOLID*):

> Nie należy wymagać, by klienty były zależne od metod, których nie używają[30].

Zasada segregacji interfejsów zaleca separowanie zagadnień przez segregację (rozdzielanie) interfejsów. W kontekście naszego przykładu oznaczałoby to utworzenie dwóch osobnych interfejsów reprezentujących dwa ortogonalne aspekty, eksport do formatu JSON oraz serializację:

```
class JSONExportable
{
 public:
   // ...
   virtual ~JSONExportable() = default;

   virtual void exportToJSON( /*...*/ ) const = 0;
   // ...
};

class Serializable
{
 public:
   // ...
   virtual ~Serializable() = default;

   virtual void serialize( ByteStream& bs, /*...*/ ) const = 0;
   // ...
};

class Document
   : public JSONExportable
   , public Serializable
{
 public:
   // ...
};
```

Ta separacja nie sprawia, że klasa `Document` staje się przestarzała i niepotrzebna. Wprost przeciwnie, klasa ta wciąż reprezentuje wymagania narzucane na wszystkie dokumenty. Jednak wprowadzona separacja zagadnień pozwala nam teraz na zminimalizowanie zależności wyłącznie do zbioru faktycznie niezbędnych funkcji:

```
void exportDocument( JSONExportable const& exportable )
{
   // ...
   exportable.exportToJSON( /* Niezbędne argumenty */ );
   // ...
}
```

[30] Robert C. Martin, *Zwinne wytwarzanie oprogramowania. Najlepsze zasady, wzorce i praktyki*, Helion.

Funkcja `exportDocument()` w przedstawionej postaci, zależna jedynie od interfejsu `JSONExportable`, nie jest już zależna od funkcjonalności związanej z serializacją ani od klasy `ByteStream`. Jak widać, segregacja interfejsów pomogła nam ograniczyć powiązania.

Być może chciałbyś zapytać: „Ale czy to nie jest jedynie separacja zagadnień?". „Czy to nie jest kolejny przykład zastosowania zasady jednej odpowiedzialności?". Faktycznie tak jest. Zgadzam się z tym, że w zasadzie zidentyfikowaliśmy dwa ortogonalne aspekty i je rozdzielili, innymi słowy zastosowaliśmy do interfejsu klasy `Document` zasadę jednej odpowiedzialności. Można by zatem powiedzieć, że zasady segregacji interfejsów oraz jednej odpowiedzialności są tym samym. Albo przynajmniej że zasada segregacji interfejsów jest szczególnym przypadkiem zasady jednej odpowiedzialności, gdyż ta pierwsza koncentruje się na interfejsach. Taka opinia wydaje się powszechna w społeczności i ja się z nią zgadzam. Niemniej jednak i tak uważam, że warto mówić o zasadzie segregacji interfejsów. Niezależnie od tego, że stanowi ona jedynie przypadek szczególny, zdecydowanie sądzę, że jest to przypadek bardzo ważny. Niestety bardzo często ulegamy pokusie łączenia w interfejsach wielu niepowiązanych ze sobą, ortogonalnych aspektów. Taki błąd sztuki — połączenie w jednym interfejsie kilku odrębnych aspektów — mógł się zdarzyć nawet *Tobie*. Oczywiście nigdy nie ośmieliłbym się sugerować, że zrobiłeś to celowo, ale być może nieświadomie lub przez przypadek… Bardzo często nie zwracamy uwagi na takie szczegóły. Rzecz jasna możesz się spierać i twierdzić, że nigdy byś czegoś podobnego nie zrobił. Jednak być może przykład przedstawiony w rozdziale 5., w podrozdziale pt. „Wytyczna 19.: Stosuj wzorzec Strategia do określania sposobu wykonywania operacji", przekona Cię, jak łatwo taki błąd popełnić. A ponieważ późniejsza zmiana interfejsu może być wyjątkowo trudna, uważam, że opłaca się uczulać i zwracać uwagę na problem z interfejsami. Właśnie z tego powodu nie pominąłem zasady segregacji interfejsów, lecz wspomniałem o niej jako o ważnym i godnym odnotowania przypadku szczególnym zasady jednej odpowiedzialności.

Minimalizacja wymagań określanych przez argumenty szablonów

Choć może się wydawać, że zasadę segregacji interfejsów można stosować wyłącznie do klas bazowych, i stosuje się ją głównie z wykorzystaniem technik programowania obiektowego, to jednak ogólną ideę minimalizacji zależności wprowadzanych przez interfejsy można także odnieść do szablonów. W ramach przykładu rozważny funkcję `std::copy()`:

```
template< typename InputIt, typename OutputIt >
OutputIt copy( InputIt first, InputIt last, OutputIt d_first );
```

W języku C++20 moglibyśmy wyrazić te wymagania przy użyciu konceptów (ang. *concepts*):

```
template< std::input_iterator InputIt, std::output_iterator OutputIt >
OutputIt copy( InputIt first, InputIt last, OutputIt d_first );
```

Funkcja `std::copy()` oczekuje przekazania pary iteratorów wejściowych, określających zakres do skopiowania, oraz iteratora wyjściowego, określającego zakres docelowy. Jawnie wymaga ona podania iteratorów wejściowych oraz iteratora wyjściowego, gdyż nie potrzebuje żadnych innych operacji. W ten sposób minimalizuje ona wymagania co do przekazywanych argumentów.

A teraz załóżmy, że funkcja `std::copy()` zamiast iteratorów `std::input_iterator` oraz `std::output_iterator` wymaga iteratorów `std::forward_iterator`:

```
template< std::forward_iterator ForwardIt, std::forward_iterator ForwardIt >
OutputIt copy( ForwardIt first, ForwardIt last, ForwardIt d_first );
```

Takie rozwiązanie niestety ograniczyłoby przydatność algorytmu `std::copy()`. Nie bylibyśmy już w stanie kopiować elementów ze strumieni wejściowych, gdyż zazwyczaj nie gwarantują możliwości wielokrotnego przejścia ani nie pozwalają na zapis. To byłoby bardzo niefortunne. Jednak z punktu widzenia zależności funkcja `std::copy()` zostałaby w takim przypadku uzależniona od operacji i wymagań, których wcale nie potrzebuje. A przekazywane do niej iteratory musiałyby udostępniać dodatkowe operacje, stąd też funkcja `std::copy()` wymusiłaby na nich dodatkowe zależności.

To czysto hipotetyczny przykład, lecz pokazuje on jak ważne w przypadku interfejsów jest separowanie zagadnień. Oczywiście rozwiązaniem powyższego problemu jest uzmysłowienie sobie, że możliwości związane z odczytem oraz z zapisem są odrębnymi aspektami. A zatem, po rozdzieleniu zagadnień i po zastosowaniu zasady segregacji interfejsów, zależności zostały znacząco zmniejszone.

Wytyczna 3.: Separuj interfejsy w celu unikania sztucznych powiązań

- Musisz mieć świadomość, że powiązania dotyczą także interfejsów.
- Stosuj zasadę segregacji interfejsów (ISP) by w interfejsach separować zagadnienia.
- Traktuj zasadę segregacji interfejsów jako szczególny przypadek zasady jednej odpowiedzialności (SRP).
- Zapamiętaj, że zasadę segregacji interfejsów z powodzeniem można stosować zarówno podczas określania hierarchii dziedziczenia, jak i w szablonach.

Wytyczna 4.: Projektuj pod kątem łatwości testowania

Jak już napisałem w podrozdziale pt. „Wytyczna 1.: Znaczenie projektu oprogramowania”, *oprogramowanie* się zmienia. Jego zmiany są czymś oczekiwanym. Jednak za każdym razem, kiedy zmieniamy coś w oprogramowaniu, ryzykujemy, że coś w nim uszkodzimy. Oczywiście niecelowo — niechcący, i to nawet pomimo wszelkich wysiłków. To ryzyko zawsze istnieje. Jednak, jako doświadczonemu programiście, nie spędza Ci to snu z oczu. Niech i będzie takie ryzyko — nie ma sprawy. Masz bowiem coś, co zabezpiecza Cię przed przypadkowymi problemami w kodzie, coś, co minimalizuje ryzyko: testy.

Testy przygotowuje się i stosuje po to, by móc zyskać pewność, że pomimo wprowadzenia zmian oprogramowanie pod względem funkcjonalnym wciąż działa poprawnie. A zatem testy są Twoją warstwą zabezpieczającą — Twoją kamizelką ratunkową. Testy mają kluczowe znaczenie! Jednak przede wszystkim trzeba zacząć od ich napisania. A żebyś mógł napisać testy i przygotować swoją kamizelkę ratunkową, Twoje oprogramowanie musi zapewniać taką możliwość: musisz je tworzyć w taki sposób, by można było pisać do niego testy, a najlepiej, żeby tworzenie i dodawanie testów było *łatwe*. Co prowadzi nas do meritum tej wytycznej: oprogramowanie należy projektować pod kątem łatwości testowania.

Jak testować prywatną funkcję składową?

Już słyszę, jak mówisz: „Ależ oczywiście, że mam testy. Każdy powinien ich używać. To przecież powszechnie wiadomo". W pełni się z tym zgadzam. I naprawdę wierzę, że Twoja baza kodu zawiera sensowny zestaw testów[31]. Jednak zaskakujące jest to, że pomimo powszechnej zgody co do konieczności stosowania testów nie wszyscy autorzy oprogramowania biorą to sobie do serca[32]. W rzeczywistości spora część kodu jest bardzo trudna do przetestowania. A czasami wynika to z faktu, że nie napisano go tak, by umożliwiał testowanie.

Aby Ci pokazać, o co chodzi, mam dla Ciebie wyzwanie. Przyjrzyj się poniższej klasie `Widget`. Przechowuje ona kolekcję obiektów `Blob`, którą co jakiś czas trzeba aktualizować. W tym celu klasa `Widget` udostępnia funkcję składową `updateCollection()`, której znaczenie właśnie zauważyliśmy, by następnie dojść do wniosku, że należy ją przetestować. I to jest właśnie to wyzwanie: W jaki sposób przetestowałbyś funkcję składową `updateCollection()`?

```
class Widget
{
   // ...

 private:
   void updateCollection( /* Argumenty konieczne do aktualizacji kolekcji */ );
   std::vector<Blob> blobs_;
   /* Prawdopodobnie inne dane składowe */
};
```

Przypuszczam, że od razu zauważyłeś, w czym tkwi największy problem: funkcja składowa `updateCollection()` została zadeklarowana w prywatnej części klasy. To oznacza, że nie ma do niej bezpośredniego dostępu z zewnątrz, a co za tym idzie — także sposobu, by ją bezpośrednio przetestować. Zastanów się nad tym przez chwilę…

Stwierdzisz pewnie: „No fakt, funkcja jest prywatna, ale to i tak nie jest żadne szczególne wyzwanie. Istnieje wiele sposobów, żeby ją przetestować". Oczywiście, to prawda: można spróbować to zrobić na wiele różnych sposobów. A zatem kontynuuj, proszę… Przemyślisz zapewne możliwości i podasz pierwsze rozwiązanie, które przyjdzie Ci do głowy: „Najprostszym sposobem przetestowania tej funkcji byłoby więc zastosowanie jakiejś innej publicznej funkcji składowej, w której kodzie umieścimy wywołanie funkcji `updateCollection()`". Jak na pierwszy pomysł, brzmi to całkiem interesująco. Załóżmy, że kolekcję trzeba aktualizować po dodaniu nowego obiektu `Blob`. Wywołanie funkcji składowej `addBlob()` powoduje wykonanie funkcji `updateCollection()`:

---

[31] Jeśli jeszcze nie przygotowałeś takiego zestawu testów, to bierz się do roboty. Mówię poważnie. Spójnym źródłem informacji o zestawach testów jest wystąpienie Bena Saksa na temat testów jednostkowych pt. *Back to Basics: Unit Tests* (*https://www.youtube.com/watch?v=_OHE33s7EKw*) na konferencji CppCon 2020. Drugim doskonałym źródłem informacji, które warto przeczytać i przemyśleć, jest cały temat o testowaniu i programowaniu opartych na testach, w szczególności jego opracowanie zawarte w książce Jeffa Langra pt. *Modern C{plus}{plus} Programming with Test-Driven Development*, wydanej przez wydawnictwo O'Reilly.

[32] Wiem, że pisanie tu o tej „powszechnej zgodzie" jest niestety nieco oderwane od rzeczywistości. Jeśli potrzebujesz dowodu na to, że jeszcze nie wszystkie projekty uwzględniają znaczenie i konieczność stosowania testów i że wciąż nie wszyscy programiści są ich świadomi, to zerknij na to zgłoszenie: *https://bugs.openfoam.org/view.php?id=406*, pochodzące z listy problemów OpenFOAM.

```
class Widget
{
 public:
   // ...
   void addBlob( Blob const& blob, /*...*/ )
   {
      // ...
      updateCollection( /*...*/ );
      // ...
   }

 private:
   void updateCollection( /* Argumenty konieczne do aktualizacji kolekcji */ );
   std::vector<Blob> blobs_;
   /* Prawdopodobnie inne dane składowe */
};
```

Choć takie rozwiązanie wygląda rozsądnie, to jednak jest jednocześnie czymś, czego w miarę możliwości należy unikać. Sugerowane rozwiązanie określa się jako testy strukturalne albo testy białej skrzynki (ang. *white box test*). Testy tego rodzaju znają wewnętrzne szczegóły implementacyjne i korzystają z tej wiedzy. Jednak rozwiązanie to wprowadza zależność kodu testu od szczegółów implementacyjnych produkcyjnego kodu. Problem związany z tym sposobem testowania polega na tym, że oprogramowanie się zmienia. Kod się zmienia. Zmieniają się jego szczegóły implementacyjne. Na przykład w przyszłości funkcja `addBlob()` może zostać przepisana w taki sposób, że już nie będzie wymagać aktualizowania kolekcji. Jeśli to się stanie, to taki test nie będzie już robił tego, do czego jest przeznaczony. Straciłbyś zatem swój test funkcji `updateCollection()`, a co gorsza, zapewne nawet byś o tym nie wiedział. Oznacza to, że testy strukturalne stwarzają zagrożenie. Tak samo jak w swoim produkcyjnym kodzie powinieneś unikać zależności i je redukować (patrz podrozdział pt. „Wytyczna 1.: Znaczenie projektu oprogramowania"), tak powinieneś unikać zależności pomiędzy testami a szczegółami kodu produkcyjnego.

Tym, czego w rzeczywistości potrzebujemy, jest **test funkcjonalny** (nazywany także **testem czarnej skrzynki**, ang. *black box test*). Testy funkcjonalne nie czynią żadnych założeń dotyczących wewnętrznych szczegółów implementacyjnych i testują jedynie oczekiwane zachowania. Oczywiście, jeśli zmienimy coś w klasie, to także w takich testach mogą wystąpić problemy, jednak nie powinno to nastąpić po zmianie jakichś szczegółów implementacyjnych, lecz jedynie po zmianie oczekiwanego zachowania testowanego kodu.

„No dobrze, rozumiem", powiesz. „Ale nie sugerujesz chyba, żeby zmienić funkcję `updateCollection()` na publiczną, prawda?" Nie, zapewniam, że niczego takiego nie sugeruję. Choć z drugiej strony czasami może to być całkiem sensownym rozwiązaniem. Jednak w naszym przypadku wątpię, by to był dobry pomysł. Funkcja `updateCollection()` nie powinna być bowiem wywoływana dla zabawy. Można ją wywoływać wyłącznie z ważnego powodu, tylko w określonym czasie i zapewne w celu zachowania jakiegoś niezmiennika. Jest ona czymś, czego nie powinniśmy oddawać w ręce użytkownika. A zatem nie — nie uważam, by ta funkcja była dobrym kandydatem do umieszczenia w sekcji `public`.

„Spoko… tylko się upewniałem. Zatem zadeklarujemy klasę testu jako klasę zaprzyjaźnioną klasy `Widget`. W ten sposób będzie ona miała pełny dostęp do klasy `Widget` i będzie mogła wywoływać jej prywatne funkcje składowe”:

```
class Widget
{
   // ...
 private:
   friend class TestWidget;

   void updateCollection( /* Argumenty konieczne do aktualizacji kolekcji */ );
   std::vector<Blob> blobs_;
   /* Prawdopodobnie inne dane składowe */
};
```

Owszem, moglibyśmy dodać klasę zaprzyjaźnioną. Załóżmy, że klasa testowa o nazwie `TestWidget` zawiera wszystkie testy klasy `Widget`. Moglibyśmy ją zadeklarować jako klasę zaprzyjaźnioną klasy `Widget`. Choć to może się wydawać sensownym rozwiązaniem, to obawiam się, że ponownie będę musiał Cię rozczarować. Owszem, z technicznego punktu widzenia to rozwiązywałoby problem, jednak z punktu widzenia projektu takie rozwiązanie oznaczałoby wprowadzenie sztucznej zależności. Aktywne dodanie do kodu produkcyjnego deklaracji klasy zaprzyjaźnionej sprawia, że ten kod produkcyjny będzie wiedział o klasie testowej. A choć jest oczywiste, że kod testu powinien wiedzieć o kodzie produkcyjnym (w końcu takie jest jego przeznaczenie), to kod produkcyjny wcale nie powinien mieć jakichkolwiek informacji o kodzie testowym. Wprowadziłoby to bowiem zależność cykliczną, która jest bardzo niekorzystną i sztuczną zależnością.

„To brzmi tak, jakby to była najgorsza rzecz na świecie. Czy to naprawdę takie straszne?” No cóż, czasami to może być całkiem rozsądne rozwiązanie. Bez wątpienia jest to rozwiązanie proste i szybkie. Jednak teraz, kiedy mamy czas, by rozważyć wszystkie dostępne opcje, bez wątpienia znajdzie się coś, co będzie lepsze od dodawania klasy zaprzyjaźnionej.

Nie chcę pogarszać sytuacji, ale w C++ nie będziemy mieć zbyt wielu przyjaciół. No tak, wiem, że to brzmi smutnie i powiało samotnością, ale oczywiście mam na myśli słowo kluczowe `friend`[33]: tak naprawdę w C++ klasy zaprzyjaźnione wcale nie są naszymi przyjaciółmi. Wynika to stąd, że użycie słowa kluczowego `friend` wprowadza powiązanie, i to w większości przypadków powiązanie sztuczne, a takich powiązań należy unikać. Oczywiście istnieją także wyjątki — przykłady dobrego wykorzystania słowa kluczowego `friend` — jak przyjaciele, bez których nie możemy się obejść. Ich przykładem mogą być „przyjaciele ukryci” (*https://www.justsoftwaresolutions.co.uk/cplusplus/hidden-friends.html*), czy też idiomatyczne zastosowania słowa kluczowego `friend`, takie jak *idiom Pesskey* (*https://arne-mertz.de/2016/10/passkey-idiom/*). Jednak klasę testów można by raczej porównać do przyjaciółki z mediów społecznościowych, zatem deklarowanie jej jako zaprzyjaźnionej nie wydaje się dobrym pomysłem.

[33] Ang. *friend* oznacza przyjaciela, znajomego — *przyp. tłum.*

Być może zatem zasugerujesz: „No dobrze, zmieńmy więc funkcję `updateCollections()` z prywatnej na chronioną i stwórzmy klasę testów, która będzie dziedziczyć po klasie `Widget`. W ten sposób test miałby do niej pełny dostęp”:

```
class Widget
{
   // ...
 protected:
   void updateCollection( /* Argumenty konieczne do aktualizacji kolekcji */ );

   std::vector<Blob> blobs_;
   /* Prawdopodobnie inne dane składowe */
};

class TestWidget : private Widget
{
   // ...
};
```

No cóż, muszę przyznać, że z technicznego punktu widzenia takie rozwiązanie by zadziałało. Jednak sam fakt, że zasugerowałeś rozwiązanie oparte na dziedziczeniu, wyraźnie mi mówi, że musimy bardzo poważnie porozmawiać o znaczeniu dziedziczenia oraz o tym, jak go prawidłowo używać. Wystarczy zacytować dwóch pragmatycznych programistów[34]:

> Dziedziczenie rzadko kiedy jest odpowiedzią.

Ponieważ tym zagadnieniem zajmiemy się stosunkowo późno, chciałbym tylko zaznaczyć, że takie rozwiązanie sprawia wrażenie, jakbyśmy nadużywali dziedziczenia tylko po to, by uzyskać dostęp do niepublicznej funkcji składowej. Jestem niemal pewny, że dziedziczenia nie wymyślono w tym celu. Zastosowanie dziedziczenia w celu uzyskania dostępu do chronionej sekcji klasy byłoby jak użycie dynamitu do otworzenia drzwi. W końcu byłoby to niemal równoznaczne z zadeklarowaniem wszystkich funkcji jako publicznych — każdy miałby do nich łatwy dostęp. Wygląda zatem na to, że faktycznie nie zaprojektowaliśmy klasy `Widget` pod kątem łatwości testowania.

„No dajże spokój, a co innego możemy zrobić? Może chcesz, żebym uciekł się do wykorzystania preprocesora i zdefiniował wszystkie etykiety `private` jako `public`?”:

```
#define private public

class Widget
{
   // ...

 private:
   void updateCollection( /* Argumenty konieczne do aktualizacji kolekcji */ );

   std::vector<Blob> blobs_;
   /* Prawdopodobnie inne dane składowe */
};
```

[34] David Thomas oraz Andrew Hunt, *Pragmatyczny programista. Od czeladnika do mistrza*, Helion.

No dobrze, chyba pora wziąć głęboki oddech. Choć to ostatnie rozwiązanie wydaje się zabawne, chyba masz świadomość, że w tym momencie przekroczyliśmy granice zdrowego rozsądku[35]. Jeśli na poważnie rozważamy zastosowanie preprocesora, by zapewnić sobie dostęp do prywatnej sekcji klasy `Widget`, to wszystko stracone.

Prawdziwe rozwiązanie: Separacja zagadnień

„No dobrze, a zatem co *powinienem* zrobić, żeby przetestować prywatną funkcję składową? Już wcześniej odrzuciłeś wszystkie opcje". Nie, nie odrzuciłem wszystkich opcji. Nie odrzuciłem jednej opcji związanej z projektem, o której wspominałem wcześniej w podrozdziale pt. „Wytyczna 2.: Projektuj pod kątem zmian". Chodzi o separację zagadnień. Chciałem zasugerować rozwiązanie polegające na wyodrębnieniu z klasy prywatnej funkcji składowej i przekształceniu jej w osobny element bazy kodu. Moim preferowanym rozwiązaniem w tym przypadku będzie wyodrębnienie funkcji składowej i przekształcenie jej w **funkcję zewnętrzną** (albo po prostu **funkcję**, ang. *free function*):

```
void updateCollection( std::vector<Blob>& blobs, /* Argumenty konieczne do aktualizacji kolekcji */ );

class Widget
{
   // ...

 private:
   std::vector<Blob> blobs_;
   /* Prawdopodobnie inne dane składowe */
};
```

Wszystkie wcześniejsze wywołania funkcji składowej można zastąpić wywołaniem wolnej funkcji `updateCollection()`, będzie to wymagać jedynie dodania do nich `blobs_` jako pierwszego argumentu. Ewentualnie, jeśli funkcja używała stanu, możemy go wyodrębnić w formie osobnej klasy. W ten sposób możemy zaprojektować wynikowy kod, który będzie łatwy, a być może nawet trywialny, do testowania:

```
namespace widgetDetails {

class BlobCollection
{
 public:
   void updateCollection( /* Argumenty konieczne do aktualizacji kolekcji */ );

 private:
   std::vector<Blob> blobs_;
};

} // Koniec przestrzeni nazw widgetDetails

class Widget
{
   // ...
```

[35] A może nawet wkroczyliśmy do przerażającego królestwa zachowań niezdefiniowanych.

```
private:
   widgetDetails::BlobCollection blobs_;
   /* Prawdopodobnie inne dane składowe */
};
```

„No chyba sobie żartujesz!", zapewne wykrzykniesz. „Czy to nie jest najgorsze ze wszystkich możliwych rozwiązań? Czy nie dzielimy sztucznie rzeczy, które powinny być zgrupowane? I w końcu — czy zasada jednej odpowiedzialności nie zaleca, by rzeczy, które są powiązane, trzymać blisko siebie?" No cóż, nie sądzę. A nawet — wprost przeciwnie, jestem głęboko przekonany, że dopiero teraz zachowujemy zgodność z zasadą jednej odpowiedzialności: zaleca ona, by separować rzeczy, które nie są ze sobą związane — rzeczy, które zmieniają się z różnych powodów. Na pierwszy rzut oka może się wydawać, że klasa `Widget` i funkcja `updateCollection()` są ze sobą związane, gdyż dana `blob_` co jakiś czas musi być aktualizowana. Jednak to, że funkcji `updateCollection()` nie można w prosty sposób przetestować, jest wyraźnym sygnałem, że projekt tej klasy nie jest dobry: jeśli nie można przetestować czegoś, co wymaga jawnego testowania, to najwyraźniej coś jest nie tak. Dlaczego mamy utrudniać sobie życie i funkcję, którą trzeba przetestować, ukrywać w prywatnej sekcji klasy `Widget`? Ponieważ testowanie odgrywa tak ważną rolę w obliczu zachodzących zmian, stanowi ono jeszcze jeden czynnik, który może nam pomagać w określaniu, jakie elementy kodu należy ze sobą łączyć. Skoro funkcja `updateCollection()` jest na tyle ważna, że chcemy ją osobno testować, to zapewne będzie się ona zmieniać z powodów innych niż sama klasa `Widget`. A to świadczy o tym, że funkcja `updateCollection()` oraz klasa `Widget` nie powinny tworzyć jednej całości. A zatem, zgodnie z zasadą jednej odpowiedzialności, funkcję tę należy wyodrębnić z klasy `Widget`.

Być może chciałbyś zapytać: „Ale czy to nie jest wbrew idei hermetyzacji? I niech Ci nawet nie przychodzi do głowy zapominać o hermetyzacji. Uważam, że hermetyzacja jest bardzo ważna!". W pełni się z Tobą zgadzam, że hermetyzacja jest bardzo ważna! Jednak hermetyzacja jest jeszcze jednym powodem do separowania zagadnień. Zgodnie z opinią Scotta Meyersa zawartą w jego książce *C++. 50 efektywnych sposobów na udoskonalenie Twoich programów* wyodrębnianie funkcji z klas jest krokiem w kierunku zwiększania hermetyzacji. Według Meyersa generalnie należy przedkładać funkcje, które nie są ani składowymi, ani funkcjami zaprzyjaźnionymi, nad funkcje składowe[36]. Wynika to z faktu, że każda funkcja składowa ma pełny dostęp do wszystkich składowych klasy, nawet tych prywatnych. Jednak po wyodrębnieniu funkcja `updateCollection()` będzie miała dostęp wyłącznie do publicznego interfejsu klasy `Widget` i nie będzie mogła używać składowych prywatnych. Oznacza to, że składowe te będą nieco bardziej hermetyzowane. Zwróć uwagę, że dokładnie ten sam argument odnosi się także do wyodrębnienia klasy `BlobCollection`: klasa ta nie ma dostępu do niepublicznych składowych klasy `Widget`, co też nieco zwiększa stopień hermetyzacji klasy `Widget`.

Zastosowanie separacji zagadnień i wyodrębnienie tego fragmentu funkcjonalności zapewnia nam kilka korzyści. Przede wszystkim, jak już wspominałem, poprawi się stopień hermetyzacji klasy `Widget`. Poza tym przetestowanie wyodrębnionej funkcji `updateCollection()` będzie bardzo proste, a może nawet trywialne. Nie będziemy nawet do tego potrzebować obiektu `Widget` — wystarczy

[36] Przekonujący argument na poparcie tej opinii można znaleźć w podrozdziale „Sposób 23" w książce *C++. 50 efektywnych sposobów na udoskonalenie Twoich programów* Scotta Meyersa.

przekazać daną typu `std::vector<Blob>` jako pierwszy argument wywołania funkcji (oczywiście nie chodzi tu o pierwszy niejawny argument przekazywany w wywołaniach wszystkich funkcji składowych, czyli wskaźnik `this`) lub wywołać publiczną funkcję składową. Co więcej, nie musimy zmieniać żadnego innego aspektu klasy `Widget`: wystarczy, że zawsze, gdy będziemy musieli zaktualizować kolekcję, przekażemy składową `blobs_` do funkcji `updateCollection()`. Nie musimy dodawać żadnej nowej publicznej funkcji pobierającej. A co najważniejsze: będziemy mogli modyfikować funkcję `updateCollection()` w izolacji, bez konieczności modyfikowania samej klasy `Widget`. A to wyraźny sygnał, że udało się nam zredukować zależności. Choć początkowo funkcja `updateCollection()` była ściśle związana z klasą `Widget` (tak, chodzi o wskaźnik `this`), to teraz znacznie rozluźniliśmy to powiązanie. Obecnie funkcja ta jest odrębną usługą, której nawet będziemy mogli wielokrotnie używać w innym kodzie.

Widzę, że wciąż masz pytania. Może po tym, co właśnie przeczytałeś, masz wrażenie, że w ogóle nie powinieneś już tworzyć i używać funkcji składowych. Dla jasności: wcale nie sugeruję, że powinieneś wyodrębniać z klas wszystkie możliwe funkcje składowe. Sugeruję jedynie, żebyś nieco dokładniej przyjrzał się funkcjom, które trzeba testować, a które są umieszczone w prywatnej sekcji klasy. Co więcej, być może się zastanawiasz, jak to rozwiązanie działa w przypadku funkcji wirtualnych. No cóż, na to pytanie nie ma szybkiej odpowiedzi, jednak z tym zagadnieniem będziemy się mierzyć na wiele różnych sposobów w dalszej części książki. Moimi podstawowymi celami zawsze będą zmniejszanie powiązań i zwiększanie możliwości testowania, nawet kosztem separacji funkcji wirtualnych.

Podsumowując: nie powinieneś pogarszać projektu i ograniczać możliwości testowania przez wprowadzanie sztucznych powiązań i barier. Projektuj pod kątem testowania. Separuj zagadnienia. Uwalniaj swoje funkcje!

Wytyczna 4.: Projektuj pod kątem łatwości testowania

- Zapamiętaj, że testy są Twoją warstwą ochronną, zabezpieczającą Cię przed przypadkowym uszkodzeniem kodu.
- Pamiętaj, że testy mają kluczowe znaczenie, podobnie jak łatwość testowania.
- Separuj zagadnienia, starając się poprawić możliwości testowania.
- Zastanów się nad przeniesieniem w inne miejsce prywatnych funkcji składowych, które muszą być testowane.
- Przedkładaj funkcje, które nie są ani składowymi, ani funkcjami zaprzyjaźnionymi, nad funkcje składowe.

Wytyczna 5.: Projektuj pod kątem rozszerzania

Istnieje pewien ważny aspekt związany ze zmieniającym się oprogramowaniem, którego jeszcze nie poruszyłem: rozszerzalność. Rozszerzalność powinna być jednym z głównych celów tworzonych projektów. Gdyż szczerze mówiąc, jeśli stracisz możliwość dodawania do kodu nowych funkcjonalności, będzie to oznaczało koniec czasu życia tego kodu. A zatem dodawanie nowych

funkcjonalności — rozszerzanie bazy kodu — ma dla nas kluczowe znaczenie. Z tego powodu rozszerzalność naprawdę powinna być jednym z naszych podstawowych celów i jednym głównych celów, z myślą o których będziemy tworzyć projekty.

Zasada otwarte-zamknięte

Projektowanie pod kątem rozszerzania niestety nie przychodzi bez wysiłku ani nie pojawia się samo, magicznie, w naszych projektach. Bynajmniej. Tę możliwość rozszerzania będziemy musieli jawnie uwzględnić, tworząc projekt oprogramowania. W podrozdziale pt. „Wytyczna 2.: Projektuj pod kątem zmian", przedstawiłem bardzo proste podejście do serializowania dokumentów. Konkretnie rzecz biorąc, przedstawiłem tam klasę `Document` deklarującą czysto wirtualną funkcję `serialize()`:

```
class Document
{
 public:
   // ...
   virtual ~Document() = default;

   virtual void serialize( ByteStream& bs, /*...*/ ) const = 0;
   // ...
};
```

Ponieważ `serialize()` jest funkcją czysto wirtualną, trzeba ją zaimplementować w każdej klasie pochodnej klasy `Document`, w tym w klasie `PDF`:

```
class PDF : public Document
{
 public:
   // ...

   void serialize( ByteStream& bs, /*...*/ ) const override;
   // ...
};
```

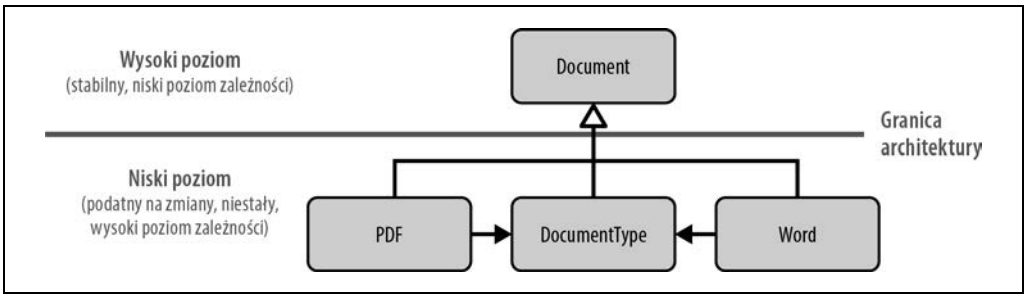
Jak dotąd wszystko jest w porządku. Rodzi się jednak interesujące pytanie: W jaki sposób możemy zaimplementować funkcję składową `serialize()`? Jednym z wymagań jest to, byśmy później mieli możliwość skonwertowania serializowanych danych ponownie do postaci obiektu klasy `PDF` (będziemy chcieli deserializować bajty do postaci obiektu `PDF`). Z tego względu konieczne jest zapisanie informacji o tym, co te bajty reprezentują. W przykładzie przedstawionym w podrozdziale pt. „Wytyczna 2.: Projektuj pod kątem zmian" użyliśmy do tego celu typu wyliczeniowego:

```
enum class DocumentType
{
   pdf,
   word,
   // ... Zapewne znacznie więcej innych typów dokumentów
};
```

Tego typu wyliczeniowego możemy teraz używać we wszystkich klasach pochodnych, by przekazać informację o typie dokumentu na samym początku strumienia bajtów. Dzięki temu podczas deserializacji będziemy mogli w prosty sposób określić, jakiego rodzaju dokument został zapisany. Niestety okazuje się, że takie rozwiązanie jest kiepską decyzją projektową. Przez wprowadzenie

takiego typu wyliczeniowego spowodowaliśmy niepożądane powiązanie wszystkich typów dokumentów: klasa `PDF` będzie teraz wiedzieć o istnieniu formatu programu Word. I rzecz jasna klasa `Word` będzie wiedzieć o istnieniu formatu PDF. Oczywiście masz rację: klasy te nie znają wzajemnie swoich szczegółów implementacyjnych, niemniej jednak coś o sobie wiedzą.

To powiązanie jest przedstawione na rysunku 1.5. Z punktu widzenia architektury typ wyliczeniowy `DocumentType` jest umieszczony na tym samym poziomie co klasy `PDF` i `Word`. Oba te typy dokumentów używają typu wyliczeniowego `DocumentType` (a co za tym idzie, są zależne).



Rysunek 1.5. Sztuczne powiązanie różnych typów dokumentów przez użycie typu wyliczeniowego DocumentType

Problem związany z takim rozwiązaniem staje się oczywisty, kiedy próbujemy rozszerzyć funkcjonalność. Oprócz dokumentów PDF i Word teraz chcemy obsługiwać prosty format XML. W idealnym przypadku powinno wystarczyć dodanie klasy `XML`, dziedziczącej po klasie bazowej `Document`. Jednak okazuje się, że dodatkowo musimy także zmodyfikować typ wyliczeniowy `DocumentType`:

```
enum class DocumentType
{
   pdf,
   word,
   xml,   // Nowy typ dokumentów
   // ... Zapewne znacznie więcej innych typów dokumentów
};
```

Ta zmiana spowoduje przynajmniej konieczność ponownej kompilacji klas wszystkich innych typów dokumentów (PDF, Word itd.). Być może teraz wzruszysz ramionami i pomyślisz: „Wielkie mi co! Wystarczy wszystko ponownie skompilować”. Cóż… zwróć uwagę, że użyłem słowa „przynajmniej”. W gorszym przypadku taki projekt znacząco ograniczy innym możliwości rozszerzania kodu, na przykład możliwość dodawania innych typów dokumentów, gdyż nie wszyscy będą mogli modyfikować typ wyliczeniowy `DocumentType`. Zdecydowanie takie powiązanie nie wydaje się czymś dobrym: klasy `PDF` i `Word` nie powinny dysponować jakimikolwiek informacjami o nowej klasie `XML`. Nie powinny niczego ani widzieć, ani czuć — choćby najmniejszej groźby ponownej kompilacji.

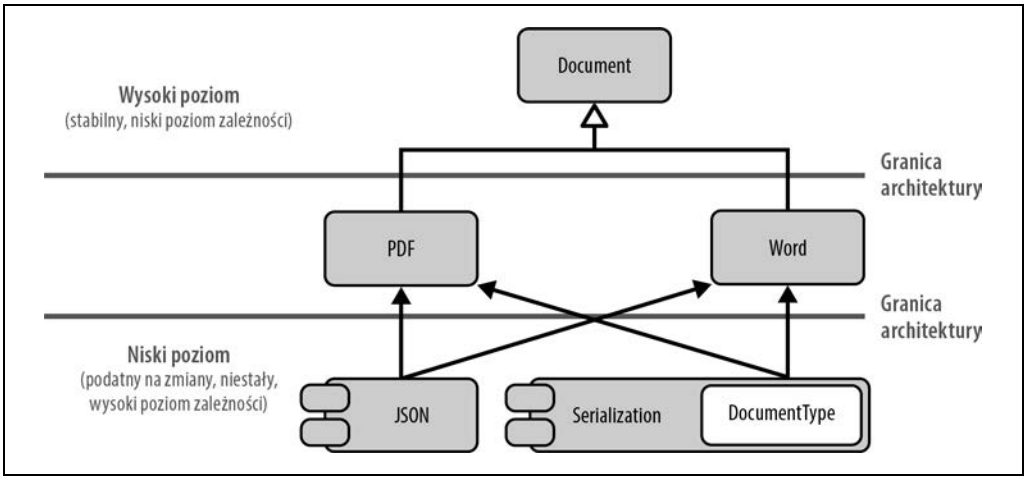
Problem występujący w tym przykładzie można przedstawić jako naruszenie zasady otwarte-zamknięte. Jest to druga z zasad zaliczanych do zestawu SOLID. Zaleca ona, by projektować oprogramowanie w sposób, który zapewni łatwość wprowadzania niezbędnych rozszerzeń[37].

> Artefakty programowe (klasy, moduły, funkcje itd.) powinny być otwarte na rozszerzanie, lecz zamknięte na modyfikację.

Zasada otwarte-zamknięte stwierdza, że oprogramowanie powinno zapewniać możliwość rozszerzania (powinno ono być otwarte na rozszerzanie). Jednak to rozszerzanie powinno być proste, a w najlepszym przypadku — możliwe przez samo dodawanie nowego kodu. Innymi słowy, rozszerzanie nie powinno wymagać modyfikowania już istniejącego kodu (oprogramowanie powinno być zamknięte na zmiany).

Teoretycznie rozszerzanie powinno być proste: w naszym przypadku wszystko powinno ograniczyć się do dodania nowej klasy `XML`. Ta nowa klasa nie wymagałaby wprowadzania jakichkolwiek modyfikacji w żadnym innym miejscu kodu. Niestety funkcja `serialize()` w sztuczny sposób wiąże ze sobą różne rodzaje dokumentów i zmusza nas do modyfikowania typu wyliczeniowego `DocumentType`. Z kolei ta modyfikacja wywiera wpływ na pozostałe typy dokumentów dziedziczące po klasie `Document`. O unikanie właśnie takich zależności chodzi w zasadzie otwarte-zamknięte.

Na szczęście już pokazałem, jak rozwiązać ten problem w przypadku naszej klasy `Document`. Właściwym sposobem jest zastosowanie separacji zagadnień (patrz rysunek 1.6).



Rysunek 1.6. Separacja zagadnień eliminuje naruszenie zasady otwarte-zamknięte

Dzięki separacji zagadnień można przez grupowanie tych rzeczy, które naprawdę są ze sobą związane, wyeliminować przypadkowe powiązania pomiędzy różnymi rodzajami dokumentów. Cały kod związany z serializacją będzie teraz prawidłowo umieszczony w komponencie `Serialization`, który logicznie może być umieszczony na innym poziomie architektury. Ten komponent `Serialization` będzie zależeć od wszystkich typów dokumentów (PDF, Word, XML itd.), jednak żaden z typów

[37] Bertrand Meyer, *Oprogramowanie zorientowane obiektowo*, Helion, 2005.

dokumentów nie będzie zależny od tego komponentu. Co więcej, żaden z typów dokumentów nie będzie nic wiedział o innych typach (czyli dokładnie tak, jak powinno być).

„Chwileczkę!", stwierdzisz. „Przecież w kodzie obsługującym serializację wciąż będziemy potrzebowali typu wyliczeniowego, prawda? W jaki inny sposób moglibyśmy przechowywać informacje o tym, co reprezentują zapisane bajty?" Cieszę się, że zwróciłeś na to uwagę. Owszem, w komponencie `Serialization` wciąż (najprawdopodobniej) będziemy potrzebowali czegoś takiego jak typ wyliczeniowy `DocumentType`. Jednak dzięki zastosowaniu separacji zagadnień udało nam się rozwiązać problem zależności. Obecnie żaden z różnych typów dokumentów nie zależy już od typu wyliczeniowego `DocumentType`. Teraz wszystkie strzałki zależności biegną z poziomu niskiego (od komponentu `Serialization`) do poziomu wyższego (`PDF` i `Word`). A ta cecha ma kluczowe znaczenie dla poprawnej, dobrej architektury.

„No a co z dodawaniem nowego typu dokumentu? Czy to nie będzie wymagać modyfikacji komponentu `Serialization`?" Oczywiście masz rację. Niemniej jednak nie będzie to naruszeniem zasady otwarte-zamknięte, która stwierdza, że nie powinna występować konieczność modyfikowania istniejącego kodu umieszczonego na tym samym lub wyższym poziomie architektury. Niemniej jednak nie można ani uniknąć, ani zapobiec modyfikacjom na niższych poziomach. Komponent `Serialiazation` *musi* zależeć od wszystkich typów dokumentów i dlatego *musi* być modyfikowany po dodaniu każdego nowego typu dokumentu. Z tego powodu komponent `Serialization` musi być umieszczony na niższym poziomie architektury (*uzależnionym* od tych wyższych).

Zgodnie z informacjami, które podałem w podrozdziale pt. „Wytyczna 2.: Projektuj pod kątem zmian", rozwiązaniem problemu przedstawionego w tym przykładzie jest separacja zagadnień. Wydaje się zatem, że rzeczywistym rozwiązaniem jest zachowanie zgodności z zasadą jednej odpowiedzialności. Właśnie z tego powodu pojawiają się głosy krytyczne stwierdzające, że zasada otwarte-zamknięte nie jest osobną zasadą projektową, lecz w rzeczywistości jest tym samym co zasada jednej odpowiedzialności. Muszę przyznać, że rozumiem to podejście. Bardzo często zdarza się, że separacja zagadnień jest w stanie zapewnić nam pożądaną rozszerzalność. Wiele razy zobaczysz to w dalszej części książki, a w szczególności w rozważaniach dotyczących wzorców projektowych. Dlatego opinia, że zasady jednej odpowiedzialności oraz otwarte-zamknięte są ze sobą powiązane, albo nawet że są tym samym, są uzasadnione.

Z drugiej strony w tym przykładzie przekonałeś się, że istnieją pewne specyficzne architektoniczne czynniki związane z zasadą otwarte-zamknięte, których nie brałem pod uwagę podczas prezentowania zasady jednej odpowiedzialności. Co więcej, jak się samemu przekonasz w rozdziale 4., w podrozdziale pt. „Wytyczna 15.: Projektuj pod kątem dodawania typów i operacji", dość często będziemy musieli podejmować jawne decyzje dotyczące tego, co będziemy rozszerzać oraz jak będziemy chcieli to robić. Te decyzje mogą mieć znaczący wpływ na sposób stosowania zasady jednej odpowiedzialności oraz projektowania oprogramowania. Dlatego zasada otwarte-zamknięte w większym stopniu niż zasada jednej odpowiedzialności zdaje się dotyczyć świadomego rozszerzania oprogramowania i podejmowania dotyczących go decyzji. Dlatego wydaje się czymś więcej niż jedynie dodatkiem do zasady *jednej odpowiedzialności*. Choć może być i tak, że to zależy[38].

[38] Odpowiedź „To zależy!" bez wątpienia usatysfakcjonuje nawet najgłośniejszych krytyków zasady otwarte-zamknięte.

W każdym razie przedstawiony przykład bezspornie pokazuje, że rozszerzalność koniecznie należy uwzględniać na etapie projektowania oprogramowania oraz że chęć rozszerzania oprogramowania w określony sposób jest doskonałym wskaźnikiem konieczności stosowania separacji zagadnień. Ważne jest, by zrozumieć, w jaki sposób oprogramowanie będzie rozszerzane, zlokalizować **punkty dostosowywania** i przygotować projekt tak, żeby rozszerzanie oprogramowania w wybrany sposób było łatwe.

Rozszerzalność podczas kompilacji

Przykład klasy `Document` sprawia wrażenie, jakby wszystkie czynniki projektowe dotyczyły polimorfizmu czasu wykonywania (ang. *runtime polymorphism*). Absolutnie nie: te same czynniki oraz te same argumenty odnoszą się także do problemów występujących podczas kompilacji. Aby to pokazać, przedstawię teraz kilka przykładów z Biblioteki standardowej. Oczywiście niezwykle istotne jest to, by ta biblioteka zapewniała możliwości rozszerzania. Oczywiście przede wszystkim mamy jej *używać*, jednak zachęca się nas także, by się na niej opierać i dodawać do niej własne elementy funkcjonalności. Z tego względu Bibliotekę standardową zaprojektowano pod kątem rozszerzania. Jednak co ciekawe, nie są do tego celu używane klasy bazowe, ale raczej przeciążanie funkcji, szablony oraz specjalizacja szablonów (klas).

Świetnym przykładem rozszerzania przez zastosowanie przeciążania funkcji jest algorytm `std::swap()`. Od momentu udostępnienia języka C++11 `std::swap()` definiuje się w następujący sposób:

```
namespace std {

template< typename T >
void swap( T& a, T& b )
{
   T tmp( std::move(a) );
   a = std::move(b);
   b = std::move(tmp);
}

} // namespace std
```

Ze względu na fakt, że algorytm `std::swap()` jest definiowany jako szablon funkcji, można go używać dla dowolnych typów: typów podstawowych, takich jak `int` lub `double`, typów z Biblioteki standardowej, takich jak `std::string`, jak również dla własnych typów danych. Niemniej jednak mogą się także pojawiać typy, które będą wymagały szczególnej uwagi, oraz typy, które nie mogą lub nie powinny być zamieniane przy użyciu `std::swap()` (na przykład dlatego, że nie można ich w efektywny sposób przenosić), choć z powodzeniem można je zamieniać na inne sposoby. Oczekuje się jednak, że dane typów wartościowych można zamieniać, co jawnie wyrażono w wytycznej podstawowej C.83 (*http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c83-for-value-like-types-consider-providing-a-noexcept-swap-function*)[39].

[39] C++ Core Guidelines (*https://isocpp.github.io/CppCoreGuidelines/*) to rozwijany przez społeczność projekt, w ramach którego gromadzi się i uzasadnia zestaw wskazówek i wytycznych dotyczących pisania dobrego kodu C++. Najlepiej reprezentują one powszechne poczucie tego, czym jest idiomatyczny język C++. Te wytyczne można znaleźć w serwisie GitHub, na stronie *https://isocpp.github.io/CppCoreGuidelines/*.

Dla typów przypominających wartościowe warto rozważyć udostępnienie funkcji zamieniającej, która nie będzie zgłaszać wyjątków (`noexcept`).

Funkcję `swap()` możemy przesłonić, dostosowując ją do własnego typu danych:

```
namespace custom {

class CustomType
{
   /* Implementacja wymagająca specyficznej wersji funkcji swap() */
};

void swap( CustomType& a, CustomType& b )
{
   /* Specjalna implementacja zamiany dwóch instancji typu CustomType */
}

} // Koniec przestrzeni nazw custom
```

Jeśli funkcja `swap()` jest używana prawidłowo, to powyższa funkcja niestandardowa wykonuje specjalny rodzaj operacji zamiany dwóch instancji typu `CustomType`[40]:

```
template< typename T >
void some_function( T& value )
{
   // ...
   T tmp( /*...*/ );

   using std::swap;     // Umożliwiamy kompilatorowi rozważenie użycia std::swap
                        // w następnym wywołaniu
   swap( tmp, value ); // Zamienia dwie wartości; dzięki niekwalifikowanemu wywołaniu
                        // oraz dzięki ADL, gdyby 'T' był typem 'CustomType', spowodowałoby to
                        // wywołanie 'custom::swap()'
   // ...
}
```

Oczywiście algorytm `std::swap()` zaprojektowano jako *punkt dostosowywania*, pozwalający na obsługę własnych typów i zachowań. To samo zresztą dotyczy wszystkich algorytmów w Bibliotece standardowej. W ramach przykładu przyjrzyjmy się dwóm innym, `std::find()` oraz `std::find_if()`:

```
template< typename InputIt, typename T >
constexpr InputIt find( InputIt first, InputIt last, T const& value );

template< typename InputIt, typename UnaryPredicate >
constexpr InputIt find_if( InputIt first, InputIt last, UnaryPredicate p );
```

Dzięki parametrom szablonów, a pośrednio także dzięki konceptom, funkcje `std::find()` oraz `std::find_if()` (jak również wszelkie inne algorytmy) niejawnie pozwalają nam przeprowadzać wyszukiwania przy użyciu własnych typów (iteratorów). Oprócz tego algorytm `std::find_if()` pozwala na określanie sposobu porównywania elementów. Jak widać, obie te funkcje wyraźnie zaprojektowano pod kątem rozszerzania i dostosowywania.

[40] ADL to skrótowe określenie terminu *Argument Dependant Lookup* (wyszukiwanie zależne od argumentów). Informacje na ten temat można znaleźć w witrynie cppreference.com (*https://en.cppreference.com/w/cpp/language/adl*) bądź też w moim wystąpieniu na konferencji CppCon 2020 (*https://www.youtube.com/watch?v=GydNMuyQzWo*).

Ostatnim rodzajem *punktów dostosowywania* jest specjalizacja szablonów. To rozwiązanie jest stosowane na przykład przez szablon klasy `std::hash`. Korzystając ponownie z typu `CustomType`, którego użyliśmy wcześniej w przykładzie funkcji `std::swap()`, szablon `std::hash` moglibyśmy jawnie wyspecjalizować w następujący sposób:

```
template<>
struct std::hash<CustomType>
{
   std::size_t operator()( CustomType const& v ) const noexcept
   {
      return /*...*/;
   }
};
```

Projekt `std::hash` pozwala dostosować jego działanie do dowolnego typu. Jeszcze bardziej godne uwagi jest to, że nie wymaga to modyfikowania żadnego istniejącego kodu; w celu zaadaptowania szablonu do specjalnych wymagań wystarczy podać jego osobną specjalizację.

Niemal całą Bibliotekę standardową C++ zaprojektowano pod kątem rozszerzania i dostosowywania. Raczej jednak nie będzie to żadnym zaskoczeniem, gdyż powinna ona znajdować się na najwyższym poziomie naszej architektury. Oznacza to, że Biblioteka standardowa nie może być zależna od żadnego naszego kodu — to nasz kod w całości musi zależeć od niej.

Unikanie przedwczesnego projektowania pod kątem rozszerzania

Biblioteka standardowa C++ jest kapitalnym przykładem projektowania pod kątem rozszerzania. Miejmy nadzieję, że ułatwi Ci ona zrozumienie, jak ważne jest zapewnienie możliwości rozszerzania pisanego kodu. Niemniej jednak, choć możliwości rozszerzania są bardzo ważne, to jednak dążenie do ich zapewnienia nie oznacza wcale, że powinniśmy automatycznie i bezrefleksyjnie umieszczać w klasach bazowych lub szablonach wszystkie szczegóły implementacyjne, tylko po to, by zagwarantować późniejsze możliwości rozszerzania. Tak samo jak nie powinniśmy przedwcześnie separować zagadnień, nie powinniśmy także przedwcześnie projektować pod kątem rozszerzania. Oczywiście, jeśli mamy dobre wyobrażenie tego, jak nasz kod będzie się zmieniał, to możemy wykorzystać tę wiedzę podczas projektowania. Pamiętaj jednak o zasadzie „nie będziesz tego potrzebował”: jeśli nie wiesz, jak kod będzie ewoluował, to być może mądrzejszym rozwiązaniem będzie poczekać, zamiast wymyślać rozszerzenie, które nigdy nie będzie potrzebne. Być może następne rozszerzenie, które się pojawi, pozwoli określić, jakie inne rozszerzenia będą potrzebne w przyszłości, co z kolei zapewni sposobność do przeprowadzenia refaktoryzacji kodu, dzięki której dodawanie kolejnych rozszerzeń będzie łatwe. W przeciwnym razie ryzykujesz, że skoncentrowanie się na jednym rodzaju rozszerzeń znacząco utrudni dodawanie innych (więcej informacji na ten temat znajdziesz w rozdziale 4., w podrozdziale pt. „Wytyczna 15.: Projektuj pod kątem dodawania typów i operacji”). To sytuacja, której powinieneś unikać, o ile tylko będzie to możliwe.

Podsumowując: projektowanie pod kątem rozszerzania jest ważną częścią projektowania pod kątem zmian. Dlatego uważnie wypatruj fragmentów funkcjonalności, co do których można oczekiwać, że będą rozszerzane, i projektuj kod w taki sposób, by dodawanie tych rozszerzeń było proste.

Wytyczna 5.: Projektuj pod kątem rozszerzania

- Preferuj projekt, który będzie zapewniał łatwość rozszerzania kodu.
- Staraj się projektować zgodnie z zasadą otwarte-zamknięte, aby pisany kod był otwarty na rozszerzanie i zamknięty na modyfikację.
- Projektuj pod kątem dodawania nowego kodu, stosując w tym celu klasy bazowe, szablony, przeciążanie funkcji oraz specjalizowanie szablonów.
- Jeśli nie masz pewności co do tego, jaki będzie następny dodatek do kodu, unikaj przedwczesnego określania abstrakcji.

Bez dobrego projektu nie będzie dobrej aplikacji!

Większość dobrych książek o C++ koncentruje się na cechach tego języka, niuansach działania czy też szczegółach i specyfice implementacji. Tymczasem o jakości oprogramowania decyduje jego projekt. To właśnie dzięki niemu można tworzyć oprogramowanie, które będzie łatwe do utrzymania, modyfikowania, rozszerzania i testowania. Problem polega na tym, że projektowanie oprogramowania jest trudnym i wymagającym zadaniem.

Ta książka jest doskonałym uzupełnieniem podręcznej biblioteczki każdego programisty C++. Opisano w niej znaczenie dobrego projektu oprogramowania i zasady tworzenia takich projektów. Omówiono szereg przydatnych wzorców projektowych, dzięki którym łatwiej można zrozumieć, jak zarządzać zależnościami i abstrakcjami, zwiększać możliwości modyfikowania i rozszerzania tworzonego kodu, a także stosować nowoczesne wzorce projektowe. Zaprezentowano wzorce wykorzystujące wiele różnych paradygmatów, w tym programowanie: obiektowe, uogólnione i funkcyjne. Pokazano też największą zaletę nowoczesnego języka C++: możliwość łączenia różnych paradygmatów i tworzenia oprogramowania, które przetrwa dziesięciolecia.

W książce między innymi:

- ocena kodu pod kątem projektowania oprogramowania
- sposoby realizacji takich celów projektowych jak łatwość modyfikowania i rozszerzania kodu
- zalety i wady różnych koncepcji projektowania
- przydatność wzorców projektowych w rozwiązywaniu problemów
- zasady wyboru form wzorców projektowych

Dr Klaus Iglberger jest niezależnym ekspertem, trenerem i konsultantem w dziedzinie programowania C++. Jest też współautorem kilku frameworków symulacyjnych i biblioteki matematycznej Blaze C++. Wcześniej był dyrektorem zarządzającym Centralnego Instytutu Obliczeń Naukowych na Uniwersytecie Erlangen-Norymberga.

Z tej książki nauczyłem się znacznie więcej, niż mógłbym oczekiwać.

Mark Summerfield, właściciel Qtrac Ltd.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-8322-720-7
9 788383 227207
Cena: 89,00 zł