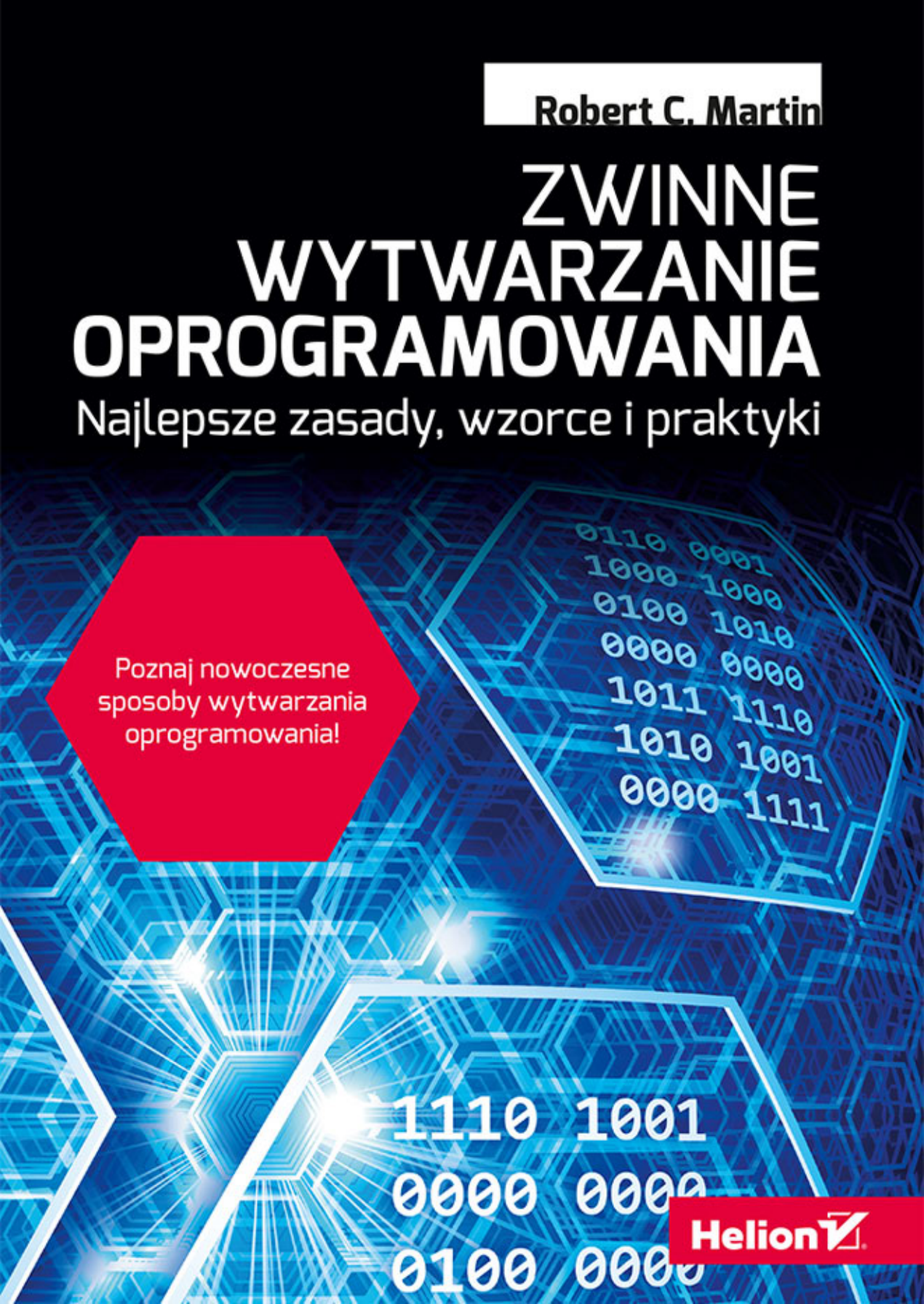
Robert C. Martin

ZWINNE WYTWARZANIE OPROGRAMOWANIA

Najlepsze zasady, wzorce i praktyki



Helion

Tytuł oryginału: Agile Software Development, Principles, Patterns, and Practices

Tłumaczenie: Radosław Meryk

Projekt okładki: Studio Gravite / Olsztyn; Obarek, Pokoński, Pazdrijowski, Zaprucki
Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock Images LLC.

ISBN: 978-83-246-9682-6

Authorized translation from the English language edition, entitled:
AGILE SOFTWARE DEVELOPMENT, PRINCIPLES, PATTERNS, AND PRACTICES;
ISBN 0135974445; by Robert C. Martin; published by Pearson Education, Inc;
publishing as Prentice Hall. Copyright © 2003 by Pearson Education, Inc.

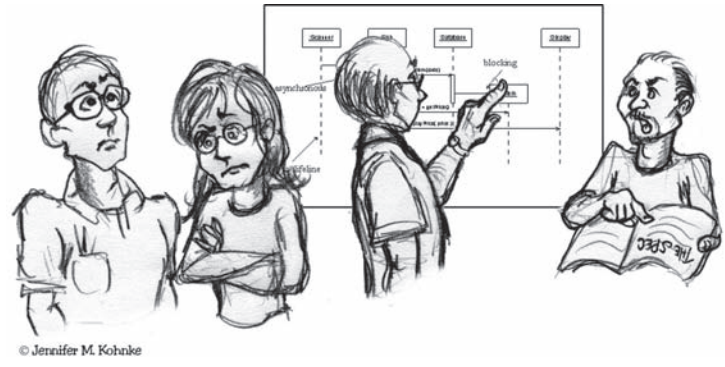
Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Spis treści

Lista wzorców projektowych

ROZDZIAŁ 7

Co to jest projekt agile?



Po zapoznaniu się z cyklem rozwoju oprogramowania doszedłem do wniosku, że jedyną dokumentacją oprogramowania, która rzeczywiście wydaje się spełniać kryteria projektu technicznego, są listingi z kodem źródłowym

— Jack Reeves

W 1992 roku Jack Reeves napisał słynny artykuł w magazynie „C++ Journal" zatytułowany *What is Software Design?*[2]. W tym artykule Reeves twierdzi, że projekt systemu oprogramowania jest udokumentowany głównie za pomocą jego kodu źródłowego. Diagramy reprezentujące kod źródłowy są uzupełniające w stosunku do projektu i same w sobie nie są projektem. Jak się okazało, artykuł Jacka stał się zwiastunem rozwoju agile.

Na kolejnych stronach tej książki często będziemy mówili o projekcie. Nie należy zakładać, że oznacza to zestaw diagramów UML oddzielonych od kodu. Zbiór diagramów UML może reprezentować części projektu, ale to nie jest projekt. Projekt oprogramowania jest pojęciem abstrakcyjnym. Wiąże się zarówno z ogólnym kształtem i strukturą programu, jak również ze szczegółowym kształtem i strukturą każdego modułu, klasy i metody. Może być reprezentowany za pomocą wielu różnych mediów, ale jego ostateczną postacią jest kod źródłowy. Ostatecznie projektem jest kod źródłowy.

Co złego dzieje się z oprogramowaniem?

Jeśli ktoś ma szczęście, to rozpoczyna projekt, mając jasny obraz tego, jaką postać ma system przyjąć. Projekt systemu jest żywym obrazem w naszym umyśle. Jeśli ktoś ma jeszcze więcej szczęścia, to klarowność towarzyszy projektowi do pierwszej wersji dystrybucyjnej.

[2] [Reeves 92] To doskonały artykuł. Gorąco zachęcam do jego przeczytania. Dołączyłem go do tej książki jako dodatek D.

Wtedy coś zaczyna iść nie tak. Oprogramowanie zaczyna się psuć jak kawałek starego mięsa. W miarę upływu czasu zepsucie rozprzestrzenia się i rośnie. W kodzie gromadzą się brzydkie, ropiejące wrzody i czyraki, sprawiając, że jego utrzymanie staje się coraz trudniejsze.

Ostatecznie wysiłek, jaki trzeba włożyć, aby wprowadzić nawet najprostsze zmiany, staje się tak uciążliwy, że deweloperzy i menedżerowie zaczynają nalegać na zaprojektowanie systemu od początku.

Takie przedsięwzięcia rzadko kończą się sukcesem. Chociaż projektanci mają dobre intencje, to okazuje się, że strzelają do ruchomego celu. Stary system ewoluuje i zmienia się, a odzwierciedlenie tych zmian musi znaleźć miejsce w nowym projekcie. Brodawki i wrzody gromadzą się w nowym projekcie, zanim powstanie jego pierwsza wersja dystrybucyjna.

Zapachy projektu — woń psującego się oprogramowania

Psujące się oprogramowanie można rozpoznać po dowolnym z następujących zapachów:

1. **Sztywność** — system jest trudny do zmiany, ponieważ każda zmiana wymusza wiele innych zmian w innych częściach systemu.
2. **Kruchość** — zmiany powodują psucie się systemu w miejscach, które na pozór nie mają związku z tą częścią, która została zmieniona.
3. **Brak mobilności** — trudno rozdzielić system na komponenty, które mogą być ponownie użyte w innych systemach.
4. **Lepkość** — stosowanie dobrych praktyk jest trudniejsze niż stosowanie złych praktyk.
5. **Zbędna złożoność** — projekt zawiera infrastrukturę, z której nie ma bezpośredniego pożytku.
6. **Zbędne powtórzenia** — projekt zawiera powtarzające się struktury, które mogłyby być ujednolicone w ramach pojedynczej abstrakcji.
7. **Nieczytelność** — system jest trudny do czytania i zrozumienia. Kod nie komunikuje dobrze swojej intencji.

Sztywność. Sztywność to cecha oprogramowania polegająca na trudnym wprowadzaniu nawet najprostszych zmian. Projekt jest sztywny, jeśli pojedyncza zmiana powoduje kaskadę kolejnych zmian w modułach zależnych. Im więcej modułów, które muszą być zmienione, tym bardziej sztywny projekt.

Większość programistów wcześniej czy później spotkała się z taką sytuacją. Ktoś prosi nas o wprowadzenie zmiany, która wydaje się być prosta. Zapoznajemy się z problemem i dokonujemy rozsądnego oszacowania nakładów wymaganej pracy. Ale później, podczas pracy nad wprowadzeniem zmiany, okazuje się, że istnieją reperkusje zmiany, których nie przewidzieliśmy. Musimy przebijać się przez ogromne porcje kodu i modyfikować znacznie więcej modułów, niż początkowo ocenialiśmy. Ostatecznie zmiany zajmują znacznie więcej czasu, niż szacowaliśmy. Na pytanie, dlaczego oszacowanie było takie niedokładne, powtarzamy tradycyjny lament programisty: „To było o wiele bardziej skomplikowane, niż sądziłem!".

Kruchość. Kruchość jest tendencją oprogramowania do psucia się w wielu miejscach po wprowadzeniu pojedynczej zmiany. Często nowe problemy pojawiają się w obszarach, które nie mają koncepcyjnego związku z obszarem, który został zmieniony. Naprawienie tych problemów prowadzi do jeszcze większej liczby problemów, a zespół deweloperski zaczyna przypominać psa, który goni własny ogon.

W miarę wzrostu kruchości modułu prawdopodobieństwo, że wprowadzenie zmiany spowoduje nieoczekiwane problemy, jest bliskie pewności. Wydaje się to absurdalne, ale takie moduły wcale nie należą do rzadkości. Są to moduły, które stale wymagają naprawy. Nigdy nie znikają z listy błędów. Programiści wiedzą, że trzeba je zaprojektować od nowa (ale nikt nie chce się zmierzyć z zadaniem ich przebudowy). Im więcej poprawek w nich wprowadzamy, tym stają się *gorsze*.

Brak mobilności. Projekt jest niemobilny, gdy zawiera części, które mogłyby być użyteczne w innych systemach, ale wysiłek i ryzyko związane z oddzieleniem tych części od pierwotnego systemu są zbyt duże. Jest to sytuacja niefortunna, ale niestety bardzo częsta.

Lepkość. Lepkość może mieć dwie formy: lepkość oprogramowania oraz lepkość środowiska.

W obliczu konieczności wprowadzenia zmiany programiści zwykle wymyślają więcej niż jeden sposób na wprowadzenie tej zmiany. Niektóre ze sposobów zachowują projekt, inne nie (są to tzw. *hacki* — ang. *hacks*). Gdy sposoby zachowujące projekt są trudniejsze do wprowadzenia niż *hacki*, to lepkość projektu jest wysoka. Łatwo wprowadza się zmiany w niewłaściwy sposób, natomiast trudno zrobić to właściwie. Należy dążyć do zaprojektowania oprogramowania w taki sposób, aby wprowadzanie zmian, które zachowują projekt, było łatwe do wykonania.

Lepkość środowiska powstaje w sytuacji, gdy środowisko programistyczne jest powolne i niewydajne. Na przykład jeśli czasy kompilacji są bardzo długie, deweloperzy mają pokusę, aby dokonywać zmian, które nie wymuszają dużych rekompilacji, choć zmiany te nie zachowują projektu. Jeśli pobranie zaledwie kilku plików z systemu kontroli kodu źródłowego wymaga godzin, to deweloperzy będą się starać wprowadzać zmiany, które wymagają jak najmniej pobrań z systemu kontroli wersji, bez względu na to, czy projekt jest zachowany, czy nie.

W obu przypadkach lepki projekt to taki, w którym trudno utrzymać strukturę oprogramowania. Należy stworzyć systemy i środowiska projektowe, które ułatwiają zachowywanie projektu.

Zbytnia złożoność. Projekt zawiera niepotrzebną złożoność, gdy zawiera elementy, które w danej chwili nie są użyteczne. To często się dzieje, gdy deweloperzy przewidują zmiany w wymaganiach i umieszczają w oprogramowaniu mechanizmy mające na celu obsługę tych potencjalnych zmian. Początkowo może się wydawać, że to jest dobre. W końcu przygotowywanie się do przyszłych zmian powinno utrzymać elastyczność kodu i zapobiec koszmarnym zmianom w późniejszym okresie.

Niestety, efekt często jest dokładnie odwrotny. Poprzez przygotowywanie się do zbyt wielu zmian projekt jest zaśmiecony konstrukcjami, które nigdy nie są używane. Niektóre spośród tych przygotowań mogą się opłacać, ale wiele innych nie. W międzyczasie projekt znosi ciężar tych niewykorzystanych elementów konstrukcyjnych. W ten sposób oprogramowanie staje się skomplikowane i trudne do zrozumienia.

Niepotrzebne powtórzenia. Operacje wytnij i wklej mogą być przydatne do edycji tekstu, ale w przypadku edycji kodu ich stosowanie może mieć katastrofalne skutki. Bardzo często systemy oprogramowania są budowane na dziesiątkach lub setkach powtarzających się elementów kodu. Dochodzi do tego w następujący sposób:

Rafał musi napisać kod, który realizuje funkcjonalność A. Przegląda inne części kodu, w których podejrzewa występowanie podobnych operacji, i znajduje odpowiedni fragment kodu. Wycina ten kod i wkleja do swojego modułu, a następnie wprowadza odpowiednie modyfikacje.

Rafał nie wie, że kod, który wyciął za pomocą myszy, został tam wprowadzony przez Tadka, który wyciął go z modułu napisanego przez Lidię. Lidia była pierwszą osobą, która zrealizowała funkcjonalność A, ale zdała sobie sprawę, że ta funkcjonalność była bardzo podobna do funkcjonalności B. Znalazła implementację tej funkcjonalności, wycięła i wkleiła ją do swojego modułu, a następnie wprowadziła niezbędne modyfikacje.

Kiedy ten sam kod pojawia się w kółko w wielu miejscach, choć w nieco innej formie, deweloperom brakuje abstrakcji. Wyszukiwanie wszystkich powtórzeń i eliminowanie ich z wykorzystaniem odpowiedniej abstrakcji może nie znajdować się zbyt wysoko na ich liście priorytetów, ale wykonanie tej czynności znacząco przyczynia się do tworzenia systemu, który jest łatwiejszych do zrozumienia i utrzymania.

Gdy w systemie jest zbędny kod, zadanie modyfikowania systemu może stać się uciążliwe. Błędy znalezione w takiej powtarzającej się jednostce muszą być poprawiane we wszystkich powtórzeniach. Ponieważ jednak każde powtórzenie nieco różni się od pozostałych, poprawka nie zawsze jest taka sama.

Nieczytelność. Nieczytelność jest cechą modułu polegającą na trudności w jego zrozumieniu. Kod może być napisany w jasny i czytelny sposób lub może być napisany w sposób nieprzejrzysty i zawiły. Kod, który ewoluuje, z biegiem czasu wykazuje tendencję do stawania się coraz bardziej nieprzezroczystym. Utrzymanie nieprzezroczystości na minimalnym poziomie wymaga stałego wysiłku. Trzeba ciągle dbać o to, by kod był czytelny i ekspresywny.

Kiedy deweloperzy piszą moduł po raz pierwszy, kod może im się wydawać jasny. To dlatego, że wgłębili się w kod i doskonale go rozumieją. Po jakimś czasie, gdy wracają do tego modułu, często zastanawiają się, jak mogli napisać coś tak okropnego. Aby temu zapobiec, deweloperzy powinni starać się stawiać w pozycji czytelników swojego kodu i podejmować wysiłki zmierzające do refaktoryzacji kodu w taki sposób, aby czytelnicy kodu mogli go zrozumieć. Kod powinien być także przeglądany przez inne osoby.

Co stymuluje oprogramowanie do psucia się?

W środowiskach, które nie stosują metodyki agile, projekty pogarszają się z powodu zmian wymagań w sposób, którego pierwotny projekt nie przewidywał. Często zmiany te muszą być wykonane szybko i mogą być wykonywane przez deweloperów, którzy nie znają pierwotnej filozofii projektu. Tak więc choć zmiana w projekcie działa, to w jakiś sposób narusza oryginalny projekt. Kawałek po kawałku, w miarę wprowadzania kolejnych zmian, naruszenia te kumulują się i projekt zaczyna „pachnieć".

Nie możemy jednak winić dryfowania wymagań za degradację projektu. Deweloperzy powinni doskonale zdawać sobie sprawę, że wymagania się zmieniają. Rzeczywiście, większość z nas zdaje sobie sprawę, że wymagania są najbardziej lotnymi elementami projektu. Jeżeli projekty nie udają się ze względu na ciągły deszcz zmieniających się wymagań, to winę za to ponoszą stosowane przez nas praktyki projektowe. Trzeba znaleźć jakiś sposób, aby projekty były odporne na zmiany, i zastosować praktyki, które zabezpieczają kod przed psuciem się.

Zespoły agile nie pozwalają psuć się oprogramowaniu

Zespół agile kwitnie w obliczu zmian. Zespół niewiele inwestuje w początkowe prace, dlatego nie jest przywiązany do wyjściowego projektu. Zamiast tego utrzymuje konstrukcję systemu w sposób maksymalnie czysty i prosty i uzasadnia to za pomocą wielu testów jednostkowych i testów akceptacyjnych. Dzięki temu projekt jest elastyczny, a wprowadzanie w nim zmian jest łatwe. Zespół korzysta z tej elastyczności w celu ciągłego poprawiania projektu. Dzięki temu każda iteracja kończy się powstaniem systemu, którego projekt jest jak najlepszy w stosunku do wymagań w tej iteracji.

Program Copy

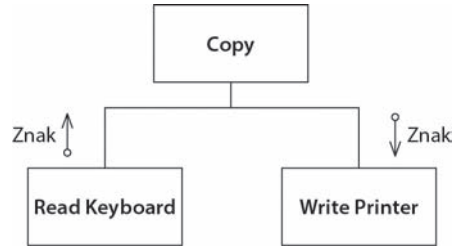
Aby zilustrować powyższe punkty, warto przyjrzeć się procesowi psucia się oprogramowania. Powiedzmy, że szef przyszedł do Ciebie w poniedziałek rano i poprosił o napisanie programu, który kopiuje znaki z klawiatury na drukarkę. Po wykonaniu w myślach kilku szybkich ćwiczeń umysłowych doszedłeś do wniosku, że taki program nie powinien zająć więcej niż dziesięć linijek kodu. Projektowanie i kodowanie nie powinny zająć więcej niż godzinę. Razem ze spotkaniami grup interdyscyplinarnych, szkoleniami jakości, codziennymi spotkaniami grup badających postępy prac i trzema bieżącymi kryzysami w branży zakończenie tego programu powinno zająć około tygodnia — jeśli zostaniesz po godzinach. Zawsze jednak nasze szacunki mnożymy przez trzy.

„Trzy tygodnie" — takiej odpowiedzi udzielasz szefowi. Szef mruczy coś pod nosem i zgadza się, pozostawiając Cię z zadaniem.

Wstępny projekt. Masz teraz trochę czasu, zanim rozpocznie się spotkanie poświęcone przeglądowi procesu, więc decydujesz, że narysujesz projekt programu. Przy zastosowaniu projektu strukturalnego doszedłeś do diagramu struktury pokazanego na rysunku 7.1.

Aplikacja składa się z trzech modułów lub podprogramów. Moduł `Copy` wywołuje dwa pozostałe. Program kopiujący pobiera znaki z modułu `Read Keyboard` i kieruje je do modułu `Write Printer`.

Patrzysz na swój projekt i oceniasz, że jest dobry. Uśmiechasz się i wychodzisz z biura, by pójść na spotkanie. Przynajmniej będzie można tam trochę pospać.



Rysunek 7.1. *Diagram struktury programu Copy*

We wtorek przyszedłeś nieco wcześniej, dzięki czemu możesz zakończyć pracę nad programem `Copy`. Niestety, jeden z kryzysów w branży dał o sobie znać w nocy. W związku z tym musisz iść do laboratorium i pomóc w debugowaniu problemu. W przerwie na obiad, na którą w końcu wyszedłeś o 15.00, udało Ci się napisać kod programu `Copy`. Efekt pokazano na listingu 7.1.

Listing 7.1. *Program Copy*

```
void Copy()
{
    int c;
    while ((c=RdKbd()) != EOF)
        WrtPrt(c);
}
```

Ledwo udało Ci się zapisać edytowany kod, kiedy zdałeś sobie sprawę, że jesteś już spóźniony na spotkanie jakości. Wiesz, że to spotkanie jest ważne. Będzie poświęcone znaczeniu produkcji bezusterkowej. Zatem chowasz swoje krakersy i colę i udajesz się na spotkanie.

W środę znów przyszedłeś wcześniej. Tym razem masz nadzieję, że nic nie stanie Ci na przeszkodzie. Otwierasz kod źródłowy programu `Copy` i zaczynasz go kompilować. I oto program kompiluje się za pierwszym razem — bez błędów! Dobre i to, bo Twój szef woła Cię na nieplanowane spotkanie na temat konieczności oszczędzania tonerów drukarek laserowych.

W czwartek, po czterech godzinach spędzonych na rozmowie przez telefon z serwisantem w Rocky Mount w Karolinie Północnej, udzieleniu mu zdalnej pomocy w debugowaniu i rejestrowaniu błędów w jednym z bardziej nieczytelnych elementów systemu, otrzymałeś podziękowania i zaczynasz testować program `Copy`. Działa! Za pierwszym razem! To dobrze, bo Twój nowy uczeń właśnie usunął główny katalog z kodem źródłowym z serwera. Musisz więc znaleźć taśmy z najnowszą kopią zapasową i odtworzyć go. Oczywiście ostatnia pełna kopia zapasowa została zrobiona trzy miesiące temu. Dlatego musisz odtworzyć jeszcze dziewięćdziesiąt cztery przyrostowe kopie zapasowe.

Piątek wydaje się zupełnie wolny od dodatkowych zajęć. To dobrze, ponieważ cały dzień zajmuje Ci pomyślne załadowanie programu Copy do systemu kontroli kodu źródłowego.

Oczywiście program odnosi niezwykły sukces i jest wdrażany w całej firmie. Twoja reputacja jako programistycznego asa ponownie została potwierdzona. Możesz teraz wygrzać się w chwale swoich osiągnięć. Przy odrobinie szczęścia może faktycznie uda Ci się napisać w tym roku trzydzieści linijek kodu!

Wymagania. One się zmieniają! Kilka miesięcy później szef przychodzi do Ciebie i mówi, że chciałby, aby program `Copy` mógł czytać z czytnika taśmy papierowej. Zgrzytasz zębami i przewracasz oczami. Zastanawiasz się, dlaczego ludzie zawsze zmieniają wymagania. Twój program nie był projektowany z myślą o obsłudze czytnika taśmy papierowej! Ostrzegasz swojego szefa, że wprowadzenie tego rodzaju zmian może naruszyć elegancję projektu. Niemniej jednak szef jest nieugięty. Mówi, że użytkownicy naprawdę potrzebują od czasu do czasu czytać znaki z czytnika taśmy papierowej.

Zatem wzdychasz i planujesz wprowadzenie modyfikacji. Chcesz dodać do funkcji `Copy` argument typu `Boolean`. Jeśli będzie miał wartość `true`, będziesz czytać z czytnika taśmy papierowej; jeśli będzie miał wartość `false`, będziesz czytać z klawiatury, tak jak przedtem. Niestety, teraz już tak wiele innych

programów korzysta z programu Copy, że nie można zmieniać interfejsu. Zmiana interfejsu spowodowałaby, że ponowna kompilacja zajęłaby całe tygodnie. Sami inżynierowie testów zlinczowaliby Cię, nie wspominając nawet o siedmiu gościach z grupy zarządzania konfiguracją. Trzeba by również poświęcić wiele czasu na przeglądanie kodu każdego modułu, który wywołuje program Copy.

Nie. Zmiana interfejsu nie wchodzi w rachubę. Ale w takim razie w jaki sposób poinformować program Copy o tym, że powinien czytać z czytnika taśmy papierowej? Oczywiście użyjesz zmiennej globalnej. Użyjesz także najlepszej i najwartościowszej funkcji grupy języków C-podobnych — operatora ?:. Efekt pokazano na listingu 7.2.

Listing 7.2. *Pierwsza modyfikacja programu Copy*

```
bool ptFlag = false;
// pamiętaj, żeby zresetować tę flagę
void Copy()
{
    int c;
    while ((c=(ptflag ? RdPt() : RdKbd())) != EOF)
        WrtPrt(c);
}
```

Programy wywołujące funkcję Copy, które chcą czytać z czytnika taśmy papierowej, muszą najpierw ustawić flagę ptFlag na true. Później mogą wywołać funkcję Copy, a ta będzie szczęśliwie czytać z czytnika taśmy papierowej. Kiedy funkcja Copy zwróci sterowanie, proces wywołujący musi zresetować flagę ptFlag. Gdyby tego nie zrobił, to następny wywołujący mógłby przez pomyłkę czytać z czytnika taśmy papierowej zamiast z klawiatury. Aby przypomnieć programistom o obowiązku zresetowania tej flagi, dodałeś odpowiedni komentarz.

Po raz kolejny publikujesz wersję dystrybucyjną. Odnosi nawet większy sukces niż wcześniej, a hordy chętnych programistów czekają na okazję, aby z niej skorzystać. Życie jest piękne.

Dać im palec... Kilka tygodni później Twój szef (który nadal jest Twoim szefem pomimo trzech reorganizacji w całej korporacji na przestrzeni wielu miesięcy) przychodzi do Ciebie i mówi, że klienci chcieliby, aby program Copy mógł pisać na dziurkarce taśmy papierowej.

Ach ci klienci! Zawsze rujnują nasze projekty. *Pisanie oprogramowania byłoby dużo łatwiejsze, gdyby nie to, że istnieją klienci.*

Mówisz szefowi, że te nieustanne zmiany mają głęboko negatywny wpływ na elegancję Twojego projektu. Ostrzegasz go, że jeśli zmiany w dalszym ciągu będą wprowadzane w tym strasznym tempie, to oprogramowanie do końca roku będzie niemożliwe do utrzymania. Twój szef kiwa głową ze zrozumieniem, a potem mówi, aby i tak dokonać zmian.

Ta zmiana w projekcie jest podobna do poprzedniej. Wystarczy kolejna zmienna globalna i kolejny operator ?:. Efekt wysiłków pokazano na listingu 7.3.

Listing 7.3. *Druga modyfikacja programu Copy*

```
bool ptFlag = false;
bool punchFlag = false;
// pamiętaj, żeby zresetować te flagi
void Copy()
{
    int c;
    while ((c=(ptflag ? RdPt() : RdKbd())) != EOF)
        punchFlag ? WrtPunch(c) : WrtPrt(c);
}
```

Szczególnie dumny jesteś z faktu, że pamiętałeś o tym, aby zmienić komentarz. Mimo to martwisz się, że struktura programu zaczyna się przewracać. Każda dodatkowa zmiana urządzenia wejściowego z pewnością zmusi Cię do całkowitej przebudowy pętli while. Być może nadszedł czas, aby odkurzyć swoje CV...

Oczekiwanie zmian. Ocenie czytelnika pozostawiam ustalenie, jak wiele z powyższego opisu było satyryczną przesadą. Sednem tej historii było pokazanie, że w obliczu zmian projekt programu może ulec szybkiej degradacji. Wyjściowy projekt programu `Copy` był prosty i elegancki. Wystarczyły jednak dwie zmiany, aby zaczął wykazywać oznaki **sztywności**, **kruchości**, **braku mobilności**, **złożoności**, **redundancji** i **nieczytelności**. Ten trend z pewnością będzie trwał, a program przekształci się w śmietnik.

Możemy usiąść i zrzucać winę za ten stan rzeczy na zmiany. Możemy narzekać, że program został dobrze zaprojektowany zgodnie z oryginalną specyfikacją, a późniejsze zmiany w specyfikacji spowodowały degradację projektu. Jednak takie podejście ignoruje jeden z najbardziej znanych faktów w produkcji oprogramowania: *wymagania zawsze się zmieniają*!

Należy pamiętać, że najbardziej ulotną rzeczą w większości projektów wytwarzania oprogramowania są wymagania. Wymagania przez cały czas są w stanie płynnym. To jest fakt, który my deweloperzy musimy zaakceptować! *Żyjemy w świecie zmieniających się wymagań, a naszym zadaniem jest zapewnienie, aby nasze oprogramowanie przetrwało te zmiany*. Jeżeli projekt naszego oprogramowania degraduje się, ponieważ zmieniły się wymagania, to nie jesteśmy agile.

Przykład programu Copy wykonanego zgodnie z metodyką agile

Produkcja agile programu `Copy` mogła rozpocząć się dokładnie tak jak wcześniej — od kodu z listingu 7.1[3]. Kiedy szef poprosił deweloperów agile, aby przystosowali program do czytania z czytnika taśmy papierowej, mogliby zareagować zmodyfikowaniem projektu w taki sposób, aby był odporny na tego rodzaju zmiany. Efekt mógłby wyglądać podobnie do kodu z listingu 7.4.

Listing 7.4. *Wersja numer 2 programu Copy wykonanego zgodnie z metodyką agile*

```
class Reader
{
    public:
        virtual int read() = 0;
};

class KeyboardReader : public Reader
{
    public:
        virtual int read() {return RdKbd ();}
};

KeyboardReader GdefaultReader;

void Copy(Reader& reader = GdefaultReader)
{
    int c;
    while ((c=reader.read()) != EOF)
        WrtPrt(c);
}
```

Zamiast próbować łatać projekt, aby sprostać nowemu wymaganiu, zespół wykorzystał okazję do poprawienia projektu w taki sposób, aby w przyszłości był odporny na tego rodzaju zmiany. Od tej pory, gdy szef poprosi o obsługę nowego rodzaju urządzenia wejściowego, zespół będzie w stanie zareagować w sposób, który nie powoduje degradacji programu `Copy`.

Zespół zastosował zasadę otwarte-zamknięte (ang. *Open-Closed Principle* — OCP), o której można przeczytać w rozdziale 9. Według tej zasady moduły powinny być projektowane w taki sposób, by można je było rozszerzać bez modyfikowania. To jest dokładnie to, co zrobił zespół. Można dostarczyć dowolne nowe urządzenie wejściowe, o które szef poprosi, bez modyfikowania programu `Copy`.

[3] W rzeczywistości jest bardzo prawdopodobne, że stosowanie praktyki produkcji sterowanej zmusiłoby do stworzenia projektu na tyle elastycznego, że udałoby mu się przetrwać bez zmian żądania szefa. Jednak w tym przykładzie zignorujemy to.

Zwróćmy jednak uwagę, że zespół nie próbował przewidywać, jak program będzie się zmieniał, gdy projektował moduł po raz pierwszy. Zamiast tego program został napisany w najprostszy możliwy sposób. Dopiero gdy zmieniły się wymagania, zespół zmienił projekt modułu w taki sposób, aby był odporny na tego rodzaju zmiany.

Można by się spierać, że członkowie zespołu wykonali tylko połowę pracy. Gdy zabezpieczali się przed różnymi urządzeniami wejściowymi, mogli również zabezpieczyć się przed różnymi urządzeniami wyjściowymi. Jednak zespół w rzeczywistości nie miał pojęcia, czy urządzenia wyjściowe kiedykolwiek się zmienią. Wprowadzenie dodatkowej ochrony w tym momencie byłoby pracą, która nie służyłaby żadnemu bieżącemu celowi. Jest oczywiste, że jeśli będzie potrzebne takie zabezpieczenie, łatwo będzie je można dodać później. W związku z tym w rzeczywistości nie ma powodu, aby dodawać je teraz.

Skąd deweloperzy agile wiedzieli, co należy zrobić?

Deweloperzy agile w przykładzie zamieszczonym powyżej stworzyli klasę abstrakcyjną, aby zabezpieczyć się przed zmianami urządzenia wejściowego. Skąd wiedzieli, jak należy to zrobić? Jest to związane z jednym z podstawowych założeń projektu obiektowego.

Początkowy projekt programu `Copy` jest nieelastyczny ze względu na *kierunek* jego zależności. Spójrzmy ponownie na rysunek 7.1. Zwróćmy uwagę, że moduł `Copy` zależy bezpośrednio od modułów `Read Keyboard` oraz `Write Printer`. W tej aplikacji `Copy` jest modułem wysokiego poziomu. Ten moduł określa strategię aplikacji. „Wie", jak kopiować znaki. Niestety, został również uzależniony od niskopoziomowych szczegółów klawiatury i drukarki. Z tego powodu zmiana niskopoziomowych szczegółów będzie miała wpływ na wysokopoziomową strategię.

Kiedy ta nieelastyczność uwidoczniła się, deweloperzy agile wiedzieli, że zależność modułu `Copy` od urządzenia wejściowego będzie trzeba *odwrócić*[4] tak, aby moduł `Copy` przestał zależeć od urządzenia wejściowego. Wtedy zastosowano wzorzec Strategia[5] w celu utworzenia żądanej inwersji.

A zatem w skrócie — deweloperzy agile wiedzieli, co należy zrobić, ponieważ:

1. Wykryli problem, ponieważ postępowali zgodnie z praktykami agile.
2. Zdiagnozowali problem poprzez zastosowanie zasad projektowania.
3. Rozwiązali problem poprzez zastosowanie odpowiedniego wzorca projektowego.

Współdziałanie tych trzech aspektów wytwarzania oprogramowania *jest* aktem projektowania.

Utrzymywanie projektu w jak najlepszej postaci

Deweloperzy agile starają się utrzymywać projekt w taki sposób, aby był jak najwłaściwszy i jak najczystszy. To zobowiązanie nie jest ani przypadkowe, ani niepewne. Deweloperzy agile nie „porządkują" projektu co kilka tygodni. Zamiast tego starają się utrzymywać oprogramowanie w maksymalnie czystej i ekspresywnej postaci w każdym dniu, o każdej godzinie i nawet w każdej minucie. Nigdy nie mówią „wrócimy do tego i naprawimy później". Nigdy nie pozwalają na to, aby oprogramowanie zaczęło się psuć.

Postawa, jaką deweloperzy agile stosują w odniesieniu do projektu oprogramowania, jest taka sama jak postawa, którą stosują chirurdzy w odniesieniu do zachowania sterylności. To właśnie zachowanie procedur sterylności sprawia, że chirurgia jest *możliwa*. Bez niej ryzyko infekcji byłoby zbyt wysokie. Deweloperzy agile stosują takie samo podejście do swoich projektów. Ryzyko pozwolenia sobie na nawet najmniejsze objawy zepsucia jest zbyt wysokie, aby można je było tolerować.

Projekt musi być utrzymywany w czystości, a ponieważ kod źródłowy jest najważniejszym wyrazem projektu, on także musi pozostać czysty. Profesjonalizm podpowiada, że programiści nie mogą tolerować psucia się kodu.

[4] Zobacz zasadę odwracania zależności (*Dependency Inversion Principle* — DIP) w rozdziale 11.

[5] Wzorzec Strategia omówimy bardziej szczegółowo w rozdziale 14.

Wniosek

A zatem co to jest projekt agile? Projekt agile jest procesem, a nie zdarzeniem. Polega na ciągłym stosowaniu zasad, wzorców i praktyk w celu poprawy struktury i czytelności oprogramowania. To zaangażowanie w ciągłe utrzymywanie projektu systemu w maksymalnie prostej, czystej i ekspresywnej postaci.

W następnych rozdziałach będziemy omawiali zasady i wzorce projektowania oprogramowania. Podczas lektury należy pamiętać, że deweloper agile nie stosuje tych zasad i wzorców do wysokopoziomowego projektu „z góry". Zamiast tego praktyki te są stosowane od iteracji do iteracji, jako próba utrzymania kodu i projektu, który on obejmuje, w jak najczystszej postaci.

Bibliografia

1. Jack Reeves, *What Is Software Design?*, „C++ Journal", wolumin 2, nr 2. 1992. Dostępny pod adresem *http://www.bleading-edge.com/ Publications/C++Journal/Cpjour2.htm*.

Skorowidz

H

I

J

K

L

M

N

O

P

R

S

T

U, W

Z

ZWINNE WYTWARZANIE **OPROGRAMOWANIA**

Najlepsze zasady, wzorce i praktyki

Czasy kaskadowego tworzenia projektów odchodzą w niepamięć. Obecne tempo rozwoju aplikacji i rynku nie pozwala poświęcać miesięcy na analizę, tworzenie dokumentacji, projektowanie, a na końcu wytwarzanie, testowanie i wdrażanie. Produkt musi być dostępny błyskawicznie! Pozwala to na natychmiastowe zebranie opinii na jego temat, dostosowanie go do oczekiwań i szybkie reagowanie na wymagane zmiany. Takie założenia może spełnić tylko zespół wytwarzający oprogramowanie w zwinny sposób!

Ta książka została w całości poświęcona zwinnym praktykom. Sięgnij po nią i przekonaj się, w jaki sposób planować kolejne iteracje, tworzyć kod, a następnie go testować. W trakcie lektury poznasz praktyczne aspekty zwinnego tworzenia kodu — zobaczysz, jak istotne są zasady: pojedynczej odpowiedzialności, podstawienia Liskov czy odwracania zależności. Znajdziesz tu także zasady projektowania pakietów oraz przydatne wzorce projektowe, omówione na konkretnych przykładach. Książka ta jest obowiązkową lekturą dla wszystkich osób zaangażowanych w wytwarzanie oprogramowania i chcących udoskonalić swój proces.

Dzięki tej książce:

- poznasz fundamenty zwinnego wytwarzania oprogramowania
- zaznajomisz się z najlepszymi praktykami
- dowiesz się, jak refaktoryzować kod
- wybierzesz właściwe wzorce i unikniesz typowych błędów
- poprawisz swój proces wytwarzania oprogramowania

Zwinne programowanie = większa szansa na sukces projektu!

Helion

27248 numer katalogowy

księgarnia internetowa

http://helion.pl

zamówienia telefoniczne

0 801 339900

0 601 339900

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

ISBN 978-83-246-9682-6

9 788324 696826

Informatyka w najlepszym wydaniu

cena: 89,00 zł