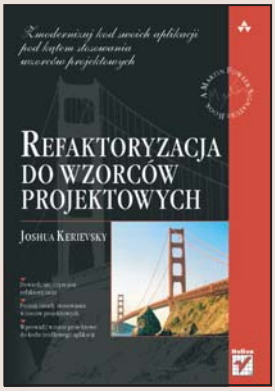
Refaktoryzacja do wzorców projektowych



Autor: Joshua Kerievsky
Tłumaczenie: Paweł Koronkiewicz
ISBN: 83-7361-930-5
Tytuł oryginału: Refactoring to Patterns
Format: B5, stron: 320

Zmodernizuj kod swoich aplikacji pod kątem stosowania wzorców projektowych

- Dowiedz się, czym jest refaktoryzacja
- Poznaj zasady stosowania wzorców projektowych
- Wprowadź wzorce projektowe do kodu źródłowego aplikacji

Refaktoryzacja to zmiana konstrukcji kodu bez modyfikowania jego działania. Najczęstszym powodem refaktoryzowania kodu jest konieczność jego „uporządkowania" lub usunięcia z niego funkcji niewykorzystywanych w projekcie. Często również stosuje się refaktoryzację, aby zmodernizować kod pod kątem zastosowania w nim wzorców projektowych. Wprowadzenie wzorców projektowych do kodu znacznie ułatwia jego późniejsze modyfikacje i ewentualne rozbudowy. Stosowanie technik programowania ekstremalnego nierozerwalnie wiąże ze sobą wzorce projektowe i refaktoryzację kodu.

Książka „Refaktoryzacja do wzorców projektowych" opisuje teoretyczne i praktyczne zagadnienia związane z refaktoryzowaniem kodu pod kątem wzorców projektowych. Przedstawia opisy niskopoziomowych przekształceń, które umożliwiają programiście bezpieczną zmianę konstrukcji kodu prowadzącą do zaimplementowania bądź usunięcia określonych wzorców z programu. Zawiera również szczegółowy opis łączenia tych przekształceń w procesie refaktoryzacji oraz sposobów implementowania wzorców w kodzie. Każde z omówionych w książce przekształceń zostało zilustrowane praktycznymi przykładami.

- Podstawowe zasady refaktoryzacji
- Zasady stosowania wzorców projektowych
- Najczęstsze powody wprowadzania wzorców do kodu
- Implementowanie wzorców projektowych
- Zmiany sposobów tworzenia obiektów
- Upraszczanie i uogólnianie kodu

Jeśli chcesz zmodernizować kod swoich aplikacji, w tej książce znajdziesz wszystkie informacje na ten temat

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

6.

Tworzenie obiektów 61

7.

Upraszczanie kodu 117

8.

Uogólnianie kodu 187

9.

Ochrona 257

4

Zapachy kodu

Gdy nauczysz się patrzeć na swoje słowa z krytycznym dystansem, zauważysz, że czytając ten sam fragment po pięć czy sześć razy z rzędu, za każdym razem odkrywasz nowy problem [Barzun, 229].

Refaktoryzację, a więc poprawianie konstrukcji kodu, rozpoczynamy od określenia, który kod wymaga poprawienia. Katalogi refaktoryzacji są tu pomocne, ale nie opisują wszystkich sytuacji. Aby rozpoznać problemy we własnych rozwiązaniach, musimy zawczasu dobrze poznać typowe problemy konstrukcyjne.

Problemy konstrukcyjne mają swoje źródło w kodzie, który jest:

- wielokrotnie powtarzany,
- niejasny,
- skomplikowany.

Tak określone kryteria na pewno ułatwią wyszukiwanie miejsc, w których można wprowadzić ulepszenia. Z drugiej strony, wielu programistów uważa, że taka lista jest zbyt ogólna. Nie wiedzą, jak wykryć duplikacje w kodzie, który nie jest w oczywisty sposób identyczny. Nie są pewni, czy dany kod w jasny sposób informuje o swoim celu. Nie potrafią odróżnić kodu prostego i złożonego.

W rozdziale „Brzydkie zapachy w kodzie" książki *Refactoring* [F] Martin Fowler i Kent Beck dają dodatkowe wskazówki dotyczące identyfikowania problemów konstrukcyjnych. Porównują te problemy do zapachów i wyjaśniają, które przekształcenia lub połączenia przekształceń najskuteczniej likwidują zapach.

Zapachy kodu Fowlera i Becka pojawiają się wszędzie: w metodach, klasach, hierarchiach, pakietach (przestrzeniach nazw, modułach) i całych systemach. Ich nazwy, takie jak Feature Envy (zazdrość o funkcje), Primitive Obsession (obsesja na punkcie wartości prostych albo prymitywna obsesja) czy Speculative Generality (ogólność spekulatywna), to propozycja bogatego i kolorowego słownictwa, które programiści mogą wykorzystać do sprawnej wymiany informacji o problemach konstrukcyjnych.

Uznałem, że warto rozważyć, które z 22 zapachów kodu Fowlera i Becka są związane z przekształceniami omawianymi w tej książce. W trakcie pracy zdefiniowałem pięć kolejnych, które powinny skłonić do przekształceń ukierunkowanych na wzorce. W sumie przekształcenia w tej książce powiązałem z 12 zapachami kodu.

Tabela 4.1 wymienia wszystkie 12 zapachów, wraz z przekształceniami, które mogą pomóc w ich usunięciu. Na kolejnych kilku stronach omówię każdy z zapachów i zalecane refaktoryzacje.

TABELA 4.1.

Zapach[a]	Przekształcenie
Duplicated Code (48) [F]	*Form Template Method* (188) *Introduce Polymorphic Creation with Factory Method* (88) *Chain Constructors* (302) *Replace One/Many Distinctions with Composite* (203) *Extract Composite* (195) *Unify Interfaces with Adapter* (223) *Introduce Null Object* (271)
Long Method (49) [F]	*Compose Method* (118) *Move Accumulation to Collecting Parameter* (280) *Replace Conditional Dispatcher with Command* (177) *Move Accumulation to Visitor* (286) *Replace Conditional Logic with Strategy* (123)
Conditional Complexity (50)	*Replace Conditional Logic with Strategy* (123) *Move Embellishment to Decorator* (136) *Replace State-Altering Conditionals with State* (154) *Introduce Null Object* (271)
Primitive Obsession (50) [F]	*Replace Type Code with Class* (258) *Replace State-Altering Conditionals with State* (154) *Replace Conditional Logic with Strategy* (123) *Replace Implicit Tree with Composite* (165) *Replace Implicit Language with Interpreter* (243) *Move Embellishment to Decorator* (136) *Encapsulate Composite with Builder* (95)
Indecent Exposure (51)	*Encapsulate Classes with Factory* (81)
Solution Sprawl (51)	*Move Creation Knowledge to Factory* (71)
Alternative Classes with Different Interfaces (51) [F]	*Unify Interfaces with Adapter* (223)
Lazy Class (52) [F]	*Inline Singleton* (111)
Large Class (52) [F]	*Replace Conditional Dispatcher with Command* (177) *Replace State-Altering Conditionals with State* (154) *Replace Implicit Language with Interpreter* (243)
Switch Statements (52) [F]	*Replace Conditional Dispatcher with Command* (177) *Move Accumulation to Visitor* (286)
Combinatorial Explosion (53)	*Replace Implicit Language with Interpreter* (243)
Oddball Solution (45)	*Unify Interfaces with Adapter* (223)

[a] Numery stron wskazują stronę książki, na której omawiam zapach. [F] oznacza, że zapach został omówiony przez Martina Fowlera i Kenta Becka w rozdziale „Brzydkie zapachy w kodzie" książki *Refactoring* [F].

Duplicated Code (powtórzenia kodu)

Zduplikowany kod to najbardziej rozpowszechniony i najostrzejszy z zapachów. Bywa wyraźny lub subtelny. Jawna duplikacja to identyczne fragmenty kodu. Powtórzenia ukryte znajdziemy w strukturach lub procedurach przetwarzania, które jedynie zewnętrznie różnią się od siebie.

Oba rodzaje duplikacji, pojawiające się w podklasach pewnej hierarchii, można usunąć przekształceniem *Form Template Method* (188). Jeżeli metoda jest podobnie implementowana w podklasach, a głównym wyróżnikiem jest etap tworzenia obiektu, przekształcenie *Introduce Polymorphic Creation with Factory Method* (88) otworzy drogę do usunięcia duplikacji przy użyciu wzorca Template Method.

Jeżeli kod powtarza się w konstruktorach klasy, można odwołać się do przekształcenia *Chain Constructors* (302).

Jeżeli inny kod przetwarza pojedynczy obiekt, a inny kolekcję, można usunąć duplikację przekształceniem *Replace One/Many Distinctions with Composite* (203).

Jeżeli każda podklasa hierarchii implementuje własny obiekt Composite, kod może okazać się identyczny. Wówczas niezbędne jest przekształcenie *Extract Composite* (195).

Jeżeli przetwarzamy obiekty w różny sposób tylko dlatego, że mają inne interfejsy, przekształcenie *Unify Interfaces with Adapter* (223) zapoczątkuje proces usuwania powtarzającego się kodu logiki przetwarzania.

Jeżeli korzystamy z pewnego kodu warunkowego do obsługi sytuacji, gdy obiekt ma wartość null, i taki kod powtarza się w wielu różnych miejscach, przekształcenie *Introduce Null Object* (271) pozwoli usunąć duplikację i uprościć system.

Long Method (długa metoda)

W swoim opisie tego zapachu Fowler i Beck [F] przytaczają kilka ważkich argumentów przemawiających za wyższością krótkich metod nad długimi. Podstawowym argumentem jest podział logiki. Gdy mamy dwie długie metody, szanse na powtórzenia kodu są duże. Gdy podzielimy je na mniejsze, znajdziemy wiele sposobów na przejrzysty podział logiki między nimi.

Fowler i Beck zwracają też uwagę na to, że małe metody funkcjonują jako naturalny opis kodu. Jeżeli nie jest oczywiste, co robi pewien fragment kodu, wyłączenie go do niewielkiej, odpowiednio nazwanej metody na pewno zwiększy przejrzystość kodu. Systemy zbudowane głównie na podstawie niewielkich metod zazwyczaj łatwiej rozbudowywać i konserwować, bo są przejrzyste i jest w nich niewiele powtórzeń kodu.

Jaki jest optymalny rozmiar takiej niewielkiej metody? Rzekłbym, że do dziesięciu wierszy kodu, przy czym większość metod nie powinna mieć więcej niż pięć. Gdy podążamy tą ścieżką, możemy wprowadzić też kilka metod bardziej rozbudowanych, o ile tylko ich konstrukcja będzie przejrzysta i nie pojawi się duplikacja kodu.

Niektórzy programiści unikają pisania małych metod w obawie przed spadkiem wydajności, który może pojawić się w efekcie wielokrotnego przekazywania wywołań. Podejście nie sprawdza się z kilku powodów. Po pierwsze, dobry projektant nie dąży do przedwczesnej optymalizacji kodu. Po drugie, łączenie kilku niewielkich metod nie wpływa zazwyczaj na wydajność — łatwo to potwierdzić, korzystając z narzędzia do profilowania. Po trzecie, gdy faktycznie pojawiają się problemy z szybkością pracy, można przeprowadzić refaktoryzację ukierunkowaną na wydajność i wcale nie rezygnować z zasady ograniczania rozmiaru metod.

Gdy natrafiam na długą metodę, jednym z moich pierwszych odruchów jest przekształcenie jej do postaci określanej jako Composed Method [Beck, SBPP]. Prowadzi do tego przekształcenie *Compose Method* (118). Jednym z niezbędnych etapów jest zazwyczaj przekształcenie *Extract Method* [F]. Jeżeli kod przekształcany do postaci metody złożonej gromadzi dane w pewnej wspólnej zmiennej, warto rozważyć przekształcenie *Move Accumulation to Collecting Parameter* (280).

Jeżeli metoda jest długa, ponieważ zawiera rozbudowaną instrukcję `switch`, odpowiedzialną za dyspozycję i obsługę żądań, można ją nieco „odchudzić” przekształceniem *Replace Conditional Dispatcher with Command* (177).

Jeżeli instrukcja `switch` służy do gromadzenia danych z wielu klas o różnych interfejsach, można zmniejszyć jej rozmiary, stosując przekształcenie *Move Accumulation to Visitor* (286).

Jeżeli metoda jest długa, ponieważ zawiera wiele wersji pewnego algorytmu i instrukcje warunkowe, określające właściwą wersję w czasie wykonywania, dobrą techniką zmniejszenia jej rozmiarów będzie przekształcenie *Replace Conditional Logic with Strategy* (123).

Conditional Complexity (złożoność warunków)

Logika wyrażeń warunkowych jest niewinna w swojej surowości, o ile tylko pozostaje prosta i nie jest bardziej rozbudowana niż kilka wierszy kodu. Niestety, nie służy jej czas i rozwój. Proste początkowo wyrażenia mogą zamienić się w nieprzenikniony gąszcz, gdy tylko dodamy kilka nowych funkcji.

Jeżeli wyrażenia warunkowe służą do wybierania jednego z wariantów obliczeń, rozważamy przekształcenie *Replace Conditional Logic with Strategy* (123).

Jeżeli wyrażenia warunkowe służą do wybierania pomiędzy różnymi nietypowymi zachowaniami klasy, możemy skorzystać z *Move Embellishment to Decorator* (136).

Jeżeli problem złożoności dotyczy wyrażeń, które decydują o zmianie stanu obiektu, można uprościć kod przekształceniem *Replace State-Altering Conditionals with State* (154).

Wyrażenia warunkowe służą często do rozpatrywania przypadku wartości null. Jeżeli ten sam schemat powtarza się w wielu miejscach aplikacji, można skrócić kod, wprowadzając obiekt null, a więc stosując przekształcenie *Introduce Null Object* (271).

Primitive Obsession (pierwotna obsesja)

Typy pierwotne albo, inaczej, typy proste — liczby całkowite, ciągi, wartości zmiennoprzecinkowe, tablice i inne elementy niskiego poziomu — mają charakter ogólny i są wykorzystywane w wielu różnych aplikacjach. Klasy, w przeciwieństwie do nich, są tworzone odpowiednio do potrzeb i mogą być dowolnie wyspecjalizowane. W wielu przypadkach są one prostszym i bardziej naturalnym modelem rzeczywistości. Dodatkowo, gdy już stworzymy pewną klasę, często odkrywamy, że możemy w niej umieścić kod z innych części systemu.

Fowler i Beck [F] piszą o zapachu *Primitive Obsession*, który pojawia się wtedy, gdy typy proste są stosowane w kodzie nadzwyczaj często. Zjawisko to występuje najczęściej, gdy nie zdecydowaliśmy jeszcze, jakie abstrakcje wysokiego poziomu mogą uczynić kod bardziej prostym i przejrzystym. Przekształcenia przedstawione przez Fowlera zawierają wiele rozwiązań takiego problemu. Wykorzystuję je w tej książce, proponując też inne.

Jeżeli wartość prosta decyduje o wykorzystywanej logice klasy i nie zapewnia zarazem bezpieczeństwa typów (tj. klient może przypisać wartość niebezpieczną lub niewłaściwą), rozważamy przekształcenie *Replace Type Code with Class* (258). Uzyskamy w ten sposób zabezpieczenie typu wartości i możliwość wprowadzania nowych zachowań (na co nie pozwalają typy pierwotne).

Jeżeli o zmianach stanu obiektu decyduje złożona logika z wyrażeniami warunkowymi, oparta na wartościach prostych, można skorzystać z przekształcenia *Replace State-Altering Conditionals with State* (154). Wynikiem będą osobne klasy reprezentujące poszczególne stany i uproszczona logika zmian stanu.

Jeżeli skomplikowana logika z wyrażeniami warunkowymi decyduje o wykorzystywanym algorytmie i ta logika opiera się na wartościach prostych, stosujemy przekształcenie *Replace Conditional Logic with Strategy* (123).

Jeżeli niejawnie tworzymy strukturę drzewiastą, ograniczając się do wartości prostych, takich jak ciąg znakowy, praca z kodem może być utrudniona i podatna na błędy. Może też prowadzić do powtórzeń kodu. Rozwiązaniem jest przekształcenie *Replace Implicit Tree with Composite* (165).

Jeżeli wiele metod klasy zapewnia obsługę licznych kombinacji wartości prostych, możemy mieć do czynienia z niejawnym językiem. Wówczas pomocne jest przekształcenie *Replace Implicit Language with Interpreter* (243).

Jeżeli wartości proste pojawiają się w klasie tylko po to, aby uzupełnić podstawowe funkcje obiektu, można rozważyć przekształcenie *Move Embellishment to Decorator* (136).

Nawet jeżeli utworzyliśmy pewną klasę, jej konstrukcja może być zbyt prosta, aby faktycznie ułatwić budowanie klientów. Tak może być w przypadku skomplikowanego w obsłudze obiektu Composite [DP]. Pomocą może być usprawnienie budowy obiektów uzyskane przez przekształcenie *Encapsulate Composite with Builder* (95).

Indecent Exposure (nieprzyzwoite obnażanie się)

Ten zapach sygnalizuje brak tego, co David Parnas nazwał „ukrywaniem informacji" [Parnas]. Pojawia się, gdy metody lub klasy, które nie powinny być widoczne dla klientów, są dla nich w pełni dostępne. Takie niefrasobliwe ujawnianie kodu powoduje, że klienty dowiadują się o elementach programu, które nie są dla nich ważne lub są ważne tylko pośrednio. Wpływa to na złożoność systemu.

Zapach usuwa przekształcenie *Encapsulate Classes with Factory* (81). Nie każda klasa, z której korzystają klienty, musi być publiczna (nie musi mieć publicznego konstruktora). Do niektórych klas można odwoływać się poprzez ogólniejsze interfejsy. Warunkiem jest właśnie ukrycie konstruktorów klasy i budowanie egzemplarzy za pośrednictwem klasy Factory.

Solution Sprawl (rozrzucanie rozwiązania)

Gdy kod i (lub) dane związane z pewnym zakresem odpowiedzialności są rozrzucone po wielu klasach, czuć zapach Solution Sprawl. Jest on często wynikiem szybkiego dodawania funkcji do systemu, z pominięciem niezbędnego etapu upraszczania i konsolidowania projektu.

Solution Sprawl to brat bliźniak Shotgun Surgery (chirurgia śrutówką), zapachu opisanego przez Fowlera i Becka [F]. Piszą oni, że poczujemy go, gdy okaże się, że wprowadzenie lub zmodyfikowanie pewnej funkcji wymaga zmian w wielu różnych miejscach w kodzie. Oba zapachy to ten sam problem, choć nieco inaczej rozpoznawany. Solution Sprawl pojawia się w trakcie obserwacji i analizy, a Shotgun Surgery — w trakcie pracy z kodem.

Move Creation Knowledge to Factory (71) to przekształcenie, które rozwiązuje problem rozrzuconego kodu tworzenia egzemplarzy.

Alternative Class with Different Interfaces (podobna klasa o innych interfejsach)

Kolejny zapach opisany przez Fowlera i Becka [F] pojawia się, gdy dwie podobne klasy mają różne interfejsy. Jeżeli w projekcie znajdziemy dwie zbliżone klasy, można często przekształcić je tak, aby korzystały ze wspólnego interfejsu.

W pewnych przypadkach nie można bezpośrednio zmienić interfejsu klasy, bo nie można modyfikować jej kodu. Najbardziej typową sytuacją tego rodzaju jest korzystanie z różnego rodzaju bibliotek. Wówczas możemy skorzystać z przekształcenia *Unify Interfaces with Adapter* (223).

Lazy Class (leniwa klasa)

Fowler i Beck piszą o tym zapachu: „Klasa, która nie zarabia na siebie, powinna zostać usunięta" [F, 83]. Jakże często można spotkać Singleton [DP], który „nie zarabia na siebie". W rzeczywistości schemat Singleton może być dodatkowym kosztem, prowadzącym do pogłębienia uzależnienia od danych udostępnianych globalnie. *Inline Singleton* (111) to szybka i humanitarna metoda eliminacji singletonów.

Large Class (duża klasa)

Fowler i Beck [F] zauważają, że obecność zbyt wielu zmiennych egzemplarza sygnalizuje zazwyczaj, że klasa ma zbyt wiele obowiązków. Duże klasy mają zazwyczaj zbyt wiele zakresów odpowiedzialności. *Extract Class* [F] i *Extract Subclass* [F] to podstawowe przekształcenia usuwające ten zapach i prowadzące do przeniesienia odpowiedzialności. Są one elementem opisywanych w tej książce złożonych przekształceń ukierunkowanych na wzorce. Ich celem jest zmniejszenie rozmiaru klasy.

Replace Conditional Dispatcher with Command (177) to przekształcenie wyłączające zachowania do osobnych klas Command [DP]. Jeżeli klasa realizuje wiele zachowań w odpowiedzi na bardzo zróżnicowane żądania, jej rozmiar można w ten sposób zmniejszyć kilkakrotnie.

Replace State-Altering Conditionals with State (154) pozwala zmniejszyć dużą klasę ze znaczną ilością kodu zmieniającego jej stan poprzez wprowadzenie delegacji do rodziny klas State [DP].

Przekształcenie *Replace Implicit Language with Interpreter* (243) zmniejsza rozmiar klasy w wyniku przekształcenia pełnego powtórzeń kodu, który w istocie emuluje pewien język, w klasę Interpreter [DP].

Switch Statements (instrukcje switch)

Instrukcje `switch` (i równoważne struktury `if...elseif...elseif...`) nie są z gruntu złe. Są szkodliwe tylko wtedy, gdy sprawiają, że kod jest nadmiernie skomplikowany lub sztywny. Wówczas powinniśmy dążyć do przekształcenia instrukcji warunkowych w konstrukcję opartą na obiektach.

Przekształcenie *Replace Conditional Dispatcher with Command* (177) to opis rozbijania dużej instrukcji `switch` na kolekcję obiektów Command [DP], które mogą być wywoływane bez korzystania z wyrażeń warunkowych.

Przekształcenie *Move Accumulation to Visitor* (286) to przykład programu, w którym instrukcje `switch` służą do pobierania danych z obiektów, które różnią się interfejsami. Wprowadzenie wzorca Visitor [DP] umożliwia usunięcie instrukcji warunkowych i zapewnia większą ogólność kodu.

Combinatorial Explosion (eksplozja kombinatoryczna)

Ten zapach to subtelny przypadek duplikacji kodu. Pojawia się, gdy w wielu miejscach kodu implementowane są te same czynności, operujące na różnych rodzajach lub ilościach danych albo obiektów.

Przykładem może być zbiór metod do generowania zapytań. Każda z nich wykonuje zapytanie oparte na pewnych warunkach i danych. Aby zapewnić obsługę większej liczby wyspecjalizowanych zapytań, dodajemy nowe metody. W efekcie uzyskujemy eksplozję metod obsługujących najróżniejsze operacje dostępu do bazy danych — niejawny język zapytań. Metody te, jak i cały zapach, można usunąć przekształceniem *Replace Implicit Language with Interpreter* (243).

Oddball Solution (osobliwe rozwiązanie)

Gdy pewien problem jest rozwiązywany w systemie w określony sposób, ale w pewnym miejscu pojawia się inne podejście do tej samej kwestii, jedno z nich należy uznać za niespójne. Ten zapach sygnalizuje zazwyczaj subtelne powtórzenia kodu.

Usuwanie tego rodzaju duplikacji rozpoczynamy od wybrania jednego z rozwiązań. W pewnych przypadkach to, które jest używane rzadziej, może być w istocie lepsze. Można wówczas przeprowadzić przekształcenie *Substitute Algorithm* [F], prowadzące do ujednolicenia rozwiązania w całym systemie. Gdy rozwiązanie jest stosowane spójnie, pojawia się możliwość zgromadzenia przetwarzania w jednym miejscu.

Zapach Oddball Solution pojawia się często tam, gdzie istnieje pewna uznana metoda wywoływania zbioru klas, których interfejsy nie są jednolite. Można wówczas rozważyć przekształcenia *Unify Interfaces with Adapter* (223) i osłonić klasy wspólnym interfejsem. Wówczas odkryjemy nowe możliwości usunięcia duplikacji kodu.