


Zaawansowane projektowanie obiektowe

Wprowadzenie do refaktoryzacji

Prowadzący: Bartosz Walter



UCZELNIA
ONLINE

Zaawansowane projektowanie obiektowe

Uczelnia ONLINE

Motto






*„Nawet idiota potrafi napisać kod zrozumiały dla komputera.
Prawdziwi programiści tworzą kod zrozumiały dla ludzi.”*

Martin Fowler

Wprowadzenie do refaktoryzacji (2)

Mottem wykładu są słowa autora popularnej książki dotyczącej refaktoryzacji, M. Fowlera. Słowa te są szczególnie ważne w kontekście tzw. metodyk zwinnych (ang. *agile methodologies*), promujących wyższość zrozumiałego kodu nad dokumentacją (szczegóły dotyczące wartości głoszonych przez metodyki zwinne można znaleźć pod adresem <http://agilemanifesto.org>). Kładą one nacisk na okresową restrukturyzację kodu w celu poprawienia jego czytelności. Refaktoryzacja jest jedną z podstawowych praktyk Programowania Ekstremalnego, najpopularniejszej spośród tych metodyk.

Zaawansowane projektowanie obiektowe


Plan wykładu


- Wprowadzenie
- Koszt refaktoryzacji
- Poprawność i jej weryfikacja
- Przykre zapachy w kodzie programu
- Metody identyfikacji przykrych zapachów

Wprowadzenie do refaktoryzacji (3)

Wykład ten jest wprowadzeniem i stanowi pierwszy z czterech poświęconych refaktoryzacji. Omówione będą rozmaite definicje refaktoryzacji, czynniki wpływające na koszt jej stosowania oraz problem poprawności przekształceń i jej weryfikacji. Większą część wykładu zajmie omówienie zagadnienia tzw. przykrych zapachów w kodzie programu, ich rodzajów oraz metod identyfikacji.

Zaawansowane projektowanie obiektowe

Plan wykładu



Wprowadzenie

Wprowadzenie do refaktoryzacji (4)

Wykład rozpoczniemy od wprowadzenia, które przybliży motywację stosowania refaktoryzacji, jej ekonomikę oraz ogólne założenia.



- Wysoki koszt pielęgnacji oprogramowania
 - Yourdon: do 80% kosztu użytkowania
 - Boehm: wytworzenie linii kodu: \$30, pielęgnacja: \$4000
- Naturalny wzrost złożoności i entropii oprogramowania
- Prawa Lehmana: konieczna ciągła restrukturyzacja

Refaktoryzacja jest jedną z technik pielęgnacji oprogramowania.

Jak wskazują badania, pielęgnacja pochłania nawet do 80% całkowitych kosztów związanych z oprogramowaniem. Znamienny jest też przykład podany przez Boehma: stworzenie linii kodu w oprogramowaniu dla Boeinga kosztowało ponad stukrotnie mniej niż jej pielęgnacja do końca życia produktu.

Ta i podobna obserwacje posłużyły M. Lehmanowi do sformułowania praw dotyczących ewolucji oprogramowania. Mówią one, że oprogramowanie w trakcie ewolucji staje się coraz bardziej złożone, a jego struktura w coraz mniejszym stopniu odpowiada wymaganiom. Jedynym sposobem przeciwdziałania temu zjawisku jest ciągła restrukturyzacja, która przywraca pierwotną prostotę projektu.


Zaawansowane projektowanie obiektowe

Intuicyjna definicja

UCZELNIA ONLINE

Refaktoryzacja to:

- zmiana wewnętrznej struktury kodu programu
- która zwiększa jego czytelność i obniża koszt pielęgnacji
- **ale nie zmienia jego obserwowalnego zachowania**



Opdyke (1991)

Wprowadzenie do refaktoryzacji (6)

Pierwsze nieformalne określenie refaktoryzacji podał w swojej rozprawie doktorskiej W. Opdyke w 1991 roku. Podkreślała ona trzy elementy:

- jest to transformacja kodu źródłowego programu (a więc czytelnego dla programisty)
- która poprawia jego pielęgnowalność poprzez uproszczenie struktury i dostosowanie jej do zmieniających się wymagań.
- ale nie zmienia jego funkcjonalności z punktu widzenia użytkownika tego oprogramowania

Szczególnie ta ostatnia własność – najważniejsza, ale i najtrudniejsza do zapewnienia – budziła wiele dyskusji. Dotyczyły one kwestii, czy program miał zachowywać się identycznie we wszystkich możliwych przypadkach, także wykraczających poza jego specyfikację (np. możliwa była inna odpowiedź systemu po przekroczeniu założonego obciążenia albo uruchomieniu w odmiennym środowisku), czy też wystarczy, jeżeli zmiana nie będzie zauważalna w typowym użyciu.

Narzucenie bardziej rygorystycznej interpretacji powodowało konieczność przeprowadzania dowodu poprawności, co w praktyce wykluczało efektywne refaktoryzowanie programów.



Refaktoryzacja jest uporządkowaną parą

$$R = (pre; T)$$

gdzie

- *pre* jest warunkiem wstępnym, który program musi spełniać, aby przekształcenie refaktoryzacyjne było poprawne
- *T* jest przekształceniem programu

Roberts (1998)

Wprowadzenie do refaktoryzacji (7)

Bardziej formalną definicję zaproponował D. Roberts w 1998 roku, opisując przekształcenie refaktoryzacyjne jako parę elementów:

- zbioru warunków wstępnych, jakie program musi spełniać przed wykonaniem zmiany, aby była ona poprawna
- i opisu samego przekształcenia.

Warto zwrócić uwagę, że definicja ta zakłada, że poprawność można stwierdzić przed wykonaniem modyfikacji, tzn. możliwe jest przewidzenie wszystkich jej skutków z góry, bez konieczności jej przeprowadzania. Oczywiście, takie założenie nie zawsze jest uzasadnione.



Wyłączenie metody

```
void scalarProduct(String[] params) {
    int[] x = prepareX(params);
    int[] y = prepareY(params);
    int[] product = computeXY(x, y);
    // ...
    for (i = 0; i < x.length; i++) {
        out.println("X = " + x[i]);
        out.println("Y = " + y[i]);
        out.println("X * Y = " + product[i]);
    }
}
```

```
void scalarProduct(String[] params) {
    int[] x = prepareX(params);
    int[] y = prepareY(params);
    int[] product = computeXY(x, y);
    // ...
    printScalarProduct(x, y, product);
}
```

```
void printScalarProduct(int[] x, int[] y,
    int[] product) {
    for (i = 0; i < x.length; i++) {
        out.println("X = " + x[i]);
        out.println("Y = " + y[i]);
        out.println("X * Y = " + product[i]);
    }
}
```

Jakie są warunki wstępne poprawności tego przekształcenia?

Ilustracją dla definicji niech będzie jedno z najpopularniejszych przekształceń: Extract Method (wyłączenie metody). Polega ono na podziale złożonej metody na mniejsze poprzez wydzielenie z niej fragmentu kodu i przeniesienie go do nowej metody. Zmienne, których definicje znajdują się poza zakresem tego fragmentu, są przekazywane do tej metody jako parametry.

W tym przypadku metoda *scalarProduct()* wykonuje dwie czynności: oblicza wartości zmiennych tablicowych *x*, *y* i *product*, a następnie wyświetla je na ekranie. Kod służący do wyświetlania wyników jest wyłączony do nowej metody *printScalarProduct()*, która następnie jest wywoływana z oryginalnej metody *scalarProduct()*. W ten sposób program z funkcjonalnego punktu widzenia zachowuje się tak samo: wywołanie metody *scalarProduct()* powoduje wykonanie odpowiednich obliczeń oraz wyświetlenie ich wyników na ekranie.

Aby spróbować określić warunki poprawności tego przekształcenia, warto zastanowić się, (1) w jakich okolicznościach stworzenie nowej metody będzie niemożliwe, (2) kiedy nie można jej przekazać wymaganych parametrów, (3) kiedy niemożliwe będzie zwrócenie wartości z tej metody. Proszę je sformułować.



Refaktoryzacja jest uporządkowaną trójką

$$R = (pre; T; P)$$

gdzie

- *pre* jest asercją, która musi być spełniona w programie, aby przekształcenie *R* było poprawne,
- *T* jest transformacją programu, a
- *P* jest funkcją przekształcającą warunki wstępne na warunki końcowe przy każdym zastosowaniu transformacji *T*. *P* określa warunki końcowe, które są prawdziwe po zastosowaniu transformacji *T*.

Roberts (1998)

Wprowadzenie do refaktoryzacji (9)

Roberts zaobserwował, że proste przekształcenia zdefiniowane przez Opdyke'a nie są stosowane w praktyce, ponieważ wprowadzają zbyt drobne zmiany. Konieczne jest zatem stosowanie większych struktur: łańcuchów przekształceń. Jednak dowodzenie poprawności całych łańcuchów było zbyt trudne, ponieważ wymagało osobnego dowodu dla każdego przekształcenia z osobna, a co za tym idzie – obliczenia warunków wstępnych będących wynikiem wykonania wcześniejszych przekształceń w łańcuchu.

Dlatego Roberts zaproponował, aby rozszerzyć definicję każdego przekształcenia o warunki końcowe, które są prawdziwe po jej wykonaniu, np. po utworzeniu metody warunkiem końcowym jest jej istnienie. Ta drobna zmiana pozwoliła jednak znacznie łatwiej wykazywać poprawność łańcuchów przekształceń, wykorzystując znane już informacje pochodzące z analizy wcześniejszych przekształceń prostych.


Zaawansowane projektowanie obiektowe

Przykład: Rename Method

UCZELNIA ONLINE

Zmiana nazwy metody

```
class KlasaA
{
    void metodaA () {
        // kod
    }
}
```



```
class KlasaA
{
    void metodaB () {
        // kod
    }
}
```

Jakie są warunki wstępne i końcowe tego przekształcenia?

Wprowadzenie do refaktoryzacji (10)

Tym razem definicja ta zostanie krótko omówiona na przykładzie refaktoryzacji Rename Method (zmiana nazwy metody). Po spełnieniu warunków wstępnych związanych z tym przekształceniem i jego wykonaniu pojawiają się inne warunki, które można wykorzystać przy wprowadzaniu dalszych zmian.

Warunki wstępne i końcowe tego przekształcenia dotyczą – podobnie jak w przypadku Extract Method – możliwości utworzenia nowej metody o podanej sygnaturze. Proszę spróbować je wymienić.

Zaawansowane projektowanie obiektowe

Plan wykładu



Koszt refaktoryzacji

Wprowadzenie do refaktoryzacji (11)

Przejdziemy teraz do omówienia kosztu związanego ze stosowaniem refaktoryzacji.



Refaktoryzacja nie zwiększa funkcjonalności programu, ale kosztuje

- identyfikacja problemu
- przekształcenie kodu
- weryfikacja poprawności

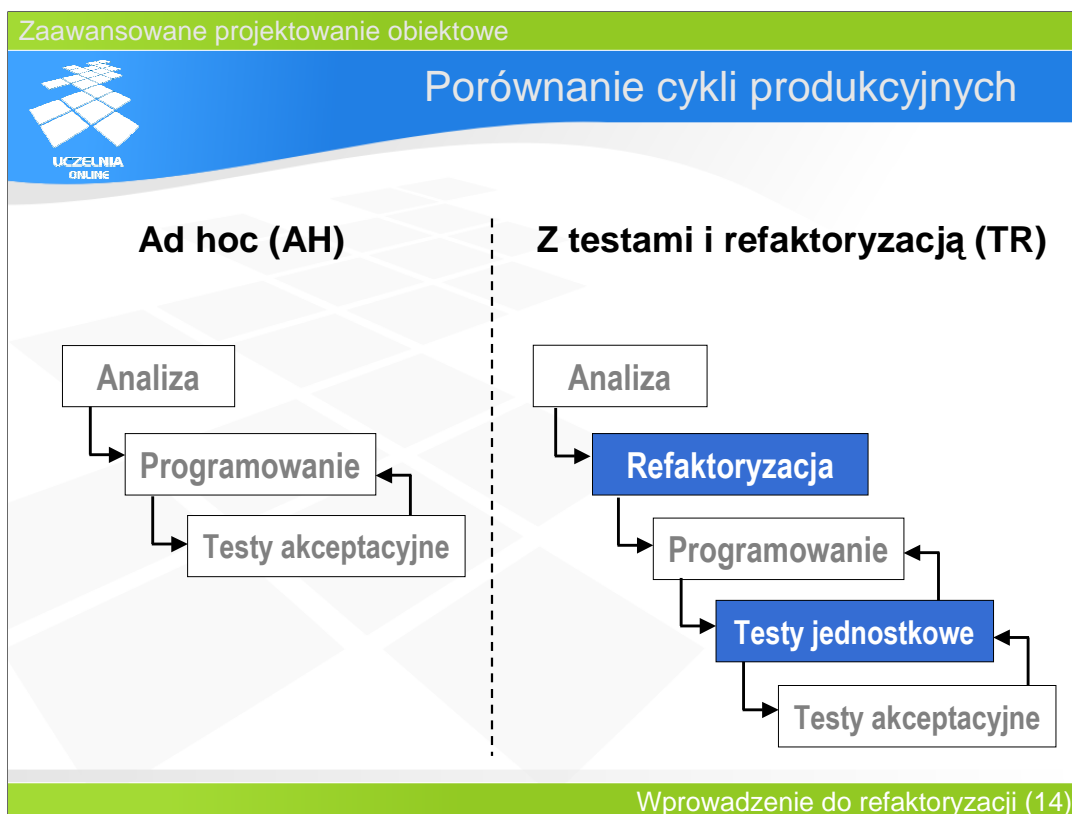
Koszt zależy od:

- własności języka programowania
- wsparcia ze strony narzędzi CASE
- natury wykonywanego przekształcenia
- liczby i jakości posiadanych testów

Refaktoryzacja jest techniką niechętnie stosowaną przez osoby odpowiedzialne za harmonogram prac i zarządzające projektem. Z refaktoryzacją związany jest bowiem dodatkowy nakład pracy (koszt), który nie powoduje wzrostu funkcjonalności systemu. Dlatego wskazane jest jego ograniczenie poprzez automatyzację lub częściową automatyzację niektórych czynności: identyfikacji obszarów kodu wymagających refaktoryzacji, samego wykonania przekształcenia, a na końcu weryfikacji jego poprawności. Nakład ten zależy od środowiska, w którym dokonywana jest refaktoryzacja: języka programowania, narzędzi, a także samego przekształcenia oraz istnienia testów jednostkowych (JUnit), które ułatwiają weryfikację poprawności.



W celu określenia wpływu refaktoryzacji na koszt produkcji oprogramowania (rozumianego jako czas jego tworzenia), przeprowadzono małej skali eksperyment. Dwie grupy osób, pracujących w nieco odmienny sposób, tworzyło ten sam system. Każdy pracował indywidualnie zgodnie z jednym z dwóch modeli: AdHoc oraz Testy & Refaktoryzacja. System był budowany w sposób przyrostowy: składało się na niego 4 przyrosty, z czego przyrost zerowy był realizowany poza warunkami kontrolowanymi i służył stworzeniu szkieletu aplikacji. Następnie, w odstępach tygodniowych, implementowane były kolejne funkcje: 2 w przyroście I, 2 w przyroście II oraz 1 w przyroście III.



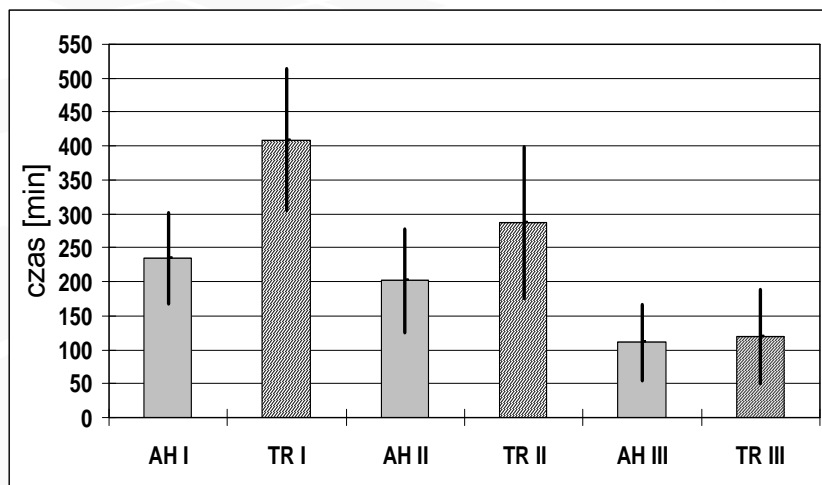
Grupa AdHoc (AH) pracowała w uproszczonym modelu spiralnym, zaczynając od analizy zadania, następnie tworząc kod programu i uruchamiając testy akceptacyjne. Niepowodzenie któregośkolwiek z testów powodowało powrót do programowania.

W porównaniu z tym cyklem model TR dodatkowo był rozszerzony o dwie fazy: refaktoryzację i testowanie jednostkowe. Refaktoryzacja następowała po fazie analizy i miała na celu właściwe wkomponowanie nowych funkcji do istniejącej struktury programu, przede wszystkim poprzez restrukturyzację. Testy jednostkowe, tworzone podczas programowania i po jego zakończeniu, służyły efektywnej refaktoryzacji oraz pozwalały programiście na bieżąco kontrolować jakość kodu.

Tak więc model AH składał się z trzech faz, natomiast model TR – z pięciu.

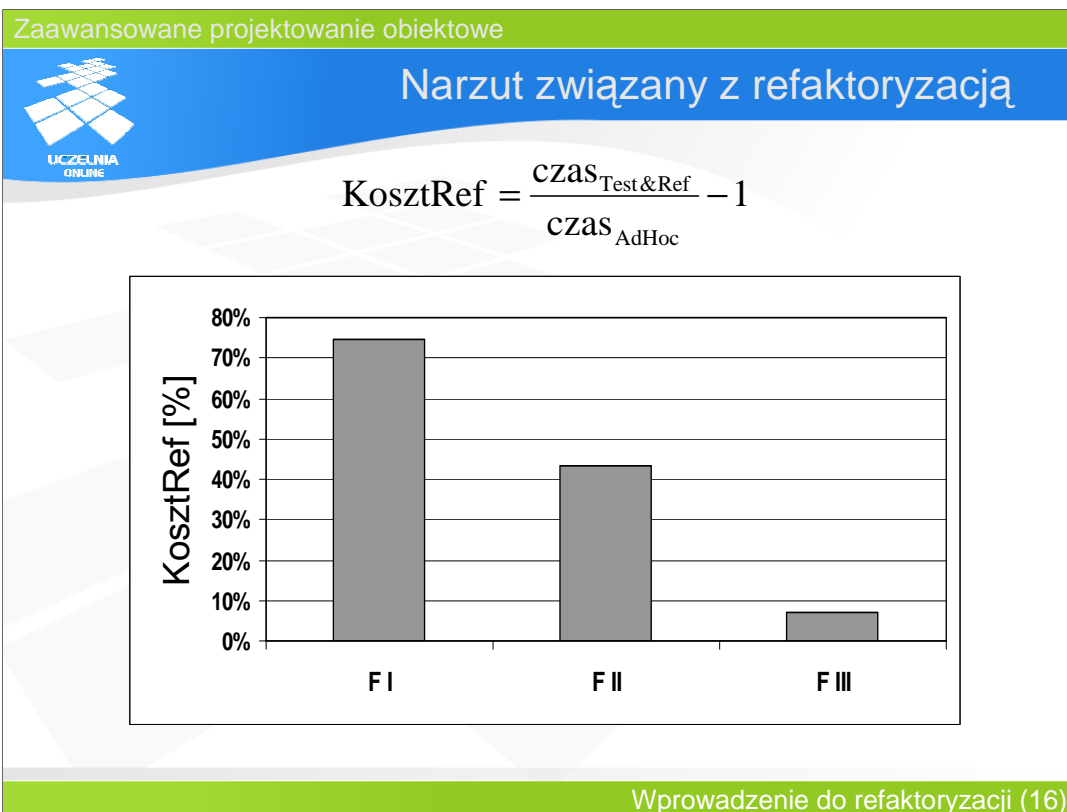


Pracochłonność każdego przyrostu



Wprowadzenie do refaktoryzacji (15)

Wykres przedstawia średni czas oraz jego odchylenie standardowe w obu grupach (AdHoc – AH i Testy&Refaktoryzacja – TR), dla trzech faz rozwoju produktu. Warto zwrócić uwagę na dość dużą wartość odchylenia standardowego we wszystkich grupach, a także gorszy wynik osiągnięty przez grupę TR we wszystkich fazach rozwoju produktu.



Analizując wyniki eksperymentu, warto zauważyć, że zastosowanie refaktoryzacji nawet w systemie o tak ograniczonej skali i niewielkiej liczbie przyrostów wprowadziło zwiększenie kosztu wytworzenia programu, ale w kolejnych fazach był on coraz mniejszy – od 74% w pierwszej fazie do 8% w trzeciej.

Wyniki tego eksperymentu, mimo stwierdzonej statystycznej istotności, mogą być kwestionowane z uwagi na akademickie środowisko jego przeprowadzenia, niewielki rozmiar i złożoność realizowanego oprogramowania, ewentualny wpływ czynników zewnętrznych, a przede wszystkim niewielką liczbę badanej grupy programistów. Wyniki wskazują jednak na pewien trend i mogą stać się argumentem nad włączeniem refaktoryzacji kodu do praktyki produkcji oprogramowania.

Zaawansowane projektowanie obiektowe

UCZELNIA ONLINE

Kiedy refaktoryzacja się opłaca?

- Do trzech razy sztuka
- Przed implementacją nowej funkcji
- Identyfikacja obszaru zmienności
- Regularne przeglądy

- Stały zakres prac
- Projekty jednorazowe
- Przedwcześnie zatwierdzone interfejsy
- Niestabilny kod
- Przy bliskich terminach


*Niedokończona refaktoryzacja przypomina dług.
Można z nim żyć, ale jest to kosztowne.*

Ward Cunningham

Wprowadzenie do refaktoryzacji (17)

Od początku wokół refaktoryzacji (podobnie jak wokół szerzej rozumianej pielęgnacji) toczy się dyskusja dotycząca opłacalności tego typu działalności. Większość modeli kosztowych uwzględnia jedynie koszt związany z wytworzeniem oprogramowania, pomijając wszelkie inne czynniki. W ten sposób są one traktowane jako dodatkowy narzut, co ma szczególnie negatywne znaczenie w przypadku refaktoryzacji, która nie wprowadza nowej funkcjonalności, a jedynie zmienia wewnętrzną strukturę programu.

Zaawansowane projektowanie obiektowe

Plan wykładu


Poprawność przekształceń refaktoryzacyjnych

Wprowadzenie do refaktoryzacji (18)

W tej części wykładu zajmiemy się kwestią poprawności przekształceń refaktoryzacyjnych, która – jak już wspomnieliśmy – jest jej najważniejszą, ale i najtrudniejszą do zapewnienia właściwością.

Zaawansowane projektowanie obiektowe

Przykład: Inline Temp



```


i = 0;
...
tab[0] = ++i;
tab[1] = ++i;
tab[2] = ++i;
...

```

1

2

3



```

i = 0;
x = i++;
tab[0] = x;
tab[1] = x;
tab[2] = x;
...

```

1

1

1

Wprowadzenie do refaktoryzacji (19)


Zmiana struktury programu może stanowić dla niego ogromne zagrożenie, jeżeli po jej zakończeniu będzie on działał inaczej. Dlatego najistotniejszym (i najtrudniejszym do spełnienia) warunkiem skutecznej i efektywnej refaktoryzacji jest zapewnienie, że nie wprowadzi ona do oprogramowania zmian w jego zachowaniu.

Rozpatrując prosty przykład przekształcenia polegającego na rozwinięciu zmiennej tymczasowej, łatwo zauważyć, że może ono spowodować nieprawidłowe zachowanie programu. Występująca w pierwotnej wersji wielokrotnie powtarzająca się instrukcja `++i`, po wyłączeniu do zmiennej tymczasowej, jest wykonywana tylko raz. Jednak w efekcie wartości pól tablicy *tab* będą różne od oczekiwanych i pierwotnych.

Zatem nawet proste przekształcenie, intuicyjne i zrozumiałe, może stanowić problem. Warto zastanowić się, jaki element jest jego faktyczną przyczyną, skoro wyłączenie wyrażenia postaci np. `i+1` nie spowoduje zmiany zachowania programu..

Jest nią efekt uboczny powodowany przez wyłączaną instrukcję: zmiana wartości zmiennej *i* wewnątrz wyłączanego wyrażenia. Zatem to przekształcenie jest poprawne tylko wówczas, gdy wyrażenie wyłączane do zmiennej nie zmienia stanu systemu, w szczególności nie modyfikuje wartości żadnej ze zmiennych.

Zaawansowane projektowanie obiektowe

Predykat *noSideEffectsP*

Wejścia

- Program odwołujący się do zmiennej *Var* o wartości początkowej 1
- Funkcja *F* potencjalnie modyfikująca wartość *Var*

Problem 1

- Czy wywołanie funkcji *F* powoduje efekt uboczny w postaci zmiany wartości zmiennej *Var*?


Lemat 1

- Problem 1 (*braku efektów ubocznych*) jest nierozstrzygalny

Wprowadzenie do refaktoryzacji (20)

Analityczne stwierdzenie, czy wykonanie wskazanego fragmentu kodu (pewnej funkcji) nie powoduje zmiany zachowania programu (modyfikacji wybranej zmiennej *Var*), okazuje się nierozstrzygalne. Spełnienie takiego warunku jest konieczne w przypadku wielu przekształceń refaktoryzacyjnych, dlatego stwierdzenie ich poprawności w dowolnym przypadku okazuje się niemożliwe. Ten przykład pokazuje, że analiza statyczna użyta jako metoda weryfikacji poprawności jest zbyt słabym narzędziem już w przypadku bardzo prostych warunków wstępnych.

Zaawansowane projektowanie obiektowe

Predykat *noSideEffectsP*

Wejścia

- Program odwołujący się do zmiennej *Var* o wartości początkowej 1
- Funkcja *F* potencjalnie modyfikująca wartość *Var*

Problem 2

- Czy istnieje zbiór wejść, który powoduje zmianę wartości zmiennej *Var*?

Lemat 2

- Problem 2 (*zmodyfikowany braku efektów ubocznych*) jest NP-zupełny.


Wprowadzenie do refaktoryzacji (21)

Po ograniczeniu tego problemu do stwierdzenia, czy istnieje taki zbiór wejść, który może spowodować zmianę stanu systemu (czyli znów modyfikację wybranej zmiennej *Var*), staje się on wprawdzie rozstrzygalny, ale jego złożoność obliczeniowa (jest NP-zupełny) nadal sytuuje go wśród problemów bardzo trudnych.

Zaawansowane projektowanie obiektowe

UCZELNIA ONLINE

Poprawność przekształceń

PROSTE	TRUDNE
<ul style="list-style-type: none">• Zautomatyzowana weryfikacja• Obecne w wielu środowiskach IDEs	<ul style="list-style-type: none">• Weryfikacja wymaga testowania• Testy muszą zostać stworzone ręcznie
	

Wprowadzenie do refaktoryzacji (22)

W praktyce istnieją dwie metody weryfikacji poprawności przekształceń:

- analityczna, wykorzystująca informacje statyczne, nie wymagające uruchamiania programu, jednak dająca w zamian dowód poprawności, i
- dynamiczna, która wymaga także analizy dynamicznej, zwykle realizowanej poprzez testowanie.

Przekształcenia, z punktu widzenia złożoności problemu weryfikacji ich poprawności, również dzielą się na dwie analogiczne grupy: przekształceń prostych oraz przekształceń złożonych. Te pierwsze wymagają jedynie analizy statycznej, natomiast w przypadku drugich konieczne jest wykonanie testów jednostkowych. Przekształceń prostych jest jednak znacznie mniej, dlatego w praktyce niemożliwe jest uniknięcie stosowania przekształceń należących do drugiej kategorii.

Zaawansowane projektowanie obiektowe

Przekształcenia proste

- Weryfikowane poprzez statyczną analizę kodu
- Można dowieść ich poprawności

warunki początkowe

warunki końcowe

Wprowadzenie do refaktoryzacji (23)

Przekształcenia proste wykorzystują jedynie statyczną analizę kodu, tzn. ich warunki początkowe można sprawdzić jedynie na podstawie znajomości kodu źródłowego i jego właściwości. Na podstawie warunków początkowych i natury przekształcenia można wskazać warunki końcowe, jakie zaistnieją po wprowadzeniu zmiany.

Zaawansowane projektowanie obiektowe

Przekształcenia trudne

- Nie można dowieść ich poprawności analitycznie
- Wymagają testów jednostkowych

warunki początkowe


warunki końcowe

Wprowadzenie do refaktoryzacji (24)

W przypadku przekształceń trudnych analiza statyczna nie jest wystarczającym narzędziem. Niektóre warunki poprawności tych przekształceń można zweryfikować jedynie poprzez analizę dynamiczną, np. wykorzystanie testów jednostkowych. Testy sprawdzają wówczas, czy asercje istotne z punktu widzenia przekształcenia są zachowane.

Należy jednak zauważyć, że testy nie pozwalają na dowiedzenie poprawności przekształcenia; przeciwnie – niepowodzenie wykonania testu wskazuje na niepoprawność, natomiast poprawności w ten sposób nie można pokazać. Dlatego weryfikacja poprawności jest realizowana dla tej kategorii przekształceń w sposób niepełny i nie dający pewności, że transformacja została przeprowadzona poprawnie.

Zaawansowane projektowanie obiektowe

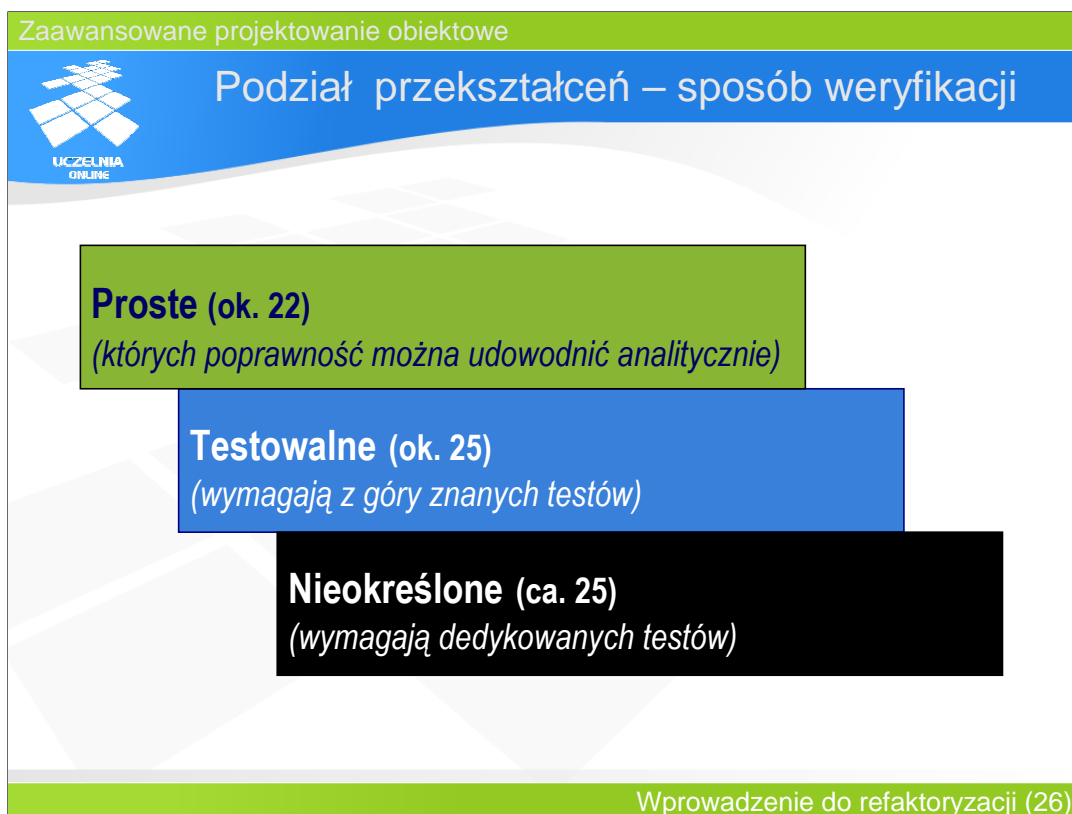
 Podział przekształceń – sposób weryfikacji

Proste (ok. 22)
(których poprawność można udowodnić analitycznie)

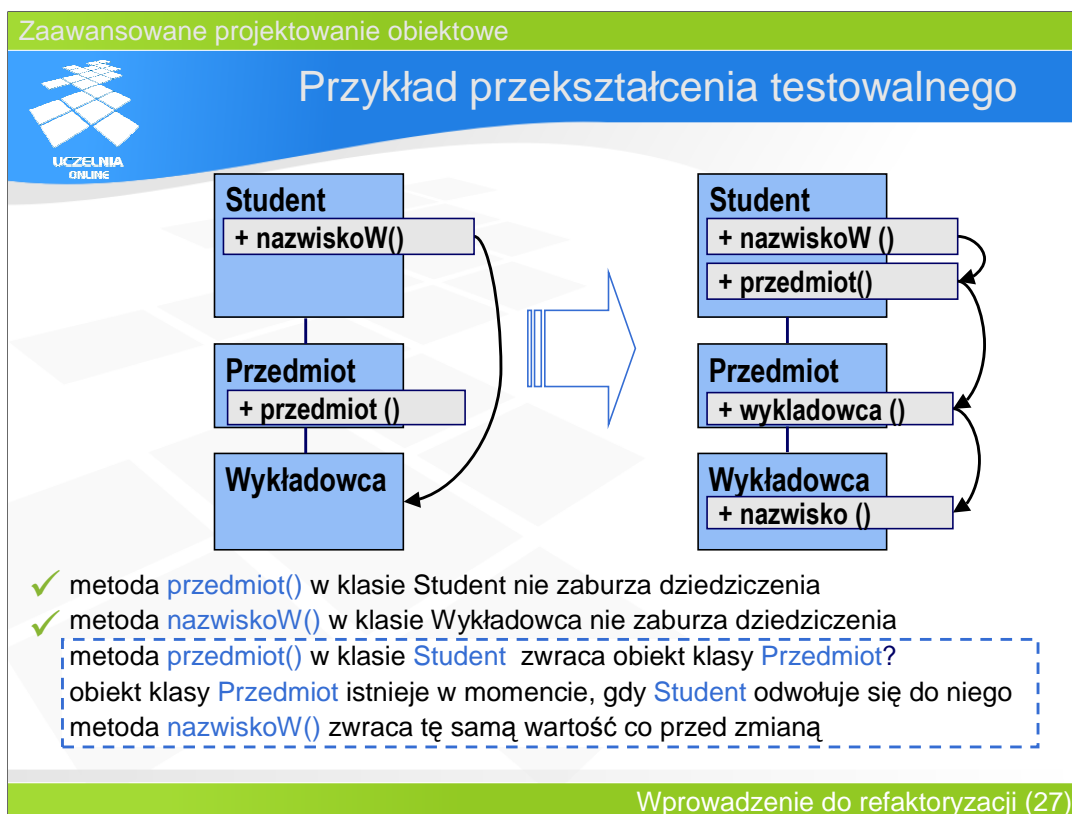
Trudne (ok.50)
(wymagają testów jednostkowych)

Wprowadzenie do refaktoryzacji (25)

Wśród przekształceń z katalogu Fowlera niecała jedna trzecia to przekształcenia proste. Znacznie większa grupa to przekształcenia trudne.




Jednak wśród przekształceń trudnych także można wyróżnić dwie podgrupy: przekształceń testowalnych i nieokreślonych. Szczególnie interesująca jest druga kategoria: należące do niej przekształcenia wprawdzie wymagają testowania, jednak można dość precyzyjnie wskazać niezmienniki, które powinny być zweryfikowane (lecz, oczywiście, nie dowiedzione!) za pomocą testów. Zatem możliwe jest zapewnienie programiście wsparcia co do rodzaju i sposobu tworzenia testów, a nawet częściowe zautomatyzowanie tego procesu. Kategoria przekształceń nieokreślonych zawiera przekształcenia, które wymagają testów, ale nie można wskazać ich natury i sposobu realizacji. Do tej kategorii należą przekształcenia tradycyjnie trudne, i w ich przypadku pracowitość nadal jest bardzo wysoka.



Przykładem przekształcenia należącego do drugiej kategorii jest Move Method. Przesunięcie metody `przedmiot()` z klasy `Przedmiot` do klasy `Student` wymaga sprawdzenia kilku warunków, z których dwa (zaznaczone "ptaszkami") można zweryfikować bez konieczności uruchamiania kodu. Pozostałe wymagają wprowadzić testowania, jednak jest możliwe wskazanie, jak powinny wyglądać takie testy, i opracowanie dla nich np. generycznych szablonów związanych z samym przekształceniem. Innymi słowy, wygenerowanie testów dla innych klas wymagałoby jedynie ukonkretnienia szablonów testowych nowymi parametrami.

Zaawansowane projektowanie obiektowe

Plan wykładu

Przykre zapachy w kodzie programów


Wprowadzenie do refaktoryzacji (28)

W tej części przedstawiony zostanie pojęcie przykrego zapachu oraz ich katalog podany przez Martina Fowlera.

Zaawansowane projektowanie obiektowe

Przykre zapachy w kodzie programów

UCZELNIA
ONLINE



**Jeżeli brzydko pachnie,
należy zmienić**


*filozofia babci Kenta Becka
dotycząca pielęgnacji niemowląt*

Wprowadzenie do refaktoryzacji (29)

O ile metryki są ilościowym wskaźnikiem pozwalającym ocenić, czy oprogramowanie wymaga refaktoryzacji, czy nie, o tyle przykre zapachy reprezentują informację jakościową. Pojęcie przykrego zapachu – jak wiele innych w obszarze zwinnych metodyk – ma anegdotyczną genezę: wg Fowlera, nazwę tę zaproponował K. Beck, "odurzony" zapachami związanymi z pielęgnacją swojej nowonarodzonej córki. Istota przykrego zapachu mieści się właśnie w tym cytacie: jeżeli coś brzydko pachnie, to należy to zmienić. Pojęcie to jest umyślnie zdefiniowane bardzo ogólnie, aby można nim było objąć najróżniejsze problemy związane ze strukturą kodu. W praktyce lista przykrych zapachów obejmuje właśnie wiele niespokrewnionych (przynajmniej bezpośrednio) problemów.

Zaawansowane projektowanie obiektowe

Duplicated Code



Nazwa
Zduplikowany kod

Objawy
Identyczny lub podobny kod znajduje się w wielu miejscach systemu

Rozwiązanie

- w jednej klasie: wyłącz wspólne fragmenty do nowej metody (*Extract Method*)
- w klasach o wspólnej nadklasie: wyłącz wspólne części (*Extract Method*), a następnie przesuń ją do nadklasy (*Pull-up the Method*)
- w klasach niezwiązanych: wyłącz wspólne części do nowej klasy (*Extract Class*) i deleguj do nich wywołania

Wprowadzenie do refaktoryzacji (30)

Najczęściej spotykanym problemem, który dotyczy niemal każdego systemu, jest duplikacja kodu. Badania wskazują, że średnio ok. 20-30% kodu to dokładne lub przybliżone duplikaty. Istotą duplikatów jest problem ze śledzeniem zmian: zwykle poprawki i aktualizacje są stosowane jedynie do jednej kopii, natomiast pozostałe – w zamyśle programisty zachowujące się identycznie jak pierwsza – pozostają niezmienione. To rodzi sporo trudnych do zlokalizowania błędów.

Istotnym problemem jest wykrywanie duplikatów. Najprostsze metody posługują się jedynie zwykłym dopasowaniem tekstu, jednak ich skuteczność jest ograniczona. Bardziej zaawansowane dopuszczają przemianowania identyfikatorów, a nawet zmiany algorytmu. W niektórych przypadkach za duplikaty uznaje się także fragmenty nie identyczne, ale w dużym stopniu podobne. Ocena podobieństwa fragmentów kodu wymaga złożonych struktur danych i ma dużą złożoność obliczeniową.

Sposób działania zależy od lokalizacji problemu. W przypadku pojedynczej klasy należy wyłączyć zduplikowany kod do wspólnej metody. W przypadku klas "bliźniaczych" przesunąć go do wspólnej nadklasy jako dziedziczoną metodę. W przypadku klas niespokrewnionych ze sobą możliwe jest m.in. utworzenie nowej klasy z duplikatami i odwoływanie się do niej lub jej instancji z dotychczasowych lokalizacji zduplikowanego kodu.

**Nazwa**

Długa metoda

Objawy

- Metoda wykonuje w rzeczywistości wiele czynności
- Brak wsparcia ze strony metod niższego poziomu

Rozwiązanie


- wyłącz fragmenty do nowych metod (*Extract Method*)
- wyłącz zmienne tymczasowe do zewnętrznych metod (*Replace Temp with Query*)
- wyłącz metodę do nowej klasy (*Replace Method with Method Object*)
- zmniejsz liczbę parametrów (*Introduce Parameter Object* lub *Preserve Whole Object*)

Kolejnym często spotykanym problemem w kodzie są długie metody. Istotna nie jest jednak bezwzględna długość (jest ona tylko objawem), a liczba funkcji, jakie metoda realizuje. Przyczyn jest wiele – jedną z nich jest niewłaściwy podział aplikacji na warstwy funkcjonalne i brak wsparcia ze strony metod w niższych warstwach. Wówczas długa metoda sama realizuje funkcje, które powinny być dostarczone przez inną warstwę aplikacji.

Podobnie jak w przypadku duplikatów, podstawową metodą usuwania tego problemu jest wyłączanie kodu do nowych metod. Jednak nie zawsze jest to możliwe – podział często jest blokowany przez zależności między zmiennymi lokalnymi w metodzie. Wówczas można zamienić je w metody, co pozwala zmniejszyć liczbę powiązań między nimi. Jeżeli to niemożliwe, warto zamienić metodę w nową klasę, zmieniając parametry metody w pola klasy.

Ponieważ długie metody posiadają często długie listy parametrów, warto również usunąć ten problem za pomocą łączenia ich we wspólne klasy.

Zaawansowane projektowanie obiektowe

Large Class

Nazwa
Nadmiernie rozbudowana klasa

Objawy

- Klasa posiada zbyt wiele odpowiedzialności
- Liczne klasy wewnętrzne, pola i metody
- Duża liczba metod upraszczających

Rozwiązanie

wydziel nową klasę i odwołuj się do niej za pomocą referencji (*Extract Class*), dziedziczenia (*Extract Subclass* lub *Extract Superclass*) lub korzystając z polimorfizmu (*Extract Interface*), a następnie przesuń do niej pola i metody (*Pull up Member*, *Push Down Member*, *Move Member*)


Wprowadzenie do refaktoryzacji (32)

Problem nadmiernej złożoności klas jest rozwinięciem poprzedniego przykrego zapachu. Klasa niekoniecznie musi być fizycznie zbyt długa – istotą problemu jest zbyt duża odpowiedzialność, jaką jest obciążona. W efekcie wewnątrz jednej klasy fizycznej znajdują się dwie lub więcej klas logicznych, czyli odrębnych zakresów odpowiedzialności.

Jest kilka objawów tego nadużycia: klasa posiada wiele metod, pól, klas wewnętrznych, dużą liczbę metod służących jedynie wygodzie programisty etc. Na poziomie pomiarów najlepszym wskaźnikiem tego przykrego zapachu jest niska spójność klasy. Oznacza ona, że klasa powinna być podzielona na mniejsze jednostki.

Właśnie to rozwiązanie jest najczęściej stosowane w celu usunięcia problemu. Nowa klasa może być niespokrewniona z klasą oryginalną, być jej pod- lub nad-klasą lub dziedziczyć wspólny interfejs.

Zaawansowane projektowanie obiektowe



Long Parameter List

Nazwa

Długa lista parametrów

Objawy

Metoda otrzymuje więcej danych niż potrzebuje

Rozwiązanie


usuń zbędne parametry (*Replace Parameter with Method, Preserve Whole Object* lub *Introduce Parameter Object*)

Wprowadzenie do refaktoryzacji (33)

Długa lista parametrów czasem jest związana z innymi przykrymi zapachami, w szczególności z długimi metodami. Najogólniej, metoda obciążona tym problemem otrzymuje zbyt wiele informacji, której albo nie wykorzystuje, albo wykorzystuje do realizacji zadań odmiennych, niespójnych z ogólnym celem metody.

Sposobem przeciwdziałania jest przede wszystkim zmniejszenie liczby parametrów (usunięcie zbędnych, połączenie niektórych parametrów w nową klasę) lub podział metody na mniejsze jednostki.

Zaawansowane projektowanie obiektowe

Comments

Nazwa
Nadmiar komentarzy

Objawy
Komentarze niepotrzebnie opisują znaczenie kodu

Rozwiązanie

- przesun nadmiernie skomentowane fragmenty do nowej metody (*Extract Method*)
- zmień nazwę metody (*Rename Method*) aby lepiej opisywała swoje przeznaczenie
- wprowadź asercję (*Introduce Assertion*) jeżeli to wyjaśnia przepływ sterowania w kodzie


Wprowadzenie do refaktoryzacji (34)

Nadmiar komentarzy jest zapachem o niewielkiej dokuczliwości, ale warto o nim wspomnieć, ponieważ budzi on naturalne i zrozumiałe kontrowersje. Pełne dokumentowanie kodu źródłowego jest przecież oczywistym postulatem inżynierii oprogramowania; zaleca się programistom, aby komentowali każdą istotną decyzję odpowiednią adnotacją.

Nadmiarem komentarzy nazywana jest sytuacja, w której komentarz zawiera dokładnie te same informacje i na tym samym poziomie czytelności, co sam kod źródłowy. W ten sposób komentarz staje się niepotrzebnym duplikatem, który sam w sobie wymaga pielęgnacji i aktualizacji.

Jeżeli struktura kodu jest tak skomplikowana, że wymaga rozbudowanego komentarza, wówczas konieczna jest jej modyfikacja. Podstawowym rozwiązaniem jest, jak w wielu podobnych przypadkach, utworzenie nowej metody. Bardzo ważne jest nazewnictwo metod, aby intuicyjnie odzwierciedlało ono zadania stojące przed metodą. Czasem czytelność jest wzmacniana poprzez wprowadzenie asercji, która w odróżnieniu od komentarza jest weryfikowana w momencie wykonywania programu, a jej naruszenie jest zgłaszane programiście.

Zaawansowane projektowanie obiektowe

Incomplete Library Classs

Nazwa
Niekompletna klasa biblioteczna

Objawy
Gotowa biblioteka nie posiada pewnej funkcjonalności

Rozwiązanie


- utwórz brakujące metody po stronie klienta i opatrz je właściwym komentarzem (*Introduce Foreign Method*)
- utwórz podklasę lub obiekt opakowujący w celu zwiększenia funkcjonalności klasy (*Introduce Local Extension*)

Wprowadzenie do refaktoryzacji (35)

Niekompletność klasy bibliotecznej nie jest wynikiem błędu programisty korzystającego z tej klasy, jednak niewątpliwie stanowi zagrożenie dla struktury programu. Brakująca funkcja musi być zaimplementowana w miejscu, w którym intuicyjnie będzie można ją odnaleźć.

Sugerowane są dwa rozwiązania: w prostych przypadkach należy zaimplementować funkcję po stronie klienta, opatrując ją komentarzem dotyczącym niewłaściwej lokalizacji i jej przyczyn. Jeżeli brakuje większej liczby funkcji, wówczas można zastosować tzw. lokalne rozszerzenie klasy bibliotecznej, czyli specjalizowaną podklasę lub opakowanie (ang. *wrapper*), posiadające te funkcje. Oba rozwiązania mają wady, które wpływają na decyzję dotyczącą ich zastosowania. Stworzenie podklasy wymaga, aby klasa była niefinalna, z drugiej strony udostępnia jednak składowe klasy aż do poziomu chronionego. Aby stworzyć opakowanie należy m.in. zaimplementować interfejs klasy bibliotecznej, co oznacza, że jego brak uniemożliwia zastosowanie tego rozwiązania. Ponadto opakowanie ma dostęp jedynie do publicznych składowych klasy bibliotecznej.

Zaawansowane projektowanie obiektowe

Switch Statements

Nazwa
Skomplikowane instrukcje warunkowe

Objawy
Metoda zawiera złożoną, wielopoziomową instrukcję *if* lub *switch*

Rozwiązanie


- wyłącz wspólne fragmenty do nowej metody (*Extract Method*)
- wyłącz gałęzie instrukcji warunkowej do polimorficznych klas (*Replace Conditional with Polimorphism/State*)
- wyłącz gałęzie instrukcji warunkowej do podklas (*Replace Conditional with Subclasses*)

Wprowadzenie do refaktoryzacji (36)

Niepoprawne wykorzystanie instrukcji wyboru i warunkowej w dużym stopniu wpływa na skomplikowanie przepływu sterowania w programie. Rozbudowane warunki oraz gałęzie kodu wykonywane po ich spełnieniu lub odrzuceniu przypominają zasady programowania strukturalnego, a przede wszystkim utrudniają zrozumienie ich sensu.

Pierwszym krokiem w kierunku usunięcia tego problemu jest ponownie wyłączenie fragmentów do nowych metod o intuicyjnych nazwach – w wielu przypadkach jest to rozwiązanie wystarczające. W innych można podzielić instrukcje warunkowe korzystając z mechanizmów obiektowych: polimorfizmu (każda gałąź staje się osobną implementacją tego samego interfejsu) lub dziedziczenia (każda podklasa reprezentuje jedną z gałęzi). Wówczas problem sterowania warunkami sprowadza się do tworzenia obiektów odpowiednich klas, a nie wyboru pomiędzy długimi blokami kodu.

Zaawansowane projektowanie obiektowe

Message Chains

Nazwa
Łańcuchy wywołań metod

Objawy

- łańcuch wywołań przez delegację
- naruszenie prawa Demeter

Rozwiązanie


- usunąć niepotrzebne wywołania i ukryć delegację (*Hide Delegate*)
- przesunąć zbędne metody do klas w dalszej części łańcucha (*Extract Method* i *Move Method*)

Wprowadzenie do refaktoryzacji (37)

Omówione wcześniej prawo Demeter stanowi regułę opisującą prawidłowy sposób wiązania ze sobą niespokrewnionych obiektów. Omawiany przykry zapach stanowi naruszenie prawa Demeter: metoda w jednym z obiektów dokonuje swoistej podróży przez cały system, wywołując metody w celu znalezienia kolejnych obiektów. Takie rozwiązanie powoduje, że metoda ta jest zależna od całej reszty systemu.

Rozwiązaniem tego problemu może być stworzenie dodatkowych metod ukrywających delegacje w kolejnych obiektach. Zwiększy to liczbę metod, ale zmniejszy liczbę powiązań pomiędzy obiektami. Można również próbować przesuwać metody między obiektami znajdującymi się wewnątrz łańcucha.

Zaawansowane projektowanie obiektowe

Data Class

Nazwa
Pojemnik na dane

Objawy
Klasa jedynie przechowuje dane i nie posiada metod oferujących użyteczne funkcje (*Data Transfer Object*)

Rozwiązanie


- przenieść część kodu klientów do klasy-pojemnika danych (*Extract Method* i *Move Method*)
- podzielić dane klasy pomiędzy klientów i usunąć ją (*Inline Class*)

Wprowadzenie do refaktoryzacji (38)

Klasa jest jednostką kodu obdarzoną odpowiedzialnością. Zatem klasy, których odpowiedzialność sprowadza się do przechowywania danych, nie są poprawne i powinny zostać zmodyfikowane. Oczywiście, istnieją technologie lub obszary zastosowań, w których tego typu obiekty są pożądane. Przykładem może być wzorzec projektowy J2EE Data Transfer Object, w którym obiekt takiej klasy służy do przenoszenia zbioru danych między rozproszonymi komponentami aplikacji. Takie rozwiązanie przynosi wówczas znaczny wzrost wydajności. Podobna argumentacja może również (choć nie powinna) dotyczyć także technologii Java Beans. W praktyce jednak zawsze powinno się ze zbiorem danych wiązać pewną funkcjonalność – i właśnie jej braku dotyczy ten przykry zapach.

Istnieją dwa rozwiązania: rozszerzenie klasy o nową funkcjonalność, aby stała się w pełni odpowiedzialna za siebie, lub przeniesienie pozostałych resztek funkcjonalności i usunięcie klasy z systemu. Wówczas dane przechowywane przez nią są rozdzielane pomiędzy korzystających z niej klientów.

Zaawansowane projektowanie obiektowe

Data Clumps

Nazwa
Zbitka danych

Objawy
Zbiór danych zawsze występuje wspólnie (np. autor, tytuł książki i wydawnictwo)

Rozwiązanie


- utwórz nową klasę ze zbitki (*Extract Class*) i przekazuj jej instancje zamiast zbitki (*Introduce Parameter Object* i *Preserve Whole Object*)
- postępuj podobnie jak z *Long Parameter List*

Wprowadzenie do refaktoryzacji (39)

Ten problem jest związany z innym przykrym zapachem – długą listą parametrów. Pojawiają się zbiory parametrów, które są wewnętrznie ze sobą związane, tworząc jeden logiczny zestaw informacji. Przekazywanie takich danych osobno ogranicza abstrakcję oraz dodatkowo komplikuje komunikację między metodami.

Usunięcie tego przykrego zapachu polega na utworzeniu nowej klasy, która będzie przechowywała dotychczasowe osobne dane jako swoje pola. Warto zauważyć, że łatwo doprowadzić w ten sposób do powstania przykrego zapachu Data Class, jednak rozbudowa funkcjonalności nowej klasy może temu zapobiec. Natomiast przekazywanie pojedynczego obiektu zamiast zbitki danych pozwala na uproszczenie struktury programu.

Zaawansowane projektowanie obiektowe

Refused Bequest

Nazwa
Odrzucony spadek

Objawy
Podklasa nie wykorzystuje odziedziczonych metod i pól

Rozwiązanie


- utwórz nową podklasę i przesuń do niej niechciane składowe z nadklasy (*Push Down Member*)
- jeżeli podklasa nie powinna posiadać interfejsu nadklasy, zastosuj delegację zamiast dziedziczenia (*Replace Inheritance with Delegation*)

Wprowadzenie do refaktoryzacji (40)

Odrzucony spadek bardzo mocno wiąże się z ogólniejszym problemem złego wykorzystania dziedziczenia do współdzielenia kodu. Ma miejsce wówczas, gdy nadklasa oferuje swoim podklasom znaczną funkcjonalność, natomiast one z niej nie korzystają. Oznacza to, że klasy te są znacznie słabiej związane ze sobą niż wskazywałyby na to rodzaj relacji między nimi.

Problem ten można usunąć przede wszystkim przesuwając wybrane metody do podklas. Pozwala to na usunięcie ich z tych podklas, które ich nie potrzebują, i jednocześnie odciążenie nadklasy. Drugim rozwiązaniem jest rezygnacja z relacji dziedziczenia na rzecz delegacji. Wówczas relacja w lepszym stopniu odzwierciedla powiązanie pomiędzy obydwoma klasami.

Zaawansowane projektowanie obiektowe

Inappropriate Intimacy

Nazwa
Niewłaściwa hermetyzacja

Objawy
Klasa bezpośrednio odwołuje się do składowych wewnętrznych innej klasy

Rozwiązanie


- przesunąć składową do właściwej klasy (*Move Member*)
- ograniczyć powiązania do jednokierunkowych (*Change Bi-directional References to Unidirectional*)
- wyłączyć nową klasę ze wspólnych elementów obu klas (*Extract Class*)
- w przypadku dziedziczenia odseparuj nadklasę od podklasy (*Replace Inheritance with Delegation*)

Wprowadzenie do refaktoryzacji (41)

Niewłaściwa hermetyzacja jest dość częstym i dobrze znanym problemem. Wiąże się ona nie tylko z niepoprawnym użyciem kwalifikatorów dostępu, ale przede wszystkim z niewłaściwym podziałem odpowiedzialności pomiędzy klasy: jeżeli wymagają one bezpośredniego dostępu do swoich prywatnych części, oznacza to, że są zbyt ze sobą związane.

Podstawowym rozwiązaniem jest przesunięcie metod i pól prywatnych do klas, które najbardziej ich potrzebują. Można także ograniczyć wiedzę o sobie dwóch klas, zmieniając relację dwukierunkową na jednokierunkową. Wrażliwe i współdzielone elementy dwóch klas można wyłączyć do nowej klasy, której obiekty będą współdzielone pomiędzy nie, zachowując jednocześnie hermetyzację. Niepoprawna hermetyzacja jest również możliwa wewnątrz hierarchii dziedziczenia – wówczas w celu odseparowania nadklasy od podklas można zmienić dziedziczenie w delegację.

Zaawansowane projektowanie obiektowe

Lazy Class

Nazwa
Bezużyteczna klasa

Objawy
Klasa nie posiada żadnej lub ograniczoną funkcjonalność

Rozwiązanie

- przesunąć wybrane funkcje z klas klienckich (*Move Member*)
- w przypadku podklas usunąć klasę z hierarchii dziedziczenia (*Collapse Hierarchy*)
- podzielić składowe pomiędzy jej klientów (*Move Member*) i usunąć ją (*Inline Class*)


Wprowadzenie do refaktoryzacji (42)

Bezużyteczna klasa jest przeciwieństwem klasy nadmiernie rozbudowanej. Jeżeli klasa nie posiada żadnej odpowiedzialności lub jest ona ograniczona do pojedynczych drobnych funkcji, warto zastanowić się nad jej modyfikacją. Klasa bezużyteczna jest często blisko powiązana z innym przykrym zapachem – Data Class. O ile jednak w ostatnim przypadku klasa przechowuje pewne dane, o tyle klasa bezużyteczna nie posiada określonego i spójnego zakresu odpowiedzialności.

Usuwanie tego zapachu może podążać dwiema drogami: zwiększając jej odpowiedzialność kosztem współpracujących z nią klas klienckich lub stopniowo ją ograniczając, a następnie usuwając z systemu. W przypadku dziedziczenia klasa jest usuwana poprzez przeniesienie jej funkcji do nadklasy i/lub podklas, natomiast w pozostałych przypadkach jej składowe są przesuwane do klientów poprzez delegacje.

Zaawansowane projektowanie obiektowe

Feature Envy



Nazwa
Zazdrość o funkcje

Objawy

- Metoda w klasie częściej korzysta z metod w obcych klasach niż we własnej
- Niska spójność klasy

Rozwiązanie


- przesun metodę do właściwej klasy (*Move Method*)
- przekaz referencję *this* do metody w drugiej klasie (wzorec projektowy *Visitor*)

Wprowadzenie do refaktoryzacji (43)

Problem ten dotyczy sytuacji, w której metoda częściej odwołuje się do metod obcych klas niż do metod własnej klasy. Zazdrość o funkcje jest także związana z niewłaściwą hermetyzacją, jednak podstawowym problemem opisywanym przez ten przykry zapach jest niepoprawne rozmieszczenie metod w klasach. Typowym ilościowym wskaźnikiem tego przykrego zapachu jest niska spójność klasy: jej metody nie realizują spójnego zbioru funkcji.

Podobnie jak w opisanym wcześniej przypadku niewłaściwej hermetyzacji, z problemem tym można poradzić sobie przenosząc składowe do klasy najbardziej je wykorzystujących. Czasem jednak jest to niemożliwe, ponieważ zwiększałoby liczbę powiązań między pozostałymi klasami. Wówczas zastosowanie znajduje np. wzorec projektowy *Visitor*, w którym obiekt wywołując metodę przekazuje jej referencję do samego siebie, umożliwiając tzw. odwrócenie sterowania. Pozwala to ograniczyć liczbę zbędnych asocjacji między klasami.

Zaawansowane projektowanie obiektowe

Parallel Inheritance Hierarchies

Nazwa
Równoległe hierarchie dziedziczenia

Objawy
Utworzenie podklasy powoduje konieczność utworzenia odpowiadającej jej podklasy w innej hierarchii dziedziczenia

Rozwiązanie


- przenieś metody między hierarchiami i ujednolich ich interfejsy (*Move Method*, *Move Field*, *Extract Interface*)
- usuń zbędne klasy (*Inline Class*)

Wprowadzenie do refaktoryzacji (44)

Równoległe hierarchie dziedziczenia mogą oznaczać błąd w przydziale odpowiedzialności do klas, ponieważ stworzenie klasy w jednej hierarchii oznacza również konieczność stworzenia jej odpowiednika w drugiej. Jednak ocena nie jest całkowicie jednoznaczna: podobne rozwiązania stosuje się w kilku wzorcach projektowych z rodziny Factory (np. Factory Method czy Abstract Factory). Wówczas obiekt tworzący jest także odseparowany od produktu, i dodanie jednego z nich wymaga dodania także drugiego. Zatem obecność tego przykrego zapachu należy badać dość ostrożnie.

Rozwiązanie polega na przeniesieniu odpowiedzialności z klas jednej hierarchii do drugiej, połączone z unifikacją ich interfejsów. Prowadzi to w dużej mierze do pozostawienia jednej hierarchii klas i usunięcia zbędnych odpowiedników w drugiej.

Zaawansowane projektowanie obiektowe

Middle Man

Nazwa
Pośrednik

Objawy
Klasa deleguje większość swojej funkcjonalności do innych klas

Rozwiązanie


- usunąć pośrednika (*Remove Middle Man*)
- usunąć zbędne metody (*Inline Method*)
- zmienić pośrednika w podklasę (*Replace Delegation with Inheritance*)

Wprowadzenie do refaktoryzacji (45)

Ten przykry zapach oznacza sytuację, w której cała odpowiedzialność pewnej klasy sprowadza się do delegowania wywołań do innych klas. Jest to zatem sytuacja odwrotna do problemu o nazwie Message Chains – w tamtym przypadku liczba pośrednich odwołań była zbyt mała, w tym – jest zbyt duża. Oczywiście, istnieją sytuacje, w których klasa będąca jedynie tłumaczem jednych metod na drugie ma uzasadnienie (np. wzorzec projektowy Adapter), ale są to wyjątki. Generalnie, klasa-pośrednik posiada zbyt mało odpowiedzialności, aby samodzielnie funkcjonować.

Rozwiązanie polega na usunięciu pośrednika, które można wykonać na kilka różnych sposobów. Można zastąpić wywołania poprzez pośrednika wywołaniami bezpośrednimi, co prowadzi do zwiększenia liczby powiązań między klientem (klasą wywołującą) a serwerem (klasą faktycznie obsługującą żądanie). Można usunąć zbędne metody delegujące wewnątrz pośrednika, co jednak jest tylko rozwiązaniem częściowym. Wreszcie, można zastąpić pośrednika dostępnego przez delegację podklasą – czyli zmienić delegowanie wywołań na rzecz ich dziedziczenia i pokrywania. To ostatnie rozwiązanie jest także stosowane przez jedną z wersji wzorca projektowego Adapter.

Zaawansowane projektowanie obiektowe

Divergent Change

Nazwa
Zmiany z wielu przyczyn

Objawy
Klasa jest wielokrotnie modyfikowana w różnych celach


Rozwiązanie
znajdź elementy zmieniające się z jednego powodu i
hermetyzuj w postaci osobnej klasy (*Extract Class*)

Wprowadzenie do refaktoryzacji (46)

Konieczność ciągłych modyfikacji klasy z wielu powodów wskazuje na jej niejasne przeznaczenie: pełni ona wiele niezależnych ról i zależy od wielu czynników.

Rozwiązaniem jest wydzielenie z niej fragmentów kodu modyfikowanych z jednego powodu i stworzenie z nich oddzielnej klasy. Dzięki temu modyfikacje będą przeniesione do oddzielnych klas, co pozwoli ograniczyć zasięg zmian.

Zaawansowane projektowanie obiektowe

Shotgun Surgery

Nazwa
Odpryskowa modyfikacja

Objawy
Zmiana w jednym miejscu powoduje konieczność modyfikacji w innych

Rozwiązanie


- umieścić zmienne elementy wewnątrz jednej klasy (*Extract Class*, *Move Method* i *Move Field*)
- usunąć zbędne klasy (*Inline Class*)

Wprowadzenie do refaktoryzacji (47)

Ten przykry zapach jest pewnego rodzaju uzupełnieniem poprzedniego. Ma miejsce wówczas, gdy zmiana w tej klasie powoduje modyfikację innych klas. Różnica polega na odwrotnym kierunku zależności: w przypadku Divergent Change są to zależności przychodzące, natomiast w tym – wychodzące. Jest to zgodne z koncepcją R. Martina niezależnych metryk dla tych dwóch rodzajów zależności: Ca i Ce.

Tym razem przeciwdziałanie temu problemowi polega na hermetyzacji w obrębie jednej klasy wszystkich obszarów podlegającym zmianie z jednego powodu – poprzez przesunięcia metod, pól, tworzenie nowych klas i usuwanie niepotrzebnych.

Zaawansowane projektowanie obiektowe

Speculative Generality

Nazwa

Spekulacyjne uogólnianie

Objawy

Klasa jest zaprojektowana pod kątem potencjalnej funkcjonalności do zaimplementowania w przyszłości

Rozwiązanie


- usunąć klasy abstrakcyjne z hierarchii (*Collapse Hierarchy*)
- usunąć zbędne delegacje do innych klas (*Inline Class*)
- usunąć zbędne parametry metod (*Remove Parameter*)
- dostosować nazwy metod do ich przeznaczenia (*Rename Method*)

Wprowadzenie do refaktoryzacji (48)

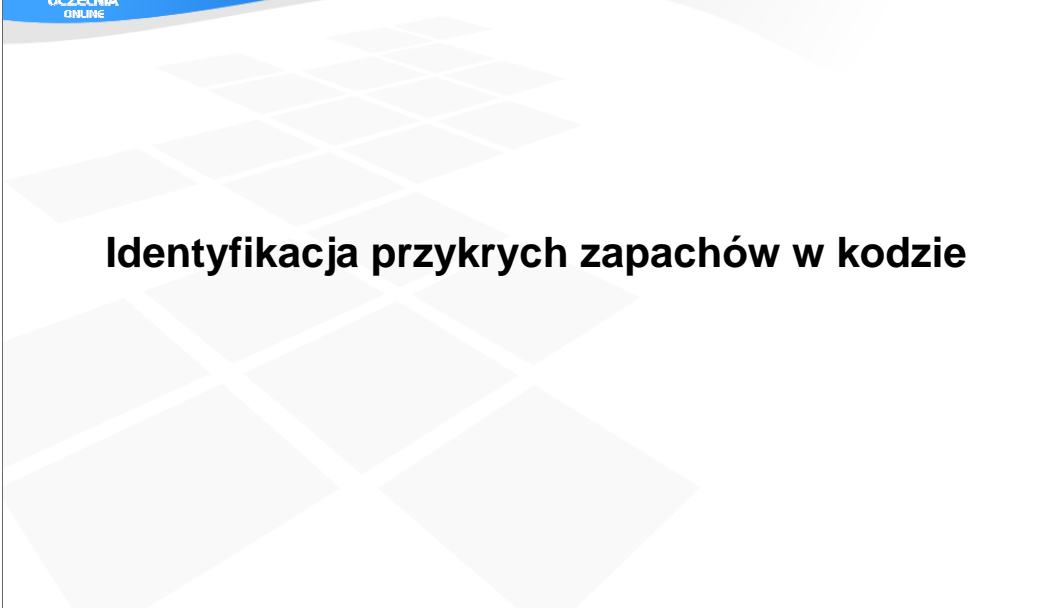
Ostatni przykry zapach podany przez Fowlera to – mówiąc kolokwialnie - nadmierna elastyczność i otwartość systemu na zmiany, które nigdy nie nastąpią. Jedną z podstawowych zasad projektowania obiektowego jest hermetyzacja obszarów zmienności, jednak pod warunkiem, że ta zmienność faktycznie występuje. Optymistyczne zakładanie zmian w pewnych kierunkach zwykle okazuje się nieuzasadnione i prowadzi do nadmiernie rozbudowanej struktury kodu w stosunku do potrzeb.

Uproszczenie jest realizowane poprzez usuwanie zbędnych klas, ograniczanie hierarchii dziedziczenia, usuwanie pośredników, metod pomocniczych, nadmiarowych parametrów etc.

Zaawansowane projektowanie obiektowe

Uczelnia
ONLINE

Plan wykładu



Identyfikacja przykrych zapachów w kodzie

Wprowadzenie do refaktoryzacji (49)

W celu skutecznego refaktoryzowania kodu, poza znajomością definicji przykrych zapachów niezbędna jest także umiejętność ich identyfikacji. W tej części wykładu krótko zostaną przedstawione metody i narzędzia wykrywania przykrych zapachów w kodzie programów.



Jak mówi definicja, obecność przykrego zapachu wskazuje na potrzebę refaktoryzacji. Jednak jak stwierdzić, czy przykry zapach faktycznie występuje? Identyfikacja wymienionych na poprzednich slajdach naruszeń zasad projektowych nie jest łatwa, ponieważ ich występowanie nie jest związane prostą zależnością przyczynowo-skutkową z jednym objawem. Przeciwnie – ten sam przykry zapach może przejawiać się na różne sposoby, które niekoniecznie występują wspólnie.

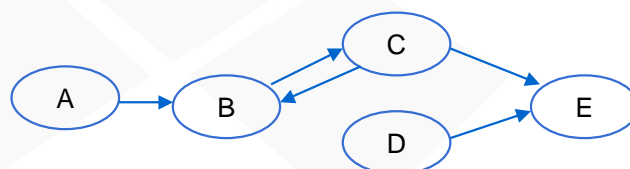
Dlatego wyróżniono sześć odrębnych źródeł, które mogą dostarczać danych o obecności (lub jej wykluczeniu) danego przykrego zapachu. Są to:

- intuicja programisty, która jest niemierzalna i trudna do zdefiniowania, a jednak pełni bardzo ważną rolę w identyfikacji złych praktyk obiektowych;
- metryki, które są podstawowym mechanizmem ilościowej oceny jakości projektu. Metryki dostarczają liczbowej i łatwej do zinterpretowania informacji o wielu aspektach jakości projektu;
- analiza Abstrakcyjnego Drzewa Składniowego (AST), które jest graficzną reprezentacją rozbioru gramatycznego programu. Obecność lub brak określonych elementów w drzewie AST wskazuje na obecność lub wyklucza obecność niektórych zapachów;
- informacja o innych zapachach pozwala wykorzystać znane już fakty o innych przykrych zapachach;
- analiza dynamiczna, która pozwala określić atrybuty programu nie dające się zidentyfikować wyłącznie poprzez analizę statyczną. Przykładem analizy dynamicznej może być wykonanie przypadków testowych;
- historia zmian kodu pochodząca z repozytorium zarządzania konfiguracją pozwala ocenić wpływ niektórych zmian na program.



Relacje pomiędzy przykrymi zapachami

- **Wsparcie proste**
 - obecność jednego zapachu sugeruje obecność innego ($A \Rightarrow B$)
- **Wsparcie wzajemne**
 - dwa przykre zapachy sugerują wzajemnie swoją obecność ($B \Leftrightarrow C$)
- **Wsparcie złożone**
 - grupa przykrych zapachów sugeruje obecność innego ($C, D \Rightarrow E$)
- **Wsparcie przechodnie**
 - przechodnie domknięcie relacji prostego wsparcia ($A \Rightarrow B \Rightarrow C$)



Przykładem informacji wykorzystywanej przy wykrywaniu przykrych zapachów jest wiedza o obecności lub nieobecności innych zapachów. Istnieje siedem rodzajów relacji pomiędzy przykrymi zapachami, które można wykorzystać w procesie ich identyfikacji:

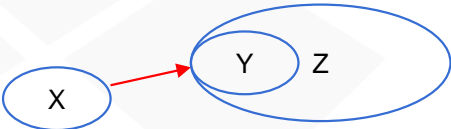
- proste wsparcie, w którym obecność jednego zapachu zwiększa prawdopodobieństwo istnienia innego
- wsparcie wzajemne, będące domknięciem symetrycznym relacji prostego wsparcia
- wsparcie złożone, w którym przesłanką relacji wsparcia jest jednoczesna obecność kilku przykrych zapachów
- wsparcie przechodnie, które składa się z łańcucha relacji wsparcia prostego (jest domknięciem przechodnim tej relacji)

Zaawansowane projektowanie obiektowe

UCZELNIA ONLINE

Relacje pomiędzy przykrymi zapachami

- **Wykluczenie**
 - obecność jednego zapachu wyklucza obecność innego ($X \not\supset Y$)
- **Zawieranie**
 - jeden przykry zapach jest szczególnym przypadkiem innego ($Y \subset Z$)
- **Wspólne przekształcenie**
 - usunięcie jednego zapachu usuwa jednocześnie drugi
 - usunięcie jednego zapachu może wprowadzić drugi



Wprowadzenie do refaktoryzacji (52)

- wykluczenie, w którym obecność jednego zapachu wyklucza obecność innego
 - zawieranie, gdzie jeden przykry zapach jest szczególnym przypadkiem innego
 - wspólne przekształcenie, które powoduje usunięcie obu przykrych zapachów albo usunięcie jednego i możliwe wprowadzenie drugiego.
- Znajomość tych relacji pozwala zwiększyć pewność wykrywania przykrych zapachów, zmniejszyć złożoność obliczeniową procesu wykrywania lub określić potrzebne przekształcenie.



- Refaktoryzacja jest techniką umożliwiającą łatwiejszą i tańszą pielęgnację oprogramowania
- Refaktoryzacja zachowuje funkcjonalność programu
- Analiza statyczna i testowanie są metodami weryfikacji poprawności refaktoryzacji
- Przykry zapach w kodzie jest określeniem złej struktury programu wymagającej poprawy
- Identyfikacja przykrych zapachów wymaga złożonego mechanizmu wspomagania decyzji

Wykład był wprowadzeniem do refaktoryzacji i miał na celu zapoznanie z celami i możliwymi korzyściami wynikającymi ze stosowania tej techniki pielęgnacji. Najważniejszą własnością refaktoryzacji jest brak zmiany zachowania programu. Warunek ten jest niezbędny do prawidłowego przekształcania programu, dlatego główne prace koncentrują się właśnie na nim. Dwie podstawowe metody weryfikacji poprawności refaktoryzacji to analiza statyczna kodu i testowanie, stanowiące formę analizy dynamicznej.

W celu określenia stanu kodu, który wskazuje na potrzebę refaktoryzacji, wprowadzono pojęcie przykrego zapachu w kodzie programu (ang. *code bad smell*). Obejmuje on wiele różnych faktycznych naruszeń zasad dobrego projektowania. Dlatego wykrywanie przykrych zapachów jest trudne i wymaga analizy wielu różnych objawów.