Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Adam Kieżun
Nr albumu: 159146

Refaktoryzacja programów w języku Java

Praca magisterska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
dr Janiny Mincer–Daszkiewicz
Instytut Informatyki

Lipiec 2001

Pracę przedkładam do oceny

Data Podpis autora pracy:

Praca jest gotowa do oceny przez recenzenta

Data Podpis kierującego pracą:

Streszczenie

Dostosowanie istniejących programów komputerowych do nowych lub zmieniających się wymagań jest trudne i czasochłonne. Jednym ze sposobów radzenia sobie z tym zadaniem jest *refaktoryzacja* – proces przekształcania kodu źródłowego niezmieniający działania programu. Ułatwia on budowę i pielęgnację oprogramowania, umożliwiając utrzymanie systemu komputerowego w stanie pozwalającym na łatwą rozbudowę.

Refaktoryzacja jest jednak procesem czasochłonnym i podatnym na błędy, wobec czego pożądana jest jej automatyzacja. Stworzenie niezawodnych i wydajnych narzędzi wspierających programistów w tym zadaniu jest niezbędnym warunkiem do tego, by refaktoryzacja mogła stać się powszechnie przyjętą techniką tworzenia i pielęgnacji programów.

W niniejszej pracy przedstawiamy refaktoryzację w kontekście języka Java. Opisujemy trudności związane ze stworzeniem niezawodnego narzędzia uwzględniającego wszystkie konstrukcje języka. Pokazujemy w jaki sposób komplikują one to zadanie, a następnie omawiamy możliwe sposoby rozwiązania problemów z tym związanych. Dokonujemy także szczegółowej analizy warunków wstępnych kilku refaktoryzacji, biorącej pod uwagę wszystkie aspekty języka.

Jednym z wyników tej pracy jest stworzenie narzędzia do refaktoryzacji, będącego obecnie składnikiem środowiska programistycznego IBM WebSphere Studio Workbench. Niniejsza praca zawiera opis projektu i implementacji tego narzędzia.

Słowa kluczowe

refaktoryzacja, Java, inżynieria oprogramowania, pielęgnacja programów

Klasyfikacja tematyczna

D. Software
D.2 Software Engineering
D.2.7 Distribution, Maintenance, Enhancement

Spis treści

1. Wstęp 5

2. Refaktoryzacja 7
2.1. Definicja refaktoryzacji 7
2.2. Przykłady 7
2.3. Refaktoryzacja a projektowanie i pielęgnacja oprogramowania 9
2.4. Sposób opisu refaktoryzacji 10
2.5. Klasyfikacja refaktoryzacji 11
2.5.1. Podział według budowy 11
2.5.2. Podział według obszaru działania 11
2.5.3. Podział według stopnia automatyzacji 12
2.6. Uzasadnianie poprawności refaktoryzacji 12
2.7. Refaktoryzacja wspomagana automatycznie 13
2.8. Istniejące narzędzia do wspomagania refaktoryzacji 14
2.8.1. The Smalltalk Refactoring Browser 14
2.8.2. jFactor 15
2.8.3. JREB 15
2.8.4. XRef Speller 16
2.8.5. JRefactory 17
2.8.6. IntelliJ Renamer 17
2.8.7. Transmogrify 18
2.8.8. Podsumowanie przeglądu narzędzi 18

3. Refaktoryzacja programów w języku Java 21
3.1. Wielokrotne dziedziczenie interfejsów oraz efekt fali 22
3.2. Przestrzenie nazw typów 25
3.3. Przesłanianie, ukrywanie i zaciemnianie nazw 31
3.4. Przeciążanie nazw metod 33
3.5. Metody natywne 33
3.6. Przypadki specjalne 35
3.7. Podsumowanie 37

4. Wsparcie refaktoryzacji w IBM WebSphere Studio Workbench 39
4.1. Opis architektury środowiska programistycznego 40
4.2. Przyjęte założenia projektowe 40
4.3. Przepływ sterowania 42
4.4. Wspomagane refaktoryzacje 42
4.4.1. Wydzielenie Metody 43

4.4.2. Zmiana Nazwy Pakietu . . . 43
4.4.3. Zmiana Nazwy Typu . . . 43
4.4.4. Zmiana Nazwy Metody . . . 43
4.4.5. Zmiana Nazwy Pola . . . 43
4.4.6. Zmiana Nazw Parametrów Metody . . . 43
4.4.7. Zmiana Nazwy Jednostki Kompilacji . . . 43
4.4.8. Przemieszczenie Jednostki Kompilacji do Innego Pakietu . . . 43
4.5. Główne części programu . . . 44
4.5.1. Klasa `Refactoring` . . . 44
4.5.2. Klasa `Change` . . . 45
4.5.3. Analizator drzew składni . . . 46
4.5.4. Interfejs użytkownika . . . 46
4.6. Wydajność i niezawodność narzędzia . . . 47

5. Podsumowanie . . . 49

A. Refaktoryzacje proste . . . 53

B. Przykładowe refaktoryzacje . . . 55
B.1. Zmiana Nazwy Typu . . . 55
B.2. Zmiana Nazw Parametrów Metody . . . 57
B.3. Zmiana Nazwy Pakietu . . . 57
B.4. Zmiana Nazwy Pola . . . 58
B.5. Przemieszczenie Jednostki Kompilacji do Innego Pakietu . . . 58

Bibliografia . . . 61

Rozdział 1

Wstęp

Programy komputerowe podlegają zmianom podczas swego istnienia. Zwykle są modyfikowane z powodu zmieniających się wymagań bądź pojawiania się nowych. Często jednak zmiana dotyczy jedynie wewnętrznej struktury programu, bez wpływu na zewnętrzne skutki jego działania. Przykładowo, programista zmienia nazwę klasy, by lepiej opisać jej przeznaczenie, zastępuje powtarzający się kod wywołaniem metody lub zmienia interfejs klasy, by uczynić ją bardziej elastyczną i użyteczną w innych częściach systemu. Zdarza się, że programiści modyfikują jakąś część programu po to, by uprościć jej zagmatwaną logikę bądź nawet jedynie ze względu na poczucie elegancji [19]. Taka modyfikacja kodu źródłowego programu, która nie zmienia jego działania jest znana pod nazwą refaktoryzacji (ang. *refactoring*). Termin *refaktoryzacja* oznacza zarówno pojedynczą transformację, jak i wieloetapowy proces przekształcania programu. Przedmiotem niniejszej pracy jest refaktoryzacja w kontekście popularnego obiektowego języka programowania – języka Java.

Refaktoryzacje można łączyć ze sobą. Wykonując wiele następujących po sobie, niewielkich przekształceń, z których każde ma tę właściwość, że nie zmienia działania programu, można zmodyfikować budowę programu w znaczący sposób – nie zmieniając jego zachowania [5, 17, 16, 19, 22].

Jest to jednak często proces żmudny i podatny na błędy. Zdarza się, że program pozostaje w swojej obecnej, niepożądanej postaci jedynie dlatego, że wprowadzenie koniecznych zmian (i testowanie ich poprawności) jest zbyt czasochłonne. Przykładowo, zła nazwy klasy prawdopodobnie nie zostanie ulepszona, jeżeli w programie istnieje bardzo wiele odwołań do tej klasy. Dlatego istotne jest, by tam, gdzie to możliwe, refaktoryzację automatyzować.

Do tego, by narzędzia do refaktoryzacji mogły być powszechnie używane, niezbędna jest ich niezawodność. Refaktoryzacja to proces głęboko ingerujący w strukturę programu – zmiana nazwy metody czy klasy często oznacza konieczność aktualizacji kodu w setkach miejsc. Z koncepcyjnego punktu widzenia, refaktoryzacja polega na wykonywaniu wielu niewielkich kroków. Każda kolejna zmiana wprowadzana do diagramów UML jest, być może, prawie niezauważalna. Faktyczna modyfikacja kodu, wymagana do wprowadzenia zmiany w życie, może być jednak bardzo znacząca. Narzędzie, któremu mamy powierzyć wprowadzenie tego rodzaju zmian w kodzie źródłowym programu musi być niezawodne.

Przed wykonaniem refaktoryzacji niezbędna jest analiza warunków wstępnych (ang. *preconditions*). Tylko wtedy, gdy są one spełnione, można bezpiecznie przeprowadzić określoną modyfikację kodu źródłowego. Możliwie pełna analiza warunków wstępnych jest konieczna do zagwarantowania poprawności refaktoryzacji i, tym samym, do osiągnięcia niezawodności narzędzi.

Java jest językiem programowania o złożonej składni i semantyce. Wielu konstrukcji wy-

stępujących w tym języku (przykładowo wielokrotne dziedziczenie, czy typy zagnieżdżone), nie rozważano, do tej pory, w kontekście refaktoryzacji. Jednym z celów tej pracy jest ukazanie znacznego wpływu niektórych cech języka Java na ten proces. Pokazujemy, że prawidłowe przeprowadzenie refaktoryzacji (w szczególności analiza warunków wstępnych) jest niemożliwe bez wzięcia tego pod uwagę. Następnie omawiamy problemy związane z występowaniem omawianych cech języka w refaktoryzowanych programach i możliwe sposoby ich rozwiązania.

Brak wyczerpujących opracowań dotyczących refaktoryzacji w języku Java może być, według nas, jedną z ważnych przeszkód w tworzeniu niezawodnych narzędzi. Kolejnym z celów tej pracy jest przedstawienie opisu kilku refaktoryzacji wraz ze szczegółową, obejmującą całość języka, analizą warunków wstępnych.

Kolejnym celem niniejszej pracy jest prezentacja projektu i implementacji zbudowanego przez nas narzędzia do automatycznego wspierania refaktoryzacji programów w języku Java. Jest ono integralną częścią IBM WebSphere Studio Workbench – środowiska programistycznego do budowy programów w języku Java. Narzędzie to jest używane na codzień przez wielu programistów. Jest dostępne (wraz z kodem źródłowym) pod adresem [24].

Zagadnienie włączenia wsparcia refaktoryzacji do środowiska programistycznego opisaliśmy w pracy [1] (wspólnie z dr Erichem Gammą i dr Dirkiem Bäumerem).

Dalsza część pracy jest zorganizowana w następujący sposób:

W rozdziale 2 przedstawiamy i precyzujemy pojęcie refaktoryzacji, uzasadniamy jej stosowanie, a także omawiamy dotychczasowe badania i osiągnięcia w tej dziedzinie, zarówno w literaturze, jak i praktyce.

W rozdziale 3 podajemy listę i opis tych właściwości języka Java, które mają istotny wpływ na refaktoryzację. Wielu z nich nie prezentowano dotychczas w tym kontekście. Uważamy, że przykłady najlepiej ilustrują omawiane przypadki, więc tam, gdzie nie daje się zwięźle opisać istoty problemu, przedstawiamy go na przykładzie.

Rozdział 4 to opis projektu i implementacji narzędzia do automatycznego wspomagania procesu refaktoryzacji. Podajemy przyjęte przez nas założenia projektowe oraz uzupełniamy listę warunków (podaną przez Robertsa [19] i rozszerzoną przez Tokudę [22]), jakie powinno spełniać każde narzędzie do wspierania refaktoryzacji.

Rozdział 5 zawiera podsumowanie i opis możliwych sposobów kontynuacji pracy.

W dodatku A przedstawiono spis refaktoryzacji prostych, pochodzący z pracy Williama Opdyke'a [17].

W dodatku B prezentujemy przykładowe refaktoryzacje wraz z pełnym opisem warunków wstępnych potrzebnych do tego, by modyfikacja kodu źródłowego nie zmieniała działania programu.

Rozdział 2

Refaktoryzacja

2.1. Definicja refaktoryzacji

Termin *refaktoryzacja* pojawił sie po raz pierwszy w pracy Williama Opdyke'a i Ralpha Johnsona [16]. Definiowali oni refaktoryzację jako takie przesztalcenie kodu źródłowego, które nie zmienia działania programu. Roberts [19] zrezygnował z wymogu niezmienności działania[1] i widział refaktoryzację jako przekształcenie kodu źródłowego dopuszczalne pod pewnymi warunkami wstępnymi (ang. *preconditions*). Fowler [6] rozszerzył definicję Opdyke'a dodając warunek mówiący, że przeprowadzenie refaktoryzacji ma na celu ulepszenie struktury programu[2].

Pozostali autorzy używają definicji Opdyke uzupełnionej o warunki wstępne [5, 14, 21].

2.2. Przykłady

W tym punkcie podamy dwa krótkie przykłady refaktoryzacji. Dużą liczbę przykładów można znaleźć także w [5, 6, 14, 16, 17, 18, 22, 23]. Pierwszy z podanych przykładów, Wydzielenie Metody, to refaktoryzacja operująca wewnątrz jednej klasy. Bywa ona uznawana za najważniejszą spośród wszystkich refaktoryzacji [6]. Refaktoryzacja Przeniesienie Metody, pokazana w drugim z przykładów, jest innego rodzaju – operuje na dwóch klasach, przemieszczając kod źródłowy pomiędzy nimi.

Przykład 2.2.1 Wydzielenie Metody (ang. *Extract Method*)

Ta refaktoryzacja tworzy nową metodę z wybranego przez użytkownika spójnego fragmentu innej metody. Następnie zastępuje ten fragment przez wywołanie nowej metody. Niezbędne jest ustalenie sygnatury tworzonej metody (nazwy, typów parametrów i typu przekazywanego wyniku), zgłaszanych wyjątków, modyfikatorów itp. Utworzenie jednej metody z fragmentu innej umożliwia uproszczenie kodu i pozwala na ponowne wykorzystanie nowej metody. Wiele przykładów wykorzystania tej refaktoryzacji znajduje się w [6].

W poniższym przykładzie wydzielamy metodę z fragmentu składającego się z wierszy oznaczonych przez `/**/` (oznaczenie to usunięto, dla uproszczenia, z kodu powstałego w wyniku refaktoryzacji).

[1] Roberts zauważył, że jeżeli weźmiemy pod uwagę także czas działania programu wówczas bardzo trudno mówić o niezmienności działania – prawie każda modyfikacja kodu wpływa na szybkość działania programu.

[2] Fowler nie precyzuje kryteriów, na podstawie których ocenia się, czy dane przekształcenie ulepsza strukturę programu.

```
    abstract class A{
        private List x;
        abstract double s() throws Exception;
        abstract void w1();
        abstract void w2(double d);
        void m(double i) throws Exception{
            w1();
            double result= i;             /**/
            Iterator iter= x.iterator();  /**/
            while (iter.hasNext()){       /**/
                A each= (A) iter.next();  /**/
                result += each.s();       /**/
            }                             /**/
            w2(result);
        }
    }

=>

    abstract class A{
        private List x;
        abstract double s() throws Exception;
        abstract void w1();
        abstract void w2(double d);
        void m(double i) throws Exception{
            w1();
            double result= e(i);
            w2(result);
        }

        double e(double i) throws Exception{
            double result= i;
            Iterator iter= x.iterator();
            while (iter.hasNext()){
                A each= (A) iter.next();
                result += each.s();
            }
            return result;
        }
    }
```

Przykład 2.2.2 Przeniesienie Metody (ang. *Move Method*)

Przeniesienie metody polega na zmianie miejsca jej zdefiniowania. Metodę przenosi się przez usunięcie jej deklaracji z klasy obecnie deklarującej do klasy jednego z pól klasy deklarującej. Często umożliwia to zmniejszenie liczby delegacji.

W poniższym przykładzie metoda `p1.A::m()` zostaje przeniesiona do klasy `p2.B`:

```
    package p1;
    import p2.*;
```

```
    public class A extends NN{
        private B b;
        public int m(){
            return b.next().x() + b.x() + b.y() + n();
        }
        public int n(){
            return super.n() + 1;
        }
    }

    package p2;
    public abstract class B{
        public abstract int x();
        public abstract int y();
        public abstract B next();
    }

=>
    package p1;
    import p2.*;
    public class A extends NN{
        private B b;
        public int m(){
            return b.m(this);
        }
        public int n(){
            return super.n() + 1;
        }
    }

    package p2;
    import p1.*;
    public class B{
        public int m(A a){
            return next().x() + x() + y() + a.n();
        }
        public abstract int x();
        public abstract int y();
        public abstract B next();
    }
```

2.3. Refaktoryzacja a projektowanie i pielęgnacja oprogramowania

W tradycyjnym procesie tworzenia oprogramowania dużą część wysiłku i środków inwestuje się w początkową fazę – analizę. Cały system projektowany jest z góry z założeniem, że wymagania nie zmienią się znacząco podczas jego budowy. W wielu przypadkach założenie to jest trafne. Jednak wtedy, gdy wymagania nieustannie się zmieniają bądź są znacznie niedo-

precyzowane, projektowanie z góry jest niezwykle trudne. Niezbędne jest wówczas stosowanie bardziej elastycznych metodologii.

Jedną z metodologii tworzenia oprogramowania, w której nacisk kładzie się na możliwość zmiany lub dodawania wymagań, jest eXtreme Programming. Proces ten został opisany po raz pierwszy przez Kenta Becka w [2] i jest z powodzeniem stosowany w rosnącej liczbie projektów. Refaktoryzacja leży w jego centrum.

Refaktoryzacja pozwala na bezpieczną modyfikację struktury systemu już po napisaniu kodu. Postępując metodą małych kroków, z których żaden nie zmienia działania programu, można wprowadzać bardzo daleko idące modyfikacje do istniejącego już systemu.

Rozważmy sytuację, w której chcemy dodać do programu nową funkcję. W tym celu możemy po prostu wprowadzić nowy kod. Często okazuje się wtedy, że z każdą wprowadzoną nową funkcją rośnie ilość powielonego kodu, powtarzającej się logiki. Prowadzi to nieuchronnie do degradacji struktury programu. Coraz trudniej wprowadza się każdą następną funkcję.

W uzasadnieniu wykonywania refaktoryzacji kryje się założenie, że opłaca się inwestować czas i wysiłek w takie przekształcenie kodu, że wprowadzenie nowej funkcji jest łatwe i nie powoduje powstawania błędów w innych częściach programu. Takie podejście umożliwia też przyspieszenie tworzenia pierwszych wersji programu – nie trzeba odkładać implementacji do końca fazy projektowania wiedząc, że będzie można później dostosować strukturę systemu do nowych wymagań. W ten sposób można także w znacznym stopniu wyeliminować nadmiarowe projektowanie czyli próbę przygotowania programu na wszystkie wymagania, które wydają się możliwe.

Refaktoryzacja nie eliminuje, rzecz jasna, potrzeby testowania. Możliwie pełny zestaw zautomatyzowanych testów zwiększa pewność poprawności wprowadzanych przekształceń i redukuje ryzyko pojawienia się błędów (a zwłaszcza *ponownego* pojawienia się tych samych błędów). Testowanie i refaktoryzacja pomagają w bardziej efektywnym tworzeniu lepszego jakościowo oprogramowania.

2.4. Sposób opisu refaktoryzacji

Refaktoryzacje opisuje się zazwyczaj w następujący sposób:

Nazwa Jednoznaczna nazwa identyfikująca daną refaktoryzację.

Argumenty Lista nazw argumentów wraz z dziedzinami, z których pochodzą (np. `String`, metoda, typ itp.).

W niniejszej pracy nie będziemy precyzowali definicji dziedzin argumentów. Dokładniejsze określenia znajdują się w [17].

Warunki wstępne Ich spełnienie jest wymagane do poprawności refaktoryzacji. Jeśli warunki wstępne nie są spełnione, a przekształcenie zostanie mimo wszystko wykonane, to program może działać inaczej niż przed transformacją bądź nie będzie się poprawnie kompilował. Warunki wstępne bywają wyrażane przy pomocy formuł języka pierwszego rzędu [5, 17, 19, 22]. Inni autorzy używają opisu słownego [14, 21].

Niekiedy uzupełnia się opis refaktoryzacji przez określenie celu jej przeprowadzenia [6, 22], słowny opis modyfikacji wprowadzanych do programu, a także spis warunków końcowych (ang. *postconditions*), będacych formalnym opisem tego, które własności programu (i w jaki sposób) ulegają zmianie podczas przeprowadzania refaktoryzacji [5, 19].

Także (nieformalne) uzasadnienie poprawności bywa załączane do opisu refaktoryzacji [5, 17].

2.5. Klasyfikacja refaktoryzacji

Refaktoryzacje możemy klasyfikować na kilka sposobów. Podstawowym, pochodzącym z pracy Opdyke'a, sposobem klasyfikacji refaktoryzacji jest ich budowa. Dwa inne podziały zostały podane przez Jacoba Nordę [14]: klasyfikujemy refaktoryzacje według tego, jakie elementy programu są modyfikowane w wyniku ich działania oraz według stopnia automatyzacji.

2.5.1. Podział według budowy

Refaktoryzacje proste Są to refaktoryzacje nieskładające się z innych. Spis refaktoryzacji prostych, wraz z podziałem podanym przez Opdyke'a w [17] zamieszczamy w dodatku A.

Refaktoryzacje złożone Refaktoryzacje złożone powstają jako łańcuch refaktoryzacji prostych. Jeśli żaden element łańcucha nie zmienia działania programu, to również cały łańcuch ma tę własność. Roberts i Cinneide przedstawiają sposoby formalnego określania warunkow wstępnych łańcucha refaktoryzacji. W tym celu używaja warunków końcowych [19, 5] refaktoryzacji składowych.

Opdyke i Johnson podają sposób stworzenia wspólnej, abstrakcyjnej nadklasy dla zbioru klas przez złożenie refaktoryzacji prostych [16]. W innej pracy omawiają przekształcenie zależności hierachicznej (ang. *inheritance relationship*) w zależność posiadania (ang. *aggregation relationship*) i odwrotnie [18]. Tokuda oraz Cinneide przedstawiają sposoby wprowadzania (przy użyciu refaktoryzacji) do istniejącego systemu wzorców projektowych (ang. *design patterns*) [5, 22]. Fowler prezentuje wiele przykładów składania refaktoryzacji w celu zwiększenia czytelności kodu [6]. Tokuda i Batory opisują jak, składając ok. 800 refaktoryzacji prostych, przeprowadzono skomplikowaną restrukturyzację dużego systemu (ponad 500000 wierszy kodu C++). Interesującą cechą ich pracy jest to, że zaprezentowali, w jaki sposób można odtworzyć przeprowadzoną w rzeczywistości restrukturyzację przy użyciu refaktoryzacji. Autorzy oceniają, że dzięki użyciu narzędzi wspomagajacych proces refaktoryzacji można ok. dziesięciokrotnie przyspieszyć ewolucję struktury systemu [22].

2.5.2. Podział według obszaru działania

Podział według obiektu działania, podany przez Nordę w [14], jest nieco arbitralny i niekiedy nie jest jasne, do której kategorii zakwalifikować daną refaktoryzację. Przykładowo, refaktoryzacja Wydzielenie Metody tworzy nową metodę, modyfikując w ten sposób obiektową strukturę programu.

Refaktoryzacje proceduralne Są to modyfikacje kodu wewnątrz jednej metody. Typowe operacje to Wydzielenie Metody, Zmiana Nazwy Parametru Metody. Użycie refaktoryzacji Wydzielenie Metody zaprezentowano w przykładzie 2.2.1.

Refaktoryzacje obiektowe Zmieniają one strukturę obiektową programu (układ klas, metod, pakietów itp). Przykładowa refaktoryzacja to Przeniesienie Metody (przykład 2.2.2) lub np. Przeniesienie Pola do Nadklasy (polegająca na usunięciu pola z danej klasy i umieszczeniu go w nadklasie).

Refaktoryzacje pojęciowe Zmieniają one bądź upraszczają logiczną strukturę programu. Przykładami są refaktoryzacje polegające na wprowadzeniu wzorca projektowego [22].

2.5.3. Podział według stopnia automatyzacji

Norda podaje w [14] następujący podział refaktoryzacji według stopnia automatyzacji.

Refaktoryzacje wykonywane ręcznie Użytkownik decyduje kiedy i jakie refaktoryzacje należy przeprowadzić i sam modyfikuje kod źródłowy programu. Duży katalog takich refaktoryzacji znajduje się w książce Fowlera [6].

Refaktoryzacje wspomagane automatycznie Użytkownik decyduje kiedy i jakie refaktoryzacje należy przeprowadzić. Następnie używa narzędzia do ich przeprowadzenia. To podejście jest dla nas najbardziej interesujące – jego omówienie znajduje się w punkcie 2.7.

Refaktoryzacje w pełni zautomatyzowane Refaktoryzacje są wykonywane automatycznie przez narzędzie po wykryciu potrzeby ich przeprowadzenia. Użytkownikowi pozostawia się możliwość korekty decyzji podjętych przez program.

Ivan Moore jest twórcą narzędzia o nazwie Guru, służącego do w pełni automatycznej refaktoryzacji programów w języku Self. Program Guru usuwa nieużywane części programu, redukuje ilość powtarzającego się kodu. Autor zauważa, że, w celu uzyskania sensownych wyników, zazwyczaj niezbędna jest pewna interwencja użytkownika [12, 13]. Ponadto przeprowadzone modyfikacje, mimo, że redukują ilość kodu, często sprawiają, że powstały w ich wyniku kod źródłowy jest znacznie mniej zrozumiały dla programistów.

Podobnych obserwacji dokonał Eduardo Cassais, autor narzędzia służącego do automatycznej restrukturyzacji programów w języku Eiffel [4].

W niniejszej pracy jest mowa o wspieranych automatycznie refaktoryzacjach prostych. Większość przedstawionych tu refaktoryzacji to refaktoryzacje obiektowe, jednakże część (jak Wydzielenie Metody) to refaktoryzacje proceduralne.

2.6. Uzasadnianie poprawności refaktoryzacji

W ogólnym przypadku udowodnienie, że dwa programy zawsze zachowują się tak samo, jest niemożliwe. Języki takie jak Java czy C++ dopuszczają formuły pierwszego rzędu jako warunki logiczne w instrukcjach iteracyjnych, a problem sprawdzenia równoważności dwóch formuł pierwszego rzędu jest nierozstrzygalny. Nasze oczekiwania co do formalnego uzasadniania poprawności refaktoryzacji (czyli tego, że dana modyfikacja kodu źródłowego nie zmienia działania programu) muszą więc pozostać dość skromne.

Badania w dziedzinie refaktoryzacji opierają się na nieformalnym uzasadnieniu poprawności [5, 14, 17, 21, 22] lub zupełnie pomijają tę kwestię [6, 19].

Według Cinneide [5], nieformalne uzasadnianie poprawności jest odpowiednim podejściem, ponieważ do pewnego stopnia odpowiada temu, co doświadczony programista robi w praktyce przed ręcznym przeprowadzeniem refaktoryzacji. Jest to interesujące, praktyczne podejscie pośrednie – pomiędzy formalnymi dowodami poprawności, a całkowitym ignorowaniem problemu. Budując takie nieformalne (lecz ustrukturyzowane) uzasadnienie tego, że przepowadzana modyfikacja kodu źródłowego nie zmieni działania programu, upewniamy się, że wprowadzane przez nas zmiany w kodzie są istotnie refaktoryzacjami. Jeśli następnie, podczas

testów, mimo wszystko odkryjemy zmianę działania programu, to możemy znaleźć i poprawić odpowiedzialny za to błąd w uzasadnieniu.

2.7. Refaktoryzacja wspomagana automatycznie

Dotychczas omawialiśmy refaktoryzacje w ogólnym ujęciu. Najbardziej interesująca jest jednak dla nas refaktoryzacja wspierana przez narzędzia. Automatyzacja procesu przekształcania programów jest bardzo pożądana – kiedy zdecydujemy już jaką refaktoryzację przeprowadzić, wówczas to, co pozostaje, to sprawdzenie warunków wstępnych i wprowadzenie zmian w kodzie. Oczywiste jest, że tego typu czynności – powtarzalne, często wykonywane, żmudne, wymagające precyzji i podatne na błędy – powinny być, gdzie to tylko możliwe, zautomatyzowane.

Podstawy automatyzacji procesu refaktoryzacji są zawarte w pracach Opdyke'a i Robertsa. Opdyke opisał zbiór refaktoryzacji programów w języku C++ [17], a następnie, w kolejnych pracach, pokazał w jaki sposób można je składać w celu stworzenia refaktoryzacji wyższego poziomu, np. przekształcenie zależności hierachicznej (ang. *inheritance relationship*) w zależność posiadania (ang. *aggregation relationship*) i odwrotnie [18]. Poprawność opisywanych refaktoryzacji Opdyke uzasadniał nieformalnie. W tym celu określił, przedstawione w [17], siedem własności programów, zachowywanych przez jego przekształcenia. Na tej podstawie wnioskował o niezmienności działania programu przed i po przeprowadzeniu modyfikacji kodu.

Praca Robertsa [19] rozszerza wyniki Opdyke'a przez wprowadzenie bardziej formalnego ujęcia składania refaktoryzacji, a także wykorzystanie dynamicznej informacji o programie[3].

Tokuda i Batory zajęli się implementacją refaktoryzacji opisanych w pracy Opdyke'a[4]. Odkryli niewystarczalność warunków podanych przez Opdyke i określił kilka nowych, których zachowanie jest konieczne do uzyskania niezmienności działania przed i po modyfikacji programu [22, 23].

System *Elbereth*, opisany w pracy Kormana [11], służy do tworzenia diagramów gwiazdowych (ang. *star diagrams*), opisanych w pracy [3]. Diagram gwiazdowy prezentuje sposób, w jaki dany element, np. pole lub metoda, jest używany w programie. Taka informacja może służyć jako punkt wyjścia do przeprowadzenia refaktoryzacji. *Elbereth* nie pozwala jednak na manipulację diagramami – służy jedynie jako narzędzie do graficznej prezentacji struktury kodu źródłowego.

Tichelaar podaje meta–model służący do opisu procesu refaktoryzacji niezależnego od języka programowania. Następnie używa go do stworzenia prototypu narzędzia do refaktoryzacji programów w językach Smalltalk i Java [21]. W odróżnieniu od nas, autor ograniczył się do rozważań pewnego, niesprecyzowanego jawnie, podzbioru języka Java. Analiza warunków wstępnych refaktoryzacji jest, wobec tego, znacznie uproszczona w stosunku do zawartej w niniejszej pracy.

Seguin [20] rozważa problemy, jakie stwarza refaktoryzacja programów w silnie typowanym języku, takim jak Java, w odróżnieniu od języka beztypowego, jak Smalltalk. Artykuł Seguina dotyczy podobnej tematyki, co niniejsza praca i ilustruje rodzaje problemów powstających przy refaktoryzacji kodu w języku o tak złożonej strukturze jak Java. Autor ogranicza się jednak do omówienia tylko jednej refaktoryzacji – Przesunięcia Pola do Nadklasy (ang. *push up field*).

[3] Język Smalltalk, którym zajmował się Roberts jest beztypowy, wobec czego wiele informacji o programie jest niedostępnych statycznie. Roberts opisał w jaki sposób wykorzystać informacje zgromadzone dynamicznie do analizy i modyfikacji programu.

[4] Stworzone przez nich narzędzie nie jest publicznie dostępne.

Cinneide zajął się refaktoryzacjami polegającymi na wprowadzaniu wzorców projektowych do istniejących programów w języku Java [5]. Opisał także i zaimplementował kilka refaktoryzacji prostych. Autor wyłączył z rozważań dużą część języka Java. Najbardziej znaczące ograniczenia polegają na nieuwzględnieniu typów zagnieżdżonych, przeciążania nazw metod oraz przestrzeni nazw typów (autor założył, że każdy typ w systemie ma unikatową nazwę).

Norda opisuje projekt i implementację systemu JREB, służącego do refaktoryzacji programów w języku Java [14]. Jego podejście różni się od naszego w tym względzie, że Norda skupia się na implementacji systemu, nie omawiając szczegółowo sprawdzanych warunków wstępnych. Norda wyłącza ze swych rozważań znaczną część języka – przede wszystkim interfejsy, a także metody i pola statyczne. Krótki opis systemu JREB i jego ograniczeń podajemy w punkcie 2.8.3.

2.8. Istniejące narzędzia do wspomagania refaktoryzacji

W tym punkcie prezentujemy krótki przegląd istniejących narzędzi służących do modyfikacji kodu i refaktoryzacji. Celem tego przeglądu nie jest porównanie istniejących rozwiązań z naszym. Chcemy raczej zaprezentować obecnie istniejące wsparcie narzędziowe dla procesu refaktoryzacji. Zobaczymy, że jest to nadal bardzo nowa dziedzina badań i istniejące rozwiązania są, w większości, niewystarczające.

Przede wszystkim należy zwrócić uwagę na to, że wiele istniejących narzędzi ignoruje bądź traktuje pobieżnie analizę warunków wstępnych refaktoryzacji – z wyróżniającym się wyjątkiem w postaci The Smalltalk Refactoring Browser. Nie można ich więc w pełni poprawnie nazwać narzędziami do refaktoryzacji – są to raczej narzędzia do modyfikacji kodu. Mimo, że wyręczają programistę w żmudnych i podatnych na błędy zadaniach modyfikacji kodu, to nie dają żadnych lub prawie żadnych gwarancji poprawności przeprowadzanych przekształceń. Jak zauważa Norda w [14] takie wsparcie może być wystarczające przy pracy nad małymi projektami, gdzie programista zna dużą część kodu. Jeżeli jednak narzędzie modyfikuje dziesiątki, a nawet setki plików z kodem źródłowym, to użytkownik nie ma realnych szans na weryfikację wszystkich wprowadzanych zmian. Jak podajemy w rozdziale 2 pewne rodzaje możliwych błędow są nie do wykrycia przez kompilator. Powstają wtedy sytuacje, gdzie program nadal kompiluje się poprawnie (co daje użytkownikowi poczucie bezpieczeństwa), jednak jego działanie się zmienia. Szybkie wykrycie zmiany działania (czyli wprowadzonego błędu) w rzadko używanym podsystemie dużego programu jest prawie niemożliwe.

Niektóre z zaprezentowanych narzędzi były dla nas niedostępne podczas przygotowania niniejszego przeglądu[5] (jFactor, XRef Speller, IntelliJ Renamer, Transmogrify). Przy ich opisie opieramy się na dostępnej dokumentacji programu, a także na opracowaniu sporządzonym przez Nordę w [14] (luty 2001 r.).

Nazwy refaktoryzacji w oryginalnym brzmieniu podajemy tylko przy pierwszym wystąpieniu.

2.8.1. The Smalltalk Refactoring Browser

The Smalltalk Refactoring Browser jest najszerzej używanym z istniejących narzędzi wspomagających proces refaktoryzacji. Powstał na University of Illinois at Urbana–Champaign w roku 1994. Jest zintegrowany z dwoma środowiskami programistycznymi języka Smalltalk

[5]Ze względów finansowych, a także prawnych – licencje tych programów zawierają warunki, których autor, będąc stażystą w firmie tworzącej produkt konkurencyjny w stosunku do omawianych, nie mógł spełnić.

– VisualWorks i VisualAge for Smalltalk. Lista wspieranych refaktoryzacji (ok. 30) znajduje się w [19]. Narzędzie jest dostępne bezpłatnie wraz z kodem źródłowym pod adresem [28].

Mimo, że jest to najbardziej znane i najczęściej używane narzędzie do refaktoryzacji, nie poświęcimy mu więcej miejsca, ponieważ służy ono jedynie do refaktoryzacji programów w języku Smalltalk. Narzędzia przedstawione w dalszych punktach tego zestawienia są przeznaczone do przekształcania programów w języku Java.

2.8.2. jFactor

Program jFactor jest zintegrowany ze środowiskiem programistycznym VisualAge for Java firmy IBM.

W chwili obecnej wspiera nastepujące refaktoryzacje:

- Wydzielenie Metody (ang. *Extract Method*)
- Wydzielenie Zmiennej (ang. *Extract Temp*)
- Rozwinięcie Wywołania Metody (ang. *Inline Method*)
- Przesunięcie Metody do Pod/Nadklasy (ang. *Push Method Down/Up*)
- Dodanie Parametru do Metody (ang. *Add Method Parameter*)
- Usunięcie Parametru Metody (ang. *Remove Method Parameter*)
- Zmiana Kolejności Parametrów Metody (ang. *Rearrange Parameters*)
- Przesunięcie Pola do Pod/Nadklasy (ang. *Push Field Down/Up*)
- Zmiana Bezpośrednich Odniesień do Pola na Odniesienia Używające Metod Dostępowych (ang. *Abstract Field*)
- Wydzielenie Interfejsu (ang. *Extract Interface*)
- Wydzielenie Nadklasy (ang. *Extract Superclass*)
- Zmiana Nazwy
  - Metody (ang. *Rename Method*)
  - Pola (ang. *Rename Field*)
  - Zmiennej Lokalnej (ang. *Rename Local Variable*)

Jest to produkt komercyjny, dostępny u producenta pod adresem [26].

W trakcie pisania niniejszej pracy nie mieliśmy dostępu do tego narzędzia. Nie możemy wobec tego ocenić jego niezawodności.

2.8.3. JREB

JREB to narzędzie przeznaczone do refaktoryzacji programów w Javie, będące częścią pracy Jakoba Nordy [14]. Interfejs użytkownika jest zintegrowany ze środowiskiem programistycznym JBuilder.

W chwili obecnej jedyne wspierane refaktoryzacje to:

- Zmiana Nazwy

  - Zmiennej Lokalnej
  - Parametru Metody
  - Pola
  - Metody
  - Klasy
- Zmiana Położenia Elementu Programu (pola, metody)
- Zmiana Bezpośrednich Odniesień do Pola na Odniesienia Używające Metod Dostępowych

Najbardziej zauważalne ograniczenie programu JREB polega na tym, że nie obejmuje on całości języka Java – w szczególności interfejsów oraz metod i pól statycznych. Zmiana nazwy metody zadeklarowanej w interfejsie nie powoduje modyfikacji deklaracji w klasach implementujacych ten interfejs (co prowadzi do błędów kompilacji). Inne ograniczenie polega na tym, że refaktoryzowany program nie może zawierać błędów kompilacji. Ponadto pewne warunki wstępne pozostają niesprawdzone – przykładowo nieprzesłanianie widoczności pól po zmianie ich nazw. JREB zupełnie ignoruje również niebezpieczeństwa związane z zaciemnianiem nazw (por. punkt 3.3), a także ze zmianą nazw lub miejsc deklaracji metod natywnych (por. punkt 3.5).

JREB przed wykonaniem każdej refaktoryzacji wczytuje cały modyfikowany program do pamięci i buduje dla niego drzewo składni. To podejście, jak podaje sam autor, prawdopodobnie ogranicza stosowanie JREB tylko do małych programów.

Program został wykonany na zlecenie firmy Ericsson i nie jest publicznie dostępny.

2.8.4. XRef Speller

XRef Speller to narzędzie zintegrowane ze środowiskami Emacs, XEmacs i Kawa. Służy do refaktoryzacji programów w językach C i Java. Jest to komercyjne narzędzie, dostępne u producenta pod adresem [20].

Wspierane refaktoryzacje (w języku Java) to:

- Dodanie Parametru do Metody
- Usunięcie Parametru Metody
- Zmiana Kolejności Parametrów Metody
- Wydzielenie Metody
- Zmiana Nazwy
  - Klasy
  - Pola
  - Metody
  - Zmiennej Lokalnej

W trakcie pisania niniejszej pracy nie mieliśmy dostępu do tego narzędzia. Nie możemy wobec tego ocenić jego niezawodności.

2.8.5. JRefactory

JRefactory to bezpłatnie dostępny (wraz z kodem źródłowym) program autorstwa Chrisa Seguina. Może być używany samodzielnie lub w integracji ze środowiskami programistycznymi Elixir oraz JBuilder.

To narzędzie przeprowadza tylko pewną część analizy warunków wstępnych, niezbędnej do zachowania działania programu. Nie jest to więc, w ścisłym znaczeniu, narzędzie do refaktoryzacji, a raczej jedynie do modyfikacji kodu źródłowego.

Obecnie wspierane są następujące modyfikacje:

- Zmiana Położenia Klasy
- Zmiana Nazwy Klasy
- Stworzenie Abstrakcyjnej Nadklasy
- Stworzenie Podklasy
- Usunięcie Klasy
- Przesunięcie Pola Obiektu do Pod/Nadklasy
- Przesunięcie Metody do Pod/Nadklasy
- Zmiana Położenia Metody

Program JRefactory jest dostępny pod adresem [27].

2.8.6. IntelliJ Renamer

IntelliJ Renamer przeznaczony jest głównie do przeprowadzania zmian nazw elementów programów, choć wspiera także przemieszczanie pewnych części programu. To narzędzie nie przeprowadza żadnej analizy warunków wstępnych – nie jest to więc program do refaktoryzacji – służy jedynie do modyfikacji kodu. Jest zintegrowany ze środowiskiem programistycznym IDEA.

Obecnie wykonywane modyfikacje to:

- Wydzielenie Metody
- Zmiana Nazwy
  - Pakietu
  - Klasy
  - Metody
  - Pola
- Zmiana Położenia
  - Pakietu
  - Klasy

Jest to komercyjne narzędzie, dostępne u producenta pod adresem [25].

2.8.7. Transmogrify

Transmogrify to bezpłatnie dostępne (wraz z kodem źródłowym) narzędzie do analizy i modyfikacji kodu źródłowego programów w języku Java. Jest ono nakładką na środowiska programistyczne JBuilder oraz Forte4Java.

Refaktoryzacje, których można za jego pomocą dokonać to:

- Zmiana Nazwy
  - Zmiennej Lokalnej
  - Metody
- Wydzielenie Metody
- Zmiana Odwołań do Zmiennej Lokalnej na Wywołanie Metody
- Rozwinięcie Definicji Zmiennej Lokalnej
- Przemieszczenie Pola do Nadklasy

Narzędzie to, podobnie jak program JREB (opisany w punkcie 2.8.3), ma znacznie ograniczoną skalowalność co wynika z faktu, że cały refaktoryzowany program musi zostać sparsowany i wczytany do pamięci. Inne ograniczenie znacznie utrudniające wykorzystanie Transmogrify w dużych projektach wynika z tego, że refaktoryzowany program musi być w całości pozbawiony błędów kompilacji (nawet w częściach, które nie są ani analizowane, ani modyfikowane).

Jak podają autorzy, Transmogrify jest nadal w fazie tworzenia i nie radzi sobie ze wszystkimi zawiłościami języka Java.

Program jest dostępny po adresem [29].

2.8.8. Podsumowanie przeglądu narzędzi

Z przedstawionego przeglądu istniejących narzędzi do automatycznego wspomagania refaktoryzacji wynika, że ta dziedzina oprogramowania znajduje się nadal w początkowej fazie rozwoju.

Możemy wyróżnić kilka podstawowych problemów związanych z istniejącymi narzędziami.

a. *Niewystarczająca weryfikacja warunków wstępnych*

   Zadaniem narzędzia do refaktoryzacji jest takie przeprowadzenie żądanej modyfikacji kodu, by działanie programu pozostało niezmienione[6]. Kilka z wymienionych narzędzi (przykładowo JRefactory, IntelliJ Renamer) traktuje pobieżnie analizę warunków wstępnych refaktoryzacji, co sprawia, że w rezultacie narzędzie modyfikuje działanie programu w nieprzewidywalny sposób. Naszym zdaniem wyklucza to zastosowanie takich narzędzi do programów nawet średniej wielkości.

b. *Ograniczenia funkcjonalności związane z nieuwzględnieniem całości języka*

[6] Użytkownik powinien mieć możliwość modyfikacji programu w dowolny pożądany przez siebie sposób. Jednakże zawsze wszelka zmiana działania programu powinna być wykonywana jedynie na wyraźne polecenie użytkownika.

Niektóre z zaprezentowanych narzędzi (JREB, Transmogrify) nie uwzględniają całości języka programowania. Przekłada się to na niewystarczającą analizę warunków wstępnych, czego bezpośrednim skutkiem jest nieprzewidywalność wprowadzanych przez narzędzie zmian w kodzie źródłowym programu. Wspomniane narzędzia zazwyczaj niejawnie ignorują istnienie pewnych konstrukcji języka. Użytkownik korzystający z takiego narzędzia nie jest więc ostrzegany o możliwości wystąpienia zmian w działaniu programu.

c. *Ograniczenia skalowalności*

   Pewna grupa przedstawionych narzędzi (JREB, Transmogrify) ma znacznie ograniczoną skalowalność ze względu na to, że w celu dokonania analizy i modyfikacji kodu narzędzia te wczytują do pamięci cały refaktoryzowany program.

   Wspomagana automatycznie refaktoryzacja jest pomocna przy pracy nad dowolnym programem. Jednakże narzędzia takie najbardziej przydają się przy pracy z dużymi, złożonymi, projektami – gdzie ręczna modyfikacja programu jest prawie niemożliwa. Podane ograniczenie bardzo utrudnia, a wręcz uniemożliwa, użycie tych narzędzi tam, gdzie byłyby najbardziej potrzebne.

d. *Brak możliwości łatwego wycofania przeprowadzonych zmian*

   Niektóre z wymienionych narzędzi (jFactor, JREB, XRef Speller) nie oferują możliwości łatwego wycofania zmian. Znacznie utrudnia to korzystanie z nich w najbardziej pożądany przez użytkowników sposób, polegający na przeprowadzeniu kilku następujących po sobie refaktoryzacji[7], ocenie (subiektywnej lub wspomaganej narzędziami) ich wpływu na budowę programu i ewentualnym wycofaniu wprowadzonych zmian jeżeli struktura programu nie uległa poprawie.

e. *Niewielka liczba wspomaganych refaktoryzacji* Katalog Opdyke'a (przytoczony w Dodatku A) zawiera ponad 20 refaktoryzacji, książka Fowlera opisuje ponad 80, najlepsze narzędzie wspomagające refaktoryzację, The Smalltalk Refactoring Browser (p. 2.8.1), wspiera ponad 30. Natomiast żadne z obecnie istniejących i znanych nam narzędzi do refaktoryzacji programów w języku Java nie zawiera wsparcia dla więcej niż 15 refaktoryzacji (średnio poniżej 10).

Dwa pozostałe problemy są bardziej ogólnej natury.

a. *Brak wsparcia dla refaktoryzacji w środowiskach programistycznych*

   Żadne z najważniejszych, istniejących na rynku, środowisk programistycznych (JBuilder, Forte4Java, Visual Age for Java) nie zawiera obecnie wsparcia dla refaktoryzacji[8]. Wszystkie przedstawione narzędzia są jedynie nakładkami. Ogranicza to stopień ich integracji ze środowiskiem.

b. *Ograniczenia w dostępie do narzędzi*

   Kilka opisanych programów do modyfikacji kodu i refaktoryzacji (jFactor, JREB, XRef Speller, IntelliJ Renamer) to narzędzia komercyjne, co nie tylko utrudnia ich wykorzystanie przez szersze grono zainteresowanych programistów, lecz także uniemożliwia ich rozszerzanie przez implementację nowych refaktoryzacji.

[7] Składanie refaktoryzacji to najważniejszy sposób ich wykorzystanie, co omówiono krótko w punkcie 2.5.1.

[8] Jedyne, znane nam, środowisko zawierające wsparcie dla refaktoryzacji to IDEA firmy IntelliJ. Krótki opis programu Renamer używanego w tym środowisku znajduje się w punkcie 2.8.6.

W rozdziale 4 omówione zostało nasze podejście do wymienionych tu problemów i sposób, w jaki rozwiązaliśmy je[9] przy konstrukcji naszego narzędzia.

[9]Stworzone przez nas narzędzie, którego opis stanowi treść rozdziału 4 zawiera rozwiązanie wszystkich wskazanych tu problemów poza liczbą wspieranych refaktoryzacji – jak podano w rozdziale 5 jest to najważniejszy obszar dalszych prac.

Rozdział 3

Refaktoryzacja programów w języku Java

Nie jest możliwa analiza wpływu, jaki specyficzne cechy języka Java wywierają na wszystkie istniejące refaktoryzacje. Liczba wszystkich takich przekształceń programu, które nie zmieniają jego działania jest nieograniczona. W dodatku A przedstawiamy zwięzłą klasyfikację refaktoryzacji prostych, z których można składać refaktoryzacje dowolnie złożone. Istnieje jednakże znaczna liczba refaktoryzacji prostych nie mieszczących się w ramach przedstawionej tam klasyfikacji – przykładową listę ponad 80 można znaleźć w książce Fowlera [6]. Stopień możliwej automatyzacji większości tych przekształceń nie został jednak dotychczas zbadany.

W poszczególnych punktach tego rozdziału będziemy przedstawiać jakie warunki wstępne powstają jako konsekwencja opisywanej cechy języka. Zazwyczaj ograniczymy się do podania warunków wstępnych dla kilku refaktoryzacji prostych. Omawiane przez nas cechy języka mają jednak wpływ na wszystkie bądź prawie wszystkie znane z literatury refaktoryzacje, których tu, z braku miejsca, nie omawiamy.

Celem niniejszego rozdziału jest rozwinięcie istniejących prac w dziedzinie refaktoryzacji o analizę wpływu specyficznych cech języka Java na proces refaktoryzacji. Przedstawiamy i omawiamy „strefy problemowe" – czyli takie obszary języka, gdzie nawet pozornie łatwe i bezpieczne przekształcenia programu mogą zmienić jego działanie. Wskazane jest, by przyszli projektanci narzędzi do refaktoryzacji zdawali sobie sprawę z wielu pułapek języka opisanych szczegółowo w tej pracy. Jest to niezbędne do powstania narzędzi na tyle wszechstronnych i niezawodnych, by mogły być używane na codzień przez zawodowych programistów.

Częścią tej pracy jest opis projektu i implementacji narzędzia wspomagającego refaktoryzację (zawarty w rozdziale 4), przy konstrukcji którego staraliśmy się wziąć pod uwagę wszystkie opisane tu obszary problemowe. W przypadku wielu z nich nasze narzędzie jest jedynym istniejącym (według naszych danych), które je uwzględnia.

Większość przykładów w tym rozdziale została celowo skonstruowana w taki sposób, by zilustrować sytuacje, w których, po wykonaniu przekształcenia, program nadal kompiluje się bez błędów, jednak jego działanie ulega zmianie. Występowanie takich przypadków jest bowiem przyczyną najtrudniejszych do wykrycia problemów.

Rzecz jasna bardzo znaczna liczba cech języka ma wpływ na refaktoryzację, a w szczególności na analizę warunków wstępnych. Przy tworzeniu spisu cech zawartego w tym rozdziale pominęliśmy lub wspomnieliśmy jedynie krótko o tych, które są już wystarczająco szczegółowo omówione w istniejących publikacjach. Staraliśmy się wskazać takie, które są, naszym zdaniem, istotne i nieuwzględniane przez znane nam narzędzia.

Niniejszy rozdział, rozdział 4 oraz dodatek B przedstawiają główny wkład merytoryczny

tej pracy.

3.1. Wielokrotne dziedziczenie interfejsów oraz efekt fali

Zanim omówimy trudności związane z refaktoryzacją programów, w których używa się wielokrotnego dziedziczenia interfejsów, krótko przedstawimy najważniejsze właściwości mechanizmu interfejsów. Szczegółowy opis znajduje się w specyfikacji języka [8].

Interfejs w języku Java może rozszerzać (ang. *extend*) dowolną liczbę innych interfejsów (cykle są niedozwolone). Analogicznie do terminologii używanej przy hierarchii klas mówimy wtedy, że interejs rozszerzany jest nadinterfejsem interfejsu rozszerzającego (który wobec tego jest jego *pod*interfejsem). Wszystkie interfejsy w programie tworzą wiec zbiór parami rozłącznych skierowanych grafów acyklicznych (kierunek strzałek jest zgodny z rozszerzaniem). Jeżeli klasa implementuje interfejs, wówczas każda jej nieabstrakcyjna podklasa musi deklarować (bądź dziedziczyć z nadklas) wszystkie metody zadeklarowane w tym interfejsie oraz jego nadinterfejsach (bezpośrednich i pośrednich). Specyfikacja języka określa ponadto, że metody te muszą być instancyjne i publiczne. Podobnie, implementowanie wielu interfejsów oznacza, że każda nieabstrakcyjna podklasa danej klasy musi deklarować bądź dziedziczyć z nadklas wszystkie metody (w sensie sumy teoriomnogościowej) zadeklarowane we wszystkich implementowanych interfejsach i ich nadinterfejsach.

Okazuje się, że wiele zagadnień refaktoryzacji jest znacznie prostszych w językach, w których nie występuje wielokrotne dziedziczenie.

W tym punkcie ograniczymy się do rozważenia tylko jednej refaktoryzacji, mianowicie zmiany nazwy metody. W języku Smalltalk oraz w podzbiorach języka Java, w których istnieje tylko pojedyncze dziedziczenie, stosuje się następujący algorytm (dana jest metoda, typ, w którym jest zadeklarowana i nowa nazwa metody):

1. Sprawdź, czy metoda o nowej nazwie (sygnaturze) nie jest zadeklarowana w hierarchii danego typu – jeśli tak, to zmiana nazwy nie jest możliwa.

2. Znajdź najbardziej abstrakcyjny typ deklarujący daną metodę.

3. Zmień nazwę metody we wszystkich podtypach znalezionego typu.

Algorytm ten jest niewystarczający w pełnym języku Java[1], co ilustruje poniższy przykład:

```
interface I1{
    public void m();
}
interface I2{
    public void m();
}
class A implements I1, I2{
    public void m(){}
}
```

Zmiana nazwy metody `I1::m()` musi pociągać za sobą zmianę nazwy `A::m()` – w przeciwnym razie powstanie błąd kompilacji. Zauważmy także, że również nazwa `I2::m()` musi ulec zmianie.

[1] Jest tak nie tylko ze względu na wielokrotne dziedziczenie, lecz także ze względu na metody niewirtualne oraz typy zagnieżdżone.

Jak widać zmiana nazwy metody jest przenoszona po grafie typów. Nazwaliśmy to zjawisko efektem fali (ang. *ripple effect*). Taka fala może się rozchodzić po wielu klasach i interfejsach. Algorytm wyszukiwania metod, których nazwy należy zmienić, powinien przeglądać graf typów w górę i w dół począwszy od maksymalnie abstrakcyjnego typu deklarującego daną metodę.

Zanim naszkicujemy algorytm znajdujący interesujący nas zbiór metod należy zauważyć, że:

1. Nie wszystkie typy wzdłuż fali muszą deklarować metodę o sygnaturze takiej, jak sygnatura metody, której nazwę chcemy zmienić. Rozważmy następujący przykład pokazujący, że wystarczy, by metoda taka była odziedziczona z nadklasy:

   ```
   interface I{
       public void m();
   }
   class A{
       public void m(){}
   }
   class B extends A implements I{
   }
   ```

   Po zmianie nazwy metody `I::m()` również nazwa metody `A::m()` musi zostać zmieniona.

2. W pewnych sytuacjach fala zatrzymuje się. Metody niewirtualne nie zastępują (ang. *override*) metod wirtualnych i wobec tego zatrzymują falę. Taką sytuację ilustruje następujący przykład:

   ```
   interface I{
       public void m();
   }
   class A{
       private void m(){}
   }
   class B extends A implements I{
       public void m(){}
   }
   ```

   Zmianie nazwy metody `I::m()` musi towarzyszyć zmiana nazwy metody `B::m()`, natomiast metoda `A::m()` może pozostać nieprzemianowana.

Punkt 2 wymaga wyjaśnienia. Przy wywołaniu metody wirtualnej, konkretna metoda jest ustalana podczas działania programu, na podstawie faktycznego typu (ang. *runtime type*) obiektu, na rzecz którego jest wywoływana. W przypadku metod niewirtualnych jest to ustalane zawczasu, podczas kompilacji. Metody niewirtualne (w języku Java wszystkie metody statyczne, a także prywatne są niewirtualne) stosuje się bardzo często (ze względu na przejrzystość, a także wydajność, powstającego dzięki temu kodu) – jeśli chcemy zbudować narzędzie do refaktoryzacji dużych programów, to musimy wziąć je pod uwagę.

W tym punkcie rozważymy jedynie metody prywatne. Zbiór metod statycznych stanowi, do pewnego stopnia, przestrzeń rozłączną ze zbiorem metod instancyjnych – w tym sensie, że metoda statyczna nie może ukrywać (ang. *hide*) metody instancyjnej z nadklasy. Podobnie, metoda instancyjna nie może zastąpić metody statycznej z nadklasy.

Ponieważ w przypadku metod niewirtualnych kompilator wie, którą dokładnie metodę należy wywołać, wobec tego zmiana nazwy takiej metody nie musi pociągać za sobą zmiany nazwy żadnej innej metody. Spójrzmy na przykład:

```
class A{
    private void m(){}
}
class B extends A{
    public void m(){}
}
```

Metody `A::m()` oraz `B::m()` są jednoznacznie odróżniane podczas kompilacji dla każdego wywołania. Oznacza to, że są to, w pewnym sensie, dwie niezależne metody[2]. Wobec tego zmiana nazwy jednej z nich nie powoduje konieczności zmiany nazwy drugiej.

W jaki sposób wobec tego ustalić zbiór metod, których nazwy muszą zostać zmienione razem z nazwą danej metody? Jedno z możliwych podejść jest następujące: należy zmienić nazwy wszystkich metod o tej samej sygnaturze, co dana metoda. O ile jest to możliwe, tzn. jeśli metoda o nowej sygnaturze nie istnieje w żadnym z deklarujących typów (ani, dodatkowo, w ich grafach dziedziczenia i hierarchiach zawierania), to taka zmiana będzie istotnie refaktoryzacją, czyli nie będzie wpływać na działanie systemu. To podejście wydaje się jednak nie do przyjęcia – jako jaskrawo sprzeczne z intuicją i oczekiwaniami użytkowników. Powinniśmy wobec tego ustalić możliwie mały zbiór takich metod. Nieco dokładniej, szukamy takiego zbioru metod (o tej samej sygnaturze i typie wyniku), że:

a. Zmiana nazw wszystkich metod z tego zbioru nie wpływa na działanie programu.

b. Znaleziony zbiór jest minimalny.

W następnym punkcie prezentujemy szukany algorytm po czym omawiamy możliwe jego użycie w przypadku innych refaktoryzacji.

Algorytm znajdowania metod znajdujących się wzdłuż fali

W tym punkcie podajemy algorytm znajdowania szukanego zbioru metod. Dokładniej, zbioru typów deklarujących szukane metody.
Dane: typ `T`, metoda `m`
Wynik: zmienna `result` zawiera listę typów, które deklarują metodę `m` (w typach z tej listy nazwa metody `m` zostanie zmieniona)
Założenie: Żaden nadtyp typu `T` nie deklaruje metody `m`

```
result:= empty set // zbior typow deklarujacych metode m
visited:= empty set //zbior typow juz odwiedzonych
q:= empty queue //kolejka typow do odwiedzenia
q.insert(T)

while (!q.isEmpty()){
    t:= q.remove();
    //assert(t jest interfejsem lub deklaruje m jako metode wirtualna)
```

[2]Odmiennie niż w przypadku metod wirtualnych, które, jeżeli mają tę samą sygnaturę, są niejako odmianami tej samej metody.

```
    //assert(!visited.contains(t))
    visited.add(t);
    result.add(t);
    forall: i in: t.subTypes do:
       if (i deklaruje m) result.add(i);
    forall: i in: t.subTypes do:
       q.insert(x)
           gdzie x jest dowolnym typem spelniajacym warunki:
              a. x jest nadtypem typu i
              b. x jest interfejsem deklarujacym m lub
                 x deklaruje m jako metode wirtualna
              c. zaden nadtyp typu x nie jest
                    interfejsem deklarujacym m oraz
                 zaden nadtyp typu x nie jest
                    klasa deklarujaca m jako metode wirtualna
              d. ! visited.contains(x)
              e. ! q.contains(x)
}
```

Inne zastosowania podanego algorytmu

Podany w poprzednim punkcie algorytm może zostać wykorzystany także w innych refaktoryzacjach niż Zmiana Nazwy Metody. W istocie prawie każda refaktoryzacja modyfikująca metody musi się posługiwać tym algorytmem.

Na przykład, refaktoryzacja polegająca na zmianie kolejności parametrów metody (ang. *Rearrange Parameters* lub *Exchange Parameters*) musi dokonać modyfikacji deklaracji i odwołań do wszystkich metod wzdłuż fali. Należy wobec tego skorzystać z wyszukującego je algorytmu. Inne przykłady to, m.in. :

- Dodanie Parametru Metody (ang. *Add Parameter*)
- Usunięcie Parametru Metody (ang. *Remove Parameter* lub *Absorb Parameter*)
- Usunięcie Metody (ang. *Remove Method* lub *Safe Remove Method*)

3.2. Przestrzenie nazw typów

W języku Java istnieją przestrzenie nazw typów. Może więc istnieć wiele typów o tej samej nazwie – jeżeli znajdują się w innych przestrzeniach nazw.

Każdy typ użyty w progamie ma nazwę prostą (np. `Object`) oraz unikatową nazwę kanoniczną (np. `java.lang.Object`). Nazwa kanoniczna składa się z (występujących w podanej niżej kolejności i oddzielonych znakami '.'):

1. nazwy pakietu, w którym dany typ się znajduje,
2. nazw (kolejno „w dół" – od typu niezagnieżdżonego) wszystkich typów, w których dany typ jest zagnieżdżony,
3. nazwy danego typu.

Przykładowo, kanoniczna nazwa typu o nazwie prostej `A` zagnieżdżonego w typie o nazwie prostej `B` i zdefiniowanego w pakiecie o nazwie `p1.p2` to `p1.p2.B.A`. W przypadku typów niezagnieżdżonych część 2 nie występuje.

W pełni kwalifikowane nazwy typów (ang. *fully qualified type names*) są odmienne od nazw kanonicznych (ang. *canonical names*) i, w ogólnym przypadku, nie są unikatowe w systemie. W niejakim skrócie można powiedzieć, że nazwa kanoniczna jest jedną z w pełni kwalifikowanych nazw typu. Różnice są szczegółowo opisane w specyfikacji języka [JLS2 6.7]. Tam gdzie to rozróżnienie nie wpływa na tok rozumowania, będziemy używali pojęcia w pełni kwalifikowanej nazwy typu.

Ze względu na zwięzłość i wygodę rzadko używa się w pełni kwalifikowanych nazw typów. Konieczność ich użycia wystepuje jedynie wówczas, gdy w jednej jednostce kompilacji znajdują się odniesienia do co najmniej dwóch typów o tej samej nazwie prostej.

Kilka konstrukcji języka Java powoduje, że podczas refaktoryzacji może zdarzyć się tak, że w refaktoryzowanym fragmencie kodu może być widocznych wiele typów o tej samej nazwie prostej. Możliwe są wówczas dwa wyjścia z takiej sytuacji:

a. nazwa jednego z tych typów przesłania (lub ukrywa) pozostałe,

b. dwie lub więcej nazw pozostaje w konflikcie i powstaje błąd kompilacji.

Przedstawimy teraz te konstrukcje języka, po czym przejdziemy do omówienia możliwych sposobów postępowania w przypadku wystąpienia niepożądanego przesłaniania lub konfliktu nazw typów.

Deklaracje importu

W celu umożliwienia odniesień do jakiegoś typu przez użycie jego nazwy prostej należy go „zaimportowac" do jednostki kompilacji. Istnieją dwa sposoby importowania typów.

- **Import pojedynczego typu** (ang. *single–type–import*) Używa się do niego w pełni kwalifikowanej nazwy importowanego typu.
- **Import na żądanie** (ang. *import–on–demand*) w tym przypadku używa się nazwy pakietu deklarującego dany typ z następującymi po niej znakami ".*"(import dotyczy wówczas wszystkich publicznych, niezagnieżdżonych typów zdefiniowanych w tym pakiecie).

  Typy zagnieżdżone również można importować na żądanie – należy wówczas użyć w pełni kwalifikowanej nazwy typu, w którym zdefiniowany jest dany typ (np. w celu takiego zaimportowania typu o nazwie `p1.p2.B.A` należy użyc wyrażenia `import p1.p2.B.*;`).

Wielu programistów używa wyłącznie importu pojedynczego typu za względu na jego jednoznaczność. Inni preferują import na żądanie ze względu na zwięzłość. Istnieją jednak poważniejsze niż estetyczne róznice między tymi dwoma sposobami importowania typów. Zgodnie ze specyfikacją języka ([JSL2 7.5.1] i [JLS2 7.5.2]), import na żądanie nigdy nie przesłania żadnych deklaracji, natomiast import pojedynczego typu przesłania deklaracje typów importowanych na żądanie oraz typów importowanych domyślnie (tzn. typów zdefiniowanych w pakiecie, w którym znajduje się importująca jednostka kompilacji oraz typów zdefiniowanych w wyróżnionym pakiecie o nazwie `java.lang`).

Kilka reguł języka Java dotyczących importowania typów leży w obszarze zainteresowania projektanta narzędzia do refaktoryzacji:

a. Żadna jednostka kompilacji nie może importować (import pojedynczego typu) dwóch lub wiecej różnych typów o tej samej nazwie prostej.

b. Żadna jednostka kompilacji nie może deklarować niezagnieżdżonego typu i jednocześnie importować (import pojedynczego typu) innego typu o tej samej nazwie prostej.

c. Jeśli jednostka kompilacji importuje dwa lub wiecej typów (import na żądanie) o tej samej nazwie prostej i występują wewnątrz niej odniesienia do jednego z tych typów używające jego nazwy prostej, to jest zgłaszany błąd kompilacji (odniesienia nie są jednoznaczne).

Spójrzmy na przykłady:

```
"A.java"
package p;
public class A{}

"B.java"
package p1;
public class B{}

"Test.java"
package test;
import p1.B;   /*1*/
import p.A;
class Test{
    B b;  /*2*/
}
```

Zmiana nazwy typu `A` na `B` (wraz z aktualizacją odniesień) daje w rezultacie błąd kompilacji w ostatniej jednostce kompilacji – która importuje wówczas (import pojedynczego typu) dwa typy o tej samej nazwie prostej (tj. `B`).

Podobnie, rozważmy następujący przykład:

```
"A.java"
package p;
public class A{}

"Test.java"
package test;
import java.util.*; //zawiera typ List
import p.*;
class Test{
    A a;
    List l; /*1*/
}
```

Jeśli zmienimy nazwę typu `A` na `List`, to odniesienie w wierszu oznaczonym przez `/*1*/` stanie się niejednoznaczne, co spowoduje błąd kompilacji.

Kolejny przykład ilustruje, jak działanie programu może ulec zmianie (bez wystąpienia błędów kompilacji) przez przesłonięcie widoczności typu zdefiniowanego w tym samym pakiecie, ale innej jednostce kompilacji.

```
"B.java"
package p1;
public class B{
    public static int x= 42;
}

"A.java"
package p;
public class A{
    public static int x= 0;
}

"Test.java"
package p;
import p1.B;
class Test{
    static int i;
    static {
        i= A.x;
    }
}
```

Jeśli jedynie zmienimy nazwę typu `p1.B` na `A` (wraz z aktualizacją wszystkich odniesień do tego typu), to pole i w klasie `Test` zostanie zainicjalizowane wartością 42, a nie 0, jak poprzednio. Pomimo zmiany w działaniu programu, pozostaje on poprawny, tzn. nie powstają błędy kompilacji. Zauważmy, że działanie programu się zmienia, mimo, że w klasie `Test` nie ma odniesień do typu `p1.B`, którego nazwę zmieniamy. Importowanie typu (i związane z tym przesłanianie) wystarcza, by w istotny sposób zmienić funkcjonowanie programu.

Ostatni przykład pokazuje, jak samo stworzenie nowej klasy[3] może zmodyfikować działanie programu.

```
package p;
import java.util.*;  // zawiera klase ArrayList
class A{
    public Object m(){
        return new ArrayList();
    }
}
```

Stworzenie w pakiecie `p` klasy o nazwie `ArrayList` spowoduje, że metoda `A::m()` przekaże obiekt klasy innej niż dotychczas, zmieniając w ten sposób działanie programu (bez powodowania błędów kompilacji).

[3]Refaktoryzacja polegająca na stworzeniu nowej klasy jest uznawana za jedną z najprostszych i nie wymaga sprawdzenia prawie żadnych warunków wstępnych w systemach, w których nie występują przestrzenie nazw [5, 17, 21]. Podany przykład pokazuje, że nie jest to tak proste w systemach używających przestrzeni nazw.

Typy zagnieżdżone

Typy można zagnieżdżać (ang. *nest*), tzn. definiować jedne wewnątrz innych. Szczegółowe omówienie reguł definiowania i semantyki typów zagnieżdżonych znajduje się w specyfikacji języka [8]. Każdy typ może być zagnieżdżony co najwyżej w jednym innym typie; wszystkie typy w programie tworzą wobec tego, niezależnie od grafu dziedziczenia, zbiór hierarchii zawierania (typy niezagnieżdżone są na szczytach tych hierarchii).

Nazwy typów zagnieżdżonych przesłaniają widoczność nazw innych typów oraz zaciemniają (patrz punkt 3.3) widoczność nazw pakietów. Spójrzmy na przykład:

```
import java.util.*;   //zawiera typ Stack
class A{
    class B{
        Object m(){
            return new Stack{};
        }
    }
}
```

Zmiana nazwy klasy dowolnej z klas `A` lub `B` na `Stack` spowoduje zmianę działania programu – bez powstania błędów kompilacji.

Mniej oczywistą, a równie ważną, konsekwencją używania typów zagnieżdżonych jest niejednoznaczność nazw typów, jaka się wówczas pojawia. Omówimy teraz to zjawisko.

Do każdego niezagnieżdżonego typu możemy odnosić się na dwa sposoby: za pomocą nazwy prostej lub nazwy kanonicznej (która jest wtedy tożsama z nazwą w pełni kwalifikowaną). Do typów zagnieżdżonych możemy odnosić się na wiele sposobów – w zależności od miejsca odniesienia. W istocie, w pewnych warunkach, liczba sposobów, na jakie możemy się odnosić do jednego, zagnieżdżonego typu, jest nieograniczona.

W podanym przykładzie wszystkie zmienne w wierszach oznaczonych przez `/**/` są tego samego typu (o kanonicznej nazwie `p.O.I.J`). Przykład jest niestety, z konieczności, dość zawiły:

```
package p;
public class O{
    public class I{
        public class J{}
        J j;          /**/
    }
    I.J ij;             /**/
}
class O1 extends O{
    class O2 extends O.I{
        O2(O o){ o.super(); }
    }
    O2.J o2j;       /**/
}
class Test{
    O.I.J       oij;    /**/
    p.O.I.J     poij;   /**/
    O1.I.J      o1ij;   /**/
```

```
    p.O1.I.J    po1ij;  /**/
    O1.O2.J     o1o2j;  /**/
    p.O1.O2.J   po1o2j; /**/
}
```

Zauważmy, że `p.O.I.J`, `p.O1.I.J` i `p.O1.O2.J` to w pełni kwalifikowane nazwy dla wskazanego typu – co ilustruje wspomnianą wcześniej nieunikatowość takich nazw.

Dodanie nowego pakietu powoduje dalsze zwiększenie liczby możliwych sposobów odniesień do tego typu. Przykład ilustrujący tę sytuację zostanie tu pominięty.

Łatwo zdać sobie sprawę z tego, jak ta cecha języka wpływa na znaczne zwiększenie możliwych przesłonięć nazw typów. Każda z podanych nazw może być przesłaniana (lub zaciemniana) niezależnie od innych. Oznacza to, że podczas refaktoryzacji musimy, po pierwsze, odszukać je wszystkie jako odniesienia to danego typu, a po drugie analizować każdą z nich oddzielnie. Zmiana nazwy lub położenia typu, dodanie nowego typu do systemu oraz wszelkie zmiany w hierarchiach dziedziczenia, czy zawierania) to kilka przykładów refaktoryzacji, które mogą być niepoprawne, jeśli nie zostaną poprzedzone analizą przypadków przedstawionych w tym punkcie. Jak pokazano w punkcie 3.3 również przekształcenia operujące na zmiennych i pakietach (dodające je do systemu, zmieniające ich nazwy lub położenie itp.) muszą analizować te przypadki.

Typy lokalnie zdefiniowane

W języku Java można definiować typy wewnątrz metod. Jest do rzadko używana cecha języka, która jednak może przyczynić się do powstania trudnych do wykrycia błędów, jeżeli narzędzie do refaktoryzacji nie weźmie jej pod uwagę.

Typy można deklarować w dowolnym miejscu metody – tak, jak zmienne lokalne. Ich nazwy przesłaniają widoczność nazw innych typów począwszy od miejsca deklaracji.

Prosty przykład ilustruje sytuację, w której nazwa typu zdefiniowanego lokalnie przesłania widoczność nowej nazwy innego typu, uniemożliwiając tym samym refaktoryzację:

```
package p;
class X{}
class Test{
    String f;
    Object m(){
        class A{};
        return new X(); /*1*/
    }
}
```

Zmiana nazwy typu o kanonicznej nazwie `p.X` na `A` spowoduje, że wywołanie metody `Test::m()` przekaże obiekt innego typu.

Postępowanie w wypadku wystąpienia konfliktu nazw typów

W przypadku wystąpienia konfliktu nazw typów narzędzie do refaktoryzacji może posłużyć się jedną z dwóch strategii:

a. Starać się naprawić kod – tzn. rozwiązać konflikt przez zamianę odniesień używających nazw prostych na odniesienia w postaci nazw w pełni kwalifikowanych.

b. Uznać wystąpienie takiej sytuacji za niepożądane i zabronić refaktoryzacji.

Opisana w punkcie a próba naprawy kodu może (choć nie musi – patrz punkt 3.3) okazać się skuteczna (konieczne mogą okazać się dodatkowe zmiany). Jednakże wprowadzenie do programu odniesień używających nazw w pełni kwalifikowanych wydaje się sprzeczne z oczekiwaniami użytkowników co do narzędzia wspomagającego refaktoryzację. Jak wspomnieliśmy, nazwy w pełni kwalifikowane są stosunkowo rzadko wykorzystywane, a ich nadużywanie jest uznawane za element złego stylu. Uważamy także, że wystąpienie w programie konfliktu nazw typów wskazuje na nieprawidłowości w używanym schemacie nadawania nazw elementom programu.

Z tego powodu uznajemy, że zabronienie refaktoryzacji jest w takich przypadkach lepszym rozwiązaniem i zdecydowaliśmy się je zaimplementować w naszym programie.

3.3. Przesłanianie, ukrywanie i zaciemnianie nazw

Trzy zjawiska dotyczące nazw leżą w kręgu zainteresowania projektanta narzędzia do refaktoryzacji programów w jezyku Java – przesłanianie, ukrywanie oraz zaciemnianie. Przedstawimy je teraz po kolei, a następnie omówimy możliwe sposoby postępowania w przypadku ich wystąpienia.

Przesłanianie

Przesłanianie (ang. *shadowing*) opisane jest w punkcie 6.3.1 specyfikacji języka Java. Deklaracje mogą być przesłaniane w częściach swych zasięgów przez inne deklaracje o tej samej nazwie. Nazwa prosta nie może być wówczas użyta w celu odniesienia do zadeklarowanego elementu. Szczegółowa lista warunków, w jakich jedne deklaracje przesłaniają inne znajduje się w specyfikacji języka.

Przesłanianie jest szeroko znaną i stosowaną cechą jezyków programowania. Wobec tego, w niniejszym punkcie podany jedynie krótki przykład pokazujący w jaki sposób może ono wpłynąć na refaktoryzację – konkretnie na Zmianę Nazwy Pola.

```
class S{
    protected int g;
}
class A extends S{
    public int m(int p){
        return g + p;
    }
}
```

Zmiana nazwy pola `g` na `p` powoduje, że jego deklaracja zostaje przesłonięta przez nazwę parametru w treści metody `A::m()`.

Ukrywanie

Pojęcie ukrywania (ang. *hiding*) odnosi się do elementów, które byłyby odziedziczone z nadtypów – jednak nie są, z powodu istnienia deklaracji w podtypie. Jest ono opisane w specyfikacji języka Java (punkty 8.3, 8.4.6.2, 8.5, 9.3, 9.5).

Ukrywanie, mimo, że jest odmienne od przesłaniania, w procesie refaktoryzacji może być traktowane podobnie.

Spójrzmy na następujący przykład:

```
class A{
    protected int f;
}
class B extends A{
    void m(){
        f= 42; /*1*/
    }
}
```

Rozważmy modyfikację polegającą na dodaniu do klasy `B` nowego pola typu `int`, o nazwie `f`. W celu zachowania dotychczasowego działania programu, należy wiersz oznaczony przez `/*1*/` zmienić na:

```
((A)this).f= 42;  /*1*/
```

Zaciemnianie

W punkcie 6.3.2 specyfikacji języka stwierdza się, że nazwy proste mogą występować w kontekstach, w których mogą zostać interpretowane jako nazwy zmiennej, typu lub pakietu. W takich sytuacjach zmienna ma pierwszeństwo przed typem, a typ przed pakietem (punkt 6.5 specyfikacji języka). To zjawisko nazywa się zaciemnianiem nazw (ang. *obscuring*). Może się więc zdarzyć, że nie możemy odwołać się do widocznego typu bądź pakietu przez jego nazwę prostą. Zaciemnianie zdarza się najczęściej wtedy, gdy nie przestrzega się konwencji nazw (opisanych w punkcie 6.8 specyfikacji języka).

```
class X{
    static int length(){ return 42;};
}
class Test extends TestSuperclass{
    String s= "hello";
    int m(){
        return X.length(); /*1*/
    }
}
```

Zmiana nazwy typu `X` na `s` nie powoduje błędów kompilacji. Działanie programu ulega jednak zmianie (wywołanie metody `Test::m()` przekazuje wartość 5, a nie 42, jak poprzednio).

Przykłady ilustrujące inne rodzaje zaciemniania (t.j. nazwa zmiennej zaciemniająca nazwę pakietu, nazwa typu zaciemniająca nazwę pakietu) są łatwe do skonstruowania i nie zostaną tu podane.

Postępowanie w przypadku wystąpienia przesłaniania, ukrywania lub zaciemniania

Podobnie jak przy konflikcie nazw typów, również w przypadkach wystąpienia przesłaniania, ukrywania lub zaciemniania możliwe jest przyjęcie jednej z dwu strategii postępowania:

a. Możemy (jako twórcy narzędzia do refaktoryzacji) starać się odpowiednio zmodyfikować kod, by uniknąć niepożądanego przesłaniania, ukrywania lub zaciemniania. Użycie konkretnej metody postępowania jest uzależnione od sytuacji – w niektórych przypadkach niezbędne jest wprowadzenie rzutowania (ang. *downcast*), w innych użycie w pełni kwalifikowanych nazw typów itd.

b. Możemy wykrywać sytuacje prowadzące do powstania przesłaniania, ukrywania lub zaciemniania i zabraniać refaktoryzacji w takich przypadkach.

W implementacji naszego narzędzia zdecydowaliśmy się na realizację podejścia b. Argumenty przemawiające za taką decyzją są podobne do przedstawionych w punkcie 3.2.

3.4. Przeciążanie nazw metod

„Jeżeli dwie metody w klasie (zadeklarowane w tej samej klasie lub obie odziedziczone z nadklasy lub jedna zadeklarowana, a druga odziedziczona) mają tę samą nazwę lecz różne sygnatury, to mówimy, że nazwa metody jest *przeciążona* (ang. *overloaded*)." [JLS2, p. 8.4.7].

Refaktoryzacje mogą prowadzić zarówno do eliminowania, jak i do powstawania przeciążania. W tej pracy omówimy tylko przypadek *powstawania* przeciążania.

Rozważmy następujący przykład:

```
class A{
    void a(String s){}
    void b(Object s){}
}
```

Zmiana nazwy metody `A::b()` na a powoduje, że wywołanie `b("hello")` musi zostać zastąpione przez `a((Object)"hello")`.

Podobnie jak w przypadku przestrzeni nazw typów istnieją dwie strategie postępowania przy zmianie nazwy metody:

a. Możemy użyć rzutowania parametrów dla każdego wywołania metody.

b. Możemy wykrywać przypadki powstawania przeciążania metod i zabraniać refaktoryzacji.

Podejście opierające się na rzutowaniu jest, naszym zdaniem, sprzeczne z intuicją i oczekiwaniami użytkowników. Narzędzie dokonywałoby wówczas w programie zmian, jakich użytkownicy się nie spodziewają. Co więcej, nadmierne użycie rzutowania jest uznawane za niepożądaną praktykę programistyczną. Z tych powodów zdecydowaliśmy się na wykrywanie przypadków przeciążania nazw metod i informowanie o tym użytkownika.

3.5. Metody natywne

Metody natywne (ang. *native*) są napisane w innych niż Java językach programowania i dołączane (ang. *link*) podczas działania programu. Zwykle jest to język C bądź C++ (i takie założenie przyjmiemy w tej pracy). Używając metod natywnych nie podaje się ich treści – jedynie deklaracje, podobnie jak metod abstrakcyjnych. „Natywny" oznacza więc w tym kontekście dwie rzeczy – po pierwsze metody w języku Java zadeklarowane jako natywne, po drugie ich deklaracje i kod po stronie C/C++. Określenie „kod natywny" oznacza kod w C/C++ implementujący metody natywne.

Refaktoryzacja programów w Javie korzystających z kodu natywnego musi być przeprowadzana uważnie, gdyż większość możliwych błędów nie może zostać wykrytych przez kompilator. Kod natywny jest ładowany i dołączany podczas działania programu i wszelkie błędy ujawniają się dopiero wtedy, tzn. przy pierwszym błędnym odwołaniu do metody natywnej.

Sposób, w jaki używa się metod natywnych i jak są one dołączane do działającego programu jest określony przez specyfikację JNI (Java Native Interface) [9]. Podaje ona dokładny opis deklaracji metod natywnych w kodzie C/C++. Wiązanie natywnych metod Javy z odpowiadającym im kodem natywnym odbywa się za pomocą dopasowania nazw. Nazwy metod natywnych po stronie C/C++ są tworzone od nazw odpowiadających im metod natywnych po stronie Javy.

Nazwa metody natywnej w C/C++ składa się z:

1. przedrostka "Java_",
2. kanonicznej nazwy klasy deklarującej odpowiadającą metodę natywną Javy,
3. znaku '_',
4. nazwy odpowiadającej metody natywnej Javy,
5. oraz (jeśli nazwa metody jest przeciążona) ciągu "__" wraz z następującą po nim sygnaturą metody (wraz z w pełni kwalifikowanymi nazwami typów argumentów). Jeżeli nazwa nie jest przeciążona, to ostatni fragment jest opcjonalny.

Przykładowo, dla podanej niżej definicji klasy

```
class A{
    native void m(B b);
    native void m(C b);
}
```

gdzie A znajduje się w pakiecie `p1.p2.p3`, natomiast B oraz C w pakiecie `p4.p5.p6`, odpowiedni fragment pliku nagłówkowego (wygenerowanego przez program `javah`) deklarującego metodę `m` w kodzie C/C++ wygląda następująco:

```
/*
 * Class:     p1_p2_p3_A
 * Method:    m
 * Signature: (Lp4/p5/p6/B;)V
 */
JNIEXPORT void JNICALL Java_p1_p2_p3_A_m__Lp4_p5_p6_B_2
  (JNIEnv *, jobject, jobject);

/*
 * Class:     p1_p2_p3_A
 * Method:    m
 * Signature: (Lp4/p5/p6/C;)V
 */
JNIEXPORT void JNICALL Java_p1_p2_p3_A_m__Lp4_p5_p6_C_2
  (JNIEnv *, jobject, jobject);
```

Przyjmujemy założenie, że nie mamy dostępu do kodu natywnego, w związku z czym nie możemy go analizować ani modyfikować. Znając reguły wiązania nazw natywnych metod Javy i odpowiadających im metod C/C++ możemy jednak ostrzegać użytkownika przed przeprowadzeniem modyfikacji programu, w następstwie której powstałyby błędy podczas wykonywania programu korzystającego z metod natywnych.

Z podanych reguł tworzenia nazw wynika, że bez zmiany działania programu nie można zmienić (w programie napisanym w Javie):

a. nazwy ani położenia metody natywnej;

b. nazwy ani położenia klasy, która deklaruje metodę natywną bądź której jakakolwiek klasa zagnieżdżona (na dowolnym poziomie) deklaruje taką metodę (odnosi sie to również do klas zdefiniowanych lokalnie w rozpatrywanej klasie oraz wszystkich klasach zagnieżdżonych i lokalnie w niej zdefiniowanych);

c. nazwy pakietu, w którego skład wchodzi klasa (na dowolnym poziomie zagnieżdżenia – także klasy lokalnie zdefiniowane), która deklaruje metodę natywną;

d. nazwy ani położenia typu, który występuje jako typ argumentu jakiejkolwiek metody natywnej (dowolnie zlokalizowanej). Ponieważ nie mamy dostępu do kodu natywnego, więc musimy założyć, że nawet przy braku przeciążania w nazwie metody natywnej użyto w pełni kwalifikowanych nazw typów parametrów;

e. nazwy pakietu, w którego skład wchodzi klasa (na dowolnym poziomie zagnieżdżenia – także klasy lokalnie zdefiniowane), która występuje jako typ argumentu jakiejkolwiek metody natywnej (dowolnie zlokalizowanej).

Sprawdzenie ostatnich dwóch warunków wymaga analizy całego programu.

Jeśli nie jest wywoływana żadna z metod natywnych, które spełniają co najmniej jeden z podanych warunków, to działanie programu pozostanie niezmienione. Kompilator Javy nie ma możliwości orzec, czy kod natywny jest prawidłowo napisany. Jeśli tak nie jest, to przy pierwszym wywołaniu metody wystąpi błąd (`UnsatisfiedLinkError`). Problem sprawdzenia, czy dana metoda jest wołana podczas działania programu jest jednak, w ogólnym przypadku, nierozstrzygalny.

Z mechanizmu JNI korzysta się raczej rzadko, więc powyższe ograniczenia nie powinny być zbyt uciążliwe w praktyce. Są jednak konieczne do tego, by można było uzyskać niezmienność działania programu.

Niestety kod natywny stanowi większe zagrożenie dla poprawności refaktoryzacji. Kod natywny może odwoływać się do dowolnego elementu programu w Javie (typu, metody, pola obiektu) co powoduje, że, podobnie jak przy użyciu mechanizmu odbicia (ang. *reflection*), narzędzie nie jest w stanie zagwarantować poprawności przekształceń w obecności metod natywnych (bez względu na to, czy dana refaktoryzacja ich dotyczy, czy nie). Jest to jedno z tych ograniczeń, którym nie sposób łatwo zapobiec. W tym celu należałoby mieć dostęp do kodu natywnego i móc go analizować. Kod natywny może być jednak napisany w dowolnym języku programowania, co całkowicie uniemożliwia jego analizę w ogólnym przypadku.

3.6. Przypadki specjalne

W języku Java obowiązuje kilka reguł specjalnych – nie mieszczących się w żadnej z podanych dotąd kategorii. Najważniejsze z nich to te, które dotyczą nierównouprawnienia nazw.

Metoda `toString()`

Metoda `toString()`, zadeklarowana w klasie `java.lang.Object` jest wywoływana domyślnie przez użycie operatora `+` wtedy, gdy co najmniej jeden z argumentów jest obiektem klasy `java.lang.String`. Ta reguła sprawia, że następujące wyrażenie jest dozwolone:

```
String f= "a" + new Exception();
```

Pewne (hipotetyczne) refaktoryzacje muszą brać ten przypadek pod uwagę. W szczególności, jest to zmiana nazwy tej metody lub jej położenia.

Pakiet `java.lang`

Wszystkie publiczne typy z pakietu java.lang są importowane niejawnie (import na żądanie) przez wszystkie jednostki kompilacji. Nie trzeba umieszczać jawnej deklaracji importu tego pakietu.

Program do refaktoryzacji musi traktować wszystkie jednostki kompilacji jako zawierające deklaracje `import java.lang.*`; Warunki wstępne refaktoryzacji, jakie się z tym wiążą, wynikają bezpośrednio z uwag podanych w punkcie 3.2.

Ponadto nazwy wielu typów zadeklarowanych w tym pakiecie są na stałe zakodowane w maszynie wirtualnej Javy. Możemy więc założyć, że nie wolno:

- zmienić nazwy ani położenia żadnego typu zadeklarowanego w tym pakiecie,
- zmienić nazwy tego pakietu.

Metoda `main`

Maszyna wirtualna Javy, podczas uruchomienia, próbuje zlokalizować metodę o sygnaturze `main(java.lang.String[])`, zadeklarowaną jako `public static void` w klasie, której w pełni kwalifikowaną nazwę podano jako argument w wierszu poleceń. Z tego powodu nie jest możliwa, bez zmiany zachowania programu, zmiana nazwy, położenia, typu wyniku, typu i liczby argumentów tej metody, jak również jej modyfikatorów. Nie można także zmienić nazwy ani położenia typu, w którym jest zadeklarowana ani żadnego z zawierających go typów. Również zmiana nazwy pakietu, w którym znajduje się typ deklarujący taką metodę spowoduje błąd. Niemożliwa jest także (bez spowodowania niemożności uruchomienia programu) zmiana nazwy lub położenia typu `java.lang.String`.

Narzędzie wspierające refaktoryzację powinno ostrzegać użytkownika o wystąpieniu tego typu sytuacji, by pozwolić, kiedy to możliwe (np. przy zmianie nazwy typu deklarującego metodę `main`) na aktualizację zewnętrznych programów uruchamiających refaktoryzowany program w Javie (skryptów, programów wsadowych i innych).

Niejawne dziedziczenie z klasy `java.lang.Object`

W specyfikacji języka napisano ([JLS2 9.2]), że : „jeżeli interfejs nie ma żadnych bezpośrednich nadinterfejsów, wówczas niejawnie deklaruje publiczną, abstrakcyjną metodę m o sygnaturze s, typie wyniku r oraz klauzuli `throws` t odpowiadającą każdej z publicznych metod instancyjnych m o sygnaturze s, typie wyniku r oraz klauzuli `throws` t zadeklarowanych w klasie `java.lang.Object`, chyba, że metoda o tej samej sygnaturze, typie wyniku i odpowiadającej klauzuli `throws` jest jawnie zadeklarowana w tym interfejsie. Wynika z tego, że zgłaszany jest błąd kompilacji, jeżeli interfejs deklaruje metodę o tej samej sygnaturze i odmiennym typie wyniku bądź nieodpowiadającej klauzuli `throws`."

Z tej reguły wynika znaczna liczba warunków wstępnych, które muszą zostać sprawdzone w celu upewnienia się, że podczas refaktoryzacji programu nigdy nie powstanie interfejs łamiący tę zasadę. Lista tych warunków jest łatwa do skonstruowania i zostanie tu pominięta.

Mechanizm odbicia

Mechanizm odbicia pozwala na dynamiczny dostęp do elementów programu używając ich nazw, które mogą być określone dopiero podczas działania systemu. Przykładowo, poniższy fragment kodu może być użyty do wywołania metody `m` obiektu `o`:

```
o.getClass().getMethod("m", null).invoke(o)
```

Zmiana nazwy metody `m` spowoduje, że efekt wykonania podanego fragmentu kodu nie będzie taki, jak poprzednio – zostanie zgłoszony wyjątek.

Ponieważ nie mamy możliwości przeprowadzenia statycznej analizy użycia mechanizmu odbicia (w szczególności tego, które elementy programu będą używane), wobec tego musimy zgodzić się na powstanie ewentualnych błędów podczas refaktoryzacji używającego go kodu.

3.7. Podsumowanie

Java jest językiem programowania o skomplikowanej składni i semantyce. W tym rozdziale pokazaliśmy, w jaki sposób pewne cechy języka Java (charakterystczne, lecz nie specyficzne dla niego) sprawiają, że refaktoryzacja programów w tym języku jest trudna i musi być przeprowadzana uważnie, by nie prowadzić do błędów.

Opisaliśmy w jaki sposób użycie wielokrotnego dziedziczenia interfejsów wpływa na proces refaktoryzacji. Na przykładzie refaktoryzacji Zmiana Nazwy Metody pokazaliśmy jak występowanie tej cechy języka wpływa na zbiór metod, które muszą być refaktoryzowane razem (w celu zachowania działania programu – w szczególności uniknięcia błędów kompilacji). Omówiliśmy wpływ metod niewirtualnych na to zjawisko i przedstawiliśmy algorytm znajdowania wszystkich metod, które muszą być refaktoryzowane wspólnie. Następnie krótko opisaliśmy zastosowania podanego algorytmu do przeprowadzania również innych refaktoryzacji.

W kolejnym punkcie pokazaliśmy możliwe problemy występujące przy refaktoryzacji programów w języku Java, wynikające z istnienia przestrzeni nazw typów. Przedstawiliśmy niektóre z przyczyn i następstw konfliktu nazw typów. Omówiliśmy możliwe strategie postępowania w przypadku wystąpienia konfliktu nazw typów, a następnie podaliśmy i uzasadniliśmy decyzję podjętą przez nas przy implementacji narzędzia do refaktoryzacji.

W punkcie 3.3 omówiliśmy trzy zbliżone do siebie zjawiska związane z nazwami elementów programu – przesłanianie, ukrywanie i zaciemnianie. Pokazaliśmy wpływ, jaki wywierają one na refaktoryzację. Następnie, podobnie jak poprzednio, omówiliśmy możliwe strategie postępowania w przypadku wystąpienia problemów związanych z występowaniem omawianych zjawisk, przedstawiliśmy i uzasadniliśmy dokonany przez siebie wybór.

Wpływ mechanizmu przeciążania nazw metod na refaktoryzację omówiony został w punkcie 3.4 – wraz z dyskują możliwych sposobów rozwiązywania związanych z tym problemów oraz przedstawieniem i uzasadnieniem podjętej decyzji.

Ostatnie dwa punkty rozdziału zawierają omówienie kilku przypadków specjalnych – zwłaszcza istnienia wyróżnionych nazw elementów programu oraz problemów związanych z refaktoryzacją programów korzystających z metod natywnych. W punkcie 3.5, omawiając metody natywne, przedstawiliśmy listę warunków wstępnych związanych z refaktoryzacją programów korzystających z tego mechanizmu.

Rozdział 4

Wsparcie refaktoryzacji w IBM WebSphere Studio Workbench

W tym rozdziale prezentujemy opis projektu i implementacji narzędzia do wspomagania procesu refaktoryzacji programów w języku Java. Obecnie jest ono integralną częścią środowiska IBM WebSphere Studio Workbench. Jest dostępne (wraz z kodem źródłowym) pod adresem [24].

Cała niniejsza praca jak i wspomniane narzędzie, którego opis stanowi jej istotną część, powstały podczas stażu autora w biurze firmy Object Technology International w Zurychu[1] pod kierownictwem dra Ericha Gammy. Projekt i implementacja narzędzia zostały wykonane przez Adama Kieżuna (ogólny projekt, wszystkie refaktoryzacje poza Wydzieleniem Metody, operacje na plikach, operacje na pakietach, infrastruktura zestawu testów, interfejs graficzny) oraz dra Dirka Bäumera (refaktoryzacja Wydzielenie Metody, operacje tekstowe, interfejs graficzny).

Integracja wsparcia refaktoryzacji ze środowiskiem programistycznym IBM WebSphere Studio Workbench została przez nas opisana w artykule [1].

Do tej pory (wersja R0.9 produktu) zaimplementowano część planowanych refaktoryzacji wraz z dużym zestawem testów towarzyszącym każdej z nich. Znaczną część pracy pochłonęło stworzenie elastycznej architektury, dzięki której, jak się spodziewamy, z czasem stosunkowo łatwo będzie dodawać kolejne refaktoryzacje. Nasze narzędzie było z powodzeniem stosowane do refaktoryzacji dużego projektu – IBM WebSphere Studio Workbench (ponad 500000 wierszy kodu źródłowego) – jest też na codzień używane przez samych autorów.

Program został w całości napisany w języku Java. Dało nam to możliwość użycia go do refaktoryzacji jego własnego kodu źródłowego – co z powodzeniem robimy podczas pracy nad kolejnymi udoskonaleniami.

IBM WebSphere Studio Workbench jest pierwszym dużym środowiskiem programistycznym zawierającym wsparcie dla refaktoryzacji. Począwszy od jednej z kolejnych wersji będzie dostępne bezpłatnie wraz z kodem źródłowym. Mamy nadzieję, że dzięki temu damy wielu programistom możliwość zapoznania się z nowym sposobem pracy, jaki umożliwia refaktoryzacja. Ponadto przez stworzenie elastycznej infrastruktury chcemy dać innym twórcom narzędzi możliwość łatwej implementacji nowych refaktoryzacji. Sądzimy, że otwartość środowiska umożliwi stworzenie lepszej jakości narzędzi do refaktoryzacji (i nie tylko).

[1] Kod źródłowy programu stanowi własność firmy Object Technology International Inc. z siedzibą w Ottawie, 2670 Queensview Drive, Ottawa, Ontario, K2B 8K1, Kanada.

4.1. Opis architektury środowiska programistycznego

Najważniejszymi częściami platformy, wykorzystywanymi w naszym programie (poza interfejsem graficznym) są: system zarządzania zasobami, kompilator Javy oraz wyszukiwarka (ang. *search engine*).

System zarządzania zasobami kontroluje dostęp do zasobów (w naszym przypadku plików i katalogów) i sposób ich modyfikacji. Innym bardzo istotnym jego zadaniem jest zarządzanie znacznikami (ang. *markers*). Znacznik to obiekt związany z zasobem. Każdy znacznik związany z plikiem ma atrybut określający jego pozycję w danym pliku. Przykładowe rodzaje znaczników to: punkty kontrolne (ang. *breakpoints*), zakładki (ang. *bookmarks*), wyniki wyszukiwań (ang. *search results*) oraz znaczniki oznaczające błędy kompilacji.

Najważniejszą cześcią kompilatora Javy, z której korzystamy w naszym programie, są drzewa składni (ang. *abstract syntax trees*, w skrócie AST). Używamy ich do analizy warunków wstępnych, a także, w kilku przypadkach, do wyszukiwania odniesień do elementów programu.

Wyszukiwarka jest, obok kompilatora i kilku innych komponentów, elementem modelu Javy (czyli tej części modułu wspomagającego programowanie w Javie, która jest pozbawiona interfejsu graficznego). Służy do efektywnego i precyzyjnego znajdowania deklaracji i odniesień do elementów programu, np. klas, metod itp.

4.2. Przyjęte założenia projektowe

Roberts stworzył listę warunków, jakie musi powinno spełniać każde narzędzie do refaktoryzacji [19]. Tokuda rozwinął tę listę, czerpiąc ze swoich doświadczeń związanych z refaktoryzacją dużych projektów [22].

Uzupełniliśmy stworzoną przez tych autorów listę o dodatkowe warunki jakie, naszym zdaniem, są niezbędne do stworzenia narzędzia o wysokiej niezawodności. Wymogi w niej podane przyjęliśmy za założenia projektowe naszego programu.

Najpierw przedstawimy warunki podane przez Robertsa i Tokudę.

Integracja ze środowiskiem programistycznym Wspomaganie refaktoryzacji musi być ściśle zintegrowane ze środowiskiem programistycznym. Przeprowadzenie refaktoryzacji powinno odbywać się tak jak każda inna czynność wykonywana przez użytkownika (najczęściej przez wybranie opcji z menu).

Możliwość wycofania zmian Musi istnieć możliwość sprawnego wycofania zmian wprowadzonych do programu przez refaktoryzację. Dopuszczalne jest przy tym, by dokonane zmiany można było wycofać tylko do pewnego momentu. Przykładowo, jeśli po przeprowadzeniu refaktoryzacji wewnątrz pliku z kodem źrodłowym usuniemy ten plik, to wycofanie refaktoryzacji może okazać się niemożliwe.

Niezawodność Zachowywanie działania programu powinno być tak pełne, jak to możliwe (biorąc pod uwagę zawiłą konstrukcję języka bardzo trudne byłoby zagwarantowanie pełnej niezmienności działania – nawet pod nieobecność metod natywnych i mechanizmu odbicia). Przyjęta przez nas definicja niezmienności działania programu znajduje się w punkcie 2.1. Dopuszczalne są jednak sytuacje, gdy z powodów wydajności pominięta zostanie część analizy warunków wstępnych. Musi to być jednak traktowane jako sytuacja wyjątkowa, a pominięte warunki wstępne muszą być spełnione przez typowe programy.

Formatowanie kodu Formatowanie kodu powinno zostać niezmienione, tzn. takie, jakie było przed refaktoryzacją. Jedynym wyjątkiem są sytuacje, gdy narzędzie generuje nowy kod. Także wtedy należy, w miarę możliwości, stosować poprzednie formatowanie kodu.

Komentarze Komentarze muszą pozostać nienaruszone tzn. takie, jakie były przed refaktoryzacją. W przyszłości nasze narzędzie będzie wspierać także modyfikacje tzw. komentarzy dokumentujących – w języku Java są to tzw. komentarze JavaDoc. Komentarze te traktuje się wówczas jako część kodu. Jednak w żadnym wypadku narzędzie nie może usuwać bądź zmieniać położenia istniejących komentarzy w kodzie programu.

Umieszczenie nowotworzonego kodu Należy dać użytkownikowi możność wyboru miejsca, w którym zostanie umieszczony kod wygenerowany podczas refaktoryzacji.

Kontrola dostępności modyfikowanych plików Narzędzie służące do refaktoryzacji musi wykrywać sytuacje, w których pewne objęte refaktoryzacją pliki są niedostępne do zapisu. Należy wówczas poinformować o tym fakcie użytkownika. Wykrycie takiej sytuacji musi być częścią sprawdzania warunków wstępnych, więc odbywa się przed dokonaniem jakichkolwiek zmian w programie.

Nazwy dla nowotworzonych bytów Każdorazowo gdy narzędzie do refaktoryzacji tworzy nowe elementy programu – klasy, metody itp., użytkownik musi mieć możliwość ustalenia dla nich nazw. Nazwa wybrana przez użytkownika może jednak powodować błędy (np. powtarzające się nazwy zmiennych) bądź wpływać na zmianę zachowania programu. Należy wówczas poinformować użytkownika o tym fakcie.

Następujące warunki zostały przez nas dodane do listy podanej przez Robertsa i Tokudę:

Pełna lista niespełnionych warunków wstępnych W miarę możliwości należy zaprezentować użytkownikowi wszystkie niespełnione warunki wstępne – w podobny sposób, jak robią to kompilatory informując o błędach kompilacji. Użytkownik może wówczas naprawić wiele błędów na raz bez konieczności ponownego uruchamiania narzędzia po wprowadzeniu każdej poprawki z osobna.

Pełna lista wprowadzanych zmian w programie Należy pokazać użytkownikowi wszystkie zmiany w programie, jakie są potrzebne do wykonania refaktoryzacji. Ponadto użytkownik musi mieć możliwość niezgodzenia się na dowolną z tych zmian (choć wtedy prawie na pewno program nie będzie działał tak, jak poprzednio). Również w tym przypadku opcja wycofania zmian musi działać bez zarzutu.

Konwencje nazw Nie możemy niczego zakładać o przestrzeganiu bądź nieprzestrzeganiu konwencji nazw (konwencje nazw są opisane w specyfikacji języka [JLS 6.8]).

Analiza warunków wstępnych Wszelka analiza wykonalności refaktoryzacji musi zostać przeprowadzona bez dokonywania zmian w programie. Wewnętrzne struktury kompilatora (jak drzewa składni) także muszą bezwzględnie pozostać nietknięte podczas tej fazy refaktoryzacji.

Transakcyjność operacji Modyfikacja kodu źródłowego programu powinna odbywać się w sposób transakcyjny. Niepożądane są sytuacje, w których narzędzie ma zmodyfikować wiele plików i, z powodu błędu w programie lub błędu zewnętrznego, modyfikuje tylko

część i przerywa pracę. Ręczne przywrócenie systemu do pierwotnego stanu może okazać się wówczas bardzo pracochłonne[2].

Zaprezentowana lista wymagań jest niezależna od środowiska programistycznego w jakim powstaje rozważane narzędzie. Ostatnie założenie projektowe dotyczy środowiska, w którym zaimplementowany jest nasz program:

Nietykalność znaczników Znaczniki (opisane w punkcie 4.1) powinny pozostać nienaruszone. Stanowią one bardzo istotną część środowiska programistycznego, w którym zaimplementowane jest nasze narzędzie. Jeśli to możliwe, powinny zostać uaktualnione (ich atrybuty), by odzwierciedlać nowy stan systemu.

4.3. Przepływ sterowania

W tym punkcie podamy typowy przepływ sterowania w naszym programie podczas wykonywania refaktoryzacji.

1. Użytkownik wywołuje refaktoryzację przez wybranie opcji z menu i podanie niezbędnych informacji (np. nazwy dla nowej metody).
2. Część refaktoryzacji uruchamia w tym momencie wyszukiwarkę w celu efektywnego odszukania odniesień do modyfikowanych elementów programu.
3. Budowane są drzewa składni dla tych jednostek kompilacji, które będą modyfikowane[3].
4. Sprawdzane są warunki wstępne refaktoryzacji.
5. Jeżeli analiza wykryje niespełnione warunki wstępne, to użytkownik ma możliwość kontynuacji – mimo prawdopodobnych błędów, jakie pojawią się w programie. Uznaliśmy, że jest istotne, by nasze narzędzie nie przeszkadzało użytkownikowi w pracy. Jeżeli chce on wprowadzić zmiany do swego programu mimo ostrzeżeń, to ma taką możliwość.
6. Obliczane są wszystkie zmiany, jakie należy wprowadzić do programu, by przeprowadzić refaktoryzację.
7. Jeżeli użytkownik chce zobaczyć zmiany przez ich wprowadzeniem, to pełna ich lista jest prezentowana w interfejsie użytkownika.
8. Narzędzie wprowadza zmiany do refaktoryzowanego programu.
9. Na stos operacji do wycofania jest wkładana informacja o tym, w jaki sposób wycofać ostatnią przeprowadzoną refaktoryzację.

4.4. Wspomagane refaktoryzacje

Poniżej podajemy słowny opis refaktoryzacji wspomaganych przez nasz program. Pomijamy, ze względu na znaczną objętość, spis warunków wstępnych każdej z refaktoryzacji – przykłady podano w dodatku B.

[2] Transakcyjność operacji nie została jeszcze zrealizowana w naszym programie.

[3] Jest to przybliżenie. Niektóre refaktoryzacje wymagają analizy większego, a niektóre mniejszego zbioru jednostek kompilacji.

4.4.1. Wydzielenie Metody

Tworzy metodę z pewnej części innej metody. Wydzielana część może być listą instrukcji lub wyrażeniem. Zamienia wydzielaną część na wywołanie nowej metody, która musi mieć odpowiednią sygnaturę, modyfikatory, deklaracje zgłaszanych wyjątków. Wiele przykładów zastosowania tej niezwykle użytecznej refaktoryzacji (wraz z opisem *części* trudności związanych z jej automatyzacją) znajduje się w [6].

4.4.2. Zmiana Nazwy Pakietu

Powoduje zmianę nazwy wybranego pakietu wraz z uaktualnieniem odniesień do tego pakietu i wszystkich typów w nim zdefiniowanych.

4.4.3. Zmiana Nazwy Typu

Powoduje zmianę nazwy wybranego typu (zdefiniowanego nielokalnie, czyli nie we wnętrzu metody) wraz z uaktualnieniem wszystkich odniesień do tego typu oraz typów w nim zagnieżdżonych. Jeżeli wybrany typ jest niezagnieżdżony i ma tę samą nazwę co jednostka kompilacji, w której się znajduje, to również nazwa tej jednostki kompilacji jest zmieniana. Jest to konieczne w wypadku typów publicznych (wymóg specyfikacji języka) i zalecane w wypadku typów widocznych w obrębie pakietu.

4.4.4. Zmiana Nazwy Metody

Powoduje zmianę nazwy metody i aktualizację wszystkich odniesień do niej. W przypadku metod niewirtualnych zmieniana jest nazwa tylko jednej metody. Przy metodach wirtualnych zmieniana jest nazwa wybranej metody i wszystkich metod w hierarchii (począwszy od maksymalnie abstrakcyjnego typu deklarujacego tę metodę), które ją zastępują. W przypadku metod zdefiniowanych w interfejsach używa się algorytmu znajdowania metod, których nazwy muszą zostać zmienione wraz z wybraną. Algortym ten jest podany w punkcie 3.1.

4.4.5. Zmiana Nazwy Pola

Powoduje zmianę nazwy pola wraz z aktualizacją wszystkich odniesień do niego.

4.4.6. Zmiana Nazw Parametrów Metody

Powoduje zmianę nazwy jednego lub więcej parametrów wybranej metody (lub konstruktora) wraz z aktualizacją wszystkich odniesień.

4.4.7. Zmiana Nazwy Jednostki Kompilacji

Powoduje zmianę nazwy wybranej jednostki kompilacji. Jeżeli jest w niej zdefiniowany niezagnieżdżony typ o tej samej nazwie, co nazwa tej jednostki, to refaktoryzacja ta ma ten sam skutek, co opisana wcześniej Zmiana Nazwy Typu. W przeciwnym przypadku zmieniana jest jedynie nazwa samej jednostki kompilacji.

4.4.8. Przemieszczenie Jednostki Kompilacji do Innego Pakietu

Powoduje przemieszczenie jednostki kompilacji wraz z aktualizacją wszystkich odniesień do publicznego typu (jeżeli istnieje), który jest w niej zadeklarowany.

4.5. Główne części programu

Nasze narzędzie składa się z czterech głównych części: klasy `Refactoring`, klasy `Change`, analizatora drzew składni i interfejsu użytkownika.

4.5.1. Klasa `Refactoring`

Każdej refaktoryzacji odpowiada jedna klasa będąca podklasą abstrakcyjnej klasy `Refactoring` i implementujaca dwie metody: `checkPreconditions` oraz `createChange`.

Metoda `checkPreconditions` jest odpowiedzialna za sprawdzenie, czy spełnione są warunki wstępne dla danej refaktoryzacji, natomiast metoda `createChange` buduje i przekazuje obiekt klasy `Change` (opisanej w punkcie 4.5.2), który jest odpowiedzialny za przeprowadzenie zmian w programie. Metoda `checkPreconditions` przekazuje obiekt opisujący wynik sprawdzenia warunków wstępnych i zbierający wszystkie te, które nie zostały spełnione. Wszelkie możliwe wyniki podzielone są na cztery kategorie, podane tu w rosnącej kolejności ważności:

Informacje (ang. *infos*) nie oznaczają problemów. Użytkownicy mogą bezpiecznie ignorować komunikaty z tej kategorii. Służą one jedynie jako forma komunikacji programu z użytkownikiem.

Ostrzeżenia (ang. *warnings*) zgłaszane są np. wtedy, gdy przeprowadzenie refaktoryzacji spowoduje pojawienie się ostrzeżeń kompilatora lub w inny negatywny lecz nie destruktywny sposób wpłynie na modyfikowany program.

Przykładowy komunikat z tej grupy pojawia się wówczas, gdy użytkownik chce zmienić nazwę metody `main` lub klasy zawierającej taką metodę. Programu, który modyfikujemy nie będzie już można wywoływać z wiersza poleceń. Przestaną też działać skrypty go uruchamiające. Nie prowadzi to do błędów kompilacji. Należy jednak poinformować użytkownika o możliwości wystąpienia problemów i przypomnieć o konieczności aktualizacji skryptów itp.

Błędy (ang. *errors*) oznaczają, że można przeprowadzić modyfikację programu, ale prawie na pewno spowoduje to wystąpienie błędów kompilacji bądź, co gorsza, zmianę działania programu bez wystąpienia błędu kompilacji.

Przykładowo zmiana nazwy metody natywnej na pewno spowoduje przerwanie wykonania programu z błędem `UnsatisfiedLinkError` przy pierwszym wywołaniu tej metody – jeśli nie zostaną dokonane zmiany w kodzie natywnym. Nasz interfejs użytkownika zezwala na przeprowadzenie modyfikacji, gdy występują jedynie ostrzeżenia. Są one prezentowane użytkownikowi w postaci listy, z opisem i miejscem wystąpienia. Użytkownik może zrezygnować z przeprowadzenia modyfikacji programu bądź mimo wszystko jej dokonać. W takim przypadku, jeśli po przeprowadzeniu modyfikacji (i zapoznaniu się z błędami kompilacji, które ona wówczas zwykle powoduje) zdecyduje sie wycofać zmiany, to może tego dokonać.

Poważne Błędy (ang. *stop errors*) są zgłaszane w sytuacjach, gdy nie można przeprowadzić refaktoryzacji.

Przykładowo, użytkownik wybrał dla publicznej, niezagnieżdżonej klasy nazwę, która nie może być nazwą klasy i pliku bądź plik o tej nazwie już istnieje. Zgodnie ze specyfikacją języka takie klasy muszą znajdować się w pliku o nazwie identycznej z nazwą klasy, a system operacyjny nie zezwoli na stworzenie drugiego pliku o tej samej nazwie.

Podobnie błąd zgłaszany jest wówczas, gdy podczas wykonywania refaktoryzacji musiałby zostać zmodyfikowany plik, do którego nie ma dostępu (np. jest otwarty tylko do odczytu).

Interfejs użytkownika prezentuje wszystkie te komunikaty (jeśli występują) i zezwala na wykonanie modyfikacji tylko wówczas, gdy nie ma wśrod nich Poważnych Błędów. Jednak, jak wspomniano poprzednio, w przypadku występowania Błędów oraz Ostrzeżeń program, po wykonaniu modyfikacji, najprawdopodobniej przestanie działać (wystąpią błędy kompilacji bądź błędy wykonania) lub jego zachowanie będzie zmienione.

Metodę `createChange` wywołuje się po sprawdzeniu warunków wstępnych. Buduje ona obiekt klasy `Change` (opisanej w punkcie 4.5.2), który następnie będzie odpowiedzialny za modyfikację kodu źródłowego programu (samo wywołanie metody `createChange` nie dokonuje żadnych zmian w programie).

4.5.2. Klasa `Change`

Wszystkie refaktoryzacje są zaimplementowane przy użyciu obiektów klasy `Change`. Są one przykładami wzorca Command [7] i odpowiadają za modyfikację kodu źródłowego. Wszystkie obiekty klasy `Change` muszą implementować metodę `perform`, która przeprowadza modyfikację kodu źródłowego i przekazuje jako wynik obiekt klasy `Change`, służący do przeprowadzenia modyfikacji odwrotnej.

Do łączenia wielu zmian razem używamy wzorca Composite [7]. Podobnie jak w programie Robertsa, również nasze zmiany są odwracalne, zarówno pojedynczo jak i razem, jako złożenie (odwrócenie sekwencji zmian polega na odwróceniu zmian składowych w odwrotnej kolejności). To czyni je bardzo efektywnym i elastycznym narzędziem. Każda, dowolnie skomplikowana, modyfikacja kodu źródłowego jest wykonywana jako ciąg zmian, z których każda potrafi odwrócić efekt swego działania.

Dodatkowo, każda zmiana może być aktywna lub nieaktywna. Zazwyczaj jest to konsekwencją tego, że użytkownik zdecydował, iż pewne zmiany są niepożądane i nie zgodził się na ich przeprowadzenie (jak wspomnieliśmy, programista ma możliwość zobaczenia wszystkich zmian, jakie narzędzie ma zamiar dokonać i niezgodzenia się na dowolną z nich). Taka zmiana staje się wówczas nieaktywna, co oznacza, że wywołanie jej metody `perform` nie ma żadnego skutku, a przekazywany jest obiekt klasy `NullChange` (przykład wzorca NullObject [6]). Dezaktywacja zmian prawie zawsze prowadzi do błędów kompilacji (ponieważ narzędzie sugeruje wprowadzenie zmiany, która jest potrzebna do przeprowadzenia refaktoryzacji, a na którą użytkownik się nie zgadza), musi zatem zawsze zostać dokonana na polecenie użytkownika – wszystkie zmiany są zawsze domyślnie aktywne. Mechanizm wycofywania radzi sobie z wycofaniem dowolnej kombinacji aktywnych i nieaktywnych zmian.

Wszystkie refaktoryzacje zostały zaimplementowane przy użyciu niewielkiej liczby podklas klasy `Change`. Poniżej podajemy ich spis:

- modyfikacje tekstowe
  - ReplaceText
  - DeleteText
  - InsertText
  - MoveText
- operacje na plikach

  - RenameFile
  - CreateFile
  - DeleteFile

- operacje na pakietach
  - RenamePackage
  - CreatePackage
  - DeletePackage

Modyfikacje tekstowe oraz operacje na plikach są niezależne od języka Java – mogą służyć do manipulowania dowolnymi plikami tekstowymi.

4.5.3. Analizator drzew składni

Do analizy wielu warunków wstępnych refaktoryzacji używamy drzew składni. Każdy węzeł takiego drzewa odpowiada pewnemu elementowi programu. Analizator jest przykładem wzorca Visitor [7] – na początku analizy każdego węzła wywoływana jest metoda `visit`, na końcu zaś `endVisit`. Odmiennie niż Roberts [19] zdecydowaliśmy się nie używać drzew składni do modyfikacji programu – dokonujemy tego bezpośrednio manipulując zawartościa plików z kodem źródłowym. Język Java ma znacznie bardziej rozbudowaną składnię niż Smalltalk, (którym zajmował się Roberts), wobec czego drzewa składni mają bardziej skomplikowaną budowę. Ponadto drzewa składni, użyte w kompilatorze, którym dysponujemy, nie niosą pełnej informacji o programie – zawierają jedynie te dane, które są niezbędne do analizy typów, generacji kodu wynikowego i tworzenia komunikatów o błędach kompilacji. Informacje o komentarzach, formatowaniu kodu, pustych blokach itp. są nieobecne.

Ze względu na wydajność nie cały program znajduje się jednocześnie w pamięci, a drzewa składni tworzone są jedynie dla tych jednostek kompilacji, dla których tego jawnie zażądamy. Jest to kosztowna operacja (trzeba wczytać jednostkę kompilacji i dokonać jej analizy składniowej; sprawdza się także typy wszystkich odwołań i wyrażeń). Wskazane jest wobec tego, by analiza oparta na drzewach składni ograniczała się do możliwie małej liczby jednostek kompilacji.

4.5.4. Interfejs użytkownika

Przy projektowaniu interfejsu użytkownika staraliśmy się wziąć pod uwagę oczekiwania użytkowników mniej doświadczonych, których podstawowym wymaganiem jest niezawodność programu i prostota jego obsługi, a także bardziej wymagających, dla których istotna jest możliwość kontrolowania działania programu i dopasowania go do własnych potrzeb.

Zdecydowaliśmy się na użycie asystentów (ang. *wizards*) – szeroko stosowanego rozwiązania polegającego na prowadzeniu użytkownika przez pewien proces (na niektórych etapach wymagający jego interwencji). W naszym programie pierwszy etap polega na pobraniu od użytkownika informacji niezbędnych do przeprowadzenia refaktoryzacji, następnie sprawdzane są warunki wstępne i prezentowana lista tych, które nie są spełnione. Kolejny etap to prezentacja wszystkich zmian, które muszą zostać wprowadzone do programu w celu przeprowadzenia refaktoryzacji. Zmiany są prezentowane w specjalnym oknie. Użytkownik widzi, dla każdej zmiany osobno, obecny stan systemu i ten, w którym system się znajdzie po przeprowadzeniu refaktoryzacji. Dla zmian złożonych jest to złożenie efektów zmian składowych. Na tym etapie użytkownik może nie zgodzić się na dowolną z proponowanych zmian – nie zostanie ona wówczas wprowadzona do programu.

4.6. Wydajność i niezawodność narzędzia

Środowisko IBM WebSphere Studio Workbench jest napisane w języku Java i tworzone przy użyciu samego siebie[4]. Umożliwia nam to testowanie wydajności i niezawodności naszego narzędzia na programie bardzo dużej wielkości (ponad 500000 wierszy kodu źródłowego). Nasze doświadczenia wskazują jednoznacznie, że efektywna refaktoryzacja nawet dużych programów, obejmująca rozległą analizę warunków wstępnych, jest możliwa.

Podczas pracy i testowania nie wykryliśmy problemów ze skalowalnością. Jest to, po części, zasługa bardzo wydajnej infrastruktury, jaką dysponujemy (szybki kompilator oraz wyszukiwarka). W pamięci przechowywane są jedynie niezbędne informacje. Nasze obserwacje (niepoparte szczegółową analizą) wskazują, że czas działania refaktoryzacji nie zależy w zauważalny sposób od wielkości programu, lecz jedynie od liczby plików, których dotyczy refaktoryzacja.

Wydajność narzędzia pokażemy na przykładach refaktoryzacji Zmiana Nazwy Pakietu oraz Zmiana Nazwy Typu. Wykonanie refaktoryzacji Zmiana Nazwy Pakietu dla pakietu, do którego istnieje ok. 100 odniesień[5] w całym programie trwa ok. 15–17 sekund (komputer 500MHz CPU, 190MB RAM). Z tego ok. 70% zajmuje wyszukanie odniesień, analiza warunków wstępnych oraz obliczenie modyfikacji programu niezbędnych do przeprowadzenia refaktoryzacji. Pozostały czas zabiera przeprowadzenie obliczonych zmian w programie i aktualizacja stanu środowiska programistycznego. Na wycofanie tej refaktoryzacji potrzeba tylko ok. 5 sekund – ponieważ nie wymaga to wyszukiwania odniesień ani analizy programu.

Na przeprowadzenie refaktoryzacji Zmiana Nazwy Typu dla często używanej klasy, do której istnieje ok. 500 odniesień potrzeba ok. 40 sekund (30 sekund wyszukiwanie odniesień, analiza warunków wstępnych oraz obliczenie koniecznych modyfikacji programu, 10 sekund wprowadzenie zmian i aktualizacja stanu środowiska). Wycofanie tej refaktoryzacji trwa ok. 7–8 sekund.

Z podanych liczb wynika, że możliwa jest efektywna refaktoryzacja nawet dużych programów, obejmująca całość języka programowania i szczegółową analizę warunków wstępnych. Pokazują one także jak wiele czasu (potrzebnego na identyfikację niezbędnych zmian w programie, ich wprowadzenie i przetestowanie) można zaoszczędzić przeprowadzając refaktoryzację z użyciem wydajnych narzędzi.

Wysoką niezawodność narzędzia zapewniono dzięki używaniu programu na codzień przez wielu programistów (także autorów) oraz zestawowi zautomatyzowanych testów, obejmującemu ok. 100 szczegółowych przypadków wykonania każdej z refaktoryzacji. Podczas pracy nad narzędziem wspomniany zestaw testów przeprowadzano po wprowadzeniu każdej istotnej zmiany do programu – co najmniej kilka razy w tygodniu. Dla każdego wykrytego w programie błędu tworzono przypadek użycia pokazujący jego wystąpienie. Błąd uznawano za naprawiony dopiero wtedy, gdy pokazujący go przypadek użycia oraz wszystkie pozostałe przypadki zawarte w zestawie testów dawały wyniki zgodne z oczekiwaniami. Metoda ta pozwoliła na wyeliminowanie w znacznym stopniu ponownego pojawiania się naprawionych wcześniej błędów.

Celem części testów było sprawdzenie zachowania narzędzia w sytuacjach, gdy jeden lub więcej warunków wstępnych refaktoryzacji nie był spełniony. Inne testy służyły do sprawdzenia, czy zmiany wprowadzane w refaktoryzowanym programie były zgodne z przewidywaniami (porównywano w nich, znak po znaku, oczekiwaną zawartość plików z kodem źródłowym z tym, co faktycznie było efektem refaktoryzacji programu.).

[4] Pracując przy tworzeniu IBM WebSphere Studio Workbench używamy tego środowiska programistycznego jako narzędzia pracy.

[5] Odniesieniem do pakietu jest deklaracja importu (pojedynczego typu lub na żądanie) typu zdefiniowanego w tym pakiecie lub w pełni kwalifikowane odniesienie do takiego typu.

Rozdział 5

Podsumowanie

W tym punkcie prezentujemy główne osiągnięcia niniejszej pracy. O ile nie wskazano inaczej, wszystkie opisane wyniki są autorstwa Adama Kieżuna.

1. Identyfikacja tych właściwości języka Java (ściślej, programów napisanych w tym języku), których występowanie sprawia, że niektóre, nawet koncepcyjnie proste, refaktoryzacje stają się trudne w realizacji. Następnie, opis i przedstawienie na przykładach problemów powstających z powodu występowania wskazanych cech języka oraz dyskusja możliwych rozwiązań.

   W szczególności, w pracy omówiono:

   - wielokrotne dziedziczenie interfejsów,
   - użycie deklaracji importu,
   - wykorzystanie typów zagnieżdżonych i lokalnie definiowanych,
   - przesłanianie, ukrywanie i zaciemnianie nazw,
   - przeciążanie nazw metod,
   - użycie metod natywnych.

2. Opis efektu fali występującego w związku z użyciem wielokrotnego dziedziczenia oraz sformułowanie algorytmu (uwzględniającego to zjawisko) znajdującego wszystkie metody objęte refaktoryzacją wraz z omówieniem możliwości jego wykorzystania.

3. Podanie warunków wstępnych wynikających z występowania metod natywnych.

4. Opis kilku, specyficznych dla języka Java, refaktoryzacji (wraz ze szczegółowym wyliczeniem warunków wstępnych, uwzględniającym opisane w punkcie 1 właściwości języka):

   - Zmiana Nazwy Pakietu,
   - Przemieszczenie Jednostki Kompilacji do Innego Pakietu.

5. Stworzenie szczegółowego opisu (uwzględniającego wymienione w punkcie 1 właściwości języka) warunków wstępnych dla kilku, rozważanych wcześniej, refaktoryzacji:

   - Zmiana Nazwy Typu,
   - Zmiana Nazwy Pola,
   - Zmiana Nazwy Metody.

‘

6. Rozwinięcie listy, podanej przez Robertsa [19] i uzupełnionej przez Tokudę [22], warunków, które powinno spełniać każde narzędzie do refaktoryzacji.

7. Projekt i implementacja wsparcia refaktoryzacji w środowisku IBM WebSphere Studio Workbench (program jest dostępny pod adresem [24]). Ten punkt został zrealizowany przez autora niniejszej pracy wspólnie z dr Dirkiem Bäumerem (szczegóły dotyczące podziału pracy wykonanej przy projekcie i implementacji podano w rozdziale 4).

   Cechy wykonanego narzędzia to:

   - Implementacja kilku refaktoryzacji uwzględniająca całość języka Java i dokonująca szczegółowej analizy warunków wstępnych.
   - Wysoka niezawodność i skalowalność sprawdzona przy refaktoryzacji dużych programów.
   - Możliwość łatwego wycofywania wprowadzonych modyfikacji kodu źródłowego.
   - Dobra integracja ze środowiskiem programistycznym.
   - Elastyczna architektura umożliwiająca łatwą implementację nowych refaktoryzacji.

8. Stworzenie opisu zagadnień związanych z integracją wsparcia refaktoryzacji ze środowiskiem programistycznym. Wyniki te opublikowaliśmy wraz z dr Erichem Gammą i dr Dirkiem Bäumerem w artykule [1].

Możliwości kontynuacji pracy

Refaktoryzacja to nadal dość nowa dziedzina badań. Pierwsza praca prezentująca to zagadnienie [17] pochodzi z roku 1992, a większość wyników badań opublikowano w ciągu ostatnich trzech lat. Znaczny wzrost zainteresowania tą dziedziną pozwala mieć nadzieję na osiągnięcie w najbliższym czasie ciekawych rezultatów.

Badania, których wynikiem jest niniejsza praca, można kontynuować na kilka sposobów.

1. Rozszerzenie podanej w dodatku B listy refaktoryzacji posiadających szczegółowy opis warunków wstępnych.

2. Sformalizowanie określenia warunków wstępnych.

3. Sformalizowanie opisu zmian w programie wprowadzanych przez refaktoryzacje.

4. Zbadanie refaktoryzacji w kontekście innych języków programowania.

5. Zbadanie zagadnienia refaktoryzacji programów napisanych w kilku językach programowania. W punkcie 3.5 omówiliśmy wpływ wykorzystania metod natywnych na refaktoryzację programów w języku Java. Ciekawe byłoby zbadanie możliwości zbudowania narzędzia umiejącego refaktoryzować programy korzystające z tego mechanizmu (tzn. narzędzia potrafiącego analizować i modyfikować zarówno kod Javy, jak i kod natywny).

   Język Java jest bardzo często wykorzystywany w połączeniu z językami skryptowymi, takimi jak Java Script, a także w powiązaniu z takimi mechanizmami jak Java Server Pages. Stworzenie narzędzia potrafiącego refaktoryzować tego typu systemy byłoby bardzo ciekawe i pożądane, ponieważ kombinacja wielu języków programowania w jednym systemie często prowadzi do znacznego wzrostu stopnia komplikacji kodu źródłowego.

Wsparcie refaktoryzacji w IBM WebSphere Studio Workbench jest nadal w początkowej fazie tworzenia. Do chwili obecnej (lipiec 2001, wersja R0.9 produktu) zaimplementowaliśmy jedynie część planowanych refaktoryzacji. Naszą pracę chcemy kontynuować w następujący sposób:

1. Zaimplementowanie większej liczby refaktoryzacji.

2. Wprowadzenie transakcyjności operacji.

3. Zwiększenie zasięgu możliwości wycofywania zmian. Obecnie wycofanie refaktoryzacji jest możliwe tylko do chwili, gdy użytkownik zacznie wykonywać inne modyfikacje kodu. Chcielibyśmy zintegrować mechanizm wycofywania zmian używany w edytorach z mechanizmem użytym w naszym narzędziu do refaktoryzacji.

4. Ułatwienie wprowadzania zmian na wyższym poziomie abstrakcji. Jak podano w punkcie 4.5.2, wszystkie modyfikacje są obecnie wykonywane bezpośrednio w kodzie źródłowym refaktoryzowanego programu. Mimo, że jest to bardzo elastyczne podejście, to wprowadzenie możliwości operowania elementami programu, a nie fragmentami plików z kodem źródłowym ułatwi implementację kolejnych refaktoryzacji.

5. Podczas wykonywania refaktoryzacji, aktualizowanie komentarzy dokumentujących (JavaDoc). Jest to ważne np. przy refaktoryzacjach Zmiana Nazwy Metody, Zmiana Położenia Typu itp.

Dodatek A

Refaktoryzacje proste

W tym dodatku prezentujemy spis refaktoryzacji prostych, podanych przez Williama Opdyke'a. Jest on powtórzony z pracy [17] i znajduje się w tym miejscu z uwagi na wygodę czytelnika. Nazwy podanych refaktoryzacji są nieco zmienione w stosunku do podanych w pracy Opdyke'a – w szczególności dostosowano je do terminologii stosowanej obecnie w literaturze.

- Stworzenie elementu programu
  - Stworzenie Pustej Klasy (ang. *Create Empty Class*)
  - Dodanie Nowej Metody do Klasy (ang. *Add Method to Class*)
  - Dodanie Nowej Zmiennej do Klasy (ang. *Add Field to Class*)
- Usunięcie elementu programu
  - Usunięcie Nieużywanego Pola (ang. *Delete Unused Field*)
  - Usunięcie Nieużywanej Metody (ang. *Delete Unused Method*)
  - Usunięcie Nieużywanie Klasy (ang. *Delete Unused Class*)
- Zmiana w elemencie programu
  - Zmiana Nazwy Klasy (ang. *Rename Class*)
  - Zmiana Nazwy Metody (ang. *Rename Method*)
  - Zmiana Nazwy Pola (ang. *Rename Field*)
  - Zmiana Typu Pola (ang. *Change Type of Field*)
  - Zmiana Widoczności Elementu Programu (ang. *Change Access Modifier*)
  - Dodanie Parametru Metody (ang. *Add Method Parameter*)
  - Usunięcie Parametru Metody (ang. *Remove Method Parameter*)
  - Zmiana Kolejności Parametrów Metody (ang. *Reorder Parameters*)
  - Wstawienie Kodu do Pustej Metody (ang. *Add Code to Empty Method*)
  - Usunięcie Całości Kodu z Metody (ang. *Delete Code From Method*)
  - Zmiana Odniesień do Pola na Wywołania Funkcji (ang. *Abstract Field*)
  - Wydzielenie Metody (ang. *Extract Method*)
  - Rozwinięcie Treści Metody w Miejscu Wywołania (ang. *Inlining Method*)
  - Zmiana Nadklasy Klasy (ang. *Reparent Class*)

- Zmiana położenia elementu programu
  - Przesunięcie Pola do Nadklasy (ang. *Push Up Field*)
  - Przesunięcie Pola do Podklasy (ang. *Push Down Field*)

Dodatek B

Przykładowe refaktoryzacje

W tej części pracy przedstawimy opisy kilku refaktoryzacji zaimplementowanych w wykonanym przez nas narzędziu. Przykłady ilustrujące sytuacje, kiedy jeden lub więcej z podanych warunków wstępnych nie jest spełniony (i problemy z tego wynikające) zostały podane w rozdziale 3 bądź zupełnie bezpośrednio wynikają z podanych tam przykładów.

Warunki wspólne dla wszystkich refaktoryzacji

1. Wszystkie pliki, które mają ulec modyfikacji podczas refaktoryzacji są dostępne do zapisu.
2. Wszystkie pliki, które mają ulec modyfikacji podczas refaktoryzacji zawierają kod źródłowy. Zakładamy bowiem, że nie możemy modyfikować kodu wynikowego.

B.1. Zmiana Nazwy Typu

Dane: typ `t` (nazwę typu `t` również będziemy oznaczać przez `t`), nowa nazwa (prosta) `n` (załóżmy, że `n` jest różna od `t`)
Założenie: `t` nie jest typem lokalnie zdefiniowanym

1. Nazwa `n` jest dozwoloną według specyfikacji języka nazwą typu.
2. Żaden typ zagnieżdżony w `t` (na dowolnym poziomie zagnieżdżenia) nie ma nazwy `n`.
3. Tylko dla typów zagnieżdżonych: żaden z typów, w których zagnieżdżony jest `t` nie ma nazwy `n`.
4. Żaden typ lokalnie zdefiniowany w `t` (bądź w typie zagnieżdżonym w `t`) nie ma nazwy `n`.
5. Dla typów niezagnieżdżonych: nie istnieje żaden niezagnieżdżony typ o nazwie `n` w tym pakiecie (w żadnej jednostce kompilacji). Dla typów zagnieżdżonych: nie istnieje żaden typ o nazwie `n` zdefiniowany w typie bezpośrednio zawierającym `t`.
6. Jeśli typ jest niezagnieżdżony i publiczny, to w katalogu, w którym znajduje się jednostka kompilacji zawierająca `t`, nie może istnieć jednostka kompilacji o nazwie `n.java`.
7. Ani w `t`, ani w żadnym typie zagnieżdżonym w `t` (bądź zadeklarowanym lokalnie w `t` lub w typie zagnieżdżonym w `t`) nie istnieje żadna metoda natywna.

8. Ani `t`, ani żaden typ zagnieżdżony w `t` (bądź zadeklarowany lokalnie w `t` lub w typie zagnieżdżonym w `t`) nie jest występuje jako typ parametru formalnego żadnej metody natywnej (w całym systemie).

9. Żadna jednostka kompilacji nie importuje (import pojedynczego typu) `t` jednocześnie deklarując niezagnieżdżony typ o nazwie `n` [JLS2 7.5.1] (w szczególności, jednostka kompilacji, w której zdefiniowany jest typ `t`, nie może importować typu o nazwie `n`).

10. W żadnym miejscu programu, gdzie występuje odniesienie do `t`, nazwa `n` nie jest użyta do określenia parametru metody, zmiennej lokalnej, widocznego pola (na żadym poziomie dziedziczenia lub zagnieżdżenia) bądź widocznego pakietu.

11. W żadnym miejscu programu, gdzie występuje odniesienie do `t`, nazwa `n` nie jest użyta do określenia widocznego typu (tylko wówczas, gdy istnieją do owego typu odniesienia używające jego nazwy prostej).

12. Żadna jednostka kompilacji nie importuje `t` (import na żądanie) i typu o nazwie `n` (import pojedynczego typu) [JLS2 7.5.1] (jeśli w owej jednostce kompilacji znajdują się odniesienia do `t` używające nazwy prostej typu).

13. Jednostka kompilacji, w której zdefiniowano typ `t` nie może importować (import na żądanie) typu o nazwie `n` (dokładniej, jeśli w tej jednostce kompilacji istnieją odniesienia do typu o nazwie `n` używające jego nazwy prostej).

14. Żadna jednostka kompilacji nie importuje (import pojedynczego typu) jednocześnie typu `t` i typu o nazwie `n`.

15. Ani `t` ani żaden z typów zagnieżdżonych w `t` nie deklaruje metody (`public static void`) o sygnaturze `main(java.lang.String[])`.

16. Typ `t` nie jest zadeklarowany w pakiecie `java.lang`.

Nawet zakładając, że nie istnieje w systemie żaden typ o nazwie prostej `n`, musimy sprawdzać warunki 1,6,7,8,10,15. Sprawdzenie dwóch z nich wymaga analizy całego programu – a przynajmniej tych jego części, gdzie typ `t` jest widoczny:

a. warunek 7 – kompilator nigdy nie wykaże żadnego błędu, jednak, jak wspomniano w punkcie 3.5, program zakończy działanie (z komunikatem błędu) przy pierwszym wywołaniu owej metody natywnej. Wykrycie natury tego błędu nie jest możliwe z poziomu języka Java i wymaga analizy kodu C/C++. Do jego naprawienia potrzebne jest wprowadzenie poprawek do kodu natywnego i rekompilacja bibliotek.

b. warunek 10 – w rozdziale omawiającym zaciemnianie pokazaliśmy, w jaki sposób prosta nazwa może być interpretowana jako nazwa zmiennej, typu bądź pakietu. By to sprawdzić musimy, dla każdego odniesienia do typu `t`, zweryfikować, że nowa nazwa nie jest już użyta przez zmienną bądź pole oraz, że nowe nazwa nie przesłoni widoczności nazwy jakiegokolwiek pakietu. Analiza ta wymaga przejścia całego stosu zasięgów widoczności oraz obu hierarchii: zagnieżdżeń oraz dziedziczenia. Ze względu na wielokrotne dziedziczenie i występowanie typów zagnieżdżonych jest to, w ogólnym przypadku, dowolnie skomplikowany acykliczny graf skierowany typów.

B.2. Zmiana Nazw Parametrów Metody

Dane: metoda `m` (o `n` parametrach, gdzie $n > 0$), nowe nazwy parametrów `p1..pn`)

1. Żadna z nowych nazw parametrów nie powtarza się wśród `p1..pn`.
2. Każda z nowych nazw parametrów jest dopuszczalna (tzn. jest dopuszczalną według specyfikacji języka nazwą parametru metody).

Jeżeli metoda posiada treść (czyli nie jest abstrakcyjna ani natywna), wówczas musimy dodatkowo sprawdzić, że:

1. Żadna ze zmiennych lokalnych metody (zadeklarowanych w niej, ale nie wewnątrz typów lokalnie w niej zdefiniowanych) nie ma nazwy równej jednej z nazw `p1..pn`. Przy czym do zmiennych lokalnych metody zaliczają się również nazwy wyjątków zadeklarowanych w klauzulach `catch`, użytych wewnątrz metody.
2. W żadnym miejscu metody nowa nazwa żadnego parametru nie przesłoni widoczności żadnego pola oraz nie zaciemni widoczności żadnego typu ani pakietu.

B.3. Zmiana Nazwy Pakietu

Na potrzeby pracy zakładamy, że na cały program składa się lista pakietów. Jest to uproszczenie, aczkolwiek niewielkie i nie wpływające w znaczacym stopniu na analizę warunków wstępnych. Rzeczywiste programy w języku Java mają często bardziej skomplikowaną strukture. Program składa się z projektów, projekty z folderów (ang. *source folders*), a dopiero foldery z pakietów. Dodatkowo w skład projektów mogą wchodzić pliki wynikowe (JAR), które również składaja się wewnętrznie z folderów i pakietów. Struktura ta nie ma jednak wpływu na zasięgi widoczności nazw itp. Służy ona jedynie grupowaniu koncepcyjnie powiązanych ze sobą części programu.

Dane: pakiet `p`, nowa nazwa `n` (zakładamy, że `n` jest różna od obecnej nazwy `p`)
Założenie: `p` ma nazwę, tzn. nie jest pakietem domyślnym, oraz `n` nie jest puste. Czyli nie dopuszczamy zmiany nazwy pakietu domyślnego ani uczynienia innego pakietu domyślnym.

Przyjmujemy tu następujące założenie, dopuszczalne przez specyfikację języka: uznajemy, że każdy pakiet stanowi samodzielny byt tzn. pod–pakiety (np. `java.lang.ref` jest pod–pakietem `java.lang`) traktujemy jak oddzielne elementy programu. Jest to dopuszczalne, ponieważ istnienie pakietów i pod–pakietów nie ma żadnego wpływu na widoczność nazw w programie.

1. Nazwa `n` jest nazwą pakietu dopuszczalną według specyfikacji języka.
2. Pakiet o nazwie `n` nie istnieje w programie.
3. Żaden typ zadeklarowany w `p` nie deklaruje metody natywnej.
4. Żadna metoda natywna w systemie nie ma parametru, którego typ jest zadeklarowany w `p`.
5. Żaden typ zadeklarowany w `p` nie deklaruje metody (`public static void`) o sygnaturze `main(java.lang.String[])`.

6. W żadnym miejscu programu, gdzie znajduje się odniesienie do `p` (czyli zazwyczaj w pełni kwalifikowane odniesienie do typu zadeklarowanego w `p`) nazwa `n` nie jest zaciemniona przez nazwę zmiennej lokalnej (także parametrów metody), widocznego pola ani typu.

7. Pakiet `p` nie jest pakietem specjalnym `java.lang`.

B.4. Zmiana Nazwy Pola

Dane: pole `f`, nowa nazwa `n` (zakładamy, że `n` jest różna od obecnej nazwy `f`)

1. Nazwa `n` jest dopuszczalną według specyfikacji języka nazwą pola.

2. Żadne pole zadeklarowane w typie deklarującym `f` nie ma nazwy `n`.

3. W żadnym miejscu programu, gdzie znajdują się odniesienia do `f`, nazwa `n` nie jest widoczna jako nazwa innego pola, które przesłania widoczność `f`. Nie jest również widoczna jako nazwa zmiennej lokalnej ani parametru metody.

4. W żadnym miejscu programu, gdzie znajdują się odniesienia do `f`, nazwa `n` nie zaciemnia nazwy żadnego typu ani pakietu.

B.5. Przemieszczenie Jednostki Kompilacji do Innego Pakietu

Jednostka kompilacji w języku Java to plik (z rozszerzeniem `.java`) zawierający pewien zbiór typów. Jednostka kompilacji składa się z deklaracji pakietu, deklaracji importu (często wielu) oraz definicji typów. Tylko jeden z typów niezagnieżdżonych, zadeklarowanych w danej jednostce kompilacji może być publiczny. Pozostałe typy niezagnieżdżone muszą być zadeklarowane jako widoczne w pakiecie (ang. *default visibility*). Każda jednostka kompilacji może zawierać wiele typów, często zagnieżdżonych w sobie. W każdym z tych typów mogą znajdować się (nie wymagające importu) odniesienia do dowolnego elementu zdefiniowanego w dowolnej jednostce kompilacji w bieżącym pakiecie. Analiza tego, czy każde z tych odwołań będzie nadal poprawne po zmianie pakietu jest, jak się okazuje, dość skomplikowana.

Dane: jednostka kompilacji `cu`, nowy pakiet `p`

Nazwy wszystkich jednostek kompilacji muszą mieć przyrostek `.java`. Będziemy go wobec tego opuszczać i oznaczać nazwę `cu` przez `n`. Również nazwę typu (jak i sam typ) zadeklarowanego w `cu` będziemy oznaczać przez `n`, jeżeli będzie ona tożsama z nazwą `cu`. Przykładowo: jeżeli `cu` ma nazwę `A.java` i deklaruje (niezagnieżdżony) typ `t` o nazwie `A`, to będziemy używać `A` jako nazwy `cu` oraz jako nazwy `t`.

Oznaczenie: `cu.package` oznacza pakiet, w którym `cu` znajduje się przed refaktoryzacją.

1. W `p` nie znajduje się inna jednostka kompilacji o nazwie `n`.

2. W `p` nie zadeklarowano żadnego niezagnieżdżonego typu o nazwie `n`.

3. Żaden typ zadeklarowany w `cu` nie deklaruje metody natywnej.

4. Żaden typ zadeklarowany w `cu` nie deklaruje metody `public static void` o sygnaturze `main(java.lang.String[])`.

5. Żadna metoda natywna w systemie nie ma parametru, którego typ jest jednym z typów zadeklarowanych w `cu`

6. W `cu` nie ma żadnych odniesień do elementów o domyślnej widoczności (widocznych tylko wewnątrz pakietu, w którym są zadeklarowane), zadeklarowanych w innej jednostce kompilacji w `cu.package`.

7. W `cu` nie znajdują się deklaracje żadnych elementów o domyślnej widoczności, do których odniesienia znajduja się w dowolnej innej niż `cu` jednostce kompilacji w `cu.package`.

8. W `cu` nie ma żadnych odniesień do elementów o chronionej (ang. *protected*) widoczności, zadeklarowanych w innej niż `cu` jednostce kompilacji w `cu.package` – chyba, że są one nadal widoczne tzn. znajdują się w podtypach typów je deklarujących. Dodatkowo każdy z typów deklarujących rozważane elementy musi być widoczny na zewnątrz pakietu czyli być publiczny i niezagnieżdżony bądź publiczny lub chroniony i zagnieżdżony w typie widocznym na zewnątrz pakietu (stosując tę definicję rekursywnie).

9. W `cu` nie znajdują się deklaracje żadnych elementów o chronionej widoczności, do których odniesienia znajdują się w dowolnej innej niż `cu` jednostce kompilacji w `cu.package` – chyba, że są one nadal widoczne tzn. znajdują się w podtypach typów je deklarujących.

10. W żadnej jednostce kompilacji w `p` żaden typ zdefniowany w `cu` nie przesłania widoczności innego typu, zaimportowanego „na żądanie" w tej jednostce.

11. W żadnej jednostce kompilacji w `p` nazwa żadnego typu zdefiniowanego w `cu` nie zaciemnia nazwy żadnego pakietu.

12. Zarówno `cu.package` jak i `p` sa rózne od pakietu specjalnego `java.lang`.

By zaimplementować tę refaktoryzację musimy sprawić, by wszystkie typy publiczne, zadeklarowane w `cu.package`, do których znajdują się odniesienia w `cu` (używające prostych, a nie w pełni kwalifikowanych nazw) były nadal widoczne w `cu` po modyfikacji programu. Możemy to osiągnąć dodając do `cu` deklaracje importu pojedynczego typu dla wszystkich tych typów – każdego z osobna. Zauważmy, że nie musimy sprawdzać, czy deklaracja innego typu o takiej samej nazwie prostej już istnieje w `cu`. Gdyby tak bowiem było, przesłaniałaby ona widoczność typów importowanych domyślnie, czyli zadeklarowanych w tym samym pakiecie. Nie trzeba także sprawdzać, czy dodana deklaracja importu przesłania widoczność jakiegoś typu importowanego w `cu` na żądanie. Gdyby tak było, to kompilator wykazywałby błędy w `cu` przed refaktoryzacją (niejednoznaczne odniesienia do typów).

Należy zauważyć, że w wielu przypadkach potrzebne będą dodatkowe deklaracje importu również w innych jednostkach kompilacji. Dokładniej: należy to zrobić w tych jednostkach kompilacji, w których znajdują się proste, nie w pełni kwalifikowane odniesienia do tych elementów zadeklarowanych w `cu`, które będą widoczne po przeniesieniu `cu` do `p` (czyli publiczych oraz chronionych). Również w tym przypadku można ograniczyć się do użycia deklaracji importu pojedynczego typu (uzasadnienie jest analogiczne do podanego dla poprzedniego przypadku).

Ciekawym rozszerzeniem tej refaktoryzacji byłoby umożliwienie przemieszczania więcej niż jednej jednostki kompilacji na raz. Wymagałoby to sprawdzenia nieco innych warunków wstępnych. Ograniczymy się do podania zarysu potrzebnej analizy.

Załóżmy, że wszystkie rozważane jednostki kompilacji przed refaktoryzacją znajdują się w tym samym pakiecie (refaktoryzacja polegająca na przeniesieniu jednostek kompilacji z wielu pakietów do jednego jest możliwa, lecz niezmiernie rzadko wykonywana w praktyce).

Najważniejsza obserwacja polega na tym, że jeżeli dwie niezagnieżdżone, niepubliczne klasy, zdefiniowane w dwu oddzielnych jednostkach kompilacji zawierają odniesienia do siebie nawzajem, to nie jest możliwe przeniesienie każdej z tych jednostek kompilacji z osobna, natomiast możliwe jest przemieszczenie obu naraz. Oznacza to, że przeniesienie więcej niż jednej jednostki kompilacji nie jest prostym złożeniem kilku przeniesień – po jednym pliku z osobna.

Warunki wstępne 1–5 oraz 10–12 są dla tej refaktoryzacji analogiczne do warunków dla przypadku jednej jednostki kompilacji. By sprawdzić warunki 6–9 możemy potraktować wszystkie typy znajdujące się we wszystkich jednostkach z rozważanego zbioru jako znajdujące się w jednej jednostce kompilacji (pomijając narzucony przez specyfikację języka warunek istnienia co najwyżej jednego publicznego, niezagnieżdżonego typu w jednostce kompilacji).

Bibliografia

[1] D. Bäumer, E. Gamma, A. Kieżun, *Integrating Refactoring Support into a Java Development Tool.* Ukaże się w OOPSLA'2001 Companion

[2] K. Beck, *Extreme Programming Explained.* Addison–Wesley, 1999.

[3] R. W. Bowdidge, *Supporting the Restructuring of Data Abstractions through Manipulation of a Program Visualisation.* Ph.D. Thesis, University of California, San Diego, 1995.

[4] E. Cassais, *Automatic Reorganisation of Object–Oriented Hierarchies: a Case Study.* Object Oriented Systems, Grudzień 1994.

[5] M. Ó Cinneide, *Automated Application of Desing Patterns: a Refactoring Approach.* Ph.D. Thesis, Univeristy of Dublin, Trinity College, 2000.

[6] M. Fowler, *Refactoring: Improving the Design of Existing Programs.* Addison–Wesley, 1999.

[7] E. Gamma, R. Helm, R. Johnson, J. Vlissedes, *Design Patterns: Elements of Reusable Object–Oriented Software.* Addison–Wesley, 1995.

[8] J. Gosling, B. Joy, G. Steele, G. Bracha, *The Java Language Specification. Second Edition.* Addison–Wesley, 2000.

[9] *Java Native Interface Specification. Second Edition.*

[10] R. E. Johnson, B. Foote, *Designing reusable classes.* Journal of Object–Oriented Programming, czerwiec/lipiec 1988.

[11] W. F. Korman, *Elbereth: Tool Support for Refactoring Java Programs.* M.Sc. Thesis, University of California, San Diego, 1998.

[12] I. Moore, *Guru: a Tool for Automatic Restructuring of Self Inheritance Hierarchies.* TOOLS USA, Prentice–Hall, 1995.

[13] I. Moore, *Automatic Inheritance Hierarchy Restructuring and Method Refactoring.* OOPSLA'96, ACM, 1996.

[14] J. Norda, *A Refactoring Tool for Java.* M.Sc. Thesis, University of Linköping, 2001.

[15] W. Opdyke, R. Johnson, *Refactoring: An Aid in Designing Application Frameworks and Evolving Object–Oriented Systems.* Proceedings of the Symposium on Object–Oriented Programming Emphasising Practical Applications, Nowy Jork, 1990.

[16] W. Opdyke, R. Johnson, *Creating Abstract Superclasses by Refactoring.*

[17] W. Opdyke, *Refactoring Object–Oriented Frameworks.* Ph.D. Thesis, University of Illinois, 1992.

[18] W. Opdyke, R. Johnson, *Refactoring and Aggregation.* Proceedings of the JSSST International Symposium on Object Technologies for Advanced Software, Kanazawa, 1993.

[19] D. B. Roberts, *Practical Analysis for Refactoring.* Ph.D. Thesis, 1999.

[20] Ch. Seguin, *Refactoring Tool Challenges in a Strongly Typed Language.* OOPSLA'00 Companion, pp. 101–102.

[21] S. Tichelaar, S. Ducasse, S. Demeyer, O. Niestrasz, *A Meta–model for Language–Independent Refactoring.* Proceedings of ISPSE 2000.

[22] L. A. Tokuda, *Evolving Object–Oriented Designs with Refactorings.* Ph.D. Thesis, University of Texas at Austin, 1999.

[23] L. A. Tokuda, D. Batory, *Evolving Object–Oriented Designs with Refactorings.* Proceedings of the 14th IEEE International Conference on Software Engineering, 1999.

[24] Strona projektu Eclipse *httlp://www.eclipse.org*

[25] Strona programu IntelliJ, *http://www.intellij.com/.*

[26] Strona programu jFactor, *http://www.instantiations.com/jfactor/.*

[27] Strona programu JRefactory, *http://users.snip.net/aseguin/csrefactory.html.*

[28] Strona programu The Smalltalk Refactoring Browser, *http://st–www.cs.uiuc.edu/~brant/Refactory/.*

[29] Strona programu Transmogrify, *http://transmogrify.sourceforge.net/.*

[20] Strona programu XRef, *http://www.www.ref–tech.com/speller/.*