Zaawansowane metody refaktoryzacji kodu Nulls Removal[1]

Krzysztof Kaczmarski

Wydział Matematyki i Nauk Informacyjnych, Politechnika Warszawska

kaczmars@mini.pw.edu.pl

Streszczenie

Refaktoryzacja (ang. refactoring) jako metoda nieustannego poprawiania i ulepszania kodu zdobył uznanie programistów obiektowych na całym świecie. Niezwykle pomocny jest tu katalog i dokładny opis możliwych podstawowych klas zmian wykonany przez Martina Fowlera Opierając się głównie na doświadczeniu proponuje on, by odnajdywać w kodzie kłopotliwe miejsca, a następnie je udoskonalać. Wśród kłopotliwych i niebezpiecznych miejsc aplikacji nie wymienia jednak wartości pustej (null), choć jest ona często przyczyną poważnych błędów. W poniższym tekście przedstawiono problemy wartości pustej w programowaniu oraz kilka możliwych przekształceń kodu, które mogą pomóc w ich usunięciu.

1. Wstęp

1.1. Błędy podczas uruchamiania programów

Każdy programista, podczas uruchomienie swojego programu wielokrotnie doświadczył błędów, które wynikały z odwoływania się do błędnego wskaźnika, niezainicjowanej zmiennej, dzielenia przez zero itd. Jest to spora klasa błędów, która nie może być łatwo wykryta podczas kompilacji programu, a jedynie podczas jego działania. Wyłapywanie ich jest dość kłopotliwe i najczęściej sprowadza się do wielokrotnego uruchamiania procedur testujących i rozbudowanego testowania systemowego.

Błędy tego rodzaju pojawiają się pomimo istnienia dobrze rozwiniętej kontroli typów (choć rzadko dopracowanej w komercyjnych językach), która ma zagwarantować poprawność stosowanych konstrukcji. Dynamiczne błędy pojawiają się w najbardziej zaskakujących momentach. Dużą grupą tych błędów (obok błędnych działań arytmetycznych) są niewłaściwe działania na zmiennych wskaźnikowych i referencjach do obiektów, a źródłem ich są najczęściej:

- luki w obiektowym systemie typów (problem kowariancji, metod binarnych lub konieczności częstego rzutowania w dół hierarchii dziedziczenia, niewłaściwe korzystanie z mechanizmu refleksji)

[1] Praca zrealizowana w ramach projektu badawczego nr 503G 1120 0026 002

- użycie zmiennych niezainicjowanych (brak wspomagania ze strony kompilatora wykrycia ich niebezpiecznego użycia)
- wprowadzone przez użytkownika błędne odwołania oraz nieprzemyślane tworzenie i niszczenie obiektów (brak panowania nad dynamicznie przydzielaną pamięcią)
- efekty uboczne działania skomplikowanych i nieznanych przez programistę procedur

Część błędów można wyeliminować stosując rozbudowane narzędzia i zaawansowane języki programowania. Jednak ciągle kluczową rolę odgrywa czynnik ludzki, bo to właśnie programista i projektant oraz to, że niezupełnie rozumieją tworzone przez siebie konstrukcje w kontekście całości sytemu, jest częstym źródłem usterek [BROO1987].

Zaistniała sytuacja powoduje, że rozwijane są liczne kontrolery kodu i narzędzia wspomagające wnioskowanie o poprawności procedur. Stosują one pewne metody kontroli przepływu sterowania, badania inicjowania zmiennych oraz osiągalności obiektów. Pomimo jednak intensywnych prac badawczych ich użycie w przemyśle jest ciągle sporadyczne (głównie ze względu na ciągle ograniczone możliwości).

W niniejszy tekst skupia się na eliminowaniu tych niebezpieczeństw podczas uruchamiania programu, które wynikają z odwoływania się do zmiennych o wartości pustej *null*[2]. Przykłady odnoszą się do języka obiektowego, w którym wartości puste są przypisywane referencjom, jednak można je uogólnić na wszelkiego rodzaju zmienne występujące w języku programowania, o ile tylko można im przypisać wartość pustą. Sens unikania wartości pustej jest głębszy i nie ogranicza się tylko do jednego języka i jednego rodzaju zmiennych.

1.2. Refaktoryzacja w doskonaleniu kodu programu

Lekkie metodyki projektowe, takie jak np. *Extreme Programming* [KENT1999] podkreślają dążenie do jak najlepszego kodu przez jego nieustanne poprawianie. Stosowane w nich metody, zgrupowane pod nazwą *refaktoryzacji (ang. refactoring)* mają właśnie na celu ulepszanie kodu, jego czytelności i niezawodności.

Środowisko obiektowe postrzega refaktoryzację przede wszystkim jako usystematyzowanie tych metod. W tej pracy opisane zostanie zastosowanie refaktoryzacji do modyfikowania kodu programu w taki sposób, by stał się mniej podatny na błędy związane z występowaniem w nim wartości pustych. Przedstawione propozycje opisują dobrze wyizolowane strategie działań, które może podejmować programista w celu podniesienia jakości swojego programu. Całość usystematyzowanych działań jest propozycją nowego schematu postępowania nazwanym *Nulls Removal*[3]. Opierając się na licznych doświadczeniach wielu projektów, jak również obserwując projekty studenckie można stwierdzić, że w pewnych sytuacjach użycie wartości null jest miejscem niebezpiecznym w programie i podobnie, jak i inne wymienione przez Fowlera *podejrzane miejsca* powinno podlegać przeróbce.

Proponowana nowa metoda refaktoryzacji może być stosowana w większości impe-

[2] W pracy stosowana jest angielska nazwa null określająca wartość pustą, inne spotykane określenia to nil, undefined, unknown

[3] Angielskie nazwy w schematach mają służyć stworzeniu systematyki podobnej do tej, którą zaproponował Fowler, a która jest powszechnie przyjęta w środowisku.

ratyw-nych języków programowania posiadających wskaźniki. W szczególności metoda została z sukcesem przetestowana dla języka Java, który może być uznany za przedstawiciela klasycznego języka zorientowanego obiektowo. Nie ma znaczenia, czy wykorzystuje się wskaźniki, czy referencje. Różnica między nimi polega zwykle na tym, że wskaźniki mogą pokazywać na dowolne miejsce pamięci bez względu na to, czy znajduje się tam sensowny obiekt oraz czy jest on odpowiedniego typu. Oczywiście takie wskaźniki z losowymi wartościami są potencjalnie tak samo niebezpieczne jak referencje z wartościami null. Powinny podlegać szczególnie wnikliwej inspekcji, po której przypisywanie im wartości przypadkowych, lub nie przypisywanie żadnej wartości powinno zostać wyeliminowane. Jest to działanie analogiczne do jednego z przedstawionych w dalszej części schematów refaktoryzacji.

2. Problem wartości pustej

Problem wartości pustych był wielokrotnie podnoszony w literaturze (np. [DATE1995, SUBI1996]). Wprowadzenie takiej konstrukcji do języka powoduje bowiem automatycznie powstanie wielu niespójności i dodatkowych komplikacji. Tak na przykład w Javie istnieje wartość null, którą można przypisać do referencji, ale nie ma analogicznej wartości dla innych typów. Dla zmiennej typu integer przewiduje się zero jako wartość domyślną, która jednak nie może spełniać roli wartości pustej. W języku SQL wprowadzenie wartości null do relacji doprowadziło do powstania wielu często nieprzewidywalnych zachowań [DATE1986] oraz utrudniło znacznie semantykę zapytań. Jednym z problemów jest fakt, że działanie tego samego zapytania może się gwałtownie zmienić w wyniku tylko tego, że raz operuje na wartościach pewnych (niepustych), a innym razem przetwarza kombinacje wartości null. Problem ten wielokrotnie podnosił Date w swoich opracowaniach. Wprowadzenie wartości pustej próbowano też włączyć w logikę trój-wartościową. Date jednak odrzuca tą koncepcję jako zupełnie nie przystającą do świata rzeczywistego i ludzkiej intuicji [DATE1992a].

Istnienie wartości pustej w typizowanym (w szczególności w silnie typizowanym) języku programowania doprowadza do jeszcze jednej bardzo nieprzyjemnej własności. Pomimo, że dana konstrukcja w programie jest prawidłowo użyta, i błędy nie zostały zidentyfikowane, to nie ma gwarancji, że zostanie wykonana poprawnie. Nie ma bowiem pewności, czy któraś ze zmiennych nie ma wartości pustej i w związku z tym, czy jakiekolwiek odwołania do niej mają sens. Szczególnie dotkliwie jest to widoczne w najbardziej popularnym przypadku referencji do obiektów. W podanym poniżej przykładzie program zgłosi błąd wykonania pomimo zaakceptowania przez system kontroli typów.

```
Point p = new Point(15,21);
...
if (...)
    p = null;
else
    ...
p.move( 10, 30 ); // NullPointerException
```

Powyższy problem polega oczywiście na tym, że poprawność ostatniej linii zależy od ścieżki wykonania. Aby wykryć błąd tego typu konieczne jest zastosowanie złożonych walidatorów kodu wykonujących analizę przepływu sterowania (*ang. control-flow*). Analiza taka może być niezwykle skomplikowana ze względu na możliwość powstania bardzo wielu ścieżek w zależności od instrukcji warunkowych, potencjalne istnienie odwołań z innych wątków i zmiennych globalnych.

Analogiczne błędy są zmorą programistów i nieustającą przyczyną komunikatów typu *Null pointer exception, Core dump, Kernel panic, Violation error* i wielu innych. We wszystkich tych przypadkach nawet najszczelniejszy system typów okazał się bezradny. Nasuwa się pytanie czy jest sens wprowadzać do języka silną kontrolę typów i jednocześnie zezwalać na istnienie zmiennych z wartościami pustymi. W językach funkcyjnych jak np. ML wyeliminowano wartość pustą na rzecz wartości opcjonalnych, które trzeba specjalnie deklarować i które są dzięki temu specjalnie traktowane [MILN1990]. Każde użycie takiej zmiennej jest szczególnie kontrolowane przez kompilator i programista jest ciągle powiadamiany o niebezpieczeństwie.

Niektóre języki prezentują odmienne podejście. Zakładają, one że wszelkie operacje na wartościach pustych dają w wyniku wartości puste lub są ignorowane. Oczywiście przeciwnicy takiego podejścia zwracają uwagę, że może to komplikować zrozumienie i działanie programu oraz być przyczyną wielu nieprzewidywalnych (błędnych) zachowań. Przeciwnym podejściem jest generowanie wyjątku przy każdym odwołaniu do wartości nil, tak jak w SmallTalku. To znów zmusza nieustająco do dodatkowego testowania czy dana wartość nie jest pusta. Częściowym rozwiązaniem jest obiekt typu *message eater*, który będzie formalnie istniał (zlikwiduje powstawanie wyjątku) ale praktycznie nic nie będzie robił [PRAT1995]. Jest on analogiczny do prezentowanego niżej obiektu pustego.

Języki obiektowe typu Java i C++ nie wprowadziły żadnego szczególnego wspomagania kontroli wartości pustych. Pewnym problemem jest tu fakt istnienia zmiennych lokalnych, które od momentu deklaracji muszą mieć jakąś wartość. Automatyczne ostrzeganie o niebezpiecznym użyciu może być niezwykle skomplikowane.

Programiści często chcąc ustrzec się ostrzeżenia o istnieniu zmiennej o niezainicjowanej wartości odruchowo przypisują jej wartość pustą do oznaczenia, że nie ma jeszcze żadnej sensownej wartości do wpisania, ale najprawdopodobniej będzie ona znana w późniejszym czasie. Oczywiście nie rozwiązuje to problemu. Podczas wykonania może to doprowadzić bowiem do dokładnie takiego samego błędu, jak odwołanie się do zmiennej o losowej wartości.

Chcąc uniknąć wyżej wymienionych błędów można zmodyfikować klasyczne języki obiektowe, tak by zbliżyły się do np. ML (co jest zresztą tematem niektórych prac np. [ABIT1997]), lub zmienić zwyczaje programistów i wpoić im bardziej odpowiedzialne traktowanie wartości null.

To pierwsze rozwiązanie, polegające na modyfikacjach języka wymaga tworzenia nowych kompilatorów lub prekompilatorów, opartych na nowej semantyce, a często i syntaktyce języka. Może być przez to uciążliwe w stosowaniu, mało stabilne, i powodować, że aplikacja staje się trudna do modyfikacji. Z tych powodów zastosowania prze-

mysłowe opierają się ciągle na pragmatycznym podejściu określonym przez ustalone i gwarantowane standardy, a wiele akademickich rozszerzeń znanych języków przeminęło bez większego echa.

Zamiast więc tworzyć kolejne rozszerzenia języka, które mogłoby poprawić niezawodność programu przez przynajmniej częściowe opanowanie problemu wartości pustej, należy położyć nacisk na właściwe standardy kodowania, które postarają się w miarę możliwości wspomóc nawet mało doświadczonego programistę w walce z tym trudnym wrogiem. Metody powszechnie znane jako refaktoryzacja kodu powinny zostać rozszerzone o takie, które szczególnie zwracają uwagę na powody użycia wartości pustej, i ewentualne możliwości jej wyeliminowania.

W następnej części, po przybliżeniu podstaw refaktoryzacji, przedstawiono szablon działań pod wspólną nazwą *nulls removal*. Mają one za zadanie zwrócenie uwagi programistów na świadome stosowanie wartości pustej oraz proponują możliwe rozwiązania prowadzące do jej usunięcia. Oczywiście ze względu na charakter komercyjnych języków programowania typu Java, C++, czy C# nie jest możliwe całkowite zrezygnowanie ze stosowania w nich wartości null we wszystkich przypadkach. Jednak unikanie jej w miarę możliwości może znacznie poprawić niezawodność aplikacji, szczególnie w dużych zespołach programistycznych, gdzie wiele osób współdzieli części kodu lub wykorzystuje wiele obcych bibliotek. W takiej sytuacji zapobieganie anomaliom idącym za wartościami pustymi ma niebagatelne znaczenie.

3. Refaktoryzacja

Termin refaktoryzacja istnieje od wielu lat i oznacza przerabianie istniejącego kodu programu. Sam problem modyfikacji kodu był tematem wielu prac badawczych [OPD1992]. Ostatnio zaś zyskał ogromną popularność po systematycznym i przystępnym zdefiniowaniu przez Fowlera [FOWL1999]. Określił go jako podnoszenie jakości i czytelności istniejącego kodu programu, bez zmiany jego funkcjonalności. Dosłownie pisze on, że zawsze gdy kod jest podejrzany, nieczytelny, gdy dorabiamy coś nowego, to zawsze trzeba go poprawić – przerobić. Fowler zaproponował wiele scenariuszy możliwych zmian (razem jest ich ponad siedemdziesiąt), które można podzielić na kilka kategorii:

- podnoszące czytelność kodu
- podnoszące jakość projektu
- ułatwiające zrozumienie działania

Podstawą działania podczas przerabiania istniejącego kodu jest systematyczne działanie, które absolutnie nie może być przypadkowe, ale ma być objęte całościowymi założeniami projektu. To właśnie założenia projektowe i standardy kodowania określają jaka postać kodu jest pożądana, a jakich konstrukcji należy unikać. Zasady te powinny zostać zebrane w postaci standardu, który doskonale znają zaawansowani programiści, a który jest przewodnikiem dla początkujących. Znajomość *podejrzanych konstrukcji* pozwala na tworzenie programów o wiele lepszych; łatwiejszych w utrzymaniu, czytelniejszych, mniejszych, szybszych oraz pomaga w wykrywaniu błędów.

Fowler w swojej książce wymienia dość długą listę konstrukcji, które uważa za niezbyt eleganckie, zaciemniające intencje programisty, niebezpieczne i błędogenne. Takie miejsca powinny być przede wszystkim obiektem przeróbek. Konieczność przerobienia projektu może być też wykazana przez pewne metryki obiektowe jak np. długość metod, wysokość i szerokość drzewa dziedziczenia oraz liczba podklas.

Wprowadzanie zmian do działającego kodu jest zawsze niebezpieczne. Dlatego należy bezwzględnie stosować przemyślaną strategię działania, która powinna składać się z następujących kroków:

- zlokalizowanie problemu
- określenie celu przeróbki i znalezienie odpowiedniego wzorca przekształcenia
- wykonanie przeróbki
- wykonanie testów regresyjnych

Zastosowanie konkretnego wzorca przekształcenia ma wyeliminować możliwość wprowadzenia błędów, które mogłyby powstać z niezrozumienia istoty przeróbki oraz z niedokładnej znajomości semantyki przerabianego kodu. Dodatkowo po każdorazowej przeróbce powinny być uruchomione testy, które wykażą, że program zachowuje się dokładnie tak samo, jak poprzednio. Istnieją prace opisujące dokonywanie zmian w sposób możliwie automatyczny, tak by uniknąć wprowadzania błędów. Osiągnięto to przez formalne określenie warunków wstępnych które muszą być spełnione, aby można było daną modyfikację wykonać [OPD1992]. Przerabianie można podzielić na klasy względem złożoności. Proste działania (jak wprowadzenie nowej metody) łatwo dają się weryfikować, podczas gdy te bardziej złożone (jak tworzenie nowej nadklasy, czy nowej podklasy) wymagają znacznej interwencji i kontroli przez człowieka. Wydaje się, że rozbudowane testy są ciągle podstawowym narzędziem, które może wspomóc programistę w zapewnieniu neutralności dokonanej zmiany [JUNI2003]. Często przywoływane w takich sytuacjach metody formalne wymagają bowiem znajomości zbyt skomplikowanych zagadnień i w obecnej formie nie są masowo stosowane w przemyśle.

Aktualnie istnieje ogromna ilość opisanych i usystematyzowanych schematów refaktoryzacji [FOWL1999, REF2003, KERI2002]. Opisują one wiele możliwych przekształceń kodu starając się zagwarantować, że program nie tylko nie zmieni swojego zachowania, lecz zostanie znacznie ulepszony. Rozeznanie w proponowanych przez środowisko obiektowe modyfikacjach kodu i zdolność lokalizacji *podejrzanych miejsc* jest cechą najlepszych programistów, podobnie jak znajomość wzorców projektowych (*ang. Design Patterns*) [GOF1995]. Tak, jak w przypadku wzorców, każdy z nich ma dokładnie przemyślaną motywację i konkretne zyski wynikające z zastosowania.

Aktualnie narzędzia typu CASE oraz przeróżne zintegrowane środowiska programistyczne zaczynają wspierać refaktoryzację. Nawet w przypadku skomplikowanych zmian są w stanie wspomóc programistę pewnymi automatycznymi działaniami. Może to być np. automatyczne przeniesienie metody do innej klasy lub jej podział na kilka różnych klas. Automatyzm działania polega najczęściej na tym, że potrafią one odszukać w kodzie odwołania do modyfikowanej klasy i również w miarę potrzeby je poprawić.

3.1. Usuwanie wartości null

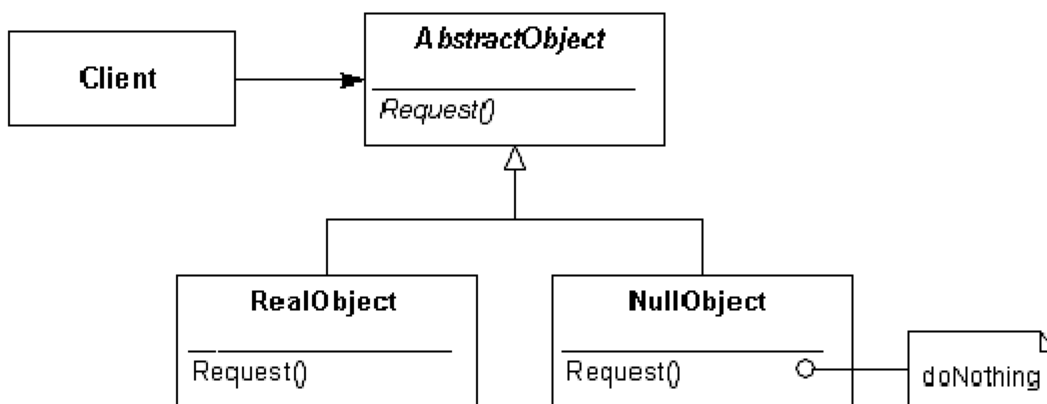
W katalogu przekształceń zaproponowanych przez Fowlera znalazł się wzorzec o nazwie *Wprowadzenie obiektu pustego (ang. Introduce null object).* Poniżej skrótowo przedstawiona jest jego propozycja, która opiera się na wykorzystaniu wzorca projektowego z klasy wzorców strukturalnych, o nazwie *The Null Object Pattern*, a opisanego przez Woolfa [WOOL1998]. Motywacją jest wyeliminowanie powtarzających się testów wartości null. Działanie specyficzne dla wartości pustej staje się częścią nowego obiektu. Wprowadzony wyspecjalizowany obiekt pusty może jednak doprowadzić do eksplozji klas i utrudnić przerabianie kodu. Zyskiem natomiast jest to, że obiekty wykorzystujące nową strukturę mogą zignorować różnice pomiędzy obiektem implementującym pewne zachowanie i obiektem nie robiącym nic.

Ogólnie znany jest przykład przeszukiwania drzewa binarnego przed i po zastosowaniu obiektu pustego. Przykład ten nie tylko usuwa wartość null, ale i upraszcza cały algorytm. Idea ta jest oparta na pomyśle stworzenia specyficznej klasy strażnika, który oznaczać będzie koniec przeszukiwania drzewa. Schematyczna modyfikacja hierarchii klas przedstawiona jest na rysunku 1. Dzięki pełnemu zastosowaniu obiektowości (polimorfizm, abstrakcja i enkapsulacja), specyficzne zachowanie strażnika jest ukryte w jego wnętrzu, a obiekty z niego korzystające nie muszą go odróżniać od pozostałych obiektów. Jest to ogromny zysk w stosunku do potrzeby ciągłego testowania poprawności referencji przy pomocy instrukcji warunkowych.

```
public class BinaryNode {
  Node left = null;
  Node right = null;
  int key;

  public boolean includes(int value)
  {
    if (key == value)
      return true;
    else if ( (value < key) &&
                (left == null) )
      return false;
    else if (value < key)
      return left.includes(value);
    else if (right == null)
      return false;
    else
      return right.includes(value);
  }
}
```

```
public class BinaryNode extends Node {
  Node left = new NullNode();
  Node right = new NullNode();
  int key;

  public boolean includes(int value)
  {
    if (key == value)
      return true;
    else if (value < key )
      return left.includes(value);
    else
      return right.includes(value);
  }
}
public class NullNode extends Node {
  public boolean includes( int value )
  {
    return false;
  }
}
```

Oczywiście zastosowanie obiektu pustego ma głębszy sens jeśli użyje się go łącznie ze wzorcem *Singleton* (patrz [GOF1995]), który zapewni, by taki obiekt powstał w systemie tylko jeden. Wszystkie bowiem obiekty puste są najczęściej dokładnie takie same i nie posiadają żadnych stanów. Dzięki temu nie ma potrzeby, by je niepotrzebnie mnożyć.



Rys. 1. Wzorzec nowej hierarchii klas po wprowadzeniu obiektu pustego w wariancie proponowanym przez Woolfa. [WOOL1998]

Główną motywacją wprowadzenia obiektu pustego jest w literaturze uproszczenie algorytmu i zamknięcie specyficznego zachowania w oddzielnej klasie. Zwróćmy jednak uwagę na to, że w tym samym momencie pozbywamy się niebezpiecznych wartości null. Nie grozi nam odwołanie się do nieistniejącego obiektu, gdyż formalnie referencja zawsze pokazuje na jakiś obiekt. Niestety nie zawsze taka strategia daje się zastosować.

Woolf podkreśla również, że wprowadzenie obiektu pustego może być trudne w przypadku, gdy różni klienci nie zgadzają się co do tego, jak powinno wyglądać *puste zachowanie*. W takiej sytuacji może dojść do dodatkowego mnożenia klas lub konieczności wprowadzenia dodatkowych, dynamicznie dołączanych delegatów, implementujących *Null* w różny sposób, w zależności od konkretnej potrzeby [WOOL1998].

W następnej części podano zaawansowane możliwości usuwania wartości pustej występującej również w innych przypadkach.

4. Zaawansowane schematy z klasy *Nulls removal*

W tej części opisano zaawansowane możliwości zastosowania refaktoryzacji do unikania problemów wynikających z zastosowania zmiennych bez wartości. Będą to więc przekształcenia z klasy prowadzących do osiągnięcia mniejszej podatności na błędy.

Proponowany schemat ma na celu podniesienie jakości programu przez usuniecie zarówno wartości null jak i potencjalnych możliwości jej powstania. Nowe techniki są próbą usystematyzowania problemów z wartościami pustymi i precyzyjnego określenia możliwości ich rozwiązania.

W dalszej części tekstu skupiono się na wartości null i powstających problemach w zmiennych typu wskaźnikowego jako jednym z najczęstszych źródeł błędów w procedurach. W większości bowiem komercyjnych języków zorientowanych obiektowo właśnie typ wskaźnikowy (czy też referencyjny) mając wartość pustą sprawia najwięcej nieoczekiwanych kłopotów. Zwykle programista stosuje w imperatywnej procedurze wartość null aby:

- przypisać jakąkolwiek wartość do nowopowstałej zmiennej
- zaznaczyć, że dana zmienna nie jest już więcej potrzebna i poinformować system

o możliwości zwolnienia pamięci

- zaznaczyć, że został osiągnięty zły stan
- jako wynik jakiejś operacji, która nie daje rezultatów

Przedstawione poniżej schematy postępowania będą mogły być zastosowane przy wszystkich z wymienionych punktów, choć prawdopodobnie w niektórych przypadkach może się to wiązać ze znacznym przerobieniem programu. Zmiany takie mogą być zakwalifikowane jako *głębokie.*

Oddzielnym problemem jest usunięcie wartości null z pól obiektów. Puste własności obiektów są bowiem związane ze specyfiką danej struktury danych. Często okazuje się, że dana własność z jakiegoś powodu nie może mieć przypisanej wartości. Środowiska bazodanowe wręcz definiują kilka rodzajów wartości pustej ze względu na przyczynę. Mogą to być *Unknown, Undefined, Irrelevant, Missing*, itd. Nie ma zgody co do takiej, lub podobnej klasyfikacji, a rozwiązanie takie znacznie komplikuje semantykę danych.

Zwróćmy uwagę na obiekt reprezentujący studenta, który w którymś momencie życia przestaje być studentem i staje się tylko zwykłą osobą. Część jego własności może zostać ustawiona na null. Można postulować, by projektant modelu danych dążył do ich lepszego dopasowania i przedstawił taki model danych, który będzie lepiej oddawał rzeczywistość. Wtedy należałoby zmienić obiekt student w obiekt osoba. Jednak taka operacja nie zawsze może być wykonana dynamicznie. Cały problem w prostej drodze prowadzi do koncepcji *dynamicznych ról*. Są one opisywane w literaturze od dawna choć nie zostały szeroko zaimplementowane przez języki komercyjne [ALBA1993, FOWL1998, JODŁ2002]. Również koncepcja wartości opcjonalnych i danych półstrukturalnych nawiązuje do lepszego dopasowania modelu [ABIT1997, XMLS2001].

4.1. Remove null from initialization

Pierwszy schemat ma na celu zrezygnowanie ze stosowania wartości null do inicjowania zmiennych. Wskaźniki w miarę możliwości powinny być natychmiast inicjowane odpowiednią wartością. Oczywiście nie zawsze jest to wykonalne, więc programista powinien wykazać się szczególną wytrwałością i inwencją w przerabianiu kodu, tak by maksymalnie zrezygnować z zastosowania wartości pustych.

4.1.1. Schemat postępowania

```
Point r = new Point();                    Point r = new Point();
Point p = null;                           ...
...                          ⇒            r.move(10,14);
r.move(10,14);                            ...
...                                       Point p = new Point(r);
p = new Point(r);
```

4.1.2. Analiza

Celem jest wyeliminowanie zmiennej o nieustalonej wartości z początkowym przypi-

saniem wartości pustej. Taka referencja posiadająca wartość null istnieje w przestrzeni nazw od momentu zadeklarowania i może być również przypadkowo, bądź nieświadomie użyta przed przypisaniem jej poprawnej wartości.

Modyfikacja ma za zadanie wyeliminowanie deklaracji zmiennej przed bezpośrednim użyciem, które to wymusza, by przez pewien czas działania programu taka zmienna miała wartość domyślą (najczęściej losową lub null). Programista powinien dążyć do deklarowania zmiennych w miejscach, które bezpośrednio poprzedzają ich użycie. W wyniku tego działania czas życia zmiennych i ich istnienia w przestrzeni nazw jest znacznie ograniczony, przez co i zmniejsza się szansa na niewłaściwe ich użycie. Przez fakt opóźnionego zadeklarowania zmiennej dopiero w momencie potrzeby, programista włącza się bardziej świadomie w ograniczenia przestrzeni nazw. Dzięki temu również kompilator może zwrócić większą uwagę na użycie danej zmiennej i wprowadzić dodatkowe optymalizacje.

Aby zlokalizować miejsca, gdzie można zastosować ten schemat wystarczy zwrócić uwagę, gdzie przy deklaracji zmiennych stosuje się słowo kluczowe null. Jeśli jest to możliwe, to powinno się tak przerobić program by go uniknąć.

Oddzielnym, choć podobnym schematem postępowania jest usunięcie wartości pustej z inicjowania pól obiektów (*Remove null from field initialization*). Tu jednak pojawiają się dodatkowe problemy wynikające z tego, że często pole obiektu może być zainicjowane wyłącznie po wykonaniu szeregu operacji. Zwykle więc nie ma żadnej sensownej wartości, którą można by mu przypisać od samego początku jego istnienia. Oczywiście zawsze można zastosować wspomniany wcześniej pusty obiekt (*NullObject*), w połączeniu z szablonem *Singleton* lub *Flyweight* jako tymczasowy substytut. Wydaje się jednak, że aż nazbyt często byłoby to rozwiązanie sztuczne. Powinno się raczej dążyć do tego, by po zakończeniu pracy konstruktora obiektu wszystkie pola zostały poprawnie ustawione. Schemat taki mógłby się nazywać *Assure full object creation*.

4.2. Remove Null from memory freeing

W niektórych językach programowania (jak np. w Javie) programista przypisuje wartość pustą do referencji by poinformować odśmiecacz pamięci (*ang. Garbage Collector – GC*) o potencjalnej możliwości zwolnienia pamięci przydzielonej na wskazywany przez nią obiekt. Problemem jest jednak fakt, że sama referencja, choć już na nic nie wskazuje, istnieje nadal i może być błędnie użyta w innym miejscu.

4.2.1. Schemat postępowania

```
Point p = new Point();                  {
...(1)...                                  Point p = new Point();
p = null;                   ⇒                 ...(1)...
...(2)...                               }
                                        ...(2)...
```

4.2.2. Analiza

Programista zamiast przypisywać wartość pustą do zmiennej, doprowadza do jej natychmiastowego usunięcia przez odpowiednie operowanie na przestrzeni nazw. Osiąga przez to dodatkowe zyski. Pamięć może być zwalniana natychmiast, a nie na końcu procedury lub innego bloku. Ponadto poza możliwością zwolnienia obiektu system może dodatkowo zwolnić pamięć która zajmowana jest przez samą referencję. Otwiera się w ten sposób drogę do nowych automatycznych optymalizacji na poziomie kompilacji i zmniejsza się zapotrzebowanie na zasoby.

Deklarację zmiennej wraz z kodem z niej korzystającym umieszcza się w oddzielnym segmencie programu, który jednocześnie ogranicza przestrzeń deklarowanych nazw. Instrukcje wymagające obecności zmiennej `p` (1) umieszcza się w tym samym zakresie. Pozostałe instrukcje (2) poza zakresem.

Aby zlokalizować miejsce poprawki należy zwrócić uwagę na przypisania wartości null, po których dana zmienna nie jest już używana. Niektóre kompilatory potrafią wykonywać taką optymalizacją automatycznie. Nie zadziałają jednak poprawnie jeśli programista przez pomyłkę użył obiektu już po jego zwolnieniu. Dzięki zamknięciu bloku programista sam odcina sobie drogę do popełnienia takiego błędu.

4.3. Remove null as a wrong state

W tym schemacie dążymy do zarzucenia stosowania wartości null jako oznaczenia złego stanu programu. Zamiast tego powinny być stosowane wyjątki i inne przewidziane w danym języku programowania konstrukcje (*triggery*, *flagi, przerwania, blokady i inne*).

4.3.1. Schemat postępowania A

```
Point p = new Point();
...(1)...
if (some_wrong_state)
   p = null;
else
   ...(2)...
...(3)...
if (p==null)
   ...(4)...
else
   ...(5)...
...(6)...
```

⇒

```
Point p = new Point();
...(1)...
try {
  if (some_wrong_state)
    throw (new WrongStateException())
  ...(2)...
  ...(3)...
  ...(5)...
}
catch( WrongStateException e ){
  ...(3)...
  ...(4)...
}
finally{
  ...(6)...
}
```

4.3.2. Analiza

Najprostsza konwersja przedstawionego przykładu tak, by wykorzystywał wyjątek nie jest niestety optymalna. Charakter zgłaszania wyjątków jest taki, że przerywana jest kompletnie bieżąca ścieżka wykonania, a sterowanie przekazywane jest natychmiast do bloku obsługującego sytuację wyjątkową. Oznacza to, że część programu, która jest wykonywana zarówno w sytuacji gdy nastąpił, jak i nie nastąpił szczególny przypadek (3), jest trudna do przekształcenia. Aby wykonać identyczne działanie przy pomocy wyjątków konieczne jest umieszczenie bloku (3) zarówno w części `try` jak i `catch`. Części tej nie można umieścić w bloku `finally` gdyż wtedy zostanie zmieniona pierwotna kolejność poleceń. Blok (4) lub (5) zostanie wykonany przed blokiem (3).

Ten schemat działania powinien więc być stosowany jedynie w szczególnych i uzasadnionych przypadkach. Może też zdarzyć się, że w niektórych sytuacjach blok (3) nie istnieje lub może być umieszczony w innym miejscu, co rozwiązuje problem.

4.3.3. Schemat postępowania B

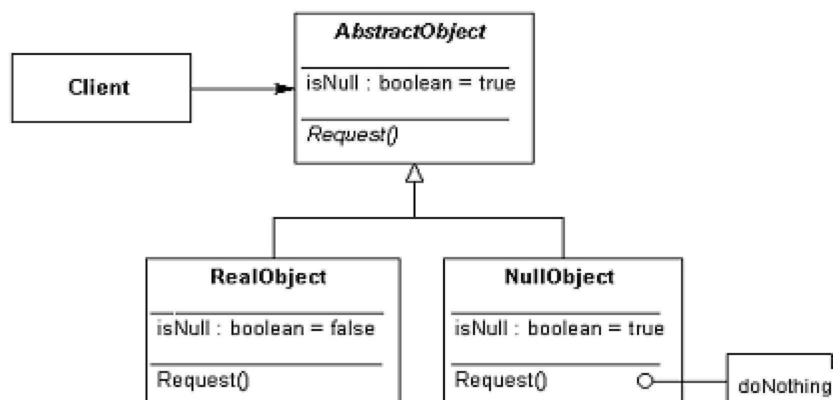
```
Point p = new Point();
...(1)...

if (some_wrong_state)
   p = null;
else
   ...(2)...

...(3)...
if (p==null)
   ...(4)...
else
   ...(5)...
...(6)...
```

⇒

```
Point p = new Point();
...(1)...
if (some_wrong_state)
   p = new NullPoint();
else
   ...(2)...
...(3)...
if (p instanceOf NullPoint)
   ...(4)...
else
   ...(5)...
...(6)...
```

4.3.4. Analiza

Jest to rozwiązanie analogiczne do zaprezentowanego przez Fowlera, które zostało przedstawione w poprzedniej części. Różnica polega na tym, że tutaj stosujemy obiekt pusty nie tylko jako zaślepkę dla pewnej szczególnej funkcjonalności, ale również jako oznaczenie zamiast wartości null, z zamysłem odróżnienia go od normalnego stanu.



...

Rys. 2. Modyfikacja obiektu pustego ułatwiająca jego identyfikowanie w stosunku do prawdziwych obiektów. Odpowiednik komunikatu isNull w SmallTalku

W niektórych językach programowania może nie istnieć operator `instanceOf`, lub jego użycie może być bardzo nieekonomiczne. Można wtedy nieco zmodyfikować schemat klas aby dodać cechę pozwalającą odróżniać obiekty puste od pozostałych (rysunek 2). Stosujemy wtedy sprawdzenie:

```
if ( p.isNull )
    ...
```

Innym możliwym rozwiązaniem jest wprowadzenie obiektu pustego jako statycznej klasy wewnętrznej dzięki czemu unika się bezpośredniego modyfikowania klasy `p`. Prowadzi to do innego rodzaju porównania:

```
if ( p == p.NullPoint )
    ...
```

Aby zlokalizować miejsca, gdzie można wykonać tą refaktoryzację należy zwrócić uwagę na przypisywanie wartości null i późniejsze testowanie, czy dana zmienna posiada taką wartość. Jest to więc sytuacja bardzo podobna do podanej przez Fowlera dlatego też i jego rozwiązanie z pustym obiektem lub jego wariacjami daje się dobrze zastosować.

4.4. Remove Null as a return value

Analogicznie do punktu poprzedniego procedury nie powinny zwracać wartości pustej jako oznaczenie jakiegoś stanu lecz powinny zamiast tego operować na wyjątkach. Procedura zwracająca wartość null, a mająca jednocześnie zadeklarowaną jakąś konkretną wartość, która powinna być zwrócona, jest czymś nieuzasadnionym i niekonsekwentnym. Zwróćmy uwagę, że w takim przypadku zupełnie traci sens deklarowanie typu zwracanej wartości. Wartość pusta może bowiem zostać dynamicznie dopasowana do każdego typu. (Obecnie jednak większość języków, przy dynamicznym wiązaniu, nie korzysta z typu zwracanej wartości.) Wartość null najczęściej może być zwrócona przez procedurę w szczególnym przypadku, który drastycznie odbiega od normalnego. W takiej jednak sytuacji bardziej zasadne wydaje się użycie odpowiednich wyjątków.

4.4.1. Schemat postępowania

```
Point p move(int x, int y)          Point p move(int x, int y)
{                                        throws WrongStateException
  ...(1)...                         {
  if (some_wrong_state)      ⇒        ...(1)...
    return null;                     if (some_wrong_state)
  ...(2)...                             throw new WrongStateException();
}                                       ...(2)...
                                     }
```

4.4.2. Analiza

Zastosowanie tego przekształcenia daje wiele korzyści, które są od dawna znane programistom używającym wyjątków. Procedura może teraz o wiele pełniej informować o różnych zaistniałych problemach. Zwracana wartość null mogła jedynie informować o jednym typie sytuacji wyjątkowej. Dzięki wyjątkom można dla każdej specjalnej sytuacji zgłosić do miejsca wywołania inny rodzaj komunikatu. W przypadku Javy daje to jeszcze dodatkowe zalety, gdyż kompilator natychmiast poinformuje programistę o możliwych sytuacjach wyjątkowych i wymusi na nim stworzenie odpowiedniej reakcji w miejscu wywołania danej procedury.

Drobna komplikacja powstaje, gdy w danym języku programowania nie ma wyjątków, bądź ich użycie jest utrudnione albo niemożliwe z innych względów. W takiej sytuacji można zastosować schemat analogiczny do wprowadzenia obiektu pustego. Trzeba zdefiniować w hierarchii klas nową klasę, która będzie reprezentowała zwracaną wartość w sytuacji wyjątkowej. Można w niej przy pomocy różnorodnych mechanizmów oznaczyć rodzaj problemu. Klasa taka powinna mieć tak dobraną funkcjonalność, by umożliwić rozpoznanie problemu przez klienta, który daną procedurę wywołał.

5. Podsumowanie

Techniki objęte terminem refaktoryzacji są jeszcze jedną odpowiedzią na problemy jakości w inżynierii oprogramowania. Zostały przyjęte i zaakceptowane nie tylko przez środowisko zwolenników lekkich metodyk ale również zdobyły znaczące uznanie w szerokich kręgach programistów obiektowych.

Zaprezentowane nowe techniki refaktoryzacji mają na celu wspomożenie programisty przy usuwaniu z programów miejsc niebezpiecznych związanych z występowaniem wartości pustej. Proponowane nowe schematy uzupełniają zbiór znanych przypadków, w których opłaca się stosować refaktoryzację. Spodziewane zyski to przede wszystkim wzrost niezawodności aplikacji i bardziej świadome korzystanie przez programistów z wartości pustych.

Programiści powinni mieć na uwadze, że wszelkie zmiany działającego kodu powinny być przeprowadzane z niezwykłą ostrożnością. Usystematyzowanie schematów zmian powinno w tym wydatnie pomóc. Katalogowanie schematów i badanie skutków ich zastosowania oraz określanie warunków wstępnych ma ułatwić opracowanie odpowiednich narzędzi i wzbogacenie zintegrowanych środowisk wspomagających automatyczną i półautomatyczną refaktoryzację.

Bibliografia

[ABIT1997] S. Abiteboul, S. Quass, S. McHugh, S. Widom i S. Wiener, *The Lorel query language for semistructured data*, International Journal

on Digital Libraries, pp. 68-88, 1, 1, 1997.

[ALBA1993] A. Albano, A. Bergamini, A. Ghelli i A. Orsini, *An Object Data Model with Roles*, 9th International Conference on Very Large Data Bases, Dublin, Ireland, Proceedings, 1993.

[BROO1987] F. P. Brooks, *No silver bullet – essence and accidents of software engineering*, IEEE Computer, pp 10-19, April, 1987.

[DATE1986] C. J. Date, *Null Values in Database Management*, Relational Database: Selected Writings, Addison-Wesley, 1986.

[DATE1992a] C. J. Date i C. J. Darwen, *Oh No Not Nulls Again*, Relational Database Writings 1989-1991, Addison-Wesley, 1992.

[DATE1995] C. J. Date, *Missing Information Problem*, An Introduction to Database Systems, Addison-Wesley, 1995.

[FOWL1998] M. Fowler, *Dealing with Roles*, Unpublished, 1998, http://www2.awl.com/cseng/titles/0-201-89542-0/roles2-1.html.

[FOWL1999] M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[GOF1995] E. Gamma, E. Helm, E. Johnson i E. Vlissides, *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley, 1995.

[JODŁ2002] A. Jodłowski, A. Habela, A. Płodzień i A. Subieta, *Objects and Roles in the Stack-Based Approach*, Database and Expert Systems Applications, 13th International Conference, DEXA 2002, September 2002.

[JUNI2003] Kent Beck i Kent Gamma, *JUnit Testing Framework*, http://www.junit.org.

[KERI2002] J. Kerievsky, *Refactoring to Patters*, draft, http://industriallogic.com/xp/refactoring/.

[KENT1999] K. Beck, *Extreme Programming Explained: Embrace Chang*, Addison-Wesley Pub Co; 1st edition October 5, 1999.

[MEIJ2001] E. Meijer i E. Perry, *Scripting .NET using Mondrian*, In Proc. ECOOP'01.

[MILN1990] R. Milner, R. Tofte i R. Harper, *The Definition of Standard ML*, MIT Press, Cambridge, 1990.

[OPD1992] W. F. Opdyke, *Refactoring Object-Oriented Frameworks*, Rozprawa Doktorska, University of Illinois at Urbana-Champain, 1992.

[PRAT1995] N. Pratt, *A Generalized Null Object Pattern, Pattern Languages of Program Design*, Redakcja. Coplien J. and Schmidt D., Addison-Wesley, 1995, http://www.awl.com/cseng/titles/0-201-60734-4.

[REF2003] Martin Fowler, *www.refactoring.com*.

[SUBI1996] K. Subieta, K. Kambayashi, K. Leszczylowski i K. Ulidowski, *Null*

Values in Object Bases: Pulling out the Head from the Sand, October 1996, http://citeseer.nj.nec.com/subieta96null.html.

[WOOL1998] B. Woolf, *The Null Object Pattern, Pattern Languages of Program Design 3*, Redakcja. Martin R. C., Riehle D, Buschmann F., Addison-Wesley, 1998, http://www.cs.wustl.edu/~schmidt/PLoP-96/woolf1.ps.gz.

[XMLS2001] W3C, *XML Schema Structures and Datatypes*, 2001, http://www.w3.org/TR/.