Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Radomir Małaczek
Nr albumu: 181276

Analiza statyczna kodu. Integracja z Platformą Eclipse

**Praca magisterska
na kierunku INFORMATYKA**

Praca wykonana pod kierunkiem
dr Janiny Mincer-Daszkiewicz
Instytut Informatyki

Czerwiec 2005

Oświadczenie kierującego pracą

Oświadczam, że niniejsza praca została przygotowana pod moim kierunkiem i stwierdzam, że spełnia ona warunki do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data | Podpis kierującego pracą

Oświadczenie autora (autorów) pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązujacymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data | Podpis autora (autorów) pracy

Streszczenie

W ramach pracy powstał moduł analizy statycznej kodu dla środowiska programistycznego Eclipse. W pracy przedstawiona została architektura rozbudowywalnej platformy ze szczególnym uwzględnieniem możliwości integrowania jej z nowymi komponentami. Znaczna część pracy została poświęcona metodom statycznej analizy kodu oraz typowym błędom programistycznym, jakie można wykryć przy ich użyciu.

Słowa kluczowe

Eclipse, statyczna analiza kodu, mechanizm wtyczek, JLint

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

D. Software
D.2. SOFTWARE ENGINEERING
D.2.4. Software/Program Verification
D.2.6. Programming Environments

Spis treści

1. Wprowadzenie . . . 7

2. Eclipse . . . 9
- 2.1. Platforma Eclipse . . . 11
  - 2.1.1. Środowisko wykonawcze Eclipse . . . 11
  - 2.1.2. Przestrzeń robocza . . . 12
  - 2.1.3. Pulpit . . . 13
- 2.2. Tworzenie rozszerzeń . . . 14

3. Reguły analizy statycznej . . . 17
- 3.1. Wstęp . . . 17
- 3.2. JLint . . . 18
- 3.3. Problemy z analizą statyczną kodu . . . 18
  - 3.3.1. Brak alarmu . . . 18
  - 3.3.2. Fałszywe alarmy . . . 19
- 3.4. Kategorie reguł analizy statycznej . . . 20
- 3.5. Sprawdzanie kontraktów metod . . . 21
  - 3.5.1. Metoda `finalize()` . . . 21
  - 3.5.2. Klonowanie obiektów . . . 21
  - 3.5.3. Metoda `equals(Object)` w Javie . . . 25
  - 3.5.4. Serializacja obiektów . . . 29
  - 3.5.5. Biblioteka rejestrująca zdarzenia Log4J . . . 29
- 3.6. Złe praktyki programistyczne . . . 29
  - 3.6.1. Wywoływanie przez konstruktor metod wirtualnych tego samego obiektu 29
  - 3.6.2. Wyjątki . . . 30
  - 3.6.3. Przepływ sterowania z wykorzystaniem wyjątków . . . 32
  - 3.6.4. Dwukrotnie sprawdzana blokada . . . 33
  - 3.6.5. Pobranie obiektu klasy przez stworzenie nowej instancji . . . 35
  - 3.6.6. Używanie += na obiektach `String` w pętli . . . 35
  - 3.6.7. Niepotrzebne tworzenie obiektów String . . . 36
  - 3.6.8. Używanie konstruktora Boolean . . . 37
  - 3.6.9. Tworzenie obiektów opakowujących . . . 38
  - 3.6.10. Puste bloki (szczególnie blok catch) . . . 38
- 3.7. Unikanie typowych błędów programistycznych . . . 38
  - 3.7.1. Nieużywany kod . . . 38
  - 3.7.2. Losowanie liczb . . . 39
  - 3.7.3. Serializacja singletonów . . . 40
  - 3.7.4. Porównanie napisów bez uwzględniania wielkości liter . . . 40

3.7.5. Metoda `Boolean.getBoolean()` 41
3.7.6. Słowo kluczowe `return` w bloku `finally` 41
3.8. Styl kodu źródłowego 42
3.9. Programowanie rozproszone 43

4. Projekt i implementacja modułu 45
4.1. Cel i kontekst 45
4.2. Założenia wstępne 45
4.3. Wymagania 46
4.4. Istniejące punkty rozszerzań 47
4.4.1. org.eclipse.ui.views 47
4.4.2. org.eclipse.core.resources.markers 47
4.4.3. org.eclipse.ui.ide.markerResolution 48
4.4.4. org.eclipse.ui.preferencePages 49
4.4.5. org.eclipse.core.resources.builders 49
4.4.6. org.eclipse.core.resources.natures 50
4.4.7. org.eclipse.ui.popupMenus 50
4.5. Nowe punkty rozszerzeń 51
4.5.1. Reguły audytu 51
4.5.2. Procedury usuwania naruszeń reguł 52
4.5.3. Widoki kodu źródłowego 52
4.6. Widoki 52
4.7. Reguły 53
4.8. Interfejs użytkownika 54

5. Podsumowanie 61

A. Zawartość płyty CD 63
A.1. Hierarchia katalogów wraz z opisem zawartości 63
A.2. Instalacja modułu statycznej analizy kodu 63

Bibliografia 65

Spis rysunków

2.1. Moduły Projektu Eclipse (zaczerpnięte z [3]) 10

4.1. Kod źródłowy reguły `EqualsParamsRule` 55
4.2. Menu kontekstowe z opcją uaktywnienia analizy kodu 56
4.3. Lista mechanizmów przyrostowego budowania projektu 57
4.4. Lista zainstalowanych reguł 57
4.5. Ustawienia reguł 58
4.6. Widok edytora z zaznaczonymi naruszeniami reguł analizy 59
4.7. Edytor z zaznaczonymi miejscami naruszeń reguł JLint 60

Rozdział 1

Wprowadzenie

Podstawowym kryterium oceny tworzonego oprogramowania jest jego niezawodność, stąd tak duży nacisk kładzie się na proces projektowania i testy. Koszt usuwania błędów popełnionych na etapie projektowania bądź programowania logiki aplikacji jest niewspółmiernie większy niż np. koszt usunięcia błędów w interfejsie użytkownika. Ponieważ informatyka stała się nieodzownym elementem naszego życia, które często zależy od jakości oprogramowania, nie możemy pozwolić sobie na poważne błędy.

Doskonały przykład zdarzeń mobilizujących informatyków do tworzenia coraz to doskonalszych narzędzi wspomagających ich pracę stanowi testowy start rakiety Ariane 5 mający miejsce 4 czerwca 1996 roku. Mimo ogromu środków włożonych w realizację projektu (7 miliardów euro) oraz poświęconych 10 lat pracy grona wysoko wykwalifikowanych specjalistów niecałe 40 sekund po starcie rakieta uległa samodestrukcji.

Co było przyczyną niepowodzenia wydawałoby się profesjonalnie prowadzonego projektu?

Niepowodzenie zostało spowodowane błędem w oprogramowaniu. Kod dla Ariane 5 został przeniesiony z oprogramowania tworzonego dla Ariane 4. Problemem okazało się być przepełnienie podczas konwersji 64-bitowej liczby zmiennoprzecinkowej na 16-bitową liczbę całkowitą za znakiem.

Naturalnym następstwem tego typu zdarzeń jest dążenie do wykrycia przyczyny poniesienia ogromnych strat oraz zapobiegnięcia podobnym wypadkom w przyszłości. Katastrofa ta zainicjowała badania nad modernizacją sposobu tworzenia i testowania oprogramowania na etapie kodowania, nad automatyzacją procesu analizy kodu. Kod Ariane był pierwszym dużym systemem przeanalizowanym narzędziami do analizy statycznej kodu. W informatyce, jak w każdej dziedzinie, w której efekty prac są zależne od wiedzy, dokładności i koncentracji człowieka, dąży się do automatyzacji czynności mogących powodować powtarzalne błędy lub uchybienia. Celem tworzenia narzędzi wspierających pracę programistów jest między innymi wykluczenie prawdopodobieństwa popełniania błędów jakie zostały już zidentyfikowane. Czerpanie wiedzy z doświadczenia stanowi podstawy do rozwoju każdej dziedziny nauki, również informatyki. Statyczna analiza jest właśnie jedną z technologii zapewniających, a przynajmniej zmniejszających możliwość wystąpienia znanego, powtarzalnego błędu w oprogramowaniu.

Podstawowym narzędziem pracy programisty jest edytor kodu źródłowego. Tym samym stanowi on doskonałą podstawę do stworzenia środowiska wspomagającego nie tylko proces tworzenia programów poprzez automatyczne uzupełnianie kodu (ang. *code completion*), ale również wsparcie dla początkowej fazy testowania poprzez nie dopuszczanie do powstawania błędów jakie mogą zostać wykryte za pomocą statycznej analizy kodu.

W swojej pracy przedstwię koncepcję stworzenia narzędzia do statycznej analizy kodu

stanowiącego rozszerzenie dla środowiska programistycznego Eclipse. W pierwszej części pracy opisany został edytor Eclipse, jego architektura, możliwości rozbudowy o nowe moduły (por. rozdz. 2). W drugiej części przedstawiam zagadnienia statycznej analizy kodu, potrzebę jej przeprowadzania oraz wybrane reguły (por. rozdz. 3). Założenia projektowe, opis zaimplementowanego modułu, sposób jego użycia oraz możliwości dalszego rozwoju zostały przedstawione w rozdz. 4.

W pracy prezentuję korzyści płynące z wykorzystywania narzędzi wspomagających audyt kodu źródłowego. Wykazuję, iż dzięki zastosowaniu odpowiedniego oprogramowania możliwe jest unikanie popularnych błędów programistycznych i że dzięki środowiskom o otwartej architekturze możliwe jest samodzielne tworzenie tego typu narzędzi.

Rozdział 2

Eclipse

W niniejszym rozdziale omawiam podstawowe pojęcia związane z Platformą Eclipse. Przedstawiam także założenia jakie przyświecały twórcom Eclipse oraz rozwiązania architektoniczne, które pozwoliły na ich realizację.

Projekt Eclipse obejmuje kilka powiązanych ze sobą elementów. W ramach projektu możemy wyróżnić następujące podprojekty:

Platforma Eclipse

Platforma Eclipse (ang. *Eclipse Platform*) jest zbiorem modułów, które pozwalają na tworzenie programów użytkowych, które działają w zintegrowanym środowisku programistycznym. Na Platformę Eclipse składają się następujące elementy, które zostaną bardziej szczegółowo opisane w dalszej części rozdziału:

- jednolity dla całej platformy mechanizm rozszerzania funkcjonalności, zwany dalej mechanizmem wtyczek (ang. *plugin*),
- interfejs użytkownika imitujący pulpit,
- przenośny zestaw kontrolek interfejsu użytkownika,
- model danych opisujący projekty oraz zasoby w nich gromadzone,
- mechanizm zarządzania zmianami zasobów, który może zostać wykorzystany w kompilatorach przyrostowych.
- mechanizm integracji z odpluskwiaczem (ang. *debugger*),
- mechanizm integracji z systemem zarządzania zmianami,

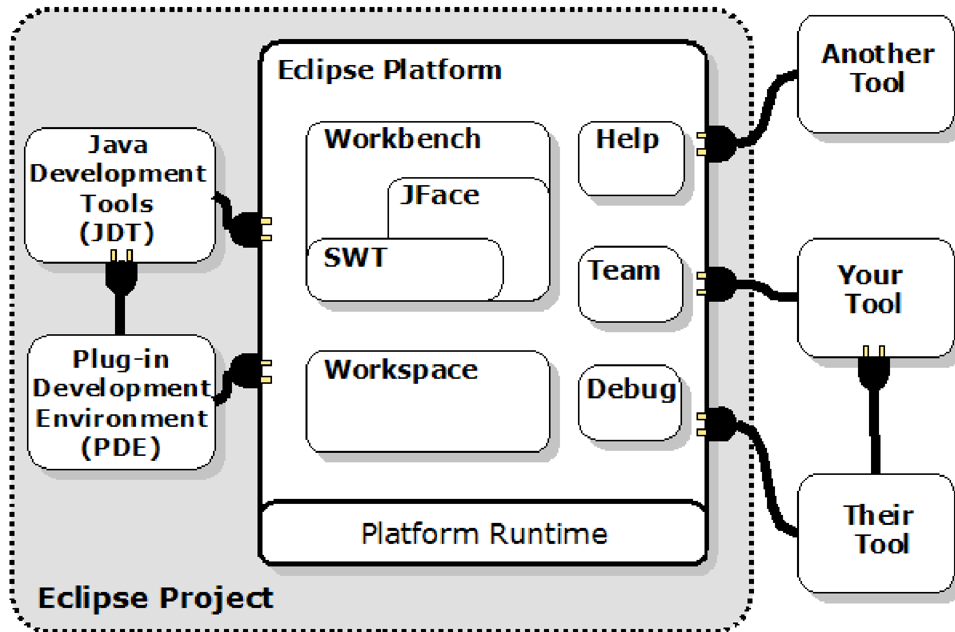
JDT

JDT (Java Development Tools) to zestaw programów użytkowych integrujących się z Platformą Eclipse dostarczający środowisko do programowania aplikacji w języku Java. Aplikacje te korzystają z mechanizmu rozszerzania funkcjonalności dostępnego w ramach Platformy.

Zestaw rozszerzeń stanowiących JDT pozwala także na wykorzystanie ich mechanizmów podczas pisania narzędzi operujących na kodzie źródłowym Javy. Dla programistów dostępny jest interfejs programistyczny pozwalający na operowanie na zasobach projektu Javy takich jak: pakiety, biblioteki, jednostki kompilacji (odpowiadające plikom źródłowym Javy). Dostarczony wraz ze środowiskiem wbudowany parser plików źródłowych Javy pozwala na przeglądanie i modyfikację ich drzew AST, natomiast wbudowany kompilator pozwala na wyszukiwanie w projekcie konkretnych klas, do których odnosi się dany fragment kodu źródłowego.

PDE

PDE (Plugin Development Environment) to zestaw widoków i edytorów, które ułatwiają tworzenie i testowanie własnych rozszerzeń dla Eclipse. PDE umożliwia edycję *plików deklaracji* (ang. *manifest file*) pozwalając na edycję dowolnych parametrów w wygodny sposób. PDE zawiera także odpluskwiacz wtyczek Eclipse.



Rysunek 2.1: Moduły Projektu Eclipse (zaczerpnięte z [3])

Zależności pomiędzy poszczególnymi elementami projektu dobrze obrazuje rysunek 2.1 zaczerpnięty z [3].

Cała funkcjonalność środowiska podzielona jest na niewielkie moduły, nazywane również wtyczkami. Moduł taki zawiera kod wykonywalny w Javie, dodatkowe dowolne zasoby (np. grafikę) oraz plik deklaracji opisujący w jaki sposób moduł rozszerza platformę. Plik deklaracji (zwany również deskryptorem) zawiera następujące informacje:

- podstawowe informacje o module: nazwa, wersja, opis;
- listę plików zawierających kod wykonywalny;
- listę bibliotek niezbędnych do prawidłowego działania modułu;
- opis zależności od innych modułów.

Oprócz tych informacji plik deklaracji zawiera opis interfejsu, jaki mogą implementować inne moduły, by rozszerzyć jego funkcjonalność. W tym samym pliku definiujemy także jakie interfejsy, udostępnione przez inne moduły, implementujemy.

Na mechanizmie definiowania interfejsów, zwanych tu *punktami rozszerzeń* (ang. *extension point*) oraz implementowania ich w innych wtyczkach opiera się modułowa budowa środowiska Eclipse. Platforma Eclipse składa się z ponad 80 wtyczek, definiujących około 85 punktów rozszerzeń. Łączna liczba ich implementacji sięga 160. Mechanizm ten jest na tyle sprawny,

że budowane są na nim takie produkty jak IBM WebSphere Application Development, w którym sama liczba wtyczek sięga 600.

Rozszerzenia z Projektu Eclipse pozwalają na wykorzystanie tylko wybranych elementów platformy, przez co łatwo możemy otrzymać środowisko gotowe do rozbudowy o dodatkową funkcjonalność bez niepotrzebnego bagażu niewykorzystywanej funkcjonalności. Programista aplikacji, który chce skorzystać z rozwiązań Projektu Eclipse może stworzyć zarówno narzędzia pozbawione graficznego interfejsu użytkownika, jak i narzędzia z interfejsem graficznym. Choć Eclipse dostarcza własne rozwiązania dotyczące interfejsu użytkownika (takie jak gotowe biblioteki komponentów interfejsu użytkownika), to można wykorzystać dowolną bibliotekę GUI.

2.1. Platforma Eclipse

Środowisko wykonawcze, przestrzeń robocza, pulpit, biblioteki SWT i JFace oraz mechanizm widoków, edytorów i perspektyw tworzą razem rozwiązanie nazywane Platformą Eclipse, które stanowi bazę do rozwoju narzędzi programistycznych. Oprócz tego platforma Eclipse zawiera także następujące elementy:

- interfejs do tworzenia systemów pomocy,
- wsparcie dla narzędzia ANT (*http://jakarta.apache.org/ant*),
- wsparcie dla narzędzia do testów jednostkowych oprogramowania (JUnit),
- wsparcie dla integracji systemów zarządzania wersjami oprogramowania wraz z implementacją dla systemu CVS,
- architekturę do tworzenia środowisk do śledzenia (ang. *debugging*) oprogramowania,
- mechanizm automatycznej instalacji nowych wersji oprogramowania (mechanizm Software Update).

2.1.1. Środowisko wykonawcze Eclipse

Środowisko wykonawcze (ang. *Platform Runtime*) jest podstawowym elementem Eclipse. Jest to ta część platformy, która odpowiada za:

- wyszukiwanie wtyczek przy starcie aplikacji,
- aktywację wtyczek (czyli ładowanie klas),
- ustalanie zależności między wtyczkami oraz odpowiednie ładowanie wtyczek zależnych,
- zarządzanie informacjami o wtyczkach oraz punktach rozszerzeń jakie dana wtyczka implementuje oraz definiuje.

Środowisko wykonawcze definiuje tylko jeden punkt rozszerzeń. Implementacje tego punktu rozszerzenia nazywane są *aplikacjami*. Mogą one wykorzystywać mechanizm ładowania wtyczek jako metodę na rozszerzenie własnej funkcjonalności. W ramach Platformy Eclipse zdefiniowana jest jedna aplikacja będąca zintegrowanym środowiskiem programistycznym. Zwykle nie ma potrzeby definiowanie nowej aplikacji. Definiujemy ją tylko wtedy, gdy nie chcemy wykorzystywać żadnych pozostałych elementów platformy (tj. interfejsu użytkownika, zarządzania zasobami).

Architektura mechanizmu zarządzającego wtyczkami wymaga, aby zbiór modułów jakie mogą być załadowane był znany przy starcie platformy. Zbiór ten nie może już być rozszerzony po uruchomieniu pierwszej wtyczki. Jeśli chcemy dodać nowy moduł do już uruchomionej platformy, to musimy zakończyć pracę i uruchomić ją jeszcze raz.

Informacje o wszystkich wtyczkach przechowywane są w tzw. rejestrze wtyczek, który zawiera wszelkie informacje z deskryptora wtyczki. Zebranie tych informacji nie pociąga za sobą aktywacji wtyczek. Aktywacja jest procesem, podczas którego ładowane są klasy zadeklarowane w deskryptorze wtyczki, co pociąga za sobą większe zapotrzebowanie aplikacji na pamięć. Należy tak projektować punkty rozszerzeń, aby opóźnić moment aktywacji wtyczki do momentu, gdy jest to już absolutnie konieczne. Przykładowo: informacje o nowym elemencie menu (czyli wyświetlanym tekście czy skrócie klawiszowym) powinny być zawarte w deskryptorze wtyczki, załadowania odpowiedniej klasy będzie wymagało dopiero kliknięcie na danym elemencie menu. Jeśli tekst, który powinien być wyświetlony w menu, wymagałby załadowania klasy Javy z wtyczki, to każda z wtyczek, która definiuje element menu musiałaby zostać aktywowana już w momencie wyświetlenia menu, nie zaś wyboru konkretnej opcji.

Dzięki możliwości opóźnienia momentu aktywowania wtyczki platforma może zawierać bardzo dużą liczbę wtyczek, co nie wpływa ujemnie na jej wydajność – ładowane są tylko te elementy platformy, które rzeczywiście są używane, zaś jej użytkownik cały czas ma dostęp do pełnej funkcjonalności.

W obecnej architekturze Eclipse nie jest możliwe dezaktywowanie raz aktywowanej wtyczki, a usunięcie wtyczki z pamięci jest możliwe tylko przy zamykaniu platformy.

2.1.2. Przestrzeń robocza

Jednym z podstawowych elementów platformy jest *przestrzeń robocza* (ang. *Workspace*). Definiuje ona strukturę zasobów – są one pogrupowane w projekty. Każdy projekt może zawierać pliki oraz katalogi z systemu plików.

Przestrzeń robocza definiuje wiele elementów, które operują na zasobach, takich jak:

- *Wcielenia projektu* (ang. *nature*). Jest to dodatkowa informacja o samym projekcie. W zależności od dostępnych wcieleń można uaktywnić specyficzną funkcjonalność. Przykładowo, projekt zawierający wcielenie projektu Javy będzie zawierał we wszelkich menu komendy specyficzne dla Javy (np. kompilowanie zasobów projektu).
- Historia modyfikacji zasobów. Przestrzeń robocza zawiera informacje o poprzednich wersjach zasobu (np. pliku).
- *Znaczniki* (ang. *markers*). Znaczniki są powiązane z zasobem i zawierają dodatkowe informacje o nim, takie jak np. miejsca błędów kompilacji, zakładki zdefiniowane przez użytkownika, rezultaty wyszukiwania, pułapki. Przestrzeń robocza jest odpowiedzialna za zarządzanie cyklem życia znaczników: zachowuje ich stan we własnym rejestrze oraz śledzi wszelkie zmiany w zasobach, tak aby automatycznie uaktualnić zawartość znaczników (np. skasowanie wiersza spowoduje usunięcie znaczników zdefiniowanych w danym wierszu).
- Wsparcie do przyrostowego budowania projektów. Przestrzeń robocza pozwala na stworzenie narzędzia do budowania projektu (ang. *builder*), które będzie otrzymywało jako dane wejściowe informacje o tym jakie zasoby zmieniły swoją zawartość od ostatniego uruchomienia procesu budowania. Narzędzie to otrzyma nie tylko informację o tym jakie zasoby zmieniły swoją zawartość, ale nawet o tym jakie zmiany zostały przeprowadzone na danych zasobach.

2.1.3. Pulpit

Pulpit (ang. *Workbench*) jest tą częścią platformy, która odpowiada za interfejs użytkownika. Implementacja opiera się na dwóch bibliotekach interfejsu użytkownika: SWT oraz JFace. Pulpit wprowadza także pojęcia edytorów, widoków oraz perspektyw.

SWT

SWT jest biblioteką kontrolek interfejsu użytkownika stworzoną w ramach Platformy Eclipse. Wraz z biblioteką standardową Javy dostarczane są dwie inne biblioteki do programowania graficznych interfejsów użytkownika: AWT oraz Swing. SWT różni się jednak znacznie od dwóch wymienionych.

SWT definiuje interfejs API niezależny od systemu operacyjnego, jednak implementacja tego interfejsu wykorzystuje mechanizmy specyficzne dla danego systemu operacyjnego bądź biblioteki interfejsu użytkownika, na której ta implementacja jest oparta. Dzięki takiemu rozwiązaniu wygląd aplikacji pisanej w Javie z użyciem biblioteki SWT nie odbiega od wyglądu innych aplikacji. SWT udostępnia także mechanizmy specyficzne dla konkretnych platform w postaci interfejsu w Javie, np. wersja biblioteki dla systemów z rodziny Windows pozwala na interakcję z obiektami ActiveX.

Platformę Eclipse zbudowano na bazie SWT. Twórcy Eclipse doszli do wniosku, że stworzenie wieloplatformowego zbioru elementów interfejsu użytkownika jest trudnym zadaniem – nie tylko pod względem dostosowania do możliwości obecnych systemów operacyjnych, ale także pod względem podobieństwa (w zachowaniu i wyglądzie) do elementów interfejsu użytkownika udostępnianych przez system operacyjny.

SWT jest czymś pośrednim pomiędzy bibliotekami AWT oraz Swing. Z jednej strony AWT zawiera duże bloki kodu macierzystego (ang. *native code*) danej platformy, z drugiej w bibliotece Swing kod macierzysty nie jest praktycznie wcale używany. SWT opakowuje zaś pojedyncze odwołania do systemu operacyjnego w odpowiednie procedury. Skorzystanie z Java Native Interface (JNI) pozwala na używanie elementów interfejsu użytkownika specyficznych dla danej platformy.

Takie podejście pozwala ograniczyć ilość kodu macierzystego wywoływanego z kodu Javy, co znacznie upraszcza wyszukiwanie i usuwanie błędów w kodzie biblioteki. W SWT nie potrzeba także mechanizmu PLAF (ang. *Pluggable Look and Feel*), który pozwala na dostosowanie wyglądu interfejsu użytkownika do danej platformy, ponieważ ta sama aplikacja uruchomiona pod kontrolą różnych systemów operacyjnych, na każdym z nich będzie korzystać z odpowiedniego dla danego systemu zestawu elementów interfejsu użytkownika. Innym plusem biblioteki SWT jest integracja z systemem operacyjnym zaawansowanych mechanizmów, takich jak np. funkcja przeciągnij i upuść (ang. *drag and drop*).

Przy projektowaniu międzyplatformowych interfejsów użytkownika pojawia się problem tzw. najmniejszego wspólnego mianownika wszystkich obsługiwanych platform. W SWT elementy interfejsu użytkownika, które nie są dostępne na danej platformie, są emulowane przy pomocy pozostałych elementów lub są rysowane na kanwie, a ich zachowanie jest odpowiednio emulowane. Przykładem może być element drzewa – obecny w systemach Windows, zaś nieistniejący w bibliotece interfejsu graficznego Motif. Implementacja SWT dla biblioteki Motif zawiera kod Javy, który dostarcza funkcjonalność elementu drzewa. Implementacja tej biblioteki dla platformy Windows wywołuje po prostu odpowiednie komendy GDI (ang. *Graphics Device Interface*).

JFace

Biblioteka ta oferuje dodatkowe kontrolki interfejsu użytkownika. JFace zbudowane jest na bazie SWT, jednak w odróżnieniu od SWT implementacja biblioteki jest niezależna od niskopoziomowej biblioteki interfejsu użytkownika. JFace upraszcza wiele standardowych zadań programisty graficznych interfejsów użytkownika takich jak:

- tworzenie okien dialogowych, kreatorów (ang. *wizards*),
- zarządzanie czcionkami, rysunkami (ogólnie zasobami interfejsu użytkownika),
- operowanie na tekście.

Biblioteka udostępnia także szkielet (ang. *framework*) do opakowania API SWT, tak, aby zmiany w modelu danej kontrolki powodowały zmiany komponentów interfejsu użytkownika (podobnie jak np. w architekturze MVC). Biblioteka ta nie zawiera kodu zależnego od platformy (kodu macierzystego).

Widoki, edytory, perspektywy

Pulpit platformy Eclipse przedstawia okno aplikacji zawierające wiele ramek, nazywanych tu *widokami* (ang. *views*) oraz jedną wyróżnioną ramkę – edytor. Widoki mogą być przeróżnie rozmieszczone na ekranie w zależności od wybranej *perspektywy* (ang. *perspective*). Do każdej z perspektyw przypisane jest pewne rozmieszczenie widoków, które może być następnie zmodyfikowane przez użytkownika. Perspektywa ma tak naprawdę za zadanie ograniczyć liczbę i rodzaj wyświetlanych widoków do podzbioru związanego z daną czynnością, scenariuszem użycia edytora. W standardowej instalacji Eclipse znajdziemy takie perspektywy jak Debug, CVS Repository Exploring, Java czy Plug-in Development.

Eclipse różnym typom zasobów przyporządkowuje różne edytory. Pliki tekstowe będą prezentowane przy użyciu prostego edytora, zaś pliki z kodem źródłowym Javy wyświetlą się w edytorze, który potrafi np. rozpoznać i podświetlić odpowiednie słowa kluczowe.

W niniejszej pracy stworzony został nowy widok, który udostępnia informacje o zainstalowanych i aktywnych regułach analizy kodu. Rozszerzona została także funkcjonalność istniejącego widoku *Problemy* tak, aby informował one o ewentualnych błędach analizy statycznej w kodzie źródłowym w taki sam sposób jak informuje o błędach występujących przy kompilacji plików źródłowych Javy.

2.2. Tworzenie rozszerzeń

Jakich zasad należy się trzymać przy tworzeniu wtyczek do Eclipse? W książce [6] autorzy przedstawili kilka zasad dotyczących architektury wtyczek oraz tworzenia nowych miejsc, w których może następnie być rozszerzana funkcjonalność środowiska.

Po pierwsze: *wszystkie elementy platformy należy traktować jako rozszerzenie dotychczas istniejącej funkcjonalności.* Oczywiście skutkuje to tym, że otrzymujemy system składający się z dużej liczby wtyczek. Przy takiej liczbie rozszerzeń twórcy Eclipse musieli zaprojektować architekturę tak, aby środowisko uruchamiało się szybko niezależnie od liczby rozszerzeń. To rodzi problem – informacje o nowej funkcjonalności zawarte w skompilowanych plikach Javy będą wymagały załadowania tych klas do pamięci, co może trwać nawet kilka sekund. Po pomnożeniu tej wartości przez liczbę wtyczek otrzymamy dość długi czas ładowania, a także dużą zajętość pamięci.

Jeśli chcemy, aby środowisko uruchamiało się szybko, to nie możemy ładować od razu do pamięci wszystkich klas. Jeśli podzielimy każde rozszerzenie na dwie części – deklarację oraz implementację – to rozszerzenia będą mogły być ładowane leniwie. Przy uruchomieniu platformy ładowane są same deklaracje, zaś ich kod wykonywalny zawarty w skompilowanych plikach `*.class` ładowany będzie dopiero wtedy, gdy będzie taka konieczność (np. użytkownik otworzy plik takiego typu, przy edycji którego powinno być aktywne dane rozszerzenie). Proces załadowania klas nazywany jest *aktywacją rozszerzenia*.

Deklaracja wtyczki to plik XML o ustalonej strukturze. Informacje z wszystkich plików deklaracji zostaną załadowane do rejestru wtyczek przy starcie platformy. Dane z rejestru wykorzystywane są do dynamicznej aktywacji wtyczek. Oto przykładowy plik deklaracji:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <plugin id="org.eclipse.debug.ui" name="%pluginName" version="
       2.1.1" provider-name="%providerName" class="org.eclipse.debug.
       internal.ui.DebugUIPlugin">
3    <!-- Pliki ze skompilowanym kodem -->
4    <runtime>
5      <library name="dtui.jar">
6         <export name="*"/>
7         <packages prefixes="org.eclipse.debug.ui,org.eclipse.debug.
              internal.ui"/>
8      </library>
9    </runtime>
10
11   <!-- Zależności od innych wtyczek -->
12   <requires>
13     <import plugin="org.eclipse.ui"/>
14     <import plugin="org.eclipse.debug.core"/>
15   </requires>
16
17  <!-- Punkty rozszerzeń -->
18  <extension-point id="consoleColorProviders" name="%
        ConsoleColorProvidersExtensionName" schema="schema/
        consoleColorProviders.exsd"/>
19
20  <!-- Rozszerzenia -->
21  <extension point="org.eclipse.ui.perspectives">
22    <perspective name="%DebugPerspective.name" icon="icons/full/
          cview16/debug_persp.gif" class="org.eclipse.debug.internal.
          ui.DebugPerspectiveFactory" id="org.eclipse.debug.ui.
          DebugPerspective">
23    </perspective>
24  </extension>
```

Warto zwrócić uwagę, iż rozszerzenie funkcjonalności poprzez dołączenie nowej wtyczki powoduje dodanie nowej funkcjonalności, nie może zaś zastąpić bądź usunąć już istniejącej. Nowe punkty rozszerzeń powinny być tak tworzone , aby mieć na uwadze tę właśnie regułę.

Podczas tworzenia nowych wtyczek należy dobrze zastanowić się, w których miejscach możemy pozwolić innym programistom rozszerzyć funkcjonalność naszej aplikacji. Przykładowo, zamiast tworzyć jeden interfejs do prezentowania wyników działania aplikacji możemy korzystając z wzorca projektowego *słuchacza* (ang. *listener*) tak zaprojektować architekturę wtyczki, aby odbiorcą wyników mogły być różne systemy. Mało tego – odbiorców może być wielu i każdy byłby niezależnie od pozostałych poinformowany o zakończeniu i wynikach

działania.

Jest rzeczą bardzo istotną, aby wszystkie rozszerzenia jednego punktu rozszerzeń były traktowane jednakowo, by nie faworyzować jednego rozszerzenia kosztem innych. Jest to tzw. zasada Fair Play – równości wszystkich rozszerzeń. Jeśli tylko programista tworzący punkt rozszerzeń nie będzie traktował w sposób szczególny np. rozszerzeń tego punktu z tej samej wtyczki, to implementacja nowych rozszerzeń będzie dużo prostsza.

Praktycznie wszystkie rozszerzenia Eclipse zachowują zasadę równości. Jedynym odstępstwem jest punkt rozszerzeń zdefiniowany przez samo jądro systemu – punkt `org.eclipse.core.runtime.applications`. Definiujemy tu aplikacje udostępnione na danej platformie – uruchamiana jest zaś tylko jedna z nich – ta przekazana jako parametr do mechanizmu uruchamiającego całą platformę.

Przedstawiona architektura Projektu Eclipse pozwala na integrację dowolnych narzędzi programistycznych. Zanim jednak omówię szczegóły implementacji modułu będącego częścią tej pracy, chciałbym dokładniej scharakteryzować analizę statyczną oraz przedstawić przykłady jej zastosowań.

Rozdział 3

Reguły analizy statycznej

3.1. Wstęp

Analiza statyczna jest to analiza struktury kodu źródłowego lub kodu skompilowanego bez jego uruchomienia. Analiza statyczna może odbywać się na etapie budowania aplikacji, ponieważ jej wyniki są dostępne już podczas kompilacji kodu źródłowego.

Analiza programów w poszukiwaniu błędów jakie mogą wystąpić podczas wykonywania jest nierozstrzygalna – nie istnieje metoda, która pozwala zawsze stwierdzić czy program zawiera błędy. Interesujące mogą być natomiast wszelkie heurystyki, które pomagają analizować kod w poszukiwaniu konstrukcji, które mogą opisywać sytuacje prowadzące do błędów podczas wykonania programu (ang. *runtime errors*).

Istnieje kilka metod analizy statycznej kodu:

- Wykorzystanie narzędzi wyszukujących takie konstrukcje w kodzie źródłowym, które można uznać za potencjalnie niebezpieczne z subiektywnego punktu widzenia.
- Formalne metody, opierające się na matematycznej definicji zachowania programu. Formalne metody wymagają zwykle opisywania aplikacji w języku aksjomatów.
- Obliczanie metryk kodu źródłowego. Dostarczają nam informacji o jakości kodu źródłowego na podstawie danych statystycznych.

W pracy chciałbym skupić się głównie na pierwszej z podanych metod analizy statycznej oraz na błędach jakie można zlokalizować przy pomocy tego typu narzędzi. Metody te mają tą przewagę nad metodami matematycznego opisu aplikacji, że zwykle nie wymagają żadnej (lub niewielkiej) dodatkowej ingerencji w kod źródłowy.

W ramach pracy magisterskiej stworzyłem moduł rozszerzający Platformę Eclipse o możliwość definiowania reguł analizy statycznej. Dołączone przykładowe reguły przy analizowaniu kodu korzystają z drzew AST.

Istnieje wiele nadrzędzi analizujących kod w poszukiwaniu błędów. Aby zintegrować je z platformą Eclipse, można wykorzystać stworzone przeze mnie rozszerzenie platformy.

Jako część pracy magisterskiej przygotowałem wtyczkę do Platformy Eclipse, rozszerzającą funkcjonalność analizatora reguł o integrację z narzędziem JLint. Dzięki temu wszelkie błędy wykrywane przez JLint są prezentowane programiście w edytorze.

W dalszej części pracy magisterskiej opisałem wiele typowych błędów jakie napotykają programiści Javy. Każdy z wymienionych problemów nadaje się do zaimplementowania jako reguła analizy statycznej korzystająca z drzewa AST.

Więcej informacji na temat analizy statycznej kodu można znaleźć na stronie Dawsona Englera [4], zaś w książce [2] można przeczytać o typowych błędach jakie popełniają programiści Javy. Wiele z tych błędów może zostać wychwyconych przez odpowiednie reguły analizy statycznej.

3.2. JLint

JLint [9] jest narzędziem, które analizuje kod źródłowy Javy w poszukiwaniu błędów. Składa się z dwóch aplikacji.

Pierwsza to program `AntiC`, które wyszukuje w kodzie Javy wszelkie konstrukcje jakich mogą używać programiści przenoszący swe nawyki z programowania w C, które mogą powodować poważne błędy nie wykrywane przez kompilator. Wykrywane i zgłaszane są sytuacje w kodzie gdzie np.

- brak jest instrukcji `break` w każdym bloku instrukcji w kodzie korzystającym ze słowa kluczowego `switch`,
- przypisania jako wyrażenie w instrukcji `while`,
- brak nawiasów w skomplikowanych wyrażeniach, co może powodować niezgodne z intencją programisty działanie piorytetów operatorów.

Drugi program (nazwany `JLint`) wyszukuje błędy, które możemy skategoryzować w trzech grupach:

- błędy synchronizacji,
- błędy, które mogły powstać przy tworzeniu podklasy,
- błędy wynikające z analizy przepływu sterowania w aplikacji.

Dokładny opis błędów jakie potrafi wyszukać JLint zawarty jest w dokumencie [9].

3.3. Problemy z analizą statyczną kodu

Automatyczna analiza programu nie jest często rzeczą łatwą. Trzeba mieć świadomość tego, iż czasami błąd jakiego szukamy nie jest łatwy do określenia co może być powodem mniejszej skuteczności analizatora. Zwykle metody analizy statycznej nie oferują 100% skuteczności. Możemy wyróżnić następujące kategorie błędów analizy kodu:

- fałszywe alarmy (gdy sygnalizujemy błąd w poprawnym kodzie),
- brak alarmu (gdy nie potrafimy poprawnie rozpoznać błędnej sytuacji).

3.3.1. Brak alarmu

Trzeba mieć świadomość, że brak błędu w wyniku przeprowadzonej analizy kodu źródłowego na etapie kompilacji przy użyciu nie oznacza, że dana reguła nie została naruszona. Jako przykład może posłużyć reguła zawarta w kompilatorze Javy opracowanym przez firmę Sun Microsystems. Kompilator analizuje kod pod kątem wyszukania tych jego fragmentów, które się nigdy nie wykonają. Przeanalizujmy następujący kod:

```
while (true) {
  System.out.println("W petli");
}
System.out.println("Koniec petli");
```

Próba kompilacji powyższego kodu zakończy się komunikatem `Unreachable code` sygnalizującym błąd w wierszu 4. Kompilator wykrył, że pętla `while` jest pętlą nieskończoną przez co nigdy nie zostanie wykonany kod znajdujący się bezpośrednio za nią. Jednak drobna zmiana powyższego kodu pozwala oszukać kompilator. Kod:

```
boolean b = true;
while (b) {
  System.out.println("W pętli");
}
System.out.println("Koniec pętli");
```

nie generuje już błędu kompilacji. Analizator kodu nie potrafi poradzić sobie z poprawną analizą powyższej konstrukcji. Reguły mogą więc wymagać spełnienia pewnych specyficznych warunków wstępnych.

Co ciekawe kompilator Javy firmy Sun nie sygnalizuje błędu `Unreachable code` w przypadku, gdy warunek zawarty w instrukcji *if* będzie prowadził do sytuacji, w której nie wykona się fragment kodu. Poniższy kod kompiluje się bez błędu:

```
void method1() {
  if (true) {
    return;
  }
  System.out.println("Koniec!");
}

void method2() {
  if (false) {
    System.out.println("Start!");
  }
}
```

Podyktowane jest to tym, że programiści często korzystają z następujących konstrukcji, a firma Sun nie chciała, aby ich używanie było uciążliwe:

```
public static final boolean DEBUG = false;

public void method() {
  if (DEBUG) {
    System.out.println("Executing metod: method()");
  }
}
```

Jak widać reguły analizujące kod często świadomie (przez ich twórców) są ograniczane, tak aby nie powodowały dodatkowej uciążliwości podczas pisania kodu.

3.3.2. Fałszywe alarmy

W wyniku analiza kodu mogą zostać zwrócone informacje o błędach w miejscach, które błędne nie są. Jako przykład jeszcze raz może posłużyć ten sam kompilator Javy firmy

Sun Microsystems. Przy analizowaniu, czy dana zmienna lokalna jest zainicjowana w danym miejscu w kodzie, kompilator stara się uwzględnić przepływ sterowania w kodzie. Przeanalizujmy następujący fragment kodu źródłowego:

```
1 int i;
2 if (true) {
3   i = 1;
4 } else {
5   System.out.println(i);
6 }
7 System.out.println(i);
```

Kod ten nie powoduje żadnych błędów kompilacji. Nigdy nie zostanie wypisana wartość zmiennej `i` w wierszu 5, zmienna `i` nie musi być zatem zainicjowana przed jej wykonaniem, zaś wypisanie wartości zmiennej `i` w wierszu 7 zawsze będzie poprzedzone wykonaniem wiersza 3 inicjującego zmienną `i`. Analogiczna zmiana warunku jak w poprzednim przykładzie:

```
1 int i;
2 boolean b = true;
3 if (b) {
4   i = 1;
5 } else {
6   System.out.println(i);
7 }
8 System.out.println(i);
```

generuje dwa błędy kompilacji informujące o użyciu niezainicjalizowanej zmiennej `i` – w wierszach 6 i 8. Możemy traktować to jako fałszywy alarm, ponieważ zmienna `i` zawsze będzie zainicjowana przed jej użyciem. Reguła ta może sygnalizować więc tylko możliwość wystąpienia błędu, nie zaś sam błąd.

3.4. Kategorie reguł analizy statycznej

Przedstawione w tym rozdziale reguły zostały dobrane tak, aby przedstawić różnorodność tematów oraz problemów, które mogą zostać zidentyfikowane przy wykorzystaniu mechanizmu audytu kodu źródłowego.

Reguły zaprezentowane w tym rozdziale podzielone zostały na następujące kategorie:

- Sprawdzanie kontraktów wymaganych przy implementacji metod

  Możemy sprawdzić czy dodatkowe warunki jakie powinny być spełnione przy definiowaniu nowej funkcjonalności są zachowane, np. czy wraz z nową implementacją metody `equals(Object)` zmieniona została też implementacja metody `hashCode()`. Warunki te są zwykle opisane w dokumentacji biblioteki.

- Złe praktyki programistyczne

  Wielu programistów podczas pisania programu implementuje pewne rozwiązania w sposób nieefektywny, co może wynikać zarówno z niewiedzy, jak i z próby uproszczenia zadania programistycznego (czyli po prostu z lenistwa).

- Typowe błędy

  Część błędów jakie mogą powstać podczas pisania programu, wynika z nieuwagi bądź niewiedzy programisty. Błędy takie zwykle dość szybko doprowadzą do nieprawidłowego działania aplikacji.

- Styl kodu źródłowego

  Utrzymanie uporządkowanego kodu źródłowego zwiększa jego czytelność.

- Programowanie rozproszone

  Istnieją konstrukcje programistyczne, które spowodują błędne działanie aplikacji tylko wtedy, gdy będzie ona działać w środowisku wielowątkowym.

3.5. Sprawdzanie kontraktów metod

W rozdziale tym chciałbym przedstawić reguły sprawdzające czy spełnione są kontrakty wymagane przez metody z biblioteki standardowej Javy.

3.5.1. Metoda `finalize()`

Metoda `finalize()` zdefiniowana w klasie `Object` pozwala wykonać pewien kod dla danego obiektu w chwili gdy odśmiecacz stwierdzi, iż nie istnieje żadna referencja do danego obiektu. Metoda jest często porównywana do destruktorów z C++, jednak jej zachowanie i sens wprowadzenia jest zupełnie inny niż destruktorów i takie porównania prowadzą do pomyłek i nieporozumień.

Osobiście odradzam zastępowanie metody `finalize()`. Obiekty, które zawierają taką metodę zastępującą implementację z klasy `Object` są traktowane w sposób szczególny przez odśmiecacz – alokacja i zwalnianie przydzielonej im pamięci trwa od kilku do kilkunastu razy wolniej niż obiektów, które nie zastępują tej metody. Poza tym maszyna wirtualna nie określa momentu kiedy taka metoda zostanie wywołana – może to nastąpić po kilku minutach, godzinach, czy dniach od chwili, gdy obiekt taki stanie się nieosiągalny. Mało tego – nie mamy gwarancji, że metoda ta zostanie wykonana choć raz dla każdego obiektu. Używanie metody `finalize()` do np. zwalniania zasobów systemowych nie jest więc dobrym pomysłem.

Jeśli jednak skorzystanie z metody `finalize()` okaże się konieczne, to należy przy jej pisaniu przestrzegać kilku reguł. Reguły te można łatwo zaimplementować korzystając z drzewa AST kodu źródłowego.

Dobrą praktyką jest wywoływanie `super.finalize()` po wykonaniu metody `finalize()`. Pozwoli to wykonać wszystkie implementacje tej metody w hierarchii danej klasy.

Metoda `finalize()`, która zawiera tylko wywołanie `super.finalize()` powinna zostać usunięta — obiekty nie zawierające tej metody będą alokowane i zwalniane szybciej.

Metoda `finalize()` powinna posiadać modyfikator `protected`, aby uniknąć umieszczenia jej w publicznym interfejsie klasy.

3.5.2. Klonowanie obiektów

Java dostarcza API do tworzenia kopii istniejącego obiektu. Należy ostrożnie korzystać z funkcji klonowania, ponieważ jest to metoda tworzenia nowych obiektów danej klasy inaczej niż przez użycie konstruktora obiektu.

Należy wystrzegać się konstrukcji, gdy klasa obiektu będącego singletonem implementuje interfejs `Cloneable`, zmienia to bowiem zachowanie metody `clone()` z klasy `Object`.

W przypadku klas, które tego interfejsu nie implementują zgłasza ona zawsze wyjątek `CloneNotSupportedException`, dla pozostałych klas tworzy nowy obiekt tej samej klasy, ale robi to z pominięciem jakiegokolwiek konstruktora oraz ustawia wartości pól obiektu na wartości identyczne z kopiowanym obiektem (tzw. płytka kopia obiektu). W przypadku, gdy klasa obiektu będącego singletonem implementuje interfejs `Cloneable`, metoda `clone()` powinna przekazać obiekt `this`.

Powyższe wymaganie możemy zapisać dość prosto jako regułę analizy statycznej, choć wymaga to poprawnego zidentyfikowania klasy jako singletonu (istnieje wiele możliwości takiej identyfikacji – poprzez dodatkową wskazówkę w komentarzu lub poprzez analizę kodu pod kątem typowych implementacji singletonu).

W przypadku pozostałych klas implementacja metody `clone()` powinna spełniać kilka warunków, aby można było uznać ją za poprawną.

Przytoczę reguły jakie powinna spełniać metoda `clone()` zgodnie z dokumentacją Javy:

- `x.clone() != x`
- `x.clone().getClass() == x.getClass()`

Powyższe warunki nie są wymagane, ale jest to spodziewane zachowanie metody `clone()`. Przy implementacji tej metody dla obiektów singletonów naruszyliśmy pierwszą z tych reguł, jednak singleton jest szczególnym przypadkiem klonowanego obiektu. Zwykle to właśnie tych dwóch własności będziemy oczekiwać od klonowanych obiektów.

Implementacja metody `clone()` w klasie `Object` jest wystarczająca w większości przypadków. Musimy zmienić jej zachowanie tylko wtedy, gdy trzeba sklonować także część pól klasy (lub stworzyć nowe obiekty). Dotyczy to głównie pól, których wartości istnieją w związku agregacji z danym obiektem.

Tworząc nowy obiekt w metodzie `clone()` należy wystrzegać się używania do tego celu konstruktora. Obiekt danej klasy należy pobrać poprzez `super.clone()`. Dzięki temu metoda `clone()` będzie działała poprawnie także w podklasach. Przy użyciu konstruktora, poprawne zaimplementowanie metody `clone()` w podklasach będzie wymagać nowej implementacji metody `clone()`, aby prawdziwe było wyrażenie `x.clone().getClass() == x.getClass()`. Implementacja taka nie zawsze jest możliwa – nie możemy wtedy skopiować nieudostępnionych na zewnątrz pól prywatnych klasy.

To zachowanie metody `clone()` (czyli kopiowanie pól prywatnych klasy) jest bardzo niebezpieczne, ponieważ można stworzyć dwie referencje do tego samego obiektu, także wtedy, gdy chcielibyśmy uniknąć takiego zachowania. Co ciekawe twórca klasy może zakładać, że istnieje tylko jedna referencja do danego obiektu i nie udostępniać obiektu jako wspomagającego klonowanie (czyli jego klasa nie implementuje obiektu `Cloneable`). Oto przykładowy kod:

```
1  public interface ZrodloDanych {
2    public int czytaj() throws IOException;
3    public void zamknij() throws IOException;
4  }
5
6  public class Plik implements ZrodloDanych {
7    private FileInputStream plik;
8
9    public ZrodoDanych(String nazwaPliku)
10     throws IOException
11   {
```

```
12      plik = new FileInputStream(nazwaPliku);
13    }
14
15    public int czytaj() throws IOException {
16      return plik.read();
17    }
18
19    public void zamknij() throws IOException {
20      plik.close();
21    }
22  }
```

Fakt przechowywania przez klasę `Plik` referencji do obiektu `FileInputStream` jest przed użytkownikiem klasy ukryty. Referencja ta przechowywana jest w polu prywatnym. Użytkownik klasy nie jest świadomy jej implementacji. Inną równie poprawną implementacją jest np.:

```
1   public class PlikBuforowany implements ZrodloDanych {
2
3     private int[] bufor;
4     private int pozycja;
5
6     public ZrodoDanych(String nazwaPliku)
7       throws IOException
8     {
9       FileInputStream plik = new FileInputStream(nazwaPliku);
10      bufor = FileUtils.buforujPlik(plik);
11      plik.close();
12    }
13
14    public int czytaj() throws IOException {
15      if (pozycja >= bufor.length)
16        return -1;
17      return bufor[pozycja++];
18    }
19
20    public void zamknij() throws IOException {
21    }
22  }
```

Załóżmy teraz, że programista chciałby, aby metoda `read` podmieniała czytane znaki przy czytaniu pliku.

```
1   public class FiltrZnakow extends Plik implements Cloneable {
2
3     private Filtr filtr;
4
5     public ZrodoDanych(String nazwaPliku, Filtr f)
6       throws IOException
7     {
8       super(nazwaPliku);
9       filtr = f;
10    }
11
```

```
12    public int czytaj() throws IOException {
13      int ret = super.czytaj();
14      if (ret != -1) {
15        ret = filtr.zamien(ret);
16      }
17      return ret;
18    }
19
20    public void zamknij() throws IOException {
21    }
22  }
```

Jak widać programista zaimplementował również interfejs `Cloneable`. Jest świadomy tego, że może stworzyć instancję nowego obiektu poprzez `clone()` oraz tego, że sklonowany obiekt będzie przechowywał referencję do tego samego obiektu `Filtr`. Niestety nie wie, że jeśli klasa będzie implementować interfejs `Cloneable`, to przy klonowaniu obiektu tej klasy skopiują się także pola prywatne z nadklasy (programista nie wie nawet jakie pola są prywatne!). Jeśli `FiltrZnakow` rozszerzałby klasę `PlikBuforowany`, to zamknięcie jednego obiektu nie wpływałoby na zachowanie drugiego. Przy rozszerzeniu klasy `Plik` wykonanie metody `zamknij()` na jednym z obiektów, spowoduje wykonanie metody `close()` na wspólnym obiekcie `FileInputStream` obu obiektów.

Doszło do takiej sytuacji ponieważ:

- programista tworzący klasę `Plik` nie spodziewał się, że może powstać klasa dziedzicząca po tej klasie, która będzie implementować interfejs `Cloneable`,
- programista tworzący klasę `FiltrZnakow` nie znał implementacji klasy `Plik`.

Aby się ustrzec przed tego typu przypadkami programista piszący klasę `Plik` powinien zabronić klonowania tej klasy w przyszłości. Mógłby to zrobić np. poprzez oznaczenie klasy jako `final`, ale wtedy nie tylko zabroniłby tworzenia podklas, które można klonować, ale także w ogóle tworzenia podklas klasy `Plik`. Inną metodą jest nadpisanie metody `clone()` implementacją, która zawsze zgłasza wyjątek `CloneNotSupportedException`. Jednak często przy tworzeniu nowej klasy nie zwraca się uwagi na to czy nadaje się ona do klonowania. Z tego powodu często programistom nie wolno korzystać z klonowania, zamiast tego proponuje się korzystanie z konstruktorów kopiujących.

Gdy jednak implementujemy metodę `clone()`, dobrze jest uwzględnić takie przypadki użycia klasy, gdy jej podklasy nie będą miały możliwości klonowania. Aby zabronić klonowania danej klasy należy w metodzie `clone()` zgłosić wyjątek `CloneNotSupportedException`. Często można jednak napotkać taką definicję metody `clone()`, która nie zawiera informacji o możliwości zgłoszenia przez nią wyjątku `CloneNotSupportedException`. W Javie wszelkie wyjątki, jakie może zgłosić metoda, które nie należą do podklasy `RuntimeException`, powinny być wyspecyfikowane w deklaracji metody. Nadpisując taką metodę w podklasie nie można wydłużyć listy możliwych do zgłoszenia wyjątków, można ją natomiast skrócić. Usunięcie z listy możliwych do zgłoszenia wyjątków klasy `CloneNotSupportedException` powoduje, że podklasy nie mogą zgłaszać tego wyjątku, przez co metoda `clone()` nie będzie mogła poprawnie (czyli zgłaszając wyjątek `CloneNotSupportedException`) poinformować wołającej ją metody o braku możliwości klonowania obiektu danej klasy.

Warto jest także sprawdzić, czy jeśli klasa implementuje metodę `clone()`, to implementuje także interfejs `Cloneable` (lub czy jest on implementowany w podklasie). Jeśli klasa nie będzie implementować interfejsu `Cloneable`, to metoda `clone()` z klasy `Object` (np. poprzez

wykonanie `super.clone()`) zgłosi wyjątek `CloneNotSupportedException`, co nie jest raczej spodziewanym rezultatem.

Podsumowując, możemy wyznaczyć następujące reguły, jakie można zaimplementować w module analizy statycznej kodu dotyczące klonowania obiektów:

- Szczególne traktowanie klonowania dla klas singletonów.
- Korzystanie z `super.clone()` zamiast z konstruktorów przy tworzeniu obiektu w metodzie `clone()`.
- Konieczność wyspecyfikowania `CloneNotSupportedException` jako możliwego do zgłoszenia wyjątku w sygnaturze metody `clone()`.
- Sprawdzenie czy klasa nadpisująca metodę `clone()` implementuje także interfejs `Cloneable`.
- Zakaz korzystania z mechanizmu klonowania.

3.5.3. Metoda `equals(Object)` w Javie

Definicja tej metody zawarta jest w klasie Object, tak więc metodę tę posiadają wszystkie obiekty. Od jej prawidłowej implementacji zależy choćby prawidłowe działanie kolekcji z pakietu `java.util`. Wszelkie błędy w jej implementacji mogą skutkować błędami, które pojawiają się bardzo niedeterministycznie, gdyż mogą zależeć od wewnętrznych mechanizmów danej kolekcji.

`equals(Object o)` i `hashCode()`

Praktycznie zawsze metoda `equals(Object)` musi pociągać za sobą implementację innej metody zdefiniowanej w klasie `Object`. Jest nią `hashCode()`. Otóż zgodnie z dokumentacją Javadoc [11] dla klasy `Object`, jeśli dwa obiekty `o1` i `o2` przekazują wartość `true` przy wywołaniu `o1.equals(o2)`, to musi być także spełniony warunek:

```
1 o1.hashCode() == o2.hashCode()
```

Problem pojawi się np. przy korzystaniu z kolekcji używających funkcji skrótu takich jak `HashMap` czy `HashSet`. Oto przykład kodu, który nie zadziała zgodnie z oczekiwaniami:

```
1  public class Test {
2    public int x;
3
4    public Test(int px) {
5      x = px;
6    }
7
8    public boolean equals(Object o) {
9      if (!(o instanceof Test)) return false;
10     return ((Test) o).x == x;
11   }
12
13   public static final void main(String[] args) {
14     Set set = new HashSet();
15     set.add(new Test(10));
16     // Poniższy wiersz wypisze false, choć zbiór zawiera
17     // podany obiekt.
```

```
18        System.out.println(set.contains(new Test(10)));
19    }
20  }
```

Sprawdzenie czy powyższy warunek jest spełniony praktycznie jest bardzo trudne do zweryfikowania metodą analizy statycznej. Zdecydowanie polecaną metodą jest tutaj użycie pakietu do przeprowadzania testów jednostkowych. Jednak często zdarza się, że programiści po prostu zapominają o implementacji tej metody. Podczas analizy statycznej można sprawdzić, czy w klasach, w których metoda `equals(Object)` została przedefiniowana, jest również przedefiniowana metoda `hashCode()`.

Opcjonalnie możemy sprawdzić istnienie metod do testów jednostkowych (np. z pakietu JUnit) metody `hashCode()`.

Deklaracja `equals(Object)`

Innym częstym błędem spotykanym przy deklaracji metody `equals()`, jest przeciążenie jej z parametrem innym niż `Object`. Jeśli metoda taka występuje obok `equals(Object)`, to nie stanowi to problemu. Jednak jeśli metoda `equals(Object)` nie jest nadpisana, oznacza to prawdopodobnie pomyłkę programisty. Z takiej metody nie będą korzystały kolekcje ze standardowej biblioteki Javy, gdyż używają one metody o sygnaturze `equals(Object)`.

Metoda `equals()` dla typów obiektowych

Jednak największym problemem jaki może sprawić wymieniona metoda jest jej poprawna implementacja dla typów obiektowych. Zgodnie ze specyfikacją zawartą w dokumentacji Javadoc [11], metoda ta musi być relacją równoważności, czyli dla dowolnych obiektów `o1`, `o2` i `o3` różnych od `null` powinny być spełnione następujące warunki:

1. Symetryczność – jeśli `o1.equals(o2)`, to `o2.equals(o1)`;
2. Zwrotność – `o1.equals(o1)`;
3. Przechodniość – jeśli `o1.equals(o2)` oraz `o2.equals(o3)`, to `o1.equals(o3)`.

Porównanie z wartością `null` powinno zawsze przekazać wartość `false`.

Zdefiniujmy metodę `equals()` dla przykładowej klasy `Punkt`:

```
1   public class Punkt {
2     private int x;
3     private int y;
4
5     public Punkt(int px, int py) {
6       this.x = px;
7       this.y = py;
8     }
9
10    public boolean equals(Object o) {
11      if (!(o instanceof Punkt)) return false;
12      Punkt p = (Punkt) o;
13      return p.x == x && p.y == y;
14    }
15
```

```
16    public int hashCode() {
17      return x + y;
18    }
19  }
```

Przedstawiona implementacja metody `equals()` spełnia kontrakt dla metody `Object.equals(Object)` dla dowolnych obiektów.

Wprowadźmy teraz podklasę klasy `Punkt`:

```
1  public PunktKolorowy extends Punkt {
2    private int kolor;
3
4    public PunktKolorowy(int px, int py, int pkolor) {
5      super(px, py)
6      kolor = pkolor;
7    }
8
9    public boolean equals(Object o) {
10     ???
11   }
12 }
```

W podklasie zdefiniowaliśmy dodatkowe pole dla obiektu typu `PunktKolorowy`. Spróbujmy uwzględnić je w metodzie `equals(Object)`, aby dwa punkty o tych samych współrzędnych, ale różnych kolorach nie były traktowane jako jeden i ten sam punkt.

```
1  public boolean equals(Object o) {
2    if (!(o instanceof PunktKolorowy))
3      return false;
4    PunktKolorowy pk = (PunktKolorowy) o;
5    return super.equals(o) && pk.kolor == kolor;
6  }
```

Pokazany kod jest jedną z najczęściej spotykanych implementacji metody `equals()`. Zakładamy w niej, że żaden punkt kolorowy nie jest równy punktowi bez kolorów. Niestety, powyższa funkcja nie spełnia warunków relacji równoważności – nie jest zachowana symetria. Dla dwóch obiektów, jednego klasy `Punkt` o nazwie `punkt` oraz drugiego klasy `PunktKolorowy` o nazwie `punktKolorowy` o tych samych współrzędnych wywołanie:

```
1  punkt.equals(punktKolorowy)
```

przekazuje wartość `true`, natomiast:

```
1  punktKolorowy.equals(punkt)
```

przekazuje wartość `false`.

Spróbujmy poprawić ten kod. Problem powoduje już pierwsza instrukcja metody, sprawdzająca czy dany obiekt jest klasy `PunktKolorowy`, a musimy przecież pozwolić, aby porównanie z obiektem z nadklasy również przekazało wartość `true`. Zmiana tego warunku zmienia nam semantykę tej metody – wszystkie punkty kolorowe będą przekazywały wartość `true` przy porównaniu z obiektem klasy `Punkt` o tych samych współrzędnych.

```
public boolean equals(Object o) {
  if (!(o instanceof Punkt))
    return false;
  if (!(o instanceof PunktKolorowy))
    return o.equals(this);
  PuktKolorowy pk = (PunktKolorowy)o;
  return super.equals(o) && pk.kolor == kolor;
}
```

Ten kod niestety również jest niepoprawny. O ile relacja `equals(Object)` zdefiniowana w podany sposób jest zwrotna i symetryczna, to niestety nie jest ona przechodnia (każde dwa obiekty `PunktKolorowy` o tych samych współrzędnych, ale różnych kolorach są równe obiektowi `Punkt` o tych współrzędnych, natomiast same nie są równe).

Jak w takim razie napisać poprawną metodę `equals(Object)`? Okazuje się, że nie jest to możliwe – metody `equals(Object)` nie można przedefiniować tak, aby uwzględniała nowe pole. Zawsze dodanie nowego pola w podklasie spowoduje opisany problem. Co można w takim razie zrobić?

Jednym z rozwiązań jest rezygnacja z dziedziczenia na rzecz kompozycji, obiekt `PunktKolorowy` nie jest podklasą klasy `Punkt`. Oto poprawne rozwiązanie:

```
public PunktKolorowy {
  private Punkt punkt;
  public int kolor;
  public PunktKolorowy(int px, int py, int pkolor) {
    punkt = new Punkt(px, py);
    kolor = pkolor;
  }

  public boolean equals(Object o) {
    if (!(o instanceof PunktKolorowy))
      return false;
    PunktKolorowy pk = (PunktKolorowy) o;
    return punkt.equals(pk.punkt) && kolor == pk.kolor;
  }

  public jakoPunkt() {
    return new Punkt(x, y);
  }
}
```

Problem ten nie wystąpiłby także wtedy, gdyby klasa `Punkt` była klasą abstrakcyjną, natomiast istniałaby klasa `PunktBezKoloru` dziedzicząca jedynie po klasie `Punkt` bez definiowania żadnych metod czy pól. W takim przypadku porównanie pomiędzy obiektami klasy `Punkt` oraz `PunktKolorowy` nigdy nie będzie mogło mieć miejsca (klasa `Punkt` jest klasą abstrakcyjną), natomiast klasy `PunktKolorowy` oraz `PunktBezKoloru` nie są związane relacją dziedziczenia, tak więc problem ten nie wystąpi. Więcej informacji o problemach z implementacją metody `equals(Object)` można znaleźć w [2], skąd został zaczerpnięty powyższy przykład.

Narzędzie do analizy statycznej powinno znaleźć takie miejsca w kodzie, gdzie metoda `equals()` zostaje przedefiniowana w klasie, w której dodano nowe pole do jej definicji, a na ścieżce dziedziczenia metoda `equals()` została już przedefiniowana.

Twórcy biblioteki standardowej Javy również nie ustrzegli się tego błędu. Implementacja klasy `java.util.Timestamp` dziedziczy implementację po klasie `java.util.Date` i dodaje do niej pole z liczbą nanosekund. Zaimplementowana w niej metoda `equals(Object)` nie zachowuje symetrii. W dokumentacji tej klasy [12] znajdujemy za to zalecenie aby nie używać obiektów klasy `Timestamp` tam, gdzie spodziewany jest obiekt klasy `Date`.

3.5.4. Serializacja obiektów

Serializacji obiektów warto poświęcić jeszcze chwilę. Jeśli tworzymy podklasę klasy, która nie implementowała interfejsu `Serializable`, zaś podklasa go implementuje, to wymaga się aby klasa, po której dziedziczymy posiadała widoczny dla podklasy konstruktor bezparametrowy. Warto jest stworzyć regułę, która sprawdzi czy taki konstruktor istnieje, aby uniknąć błędów podczas wykonania programu. Konstruktor ten potrzebny jest przy deserializacji obiektu, aby zainicjować pola klas nieserializowalnych (ich wartości inicjowane są tym właśnie konstruktorem).

3.5.5. Biblioteka rejestrująca zdarzenia Log4J

Twórcy biblioteki Log4J zalecają, aby korzystając z ich produktu używać pewnych stałych konstrukcji w kodzie, by jego utrzymanie było łatwe. Proponują aby każda klasa, w której będzie wykorzystany moduł rejestrowania zdarzeń, definiowała prywatną zmienną statyczną o ustalonej, zawsze tej samej nazwie (np. `logger`). Zmienna ta inicjowana powinna być obiektem klasy `Logger` z przekazanym jako parametr obiektem klasy, w której ta zmienna jest zdefiniowana. Wszelkie odwołania z bloków zdefiniowanych w tej klasie do systemu rejestrowania zdarzeń powinny odbywać się poprzez to statyczne pole. Nie jest to co prawda konieczne, ale wymuszenie stosowania tej zasady znacznie ułatwi zarządzanie systemem rejestrowania oraz poprawi czytelność kodu.

Reguła powinna sprawdzać, czy wywołania metod rejestrujących zdarzenia wykonywane są wyłącznie na obiekcie, którego referencja przechowywana jest w polu o nazwie `logger` oraz czy deklaracja pola `logger` wygląda następująco:

```
1 private static final Logger logger = Logger.getLogger(NazwaKlasy.
      class);
```

3.6. Złe praktyki programistyczne

Istnieje kategoria problemów polegających na tym, że programista implementuje pewne rozwiązania w sposób nieefektywny lub wręcz błędny. W tej części rozdziału przedstawiam kilka takich konstrukcji. Każdą z wymienionych dalej reguł można zaimplementować przy pomocy mechanizmu analizy statycznej wykorzystującego analizę drzew AST.

3.6.1. Wywoływanie przez konstruktor metod wirtualnych tego samego obiektu

Jeśli konstruktor wywołuje metody, które mogą zostać nadpisane w podklasach, to może się zdarzyć, że metody te zostaną wykonane na obiekcie, który nie został w pełni skonstruowany. W szczególności metody te zostaną wykonane zanim wykonany będzie konstruktor w podklasie.

```
1  class A {
2    private int size;
3
4    A(int p_size) {
5      size = p_size;
6    }
7
8    A() {
9      x = getDefaultSize();
10   }
11
12   int getDefaultSize() {
13     return 10;
14   }
15
16   int getSize() {
17     return size;
18   }
19 }
20
21 class B extends A {
22   private A container;
23
24   B() {
25     container = new A();
26   }
27
28   int getDefaultSize() {
29     return container.getMaxSize();
30   }
```

W podanym przykładzie próba stworzenia obiektu klasy `B` poprzez użycie konstruktora `B()` zakończy się zgłoszeniem wyjątku `NullPointerException`. Zanim zostanie wykonane przypisanie `container = new A()` pole `container` ma wartość `null`, zaś wcześniej zostanie wykonany bezparametrowy konstruktor z nadklasy `A()`. W konstruktorze tym występuje wywołanie metody `getDefaultSize()` (czego autor klasy B nie musi być świadomy). Nadpisanie tej metody w klasie `B` prowadzi do próby wykonania metody `getMaxSize()` na obiekcie, do którego referencja przechowywana jest w polu `container`. Jako że konstruktor `B()` nie został jeszcze wykonany, zmienna `container` zawiera wartość `null` co prowadzi właśnie do zgłoszenia wspomnianiego wyjątku.

Do takiej sytuacji może dochodzić bardzo często, ponieważ programista tworzący podklasę często nie zna (i tak naprawdę nie powinno się od niego wymagać tej znajomości) implementacji konstruktora w nadklasie. Musi on mieć pewność, że wszelkie zdefiniowane przez niego metody będą wywoływane tylko na w pełni skonstruowanych obiektach. Dlatego też należy dbać o to, aby w konstruktorze mogły być wywoływane tylko te metody, które nie mogą być nadpisane w podklasach.

3.6.2. Wyjątki

Programiści często przechwytują wyjątki zbyt ogólne – bądź to klasy `RuntimeException`, `Exception`, czy wręcz `Throwable`. O ile przechwytywanie dwóch pierwszych klas wyjątków nie jest jeszcze błędem, to przechwytywanie wszystkich obiektów `Throwable` nie powinno pod

żadnym pozorem znajdować się w kodzie.

W hierarchii klas, podklasami `Throwable` są nie tylko klasy dziedziczące po klasie `Exception` (będące wyjątkami), ale także klasy dziedziczące po klasie `Error`. Są to klasy błędów, z którymi aplikacja Javy nie będzie potrafiła sobie poradzić, takie jak np.

`OutOfMemoryError` — błąd zgłaszany, gdy maszyna wirtualna nie potrafi poprawnie kontynuować pracy z powodu braku pamięci lub sygnalizuje błąd w implementacji maszyny wirtualnej;

`ThreadDeath` — błąd zgłaszany w wątku w chwili, gdy na jego obiekcie zostanie wywołana metoda `stop()`. Jeśli błąd ten zostanie przechwycony, to musi zostać zgłoszony ponownie, aby wątek mógł się poprawnie zakończyć. Co ciekawe błąd ten został specjalnie umieszczony w hierarchii jako podklasa `Error` zamiast `Exception` pomimo tego, że nie sygnalizuje sytuacji błędu. Projektanci biblioteki Javy stwierdzili, że często występuje w kodzie sytuacja, gdy przechwytywany jest wyjątek ogólnej klasy `Exception`, a następnie jest on porzucany. Jeśli `ThreadDeath` byłby wyjątkiem, to aplikacje te nie potrafiłyby poprawnie obsłużyć sytuacji zakończenia wątku;

`AssertionError` — błąd naruszenia asercji;

`InternalError` **lub** `UnknownError` — błędy zgłaszane przez maszynę wirtualną, jeśli znajdzie się ona w nieprzewidzianym stanie.

Podobnie jak przechwytywanie `Throwable`, zakazane powinno być oczywiście przechwytywanie wyjątków klasy `Error`. Jednak jako dobrą praktykę programistyczną należy przyjąć zasadę unikania przechwytywania wyjątków zgłaszanych przez maszynę wirtualną takich jak `NullPointerException`, `ArrayIndexOutOfBoundsException`, `ClassCastException`, `ConcurrentModificationException`, `ArrayStoreException`. Kod powinien być napisany tak, aby programista nie musiał zakładać, iż zgłoszenie jednego z tych wyjątków jest sytuacją jaką powinien przewidzieć i obsłużyć. Zgłoszenie tego typu wyjątków powinno sygnalizować błędy w kodzie źródłowym — nie powinny one stanowić części poprawnego przepływu sterowania w aplikacji.

Kolejnym często popełnianym błędem jest deklarowanie w sygnaturze metody przy użyciu słowa kluczowego `throws` faktu zgłaszania przez metodę wyjątków klasy `Exception`. Praktyka taka w zasadzie uniemożliwia poprawną obsługę wyjątków – sygnatura nie informuje o możliwych klasach wyjątków jakie mogą zostać zgłoszone, przez co wywołania tych metod zwykle są otoczone blokiem `try/catch` przechwytującym w jednej klauzuli `catch` wszystkie wyjątki klas `Exception`. W jednakowy sposób są wtedy traktowane wyjątki, których programista spodziewa się i potrafi je poprawnie obsłużyć (np. wyjątki zerwania połączenia do bazy danych), jak również takie wyjątki jak `NullPointerException` zwykle sygnalizujące błędy w kodzie źródłowym.

Podobnie jak nie jest zalecane przechwytywanie zbyt ogólnych wyjątków, tak nie jest zalecane zgłaszanie wyjątków takich klas jak np. `Exception`, `Error`, `Throwable`, `RuntimeException`. Również wyjątki typu `NullPointerException` powinny być zarezerwowane dla maszyny wirtualnej jako sygnał, iż wystąpiła niespodziewana sytuacja w programie. Czasami programiści zakładają, że metody mogą zgłaszać wyjątek `NullPointerException` jeśli parametry wywołania metody nie są poprawne (np. jeśli wartość parametru jest spoza ustalonego i udokumentowanego zakresu). Poprawnym zachowaniem aplikacji w takim wypadku jest jednak wykrycie takiej sytuacji i zgłoszenie np. `IllegalArgumentException` lub jego podklasy.

Wszystkie powyższe zalecenia można zdefiniować jako reguły dla mechanizmów analizy statycznej. Reguły te wykonają proste testy na drzewie AST kodu źródłowego.

3.6.3. Przepływ sterowania z wykorzystaniem wyjątków

Programiści czasami mają ochotę używać wyjątków jako naturalnych elementów działania aplikacji, nie traktując ich wystąpienia jako sytuacji wyjątkowych – informujących o błędach krytycznych. O ile kodu zastępującego kod:

```
1 if (x != null)
2   y = x.oblicz();
3 else
4   y = 0;
```

na przykład takim:

```
1 try {
2   y = x.oblicz();
3 } catch (NullPointerException ex) {
4   y = 0;
5 }
```

raczej się nie spotyka, to następująca konstrukcja jest popularna wśród programistów:

```
1 Iterator iter = col.iterator();
2 try {
3   while (true) {
4     Object o = iter.next();
5     processObject(o);
6   }
7 } catch (NoSuchElementException ex) { }
```

A przecież poprawna wersja nie jest dużo bardziej skomplikowana. Oto tradycyjny sposób iterowania po obiektach w kolekcji:

```
1 Iterator iter = col.iterator();
2 while (iter.hasNext()) {
3   Object o = iter.next();
4   processObject(o);
5 }
```

Programiści korzystający z poprzedniej konstrukcji zakładają, że metoda `Iterator.next()` po przeiterowaniu całej kolekcji zgłosi wyjątek `NoSuchElementException` – i tak rzeczywiście jest. Przytaczanym argumentem za używaniem takich konstrukcji w kodzie jest wydajność: rezygnujemy z wywołań metody `Iterator.hasNext()`, przez co pętla wykonuje się szybciej.

Nie jest to jednak prawda. Rzeczywista wydajność obu rozwiązań w dużej mierze zależy od użytej maszyny wirtualnej Javy. Często obsługa sytuacji wyjątkowych powoduje nieznaczny spadek wydajności danego fragmentu kodu niwelując potencjalne zyski. Większym problem niż brak spodziewanego zysku wydajności jest natomiast zmieniona semantyka programu. Otóż jeśli metoda `processObject()` zgłosi wyjątek `NoSuchElementException` (jest to wyjątek dziedziczący po `RutimeException`, więc metoda może go zgłosić mimo braku takiej

informacji w sygnaturze metody), to kod sterowany wyjątkiem potraktuje go jako informację o zakończeniu iterowania po danej kolekcji i informacja o wystąpieniu sytuacji wyjątkowej w metodzie `processObject()` zostanie utracona. Oprócz tego kod będzie wykonywany dalej, prawdopodobnie z założeniem, iż wszystkie elementy kolekcji zostały odwiedzone, co nie musi być prawdą.

Aby zabezpieczyć się przed takimi praktykami należy wychwycić konstrukcje w kodzie, w których pętla nieskończona jest otoczona blokiem `try/catch` lub gdy blok `catch` jest pusty.

3.6.4. Dwukrotnie sprawdzana blokada

Programiści Javy często są przekonani, że koszt synchronizacji jest wysoki, przez co starają się go unikać, aby nie powodować spadków wydajności. Swego czasu pojawiła się metoda omijania kosztów synchronizacji nazwana dwukrotnie sprawdzaną blokadą (ang. *Double Checked Locking*).

Oto przykładowa definicja klasy implementującej wzorzec singletona:

```
1  /* Wersja jednowątkowa */
2
3  class Foo {
4    private static final Singleton singleton;
5
6    public static Singleton getSingleton() {
7      if (singleton == null)
8        singleton = new Singleton();
9        return singleton;
10     }
11 }
```

Kod ten działa poprawnie w środowisku jednowątkowym. Jednak jeśli w tym samym czasie dwa różne wątki mogą wywołać metodę `getSingleton()`, to jest konieczna synchronizacja dostępu do metody, gdyż w przeciwnym razie konstruktor `new Singleton()` może zostać wykonany dwukrotnie:

```
1  /* Prawidłowa wersja wielowątkowa */
2
3  class Foo {
4    private static final Singleton singleton;
5
6    public static synchronized Singleton getSingleton() {
7      if (singleton == null)
8        singleton = new Singleton();
9      return singleton;
10   }
11 }
```

Aby uniknąć kosztów wielokrotnej (i niepotrzebnej) synchronizacji już po skonstruowaniu obiektu singletonu, wymyślono mechanizm Double-Checked Locking:

```
1  /* Double-Checked Locking */
2
3  class Foo {
4    private static final Singleton singleton;
5
6    public static Singleton getSingleton() {
7      if (singleton == null)
```

```
 8         synchronized (Foo.class) {
 9           if (singleton == null)
10             singleton = new Singleton();
11         }
12       return singleton;
13     }
14   }
```

Idea tego kodu jest prosta. Najpierw porównujemy czy obiekt singletonu przypisany został na zmienną `singleton`. Jeśli nie, to dopiero w tym momencie synchronizujemy się na obiekcie klasy `Foo`. Po przypisaniu obiektu singletonu na zmienną `singleton` pierwszy test nigdy nie będzie prawdziwy, co pozwoli uniknąć synchronizacji i jej kosztów przy kolejnych wywołaniach metody `getSingleton()`.

Powyższa idea wygląda bardzo sprytnie, lecz niestety jej zastosowanie może prowadzić do sytuacji, w której konstruktor klasy `Singleton` będzie wykonany więcej niż jeden raz. Dokładny opis przyczyny takiego zachowania maszyny wirtualnej można przeczytać w artykule [1].

Jeśli zidentyfikujemy poprawnie takie konstrukcje w kodzie źródłowym, w którym zastosowano wzorzec Double Checked Locking, to możemy poinstruować użytkownika o kłopotach z tym związanych oraz zaproponować rozwiązanie alternatywne (np. z synchronizowaną metodą `getSingleton()`.

Dla klas singletonów możemy jeszcze skorzystać z następującej konstrukcji:

```
1 /* Dla singletonów */
2
3 class Foo {
4   public static final Singleton singleton = new Sigleton();
5 }
```

Double Checked Locking działa poprawnie dla typów prostych o długości maksymalnie 32-bitów (np. `int`, `char`). Dla obiektów typu `long` czy `double` poniższy kod może dać jednak niepoprawne wyniki.

```
 1 /* Działająca wersja DCL dla 32-bitowych zmiennych */
 2
 3 public int getValue() {
 4   int v = value;
 5   if (v == 0) {
 6     synchronized (this) {
 7       if (value != 0)
 8         return value;
 9       v = computeValue();
10       value = v;
11     }
12   return v;
13 }
14
15 }
```

Jeśli `computeValue()` nie ma skutków ubocznych, to możemy pominąć także blok `synchronized` (metoda ta może zostać wykonana kilka razy, ale przekaże ten sam wynik, dzięki czemu wartość dostarczana przez metodę `getValue()` będzie poprawna).

```
1 public int getValue() {
2   int v = value;
```

```
3     if (v == 0) {
4       v = computeValue();
5       value = v;
6     }
7     return v;
8   }
```

3.6.5. Pobranie obiektu klasy przez stworzenie nowej instancji

Instancje klasy `Class` reprezentują klasy i interfejsy w aplikacji Javy. Mamy kilka możliwości pobrania takiego obiektu:

- Jeśli posiadamy instancję danej klasy, to możemy otrzymać obiekt `Class` odpowiadający tej klasie poprzez wywołanie metody `getClass()` na tym obiekcie.
- Specyfikacja Javy definiuje specjalną konstrukcję `NazwaKlasy.class`, która pozwala nam otrzymać obiekt klasy `Class` nawet wtedy, gdy nie mamy instancji danej klasy.
- Możemy skorzystać ze statycznej metody klasy `Class`: `Class.forName("NazwaKlasy")`.

Zdarza się, że programiści używają konstrukcji

```
1 new String().getClass()
```

czyli tworzą najpierw tymczasowy obiekt danej klasy tylko po to, aby otrzymać obiekt opisujący jego klasę, zamiast zapisać po prostu:

```
1 String.class
```

Jest to konstrukcja, która może zostać wyłapana przez mechanizm analizy statycznej.

Zastąpienie statycznej metody `Class.forName("NazwaKlasy")` konstrukcją `NazwaKlasy.class`, nie jest już takie oczywiste. Nawet pomijając fakt, że pozwala ona na parametryzowanie wywołania dowolnym napisem i wyszukując tylko te wywołania metody, gdzie parametr jest stałą napisową, przy proponowaniu takiego zastąpienia trzeba zachować ostrożność. Takie użycie pozwala nam przechwycić wyjątek zgłoszony przez mechanizm ładujący klasy, w przypadku gdy klasa taka nie istnieje lub nie może zostać załadowana. Jednak jeśli klasa ta jest także zadeklarowana w danym pliku (np. jest wyspecyfikowana w klauzulach import), oznacza to, że klasa zostanie załadowana jeszcze przed wykonaniem tego kodu. Całkiem bezpieczne jest wtedy zastąpienie podanej konstrukcji przez `Class.forName("NazwaKlasy")`.

3.6.6. Używanie += na obiektach `String` w pętli

Obiekty typu `String` są w Javie traktowane w specjalny sposób. Klasa `String` jako jedyna posiada przeciążony operator +, dzięki czemu możliwe są konstrukcje:

```
1 String s1 = computeResult();
2 String s2 = "Wynik: " + s1;
```

Obiekty typu `String` są w Javie traktowane jako niemodyfikowalne. Własność ta oznacza, że użytkownik danej klasy nie może, poprzez wywołania metod, zmienić zachowania danej instancji, przez co wynik przekazywany przez metody danej klasy (w szczególności przez metody `hashcode()` oraz `equals()`) będzie zawsze taki sam. Operacja + na dwóch obiektach klasy String stworzy nową instancję takiego obiektu. Powyższy fragment kodu jest tożsamy z następującym:

```
1 String s1 = computeResult();
2 String s2 = new StringBuffer().append("Wynik: ").append(s1).
      toString();
```

Operacja `toString()` na obiekcie `StringBuffer` tworzy nowy obiekt klasy String, który współdzieli tablicę znaków z obiektem `StringBuffer`. Przy pierwszej modyfikacji obiektu `StringBuffer` już po wykonaniu operacji `toString()` tablica ta zostałaby przekopiowana, aby nie została zmieniona wewnętrzna reprezentacji klasy `String`.

Jak widać konkatenacja napisów przy użyciu operatora + tworzy co najmniej trzy nowe obiekty: `String`, `StringBuffer` oraz tablicę znaków do przechowywania reprezentacji znakowej napisu.

Możemy utworzyć regułę analizującą występowanie w kodzie źródłowym konstrukcji, zawierających konkatenację w pętli, jak np.

```
1 String s = "Wyniki: ";
2 for (int i = 0; i < tablica.length; i++) {
3   s += tablica[i];
4   if (i != tablica.length - 1)
5     s += ", ";
6 }
```

3.6.7. Niepotrzebne tworzenie obiektów String

Twórcy Javy wprowadzili w bibliotece standardowej konstruktor `new String(String)`. Warto dodać – konstruktor całkiem niepotrzebny.

Obiekty klasy `String` są obiektami niemodyfikowalnymi, tzn. nie można zmienić raz utworzonego obiektu. Ilekroć będziemy chcieli zmodyfikować taki obiekt, stworzona zostanie jego druga instancja. W poniższym fragmencie kodu:

```
1 String a = "To jest ";
2 a += "napis";
```

stworzony zostanie nowy obiekt `String` o odpowiedniej długości, który zostanie potem przypisany do zmiennej a. Konsekwencją tego, że napisy są obiektami niemodyfikowalnymi jest to, iż nie potrzebujemy różnych instancji tego obiektu o tych samych własnościach, ponieważ możemy używać wszędzie jednej referencji do niego. Oznacza to, że nie potrzebujemy konstruktora kopiującego dla tych obiektów.

Czasem możemy za to napotkać następującą konstrukcję:

```
1 String a = "Napis1";
2 String b = "Napis2";
3 String c = new String(a + b);
```

lub taką:

```
1 Object o = ...;
2 String d = new String("Wartosc: " + o);
```

W obu przypadkach wystarczą konstrukcje:

```
1 String c = a + b;
2 String d = "Wartosc: " + o;
```

Opakowanie tych wyrażeń w `new String(...)` powoduje stworzenie jeszcze jednej, zupełnie niepotrzebnej instancji obiektu `String`.

Reguła analizy statycznej powinna zabraniać wywoływania konstruktora `new String(String)`.

3.6.8. Używanie konstruktora Boolean

Podobnie wygląda przypadek tworzenia nowych obiektów klasy `Boolean`. Obiekty tej klasy reprezentują wartość logiczną: prawdę lub fałsz. Są to dwie jedyne możliwe wartości reprezentowane przez obiekt tej klasy. Tak jak w obiektach klasy `String`, pól obiektów tej klasy nie można modyfikować, co za tym idzie, nie można zmienić wartości przechowywanej przez obiekt. Wystarczą więc dwa obiekty do reprezentowania wszystkich wartości, jeden reprezentujący prawdę a drugi fałsz.

Takie obiekty są tworzone przy ładowaniu klasy `Boolean` i dostępne jako zmienne statyczne `Boolean.TRUE` oraz `Boolean.FALSE`. Nie ma więc potrzeby tworzenia nowych obiektów klasy `Boolean`. Zamiast pisać:

```
1 Boolean b1 = new Boolean(true);
2 Boolean b2 = new Boolean(false);
3 Boolean b3 = new Boolean("true");
4 Boolean b4 = new Boolean(b1 || b2);
```

lepiej jest napisać:

```
1 Boolean b1 = Boolean.TRUE;
2 Boolean b2 = Boolean.FALSE;
3 Boolean b3 = Boolean.valueOf("true");
4 Boolean b4 = Boolean.valueOf(b1 || b2);
```

Ostatni wiersz sprawia jednak drobny problem. Metoda `Boolean.valueOf(boolean)` została wprowadzona dopiero w bibliotece standardowej Javy w wersji 1.4. Jeśli kod powinien uruchamiać się na starszych wersjach Javy, to trzeba wskazany wiersz zapisać następująco:

```
1 Boolean b4 = (b1 || b2) ? Boolean.TRUE : Boolean.FALSE;
```

Jest bardzo ważne, aby przy implementowaniu reguł analizy statycznej mieć świadomość, że nowe konstrukcje nie zawsze będą poprawne we wszystkich wersjach maszyny wirtualnej Javy czy wersjach biblioteki standardowej. Jeśli to możliwe, dobrze jest ostrzec użytkownika o możliwym braku zgodności wstecz proponowanych konstrukcji.

Reguła analizy statycznej oczywiście powinna proponować skorzystanie z jednej z powyższych konstrukcji zamiast z konstruktorów `Boolean(boolean)` czy `Boolean(String)`.

3.6.9. Tworzenie obiektów opakowujących

Konstrukcją jaką należy wyeliminować z kodu jest niepotrzebne tworzenie nowych obiektów przy konwersji typów prostych lub napisów, np.

```
1 String s = new Integer(i).toString();
2 int j = new Integer(s).intValue();
3 boolean b = new Boolean(s).booleanValue();
```

Takie konwersje można wykonać bez pomocy obiektów opakowujących typy proste:

```
1 String s = Integer.toString(i); // lub String.valueOf(i);
2 int i = Integer.parseInt(s);
3 boolean b = Boolean.valueOf(s);
```

3.6.10. Puste bloki (szczególnie blok catch)

Puste bloki zawarte w kodzie źródłowym często są wynikiem błędu programisty. Błędy takie zostaną jednak zwykle szybko wykryte podczas wykonania programu, wpływają one na semantykę metody, a więc i całej aplikacji, np.

```
1 if (a == 0); {
2   System.out.println("a = 0");
3 }
```

Innym problemem są często spotykane konstrukcje, zwłaszcza u mniej zaawansowanych programistów, w których przechwytywany wyjątek nie jest w ogóle obsługiwany, np.

```
1 try {
2   InputStream stream = new FileInputStream("plik.txt");
3   loadData(stream);
4   stream.close();
5 } catch (FileNotFoundException ex) { };
```

Brak obsługi wyjątku często nie jest świadomym działaniem programisty. Brak jest jakiegokolwiek powiadomienia użytkownika o wystąpieniu sytuacji wyjątkowej. Dobrą praktyką w przypadku wystąpienia błędu, jest odnotowanie tego faktu w systemie rejestrowania błędów (np. korzystając z klas zawartych w pakiecie `java.util.logging` lub bibliotek takich jak Log4J). Jeśli wystąpienie wyjątku jest spodziewaną sytuacją, przy której nie należy podejmować żadnych działań, to dobrze jest umieścić chociaż komentarz, aby osoby czytające kod miały pewność co do intencji autora.

Podczas analizy kodu należy wyszukać wszystkie bloki, które nie zawierają instrukcji i wymagać, aby znalazł się w nich przynajmniej komentarz.

3.7. Unikanie typowych błędów programistycznych

3.7.1. Nieużywany kod

Reguły analizy statycznej wykorzystujące analizę drzewa AST mogą także służyć do wyszukiwania nieużywanego kodu programu. O ile kompilator Javy znajduje wszystkie miejsca do których nie dochodzi przepływ sterowania w blokach, to ignoruje on fakt istnienia metod, zmiennych lokalnych, pól czy parametrów, które nie są używane.

Ogólnie rzecz biorąc znalezienie nieużywanego kodu źródłowego, powinno zostać wykonane z wykorzystaniem narzędzia analizującego graf wywołania metod. Prawidłowe wykrycie kodu, który nie jest wykonywany, jest problematyczne z uwagi na polimorfizm w językach obiektowych. Całą sytuację dodatkowo komplikuje fakt, iż nie wiemy z góry w jakim środowisku zostanie uruchomiona nasza aplikacja – być może będzie wykorzystana po prostu jako biblioteka klas dla innej aplikacji. Założenie takie praktycznie oznacza, że każda metoda publiczna jest traktowana jako kod potencjalnie używany.

Istnieją jednak narzędzia, które starają się sprawdzić jakie fragmenty kodu w jakim stopniu są używane. Narzędzia takie działają zwykle już na etapie uruchomienia programu. Rzadko używane ścieżki przepływu sterowania w aplikacji mogą zostać jednak fałszywie zaklasyfikowane jako kod niewykorzystywany.

Kilka prostych reguł pozwoli nam wykryć te fragmenty kodu źródłowego, co do których możemy mieć pewność, że nie są na ścieżkach przepływu sterowania. Wszelkie pola prywatne oraz metody prywatne, które nie są używane w danej klasie mogą zostać usunięte.

3.7.2. Losowanie liczb

Często zdarza się, że potrzebujemy liczby losowej z pewnego zakresu. Standardowa biblioteka Javy udostępnia klasę `java.util.Random` do generowania liczb pseudolosowych. Jedną z ciekawych praktyk losowania liczb z zadanego zakresu jest konstrukcja typu:

```
1  Math.abs(random.nextInt()) % n
```

Metoda `random.nextInt()` przekazuje liczbę pseudolosową typu `int`, którą następnie rzutujemy na zbiór liczb całkowitych dodatnich modulo n. Niestety metoda ta zawiera błąd, trudny do zauważenia, praktycznie niewykrywalny w jakichkolwiek testach. Przeanalizujmy ją dokładniej.

`Random.nextInt()` przekazuje liczbę typu `int` z zakresu od `Integer.MIN_VALUE` do `Integer.MAX_VALUE`. Funkcja `Math.abs(int)` przekazuje moduł liczby całkowitej. Z małym wyjątkiem. Oto opis metody `Math.abs(int)` zaczerpnięty z dokumentacji JavaDoc:

> Returns the absolute value of an int value. If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned. Note that if the argument is equal to the value of `Integer.MIN_VALUE`, the most negative representable int value, the result is that same value, which is negative.

Jak widać dla wartości `Integer.MIN_VALUE` (czyli -2^{31}) przekazana wartość będzie identyczna z parametrem `i` oraz – co najważniejsze – będzie liczbą ujemną! Operacja modulo na liczbie ujemnej przekaże liczbę niedodatnią! Błąd ten objawiać się będzie średnio raz na ponad 4 miliardy wywołań tej funkcji!

Można uniknąć tego typu problemów używając gotowej funkcji bibliotecznej losującej liczbę z podanego zakresu. W Javie jest to funkcja `Random.nextInt(int)` losująca liczbę z zakresu $[0, n-1]$. Funkcja ta znajduje się w bibliotece Javy począwszy od wersji 1.2.

Poprawny wynik przekazuje także metoda, w której odwrotnie złożymy funkcje:

```
1  Math.abs(random.nextInt() % n)
```

Definiując regułę analizy statycznej należy znaleźć następujące wyrażenia w kodzie:

```
1  Math.abs(random.nextInt()) % n
```

gdzie:

random – obiekt klasy java.util.Random,

n – dowolne wyrażenie typu `int`.

3.7.3. Serializacja singletonów

Jeśli podczas pisania programu konieczne jest wymuszenie, aby konkretna klasa mogła być zinstancjonowana tylko raz, to korzystamy ze wzorca singletonu opisanego dokładnie w książce [7].

Szczególną uwagę należy poświęcić serializacji obiektów. Sama serializacja obiektu następuje wtedy, gdy chcemy zapisać obiekt na dysku czy w bazie danych lub przesłać go do innego systemu. Serializacja singletonów sprawia jednak pewien problem. Ponowne wczytanie takiego obiektu poprzez deserializację stworzy nam nowy obiekt, powodując iż w systemie będą znajdować się dwie instancje tej samej klasy co przeczy istocie singletonu. Aby tego uniknąć należy we wszystkich klasach singletonów, które implementują interfejs `Serializable` dodać metodę `readResolve()`, która powinna przekazać obiekt singletonu. Sygnatura tej metody musi być następująca:

```
1  [dowolny modyfikator] Object readResolve() throws
       ObjectStreamException;
```

Jest to konieczne, jeśli nie chcemy, aby w systemie mogły znaleźć się dwie instancje singletonu. Reguła analizy statycznej powinna zawsze wymagać, aby obiekty będące singletonami implementujące interfejs `Serializable` zawsze miały zaimplementowaną metodę `readResolve()`.

3.7.4. Porównanie napisów bez uwzględniania wielkości liter

Gdy trzeba porównać dwa napisy, tak aby nie uwzględniać wielkości liter, jednym z rozwiązań jest zamiana liter w obu porównywanych napisach na wyłącznie małe lub wyłącznie duże litery, a następnie porównanie takich napisów. W Javie kod wyglądałby mniej więcej w ten sposób:

```
1  napis1.toLowerCase().equals(napis2.toLowerCase())
```

Kod ten wykonany w Javie jest obarczony dwoma wadami:

- Aby wykonać to porównanie stworzone zostaną cztery nowe obiekty: dwa obiekty klasy `String`, każdy z nich zawierający tablicę znaków o długości równej długości odpowiedniego napisu. Obiekty te zostaną skonstruowane niezależnie od wyniku porównania, co oznacza, że każdy ze znaków obu napisów będzie musiał zostać przekodowany na małą literę. Jeśli porównywane napisy często różnią się między sobą (co można by np. od razu wykazać prostym testem porównującym długości napisów), to zmiana wielkości liter wykonuje się zupełnie niepotrzebnie.

- Funkcja `toLowerCase()` bierze pod uwagę ustawienia narodowe (ang. *locale*) zdefiniowane w systemie. W zależności od tych ustawień procedura tłumaczenia znaków na małe litery może przebiegać różnie. Tak dzieje się na przykład dla ustawień tureckich dla litery I. Przy tych ustawieniach małą literą dla I jest małe i bez kropki, a odpowiadającą wielką literą dla i jest wielka litera I z kropką. Test ten może dawać różne wyniki w zależności od ustawień narodowych.

Warunek:

```
1 "APLIKACJA".toLowerCase().equals("aplikacja".toLowerCase())
```

dla ustawień polskich (czy angielskich) przekaże prawdę, a dla ustawień tureckich fałsz. Wykrycie tego typu błędu może okazać się bardzo trudne.

Poprawna konstrukcja takiego warunku to:

```
1 napis1.equalsIgnoreCase(napis2)
```

Porównanie takie nie zależy od ustawień lokalnych, nie tworzy żadnych nowych obiektów, a sam algorytm porównania kończy się z chwilą napotkania pierwszej niezgodności znaków (przy czym przed porównaniem napisów porównywane są ich długości)

Reguła analizy statycznej powinna wyszukiwać w kodzie wszelkich konstrukcji typu:

```
1 obiektString.toLowerCase().equals(obiektString2)
2 obiektString.equals(obiektString2.toLowerCase())
```

3.7.5. Metoda `Boolean.getBoolean()`

Biblioteka standardowa Javy zawiera klasę `Boolean` do operowania na typach `boolean`. Często zachodzi potrzeba przeanalizowania zawartości napisu i przekształceniu jej na wartość typu `boolean`. Nazwa metody statycznej `boolean Boolean.getBoolean(String)` może sugerować właśnie takie zachowanie. Niestety jest to błąd. Zachowanie takie realizuje wyrażenie `Boolean.getBoolean(String).booleanValue()`, zaś pierwsza metoda przekazuje wartość `true`, gdy na zmiennej systemowej o nazwie przekazanej jako parametr znajduje się napis `true`.

Najlepiej jest zabronić w kodzie wywoływania metody `Boolean.getBoolean(String)`. Ustawienie znacznika javadoc o nazwie `@deprecated` pozwoliłoby łatwo wykryć wszelkie takie sytuacje. Ponieważ modyfikacja kodu biblioteki nie jest możliwa, powinna istnieć możliwość ustalenia listy metod niebezpiecznych (mogących powodować błędy).

3.7.6. Słowo kluczowe `return` w bloku `finally`

Trzeba zwrócić szczególną uwagę na kod zawarty w bloku `finally`. Nie powinien on zgłaszać żadnych wyjątków, aby nie powodować utraty informacji o innym, zgłoszonym wcześniej wyjątku. Zachowanie to opisane jest w specyfikacji Javy [8] w rozdziale 14.19.2. Oto przykład takiego kodu:

```
1  public void process(Connection conn, String data) throws
       ProcessingException, DataFormatException {
2    ...
3  }
4
5  public boolean processData(String data) {
6    Connection conn = getConnection();
7    try {
8       process(conn, data);
9       return true;
10   } catch (ProcessingException ex) {
11      Logger.log(ex);
```

```
12        return false;
13     } catch (DataFormatException ex) {
14        Logger.log(ex);
15        throw ex;
16     } finally {
17        if (conn != null)
18          conn.close();
19     }
20  }
```

Metoda `Connection.close()` nie powinna zgłaszać żadnych wyjątków, ponieważ spowoduje to utratę informacji o zgłoszeniu wcześniej innego wyjątku (np. `DataFormatException` lub dowolnego innego wyjątku dziedziczącego po klasie `RuntimeException`).

Groźniejszą sytuacją jest jednak konstrukcja, w której w bloku `finally` znajduje się słowo kluczowe `return`. Jeśli w powyższym przykładzie przeniesiemy wiersz 9 za wiersz 18, tak aby otrzymać:

```
1  public void process(Connection conn, String data) throws
       ProcessingException, DataFormatException {
2     ...
3  }
4
5  public boolean processData(String data) {
6     Connection conn = getConnection();
7     try {
8        process(conn, data);
9     } catch (ProcessingException ex) {
10       Logger.log(ex);
11       return false;
12    } catch (DataFormatException ex) {
13       Logger.log(ex);
14       throw ex;
15    } finally {
16       if (conn != null)
17         conn.close();
18       return true;
19    }
20 }
```

to ani zgłoszenie wyjątku w wierszu 14, ani przekazanie wartości w wierszu 11 nie dotrze nigdy do wywołującej metody. Błąd ten może być trudny do namierzenia, ponieważ zwykle sytuacje, w których zgłaszane są wyjątki występują rzadko – nie zostaną więc wykryte podczas normalnej pracy aplikacji. O ile wyjątki `ProcessingException` i `DataFormatException` z powyższego przykładu zostaną przekazane do systemu rejestrowania zdarzeń, to zgłoszenie jakiegokolwiek innego wyjątku (czyli wszystkich dziedziczących po klasie `RuntimeException`) nie będzie w żaden sposób odnotowane.

3.8. Styl kodu źródłowego

Podczas analizy kodu dobrze jest przyjrzeć się nazewnictwu zmiennych. Brak konsekwencji może powodować problemy przy czytaniu kodu przez nowych członków zespołu czy nawet wprowadzać w błąd. Programując w Javie najczęściej można spotkać się z konwencją zalecaną przez firmę Sun Microsystems opisaną szczegółowo w dokumencie [10]. Porządkuje ona nie

tylko kwestie nazewnictwa, ale także wszelkie sprawy dotyczące wyglądu kodu źródłowego (wcięcia, odstępy, umiejscowienie klamer rozpoczynających i kończących bloki).

Aby kod był łatwiejszy w utrzymaniu, powinien być tak napisany, aby zapoznanie się z nim nie stwarzało problemów.

Jednym z problemów są kwestie związane z nazewnictwem pakietów, klas, metod, pól czy zmiennych lokalnych. Automatyczna kontrola poprawności tych nazw jest niestety trudna. Możemy jednak wykorzystać narzędzia analizy statycznej do zaimplementowania reguł, które sprawdzą np. długości nazw klas lub pól (dobrze jest odrzucić te, które mają jeden lub dwa znaki). Inną regułą może być wymuszenie nazywania klas abstrakcyjnych od słowa `Abstract`.

Dobrze jest ograniczyć też wolność jaką daje Java w wyborze nazw identyfikatorów. Powinno się zabronić używania identyfikatorów zawierających znaki spoza standardowego zestawu znaków (na szczęście niewielu programistów wie, że takie identyfikatory są w Javie prawidłowe). Podobnie identyfikatory zawierające znak dolara powinny być zabronione.

Chciałbym poruszyć jeszcze jedną kwestię dotyczącą nazewnictwa identyfikatorów.

W wielu edytorach czcionka nie pozwala na łatwe odróżnienie identyfikatora l (litery) od 1 (cyfry). Może to prowadzić do pomyłek przy analizowaniu kodu, np.

```
1 long l = 100l; // Przypisanie na l liczby 100 czy 1001?
2 long k = l;    // Przypisanie na k liczby 1 czy wartości
      identyfikatora l?
```

Litera `l` pozwala oznaczyć typ stałej liczbowej jako `long`. Aby czytelnik nie miał problemów z odczytaniem intencji autora kodu, wystarczy wymusić używanie w takich wypadkach dużej litery `L`. Aby uniknąć mylenia identyfikatora `l` z cyfrą `1` należy zabronić używania litery `l` jako identyfikatora oraz identyfikatorów, które składają się tylko z liter `l` oraz cyfr (np. `l1`, `l9`).

3.9. Programowanie rozproszone

Programiści piszący programy mające działać poprawnie w środowiskach wielowątkowych muszą zwrócić szczególną uwagę na sekcje krytyczne przy korzystaniu ze współdzielonych zasobów. Problemy synchronizacji wątków ujawniają nową klasę błędów, które może popełnić programista – możliwość powstawania zakleszczeń lub brak synchronizacji wątków przy dostępie do współdzielonych zasobów. Błędy te są najczęściej trudne do wykrycia, gdyż mogą objawiać się tylko od czasu do czasu lub tylko w szczególnych warunkach pracy programu, np. przy dużym obciążeniu komputera. Mogą objawiać się także tylko na wybranych architekturach, przez co przeprowadzenie dobrych testów w czasie wykonania jest bardzo trudne. Moduł integrujący narzędzie JLint z edytorem wprowadza wiele reguł analizy statycznej dotyczących programowania rozproszonego do platformy Eclipse.

Rozdział 4

Projekt i implementacja modułu

4.1. Cel i kontekst

Podstawowym celem stawianym narzędziom wspomagającym tworzenie oprogramowania jest zwiększenie wydajności pracy programisty przy jednoczesnym zminimalizowaniu prawdopodobieństwa błędu. Do takich narzędzi należą:

- oprogramowanie umożliwiające generowanie kodu na podstawie projektu,
- narzędzia umożliwiające konstruowanie interfejsu użytkownika z wykorzystaniem technologii przeciągnij i upuść,
- aplikacje automatyzujące proces testowania,
- oprogramowanie wspomagające proces zarządzania znalezionymi w tworzonej aplikacji błędami,
- oprogramowanie analizujące kod źródłowy w poszukiwaniu błędów.

Celem niniejszej pracy magisterskiej było stworzenie narzędzia kwalifikującego się do ostatniej z wymienionych grup integrującego się ze środowiskiem Eclipse.

4.2. Założenia wstępne

Jednym z podstawowych elementów Platformy Eclipse jest zbiór wtyczek wspierających pracę przy projektach w języku Java, czyli Java Developer Toolkit. Środowisko to zapewnia narzędzia do edycji kodu, jego kompilacji, testowania, śledzenia oraz liczne mechanizmy wspomagające pracę programisty. Do podstawowych narzędzi należą widoki umożliwiające spojrzenie na projekt z różnych perspektyw. Oferowanymi rozwiązaniami stanowiącymi podstawowe wsparcie dla programisty są widoki prezentujące strukturę tworzonego projektu, hierarchię klas, strukturę aktualnie edytowanego pliku, informację o błędach, hierarchię wywołania metod czy listę zaplanowanych do wykonania zadań. Dodatkowo istnieją mechanizmy podpowiedzi, sugerujące na podstawie kontekstu możliwe rozwinięcie kodu (np. poprzez wyświetlenie listy metod danej klasy) oraz poprawiania błędów, sygnalizujące miejsca w kodzie źródłowym, gdzie wystąpił błąd kompilacji oraz proponujące listę czynności jakie mogą prowadzić do usunięcia błędu.

Jednym z podstawowych założeń projektu realizowanego w ramach niniejszej pracy była jego całkowita „przezroczystość” – dodany moduł powinien stanowić integralną część platformy nie różniącą się interfejsem użytkownika od pozostałych części. Innym założeniem było

wykorzystanie istniejących mechanizmów oznaczania błędów oraz podpowiadania właściwych rozwiązań.

Dostępne zasoby jakie mogły zostać wykorzystane w ramach projektu powinny zostać ograniczone do podstawowych wtyczek z jakimi rozpowszechniana jest dystrybucja Eclipse.

4.3. Wymagania

Oto zbiór wymagań, jakie powinien spełniać system wspomagający automatyczny audyt kodu.

Prosta adaptacja istniejących rozwiązań

Powinna być możliwość integracji programów do analizy statycznej, zawierających gotowe zbiory reguł. Pozwoli to na podłączenie ich do środowiska programistycznego oraz zintegrowanie wielu aplikacji do analizy statycznej w jednym spójnym interfejsie użytkownika.

API do tworzenie nowych reguł

Program powinien posiadać prosty i elastyczny interfejs programistyczny do tworzenia nowych reguł. Programista tworzący regułę powinien skupić się tylko na określeniu czy reguła została naruszona. Jak się okaże większość przedstawionych wcześniej reguł można zapisać bardzo łatwo przy użyciu szablonu reguł korzystającego z widoku drzewa AST kodu źródłowego zawierającego informację o połączeniach z kodem źródłowym innych klas (a w zasadzie z ich drzewami AST). API powinno wykorzystywać dostępne elementy platformy, tak by nie kopiować istniejącej funkcjonalności.

Możliwość definiowania sposobu poprawiania kodu dla konkretnych naruszeń reguł

System powinien udostępniać mechanizm definiowania reguł audytu oraz sposobów rozwiązywania naruszeń tych reguł. Mechanizm tworzenia reguł poprawy kodu powinien być niezależny od sposobu definiowania samych reguł. Wynika to z faktu, iż sposób znalezienia błędu w kodzie i sposób jego naprawy będą definiowane przez dwa zwykle zupełnie różne, niezależne od siebie fragmenty kodu. Kod naprawy danego błędu można napisać przy wykorzystaniu zupełnie innych narzędzi niż przy jego znajdowaniu. Ponadto wiele narzędzi do analizy kodu pozwala jedynie na wyszukiwanie błędów w programie, nie definiuje zaś metod poprawy kodu. Programista sam mógłby napisać reguły poprawiania kodu także dla reguł z tych zewnętrznych systemów.

Spójność interfejsu użytkownika z platformą Eclipse

Program powinien integrować się z platformą Eclipse także na poziomie interfejsu użytkownika. Należy zachować jednakowy sposób interakcji z użytkownikiem, informowania go o wykrytych błędach. Zachowanie programu powinno być analogiczne do zachowania modułu JDT. Należy wykorzystać istniejące mechanizmy oznaczania błędów, propozycji ich rozwiązania oraz konfiguracji.

Niezależność reguł od języka programowania

System nie powinien robić założeń co do poziomu abstrakcji kodu źródłowego. Powinna istnieć możliwość tworzenia reguł zarówno dla konkretnego języka programowania, jak i dla klasy języków.

Skalowalność

System powinien być w stanie obsłużyć dowolną liczbę reguł, mieć otwartą architekturę na podczepianie innych systemów lub tworzenie nowych, pojedynczych reguł.

4.4. Istniejące punkty rozszerzań

W niniejszym rozdziale prezentuję punkty rozszerzeń jakie zostały wykorzystane w aplikacji. Opisane zostały elementy, których rozszerzenie pozwoliło na zaimplementowanie założonej funkcjonalności. W kolejnych sekcjach opisane zostały elementy platformy Eclipse odpowiedzialne za tworzenie widoków, mechanizmy oznaczania błędów, mechanizmy przyrostowego budowania projektu i wiele innych, które zostały rozszerzone przez moduł statycznej analizy kodu. Opis pojedynczego punktu rozszerzenia składa się z odpowiedniego fragmentu pliku deklaracji, opisu funkcji za jakie jest on odpowiedzialny oraz informacji o wykorzystaniu go w tworzonym projekcie wspierającym proces analizy statycznej kodu.

4.4.1. org.eclipse.ui.views

```
1  <extension
2          point="org.eclipse.ui.views">
3      <category
4            name="Audytor kodu"
5            id="auditor.mainCategory">
6      </category>
7      <view
8            name="Naruszenia reguł"
9            icon="icons/sample.gif"
10           category="auditor.mainCategory"
11           class="auditor.views.ProblemsListView"
12           id="auditor.views.ProblemsListView">
13     </view>
14  </extension>
```

Punkt ten odpowiedzialny jest za tworzenie nowych widoków oraz umieszczanie ich w hierarchicznej strukturze. Stworzona została nowa kategoria widoków (widoki związane z analizą kodu) oraz dołączony został pierwszy widok, który wyświetla listę wszystkich zarejestrowanych reguł do analizy kodu źródłowego wraz z ich identyfikatorami.

4.4.2. org.eclipse.core.resources.markers

```
1  <extension
2      id="violationmarker"
3      name="Naruszenie reguly"
4      point="org.eclipse.core.resources.markers">
5   <super
6         type="org.eclipse.core.resources.problemmarker">
7   </super>
8   <attribute
9         name="ruleId">
10  </attribute>
11  <persistent value="true"/>
```

```
12 </extension>
```

W punkcie tym możemy definiować nowe znaczniki, którymi możemy następnie oznaczać kod (por. p. 2.1.2). Zdefiniowany został znacznik o identyfikatorze `auditor.violationmarker`, dziedziczący po `org.eclipse.core.resources.problemmarker`. `Problemmaker` jest rodziną znaczników, które zawierają atrybuty pomocne przy ustawianiu miejsc występowania błędów w kodzie źródłowym.

Jednym ze standardowych widoków w środowisku Eclipse jest widok Problems zawierający listę wszelkich błędów w projekcie. Standardowo rozszerzenia, takie jak JDT czy PDE, umieszczają tam wszelkie komunikaty związane z błędami kompilacji, ostrzeżenia wygenerowane przez kompilator czy informacje o problemach z walidacją pliku XML-owego będącego opisem wtyczki. Aby umieścić informację o błędzie w kodzie źródłowym, nie trzeba jednak znać szczegółów implementacji widoku `Problems`. Widok ten sam pobiera informacje o wszystkich znacznikach typu `org.eclipse.core.resources.problemmarker` oraz odpowiednio je prezentuje. Aby umieścić tam również własne komunikaty o błędach, należy oznaczyć kod znacznikami `problemmarker` lub innymi zawierającymi jako jedną z nadklas właśnie ten znacznik.

Takie użycie znaczników jest tylko jednym z możliwych rozwiązań. Sam mechanizm jest bardzo elastyczny. Eclipse wykorzystuje go do wielu różnych celów, takich jak oznaczanie pułapek w kodzie źródłowym (Breakpoints), zakładki (Bookmarks), czy możliwość definiowania przez programistę miejsc w kodzie, które nie zostały jeszcze zaimplementowane (TODO List).

Eclipse sam dba o poprawność znaczników. Jakiekolwiek modyfikacje w kodzie źródłowym, takie jak dopisanie lub usunięcie fragmentu kodu, automatycznie zmieniają położenie znacznika aby odpowiadał danemu miejscu w pliku. Istnieje też możliwość zdefiniowania znaczników jako trwałych (ang. *persistent*) dzięki czemu są one automatycznie zapisywane przy zamykaniu oraz odtwarzane przy starcie środowiska.

Zdefiniowany nowy znacznik (`auditor.violationmarker`) zawiera nie tylko informację o miejscu i rodzaju błędu, ale również referencję do identyfikatora reguły, która znalazła dane naruszenie. Dzięki temu mechanizm automatycznego usuwania błędów może łatwo otrzymać informację o kategorii błędu.

4.4.3. org.eclipse.ui.ide.markerResolution

```
1    <extension
2          point="org.eclipse.ui.ide.markerResolution">
3      <markerResolutionGenerator
4               class="auditor.AuditViolationMarkerResolutionGenerator"
5               markerType="auditor.violationmarker"/>
6  </extension>
```

`markerResolution` jest punktem, w którym możemy dołączyć klasę, która będzie implementowała mechanizm rozwiązywania problemów związanych ze znacznikami. Do każdego ze znaczników możemy dołączyć klasę (lub ich dowolną liczbę) implementującą interfejs `IMarkerResolutionGenerator`. Interfejs ten zawiera jedną metodę `getResolutions(IMarker)`, która powinna przekazać tablicę obiektów `IMarkerResolution`.

Warto zwrócić uwagę w jaki sposób autorzy Eclipse radzą sobie z wsteczną zgodnością interfejsów. W wersji 2.0 środowiska Eclipse stworzony został wspomniany wyżej interfejs `IMarkerResolutionGenerator`. Niestety interfejs ten okazał się zbyt ograniczony. Zawiera on tylko metodę do pobierania tablicy elementów, zaś ze względów wydajnościowych okazało

się, że przydatna byłaby metoda, która jedynie sprawdza czy dana klasa potrafi wygenerować obiekty `IMarkerResolution` dla danego znacznika. Rozszerzenie interfejsu o nową metodę spowodowałoby poważne problemy ze zgodnością już istniejących aplikacji. Dlatego w wersji 2.1 autorzy Eclipse stworzyli nowy interfejs o nazwie `IMarkerResolutionGenerator2`, który dziedziczy po `IMarkerResolutionGenerator` oraz zawiera nową metodę. Niestety, takie podejście wymusza potem testowanie klasy operatorem `instanceof`, ale jest konieczne jeśli chcemy zachować zgodność z już napisanym aplikacjami.

Punkty rozszerzeń często wymagają, aby podać nazwę klasy implementującej pewien interfejs jako jeden ze swoich parametrów. Wraz z nim dostępna jest klasa abstrakcyjna, która dany interfejs implementuje. W takich miejscach możemy rozszerzyć interfejs bazowy o nowe metody bez utraty zgodności o ile wszystkie implementacje dziedziczą po wskazanej klasie abstrakcyjnej.

Obiekty `IMarkerResolution` zawierają (oprócz nazwy) metodę `run(IMarker)`, która powinna podjąć odpowiednie działania w celu usunięcia przyczyny powstania znacznika.

Wykorzystuję ten punkt rozszerzeń, aby podczepić pod znaczniki typu `auditor.violationmarker` (znaczniki te wskazują na te miejsca w kodzie programu, w których wystąpiły naruszenia reguł) metody ich usuwania. Udostępniony został punkt rozszerzenia o nazwie auditor.resolutions, który można rozszerzać o sposoby usuwania naruszeń reguł w kodzie źródłowym.

4.4.4. org.eclipse.ui.preferencePages

```
1  <extension
2        point="org.eclipse.ui.preferencePages">
3     <page
4           class="auditor.preferences.JLintPreferencePage"
5           category="auditor.preferences.main"
6           name="Ustawienia JLint"
7           id="auditor.preferences.JLintPreferencePage"/>
8     <page
9           class="auditor.preferences.AuditorPreferencePage"
10          name="Audytor"
11          id="auditor.preferences.main"/>
12 </extension>
```

Pulpit w środowisku Eclipse (por. p. 2.1.3) dostarcza jedno wspólne dla wszystkich rozszerzeń okno dialogowe dla wszelkich ustawień. Rozszerzenie tego punktu pozwala na stworzenie nowych stron w oknie ustawień.

Strony ustawień są pogrupowane w hierarchiczną strukturę. To rozszerzenie definiuje kategorię o identyfikatorze `auditor.preferences.main`, pod którą zbiory reguł mogą umieszczać swoje strony z ustawieniami. Wtyczka integrująca oprogramowanie JLint z platformą Eclipse właśnie w ten sposób definiuje własną stronę z ustawieniami.

4.4.5. org.eclipse.core.resources.builders

```
1  <extension
2        id="builder"
3        name="Auditor Builder"
4        point="org.eclipse.core.resources.builders">
5     <builder>
```

```
6          <run class="auditor.AuditorBuilder"/>
7       </builder>
8    </extension>
```

Ten punkt rozszerzenia pozwala na zdefiniowanie mechanizmu przyrostowego budowania projektu na platformie Eclipse. Mechanizm ten otrzyma na wejście zbiór zmian jakie dokonały się w zasobach zdefiniowanych w obrębie projektu. Na podstawie tej informacji może następnie ponownie skompilować pliki, w których pojawiły się zmiany.

Ważne jest, aby zdefiniowany mechanizm budowania wykonywał się możliwie szybko. Czas wykonania jednego przebiegu powinien zależeć od liczby zmian w zasobach, nie zaś od całkowitej liczby zasobów w projekcie.

Zdefiniowany tu mechanizm budowania ponownie uruchamia wszystkie aktywne reguły audytu na zasobach, które się zmieniły. Uruchomienie tych reguł odbywa się automatycznie, jej autor nie musi implementować tej funkcjonalności.

4.4.6. org.eclipse.core.resources.natures

```
1    <extension
2            id="nature"
3            name="Auditor Nature"
4            point="org.eclipse.core.resources.natures">
5       <runtime>
6          <run class="auditor.AuditorNature"/>
7       </runtime>
8    </extension>
```

Eclipse wprowadza pojęcie wcieleń projektu (por. p. 2.1.2). Wcielenie pozwala na wprowadzenie faz aktywacji i dezaktywacji dla mechanizmów zdefiniowanych dla projektów. Wcielenia mogą również zawierać po prostu zbiór czynności jakie należy wykonać, aby dodać lub usunąć funkcjonalność dla danego projektu.

Ten punkt rozszerzenia pozwala także na zdefiniowanie mechanizmów budowania jakie powinny być aktywowane i dezaktywowane na projekcie wraz z nadaniem mu odpowiedniego wcielenia.

Przedstawione rozszerzenie definiuje wcielenie o identyfikatorze `auditor.nature`. Wraz z nadaniem projektowi tego wcielenia (np. poprzez wykonanie polecenia z menu kontekstowego dla zasobu projektu) aktywowany jest mechanizm przyrostowego budowania projektu uruchamiający reguły audytu kodu.

4.4.7. org.eclipse.ui.popupMenus

```
1    <extension
2            point="org.eclipse.ui.popupMenus">
3       <objectContribution
4               objectClass="org.eclipse.core.resources.IProject"
5               adaptable="true"
6               id="auditor.actions.convert">
7          <action
8                  label="Włącz/wyłącz audyt"
9                  class="auditor.actions.ConversionAction"
10                 enablesFor="1"
11                 id="auditor.actions.convert">
12         </action>
```

```
13      </objectContribution>
14   </extension>
```

Ten punkt rozszerzenia jest wykorzystywany do dodawania nowych poleceń w menu kontekstowym. Polecenia mogą być przyporządkowane do konkretnego typu obiektu (tak jak powyżej zostało to zdefiniowane dla obiektów projektu – `org.eclipse.core.resources.IProject`) poprzez element `objectContribution` lub może dotyczyć konkretnego widoku czy edytora (element `viewerContribution`).

Polecenie zdefiniowane podaną deklaracją nadaje bądź odbiera wcielenie o identyfikatorze `auditor.nature`. Kod odpowiedzialny za tą czynność znajduje się w klasie `auditor.actions.ConversionAction`.

4.5. Nowe punkty rozszerzeń

Aplikacja definiuje nowe punkty rozszerzeń. Oferują one wysokopoziomowy interfejs programistyczny do tworzenia nowych reguł. W tym rozdziale załączam opis każdego z nich wraz z fragmentem pliku deklaracji opisującego dany punkt.

4.5.1. Reguły audytu

```
1 <extension-point id="rules" name="Auditor Rules" schema="schema/
      auditor.rules.exsd"/>
```

Punkt ten pozwala na zdefiniowanie nowych reguł audytu, które zostaną wykonane na kodzie źródłowym. Implementacja reguły zawarta jest w klasie, która powinna implementować interfejs `auditor.skel.AuditorRule`. Wraz z modułem dostarczone są gotowe klasy, na których można oprzeć własną implementację reguły.

Metody interfejsu `auditor.skel.AuditorRule` to:

- `init(AnalyzerState)`

  Metoda ta wywoływana jest przy inicjowaniu mechanizmu audytu. Reguła powinna zainicjalizować wszelkie swoje struktury oraz może umieścić w obiekcie `AnalyzerState` trwałe obiekty specyficzne dla danej reguły.

- `isValidFor(AnalyzerState, AnalyzerObject)`

  Metoda wywoływana przed uruchomieniem reguły. Reguła na podstawie przekazanych parametrów powinna podjąć decyzję czy potrafi zanalizować podany obiekt. Test ten powinien być możliwie prosty. Typowa implementacja polega na sprawdzeniu typu pliku lub – jeśli wykorzystywany jest mechanizm widoków – wynik testu zależy od tego czy możliwe jest skonstruowanie konkretnego widoku dla danego pliku.

- `analyze(AnalyzerState, AnalyzerObject)`

  Metoda analizująca obiekt. Może skorzystać z mechanizmu widoków – dostępne są one poprzez obiekt `AnalyzerState`. Każde naruszenie reguły powinno być zasygnalizowane poprzez stworzenie obiektu klasy `AuditorViolation` zawierającego informacje o błędzie i miejscu jego wystąpienia. Obiekt ten powinien zostać przekazany do metody `addViolation` obiektu `AnalyzerState`.

4.5.2. Procedury usuwania naruszeń reguł

```
1 <extension-point id="resolutions" name="Auditor Resolutions"
      schema="schema/resolutions.exsd"/>
```

Rozszerzenia tego punktu pozwalają na wykonanie procedur usuwania błędów w zależności od reguł jakie zostały naruszone.

Każde z rozszerzeń tego punktu powinno zawierać następujące informacje:

- Jednowierszowy opis działań jakie zostaną podjęte.
- Dłuższy opis działań jakie zostaną podjęte (informacja ta będzie zaprezentowana użytkownikowi).
- Identyfikator reguły, której naruszenie mogą zostać naprawione przez daną procedurę.
- Nazwa klasy zawierającej implementację procedur usuwania błędu.

Klasa zawierająca procedurę usuwania błędu dostarcza m. in. metodę `repair`, która jest odpowiedzialna za zmianę stanu analizowanego obiektu tak, aby usunąć przyczynę błędu.

Reguły mogą mieć dowolną liczbę procedur usuwających znalezione przez nie błędy (mogą też nie mieć ich wcale – mechanizm analizy kodu będzie potrafił zidentyfikować naruszenia reguły w kodzie źródłowym, nie będzie zaś metody ich automatycznej poprawy).

4.5.3. Widoki kodu źródłowego

```
1  <extension-point id="documentProvider" name="Auditor Document
       Provider" schema="schema/documentProvider.exsd"/>
```

Punkt ten pozwala na zdefiniowanie dodatkowego widoku na pliki z kodem źródłowym. W tym punkcie rozszerzenia należy zarejestrować nowe fabryki (ang. *factory*) dostarczycieli dokumentów. Mechanizm widoków opisany zostanie w następnym rozdziale.

4.6. Widoki

Obecnie istnieje kilka systemów do analizy statycznej kodu. Aby zintegrować je ze środowiskiem Eclipse będziemy potrzebować elastycznej architektury, która pozwoli łatwo dołączać inne systemy.

Systemy z jakimi możemy się spotkać mogą zawierać różne funkcje. Równie dobrze może to być słownik wyszukujący błędy ortograficzne w pliku tekstowym lub komentarzach dowolnego języka programowania, jak i zbiór reguł specyficznych dla języków obiektowych. Najczęściej będzie to jednak zbiór reguł dla pewnego konkretnego języka programowania.

W tym wypadku musimy uwzględnić konieczność obsługi różnych modeli tego samego dokumentu (tzw. widoków). Architektura rozszerzenia powinna umożliwiać korzystanie z różnych widoków przez tę samą wtyczkę. Przykładowe widoki na ten sam dokument:

1. Widok AST – Abstract Syntax Tree (dla konkretnego języka programowania).
2. Widok standardowego wyjścia pliku z kodem źródłowym po przetworzeniu przez aplikację (dowolną).

3. Widok języka programowania jako dokumentu XML (aby reguła mogła wyszukiwać elementy przy użyciu języka Xpath).

4. Widok tekstowy dokumentu, w którym mamy dostęp do poszczególnych wierszy programu.

5. Widok filtrujący tylko niektóre elementy kodu np. dający łatwy dostęp do komentarzy w kodzie źródłowym, niezależny od języka programowania.

Reguły audytu mogą posługiwać się dowolnym z tych widoków, mogą też działać tylko dla plików, dla których uda się zbudować odpowiedni dokument (np. w pliku z kodem źródłowym z poważnymi błędami składniowymi może nie udać się zbudowanie drzewa AST). Reguła może też korzystać z dowolnej liczby widoków przy podejmowaniu decyzji o jej naruszeniu.

Oczywiście to tylko przykłady widoków. Programista może stworzyć dowolny inny widok (poprzez rozszerzenie punktu o nazwie `auditor.document`), który może być następnie wykorzystany w regułach.

Widoki umożliwiają także tworzenie reguł niezależnych od języka programowania. Przykładowo, stworzeniem drzewa AST dla języków C oraz Java zajmowały się będą dwie zupełnie różne klasy. Będą to dwa różne widoki (czyli rozszerzenia punktu `auditor.document`), choć oba będą tworzyły takie same drzewa AST. Oba te widoki powinny występować pod tą samą upublicznioną nazwą. Reguła identyfikuje widok na podstawie jego nazwy, nie ma jednak dla niej znaczenia jaką implementację otrzyma. Mechanizm wyszukiwania widoków przekaże implementację specyficzną dla danego języka, co umożliwia stworzenie reguł obejmujących wiele języków programowania. Praktycznie wszystkie reguły obliczające statystyki dla kodu źródłowego (np. reguła, by żadna metoda nie była dłuższa niż 50 wierszy) mogą korzystać z takich widoków.

4.7. Reguły

Do implementacji poszczególnych reguł zdefiniowany został nowy punkt rozszerzenia o nazwie `auditor.rules`. Reguły powinny definiować warunki jakie muszą być spełnione dla danego pliku z kodem źródłowym, aby było możliwe zastosowanie dla niego danej reguły. Architektura nakłada niewiele ograniczeń na reguły. Przede wszystkim uruchomienie danej reguły powinno być w miarę szybkie dla danego pliku źródłowego. Choć architektura systemu uruchamiającego poszczególne reguły będzie minimalizować liczbę obiektów na jakich będzie uruchomiony zestaw reguł, to mimo wszystko uruchamianie reguł powinno trwać krótko. Jeśli dla zbioru reguł należy stworzyć pewne pomocnicze struktury danych, to należy to zrobić tylko raz i umieścić ją w *kontekście audytu* – obiekcie, w którym możliwe jest umieszczanie obiektów:

- na czas uruchomienia jednej reguły,
- na czas uruchomienia kompletu reguł dla danego zasobu (pliku),
- na czas uruchomienia kompletu reguł dla danego projektu.

Innym sposobem jest wydzielenie tych struktur jako oddzielnego widoku na dany dokument (kod źródłowy) i udostępnienie go na zewnątrz wtyczki danej reguły (czyli zaimplementowanie punktu rozszerzenia `auditor.documentProvider`).

Na rysunku 4.1 uchwycony został edytor z przykładową regułą.

4.8. Interfejs użytkownika

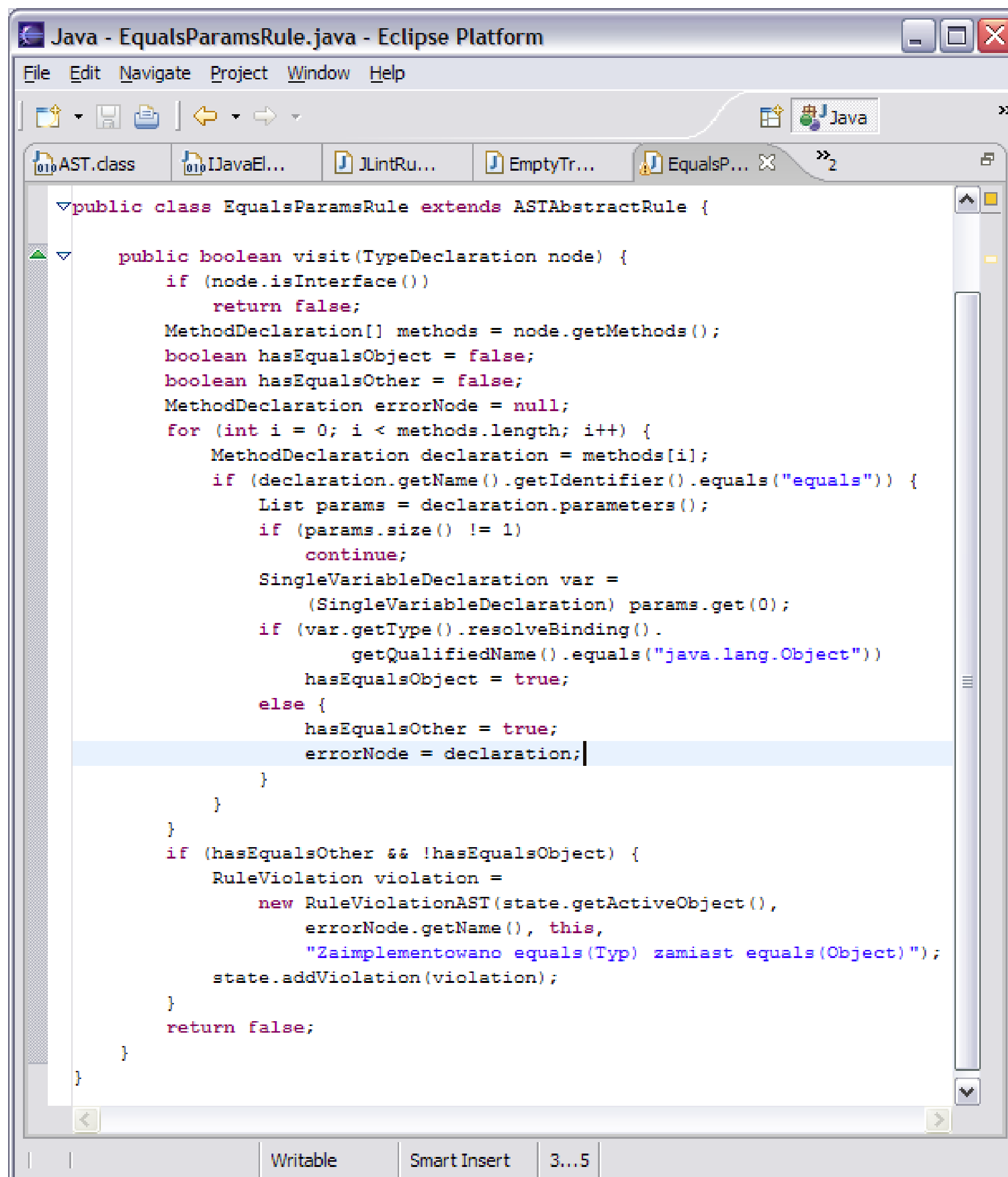
Po zainstalowaniu rozszerzeń w środowisku Eclipse należy uaktywnić mechanizm analizy statycznej poprzez wybór odpowiedniej opcji z menu kontekstowego dla danego projektu (por. rys. 4.2).

Wybranie tej opcji spowoduje włączenie mechanizmu przyrostowej analizy kodu (por. rys. 4.3). Za każdym razem, gdy zmieni się zasób, zostaną wykonane dla niego zarejestrowane reguły analizy statycznej. Przy odświeżeniu zawartości projektu lub przy całkowitym przebudowaniu projektu, mechanizm ten zostanie uruchomiony dla wszystkich zasobów. Z uwagi na to, że taka analiza może trwać wtedy dość długo, zostanie ona wykonana w tle, dzięki czemu użytkownik będzie mógł bez przeszkód pracować. Mechanizm tego automatycznego audytu kodu możemy w każdej chwili wyłączyć poprzez dezaktywowanie odpowiedniego wpisu we właściwościach projektu.

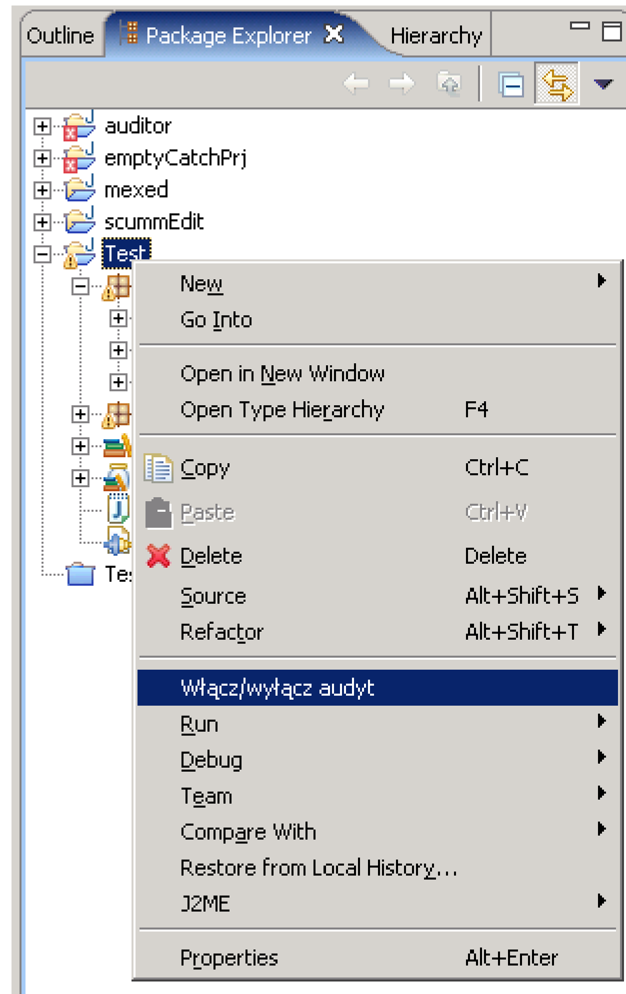
W każdej chwili użytkownik ma dostęp do listy aktywnych reguł, jakie zostaną uruchomione w danym projekcie (por. rys. 4.4). Lista ta jest prezentowana w nowym widoku o nazwie *Naruszenia reguł*. W oknie tym możemy też ręcznie wymusić uruchomienie mechanizmu testowania poprawności kodu. Funkcja ta jest szczególnie przydatna wtedy, gdy nie korzystamy z mechanizmu przyrostowej analizy kodu.

Część reguł może umożliwiać zmianę parametrów analizy. Wszelkie ustawienia można odnaleźć w oknie *Preferences*, gdzie znajdują się wszystkie ustawienia dotyczące reguł (por. rys. 4.5). Każda z reguł może definiować własne ustawienia.

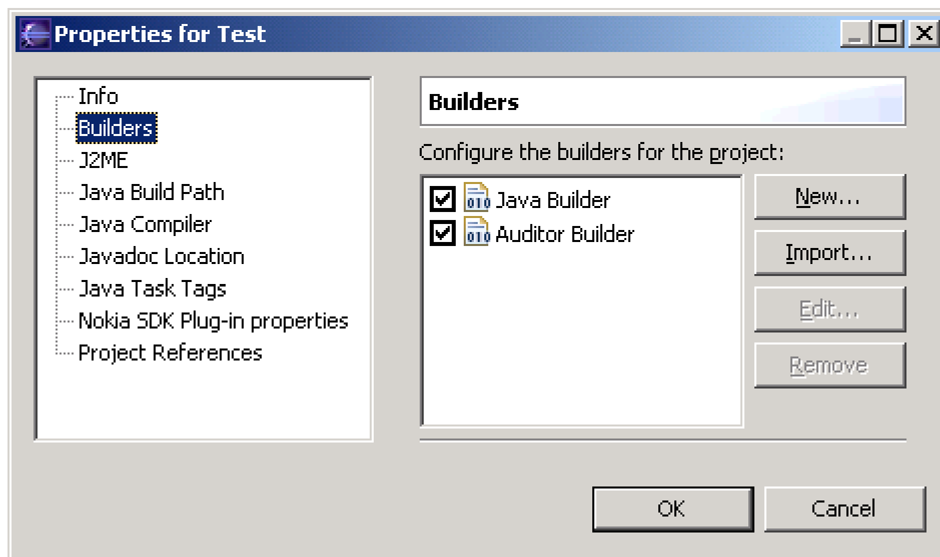
Gdy zakończymy konfigurację poszczególnych reguł, możemy rozpocząć pracę. Analizator umieści informacje o naruszeniach reguł w widoku *Problems*, obok innych komunikatów informujących o błędach w kodzie źródłowym (por. rys. 4.6 oraz 4.7). Ponadto, każde naruszenie reguły będzie widoczne w edytorze kodu w sposób właściwy dla środowiska Eclipse czyli poprzez pojawienie się w miejscu ewentualnego błędu znaku wykrzyknika oraz zaznaczenie fragmentu kodu, którego dotyczy naruszenie. Kliknięcie w oknie *Problems* na jedno ze zgłoszonych naruszeń reguł powoduje wyświetlenie proponowanych przez aplikację rozwiązań zgłoszonego problemu, oczywiście o ile wtyczka z rozwiązaniem tego problemu została zainstalowana.



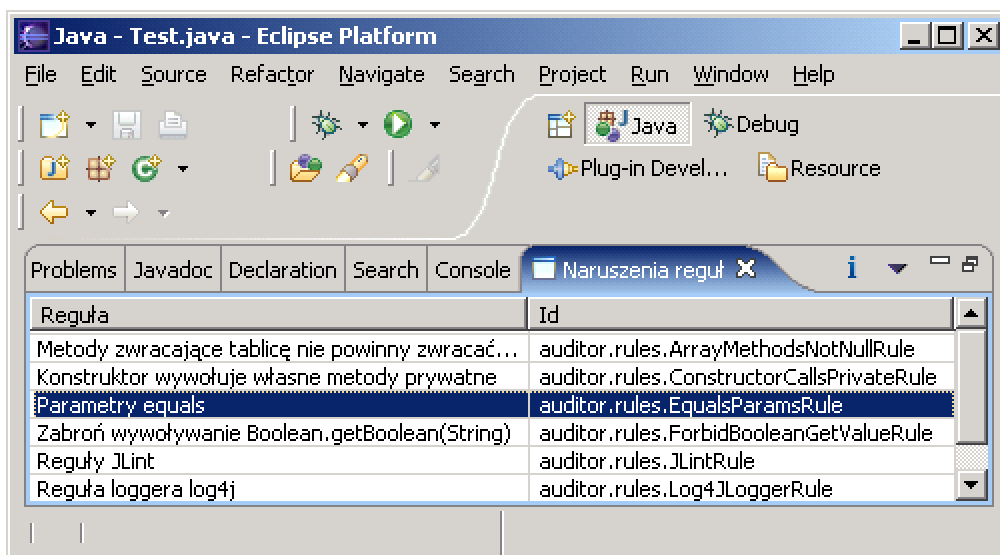
Rysunek 4.1: Kod źródłowy reguły `EqualsParamsRule`



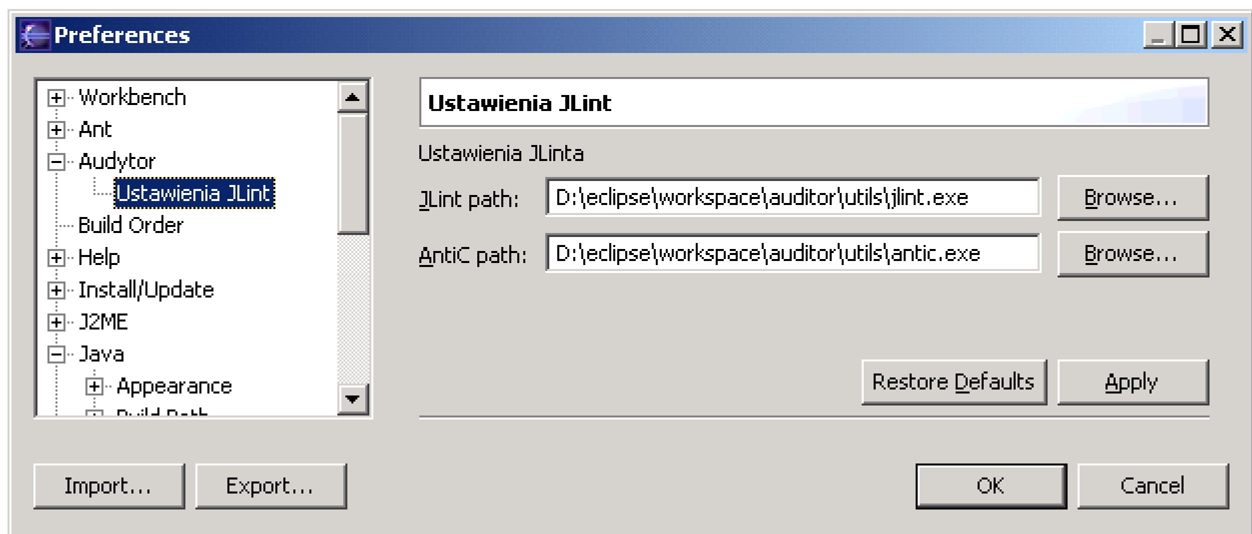
Rysunek 4.2: Menu kontekstowe z opcją uaktywnienia analizy kodu



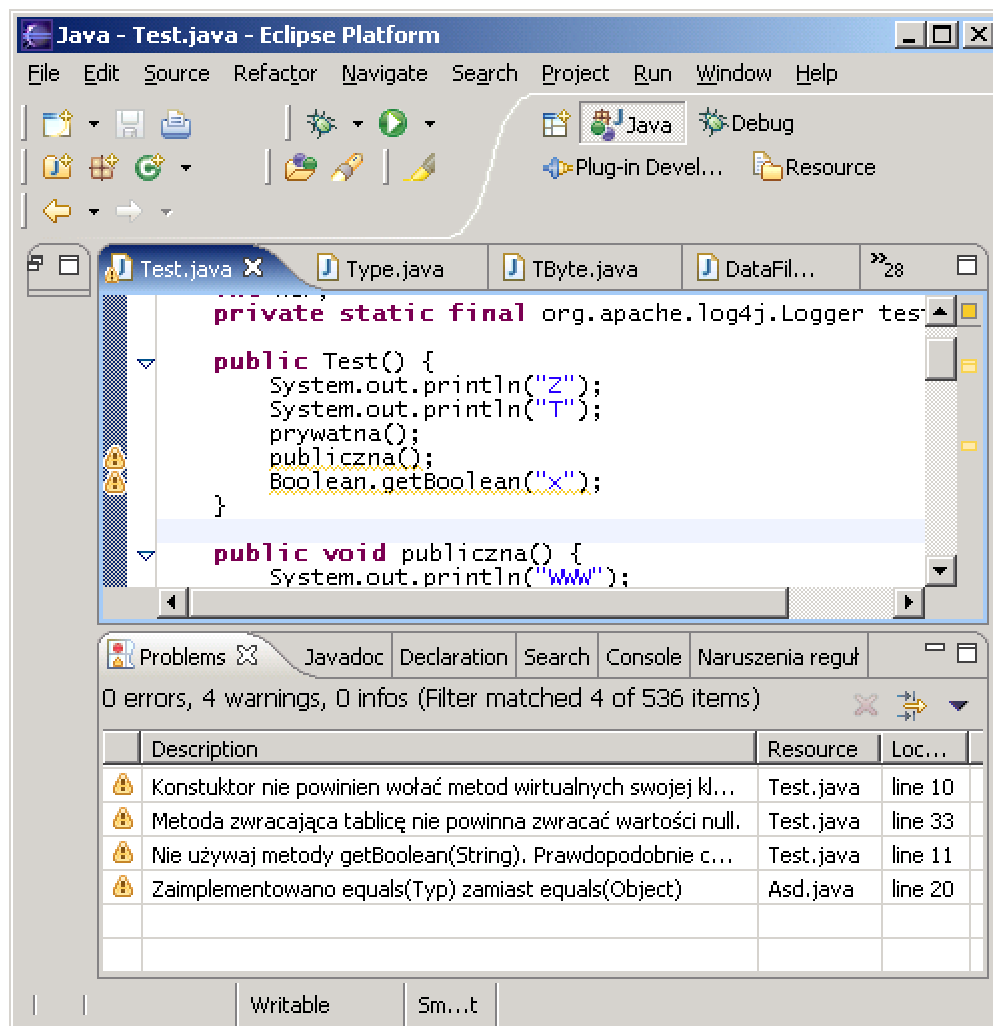
Rysunek 4.3: Lista mechanizmów przyrostowego budowania projektu



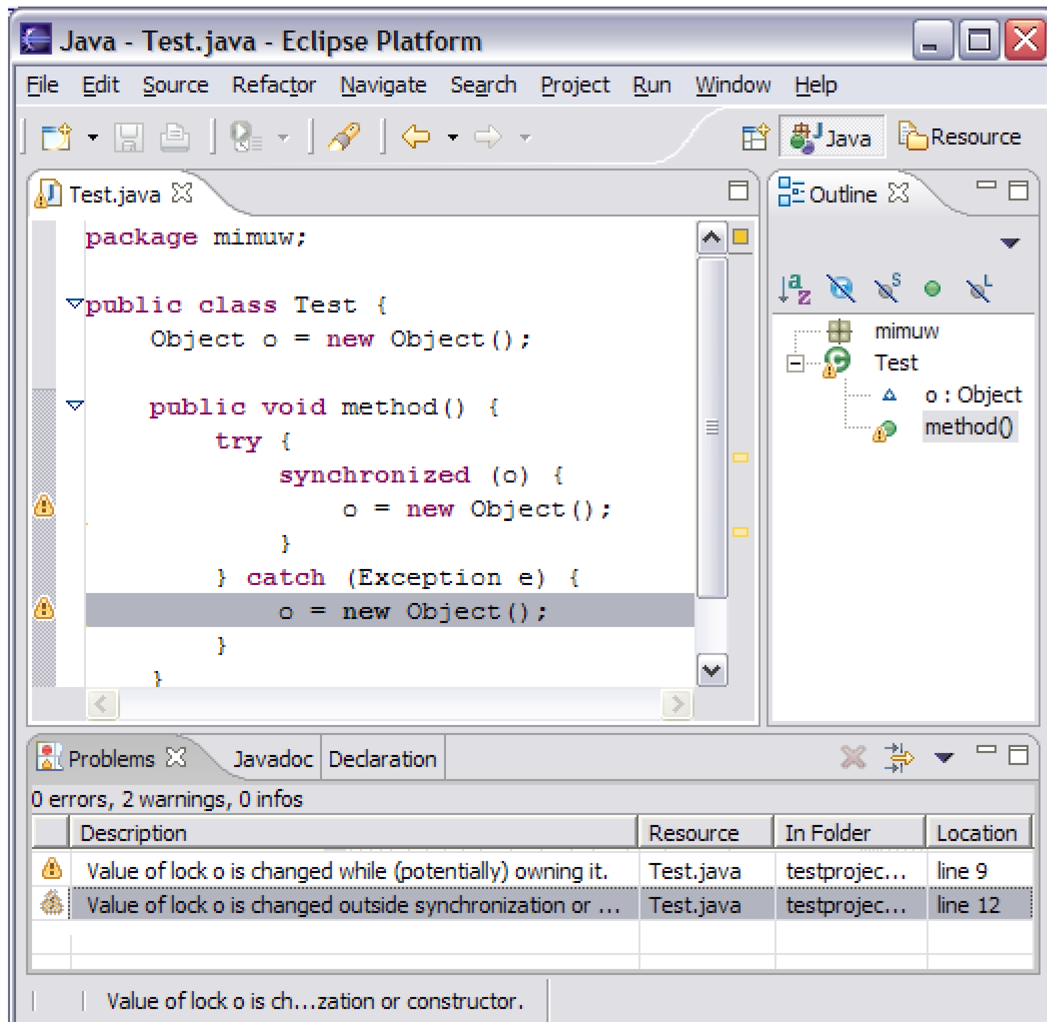
Rysunek 4.4: Lista zainstalowanych reguł



Rysunek 4.5: Ustawienia reguł



Rysunek 4.6: Widok edytora z zaznaczonymi naruszeniami reguł analizy



Rysunek 4.7: Edytor z zaznaczonymi miejscami naruszeń reguł JLint

Rozdział 5

Podsumowanie

Podstawowy cel pracy jakim było stworzenie w pełni funkcjonalnego modułu analizy statycznej kodu dla środowiska programistycznego Eclipse został osiągnięty. Opracowany został komponent programistyczny umożliwiający tworzenie reguł audytu kodu i rozszerzający zbiór udogodnień platformy o nowy mechanizm kontroli poprawności tworzonego przez programistę kodu. Zaimplementowano również przykładowe wykorzystanie stworzonego oprogramowania jako elementu integrującego środowisko Eclipse z popularnym narzędziem analizy kodu źródłowego Javy – JLint.

Eclipse jest bardzo elastycznym środowiskiem programistycznym. Rozbudowanie go o moduł analizy statycznej kodu otwiera możliwość dalszego rozwoju w tym kierunku. Inni programiści mogą tworzyć reguły analizy statycznej, które będą wychwytywać typowe błędy jakie powstają w ich zespołach programistycznych. Aby tworzyć te reguły nie muszą mieć dużej wiedzy na temat wewnętrznych mechanizmów Eclipse.

Zaprezentowane w pracy reguły analizy statycznej stanowią także ciekawy zbiór informacji na temat typowych błędów jakie popełniają programiści. Zapoznanie się z nimi jest z pewnością pożyteczne, natomiast ich częste występowanie uświadamia jak pomocne może być narzędzie, które automatycznie potrafi te błędy odnaleźć.

Dodatek A

Zawartość płyty CD

Do pracy dołączona została płyta CD z kodami źródłowymi zaimlementowanego modułu analizy statycznej kodu dla środowiska Eclipse.

A.1. Hierarchia katalogów wraz z opisem zawartości

- `doc` – praca magisterska,
- `src` – kody źródłowe zaimplementowanego modułu,
- `bin` – skompilowany moduł,
- `eclipse` – paczka instalacyjna środowiska Eclipse 3.0.

A.2. Instalacja modułu statycznej analizy kodu

Aby zainstalować moduł stanowiący rozszerzenia dla środowiska Eclipse należy:

1. zainstalować środowisko Eclipse,
2. rozpakować zawartość archiwum `audit.zip` w katalogu `plugins/` w miejscu gdzie został zainstalowany Eclipse.

Bibliografia

[1] David Bacon (IBM Research), Joshua Bloch (Javasoft), Jeff Bogda, Cliff Click (Hotspot JVM project), Paul Haahr, Doug Lea, Tom May, Jan-Willem Maessen, John D. Mitchell (jGuru), Kelvin Nilsen, Bill Pugh, Emin Gun Sirer, *The "Double-Checked Locking is Broken" Declaration*, http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html.

[2] Joshua Bloch, *Effective Java Programming Language Guide*, Addison Wesley, June 01, 2001.

[3] Eclipse Project, *Eclipse Project Slide Presentation*, http://www.eclipse.org/eclipse/presentation/eclipse-slides_files/v3_document.htm

[4] Dawson Engler, http://www.stanford.edu/~engler/

[5] David Gallardo, Ed Burnette, Robert McGovern, *Eclipse in Action*, Manning Publications, 2003.

[6] Erich Gamma, Kent Beck, *Contributing to Eclipse: Principles, Patterns, and Plug-Ins*, Addison Wesley Professional, Oct 20, 2003.

[7] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison Wesley Professional Computing Series, October 1994.

[8] Bill Joy, Guy Steele, James Gosling, Gilad Bracha, *Java(TM) Language Specification (2nd Edition)*, Addison Wesley Professional, 05 June, 2000.

[9] Konstantin Knizhnik, Cyrille Artho, *Jlint manual*, http://artho.com/jlint/manual.html.

[10] Sun Microsystems, *Code Conventions for the Java Programming Language*, http://java.sun.com/docs/codeconv/, 1999.

[11] Sun Microsystems, *Java 2 Platform API Specification*, http://java.sun.com/j2se/1.4.2/docs/api/index.html

[12] Sun Microsystems, *Java 2 Platform API Specification: Timestamp class*, http://java.sun.com/j2se/1.4.2/docs/api/java/sql/Timestamp.html