helion.pl

Idź do
• Spis treści
• Przykładowy rozdział

Katalog książek
• Katalog online
• Zamów drukowany katalog

Twój koszyk
• Dodaj do koszyka

Cennik i informacje
• Zamów informacje o nowościach
• Zamów cennik

Czytelnia
• Fragmenty książek online

Mistrz programowania. Zwiększ efektywność i zrób karierę

Autor: Neal Ford, David Bock
ISBN: 978-83-246-2650-2
Tytuł oryginału: The Productive Programmer
Format: 168×237, stron: 216

Poznaj efektywne narzędzia oraz mistrzowskie techniki pracy!

- Jak efektywnie zarządzać cyklem życia obiektów?
- Jak upraszczać trudne zadania przy użyciu technik metaprogramowania?
- Jak wykorzystać mądrość starożytnych filozofów w programowaniu?

Każdy profesjonalista marzy o tym, aby w jak najkrótszym czasie zrobić i zarobić jak najwięcej – dotyczy to również programistów. Autor niniejszej książki, Neal Ford, wychodzi naprzeciw tym marzeniom i stawia tezę, że kluczem do sukcesu jest mistrzostwo w posługiwaniu się dostępnymi narzędziami... w połączeniu z określoną metodologią pracy, opartą na mądrości starożytnych myślicieli. Jak uzyskać tę wyrafinowaną efektywność i tworzyć wydajne programy, dowiesz się podręcznika, który trzymasz w rękach.

Książka „Mistrz programowania. Zwiększ efektywność i zrób karierę" zawiera mnóstwo bezcennych porad, dotyczących korzystania z narzędzi zwiększających produktywność, które możesz zastosować natychmiast! Dowiesz się z niej, jak unikać najczęstszych pułapek oraz w jaki sposób pozbyć się czynników dekoncentrujących, zdążając w kierunku wydajnej i efektywnej pracy. Nauczysz się tworzyć kod o jednolitym poziomie abstrakcji, pisać testy przed napisaniem testowanego kodu, zarządzać cyklem życia obiektów i stosować techniki metaprogramowania. Dzięki temu podręcznikowi zdobędziesz potrzebną wiedzę i przyswoisz sobie najlepszą metodologię pracy – a to szybko doprowadzi Cię do mistrzostwa w Twoim zawodzie.

- Tworzenie płyty startowej
- Rejestrator makr
- Makra klawiszowe
- Wyszukiwanie zaawansowane
- Widoki zakorzenione
- Automatyzacja interakcji ze stronami WWW
- Mapy danych
- Projektowanie oparte na testowaniu
- Generowanie metryk kodu
- Metaprogramowanie

Zostań najbardziej poszukiwanym i najlepiej opłacanym programistą!

Kontakt

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl
© Helion 1991-2008

Spis treści

Przedmowa 7

Wstęp 9

1. Wprowadzenie 13
Dlaczego książka o produktywności programistów 14
O czym jest ta książka 15
Co dalej 17

I Mechanika 19

2. Przyspieszenie 21
Płyta startowa 22
Akceleratory 31
Makra 44
Podsumowanie 46

3. Skupienie 47
Pozbądź się czynników rozpraszających 48
Wyszukiwanie przebija nawigowanie 50
Wyszukiwanie zaawansowane 52
Widoki zakorzenione 54
Ustawianie atrybutów trwałych 56
Skróty do wszystkich zasobów projektu 57
Używaj kilku monitorów 57
Porządkowanie miejsca pracy na wirtualnych pulpitach 57
Podsumowanie 59

4. **Automatyzacja** ... **61**
Nie wynajduj ponownie koła 63
Zapisuj zasoby na dysku 63
Automatyzacja interakcji ze stronami internetowymi 64
Kanały RSS 64
Wykorzystanie Ant do zadań niezwiązanych z kompilacją 66
Wykorzystanie narzędzia Rake do codziennych zadań 67
Wykorzystanie Selenium do odwiedzania stron internetowych 68
Użyj basha do zliczania wyjątków 70
Zastąp pliki wsadowe interpreterem Windows Power Shell 71
Używaj Automatora z Mac OS X do usuwania starych plików 72
Oswajanie wiersza poleceń Subversion 73
Pisanie rozdzielacza plików SQL w języku Ruby 74
Argumenty za automatyzacją 75
Nie strzyż jaka 76
Podsumowanie 77

5. **Kanoniczność** ... **79**
Kontrolowanie wersji 80
Kanoniczny komputer kompilujący 82
Pośredniość 83
Niedopasowanie falowe 90
Podsumowanie 102

II **Praktyka** ... **105**

6. **Projektowanie oparte na testowaniu** ... **107**
Ewolucja testów 109

7. **Analiza statyczna** ... **117**
Analiza kodu bajtowego 118
Analiza kodu źródłowego 120
Generowanie metryk kodu za pomocą programu Panopticode 122
Analiza języków dynamicznych 124

8. **Dobry obywatel** ... **127**
Łamanie zasady hermetyzacji 128
Konstruktory 129
Metody statyczne 129
Zachowania patologiczne 134

9. **Nie będziesz tego potrzebować** 135

10. **Starożytni filozofowie** 141
Arystotelesowskie własności akcydentalne i istotne 142
Brzytwa Ockhama 143
Prawo Demeter 146
Tradycje programistyczne 147

11. **Kwestionuj autorytety** 149
Wściekłe małpy 150
Płynne interfejsy 151
Antyobiekty 153

12. **Metaprogramowanie** 155
Java i refleksja 156
Testowanie Javy za pomocą języka Groovy 157
Pisanie płynnych interfejsów 158
Dokąd zmierza metaprogramowanie 160

13. **Metody i SLAP** 161
Wzorzec composed method w praktyce 162
SLAP 166

14. **Językoznawstwo** 171
Jak do tego doszło 172
Dokąd zmierzamy 175
Piramida Oli 179

15. **Dobór odpowiednich narzędzi** 181
Poszukiwanie idealnego edytora tekstu 182
Wybór właściwego narzędzia 186
Pozbywanie się niechcianych narzędzi 192

16. **Podsumowanie — kontynuujmy dyskusję** 195

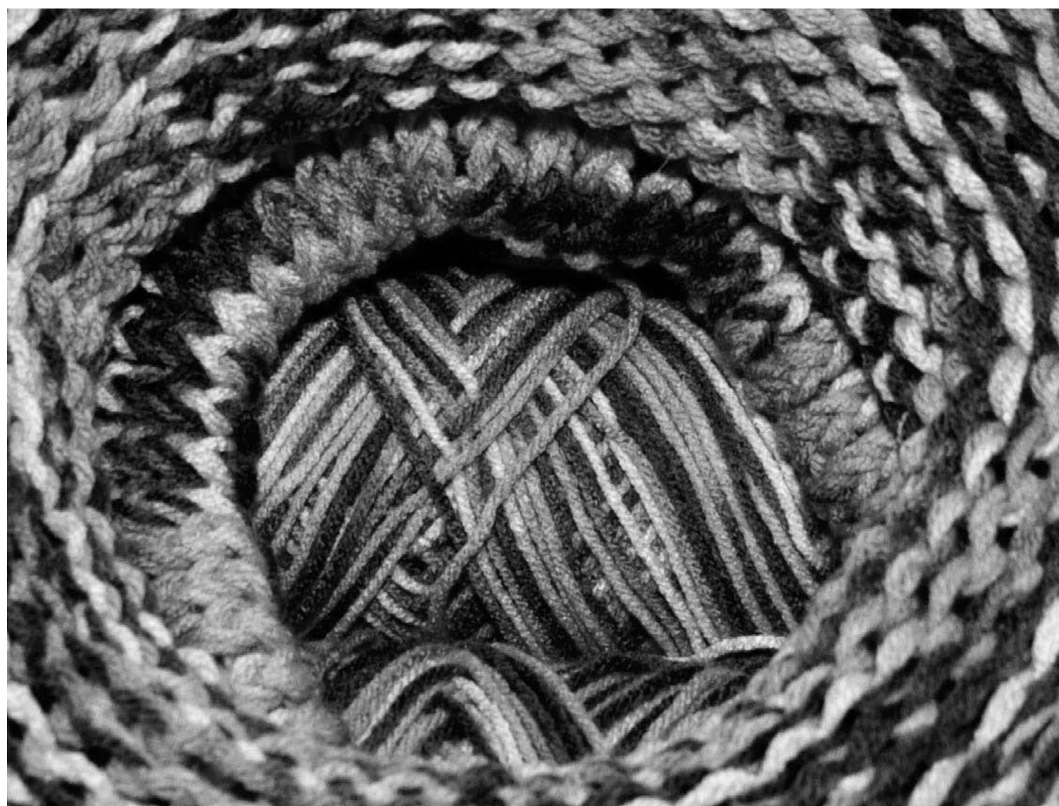
Dodatki 197

A **Elementy składowe** 199

Skorowidz 207

ROZDZIAŁ 7.

Analiza statyczna



Jeśli używasz języka programowania o statycznej kontroli typów (jak Java lub C#), masz do dyspozycji **doskonałe narzędzie umożliwiające znalezienie takich rodzajów błędów, które bardzo trudno wykryć za pomocą przeglądów kodu i innych standardowych technik**. Mowa o **analizie statycznej** — metodzie wyszukiwania w kodzie określonych oznak występowania nieprawidłowości.

Narzędzia do analizy statycznej można podzielić na dwie kategorie: narzędzia przeszukujące skompilowane artefakty (tzn. pliki klas lub kod bajtowy) i narzędzia analizujące pliki źródłowe. W rozdziale tym opiszę przykładowe aplikacje każdego z tych rodzajów, a jako przykładowego języka użyję Javy, ponieważ zestaw dostępnych narzędzi jest w tym przypadku bardzo bogaty. Należy jednak pamiętać, że nie jest to technika przeznaczona wyłącznie do stosowania w Javie. Z podobnych narzędzi można korzystać we wszystkich najważniejszych językach programowania o statycznej kontroli typów.

Analiza kodu bajtowego

Analizatory kodu bajtowego szukają w kodzie źródłowym określonych oznak występowania błędów. Wynikają z tego dwa wnioski: po pierwsze, niektóre języki programowania zostały już na tyle dobrze poznane, że w ich skompilowanym kodzie bajtowym można wyszukiwać często występujące wzorce oznaczające błędy, a po drugie, narzędzia te nie znajdują wszystkich błędów, lecz tylko takie, których wzorce zostały zdefiniowane. To nie oznacza, że narzędzia te są mało przydatne. Niektóre znajdowane przez nie błędy są niezwykle trudne do wykrycia innymi metodami (tzn. przez wielogodzinne bezproduktywne wlepianie wzroku w ekran programu diagnostycznego).

Jednym z tego typu narzędzi jest otwarty program FindBugs, będący projektem przygotowanym na Uniwersytecie Maryland. Program ten może działać w kilku trybach: z poziomu wiersza poleceń, zadania Ant lub środowiska graficznego. Graficzny interfejs użytkownika aplikacji FindBugs przedstawiono na rysunku 7.1.

FindBugs działa w kilku obszarach, do których należą:

Poprawność

Program określa prawdopodobieństwo wystąpienia błędu.

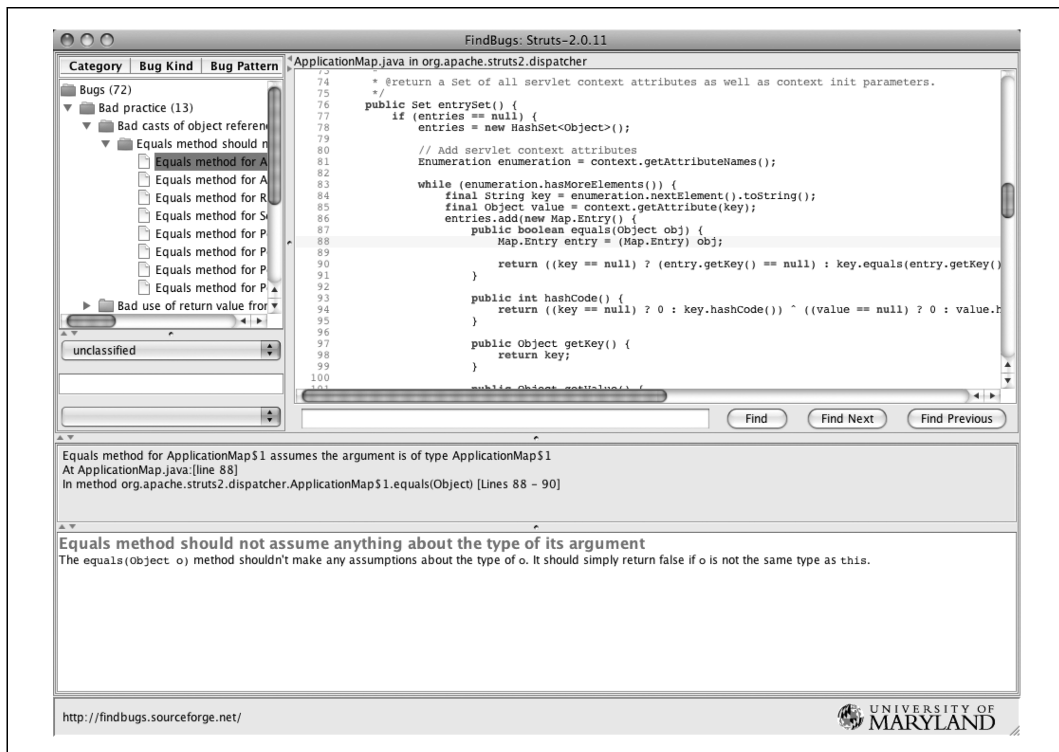
Zła praktyka

Wykrywa przypadki złamania jednego z podstawowych zaleceń dotyczących pisania kodu (na przykład przesłonięcia metody `equals()` i jednoczesnego nieprzesłonięcia metody `hashCode()`).

Podejrzane elementy

Znajduje niejasne fragmenty kodu, przypadki dziwnych zastosowań, anomalie, kod słabej jakości.

W celu zilustrowania sposobu działania programu musiałem znaleźć ofiarę. Padło na otwarty szkielet sieciowy Struts. Odkryłem kilka prawdopodobnie fałszywych alarmów z kategorii złych praktyk, które zostały opisane następująco: „W metodzie `equals` nie powinno przyjmować się żadnych założeń dotyczących typu jej argumentów”. Zalecaną praktyką definiowania metody `equals()` w Javie jest sprawdzenie rodowodu przekazywanego do niej obiektu w celu upewnienia się, że operacja porównywania ma sens. Oto fragment kodu Struts, który sprawia te problemy. Znajduje się on w pliku *ApplicationMap.java*:



Rysunek 7.1. Graficzny klient aplikacji FindBugs

```
entries.add(new Map.Entry() {
   public boolean equals(Object obj) {
       Map.Entry entry = (Map.Entry) obj;
       return ((key == null) ? (entry.getKey() == null) :
            key.equals(entry.getKey())) && ((value == null) ?
                (entry.getValue() == null) :
                value.equals(entry.getValue()));
   }
```

Uważam, że to może być fałszywy alarm, ponieważ mamy tu do czynienia z definicją anonimowej klasy wewnętrznej, więc autor prawdopodobnie zawsze zna typy argumentów. Mimo wszystko coś tu jest nie tak.

Poniżej przedstawiono oczywisty błąd znaleziony przez FindBugs. Ten fragment kodu znajduje się w pliku *IteratorGeneratorTag.java*:

```
if (countAttr != null && countAttr.length() > 0) {
       Object countObj = findValue(countAttr);
       if (countObj instanceof Integer) {
           count = ((Integer)countObj).intValue();
       }
       else if (countObj instanceof Float) {
           count = ((Float)countObj).intValue();
       }
       else if (countObj instanceof Long) {
           count = ((Long)countObj).intValue();
       }
       else if (countObj instanceof Double) {
           count = ((Long)countObj).intValue();
       }
```

Przyjrzyj się uważnie ostatniemu wierszowi tego kodu. Znajdujący się tam błąd, należący do kategorii poprawności programu FindBugs, został określony jako „niemożliwe rzutowanie”. Ostatni wiersz powyższego listingu zawsze będzie powodował wyjątek rzutowania klas. W istocie nie ma takiej sytuacji, w której uruchomienie tego kodu nie wywoła problemu. Programista sprawdza, czy obiekt o nazwie `countObj` jest typu `Double`, i od razu rzutuje go na typ `Long`. Aby dowiedzieć się, jak to się stało, wystarczy spojrzeć na znajdującą się wyżej instrukcję `if`: to jest błąd kopiowania i wklejania. Tego typu błąd trudno jest znaleźć metodą przeglądania kodu — bardzo łatwo go przeoczyć. Najwyraźniej w bazie kodu Struts nie ma testu jednostkowego pokrywającego ten wiersz kodu, ponieważ w przeciwnym razie natychmiast by go wykryto. Co gorsza, ten błąd występuje w trzech miejscach kodu Struts: we wspomnianym pliku *IteratorGeneratorTag.java* i dwa razy w pliku *SubsetIteratorTag.java*. Powód? Nietrudno zgadnąć. Kod ten został skopiowany i wklejony we wszystkich tych trzech miejscach (FindBugs tego nie wykrył, sam zauważyłem, że te fragmenty są do siebie podejrzanie podobne).

Dzięki możliwości zautomatyzowania uruchamiania programu FindBugs przez Ant lub Maven jako części procesu kompilacji aplikacja ta pełni funkcję bardzo taniej polisy ubezpieczeniowej na wypadek wystąpienia rozpoznawanych przez nią błędów. Oczywiście nie można zagwarantować w ten sposób, że kod będzie całkowicie wolny od błędów (i nie zwalnia to z obowiązku pisania testów jednostkowych), ale program ten może Cię uratować przed paroma nieprzyjemnymi wpadkami. Potrafi on nawet znaleźć takie błędy, które trudno wykryć w testach jednostkowych, na przykład problemy z synchronizacją wątków.

Narzędzia do analizy statycznej stanowią łatwy sposób weryfikacji.

Analiza kodu źródłowego

Narzędzia do analizy kodu źródłowego przeszukują kod źródłowy w celu znalezienia określonych oznak występowania błędów. W zaprezentowanym przykładzie użyłem otwartego narzędzia Javy o nazwie PMD. Program ten działa z poziomu wiersza poleceń, Ant oraz ma wtyczki do wszystkich najważniejszych środowisk programistycznych. Aplikacja ta znajduje następujące typy nieprawidłowości:

Potencjalne błędy

Na przykład puste bloki `try…catch`.

Martwy kod

Nieużywane zmienne lokalne, parametry i zmienne prywatne.

Nieoptymalny kod

Marnotrawne użycie łańcuchów.

Nadmiernie skomplikowane wyrażenia

Wielokrotne użycie kodu spowodowane kopiowaniem i wklejaniem.

Duplikaty kodu (obsługiwane przez pomocnicze narzędzie o nazwie CPD)

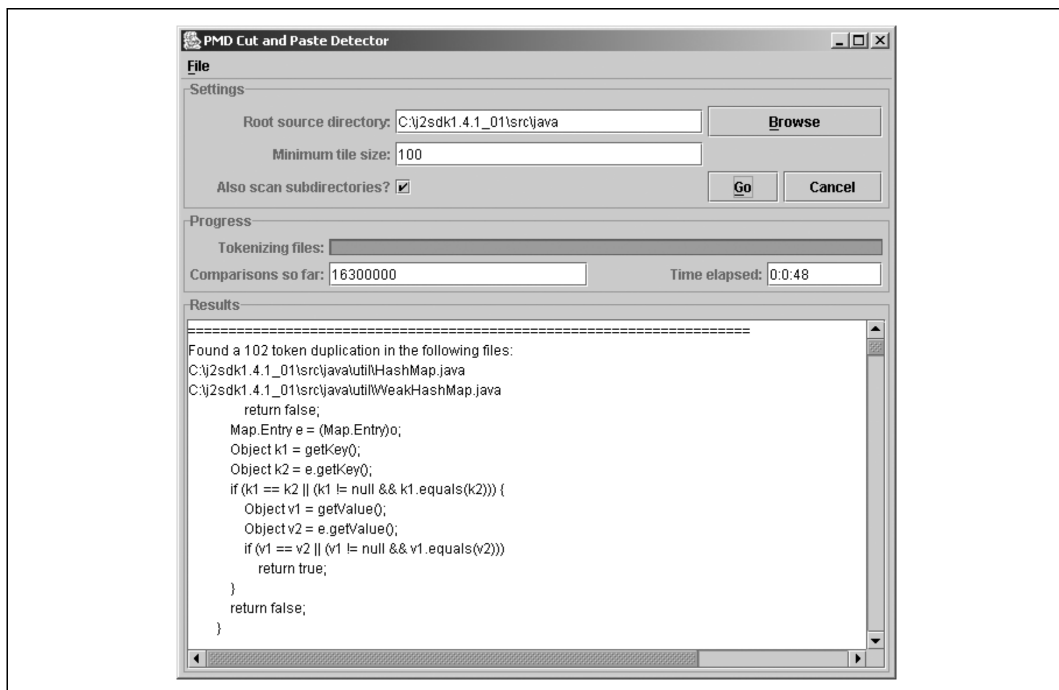
Wielokrotne użycie kodu spowodowane kopiowaniem i wklejaniem.

PMD można określić jako coś pomiędzy zwykłym narzędziem do sprawdzania stylu (takim jak CheckStyle, które weryfikuje, czy kod został napisany zgodnie z wytycznymi dotyczącymi stylu, na przykład czy zostały zastosowane odpowiednie wcięcia) a programem FindBugs, który analizuje kod bajtowy. Rodzaje błędów, w wykrywaniu których specjalizuje się PMD, pokażę na przykładzie metody napisanej w języku Java:

```
private void insertLineItems(ShoppingCart cart, int orderKey) {
    Iterator it = cart.getItemList().iterator();
    while (it.hasNext()) {
        CartItem ci = (CartItem) it.next();
        addLineItem(connection, orderKey, ci.getProduct().getId(),
            ci.getQuantity());
    }
}
```

PMD poinformuje, że metodę tę można zoptymalizować przez dodanie do pierwszego parametru (`ShoppingCart cart`) modyfikatora `final`. Dzięki zastosowaniu się do tej rady można zmusić kompilator do wykonania za nas większej ilości pracy. Ponieważ w Javie wszystkie obiekty są przekazywane przez wartość, nie można zmiennej `cart` przypisać referencji do nowego obiektu wewnątrz tej metody, a próba zrobienia tego spowoduje wystąpienie błędu. PMD oferuje kilka tego typu wskazówek, które pomagają zoptymalizować działanie różnych narzędzi (na przykład kompilatora).

Program PMD ma również funkcję wykrywania podejrzanie wyglądających skopiowanych i wklejonych fragmentów kodu (jak wcześniej pokazany kod Struts) o nazwie CPD (ang. *Cut-Paste Detector*). Interfejs użytkownika CPD został zbudowany przy użyciu biblioteki Swing. Są w nim wyświetlane stan i problematyczny fragment kodu (rysunek 7.2). Sposób jego działania oczywiście również jest związany z zasadą DRY, którą opisałem w rozdziale 5.



Rysunek 7.2. Narzędzie Cut and Paste Detector programu PMD

Większość podobnych narzędzi do analizy kodu źródłowego ma API dające się dostosować do indywidualnych potrzeb, w którym można utworzyć własne zestawy reguł (takie API ma zarówno FindBugs, jak i PMD). Większość z nich oferuje ponadto tryb interaktywny i, co jeszcze ważniejsze, pozwala uruchamiać się jako część zautomatyzowanych procesów, na przykład ciągłej integracji. Uruchamianie tych narzędzi przed każdym wysłaniem plików do repozytorium jest bardzo prostym sposobem na uniknięcie typowych błędów. Ktoś zadał sobie trud zidentyfikowania błędów, a Ty możesz z tego skorzystać.

Generowanie metryk kodu za pomocą programu Panopticode

Zagadnienia związane z metrykami kodu wykraczają poza zakres tematyczny tej książki, ale *produktywność* w ich generowaniu nie. Jeśli chodzi o języki o statycznej kontroli typów (jak Java i C#) jestem wielkim zwolennikiem ciągłego gromadzenia metryk, aby mieć pewność, że problem zostanie rozwiązany możliwie jak najszybciej. Oznacza to, że zwykle mam cały zestaw narzędzi do mierzenia jakości kodu (włącznie z programami FindBugs i PMD/CPD), które uruchamiam w ramach procesu ciągłej integracji.

Konfigurowanie tych wszystkich programów w każdym projekcie jest kłopotliwe. Podobnie jak w przypadku programu Buildix (zobacz podrozdział „Nie wynajduj ponownie koła” w rozdziale 4.) chciałbym mieć wstępnie skonfigurowaną całą infrastrukturę. W tym właśnie najlepszy jest Panopticode.

Podobny problem miał kiedyś jeden z moich kolegów (Julias Shaw), ale on zamiast narzekać jak ja, wziął się do roboty. Panopticode[1] to program typu open source zawierający mnóstwo wstępnie skonfigurowanych narzędzi do generowania metryk kodu. Jego jądro stanowi plik kompilacji Ant, zawierający wiele projektów open source i ich wstępnie skonfigurowanych plików JAR. Wystarczy podać ścieżkę do swoich plików źródłowych, ścieżkę do bibliotek (innymi słowy, ścieżkę do wszystkich plików JAR potrzebnych do skompilowania projektu) oraz katalog testów, a następnie uruchomić plik kompilacyjny Panopticode. Program zajmie się resztą.

Emma

Narzędzie typu open source do mierzenia pokrycia kodu testami (zobacz podrozdział „Pokrycie kodu” w rozdziale 6.). Zmieniając jeden wiersz w pliku kompilacji Panopticode, można z tego narzędzia przełączyć się na narzędzie Cobertura (kolejny program open source do mierzenia pokrycia kodu źródłowego Javy).

CheckStyle

Otwarty weryfikator stylu kodu. Można utworzyć dostosowane do własnych potrzeb zestawy reguł za pomocą jednego wpisu w pliku kompilacji Panopticode.

JDepend

Otwarte narzędzie zbierające metryki numeryczne na poziomie pakietów.

---

[1] Do pobrania pod adresem *http://www.panopticode.org*.

JavaNCSS

Otwarte narzędzie do mierzenia złożoności cyklomatycznej (zobacz podrozdział „Złożoność cyklomatyczna" w rozdziale 6.).

Simian

Komercyjny program służący do znajdowania powtarzających się fragmentów kodu. Wersję dostępną w Panopticode można bezpłatnie testować przez 15 dni. Po upływie tego czasu należy usunąć program lub za niego zapłacić. Planowane jest umożliwienie jego wymiany na CPD.

Panopticode Aggregator

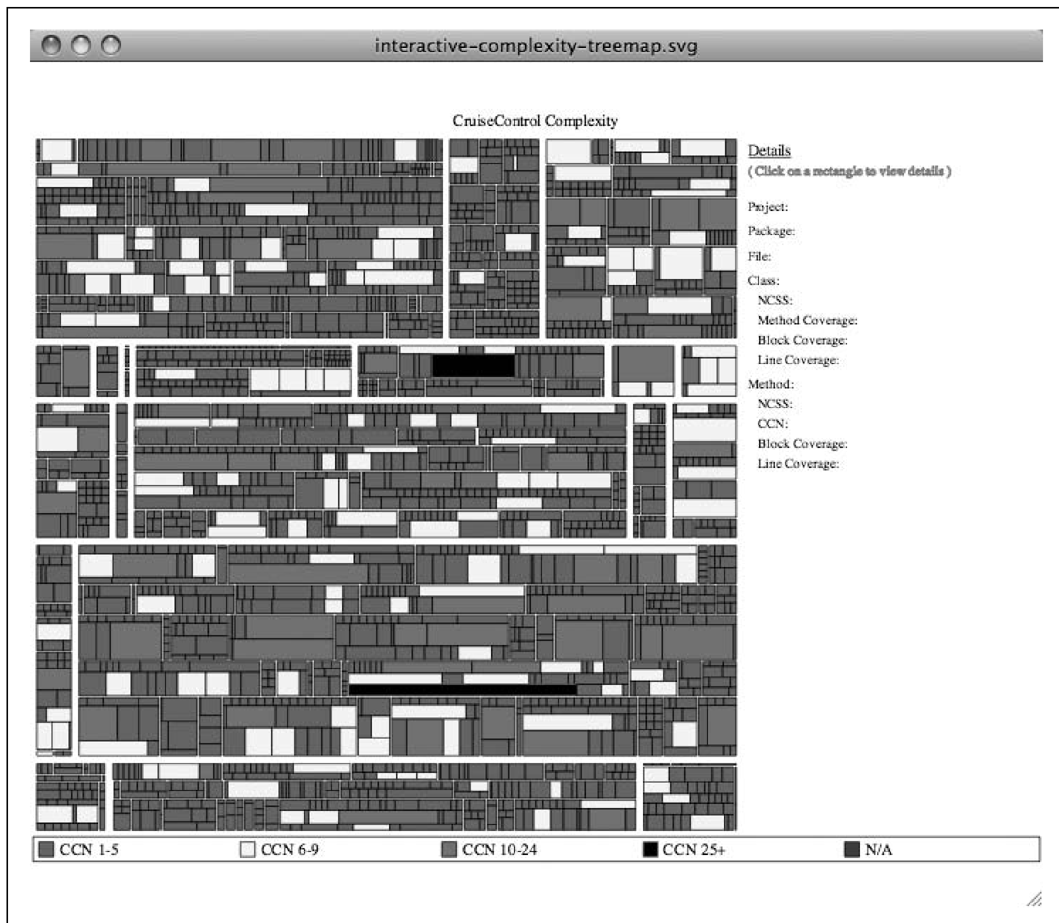
Narzędzie przedstawiające raporty wszystkich narzędzi Panopticode w formie tekstowej i graficznej (w postaci schematów).

Po uruchomieniu Panopticode przez jakiś czas przeżuwa kod źródłowy, a następnie wypluwa raport zawierający informacje o jego jakości. Program ten tworzy także bardzo ładne schematy w postaci zaawansowanych obrazów w formacie SVG (większość przeglądarek potrafi wyświetlać tego typu grafikę). Schematy te mają dwie zalety. Pierwszą z nich jest to, że stanowią graficzny obraz wartości określonej metryki. Na przykład na rysunku 7.3 została przedstawiona złożoność cyklomatyczna projektu CruiseControl. Zacienione obszary reprezentują różne przedziały wartości dla określonej metody. Grube białe linie oznaczają granice pakietów, a cienkie linie — granice klas. Każde pole odpowiada jednej metodzie.

Drugą zaletą schematów jest ich interaktywność. To nie jest tylko ładny obrazek — to obrazek interaktywny. Jeśli rysunek ten zostanie wyświetlony w przeglądarce internetowej, kliknięcie dowolnego pola spowoduje wyświetlenie po prawej stronie metryk odpowiadającej mu metody. Schematy te umożliwiają przeanalizowanie kodu w celu znalezienia metod i klas, które sprawiają problemy.

Panopticode wyświadcza programiście dwie bardzo istotne przysługi. Po pierwsze, nie trzeba w każdym projekcie ciągle konfigurować tych samych rzeczy. W typowym projekcie konfiguracja programu trwa nie więcej niż 5 minut. Po drugie, program umożliwia tworzenie schematów, które służą jako **agregatory informacji**. W projektach zwinnych agregator informacji (ang. *information radiator*) to reprezentacja stanu projektu umieszczona w jakimś widocznym miejscu (na przykład w pobliżu ekspresu do kawy). Dzięki niemu członkowie zespołu nie muszą otwierać żadnych załączników, aby sprawdzić stan projektu, ponieważ mogą to zrobić, idąc po kawę.

Jeden z rodzajów schematów, jakie można tworzyć za pomocą Panopticode, to ilustracje pokrycia kodu (zobacz podrozdział „Pokrycie kodu" w rozdziale 6.). Jeśli metryka wygląda jak wielka czarna plama, nie jest dobrze. Zwiększanie pokrycia kodu zacznij od znalezienia największej możliwej kolorowej drukarki i wydrukuj na niej swój schemat (właściwie za pierwszym razem zwykle wystarcza biało-czarna, ponieważ i tak wydruk jest cały czarny). Powieś ten wydruk w widocznym miejscu. Po zakończeniu jakiegoś ważnego etapu pisania testów wydrukuj nowy schemat i powieś go obok poprzedniego. Jest on bardzo skutecznym motywatorem wszystkich członków zespołu — nikt nie chce być świadkiem ponownego pogrążania się projektu w ciemności. Jest to także dobry sposób na pokazanie kierownikowi, że pokrycie kodu testami jest coraz większe. Kierownicy lubią takie obrazy z rozmachem, a ten na dodatek zawiera użyteczne informacje!



Rysunek 7.3. Mapa obrazująca złożoność cyklomatyczną projektu CruiseControl

Analiza języków dynamicznych

Mimo że języki dynamiczne są uważane za jedne z najefektywniejszych w wielu dziedzinach programowania, brakuje dla nich narzędzi analitycznych, których istnieje mnóstwo dla języków o statycznej kontroli typów. Tworzenie narzędzi do analizy kodu źródłowego języków dynamicznych jest trudniejsze, ponieważ nie można oprzeć się na cechach systemu typów.

W przypadku języków dynamicznych bada się głównie złożoność cyklomatyczną (którą można analizować w każdym języku o blokowej strukturze kodu) i pokrycie kodu testami. Przykładowym narzędziem do sprawdzania pokrycia kodu w języku Ruby jest rcov. W Ruby on Rails narzędzie to jest nawet dostępne standardowo (przykładowy raport rcov przedstawiono na rysunku 15.1). Do mierzenia złożoności cyklomatycznej można użyć programu Saikuro[2].

[2] Do pobrania pod adresem *http://saikuro.rubyforge.org*.

Z powodu niedostępności typowych statycznych narzędzi analitycznych programiści języka Ruby stali się sprytniejsi. W wyniku tej sytuacji powstało kilka ciekawych propozycji mierzenia kodu nietypowymi metodami. Jedna z nich to flog[3]. Program ten analizuje przypisania, rozgałęzienia oraz wywołania, którym nadaje wagę. Przypisuje pewną wartość każdemu wierszowi metody i przedstawia wyniki w następującej postaci (jest to wynik zwrócony dla przykładu `SqlSplitter` z podrozdziału „Refaktoryzacja programu SqlSplitter w celu umożliwienia testowania” rozdziału 15.):

```
SqlSplitter#generate_sql_chunks: (32.4)
    20.8: assignment
     7.0: branch
     3.4: downcase
     3.0: +
     2.9: ==
     2.8: create_output_file_from_number
     2.8: close
     2.0: lit_fixnum
     1.7: %
     1.4: puts
     1.4: lines_o_sql
     1.2: each
     1.2: make_a_place_for_output_files
SqlSplitter#create_output_file_from_number: (11.2)
     4.8: +
     3.0: |
     1.8: to_s
     1.2: assignment
     1.2: new
     0.4: lit_fixnum
```

Z tego raportu wynika, że najbardziej złożoną metodą w klasie jest `generate_sql_chunks`, której wartość wynosi `32.4` (suma wartości znajdujących się pod nią). Najwyższa złożoność w tej metodzie jest związana z przypisaniami. Aby więc ją uprościć, należałoby zacząć od przyjrzenia się instrukcjom przypisania, których jest tam całe mnóstwo.

Nietypowym przypadkiem jest język Groovy, ponieważ sam w sobie jest dynamiczny, ale wytwarza kod bajtowy w języku Java. Oznacza to, że można używać standardowych narzędzi do analizy kodu Java, ale pod warunkiem że będą one uruchamiane na kodzie bajtowym. Wyniki mogą jednak nie być satysfakcjonujące. Groovy do wykonania niektórych swoich sztuczek potrzebuje utworzenia wielu klas pośrednich i osłonowych, dlatego w raporcie zostanie wykazanych wiele klas, których w rzeczywistości nie utworzyłeś. Niektórzy programiści zajmujący się tworzeniem narzędzi do analizy kodu napisanego w Javie zaczęli już brać pod uwagę język Groovy (przykładowym narzędziem, które może być stosowane do analizy kodu w tym języku, jest Cobertura), ale ich prace są dopiero w początkowej fazie.

---

[3] Do pobrania pod adresem *http://ruby.sadi.st/Flog.html*.