Mieszko Jabłoński, Michał Karbowańczyk
Politechnika Łódzka Instytut Informatyki
ul. Wólczańska 215, 90-924 Łódź
email: michal.karbowanczyk@p.lodz.pl

AUTOMATYCZNA OCENA KOMPLETNOŚCI PROJEKTU PROGRAMISTYCZNEGO

Streszczenie – W artykule przedstawiono projekt oraz kluczowe aspekty realizacji biblioteki służącej do automatycznego testowania kompletności implementacji danego problemu w języku Java poprzez testy jednostkowe. Dzięki użyciu mechanizmów refleksji oraz programowania aspektowego prezentowane rozwiązanie umożliwia przeprowadzenie standardowych testów jednostkowych na z góry nieznanych, dostarczonych z zewnątrz klasach. Przedstawiona biblioteka jest rozwijana w ramach pracy inżynierskiej o tym samym tytule.

Słowa kluczowe: Java, testy jednostkowe, programowanie aspektowe, programowanie refleksyjne

1 Wprowadzenie

Testowanie automatyczne, w tym testowanie jednostkowe [1] jest obecnie jednym z podstawowych sposobów zapewnienia niezawodności oprogramowania. Właściwie sformułowane i regularnie przeprowadzane testy umożliwiają nie tylko weryfikację zgodności implementacji z założonym kontraktem na różnych poziomach (metod, klas, modułów, usług), lecz także wykrywanie tzw. regresji, kiedy to zmiana w jednym z komponentów aplikacji powoduje błędne działanie w innym, czasem pozornie nie związanym z tą zmianą komponencie.

Skuteczność testowania doprowadziła do upowszechnienia tej techniki, a także do wytworzenia metodyki tworzenia oprogramowania zwanej programowaniem sterowanym testami (ang. Test Driven Development, TDD) [2]. Według zasad tej metodyki kontrakt danego fragmentu aplikacji jest formułowany w całości poprzez zestaw scenariuszy testowych. Początkowo, wobec braku implementacji odpowiedniej funkcjonalności, taki zestaw testów daje wyniki wyłącznie negatywne lub wręcz jest niemożliwy do skompilowania i uruchomienia. Wraz z postępem implementacji odsetek testów dających negatywne wyniki zmniejsza się, aż do osiągnięcia stanu w którym wszystkie testy wyrażające kontrakt dają wynik pozytywny. Stwierdzamy wówczas, że dana funkcjonalność została skompletowana.

W artykule przedstawiono założenia, sposób realizacji oraz przykład użycia nowego rozwiązania, którego celem jest zautomatyzowanie oceny kompletności projektu programistycznego poprzez testy jednostkowe przy braku statycznego powiązania pomiędzy kodem testującym i kodem testowanym.

2 Automatyczna ocena projektu przez testowanie

Celem prac przedstawionych w niniejszym artykule jest dostarczenie metody automatycznej oceny kompletności projektu programistycznego poprzez testy jednostkowe. Metoda taka jest znana (i opisana powyżej) w przypadku wytwarzania oprogramowania; wystarczy wyrazić kontrakt danego fragmentu aplikacji poprzez testy, których pozytywny przebieg potwierdzi ukończenie implementacji. Kod testów jest wówczas częścią realizowanego projektu i jest z nim w pełni zintegrowany. W szczególności kod testów może założyć stosowanie z góry ustalonych identyfikatorów klas czy metod, do których w tej sytuacji można odwoływać się statycznie.

Sytuacja jest zupełnie inna, gdy celem implementacji nie jest wytworzenie oprogramowania samo w sobie, lecz ocena kompetencji osoby lub zespołu tworzącego oprogramowanie. Typowymi przykładami są tu: rekrutacja pracowników oraz sprawdzanie zadań w ramach procesu dydaktycznego. W takim przypadku sformułowanie testu należy do *strony testującej* (*oceniającej*), zaś dostarczenie implementacji – do *strony testowanej* (*ocenianej*). Ocena implementacji pod kątem kompletności i zgodności z kontraktem wymaga zintegrowania ze sobą obu części i doprowadzenia do uruchomienia testów. Zadanie to jest pozornie proste – wymagana jest jedynie integracja dwóch rozłącznych drzew plików oraz przeprowadzenie w tak uzyskanym jednolitym projekcie kompilacji i uruchomienia testów. W praktyce jednak występują tu co najmniej dwa fundamentalne problemy:

- określenie *punktów wejściowych*, czyli plików, klas, metod lub funkcji, do których można odwołać się w kodzie testu aby wykonać procedurę testową;
- pozyskanie wyniku wykonania.

Aplikacja Codility

Publicznie dostępnym narzędziem umożliwiającym zautomatyzowaną ocenę rozwiązań zadań różnego typu, w tym – programistycznych, jest aplikacja internetowa Codility [3]. Umożliwia ona formułowanie zadań polegających na stworzeniu lub zoptymalizowaniu istniejącego kodu w jednym z 14 języków programowania. Rys. 1 przedstawia przykładowy stan początkowy zadania polegającego na optymalizacji funkcji języka JavaScript liczącej wartość wyrazu ciągu Fibonacciego.

Rys. 1. Codility – stan początkowy zadania

Realizacja testu polega w tym przypadku na wykonaniu funkcji *fibonacci* z określoną wartością argumentu i porównaniu wartości zwróconej do wartości oczekiwanej. W tym zakresie działanie aplikacji jest zbliżone do idei testu jednostkowego. Jeżeli wartości są zgodne, pod uwagę brany jest czas znajdowania wyniku przez zaproponowany przez ocenianego algorytm. Z technicznego punktu widzenia natomiast integracja testu z ocenianym kodem jest możliwa dzięki temu, że test narzuca identyfikatory punktów wejściowych – zarówno widocznej w implementacji zmiennej *yourself*, jak i funkcji *function*. Zmiana któregokolwiek z tych identyfikatorów powoduje, że test jest niemożliwy do przeprowadzenia, co ilustruje Rys. 2.

Rys. 2. Codility – zmiana identyfikatora zmiennej stanowiącej punkt wejściowy

Aplikacja BaCa

Aplikacja BaCa [4] została utworzona i wdrożona w Instytucie Informatyki i Matematyki Komputerowej Uniwersytetu Jagiellońskiego. Aplikacja umożliwia studentowi zamieszczanie plików źródłowych stanowiących rozwiązanie zadania sformułowanego przez prowadzącego dane zajęcia. Aplikacja zapewnia złączenie plików dostarczonych przez obie strony oraz kompilację i uruchomienie programu wynikowego. Podobnie jak w przypadku Codility występują tu statyczne odwołania do nazw plików, klas, metod czy funkcji będących punktami wejściowymi, dla przykładu – w przypadku zadań realizowanych w języku C/C++ określona jest nazwa pliku źródłowego, który jest włączany do kodu testu dyrektywą preprocesora *#include*; ponadto określone są nazwy funkcji, do których odwołuje się kod funkcji *main* stanowiącej implementację testu. Jeżeli chodzi o pozyskiwanie wyników wykonania aplikacja BaCa pozwala na zdefiniowanie oczekiwanych wzorców dla standardowego strumienia wyjścia uruchamianego w ramach testu procesu, co przybliża całe rozwiązanie do kategorii testów integracyjnych/funkcjonalnych. Ponadto jest możliwe ustanowienie ograniczenia pamięci dostępnej dla procesu, a także ocena efektywności implementacji poprzez pomiar czasu wykonania.

3 Określenie potrzeb

Przegląd istniejących rozwiązań w zakresie zautomatyzowanego testowania oprogramowania ujawnił ich wspólną cechę, polegającą na statycznym określeniu punktów wejściowych łączących kod testu z kodem testowanym. Motywacja takiego podejścia jest zrozumiała – umożliwia ono skuteczną integrację i wykonanie testów w dowolnym języku programowania, nie uciekając się do niestandardowych technik. Jednak z dydaktycznego punktu widzenia rozwiązanie to ma zasadniczą wadę: narzucenie statycznej identyfikacji punktów wejściowych przez kod testu determinuje w znacznym stopniu strukturę kodu testowanego. Jeżeli zadanie skupia się na problemach algorytmicznych wada ta ma charakter drugorzędny, jednak jeżeli celem zadania jest stworzenie np. modelu obiektowego lub modelu komponentów biznesowych, to w efekcie znaczący aspekt oceny zadania musi zostać pominięty, ponieważ określenie struktury jest realizowane po stronie testującej, nie zaś po stronie testowanej.

Zestawiając wynik powyższej analizy z rozpoznanymi potrzebami dydaktycznymi zaproponowano alternatywną realizację mechanizmu zautomatyzowanego testowania oprogramowania, dla której określono następujące założenia:

- ograniczenie zastosowania do języka Java i wykorzystanie możliwości dostępnych dla tego języka;

- możliwość zdefiniowania punktów wejściowych testu poprzez metadane, nie zaś poprzez statyczne identyfikatory;
- możliwość dostarczania kodu testowanego w formie archiwum JAR zawierającego skompilowany kod testowany, brak konieczności dostarczania kodu źródłowego i jego kompilacji;
- integracja rozwiązania ze standardowymi narzędziami (biblioteka testów jednostkowych JUnit [1][5], zarządca projektów Maven [6]).

4 Projekt AESPCT

Charakterystyka ogólna

Projekt AESPCT (ang. Automated Evaluation of Software Project CompleTeness) jest rozwiązaniem umożliwiającym ocenę kompletności realizacji projektu programistycznego w języku Java poprzez wykonywanie testów jednostkowych. Projekt ma stanowić realizację założeń przedstawionych w poprzednim rozdziale.

Z perspektywy użytkowej rozwiązanie ma postać trzech archetypów Maven o następujących nazwach i zastosowaniu:

- *aespct-lib* – zawiera adnotacje potrzebne zarówno dla definiującego testy, jak i implementującego rozwiązanie;
- *aespct-core* – archetyp dla definiującego testy; archetyp ten rozszerza się o klasy testów, które następnie zostaną wykonane na dostarczonych rozwiązaniach;
- *aespct-impl* – elementarny archetyp dla implementującego rozwiązanie; dostarcza on zależność z archetypem *aespct-lib* i stanowi punkt wyjścia dla implementacji rozwiązania, które zostanie poddane testom.

Kluczową cechą proponowanego rozwiązania jest brak statycznej zależności między archetypem definiującego testy (*aespct-core*) a archetypem implementującego rozwiązanie (*aespct-impl*) innej, niż współdzielona biblioteka (*aespct-lib*). Inaczej mówiąc, kod wytworzony przez stronę testującą i przez stronę testowaną nie mają wzajemnych, bezpośrednich odwołań do siebie.

Opisany powyżej efekt został osiągnięty poprzez zastosowanie zaawansowanych technik, a zarazem paradygmatów dostępnych w języku Java: programowania aspektowego oraz programowania refleksyjnego.

Programowanie aspektowe

Istotą paradygmatu programowania aspektowego (ang. aspect-oriented programming, AOP) [7] jest rozdzielenie programu na części niezwiązane funkcjonalnie. W przypadku prezentowanego rozwiązania

zastosowanie programowania aspektowego pozwala oddzielić warstwę automatyzacji testów od logiki ich definiowania oraz kodu testowanego.

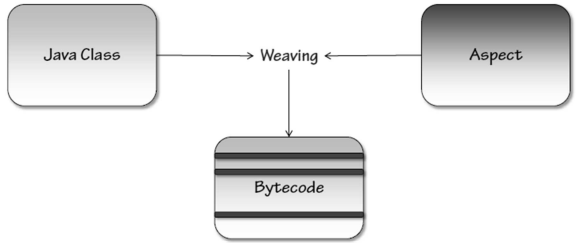
Programowanie aspektowe jest dostępne dla języka Java dzięki rozszerzeniu AspectJ [7][8]. Kluczowymi konceptami definiowanymi przez AspectJ są: punkty złączenia, punkty przecięcia oraz rady.

Punkt złączenia (ang. joinpoint) to dowolny, identyfikowalny z punktu widzenia semantyki języka punkt programu. W praktyce najczęściej wykorzystywane są punkty złączenia związane z oddziaływaniem z daną klasą lub obiektem: wykonanie metody, odwołanie do pola, obsługa wyjątku czy statyczna inicjalizacja klasy. W przypadku testów jednostkowych użyteczne są punkty złączenia związane z wykonywaniem metod (*execution*) przez test. Punkty złączenia są logicznie grupowane w *punkty przecięcia* (ang. pointcuts) – zbiory punktów złączenia podlegające wspólnej konfiguracji. W szczególności każdy punkt złączenia jest też punktem przecięcia.

Rada (ang. *advice*) to blok kodu zostanie wykonany w przypadku napotkania punktu przecięcia. Definicja rady składa się z określenia momentu jej uruchomienia, definicji punktu przecięcia oraz ciała (bloku kodu). Wyróżniamy trzy typy rad. Rada *before* wykonywana jest przed wystąpieniem punktu przecięcia. Ten typ rady może służyć na przykład do weryfikacji wartości argumentów wykonania metody. Rada *after* wykonywana jest po wystąpieniu punktu przecięcia. W związku z tym, że podczas definiowania rady after nie wiadomo, czy kod w punkcie przecięcia wykona się poprawnie, wyróżniono dwa typy rady after: *after returning* – kod w punkcie przecięcia wykonał się poprawnie oraz *after throwing* – wykonanie kodu w punkcie przecięcia zakończyło się rzuceniem nie obsłużonego wyjątku.

Najbardziej złożona rada *around* otacza całe wykonanie przecięcia, a zatem wykonuje się przed punktem przecięcia i kontroluje jego właściwe wykonanie oraz to, co dzieje się po jego zakończeniu. Rada ta pozwala całkowicie zmodyfikować zachowanie rozważanego programu w danym przecięciu; w szczególności można pominąć wykonanie pierwotnego bloku, dla którego zdefiniowane jest przecięcie.

Należy zwrócić uwagę, że nałożenie rady na punkt przecięcia nie wymaga ani modyfikacji, ani nawet ponownej kompilacji kodu źródłowego objętego radą. AspectJ działa bowiem na poziomie bajtkodu Java, wstrzykując bajtkod rady w bajtkod punktu przecięcia objętego radą. Proces ten, zilustrowany na Rys. 3, nazywany jest splataniem lub tkaniem (ang. *weaving*).



Rys. 3. Proces splatania bajtkodu aspektu (rady) z bajtkodem punktu przecięcia (klasy Java) [9]

W prezentowanym rozwiązaniu programowanie aspektowe zostało wykorzystane w celu automatycznej zamiany wywołań metod w kodzie testującym na odpowiednie wywołania metod w kodzie testowanym.

Adnotacje i programowanie refleksywne

Adnotacja [10] jest mechanizmem wprowadzonym w platformie Java 5, pozwalającym na dołączenie do bajtkodu klas Java dodatkowych informacji – metadanych. Adnotacje mogą być stosowane na poziomie różnych elementów klasy Java – od zmiennej lokalnej aż po pakiet, natomiast najbardziej popularne i zarazem wykorzystane w prezentowanym rozwiązaniu są adnotacje na poziomie klas i metod.

Należy przy tym pamiętać, że zastosowanie adnotacji nie wpływa na bajtkod klasy wynikający z procesu kompilacji, w szczególności nie wpływa na wykonywanie kodu metod. Fakt zastosowania adnotacji dla danego elementu, wraz z ewentualnymi wartościami atrybutów adnotacji, zostaje dołączony do bajtkodu klasy i może zostać przeanalizowany w czasie wykonania poprzez mechanizm refleksji. Metadane niesione przez adnotacje wpływają wówczas na sposób działania mechanizmów otaczających wykonanie. Takie zastosowanie adnotacji jest rozpowszechnione na przykład w kontenerach platformy Java EE (kontener CDI, kontener EJB) [11][12].

Mechanizm refleksji, reprezentowany w języku Java przez klasę *java.lang.Class* i pakiet klas *java.lang.reflect*, pozwala na analizę kodu aplikacji w czasie działania jej samej. Umożliwia on także zastosowanie programowania refleksyjnego [13], w którym w miejscu statycznych odwołań do klas i składowych można zastosować odwołania dynamiczne, w których identyfikator składowej jest zmienną programu.

W prezentowanym rozwiązaniu mechanizm adnotacji został wykorzystany do oznaczenia klas i metod podlegających testowaniu, zaś mechanizm refleksji – do znajdowania tak oznaczonych elementów w kodzie testowanym. W ten sposób zlikwidowany został formalny związek między nazwami klas i metod zastosowanymi w konstrukcji testu a tymi zastosowanymi w kodzie testowanym. Definicje adnotacji

zaprezentowane zostały na Rys. 4 oraz Rys. 5. W obu przypadkach związek między elementem testu a elementem kodu testowanego ustalany jest na podstawie zgodnej wartości atrybutu *name*.

```
/.../

package pl.lodz.p.edu.mieszkojablonski.aespct.lib.annotation;

import java.lang.annotation.*;

/**...*/
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface ClassUnderTest {

    String name();

}
```

Rys. 4. Adnotacja @ClassUnderTest – poziom klasy

```
/.../

package pl.lodz.p.edu.mieszkojablonski.aespct.lib.annotation;

import java.lang.annotation.*;

/**...*/
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface MethodUnderTest {

    String name();

}
```

Rys. 5. Adnotacja @MethodUnderTest – poziom metody

Definiowanie testu

Poprawne zdefiniowanie testu w prezentowanym rozwiązaniu wymaga odwołania się do artefaktu *aespct-core*. W pierwszej kolejności należy zmodyfikować interfejs *DesiredInterface* dodając do niego sygnatury metod, które mają zostać poddane testowi.

```
/.../

package pl.lodz.p.edu.mieszkojablonski.aespct.core.domain;

import ...

/**...*/
public interface DesiredInterface {

    /**...*/
    String exampleTestedMethod();

    String someOther(BigDecimal one, BigDecimal two, Date three);

    String someOther(List<Boolean> one, BigDecimal two, BigDecimal three);

}
```

Rys. 6. Interfejs DesiredInterface

Archetyp *aespct-core* dostarcza także klasę abstrakcyjną *Proxy* implementującą *DesiredInterface*, która odgrywa kluczową rolę w procesie łączenia kodu testującego z kodem testowanym. Kolejnym krokiem jest zatem zdefiniowanie klasy wzorcowej rozszerzającej *Proxy*, której zadaniem jest określenie poprzez adnotacje identyfikacji klasy i metod, które rzeczywiście zostaną poddane testowi. Należy zwrócić uwagę, że ciała testowanych metod są niezbędne tylko ze względów formalnych; w czasie wykonywania testu ich wykonania zostaną poprzez zastosowanie aspektów zamienione na wykonania odpowiednich metod *faktycznej* klasy testowanej.

```
/.../

package pl.lodz.p.edu.mieszkojablonski.aespct.core.domain;

import ...

@ClassUnderTest(name = "Klasa_A")
public class SimpleProxy extends Proxy {

    public SimpleProxy(Object instance) { super(instance); }

    @MethodUnderTest(name = "Metoda_A")
    @Override
    public String exampleTestedMethod() {
        return "FAIL";
    }

    @MethodUnderTest(name = "Metoda_B")
    @Override
    public String someOther(BigDecimal one, BigDecimal two, Date three) {
        return null;
    }

    @MethodUnderTest(name = "Metoda_C")
    @Override
    public String someOther(List<Boolean> one, BigDecimal two, BigDecimal three) {
        return null;
    }


}
```

Rys. 7. Klasa wzorcowa

Ostatnim krokiem jest utworzenie klasy samego testu. W przedstawianym przykładzie wykorzystano zgodnie z założeniami projektu będący *de facto* standardem framework testów jednostkowych JUnit. Rozszerzeniem w stosunku do standardowej budowy testu jest konieczność wykorzystania obiektu pośrednika (pole *proxy*), który musi zostać zainicjowany przed wykonywaniem testów (metoda *prepareProxy()*).

```
package pl.lodz.p.edu.mieszkojablonski.aespct.core.domain;

import ...

public class SimpleProxyTest {

    private static Proxy proxy;

    @BeforeClass
    public static void prepareProxy() {
        TestedTypeFactory<SimpleProxy> testedTypeFactory = new TestedTypeFactory<>(SimpleProxy.class);
        SimpleProxyTest.proxy = testedTypeFactory.getProxyInstance();
    }

    @Test
    public void exampleTestedMethod() throws Exception {
        assertEquals( expected: "SUCCESS", proxy.exampleTestedMethod());
    }

    @Test
    public void etn2() throws Exception {
        assertEquals( expected: "6", proxy.someOther(new BigDecimal( val: 2), new BigDecimal( val: 3), new Date()));
        assertEquals( expected: "9", proxy.someOther(new BigDecimal( val: 3), new BigDecimal( val: 3), new Date()));
    }

    @Test
    public void etn3() throws Exception {
        List<Boolean> booleans = new ArrayList<>();
        booleans.add(Boolean.TRUE);
        booleans.add(Boolean.FALSE);
        booleans.add(Boolean.TRUE);
        assertEquals( expected: "60", proxy.someOther(booleans, new BigDecimal( val: 10), new BigDecimal( val: 3)));
    }
}
```

Rys. 8. Test JUnit

Definiowanie klasy testowanej

Twórca klasy testowanej musi odwołać się do archetypu *aespct-impl*, aby móc skorzystać z adnotacji *@ClassUnderTest* oraz *@MethodUnderTest*. Adnotacje te muszą zostać zastosowane zgodnie z wzorcem narzuconym przez definiującego test w formie klasy wzorcowej, w założeniach jawnej. Poza tym nie są wymagane żadne zgodności, w szczególności implementowanie określonych interfejsów czy rozszerzanie określonych klas. Bajtkod dostarczony przez twórcę klasy testowanej zostanie poprzez mechanizm aspektów wykonany zamiast wykonania odpowiednich metod klasy wzorcowej.

```
/.../

package pl.lodz.p.edu.mieszkojablonski.aespct.student.implementation;

import ...

@ClassUnderTest(name = "Klasa_A")
public class StudentClassImpl {

    @MethodUnderTest(name = "Metoda_A")
    public String testMethod() { return "SUCCESS"; }

    @MethodUnderTest(name = "Metoda_B")
    public String testMethodThree(BigDecimal one, BigDecimal two, Date three) {
        return String.valueOf(one.multiply(two));
    }

    @MethodUnderTest(name = "Metoda_C")
    public String testMethodTwo(List<Boolean> one, BigDecimal two, BigDecimal three) {
        BigDecimal m = BigDecimal.ZERO;
        for (Boolean aBoolean : one) {
            if (Boolean.TRUE.equals(aBoolean)) {
                m = m.add(two);
            }
        }
        return String.valueOf(m.multiply(three));
    }
}
```

Rys. 9. Przykładowa klasa testowana

Przykład wykonania

Dzięki wykorzystaniu frameworku JUnit samo wykonanie testu nie odbiega od typowego postępowania. Poniższy listing prezentuje przebieg testu uruchomionego za pomocą zarządcy Maven:

```
-------------------------------------------------------------------------------
Test set: pl.lodz.p.edu.mieszkojablonski.aespct.core.domain.SimpleProxyTest
-------------------------------------------------------------------------------
Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.216 sec –
in pl.lodz.p.edu.mieszkojablonski.aespct.core.domain.SimpleProxyTest
```

Porównując warunki określone w teście (Rys. 8) z definicją klasy wzorcowej (Rys. 7) łatwo zauważyć, że metody klasy wzorcowej nie spełniają warunków testu. Wykonanie testu zakończyło się jednak sukcesem, ponieważ faktycznie testowaniu podlegała klasa o implementacji zgodnej z warunkami testu (Rys. 9). Bezpośrednim dowodem działania prezentowanego rozwiązania jest przebieg wykonania testu w przypadku, gdy z klasy testowanej usunięta zostanie adnotacja *@ClassUnderTest*. Test nie zostaje wówczas wykonany z powodu niemożliwości wyszukania klasy testowanej i dostarczenia jej obiektu.

```
-------------------------------------------------------------------------------
Test set: pl.lodz.p.edu.mieszkojablonski.aespct.core.domain.SimpleProxyTest
-------------------------------------------------------------------------------
Tests run: 1, Failures: 0, Errors: 1, Skipped: 0, Time elapsed: 0.118 sec <<<
FAILURE! -
in pl.lodz.p.edu.mieszkojablonski.aespct.core.domain.SimpleProxyTest
pl.lodz.p.edu.mieszkojablonski.aespct.core.domain.SimpleProxyTest
Time elapsed: 0.117 sec  <<< ERROR!
java.lang.IllegalArgumentException: Could not find any class
annotated with ClassUnderTest
```

5 Podsumowanie

Celem projektu było dostarczenie możliwości testowania jednostkowego nieznanych z góry klas Java bez konieczności zapewniania formalnej zgodności (implementacja interfejsu, rozszerzanie klasy, zgodność identyfikatorów klas czy metod) między klasą wzorcową testu a klasą testowaną. Jedynymi cechami wspólnymi obu klas są: zgodność sygnatur metod (co jest w tym przypadku oczywiste), a także zastosowanie adnotacji *@ClassUnderTest* oraz *@MethodUnderTest* jako informacji łączącej elementy obu klas. Wykonanie testu na z góry nieznanej klasie jest możliwe dzięki wykorzystaniu zaawansowanych paradygmatów programowania dostępnych dla języka Java: programowania aspektowego oraz programowania refleksywnego.

Projekt znajduje się obecnie w fazie dowodu słuszności koncepcji (proof of concept). W krótkoterminowej perspektywie rozwoju planowane są dalsze usprawnienia prowadzące do uproszczenia procedury przygotowywania testu. W perspektywie długoterminowej planowana jest integracja z mechanizmami zarządzania rozwojem oprogramowania, takimi jak repozytorium Git wraz z jego mechanizmem reagowania na zmiany Git Hooks [14] czy też system wsparcia ciągłej integracji Jenkins [15].

Bibliografia

[1] Hunt A., Thomas D., JUnit. Pragmatyczne testy jednostkowe w Javie. Helion, 2006

[2] Beck K., Test Driven Development: By Example. Addison-Wesley, 2002

[3] Aplikacja Codility, https://codility.com/ (dostęp 09.2017)

[4] Aplikacja BaCa, http://baca.ii.uj.edu.pl/ (dostęp 09.2017)

[5] Witryna projektu JUnit, http://junit.org/junit4/ (dostęp 10.2017)

[6] Witryna projektu Maven, https://maven.apache.org/ (dostęp 10.2017)

[7] Colyer A., Clement A., Harley G., Webster M., Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools. Addison-Wesley, 2004
[8] Witryna projektu Eclipse AspectJ, http://www.eclipse.org/aspectj/docs.php (dostęp 10.2017)
[9] Gurrion A., Spring AOP. https://albertogurrion.wordpress.com/spring-aop/ (dostęp 10.2017)
[10] Horstmann C. S., Java. Techniki zaawansowane. Wydanie X. Helion 2017
[11] Gupta A., Java EE 7 Essentials, 1st edition. O'Reilly, 2013
[12] Dokumentacja platformy Java EE 7, https://docs.oracle.com/javaee/7/tutorial/doc/ (dostęp: 10.2017)
[13] Horstmann C. S., Java. Podstawy. Wydanie X. Helion 2016
[14] Chacon S., Straub B., Pro Git. Apress, 2009
[15] Witryna projektu Jenkins, https://jenkins.io/ (dostęp 10.2017)

AUTOMATIC ASSESSMENT OF THE COMPLETENESS OF THE PROGRAMMING PROJECT

Summary – In this article a project and key implementation aspects of library, used for automatic testing of the completeness of implementation of a given problem in Java through unit testing, are presented. Thanks to the use of reflection mechanism and aspect-oriented programming, the presented solution enables standard unit tests to be performed on *a priori* unknown, external classes. The presented library is developed as part of engineering diploma thesis work with the same title.

Keywords: Java, unit testing, aspect-oriented programming, reflective programming