EduAkcja. Magazyn edukacji elektronicznej
nr 1 (13)/2017, str. 95—107

Skrypty w systemie automatycznej oceny rozwiązań zadań programistycznych. Część II. Działanie

Krzysztof Barteczko
Polsko-Japońska Akademia Technik Komputerowych
kb@pjwstk.edu.pl

Streszczenie: System Doskonalenia Kwalifikacji Programistycznych (SDKP), wdrożony w Polsko-Japońskiej Akademii Technik Komputerowych umożliwia nauczycielom akademickim tworzenie skryptów, wspomagających ocenę rozwiązań zadań programistycznych. W poprzedniej części artykułu przedstawiono krótką informację o SDKP na tle innych systemów automatycznej oceny rozwiązań zadań programistycznych oraz główne motywy i założenia użycia skryptów w systemie. W obecnej części bardziej szczegółowo omówione zostaną rodzaje skryptów i sposoby ich tworzenia. Prezentacja skryptów weryfikacyjnych, obejmująca przypadki z realnej praktyki i akcentująca dydaktyczny wymiar procesu oceny rozwiązań, ma na celu dostarczenie Czytelnikowi konkretnych informacji o możliwościach systemu. Artykuł kończy syntetyczny opis zastosowania SDKP w realnym procesie dydaktycznym oraz krótkie podsumowanie.

Słowa kluczowe: nauczanie programowania, automatyczna ocena rozwiązań, statyczna analiza kodu, dynamiczna analiza kodu, testowanie oprogramowania, skrypty, Java, Groovy, metaprogramowanie, analiza AST, refleksja

1. Wprowadzenie

Ważną częścią Systemu Doskonalenia Kwalifikacji Programistycznych (SDKP), zrealizowanego w Polsko-Japońskiej Akademii Technik Komputerowych, są moduły automatycznej oceny rozwiązań zadań programistycznych pod kątem:

- poprawności programów,
- spełniania postawionych w zadaniach wymagań oraz stylu programowania,
- efektywności, uniwersalności, elastyczności, skalowalności rozwiązań.

Specyfikując zadania do wykonania i sprawdzając później ich rozwiązania, nauczyciel może nie tylko korzystać z rozlicznych wbudowanych w system narzędzi weryfikacji (jak w tradycyjnych systemach automatycznej oceny), ale również dostarczać skryptów testujących poprawność wyników oraz jakość kodu. W skryptach można używać bogatych środków przetwarzania informacji, w tym środków statycznej i dynamicznej analizy kodów programów. Skrypty mogą być pisane w języku Java, ale ze względu na składniowe udogodnienia oraz bogactwo dodatkowych bibliotek, preferowana pod tym względem jest platforma języka Groovy (The Groovy Programming Language, 2016). Nauczycielom przedmiotów programistycznych tworzenie skryptów nie powinno sprawiać trudności, a przy tym może znacząco poprawiać jakość analizy rozwiązań.

W poprzedniej części artykułu (Barteczko, 2016) przedstawiono krótką informację o SDKP na tle innych systemów automatycznej oceny rozwiązań zadań programistycznych oraz główne motywy i założenia użycia skryptów w systemie. Czytelnik znajdzie tam również wiadomości na temat rodzajów testów, a także formatów oraz ogólnych zasad definiowania zadań w systemie. W szczególności, w SDKP dostępne są testy oceniające poprawność wyników programów (*run*-testy), scenariusze behawioralne (oceniające poprawność zachowania programów), testy spełniania wymagań oraz testy stylu. Weryfikacja efektywności, uniwersalności, elastyczności

i skalowalności programów dokonywana jest za pomocą kombinacji wyżej wymienionych testów.

Testowanie rozwiązań studentów wymaga przygotowania odpowiednich definicji zadań. Dla przypomnienia, elementy definicji zadania (w formacie YAML) przedstawiono na Schemacie 1.

```
pts:     # max punktów do uzyskania
str:     # struktura zadania (jeśli testy wymagają jej określenia)
  _dir:
    plik: |
      zawartość
    .....
test:
  // ...
  run:   # definicje run-testów
  beh:   # definicje scenariuszy behawioralnych
  req:   # definicje testów spełniania wymagań
  // ..
```

Schemat 1. Wybrane elementy definicji zadania w SDKP

Dla testów typu *run* i *req* skrypty testujące podawane są w definicji pod kluczem *script*, dla testów typu *beh* – pod kluczem *scen*.

2. Skrypty oceniające poprawność rozwiązań na podstawie analizy wyników

Procedura testowania poprawności rozwiązań na podstawie analizy wyników za pomocą skryptów polega na:

- przygotowaniu wariantów danych wejściowych dla programu studenta (mogą być zapisywane w plikach lub podawane jako argumenty programu),
- przygotowaniu odpowiednich wariantów oczekiwanych wyników; warianty danych wejściowych i oczekiwanych wyników stanowią tzw. przypadki testowe,
- dla każdego przypadku testowego – uruchomieniu programu studenta w bezpiecznym środowisku, w którym dozwolone są tylko określone operacje (np. wczytanie konkretnego pliku z danymi wejściowymi, zapis wyników do konkretnego pliku),
- dla każdego przypadku testowego – przejęcie wyników programu studenta (w zależności od specyfikacji testu – z konsoli lub z pliku wyjściowego) i uruchomienie skryptu testującego (zapisanego w definicji testu pod kluczem *script*), który w odpowiednich zmiennych uzyskuje opisy przypadków testowych, wyniki programu studenta oraz oczekiwane wyniki.

Prosty skrypt testujący oraz jego działanie przedstawimy na przykładzie zadania, w którym student ma wczytać tekst z pliku i policzyć liczbę wystąpień poszczególnych słów w tym tekście. Definicję testu dla tego zadania przedstawia Listing 1.

```
pts: 10
# ...
test:
  run:
    claz: Main  # <-- nazwa klasy, od której zaczyna się wykonanie programu
    inputFrom: textforwords.txt
    out: '@console'  # <-- wyjście na konsolę
    data:
      - caseMsg:
          - 0
          - 'dla testowego wejścia 1 - źle policzona liczba wystąpień'
        inp: 'ala ma kota ala ala kota pies pies pies'
        exp: |
          ala 3
          ma 1
          kota 2
          pies 3

      - caseMsg:
          - 0
          - 'dla testowego wejścia 2 - źle policzona liczba wystąpień'
        inp: 'ala! ma? kota: ala,...ala {kota) pies, pies. pies!'
        exp: |
          ala 3
          ma 1
          kota 2
          pies 3
    script: |
      def expsym = exp.tokenize() #1
      def outsym = out.tokenize() #2
      def outjoined = ' '  + outsym.join(' ') #3
      def err = []
      for (int i=0; i < expsym.size()-1; i+=2) { #4
        if (!outjoined.contains(' ' + expsym[i] + ' ' +expsym[i+1] + ' '))
            err << [ -1, caseMsg[1] + " słowa " + expsym[i] ]
      }
      // Sprawdzenie formatu wyjścia:
      // niepuste wiersze winny zawierać słowo a po odstępie lub odstępach - liczbę
      def notempty = out.readLines().findAll { it.trim() != '' }
      if (notempty.find { !(it ==~ /\S+?\s+?\d+\s*/) })  // dopasowanie do regex
          err << [ -1, 'wadliwy format wyjścia' ]
      return err
```

Listing 1. Przykładowy prosty test działania programu

Występują tu dwa przypadki testowe, z danymi wejściowymi zapisanymi pod kluczami *inp*. W pierwszym dane wejściowe nie zawierają znaków interpunkcji, w drugim – do słów dołączone są takie znaki. Dla każdego przypadku testowego dane te są zapisywane do pliku textforwords.txt, po czym uruchamiany jest program studenta, a standardowe wyjście tego programu (wynik na konsoli) przechwytywane i wraz z oczekiwanym wyjściem (*exp*) i opisem przypadku testowego (*caseMsg*) przekazywane do uruchomionego skryptu jako zmienne *out*, *exp* i *caseMsg*. W skrypcie w wierszach #1 i #2 uzyskujemy listy symboli (czyli ciągów znaków nie zawierających odstępów) oczekiwanego i faktycznego wyjścia, a w wierszu #3 – napis powstały z połączenia wszystkich symboli faktycznego wyjścia, rozdzielonych spacją. W pętli #4 sprawdzamy, czy właściwie policzono liczbę wystąpień poszczególnych słów, np. dla słowa „pies", sprawdzane jest, czy napis powstały z połączenia wszystkich symboli faktycznego wyjścia zawiera ciąg znaków " pies 3 ". Za każdy błąd w zliczaniu wystąpień słów odejmowany jest 1 punkt. Zauważmy, że student może dobrze policzyć liczbę wystąpień poszczególnych słów, ale przedstawić

wyniki w nieco innej niż oczekiwana formie, np. poprzedzając każde słowo jakimś dodatkowym napisem. Procedura testująca jest na to odporna i uzna takie wyniki za poprawne, ale niedostosowanie do wymaganego formatu wyjścia powinno być jakoś „ukarane". Dlatego końcowa część skryptu weryfikuje wymagany format wyjścia i w przypadku niezgodności ze specyfikacją obniża punktację o 1.

Skrypty mogą być zapisywane w bazie skryptów i stamtąd pobierane przy wykonywaniu testów. Oczywiście, w przypadku skryptów ściśle dostosowanych do konkretnego zadania nie ma takiej potrzeby i lepszym rozwiązaniem jest dostarczenie ich kodu wraz z definicją zadania. Natomiast skrypty ogólniejsze, dające się zastosować do oceny rozwiązań różnych zadań, warto zapisać jednokrotnie w bazie danych skryptów i odwoływać się do nich poprzez identyfikatory. Dla przykładu, skrypt na Listingu 2 może być zastosowany do oceny rozwiązań wszystkich zadań, które wytwarzają wynik w postaci zestawu wierszy i dla których podany jest oczekiwany zestaw (różnych) wierszy. Przy weryfikacji z każdego wiersza usuwane są odstępy (białe znaki), a kolejność (i ewentualne powtórzenia) wierszy na wyjściu nie mają znaczenia. Punktacja jest wyliczana na podstawie różnicy pomiędzy zestawem wierszy oczekiwanych i wierszy na wyjściu.

```
if (out.trim() == '') return [0, 'brak informacji na wyjściu']
def olines = out.readLines().collect { it.tokenize().join() } as Set #1
def elines = exp.readLines().collect { it.tokenize().join() } as Set #2
def n = elines.size()
def diff = elines - olines                                          #3
if (diff.size() > 0) {
  def p = (int) (caseMsg[0]*diff.size()/n)                            #4
  return [p, caseMsg[1] ]                                           #5
}
else return []
```

Listing 2. Przykład ogólnego skryptu dla oceny rozwiązań wielu różnych zadań.

Komentarze do kodu (zgodnie z oznaczonymi fragmentami skryptu):
#1 – zbiór niepowtarzających się wierszy wyjścia z usuniętymi odstępami – olines,
#2 – zbiór niepowtarzających się wierszy oczekiwanych z usuniętymi odstępami – elines,
#3 – różnica zbiorów (elementy z elines, których brak w olines)
#4 – zmniejszenie punktacji, ustalane jako część maksymalnego zmniejszenia punktacji, podanego w *caseMsg* definicji testu, odpowiadająca odsetkowi brakujących na wyjściu wierszy,
#5 – w przypadku niezgodności zwracane jest obniżenie punktacji i opis błędu, zawarty w *caseMsg* testu.

Załóżmy teraz, że skrypt z Listingu 2 został zapisany do bazy skryptów pod identyfikatorem LINESDIFF. Możemy go teraz użyć w różnych definicjach zadań (na przykład takiej, jak na listingu 3).

```
test:
  run:
    claz: XListImpl
    inputFrom: input.txt
    out: output.txt
    data:
      - caseMsg: [-10, 'wyniki przetwarzania list niezgodne z wymaganymi']
        expInFile: expected.txt
    script: LINESDIFF
```

Listing 3. Zastosowanie skryptu z bazy danych

W tym przypadku oczekiwane wyjście programu zapisane jest w pliku expected.txt. Program studenta na podstawie danych wejściowych (input.txt) generuje plik z wynikami (output.txt). Zawartość plików output.txt i expected.txt jest dostępna w skrypcie w zmiennych *out* i *exp*. Skrypt z Listingu 2 sprawdza poprawność wyniku. Może się teraz okazać, że student wykonał tylko połowę zadania (wyjście jego programu zawiera tylko połowę oczekiwanych wierszy). Wtedy skrypt zwróci informację o błędzie w postaci [-5, 'wyniki przetwarzania list niezgodne z wymaganymi'], a prezentacja rezultatów testu dla studenta będzie zawierać też oczekiwane i faktyczne wyniki, tak by łatwo było odnaleźć błąd.

Definiowanie testów wyników wykonania jest elastyczne i pozwala na różne sposoby testowania. W szczególności, sekcja *data* z przypadkami testowymi może być całkowicie pominięta, a wszystkie czynności związane z oceną rozwiązania i komunikowaniem o błędach – wykonywane w skrypcie testującym. Możliwa jest także chwilowa podmiana wybranych klas w programie studenta dla potrzeb testów, co będzie użyteczne w przypadku, gdy w definicji zadania określono ściśle obowiązującą zawartość jakiejś klasy, a w testach chcemy sprawdzić czy rozwiązanie jest właściwe również dla innych przypadków. Wraz z oceną poprawności wyniku można oceniać czas wykonania programu (czy spełnia zadane ograniczenia), a także definiować skrypty, które same uruchamiają testowane programy i badają ich zachowanie. Ten ostatni sposób weryfikacji rozwiązań dostępny jest także w ramach omawianych dalej zastosowań scenariuszy behawioralnych i tam właśnie zostanie przedstawiony bardziej szczegółowo.

3. Testowanie poprawności rozwiązań za pomocą skryptów – scenariuszy behawioralnych

Moduł scenariuszy behawioralnych służy przede wszystkim do testowania poprawności programów poprzez sprawdzanie, czy poszczególne elementy programu (takie jak np. metody jakiejś klasy) działają we właściwy sposób.

W SDKP odbywa się to w kategoriach testowania zachowania programu z użyciem scenariuszy, wykonywanych w zintegrowanym z systemem środowisku testowania behawioralnego *easyb* (Projekt Easyb, 2016). Scenariusze behawioralne tworzone są przy użyciu DSL (*domain specific language* – wyspecjalizowanego języka dziedzinowego) *easyb*, który wprowadza odpowiednią strukturę testów np.

```
scenario "Wypłaty z konta"
  given "Stan konta" {...}
  when "Wypłacono dopuszczalną kwotę" { ...}
  then "Stan konta zmniejszył się o tę kwotę"   { ... }
  when "Próba wypłaty niedozwolonej lub za dużej kwoty" {...}
  then "Program sygnalizuje błąd" { .... }
  and "Stan konta nie zmienił się" { ...}
```

a w klauzulach *given*, *when*, *then*, and pozwala stosować dowolne konstrukcje języka Groovy, a także ułatwienia DSL *easyb*. Dodatkowo dostępne są konstrukcje where (pozwalająca na łatwe definiowanie tzw. data-driven tests) oraz fail – umożliwiająca arbitralne zgłoszenie niepowodzenia testu dla danej klauzuli *then* lub and. Zamiast klauzul and można stosować kolejne klauzule *then*, dlatego dalej będzie mowa tylko o klauzulach *then*.

System testowania SDKP wykorzystuje opisy w klauzulach *given*, *when*, *then*, do oceny rozwiązań i prezentowania jej studentom. Ogólna postać opisu wygląda następująco:

kontekst@wyjaśnienie błędu@ 0|-n

Za ostatnim separatorem @ znajduje się liczba punktów. Wartość zero oznacza, że przy takim błędzie rozwiązanie uzyskuje 0 punktów, wartości ujemne oznaczają zmniejszenie punktacji przy wystąpieniu danego błędu. Dla klauzul *given* i where wszystkie części opisu są opcjonalne (nawet opis może być pusty). Dla klauzul *then* musi wystąpić przynajmniej wyjaśnienie błędu

i obniżenie punktacji.

Prosty przykład scenariusza behawioralnego rozważymy na przykładzie następującego zadania:

Zadanie.
Napisać program, który symuluje operacje na kontach bankowych.
Konta są obiektami klasy Account. Każde konto ma aktualny stan i można:

- wpłacać pieniądze - metoda deposit(ile),
- wypłacać pieniądze - metoda withdraw(ile),
- przelewać na inne konto - metoda transfer(konto2, ile),
- pobierać aktualny stan konta - metoda getBalance().

Konta posiadają klienci banku (obiekty klasy BankCustomer). W roli klienta występuje osoba (obiekt klasy Person).

Skrypt na Listingu 4 przedstawia scenariusz testujący wpłaty na konto.

```
scenario 'Wplaty na konto',{
  given 'Mamy konto',{
    customer = new BankCustomer(new Person('Ala'))
  }
  when 'przy stanie 0 - wplata to #wplata',{
    customer.getAccount().deposit(wplata)
  }
  then 'stan konta winien byc #stan@-1', {
    customer.getAccount().getBalance().shouldBe stan
  }
  where "Podane dane to", {
    wplata = [100, 100.5,  -10]
    stan = [100, 100.5, 0]
  }
}
```

Listing 4. Scenariusz behawioralny dla wpłat na konto.

Przypadki testowe opisuje klauzula where. Dla każdego przypadku testowego wykonywane są klauzule *when* i *then*.

W klauzuli *given* tworzymy obiekt klasy Bank*CU*stomer za pomocą konstruktora z programu studenta. W klauzuli *when* wywoływane są odpowiednie metody z kodu studenta, mające skutkować wpłatą na konto. Konkretne wpłaty pochodzą z listy *wplata z where* i są kolejno używane przy wywołaniu metody *deposit* oraz wstawiane do opisu w klauzuli *where* w miejsce symbolu #wplata. Klauzula *then* weryfikuje, czy stan konta po próbie wpłaty jest właściwy. Prawidłowe stany konta są podane w *where* jako lista *stan*. Kolejne elementy tej listy zostaną wykorzystane przy weryfikacji i w razie błędu odpowiednia kwota pojawi się w komunikacie w miejscu #stan. W klauzuli *then* wykorzystano jedną z konstrukcji DSL *easyb* – *shouldBe* – która dokonuje weryfikacji i w razie błędu generuje odpowiedni jego opis.

SDKP uruchamia skrypt scenariusza behawioralnego i przejmuje wygenerowane przez niego opisy błędów. Są one zapisywane do bazy danych wyników, a po odpowiednim przetworzeniu – prezentowane studentom (i nauczycielowi). Formę prezentacji wyników skryptu z Listingu 3 przedstawia Rys. 1.

```
Laboratorium: 1, zadanie: 1
Maksimum punktów do uzyskania: 4
_____________________________________________________________

Indeks: xxxx
Autor: aaa bbbb
Punkty uzyskane w wyniku testów: 3
_____________________________________________________________

Wyniki testów zachowania - zmniejszenie punktacji = -1

ZAŁOŻENIE: Mamy konto

GDY: przy stanie 0 - wplata to -10

  - stan konta winien byc 0
  ( oczekiwane 0, a jest -10.0 )
  > zmniejszenie punktacji = -1
```

Rysunek 1. Przykładowe wyniki scenariusza behawioralnego

Jak widać, tutaj student nie uwzględnił, że ujemne wartości wpłat powinny być odrzucane i nie mogą zmieniać stanu konta.

Oczywiście, rozwiązanie całego zadania o kontach bankowych testowane jest za pomocą zestawu scenariuszy (oprócz wpłat również dla wypłat i dla transferów). Zestaw scenariuszy podawany jest pod kluczem *scen*: definicji testu behawioralnego, która ma następującą ogólną postać:

```
beh:
  maxtime: maks. czas wykonania, jeśli różny od domyślnego
  gui: [ true | false ] - czy testowanie GUI, domyślnie false
  scen: skrypt_zestawu_scenariuszy | id_skryptu_w_BD
```

Zauważmy, że skrypty scenariuszy behawioralnych mogą zawierać dowolne konstrukcje. Możliwe jest np. wykorzystanie bibliotek zewnętrznych do testowania graficznych interfejsów użytkownika (wtedy znacznik gui:, mający wartość true, spowoduje dołączenie odpowiednich bibliotek do środowiska uruchomieniowego testów).

Wykonanie scenariuszy behawioralnych wymaga jakiejś specyfikacji zadania: np. że klasa nazywa się *Klasa* i ma metodę *metoda()*. Często przy takiej (choćby częściowej) specyfikacji warto podać fragmenty programu, które student ma uzupełnić tak, by uzyskać oczekiwane wyniki. Dzięki temu będzie on mógł lepiej zrozumieć, w jaki sposób należy wykonać zadanie. Przy tym jednak może się zdarzyć, że dopisany przez studenta kod oraz zdefiniowane przez niego inne klasy programu są właściwe dla uzyskania podanego w zadaniu wyniku, ale nie dość uniwersalne z punktu widzenia ogólnej specyfikacji zadania. Proste testowanie za pomocą scenariuszy behawioralnych może tego nie wykryć, zatem potrzebne są bardziej zaawan-sowane środki.

Rozważmy to na przykładzie następującego zadania.

Zadanie

Stworzyć sparametryzowane interfejsy:

- Selector – z metodą select, zwracającą true, jeśli argument spełnia warunek zapisany w metodzie i false w przeciwnym razie
- Mapper – z metodą map, będącą dowolną funkcją: argument -> wynik

oraz sparametryzowaną klasę ListCreator, zawierającą:

- statyczną metodę collectFrom (lista)
- metodę *when*
- metodę mapEvery

które działają w taki sposób, że zapis:

collectFrom(list1).when(selektor).mapEvery(mapper)

spowoduje utworzenie listy wynikowej, której elementy stanowią wybrane przez selektor elementy listy *list1*, przekształcone za pomocą podanego mappera *mapper*.

Działanie wyjaśnia poniższy, przykładowy fragment programu.

```
import java.util.*;

public class Main {

  public Main() {
    List<Integer> src1 = Arrays.asList(1, 7, 9, 11, 12);
    System.out.println(test1(src1));
  }

  public List<Integer> test1(List<Integer> src) {
    Selector  /*<-- definicja selektora, nazwa zmiennej sel */
    Mapper  /*<-- definicja mappera; nazwa zmiennej map */
    return ListCreator.collectFrom(src).when(sel).mapEvery(map);
  }

  public static void main(String[] args) {
    new Main();
  }
}
```

W kodzie metody *test1* należy uzupełnić fragmenty zaznaczone przez /*<-- */. Innych fragmentów klasy Main nie wolno modyfikować.

Gdy w metodzie *test1* selektor wybiera z listy liczby < 10, a mapper zwraca liczbę-argument po-większoną o 10, to na konsoli powinniśmy zobaczyć:

[11, 17, 19]

Naturalnie, sprawdzenie tego, czy wywołanie metody *test1* zwraca oczekiwany wynik będzie bardzo proste. Ale to nie wystarczy do właściwej oceny rozwiązania. Istota zadania polega na budowie sparametryzowanych interfejsów oraz sparametryzowanej klasy ListCreator, w taki sposób, by metody klasy mogły przetwarzać listy elementów dowolnego typu. Tymczasem student w rozwiązanie może:

- w ogóle nie sparametryzować interfejsów i klasy ListCreator, dostarczając tylko ich definicji w kategoriach tzw. typów surowych – „raw types",
- przy budowie interfejsów i klasy użyć argumentów typu, dostosowanych do konkretnego przypadku metody *test1* (tu argumentem typu będzie Integer).

Zatem skrypt testujący winien zawierać trzy scenariusze sprawdzające:

1. Czy podana metoda *test1* daje wymagane wyniki?
2. Czy interfejsy i klasa ListCreator nie są „surowe" (bez parametrów i/lub argumentów typu)?

3. Czy parametryzacja interfejsów i klasy jest dostatecznie ogólna, tak by móc działać na innych, niż podane w *test1*, typach elementów?

Scenariusz 1 jest banalny, możemy go zatem pominąć. W scenariuszu 2 (zob. Listing 5) w celu sprawdzenia czy interfejsy i klasa są właściwie sparametryzowane zastosujemy środki refleksji. Refleksja oznacza m.in. możliwość uzyskiwania w trakcie działania programu pełnej informacji o charakterystykach klasy, m.in. o polach, konstruktorach, metodach, ale – również – częściowych informacji o parametrach typu (Barteczko, 2015a).

```
def isRightParameterized(claz, closure) {
  def n = claz.getTypeParameters().size()
  closure(n)
}

scenario "Scenariusz B", {
  when "Sprawdzamy czy jest właściwa parametryzacja klas/interfejsów", {  }
  then "Selector winien mieć jeden parametr typu @-1", {
     if (!isRightParameterized(Selector.class, { n->  n == 1 })) {
         fail "a nie ma"
     }
  }
  then "Mapper winien mieć dwa parametry typu @-1", {
     if (!isRightParameterized(Mapper.class, { n->  n == 2 })) {
         fail "a nie ma"
     }
  }
  then "ListCreator winien mieć co najmniej jeden parametr typu @-1", {
     if (!isRightParameterized(ListCreator.class, { n->  n >= 1  })) {
         fail "a nie ma"
     }
  }
}
```

Listing 5. Scenariusz weryfikujący parametryzację interfejsów i klasy

W skrypcie wykorzystujemy metodę *fail* z DSL easyB, która pozwala zgłosić niepowodzenie danego przypadku testowego (z *then*) i podać odpowiedni komunikat, który będzie widoczny w wynikach testu.

Warta uwagi jest też swoboda definiowania skryptów scenariuszy behawioralnych. Na przykład, w skryptach możemy definiować własne funkcje. Tu zdefiniowano funkcję o nazwie *isRightParameterized*, pomocną w realizacji scenariusza. Podczas stosowania metody *getTypeParameters()* ze standardowej biblioteki Javy uzyskiwana jest informacja o zestawie parametrów/argumentów typu danej klasy czy interfejsu i za pomocą przekazanego domknięcia (*closure*) sprawdza, czy ich liczba jest zgodna z wymaganiami. W efekcie zmniejszenia punktacji dotyczy tych rozwiązań, w których niewłaściwie sparametryzowano interfejsy i klasę.

Ale wciąż nie jest to wystarczające do oceny rozwiązana. Niestety, „biblioteczna" metoda *getTypeParameters()*, użyta w naszej funkcji *isRightParameterized*, nie rozróżnia parametrów od argumentów typu. Dla wyjaśnienia: parametr typu to – ogólnie – dowolny typ, argument typu – to właśnie konkretny typ podstawiany w miejsce parametru. Przykładowo, metoda *getTypeParameters()* użyta wobec definicji List<T> i List<Integer> zwróci zestaw „parametrów typu": w pierwszym przypadku [T], a w drugim [Integer]. Oczywiście, w tej sytuacji tylko T jest parametrem typu, natomiast Integer jest argumentem typu, który został w danym momencie użyty. Wobec tego nasza metoda *isRightParameterized* zaakceptuje i taki przypadek: interface Mapper<Integer , Integer > { ... }. Tymczasem chcielibyśmy, by program studenta był przygotowany na dowolne transformacje, a nie tylko Integer → Integer. Jak można to sprawdzić?

Najprostszym rozwiązaniem tego problemu byłaby zamiana treści metody *test1* na taką, w której zostałyby użyte inne argumenty typu. Ale przecież metoda *test1* jest już zdefiniowana w skompilowanym programie studenta. Nie mamy możliwości modyfikacji kodu źródłowego i ponownej jego kompilacji. Na pomoc przychodzą środki metaprogramowania platformy Groovy (The Groovy Programming Language, 2016). Groovy pozwala na łatwe modyfikacje istniejących klas (w tym „gotowych" klas Javy), a nawet na modyfikację *ad hoc* sposobów komunikowania się z konkretnym obiektem. Te modyfikacje dotyczą m.in. dodawania nowych metod lub zastępowania treści metod istniejących.

Wobec tego nie sprawi nam kłopotu przygotowanie scenariusza, w którym metoda *test1* zostanie zmodyfikowana w taki sposób, by sprawdzić, czy Selector, Mapper oraz ListCreator dobrze działają na zupełnie innych typach danych niż podano w przykładowym programie w treści zadania. Definicję metody *test1* w rozwiązaniu studenta zamienimy nową definicją podaną w teście (Listing 6).

```
scenario "Scenariusz C",{
  def main = new Main()
  when "czy rozwiązanie jest przygotowane na inne implementacje interfejsów?", {
  }
  then "dla Selector<Double> s= d->d>100; Mapper<Double,String> m=d->'L'+d@-5", {
    main.metaClass.test1 = { List list ->                    #1
      Selector sel = {Double d -> d > 100 } as Selector
      Mapper map = {Double d -> 'L'+d } as Mapper
      ListCreator.collectFrom(list).when(sel).mapEvery(map)
    }
    try {
      def r = main.test1([111.111, 20.20, 333.333])
      if (r != ['L111.111', 'L333.333']) {
        fail "nie, test1([111.111, 20.20, 333.333]) zwraca " + r +
        ", zamiast [L111.111, L333.333]"
      }
    } catch(Exception exc) {
       fail "nie, test1([111.111, 20.20, 333.333]) powoduje wyjątek: " + exc
    }}}
```

Listing. 6. Metaprogramowanie w skryptach

Na Listingu 6 kluczowe jest odwołanie *main.metaClass.test1* = { ... }. Powoduje ono zamianę aktualnej definicji metody *test1* na nową, której kod podano po znaku równości. Taka możliwość oznacza, że „zamrożone" w specyfikacji zadania fragmenty kodów mogą być dynamicznie zastępowane w trakcie wykonania programu przez kody podane w skrypcie. Jest to przydatna właściwość, pozwalająca łatwo budować bardzo elastyczne i uniwersalne testy.

4. Skrypty testujące spełnianie wymagań

Zauważmy, że przy okazji ostatniego zadania i skryptów testujących rozwiązanie, weryfikacji poddawane były nie tylko konkretne wyniki, ale i spełnianie wymagań postawionych w zadaniu (interfejsy i klasy winny być sparametryzowane w taki sposób, by możliwe było działanie na dowolnych typach danych). Spełnianie wymagań może być weryfikowane za pomocą skryptów-scenariuszy behawioralnych, ale dostępny jest też moduł testujący, który weryfikuje to na podstawie analizy kodów źródłowych. Weryfikacja spełniania wymagań za pomocą list wyrażeń regularnych, sprawdzających, czy w podanych źródłach występują określone fragmenty tekstu, była szczegółowo omówiona w artykule autora (Barteczko, 2015b). Tamże zasygnalizowano możliwość użycia skryptów do weryfikacji spełniania wymagań. Bez powtarzania tych treści, warto tu tylko pokazać dwa przykładowe skrypty testujące. Pierwszy, niezwykle prosty, pozwala

na weryfikację spełnienia wymagania, aby w kodzie rozwiązania nie używać więcej niż jednej instrukcji try (Listing 7).

```
// zmienna src zawiera stokenizowany kod programu
// tokenizacja wyróżniła poszczególne słowa
if (src.count(' try ') > 1)
  return [ -2, 'liczba instrukcji try w rozwiązaniu > 1' ]
```

Listing 7. Prosty skrypt testujący spełnianie wymagań

W skryptach można stosować analizę drzewa składniowego kodów źródłowych (AST). W tym celu z SDKP zintegrowano bibliotekę typu open-source o nazwie JavaParser (Java Parser and Abstract Syntax Tree, 2016).

Załóżmy teraz, że należy sprawdzić, czy w rozwiązaniach studenckich zastosowano polimorfizm w hierarchii dziedziczenia klas programu. Wymaganie to uznamy za spełnione, jeśli w jakiejkolwiek klasie rozwiązania przedefiniowano lub implementowano metody z innej klasy rozwiązania albo przedefiniowano metody domyślne (ze specyfikatorem *default*) z dowolnego występującego w rozwiązaniu interfejsu. Implementacje abstrakcyjnych metod interfejsu w klasach nie będą traktowane jako zastosowanie polimorfizmu.

W skrypcie, pod zmienną *src*, będzie tym razem dostępna mapa, w której klucze, stanowiące nazwy plików, wskazują na zbudowane przez JavaParser drzewa składniowe źródeł tych plików (inaczej zwane jednostkami kompilacji – *CompilationUnits* – *CU*).

Od mapy *src* możemy pobrać kolekcję wartości (kolekcję *CU*) i dla każdej takiej jednostki kompilacji znaleźć wszystkie deklaracje klas i interfejsów, a dla każdej takiej deklaracji zidentyfikować dziedziczone klasy i implementowane interfejsy, a także odnaleźć wszystkie deklaracje metod. Informacje o dziedziczeniu i implementacji interfejsów zostaną zapisane w mapie *emap* (klucz = nazwa klasy, wartość = lista dziedziczonych przez nią klas i implementowanych interfejsów, a mapa *mmap* pod kluczem równym nazwie klasy lub interfejsu będzie zawierać listę sygnatur metod z tej klasy lub tego interfejsu. Dla interfejsów do *mmap* będziemy dodawać tylko metody domyślne. Po zebraniu tych informacji, dla stwierdzenia występowania polimorfizmu, wystarczy określić, czy którakolwiek z list sygnatur metod ma część wspólna z inną taką listą. Kod skryptu przedstawiono na Listingu 8.

```
def emap = [:].withDefault { [] }
def mmap = [:].withDefault { [] }

// Zbieranie informacji
src.values().each { cu ->
  for (TypeDeclaration dcl in cu.getTypes()) {
    if (dcl instanceof ClassOrInterfaceDeclaration) {
      def className = dcl.getName()
      def superTypes = []
      if (dcl.getExtends()) superTypes = dcl.getExtends()*.getName()
      if (dcl.getImplements())  superTypes += dcl.getImplements()*.getName()
      emap[className] = superTypes
      for (mem in dcl.getMembers()) {
        if (mem instanceof MethodDeclaration) {
          if (!dcl.isInterface() || mem.isDefault())
            mmap[className] << mem.name + mem.parameters*.type
        }
      }
    }
  }
}
// Sprawdzenie czy występuje polimorfizm
for (entry in emap) {
  def clasmets = mmap[entry.key]
  for (superType in entry.value) {
    def supermets = mmap[superType]
    if (supermets && clasmets.intersect(supermets)) return [] // tak, pusta errlist
  }
}
return [-2, 'brak polimorfizmu w klasach rozwiązania'
```

Listing 8. Analiza AST przy weryfikacji spełniania wymagań

5. Praktyka zastosowania SDKP i skryptów testujących

SDKP jest stosowany w nauczaniu przedmiotów programistycznych w PJATK już od dwóch semestrów. Szczegółowa analiza wyników użycia SDKP w procesie dydaktycznym to oczywiście materiał na odrębny artykuł. W kontekście obecnego artykułu, niejako dla ogólnego potwierdzenia praktycznej użyteczności proponowanych rozwiązań, syntetycznie przedstawione zostaną dane, ilustrujące bieżące zastosowanie SDKP (zob. Tabela 1), a także opinie nauczycieli, uzyskane w trakcie wielu dyskusji.

Tabela 1. Zastosowanie SDKP w PJATK w okresie 1 marca – 15 kwietnia 2016 r.

Liczba definicji zadań w systemie	136
Liczba definicji zadań w systemie, zawierających testy	110
Liczba definicji zadań w systemie, zawierających testy ze skryptami	69
Liczba przedmiotów prowadzonych z użyciem SDKP	8
Liczba nauczycieli prowadzących zajęcia z użyciem SDKP	7
Liczba wydanych zadań	54
Liczba wydanych zadań z testami	52
Liczba wydanych zadań z testami, zawierającymi skrypty	43
Liczba studentów, przekazujących rozwiązania	647
Liczba przekazanych rozwiązań	5645

Liczba przekazanych rozwiązań, podlegających testom	5519
Liczba przekazanych rozwiązań, testowanych skryptami	4456
Liczba wykrytych błędów w formalnej strukturze projektów	176
Liczba wykrytych błędów w kompilacji	451
Liczba wykrytych błędów w rozwiązaniach testowanych bez skryptów	644
Liczba wykrytych błędów w rozwiązaniach testowanych skryptami	3342

Jak widać, definicje zadań ze skryptami testującymi są chętniej używane przez nauczycieli i pozwalają na wykrycie większej liczby błędów niż w przypadku testów nie zawierających skryptów (relacja liczby wykrytych błędów do liczby rozwiązań wynosi w pierwszym przypadku ok. 0,75, a w drugim ok. 0,61, co oczywiście nie znaczy, że 75 czy 61 procent rozwiązań zawierało błędy).

Właśnie możliwość lepszej identyfikacji błędów i lepszego ich wytłumaczenia studentom była w dyskusjach nauczycieli traktowana jako główny atut skryptowego testowania. Zwracano także uwagę na inną zaletę tego podejścia: możliwość nieostrych specyfikacji zadań do wykonania (a więc pobudzania twórczego myślenia studentów) przy jednoczesnym dostarczeniu testów, które wykrywają wady rozwiązań. W opinii nauczycieli istotna okazuje się też łatwość edycji skryptów w przypadkach, gdy wymagają one dopracowania.

Tworzenie nowych definicji zadań ze skryptami testującymi oceniane jest przez nauczycieli jako pracochłonne (co oczywiście dotyczy wszelkiego konstruowania testów jakiegokolwiek oprogramowania). Sytuację łagodzi możliwość użycia bardzo dobrze im znanego języka Java oraz stopniowe wprowadzanie do skryptów konstrukcji języka Groovy, znacznie upraszczającego programowanie.

Naturalnie, w dyskusjach pojawiały się też postulaty doskonalenia SDKP, w szczególności dotyczące schematów punktacji, a także rozwiązań technicznych.

6. Podsumowanie

Wśród różnorodnych, bogatych w możliwości środków automatycznej oceny rozwiązań zadań programistycznych w SDKP szczególną role pełnią skrypty.

Pozwalają one na testowanie rozwiązań pod różnymi kątami, w różnych trudnych przypadkach, na przykład:

- dla nie całkiem precyzyjnych specyfikacji zadań (m.in. w zakresie formatu wyjścia, ale nie tylko),
- dla zaawansowanych zadań (na przykład z zakresu zaawansowanego programowania obiektowego, programowania współbieżnego czy graficznych interfejsów użytkownika).

Dzięki temu istotnie poszerza się z obszar tematyczny zadań, których rozwiązania mogą być poddawane automatycznej weryfikacji. Również dzięki skryptom informacje o błędach popełnianych przez studentów mogą być bardziej szczegółowe.

Ważne jest również to, że skrypty do weryfikacji rozwiązań mogą być tworzone samodzielnie przez nauczycieli, tak aby były dostosowane do ich potrzeb. Jasne jest, że definiowanie skryptów w językach Groovy/Java nie będzie trudne dla osób, które prowadzą zajęcia z języka programowania Java.

Łatwość, elastyczność i moc skryptowego testowania rozwiązań zadań programistycznych jest jedną z cech wyróżniających SDKP wśród systemów automatycznej oceny.

7. Bibliografia

1. Barteczko, K. (2016). Skrypty w systemie automatycznej oceny rozwiązań zadań programistycznych. Część I. Kontekst i motywacje. Magazyn Edukacji Elektronicznej EduAkcja, 2(12).
2. Barteczko, K. (2015a). Java. Uniwersalne techniki programowania (rozdz. 13. Dynamiczna Java). Warszawa: Wydawnictwo Naukowe PWN.
3. Barteczko, K. (2015b). Weryfikacja stylu programowania w rozwiązaniach zadań programistycznych. Ma-

gazyn Edukacji Elektronicznej EduAkcja, 2(10).

4. Java Parser and Abstract Syntax Tree (2016). Pobrano z: http://javaparser.org
5. Projekt Easyb (2016). Pobrano z: http://easyb.org, https://github.com/easyb
6. The Groovy Programming Language (2016). Pobrano z: http://groovy-lang.org

Scripts in Automatic Assessment of Programming. Assignments. Part II. Action

Keywords: teaching programming, automatic assessment of programming assignments, static code analysis, dynamic code analysis, software testing, scripts, Java, Groovy, metaprogramming, AST, reflection

Abstract: In the Programming Skills Development System (PSDS), created and implemented in the Polish-Japanese Academy of Information Technology, teachers can define scripts for automatic assessment of programming exercises. Advanced tools for static and dynamic code analysis are available for scripts definition. Writing any *script* is easy for teachers of programming languages, yet using it in process of automatic assessment could improve the quality of solutions analysis.