EduAkcja. Magazyn edukacji elektronicznej
nr 1 (3)/2012, str. 57—69

Zadania programistyczne w zdalnym nauczaniu

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

Streszczenie: Artykuł omawia cele nauczania programowania, związane z nimi rodzaje zadań programistycznych, kryteria oceny rozwiązań oraz sposoby ich sprawdzania. Wieloaspektowość oceny wymaga, szczególnie w zdalnym nauczaniu programowania, użycia specjalnych narzędzi sprawdzających. Rozważane są możliwości i metody ich automatycznego stosowania. Zaproponowano integrację narzędzi automatycznej oceny w kompleksowym systemie, mającym na celu – głównie poprzez bogatą w treści interakcję ze studentami – znaczące podniesienie jakości zdalnego kształcenia w zakresie języków i metod programowania. Prezentowane w artykule przykłady w większości dotyczą programów i narzędzi platformy Java.

Słowa kluczowe: zadania programistyczne; programowanie „agile"; wzorce projektowe; wykrywanie plagiatów; testowanie aplikacji, styl programowania, metryki programu

1. Programowanie: czego uczyć, co i jak oceniać?

Umiejętność programowania jest wieloaspektowa. Łączy ze sobą poznanie dyscypliny naukowej, jaką jest informatyka, z cierpliwie praktykowanym i doskonalonym rzemiosłem, a nawet sztuką. Jest to wyjątkowe połączenie. Programowanie można postrzegać jako prawdziwą, podobną muzyce, poezji czy malarstwu twórczość (Knuth, 1974; Hunt, Thomas, 2001), a nawet – przez wielką elastyczność tworzywa, którym jest kod programu – jako niemal magiczne medium kreacji skomplikowanych koncepcyjnie struktur. Właściwe zaklęcie wprowadzone z klawiatury natychmiast tworzy światy, których dotąd nie było, a nawet być nie mogło (Brooks, 1995).

Jednocześnie programowanie nie jest sztuką dla sztuki. Służy ono przede wszystkim wytworzeniu funkcjonalnych, spełniających konkretne wymagania produktów. Co więcej, magia przestaje działać, gdy w programie popełniono błędy, kolejne poprawki wprowadzają nowe, utrzymanie i rozwój aplikacji są utrudnione przez nieczytelność kodu i jego wady strukturalne. Wtedy bajkowe tworzenie wirtualnych światów przeradza się w koszmar zbyt drogich, opóźnionych, wreszcie – niezrealizowanych projektów (Brooks, 1995).

To połączenie swobody twórczości z rygorami wymagań jest w programowaniu chyba najciekawsze. Może stanowić problem (na co zwraca uwagę Frederick Brooks), ale także jest wyznacznikiem prawdziwej umiejętności. Jak powiedział Andy Hertzfeld: *Programowanie jest jedynym zajęciem, przy którym mogę być jednocześnie inżynierem i artystą. Jest w nim niewiarygodnie ścisły techniczny element, który lubię, bo zmusza do bardzo precyzyjnego myślenia. A z drugiej strony jest tu pole dla olbrzymiej kreatywności, której ograniczeniem jest tylko wyobraźnia* (Lammers, 1986).

Myślę, że w nauczaniu programowania należy zwracać baczną uwagę na obie jego strony: tę twórczą i tę inżynieryjną. Nie zawsze zresztą można je ściśle od siebie oddzielić. Mamy pewne zasady, reguły, wzorce, należące do świata inżynierii programowania. Niewątpliwie student musi je poznać. Ale to wciąż nie przesądza o ich właściwym doborze czy zastosowaniu, a ich zastosowanie nie przesądza o jakości programu. Wytworzenie dobrego programu, nawet w oparciu o dużą

wiedzę „inżynieryjną", wciąż jest sztuką, a nie zautomatyzowanym zastosowaniem takich czy innych gotowych rozwiązań[1].

Co stanowi o jakości programu, jak dobrze programować? – na te pytania starali się odpowiadać już klasycy informatyki (choćby przywoływani Donald Knuth i Richard P. Gabriel). Znajdziemy też wiele świetnych książek, prezentujących zasady dobrego programowania, reguły o wdzięcznych nazwach DRY („nie powtarzaj się") czy KISS („utrzymuj prostotę") i te dotyczące np. abstrakcji, otwartości, współzależności, spójności kodu (McConnell, 2004; Martin, 2008). Również metodologie inżynierii oprogramowania zawierają wskazówki co do kodowania; szczególnie dotyczy to metodologii nowych, skupionych wokół ogólnego ruchu *Agile programming* (Beck, 2001; Beck, Andres, 2004; Poppendieck, Poppendieck, 2003; Shore, Warden, 2008).

Nie ma tu miejsca na szczegółową dyskusję tych zagadnień. Jednak w oparciu o wymienione źródła, a także własne doświadczenia, można zaproponować następującą listę umiejętności, które należałoby kształcić w trakcie nauki programowania[2]:

1. Twórcze projektowanie rozwiązań problemów.
2. Umiejętność doboru i stosowania właściwych konstrukcji językowych.
3. Umiejętność doboru i stosowania właściwych środków (bibliotek, szkieletów, wzorców projektowych).
4. Trzymanie się specyfikacji i spełnianie wymagań.
5. Pisanie poprawnego kodu z uwzględnieniem granicznych przypadków.
6. Pisanie kodu dobrej jakości (czytelność, zgodność ze wspomnianymi wcześniej zasadami dobrego programowania).
7. Pisanie uniwersalnego i elastycznego kodu:
   – o słabych powiązaniach pomiędzy różnymi częściami (*loose coupling*),
   – kodu przygotowanego na zmiany wymagań i funkcjonalności,
   – skalowalnego.
8. Pisanie kodu efektywnego.

Projektowanie rozwiązań (p. 1) oznacza, że zanim student przystąpi do programowania, musi zaplanować algorytmy, strukturę aplikacji, interfejs użytkownika itp. Jest ważne, by w dzisiejszych czasach łatwych w użyciu środowisk programistycznych (IDE) uświadomić studentom potrzebę takiego działania (zwykłą praktyką studentów jest postępowanie wedle następującego schematu: „mam problem do rozwiązania → otwieram IDE → metodą prób i błędów piszę program"). Nie wymagajmy przy tym budowy najpierw zamkniętego projektu, a później przekształcania go w działający program. To może, a nawet powinna być procedura iteracyjna: jest wstępny projekt → kodowanie, testowanie, wdrażanie wykazują jego braki → następuje uściślanie, doskonalenie projektu → po czym refaktoryzacja i uzupełnianie kodu itd. (Fowler, 2004). Pozwólmy studentom w tym zakresie pozostać na razie metodologicznie „zwinnymi" (*agile*), sprzyja to bowiem akcentowaniu właśnie procesu programowania.

Umiejętność doboru środków wyrazu (p. 2 i 3) wydaje się oczywista, ale wielokrotnie można obserwować, jak studenci stosują niewłaściwe konstrukcje językowe (np. niewłaściwy

[1] Richard P. Gabriel (Gabriel, 1996) dobrze opisał nieco zbyt optymistyczne próby przeniesienia na grunt oprogramowania koncepcji wzorców projektowych wybitnego architekta Christophera Alexandra. W oprogramowaniu do dziś zastosowanie wzorców projektowych *per se* nie przyczynia się do osiągnięcia czegoś, co Alexander nazywa *quality without a name*, a co odróżnia dobre pod każdym względem konstrukcje od tych nieudanych (Alexander, 1979). Można mieć też wątpliwości, czy w architekturze praktyczne zastosowanie jego koncepcji języka wzorców jako dostępnego dla każdego narzędzia tworzenia „jakości bez nazwy" ma rację bytu. Dobra architektura jest sztuką i dobre programowanie też.

[2] Programowanie poprzedza przedmioty związane z bardziej ogólnie rozumianą inżynierią oprogramowania, bowiem umiejętność programowania (i to w wielu językach) jest wymaganiem wstępnym nauki inżynierii oprogramowania. W związku z tym skupiamy się tu raczej na nauce dobrego kodowania, z uwzględnieniem elementów projektowania rozwiązania problemu.

rodzaj instrukcji decyzyjnych) lub wyważają otwarte drzwi, programując funkcjonalności już zawarte w standardowych bibliotekach. Głęboka wiedza o języku i jego platformie (w tym: dobrych bibliotekach zewnętrznych) nie jest trywialna. Wymagania, już nie tylko co do wiedzy, ale i sztuki programowania rosną, gdy chodzi o dobór wzorców projektowych czy najlepszych szkieletów (*frameworks*).

Trzymanie się specyfikacji i spełnianie postawionych wymagań (p. 4) jest dla wielu studentów trudne. Często okazuje się, że nadesłane rozwiązanie dotyczy nieco innego problemu, albo że nie wszystkie postawione wymagania funkcjonalne są w nim uwzględnione. Powodem może być nie tylko brak umiejętności („robię to co umiem, a nie co miało być zrobione"), ale też nieuwaga, brak skrupulatności, nadmierna fantazja. Wydaje się zatem, że na ten aspekt nauki programowania należy zwrócić baczną uwagę.

Poprawność kodu (p. 5) jest niewątpliwie kluczowa. Nie chodzi przy tym o to, by program działał i podawał jakiś poprawny wynik. Zazwyczaj studenci kończą swoje testowanie rozwiązania właśnie na tym etapie, tymczasem pozostaje wiele przypadków (np. błędne dane lub działanie w środowisku współbieżnym), dla których program jest wadliwy. Należy zatem wymagać sprawdzenia poprawności kodu dla wielu przypadków, a być może także proponować studentom oparte na testach metody tworzenia i rozwijania programów.

Dobra jakość kodu (p. 6), m.in. jego czytelność, a zarazem zwięzłość, zgodność z zasadami i praktykami dobrego programowania jest ważna dla jego dalszych modyfikacji, związanych czy to z usuwaniem błędów, czy z rozwojem aplikacji.

Kod dobry – to również kod uniwersalny i elastyczny (p. 7). Przede wszystkim przygotowany na ewentualne zmiany wymagań czy wprowadzanie dodatkowych funkcjonalności, co szczególnie akcentuje ważność zasad *loose coupling* i otwartości na modyfikacje i uzupełnienia[3]. Chodzi też o skalowalność rozwiązania: program winien dobrze działać przy dużym obciążeniu (duże zestawy danych, duża liczba konkurencyjnych wątków itp.). Należy podkreślić, że omawiane tu wymagania można stawiać raczej większym, bardziej zaawansowanym, zadaniom w późniejszych etapach nauki programowania.

Efektywność kodu (p. 8) nie może być w nauce programowania stawiana na pierwszym miejscu, zarówno pod względem znaczenia, jak i kolejności działań (Knuth, 1974). Na pewno jednak należy zwracać uwagę na bardzo nieefektywne rozwiązania i pokazywać, jak ich unikać.

Proponowane tu osiem punktów–celów nauczania programowania musi znaleźć odbicie w różnych rodzajach zadań, przygotowywanych dla studentów. Nie każde zadanie w równym stopniu będzie realizowało każdy z wymienionych punktów, nie w każdej fazie nauki każdy z tych punktów będzie tak samo istotny. Ale w całym zestawie kursów programistycznych powinniśmy dążyć do uwzględnienia wszystkich punktów z przedstawionej listy.

Zdalne nauczanie ma oczywiście znane wyróżniające cechy. Zatem przygotowanie zadań winno być szczególnie staranne, często wsparte wizualizacjami (np. dokładnie ilustrującymi wymagania wobec programu) i innymi materiałami pomocniczymi (podpowiedzi, analogiczne kody, przykładowe zestawy testów jednostkowych, odsyłacze do odpowiednich zasobów Internetu). Interakcja ze studentami oprócz zwyczajowych czatów czy komunikacji mailowej winna obejmować też bardziej sformalizowany i zarazem dogłębny proces oceny nadesłanych rozwiązań oraz udostępniania tych ocen wraz z komentarzami.

Kryteria oceny rozwiązań wynikają bezpośrednio z przedstawionych ośmiu celów nauczania (punkty te stanowią też kryteria oceny). Taka kompleksowa ocena nie jest łatwa nawet dla pojedynczego zadania. Szczególnie w zdalnym nauczaniu (gdy mamy kontakt z dużą liczbą

[3] Znaczenie tego postulatu w dużej skali dobrze ilustruje ponad 10-letnia (wciąż nie zakończona) historia informatyzacji ZUS. Problemem były tu oczywiście często zmieniające się przepisy prawne, ale być może bardziej uniwersalna, elastyczna i otwarta koncepcja KSI ZUS (a zatem i bardziej uniwersalny i elastyczny kod oprogramowania) mogłyby przyczynić się do skrócenia czasu realizacji projektu oraz jego kosztów.

studentów) wymaga zastosowania dodatkowych narzędzi oraz – przynajmniej częściowej – automatyzacji procesu oceny i indywidualnego komentowania poszczególnych rozwiązań.

W dalszej części artykułu rozważone zostaną możliwości zastosowania różnych podejść i narzędzi do automatyzacji procesu oceny rozwiązań zadań programistycznych. Wcześniej jednak warto poświęcić kilka słów problemowi plagiatów w programowaniu, ponieważ przed przystąpieniem do merytorycznej oceny rozwiązań powinniśmy wyeliminować te z nich, o których na pewno wiadomo, że zostały powielone.

2. Kilka słów o plagiatach i walce z nimi w zdalnym nauczaniu

Plagiat w programowaniu polega na przedstawianiu zapożyczonych fragmentów kodu programu – po ewentualnej, nie wymagającej zbyt dużej wiedzy ani zbytniego wysiłku transformacji – jako kodu własnego autorstwa[4]. Jest to wielki problem w nauce programowania[5]. Można powiedzieć, że w zdalnym nauczaniu programowania problem jest szczególnie ostry. Inaczej niż w trybie stacjonarnym, prowadzący zajęcia nie ma bezpośredniego kontaktu ze studentami w trakcie wykonywania przez nich zadań, zatem pozbawiony jest całego arsenału środków obserwacyjnych, które mogłyby zapobiegać ściąganiu. Pozostaje wyłącznie analiza nadesłanych przez studentów rozwiązań.

Istnieje szereg podejść i narzędzi do wykrywania podobieństw kodu, m.in.:

- porównywanie metryk, takich jak liczba linii, słów, znaków czy związanych ze złożonością – np. metryk Haelsteada oraz kombinacji tych miar (Jones 2001, Halstead 1977),
- statyczna analiza składniowa, polegająca na takiej czy innej tokenizacji kodów i zastosowaniu wobec przekształconego kodu algorytmów wykrywania podobnych fragmentów; przykłady takich algorytmów to *greedy string tiling* (Wise, 1993), który jest używany np. w rodzinie programów YAP oraz programie CPD, czy *winnowing* (Schleimer et al., 2003), używanym w systemie MOSS,
- analiza AST[6] (Li, Zhong, 2010),
- zastosowanie PDG[7] (Liu et al., 2006).

Głównym problemem w zastosowaniu tych podejść/narzędzi do wykrywania plagiatów przy zdalnym nauczaniu programowania nie jest nawet to, że większość z nich nie jest odporna na bardziej zaawansowane przeformułowania ściągniętego kodu (np. dopisanie nie mających znaczenia wierszy). Prawdziwy problem polega na tym, że mogą one dawać fałszywie pozytywne wyniki. Bardzo proste zadania, szczególnie te z zakresu programowania obiektowego, skutkują w bardzo podobnych kodach programów bez żadnych złych intencji ze strony studentów. Studenci korzystają z podobnych materiałów dydaktycznych, skąd czerpią przykłady rozwiązań, a często też współpracują (co należy oceniać pozytywnie) i na skutek takiej współpracy nad koncepcjami rozwiązań mogą pojawić się i bez ściągania dość podobne kody źródłowe. Rozwiązania tego samego zadania, szczególnie z początkowego kursu programowania, z natury rzeczy będą do siebie dość podobne (Mann, Frew 2006).

Warto w tym kontekście podkreślić, że na etapie automatycznej oceny rozwiązań należałoby wykrywać i eliminować tylko nie budzące żadnych wątpliwości przypadki plagiatów, a zatem:

- dla krótkich, prostych zadań – porównywać literalnie całość kodu i/lub szukać znamion

[4] Nie jest to do końca precyzyjna definicja, zresztą na temat różnych jej uściśleń toczą się w literaturze dyskusje (Johnston, 2003). Na potrzeby tego artykułu nieco intuicyjne pojęcie plagiatu w programowaniu wydaje się jednak wystarczające.

[5] Jak wykazały badania na uniwersytetach Monash i Swinburne, w Australii ponad 70% studentów pierwszego roku kierunków IT popełnia plagiaty, wykonując postawione zadania (Sheard et al., 2003).

[6] AST (Abstract Syntax Tree) to rodzaj drzewa, przedstawiającego strukturę programu komputerowego.

[7] PDG (Program Dependency Graph) to graf obrazujący zależności pomiędzy danymi i przepływem sterowania w programie.

plagiatu w takim samym (niezgodnym z zasadami) formatowaniu[8], takich samych komentarzach, takich samych błędach kompilacji,
- dla większych zadań stosować porównanie struktury kodów, odporne tylko na zmiany identyfikatorów i literałów.

Bardziej zaawansowane metody można stosować wyłącznie w celu oznaczenia podejrzanych rozwiązań do wyjaśnienia na zaliczeniu bezpośrednim. Jednak nawet najlepsze narzędzia wykrywania plagiatów mogą działać tylko na ograniczonej bazie do porównań: tu – zestawie nadesłanych rozwiązań tego samego zadania i ewentualnie gdzieś wcześniej udostępnionych rozwiązaniach prawidłowych. Studenci mają do dyspozycji ogromne (szczególnie w zakresie programowania) zasoby Internetu i często ściągają pasujące w danym kontekście fragmenty rozwiązania.

Warto też pamiętać, że w zdalnym nauczaniu tak naprawdę nie wiemy, kto jest autorem nadsyłanych rozwiązań (może ktoś wynajęty), zatem wszystko i tak musi rozstrzygać się poprzez bezpośrednie sprawdzenie wiedzy i umiejętności studenta na zaliczeniu.

3. Poprawność rozwiązań

Automatyczna weryfikacja poprawności programów nie jest zadaniem łatwym, a w ogólnym przypadku jest to na razie zadanie niewykonalne. Wszakże podejścia bazujące na formalizmach matematycznych (zmierzające do automatycznego dowodzenia poprawności) rozwijają się i notują znaczące sukcesy[9]. Podejścia te są jednak niezwykle pracochłonne (wymagają żmudnych przygotowań do przeprowadzenia formalnego dowodu poprawności). Dopiero całkiem niedawno pojawiły się koncepcje i narzędzia ułatwiające formalną weryfikację, rozwijane na razie bardziej jako środki budowy poprawnych programów (idea design by contract), niż dla celów dowodzenia poprawności dowolnych, z zewnątrz wziętych, kodów[10]. Nie obarczają one użytkownika koniecznością poznania zaawansowanych formalizmów, wymagają za to zapisania w kodzie źródłowym, w specjalnej konwencji, warunków wstępnych, końcowych i niezmienników (dla klas, metod, pętli). Zatem oba podejścia: „surowy" formalizm logiczno-matematyczny oraz „ułatwiające życie" operowanie tylko warunkami i niezmiennikami nie dają się raczej zastosować do sprawdzania poprawności programów studentów. Pozostaje testowanie nadesłanych rozwiązań, co oczywiście pozwala wykryć jakiś zakres błędów, ale nie dowodzi, że program jest poprawny w każdym przypadku. Naturalnie chodzi też o to by testowanie rozwiązań mogło odbywać się automatycznie.

Automatyczne testowanie poprawności działania możliwe jest dla następujących klas zadań programistycznych:
- DOPASOWANIE DO KODU: dany jest gotowy fragment uruchomieniowy, wyprowadzający wyniki na konsolę lub do pliku, a zadanie polega na dopisaniu brakujących fragmentów (klas, metod),
- CZARNA SKRZYNKA: zadanie polega na wczytaniu danych wejściowych i zapisaniu wyników (wejście: pliki, bazy danych, wyjście: pliki, bazy, konsola),
- PO SPECYFIKACJI: zadanie zawiera dość szczegółową specyfikację (jakie klasy, metody itp. należy zdefiniować).

Oczywiście, rozwiązania nie mogą być błędne składniowo. Rozwiązania z błędami kompilacji są odrzucane na wstępie.

Przykładem DOPASOWANIA DO KODU może być następujące zadanie: zdefiniować klasę Incrementer w taki sposób, aby program pokazany w ramce Kod 1 dał wyniki przedstawione

[8] Projekt, bazujący na podobnej obserwacji, realizowany jest od 2008 roku w UCD School of Computer Science & Informatics w Dublinie (zob. opis *Using whitespace patterns to detect plagiarism in program code*, http://www.csi.ucd.ie/content/using-whitespace-patterns-detect-plagiarism-program-code).

[9] Dla przykładu, formalnie dowiedziono poprawności ponad 6000 linii kodu mikrojądra seL4 systemu operacyjnego (Klein G. et al, 2009).

[10] Należy do nich m.in. zestaw narzędzi skupiony wokół JML – Java Modelling Language (Leavens, Clifton, 2008).

w ramce Wydruk 1.

```
import static incr.Incrementer.*;

public class Test {

  public static void main(String[] args) {

    // najprostsza iteracja – krok = 1
    for (int k : in(1, 10)) System.out.print(k + " ");
    System.out.println();

    // Podany krok
    for (int k : in(1, 10).by(2)) System.out.print(k + " ");
    System.out.println();

    // Można w odwrotną stronę – tu domyślnie krok = -1
    for (int k : in(10, 1)) System.out.print(k + " ");
    System.out.println();

    // Zakres od min do max, a podany krok będzie
    // decydował o kierunku iteracji
    for (int k : in(1, 10).by(-1)) System.out.print(k + " ");

  }
}
```

Kod 1. Przykładowy program testowy w zadaniu typu DOPASOWANIE DO KODU

```
1 2 3 4 5 6 7 8 9 10
1 3 5 7 9
10 9 8 7 6 5 4 3 2 1
10 9 8 7 6 5 4 3 2 1
```

Wydruk 1. Wymagany wynik działania programu z ramki Kod 1

Automatyczne testowanie polegać tu będzie na tym, że dla każdego nadesłanego rozwiązania, dla różnych automatycznie generowanych klas Test i odpowiadających im zestawów wyników nastąpi uruchomienie programu i przechwycenie standardowego wyjścia oraz standardowego strumienia błędów. Rozwiązanie jest niepoprawne, gdy wystąpią błędy fazy wykonania lub wyjście programu nie jest zgodne z wymaganym wynikiem.

W tak formułowanych zadaniach student musi dopasować swoje rozwiązanie do podanych fragmentów kodu, co wymusza kierunek myślenia i może służyć ćwiczeniu użycia wybranych przez prowadzącego konstrukcji językowych.

Zadania typu CZARNA SKRZYNKA są z punktu widzenia automatycznego testowania bardzo podobne, ale pozwalają na większą swobodę w doborze sposobu rozwiązania. Automatyczne testowanie przebiega tu w 3 krokach:

- generowanie zestawów danych wejściowych (zawierających przypadki graniczne),
- uruchamianie ocenianych programów i weryfikacja ewentualnych błędów fazy wykonania,
- synchroniczne sprawdzanie poprawności wyników.

Przykładowe zadanie tego typu może wyglądać następująco: wczytaj z pliku podanego jako pierwszy argument programu liczby całkowite, utwórz z nich tablicę, w tablicy znajdź element maksymalny oraz indeksy na których występuje, wynik zapisz (w formie: max i1 i2 ... iN, gdzie max – maksymalna wartość, i1 ... iN – indeksy) do pliku podanego jako dru gi argument programu.

Ważne jest, tu by przy testowaniu sprawdzać warunki graniczne (np. brak pliku wejściowego, plik wejściowy nie zawiera (tylko) liczb, zawiera bardzo dużo liczb, zawiera tylko liczby Integer.MIN_VALUE).

Przy testowaniu rozwiązań za pomocą weryfikacji wyjścia programu należy w zadaniu precyzyjnie określić format wyniku, ale i wtedy trzeba się liczyć z tym, że nawet dobre rozwiązania

mogą go nie przestrzegać (fantazja lub niedbałość studentów). To oczywiście łatwo wykryć, ale odnieść należy raczej do grupy błędów związanych z niespełnieniem wymagań. Weryfikacja poprawności wyników programu będzie natomiast wymagać możliwie najbardziej uogólnionej analizy składniowej wyjścia (w przykładowym zadaniu należy np. założyć, że w wynikach mogą znaleźć się oprócz liczb jakieś opisy, że indeksy mogą być pokazywane w dowolnej kolejności, że mogą być oddzielane dowolnymi białymi znakami).

Bardziej szczegółowe testowanie programu polega na automatycznym sprawdzaniu, czy poszczególne elementy programu (takie jak np. metody jakiejś klasy) działają we właściwy sposób. To podejście, które wcześniej nazwane zostało tu podejściem PO SPECYFIKACJI, jest podstawą metodologii testów jednostkowych (Meszarosz, 2007). Testy jednostkowe i wspierające je narzędzia zadomowiły się w praktyce programowania, zmieniając nawet filozofię wytwarzania kodu (Kent, 2003). Dla naszych celów automatycznego sprawdzania poprawności rozwiązań wygodnie będzie myśleć o testach jednostkowych w kategoriach testowania zachowania programu (North, 2006). Z praktycznego punktu widzenia dobrym wyborem wydaje się zastosowanie środowiska testowania behawioralnego EasyB[11], udostępniającego prosty w użyciu, wyspecjalizowany język budowy testów (DSL), a jednocześnie całą moc języka Groovy.

W EasyB za pomocą DSL tworzymy narracje, a wewnątrz nich scenariusze, np.:

```
story "Operacje bankowe"
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 "Sygnalizacja błędu"
 and "Stan konta nie zmienił się"
```

Zastosowanie mocy języka Groovy (w klauzulach given i when) oraz proste konstrukcje sprawdzające EasyB (wstawiane w klauzulach then) dają możliwość szybkiego tworzenia testów, wyniki są łatwe do automatycznego przetworzenia, a przekazane studentom pozwalają dobrze zidentyfikować miejsca i przyczyny popełnionych błędów.

Naturalnie, aby zastosować to podejście, zadanie musi zawierać dość dokładną specyfikację, choć niekoniecznie doprecyzowaną we wszystkich szczegółach (co daje pole do swoistego sprawdzenia trafności decyzji projektowych studenta).

Jako przykład rozważmy następujące zadanie:

Stwórz klasę zbiorników o nazwie WaterTank.
Każdy zbiornik charakteryzuje się:
- *numerem (właściwość o nazwie* nr*),*
- *pojemnością (właściwość o nazwie* capacity*),*
- *stanem wody (właściwość o nazwie* currVol – *inicjalnie zero).*

Numery są nadawane automatycznie w kolejności tworzenia obiektów-zbiorników.
Zdefiniuj jednoargumentowy konstruktor, pozwalający zainicjować pojemność.
Zdefiniuj metody:
addWater – *dolewającą wodę do zbiornika,* removeWater – *odlewającą wodę ze zbiornika,* transferWater – *przelewającą wodę do innego zbiornika.*
Zapewnij obsługę błędów poprzez zgłaszanie wyjątków.

Fragment scenariusza EasyB testującego rozwiązania przedstawia Kod 2.

[11] Zob. http://www.easyb.org.

```
scenario "Tworzenie zbiorników",{
  def pojTest = [100, 50, 20]
  when "Utworzymy zbiorniki z pojemnosciami $pojTest", {
    listaZ = pojTest.collect { new WaterTank(it) }
  }
  then "zbiorniki mają podane pojemności, kolejne numery i stan wody = 0", {
    listaZ.eachWithIndex { z, i ->
      ensure( z ) {
        has ( [capacity: pojTest[i], nr: i+1, currVol: 0 ])
      }
    }
  }
  when "Tworzymy zbiorniki o wadliwych pojemnościach <= 0", {
    throwingCode1 = { z = new WaterTank(0)}
    throwingCode2 = { z = new WaterTank(-10) }
  }
  then "powinien byc zgloszony wyjatek", {
    ensureThrows(Exception){throwingCode1() }
    ensureThrows(Exception){throwingCode2() }
  }
}
```

Kod 2. Fragment scenariusza testowania klasy zbiorników

Wyniki tego testu-scenariusza dla rozwiązania z licznymi wadami (nieumiejętność automatycznego numerowania zbiorników, brak obsługi wadliwych pojemności) przedstawia Wydruk 2.

```
scenario Tworzenie zbiorników
      when Utworzymy zbiorniki z pojemnościami [100, 50, 20]
      then zbiorniki mają podane pojemności, kolejne numery
           i inicjalne stany wody = 0 [FAILURE: WaterTank.nr doesn't equal 2]
      when Tworzymy zbiorniki o wadliwych pojemnościach <= 0
      then powinien być zgłoszony wyjątek [FAILURE: expected exception]
```

Wydruk 2. Wyniki scenariusza dla wadliwego rozwiązania

W celu automatycznego testowania rozwiązań należy przygotować właściwy dla danego zadania zestaw scenariuszy testujących, a następnie – w trybie automatycznym – uruchamiać scenariusze dla każdego rozwiązania i przetwarzać ich wyniki w celu oceny poprawności rozwiązania, a także przygotowania indywidualnych raportów dla studentów, pokazujących jakie i gdzie błędy popełnili.

EasyB daje się łatwo integrować z innymi środowiskami testowania (np. dla aplikacji GUI – tu można zastosować środowisko Fixtures for Easy Software Testing[12], czy dla web aplikacji, gdzie z poziomu EasyB mamy prosty dostęp do takich platform testujących, jak HtmlUnit czy Selenium[13]). Przykładowy scenariusz testowania rozwiązań prostego zadania, polegającego na stworzeniu klasy o nazwie GUI, reprezentującej okno ramowe z umieszczonym w nim przyciskiem oraz obsłudze zdarzeń naciśnięcia przycisku (kolejne wciśnięcia zwiększają liczbę pokazywaną na przycisku o 1) przedstawiono w ramce Kod 3.

[12] Zob. http://code.google.com/p/fest/.

[13] Zob. http://htmlunit.sourceforge.net oraz http://seleniumhq.org.

```
scenario "Przycisk-licznik",{

given "GUI utworzone i widoczne",{
    window = new FrameFixture( new GUI() )
    window.show()
}

when "Pierwsze naciśnięcie na przycisk", {
    butt = window.button(JButtonMatcher.withText("0"))
    butt.click()
}

then "tekst na przycisku ma być 1", {
    butt.text() == '1'
}
// ... inne przypadki wielokrotnego naciskania na przycisk
}
```

Kod 3. Przykładowy scenariusz testowania klasy GUI

Przy testowaniu aplikacji współbieżnych problemem są niedeterministyczne wyniki złych rozwiązań. Normalne wielokrotne testy mogą ich nie wykryć (złe wyniki w złym rozwiązaniu zdarzają się przypadkowo i nieczęsto, bo np. konteksty nie są często przełączane). Wyjściem z sytuacji może być przeprowadzanie testów w mniej deterministycznym środowisku poprzez taką instrumentację testowanego kodu, że przełączenia kontekstów zdarzają się często[14].

We wszystkich omawianych tu podejściach do automatycznego sprawdzania poprawności programów studentów konieczna jest ochrona przed testowanym kodem (który po uruchomieniu może np. dokonywać jakichś „złośliwych" operacji na lokalnym systemie plikowym). Możliwe rozwiązania tego problemu obejmują: testowanie kodów na wydzielonym komputerze lub w środowisku zapewniającym ochronę przed niedozwolonymi operacjami (*sandbox*) oraz statyczne sprawdzanie kodu na obecność niebezpiecznych konstrukcji (np. za pomocą analizy składniowej z użyciem wyrażeń regularnych).

Warto na koniec zauważyć, że generowanie automatycznej oceny (punktacji) za poprawność kodu wymaga przemyślanej strategii: na ile częściowo poprawne kody (np. generalnie prawidłowe, ale nie uwzględniające pewnych granicznych przypadków) uznajemy za do przyjęcia? Dobrym podejściem wydaje się tu przyznawanie jakiejś minimalnej liczby punktów za podstawową poprawność i zwiększanie punktacji za uwzględnienie różnych granicznych przypadków. Automatyczne procedury oceniania poprawności wymagają zatem nie tylko starannego przygotowania scenariuszy testowania dla każdego zadania, ale też skojarzenia z nimi odpowiednich strategii punktowania. Wynikiem winna być nie tylko automatyczna ocena (punktacja), ale również automatycznie generowane indywidualne raporty, wskazujące poszczególnym studentom, jakie błędy popełnili i na ile zmniejszyło to ich punktację.

4. Wymagania na środki realizacji

Oceniając programy studentów powinniśmy postawić również pytanie: czy w rozwiązaniu zastosowano właściwe środki realizacji (konstrukcje językowe, podejścia, biblioteki)? Dotyczy to nie tylko zadań, w których bezpośrednio postawiono wymagania co do zastosowania jakichś konstrukcji, ale również sytuacji, w których student całkowicie samodzielnie dobiera środki realizacji zadania. W obu przypadkach można zaproponować ten sam mechanizm automatycznego sprawdzenia i oceny: zastosowanie statycznej analizy kodu (z użyciem regularnych wyrażeń lub analizy AST), w razie potrzeby wsparty testami fazy wykonania (być może introspekcją, może specjalnymi scenariuszami testowymi).

[14] Przykładem takiego rozwiązania jest środowisko ConTest, zob. www.alphaworks.ibm.com/tech/contest.

Rozważmy przykład:

Zdefiniować klasę JavaBean o nazwie Purchase z trzema właściwościami: produkt (typ String, nazwa item*), data zakupu (typ Date, nazwa* date*) i cena (typ Double, nazwa* price*). Właściwość item jest prosta, właściwości date i price są związane (bounded) i ograniczane (constrained).*

Wczytać z przykładowego pliku dane o zakupach produktów.

Dostarczyć GUI, które pokazuje te dane, a jednocześnie umożliwia edycję dat i cen.

Za pomocą mechanizmu nasłuchu i wetowania zmian właściwości umożliwić:

- *zapis do pliku o nazwie* changes.log *wszystkich zmian dat i cen,*
- *kontrolę poprawności zmian:*
  - *nie można zmienić daty na późniejszą niż „dziś”,*
  - *nie można zmienić ceny na liczbę niedodatnią.*

Tutaj testowanie rozwiązań za pomocą scenariusza zmian właściwości nie wystarczy, bo zapis zmian i odrzucanie niedozwolonych zmian może być zrealizowane innymi od wymaganych środkami. Do weryfikacji spełnienia wymagań można użyć regularnych wyrażeń. Ale scenariusz też się przyda – nie tylko do stwierdzenia poprawności działania, ale też do ograniczenia liczby potrzebnych wyrażeń regularnych.

W tym przypadku wystarczą regularne wyrażenia do stwierdzenia czy settery dla właściwości *date* i *price* zawierają odpowiednie konstrukcje. Na przykład (po usunięciu białych znaków z plików źródłowych) takie wyrażenia mogą mieć postać:

```
publicvoidsetPrice.+?throwsPropertyVetoException\{.+?fireVetoableChange.+?firePropertyChange.+?\}
publicvoidsetDate.+?throwsPropertyVetoException\{.+?fireVetoableChange.+?firePropertyChange.+?\}
```

a scenariusz testujący zachowanie (EasyB) pozwoli stwierdzić, czy są zdefiniowani i przyłączeni słuchacze zmian właściwości oraz czy dobrze działają.

5. Jakość kodu, jego uniwersalność, skalowalność i efektywność

Wyniki testowania poprawności to dopiero połowa drogi w ocenie rozwiązań nadsyłanych przez studentów. Pozytywna ocena poprawności nie oznacza jeszcze, że kod jest dobry. Jak wspomniano już w punkcie 1, ważnym kryterium oceny jest jakość kodu – m.in. jego czytelność, a zarazem zwięzłość, zgodność z zasadami i praktykami dobrego programowania. Wydaje się, że ta sfera (dotykająca w istocie sztuki programowania) jest domeną ocen przede wszystkim jakościowych, formułowanych na podstawie przeglądania kodu. Jednak również w tym obszarze istnieją możliwości automatyzacji procesu oceny.

W tym celu można wykorzystać gotowe środowiska i programy statycznej analizy kodu. W ocenie rozwiązań studentów wydaje się właściwe zastosowanie jednego z kilku prostych w użyciu, darmowych narzędzi, takich jak Checkstyle (zob. http://checkstyle.sourceforge.net), FindBugs (zob. http://findbugs.sourceforge.net), PMD (zob. http://pmd.sourceforge.net) lub ich kombinacji. Za ich pomocą można automatycznie analizować kody rozwiązań m.in. pod kątem:

- powtórzenia fragmentów (spełnienie zasady DRY),
- niepotrzebnych fragmentów (np. deklaracji nieużywanych zmiennych),
- błędów, które mogą nie wychodzić w testach (np. użycie == w porównaniu napisów)
- złych praktyk (np. ignorowania wyjątków),
- konwencji nazewniczych i stylu programowania (zbyt krótkie, zbyt długie, niezgodne z zasadami języka nazwy; wadliwe formatowanie kodu),

- złożoności, czytelności, elastyczności, uniwersalności kodu (z zastosowaniem rozlicznych metryk)[15].

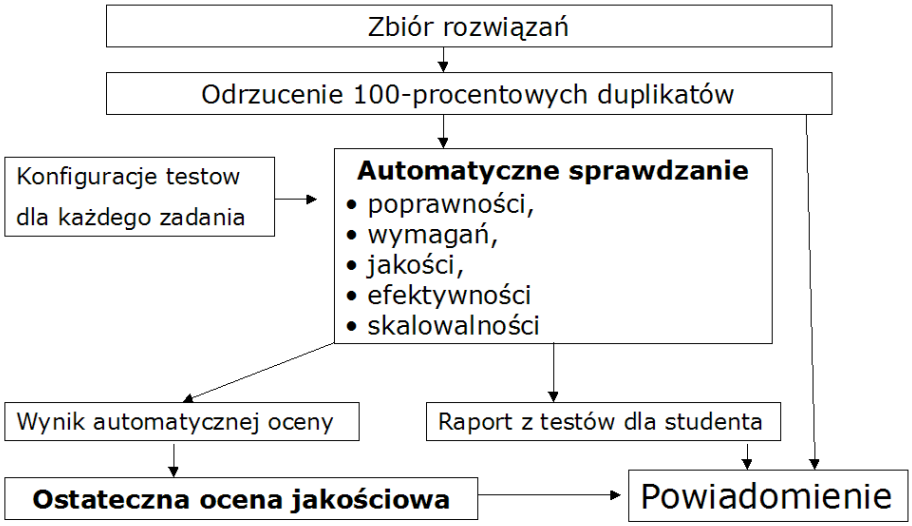
Testy jakości kodu muszą być dobrane do konkretnych zadań, co oznacza właściwy dobór oprogramowania sprawdzającego i jego odpowiednie przygotowanie (określenie co i jak ma być sprawdzane, a więc skonfigurowanie dostępnych testów i ewentualnie wprowadzenie dodatkowych rozszerzeń). Należy też określić graniczne dla danego zadania wartości wybranych metryk (być może w oparciu o wzorcowe rozwiązanie). Tak przygotowany swoisty system będzie następnie uruchamiany na zbiorze rozwiązań i posłuży automatycznej generacji ocen jakości kodu, a także indywidualnych raportów, opisujących niedociągnięcia w tym zakresie.

Na ogólną ocenę studenckich kodów powinny mieć też wpływ ich efektywność i skalowalność. Najprostszym sposobem oceny skalowalności i efektywności rozwiązań jest uruchamianie programów w warunkach dużego obciążenia (duże zestawy danych wejściowych, duża liczba wątków czy użytkowników) i mierzenie ogólnego czasu ich wykonania. Zautomatyzowana procedura, zbierająca wyniki pomiarów winna odnosić je do wartości uzyskiwanych przez rozwiązania wzorcowe.

Bardziej zaawansowane analizy efektywności i skalowalności mogą opierać się na użyciu profilerów, które za pomocą zewnętrznej instrumentacji kodu rozwiązań dostarczają szczegółowej informacji o fragmentach programu mających największy wpływ na niewystarczającą efektywność czy skalowalność[16].

6. Podsumowanie: integracja i interakcja

Omówione sposoby automatycznego sprawdzania i oceny rozwiązań powinny być zintegrowane w jednym systemie. Jego koncepcyjne ramy przedstawiono na rys. 1.



Rysunek 1. Koncepcja zintegrowanego systemu sprawdzania i oceny rozwiązań

Zalety zastosowania takiego systemu w zdalnym nauczaniu programowania polegają nie tylko na wsparciu pracy nauczyciela. Jedynie dzięki integracji automatycznych procedur

[15] Jest wiele metryk, opisujących jakość programu (Fenton & Pfleeger 1998). Dla prostych zadań można zastosować te podstawowe, udostępniane przez omawiane narzędzia (np. JavaNCSS czy CyclomaticCompexity). Dla bardziej złożonych zadań można próbować określać m.in. spójność (*cohesion*) i współzależności (*coupling*) fragmentów kodu za pomocą metryk obiektowych np. metryk Chidambera i Kemerera (Chidamber, Kemerer 1994), dostępnych m.in. w pakiecie ckjm (zob. http:// www.spinellis.gr/sw/ckjm).

[16] Taka informacja jest szczególnie cenna dla studentów, a dzięki odpowiednim narzędziom może być ona generowana automatycznie. Do narzędzi takich należy np. Jensor (zob. http://jensor.sourceforge.net).

sprawdzających można uzyskać wieloaspektową, kompleksową ocenę rozwiązań według kryteriów przedstawionych w pierwszym punkcie artykułu. Szczegółowa informacja o wynikach różnych testów, o tym gdzie i jakie błędy zostały popełnione, udostępniana na studentom czy to na platformie edukacyjnej, czy to w postaci indywidualnie generowanych maili znacząco podniesie jakość kształcenia. Interakcja ze studentami jest w tym kontekście kluczowa. W perspektywie należałoby dążyć do jej rozwijania np. w kierunku iteracyjnego przygotowywania rozwiązań przez studentów: kolejne wersje rozwiązania są automatycznie sprawdzane i oceniane, automatycznie generowany raport z testów służy studentowi do przygotowania następnej, udoskonalonej wersji rozwiązania. Po kilku iteracjach formułowana jest ocena ostateczna, a rejestracja zmian dokonywanych w kolejnych wersjach rozwiązań pozwala dodatkowo ocenić skalę postępów w nauce, a także lepiej zidentyfikować bariery rozumienia treści przedmiotu i ukierunkować indywidualne konsultacje.

Wdrożenie omawianej tu koncepcji nie jest zadaniem łatwym. Zadania muszą być specjalnie przygotowane, zarówno pod względem treści, jak i wymagań formalnych (np. co do struktury projektów, pakietów, a nawet nazw klas uruchomieniowych). Jak wykazuje doświadczenie, studenci mają duże problemy z podporządkowaniem się takim formalnym regułom, dlatego należy udostępniać im narzędzia łatwego tworzenia projektów o wymaganej strukturze[17].

Znacznie większe trudności wiążą się z właściwym, w pełni miarodajnym przygotowaniem scenariuszy testowania poprawności czy testów sprawdzających spełnianie wymagań, jakość kodu itp. Użycie automatycznych procedur sprawdzających (szczególnie w proponowanym iteracyjnym rozwiązywaniu zadań przez studentów) wyklucza tolerancję dla jakichkolwiek błędów w testach. Trzeba też pamiętać o tym, że automatycznie można sprawdzać tylko to, co daje się sprawdzić za pomocą proponowanych narzędzi ze stuprocentową trafnością.

Należy też zwrócić uwagę na jeszcze jedną kwestię. Artykuł zaczynał się od stwierdzenia, że programowanie jest tak naprawdę sztuką. Okazało się potem, że niektóre elementy tej sztuki poddają się automatycznej ocenie, co jednak zazwyczaj wymaga jakichś specjalnych zabiegów (np. odpowiedniego przygotowania treści zadań czy interpretacji metryk kodu). Automatyczne sprawdzanie i ocena rozwiązań ma wspomniane już zalety, łatwo więc ulec pokusie przygotowania zadań „pod" taką automatyzację, a to może blokować rozwijanie u studentów twórczego podejścia do programowania. Pamiętajmy też o tym, że ostateczna ocena rozwiązań studentów nie może ograniczać się tylko do zebrania wyników automatycznych testów. Będą one na pewno pomocne, ale po to, by naprawdę ocenić „sztukę programowania", trzeba zajrzeć do kodu ludzkim okiem, a po to, by ją rozwijać, trzeba też – oprócz zadań poddających się automatycznej ocenie – oferować studentom zadania w pełni twórcze, bez żadnych ograniczeń.

7. Bibliografia

1. Alexander, C. (1979). The Timeless Way of Building. Oxford University Press.
2. Beck, K., Andres, C. (2004). Extreme Programming Explained: Embrace Change. Wyd. 2, Addison-Wesley.
3. Beck, K. et al. (2001). Agile Manifesto. http://agilemanifesto.org.
4. Beck, K. (2003). Test-Driven Development by Example. Addison Wesley.
5. Brooks, F. (1995). The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition (2nd Edition). Addison-Wesley Professional.
6. Liu, C. et al. (2006). GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis. KDD'06, August 20–23, 2006, Philadelphia.
7. Chidamber, S.R., Kemerer, C.F. (1994). A metrics suite for object-oriented design. IEEE Transactions on Software Engineering, Vol. 20, No 6.

[17] Np. już teraz w praktyce zdalnego nauczania programowania w PJWSTK udostępniam studentom prosty generator projektów Eclipse, którego użycie gwarantuje, iż rozwiązania będą miały odpowiednią formę.

8. Cosma, G., Joy, M. (2006). Source-code plagiarism: A UK academic perspective Department of Computer Science. The University of Warwick, Research Report No. 422.
9. Fenton, N.E., Pfleeger, S.L.(1998). Software Metrics: A Rigorous and Practical Approach. Course Technology. Wyd.2.
10. Fowler, M. (2004). Is Design Dead?. Strona Martina Fowlera
11. Gabriel, R.P. (1996). Patterns of Software. Tales from the Software Community. Oxford University Press
12. Halstead, M.H. (1977). Elements of Software Science. Elsevier North-Holland.
13. Hunt, A., David, T. (2001). The Art in Computer Programming. The Pragmatic Programmers, LLC.
14. Johnston, B. (2003). The concept of plagiarism. Learning and Teaching in Action. Volume 2, No 1.
15. Jones, E.L. (2001). Metrics based plagiarism monitoring. 6th Annual CCSC Northeastern Conference, Middlebury, Vermont, April 20–21, 2001.
16. Klein, G. et al. (2009). seL4: Formal verification of an OS kernel. 22nd ACM Symposium on Operating System Principles. Big Sky, MT.
17. Knuth, D. (1974). Computer Programming as an Art. CACM, December 1974.
18. Lammers, S. (1986). Programmers at Work. Microsoft Press, Redmond.
19. Leavens, G., Clifton, C. (2008). Lessons from the JML Project. Meyer, B. Woodcock, J. (eds.). Verified Software: Theories, Tools, Experiments, Volume 4171 of Lecture Notes in Computer Science, Springer Verlag.
20. Li, X., Zhong, J.X. (2010). The Source Code Plagiarism Detection Using AST. Proceedings of the 2010 International Symposium on Intelligence Information Processing and Trusted Computing. IEEE Computer Society, Washington.
21. Liu, C. et al (2006). GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis. KDD'06, August 20–23, 2006, Philadelphia.
22. Mann, S., Frew, Z. (2006). Similarity and originality in code: plagiarism and normal variation in student assignments. 8 Australasian Computing Education Conference, Hobart, Tasmania, Australia, Conferences in Research and Practice in Information Technology, Vol. 52.
23. Martin, R.C. (2008). Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall.
24. McCabe, T. (1976). A Complexity Measure. IEEE Transactions on Software Engineering.
25. McConnell, S. (2004). Code Complete: A Practical Handbook of Software Construction. Wydanie 2, Microsoft Press.
26. Meszarosz, G. (2007). xUnit Test Patterns: Refactoring Test Code. Pearson Education Inc.
27. North, D. (2006). Introducing BDD. Better Software, March 2006.
28. Poppendieck, M., Poppendieck, T. (2003). Lean Software Development: An Agile Toolkit. Addison-Wesley Professional.
29. Schleimer, S., Wilkerson, D.S., Aiken, A. (2003). Winnowing: Local Algorithms for Document Fingerprinting. SIGMOD 2003, San Diego.
30. Sheard, J., Dick, M., Markham, S., Macdonald, I., Walsh, M. (2002). Cheating and plagiarism: Perceptions and practices of first year IT students. Proceedings of the 7th Annual Conference on Innovation and Technology in Computer Science Education. Aarhus, Denmark, June 2002. ACM Press, New York.
31. Shore, J., Warden, S. (2008). The Art of Agile Development. O'Reilly Media, Inc.
32. Wise, M.J. (1993). Running Karp–Rabin Matching and Greedy String Tiling. Basser Department of Computer Science, Technical Report 463.

Programming Tasks in e-Learning

Summary

Keywords: agile programming; design patterns; plagiarism detection; behavior driven development; programming style; program metrics

The article discusses the goals of teaching programing languages, kinds of programming tasks, evaluation criteria and methods for solutions checking. Many aspects of the assessments need, especially within e-learning framework, dedicated tools for solutions checking. Considered are the possibilities and methods for their automatic application. Integration of automatic evaluation tools in a consistent system is proposed. Through the rich content of the interaction with students such a system would lead to increase of e-learning quality. Examples presented in this article apply to programs and tools for the Java platform.