Samodejno preverjanje pravilnosti študentskih nalog iz programiranja

Aleš Čep, Damijan Novak, Jani Dugonik
Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, Koroška cesta 46, 2000 Maribor
ales.cep@um.si, damijan.novak@um.si, jani.dugonik@um.si

Izvleček

Fakultete se lahko soočijo s problematiko preverjanja pravilnosti delovanja velikega števila oddanih nalog, predvsem pri predmetih iz programiranja. Predlagan sistem za avtomatizacijo preverjanja oddanih nalog lahko čas testiranja nalog bistveno skrajša in s tem razbremeni pedagoško osebje. Povratne informacije o točnosti oddane naloge so deležni tudi študenti, ki jim sistem poda opis napake, ki jo lahko v ponovnem poskusu oddaje odpravijo. Sistem omogoča preverjanje nalog v dveh programskih jezikih (C# in C++), komunikacija med pedagogom in študentom pa spada v kategorijo e-učenja. Prednost uporabe predlaganega sistema je popolna nepristranost ocenjevalca, slabost pa se kaže v večjem začetnem časovnem vložku, potrebnem za sestavo naloge. V članku bo podrobneje predstavljena notranja zgradba sistema za preverjanje pravilnosti delovanja nalog iz programiranja.

Ključne besede: Docker, e-učenje, Moss, ocenjevanje študentskih nalog, testiranje enot, testno voden razvoj.

Abstract

Faculties may face the problem of verification of many submitted assignments, especially in subjects with programming assignments. The proposed system for verification automation of submitted assignments can significantly reduce the testing times and relieve the teaching staff. Feedback about the submitted assignments' correctness is also given to the students, as well as descriptions of the errors so that they can correct the assignments and submit them again. The system supports the verification for two different programming languages (C# and C++). The communication between the teacher and the student puts the proposed system into the E-learning category. The advantage of using the proposed system is the complete impartiality of the assessor while the disadvantage is that more time is needed to compose an assignment. This paper presents the detailed internal structure of the assignment verification system.

Keywords: Docker, e-learning, grading of the student assignments, Moss, Test Driven Development, Unit testing.

1 UVOD

Pri predmetih iz programiranja se študente seznani s pravilno sintakso in uporabo specifičnih programskih jezikov, s pristopi sestavljanja postopkov oz. algoritmov ter s predstavitvijo podatkov in podatkovnih struktur.

Na fakultetah pri predmetih z večjim številom študentov (več kot 100) predstavlja preverjanje oddanih nalog velik izziv tako z vidika zagotavljanja enakega kriterija ocenjevanja kot tudi časovne obremenitve pedagoškega osebja. Zaradi teh razlogov smo se odločili za avtomatizacijo postopka preverjanja študentskih nalog, ki se že nekaj let uporablja na Fakulteti za elektrotehniko, računalništvo in informatiko. Sistem se uporablja pri predmetu Podatkovne strukture v 1. letniku visokošolskega študija, na dveh študijskih smereh (računalništvo in informatika), kjer je v posameznem študijskem letu skupno vpisanih več kot 200 študentov. Študentje lahko izbirajo med dvema programskima jezikoma: C# in C++. Pri vsaki nalogi so na voljo navodila in predloga programa, ki vsebuje vnaprej deklarirane razrede, programske strukture ter metode. Metode, ki jih morajo implementirati študenti na osnovi podanih navodil, so podane samo s prototipi in komentarji z razlago.

Od samega začetka je bil sistem zasnovan z mislijo na vpetost študentov v učni proces in na njihovo uporabniško izkušnjo. Če študent pri implement-

aciji sledi navodilom naloge, mu sistem ob morebitni storjeni napaki pri osvajanju zahtevane snovi oziroma opravljanju naloge poda jasen povraten odziv. Pedagog in študent sta tako aktivno vključena v prepletenost učnega procesa in Informacijsko-Komunikacijske Tehnologije (IKT) sistema, kar na kratko imenujemo e-učenje.

Predlagan sistem nadaljuje razvoj sistema za preverjanje pravilnosti študentskih nalog (Zemljak, Novak, Čep, & Verber, 2016). Glavni napredek je storjen v stopnji avtomatizacije s prenosom nalog s spletne učilnice in pri sestavljanju razpredelnice z odzivi, kar pohitri objavo rezultatov. Prav tako smo v sistem dodali zabojnike Docker, s čimer smo poskrbeli za dodatno izolacijo med izvajanjem testov. Za preverjanje podobnosti izvorne kode smo dodali uporabo sistema Moss.

Preostanek članka je organiziran na naslednji način. Poglavje **2** opiše področja in orodja, uporabljena v našem sistemu. Poglavje **3** predstavi sorodne sisteme s področja računalniškega popravljanja nalog. Predlagan sistem je opisan v poglavju **4**. Poglavje **5** vsebuje predstavitev primera ogrodja in testne metode. Poglavje 6 vsebuje zaključek članka.

2 PREDSTAVITEV PODROČJA

V tem poglavju predstavljamo področja, ki so ključna za razumevanje delovanja sistema in so pomembna za njegovo gradnjo oziroma izvajanje. Najprej opredelimo definicijo e-učenja kot ključno kategorijo, v katero se umešča ta sistem. Nato opišemo testno voden razvoj in testiranje enot, ki imata ključno vlogo pri gradnji nalog in njihovem preverjanju. V sistemu platforma Docker poskrbi za izolacijo testov med izvajanjem, sistem Moss pa je uporabljen za preverjanje podobnosti programske kode z namenom preprečevanja plagiatorstva.

2.1 E-učenje

Zgodovinsko gledano ima e-učenje oziroma e-izobraževanje (angl. E-Learning) korenine v učenju na daljavo. Z nastankom interneta je namreč postalo mogoče, da sta pedagog in učenec ločena krajevno in prostorsko, med njima pa poteka komunikacija (E-izobraževanje - Wikipedija, prosta enciklopedija, 2019). Pri tem ne gre samo za deljenje gradiv v elektronski obliki, v smislu izmenjave preprostih elektronskih sporočil, ampak se e-učenje definira kot pedagogika (umetnost in znanost učinkovitega učenja), ki je še dodatno obogatena s sodobno IKT. E-učenje je tako povezava med IKT in učenjem, vodilno mesto v tej povezavi pa pripada pedagogiki. IKT mora biti zanesljiva in dovolj preprosta za uporabo, da ne moti učnega procesa (Nichols, 2007).
Dandanes se v pojem e-učenja vključujejo:

- učenje na daljavo (ni samo v spletni obliki, npr. DVD z naloženo vsebino za učenje),
- mešani načini deljenja virov (npr. spletni viri dopolnjujejo učenje v učilnici),
- spletno učenje (celotna komunikacija poteka izključno preko spleta),
- interaktivnost vsebin (od preprostih povezav na spletne strani do učnih okolij, ki so popolnoma prilagojena uporabniku),
- sistemi za nadzor učenja (z vključenimi možnostmi ocenjevanja) in
- pedagogika.

2.2. Testno voden razvoj

Testno voden razvoj (angl. Test-Driven Development – TDD) je metodologija, ki s pomočjo zelo kratkih razvojnih ciklov omogoča razvoj programskih in informacijskih rešitev, kar jo uvršča med agilno-iterativne procese. Razvoj poteka tako, da se v posameznem ciklu najprej napiše specifična testna metoda (oz. test), šele nato se izvede implementacija funkcionalnosti (oz. dela specifikacij sistema, ki ga razvijamo). Testna metoda tako vodi proces razvoja implementacije funkcionalnosti. Cikel je zaključen, ko se test izvede pravilno, implementacija funkcionalnosti pa je ustrezno vključena v rešitev. Za posamezno funkcionalnost se lahko ponovi več ciklov. Testi morajo biti spisani tako, da jih odlikujejo lastnosti, kot so: neodvisnost, razumljivost in hitrost.

2.3. Testiranje enot

Testiranje enot (angl. Unit testing) (Myers, Sandler, & Badgett, 2011) je metoda testiranja programske kode, pri kateri se testi izvajajo na nivoju posamezne enote (npr. metode, funkcije ali razreda) v programu. Pri pisanju testov je najpogosteje v uporabi t. i. AAA (angl. Arrange-Act-Assert) vzorec, ki predlaga razdelitev testne metode na tri stopnje:

- urejanje (angl. Arrange) – začetna stopnja, kjer se pripravijo vhodni podatki za testirano enoto in določi pričakovan izhod,
- izvajanje (angl. Act) – klic testirane enote s pripravljenimi vhodnimi podatki,

- potrjevanje (angl. Assert) – končna stopnja, kjer se izvede primerjava med dobljenim izhodom iz testirane enote ter predvidenim izhodom iz začetne stopnje.

2.4. Platforma Docker

Platforma Docker (Docker Inc., 2019; Gradišnik & Majer, 2016) je odprtokodna platforma za razvoj aplikacij, njihovo distribucijo in zagon na različnih infrastrukturah z uporabo t. i. zabojnikov. Zabojnik Docker (angl. Docker container) temelji na tehnologiji zabojnikov iz operacijskega sistema Linux in predstavlja nivo abstrakcije na aplikacijskem nivoju. Njegov namen je izolacija programske opreme od izvajalnega sistema. Znotraj zabojnika se zažene slika Docker (angl. Docker image), ki je izvedljiv paket z vsem potrebnim za zagon aplikacije − izvajalno kodo, datotečnim sistemom in vsemi potrebnimi knjižnicami.

2.5. Sistem Moss

Kot pomoč pri odkrivanju plagiatov uporabljamo sistem za določanje podobnosti programske kode, imenovan Moss (Aiken, 2018) (angl. Measure Of Software Similarity). Algoritem, uporabljen za odkrivanje prevar v sistemu Moss, v primerjavi z drugimi podobnimi algoritmi dosega veliko boljše rezultate (Bowyer & Hall, 1999). Sistemu posredujemo seznam programskih kod, ki jih želimo medsebojno primerjati, ta pa nam kot rezultat vrne datoteko HTML s poudarjenimi deli programskih kod, ki nakazujejo podobnost, in odstotke podobnosti.

Dodatna funkcionalnost sistema je možnost odstranjevanja t. i. lažnih ujemanj, saj se iz primerjave odstrani t. i. skupna programska koda (npr. knjižnice, že napisana programska koda itd.). Sistem Moss trenutno podpira več programskih jezikov: C, C++, Java, C#, Python, Visual Basic, Javascript, FORTRAN, ML, Haskell, Lisp, Scheme, Pascal, Modula2, Ada, Perl, TCL, Matlab, VHDL, Verilog, Spice, MIPS assembly, a8086 assembly, a8086 assembly in HCL2.

Ker sistem Moss poda zgolj odstotek ujemanja posameznih programskih kod, ni namenjen popolnoma avtomatiziranemu odkrivanju plagiatorstva, temveč pregledovalcu služi zgolj kot pripomoček.

3 SORODNE IDEJE

Računalniško preverjanje, vrednotenje in ocenjevanje nalog iz programiranja je staro skoraj toliko kot programiranje samo. Sisteme preverjanja nalog iz programiranja delimo glede na (Ihantola, Ahoniemi, Karavirta, & Seppälä, 2010; Romli, Sulaiman, & Zamli, 2010): stopnjo sposobnosti samodejnega preverjanja, število podprtih programskih jezikov, izvajalno okolje preverjanih programov itd. Uporabljeni so na različnih institucijah (Auffarth, López-Sánchez, Campos i Miralles, & Puig, 2008; Edwards, 2003), kamor moramo prišteti tudi aktualne spletne akademije (Massive open online course - Wikipedia, 2019). Primer takšnih so Coursera (Coursera Inc., 2019), edX (edX Inc., 2019), Udemy (Udemy, Inc., 2019), LinkedIn Business (LinkedIn Learning, 2019), Udacity (Udacity, Inc., 2019) itd., ki so namenjene skoraj neomejenim številom udeležencem. Večina spletnih akademij poleg tradicionalnih gradiv (posneta predavanja, branje, množica problemov) ponuja interaktivne tečaje z uporabniškimi forumi, ki podpirajo interakcijo med študenti, učitelji in asistenti, omogočajo pa tudi takojšne povratne informacije za kvize in naloge. Za podrobnejši pregled orodij za samodejno preverjanje nalog vestnemu bralcu svetujemo ogled dodatne literature (Douce, Livingstone, & Orwell, 2005).

V literaturi (Zeller, 2000) je predstavljen sistem Praktomat, ki študentom omogoča medsebojno pregledovanje in ocenjevanje programov z namenom izboljšanja kakovosti ter sloga programske kode. Študent ima po oddani svoji nalogi možnost pridobiti nalogo drugega študenta, ki jo nato pregleda. Po pregledu študent dobi povratno informacijo o oddaji svoje naloge in možnost ponovne, popravljene oddaje. Sistem pregledovanja je neodvisen od ocenjevanja. Možnost plagiatorstva je zmanjšana s poosebljenimi nalogami in samodejnim testiranjem oddanih nalog.

4 PREDSTAVITEV NAŠEGA SISTEMA

Predstavljen sistem nadaljuje delo predstavljeno v (Zemljak, Novak, Čep, & Verber, 2016). Omogoča preverjanje oddaj študentskih nalog z namenom dviga kakovosti študijskega procesa, saj gre za nepristranost (objektivnost preverjanja pravilnosti programske kode), hitrejše preverjanje študentskega razumevanja učne snovi ter poenotene odzive, ki jih študentje dobijo po oddaji naloge. Poenoteni odzivi študentom ponudijo večjo jasnost pri razumevanju lastnih lukenj v znanju, hkrati pa se jim zagotovi hitrejša in bolj strukturirana pomoč pri odpravi morebitnih napak. Prednost sistema je tudi razbremenitev pedagoškega osebja, saj čas preverjanja nalog ne ra-

ste več linearno s količino oddaj. Treba je pa vzeti v zakup, da je začetna priprava naloge za samodejno preverjanje časovno daljša in je potreben razmislek, ali je uporaba takšnega sistema časovno upravičena pri manjšem številu oddanih nalog.

Delovanje sistema za preverjanje študentskih oddaj je razdeljeno v šest faz:

1. začetna priprava naloge,
2. izdelava in objava predloge, vključno z navodili,
3. prenos in razvrščanje oddanih nalog,
4. preverjanje plagiatorstva,
5. obdelava in testiranje nalog,
6. objava rezultatov.

Prva faza je opravljena s strani upravljavcev sistema in vključuje beleženje funkcionalnih zahtev, implementacijo rešitve s pomočjo TDD, ki bo ustrezala zapisanim zahtevam, ter sestavo besedila naloge. V prvi fazi se tekom ciklov TDD-ja (poglavje 4.1): a) ustvari množica testov, ki služijo kot specifikacija naloge, b) tekom preoblikovanj formira končna arhitektura naloge. Pomemben poudarek pri razumevanju pomena prve faze je, da se testi tekom razvoja v tej fazi ne pišejo z namenom validacije oz. preverjanja pravilnosti delovanja sistema, ampak samo kot specifikacija zahtev. Ko je razvoj zaključen in končna arhitektura naloge z vsemi funkcionalnostmi implementirana, se te iste teste (lahko) prekvalificira po uporabi in se jih s pridom uporabi še za validacijo pravilnosti delovanja študentskih oddaj (tako kot je to v našem sistemu − peta faza). Še en pomemben poudarek: testi so namenjeni samo razvijalcem ter jih študentje pri reševanju naloge ne pišejo (razen, če je to njihova želja, oz. opazijo potrebo po testih za namen validacije lastne rešitve).

Rešitev je treba implementirati v dveh programskih jezikih, saj se mora pri opravljanju vaj zaradi dveh različnih študijskih smeri študentom omogočiti možnost izbire programskega jezika (C# ali C++). V praksi to poteka tako, da se najprej s TDD ustvari rešitev v enem programskem jeziku, nato pa sledi ročna preslikava kode še za drugi programski jezik. Pri tem skrbimo za ohranitev strukture razredov, metod, testov in notranjih implementacij.

Druga faza je namenjena pripravi predloge, ki jo dobijo študentje kot osnovo in vsebuje prototipe metod, ki jih morajo implementirati. Predlogi (za vsak programski jezik ena) se vključno z navodili naloge objavita študentom v spletni učilnici. Sledi čas, namenjen reševanju naloge s strani študentov.

Tretja faza je izpeljana po pretečenem roku za oddajo naloge. Sistem iz spletne učilnice samodejno prenese oddane naloge in jih razdeli v ločene mape (vsak študent ima svojo mapo).

Četrta faza je namenjena preverjanju podobnosti oddanih nalog pred testiranjem. V ta namen se uporabi skripta (Algoritem 4.4), ki za preverjanje uporabi sistem Moss. Odziv je datoteka HTML, ki vsebuje seznam parov datotek z odstotki ujemanja podobnosti. Pare datotek s (pre)velikim odstotkom ujemanja pedagoško osebje ročno preveri.

Peta faza je namenjena obdelavi in testiranju nalog. Pred začetkom testiranja se v vsako študentsko mapo dodata dve datoteki: prva vsebuje teste, druga pa je namenjena gradnji sistema (angl. Makefile). Za vsakega študenta se zažene zabojnik Docker (poglavje 4.3), ki ima dostop samo do študentske mape. Znotraj zabojnika se izvede še zadnji korak predpriprave, kjer se oddana naloga prilagodi z datoteko za gradnjo sistema (poglavje 4.4). Sledi začetek testiranja, ki se izvede po principu črne škatle, in sicer po metodi testiranja enot. Enote so metode, ki jih implementirajo študenti. Za posamezno enoto je zapisanih več testov (poglavje 5), katerih povratne informacije se shranijo v t. i. datoteki odzivov. Testiranje se ovrednoti kot uspešno samo v primeru, če je testirana rešitev uspešno prestala vse teste. Sistem ima za izvajanje testov predvideno tudi časovno omejitev, znotraj katere se morajo testi zaključiti. V nasprotnem primeru se testiranje prekine in test se ovrednoti kot neuspešen.

Šesta faza je objava rezultatov. Uporabi se ocenjevalna razpredelnica, katere prenos je mogoč iz spletne učilnice. Razpredelnica vsebuje podatke za vse študente, ki so bili vključeni v nalogo. Za vsakega študenta z oddano nalogo se izpolnita dva podatka: komentar in ocena. Podatka pridobimo iz datoteke odzivov. S komentarjem se študentu podajo povratne informacije testov. Samo ocenjevanje pa poteka binarno – študent lahko pridobi vse točke ali nič točk. Ocena se določi glede na to, ali je študentova rešitev uspešno prestala vse teste ali ne. Dopolnjeno razpredelnico upravljavci sistema na koncu naložijo v spletno učilnico, s čimer se zaključi ocenjevanje naloge.

4.1. Funkcionalne zahteve in priprava testov

Naloge pri posameznih snoveh iz programiranja morajo biti sestavljene z namenom osvajanja točno določenih konceptov. Na primer preverjanje znanja

uporabe podatkovnih struktur (implementiranih v programerskih knjižnicah) zajema tako preverjanje njihovih metod kot tudi širši pomen uporabe le-teh znotraj algoritmov. Ko je znanje, ki ga mora študent osvojiti, dobro opredeljeno, pa se lahko oblikujejo specifikacije naloge ter funkcionalne zahteve, ki jih mora naloga vsebovati. Seznam funkcionalnih zahtev nato uporabimo kot osnovo za pisanje testov s pomočjo TDD.

Celoten postopek (Beck, 2003) razvoja programske kode po TDD zajema naslednje korake:

1. Zapiše se test, namenjen preverjanju majhnega koščka funkcionalnosti naloge, in se ga požene.
2. Test pade.
3. Zapiše se najmanjša možna koda, ki zadošča testu.
4. Izvedejo se vsi do takrat napisani testi (čemur pravimo tudi regresijsko testiranje), ki morajo biti zdaj uspešni.
5. Izvede se preoblikovanje programske kode (angl. Refactoring), ki izboljša njeno notranjo strukturo, vendar ne spremeni obnašanja navzven. Ponovno se izvedejo vsi testi.
6. Če niso izpolnjene vse na začetku zapisane funkcionalne zahteve, ponovimo vse do zdaj zapisane korake s pisanjem novega testa za neimplementirano funkcionalnost. Sicer je razvoj zaključen.

Na tem mestu podajmo praktičen primer testa za implementacijo ene izmed funkcionalnosti naloge:

Študent mora osvojiti znanje uporabe statične podatkovne strukture Tabela. Ena izmed njenih osnovnih metod je metoda pripravi(velikost), ki poskrbi za začetno inicializacijo tabele s podano celoštevilsko vrednostjo velikost, ki jo metoda prejme kot argument. Funkcionalnost na seznamu zahtev za implementacijo je naslednja: »Metoda pripravi(velikost) mora pri prejetju vrednosti velikosti, ki je manjša ali enaka 0, uvodno nastaviti ter vrniti celo-številsko polje ničelne velikosti.«

Test, ki bo izpolnjeval preverjanje te funkcionalnosti, po izvedbi prvih dveh korakov vsebuje naslednjo kodo:

```
// 1. Velikost polja je negativno število
int velikost = -5;
if (Tabela.pripravi(velikost).Length != 0)
    return false;

// 2. Velikost polja je 0
int velikost2 = 0;
if (Tabela.pripravi(velikost2).Length != 0)
    return false;

return true;
```

Algoritem 4.1: **Test za preverjanje pravilnosti implementacije metode pripravi(velikost).**

Implementacija metode *pripravi(velikost)* z izpolnjeno opisano funkcionalno zahtevo po izvedbi tretjega koraka TDD cikla vsebuje:

```
int[] polje = new int[0];
if (velikost <= 0)
       return polje;
```

Algoritem 4.2: **Del kode, s katero bo metoda pripravi(velikost) izpolnjevala funkcionalno zahtevo.**

Ker imamo do zdaj napisan samo en test, lahko četrti korak preskočimo (pri dodanih novih testih pa se bo ta korak vedno izvedel). Sledi peti korak cikla TDD, kjer izvedemo preoblikovanje napisane kode –

odstranitev nepotrebne spremenljivke polje (zmanjša se število vrstic kode, sprostila se je deklaracija spremenljivke pri nespremenjenem delovanju) – ter izvedemo vse teste.

```
if (velikost <= 0)
    return new int[0];
```

Algoritem 4.3: **Del kode, s katerim bo metoda pripravi(velikost) izpolnjevala funkcionalno zahtevo.**

Peti korak cikla (preoblikovanje) je v osnovi izredno pomemben gradnik TDD-razvoja. Na danem primeru je zaradi preprostosti sicer prišlo samo do odstranitev spremenljivke, vendar bi z nadaljevanjem šestega koraka (dodajanje novih zahtev) hitro prišlo do kompleksnejših preoblikovanj, ki bi imele pomemben vpliv na strukturo končne naloge (npr. *pripravi(velikost)* bi postala razredna metoda, spremenila bi se vidnost metode, metoda bi se lahko razdelila na več delov itd.).

4.2. Preverjanje plagiatorstva

Skripta za preverjanje plagiatorstva kot vhod prejme absolutno pot do mape študentskih oddaj, ločenih po programskih jezikih. Za vsak jezik se ustvari seznam nalog, ki se posreduje na strežnik Moss s pomočjo datoteke *moss.pl* (del sistema Moss). Odgovor sistema Moss je datoteka HTML z rezultati.

```
#!/bin/bash
#vhodni parameter ($1) predstavlja pot do oddaj
assignment="$1"
# ustvarimo seznam nalog za C++
parameters=""
for source in $assignment/cpp/*; do
    parameters="$parameters $source"
done
# pošljemo seznam nalog na strežnik Moss
results=$(./moss.pl -l cc $parameters | tail -n 1)
# pridobimo rezultate preverjanja v obliko HTML
wget -O ${assignment}_cpp.html $results 2> /dev/null
# ustvarimo seznam nalog za C#
parameters=""
for source in $assignment/cs/*; do
    parameters="$parameters $source"
done
# pošljemo seznam nalog na strežnik Moss
results=$(./moss.pl -l csharp $parameters | tail -n 1)
# pridobimo rezultate preverjanja v obliko HTML
wget -O ${assignment}_cs.html $results 2> /dev/null
```

Algoritem 4.4: **Skripta za preverjanje plagiatorstva s sistemom Moss.**

4.3. Zabojnik Docker

Zabojnik Docker je v sistemu uporabljen z namenom izolacije testov, zmanjševanja zunanjih vplivov na testiranje ter za varovanje računalnika upravljavca sistema pred morebitnimi zlorabami.

Za zagon zabojnika je bilo treba ustvariti sliko Docker (Algoritem 4.5), kjer je bila kot osnova uporabljena uradna slika okolja Mono (Docker Inc, 2019). Okolje Mono je odprtokodna implementacija .Net ogrodja in omogoča razvoj aplikacij, napisanih v jeziku

C# na operacijskem sistemu Linux. Uradna slika je razširjena z razvojnimi orodji (angl. build-essential). Gradnja slike (dodatne informacije o tem koraku najdete na (Docker Inc., 2019)) se izvede enkrat in objavi na središču Docker (Docker Inc., 2019).

```
#osnovna slika - podpora za C#
FROM mono
#dodatek - podpora za C++ in makefile
RUN apt-get update && apt-get -y install build-essential
```

Algoritem 4.5: **Izdelava slike Docker. Zabojnik se zažene ločeno za vsakega študenta z ukazom:**

Zabojnik se zažene ločeno za vsakega študentra z ukazom:

```
docker run --name Naziv -v \"" + pot + ":/app\" slika bash -c \"cd /app/ && make\""
```

kjer se s parametrom *name* poda ime zabojnika (v našem primeru je to Naziv). Parameter *v* pritrdi mapo s študentsko oddajo (parameter pot) na pot */app/* znotraj zabojnika. S tem se aplikaciji znotraj zabojnika omogoči dostop do vseh datotek v mapi s študentsko oddajo. Parameter *slika* vsebuje povezavo do slike Docker na središču Docker. Na koncu je podan še skriptni ukaz, ki se izvede ob zagonu zabojnika. Ob izvedbi ukaza pride do premika v mapo *app* in zagona skripte za gradnjo sistema.
Zabojnik je po koncu uporabe treba vedno odstraniti, čemur je namenjen ukaz:

```
docker rm -f Naziv
```

4.4. Datoteki za gradnjo sistema

Za zagon testov se uporabljata skripti za gradnjo sistema: C# (Algoritem 4.6) in C++ (Algoritem 4.7). Skripti najprej iz študentskih nalog odstranita vse nepotrebne knjižnice. Sledi zamenjava imena glavnega podprograma iz *main* v *xmain* (podprogram *main* je namreč že vključen v testih). Po zamenjavah se združita datoteka s testi in študentska naloga. Združena programska koda se prevede v izvajalno datoteko, ki se nato zažene. Odzivi testov se shranijo v datoteko *output*.

```
SRCFILE_EXT = cs
SOURCES = $(wildcard *.$(SRCFILE_EXT))
TARGETS = $(SOURCES:.$(SRCFILE_EXT)=.exe)
# Mono.NET prevajalnik
CC = mcs

.PHONY: all
all: $(TARGETS)

%.exe: %.$(SRCFILE_EXT)
    @sed 's/namespace .*$//namespace NalogaTestiranje/g' $< > $*.ccs0
    @sed 's/Main/xmain/g' $*.ccs0 > $*.ccs
    @cat _test/test.cs >> $*.ccs
    @$(CC) _test/TestFramework.cs /out:$*.exe $(*F).ccs /NoWarn:219
    mono ./$*.exe
```

Algoritem 4.6: **Datoteka za gradnjo sistema za C#.**

```
SRCFILE_EXT = cpp
SOURCES = $(wildcard *.$(SRCFILE_EXT))
TARGETS = $(SOURCES:.$(SRCFILE_EXT)=.exe)
#C++ prevajalnik
CC = g++

.PHONY: all
all: $(TARGETS)

%.exe: %.$(SRCFILE_EXT)
    @sed 's/stdafx.h/cstdio/g' $< > $*.cc0
    @sed -i '1i #include <cstdlib>' $*.cc0
    @sed -i -r '/using namespace std/! s/namespace .*([;{])/namespace
NalogaTestiranje\1/g' $*.cc0
    @sed 's/main/xmain/g' $*.cc0 > $*.cc1
    @sed 's/#include "pch.h"//g' $*.cc1 > $*.cc
    @cat  _test/test.cpp >> $(*F).cc
    @$(CC) -o $*.exe $(*F).cc
    ./$*.exe
```

Algoritem 4.7: **Datoteka za gradnjo sistema za C++.**

4.5. Namizna aplikacija

Za lažjo uporabo sistema je bila s programskim ogrodjem Electron (Electron, 2019) razvita prenosljiva namizna aplikacija, ki deluje kot vodič skozi opisan postopek. Aplikacija izvede samodejen prenos nalog, razvrstitev oddaj v ločene mape ter zgoraj opisane skripte.

5 PREDSTAVITEV OGRODJA IN PRIMERA

Preverjanje pravilnosti študentskih oddaj poteka na osnovi vnaprej pripravljenih testov enot. Enote so v našem sistemu metode, ki jih študenti v predlogi implementirajo v skladu s podanimi navodili. Pri uporabi našega testnega ogrodja študentu ni treba dodati niti ene vrstice kode, vendar pa ogrodje študentu vseeno omogoča implementacijo lastnih metod in razredov.

Osnovno ogrodje za testiranje je skupno vsem nalogam in je implementirano v obeh jezikih: C# (Algoritem 5.1) in C++ (Algoritem 5.2). Koda je pri obeh jezikih zelo podobna, s čimer se olajšata sestavljanje nalog in pisanje testov. Glavne komponente ogrodja so: razred *Scenarij* (od sedaj naprej scenarij), metodi za zagon testov in testne metode. V ogrodju je Scenarij razred, ki vsebuje podatek o nazivu testirane enote, kratek opis scenarija ter referenco na testno metodo, ki se bo izvedla. V metodi za zagon testov (metoda *test_vse*) se najprej deklarirajo vsi scenariji, ki se dodajo na seznam. Sledijo zaporedni klici metode *zazeni_test_scenarij*.

```
public class Scenarij{
    public string opis_scenarija;
    public string ime_testirane_enote;
    public System.Func<string, string, bool> referenca;
    public Scenarij(string opis,
        string ime_enote,
        Func<string, string, bool> testna_metoda){
        opis_scenarija = opis;
        ime_testirane_enote = ime_enote;
        referenca = testna_metoda;
    }}
public static bool test_vse (){
    //so vsi testi prestali testiranje?
    bool uspesni = true;
    //seznam testnih scenarijev
    var testniScenariji = new LinkedList<Scenarij>();
    //deklaracija testnega scenarija
    var test_scenarij = new Scenarij(
        "Queue<Karta> zdruzi(Queue<Karta> posl, Queue<Karta> ekon)",
        "Zdruzi prazni vrsti potnikov",
        test_zdruzi_prazni_vrsti_potnikov);
    testniScenariji.AddLast(test_scenarij);
    //dodamo se druge testne scenarije
    //....
    //zagon testov
    foreach (Scenarij trenutni in testniScenariji){
        if (!TestFrameworkClass.zazeni_test_scenarij(
                trenutni.opis_scenarija,
                trenutni.ime_testirane_enote,
                trenutni.referenca))
            uspesni = false;
    }
    return uspesni;}
```

Algoritem 5.1: Razred Scenarij in funkcija za zagon testov v C#.

```
class Scenarij{
    public:
        string opis_scenarija;
        string ime_testirane_enote;
        bool (*referenca)(const string, const string);
        Scenarij(string opis,
            string ime_enote,
            bool (*testna_metoda)(const string, const string)){
            opis_scenarija = opis;
            ime_testirane_enote = ime_enote;
            referenca = testna_metoda;
        }};
bool test_vse() {
    //so vsi testi prestali testiranje?
    bool uspesni = true;
    //seznam testnih scenarijev
    list<Scenarij> testniScenariji;
    //deklaracija testnega scenarija
    Scenarij test_scenarij(
        "Zdruzi prazni vrsti potnikov",
        "queue<Karta> zdruzi(queue<Karta> &posl, queue<Karta> &ekon)",
        test_zdruzi_prazni_vrsti_potnikov);
    testniScenariji.push_back( test_scenarij );
    //dodamo se druge testne scenarije
    //....
    //zagon testov
    for(Scenarij test_scenarij : testniScenariji){
        if (!zazeni_test_scenarij(
            test_scenarij.opis_scenarija,
            test_scenarij.ime_testirane_enote,
            test_scenarij.referenca))
            uspesni = false;
    }
    return uspesni;}
```

Algoritem 5.2: Razred Scenarij in funkcija za zagon testov v C++.

Metoda *zazeni_test_scenarij* (Algoritem 5.3) je del osnovnega ogrodja, njena naloga je izvedba testa s klicem implementirane testne metode. Metoda vrne informacijo o uspešnosti izvedenega testa. Zaradi možnosti napak med izvajanjem (angl. run time errors) je klic metode obdan s prestreznim delom kode (angl. try…catch), ki v primeru napake v datoteko odzivov zapiše poročilo o napaki, ime testirane enote ter opis scenarija, celoten test pa se tudi označi kot neuspešen.

```
public static bool zazeni_test_scenarij(
    string opis_scenarija,
    string ime_metode,
    Func<string, string, bool> testna_metoda){
    bool uspesen = false;
    try { uspesen = testna_metoda(opis_scenarija, ime_metode); }
    catch (Exception e){
        izpisiNapakoVDatotekoOdzivov(opis_scenarija, ime_metode, e);
        uspesen = false;
    }
    return uspesen;
}
```

Algoritem 5.3: **Funkcija za zagon posameznega testa.**

Testna metoda (angl. test case) vsebuje implementacijo testa in se drži vzorca AAA. Najprej se pripravijo vhodni podatki, sledi izvedba testirane enote in na koncu še preverjanje, ali se rezultati enote ujemajo s pričakovanimi rezultati. Vsaka testna metoda sprejme dva parametra: opis scenarija in naziv testirane enote. Ta podatka se v primeru neuspešno izvedenega testa skupaj z razlago zapišeta v datoteko odzivov. Metoda vrne informacijo o uspešnosti testa. Primer testne metode je predstavljen v Algoritem 5.4.

```
private static bool test_zdruzi_prazni_vrsti_potnikov(
        string opis_scenarija,
        string ime_metode){
    //Arrange - urejanje
    bool uspesno = true;
    var poslovni = new Queue<Karta>();
    var ekonomski = new Queue<Karta>();
    var sim = new Simulacija(); //razred v nalogi

    //Act - izvajanje
    var dejanskoZdruzeni = sim.zdruzi(poslovni, ekonomski);

    //Assert - potrjevanje
    if (dejanskoZdruzeni.Count != 0){
        TestFrameworkClass.izpisiNapakoVDatotekoOdzivov(
            opis_scenarija,
            ime_metode,
            "dodala potnika/-e v izhodno vrsto zdruzenih potnikov.");
        uspesno = false; }
    if (poslovni.Count != 0){
        TestFrameworkClass.izpisiNapakoVDatotekoOdzivov(
            opis_scenarija,
            ime_metode,
            "dodala potnika/-e v vhodno vrsto poslovnih potnikov.");
        uspesno = false; }
    if (ekonomski.Count != 0){
        TestFrameworkClass.izpisiNapakoVDatotekoOdzivov(
            opis_scenarija,
            ime_metode,
            "dodala potnika/-e v vhodno vrsto ekonomskih potnikov.");
        uspesno = false; }
    return uspesno;
}
```

Algoritem 5.4: **Primer testne metode.**

6 DISKUSIJA

Predstavljen sistem se od sorodnih razlikuje predvsem v sočasni uporabi dveh programskih jezikov (C# in C++), kar je tudi največja prednost našega sistema. Omogoča objektivno ocenjevanje, kar pa je pri ročnem pregledovanju in testiranju nalog pri več kot 100 študentih in več članih pedagoškega osebja zelo težko doseči. Prav tako se skrajšata čas pregledovanja in testiranja nalog (Tabela 1) ter čas objave rezultatov, ki vsebujejo poenotene odzive o morebitnih napakah. Odzivi študentom povedo, v kateri metodi in pri katerem testu je prišlo do morebitne napake, s čimer lahko pedagoško osebje oziroma študenti hitreje najdejo mesto napake. To jim je v pomoč pri popravljanju naloge pred ponovno oddajo iste naloge.

Slabosti uporabe sistema so: a) podaljšanje časovnega vložka, potrebnega za pripravo naloge, saj mora pedagoško osebje spisati funkcionalne zahteve, implementirati rešitev s pomočjo TDD in sestaviti predlogo, b) testov se ne objavi, tako da morajo študenti testirati lastno kodo, c) binarno ocenjevanje, kjer že en neuspešen test pomeni nič točk pri nalogi in d) sistem je bolj koristen pri predmetih z večjim številom študentov.

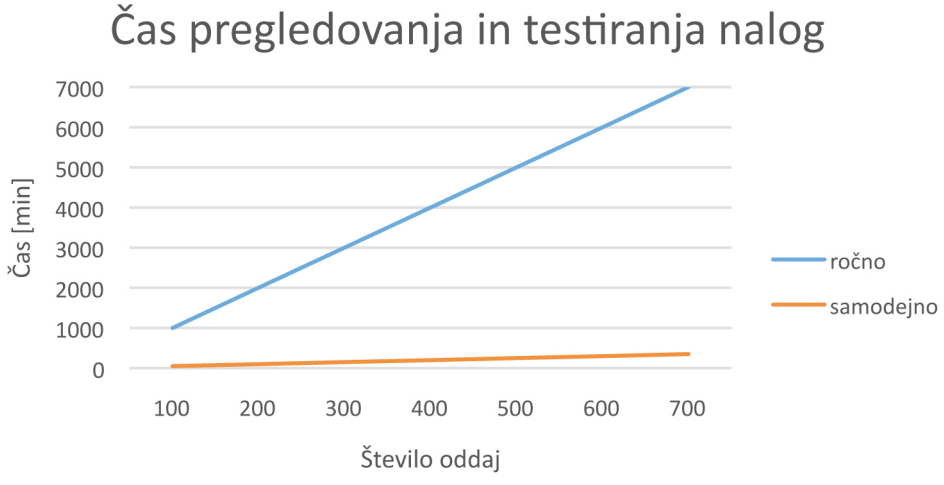
Pri ročnem pregledovanju in testiranju nalog je treba poudariti, da se naloge razlikujejo po težavnosti (Tabela 1), s čimer se čas testiranja poveča. Iz tega razloga smo v Slika 1 za ročni pregled in testiranje nalog uporabili povprečen čas. Za ročni pregled in testiranje ene oddane naloge se glede na naše izkušnje v povprečju porabi približno 10 minut, medtem ko se za samodejni pregled in testiranje ene oddane naloge potrebuje največ 30 sekund.

Tabela 1: **Prikaz težavnosti in časa ročnega pregledovanja in testiranja ene oddaje naloge**

Vrstilec	Stopnja težavnosti	Čas ročnega pregledovanja in testiranja [min]
Naloga 1	4	5
Naloga 2	3	5
Naloga 3	7	10
Naloga 4	8	10
Pregledna naloga	10	20
Povprečje	6,4	10

Slika 1 prikazuje razliko v času med ročnim ter samodejnim pregledovanjem in testiranjem nalog glede na število oddanih nalog, kjer se v povprečju odda 650 nalog na semester. Slika 1 prikazuje potreben čas za pregledovanje in testiranje oddane naloge za enega pedagoškega delavca. Pri samodejnem pregledovanju in testiranju oddane naloge se čas z več pedagoškimi delavci ne bi spremenil, bi se pa porazdelil pri ročnem.

Med študenti smo izvedli tudi krajšo anketo o uporabi našega sistema. Večjih pripomb glede sestave in razumevanj nalog niso imeli. Podali pa so željo, da bi imeli dostop do testov, s čimer bi lahko naloge osebno testirali, preden jih oddajo v ocenjevanje. Kritika sistema je bila izrečena glede poenotenih odzivov o nepravilno delujoči nalogi, saj se študentje niso znali orientirati, kako odpraviti napako (mnenje/utemeljitev avtorjev članka: študentom je v odzivu podana informacija o metodi in krajši opis testa, pri katerem je prišlo do napake, ne pa točna lokacija napake v programski kodi).



Slika 1: **Primerjava časov ročnega in samodejnega pregledovanja in testiranja oddanih nalog**

7 SKLEP

Za razvoj lastnega ogrodja za testiranje enot smo se odločili, ker smo želeli imeti sistem, ki bo notranje podpiral dva programska jezika (C# in C++), navzven pa ponudil identične funkcionalnosti ter uporabniško izkušnjo. S tem se je dosegla lažja integracija implementiranega sistema v študijski proces, pohitrilo pa se je tudi pregledovanje in testiranje nalog. Ena izmed primarnih zahtev, ki smo si jo zadali pri implementaciji sistema, je bila tudi ta, da študentom ne dvignemo kognitivne obremenitve pri osvajanju novega znanja še s prilagajanjem na naše ogrodje. To nam je uspelo, saj študentje ni treba prilagajati svojega stila programiranja predlogi, če tega ne želijo. Jim pa je vseeno prepuščena svoboda v primeru, če želijo dodati novi razredi, strukture, metode itd.

V prihodnosti želimo v sistem dodati tudi funkcionalnosti, ki bi študentom omogočile samostojno rabo celotnega sistema, vendar v okrnjeni obliki (brez vseh testov), kot tudi dodati možnosti za predtestiranja lastnih rešitev za bolj splošne napake (neveljavna sprememba strukture predloge; metoda vsebuje nedovoljene klice iz programskih knjižnic, kot je npr. Array.Sort(); …).

LITERATURA

[1] Aiken, A. (15. 12 2018). *A System for Detecting Software Similarity*. Pridobljeno 9. 9 2019 iz https://theory.stanford.edu/~aiken/moss/

[2] Auffarth, B., López-Sánchez, M., Campos i Miralles, J., & Puig, A. (2008). *System for automated assistance in correction of programming exercises (sac).* Proceedings of CIDUI 2008, (str. 104-113). Lleida.

[3] Beck, K. (2003). *Test-driven development: by example.* Addison-Wesley Professional.

[4] Bowyer, K. W., & Hall, L. O. (1999). *Experience using „MOSS“ to detect cheating on programming assignments.* FIE'99 Frontiers in Education. 29th Annual Frontiers in Education Conference. Designing the Future of Science and Engineering Education. Conference Proceedings (IEEE Cat. No. 99CH37011) (str. 13B3-18). IEEE.

[5] Coursera Inc. (1. 09 2019). *Coursera.* Pridobljeno iz https://www.coursera.org/

[6] Docker Inc. (25. 8 2019). *Mono Docker slika.* Pridobljeno 25. 8 2019 iz Spletna mesto uradne Mono slike: https://hub.docker.com/_/mono/

[7] Docker Inc. (02. 09 2019). *Docker izgradnja slike.* Pridobljeno iz https://docs.docker.com/engine/reference/commandline/image_build/

[8] Docker Inc. (20. 8 2019). *Docker središče.* Pridobljeno 20. 08 2019 iz https://hub.docker.com/

[9] Docker Inc. (10. 9 2019). *Enterprise Container Platform | Docker.* Pridobljeno 25. 8 2019 iz https://www.docker.com/

[10] Douce, C., Livingstone, D., & Orwell, J. (2005). *Automatic test-based assessment of programming: A review*. Journal on Educational Resources in Computing (JERIC), 3.

[11] Edwards, S. H. (2003). *Using test-driven development in the classroom: Providing students with automatic, concrete feedback on performance.* Proceedings of the international conference on education and information systems: technologies and applications EISTA. 3. Citeseer.

[12] edX Inc. (1. 9 2019). *edX | Free Online Courses by Harvard, MIT, & more.* Pridobljeno 1. 9 2019 iz https://www.edx.org/

[13] *E-izobraževanje - Wikipedija, prosta enciklopedija.* (28. 8 2019). Pridobljeno 1. 9 2019 iz https://sl.wikipedia.org/wiki/E-izobra%C5%BEevanje

[14] Electron. (1. 9 2019). *ElectronJS.* Pridobljeno iz https://electronjs.org/

[15] Gradišnik, M., & Majer, Č. (2016). *Mikrostoritve in zabojniki Docker.* V M. Heričko, & K. Kous (Ured.), OTS 2016 Sodobne tehnologije in storitve, (str. 10-20). Maribor.

[16] Ihantola, P., Ahoniemi, T., Karavirta, V., & Seppälä, O. (2010). Review of recent systems for automatic assessment of programming assignments. *Proceedings of the 10th Koli Calling International Conference on Computing Education Research* (str. 86-93). Koli: ACM.

[17] *LinkedIn Learning.* (1. 9 2019). Pridobljeno 1. 09 2019 iz https://www.linkedin.com/learning/topics/business

[18] *Massive open online course - Wikipedia.* (1. 9 2019). Pridobljeno 1. 09 2019 iz https://en.wikipedia.org/wiki/Massive_open_online_course

[19] Myers, G. J., Sandler, C., & Badgett, T. (2011). *The art of software testing.* John Wiley & Sons.

[20] Nichols, M. (2007). No. 1: E-Learning in context. *E-Primer Series, 27.*

[21] Romli, R., Sulaiman, S., & Zamli, K. (2010). Automatic programming assessment and test data generation a review on its approaches. *2010 International Symposium on Information Technology* (str. 1186-1192). IEEE.

[22] Udacity, Inc. (1. 9 2019). *Learn the Latest Tech Skills; Advance Your Career | Udacity.* Pridobljeno 1. 09 2019 iz https://www.udacity.com/

[23] Udemy, Inc. (1. 9 2019). *Online Courses - Learn Anything, On Your Schedule | Udemy.* Pridobljeno 1. 9 2019 iz https://www.udemy.com/

[24] Zeller, A. (2000). Making students read and review code. *ACM SIGCSE Bulletin* (str. 89-92). ACM.

[25] Zemljak, A., Novak, D., Čep, A., & Verber, D. (2016). Preverjanje pravilnosti študentskih nalog programiranja s testiranjem enot. V M. Orel (Ured.), *Sodobni pristopi poučevanja prihajajočih generacij* (str. 556-564). Ljubljana: Polhov Gradec : Eduvision. Pridobljeno iz http://www.eduvision.si/Content/Docs/Zbornik%20prispevkov%20EDUvision_2016_SLO.pdf

▪

Aleš Čep je asistent in doktorski študent na Fakulteti za elektrotehniko, računalništvo in informatiko Univerze v Mariboru. Njegova raziskovalna področja vključujejo umetno inteligenco v igrah ter evolucijsko računanje.

▪

Damijan Novak je leta 2011 diplomiral na Fakulteti za elektrotehniko, računalništvo in informatiko Univerze v Mariboru. Trenutno je asistent in doktorski študent z večjim številom znanstvenih člankov in prispevkov. Njegova raziskovalna področja obsegajo računsko kompleksnost v igralnih svetovih, umetno inteligenco v igrah ter vzpodbujevalno učenje (veja strojnega učenja). Na mednarodni znanstveni konferenci CGAT 2017 mu je bilo za članek "An aspect-based classification of real-time strategy game worlds" podeljeno tudi priznanje za najboljši študentski znanstveni prispevek.

▪

Jani Dugonik je leta 2010 diplomiral in leta 2013 magistriral na Fakulteti za elektrotehniko, računalništvo in informatiko, ki je članica Univerze v Mariboru. Trenutno je doktorski študent in asistent na fakulteti za elektrotehniko, računalništvo in informatiko. Njegova raziskovalna področja vključujejo evolucijsko računanje, optimizacijo, procesiranje naravnega jezika in globoko učenje.