Универзитет „Св. Кирил и Методиј“ во Скопје
ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО

Паметно генерирање на прашања за програмски јазици

- Докторска дисертација -

Кандидат:
м-р Емил Станков

Ментор:
проф. д-р Ана Мадевска Богданова

Скопје, 2021

Тема: **Паметно генерирање на прашања за програмски јазици**

Автор: **м-р Емил Станков**

Научна област: **Информатика**

Ментор: **Проф. д-р Ана Мадевска Богданова**

редовен професор на Факултетот за информатички науки и компјутерско инженерство, УКИМ, Скопје

Членови на комисија: **Проф. д-р Невена Ацковска**

редовен професор на Факултетот за информатички науки и компјутерско инженерство, УКИМ, Скопје

Проф. д-р Ана Мадевска Богданова

редовен професор на Факултетот за информатички науки и компјутерско инженерство, УКИМ, Скопје

Проф. д-р Марјан Гушев

редовен професор на Факултетот за информатички науки и компјутерско инженерство, УКИМ, Скопје

Проф. д-р Миле Јованов

вонреден професор на Факултетот за информатички науки и компјутерско инженерство, УКИМ, Скопје

Проф. д-р Азир Алиу

вонреден професор на Факултетот за современи науки и технологии, Универзитет на Југоисточна Европа, Скопје

Датум на одбрана: _____

м-р Емил Станков

Паметно генерирање на прашања за програмски јазици

Резиме:

Унапредувањето на наставата по програмирање е активност која станува сè поактуелна. Проверката на знаењето на студентите кои слушаат воведни курсеви за програмирање делумно може да се изведува преку презентирање на едноставни (фрагменти од) програмски кодови на студентите. Една од наставните цели кај овие курсеви е студентите да бидат оспособени да разбираат веќе напишан програмски код, па според тоа тие треба да бидат способни да одговорат точно на прашањето: „Кој е излезот од даденият код?“. При подготовката на кодовите, наставниците треба да бидат свесни за нивната сложеност. Уште повеќе, при подготовката на повеќе верзии од еден ист тест (заради проверка на знаење на голема група од студенти), тие треба да се обидуваат да составуваат прашања што содржат програмски кодови со иста или слична сложеност за сите студенти. Едно можно решение на овој проблем е да се користи автоматско генерирање на прашања што содржат програмски кодови.

Во оваа докторска дисертација е дефиниран нов модел за паметно автоматско генерирање на прашања што содржат програмски кодови. Примената на овој модел ќе овозможи да се постигне целосно објективна и фер проверка на знаење на кој било испит од воведен курс за програмирање, бидејќи на сите студенти кои го полагаат испитот ќе може да им се поставуваат прашања со иста или многу слична сложеност т.е. прашања кои побаруваат исто ниво на знаење и вложување на приближно ист напор за да се даде точен одговор. Со тоа моделот ќе придонесе за значително подобрување на квалитетот на процесот на проверка на знаење кај воведните курсеви за програмирање.

Со цел да се постигне конзистентност на комплексноста во процесот на автоматско генерирање на прашања за курсевите за програмирање, во рамките на овој модел се предлага употреба на нова метрика за автоматско мерење на сложеност на програмските кодови. Новата метрика, дефинирана во оваа дисертација, ја разгледува сложеноста на даден програмски код од аспект на напорот што треба да го вложи студентот за рачно да го пресмета излезот, ако се познати вредностите на сите програмски променливи. Таа ја мери сложеноста користејќи когнитивни тежински вредности придружени на секој од

операторите и контролните структури (наредби) во кодот, кои треба да ги претставуваат нивните „сложености“, во смисла на напорот и времето што треба да го потроши студентот за рачно да ги изврши соодветните операции/наредби и да ги пресмета соодветните резултати. Во дисертацијата се претставени аргументи кои потврдуваат дека метриката е соодветна за разгледуваниот проблем.

Исто така, во оваа дисертација се опишани наоѓањата од истражувањето коешто беше спроведено со цел да се определат соодветни тежински вредности за аритметичките оператори. Главна цел на ова истражување беше подобрување на прецизноста на пресметувањето на сложеноста на кодовите за потребите на автоматското генерирање на прашања што содржат програмски кодови, преку определување на тежински вредности за основните аритметички операции со наједноставни типови на операнди. Овде се претставени резултатите и заклучоците од експериментите кои беа спроведени во рамките на истражувањето.

Конечно, во дисертацијата е претставен и нов софтверски систем за паметно автоматско генерирање на прашања што содржат програмски кодови. За даден почетен програмски код внесен од страна на наставник, системот ја пресметува сложеноста користејќи ја предложената метрика, и со примена на соодветни техники на модификација генерира посакуван број на нови програмски кодови кои имаат иста или произволно блиска сложеност еден со друг. Овој систем може да се користи за автоматско креирање на соодветни прашања што содржат програмски кодови со конзистентна сложеност, па како таков претставува имплементација на предложениот модел за паметно автоматско генерирање на прашања. Според тоа, истиот може да се употребува во практика за потребите на процесот на проверка на знаење кај воведните курсеви за програмирање.

Emil Stankov, MSc.

Smart task generation for programming languages

Abstract:

The process of teaching programming receives significant attention nowadays. Assessment of students' knowledge in introductory programming courses can (partly) be done by presenting simple (fragments of) source codes. One of the learning goals in these courses is students to become able to understand and comprehend an already written program code. Hence, they should be able to correctly answer the question: "What is the output of the given code?" When preparing the source codes, teachers must be aware of their complexity. Particularly, when preparing many different versions of the same test (to assess a huge number of students), they have to provide same or similar complexity tasks (questions) for all students. A possible solution to this problem is to turn to automatic generation of tasks containing program codes.

This doctoral dissertation defines a new model for smart automatic generation of tasks containing program codes. The application of this model will enable achievement of a completely objective and fair assessment on any introductory programming course exam, since all students taking the exam will be asked questions of same or very similar complexity, i.e. questions that require the same level of knowledge and approximately the same effort to provide a correct answer. Thus, the model will contribute to a significant improvement in the quality of the assessment process in the introductory programming courses.

In order to achieve complexity consistency in the process of automatic production of questions for programming courses, within this model, the use of a new metric for automatic measurement of complexity of program codes is proposed. The new metric, defined in this dissertation, considers the source code complexity from a perspective of the student's effort required for manual calculation of the program output, if the values of all the program variables are known. It measures the complexity using cognitive weight values assigned to each of the operators and control flow statements in the code, which should represent their "complexities", in terms of the effort and time that a particular student needs to spend to manually perform the corresponding operations/statements and calculate the respective results. The dissertation presents arguments which confirm that the metric is suitable for the problem under consideration.

Furthermore, in this dissertation, the findings of the research that was conducted in order to determine an appropriate weight values for the arithmetic operators are also described. The main goal of this research was to improve the accuracy of the code complexity calculation for automatically generated tasks that contain source codes, by determining weight values for the basic arithmetic operations with simplest operand types. Here, the results and conclusions from the experiments that were conducted as part of the research are presented.

Finally, the dissertation introduces a new software system for smart automatic generation of tasks containing program codes. For a given (initial) source code entered by a teacher, the system calculates the complexity using the proposed metric, and by applying appropriate modification techniques, it generates a desired number of new program codes that have the same or arbitrarily close complexity one to another. This system can be used for automatic creation of appropriate tasks that contain program codes of consistent complexity, so, as such – it represents an implementation of the proposed model for smart automatic task generation. Therefore, it can be used in practice for the purposes of the knowledge assessment process in the introductory programming courses.

Благодарност

Би сакал да се заблагодарам на мојот ментор, проф. д-р Ана Мадевска Богданова, за сите нејзини совети, насоки, упатства, предлози и најдобронамерни критики при изработката на оваа докторска дисертација. Таа има значаен придонес за мојата изградба како истражувач, како професионалец, па и како личност, почнувајќи од менторството за мојата дипломска работа, преку магистерскиот труд, па ете – заокружувајќи со дисертацијата.

Голема благодарност упатувам и до проф. д-р Миле Јованов, за сите негови идеи, сугестии, размислувања, несебичното залагање и поддршка во текот на целата моја досегашна академска и научна кариера, а посебно за време на работата на докторската дисертација. Во соработка со менторот, постојано ми даваше корисни совети кои секогаш беа мотивација како да продолжам понатаму.

Конечно, благодарност на моите најсакани, моето најблиско семејство, за огромното разбирање, поддршка и љубов, кои ги добивав постојано и во неограничени количини низ изминативе години. Без вас не би можел да ја чекорам успешно мојата животна патека!

Кратка содржина

Содржина

Слики

Табели

1. Вовед

1.1. Потребата за автоматско генерирање на прашања кај курсевите за програмирање

Во денешно време, кога компјутерите и технологијата се помеѓу фундаменталните двигатели на целокупниот развој на човештвото, изведувањето на настава по програмирање, како и нејзиното континуирано надградување и подобрување, претставуваат активности на кои без сомнение треба да се посвети посебно внимание. Сепак, наставата по програмирање, особено кога станува збор за курсеви посетувани од страна на голем број студенти, носи со себе голем број предизвици, а еден од најсериозните помеѓу нив е секако – проверката на знаењето.

Проверката на знаењето на студентите претставува исклучително важен дел од процесот на едукација. R. Sprinthall, N. Sprinthall и Oja (1998) посочуваат дека проверката на знаењето е клуч на учењето, бидејќи лошо спроведената проверка на знаење повлекува два значајни проблеми: од една страна не ги истакнува доволно прецизно слабите студенти, а од друга страна не претставува предизвик на соодветно ниво за надежните, подобри студенти. Докажан и релативно успешен пристап кој овозможува анализирање на способноста на студентите да пишуваат програмски кодови е примената на методите на автоматско оценување коишто вклучуваат машинска анализа на излезот од студентските програми (Ala-Mutka, 2005; Daly и Waldron, 2004). Но, ова не е единствената наставна цел што треба да се постигне кај почетничките (воведните) курсеви за програмирање, бидејќи секој студент – почетник во програмирањето мора претходно да биде оспособен да разбира веќе напишан програмски код.

Пишувањето на компјутерска програма е релативно тешка когнитивна вештина за совладување, а исто така доста е тешко и да се измери до кој степен некој човек ја поседува оваа вештина (Lister *et al*., 2010; Spiro и Jehng, 1990). И покрај големите напори на голем број наставници, многу студенти сеуште наидуваат на потешкотии при учењето на програмирањето. За да можат да напишат една едноставна програма, тие мора да поседуваат основно знаење од променливи, влез/излез на податоци, контролните структури, вклучително и математички области.

Проверката на знаењето, особено кај почетничките курсеви за програмирање, делумно може да се изведува и преку презентирање на едноставни фрагменти од изворни програмски кодови (кодни фрагменти) на студентите и определување на нивното базично ниво на знаење и разбирање на програмскиот јазик во кои се напишани соодветните фрагменти. Еден соодветен

начин да се постигне ова е преку поставување на прашања со избор од повеќе можни одговори – ПМО прашања (англ. *multiple choice questions* – MCQs), од обликот: „Кој е излезот од дадениот код?“. Многу честа предрасуда во едукативната заедница е дека тестовите составени од ПМО прашања се „лесен“ начин на проверка на знаењето типично применуван од страна на мрзливите наставници. Lister, кој прв предлага ПМО прашања коишто вклучуваат фрагмент од програмски код и прашање во врска со однесувањето на истиот за проверка на знаење кај почетнички курсеви за програмирање (Lister, 2000), потенцира дека „квалитетните ПМО прашања не се дело на мрзливи луѓе“ (Lister и Leaney, 2003). Иако овие типови на ПМО прашања тешко можат да ги оценат способностите како што се решавањето на проблеми, алгоритамското размислување или длабокото логичко резонирање, сепак тие можат да дадат добар увид во разбирањето на основните програмски конструкции на употребениот програмски јазик, како и во разбирањето на некои основни концепти на програмирањето воопшто (Shuhidan, Hamilton и D'Souza, 2010; Simkin и Kuechler, 2005). Според Блумовата таксономија на едукативни цели (Bloom *et al*., 1956), разбирањето е етапа која претходи на примената, па според тоа добро е да се испита способноста за разбирање на студентите, пред да се побара од нив да го применат своето знаење. Исто така, повеќе спроведени истражувања потврдуваат дека студентите не можат да научат да пишуваат програмски кодови без прво да научат да го следат извршувањето на кодовите (англ. *code tracing*), како претходна неопходна вештина (Cunningham *et al*., 2017; Kumar, 2015; Lopez *et al*., 2008).

Она што понекогаш во литературата се истакнува како недостаток на ПМО прашањата (Clark, 2004; Nicol, 2007) е можноста за слепо или информирано погодување на одговорот, со што студентите може да освојат незаслужени поени. Кога станува збор за прашања кои вклучуваат програмски кодови, ова може да биде елегантно избегнато. Скоро секој програмски код, а особено оној преку кој се оценуваат базичните вештини, може да биде конструиран така што како излез ќе дава еден број, еден знак или еден стринг (низа од знаци). На тој начин, преку воведување на прашања со многу краток одговор, може комплетно да се исклучи можноста за погодување од страна на студентите.

За да се постигне објективна и фер проверка на знаење на конкретен испит од некој курс, на сите студенти коишто го полагаат испитот мора да им се постават прашања со иста или многу слична комплексност, односно прашања кои побаруваат исто ниво на знаење и вложување на (приближно) ист напор за да се даде точен одговор. Кога станува збор за воведните курсеви за програмирање и за употребата на претходно споменатите типови прашања, ова го наметнува проблемот на одржување на конзистентност на комплексноста од страна на наставниците при креирањето на прашања што содржат кодни

фрагменти за студентските тестови, особено кога се работи со големи групи студенти. При подготовка на повеќе верзии од еден ист тест (заради проверка на знаење на голема група од студенти), наставниците треба да се свесни за сложеноста на програмските кодови и треба секогаш да се обидуваат да составуваат прашања што содржат кодни фрагменти со иста или слична сложеност. Едно соодветно решение на овој проблем е да се користи автоматско генерирање на прашања.

1.2. Мотивација и цели

Во последниве десетина години направени се голем број истражувања на полето на автоматското генерирање на прашања за потребите на едукативната проверка на знаење (Kurdi *et al.*, 2019). Најголемиот дел од предложените софтверски алатки во литературата, декларирани како алатки за автоматско генерирање на прашања, всушност нудат автоматско генерирање на тестови. Примери за вакви алатки се ExamGen (Rhodes, Bower и Bancroft, 2004) и PAT (Chatzopoulou и Economides, 2010). Иако на прв поглед се чини дека овие два термини („генерирање прашања“ и „генерирање тестови“) претставуваат синоними за еден ист процес, всушност постои значајна разлика: кај алатките за автоматско генерирање на прашања самите прашања се создаваат автоматски, додека алатките кои нудат автоматско генерирање на тестови едноставно на корисникот му обезбедуваат случајно избрано подмножество од една голема база со прашања (претходно креирани од страна на човек).

Traynor и Paul Gibson (2005) опишуваат модел на систем за автоматско генерирање на ПМО прашања со добар квалитет за воведни курсеви за програмирање. Во процесот на креирање на прашања, тие користат два различни пристапи за автоматско генерирање на кодни фрагменти, и за секој таков генериран коден фрагмент тие потоа обезбедуваат ограничен број на „интелигентно одбрани“ можни одговори (излези) за да го комплетираат новоформираното прашање. Главен недостаток на овој модел е фактот што тој не ја зема предвид комплексноста на генерираните програмски кодови во ниеден од двата пристапи, а како последица на тоа – ја занемарува и комплексноста на добиените прашања. Всушност, анализата на постојните системи за генерирање прашања, која ќе биде претставена во Глава 2, покажува дека ниту еден од нив не го разгледува проблемот на сложеноста на прашањата, што е сериозен недостаток во контекст на нивната употреба за проверка на знаење кај курсевите за програмирање.

Со цел да се постигне посакуваната конзистентност на комплексноста во процесот на автоматско генерирање на прашања што содржат кодни фрагменти за курсевите за програмирање, мора да имаме начин на автоматско мерење на сложеноста на програмските кодови. Во литературата дефинирани се голем број

на софтверски метрики кои се користат за мерење на сложеноста на даден програмски код (Chitti Babu, Prasad и Sudhakar, 2013; Kushwaha и A. Misra, 2006; S. Misra, 2007; Yu и Zhou, 2010; Zuse, 1991). Две вакви метрики, коишто се сметаат за најпроминентни и кои се користат доста често, се сложеноста на Halstead (Halstead, 1977) и цикломатската сложеност (позната и како сложеност на McCabe) (McCabe, 1976). Различни метрики мерат различни аспекти на сложеноста на кодот: некои метрики ја мерат пресметковната сложеност (временска и просторна сложеност), други ја мерат симболичката сложеност (големината изразена преку број на кодни линии, број на оператори, итн.), постојат метрики коишто ја мерат структурната сложеност (број на разгранувања, број на циклуси, итн.), сложеноста на податочниот тек (број на операнди, број на променливи), итн. Некои од постојните метрики се обидуваат да ја квантификуваат дури и когнитивната сложеност на програмскиот код (Jakhar и Rajnish, 2014; Shehab *et al*., 2015; Wang, 2009).

Во Глава 2 (поглавје 2.3) ќе биде даден преглед и направена анализа на дел од најважните и најчесто користени претставници на различните категории на метрики за мерење на сложеност на програмските кодови. Еден од заклучоците што е изведен на крајот од оваа глава е дека сите постојни метрики, независно од категоријата на којашто припаѓаат, ја мерат сложеноста земајќи го во предвид комплетниот код, а не само оној дел од кодот кој всушност „ќе биде посетен“ во текот на извршувањето на програмата, при познати вредности на програмските променливи. Според тоа, ниту една од нив не може директно да се примени за наведениот проблем во едукативна околина.

Целта на истражувањето на оваа докторска дисертација е значително подобрување и олеснување на процесот на едукативна проверка на знаење кај воведните курсеви за програмирање, преку дефинирање на соодветен модел за паметно автоматско генерирање на прашања што содржат програмски кодови. Со оглед на тоа што повеќето постојни модели за автоматско генерирање на прашања во суштина нудат автоматско генерирање на тестови, а пак оние кои нудат вистинско автоматско креирање на прашања што содржат програмски кодови не посветуваат внимание на проблемот на сложеноста на кодовите и нејзината конзистентност во текот на генерирачкиот процес, дефинирањето на модел кој ќе ги реши овие проблеми претставува главна мотивација и водечка насока на истражување во дисертацијата. Имплементирањето на дефинираниот модел ќе овозможи автоматизација на процесот на креирање на прашања, при што програмските кодови што ќе се добиваат во текот на овој процес ќе имаат сложеност којашто ќе може да се контролира (од страна на корисникот т.е. наставникот). На ваков начин, на даден испит од некој курс за програмирање ќе може да се гарантира дека секој од студентите што го полага испитот ќе добие прашања со конзистентна сложеност (иста или произволно блиска – по желба на

наставникот), соодветно, со што целосно ќе се овозможи фер и објективна проверка на знаењето.

За да може да се дефинира ваков модел, неопходно е да се дефинира соодветна метрика за мерење на сложеноста на даден програмски код, па ова е една од потцелите на истражувањето на оваа дисертација, пред реализација на главната цел. За да биде применлива во едукативна околина, дефинираната метрика треба да го разгледува секој програмски код од аспект на тоа колкав напор ќе треба да вложи студентот за рачно да го изврши истиот, па поголема пресметана вредност за оваа метрика ќе означува и соодветно поголем напор односно потрошено време за извршување.

Уште една цел на истражувањето е и имплементација на нова софтверска алатка која ќе ја применува дефинираната метрика за генерирање на нови програмски кодови, кои ќе бидат со слична или иста сложеност еден со друг. Оваа алатка ќе може да се користи за автоматско креирање на прашања што содржат програмски кодови со контролирана сложеност, па како таква ќе претставува имплементација на предложениот модел за паметно автоматско генерирање на прашања и ќе може да се користи во практика за потребите на едукативната проверка на знаење кај почетничките курсеви за програмирање.

1.3. Хипотези

Хипотеза 1: можно е дефинирање и имплементација на нов модел за паметно автоматско генерирање на прашања што содржат програмски код, кој ќе овозможи побрза и унифицирана проверка на знаење за голем број студенти кај воведните курсеви за програмирање.

Хипотеза 2: може да се даде нова, подобра и посоодветна метрика за утврдување на сложеност на програмски код што се користи во прашања за проверка на знаењето за одреден програмски јазик.

Моделот, којшто ќе биде дефиниран во рамките на оваа дисертација, ќе вклучува примена на нова, соодветно дефинирана метрика за мерење на сложеноста на даден програмски код. Оваа метрика ќе вклучува емпириски утврдени тежински вредности кои прецизно ќе го одразуваат човечкиот напор при анализа и рачно извршување на кодот, па со тоа ќе биде погодна за користење во разгледуваниот домен. На ваков начин, користејќи ја оваа метрика, врз основа на даден (почетен) програмски код ќе може, со примена на соодветни модификациски техники, да се генерираат нови програмски кодови кои ќе имаат иста или приближно иста сложеност – меѓусебно, но и во однос на почетниот код. Вака добиените програмски кодови со конзистентна сложеност ќе можат понатаму да се користат за креирање на прашања со конзистентна сложеност, кои ќе бидат директно употребливи во процесот на проверка на знаење кај споменатите курсеви за програмирање.

Преку задавање на тестови составени од прашања коишто имаат конзистентна сложеност за секој студент и коишто побаруваат исто ниво на знаење од страна на студентот за да се даде точен одговор, би се овозможила објективна и фер проверка на знаење на секој испит од курс за програмирање. Студентите би можело многу почесто во тек на курсот да бидат тестирани, со што ќе се обезбеди континуирана повратна информација кон нив за нивниот напредок и во услови на масовни курсеви. Притоа, резултатите од овие проверки ќе може да се земаат како релевантен одраз на знаењето на секој студент и како фактор во изведување на нивната конечна оценка. Воедно, автоматизацијата на процесот на креирање на прашања би им овозможила на наставниците кои предаваат програмирање да посветат повеќе време на подготовката на наставниот материјал и изнесувањето на истиот, со што индиректно би придонела за подигнување на квалитетот на целокупниот образовен процес кај овие курсеви.

1.4. Главни придобивки

Главните придобивки од дисертацијата можат да се резимираат на следниот начин:

1. Предложена е нова софтверска метрика за мерење на сложеност на програмски кодови. Предложената метрика ја разгледува сложеноста на даден програмски код од аспект на напорот што треба да го вложи човек за да го разбере кодот (когнитивен процес), рачно да ги изврши сите операции и наредби во него и да го пресмета излезот, ако се познати вредностите на сите програмски променливи. Споредена е и истакната предноста во однос на постојните метрики за мерење на сложеност на програмските кодови;
2. Предложен е нов модел за паметно автоматско генерирање прашања што содржат програмски кодови. Овој модел вклучува употреба на предложената метрика за мерење на сложеност на програмски кодови за генерирање на прашања што содржат програмски кодови со кориснички-контролирана (конзистентна) сложеност. Со примена на овој модел ќе може да се постигне потполно објективна и фер проверка на знаење на кој било испит од воведен курс за програмирање, бидејќи на сите студенти коишто го полагаат испитот ќе може да им се поставуваат прашања со иста или многу слична сложеност (дефинирано од наставникот), односно прашања кои побаруваат исто ниво на знаење и вложување на приближно ист напор за да се даде точен одговор. На ваков начин, предложениот модел ќе придонесе за значително подобрување на квалитетот на процесот на проверка на знаење кај воведните курсеви за програмирање;

3. Развиен е софтверски систем за паметно автоматско генерирање на прашања што содржат програмски кодови. За даден (почетен) програмски код, овој систем ја пресметува сложеноста користејќи ја предложената метрика, и со примена на соодветни модификациски техники генерира посакуван број на нови програмски кодови со иста или произволно блиска сложеност еден со друг (користејќи кориснички-дефинирана прагова вредност за контрола на девијацијата на сложеностите). Овој систем може да се користи за автоматско креирање на соодветни прашања што содржат програмски кодови со конзистентна сложеност, па како таков претставува имплементација на предложениот модел за паметно автоматско генерирање на прашања и може да се користи во практика за потребите на процесот на проверка на знаење кај воведните курсеви за програмирање.

1.5. Објавени релевантни трудови

Stankov, E., Jovanov, M., Gjorgjiev, K., Madevska Bogdanova, A. (2015). "A New Tool for Calculation of a New Source Code Metric". In: *Proceedings of the 12th Conference on Informatics and Information Technology (CiiT2015)*, Faculty of Computer Science and Engineering, UKIM, Skopje, 2015, Bitola, Macedonia, pp. 25-30.

Stankov, E., Jovanov, M., Andonov, J., Madevska Bogdanova, A. (2016). "Improving the Accuracy of the Code Complexity Calculation for Automatically Generated Tasks with Programming Codes". In: *Proceedings of the 2016 IEEE Global Engineering Education Conference (EDUCON 2016)*, pp. 686-692.

Stankov, E., Jovanov, M., Kostadinov, B., Madevska Bogdanova, A. (2015). "A New Model for Collaborative Learning of Programming Using Source Code Similarity Detection". In: *Proceedings of the 2015 IEEE Global Engineering Education Conference (EDUCON 2015)*, pp. 709-715.

Stankov, E., Jovanov, M., Madevska Bogdanova, A. (2017). "Improved Approach for Measuring Complexity of Code Snippets for Introductory Programming Tasks". In: *Proceedings of the 10th annual International Conference of Education, Research and Innovation (ICERI 2017)*, pp. 5892-5899.

Stankov, E., Madevska Bogdanova, A., Ilijoski, B., Jovanov, M. (2018). "A Survey on Software Complexity Metrics in the Context of Their Application in Educational Environment". In: *Proceedings of the 12th annual International Technology, Education and Development Conference (INTED 2018)*, pp. 9395-9404.

Stankov, E., Jovanov, M., Madevska Bogdanova, A., Gusev, M. (2013). "A New Model for Semiautomatic Student Source Code Assessment". *CIT. Journal of Computing and Information Technology*, ISSN 1330-1136, vol. 21, no. 3, pp. 185-194.

Angelovski, D., **Stankov, E.**, Jovanov, M. (2021). "DEMAx Tool Based on an Improved Model for Semiautomatic C/C++ Source Code Assessment". In:

Proceedings of the 6th International Conference on Information and Education Innovations (ICIEI 2021) (in print).

Ackovska, N., Erdosne Nemeth, A., **Stankov, E.**, Jovanov, M. (2015). "Report of the IOI Workshop "Creating an International Informatics Curriculum for Primary and High School Education"". Journal *Olympiads in Informatics*, ISSN 1822-7732, vol. 9, pp. 205-212.

Jovanov, M., **Stankov, E.**, Mihova, M., Ristov, S., Gusev, M. (2016). "Computing as a New Compulsory Subject in the Macedonian Primary Schools Curriculum". In: *Proceedings of the 2016 IEEE Global Engineering Education Conference (EDUCON 2016)*, pp. 680-685.

Jovanov, M., Ackovska, N., **Stankov, E.**, Mihova, M., Gusev, M. (2017). "A Decade of Engineering Computer Engineers". In: *Proceedings of the 2017 IEEE Global Engineering Education Conference (EDUCON 2017)*, pp. 1309-1315.

Jovanov, M., Ilijoski, B., **Stankov, E.**, Armenski, G. (2017). "Creation of Educational Games – Project Based Learning in e-Learning Systems Course". In: *Proceedings of the 2017 IEEE Global Engineering Education Conference (EDUCON 2017)*, pp. 1274-1281.

Ilijoski, B., Popeska, Z., **Stankov, E.** (2018). "Creating Effective Quizzes by Balancing the Distribution of Question Types". In: *Proceedings of the 12th annual International Technology, Education and Development Conference (INTED 2018)*, pp. 6631-6637.

Kostadinov, B., Jovanov, M., **Stankov, E.** (2018). "Platform for Analysing and Encouraging Student Activity on Contest and E-learning Systems". Journal *Olympiads in Informatics*, ISSN 1822-7732, vol. 12, pp. 85-98.

Dimitrievska Ristovska, V., **Stankov, E.**, Sekuloski, P. (2021). "Teaching and Examination Process of Some University Courses before vs during the Corona Crisis". Journal *Olympiads in Informatics*, ISSN 1822-7732, vol. 15, pp. 91-104.

Jovanov, M., Mihova, M., Kostadinov, B., **Stankov, E.** (2018). "New Approach for Comparison of Countries' Achievements in Science Olympiads". Journal *Olympiads in Informatics*, ISSN 1822-7732, vol. 12, pp. 53-68

Mihova, M., Jovanov, M., **Stankov, E.** (2015). "On the Role of Challenging Math Problems in the Discrete Mathematics Courses". In: *Proceedings of the 2015 IEEE Global Engineering Education Conference (EDUCON 2015)*, pp. 730-736.

Mihova, M., Stojkovikj, N., Jovanov, M., **Stankov, E.** (2014). "On Maximal Level Minimal Path Vectors of a Two-Terminal Network". Journal *Olympiads in Informatics*, ISSN 1822-7732, vol. 8, pp. 133-144.

Mihova, M., Stojkovikj, N., Jovanov, M., **Stankov, E.** (2016). "Maximal Level Minimal Path Vectors of a Two-Terminal Undirected Network". *Journal IEEE Transactions on Reliability, JCR i.f. 3,177*, ISSN 0018-9529, vol. 65, no. 1, pp. 282-290.

1.6. Организација на докторската дисертација

Во **Глава 2** е даден преглед на релевантната литература од областа на истражување на докторската дисертација. Овде прво се прави осврт на

проверката на знаење, со посебен акцент на проверката на знаење преку електронско тестирање, а особено онаа кај воведните курсеви за програмирање. Следно, претставена е класификација на системите за автоматска проверка на знаење од програмирање и направен е краток преглед со анализа на системите за автоматско генерирање на прашања што содржат програмски код. Заклучокот од оваа анализа е дека ниту еден од постојните системи не обрнува внимание на сложеноста на прашањата (како и нејзината конзистентност) во процесот на генерирање. Потоа е изложен и краток преглед на литературата за мерење на сложеност на програмски кодови. Образложена е дефиницијата за софтверска сложеност на Basili (1980) и претставена е поделба на метриките за мерење на сложеност во неколку категории, според аспектот на сложеноста на кодот што го разгледуваат. Во продолжение подетално се опишани дел од најважните и најчесто користени претставници на различните категории на метрики за мерење на сложеност. Овде се дискутирани својствата на метриките, при што се идентификувани нивните предности и недостатоци. Кон крајот од оваа глава направена е и анализа на применливоста на метриките за мерење на сложеност во едукативен контекст, односно за проверка на знаење кај воведните курсеви за програмирање. Заклучокот којшто е извлечен е дека ниту една од постојните метрики не може директно да се примени за овие потреби.

Во **Глава 3** прво е дефиниран новиот модел за паметно автоматско генерирање на прашања што содржат програмски код. Образложени се чекорите што ги предвидува моделот, а кои се неопходни во процесот на паметно генерирање на вакви прашања. Овој модел вклучува употреба на соодветна метрика за мерење на сложеност на програмските кодови, па во продолжение е предложена и токму таква метрика. Опишаната метрика е заснована на тежински вредности придружени на операторите/контролните наредби од програмскиот јазик и ја анализира сложеноста од гледна точка на напорот и времето што треба да го потроши кој било студент при рачно извршување на наредбите/операторите заради пресметување на излезот од кодот, па оттука е соодветна за намената за која е дефинирана. Конечно, во рамките на оваа глава е претставен и нов софтверски систем за паметно автоматско генерирање на прашања што содржат програмски код. Даден е опис на архитектурата на системот, практичната имплементација на генерирачкиот процес, а исто така детално е опишана и употребата на системот од перспектива на различните улоги на корисници. Претставениот систем овозможува генерирање на прашања што содржат програмски кодови со конзистентна сложеност (мерена користејќи ја предложената метрика), со што претставува практична реализација на дефинираниот нов модел.

Една од најактуелните и најчесто користени платформи за е-учење во денешно време, без сомнение, е системот Moodle. Оваа платформа, меѓу другото, се користи и при спроведувањето на наставниот процес на матичната

институција, Факултетот за информатички науки и компјутерско инженерство (ФИНКИ) при Универзитетот „Св. Кирил и Методиј“ во Скопје. Во **Глава 4** е претставен нов додаток за Moodle, којшто е развиен со цел да се овозможи едноставна и лесна имплементација на моделот за паметно автоматско генерирање на прашања во процесот на проверка на знаење кај воведните курсеви за програмирање на институции коишто ја користат оваа платформа, како што е ФИНКИ. Овој додаток нуди паметно автоматско генерирање на прашања (што содржат програмски код) од нов Moodle тип, именуван CodeCPP. И за овој софтверски систем, исто така, прикажан е опис на архитектурата, реализацијата на генерирачкиот процес, како и употребата од гледна точка на секоја од различните улоги на корисници.

Следејќи ги истражувачките цели на докторската дисертација, во **Глава 5** прво е претставена нова (помошна) софтверска алатка којашто помогна за подобрување на прецизноста на пресметувањето на кодната сложеност при автоматското генерирање на прашања што содржат програмски кодови. Потоа се опишани двете серии од експерименти спроведени користејќи ја оваа алатка, со цел утврдување на соодветни тежински вредности што би се придружиле на аритметичките оператори при мерењето на сложеност на кодови со предложената метрика. За секој од експериментите, анализирани се резултатите и извлечени се соодветни заклучоци. Главниот заклучок е дека постои корелација помеѓу времињата потребни за рачно извршување на аритметичките операции, што дозволува определување на односите помеѓу паровите од различни операции, а со тоа и на тежините. Уште повеќе, врз основа на резултатите од експериментите, претставени се и пресметаните односи помеѓу повеќе парови од операции. На крајот од оваа глава е изложено истражувањето коешто е изведено со цел да се евалуира степенот на успешност на автоматската проверка на знаење кај воведните курсеви за програмирање, со користење на предложениот систем за паметно автоматско генерирање на прашања што содржат програмски код, односно додатокот CodeCPP во рамки на платформата за е-учење Moodle. Во рамки на ова истражување спроведени се два експерименти, со кои е направена евалуација преку едногодишна примена на системот во настава по програмирање од една страна, и повратен одговор од избрани наставници обучени за користење на системот од друга страна. И за овие експерименти, исто така, дадена е анализа на резултатите и извлечени се соодветни заклучоци.

Конечно, во **Глава 6** се дадени заклучни согледувања и посочени се можни насоки за понатамошно истражување.

2. Преглед на релевантна литература

2.1. Проверка на знаење преку електронско тестирање

2.1.1. Проверка на знаење

Во образованието се разликуваат три различни процеси: **подучување** (или предавање/држење настава, англ. *teaching*), **учење** (најчесто се однесува на процесот кај ученикот, англ. *learning*) и **проверка на знаење** (испитување, кое вклучува задавање на прашања, задачи, проекти, заради оценка на знаењето и вештините, англ. *assessment*). Проверката на знаење во себе го вклучува **оценувањето** (т.е. доделување на некаква вредност по спроведена проверка на знаење, англ. *grading*, *marking*).

Како причини за испитување, Biggs (2003) ги наведува: избор на ученици, контролирање/мотивација на учениците, директно подучување/учење и идентификација на силните и слабите страни на подучувањето/учењето. Тој го дефинира феноменот на ехо (англ. *backwash*), според којшто учениците го учат она за кое мислат дека ќе бидат оценувани, а со тоа испитувањето повеќе одредува што и како учениците ќе учат, отколку самата наставна програма. Ваквиот феномен може да биде искористен за поефикасно и поефективно учење, преку почести проверки на знаењата кај учениците.

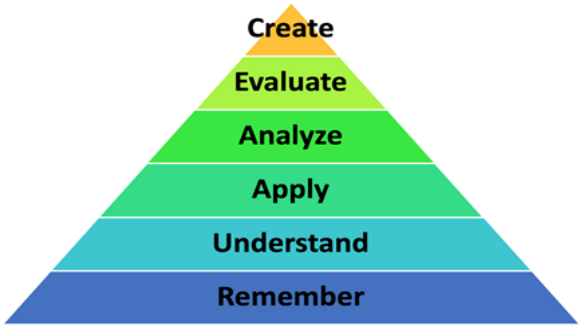
Во својот труд (Tyler, Gagne и Scriven, 1967), Scriven ги воведува термините „формативно вреднување“ и „сумативно вреднување“. **Формативното испитување** (вреднување) се користи за повратна информација кон ученикот за време на курсот, не треба (во целост) да влијае на оценката и е корисно за набљудување на процесот на учење. **Сумативното испитување** се случува на крај на курсот и се користи за оценка на учениците.

Ако немаат јасно разбирање на целите на наставната програма, учениците веројатно ќе се дезориентираат и ќе трошат време да утврдат што точно тие треба да научат. Затоа, секоја програма треба да има јасно дефинирани исходи од учењето. Исходите од учење се состојат од 3 дела: дејствување на ученикот, содржина (материјал) и стандард кој е потребен за да се достигне целта. Исходите се дефинираат на повеќе нивоа: на цела наставна програма, на конкретен модул или дури и на една наставна единица.

Исходите од учењето, методите на настава и проверката на знаење се меѓусебно зависни. Само со вистинска нивна интеграција може да се добие ефикасно учење. Во (O'Leary *et al.*, 2006) оваа интеграција се именува како конструктивно порамнување.

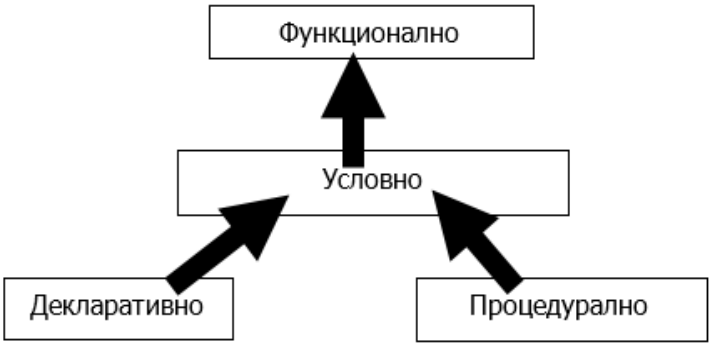
Знаењето не може да се наметне или пренесе преку директна настава – тоа се креира од учениците преку нивните сопствени активности за учење.

Во педесеттите години на дваесеттиот век создадена е таканаречената **Блумова таксономија** (Bloom *et al.*, 1956), која прави класификација на знаењето во 6 нивоа. Подоцна, во 1999 година, се прави ревизија на таксономијата и се воведуваат мали измени. Тогаш се дефинирани следните 6 нивоа на знаење, од најниско кон највисоко: памтење, разбирање, примена, анализа, вреднување и креирање (Слика 2.1).



Слика 2.1: Нивоа на знаење во Блумовата таксономија

Друг поглед го дели ниското ниво на знаење во два ентитета: декларативно знаење и процедурално знаење (Слика 2.2). Декларативно знаење е знаење на нешто, на факти – запишани во книги или кажани на предавање. Тоа се здобива од истражување, а не од сопствено искуство, и истото е проверливо и искористливо. Процедуралното е знаење засновано на вештина. Тоа претставува секвенца од дејствија и подразбира поседување на вистинските компетенции. Знаење кое ги обединува овие две е т.н. условено знаење. Тоа е знаење кога да се направи нешто, што да се направи и зошто да се направи (Gagne, 1985; Paris, Lipson и Wixson, 1983). Преку ова знаење се прави преод кон највисокото, т.н. функционално знаење. Функционалното знаење е знаење во рамки на искуството на ученикот, кој може да го употреби декларативното знаење во решавање проблеми. Ова знаење се опишува како изведба базирана на разбирање.



Слика 2.2: Хиерархија на знаењето

Постојат повеќе видови на проверка на знаење, т.е. проверката може да се спроведе преку различни методи. Во продолжение ќе бидат наведени некои од најчесто користените методи на проверка на знаење, како и нивните карактеристики.

1. **Испити** (англ. *exams*)
    i. Обично невидени од учениците пред денот на испитот;
    ii. Испитувачот избира дел од материјалот и ја тестира работата на испитаникот;
    iii. Овој метод ги поттикнува учениците да меморираат информации наместо да се обидуваат да ги разберат како дел од целиот курс.
2. **Домашни задачи** (англ. *homework assignments*)
    i. Се дава време на учениците, кое може да биде и неколку месеци;
    ii. Се дава доволно време за да се утврди теоријата предадена на час;
    iii. Овозможуваат учениците да развијат сопствен модел, преку проба и грешка;
    iv. Побаруваат учениците да работат без помош во најголем дел од времето.
3. **Лабораториски вежби** (англ. *laboratory exercises*)
    i. Овозможуваат моментална повратна информација од наставник;
    ii. Може да се набљудува процесот на учење;
    iii. Овозможуваат да се идентификуваат силните и слабите страни на ученикот.
4. **Презентации на час** (англ. *in-class presentations*)
    i. Обезбедуваат можност да се оцени и структурата и материјалот;
    ii. Критериумите за оценување треба да се јасни за учениците;
    iii. Кај овој метод, комуникациските вештини се клучни преносливи вештини.
5. **Есеи / истражувања** (англ. *essays / research papers*)
    i. Задај задачи, анализирај, вреднувај...;
    ii. Два пристапи кон оценување: аналитички и целокупен.
6. **Електронски тестови** (англ. *e-tests*)
    i. Содржат прашања со повеќе одговори;
    ii. Обезбедуваат комплетна објективност во оценувањето;

iii. Овој метод широко се користи за оценка на знаењето на фактите и разбирањето на концептите;

iv. Се користат за тестирање на големи групи;

v. Побаруваат многу време за подготовка;

vi. Не се добиваат поени за делумен одговор;

vii. Може да се стимулира погодување од страна на учениците.

7. **Учење базирано на проблем** (англ. *problem based learning*)

i. Наставен метод кој ги предизвикува учениците да научат да учат;

ii. Кооперативно работење во групи, за да се најдат решенија на реални проблеми;

iii. Се користи за да се заинтригира ученикот (да му се ангажира љубопитноста) и да се иницира учење на материјалот;

iv. Ги подготвува учениците да размислуваат критички и аналитично, и да наоѓаат и користат соодветни извори за учење.

2.1.2. Историјат на примена на компјутерите за проверка на знаење

Историјата на примената на компјутерите за проверка на знаење (англ. *computer based assessment* – CBA) започнува од 70-тите години на минатиот век. Меѓутоа, тогашната висока цена на компјутерите и нивните технички можности ја ограничиле нивната примена во тестирањето. Со текот на времето, напредокот на технологијата овозможил развој и примена на компјутерите за тестирање во многу области, вклучувајќи и во образованието (Drasgow, 2002).

Еден од најраните обиди за примена на компјутерите како помошно средство во наставниот процес и за проверка на знаење се случил во 1960-тите години, кога е започнат проектот PLATO (англ. *Programmed Logic for Automatic Teaching Operations*) на Универзитетот Илиноис, во САД. Проектот TICCIT (англ. *Time-Shared Interactive Computer-Controlled Information Television*), започнат во 1967 година, претставува уште еден пример за примена на компјутерите во образованието од овој период.

Историјата на електронската евалуација вклучува и примена на компјутерите за оценување на задачи од областа на програмирањето. Еден од првите обиди за примена на компјутер за автоматизација на процесот на проверка на задачи од програмирање бил т.н. „автоматски оценувач“ (англ. *automatic grader*). Автоматскиот оценувач (Forsythe и Wirth, 1965) не бил користен како компајлер, туку негова задача била да им помага на студентите додека учат да програмираат, а воедно и да му помага на наставникот во

следењето на големиот број на студенти коишто го посетувале курсот. Овој систем нашол своја примена и во учењето на далечина. Автоматскиот оценувач го користеле студентите на Универзитетот Стенфорд (англ. *University of Stanford*), кои го посетувале курсот за нумеричка анализа, за оценување на нивните вежби од програмирање.

Во текот на 1980-тите години, па и подоцна, сè повеќе растел интересот за примена на компјутерите во наставата, при што истите се користеле и за автоматизација на некои задачи на наставниот модел. Набргу потоа се појавуваат системи за проверка на знаењето и во други области, како што се математика, хемија, учењето странски јазици (Al-Smadi и Gütl, 2008).

Најголемо влијание врз примената на компјутерите во образованието во текот на 1990-тите години имал развојот на светската мрежа од сите мрежи – вебот (англ. *World Wide Web* – WWW). Оттогаш, системите за проверка на знаење почнуваат да стануваат засновани на вебот. Примери за некои вакви системи односно системи кои во себе вклучуваат и проверка на знаење се WebCT, Blackboard, Moodle, ATutor, Claroline и Desire2Learn (Keles и Ozel, 2016).

2.1.3. Предности и недостатоци на компјутерски помогнатите тестови

Проверката и оценувањето на знаењето им овозможува на наставниците да осознаат до кој степен студентите ги усвоиле наставните содржини и стекнале одредени знаења, како и да добијат повратна информација за својата наставна работа, за применетите наставни методи и облици на работа, за да можат да ги подобрат истите. Како што беше наведено претходно, различни методи на проверка на знаењето се применуваат во различни контексти (на пример, презентација на час, пишување на есеи, проекти, итн.). Сепак, најчесто користената „алатка“ за проверка на знаење, а којашто воедно се смета и за најобјективно средство за мерење на знаењето (Gyll и Ragland, 2018), е тестот на знаење. **Тестови на знаење** (или наставни тестови) се тестови со чија помош се одредува колкаво знаење стекнал некој човек преку одредена активност или во одреден период на учење. Тие се состојат од прашања или задачи (зададени во специјални облици) преку коишто се испитува знаењето на испитаникот од целокупната наставна содржина на еден предмет, или пак од одредени делови од неа.

Како што компјутерот сè повеќе се користел како наставно средство, така неговата употреба се проширувала низ сите нивоа на образованието, па почнал да се применува и за проверка на знаењето. Со тоа се појавуваат и тестовите кои се изработуваат на компјутер т.е. **компјутерски помогнати (КП) тестови** (англ. *computer based tests*).

Покрај достапноста на компјутерите, исклучителниот напредок на КП тестовите е предизвикан и од бројните предности кои ги имаат тие во однос на конвенционалните тестови, кои во литературата често се нарекуваат „тестови со молив и хартија“ (англ. *paper-and-pencil tests*). Најчесто спомнувана предност е брзото администрирање на тестот и оценувањето, како и можноста за добивање на повеќе информации за испитаникот.

Тестирањето со користење на компјутер овозможува подобра контрола над условите на тестирањето (Khoshsima и Hashemi Toroujeni, 2017). Тестот се задава на ист начин за сите испитаници и сите добиваат исти инструкции. Доколку истече времето предвидено за решавање на тестот, компјутерот го запира процесот на тестирање. Идентичните услови за испорака на тестот и апликацијата за тестирање за сите испитаници ги елиминира грешките при мерење кои можат да настанат заради неисполнетоста на овие услови. На ваков начин се зголемува доверливоста и валидноста на тестовите.

Како една од предностите на КП тестовите може да се наведе и зголемената флексибилност (можноста за додавање на пораки кои нудат дополнителни објаснувања, упатства и потсетници; можноста за исправка на грешки), како и поголемата мотивација и задоволство на испитаникот во споредба со оние испитаници коишто решаваат класичен тест (Matthiasdottir и Arnalds, 2016). Тестирањето со употреба на компјутери е поекономично, овозможува подобро искористување на човечките ресурси, а мултимедијалните можности на компјутерот нудат бројни предности во начинот на претставување на прашањата (аудио, видео, итн.).

Секако, КП тестовите имаат и некои недостатоци. Во прв ред, потребна е одредена компјутерска писменост на испитаникот за да може да се избегне влијанието на начинот на испорака на тестовите врз постигнатите резултати (Alderson, 2000). Друга важна особина на испитаникот која може да влијае на неговиот резултат, при користење на КП тест, е ставот кој го има тој кон компјутерот.

Некои луѓе не се запознаени со технологијата и не можат да држат чекор со нејзиниот брз развој. Оттука, тие не сакаат да се бават или да користат кој било облик на технологија, ниту пак да ја применуваат во своето академско образование или во социјалниот живот (Al-Amri, 2008). Значаен недостаток на тестирањето преку компјутер е тоа што полесно може да се погреши при читањето на текст од монитор отколку при читањето на текст напишан на хартија, а постојаното префрлање на погледот од мониторот на компјутерот на листот хартија со белешки може да го попречува испитаникот во неговата работа.

2.1.4. Проверка на знаење кај воведни курсеви за програмирање

Воведните курсеви за програмирање на високообразовните институции може да вклучуваат студенти со значително различно предзнаење. Некои од студентите може да посетувале курсеви за програмирање во средношколското образование, некои можеби се самоуки, но некои може да немаат и никакво искуство во програмирањето. Студентите со мало или никакво искуство од програмирање особено имаат потреба од повратна информација (англ. *feedback*) од наставникот, уште во раната фаза од спроведување на курсот. Од тие причини, на овие курсеви, во текот на семестарот вообичаено се задаваат еден или повеќе тестови, коишто му претходат на финалниот испит што се полага на крајот од семестарот, на којшто се бара од студентите да пишуваат програми за решавање на одредени проблеми. Доста често овие тестови се состојат од прашања со избор од повеќе можни одговори – ПМО прашања (англ. *multiple choice questions* – MCQs). Предноста кај ваквите тестови е во можноста за автоматско оценување, а со тоа и брза повратна информација. Но, од друга страна, недостаток на тестовите составени од ПМО прашања е фактот дека им овозможуваат на оние студенти кои не го знаат точниот одговор да погодуваат.

Во литературата за проверка на знаење кај курсевите за програмирање, повеќе автори предлагаат примена на Блумовата таксономија при дизајнот на испитувачката методологија (Dorodchi, Dehbozorgi и Frevert, 2017; Gluga *et al*., 2012; Lajis, Nasir и Aziz, 2018; Lister и Leaney, 2003; Thompson *et al*., 2008). Clark (2004) опишува класификација на ПМО прашања за воведни курсеви за програмирање според когнитивното ниво на знаење што го побаруваат, во согласност со Блумовата таксономија. Типовите на прашања што ги предлага тој се објаснети во продолжение.

1. **Прашања на нивото „памтење“** (англ. *knowledge questions*). Кај воведните курсеви за програмирање, почетно знаење кое е потребно е знаењето на синтаксата на програмскиот јазик, како и околината за програмирање. До крајот на курсот ова знаење се претпоставува и не се испитува посебно во финалниот испит. Ова не значи дека се омаловажува тоа знаење, бидејќи другите когнитивни вештини зависат од него. Прашањето на Слика 2.3 е пример за прашање на нивото „памтење“.

Со кој од следниве повици до функции од стандардната библиотека на програмскиот јазик C можеме да провериме дали настапил крај на дадена датотека која што е асоцирана со покажувачот `fpok` од тип `FILE*` ?

а) `fclose(fpok)` б) `ferror(fpok)` в) `fseek(fpok)` г) `feof(fpok)`

Слика 2.3: Прашање на нивото „памтење“ за воведен курс за програмирање

2. **Прашања на нивото „разбирање“** (англ. *comprehension questions*). Разбирањето мора да оди веднаш „зад петиците“ на памтењето. Тоа го консолидира знаењето и овозможува негова интернализација. Вообичаени прашања од овој тип се прашања кадешто од студентите се бара рачно да го следат извршувањето на даден програмски код за да го пресметаат излезот (најчесто вредноста на некоја програмска променлива). На Слика 2.4 е даден пример за прашање на нивото „разбирање“.

Кој е излезот од следниот програмски код?

```
#include <stdio.h>
int main()
{
    int i = 0;
    while (i < 5) {
        printf("\%d", i % 2);
        i++;
    }
    return 0;
}
```

а) 01010 в) 10101

б) 01234 г) 012012

Слика 2.4: Прашање на нивото „разбирање“ за воведен курс за програмирање

3. **Прашања на нивото „примена“** (англ. *application questions*). За кој било програмер, примената претставува способност за пишување на програми, како и, на пониско ниво на грануларност – за пишување на функција или модул кој ќе извршува некоја специфична задача. Оваа способност се стекнува преку практични задолженија (на лабораториски вежби) и најдобро може да се тестира преку стандарден испит со пишување програми. Сепак, на нивото „примена“, може да се задаваат и ПМО прашања од обликот „комплетирај го кодот“. Приказ на пример за вакво прашање е даден на Слика 2.5.

4. **Прашања на нивото „анализа“** (англ. *analysis questions*). Една од најчестите задачи при одржување на даден софтвер е да се утврди што работи некој код. Популарен тип на прашање кај финалните испити е „Што работи оваа функција?“. Една варијација на овој тип на прашање би била: „При кои услови функцијата работи коректно?“. Овие прашања побаруваат анализа на кодот. ПМО прашањата кои прават проверка на знаење на нивото на анализа не се воопшто едноставни за составување, но со малку работа некои од прашањата од нивото „разбирање“ може да се претворат во прашања од нивото „анализа“. Пример за прашање „од анализа“ е даден на Слика 2.6.

Низата `niza` содржи `n` елементи. Дадениот код треба да изврши циклично поместување на елементите на `niza` за едно место во лево. На пример, од почетната низа

7	3	8	1	0	5

треба да се добие низата

3	8	1	0	5	7

```
naredba1
for (i = 0; i < n - 1; i++)
    niza[i] = niza[i + 1];
naredba2
```

За кодот да работи точно, наредбите `naredba1` и `naredba2` треба да бидат:

а)
```
temp = niza[0];
niza[0] = temp;
```

б)
```
temp = niza[0];
niza[n-1] = temp;
```

в)
```
temp = niza[n-1];
niza[0] = temp;
```

г)
```
temp = niza[n-1];
niza[n-1] = temp;
```

Слика 2.5: Прашање на нивото „примена“ за воведен курс за програмирање (Clark, 2004)

Под претпоставка дека е точен условот: `x >= 0`, кој од следните искази ќе биде **неточен** откако ќе се изврши кодот?

```
y = 1;
do
    y = y * 2;
while (y <= x);
```

а) y може да биде непарен

б) y може да биде еднаков на x

в) y може да биде еднаков на $x + 1$

г) y мора да биде степен на 2

Слика 2.6: Прашање на нивото „анализа“ за воведен курс за програмирање (Clark, 2004)

Во оваа докторска дисертација фокусот е на прашања на нивото „разбирање“. За да се избегне можноста студентите да погодуваат, наместо да се користат ПМО прашања, едноставно решение е да се побара од студентите да го напишат точниот одговор (откако ќе го пресметаат). Токму затоа, во дисертацијата ќе бидат разгледувани прашања со краток одговор на нивото „разбирање“.

2.2. Системи за автоматска проверка на знаење од програмирање

2.2.1. Класификација на системите за автоматска проверка на знаење од програмирање

Во литературата се опишани поголем број на системи коишто овозможуваат автоматска евалуација на знаење од програмирање. Постојат повеќе начини да се класифицираат постојните системи во оваа област. Најзначајната карактеристика којашто ги раздвојува овие системи е секако типот на знаење коешто се обидуваат да го оценат. Генерално, се разликуваат три типа на знаење кои се очекува да ги стекне кој било студент на воведен курс за програмирање: 1) знаење на синтаксата на програмскиот јазик (како да се пишуваат валидни програми); 2) знаење на семантиката (како функционираат различните програмски конструкции); 3) прагматично знаење (како да се решаваат проблеми преку пишување на соодветни програми). Сите три типа на знаење типично се евалуираат користејќи разни тестови и задачи.

Во текот на изминатите триесетина години, истражувачите во областа на автоматската проверка на знаење од програмирање главно се фокусираат на знаењето на семантиката и прагматичното знаење. Ова е така бидејќи самостојното познавање само на синтаксата се смета за многу помалку важно во денешно време, во споредба со раните денови на едукацијата од областа на компјутерските науки. Оттука, доминантни се два типа на системи за автоматско оценување на знаење:

1. Системи кои ја оценуваат *способноста на студентите за рачна евалуација (рачно извршување) на готови програми*, и
2. Системи кои ја оценуваат *способноста за пишување на сопствени програми*.

Системите засновани на рачна евалуација се обидуваат да го тестираат знаењето на семантиката што го поседуваат студентите. Тие побаруваат од студентите рачно да го следат (евалуираат) извршувањето на дадена (готова) програма, за потоа да ги оценат добиените резултати од оваа рачна евалуација. Системите засновани на пишување програми се обидуваат да го тестираат прагматичното знаење на студентите. Тие побаруваат од студентите да напишат програма за решавање на некој проблем, а потоа ја оценуваат продуцираната програма користејќи соодветна техника.

2.2.2. Системи засновани на пишување на програми

Со оглед на тоа што способноста за решавање на програмски (алгоритамски) проблеми и продукција на функционален програмски код се

смета за најважна (во областа на компјутерските науки и компјутерското инженерство), најголем дел од истражувањата во врска со автоматската евалуација на знаење од програмирање се посветени на системите што ја оценуваат оваа способност.

Развиени се поголем број на системи за оценување на студентски програми во текот на изминатите триесетина години. Од гледна точка на оценувањето на знаење, овие системи се разликуваат во два аспекти: количината на код што се побарува да ја напише студентот и нивото на анализа на кодот. **Системите ориентирани кон прашања** (англ. *question-oriented systems*) типично побаруваат од студентот да надополни некоја постојна програма со релативно мал број на линии код (неколку наредби) (Arnow и Barshay, 1999; Garner, 2002). **Системите ориентирани кон задачи** (англ. *assignment-oriented systems*) оценуваат комплетни програмски задачи, при што програмите за решавање на овие задачи треба да се напишат почнувајќи „од нула“ (односно без да е зададен каков било почетен код) или пак почнувајќи од мал „скелет“ (односно со зададен почетен код) (Benford *et al*., 1994; Daly, 1999; Gotel и Scharff, 2007; Higgins *et al*., 2003; Jackson и Usher, 1997; Joy, Griffiths и Boyatt, 2005; Maggiolo и Mascellani, 2012; Reek, 1989; Saikkonen, Malmi и Korhonen, 2001). Во нашата институција развиени се два системи за автоматско оценување ориентирани кон задачи, кои тековно се користат во некои од курсевите: МЕНДО (Jovanov, Kostadinov и Stankov, 2010) и E-Lab (Delev и Gjorgjevikj, 2012). Иако технологијата зад овие системи често пати е слична, системите ориентирани кон прашања претежно ја оценуваат способноста на студентите да користат програмски конструкции во одреден контекст, додека системите ориентирани кон задачи се фокусираат на оценување на способноста за решавање на програмски (алгоритамски) проблеми. Овие вештини припаѓаат на различни нивоа од Блумовата таксономија на едукативни цели (Bloom *et al*., 1956).

Пишувањето на компјутерска програма е релативно тешка когнитивна вештина за совладување, а исто така доста е тешко да се измери до кој степен некој човек ја поседува оваа вештина (Lister *et al*., 2010; Spiro и Jehng, 1990). И покрај големите напори на голем број наставници, многу студенти сеуште наидуваат на потешкотии при учењето на програмирањето. За да можат да напишат една едноставна програма, тие мора да поседуваат основно знаење од променливи, влез/излез на податоци, контролните структури, па и други области. Дури и да го поседуваат теоретското знаење коешто е неопходно, студентите се соочуваат со проблем кога ќе треба да го применат своето знаење како целина и всушност да напишат функционален програмски код (Maravić Čisar *et al*., 2011).

Повеќе истражувачи предлагаат користење на прашања кои побаруваат рачно следење на извршувањето на даден програмски код (англ. *code tracing*) за оценување на програмерските вештини на студентите (Lister *et al*., 2004; Xie *et al*., 2019). Спроведените истражувања потврдуваат дека студентите не можат да научат да пишуваат програмски кодови без претходно да научат да го следат извршувањето на кодовите, како предусловна вештина (Cunningham *et al*., 2017; Izu *et al*., 2019; Kumar, 2015; Lopez *et al*., 2008; McCracken *et al*., 2001; Venables, Tan и Lister, 2009). Фокусот на оваа докторска дисертација е токму на проверката на знаење преку автоматско генерирање на прашања што содржат програмски код за којшто се побарува рачна евалуација од страна на студентите, па во следното поглавје е дадена подетална анализа и преглед на постојните системи од овој тип.

2.2.3. Системи засновани на рачна евалуација на готови програми / кодни фрагменти

Системите засновани на рачна евалуација (следење на извршувањето) на програмите во минатото ја претставуваа помалку популарната класа на системи за проверка на знаење од програмирање. Сепак, во последниве десетина години интересот за овие системи станува сè поголем. Во овој период, едукативните истражувачи во областа на компјутерските науки предлагаат мноштво од разновидни пристапи како поддршка на учењето програмирање, кои одат понатаму од вообичаените програмски вежби (со задачи во кои се бара пишување на соодветни програми) на седмично ниво. Рачната евалуација на програми е еден таков пристап, за којшто истражувачите сугерираат дека е ефективен (Lister, Fidge и Teague, 2009; Lopez *et al*., 2008; Kumar, 2015). Како пример, Kumar (2015) спровел контролирано истражување коешто покажало дека рачното следење на извршувањето на програмски кодови (англ. *code tracing*) ја подобрува способноста на студентите за разбирање и пишување на програми.

Систематското задавање на повеќе множества од прашања во кои се побарува рачна евалуација на програмски кодови на студентите, со градација на тежината и можност за повторно решавање на слични (или и исти) прашања, е исто така и во согласност со теоријата на учењето (Anders Ericsson *et al*., 2018). Според Thomas (Thomas *et al*., 2019), еден идеален систем за поддршка на учењето на програмирање, покрај редовните програмски вежби, треба да овозможува и генерирање на голем број на прашања со рачна евалуација на кодови од соодветните теми од програмирањето кои се предмет на изучување. Ваквиот систем треба да му овозможува на наставникот да нагодува одредени општи параметри (број на прашања, ниво на сложеност на секое прашање, итн.), треба да е ефикасен во генерирачкиот процес за да обезбеди скалирање на потенцијално голем број на студенти, и треба да е лесно достапен за студентите

(без разлика каде и да се наоѓаат) за да ги охрабрува постојано да вежбаат решавајќи што е можно повеќе прашања.

Од друга страна, прашањата кои побаруваат рачна евалуација на програмски кодови се погодни и за оценување на знаењето од семантиката што го стекнале студентите. Иако овие типови на прашања тешко можат да ги оценат способностите како што се решавањето на проблеми, алгоритамското размислување или длабокото логичко резонирање, сепак тие можат да дадат добар увид во разбирањето на основните програмски конструкции на употребениот програмски јазик, како и во разбирањето на некои основни концепти на програмирањето воопшто (Shuhidan, Hamilton и D'Souza, 2010; Simkin и Kuechler, 2005). Според Блумовата таксономија на едукативни цели, разбирањето е етапа која претходи на примената, па според тоа добро е да се испита способноста за разбирање на студентите, пред да се побара од нив да го применат своето знаење. Но, за да се постигне објективна и фер проверка на знаење на конкретен испит од некој курс, на сите студенти коишто го полагаат испитот мора да им се поставаат прашања со иста или многу слична комплексност, односно прашања кои побаруваат исто ниво на знаење и вложување на (приближно) ист напор за да се даде точен одговор. Кога станува збор за воведните курсеви за програмирање и за употребата на споменатите типови прашања со рачна евалуација на кодови, ова го наметнува проблемот на одржување на конзистентност на комплексноста од страна на наставниците при креирањето на прашања што содржат кодни фрагменти за студентските тестови, особено кога се работи со големи групи студенти. При подготовка на повеќе верзии од еден ист тест (заради проверка на знаење на голема група од студенти), наставниците треба да се свесни за сложеноста на програмските кодови и треба секогаш да се обидуваат да составуваат прашања што содржат кодни фрагменти со иста или слична сложеност. Соодветно решение на овој проблем е да се користи автоматско генерирање на прашања.

Од погорната дискусија може да се заклучи дека користењето на системи за автоматско генерирање на прашања базирани на рачна евалуација на програмски кодови е соодветно во процесот на учење на програмирањето од една страна, но и во процесот на проверка на знаење кај курсевите за програмирање – од друга страна. Еден начин да се генерираат голем број на прашања е со користење на т.н. **шаблони** (*шаблонски прашања*, англ. *question templates*), кои дозволуваат наставникот да специфицира кои делови од прашањето ќе бидат случајно генерирани или избрани, што потоа овозможува автоматско оценување (Brusilovsky и Sosnovsky, 2005; Dancik и Kumar, 2003; Fernandes и Kumar, 2004; Hoffman, Lu и Pelton, 2011; Hsiao, Brusilovsky и Sosnovsky, 2008; Hsiao, Sosnovsky и Brusilovsky, 2010; Kumar, 2005; Prados *et al*., 2005; Shah и Kumar, 2002; Singhal и Kumar, 2000; Traynor и Paul Gibson, 2005). Според тоа, секој шаблон може да се инстанцира со случајно избрани параметри

за да се креира ново прашање. Како последица, секој шаблон, во зависност од бројот на параметри, може да генерира голем (па и огромен) број на различни прашања. Оттука, прашањата засновани на шаблони се справуваат со проблемите како што се плагијаризмот, големото време потребно на наставниците за креирање на голем број на различни прашања, како и времето што се троши на оценување во рамките на процесот на проверка на знаење. Од друга страна, студентите можат да ги користат овие шаблони за да го генерираат истото прашање, но со различни параметри. Ова ќе им овозможи истото прашање да го решаваат повеќе пати, односно сè дури не го совладаат концептот што е предмет на интерес на прашањето.

Во продолжение следува краток преглед на постојните системи за генерирање на прашања што содржат програмски код што се опишани во литературата. Повеќето од нив се засновани токму на опишаниот пристап со користење на шаблони. Овде треба да се истакне значењето на истражувачката работа на Amruth Kumar, кој има развиено серија од системи за подучување (тутори) за програмирање, заедно со уште неколку други истражувачи. Иако овој автор има предложено уште неколку други тутори наменети да им помогнат на студентите во учењето програмирање преку решавање проблеми, фокусот на овој преглед е само на оние што овозможуваат генерирање прашања базирани на рачна евалуација на програмски кодови.

Првиот од туторите на Kumar од оваа категорија е воведен во трудот (Singhal и Kumar, 2000). Овој систем генерира ПМО прашања што содржат C/C++ програмски кодови, за темата „вгнездени наредби за избор“, базирани на еден шаблон. Шаблонот што се користи за генерирање е генеричка C/C++ наредба `if`, чијшто услов односно `if`/`else` блокови се инстанцираат на случаен начин. Студентите можат да го користат овој систем за самооценување на своето знаење преку постојано генерирање на нови и нови (различни) прашања, следење на извршувањето на `if` наредбите со различни нивоа на вгнездување, и проверка на точноста на своите одговори, сè додека не ја совладаат темата.

Фокусирајќи се специфично на доменот на јамки контролирани од бројачи (англ. *counter-controlled loops*), Dancik и Kumar (2003) развиваат систем за генерирање на C++ кодни фрагменти и прашања, заснован на шаблони. Системот е наменет да се користи како тутор за учење и тестирање на концепти од овој домен на воведното програмирање. Авторите ја евалуирале ефикасноста на системот (преку дизајнирање на соодветен тест), а добиените резултати потврдуваат дека користењето на системот значително го подобрува учењето на концептите од страна на студентите.

Уште еден туторски систем базиран на шаблони, фокусиран точно на една област од програмирањето, е претставен од страна на Shah и Kumar (2002). Овој систем генерира целосни програми засновани на специфицирани шаблони,

како и соодветни прашања на тема „различни механизми за пренесување параметри“. Слично како и двата претходно споменати тутори, и овој тутор им помага на студентите при учењето на специфичните програмерски концепти преку овозможување на повторено решавање прашања и обезбедување на повратни информации за нивните предадени решенија. Повторно, резултатите од евалуацијата на ефективноста на системот во обезбедувањето поддршка за учењето на студентите се доста позитивни.

Traynor и Paul Gibson (2005) опишуваат модел на систем за автоматско генерирање на ПМО прашања со добар квалитет, наменет за воведни курсеви за програмирање. Во процесот на креирање на прашањата, тие користат два различни пристапи за автоматско генерирање на кодни фрагменти, и за секој таков генериран коден фрагмент тие потоа обезбедуваат ограничен број на „интелигентно одбрани“ можни одговори (излези) за да го комплетираат новоформираното прашање. Емпириската евалуација на овој систем (Traynor, Bergin и Paul Gibson, 2006) го потврдува потенцијалот на овој пристап за проверка на знаењето од областа на програмирањето што го поседуваат студентите. Врз основа на субјективната евалуација, спроведена преку анкетирање на студентите коишто биле тестирани и на прашања генерирани од страна на системот, но и на „класични“ задачи со решавање на проблеми преку пишување на програми, изведен е заклучок дека ПМО прашањата со програмски кодови „обезбедуваат попрецизна индикација за способноста за програмирање на студентите“. Имено, кај „класичните“ тестови што најчесто се користат кај воведните курсеви за програмирање, при оценувањето на студентските решенија наставниците вообичаено доделуваат делумни поени само за вложениот труд (кои некогаш можат да бидат доволни и за положување на испитот од курсот), што ги поттикнува послабите студенти да учат кодови „на памет“, односно да стекнуваат „кревко“ знаење (англ. *fragile knowledge*).

Brusilovsky и Sosnovsky (2005) развиваат систем, именуван QuizPACK, наменет за самооценување во онлајн услови – надвор од училница, за потребите на студентите на воведните курсеви за програмирање. Во нивното истражување, студентите одговарале на прашања генерирани врз основа на шаблони, што резултирало со детекција на значително подобрување на студентското разбирање на семантиката на програмите. Варијабилноста на генерираните прашања од страна на QuizPACK се постигнува со користење на рандомизирани (случајно генерирани) параметри во рамките на високо структурирани кодни фрагменти (Brusilovsky и Sosnovsky, 2005).

Prados (Prados *et al*., 2005) го преискористува својот претходно развиен систем, менувајќи го програмскиот јазик на генерираните програмски кодови.

Работејќи во доменот на делокруг (областа на живот) на програмските променливи, Fernandes и Kumar (2004) откриваат подобрување на учењето при

користењето на пар од системи: генератор на прашања и систем за повратни информации. Сличен ефект од користењето на ваков системски пар е елаборирaн и од страна на Kumar (2005). Тој опишува систем за генерирање на прашања којшто воедно е способен да обезбедува повратни информации на различни концептуални нивоа.

Hoffman, Lu и Pelton (2011) го претставуваат C-doku: веб-базиран систем кој поддржува евалуација и подобрување на вештините за читање програмски кодови. C-doku генерира четири различни типови на прашања, базирани на шаблони зададени од наставници, и обезбедува пакувања на прашањата во квизови за потребите на едукативното оценување. Системот ги прикажува програмите како HTML веб-форми, каде што неколку влезови или излезот од кодот се заменети со (празни) текстуални полиња. За оценување на знаењето од програмирање, од студентот се бара да го следи извршувањето на кодот и да ги пополни празните полиња со потребните вредности. Решението на студентот потоа автоматски се проверува и се даваат соодветни повратни информации (само во случај ако системот се користи во режим за вежбање, а не за оценување!). C-doku поддржува четири различни програмски јазици: C, C++, Java и Python.

Развојот на технологијата во овој период овозможува повеќе истражувачи да се одлучат за изградба на системи дизајнирани за работа на мобилни уреди. Еден пример за ваков систем е оној на Deb, Fuad и Kanan (2017). Овој систем овозможува оценување на способноста за рачна евалуација на програми од страна на студентите, но и давање на повратни информации за секое прашање. Во поскоро време, Ericson, Margulieux и Rick (2017) заклучуваат дека автоматското генерирање на прашања што содржат програмски кодови, со истовремено давање на повратни информации за студентите, обезбедува значајна редукција на потрошеното време од страна на наставниците на подготовка на соодветните ресурси. Воедно, тие посочуваат дека ваквите системи може да се користат за учење на програмирање (преку рачна евалуација на програмите кои се составен дел на прашањата), при што резултатите што се постигнуваат се еквивалентни на оние од активностите што се фокусираат на пишување на код.

Наспроти малата популарност кај воведните курсеви за програмирање, системите засновани на рачна евалуација на готови програмски кодови се доста популарен начин на оценување на студентското знаење кај курсевите од областа на алгоритми и податочни структури. Примери за системи развиени за оваа намена се PILOT (Bridgeman *et al.*, 2000), TRAKLA (Korhonen и Malmi, 2000), TRAKLA2 (Malmi *et al.*, 2005) и MA&DA (Krebs *et al.*, 2005). Овие системи побаруваат од студентот ментално да изврши даден познат алгоритам,

почнувајќи од одредена специфична состојба, и да ги предаде финалните резултати од извршувањето.

Hsiao, Brusilovsky и Sosnovsky (2008) детално го опишуваат QuizJET – систем за генерирање на прашања од објектно-ориентирано програмирање (ООП). Системот QuizJET се фокусира на мало подмножество од клучни концепти од ООП, а прашањата ги генерира врз основа на околу 100 рачно создадени шаблони за генерирање. QuizJET демонстрирал дека активната употреба на систем базиран на прашања засновани на шаблони доведува до подобри резултати на финалниот испит од соодветниот курс за ООП.

Во поново време се појавуваат и некои системи кои го напуштаат трендот на генерирањето на прашања врз основа на шаблони. На пример, системот предложен од Zavala и Mendoza (2018) користи онтолошки елементи за да обезбеди рандомизирани прашања. Од друга страна, Thomas (Thomas *et al.*, 2019) користи стохастички алгоритам (заснован на дрва) за автоматско генерирање на ПМО прашања од неколку теми на воведните курсеви за програмирање (доделување, наредби за разгранување, наредби за итерација и низи).

Не е потребно да се направи длабока анализа за да се заклучи дека ниту еден од постојните системи кои нудат автоматско генерирање на прашања што содржат програмски кодови (за кои се побарува рачна евалуација од страна на студентите) не посветува внимание на проблемот на сложеноста на кодовите и нејзината конзистентност во текот на генерирачкиот процес. Главната мотивација и водечката насока на истражување на оваа докторска дисертација е дефинирање на модел за паметно автоматско генерирање на прашања што содржат програмски кодови кој ќе ги реши овие проблеми, како и развивање на соодветен систем кој ќе претставува практична реализација на овој модел и ќе овозможи значително подобрување на квалитетот на (само)проверката на знаење кај воведните курсеви за програмирање.

2.3. Мерење на софтверска сложеност

Истражувањето кое е во фокусот на оваа докторска дисертација се однесува на автоматска продукција на повеќе верзии од прашања за електронско тестирање, при што се внимава на сложеноста на кодот кај продуцираните верзии. За таа цел овде се прави преглед на литературата поврзана со мерење на софтверската сложеност.

Basili (1980) ја дефинира софтверската сложеност како мерка за ресурсите коишто ги користи даден систем, додека истиот е во интеракција со парче софтвер, за да изврши одредена задача. Ако системот што стапува во интеракција со софтверот е компјутер и задачата е извршување на дадена програма, сложеноста може да се дефинира како потребните меморија и време

за да се извршат пресметките во рамките на програмата. Ако системот што стапува во интеракцијата е инженер и задачата е разбирање, одржување или дебагирање на софтвер, сложеноста може да се дефинира како тешкотијата на извршување на овие задачи во даден временски период. Ако пак системот што стапува во интеракцијата е студент и задачата е рачно извршување на дадена програма, тогаш сложеноста може да се дефинира како напорот што е потребен за рачно да се извршат сите операции/наредби и да се пресмета излезот од програмата.

Во литературата предложени се голем број на разновидни софтверски метрики за мерење на сложеноста на даден програмски код. Софтверската сложеност е проучувана од неколку различни перспективи, при што секоја се фокусира на точно еден аспект од сложеноста на кодот. Некои од најчесто споменуваните перспективи кон софтверската сложеност се следниве:

1. **Пресметковна сложеност** (англ. *computational complexity*) – оваа перспектива примарно се однесува на алгоритамската сложеност на софтверот.
2. **Големина на софтверот** или **симболичка сложеност** (англ. *software size* или *symbolic complexity*) – фокусот кај оваа перспектива е на броење на некои основни градбени елементи, како што се кодни линии или операнди.
3. **Цикломатска сложеност** (англ. *cyclomatic complexity*) – се фокусира на сложеноста на контролниот тек на софтверот.
4. **Сложеност на податочниот/информацискиот тек** (англ. *data/information flow complexity*) – ја анализира сложеноста на влезовите, излезите, како и податочните меѓузависности во рамките на софтверот (или неговите делови).
5. **Психолошка и когнитивна сложеност** (англ. *psychological and cognitive complexity*) – фокусот кај оваа перспектива е на аспектот на сложеноста кој се однесува на човечкото разбирање.

Во ова поглавје ќе биде направен преглед на некои од најважните и најчесто употребувани метрики за софтверска сложеност. Ќе бидат дискутирани предностите и недостатоците од користењето на секоја од метриките, и ќе биде направена анализа и споредба на нивните особини.

2.3.1. Кодни линии

Најстарата, наједноставната, но и истовремено – една од најраспространетите метрики за мерење на сложеноста на даден програмски код е **кодни линии** (англ. *Lines of Code* – LOC). Како што сугерира и нејзиното име, оваа метрика се пресметува преку едноставно броење на линиите во

програмскиот код, сметајќи дека секоја наредба е во посебна линија. Според тоа, метриката ја мери физичката големина т.е. симболичката сложеност на кодот.

Дефинирани се две варијации на LOC. Првата од нив е **некоментирани кодни линии** (англ. *Non-Commented Lines of Code* – NCLOC), и таа ги брои кодните линии исклучувајќи ги линиите што содржат само коментари, како и празните линии. Втората варијација е **извршливи кодни линии** (англ. *Executable Lines of Code* – ELOC), и таа ги брои само оние линии коишто содржат кодни токени. Овде, заградите (на пример, '{' и '}'), коментарите и декларациите, не се сметаат за кодни токени.

Метриката LOC е мошне лесна за разбирање и пресметување, а може да се користи за кој било програмски јазик. Исто така, таа се има покажано како корисен предвидувач на напорот потребен за пишување на дадена програма (Basili, 1980). Многу емпириски студии ја покажале корисноста на LOC во различни активности од развојот на софтверот (мониторинг на развојниот прогрес, планирање, предвидување, итн.), а исто така метриката била користена и како основа за емпириска евалуација на многу други метрики. Сепак, еден сериозен недостаток на LOC е тоа што истата целосно ја игнорира сложеноста на секоја линија. На пример, наредба за едноставно доделување на вредност на променлива, како што е "`x = 5`", се брои исто како и наредба која побарува изведување на неколку аритметички операции за да се пресмета резултатот, како што е наредбата "`x = ((a + b) * c) / ((b - a) * d)`" (под претпоставка дека `a`, `b`, `c` и `d` се целобројни променливи на коишто претходно им се доделени конкретни вредности). Но, очигледно, втората наредба е доста посложена во однос на првата, особено ако истата треба да се евалуира рачно (од страна на човек). Уште еден сериозен недостаток на користењето на LOC како мерка за сложеност на програмските кодови е тоа што таа исто така ја занемарува структурата на програмата. Како илустрација, коден сегмент од 30 линии којшто не содржи разгранувања или скокови ќе биде евалуиран како посложен од страна на LOC отколку коден сегмент од 20 линии што содржи неколку разгранувања/скокови – нешто што, се разбира, не е разумно. Уште повеќе, LOC не може да се користи за споредба на програми напишани во различни програмски јазици.

2.3.2. Сложеност на Halstead

Метриките за сложеност на Halstead (Halstead, 1977) се меѓу најстарите мерки за сложеност на програмските кодови. Воведени се во 1977 година од страна на Maurice Howard Halstead, како обид да се даде квантитативна оценка за напорот што го вложува програмерот при пишување на изворниот код на дадена програма. Целта на истражувањето на Halstead во тоа време била да се

идентификуваат мерливите карактеристики на софтверот, како и да се утврдат релациите помеѓу нив.

Halstead го интерпретира изворниот код на дадена програма како секвенца од токени, и притоа секој токен тој го класифицира како **оператор** (традиционален оператор како што е ‘`+`’, ‘`*`’, или ‘`>`’, сепаратор на наредби како што е ‘`;`’, или пак клучен збор како што е “`if`”, “`return`” или “`continue`”) или **операнд** (литерален израз, константа или променлива). Четирите основни метрики дефинирани од страна на Halstead се следниве:

- n_1 – *број на уникатни (различни) оператори* во програмата
- n_2 – *број на уникатни (различни) операнди* во програмата
- N_1 – *вкупен број на оператори* во програмата
- N_2 – *вкупен број на операнди* во програмата

Сите основни метрики се пресметуваат преку утврдување на фреквенциите (честотите) на појавување на секој оператор и секој операнд во изворниот код на програмата. Преостанатите метрики на Halstead се изведени од нив како што е објаснето во продолжение:

- *Речник на програмата* (n) – се дефинира како збир од бројот на различни оператори и бројот на различни операнди, т.е.

$$n = n_1 + n_2$$

- *Должина на програмата* (N) – се дефинира како збир од вкупниот број на оператори и вкупниот број на операнди, т.е.

$$N = N_1 + N_2$$

  Оваа вредност може да се пресмета само кога програмата е комплетирана, па затоа Halstead вовел и оценувач за должината на програмата.

- *Оценета должина на програмата* ($\hat{N}$) – оваа метрика може да се користи за мерење на релацијата помеѓу должината (N) и речникот (n) на програмата. Се дефинира со следнава равенка:

$$\hat{N} = n_1 \cdot \log_2(n_1) + n_2 \cdot \log_2(n_2)$$

  Метриката претставува корисен оценувач за должината на програмата, бидејќи n_1 и n_2 може да се определат веќе на почетокот од фазата на кодирање. Една од хипотезите на Halstead е дека колку вистинската вредност на должината е поблиска до оценетата вредност, толку е

подобар и квалитетот на програмата. Исто така, тој тврди дека должината на добро структурирана програма зависи само од бројот на различни оператори (n_1) и операнди (n_2).

- *Волумен на програмата* (V) – оваа метрика ја опишува големината на имплементацијата на даден алгоритам, изразена во математички битови. Поинаку кажано, волуменот на програмата е вистинската големина што ја зафаќа програмата во компјутерот ако се користи униформно бинарно енкодирање за програмскиот речник (Shen, Conte и Dunsmore, 1983). Може да се пресмета преку равенката

$$V = N \cdot \log_2(n)$$

Halstead го интерпретирал волуменот на програмата како број на ментални споредби што се потребни за да се напише програма со должина N. Тој ја дефинирал оваа метрика врз основа на претпоставката дека луѓето го користат алгоритмот за бинарно пребарување за да го изберат следниот токен од програмски речник што содржи n симболи.

- *Тежина* (D) – позната уште и како *склоност кон грешки*. Според Halstead, нивото на тежина на дадена програма е пропорционално со бројот на различни оператори, како и со количникот помеѓу вкупниот број на операнди и бројот на различни операнди, т.е.

$$D = \frac{n_1}{2} \cdot \frac{N_2}{n_2}$$

Ова значи дека ако истиот(те) операнд(и) го(и) искористиме повеќе пати во нашата програма, таа ќе биде посклона кон грешки. Оваа метрика е заснована врз едноставната теорија на когнитивна сложеност која вели дека повторувањето на операнди и воведувањето на нови оператори во програмата ја зголемува тешкотијата на разбирањето на програмата (S. Misra и Bhavsar, 2003).

- *Ниво на програмата* (L) – се дефинира како реципрочна вредност од нивото на тежина на програмата, т.е.

$$L = \frac{1}{D}$$

Ова значи дека програма со високо ниво е помалку склона кон грешки отколку програма со ниско ниво.

- *Напор* (E) – оваа метрика се однесува на менталниот напор којшто е потребен за да се имплементира или разбере некоја програма, мерен во елементарни ментални дискриминации. За секоја ментална споредба (во волуменот V), во зависност од тежината на програмата (D), човечкиот

мозок треба да изведе повеќе елементарни ментални дискриминации. Според тоа, напорот е пропорционален со нивото на тежина и со волуменот на програмата:

$$E = V \cdot D = \frac{n_1 \cdot N_2 \cdot N}{2 \cdot n_2} \cdot \log_2(n)$$

- *Време* (T) – ова е една од поконтроверзните метрики предложени од Halstead. Оваа метрика се однесува на времето поминато во кодирање т.е. времето коешто е потребно за да се имплементира или разбере дадена програма, изразено во секунди. Како што би се очекувало, ова време е пропорционално со напорот што е потребен за да се напише програмата. Halstead експериментално утврдил дека добра апроксимација за времето може да се добие кога напорот ќе се подели со 18 ($T = \frac{E}{18}$), но може да се спроведат дополнителни експерименти за калибрирање на оваа мерка.

Некои од предностите на метриките за сложеност на Halstead се фактите дека тие се едноставни за пресметување, применливи за кој било програмски јазик, како и дека не побаруваат длабока анализа на програмската структура. Овие метрики го мерат целокупниот квалитет на програмите и можат да го предвидат потребниот напор за нивното одржување (Curtis *et al.*, 1979). Многу истражувања ја поддржуваат примената на метриките на Halstead за предвидување на програмерскиот напор, како и на бројот на програмерски грешки (Basili и Phillips, 1981; H. Zhang, X. Zhang и Gu, 2007). Сепак, овие метрики воопшто не ја разгледуваат сложеноста на контролниот тек. Како што посочуваат Yu и Zhou (2010), при пресметување на метриките за дадена програма, операндите и операторите, со или без разгранувања или скокови, се бројат без никаква разлика. Но, јасно е дека присуството на разгранувања/скокови во програмата несомнено ја зголемува нејзината сложеност.

Од практична гледна точка, не постои општоприфатен договор околу тоа како ќе се бројат операторите и операндите. На пример, не е јасно дали операторите коишто се појавуваат во пар (како што се заградите: ‘{’ и ‘}’, ‘(’ и ‘)’, “`begin`” и “`end`”, итн.) треба да се бројат како еден или два. Шемата на броење е зависна од програмскиот јазик, па оттука и метриките на Halstead се многу осетливи на јазикот (Elshoff, 1978). Според тоа, тие не можат да се применат за споредба на програми напишани во различни програмски јазици.

Со цел да се надминат погоре наведените недостатоци, во литературата се предложени и некои варијации на метриките за сложеност на Halstead (Fei, Zhi и Chao, 2004). Сепак, како што е изложено од страна на Yu и Zhou (2010), овие варијации воведуваат повеќе проблеми отколку што разрешуваат.

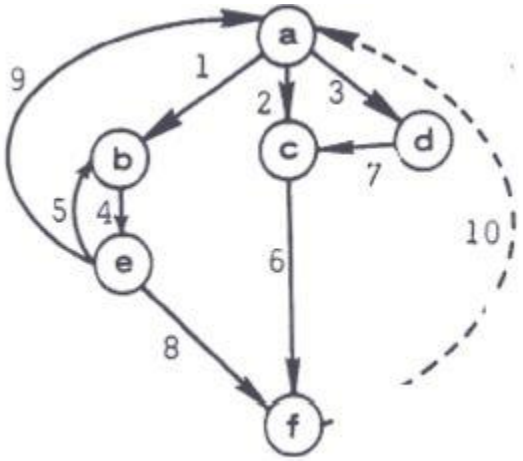
2.3.3. Цикломатска сложеност (сложеност на McCabe)

Сложеноста на McCabe (McCabe, 1976), позната уште и како **цикломатска сложеност**, е една од најшироко прифатените софтверски метрики, и без сомнение – најшироко употребуваната статичка софтверска метрика. Развиена е во 1976 година од страна на Thomas J. McCabe. Оваа метрика го мери бројот на линеарно независни патишта на извршување (извршни патишта) низ изворниот код на дадена програма. На пример, ако разгледуваниот програмски код не содржи точки на одлука (како што се `if` наредби или `for`/`while` циклуси), неговата сложеност ќе биде 1, бидејќи постои само еден пат низ овој код. Од друга страна, ако програмскиот код содржи точно една точка на одлука, тогаш ќе има два пата низ кодот: еден пат каде условот што одговара на точката на одлука ќе се евалуира како логички точен и уште еден пат каде условот ќе се евалуира како логички неточен.

Генерално, контролниот тек на која било програма може да се претстави преку граф којшто се нарекува **граф на контролниот тек** или **контролен граф**. Во овој граф, јазлите ги претставуваат групите (блоковите) од наредби на програмата коишто можат да се извршат само во (еден можен) редослед. Два јазли се поврзани со ориентирано ребро ако првата наредба во целниот (завршниот) јазол би можела да се изврши веднаш по извршувањето на последната наредба во изворниот (почетниот) јазол. Вообичаено, ребрата во контролниот граф се резултат на точките на одлука во програмата, односно условното извршување на блок од наредби, како што се `if` разгранувања или условни циклуси.

Контролниот граф за едноставна програма има еден влезен и еден излезен јазол. Од влезниот јазол на графот може да се пристапат сите јазли, а пак излезниот јазол може да го пристапат сите други јазли. Ако се додаде едно вештачко ориентирано ребро од излезниот до влезниот јазол, тогаш контролниот граф станува цврсто сврзан. Пример за изглед на еден контролен граф е прикажан на Слика 2.7.

Претставувањето на софтверот во облик на контролен граф е интересно бидејќи ги визуелизира можните патишта на извршување низ изворниот код. Идејата на McCabe била сложеноста на програмата да се мери преку утврдување на бројот на патишта во нејзиниот контролен граф. Сепак, дури и кога станува збор за едноставни програми, ако тие содржат циклус тогаш бројот на патишта станува бесконечен. Оттука, McCabe одлучил да ги разгледува само независните патишта: комплетни патишта (патишта коишто започнуваат од влезниот јазол и коишто завршуваат со излезниот јазол на графот), такви што нивните линеарни комбинации ги генерираат сите множества од комплетни патишта во контролниот граф на програмата.



Слика 2.7: Пример за контролен граф со 10 ребра, 6 јазли и 6 региони (McCabe, 1976). Цикломатската сложеност на програма претставена со овој граф е 6

Формално, McCabe ја дефинирал **цикломатската сложеност** (M) **на структурирана програма** со равенката

$$M = E - N + 2 \cdot P$$

кадешто E е бројот на ребра, N е бројот на јазли и P е бројот на сврзани компоненти во графот на контролниот тек (контролниот граф) на програмата. За единична програма, P е секогаш еднаков на 1, па формулата станува

$$M = E - N + 2$$

Оваа равенка потекнува од теоријата на графови и е изведена од формулата за број на независни патишта во цврсто сврзан граф.

Сврзаните компоненти во контролниот граф ги претставуваат индивидуалните функции или потпроцедури во програмата. Цикломатската сложеност може да се примени (пресмета) и за индивидуални функции, модули, методи или класи во рамките на програмата.

Бројот на ребра и јазли во контролниот граф не е тривијален за пресметување кога станува збор за големи изворни кодови. За среќа, McCabe успеал да докаже дека цикломатската сложеност на која било структурирана програма со една влезна и една излезна точка е еднаква на бројот на точки на одлука плус еден. Сепак, треба да се забележи дека ова се однесува само на точки на одлука на најниско ниво (инструкции на машинско ниво). При пишувањето на програми во високо-нивовски јазици, програмерите често користат сложени услови и овие точки на одлука треба да се бројат во однос на предикатните променливи што се употребени во сложениот услов (на пример, "`if услов1 and услов2 then …`" треба да се брои како две точки на одлука, бидејќи е еквивалентна со "`if услов1 then if услов2 then …`" на машинско ниво). За програми со повеќе од една излезна точка, цикломатската сложеност

може да се пресмета како $d - e + 2$, кадешто d е бројот на точки на одлука, а e е бројот на излезни точки.

Заради својата едноставност, последниов начин на пресметување на цикломатската сложеност веројатно е и најчесто користениот. Од друга страна, за помали изворни кодови коишто можат да се претстават визуелно преку цврсто сврзани контролни графови, постои дополнително поедноставување во пресметувањето на цикломатската сложеност – во овој случај, таа може да се утврди преку броење на регионите на соодветниот контролен граф.

Цикломатската сложеност на McCabe има цврста математичка подлога и се чини дека го опфаќа (барем делумно) интуитивното значење на сложеноста на контролниот тек. Уште повеќе, McCabe делумно ја валидирал неговата метрика експериментално покажувајќи дека постои корелација помеѓу цикломатската сложеност и некои други атрибути коишто неспорно влијаат врз сложеноста на контролниот тек, како што е сигурноста (McCabe, 1976). Врз основа на овие истражувања, McCabe извел емпириско правило кое вели дека цикломатската сложеност на кој било модул не смее да ја надмине вредноста 10.

Една од најважните предности на цикломатската сложеност како метрика е тоа што таа може да се користи за управување на процесот на динамичко тестирање на функционалноста на програмите (користејќи тест примери). Бидејќи цикломатската вредност ја опишува сложеноста на контролниот тек, јасно е дека на програмите со високи цикломатски вредности им се потребни повеќе тест примери отколку на програмите со ниски цикломатски вредности. Исто како и метриките за сложеност на Halstead, цикломатската сложеност е лесно применлива за кој било програмски јазик, но може да се пресметува порано во животниот циклус на програмата отколку метриките на Halstead (програмата не мора да е комплетирана за да може да се пресмета нејзината сложеност).

Сепак, користењето на оваа метрика има и некои недостатоци. Еден од најсериозните недостатоци е тоа што цикломатската метрика ја игнорира сложеноста што произлегува од податочниот тек на програмите. Како пример, на линеарна програма (којашто содржи само наредби што треба да се извршат секвенцијално, без разгранувања/скокови) од 100 кодни линии ќе ѝ биде доделена идентична цикломатска вредност како и на програма којашто содржи една линија код – што илустрира дека текот на податоците воопшто не се зема во предвид од страна на оваа метрика. Уште повеќе, метриката исто така ја игнорира и сложеноста што произлегува од вгнездување на програмските структури (Suleman Sarwar, Shahzad и Ahmad, 2013), и не прави разлика помеѓу различните контролни структури (на пример, `if` и `if/else`).

Слично како со метриките за сложеност на Halstead, предложени се варијации и проширувања на цикломатската сложеност на McCabe како обиди да се надминат нејзините недостатоци (Henderson-Sellers, 1992; Henderson-Sellers и Tegarden, 1994; Jian-hua и Jia-pei, 2006; Madi, Zein и Kadry, 2013; Myers, 1977; Piwowarski, 1982; Sagri, 1989), но без вистински успех.

2.3.4. Oviedo-вата сложеност на податочниот тек

Инспириран од техники на компајлерската оптимизација, Oviedo (1980) дефинирал метрика за сложеност на програмскиот код, којашто е претставник на перспективата кон софтверската сложеност која се фокусира на податочниот тек. За да може да се објасни оваа метрика, неопходно е да се воведе соодветна терминологија.

- *Дефиниција на променлива* (англ. *variable definition*) претставува појавување на променлива на левата страна од наредба за доделување, или пак како влезен параметар на потпрограма (процедура).
- *Референцирање на променлива* (англ. *variable reference*) претставува појавување на променлива на десната страна од наредба за доделување, како дел од некој предикатен израз, или пак во наредба за излез од потпрограма (“`return`”).
- *Локално достапна (ре)дефиниција на променлива* (англ. *locally available variable (re)definition*) претставува (ре)дефиниција на променлива во рамките на некој блок од наредби.
- *Локално изложено референцирање на променлива* (англ. *locally exposed variable reference*) е референцирање на променлива во рамките на блок во кој таа не е (ре)дефинирана.
- За една дефиниција на променлива направена во блокот k се вели дека *го достигнува блокот* m ако таа променлива не се (ре)дефинира долж патот од блокот k до блокот m.
- (Ре)дефиницијата на променлива во некој блок *ги уништува* сите претходни дефиниции на таа променлива кои би можеле да го достигнат блокот.

Нека V_k е множеството од локално изложени променливи во блокот k, и нека R_k е множеството од дефиниции на променливи што го достигнуваат блокот k. Овде треба да се нагласи дека за секоја променлива што е референцирана во блокот k може да постојат повеќе дефиниции, во зависност од бројот на контролни патишта кои водат до блокот k.

Сложеноста на податочниот тек DF_k *на даден блок* k се дефинира како број на (ре)дефиниции кои го достигнуваат блокот k, од сите променливи што се референцирани (локално изложени) во тој блок:

$$DF_k = \sum_{i=1}^{|V_k|} DF(v_i)$$

кадешто $|V_k|$ е бројот на локално изложени променливи во блокот k, а $DF(v_i)$ е бројот на дефиниции на променливата v_i кои го достигнуваат блокот k. **Сложеноста на податочниот тек на дадена програма** претставува збир од сложеностите на податочните текови на сите блокови во таа програма.

Метриката за сложеност на податочниот тек дефинирана од страна на Oviedo е една од првите метрики којашто се концентрира на сложеноста на манипулацијата на податоци во даден блок од код. Ова сугерира дека оваа метрика е зависна од контекстот, што е нетипично својство за една метрика за мерење на сложеност на програмските кодови. Имајќи го во предвид начинот на којшто е дефинирана, може да се каже дека метриката на Oviedo е (барем теоретски) ортогонална на цикломатската сложеност на McCabe. Како што ќе може да се види од дискусијата во следното поглавје, комбинацијата од овие две метрики е во основата за дефиницијата на метриките за когнитивна сложеност, коишто се мошне популарно поле на истражување во последниве петнаесетина години.

2.3.5. Когнитивна сложеност

Имајќи во предвид дека софтверот е производ на човечка креативна активност, Shao и Wang (2003) посочуваат дека „когнитивната информатика игра важна улога во разбирањето на фундаменталните карактеристики на софтверот". При мерењето на софтверската сложеност, многу од класичните метрики ја земаат во предвид и когнитивната (психолошката) сложеност. Како пример, едно истражување (Curtis *et al.*, 1979) покажало дека метриките за сложеност на Halstead и цикломатската сложеност на McCabe се корелирани со тешкотиите со кои се соочуваат програмерите при разбирањето и модифицирањето на софтверот. Како и да е, во последно време се појавуваат метрики за сложеност кои всушност се фокусираат на мерење на когнитивната сложеност.

Wang (2006) ја дефинира **когнитивната сложеност** како „степенот на релативна тежина или напор потрошен во време за разбирање на функциите и семантиката на дадена програма". Повеќето когнитивни метрики се базирани на *основните контролни структури* (англ. *basic control structures* – BCS) – множество од есенцијални механизми за контрола на текот кои се користат за

изградба на логичката архитектура на софтверот. Овие контролни структури се прикажани во првата редица од Табела 2.1. Како што може да се види од Табела 2.1, на секоја BCS структура ѝ е придружена когнитивна тежина. Тежинската вредност на секоја BCS структура е определена врз основа на претпоставката дека времето што е потребно за разбирање на функционалноста и семантиката на конкретна BCS структура е пропорционално со нејзината когнитивна сложеност. Овој пристап се оправдува со фактот дека релативниот напор (или потрошеното време) е статистички стабилен за сите BCS структури, иако тој може да варира од еден до друг човек (Wang, 2006).

За жал, не постои консензус за когнитивните тежински вредности доделени на секоја од BCS структурите. Табела 2.1 ги прикажува тежинските вредности предложени од страна на Shao и Wang (2003), односно на Wang (2006), во втората односно третата редица, соодветно. Сепак, Gruhn и Laue (2007) посочуваат повеќе проблеми во врска со експериментите коишто биле спроведени за определување на овие тежински вредности и потенцираат дека актуелните тежини на BCS структурите не се задоволителни.

	S1	**S2**	**S3**	**S4**	**S5**	**S6**	**S7**	**S8**	**S9**	**S10**
Когнитивна тежина (Shao/Wang)	1	2	3	3	3	3	2	3	4	4
Когнитивна тежина (Wang)	1	3	4	7	7	8	7	11	15	22

Табела 2.1: Когнитивни тежини за различните основни контролни структури (англ. *basic control structures* – BCS)

Легенда: “**S1**” – Редоследна структура; “**S2**” – Разгранување: `if-then-else`; “**S3**” – Разгранување: `switch-case`; “**S4**” – Итерација: `for` циклус; “**S5**” – Итерација: `repeat-until` циклус; “**S6**” – Итерација: `while` циклус; “**S7**” – Функциски повик; “**S8**” – Рекурзија; “**S9**” – Конкурентност: паралелно извршување; “**S10**” – Конкурентност: извршување со прекини.

Когнитивната сложеност на дадена програма се пресметува користејќи ги тежините придружени на BCS структурите. Ако не постојат вгнездени контролни структури, сложеноста на програмата може да се пресмета како збир од тежините на сите контролни структури присутни во неа. Сепак, вообичаена ситуација во повеќето програми е да постојат контролни структури коишто се вгнездени во други контролни структури. Во ваквите случаи, едноставното собирање нема да е доволно.

Во најгенералниот случај, Wang (2006) ја дефинира **когнитивната сложеност** $C_c(S)$ на софтверски систем S како производ од неговата оперативна сложеност $C_{op}(S)$ и архитектурната сложеност $C_a(S)$ т.е.

$$C_c(S) = C_{op}(S) \cdot C_a(S)$$

Оперативната сложеност $C_{op}(S)$ на даден систем S што содржи q линеарни блокови од код, при што секој блок содржи по m_j вгнездени нивоа и секое ниво содржи по $n_{j,k}$ индивидуални BCS структури во секвенца, се дефинира со

$$C_{op}(S) = \sum_{j=1}^{q} \left(\prod_{k=1}^{m_j} \sum_{i=1}^{n_{j,k}} w(j,k,i) \right)$$

каде $w(BCS)$ е тежинската вредност придружена на конкретна BCS структура, во согласност со Табела 2.1. *Архитектурната сложеност* $C_a(S)$ на системот S се дефинира како збир од бројот на глобални променливи и бројот на локални променливи што се користат во S.

Wang ги разгледувал оперативната и архитектурната (или податочната) сложеност како ортогонални димензии коишто мора да се вклучат во целокупната сложеност на софтверот. Тој ја дефинирал единицата за оперативна сложеност како единична секвенцијална операција наречена *функција*, којашто се означува со F. Од друга страна, единица за архитектурна сложеност е единичен *податочен објект*, којшто се означува со O, и над кој се применува функција. Според тоа, единица за когнитивна сложеност е единичен *функција-објект* (FO), што одговара на сложеноста на систем којшто изведува единична секвенцијална операција (функција) над единичен податочен објект.

Врз основа на метриката опишана погоре, предложени се повеќе модифицирани метрики за когнитивна сложеност (Jakhar и Rajnish, 2014; S. Misra *et al.*, 2017; Shehab *et al.*, 2015). Меѓу другите, тука спаѓаат и метриката за *когнитивна информациска сложеност* (англ. *Cognitive Information Complexity Measure* – CICM) (Kushwaha и A. Misra, 2006), како и метриката за *когнитивна програмска сложеност* (англ. *Cognitive Program Complexity Measure* – CPCM) (S. Misra, 2007).

2.3.6. Анализа на применливоста на постојните метрики за софтверска сложеност во едукативен контекст

Како што беше објаснето во Глава 1, истражувањето во рамките на оваа дисертација се фокусира на автоматското генерирање на прашања што содржат програмски код за потребите на едукативната проверка на знаење кај воведните курсеви за програмирање. Со цел да се постигне конзистентност на сложеноста

во процесот на автоматско генерирање на прашања, потребен е начин на автоматско мерење на сложеноста на генерираните кодови. Според тоа, неопходна е соодветна метрика за мерење на сложеноста на даден програмски код, којашто ќе ја разгледува сложеноста во однос на напорот што треба да го вложи конкретен студент за да ги разбере и изврши сите наредби во кодот. Во ова поглавје ќе биде оценет потенцијалот за примена во едукативна околина на секоја од метриките што беа дискутирани во претходните поглавја, во контекст на опишаниот проблем.

Очигледно е дека метриката LOC, сама за себе, не е доволна за мерење на сложеноста (во однос на напорот или потрошеното време) на даден коден сегмент, којшто треба да биде извршен рачно (од страна на студент), како дел од прашање. Се разбира, пожелно е студентите на кој било конкретен испит да добијат прашања што содржат кодни сегменти со исти (или многу блиску до исти) LOC вредности. Сепак, како што беше објаснето и претходно, метриката LOC целосно ја игнорира сложеноста на секоја индивидуална линија и ја занемарува структурата на кодот, па не можеме да се потпреме само на неа.

Метриките за сложеност на Halstead исто така ја занемаруваат структурата на кодот – што е сериозен недостаток. Контролните структури играат голема улога во сложеноста на програмите, од аспект на студентскиот напор потребен за разбирање и извршување на програмските наредби. Бројот на оператори и бројот на операнди во кодот, кои се основа за пресметување на овие метрики, несомнено имаат силно влијание врз неговата сложеност. Како и да е, постојат различни типови на оператори (аритметички, релациони, логички, итн.) што треба да се разгледаат, а уште повеќе и – различни типови на операнди. Овде станува збор за операции што се извршуваат рачно, па собирање на едноцифрен со двоцифрен операнд не е операција со иста сложеност како операцијата собирање на 4-цифрен со 5-цифрен операнд – втората операција секако би земала повеќе време за пресметка во однос на првата. Уште повеќе, би се очекувало дека множењето на двоцифрен со двоцифрен операнд би требало да земе повеќе време отколку собирањето на два операнди од истиот тип. Истиве забелешки се однесуваат и на метриката на Oviedo – оваа метрика го опфаќа текот на податоците во кодот, но ги игнорира контролните структури, како и типот на податоците и типот на манипулација што се изведува над нив.

Од друга страна, метриката за сложеност на McCabe е теоретски ортогонална на метриките на Halstead и Oviedo – таа се фокусира на контролниот тек, но целосно ја занемарува сложеноста на податочниот тек на програмите. Уште повеќе, метриката не прави разлика помеѓу различните контролни структури, што исто така е неприфатливо за разгледуваниот проблем. Како пример, наредбите `if` и `if`/`else` имаат иста цикломатска сложеност бидејќи имаат ист број на извршни патишта. Сепак, кога условот на

дадена наредба `if` ќе се евалуира како логички неточен, студентот нема да треба да извршува ниту една наредба поврзана со оваа контролна структура, додека во случај кога условот на дадена наредба `if`/`else` ќе се евалуира како логички неточен – студентот ќе треба да ги изврши сите наредби што се наоѓаат во `else` блокот (што би можело да биде значително поголемо оптоварување!). Конечно, вгнездувањето на контролни структури може значително да го зголеми бројот на пресметки што ќе треба да ги направи студентот – ако даден `for` циклус кој прави 10 итерации е вгнезден во телото на друг `for` циклус со 10 итерации, дури и ако е потребна само една пресметка во телото на внатрешниот циклус, ова ќе значи дека студентот ќе треба да направи 100 пресметки.

Се чини дека когнитивната сложеност на Wang поседува најмногу особини кои може да се сметаат како посакувани за метрика којашто би била применлива за едукативни цели. Поимот за когнитивна сложеност (тежина) којашто се придружува на секоја од основните контролни структури, како мерка за потребниот напор за разбирање на нејзината функционалност и семантика, дефинитивно треба да се вклучи во дефиницијата на посакуваната метрика. Различните контролни структури побаруваат различно ниво на напор (и време) за да бидат разбрани и рачно извршени од страна на студентите. Метриката на Wang исто така се справува добро и со вгнездувањето на структури, а се обидува да го инкорпорира и податочниот тек во мерењето на сложеноста (преку броење на променливите). Но, исто како и метриките на Halstead и Oviedo, таа не ги разгледува типовите на операции и операнди со коишто ќе треба да се справуваат студентите.

Конечно, сите анализирани метрики ја мерат сложеноста преку разгледување на комплетниот изворен код, а не само на „кодот што ќе биде посетен“ при познати вредности на програмските променливи. Според тоа, покрај причините што беа елаборирани погоре, и овој аргумент исто така води кон заклучокот дека ниту една од метриките не е директно применлива за опишаниот проблем во едукативна околина.

2.4. Странични истражувања и придонеси во сродни процеси

Како поддршка на главниот фокус на оваа докторска дисертација, паралелно се спроведени повеќе странични истражувања и постигнатите резултати се објавени во повеќе трудови во меѓународни списанија или научни конференции од областа на информатичката едукација. Во ова поглавје е направен нивен краток преглед.

Во магистерскиот труд (Станков, 2013) предложен е нов, оригинален начин на претставување на програмските кодови во форма на вектори од атрибутски вредности. Сите атрибути во предложената векторска

репрезентација на кодовите ја опишуваат нумерички нивната структура. За потребите на генерирањето на опишаните векторски репрезентации на програмските кодови, во рамките на ова истражување развиена е и соодветна софтверска алатка (Stankov *et al*., 2013a). Исто така, предложен е и нов хибриден пристап за откривање на сличност помеѓу програмските кодови, којшто се заснова на користење на кластерирачки методи на податочното рударење над атрибутските репрезентации на кодовите. Со експериментална евалуација на предложениот пристап, користејќи два конкретни методи (K-Means и EM), покажано е дека кластерирачките методи можат да се користат за определување на сличност помеѓу програмските кодови со задоволителна точност (Stankov, Jovanov и Madevska Bogdanova, 2013). Конечно, врз база на овој пристап, *предложен е и нов алгоритам за откривање на кандидат – програми за повторно прегледување, односно нов модел за полуавтоматско оценување на програмски кодови* (Stankov *et al*., 2013b). Воедно, експериментално е тестирана и неговата корисност над конкретни множества од програмски решенија предадени од страна на студентите во рамките на воведен курс за програмирање.

Врз основа на овие идеи, во трудот (Stankov *et al*., 2015b) *предложен е нов модел за колаборативно учење на програмирање*, кој може да се примени кај системите за автоматско оценување на програмски решенија (со користење на тест примери). Во овој модел се користат кластерирачки техники на податочното рударење за да се кластерираат претходно предадените точни решенија за некоја програмска задача, а со цел да се открие кои од овие решенија се најслични со новопредаденото (неточно) решение од некој друг корисник. Ова му овозможува на системот за автоматско оценување да ги контактира авторите на соодветните точни решенија за тие да дадат наративна повратна информација на тој корисник (колаборативност). Спроведените експерименти и добиените резултати од нив претставуваат валидација за предложениот модел.

Во трудот (Angelovski, Stankov и Jovanov, 2021) е предложена *нова алатка за полуавтоматско оценување на програмски кодови*. Таа ги анализира C/C++ програмските кодови и нивните резултати при автоматската евалуација (со користење на тест примери), и со помош на машинско учење, обезбедува информации во врска со веројатноста дека дадено програмско решение (предадено од корисник) треба да биде рачно прегледано. Фокусот во овој труд е на подобрување на моделот за полуавтоматско оценување на програмски кодови предложен во претходната работа (Stankov *et al*., 2013b). Подобрувањата вклучуваат директна статичка анализа на кодовите што не компајлираат, како и метрики за рангирање на програмските кодови. Овде се претставени и резултатите од примената на подобрениот модел над соодветни тестни податоци, коишто даваат солидна основа за понатамошна употреба.

Интересот за истражувања во насока на подобрување на информатичкото образование во државата и глобално, како преку подобрување на наставните програми, така и преку вон-наставните активности како што се натпреварите по информатика, е отсликан во постигнувањата изложени во следните трудови.

Во (Ackovska *et al*., 2015) е изложен преглед на актуелната ситуација со информатичката едукација во основното и средното образование во тринаесет држави ширум светот. Врз основа на спроведеното истражување, овде се дадени некои насоки и препораки во врска со креирање на интернационална наставна програма од информатика за овие нивоа на образование, меѓу кои е и задолжителното вклучување на наставни методи и техники за развој на алгоритамско размислување и основни програмерски вештини кај учениците. Фокусот на трудот (Jovanov *et al*., 2016) е на актуелната промена во наставната програма за основно образование во Македонија во 2015-та година: воведувањето на предметот „Работа со компјутери и основи на програмирање“. Овде е претставена предложената (а воедно и прифатена) наставна програма, со акцент на темите во врска со алгоритамското размислување и програмирањето. Исто така, дискутирани се некои достапни софтверски апликации и алатки што се соодветни за имплементацијата на овие теми.

Во (Jovanov *et al*., 2017a) се елаборирани дел од многуте фактори што влијаеа на генерациите од компјутерски научници и инженери кои се стекнаа со диплома за завршено високо образование на ФИНКИ, како и на институциите што му претходеа на овој факултет пред 2011-та година. Некои такви фактори, истражувани во овој труд, се промените во образовните системи на основното и на средното образование и аспектите на овие системи кои влијаат врз информатичкото знаење на учениците, од каде што се извлекуваат некои заклучоци за начинот на оценување на знаењата за програмирање. Трудот (Dimitrievska Ristovska, Stankov и Sekuloski, 2021) дава компаративна анализа на спроведувањето на некои курсеви на ФИНКИ, во класични услови наспроти во услови на далечинско учење (за време на корона кризата). Анализата е направена од аспект на пристапот кон држењето на наставата, како и од аспект на спроведувањето на испитите и постигнатите резултати. Како заклучок, извлечени се некои позитивни и негативни страни на далечинското учење, во споредба со класичната настава.

Во (Kostadinov *et al*., 2015) анализирани се неколку различни типови на натпревари, начинот на којшто придонесува секој од нив за промоција на компјутерските науки, како и начинот на којшто тие можат да помогнат во остварување на иницијален избор на талентирани ученици за подоцнежните етапи од натпреварите. Врз основа на ова, предложени се седум критериуми за класификација на натпреварите. Конечно, врз основа на долгогодишното искуство со организацијата на македонските натпревари по информатика,

предложени се и различни пристапи за правењето на иницијален избор на талентирани ученици за натпреварите во програмирање, а дискутирани се и можни начини за одбегнување на некои од проблемите со стандардните натпревари, на пример – преку обезбедување на повратни информации и воведување на поделби според тежината. (Kostadinov, Jovanov и Stankov, 2018) презентира платформа којашто може да ја анализира активноста на кој било ученик/студент на различни онлајн системи за натпревари по информатика и за е-учење, да ги комбинира и процесира податоците, а потоа да ги прикажува на разни начини коишто се лесни за разбирање.

Во (Jovanov *et al*., 2017b) е претставен успешен пример за имплементација на учење базирано на проекти, во рамки на курсот ИТ системи за учење кој се држи на ФИНКИ. Даден е преглед на организацијата на курсот, објаснета е важноста на учењето базирано на проекти како метод на учење, придобивките од тимската работа, а опишана е и намената на едукативните игри.

Во (Ilijoski, Popeska и Stankov, 2018) е претставена споредба на ефикасноста на одговорите помеѓу прашањата со избор од повеќе можни одговори (ПМО прашања) и есејските прашања (прашања со внесување на одговор), кај неколку курсеви од областа на компјутерските науки. Идејата на ова истражување е да се открие корелацијата помеѓу овие две групи од одговори, да се споредат остварените просечни резултати (просечен број на освоени поени), и да се процени минималниот процент на прашања со кратки одговори/есејски прашања што е потребен за оценување на знаењето на даден студент. Ова би им помогнало на наставниците да креираат ефикасни тестови за оценување на знаењето што го поседуваат студентите.

3. Нов модел за паметно автоматско генерирање на прашања што содржат програмски код

3.1. Дефиниција на моделот

Во оваа докторска дисертација се предлага нов модел за паметно автоматско генерирање на прашања што содржат програмски код, којшто ќе биде директно применлив при процесот на проверка на знаење кај воведни курсеви за програмирање. Моделот ги предвидува следниве чекори:

1. **Креирање на почетен шаблон** (актер: наставник) – се креира прашање што содржи почетен програмски код – шаблон. Текстот на прашањето е „Кој е излезот од дадениот код?“;
2. **Компајлирање, парсирање и детекција на локации од интерес во шаблонскиот код** (актер: систем) – шаблонскиот код прво се компајлира. Во случај ако се пронајдени синтаксни грешки истиот се отфрла. Во спротивно, кодот се парсира за да се детектираат локациите од интерес. *Локации од интерес* се позиции во кодот над кои е дозволено да се извршуваат модификации, користејќи соодветни правила.
3. **Пресметување на сложеност на шаблонскиот код** (актер: систем) – со користење на метрика за мерење на сложеност на програмски кодови (соодветна за намената – едукативна проверка на знаење), се пресметува сложеноста на шаблонскиот програмски код;
4. **Дефинирање на правила за модификација на шаблонот и дефинирање прагова вредност за сложеноста на кодовите** (актер: наставник) – се дефинираат правилата за модификација на локациите од интерес во шаблонот. За секоја локација се одредува дали ќе биде модифицирана, како и начинот на којшто ќе се изведува модификацијата доколку таа е дозволена за соодветната локација. Исто така, се дефинира и прагова вредност, која служи за контрола на девијацијата на сложеноста на кодовите што ќе се генерираат, во однос на онаа на почетниот (шаблонски) код.
5. **Формирање на нови прашања преку генерирање на нови програмски кодови од шаблонскиот код** (актер: систем) – со примена на дефинираните правила за модификација, се генерираат прашања што содржат нови програмски кодови добиени од шаблонскиот код. За секој нов програмски код се пресметува неговата сложеност (повторно со користење на метриката за мерење на сложеност на кодови). Прашањата што содржат кодови чијашто сложеност го надминува дозволеното отстапување во однос на сложеноста на шаблонскиот код – се отфрлаат.

Во следното поглавје ќе биде претставена метрика за мерење на сложеност на програмски кодови, којашто е соодветна за употреба во рамките на овој модел.

3.2. Нова метрика за мерење на софтверска сложеност

3.2.1. Опис на новата метрика

Во овој дел ќе биде опишана нова метрика за мерење на сложеност на програмски кодови (Stankov *et al.*, 2015a), којашто ја разгледува сложеноста на даден код од аспект на напорот што треба да го вложи студентот за да го разбере кодот (когнитивно процесирање), рачно да ги изврши сите операции и наредби присутни во него и да го пресмета излезот, под претпоставка дека се познати вредностите на сите променливи. Оваа метрика може да се користи за мерење на сложеноста на изворен код на програма, напишан во програмскиот јазик C/C++. Како и да е, истата може да се прошири за употреба за програмски кодови напишани во кој било програмски јазик.

Во пристапот што се предлага, се претпоставува дека на сите наредби за разгранување и итерација на програмскиот јазик C/C++ (`if`, `switch`, `while`, `do-while` и `for`), како и на најчесто користените C/C++ оператори (аритметичките оператори: `+`, `-`, `*`, `/` и `%`; релационите оператори: `<`, `>`, `<=`, `>=`, `!=` и `==`; логичките оператори: `!`, `&&` и `||`; итн.), им е придружена специфична когнитивна тежинска вредност. Секоја од овие тежински вредности го претставува напорот потребен (од страна на човек) за рачно изведување на соодветната операција или рачно извршување на соодветната наредба за разгранување/итерација. Ако се познати тежините на сите оператори и сите наредби за разгранување/итерација, тогаш сложеноста C на даден C/C++ код е дефинирана со

$$C = \sum_{i=1}^{n} w_i \cdot e_i$$

кадешто n е бројот на линии во кодот, w_i е тежината придружена на линијата i, а e_i е бројот на извршувања на линијата i во едно извршување на програмскиот код. Тежината придружена на дадена линија претставува збир од тежините на сите оператори и сите наредби за разгранување/итерација коишто се присутни во таа линија.

Како што беше објаснето погоре, дефинираната метрика се заснова на когнитивни тежински вредности придружени на секој од операторите и секоја од контролните наредби во кодот, кои треба да ги претставуваат нивните „сложености“, во смисла на напорот и времето коешто треба да го потроши конкретен студент за рачно да ги изврши соодветните операции/наредби и да ги пресмета соодветните резултати. Според тоа, придружените тежински

вредности треба колку што е можно попрецизно да ги претставуваат односите помеѓу времињата што му се потребни на студентот за извршување на соодветните операции/наредби. Во Глава 5 ќе бидат претставени експериментите и опишани добиените резултати во рамките на истражувањето (Stankov *et al*., 2016; Stankov, Jovanov и Madevska Bogdanova, 2017) коешто беше спроведено со цел определување на соодветни тежински вредности за аритметичките оператори.

Предложената метрика го зема во предвид фактот дека понекогаш, дури и кај многу сложен код, студентот може едноставно да го пресмета излезот од кодот во случаите кога тој е независен од голем дел од него. Таа ја опфаќа сложеноста која произлегува и од контролниот и од податочниот тек, и има потенцијал да ги разреши сите проблеми што беа идентификувани кај другите метрики (Stankov *et al*., 2018) во Глава 2.

Начинот на којшто е дефинирана метриката, според погорната дискусија, суштински овозможува таа да се користи кај проблемот што се разгледува.

3.2.2. Пресметување на сложеност на програмски код користејќи ја новата метрика

За пресметување на сложеноста на даден програмски код со користење на предложената метрика, потребно е да се знаат тежините придружени на сите оператори и контролни наредби, тежината на секоја линија, како и бројот на извршувања на секоја линија од кодот.

Ако се познати вредностите за тежините на операторите и контролните наредби, истите може да се применат на даден код така што прво од кодот ќе се генерира апстрактното синтаксно дрво (англ. *abstract syntax tree* – AST), коешто може да се употреби за детектирање на операторите/наредбите. Изминувајќи го систематски целото дрво, за секој(а) оператор/наредба од интерес се забележува неговата/нејзината сложеност (тежинска вредност), како и линијата во која се наоѓа тој/таа оператор/наредба. Како резултат на ова изминување на дрвото се добива сложеноста на секоја линија во кодот.

Доколку не се користи апстрактно синтаксно дрво, обидот да се пресмета тежината на секоја линија преку едноставно изминување на токените од програмскиот код, за јазици (како што е C/C++) кои се тешки за процесирање, може да доведе до проблем на двосмисленост (Ritchie, 1993). Пример за израз, којшто би довел до овој проблем при ваквиот пристап на процесирање, е изразот "`a * b`" во програмскиот јазик C/C++. Овој израз би можел да се интерпретира како „множење на операндот `a` со операндот `b`", но исто така би можел да се интерпретира и како „декларирање на покажувач `b` од тип `a`".

За разлика од тежините на линиите, кои се пресметуваат преку парсирање на кодот, бројот на извршувања на секоја линија најлесно може да се пресмета со помош на специјални готови алатки кои го извршуваат кодот и истовремено прибираат информации во врска со неговото извршување. Овие алатки вообичаено се употребуваат за тестирање на даден софтвер, со цел да се утврди кои линии од неговиот код ги опфаќа секој од множество на преддефинирани тест случаи.

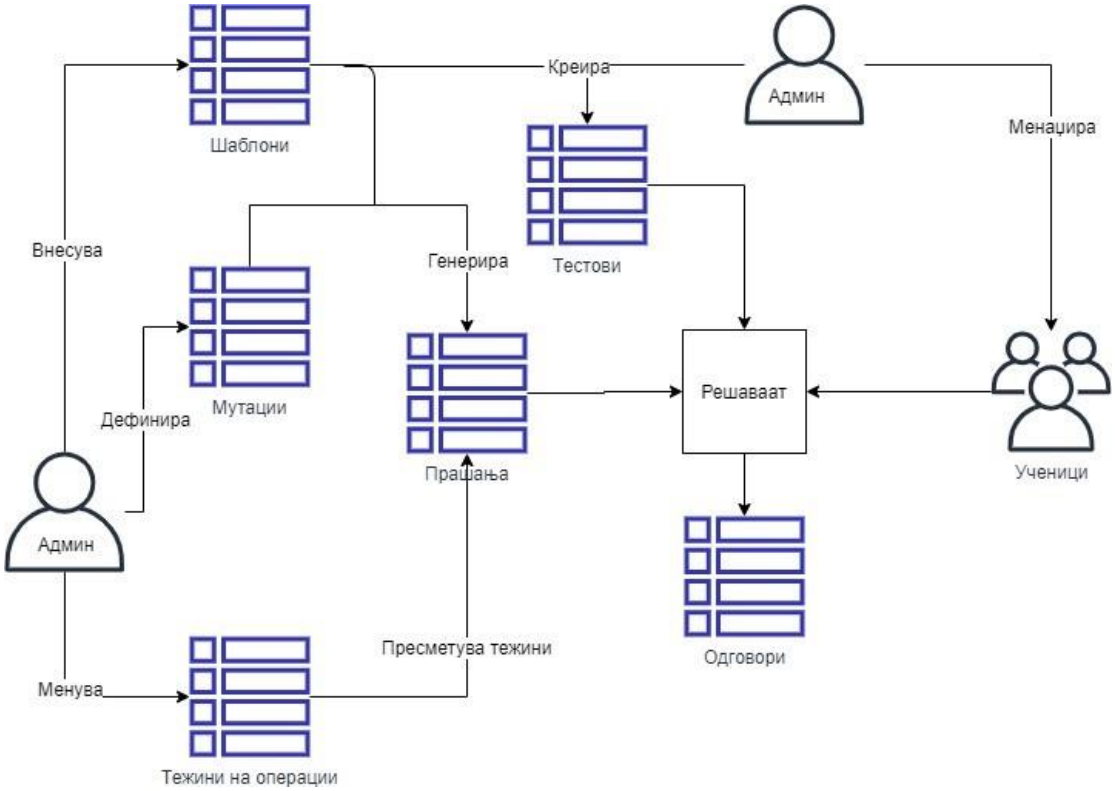
3.3. Систем за паметно автоматско генерирање на прашања што содржат програмски код

Во ова поглавје ќе биде претставен нов софтверски систем за паметно автоматско генерирање на прашања што содржат програмски кодови. За даден (почетен) програмски код, овој систем ја пресметува сложеноста користејќи ја предложената метрика, и со примена на соодветни модификациски техники генерира голем број на нови програмски кодови со иста или произволно блиска сложеност еден со друг (користејќи кориснички-дефинирана прагова вредност за контрола на девијацијата на сложеностите). Овој систем може да се користи за автоматско креирање на соодветни прашања што содржат програмски кодови со конзистентна сложеност, па како таков претставува имплементација на предложениот модел за паметно автоматско генерирање на прашања. Покрај креирањето на прашања што содржат програмски кодови генерирани од даден (шаблонски) код, системот исто така овозможува и формирање на тестови составени од прашања добиени во процесот на генерирање од различни шаблони (по избор на наставникот), како и спроведување на самото тестирање. Според тоа, истиот може да се користи во практика за потребите на процесот на проверка на знаење кај воведните курсеви за програмирање.

3.3.1. Опис на системот

Системот за паметно автоматско генерирање на прашања во суштина е веб апликација до која може да се пристапи со обичен веб прелистувач. На Слика 3.1 е претставен поглед од високо ниво на архитектурата на софтверскиот систем.

Постојат два типа (две улоги) на корисници: администратори (наставници) и студенти. За администраторот на апликацијата постои посебен администраторски панел, кадешто му е овозможен преглед на сите шаблони, правилата за модификација (мутација) што се користат за генерирање на нови кодови од секој шаблон, вредностите на тежините на сите операции/наредби, тестовите и корисниците на апликацијата. Од друга страна, студентите имаат ограничен пристап во системот, којшто се состои само од решавање на даден тест или пак преглед на резултати од тестирање во коешто учествувале.



Слика 3.1: Поглед на архитектурата на системот за паметно автоматско генерирање на прашања што содржат програмски код

На почеток, администраторот внесува програмски код – шаблонски код за генерирање нови кодови (секој код како составен дел од прашање), кој прво се компајлира за да се провери дали претставува валиден C/C++ код. Подоцна, истиот се парсира за да се детектираат сите локации од интерес над кои може да се извршат модификации (мутации) за дадениот шаблонски код. Администраторот има преглед на можните мутации за секој шаблонски код и истите може да ги вклучи/исклучи, како и да им ги конфигурира соодветните параметри. Кога е задоволен со параметрите, администраторот има можност да овозможи генерирање на прашања од шаблонот. Во позадина, независен сервис континуирано ја следи листата на шаблони и за секој шаблон за кој е овозможено генерирање на прашања, го прави истото додека не се добие задоволителен број на нови прашања од соодветниот шаблон.

Администраторот исто така има можност и за креирање и закажување на тестови. За секој тест потребно е да се наведе терминот во кој ќе се одржи, времетраењето на тестот, шаблоните од кои ќе се бираат прашања и корисниците (студентите) кои ќе учествуваат во тестот. Со ова, секој предвиден учесник во тестот ќе може да пристапи до истиот во закажаниот термин за спроведување на тестирањето.

При решавање на даден тест, секој студент добива по едно од генерираните прашања од секој шаблон што е вклучен во тестот, при што за

секое прашање се бара да се внесе одговор – излезот од соодветниот код содржан во прашањето. Притоа, за секое прашање од тестот што го решавал кој било студент, се води евиденција за добиениот одговор, како и за времето коешто му било потребно на студентот за да ги изврши соодветните операции и наредби во кодот и да го одговори прашањето (да го пресмета излезот). Овие информации може да се искористат за понатамошно калибрирање на тежините на сите оператори/наредби од програмскиот јазик што е предмет на интерес.

3.3.2. Генерирање на прашања

Сите прашања коишто се генерираат од страна на системот содржат точно по еден C/C++ програмски код, добиен во текот на генерирачкиот процес, како и текст од обликот: „Кој е излезот од дадениот код?“. Секој програмски код, којшто е составен дел на генерирано прашање, се добива од даден почетен т.н. шаблонски програмски код, со примена на техниката на модифицирање (мутирање) на одредени делови од него.

За да се отпочне процесот на генерирање, од администраторот (наставникот) се очекува да внесе шаблонски код, како и да дефинира соодветни правила според кои ќе се вршат модификациите (мутациите) врз истиот. Внесениот шаблонски код потоа се парсира, за да се пресмета неговата сложеност користејќи ја предложената метрика, како и за да се детектираат локациите од интерес над кои може да се извршат мутации. *Локации од интерес* во даден шаблонски код се позиции во кодот кадешто се присутни литерали (нумерички или ненумерички константи) или оператори. Во зависност од типот на вредноста којашто е присутна во локацијата од интерес во шаблонскиот код, за мутација можат да се користат правила од обликот:

- Промена на целобројна (англ. *integer*) константа;
- Промена на константа – број со подвижна точка (англ. *float*);
- Промена на знаковна (англ. *char*) константа;
- Промена на текстуална (англ. *string*) константа;
- Промена на оператор.

За секое вакво правило, администраторот може да изврши дефинирање на доменот на вредностите од којшто ќе се пополнуваат локациите од интерес во секој нов код што ќе се генерира од шаблонскиот код. На пример, ако станува збор за локација од интерес на којашто е присутна целобројна вредност во шаблонскиот код, може да се дефинира дека во секој новогенериран код таа локација ќе се пополнува со цел број од опсегот [–10, 10]. Опциите за конфигурирање на различните правила за мутација во рамките на системот се подетално опишани во следното поглавје.

По детектирањето на сите локации од интерес во шаблонскиот код од страна на апликацијата и дефинирањето на правилата за мутација на секоја локација од страна на наставникот, сѐ што преостанува е да се изминат сите локации и за секоја од нив да се генерира нова вредност за секој нов код (ново прашање) што се креира од шаблонот. Потоа, користејќи ја предложената метрика, се пресметува сложеноста на секој од вака добиените кодови. За да се одржи конзистентност на сложеноста во процесот на генерирање на програмски кодови, се наметнува ограничувањето секој новодобиен програмски код задолжително да има сложеност којашто не го надминува збирот од сложеноста на шаблонскиот код и преддефинирана прагова вредност (којашто, исто така, може да ја конфигурира администраторот за секој шаблон поединечно). Во случај ако сложеноста на кодот го надминува овој збир – кодот се отфрла.

Бидејќи секое прашање добиено во генерирачкиот процес сеуште содржи валиден програмски код, истиот може да се изврши за да се пресмета неговиот излез. Излезите од кодовите подоцна се употребуваат при валидација на точноста на одговорите на соодветните прашања предадени од студентите за време на тестирање.

3.3.3. Употреба на системот

При пристапување кон веб апликацијата, на корисниците им е претставена страница за логирање кадешто може да се најават со своето корисничко име и лозинка. Бидејќи се работи за апликација преку којашто можат да се реализираат тестирања организирани од наставници, креирањето на нови корисници им е овозможено само на администраторите.

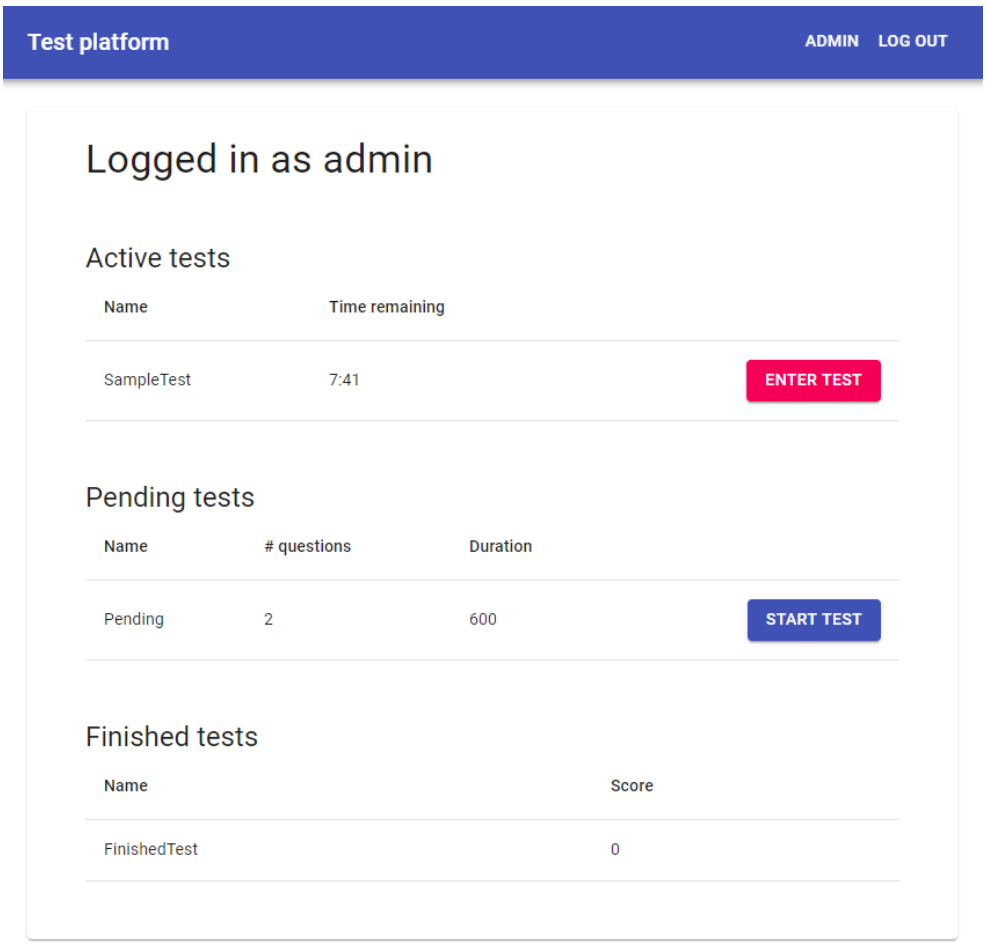
При првото најавување се запишува колаче во веб прелистувачот на корисникот, кое потоа се употребува при секое последователно посетување на сајтот. Колачето важи 24 часа, или пак додека корисникот не се одјави своеволно.

3.3.3.1 Перспектива на студент

Веднаш по најавувањето, студентот пристапува кон почетната страница, кадешто се прикажани сите тестови во кои тој (може да) учествува (Слика 3.2). Постојат 3 типа на тестови:

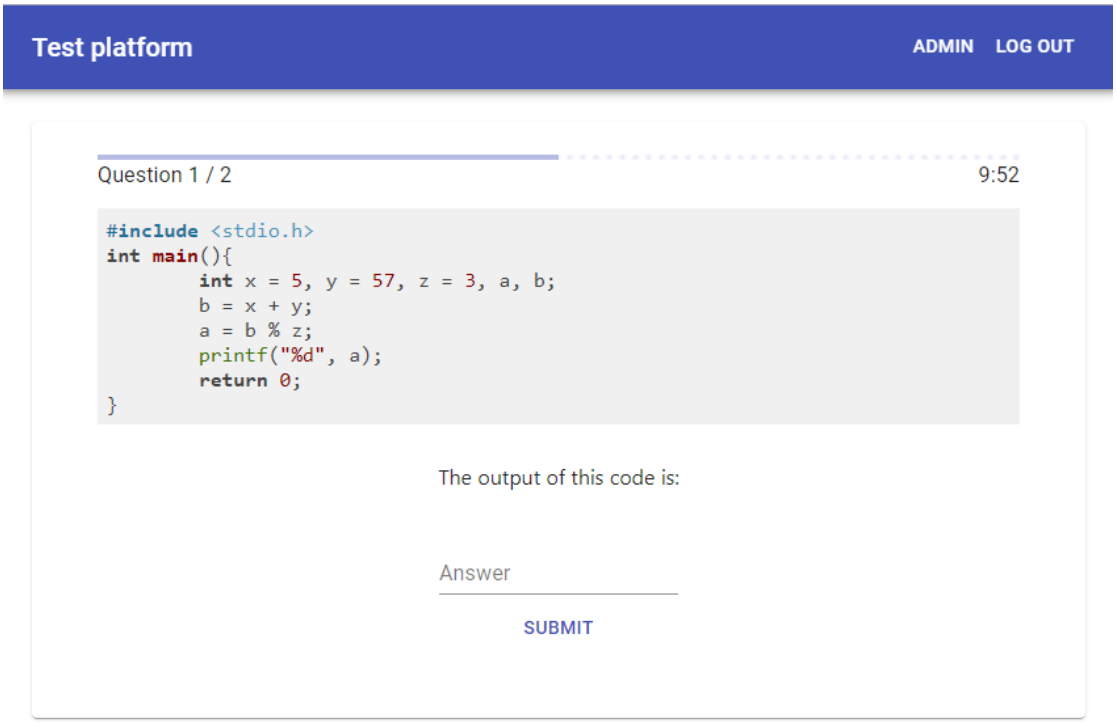
- *Активни тестови* – тестови кои веќе се започнати и за коишто тече времето. Со кликнување на копчето “enter test” се пристапува кон тестот.
- *Незапочнати тестови* – тестови кои се достапни на корисникот, но кои сеуште не се започнати и за истите не тече времето. Со кликнување на копчето “start test” тестот станува активен.

- *Завршени тестови* – тестови кои се веќе комплетирани (или се одговорени сите прашања од тестот, или пак времето за решавање е истечено). Тука се прикажуваат резултатите за сите завршени тестови.



Слика 3.2: Поглед на почетната страница што се појавува при најавување на корисник на системот

Доколку студентот пристапи кон некој закажан тест до којшто му е доделен пристап (од страна на администратор), тој започнува со одговарањето на прашања (Слика 3.3). Во овој приказ, студентот има целосен преглед на тестот. Најгоре е прикажана лентата со прогрес којашто дава информации за бројот на одговорени прашања, бројот на преостанати прашања, како и преостанатото време за решавање на тестот. Под овие информации е даден кодот на прашањето, прикажан со „означување на синтаксата“ (англ. *syntax highlighting*) заради подобра читливост за студентите. Најдолу се наоѓа текстуално поле во коешто се очекува од студентот да го внесе бараниот одговор за прашањето. При кликнување на копчето “submit”, внесениот одговор се испраќа кон серверот, заедно со времето потрошено на решавање на соодветното прашање од страна на студентот, по што се прикажува следното прашање. Во случај пак ако се одговорени сите прашања, тестот е завршен и корисникот се пренасочува кон почетната страница.



Слика 3.3: Решавање на тест од страна на студент во системот – приказ на едно прашање заедно со текстуално поле за внесување на одговорот

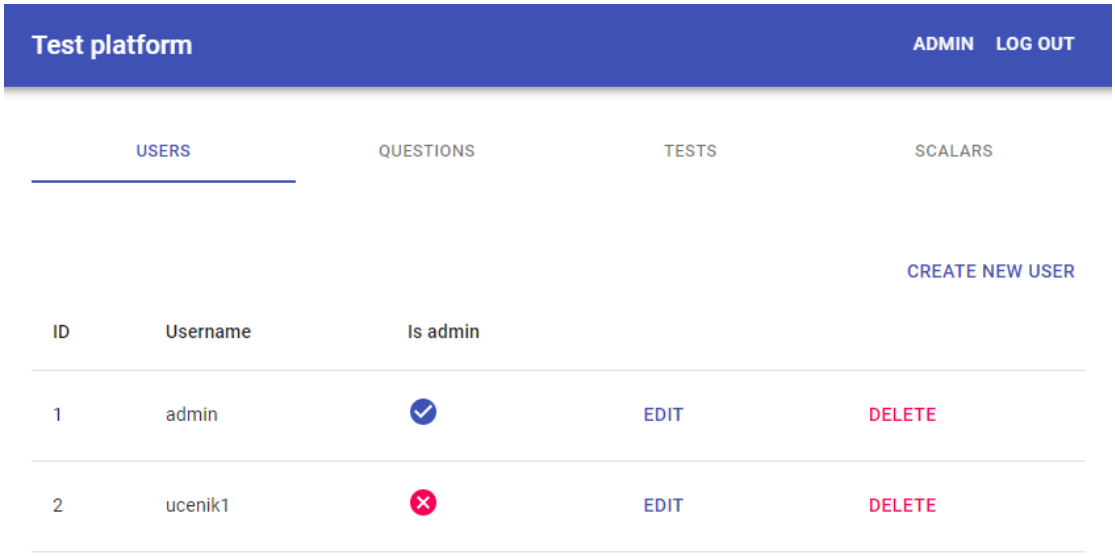
3.3.3.2 Перспектива на администратор

Администраторот е единствениот тип на корисник кој има пристап до администраторскиот панел (Слика 3.4). Тој исто така може да пристапува и да решава тестови, како и кој било студент. При пристап кон администраторскиот панел се прикажува мени кадешто може да се избере помеѓу управување со корисници (users), шаблони (questions), тестови (tests) или тежини на оператори/наредби (scalars).

Слика 3.4: Поглед на администраторскиот панел на системот

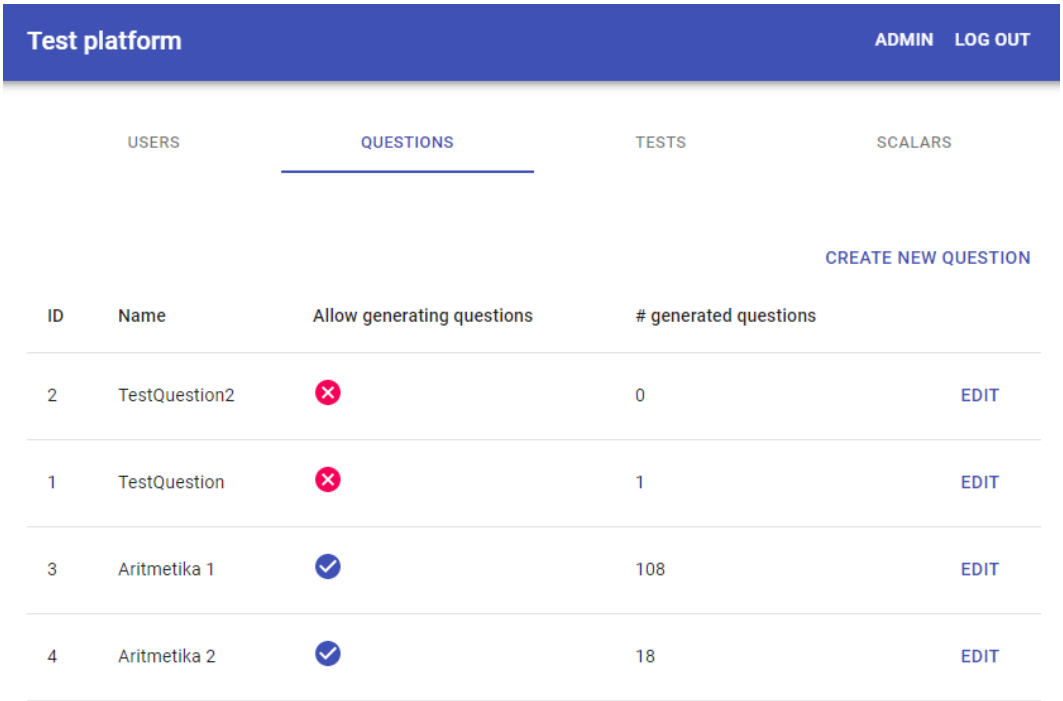
При отворање на панелот за управување со корисници (панелот users), се прикажува листа на сите регистрирани корисници на апликацијата (Слика 3.5). За секој корисник се чува корисничкото име, лозинката (во енкриптирана форма), како и логичка вредност (англ. *boolean value*) – дали корисникот е администратор. Покрај секој корисник постојат копчиња за менување на неговите податоци и за бришење на корисникот. Исто така, во горниот десен агол на овој панел постои и копче “create new user” кое овозможува внесување на нов корисник во системот. При кликнување на некое од копчињата “create

new user" или "edit" (Слика 3.5) се појавува форма преку којашто може да се креира или измени корисник (може да се променат корисничкото име, лозинката и/или администраторскиот статус).



Слика 3.5: Поглед на панелот за управување со корисниците на системот

При отворање на панелот за управување со шаблони (панелот questions), се прикажува листа на сите шаблони кои се внесени во апликацијата (Слика 3.6).

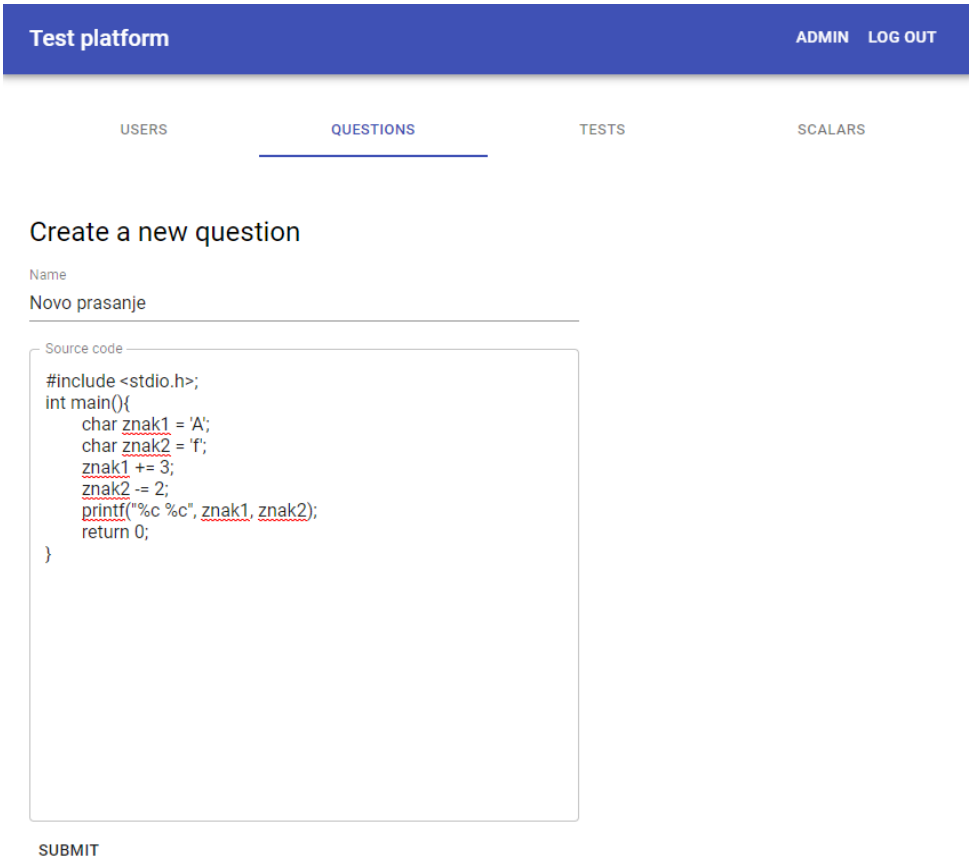


Слика 3.6: Поглед на панелот за управување со шаблоните за генерирање на нови прашања во системот

За секој шаблон е прикажано името, бројот на генерирани прашања, како и логичка вредност – дали е овозможено генерирање на нови прашања од тој

шаблон. Слично како кај панелот за управување со корисници, до секој шаблон постои копче “edit” за менување на истиот.

Копчето “create new question” овозможува внесување (креирање) на нов шаблон во апликацијата. При креирање на нов шаблон, се внесува името и програмскиот код за шаблонот (Слика 3.7). При кликнување на копчето “submit”, внесениот код се компајлира на серверот за да се утврди неговата валидност и за да се осигура дека истиот ќе може да се парсира и извршува.

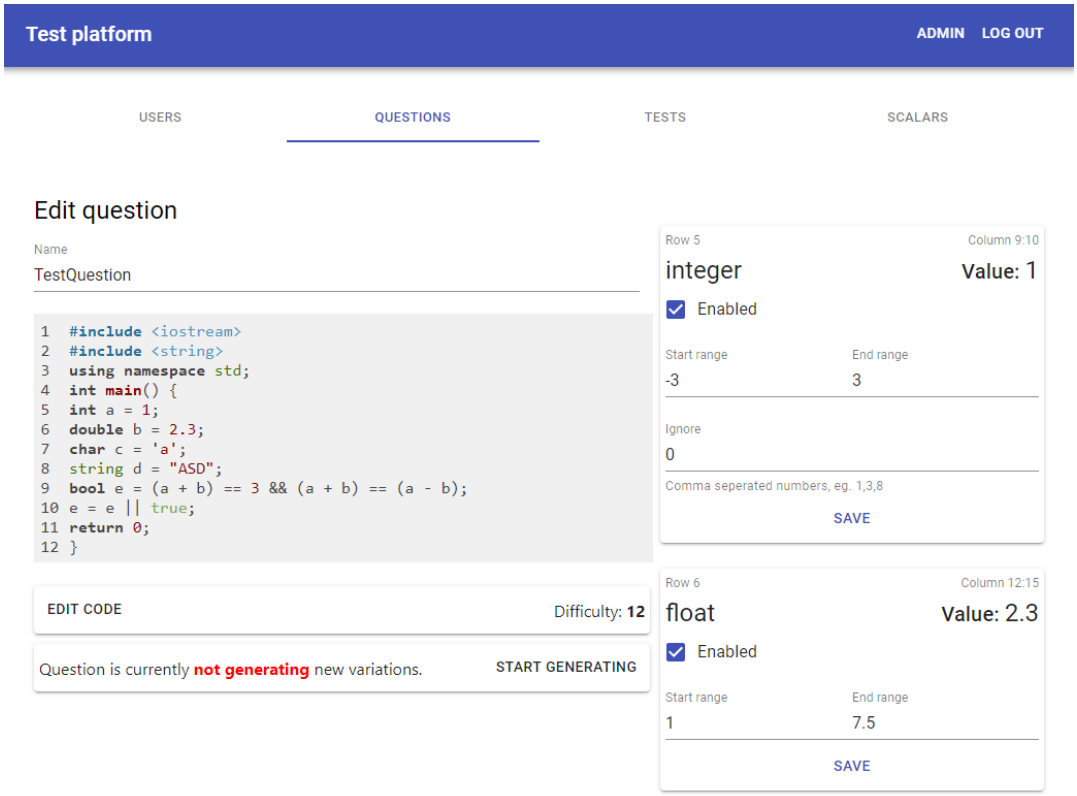


Слика 3.7: Внесување на шаблон во системот

Со кликнување на копчето “edit” за даден шаблон од листата на шаблони, се добива приказ преку којшто може целосно да се контролира генерирањето на нови прашања од шаблонот (Слика 3.8). Покрај можноста да се измени насловот и кодот на шаблонот, од десната страна се прикажани и сите детектирани локации од интерес во кодот над кои може да се извршат мутации за избраниот шаблон. За секоја локација од интерес се прикажува нејзината позиција во кодот (реден број на линија, како и редни броеви на позициите што се опфатени во линијата), вредноста што е присутна на таа локација во шаблонскиот код, како и нејзиниот тип (*integer* – за целобројни вредности, *float* – за броеви со подвижна точка, *char* – за знаци, *text* – за стрингови, *binary_op* / *unary_op* / *logical* – за оператори). Администраторот може да избере дали ќе се извршуваат или нема да се извршуваат мутации над секоја од локациите од

интерес, преку (не)селектирање на полето за избор “Enabled”. Во случај ако се избере да не се извршуваат мутации над дадена локација од интерес, тогаш таа локација ќе си ја зачува подразбирливата вредност – вредноста што е присутна на истата локација во шаблонскиот (почетниот) код.

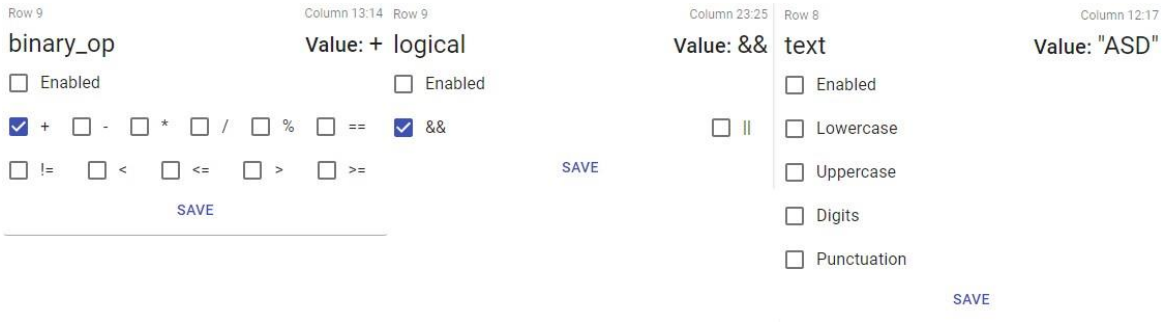
Како што е прикажано на Слика 3.8, во овој поглед администраторот може да изврши и конфигурирање на доменот на вредности за локациите од интерес детектирани во кодот на шаблонот (за оние локации за кои е селектирано полето “Enabled”). За нумерички локации од интерес, администраторот може да го конфигурира опсегот на вредности од којшто може да се пополни секоја локација. Уште повеќе, тој може експлицитно да специфицира множество од вредности коишто треба да се исклучат од овој опсег т.е. коишто не смеат да се појават на конкретната локација во ниту еден од новите кодови што ќе се генерираат од шаблонот. Ако внесените вредности се броеви со подвижна точка (англ. *floating point numbers*), тогаш генерираните вредности ќе содржат најмногу една цифра десно од децималната точка. За ненумерички локации од интерес, администраторот може да го конфигурира множеството од знаци од коешто ќе се пополнува локацијата. Множеството од дозволени знаци може да се конфигурира да ги вклучува (или исклучува) цифрите, малите букви од абецедата, големите букви од абецедата и специјалните знаци, преку (не)селектирање на соодветното поле за избор (Слика 3.9).



Слика 3.8: Преглед и конфигурирање на шаблон во системот

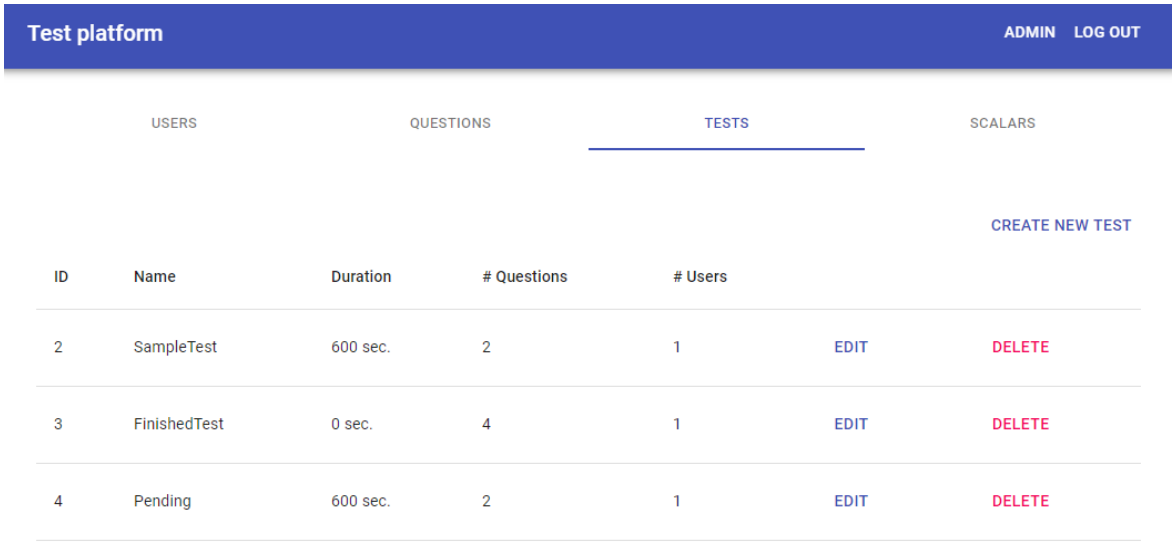
Конечно, ако станува збор за локација од интерес што содржи оператор, може да се конфигурира со кој(и) оператор(и) од понудена листа со дозволени оператори може да биде заменет соодветниот оператор присутен во шаблонскиот код. На Слика 3.9 се прикажани опциите за конфигурирање за различни типови на локации од интерес.

Во долниот дел на приказот – под кодот на шаблонот (Слика 3.8), се прикажуваат неговата сложеност (пресметана според новата предложена метрика), како и соодветни контроли за директно менување на наредбите во кодот и вклучување/исклучување на генерирањето на нови прашања со примена на дефинираните правила за мутирање над локациите од интерес.



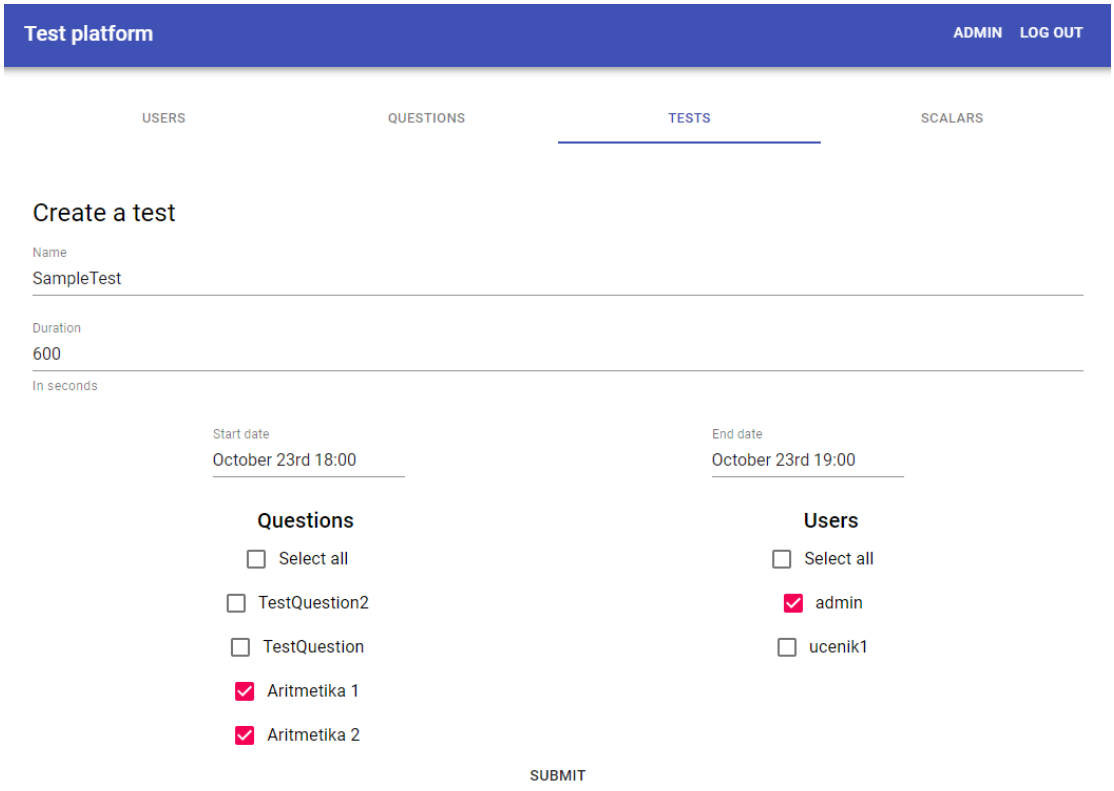
Слика 3.9: Конфигурирање на домен на вредности за различни видови на локации од интерес во шаблон

Слично како и кај претходно опишаните панели, при отворање на панелот за управување со тестови (панелот tests) прво се прикажува листа на сите тестови кои се внесени во апликацијата (Слика 3.10). За секој тест се прикажува: наслов, времетраење, број на прашања што ги содржи тестот, како и број на корисници кои имаат пристап до него.



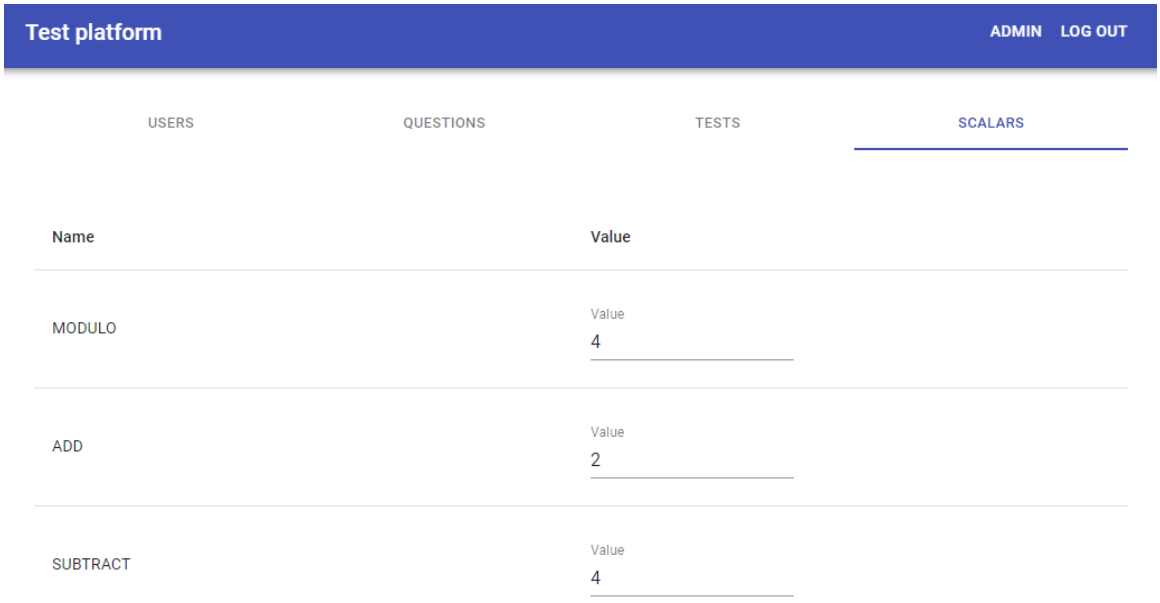
Слика 3.10: Поглед на панелот за управување со тестовите во системот

При кликнување на копчето “edit” заради менување на некој постоечки тест, или на копчето “create new test” заради креирање на нов тест, се отвора форма преку којашто може да се менуваат деталите за тестот (Слика 3.11). За секој тест може да се наведе наслов, времетраење (во секунди), почетен и завршен термин – кои го дефинираат временскиот период во којшто ќе биде достапен, листа на шаблони од кои ќе бидат вклучени прашања во тестот, како и листа на корисници со пристап до тестот. Почетниот и завршниот термин го ограничуваат отпочнувањето на тестот, но не и неговото времетраење.



Слика 3.11: Внесување на нов тест во системот

Конечно, последниот панел (панелот scalars) обезбедува интерфејс за конфигурирање на тежинските вредности за сите оператори/контролни наредби (Слика 3.12). Како што беше објаснето и претходно, овие тежини се користат при пресметката на сложеноста на кодовите содржани во прашањата. Интерфејсот дава табеларен преглед на сите овие тежини, кадешто секоја редица соодветствува на една тежина придружена на конкретен оператор или наредба. Како што може да се види од Слика 3.12, секоја тежина си има своја подразбирлива вредност. Тежинските вредности можат лесно да се менуваат со внесување на нови вредности во соодветните полиња во втората колона од табелата. Овде администраторот може исто така да ја менува и праговата вредност, со задавање на соодветна вредност за неа во табелата. На ваков начин, тој може да ја контролира дозволената девијација на сложеностите на генерираните програмски кодови од сложеноста на соодветниот шаблонски код.



Слика 3.12: Поглед на (дел од) панелот за менување на тежините на операторите/наредбите, потребни за пресметување на сложеноста на кодовите

3.3.4. Забележани проблеми при генерирањето на нови кодови и справување со нив

3.3.4.1 Бесконечен код и празен код

Да претпоставиме дека наставникот внел шаблонски код каков што е прикажан на Слика 3.13. Исто така, да претпоставиме дека при конфигурирањето на правилата за мутација на локациите од интерес, за локацијата на којашто се наоѓа константата ‘0’ (почетната вредност на променливата `i`) е дефиниран целобројниот опсег на вредности [0, 10], за бинарниот оператор ‘<’ е дефинирано дека може да се менува со оператор од листата: ‘<’, ‘<=’, ‘>’, ‘>=’, а за останатите локации од интерес е избрано дека нема да се извршуваат никакви мутации.

Еден од кодовите кој може да се добие со примена на вака дефинираните правила за мутација врз шаблонскиот код од Слика 3.13 е прикажан на Слика 3.14. Може да се забележи дека станува збор за „бесконечен“ код т.е. код чиешто извршување не може да заврши заради постоење на бесконечен циклус во него. Во почетната верзија на системот, ваквиот код предизвикуваше проблем – генерирачкиот процес не терминираше и не се добиваа никакви кодови односно прашања. Овој проблем сега е решен со задавање на терминирачки рок (англ. *timeout*) од 500ms при извршување на кодовите: оние кодови коишто не завршуваат со извршување во рамките на овој рок – се терминираат од страна на системот. Воедно, прашањата што содржат вакви кодови не се вклучуваат во листата на генерирани прашања т.е. се отфрлаат.

```
//   Initial code
#include <stdio.h>

int main()
{
    int i = 0;
    while (i < 5)
    {
        printf("\%d", i % 2);
        i++;
    }
    return 0;
}
```

Слика 3.13: Пример за шаблонски код којшто при соодветна конфигурација на правилата за мутација доведува до генерирање на „бесконечен“ код или „празен“ код

```
//   Infinite loop
#include <stdio.h>

int main()
{
    int i = 6;
    while (i > 5)
    {
        printf("\%d", i % 2);
        i++;
    }
    return 0;
}
```

Слика 3.14: „Бесконечен“ код генериран од шаблонскиот код на Слика 3.13

Друг проблематичен код кој може да биде генериран (со истата конфигурација на правилата за мутација) од шаблонскиот код на Слика 3.13 е оној на Слика 3.15. Овој код веројатно би ги збунил студентите, бидејќи станува збор за „празен“ код – код којшто не дава никаков излез. Уште поважно, оние студенти кои нема да го одговорат прашањето што ќе го содржи кодот – ќе имаат точен одговор!

Слично како со прашањата што содржат „бесконечен“ код, и ваквите прашања со „празни“ кодови не се вклучуваат во генерираните прашања.

3.3.4.2 Делење со нула

Да претпоставиме дека наставникот внел шаблонски код каков што е прикажан на Слика 3.16, како и дека при конфигурирањето на правилата за мутација на локациите од интерес, за локацијата на којашто се наоѓа константата ‘5’ (почетната вредност на променливата `i`) е дефиниран

целобројниот опсег на вредности [1, 5], а за останатите локации од интерес е избрано дека нема да се извршуваат никакви мутации.

```
//   Empty output
#include <stdio.h>

int main()
{
    int i = 10;
    while (i < 5)
    {
        printf("\%d", i % 2);
        i++;
    }
    return 0;
}
```

Слика 3.15: „Празен“ код генериран од шаблонскиот код на Слика 3.13

```
//   Initial div0 code
#include <stdio.h>

int main()
{
    int i=5;
    printf("%d", 10 / (i - 1));
    return 0;
}
```

Слика 3.16: Пример за шаблонски код којшто при соодветна конфигурација на правилата за мутација доведува до генерирање на проблематичен код заради „делење со нула“

Еден од петте различни кодови кој може да се добие со примена на вака дефинираните правила за мутација врз шаблонскиот код од Слика 3.16 е прикажан на Слика 3.17. Во овој случај, се јавува проблемот на „делење со нула“.

```
//   div0 code
#include <stdio.h>

int main()
{
    int i=1;
    printf("%d", 10 / (i - 1));
    return 0;
}
```

Слика 3.17: Проблематичен код (заради „делење со нула“) генериран од шаблонскиот код на Слика 3.16

За да не се оптоваруваат наставниците да внимаваат дали примената на правилата за мутација (што ги дефинирале) врз шаблонскиот код (којшто го внеле) евентуално би довела до генерирање на проблематичен код од ваков облик, во системот ова е решено преку детектирање на грешката којашто би ја „фрлил“ ваквиот код за време на извршувањето и отфрлање на прашањето што го содржи кодот.

3.3.5. Развоен процес на системот

3.3.5.1 Заден дел

Задниот дел (англ. *back-end*) на системот е изработен во програмскиот јазик Python, со користење на рамката Flask. Flask (2010) е микро рамка за развивање на веб апликации, која работи врз интерфејсот WSGI (англ. *Web Server Gateway Interface*) за Python веб апликации. Создадена е од страна на Armin Ronacher на 1 април 2010 година, како првоаприлска шега која се популаризирала, за подоцна да прерасне во сериозен проект. Предноста на Flask е минимализмот кој овозможува брзо и лесно креирање на едноставни веб апликации, но таа сепак дозволува и скалирање во комплексни апликации.

Со помош на рамката дефинирани се т.н. „погледи“ (англ. *views*) – функции кои како влезен параметар го добиваат HTTP барањето кое пристигнало до серверот, а како резултат можат да вратат одговор во форма на HTML или JSON. Во случај на враќање на статичен HTML, рамката нуди посебен погон за темплејти (англ. *template engine*), со цел полесно развивање на апликацијата и одвојување на дизајнот од логиката на апликацијата. За крај, секоја функција се мапира на посебна рута – униформен локатор на ресурси (URL) кој води кон таа функција. Задниот дел е имплементиран како REST API, така што секоја поглед-функција враќа одговор во JSON формат.

За складирање на податоците се користи PostgreSQL база на податоци, кон која се пристапува со помош на библиотеката SQLAlchemy. *SQLAlchemy* (2006) е релациски пресликувач на објекти (англ. *object relational mapper* – ORM) за Python апликации, кој овозможува полесно користење на SQL бази преку дефинирање на модели. За секој модел потребно е да се дефинираат податоците кои ќе ги чува. После дефинирањето на моделите, SQLAlchemy генерира SQL прашалници, кои пак генерираат соодветна табела во базата за секој модел.

3.3.5.2 Преден дел

Предниот дел (англ. *front-end*) е изработен во JavaScript со помош на библиотеката React. *React* (2013) е библиотека со отворен изворен код, развиена од страна на Facebook, којашто се користи за развивање на кориснички интерфејси. Оваа библиотека овозможува употреба на JSX (JavaScript XML)

синтакса во рамките на JavaScript, која е многу слична со стандардниот HTML. Со помош на JSX се креираат компоненти кои подоцна многу лесно можат да се употребат во JavaScript кодот.

Покрај React, во апликацијата се користат уште неколку JavaScript библиотеки:

- *React-Router* – библиотека за рутирање на едностранi апликации (англ. *single page applications* – SPA). Благодарение на оваа библиотека, целата апликација се вчитува во првото отворање, со сите компоненти и дизајни. Ова ја прави понатамошната комуникација помеѓу серверот и клиентот значително полесна, бидејќи се праќаат податоци единствено во JSON формат, наместо целосни HTML датотеки.
- *Redux* – библиотека за менаџирање на состојби кај комплексни апликации. Без Redux, секоја React компонента би требало да чува сопствени податоци. Додека за мали апликации ова работи беспрекорно, при изработка на покомплексни веб апликации се појавува поголема потреба за комуникација помеѓу компонентите и споделување на информации од нивните состојби. Наместо дефинирање на посебни методи за комуникација помеѓу компонентите, со Redux се креира една централна состојба за целата апликација, која ги чува сите потребни информации. Благодарение на Redux, секоја компонента низ целата апликација може да пристапи до кој било податок.
- *Material-UI* – библиотека за кориснички интерфејс базирана на Material дизајнот на Google. Налик на Bootstrap и Semantic UI, Material е CSS рамка која овозможува полесно дизајнирање на кориснички интерфејси со помош на HTML. Material-UI исто така нуди готови React компоненти за уште полесна интеграција.
- *Moment.js* – библиотека за управување со времиња и временски зони. Самиот јазик JavaScript нуди ограничена поддршка за времиња. Со помош на Moment.js, лесно може да се пресмета времето до почетокот или до крајот на даден тест во рамките на развиениот систем. Исто така, Moment.js овозможува серверот да работи со универзалното координирано време (англ. *Coordinated Universal Time* – UTC), независно од временските зони, но сепак времињата да се соодветно прикажани за секој корисник според неговиот уред.
- *React-cookie* – библиотека која овозможува пристап до колачињата (англ. *cookies*) кои се поставени на уредот на корисникот, како и поставување на нови колачиња.

3.3.5.3 Имплементација на процесирањето на C/C++ програмски кодови во системот

За потребите на развиената апликација, неопходни се следниве функционалности кои се однесуваат на процесирањето на програмски кодови:

- Компајлирање на C/C++ програмски кодови;
- Извршување на компајлирани C/C++ програмски кодови;
- Генерирање на апстрактно синтаксно дрво за даден C/C++ програмски код;
- Определување на бројот на извршувања на секоја линија за даден C/C++ програмски код.

За имплементација на повеќето од овие функционалности се користи *GNU Compiler Collection* (*GCC*, 1987) – колекција од алатки за компајлирање на повеќе програмски јазици. Колекцијата GCC доаѓа како вклучена во скоро секоја дистрибуција на оперативниот систем Linux, па според тоа таа е лесно достапна. Со помош на g++, којшто претставува компајлер за програмскиот јазик C/C++ од колекцијата GCC, се реализира компајлирањето на C/C++ кодови во рамките на апликацијата.

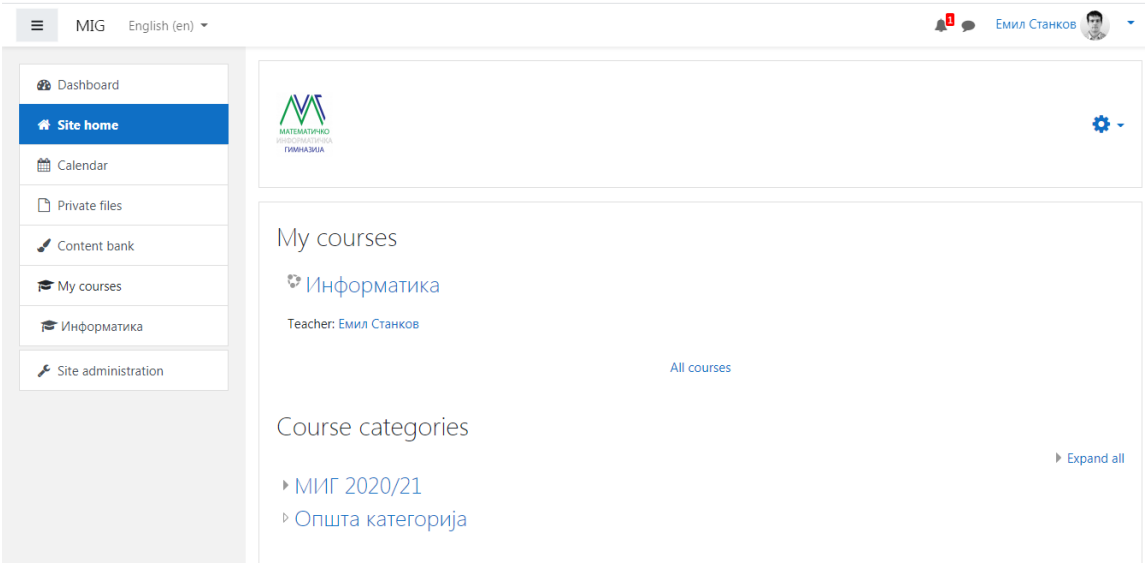
Покрај g++, од колекцијата GCC во апликацијата се користи и gcov, која претставува алатка за профилирање. Алатките за профилирање се алатки кои нудат некои информации за извршувањето на даден изворен код, како што се: кои линии од кодот се извршени, колку пати е извршена секоја линија, колку процесорско време е потрошено за секоја секција од кодот, итн. Во рамките на развиената апликација, gcov се користи за пресметување колку точно пати е извршена секоја линија при одредено извршување на даден C/C++ програмски код, при што резултатот се запишува во JSON формат, којшто е лесен за обработка.

За генерирање на апстрактно синтаксно дрво за даден C/C++ програмски код се користи *Clang* (2007), којшто е преден дел за LLVM компајлерите од семејството на програмскиот јазик C. Clang нуди интеграција со Python, што овозможува лесно генерирање и изминување на апстрактното синтаксно дрво за програмските кодови што се процесираат во рамките на апликацијата.

4. Moodle додаток за паметно генерирање на прашања за програмскиот јазик C/C++

4.1. Платформата за е-учење Moodle

Moodle (акроним за Модуларна Објектно Ориентирана Динамичка Околина за Учење, англ. *Modular Object Oriented Dynamic Learning Environment*) (Dougiamas и Taylor, 2002) е еден од најпопуларните и најраспространети системи за управување со учењето (англ. *Learning Management System* – LMS) низ целиот едукативен свет. Станува збор за бесплатен софтвер со отворен изворен код, којшто е напишан во програмскиот јазик PHP и дистрибуиран под условите на лиценцата GNU General Public License (Brandl, 2005). На Слика 4.1 е прикажан изглед на една едноставна Moodle страница.



Слика 4.1: Изглед на една Moodle страница

Moodle е изработен од страна на австралијанецот Martin Dougiamas, со цел да им се овозможи на професорите и луѓето кои се занимаваат со едукација лесно да креираат и управуваат онлајн курсеви (Xiao, 2020). Првата верзија е пуштена во употреба на 20 август 2002 година, а најнова стабилна верзија е Moodle 3.11, пуштена во употреба на 17 мај 2021 година (*Moodle Releases*, 2021). Денес постојат повеќе од 50 програмери кои се задолжени за унапредување и одржување на платформата, а нејзиниот развој постојано е потпомогнат од заедницата на програмери кои работат на софтвер со отворен изворен код (англ. *open-source programmers*).

Системот овозможува креирање на приватни веб сајтови со онлајн курсеви за секој наставен предмет, како и доделување на улоги како што се професор, асистент или студент. Курсевите можат да се користат за

поставување на материјали за учење, објавување на соопштенија за студенти, доделување задачи и домашни работи, евалуација на знаењето на студентите, следење на нивниот прогрес во учењето, комуникација помеѓу сите учесници на курсот, креирање на форуми за дискусија, итн. На корисниците со повисоки привилегии (најчесто – професори), Moodle исто така им овозможува да прикачуваат видео или аудио материјали, да водат дискусија во реално време, да доделуваат, собираат, прегледуваат и оценуваат задачи и да создаваат различни видови на тестови.

Moodle се користи секојдневно во повеќе од 200 држави, од страна на околу 275 милиони корисници ширум светот (*Moodle Statistics*, 2021). Според едно истражување направено во 2017-та година (Armenski, 2017), постојат голем број на средни училишта кои користат специјално развиени платформи или пак ја користат Moodle платформата за е-учење, што претставува добар податок за дигитализацијата на училиштата и користењето на дополнителни начини на пренесување на знаењето. Како илустрација, кај нас оваа популарна платформа се користи во средните училишта СОУ „Ристе Ристески Ричко“, СОУ „Ѓорче Петров“ и СОЕПТУ „Кузман Јосифоски – Питу“ од Прилеп, СОУ „Гимназија Добри Даскалов“ од Кавадарци, СОТУ „Ѓорѓи Наумов“ од Битола, ДСУ „Математичко-информатичка Гимназија“ од Скопје, и др. Од друга страна, огромен е бројот и на високообразовни институции коишто се долгогодишни корисници на Moodle. Според Hill (2017), трите најкористени системи за управување со учењето од страна на академските институции во САД се Blackboard, Canvas и Moodle, а слична е состојбата и со академските институции во Европа и Јужна Америка. Примери за вакви институции кои го користат Moodle кај нас се Европскиот Универзитет, Американскиот Универзитет на Европа – ФОН, Факултетот за електротехника и информациски технологии (ФЕИТ), како и Факултетот за информатички науки и компјутерско инженерство (ФИНКИ) при Универзитетот „Св. Кирил и Методиј“, во Скопје.

Moodle најчесто се користи за комбинирано учење, учење на далечина и останати проекти за електронско учење (е-учење), во основни и средни училишта, универзитети, како и на работни места (Costello, 2013; Horvat *et al.*, 2015). Во рамките на системот постои голем избор на функционалности коишто можат да се прилагодат за постигнување на одредени цели на учењето. Основното множество функционалности кое се нуди во Moodle најчесто ги задоволува потребите за изведување на најразлични курсеви, но постои и можност за проширување и прилагодување на платформата со користење на различни компоненти развиени од заедницата. Меѓу позначајните предности на овој систем е фактот дека овозможува персонализирање на изгледот и функциите преку изработка на нови или користење на претходно креирани теми (англ. *themes*) и додатоци (англ. *plugins*), но исто така и можноста за користење на платформата преку мобилен телефон.

Во рамките на оваа докторска дисертација беше разгледана и искористена токму опцијата за креирање на сопствен додаток кој овозможува проширување на основното множество функционалности. Целта на креирањето на овој додаток беше да се овозможи лесна и едноставна имплементација и примена на предложениот модел за паметно автоматско генерирање на прашања за воведни курсеви за програмирање, во процесот на проверка на знаење кај соодветни курсеви на сите образовни институции кои ја користат платформата Moodle.

4.2. Развивање на сопствени додатоци за Moodle

4.2.1. Типови на додатоци и нивно вклучување

Во рамките на акронимот „Moodle“, буквата ‘M’ означува модуларност. Како што беше објаснето и во претходното поглавје, доколку основните функционалности што ги нуди платформата Moodle не ги задоволуваат потребите на наставниот процес, постои можност за вклучување на претходно развиен или пак развивање и вклучување на нов сопствен додаток (англ. *plugin*), којшто би овозможил проширување на множеството функционалности според потребите.

Постојат различни типови на додатоци, а постојано се развиваат и нови заради додавање на нови функционалности. Некои од почесто користените типови на Moodle додатоци се прикажани во Табела 4.1, а комплетната актуелна листа на типови на додатоци може да се погледне на https://moodle.org/plugins/ (*Moodle Plugins*, 2021).

Тип на додаток	Опис	Верзија на Moodle
Activity modules	Нудат активности на Moodle. Примери: Forum, Quiz, Assignment.	1.0+
Blocks	Информациски прикази или алатки кои може да се поставуваат на различни позиции во дадена Moodle страна.	2.0+
Question types	Различни типови на прашања кои можат да се користат во квизови или други активности.	1.6+
Gradebook export	Овозможуваат експортирање на оценките од дневникот со оценки за даден курс во различни формати	1.6+
Enrolment plugins	Овозможуваат различни начини за контролирање на вклучувањето на учесници во одреден курс.	2.0+

Табела 4.1: Некои почесто користени типови на Moodle додатоци

При преземање на изворниот код на Moodle, може да се забележи дека истиот е организиран во голем број на датотеки. Кодот се состои од јадро (англ. *Moodle core*), библиотеки од надворешни извори и додатоци. Датотечната организација на прв поглед може да биде збунувачка, но секој тип на додаток е сместен на своја локација, па така вклучувањето (импортирањето) на новокрерираните сопствени додатоци во Moodle се врши со копирање на новиот код во соодветната локација, според типот.

Типот на додаток којшто е од интерес за оваа докторска дисертација е Question types.

4.2.2. Типот на додаток Question types

Со стапувањето на сцена на верзијата 2.1 од Moodle, беше претставен и новиот погон за прашања (англ. *question engine*) кој функционира заедно со типовите на прашања (англ. *Question types*) при креирањето на квизови и други активности кои го користат овој тип на додаток. Платформата подразбирливо доаѓа со повеќе типови на прашања, меѓу кои: „точно/неточно“ прашања (англ. *true/false questions*), прашања со краток одговор (англ. *short answer questions*), есејски прашања (англ. *essay questions*), прашања со избор од повеќе можни одговори (англ. *multiple choice questions*), и др. Исто така, Moodle дозволува и користење на веќе изработени (готови) типови на прашања од надворешни извори, како што се на пример: „влечење и пуштање“ во текст (англ. *drag and drop into text*), избор на зборови што недостасуваат (англ. *select missing words*), прашања со пресметување и внесување на нумерички одговор (англ. *variable numeric question type*), итн. Конечно, системот овозможува и креирање на нов, сопствен тип на прашање кој ќе одговара на потребите на наставниот процес за курсот на којшто ќе се користи. Типовите на прашања (впрочем како и целата платформа) се напишани во програмскиот јазик PHP, па креирањето на нов тип на прашање побарува познавање на овој програмски јазик.

При изработката на додаток од тип Question types, потребно е на Moodle да му се достават повеќе датотеки (претежно со екстензија .php) коишто овозможуваат платформата да го препознае и додаде соодветниот нов тип на прашање:

- *edit_QTYPENAME_form.php* – код којшто дефинира како ќе изгледа прозорецот кадешто се поставуваат својствата на секое прашање;
- *questiontype.php* – дава дефиниција на класата `qtype_mytype`, којашто задолжително мора да ја проширува класата `question_type`;
- *question.php* – дава дефиниција на класата `qtype_mytype_question`, којашто задолжително мора да ја проширува класата `question_definition`;

- *renderer.php* – содржи дефиниција за класата `qtype_mytype_renderer`;
- *tests/...* – содржи тестови за соодветниот тип на прашање. Строго се препорачува пишување на тестови заради осигурување на безбедно функционирање на додатокот, како и за безбедноста на целиот систем кадешто е поставена платформата;
- *lang/en/qtype_myqtype.php* – текстуални низи или зборови за овој тип на прашање, на англиски јазик. Може, исто така, да се користат и други јазици;
- *db/install.xml*, *db/upgrade.php* – се користат за креирање на бази на податоци кои ќе ги користи соодветниот тип на прашање.

Како што беше образложено и претходно, за вклучување (импортирање) на нов додаток, потребно е изворниот код да се додаде во локацијата којашто е наменета за соодветниот тип на додаток. Постојат три начини на вклучување на нов додаток. Првите два овозможуваат додавање преку интерфејсот на Moodle, односно преку директориумот на Moodle додатоци (англ. *Moodle plugins directory*), или преку прикачување на ZIP архива која го содржи изворниот код. Третиот начин е рачно вклучување на изворниот код на додатокот директно на Moodle серверот.

Во рамките на оваа дисертација беше применет третиот начин – рачно вклучување на сопствено развиен додаток од типот Question types. За таа цел беше потребно поставување на неговиот изворен код на соодветна локација на серверот, која за додатоци од тип Question types има ваков изглед: "/path/to/moodle/question/type/". По поставувањето на изворниот код, Moodle веднаш автоматски го препознава додатокот и овозможува негово користење, без потреба од дополнителни инсталации.

4.3. CodeCPP – нов додаток за паметно автоматско генерирање на прашања што содржат програмски код

4.3.1. Опис на додатокот

Развиениот Moodle додаток, именуван CodeCPP, претставува систем што овозможува паметно автоматско генерирање на прашања што содржат програмски код во согласност со моделот предложен во Глава 3, а кој едноставно се интегрира во популарниот систем за управување со учењето Moodle.

Најчесто еден веб базиран систем, без разлика на функционалноста што ја нуди, би требало да содржи дел којшто е задолжен за креирање на нови

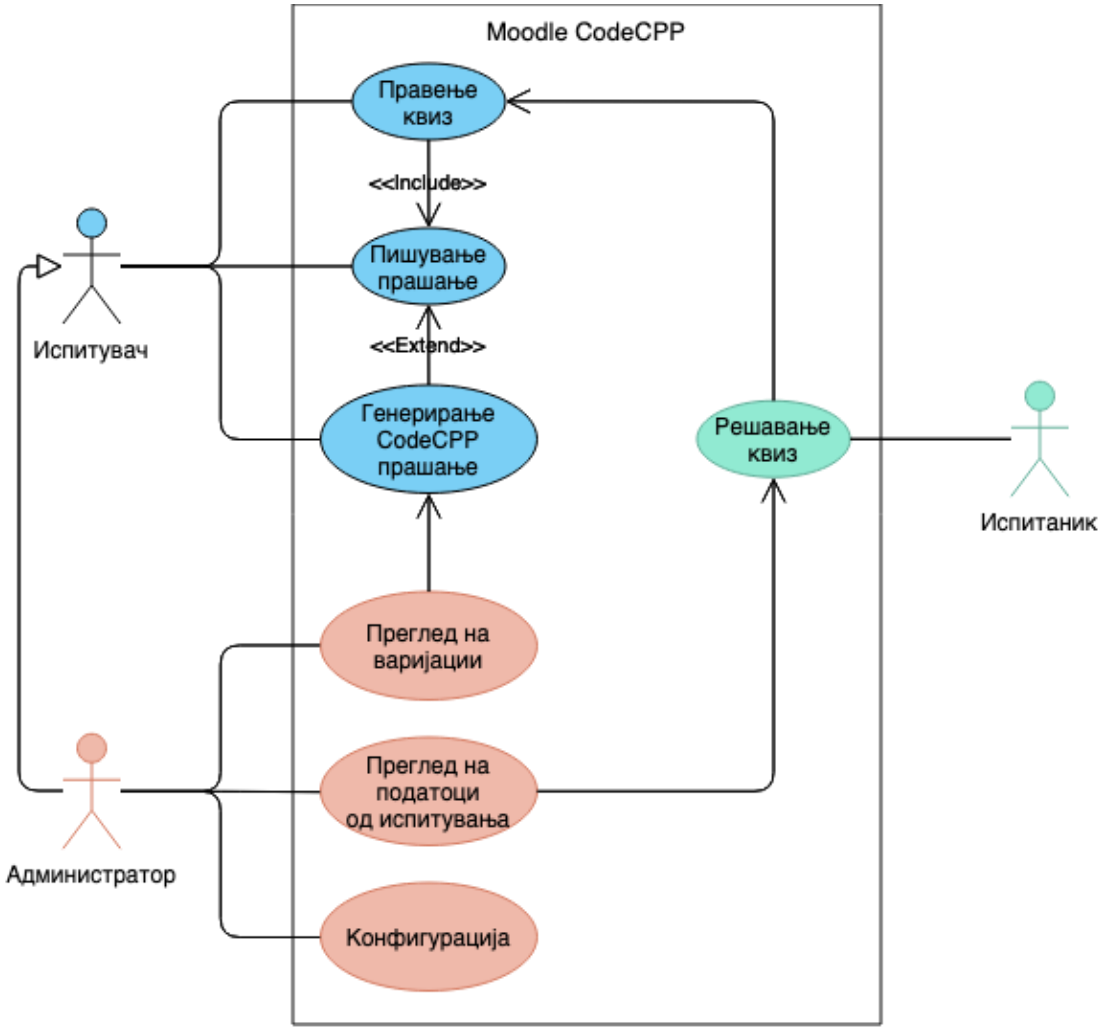
корисници, доделување на улога за секој корисник, како и дел за автентикација и авторизација. Во конкретниов случај, со оглед на тоа дека се изведуваше надополнување на веќе постоечки систем (Moodle) со нов додаток, предноста беше во тоа што немаше потреба од развивање на вакви делови бидејќи Moodle платформата секако содржи функционални компоненти задолжени за овие активности, кои постојано и секојдневно се тестирани на милиони корисници. Но, од друга страна, недостаток на ваквиот пристап на надополнување на веќе постоечки систем е фактот дека при практичната реализација потребно е да се прилагодат замислените идеи кон можностите кои ги нуди самиот систем што се надополнува.

Moodle е комплексен систем во кој се дефинирани повеќе различни улоги на корисници: наставник со право на менување на содржините на даден курс (англ. *Teacher*); наставник без право на менување на содржините на даден курс (англ. *Non-editing teacher*); студент (англ. *Student*); креатор на курс (англ. *Course creator*) – кој може да креира курсеви; менаџер (англ. *Manager*) – кој може да пристапува и менува курсеви, но вообичаено не учествува во нив; администратор (англ. *Administrator*) – корисник кој има највисоки привилегии и може да прави конфигурација на целиот систем.

За потребите на развиениот (пот)систем CodeCPP воведени се 3 различни улоги. Тоа се: Испитувач (наставник) – корисник којшто може да креира и закажува Moodle квизови, како и да креира и/или вклучува прашања во даден квиз (меѓу кои и прашања од тип CodeCPP); Испитаник (студент) – корисник кој може да се тестира (да решава квиз до којшто му е дозволен пристап) и да прегледува резултати од тестирањата во кои учествувал; и Администратор – корисник кој има привилегии како и Испитувачот, но може да извршува и некои други активности, како конфигурирање на некои параметри за функционирањето на CodeCPP, преглед на податоци од различни тестирања (каде се користеле прашања од тип CodeCPP) и сл.

Иницијалната идеја беше Испитувач и Администратор да бидат еден ист корисник, но поради ограничувањата наметнати од Moodle беше неопходно да се воведе Администратор како засебна улога. На Слика 4.2 е прикажан use-case дијаграмот на системот.

Во продолжение следува краток опис на главната функционалност на CodeCPP, а во следното поглавје ќе бидат опишани деталите за практичната примена на системот, од перспектива на секоја од трите улоги на корисници.



Слика 4.2: Use-case дијаграм на (пот)системот CodeCPP интегриран во системот Moodle

Слично како кај софтверскиот систем опишан во Глава 3, како прв чекор, наставникот креира прашање чиј составен дел е програмски код – шаблон за генерирање на нови прашања. Шаблонскиот код прво се компајлира за да се провери дали претставува валиден C/C++ код. Потоа, истиот се парсира со цел да се детектираат сите локации од интерес над кои може да се извршат модификации (мутации). Наставникот има преглед на можните мутации за секој шаблон и истите може да ги вклучи/исклучи, како и да им ги конфигурира соодветните правила. По завршување со конфигурацијата, наставникот го зачувува прашањето (шаблонот) што го содржи шаблонскиот код, а со тоа отпочнува генерирањето на нови прашања чии кодови се добиваат со примена на дефинираните правила за мутација врз шаблонскиот код. Новите програмски кодови добиени на ваков начин од почетниот шаблонски код ќе бидат нарекувани **кодни варијации на шаблонскиот код**. На крајот од процесот, се добиваат одреден број на нови прашања (верзии на шаблонот), при што секое од нив содржи точно по една кодна варијација добиена од шаблонскиот код.

Самиот процес на автоматско генерирање на прашања е идентичен со оној кај софтверскиот систем претставен во Глава 3. Секако, како и кај тој систем, и овде се зема во предвид сложеноста на сите програмски кодови (пресметана според предложената нова метрика) и се врши контрола на девијацијата на сложеностите на кодните варијации од онаа на соодветниот шаблонски код, преку користење на соодветна прагова вредност. Со други зборови, CodeCPP исто така овозможува генерирање на прашања што содржат кодови со конзистентна сложеност.

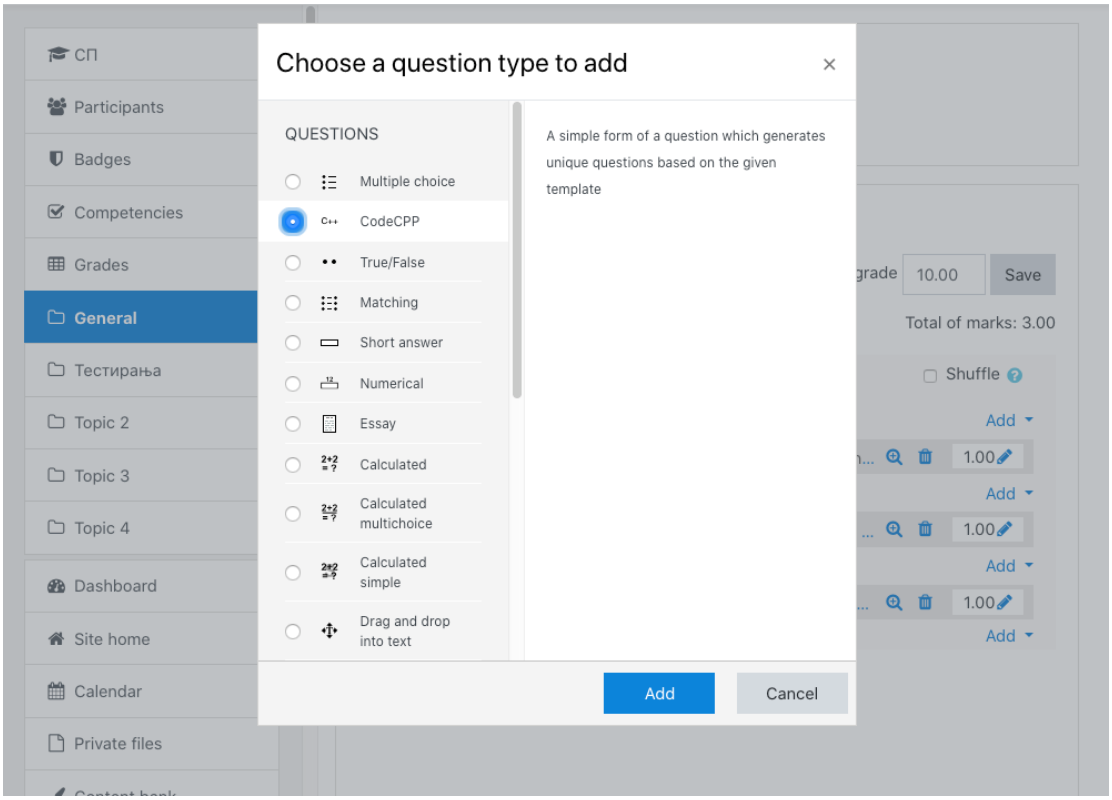
При решавање на даден квиз што содржи прашања од тип CodeCPP, секој студент добива по едно од генерираните нови прашања (една од кодните варијации) од секој шаблон (шаблонски код) што е вклучен во квизот, при што за секое прашање се бара да се внесе одговор – излезот од соодветниот код содржан во прашањето. Редоследот на одговарање на прашањата (според шаблонот од којшто се генерирани) е задолжително ист за сите студенти (ова е неопходно за да може да се измери времето што му било потребно за решавање на секое прашање на секој од студентите – учесници во квизот). Евалуацијата на прашањата се врши со едноставна споредба на одговорот што го внел студентот со излезот што се добива при извршувањето на кодот.

4.3.2. Употреба

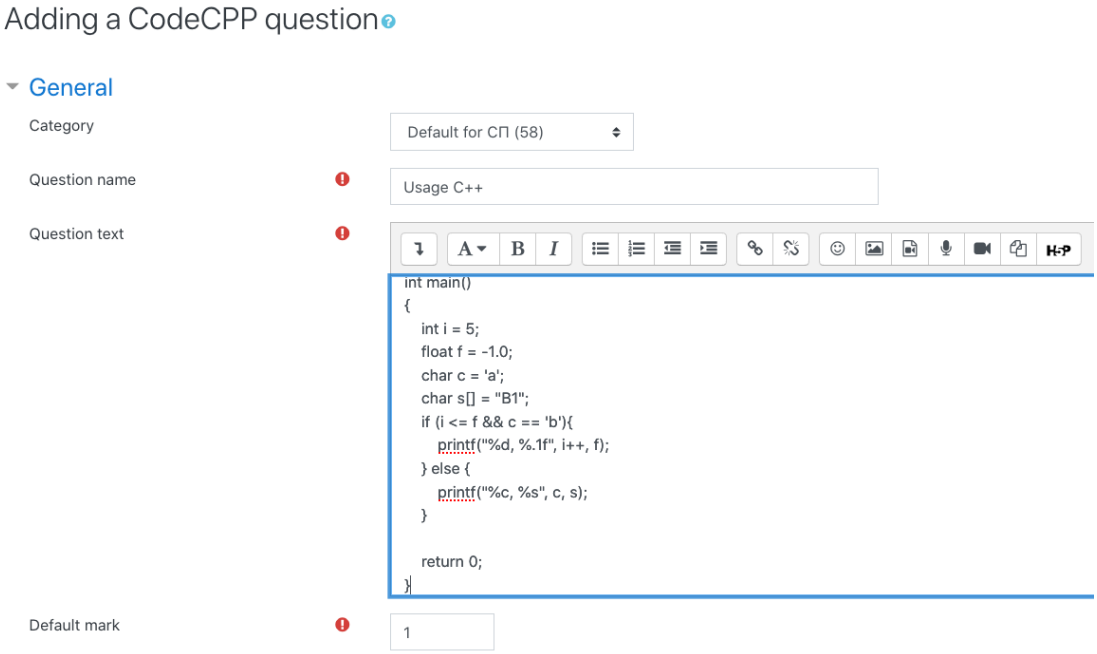
4.3.2.1 Перспектива на испитувач (наставник)

При креирање на квиз во Moodle заради проверка на знаење на студентите на одреден курс, потребно е наставникот да додаде прашања кои ќе го сочинуваат квизот. На Слика 4.3 е прикажан дијалог прозорецот којшто се појавува при додавање на ново прашање во даден квиз. Во понудените типови на прашања, може да се забележи дека е присутен и новокреираниот тип – CodeCPP.

По избирањето на овој тип прашање и кликнување на копчето “Add”, Moodle го пренасочува наставникот кон нов прозорец (Слика 4.4), кадешто од него се бара да ја избере категоријата на прашањето што го внесува (банката со прашања кај секој Moodle курс е хиерархиски организирана во категории формирани по желба на наставникот), да го внесе името на прашањето, како и шаблонскиот (почетен) код од којшто треба да се генерираат кодни варијации за тоа прашање. Шаблонскиот код се внесува во полето “Question text”, при што наставникот треба да внимава кодот да биде валиден (односно да не содржи синтаксни грешки). Во случај на внесување на невалиден код, истиот ќе биде отфрлен.



Слика 4.3: Избор на тип на прашање при креирање на квиз во Moodle

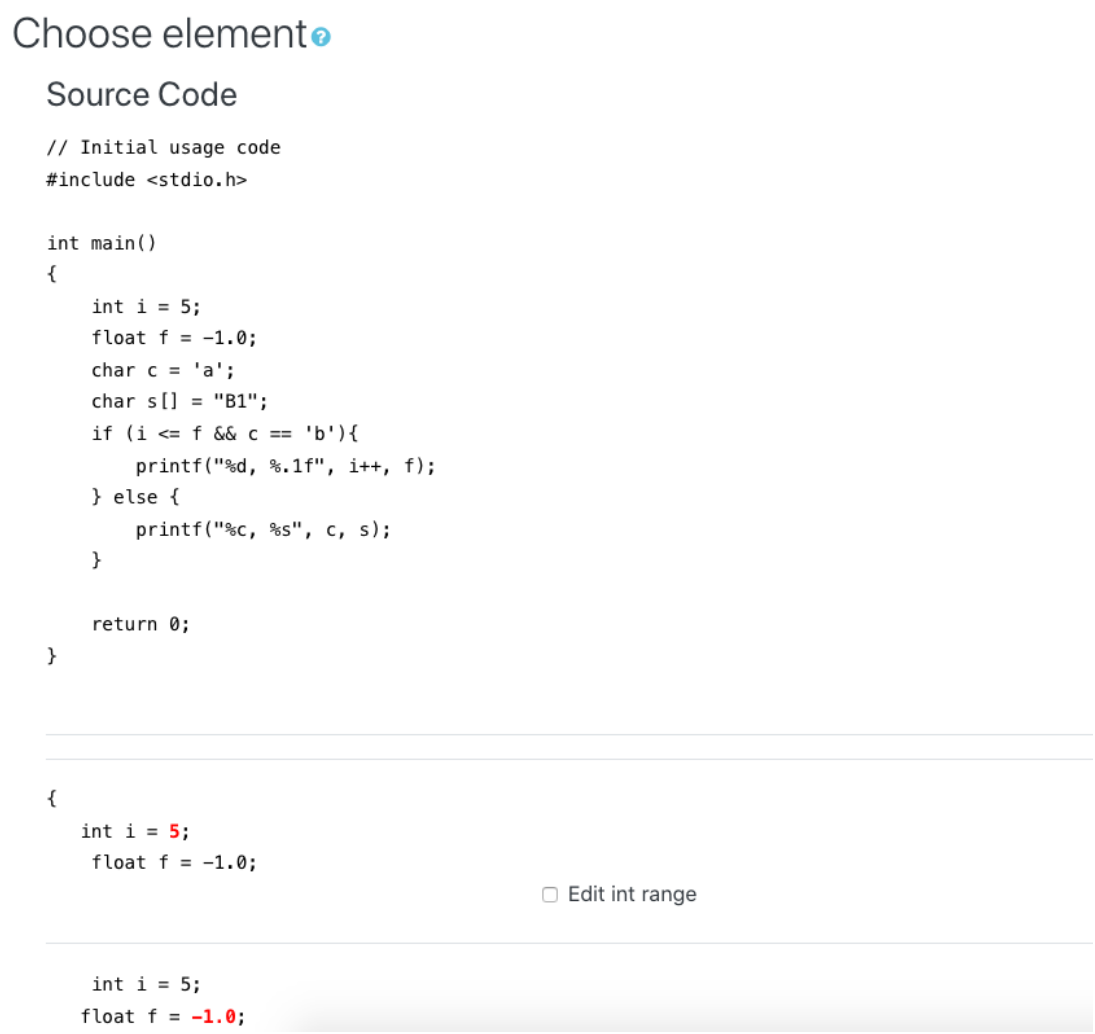


Слика 4.4: Внесување на почетен код – шаблон за генерирање кодни варијации за прашање од тип CodeCPP

Откако наставникот ќе заврши со внесување и ќе кликне на копчето “Save changes”, шаблонот се зачувува во системот и започнува процесот на генерирање и парсирање на апстрактното синтаксно дрво за истиот. Со ова се

извршува подготовка на податоците кои ќе му се прикажат на наставникот во следниот прозорец (Слика 4.5).

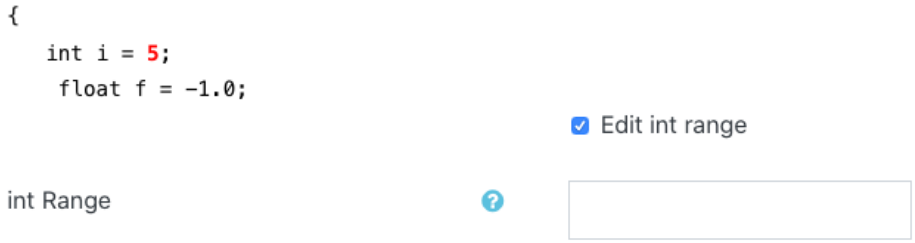
Како што може да се види од Слика 4.5, во овој чекор наставникот може да направи преглед на шаблонскиот код, како и преглед на сите можни локации од интерес во кодот над кои може да се извршат мутации за соодветниот шаблон. За секоја локација од интерес се прикажува и контекстот во којшто се појавува: одреден број на линии од кодот пред и после линијата во која се наоѓа таа локација (бројот на линии може да се конфигурира; подразбирливата вредност е 1). Наставникот може да избере дали ќе се извршуваат или нема да се извршуваат мутации над секоја од локациите од интерес, преку (не)селектирање на соодветното поле за избор "Edit". Во случај ако се избере да не се извршуваат мутации над дадена локација од интерес, тогаш таа локација ќе си ја зачува подразбирливата вредност – вредноста што е присутна на истата локација во шаблонскиот (почетниот) код.



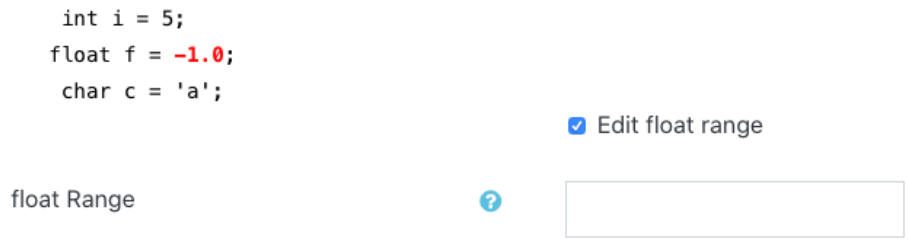
Слика 4.5: Приказ на шаблонски код, со (дел од) детектираните локации од интерес (означени со црвена боја)

Слично како кај системот опишан во Глава 3, во овој поглед наставникот може да изврши и конфигурирање на доменот на вредности за локациите од интерес детектирани во кодот на шаблонот (за оние локации за кои е селектирано полето “Edit”). За нумерички локации од интерес, наставникот може да го конфигурира опсегот на вредности од којшто може да се пополни секоја локација. Опсегот се дефинира во соодветно текстуално поле. Притоа, тој може да се зададе во обликот “`pochetok : kraj`” – што означува сите целобројни вредности од интервалот [`pochetok`, `kraj`], да се зададе со наведување на дозволените нумерички вредности меѓусебно разделени со запирка, или пак со комбинација од двата начина. На пример, “1 : 5, 7” означува дека доменот на дозволени вредности е {1, 2, 3, 4, 5, 7}. Во случај ако внесените вредности за `pochetok` и `kraj` се броеви со подвижна точка (англ. *floating point numbers*), тогаш “`pochetok : kraj`” ќе значи избор на униформно распределени случајни вредности од интервалот (`pochetok`, `kraj`). Дополнително, наставникот може експлицитно да специфицира вредности коишто треба да се исклучат од опсегот т.е. коишто не смеат да се појават на конкретната локација во ниту еден од новите кодови што ќе се генерираат од шаблонот. Ова се прави со користење на знакот ‘^’. Според тоа, “1 : 5, 7, ^3” го означува доменот {1, 2, 4, 5, 7}.

На Слика 4.6 и Слика 4.7 се прикажани полињата за конфигурирање на опсег кај двата вида на нумерички локации од интерес (“int” и “float”).



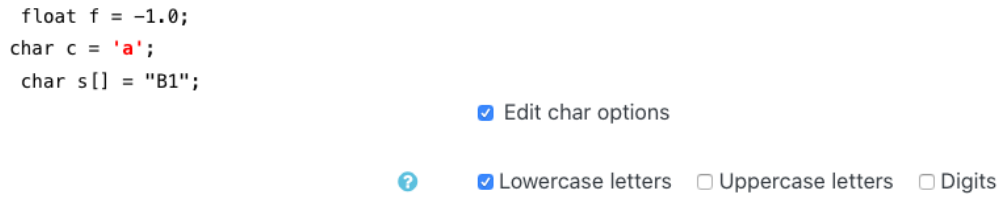
Слика 4.6: Дефинирање на опсег за целобројни локации од интерес



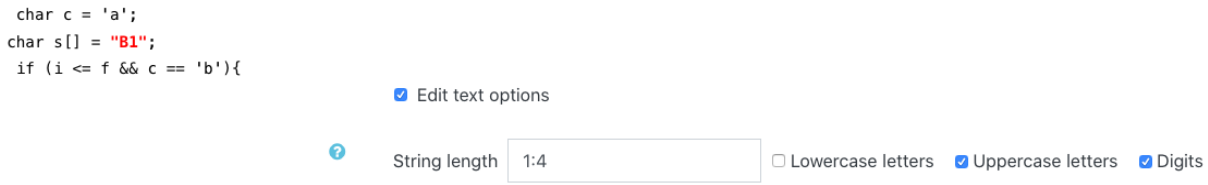
Слика 4.7: Дефинирање на опсег за “float” локации од интерес

За локации од интерес што содржат единечен знак (англ. *char*), наставникот може да го конфигурира множеството од знаци од коешто ќе се пополнува локацијата. Множеството од дозволени знаци може да се конфигурира да ги вклучува (или исклучува) цифрите, малите и големите букви

од абецедата, преку (не)селектирање на соодветното поле за избор (Слика 4.8). Од друга страна, за локации од интерес што содржат текстуална низа (англ. *string*) (Слика 4.9), може да се конфигурира должината на низата, како и множеството од знаци од коешто ќе се избира секој елемент од низата при пополнување на соодветната локација од интерес (секој елемент од низата се пополнува како да е посебна локација од интерес од тип "char"). За должината на низата, повторно се специфицира опсег од обликот "`pochetok` : `kraj`", којшто дефинира целоброен интервал [`pochetok`, `kraj`] од кој се избира случаен цел број за должината.

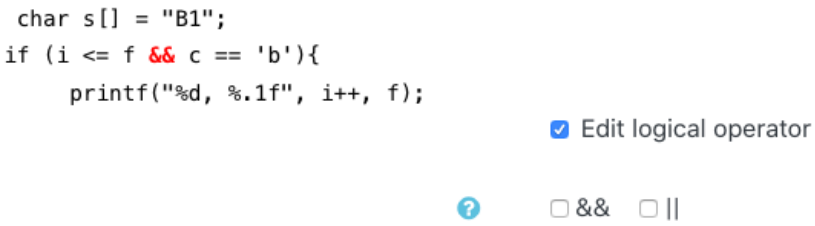


Слика 4.8: Конфигурирање на опции за локации од интерес од типот "char"

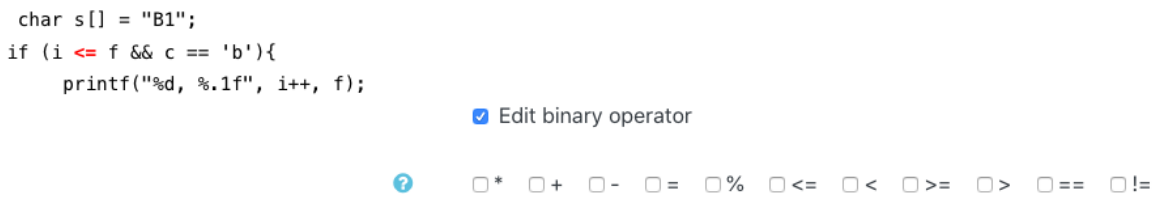


Слика 4.9: Конфигурирање на опции за локации од интерес од типот "string"

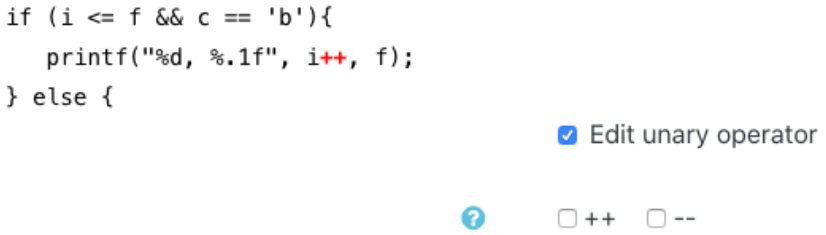
Конечно, ако станува збор за локација од интерес што содржи оператор, може да се конфигурира со кој(и) оператор(и) од понудена листа со дозволени оператори може да биде заменет соодветниот оператор присутен во шаблонскиот код. На Слика 4.10, Слика 4.11 и Слика 4.12 прикажани се опциите за конфигурирање на локации од интерес што содржат различни типови на оператори: логички, унарен (инкремент/декремент) и бинарен (аритметички или релациски), соодветно.



Слика 4.10: Конфигурирање на опции за локација од интерес што содржи логички оператор

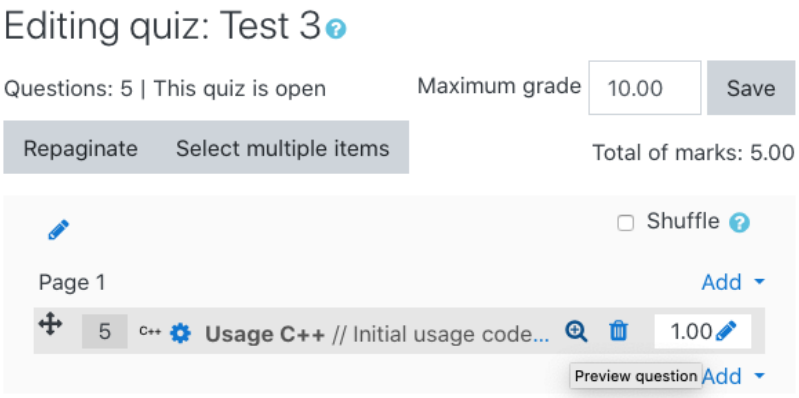


Слика 4.11: Конфигурирање на опции за локација од интерес што содржи бинарен (аритметички или релациски) оператор



Слика 4.12: Конфигурирање на опции за локација од интерес што содржи унарен оператор (инкремент/декремент)

Откако ќе заврши со конфигурацијата на правилата за мутација за локациите од интерес во шаблонот, наставникот треба да кликне на копчето "Save changes", со што отпочнува процесот на автоматско генерирање на нови прашања кои ќе содржат кодни варијации формирани од внесениот шаблонски код. На крајот од овој процес генерираните прашања ќе бидат зачувани во базата со прашања. Слика 4.13 дава приказ на погледот што го добива наставникот по додавање на прашање од типот CodeCPP во Moodle квиз.

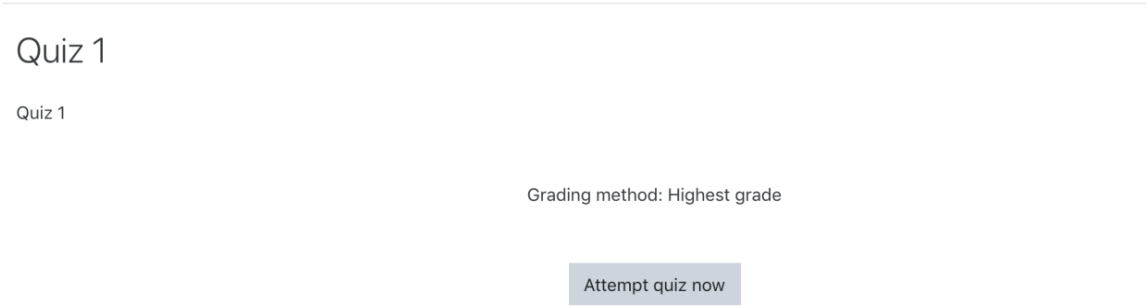


Слика 4.13: Приказ на квиз во Moodle во кој е додадено прашање од типот CodeCPP

4.3.2.2 Перспектива на испитаник (студент)

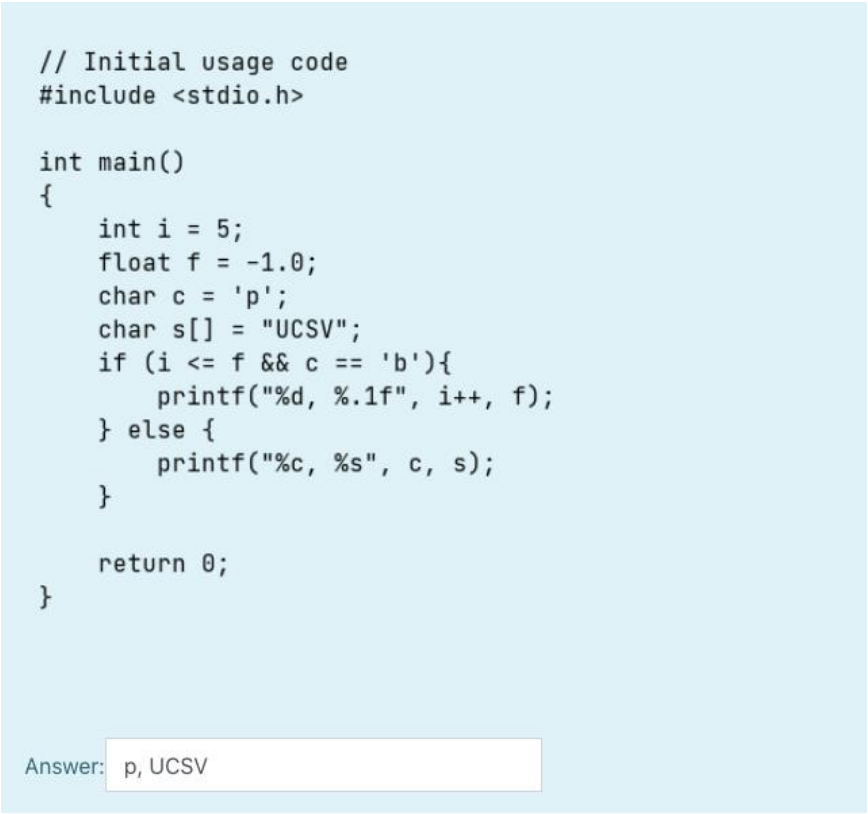
Ако на Moodle курсот, на којшто е учесник даден студент, е закажан квиз од страна на наставник за определен термин, студентот ќе може да го пристапи квизот во тој термин и да започне со тестирање. За таа цел, тој треба прво да го

пристапи Moodle сајтот на курсот, а потоа да го пристапи и квизот (со кликнување на хиперврската со неговото име). За да отпочне со тестирање, потребно е во овој поглед да кликне на копчето "Attempt quiz now" (Слика 4.14).



Слика 4.14: Копче за отпочнување на квиз во Moodle (перспектива на студент)

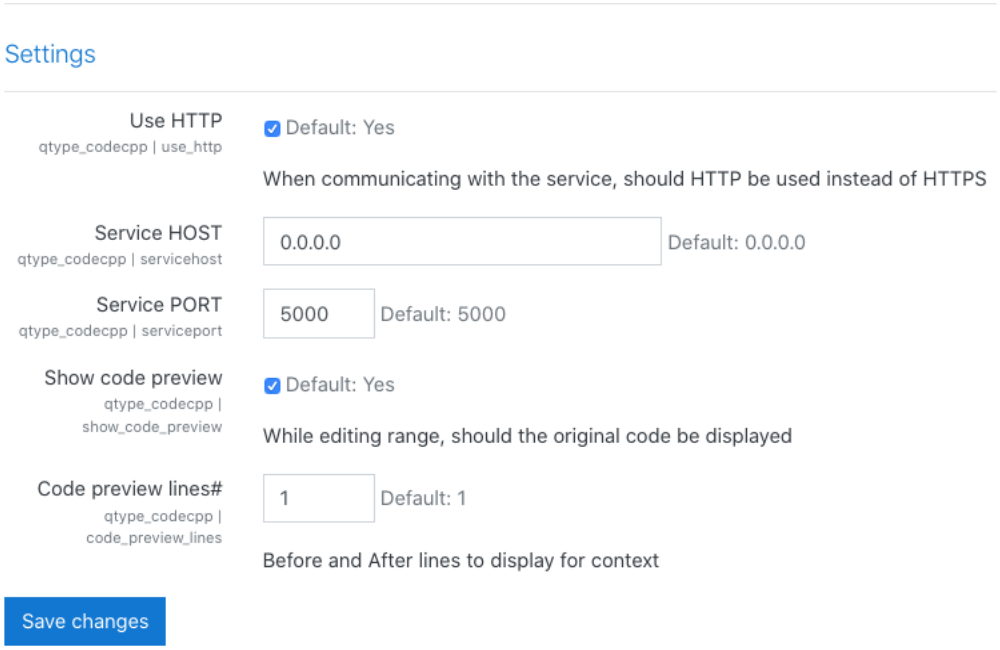
Во текот на тестирањето, за секое прашање (шаблон) од тип CodeCPP што е поставено во квизот од страна на наставникот, студентот ќе добие прашање коешто ќе содржи една од генерираните кодни варијации од соодветниот шаблонски код внесен за тоа прашање. За секое вакво прашање се прикажува програмскиот код (кодната варијација), заедно со текстуално поле каде студентот може да го запише својот одговор (Слика 4.15).



Слика 4.15: Поглед на прашање од тип CodeCPP од перспектива на студент, за време на тестирање

4.3.2.3 Перспектива на администратор

Една од основните привилегии коишто ги има исклучиво администраторот е можноста за конфигурација на некои генерални параметри во врска со додатокот CodeCPP. На Слика 4.16 е даден поглед на страницата за конфигурација на CodeCPP, којашто е достапна само за администраторот.



Слика 4.16: Поглед на администраторската страница за конфигурација на додатокот CodeCPP на Moodle

Опции коишто можат да се конфигурираат се следниве:

- *Use HTTP*, *Service HOST* и *Service PORT* – се однесуваат на локацијата кадешто е поставен Python/Flask веб сервисот потребен за процесирањето и генерирањето на програмските кодови (поглавје 4.3.3);
- *Show code preview* – поле за избор кое одредува дали ќе се прикаже или нема да се прикаже целиот шаблонски код, во погледот за преглед на детектираните локации од интерес за внесен шаблон (Слика 4.5);
- *Code preview lines#* – поле за дефинирање на приказот на контекстот во кој се појавува секоја детектирана локација од интерес во шаблон: бројот на линии пред и после линијата што ја содржи локацијата од интерес, кои треба да бидат прикажани во погледот за преглед на детектираните локации од интерес за внесен шаблон (Слика 4.5).

Друга значајна привилегија којашто ја поседува единствено администраторот е опцијата за преглед на сите генерирани прашања од тип CodeCPP присутни во базата со прашања. Прашањата во овој поглед (Слика 4.17) се организирани на таков начин што сите прашања (коишто содржат кодни

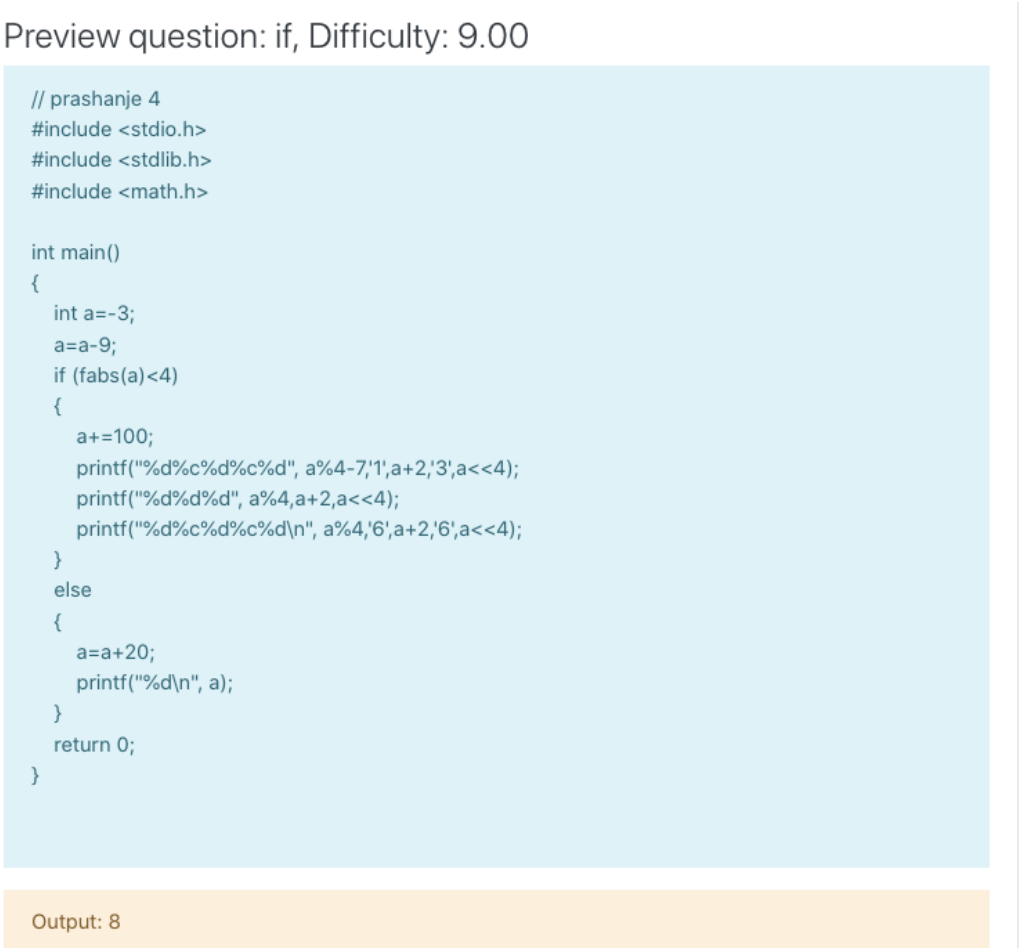
варијации) генерирани од еден ист шаблон се прикажуваат како засебна категорија. За секоја категорија е прикажано нејзиното име (тоа е всушност текстот што наставникот го внел како име на прашањето при внесување на шаблонот), како и хиперврски до прашањата што припаѓаат во неа (означени со: варијација 1, варијација 2, итн.). Исто така, до секое прашање прикажана е и пресметаната сложеност на кодот.

Како илустрација, на Слика 4.17 се прикажани 4 вакви категории на прашања, именувани “if”, “while”, “switch” и “operations”, при што секоја категорија содржи по 3 прашања генерирани од соодветниот шаблон.

Слика 4.17: Поглед на сите генерирани прашања од тип CodeCPP (кодни варијации на различни шаблони) во базата со прашања, достапен само за администраторот

Со кликнување на хиперврската за дадено прашање се отвора скок-прозорец (англ. *pop-up window*), во кој се прикажува кодот на прашањето, заедно со излезот (Слика 4.18).

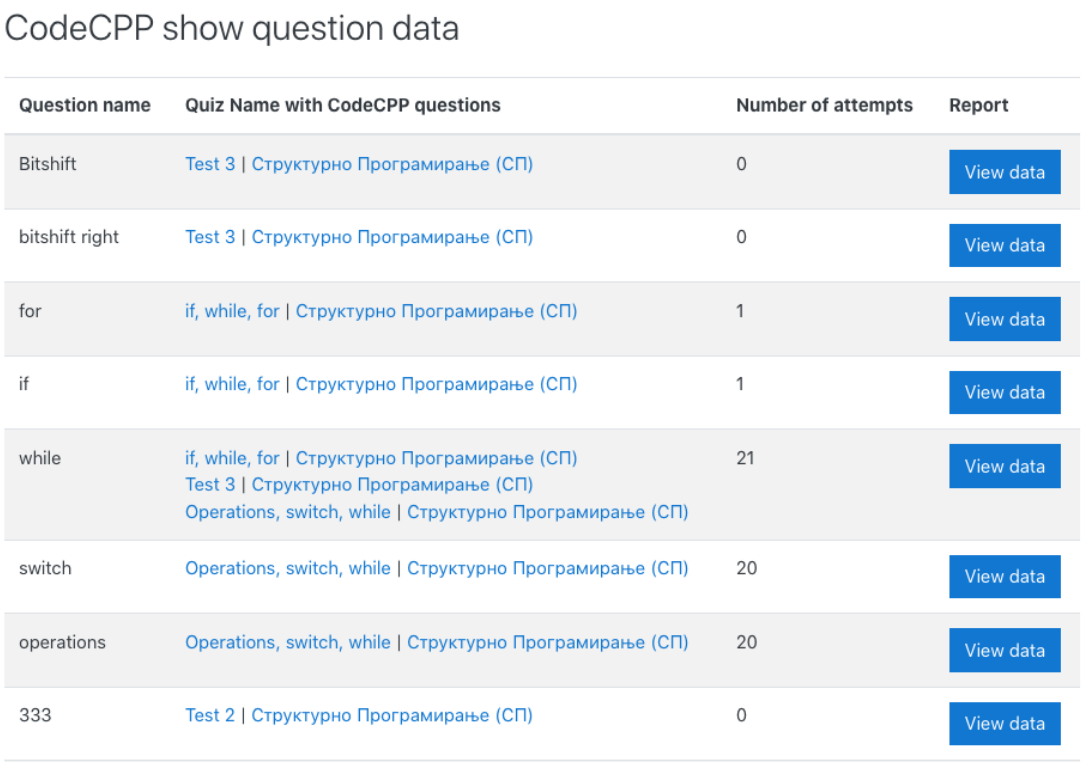
За потребите на понатамошното калибрирање на тежинските вредности што се користат при пресметување на сложеноста на програмските кодови со помош на предложената нова метрика, згодно е да се зачувуваат податоци од спроведени тестирања кои вклучувале прашања од тип CodeCPP, заради нивна понатамошна анализа. Пред сè, од интерес се просечните времиња што студентите ги потрошиле на рачното извршување на одделните операции/контролни наредби, за да го пресметаат излезот од секој код односно да дадат (точен) одговор на секое прашање. Од овие причини, воведена е и можноста за преглед на сите Moodle квизови кои содржат прашања од тип CodeCPP, којашто исто така е достапна само за администраторот (Слика 4.19).



Слика 4.18: Преглед на едно прашање од базата со прашања од тип CodeCPP (перспектива на администратор)

Како што може да се види од Слика 4.19, во овој поглед на администраторот му е даден табеларен приказ на сите прашања од тип CodeCPP, при што за секое прашање е дадено името, листа од сите квизови каде се појавила барем една кодна варијација генерирана од (шаблонскиот код за) тоа прашање, како и вкупниот број на кодни варијации за тоа прашање решавани од страна на студенти во рамките на наведените квизови.

При кликнување на копчето “View data” за одредено прашање во овој табеларен приказ, се прикажуваат статистики за сите кодни варијации на прашањето решавани во соодветните квизови (Слика 4.20). За секоја одделна варијација се прикажува бројот на обиди, минималното, максималното и просечното време (во секунди) на давање точен одговор, како и стандардната девијација. Се разбира, овде се земаат во предвид само податоците за обидите во коишто е даден точен одговор на соодветната варијација, додека останатите обиди се игнорираат.



CodeCPP show question data

Question name	Quiz Name with CodeCPP questions	Number of attempts	Report
Bitshift	Test 3 | Структурно Програмирање (СП)	0	View data
bitshift right	Test 3 | Структурно Програмирање (СП)	0	View data
for	if, while, for | Структурно Програмирање (СП)	1	View data
if	if, while, for | Структурно Програмирање (СП)	1	View data
while	if, while, for | Структурно Програмирање (СП) Test 3 | Структурно Програмирање (СП) Operations, switch, while | Структурно Програмирање (СП)	21	View data
switch	Operations, switch, while | Структурно Програмирање (СП)	20	View data
operations	Operations, switch, while | Структурно Програмирање (СП)	20	View data
333	Test 2 | Структурно Програмирање (СП)	0	View data

Слика 4.19: Приказ на сите квизови кои содржат прашање од тип CodeCPP

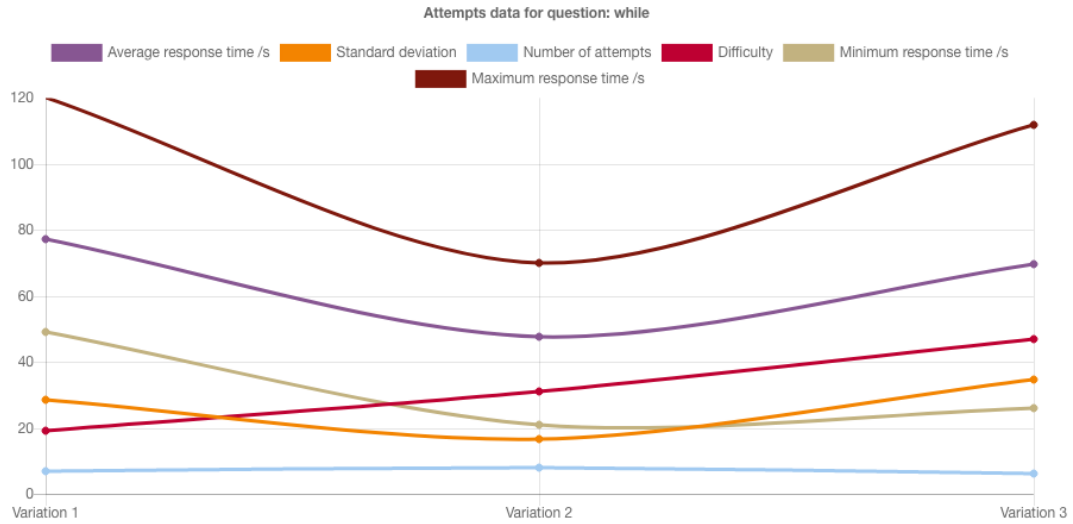
Постои и опција да се преземат сите овие податоци во CSV датотека (со кликнување на хиперврската “Download CSV file with response data”, Слика 4.20), за понатамошни анализи.

4.3.3. Развоен процес на додатокот CodeCPP и имплементација на процесот на генерирање на прашања

Во суштина, имплементацијата на моделот за паметно автоматско генерирање на прашања што содржат програмски код на платформата Moodle побаруваше реализација на два засебни елементи коишто ќе комуницираат меѓусебно.

Првиот елемент е додатокот за Moodle од типот Question types, именуван CodeCPP. Овој додаток беше изработен во програмскиот јазик PHP. Како што беше објаснето и во претходните поглавја, за изработка на додаток од тип Question types потребно е развивање на повеќе датотеки кои овозможуваат негово правилно функционирање. Во овој дел се дефинира природата на прашањето, податоците што секое прашање ќе ги носи со себе, начинот на којшто прашањата ќе комуницираат со активноста во која се наоѓаат (најчесто квиз), тестирањето на прашања, креирањето на база на податоци која ќе ги чува прашањата, итн. Исто така, при развивањето на овој додаток беше неопходно да се внимава да се задржат функционалностите кои типично ги нудат различните типови на прашања во Moodle, а на коишто наставниците вообичаено се

навикнати. Овде спаѓаат можноста за вклучување на прашања од банката со прашања (англ. *question bank*) на Moodle курсот во даден квиз, како и нивно зачувување и изменување, можноста за експортирање на прашања од банката со прашања од еден курс и импортирање во банката на друг курс, и сл.



Hide chart data

Attempts data for question: while

	Average response time /s	Standard deviation	Number of attempts	Difficulty	Minimum response time /s	Maximum response time /s
Variation 1	77.28571428571429	28.44902351180986	7	19.0000	49	120
Variation 2	47.5	16.665833312498957	8	31.0000	21	70
Variation 3	69.5	34.582028087818486	6	47.0000	26	112

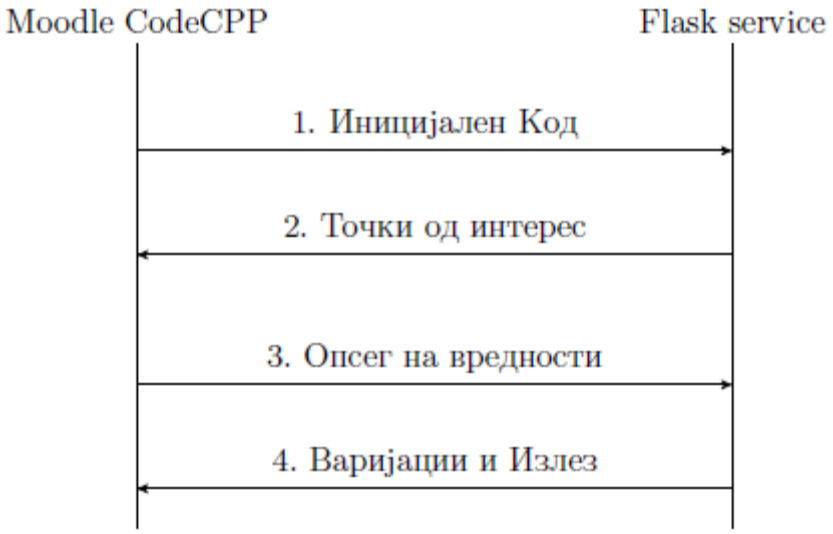
Download CSV file with response data

Слика 4.20: Приказ на статистиките од сите обиди за прашање од тип CodeCPP

Вториот елемент е Python/Flask веб сервисот, којшто веќе беше развиен во рамките на системот за паметно автоматско генерирање на прашања опишан во поглавјето 3.3. Овој сервис ги овозможува сите функционалности во врска со процесирањето на програмските кодови, како што се: генерирање на апстрактно синтаксно дрво за кодот-шаблон и негово парсирање, компајлирање на кодот, извршување на кодот, определување на бројот на извршувања на секоја линија во кодот, како и пресметување на сложеноста на кодот според предложената нова метрика. Конечно, сервисот го извршува и генерирањето на кодни варијации за секој шаблон користејќи зададени конфигурации за правилата за мутирање, а со тоа и генерирањето на прашањата.

Слика 4.21 дава графички приказ на комуникацијата помеѓу двата елемента. Оваа комуникација се состои од следниве последователни чекори:

- CodeCPP ги прибира внесените податоци за секое прашање (шаблонскиот код) и ги проследува до Python веб сервисот;
- Python сервисот прави анализа на добиениот код и ги праќа сите локации од интерес назад до CodeCPP;
- Добиените информации за шаблонскиот код CodeCPP ги прикажува во нов Moodle прозорец, кадешто чека од наставникот да внесе конфигурациски податоци за правилата за мутација за секоја локација од интерес;
- Внесените податоци од наставникот CodeCPP ги испраќа повторно до Python сервисот, којшто започнува со генерирање на кодни варијации за соодветниот шаблонски код врз основа на овие податоци. За секоја кодна варијација истовремено се пресметува нејзината сложеност, па во зависност од неа – кодната варијација се зачувува или отфрла;
- По завршувањето на својата работа, Python сервисот ги праќа сите генерирани изворни кодови до CodeCPP, по што Moodle го пренасочува наставникот на почетниот прозорец на квизот којшто го креирал, а генерираните прашања се зачувуваат во база на податоци за подоцнежно користење.



Слика 4.21: Дијаграм на комуникација помеѓу CodeCPP и Python веб сервисот во процесот на генерирање на прашања

За функционирање на додатокот CodeCPP потребен е веќе функционален сервер на кој што е инсталиран Moodle. Од друга страна, Python сервисот може, но и не мора да биде поставен на истиот сервер на кој што е инсталиран Moodle.

5. Експерименти и резултати

Во оваа глава ќе бидат опишани експериментите изведени во рамките на спроведеното истражување и ќе бидат анализирани добиените резултати. На крајот ќе бидат извлечени и соодветни заклучоци, како и насоки за понатамошно истражување.

5.1. Определување на тежински вредности за кодните операции

Како што беше објаснето во претходната глава, предложената софтверска метрика се заснова на тежински вредности што се придружуваат на секој од операторите и контролните структури во програмскиот код. Овие тежински вредности треба да ги претставуваат нивните „сложености“, во смисол на напорот и времето што треба да ги потроши даден студент за рачно да ги изврши соодветните операции/наредби и да ги пресмета соодветните резултати. Во овој труд се задржуваме на тежинските вредности за аритметичките оператори.

Без сомнение, собирањето на едноцифрен со двоцифрен операнд не претставува операција со иста сложеност како собирањето на четирицифрен со петоцифрен операнд – за пресметување на резултатот од втората операција секако е потребно повеќе време отколку за пресметување на резултатот од првата. Уште повеќе, би се очекувало дека множењето на два двоцифрени операнди ќе потроши повеќе време отколку собирањето на два операнди од истиот тип. Прашање што природно се наметнува овде е: колку повеќе време? Односно, попрецизно: кои се односите на времињата што им се потребни на студентите за изведување на секоја од аритметичките операции (над специфични типови на операнди)? За да се најде одговор на ова прашање, неопходно е да се спроведат мноштво експерименти.

Од погорната дискусија може да се заклучи дека определувањето на соодветни тежински вредности за аритметичките оператори не е тривијален процес. За постигнување на добра прецизност, потребно е да се соберат и анализираат големи множества од податоци.

Бидејќи сеуште не беше развиен системот за паметно генерирање на прашања опишан претходно, за потребите на ова истражување беше развиена помошна софтверска алатка. Таа овозможува истовремено да се тестира способноста на студентите за анализирање на базични програмски кодови, како и да се прибираат неопходните податоци во текот на самиот процес на тестирање. На ваков начин студентите нема да учествуваат во (од нивна гледна точка) „непродуктивна“ активност. Основната идеја е дека студентите ќе бидат мотивирани да учествуваат и да го дадат максимумот од своето знаење,

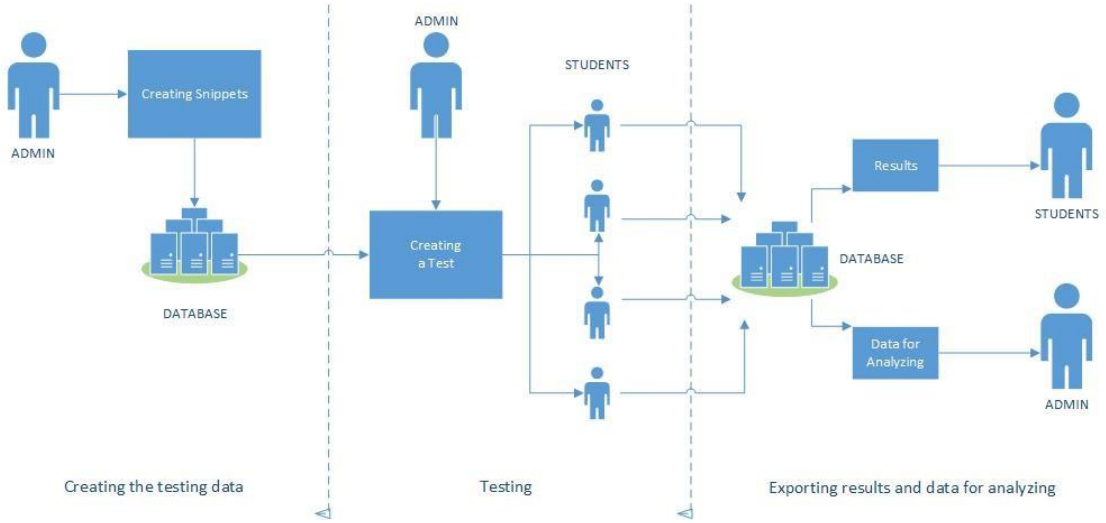
очекувајќи подобри резултати на крајот. Само во вакви околности резултатите можат да се земат за вредни т.е. точни.

5.1.1. Опис на новата софтверска алатка развиена за потребите на истражувањето

Развиената алатка претставува веб сервис којшто може да служи како платформа за собирање податоци и посредник помеѓу истражувањето и студентите.

Во суштина, софтверската алатка ги испитува студентите задавајќи им одреден број на прашања, едноподруго. Прашањата содржат кратки фрагменти од програмски кодови со зададен влез, па сè што студентите треба да направат е да го следат извршувањето на дадениот коден фрагмент, земајќи го во предвид зададениот влез, и да го пресметаат излезот од кодот. Секое прашање се испорачува откако ќе се одговори претходното прашање. Користејќи ја алатката, се мери времето потребно за одговарање (решавање) на секое од прашањата. Овој резултат се зема во предвид како критериум според кој се одлучува колку е компликуван (сложен) даден коден фрагмент.

На Слика 5.1 е претставен поглед од високо ниво на архитектурата на алатката.



Слика 5.1: Поглед на архитектурата на софтверската алатка развиена за потребите на спроведувањето на експериментите

Постојат два типа на корисници: администратори и студенти. Администраторите имаат пристап и контрола врз сите главни компоненти на системот, додека студентите имаат ограничен пристап кој се состои само од решавање на даден тест или пак преглед на резултати од тестирање.

Како што може да се види од дијаграмот на Слика 5.1, алатката се состои од три главни модули. Во продолжение ќе биде дадено детално објаснување на

модулите и сите функционалности достапни за различните типови на корисници.

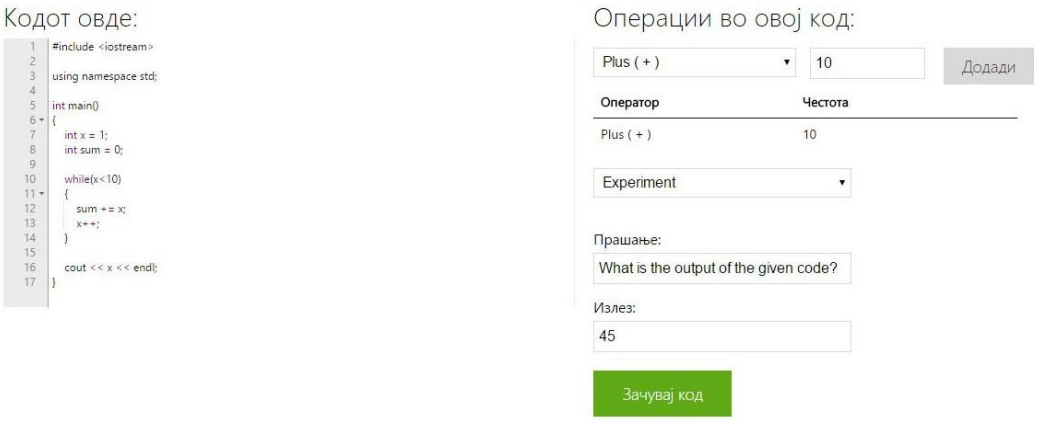
5.1.1.1 Модул за креирање на податоци за тестирање

Беше одлучено дека е најдобро да се раздели модулот за креирање на податоци за тестирање од самото тестирање. На овој начин корисникот ќе може независно да креира податоци за тестирање и со леснотија да ги преискористува истите податоци во различни тестови, со што се заштедува значително време – кога станува збор за процесот на креирање на нови тестови.

Податоците за тестирање може да се креираат или со користење на кориснички-пријателскиот интерфејс којшто е интегриран во самиот софтвер, или пак со рачно вметнување на податоците за тестирање во базата на податоци. Секој запис од податоци за тестирање во базата на податоци се состои од едно темелно опишано прашање што содржи коден фрагмент (влезот, изворниот код на кодниот фрагмент, точниот излез, како и листата од аритметички оператори што се појавуваат во кодниот фрагмент).

Процесот на креирање на податоци за тестирање започнува со навигација кон кориснички-пријателскиот интерфејс за креирање на податоци за тестирање, којшто е достапен само за администраторите (Слика 5.2). Штом ќе го достапи овој интерфејс, администраторот може да креира нови податоци за тестирање – еден по еден податок. Еден податок за тестирање (запис во базата на податоци) претставува единечно прашање што содржи коден фрагмент. Како што беше објаснето и претходно, секое вакво прашање се состои од краток и доста едноставен фрагмент од програмски код, на којшто е придружено прашање во врска со кодот. При креирањето на ново прашање, администраторот треба да го напише прашањето, да го вметне програмскиот код којшто го дефинира кодниот фрагмент, да го специфицира точниот одговор на прашањето во врска со кодниот фрагмент, да ги специфицира аритметичките оператори што се појавуваат во кодниот фрагмент, и конечно – да ја специфицира кодната група на којашто припаѓа тоа прашање. Секое прашање припаѓа на одредена кодна група. На овој начин, прашањата се распределуваат по групи, пришто секоја група има различна намена. На пример, една кодна група може да содржи прашања за проверка на знаење од „`for` циклуси“, друга група да содржи прашања за „основни аритметички операции“, а трета – за „`if` наредби“.

По внесувањето на сите неопходни податоци, прашањето се зачувува во базата на податоци и може да се преискористува во повеќе тестови. Администраторот потоа може да продолжи со внесување на прашања во базата на податоци, или пак да го напушти интерфејсот за креирање на податоци за тестирање.



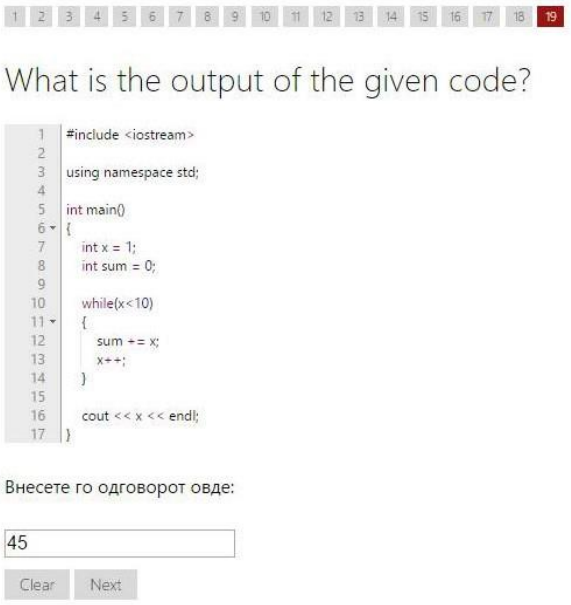
Слика 5.2: Поглед на интерфејсот за креирање на податоци за тестирање на софтверската алатка. На левата страна инструкторот (администраторот) го внесува кодниот фрагмент, а на десната страна се внесуваат дескрипторите на кодот, како и текстот на прашањето за студентите

5.1.1.2 Модул за тестирање

Како што беше спомнато и претходно, заради модуларниот дизајн на системот и можноста за преискористување на податоците за тестирање во различни тестови, за креирање на нов тест потребно е време кое се мери во секунди.

Како прв чекор, администраторот навигира кон интерфејсот за креирање на нов тест. Овде тој треба да пополни неколку полиња: време на започнување на тестот, време на завршување, датум, како и кодна група од којашто ќе се земат прашањата за тестот. Откако администраторот ќе ги специфицира сите потребни информации, новиот тест ќе биде закажан (за започнување во специфицираното време) и зачуван во базата на податоци. Еден целосен тест се состои од повеќе поединечни прашања, како што беше опишано и претходно.

По креирањето на тестот, сè што студентите треба да направат е да се најават на нивните кориснички сметки и да навигираат кон интерфејсот за тестирање. Овде тие ќе видат бројач (тајмер) којшто го одбројува преостанатото време до започнувањето на следниот достапен (закажан) тест. Штом бројачот ќе достигне до нула, студентите можат да го започнат тестот со кликнување на копчето “start”. При кликнување на ова копче, тестот започнува и на студентите им се задаваат прашања од серверот, едно по едно (Слика 5.3). Ова значи дека студентите не можат произволно да „прескокнуваат“ од едно на друго прашање и мораат да ги одговараат прашања во редоследот на појавување.



Слика 5.3: Поглед на прашање (што содржи коден фрагмент) зададено на корисник на софтверската алатка (студент) за време на тестирање. Во полето под кодот потребно е да се внесе одговорот (вообичаено тоа е вредноста на одредена променлива, по извршувањето на кодот)

Штом ќе го одговорат и последното прашање од некој тест, тестот веднаш завршува и студентите можат да ги видат сопствените резултати: бројот на точно одговорени прашања, како и времето што го потрошиле на решавање на тестот.

Податоците коишто всушност беа искористени за истражувањето се објаснети во продолжение.

5.1.1.3 Модул за експортирање на резултати од тестирање

Во овој дел ќе бидат опишани типовите на податоци што се зачувуваат во базата на податоци при тестирање. Студентите, вообичаено, се заинтересирани единствено за бројот на точно одговорени прашања и за времето што го потрошиле во решавање на тестот. Информациите што беа потребни за истражувањето се малку поинакви, па оттука и резултатите што можат да се експортираат од системот се доста пообемни во однос на оние што можат да ги видат студентите по завршувањето на тестот.

Додека студентите решаваат даден тест, за секое прашање што се обидуваат да го одговорат, во базата на податоци се зачувува нивниот одговор, како и времето што им било потребно за да ги извршат соодветните операции содржани во кодниот фрагмент (кој е составен дел на тоа прашање) и да го предадат одговорот на прашањето. Индивидуалното време на одговарање на секое прашање се зачувува бидејќи за истражувањето од интерес се времињата

потребни за изведување на секоја поединечна аритметичка операција во кодните фрагменти. Преку соодветна анализа на овие времиња може да се изведат заклучоци во врска со сложеноста на секоја од основните аритметички операции, кога истите се користат во програмски код – индивидуално или пак заедно со други операции. За да се добијат валидни резултати, мора да се разгледуваат само времињата потребни за предавање на одговор за точно одговорените прашања, па од овие причини се зачувуваат исто така и одговорите на студентите.

Од друга страна, бидејќи студентите се заинтересирани за нивното глобално време на решавање т.е. времето потрошено од започнувањето на тестот до самиот негов крај, во базата на податоци се зачувува и оваа информација – покрај индивидуалното време на одговарање на секое поединечно прашање. Освен за задоволување на љубопитноста на студентите во врска со нивните перформанси, глобалното време може да послужи и како дополнителен критериум според којшто може да се рангираат студентите по завршувањето на тестот.

По завршувањето на даден тест, администраторот може да ги експортира сите податоци во врска со тестот (зачувани во базата на податоци) во Excel документ за понатамошна анализа.

5.1.2. Детали за анализата на присобраните податоци

Имајќи ја листата од сите аритметички оператори содржани во конкретен коден фрагмент, како и времето што им било потребно на студентите за предавање на одговор на соодветното прашање, беше возможно да се анализира времето потребно за рачно извршување на секоја од соодветните аритметички операции. Се започна со разгледување на кодни фрагменти што содржат само една инстанца на еден оператор (во истражувањето беше вклучен барем по еден ваков фрагмент за секоја од аритметичките операции кои се предмет на интерес), потоа се продолжи со кодни фрагменти што содржат повеќекратни инстанци на еден оператор, и конечно се анализираа кодни фрагменти што содржат мешавина од различни оператори. Се анализираше времето на извршување на операциите над едноцифрен и двоцифрен операнд, како и над два двоцифрени операнди. Беше одлучено да се стават во фокус само овие два случаи бидејќи искуството покажува дека нема потреба студентите да се оптоваруваат со поголеми операнди (броеви) при оценување на нивното знаење од аритметичките оператори. Поголемите операнди само ќе предизвикаат подолго време на извршување на операцијата и нема да покажат ништо ново во врска со нивното знаење.

Целта на истражувањето е, врз основа на спроведената анализа, да се определат соодветни тежински вредности коишто би можеле да се придружат

на секој од операторите. Придружените тежински вредности, како што беше објаснето и претходно, треба колку што е можно попрецизно да ги претстават односите на просечните времиња потребни за извршување на соодветните операции од страна на студентите.

Во двете поглавја што следуваат (поглавје 5.2 и поглавје 5.3), прво ќе се опишат експериментите кои беа спроведени со користење на претходно воведената алатка, со цел да се утврдат соодветни тежини за аритметичките оператори, а потоа ќе се анализираат добиените резултати и ќе се дискутираат набљудувањата над податоците и над самиот процес.

5.2. Експерименти 1

5.2.1. Опис на експериментите

Експериментите беа спроведени како дел од лабораториските вежби од курсот Структурно програмирање, којшто се држи во прва година на нашата институција, Факултетот за информатички науки и компјутерско инженерство во Скопје, за време на зимскиот семестар на учебната 2015/2016 година. Вкупно 102 студенти (кои го следеа курсот во овој семестар) учествуваа во експериментите.

Изведени беа два различни експеримента, со две дисјунктни групи од студенти. Единствената разлика во експерименталните поставки беше мотивацијата која им беше дадена на студентите. Имено, на првата група од 35 студенти им беше кажано дека тие ќе учествуваат во експеримент со решавање на тест, чија цел е да им помогне на наставниците на курсот при одлуката за тежината на задачите кои треба да се задаваат на идните испити, врз основа на постигнатите резултати од студентите-учесници во експериментот. Од друга страна пак, на втората група од 67 студенти им беше кажано дека тие ќе учествуваат во решавање на тест кој ќе замени дел од редовните лабораториски вежби од курсот, а со тоа резултатите од тестот ќе бидат земени во предвид во резултатите од активностите на студентите на курсот за тековната седмица.

Сите учесници во експериментите добија за задача да решат тест кој се состоеше од 18 прашања. Секое од прашањата содржеше извадок (фрагмент) од едноставен изворен код кој се фокусира на една (или комбинација од две) од четирите аритметички операции: собирање (+), одземање (-), множење (*), како и делење по модул (%). Прашањето придружено кон секој од кодните фрагменти беше исто – се бараше вредноста на една конкретна променлива од зададениот коден фрагмент. Должината (изразена во линии од код) беше иста за сите кодни фрагменти кои беа вклучени во прашањата, така што студентите требаше да извршат ист број на наредби за да го пресметаат точниот одговор на секое од прашањата. Единствен исклучок беа двете прашања со кодни

фрагменти кои се однесуваа точно на операцијата множење, коишто од практични причини беа пократки за една линија во однос на останатите прашања.

Прашањата беа поделени во 4 групи, како што е објаснето во продолжение.

Првата група се состоеше од 8 прашања, пришто секое од нив побарува од студентот да изврши точно една од аритметичките операции. Секоја аритметичка операција беше претставена со две прашања – едно со извршување на операцијата над еден едноцифрен и еден двоцифрен операнд, и уште едно со извршување на операцијата над два двоцифрени операнди. На Слика 5.4 се прикажани примери за вакви прашања, коишто беа вклучени во тестот, а коишто побаруваа извршување на операцијата одземање над еден едноцифрен и еден двоцифрен операнд, односно операцијата множење над два двоцифрени операнди, соодветно.

Која е вредноста на променливата `a` по извршувањето на следниот C++ програмски код?

```
int x = 85, y = 8, z = 9, a, b;
b = x - y;
a = b - z;
```

Која е вредноста на променливата `x` по извршувањето на следниот C++ програмски код?

```
int a = 16, b = 18, x;
x = a * b;
```

Слика 5.4: Прашања од првата група коишто ги претставуваа операциите одземање (над едноцифрен и двоцифрен операнд) и множење (над два двоцифрени операнди) во експерименталниот тест

Втората група на прашања се фокусираше на комбинации од по 2 различни аритметички операции. Во секое од прашањата се бараше прво да се пресмета вредноста на променливата којашто го чува резултатот од извршувањето на првата операција, а потоа да се изврши втората операција над таа променлива и уште еден операнд и добиениот резултат да се зачува во нова променлива. Групата вклучуваше 4 прашања, по едно за секој од следните парови операции: 1) делење по модул и одземање; 2) множење и собирање; 3) делење по модул и собирање; и 4) множење и одземање.

Третата и четвртата група на прашања беа дефинирани со цел да се истражи влијанието од примената на наредбите `if` (условно извршување) и `for` (циклуси) за контрола на програмскиот тек, врз времето за извршување на аритметичките операции вгнездени во нивните тела. Двете прашања коишто беа дел од третата група вклучуваа извршување на една операција на

собирање/одземање, во зависност од едноставен логички услов во рамките на една наредба `if`. Четирите прашања коишто беа содржани во четвртата група побаруваа извршување на една аритметичка операција точно 5 пати – по еднаш за секоја од петте итерации на единечната наредба `for` присутна во кодниот фрагмент. Повторно, единствениот исклучок во групата беше прашањето што се фокусираше на операцијата множење, коешто заради аритметички ограничувања побаруваше извршување на операцијата 3 пати (наместо 5), како 3 итерации на наредбата `for` во којашто беше вгнездена. Примери за прашања од третата и четвртата група се дадени на Слика 5.5 и Слика 5.6, соодветно.

Која е вредноста на променливата `y` по извршувањето на следниот C++ програмски код?

```
int x = 23, y = 14;
if(y <= x)
      y = y + 8;
```

Слика 5.5: Прашање од третата група во експерименталниот тест – операцијата собирање вгнездена во телото на наредба `if`

Која е вредноста на променливата `z` по извршувањето на следниот C++ програмски код?

```
int i, z = 12;
for(i = 1; i <= 5; i++)
      z = z + i;
```

Слика 5.6: Прашање од четвртата група во експерименталниот тест – операцијата собирање вгнездена во телото на наредба `for`

5.2.2. Резултати и анализа

Како што беше претходно објаснето, за секој учесник во експериментите беа присобрани сите податоци околу потребното време за решавање на секое прашање. Имајќи ја во предвид точноста на решенијата, во понатамошната анализа беа земени само времињата на решавање за точно одговорените прашања.

Главна цел на поширокото истражување е да се утврди просечниот однос помеѓу просечните времиња потребни за извршување на аритметичките операции од страна на студентите (на пример, да се заклучи дека времето за извршување на одземање на 2 двоцифрени операнди е за 1,2 пати повеќе од времето за извршување на собирање на 2 двоцифрени операнди). За да може да се земат во предвид односите, неопходно е да се утврди постоењето на корелација помеѓу нив.

Беше направена пресметка на коефициентите на корелација за секој пар од операции од интерес. Ова беше направено одделно за двата експеримента.

Заради недостаток од мотивација кај првата група на студенти, беа забележани значително пониски коефициенти на корелација кај првиот експеримент, па беше одлучено да се занемарат резултатите од истиот.

Табела 5.1 ги прикажува пресметаните коефициенти на Пирсонова корелација (Pearson, 1895) помеѓу времињата потребни за извршување на различните операции кои беа предмет на анализа, за вториот експеримент. Некои од податоците користени при пресметките беа директните записи во базата на податоци за соодветните прашања, а за повеќето прашања истите беа одраз на впросечување (на пример, ако имаше две операции од ист тип во дадено прашање, времето потребно за решавање на соодветното прашање беше поделено со 2). Земајќи предвид дека се разгледуваа само добиените времиња за точно решените прашања, секоја пресметка беше заснована на 30 или повеќе податоци (односно, потребните времиња на решавање за секое прашање на 30 или повеќе студенти – оние коишто дале точен одговор), освен за прашањето коешто вклучуваше `for` наредба која го повторуваше извршувањето на операцијата одземање 5 пати – за него беа достапни само 25 податоци.

	+ 2 1	**+ 2 2**	**- 2 1**	**- 2 2**	*** 2 1**	*** 2 2**	**% 2 1**	**% 2 2**	**IF - 2 2**	**IF + 2 1**	**FOR + 2 1**	**FOR - 2 2**	**FOR * 2 1**
+ 2 1	1	**0.61**	0.40	**0.66**	0.46	0.30	**0.64**	**0.53**	0.42	0.37	**0.58**	0.36	0.23
+ 2 2	**0.61**	1	**0.67**	**0.54**	0.31	0.20	**0.54**	0.44	0.40	0.04	**0.53**	0.19	0.25
- 2 1	0.40	**0.67**	1	**0.57**	0.17	0.11	**0.60**	0.23	0.30	0.23	0.45	0.28	0.49
- 2 2	**0.66**	**0.54**	**0.57**	1	0.42	0.18	0.47	**0.57**	**0.52**	0.37	**0.50**	**0.50**	0.32
*** 2 1**	0.46	0.31	0.17	0.42	1	0.22	**0.57**	0.33	0.34	0.08	**0.53**	-0.02	-0.24
*** 2 2**	0.30	0.20	0.11	0.18	0.22	1	0.25	0.11	-0.03	0.07	0.15	0.11	0.43
% 2 1	**0.64**	**0.54**	**0.60**	0.47	**0.57**	0.25	1	**0.56**	**0.58**	**0.55**	**0.63**	0.34	0.21
% 2 2	**0.53**	0.44	0.23	**0.57**	0.33	0.11	**0.56**	1	0.23	0.14	**0.56**	**0.55**	0.44
IF - 2 2	0.42	0.40	0.30	**0.52**	0.34	-0.03	**0.58**	0.23	1	**0.77**	**0.64**	0.42	0.24
IF + 2 1	0.37	0.04	0.23	0.37	0.08	0.07	**0.55**	0.14	**0.77**	1	0.37	**0.54**	0.25
FOR + 2 1	**0.58**	**0.53**	0.45	**0.50**	**0.53**	0.15	**0.63**	**0.56**	**0.64**	0.37	1	**0.60**	0.49
FOR - 2 2	0.36	0.19	0.28	**0.50**	-0.02	0.11	0.34	**0.55**	0.42	**0.54**	**0.60**	1	0.34
FOR * 2 1	0.23	0.25	0.49	0.32	-0.24	0.43	0.21	0.44	0.24	0.25	0.49	0.34	1

Табела 5.1: Коефициенти на Пирсонова корелација помеѓу времињата потребни за извршување на различните аритметички операции анализирани во Експеримент 1 (Stankov *et al.*, 2016)

Легенда:

“**+ 2 1**” – Собирање на еден двоцифрен и еден едноцифрен операнд;

“**+ 2 2**” – Собирање на два двоцифрени операнди;

“**– 2 1**” – Одземање на едноцифрен од двоцифрен операнд;

“**– 2 2**” – Одземање на двоцифрен од двоцифрен операнд;

“*** 2 1**” – Множење на еден двоцифрен и еден едноцифрен операнд;

“*** 2 2**” – Множење на два двоцифрени операнди;

“**% 2 1**” – Делење по модул на двоцифрен со едноцифрен операнд;

“**% 2 2**” – Делење по модул на двоцифрен со двоцифрен операнд;

“**IF – 2 2**” – Одземање на двоцифрен од двоцифрен операнд, вгнездена во наредба ‘IF’;

“**IF + 2 1**” – Собирање на еден двоцифрен и еден едноцифрен операнд, вгнездена во наредба ‘IF’;

“**FOR + 2 1**” – Собирање на еден двоцифрен и еден едноцифрен операнд, вгнездена во наредба ‘FOR’ (5 итерации);

“**FOR** – 2 2” – Одземање на двоцифрен од двоцифрен операнд, вгнездена во наредба ‘FOR’ (5 итерации);

“**FOR * 2 1**” – Множење на еден двоцифрен и еден едноцифрен операнд, вгнездена во наредба ‘FOR’ (3 итерации).

Од Табела 5.1 се гледа дека *повеќето од операциите покажуваат значителна корелација една со друга*, но постојат и некои операции кои се едноставно незначително корелирани со други операции.

Највисоката корелација од 0,77 е забележана помеѓу 2 прашања, кои вклучуваат по една наредба `if` со единствена аритметичка операција во нејзиното тело: операција собирање (над едноцифрен и двоцифрен операнд) во првото прашање, и операција одземање (над 2 двоцифрени операнди) во второто прашање. Ова овозможува да се утврди односот помеѓу операцијата одземање на 2 двоцифрени операнди и операцијата собирање на едноцифрен со двоцифрен операнд. Утврдениот однос е ***1,42***.

Интересно е да се набљудува корелацијата помеѓу двата типа на операции на множење и останатите операции. Како што може да се види од Табела 5.1, не постои речиси никаква значителна корелација со ниту една од останатите операции, па дури ниту помеѓу нив две. Главната разлика помеѓу двете прашања кои ги претставуваа овие операции на множење и останатите прашања е тоа што во овие две прашања се појавуваше само една инстанца (наместо две инстанци) на соодветната операција. Ова треба дополнително да се истражи.

Уште едно мошне интересно набљудување е следново: првите 2-3 прашања покажуваат подолги времиња на решавање, иако операцијата со којашто требаше да се справат студентите во овие прашања беше собирање на едноцифрен со двоцифрен операнд или собирање на два двоцифрени операнди, за којашто не е разумно да побарува повеќе време за пресметување во однос, на пример, на множењето на операнди од истиот тип. Оваа појава може да се

објасни со фактот дека студентите не биле запознати со изгледот на прашањата, па им било потребно да погледнат, анализираат и одговорат неколку од нив пред да го усвојат истиот. Уште повеќе, сепак постои висока корелација помеѓу времињата на одговарање на првите две прашања и на оние на повеќе други прашања, и ова го потврдува тврдењето дека добиените времиња за двете прашања не се случајни, туку едноставно се последица на потребата на студентите да се навикнат на изгледот на прашањата.

Претходниот заклучок очигледно го потврдуваат и самите резултати. На пример, корелацијата помеѓу времињата на извршување на операцијата собирање на два двоцифрени операнди и времињата на извршување на операцијата собирање на едноцифрен со двоцифрен операнд (првите две прашања) е 0,61, но пресметаниот однос е 0,9. Логично, би се очекувало дека операцијата над два двоцифрени операнди ќе побарува повеќе, а не помалку време. Ова потврдува дека во второто прашање студентите биле повеќе навикнати на формата на прашањето врз основа на претходно одговореното прашање. Оваа појава продолжува да биде присутна, исто така, и кај следните две прашања, но ефектот е редуциран. Конечно, пресметаниот однос помеѓу двете операции на одземање (два двоцифрени операнди наспроти едноцифрен со двоцифрен операнд) е 1,05 (корелацијата е 0,57), и ова е вредност којашто сеуште не е според очекувањата, но барем е поголема од 1.

Врз основа на горната анализа беше донесен заклучок *во следните експерименти да бидат ставени 3 или 4 прашања коишто што ќе ја имаат само подготвителната улога да ги запознаат студентите со изгледот на вистинските прашања и коишто нема да бидат вклучени во анализата на експерименталните резултати*.

За да се добијат валидни резултати, *многу е важно да се елиминираат факторите кои би можеле да имаат влијание на времето потребно за извршување на секоја од аритметичките операции, како што е непознавањето на изгледот на прашањата објаснето погоре*. Уште еден ваков фактор е редоследот на прашањата во тестот – студентите би можеле да откриваат шаблони во редоследот на прашањата и да го искористат ова за предвремено да ги пресметуваат одговорите на овие прашања (резултатите од операциите) веднаш штом ги видат операндите (пред дури и да го погледнат операторот што треба да се примени!). Пример за ваков шаблон би била секвенца од прашања во кои наизменично се менуваат операндите на некоја конкретна операција: прашање во кое дадена операција треба да се изврши над едноцифрен и двоцифрен операнд, по кое следува прашање во кое истата операција треба да се изврши над два двоцифрени операнди, и ова се повторува последователно за секоја од операциите. Едно можно решение на овој проблем е да се измешаат прашањата на таков начин што нема да се појавуваат вакви шаблони во

тестовите што се задаваат на студентите. Оттука беше извлечен уште еден заклучок: во експериментите што ќе следуваат, *да се направат пермутации на редоследот на прашањата за да се елиминира присуството на какви било шаблони*.

Како што можеше да се заклучи и од првиот од двата спроведени експерименти, фактор кој би можел да има многу силно влијание врз времињата на извршување на аритметичките операции е *мотивираноста на студентите да учествуваат во експерименталната активност*. Потребно е да се посвети големо внимание на мотивирањето на студентите на доста високо ниво, можеби преку симулирање на околина на испит од курс, натпревар или пак некоја друга активност која би ги стимулирала студентите да вложат максимален напор во обидот да ги одговорат точно сите зададени прашања во најкраток можен временски интервал.

Наспроти пристапот којшто беше преземен во овие експерименти, потенцијално подобар пристап за прецизно да се измери времето потребно за рачно извршување на конкретна аритметичка операција би можел да биде пристапот во кој *би се користеле прашања кои не побаруваат никакво знаење од програмскиот јазик, односно „чисти“ прашања кои содржат само два операнди (во облик на константни вредности), како и операторот што треба да се примени над нив, без променливи или доделувања на вредности*. За експериментите што ќе следуваат беше заклучено да се изведат токму со овој пристап.

5.2.3. Заклучоци од експериментите

Главниот заклучок од спроведените експерименти е дека постои корелација помеѓу времињата потребни за извршување на одредени операции (аритметички операции, како дел од различни програмски структури), што овозможува спроведување на нови, дополнителни експерименти за подобро (попрецизно) определување на односите помеѓу различните операции. Уште повеќе, беа претставени дури и некои прелиминарно пресметани односи.

Беа извлечени и значајни заклучоци во врска со изгледот на идните експерименти за определување на „тежини“ за операциите. На пример, би било пожелно да се воведат 3 или 4 почетни прашања коишто ќе ја имаат само подготвителната улога да ги запознаат студентите со изгледот на вистинските прашања и коишто нема да бидат вклучени во анализата на експерименталните резултати. Исто така, би требало да се разгледа можноста за пермутирање на редоследот на прашањата, со цел да се избегне детекција на некои шаблони од страна на студентите. Конечно, пред да се прифатат резултатите од експериментите, неопходно е внимателно да се разгледа мотивираноста на студентите за учествување во експерименталната активност.

Уште еден значаен заклучок, којшто беше донесен во врска со новите експерименти коишто ќе следуваат, е да се користи поинаков пристап заради попрецизно мерење на времето што е потребно за рачно извршување на аритметичките операции. Имено, во тие експерименти ќе бидат употребени исклучиво едноставни прашања кои не побаруваат знаење од програмскиот јазик, односно „чисти“ прашања кои содржат само два операнди (во облик на константни вредности) заедно со операторот што треба да се примени над нив, без променливи или доделувања на вредности.

5.3. Експеримент 2

Во овој дел ќе биде опишан новиот, подобрен експеримент кој беше спроведен со користење на претходно опишаната алатка, со цел да се калибрираат колку што е можно попрецизно тежините за аритметичките оператори. Исто така, ќе бидат анализирани добиените резултати и ќе бидат дискутирани подобрувањата постигнати со користењето на овој нов пристап.

5.3.1. Опис на експериментот

Експериментот беше спроведен како дел од лабораториските вежби од курсот Структурно програмирање, којшто се држи во прва година на нашата институција, Факултетот за информатички науки и компјутерско инженерство во Скопје, за време на зимскиот семестар на учебната 2017/2018 година. Вкупно 135 студенти (кои го следеа курсот во овој семестар) учествуваа во експериментот. При дизајнирањето на експерименталната активност, внимателно беа разгледани сите заклучоци и сугестии од претходно спроведеното истражување што беше опишано во претходното поглавје (поглавјето 5.2).

Сите учесници во експериментот добија за задача да решат тест кој се состоеше од 18 прашања, при што секое од нив содржеше едноставен аритметички израз кој се фокусираше на една (или комбинација од две) од четирите аритметички операции: собирање (+), одземање (-), множење (*), како и делење по модул (%). Во секое од овие прашања од студентите се бараше да ја пресметаат вредноста на аритметичкиот израз содржан во прашањето. Притоа, кај сите прашања од тестот, операндите во аритметичките изрази беа константни вредности – не се појавуваа променливи и немаше доделување на вредности. Попрецизно, операндите секогаш беа едноцифрени или двоцифрени цели броеви: во согласност со објаснувањето дадено во поглавјето 5.1.2, идентично како и во претходните експерименти, анализирани беа само случаите кадешто соодветната аритметичка операција треба да се изврши над едноцифрен и двоцифрен операнд, како и над два двоцифрени операнди.

Прашањата од експерименталниот тест беа поделени во 4 групи, како што следува.

Првата група од прашања е подготвителната група. Целта на оваа група беше да ги запознае студентите со изгледот на прашањата – како што беше објаснето погоре, сите прашања од тестот имаа ист изглед, кој се состои од аритметички израз и прашање кое ја побарува вредноста на тој израз. Како одговор на секое од прашањата, од студентите се бараше да предадат еден цел број – бараната вредност на соодветниот аритметички израз. Групата вклучуваше 3 прашања, каде секое прашање се однесуваше на различна аритметичка операција. Со оглед на подготвителната улога на групата, при анализата на резултатите од експериментот не беа разгледувани перформансите на студентите на овие прашања.

Втората група се состоеше од 8 прашања, при што секое од нив побаруваше од студентите да извршат точно една од четирите аритметички операции кои беа предмет на интерес во експериментот. Секоја аритметичка операција беше претставена со по две прашања во оваа група – едно кое побарува од студентите да ја извршат операцијата над еден едноцифрен и еден двоцифрен операнд, и уште едно со извршување на операцијата над два двоцифрени операнди. На Слика 5.7 се прикажани примери за прашања од втората група коишто беа вклучени во тестот, а коишто ја претставуваа операцијата множење над еден едноцифрен и еден двоцифрен операнд, односно операцијата собирање над два двоцифрени операнди, соодветно.

Која е вредноста на следниот аритметички израз во програмскиот јазик C++?

```
19 * 8
```

Која е вредноста на следниот аритметички израз во програмскиот јазик C++?

```
28 + 46
```

Слика 5.7: Прашања од втората група коишто ги претставуваа операциите множење (над двоцифрен и едноцифрен операнд) и собирање (над два двоцифрени операнди) во експерименталниот тест

Третата група од прашања се фокусираше на комбинации од по две различни аритметички операции. Во секое од прашањата од оваа група се бараше прво да се изврши една од операциите над два конкретни операнди за да се добие одреден меѓурезултат, а потоа да се искористи овој резултат заедно со уште еден операнд за да се изврши втората операција. Групата вклучуваше 4 прашања, по едно за секој од следните парови операции: 1) делење по модул и одземање; 2) множење и собирање; 3) делење по модул и собирање; и 4)

множење и одземање. Пример за прашање од третата група е илустриран на Слика 5.8.

Четвртата и последна група од 3 прашања е групата за „проверка на конзистентноста“. Повторно, секое од прашањата во групата побаруваше од студентите да извршат по една различна аритметичка операција: собирање, одземање или множење. Аритметичките изрази во овие прашања вклучуваа исклучиво двоцифрени операнди. Причина за воведување на оваа група од прашања во експерименталниот тест беше да се провери конзистентноста на однесувањето на студентите т.е. да се провери дали пресметаните односи помеѓу времињата потребни за извршување на различните операции за секој конкретен студент може да се сметаат за релевантни, или пак истите биле добиени случајно.

Која е вредноста на следниот аритметички израз во програмскиот јазик C++?

```
58 - (83 % 12)
```

Слика 5.8: Прашање од третата група во експерименталниот тест – резултатот од извршувањето на операцијата делење по модул треба да се употреби како втор операнд во извршувањето на операцијата одземање

Заради нивната подготвителна улога, прашањата од првата група се појавија како први три прашања во редоследот на појавување на прашања во експерименталниот тест. Прашањата од сите други групи се појавуваа во мешан редослед, но притоа беше внимавано да не се случи две прашања од иста група да се појават последователно во тестот. Намерата со овој распоред на прашања беше да се елиминира (или барем минимизира) можноста за детекција на каков било шаблон на прашања во тестот од страна на студентите кој би можел да влијае врз нивните времиња на решавање (одговарање) на прашањата.

5.3.2. Резултати и анализа

Како што беше објаснето и претходно, при анализирањето на потребните времиња за решавање на секое од прашањата во тестот за секој учесник во експериментот, беа разгледувани само точно одговорените прашања. Времињата потребни за решавање на оние прашања за кои студентите не предале точен одговор беа отстранети пред да се започне со анализата. Уште повеќе, комплетно беа отстранети резултатите (за сите прашања од тестот) на оние студенти кои немаа предадено точен одговор на 3 или повеќе прашања од тестот – беше заклучено дека не може да се извлечат релевантни заклучоци во врска со потребните времиња за решавање од студенти кои покажуваат релативно висока рата на грешки при изведувањето на едноставни аритметички операции.

Табела 5.2 ги прикажува пресметаните Пирсонови коефициенти на корелација помеѓу времињата потребни за извршување на различните операции кои беа предмет на анализа во експериментот. Знаејќи го фактот дека се разгледуваа само добиените времиња за точно решените прашања, како и дека се земаа во предвид само резултатите на оние студенти кои имаа предадено погрешен одговор на најмногу 2 прашања, овде мора да се нагласи дека во секоја „колона“ (која ги содржеше времињата на решавање на точно едно од прашањата) беа присутни 30 или повеќе податоци.

	+ 2 1	**+ 2 2**	**- 2 1**	**- 2 2**	*** 2 1**	*** 2 2**	**% 2 1**	**% 2 2**	**+ 2 2 C**	**- 2 2 C**	*** 2 2 C**
+ 2 1	1	0,41	**0,56**	**0,69**	0,47	0,09	0,33	0,32	0,43	**0,64**	0,22
+ 2 2	0,41	1	0,48	**0,61**	**0,53**	0,05	**0,51**	0,42	**0,85**	**0,57**	0,13
- 2 1	**0,56**	0,48	1	**0,56**	0,48	0,11	0,42	**0,54**	0,39	**0,51**	-0,01
- 2 2	**0,69**	**0,61**	**0,56**	1	**0,59**	0,45	0,46	**0,70**	**0,57**	**0,87**	0,19
*** 2 1**	0,47	**0,53**	0,48	**0,59**	1	0,44	0,39	**0,57**	0,41	**0,50**	0,16
*** 2 2**	0,09	0,05	0,11	0,45	0,44	1	0.07	0.19	0.24	0.29	**0.55**
% 2 1	0,33	**0,51**	0,42	0,46	0,39	0.07	1	**0,69**	0,46	0,23	-0,05
% 2 2	0,32	0,42	**0,54**	**0,70**	**0,57**	0.19	**0,69**	1	0,41	0,43	-0,13
+ 2 2 C	0,43	**0,85**	0,39	**0,57**	0,41	0.24	0,46	0,41	1	**0,58**	0,31
- 2 2 C	**0,64**	**0,57**	**0,51**	**0,87**	**0,50**	0.29	0,23	0,43	**0,58**	1	0,25
*** 2 2 C**	0,22	0,13	-0,01	0,19	0,16	**0.55**	-0,05	-0,13	0,31	0,25	1

Табела 5.2: Коефициенти на Пирсонова корелација помеѓу времињата потребни за извршување на различните аритметички операции анализирани во Експеримент 2 (Stankov, Jovanov и Madevska Bogdanova, 2017)

Легенда:

“**+ 2 1**” – Собирање на еден двоцифрен и еден едноцифрен операнд;

“**+ 2 2**” – Собирање на два двоцифрени операнди;

“**– 2 1**” – Одземање на едноцифрен од двоцифрен операнд;

“**– 2 2**” – Одземање на двоцифрен од двоцифрен операнд;

“*** 2 1**” – Множење на еден двоцифрен и еден едноцифрен операнд;

“*** 2 2**” – Множење на два двоцифрени операнди;

“**% 2 1**” – Делење по модул на двоцифрен со едноцифрен операнд;

“**% 2 2**” – Делење по модул на двоцифрен со двоцифрен операнд;

“**+ 2 2 C**” – Собирање на два двоцифрени операнди (проверка на конзистентност);

“– **2 2 C**” – Одземање на двоцифрен од двоцифрен операнд (проверка на конзистентност);

“*** 2 2 C**” – Множење на два двоцифрени операнди (проверка на конзистентност).

Од Табела 5.2 се гледа дека постои висока корелација помеѓу времињата на решавање за прашањата „од ист тип“ (секое од прашањата, коишто вклучуваа извршување на точно една аритметичка операција над два двоцифрени операнди, од втората група прашања во експерименталниот тест, и неговото соодветно прашање за „проверка на конзистентноста“, од четвртата и последна група прашања). Ова потврдува дека студентите покажале конзистентност во однесувањето, па може да се заклучи дека пресметаните односи помеѓу времињата потребни за извршување на различните операции за секој конкретен студент може да се сметаат за релевантни. Исто така, од Табела 5.2 може да се забележи дека постои значајна корелација помеѓу операциите собирање и одземање, но, на пример, операцијата множење над два двоцифрени операнди покажува ниска корелација кон сите други разгледувани аритметички операции.

Главниот заклучок од експериментот е дека може да се утврдат односите помеѓу одредени парови од различни операции, но задолжително треба да се обрне внимание при дефинирањето на односот помеѓу некои од операциите (на пример, при дефинирање на односот помеѓу множењето над два двоцифрени операнди и собирањето или одземањето). Уште повеќе, врз основа на добиените резултати од експериментот, беа утврдени следниве односи помеѓу парови од операции: а) односот помеѓу операцијата собирање на еден двоцифрен и еден едноцифрен операнд (+ 2 1) и операцијата одземање на двоцифрен од двоцифрен операнд (- 2 2) е **0,63**; б) односот помеѓу операцијата собирање на двоцифрен и двоцифрен операнд (+ 2 2) и операцијата одземање на двоцифрен од двоцифрен операнд (- 2 2) е **1,00**; в) односот помеѓу операцијата одземање на двоцифрен од двоцифрен операнд (- 2 2) и операцијата делење по модул на двоцифрен со двоцифрен операнд (% 2 2) е **0,35**; г) односот помеѓу операцијата собирање на еден двоцифрен и еден едноцифрен операнд (+ 2 1) и операцијата одземање на едноцифрен од двоцифрен операнд (- 2 1) е **0,78**; д) односот помеѓу операцијата собирање на двоцифрен и двоцифрен операнд (+ 2 2) и операцијата множење на двоцифрен и едноцифрен операнд (* 2 1) е **0,75.**

5.3.3. Заклучок од експериментот

Следејќи го претходното истражување и експериментите коишто беа спроведени како дел од истото, новиот експеримент беше спроведен со цел да се подобри мерењето на сложеноста на програмските кодови. Овој експеримент беше дизајниран внимателно, земајќи ги во предвид недостатоците на претходните експерименти и водејќи се од заклучоците и сугестиите кои произлегоа од истите. Резултатите што се добија од експериментот покажуваат дека *постои значајна корелација помеѓу операциите собирање и одземање*, но, на пример, операцијата множење над два двоцифрени операнди покажува ниска корелираност кон сите други разгледувани аритметички операции.

Главниот заклучок од експериментот е дека *може да се утврдат односите помеѓу одредени парови од различни операции, но дека треба да се обрне внимание при дефинирањето на односот помеѓу некои од операциите*. Уште повеќе, врз основа на резултатите, беа претставени пресметаните односи помеѓу некои парови од операции.

Како идна работа, се планира да се дизајнираат и спроведат понатамошни експерименти користејќи ја развиената софтверска алатка, со цел да се утврдат соодветни тежински вредности за некои други типови на оператори (релациони, логички), како и за наредбите за контрола на програмскиот тек на програмскиот јазик C/C++ (`if`, `switch`, `while`, `do-while`, `for`).

5.4. Експеримент 3

Во овој дел ќе биде изложено истражувањето кое беше изведено со цел да се евалуира степенот на успешност на автоматската проверка на знаење кај воведните курсеви за програмирање, со користење на додатокот CodeCPP за паметно автоматско генерирање на прашања што содржат програмски код, во рамки на платформата за е-учење Moodle. Прво ќе биде опишан експериментот што беше спроведен во рамките на ова истражување, а потоа ќе бидат претставени резултатите и ќе биде извлечен соодветен заклучок.

5.4.1. Опис на експериментот

Експериментот беше спроведен во новоформираната Математичко-информатичка гимназија (МИГ) во Скопје, која започна со работа од учебната 2020/2021 година. Оваа гимназија претставува средно училиште од посебен интерес за државата, во кое треба да се образуваат одбрани талентирани ученици, особено во полето на математиката и информатиката. Експериментот беше реализиран во текот на наставата од курсот Програмирање, којшто според наставната програма за оваа гимназија се слуша во прва година и претставува воведен курс кој не претпоставува никакво предзнаење од областа на програмирањето. Целта на ова истражување беше да се направи достапен системот за генерирање на прашања CodeCPP до наставникот на курсот, со цел да се оцени неговата корисност на ригорозен начин, преку испитување на учениците во одредени временски интервали во текот на учебната година, како и преку соодветни интервјуа на крајот на учебната година. Наставната програма беше реализирана во година во која заради пандемијата од COVID-19 наставата се реализираше онлајн, а беа опфатени 26 ученици во 2 паралелки.

Заради природата на материјалот и условите на онлајн настава, наставникот го искористи CodeCPP за генерирање на 4 теста (со соодветни прашања за тековната наставна тема во кои се побарува рачна евалуација на

даден програмски код), во текот на второто и третото тромесечје од курсот. Се разбира, секој од учениците при решавање на даден тест добиваше различна верзија од секое прашање, односно различна кодна варијација на шаблонскиот код внесен од страна на наставникот за соодветното прашање. Во Додаток Б е даден приказ на сите прашања (шаблони) кои беа составен дел на еден од зададените тестови, на тема „Низи“.

5.4.2. Резултати и анализа

Во Табела 5.3 се прикажани деталните резултати постигнати од учениците на секој од тестовите, нивниот просечен резултат (изразен во проценти), како и просечната оценка за курсот (заведена во дневник) на секој од учениците постигната во второто и третото тромесечје. Оценката за секој ученик во секое од двете тромесечја е изведена со помош на оценување преку портфолио, односно со следење на редовната активност на ученикот, задавање домашни задачи, испрашување и решавање задачи на час, активност на час, како и дополнителен интерес и учество во дополнителни активности за подготовка за натпревари по информатика. Само двајца од учениците отсуствуваа од еден од тестовите, па кај нив е земена просечната вредност од резултатите од останатите 3 теста.

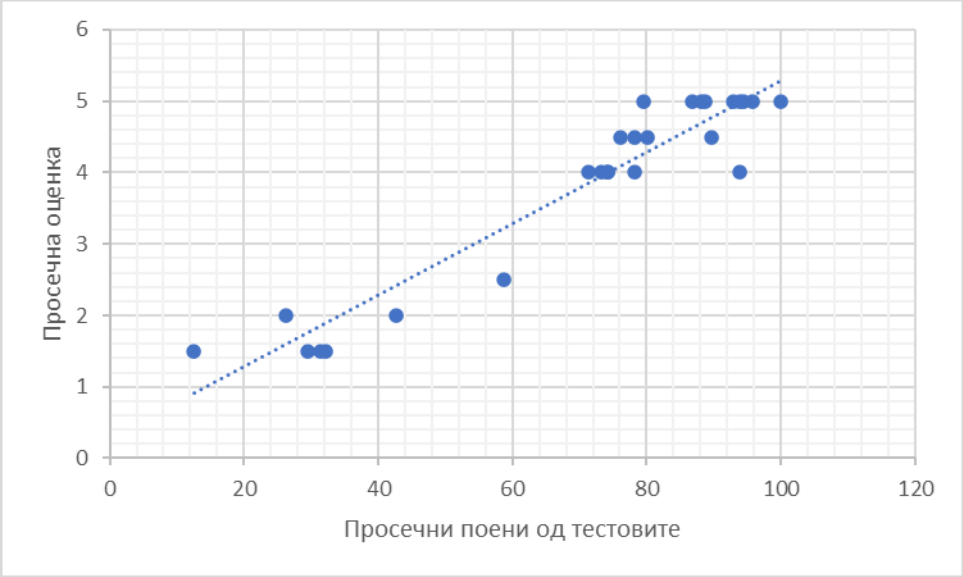
Како што може да се забележи од графиконот на Слика 5.9, оценката на учениците го следи резултатот кој тие го постигнале на тестовите. Корелацијата е очигледна и од графиконот, а пресметаната Пирсонова корелација е 0,955. Овој резултат покажува дека со добро подготвени тестови, учениците можат да добиваат редовна и моментална повратна информација за нивниот прогрес во учењето и нивото на познавање на материјата во даден момент.

По завршувањето со целогодишната активност, беа спроведени и интервјуа со наставникот на курсот и со учениците. Согледувањата на наставникот може да се резимираат со следните неколку заклучоци:

1. CodeCPP е додаток на Moodle кој е многу лесен за користење, и со него за неколку минути може да се подготви тест за одреден дел од материјата.
2. CodeCPP овозможува огромна флексибилност за наставникот, со тоа што нуди големо множество на точки од интерес во програмскиот код кои може да се изменуваат. На ваков начин наставникот може точно да избере какви ќе бидат верзиите на прашањата од секој тип кои ќе ги добиваат учениците, а со тоа да го провери точно она знаење за кое е заинтересиран.

Ученик	Тест 1 (10 п.)	Тест 2 (10 п.)	Тест 3 (10 п.)	Тест 4 (10 п.)	Тест вкупно (%)	Просечна оценка за двете тромесечја
Ученик 1	7,14	10	10	8,33	88,675	5
Ученик 2	10	8,571	10	6,67	88,1025	5
Ученик 3	8,57	5,714	7	10	78,21	4,5
Ученик 4	10	5,714	9	10	86,785	5
Ученик 5	7,14	4,286	9	10	76,065	4,5
Ученик 6	5,71	7,143	9	6,67	71,3075	4
Ученик 7	10	10	-	8,33	94,4333	5
Ученик 8	7,14	8,571	9	5	74,2775	4
Ученик 9	10	10	10	10	100	5
Ученик 10	8,57	10	9	10	93,925	5
Ученик 11	8,57	7,143	8	8,33	80,1075	4,5
Ученик 12	8,57	10	9	8,33	89,75	4,5
Ученик 13	10	7,143	10	10	92,8575	5
Ученик 14	4,29	7,143	4	1,67	42,7575	2
Ученик 15	1,43	1,429	3	6,67	31,3225	1,5
Ученик 16	5,71	8,571	10	5	73,2025	4
Ученик 17	2,86	0	5	5	32,15	1,5
Ученик 18	8,57	5,714	7	10	78,21	4
Ученик 19	10	10	10	8,33	95,825	5
Ученик 20	2,86	-	5	0	26,2	2
Ученик 21	10	7,143	8	6,67	79,5325	5
Ученик 22	7,14	5,714	4	6,67	58,81	2,5
Ученик 23	8,57	7,143	9	5	74,2825	4
Ученик 24	4,29	2,857	3	1,67	29,5425	1,5
Ученик 25	10	8,571	9	10	93,9275	4
Ученик 26	0	0	0	5	12,5	1,5

Табела 5.3: Резултати од тестовите (генерирани со користење на CodeCPP) и просечни оценки за разгледуваните две тромесечја, за секој од учениците што учествуваа во експерименталната активност (Експеримент 3)



Слика 5.9: Графички приказ на просечниот број на освоени поени на тестовите наспроти просечната оценка за разгледуваните тромесечја, во Експеримент 3

3. Учениците добиваат различни тестови, а уште попрецизно – комплетно различни прашања. Тоа му овозможило на наставникот уште при првото тестирање да утврди не само кој ученик се обидел да препишува, туку и кој ученик бил изворот на решенијата (со оглед на тоа дека во услови на онлајн настава е полесно да се споделуваат решенија меѓу учениците). Ова овозможило учениците уште после првиот тест да бидат обесхрабрени да се обидуваат да препишуваат на следните тестови.
4. Прикажувањето на кодот од прашањето во облик на слика е исто така одлична заштита од мамење, со тоа што се оневозможува ученикот да го копира кодот и да се обиде да го изврши локално, користејќи компајлер.
5. Наставникот е сигурен дека ќе го користи системот и во следната учебна година, и планира да спроведе и уште поголем број на тестови.

Согледувањата на учениците може да се резимираат со следните неколку заклучоци:

1. Учениците генерално сакаат активности какви што се онлајн тестовите, бидејќи добиваат резултат веднаш по завршувањето на тестот.
2. Тие се задоволни што имале повратна информација за нивното моментално знаење преку спроведување на тестовите.
3. Учениците сметаат дека тестовите биле фер бидејќи прашањата за секој од учениците биле практично исти, односно со иста меѓусебна „тежина“.
4. Подобрите ученици се особено задоволни и од неможноста за мамење на тестот, и од објективноста на резултатите.

5.4.3. Заклучок од експериментот

Земајќи ги предвид погоре презентираните резултати и согледувања, може да се заклучи дека CodeCPP е одлично прифатена алатка, како од наставникот така и од учениците, и дека може да се очекува нејзина понатамошна голема примена во образовниот процес.

Со цел да се добие потврда за погорните заклучоци, беше спроведен и Експериментот 4 опишан во продолжение.

5.5. Експеримент 4

5.5.1. Опис на експериментот

Со цел да се добие стручна оценка за корисноста на Moodle додатокот CodeCPP за паметно автоматско генерирање на прашања што содржат програмски код, како и да се утврди ставот на наставниот кадар во врска со употребливоста и евентуалните придобивки и/или недостатоци од користењето на CodeCPP во процесот на проверка на знаење, беше организирана обука и спроведена анкета за наставници од основните и средните училишта во нашата држава. На оваа обука беше изведена практична илустрација за начинот на употреба на CodeCPP за креирање (и конфигурирање) на соодветни прашања што содржат програмски код и нивно вклучување во квиз на платформата Moodle. Обуката беше спроведена во месец јуни 2021 година, а во неа учествуваа одбрани наставници со повеќегодишно искуство (5 и повеќе години) во предавање на воведни и напредни курсеви за програмирање за ученици од основното и средното образование. Со оглед на актуелната состојба кај нас (па и во светски размери) со пандемијата на COVID-19, целиот образовен процес (на училишно ниво) во текот на претходната учебна година (2020/2021) се одвиваше во онлајн услови, користејќи ја националната платформа за учење на далечина, па според тоа, сите наставници веќе имаа искуство со креирање и користење на онлајн квизови/тестови.

Во рамки на обуката, прво беше илустриран процесот на креирање на квиз на платформата Moodle, како и креирањето на неколку основни типови на прашања (ПМО прашања, точно/неточно прашања, есејски прашања) и нивното вклучување како дел од одреден квиз. Потоа беше прикажана и постапката за внесување на прашање од типот CodeCPP, односно на шаблон за генерирање на поголем број прашања со иста или слична сложеност, како и конфигурирањето на доменот на вредности за различните типови на локации од интерес што може да се појават во еден шаблонски код. Исто така, на наставниците им беше објаснето дека со зачувувањето на дадено прашање (шаблон) истовремено во позадина отпочнува процесот на генерирање на поголем број на нови прашања (верзии на шаблонот), чии што програмски кодови претставуваат кодни

варијации добиени од внесениот шаблонски код со соодветна мутација на локациите од интерес – според зададената конфигурација за секоја од нив. Наставниците беа информирани дека секој ученик за време на тестирање ќе добива точно една од овие верзии на шаблонот, односно една од кодните варијации на шаблонскиот код (избрана на случаен начин) со контролирана сложеност (преку соодветна прагова вредност). Обуката беше комплетирана со практична активност: секој од наставниците имаше можност да ја испроба функционалноста на CodeCPP, на експериментален Moodle курс кој беше креиран специјално за потребите на обуката.

По завршувањето на обуката, од наставниците беше побарано да пополнат соодветна анкета во која ќе ги искажат своите ставови, размислувања и оценки за употребливоста на CodeCPP во наставата по програмирање, како и за продобивките и/или недостатоците од евентуалното користење во наставниот процес. Прашањата што ја сочинуваа анкетата се претставени во целост во Додаток В.

Првите 4 прашања (П1 – П4) се однесуваа на искуството со предавање на курсеви за програмирање, искуството со користењето на платформата Moodle воопшто, како и специфично – за искуството со користење на Moodle квизови. Од сите анкетирани наставници, најголем дел (41,66%) имаат искуство од 20 или повеќе години во држење на настава по програмирање, 25% се со искуство помеѓу 10 и 15 години, 16,67% имаат искуство од 15 до 20 години, и исто толку (16,67%) се со искуство помеѓу 5 и 10 години. Во однос на платформата Moodle, 91,67% од испитаниците се изјаснија дека претходно ја користеле како слушатели на курс, 75% ја користеле како наставници на курс, а 83,33% имале претходно искуство со креирање на Moodle квиз.

Останатите прашања од анкетата (П5 – П12) беа целосно ориентирани кон користењето на CodeCPP.

5.5.2. Резултати и анализа

Повратните информации добиени од испитаниците, како одговори на прашањата во врска со CodeCPP во рамките на анкетата, беа исклучително позитивни. Овој заклучок го потврдува и графиконот претставен на Слика 5.10. Сите испитаници (100%) целосно се согласуваат дека CodeCPP е соодветен за користење во услови на онлајн настава (прашање П11), дека овој Moodle додаток ќе им помогне во реализацијата на наставата по програмирање (П5), како и дека истиот може да се користи за испорака на повратни информации до учениците околу нивниот прогрес во изучувањето на програмирањето (П6). Околу примената во процесот на проверка на знаењето од програмирање на учениците, 91,67% целосно се согласуваат дека CodeCPP може да се користи за проверка на знаење од програмирање, а останатите 8,33% ја одбраа опцијата

„Делумно се согласувам“ (ниту еден наставник не ги одбра опциите „Делумно не се согласувам“ и „Воопшто не се согласувам“!) на ова анкетно прашање (П7). Значајно голем дел од испитаниците (83,33%) исто така целосно се согласуваат дека CodeCPP би помогнал да се намали препишувањето како појава за време на испитување на учениците (останатите 16,67% се согласуваат делумно, П10). Околу честотата на користење во наставата (П8), 41,67% од испитаниците се изјасниња дека CodeCPP би го користеле на секој трети наставен час, по 25% од испитаниците би го користеле секој петти наставен час или пак еднаш месечно, а 8,33% би го користеле на секој втор наставен час. Анкетата содржеше и прашање (П9) во врска со времето коешто би им било потребно на испитаниците (според нивна лична проценка) за да подготват еден Moodle квиз со 5 различни прашања од тип CodeCPP (5 различни шаблони). Половина од нив (50%) сметаат дека би потрошиле до 10 минути на подготовката на ваков квиз, една третина (33,33%) сметаат дека би потрошиле од 10 до 20 минути, а само 16,67% се изјасниња дека би им бил потребен временски период во рангот од 20 до 30 минути.

Слика 5.10: Ставови на наставниците во врска со употребата на CodeCPP во настава по програмирање

Позитивниот став на наставниците во врска со користењето на предложениот Moodle додаток во наставата по програмирање се согледува и од одговорите што ги доставија на последното прашање од анкетата (П12). Ова прашање беше есејско и побаруваше од нив да опишат некои придобивки за кои сметаат дека би ги овозможил CodeCPP (а кои не се спомнати во ниту едно од претходните анкетни прашања), како и да споделат свои размислувања и коментари околу неговата употреба. Некои од размислувањата споделени од наставниците беа дека CodeCPP „е корисен за наставниците бидејќи тие најчесто имаат голем број на ученици, па тој ќе им ја олесни многу работата“, како и дека овозможува „индивидуалност во работата на учениците“. Во врска со проверката на знаење од програмирање преку прашања од тип CodeCPP, некои од ставовите се дека „ова е еден нов и интересен начин на оценување на

учениците, што овозможува заштеда на време и поголема флексибилност при формирање на оценка на учениците од страна на наставникот“, односно дека „со ова се овозможува соодветно оценување и реално проверување на знаењата на учениците, без да се одзема многу време на наставникот за подготовка на голем број различни прашања“. Интересен е коментарот на еден наставник, кој вели: „Одлично е. Ми се допадна фактот што прашањата се генерираат како слика. Секако, ми се допаѓа и тоа што се прават различни прашања за секој ученик и со тоа се мотивираат учениците самостојно да размислуваат“. Конечно, покрај задоволството, наставниците ја искажаа и својата љубопитност и нетрпеливост да започнат со користење на овој пристап во наставата. Еден наставник даде краток коментар: „Ми недостигаше нешто вакво!!“, друг надополнува дека „Ова е алатка која многу ќе ја користиме“, а трет инсистира „да се овозможи заинтересираните наставници да ја користат оваа одлична алатка на LMS“.

5.5.3. Заклучок од експериментот

Резултатите и од овој експеримент ја потврдуваат и нагласуваат вредноста на CodeCPP како додаток за Moodle базиран на истражувањето претставено во оваа докторска дисертација.

6. Заклучок и идна работа

Компјутерските науки воопшто, а програмирањето (како нивен суштински дел) особено, се многу популарни во денешно време. Како илустрација, Асоцијацијата за компјутерски истражувања (англ. *Computing Research Association* – CRA) известува за драматично зголемување на бројот на уписи на академски институции од областа на компјутерските науки во периодот од 2006-та до 2017-та година во САД: 60% од институциите кои учествувале во истражувањето имале повеќе од двојно зголемен број на уписи (Guzidal, 2017). Овој тренд се предвидува дека ќе продолжи и во годините што следуваат (*NASEM*, 2018). Програмирањето е присутно како задолжителен предмет во секоја наставна програма од областа на компјутерските науки, па според тоа, обично голем број на студенти по компјутерски науки запишуваат курсеви за програмирање. Како последица, наставниците на овие курсеви мораат да се соочуваат со повеќе предизвици, а еден од најзначајните помеѓу нив е проверката на знаењето на студентите.

Проверката на знаење е исклучително важен дел од процесот на едукација. Еден погоден пристап за оценување на базично знаење кај воведни курсеви за програмирање е преку претставување на програмски код и прашање во врска со неговото однесување, од обликот „Кој е излезот од дадениот код?“. Главни квалитети на процесот на проверка на знаење на конкретен испит од некој курс се неговата објективност и праведност. Но, за да се одржат овие квалитети, наставниците на воведните курсеви за програмирање мораат да дизајнираат прашања што содржат програмски кодови со иста или слична сложеност (исто ниво на знаење и приближно исто потребно време за решавање) за сите студенти, што претставува огромен предизвик кога се работи со големи студентски групи. Овој проблем може да се реши преку употреба на автоматско генерирање на вакви прашања, но за таа цел неопходен е начин за автоматско мерење на сложеноста на програмските кодови.

Во литературата за мерење на софтверска сложеност се среќаваат мноштво од метрики, кои користат различни пристапи и разгледуваат различни аспекти од сложеноста на програмскиот код. Во Глава 2 е изложен преглед на некои најзначајни и најчесто применувани метрики за мерење на софтверска сложеност, кои воедно се најпроминентни претставници на своите категории според аспектот што го разгледуваат. Дискутирани се својствата на метриките, идентификувани се предностите и недостатоците од употребата на секоја од нив, а потоа е анализиран и нивниот потенцијал за примена во едукативна околина, во контекст на проблемот што се разгледува. Врз основа на претставената анализа, заклучено е дека ниту една од опишаните, а и воопшто – ниту една од постојните метрики не може директно да се примени при едукативната проверка на знаење кај курсевите за програмирање.

Главната цел на истражувањето елаборирано во оваа докторска дисертација е да се подобри квалитетот на процесот на проверка на знаење од областа на основните концепти на програмирањето, кај воведните курсеви за програмирање. Следејќи ја оваа цел, во Глава 3 е претставен нов модел за паметно автоматско генерирање на прашања што содржат програмски код. Истиот ги вклучува следните чекори: 1) Креирање на почетен шаблон (прашање што содржи програмски код); 2) Компајлирање, парсирање и детекција на локации од интерес во шаблонскиот код; 3) Пресметување на сложеност на шаблонскиот код; 4) Дефинирање на правила за модификација на шаблонот и дефинирање прагова вредност за сложеноста на кодовите; 5) Формирање на нови прашања преку генерирање на нови програмски кодови од шаблонскиот код. При пресметување на сложеноста на програмските кодови (чекори 3) и 5)), моделот претпоставува користење на соодветна метрика за мерење на сложеност. Затоа, во продолжение е дефинирана нова метрика, која е применлива при имплементацијата на моделот.

За пресметување на сложеност со новата метрика, се претпоставува дека се доделени соодветни (когнитивни) тежински вредности на секој оператор и секоја наредба за контрола на текот од програмскиот јазик во којшто е напишан кодот што се разгледува. Тежинските вредности претставуваат квантитативна репрезентација на „сложеноста“ на секој оператор/наредба, во смисла на времето и напорот потребен да се разбере (когнитивно просецирање) и рачно да се изврши секоја од нив од страна на човек (студент). Според начинот на којшто е дефинирана оваа метрика, лесно се согледува нејзината соодветност за намената.

Врз основа на предложениот модел, имплементиран е и систем за паметно автоматско генерирање на прашања. Користејќи ја предложената метрика за мерење на сложеност, системот овозможува генерирање на нови прашања од зададено почетно прашање (шаблон) што содржи програмски код, пришто сите новодобиени прашања ќе содржат кодови со конзистентна сложеност (произволно блиска сложеност еден во однос на друг). Овој систем, исто така, е опишан во детали во Глава 3.

Заради поедноставна имплементација на предложениот модел во наставниот процес кај воведни курсеви за програмирање на матичната институција, Факултетот за информатички науки и компјутерско инженерство (ФИНКИ) при Универзитетот „Св. Кирил и Методиј“ во Скопје, дополнително е развиен и додаток за популарниот систем за управување со е-учењето со отворен код Moodle што се користи на факултетот. Додатокот овозможува креирање и вклучување на прашање од нов тип, именуван CodeCPP, коешто содржи програмски код, во рамките на кој било Moodle квиз. Врз основа на креираното прашање потоа се генерираат (посакуван број) нови прашања што

содржат програмски кодови со конзистентна сложеност, кои можат да се користат во процесот на тестирање. Описот и функционалните детали за овој систем се претставени во Глава 4.

Целта на спроведувањето на првите два експерименти опишани во Глава 5 е утврдување на соодветни (когнитивни) тежински вредности за аритметичките оператори, коишто би се користеле при пресметување на сложеност на програмските кодови со користење на новата метрика. Резултатите од експериментите елаборирани во оваа глава покажуваат дека постои корелација помеѓу просечните времиња што се потребни за рачно изведување на различни аритметички операции над едноставни операнди (иако постојат и некои исклучоци). Ова дозволува да се утврдат односите помеѓу соодветните операции, а преку тоа – и тежините за секоја од нив. Бидејќи во периодот на реализацијата на експериментите сеуште не беше развиен софтверскиот систем опишан во Глава 3, за потребите на спроведувањето на експериментите во рамките на ова истражување беше развиена дополнителна (помошна) софтверка алатка.

Вторите два експерименти (Експеримент 3 и Експеримент 4) опишани во Глава 5 се спроведени заради евалуација на предложениот софтверски систем, односно додатокот CodeCPP за Moodle, а преку него – и на моделот за паметно автоматско генерирање на прашања дефиниран во рамки на дисертацијата. Целта на оваа евалуација е да утврди степенот на успешност на автоматската проверка на знаење кај воведните курсеви за програмирање, при примена на овој модел. Евалуацијата вклучуваше едногодишно користење на системот во настава по програмирање од една страна, како и повратен одговор од избрани наставници обучени за користење на системот од друга страна. Резултатите од евалуацијата со употреба на системот во наставата потврдуваат дека со примена на моделот за задавање на онлајн тестови составени од прашања со конзистентна сложеност се овозможува фер и објективна проверка на знаење на кој било испит од курс за програмирање. Исто така, со автоматизацијата на процесот на генерирање на прашања, ваквите тестови може да се изведуваат многу почесто во тек на даден курс, со што се овозможува студентите да добиваат континуирана повратна информација за нивниот прогрес во учењето на програмирањето. Од друга страна, повратните информации добиени од наставниците со долгогодишно искуство во предавање курсеви за програмирање, на анкетата (претставена во оваа глава) за употребливоста на CodeCPP во наставата по програмирање, како и за продобивките и/или недостатоците од користењето во наставниот процес, се исклучително позитивни.

Потенцијалот на моделот за паметно автоматско генерирање на прашања предложен во оваа докторска дисертација е голем. Моделот може да се примени

во најразлични апликации, како што е креирање и евалуација на нови софтверски системи кои го имплементираат моделот, со различна намена. Софтверските системи би можеле да се користат за учење на програмски јазик (јазици) преку генерирање на голем број соодветни прашања за самопроверка на стекнатото знаење од страна на студентите, или пак за спроведување на проверка на знаењето на студентите во рамките на одреден курс за програмирање – од страна на наставниците. Примените кои овој модел ги остава отворени за идните истражувачи се огромни.

Како насока за понатамошно истражување, секако, очигледно се наметнува потребата од дизајнирање и спроведување на дополнителни експерименти со цел утврдување на соодветни тежински вредности за останатите типови на оператори (релациски, логички), како и за наредбите за контрола на текот од програмскиот јазик C/C++ (`if`, `switch`, `while`, `do-while`, `for`). Иако некои автори, како на пример Wang (2006), веќе имаат предложено одредени експериментално утврдени вредности за когнитивните тежини на различните контролни структури кај програмските јазици воопшто, сепак не постои консензус во врска со нив помеѓу експертите од оваа област и сеуште не се дефинирани задоволителни вредности. Според тоа, спроведувањето на вакво поопсежно истражување, кое притоа ќе ги земе во предвид согледувањата, заклучоците и сугестиите кои се изложени во литературата од претходните истражувања, би претставувало одличен предизвик.

Исто така, друга можна насока за истражување би била разгледување на можностите за проширување на системот (а воедно и Moodle додатокот CodeCPP) за паметно автоматско генерирање на прашања што содржат програмски код, претставен во дисертацијата. На пример, едно проширување кое потенцијално би можело да се разгледа е додавањето на нови начини (пристапи) за модификација (мутација) на даден код, како и повеќе опции за контрола на актуелниот процес на мутација при генерирањето нови прашања. Дополнително, може да се воведат опции за конфигурирање на одделни тежински вредности за операциите кога се применуваат над различни типови на операнди (едноцифрени, двоцифрени, итн.), со што системот би ја пресметувал сложеноста на кодовите применувајќи ги соодветните тежини за секој оператор – земајќи ги во предвид и операндите. Конечно, системот би можел да се прошири и преку воведување на поддршка за генерирање на прашања и за некои други програмски јазици.

Референци

Ackovska, N., Erdosne Nemeth, A., Stankov, E., Jovanov, M. (2015). *Report of the IOI workshop "Creating an International Informatics Curriculum for Primary and High School Education"*. Journal Olympiads in Informatics, ISSN 1822-7732, vol. 9, pp. 205–212.

Ala-Mutka, K. (2005). *A survey of automated assessment approaches for programming assignments*. Computer Science Education, vol. 15, no. 2, pp. 83–102.

Alderson, J. C. (2000). *Technology in testing: The present and the future*. J. System, vol. 28, no. 4, pp. 593–603.

Al-Amri, S. (2008). *Computer-based testing vs. paper-based testing: a comprehensive approach to examining the comparability of testing modes*. In: Essex Graduate Student Papers in Language & Linguistics 10, pp. 22–44.

Al-Smadi, M., Gütl, C. (2008). *Past, present and future of e-assessment: Towards a flexible e-assessment system*. In: Proceedings of the ICL2008 Conference, Villach, Austria, pp. 1–8.

Anders Ericsson, K., Hoffman, R. R., Kozbelt, A., Williams, A. M. (2018). *The Cambridge handbook of expertise and expert performance, 2nd Edition*. Cambridge University Press.

Angelovski, D., Stankov, E., Jovanov, M. (2021). *DEMAx tool based on an improved model for semiautomatic C/C++ source code assessment*. In: Proceedings of the 6th International Conference on Information and Education Innovations (ICIEI 2021) (in print).

Armenski, G. (2017). *Virtual networks for professional development of vocational teachers and trainers in Macedonia*. Open Space ETF report, https://openspace.etf.europa.eu/wikis/macedonia-virtual-networks-cpd, сајтот е пристапен на 23.06.2021.

Arnow, D., Barshay, O. (1999). *On-line programming examinations using WebToTeach*. In: Proceedings of the 4th Annual SIGCSE/SIGCUE Conference on Innovation and Techology in Computer Science Education (ITiCSE '99), pp. 21–24.

Basili, V. R. (1980). *Qualitative software complexity models: A summary*. In: Tutorial on Models and Methods for Software Management and Engineering, pp. 158–162. Los Alamitos, CA, USA: IEEE Computer Society Press.

Basili, V. R., Phillips, T. (1981). *Evaluating and comparing software metrics in the software engineering laboratory*. In: Proceedings of the 1981 ACM Workshop/Symposium on Measurement and Evaluation of Software Quality, pp. 95–106. New York, NY, USA: ACM Press.

Benford, S. D., Burke, E. K., Foxley, E., Gutteridge, N., Zin, M. A. (1994). *Ceilidh as a course management support system*. Journal of Educational Technology Systems, vol. 22, no. 3, pp. 235–250.

Biggs, J. B. (2003). *Teaching for quality learning at university*. UK: Open University Press.

Bloom, B. S., Engelhart, M. D., Furst, E. J., Hill, W. H., Krathwohl, D. R. (1956). *Taxonomy of educational objectives: The classification of educational goals, Handbook I: Cognitive domain*. New York, NY, USA: David McKay Company.

Brandl, K. (2005). *Are you ready to "Moodle"?*. Language Learning & Technology Journal, vol. 9, no. 2, pp. 16–23.

Bridgeman, S., Goodrich, M. T., Kobourov, S. G., Tamassia, R. (2000). *PILOT: An interactive tool for learning and grading*. In: ACM SIGCSE Bulletin, vol. 32, no. 1, pp. 139–143. New York, NY, USA: ACM Press.

Brusilovsky, P., Sosnovsky, S. (2005). *Individualized exercises for self-assessment of programming knowledge: An evaluation of QuizPACK*. ACM Journal on Educational Resources in Computing, vol. 5, no. 3, pp. 6–es.

Chatzopoulou, D. I., Economides, A. A. (2010). *Adaptive assessment of student's knowledge in programming courses*. Journal of Computer Assisted Learning, vol. 26, no. 4, pp. 258–269. Wiley-Blackwell, 2010.

Chitti Babu, P., Prasad, A. N., Sudhakar, D. (2013). *Software complexity metrics: a survey*. International Journal of Advanced Research in Computer Science and Software Engineering, vol. 3, no. 8, pp. 1359–1362.

Clang (2007), проект којшто обезбедува преден дел за LLVM компајлерите за програмски јазици од фамилијата на јазикот C. https://clang.llvm.org/, сајтот е пристапен на 22.06.2021.

Clark, D. (2004). *Testing programming skills with multiple choice questions*. Journal Informatics in Education, vol. 3, no. 2, pp. 161–178.

Costello, E. (2013). *Opening up to open source: looking at how Moodle was adopted in higher education*. Open Learning: The Journal of Open, Distance and E-Learning, vol. 28, no. 3, pp. 187–200.

Cunningham, K., Blanchard, S., Ericson, B., Guzidal, M. (2017). *Using tracing and sketching to solve programming problems: Replicating and extending an analysis of what students draw*. In: Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER '17), pp. 164–172.

Curtis, B., Sheppard, S. B., Milliman, P., Borst, M. A., Love, T. (1979). *Measuring the psychological complexity of software maintenance tasks with the*

Halstead and McCabe metrics. IEEE Transactions on Software Engineering, vol. SE-5, no. 2, pp. 96–104. IEEE, 1979.

Daly, C. (1999). *RoboProf and an introductory computer programming course*. In: ACM SIGCSE Bulletin, vol. 31, no. 3, pp. 155–158. New York, NY, USA: ACM Press.

Daly, C., Waldron, J. (2004). *Assessing the assessment of programming ability*. In: Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education, pp. 210–213. New York, NY, USA: ACM Press.

Dancik, G., Kumar, A. N. (2003). *A tutor for counter-controlled loop concepts and its evaluation*. In: Proceedings of the 33rd Annual Frontiers in Education Conference (FIE '03), vol. 1, pp. T3C–7. IEEE, 2003.

Deb, D., Fuad, M. M., Kanan, M. (2017). *Creating engaging exercises with mobile response system (MRS)*. In: Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education, pp. 147–152. New York, NY, USA: ACM Press.

Delev, T., Gjorgjevikj, D. (2012). *E-Lab: web based system for automatic assessment of programming problems*. In: Web Proceedings of the ICT Innovations Conference 2012, pp. 75–83.

Dimitrievska Ristovska, V., Stankov, E., Sekuloski, P. (2021). *Teaching and examination process of some university courses before vs during the corona crisis*. Journal Olympiads in Informatics, ISSN 1822-7732, vol. 15, pp. 91–104.

Dorodchi, M., Dehbozorgi, N., Frevert, T. K. (2017). "*I wish I could rank my exam's challenge level!*": *An algorithm of Bloom's taxonomy in teaching CS1*. In: Proceedings of the 2017 IEEE Frontiers in Education Conference (FIE '17), pp. 1–5.

Dougiamas, M., Taylor, P. C. (2003). *Moodle: Using learning communities to create an open source course management system*. In: Proceedings of the 15th World Conference on Educational Multimedia, Hypermedia & Telecommunications (ED-MEDIA 2003), pp. 171–178.

Drasgow, F. (2002). *The work ahead: A psychometric infrastructure for computerized adaptive tests*. In: Mills, C. N., Potenza, M. T., Fremer, J. J., Ward, W. C. (Eds.), Computer-based Testing: Building the Foundation for Future Assessments, pp. 67–88. Hillsdale, NJ, USA: Lawrence Erlbaum, 2002.

Elshoff, J. (1978). *An investigation into the effects of the counting method used on software science measurement*. In: ACM SIGPLAN Notices, vol. 13, no. 2, pp. 30–45. New York, NY, USA: ACM Press, 1978.

Ericson, B. J., Margulieux, L. E., Rick, J. (2017). *Solving parsons problems versus fixing and writing code*. In: Proceedings of the 17th Koli Calling International Conference on Computing Education Research, pp. 20–29.

Fei, Y. Y., Zhi, Z., Chao, Z. S. (2004). *Improvements about Halstead model in software science*. Journal of Computer Applications, pp. 130–132. Sichuan, China: Chinese Academy of Sciences.

Fernandes, E., Kumar, A. N. (2004). *A tutor on scope for the programming languages course*. In: ACM SIGCSE Bulletin, vol. 36, no. 1, pp. 90–93. New York, NY, USA: ACM Press.

Flask (2010), рамка за развој на веб апликации. https://palletsprojects.com/p/flask/, сајтот е пристапен на 22.06.2021.

Forsythe, G. E., Wirth, N. (1965). *Automatic grading programs*. Communications of the ACM, vol. 8, no. 5, pp. 275–278.

GCC – GNU Compiler Collection (1987), колекција од алатки за компајлирање на повеќе програмски јазици. https://gcc.gnu.org/, сајтот е пристапен на 22.06.2021.

Gagne, R. M. (1985). *The conditions of learning* (*4th edition*). New York, NY, USA: Holt, Rinehart & Winston.

Garner, S. (2002). *Reducing the cognitive load on novice programmers*. In: Proceedings of the 14th World Conference on Educational Multimedia, Hypermedia & Telecommunications (ED-MEDIA 2002), pp. 578–583.

Gluga, R., Kay, J., Lister, R., Kleitman, S., Lever, T. (2012). *Coming to terms with Bloom: an online tutorial for teachers of programming fundamentals*. In: Proceedings of the 14th Australasian Conference on Computing Education, vol. 123, pp. 147–156. New York, NY, USA: ACM Press.

Gotel, O., Scharff, C. (2007). *Adapting an open-source web-based assessment system for the automated assessment of programming problems*. In: Proceedings of the 6th International IASTED Web-based Education Conference, pp. 437–442.

Gruhn, V., Laue, R. (2007). *On experiments for measuring cognitive weights for software control structures*. In: Proceedings of the 6th IEEE International Conference on Cognitive Informatics (ICCI 2007), pp. 116–119. IEEE, 2007.

Guzidal, M. (2017). *Generation CS' drives growth in enrollments*. In: Communications of the ACM, vol. 60, no. 7, pp. 10–11.

Gyll, S., Ragland, S. (2018). *Improving the validity of objective assessment in higher education: Steps for building a best-in-class competency-based assessment program*. The Journal of Competency-Based Education, vol. 3, no. 1.

Halstead, M. H. (1977). *Elements of software science*. New York, NY, USA: Elsevier North-Holland.

Henderson-Sellers, B. (1992). *Modularization and McCabe's cyclomatic complexity*. In: Communications of the ACM, vol. 35, no. 12, pp. 17–19.

Henderson-Sellers, B., Tegarden, D. (1994). *The theoretical extension of two versions of cyclomatic complexity to multiple entrylexit modules*. Software Quality Journal, vol. 3, pp. 253–269.

Higgins, C. A., Hegazy, T., Symeonidis, P., Tsintsifas, A. (2003). *The CourseMarker CBA system: Improvements over Ceilidh*. Education and Information Technologies Journal, vol. 8, pp. 287–304.

Hill, P. (2017). *Academic LMS market share: A view across four global regions*. In: e-Literate [онлајн], објавено на 29.06.2017. https://eliterate.us/academic-lms-market-share-view-across-four-global-regions/, сајтот е пристапен на 22.06.2021.

Hoffman, D. M., Lu, M., Pelton, T. (2011). *A web-based generation and delivery system for active code reading*. In: Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE '11), pp. 483–488. New York, NY, USA: ACM Press.

Horvat, A., Dobrota, M., Krsmanovic, M., Cudanov, M. (2015). *Student perception of Moodle learning management system: a satisfaction and significance analysis*. Interactive Learning Environments, vol. 23, no. 4, pp. 515–527.

Hsiao, I.-H., Brusilovsky, P., Sosnovsky, S. (2008). *Web-based parameterized questions for object-oriented programming*. In: Proceedings of E-Learn 2008 – the World Conference on E-Learning in Corporate, Government, Healthcare, and Higher Education, pp. 3728–3735. San Diego, CA, USA: Association for the Advancement of Computing in Education (AACE).

Hsiao, I.-H., Sosnovsky, S., Brusilovsky, P. (2010). *Guiding students to the right questions: Adaptive navigation support in an e-learning system for Java programming*. Journal of Computer Assisted Learning, vol. 26, no. 4, pp. 270–283.

Ilijoski, B., Popeska, Z., Stankov, E. (2018). *Creating effective quizzes by balancing the distribution of question types*. In: Proceedings of the 12th annual International Technology, Education and Development Conference (INTED 2018), pp. 6631–6637.

Izu, C., Schulte, C., Aggarwal, A., Cutts, Q., Duran, R., Gutica, M., Heinemann, B., Kraemer, E., Lonati, V., Mirolo, C., Weeda, R. (2019). *Fostering program comprehension in novice programmers – learning activities and learning trajectories*. In: Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education (ITiCSE – WGR '19), pp. 27–52.

Jackson, D., Usher, M. (1997). *Grading student programs using ASSYST*. In: Proceedings of the 28th SIGCSE Technical Symposium on Computer Science Education, pp. 335–339. New York, NY, USA: ACM Press.

Jakhar, A. K., Rajnish, K. (2014). *A new cognitive approach to measure the complexity of software*. International Journal of Software Engineering and Its Applications *(IJSEIA)*, vol. 8, no. 7, pp. 185–198.

JetBrains Mono (2020), a free and open source typeface for developers. https://www.jetbrains.com/lp/mono/, сајтот е пристапен на 25.06.2021.

Jian-hua, Z., Jia-pei, W. (2006). *Research of the method of measuring program complexity based on pseudo-path*. Journal of Zhongkai Agrotechnical College, pp. 42–45. Guangdong, China: Zhongkai Agrotechnical College.

Jovanov, M., Ackovska, N., Stankov, E., Mihova, M., Gusev, M. (2017a). *A decade of engineering computer engineers*. In: Proceedings of the 2017 IEEE Global Engineering Education Conference (EDUCON 2017), pp. 1309–1315.

Jovanov, M., Ilijoski, B., Stankov, E., Armenski, G. (2017b). *Creation of educational games – project based learning in e-learning systems course*. In: Proceedings of the 2017 IEEE Global Engineering Education Conference (EDUCON 2017), pp. 1274–1281.

Jovanov, M., Kostadinov, B., Stankov, E. (2010). *A new design of a system for contest management and grading in informatics competitions*. In: Web Proceedings of the ICT Innovations Conference 2010, pp. 87–96.

Jovanov, M., Stankov, E., Mihova, M., Ristov, S., Gusev, M. (2016). *Computing as a new compulsory subject in the Macedonian primary schools curriculum*. In: Proceedings of the 2016 IEEE Global Engineering Education Conference (EDUCON 2016), pp. 680–685.

Joy, M., Griffiths, N., Boyatt, R. (2005). *The BOSS online submission and assessment system*. ACM Journal on Educational Resources in Computing, vol. 5, no. 3, pp. 2–es.

Keles, M. K., Ozel, S. A. (2016). *A review of distance learning and learning management systems*. In: Virtual Learning, Intech Open [онлајн]. https://www.intechopen.com/books/virtual-learning/a-review-of-distance-learning-and-learning-management-systems, сајтот е достапен на 25.06.2021

Khoshsima, H., Hashemi Toroujeni, S. M. (2017). *Technology in education: Pros and cons of using computer in testing domain*. International Journal of Language Learning and Applied Linguistics World, vol. 14, no. 2, pp. 32–49.

Korhonen, A., Malmi, L. (2000). *Algorithm simulation with automatic assessment*. In: Proceedings of the 5th Annual SIGCSE/SIGCUE Conference on

Innovation and Technology in Computer Science Education (ITiCSE ’00), pp. 160–163.

Kostadinov, B., Jovanov, M., Stankov, E. (2018). *Platform for analysing and encouraging student activity on contest and e-learning Systems*. Journal Olympiads in Informatics, ISSN 1822-7732, vol. 12, pp. 85–98.

Kostadinov, B., Jovanov, M., Stankov, E., Mihova, M., Risteska Stojkoska, B. (2015). *Different approaches for making the initial selection of talented students in programming competitions*. Journal Olympiads in Informatics, ISSN 1822-7732, vol. 9, pp. 113–125.

Krebs, M., Lauer, T., Ottmann, T., Trahasch, S. (2005). *Student-built algorithm visualizations for assessment: flexible generation, feedback and grading*. In: ACM SIGCSE Bulletin, vol. 37, no. 3, pp. 281–285. New York, NY, USA: ACM Press.

Kumar, A. N. (2005). *Generation of problems, answers, grade, and feedback – case study of a fully automated tutor*. ACM Journal on Educational Resources in Computing, vol. 5, no. 3, pp. 3–es.

Kumar, A. N. (2015). *Solving code-tracing problems and its effect on code-writing skills pertaining to program semantics*. In: Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE ’15), pp. 314–319.

Kurdi, G., Leo, J., Parsia, B., Sattler, U., Al-Emari, S. (2019). *A systematic review of automatic question generation for educational purposes*. International Journal of Artificial Intelligence in Education, vol. 30, pp. 121–204.

Kushwaha, D. S., Misra, A. K. (2006). *A modified cognitive information complexity measure*. In: ACM SIGSOFT Software Engineering Notes, vol. 31, no. 1, pp. 1–4. New York, NY, USA: ACM Press.

Lajis, A., Nasir, H. M., Aziz, N. A. (2018). *Proposed assessment framework based on Bloom taxonomy cognitive competency: Introduction to programming*. In: Proceedings of the 2018 7th International Conference on Software and Computer Applications (ICSCA 2018), pp. 97–101.

Lister, R. (2000). *On blooming first year programming, and its blooming assessment*. In: Proceedings of the Australasian Conference on Computing Education (ACSE ’00), pp. 158–162. New York, NY, USA: ACM Press.

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J. E., Sanders, K., Seppälä, O., Simon, B., Thomas, L. (2004). *A multi-national study of reading and tracing skills in novice programmers*. In: ACM SIGCSE Bulletin, vol. 36, no. 4, pp. 119–150. New York, NY, USA: ACM Press.

Lister, R., Clear, T., Simon, Bouvier, D. J., Carter, P., Eckerdal, A., Jacková, J., Lopez, M., McCartney, R., Robbins, P., Seppälä, O., Thompson, E. (2010). *Naturally occurring data as research instrument: analyzing examination responses to study the novice programmer*. In: ACM SIGCSE Bulletin, vol. 41, no. 4, pp. 156–173. New York, NY, USA: ACM Press.

Lister, R., Fidge, C., Teague, D. (2009). *Further evidence of a relationship between explaining, tracing and writing skills in introductory programming*. In: ACM SIGCSE Bulletin, vol. 41, no. 3, pp. 161–165. New York, NY, USA: ACM Press.

Lister, R., Leaney, J. (2003). *Introductory programming, criterion-referencing, and Bloom*. In: Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education, pp. 143–147. New York, NY, USA: ACM Press.

Lopez, M., Whalley, J., Robbins, P., Lister, R. (2008). *Relationships between reading, tracing and writing skills in introductory programming*. In: Proceedings of the 4th International Workshop on Computing Education Research (ICER '08), pp. 101–112.

MUC (2012), универзален кеш на Moodle воведен со верзијата Moodle 2.4. https://docs.moodle.org/dev/The_Moodle_Universal_Cache_(MUC), сајтот е пристапен на 23.06.2021.

Madi, A., Zein, O. K., Kadry, S. (2013). *On the improvement of cyclomatic complexity metric*. International Journal of Software Engineering and Its Applications, vol. 7, no. 2, pp. 67–82.

Maggiolo, S., Mascellani, G. (2012). *Introducing CMS: a contest management system*. Journal Olympiads in Informatics, ISSN 1822-7732, vol. 6, pp. 86–99.

Malmi, L., Karavirta, V., Korhonen, A., Nikander, J. (2005). *Experiences on automatically assessed algorithm simulation exercises with different resubmission policies*. ACM Journal on Educational Resources in Computing, vol. 5, no. 3, pp. 7–es.

Maravić Čisar, S., Pinter, R., Radosav, D., Čisar, P. (2011). *Effectiveness of program visualization in learning Java: A case study with Jeliot 3*. International Journal of Computers Communications & Control, vol. 6, no. 4, pp. 668–680.

Matthiasdottir, A., Arnalds, H. (2016). *E-assessment: students' point of view*. In: Proceedings of the 17th International Conference on Computer Systems and Technologies (CompSysTech '16), pp. 369–374. New York, NY, USA: ACM Press.

McCabe, T. J. (1976). *A complexity measure*. IEEE Transactions on Software Engineering, vol. 2, no. 4, pp. 308–320. IEEE, 1976.

McCracken, M., Almstrum, V., Diaz, D., Guzidal, M., Hagan, D., Kolikant, Y. B.-D., Laxer, C., Thomas, L., Utting, I., Wilusz, T. (2001). *A multi-national, multi-*

institutional study of assessment of programming skills of first-year CS students. In: ACM SIGCSE Bulletin, vol. 33, no. 4, pp. 125–180. New York, NY, USA: ACM Press.

Misra, S. (2007). *Cognitive program complexity measure*. In: Proceedings of the 6th IEEE International Conference on Cognitive Informatics (ICCI 2007), pp. 120–125. IEEE, 2007.

Misra, S., Adewumi, A., Damasevicius, R., Maskeliunas, R. (2017). *Analysis of existing software cognitive complexity measures*. International Journal of Secure Software Engineering, vol. 8, no. 4, pp. 51–71.

Misra, S. C., Bhavsar, V. C. (2003). *Measures of software system difficulty*. In: Software Quality Professional, vol. 5, no. 4, pp. 33–41.

Moodle (2002), систем за управување со учењето. https://docs.moodle.org/, сајтот е пристапен на 23.06.2021.

Moodle Plugins (2021), преглед на актуелните типови на додатоци за Moodle [онлајн]. https://moodle.org/plugins/, сајтот е пристапен на 23.06.2021.

Moodle Releases (2021), хронолошки преглед на сите верзии на Moodle пуштени во употреба [онлајн]. https://docs.moodle.org/dev/Releases, сајтот е пристапен на 23.06.2021.

Moodle Statistics (2021), преглед на статистики за користењето на Moodle [онлајн]. https://stats.moodle.org/, сајтот е пристапен на 23.06.2021.

Myers, G. J. (1977). *An extension to the cyclomatic measure of program complexity*. In: ACM SIGPLAN Notices, vol. 12, no. 10, pp. 61–64. New York, NY, USA: ACM Press, 1977.

NASEM – National Academies of Sciences, Engineering, and Medicine (2018). *Assessing and responding to the growth of computer science undergraduate enrollments*. Washington, DC, USA: The National Academies Press, 2018.

Nicol, D. (2007). *E-assessment by design: using multiple-choice tests to good effect*. Journal of Further and Higher Education, vol. 31, no. 1, pp. 53–64.

O'Leary, C., Lawless, D., Gordon, D., Carroll, D., Mtenzi, F., Collins, M. (2006). *3D alignment in the adaptive software engineering curriculum*. In: Proceedings of the 36th Annual Frontiers in Education Conference, pp. 1–6.

Oviedo, E. I. (1980). *Control flow, data flow, and program complexity*. In: Proceedings of the Fourth International IEEE Computer Software and Applications Conference (COMPSAC 1980), pp. 146–152. IEEE, 1980.

Paris, S. G., Lipson, M. Y., Wixson, K. K. (1983). *Becoming a strategic reader*. Contemporary Educational Psychology, vol. 8, pp. 293–316.

Pearson, K. (1895). *Notes on regression and inheritance in the case of two parents*. In: Proceedings of the Royal Society of London, vol. 58, pp. 240-242. Taylor & Francis, 1895.

Piwowarski, P. (1982). *A nesting level complexity measure*. In: ACM SIGPLAN Notices, vol. 17, no. 9, pp. 44–50. New York, NY, USA: ACM Press, 1982.

Prados, F., Boada, I., Soler, J., Poch, J. (2005). *Automatic generation and correction of technical exercises*. In: Proceedings of the International Conference on Engineering and Computer Education (ICECE ’05).

React (2013), JavaScript библиотека за изградба на кориснички интерфејси. https://reactjs.org/, сајтот е пристапен на 22.06.2021.

Reek, K. A. (1989). *The TRY system or how to avoid testing student programs*. In: Proceedings of the 20th SIGCSE Technical Symposium on Computer Science Education, pp. 112–116. New York, NY, USA: ACM Press.

Rhodes, A., Bower, K., Bancroft, P. (2004). *Managing large class assessment*. In: Proceedings of the 6th Australasian Conference on Computing Education, vol. 30, pp. 285–289. Darlinghurst, Australia: Australian Computer Society.

Ritchie, D. M. (1993). *The development of the C language*. In: ACM SIGPLAN Notices, vol. 28, no. 3, pp. 201–208. New York, NY, USA: ACM Press, 1993.

Rosendorf, T. (2009). *The typographic desk reference*. New Castle, Delaware: Oak Knoll Press, pp. 12.

SQLAlchemy (2006), SQL алатник и релациски пресликувач на објекти за програмскиот јазик Python. https://www.sqlalchemy.org/, сајтот е пристапен на 22.06.2021.

Sagri, M. (1989). *Rated and operating complexity of program – an extension to McCabe’s theory of complexity measure*. In: ACM SIGPLAN Notices, vol. 24, no. 8, pp. 8–12. New York, NY, USA: ACM Press, 1989.

Saikkonen, R., Malmi, L., Korhonen, A. (2001). *Fully automatic assessment of programming exercises*. In: ACM SIGCSE Bulletin, vol. 33, no. 3, pp. 133–136. New York, NY, USA: ACM Press.

Shah, H., Kumar, A. N. (2002). *A tutoring system for parameter passing in programming languages*. In: Proceedings of the 7th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE ’02), pp. 170–174.

Shao, J., Wang, Y. (2003). *A new measure of software complexity based on cognitive weights*. Canadian Journal of Electrical and Computer Engineering, vol. 28, no. 2, pp. 69–74.

Shehab, M. A., Tashtoush, Y. M., Hussien, W. A., Alandoli, M. N., Jararweh, Y. (2015). *An accumulated cognitive approach to measure software complexity*. Journal of Advances in Information Technology, vol. 6, no. 1, pp. 27–33.

Shen, V. Y., Conte, S. D., Dunsmore, H. E. (1983). *Software science revisited: A critical analysis of the theory and its empirical support*. IEEE Transactions on Software Engineering, vol. 9, no. 2, pp. 155–165. IEEE, 1983.

Shuhidan, S., Hamilton, M., D'Souza, D. (2010). *Instructor perspectives of multiple-choice questions in summative assessment for novice programmers*. Computer Science Education, vol. 20, no. 3, pp. 229–259.

Simkin, M. G., Kuechler, W. L. (2005). *Multiple-choice tests and student understanding: what is the connection?*. Decision Sciences Journal of Innovative Education, vol. 3, no. 1, pp. 73–98.

Singhal, N., Kumar, A. N. (2000). *Facilitating problem-solving on nested selection statements using C/C++*. In: Proceedings of the 30th Annual Frontiers in Education Conference, vol. 1, pp. T4C/1-T4C/6. IEEE, 2000.

Spiro, R. J., Jehng, J. C. (1990). *Cognitive flexibility and hypertext: Theory and technology for the nonlinear and multidimensional traversal of complex subject matter*. In: Nix, D., Spiro, R. J. (Eds.), Cognition, Education, and Multimedia: Exploring Ideas in High technology, pp. 163–205. Mahwah, NJ, USA: Lawrence Erlbaum Associates, 1990.

Sprinthall, R. C., Sprinthall, N. A., Oja, S. N. (1998). *Educational psychology, 7th Edition*. McGraw-Hill Education, USA.

Stankov, E., Jovanov, M., Andonov, J., Madevska Bogdanova, A. (2016). *Improving the accuracy of the code complexity calculation for automatically generated tasks with programming codes*. In: Proceedings of the 2016 IEEE Global Engineering Education Conference (EDUCON 2016), pp. 686–692.

Stankov, E., Jovanov, M., Bojchevski, A., Madevska Bogdanova, A. (2013a). *EMAx: Software for C++ source code analysis*. Journal Olympiads in Informatics, ISSN 1822-7732, vol. 7, pp. 123–131.

Stankov, E., Jovanov, M., Gjorgjiev, K., Madevska Bogdanova, A. (2015a). *A new tool for calculation of a new source code metric*. In: Proceedings of the 12th International Conference on Informatics and Information Technology (CiiT 2015), pp. 25–30.

Stankov, E., Jovanov, M., Kostadinov, B., Madevska Bogdanova, A. (2015b). *A new model for collaborative learning of programming using source code similarity detection*. In: Proceedings of the 2015 IEEE Global Engineering Education Conference (EDUCON 2015), pp. 709–715.

Stankov, E., Jovanov, M., Madevska Bogdanova, A. (2013). *Source code similarity detection by using data mining methods*. In: Proceedings of the 35th International Conference on Information Technology Interfaces (ITI 2013), pp. 257–262.

Stankov, E., Jovanov, M., Madevska Bogdanova, A. (2017). *Improved approach for measuring complexity of code snippets for introductory programming tasks*. In: Proceedings of the 10th annual International Conference of Education, Research and Innovation (ICERI 2017), pp. 5892–5899.

Stankov, E., Jovanov, M., Madevska Bogdanova, A., Gusev, M. (2013b). *A new model for semiautomatic student source code assessment*. CIT. Journal of Computing and Information Technology, ISSN 1330-1136, vol. 21, no. 3, pp. 185–194.

Stankov, E., Madevska Bogdanova, A., Ilijoski, B., Jovanov, M. (2018). *A survey on software complexity metrics in the context of their application in educational environment*. In: Proceedings of the 12th annual International Technology, Education and Development Conference (INTED 2018), pp. 9395–9404.

Suleman Sarwar, M. M., Shahzad, S., Ahmad, I. (2013). *Cyclomatic complexity: The nesting problem*. In: Proceedings of the Eighth International Conference on Digital Information Management (ICDIM 2013), pp. 274–279. IEEE, 2013.

Thomas, A., Stopera, T., Frank-Bolton, P., Simha, R. (2019). *Stochastic tree-based generation of program-tracing practice questions*. In: Proceedings of the 50th ACM Technical Symposium on Computer Science Education, pp. 91–97. New York, NY, USA: ACM Press.

Thompson, E., Luxton-Reilly, A., Whalley, J. L., Hu, M., Robbins, P. (2008). *Bloom's taxonomy for CS assessment*. In: Proceedings of the 10th Australasian Conference on Computing Education, vol. 78, pp. 155–161. Darlinghurst, Australia: Australian Computer Society.

Traynor, D., Bergin, S., Paul Gibson, J. (2006). *Automated assessment in CS1*. In: Proceedings of the 8th Australasian Conference on Computing Education, vol. 52, pp. 223–228. Darlinghurst, Australia: Australian Computer Society.

Traynor, D., Paul Gibson, J. (2005). *Synthesis and analysis of automatic assessment methods in CS1: Generating intelligent MCQs*. In: Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education, pp. 495–499. New York, NY, USA: ACM Press.

Tyler, R. W., Gagne, R. M., Scriven, M. (Eds.) (1967). *Perspectives of curriculum evaluation*. American Educational Research Association Monograph Series on Curriculum Evaluation, vol. 1. Chicago, USA: Rand McNally.

Venables, A., Tan, G., Lister, R. (2009). *A closer look at tracing, explaining and code writing skills in the novice programmer*. In: Proceedings of the Fifth International Workshop on Computing Education Research (ICER '09), pp. 117–128.

Wang, Y. (2006). *Cognitive complexity of software and its measurement*. In: Proceedings of the 5th IEEE International Conference on Cognitive Informatics (ICCI 2006), pp. 226–235. IEEE, 2006.

Wang, Y. (2009). *On the cognitive complexity of software and its quantification and formal measurement*. International Journal of Software Science and Computational Intelligence, vol. 1, no. 2, pp. 31–53.

Xiao, Q. (2020). *Using open-source learning platform (Moodle) in university teachers' professional development*. J. Phys.: Conf. Ser. 1646:012036.

Xie, B., Loksa, D., Nelson, G. L., Davidson, M. J., Dong, D., Kwik, H., Hui Tan, A., Hwa, L., Li, M., Ko, A. J. (2019). *A theory of instruction for introductory programming skills*. Computer Science Education, vol. 29, no. 2-3, pp. 205–253.

Yu, S., Zhou, S. (2010). *A survey on metric of software complexity*. In: Proceedings of the 2nd IEEE International Conference on Information Management and Engineering (ICIME), pp. 352–356. IEEE, 2010.

Zavala, L., Mendoza, B. (2018). *On the use of semantic-based AIG to automatically generate programming exercises*. In: Proceedings of the 49th ACM Technical Symposium on Computer Science Education, pp. 14–19. New York, NY, USA: ACM Press.

Zhang, H., Zhang, X., Gu, M. (2007). *Predicting defective software components from code complexity measures*. In: *PRDC 2007*: 13th IEEE International Symposium on Pacific Rim Dependable Computing, pp. 93–96. Melbourne, Qld., Australia.

Zuse, H. (1991). *Software complexity: measures and methods*. Hawthorne, NJ, USA: Walter de Gruyter & Co.

Станков, Е. (2013). *Оценување на перформанси на програмски кодови преку методи на податочно рударење*. Магистерски труд на Факултетот за информатички науки и компјутерско инженерство при Универзитетот „Св. Кирил и Методиј", Скопје.

Додаток А. Дополнителни функционалности на CodeCPP

А.1. Автоматско почетно конфигурирање на одредени локации од интерес во шаблонски код

Повторувањето на една иста работа од страна на човек може да доведе до правење ненамерни грешки. Во рамките на употребата на Moodle додатокот CodeCPP за креирање на прашања што содржат програмски код од страна на наставниците, репетитивна работа којашто би се изведувала при секое внесување на шаблон за генерирање би била конфигурацијата на доменот на вредности за локациите од интерес детектирани во кодот на шаблонот. Оваа активност на прв поглед може да изгледа интересно за наставникот. Сепак, дури и ако треба да се формира тест со само 10 различни прашања што содржат програмски код (заради проверка на знаење од различни теми: аритметички оператори, логички оператори, циклуси, итн.), при што во шаблонскиот код за секое прашање се присутни само по 3 локации од интерес, ова ќе значи дека наставникот ќе треба да изврши 30 конфигурации. Во вакви околности, работата станува напорна и досадна, со што уште повеќе се зголемува веројатноста да дојде до пад на концентрацијата на наставникот и појава на несакани грешки.

Од овие причини, со детектирањето на локациите од интерес за кои може да се врши конфигурирање на доменот на дозволени вредности во даден шаблонски код, CodeCPP му предлага и подразбирливи (почетни) вредности на наставникот при пополнување на формата за конфигурација на шаблонот. Во оваа верзија на системот се опфатени неколку сценарија за кои се изведува автоматско почетно пополнување на формата во Moodle при креирањето на CodeCPP прашање. Секако, ова не го спречува наставникот рачно да го промени доменот на вредности или пак целосно да го исклучи мутирањето за некои (или и сите) локации од интерес за кои системот веќе поставил почетна конфигурација. Целта на автоматската почетна конфигурација што ја нуди CodeCPP е едноставно да се олесни работата на наставникот во процесот на креирање нови прашања.

А.1.1. Автоматско конфигурирање на низа

Да претпоставиме дека наставникот внел шаблонски код каков што е претставен на Слика А.1. При парсирање на апстрактното синтаксно дрво за кодот, се детектира декларација на низа од целобројен податочен тип, со зададена иницијализација на нејзините елементи со одредени вредности (целобројни литерали). Без разлика на димензионалноста на низата што е декларирана, се изминуваат сите вредности (елементи на низата) од наредбата

за декларација, при што се зачувуваат сите различни (уникатни) вредности. Дополнително, од сите различни вредности што се пронајдени, се извлекуваат и интервали од последователни цели броеви (доколку постојат такви!). Потоа, при приказот на формата за конфигурирање на правилата за мутација на локациите од интерес во шаблонскиот код, зачуваните вредности се поставуваат како подразбирливи вредности што го сочинуваат опсегот од којшто ќе се пополнува секој од елементите на низата при генерирањето на кодни варијации за тој шаблонски код. На Слика А.2 е прикажан почетниот изглед на дел од формата за конфигурирање на локациите од интерес за шаблонскиот код од Слика А.1, со автоматски поставени опсези од дозволени вредности за првите два елементи од низата (истите опсези се појавуваат и кај сите останати елементи).

```
//   Initial array code
#include <stdio.h>

int main()
{
    int a[2][3] = {{1, 2, 3}, {-1, -2, 5}};

    for (int i=0; i<2; ++i) {
        for (int j=0; j<3; ++j) {
            printf("%d", a[i][j])
        }
    }
    return 0;
}
```

Слика А.1: Пример за шаблонски код којшто содржи декларација на низа

Во случај ако станува збор за декларација (со иницијализација) на низа чии елементи се броеви со подвижна точка (`float`), опсегот на дозволени вредности којшто ќе го предложи CodeCPP за секој елемент од низата е од обликот "`min` : `max`", каде `min` и `max` се минималната и максималната вредност што се појавуваат во наредбата за декларација, соодветно. На пример, ако низата во шаблонскиот код од Слика А.1 беше декларирана како низа од податочниот тип `float` (наместо `int`), предложениот опсег за секој елемент (Слика А.2) би бил "–2.0 : 5.0".

А.1.2. Автоматско конфигурирање на "switch/case" локации

Да претпоставиме дека наставникот внел шаблонски код каков што е прикажан на Слика А.3. За да се предложи автоматско конфигурирање во ваква ситуација, пред да биде испарсирано апстрактното синтаксно дрво за кодот заради детекција на локациите од интерес, истото се изминува со цел да се зачуваат вредностите од сите `case` лабели, како и за да се поврзат зачуваните вредности со соодветната променлива чија вредност се евалуира од страна на

наредбата `switch` (во конкретниот пример – променливата `i`). По парсирањето на апстрактното синтаксно дрво се прикажува формата за конфигурирање на правилата за мутација на локациите од интерес во шаблонскиот код, при што зачуваните вредности се поставуваат како подразбирливи вредности што го сочинуваат опсегот од којшто ќе се пополнува локацијата од интерес што одговара на вредноста (константата) што се доделува на променливата. Слично како и кај низите, и во овој случај исто така се извлекуваат интервали од последователни цели броеви (доколку постојат такви!), од сите вредности што се пронајдени како `case` лабели во наредбата `switch`. Конечно, за да се опфати и `default` лабелата (во случаите кога таа е присутна), во предложените вредности се додава и дополнителна вредност којашто е еднаква на `max + 1`, каде `max` е максималната од сите вредности што се појавуваат во `case` лабелите.

Слика А.2: Почетен изглед на делот од формата за конфигурирање на локациите од интерес за шаблонскиот код од Слика А.1, што се однесува на првите два елементи од декларираната низа

На Слика А.4 е прикажан почетниот изглед на дел од формата за конфигурирање на локациите од интерес за шаблонскиот код од Слика А.3, со автоматски поставен опсег од дозволени вредности за променливата `i` (чија вредност се евалуира од страна на наредбата `switch` во кодот).

А.1.3. Автоматско конфигурирање на “string” локации

Да претпоставиме дека наставникот внел шаблонски код каков што е прикажан на Слика А.5.

```
//   Initial switch code
#include <stdio.h>

    int main() {
          int i = -2;
         switch(i) {
           case -2:
               printf("123");
          case -1:
               printf("1");
               break;
          case 9:
               printf("6");
               break;
          default:
               printf("1");
      }
      return 0;
}
```

Слика А.3: Пример за шаблонски код којшто содржи наредба `switch`

```
int main() {

int i = -2;

   switch(i) {
```

Edit int range

int Range -2:-1,9:10

Слика А.4: Почетен изглед на делот од формата за конфигурирање на локациите од интерес за шаблонскиот код од Слика А.3, што се однесува на конфигурирањето на вредноста на променливата `i`

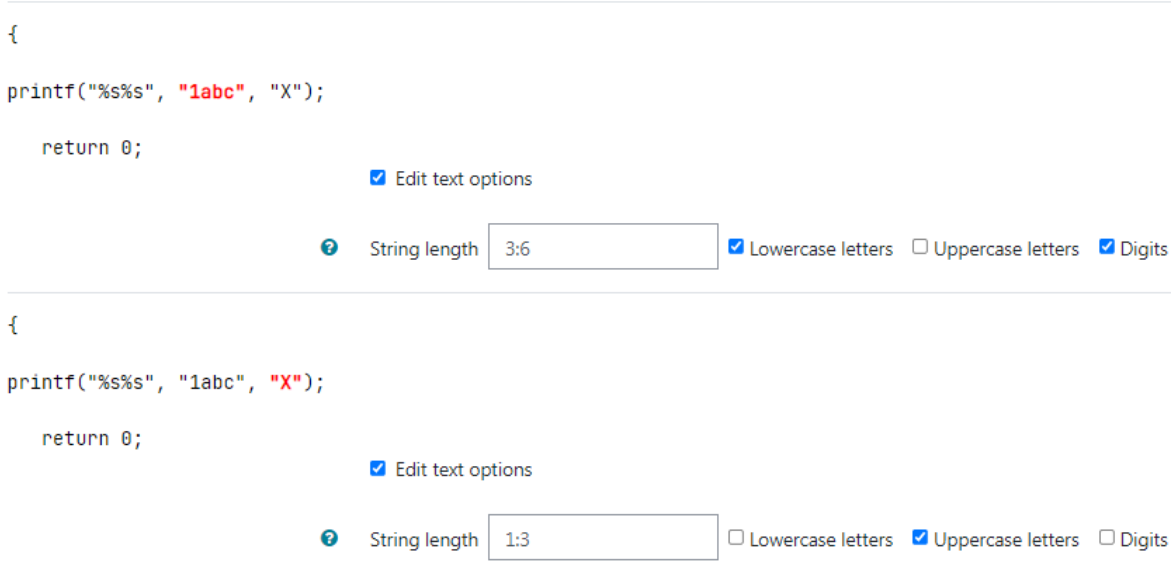
```
//   Initial string code
#include <stdio.h>

int main()
{
     printf("%s%s", "1abc", "X");
     return 0;
}
```

Слика А.5: Пример за шаблонски код којшто содржи текстуални низи (стрингови)

Во овој случај, за секој знак од текстуалните низи (стрингови) што се присутни во кодот (со исклучок на спецификаторите на формат што се користат

кај функциите `scanf()` и `printf()`, за кои не е дозволено менување!), CodeCPP ќе одреди дали станува збор за мала буква, голема буква или пак цифра. Врз основа на ова, при приказот на формата за конфигурирање на правилата за мутација на локациите од интерес во шаблонскиот код (Слика А.6), ќе бидат селектирани соодветните полиња за избор (за мали букви, големи букви и/или цифри, соодветно), со што ќе биде дефинирано (предложено) почетно множество од дозволени знаци за пополнување на соодветната локација од интерес што содржи текстуална низа. Од друга страна, за конфигурацијата на должината на стринговите, CodeCPP предлага опсег `max{1, L - 1} : L + 2`, каде `L` е должината на стрингот што е присутен во истата локација од интерес во шаблонскиот код.

Слика А.6 ја прикажува почетната конфигурација на локациите од интерес што ја предлага CodeCPP за шаблонскиот код од Слика А.5. Може да се забележи дека за локацијата од интерес на која се наоѓа стрингот `"1abc"` во шаблонскиот код, селектирани се полињата за цифри и мали букви, а за должината на стринговите што ќе се добиваат во генерирачкиот процес е предложен целобројниот опсег 3:6 (бидејќи должината на `"1abc"` е 4). Од друга страна, за локацијата од интерес на која се наоѓа стрингот `"X"` во шаблонскиот код, селектирано е само полето за големи букви, додека за должината на стринговите е предложен целобројниот опсег 1:3 (должината на `"X"` е 1).



Слика А.6: Почетен изглед на формата за конфигурирање на локациите од интерес за шаблонскиот код од Слика А.5

А.2. Заштита од мамење

Основната цел на тестовите на знаење кои најчесто се задаваат е проверка и оценување на знаењето на испитаниците. Но, скоро секогаш се јавува и некој којшто ќе сака да препише или да мами (англ. *cheat*). Една од

причините за развивањето на системот за паметно автоматско генерирање на прашања што содржат програмски код, а воедно и на Moodle додатокот CodeCPP, е да се намали веројатноста дека студентите коишто полагаат испит (тест) од воведен курс за програмирање ќе препишуваат еден од друг. Ова е остварено преку генерирањето на поголем број верзии (кодни варијации) од секое прашање (при што секоја верзија е добиена од ист почетен шаблон, со извршување на соодветни модификации над локациите од интерес во кодот на шаблонот, внимавајќи на сложеноста на кодовите што се генерираат) што го овозможува системот. Но, ништо не го спречува студентот додека го решава тестот користејќи веб прелистувач на компјутерот на којшто работи, да го ископира кодот од кое било прашање, да го искомпајлира и изврши (на пример, во некоја развојна околина којашто му е достапна), а со тоа да дојде до точниот одговор (односно излезот од кодот) иако можеби не го поседува потребното знаење. За да се спречи ова, додадена е дополнителна функционалност на CodeCPP, со којашто од програмскиот код на секое прашање се генерира слика која му се прикажува на студентот за време на тестирање. Оваа функционалност значително го отежнува мамењето на ваков начин – кодот сега не може да се копира, па студентот единствено би можел само рачно да го препише во некоја развојна околина (што секако побарува значително повеќе време од едноставно копирање на кодот). Сепак, типкањето на тастатура е активност којашто лесно се забележува (се прават специфични движења со рацете; се генерира и звук од типките), па би се очекувало дека обидот за мамење со препишување на код многу едноставно и брзо би бил детектиран од наставниот кадар задолжен за надзор за време на полагањето.

А.2.1. Генерирање на слика од програмски код во прашање од тип CodeCPP

При имплементирањето на оваа функционалност на системот имаше неколку предизвици. Првиот предизвик беше да се најде начин како да се изгенерира квалитетна слика за даден програмски код. Програмскиот јазик PHP нуди можност за генерирање слика, како и за вметнување на текст во сликата (со малку математички пресметки). Исто така, постои и опција за користење на сопствен фонт за текстот. Имајќи предвид дека за прикажување на изворен код на програми најдобро (од естетска гледна точка, но и заради прегледност и читливост) е да се користи непропорционален фонт (фонт со фиксна широчина на буквите и знаците, англ. *monospaced font*) (Rosendorf, 2009), беше избран JetBrainsMono-Regular.ttf од фамилијата JetBrains Mono (*JetBrains Mono*, 2020), којшто е бесплатен фонт со отворен изворен код.

Изгледот на една изгенерирана слика од програмски код во рамки на системот е прикажан на Слика А.7. Треба да се напомене дека постои и можност одредени делови од кодот (како на пример, клучните зборови од програмскиот

јазик во којшто е напишан кодот) да бидат прикажани со различна боја, со што уште повеќе би се подобрила читливоста, но ова останува да се имплементира во некоја следна верзија на CodeCPP.

```
//prashanje 1
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int a=7;
    int b=-4;
    printf("%d\n", (a-b)%5);
    return 0;
}
```

Слика А.7: Изглед на изгенерирана слика од програмски код во рамки на прашање од тип CodeCPP во Moodle

А.2.2. Проблемот на латентност предизвикана од генерирањето на слики

Во споредба со сите други операции кои се имплементирани во рамките на Moodle додатокот CodeCPP, генерирањето на слика побарува далеку поголемо процесорско време. Ова беше вториот (а воедно и потежок) предизвик при имплементирањето на оваа функционалност. За разумен код, каков што е оној на Слика А.7, потребното време за генерирање на една слика (на современ лаптоп со просечна конфигурација) е во рангот 40 – 60ms (за слики со дефинирана позадина), односно во рангот 100 – 120ms (за слики со транспарентна позадина). Кај CodeCPP се изведува генерирање на слики со транспарентна позадина, па како просечна вредност за потребното време на генерирање на една слика може да се земе 110ms. Во пракса, каде Moodle се извршува на сервер(и), можеби и ова време би делувало дека е занемарливо. Но, во продолжение е опишано сценарио кое појасно го претставува проблемот што се појавува.

Да претпоставиме дека еден просечен тест (квиз во Moodle) би содржел 10 прашања од тип CodeCPP, при што за секое прашање би се генерирале во просек по 30 различни верзии, односно по 30 различни кодни варијации од шаблонскиот код за прашањето. Ова би значело дека ќе треба да се изгенерираат вкупно 300 различни слики од програмски кодови за тестот. Ако се користи едноставен пристап при којшто сите слики ќе се генерираат при стартувањето на тестот, потребното време за генерирање би било 300 × 110ms = 33 000ms ≈ 33s. Дури и да се користи двојно побрз компјутер, ќе стане збор за време од околу 15s, што би значело дека при старт на обидот (што најчесто го

прават сите испитаници одеднаш) би се предизвикало многу долго чекање кај сите испитаници. Корисниците на Интернет во денешно време се навикнати на „нормално“ доцнење кое вообичаено е во рангот од 2 – 3s (поради пречки во Интернетот, бавен компјутер/телефон, итн.), така што секое време на чекање над овие вредности би предизвикало фрустрација кај испитаниците. Оттука се јавува потреба да се намали ова доцнење предизвикано од големото време на генерирање на сликите, односно истото да се сведе на некое „разумно“ време.

Како решение на овој проблем беа разгледувани повеќе опции. Првата од нив е при зачувување на генерираните кодни варијации на даден шаблонски код (уште во фазата на креирање на прашања за тестот) да се зачувува и генерираната слика за секоја од нив. Овој пристап не е практичен од две причини. Како прво, бидејќи се работи за слика (без разлика на нејзината величина) која се зачувува во база на податоци, ова значи дека базата со текот на времето ќе стане многу голема (т.е. ќе зафаќа голем мемориски простор). Како второ, еднаш генерираните слики остануваат такви какви што се. Во случај ако корисниците (наставниците) одлучат да направат некаква промена во изгледот на кодовите (на пример, промена на големината на текстот, бојата или просторот околу текстот), преку конфигурирање на соодветните опции за кои веќе беше одлучено дека треба да им бидат понудени од страна на CodeCPP (поглавје А.3), ќе биде неопходно сите слики одново да се изгенерираат и зачуваат. Ова би било тешко одржливо, па затоа првиот разгледан пристап за решавање на проблемот беше отфрлен.

Вториот пристап, којшто беше разгледан, е генерирањето на сликите за сите кодни варијации на шаблонскиот код за кое било прашање да се изведува само тогаш кога се прикажува тоа прашање за време на тестирање, наместо при стартувањето на тестот. Со примена на овој пристап кај погоре разгледуваното сценарио на просечен тест со 10 прашања од типот CodeCPP, уште на почетокот на тестирањето проблемот се редуцира на генерирање на 30 слики само за првото прашање, а останатите би се генерирале како што би се движеле испитаниците низ следните прашања. Иницијалниот „напад на слики“ кои треба да се изгенерираат на ваков начин е намален само на едно прашање, наместо да се генерираат сликите за сите прашања од тестот одеднаш. Според тоа, времето на чекање (потребното време за генерирање на слики) се сведува на 30 × 110ms = 3 300ms ≈ 3s. Потоа, многу малку веројатно е дека сите испитаници во ист момент ќе го отворат второто, па третото прашање од тестот, итн., па со тоа проблемот на долго чекање за време на тестирање (барем навидум) е решен.

Сепак, овде потенцијално може да се појави и уште еден проблем. Ништо не ги спречува испитаниците постојано да ја освежуваат страницата на којашто е прикажано едно прашање од тестот, при што серверот ќе треба одново и одново да генерира (нова) слика од кодот, па според тоа непотребно ќе се

одзема драгоцено процесорско време на серверот. Овој проблем може да се реши со користење на кеш (англ. *cache*). За да не се развива цел систем кој што би правел кеширање, беше одлучено да се искористи постојниот т.н. универзален кеш на Moodle (англ. *Moodle Universal Cache* – MUC) (*MUC*, 2012). Времето коешто е потребно за извлекување на еден податок од MUC е приближно 1ms, што е значително подобрување во однос на потребното време за генерирање на една слика. Со искористувањето на MUC, во случај ако испитаникот направи освежување на страница со прашање, наместо да се генерира нова слика од кодот, ќе биде извлечена старата (претходно генерирана) слика од MUC, со што се прави значителна заштеда на процесорско време. Со ова е решен и проблемот на „постојано освежување“ на прашања и латентноста предизвикана од истото.

Бидејќи MUC се користи низ многу места на платформата Moodle, постојат голем број на опции во врска со него кои може да се нагодат за кој било додаток на оваа платформа. Овде треба да се потенцира дека за додатокот CodeCPP беше дефинирано дека во даден момент ќе можат да се памтат најмногу 900 слики од програмски кодови во MUC. Генерирана слика за еден разумен код има просечна големина од 300 Kb, па за 900 слики би биле потребни 900 × 300 Kb = 270 000 Kb ≈ 270 Mb, што е разумна големина на податоци кои би се чувале во кешот (во најлош случај) во секој момент.

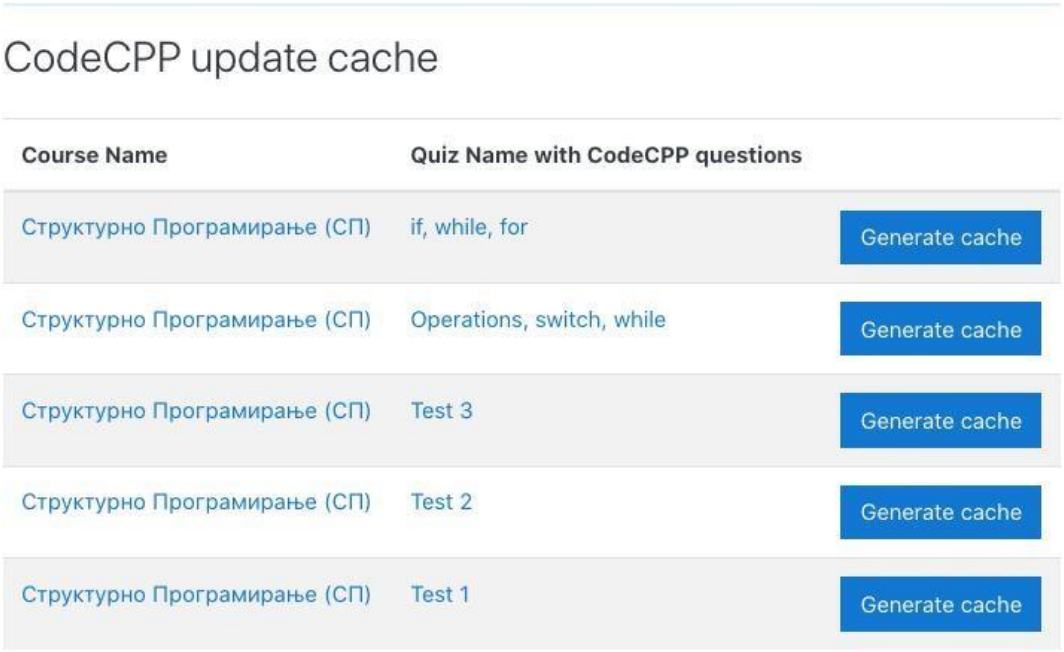
Конечно, поаѓајќи од идејата за кеширање на сликите заради решавање на проблемот на „постојано освежување“ на прашањата за време на тестирање, беше донесена одлука и за финалниот пристап кој беше имплементиран како решение на главниот проблем на долго време на чекање предизвикано од процесот на генерирање на слики од програмските кодови за прашањата. Во имплементираниот пристап, сите слики (од сите кодни варијации, за сите прашања) коишто се потребни за одреден тест можат да се искешираат (во MUC) уште пред да биде стартуван тој. Оваа операција е овозможена само за администраторот, преку соодветен поглед во којшто тој може да одбере за кој од постојните тестови што содржат прашања од типот CodeCPP би сакал да изврши кеширање. Овој поглед е прикажан на Слика А.8. Секако, при кликнување на копчето “Generate cache” заради кеширање на некој тест (Слика А.8), администраторот треба да очекува временски подолга операција.

Во случај ако не се направи кеширање на тестот пред почетокот на тестирањето, генерирањето на сликите се одвива „прашање-по-прашање“ во текот на тестирањето, како што беше објаснето погоре.

А.3. Дополнителни администраторски опции

Покрај опциите објаснети во поглавјето 4.3.2, за администраторот дополнително беа воведени и уште неколку опции за конфигурирање на

CodeCPP, а кои се однесуваат на генерирањето на слики од програмски кодови. На Слика А.9 е прикажан делот којшто беше додаден во администраторската страница за конфигурација, со цел да се овозможи конфигурирање на некои параметри за оваа функционалност.



Слика А.8: Поглед на сите Moodle квизови што содржат прашања од тип CodeCPP, со копче за кеширање на секој од нив, достапен само за администраторот

Слика А.9: Поглед на делот од администраторската страница за конфигурација на CodeCPP што беше додаден со цел да се овозможи конфигурирање на параметри за генерирањето на слики од кодови

Како што може да се види од Слика А.9, администраторот има можност да го вклучи/исклучи генерирањето на слика од секој програмски код, преку

(не)селектирање на полето за избор именувано “Generate image from text”. Во случај ако е вклучена оваа функционалност, администраторот исто така може да ја прилагоди големината на фонтот, просторот околу текстот, како и бојата на фонтот со која ќе биде прикажан кодот во секоја генерирана слика.

Додаток Б. Изглед на шаблонските прашања за еден од тестовите од Експеримент 3

Сите прашања имаат единствен текст: Кој е излезот од следниот код? Кодовите на прашањата од квизот даден на тема „Низи“ се дадени во продолжение.

Прашање 1.

```
#include <iostream>
using namespace std;
int main()
{
    int niza[6]={9, 2, 2, 1, 8, 8};
    for (int i=0; i<6; i++)
    {
        cout << niza[i];
    }
    return 0;
}
```

Одговор: 922188

Прашање 2.

```
#include <iostream>
using namespace std;
int main()
{
    int niza[8]={4, 9, 7, 5};
    for (int i=0; i<6; i++)
    {
        cout << niza[i];
    }
    return 0;
}
```

Одговор: 497500

Прашање 3.

```
#include <iostream>
using namespace std;
int main()
{
    int niza[10]={9, 1, 8, 3, 7, 4, 0, 0, 9, 7};
    for (int i=1; i<10; i=i+3)
    {
        cout << niza[i];
    }
    return 0;
}
```

Одговор: 170

Прашање 4.

```
#include <iostream>
using namespace std;
int main()
{
    int niza[10]={5, 0, 3, 3, 7, 8, 8, 6, 8, 7};
    for (int i=5; i>0; i--)
    {
        cout << niza[i];
    }
    return 0;
}
```

Одговор: 87330

Прашање 5.

```
#include <iostream>
using namespace std;
int main()
{
    int niza[10]={3, 1, 9, 5, 8, 9, 2, 1, 2, 9};
    for (int i=0; i<10; i++)
    {
        if (niza[i]%2==0)
            cout << niza[i];
    }
    return 0;
}
```

Одговор: 822

Прашање 6.

```
#include <iostream>
using namespace std;
int main()
{
    int niza[10]={4, 2, 3, 8, 0, 5, 6, 4, 1, 9};
    for (int i=0; i<10; i++)
    {
        if (niza[i]>3)
            cout << niza[i];
    }
    return 0;
}
```

Одговор: 485649

Прашање 7.

```
#include <iostream>
using namespace std;
int main()
{
    int niza[10]={9, 0, 0, 9, 3, 7, 9, 1, 5, 1};
    int k=4;
    for (int i=0; i<10; i++)
    {
        if (niza[i] < k)
            cout << niza[i];
    }
    return 0;
}
```

Одговор: 00311

Прашање 8.

```
#include <iostream>
using namespace std;
int main()
{
    int niza[10]={6, 9, 6, 5, 5, 5, 7};
    int k=3;
    for (int i=1; i<6; i++)
    {
        if (niza[i] <= k)
            cout << niza[i];
          else
            cout << 4;
    }
    return 0;
}
```

Одговор: 44444

Прашање 9.

```
#include <iostream>
using namespace std;
int main()
{
    int niza[10]={9, 2, 3, 5, 5, 5, 0};
    int k=-10;
    for (int i=0; i<7; i++)
    {
        if (niza[i] > k)
            k=niza[i];
    }

    cout<<k;
    return 0;
}
```

Одговор: 9

Прашање 10.

```
#include <iostream>

using namespace std;

int main()

{

    int niza[10]={5, 3, 6, 2, 4, 6, 2};

    int k=70;

    for (int i=0; i<7; i++)

    {

        if (niza[i]<k)

            k=niza[i];

    }


    for (int i=0; i<7; i++)

    {

        cout<<niza[i]+k;

    }

    return 0;

}
```

Одговор: 7584684

Додаток В. Изглед на прашањата од анкетата од Експеримент 4

П1. Колкаво е вашето искуство во предавање курсеви за програмирање? (внесете број на години)

Одговор: ___ години

П2. Дали сте го користеле Moodle како корисници – слушатели на курс?

а) Да

б) Не

П3. Дали сте го користеле Moodle како наставници на курс?

а) Да

б) Не

П4. Дали имате искуство со креирање на квиз во Moodle?

а) Да

б) Не

П5. Алатката CodeCPP ќе ми помогне во реализацијата на наставата.

а) Целосно се согласувам

б) Делумно се согласувам

в) Делумно не се согласувам

г) Воопшто не се согласувам

П6. Алатката CodeCPP може да се користи за испорака на повратни информации (англ. *feedback*) до учениците, околу нивниот прогрес во изучувањето на програмирањето.

а) Целосно се согласувам

б) Делумно се согласувам

в) Делумно не се согласувам

г) Воопшто не се согласувам

П7. Алатката CodeCPP може да се користи за оценување на знаењето на учениците од програмирање.

а) Целосно се согласувам

б) Делумно се согласувам

в) Делумно не се согласувам

г) Воопшто не се согласувам

П8. Колку често би ја користеле алатката CodeCPP во наставата?

а) Не би ја користел(а)

б) Секој наставен час

в) Секој втор наставен час

г) Секој трет наставен час

д) Секој петти наставен час

ѓ) Еднаш месечно

е) Еднаш во тромесечјето

ж) Еднаш во полугодието

з) Еднаш во учебната година

П9. Колку време сметате дека ќе Ви биде потребно да подготвите еден краток квиз со 5 прашања, користејќи ја алатката CodeCPP?

а) до 5 минути

б) помеѓу 5 и 10 минути

в) помеѓу 10 и 20 минути

г) помеѓу 20 и 30 минути

д) повеќе од 30 минути

П10. Алатката CodeCPP би помогнала да се намали препишувањето како појава за време на испитување на учениците.

а) Целосно се согласувам

б) Делумно се согласувам

в) Делумно не се согласувам

г) Воопшто не се согласувам

П11. Алатката CodeCPP е соодветна за користење во услови на онлајн настава.

а) Целосно се согласувам

б) Делумно се согласувам

в) Делумно не се согласувам

г) Воопшто не се согласувам

П12. Опишете накратко некои придобивки за кои сметате дека ќе ги овозможи алатката CodeCPP. Дополнете и ако имате некои коментари во однос на претходните прашања.

Одговор: