Univerzitet u Beogradu
Matematički fakultet

Kocić Ognjen

Ocenjivač zadataka iz programiranja - implementacija složenog softvera u modernom C++-u

MASTER RAD

Beograd, Oktobar 2015

Univerzitet u Beogradu – Matematički fakultet
MASTER RAD

Autor:	Ognjen Kocić
Naslov:	Ocenjivač zadataka iz programiranja - implementacija složenog softvera u modernom C++-u
Mentor:	dr Saša Malkov, Matematički fakultet
Članovi komisije:	dr Milena Vujošević-Janičić, Matematički fakultet, dr Aleksandar Kartelj, Matematički fakultet
Datum:	8.10.2015

Rezime

Velika važnost problema automatizacije ocenjivanja programerskih zadataka dolazi sa tendencijama da se proces nastave što više automatizuje. Dodatno, ovaj problem je zanimljiv i sa informatičkog aspekta jer je za njegovo rešavanje potrebno povezivanje više različitih oblasti računarstva, kao što su operativni sistemi, mrežno programiranje, paralelno programiranje, algoritmi i druge.

U ovom radu biće prikazano jedno rešenje ovog problema i njegova implementacija u programskom jeziku C++ na operativnom sistemu Linux. Osim predstavljanja složene distribuirane višeprocesne arhitekture rešenja, dodatno će biti objašnjene pogodnosti i idiome modernog C++-a koje olakšavaju svakodnevni razvoj softvera.

Treba napomenuti da se ocenjivač neće baviti interpretacijom uspešnosti izvršavanih programa u vidu određivanja broja poena koje je program osvojio. Fokus će isključivo biti na pružanju informacija vezanih za test primere tj. da li je rezultat rada programa ispravan, da li je došlo do greške u izvršavanju, greške pri kompilaciji i drugo.

Ključne reči: ocenjivanje programa, automatizacija nastave, programiranje, C++

Sadržaj

1 Uvod

1.1 Kratak uvod i motivacija

Programiranje postaje jedno od najtraženijih zanimanja u svetu i ta popularnost povlači sve veći broj studenata koji žele da se bave ovom oblašću. To se vidi i iz upisnih statistika. Broj bodova potrebnih za upis informatičkog smera Matematičkog fakulteta u Beogradu u prvom upisnom roku od 2008. do 2015. godine dat je u Tabeli 1.

Godina upisa	Broj bodova potrebnih za budžet
2008	50.16
2009	53.16
2010	55.40
2011	52.62
2012	57.92
2013	57.18
2014	73.66
2015	77.8

Tabela 1: Broj bodova potrebnih za upis informatičkog smera od 2008-2015. godine.

Kako je računarstvo široka oblast, gradivo koje je potrebno savladati da bi se uspešno položio predmet je često obimno. To predstavlja problem, pogotovo studentima nižih godina koji se prvi put sreću sa novim terminima, idejama i konceptima. Studenti koji kvalitetno nauče osnove programiranja na ovim uvodnim kursevima, lako svoje znanje u nastavku školovanja primenjuju i proširuju. Na vežbama se u okviru jedne sesije (jednog viđanja) pređu i do tri teme. Kvalitet nastave bi se značajno poboljšao kada bi studenti mogli da dobiju domaće zadatke koji pokrivaju sve pređeno u okviru jednih vežbi. Međutim, zbog velikog broja studenata nastavno osoblje nije u mogućnosti da ove domaće zadatke sprovede kao formalnu obavezu. Čak i oni studenti koji urade domaći, nemaju pouzdan način da provere da je njihov kod zaista dobar. Da bismo se uverili da pred nama leži stvaran problem dovoljno je pogledati broj studenata koji su slušali kurseve “Programiranje 1” i “Programiranje 2” u školskoj 2014/15 godini. Navedeni kursevi su uvodni kursevi iz programiranja u programskom jeziku C, koji se pohađaju na prvoj

godini osnovnih studija informatičkog smera Matematičkog fakulteta u Beogradu. Relevantni podaci se nalaze u Tabeli 2.

Predmet i modul	Ukupno studenata	Novih studenata	Starih studenata
Programiranje 1 Inf.	223	136	**8**7
Programiranje 2 Inf.	276	136	140
Programiranje 1 Mat.	393	250	143
Programiranje 2 Mat.	483	250	233

Tabela 2: Broj studenata na uvodnim kursevima iz programiranja u školskoj 2014/2015.

U poslednjoj koloni Tabele 2 može se primetiti da imamo veliki broj studenata koji kurs nisu položili u prvoj godini slušanja. U slučaju predmeta "Programiranje 2" za modul Informatika, rezultati su toliko loši da je većina studenata upisanih na kurs baš iz te kategorije. Situacija nije mnogo bolja ni za modul Matematika.

Upotrebom automatskog ocenjivača programa proces nastave se značajno olakšava. Izgradnjom odgovarajućeg interfejsa za ocenjivač, jedan student može veći broj puta da pošalje svoje rešenje na evaluaciju i da u tom procesu sam pronađe i popravi svoje greške. Za to vreme nastavno osoblje je u potpunosti rasterećeno i može nesmetano da odgovara na potencijalna pitanja vezana za postavljeni domaći zadatak. Kao dodatnu pogodnost i uz malo truda, nastavnici mogu da prikupe podatke vezane za ove domaće zadatke i da vide koje oblasti studentima idu bolje ili lošije i da se na njih više fokusiraju. Korišćenjem ovih pogodnosti može da se postigne radno okruženje u kome i studenti i nastavnici mogu više da napreduju jedni u savladavanju, a drugi u prenošenju znanja.

S obzirom na to da su prednosti ovakvog softvera prepoznate i ranije, dokument kojim je ocenjivač predstavljen nije prvi rad na ovu temu. Jedna od prvih upotreba automatskog ocenjivanja programa datira još iz 1965. godine i predstavlja korišćenje automatskog ocenjivača za programski jezik Algol [39]. Danas se postojeći softver može podeliti u dve kategorije. Prvu kategoriju čine sistemi pravljeni za takmičenja iz oblasti informatike, kao što su na primer *SPOJ* [40] i *TopCoder* [41]. Ovi sistemi svoj fokus usmeravaju na proveru da li je postavljeno rešenje zadovoljilo memorijske i vremenske kriterijume (pored tačnosti samog rezultata). Sa druge strane, sistemi koji pripadaju drugoj kategoriji, kao što su *WebCAT* [42] i *Marmoset* [43], pažnju više posvećuju pružanju korisnih informacija koje daju dobar opis zašto studentsko rešenje ne radi kako treba.

2 Problem

2.1 Pristup problemu

Na prvi pogled, automatsko ocenjivanje zadataka ne izgleda kao posebno težak problem. U najjednostavnijem slučaju moguće je napisati skriptu kojom se predefinisana datoteka prosleđuje na standardni ulaz programa koji treba oceniti. Dobijeni izlaz iz ovog programa bi se preusmeravao i upoređivao sa očekivanim.

Primetimo da ovakva skripta ne vrši nikakve bezbednosne provere potencijalno zlonamernog studentskog programa. Takođe, ne postoji ni ograničavanje vremena i memorije koji su dostupni tom programu, pri čemu to nije kraj liste nedostataka. Naredna sporna stavka je poređenje rezultata. Uzmimo na primer obradu slika i poređenje 2 slike. Nije ih moguće porediti kao niske. Postoji i još jednostavniji primer - poređenje 2 realna broja zapisana u računaru. Često će se desiti da dva različita koda, koja uspešno rešavaju jedan problem, daju rezultat koji je se razlikuje na petoj ili šestoj decimali. Umesto leksičke jednakosti, zgodnije bi bilo porediti ove brojeve sa određenom tačnošću.

Konačno, jednostavna skripta ne ispunjava ni osnovne funkcionalnosti koje treba da pruži da bi mogla da pregleda jednostavan zadatak iz predmeta “Programiranje 1” prve godine Matematičkog fakulteta. Tipičan zadatak, preuzet sa Veb strane jedne asistentkinje, glasi:

Sastaviti program koji sa standarnog ulaza učitava imena dve datoteke (ulazna i izlazna datoteka) i iz ulazne datoteke kopira u izlaznu svaki drugi karakter polazeći od prvog pročitanog karaktera. U slučaju greške u otvaranju i zatvaranju datoteka, prijaviti odgovarajuci komentar na stdout ili stderr.

Iz zadatka se vidi da se izlaz od interesa nalazi u datoteci čije se ime dinamički zadaje, a ne na standardnom izlazu, pa tako skripta ne bi mogla da poredi dobijeni izlaz sa očekivanim.

2.2 Zahtevi

2.2.1 Osnovni zahtevi

Kako postoji mnogo potrebnih kriterijuma koje osnovna skripta ne ispunjava, potrebno je sagledati funkcionalna očekivanja postavljena pred ocenjivač:

1. *Mora da omogući kompilaciju izvornih kodova* - time će ocenjivač proizvesti izvršni kod koji će biti procesiran.

2. *Mora da omogući prosleđivanje ulaza iz više različitih kanala* – razumne ulazne kanale predstavljaju standardni ulaz, argumenti komandne linije i sistem datoteka. Dodatno, treba omogućiti prosleđivanje više datoteka kao ulaz, pri čemu te datoteke mogu da se nalaze u različitim direktorijumima.
3. *Mora da omogući prosleđivanje izlaza na više različitih kanala* - potrebno je omogućiti evaluciju izlaza programa i kada je ispisan na standardnom izlazu kada se nalazi u izlaznim datotekama.
4. *Mora da omogući detekciju prekoračenog zadatog vremenskog ograničenja* - za uspešno ocenjivanje programa u nekim oblastima računarstva potrebno je da se program izvršava za određeno vreme. Na primer, u zavisnosti od rasta vremena izvršavanja u odnosu na veličinu ulaza, moguće je proceniti vremensku složenost programa što je jako bitno za ocenjivanje rešenja algoritamskih problema.
5. *Mora da omogući detekciju prekoračenog zadatog memorijskog ograničenja* - memorijsko ograničenje je korisno iz sličnog razloga kao i vremensko ograničenje.
6. *Mora da omogući pouzdane i jasne informacije u poznatom formatu o rezultatima izvršavanja ocenjivanog programa* - ovo je jasan zahtev zato što predstavlja preduslov da studenti pronađu potencijalnu grešku u svom rešenju. Osim toga, i nastavnicima se pruža uvid u studensku grešku bez potrebe da je oni sami traže.

Navedeni zahtevi predstavljaju osnovu koju ocenjivač mora da implementira da bi ispravno radio za najjednostavnije slučajeve.

Međutim, pokazuje se da postoji još dosta uslova koje ocenjivač mora da ispoštuje kako bi se uspešno odgovorilo na problem. Radi ilustracije, posmatrajmo *zahtev 4*. Njega je lako implementirati jednostavnim merenjem vremena izvršavanja pokrenutog programa. To ipak nije dobar pristup, što se vidi ako pogledamo šta će se desiti sa studentskim programom koji ima beskonačnu petlju. Program se nikada neće završiti, dakle potrebno ga je nasilno prekinuti nakon isteka predviđenog vremena izvršavanja.

Zlonameran studentski kod je još jedna od realnih situacija. Mogućnosti za napad ima mnogo, počevši od jednostavnog brisanja datoteka koje pripadaju korisničkom nalogu pod kojim se ocenjivač pokreće, do onih napada za koje studenti mogu da tvrde da su samo greška. Na primer, otvaranje prevelikog broja datoteka ili pokretanje prevelikog broja procesa, takođe mogu da ugroze rad računara na kome se ocenjivač nalazi.

Na samom kraju, ocenjivač je najkorisniji ako je deo nekog šireg sistema, pa

treba razmotriti i faktore koji su bitni za njegovu nesmetanu integraciju u te sisteme.

2.2.2 Zahtevi za kontrolu izvršavanja

Da bi se izbegli prethodno navedeni problemi potrebno je uvesti još neke zahteve vezane za kontrolu izvršavanja ocenjivanog programa:

1. *Po isteku predviđenog vremena izvršavanja studentski program mora biti nasilno prekinut* - ovim se direktno izbegava problem beskonačne petlje. Indirektno, računar na kome ocenjivač radi se rasterećuje izbegavanjem daljeg izvršavanja programa, koje je sigurno nepotrebno jer program nije ispunio vremenski kriterijum.
2. *Po prekoračenju dozvoljene memorije, u bilo kom trenutku rada programa, ocenjivač mora nasilno prekinuti program* - dodatno se rasterećuje sistem na isti način kao kod prethodnog zahteva.
3. *Ocenjivač mora ograničiti broj datoteka koje studentski program može da otvori i u slučaju prekoračenja mora nasilno prekinuti njegovo izvršavanje* - otvaranje prevelikog broja datoteka može ostaviti sve programe, koji se izvršavaju na računaru gde je ocenjivač, bez mogućnosti da otvore novu datoteku.
4. *Ocenjivač mora ograničiti broj procesa koje studentski program može pokrenuti i u slučaju prekoračenja mora nasilno prekinuti njegovo izvršavanje* - ovim uslovom se računar na kome je ocenjivač brani od pokretanja previše procesa i usporavanja sistema.
5. *Ocenjivač mora ograničiti studentske programe da ne mogu da prepune disk* - do problema može doći kada studentski programi pišu proizvoljnu lokaciju na disku. Ukoliko ocenjivač ne obriše napisano kada se program završi, doći će do zagušenja diska.

2.2.3 Bezbednosni zahtevi

Navedeni zahtevi su potrebni da bi ocenjivač ispravno radio uz pretpostavku da studentski kodovi nisu zlonamerni, već da je njihovo ponašanje isključivo nenamerna greška u programiranju.

Ocenjivač nije softver koji je samom sebi dovoljan. On bi trebalo da se uklapa u šire sisteme i da čini njihovu važnu komponentu. Zbog toga se dolazi do naredne liste zahteva:

1. *Ocenjivač mora podržavati različite scenarije integracije u postojeće sisteme* - jedan od najvažnijih ovakvih sistema je i Veb aplikacija koja se danas često koristi kao grafički interfejs za ocenjivače. Međutim,

jednako jednostavno treba da bude i pregledanje kolokvijuma ili ispita, upotrebom ocenjivača bez dodatnih grafičkih interfejsa.

2. *Ocenjivač mora da bude konfigurabilan i proširiv* - omogućavanje dodavanja komponenti koje ocenjuju nove programske jezike je obavezno. Poređenje izlaza iz programa sa očekivanim izlazom takođe mora da bude deo ocenjivača koji je moguće nadograditi. Dodatno, mora da se pruži i mogućnost podešavanja ocenjivača u zavisnosti od softvera sa kojim je integrisan.
3. *Ocenjivač mora da omogući upotrebu sa različitih operativnih sistema* - sam postupak postavljanja studentskog koda na ocenjivanje, kao i prijem rezultata ocenjivanja, moraju da budu aktivnosti nezavisne od operativnog sistema.
4. *Ocenjivač mora imati interfejs prema različitim programskim jezicima* - da bi se ocenjivač lakše integrisao u neku novu ili postojeću programersku aplikaciju, zgodno je da postoji pristup ocenjivaču u programskom jeziku na kome je aplikacija pisana.

3 Implementacija

3.1 Izbor arhitekture

Na izbor arhitekture ocenjivača programa su najviše uticali zahtevi vezani za njegovu upotrebljivost. Od ostalih grupa zahteva bitni su i oni koji se odnose na kontrolu izvršavanja studentskog programa.

3.1.1 Strukturna organizacija ocenjivača

Kako ocenjivač mora da podrži integraciju u više različitih sistema, jedna od mogućih ideja je da se on imlementira u vidu dinamičke biblioteke, koju bi druga okruženja mogla da koriste. Nakon implementacije u izabranom jeziku, potrebno je pronaći glavne funkcionalnosti kojima će se pružiti direktan pristup iz spoljašnjih aplikacija. Za ovaj interfejs se zatim pravi omotač u programskom jeziku C. Bira se C, zato što većina programskih jezika ima direktan način komunikacije sa programskim jezikom C. Neki od načina kombinovanja C funkcija i popularnih programskih jezika su:

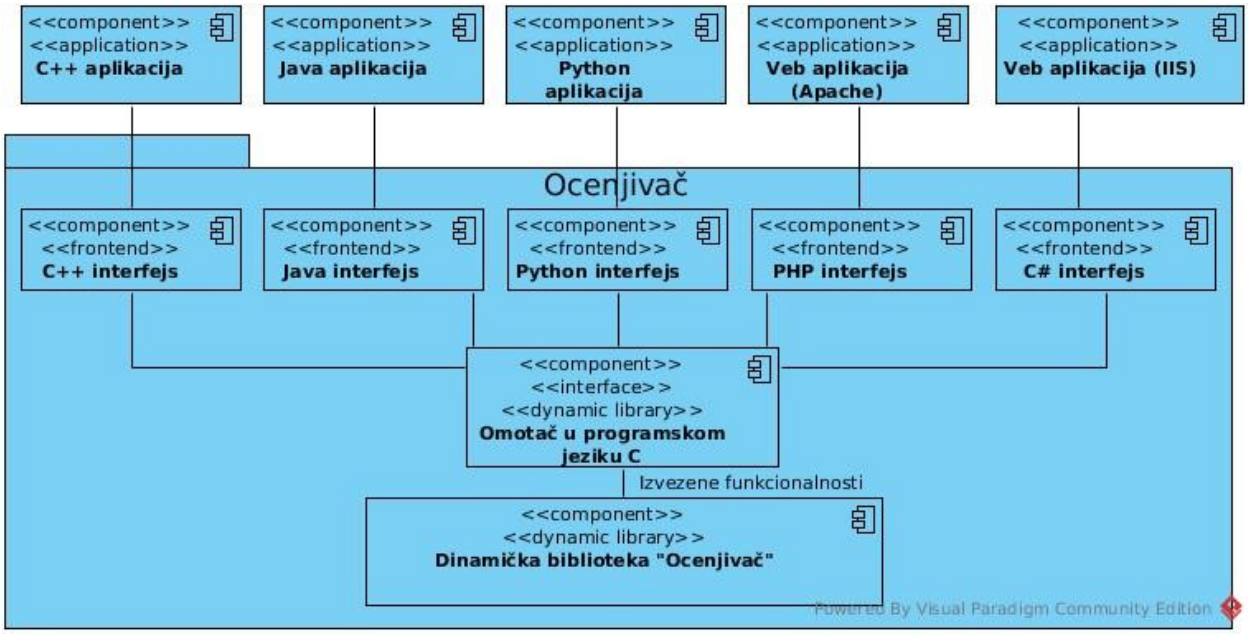
1. *C++* - direktno pozivanje funkcija.
2. *Java* - pozivanje funkcija preko *JNI* (eng. Java Native Interface) [1] koji je dizajniran u te svrhe.
3. *Python* - modul koji omogućava normalno pozivanje C funkcija se zove

ctypes [2].

4. *C#* - moguće je koristiti C funkcije pomoću direktive *DllImport* [3].
5. *Objective-C* - direktno pozivanje funkcija.
6. *PHP* - pravljenje PHP ekstenzija [4].
7. *Javascript* - ovaj jezik se najviše koristi iz Veb pregledača za uređenje vizuelne prezentacije Veb aplikacija. Zbog toga programski jezik C i nema neku očiglednu primenu. Međutim, postoje projekti kompanije Google, pod nazivom *NaCL* [5] i *PNaCL* [6], koji omogućavaju pozive C funkcija iz Javascripta.

Prednost ovakvog pristupa je niska cena ponovne upotrebljivosti koda, ako se posmatraju performanse izvršavanja. Jedina cena koja se plaća je od jedan do nekoliko dodatnih poziva funkcija, da bi se od funkcije u ciljanom jeziku, a kroz C omotač, izvršilo pozivanje bibliotečke funkcije. Ovaj pristup ima dodatnu prednost u vidu alata koji omogućavaju automatsko obmotavanje klasa, ili funkcija, za različite programske jezike. Na Slici 1 je prikazana predložena arhitektura.

Alternativa navedenom pristupu bi bila kreiranje novog, ili upotreba postojećeg formata za komunikaciju softvera koji su pisani u različitim programskim jezicima i na različitim platformama. Primer jednog ovakvog postojećeg formata je *jezik za opis veb servisa* (eng. Web Services Description Language [7]). Ovo je XML zasnovan format koji pruža informacije o interfejsu i funkcionalnostima Veb servisa. Semantički odgovara potpisu funkcije u programskom jeziku. Ono što je nezgodno kod razmene podataka korišćenjem ovakvih formata, je što poruke koje se razmenjuju sadrže dodatne informacije koje samo opisuju podatke i time povećavaju veličine ovih poruka. Dodatno, gubi se određeno vreme na parsiranje ovih formata pa je izabran prethodno navedeni pristup.



Slika 1: Organizacija ocenjivača kao skup dinamičkih biblioteka.

Konfigurabilnost i proširivost softvera upućuju na arhitekturu softvera čiji su neki delovi implementirani kao dodaci (eng. plug-in). Dakle, da bi ocenjivači za programske jezike mogli da budu naknadno dodavani, treba ih implementirati kao dodatke koji se dinamički pokreću i biraju u toku samog izvršavanja. To će omogućiti programiranje ocenjivača za programske jezike koji će tek doći, a sve to bez izmena postojećeg sistema. Slično važi i za dodavanje ocenjivača za nove verzije programskih jezika, kao što su na primer Java 8 [8], C++14 [9] i druge.

Postoje još neke komponente ocenjivača koje bi bilo zgodno implementirati kao dodatake. Radi ilustracije, vratimo se na problem upoređivanja očekivanog izlaza programa sa dobijenim izlazom. Nazovimo deo ocenjivača koji ovaj posao obavlja *upoređivač*. Jedan od zadataka iz rekurzije, u okviru predmeta “Programiranje 2”, može da glasi:

Napisati rekurzivnu funkciju koja za dato n iscrtava trougao dimenzije n. Na primer za n=5:

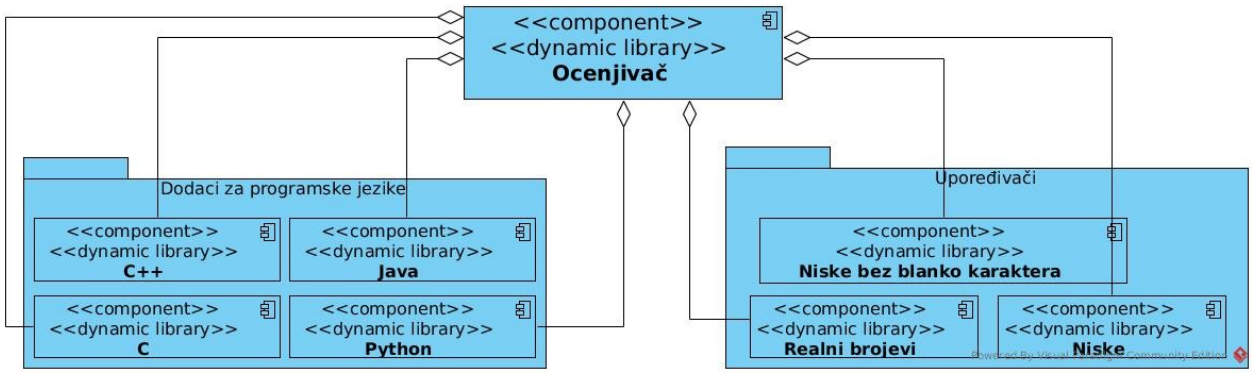
```
    +
   ++
  +++
 ++++
+++++
```

Da bismo ispravno uporedili očekivani i dobijeni izlaz dovoljno je porediti ih karakter po karakter. Međutim, taj način poređenja nije adekvatan za neke

druge vrste izlaza. Ukoliko je očekivani izlaz samo jedan broj, a studentski program ispisuje ispravan broj i novi red nakon njega, taj program ne bi bio smatran ispravnim bez obzira na to koji je broj ispisan. Jednostavnom modifikacijom upoređivača izlaza da ignoriše blanko karaktere (‘ ‘, ‘\t’, ‘\r’, ‘\n’), pre prvog i posle poslednjeg ne-blanko karaktera, dobijamo semantički ispravno poređenje i studentski program koji ispisuje broj i novi red postaje tačan. Međutim, ovakav upoređivač nam ne odgovara za prvo navedeni zadatak, zato što bi izlaz oblika:

```
+

    ++

   +++

  ++++

+++++
```

takođe bio smatran ispravnim, a on to nije. Poređenje dve slike ili dva realna broja opet uvodi novu problematiku dozvoljenog odstupanja od očekivanog rešenja. Zbog svega navedenog, implementacija više upoređivača i njihov odabir za konkretan zadatak predstavlja dobar izbor. Kako je nemoguće predvideti sve potrebne vrste upoređivača, jedino je ispravno da se oni implementiraju kao dodaci. Organizacija ocenjivača zasnovana na dodacima je prikazana na Slici 2.



Slika 2: Delovi ocenjivača implementirani kao dodaci.

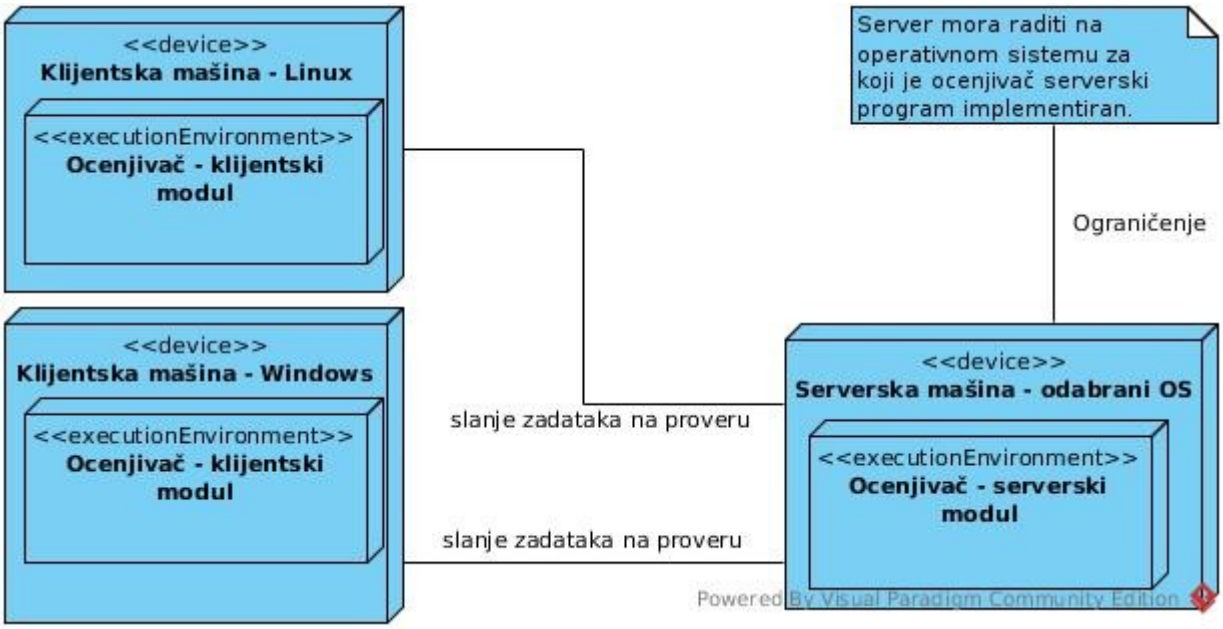
3.1.2 Fizička organizacija ocenjivača

Da bi se ocenjivač uspešno koristio nezavisno od operativnog sistema na kome radi, njegov izvorni kod mora da bude prenosiv. Ipak, za kontrolisanje toka izvršavanja studentskog programa, potrebna je konkretna podrška koja se razlikuje od operativnog sistema do operativnog sistema. To ne važi samo za kontrolu, već se i način pokretanje novog procesa razlikuje. Na primer, sistemski poziv *fork()* [10] koji postoji na Unix zasnovanim operativnim

sistemima, ne postoji na operativnom sistemu Windows. Štaviše, još uvek ga nije moguće korektno implementirati, mada postoje neki pokušaji koji imaju nedostataka [11].

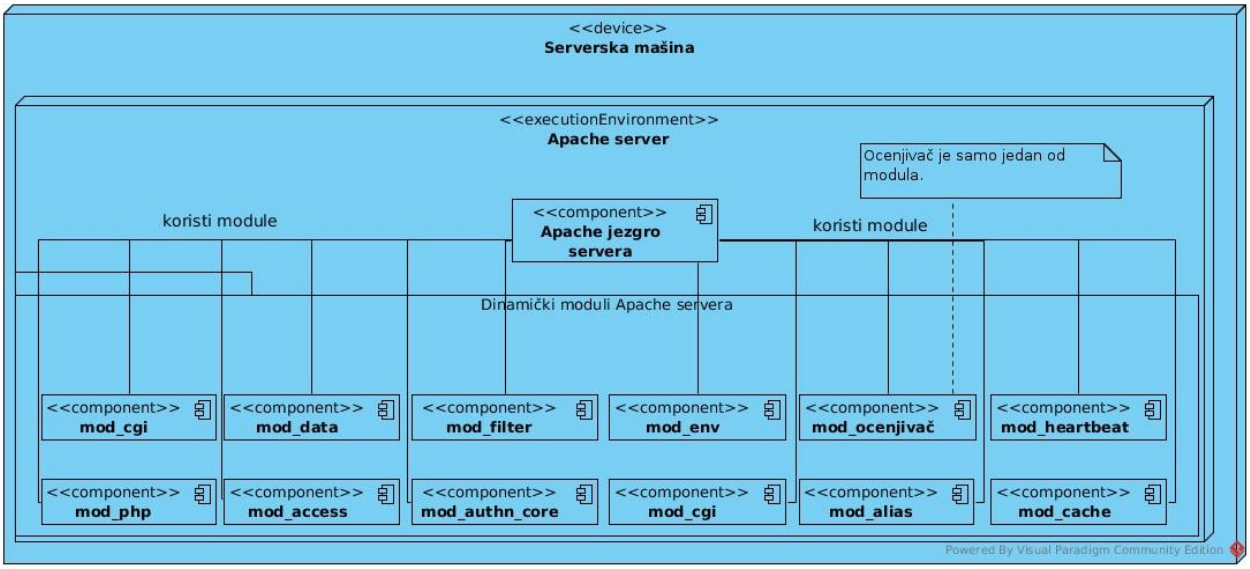
Delimično rešenje problema prenosivosti koda se postiže primenom klijent-server arhitekture (Slika 3). Ako se izvrši takva podela posla da klijentski deo ocenjivača služi samo za komunikaciju sa serverskim delom ocenjivača, onda taj deo može biti implementiran nezavisno od operativnog sistema bez dupliranja izvornog koda. Ocenjivač bi studentske kodove pokretao na serveru. Sistemski pozivi vezani za mrežnu komunikaciju se često razlikuju među operativnim sistemima, ali postoji veliki broj prenosivih biblioteka, odnosno programskih jezika, koji ove razlike prevazilaze objedinjenim interfejsom koji je isti na svim platformama. Kako kontrola procesa izvršavanja programa nije čest zahtev, za implementaciju se moraju direktno koristiti funkcionalnosti operativnog sistema (ne postoje odgovarajuće biblioteke). Na kraju, klijent-server arhitektura ne predstavlja potpuno rešenje, zato što će serverski deo raditi samo na određenom operativnom sistemu, ili određenoj grupi operativnih sistema, zbog čega se dupliranje programskog koda na duže staze ne može izbeći.

Poslednja stavka koju treba razmotriti je integracija sa Veb aplikacijom. Moderni Veb serveri kao što su Apache [12] i IIS (eng. Internet Information Services) [13] pružaju mogućnost proširenja servera novim dodacima. Tako se u slučaju servera Apache dinamičke PHP strane obrađuju korišćenjem dodatnog modula koji se zove *mod_php*. Bez njega PHP ne bi radio. Pisanje ovakvih modula, iako relativno loše dokumentovano, nije previše komplikovano, zbog toga što se ovi moduli povezuju sa serverom jako uskim interfejsom. Na prvi pogled ovo izgleda kao odličan pristup. Ocenjivač bi mogao da bude samo još jedan od servisa koje Veb server pruža (Slika 4). Broj Veb servera koji se intenzivno koriste nije veliki, pa bi se posao sveo na pisanje par modula za različite servere.



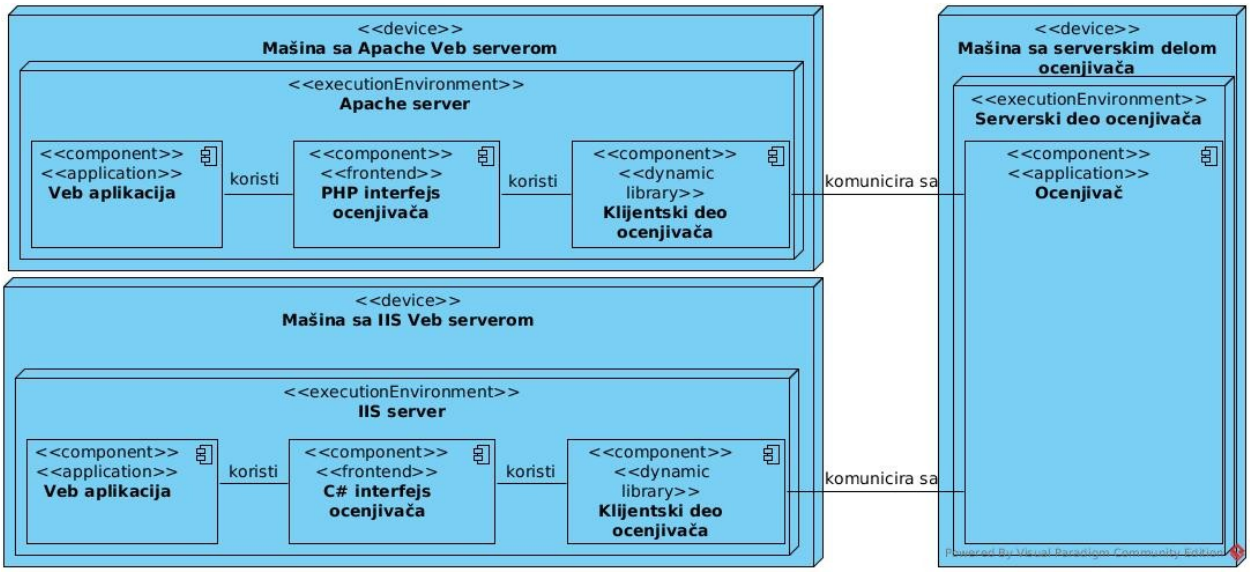
Slika 3: Klijent-server arhitektura ocenjivača

Ipak, ovaj pristup ima mnogo problema. Ilustrujmo prvi posmatranjem rada servera Apache na operativnom sistemu Linux. Inicijalno, server Apache pokreće jedan proces pod korenim korisničkim nalogom (eng. root user), koji ima sve privilegije na sistemu. Zatim se pokreće određeni broj radnih procesa (eng. worker process), kojima se predaju i u kojima se obrađuju prihvaćeni HTTP zahtevi. Ovi procesi se pokreću pod sistemskim korisničkim nalogom koji nema pune privilegije (najčešće sa nazivom *www-data*). Recimo da se u određenom trenutku vremena u jednom radnom procesu obrađuje jedan studentski program, a u drugom drugi. Kako će studentski programi biti pokrenuti pod istim korisnikom *www-data,* ništa ne sprečava prvi program da briše test primere namenjene drugom programu, ili da ih menja. Dalje, ukoliko dođe do prekida izvršavanja ocenjivača usled greške, ta greška će se propagirati na radni proces Apache servera pa će i on biti prekinut. Do ovakvog ponašanja dolazi zato što se ocenjivač poziva u okviru radnog procesa Apache servera. Gašenje radnog procesa direktno utiče na ostale funkcionalnosti Veb servera koje uopšte ne moraju da budu povezane sa ocenjivačem. Važi i obrnuto, svaka greška u Apache serveru će se propagirati na ocenjivač.



Slika 4: Ocenjivač implementiran kao serverski modul

Ocenjivač će pružiti pristup svojim funkcionalnostima iz više programskih jezika, pa se oni mogu koristiti za povezivanje sa Veb serverom. Za server Apache jezik koji se najčešće koristi za generisanje dinamičke Veb prezentacije je PHP, dok je za IIS to C#. Ostaje još da se adresira problem vezan za to u kom procesu radi ocenjivač. Ako ocenjivač pruži direktan pristup serverskoj komponenti iz navedenih programskih jezika, imaćemo iste probleme kao kod prethodnog rešenja, jer nismo promenili činjenicu da ocenjivač izvršava svoje operacije u okviru radnog procesa servera. Dakle, programski jezici moraju pristupati serveru kroz interfejs izgrađen nad klijentskom komponentom ocenjivača. Na taj način serverski proces ocenjivača može da radi u pozadini nezavisno od Veb servera i da samo prima zahteve za procesiranjem studentskih rešenja. Veb server i ocenjivač ne moraju ni da rade u okviru iste mašine (Slika 5).



Slika 5: Ocenjivač implementiran kao poseban servis

3.1.3 Sažetak

Podsetimo se na kraju u kratkim crtama koji su izbori doneti pri odabiru arhitekture ocenjivača:

1. *Ocenjivač će biti implementiran kao skup dinamičkih biblioteka* - time se indirektno omogućava pristup iz različitih programskih jezika.
2. *Biće korišćena klijent-server arhitektura* - serverski deo će raditi na tačno određenom operativnom sistemu dok će klijentski deo biti prenosiv.
3. *Klijent će vršiti samo transfer podataka* - osnovno i jedino zaduženje klijenta je slanje, primanje, obrada i pružanje informacija vezanih za studentska rešenja zadataka. Omotači za programske jezike iz kojih će ocenjivač moći da se koristi su takođe deo klijenta.
4. *Serverski deo će pokretati studentska rešenja* - kako se u okviru ovog dela pokreću studentski programi, ova komponenta ocenjivača je jedina problematična tačka sa stanovišta bezbednosti, pa se preporučuje njeno izdvajanje na posebnu mašinu.
5. *Prošireni delovi ocenjivača će biti implementirani kao dodaci* - ovim dodacima će se rukovati za vreme izvršavanja programa, pa će biti potrebne i određene konfiguracione datoteke, kojima će se regulisalo ponašanje celog sistema.

3.2 Izbor platforme

Kao što je već opisano u prethodnoj sekciji, jasno je da će serverski deo ocenjivača raditi na unapred određenoj platformi. Razmatrane su 3 grupe operativnih sistema:

1. *Unix zasnovani*
2. *Windows*
3. *Ostali*

Zbog nepristupačnosti dokumentacije, male grupe korisnika i drugih faktora koji mogu negativno da utiču na razvoj softvera, treća grupa je ovom prilikom zanemarena.

Razmotrimo dalje operativni sistem Windows. Jednu od glavnih implementacionih poteškoća predstavlja kreiranje novog procesa u kome bi se izvršavao studentski kod za jedan test primer. Ovaj proces mora da se izvršava u skladu sa već objašnjenim zahtevima za kontrolu izvršavanja. Osnovna ideja za implementaciju bezbednog izvršavanja, potencijalno štetnog studentskog programa, je da se on pokrene u procesu pod neprivilegovanim korisničkim

nalogom. Sistemski poziv koji se čini adekvatnim je *CreateProcessAsUser* [14], čiji je efekat taj da se prosleđeni korisnički nalog prijavljuje na sistem, pre izvršavanja odabranog procesa pod tim korisničkim nalogom. Windows ima takvu arhitekturu da su svi bitni resursi operativnog sistema implementirani kao *kernel objekti* (eng. kernel objects) [15]. To uključuje procese, datoteke, "cevi" (eng. pipes), imenovane cevi (eng. named pipes) i drugo. Kernel objekti su bezbedni i moguće ih je štititi korišćenjem mehanizma *ACL* (eng. Access Control List) [16]. Ovaj mehanizam resursima pridružuje listu identifikatora sesije (eng. SID) zajedno sa njihovim pravima pristupa. Kernel objekti su takođe razdvojeni različitim prostorima imena (eng. namespace) za različite *jedinstvene identifikatore koji se dodeljuju korisnicima prilikom prijavljivanja na sistem* (eng. logon SID) [17]. Zbog *ACL* zaštite, dva korisnika koji su istovremeno prijavljeni na sistem, ne mogu da pristupe međusobnim kernel objektima, što je i očekivano ponašanje. Pogledajmo sada šta se dešava prilikom sistemskog poziva *CreateProcessAsUser*. Jedinstveni identifikator prosleđenog korisnika će se razlikovati od identifikatora sesije trenutno ulogovanog korisnika. Dakle, imamo da je korisnik **A** ulogovan na sistem i pokušavamo da pokrenemo program *app.exe* kao korisnik **B** u okviru interaktivne sesije korisnika **A**. Recimo da je program koji pokrećemo najjednostavniji program sa grafičkim interfejsom. On kao takav mora da pristupi radnoj površini operativnog sistema Windows čiji se kernel objekti nalaze u prostoru imena korisnika **A,** zato što on trenutno ima interaktivnu desktop sesiju. Kako korisnik **B** ne može da pristupi navedenim kernel objektima, samim tim ni pokrenuti proces ne može da im pristupi. Na kraju, željeni proces ne bude ni pokrenut. Iako je moguće implementirati bezbedan proces na Windows-u korišćenjem dodatnih trikova (Chromium sandbox [18] i slična rešenja), pokazuje se da zato što omogućavaju jednostavniju implementaciju, Unix zasnovani operativni sistemi predstavljaju bolji izbor platforme.

Kod Unix zasnovanih operativnih sistema se izdvajaju 3 podgrupe:

1. *MacOS*
2. *Linux distribucije*
3. *FreeBSD*

MacOS je komercijalni operativni sistem koji dolazi isključivo uz računare koje je proizvela kompanija Apple pa zbog toga nije dobar kandidat. Linux i FreeBSD su podjednako dobar izbor. FreeBSD ima prednost u vidu sistemskog poziva *jail()* [19] koji implementira sigurnosne mehanizme za kontrolu procesa, dok Linux ima veći broj korisnika i veći broj distribucija. Ono što je problematično kod sistemskog poziva *jail()* je to što postoji samo na FreeBSD-u.

Njegova upotreba bi dala samo privid da je kontrola procesa dobro implementirana. To bi naročito bilo problematično u slučaju migracije koda i na Linux i Windows, koji nemaju takav mehanizam. Dakle, zbog dobre dokumentacije, velikog broja korisnika i zbog otvorenog koda operativnog sistema, za razvojnu platformu je izabran Linux.

3.3 Izbor alata

Ostalo je još da se razmotri izbor odgovarajućeg programskog jezika za implementaciju i pratećih alata. Implementacija ocenjivača zahteva korišćenje sistemskih poziva koji su direktno dostupni samo iz programskog jezika C, a samim tim i iz programskog jezika C++. Zbog toga su viši programski jezici poput Jave ili Pythona neadekvatni za ovu vrstu posla. Mogući su i neki hibridni pristupi. Oni uključuju pisanje omotača oko sistemskih poziva, pa zatim realizaciju preostalih delova ocenjivača u višem programskom jeziku. Međutim, pokazalo se da je ovaj problem memorijski zahtevan, zbog čega ni Python ni Java nisu dobar izbor, jer koriste sakupljače otpadaka (eng. garbage collector) koji onemogućavaju eksplicitno rukovanje resursima.

Izabran je programski jezik C++, zbog određenih mogućnosti ovog jezika, koje programski jezik C ne poseduje. Dodatni razlog je i prisustvo kvalitetnijih biblioteka koje su na raspolaganju. Pri implementaciji je korišćena biblioteka Boost verzija 1.54 [20].

Generisanje interfejsa prema ocenjivaču u drugim programskim jezicima je obavljeno korišćenjem alata SWIG 2.0 [21].

Za rukovanje procesom izgradnje softvera (eng. managing build system) korišćen je CMake verzija 3.2 [22], dok je za radno okruženje izabran CLion kompanije JetBrains [23]. Projekat je kompajliran kompajlerom g++ verzija 4.9 [24].

3.4 Detaljnije o arhitekturi

3.4.1 Komponente ocenjivača

Prethodno navedeni opis arhitekture ocenjivača predstavlja pogled visokog nivoa na njegovu arhitekturu. Sledi precizniji opis implementiranog sistema.

Sa stanovišta implementacije ocenjivač se sastoji od sledećih komponenti:

1. *Komponenta zadužena za čuvanje poruka (eng. logging)*
2. *Komponenta koja implementira mehanizam dodataka*
3. *Komponenta koja enkapsulira pokretanje procesa u okviru ocenjivača*

4. *Mrežna komponenta na klijentu*
5. *Mrežna komponenta na serveru*
6. *Centralni deo ocenjivača*
7. *Dodaci koji implementiraju ocenjivače za konkretne programske jezike*
8. *Upoređivači rezultata*
9. *Pozadinski proces - demon (eng. daemon)*
10. *Omotač odabranih funkcija ocenjivača pisan u programskom jeziku C*
11. *Interfejsi za pristup ocenjivaču iz raznih programskih jezika*

3.4.2 Klijentski deo

Na klijentskoj strani se nalaze odgovarajuća mrežna komponenta, C omotač i interfejsi potrebni za pristup ocenjivaču. Aplikacija koja koristi ocenjivač se takođe nalazi na klijentu. U okviru te aplikacije pozivaju se funkcije ocenjivača korišćenjem programskog jezika aplikacije. Jedini zadatak klijenta je da bude most između serverskog dela ocenjivača i pomenute aplikacije. Ilustracije radi, PHP Veb aplikacija bi mogla da pošalje izvorni studentski kod na ocenjivanje sledećom naredbom:

```
$taskId = grader::submit_task($testsDir, $fileName, $fileCont);
```

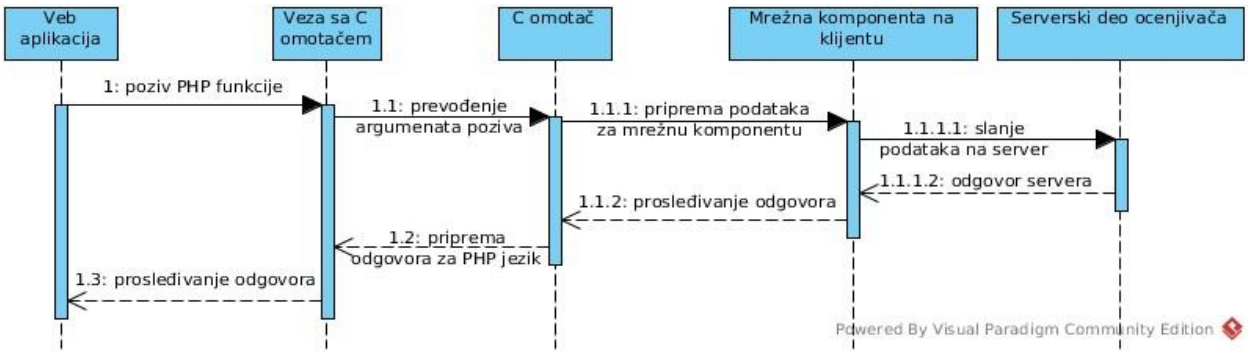
Pozvana PHP funkcija samo prosleđuje argumente odgovarajućoj C funkciji koja se nalazi u C omotaču interfejsa:

```
const char* submit_task(const char* pathToTestDir, ...);
```

Ova C funkcija zatim koristi mrežnu komponentu, koja se nalazi na klijentu, da studentski kod sa pratećim informacijama pošalje na server (Slika 6). Tok događaja je sličan i ako izvedemo istu radnju iz nekog drugog programskog jezika. Jedina razlika koja postoji prilikom poziva funkcije:

```
Grader.submitTask(pathToTestDir, sourceName, sourceContent);
```

iz programskog jezika Java, je u sloju koji prosleđuje argumente do C omotača.



Slika 6: Tok pozivanja funkcija prilikom izvršavanja aplikacije

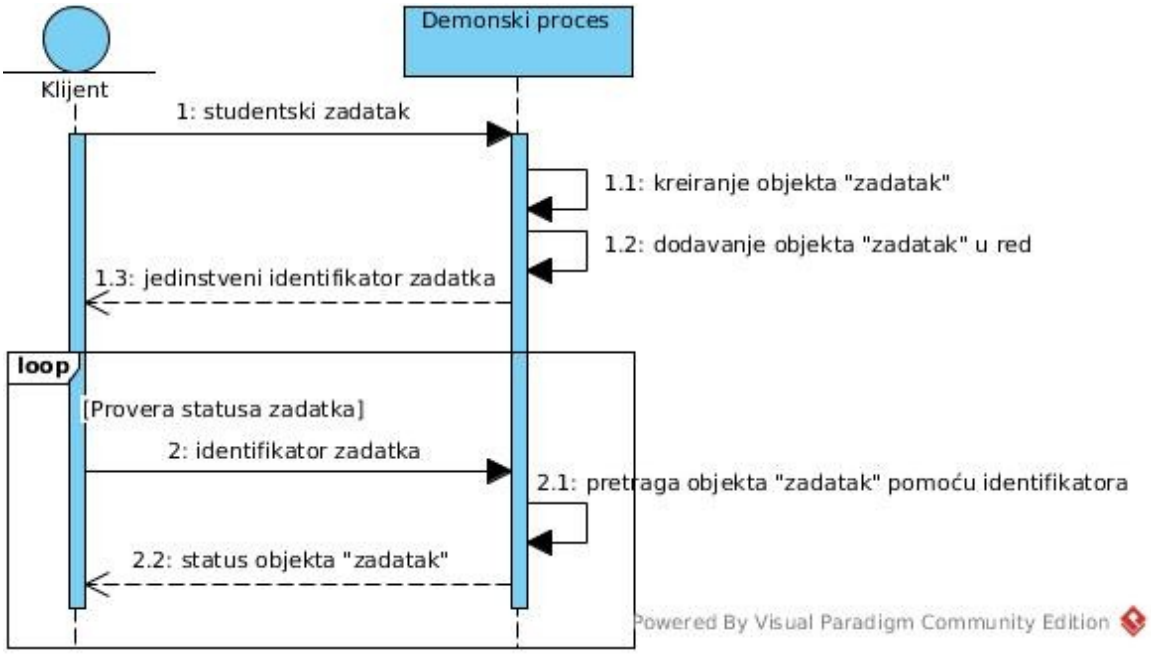
3.4.3 Serverski deo

Serverski deo je implementiran kao Unix demon [25]. Prilikom startovanja ocenjivača pokreće se jedan pozadinski proces koji radi pod korenim nalogom - nazovimo ga *demonski proces*. Ovaj proces je zadužen za upravljanje ostalim procesima i za mrežnu komunikaciju sa klijentima.

Kada pristigne studentski kod sa pratećim informacijama, demon pravi odgovarajući *objekat "zadatak"* (eng. task) u memoriji deljenoj između procesa (eng. interprocess shared memory). Dodatno pravi i koreni direktorijum za čuvanje svih izlaza iz operacija koje će biti sprovođene nad ovim objektom. Svaki objekat "zadatak" ima jedinstveni identifikator koji mu je prudružen. Ovaj identifikator se vraća kao odgovor klijentu, koji je poslao studentski zadatak na obradu, da bi ga klijentska aplikacija koristila za pristup odgovarajućem objektu. Preko ovog identifikatora može se pratiti izvršavanje studentskog programa i to:

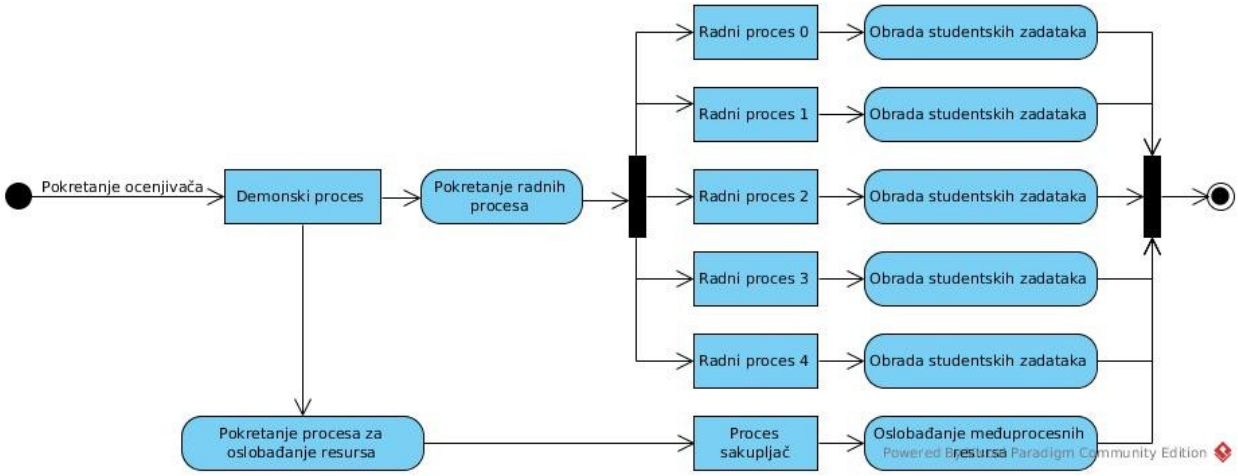
1. *Da li je slanje studentskog koda bilo uspešno* - u slučaju greške klijentska aplikacija će dobiti *null* identifikator.
2. *Da li je kompilacija kod bila uspešna* - u slučaju greške pri kompajliranju koda poruka o grešci će biti dostupna klijentskoj aplikaciji.
3. *Ispravnost rešenja na svim test primerima koji su testirani do tog trenutka* - čim se izvrši testiranje za neki test primer rezultati su odmah dostupni.
4. *Konačan izveštaj o izvršavanju programa* - uključuje informacije o izvršavanju za sve test primere.
5. *Dopunjeni izveštaj o izvršavanju programa* - za razliku od prethodnog izveštaja, ovim se pruža pristup izlazima koje je studentski program generisao.

Nakon pravljenja objekta i odgovarajućeg direktorijuma, objekat "zadatak" se zatim dodaje u red, koji se takođe nalazi u memoriji deljenoj između procesa (u daljem tekstu *međuprocesni red*). Ukoliko prilikom neke od ovih akcija dođe do greške, jasna poruka o grešci se čuva u odgovarajuću datoteku. Cela procedura je prikazana na Slici 7.



Slika 7: Obrada objekta “zadatak” - demonski proces

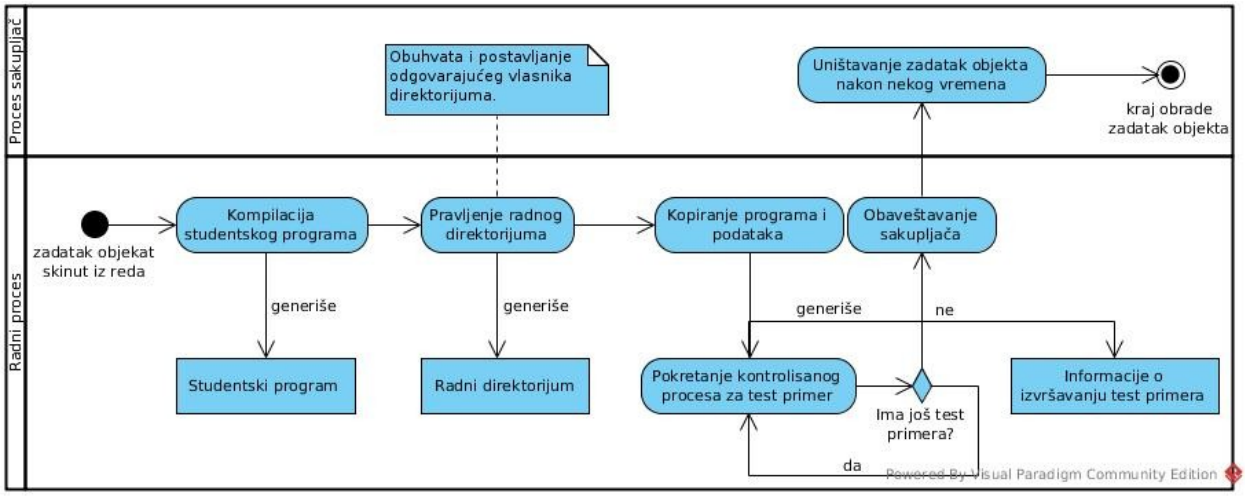
Demon prilikom pokretanja kreira dve grupe procesa (Slika 8).



Slika 8: Organizacija procesa prilikom pokretanja ocenjivača

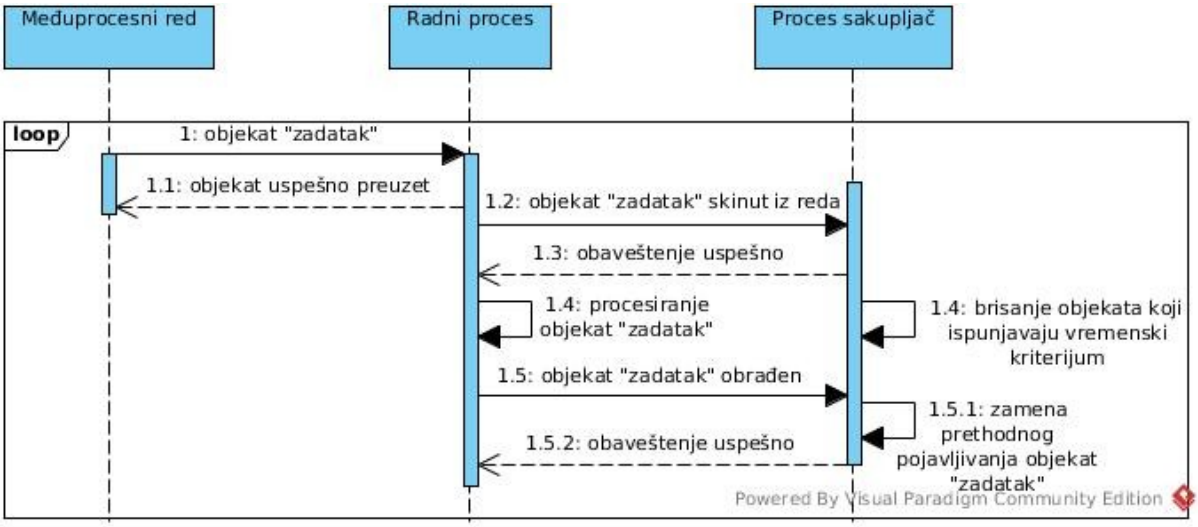
Prvu grupu čine *radni procesi*. Njihovo zaduženje je obrada objekata “zadatak”. Drugu grupu čini *proces sakupljač objekata “zadatak”* (u daljem tekstu PS), koji briše obrađene objekte “zadatak” iz međuprocesne memorije. Radni procesi preuzimaju objekte “zadatak” iz međuprocesnog reda i zatim ih obrađuju. Nakon uzimanja objekta iz reda sledi kompilacija studentskog izvornog koda u program. Zatim se priprema radno okruženje: pravi se *radni direktorijum* (eng. working directory) i u njega se kopira program zajedno sa svim datotekama i direktorijumima potrebnim za izvršavanje (ukoliko se ulaz zadaje iz sistema datoteka). Kada demonski proces pokreće radne procese, on takođe pravi i neprivilegovane sistemske korisnike koje radni procesi treba da

koriste za pokretanje studentskih programa. Dakle, radni proces postavlja odgovarajući sistemski nalog za vlasnika radnog direktorijuma. Zatim se, za svaki test primer, studentski program pokreće u kontrolisanom procesu, koji radi pod pomenutim sistemskim korisnikom. Kako različiti radni procesi imaju pridružen različiti sistemski nalog, nije moguće da studentski program pokrenut iz jednog radnog procesa, pristupi podacima potrebnim studentskom programu pokrenutom u drugom radnom procesu. Kada su svi testovi završeni, PS biva obavešten od strane radnog procesa da mu taj objekat "zadatak" više nije potreban. PS nakon nekog vremena oslobađa memoriju koja odgovara objektu "zadatak" (Slika 9).



Slika 9: Obrada objekta "zadatak" - radni proces i PS

Potrebno je naglasiti da ovde imamo jedan potencijalni problem. Ukoliko radni proces preuzme objekat "zadatak" iz međuprocesnog reda i u toku njegove obrade bude prekinut zbog greške, imaćemo curenje resursa (eng. resource leak). Kako je objekat "zadatak" uzet iz međuprocesnog reda, a nije prosleđen PS-u na uništavanje, resursi koje taj objekat ima nikad neće biti oslobođeni. Zbog toga se uvodi brisanje objekata "zadatak" iz dve etape. U prvoj etapi se ovi objekti, odmah nakon uzimanja iz reda, obeležavaju za uništavanje (obaveštavanjem PS-a). Međutim, PS ove objekte neće odmah uništiti. Oni se brišu posle znatno većeg vremenskog intervala nego objekti "zadatak" koji su uspešno obrađeni. Nakon procesiranja objekta "zadatak", radni proces ponovo obaveštava PS da objekat treba uništiti. PS proveri da li je ovaj objekat "zadatak" već bio obeležen za uništavanje i briše prethodna obeležavanja istog objekta. Ovom proverom se izbegava ponovljeni pokušaj brisanja jednog objekta. Sa druge strane, ako radni proces prekine izvršavanje zbog neke greške, objekat "zadatak" je obeležen odmah nakon uzimanja iz reda i biće ispravno obrisan nakon pomenutog većeg vremenskog intervala (Slika 10).



Slika 10: Procedura brisanja objekta “zadatak”

Još jedno zaduženje demonskog procesa predstavlja rukovanje radnim procesima. Ovo rukovanje obuhvata dve aktivnosti:

1. *Obaveštavanje o ponovnoj inicijalizaciji*
2. *Ponovno pokretanje radnog procesa u slučaju prekida rada zbog greške*

Što se prve aktivnosti tiče, ona je potrebna usled promene sadržaja konfiguracionih datoteka ili dostupnih dodataka. Dakle, u slučaju promene nekog konfiguracionog parametra, demonski proces tu promenu detektuje i obaveštava radne procese da je došlo do promene. Oni zatim reaguju tako što nakon obrade trenutnog objekta “zadatak”, ponovo učitaju sve konfiguracione parametre i osveže listu dostupnih dodataka. Ova akcija se izvršava tek nakon obrade trenutnog objekta “zadatak”, zato što je moguće poremetiti procesiranje aktuelnog objekta promenom nekog dodatka koji se za tu obradu koristi.

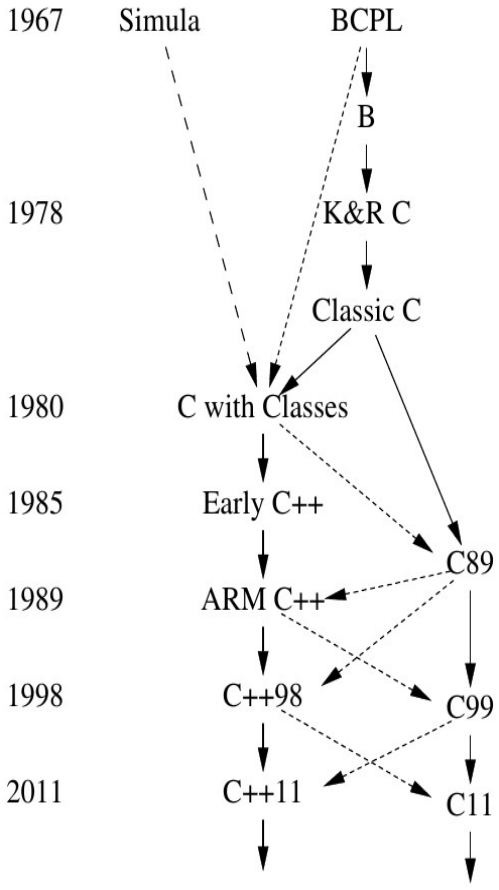
Posmatrajmo ponovno pokretanje radnog procesa u slučaju prekida rada zbog greške. Na operativnom sistemu Linux razlike između pojmova niti i procesa su minimalne, kako konceptualno, tako i implementaciono (obe koriste sistemski poziv *clone()* [26]). Prednosti niti su niža cena samog pokretanja i malo brža promena konteksta (eng. context switch). Sa druge strane procesi ne dele memorijski prostor što pruža dobru izolaciju i ostavlja manji prostor za grešku. Postoji još jedna razlika između procesa i niti koja je bitna za implementaciju ocenjivača. Kod upotrebe niti, ne postoji asinhroni direktan način da glavna nit (eng. main thread) zna da je radna nit (eng. worker thread) završila sa izvršavanjem. Ovo je moguće postići korišćenjem *uslovnih promenljivih* (eng. condition variable) i *muteksa* (eng. mutex). Sa druge strane, ako dete proces završi sa izvršavanjem, Linux kernel će poslati signal SIGCHLD roditeljskom procesu. To je posebno zgodno zato što roditelj može da

implementira *rukovaoca signalom* (eng. signal handler) za SIGCHLD, u kome će proveriti razlog prekida izvršavanja radnog procesa i ponovo pokrenuti novi radni proces.

Predstavljeno izvršavanje ocenjivača je komplikovan proces i zato je kao takvo predstavljeno iz više delova, počevši od arhitekture višeg nivoa.

3.5 Kratak uvod u programski jezik C++

C++ je nastao još davne 1979. godine. Malo preciznije, tada su u toku bili početni radovi na ovom jeziku, koji je bio poznatiji pod imenom "C sa klasama". Pod nazivom C++ je poznat od 1983. godine, dok je prva knjiga koja ga opisuje izdata 1985. Na razvoj C++-a je uticalo više programskih jezika (Slika 11).



Slika 11: Razvojni put C++-a. Slika je pozajmljena iz knjige "The C++ Programming Language 4th edition" [31].

Tokom svog razvoja C++ je pretrpeo dosta velikih promena i različitih primena, međutim danas se radi o modernom programskom jeziku koji podržava više programskih paradigmi. Kako je skoro potpuno kompatibilan sa C-om, u C++-u je moguće programirati proceduralno. Sa druge strane, C++ ima koncept klase i podršku za nasleđivanje i hijerarhijski polimorfizam, čime je u

isto vreme i objektno-orijentisan. Ako pogledamo C++11, poslednji standard C++-a koji moderni kompajleri implementiraju, korišćenjem *funktora* (eng. functor), *lambda izraza* (eng. lambda expressions) i *semantike premeštanja* (eng. move semantics) moguće je čisto funkcionalno programiranje u C++-u. Na kraju, C++ šabloni, koji su Tjuring kompletni [27], podržavaju generičku paradigmu.

Kako ovaj rad nema za cilj detaljno predstavljanje C++-a, biće korišćeni standardi C++11 i ponegde C++14 (nije do kraja implementiran u kompajlerima).

Višestrukost paradigmi nije jedina specifičnost C++-a. Prosleđivanje argumenata funkcijama, ograničenja konstantnosti, način upravljanja resursima, način obrade grešaka i šabloni. Takođe, koncept *dinamičkog vezivanja funkcija* (eng. function overriding) je drugačije osmišljen – moguće je izabrati da li želimo da do nadglašavanja dođe ili ne.

3.5.1 Argumenti funkcija

U programskom jeziku C prosleđivanje argumenata funkciji je uvek po vrednosti. Sa druge strane, u Javi se argumenti uvek prosleđuju po referenci (izuzetak su primitivni tipovi). C++ daje slobodu izbora načina prosleđivanja ovih argumenata. Dve deklaracije:

1. `void uvecaj(int& x);`
2. `int uvecaj(int x);`

predstavljaju funkcije koje različito prosleđuju argumente, prva po referenci, a druga po vrednosti.

Često se može videti i deklaracija slična sledećoj:

```
void ispis(const lista& l);
```

Ovakva jedna funkcija bi mogla da ispisuje listu na standardni izlaz. Međutim, ono što je bitno ovde je pojam konstantne reference i konstantnosti uopšte. Ono što je zagarantovano ovakvom deklaracijom, je da funkcija *ispis* neće promeniti vrednost objekta *l*. Ovu garanciju nam pruža kompajler. Posmatrajmo dalje sledeće deklaracije funkcija članica klase *lista*:

```
void lista::dodaj_u_listu(int x);
const int& lista::pocetak() const;
```

Prva funkcija neće moći da se pozove u okviru implementacije funkcije za ispis liste, zato što nije deklarisana sa ključnom reči *const*. Funkcija *lista::pocetak* svojom *const* deklaracijom garantuje da članovi objekta klase lista neće biti promenjeni u telu ove funkcije. Ovako deklarisane funkcije će moći nesmetano da se pozivaju u okviru ispisa liste. Globalni princip je da

nešto što ne garantuje nepromenljivost objekta, ne može da se koristi tamo gde se ova nepromenljivost zahteva. Upotreba *const* deklaracija funkcija i *const* referenci je koncept koji značajno smanjuje broj grešaka u programiranju.

3.5.2 Upravljanje resursima

Što se upravljanja resursima tiče, C++ pored *konstruktora* uvodi i koncept *destruktora*. To je funkcija u kojoj treba da se napiše logika oslobađanja resursa. Garancije koje jezik daje vezane za automatsko pozivanje destruktora omogućavaju bolju kontrolu nad resursima. Za lokalnu promenljivu važi da će destruktor biti pozvan na kraju opsega važenja te promenljive. To je ispunjeno čak i u slučaju da dođe do izuzetka i nije ispunjeno samo u slučaju prekida programa zbog greške. Statičke promenljive, koje imaju životni vek programa, imaju takvo ponašanje da će se destruktora pozvati i u slučaju prestanka rada programa zbog greške. Korišćenje paradigme konstruktor-destruktor predstavlja bitnu razliku u odnosu na jezike koji koriste sakupljače otpadaka. Na primer, u Javi ponašanje programa nije determinističko i ne možemo da kontrolišemo kada će memorija biti oslobođena ili datoteka zatvorena. Sasvim suprotno od toga, u programskom jeziku C++ mi diktiramo kako će se program izvršavati.

Nije samo oslobađanje resursa ostavljeno na slobodu programeru. Kopiranje objekata se takođe može prilagođavati potrebama konkretne klase. To se postiže implementacijom *konstruktora kopije* (eng. copy constructor) i *operatora dodele* (eng. assignment operator). Ilustracije radi, pokazivač na blok memorije, koji je deo nekog objekta, se može jednostavno dodeliti kopiji objekta, ili se ovaj blok može kopirati tako da dve instance klase ne sadrže pokazivač na istu memoriju, već koriste odvojenu memoriju sa jednakim sadržajem.

Nekada nam nije želja da kopiramo objekat, već samo da ga premestimo. Da bi bilo malo jasnije posmatrajmo sledeći potpis funkcije:

```
std::string std::to_string(int x);
```

Može se zaključiti da će u telu ove funkcije biti napravljen objekat klase *std::string*, koji zatim treba vratiti kao rezultat. Prilikom vraćanja ovog objekta doći će do njegovog kopiranja u lokalnu promenljivu *s* koja će primiti vrednost navedene funkcije.

```
std::string s = std::to_string(5);
```

Postavlja se pitanje zašto nepotrebno kopirati ovaj objekat? Zgodnije bi bilo premestiti ga u lokalnu promenljivu s. Ovo se postiže definisanjem *konstruktora premeštaja* (eng. move constructor) i *operatora premeštaja* (eng. move assignment operator). Efekat je izuzetno efikasno prosleđivanje objekata

po vrednosti, kod kojeg se izbegava kopiranje resursa (u navedenom slučaju bloka memorije). Da rezimiramo, potpuna kontrola nad resursima u programskom jeziku C++ se postiže sledećim elementima jezika:

1. *Konstruktor*
2. *Destruktor*
3. *Konstruktor kopije*
4. *Operator dodele*
5. *Konstruktor premeštaja*
6. *Operator premeštaja*

3.5.3 Obrada grešaka

Obrada grešaka je jedan od bitnijih aspekata programskog jezika. Programski jezik C koristi celobrojne kodove grešaka koji se upisuju u predviđenu globalnu promenljivu (promenljiva *errno*). Tekstualna poruka se od ovih kodova dobija korišćenjem funkcije *strerror*. Sa druge strane, u većini objektno-orijentisanih jezika se koriste izuzeci. Ono što je zanimljivo je da nijedan od ova dva pristupa nije superioran. Izuzeci omogućavaju jasnije izražavanje namere i bolju apstrakciju grešaka. Kodiranje grešaka u promenljivu je pogodnije kod asinhronog izvršavanja delova koda. Onog momenta kada nam rezultat bude potreban proverićemo da li je došlo do greške i onda reagovati. C++ omogućava oba pristupa. Dodatno, omogućava i njihovo kombinovanje.

3.5.4 Šabloni

Šabloni u C++ programskom jeziku se značajno razlikuju od generičkih funkcija i klasa u drugim programskim jezicima (izuzetak je programski jezik D koji je nastao od C++-a). Lista razlika je zapravo veća nego lista sličnosti. Zbog kompleksnosti ove teme, ona neće biti dublje razmatrana u radu. Šabloni će biti preciznije objašnjeni samo na mestima gde se javljaju u samom kodu ocenjivača. Za zainteresovane čitaoce, knjiga “C++ Templates: The Complete Guide” [28] predstavlja dobru referencu za C++ šablone, dok se u knjizi “Modern C++ Design” [29] mogu naći neki neobični, ali korisni načini upotrebe.

3.5.5 Dinamičko vezivanje funkcija

Pogledajmo dalje implementaciju dinamičkog vezivanja funkcija u programskom jeziku C++. Za označavanje funkcije čiji se poziv bira dinamički, koristi se ključna reč *virtual*. Osnovna ideja prilikom dizajna je bila da korisnik C++-a ne mora da izgubi na performansama zbog nečega što ne koristi.

Uopšteno gledano, korišćenje funkcija koje se biraju dinamički zahteva proširivanje objekta u memoriji. Ovo proširenje je posledica dodavanja dodatnog pokazivača koji pokazuje na *tabelu virtuelnih poziva funkcija* (eng. vtable). U Javi svi objekti imaju ovo proširenje. Kod C++-a nije tako, objekti klasa koje ne koriste dinamičko vezivanje funkcija imaju veličinu koja odgovara zbiru veličina njihovih članova, čime se izbegava nepotrebno zauzimanje memorije za nešto što neće biti korišćeno. Naravno, ako klasa u C++-u ima bar jedan virtuelni metod onda se i veličina njenih objekata proširuje pomenutim pokazivačem.

Ovaj kratak pregled specifičnosti C++-a ne obuhvata sve ono što je za ovaj jezik jedinstveno. Osnovna namena ove sekcije je bila da istakne dve činjenice. Prva je da se C++ značajno razlikuje od svih ostalih programskih jezika. Druga i mnogo bitnija je da je njegovim dizajnom i razvojem upravljala želja za efikašnošću i potpunom kontrolom nad izvršavanjem programa.

3.6 Implementacija komponenti ocenjivača

3.6.1 Komponenta za čuvanje poruka

Ocenjivač je sistem koji treba da bude dostupan različitim korisnicima. Greška u sistemu prilikom izvršavanja programa korisnika A, ne bi smela da utiče na izvršavanje programa korisnika B. Zbog toga prekidanje rada ocenjivača u slučaju greške nije opcija.

Greške je moguće podeliti u kategorije u odnosu na to koliko su one ozbiljne i koliko utiču na sistem u celini. Što se ocenjivača tiče razlikujemo:

1. *Upozorenja* – ovo su manja odstupanja od očekivanog ponašanja.
2. *Greške* – predstavljaju većinu grešaka koje se događaju u radu sistema. Ovo su konkretni problemi vezani za jednu instancu izvršavanja, kao što je na primer obrada jednog studentskog rešenja.
3. *Fatalne greške* – ove greške su najčešće posledica problema u konfiguraciji sistema. Javljaju su ukoliko ne postoje datoteke i direktorijumi potrebni za uspešno pokretanje ocenjivača.

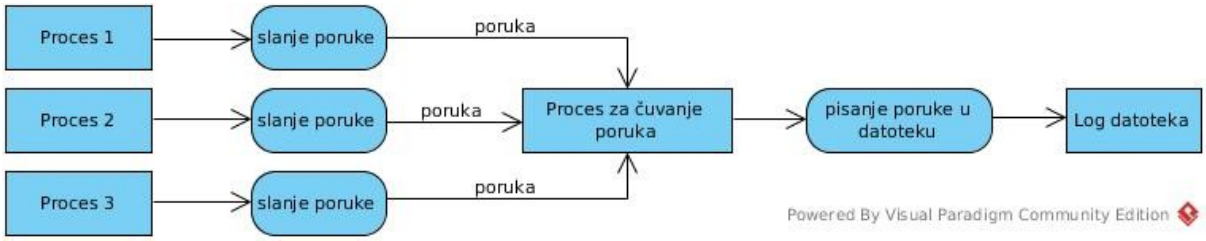
Kod složenog distribuiranog sistema, kao što je ocenjivač, greške nisu jedine poruke koje treba sačuvati. Nekada je zgodno imati još neke informacije, kao na primer kada je studentski zadatak pristigao, kada je obrađen, uništen i drugo. Čuvanje ovakvih poruka ne predstavlja opterećenje sa stanovišta performansi, a može značajno da ubrza dijagnostiku potencijalnih problema u radu sistema.

Na klijentskom delu nije rađeno čuvanje poruka. Razlozi za ovakvu odluku su

višestruki. Prvo, klijentski deo se sastoji od jednog procesa, a ne više njih, pa praćenje grešaka nije tako složen posao. Drugo, jedina logika prisutna na klijentu je transfer podataka preko mreže, pa je to i jedina stvar koju treba ispitati u slučaju potencijalnih problema. Na samom kraju, klijentska aplikacija možda ne želi da čuva bilo kakve poruke. Zbog ovih, ali i drugih razloga, na klijentskoj strani ocenjivača poruke o greškama se ispisuju na standardni izlaz za greške. Aplikacija koja koristi ocenjivač može ovaj izlaz, po želji, preusmeriti u datoteku ili potpuno ignorisati.

Komponenta za čuvanje poruka je implementirana korišćenjem sistemskog interfejsa za čuvanje poruka na operativnom sistemu Linux. Osnovni problem kod čuvanja poruka u ocenjivaču je bilo to, što se piše u jednu datoteku iz više procesa. Zbog toga je bila potrebna određena vrsta sinhronizacije između procesa. Postoje C++ biblioteke za čuvanje poruka koje ispunjavaju zadati kriterijum. Ove biblioteke nisu korišćene pri implementaciji da ocenjivač ne bi zavisio od njih. Komponenta za čuvanje poruka nije srž sistema već pomoćni deo, pa pravljenje pomenutih zavisnosti nije delovalo opravdano. Biblioteka Boost, koju projekat koristi i u druge svrhe, takođe ima deo za čuvanje poruka, međutim ovaj deo ne podržava pisanje u log datoteku iz više procesa.

Osvrnimo se malo na implementaciju višeprocesne komponente za čuvanje poruka. Najefikasniji način za zapis ovih poruka je da se pokrene poseban proces koji piše u log datoteku. Svi ostali procesi samo šalju poruke ovom procesu, korišćenjem nekog mehanizma za komunikaciju između procesa (Slika 12).



Slika 12: Arhitekura čuvanja poruka iz više procesa

Sličnu arhitekturu ima i implementacija sistemskog loga na Linux-u. Osnovni sistemski pozivi koji se na operativnom sistemu Linux koriste za logovanje poruka su [30]:

```
1. void openlog(const char *ident, int option, int facility);
2. void syslog(int pri, const char *fmt, ...)
3. void closelog(void);
```

Prvi sistemski poziv služi za obaveštavanje operativnog sistema Linux da želimo da koristimo sistemsko čuvanje. Ovim pozivom se iz tekućeg procesa otvara TCP/IP konekcija ka *sistemskom logeru* (proces koji upisuje poruke u log datoteku). Drugim sistemskim pozivom šaljemo poruke, dok poslednji služi za

zatvaranje konekcije sa *sistemskim logerom*.

Svaki od radnih procesa ocenjivača treba da ima samo jednu komponentu za logovanje koja će se koristiti u okviru ovog procesa. Zbog ovog zahteva, za implementaciju komponente za logovanje je korišćen uzorak za projektovanje Unikat (eng. Singleton). C++ ima jednostavan način za implementaciju ovog uzorka (Primer 1):

```
class logger
{
private:
  logger() {...}
  logger(const logger&) = delete;
  logger& operator=(const logger&) = delete;
  ~logger(){...}
public:
  void log(severity level, const std::string& msg) {...}
  static logger& instance()
  {
    static logger l;
    return l;
  }
};
```

Primer 1: Implementacija uzorka Unikat za projektovanje – primer logera

Delovi koda koji nisu relevantni za trenutni prikaz su izostavljeni i ovo će biti praksa i u nastavku rada. Prva stvar koju treba da primetimo je da je vraćanje reference na statičku lokalnu promenljivu ***l*** u funkciji *instance()* potpuno bezbedno, jer ova promenljiva ima životni vek aplikacije (procesa). Kopiranje *logger* objekta je sprečeno time što su *konstruktor kopije* i *operator dodele* zabranjeni ključnom rečju *delete*. Korisnik klase *logger* ne može da pozove destruktor za instancu ove klase. Kod ovakve implementacije uzorka Unikat javno dostupne metode su samo metoda koja instancira objekat i javni interfejs klase. Korišćenjem standarda C++11 ili novijeg, postoji dodatna garancija da je ovakav Unikat bezbedan i ako se koristi iz više niti (eng. thread safe). Dakle, napravili smo bezbedan Unikat u samo par linija koda.

Fokusirajmo se dalje na samu *log()* funkciju da bismo ilustrovali još jednu naprednu tehniku C++ programskog jezika – *šablone sa proizvoljnim brojem argumenata* (eng. variadic templates). Osnovno što ova tehnika pruža je prosleđivanje proizvoljnog broja argumenata različitog tipa. Jedna od primena je implementacija tipa *std::tuple<Args...>* koji predstavlja kolekciju tipova i čija je namena čuvanje raznorodnih elemenata unutar jedne strukture. Korišćenjem *std::tuple* se izbegava pisanje koda za svaku konkretnu strukturu koja sadrži par, tri ili više elemenata različitog tipa. Ilustracije radi:

```
using WorkersTableRow = std::tuple<ulong/*id*/,
                                   std::string/*ime*/,
                                   std::string/*prezime*/,
                                   my_namespace::address_t/*adresa*/,
                                   double/*plata*/>;
```

Primer 2: Korišćenje kolekcije *std::tuple* tipova

navedeni kod u Primeru 2 deklariše novi tip, koji odgovara kolonama zamišljene tabele u bazi podataka, koja čuva neke informacije o zaposlenima u firmi i to u samo jednoj naredbi.

Što se *log()* funkcije tiče ona je deklarisana sa:

```
template <typename... Args>
void log(severity level, Args... args);
```

pri čemu prvi argument predstavlja nivo ozbiljnosti poruke, a ostali argumenti se odnose na samu poruku. Nivoi ozbiljnosti poruka koje klasa *logger* razlikuje su:

1. *debug* – poruke ovog tipa se koriste samo pri pronalaženju grešaka u implementaciji sistema.
2. *info* – ove poruke omogućavaju praćenje nekih osnovnih događaja u radu ocenjivača (pristizanje novog zadatka, završetak njegove obrade i sl.).
3. *warn* – ovo su poruke upozorenja na potencijalne probleme.
4. *error* – predstavljaju izolovane greške pri obradi studentskih zadataka.
5. *fatal* – kritične greške nakon kojih sistem prestaje sa radom.

Osnovna namera pri implementaciji funkcije *log* je bila da se prosleđeni argumenti, pomoću C++ toka (eng. Stream), prebace u nisku koja će predstavljati log poruku. Posledica realizacije ove namere je da je moguće ukomponovati niz proizvoljnih tipova u jednu poruku:

```
glog_st.log(severity::fatal, "Getting worker user '", workerUser, "'
                  failed. Message: '", ::strerror(errno), "'.");
```

Primer 3: Upotreba log funkcije (**glog_st** je objekat klase *grader::logger*). Jedna generisana poruka bi mogla da izgleda ovako: *“Getting worker user 'grader_worker0' failed. Message: 'No such user'.”*.

Implementacija *log* funkcije koristi dodavanje prosleđenih argumenata u C++ tok tipa *std::stringstream* prilikom otpakivanja argumenata šablona. Otpakivanje argumenata se vrši C++ *operatorom “...”*. Argumenti se otpakuju razdvojeni zarezima. Da bi bilo jasnije, za sledeći poziv *log* funkcije:

```
glog_st.log(severity::debug, 1, 2, 3, 4, 5, "kraj");
```

korišćenjem *operatora "..."* u telu funkcije:

```
template <typename... Args>
void log(severity level, Args... args)
{
  args...
  /*
   * Ostatak koda
   */
}
```

kompajler generiše ovakav kod:

```
template <typename... Args>
void log(severity level, Args... args)
{
  1,2,3,4,5,"kraj"
  /*
   * Ostatak koda
   */
}
```

Pre konačnog navođenja cele implementacije log funkcije razmotrimo još i bezbednost *logger*-a u kontekstu niti. Jedini deo koda koji nije bezbedan je upravo *log* funkcija, koju bi trebalo zaključati pri konkurentnom izvršavanju. Umesto pisanja dve klase *logger*, jedne koja je bezbedna u kontekstu niti i druge koja nije, moguće je šablonizovati ovu klasu, odnosno objekat, po muteks tipu. Za logovanje pri konkuretnom izvršavanju treba koristiti *std::mutex* kao muteks tip, dok se u drugom slučaju koristi muteks tip oblika:

```
struct dummy_mutex
{
  void lock() {}
  void unlock() {}
  bool try_lock() { return true; }
};
```

Primer 4: Muteks tip pogodan za korišćenje u kodu sa jednom niti izvršavanja

Dakle, zaključavaćemo *log* funkciju, međutim eliminisaćemo cenu zaključavanja u kodu koji nije konkurentan korišćenjem *dummy_mutex* tipa.

U narednom primeru dat je potpun kod *log* funkcije:

```
template <typename... Args>
void log(severity level, Args... args)
{
```

```
  if (static_cast<int>(level) > static_cast<int>(m_log_level))
    return;
  std::lock_guard<mutex_type> lock(m_lock);
  empty_struct append_all_args_hack[] = { append_to_stream(args)... };
  (void) append_all_args_hack;
  ::syslog(static_cast<int>(level), "%s", m_stream.str().c_str());
  m_stream.str("");
  m_stream.clear();
}
```

Primer 5: Implementacija log funkcije. **append_to_stream** je funkcija koja prosleđeni argument dodaje u C++ tok **m_stream,** članicu klase logger. **empty_struct** je prazna struktura koja se koristi, jer konstrukcija *void niz[]* nije ispravna.

3.6.2 Komponenta koja implementira mehanizam dodataka

C++ kao programski jezik nije zgodan za implementaciju sistema dodataka. Postoji nekoliko razloga zašto nije.

Prvo, C++ nema ugrađenu podršku za refleksiju. Dobijanje informacija o tipovima za vreme izvršavanja programa je površno. U te svrhe se koriste operator *typeid* i struktura *std::type_info*, koja se dobija primenom ovog operatora. Nije moguće saznati koje metode objekat ima, niti koje članice ga čine. Od takvih informacija jedino je moguće dobiti ime klase kojoj objekat pripada. Međutim, C++ standard ne propisuje kako ime treba da izgleda u odnosu na deklaraciju klase, pa različiti kompajleri mogu implementirati različite šeme imenovanja. Dodatno, C++ standard ne zahteva da se za dve različite klase vrati različito ime.

Drugi problem je vezan za *binarni interfejs* C++ tipova (eng. Application Binary Interface – ABI). Ovaj interfejs se bavi problematikom pozivanja funkcija, polaganja struktura u memoriji, veličinama tipova podataka, njihovim poravnanjem sa adresama i drugo. Ishod toga što C++ ABI nije standardizovan je da dve dinamičke biblioteke, kompajlirane različitim kompajlerima, ne moraju da budu kompatibilne (najčešće i nisu). Ilustrujmo zašto je ovo problem jednim jednostavnim primerom: memoriju alociranu u jednoj dinamičkoj biblioteci nije moguće ispravno osloboditi u drugoj dinamičkoj biblioteci, ukoliko nisu kompajlirane istom verzijom kompajlera.

Postoji još jedan direktan problem. Prilikom implementacije sistema dodataka, dinamička biblioteka (dodatak) se mapira u memoriji odgovarajućim sistemskim pozivom. Zatim je ideja da se pozove konstruktor C++ objekta koji nam treba. Ovaj konstruktor se nalazi negde u memoriji predviđenoj za dodatak. Adresa svake funkcije, pa i ovog konstruktora, se dohvata iz memorije na osnovu imena funkcije. Tu nailazimo na treći problem, a to je

C++ preimenovanje (eng. name mangling). Sve funkcije članice klase, i uopšte sve funkcije bivaju preimenovane od strane kompajlera. Na primer, funkciju bez argumenata i povratne vrednosti *f*, će g++ kompajler preimenovati u *_Z1fv*. Preimenovanje nije standardizovano, pa samim tim kompajleri ne koriste iste šeme preimenovanja. To znači da je kompajler-nezavisno dohvatanje konstruktora klase nemoguće.

Dakle, za implementaciju dodataka potrebno je napraviti neki mehanizam koji nije podržan od strane samog C++-a. Programski jezik C ne vrši preimenovanje funkcija. Zbog toga je zgodna njegova upotreba u ovom kontekstu. Osnovu korišćenja C++ objekta neke klase čini definisanje C funkcija za njegovo pravljenje i uništavanje, kao što je prikazano u okviru Primera 6. Ove funkcije će u nastavku rada biti referisane kao *C konstruktor* i *C destruktor*.

```
extern "C" void* createObjectOfA()
{
  return static_cast<void*>(new A);
}
extern "C" void deleteObjectOfA(void* obj)
{
  delete static_cast<A*>(obj);
}
```

Primer 6: Funkcije koje prave i brišu objekat klase A. Navođenjem deklaracije **extern "C"** C++ kompajleru govorimo da tretira deklarisanu funkciju kao C funkciju.

Oko navedene upotrebe C omotača razvijen je mehanizam koji omogućava instanciranje objekta neke klase samo na osnovu imena te klase. Ova konstrukcija, inspirisana postojećom u programskom jeziku Java, je izabrana zbog jednostavnosti upotrebe (Primer 7).

```
pointer ptrToObjectOfA = lib->make_object<BaseClass>("A");
```

Primer 7: Primer instanciranja objekta klase **A** korišćenjem imena te klase. **lib** je apstrakcija dinamičke biblioteke u kojoj je klasa **A** kompajlirana. **BaseClass** je bazna klasa hijerarhije kojoj klasa **A** pripada.

Implementacija koja će biti predstavljena ima određena ograničenja:

1. *Korišćenje dve klase koje imaju isto ime, ali se nalaze u različitim prostorima imena nije direktno podržano (moguće je uz dodatni napor)*
2. *Klase koje se koriste moraju da imaju podrazumevani konstruktor*
3. *Klase koje se koriste moraju da naslede klasu dynamic::object (u slučaju hijerarhije samo bazna klasa mora da je nasledi)*

Ceo mehanizam se sastoji iz 4 elementa:

1. Bazna klasa za sve objekte koji mogu da se instanciraju dinamički –

dynamic::object.

2. Klasa koja apstrahuje dinamičku biblioteku – *dynamic::shared_lib*.
3. Pomoćne klase za automatsku registraciju C konstruktora i destruktora – *dynamic::registrer_creators*.
4. Makroi koji olakšavaju korišćenje mehanizma generisanjem potrebnih funkcija i objekata

Klasa *dynamic::object* ima dve funkcije. Prva je vraćanje pokazivača na objekat klase kroz pokazivač tipa *dynamic::object**. Druga funkcija je čuvanje statički deklarisanog kataloga, u kome kao ključ figuriše ime klase, a kao vrednost par koji sadrži adrese C konstruktora i C destruktora. Dakle, za registraciju dodatka potrebno je ažurirati ovaj katalog korišćenjem odgovarajućih vrednosti.

Klasa *dynamic::shared_lib* mapira dinamičku biblioteku u memoriji i služi za dohvatanje funkcija iz nje. Paradigmom konstruktor-destruktor objekat ove klase garantuje da će dinamička biblioteka biti uspešno mapirana, odnosno odstranjena iz memorije. To je posebno zgodno u slučaju izuzetka, jer nećemo imati curenje resursa u vidu dinamičke biblioteke koja je greškom ostala mapirana u memoriji. Dodatno, *dynamic::shared_lib* koristi registar klase *dynamic::object* i uz pomoć tih informacija instancira željene objekte.

U C++-u ne postoji pojam *statičkog konstruktora* – konstruktora koji se izvršava jednom za svaku klasu koja ga implementira, a ne za objekte. Međutim, odgovarajući efekat se može postići dodavanjem, sa ovom namenom konstruisanog, statičkog objekta klasi. Inicijalizacija ovog objekta služi kao statički konstruktor. To je ujedno i svrha klase *dynamic::registrer_creators*. Celokupan interfejs ove klase čini konstruktor, koji prima ime klase koja se registruje, adresu C konstruktora i adresu C destruktora. U okviru konstruktora, objekat klase *dynamic::registrer_creators* dodaje prosleđene argumente u registar klase *dynamic::object*. Korišćenjem jednog statičkog objekta klase *dynamic::registrer_creators,* obezbeđujemo da će se registracija klase desiti tačno jednom.

Korisnik mehanizma za dodatke bi mogao da napiše kod za C konstruktor i C destruktor za svoju klasu. Prosleđivanjem imena klase i adresa ovih funkcija objektu klase *dynamic::registrer_creators*, on bi uspešno izvršio registraciju. Međutim, krajnjeg korisnika jedino zanima da se njegova klasa registruje, a ne kakva su imena funkcija i koje objekte sve treba da napravi. Zbog toga je napisan sistem makroa koji ovaj posao radi za njega. Sve što korisnik treba uradi je da ispod deklaracije klase iskoristi jedan makro:

```
REGISTER_DYNAMIC_ST(MojaKlasa)
```

Ekspanzijom ovog makroa se dobija sledeći izvorni kod:

```
extern "C"
void __dynamic_destroy_MojaKlasa(void* obj)
{
  MojaKlasa* realObj = static_cast<MojaKlasa*>(obj);
  delete realObj;
}
extern "C"
void* __dynamic_create_MojaKlasa()
{
  MojaKlasa* newObj = nullptr;
  try
  {
    newObj = new MojaKlasa{};
  }
  catch (const std::bad_alloc&)
  {
    return nullptr;
  }
  return static_cast<void*>(newObj);
}
std::unique_ptr<dynamic::register_creators>__dynamic_object_MojaKlasa{
        new dynamic::register_creators{"MojaKlasa",
                                       &__dynamic_create_MojaKlasa,
                                       &__dynamic_destroy_MojaKlasa}
};
```

Primer 8: Kod koji generiše makro REGISTER_DYNAMIC(MojaKlasa).

Postoji još jedna stavka koju je bitno razmotriti, a to je brisanje ovako napravljenih objekata. U C++-u postoji koncept takozvanih "pametnih" pokazivača (eng. smart pointers). Ovo su zapravo klase čija je članica pokazivač na objekat. Međutim, kako ove klase implementiraju *operator ->* i *operator **, njihovi objekti se mogu koristiti baš kao pokazivači. Dodatna prednost im je i to što nije potrebno eksplicitno oslobađanje memorije na koju pokazivači pokazuju. Objekti ovih klasa će u svom destruktoru to uraditi za nas. U C++11 standardu se trenutno nalaze dve ovakve klase: *std::unique_ptr* i *std::shared_ptr* [31]. Prva klasa predstavlja pokazivač koji nije deljen između nekih drugih objekata. Druga klasa implementira baš suprotno, pokazivač na koji referiše više objekata. Broje se reference na ovaj pokazivač (uvećava se brojač prilikom dupliranja objekta). Brojač se smanjuje pozivom destruktora i kada taj broj dođe do nule memorija se oslobađa. Ovi pametni pokazivači takođe imaju još jedan argument – funkciju za oslobađanje memorije na koju

pokazivač pokazuje. Ovo je izuzetno zgodno u kontekstu gde je objekat alociran u jednoj dinamičkoj biblioteci, a u drugoj postoji potreba za njegovim uništenjem. Zbog toga, umesto da objekti klase *dynamic::shared_lib* vraćaju čist pokazivač na željeni objekat, oni će vratiti *std::unique_ptr* objekat. Prilikom pravljenja ovog objekta biće mu prosleđen C destruktor klase koji će obezbediti ispravno oslobađanje memorije na koju pokazivač pokazuje (Primer 9).

```
template <typename Derived>
unique_ptr<Derived,object_dtor> make_object(const string& className)
const
{
  // Get constructor pointer class name
  auto ctor = object::constructor(className);

  // Create new object
  Derived* dobj = static_cast<Derived*>((*ctor)());
  return unique_ptr<Derived,object_dtor>{dobj,
                                         object::destructor(className)};
}
```

Primer 9: **make_object** funkcija klase *dynamic::shared_lib*. Metoda *dynamic::object::destructor()* vraća adresu C destruktora za registrovanu klasu.

3.6.3 Komponenta za pokretanje procesa u okviru ocenjivača

Pokretanje spoljašnjeg procesa u okviru aplikacije, predstavlja čestu potrebu današnjeg složenog softveru. Međutim, bez obzira na tu činjenicu, programski jezik C++, odnosno C++ standard, ovu funkcionalnost ne omogućavaju. Preciznije, postoji funkcija *std::system* koja omogućava prosleđivanje komande koja će biti pokrenuta u podrazumenvanom terminalu operativnog sistema. Povratna vrednost ove funkcije je izlazni kod programa, odnosno prosleđene komande. Ipak, *std::system* nije mnogo korisna funkcija. Ova funkcija blokira izvršavanje tekućeg procesa dok se komanda ne izvrši. Dodatno, nije moguće proslediti podatke, poznate u originalnoj aplikaciji, kao standardni ulaz komande koja će biti pokrenuta. Takođe, nije moguće ni preusmeriti standardni izlaz te komande u recimo C++ tok ili *std::string*.

Biblioteka Boost, koja je već korišćena na projektu, ne implementira nikakvu funkcionalnost vezanu za pokretanje procesa. Bilo je pokušaja da se jedna ovakva biblioteka integriše u Boost, međutim nisu bili ispunjeni svi kriterijumi za njeno prihvatanje.

Zbog navedenog, u okviru ocenjivača je od implementirana nova biblioteka za pokretanje spoljašnjih procesa i interakciju sa njima. Dodatno, implementirana je i klasa koja omogućava bezbedno pokretanje programa koji

su potencijalno maliciozni.

Klasa koja služi za pokretanje spoljašnjeg procesa, bez ikakvih restrikcija nad tim procesom, je *grader::process*. Interfejs je modelovan po uzoru na *std::thread*, zbog čega se odmah prilikom konstrukcije objekta pokreće novi proces. Konstruktori klase *grader::process* su dati u Primeru 10.

```
explicit process();
explicit process(std::function<void(void)> executable,
                 const std::string& workingDir = std::string(),
                 const environment& e = environment());
explicit process(const std::string& executable,
                 const std::vector<std::string>& args = no_args,
                 pipe_ostream* stdinStream = nullptr,
                 pipe_istream* stdoutStream = nullptr,
                 pipe_istream* stderrStream = nullptr,
                 const std::string& workingDir = std::string(),
                 const environment& e = environment());
```

Primer 10: Konstruktori klase grader::process.

Konstruktor bez argumenata je implementiran zbog kompletnosti samog tipa podataka i ne pokreće nikakav proces. Funkcionalnosti koje druga dva konstruktora pružaju, pored pokretanja programa ili funkcije, su:

1. *Prosleđivanje argumenata komandne linije*
2. *Preusmeravanje standardnog ulaza*
3. *Preusmeravanje standardnog izlaza*
4. *Preusmeravanje izlaza za greške*
5. *Pokretanje procesa u odabranom radnom direktorijumu*
6. *Prosleđivanje okruženja (eng. environment variables)*

Recimo da hoćemo da pokrenemo neki program, čiju putanju znamo, u spoljašnjem procesu. Na operativnom sistemu Linux to se realizuje kombinacijom sistemskog poziva *fork()* i nekog od sistemskih poziva *exec()*. Nakon izvršavanja funkcije *fork()*, u dete procesu se poziva *exec()* sa prosleđenim parametrima. Poziv jedne od funkcija iz familije *exec()*, zamenjuje trenutni proces deteta novim procesom u kome se izvršava prosleđeni program. Primetimo da se roditeljski proces ne blokira nakon poziva funkcije *fork()*, već se normalno nastavlja. Implementacija pokretanja procesa je data u Primer 11. Na operativnom sistemu Windows funkcija *fork()* ne postoji, međutim Windows ima sistemski poziv *CreateProcess()* [32] koji pruža potrebnu funkcionalnost.

```
process::process(const string& executable, const vector<string>& args,
                 pipe_ostream* stdinStream, pipe_istream* stdoutStream,
                 pipe_istream* stderrStream,
                 const string& workingDir, const environment& e)
{
  // Get environment variables buffer
  const vector<string>& envVars = e.data();

  // Get working directory
  const char* cWorkDir = workingDir.empty() ? nullptr :
                         workingDir.c_str();

  // Get command line arguments
  vector<char*> argv = command_line_args(executable, args);

  // Create pipes
  grader::pipe stdinPipe, stdoutPipe, stderrPipe;

  // Fork child
  handle child = fork();
  if (child == invalid_handle)
  {
    THROW_SMART(process_exception, ::strerror(errno));
  }
  else if (child) // Parent
  {
    // Initialize child handle and set exception handling
    m_childHandle = child;
    autocall handleStreamFailure(
      [&](){
        stdinPipe.close_both();
        stdoutPipe.close_both();
        stderrPipe.close_both();
        destroy();
    });

    // Open pipe streams and release exceptions safety
    set_up_parent_pipes(stdinPipe, stdoutPipe, stderrPipe,
                        stdinStream, stdoutStream, stderrStream);
    handleStreamFailure.release();
  }
  else //Child
  {
    // Set up working directory
    if (cWorkDir)
    {
      if (::chdir(cWorkDir) == -1)
```

```
        THROW_SMART(process_exception, ::strerror(errno));
    }
    // Set up environment variables
    for (const auto& envVar : envVars)
    {
      const char* cEnvVar = envVar.data();
      if (::putenv(const_cast<char*>(cEnvVar)) != 0)
        THROW_SMART(process_exception, ::strerror(errno));
    }

    // Set up child pipes
    set_up_child_pipes(stdinPipe, stdoutPipe, stderrPipe,
                       stdinStream, stdoutStream, stderrStream);

    // Replace process image
    ::execvp(argv[0], argv.data());
    THROW_SMART(process_exception, ::strerror(errno));
  }
}
```

Primer 11: Implementacija jednog konstruktora klase *grader::process*.

Fokusirajmo se dalje na implementaciju klase koja omogućava bezbedno pokretanje potencijalno malicioznih programa – *grader::safe_process*. Ova klasa ne može biti sama sebi dovoljna, već se mora osloniti na postojanje određenih elemenata van samog koda klase.

Operativni sistemi imaju pojam *korisnika* koji se koristi za regulisanje bezbednosti u vidu prava pristupa. Ova prava određuju da li korisnik može da čita, piše ili pokrene neku datoteku kao program. Pre bilo koje akcije korisnika jezgro operativnog sistema vrši provere ovih prava. Jedini izuzetak je *koreni korisnik* za koga se provere preskaču (on ima sva prava).

Da bi se ostatak sistema izolovao od procesa koji pokreće klasa *grader::safe_process*, ovaj proces mora da bude pokrenut pod korisničkim nalogom sa odgovarajućim privilegijama. Kako bi bilo skupo da objekti ove klase svaki put prave novog korisnika, podrazumeva se da je on već napravljen i očekuje se da bude prosleđen konstruktoru klase.

Nakon izolovanja procesa od sistema potrebno je ograničiti broj datoteka koje ovaj proces može da otvori, kao i broj procesa koje može da pokrene. Na operativnom sistemu Linux, za čitanje i postavljanje ograničenja resursa koriste se sistemski pozivi *getrlimit()*, *setrlimit()* i *prlimit()* [33]. Prvi služi za čitanje ograničenja trenutnog procesa, drugi za postavljanje ograničenja za trenutni proces, dok poslednji služi za postavljanje ograničenja za neki treći proces čiji identifikator znamo (eng. process id – *pid*).

Što se memorijskog i vremenskog ograničenja tiče, ispostavlja se da funkcije za postavljanje limita nisu adekvatne.

Koncetrišimo se prvo na memorijsko ograničenje. Funkcija *setrlimit()* ima mogućnost postavljanja dva različita ograničenja koja se tiču memorije. Korišćenjem opcije *RLIMIT_DATA*, u pozivu funkcije *setrlimit()*, ograničava se memorija programa koja se odnosi na segment podataka i na programski hip. Na prvi pogled ova opcija izgleda odgovarajuće. Međutim, pogledajmo najjednostavniji C program koji alocira neku memoriju. Moderne *malloc()* implementacije ne koriste samo hip memoriju za alociranje objekata. U slučaju većih alokacija koristi se direktno mapiranje memorije koje kernel pruža. Na ovako mapiranu memoriju *RLIMIT_DATA* opcija ne utiče. Drugi način ograničavanja studentskog programa je korišćenje opcije *RLIMIT_AS*. Upotrebom ove opcije postavlja se limit na celokupnu virtuelnu memoriju pokrenutog procesa. Ovim se rešava pomenuti problem vezan za *malloc()* funkciju. Ipak ni ovo nije dobar pristup. Za jedan proces, na operativnom sistemu Linux razlikujemo 4 interpretacije zauzeća memorije:

1. *Veličina virtuelne memorije (VSS)* – odgovara ukupnom adresnom prostoru procesa (uključujući i memoriju koja se ne nalazi u RAM-u).
2. *Veličina rezidentne memorije (RSS)* – odgovara memoriji koju proces koristi, a koja se nalazi u RAM-u.
3. *Proporcionalna veličina rezidentne memorije (PSS)* – modifikacija vrednosti RSS. Dobija se drugačijim računanjem memorije koja odgovara mapiranim dinamičkim bibliotekama koje proces koristi. Na primer, ako mapiranu biblioteku *lib.so*, koja zauzima 2MB memorije, koriste dva procesa, računaće se samo 1MB po procesu, umesto 2MB. Umesto da se računa celokupna memorija mapirane dinamičke biblioteke, ova memorija se deli na broj procesa koji biblioteku koriste.
4. *Veličina privatne memorije procesa (USS)* – odgovara memoriji koju proces koristi za svoje potrebe. Ovaj broj predstavlja memoriju koja će biti vraćena operativnom sistemu nakon završetka programa (procesa).

Što se tiče praćenja memorije za studentske programe, očigledno je da je *USS* vrednost koja nas zanima. Nažalost, ne postoji način da se ova vrednost ograniči korišćenjem *setrlimit()* funkcije. Sa druge strane, postoji način da se ova vrednost izračuna za bilo koji proces u bilo kom trenutku. *Sistem pseudo datoteka* (eng. pseudo-file system) */proc* [34] sadrži informacije o izvršavanju svih procesa. Za proces sa identifikatorom 1720, podaci vezani za upotrebu memorije se nalaze u pseudo datoteci na adresi */proc/1720/statm*. Ove informacije se mogu iskoristiti za ograničavanje privatne memorije procesa. Međutim, praćenje memorije zahteva od roditeljskog procesa čitanje pseudo

datoteke, koja odgovara dete procesu, u petlji, zbog čega bi roditeljski proces morao da blokira. Razlog za čitanje u petlji je taj, što nije moguće koristiti Linux API za asinhrono čekanje na promene datoteke, jer se radi o pseudo datoteci koja ne postoji na disku i čiji sadržaj kernel generiše po potrebi.

Pre prikazivanja same implementacije *grader::safe_process* klase, osvrnimo se na problem vremenskog ograničenja. Funkcija *setrlimit()* ima opciju *RLIMIT_CPU* kojom se može ograničiti procesorsko vreme u sekundama. Međutim ne postoji način da se ograniči vreme u milisekundama, što je razuman zahtev. Jedno od rešenja je da se u roditeljskom procesu pokrene *merač vremena* (eng. timer), korišćenjem sistemskog poziva *setitimer()* [35], nakon čijeg isteka će roditeljski proces nasilno prekinuti rad dete procesa. Ovde se javlja sledeći problem: ako roditeljski proces pokrene više potomaka, kako će on znati kom potomku (dete procesu) je isteklo vreme? Kako je potpis *funkcije koja će rukovati istekom vremena* (eng. alarm signal handler) takav, da prima samo jedan podatak tipa *int*, ne postoji jednostavan način da se to isprati (koje dete proces je prekoračilo vremensko ograničenje). Dodatno, postoje i problemi vezani za sinhronizaciju merača vremena.

Rešenje navedenih problema je arhitektura klase *grader::safe_process* koja se sastoji od 3 generacije procesa. Dakle, razlikovaćemo *roditeljski proces*, *nadzorni proces* (dete proces) i *izvršni proces* (unuk proces). Roditeljski proces odgovara procesu same aplikacije. Nadzorni je proces zadužen za kontrolu izvršnog procesa. Bavi se pokretanjem merača vremena i čitanjem pseudo datoteke u kojoj se nalaze informacije o upotrebi memorije izvršnog procesa. Izvršni proces odgovara nebezbednom studentskom programu. Nakon završetka ovog procesa, nadzorni proces analizira izvršavanje i obustavlja svoj rad izlaznim kodom koji je odgovarao ponašanju izvršnog procesa. Na primer, ako je izvršni proces prekinuo izvršavanje sa izlaznim kodom 1, tada i nadzorni proces prekida svoje izvršavanje istim izlaznim kodom. Sa druge strane, ako izvršni proces prekorači vremensko ograničenje, nadzorni proces će završiti svoj rad predefinisanim izlaznim kodom. Ovaj izlazni kod dalje postaje dostupan u roditeljskom procesu, koji donosi zaključke o izvršavanju izvršnog procesa na osnovu njega. Konstruktor klase *grader::safe_proces* dat je kao Primer 12.

```
safe_process::safe_process(const string& executable,
                           user_id unprivilegedUser,
                           const string& jailDir,
                           const string& workDir,
                           rlim_t timeLimit,
                           rlim_t memoryLimit,
                           const vector<string>& args,
                           pipe_ostream* stdinStream,
```

```
                              pipe_istream* stdoutStream,
                              pipe_istream* stderrStream,
                              const enviroment& e)
: process(), m_exitCode(invalid_exit_code)
{
  const vector<string>& envVars = e.data();
  const char* cJailDir = jailDir.empty() ? nullptr : jailDir.c_str();
  const char* cWorkDir = workDir.empty() ? nullptr : workDir.c_str();
  vector<char*> argv = command_line_args(executable, args);

  // Fork child
  grader::pipe stdinPipe, stdoutPipe, stderrPipe;
  handle child = fork();
  if (child == invalid_handle)
  {
    THROW_SMART(process_exception, ::strerror(errno));
  }
  else if (child) // Parent
  {
    m_childHandle = child;

    // Prepare safety
    autocall handleStreamFailure{[&](){
      stdinPipe.close_both();
      stdoutPipe.close_both();
      stderrPipe.close_both();
      destroy();
    }};

    set_up_parent_pipes(stdinPipe, stdoutPipe, stderrPipe,
                        stdinStream, stdoutStream, stderrStream);
    handleStreamFailure.release();
  }
  else //Child
  {
    // Fork again
    pid_t grandchild = ::fork();

    if (grandchild == invalid_handle)
      THROW_SMART(process_exception, ::strerror(errno));

    if (grandchild) // Still in child
    {
      safe_process::grandchild_pid = grandchild;
      autocall killDescendants = []() {
        // Kill all kids of grandchild
```

```
      ::kill(-safe_process::grandchild_pid, SIGKILL);

      // Kill grandchild in case that process group change failed
      ::kill(safe_process::grandchild_pid, SIGKILL);
    };

    // Change group id for grandchild
    if (::setpgid(grandchild, grandchild) == -1 && errno != EACCES)
      THROW_SMART(process_exception, ::strerror(errno));

    // Set SIGCHLD handler
    init_child_finished_signal_handler();

    // Set SIGTERM handler
    if (::signal(SIGTERM, &handle_sigterm) == SIG_ERR)
      THROW_SMART(process_exception, ::strerror(errno));

    stdinPipe.close_both();
    stdoutPipe.close_both();
    stderrPipe.close_both();

    time_limit(timeLimit);
    memory_limit(memoryLimit);
  }
  else // In grandchild
  {
    // Change group id for
    if (::setpgid(0, 0) == -1)
      THROW_SMART(process_exception, ::strerror(errno));

    // Ignore SIGHUP that will occur due to chroot and setuid calls
    if (::signal(SIGHUP, SIG_IGN) == SIG_ERR)
      THROW_SMART(process_exception, ::strerror(errno));

    // Set up jail
    if (cJailDir && ::chdir(cJailDir) == -1)
      THROW_SMART(process_exception, ::strerror(errno));
    if (cJailDir && ::chroot(cJailDir) == -1)
      THROW_SMART(process_exception, ::strerror(errno));
    if (::setuid(unprivilegedUser) == -1)
      THROW_SMART(process_exception, ::strerror(errno));

    set_limits();
    secure_process();

    // Set up working directory
    if (cWorkDir && ::chdir(cWorkDir) == -1)
```

```
        THROW_SMART(process_exception, ::strerror(errno));

      // Set up environment variables
      for (const auto& envVar : envVars)
      {
        const char* cEnvVar = envVar.data();
        if (::putenv(const_cast<char*>(cEnvVar)) != 0)
          THROW_SMART(process_exception, ::strerror(errno));
      }

      set_up_child_pipes(stdinPipe, stdoutPipe, stderrPipe,
                         stdinStream, stdoutStream, stderrStream);

      // Replace process image or throw on return
      int exitCode = ::execvp(argv[0], argv.data());
      THROW_SMART(process_exception, ::strerror(errno));
    }
  }
}
```

Primer 12: Konstruktor klase *grader::safe_process*.

Nakon prikazivanje samog konstruktora, potrebno je naglasiti još neke detalje. Recimo da studentski program pokrene određene procese. Ako ovaj program bude nasilno prekinut, usled prekoračenja nekog od ograničenja, a procesi koje je on napravio nastave da rade, doći će do curenja resursa u vidu zaostalih procesa. Radi lakšeg upravljanja mnoštvom procesa, uvodi se pojam *grupe procesa*. Svaka grupa procesa ima svog lidera čiji je identifikator ujedno i identifikator cele grupe. Korišćenjem ovog identifikatora i sistemskog poziva *kill()*, moguće je čitavoj grupi procesa poslati signal. Ova činjenica je iskorišćena, pa se izvršni proces postavlja za lidera grupe. On će biti lider svim procesima koje napravi, ali ne i nadzornom procesu i roditelj procesu. Kao posledica toga, slanje signala SIGKILL će ispravno prekinuti izvršni proces i sve njegove potomke. Međutim, ukoliko studentski program napravi proces i postavi ga za lidera svoje grupe, taj proces nece biti prekinut. Zato je sledeći korak zabrana promene lidera grupe studentskom programu i njegovim potomcima. To se postiže blokiranjem sistemskog poziva *setpgid()*, kojim se postavlja lider grupe. Operativni sistem Linux, počevši od kernela sa verzijom 3.5, omogućava primenu specijalnih akcija ukoliko proces pokuša da izvrši određeni sistemski poziv. Ilustracije radi, moguće je nasilno prekinuti proces ukoliko pokuša da piše bilo gde osim na standardni izlaz. Biblioteka *seccomp* [36] koristi ove funkcionalnosti i pruža interfejs koji je lakši za korišćenje od interfejsa koji sam kernel pruža. Blokiranje sistemskog poziva *setpgid()* je implementirano u funkciji *secure_process()* klase *grader::safe_process* (Primer

13).

```
void safe_process::secure_process() const
{
  // Initialize library structure
  scmp_filter_ctx ctx = seccomp_init(SCMP_ACT_ALLOW);
  if (!ctx)
    THROW_SMART(process_exception, ::strerror(errno));

  // Fail setpgid() with EPERM
  if (seccomp_rule_add(ctx,
                       SCMP_ACT_ERRNO(EPERM),
                       SCMP_SYS(setpgid),
                       0)
      < 0)
    THROW_SMART(process_exception, ::strerror(errno));

  // Load rules into kernel
  if (seccomp_load(ctx) < 0)
    THROW_SMART(process_exception, ::strerror(errno));
}
```

Primer 13: Funkcija koja blokira sistemski poziv *setpgid()*. Postavljena akcija je da poziv ne uspe uz grešku EPERM.

Da rezimiramo, *grader::safe_process* sa aspekta sigurnosti pruža sledeće:

1. *Izolaciju sistema datoteka od procesa (osim datoteka koje su eksplicitno dozvoljene)*
2. *Nizak nivo privilegija procesa u kontekstu komandi koje može da izvrši*
3. *Ograničavanje broja potomaka koje proces može da ima*
4. *Ograničavanje broja datoteka koje proces može da otvori*
5. *Vremensko ograničenje izvršavanja*
6. *Memorijsko ograničenje izvršavanja*
7. *Onemogućava nastajanje zombi procesa*
8. *Onemogućava nastajanje procesa koji su siročići (eng. orphaned processes)*

3.6.4 Mrežne komponente ocenjivača

Postoje dve mrežne komponente ocenjivača, ona koja pripada demonu i ona koja se nalazi na klijentu. Kako su obe komponente izgrađene direktno iznad TCP/IP sloja, bilo je potrebno ustanoviti protokol komunikacije. Podsećanja

radi, kada se koristi mrežna komunikacija na ovom nivou, osnovno sredstvo komunikacije predstavlja soket (eng. socket). Soketi se donekle ponašaju kao datoteke, iz njih se može čitati i u njih pisati. Razliku u odnosu na datoteke predstavlja činjenica da nemamo direktan način da proverimo koja količina podataka će stići. Sa druge strane, ako pokušamo da pročitamo više bajtova (eng. byte) nego što je primljeno, program će se blokirati na toj operaciji. Dakle, potreban je određeni protokol za komunikaciju (kao što je na primer HTTP). Korišćen je jednostavan protokol:

1. Ukoliko je veličina sadržaja poruke nepromenljiva, za taj tip poruka, samo se sadržaj i šalje. Jednu ovakvu poruku predstavlja i identifikator objekta “zadatak” koji je uvek dužine 36 bajtova.
2. Kada sadržaj poruke varira, na početak poruke se dodaje broj bajtova originalnog sadržaja poruke. Primalac pročita 4 bajta i na osnovu njih zna koliko još bajtova treba da pročita da bi primio celu poruku.

Komunikacija je jednosmerna pa klijent uvek zahteva informaciju ili uslugu, a server samo šalje odgovarajući odgovor. Shodno tome, za svaku operaciju ocenjivača klijent koristi odgovarajuću konstantu koja ovu operaciju kodira. Sadržaj poruke započinje jednom od ovih konstanti, za kojom slede potrebni argumenti za izvršavanje operacije na serverskoj strani.

Za samu implementaciju opisane komunikacije korišćena je biblioteka Boost ASIO [37]. Ova moderna biblioteka je implementirana korišćenjem uzorka za projektovanje Proaktor [38]. Bez detaljnijeg opisivanja ove biblioteke i navedenog uzorka za projektovanje, potrebno je naglasiti neke njene osobine koje su bitne za ocenjivač. Boost ASIO omogućava sinhrono i asinhrono izvršavanje mrežnih operacija. Prilikom asinhronih operacija, a zbog upotrebe uzorka Proaktor, jedna nit izvršavanja može da obradi više asinhronih zahteva. Ovim se postiže apstrahovanje konkurentnog programiranja i izbegava potreba za pokretanjem većeg broja niti kako bi se povećala propusnost (konkurentnost) aplikacije.

Vratimo se na ocenjivač i klijentsku i serversku mrežnu komponentu. Dok čeka na odgovor servera, klijent nema nikakvog posla koji može da obavlja. Kao posledica toga su za slanje i primanje poruka na klijentu korišćene sinhrone mrežne operacije. Sa druge strane, jedna serverska nit izvršavanja može da opsluži više asinhronih zahteva (kao što je napomenuto u prethodnom pasusu). Zbog toga se na serverskoj strani koriste asinhrone mrežne operacije. Kod asinhronog izvršavanja delova programa, često je potrebno obraditi rezultat ovog izvršavanja. Uzorak Proaktor ima učesnika koji baš za to služi - to je *rukovalac završetkom* (eng. completition handler). Ovo je u biblioteci Boost ASIO implementirano kao funkcija koja će se pozvati

po završetku asinhrone operacije. Korišćenjem standarda C++11, funkcije ovog tipa se jednostavno mogu zadati lambda izrazima. Primer jednostavnog slanja poruke od servera ka klijentu je dat u Primer 14.

```
void tcp_connection::do_write()
{
  // Send back response
  auto self = std::shared_from_this();
  aio::async_write(m_socket, aio::buffer(m_response),
      [this, self](sys::error_code ec, size_t bytesWritten)
      {
      if (ec)
            glog_st.log(severity::error, "Error occurred when data was"
                                         " sent to client, message: '",
                                         ec.message(), "'.");
      else
            this->do_read();
      }
  );
}
```

Primer 14: Funkcija koja šalje poruku od servera ka klijentu. Objekat **tcp_connection** predstavlja jednu konekciju između servera i klijenta, pri čemu je **m_socket** odgovarajući soket članica klase. **async_write** je funkcija za asinhrono pisanje biblioteke Boost ASIO. Poruka koja se šalje nalazi se u članici konekcije **m_response**. Kako se objekat **this** koristi u rukovaocu završetkom, on je takođe obuhvaćen (eng. captured) lambda izrazom.

3.6.5 Omotač odabranih funkcija ocenjivača pisan u programskom jeziku C

Ocenjivač je sistem za sebe i kao takav treba da bude zatvoren za spoljašnje sisteme. Međutim, oni moraju biti u stanju da koriste ocenjivač kao komponentu. Zbog toga je pažljivo izabran interfejs koji će biti dostupan spoljašnjim sistemima. Odabir ovog interfejsa je bio vođen funkcionalnostima koje je potrebno pružiti. Ove funkcionalnosti su:

1. *Čuvanje direktorijuma sa test primerima na mašini na kojoj se nalazi ocenjivač* – kao što je već navedeno, ocenjivač i klijentska aplikacija ne moraju da se nalaze na istoj fizičkoj odnosno virtuelnoj mašini. Da bi se izbeglo ponovljeno slanje istog test direktorijuma prilikom postavljanja različitih studentskih rešenja na proveru, ocenjivač pruža mogućnost da se ovaj test direktorijum jednom snimi, a kasnije više puta koristi sa mašine gde je ocenjivač. Na ovaj način se značajno štedi na transportu između mrežnih komponenti ocenjivača. Funkcija koja služi za izvršavanje ove operacije je:

```
extern "C" bool save_tests(const char* relativePathToTestDir,
                           const char* zippedData, size_t len);
```

Dakle, ova funkcija prima relativnu putanju do test direktorijuma, sadržaj kompresovanog direktorijuma i njegovu veličinu. Prosleđuje se relativna putanja zato što klijentske aplikacije ne bi trebale da znaju ništa o mašini na kojoj se nalazi ocenjivač. Serverski deo ocenjivača ovu relativnu putanju pretvara u apsolutnu, dodavanjem putanje baznog direktorijuma na početak relativne putanje. Za bazni direktorijum */opt/testdirs* i prosleđenu relativnu putanju *webapp/assignment1* apsolutna putanja otpakovanog test direktorijuma na mašini gde se nalazi ocenjivač je */opt/testdirs/webapp/assignment1*. Sama logika ove funkcije je u potpunosti enkapsulirana u mrežnoj komponenti klijenta (klasa *grader::tcp_client*), što se može videti u Primeru 15.

```
bool save_tests(const char* relativePathToTestDir,
                const char* zippedData, size_t len)
{
  return tcp_client::instance().save_tests(relativePathToTestDir,
                                           zippedData, len);
}
```

Primer 15: Implementacija funkcije za čuvanje test direktorijuma na udaljenoj mašini.

2. *Postavljanje studentskih rešenja na ocenjivanje (eng. solution submission)* – ova funkcionalnost je implementirana funkcijom:

```
extern "C" const char* submit_task(const char* testDirPath, ...);
```

Primetimo da je ovo C funkcija koja prima proizvoljan broj argumenata. U zavisnosti od broja prosleđenih argumenata implementirane su različite akcije. Da bi funkcija ispravno radila treba proslediti 2 ili 3 argumenta. Prvi argument je uvek isti i predstavlja putanju do test direktorijuma. Ukoliko je ova putanja relativna, apsolutna se pravi dodavanjem baznog direktorijuma na već opisan način. U slučaju kada imamo poziv ove funkcije sa dva argumenta, drugi argument predstavlja apsolutnu putanju do izvornog koda studentskog rešenja. Ovo je zgodno kada koristimo ocenjivač da pregledamo studentska rešenja koja se nalaze na računaru gde se nalazi i ocenjivač (na primer za pregledanje kolokvijuma). Kada imamo tri argumenta, drugi predstavlja ime datoteke pod kojim treba snimiti izvorni kod, koji je treći argument funkcije. Ovakav poziv funkcije je najčešći za klijentske aplikacije (na primer Veb aplikaciju). Povratnu vrednost ove funkcije predstavlja identifikator postavljenog studentskog rešenja, koji će biti

korišćen za dohvatanje informacija o ovom rešenju.

3. *Dohvatanje statusa izvršavanja studentskog rešenja* – kako je potrebno da sve radi asinhrono, klijentskim aplikacijama je pružen interfejs za proveru trenutnog stanja studentskog rešenja. Ove aplikacije to rade pozivom funkcije:

   ```
   extern "C" const char* get_task_status(const char* taskId);
   ```

   Na osnovu identifikatora postavljenog studentskog rešenja koji se prosleđuje ovoj funkciji, serverski deo ocenjivača pronalazi objekat koji odgovara ovom rešenju. Zatim, serverski deo ocenjivača šalje odgovarajuću statusnu nisku klijentskom delu ocenjivača. Ova niska je dobijena od pronađenog objekta i predstavlja povratnu vrednost funkcije *get_task_status()*. Ukoliko se traženi objekat ne nalazi u memoriji ocenjivača, studentsko rešenje je odavno obrađeno, a povratna vrednost funkcije je *nullptr*. Format statusne niske je takav da ona predstavlja JSON objekat sa sledećim poljima:

   1) *state* – predstavlja trenutno stanje u smislu provere studentskog rešenja na test primerima. Moguće vrednosti ovog polja su:

      a) *waiting* – rešenje još nije stiglo na red za ocenjivanje.

      b) *compiling* – izvorni kod se trenutno kompajlira.

      c) *running* – rešenje je uspešno kompajlirano i trenutno se pokreće za neki test primer. Poznati su rezultati izvršavanja za sve prethodne test primere.

      d) *compile_err* – rešenje nije ispravno postoji greška pri kompilaciji. Poznata je poruka o grešci i nalazi se u polju *message* JSON objekta.

      e) *finished* – rešenje je provereno na svim test primerima i rezultati su poznati.

      f) *internal_error* – došlo je do greške u samom ocenjivaču prilikom pokretanja studentskog rešenja.

      g) *malformed_request* – stanje do koga dolazi prilikom greške pri konstrukciji objekta “zadatak” koji odgovara studentskom rešenju. Najčešći razlozi za ovo stanje su pogrešne putanje do test direktorijuma ili datoteke sa izvornim kodom. Poznata je poruka o grešci.

   2) *message* – predstavlja poruku o grešci koja se dogodila. Tip greške se nalazi u polju *state*.

3) *test0, test1, ..., testN* – ova polja sadrže informacije o uspešnosti studentskog rešenja za odgovarajuće test primere. Polje *test0* je vezano za prvi test primer, *test1* za drugi itd. Vrednosti jednog ovakvog polja mogu biti:

   a) *time_limit* – označava da je rešenje probilo vremensku granicu za taj test primer.

   b) *memory_limit* – označava da je rešenje probilo memorijsku granicu za taj test primer.

   c) *runtime_error* – označava da je došlo do neke greške pri izvršavanju studentskog programa za taj test primer. Najčešće označava da je u pitanju greška u radu sa memorijom (eng. Segmentation fault).

   d) *incorrect_result* – označava da izlaz iz studentskog programa nije jednak očekivanom za taj test primer.

   e) *correct_result* – označava da program radi ispravno za taj test primer.

Dohvatanje statusne niske za studentsko rešenje koje je uveliko obrađeno je omogućeno sledećom funkcionalnošću.

4. *Dohvatanje kompletnih informacija o ocenjivanju obrađenog studentskog rešenja* – izlazi iz studentskih programa nisu dostupni u okviru statusne poruke. Ova odluka je donešena da se ne bi bez potrebe opterećivao mrežni transport. Dodatno, uključivanje ovih informacija u statusnu poruku bi bilo memorijski zahtevno pa bi štetilo performansama ocenjivača. Sa druge strane, izlazi iz studentskih programa nekada mogu biti od značaja, pa se samim tim čuvaju na disku u okviru ocenjivača. Za svaki obrađeni objekat "zadatak", postoji direktorijum na disku koji sadrži njegovu statusnu poruku u datoteci *status* i izlaze koje je program proizveo za test primere u datotekama *0.output*, *1.output*, ..., *N.output*. Klijentska aplikacija sadržaj ovog direktorijuma može da zatražiti pozivom funkcije:

   ```
   extern "C" const char* get_from_graveyard(const char* taskId);
   ```

   Serverski deo ocenjivača pronalazi lokaciju direktorijuma na osnovu identifikatora objekta "zadatak", koji ova funkcija prima, a koji klijentski deo ocenjivača prosleđuje serverskom. Zatim, serverski deo ocenjivača kompresuje traženi direktorijum i šalje ga klijentskom delu. Kompresovani direktorijum je ujedno i rezultat navedene funkcije. Klijentska aplikacija može ovaj kompresovani direktorijum da obrađuje direktno u memoriji bez pisanja na disk. Treba napomenuti da su na

ovom delu ocenjivača moguće određene optimizacije, kao što su asinhrono pisanje sadržaja direktorijuma u kome se nalaze informacije vezane za jedan objekat "zadatak", lenja kompresija ovih direktorijuma, eksplicitno navođenje za koje zadatak objekte ovi direktorijumi uopšte treba da postoje i druge.

5. *Pružanje metainformacija o ocenjivaču* – ocenjivač je sistem koji za ocenjivanje studentskih izvornih kodova koristi dodatke. Postavlja se pitanje koje dodatke ocenjivač ima? Da bi klijentskoj aplikaciji bila pružena mogućnost da proveri koji programski jeziki su podržana i koji tipova upoređivača rezultat postoje uvedena je funkcija:

   ```
   extern "C" const char* get_metadata();
   ```

   Ona vraća nisku čijim se parsiranjem može dobiti JSON objekat. Ovaj JSON objekat bi od polja imao 2 niza. U prvom nizu, polje *languages*, bi se nalazili podržani programski jezici, dok bi elementi drugog niza, polje *matchers*, bila imena upoređivača rezultata koji su dostupni ocenjivaču. Na primer, povratna vrednost funkcije *get_metadata()* bi mogla da bude niska:

   ```
   {
    "languages":["Java","C99","Python","C","C++","C++11"],
    "matchers":["StringMatcher","NumberMatcher"]
   }
   ```

U prethodnim pasusima ovog odeljka prikazane su funkcije interfejsa za programski jezik C, a često je referisana klijentska aplikacija. Treba razjasniti da to ne znači da klijentska aplikacija mora da bude pisana na programskom jeziku C. Ona može biti napisana na bilo kom programskom jeziku koji ocenjivač podržava (kao što je objašnjeno u 3.1.1). Međutim, ispod interfejsa u svim tim programskim jezicima pozivaju se baš ove C funkcije, pa su zato samo one predstavljene.

3.6.6 Centralni deo ocenjivača

Centralni deo ocenjivača podrazumeva deo koda koji implementira sledeće funkcionalnosti:

1. *Konfigurisanje ocenjivača* – ocenjivač je podesiv softver. Moguće je konfigurisati port na kome osluškuje serverski deo ocenjivača, veličinu međuprocesne memorije za čuvanje objekata "zadatak", putanju do direktorijuma u kome se nalaze dodaci, putanju do baznog direktorijuma gde se čuvaju test direktorijumi i drugo. Ovi parametri su zapisani u konfiguracionoj datoteci koja se nalazi na predefinisanoj putanji (/etc/grader/configuration.txt na Linux-u). Različiti delovi

ocenjivača koriste ove parametre. Da ne bi logika čitanja konfiguracione datoteke bila prisutna na više mesta u kodu, ona je enkapsulirana u klasu *grader::configuration*. Ova klasa čita konfiguracionu datoteku i smešta pročitano u mapu. Kako nam je od interesa da imamo samo jedan objekat koji apstrahuje konfiguraciju, klasa *grader::configuration* je implementirana kao Unikat.

U programskom jeziku C++ se savetuje izbegavanje makroa u korist *const* i *constexpr* deklaracija, odnosno definicija promenljivih. Međutim, pogledajmo kako makroi mogu da olakšaju rad sa konfiguracionim parametrima.

Nazivu svakog parametra odgovara vrednost koja ga opisuje. Na primer, parametar sa nazivom *port* može imati vrednost 7777. U konfiguracionoj datoteci se za konfigurisanje parametara koristi format *ime_parametra* = *vrednost_parametra*. Ovaj format je jednostavniji i od JSON i od XML formata, a dovoljno je izražajan za ono što ocenjivaču treba.

Vratimo se na korišćenje ovih parametara u kodu ocenjivača. Da bi se izbeglo ponovljeno navođenje istih konstatnih niski u kodu, podržani parametri su samo jednom definisani i to u klasi *grader::configuration*. Kako su promene naziva parametara u toku razvoja česte, dobra je praksa da ta imena postoje samo na jednom mestu. Zbog toga što redosled inicijalizacije statičkih promenljivih nije definisan u programskom jeziku C++, umesto statičkih konstantnih niski korišćene su statičke *inline* funkcije (Primer 16). Ova odluka je doneta zato što može doći do problema, kada jedna statička promenljiva koristi drugu, koja u tom trenutku nije još inicijalizovana.

```
inline static const std::string& server()
{
      static const std::string value = "server";
      return value;
}
```

Primer 16: Korišćenje statičke *inline* funkcije za definisanje parametra **server**.

C++ standard garantuje da će promenljiva *value* iz Primera 16 biti inicijalizovana na mestu gde se poziva funkcija *server()*. Za svaki parametar bi trebalo definisati ovakvu jednu funkciju. Dodatno, za dohvatanje imena parametara bi morala da se koristi konstrukcija *grader::configuration::ime_parametra()*, koja je sintaksno dugačka. Korišćenjem makroa iz Primera 17, definisanje novog parametra se svodi na navođenje makroa *COFIG_DEF(ime_parametra)* u telu

deklaracije klase *grader::configuration*. Pristupanje samom imenu parametra se vrši upotrebom *$(ime_parametra)*. Dohvatanje vrednosti za parametar, kojim se određuje adresa servera na kome se ocenjivač nalazi se može obaviti:

```
string serverName = gconf.get($(server));
```

pri čemu je *gconf* objekat klase *grader::configuration*, a metodom *get()* se dohvata vrednost za traženi konfiguracioni parametar.

```
#define CONFIG_DEF(name) inline static const std::string& name() \
                         { \
                            static const std::string value = #name; \
                            return value; \
                         }
#define $(key) grader::configuration::key()
```

Primer 17: Pomoćni makroi za rad sa konfiguracionim parametrima.

2. *Rukovanje dodacima* – kao i kod konfiguracije, dodaci su još jedan element programa koji se koristi na više mesta u kodu. Za upravljanje dodacima koristi se klasa *grader::plugin_manager*. Na osnovu putanje direktorijuma gde se nalaze dodaci, a koja se dobija od konfiguracije, instanca ove klase učitava dodatke u odgovarajući skup. Na osnovu imena programskog jezika, ili naziva upoređivača rezultata, iz skupa se dohvata željeni dodatak. Sami dodaci su enkapsulirani u strukturi *grader::plugin_info* (Primer 18).

```
struct plugin_info
{
  std::shared_ptr<dynamic::shared_lib> lib;
  std::string class_name;
  std::string lang_name;

  explicit plugin_info();
  explicit plugin_info(const std::string& lpath);
};
```

Primer 18: Deklaracija klase **plugin_info**.

Članica ove strukture *lib* predstavlja apstrakciju dinamičke biblioteke u kojoj je dodatak implementiran, *class_name* je ime klase (ocenjivači) ili funkcije (upoređivači), koja implementira funkcionalnost dodatka, a *lang_name* je članica koja predstavlja ime programskog jezika za koji se dodatak koristi (ima istu vrednost kao *class_name* kod upoređivača).

Dakle, u skupu koji je članica instance klase *grader::plugin_manager* se nalaze objekti tipa *grader::plugin_info*. Standardna biblioteka

programskog jezika C++ pruža dve implementacije skupa *std::set* i *std::unordered_set*. Za trenutnu implementaciju je izabrana druga opcija, međutim u C++-u je moguće dodatno apstrahovati korišćeni tip uporebom *using* definicija (Primer 19), čime se omogućava laka promena konkretne klase koja se koristi.

```
template <typename T>
using set_type = std::unordered_set<T>;
set_type<plugin_info> m_plugins;
```

Primer 19: Korišćenje *using* definicije za apstrakciju konkretnog tipa koji implementira skup. **m_plugins** je članica klase *grader::plugin_manager* u kojoj se čuvaju učitani dodaci.

Jednostavnom zamenom navedenog tipa u *using* definiciji, u Primeru 19, sa *std::unordered_set<T>* na *std::set<T>*, izvršila bi se promena tipa kojim je skup implementiran za celu klasu.

Pogledajmo sad još jedan koncept programskog jezika C++ – specijalizaciju šablona sa ciljem proširivanja standardne biblioteke. Razlika između *std::set* i *std::unordered_set* klasa je u tome što prva koristi drvoliku implementaciju i uređenje za upravljanje elementima, dok druga koristi heširanje. Da bi se klasa *grader::plugin_info* koristila kao argument šablona za *std::unordered_set*, ona mora da implementira heširanje i operator jednakosti. Najelegantniji pristup je da se iskoristi specijalizacija funktora standardne biblioteke, *std::hash<T>* i *std::equal<T>*, koje *std::unordered_set* podrazumevano koristi. Specijalizacija je data u Primer 20.

```
namespace std
{
  template <>
  struct hash<grader::plugin_info>
  {
    std::size_t operator()(const grader::plugin_info& plg) const
    {
      return std::hash<std::string>()(plg.lang_name);
    }
  };

  template <>
  struct equal_to<grader::plugin_info>
  {
    bool operator()(const grader::plugin_info& lhs,
                    const grader::plugin_info& rhs) const
    {
      return lhs.lang_name == rhs.lang_name;
```

```
    }
  };
}
```

Primer 20: Specijalizacija šablona *std::hash<T>* i *std::equal<T>* za klasu *grader::plugin_info*. Heširanje i poređenje na jednakost se rade na osnovu programskog jezika za koji se dodatak koristi, da bi se onemogućilo da postoje dva dodatka za isti programski jezik. Kako proširujemo standardnu biblioteku, celokupan kod mora da se nalazi u prostoru imena *std*.

Na samom kraju, prikažimo mehanizam koji je korišćen pri učitavanju dodataka. Dinamičke biblioteke koje implementiraju dodatke su pisane tako da sadrže metapodatke o sebi. Ovaj metapodatak je zapravo jedan globalni objekat strukture programskog jezika C – *plugin_metadata* (Primer 21).

```
extern "C"
{
 typedef struct
 {
   const char* class_name;
   const char* lang_name;
 } plugin_metadata;
}
```

Primer 21: C struktura kojojm su predstavljeni metapodaci za dodatke.

Svaka dinamička biblioteka koja predstavlja dodatak u svojoj implementacionoj datoteci (.cpp datoteka) ima liniju sličnu:

```
plugin_metadata metadata = { .class_name="javagrader",
                             .lang_name="Java" };
```

Na primer, navedeni kod definiše metapodatke za ocenjivač koji pregleda Java programe i koji je implementiran u klasi *javagrader*. Dakle, instanca klase *plugin_manager* učita dinamičku biblioteku dodatka u objekat klase *grader::shared_lib*. Zatim se dohvata pokazivač na globalnu *metadata* promenljivu. Vrednosti ove promenljive se dalje čitaju i upotrebljavaju za konstrukciju objekata tipa *grader::plugin_info*. Konstruktor klase *grader::plugin_info* zapravo prima putanju na kojoj se nalazi dinamička biblioteka, a opisana logika je implementirana u konstruktoru (Primer 22).

```
plugin_info::plugin_info(const string& lpath)
: lib(make_shared<dynamic::shared_lib>(lpath))
{
  plugin_metadata* metadata=static_cast<plugin_metadata*>(
                            lib->get_c_symbol("metadata"));
  if (!metadata || !metadata->class_name || !metadata->lang_name)
```

```
    THROW_SMART(plugin_exception, "No plugin metadata!");
  class_name = metadata->class_name;
  lang_name = metadata->lang_name;
}
```

Primer 22: Konstruktor klase *grader::plugin_info*. **THROW_SMART** je makro koji prilikom bacanja izuzetka osim navedene poruke zapisuje i liniju i datoteku u kojoj se izuzetak dogodio.

3. *Bazni deo ocenjivača za sve programske jezike* – ova potkomponenta ocenjivača je pažljivo dizajnirana sa ciljem da pisanje dodataka za konkretne programske jezike bude što jednostavnije. Implementirana je klasom *grader::base_grader*. Interfejs ove klase se može podeliti na dva dela. Prvi deo je javni interfejs klase koji sastoji se od samo tri funkcije date u Primeru 23.

```
static pointer get_grader(const fs::path& sourcePath,
                          const fs::path& testsPath,
                          task::pointer t);
virtual const char* name() const;
void grade(safe_process::user_id uid);
```

Primer 23: Javni interfejs klase *grader::base_grader*. **task::pointer** predstavlja pokazivački tip klase *grader::task*, koja je apstrakcija objekta "zadatak". **safe_process::user_id** je tip identifikatora sistemskog korisnika pod kojim se studentska rešenja pokreću.

Funkcija *get_grader()* je ustvari proizvodni metod koji vraća pokazivački tip klase *grader::base_grader* definisan kao:

```
using pointer = dynamic::shared_lib::object_pointer<base_grader>;
```

pri čemu je *shared_lib::object_pointer<T>* zapravo:

```
template <typename Derived>
using object_pointer = std::unique_ptr<Derived, object_dtor>;
```

Ova funkcija služi za kreiranje objekta klase, koja nasleđuje *grader::base_grader* i predstavlja ocenjivač za konkretan programski jezik. Za pravljenje ovog objekta se koristi baš proizvodni metod, zato što je nakon konstrukcije objekta potrebna njegova inicijalizacija, pa je on zgodan za tu operaciju. Naknadna inicijalizacija je potrebna, zbog toga što mehanizam koji implementira dodatke podržava samo podrazumevani konstruktor. Podaci potrebni za inicijalizaciju ove klase se dobijaju od pokazivača na objekat "zadatak" (*task::pointer* tip), koji predstavlja studentsko rešenje koje treba oceniti. Pomoću argumenata funkcije *get_grader()* određuje se za koji programski jezik treba instancirati ocenjivač. Drugi metod, naveden u Primeru 23, nema neku specijalnu svrhu, već je posledica klase *dynamic::object* koju

grader::base_grader nasleđuje. Metod *name()* vraća ime klase. Poslednje navedeni metod *grade()* služi za ocenjivanje objekta "zadatak". Kao argument prima sistemskog korisnika bez privilegija pod čijim će se nalogom studentsko rešenje pokretati prilikom testiranja.

Drugi deo interfejsa klase *grader::base_grader* je kvalifikovan kao *protected*, a ujedno je i virtuelan. Ovaj interfejs omogućava klasama koje će implementirati dodatke da na određene načine podese izvršavanje studentskih rešenja. Primer 24 sadrži četiri najbitnije funkcije ovog interfejsa.

```
virtual std::string compile() const;
virtual fs::path default_executable_path() const;
virtual std::vector<std::string> get_compile_cmd(
        const fs::path& sourcePath) const = 0;
virtual safe_process start_executable(const fs::path& executable,
        safe_process::user_id uid,
        std::size_t timeLimit,
        std::size_t memoryLimit,
        const std::vector<std::string>& args = safe_process::no_args,
        pipe_ostream* execStdin = nullptr,
        pipe_istream* execStdout = nullptr,
        pipe_istream* execStderr = nullptr,
        const std::string& jailDir = std::string{},
        const std::string& workDir = std::string{},
        const enviroment& env = enviroment{}) const;
```

Primer 24: *Protected* interfejs klase *grader::base_grader*.

Kako neki programski jezici ne zahtevaju kompilaciju (npr. Python), pružena je mogućnost da se podrazumevana implementacija kompajliranja programa preinači. Za Python, dovoljno je napisati prazan metod *compile()* u klasi koja nasleđuje *grader::base_grader*.

Takođe, programski jezici imaju različiti tip naziva izvršnih programa. Ilustracije radi, u programskim jezicima C i C++ podrazumevani naziv kompajliranog programa je *a.out*. U Javi, on odgovara nazivu izvorne datoteke gde se nalazi glavni metod (eng. main method), pa se na primer kompilacijom datoteke *Main.java* dobija izvršni program *Main.class*. Što se Python-a tiče ne postoji kompilacija, izvorna datoteka ima ulogu izvršne (direktno se interpretira). Da bi bazna klasa svih ocenjivača mogla ispravno da rukuje izvršnim programom za bilo koji jezik, potrebno je u potklasama implementirati logiku imenovanja programa. To se radi u metodu *default_executable_path()*. Dodatna napomena je da ovaj metod treba da vrati apsolutnu putanju, na kojoj se program nalazi nakon kompilacije izvorne datoteke. Za Python dovoljno je vratiti putanju izvorne datoteke.

Da bi bazna klasa uspešno kompajlirala izvorne datoteke za programske jezike za koje je to potrebno, uvodi se metod *get_compile_cmd()*. Na osnovu putanje izvorne datoteke, potklase su dužne da kroz povratnu vrednost funkcije proslede odgovarajuću komandu kojom će se program kompajlirati. Povratna vrednost ima takav očekivani format, da se u objektu *std::vector<std::string>* prvo navode opcije kompajliranja (eng. compile flags), a na kraju sam kompajler. U *compile()* funkciji bazna klasa svih ocenjivača će izdvojiti poslednji element ovog vektora, a njegov ostatak će direktno moći da koristi kao argumente prilikom pokretanja novog procesa za kompilaciju. Proces se pokreće pravljenjem objekta klase *grader::process* (treći konstruktor dat u Primeru 10).

Ostalo je još da se reši pitanje pokretanja studentskog programa za jedan test primer (sve test primere pokrećemo u petlji istim postupkom). C i C++ ne zahtevaju nikakav poseban mehanizam, već se programi direktno pokreću navođenjem njihove putanje. Što se tiče Jave, pokretanje izvršne datoteke *Main.class* se vrši pozivom *java Main.class* u komandnoj liniji (podrazumevano je da se Java interpreter nalazi na sistemskoj putanji). Za Python, važi da izvorna datoteka mora da ima privilegije za izvršavanje i da prva linija mora da sadrži komentar sa putanjom do interpretera. Drugi način za Python je direktno prosleđivanje ove datoteke interpreteru, na primer *python Main.py* (podrazumevano je da se Python interpreter nalazi na sistemskoj putanji). Kako različiti programski jezici zahtevaju drugačiji tretman prilikom pokretanja programa, uvedena je funkcija *start_executable()* da ovu operaciju apstrahuje. Klase koje nasleđuju *grader::base_grader* klasu treba da implementiraju logiku pokretanja u ovom metodu.

Dakle, bazna klasa ocenjivača radi većinu posla vezanog za pokretanje studentskog programa za test primere. Konkretne potklase služe samo da podese ovo izvršavanje na opisan način.

4. *Rukovanje međuprocesnom memorijom* – nakon kreiranja, objekti “zadatak” se dodaju u međuprocesni red. Dakle, imamo da se čitava struktura podataka (red), nalazi u međuprocesnoj memoriji. Dodatno, i objekti “zadatak” se moraju nalaziti u međuprocesnoj memoriji. Ono što možda nije odmah očigledno je, da i memorija alocirana za članice klase *grader::task* mora biti deo međuprocesne memorije. Najbolje je ilustrovati jednim primerom zašto je ovo baš ovako. Recimo da klasa *grader::task* ima samo jednog člana i neka je tip tog člana *std::vector<grader::test_result>*. Objekti ove klase se prave u demonskom procesu, a ocenjivanje se vrši u radnim procesima. Prilikom spomenutog ocenjivanja, radni proces će morati da modifikuje

navedenu članicu za *grader::task* objekat koji se ocenjuje. Ukoliko bi memorija, koju objekat *std::vector< grader::test_result >* interno koristi, bila alocirana u demonskom procesu, došlo bi do greške prilikom pokušaja njenog modifikovanja iz radnog procesa.

Korišćena biblioteka *Boost Interprocess* odlično apstrahuje ovu komplikovanu upotrebu međuprocesne memorije uz pomoć odgovarajućih *alokatora memorije* (eng. memory allocator). Alokatori su mehanizam koji standardna biblioteka koristi pri implementaciji struktura podataka kako bi alociranje memorije bilo podesivo. Kompletna deklaracija klase *std::vector<T>* koja koristi alokatore je data u Primeru 25.

```
template<class T, class Allocator = std::allocator<T>>
class vector;
```

Primer 25: Puna deklaracija klase *std::vector<T>*.

Boost Interprocess implementira više alokatora, koji različitim algoritmima alociraju međuprocesnu memoriju za potrebe struktura podataka kao što je vektor. Takođe, ova biblioteka omogućava pravljenje imenovanih objekata i primitivnih tipova u međuprocesnoj memoriji. To da su imenovani znači da im dva različita procesa koja dele istu međuprocesnu memoriju mogu pristupiti na osnovu imena.

Navedene funkcionalnosti *Boost Interprocess* biblioteke nam za sada omogućavaju da ispravno pravimo red sa objektima “zadatak”. Međutim, potrebno je te objekte “zadatak” uzimati iz reda tako da za jedan objekat može da konkuriše više procesa. Naravno, samo jedan proces će na kraju dobiti vlasništvo nad ovim objektom, koji će ujedno biti i odstranjem iz međuprocesnog reda. Za ovaj zahtev nam je potrebna određena sinhronizacija procesa. *Boost Interprocess* pruža mehanizme sinhronizacije u vidu muteksa i uslovnih promenljivih (eng. condition variable) koje funkcionišu na isti način kao kod sinhronizacije niti. Efekat koji je postignut je da je ispravna implementacija međuprocesnog reda ista kao i implementacija reda koji je bezbedan pri konkuretnom izvršavanju niti. Jedina razlika je u tipu muteksa i uslovne promenljive.

Kako je potrebna samo jedna, velika, međuprocesna memorija iz koje će se dalje alocirati po potrebi, cela logika pravljenja i uništavanja objekata je enkapsulirana u klasi *grader::shared_memory* koja je implementirana kao Unikat.

5. *Objekat “zadatak”* – funkcija objekta “zadatak” je da čuva podatke koji su potrebni za obradu studentskog rešenja koje mu odgovara. Dodatno,

ovaj objekat treba da čuva rezultate izvršavanja test primera i eventualne greške pri tom izvršavanju, kao što su greška pri kompilaciji, interna greška ocenjivača i sl. Potrebno je još i da generiše statusnu poruku na osnovu informacija koje poseduje, čim je ona zatražena. Informacije o tome dokle se stiglo sa obradom objekta "zadatak", se čuvaju u strukturi *grader::progress* (Primer 26).

```
struct progress
{
  state state_;
  shared_memory::shm_vector<test_result> tests;
  shared_memory::shm_string error_msg;
  progress();
  progress(const progress&) = delete;
  progress& operator=(const progress&) = delete;
  progress(progress&& oth);
  progress& operator=(progress&&) = delete;
};
```

Primer 26: Deklaracije strukture *grader::progress*.

Ova struktura je ugnježdena struktura (eng. nested structure) klase *grader::task*. Klasa *grader::task* sadrži članicu tipa *grader::progress*. Na taj način *grader::task* objekti čuvaju ažurne informacije o ocenjivanju. Članica strukture *grade::progress state_* označava u kom se stanju nalazi objekat "zadatak". Značenje mogućih vrednosti stanja je već objašnjeno prilikom opisa statusne poruke u JSON formatu. Takođe, *error_msg* sa logičke strane odgovara polju *message* JSON objekta koji nastaje parsiranjem statusne poruke. Na kraju, *grader::test_result* predstavlja rezultate testiranja koji su već opisani na istom mestu u tekstu, kao moguće vrednosti polja *test0*, *test1*, ..., *testN* JSON objekta.

Osim članice tipa *grader::progress* klasa *grader::task* sadrži još i:

1) Identifikator objekta "zadatak" koji je ujedno i ime ovog objekta konstruisanog u međuprocesnoj memoriji.
2) Putanju do izvorne datoteke koju treba oceniti.
3) Putanju do test direktorijuma.
4) Muteks za sinhronizaciju radnog procesa u kome se objekat "zadatak" ocenjuje i demonskog procesa u kome se potražuje statusna poruka.
5) Indikator da li je objekat "zadatak" spreman za uništenje. Ovaj indikator je tačan kada je objekat "zadatak" u nekom od završnih stanja. Završna stanja su stanja greške i stanje *finished*.

Identifikator objekta “zadatak” se koristi pri pravljenu putanja za direktorijume koji su potrebni za obradu studentskog rešenja. Dve nabrojane putanje koriste objekti bazne klase ocenjivača, kako bi izvorni kod bio kompajliran i test primeri izvršeni. Poslednje nabrojanu stavku, tj. indikator da li je objekat spreman za uništenje, koristi sakupljač zadatak objekata u svojoj implementaciji oslobađanja resursa.

Da bi enkapsulacija bila potpuna, status objekta klase *grader::task* ne može da se menja van koda ove klase. Umesto toga, uvedeni su događaji o kojima objekat bazne klase ocenjivača obaveštava odgovarajući objekat “zadatak”. Moguće vrednosti ovih događaja su date u Primeru 27.

```
enum class event: unsigned char
{
  popped,
  compiled,
  done_test,
  done_all,
  compilation_failed,
  grading_failed,
  construction_error
};
```

Primer 27: Mogući događaji o kojima objekti klase *grader::base_grader* obaveštavaju objekat klase *grader::task* koji ocenjuju.

Događaji iz navdenog primera označavaju redom:

1) Objekat “zadatak” je preuzet iz međuprocesnog reda.
2) Uspešno izvršena kompilacija.
3) Završen je određeni test primer.
4) Program je pokrenut za sve test primere.
5) Greška pri kompilaciji.
6) Interna greška zbog koje ocenjivanje nije završeno.
7) Greška pri konstrukciji objekta “zadatak”.

Poslednja od navedenih stavki se signalizira u okviru proizvodnog metoda klase *grader::task*. Za obaveštavanje o događijima se koriste specijalizacije šablonske metode *notify<event>* klase *grader::task* (Primer 28).

```
template<event>
void notify(const boost::any& data);
```

```
template<>
void notify<event::popped>(const boost::any& /**/);
template<>
void notify<event::compiled>(const boost::any& /**/);
template<>
void notify<event::done_test>(const boost::any& data);
template<>
void notify<event::done_all>(const boost::any& /**/);
template<>
void notify<event::compilation_failed>(const boost::any& data);
template<>
void notify<event::grading_failed>(const boost::any& data);
template<>
void notify<event::construction_error>(const boost::any& data);
```

Primer 28: Deklaracija specijalizacija šablona koji se koriste za obaveštavanje objekata klase *grader::task* o događajima.

Primetimo da u navedenom primeru postoje određene specijalizacije koje koriste argument koji primaju i postoje one koje ga ne koriste. Koncentrišimo se na specijalizacije funkcije *notify()* kojima se nešto prosleđuje. Tip ove promenljive u potpisu funkcije je *boost::any*. On omogućava prosleđivanje, odnosno čuvanje, bilo kog tipa koji je vrednosni tip. Na primer, objekat klase *boost::any* može čuvati *std::string* ili *std::vector* ili običan *int* . Ne može čuvati reference. Klasa *boost::any* je korišćena zato što specijalizacija po vrednosti *event::done_test* očekuje *grader::test_info*, a sve ostale specijalizacije očekuju *const char**. Međutim, ono što je zanimljivo je prikazivanje tehnike kojom *boost::any* omogućava čuvanje objekta proizvoljnog tipa.

Pomenuta napredna tehnika se zove *brisanje tipa* (eng. *type erasure*). Pri implementacije ove tehnike imamo 3 komponente. Prva je spoljašnji omotač koji čuva pokazivač na baznu klasu za sva takozvana *skladišta*. Ta bazna klasa je druga komponenta. Sama skladišta su treća komponenta. Počnimo od prve komponente. Njen konstruktor je šablonizovan, dakle može da primi različite tipove podataka. Osnovna ideja je da se za prosleđeni tip napravi skladište, koje može da čuva vrednost te promenljive. Zbog toga se skladišta implementiraju kao šablonizovana klasa. Sa druge strane, šablonizovanje cele spoljašnje komponente se izbegava time što sva skladišta nasleđuju istu klasu. Spoljašnja komponenta čuva napravljeno skladište u pokazivaču na baznu klasu svih skladišta, a taj pokazivač je član spoljašnje komponente. Da bi konstruktor kopije i operator dodele bili ispravno implementirani, potreban nam je još i jedinstven način da kopiramo vrednost koja se nalazi sačuvana u pokazivaču na baznu klasu. To se

postiže definisanjem čisto virtuelnog metoda, u pomenutoj baznoj klasi, čija je uloga dupliranje objekta. Kada šablonizovano skladište nasledi baznu klasu i implementira ovaj metod, on će automatski biti implementiran za sve tipove koji će konkretne instance skladišta čuvati. Spoljašnja komponenta će taj metod uspešno koristiti kroz pokazivač na baznu klasu. Minimalna implementacija opisanog koncepta je data u Primeru 29.

```
struct my_any
{
  my_any() = default;

  template <typename T>
  my_any(T const& v)
  : _storage(new storage<T>(v))
  { }

  my_any(my_any const& other)
  :_storage(other._storage ? std::move(other._storage->clone())
                           : nullptr)
  {}

  void swap(my_any& other)
  {
    _storage.swap(other._storage);
  }

  friend void swap(my_any& a, my_any& b)
  {
    a.swap(b);
  }

  my_any& operator=(my_any other)
  {
    swap(other);
    return *this;
  }
private:
  struct storage_base
  {
    virtual std::unique_ptr<storage_base> clone() = 0;
    virtual ~storage_base() = default;
  };

  template <typename T>
  struct storage : storage_base
```

```
  {
    T value;
    explicit storage(T const& v)
    : value(v)
    {}
    std::unique_ptr<storage_base> clone()
    {
      return std::unique_ptr<storage_base>(new
                                           storage<T>(value));
    }
  };

  std::unique_ptr<storage_base> _storage;

  template<typename T> friend T& any_cast(my_any&);
  template<typename T> friend T const& any_cast(my_any const&);
};

template <typename T>
T& any_cast(my_any& a)
{
  if (auto p=dynamic_cast<my_any::storage<T>*>(a._storage.get()))
    return p->value;
  else
    throw std::bad_cast();
}

template <typename T>
T const& any_cast(my_any const& a)
{
  if (auto p =
      dynamic_cast<my_any::storage<T>const*>(a._storage.get()))
    return p->value;
  else
    throw std::bad_cast();
}
```

Primer 29: Primer implementacije tehnike *brisanja tipa podataka*. Kompletna semantika pomeranja je zanemarena.

6. *Konfigurisanje izvršavanja test primera* – osim test primera, test direktorijum sadrži i konfiguracionu datoteku *test_configuration.txt*, kojom se regulišu različiti aspekti ocenjivanja studentskih rešenja. U ovoj datoteci se nalaze podešavanja vezana za:

   1) Programski jezik studentskog rešenja.

   2) Broj test primera u direktorijumu.

3) Uporeditač očekivanog i dobijenog rezultata.
4) Vremensko ograničenje u milisekundama.
5) Memorijsko ograničenje u bajtovima.
6) Tip izlaza koji se očekuje. Omogućeno da se za izlaz iz studentskog programa uzima ili standardni izlaz ili datoteka čija se putanja navodi u test konfiguraciji.

Očekuje se da svaki konfiguracioni parametar stoji u zasebnom redu. Imena podržanih programskih jezika odgovaraju onim imenima, koja se dobijaju kao deo JSON objekta koji je povratna vrednost već opisane funkcije *get_metadata()*. Isto važi i za nazive uporeditača. Ako studentski program treba da ispisuje izlaz na standardni izlaz, u konfiguraciji je potrebno staviti broj *0*. U suprotnom, tj. ako se očekuje izlaz u nekoj datoteci, potrebno je koristiti broj *2*. Dodatno, u ovom slučaju se, u nastavku linije, mora navesti i relativna putanja na kojoj će ocenjivač tražiti datoteku sa izlazom nakon završetka pokretanja programa za jedan test primer.

Struktura test diretorijuma je takva, da se u njemu, pored obavezne datoteke *test_configuration.txt*, može nalaziti još četiri vrste datoteka. U slučaju da je broj testova *N+1*, u direktorijumu se nalaze sledeće datoteke:

1) *0.input, 1.input, ..., N.input* – sadržaj datoteka sa ekstenzijom *.input* će biti prosleđivan studentskom programu na standardni ulaz. Sadržaj datoteke *i.input* će biti dostupan programu prilikom *i+1* pokretanja. Postojanje ovih datoteka nije obavezno. Njihovim izostavljanjem se dobija prazan standardni ulaz.
2) *0.args, 1.args, ..., N.args* – sadržaj datoteka sa ekstenzijom *.args* se prosleđuje studentskom programu u vidu argumenata komandne linije. Takođe, sadržaj datoteke *i.args* će biti dostupan programu u *i+1* pokretanju. Ove datoteke nisu obavezne.
3) *0.zip, 1.zip, ..., N.zip* – predstavljaju kompresovane hijerarhije direktorijuma sa pratećim datotekama koje se u njima nalaze. U krajnjem slučaju, *i.zip* može predstavljati samo jednu kompresovanu datoteku (na primer *ulaz.txt*). Sa druge strane moguće je da predstavlja i komplikovaniju strukturu, kao što je na primer direktorijum sa 10 potdirektorijuma, pri čemu se u svakom od njih nalazi po 5 datoteka.

   Studentski program se, pre narednog pokretanja, premešta u prazan direktorijum, čiji je vlasnik sistemski korisnik pod čijim nalogom će

program biti pokrenut. Pre *i+1* pokretanja, u ovaj direktorijum će biti otpakovana i *i.zip*. Dodatno, celokupan sadržaj otpakovane arhive će biti podešen da pripada navedenom sistemskom korisniku. Ovim se postiže velika fleksibilnost pri načinu zadavanja ulaza u program iz sistema datoteka. Arhive nisu obavezan deo test direktorijuma.

4) *0.output, 1.output, ..., N.output* – ove datoteke sadrže očekivane izlaze za test primere i one su obavezne. Gde god da se nalazi izlaz studentskog programa, on će biti upoređivan sa sadržajem ovih datoteka.

Kombinacijom standardnog ulaza, argumenata komandne linije i hijerarhijom datoteka koje će biti na raspolaganju studentskom programu, moguće je zadati probleme koji ispituju snalaženje programera početnika u samom sistemskom okruženju. Pored navedenog, ova funkcionalnost omogućava zadavanje programerskih zadataka u oviru predmeta koji su vezani za operativne sisteme. Ovo je značajan doprinos ocenjivača, koji ga izdvaja od sličnog, postojećeg softvera.

7. *Sakupljač zastarelih objekata "zadatak"* – nakon ocenjivanja studentskog rešenja, odgovarajući objekat "zadatak" se još neko vreme nalazi u međuprocesnoj memoriji. Osnovna ideja je da bude dostupan klijentskoj aplikaciji, koja bi trebala da proveri njegov status. Nakon isteka tog vremena, objekat klase *grader::task_garbage_collector* će ovaj objekat "zadatak" uništiti. Celokupno ponašanje je osmišljeno tako da međuprocesna memorija služi kao keš memorija za klijentske aplikacije. Logika kojom objekti klase *grader::task_garbage_collector* brišu objekte "zadatak" i oslobađaju međuprocesnu memoriju je već opisana pri kraju sekcije 2.4.3.

   Objekat klase *grader::task_garbage_collector* se pravi u okviru demonskog procesa. Prilikom konstrukcije ovog objekta biće pokrenut dodatni proces koji je zadužen za brisanje objekata "zadatak". Ovaj proces u petlji proverava da li je nekom objektu "zadatak" isteklo predviđeno vreme čuvanja od završetka ocenjivanja i ako jeste uništava ga.

   Bitno je još preciznije objasniti kako se objektu klase *grader::task_garbage_collector* prosleđuju objekti "zadatak" koje treba uništiti. Problem predstavlja međuprocesna komunikacija, zato što je potrebno da radni procesi obaveste proces, enkapsuliran u instancu ove klase, o objektu "zadatak" koji je ocenjen. Radni procesi nemaju jednostavan način da znaju koji proces je proces sakupljač. Trenutna

implementacija koristi komunikaciju preko demonskog procesa, koji je roditeljski proces i za radne procese i za proces sakupljač. Obaveštenje o tome da je završeno ocenjivanje objekta "zadatak" stiže od radnog do demonskog procesa, a zatim se u demonskom procesu na odgovarajući način modifikuje objekat klase *grader::task_garbage_collector*.

3.6.7 Upoređivači rezultata

Motivacija iza upoređivača rezultata je već objašnjena u prethodnim sekcijama. Fokusirajmo se sada na njihovu implementaciju. Tip kojim se predstavljaju upoređivači u programskom kodu je:

```
typedef int (*matcher_type)(const char*, const char*, const char*);
```

Dakle, u pitanju je pokazivač na funkciju, odnosno funkcija, koja prima tri argumenta tipa *const char**. Prvi argument predstavlja nisku koja sadrži očekivani izlaz iz programa, dok drugi argument je niska u kojoj se nalazi izlaz koji je program proizveo. Treći argument omogućava prosleđivanje dodatne niske. Nakon navođenja naziva upoređivača, a u istoj liniji datoteke *test_configuration.txt*, moguće je navesti ovu dodatnu nisku. Prilikom ocenjivanja svakog test primera ova niska će biti dostupna u trećem argumentu upoređivača. C++ kod jednostavnog upoređivača niski je dat u Primeru 30.

```
pair<const char*, const char*> get_range(const char*data)
{
  // Get size and predicate
  size_t dataLen = strlen(data);
  auto predicate = [](const char c) { return !isspace(c); };

  // Get pointer to first non-whitespace character
  const char*first = find_if(data, data + dataLen, predicate);

  // Get pointer to last non-whitespace character
  const char* last = &(*find_if(reverse_iterator<const char*>(data +
           dataLen), reverse_iterator<const char*>(data), predicate));
  // Return range [first, last)
  return make_pair(first, last + 1);
}

int StringMatcher(const char* expected, const char* real, const char* )
{
  // Get range for expected
  auto expectedRange = get_range(expected);
  auto realRange = get_range(real);
```

```
  // Check that we have ranges of same length
  if (expectedRange.second - expectedRange.first != realRange.second -
                                                    realRange.first)
    return false;

  // Return if ranges are equal
  return equal(expectedRange.first, expectedRange.second,
                                    realRange.first);
}
```

Primer 30: Programski kod upoređivača koji poredi dve niske na jednakost ignorišući pritom beline sa početka i kraja. **StringMatcher** je upoređivač, a **get_range** pomoćna funkcija.

Da bi bilo jasnije kako se treći argument upoređivača koristi pogledajmo implementaciju upoređivača realnih brojeva. Očigledno je da se poređenje mora vršiti u odnosu na određenu tačnost. Umesto propisivanje fiksne tačnosti za sve zadatke, tačnost se može prosleđivati kao treći argument upoređivača. Konkretno, pri implementaciji *NumberMatcher* upoređivača realnih brojeva, korišćena je konvencija da se prosleđuje dozvoljeno odstupanje od očekivanog rezultata. Dakle, u trećem redu datoteke *test_configuration.txt* bi mogla da stoji linija:

NumberMatcher 0.00005

što bi dovelo do prosleđivanja niske *"0.00005"* izabranom upoređivaču. Implementacija ovog upoređivača je data u Primeru 31.

```
int NumberMatcher(const char *expected, const char *real,
                  const char *arg)
{
  // Initialize epsilon
  static const double default_epsilon = .0001;
  double epsilon = arg ? stod(arg) : default_epsilon;

  // Initialize streams
  istringstream expectedStream{expected};
  istringstream realStream{real};

  // Compare doubles
  bool expectedHad = false, realHad = false;
  double ex, re;
  while ( (expectedHad = (expectedStream >> ex)) &&
          (realHad = (realStream >> re)) )
    if (fabs(ex - re) > epsilon)
      return false;
```

```
  // Return if outputs are equal
  return !expectedHad && realHad && !(realStream >> re);
}
```

Primer 31: Implementacija *NumberMatcher* upoređivača. Ova funkcija poredi dva niza od jednog ili više realnih brojeva sa prosleđenim odstupanjem. Ukoliko odstupanje nije navedeno u *test_configuration.txt* koristi se podrazumevana.

3.6.8 Interfejsi za pristup ocenjivaču iz različitih programskih jezika

Iako svi programski jezici navedeni na početku sekcije 3.1.1 imaju način da komuniciraju sa programskim jezikom C, ovi načini se mogu znatno razlikovati. Da bi se premostile ove razlike, za generisanje koda koji povezuje programske jezike sa C interfejsom korišćen je *SWIG*. Ovaj alat omogućava da se za sve programske jezike piše samo jedna datoteka – SWIG interfejs. SWIG pruža brojne opcije, ali za ocenjivač je bio dovoljan najjednostavniji slučaj koji će i biti prikazan. Datoteka sa opisom interfejsa, koja odgovara funkcijama koje su navedene u sekciji 3.6.5, je data u Primeru 32.

```
%module grader
%{
#include <stdbool.h>
#include <stddef.h>
extern bool save_tests(const char* relativePathToTestDir,
                       const char* zippedData, size_t len);
extern const char* submit_task(const char* pathToTestDir,
                               const char* sourceName,
                               const char* sourceContent);
extern const char* get_task_status(const char* taskId);
extern const char* get_metadata();
extern const char* get_from_graveyard(const char* taskId);
%}
extern bool save_tests(const char* relativePathToTestDir,
                       const char* zippedData, size_t len);
extern const char* submit_task(const char* pathToTestDir,
                               const char* sourceName,
                               const char* sourceContent);
extern const char* get_task_status(const char* taskId);
extern const char* get_metadata();
extern const char* get_from_graveyard(const char* taskId);
```

Primer 32: SWIG interfejs za izložene funkcije ocenjivača.

Deklaracijom *%module* navodi se ime odgovarajućeg modula. U nastavku datoteke, a između simbola %{ i }%, pišu se potrebne C deklaracije i uključuju se dodatna zaglavlja. U navedenom primeru, zaglavlje *stdbool.h* sadrži

definiciju bulovske promenljive programskog jezika C, pa je zato moralo da bude navedeno. Nakon bloka %{}% slede funkcije koje želimo da izložimo kroz SWIG interfejs. PHP ekstenziju od koda navedenog u Primeru 32 možemo dobiti pokretanjem sledećih naredbi u komandnoj liniji:

$ swig -php grader.i
$ g++ `php-config --includes` -fpic -c grader_wrap.c grader_interface.cpp
$ g++ -shared grader_wrap.o -o libgrader_php.so

Navođenjem *`php-config –includes`* prilikom kompajliranja datoteke *grader_wrap.c*, koju SWIG generiše na osnovu interfejs datoteke, obezbeđuju se zaglavlja potrebna za pravljenje PHP ekstenzije. Datoteka *grader_interface.cpp* sadrži implementaciju izloženih funkcija, pa se i ona mora uključiti u proces kompilacije.

Jedan od proizvoda opisanog procesa je i PHP datoteka koju treba uključiti u PHP kodu koji koristi ocenjivač. U njoj će se nalaziti kod odgovarajuće PHP klase (Primer 33). Datoteku će SWIG automatski nazvati *grader.php*.

```
abstract class grader {
   static function save_tests($relativePathToTestDir,$zippedData,$len){
      return save_tests($relativePathToTestDir,$zippedData,$len);
   }
   static function submit_task($pathToTestDir,$sourceName,
                               $sourceContent) {
      return submit_task($pathToTestDir,$sourceName,$sourceContent);
   }
   static function get_task_status($taskId) {
      return get_task_status($taskId);
   }
   static function get_metadata() {
      return get_metadata();
   }
   static function get_from_graveyard($taskId) {
      return get_from_graveyard($taskId);
   }
}
```

Primer 33: PHP klasa koja se dobije od SWIG interfejsa datog u Primeru 32.

Da bi opisana PHP ekstenzija uspešno radila, potrebno je kopirati nastalu dinamičku biblioteku *libgrader_php.so* u predefinisani direktorijum u kome se nalaze sve PHP ekstenzije. Na samom kraju, potrebno je još i modifikovanje *php.ini* konfiguracione datoteke, u kojoj se dodavanjem linije *extension=libgrader_php.so* odobrava upotreba opisane ekstenzije.

3.6.9 Pozadinski proces – demon

Kako je logika ove komponente detaljno opisana u odeljku 3.4.3, sama implementacija neće biti navođena. Ipak, trebalo bi spomenuti neke specifičnosti vezane za sam koncept demonskog procesa. Da bi proces bio demonski trebalo bi da ispunjava sledeća dva kriterijuma:

1. *To je dugotrajan proces* – ovakvim procesima se najčešće opisuje posao koji traje dok god operativni sistem radi. Dobar primer upotrebe demonskih procesa predstavljaju različiti Linux servisi kao što su *cron*, *sshd* i *inetd*. Prvi omogućava izvršavanje komande u tačno određenom trenutku, drugi omogućava logovanje i rad sa udaljene mašine, dok je treći zadužen za osluškivanje konekcija na TCP/IP portovima.
2. *Izvršava se u pozadini bez terminala koji ga kontroliše* – posledica je da kernel ne može automatski da generiše signale, koje terminali koriste prilikom kontrole izvršavanja, za demonski proces. Jedan od ovih signala je i SIGHUP, koji se zbog toga interno koristi za ponovnu inicijalizaciju demonskog procesa.

Postoje određene tehničke poteškoće prilikom odvajanja inicijalnog procesa od terminala koji ga kontroliše [30], ali na operativnom sistemu Linux postoji funkcija koja ih rešava za nas:

```
int daemon (int nochdir, int noclose);
```

Ako se ovoj funkciji proslede dve nule kao argumenti, onda će se radni direktorijum promeniti na *"/"*, a standardni izlazi i ulazi na *"/dev/null"*. Za bilo koje druge argumente ništa se ne menja u odnosu na inicijalni proces.

4 Diskusija

Predstavljeni ocenjivač je složen softver. Zbog toga bi u početku bilo dobro pratiti rad ocenjivača sa ciljem da se otklone potencijalni propusti. Osnovna ideja je da se odobri niži nivo logovanja poruka i da se nakon nekog vremena rada prikupe informacije o problemima. Takođe, očigledno poboljšanje predstavlja proširivanje softvera novim ocenjivačima za programske jezike koji do tog trenutka nisu bili podržani. Isto važi i za upoređivače. Sa druge strane, bilo bi posebno zanimljivo implementirati automatsko generisanje test primera na osnovu tačnog rešenja. Ova funkcionalnost bi omogućila alternativni način zadavanja test primera – zadavanje tačnim rešenjem.

Unapređeni softver bi dalje trebalo testirati sa stanovišta performansi. Ovaj test bi poslužio da se pronađu uska grla u kodu i ako je moguće otklone. Dodatno, od interesa bi bilo memorijsko testiranje kojim bi se odredilo koliko

prosečnih objekata “zadatak” staje u blok međuprocesne memorije veličine 1GB.

Sledeći korak nakon testiranja bi mogao da bude vezan za horizontalno skaliranje sistema. Trenutni ocenjivač ne može da radi na više mašina, čime je horizontalno skaliranje onemogućeno. U slučaju obimnije upotrebe ovog softvera predložena funkcionalnost predstavlja logičan sledeći korak. Jedna od ideja je implementacija balansera koji bi raspoređivao zahteve klijentske aplikacije na više računara, pri čemu na svakom od njih bio pokrenut ocenjivač.

5 Zaključak

U radu je opisan softver koji omogućava automatsko pregledanje studentskih programa. Osnovna ideja je da se ovaj softver koristi kao deo nekog sistema, koji bi služio kao interfejs prema ocenjivaču i olakšavao njegovu upotrebu. Osim masovne upotrebe ocenjivača, predviđena je i individualna upotreba, koja bi se realizovala pristupanjem ocenjivaču, pokrenutom na ličnom računaru, iz nekog od podržanih skript jezika kao što je Python.

Pored samog softvera, prikazane su prednosti i poneka mana modernog C++-a. Tokom opisa realizacije komponenti koje čine ocenjivač, određene tehnike programskog jezika C++ su stavljene u odgovarajući kontekst, kojim se prikazuje kako se njihovom upotrebom olakšava implementacija željenih funkcionalnosti.

Reference

[1] Dokumentacija za *Java Native Interface* (http://docs.oracle.com/javase/7/docs/technotes/guides/jni/)
[2] Python dokumentacija za *ctypes* modul (https://docs.python.org/3/library/ctypes.html)
[3] Upotreba *DllImport* direktive C# jezika na Windows i Unix platformi (https://msdn.microsoft.com/en-us/library/aa984739(v=vs.71).aspx) (http://www.mono-project.com/docs/advanced/pinvoke/)
[4] Golemon, Sara. *Extending and Embedding PHP*. Carmel: Sams Publishing, 2006.
[5] Dokumentacija za *NaCL* projekat (https://developer.chrome.com/native-client)
[6] Dokumentacija za *PNaCL* projekat (https://www.chromium.org/nativeclient/pnacl/introduction-to-portable-native-client)
[7] Dokumentacija za *Web Service Description Language* (http://www.w3.org/TR/wsdl)
[8] Java 8 dokumentacija (http://docs.oracle.com/javase/8/)
[9] "*ISO/IEC 14882:2014 -- Information technology -- Programming languages -- C++*". ISO. 14 January 2014.
[10] Kerrisk, Michael. “Process Creation”. *The Linux Programming Interface*. San Francisco: No Starch Press, 2010. 513 - 530.
[11] Cygwin implementacija *fork()* funkcije (https://www.cygwin.com/faq.html#faq.api.fork)
[12] Apache server dokumentacija (http://httpd.apache.org/docs/)
[13] Internet Information Server zvanična stranica (https://www.iis.net/home)
[14] Zvanična Microsoft dokumentacija *CreateProcessAsUser* funkcije (https://msdn.microsoft.com/en-us/library/windows/desktop/ms682429(v=vs.85).aspx)
[15] Richter, Jeffrey i Nasarre, Christophe. “Kernel objects”. *Windows® via C/C++, Fifth Edition*. Microsoft Press, 2007. 33-66.
[16] Microsoft dokumentacija za ACL (https://msdn.microsoft.com/en-us/library/windows/desktop/aa374872(v=vs.85).aspx)
[17] Microsoft dokumentacija nekih poznatih *SID*-ova (https://msdn.microsoft.com/en-us/library/windows/desktop/aa379649(v=vs.85).aspx)
[18] Zvanična dokumentacija *Chromium sandbox*-a

(https://www.chromium.org/developers/design-documents/sandbox)
[19] FreeBSD dokumentacija *jail()* koncepta
(https://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/jails.html)
[20] Dokumentacija biblioteke Boost za verziju 1.54
(http://www.boost.org/doc/libs/1_54_0/)
[21] Dokumentacija SWIG 2.0 alata
(http://www.swig.org/Doc2.0/SWIGDocumentation.html)
[22] Dokumentacija CMake 3.2 alata
(http://www.cmake.org/cmake/help/v3.2/)
[23] CLion
(https://www.jetbrains.com/clion/)
[24] Zvanična stranica gcc 4.9 kompajlera
(https://gcc.gnu.org/gcc-4.9/changes.html)
[25] Kerrisk, Michael. "Daemons". *The Linux Programming Interface*. San Francisco: No Starch Press, 2010. 767-782.
[26] Kerrisk, Michael. "Process Creation And Program Execution In More Detail". *The Linux Programming Interface*. San Francisco: No Starch Press, 2010. 598-609.
[27] Todd L. Veldhuizen. *C++ templates are Turing complete*. Indiana University Computer Science department.
[28] Vandervoode, David, Josuttis M., Nicolai. *C++ Templates: The Complete Guide.* Boston: Addison-Wesley Professional, 2002.
[29] Alexandrescu , Andrei. *Modern C++ Design.* Boston: Addison-Wesley Professional, 2001.
[30] Kerrisk, Michael. "Daemons". *The Linux Programming Interface*. San Francisco: No Starch Press, 2010. 775-781.
[31] Stroustrup, Bjarne. "Memory and Resources". *The C++ programming language 4th edition*. Boston: Addison-Wesley Professional, 2013.
[32] Richter, Jeffrey i Nasarre, Christophe. "Processes". *Windows® via C/C++, Fifth Edition*. Microsoft Press, 2007. 89-104.
[33] Kerrisk, Michael. "Process resources". *The Linux Programming Interface*. San Francisco: No Starch Press, 2010. 753-765.
[34] Dokumentacija */proc* sistema pseudo datoteka:
(http://linux.die.net/man/5/proc)
[35] Kerrisk, Michael. "Timers and sleeping". *The Linux Programming Interface*. San Francisco: No Starch Press, 2010. 479-513.
[36] Dokumentacija *seccomp* biblioteke:
(https://github.com/seccomp/libseccomp)
[37] Dokumentacija Boost ASIO 1.54 biblioteke:
(http://www.boost.org/doc/libs/1_54_0/doc/html/boost_asio.html)
[38] D. Schmidt et al. "Event handling patterns". *Pattern Oriented Software Architecture Volume 2.* New Jersey: Wiley, 2000. 181-218.
[39] George E. Forsythe and Niklaus Wirth. *Automatic Grading Programs.*

CACM 8(5), 1965. 275-278.
[40] Veb lokacija sistema *SPOJ*:
(http://www.spoj.com/)
[41] Veb lokacija sistema *TopCoder*:
(https://www.topcoder.com/)
[42] Veb lokacija sistema *WebCAT*:
(http://web-cat.cs.vt.edu/Web-CAT/WebObjects/Web-CAT.woa)
[43] Veb lokacija sistema *Marmoset*:
(http://marmoset.cs.umd.edu/index.shtml)