TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Tarkvarateaduse instituut

Joosep Franz Moorits Alviste 143042IAPB

PROGRAMMEERIMISÜLESANNETE AUTOMAATTESTIMISSÜSTEEMI LIIDESTUS MOODLE'I KESKKONNAGA JA MUGAV KASUTAJALIIDES TUDENGITE HINDAMISEKS

Bakalaureusetöö

Juhendaja: Ago Luberg
MSc

Tallinn 2017

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autor: Joosep Franz Moorits Alviste

22.05.2017

Annotatsioon

Antud töö raames luuakse ülesande moodul Moodle'i õppekeskkonda. Moodul võimaldab automaattestidega kontrollitava programmeerimisülesande loomist ning suhtleb välise testeriga, mis saadab moodulile ülesande eest saadud tulemusi. Lisaks sisaldab moodul sellega loodavate ülesannete kontrollimiseks ja hindamiseks õppejõule mugavat kasutajaliidest. Luuakse ka evituse (*deploy*) protsess, et ained.ttu.ee Moodle'i õppekeskkonda installida või uuendada. Töö on jaotatud analüüsi, arendusprotsessi, metoodika ning lahenduse osaks. Analüüsis täpsustatakse loodava mooduli funktsionaalsust, kirjeldatakse andmebaasi skeemi ning pannakse kirja tähtsamad kasutusjuhud. Arendusprotsessi ja metoodika peatükis tegeletakse ained.ttu.ee keskkonna evituse protsessiga ning arendusprotsessi korraldamisega. Lahenduse peatükis kirjeldatakse analüüsis spetsifitseeritud mooduli loomist.

Töö eesmärkideks oli luua testeriga suhtlev moodul Moodle'i keskkonda, automatiseerida evituse protsessi ning luua kasutajasõbralik hindamiskeskkond. Selleks loodi kasutades PHP ja JavaScripti programmeerimiskeeli ning Laraveli ja Vue raamistikke Moodle'isse tegevuse moodul. Moodul sisaldab hindamiseks kasutajaliidest. Lisaks kasutati Jenkinsit ning Dockerit, et keskkonna evitust automatiseerida. Töö raames lahendati kõik eesmärkides püsitatud ülesanded.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 33 leheküljel, 5 peatükki, 10 joonist, 0 tabelit.

Abstract

Interfacing automated testing system with Moodle and convenient user interface for grading students

The purpose of this thesis is to create an activity module for Moodle learning management system. The module enables the creation of a programming task, which is validated using automated tests. There is an external tester, which communicates with the module and sends it results for the tasks. In addition, the module contains a convenient user interface for teachers to check and grade submissions. Also, a deploy process is created for installing or updating the ained.ttu.ee Moodle learning environment. The thesis contains analysis, development process, methods and the created solution. The analysis section specifies the functionality of the module, shows the database schema and most important use cases. The development process and methods section contains the deployment process of ained.ttu.ee and organizing the development process. The solution chapter details the creation of the module.

The aim of this thesis was to create a module that communicates with the tester, automate the deployment process of ained.ttu.ee environment and to create a user-friendly grading interface. To achieve these goals, the author used PHP and JavaScript languages, as well as Laravel and Vue frameworks to create a Moodle activity module. The module contains the grading interface. Moreover, Jenkins and Docker were used to automate the deployment process of the Moodle environment. The goals set in the thesis were realized fully.

The thesis is in Estonian and contains 33 pages of text, 5 chapters, 10 figures, 0 tables.

Lühendite ja mõistete sõnastik

AJAX	Asynchronous JavaScript and XML, kasutatakse kliendi ja serveri vahel asünkroonseks suhtluseks
API	Application Programming Interface, programmi liides
ES2015	Ecma International firma poolt loodud JavaScripti standard 2015 aastal
Git	Versiooniohjesüsteem
JSON	JavaScript Object Notation, tekstiformaat, mida kasutatakse andmete edastuseks
Moodle	Õpihaldusesüsteem
Mähis, *wrapper*	Õhuke koodikiht, mis tõlgib mõne liidese paremini kasutatavaks liideseks
ORM	Object-relational mapping, virtuaalne objektide andmebaas mis seob andmebaasi ja programmi
Repositoorium	Giti salv
SSL	Secure Sockets Layer, kasutatakse krüpteeritud suhtluse jaoks kliendi ja serveri vahel
WebSocket	Ühendus browseri ja serveri vahel. Kasutatakse andmete vahetamiseks

Sisukord

Jooniste loetelu

1 Sissejuhatus

Programmeerimisülesannete lahenduste käsitsi kontrollimine on suure tudengite arvu korral ajamahukas. Kontrollimiseks saab kasutada automaatteste, kuid siis peab need iga tudengi kohta eraldi käima panema ja tulemust kontrollima. Parem oleks siduda automaattestid kasutusel oleva Moodle'i õppekeskkonnaga nii, et kui tudeng laeb lahenduse Giti üles, siis testitakse tema lahendust ja salvestatakse testide tulemused automaatselt Moodle'isse. Seejärel saab õppejõud Moodle'is hindeid hallata ja muuta, kui selleks vajadus on.

Antud töö eesmärgiks on luua Moodle'i moodul, millega saab tudengi esitatud programmeerimisülesande lahenduse testitulemused automaatselt õppekeskkonda integreerida. Testimine toimub väljaspool loodavat moodulit. Kui tudeng laeb oma lahenduse Giti, siis laeb selle alla tester ning testib seejärel koodi automaattestidega. Testide tulemus saadetakse loodavasse moodulisse Moodle'is. Õppejõule luuakse mugav kasutajaliides, kus saab tudengi esitusi, esitatud faile, tulemusi ja testeri poolt genereeritud tagasisidet näha. Lisaks saab tudengi esituse hindeid muuta.

Moodul peab olema dokumenteeritud ning automaattestidega kaetud. Valminud moodul paketeeritakse tervikliku moodulina, mida saab kasutada ka mõne teise Moodle'i installatsiooni peal. Lisaks moodulile on vaja realiseerida automaatne mooduli uuendamine ilma kasutuses oleva lehe tööd häirimata.

2 Analüüs

Antud peatükis täpsustatakse loodava mooduli funktsionaalsust. Lisaks luuakse andmebaasi skeem ning pannakse kirja tähtsamad kasutusjuhud. Analüüsi teoreetilisi tulemusi realiseeritakse töö lahenduse peatükis.

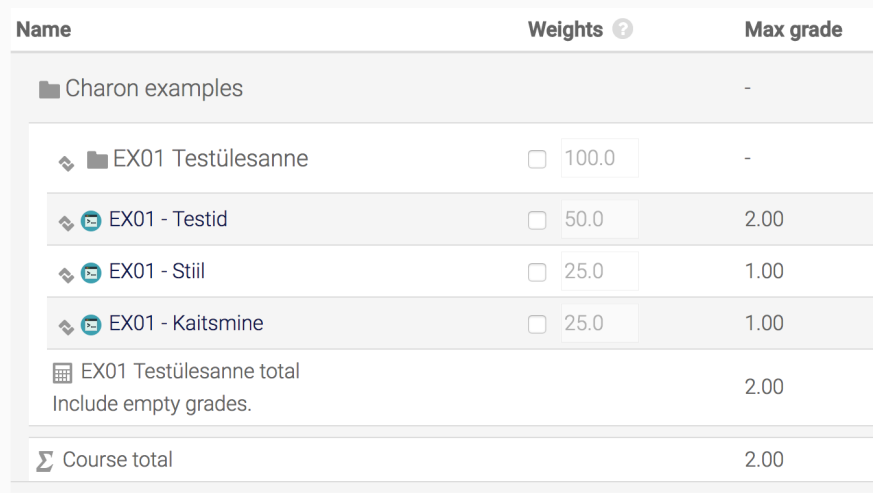
2.1 Tulemuse spetsifikatsioon

Loodav moodul (edaspidi Charon) võimaldab luua Moodle'isse ülesande, mille tulemused saadakse väliselt teenuselt. Ülesande loomisel saab täpsemalt määrata, milliseid tulemusi oodatakse. Üks tulemus võib sisaldada mitut erinevat hinnet. Näiteks võib väline teenus saata tudengi kohta 3 hinnet. Ülesande loomisel saab määrata, milliste nimedega need 3 hinnet Moodle'i hinderaamatusse pannakse, samuti saab määrata loodavate hinnete struktuuri (Moodle'i hinde kategooria) ning arvutusvalemit (Moodle'i hinderaamatu funktsionaalsus).

Charonit saab kasutada programmeerimisülesannete loomiseks. Antud moodul on TTÜs juba aktiivselt kasutuses programmeerimisainete ülesannete testimisel. Välise teenusena on TTÜs loodud automaattester, mis Charonile tulemusi saadab. Tester pole osa sellest lõputööst. Programmeerimisülesande lahenduse hindamisel on võimalik arvestada mitmeid töö aspekte. Enamasti on TTÜ programmeerimisainetes kasutusel automaattestid ning ülesande lahenduse kaitsmine, kuid kasutatakse ka stiilikontrolli. Automaattestidega revideeritakse tudengi programmi tööd ning nõuetele vastamist. Stiilikontrolliga kindlustatakse esitatud koodi ühtne ning mõistlik koodistiil. Lahenduse kaitsmisel kontrollib õppejõud, kas tudeng on oma programmi ise kirjutanud ning, et kas ta oskab programmeerides tehtud otsuseid põhjendada. Kodutöö eest, millel on nõutud kaitsmine, peaks tulemuse saama alles siis, kui kaitsmine on läbitud. Charon võimaldab ülesande loomisel märkida ära milliseid külgi antud ülesande kontrollimisel hinnatakse. Hindamiseks saab kasutada automaatteste, stiilikontrolli ja manuaalset hinnet (näiteks kaitsmine). Nii automaattestide kui ka stiilikontrolli hinne tuleb testerist, mis pole antud mooduliga otseselt seotud. Tester käivitab pärast lahenduse esitamist automaattestid ning saadab tulemuse Charonisse. Lisaks kontrollib tester koodistiili.

Manuaalne hinne on esitustes alati 0, seda peab õppejõud käsitsi muutma. Manuaalset hinnet saab kasutada näiteks boonuspunktide jaoks. Lisaks lisapunktidele saab seda kasutada kaitsmise kirja panemiseks. Kui tudeng on oma töö edukalt õppejõule kaitsnud, saab õppejõud selle märkida käsitsi kaitsmise hinde sisse kandmisega. Igat hinde tüüpi saab olla mitu, näiteks kaks manuaalset hinnet või mitmed automaattestimise hinded. Ülesandega on seotud valem, mille järgi lõpphinne arvutatakse. Valem koosneb ülesande hinnete tulemustega arvutamisest. Näiteks võib testide tulemusele liita stiilikontrolli tulemuse.

Programmeerimisülesande näitena võib tuua ülesannet, kus tudengid peavad looma teatud funktsionaalsuse. Ülesande õige funktsionaalsuse eest on võimalik saada kaks punkti. Lisaks nõutakse koodi vastavust stiilinõuetele. Ülesande tulemus läheb arvesse vaid siis, kui koodi stiil on korrektne. Selleks, et seda saavutada, saab korrektse stiili eest 1 punkti ning ülesande punktid korrutatakse sellega läbi. Iga ülesanne tuleb õppejõule kaitsta. Selleks kasutatakse omakorda hinnet "kaitsmine", kuhu õppejõud kirjutab käsitsi punktid (tavaliselt 1p). See korrutatakse samuti kogu eelneva tulemusega läbi. Seega ülesande eest kokku saadav punktisumma oleks: funktsionaalsuse punktid (automaattestid) * stiilipunktid * kaitsmine. Ehk, kui tudeng saab pooltest testidest läbi, läbib stiilikontrolli ning kaitsmise, siis saab ta kokkuvõttes ühe punkti (1 * 1 * 1). Moodle'isse tekiks siis kolm hinnet, kuid ülesannet oleks üks. Joonis 1 kujutab kirjeldatud näiteülesannet õppekeskkonna hinderaamatus. Ülesanne koosneb kolmest hindest: testid, stiil ja kaitsmine. Selliselt hinde tüübi Moodle'i hindega sidumise jaoks kasutatakse antud töös terminit hinde vastavus (*grademap*).

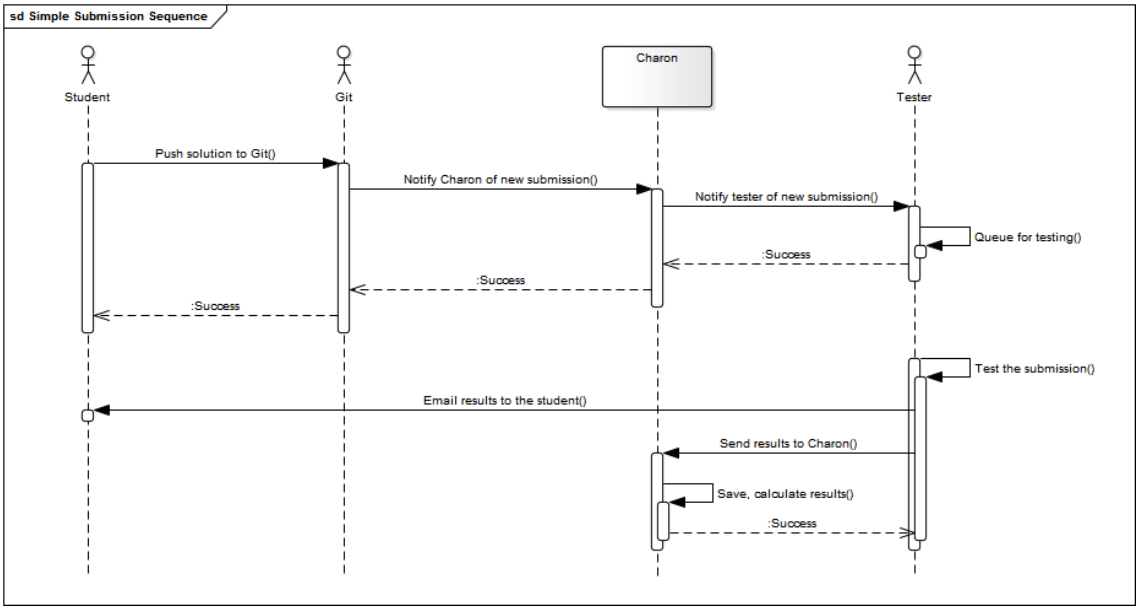


Joonis 1. Näiteülesanne õppekeskkonna hinderaamatus.

Ülesandele on võimalik lisada tähtaegu pärast mida saab tudeng esituse eest vähem punkte. Kui pärast tähtaega lahendus esitada, siis korrutatakse tulemus läbi tähtaja juures märgitud protsendiga. Tähtaeg mõjub vaid testide hindele, see tähendab, et stiili eest saadud tulemused on alati samad. Lisaks sellele saab ülesannet luues täpsustada valemit, mille järgi ülesande tulemus kujunema peaks. Lisatav valem on tavaline Moodle'i valem, mis tähendab, et selles võivad ka teised aine ülesanded sisalduda. Tähtaegade ja valemi näiteks võib võtta eelmises lõigus kirjeldatud näiteülesande ja sellele lisada tähtaja koefitsiendiga 80%. Lisatakse hinde arvutamise valem, kus kirjeldatakse, et ülesande kogutulemus arvutatakse hinnete EX01 – Testid, EX01 – Stiil ja EX01 – Kaitsmine korrutisena. Ehk kui tudeng saab kõikides hindamiseaspektides maksimumpunktid, siis saab ta ülesande eest kaks punkti. Kuna tester manuaalse hinde, EX01 – Kaitsmine, tulemust ei saada (see on null), siis peab tudeng hinde saamiseks õppejõule oma töö kaitsma. Edukalt kaitstud töö korral sisestab õppejõud tudengile EX01 – Kaitsmine hinde eest ühe punkti ning tudeng saab terve ülesande eest tulemuse kätte. Pärast tähtaega esitades saab tudeng lahenduse eest vaid 80% sellest, mis ta õigel ajal esitades oleks saanud. Pooli teste läbides saaks tudengi lahendus vaid 0,8 punkti. Kui tähtaega poleks punktide arvutamisel kasutatud, oleks tudeng saanud ühe punkti.

Programmeerimisülesande esitamine toimub kasutades Giti versioonikontrollisüsteemi. Igal tudengil on oma salv, kuhu ta oma lahendused ülesse laeb. Iga Giti kehtestuse (*commit*) ja *push*'iga saadetakse teavitus Charonile, mis edastab selle testerile. Tester on

Charonist eraldi olev rakendus, mille arendamisega autor ei tegelenud. Tester käivitab tudengi koodi peal automaattestid ning saadab testide tulemused Charonile. Seejärel registreeritakse tulemused automaatselt Moodle'isse. Hindamise etapis rakendatakse tulemustele tähtaegade koefitsiente. Lisaks protsendilisele tulemusele, kui mitu testi tudengi kood edukalt läbis, saadab tester moodulile testide väljundi (*stdout*), veateated (*stderr*), esitatud koodi ning tudengile e-postile saadetud tagasiside. Joonis 2 kujutab lihtsustatult lahenduse esitamist.



Joonis 2. Lihtsustatud lahenduse esitamine.

Lisaks ülesande loomisele ja esituste esitamisele luuakse õppejõule hindamise jaoks mugav ja kiire keskkond. Hindamiskeskkonnas saab otsida iga ülesande ja tudengi kohta esitusi. Esituste kohta näeb hinnete tulemusi, Giti kehtestuse infot, testeri saadetud väljundeid ning esitatud koodi. Hindamiskeskkonnas näeb ka tudengi terve kursuse hinnete ülevaadet. Õppejõud saab iga tudengi ja ülesande kohta lisada kommentaari, mida näidatakse teistele õppejõududele. Selliselt saab kirja panna spikerdamise ja koodi kopeerimise infot nii, et ka teised õppejõud seda näeksid.

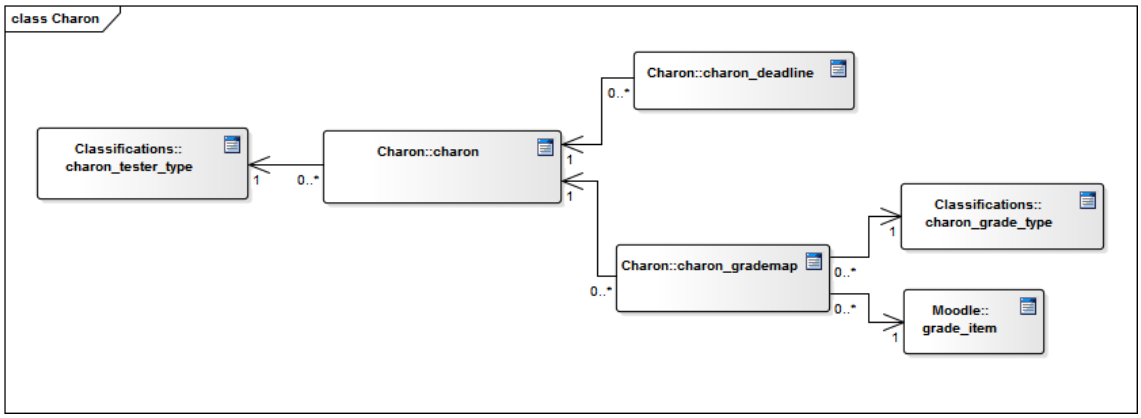
Moodul sisaldab tudengi ülesande vaadet, milles näeb tudeng ülesande kirjeldust, hinde osi (*grademap*) ning tähtaegu. Lisaks näeb tudeng antud ülesande kohta kõiki oma esitusi koos punktidega ning esitatud faile.

Kuna ülesannetel on palju konfiguratsiooni, otsustas autor võtta kasutusele eelkonfigureeritud ülesannete mallid (*preset*), milles saab kirjeldada enamus ülesande

välju. Kui ülesande loomisel valida mõni mallidest, siis täidetakse automaatselt võimalikult paljud väljad ära. Selliselt saab õppejõud keskenduda ülesande sisule ning vähem mõelda sellele, kuidas punkte jagada ja mis valemiga ülesande punktid kokku arvutatakse. Malle saab iga kursuse kohta lisada, kuid on mõned eelkonfigureeritud mallid. Mallideks võib olla üks ülesandetüüp, näiteks tunniülesanded. Kui sellises mallis kirjeldada ära, et tunniülesannetel on automaattestid ja stiilikontroll ning ülesanne koondhinne kujuneb nende tulemuste korrutamisel, siis seda malli kasutades ülesande loomisel sätitakse need väljad automaatselt paika.

2.2 Andmebaas

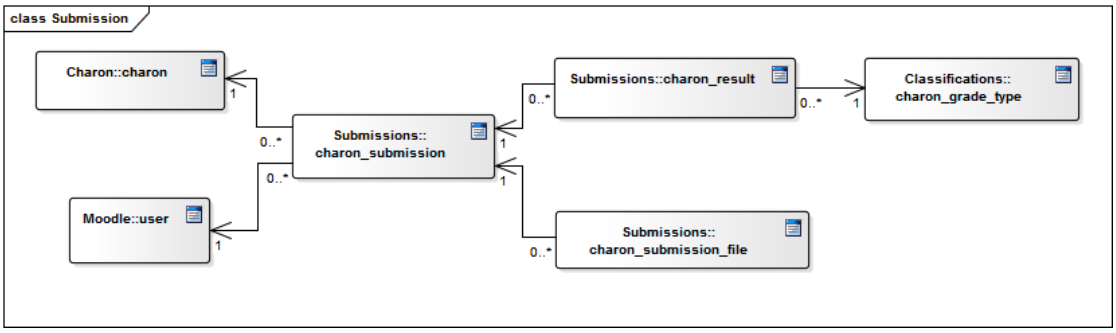
Järgnevalt on toodud välja lihtsustatud andmebaasi arhitektuur klassidiagrammide näol. Joonistes ei ole välja toodud tabelite tulpade kirjeldusi, kuid on näidatud olemite vahelisi seoseid. Tähtsaimad andmebaasi objektid on seotud ülesandega (Joonis 3) ning tudengi esitusega (Joonis 4).



Joonis 3. Ülesande isendi lihtsustatud klassidiagramm.

Ülesannete hoidmiseks kasutatakse andmebaasis tabelit `charon`, kuna see on mooduli nimi. Moodle nõuab, et moodulite primaarsete tabelite nimed vastaksid mooduli nimele. Iga ülesandega seostub testeri tüüp, tabelis `charon_tester_type`, mis näitab millist testeri tüüpi automaattestide käivitamiseks kasutama peab. Tavaliselt on testeri tüüp sama, mis programmeerimiskeel, mida ülesandes kasutatakse. Näiteks on testeriteks Java ja Pythoni testerid. Charoni põhitabelis hoitakse lisaks testeri tüübile veel ülesande pealkirja, kirjeldust, Giti projekti kausta nime (kaust, kuhu tudengid oma lahenduse panevad) ja Moodle'i hinnete kategooria viidet.

Iga Charoniga saab olla seotud mitu hinde vastavust (*grademap*) ja tähtaega. Hinde vastavus kirjeldab suhet Moodle'i hinde (*grade item*) ja hinde tüübi koodi vahel. Hinde tüüp näitab kas tegemist on automaattestide tulemuste eest saadud hindega, koodi stiili hindega või manuaalse hindega. Hinde tüüpide koodid on järgnevad: 1 – 100 on automaattestide hinded, 101 – 1000 on stiilikontrolli hinded ja 1000+ on manuaalsed hinded. Moodle *grade item*'is sisaldub informatsioon selle kohta, kui palju hinde eest võimalik punkte saada on. Tähtajad pakuvad võimalust esituste eest punktide saamist ajaga piirata. See tähendab, et kui pärast sisestatud tähtaega töö esitada, siis saab selle eest vähem punkte. Tabelis `charon_deadline` hoitakse tähtaja aega kuupäeva ja kellajana ning protsenti, mis näitab mitme protsendiga pärast tähtaega esitatud esituse punktide arv läbi korrutatakse.



Joonis 4. Esituse isendi lihtsustatud klassidiagramm.

Tudengi esitusi hoitakse tabelis `charon_submission`, mis sisaldab endas viidet ülesandele, ehk Charonile, ja õpilasele. Lisaks viidetele hoitakse esituse tabelis Giti kehtestuse (*commit*) andmeid, nende hulgas on näiteks kehtestuse räsi ja aeg. Veel hoitakse `submission` tabelis infot selle kohta, kas esitus on kinnitatud või mitte, millal esitus loodi, mis on testeri väljund selle esituse kohta (nii vead kui ka tavaline väljund) ning tudengile saadetud tagasiside. Esituse kinnitamine tähendab, et õppejõud on selle hinde kinnitanud. Kinnitatud hinnet ei uuendata automaatselt isegi kui tudeng saab järgmise esitusega parema tulemuse. Esituse tabelis ei hoita punktilisi tulemusi vaid seda tehakse `result` (tulemus) tabelis, kus on iga hinde vastavuse tulemus eraldi välja toodud. Tulemusi võib iga esituse kohta olla mitu. Iga tulemus vastab läbi hinde tüübi olemasolevale hinde vastavusele. Näiteks võib võtta eelnevalt kasutatud näiteülesannet, mis koosneb automaattestidest, stiilikontrollist ja kaitsmisest. Kui tudeng esitab oma lahenduse, tekib andmebaasi `submission` (esitus) tabelisse üks sissekanne ja `result` (tulemus) tabelisse kolm sissekannet. Iga hinde tüübi kohta on üks tulemus, ehk tudeng

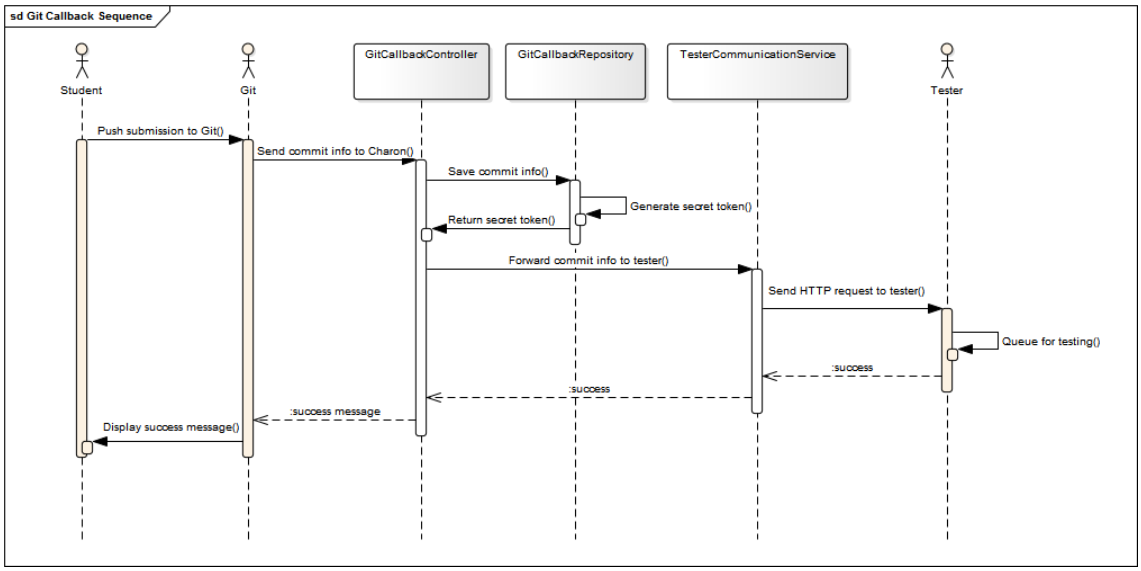
saab tulemuse testide, stiili ja kaitsmise eest. Tulemuse tabelis hoitakse testerilt saadetud testide tulemuse protsenti, kalkuleeritud tulemust (pärast tähtaegade sisse arvestamist) ning testeri väljundeid antud hinde testide kohta.

2.3 Kasutusjuhud

Järgnevalt tuuakse näited mõnest kasutusjuhust ning nende kohta käivatest jadadiagrammidest. Näited tuuakse tudengi lahenduse esitamise protsessi kohta.

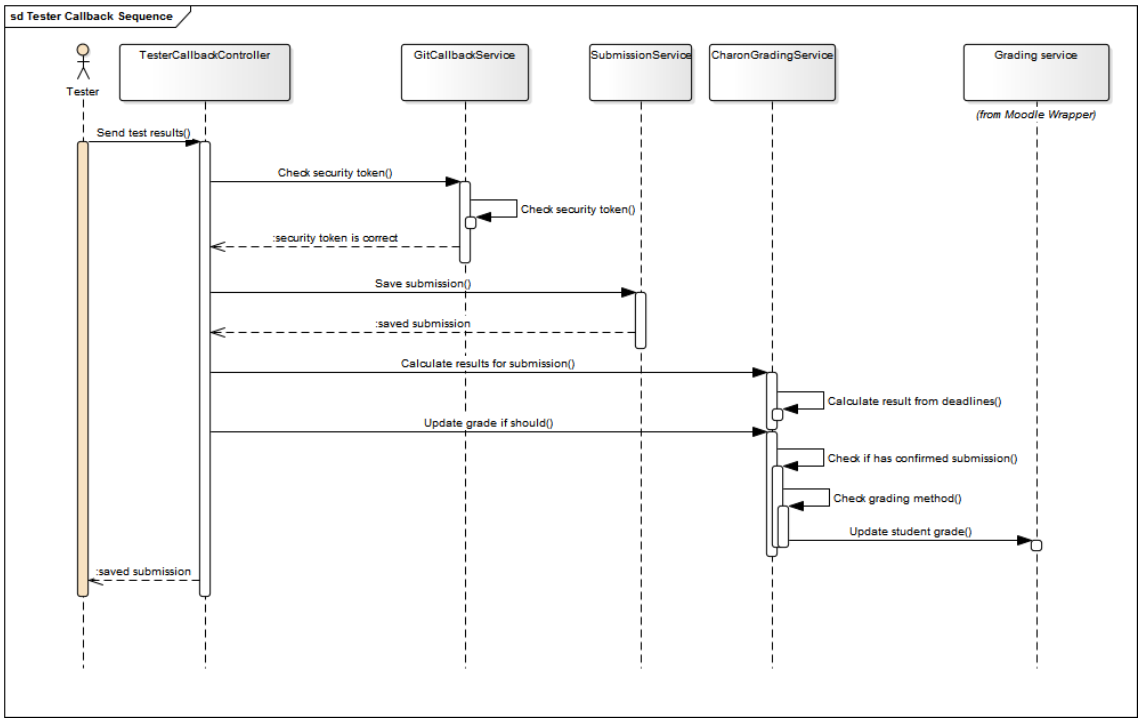
Lahenduse esitamise protsess koosneb kahest osast. Esiteks laeb tudeng oma lahenduse Giti üles. See info saadetakse Charonile ning seejärel saadab Charon selle edasi testerile. Tester paneb esituse testimisejärjekorda (Joonis 5). Järgmisena testib tester lahenduse ära ning saadab tulemused Charonile (Joonis 6). Allpool on toodud tudengi esituse esitamise ja testerilt tulemuse saamise jadadiagrammid.

Joonis 5 kujutab tudengi esituse esitamist Giti kehtestuse ja *push*'imise näol. Giti haagiga (*hook*) on korraldatud Charonile kehtestuse info saatmine. Saadetakse HTTP GET päring Giti repositooriumi aadressiga ning lisatakse õpilase Uni-ID. Seejärel genereeritakse testeriga turvaliseks suhtluseks salajane kood ning salvestatakse *commit*'i andmed andmebaasi. Salvestatud andmed saadetakse testerile ning tester paneb esituse testimisejärjekorda. Viimaseks tagastatakse õnnestumise teade Giti ning Git näitab seda teadet tudengile. Peale kirjeldatud sündmuste käiku alustab tester ülesande testimist ning saadab testi tulemused Charonile. Testimine toimub asünkroonselt eraldi protsessina.



Joonis 5. Esituse esitamise jadadiagramm.

Joonis 6 kujutab testide tulemuste Charonile saatmist testeri poolt. Testeri poolt saadetud päringus peab olema lisatud eelnevalt genereeritud salajane kood. Kontrollitakse koodi olemasolu ja õigsust. Seejärel salvestatakse esitus ning arvutatakse esituse hinnete eest saadud lõpptulemus. Iga tulemus sisaldab endas testide läbimise protsenti, mis korrutatakse hinde maksimumpunktidega läbi. Lõpphinde arvutamises võetakse lisaks arvesse tähtajad. Viimaseks kontrollitakse, kas tudengi hinnet Moodle'is peaks uuendama või mitte. See sõltub sellest, kas tudengil on juba kinnitatud esitus ning hindamise meetodist. Kui hindamise meetod on *prefer last*, ehk uusima esituse eelistamine, siis uuendatakse Moodle'i hinnet alati (kui pole juba kinnitatud hinnet). Kui aga hindamise meetodiks on *prefer best*, ehk parima esituse eelistamine, siis kontrollitakse, kas uus esitus on parem kui eelmised. Kui esitus on parem, siis uuendatakse hinnet Moodle'is.



Joonis 6. Testide tulemuste vastuvõtmise jadadiagramm.

3 Arendusprotsess ja metoodika

Antud töös on kasutusel mitmeid laialt levinud arendusprotsessi võtted ning metoodikaid. Järgnevalt tutvustatakse kasutatud metoodikaid ning nende kasutamise põhjusi.

3.1 Arendusmetoodika

Antud töös on kasutatud mitmeid arendusmetoodikaid. Mooduli arenduses on tähtis versiooniohjesüsteemi korrastatus ning kerge kehtestuste vahel liikumine. Hästi korraldatud Giti ajalugu lubab tekkinud probleeme ja defekte leida ning nende tekkepõhjuseid kergelt tuvastada.

Evituse (*deploy*) protsess on tihti aeganõudev ja veaohtlik. Samuti on ebamugav kahte erinevat keskkonda kasutada, näiteks kasutuses olev *live* keskkond ja testimiseks mõeldud *develop* keskkond. Keskkondadel on tihti erinevad kasutusse viimise etapid. Näiteks erineb testkeskkonna andmebaasi konfiguratsioon kasutuses olevast keskkonnast. Lisaks tuleb enne kasutusse viimist programmi testida ning teha kindlaks, et kõik töötab. Kõike seda meeles pidada ja käsitsi evitusprotsessi läbi viia on tülikas ja tihti võib midagi valesti minna. Evituse probleeme aitab vähendada pidevkooste (*continuous integration*). Pidevkooste on tarkvara evituse viis, kus evitus toimub pidevalt ja automaatselt ning enne igat koostet testitakse programmi, et võimalikult kiiresti defektidest teadlik olla. Kasutades pidevkoostet saab ühe nupuvajutusega uut programmi versiooni kasutusse viia [7].

Giti harude manageerimiseks kasutatakse Git flow metoodikat. Evituse (*deploy*) jaoks kasutatakse pidevkooste programmi Jenkins ning virtuaalset konteinerit Docker. Järgnevalt on täpsemalt Git flowd ja evitust kirjeldatud.

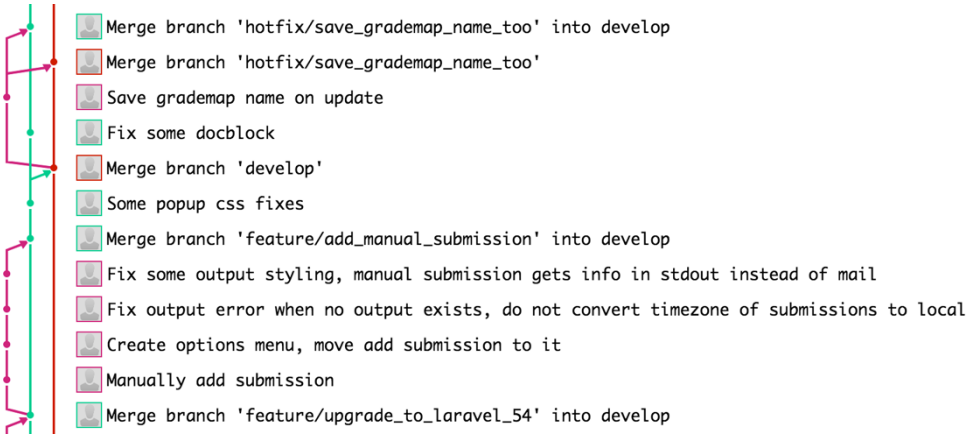
3.1.1 Git flow metoodika

Git flow on Giti repositooriumi manageerimise viis. Enamasti kirjeldab Git flow seda, kuidas harusid (*branch*) kasutama ja nimetama peaks. Git flowi metoodikas on kasutusel kaks põhilist haru: `master` ja `develop`. `Master` harus hoitakse kasutuses olevat lõpptoodet ja `develop` harus arendusel olevat seisu. Lisaks nendele põhiharudele on kasutuses mitu erineva eesmärgiga harude liiki. Igal haru tüübil on oma eesmärk ja

nende harust pärinemisega ning mestimisega on seotud kindlad reeglid. Harude tüüpideks on funktsionaalsuse (*feature*), kiirparanduse (*hotfix*) ja väljalaske (*release*) harud. Sama liigiga harude nimedele liidetakse ühine eesliide. Funktsionaalsuse harud on mõeldud täiendava funktsionaalsuse arendamiseks, mis tulevases versioonis lisatakse, ning neile liidetakse eesliide `feature/`. Funktsionaalsuse harud saavad oma algseisu `develop` harust ning funktsionaalsuse valmides mestitakse see tagasi `develop` harru. Tähtsate muudatuste tegemiseks ja bugide parandamiseks on kasutusel kiirparanduse harud, mille eesliide on `hotfix/`. Juhul kui kasutuses olevas versioonis on mõni defekt, luuakse selle parandamiseks `master` harust uus kiirparanduse haru. Paranduse valmides mestitakse loodud haru tagasi `master` ja `develop` harru Rakenduse versioonide halduseks on kasutusel väljalaskeharud ning nende eesliide on `release/`. Väljalaske harudes valmistatakse tarkvara uue versiooni väljalaskeks ette. Parandatakse defekte, lisatakse versiooninumbrid ning tehakse muud enne väljalaset vajalikud tegevused. Väljalaske haru luuakse `develop` harust ning mestitakse väljalaske valmides nii `develop` kui ka `master` harru [2].

Antud töös on kasutuses veidi modifitseeritud Git flow metoodika. Kasutuses on `master` ja `develop` harud ning lisaks funktsionaalsuse ja kiirparanduste harud. Arendajal on üldiselt kasutusel `develop` haru ning kui on vaja arendada uut funktsionaalsust, siis tehakse uus haru nimega `feature/uus_funktsionaalsus_x`. Kogu funktsionaalsuse arendamine toimub selles harus. Kui funktsionaalsus on valminud, mestitakse (*merge*) selle jaoks loodud haru `develop` harusse. Piisava funktsionaalsuse valmides ja väljalaske jaoks mestitakse `develop` haru `master`'isse. Seejärel saab `master` haru rakenduse seisu kasutusse anda. Juhul, kui kasutuses oleval versioonil tekib mõni probleem või defekt, luuakse selle parandamiseks `master` harust kiirparanduse haru nimega `hotfix/defekt_mida_lahendatakse`. Kui defekt saab parandatud, siis mestitakse kiirparanduse haru nii `develop` kui ka `master` harru.

Erinevat tüüpi harude vahel töö jagamine teeb kergemaks versioonihalduse ajaloo jälgimise. Defektide tekkekoha leidmiseks peab `develop` harus vaid osafunktsionaalsuse mestimise kehtestusi (*commit*) kontrollima. Leides defektse funktsionaalsuse haru, saab seda täpsemalt edasi uurida. Joonis 7 kujutab loodava mooduli Giti kehtestuste logi, kus on erinevad harud ning nende mestimine vastavalt Git flow metoodikale. Ajatelg on joonisel alt üles.



Joonis 7. Charoni Giti kehtestuste logi.

3.1.2 Docker

Docker on minimaalne virtuaalmasin, mis hoiab endas operatsioonisüsteemi. Selles virtuaalmasinas käivitatakse programm ning peale programmi töö lõppu peatatakse Docker.

Üheks Dockeri suurimaks eeliseks on ühene evituse (*deploy*) protsess igas masinas. On vaid üks fail, kus kirjeldatakse, mis programmid ja failid Dockeri pilti (*image*) lähevad. Selle faili järgi saab genereerida Dockeri pilti, see serverisse üles laadida ning seejärel käivitada. Veel üheks eeliseks on turvalisus. Kuna terve Dockeri konteineri sisu on virtuaalmasin, siis on see eraldatud serveri operatsioonisüsteemist ja failidest. Kui Dockeri konteineri sees mõni turvaauk on, siis on kahju vaid virtuaalmasinas, mitte serveris. Samuti on evituse protsess kindlustatud serveri katki minemise vastu. Kui serveriga midagi juhtub, siis võib Dockeri lihtsalt teises masinas tööle panna ja see töötab nagu varem [5].

TTÜs kasutusel olev Moodle'i keskkond ained.ttu.ee veebileht töötab kahes keskkonnas: *production*, ehk kasutusel olev seis, ja *develop*, ehk arenduses olev seis. Igapäevaselt kasutavad tudengid *production* keskkonda ning Charoni Gitis hoitakse selle keskkonna seisu `master` branchis. Testimiseks ja katsetamiseks on *develop* keskkond. Seal hoitakse `develop` haru seisu. Tänu Dockerile on mõlema keskkonna installeerimine väga sarnane ning erineb vaid konfiguratsioonifailide poolest.

Antud töös kasutatud evituse protsessis on Dockerit kasutatud terve ained.ttu.ee veebilehe ning serveri hoidmiseks. Dockeris on Moodle ning kõik kasutusel olevad moodulid, kaasa arvatud Charon. Dockeri pilt viiakse *production* või *develop* serverisse

ning käivitatakse seal Apache veebiserver[1], mis serveerib ained.ttu.ee keskkonda. Kuna Dockeri failid taastatakse algseisu peale Apache serveri töötamise lõppemist, siis tuleb vajalikud failid ja kaustad vastendada (*map*) failidega serveris. Nendeks failideks on näiteks kasutajate poolt üles laetud failid `moodledata` kaustas ja süsteemi poolt genereeritud Apache ja Charoni logifailid. Dockeris asuvad vastendatud failid pannakse vastavusse serveris olevate kaustade ja failidega. See võimaldab näiteks logifaile vaadata otse serveris.

Kasutuses olev Dockeri pilt põhineb uusimal Ubuntu[2] versioonil, milleks antud töö kirjutamise ajal on 17.04. Dockeri pildi koostamisel installeeritakse vajalikud paketid, nende hulgas on näiteks PHP[3] ja Apache. Seejärel lisatakse eelnevalt alla laetud Moodle'i kaust ning moodulite kaustad. Need kaustad sisaldavad mooduleid, mis ained.ttu.ee keskkonda installeeritakse. SSL konfigureerimiseks kopeeritakse olemasolevad Apache konfiguratsioonifailid Dockeri pildis vastavatesse kaustadesse. Sarnaselt Apache konfigureerimisele sätestatakse Postfix[4], mis tegeleb e-posti saatmisega ning tehakse PHP täpsem seadistus. Kui Dockeri pilt serveris tööle panna, siis tuleb sinna lisada mõned konfiguratsioonifailid ning käivitada installi või uuendamise Bashi skript. See, milliseid konfiguratsioonifaile Dockerile lisatakse, otsustab Jenkinsi skript vastavalt sellele, mis keskkonda, *environment*'i, ehitatakse. Nii *develop* kui ka *production* keskkonnal on erinevad konfiguratsioonifailid. Konfiguratsioonifailid asuvad alati serveris, mitte versioonikontrollis (Gitis), et tundlik info ei oleks kõikidele arendajatele kättesaadav.

Installeerimise skript koosneb mitmest etapist. Esimesena loetakse serveri konfiguratsioonifailist parameetrid, mida kasutades Moodle installeeritakse. Parameetriteks on näiteks andmebaasi aadress ja selle kasutaja andmed ning ained.ttu.ee veebilehe administraatori kasutaja nimi ja parool. Need näitajad on *develop* ja *production* keskkonnas erinevad, kuna parameetrid loetakse erinevatest failidest. Järgmiseks käivitatakse Postfixi `cron` teenus ning kopeeritakse SSL võtmed Apachele

[1] https://httpd.apache.org/

[2] https://www.ubuntu.com/

[3] https://secure.php.net/

[4] http://www.postfix.org/

vajalikku kausta. Seejärel installeeritakse Moodle, mille käigus täidetakse vajadusel andmebaas Moodle'i esialgsete andmetega ning käivitatakse Moodle'i `cron` teenus. Kasutusel olevate moodulite installeerimine on järgmine samm, selle käigus käivitatakse mõned Charoni mooduli nõutavad käsud. Lisatakse eesti keele pakett, mis laetakse Moodle'i lehelt `wget` programmi kasutades alla. Samuti toimub siin Moodle'i täpsem konfigureerimine. Näiteks muudetakse lehe väljanägemist muutev teema (*theme*), mõnesid administraatori seadeid ning lehe välimuse seadeid. Lõpuks tühjendatakse salvestatud vahemälu ning käivitatakse Apache server, mis serveerib nüüd ained.ttu.ee keskkonda.

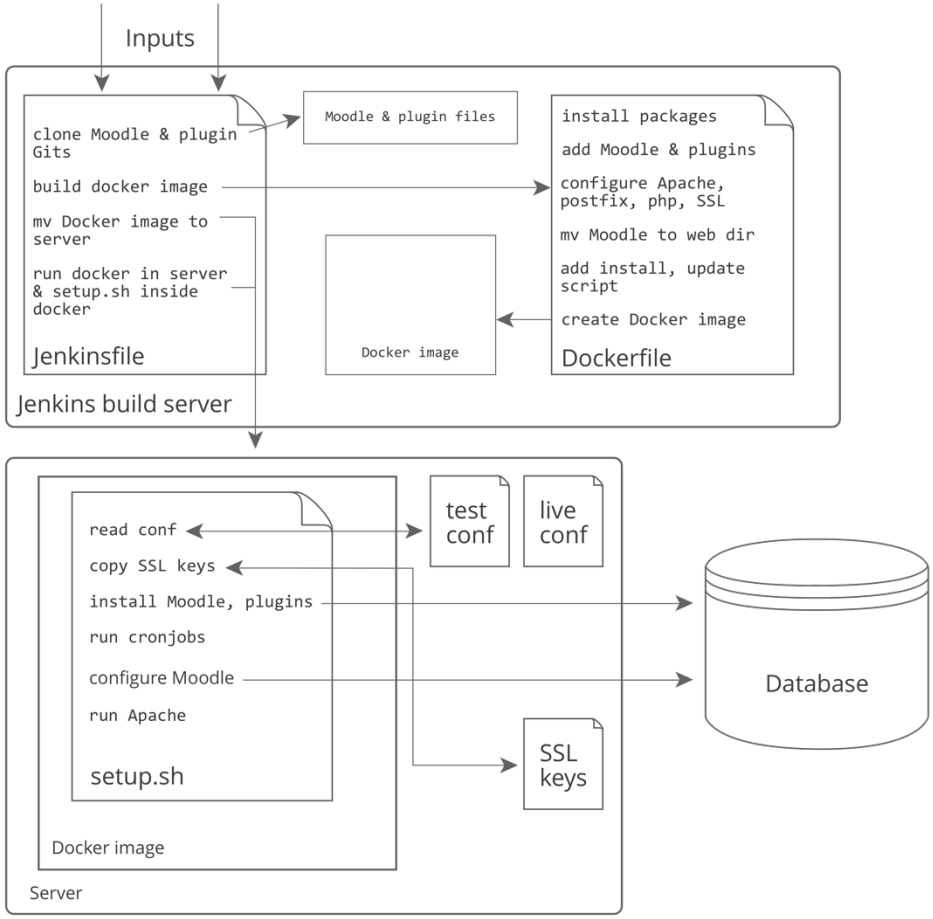
Uuendamise skript uuendab vaid mooduleid, kuna Moodle'it ei uuendata väga tihti. Vanad moodulite failid kirjutatakse uute poolt üle ning Charoni installeerimiseks käivitatakse sarnased käsud nendele, mis installi skriptis olid. Uuendamise skripti töötades lülitatakse ained.ttu.ee leht automaatselt hooldusrežiimile ning valmides lülitatakse see tagasi tavalisse režiimi.

3.1.3 Jenkins

Jenkins on tarkvara, mida kasutades saab igasuguseid programmide poolt täidetud ülesandeid automatiseerida. Näiteks saab automatiseerida evituse protsessi ja testimist. Automatiseeritus on eelis, kuna tähtsatest protsessidest eemaldatakse tundlikkus inimvigadele. Võrreldes käsitsi lehe uuendamisega on automatiseeritud evituse protsess palju veakindlam. See tähendab, et inimlikud vead, mis muidu evituse protsessi juures tekkida võivad, on kõrvaldatud. Automaatse evituse tulemus on alati sama ning ei sõltu inimlikest vigadest. Samuti võtab automaatne evituse protsess manuaalsest palju vähem aega [6].

Kasutusel on Jenkinsi keskkond, mille abil saab ühe nupuvajutusega uusima versiooni ained.ttu.ee keskkonnast laialdasse kasutusse lasta. Jenkinsis saab valida, kas installida terve keskkond uuesti või uuendada ainult mooduleid. Uuendamise eeliseks installimise ees on kiirus. Install võtab aega umbes kümme minutit, kuid ainult moodulite uuendamise peale läheb pool minutit.

Antud töös on terve evituse protsess kirjeldatud Jenkinsfile'is. Jenkinsfile on Groovy[1] keeles kirjutatud ning koosneb sammudest, mis evituse jaoks tehakse. Terve evituse protsess sõltub suuresti evituse käivitamise parameetritest. Nendeks parameetriteks on keskkond, kuhu evitus tehakse ning tegevus, mis täidetakse. Keskkond võib olla kas *develop* või *production* ning tegevus on kas install uuesti installeerimiseks või *update* olemasoleva keskkonna uuendamiseks. Joonis 8 kirjeldab evituse etapid ning sammud, mida läbitakse Jenkinsi ja Dockeri poolt.



Joonis 8. Evituse etapid.

Esimeseks sammuks installeerimisel on vajalike moodulite ja Moodle'i alla laadimine. Alla laetavad moodulid on kirjeldatud JSON kujul konfiguratsioonis (Lisa 6 - Alla laetavate moodulite konfiguratsioon) ning need kloonitakse Gitist. Giti haru nimetus,

[1] http://groovy-lang.org/

mida kloonimiseks kasutatakse, on samuti konfiguratsioonis kirjeldatud. Näiteks *production* keskkonda evitades kloonitakse Charoni `master` haru ning *develop* keskkonda evitades kloonitakse `develop` haru kasutades. Järgmiseks valmistatakse ning pakitakse `tar`[1] faili Dockeri pilt. See fail liigutatakse `scp` programmi kasutades serverisse ning käivitatakse seejärel vajalike parameetrite ja konfiguratsiooniga. Installides tuleb enne Dockeri faili käivitamist eelnevalt käiv Dockeri instants kinni panna. Uue Dockeri käivitamisel tuleb määrata port, mida Apache kasutab, lisada Dockerisse konfiguratsiooni ja SSL võtmete failid, vastendada (*map*) logi failid ning käivitada Dockeri sees olev installiskript. Konfiguratsioonifailid on *production* ja testkeskkonnas erinevad. Installiskripti sisust on eelnevalt Dockeri peatükis räägitud.

Uuendades laetakse Moodle'i moodulite uue versiooni (viimane seis Gitis) failid alla. Moodulite failid pakitakse kokku ning liigutatakse serverisse. Seejärel lisatakse need juba töötavale Dockerile, pakitakse Dockeri sees lahti ning liigutatakse Moodle'i moodulite kausta. Viimaseks käivitatakse uuendamise skript.

3.2 Raamistikud ja teegid

Antud töös on kasutatud kahte suuremat raamistikku, Laraveli ja Vuejsi. Raamistikud ja teegid teevad ära suure osa tööst, mis pole rakenduse loogikaga seotud. Projektiga, milles on kasutatud mõnda raamistikku, on kergem uutel arendajatel liituda, kuna raamistiku tundmine teeb koodis orienteerumise kergemaks.

3.2.1 Laravel

Laravel on tuntud PHP raamistik, mille abil saab arendada MVC (*model, view, controller*) mustrit kasutavaid veebirakendusi. Kasutades Laraveli raamistikku, peab programmeerija arendama vaid rakenduse funktsionaalsust, mitte tegelema päringute suunamise, HTML failide edastamisega või konteineri ja sõltuvusesüstiga (*dependency injectioniga*).

[1] https://www.gnu.org/software/tar/

Laravel pakub mitmeid mugavaid teenuseid, mida tavaliselt veebirakenduste arendamiseks vaja läheb. On võimalus kasutada Blade[1] malliprotsessorit (*template engine*), mis lubab mugavdatud PHP süntaksiga malle luua. Lisaks on Laravelis lisatud kergekaaluline objekt-relatsioonvastenduse (ORM) pakett Eloquent[2]. Eloquent pakub andmebaasiga suhtluseks väga mugavat ja voolavat süntaksit. Laravel automatiseerib sõltuvusesüsti (*dependency injection*). Teenuste konteiner loob kõik vajalikud teenuste objektid, mida klassidel vaja läheb ning sisestab need automaatselt klassi konstruktorisse. Veel on Laravelil käsurea liides Artisan[3], millega saab seadistada ning hallata erinevaid komponente. Näiteks saab Artisani abil genereerida faile, manageerida andmebaasi või teha muid administratiivseid tegevusi [1].

Kasutatud on Laraveli raamistikku, kuna töö autor on seda PHP raamistikku kõige rohkem kasutanud. Lisaks sellele pakub autorile huvi Laraveli integreerimine teiste süsteemidega. Selliselt saab õppida Laraveli struktuuri ja süsteemi paremini tundma. Laravel pakub testimiseks väga mugavat APIt. Testides saab kasutada andmebaasi nullist genereerimist või andmebaasitehinguid, et mitte mõjutada olemasolevat andmebaasi. Lisaks sellele saab andmebaasi testandmetega väikese vaevaga täita.

3.2.2 Vue

Töös on kasutuses JavaScripti raamistik Vue. Vued kasutatakse dünaamiliste kasutajaliideste ning *single page applicationite* koostamiseks.

Sarnaselt Reactile[4] kasutab Vue virtuaalset DOMi, komponentidel põhinevat arendust, kompaktset ja minimaalset koodituuma ning mitmeid teeke, mis sisaldavad lisafunktsionaalsust, nagu näiteks päringute suunamine ja globaalse oleku haldamine. Vue eelis Reacti ja teiste sarnaste raamistike (nt Angular[5]) ees on arendajasõbralikkus, kiirus ja komponentides vaid HTMLi kasutamine. Vue kasutamine on uutele arendajatele palju kergem, kuna see ei eelda mitmete JavaScripti teekide ja

[1] https://laravel.com/docs/5.4/blade

[2] https://laravel.com/docs/5.4/eloquent

[3] https://laravel.com/docs/5.4/artisan

[4] https://facebook.github.io/react/

[5] https://angularjs.org/

kompileerimisvõtete oskust. Samuti ei eelda see JSX kasutamise oskust, mida Reacti kasutajad standardselt kasutavad, vaid kasutada saab tavalist HTMLi ja JavaScripti. Kiiruse võrdlustestis võidab Vue Reacti ligi kahekordse kiiruse eelisega [3].

Antud töös on kasutatud Vued, kuna töö autoril on sellega JavaScripti raamistikest kõige rohkem kogemusi. Vue on kasutuses kõikidel lehtedel ja vormidel, et need dünaamiliseks teha. Kõige rohkem on seda kasutatud hindamise liideses, mis on realiseeritud *single page application*‘ina ning koosneb ainult Vue komponentidest.

Kasutades Vued ning selle komponentide süsteemi saab *front-end* koodi jagada osadeks. Iga komponent tegeleb vaid enda ülesannetega ja ei mõjuta otseselt teisi komponente. Selliselt tekib palju faile, kuid failide pikkused on väikesed. Kui mõne komponendi tööd on tarvis muuta, siis saab seda lühikestes failides kergemalt teha kui pikkades failides.

4 Lahendus

Järgnevalt on kirjeldatud eesmärgi lahendamine. Lahti on seletatud mooduli funktsionaalsus, projekti struktuur, Moodle'i mähis ning mooduli automaattestimine.

4.1 Funktsionaalsus

Moodul peab toetama tavalisi Moodle'i mooduli võimalusi nagu ülesande loomine, muutmine ja vaatamine. Lisaks peab moodul võtma vastu info selle kohta, kui tudeng laeb oma koodi Giti ning selle info edastama testerile. Tester saadab moodulile testide tulemused ning moodul salvestab need. Tulemuste saatmine on turvatud, nimelt saadab moodul iga esitusega koos testerile salajase koodi, mida ootab testrilt tagasi. Kontrollitakse, et testeri saadetud kood vastab sinna saadetud koodile.

Iga kursusega on seotud *preset*'id, ehk ülesande loomise jaoks kasutatavad mallid. Mallid koosnevad nimest, testerile saadetavatest ekstra parameetritest, ülesande eest saadavad maksimumpunktid (kõik hinde vastavused kokku), hindamise meetodist, hinde arvutamise valemist ning hinde vastavustest (*grademap*). Malli rakendatakse ülesande loomise vormis, malli valides rakendatakse kõik *preset*'i väljad ülesandele.

4.1.1 Vormid

Töös on kasutusel kaks olulisemat vormi: kursuse seadete vorm ja ülesande loomise vorm. Järgnevalt kirjeldatakse neid vorme.

Iga kursusega on seotud kursuse seaded (Lisa 1 - Kursuse seaded). Seadetes saab märkida kursuse ülesannete kohta ühtset testeri tüüpi (programmeerimise keelt) ning automaattestide Giti. Samuti saab kursuse seadetes lisada, muuta ja vaadata ülesannete malle.

Ülesande loomise vorm, ehk *instance form*, koosneb kolmest osast: ülesande info, hindamise info ja tähtajad. Ülesande info ja hindamise sektsioon on algselt välja näidatud minimeeritud kujul (Lisa 2 - Ülesande loomise minimaalne vorm). Minimeeritud vormis saab valida vaid ülesande malli ning näha ülesande andmeid. Muuta saab ülesande pealkirja ja kirjeldust. Minimaalse vaate kasutaja valib endale sobiva malli, sellega ülesande eest punktide saamise konfigureerimine piirdub. Selline

minimaalne vaade aitab vähendada vormis olevat info üleküllust ning kiirendab ülesande loomise etappi.

Ülesande vormis on detailsem vaade (*advanced* vaade), mis näitab ülesande ja hindamise kohta rohkem infot ning võimaldab nende seadete muutmist (Lisa 3 – Ülesande loomise detailne vorm). Ülesande info sektsioonis saab muuta testerile saadetavaid ekstra parameetreid ja testeri tüüpi. Hindamise osas saab muuta hindamise meetodit, hinnete (*grademap*) infot, ülesande eest saadavaid maksimumpunkte ja ülesande hinde arvutamise valemit.

4.1.2 Hindamise liides

Moodul sisaldab õppejõule mugavamat ja kiiremat hindamise kasutajaliidest (*popup*), mida kasutades saab programmeerimisülesandeid kontrollida ja hinnata. Kuna Moodle'i hinderaamatus laetakse terve kursuse kõikide tudengite kõik hinded korraga suurde tabelisse, on see mahukate kursuste puhul väga aeglane ning sealt õigeid hindeid ja tudengeid leida on ebamugav. Loodav hindamise liides on Moodle'i hinderaamatust kiirem ja mugavam, kuna kasutatakse AJAX tehnoloogiat, et näidata vaid küsitud välju. Liides näitab Charoniga seotud välju nagu tudengi esitatud kood ning esituse Giti kehtestuse info. Lisaks laetakse alati vaid minimaalne info, mida antud olukorras vaja läheb. Näiteks ühe tudengi ühe ülesande esitustes sisaldub vaid esituse aeg, info esituse kinnitamise kohta ja tulemuste punktid.

Hindamise alustuseks otsib õppejõud tudengit kasutades automaatselt tudengeid soovitavat otsingukasti (*autocomplete*). See tähendab, et ei pea kõikide õpilaste nimede hulgast õiget otsima, vaid saab neid nime järgi filtreerida. Seejärel saab valida ülesannet, mille esitusi kontrollima hakatakse. Liidest kasutades saab hinnata vaid Charoni ülesandeid. Kui valitud on tudeng ja ülesanne, siis näidatakse viimaseid tudengi esitusi selle ülesande kohta (Lisa 4 – Hindamise liidese esituste vaade). Kui klõpsida esituse peal, siis näidatakse välja ühe esituse info (Lisa 5 – Hindamise liidese ühe esituse vaade). Sealhulgas on näiteks tudengile testerist saadud tagasiside, tudengi programmikood ning testide jooksutamise väljund. Esituse täpsema info vaates saab muuta esituse eest saadud hindeid. Lisatud on tulemusele maksimaalsete punktide panemise nupp, millega saab ühe hinde eest tudengile maksimaalsed punktid panna. Nupu kasutamine kiirendab oluliselt hindamisprotsessi.

Hindamise liideses näeb iga tudengi edukust. Näidatakse iga tudengi tervet kursuse hinderaamatut, mille genereerib Moodle. Hinderaamatus on kõik aine ülesanded, nende hulgas on lisaks Charoni ülesannetele tavalised ülesanded ja nende tulemused. Näeb kui palju tudengil kursuse peale punkte kokku on ning kui palju punkte kursusel maksimaalselt teenida saab.

4.1.3 Lahenduse esitamine

Lahenduse esitamine toimub kahes etapis. Esiteks laeb tudeng oma lahenduse enda Giti salve üles. Gitis on üles seatud haak (*hook*), mis saadab iga Giti kehtestuse ja *push*'imisega Charonile sellest teate HTTP GET päringuga. Charonile saadetakse tudengi Uni-ID, salve aadress ja lisaks muu info, mida testeril Gitist lahenduse kloonimiseks vaja võib minna. Kui päring jõuab Charonile, salvestatakse selle kohta info andmebaasi. Lisaks genereeritakse salajane võti, mida kasutatakse testeriga turvaliseks suhtluseks. Pärast salvestamist käivitatakse PHP koodis Laraveli sündmus `GitCallbackReceived`, mille kuulajad on märgitud `TTU\Charon\Providers\EventServiceProvider` klassis `$listen` atribuudis. Sellele sündmusele reageerib kuulaja `ForwardGitCallbackToTester`. Kuulaja kasutab `TesterCommunicationService` klassi, et testerile vajalikud kehtestusega seotud andmed edasi saata. Samuti saadetakse testerile URL, kuhu tester automaattestide tulemused saadab. Kui tester esitamise kohta info saab, siis pannakse esitus testimise ootejärjekorda ja tagastatakse Charonile sõnumi, et andmed on kätte saadud. Testimine toimub testeris asünkroonselt nii, et Charon ei pea ühe päringu vältel ootama, kuni testimine valmis saab. Charon tagastab päringu lõpus eduka esitamise sõnumi Giti ning seda näidatakse `git push` käsu tulemusel käsureal tudengile.

Kui tester on automaattestidega valmis saanud, saadetakse testeri poolt tudengile e-post, kus on kirjas testide tulemused. Lisaks saadetakse Charonile HTTP POST päring testide tulemusega. Lisaks tulemustele on päringus enne mooduli poolt genereeritud salajane võti, Charoni instantsi `id`, tudengi Uni-ID, Giti kehtestuse info ja esitatud failid. Charon kontrollib päringut vastu võttes salajase võtme õigsust ning seda, kas esituse info on andmebaasis olemas. Kui turvaetapid on läbitud, siis salvestatakse esitus ning arvutatakse sellele hinded.

Kuna tester ei saada manuaalsete hinnete tulemusi, vaid ainult automaattestide ja stiilikontrolli tulemused, siis genereeritakse need Charoni poolt nulli punktiga. Lisaks genereeritakse igaks juhuks kõik tulemused, mida tester ei saatnud, aga mille kohta hinde vastavus on olemas. Iga tulemuse kohta hoitakse testide läbimise protsenti, mis tester saadab ning kalkuleeritud tulemust, mis sõltub hinde maksimumpunktidest ja tähtajast ning mis arvutatakse Charoni poolt. Hinde arvutamisel kontrollitakse esiteks, kas tulemus on testide, stiili või manuaalne hinne. Kui tulemus on stiili või manuaalse hinde kohta, siis ei rakendata sellele tähtaega, vaid arvutatakse testeri saadetud protsentide järgi kalkuleeritud hinne maksimumi kohta. Automaattestide hindele lisatakse tähtaja kontrollimine.

Näiteks võib võtta teises peatükis kasutatud näiteülesannet, mis koosnes testidest (kaks punkti), stiilist (üks punkt) ja kaitsmisest (üks punkt). Tester saadaks siis esitusega koos kaks tulemust, testide ja stiili kohta. Manuaalse hinde (kaitsmise) kohta tester tulemust ei saada. Kui ülesandel on tähtaeg, pärast mida saab 80% punktidest, siis kui tudeng sai pärast tähtaega esitades pooled testid läbi (50%) ning stiilikontrolli täiesti läbi (100%), saadab tester tulemustes 50% testide eest ja 100% stiili eest. Edasi kontrollib Charon testide hinde tähtaega ning arvutab lõpptulemuse kokku. Ilma tähtaega arvestamata on kalkuleeritud tulemuseks üks punkt (2p korda 50%), kuid siis arvestatakse lisaks tähtaega ning saadakse lõpptulemuseks 0,8 punkti (2 * 0,5 * 0,8). Stiili hindele tähtaeg ei rakendu ning tulemuseks on üks punkt. Kaitsmise hinde loob Charon ning paneb selle punktideks null punkti. Nüüd on tudengil esitus, mille eest on ta kokku saanud null punkti (0,8p testidest korda 1p stiili eest korda 0p kaitsmise eest). Kui tudeng kaitseb oma lahenduse edukalt õppejõule, siis muudab õppejõud kaitsmise punktid ühe punkti peale. Ehk lõpptulemus on 0,8 punkti (0,8p korda 1p korda 1p kaitsmise eest).

Järgmisena kontrollitakse, kas saadud tulemus peaks üle kirjutama Moodle'is oleva eelneva tulemuse. Moodle'is arvestatud tulemus on see, mida Moodle'i hinderaamatus näidatakse ning mida valemite järgi hinnete arvutamiseks kasutatakse. Ehk Moodle'is arvestatud hinne on aktiivne. Selleks, kas uus esitus peaks olema aktiivne, kontrollitakse ülesande loomisel sätestatud hindamise meetodit. Hindamise meetodeid on kaks: parima eelistamine ja uusima eelistamine. Kui eelistatakse parimat esitust, siis kontrollitakse varem aktiivse esituse punkte ja võrreldakse neid uue esitusega. Kui uuel esitusel on rohkem punkte, siis aktiveeritakse see. Kui eelistatakse uusimat esitust, siis aktiveeritakse uus esitus alati. Lisaks sellele kinnitatakse esitus siis, kui õppejõud

tudengile hinde paneb. Esituse kinnitamine tähendab, et järgnevaid automaattesterilt saadetud esitusi ei aktiveerita ning kinnitatud esitus on alati aktiivne. Kinnitamine aitab parandada probleemi, kui tudeng esitab kaitsmisega ülesande ning kaitseb ülesande ära, kuid siis esitab ülesande uuesti. Uuesti esitades läheksid saadud punktid kaotsi, sest võibolla aktiveeritakse uusim esitus, millel pole kaitsmise eest hinnet. Ehk, kui on mõni esitus, mis on kinnitatud, siis uusi esitusi ei kinnitata. Järgmiseks alustatakse hinnete Moodle'is aktiveerimist, kui seda vaja teha on. Iga tulemuse kohta uuendatakse hinde vastavust Moodle'is. Kasutades autori loodud Moodle'i mähise teenuse `GradingService` meetodit `updateGrade` uuendatakse tudengi hinne Moodle'is. Ei kasutata Moodle'i hindamise funktsioone otse, vaid Moodle'i APIga suhtlemine toimub läbi mähise.

Testeriga suhtluse päringud ja vastused on ära kirjeldatud kasutades Swaggerit[1] ning selle järgi on genereeritud veebileht Charoni API dokumentatsioon[2]. Kirjeldatud on seda, milliseid päringuid tester Charoni API vastu teha saab ning mis nende päringute vastused on. Samuti kirjeldatakse päringuid, mida Charon testerile teeb ning mida tagasi oodatakse.

4.2 Struktuur

Plugina failid on jaotatud üldiselt kolmeks. Esiteks on Moodle'i soovitud failid, mis peaksid igal moodulil olema ja asuvad juurkaustas. Teiseks on PHP failid, mis moodustavad Laraveli projekti ja asuvad `plugin` kaustas. Kolmanda projekti osa moodustavad JavaScripti failid, mis asuvad `plugin/resources/assets/js` kataloogis. Järgnevalt on kirjeldatud iga projekti osa struktuuri.

4.2.1 Moodle'i failid

Moodle ootab, et igal moodulil oleksid kindlad PHP failid, mida saab importida ja vajadusel käivitada. Moodle'ile vajalikud failid on järgnevad:

[1] http://swagger.io/

[2] https://ained.ttu.ee/mod/charon/documentation

- `version.php` – Sisaldab mooduli ja vajaliku Moodle'i versiooni numbreid. Moodle kontrollib pidevalt versiooni numbrit ning selle muutudes käivitab mooduli uuendamise skripti. Uuendamise skriptis modifitseeritakse tavaliselt andmebaasi või muudetakse seal andmeid.

- `lib.php` – Sisaldab mitmeid funktsioone, mida Moodle välja kutsub, kui plugina isendit lisatakse, muudetakse või kustutatakse. Lisamiseks on antud moodulis funktsioon `charon_add_instance`, muutmiseks `charon_update_instance` ja kustutamiseks `charon_delete_instance`. Lisaks saab laiendada mõnda Moodle'i funktsionaalsust, näiteks erinevatel lehekülgedel olevad navigatsioonimenüüd. Funktsioon `charon_extend_navigation_course` lisab kursuse kontekstis asuvatele menüüdele lingi Charoni popupile ja kursuse seadetele.

- `settings.php` – Siin asuvad mooduli seaded, mida saab muuta vaid lehe administraator. Need seaded käivad Charoni kohta terve õppesüsteemi ulatuses. Charonil saab selliselt muuta testeri URLi. See on URL, mida kasutatakse testeriga suhtlemiseks.

- Kaust `lang/` - Sisaldab mooduli tekstide tõlkeid erinevatesse keeltesse. Charonil on tõlgitud plugina tekstid nii eesti, kui inglise keelde.

- Kaust `db/` - Plugina installeerimise ja uuendamise skriptid. Fail `install.xml` kirjeldab andmebaasi skeemi XML kujul. Mooduli installi ajal genereerib Moodle sellest failist andmebaasi täitmiseks SQL laused ning käivitab need.

- `db/events.php` – Moodle'i sündmuste kuulajate registreerimine. Moodle käivitab mitmeid erinevaid sündmusi, mida moodulid kuulata saavad. Antud moodul kuulab sündmusi `course_module_created` (ülesanne loodi) ja `course_module_updated` (ülesannet muudeti).

- `db/access.php` – Charoniga seostuvate õiguste kirjeldamine. Õiguste alla käib näiteks see, mis rolliga kasutajad Charoni ülesannet muuta saavad. Kasutatakse Moodle'i õiguseid ning kloonitakse kursuse ülesannete manageerimise õigused. Ehk Charoni muutmiseks on samad õigused, mis kursuse teiste ülesannete muutmiseks.

- Kaust `classes/` - Sündmuste kuulajate klassid. Nimeruumis `mod_charon` asuvad ülalpool kirjeldatud sündmuste kuulaja klassid. Igas kuulaja klassis on üks funktsioon, mis käivitatakse kui sündmus käivitatakse.

- kaust `pix/` - Siin asuvad mooduliga seotud pildid ja ikoonid. Ikoon on `png`, `gif` ja `svg` formaadis. Ikoon on autori poolt loodud.

Kõikides ülaltoodud failides on kasutatud Moodle'i programset liidest või loodud uus Laraveli rakenduse instants, mis tegeleb päringuga. Failis `lib.php` asuvad mooduli instantsi lisamise ja muutmise funktsioonid. Lisamise ja muutmise päringud ei lähe otse Laraveli, vaid need funktsioonid kutsutakse Moodle'i poolt välja. Selle pärast luuakse uus Laraveli rakendus (`app`) ja päring (`request`) failis `plugin/bootstrap/helpers.php`. Rakendusele antakse ette genereeritud päring ning käivitatakse Laraveli päringule vastamise protsess. Vastus tagastatakse ülesande lisamise funktsioonis Moodle'ile.

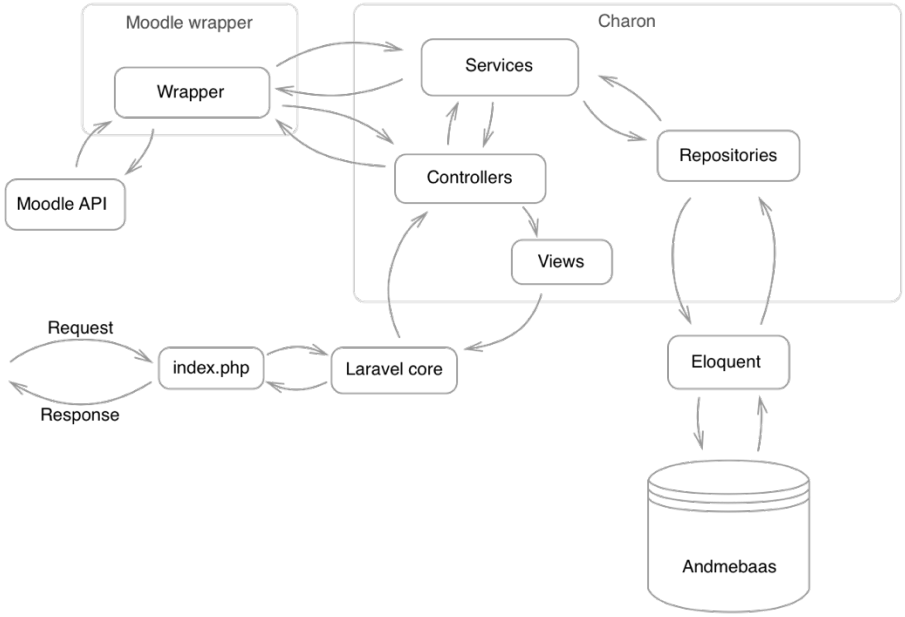
4.2.2 Laraveli projekti struktuur

Rakenduse põhiline loogika asub PHP failides kaustas `app/`, andmebaasiga seotud failid asuvad `database/` kaustas, rakenduse konfiguratsioon asub `config/` kaustas, ressursid nagu JavaScript, CSS ja HTML mallid asuvad `resources/` kaustas, ruutimise konfiguratsioon asub `routes/` kaustas ning testid asuvad `tests/` kaustas.

Rakenduse äriloogika on üldiselt jaotatud MVC põhimõtetel mudeliks (*model*), vaadedeteks (*view*) ja kontrolleriteks (*controller*). Lisaks eelnevatele on kasutuses teenused, sündmused ja hoidla (*repository*). Teenused, ehk *service*'id, pakuvad varieeruvaid teenuseid, mida kontroller välja kutsub. Teenused aitavad vähendada kontrollerites toimuvaid tegevusi ning hoida kontroller võimalikult minimaalsena. Sündmustega (*event*) lahutatakse osa tegevustest rakenduse tööst ja suunatakse see ümber sündmuse kuulajatele. Kasutusel on sündmused `CharonCreated` ja `GitCallbackReceived`, mida kuulavad ja millele reageerivad `SendAddProjectInfoToTester` ja `ForwardGitCallbackToTester` sündmuste kuulajad. Sündmuste ja kuulajate vahel luuakse seosed `TTU\Charon\Providers\EventServiceProvider` klassis. Sündmustele saab tulevikus vajadusel rohkem kuulajaid registreerida ning neid WebSocketeid kasutades kliendi JavaScripti edastada. Hoidlad loevad ja salvestavad andmeid andmebaasi.

Kasutades hoidlaid saab andmebaasi kasutavat kihti makettida (*mock*) ning teenuseid kergemini ühiktestida. Samuti on siis kogu andmebaasiga suhtlus koondatud paari faili.

Joonis 9 kujutab üldist süsteemi arhitektuuri. Kõik päringud võtab vastu juurkaustas olev `index.php` fail, mis loob Laraveli instantsi ning annab päringu sellele edasi. Laravel leiab päringu URLile vastava kontrolleri ja selle meetodi, mis pannakse käima. Kontroller kasutab nii teenuseid kui ka autori poolt loodud Moodle'i mähist. Teenused kasutavad repositooriume, et andmebaasiga suhelda, ning Moodle'i mähist, et Moodle'i teenuseid kasutada. Repositooriumid kasutavad andmebaasiga suheldes Eloquent ORMi. Kontroller tagastab vastuse, mis saadetakse läbi Laraveli kliendile tagasi.



Joonis 9. Süsteemi arhitektuur.

4.2.3 JavaScripti failide struktuur

Töös on kasutuses kolme liiki JavaScripti faile: JavaScripti ES2015 süntaksiga failid, Vue komponentide failid ning kompileeritud ja minimeeritud tavalise JavaScripti failid. ES2015 JavaScripti ja Vue komponentide failid asuvad kaustas `plugin/resources/assets/js` ning kompileeritud JavaScripti failid asuvad kaustas `plugin/public/js`. Komponentide failid on jaotatud lehekülgede kaupa

`pages` kaustas ülesande vaate, kursuse seadete, instantsi vormi ja *popup*'i kaustadesse. Üldised Vue komponendid, mida kasutatakse mitmes vaates, asuvad `plugin/resources/assets/js/components` kaustas.

JavaScripti kood on sarnaselt PHP koodiga jaotatud mudeli (*model*) ja vaate (*view*) osadeks, kus mudel tegeleb Charoni APIga suhtlusega ning *view* määrab kasutajaliidese välimuse ning käitumise. Mudelid asuvad *models* kaustas, kus on staatilised klassid. Igal klassil on meetodid, mille abil saab APIga suhelda. Mudelid saadavad andmete lugemise, muutmise ja kustutamise päringuid. Klassideks on näiteks `Charon`, millel on staatiline `all` meetod, mis tagastab kõik Charoni isendid kursuses. HTTP päringute saatmiseks ja vastuvõtmiseks on kasutusel teek axios[1]. Kõik mudelite klassid pakuvad võimalust argumendina kaasa anda tagasikutse funktsiooni, mis pärast päringu õnnestumist käivitatakse. Selliselt saab pärast andmete küsimist neid võimalikult kiiresti kasutada.

Iga Vue komponendi fail, ehk JavaScripti vaate osa, on jagatud kolmeks. `Template` HTML sildi (*tag*) sees asub komponenti kirjeldav HTML, `script` sildis asub komponendi funktsionaalsust kirjeldav JavaScript ning `style` sildis asub komponendile mõjuv stiil. Antud töös ei ole stiil paigutatud iga komponendi faili vaid see on kirjeldatud eraldi SCSS[2] failides.

Vue komponentide vaheline suhtlus käib kas lapselt vanemale sündmuste saatmisega või kasutades globaalset sündmuste jaoks loodud Vue objekti `VueEvent`, mis käitub nagu sündmuste *bus*. Globaalsete sündmuste välja saatmiseks saab kasutada `VueEvent` objekti `$emit` meetodit. Globaalsele sündmusele reageerimiseks tuleb registreerida sündmuse kuulaja tagasikutse, ehk *callback* kasutades `VueEvent`'i `$on` meetodit. Näiteks sündmuse välja laskmine: `VueEvent.$emit('show-notification')`.

Töös on kasutuses andmeid hoidvad konteksti objektid, mis asuvad `classes` kaustas. Kontekst on lehe andmete hetkeseisund. Kontekst võimaldab vormi väljade väärtusi mälus hoida ja neid JavaScriptis kontrollida. Charoni instantsi vormis on kontekstis

[1] https://github.com/mzabriskie/axios

[2] http://sass-lang.com/

salvestatud kõikide väljade väärtused ning see pakub *grademap*'ide ja *deadline*'ide lisamiseks ja eemaldamiseks liidest. Isendi vormi kontekst tegeleb ülesande malli valimise kasutusjuhuga, mis juhul uuendatakse kõiki konteksti vormi välju selliselt nagu valitud mall seda ette näeb. Hindamise liidese kontekst hoiab mälus kursust, aktiivset tudengit, aktiivset Charonit ja aktiivset esitust.

Kuna projektis on kasutatud SCSSi, ES2015 süntaksiga Javascripti ja Vue komponente, peab need failid kompileerima tavaliseks JavaScriptiks ja CSSiks. Laravel soovitab selleks NodeJS[1] teeki Laravel Mix, mis sisaldab endas webpacki[2] konfiguratsiooni ja lihtsat APIt, mida kasutades saab kompileerimise etappe kirjeldada. Laravel Mixi abiga saab ilma webpacki konfiguratsiooni muutmata kompileerida Vue komponente, ES2015 süntaksiga JavaScripti, ja SCSSi. Failis `webpack.mix.js` on kirjeldatud, milliseid faile webpack kompileerib. Selleks tuleb kasutada Laravel Mix teegist imporditud muutuja `mix` meetodeid `js` ja `sass` [4].

4.3 Moodle'i mähis

Kuna Moodle on suur ning vana süsteem (esimene väljalase aastas 2002), siis on Moodle'i koodibaasis kasutusel vanamoodne ning raskesti kasutatava liidesega kood. Kasutatakse rohkesti globaalseid muutujaid ja funktsioone ning ei kasutata nimeruume. Samuti on rakendusliidese kasutamine vaevarikas, kuna tihti peab lihtsate asjade tegemiseks mitu rida koodi kirjutama. Kõik see teeb arendamise vaevarikkaks ning uusi arendajaid eemale tõrjuvaks. Samuti ei saa Moodle'i klassidega kasutada Laraveli automaatset sõltuvusesüsti (*dependency injection*) kuna Moodle'i klasside sõltuvusi Laravel lahendada ei oska.

Moodle'i rakendusliidese mugavamaks kasutamiseks on autor loonud mähise, mis lubab kasutada Moodle'i API funktsionaalsust läbi kasutajasõbraliku liidese. Mähise arenduses on rõhku pööratud Laraveli automaatse sõltuvusesüstimise võimaldamisele ja kasutajamugavale liidesele. Mähises on mähitud vaid selline funktsionaalsus, mida

[1] https://nodejs.org/en/

[2] https://webpack.github.io/

Charon kasutab, kuid autor on avatud selle funktsionaalsuse laiendamisele. Kõik mähise klassid asuvad `Zeizig\Moodle` nimeruumis, et vältida konflikti teiste teekidega.

Loodud mähis on kollektsioon mitmetest teenustest ning abifunktsioonidest, mudelitest ja globaalseid muutujaid mähkivatest klassidest. Teenused asuvad `Services` kaustas ning pakuvad üldist moodulil vaja minevat funktsionaalsust. Näiteks on `GradebookService`, mis tegeleb Moodle'i hinderaamatu muutmisega, sinna *grade item'ite* ja kategooriate lisamisega ning arvutuste valemite parsimisega. Abistavad globaalsed funktsioonid, *helper*'id, mähivad endas Moodle'i funktsioone, mida on mugav globaalselt kasutada. Näiteks kasutatakse HTML mallide failides `translate` funktsiooni, mis tõlgib antud sõna kasutaja valitud keelde. Tõlkimise funktsiooni kasutamine käib järgmiselt: `translate('sõna')`, Moodle'i funktsioonina peaks seda nii kasutama: `get_string('sõna', 'mod_charon')`, mis ei ole nii mugav. Mudelid on Eloquent ORMi mudelid, mis kirjeldavad Moodle'i baastabeleid. Kuna selliseid mudeleid läheb paljudes moodulites vaja, on autor otsustanud need paketti lisada. Näiteks on mudeliteks `User`, `Course` ja `GradeItem`. Globaalsete muutujate mähised asuvad `Globals` kaustas. Lisaks otseste atribuutide mähkimisele pakuvad need klassid kergemini kasutatavaid lisafunktsioone, nagu näiteks `User::isEnrolled` meetod, mis kontrollib tudengi kursuse õpilaseks olemise staatust. Globaalsete muutujate mähisteks on näiteks Moodle'i `$USER` muutujat mähkiv `User` klass ning `$PAGE` muutujat mähkiv `Page` klass.

4.4 Testimine

Moodle'i baaskood ei ole alati jaotatud klassidesse vaid kasutatakse globaalseid funktsioone. Selle miinuseks on see, et funktsioone ei saa sõltuvusesüstiga objektidesse sisestada ning nende testide jaoks makettimine (*mock*) on raske. Moodle'i mähise objekte saab sõltuvuse süstiga Charoni objektidele kaasa anda. Selliselt objektidele kaasa antud objekte saab testimisel kergelt makettida ning vähendada ühiktestide skoopi. Kui testides selliseid sisendeid makettida ei saa, siis pannakse sõltuvustes olev funktsionaalsus igas testis päriselt tööle. Näiteks ilma makettimiseta pannakse tudengile igal testil hindeid või saadetakse testerile päringuid.

Antud töö automaattestimiseks kasutatakse testimise raamistikku PHPUnitit[1] ning makettide loomiseks Mockito[2] teeki. Lisaks sellele pakub Laravel mugavaid testimise tööriistu.

Testimine on jaotatud kaheks: ühiktestimine (*unit tests*) ja funktsionaalsuse testimine (*feature tests*). Testid käituvad nagu dokumentatsioon. See tähendab, et testide nimedest saab välja lugeda nõutud funktsionaalsust. Näiteks test `create_form_shows_previous_values_after_error` näitab, et peale mõnda viga Charoni loomise vormis peab sisendikastides vanu väärtuseid välja näitama. Veel ühe näitena võib tuua testi `creating_a_charon_notifies_tester_of_the_new_charon` mis nõuab, et uue Charoni isendi salvestamisel saadetakse testerile selle kohta info.

Järgnevalt on toodud välja üks test, mis illustreerib, kuidas antud töös testimine teostatud on (Joonis 10). Testi nimi väljendab võimalikult täpselt seda, mida testitakse ning käitub dokumentatsioonina. Joonisel olev test kontrollib, et luues uut Charonit saadetakse testerile selle kohta info. Kuna testimise ajal ei tohiks andmeid tegelikult testerile saata, siis kasutatakse Laraveli poolt pakutud meetodit `Event::fake()`, mis ei lase sündmuste kuulajatel sündmustele reageerida. See tähendab, et kuulajat, mis saadab testerile infot, ei panda testi ajal tööle. Järgmisena seatakse üles kursus koos kursuse seadetega (kasutades `factory` funktsiooni). Seejärel genereeritakse suvalised uut Charonit kirjeldavad parameetrid `getRandomRequest` meetodiga. Genereeritud andmeid kasutades tehakse päring, et luua uus Charon. Loodud Charon küsitakse andmebaasist ning tagastatakse `makeStoreRequestAndGetCharon` meetodist. Viimaseks nõutakse `Event::assertDispatched` klausliga sündmuse õigete parameetritega välja saatmist. Kuna kuulaja `SendAddProjectInfoToTester` kuulab `CharonCreated` sündmust, siis pole vaja kuulajat reaalset tööle panna.

[1] https://phpunit.de/

[2] http://site.mockito.org/

```
public function creating_a_charon_notifies_tester_of_the_new_charon()
{
    Event::fake();

    $courseSettings = factory(CourseSettings::class)->create();
    $requestParams = $this->getRandomRequest($courseSettings->course_id);

    $charon = $this->makeStoreRequestAndGetCharon($requestParams);

    Event::assertDispatched(CharonCreated::class, function ($e) use ($charon)
{
        return $e->charon->id === $charon->id;
    });
}
```

Joonis 10. Testi näide.

5 Kokkuvõte

Töö eesmärkideks oli luua Moodle'i õppekeskkonda moodul, mis võimaldab luua programmeerimisülesandeid. Moodul peab testerilt võtma vastu ülesannete tulemusi. Lisaks oli tarvis luua õppejõule mugav liides, mida kasutades saab mooduli ülesandeid hinnata. Viimaseks oli vaja realiseerida automaatne mooduli uuendamine ning ained.ttu.ee keskkonna evituse protsess.

Eesmärgi saavutamiseks loodi ülesande moodul, mis võtab testerilt vastu tulemusi ning salvestab need. Moodul loodi kasutades Laraveli raamistikku. Loodi 172 PHP faili 5700 koodireaga, 47 Vue faili 2600 koodireaga ning 36 JavaScripti faili 660 koodireaga. Lisaks loodi mitmeid eri liiki faile nagu SCSS, Blade ja XML failid. Moodul suhtleb testeriga turvaliselt, kasutades salajast võtit. Loodud moodul pakub ka õppejõule mugavat hindamise liidest.

Teise eesmärgi saavutamiseks realiseeriti ained.ttu.ee keskkonna evituse protsess ning automaatne mooduli uuendamine. Selleks kasutati pidevkoostet ja Dockerit ning Jenkinsit. Jenkinsfile'is ning Dockerfile'is kirjeldati evituse etapid. Tulemuseks on võimalik ühe nupuvajutusega ained.ttu.ee keskkonda installeerida või uuendada. Jenkinsfile koosnes 230st koodireast ning Dockerfile koos installiskriptiga 200st reast.

Moodul on juba terve 2017 aasta kevadsemestri olnud kahes TTÜ programmeerimisaines aktiivses kasutuses ning seda kasutatakse ka edaspidi. Esitusi on semestri jooksul laekunud umbes 50 000 ja kasutades loodud hindamise liidest on hindamist sooritatud 3600 korda. Sügissemestril kasutatakse moodulit rohkemates ainetes. Plaanis on leida tudengeid, kes moodulit hooldama hakkavad ning lisafunktsionaalsust sellesse lisavad. Kogu lähtekood on üles laetud TTÜ Giti salve aadressil https://gitlab.cs.ttu.ee/ained/charon.

Kasutatud kirjandus

[1] 9 Benefits of Laravel Application Development. [WWW] http://www.cmarix.com/9-benefits-of-laravel-application-development/ (12.05.2017)

[2] A successful Git branching model. [WWW] http://nvie.com/posts/a-successful-git-branching-model/ (12.05.2017)

[3] Comparison with Other Frameworks. [WWW] https://vuejs.org/v2/guide/comparison.html (15.04.2017)

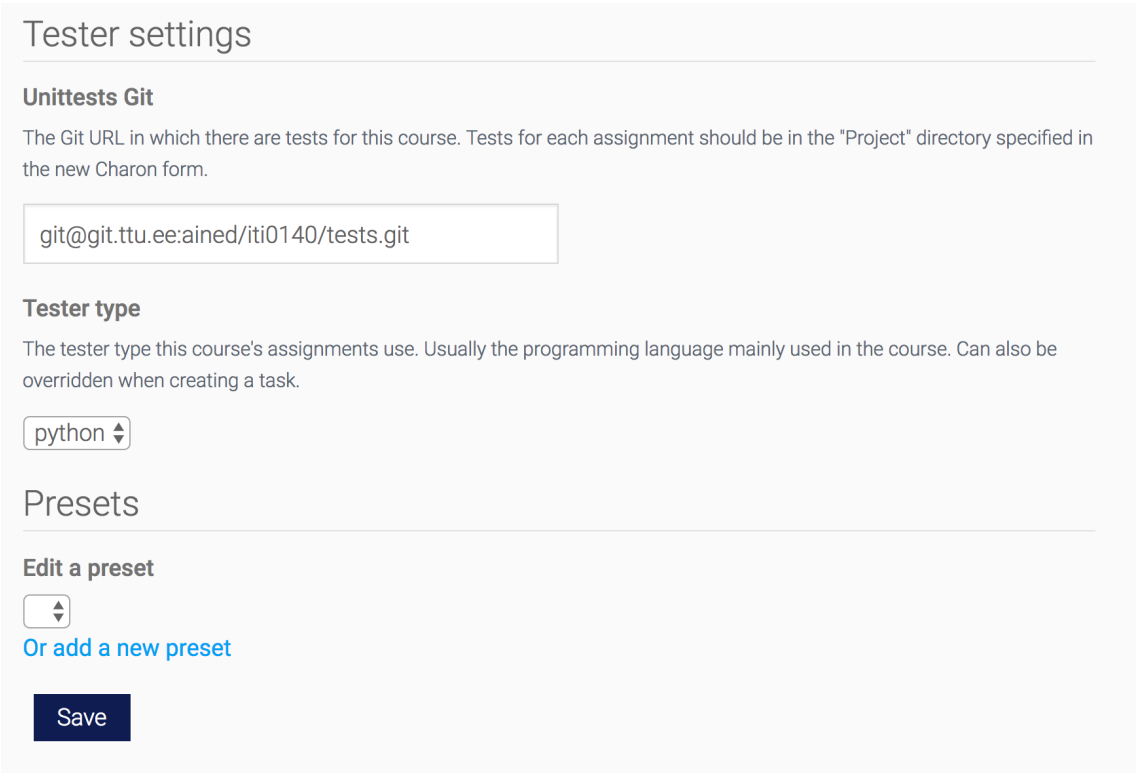
[4] Compiling Assets (Laravel Mix). [WWW] https://laravel.com/docs/5.4/mix (18.04.2017)

[5] Docker. [WWW] https://www.docker.com/ (10.05.2017)

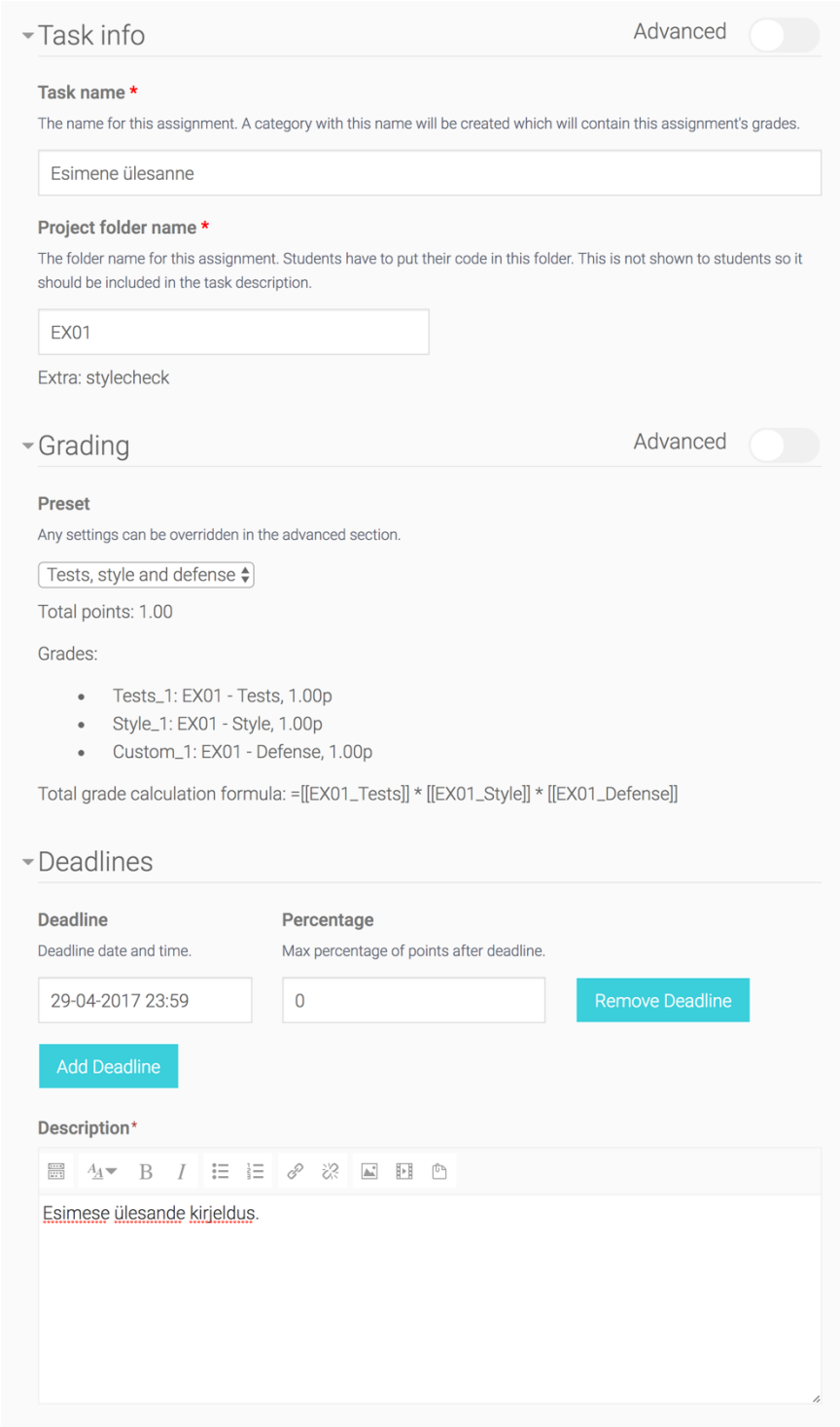
[6] Jenkins. [WWW] https://jenkins.io/. (10.05.2017)

[7] Why Continuous Integration Is Important. [WWW] https://blog.codeship.com/continuous-integration-important/ (12.05.2017)

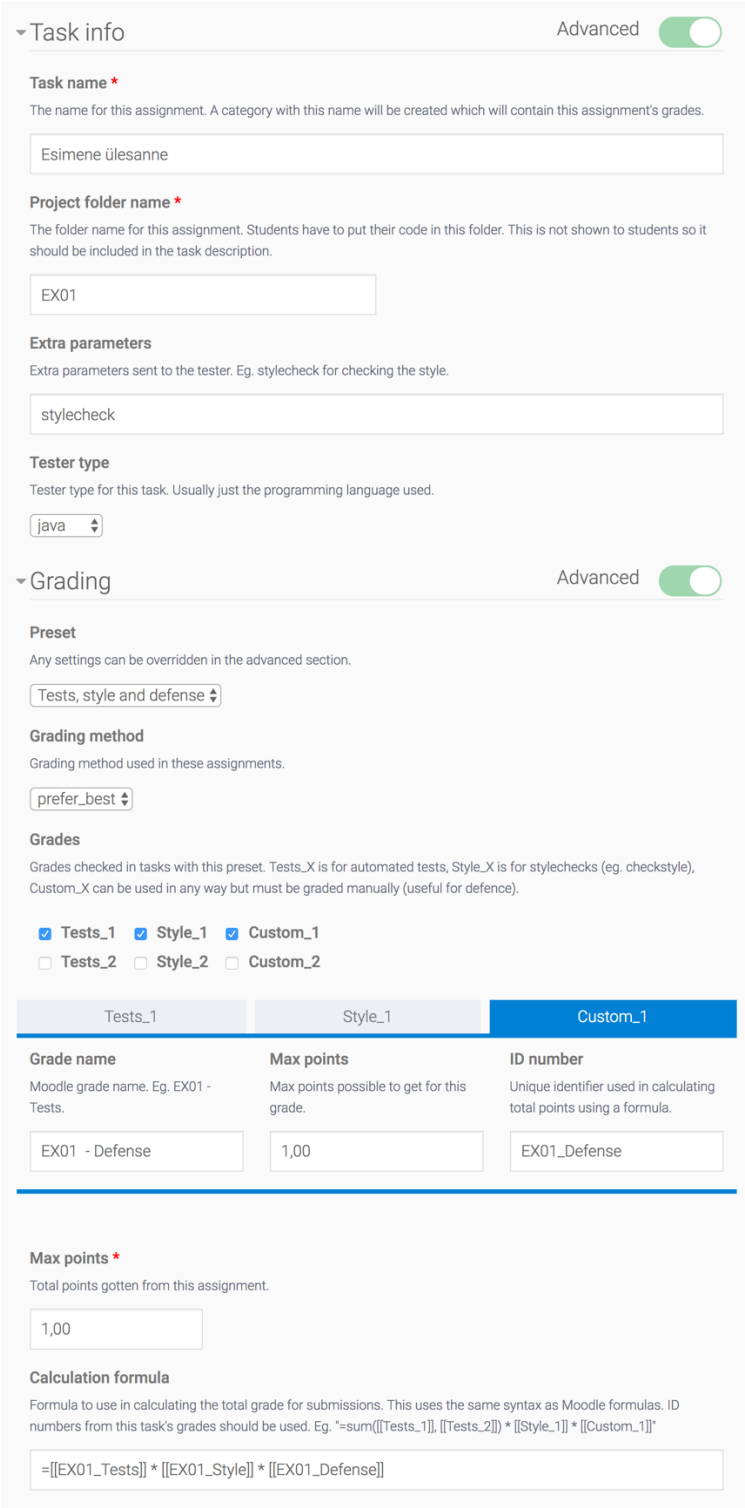
Lisa 1 - Kursuse seaded



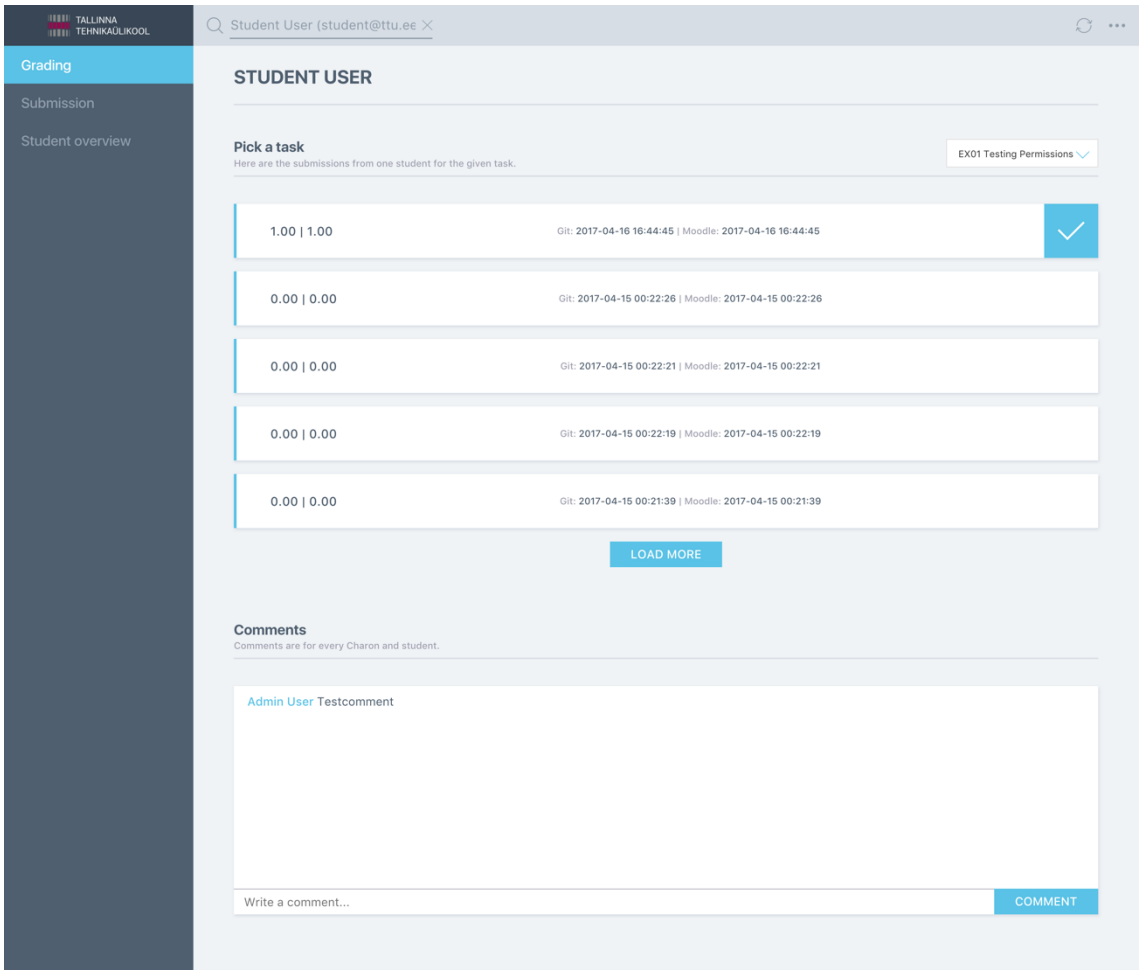
Lisa 2 - Ülesande loomise minimaalne vorm



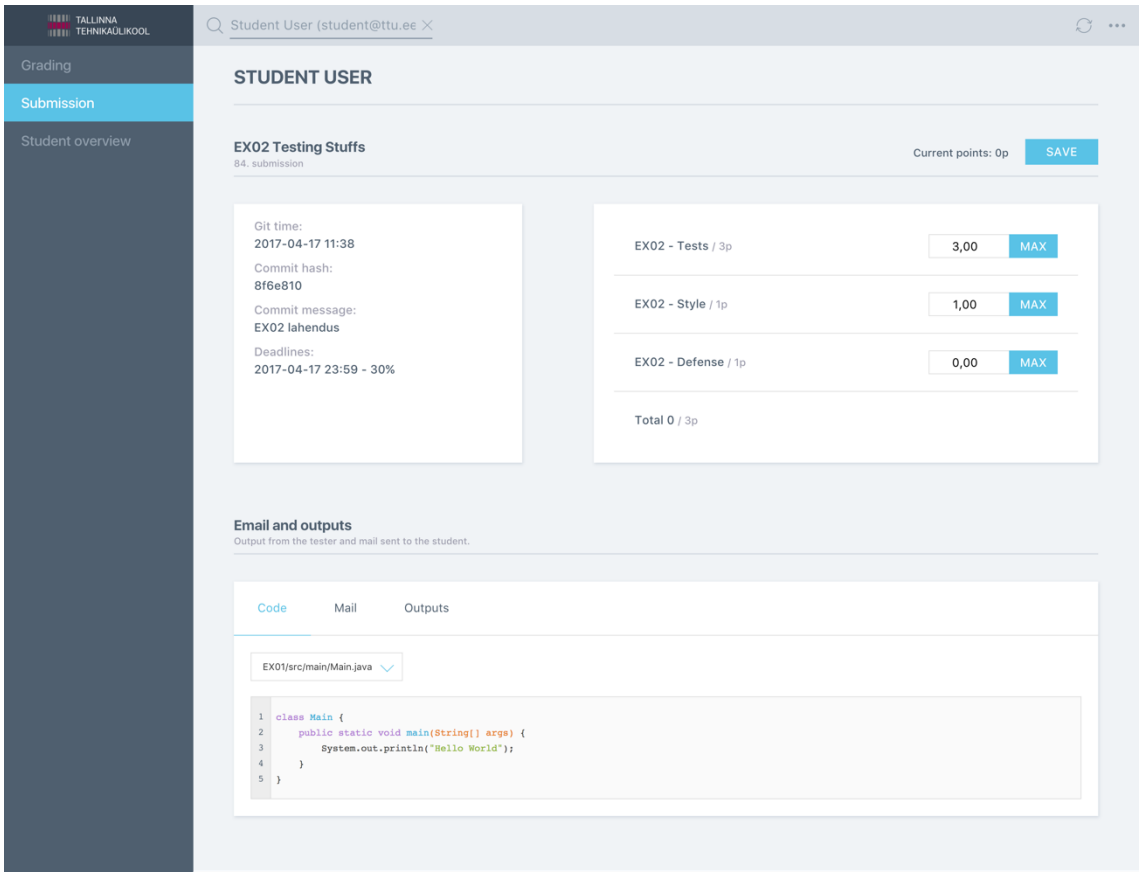
Lisa 3 – Ülesande loomise detailne vorm



Lisa 4 – Hindamise liidese esituste vaade



Lisa 5 – Hindamise liidese ühe esituse vaade



Lisa 6 - Alla laetavate moodulite konfiguratsioon

```
{
    "gits": [
        {
            "url": "git@git.ttu.ee:devel/ained/moodle/capella.git",
            "branch_develop": "devel",
            "branch_production": "master",
            "type": "mod",
            "destination_directory": "capella",
            "update_script": "cd mod/capella && git pull && php composer.phar
install && cd ../../"
        },
        {
            "url": "git@git.ttu.ee:devel/ained/moodle/charon.git",
            "branch_develop": "develop",
            "branch_production": "master",
            "type": "mod",
            "destination_directory": "charon",
            "update_script": "cd mod/charon && git pull && php composer.phar
install && cd ../../"
        },
        {
            "url": "git@git.ttu.ee:devel/ained/moodle/theme.git",
            "branch_develop": "devel",
            "branch_production": "master",
            "type": "theme",
            "destination_directory": "snap",
            "update_script": "cd theme/snap && git pull && cd ../../"
        },
        {
            "url": "https://github.com/Microsoft/moodle-auth_oidc.git",
            "branch_develop": "MOODLE_32_STABLE",
            "branch_production": "MOODLE_32_STABLE",
            "type": "auth",
            "destination_directory": "oidc",
            "update_script": ""
        },
        {
            "url": "https://github.com/marinaglancy/moodle-
mod_groupselect.git",
            "branch_develop": "master",
            "branch_production": "master",
            "type": "mod",
            "destination_directory": "groupselect",
            "update_script": ""
        },
        {
            "url": "https://github.com/bostelm/moodle-mod_scheduler.git",
            "branch_develop": "master",
```

```
            "branch_production": "master",
            "type": "mod",
            "destination_directory": "scheduler",
            "update_script": ""
        }
    ]
}
```