TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Informaatikainstituut

IDK40LT

Valentin Djomin 134697IAPB

KOLMEKIHILISE VEEBIRAKENDUSE ARHITEKTUURI ANALÜÜS NING ARENDAMINE KANDIDAADI HALDUSSÜSTEEMI JAOKS

Bakalaureusetöö

Juhendaja: Jekaterina Tsukrejeva
Magistrikraad
Assistent

Tallinn 2016

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: Valentin Djomin

23.05.2016

Annotatsioon

Antud töö eesmärk oli uurida veebirakenduste kolmekihilist arhitektuuri, ehitada arhitektuuri prototüüp ning arendada veebirakendus kasutades ehitatud prototüüpi. Kolmekihilist arhitektuuri kasutati kandidaadi haldussüsteemis, mis oli mõeldud värbamisfirmadele ning ka erinevate ettevõtete kaadriosakonnale.

Autor näitas antud töös, et arhitektuuri prototüübi kasutamisele õige lähenemine vähendab üldsüsteemi keerukust. Antud lähenemine aitab struktureerida rakendust sõltumata erinevate andmete, klasside ja pakettide struktuuride arvust.

Töö tulemusena arendati kolmekihilise arhetiktuuri prototüüp ja realiseeriti prototüübi alusel veebirakendus. Veebirakenduse realiseerimine tehti Java programmeerimiskeeles, sobivate raamistike kasutamisel ning ka „Clean Code" mõistete ja OOP printsiibi kasutamisel.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 58 leheküljel, 5 peatükki, 22 joonist, 5 tabelit.

Abstract

Analysis and Implementation of a Three-tier Architecture for the Candidate Management System

The objective of this thesis was to analyze a three-tier architecture and construct a prototype of this architecture to be used in the Candidate Management System, an application designed for recruitment companies and HR departments of different companies.

It was shown that the right approach to the architecture prototype reduces the complexity of the general system. This approach helps structure an application independently of the amount of data, classes and packages. A connection was established between system requirements and system architecture. As a result of this thesis, a three-tier architecture prototype was developed, on which the web application will be based.

The Candidate Management System will be developed using the Java 8 programming language with appropriate libraries and frameworks. To ensure code quality, the principles of "Clean Code" and object-oriented programming were employed.

The thesis is written in Estonian and contains 58 pages of text, 5 chapters, 22 figures, 5 tables.

Lühendite ja mõistete sõnastik

IEEE SA	***Institute of Electrical and Electronics Engineers Standards Association*** Tehnika valdkonna spetsialistide rahvusvaheline mittetulundusassotsiatsioon. Rahvusvaheline liider standardite arendamise valdkonnas.
SQL	***Structured Query Language*** Struktureeritud päringukeel. Kasutatakse andmete loomiseks, modifitseerimiseks ja haldamiseks suvalises relatsiooni andmebaasis.
OOP	***Object Oriented Programming*** Programmeerimise metodoloogia, milles programm on esitatud objektide ja nende omavaheliste sidemete kujul.
TWC	***Trustworthy Computing*** Turvalised informatsioonisüsteemid. Kasutatakse turvaliste, kättesaadavate ja ohutute süsteemide puhul.
SDLC	***System Development Life Cycle*** Tarkvara elutsükkel.
ISO	***International Organization for Standardization*** Rahvusvaheline organisatsioon, kes tegeleb standardite aktsepteerimisega kõikideks valdkondadeks.
OSI	***Open Systems Interconnection model*** Avatud süsteemide koostöö baasmudel, mis kirjeldab võrgu arhitektuuri erinevate seadmetega.

CRC	***Class-Responsibility-Collaborator*** Klass-vastutus-kooperatsioon. CRC-kaardid on mõeldud tarkvara objektorienteeritud projekteerimiseks. Vähendavad disaini keerulisust.
QA	***Quality Assurance*** Kvaliteedi tagamine. Toote vajalike omaduste ja karakteristikute kujundamise protsess ning kvaliteedi kontroll.
HTML	***HyperText Markup Language*** Veebidokumentide märgistuse standartiseeritud keel.
CSS	***Cascading Style Sheets*** Kaskaadlaadistik. Dokumendi välisvaate kirjeldamise formaalne keel, mis on kirjutatud märgistuse keele kasutamisel.
TDD	***Test Driven Development*** Arendamine testimise kaudu. Alguses kirjutatakse tekst, seejärel kirjutatakse funktsionaal, mis lubab läbida testi.
MVC	***Model-View-Controller*** Projekteerimismuster liidese realiseerimiseks kasutaja arvutis.
DTO	***Data Transfer Object*** Andme edastamise objekt. Projekteerimismuster, mida kasutatakse andmete edastamiseks rakenduse allsüsteemide vahel.
POJO	***Plain Old Java Object*** Lihtne vana Java objekt. Ei ole tuletatud teisest objektist ja ei realiseeri mingeid ametiliideseid.
JPA	***Java Persistence API*** API, mis annab võimaluse salvestada Java objekte mugaval kujul andmebaasis.

Sisukord

1 Sissejuhatus 11
1.1 Taust ja probleem 11
1.2 Ülesande püstitus 12
1.3 Metoodika 12
1.4 Ülevaade tööst 12
2 Tarkvara arhitektuur 14
2.1 Arhitektuuri mõiste 14
2.2 Arhitektuuri tekkimine 15
2.3 Tarkvara arhitektuuri eesmärk ja vajalikkus 16
2.4 Tarkvara kvaliteet 16
2.5 Tarkvara mitmekihiline arhitektuur 17
2.6 Tarkvara kolmekihiline arhitektuur 20
2.6.1 Esitluskiht 21
2.6.2 Äriloogikakiht 21
2.6.3 Andmekiht 21
2.7 Testimine 22
3 Arhitektuuri prototüübi ehitamine 23
3.1 Ainevaldkonna ülevaade 23
3.2 Nõudmised süsteemile 24
3.3 Kasutusklassiskeem ja süsteemi soovilood 25
3.4 Arhitektuuri komponentide määramine 31
3.4.1 Esitluskiht 31
3.4.2 Äriloogikakiht 32
3.4.3 Andmekiht 35
4 Prototüübi realiseerimine 36
4.1 Rakenduse moodulite realiseerimine 36
4.2 Moodulite struktuuri realiseerimine 38
4.3 Liideste ja abstraksete klasside loomine 39

4.4 Mustrid ... 41
4.4.1 Tootmismuster ... 42
4.4.2 Ehitusmuster ... 43
4.4.3 Singleton muster ... 43
4.4.4 MVC muster ... 44
4.4.5 DTO muster ... 45
5 Kokkuvõte ... 46
Kasutatud kirjandus ... 48
Lisa 1. Rakenduse moodulite realiseerimine ... 50
Lisa 2. Kandidaadi haldussüsteemi mooduli struktuur ... 51
Lisa 3. Kandidaadi haldussüsteemi tuletamise lähtekood ... 52
Lisa 4. Tootmismustri lähtekood ... 53
Lisa 5. Ehitusmustri lähtekood ... 55
Lisa 6. HomeController'i lähtekood. ... 57
Lisa 7. DTO mustri lähtekood ... 58

Jooniste loetelu

Joonis 1. OSI 7-kihiline mudel. 18
Joonis 2. Üksikkihi CRC diagramm. 19
Joonis 3. Abstraktsiooni tasemed mitmekihilises arhitektuuris. 20
Joonis 4. Kolmekihiline arhitektuur. 21
Joonis 5. Kandidaadi haldussüsteemi kasutusklassiskeem. 26
Joonis 6. Esitluskihi komponendid. 32
Joonis 7. Äriloogikakiht ja veatöötlus. 33
Joonis 8. Äriloogikakiht ja logimine. 33
Joonis 9. Äriloogikakiht ja testimine. 34
Joonis 10. Kandidaadi haldussüsteemi juurkausta struktuur. 37
Joonis 11. Allprojektide liitumine juurprojektiga. 37
Joonis 12. Kolmekihiline arhitektuur moodulite sõltuvusega. 37
Joonis 13. Sõltuvus web- ja core-mooduli vahel. 38
Joonis 14. Sõltuvus core- ja persistence-mooduli vahel. 38
Joonis 15. LiveDataSourceProperties klassi sisu. 40
Joonis 16. TestDataSourceProperties klassi sisu. 40
Joonis 17. DataSourceProperties klassi sisu. 40
Joonis 18. Klasside tuletamine ja sõltuvus. 41
Joonis 19. JpaRepository liidese tuletamine. 41
Joonis 20. Kandidaadi haldussüsteemi tootmismustri skeem. 42
Joonis 21. Tüüpiline klass ja Singleton muster. 43
Joonis 22. MVC andmevoo diagramm. 44

Tabelite loetelu

Tabel 1. Kandidaadi haldamine süsteemis. 27

Tabel 2. Kliendi haldamine süsteemis. 28

Tabel 3. Konkursi haldamine süsteemis. 29

Tabel 4. Kandidaadi kandideerimisprotsessis haldamine. 30

Tabel 5. Kandidaadi haldussüsteemi moodulite struktuur 39

1 Sissejuhatus

Arhitektuur on programmi luustik, mis võimaldab struktureerida süsteemi komponente ja ühendada neid omavahel. Käesoleval ajal arendatakse suuri mahukaid süsteeme, mis on väga keerulised. Arhitektuur loob tarkvara mustandvisandi ja ilma vastava analüüsita ei saa süsteemi mõista. Nõudmised annavad võtme panuse programmi arhitektuuri ja samas võib tekkida vajadus korrigeerida või selgitada nõudmisi, et saavutada kindlat arhitektuuri. Tarkvara arhitektuuri ei ole võimalik kirjeldada mingi ühe diagrammiga, sest arhitektuur on küllalt lai mõiste. Tarkvara arhitektuuri valdkonna põhimõisted on dekompositsioon, mitmekihiline esitlus, mustrid ja komponentprogrammeerimine. Need on pigem riistad, mida arhitekt saab kasutada enda äranägemisel, kui tarkvara elemendid, mida tuleb kindlasti kasutada. Programmi arhitektuuri põhiliseks eesmärgiks on keerukuse vähendamine.

1.1 Taust ja probleem

Kui programmi arhitektuur ei ole struktureeritud, kulutavad programmeerijad suurt osa oma ajast sellele, et saada aru programmi lähtekoodist. Selliste süsteemide arendamine, toetamine ja haldamine läheb firmadele kallilt maksma. Et vältida probleemi programmi toetusega tulevikus ja vähendada kulusid selle haldamisele ja arendamisele, tuleb alustada programmi arendamist hästi läbimõeldud arhitektuurist. Selle töö põhiline kasulikkus seisneb selles, et see annab lugejale üldise ettekujutuse arhitektuurist.

Antud töö on vajalik arendajatele, testijatele, QA-mänedzeritele, arhitektidele ja teistele, kes on seotud programmi arendamisega. Töös näidatakse seost nõudmiste tootele ja arhitektuuri vahel, ehitatakse arhitektuuri prototüüp. Arhitektuuri prototüüp sisaldab programmi komponentide määramist ja annab ülevaate projekteerimise mustritest, mida hakatakse kasutama arhitektuuri realiseerimisel.

Arhitektuuri arendatakse kandidaadi haldussüsteemi jaoks, mille tellijaks on ettevõte Ignite OÜ. Süsteemi lõppkasutajaks on värbamisfirmad ja suurte ettevõtete personaliosakonnad. Autori osa antud rakenduse arendamises on taustarendaja. Kandidaadi haldussüsteem on arendatud alates veebruarist 2016. a ja selle arendamine jätkub.

1.2 Ülesande püstitus

Töös on kolm põhieesmärki. Töö üks eesmärk on uurida tarkvara mitmekihilist arhitektuuri. Teine eesmärk on ehitada kandidaadi haldussüsteemi arhitektuuri prototüüp. Kolmas eesmärk on ehitatud prototüübi alusel realiseerida kandidaadi haldussüsteemi arhitektuur.

1.3 Metoodika

Autori poolt püstitatud eesmärkide saavutamiseks koostati arhitektuuri teoreetiline alus, kuhu kuulub tarkvara arhitektuuri mõiste ja ajalugu, tehti mitmekihilise arhitektuuri ülevaade. Struktureeriti nõudmised süsteemile, et nende alusel võiks määrata arhitektuuri komponendid ja ehitada prototüüp. Arhitektuuri prototüüp realiseeriti kasutades projekteerimise mustreid ja „Clean Code“ põhimõtteid kandidaadi haldussüsteemi jaoks Java programmeerimiskeeles.

1.4 Ülevaade tööst

Bakalaureusetöös on kolm peatükki. Esimeses peatükis käsitletakse tarkvara arhitektuuri teoreetilisi aspekte. Antakse arhitektuuri mõiste kinnitatud standardi alusel, kirjeldatakse tarkvara mitmekihilise arhitektuuri ajalugu, selle eesmärki, vajalikkust ja tüüpe. Autor annab tarkvara kolmekihilise arhitektuuri detailset ülevaadet iga kihi kirjeldusega. Teises peatükis tehakse ainevaldkonna ülevaadet sellest, milleks hakatakse realiseerima arhitektuuri, määratakse nõudmised süsteemile. Määratakse arhitektuuri komponendid ja projekteerimise mustrid, mida hakatakse ka kasutama arhitektuuri realiseerimisel. Kolmandas peatükis realiseerib autor programmiliselt arhitektuuri prototüübi kasutades projekteerimise mustreid. Kirjeldatakse iga komponendi saadud struktuur.

Programmi kolmekihist arhitektuuri arendatakse reaalse projekti jaoks – kandidaadi haldussüsteemi jaoks. Süsteemi kasutajateks on värbamisagentuuride ja suurte ettevõtete personaliosakondade spetsialistid. Programmi arendamises osaleb arendajate meeskond: vanemarendaja Andrei Sljusar, taustarendaja Valentin Djomin, arendaja Valeriia Shpychka ja QA-mänadzer Seda Sahradyan. Rakenduse tellija ja omanik on ettevõte Ignite OÜ.

2 Tarkvara arhitektuur

Peatükis käsitletakse tarkvara arhitektuuri teoreetilisi aspekte: mõiste, teke, eesmärk, vajadus, tüübid. Tuuakse tarkvara kolmekihilise arhitektuuri mudel ja vaadeldakse igat kihti tarkvara kolmekihilises struktuuris.

2.1 Arhitektuuri mõiste

Vastavalt IEEE 1417 SA vastuvõetud IEEE 1417 standardile on arhitektuur süsteemi baasstruktuur, mis on teostatud selle komponentides, nende suhted omavahel ja ümbruskonnaga ning süsteemi projekteerimist ja arendamist määravad põhimõtted. Selles standardis on välja toodud järgmised arhitektuuriga seotud mõisted:

- Süsteem on kindla funktsiooni või funktsioonide komplekti täitmiseks ühendatud komponentide komplekt. Termin süsteem hõlmab üksikrakenduste süsteeme traditsioonilises mõttes, allsüsteeme, süsteemide süsteeme, produkte, produktide sarju ja teisi objekte. Süsteem on mõeldud ühe või mitme missiooni täitmiseks oma ümbruskonnas.

- Süsteemi ümbrus või kontekst – ümbrus võib ühendada endas teisi süsteeme või komponente, mis töötavad koos huvisüsteemiga nii liideste kaudu, kui ka kaudselt teiste koostöö viiside kaudu. Ümbrus määrab huvisüsteemi valdkonna raame teiste süsteemide suhtes.

- Missioon – see on rakendus või tegevus, mille jaoks üks või mitu huvitatud isikut kavatsevad kasutada süsteemi vastavalt mõnele tingimuste kogumile.

- Huvitatud isik on füüsiline isik, rühm või organisatsioon (või selle kategooriad), kes on huvitatud süsteemi funktionaalsusest.

IEEE 1471 standardi spetsifikatsioonis kasutatakse sageli terminit „komponent“. Kuigi suurem osa arhitektuuri mõistetest ei määra terminit „komponent“. Seda on tehtud

meelega, et mõiste komponent vastaks hulgale võimalikest tõlgendustest: objektid, programmipaketid, programmimoodulid, allsüsteemid, süsteemi süsteemid jm. [1]

2.2 Arhitektuuri tekkimine

Arvutiteaduste valdkonna tekke momendist on hakanud ilmuma probleemid, mis on seotud programmi süsteemide keerukusega. Varem lahendati neid probleeme enda andmete struktuuri ja algoritmide juurutamise ning volituste piiretlemise metodoloogia kasutamise teel.

Termin „tarkvara arhitektuuri kihid“ hakkas ilmuma klient-serveri süsteemi levikuga 90-ndatel aastatel. Need olid kahekihilised süsteemid: klient vastutas kasutaja liideste ja rakenduse koodi eest, aga server oli relatsiooni andmebaas. VB, Powerbuilder ja Delphi olid kõige levinumad kliendi vahendid, mis võimaldasid kergelt ehitada andmemahukaid rakendusi, sest neil olid SQL tundvad kasutajaliideste vidinad. Järelikult, võis ehitada kasutajaliideseid lohistades kontrollereid disaini alale ja seejärel, kasutades kontrollerite omadusi, liita andmebaasiga.

Kui rakendus on loodud selleks, et kuvada ja muuta andmeid relatsiooni andmebaasis, siis sel juhul töötavad klient-server väga hästi. Probleem tekib domeenide loogikas: ärireeglid, valideerimised, kalkulatsioonid ja teised. Tavaliselt realiseeritakse domeenide loogikat kliendi peal, kuid see on ebamugav ja harilikult on tehtud loogika juurutamise teel otseselt kasutajaliidestesse. Seetõttu, et domeenide loogika on üsna keeruline, muutub kood programmeerijatele arusaamiseks raskeks. Juurutades loogikat kasutajaliidestesse, suureneb koodi dubleering, mille tõttu viib väike muudatus analoogse koodi otsingule paljudes kasutajaliidestes.

Alternatiivina võis domeeni loogikat välja tuua andmebaasi, salvestatud protseduuridena. Kuid salvestatud protseduurid piirasid struktureerimise mehhanismi, mis jällegi viis ebamugavale koodile. Samuti meeldib paljudele inimestele kasutada relatsiooni andmebaase, sest nendes kasutatakse SQL standardit. SQL standard võimaldab kergesti vahetada andmebaasi.

Tol ajal, kui klient-server süsteemid kogusid populaarsust, hakkas ilmuma objektorienteeritud programmeerimise maailm. OOP ühisus leidis domeeni loogika probleemi lahenduse: üleminek kolmekihilisele süsteemile. Sellises lähenemises on

olemas kuvamise kiht kliendiliidese jaoks, domeeni kiht domeeni loogika jaoks ja kolmas kiht on andmeallikas (andmebaas). Sel juhul saab programmeerija domeeni loogikat välja tuua kasutajaliidestest domeenikihti, kus luuakse sobivaid objekte ja loogika muutub struktureerituks. [2]

2.3 Tarkvara arhitektuuri eesmärk ja vajalikkus

Keeruliste arvutisüsteemide (näiteks, kaubandussüsteemi) arhitektuuri arendamine võtab palju aega. Hea ettevõtluse arhitektuuri üks tähtsamaid aspekte on võimalus toetada paindlikkust ja tõhusust arendamises, mis peab viima innovatiivse lahenduse realiseerimisele turu jaoks. Väga tähtis on jääda konkurentsivõimeliseks kiiresti muutuva turu taustal, seetõttu põhiakstendid sisaldavad võimalust:

- arendada uusi ja olemasolevaid teenuseid lühikese ajaga;
- rahuldada individuaalselt klientide nõudmisi;
- garanteerida kliendile esimesest kohtumisest, et tema nõudmised täidetakse;
- toetada koodi arendamise võimalust erinevate ettevõtte osakondade poolt. [3]

2.4 Tarkvara kvaliteet

Tarkvara kvaliteet erineb sõltuvalt seisukohast. Arendaja ja kasutaja seisukoht oluliste programmi karakteristikute suhtes erineb.

Kasutaja seisukohast on kõige olulisemad järgmised tarkvara omadused :

- Töövõime. Tarkvara peab täitma seda, mida sellelt nõutakse.
- Sõbralik kasutajaliides. Programmi kasutamine peab olema mugav ja programm peab töötama minimaalse koormusega kasutaja arvutisüsteemile.
- Turvalisus, konfidentsiaalsete andmete kaitse. Programm peab turvaliselt ja stabiilselt töötama ja tagama nii programmi, kui ka selle poolt kasutatavate loodavate ja töödeldavate andmete kaitset võimalikke välisohtude ja rünnakute eest. [4]

Arendaja seisukohast on kõige tähtsamad järgmised programmi omadused:

- Moodulite korduv kasutatavus. Kuna kasutatakse komponent orienteeritud programmeerimist, ei ole vaja programmeerida ühetüübilist koodi korduvalt, selle asemel tuleb kasutada igal arengu etapil juba valmis ja kontrollitud koodi, sest sarnase koodi dubleerimine võib halvendada programmi turvalisust.

- Moodulsus. Moodulsus on programmi arendamine omavahel seotud terviku kujul sõltumatute programmi komponentide moodulite suhtes, igaüks neist lahendab oma kindlat ülesannet.

- Efektiivsus. Efektiivsus on programmi optimaalsus vastavalt sellele või teisele kriteeriumile (näiteks, minimaalne täitmise aeg).

- Ülekantavus. Ülekantavus on programmi ülekandmise võimalus selle koodi muutmata teisele platvormile ja selle järgnev kasutamine teisel platvormil mingite muutmisteta.

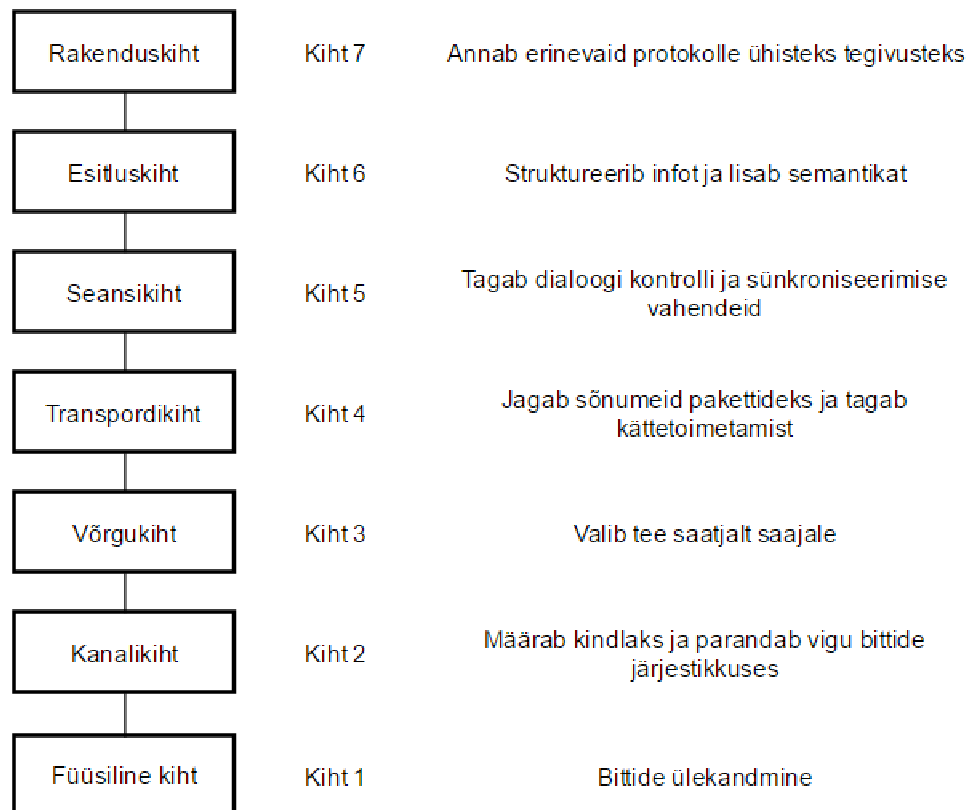
- Mõistmine. Mõistmine on programmi koodi selgus, arusaaduvus, isedokumenteeritavus, ühtsete standardite kasutamine selle kujundamisel, mis soodustavad järgnevat koodi õppimist ja muutmist (reeglina teiste programmeerijate poolt), programmis vigade parandamise eesmärgil või selle funktionaalsuse laiendamiseks.

- TWC ja SDLC põhimõtete rakendamine projekteerimisel ja realiseerimisel. On vaja arvestada turvalisuse ja töökindluse nõudeid alates programmi arendamise tsükli kõige varasematest etappidest ja selle arendamise kõigil etappidel, muidu tekivad tõsised turvalisuse probleemid programmi ekspualteerimisel, mis lõppkokkuvõttes võib viia programmi edasise kasutamise võimatuseni. [4] [5]

2.5 Tarkvara mitmekihiline arhitektuur

Mitmekihilise arhitektuuri kõige tuntum näide on võrguprotokollid. Võrguprotokollid sisaldavad reeglite ja kokkulepete kogumeid, mis kirjeldavad, kuidas arvutiprogrammid vahetavad omavahel andmeid masina raames. Kõigi sõnumite, formaat, sisu ja tähendus on määratud. Kõik stsenaariumid on detailseid kirjeldatud, tavaliselt järjestikkuse

skeemina. Protokollid määravad kokkulepped abstraktsiooni kihtide vahel, alustades bittide ülekandmise detailidest ja lõpetades kõrgetasemelise programmi loogikaga. Seetõttu kasutavad arhitektuuri disainerid allprotokolle ja organiseerivad neid kihtideks. Igal kihil on tegemist kindla kommunikatsiooni aspektiga ja iga kiht kasutab alljärgneva kihi teenuseid. Rahvusvaheline Standardimisorganisatsioon (ISO) määrab järgneva arhitektuuri mudeli – OSI 7-kihiline mudel:



Joonis 1. OSI 7-kihiline mudel.

Mitmekihilise arhitektuuri ehitamise lähenemist peetakse paremaks praktikaks, kui protokolli implementatsiooni monoliitplokina, sest mitmekihiline arhitektuur aitab arendada tarkvara käskude järgi, toetab koodi suurenevat hulka ja toetab süsteemi testimist kihtide järgi. [6] [7]

Ettevõtluse arendamises luuakse suuri süsteeme, mis nõuavad dekomposeerimist. Dekomposeerimine on selline lähenemine, mis võimaldab asendada ühe suure ülesande lahendamist väikeste omavahel seotud, kuid lihtsamate ülesannete seeriaga. Kõrge kihi seisukohalt on lahendus piisavalt lihtne. Süsteemi tuleb struktureerida kihtide sobivaks arvuks ja panna neid üks teise kohale. Kihtide loendust alustatakse abstraktsiooni kõige

väiksemast kihist (Kiht 1), see kiht on süsteemi baasosa. Abstraktsiooni kiht suureneb iga kihiga (Kiht J), mida pannakse eelmisele (Kiht J - 1). Skeemi tipul on funktsionaalsuse kõrgeim kiht (Kiht N).

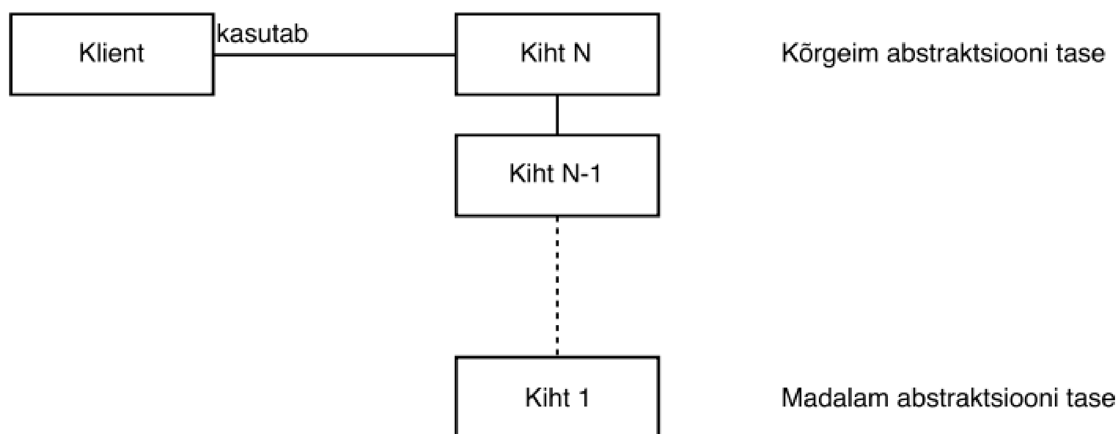
Tuleb märkida, et see ei eelda reaalsete disainikihtide järjekorda, see annab vaid süsteemi kontseptuaalse kuju. See ka ei tähenda seda, et Kiht J üksikkiht peab olema keeruline allsüsteem, mida on vaja dekomposeerida ja ka seda, et see Kiht J peab edastama päringuid Kiht J+1'lt Kiht J-1'le. Kuid see on ikkagi oluline, et konkreetne kiht sisaldab põhikomponente, mis töötavad ühel abstraktsiooni kihil.

Enamus teenuseid, mida pakub Kiht J, on eelmise Kiht J-1 kihi teenuste koosseisus. Teiste sõnadega, iga kihi teenused realiseerivad strateegiat alumise kihi teenuste mõtestatud kombineerimiseks. Kihi J teenused võivad olla seotud Kihi J teenustega. Üksikkiht võib olla kirjeldatud järgmise CRC diagrammiga:

Klass Kiht J	**Koostööline** • Kiht J-1
Vastutus • Kasutab teenuseid, mida kasutab Kiht J+1. • Deligeerib alamülesandeid Kihile J-1.	

Joonis 2. Üksikkihi CRC diagramm.

Mitmekihilise arhitektuuri põhiline struktuurne omadus on see, et Kiht J kihi teenuseid kasutatakse ainult Kiht J+1 kihis, puudub otsene sõltuvus kõrgematest kihtidest. Antud struktuuri saab võrrelda pinumäluga. Iga üksik kiht kaitseb kõiki alumisi kihte kõrgemate kihtide otsese juurdepääsu eest. [6] [8]



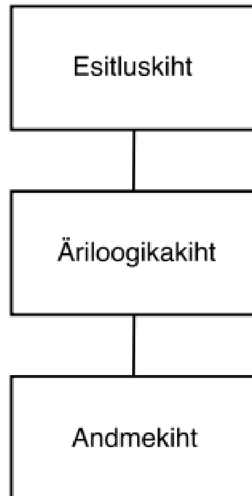
Joonis 3. Abstraktsiooni tasemed mitmekihilises arhitektuuris.

2.6 Tarkvara kolmekihiline arhitektuur

Tarkvara kolmekihiline arhitektuur on programmiarhitektuur, mis eeldab kolme komponendi olemasolu selles: esitluskiht, äriloogikakiht ja andmekiht. Kolmekihilise arhitektuuri skeem on välja toodud Joonisel 4.

Kolmekihilise arhitektuuri põhiplussideks: koodi kõrge struktureeritus, mis aitab vältida koodi spaghetti ja lihtsustab koodi mõistmist; suurem skaleerimine, st server rakenduste horizontaalse skaleerimise arvelt; suur konfigureeritavus, st toimub kihtide isoleeritavuse teel üks teisest; esitluskiht võib salvestada vahemälusse päringuid, mille tõttu väheneb andme- ja äriloogikakihtide koormus; kättesaadavus, st esitluskiht saab töödelda veebipäringuid kasutades vahemälu isegi kui server on väljalülitatud; muutmised kasutajaliidestes toimuvad kiiremini, st väga lihtne on muuta esitluskihti, sest see ei olene äriloogika- ja andmekihtidest; ohutum juurdepääs andmebaasile, st kliendil ei ole otsest ligipääsu andmebaasile. [5] [9] [10]

Kolmekihilise arhitektuuri põhimiinusteks: ehitada prototüüpi nullist on keerulisem; on võimalik koodi dubleerimine. [9]



Joonis 4. Kolmekihiline arhitektuur.

2.6.1 Esitluskiht

Esitluskiht aitab hallata koostööd kasutaja ja tarkvara vahel. See võib olla nii lihtne käsurida, kui ka graafiliste elementide rikas kasutajaliides. Esitluskihi põhikohustuseks on informatsiooni kuvamine kasutaja jaoks ja kasutaja käskude interpretatsioon äriloogika ja andmete kihtide toimingutes. [2]

2.6.2 Äriloogikakiht

Äriloogikat nimetatakse tihti terminitega „ärireeglid“ või „domeeniloogika“. See kiht sisaldab sisestatud või salvestatud informatsioonil põhinevaid arvutusi, andmete validatsiooni, mis tulevad esitluskihist ja määramist, millist informatsiooni tuleb võtta andmebaasist sõltuvalt saadud käskudest. Mõnikord on kihid organiseeritud selliselt, et äriloogikakiht varjab täielikult esitluskihi andemetekihti. Kuid praktikas töötab kõige paremini lähenemine, kui esitluskiht toimib otseselt koos andmekihiga. Sel juhul interpreteerib kasutaja käsku vajalikku informatsiooni lugemiseks andmebaasist ja pärast seda, kui vastavad andmed on maha loetud, äriloogikakiht töötleb neid. [2]

2.6.3 Andmekiht

Andmekiht võimaldab rakendusel suhelda teiste süsteemidega ning käivitada erinevaid ülesandeid rakenduse nimel. Teiste süsteemide rollis võivad olla transaktsioonide monitooring, teised rakendused, sõnumite vahetuse süsteemid jm. Andmebaas on

andmekihi põhiline osa enamuse ärirakenduste jaoks. Seda kasutatakse kõigepealt püsiandmete salvestamiseks. [2]

2.7 Testimine

Kui rühm arendajaid jõuab ühisele arvamusele funktionaalse spetsifikatsiooni valmidusest, siis sel juhul nimetatakse seda „lõpetatuks“ või „allakirjutatuks“. Peale seda kirjutavad programmeerijad ja testijad lähtekoodi ja testivad programmi, kasutades funktsionaalset spetsifikatsiooni etalonina. Testimise ajal tehakse programmi tegevuste võrdlust oodatutega, mis on määratud spetsifikatsioonis [11]. Loogika testimiseks kandidaadi haldussüsteemis kasutatakse ühikteste. Ühiktestide abil saab kontrollida programmi funktsionaali töö õigsust. Realiseeritud arhitektuuri testimine on küllalt suur osa, milles peavad olema rahuldatud funktsionaalsed nõudmised süsteemile. Arhitektuur peab olema testitud mitte funktsionaalsete nõudmiste suhtes: süsteemi kättesaadavuse testimine kindlal ajavahemikul; vigade töötluse testimine; süsteemi turvalisuse testimine; rakenduse mitmevoogulisuse testimine.

Antud bakalaureusetöö ei sisalda kandidaadi haldussüsteemi arhitektuuri testimist. Arhitektuuri testimine on üsna mahukas osa ja seda saab käsitleda edaspidise uurimisena.

3 Arhitektuuri prototüübi ehitamine

Peatükis on kirjedatud süsteemi rakendamise valdkonna, käsitletakse nõudeid produktile. Nõuete põhjal süsteemile teostatakse arhitektuuri prototüübi ehitamist. Prototüüp sisaldab mooduliteks jagamise süsteemi, moodulite süsteemi ja üksikosade funktsionaalsuse kirjeldust.

3.1 Ainevaldkonna ülevaade

Tänapäeval on kogu maailmas populaarsed nn värbamisettevõtted, kelle põhiülesanneteks on personali otsing, hindamine ja valik tööandjate tellimusel. Kandidaadi õige valik ametikohale võimaldab suurendada tööviljakust kasumit ning tõsta ettevõtte poolt pakutavate teenuste kvaliteeti. Personali valik algab ameti, millele töötaja hakkab kandideerima, määramisest ja kirjeldamisest. Samuti on väga tähtis määrata nõudeid kandidaadile ja nõutavat kogemust. Pärast ameti kinnitamist algab kandidaatide otsimise etapp.

Värbamisettevõtetel on oma kandidaatide sisebaas, mis töökäigus täieneb. Kui oma kandidaatide baasis ei ole sobivaid kandidaate, siis värbaja hakkab teostama välisotsingut: otsing sotsiaalvõrkudes, trükimeedia kaudu, kõrgkoolide vilistlaste ja vanemate kursuste tudengite kaasamine ja muu. Kõik leitud kandidaadid sisestatakse värbamisettevõtte sisebaasi selleks, et tulevikus otsida kliente selle kaudu [12] [13].

Kandideerimisprotsess sisaldab etappe ja faase, mille põhjal tehakse otsus tööle võtmise kohta. Faasid erinevad sõltuvalt ametist, kuid kõige levinumad on: esimene kontakt, vestlus, testimine ja tööpakkumine. Tuleb märkida, et kandidaat võib olla tööle võetud kohe peale esimest kontakti. Sel juhul teda aktsepteeritakse antud ametikohale ja pakutakse töö. Kandidaat võib mitte nõustuda lepingu tingimustega ja algab läbirääkimiste etapp. Tööleping sõlmitakse sel juhul, kui läbirääkimised olid edukad ja pooled jõudsid kokkuleppele. Kui aga peale suhtlemist kandidaatiga on raske teha otsus, siis kandideerimise protsess jätkub. Otsus selle kohta, et kandidaat ei sobi ametikohale

tehakse läbitud faaside analüüsi põhjal. Sel juhul kandidaati informeeritakse sellest, et ta ei läbinud kandideerimist ja võidakse pakkuda sobivamat ametikohta.

3.2 Nõudmised süsteemile

Süsteemi arhitektuuri projekteerimise staadiumil kasutatakse nõudmiste kogumit, mille põhjal ehitatakse süsteemi. Nõudmine süsteemile on toote kasutamise vajaliku funktsionaalsuse või erilisuse kirjeldus. Sagedamini saabuvad nõudmised struktureerimata kujul, mis seejärel kujunevad ümber nõudmisteks tootele või süsteemseteks nõudmisteks ja täienevad puuduvate andmetega, sealhulgas funktsionaalsete ja miitefunktsionaalsete aspektiga nõudmiste arendamisel. Nõudmised jagunevad funktsionaalseteks ja mittefunktsionaalseteks. Funktsionaalsed nõudmised kirjaldavad süsteemi funktsionaalsust äri seisukohalt; kirjeldavad süsteemi kasutajate essmärke ja ülesandeid, mida peavad täitma kasutajad loodava programmi süsteemi abil. Mittefunktsionaalsed nõudmised kirjeldavad erinevaid süsteemi käitumise või ekspluatatsiooni erilisusi, näiteks nõudmisi kasutamise mugavusele, turvalisusele, tootlikkusele jms. Selleks, et nõudmised süsteemile oleks arusaadavad kõigile osapooltele (toote omanikule, programmeerijale, süsteemi arhitektile jt.), määratakse terminoloogia [14]. Kandidaad haldussüsteemis on mõisted: värbaja, klient, konkurss, kandidaat. Värbaja on süsteemi kasutaja, kes teostab toimunguid süsteemis: vaatab läbi, lisab, muudab ja kustutab infot. Klient on juriidiline isik (firma või ettevõte), kelle jaoks teostatakse kandidaatide otsingut määratud töökohale. Konkurss on projekt, mis määrab töö, milleks valitakse kandidaate. Konkursil on määratud alguse kuupäev, lõpu kuupäev ja ameti kirjeldus. Kandidaat on füüsiline isik, keda käsitletakse võimaliku töötajana kindlale ametikohale. Kandidaadil on inimese isiklik informatsioon.

Mõisteid kasutatakse nõudmiste kindlaks tegemiseks. Kandidaadi haldussüsteemis on kindlaks määratud järgmised värbajapoolsed funktsionaalsed nõudmised: lisab kandidaate; otsib kliente; vaatab läbi kliendiga seotud konkursse; muudab kliendi andmeid; lisab konkursse; lisab kandidaate konkursile; vaatab läbi kandidaatide hetkelist staatust; otsib konkursse; lõpetab konkursse; lisab dokumente konkursile; lisab kandidaate; muudab kandidaadi andmeid; lisab kommentaare kandidaadile; lisab dokumente kandidaadile; otsib kandidaate; vaatab läbi kandidaadiga seotud konkurse;

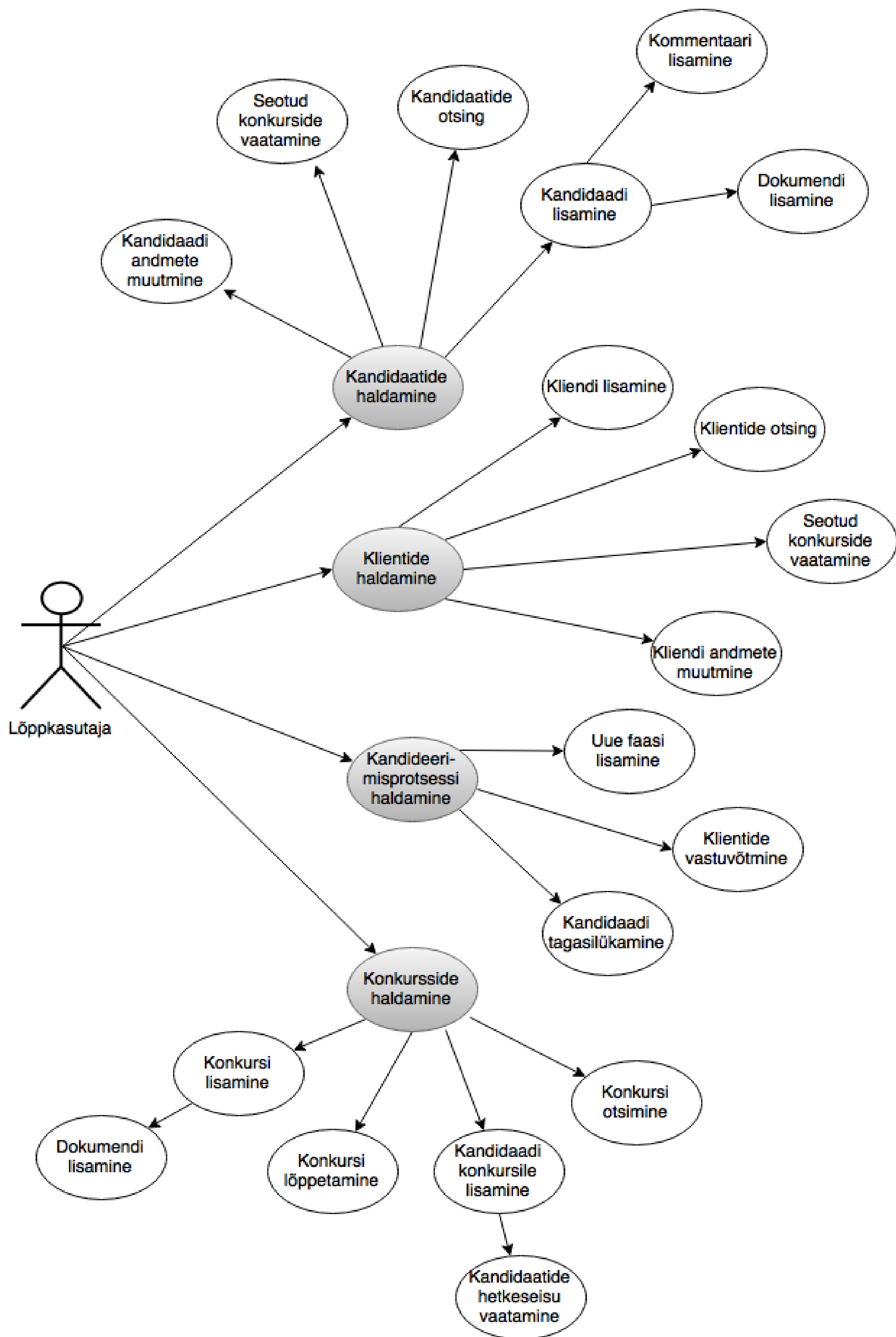
lisab uusi faase kandideerimisprotsessile; lükkab tagasi kandidaati; aktsepteerib kandidaati ametikohale.

Kandidaadi haldussüsteemis on kindlaks määratud järgmised mittefunktsionaalsed nõudmised: kättesaadavus, st süsteem peab töötama pidevalt (24/7); ohutus, st süsteem peab töötlema erakorralisi situatsioone ja jätkama oma tööd; turvalisus, st süsteem peab turvaliselt säilitama isiklikke andmeid; tootlikkus, st süsteem peab andma võimaluse töötada mitmekasutajaga üheagselt.

Nõudmised määras ja koostas arendajate meeskonna QA-mänadzer ja need on võetud programmi kasutusjuhendist.

3.3 Kasutusklassiskeem ja süsteemi soovilood

Funktsionaalsete nõudmiste alusel oli ehitatud kasutusklassiskeem, mis kirjeldab süsteemi käitumist, kui see töötab koos väliskeskkonnaga. Kasutusklassiskeem on toodud välja Joonis 5. Kandidaadi haldussüsteemis on tegutseja värbaja, kes asub väliskeskkonnas. Kasutusklassiskeem kirjeldab, mida saab värbaja teha käsitletava süsteemiga. Antud metoodikat kasutatakse süsteemi käitumise detailsemate nõudmiste selgitamiseks.



Joonis 5. Kandidaadi haldussüsteemi kasutusklassiskeem.

Kasutussoovilood on süsteemile nõudmiste kirjeldusviis, mis on sõnastatud ühe või enama lausena nii igapäevases, kui ka ärikeeles. QA-mänadzeri poolt olid moodustatud funktsionaalsete nõudmiste alusel süsteemile kasutussoovilood, mis on piiratud mahult ja keerukuselt. Kasutussoovilugude põhieeliseks on võimalus jagada süüsteemi arendamist väiksemateks etappideks. See aspekt võimaldab saavutada struktureeritumat koodi, mis lubab tulevikus programmeerijatel kergelt muuta juba olemasolevat koodi ja välja töötada uued funktsionaalsused.

Süsteem on jagatud neljaks suuremaks teemaks (soovilugude kogumiteks) süsteemi funktsionaalsete alamosade järgi:

1. Kandidaatide haldamine
2. Konkursside haldamine
3. Klientide haldamine
4. Kandidaadi haldamine kandideerimisprotsessis

Tabel 1. Kandidaadi haldamine süsteemis.

Üldine soovilugu: **Värbamisspetsialistina soovin hallata kandidaate süsteemis**		
#	Soovilugu	Vastuvõetavuse kriteeriumid
1	Värbamisspetsialistina soovin lisada kandidaate süsteemi, et hoida kõik kandidaadid ühes kohas.	1. Saab lisada kandidaadi isiku- ja kontaktandmed. 2. Kandidaadile saab määrata. 3. Kasutaja peab täitma kõik kohustuslikud väljad. 4. Kandidaatide nimekiri on kuvatud tähestikulises järjekorras.
2	Värbamisspetsialistina soovin lisada kandidaadi profiilile dokumente selleks, et hoida kandidaadiga seotud dokumendid tema profiiliga ühes kohas.	1. Dokumentide formaadid saavad olla: pdf, doc, pages, txt, odt, odp, ods, rtf. 2. Dokumendi suurus saab olla kuni 20 MB. 3. Kasutaja saab lisada mitu dokumenti. 4. Lisatud dokumente saab kustutada.

3	Värbamisspetsialistina soovin lisada kandidaadi profiilile kommentaare, et teada viimast seisu.	1. Saab lisada mitu kommentaari. 2. Kasutaja näeb kommentaari lisamise aega. 3. Kommentaari saab kustutada.
4	Värbamisspetsialistina soovin otsida kandidaate, et leida sobilikud kandidaadid konkursi jaoks.	1. Kandidaate saab otsida nime ja positsiooni järgi. 2. Otsingu tulemus saab olla 0 või rohkem.
5	Värbamisspetsialistina soovin näha kandidaadi profiilis temaga seotud konkursse, et saada kandidaadist ülevaade.	1. Kasutaja näeb kandidaadiga seotud konkursse. 2. Kasutaja näeb kandidaadi staatust konkursis.
6	Värbamisspetsialistina soovin muuta kandidaadi andmeid, et hoida tema profiil ajakohasena.	1. Peale andmete muutmist kasutaja näeb ainult uuendatud andmeid. 2. Kasutaja peab täitma kõik kohustuslikud väljad.

Tabel 2. Kliendi haldamine süsteemis.

Üldine soovilugu: **Värbamisspetsilistina soovin hallata kliente süsteemis**		
#	Soovilugu	Vastuvõetavuse kriteeriumid
1	Värbamisspetsialistina soovin lisada kliente süsteemi, et hoida kõik kliendid ühes kohas.	1. Saab lisada kliendi nime, kontaktandmed ja kirjelduse. 2. Kasutaja peab täitma kõik kohustuslikud väljad. 3. Klientide nimekiri on kuvatud tähestikulises järjekorras.
2	Värbamisspetsialistina soovin otsida kliente, et leida vajalikud kliendid.	1. Kliente saab otsida nime järgi. 2. Otsingu tulemus saab olla 0 või rohkem.
3	Värbamisspetsialistina soovin näha kõiki konkursse, mis on seotud kindla kliendiga, et saada kliendist ülevaade.	1. Kasutaja näeb kliendiga seotud konkursse. 2. Kasutaja näeb aktiivsed ja lõppenud konkursse.
4	Värbamisspetsialistina soovin muuta kliendi andmeid, et hoida tema profiil ajakohasena.	1. Peale andmete muutmist kasutaja näeb ainult uuendatud andmeid. 2. Kasutaja peab täitma kõik kohustuslikud väljad.

Tabel 3. Konkursi haldamine süsteemis.

Üldine soovilugu: **Värbamisspetsialistina soovin hallata konkursse süsteemis**		
#	Soovilugu	Vastuvõetavuse kriteeriumid
1	Värbamisspetsialistina soovin lisada süsteemi konkursse, et saaksin seal konkursse läbi viia.	1. Saab lisada konkursi nime, kirjelduse ja kestvuse. 2. Konkursile saab määrata kliendi. 3. Kasutaja peab täitma kõik kohustuslikud väljad. 4. Konkursside nimekiri on kuvatud kronoloogilises järjekorras. 5. Konkursid on jagatud kahte rühma: aktiivsed ja lõpetatud. 6. Lisatud konkursid peavad ilmuma aktiivsete konkursside nimekirja.
2	Värbamisspetsialistina peale konkursi loomist soovin lisada konkursile kandidaate, selleks et näha konkursis osalevaid kandidaate.	1. Kandidaadid on jagatud kolme gruppi: aktiivsed, vastu võetud ja tagasilükatud. 2. Kandidaadi lisamisel saab määrata tema leidmise allika (kas on ise kandidaarinud või teda on leitud sihtotsingu kaudu).
3	Värbamisspetsialistina soovin näha kandidaatide nimekirjas, mis on iga kandidaadi viimane faas ja staatus selleks, et omada kiiret ülevaadet kõikidest konkursis osalevatest kandidaatidest.	1. Kandidaadi viimane faas peab olema kuvatud iga kandidaadi kohta. 2. Kandidaadi staatus peab olema nähtav vaid peale seda, kui tema kohta on otsus langetatud (tagasilükatud, vastu võetud).
4	Värbamisspetsialistina soovin konkursi lõpetada selleks, et teada et antud konkursiga ei ole vaja enam tegeleda.	1. Lõpetatud konkurss peab liikuma aktiivsete konkursside nimekirjast lõpetatud konkursside nimekirja. 2. Konkurssi peab saama uuesti muuta aktiivseks.
5	Värbamisspetsialistina soovin lisada konkursile dokumente (näiteks	1. Dokumentide formaadid saavad olla: pdf, doc, pages, txt, odt, odp, ods, rtf. 2. Dokumendi suurus saab olla

	töökuulutus), et hoida konkursiga seotud dokumendid konkursiga ühes kohas.	kuni 20 MB. 3. Kasutaja saab lisada mitu dokumenti. 4. Lisatud dokumente saab kustutada.
6	Värbamisspetsialistina soovin otsida konkursse, et leida üles vajalikud konkursid.	1. Konkursse saab otsida nime järgi. 2. Otsingu tulemus saab olla 0 või rohkem.

Tabel 4. Kandidaadi kandideerimisprotsessis haldamine.

Üldine soovilugu: **Värbamisspetsialistina soovin hallata kandidaati kandideerimisprotsessis**		
#	Soovilugu	Vastuvõetavuse kriteeriumid
1	Värbamisspetsialistina soovin lisada kandidaadile uue faasi selleks, et liigutada kandidaat järgmisesse faasi.	1. Faasi lisamisel saab määrata faasi tüübi (intrevjuu, testimine jne). 2. Faasile saab lisada sisu. 3. Kasutaja näeb, millal iga faas on lisatud.
2	Värbamisspetsialistina soovin kandidaadi määrata vastuvõetuks selleks, et teada, et kandidaat on tööle võetud.	1. Kasutaja saab lisada kommentaari, kui määrab kandidaadi vastuvõetuks. 2. Peale seda, kui kandidaat on määratud vastuvõetuks liigub ta aktiivsete kandidaatide nimekirjast vastuõetud kandidaatide nimekirja. 3. Kandidaadi staatus muutub vastuvõetuks.
3	Värbamisspetsialistina soovin kandidaadi määrata tagasilükatuks selleks, et teada, et kandidaat ei ole enam konkursis aktiivne.	1. Kasutaja saab lisada kommentaari, kui määrab kandidaadi tagasilükatuks. 2. Peale seda, kui kandidaat on määratud tagasilükatuks liigub ta aktiivsete kandidaatide nimekirjast tagasilükatud kandidaatide nimekirja. 3. Kandidaati tagasilükamisel kasutaja saab määrata, kas kandidaat ei ole sobilik või ta ise on loobunud konkursis edasi osalemisest.

		4. Kandidaadi staatus muutub tagasilükatuks.

Kandidaadi haldussüsteemi arendati ettevõttes Ignite OÜ. Arendajate meeskonna QA-mänadzer on Seda Sahradyan, kes kaitseb selles semestris magistritöö teemal: „Ärianalüüsi protsessi kujundamine ja rakendamine värbamissüsteemi tarkvara loomiseks IGNITE OÜ näitel“. Analüüsi tulemusel, määras Seda nõudmised süsteemile ja koostas nende alusel kasutaja soovilood tabelite kujul, mis on kasutatud antud töös.

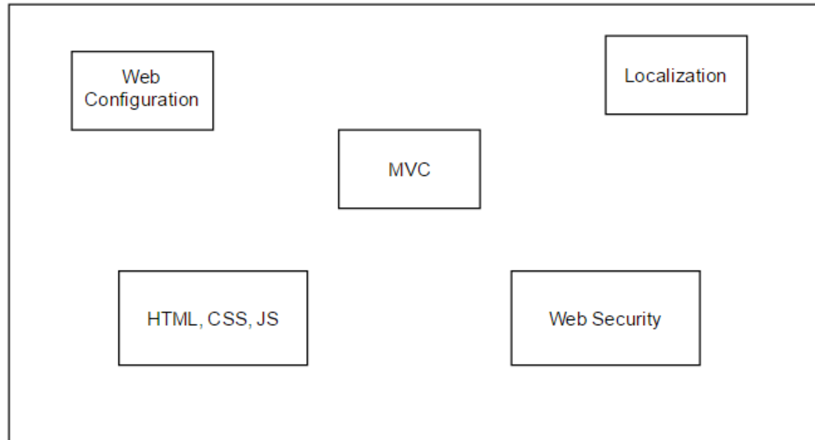
3.4 Arhitektuuri komponentide määramine

Kandidaatide haldus süsteem hakkab sisaldama piisavalt palju funktsionaali, seetõttu on oluline valida sobiv jaotus vorm. Martin Fowler kirjutas jaotus vormi kohta järgmise „Vatamata sellele, et määrata ettevõtte kolm rakenduse kihti ei ole raske, kihtideks jagamine sõltub rakenduse keerulisusest tervikuna. Lihtne skript andmete laadimiseks andmebaasist ja nende andmete kuvamiseks veebilehel võib asuda ühe meetodi kehas. Mul on püüdlus jagada skript kolmeks kihiks, kuid konkreetselt sellel juhul saan seda teha iga kihi käitumise väljatoomisega allfunktsioonidesse, kui süsteem muutub keerulisemaks, siis jaotaksin rakenduse kihid kasutades eraldatud klasse. Kui keerulisus kasvab veel rohkem, siis eraldatud pakettideks. Minu põhiline nõuanne on valida kõige sobivama programmi klassideks tükeldamisvormi sõltuvalt püstitatud probleemist“.

Rakenduse arhitektuuri põhiüksus on komponent. Üks komponent vastab ühele kihile kolmekihilises arhitektuuri mudelis. Kuna rakenduse arhitektuuril on kolm kihti, siis iga kihi jaoks tuleb luua oma komponent. Kokku tuleb kolm komponenti, mis on omavahel seotud sõltuvusega.

3.4.1 Esitluskiht

Esimene komponent on esitluskiht, mis on seotud süsteemi kasutajaga. Esitluskihti kasutatakse andmete saamiseks kasutajalt ja nende edastamiseks äriloogikakihile edasiseks töötlemiseks ja kui andmed on saadud objektis, see vastutab objekti esitluse eest sobilikul kujul, mis on arusaadav kasutaja jaoks. Esitluskiht hakkab sisaldama kontrollereid, süsteemi turvalisuse konfiguratsiooni faile, lokalisatsionni faile, lehekülje HTML-i mustreid, CSS ja js faile.

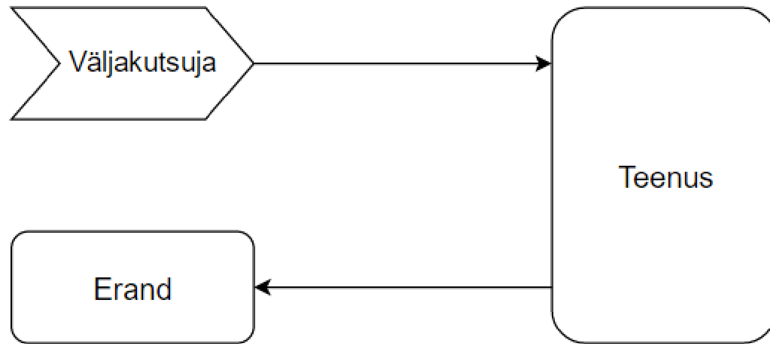


Joonis 6. Esitluskihi komponendid.

3.4.2 Äriloogikakiht

Teise komponendina esineb äriloogikakiht. Äriloogikakiht töötab sillana esitlus- ja andmekihi vahel. Kõik esitluskihilt saadud kasutaja muutujad edastatakse äriloogikakihti. Andmekihilt saadud tulemused on objektikujul esitatud andmed. Äriloogikakiht on kõige olulisem klass kogu arhiterktuuris, kuna see sisaldab programmi kogu loogikat. Äriloogikakihi põhiliseks elemendiks on teenused. Nendes sisaldub loogika milles teostatakse erinevaid operatsioone andmetega: loomine, lugemine, muutmine, kustutamine. Teenuste funktsioone hakatakse välja kutsuma kontrolleritest, mis asuvad esitluskihis.

Programmis võib tekkida erandolukord, kui seda ei töödelda vajalikul viisil, siis programm läheb rivist välja. Et seda vältida, hakkab äriloogikakihis toimuma erandite töötlus ja ettenägematute olukordade korral töötleb programm vea ja jätkab oma tööd. Mõnedel juhtudel hakkab erand välja viskuma esitluskihti selleks, et kuvada kasutajale viga, mida ta võib lugeda, mis näitab miks tegevus ei õnnestu ja milles on vea põhjus. Veatöötlus on skemaatiliselt kujutatud Joonisel 7.

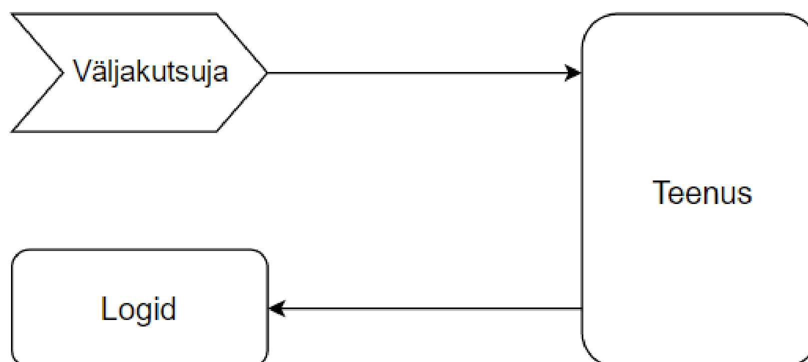


Joonis 7. Äriloogikakiht ja veatöötlus.

Skeemil on kujutatud teenuse ja väljakutsuja koostöö. Väljakutsuja kutsub välja teenuses meetodi. Meetodis toimuvad mõned arvutused ja nende arvutuste tulemusel ilmuvad erandid. Sel hetkel arvutused lõpevad ja teenus viskab välja erandi väljakutsujale [15].

Logimine on programmi lahutamatu osa, mis lubab analüüsida programmis toimuvaid sündmusi. Peale rikete tekkimist vigade registreerimise log faili sisu uurimine võimaldab sageli mõista nende põhjusi ja määrata etapp, millel tekkis rike. On väga oluline, et äriloogikakihis toimuks kõik tegevused korrekselt ja õiges järjekorras, just seetõttu hakatakse siin kasutama logimist.

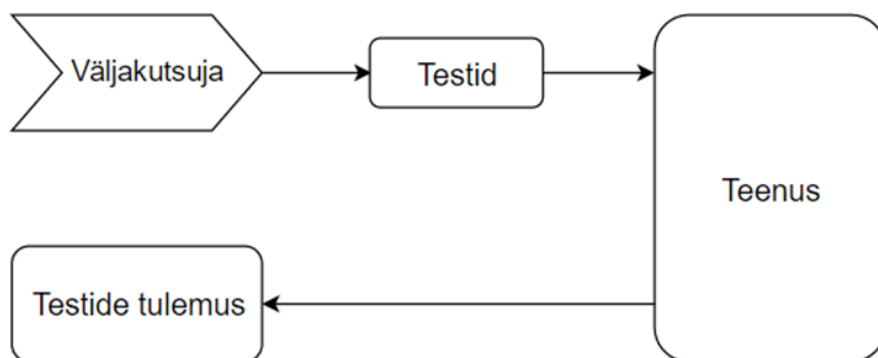
Põhilised tegevused, mis toimuvad teenustes salvestuvad spetsiaalses ajutises failis. Logfailis kuvatakse WARN ja INFO tasemed. INFO-taset kasutatakse mõne sündumse kohta info kuvamiseks. WARN-taset kasutatakse selleks, et kuvada info vea või ebatavaliseolukorra kohta, mis on potentsiaalselt ohtlik. Teenuste logimine on kujutatud Joonisel 8.



Joonis 8. Äriloogikakiht ja logimine.

Skeemil on kujutatud väljakutsuja, teenuse ja logide kootöö. Väljakutsuja kutsub välja meetodi teenuses, kus toimuvad tegevused andmetega. Kõik päringud andmebaasi ja tegevused andetega salvestuvad log-faili [15].

Tänapäeval saavutab ühe suuremat populaarsust TDD. TDD on tarkvara arenduse tehnika, milles kirujatatakse alguses test kindlale funktsionaalile ja seejärel kirjutatakse selle funktsionaali realiseerimine. Tulemusel on kood mitte ainult kirjutatud ja testitud, vaid testid mitteotseselt esitavad nõudmisi funktsionaalile ning näitavad selle funktsionaali kasutamise näided. Erilist rolli mänguvad siin raamistik JUnit, mis lubab testida kindlat funktsionaali ühiktestidega. Teenused peavad olema kaetud ühiktestidega programmi kriitiliste punktide kattega. Testide väljakutsumisel kutsutakse välja ka vastavad teenuste meetodid. Tuleb märkida, et on olemas erinevad programmi koodi katte viisid: operaatorite kate, tingimuste kate, teede kate, funktsioonide kate, väljund/sisend kate, parameetrite väärtuste kate. Kandidaadi haldussüsteemis on rõhk parameetite väärtuste katel. Teenuste testimine on kujutanud Joonisel 9.



Joonis 9. Äriloogikakiht ja testimine.

Skeemil on kujutatud teenuste koostöö ühiktestidega. Väljakutsuja käivitab testi. Testid võivad olla käivitatud nii ühiktestidena, kui ka tervete klassidena, mis sisaldavad ühikteste [15]. Testklassid jagavad kõige sagedamini kõiki ühikteste funktsionaalsuse järgi. Ühiktestis saadetakse andmeid teenusesse, seal neid töödeldakse ja teenus tagastab tegevuse faktilise tulemuse. Oodatavad tulemust võrreldakse faktilisega ja selle põhjal saadakse testi tulemus. Kui test oli edukalt läbitud, siis on suur tõenäolisus, et tegevused teenuses kulgevad õiges järjekorras ja loogika on kirjutatud õigesti. Tuleb märkida, et programmeerijad võrdlevad väga sageli oodatavat ja faktilist tulemust pöördjärjestikkuses, mis on vastuolus raamisitku kooskõlaga.

3.4.3 Andmekiht

Kolmanda komponendina esineb andmekiht. Andmekiht on süsteemi admetehoidla. Selles hakkatakse sisaldama konfiguratsioonid andmebaasiga ühendamiseks, kõik mudelite ja repositooriumide klassid. Repositooriumid hakkavad realiseerima JPARepository<Object, Long> liideseid, mis võimaldab pöörduda otse andmebaasi, kasutades vastavat meetodit. Mudelid on klassid, mis sisaldavad tabeli veergudele vastavaid välju ning getter'eid ja setter'eid. Koodis on mugav töötada objektiga, mis sisaldab mõnda andmetevalikut, seetõttu kasutatakse selliseid mudeleid andmete edastamiseks ühelt klassilt teisele. Toetudes nõudmistele võib oletada, et andmebaasi põhilisteks tabeliteks on: värbaja, konkurss, klient ja kandidaat.

4 Prototüübi realiseerimine

Kandidaadi haldussüsteem on üsna keeruline, seetõttu on väga tähtis põhjalikult läheneda programmi osadeks jagamise sobiva vormi valikule. Programmi kihtide määramise üks variantidest on pakettide loomine projekti juurkausta sees. Kandidaadi haldussüsteemi puhul on kõige sobivam jagamisviis allprojektide loomine. See viis sobib süsteemi arendamisel rohkem, sest:

- On võimalus liita lisaraamistikud, mis kuuluvad kindlasse kihti. Raamistikud lisatakse vaid sellele allprojektile ja sellest sõltuvustele.
- Projektideks jagamine on kõige üldisem meetod. Pakettideks jagunemine hakkab toimuma iga allprojekti sees, et ühendada loogikalt sarnaseid klasse.
- Rakenduse arhitektuuri mõistmine on kõige lihtsam, mis on väga hea rakenduse järgnevaks arendamiseks teiste arendajate poolt.

Kandidaadi haldussüsteemi prototüübi realiseerimine arendatakse Java 8 programmeerimiskeeles. Lisaramistikena kasutatakse: Spring, Gradle, Hibernate, Thymeleaf, Log4J, Flyway, JUnit.

4.1 Rakenduse moodulite realiseerimine

Ükski projekt Java platvormil ei tule toime ilma kokkupaneku tööristadeta. Kandidaadi haldussüsteemi projekti kokkupaneku tööriistana oli otsustatud kasutada gradle raamistikku. Rakenduse moodulite realiseerimiseks, arhitektuuri kolmeks kihiks põhilise jagamisprintsiibina oli otsustatud kasutada gradle raamistiku funktsionaale, aga täpsemalt juurprojekti loogika jagamist allprojektideks. Programmi juurkausta struktuur näeb välja build-failide suhtes järgmiselt.

```
recruitmentdrive/
        build.gradle
        settings.gradle
        web/
        build.gradle
        core/
                build.gradle
        persistence/
                build.gradle
```

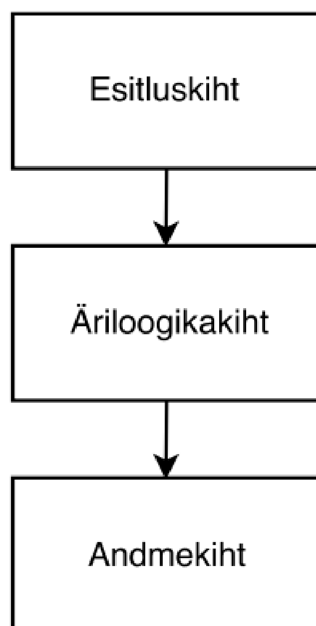
Joonis 10. Kandidaadi haldussüsteemi juurkausta struktuur.

Projekti juurkaustas on kaks faili: build.gradle ja settings.gradle. Build.gradle failis on määratud järgmised projekti omadused: rakenduse encoding, java versioon, maven repositooriumid ja raamistikud, mida hakatakse kasutama kogu projektis sõltumata arhitektuuri kihist. Globaalsete raamistike hulka kuuluvad slf4j andmete logimiseks ja Junit raamistike testideks. Allprojektide genereerimiseks oli kasutatud settings.gradle juurfail, mis sisaldab allprojektide liitumist juurprojektiga:

```
include "persistence", "web", "core"
```

Joonis 11. Allprojektide liitumine juurprojektiga.

Kandidaadi haldussüsteemis realiseeritakse kolmekihilist arhitektuuri, milles iga kiht sõltub eelnevast. Igale kihile vastab eraldi rakenduse moodul, aga see tähendab, et moodulitel on analoogiline sõltuvus.



Joonis 12. Kolmekihiline arhitektuur moodulite sõltuvusega.

Mooduli juurkaustas on build.gradle fail, milles määratakse selle mooduli sõltuvust teistest. Samuti on selles failis konkretsele kihile kuuluvad raamistikud. Web-moodul on esitluskiht ja see sõltub äriloogikakihist, sõltuvus antakse web/build.gradle failis ja näeb välja järgmiselt:

```
dependencies {
    compile project(':core')
}
```

Joonis 13. Sõltuvus web- ja core-mooduli vahel.

Core-moodul on äriloogiikakiht, mis sõltub admekihist. Sõltuvus määratakse core/build.gradle failis ja see näeb välja järgmiselt:

```
dependencies {
    compile project(':persistence')
}
```

Joonis 14. Sõltuvus core- ja persistence-mooduli vahel.

Persistence-moodul on andmekiht ja see ei sõltu teistest moodulitest, persistence/build.gradle fail on tühi. Moodulite seosed on välja toodud Lisas 1.

4.2 Moodulite struktuuri realiseerimine

Väga oluline aspekt tarkvara arenduses on moodulite struktuuri realiseerimine. Moodulite struktureerimine ühe mustri järgi lubab jagada programmikoodi eraldi loogilisteks komponentideks, mis aitab programmeerijatel kergemini orienteeruda koodis. Moodulite väär struktuur segab programmeerijaid OOP põhimõtete kasutamisel, vähendab üldise süsteemi mõistmist ja teeb võimatuks edasist süsteemituge teiste arendajate meeskondade poolt. Kandidaadi haldussüsteemis on iga moodul struktureeritud ühtse Maven Directory Structure mustri järgi. Antud struktuur oli valitud seetõttu, et see on kõige levinum ettevõtluse tarkvara arendamises ning on arendajatele tuttav [16].

Tabel 5. Kandidaadi haldussüsteemi moodulite struktuur.

Tee	Kirjeldus
src/main/java	Rakenduse lähtekood
src/main/resources	Lähtekoodiga seotud failid (ressursid)
src/test/java	Testide lähtekood
src/test/resources	Testide lähtekoodiga seotud failid (testide ressursid)
README.txt	Mooduli kohta infot sisaldav tekstifail

Kandidaadi haldussüsteemi mooduli struktuuri näide on välja toodud Lisas 2.

4.3 Liideste ja abstraksete klasside loomine

Süsteemi struktureerimise järgmine tase on liideste ja abstraksete klasside loomine. Java on objektorienteeritud programmeerimiskeel, mille lahutamatuks osaks on tuletamine. Tuletamise kasutamisel loob programmeerija uut klassi ja tuletab seda vanast, juba määratud klassist. Koodis seda kirjutatakse „extends“ („implements“ liideste puhul), pärast seda tuleb baasklassi nimi. Sellega saab uus klass juurdepääsu kõigile baasklassi väljadele ja meetoditele. Kasutades tuletamist, saab luua üldklassi, mis määrab üldomadusi seotud elementide hulgale. Seejärel saab tuletada sellisest baasklassist ja luua uus klass, millel on oma unikaalsed väljad. Tuletatavat põhiklassi nimetatakse Javas superklassiks. Tuletavat klassi nimetatakse algklassiks. Tuleb välja, et algklass on superklassi spetsialiseeritud versioon, mis tuletab kõiki superklassi liikmeid ja lisab enda unikaalseid elemente.

Abstraktne klass on baasklass, milles on lubatud meetodite realiseerimine. Liides on abstraktne klass, millel kõik meetodid on avalikud ja ei ole realiseeritud. Java klassi saab tuletada paljudest liidestest, kuid ainult ühest abstraktsest klassist [17].

Kandidaadi haldussüsteemis on loodud DataSourceProperties abstraktne klass, milles määratakse andmebaas, kasutaja ja salasõna, mida hakatakse kasutama programmis. Selle

abstraktse klassi implementeerimine sõltub sellest, mis profiil on antud hetkel aktiivne. Kandidaadi haldussüsteemis on kaks profiili: test ja live. Test-profiili kasutatakse programmi testimiseks. Live-profiili kasutatakse tavaliseks tööks. LiveDataSourceProperties ja TestDataSourceProperties klasse tuletatakse abstraktsest DataSourceProperties klassist.

```
@Configuration
@Profile("live")
@PropertySource("classpath:persistence-live.properties")
public class LiveDataSourceProperties extends DataSourceProperties {}
```

Joonis 15. LiveDataSourceProperties klassi sisu.

```
@Configuration
@Profile("test")
@PropertySource("classpath:persistence-test.properties")
public class TestDataSourceProperties extends DataSourceProperties {}
```

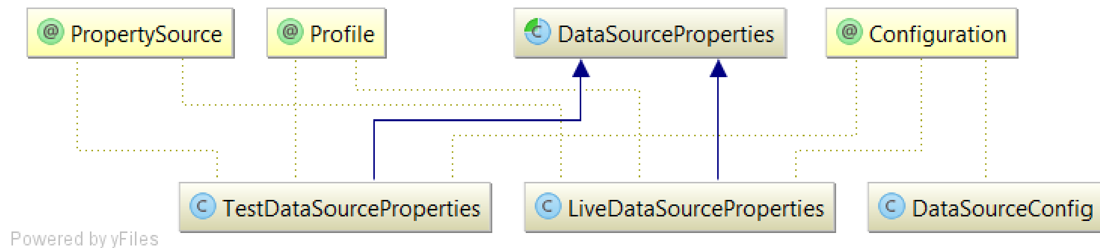
Joonis 16. TestDataSourceProperties klassi sisu.

LiveDataSourceProperties ja TestDataSourceProperties klassidel on annotatsioon @Configuration, mis ütleb Spring raamistikule, et antud klass on konfiguratsioonifail. Annotatsioon @Profile osutab rakenduse jooksvale aktiivsele profiilile. Olenevalt sellest, mis profiil on märgitud VM seadistustest, hakatakse kasutama üht pakutud klassi realiseerimist. Andmebaasi ühendamiseks võetakse andmeid omadusfailist, mida valitakse sõltuvalt aktiivsest konfiguratsioonist.

```
public abstract class DataSourceProperties {
    @Resource
    private Environment env;

    public String getDriverClassName() {
        return org.postgresql.Driver.class.getName();
    }

    public String getJdbcUrl() {
        return "jdbc:postgresql://localhost:5432/" +
env.getRequiredProperty("db.database");
    }

    public String getUsername() {
        return env.getRequiredProperty("db.username");
    }

    public String getPassword() {
        return env.getRequiredProperty("db.password");
    }
}
```

Joonis 17. DataSourceProperties klassi sisu.

Abstraktne baasklass sisaldab getter'eid. Getter'id tagastavad väärtust omadusfailist sõltuvalt aktiivsest profiilist. Neid väärtusi kasutatakse süsteemi edaspidiseks konfigureerimiseks, et see võiks töötada koos andmebaasiga. Kandidaadi haldussüsteemi tuletamise näide on välja toodud Lisas 3.



Joonis 18. Klasside tuletamine ja sõltuvus.

Programmis kasutatakse ka liideseid. Näiteks, pöördumiseks andmebaasi poole kasutatakse JpaRepository<Object, Long>, mis pakub standartsete JPA meetodite komplekti (Joonis 19).

```
@Repository
public interface ClientRepository extends JpaRepository<Client, Long>
{

}
```

Joonis 19. JpaRepository liidese tuletamine.

@Repository annotatsioon osutab sellele, et klass on repositoorium admetele ligipääsemiseks.

4.4 Mustrid

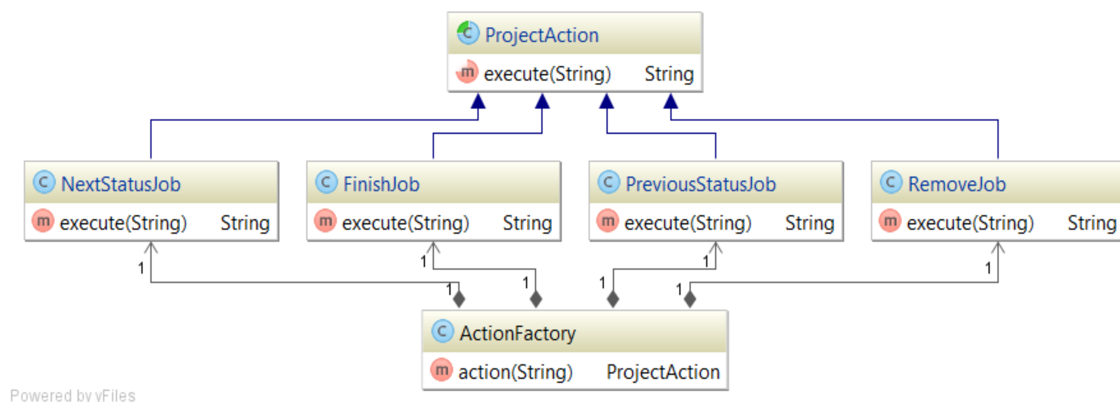
Tarkvaraarendajad kogenud kasutavad mustrite disaini, mis aitavad lahendada tarkvara arendamise põhiprobleeme. Muster annab lahendusele oma nime, mis lihtsustab kommunikatsiooni arendajate vahel, võimaldades viidata tarkvara projekteerimise tuntud mustritele [18].

4.4.1 Tootmismuster

Tootmismuster annab liidese seotud ehk sõltuvate objektide loomiseks määramata nende konkreetset klassi. Arendamise protsessis peavad mõned objektid olema koordineeritult initsialiseeritud, st initsialiseeritud ainult kui on täidetud kindlad tingimused. Näiteks, töös kasutajaliidestega peab üks süsteem kasutama üht hulka objekte, teine süsteem peab kasutama teist hulka objekte. Tootmismuster tagab seda, et süsteemil on alati õiged objektid konkreetse situatsiooni jaoks. [19] [20]

Kandidaadide haldussüsteemis on realiseeritud tootmismuster, mis aitab kindlat toimingut konkursiga: lõpetada konkurss; kustutada konkurss; üle minna järgmisele staatusele; üle minna eelmisele staatusele.

Üks neljast toimingutest edastub ActionFactory objekti action(String) meetodisse, kus otsitakse sobivat objekti selle käsu täitmiseks. Kui sobiv käsk ei ole leitud, siis meetod ei tagasta objekti, tekib RuntimeException erand. Peab ka märkima, et kõik neli objekti (RemoveJob, NextStatusJob, PreviousStatusJob, FinishJob) tulevad ProjectAction abstraksest klassist ja neil on oma execute() meetod. Tagastatud objektil kutsutakse välja execute(String) argumentiga id-konkurss. Argument edastatakse selleks, et määrata kindlaks, mis konkursile kasutatakse toiminguid. Tootmismustri lähtekood on toodud välja Lisas 5.



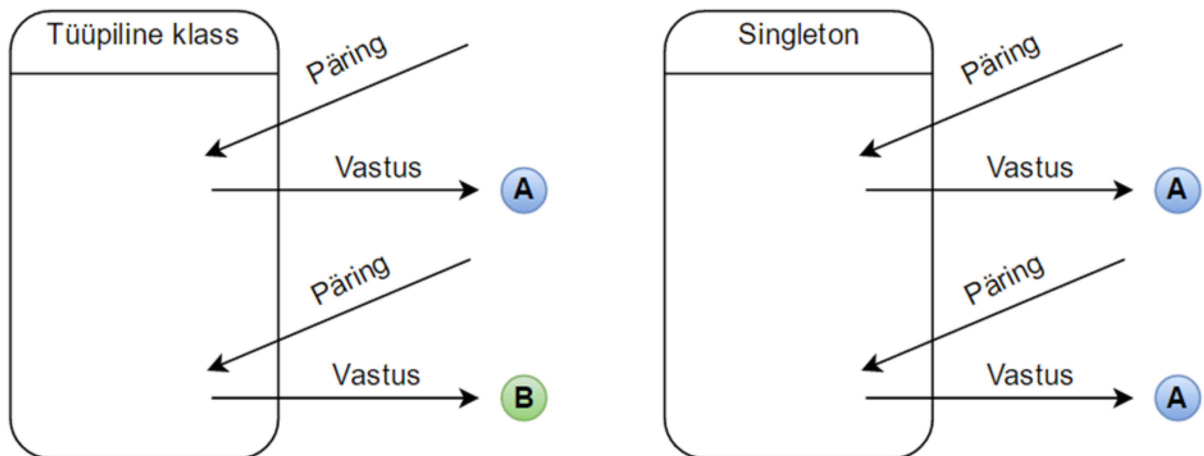
Joonis 20. Kandidaadi haldussüsteemi tootmismustri skeem.

4.4.2 Ehitusmuster

Ehitusmuster lahendab objekti hulga konstruktorite loomise probleemi. Selle asemel, et luua ja kasutada hulka konstruktoreid, kasutatakse vaheobjekti – builder'id, mis sisaldab meetodeid. Igat meetodid kutsutakse välja järjestikku ja tehakse vajalikku sammu objekti loomise protsessis. Kui objekt on valmis kutsutakse välja konstrueeritud objekti tagastavat spetsiaalset meetodit [18]. Kandidaadi haldussüsteemi ehitusmuster on välja toodud Lisas 6.

4.4.3 Singleton muster

Singleton muster on projekteerimismuster, mis tagab, et rakenduses on üks klassi eksemplaar globaalse juurdepääsu punktiga. Singleton mustril on spetsiaalne meetod soovitud objekti initsialiseerimiseks. Kui seda meetodit kutsutakse välja, siis see kontrollib klassiobjekti olemasolu. Kui objekt on olemas, siis meetod lihtsalt tagastab sellele viite. Kui ei ole, siis meetodis initsialiseerub klassi uus objekt ja tagastatakse viide uuele eksemplarile. [21]



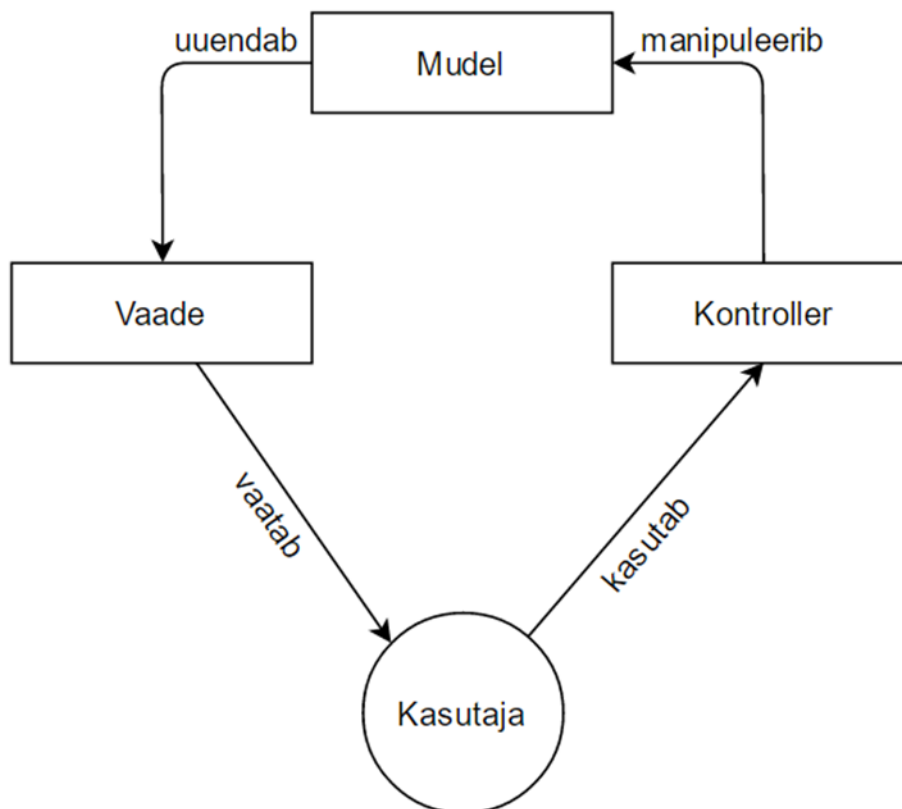
Joonis 21. Tüüpiline klass ja Singleton muster.

Singleton'i klass tagastab alati üht ja sama eksemplari sõltumata sellest, mitu korda kutsutakse välja. Tavaline klass lubab luua kui palju tahes klassi eksemplaare, aga singleton'il võib olla ainult üks klassi eksemplar ühe protsessi jaoks. Singleton'i objekt võimaldab globaalset liigipääsu kõigile ressursidele klassi sees. Singleton'i kasutatakse siis, kui on eelistatav üks liigipääsupunkt. [22]

Plussideks on kontrollitud juurdepääs ainsale eksemplarile ja üks juurdepääsu punkt. Miinusteks on globaalsed objektid võivad olla kahjulikud objektsele programmeerimisele, mõnel juhul viivad mastabeerimata projekti loomisele ja singleton muudab keerulisemaks moodultestide kirjutamise ja TDD järgimist. [18]

4.4.4 MVC muster

MVC on arhitektuuri muster, mis võimaldab jagada rakendust kolmeks omavahel seotud osaks. Mudel annab võimalusi kasutada teadmisi: objekt või JAVA POJO, mis reageerib päringutele muutes oma seisundid. Ei sisalda informatsiooni, kuidas neid teadmisi saab visualiseerida. Vaade vastutab informatsiooni kuvamise ja visualiseerimise eest. Tihti esineb graafiliste elementidega akna või vormi kujul. Kontroller tagab sidet kasutaja ja süsteemi vahel, kontrollib andmete sisestamist kasutaja poolt ja kasutab mudelit andmete uuendamiseks vaates. Vaade ja mudel on kontrolleris eraldi [20]. HomeController'i lähtekood välja toodud Lisas 6.



Joonis 22. MVC andmevoo diagramm.

4.4.5 DTO muster

DTO on objekt, mis kannab üle andmeid protsesside vahel selleks, et vähendada väljakutsete meetodite arvu. Töös kaugel asuva liidesega on iga päring sellele küllalt kulukas. Tulemusena tuleb vähendada väljakutsete arvu, mis tähendab suurema hulga andmete ülekandmise vajadust ühe väljakutsega. Üks viisidest seda realiseerida on hulga parameetrite kasutamine. Kuid seejuures pahatihti tuleb välja kood, mis on kohmakas ja ebamugav. Samuti on see sageli võimatu sellistes keeltes nagu Java, mis tagastavad vaid ühe väärtuse. Lahenduseks on DTO muster, mis saab säilitada kogu väljakutseks vajaliku info. See peab olema serialiseeritud mugavaks edastamiseks võrgus. Tavaliselt kasutatakse kogumis-objekti andmete edastamiseks DTO ja rakenduse objekti vahel. [23]

Kandidaadi haldussüsteemis saab DTO mustri näitena tuua FileForm klassi. Selles on kaks välja: MultipartFile file ja Candidate candidate. MultipartFile file väljas säilitatakse dokumenti kandidaadi kohta. Candidate candidate väljas säilitatakse kõiki kandidaadiga seotud andmeid. FileForm klassi loomise põhieesmärk on vähendada välja kutseid meetoditesse. DTO näide on toodud Lisas 7.

5 Kokkuvõte

Käesoleval ajal arendatakse suuri ja keerulisi süsteeme. Nende realiseerimiseks peab olema ehitatud struktureeritud arhitektuur, mis aitab edaspidi arendada uusi funktsionaalsusi minimaalselt muutes olemasolevat koodi ning vähendab kulusid süsteemi haldamisele ja toetamisele. Iga programmeerija, sõltumata kogemusest, kulutab suurt osa oma ajast kirjutatud koodi analüüsimisele ja mõistmisele. Õigesti ehitatud arhitektuur lubab vältida koodi dubleerimist ja paisutatust ning lihtustada programmeerija elu.

Antud töö põhieesmärk on uurida üht kõige kasutatavamat mitmekihilist arhitektuuri ettevõtluse arendamises – kolmekihilist arhitektuuri. Toetudes teoreetilisele alusele pidi autor ehitama arhitektuuri prototüübi ja realiseerima selle reaalses programmis.

Autor saavutas antud töös püstitatud eesmärgid, ehitas struktureeritud kolmekihilise arhitektuuri reaalses programmis - kandidaadi haldussüsteemis. Esimeses peatükis tegi autor arhitektuuri üldist ülevaate: tõi arhitektuuri mõiste standardi IEEE 1417 alusel, kirjaldas arhitektuuri tekke protsessi ja eesmärgid. Samuti tõi autor tarkvara mitmekihilise arhitektuuri ülevaade ja näitas detailsemalt tarkvara komekihilist arhitektuuri igi kihi kirjeldusega. Teises peatükis kirjeldas autor aine valdkonda, milleks hakatakse süsteemi realiseerima ning esitas funktsionaalsed ja mittefunktsionaalsed nõudmised süsteemile. Määras ja kirjeldas komponendid, mis kuuluvad arhitektuuri. Kolmandas peatükis realiseeris arhitektuuri prototüübi Java programmeerimiskeeles, kasutades lisaraamistiku sobivat funktsionaali: lõi sidemed rakenduse moodulite vahel; struktureeris moodulite pakettid; näitas klasside pärinemist süsteemis ja realiseeris projekteerimise mustrid.

Analüüsides arhitektuuri tuli autor järeldusele, et keeruliste süsteemide arendamist tuleb alustada programmi arhitektuuri määramisest. Selleks tuleb teha ainevaldkonna ülevaade, milles hakatakse kasutama süsteemi ja määrata funktsionaalsed ja mittefunktsionaalsed nõudmised süsteemile. Tarkvara arhitektuuri prototüüp ehitatakse nõudmiste alusel süsteemile. Autor tõdes ka, et õigesti ehitatud arhitektuur aitab struktureerida süsteemi ja

vähendab selle keerulisust, mis lõppkokkuvõttes vähendab kulusid süsteemi edaspidisele arendamisele, toetamisele, haldamisele. Autor tegi kindlaks, et puhta koodi põhimõtete ja projekteerimismustrite kasutamine võimaldavad vähendada koodi dubleerimist ja teha kood abstraktsemaks. Teha muudatusi või arendada edasi funktsionaali on selles koodis väga lihtne.

Vaatamata sellele, et programmi loogikat testitakse ühiktestide abil, ei puuduta antud töö arhitektuuri testimist, st realiseeritud arhitektuuri vastavust algselt määratud nõudmistele süsteemile. Arhitektuuri tesitimist saab käsitleda edaspidise uuringuna.

Kasutatud kirjandus

[1] R. Hilliard, „ISO/IEC/IEEE 42010," 6 Detsember 2015. [Võrgumaterjal]. Available: http://www.iso-architecture.org/ieee-1471/. [Kasutatud 15 Märts 2016].

[2] M. Fowler, Patterns of Enterprise Application Architecture, Boston: Addison-Wesley, 2002.

[3] D. N. Chorafas, Enterprise Architecture and New Generation Information Systems, CRC Press, 2001.

[4] „НОУ ИНТУИТ | Лекция | Обзор архитектуры современных программных систем," 10 Mai 2016. [Võrgumaterjal]. Available: http://www.intuit.ru/studies/courses/2314/614/lecture/13316. [Kasutatud 16 Mai 2016].

[5] „Technical Architecture," Oracle, Inc., 8 Aprill 2016. [Võrgumaterjal]. Available: https://docs.oracle.com/cd/E12456_01/rwms/pdf/141/html/impg/technical-architecture.htm. [Kasutatud 13 Aprill 2016].

[6] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad ja M. Stal, Pattern-Oriented Software Architecture Volume 1: A System of Patterns, Wiley, 1996.

[7] M. Fowler, K. Beck, J. Brant, W. Opdyke ja D. Roberts, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.

[8] D. Schmidt, F. Buschmann, H. Rohnert ja M. Stal, Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects, Wiley, 2000.

[9] „Advantages and Disadvantages of using 3 tier architecture," 2 Aprill 2016. [Võrgumaterjal]. Available: http://asp-net-by-parijat.blogspot.com.ee/2014/12/advantages-and-disadvantages-of-using-3.html. [Kasutatud 4 Aprill 2016].

[10] R. Sinha, J. Boutelle ja A. Ranjan, „3 Tier Architecture," 2008. [Võrgumaterjal]. Available: http://www.slideshare.net/guestd0cc01/3-tier-architecture. [Kasutatud 17 Märts 2016].

[11] E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley, 2003.

[12] R. Buettner, „A Framework for Recommender Systems in Online Social Network Recruiting: An Interdisciplinary Call to Arms," 2014. [Võrgumaterjal]. Available: https://www.researchgate.net/publication/261844032_A_Framework_for_Recommender_Systems_in_Online_Social_Network_Recruiting_An_Interdisciplinary_Call_to_Arms. [Kasutatud 20 Märts 2016].

[13] R. Buettner, „A Systematic Literature Review of Crowdsourcing Research from a Human Resource Management Perspective.," 2015. [Võrgumaterjal]. Available: https://www.computer.org/csdl/proceedings/hicss/2015/7367/00/7367e609.pdf. [Kasutatud 20 Märts 2016].

[14] К. И. Вигерс ja Д. Битти, Разработка требований к программному обеспечению, БХВ-Петербург, 2014.

[15] R. Liivrand, „Service Layer," 2015. [Võrgumaterjal]. Available: http://maurus.ttu.ee/ained/IDU0200_2015/doc/78/ServiceLayer.pdf. [Kasutatud 11 Aprill 2016].

[16] J. v. Zyl, „Maven – Introduction to the Standard Directory Layout," The Apache Software Foundation, 9 Märts 2014. [Võrgumaterjal]. Available: https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html. [Kasutatud 1 Mai 2016].

[17] „Object-Oriented Programming Concepts," Oracle, Inc., 14 September 2015. [Võrgumaterjal]. Available: https://docs.oracle.com/javase/tutorial/java/concepts/. [Kasutatud 14 Mai 2016].

[18] E. Freeman, E. Robson, B. Bates ja S. K., Head First Design Patterns, O'Reilly Media., 2004.

[19] T. Cohen ja J. Gil, „Better Construction with Factories," 2007. [Võrgumaterjal]. Available: http://www.jot.fm/issues/issue_2007_07/article3.pdf. [Kasutatud 5 Aprill 2016].

[20] tutorialspoint.com, „Design Patterns in Java Tutorial," 2016. [Võrgumaterjal]. Available: http://www.tutorialspoint.com/design_pattern/. [Kasutatud 5 Aprill 2016].

[21] A. Shalloway ja J. R. Trott, Design Patterns Explained: A New Perspective on Object Oriented Design, 2nd Edition, Addison-Wesley, 2004.

[22] A. Inc., „Singleton," Apple Inc., 2015. [Võrgumaterjal]. Available: https://developer.apple.com/library/mac/documentation/General/Conceptual/DevPedia-CocoaCore/Singleton.html. [Kasutatud 6 Aprill 2016].

[23] M. Fowler, „P of EAA: Data Transfer Object," Martin Fowler, 2016. [Võrgumaterjal]. Available: http://martinfowler.com/eaaCatalog/dataTransferObject.html. [Kasutatud 2 Aprill 2016].

Lisa 1. Rakenduse moodulite realiseerimine

recruitmentdrive › web › build.gradle

web ×

```
buildscript {
    repositories {
        maven { url "http://repo.spring.io/libs-release" }
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:${springBootVersion}")
    }
}

apply plugin: 'spring-boot'

dependencies {
    compile project(':core')
    compile "org.springframework.boot:spring-boot-starter-thymeleaf:${springBootVersion}"
    compile "org.springframework.boot:spring-boot-starter-web:${springBootVersion}"
    compile "org.springframework.boot:spring-boot-devtools:${springBootVersion}"
    compile "org.springframework.security:spring-security-web:${springSecurityVersion}"
    compile "org.springframework.security:spring-security-config:${springSecurityVersion}"

    compile "commons-io:commons-io:${commonsIOVersion}"
    compile "javax.servlet:jstl:${jstlVersion}"
    compile "taglibs:standard:${jstlVersion}"

    compile "org.springframework:spring-webmvc:${springVersion}"
}
```

recruitmentdrive › persistence › build.gradle

persistence ×

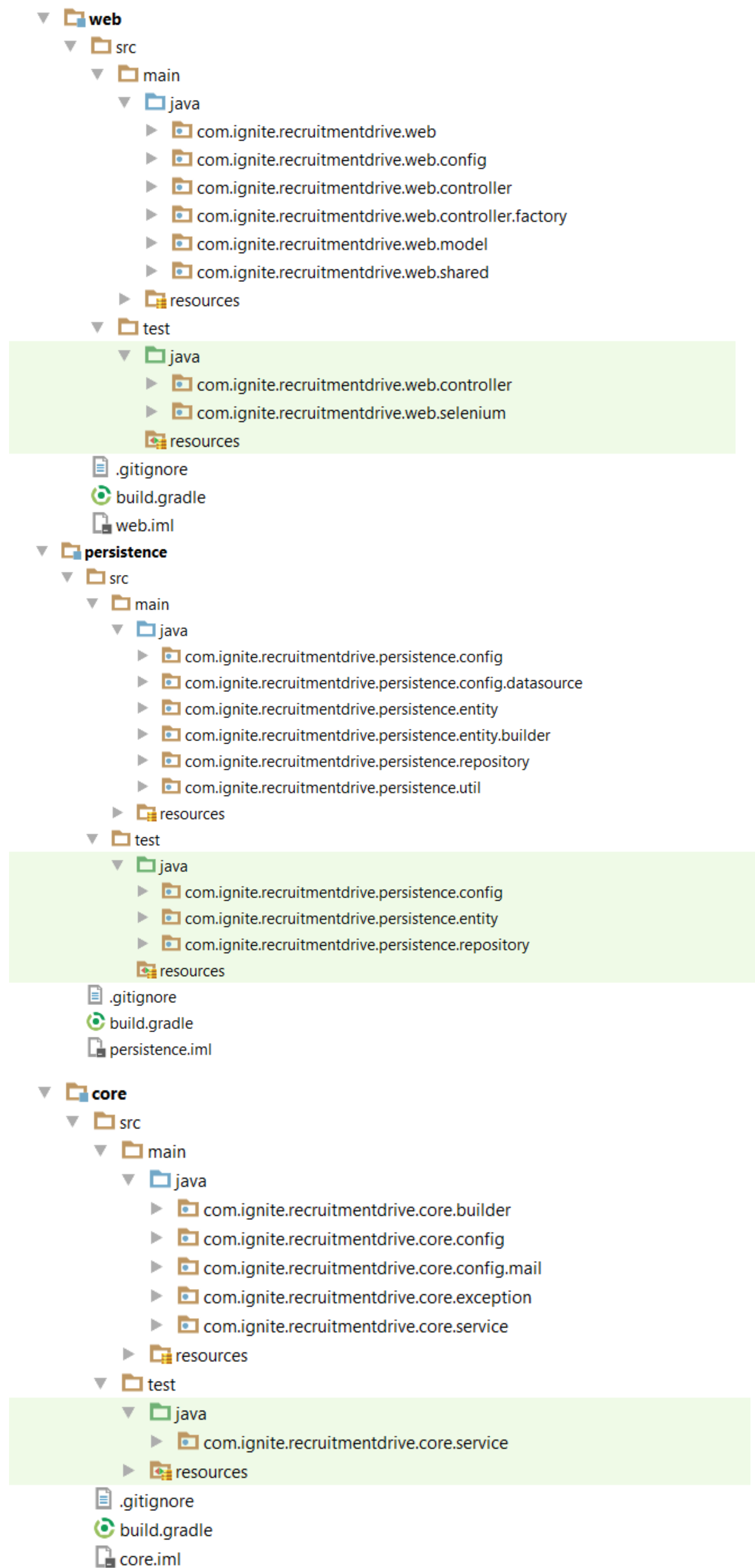
```
dependencies {
    compile project(':logging')
    compile "org.springframework:spring-orm:${springVersion}"
    compile "org.flywaydb:flyway-core:${flywayVersion}"
    compile "org.hibernate:hibernate-core:${hibernateVersion}"
    compile "org.hibernate:hibernate-validator:${hibernateValidatorVersion}"
    compile "org.hibernate:hibernate-entitymanager:${hibernateVersion}"
    compile "org.postgresql:postgresql:${postgresqlVersion}"
    compile "org.springframework.data:spring-data-jpa:${springDataJpaVersion}"
    compile "com.zaxxer:HikariCP:${hikariCPVersion}"
    compile "com.fasterxml.jackson.core:jackson-annotations:${jacksonAnnotations}"

}
```

recruitmentdrive › core › build.gradle

core ×

```
dependencies {
    compile project(':persistence')

    compile "org.thymeleaf:thymeleaf-spring4:${thymeleafVersion}"
    compile "org.apache.commons:commons-email:${commonsEmailVersion}"
    compile "commons-io:commons-io:${commonsIo}"
}
```

Lisa 2. Kandidaadi haldussüsteemi mooduli struktuur



Lisa 3. Kandidaadi haldussüsteemi tuletamise lähtekood

```java
public abstract class DataSourceProperties {

    @Resource
    private Environment env;

    public String getDriverClassName() {
        return org.postgresql.Driver.class.getName();
    }

    public String getJdbcUrl() {
        return "jdbc:postgresql://localhost:5432/" +
env.getRequiredProperty("db.database");
    }

    public String getUsername() {
        return env.getRequiredProperty("db.username");
    }

    public String getPassword() {
        return env.getRequiredProperty("db.password");
    }

}

@Configuration
@Profile("live")
@PropertySource("classpath:persistence-live.properties")
public class LiveDataSourceProperties extends DataSourceProperties {
}

@Configuration
@Profile("test")
@PropertySource("classpath:persistence-test.properties")
public class TestDataSourceProperties extends DataSourceProperties {
}

@Configuration
public class DataSourceConfig {

    @Inject
    private DataSourceProperties dataSourceProperties;

    @Bean
    public DataSource dataSource() {
        HikariDataSource dataSource = new HikariDataSource();

dataSource.setDriverClassName(dataSourceProperties.getDriverClassName());
        dataSource.setJdbcUrl(dataSourceProperties.getJdbcUrl());
        dataSource.setUsername(dataSourceProperties.getUsername());
        dataSource.setPassword(dataSourceProperties.getPassword());
        dataSource.setConnectionTimeout(3000);
        dataSource.setMaximumPoolSize(5);
        return dataSource;
    }
}
```

Lisa 4. Tootmismustri lähtekood

```
@Component
public class ActionFactory {

    @Inject
    private FinishJob finishJob;

    @Inject
    private RemoveJob removeJob;

    @Inject
    private PreviousStatusJob previousStatusJob;

    @Inject
    private NextStatusJob nextStatusJob;

    public ProjectAction action(String action) {
        switch (action) {
            case "remove":
                return removeJob;
            case "finish":
                return finishJob;
            case "nextStatus":
                return nextStatusJob;
            case "previousStatus":
                return previousStatusJob;
            default:
            throw new RuntimeException("Unknown action: " + action);
        }
    }
}

public abstract class ProjectAction {
    public abstract String execute(String projectId);

}

@Component
public class PreviousStatusJob extends ProjectAction {

    @Inject
    private JobService jobService;

    @Override
    public String execute(String projectId) {
        jobService.changeStatus(Long.valueOf(projectId), -1);
        return "redirect:/jobs";
    }
}
```

```java
@Component
public class NextStatusJob extends ProjectAction {

    @Inject
    private JobService jobService;

    @Override
    public String execute(String projectId) {
        jobService.changeStatus(Long.valueOf(projectId), 1);
        return "redirect:/jobs";
    }

}

@Component
public class RemoveJob extends ProjectAction {

    @Inject
    private JobService jobService;

    @Override
    public String execute(String projectId) {
        jobService.remove(Long.valueOf(projectId));
        return "redirect:/jobs";
    }
}

@Component
public class FinishJob extends ProjectAction {

    @Override
    public String execute(String projectId, String description) {
        //TODO: Logic is not implemented yet!
        return null;
    }
}
```

Lisa 5. Ehitusmustri lähtekood

```
public class IndividualBuilder extends Builder<Individual> {

    public static IndividualBuilder individual() {
        return new IndividualBuilder();
    }

    public IndividualBuilder setEmail(String email) {
        object.setEmail(email);
        return this;
    }

    public IndividualBuilder setFirstName(String firstName) {
        object.setFirstName(firstName);
        return this;
    }

    public IndividualBuilder setLastName(String lastName) {
        object.setLastName(lastName);
        return this;
    }


    public IndividualBuilder setCity(String city) {
        object.setCity(city);
        return this;
    }


    public IndividualBuilder setCountry(String country) {
        object.setCountry(country);
        return this;
    }

    public IndividualBuilder setCity() {
        object.setCity("Tallinn");
        return this;
    }


    public IndividualBuilder setCountry() {
        object.setCountry("Estonia");
        return this;
    }

}
```

```java
public abstract class Builder<T> {

    protected T object;

    public Builder() {
        Type type = ((ParameterizedType)
getClass().getGenericSuperclass()).getActualTypeArguments()[0];
        try {
            object = (T) ((Class) type).newInstance();
        } catch (InstantiationException | IllegalAccessException e) {
            throw new RuntimeException(e);
        }

    }

    public T build() {
        return object;
    }

}
```

Lisa 6. HomeController'i lähtekood.

```
@Controller
@RequestMapping("/")
public class HomeController {

    @RequestMapping(value = {"", "/"}, method = RequestMethod.GET)
    public String index() {
        return "home";
    }

    @RequestMapping(value = "login", method = RequestMethod.GET)
    public String login() {
        return "login";
    }

}
```

Lisa 7. DTO mustri lähtekood

```
public class FileForm {

    private MultipartFile file;

    @Valid
    private Candidate candidate;

    public MultipartFile getFile() {
        return file;
    }

    public void setFile(MultipartFile file) {
        this.file = file;
    }

    public Candidate getCandidate() {
        return candidate;
    }

    public void setCandidate(Candidate candidate) {
        this.candidate = candidate;
    }
}
```