TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Arvutiteaduse instituut
Võrgutarkvara õppetool

ITV40LT
Igor Pletnjov 135213IAPB

SÕLTUVUSTE SISESTAMISE RAAMISTIK JAVAS

bakalaureusetöö

Juhendaja: Ago Luberg
MSc
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: Igor Pletnjov

23.05.2016

Annotatsioon

Antud lõputöö põhiliseks eesmärgiks on leida lahendus sõltuvuste sisestamise disainimustri kasutamisele personaalsetes ja professionaalsetes tarkvaraprojektides. Sellise lahenduseni jõudmiseks teostatakse lõputöös olemasolevate lahenduste analüüs, kaalutakse laiendamise varianti ning käsitletakse täiesti uue raamistiku arendust.

Töös uuritakse ja võrreldakse olemasolevaid variante, analüüsitakse nende funktsionaalsust ja erinevaid disainiotsuseid. Selle analüüsi tulemusena leitakse, et eksisteerivad raamistikud ei vasta nõuetele ning nende laiendamine on sama keeruline kui uue raamistiku arendus.

Antakse ülevaade uue raamistiku funktsionaalsusest, disaini põhimõtetest ning arenduse käigus tekkinud probleemidest. Lisaks kirjeldatakse detailselt raamistiku sisemist struktuuri ning sõltuvuste sisestamise protsessis kasutatud algoritme. Tulemusena arendatud raamistik vastab kõikidele nõuetele.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 35 leheküljel, 5 peatükki, 10 joonist, 2 tabelit.

Abstract

"Dependency Injecton Framework In Java"

The basic objective of this thesis is to find a solution to using the dependency injection design pattern in personal and professional software projects. In order to complete this objective, an analysis of existing frameworks is performed, their extensibility is questioned and an entirely new framework is developed.

Existing frameworks are researched and compared by analysing their functionality and different design decisions. As a result of the analysis, it is concluded that existing solutions do not meet the requirements and extending them would be as difficult as the creation of a new framework.

An overview of the new framework is provided, including its functionality, design principles and issues during development. In addition, the internal structure of the framework and algorithms used in the dependency injection process are described in detail. The resulting framework meets all requirements.

The thesis is in Estonian and contains 35 pages of text, 5 chapters, 10 figures, 2 tables.

Lühendite ja mõistete sõnastik

Metaprogrammeerimine	Programmeerimise liik, milles kasutatakse andmetena programme, nende lähtekoodi või struktuuri.
Refleksioon	Programmi võime uurida ja muuta enda struktuuri käitusajal.
Annotatsioon	Atribuut, millega Javas on võimalik programmi struktuuri märgistada ja metaandmeid kaasa anda.
Raamistik	Tarkvaraarenduse tööriist, mis erinevalt teegist, suhtleb ise programmi teatud osaga ning manipuleerib seda.
IOC, Inversion of Control	Tarkvaraarenduse disainiprintsiip, mis näeb ette programmi voo mingi osa delegeerimist välisele raamistikule.
DI, Dependency Injection	Sõltuvuste sisestamine disainimuster, mis näeb ette objektide loomise eraldamist programmi käitumisest.
Classpath	Java virtuaalmasina ja kompilaatori parameeter, milles on loetletud programmis kasutatavate klasside ja pakettide asukohad.
Sõltuvuste graaf	Suunatud tsükliteta graaf, mis kujutab sõltuvusi objektide vahel.

Sisukord

1 Sissejuhatus 9
2 Metaprogrammeerimine 10
2.1 Metaprogrammeerimine Javas 11
2.2 Refleksiivne annotatsioonide töötlemine 11
2.3 Sõltuvuste sisestamine 12
3 Analüüs 14
3.1 Olemasolevad lahendused 14
3.2 Detailsem ülevaade 17
3.2.1 Dagger 17
3.2.2 Google Guice 18
3.2.3 Spring Framework 19
3.3 Arutelu 20
4 Arendus 22
4.1 Disain 22
4.2 Implementatsioon 24
4.2.1 Üldine ülevaade 25
4.2.2 Annotatsioonid 26
4.2.3 Sobivate klasside otsimine 27
4.2.4 Klassi elementide töötlemine 28
4.2.5 Sõltuvuste tuvastamine 29
4.2.6 Sõltuvuste lahendamine 29
4.2.7 Instantside loomine 32
4.2.8 Instantside sisestamine 33
5 Kokkuvõte 34
Kasutatud kirjandus 35

Jooniste loetelu

Joonis 1 Klassi Garage konstruktorid 28
Joonis 2 Lihtne sõltuvusgraaf 30
Joonis 3 Graaf tsükliliste sõltuvustega 30
Joonis 4 Sõltuvusteta kirjete pärimine 31
Joonis 5 Loendi kirje pärimine ja tema sõltlaste leidmine 31
Joonis 6 Sõltuvuse eemaldamine ja uue sõltuvusteta kirje lisamine 31
Joonis 7 Graafi sõltuvuste kontrollimine 31
Joonis 8 Topoloogiliselt sorteeritud graaf 32
Joonis 9 @InjectInstance annoteeritud väli 33
Joonis 10 @InjectInstance annoteeritud meetod 33

Tabelite loetelu

Tabel 1 Olemasolevate variantide võrdlus 16
Tabel 2 Uue raamistiku ja Springi annotatsioonide võrdlus 26

1 Sissejuhatus

Tarkvarateekide ja tööriistade kasutamine ning arendamine on olnud programmeerimise algusaastatest oluline osa arendajate elust, et enda ning teiste arendustegevust lihtsustada ja kiirendada.

Lähtudes oma tööelust, tekkis mul soov kasutada tuttavaid raamistikke personaalsete projektide otstarbeks. See osutus raskeks, sest paljud sellised raamistikud on enamasti orienteeritud suurte äriloogiliste teenuste arenduseks. Eriti huvitatud olin ma *dependency injection* ehk sõltuvuste sisestamise disainimustri kasutamises oma projektides, kuna olin sellega harjunud töötades Java Enterprise Editioni ja Spring Frameworkiga.

Sõltuvuste sisestamine võimaldas lihtsalt ja automatiseeritult luua ning häälestada objekte, andes objektile kaasa vaid vastavat märgistust ehk annotatsiooni. Mul tekkis huvi selle vastu, kuidas annotatsioone programmeerida ja sellist automatiseeritust saavutada.

Küsimuseks kerkis ka olemasolevate raamistike vastamine minu personaalete projektide nõuetele. Kas neid on mõistlik kasutada *desktop* rakenduste või mängude arendamiseks? Kuivõrd otstarbekas ja lihtne oleks uue raamistiku arendamine vastavalt minu soovidele?

2 Metaprogrammeerimine

Metaprogrammeerimiseks nimetatakse programmeerimise sellist valdkonda, kus kirjutatud arvutiprogrammid on võimelised andmetena kasutama ennast või teisi programme [1]. Sõltuvalt metaprogrammeerimise liigist, tunneks selline metaprogramm iseenda struktuuri ja suudaks seda teatud piirides vastavate rakendusliideste abil manipuleerida, saaks genereerida endale koodi juurde, või hoopis täielikult muuta oma lähtekoodi ja sellega ka oma käitumist. Need operatsioonid ei piirdu programmi käitusajaga vaid võivad toimuda ka enne programmi käivitamist ehk koodi kompileerimise ajal.

Metaprogrammeerimise abil on võimalik realiseerida loogikat, mis oleks muidu võimatu või väga keeruline ja aeganõudev. Metaprogrammeerimise otseste implementatsioonidena võib tegelikult vaadelda kompilaatoreid, sest nad võtavad sisse koodi ja teevad sellega vastavaid transformatsioone, et muuta see mingis keskkonnas käivitatavaks programmiks [2].

Keelt, milles metaprogrammid on kirjutatud, nimetatakse *metakeeleks*. Metakeelte poolt manipuleeritavate programmide keeli nimetatakse *objektikeelteks*. Keele omadust olla iseenda metakeeleks nimetatakse refleksiooniks ehk peegelduseks. See on oluline tunnus, mis võimaldab metaprogrammide kirjutamist paljudes keeltes [3].

Refleksioon on programmeerimiskeele omadus näha ja muuta iseenda struktuuri kirjutatud programmi käitusajal [4]. Struktuuri all mõeldakse andmeid, mis mingil moel kirjeldavad programmi seisundit. Sõltuvalt spetsiifilisest keelest, võib struktuur sisaldada klasse, meetodeid, parameetreid või metaandmeid.

Peegelduse puhul on tähtis programmi struktuuri metaandmete lugemine, mida tavaliselt saavutatakse atribuutide abil. Atribuutidele orienteeritud programmeerimist võib seega lugeda tavalise refleksiooni alamosaks. Nimelt võimaldavad atribuudid märgistada programmi struktuuri, nagu näiteks klasse ja funktsioone, andes kaasa mingisuguseid

metaandmeid. Refleksiooni abil on neid andmeid võimalik lugeda ja selle abil otsustada, kuidas antud struktuuriga käituda.

2.1 Metaprogrammeerimine Javas

Java on üks populaarsemaid programmeerimiskeeli maailmas [5], mille jaoks on aastate jooksul arendatud mitu suurt raamistikku nagu näiteks Spring Framework ja Hibernate. Sõltumata nende täpsest funktsionaalsusest, põhinevad nad suurel osal metaprogrammeerimisel, täpsemalt refleksioonil ja metaandmete käsitlemisel.

Metaandmete jaoks on Javas olemas atribuudisarnane struktuur nimega *annotatsioon*. Annotatsioonid said esmalt defineeritud Java spetsifikatsioonis "JSR 175 A Metadata Facility for the Java Programming Language*"* [6] ning on kasutusel olnud alates nende integratsioonist versiooni J2SE 5.

Annoteerida on võimalik klasse, liideseid, annotatsiooni-tüüpe, meetodeid, parameetreid, lokaalseid muutujaid ja pakette. Annotatsioon võib olla kättesaadav (*retained*) kompileerimise ajal ja käitusajal. Annotatsiooni võib ka täielikult ignoreerida, mis oleks mõistlik puhtalt informatiivsete annotatsioonide puhul. Metaandmetena tohib kaasa anda ainult Java mõistes konstante.

2.2 Refleksiivne annotatsioonide töötlemine

Refleksiooni võimaldab Javas kasutada pakett *java.lang.reflect*, mille abil saab uurida ja muuta Java klasside sisemist struktuuri. Annotatsiooni puhul on võimalik kasutada refleksiivseid operatsiooni siis, kui annotatsioon on programmi töö ajal kättesaadav. Seda defineerib meta-annotatsioon `@RetentionPolicy`, mille vaikimisi väärtuseks iga annotatsiooni puhul on kättesaadavus vaid kompileerimise ajal [7].

Refleksiivset annotatsioonide töötlust on võimalik kasutada iga Java klassi puhul, mis pärib liidest `AnnotatedElement`. Seda implementeerivad sellised klassid nagu `Class`, `Method`, `Field`, `Constructor`, `Package` ja `Parameter`. Loetletud elementidel põhineb tegelikult suur osa refleksiooni võimalustest Javas.

2.3 Sõltuvuste sisestamine

Sõltuvuste sisestamine on levinud tarkvara disainimuster, mille eesmärgiks on sõltuvuste lahendamise (*dependency resolution*) eraldamine programmi muust käitumisest [8]. On võimalik öelda, et antud disainimuster implementeerib *inversion of control* printsiipi sõltuvuste lahendamise jaoks. *Inversion of control* tähendab, et kogu või mingit osa programmi voost antakse välise raamistiku kätte selle asemel, et seda täies mahus programmi sees teostada [9]. Tavaliselt kutsub programm ise välja funktsioone, et lahendada ära programmi vooga seotud operatsioone, kuid *inversion of control* puhul kutsub sellised funktsioone välja väline raamistik.

Sõltuvuse sisestamine võimaldab sõltuva objekti muud käitumist eraldada tema loomise detailidest, mis annab võimaluse kirjutada madala sõltuvusega (*loosely coupled*) tarkvara. Tavalistes imperatiivsetes keeltes ei tähenda see tehnika muud kui automatiseeritud moel parameetrite edasiandmist objektile.

Sõltuvuste sisestamises eristatakse nelja peamist osapoolt [10]:

- Teenus - sisestatav objekt
- Klient - teenusest sõltuv objekt.
- Liidesed – defineerivad, kuidas klient teenust kasutada saab.
- Sisestaja - tavaliselt raamistik, mille ülesandeks on teenuseid konstrueerida ning neid õigesti klientidesse sisestada.

Need rollid on ainult sõltuvuste sisestamise operatsioonide skoobis ning ei määra kuidagi objektide kontekstuaalset tähendust [10]. Sama programmi üks ja sama klass võib olla nii klient kui teenus sõltuvalt täpsest sisestamise operatsioonist.

Kolm peamist viisi sõltuvuste sisestamise teostamiseks on [8]:

1. Setter-sisestamine - Kasutatakse nn. *setter* mustri meetodeid kliendi sõltuvuste lahendamiseks
2. Konstruktor-sisestamine - Kasutatakse kliendi konstruktorit, sõltuvused sisestatakse konstruktori otseste parameetrite kaudu
3. Liides-sisestamine - Kasutatakse ühist liidest, mida sisestaja kasutab, et kõiki liidest implementeerivatele klassidele sõltuvusi sisestada

Sõltuvuste sisestamine lihtsustab kliendiklasside häälestamist. Liideste abil sisestamine vähendab klasside sõltuvust täpsetest implementatsioonidest, mis lihtsustab refaktoreerimist ja ühiktestimist. *Dependency injection* suurendab koodi loetavust ning vähendab korduvat koodi, kuna loomisega seotud operatsioone delegeeritakse välisele raamistikule.

3 Analüüs

Kuna lõputöö ajendiks on sõltuvuste sisestamise kasutamine personaalsetes projektides, on probleemi lahenduseks kolm põhilist võimalust:

1. Mõne olemasoleva raamistiku kasutamine
2. Olemasoleva raamistiku funktsionaalsuse laiendamine vastavalt nõuetele
3. Uue raamistiku arendamine

Iga variandi puhul toome välja eelised ja puudused, kaalutledes teeme otsuse lõpliku lahendusviisi kohta.

Otsuse tegemisel aitab konkreetsete nõudmiste olemasolu, ehk põhilise funktsionaalsuse osas peaks ideaalne raamistik olema võimeline teostama:

1. Tavalise klassi loomist ja väärtuste sisestamist klassi väljadesse.
2. Liidese ja abstraktse klassi abil sisestatava implementatsiooni varjamist. See omakorda vajab päritavuse tuge, ehk täpset alamklassi saab varjata mistahes ülemklassi abil.
3. Anonüümsete klasside loomist ja sisestamist vastavate liideste abil. Loomine võiks toimuda nii meetodite abil kui ka klassi välju kasutades.
4. Instantsi loomist kasutades parametriseeritud konstruktorit või muu sisestatava klassi meetodit.
5. Meetodite abil ehk *setter*-tüüpi sisestamist

Raamistiku disaini ja kasutuse puhul on oluline lihtsus, skaleeritavus ja laienduste teostamise võimalused. Sellised nõudmised on enamasti subjektiivsed ja võivad väljenduda mitmel erineval viisil.

3.1 Olemasolevad lahendused

Sõltuvuste sisestamise disainimuster on Javas leidnud üsna laia kasutust ning selle jaoks on aastate jooksul arendatud mitu raamistikku. Eriti palju on seda kasutatud suuremahulise äriloogikale orienteeritud tarkvara loomisel, tavaliselt lähtudes platvormist, mille on defineerinud Java Enterprise Edition. Alates versioonist Java EE 6

on olemas standardne sõltuvuste sisestamise annotatsioonide liides, mille on määratlenud standard “JSR 330 Contexts and Dependency Injection for the Java EE platform” [11] ehk lühidalt CDI.

Tegemist on Java Community Process organisatsiooni poolt soovitatud standardiga Java EE skoobis, mille Oracle formaalselt kinnitas ja platvormi integreeris. Selle järgimine raamistiku arendamisel ei ole mingil juhul kohustuslik.

Antud standard defineerib annotatsioone, mis katavad ära põhilise osa sõltuvuse sisestamise kasutusjuhtudest. Liides näeb ette tavalist instantside loomist ja sisestamist, nimesid omavate instantside deklareerimist ja sisestamist, skoobi operatsioone ja konfigureeritavat sisestajat. Sellest lähtuvad ja selle annotatsioonide kasutamist toetavad raamistikud nagu Google Guice [12] ja Spring Framework [13].

Javas on tõesti palju erinevaid sõltuvuste sisestamisele orienteeritud raamistikke, seega tutvume veidi lähemalt viie põhilise variandiga:

- Spring Framework 4.2.6 [13]
- CDI (Weld 1.2.4) [14]
- Google Guice 4.0 [12]
- Dagger 1.2.5 [15]
- PicoContainer 2.14 [16]

Neid raamistikke on võimalik võrrelda puhtalt funktsionaalsuse poolest. Samas võib võtta luubi alla igaühe omapärasid ja võrrelda erinevaid disainiotsuseid.

Tabel 1 näitab osa teatud funktsionaalsuse olemasolust igas raamistikus ning nende liidestust standardiga

Tabel 1 Olemasolevate variantide võrdlus

	Class	Interface/ Abstract	Annotation-type	Method/ Constructor	Follows JSR-330
Dagger	Yes	Yes	No	Partial	Partial/Unknown
Guice	Yes	Partial	Unknown	Yes	Partial
Spring	Yes	Yes	Partial/Unknown	Yes	Partial
PicoContainer	Yes	Yes	No	Yes	No
CDI (Weld)	Yes	Yes	No	Yes	Yes

Lähtudes tabelist **Tabel 1**, saab järeldada, et enamik teadaolevatest implementatsioonidest rahuldavad vähemalt poole nõudmistest. Kuid mõne kriteeriumi puhul tekib küsimus, kuidas programm käitub spetsiifilises olukorras, ehk kas tõesti kõik mõistlikud kasutusjuhud on raamistiku poolt rahuldatud.

Selle küsimuse tekitab nii ebaselge annotatsioonide disain kui ka üldine dokumentatsiooni puudulikkus. Spring DI puhul on näiteks annotatsiooni-tüüpi instantsieerimine näiliselt võimalik, kuid ei ole dokumenteeritud, kuidas selline protsess toimub ja mis täpsemalt selle käigus luuakse. Ka programmi käivituse ajal pole selge, mis sellise tüübi instantsiga juhtub.

Üheks oluliseks punktiks on kindlasti JSR-330 spetsifikatsiooni järgimine, mis ühelt poolt lihtsustab refaktoreerimise mõne varasema implementatsiooni pealt, teiselt poolt aga piirab oluliselt raamistiku funktsionaalsust. Ideaalis, kui kaks raamistiku implementeerivad CDI liidest, on võimalik neid omavahel ära vahetada, muutes vaid vastavat programmi sõltuvust. Nii saaks mõnda vana rakendust lihtsalt ja kiiresti migreerida uuemale, rohkem eelistatud implementatsioonile.

Samal ajal ei ole võimalik muuta ega oluliselt laiendada standardsete annotatsioonide disaini ja ettenähtud funktsionaalsust. Google Guice'i dokumentatsioonis on arendajatele soovitatud, et nad kasutaksid siiski Guice'i annotatsioone, kuna nad katavad oluliselt laiema osa võimalikest kasutusjuhtudest.

Järgnevalt uurime kolme erineva suurusega raamistiku spetsiifikat - Dagger, Google Guice ja Spring.

3.2 Detailsem ülevaade

Google Guice, Dagger ja Spring DI on erineva suuruse, skaleeritavuse ja mõnes mõttes ka erinevate otstarvete jaoks loodud sõltuvuste sisestamise raamistikud.

3.2.1 Dagger

Võrreldavatest teekidest kõige väikesemahulisem ja lihtsaim, Daggerit kasutatakse laialt Android arendamise maailmas.

Antud raamistik kasutab oma sõltuvuste loomisel ja sisestamisel nn. objektide graafi, mida on võimalik objektina kätte saada lähtuvalt sisestatavast moodulist. Sisestamise alustamiseks on objekti graafi väljakutse programmi soovitud alguspunktis [15]. Selline disain on mugav, sest arendaja saab vabalt juhtida kogu sisestamise protsessi ning selle alguspunkti.

Dagger defineerib endas kahte annotatsiooni - `@Module` ja `@Provides`. Esimest kasutatakse objektigraafis sisaldavate objektide ehk moodulite loomiseks, viimast sisestatavat väärtust loovate meetodite ehk varustajate annoteerimiseks. Selline annotatsioonide vähesus on tingitud sellest, et raamistik on osaliselt liidestatud JSR-330 annotatsioonidega. Sisestamise jaoks on vaja kasutada `@Inject` ning ühekordsete moodulite täpsustamiseks annotatsiooni `@Singleton` [15].

Raamistikus toimub sõltuvuste valideerimine ja vigade tuvastamine juba kompileerimise ajal, mis teeb arenduse oluliselt kiiremaks, sest rakenduse käivitamine sisestamisprotsessi testimiseks võib muidu olla ajakulukas. Sellist funktsionaalsust

saavutab Dagger standardse annotatsioonide töötlemise liidese ehk Java Annotation Processor API abil.

Puudub annotatsioonide lihtsus ehk vähesus, kuna sõltutakse otseselt standardsetest annotatsioonidest, kuid siiski ei toetata kogu standardset funktsionaalsust. Raamistik kasutab välist konfiguratsiooni, et defineerida konkreetne objekti graaf. Puudu on meetodite ja klassi väljade abil sisestamise funktsionaalsus. Daggeri klasside laiendamine on üsna keeruline ja enamus liideseid on mõeldud raamistiku sisemise arenduse otstarbeks.

3.2.2 Google Guice

Guice on Google poolt arendatav vaba lähtekoodiga raamistik, mida firma kasutab ka sisemiselt. Guice on Daggerist tunduvalt suurem ja funktsionaalsuse poolest rikkalikum, kuid disaini poolest üsna sarnane.

Antud raamistik on täielikult liidestatud standardse spetsifikatsiooniga, kuigi tema enda annotatsioonid erinevad veidi JSR-330 omadest. Sarnaselt Daggerile kasutab Guice moodulite ja *Provider*'ite ehk varustajate stereotüüpe, kuid mitte lihtsate annotatsioonide kujul [12]. Arendaja peab ise klassi tasemel defineerima moodulid, mille abil ta seob omavahel kokku liideseid ja neid implementeerivaid klasse. Moodulid moodustavad objektide graafi, mille abil sisestaja oma ülesandeid täidab.

Moodulite sees kasutatakse *Binder*'eid ehk sidujaid, mis ütlevad sisestajale ette, millist tüüpi objekti kuhu ja mis tingimustel sisestada [12]. Selline funktsionaalsus loob võimaluse arendada väga põhjalikku konfiguratsiooni rakenduse jaoks, defineerides abistavaid sidumisannotatsioone ning luues atribuutide abil täpsed sisestamise reeglid. Ka selliste klasside nagu `String` ja `Integer` instantse on vabalt võimalik siduda Guice annotatsiooniga.

Guice põhiline puudus ongi tegelikult tema suur sõltuvus täpsest koodisisesest konfiguratsioonist. Kuigi raamistik toetab eraldi ka käsitsi sisestamist, võib tegelikult automaatse sisestamise protseduuri samuti käsitsi sisestamisek nimetada, sest automaatne protsess toimub ainult vastavate sidujate sidumismeetodite väljakutsel. See loob suure probleemi skaleeritavuse osas, moodulid paisuvad suuremate rakenduste

puhul väga mahukateks, mille tõttu on raske neid hallata ja neile uut funktsionaalsust lisada. Lisaks pole raamistik iseseisev, vaid sõltub Google poolt arendatud Guavast ja teistest teekidest.

Annotatsioone ja nendes sisladavaid atribuute on palju, mis teeb raamistiku kasutamise veidi keerukaks. Guice pakub peaaegu täielikku liidestust standardsete annotatsioonidega, mis teeb refaktoreerimise lihtsaks.

3.2.3 Spring Framework

Spring Framework on ulatuslik raamistik *enterprise* rakenduste jaoks, loodud alternatiivina järjest keerukamaks muutuva Java Enterprise Editionile. Springil on oma sõltuvuste sisestamise moodul ehk üldisemalt *inversion of control* konteiner, mis erineb oluliselt Daggeri ja Guice'i mudelist.

Sõltuvalt Springi versioonist ja raamistiku täpsetest moodulitest, on Springi sõltuvuste sisestamist võimalik juhtida kas väliste XML failide abil või annotatsioonidega. Sõltumatult kasutatud versioonist, on IoC konteineri keskseks liideseks `BeanFactory`, mis loob ja haldab Springi *bean*'e [13]. *Bean*'id on objektid, mis osalevad sõltuvuste sisestamise protsessis. Igal *bean*'il on oma nimi, skoop ja muud elutsükli haldamisega seotud metaandmed, mille oskab lugeda ja kasutada vastav `BeanFactory`.

Vaatleme eelkõige Springi annotatsiooni-põhist mudelit, kus kesksеteks annotatsioonideks on `@Component`, `@Autowired` ja `@Bean`. Komponent on mistahes klass, mida Spring peab looma konstruktori abil, *factory* meetod vajab komponentide puhul liidese kasutamist [13]. Sellel annotatsioonil on olemas ka laiendid ehk stereotüübid, nagu `@Service` ja `@Repository`, kuid tegemist on pigem informatiivsete täpsustustega, millest lähtuvad mõned teised raamistikud.
`@Bean` annotatsiooni abil toimub meetodi põhine beanide loomine, ehk beani luuakse meetodi parameetrite ja teiste klassi elementide abil ning tagastatakse meetodi väljakutse tulemusena.

Sisestamine ehk *autowire*'imine võib toimuda annoteerides selleks vastavat klassi, konstruktorit, meetodit või välja. Seda protsessi võib ka mõne beani jaoks valikuliseks

muuta, kui kasutusjuht sellist situatsiooni ette näeb. Sisestamisannotatsiooni üheks laiendiks on `@Qualifier`, mis lubab täpsustada sisestatava beani nime.

Springi põhiliseks eeliseks Daggeri ja Guice'i ees on tema lihtsus - raamistik ei nõua sõltuvuste käsitsi täpsustamist või raamistiku häälestamist. Sisuliselt kogu protsess toimub taustal, arendaja ülesandeks on vaid vastavate elementide annoteerimine. Spring ei sõltu oma annotatsioonide puhul standardist, kuid annotatsioone on siiski palju. Peamiseks puuduseks on Springi üldine suuremahulisus. Raamistik on keeruliselt üles ehitatud, teda on raske laiendada ja ta sobib põhiliselt suurte veebi-kesksete projektide jaoks.

Probleeme tekitab ka Springi veahaldus ja juhuslik käitumine, kui mõne klassi täpse nimega instants ei eksisteeri, siis sisestatakse selle asemele mistahes teine instants samast klassist. Selline käitumine ei ole täpselt dokumenteeritud ja nõuab kasutajapoolset käsitsi testimist. Erind visatakse ainult kriitilistes situatsioonides, kus ei leita sobivaid alternatiive. Juhusliku käitumise situatsioone on kahjuks palju ja nende *debug*'imine raiskab aega.

Üldiselt pooldan mina Springi täielikult automatiseeritud mudelit, kus arendajal on vaja tunda vaid annotatsioone. Väline konfiguratsioon on tülikas ja loob probleeme skaleeritavuse küsimustes. Sisestamise häälestus võiks olla detsentraliseeritud ja täielikult annotatsiooni põhine.

3.3 Arutelu

Analüüsist on selgunud, et ükski olemasolev lahendus ei vasta täielikult püstitatud nõuetele. Kõikidest variantidest on puudu väljade abil instantside loomine ning üksikutest lahendustest ka muu osa funktsionaalsusest. Nende laiendamine sellise baasfunktsionaalsuse lisamiseks on võimatu või äärmiselt keerukas. Oleks vaja raamistiku sisemise struktuuri, sisestamise protsessis kasutatud algoritmide, andmestruktuuride ja objektide muutmist ehk *fork*'imist. See nõuab omakorda, et arendaja tunneks kogu raamistikku väga põhjalikult.

Selle põhjal on võimalik järeldada, et täiesti uue raamistiku loomine on laiendamisega vähemalt samaväärne. Uus implementatsioon oleks siiski võimeline rahuldama kõiki nõudeid ja lisaks andma vabadust selle disaini ja kasutusviisi suhtes. Algoritmide valik, annotatsioonide disain, sisemise struktuuri arhitektuur, skaleeritavus ja laiendatavus on kõik arendaja otsustada. Ka tulevikus uute nõudmiste tekkimisel on arendaja võimeline kiiresti tegema oma raamistikule täiendusi ja parandusi, mistõttu osutub see variant kõige sobilikumaks.

4 Arendus

Arendatud raamistik peab rahuldama kõiki analüüsi käigus püstitatud eesmärke, ehk lähtuda olemasolevate variantide puudustest ja pakkuda nõudmistele vastavat lahendust sõltuvuste sisestamiseks.

Antud peatükkis kirjeldan ja seletan lahti raamistiku disainiprintsiipe, sõltuvuste sisestamise protsessi implementatsiooni, iga protsessis osaleva klassi rolli ja tähtsust, ning kasutatud algoritme. Täpsemalt tuleb luubi alla annotatsiooni-põhine klasside pärimine, sisestamiseks vajalike klasside loomine ja salvestamine, instantside unikaalne identifitseerimine ning loomise ja sisestamise õige järjekorra tagamine. Arenduse käigus lähtusin üldistest Java pakettide ja klasside nimetamistavadest ja *clean code*'i loetavuse standarditest.

4.1 Disain

Raamistiku põhiline disainifilosoofia peitub lihtsuses ja konfigureeritavuses, kuid väga mitmete otsuste puhul lähtutakse Springi disainist ja sisestamise üldisest mudelist.

Projekti arendatakse põhiliselt teiste arendajate jaoks, mis tähendab, et silmas tuleks pidada skaleeritavust, laiendatavust ja erinevate variantide olemasolu raamistiku kasutamise osas. Arendajatel võib olla soov kasutada raamistikku mistahes projekti jaoks, kas pisikese isikliku rakenduse või suure *enterprise* veebiliidese arendamisel. Seejuures tuleks anda arendajale rohkelt valikuvõimalusi - ehk raamistiku tööd peaks olema võimalik parameetrite abil juhtida ja optimeerida. See nõuab omaette, et raamistiku põhiosad oleksid mingil määral laiendatavad - arendaja võiks ise endale sobiliku sisestamisalgoritmi arendada või hoopis muuta klasside pärimise metoodikat.

Laiendatavuse tagamiseks on raamistiku põhikomponentidel liidesed, millel on vähe implementatsiooni-spetsiifilisi nõudmisi.

Vastavalt lihtsuse nõudmisele on raamistikul ainult kaks põhilist annotatsiooni - protsessis osalevate objektide loomiseks ning objektide sisestamiseks.

Antud disainiotsuse puhul on oluline, et annotatsioonidel oleksid parameetrid, mis kompenseeriksid sellist vähesust. Parameetritel peaksid olema mõistlikud vaikimisi väärtused, ehk nende väärtustamine on valikuline.

Üle kogu raamistiku kasutatavaid korduvaid meetodeid hoitakse eraldi staatilistes *utility* klassides, mida saaksid kasutada ka teised arendajad laiendamistööde käigus. Selline tegevus parandab ka koodi loetavust.

Protsessi ülevaade ja probleemid

Disaini põhimõtete järgimisega seoses tekkis implementatsiooni ajal mitu raskust:

- Sõltuvuste sisestamise protsessi kulgemine, selle osadeks jagamine ja optimeerimine
- Laiendatavuse ulatus enamasti monoliitse protsessi puhul
- Andmete hoidmine ja objektide identifitseerimine

Esialgsel hinnangul võis sõltuvuste sisestamise protsessi jagada üldistes joontes kaheks - objektide loomiseks ja objektide sisestamiseks. Esimeses etapis luuakse lihtsad sõltuvusteta instantsid, nii konstruktorite kui meetodite abil. Sellele järgnev klassi väljade sisestamine toimuks ainult eelneva protseduuri tulemusena tekkinud instantsidega.

Sellise disaini tegi raskeks parameetriseeritud konstruktorite ja meetodite ehk sõltuvustega instantside olemasolu. Selle võis ära lahendada viies nende loomise ja sõltuvuste lahendamise protsessi lõppu, mille naiivseks eelduseks oli, et kõik nende loomiseks vajalikud instantsid olid juba olemas. Klasse oli tarvis jagada “lihtsateks” ja “keerulisteks” sõltuvuste olemasolu järgi ning käsitsi järjestada nende tekkimist ja sisestamist. Selline voog kujunes liiga keeruliseks ja otsustasin arusaadavama ning homogeensema lahenduse nimel, kus omavaheliste sõltuvuste lahendamiseks kasutatakse algoritmi, mida rakendatakse kõikidele loomises osalevatele objektidele.

Sõltuvuste sisestamise õige järjekorra tagamiseks on igal juhul tarvis abistavat algoritmi. Sõltuvaid objekte on mõistlik kujutada graafina ning nende lahendamiseks kasutada topoloogilise sorteerimise algoritmi.

Nõudmistest ja protsessi kulgemise analüüsist kujunes välja sõltuvuste sisestamise lõplik protsess:

1. Sobivate elementide leidmine
2. Elementide introspektsioon, sõltuvuste loomine iga elemendi jaoks
3. Sõltuvuste graafi koostamine elementidest ja sõltuvustest
4. Topoloogiline sorteerimine õige järjekorra tagamiseks
5. Instantside loomine ja sisestamine
6. Väljade sisestamine
7. Meetodite sisestamine

Neid operatsioone teostatakse sõltumata sellest, kas elemendil on loomise ajal väliseid sõltuvusi või mitte. Väljade ja meetodite sisestamise etapp on siiski omaette osa, eraldamaks instantse loovat sisestamist ja instantse mitteloovat sisestamist. Lisaks on seda osa raske efektiivselt integreerida üldisesse algoritmi, mis igas etapis eeldab objektide loomist.

Raamistiku laiendatavuse osas tekkis küsimus, kui suurt osa raamistikust teised arendajad ise muuta tohiks. Kui teostada kõik osad protsessist liidestega, siis muutub raamistik eeskätt spetsifikatsiooniks, mille jaoks antud lõputöö loob vaikimisi lahenduse. Otsustasin siiski keskenduda sõltuvuste sisestamise raamistiku konkreetse implementatsiooni arendamisele. Lisaks saab sellise monoliitse protsessi puhul laiendada ainult mõnda iseseisvat etappi.

Liideste, abstraktsete klasside ja päritavuse käsitlemine oli samuti väike raskus disaini ja implementatsiooni ajal. Antud tüübid võiksid näiliselt osaleda sõltuvuste sisestamises, kuid reaalselt esindaks neid vastavad alamklassid. Sellist olukorda lahendas Google Guice välise konfiguratsiooni abil, kuid antud raamistikus ei ole otstarbekas sellist süsteemi arendada. Oleks vajalik detsentraliseeritud lähenemine, ehk alamklassi häälestus otse annotatsiooni sees nii loomise kui sisestamise puhul.

4.2 Implementatsioon

Raamistik on kirjutatud programmeerimiskeeles Java 8, mis tänu lambda funktsioonidele ja Stream API-le tegid arenduse mugavamaks.

Maketool'ina on kasutatud Gradle'it, et lihtsustada kompileerimist, testide käivitamist ja raamistiku pakkimist arhiivifaili. Ühiktestimiseks kasutatakse JUnit raamistiku. Raamistikku ise on soovitatav kasutada Java arhiivi ehk JAR failina, mis tuleb oma projektile lisada. Pakitud raamistikku on võimalik kasutada ka teistes JVM keeltes nagu Scala või Clojure, mis toetavad Java arhiivide importimist ja kasutamist.

Lähtekoodi versioonihalduseks kasutatakse Git'i, mille peamine *remote* asub Atlassiani BitBucket teenuses[1]. Raamistik on avatud lähtekoodiga.

Järgnevalt kirjeldatakse, kuidas autor implementeeris sõltuvuste sisestamise protsessi voogu ja teisi disaini käigus tekkinud mõtteid ning printsiipe.

4.2.1 Üldine ülevaade

Sõltuvuste sisestamise kesksek klassiks raamistikus on `InjectionProcessManager`, mis vastutab protsessi häälestamise, käivitamise ning suurema osa loogika eest. Antud klass ühendab kõiki teisi protsessiga seotud komponente.

Andmete hoidmiseks protsessi ajal kasutatakse klassi nimega `InstanceRepository`, mis kujutab endast kõiki sisestamise tulemusena loodud instantside mälusisest andmebaasi, kuhu on iga hetk võimalik uusi instantse salvestada või olemasolevaid pärida. Seda klassi ei saa oluliselt laiendada ega asendada, ta on protsessi nurgakiviks.

Sõltuvuste lahendamisel kasutatakse sõltuvuste graafi, mille vastavaks liideseks on `DependencyGraph`. See on esimene protsessi vahetatavatest komponentidest, mille puhul raamistik pakub vaikimisi implementatsiooni.

Classpath'ist klasside pärimiseks kasutab raamistik liidest `ClasspathScanner`, mille ainsaks nõudmiseks on kõikide protsessi jaoks sobivate klasside tagastamine. Liidese vaikimisi implementatsioon sisaldab rohkem häälestamise võimalust.

[1] https://bitbucket.org/IgorPletnjov/injectionengine/src

4.2.2 Annotatsioonid

Raamistik kasutab ainult kahte eraldi annotatsiooni - `@Injectable` protsessis osaleva objekti loomiseks ja `@InjectInstance` objekti sisestamiseks. Annotatsioonid toetavad kõiki nõudmiste katmiseks vajalikke klassi elemente. Võrreldes analüüsi käigus uuritud variantidega, võtavad need kaks iga raamistiku peale kokku 2 kuni 3 annotatsiooni. Tabelis Tabel 2 on uue raamistiku annotatsioonid võrreldud Spring Frameworki vastavate annotatsioonidega.

Tabel 2 Uue raamistiku ja Springi annotatsioonide võrdlus

Spring Framework	Uus raamistik
@Autowired @Qualifier	@InjectInstance
@Component @Bean	@Injectable

Sisemise struktuuri ehk parameetrite poolest on nad väga sarnased. Mõlemad toetavad nime ja alamklassi täpsustamist sisestavate objektide puhul. Iseseisvalt ei realiseeri antud annotatsioonid mingit funktsionaalsust, nende lugemise ja interpreteerimise eest vastutab sõltuvuste sisestamise protsess.

Kõige suurem tehniline erinevus annotatsioonide vahel on `@InjectInstance` piirang, mille kohaselt ei tohi antud annotatsiooni kasutada tüüpide ehk klasside annoteerimiseks.

Alamklasside lahendamiseks ehk `assignableClass()` parameetri töötlemiseks on raamistikus eraldi liides `AssignableResolver`. Antud liides vastutab selle eest, missugust alamklassi sisestamises või loomises tegelikult kasutada, juhul kui vastav parameeter on annotatsioonis väärtustatud. Liidesel on kaks implementatsiooni, `SingleAssignableResolver` mis pärib ainult esimese annotatsiooni parameetrites täpsustatud klassi, ning `MultiAssignableResolver`, mille algoritm võimaldab sügavamalt ehk mitme põimitud annotatsiooni pealt lõplikku päritavat klassi leida.

Järgnevalt kirjeldatakse detailselt kuidas toimib kogu protseduur. Seletatakse üksikasjalikumalt lahti kõigi komponentide rolli, funktsionaalsust ja omavahelisi seoseid.

4.2.3 Sobivate klasside otsimine

Esimeseks sammuks protsessis on sobivate klasside leidmine, mis tähendab kõiki `@Injectable` annoteerituid klasse. Mitte-annoteeritud klasse, millel on annoteeritud sisemised elemendid, ignoreeritakse täielikult. Sellise protseduuri teostamiseks on tarvis *classpath*'i skanneerimine *classloader*'ite abil. Järgnevalt kirjeldatakse skanneri vaikimisi implementatsiooni `DefaultClasspathScanner` funktsionaalsust.

Esimese sammuna saame kätte süsteemi classloaderi ning turvalisuse jaoks ka kontektsi ehk antud lõime classloaderi, sest sisestamisprotsess ei pruugi töötada *main* lõime peal, kuigi see on soovitatud. Antud classloaderid töötavad URL-ide põhiselt, nende käest võib pärida pakette ja klasse sisaldavate põhikataloogide asukohti.

Põhikataloogideks loetakse ka *classpath*'is olevaid Java arhiive ehk JAR faile. Skanneri globaalne parameeter `scanJars` määrab, kas annoteeritud klasse tuleks otsida ka JAR-ide seest. Vaikimisi ei ole see funktsionaalsus aktiivne, aga seda võib muuta kasutades `withJarsIncluded` meetodit.

Saadud põhikataloogid itereeritakse *stack*'i-põhise sügavotsingu abil läbi, salvestades igat sobivat klassi loendisse, mis tagastatakse otsingu väljundina. JAR-ide puhul päritake nende sisemised elemendid ja kasutades uut *classloader*'it laetakse klassid sisse, et neid saaks refleksiivselt uurida ja manipuleerida. Skannerile on võimalik ette anda `withBasePackages` parameeteriga, millistes pakettides olevaid klasse tuleks otsinguna tagastada. See võib oluliselt kiirendada otsingu protseduuri, sest skanner ignoreeriks klasse, mis ei asu nendes etteantud pakettides.

Arenduse käigus oli kaalutud ka refleksiooni kasutamine classloaderite asemel, kuid see osutus viljatuks. Staatiline refleksiivne meetod `Package.getPackages()`, mis tagastab kõiki teadaolevaid pakette, sõltub siiski kasutatava lõime classloaderi tööst.

4.2.4 Klassi elementide töötlemine

Järgmise sammuna on tarvis refleksiivselt uurida klasside elemente, nimelt annoteeritud konstruktoreid, meetodeid ja välju.

Esmalt võetakse kõik `@Injectable` annoteeritud klassid ning leitakse iga ühe seest instantsi loomiseks sobiv konstruktor. Sobivad nii mitte-annoteeritud argumentideta kui ka annoteeritud argumentidega konstruktorid. Kui ühtegi sellist ei leita, peatatakse protsess ja kasutajale antakse probleemi kirjeldav veateade. Annoteeritud konstruktorid on eelistatud, kuid rohkem kui ühe sellise variandi leidmisel palutakse kasutajalt täpsustust. Joonisel Joonis 1 on toodud näide `Garage` klassi konstruktoritest, kus antud situatsioonis valitakse sisestamiseks `@Injectable` annoteeritud konstruktor.

```
@Injectable
public class Garage {
    Vehicle vehicle;

    public Garage() { }

    @Injectable
    public Garage(Vehicle vehicle) { this.vehicle = vehicle; }
}
```

Joonis 1 Klassi Garage konstruktorid

Kõik leitud konstruktorid lisatakse sõltuvuste graafi, luues selle jaoks *GraphEntry*-tüüpi objektid, mis sisaldavad konstruktorit ja tema abil loodud objekti identifikaatorit. Konkreetsed objektid muidugi käesoleva tegevuse ajal ei eksisteeri, vaid kasutatakse nende tulevaseid identifikaatoreid. Identifikaator ise põhineb sõnel, mis koosneb instantsi klassi kanoonilisest nimest ja annotatsioonist võetud nimest, mille vaikimisi eraldajaks on kooloni sümbol. Nimi võib olla kas annotatsioonides kasutaja poolt täpsustatud `value()` parameetri väärtus või konstandis defineeritud vaikimisi väärtus *unnamedInstance*.

Järgmises meetodite töötlemise funktsioonis leitakse esmalt kõik sobivalt annoteeritud meetodid ning korratakse sama protseduuri mis konstruktorite puhul. Meetodite puhul kontrollitakse lisaks, kas meetod on võimeline käivitamise tulemusena looma objekti. Sisuliselt vaadatakse, kas tagastatava objekti klassiks ei osutu `void.class`. Seejärel lisatakse meetod graafi, koos oma vastava metainformatsiooniga.

Samu operatsioone tehakse ka väljade puhul. Iga `AccessibleObject` alamklassi puhul on protsess veidi teistsugune, mille tagajärjel loodi kolm erinevat abistavat meetodit.

4.2.5 Sõltuvuste tuvastamine

Andmete kogumisele järgnevaks protsessiks on reaalsete sõltuvuste lisamine sõltuvusgraafi. Seda ei ole tarvis alamklassi-haaval teha, sest meetoditel, väljadel ja konstruktoritel ei ole piisavalt palju programmaatilisi erinevusi. Küll on iseärasusi selles, kuidas käitutakse parametriseeritud elementide, ehk konstruktorite ja meetoditega, ning klassi väljadega. Objekte üldistatakse klassi `Executable` abil, mis võtab kokku meetodeid ja konstruktoreid, sest need on Java mõistes "käivitatavad" elemendid. Nendele lisatakse sõltuvustena graafi kõik vastavad parameetrid. Parameetrite annoteerimine `@InjectInstance`'ga võimaldab teha nime ja alamklassi täpsustusi, aga see ei ole kohustuslik.

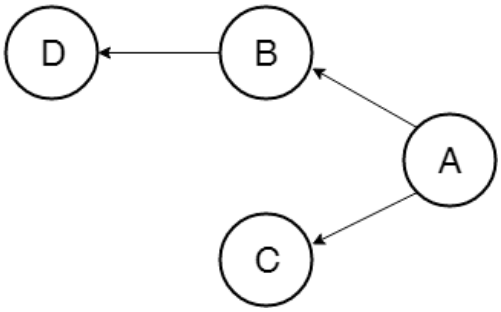
Igal elemendil peale konstruktori on ka sõltuvus oma enda deklareeritava klassi suhtes. Kui klassi instantsi ei eksisteeri, ei ole võimalik tema meetodi või välja abil luua instantse sisestamiseks. Graafi sorteerimisel peab kindlasti arvestama sellega, et klass ei asuks temas deklareeritud elemendist eespool. Kahjuks ei defineeri ülemklass `AccessibleObject` ühtset viisi deklareeritava klassi pärimiseks, seda peab tegema alamklasside kaupa.

4.2.6 Sõltuvuste lahendamine

Kui graaf on protsessis osalevate objektidega ning nende sõltuvustega täidetud, kasutatakse topoloogilise sortimise algoritmi, et saavutada õige järjekord sõltuvuste lahendamiseks.

Sõltuvuste graaf on suunatud tsükliteta graaf, mis kujutab tippudena objekte ja servadena nendevahelisi sõltuvusi [17].

Joonis 2 kujutab lihtsat sõltuvusgraafi, kus tipp A sõltub B-st ja C-st ning B sõltub D-st. Tippud D ja C on sõltuvusteta.

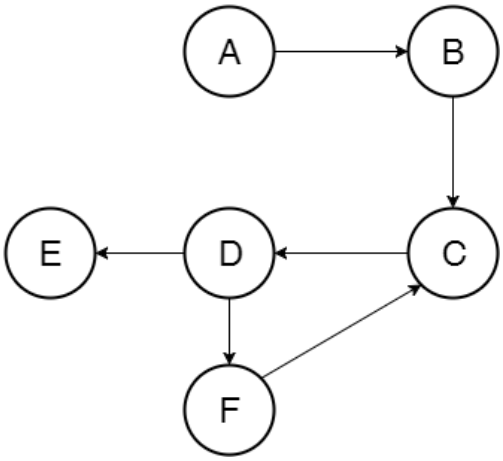


Joonis 2 Lihtne sõltuvusgraaf

Raamistiku vaikimisi graaf on kujutatud Map struktuurina, mis kaardistab objekte ja loendit nende sõltuvustest. Iga objekti sõltuvus peab olema ka graafi elemendiks ehk HashMap-i võtmeks. Struktuur on liidese abil abstraheeritud, liidese kasutajal on tarvis implementeerida vaid graafi lisamine, pärimine ja selle sorteerimine.

Graafi vaikimisi implementatsioon `DefaultDependencyGraph` kasutab raamistiku spetsiifikaga kohandatud versiooni Arthur B. Kahni topoloogilise sorteerimise algoritmist [18]. Tegemist on algoritmiga, mis võimaldab objekte ja nende sõltuvusi järjestada nii, et objekti sõltuvused asuvad alati nimekirjas objektist endast eespool. See tähendab meie funktsionaalsuse puhul, et neid objekte luuakse nendest sõltuvatest objektidest varem. Algoritm võib ebaõnnestuda siis, kui tuvastatakse sõltuvuste tsükkel. Selline tsükkel tekib, kui kaks objekti on omavahelises sõltuvuses, mille tagajärjel ei ole topoloogiline järjestamine võimalik.

Joonisel Joonis 3 on sõltuvuste tsükli olukorra näide, kus tipp C sõltub D-st, mis sõltub F-ist, mis omakorda sõltub C-st. Seega tippud C ja F on omavahelises sõltuvuses.



Joonis 3 Graaf tsükliliste sõltuvustega

Esmalt päritakse graafist kõiki objekte, millel ei ole väljaminevaid sõltuvusi, ehk nende sõltuvuste loend on tühi. Seda protsessi kujutatakse joonisel Joonis 4.

```
// List of nodes with no outgoing dependencies
List<GraphEntry> nonDependentEntries = entries.keySet().stream()
        .filter( entry -> getDependencies(entry).isEmpty() )
        .collect( Collectors.toList() );
```

Joonis 4 Sõltuvusteta kirjete pärimine

Seejärel võetakse loendist esimene kirje ning lisatakse see kohe järjestatud loendisse, mida algoritmi lõpus tagastatakse. Graafist päritakse kõiki antud kirjest sõltuvaid elemente, eemaldades neist ükshaaval käesolevat sõltuvust. Antud operatsioon on kujutatud joonisel Joonis 5.

```
while ( !nonDependentEntries.isEmpty() ) {
    GraphEntry currentNonDependent = nonDependentEntries.remove(0);
    sortedList.add(currentNonDependent);

    List<GraphEntry> currentInList = getIngoingDependencies(currentNonDependent)
```

Joonis 5 Loendi kirje pärimine ja tema sõltlaste leidmine

Igat sõltuvust eemaldatakse *while-loopi* abil, sest ei ole välistatud sama sõltuvuse esinemine mitu korda. Näiteks võib meetodil või konstruktoril olla mitu sama klassi ja nimega parameetrit. Juhul, kui vaadeldaval elemendil pole enam sõltuvusi, lisatakse see sõltuvusteta elementide loendisse, nagu on näidatud joonisel Joonis 6.

```
// List may contain multiple instances of this dependency
while ( getDependencies(currentDependent).contains(currentNonDependent) )
    getDependencies(currentDependent).remove(currentNonDependent);

// If dependent node no longer has any dependencies, add it to the list
if ( getDependencies(currentDependent).isEmpty() )
    nonDependentEntries.add(currentDependent);
```

Joonis 6 Sõltuvuse eemaldamine ja uue sõltuvusteta kirje lisamine

Seda protseduuri tehakse, kuni sõltuvusteta kirjete loend on tühi, ning kontrollitakse, kas graafi mingil elemendil on veel sõltuvusi alles jäänud. Kontrollimiseks kasutatud meetod on näidatud joonisel Joonis 7.

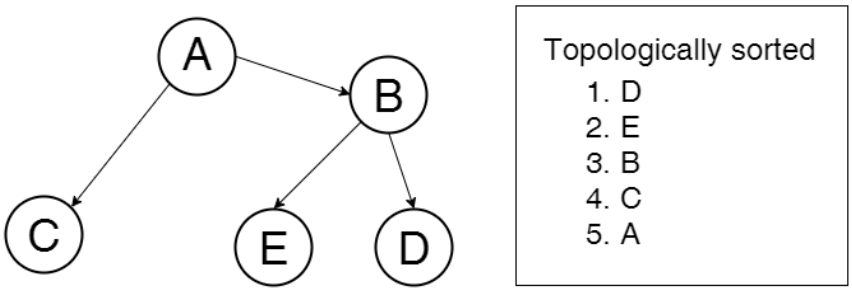
```
private boolean hasNoDependencies() {
    return entries.values().stream().allMatch(List::isEmpty);
}
```

Joonis 7 Graafi sõltuvuste kontrollimine

Sõltuvuste olemasolul on tegemist tsükliliste sõltuvuste situatsiooniga, ehk graafi kaks kirjet sõltuvad üksteisest ning topoloogiline sortimine on võimatu.

Vastasel juhul on sorteerimisprotsess edukalt lõppenud ning meetod tagastab kasutajale õiges topoloogilises järjekorras olevate kirjete listi.

Joonisel Joonis 8 on näide sõltuvusgraafist ja temale topoloogilise sortimise rakendamise tulemusest



Joonis 8 Topoloogiliselt sorteeritud graaf

Tihti esineb olukordi, kus graafi saab mitmel erineval moel topoloogiliselt sorteerida ning saada erinev, kuid korrektne tulemus.

4.2.7 Instantside loomine

Järjestatud sõltuvused salvestatakse ükshaaval hoidlasse, rakendades iga objekti korral temale vastavat loomise protseduuri. Meetodite ja konstruktorite korral käivitatakse neid, kasutades parameetritena graafis olevat sõltuvuste loendit. Eelduseks on, et sortimisalgoritmi õnnestumise tulemusena on kõik vajalikud parameetrid enne vastava elemendi käivitust juba hoidlas olemas.

Klassi väljade puhul võetakse väljas sisaldatav väärtus ja lisatakse otse hoidlasse. Väärtusena sobib ka null.

Olukorras, kus graafi kirjel puudub klassi element ehk `AccessibleObject` klassi instants, loetakse käesolevat kirjet tundmatuks sõltuvuseks. Sellise objekti tekkimine on tavaliselt kasutaja viga, näiteks kui annotatsioonis ei ole täpsustatud nimi, või nimi on kirjutatud valesti. Kasutajale kuvatakse selle peale vastav viga olukorra parandamiseks.

4.2.8 Instantside sisestamine

Kõige viimases faasis teostatakse klassi väljade ja setter-tüüpi meetodite sisestamine. Antud faas on üldisest protsessist eraldi, sest selle käigus ei looda ühtki uut objekti. Setter-tüüpi meetodite all mõeldakse kõiki meetodeid, mille puhul käivitamise tulemust ei ole vaja kontrollida ega objektide hoidlasse salvestada. Ehk tegelikult ei järgita otseselt setterite mustrit, vaid tegemist on veidi üldisema lahendusega.

Nii väljade kui meetodite puhul päritakse sisestamiseks vajalik refleksiivne informatsioon. Iga välja või meetodi puhul võetakse objektihoidlast neid sisaldava klassi instants.

Joonisel Joonis 9 on näide `@InjectInstance` annoteeritud ning parametriseeritud väljast

```
@InjectInstance(value = "Audi", assignableClass = Car.class)
Vehicle vehicle;
```

Joonis 9 @InjectInstance annoteeritud väli

Väljadele omistatakse hoidlast päritud väärtus, mis vastab välja tüübile ja `@InjectInstance` annotatsioonis sisalduvale metainformatsioonile. Meetodite puhul toimub ka parameetrite pärimine objektihoidlast ning lõpuks meetodi käivitamine. Parameetrite all peetakse silmas, kas neile on omistatud täpsustav `@InjectInstance` annotatsioon, kuigi annoteerimine pole kohustuslik.

Joonisel Joonis 10 on näide `@InjectInstance` annoteeritud meetodist, kus sama annotatsiooni on rakendatud kahele meetodi argumendile.

```
@InjectInstance
public void initializeGarage(Car primaryCar,
                             @InjectInstance("Audi") Car secondaryCar,
                             @InjectInstance(value = "RustyBicycle", assignableClass = Bicycle.class)
                                Vehicle backupVehicle) {
    this.primaryCar = primaryCar;
    this.secondaryCar = secondaryCar;
    this.backupVehicle = backupVehicle;
}
```

Joonis 10 @InjectInstance annoteeritud meetod

Sellega on raamistiku sõltuvuste sisestamise protsess lõppenud ja kasutajale tagastatakse selle käigus loodud InstanceRepository objekt.

5 Kokkuvõte

Antud lõputöö käigus leidsin lahenduse sõltuvuste sisestamise kasutamisele oma projektides. Tutvusin metaprogrammeerimise ühe alamosaga, sain teadmisi refleksiivse programmeerimise kasutamisest Javas ja rakendasin seda uue raamistiku arendamise käigus. Eksisteerivate lahenduste uurimisel ja võrdlemisel sai selgeks, kuidas selline raamistik võiks olla disainitud.

Uus raamisik vastab kõikidele funktsionaalsuse nõuetele ning on võimeline konkureerima olemasolevate variantidega. Ta on disainitud lähtudes Spring Frameworki mudelist ning põhineb lihtsal sõltuvuste sisestamise protsessil. Raamistiku kasutamine on samuti tehtud võimalikult lihtsaks, kuid rohkete laiendamise ja täpsustamise võimalustega. Tema kasutusvõimalused ei ole piiratud arendatava programmi tüübi ega eesmärgiga.

Raamistikul on palju täiendusruumi nii uue funktsionaalsuse kui ka olemasolevate algoritmide optimeerimise osas. Plaanin tulevikus raamistikku aktiivselt täiendada ja refaktoorida, vastavalt uute nõuete tekkimisele selle kasutamise käigus.

Kasutatud kirjandus

[1] K. Czarnecki and U. W. Eisenecker, Generative programming : methods, tools, and applications, Boston: Addison-Wesley, 2000.

[2] "Wikipedia Metaprogramming," Wikimedia Foundation, [Online]. Available: https://en.wikipedia.org/wiki/Metaprogramming. [Accessed 18 May 2016].

[3] "Wikipedia Reflection," Wikimedia Foundation, [Online]. Available: https://en.wikipedia.org/wiki/Reflection. [Accessed 18 May 2016].

[4] J. Malenfant, M. Jacques and F.-N. Demers, "A Tutorial on Behavioral Reflection and its Implementation," University of Montreal, Montreal.

[5] "TIOBE Index," TIOBE software BV, May 2016. [Online]. Available: http://www.tiobe.com/tiobe_index?page=Java. [Accessed 18 May 2016].

[6] "JSR 175: A Metadata Facility for the JavaTM Programming Language," Java Community Process; Oracle Corporation, 30 September 2004. [Online]. Available: https://jcp.org/en/jsr/detail?id=175. [Accessed 18 May 2016].

[7] "Java Reflection API," Oracle Corporation, [Online]. Available: http://docs.oracle.com/javase/7/docs/technotes/guides/reflection/. [Accessed 18 May 2016].

[8] M. Fowler, "Inversion of Control Containers and the Dependency Injection pattern," ThoughtWorks, 14 January 2004. [Online]. Available: http://martinfowler.com/articles/injection.html. [Accessed 18 May 2016].

[9] R. E. Johnson and B. Foote, "Designing Reusable Classes," *Journal of Object-Oriented Programming* , vol. 1, no. 2, pp. 22-35, 1988.

[10] D. Nene, "A beginners guide to Dependency Injection," TechTarget, 1 July 2005. [Online]. Available: http://www.theserverside.com/news/1321158/A-beginners-guide-to-Dependency-Injection. [Accessed 18 May 2016].

[11] "JSR 330: Dependency Injection for Java," Java Community Process; Oracle Corporation, 14 October 2009. [Online]. Available: https://jcp.org/en/jsr/detail?id=330. [Accessed 18 May 2016].

[12] "Google Guice User Guide," Google, [Online]. Available: https://github.com/google/guice/wiki/Motivation. [Accessed 18 May 2016].

[13] "Spring Framework Introduction," Pivotal Software Inc, [Online]. Available: http://projects.spring.io/spring-framework/. [Accessed 18 May 2016].

[14] "Contexts and Dependency Injection for the Java Platform.," JBoss, [Online]. Available: http://weld.cdi-spec.org/.

[15] "Dagger Website," Square Inc, [Online]. Available: http://square.github.io/dagger/. [Accessed 18 May 2016].

[16] "PicoContainer Introduction," PicoContainer, [Online]. Available: http://picocontainer.com/introduction.html.

[17] H. A. Al-Mutawa, J. Dietrich, S. Marsland and C. McCartin, "On the Shape of Circular Dependencies in Java Programs," in *2014 23rd Australian Software Engineering Conference*, Milsons Point, 7-10 April 2014.

[18] A. B. Kahn, "Topological sorting of large networks," *Communications of the ACM*, vol. 5, no. 11, pp. 558-562, 1962.

Lisa 1 – Raamistiku kasutamine

Antud lõputöö lisaosas kirjeldatakse raamistiku kasutusvõimalusi, tuuakse näiteid ning võrreldakse teise raamistikuga lihtsa programmi näitel.

1.1 Konfiguratsioon ja protsessi alustamine

Enne sisestamisprotsessi algust on `InjectionProcessManager`'ile võimalik ette öelda mõned komponendid ja nende vastavad parameetrid:

1. Meetodiga `withClasspathScanner` saab täpsustada oma implementatsiooni *classpath*'i otsijast. Sellega on samuti võimalik sügavamalt konfigureerida vaikimisi implementatsiooni, anda ette JAR-ide otsimise käsu või baaspakettide loetelu.

    1.1. Classpathi otsimist saab ka täielikult protsessist välja jätta, kui `withClasses` abil ette öelda täpne klasside loetelu, millega tuleks protsessi käigus toimetada. Selline funktsionaalsus on mõeldud väiksemate programmide otstarbeks.

2. Meetod `withDependencyGraph` võimaldab arendajal implementeerida oma graaf ja sorteerimisalgoritm, et seda kasutada sisestamise protsessis. Vaikimisi graafil muud parameetrid puuduvad.

3. On võimalik juhtida loogikat, mis vastutab päritava klassi tuvastamise eest, kasutades `withAssignableResolver` meetodit. Vaikimisi pakub raamistik kahte erinevat versiooni sõltuvalt päritavuse lahendamise algoritmi sügavusest.

Kõikide ülaltoodud meetodite puhul tagastatakse sama instants `InjectionProcessManager` klassist, mis võimaldab kogu konfiguratsiooni "ühe rea" peal hoida.

Joonisel Joonis 11 on laiema konfiguratsiooni näide, kus täpsustatakse vaikimisi classpath otsija parameetreid, kohandatud sõltuvusgraafi ja päritavuse lahendaja implementatsiooni.

```
InstanceRepository instanceRepository = new InjectionProcessManager()
        .withClasspathScanner( new DefaultClasspathScanner()
            .withJarsIncluded(true)
            .withPackagesIncluded("ee.ttu.game", "com.example.library") )
        .withDependencyGraph( new CustomDependencyGraph() )
        .withAssignableResolver( new MultiAssignableResolver() )
    .inject();
```

Joonis 11 Laiendatud seadistus

1.2 Kasutamise näited

Antud alampeatükkis tuuakse lähtekoodina lihtsaid näiteid raamistiku põhifunktsionaalsuse kasutamisest.

1.2.1 Liidesega sisestamine

Luuakse `@Injectable` annoteeritud klassi `Car` instants, mis implementeerib liidest `Vehicle`, nagu on kujutatud joonisel Joonis 12.

```
@Injectable
public class Car implements Vehicle {

    @Override
    public void run() {
        System.out.println("Car is now running");
    }
}
```

Joonis 12 Liidest Vehicle implementeeriv klass Car

Loodud `Car` objekti sisestatakse joonisel Joonis 13 näidatud klassi `Garage` `@InjectInstance` annoteeritud välja sisse. Klassi välja tüübiks on tegelikult ülemklass `Vehicle`, õiget sisestatavat alamklassi lahendatakse `assignableClass` parameetri abil.

```java
@Injectable
public class Garage {

    @InjectInstance(assignableClass = Car.class)
    Vehicle vehicle;
}
```

Joonis 13 Klass Garage

1.2.2 Sisestamine liidese ja ainsa implementatsiooniga

Olukorras, kus liidesel on ainult üks implementeeritav alamklass, on mõistlik `@Injectable` annotatsiooni kasutada liidese enda peal ning kohe täpsustada alamklassi. Selline olukord on näidatud on joonisel Joonis 14, `HttpService` klassi näitel.

```java
@Injectable(assignableClass = HttpServiceImpl.class)
public interface HttpService {
    <T> T sendMessage(String message, Class<T> returnedClass);

}
```

Joonis 14 Annoteeritud HttpService liides

1.2.3 Setterite sisestamine

On võimalik teostada meetodi argumentide sisestamist, mis on mõistlik setter meetodite puhul. Selline sisestamine on näidatud joonisel Joonis 15.

```java
@Injectable
public class MessageSender {

    HttpService httpService;

    @InjectInstance
    public void setHttpService(HttpService httpService) {
        this.httpService = httpService;
    }

    public void sendMessage(String destination, String message) {
        httpService.send(URI.create(destination), message, "POST");
    }
}
```

Joonis 15 Klass MessageSender ja tema annoteeritud setter

Meetod ei pea tingimata olema void-tüüpi, ega olema kasutatud vaid setteri rollis. Üheks alternatiiviks sellisele sisestamisele on klassi välja sisestamine, sest raamistik ei tee vahet *private*, *protected* ega *public* väljade vahel.

1.2.4 Sõltuvuste tsükkel

Joonisel Joonis 16 on kujutatud ühe klassi sisene sõltuvuste tsükkel.

```
@Injectable
public class Business {

    Capital startingCapital;

    @Injectable
    private Business(Capital capital) {
        this.startingCapital = capital;
    }

    @Injectable
    public Capital produceValue() {
        return new Capital();
    }

}
```

Joonis 16 Valesti seadistatud klass Business

Klassi `Business` annoteeritud konstruktor nõuab klassi `Capital` nimeta objekti. Samal ajal `produceValue` meetod loob oma käivitamise tulemusena nimeta objekti klassist `Capital`. Järelikult on klassi `Business` loomine sõltuv tema enda meetodi käivitamise tulemusest ning tegemist on sõltuvuste tsükkliga.

Antud programmi saaks parandada, omistades `@InjectInstance` konstruktori parameetrile või `@Injectable` meetodile korrektse nime.

1.3 Võrdlus raamistikuga Dagger

Võrdlemine olemasoleva raamistikuga Dagger toimub lihtsa kohvi valmistamise programmi näitel. Seda näidet kasutab Dagger oma kodulehel funktsionaalsuse demonstratsioonina[1].

Antud näite kogu lähtekoodi saab vaadata Daggeri avalikus Githubi salves[2].

Tuuakse välja klassid, mis erinevad olemasoleva lahenduse implementatsioonist.

[1] http://square.github.io/dagger/

[2] https://github.com/square/dagger/tree/master/examples/simple/src/main/java/coffee

```
@Injectable
public class CoffeeApp implements Runnable {

    @InjectInstance
    CoffeeMaker coffeeMaker;

    @Override
    public void run() {
        coffeeMaker.brew();
    }

    public static void main(String[] args) {
        InstanceRepository repository = new InjectionProcessManager().inject();
        CoffeeApp coffeeApp = (CoffeeApp) repository.getInstance(CoffeeApp.class);
        coffeeApp.run();
    }
}
```

Joonis 17 Klassi CoffeeApp implementatsioon

Programmi sisendklassis, mida kujutab Joonis 17, lisandub annotatsioon `@Injectable`, et tagada klassi osalemine sõltuvuste sisestamise protsessis. Kõik muu on sisuliselt sama, kuid uues raamistikus on võimalus teha väli `coffeeMaker` staatiliseks ning eemaldada instantsi pärimine andmehoidlast.

```
@Injectable
public class ElectricHeater implements Heater {
    boolean heating;

    @Override
    public void on() {
        System.out.println("~ ~ ~ heating ~ ~ ~");
        this.heating = true;
    }

    @Override
    public void off() {
        this.heating = false;
    }

    @Override
    public boolean isHot() {
        return heating;
    }
}
```

Joonis 18 Heater liidest implementeeriv klass ElectricHeater

Klassi `ElectricHeater` annoteeritakse sama moodi, luues kohe uue objekti sisestamiseks. Kõik klassi teised osad ning liides `Heater` on jäetud samasugusteks.

```
@Injectable
public class Thermosiphon implements Pump {
    private Heater heater;

    @Injectable
    Thermosiphon(@InjectInstance(assignableClass = ElectricHeater.class)
                 Heater heater ) {
        this.heater = heater;
    }

    @Override
    public void pump() {
        if ( heater.isHot() ) {
            System.out.println("=> => pumping => =>");
        }
    }
}
```

Joonis 19 Liidest Pump implementeeriv klass Thermosiphon

Klassi `Thermisiphon` puhul luuakse `@Injectable` annoteeritud konstruktor ning seadistatakse `ElectricHeater` tüüpi `Heater`'i sisestamine konstruktori parameetrina.

```
@Injectable
public class CoffeeMaker {

    @InjectInstance(assignableClass = ElectricHeater.class)
    Heater heater;

    @InjectInstance(assignableClass = Thermosiphon.class)
    Pump pump;

    public void brew() {
        heater.on();
        pump.pump();
        System.out.println(" [_]P coffee! [_]P ");
        heater.off();
    }
}
```

Joonis 20 Klass CoffeeMaker ning selle seadistus

Objekti `CoffeeMaker` välju häälestatakse `@InjectInstance` annotatsioonide abil otse klassi sees, täpsustades liideste implementatsioone. Seda on näidatud joonisel Joonis 20.

Uue raamistiku implementatsioonist jäid täielikult välja klassid `DripCoffeeModule` ning `PumpModule`, mis algses näites vastutasid sisestamisprotsessi häälestamise eest. Ehk antud programmi puhul saime annotatsioonipõhise seadistamisega lahti kahest lähtekoodi failist ning nende liigsetest sõltuvustest. Samuti ei olnud vajalik

`InjectionProcessManager` objekti seadistamine vastavalt moodulile, nii nagu see Daggeri `ObjectGraph` objekti puhul on.