Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Aleksandar Petrović

RAZVOJ MODULARNEGA ORODJA ZA POTREBE RAČUNSKE BIOLOGIJE

DIPLOMSKO DELO
VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM PRVE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

Mentor: prof. dr. Miha Mraz

Ljubljana, 2013

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00522 / 2013
Datum: 5.9.2013

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **ALEKSANDAR PETROVIĆ**

Naslov: **RAZVOJ MODULARNEGA ORODJA ZA POTREBE RAČUNSKE BIOLOGIJE**
DEVELOPMENT OF A MODULAR FRAMEWORK FOR COMPUTATIONAL MODELLING APPROACHES

Vrsta naloge: Diplomsko delo visokošolskega strokovnega študija prve stopnje

Tematika naloge:

Kandidat naj v svojem delu preuči programerske pristope, ki se v današnjem času uporabljajo za potrebe razvoja modelirnih okolij na področju računske biologije s poudarkom na sintezni in sistemski biologiji. V nadaljevanju naj se loti izdelave modularnega ogrodja, ki bo omogočalo vgradnjo posameznih metod modeliranja bodočim razvijalcem. Z zgledi osnovnih gradnikov naj demonstrira možnosti dograjevanja ogrodja in način dela z orodjem natančno opiše v dokumentaciji.

Mentor:

prof. dr. Miha Mraz

Dekan:

prof. dr. Nikolaj Zimic

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Spodaj podpisani izjavljam, da sem avtor dela, da slednje ne vsebuje materiala, ki bi ga kdorkoli predhodno že objavil ali oddal v obravnavo za pridobitev naziva na univerzi ali drugem visokošolskem zavodu, razen v primerih kjer so navedeni viri.
S svojim podpisom zagotavljam, da:

- sem delo izdelal samostojno pod mentorstvom prof. dr. Mihe Mraza,
- so elektronska oblika dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko in
- soglašam z javno objavo elektronske oblike dela v zbirki "Dela FRI".

— Aleksandar Petrović, Ljubljana, november 2013.

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Aleksandar Petrović

Razvoj modularnega orodja za potrebe računske biologije

POVZETEK

Številna računska orodja, ki z matematični modeli omogočajo simuliranje delovanja in analizo vzpostavljenih modelov bioloških sistemov na področju sintezne in sistemske biologije že obstajajo. Največjo pomanjkljivost obstoječih orodji vidimo v pomanjkanju modularnosti. Glavni cilj diplomskega dela je razviti modularno ogrodje, ki bo omogočalo razvoj in povezovanje različnih gradnikov predvsem za potrebe sintezne in sistemske biologije in bo bodočim uporabnikom in raziskovalcem na enostaven način omogočalo dodajanje lastnih modulov z implementacijo modernih pristopov za modeliranje, analizo in načrtovanje bioloških sistemov. V delu so predstavljena metode, orodja in sam razvoj vzpostavitve modularnega sistema ter napotki in navodila za bodoče razvijalce.

Kljucne besede: računska biologija, modularno ogrodje, OSGi, Java, NetBeans, RCP

University of Ljubljana
Faculty of Computer and Information Science

Aleksandar Petrović

Development of a modular framework for computational modelling approaches

ABSTRACT

Number of computational tools of mathematical models allow simulations of performance and analysis of the established models of biological systems in the field of synthetic and systems biology are already in place. The biggest drawback of the existing tools is the lack of modularity. The main objective of the thesis is to develop a modular framework that will enable development and integration of various building blocks mainly for synthetic and systems biology, and to enable future users and researchers to easily add their own modules that implement modern approaches to modeling, analysis and design of biological systems. This paper introduces methods, tools, and development itself for establishing such a modular system, as well as instructions and guidance for future developers.

Key words: computational modeling, modular framework, OSGi, Java, NetBeans, RCP

ZAHVALA

Rad bi se zahvalil svojemu mentorju prof. dr. Mihi Mrazu in asistentu dr. Mihi Moškonu za vodenje, strokovno pomoč ter spodbudo pri nastajanju diplomskega dela. Hvala tudi Mattii, ki je pomagal pri zasnovi projekta. Zahvaljujem se tudi družini in prijateljem, še posebej svojim staršem, ki so mi omogočili študij, me podpirali ter verjeli vame. Posebna zahvala pa gre moji Petri, ki mi je ves čas stala ob strani in mi pomagala.

— Aleksandar Petrović, Ljubljana, november 2013.

KAZALO

1 Uvod

Računska biologija (angl. *comutational biology*) je znanstveno področje, ki uporablja računske metode pri proučevanju različnih bioloških sistemov. Na področju sintezne in sistemske biologije se je v zadnjih letih uveljavilo kar nekaj metod, ki večinoma temeljijo na vzpostavitvi računskih modelov. Ti modeli nam dovoljujejo, da opazujemo obnašanje določenega biološkega sistema v danem okolju. Natančni kinetični podatki, ki opisujejo osnovne dinamike, so običajno nujno potrebni za pridobitev relevantnih izsledkov na podlagi vzpostavljenih modelov. V določenih primerih je kinetične podatke težko ali celo nemogoče eksperimentalno pridobiti. Za premostitev tega problema lahko uporabimo različne računske tehnike ocenjevanja parametrov. Z uveljavitvijo računskih modelov lahko izvajamo računske analize, kot so analiza obnašanja, robustnosti, občutljivosti in stabilnosti. Poleg tega lahko te tehnike uporabimo za zmanjševanje količine eksperimentalnega dela, kar nas posredno pripelje do lažje konstrukcije izvirnih bioloških sistemov v kontekstu sintezne biologije.

Uporaba tehničnih orodij, ki se zanašajo na razne računske pristope, je bolj ali manj pogosta na področju sistemske biologije. Kot kombinacija kemije, biologije in inženirstva,

so ti pristopi pridobili še večjo vlogo na področju sintezne biologije. Obstoječi pristopi so osnovani predvsem na vzpostavitvi raznih računskih modelov. Če so predvideno okolje in začetni pogoji podani, so ti modeli sposobni posnemati obnašanje skoraj poljubnega biološkega sistema v okviru določenih omejitev. Glede na njihovo izgradnjo obstajata dva glavna pristopa, od zgoraj navzdol (angl. *top-down approach*) in od spodaj navzgor (angl. *bottom-up approach*). Pristop od zgoraj navzdol je osnova za vzpostavitev računskih modelov, pri čemer se uporablja eksperimentalno izmerjene podatke iz že obstoječih bioloških sistemov. Ta pristop je značilen za sistemsko biologijo in je zasnovan na obratnem inženiringu računskih modelov iz eksperimentalno izmerjenih podatkov. Po drugi strani je dopolnilen pristop od spodaj navzgor uporabljen za poenostavljanje izgradnje izvirnih bioloških sistemov s predhodno določenimi funkcionalnostmi. Ti modeli so običajno zgrajeni modularno iz modelov enostavnejših bioloških sistemov, ki se jih lahko vzpostavi s pristopom od spodaj navzgor.

Točnost računskih modelov je tesno povezana s pravilnostjo vrednosti parametrov, ki določajo dinamiko opazovanih bioloških sistemov. Te parametre je včasih težko ali celo nemogoče natančno določiti. Da bi se izognili temu problemu, lahko uporabimo številne pristope za ocenjevanje parametrov. Prav tako že obstajajo številni pristopi, ki omogočajo rekonstrukcijo modelov obstoječih bioloških sistemov iz eksperimentalnih podatkov na primer, z uporabo metode mehke logike.

Vzpostavljene modele, ki izražajo zadovoljivo natančnost, lahko uporabimo za pridobitev specifičnega razumevanja obnašanja določenega biološkega sistema. Ko govorimo o modelih obratnega inženiringa, ti sistemi že obstajajo. Modeli so vzpostavljeni z namenom analiziranja različnih lastnosti sistema, ki jih je težko ali celo nemogoče pridobiti eksperimentalno. Poleg tega lahko modelirne rezultate uporabimo za optimizacijo opazovanih bioloških sistemov. Po drugi strani se modele vzpostavljene s pristopom od spodaj navzgor lahko uporabi za računsko konstrukcijo bioloških sistemov, s čimer zmanjšamo količino eksperimentalnega dela. Nekaj računskih pristopov za avtomatsko konstrukcijo že obstaja. Ti na primer temeljijo na hevristični optimizaciji, ki išče optimalne rešitve na podlagi različnih kriterijskih funkcij [1].

Številna računalniška orodja, ki omogočajo uporabo naštetih pristopov že obstajajo (npr. Tinkercell [2], Copasi [3], Simbiology [4]). Ponavadi so ta orodja usmerjena v točno določen problem ali v ozko kategorijo računskih pristopov (kot je na primer analiza robustnosti). Problem obstoječih pristopov je tudi v slabi razširljivosti, saj so tudi v primeru

odprtokodnosti za zunanjega programerja (t.j. programerja, ki ni sodeloval pri gradnji osnove orodja) težko obvladljiva. Cilj našega dela je bila vzpostavitev odprtega, splošnega in modularnega orodja. Ker posamezni moduli med seboj komunicirajo preko standardiziranih vhodov in izhodov, je dodajanje novih modulov enostavno. Rezultat diplomskega dela predstavlja ogrodje orodja z delovnim imenom Computatinal Biology Worskspace (CBW), ki bo olajšalo delo raziskovalcem pri vzpostavljanju modelov bioloških sistemov, njihovi analizi in testiranju novih računskih pristopov na področju sintezne in sistemske biologije. Poleg tega bo orodje dober učni poligon za bodoče študente pri spoznavanju z zanimivim področjem sintezne biologije, ki po eni strani s svojimi aplikacijami posega tudi na področje računalništva, po drugi strani pa je pri njegovem nadaljnjem razvoju znanje računalničarjev nepogrešljivo.

V 2. poglavju predstavimo metode, orodja ter samo implementacijo programskega ogrodja in posameznih komponent. V 3. poglavju bolj podrobno opišemo proces razvijanja novih komponent oziroma modulov in razložimo uporabo orodja. Na koncu povzamemo naše doprinose in podamo sklepne ugotovitve.

2 Opis rešitve

Glavni cilj diplomskega dela je bil vzpostavitev ogrodja okolja za simuliranje in analizo reakcij, ki predstavljajo zapis modeliranega biološkega sistema. Okolje bi moralo biti odprto za dodajanje novih modulov, ki so pogojno neodvisni eden od drugega. Orodje temelji na osnovnih konceptih programiranja za debele kliente in modularnosti v Javi.

Pri razvoju rešitve smo predpostavljali, da uporabnik za osnovno delo potrebuje komponente, ki pripadajo sledečim skupinam:

- **komponente za postavitev modela**: omogočajo definiranje kemijskih zvrsti, reakcij, ter drugih parametrov, ki so specifični za posamezen model,
- **komponente za ovrednotenje parametrov**: namen teh komponent je definicija začetnih vrednosti kemijskih zvrsti, brez poseganja v že zastavljen model; z ločitvijo od modela pridobimo možnost vzporedne simulacije istega modela z različnimi začetnimi vrednostmi,
- **komponente za simulacijo**: te komponente omogočajo poganjanje simulacij nad vzpostavljenimi modeli; odločimo se lahko za različne matematične pristope simuliranja definiranega modela oziroma izberemo procesiranje, ki najbolj ustreza

modelu; da bi bilo računanje čimbolj optimalno je predvidena tudi uporaba domorodne kode (angl. *native code*),

- **komponente za prikaz podatkov**: skrbijo za prikaz podatkov, ki jih pridobijo iz komponent za simulacijo; so ključnega pomena za končno analizo; prikaz je neodvisen od modela in je lahko v tekstovni ali grafični (2D ali 3D) obliki.

Modularnost orodja omogoča razvoj novih komponent posameznih tipov in tudi dodajanje novih tipov komponent.

2.1 Uporabljene tehnologije

V nadaljevanju so predstavljene tehnologije uporabljene pri razvoju orodja.

2.1.1 Reflections

Pri razvoju je bil med drugimi uporabljem programski vmesnik (angl. *application programming interface, API*) *Reflections* [5]. Pisanje takšne kode ima sicer kar nekaj pomanjkljivosti [6, 7]:

- izgubimo vse prednosti močne tipizacije Jave,
- pisanje takšne kode je nerodno in gostobesedno,
- hitrost procesiranja pade tudi za 200%.

Zaradi vseh teh slabosti programiranje z *Java Reflections* ni priporočljivo oziroma se ga poskusimo izogniti, če le lahko. Projekt lahko postane prehitro zelo kompleksen in težko vzdržljiv. Pride lahko tudi do večjega števila napak, še posebej ko pride do sprememb v kodi, na katere nas bi opozorilo že integrirano razvojno okolje (angl. *integrated development environment, IDE*) in nato še prevajalnik (angl. *compiler*).

Čeprav za modularno programiranje ni najboljša rešitev, to še zdaleč ne pomeni, da pri modularnem programiranju ni uporaben. Njegov programski vmesnik je zelo močan in omogoča stvari, ki jih z navadnimi pristopi ni mogoče izvesti. Zato se ga poslužujemo, ko drugače ne gre.

2.1.2 Maven

Maven je programska oprema za upravljanje in gradnjo (predvsem) Java projektov [8]. Primarni namen je bil nadomestiti oziroma izboljšati obstoječe orodje za gradnjo *Ant*, ki olajša prevajanje in povezovanje programov z upoštevanjem soodvisnosti. Čeprav je Ant programerjem velikokrat v pomoč, zna postati upravljanje na obsežnih projektih ali različnih projektih, kjer je vsaka Ant datoteka malo drugačna, precej kompliciranо. Še posebej, ker ni povsem standardizirano.

Maven tako nadgradi Ant (čeprav gre za neodvisen projekt) in doda še dodatne funkcionalnosti, ki s svojo *POM* (angl. *Project Object Model*) strukturo poenostavijo razvijanje aplikacij, ki imajo veliko odvisnosti in modularnosti, kar je za projekt zastavljen v diplomski nalogi primarnega pomena. Dotakne se tudi življenjskega cikla aplikacije (angl. *application lifecycle management* – *ALM*), vendar ne v taki meri kot orodja specifično namenjena temu.

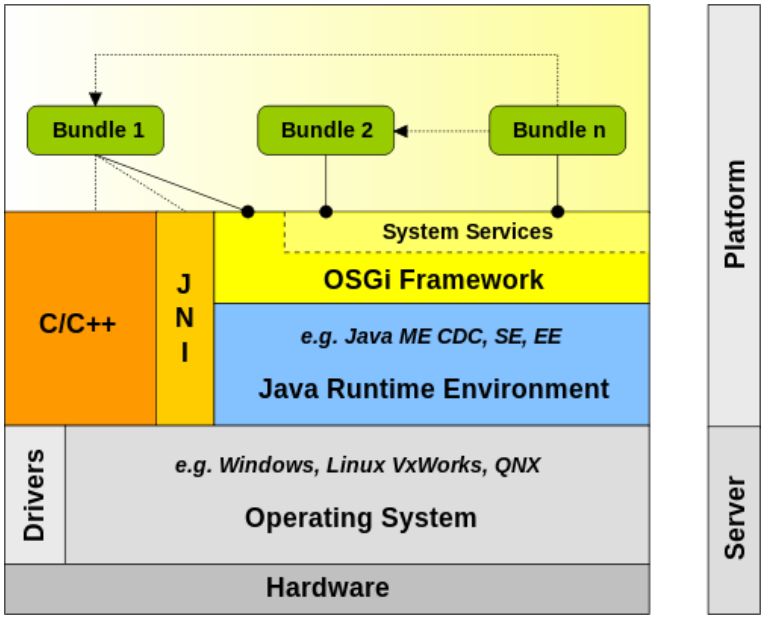
Z Maven okoljem lahko ustvarimo tudi svoj lokalni repozitorij, kjer bi se vsak modul razvijal neodvisno od drugih. Povezovanje modulov bi lahko rešili z deklarativnimi soodvisnostmi, v katerih je zapisana tudi verzija. S tem pridobimo nemoten razvoj in testiranje lastnih modulov ter morebitne spremembe drugih modulov, ki še niso kompatibilne z našim, rešimo na koncu.

2.1.3 OSGi – Open Services Gateway Initiative

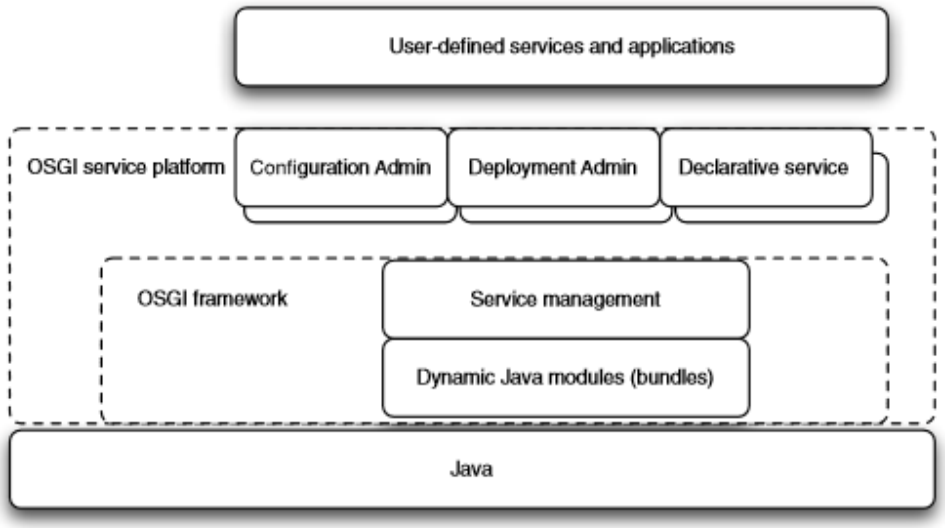
Čeprav je Maven odlično okolje za razijanje in vzdrževanje projektov z večjim številom modulov, ne ponuja nobenih prednosti v samem delovanju aplikacije. Tu pride v poštev OSGi okolje. OSGi je v najožjem opisu nadgradnja Jave z dinamičnim modularnim sistemom in storitvami (glej sliki 2.1 in 2.2) [9].

Glavne prednosti tega okolja so:

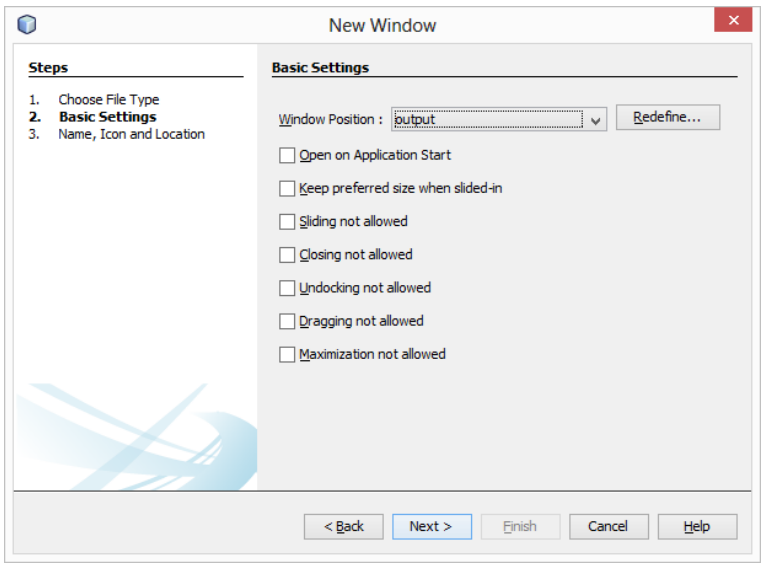
- prenosno in varno okolje za izvajanje kode,
- podpira upravljanje storitev, ki se lahko uporablja za registracijo in skupne storitve,
- vsebuje dinamičen sistemski modul, ki se lahko uporablja za dinamično namestitev in odstranitev Java modulov,
- predstavlja lahko in prilagodljivo rešitev.



Slika 2.1 OSGi sistemske plasti [10].



Slika 2.2 Hierarhija OSGi platforme.



Slika 2.3 Izdelava novega okna preko NetBeans čarovnika.

Ob okviru OSGi programskega ogrodja, OSGi storitvena platforma vključuje več storitev splošnega namena. Te storitve bi lahko tudi opisali kot domorodne aplikacije v okviru OSGi platforme. Nekatere od teh storitev so horizontalne funkcijam, ki so večinoma vedno potrebne, kot na primer beleženje storitev in konfiguracija storitve.

OSGi je načeloma le specifikacija. Tako poznamo več odprtokodnih implementacij, kot so na primer *Equinox* (Eclipse), *Apache Felix* in *Knopflerfish OSGi* [11].

2.1.4 NetBeans IDE in NetBeans platforma

Izbira programskega vmesnika (*IDE*), ki podpira vsa potrebna orodja, igra pomembno vlogo za izdelavo modularnih aplikacij. Tako *Eclips* kot *NetBeans* podpirata OSGi in sta tudi sama implementirana na podlagi le-tega. Oba ponujata tudi svoj *RCP* (angl. *rich client platform*) in temeljita na OSGi ogrodju. Vsak ponuja tudi svoje dodatne programske vmesnike (*API*), predvsem za upravljanje oken in menijev.

Po primerjavi funkcionalnostih in krajši raziskavi, ki je temeljila na izdelavi preproste aplikacije, nismo odkrili nobene pretirane prednosti enega okolja od drugega. Na koncu smo se na podlagi boljše dokumentacije ter podpore na njihovi spletni strani, ki bo lahko v veliko pomoč tudi ostalim, odločili za NetBeans. Poleg tega ima uporabniku bolj prijazen vmesnik za izdelavo novih modulov, oken, menijev itd., saj lahko veliko stvari

naredimo tudi preko vgrajenega čarovnika (glej sliko 2.3), ki namesto nas ustvari potrebne datoteke in okostje. NetBeans že od verzije 6.9 naprej, podpira OSGi specifikacijo. Za projekt je bila uporabljena verzija 7.2. Čeprav podpira standard OSGi in bi projekt lahko temeljil na implementaciji OSGi, kot je naprimer Equinox ali Felix, smo se raje odločili za uporabo *NetBeans Runtime Container*. Le-ta je privzeto del NetBeans RCP in je zelo podoben OSGi implementaciji. S tem smo se znebili dodatnega OSGi vsebnika, saj je *NetBeans Runtime Container* kompatibilen z OSGi moduli in dopušča integracijo le-teh. *NetBeans Runtime Container* prinese tudi nekaj izboljšav, kot so na primer anotacije, ki jih OSGi ne more uporabljati, saj trenutno najbolj podprta implementacija 4.2 temelji na JDK 1.3 [12, 13].

2.1.5 NetBeans Visual Libary

Knjižnica *NetBeans Visual Library API* je generična knjižnica za vizualizacijo različnih struktur. Še posebej primerna je za predstavitev podatkov v obliki grafov. Vizualna knjižnica je del standardne platforme NetBeans in sam *NetBeans IDE* ga uporablja v številnih modulih in področjih, kot je na primer vizualno modeliranje *MIDlet* aplikacij za *Java Micro Edition* (JME). Uporabo knjižnice *Visual Library API* omogočimo preprosto z določitvijo odvisnosti v modulu, za ostalo pa poskrbi NetBeans sam. Komponente knjižnice *Visual Library API* imajo podobno strukturo kot komponente knjižnice *Swing*. Obe imata drevesno strukturo in jih lahko izmenično gnezdimo. Posamezen gradnik lahko tako vsebuje več drugih gradnikov. Vsak gradnik ima položaj, ki je določen glede na svoj starševski gradnik. Nadrazred vseh grafičnih elementov je razred *Widget* in je odgovoren za predstavitev robov, ozadja ter ostalih lastnosti gradnikov. Kot *Swing* vsebnik (angl. *container*), ima gradnik določeno razporeditev za pozicioniranje svojih gradnikov, ki jih vsebuje. Gradniki so lahko tudi odvisni eden od drugega in so s tem obveščeni o njihovih spremembah. Gradnik omogoča vrsto akcij, ki se izvršijo, ko pride do specifičnih uporabniških dogodkov [13, 14].

2.1.6 Java FX

Java FX, je platforma za pisanje obogatenih spletnih aplikacij (angl. *Rich internet applications, RIA*) [15]. Platforma je bila na začetku namenjena pisanju spletnih apletov, a je prerasla v veliko več. Od Java verzije 7.6 je tudi vključena v namestitev Java SE. Čeprav ponuja ogromno programskih vmesnikov (*API*) in orodij, jo bomo v tem projektu

porabili le za vizualizacijo rezultatov simulacij in analiz. Rezultati bodo predstavljeni kot graf.

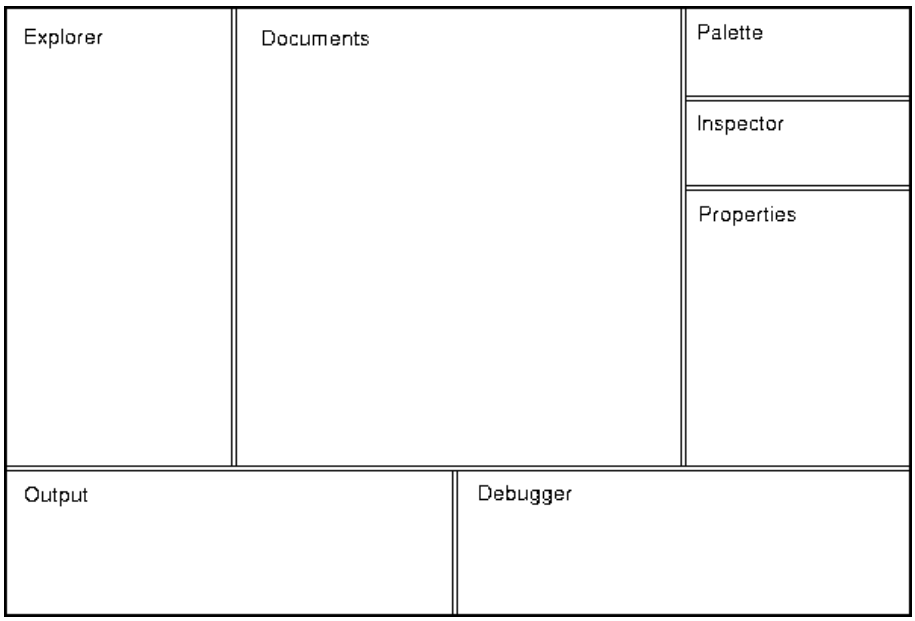
2.2 Predvidene funkcionalnosti orodja

Projekt je bil zasnovan tako, da bi različne module oziroma orodja za računanje lahko povezovali skupaj in na koncu rezultat vizualizirali z grafom. Moduli naj bi bili združeni po funkcionalnosti. V aplikaciji bi bil prostor oziroma delovna površina, na katero bi lahko dodajali module iz palete in jih povezovali med sabo. Vsak modul bi nato lahko uporabnik odprl, v novem oknu pa bi se mu pojavila vnosna polja za vnos začetnih stanj reakcij in reaktantov. Ko bi izpolnil vse podatke, bi uporabnik odprl modul za izpis in izris podatkov ter od tam pognal celoten proces simulacije. Uporabnik bi lahko shranil svoje delo ali pa ga izvozil v XML format.

2.3 Programiranje orodja

Pri programiranju so bila uporabljena orodja in pristopi opisani v poglavju 2.1. Najprej smo ustvarili nekaj testnih projektov, kjer so bila uporabljena orodja testirana. Sledila je povezava z NetBeans RPC okoljem. Ustvarjena je bila preprosta aplikacija, s katero smo testirali kako lahko moduli med seboj delujejo in kako med njimi nastaviti soodvisnoti. Na ta način smo ugotovili, da modula ne smeta biti soodvisna. Samo eden je lahko odvisen od drugega s čimer se izognemo sklenjenemu krogu med moduli. Zato mora biti en modul, ki upravlja z ostalimi moduli jedro aplikacije (lahko bi imel tudi več različnih središč). Vse soodvisnosti so zapisane v *project.xml* datoteki, ki jo ima vsak modul. Pri odvisnosti določimo tudi verzijo in strogost, ki se je držimo. Lahko nastavimo, da želimo imeti točno določeno verzijo na primer 1.3, ali pa dopuščamo možnost in sprejmemo katerokoli verzijo glavne objave npr. 1.x. Pri tem moramo vedeti kaj šteje za velike in kaj za male objave in jih tudi pravilno voditi.

Po povezovanju med moduli smo se lotili grafičnega vmesnika. *NetBeans RPC* ima svojega upravljalca oken, ki prihrani ogromno dela, saj za večino stvari poskrbi sam. Še vedno imamo možnost ročnega upravljanja, če privzeto obnašanje ni ustrezno. Vsako okno oziroma komponenta se začne najprej z NetBeans razredom *TopComponent*, kateremu definiramo privzeto pozicijo in določene parametre, kot je na primer prikaz komponente ob samem zagonu aplikacije. NetBeans ima določene privzete pozicije, ki tudi



Slika 2.4 Privzeta postavitev strani *NetBeans Window* modela.

vplivajo kako se bo okno modula prikazovalo in obnašalo (glej sliko 2.4).

2.3.1 Jedro

Najprej smo ustvarili jedrni modul (angl. *core*), ki skrbi za primarno okno aplikacije. Iz tega modula se nadzoruje prikaz platna in palete ter ostalih funkcij, ki so na nivoju celotne aplikacije, kot je npr. shranjevanje. Modul skrbi tudi za odpiranje drugih modulov in pravilno postavitev njihovih oken.

2.3.2 Generične komponente

Vsi novi moduli so primarno komponente, katere dodajamo na delovno površino. Da bi vse komponente bile narejene po določenem kopitu in pravilih smo naredili generični modul – *GenericTool*. V tem modulu smo definirali abstraktni razred *AbstractGenericTool*, ki definira obnašanje in obvezne parametre vseh novih modulov tipa komponent. Ker bo v aplikaciji več tipov komponent, smo definirali še abstraktne podrazrede (*AbstractModelTool, AbstractParamEvalTool, AbstractPlotTool, AbstractReactionType, AbstractEngineTool*), ki dedujejo *AbstractGenericTool* razred, kjer lahko še bolje definiramo skupne ali abstraktne metode, ki jih bodo morali implementirati avtorji bodočih modulov. Na

ta način je možno naknadno popraviti ali spremeniti *API* za novejše verzije aplikacije, moduli, ki so dodani, pa so primorani delovati po novem *API*. Seveda je po takih spremembah obvezno potrebno popraviti tudi verzijo modula ter s tem obvestiti module o morebitnih nekompatibilnostih.

2.3.3 Platno

Glavno delovno površino oziroma *platno*, kamor bo uporabnik dodajal in povezoval module, smo v modulu *Core* dodali z razredom *CoreTopComponent*, ki je implementacija NetBeans komponente *TopComponent*. Le-ta je nastavljen tako, da se pokliče z vsakim zagonom aplikacije in se pojavi na *Documents* poziciji (glej sliko 2.4). Sama komponenta ima še dve dodatni podkomponenti. Prva je navadna *Swing* komponenta – *jPanel*, v kateri so gumbi za shranjevanje, odpiranje in čiščenje delovne površine. Druga pa je komponenta tipa *jScrollPane*, katero razširimo s komponento *GraphScene* z *NetBeans Visual Library*. Na začetku smo uporabili standardno implementacijo razreda *GraphScene*, katero smo zatem zamenjali s svojo razširjeno implementacijo *ToolGraphSceneImpl*. Komponenta *GraphScene* [16] je v bistvu razred, ki vsebuje in upravlja z grafično usmerjenim modelom. V tem primeru je graf sestavljen iz vozlišč in povezav, kjer ima vsaka povezava vir in ciljno vozlišče. Vozlišča bodo v naši aplikaciji posamezni moduli, povezave pa bodo predstavljale potek in smer izvajanja analize.

2.3.4 Paleta

Da bi lahko dodajali module na delovno površino potrebujemo paleto. Zato smo dodali nov modul *ToolPalette*. Le-ta ne bo imel svojega *TopComponent* prikaza, saj ne bo vedno na voljo za prikaz. Prikaz komponente bo vezan na delovno površino. Tudi inicalizacija palete se bo zgodila vzporedno z inicalizacijo delovne površine.

Za izdelavo palet ima NetBeans platforma svoj *API*, ki dopušča, da paleto naredimo programsko, ali pa jo podamo kot XML datoteko. XML zapis bi bil preveč statičen za potrebe te aplikacije zato smo jo spisali z uporabo *NetBeans Palette API*. Paleta je bila razdeljena na skupine (komponenta za modeliranje, parametirizacijo, računanje in risanje). Vsaka skupina je, programsko gledano, glavno vozlišče palete in vsako vozlišče vsebuje še svoj podseznam komponent, ki spadajo v to skupino. Skupine se napolnijo dinamično glede na prisotne module v sistemu. Te je moč poiskati z uporabo *Lookup API*, ki vrne seznam vseh modulov in razredov, ki implementirajo eno od generičnih

komponent.

Zadnji element v tej drevesni strukturi palete je nov objekt *ToolNode* tipa *AbstractNode*. Ta objekt skrbi za prikaz modula v sami paleti ter drži povezavo na modul, ki smo jo dobili preko *Lookup API* klica. Tako se lahko tudi ime in avtor modula napolnita dinamično, ker se informacija pridobi iz samega modula. Z manjšo spremembo modula bi bilo moč dodati tudi ikone ali dodaten opis modula.

2.3.5 Povezava palete in platna

Med platnom in paleto je sedaj le šibka povezava. Paleta se inicalizira in odpre skupaj s platnom. Omogočiti je bilo potrebno še prenos objektov z ene na drugo. Želeno je bilo, da sistem deluje na principu povleči in spusti (*Drag and drop*), za kar smo uporabili Javanski programski vmesnik. Znotraj *Swing* knjižnice se tem operacijam kolektivno reče *Data transfer* [17] in omogoča prenašanja informacij med komponentami. Tako je bilo potrebno poskrbeti, da izvorna in ponorna komponenta znata oddati in sprejeti drugo komponento, ter narediti nov objekt, ki bo prenašal te informacije. Nov objekt je potreben, ker mora implementirati razred *Transferable* in direktno prenašanje glavnega objekta (modula) ni priporočljivo (če bi želeli dodati še kako informacijo o prenosu, to ne bi bilo mogoče). Zato smo ustvarili nov objekt *ToolWrapper*, ki implementira vse potrebne metode za prenašanje le-tega (npr. *getTransferData, isDataFlavorSupported*, itd.). Podpore za prenos objektov v paleti ni bilo težko prilagoditi, sej je bilo potrebno le zamenjati direktno povezavo na modul z objektom *ToolWrapper* in objektu *ToolNode* dodati oziroma prepisati metodo *drag* (razred *AbstractNode*, ki ga deduje, že privzeto podpira akcijo povleči in spusti), da vrača objekt tipa *ToolWrapper*. Malo bolj pa je bilo potrebno prilagoditi platno.

Implementaciji *GraphScene* platna smo najprej prilagodili prvi generičen parameter *ToolWrapper*. Ta predstavlja vozlišča na platnu, povezave pa ostajajo tipa *String*. Nato smo platnu dodali še akcijo, ki omogoča sprejemanje objektov preko metode povleci in spusti. Le-ta prenese instanco *ToolWrapper* s palete na platno. Ker morajo biti instance na platnu unikatne, pride do napake, ko želimo ponovno dodati isto komponento. Tega smo se izognili tako, da se ob dodajanju instance na platno najprej naredi kopija, šele nato pa se jo doda. V Javi to omogoča metoda *clone*, ki je del *Cloneable* vmesnika, katerega smo morali dodati *ToolWrapper* razredu. Na ta način lahko na platno dodajamo več istih komponent, vsako od njih pa spreminjamo neodvisno od drugih.

Za prikaz vozlišča oziroma komponente na platnu se kliče metoda *attachNodeWidget*, ki je v dedovanem razredu *GraphScene*. To metodo smo v svoji implementaciji razreda prepisali s svojo, kateri smo dodali izris dodanega gradnika na platnu. Za prikaz gradnika smo uporabil razred *IconNodeWidget* [18]. Ta se izriše kot preprost pravokotnik z imenom komponente v sredini. Omogoča tudi izris ikone, če bi jo komponenta vsebovala. Dodali smo tudi možnost premikanja komponent po platnu z miško.

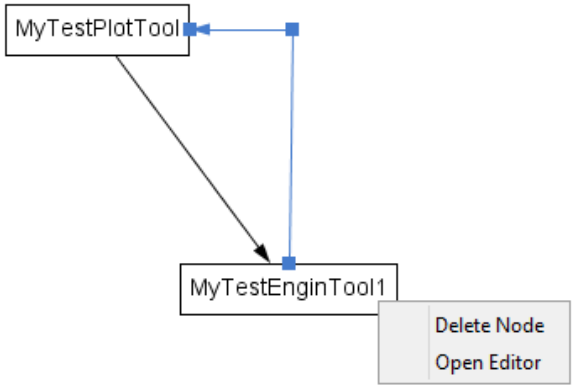
2.3.6 Povezovanje in urejanje komponent

Za uspešno izvedbo simulacije je potrebno povezati različne komponente skupaj. Odločili smo se, da bodo logične povezave med komponentami vizualizirane s puščicami, katere bo uporabnik dodal s pomočjo miške (glej sliko 2.5). To funkcionalnost smo omogočil preko *NetBeans Visual API* knjižnice, ki že ima orodja za ravno takšno povezovanje gradnikov. Ustvarili smo nov razred *SceneConnectProvider*, ki implementira razred *ConnectProvider*, ki je del *NetBeans Visual API* ter ga prilagodili potrebam projekta. Dodali smo še nov interakcijski sloj, kjer se povezave tudi izrišejo. Povezovanje gradnikov deluje po principu klikni in poveži. Uporabnik mora držati tipko *Control* ter nato z miško izbrati objekt in potegniti črto do želenega objekta. Če na drugi strani povezave ni pravilnega objekta (komponenta), se povezava ne ustvari in posledično ne zapiše. Prav tako se povezava ne zapiše, če povezava med objektoma že obstaja. Ker so povezave usmerjene, to velja samo za eno smer, kar pomeni, da sistem dopušča obojestransko komunikacijo med komponentami. Ta funkcionalnost je spisana v svojem razredu *SceneReconnectProvider*, ki je implementacija *NetBeans Visual API ReconnectProvider*.

Ker bo pri uporabi prišlo do napak oziroma pomot, smo v uporabniški vmesnik dodali še urejanje grafa povezav. Dodali smo dva poslušalca, ki se aktivirata ob kliku desnega gumba miške, če je le-ta nad gradnikom komponente ali povezave. Ob kliku na enega izmed objektov se odpre spustni meni z različnimi akcijami. Pri povezavi smo dodali možnost brisanja izbrane povezave in dodajanja dodatne točke na povezavo, s katero lahko naredimo povezave med komponentami bolj berljive.

Z desnim klikom na gradnik tipa komponenta lahko uporabnik s spustnega menija izbere opcijo brisanja gradnika in s tem tudi vseh povezav, katerih je deležen ter akcije, ki sproži urejanje komponente. Slednja sproži funkcijo *openEditor* v razredu *AbstractGenericTool*, katero implementira vsaka komponenta posebej.

Maska za urejanja komponente je odvisna od posameznega modula, oziroma je pov-



Slika 2.5 Urejanje gradnikov in povezav.

sem prepuščena avtorju modula. Slediti mora le nekaterim navodilom. Ker aplikacija dopušča dodajanje več instanc iste komponente, smo morali prilagoditi tudi masko za urejanje. Zato *Lookup API* na tem mestu ne pride v upoštev, saj mora obstajati povezava med komponento in masko, ki mora kot komponenta imeti svojo instanco. Zato smo v razred *AbstractGenericTool* dodali abstraktno metodo *getTopComponentClass*, katero mora implementirati vsak, ki deduje ta razred. Metoda mora vrniti objekt *Class* urejevalnika, ki natančno pove kako instancirati razred.

Kot že omenjeno, NetBeans uporablja razred *TopComponent* kot primarno komponento za grafični prikaz. Zato morajo tudi vsi urejevalniki komponent uporabljati *TopComponent* kot glavni razred za prikaz urejevalnika. Problem nastane, ko želi komponenta dostopati do informacije predhodne komponente, ki je določena na delovni površini. Zato je bila ustvarjena lastna implementacija *ToolTopComponent*, ki deduje razred *TopComponent*. V svoji implementaciji smo dodali konstruktor, ki zahteva dva parametra. Prvi parameter je tipa *GraphScene*, ki kaže na delovno površino, drugi pa je tipa *IconNodeWidget*, ki kaže na gradnik, s katerega je uporabnik prišel v urejevalnik. Razredu smo dodali še metodi *getToolWrapper* in *getGenericTool*, ki iz danih kazalcev izluščita glavni razred komponente, ki ga urejamo. Tako imajo komponente dostop do predhodne komponente in s tem do njenih podatkov. Preko te povezave lahko komponenta na primer dobi podatke za izračun iz komponente za modeliranje.

2.3.7 Shranjevanje in nalaganje delovne površine

Da bi lahko uporabnik delo prekinil in kasneje nadaljeval tam kjer je končal, oziroma delil svoje delo z drugimi uporabniki, program omogoča shranjevanje in nalaganje. Shranjevanje je implementirano v obliki XML zapisa, v katerem so serializirani javanski objekti in zapisani z znaki z osnovo 64. Najprej se shrani delovna površina. Zapišejo se lokacije komponent in povezav med njimi v obiki XML zapisa. Nato se vsaka komponenta serializira. Serializira se le razred, ki deduje *AbstractGenericTool*. Le ta že implementira *Serializable*, ki je del *Java IO API*. Serializacija objekta zapiše tudi vse vrednosti spremenljivk, vendar samo tistih ki niso definirane s predpono *transient*. *Transient* [19] je indikator, katerega Java upošteva, ko se objekt serializira in tako označene spremenljivke spusti. Ko se objekt deserializira, dobijo neserializirane spremenljivke vrednost *null*. Serializacija vrne vrednost kot objekt tipa *ByteArrayOutputStream*. Le-ta ni primeren za zapis v tekstovno datoteko. Zato smo binarni zapis pretvorili v tekstovni zapis z osnovo 64. Uporabniku se nato pokaže okno za shranjevanje datoteke in podatki se zapišejo na disk. Ta datoteka je lahko uporabljena v poljubnem okolju, ampak le če je nameščena aplikacija z vsemi moduli, ki so bili uporabljeni v izdelavi te datoteke.

Delovna površina vsebuje tri gumbe in sicer Shrani, Naloži in Pobriši, s katerimi lahko shranjujemo in nalagamo podatke ter izpraznimo površino.

Ker obstaja možnost, da bodo moduli držali določene informacije izven območja, ki se bo serializiralo, *ToolTopComponent* vsebuje abstraktno metodo *doSave*. Le-to pokliče metoda, ki izvaja shranjevanje pred serializacijo objekta. Tako se lahko podatki, ki so trenutno le na grafičnem prikazovalniku, zapišejo v glavni *AbstractGenericTool* razred, ki se serializira.

2.4 Prototipne komponente

Za lažjo izdelavo dodatnih modulov in primer pravilnega delovanja, smo izdelali prototipne module za vsak tip komponente. Podrobnejši opis izdelave novih modulov je opisan v 3. poglavju.

2.4.1 TestModelTool

Pri komponenti za modeliranje je bilo največ dela z masko. Preko čarovnika je bil ustvarjen nov *TopComponent* razred. Razred je bil prirejen, tako da namesto *TopComponent*

deduje *ToolTopComponent*. Temu so dodane še tabele, kjer je mogoče dodajati spojine in opisovati reakcije ter metode, ki dopuščajo manipulacijo s podatki. V modulu je prikazano tudi kako se podatki zapisujejo v razred, ki implementira *AbstractModelTool*, saj se ta edini ob shranjevanju serializira. Vsaka reakcija, ki jo s pomočjo urejevalnika opišemo, predstavlja svoj razred. Vsak modul mora narediti svojo implementacijo reakcije. Za ta primera služi preprost razred z metodama *getReaction* in *setReaction*, ki deduje razred *AbstractReactionType*.

2.4.2 TestParamEvalTool

Komponenta služi kot nadgradnja komponente za modeliranje. V tej komponenti določimo začetne vrednosti v reakcijah. Olajša nam delo, saj imamo lahko več simulacij z istim modelom vendar z različnimi začetnimi vrednosti. V testni implementaciji smo dodali masko, kjer določamo začetne vrednosti spojin, katere dobimo od predhodne komponente, ki mora biti tipa *AbstractModelTool*. Spojinam lahko določamo začetno vrednost, brisanje in dodajanje pa se mora zgoditi na predhodnem modulu. Dodan je bil tudi gumb, ki ponastavi vse spremembe in ponovno naloži seznam spojin s predhodnega modula, kar je potrebno narediti ob spremembah.

2.4.3 TestEngineTool

Gre za preprost modul, ki deduje *AbstractEngineTool*. Modul nima predvidene maske za urejanje, saj je tudi ne potrebuje. Namen modula je izvedba računskih operacij nad podatki predhodnega modula. Ta mora biti tipa *AbstractModelTool* oziroma *AbstractParamEvalTool*. Razredu *AbstractEngineTool* smo dodali spremenljivko *lineChartData*, ki je tipa *ObservableList <XYChart.Series <Double, Double>>* in vsebuje podatke za risanje na graf. Spremenljivka se ne serializira, zato se podatki po zaprtju programa ne shranijo, razen če modul tega ne naredi eksplicitno. Poleg spremenljivke, ki je javno dostopna preko metod, razred vsebuje še abstraktno metodo *calculate*, katero mora implementirati vsaka komponenta posebej. Funkcijo *calculate* se praviloma kliče s komponente za prikaz podatkov, ki sproži izračun podatkov. Zato jo moramo sprožiti preden želimo dostopati do podatkov, oziroma jo moramo poklicati, ko želimo podatke osvežiti.

Ker lahko računanje postane zelo obsežno in požrešno, je mogoče računske operacije prestaviti iz Jave v domorodno kodo (angl. *native code*). *Java Native Interface* (JNI) je Javansko programsko okolje, ki omogoča Javanski kodi, ki teče na Virtualnem stroju,

poganjati domorodne aplikacije in knjižnice napisane v jezikih kot so C, C++ in tudi assembler. Ta funkcionalnost je bila še posebej uporabna v preteklosti, ko Java še ni imela dostopa do določenih operacijskih objektov (npr. register) in podedovanih sistemskih knjižnic za optimizacijo in pospešitev kritičnih delov aplikacije. Z novejšimi verzijami Jave, ki ponujajo boljšo podporo manjkajočih funkcij, je potreba po tem zapadla, po drugi strani pa so se razvili bolj specifični programski vmesniki. Z verzijo Jave 1.3 se je močno optimiziral tudi JVM in matematični paketi, kot je npr. *BigInteger*, so bili v celoti predelani in podrobno uglašeni za najboljšo performanco. Zato je potrebno izvesti analizo naše kode, ter preveriti smiselnost pisanja domorodne kode. Moramo se zavedati, da aplikacija z domorodno kodo izgubi neodvisnost od sistema in mora biti prilagojena za vsak operacijski sistem posebej. Podrobnosti o klicanju domorodne kode so v razdelku 3.5 [7].

2.4.4 TestPlotTool

Modul poleg glavnega razreda *TestPlotTool* vsebuje razred *TestPlotToolTopComponent* za prikaz grafa in razred *SampleTableModel* za hranjenje podatkov grafa. Razredu *TestPlotToolTopComponent* vsebuje *JScrollPane* komponento, kjer bo prikazan graf ter gumb, ki sproži ponovni izris grafa. Sam izris grafa se sproži že ob odprtju komponente, vendar le če so podatki v predhodnem modulu tipa *EngineTool* že na voljo. Ob klicu funkcije za izris podatkov se le-ti najprej interpretirajo v razredu *SampleTableModel*. V tem razredu se oblikujejo v format primeren za izris, za kar poskrbi *JavaFX*. Za prikaz podatkov je uporabljen razred tipa *LineChart*, ki podatke prikazuje v obliki črt na dveh dimenzijah.

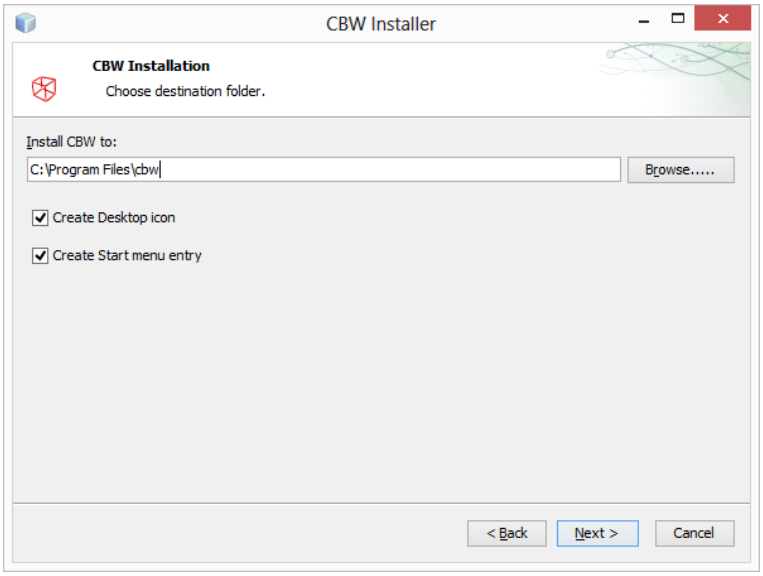
2.5 Izdelava namestitvenega paketa

NetBeans nam omogoča, da lahko aplikacije zapakiramo za distribucijo na več načinov. Med drugimi lahko aplikacijo izvozimo kot *Mac OS X* aplikacijo, namestitveni program za *Microsoft Windows* okolje, ali pa kot *ZIP* paket.

2.6 Namestitev in uporaba aplikacije

Namestitev in uporaba aplikacije bo prikazana na *Microsoft Windows* okolju. Namestitev je na vsakem operacijskem sistemu drugačna, vendar je uporaba zaradi Jave enaka, le

vmesnik je bolj prilagojen zgledu operacijskega sistema.



Slika 2.6 Namestitveni čarovnik CBW aplikacije.

2.6.1 Namestitev aplikacije

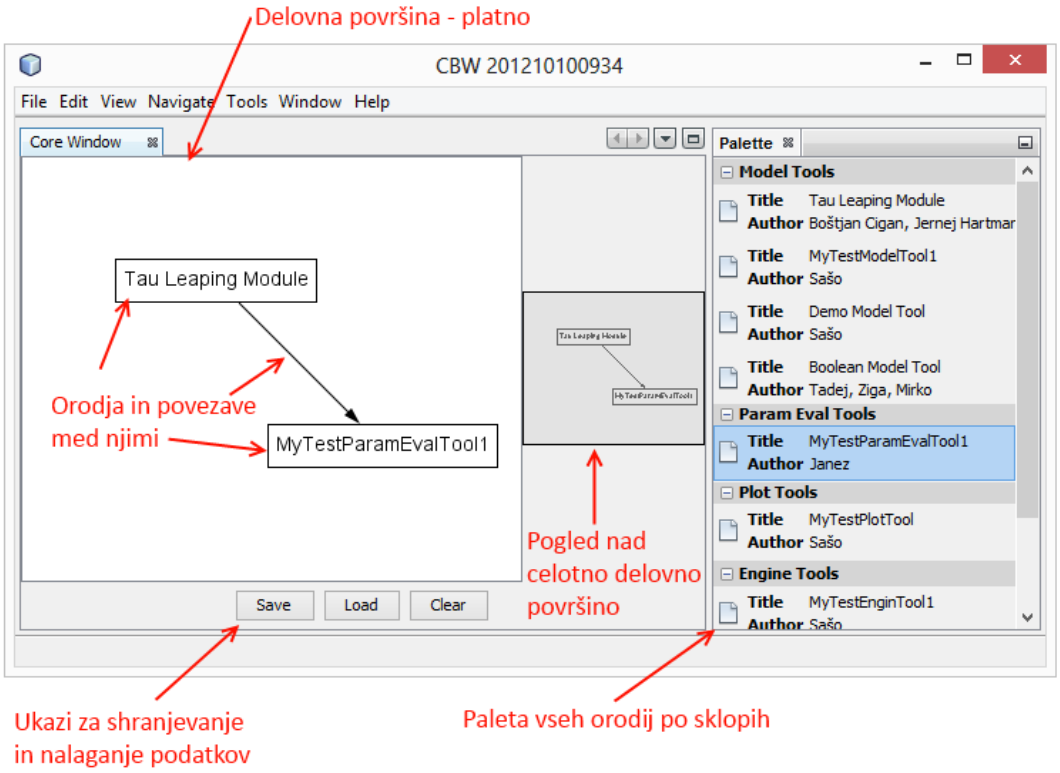
Aplikacijo lahko na *Microsoft Windows* okolje namestimo preko *namestitvenega čarovnika*, kjer samo izberemo lokacijo namestitve (glej sliko 2.6). Čarovnik poskrbi za razširitev datotek, ustvari bližnjico do aplikacije ter uporabniku specifično mapo, kjer zapisuje uporabniške nastavitve tekom uporabe programa. Namestitev aplikacije ustvari tudi program za odstranjevanje (*uninstall.exe*), ki odstrani aplikacijo kot tudi lokalne nastavitve.

Alternativa je uporaba *ZIP* distribucije, kjer aplikacijo preprosto razširimo v poljubno mapo in jo poženemo.

2.6.2 Uporaba aplikacije

Osnovna dva dela aplikacije sta paleta (*Palette*), ki vsebuje razpoložljive komponente (*Tools*) in delovna površina znotraj glavnega okna (*Core window*), v katero razpoložljive gradnike razmeščamo in med seboj povezujemo (glej sliko 2.7). Komponente na delovni površini obravnavamo kot vozlišča (*nodes*), ki so medsebojno povezana z usmerjenimi povezavami, pri čemer usmerjenost povezav nakazuje tok podatkov.

Komponente lahko med seboj povežemo tako, da držimo tipko *Ctrl*, izberemo iz-



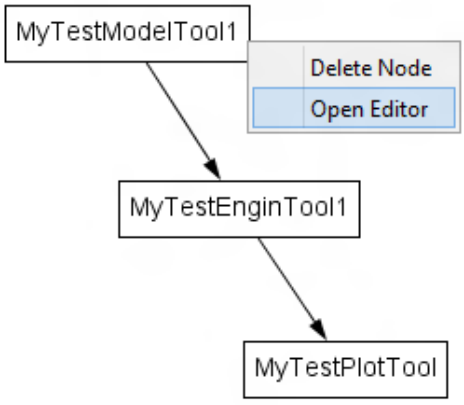
Slika 2.7 Opis komponent aplikacije.

hodiščno komponento ter z miško povlečemo povezavo v obliki puščice do ciljne komponente. Obstoječe povezave lahko izbrišemo z izbiro ukaza iz spustnega menija, ki ga dobimo ob desnem kliku miške na povezavo, z desnim klikom na komponento pa se nam prikaže spustni meni, ki nam ponudi brisanje ali dodatno urejanje komponente (glej sliko 2.8).

Čeprav aplikacija trenutno dopušča povezovanje vseh komponent med seboj je potrebno biti pazljiv, saj so komponente odvisne od predhodnikov (predhodnik komponente *plot* mora na primer biti komponenta *engine*). Primer pravilnega povezovanja je viden na sliki 2.8. Komponenta tipa *PlotTool* dobi podatke iz komponente tipa *EngineTool*, le-ta pa iz komponente tipa *ModelTool*.

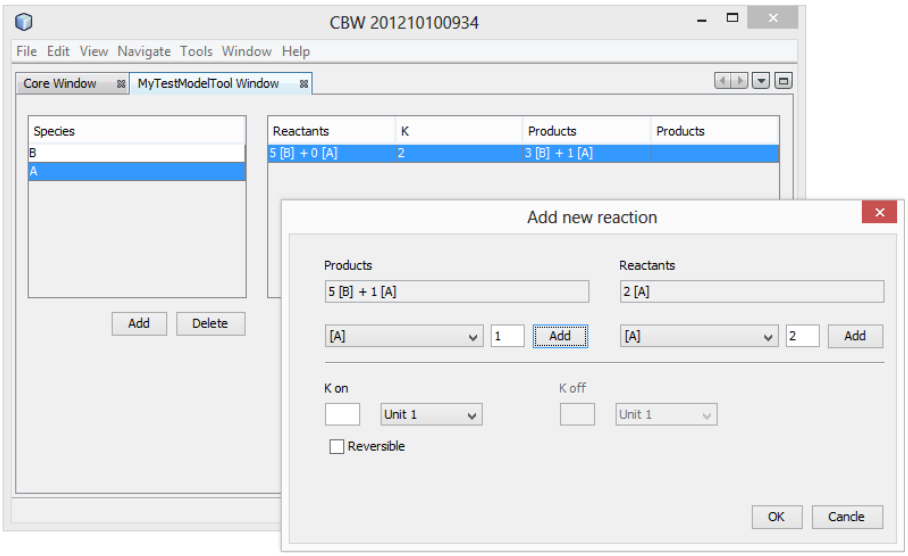
Postavitev modela

Znotraj urejevalnika za komponente tipa model, postavimo model, kjer se bodo izvajale reakcije, ki jih bomo simulirali. Zato znotraj modela, poleg drugih modelu specifičnih



Slika 2.8 Povezovanje komponent.

parametrov, določimo kemijske zvrsti in opišemo reakcije, ki se bodo izvajale (slika 2.9).



Slika 2.9 Urejanje modela.

Določitev parametrov

Ta komponenta deluje v sožitju s komponentami tipa model saj se tu določi začetne vrednosti sistema, ki smo ga definirali. Te bi lahko določili že znotraj modela, vendar lahko na ta način simuliramo enak model z različnimi začetnimi pogoji brez spreminjanja osnovno definiranega modela. Pozorni moramo biti, da po spremembah modela

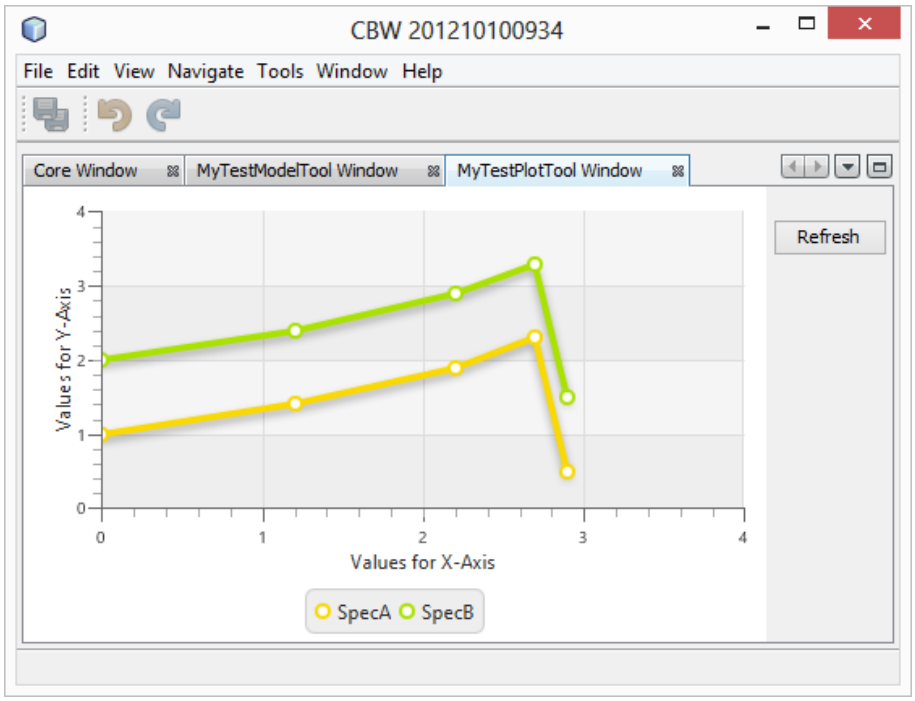
popravimo tudi vse odvisne komponente za določanje parametrov. To še posebej velja za dodajanje ali odstranjevanje spojin.

Simulacija

Praviloma komponenta nima maske za urejanje, saj znotraj nje potekajo računske operacije, ki simulirajo reakcije definirane v modelu. Vsaka komponenta implementira svoj matematični pristop k reševanju oziroma simuliranju kemijskih reakcij, katere niso povsem kompatibilne med seboj in so pogojno odvisne od tipa modela. Zato moramo pri izbiri biti pazljivi.

Prikaz podatkov

Komponenta za prikaz podatkov je še najbolj neodvisna komponenta, saj z izbiro poljubne komponente vplivamo le na prikaz, sami podatki pa ostanejo nespremenjeni. Prikaz podatkov je lahko v tekstovni obliki, kot dvodimenzionalni graf ali v kakšni drugi obliki (glej sliko 2.10).



Slika 2.10 Prikaz podatkov simulacije.

3 Uporaba komponent za bodoče programerje

Za lažje razvijanje novih komponent oziroma modulov bomo v tem razdelku bolj podrobno predstavili strukturo aplikacije in opisali njene glavne module, ter prikazali celoten postopek izdelave novega modula.

3.1 Okolje

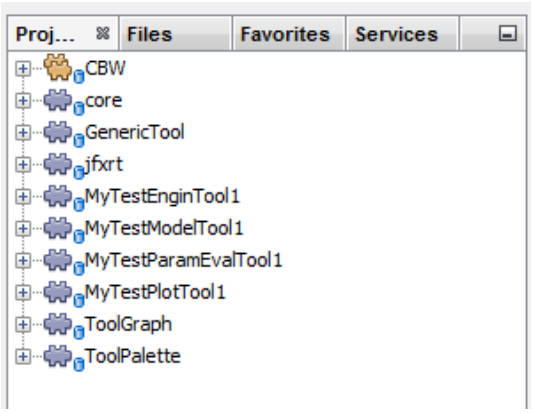
Aplikacija za svoje ogrodje uporablja *NetBeans Platform*. Za največjo kompatibilnost razvijanja je potrebno, da ima programer nameščeno različico *NetBeans IDE 7.2.1*. Aplikacija je spisana za delovanje na *Microsoft Windows* operacijskem sistemu in zna povzročati težave na drugih platformah (zaradi JavaFX knjižnjic, ki se nahajajo v *./release/modules/bin*).

3.2 Struktura aplikacije

Aplikacija je razdeljena na glavne module in razširitve. Glavni moduli so:

- **Core**: glavno okno, ki se odpre ob zagonu aplikacije in je ogrodje za *ToolGraph* in *ToolPalette* module,

- **ToolGraph**: je implementacija *GraphScene* razreda in omogoča dodajanje vozlišč in povezovanje le-teh; imenujemo jo kar *delovna površina*,
- **ToolPalette**: je posebno okno v *NetBeans* platformi in se veže na obstoječe okno; v tem primeru se veže na delovno površino in vsebuje vse module, ki so najavljeni *NetBeans Lookup* servisu in so otroci razreda *GenericTool*,
- **GenericTool**: je paket za vse razširitvene module; glavni razred je *AbstractGenericTool*, katerega dedujejo vsi ostali abstraktni razredi: *AbstractEngineTool*, *AbstractPlotTool*, *AbstractModelTool* in *AbstractParamEvalTool* (ki deduje *AbstractModelTool*); modul ima še razred *ToolTopComponent*, katerega mora uporabiti vsaka komponenta, ki bo imela grafični vmesnik ter razred *AbstractReactionType*, ki je namenjen prenašanju informacij reakcije med komponentami,
- **jfxt**: knjižnica *JavaFX*, za risanje grafov.



Slika 3.1 Drevo modulov.

Ostali moduli so razširitveni demo moduli, ki nakazujejo delovanje in uporabo (glej sliko 3.1).

3.3 Generične komponente

Vse generične komponente so abstraktne implementacije abstraktnega razreda *AbstractReactionType*. Vsak razred oziroma komponenta ima svojo funkcijo. Za boljše razumevanje si bomo tukaj bolj podrobno pogledali 4 osnovne tipe komponent (mogoče je dodati tudi svojega):

- **AbstractModelTool**: primarna komponenta kjer definiramo model preko opazovanih kemijskih reakcij in njihovih parametrov, zato mora biti prva komponenta v mreži, ki jo kreiramo na delovni površini; model ima dve spremenljivki, *species* in *reactions*, ter njihove *set* in *get* metode; obe spremenljivki sta tipa *List*, le da je spremenljivka *species* seznam tipa *String*, v katerem hranimo seznam vseh kemijskih zvrsti, spremenljivka *reactions* pa je seznam naše implementacije abstraktnega razreda *AbstractReactionType*; vsak element v spremenljivki *reactions* predstavlja reakcijo, katero moramo sami definirati (reaktanti, produkti itd.); tako imajo nasledniki komponente (povezani na delovni površini) dostop do podatkov celotnega modela preko *get* metod,

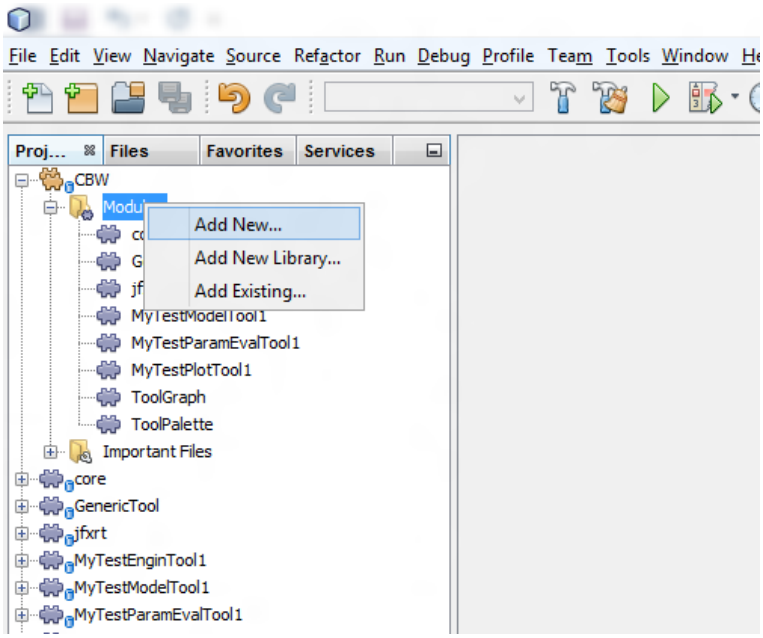
- **AbstractParamEvalTool**: komponenta deduje *AbstractModelTool* ampak ni namenjena definiranju reakcij, temveč omogoča, da v že definiranem modelu spremenimo vrednosti parametrov reakcij, ne da bi vplivali na izvirni model; tako lahko simuliramo več nastavitev hkrati; ker deduje *AbstractModelTool* ima tudi ta razred že definirane metode, katere mora prepisati, saj ne želimo da vrača izvirne podatke temveč spremenjene; paziti moramo tudi da naredimo svojo instanco podatkov, ter tako ne prepišemo izvornih,

- **AbstractEngineTool**: ko imamo definiran model, z *Engine* komponento definiramo začetno stanje v sistemu ter poženemo simulacijo; ker želimo podatke vizualizirati mora komponenta implementirati metodo *getLineChartData*, ki vrača spremenljivko tipa *ObservableList <XYChart.Series <Double, Double>>*; ker se spremenljivka neposredno prenese v inicializacijo grafa lahko tako vplivamo, kako se bodo podatki izrisali; komponenta ima tudi metodo *calculate*, s katero naj bi sprožili izračun,

- **AbstractPlotTool**: komponenta namenjena vizualizaciji rezultatov simulacije; predhodnik komponente na delovni površini mora biti tipa *AbstractEngineTool*, katerega podatke prikaže; za prikaz naj bi se uporabljala knjižnica *JavaFX*, ki podpira prikaz grafov različnega tipa in dinamično manipuliranje.

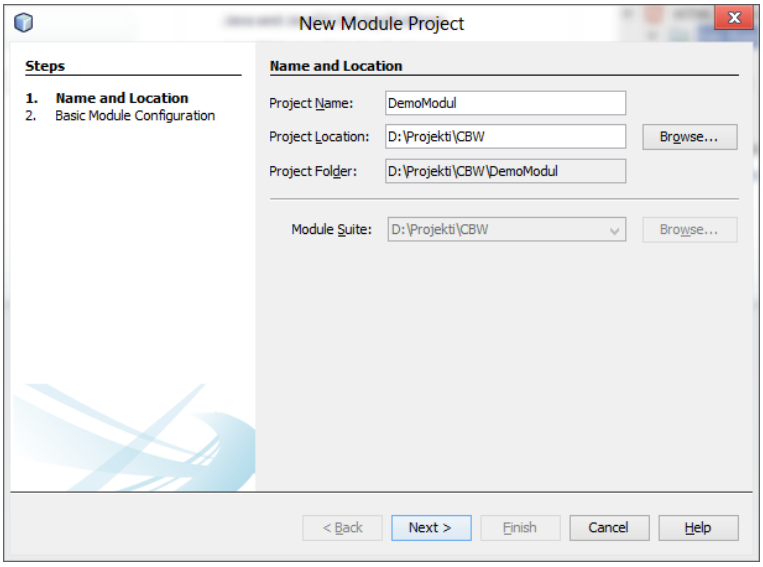
3.4 Pisanje novega modula

1. Začnemo tako, da s čarovnikom ustvarimo nov *NetBeans* modul (glej sliko 3.2).



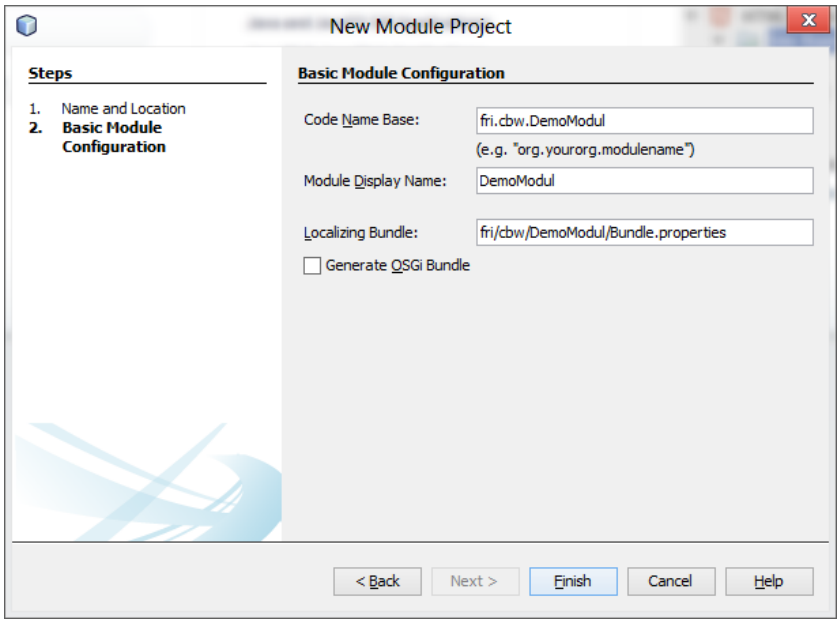
Slika 3.2 Dodajanje novega modula.

2. Izberemo ime modula in se prepričamo, da je lokacija projekta znotraj glavnega projekta (glej sliko 3.3).



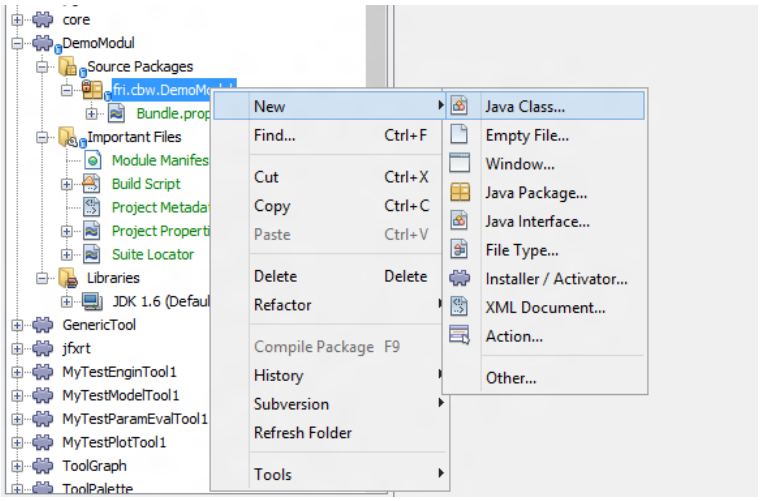
Slika 3.3 Izbiranje imena in mesta projekta.

3. Izberemo ime paketa in tako zaključimo s čarovnikom (glej sliko 3.4).



Slika 3.4 Definiranje paketa.

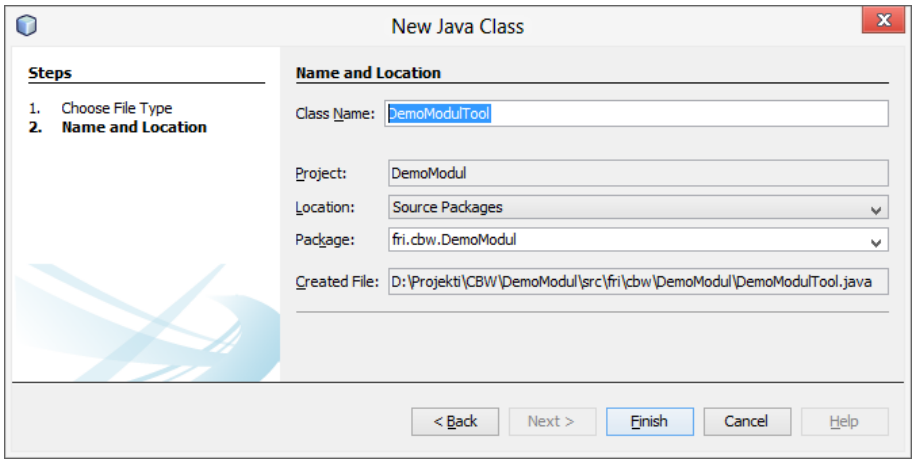
4. Začnemo s programiranjem lastne komponente. Najprej dodamo nov razred (glej sliki 3.5, 3.6), ki bo dedoval enega izmed abstraktnih razredov. S tem se bo prikazoval tudi na *ToolPalette* in ga bo moč prenesti na *ToolGraph*.



Slika 3.5 Dodajanje novega Java razreda.

V tem primeru bomo implementirali komponento za modeliranje, zato bo razred dedoval *AbstractModelTool*.

```
package fri.cbw.DemoModul;
import fri.cbw.GenericTool.AbstractModelTool;
public class DemoModulTool extends AbstractModelTool {


}
```



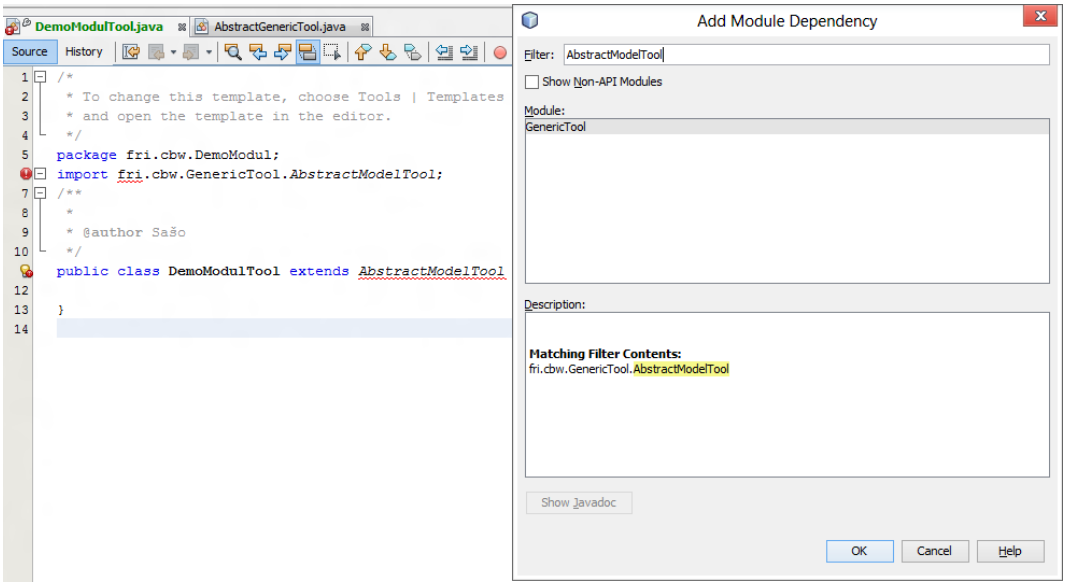
Slika 3.6 Definicija razreda.

Tu že vidimo prve probleme modularnega razvijanja, saj se moduli med seboj ne vidijo dokler ne določimo odvisnosti med njimi. V *NetBeans* okolju to preprosto naredimo preko iskalnika *Add Module Dependency* (glej sliko 3.7).

5. Modul *ToolPalette* sestavi paleto z razpoložljivimi komponentami s pomočjo knjižnice *Lookup API*, ki preišče vse module, ki imajo najavljene storitve (angl. *services*) določenega tipa razreda. Zato je potrebno modul najaviti sistemu *NetBeans service lookup*.

```
package fri.cbw.DemoModul;
import fri.cbw.GenericTool.AbstractModelTool;
```

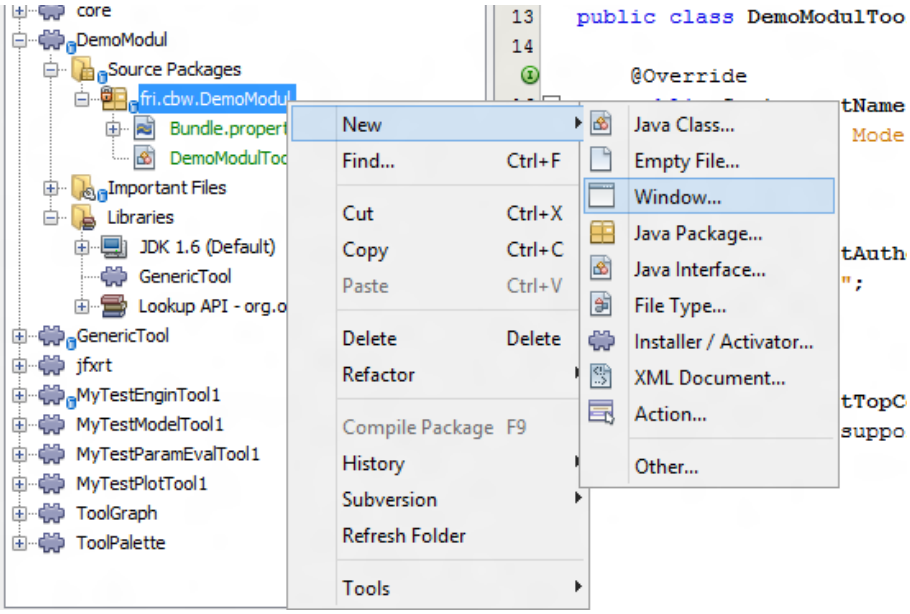
```java
@ServiceProvider(service = AbstractModelTool.class)
public class DemoModulTool extends AbstractModelTool {

    @Override
    public String getName() {
        return "Demo Model Tool";
    }

    @Override
    public String getAuthor() {
        return 'Avtor';
    }

    @Override
    public Class getTopComponentClass() {
         // Metodo bomo implementirali sele ko bomo naredili GUI za modul
        throw new UnsupportedOperationException("Not supported yet.");
    }
}
```



Slika 3.7 Urejanje odvisnosti.

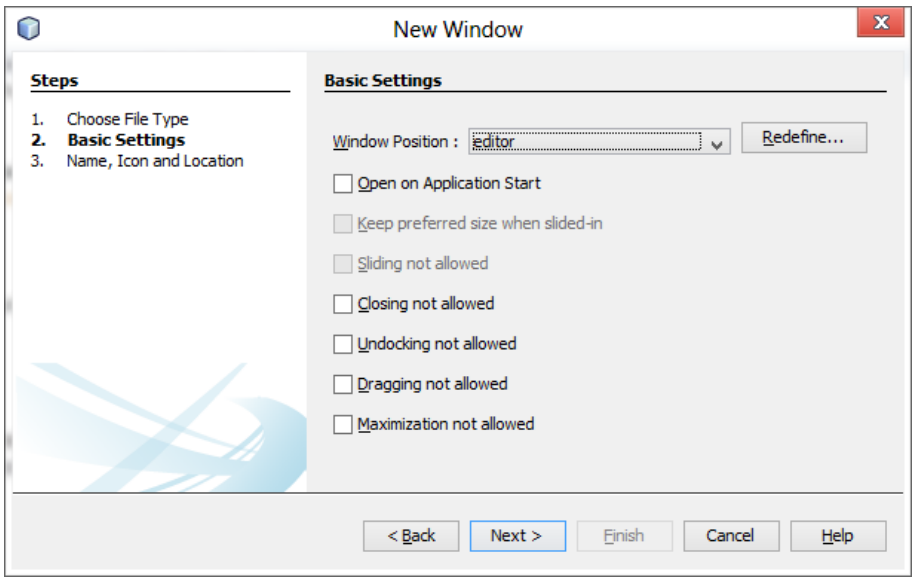
6. Registracijo naredimo na 2 načina. Lahko ustvarimo datoteko z imenom storitve v paketu *META-INF.services* (kot je to narejeno v *MyTestModelTool1*) ali pa preko notacij (kot je to narejeno v *MyTestEngineTool1*). V tem primeru ga bomo registrirali preko notacije in to z *@ServiceProvider(service = AbstractModelTool.class)*, ki bo javljala napako dokler modulu ne dodamo odvisnosti za *Lookup API*. Če zdaj zaženemo aplikacijo, bi morala biti komponenta *DemoModul* na paleti.

7. Komponenta potrebuje še grafični urejevalnik. *NetBeans* nam tu zopet olajša zadevo s čarovnikom *New Window* (slika 3.8).



Slika 3.8 Dodajanje novega okna.

8. V primeru želimo, da se okno odpre (slika 3.9) na poziciji *Editor* (osrednji del, o razporeditvi si lahko preberete več na http://ui.netbeans.org/docs/ui/ws/ws_spec-netbeans_ide.html). Poleg tega ne želimo da se okno odpre že ob zagonu applikacije (odpreti ga želimo šele ob sprožitvi urejanja komponente). Dodamo predpono in s čarovnikom smo zaključili. *NetBeans* nam zdaj odpre *Swing* urejevalnik preko katerega lahko z *drag and drop* komponentami ustvarimo vmesnik. Opazimo lahko, da je *NetBeans* čarovnik dodal tudi vse potrebne odvisnosti do knjižnic (npr. *Window System API*).



Slika 3.9 Postavitev okna.

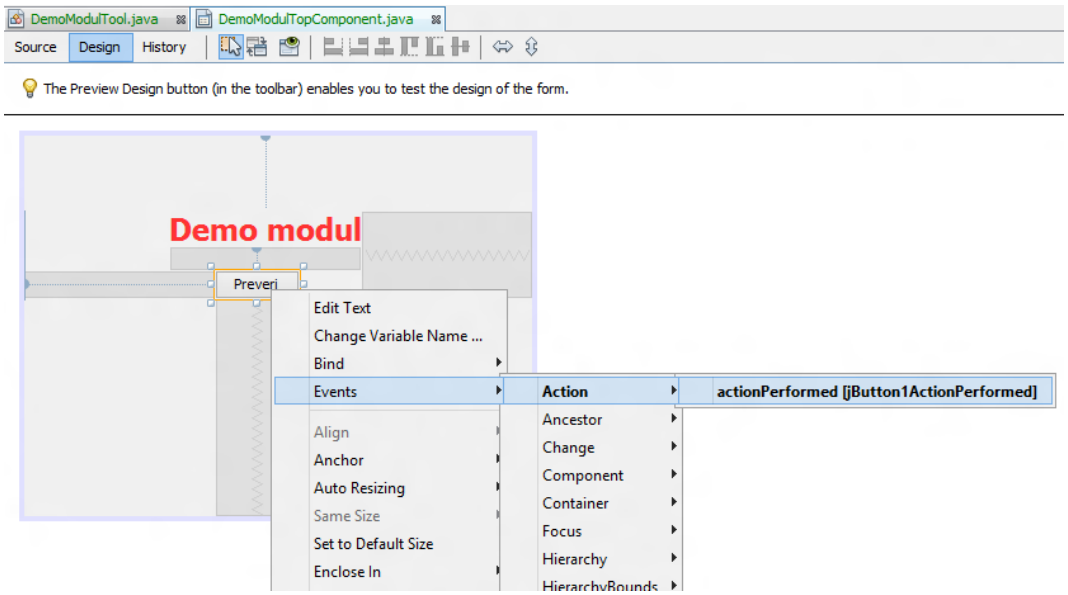
9. Preden se lotimo izdelave vmesnika ga najprej naredimo kompatibilnega s celotno aplikacijo. Naš *DemoModulTopComponent* razred trenutno deduje *TopComponent*. To je potrebo spremeniti v *ToolTopComponent*, saj nam ta omogoča povezavo do delovne površine in tako povezav med njim in ostalimi moduli. Spremenjeno dedovanje javi napako konstruktorja, saj ne obstaja super konstruktor brez parametrov. Zato našemu konstruktorju *DemoModulTopComponent()* dodamo parametre *GraphScene scene* in *IconNodeWidget toolNode* in v njem kličemo metodo *super(scene, toolNode)*.

```
public DemoModulTopComponent(GraphScene scene, IconNodeWidget toolNode) {
    super(scene, toolNode);
    initComponents();
    setName(Bundle.CTL\_DemoModulTopComponent());
    setToolTipText(Bundle.HINT_DemoModulTopComponent());
}
```

Spremenimo še *TopComponent* anotacijo *TopComponent.PERSISTENCE_ALWAYS* v *TopComponent.PERSISTENCE_NEVER*, da se okno po-nastavi ob izhodu apli-

kacije in zbrišemo anotacije *@ActionReference*, *@TopComponent.OpenActionRegistration*, *@ConvertAsProperties*, ki doda akcijo v meni *Window*. S tem onemogočimo prikaz urejevalnika, ne da bi imeli povezavo na delovno površino, kar bi povzročalo napake pri delovanju aplikacije.

Popravimo še *DemoModulTool* razred in implementiramo metodo *getTopComponentClass()*. Dodamo ji samo *return* stavek z vrednostjo *DemoModulTopComponent.class*. Zdaj lahko zaženemo aplikacijo. Dodamo komponento na delovno površino in z desnim klikom odpremo urejevalnik, ki je trenutno še prazen.



Slika 3.10 Dodajanje akcije.

10. Za Demo primer smo na urejevalni površini dodali oznako (angl. *label*) in gumb (slika 3.10), ki bo preveril ali je predhodna komponenta tipa *AbstractModelTool*.

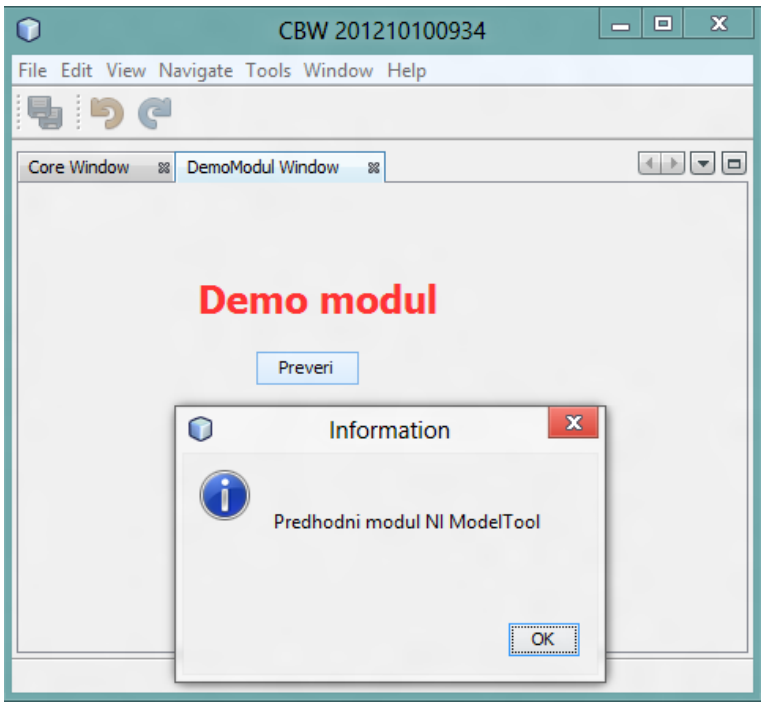
    Koda za gumb je sledeča (ne pozabite dodati odvisnosti: *Visual Library API, ToolGraph, Dialogs API, Utilities API*):

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt){
  NotifyDescriptor nd;

  try {
```

```
    ToolWrapper tool = (ToolWrapper)getScene().
                      findObject(getToolNode());
    ToolWrapper prevTool = tool.getPrevNode(getScene());
    AbstractModelTool gt = (AbstractModelTool)
                          prevTool.getNodeGenericTool();

    nd = new NotifyDescriptor.Message("Predhodni modul JE ModelTool");

  } catch(ClassCastException e){
    Logger.getAnonymousLogger().severe(e.getMessage());
    nd = new NotifyDescriptor.Message("Predhodni modul NI ModelTool");
  } catch(NullPointerException e){
    Logger.getAnonymousLogger().severe(e.getMessage());
    nd = new NotifyDescriptor.Message("Predhodni modul ne obstaja");
  } finally {
    DialogDisplayer.getDefault().notify(nd);
  }
}
```

Funkciji *getScene()* in *getToolNode()* se morata nahajati v *ToolTopComponent* razredu in sta referenci na delovno površino in vozlišče komponente, ki jo trenutno urejamo. S tem dobimo *ToolWrapper*, na delovni površini, ki ima metodo *getPrevNode()*. Ta vrne prvo predhodno komponento (ali *null*, če predhodna komponenta ne bostaja). V *ToolWrapper*-ju se nahaja tudi sama komponenta, ki jo dobimo z metodo *getNodeGenericTool*. Aplikacija bi na koncu morala izgledati kot slika 3.11.

11. Aplikacija omogoča shranjevanje in nalaganje delovne površine z vsemi nastavitvami. Akcije se izvedejo v modulu *ToolGraph* in je kombinacija XML strukture in serializacije objektov. Shrani se samo delovna površina (vozlišča in povezave) ter komponente vezana na njo (objekti ki dedujejo *AbstractGenericTool*). Grafični urejevalnik posamezne komponente se ne shrani, zato moramo poskrbeti, da se vsebina na maski prenese na objekt. To lahko naredimo na dva načina. Lahko poslušamo spremembe na maski ter podatke istočasno zapisujemo na objekt, lahko pa prepišemo (*@Override*) metodo *doSave()*, ki se nahaja v razredu *ToolTopComponent*. Za demonstracijo bomo na našo masko dodali še komponento *JTextField*,



Slika 3.11 Preverjanje predhodnega modula.

katere vrednost se bo shranila v naš *DemoModulTool*. V *DemoModulTool* dodamo novo spremenljivko *savedText* tipa *String* ter kreiramo set in get metodi. *JTextField* dodamo *focusLost event*, kjer dodamo kodo za shranjevanje teksta.

```
private void jTextField1FocusLost(java.awt.event.FocusEvent evt) {
        ((DemoModulTool)getGenericTool())
            .setSavedText(jTextField1.getText());
    }
```

Metoda *getGenericTool()* vrne naš *DemoModulTool*, od koder lahko dostopamo do *saveText* spremenljivke. Tekst bo tako ob shranjevanju serializiran, ampak ob nalaganju shranjenega dela, ne bo prikazan na maski. Pri inicializaciji *DemoModulTopComponent* moramo ovrednotiti vsa polja, ki smo jih shranili. V primeru bi na koncu konstruktorja dodali še:

```
jTextField1.setText(((DemoModulTool)getGenericTool()).getSavedText());
```

Če imamo v komponenti spremenljivke, za katere ne želimo shranjevanja (npr. izračun podatkov) jim moramo ob deklaraciji predpisati tip transient (npr. *private transient String s*;).

3.5 Klicanje domorodne kode

Če predvidevamo, da bo naš modul deloval veliko hitreje spisan v domorodni kodi, lahko to storimo preko JNI knjižnice. V spodnjem primeru je opisan postopek pisanja in klicanja C kode.

1. **Ustvarimo nov razred** *HelloJNI.java*, ki bo uporabljal kodo napisano v jeziku C,

```
public class HelloJNI {
   static {
      // Nalozi knjiznico helloJNI.dll (Windows) ali helloJNI.so (Unixes)
      System.loadLibrary("helloJNI");
   }
   // Domorodna metoda ki ne sprejme nobenega parametra in vraca vodi
   private native void helloWorld();

   public static void main(String[] args) {
      // Ob zagonu se sprozi klic domorodne metode
      new HelloJNI().helloWorld();
   }
}
```

ter ga prevedemo.

```
javac HelloJNICpp.java
```

2. **Ustvarimo datoteko z glavo**, katero bomo uporabili v naši C kodi. Naš razred poženemo čez orodje *javah*, ki bo ustvarilo datoteko – *HelloJNI.h*.

```
javah HelloJNI
```

Rezultat ukaza je sledeč:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class HelloJNI */

#ifndef _Included_HelloJNI
#define _Included_HelloJNI
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:    HelloJNI
 * Method:   helloWorld
 * Signature:
 */
JNIEXPORT void JNICALL Java_HelloJNI_helloWorld
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

Naša funkcija *helloWorld* se je preimenovala v *Java_HelloJNI_helloWorld*, po sledečem vzorcu poimenovanja – *Java_{paket_in_ime_razreda}_{ime_funkcije}(JNI argumenti)*. Vse pike se zamenjajo s podčrtaji.

3. **Implementiramo C kodo** v datoteki *HelloJNI.c*,

```
#include <jni.h>
#include <stdio.h>
#include "HelloJNI.h"

JNIEXPORT void JNICALL Java_HelloJNI_helloWorld(JNIEnv *env, jobject
    thisObj) {
   printf("Hello World!\n");
   return;
}
```

ter jo prevedemo s spodnjim ukazom (velja le če uporabljamo prevajalnik MinGW GCC na OS Windows).

```
gcc -Wl,--add-stdcall-alias -I"<JAVA_HOME>\include"
    -I"<JAVA_HOME>\include\win32" -shared -o hello.dll HelloJNI.c
```

Podrobnejšo razlago uporabljenih parametrov ter več primerov kode lahko najdemo v [20].

4 Sklepne ugotovitve

V okviru diplomske naloge smo razvili modularno orodje, ki je v prvi vrsti namenjeno vzpostavitvi in analizi modelov na področju sintezne in sistemske biologije. Orodje omogoča povezovanje različnih komponent, s katerimi lahko modeliramo, analiziramo in načrtujemo biološke sisteme. Glavna prednost orodja je njegova modularnost in odprtost za nadaljnji razvoj. S tem omogočamo uporabnikom in raziskovalcem možnost nadgradnje orodja v predvidevanih okvirjih z lastnimi komponentami ali celo izdelavo novih tipov komponent in storitev, ki v prvi fazi razvoja niso bile zamišljene.

V prihodnosti bi lahko orodje izboljšali z bolj striktno definicijo komponent ter povezljivosti med njimi. Ta bi na prvi pogled omejila prosto programiranje, ki ga zdaj dopušča, s tem pa bi zagotovili boljšo kompatibilnost med moduli ter zagotovili pravilen prenos podatkov med njimi. Večji poudarek bi lahko namenili tudi storitvam, katere bi ponujali in se jih posluževali moduli znotraj orodja. Tako bi lahko nove komponente nadgrajevale obstoječe komponente brez eksplicitne povezave s strani uporabnika. Potrebno se je dotakniti tudi grafičnega vmesnika in uporabniku zagotoviti boljšo izkušnjo, predvsem z jasnimi opozorili, smernicami in namigi.

LITERATURA

[1] M. Moškon, J. Bordon, M. Mraz, N. Zimic, M. Petroni, *Recent advances in systems biology*, Computational approaches in synthetic and systems biology, str. 1 – 27, Nova Science Publishers, 2013.

[2] D. Chandran, F. Bergmann, and H. Sauro, "*Tinkercell: modular CAD tool for synthetic biology*," Journal of Biological Engineering, vol. 3, no. 1, str. 19, 2009.

[3] P. Mendes, S. Hoops, S. Sahle, R. Gauges, J. Dada, and U. Kummer, "*Computational modeling of biochemical networks using COPASI*," in Systems Biology, ser. Methods in Molecular Biology, I. V. Maly, Ed. Humana Press, 2009, vol. 500, str. 17 – 59.

[4] Mathworks, 2013, http://www.mathworks.com/products/simbiology/.

[5] Oracle, 2012, *The Reflection API*, The Java^{TM}Tutorials, 15. oktober 2012, http://docs.oracle.com/javase/tutorial/reflect/index.html.

[6] I. R. Forman, N. Forman. *Java Reflection in Action*, 2004, Manning publications.

[7] J. Bloch, *Effective Java: Second Edition*, 2008, Addison-Wesley.

[8] The Apache Software Foundation, 2012, *What is maven*, Maven, 30. oktober 2012, http://maven.apache.org/what-is-maven.html.

[9] A. de Castro Alves, *OSGi in depth*, 2011, Manning publications.

[10] M. Grammling, B. Streckfus, 2009, *OSGi Layering*, 13. junij 2013, http://en.wikipedia.org/wiki/File:Osgi-system-layering.svg.

[11] J. Tulach, 2009, *OSGi*, Practical API Design, 5. november 2012, http://wiki.apidesign.org/wiki/OSGi.

[12] NetBeans, 2012, *NetBeans Platform Quick Start Using OSGi*, NetBeans, 30. oktober 2012, https://platform.netbeans.org/tutorials/nbm-osgi-quickstart.html.

[13] H. Böck, *The Definitive Guide to NetBeans™ Platform 7*, 2011, Apress.

[14] NetBeans, 2012, *NetBeans Visual Library*, NetBeans, 5. november 2012, https://platform.netbeans.org/graph/.

[15] Oracle, 2012, *What Is JavaFX?*, JavaFX Documentation, 12. november 2012, http://docs.oracle.com/javafx/2/overview/jfxpub-overview.htm.

[16] NetBeans, 2012, *NetBeans API List*, NetBeans API, 17. november 2012, http://bits.netbeans.org/7.3/javadoc/.

[17] Oracle, 2012, *Introduction to DnD*, The Java™ Tutorials, 17. november 2012, http://docs.oracle.com/javase/tutorial/uiswing/dnd/intro.html.

[18] NetBeans, 2012, *IconNodeWidget*, Visual Library API, 22. november 2012, http://bits.netbeans.org/dev/javadoc/org-netbeans-api-visual/org/netbeans/api/visual/widget/general/IconNodeWidget.html.

[19] Oracle, 2012, *Annotation Type Transient*, Java Platform SE 7, 30. november 2012, http://docs.oracle.com/javase/7/docs/api/java/beans/Transient.html.

[20] C. Hock-Chuan, 2013, *Java Native Interface (JNI)*, Yet another insignificant... programming notes, 20. januar 2013, http://www3.ntu.edu.sg/home/ehchua/programming/java/JavaNativeInterface.html.