Univerza v Ljubljani
Fakulteta za računalništvo in informatiko ter
Fakulteta za matematiko in fiziko

Nejc Škofič

Informacijski sistem za planiranje operacij v skladišču

diplomsko delo
na interdisciplinarnem univerzitetnem študiju
računalništva in matematike

Mentor: akad. prof. dr. Ivan Bratko

Ljubljana 2012

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

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: 00041/2012
Datum: 03.10.2012

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

Kandidat: **NEJC ŠKOFIČ**

Naslov: **INFORMACIJSKI SISTEM ZA PLANIRANJE OPERACIJ SKLADIŠČA**

INFORMATION SYSTEM FOR OPERATION PLANNING IN A WAREHOUSE

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Naloga je razviti informacijski sistem za podporo planiranja zbiranja blaga za naročila ob izdaji blaga. Namen planiranja je optimizacija operacije fizičnega nabiranja blaga, ki jo izvajajo delavci v skladišču. Sistem naj bo zgrajen generično, tako da ga je mogoče uporabljati prožno in ima naslednje lastnosti:
(1) možno ga je uporabiti za različna skladišča, ki imajo različne topologije,
(2) možno ga je navezati na različne metode planiranja operacij, ki se vstavijo v sistem kot planirni modul,
(3) omogoča uporabo v povezavi s podatkovno bazo skladišča, v kateri se vodi stanje zalog,
(4) sistem se smiselno odziva na nepričakovane dogodke, kot je neskladje med stanjem zalog v bazi in dejansko opaženim stanjem v skladišču.

Mentor:

akad. prof. dr. Ivan Bratko

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan Fakultete za računalništvo in informatiko:

prof. dr. Nikolaj Zimic

Dekan Fakultete za matematiko in fiziko:

akad. prof. dr. Franc Forstnerič

UNIVERZA V LJUBLJANI • Fakulteta za matematiko in fiziko • 1

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Nejc Škofič, z vpisno številko **63060246**, sem avtor diplomskega dela z naslovom:

Informacijski sistem za planiranje operacij v skladišču

S svojim podpisom zagotavljam, da:

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

V Ljubljani, dne 11. septembra 2012 Podpis avtorja:

Za mentorstvo, vodenje in pomoč pri izdelavi diplomske naloge se zahvaljujem dr. Ivanu Bratku.

Za številne diskusije se zahvaljujem dr. Mateju Guidu in dr. Aleksandru Sadikovu. Marsikatera ideja je bila plod teh razprav.

Zahvaljujem se tudi Dejanu Reichmannu in Juriju Šorliju iz podjetja 3R.TIM za praktične poglede in probleme informacijskih sistemov v skladiščih.

Za testiranje in uporabo razvitega sistema in testnega okolja se zahvaljujem Maticu Horvatu. Z njegovimi idejami in mnenji je sistem postal prijaznejši za uporabo.

Zahvaljujem se tudi sošolcem, še posebej Roku, Anžetu, Gaji in Slavku. Brez njih študij ne bi bil enak.

Zahvala gre tudi družini in Meti za vso podporo in spodbudo.

Kazalo

Seznam uporabljenih kratic in simbolov

API Application programming interface – programski vmesnik

IDE Integrated development environment – razvojno okolje

JAR Java archive – arhiv, ki predstavlja javansko knjižnico ali aplikacijo

Java SE Java Standard Edition – standardna verzija programskega jezika Java

JDBC Driver Java Database Connectivity Driver – javanski gonilnik za povezavo s podatkovno bazo

JDK Java Development Kit – orodja za razvijanje aplikacij v programskem jeziku Java

JPA Java Persistence API – programski vmesnik in ogrodje za upravljanje s podatki v relacijskih podatkovnih bazah znotraj javanskega okolja

JVM Java Virtual Machine – javanski virtualni stroj

MS SQL Server – relacijska podatkovna baza podjetja Microsoft

URL Uniform resource locator – naslov URL

WMS Warehouse management system – sistem za upravljanje skladišča

XML Extensible markup language – razširljivi označevalni jezik

Povzetek

Večina skladišč izpolnjuje naročila po principu pobiralec k blagu. Ta način dela je v večini skladišč realiziran tako, da pobiralec sam načrtuje vrstni red pobiranja in pot v skladišču. To zahteva najmanj avtomatizacije v skladišču, hkrati pa zahteva največ dela od pobiralcev. Premiki pobiralcev v skladišču, medtem ko izpolnjujejo naročila, zahtevajo tudi do polovico vsega delovnega časa. Ta čas lahko zmanjšamo z optimizacijo plana pobiranja in vodenjem pobiralcev v skladišču do želenega blaga.

V tem diplomskem delu smo v sodelovanju s podjetjem *3R.TIM* in Laboratorijem za umetno inteligenco, Fakultete za računalništvo in informatiko, Univerze v Ljubljani razvili informacijski sistem, ki omogoča izdelavo planerjev za optimizacijo dela pobiralcev. Naša rešitev predstavlja pomožni sistem že obstoječim sistemom za upravljanje skladišč. Sposobna je modeliranja skladišč in izvajanja planiranja s poljubnimi planerji. Z razvitim ogrodjem za pisanje planerjev je mogoče enostavno implementirati poljubne planerje in jih uporabiti pri planiranju. Za pomoč pri testiranju in vizualizaciji ustvarjenih planov dela smo razvili tudi testno okolje in uporabniški vmesnik z vizualizacijo simulacije planiranja. Z razvitim sistemom smo omogočili možnost optimizacije dela pobiralcev tudi v skladiščih, kjer zaradi načina dela ta do sedaj ni bila na voljo.

Ključne besede:
Skladišče, Pobiranje naročil, Informacijski sistem, Planiranje

Abstract

Most of the warehouses employ "picker-to-parts" order picking system. This system is usually implemented so that a picker himself plans the order of picking and the route through the warehouse. This requires little warehouse automation but on the other hand pickers do all the work required for order fulfillment. Tipically their picking plans are far from optimal. Time spent for picker movement can be as much as half of the total working time. With optimized working plan and efficient pickers' routing it is possible to lower required movement time and thus optimize order picking in a warehouse.

In this thesis we developed information system for optimizing order picking process in cooperation with the *3R.TIM* company and Artificial Intelligence Laboratory, Faculty of computer and information science, University of Ljubljana. Our solution is an additional system to warehouse management systems which is capable of warehouse modeling and running of arbitrary planners. New planners can be easily implemented in the developed framework for planner writing and running. We developed a testing environment and a user interface with visualization of plan execution for easy planner testing and plan visualization. With our information system it is possible to optimize order picking even in warehouses where such optimization was not available because of "picker-to-parts" order picking system.

Keywords:
Warehouse, Order picking, Information system, Planning

Poglavje 1

Uvod

Informacijski sistemi v skladiščih omogočajo upravljanje skladišča in pregled nad stanjem skladišča. Primeri takih sistemov so sistemi za upravljanje virov skladišča in sistemi za upravljanje z zalogo v skladišču. Avtomatizirana skladišča vsebujejo tudi podsisteme in krmilnike, ki skrbijo za avtomatizirano polnjenje in vračanje blaga ter optimizacijo skladiščnega prostora. Vendar je število avtomatiziranih skladišč majhno - v zahodni Evropi le okrog 20% [1]. V večini skladišč se delavci z vozički ali viličarji premikajo med regali in skladiščijo oziroma pobirajo blago. Plan dela glede na pričakovano dobavo blaga in naročila ustvarjajo vodje skladišča ali delavci sami.

Z vse zahtevnejšim trgom in vse večjo težnjo po manjšanju stroškov, se je pojavila želja po hitrejšem in učinkovitejšem izpolnjevanju naročil tudi v neavtomatiziranih skladiščih. Pobiralec v teh skladiščih naj bi porabil približno polovico svojega časa za premikanje po skladišču [1]. Da bi zmanjšali ta čas, so bile predlagane rešitve, ki narekujejo način skladiščenja, planiranje pobiranja blaga glede na naročila in vodenje pobiralca po skladišču. Le hkratna optimizacija vseh treh vrst rešitev naj bi dala globalno rešitev problema minimizacije pobiralčevega časa sprehoda po skladišču pri pobiranju blaga.

Podjetje *3R.TIM* je slovenski ponudnik programskih rešitev s področja logistike in e-poslovanja. Njihova rešitev *Hydra@Warehouse* omogoča upravljanje skladišča, pregled nad zalogo in posredovanje naročil pobiralcem z

brezžično tehnologijo v neavtomatiziranih skladiščih. V sodelovanju z Laboratorijem za umetno inteligenco na Fakulteti za računalništvo in informatiko, Univerze v Ljubljani, smo razvili rešitev, ki omogoča planiranje dela pobiralcev v skladišču.

Naša rešitev predstavlja dodatni sistem sistemom upravljanja skladišča, ki omogoča ustvarjanje optimiziranih planov dela pobiranja blaga glede na naročila. Sistem sprejme naročila in vrne plan dela za pobiralce, ki je odvisen od uporabljenega planerja, naročil, skladišča in trenutne zaloge. Plan dela generirajo poljubni planerji, ki so implementirani znotraj razvitega okolja za pisanje planerjev. Sistem se odziva na izjemne dogodke, kot je neujemanje elektronske zaloge z dejansko, in zna vzpostaviti svoje stanje v primeru sesutja. Razvili smo tudi grafično vmesnik z vizualizacijo simulacije izvajanja plana pobiranja naročil.

Preostanek diplomskega dela je razdeljen na sledeča poglavja. V Poglavju 2 je opisan pomen skladiščenja, operacije v skladišču ter različni tipi skladišč in informacijskih sistemov za upravljanje skladišč. V Poglavju 3 je opisana specifikacija naše rešitve in uporabljene tehnologije. V Poglavju 4 je predstavljena celotna struktura rešitve, ki je podrobneje opisana v naslednjih treh poglavjih. V Poglavju 5 je opisan naš način modeliranja skladišča, v Poglavju 6 naša implementacija ogrodja za pisanje planerjev in v Poglavju 7 kontrolna enota, ki skrbi za komunikacijo z zunanjim okoljem. V Poglavju 8 je predstavljen uporabniški vmesnik za vizualizacijo simulacije pobiranja blaga in s Poglavjem 9 zaključimo diplomsko delo.

Poglavje 2

Pomen skladiščenja v logistiki

Skladiščenje je pomemben proces, ki povezuje proizvajalce s potrošniki oziroma končnimi ponudniki izdelkov. Proces skladiščenja je definiran kot del logističnega sistema podjetja, ki shranjuje blago (surovine, polizdelke ali končne izdelke) kot vmesno stopnjo med proizvajalci blaga in porabniki. Poleg shranjevanja blaga omogoča nadzor nad stanjem zaloge, lokacijo blaga znotraj skladišča [2].

V naslednjih odstavkih bomo opisali nekatere izmed ciljev, ki jih poskušamo doseči s skladiščenjem blaga [2, 3].

Zagotavljanje zaloge potrebno za izpolnitev naročil: Stranke od skladišča pričakujejo takojšnjo izpolnitev naročil. Če časi izdaje naročil niso vnaprej znani oziroma če se vrsta blaga in količina med naročili razlikuje, mora zaloga skladišča vsebovati dovolj blaga, da lahko pokrije pričakovana naročila.

Zagotavljanje nemotene oskrbe: S skladiščenjem blaga se zagotavlja dobavo tudi v primeru motenj v produkciji ali dobavi. Tak primer so naravne nesreče, zaradi katerih trpi proizvodnja ali motnje v transportu, kot so na primer stavke prevoznikov. V teh primerih skladišče še vedno vsebuje zalogo, ki jo lahko dostavi strankam.

Zmanjšanje cene transporta blaga: Eden glavnih razlogov za uporabo skladišč je zmanjšanje cene transporta blaga z organizacijo dobave, kjer

je transportno sredstvo kar najbolj izkoriščeno. Skladišče naroča od proizvajalca veliko količino blaga, ki jo dostavlja svojim strankam po njihovih potrebah. Cenejši transport in cena blaga zaradi velikega naročila v kombinaciji z lokacijo skladišča, ki je običajno v bližini svojih strank, omogočata bolj učinkovit in cenejši transport.

Zagotavljanje centralne oskrbe z blagom večjega števila proizvajalcev: Skladišče lahko upravlja z zalogami večjega števila proizvajalcev. Stranka tako lahko na enem mestu naroča blago različnih proizvajalcev, ki bodo dostavljeni z enim transportom.

Opravljanje storitev z dodano vrednostjo: Skladišča poleg hrambe blaga opravljajo tudi storitve z dodano vrednostjo. Primera takih storitev sta prepakiranje blaga ter lepljenje nalepk in etiket, ki jih opravijo v skladu z željami strank.

2.1 Operacije v skladišču

Skladiščenje ima tri osnovne funkcije - premikanje in shranjevanje blaga ter prenos informacij.

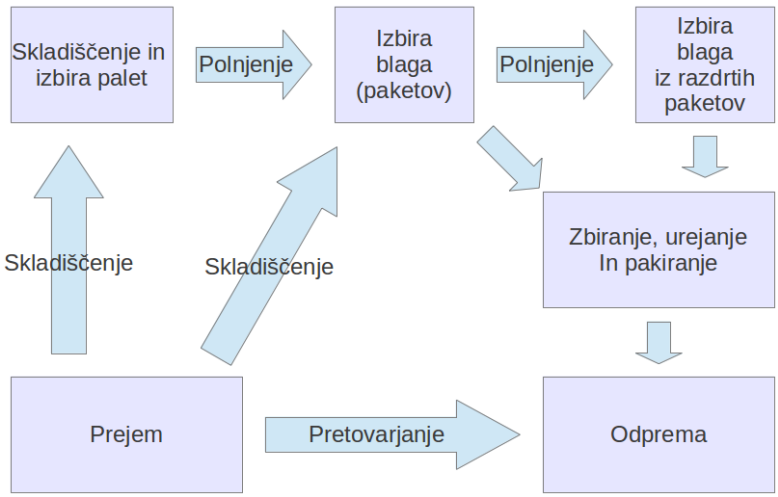
Premik blaga od sprejema do odpreme lahko razdelimo na pet korakov [2, 3]:

Sprejem blaga - blago se raztovori, pregleda se njegovo ustreznost glede na podatke na dobavnici, pregleda se stanje blaga ter posodobi stanje zaloge.

Skladiščenje - blagu se dodeli lokacije znotraj skladišča, izvede se transport blaga v skladišče in posodobi se lokacija zaloge.

Izbira blaga/pobiranje naročil - delavci ali stroji izberejo blago iz zaloge, ki ustrezajo zahtevam in količinam strankinega naročila.

Odprema blaga - blago, namenjeno določeni stranki, se pregleda, če je potrebno zapakira ali prepakira, naloži na transportno vozilo in posodobi podatke o zalogi.



Slika 2.1: Operacije v skladišču (povzeto po [2])

Pretovarjanje blaga - blago je lahko tudi odposlani takoj, ko je prejeto. V tem primeru se ga raztovori, če je potrebno razdre in prepakira v manjše skupine, ter takoj odpošlje stranki. Skladiščenje v tem primeru ni potrebno.

Na sliki 2.1 so prikazane operacije v skladišču. Puščice med operacijami prikazujejo premik blaga znotraj skladišča.

Shranjevanje blaga razdelimo na začasno shranjevanje in pol-trajno shranjevanje [2]. Začasno shranjevanje je del funkcije premikanja blaga. Primer je začasno shranjevanje blaga v sprejemni coni pred skladiščenjem ali pretovarjanjem. Pri pretovarjanju se uporablja le začasno shranjevanje. Pol-trajno shranjevanje predstavlja shranjevanje presežka blaga kot zalogo skladišča. Blago iz zaloge se uporabi za izpolnjevanje naročil.

Prenos informacij poteka sočasno s premikanjem in shranjevanjem blaga. Za učinkovito vodenje skladišča so namreč potrebne natančne informacije o stanju skladišča. Zbiranje informacij in spremljanje stanja je naloga informacijskih sistemov za upravljanje skladišč s pomočjo različnih tehnologij (na primer s čitalniki črtne kode).

2.2 Tipi skladišč glede na način pobiranja naročil

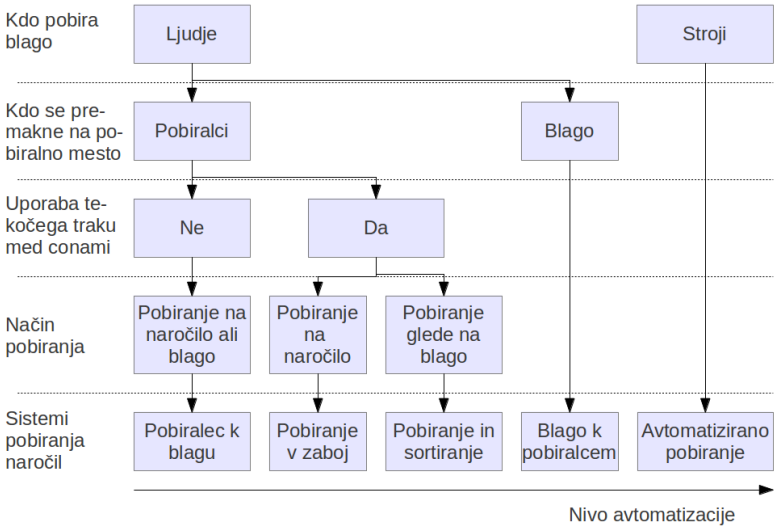
Skladišča uporabljajo različne načine pobiranja blaga in različne nivoje avtomatizacije. Zato je v [4] bila predlagana razdelitev skladišč po naslednjih kriterijih:

Kdo pobira blago Blago pobirajo ljudje ali stroji

Kdo se premakne na pobiralno mesto Do blaga odidejo ljudje ali pa ga stroj dostavi na posebno pobiralno mesto

Uporaba tekočega traku med conami Skladišča lahko uporabljajo tekoče trakove med posameznimi conami v skladišču

Način pobiranja Pobira se na naročilo ali več naročil oziroma manjše število različnih vrst blaga



Slika 2.2: Sistemi pobiranja naročil (povzeto po [4])

Na sliki 2.2 vidimo, da glede na zgoraj opisane kriterije, razdelimo sisteme pobiranja naročil na pet tipov, ki si sledijo od najmanj do najbolj avtomatiziranih:

Pobiralec k blagu Predstavlja najpogosteje uporabljen sistem, kjer se pobiralci premikajo med regali in pobirajo blago glede na naročilo ali skupino naročil. Če izpolnjujejo več naročil hkrati, blago takoj razvrstijo.

Pobiranje v zaboj Skladišče je razdeljeno na več con, ki so med seboj povezane s tekočim trakom, po katerem se premikajo zaboji z naročili. Znotraj ene cone pobira blago majhno število pobiralcev, ki jih dodajajo v zaboje. Ker en zaboj pripada enemu naročilu, končno razvrščanje ni potrebno. Primerno za skladišča, ki operirajo z majhnim blagom in naročili z manjšo količino blaga.

Pobiranje in sortiranje Pobiralci v pobiralni coni prenesejo eno vrsto blaga v količini, ki ustreza večjemu številu naročil, na tekoči trak. Tekoči trak zbira blago iz več pobiralnih con. Avtomatiziran sistem nato blago loči in ga pošlje na različne ponore, ki ustrezajo različnim naročilom. Ta sistem običajno deluje v valovih naročil, kjer je prejšnji val popolnoma razvrščen, preden se odpošlje naslednji val. Posledično je tudi število hkratnih naročil visoko - vsaj 20 naročil na val.

Blago k delavcem Avtomatiziran sistem dostavlja blago iz shranjevalnega dela v pobiralno cono. Pobiralec izbere želeno količino blaga, sistem pa preostalo blago ponovno shrani. Primer takih sistemov so sistemi AS/RS (angl. *Automatic storage and retrieval systems*).

Avtomatizirano pobiranje Popolnoma avtomatiziran sistem pobiranja, kjer ni več človeških pobiralcev.

Sistemi pobiralec k blagu so najbolj pogosti, po nekaterih ocenah predstavljajo 80% vseh skladišč v zahodni Evropi [1]. Pobiralci naj bi v teh sistemih

porabili tudi do polovico svojega časa za premike po skladišču do želenega blaga [1]. Ker gre za najmanj avtomatiziran in optimiziran proces, so z optimizacijo pobiranja, premikov v skladišču in shranjevanje zaloge možne velike pridobitve v učinkovitosti pobiralcev v skladišču.

2.3 Informacijski sistemi za upravljanje skladišča

Za informatizacijo skladišč so razviti številni sistemi, ki jih običajno imenujemo sistemi za upravljanje skladišča ali WMS (angl. *warehouse management systems*). Namen teh sistemov je nadzor in upravljanje pretoka blaga v skladišču, torej upravljanje zaloge, nadzor in planiranje dobave ter odpreme blaga.

Sisteme WMS sestavljajo različni moduli, ki implementirajo specifično funkcionalnost znotraj sistema [3]. Nekatere izmed modulov bomo predstavili v naslednjih odstavkih.

Sistemi za upravljanje s trgovskim blagom (angl. *Merchandise management systems* - MMS): Sistemi za pregled nad zalogo in pretoki blaga, kot jih uporabljajo v marketingu in knjigovodstvu. Operirajo z vrednostjo zaloge in podatki strank.

Vodstveni informacijski sistemi (angl. *Management information system* - MIS): Sistemi, ki procesirajo in združujejo informacije potrebne za vodstvene odločitve. Običajno so del sistemov MMS.

Sistemi planiranja in nadzora proizvodnje (angl. *Production planning and control* - PPC): Proizvodna podjetja uporabljajo sisteme PPC za načrtovanje porabe virov podjetja in optimizacijo zaloge glede na naročila strank.

Sistemi za upravljanje virov podjetja (angl. *Enterprise resource planning* - ERP): Sistemi za globalno upravljanje virov podjetja. V podjetjih, ki imajo več proizvodnih obratov s sistemi PPC, so sistemi ERP uporabljeni za globalen pregled in nadzor.

Krmilnik pretoka materialov (angl. *Material flow controller* - MFC): Delno ali v celoti avtomatiziran pretok materialov upravlja krmilnik MFC. Nadzoruje avtomatizirane dele skladišča.

Sistem za nadzor skladišča (angl. *Warehouse control system* - WCS): Tako kot sistem MFC skrbi za pretok materialov v skladišču. Običajno vsebuje še določene funkcionalnosti sistemov WMS, kot so pregled in nadzor nad zalogo.

Poglavje 3

Informacijski sistem za planiranje operacij v skladišču

Naša rešitev predstavlja pomožni informacijski sistem sistemom WMS za uporabo v skladiščih, kjer se pobiranje opravlja po sistemu pobiralec k blagu. Sistem implementira ogrodje, znotraj katerega se lahko izvedejo poljubni planerji, ki optimizirajo pobiranje blaga v skladišču glede na naročila.

3.1 Specifikacija rešitve

Informacijski sistem za planiranje operacij v skladišču izpolnjuje naslednje zahteve:

- Modelira skladišče. Model skladišča je sestavljen iz topologije skladišča, blaga v skladišču in pobiralcev. Topologija skladišča vključuje ceste in križišča v skladišču, regale s policami in prevzemno-odpremne prostore.
- Modelira proces pobiranja blaga po naročilih strank.
- Omogoča nalaganje in shranjevanje modela preko podatkovne baze.
- Definira vmesnik API za pisanje poljubnih planerjev, ki generirajo plane za pobiranje blaga glede na podana naročila.

- Sistem je sposoben obnoviti svoje stanje v primeru sesutja.
- Definira vmesnik API, ki omogoča interakcijo s sistemom.
- Vsebuje uporabniški vmesnik za potrebe vizualizacije izvedbe generiranih planov.

3.2 Razvojno okolje in uporabljene tehnologije

Za razvoj naše rešitve smo uporabili splošno-namenski programski jezik *Java 7 SE*. S številnimi dodatnimi knjižnicami in z možnostjo izvajanja aplikacije na katerikoli platformi, za katero je implementiran *JVM*, omogoča hiter razvoj z neodvisnostjo od operacijskega sistema, na katerem se aplikacija razvija in poganja. *JDK* (angl. *Java Development Kit*) je na voljo na spletni strani www.oracle.com/technetwork/java/javase/.

Za programski dostop do podatkovne baze smo uporabili javansko ogrodje *JPA* (angl. *Java Persistence API*). Ogrodje definira vmesnike in metode, preko katerih je mogoč dostop in upravljanje s podatki v relacijskih podatkovnih bazah. *JPA* omogoča uporabo in manipulacijo s podatki preko javanskih razredov. Ponudniki implementacije implementirajo pretvorbo med podatki v podatkovni bazi in javanskimi razredi. Za svoje delovanje potrebujejo še implementacijo gonilnika *JDBC* (angl. *Java Database Connectivity*), ki omogoča povezavo s podatkovno bazo in izvedbo poizvedb *SQL*. Kot ponudnika *JPA* smo uporabili *EclipseLink* (www.eclipse.org/eclipselink/), kot gonilnik *JDBC* pa gonilnik *jTDS* (jtds.sourceforge.net/), ki omogoča povezavo do Microsoftove podatkovne baze *MS SQL*. Konfiguracija in uporaba ogrodja *JPA* je opisana v knjigi [5].

Kot razvojno kolje smo uporabili *Eclipse IDE for Java EE Developers*, ki z enostavno konfiguracijo različnih tipov projektov in omogoča razširitev funkcionalnosti z različnimi vtičniki. Na voljo je na spletni stran www.eclipse.org/.

Znotraj razvojnega okolja smo namestili tudi vtičnik za integracijo z upravljalnikom projektov *Maven*. *Maven* omogoča upravljanje pomožnih knjižnic projekta, samodejno testiranje in izgradnja arhivov *JAR* (angl. *Java archive*). Datoteka s parametri je *pom.xml* in se nahaja v korenskem imeniku projekta. Uporaba upravljalnika projektov *Maven* je opisana v spletni publikaciji, dostopni na spletnem naslovu www.sonatype.com/books/mvnref-book/reference/.

Za optimizacijo izvajanja sistema in pregled nad porabo delovnega pomnilnika smo uporabili orodje *VisualWM*, ki je del JDK podjetja Oracle. Omogoča vizualizacijo porabe centralne procesne enote in delovnega pomnilnika, analizo poteka izvajanja aplikacije in vročih točk (najpogosteje izvedenih delov aplikacije) ter pregled nad uporabljeno kopico in objekti, ki se nahajajo na njej. Za zagon iz okolja Eclipse IDE smo namestili zaganjalnik, ki je na voljo na spletni strani visualvm.java.net/eclipse-launcher.html.

Poglavje 4

Pregled strukture aplikacije

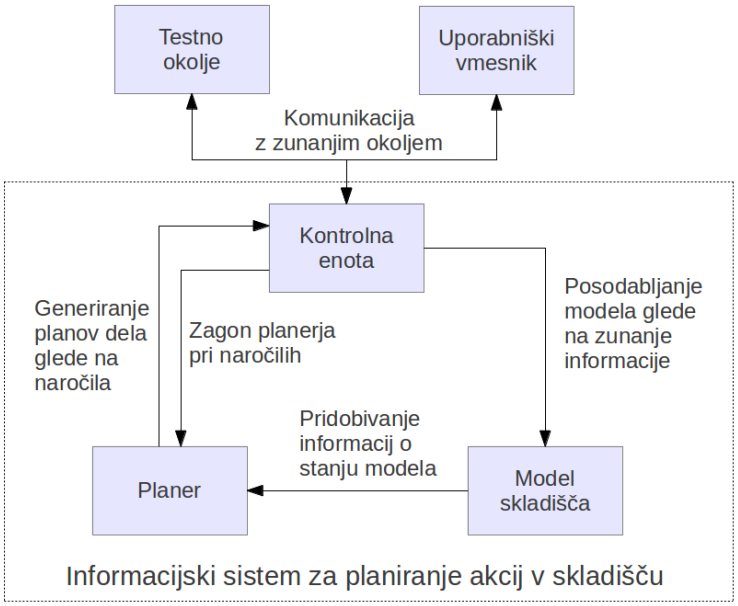
Aplikacija je razdeljena na pet programskih sklopov: model skladišča, programski vmesnik za pisanje in poganjanje planerjev, kontrolno enoto, uporabniški vmesnik in testno okolje. Model skladišča, programski vmesnik za pisanje planerjev in kontrolna enota tvorijo informacijski sistem za planiranje operacij v skladišču.

Model skladišča implementira trenutno stanje skladišča. Definira topologijo skladišča, ki vključuje ceste in križišča v skladišču, regale in police na regalih, dostop do regalov in vstopne točke v skladišče. Poleg topologije je model zadolžen za hranjenje stanja zaloge, lokacije blaga in vrsto ter pozicijo pobiralcev v skladišču.

Programski vmesnik za pisanje in poganjanje planerjev definira ogrodje za pisanje in poganjanje poljubnih planerjev. Implementira splošne razrede, ki jih uporabljajo planerji pri planiranju.

Kontrolna enota skrbi za komunikacijo z zunanjim okoljem. Med njenimi nalogami so sprejetje naročil, zagon planerjev, preverjanje pravilnosti generiranih planov, vračanje planov in spreminjanje stanja modela glede na izvršene akcije v skladišču.

Uporabniški vmesnik komunicira s kontrolno enoto in razredom, ki implementira grafični pogled skladišča. Omogoča izris topologije skladišča, naključno polnjenje skladišča, ustvarjanje naročil, ki so oddana kontrolni



Slika 4.1: Shematski prikaz sklopov aplikacije in tok informacij

enoti v procesiranje, in prikaz dela pobiralcev glede na generirane plane.

Testno okolje je namenjeno testiranju in primerjanju planerjev, ki implementirajo različne algoritme za optimizacijo pobiranja blaga. Omogoča avtomatizirano poganjanje izvajanja aplikacije glede na vnaprej definirane parametre. Komunicira s kontrolno enoto.

Na sliki 4.1 lahko prikažemo tipičen pretok informacij v aplikaciji. Kontrolna enota iz zunanjega okolja prejme seznam naročil. Naročila skupaj z pobiralci, ki so na voljo, odpošlje planerju, ki mora izdelati plan dela. Planer iz modela skladišča pridobi informacije o stanju skladišča in zaloge, ter za vsakega izmed pobiralcev izdela plan dela. Izdelane plane prejme kontrolna enota, ki preveri njihovo pravilnost, kar vključuje preverjanje izpolnjenosti naročil in smiselnost akcij. Preverjene plane vrne v zunanje okolje. Zunanje okolje potrdi akcijo plana, ki jo je opravil pobiralec (v dejanskem skladišču ali v simulatorju). V tem primeru kontrolna enota označi akcijo za izvršeno in ustrezno spremeni stanje modela skladišča.

Poglavje 5

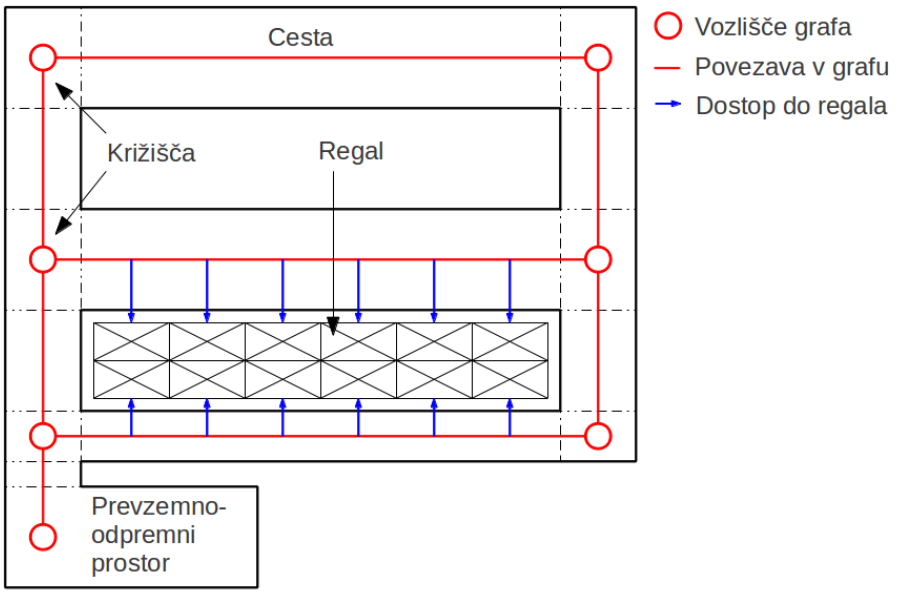
Modeliranje skladišča

V naši rešitvi je model skladišča instanca razreda *WarehouseModel* v paketu *lui.warehouseplanning.core.model* skupaj s pomožnimi razredi iz tega paketa. Ker kompleksnejši planerji potrebujejo hiter dostop do podatkov modela, smo se odločili, da bo naša implementacija modela v celoti vsebovana v delovnem polnilniku računalnika in se ne bo zanašala na dostop do podatkovne baze ali drugega sistema. Vse podatke potrebne za predstavitev modela skladišča se pridobi ob inicializaciji modela.

5.1 Implementacija topologije

Topologija skladišča vsebuje ceste, križišča, prevzemne in odpremne prostore, regale ter police na regalih. V literaturi zasledimo implementacijo topologije s pomočjo razdelitve skladišča na mrežo, kjer je vsaka celica cesta, križišče, regal ali prazen blok (primer takšne topologije je opisan v članku [6]). V naši rešitvi smo se odločili za implementacijo z grafom, kjer vsako križišče predstavlja vozlišče v grafu in cesta povezavo med dvema vozliščema. Za takšno implementacijo smo se odločili, ker:

- je mogoče z grafom natančneje opisati dimenzije križišč in cest kot z celicami mreže, kjer je natančnost omejena z velikostjo celice



Slika 5.1: Transformacija topologije skladišča v graf

- je implementacija iskanja najkrajše poti, potrebna za učinkovito usmerjanje pobiralcev, definirana kar z iskanjem najkrajše poti v grafu
- je mogoče natančneje določiti dostopno točko, iz katere je dostopen regal
- je z natančnimi dimenzijami enostavno implementirati vizualizacijo skladišča

Potrebni podatki za modeliranje topologije skladišča so širina in dolžina cest, križišč, prevzemnih in odpremnih prostorov, ter povezave med njimi. Za modeliranje regalov so potrebni podatki poleg dolžine in širine še višina, število polic in višina vsake police ter cesta ob kateri se regal nahaja.

Na sliki 5.1 je prikazan primer skladišča in pretvorba topologije v graf. Naše predpostavke glede topologije skladišča je, da so skladišča pravokotna. V takih skladiščih so ceste lahko usmerjene vzdolž abscisne ali ordinatne osi, regali pa se nahajajo neposredno ob cestah. Zaradi poenostavitve problema

smo tudi predpostavili, da pobiralci potujejo po sredini cest in ne izvajajo mehkih zavojev. Topologijo skladišča pretvorimo v graf, ki je na sliki 5.1 prikazan z rdečo barvo. Križišča postanejo vozlišča grafa brez dimenzije, ceste pa povezave v grafu z dolžino. Dolžina povezav se izračuna po enačbi (5.1), kjer je c cesta, vz in vk pa pripadajoči križišči.

$$d(c, vz, vk) = \begin{cases} c_x + vz_x/2 + vk_x/2, & c \text{ poteka vzdolž abscisne osi} \\ c_y + vz_y/2 + vk_y/2, & c \text{ poteka vzdolž ordinatne osi} \end{cases} \tag{5.1}$$

Prevzemni in odpremni prostori v skladišču so opisani kot posebna vozlišča, ki imajo običajno le eno pripadajočo povezavo. V realnih skladiščih predstavljajo nakladalne rampe oziroma predprostore, kjer se zbira blago za odpremo ali skladiščenje. V modelu lahko tem prostorom omejimo maksimalno količino blaga, ki se tam lahko nahaja.

Z grafom kot osnovo topologije definiramo tudi topološko pozicijo pobiralcev in objektov v skladišču. Pobiralec se v skladišču lahko nahaja na križišču ali na cesti. Na križišču se nahaja natanko tedaj, ko je točno v centru križišča. V tem primeru je njegova pozicija natanko določena z ID-jem vozlišča, ki predstavlja križišče. V nasprotnem primeru se nahaja na eni izmed cest. Čeprav ceste niso enosmerne, povezave, ki jih predstavljajo, vsebujejo informacijo o začetnem in končnem vozlišču, s katero definiramo usmerjenost. Pozicijo pobiralca zato definiramo kot ID povezave, ki predstavlja cesto, na kateri se nahaja, skupaj z odmikom od začetnega vozlišča. Topološka pozicija je definirana v paketu *lui.warehouseplanning.core.position*.

Regali vsebujejo police, na katerih se nahaja blago. Vsaka polica ima definirano višino od tal. Točko dostopa do regalov smo definirali na enak način, kot smo definirali pozicijo pobiralcev. Vsak regal ima definirano topološko pozicijo, ki predstavlja točko na povezavi, iz katere lahko pobiralec dostopa do regala, in informacijo o tem, na kateri strani povezave se nahaja. Možne vrednosti so *LEVO* ali *DESNO* glede na usmerjenost povezave. Na sliki 5.1 so z modrimi puščicami prikazane točke na cesti, od koder lahko pobiralci dostopajo do regalov.

5.1.1 Iskanje najkrajše poti v skladišču

Za optimizacijo premikov pobiralcev so potrebne natančne informacije o dolžinah poti med posameznimi topološkimi pozicijami. Ker za osnovo topologije uporabljamo graf, smo za iskanje najkrajše poti med dvema vozliščema uporabili *Dijkstrov algoritem* za iskanje najkrajših poti v grafu. Iskanje najkrajše poti med dvema poljubnima topološkima pozicijama pa je opisano s psevdokodo 5.2. Ker skladišča vsebujejo veliko več regalov, kot vsebujejo križišč, lahko na ta način dovolj učinkovito izračunamo najkrajše poti znotraj skladišča. Da se izognemo ponovnemu računanju poti, shranjujemo tako poti, ki jih je našel *Dijkstrov algoritem*, kot tudi poti, ki smo jih našli z algoritmom 5.2. Podobna rešitev iskanja najkrajše poti je opisana v članku [6].

5.1.2 Pretvorba topologije v kartezične koordinate

Za potrebe vizualizacije skladišča smo morali pretvoriti topologijo skladišča, ki je implementirana z grafom, v kartezične koordinate. S to pretvorbo smo dosegli enostaven izris topologije v grafičnem vmesniku. Pretvorba poteka v dveh korakih:

1. Izberemo začetno vozlišče, in ga označimo kot izhodišče s koordinatama $(0, 0)$. Nato z iskanjem v širino obiščemo vsa preostala vozlišča preko povezav. Za vsako vozlišče se sprehodimo po povezavah iz vozlišča in izračunamo koordinate povezav in koordinate regalov ob povezavah. Hranimo tudi minimalni koordinati, ki smo ju izračunali pri iskanju v širino.

2. Če sta minimalni koordinati različni od 0, je potrebno izvesti premik elementov za razliko. Spet začnemo v poljubnem vozlišču in z iskanjem v širino obiščemo vse elemente. Za vsak element izvedemo translacijo za minimalni koordinati, izračunani v prejšnjem koraku. S tem korakom so vse koordinate v prvem kvadrantu koordinatnega sistema.

Data: Topološki poziciji t_1 in t_2

Result: Najkrajša pot med t_1 in t_2

if t_1 *pripada vozlišču* $\wedge$ t_2 *pripada vozlišču* **then**
  return $DijkstrovAlg(t_1, t_2)$
else if t_1 *pripada vozlišču* $\wedge$ t_2 *pripada povezavi* **then**
  $v_1 \longleftarrow$ začetno vozlišče t_2
  $v_2 \longleftarrow$ končno vozlišče t_2
  $pot_1 \longleftarrow DijkstrovAlg(t_1, v_1) + PotMed(v_1, t_2)$
  $pot_2 \longleftarrow DijkstrovAlg(t_1, v_2) + PotMed(v_2, t_2)$
  return $NajkrajsaPot(pot_1, pot_2)$
else if t_1 *pripada povezavi* $\wedge$ t_2 *pripada vozlišču* **then**
  $v_1 \longleftarrow$ začetno vozlišče t_1
  $v_2 \longleftarrow$ končno vozlišče t_1
  $pot_1 \longleftarrow PotMed(t_1, v_1) + DijkstrovAlg(v_1, t_2)$
  $pot_2 \longleftarrow PotMed(t_1, v_2) + DijkstrovAlg(v_2, t_2)$
  return $NajkrajsaPot(pot_1, pot_2)$
else
  $v_1 \longleftarrow$ začetno vozlišče t_1
  $v_2 \longleftarrow$ končno vozlišče t_1
  $v_3 \longleftarrow$ začetno vozlišče t_2
  $v_4 \longleftarrow$ končno vozlišče t_2
  $pot_1 \longleftarrow PotMed(t_1, v_1) + DijkstrovAlg(v_1, v_3) + PotMed(v_3, t_2)$
  $pot_2 \longleftarrow PotMed(t_1, v_2) + DijkstrovAlg(v_2, v_3) + PotMed(v_3, t_2)$
  $pot_3 \longleftarrow PotMed(t_1, v_1) + DijkstrovAlg(v_1, v_4) + PotMed(v_4, t_2)$
  $pot_4 \longleftarrow PotMed(t_1, v_2) + DijkstrovAlg(v_2, v_4) + PotMed(v_4, t_2)$
  return $NajkrajsaPot(pot_1, pot_2, pot_3, pot_4)$
end

Slika 5.2: Psevdokoda algoritma za iskanje najkrajše poti

Data: Graf topologije G

Result: Elementi vsebujejo koordinato spodnjega levega oglišča

$vrsta \leftarrow$ prazna vrsta
$minimum \leftarrow (0, 0)$
$V \leftarrow$ vozlišče iz grafa G
koordinate $V \leftarrow minimum$
V obiskan
V dodamo v vrsto
while *vrsta ni prazna* **do**
    $V_z \leftarrow$ prvi element iz vrste
    for *povezava* E *iz* V_z **do**
        koordinate $E \leftarrow$ izračunaj koordinate glede na V_z
        $minimum \leftarrow MinKoordinate(minimum,$ koordinate $E)$
        for *regal* R *na povezavi* E **do**
            koordinate $R \leftarrow$ izračunaj koordinate glede na E
            $minimum \leftarrow MinKoordinata(minimum,$ koordinate $R)$
        end
        $V_k \leftarrow$ končno vozlišče povezave E
        if V_k *ni obiskan* **then**
            koordinate $V_k \leftarrow$ izračunaj koordinate glede na E
            $minimum \leftarrow MinKoordinata(minimum,$ koordinate $V_k)$
            V_k obiskan
            V_k dodamo v vrsto
        end
    end
end

Slika 5.3: Psevdokoda algoritma za izračun kartezičnih koordinat elementov skladišča

Prvi korak algoritma je opisan s psevdokodo 5.3. Grafični vmesnik uporabi izračunano velikost skladišča oziroma omejujoči pravokotnik za izvedbo skaliranja koordinat elementov, ki jih potem izriše na zaslon.

5.2 Modeliranje blaga

Opis blaga v modelu je del abstraktnega razreda *AbstractItem*, ki je skeletna implementacija blaga z naslednjimi lastnostmi:

- ime oziroma ID blaga
- lokacija, na kateri se blago nahaja (lahko je polica, pobiralec ali prevzemni/odpremni prostor)
- datum, ko je blago prišlo v skladišče
- datum, ko blagu poteče rok uporabe (če blago nima roka uporabe je to 1.1.2200)
- količina blaga, ki jo ta objekt predstavlja
- težo na enoto blaga

En objekt lahko predstavlja večjo količino blaga iste vrste z enakimi lastnostmi, če se nahaja na isti lokaciji. Na ta način varčujemo z delovnim pomnilnikom in poenostavimo opis zaloge. Planerji lahko zahtevajo objekt s specifično količino, ki bo nastal z razdelitvijo že obstoječega objekta na dva dela. Združitev objektov preprečimo z zaklepanjem.

Skladišča lahko vsebujejo različne tipe pakiranja blaga. V naši rešitvi je trenutno implementiran le tip *SIMPLE_ITEM*, ki razširja abstraktni razred *AbstractItem* in ne vsebuje nobene dodatne lastnosti. Predstavlja najosnovnejšo nedeljivo vrsto blaga.

Abstraktni razred *AbstractItem* definira tudi globalni mehanizem zaklepanja. Model uporablja metode zaklepanja za označitev blaga, ki je del predvidenega plana dela. Zaklenjenega blaga ni mogoče uporabiti pri planiranju.

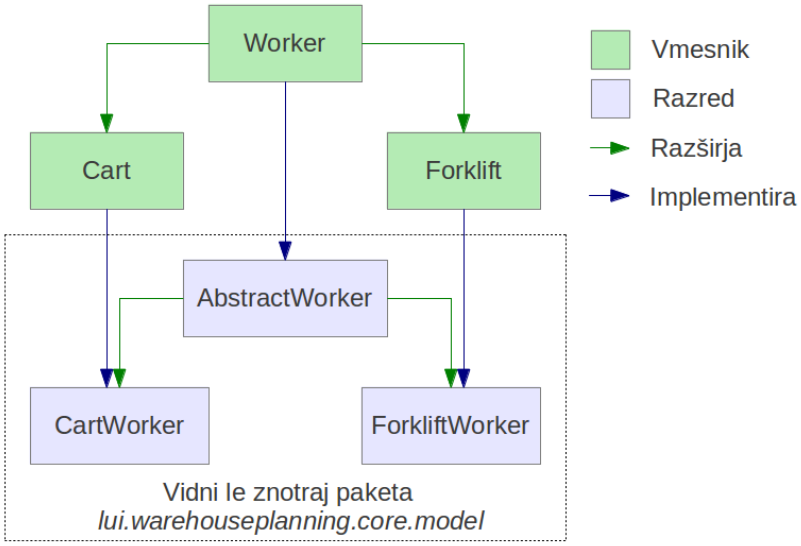
5.3 Modeliranje pobiralcev

Na sliki 5.4 je predstavljena hierarhija vmesnikov in razredov, ki predstavljajo pobiralce v modelu. Vmesnik *Worker* je glavni vmesnik, ki definira splošne lastnosti in metode pobiralcev v modelu:

- ID in tip pobiralca
- trenutna topološka pozicija
- hitrost premikanja
- metode za dodajanje in odvzemanje blaga ter pregled blaga, ki ga trenutno prevaža pobiralec
- praznost ali polnost pobiralca (pobiralec je poln, ko ne more več pobrati nobenega blaga)
- število predelov, ki jih vsebuje pobiralna naprava
- metode za tekstovni izpis informacij o pobiralcu

Vmesnik *Worker* razširjata dva vmesnika, ki predstavljata različne tipe pobiralcev: *Cart* in *Forklift*. Vmesnik *Cart* predstavlja pobiralce, ki pri svojem delu uporabljajo voziček. Voziček ima lahko več predelov, zato lahko pobiralci pobirajo za več naročil hkrati. Poleg števila predelov pa vmesnik določa še najvišjo višino, ki jo pobiralec še doseže. Do višjih polic na regalih se dostopa z viličarji, ki jih definira vmesnik *Forklift*. Viličarji lahko delajo le na enem naročilu hkrati. Vmesnik vsebuje še metode za dviganje in spuščanje vilic, ter parametre o hitrosti dviganja in spuščanja le teh.

Abstraktni razred *AbstractWorker* je skeletna implementacija vmesnika *Worker*. Razširjata ga razreda *CartWorker* in *ForkliftWorker*, ki vsak implementira svoj vmesnik kot je prikazano na sliki 5.4. Ti razredi niso dostopni izven paketa *lui.warehouseplanning.core.model*, ker vsebujejo metode, ki spreminjajo stanje modela in so rezervirane izključno za uporabo znotraj modela skladišča.



Slika 5.4: Hierarhija vmesnikov in razredov, ki predstavljajo pobiralce

5.4 Shranjevanje in nalaganje modela

V paketu *lui.warehouseplanning.core.persistence* sta implementirana dva razreda za shranjevanje oziroma nalaganje modela: *FileIO*, ki bere in piše tekstovne datoteke, in *DatabaseBridge*, ki omogoča branje in shranjevanje modela v podatkovno bazo. Preostali razredi znotraj paketa definirajo prehodne strukture, ki skrbijo za komunikacijo med modelom in nalagalniki oziroma shranjevalniki modela. Nalagalniki modela proizvedejo razrede iz tega paketa, ki jih model skladišča uporabi za svojo inicializacijo, shranjevalniki pa uporabijo te razrede, ki jih proizvede model, za shranjevanje modela. S to implementacijo smo omogočili enostavno dodajanje novih nalagalnikov ali shranjevalnikov z uporabo programskega vmesnika modela.

5.4.1 Shranjevanje in nalaganje iz datotek

Razred *FileIO* implementira nalaganje in shranjevanje modela skladišča iz tekstovnih datotek. Uporablja se ga predvsem za testiranje sistema in pla-

nerjev. Za ta namen uporablja tri datoteke:

- datoteka s končnico *.model*, ki vsebuje opis topologije skladišča
- datoteka s končnico *.items*, ki vsebuje opis zaloge v skladišču
- datoteka s končnico *.paths*, ki vsebuje najkrajše poti v skladišču

Za inicializacijo modela je potrebno podati le datoteko s končnico *.model*. Preostali dve datoteki sta neobvezni in bosta uporabljeni, če se nahajata v istem imeniku kot datoteka modela in imata isto ime (končnice so seveda različne).

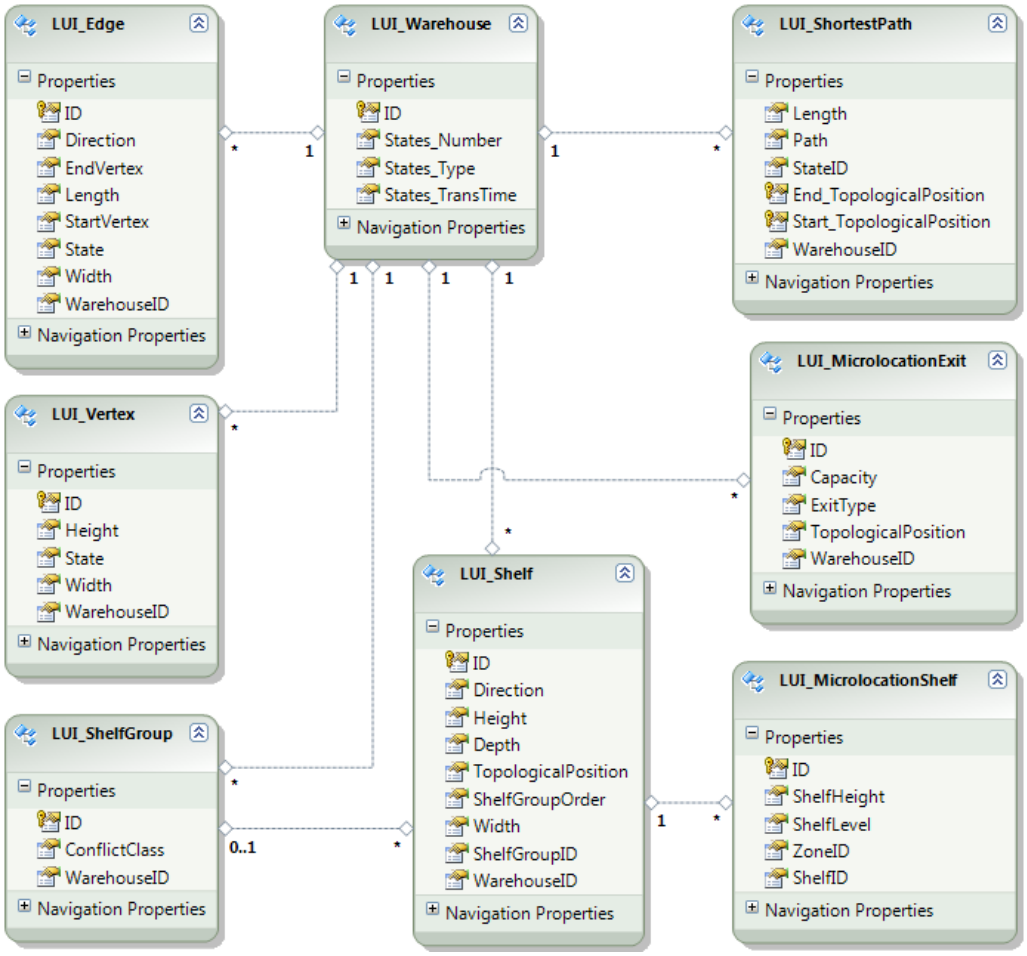
Formati vseh treh datotek so opisani v dodatku D.

5.4.2 Shranjevanje in nalaganje iz podatkovne baze

Ker rešitev *Hydra@Warehouse* uporablja podatkovno bazo za shranjevanje in upravljanje s podatki, smo razredu *DatabaseBridge* implementirali tudi nalagalnik in shranjevalnik modela v podatkovno bazo. Za dostop smo uporabili tehnologijo JPA, opisano v Poglavju 3.2. Entitetni model, ki opisuje entitete potrebne za nalaganje in shranjevanje modela, je prikazan na sliki 5.5.

Entiteta *LUI_Warehouse* vsebuje splošne podatke o skladišču, kot je na primer ID skladišča in ustreza razredu *PTWarehouse*. Skladišče ima lahko več regalov predstavljenih z entiteto *LUI_Shelf* in vsak regal ima lahko več polic predstavljenih z entiteto *LUI_MicrolocationShelf*. Ustrezni implementaciji v Javi sta razreda *PTRack* in *PTRackMicrolocation*. Regale lahko tudi združujemo v skupino, kar je opisano z entiteto *LUI_ShelfGroup* oziroma razredom *PTRackGroup*. Ceste in križišča predstavljajo entitete *LUI_Edge* in *LUI_Vertex* oziroma razredi *PTEdge* ter *PTVertex*. Najkrajše poti opisuje entiteta *LUI_ShortestPath* oziroma razred *PTShortestPath*.

Entitetni model ne vsebuje entitete za blago znotraj skladišča. Ker naša rešitev predstavlja dopolnilni sistem rešitvi *Hydra@Warehouse*, smo se odločili, da ne bomo podvajali podatkov v podatkovni bazi. Naša rešitev se



Slika 5.5: Entitetni model modela skladišča

zanaša na podatke o zalogi, ki jih sprejme iz glavnega sistema in jih pretvori v instance razreda *PItem*, s katerimi napolni model skladišča.

Za inicializacijo modela iz podatkovne baze potrebujemo URL do podatkovne baze, ime gonilnika s katerim se bomo priklopili na podatkovno bazo, uporabniško ime in geslo ter ID skladišča, ki ga želimo naložiti. Javanski arhiv z gonilnikom JDBC se mora nahajati v istem imeniku kot aplikacija ali pa mora biti podan kot parameter pri zagonu.

5.5 Programski vmesnik za manipuliranje modela

Programski vmesnik za manipulacijo modela je razdeljen na vmesnik *QueryModel* in razred *WarehouseModel*. Vmesnik definira metode, ki so namenjene poizvedovanju o trenutnem stanju in ne spreminjajo stanja modela. Razdelimo jih lahko na pet funkcionalnih sklopov:

- metode za poizvedovanje o topologiji modela, kot je metoda za iskanje najkrajše poti (*getShortestPath(...)*) ali metoda, ki vrne vse prevzemne/odpremne prostore (*getExitMicrolocation(...)*)
- metodi, ki vračata pobiralce registrirane v modelu (*getAllWorkerIDs()* in *getWorker(...)*)
- metode, katerih ime se konča z **item* ali **items*, in iz indeksov v modelu vračajo blago glede na podane parametre (na primer metodi *items(...)* in *getBestItem(...)*)
- metoda, ki omogoča razbijanje objektov, ki predstavljajo večjo količino blaga, če želimo operirati z manjšo količino (metoda *split(...)*)
- metode, katerih ime se začne z *virtual** in so namenjene planerjem za uporabo pri planiranju

Planerji lahko do modela skladišča dostopajo le preko metod definiranih v vmesniku *QueryModel*. Te metode ne dopuščajo spreminjanja modela, planerji pa morajo biti sposobni izvesti simulacijo planirane akcije in oceniti njen rezultat. Zato vmesnik vsebuje tudi metode, katerih ime se zančne na *virtual**. Z metodo *virtualNewVirtualCopy()* planer stvari kopijo skladišča, ki jo potem z preostalimi metodami manipulira. Ko konča z uporabo kopije jo uniči z metodo *destroyVirtualCopy(...)*.

Razred *WarehouseModel* implementira vmesnik *QueryModel* in definira drugi del programskega vmesnika. Ta razred vsebuje metode, ki spreminjajo

stanje modela. Te metode potrebuje za svoje delovanje kontrolna enota. Razdelimo jih lahko na več sklopov:

- metode, katerih ime se začne na *set**, *add** ali *create**, ki jih nalagalniki uporabljajo za inicializacijo modela
- metode, katerih ime se začne na *getPersistent**, ki jih shranjevalniki uporabljajo za shranjevanje modela
- metodo *fillWarehouse(...)*, ki naključno napolni skladišče in je uporabljena v simulatorjih
- metodo *removeItemsFromWarehouseFromExit(...)*, ki simbolizira odpremo blaga - odstrani blago, ki se nahajajo v odpremnih prostorih v modelu
- metodi *lockItem(...)* in *unlockItem(...)* za odklepanje in zaklepanje blaga
- metodo *execute(...)*, ki izvrši del plana in s tem spremeni stanje znotraj modela skladišča

5.6 Poraba delovnega pomnilnika

Z orodjem *VisualWM* smo preverili porabo delovnega pomnilnika pri modeliranju skladišča. V ta namen smo definirali tri modele z različnimi velikostmi in parametri, ki so opisani v tabeli 5.1.

	Št. križišč	Št. cest	Št. regalov	Št. polic
Majhno skladišče	50	83	384	1536
Srednje skladišče	87	148	720	2880
Veliko skladišče	137	238	1200	4800

Tabela 5.1: Lastnosti testnih skladišč

Za vsak model skladišča smo preverili porabo delovnega pomnilnika v petih različnih primerih:

1. porabo modela, ko ne vsebuje blaga
2. porabo modela, kjer ima vsaka polica 25 enot blaga in je v modelu 100 različnih vrst blaga
3. porabo modela, kjer ima vsaka polica 25 enot blaga in je v modelu 1000 različnih vrst blaga
4. porabo modela, kjer ima vsaka polica 50 enot blaga in je v modelu 100 različnih vrst blaga
5. porabo modela, kjer ima vsaka polica 50 enot blaga in je v modelu 1000 različnih vrst blaga

Blago v modelu skladišča je bili naključno ustvarjeno z metodo *fillWarehouse(...)* razreda *WarehouseModel*. Razporejeno je bilo popolnoma neurejeno, s čimer smo simulirali najslabši možni scenarij, kjer je potrebno veliko število objektov za opis zaloge v skladišču. Vse vrste blaga imajo enako distribucijo pojavnosti v modelu.

V tabeli 5.2 je prikazana poraba delovnega pomnilnika za modeliranje skladišča glede na zgoraj naštete parametre. Z velikostjo skladišča narašča tudi velikost uporabljenega pomnilnika, največji del pomnilnika pa je uporabljen za opis zaloge. Ker smo zalogo razporedili popolnoma naključno, se na polici nahaja večje število različnih vrst blaga. Zato model ne more opisati zaloge na polici le z enim objektom, kot v primeru, ko je na eni polici natanko ena vrsta blaga. V tabeli 5.2 zato vidimo, da velikost uporabljenega pomnilnika narašča tako s povečanjem števila enot blaga na polico, kot tudi z povečanjem števila različnih vrst blaga v modelu. Naraščanje je linearno glede na velikost zaloge.

Maksimalno porabo pomnilnika smo zabeležili za veliko skladišče, ki je vsebovalo 50 enot blaga na polico in 1000 različnih vrst blaga v skladišču. Po-

	1. Prazno	2. 25/100	3. 25/1000	4. 50/100	5. 50/1000
Majhno	0,85 MB	11,3 MB	13,0 MB	20,0 MB	24,5 MB
Srednje	1,59 MB	21,3 MB	24,0 MB	37,7 MB	45,6 MB
Veliko	2,65 MB	35,4 MB	40,5 MB	62,5 MB	77,6 MB

Tabela 5.2: Poraba pomnilnika pri različnih modelih in parametrih

raba je bila manj kot 80 MB, s čimer smo pokazali, da lahko dovolj učinkovito modeliramo tudi večja in kompleksnejša skladišča.

Poglavje 6

Programski vmesnik za pisanje planerjev

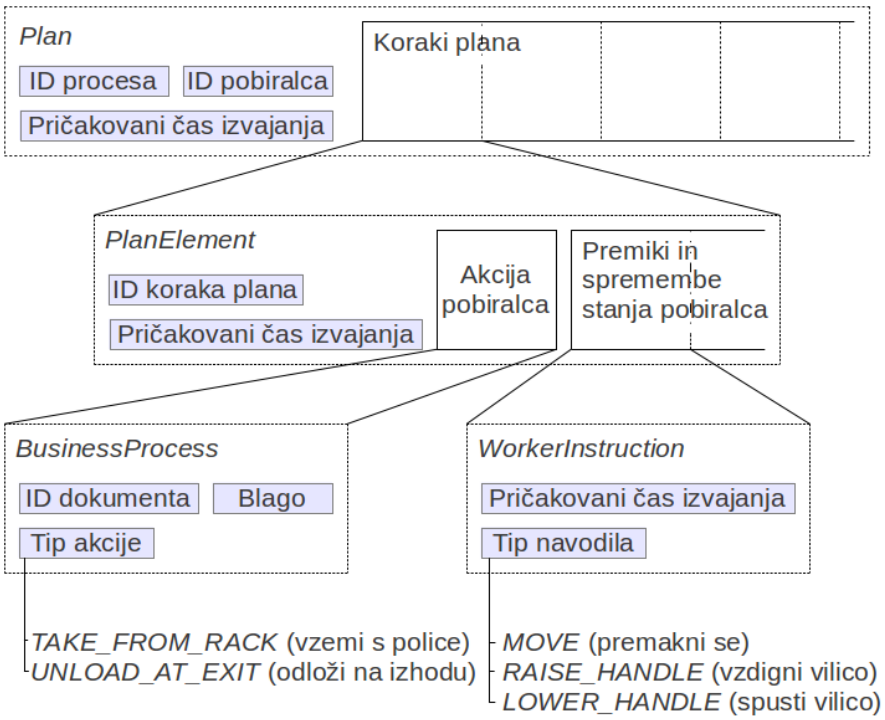
Naša rešitev implementira ogrodje za pisanje in poganjanje različnih planerjev. Planerji so razredi, ki razširjajo razred *Planner*, ki se nahaja v paketu *lui.warehouseplanning.planners*, in upoštevajo pravila implementacije planerjev. Pravila določajo način inicializacije planerjev, metode in funkcije, ki jih planer lahko uporablja, način sestavljanja planov ter registracijo planerja s sistemom. Namen ogrodja je implementirati okolje, ki razvijalcu omogoča dovolj svobode za učinkovito implementacijo poljubnih planerjev, hkrati pa skrije nepotrebne podrobnosti implementacije sistema in varuje model skladišča pred nezaželenimi spremembami, kot morebitnimi posledicami delovanja planerja.

Ogrodje za poganjanje planerjev omogoča tudi asinhrono vračanje delnih planov oziroma elementov plana. Vsak delen korak plana je lahko nemudoma poslan sistemu, ki obvesti vse registrirane poslušalce. Na ta način ni potrebno čakati na izdelavo celotnega plana, ampak se lahko delo začne takoj, ko je prvi korak znan.

Kot primer implementacije smo implementirali planer *SimplePlanner*, ki predstavlja požrešni planer. Planer vedno napoti pobiralca k najbližjemu blagu v skladišču, ki ga mora pobiralec pobrati glede na podana naročila.

6.1 Sestavljanje planov

Planerji generirajo plane, ki opisujejo potek dela za pobiralce. Kot rezultat vračajo seznam instanc razreda *Plan*, ki za vsakega delavca opisuje navodila za delo. Razčlenitev razreda *Plan* in njegovih pod-struktur je prikazana na sliki 6.1.



Slika 6.1: Struktura plana dela za enega delavca

Vsaka instanca razreda *Plan* vsebuje ID procesa, pod katerim se izvaja planer, ID pobiralca, za katerega je plan namenjen, pričakovani čas izvajanja celotnega plana in seznam posameznih korakov plana.

Korak plana je implementira v razredu *PlanElement*. Vsaka instanca razreda *PlanElement* vsebuje ID, ki je 64-bitno število in vsebuje informacijo o ID-ju procesa, ID-ju pobiralca in zaporednem številu koraka. Na ta način je mogoče sestaviti celotne plane, čeprav imamo na voljo le skupino instanc

razreda *PlanElement*. Instanca vsebuje še pričakovani čas izvajanja tega koraka, seznam navodil za premikanje ali spremembe stanja pobiralca, in končno akcijo, ki naj jo pobiralec izvede.

Seznam navodil pobiralcu za premik oziroma spremembo stanja opisuje navodila, ki jih mora izvesti pobiralec, preden lahko izvede akcijo na blagu v skladišču. Navodila definira vmesnik *WorkerInstruction*, ki je del paketa *lui.warehouseplanning.core.instructions*. Vsako navodilo mora vsebovati pričakovani čas izvajanja in tip navodila. Vsak posamezen tip navodila lahko vsebuje še druge atribute. Primer je navodilo za premik iz ene topološke pozicije na drugo, ki je implementirano v razredu *MoveInstruction* in kot atribut vsebuje pot, ki naj jo opravi pobiralec. Naša rešitev pozna še dva druga tipa navodil, ki sta namenjena pobiralcem z viličarji. To sta navodili za dvig in spust vilice, ki sta definirani v razredih *RaiseHandleInstruction* in *LowerHandleInstruction*.

Akcija pobiralca opisuje akcijo na blagu v skladišču potem, ko pobiralec izvede navodila premikanja. Definirana je z vmesnikom *BusinessProcess*, ki je del paketa *lui.warehouseplanning.core.bussinessprocess*. Vsaka akcija mora vsebovati ID naročila, na katerega se ta akcija nanaša in tip akcije. Večina akcij bo vsebovala tudi seznam blaga, nad katerimi naj se izvedejo. Posamezne akcije lahko vsebujejo še dodatne atribute. Naša rešitev pozna dve akciji - akcijo pobiranja blaga iz police, ki je implementirana v razredu *BPTakeFromRack* in kot atribut vsebuje še polico, iz katere naj se vzame blago, in akcijo odlaganja blaga na izhodu, implementirano v razredu *BPUnloadAtExit*, ki kot atribut vsebuje še odpremni prostor, na katerega naj se odloži blago.

Plani, ki jih generirajo planerji, morajo biti smiselni. Primer nesmiselnega plana bi bil plan, ki narekuje pobiranje blaga s police, ki ne vsebuje tega blaga, ali s police, zraven katere se pobiralec trenutno ne nahaja. Čeprav planerji lahko ustvarijo tudi take plane, bodo ti, po preverjanju pravilnosti v kontrolni enoti in modelu skladišča, zavrnjeni.

6.2 Funkcije ocene cene pobiranja blaga

Planerji lahko pri svojem delu uporabljajo tudi funkcije ocene cene pobiranja blaga. Te funkcije razširjajo abstraktni razred *ItemEvaluator* v paketu *lui.warehouseplanning.core.evaluators* in vsakemu objektu, ki predstavlja blago, priredijo pozitivno realno vrednost, ki oceni ceno akcije pobiranja tega blaga in omogoča rangiranje akcij. Manjša ocena predstavlja bolj zaželeno akcijo.

Naša rešitev že vsebuje tri splošne funkcije implementirane v naštevnem tipu *ItemEvaluators*:

- *DISTANCE* - ocena sestoji iz razdalje med referenčno topološko pozicijo in topološko pozicijo regala, na katerem se nahaja blago
- *DAYS_TILL_EXPIRATION* - ocena sestoji iz števila dni preden blagu poteče rok veljavnosti
- *COMBINED* - ocena sestoji iz linearne kombinacije ocen funkcij *DISTANCE* in *DAYS_TILL_EXPIRATION*, uteži so določene s parametri

Funkcija *getEvaluator(...)* vrne instanco podrazreda *ItemEvaluator* in sprejme kot argument model skladišča, referenčno topološko pozicijo, ki običajno predstavlja pozicijo delavca v skladišču, ter slovar uteži, ki vplivajo na obnašanje funkcije ocene.

Planer lahko funkcijo ocene uporabi kot argument metodam vmesnika modela, ki vračajo blago. Vrnjeno blago bo razvrščeni glede na podano funkcijo ocene. S funkcijama *getBestItem(...)* in *virtualGetBestItem(...)* lahko planer tudi pridobi le najbolje ocenjeno blago.

6.3 Abstraktni razred Planner

Abstraktni razred *Planner* vsebuje skeletno implementacijo planerja in definira vstopno točko v planer ter pogosto uporabljene metode pri planiranju. Vsak planer mora razširiti razred *Planner* in vsebovati konstruktor,

ki sprejme dva objekta: *QueryModel*, ki predstavlja programski vmesnik do modela skladišča, in vrsto *java.util.concurrent.BlockingQueue*, v katero se dodaja elemente plana. Inicializacija razreda se zanaša na obstoj konstruktorja s temi parametri in v primeru, da le ta ne obstaja, planer ne bo bil inicializiran.

Vsak planer se instancira samo enkrat. Zato ni priporočljivo shranjevati parametrov planiranja kot atribute planerja. V primeru, da se želi prenašati parametre med posameznimi izvajanji planiranja, je potrebno poskrbeti za ustrezno zaklepanje in sinhronizacijo podatkov.

Vsak planer mora implementirati metodo *getCapabilities()*, ki sporoča, za katere vrste dokumentov, ki opisujejo cilje, je planer sposoben ustvariti plane. Ker naša rešitev trenutno podpira le proces pobiranja naročil, planerji vračajo vrednost *PlannerCapabilities.ORDER*.

Vstopna točka v planer je metoda *getPlan(...)*. Inicializira funkcijo ocene cene pobiranja blaga, parametre te funkcije in začasni model skladišča, ki ga lahko planer uporablja pri planiranju. Nato se pokliče abstraktna metoda *getPlans(...)*, ki jo morajo implementirati vsi planerji. Njeni parametri so:

int *processID* ID procesa, pod katerim teče planer in se uporablja pri inicializaciji novih planov

Set<Integer> *workerIDs* množica ID-jev vseh pobiralcev, ki so na voljo pri planiranju

List<Document> *documents* seznam dokumentov, ki opisujejo naročila, ki jih je potrebno izpolniti

ItemEvaluators *evaluator* naštevni tip, ki definira funkcijo ocene cene pobiranja blaga

Map<String, Double> *evalProperties* slovar parametrov funkcije ocene cene pobiranja blaga, ki je lahko prazen

int *virtualHandle* ID začasnega modela, s katerim lahko planer kliče funkcije modela, ki se začnejo z *virtual**

int *planLength* skupna maksimalna dolžina elementov plana, ki jih lahko planer ustvari, če je enaka 0, potem ni omejitve

Map<String,String> *properties* slovar parametrov planerja, ki je lahko prazen

Metoda vrača seznam planov, kjer vsak plan pripada natanko enemu pobiralcu. V primeru, da pride do nepričakovane napake pri procesiranju, lahko planer izvrže napako *PlannerException*, ki bo bila zabeležena v sistemskem dnevniku.

Abstraktni razred *Planner* definira tudi dve pomožni metodi, ki olajšata generiranje elementov plana. Metoda *addPickItemToPlan(...)* omogoča kreiranje novega elementa plana, kjer delavec pobere blago iz določene police. Kot argumente sprejme dosedanji plan, pobiralca za katerega se izdeluje plan, blago, ki se naj pobere, naročilo, kateremu blago pripada in ID začasnega sveta, ki ga uporablja planer. Metoda ustrezno posodobi stanje naročila, pobiralca, doda k planu nov element in posodobi stanje kopije modela na stanje, po izvedbi pravkar generiranega elementa plana. Druga metoda je *addUnloadItemsAtExitToPlan*, ki omogoča generiranje elementa plana odlaganja blaga v odpremnem prostoru skladišča. Kot argumente sprejme dosedanji plan, pobiralca za katerega se izdeluje plan, izhod in ID kopije modela, ki ga uporablja planer. Generira se eden ali več elementov plana, kjer vsak element opisuje odlaganje blaga za eno naročilo (na primer, ko pobiralec pobira za več naročil hkrati) in ustrezni posodobi stanje delavca in kopije modela.

Obe pomožni metodi, opisani v prejšnjem odstavku, dodata generirane korake planov v vrsto. Sistem namreč upošteva le korake planov, ki pridejo preko vrste. Kljub temu mora planer po končanem procesiranju vrniti celoten plan dela. Le tako se proces planiranja pravilno zaključi.

Preden lahko planer uporabimo, ga je potrebno registrirati s sistemom. Polno ime razreda, skupaj s paketom v katerem se nahaja, je potrebno vnesti v svojo vrstico v datoteko *planners.properties*, ki se nahaja v imeniku *./src/main/resources*. Sistem bo ob zagonu prebral to datoteko in registriral vse planerje, katerih imena so v njej.

6.4 Primer implementacije lastnega planerja

Prikazali bomo primer implementacije preprostega planerja *DumbPlanner*, ki generira zelo neučinkovit plan dela za enega pobiralca.

```
public class DumbPlanner extends Planner {

  public SimplePlanner(QueryModel model,
      BlockingQueue<PlanElement> queue) {
        super(model, queue);
  }

  public EnumSet<PlannerCapability> getCapabilities() {
    return EnumSet.of(PlannerCapability.ORDER);
  }

  protected List<Plan> getPlans(int processID, Set<Integer>
      pickersID, List<? extends Document> documents,
      ItemEvaluators evaluator, Map<String, Double>
      evalProperties, int virtualHandle, int planLength,
      Map<String, String> properties) throws PlannerException
  {...}

  private void getItems(Plan plan, Worker worker, int handle,
      OrderDocument doc, String itemID, BigDecimal quantity)
  {...}
}
```

Koda 6.1: Osnovno ogrodje planerja

Implementacija ogrodja našega planerja je prikazana v kodi 6.1. Implementirali smo metodo *getCapabilities()*, ki vrača dokumente, ki jih planer lahko sprejme. Ker naša rešitev trenutno omogoča le planiranje pobiranja blaga, imajo vsi planerji enak atribut, in bodo prejeli v obdelavo le naročila.

V kodi 6.2 je implementirana metoda *getPlans(...)*. Metoda vzame prvega pobiralca, ki je na voljo in zanj ustvari nov plan. Nato za vsako naročilo in vsak vrsto blaga v naročilu pokliče metodo *getItems(...)*. Ko je naročilo

```
Worker worker = model.getWorker(workersID.iterator().next());
Plan plan = new Plan(processID, worker.getId());
for (Document doc : documents) {
  OrderDocument ordDoc = (OrderDocument)doc;
  for (DocumentEntry entry : ordDoc.getUnprocessedEntries()) {
    getItems(plan, worker, virtualHandle, ordDoc, entry
        .getItemIdentifier(), entry.getQuantity());
  }
  // izpraznimo pobiralca pri spremembi naročil
  if (!worker.isEmpty()) {
    ExitMicrolocation exit = model
        .getExitMicroLocations(ExitType.REMOVE).get(0);
        addUnloadItemsAtExitToPlan(plan, worker, exit,
            virtualHandle);
  }
}
return Collections.singletonList(plan);
```

Koda 6.2: Implementacija metode *getPlans(...)*

izpolnjeno, preveri, če pobiralec prevaža blago. V tem primeru generiramo navodila za odlaganje blaga s pomočjo metode nadrazreda *addUnloadItemsAtExitToPlan(...)*. S tem preprečimo, da bi imel pobiralec pri sebi hkrati blago za več naročil.

V kodi 6.3 je implementirana metoda *getItems(...)*. Namen metode je generiranje navodil, za pobiranje specifičnega vrste blaga v želeni količini. Pobiralec bo pobiral blago, dokler ne nabere zadostne količine ali ni več blaga, ki bi ga lahko pobral. Za generiranje navodil za pobiranje blaga smo uporabili metodo nadrazreda *addPickItemToPlan(...)*. Po vsakem pobranem blagu preverimo, če je pobiralec že dosegel maksimalno kapaciteto. V tem primeru generiramo navodilo za izpraznitev z metodo nadrazreda *addUnloadItemsAtExitToPlan(...)*.

Planer *DumbPlanner* prikazuje uporabo ogrodja za pisanje in poganjanje planerjev. Čeprav je to primer delujočega planerja, nismo upoštevali neka-

```
while (!quantity.equals(BigDecimal.ZERO)) {
  // izvajamo, dokler ni izpolnjena količina
  ItemEvaluator eval = ItemEvaluators.DISTANCE.getEvaluator(
      model, worker.getPosition(), null);
  // izberemo najbližje blago
  EvaluatedObject<AbstractItem> ei = model.getBestItem(eval,
      BusinessProcess.Name.TAKE_FROM_RACK, itemID);
  if (ei == null) {
    // ni blaga v skladišču
    return;
  }
  AbstractItem item = ei.getObject();
  if (quantity.compareTo(item.getQuantity()) < 0) {
    // potrebujemo manjšo količino
    item = model.split(item, quantity);
  }
  addPickItemToPlan(plan, worker, item, doc, handle);
  quantity = quantity.subtract(item.getQuantity());
  if (worker.isFull()) {
    // pobiralec je poln in ga moramo izprazniti
    ExitMicrolocation exit = model.getExitMicroLocations
        (ExitType.REMOVE).get(0);
        addUnloadItemsAtExitToPlan(plan, worker, exit, handle);
  }
}
```

Koda 6.3: Implementacija metode *getItems(...)*

terih parametrov in smernic. Planer ne upošteva parametra o funkciji ocene cene pobiranja blaga, ampak uporablja kot oceno kar razdaljo. Prav tako ne omejuje dolžine generiranega plana, ki je podana s parametrom *planLength*. Ker je planer preprost, smo izpustili tudi preverjanje, če je bil podan ukaz za prekinitev izvajanja, ki je pri kompleksnejših planerjih nujen za odzivnost sistema.

Poglavje 7

Kontrolna enota

Kontrolna enota predstavlja del naše rešitve, ki je odgovoren za inicializacijo sistema in za komunikacijo z zunanjim okoljem. Definira programski vmesnik preko katerega zunanji sistemi uporabljajo našo rešitev. Naloge kontrolne enote so naslednje:

- inicializacija modela in planerjev
- vzpostavitev sistema v primeru sesutja
- dodajanje in brisanje pobiralcev
- sprejemanje dokumentov, ki predstavljajo naročila, zagon ustreznih planerjev in sinhrono ali asinhrono vračanje planov
- preverjanje osnovne pravilnosti generiranih planov
- ustrezno posodabljanje modela glede na dogodke, kot so potrditev izvedbe koraka plana ali spremenjeno stanje na polici regala
- odziv na izjemne dogodke (na primer neujemanje elektronske zaloge z dejansko)

7.1 Inicializacija in delovanje

Kontrolna enota je implementirana v paketu *lui.warehouseplanning.core*. Inicializira se izvede s klicem funkcije *newDispatcher(...)* v razredu *Dispatcher*, ki kot argumente sprejme razred, ki implementira vmesnik *ModelLoader*, razred, ki implementira *DispatcherStateManager*, in definicijo planerjev, ki jih bo uporabljal sistem. Vzpostavitev sistema se začne z nalaganjem modela preko funkcije *loadModel()*, ki jo definira vmesnik *ModelLoader*. Nato se izvede inicializacija razreda *PlannerManager*, ki je drugi del kontrolne enote in skrbi za inicializacijo in zagon planerjev. Instanca razreda *PlannerManager* naloži planerje, ki so bili specificirani v konfiguraciji, in jih pripravi za izvajanje. Osnovna inicializacija sistema se s tem zaključi. Sistem preveri s funkcijami, definiranimi v vmesniku *DispatcherStateManager*, če je prišlo do nepredvidenega zaprtja aplikacije. V tem primeru izvede še obnovitev stanja sistema.

Ko je kontrolna enota inicializirana, lahko sprejme naročila preko funkcije *submitDocuments(...)* v razredu *Dispatcher*, ki kot argument sprejme zbirko instanc razreda *lui.warehouseplanning.core.documents.Order*, ki predstavljajo naročila. Če je v sistemu aktiven vsaj en pobiralec bo kontrolna enota zagnala planer in vrnila ID procesa, pod katerim se izvaja planiranje. Celoten plan lahko dobimo s klicem funkcije *getPlans(...)* v razredu *Dispatcher*, ki kot argument sprejme ID procesa in število milisekund, ki lahko pretečejo preden funkcija zaključi izvajanje. Če plan ni na voljo pred potekom podanega števila milisekund, funkcija vrne *null*. Drugi način je asinhrono prejemanje korakov plana, ki je mogoče z registracijo poslušalca *PlanElementListener*. Kontrolna enota bo za vsak korak plana poklicala funkcijo *planElementProduced(...)*, z instanco razreda *PlanElement* iz razreda *lui.warehouseplanning.planners*. Na ta način ni potrebno čakati na celoten plan ampak se delo lahko izvaja vzporedno s planiranjem.

V razredu *Dispatcher* je implementirano tudi osnovno preverjanje pravilnosti generiranih korakov plana. Za vsako instanco razreda *PlanElement* se preveri naslednje:

- korak plana vsebuje veljaven ID procesa in ID delavca
- vrstno število koraka je za ena večje od prejšnjega prejetega koraka (kontrolna enota ne podpira prejemanja korakov plana v mešanem vrstnem redu)
- plan še ni bil zaključen
- obstaja dokument na katerega se sklicuje korak plana

V primeru, da korak plana ne prestane preverjanja, se izpiše opozorilo in zavrže element. Metoda *planElementProduced(...)* se v tem primeru ne pokliče.

7.2 Vzpostavitev sistema v primeru sesutja

Kontrolna enota je sposobna vzpostaviti svoje prejšnje stanje, če vsebuje funkcionalno implementacijo vmesnika *DispatcherStateManager*. Če ta funkcionalnost ni zaželena, lahko pri inicializaciji kot argument podamo *Initializer.DUMMY_STATE_MANAGER* iz paketa *lui.warehouseplanning*. Vzpostavitev stanja poteka v več fazah.

V pripravljalni fazi se v sistem registrira vse pobiralce, ki jih vrne metoda *getAllWorkers()*, in aktivira vse aktivne pobiralce. Pridobi se vsa aktivna oziroma nedokončana naročila preko funkcije *getOrders()*. Če obstajajo naročila in aktivni delavci, potem se izvede naslednja faza, ki poskuša vzpostaviti stanje pred zaprtjem.

Preko funkcije *getPlanElements()* se pridobijo vsi koraki planov, ki se jih uredi od najmanjšega do največjega po njihovih ID-jih. Starejši koraki planov imajo manjši ID kot mlajši. Sledi vzpostavljanje stanja v treh korakih:

1. Najprej se kontrolna enota sprehodi po korakih planov od najnovejšega do najstarejšega. Razreši reference do blaga, na katere se sklicujejo koraki plana, označeni kot izvedeni. Poustvari stanje delavcev in odpremnih prostorov, kot je bilo v času sesutja. Posodobi stanje dokumentov, da ustrezajo trenutnemu stanju skladišča.

2. Nato se kontrolna enota sprehodi po korakih plana v vrstnem redu. Razreši reference do blaga, ki jih opisujejo neizvedeni koraki planov.

3. V zadnjem koraku se kontrolna enota še enkrat sprehodi po korakih planov v vrstnem redu. Ustvari ustrezne interne strukture in vanje vstavi že izvedene korake plana. Neizvedene korake plana preveri na isti način, kot pravkar generirane korake iz planerja. Če ne prestanejo preverjanja so zavrženi.

Po končani vzpostavitvi stanja, kontrolna enota preveri, če obstaja naročilo, ki še nima celotnega plana dela. V tem primeru kontrolna enota ponovno zažene planer.

7.3 Odziv sistema na izjemne dogodke

Naša rešitev deluje v okolju, nad katerim nimamo nadzora. Zato mora biti sistem sposoben prilagajati in posodabljati svoje stanje tudi ob izjemnih dogodkih. Primeri takih dogodkov so neujemanje elektronske zaloge z dejansko, nezmožnost izpolnitve koraka plana pobiranja zaradi poškodbe blaga ali nepredviden odhod pobiralca. Kontrolna enota implementira odziv sistema na takšne dogodke preko metode *control(...)*. Prvi argument je tip dogodka, preostala dva pa sta dodatna argumenta. Tipi dogodkov in odzivi nanje bodo opisani v naslednjih odstavkih.

Neujemanje elektronske zaloge z dejansko je posledica pomanjkljivega evidentiranja akcij v skladišču s sistemi WMS ali človeške napake. Do ugotovitve o neujemanju pride ob pregledu police oziroma inventuri ali pri pobiranju blaga po danem planu dela. Če je neujemanje ugotovljeno pri pobiranju blaga, nabiralec zavrne trenutni korak plana dela. Sistem WMS pokliče metodo kontrolne enote z naštevnim tipom *ControlKey.REFUSE_PLAN_ELEMENT* in ID-jem koraka plana, ki ga pobiralec zavrača. Če je pobiralec že pobral nekaj blaga, njihovo količino sporoči sistem WMS kot tretji argument. Kontrolna enota v tem primeru razveljavi oziroma posodobi korake plana, na

katere vpliva zavrnitev. Ponovno zažene tudi planer s tistimi naročili, ki ob trenutnem planu dela ne bodo izpolnjena.

Novo stanje police sistem WMS sporoči s klicem metode kontrolne enote z naštevnim tipom *ControlKey.WARE_CHANGED* in ID-jem police, s spremenjenim stanjem. Kontrolna enota nato pokliče metodo *getItemListOnRackMicrolocation(...)* vmesnika *ModelLoader*, ki je odgovoren za nalaganje modela. Preko te metode kontrolna enota pridobi novo stanje police in ustrezno posodobi model. Preveri tudi vpliv spremembe na trenutne plane dela. Če kateri izmed korakov ni več mogoč, ga kontrolna enota razveljavi in ustrezno posodobi korake plana, na katere vpliva razveljavitev. Ponovno zažene tudi planer s tistimi naročili, ki ob trenutnem planu dela ne bodo izpolnjena. Ker naša rešitev trenutno ne implementira planiranja skladiščenja blaga, lahko s klicem te funkcije sistem WMS obvesti kontrolno enoto tudi o spremembi stanja zaloge zaradi skladiščenja.

Nezmožnost izpolnitve koraka plana pobiranja zaradi neustreznega stanja blaga je posledica poškodovanega ali okvarjenega blaga. Pobiralec pri izpolnjevanju koraka plana pobiranja pregleda blago in ugotovi njegovo neustreznost. V tem primeru zavrne korak plana. Ker blago zaradi neustreznosti ni več na voljo, je postopek enak kot pri neujemanju elektronske zaloge z dejansko.

Prihodi in odhodi pobiralcev vplivajo na to, kdo je na voljo za izvrševanje plana dela. Naša rešitev predpostavlja, da je pred planiranjem znano število pobiralcev. Odhodi pobiralcev oziroma prihodi so zato izjemni dogodki. Prihod novega pobiralca sistem WMS sporoči s klicem metode kontrolne enote z naštevnim tipom *ControlKey.ACTIVATE_WORKER* in ID-jem pobiralca. Pobiralec mora biti pred tem registriran z metodo kontrolne enote *createNewWorker(...)*. Pobiralec bo bil na voljo ob naslednjem planiranju dela. Odhod delavca sistem WMS sporoči s klicem metode kontrolne enote z argumentoma *ControlKey.DEACTIVATE_WORKER* in ID-jem pobiralca. Pobiralec ne bo več na voljo pri planiranju dela. Kontrolna enota ne razveljavi že ustvarjenega plana dela za odsotnega pobiralca. Sistem WMS

lahko plan dela preda drugemu pobiralcu ali pa ga razveljavi s klicem metode kontrolne enote z naštevnim tipom *ControlKey.REFUSE_PLAN_ELEMENT* in ID-jem koraka plana za vsak korak, ki je del plana odsotnega pobiralca.

7.4 Programski vmesnik za manipulacijo sistema

Do sistema se dostopa preko programskega vmesnika, definiranega v razredu *Dispatcher*. Razred definira naslednje metode:

- *submitDocuments(...)*, preko katere sistem sprejme naročila in začne procesiranje
- *getPlans(...)*, preko katere lahko pridobimo generirane plane
- *getOrder(...)*, ki vrne že sprejeto naročilo s podanim ID-jem
- *createNewWorker()*, ki v sistem registrira novega pobiralca
- *getPlannersData()*, ki vrne konfiguracijo planerjev
- *getLoaderInfo()*, ki vrne opis razreda, odgovornega za nalaganje modela skladišča
- *control(...)*, ki omogoča izvedbo različnih ukazov v odvisnosti od prvega parametra

V razredu *Dispatcher* lahko registriramo tudi tri vrste poslušalcev. *PlanElementListener* omogoča asinhrono sprejemanje korakov plana in sprejemanje obvestil o spremembah ali razveljavitvah plana dela. Registrira se ga z metodo *setPlanElementListener(...)*. *DocumentListener* omogoča prejemanje obvestil o spremembah stanja naročil, kot je izpolnitev naročila. Registrira se ga z metodo *setDocumentListener(...)*. *ModelResponseListener* je vmesnik, ki se uporablja pri razhroščevanju. V primeru, da je pri spremembi stanja modela skladišča prišlo do napake, lahko preko tega vmesnika dobimo

natančno sporočilo o vzroku. Registrira se ga z metodo *setModelResponseListener(...)*.

Metoda *control(...)*, kot prvi parameter prejme ID akcije, ki naj jo izvede, preostali parametri pa predstavljajo argumente te akcije. Akcije so naslednje:

CANCEL_PROCESS Ustavi proces z ID-jem podanim kot drugi parameter. Vrne *true*, če proces obstaja in še ni bil zaključen.

CONFIRM_PLAN_ELEMENT Potrdimo izvedbo koraka plana z ID-jem podanim kot drugi parameter. Kontrolna enota posodobi stanje modela. Če je kot tretji argument podana količina blaga, ki se ne ujema s količino opisano v koraku plana, kontrolna enota v model doda razliko.

DAY_PASSED Datum se premakne za en dan naprej.

ACTIVATE_WORKER Aktivira pobiralca z ID-jem, podanim kot drugi argument. Pobiralec postane na voljo za planiranje. Pobiralec mora biti prej v sistemu registriran z metodo *createNewWorker(...)*.

DEACTIVATE_WORKER Pobiralec ni več na voljo za planiranje. Plani, ki so zanj že bili generirani, so še vedno na voljo.

REFUSE_PLAN_ELEMENT Zavrne korak plana z ID-jem podanim kot drugi argument. S tretjim argumentom povemo količino, ki je lahko nič ob polni zavrnitvi ali večja od nič ob delni zavrnitvi. Kontrolna enota poskuša vzpostaviti konsistentno stanje in ponovno izvesti planiranje.

REMOVE_WARES_FROM_EXIT Zapre naročilo z ID-jem podanim kot drugi argument in odstrani vse blago iz odpremnega prostora, ki pripadajo temu naročilu. Vsi koraki planov, ki so delali na tem naročilu, so odstranjeni iz sistema.

WARE_CHANGED Obvesti kontrolno enoto, da je prišlo do spremembe na polici z ID-jem podanim kot drugi argument. Kontrolna enota pri-

dobi dejansko stanje police preko metode *getItemListOnRackMicrolocation(...)* vmesnika *ModelLoader*.

Poglavje 8

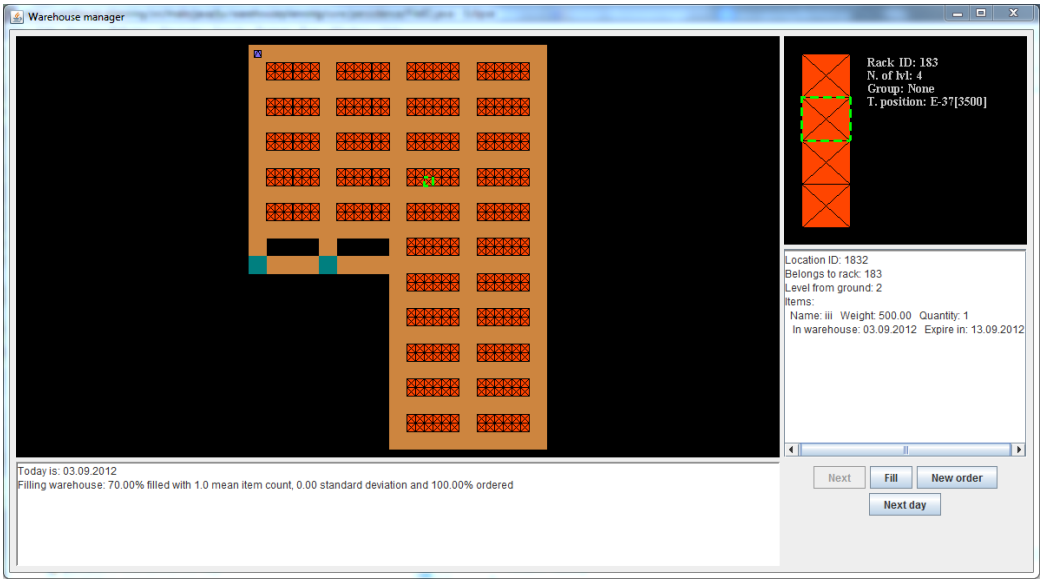
Vizualizacija izvajanja simulacije

Za potrebe vizualizacije modela skladišča in delovanja planerjev smo implementirali vizualizacijo izvajanja simulacije. Vizualizacijo se požene iz ukazne vrstice. Primer zagona je naslednji:

```
> java -jar warehouse-planning-0.5.1-SNAPSHOT.jar -G --seed 666 -s
./simulator.properties -m ./warehouse.model SimplePlanner[evaluato
r=DISTANCE]
```

Datoteka *warehouse.model* vsebuje definicijo modela skladišča in datoteka *simulator.properties* vsebuje parametre simulacije. Formati obeh datotek so opisani v dodatku D. V zgornjem primeru zagona se obe datoteki nahajata v imeniku, iz katerega zaganjamo aplikacijo. Podrobnejši opis različnih načinov zagona aplikacije je opisan v dodatku B.

Ob zagonu se odpre okno prikazano na sliki 8.1. Razdeljeno je na pet razdelkov. Zgornji levi razdelek, ki je tudi največji, prikazuje skladišče. Ceste in križišča so rjavo obarvane, regali so rdeči pravokotniki s križem in zelenomodra križišča so prevzemni/odpremni prostori. V skrajnem levem zgornjem kotu je temno moder pravokotnik, ki ponazarja pobiralca. Na vse objekte z izjemo križišč in cest je mogoče klikniti. Aktivni objekt je obrobljen z svetlo zeleno prekinjeno črto. Ob kliku na izhod ali regal se v predelku zgoraj desno



Slika 8.1: Uporabniški vmesnik vizualizacije simulacije

prikaže podrobnejši opis. Na sliki 8.1 je prikazan stranski pogled na regal s policami. Prikazane so tudi osnovne informacije regala, kot so njegov ID, število polic, skupina in topološka pozicija. Na posamezne police je mogoče klikniti. Ob kliku se v desnem sredinskem predelku izpišejo podrobnejše informacije. Na sliki 8.1 je izbrana druga polica in prikazane so informacije o tej polici, kot je ID, kateremu regalu pripada, nivo police in blago, ki ga vsebuje.

Klikniti je mogoče tudi na ikone pobiralcev. V desnem sredinskem oknu so v tem primeru izpisane informacije o pobiralcu, kot so ID, tip (pobiralec z viličarjem ali voziček), kapaciteta, število predelkov, trenutna topološka pozicija in blago, ki ga pobiralec trenutno prevaža.

Levo spodnje okno je namenjeno izpisu statusa skladišča. Vsaka izvedena akcija je opisana v tem oknu. Na sliki 8.1 je prikazan izpis trenutnega datuma in parametrov s katerimi je bilo izvedeno naključno polnjenje skladišča.

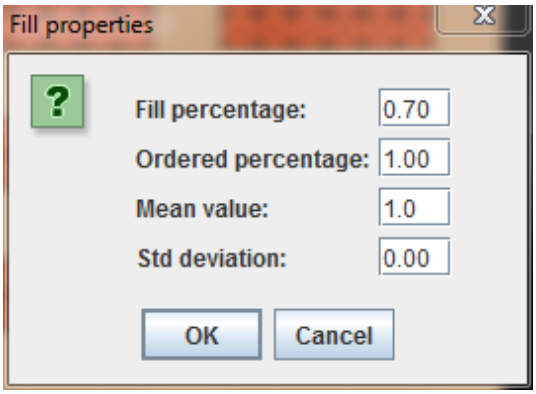
Spodnji desni del je kontrolna plošča, kjer so kontrole za upravljanje s skladiščem. Implementirane so naslednje kontrole:

Next Kontrola je aktivna, kadar je na voljo plan dela. Izvrši naslednji korak plana.

Fill Odpre meni, v katerem vnesemo parametre za naključno polnjenje skladišča. Po potrditvi se izvede polnjenje s podanimi parametri. Kontrola je aktivna, če ni na voljo plan dela.

New order Odpre meni za vnos novih naročil. Po potrditvi pošlje naročila v sistem. Kontrola je aktivna, če ni na voljo plan dela.

Next day Premakne interni datum skladišča za en dan naprej. Kontrola je aktivna, če ni na voljo plan dela.



Slika 8.2: Meni za naključno polnjenje skladišča

Na sliki 8.2 je prikazan meni s parametri za naključno polnjenje skladišča. Parametri so naslednji:

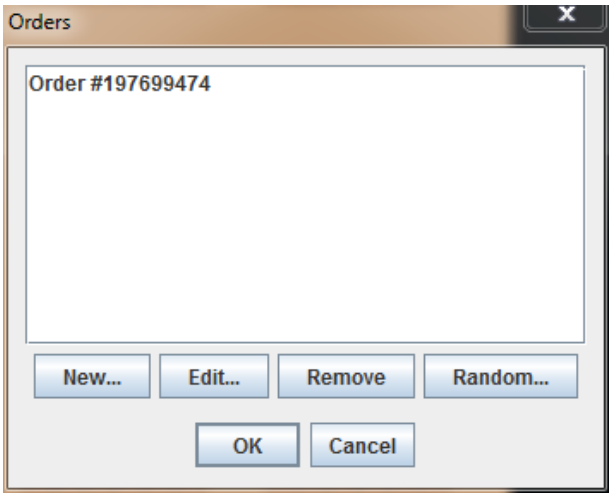
Fill percentage Opisuje odstotek polic v skladišču, ki bodo vsebovale blago. Vrednost parametra je realno število med 0 in 1.

Ordered percentage Opisuje odstotek urejenosti skladišča. Vrednost 1 predstavlja popolnoma urejeno skladišče, kjer so enake vrste blaga na sosednjih policah. Vrednost 0 predstavlja popolnoma neurejeno skladišče, kjer je blago povsem naključno razporejeno. Vrednost parametra je realno število med 0 in 1.

Mean value Povprečna količina blaga na polici. Vrednost parametra je pozitivno realno število.

Std deviation Standardna deviacija od povprečne količine blaga na polici. Vrednost parametra je pozitivno realno število.

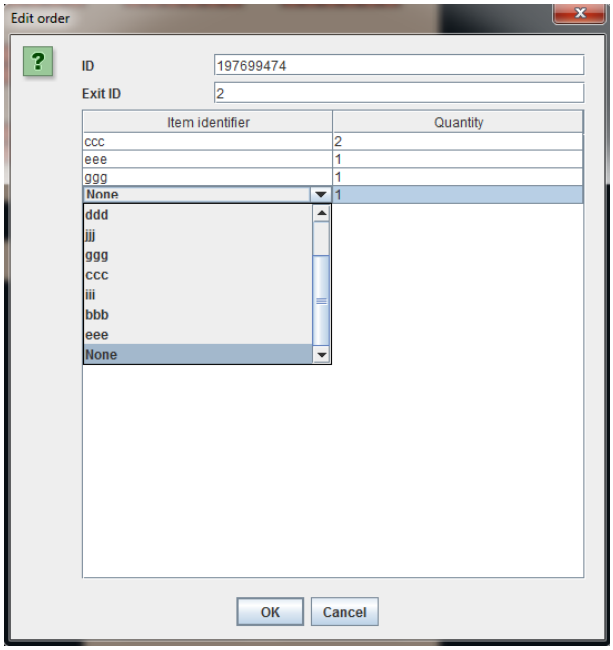
V grafični vizualizaciji bo končna količina blaga na polici vedno celo število. Sistem sicer zna upravljati s poljubnimi decimalnimi količinami.



Slika 8.3: Meni za upravljanje naročil

Na sliki 8.3 je prikazan meni za upravljanje naročil. V glavnem oknu so navedena vsa naročila, ki smo jih vnesli ali generirali. Z gumbom *New...* ustvarimo novo naročilo, z gumbom *Edit...* urejamo označeno naročilo in z gumbom *Remove...* izbrišemo označeno naročilo. Gumb *Random...* odpre nov meni, v katerem vnesemo število naročil in skupno količino vsega blaga na naročilo, nakar se naročila naključno ustvarijo.

Na sliki 8.4 je prikazan meni za urejanje ali ustvarjanje naročil. ID naročila in ID izhoda sta že izpolnjena, pri urejanju pa tudi želeno blago in količine. Z klikom na polje *Item identifier* se odpre drsni meni, iz katerega izberemo ID blaga. Nato v stolpcu *Quantity* vpišemo želeno količino. Če želimo nov vnos, pritisnemo tipko *Enter*. Ustvari se nova vrstica tabele, v katero vnesemo podatke.

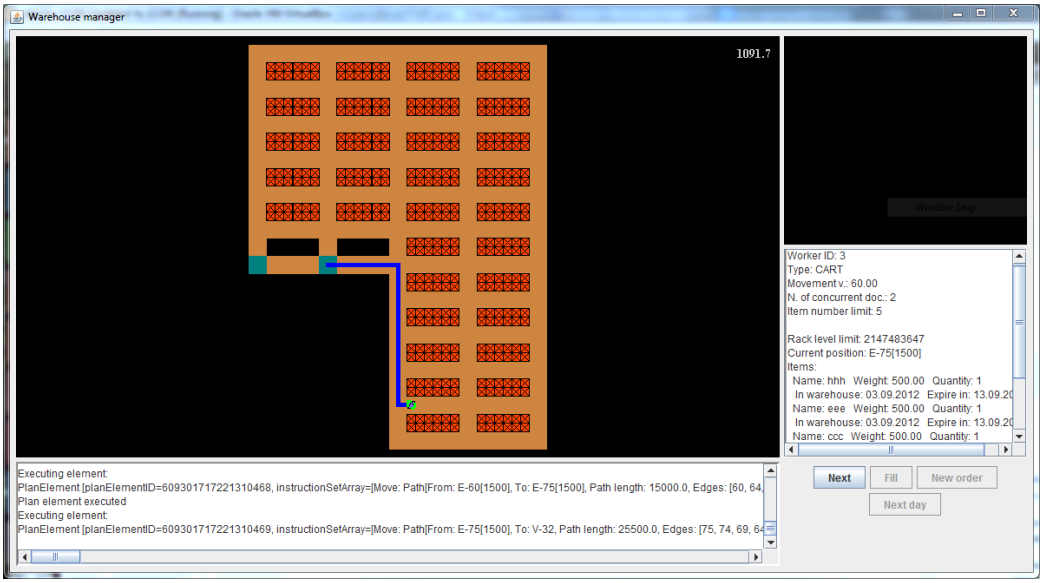


Slika 8.4: Meni za urejanje ali ustvarjanje naročil

Po potrditvi naročil aplikacija pošlje naročila sistemu, ki zažene planer. Med delovanjem planerja so vse kontrole neaktivne. Aplikacijo lahko še vedno zapremo z zaprtjem glavnega okna. V statusnem predelu glavnega okna se izpisujejo generirani koraki plana. Ko je na voljo celoten plan dela, se aktivira kontrola *Next*, s katero lahko simuliramo izvajanje plana.

Na sliki 8.5 je prikazan uporabniški vmesnik po nekaj izvedenih korakih plana. Vsak korak je sestavljen iz dveh stopenj: Najprej je prikazana pot, ki jo bo pobiralec opravil (modra črta na sliki 8.5), šele v naslednji stopnji se pobiralec premakne na pozicijo in izvede akcijo. Na sliki je označen pobiralec z ID 3, ki se odpravlja na izhod, da odloži izdelke. Nekateri od izdelkov, ki jih ima v vozičku so vidni v desnem sredinskem oknu.

V statusnem oknu se izpisujejo koraki plana, ki jih trenutno izvajamo, v zgornjem desnem kotu predelka, ki prikazuje skladišče, pa se izpisujejo ocenjeni časi izvajanja plana.



Slika 8.5: Grafični vmesnik pri izvajanju simulacije

Čeprav je na slikah 8.1 in 8.5 prikazan le en pobiralec, jih aplikacija podpira poljubno mnogo. Število in lastnosti pobiralcev so del datoteke s parametri simulacije.

Kontrola *Next* je na voljo, dokler celoten plan ni izvršen. Po izvršitvi postane neaktivna, preostale kontrole pa se ponovno aktivirajo.

Poglavje 9

Sklepne ugotovitve

V tem diplomskem delu smo razvili informacijski sistem, ki omogoča planiranje pobiranja blaga za naročila v skladiščih, kjer pobirajo blago po načelu pobiralec k blagu. Naš sistem je implementiran v programskem jeziku Java in je sestavljen iz treh modulov - model skladišča, ogrodje za pisanje in izvajanje planerjev ter kontrolne enote, ki omogoča komunikacijo z zunanjim okoljem in zagon planerjev. Poleg samega sistema smo razvili tudi vizualizacijo izvajanja ustvarjenega plana dela, s katero lahko preverjamo delovanje planerjev za poljubna naročila in prikažemo izvedbo ustvarjenih planov dela. Implementirali smo tudi testno okolje s katerim lahko samodejno simuliramo prihode naročil, ustvarjanje plana dela in njegovo realizacijo. S tem okoljem lahko na enostaven način ocenimo učinkovitost delovanja različnih planerjev.

Glavna prednost našega sistema je možnost optimizacije dela pobiralcev tudi v skladiščih, kjer blago pobirajo po principu pobiralec k blagu in običajno ne vsebujejo sistemov za optimizacijo dela. S fleksibilno definicijo modela je možno modeliranje poljubnega skladišča, ki ga opišemo s tekstovno datoteko ali podatki v relacijski podatkovni bazi. Implementacija modela temelji na grafu, kar predstavlja nov pristop k predstavitvi skladišča. V literaturi namreč zasledimo le implementacijo z mrežo. Naš sistem nudi tudi ogrodje za razvoj poljubnih planerjev, ki so lahko napisani specifično za dano skladišče. Ker planerji vsak korak plana vračajo v kontrolno enoto, lahko delo

steče takoj, ko je znan prvi korak plana za vsakega pobiralca. Prednost naše rešitve je tudi enostavna uporaba sistema. Kontrolna enota definira vmesnik, preko katerega je mogoče enostavno implementirati povezavo med našo rešitvijo in sistemi ali aplikacijami, ki jo bodo uporabljale.

Naša rešitev je bila načrtovana kot pomožni sistem sistemu WMS. Zato ne podvajamo zapisov v podatkovni bazi o stanju zaloge, temveč se zanašamo na sistem WMS, preko katerega pridobimo potrebne podatke. Naš sistem ne nadomesti sistema WMS v skladišču, ampak ponudi dodatno funkcionalnost, ki se jo lahko tudi izklopi.

Implementacija sistema je napisana v programskem jeziku Java. Ta omogoča izvajanje sistema na katerem koli operacijskem sistemu, za katerega je implementiran javanski virtualni stroj. Tudi uporabljeno podatkovno bazo lahko zlahka spremenimo, če priložimo ustrezen gonilnik JDBC in ustrezno nastavimo parametre za dostop do podatkovne baze. Čeprav je implementacija modela popolnoma vsebovana v delovnem pomnilniku, ne zahteva velike količine pomnilnika. Običajno zadošča, če je sistemu na voljo pol gigabajta delovnega pomnilnika. V primeru kompleksnejših planerjev so mogoče tudi večje potrebe.

Glavni slabosti našega sistema sta poenostavljeno modeliranje skladišča in planiranje akcij v okolju, nad katerim nimamo nadzora. S poenostavitvijo modela smo definirali, da se pobiralci premikajo po sredini cest in križišč, ter da je njihovo premikanje konstantno. V resnici se pobiralci srečujejo na cestah, umikajo en drugemu, gredo na odmore, kar vpliva na dejanski čas izvajanja plana, ki ni več nujno enak pričakovanemu. Tudi modeliranje blaga je zelo enostavno, ker ne modeliramo tistih vrst, ki jih je mogoče razdreti in pobirati v manjših kosih. Pobiralci lahko tudi naredijo napako pri pobiranju ali zavrnejo korak plana, na kar se mora sistem ustrezno odzvati. Naš sistem sicer vsebuje preproste odzive, vendar bi potreboval kompleksnejšo logiko za upravljanje v teh primerih.

Slabost našega sistema je tudi pomanjkanje logike za implementacijo planerjev za skladiščenje. Le hkratna optimizacija skladiščenja in pobiranja

blaga ter učinkovito vodenje pobiralcev, bi dala globalno optimalno rešitev upravljanja skladišča. Trenutno naš sistem podpira le možnost planiranja pobiranja blaga in vodenje pobiralcev.

S sistemom za planiranje akcij v skladišču smo želeli implementirati dodatni sistem rešitvi za upravljanje skladišča *Hydra@Warehouse*, razviti v podjetju *3R.TIM*. Z našo rešitvijo in implementacijo ustreznih planerjev lahko svojim strankam ponudijo napredno izdelavo plana dela pobiranja blaga za naročila. S tem planom dela lahko zmanjšajo čas, ki ga pobiralci porabijo za sprehod po skladišču, in dosežejo hitrejše izpolnjevanje naročil, s tem pa tudi učinkovitejše delovanje skladišča.

Naša rešitev implementira tudi testno okolje, s katerim lahko testiramo delovanje različnih planerjev znotraj simulacije skladišča. Na ta način lahko ovrednotimo planerje glede na različne parametre, preden se planerji uporabijo v realnem skladišču. Testno okolje je bilo uporabljeno v diplomskem delu Matica Horvata [7], v katerem je raziskal medsebojni vpliv med delovanjem različnih pristopov k planiranju dela pobiralcev in taktiko skladiščenja. Pokazal je, da lahko v neurejenih skladiščih dosežemo tudi do 14% krajše obhode kot v skladiščih, kjer se blago iste vrste nahaja na sosednjih lokacijah. Primerjava s trenutno uporabljenimi načini pobiranja ostaja področje za nadaljnje delo.

Prihodnji razvoj sistema bi bil usmerjen v podrobnejše modeliranje blaga. Dodali bi modeliranje blaga, ki ga pobiralci lahko razdrejo, in ukaze, ki bi podpirali takšen način dela. Izboljšati bi bilo potrebno tudi implementacijo odzivov na izjemne dogodke, s katero bi dosegli boljšo odzivnost in stabilnost sistema. Za uporabo sistema z rešitvijo *Hydra@Warehouse* bi bilo potrebno tudi implementirati povezavo med sistemoma. Z implementacijo te povezave bi dosegli praktično uporabnost naše rešitve. Kot razširitev funkcionalnosti sistema bi lahko implementirali logiko, ki bi omogočala planiranje skladiščenja. S to razširitvijo bi izdelali zaokrožen sistem, s katerim bi bilo hkrati mogoče optimizirati upravljanje zaloge in pobiranje blaga za naročila.

Literatura

[1] R. de Koster, T. Le-Duc, and K. J. Roodbergen, "Design and control of warehouse order picking: A literature review," *European Journal of Operational Research*, vol. 182, no. 2, pp. 481 – 501, 2007.

[2] D. Lambert, J. Stock, and L. Ellram, *Fundamentals of logistics management*, ch. 8. The Irwin/McGraw-Hill series in marketing, Boston [u.a.]: Irwin/McGraw-Hill, 1998.

[3] M. ten Hompel and T. Schmidt, *Warehouse Management: Automation and Organisation of Warehouse and Order Picking Systems.* Springer, 2007.

[4] F. Dallari, G. Marchet, and M. Melacini, "Design of order picking system," *The International Journal of Advanced Manufacturing Technology*, vol. 42, pp. 1–12, 2009. 10.1007/s00170-008-1571-9.

[5] M. Keith and M. Schincariol, *Pro JPA 2: Mastering the Java Persistence API.* Berkely, CA, USA: Apress, 1st ed., 2009.

[6] Z. Haijun and L. Bingwu, "Data structure design and order-picking optimization for irregular warehouse," in *Management and Service Science, 2009. MASS '09. International Conference on*, pp. 1 –4, sept. 2009.

[7] M. Horvat, "Pristop k optimizaciji zbiranja blaga za odpremo v skladiščih," Diplomsko delo, Fakulteta za računalništvo in informatiko, Univerza v Ljubljani, Ljubljana, 2012.

Dodatek A

Navodila za vzpostavitev okolja in prevajanje aplikacije

V razvojnem okolju mora biti nameščena *Java 7 JDK SE* (dostopna na strani www.oracle.com/technetwork/java/javase/) in *Eclipse Java EE IDE* (dostopen na strani www.eclipse.org). Konfiguracija bo prikazana za *Eclipse Java EE IDE Indigo Service Release 2* nameščenem na *Windows 7 SP 1*. Pojdite v imenik, kjer je nameščen *Eclipse IDE*, in odprite datoteko *eclipse.ini*. V datoteki popravite argument *-vm* na lokacijo *javaw.exe* znotraj *Java JDK*. Primer konfiguracije je:

```
-vm
C:/Program Files/Java/jdk1.7.0_01/bin/javaw.exe
-vmargs
-Xms128m
-Xmx1024m
```

Pazite, da je argument *-vmargs* za argumentom *-vm*. Po želji lahko spreminjate začetno in maksimalno velikost delovnega pomnilnika, ki je na voljo *JVM*.

V *Eclipse IDE*-ju namestite vtičnik za integracijo z upravljavcem projektov *Maven*. Odprite meni *Help* in kliknite *Install New Software....* V *Work with...* vpišite http://download.eclipse.org/technology/m2e/releases. Označite *Maven Integration for Eclipse* in sledite navodilom za namestitev. Po končani

namestitvi odprite meni *Window* in kliknite *Preferences*. Navigirajte po drevesu do *Java* in kliknite na *Installed JREs*. Dodajte nov *JRE* z izbiro imenika, v katerem je nameščen *Java 7 JDK*. Označite nov *JRE* kot privzeti. Zaprite *Eclipse IDE*. Odprite *Control Panel* in izberite *System and Security*, *System*, *Advanced System Settings* in *Environment Variables...*. Ustvarite novo sistemsko spremenljivko *JAVA_HOME*, ki kaže na imenik, kjer je nameščen *Java 7 JDK*. Primer spremenljivke:

```
JAVA_HOME: C:\Program Files\Java\jdk1.7.0_01
```

Odprite *Eclipse IDE*. Kliknite meni *File* in izberite *Import...*. Izberite *Maven* in *Existing maven projects*. Kot *Root directory* izberite imenik, v katerem se nahaja projekt. Označite *pom.xml lui:warehouseplanning:0.5.1-SNAPSHOT* in pritisnite *Finish*.

Če želite uporabljati podatkovno bazo, morate projekt pretvoriti še v projekt *JPA*. Z desnim klikom kliknite na projekt, izberite *Configure* in *Convert to JPA Project...*. V levem oknu naj bosta izbrana le Java verzija 1.7 in *JPA* 2.0. Kliknite dvakrat *Next...* in nato v oknu *JPA Facet* nastavite naslednje komponente:

```
Platform: EclipseLink 2.2.x
JPA Implementation Type: Disable Library Configuration
Persistent class management: Annotated classes must be listed in
  persistence.xml
```

Da sistem lahko komunicira s podatkovno bazo, potrebuje še gonilnik *JDBC* za želeno podatkovno bazo. Prikazana bo konfiguracija za *MS SQL Server 2008*. Iz spletne strani http://jtds.sourceforge.net/ si snemite gonilnik *JDBC* in razširite arhiv. Kliknite z desnim klikom na projekt, izberite *Properties* in nato *JPA*. Pod poljem *Connection* klinkite *Add connection...*. Izberite *Generic JDBC* in kliknite *Next...*. Pod *Drivers* kliknite *New Driver Definition*. Kliknite *Generic JDBC Driver* in ga preimenujte v *jTDS Driver*. Kliknite zavihek *JAR List*. Z *Add Jar/Zip...* izberite izberite datoteko *jtds-<verzija>.jar* v imeniku, ki vsebuje *jTDS driver*. Nato kliknite zavihek *Properties*. Izpolnite polja s sledečimi vrednostmi:

```
Connection URL: jdbc:jtds:sqlserver://<IP>:<port>/
Database Name: <ime podatkovne baze>
Driver Class: net.sourceforge.jtds.jdbc.Driver
User ID: <uporabniško ime>
```

Kliknite *OK* in vpišite še geslo za dostop do podatkovne baze ter označite *Save password*. Preverite, če povezava deluje s *Test connection*. V primeru neuspeha preverite, da so vsi parametri povezave pravilni. Po uspešnem testu kliknite *Finish* in *OK*.

Znotraj *Eclipse IDE* se aplikacija samodejno prevaja. Če pa želimo samostojno aplikacijo, je potrebno s pomočjo upravljalnika projektov *Maven* ustvariti arhiv JAR. Pojdite v meni *Run* in izberite *Run Configurations...*. Dvakrat kliknite profil *Maven build* in definirajte sledeče parametre:

```
Base directory: <imenik projekta>
Goals: package
Profiles: simulation | deployment
```

Za profil si izberite *simulation* ali *deployment*. Profil *simulation* bo zapakiral celotno aplikacijo z vsemi potrebnimi knjižnicami v en arhiv JAR. Arhiv se bo nahajal znotraj imenika *target* z imenom *warehouse-planning-<verzija>-SNAPSHOT.jar*. Namenjen je testiranju aplikacije.

Profil *deployment* je namenjen izgradnji polno funkcionalne aplikacije. Aplikacija se nahaja znotraj imenika *target*, vse potrebne knjižnice pa znotraj imenika *target\lib*. Med knjižnicami je tudi gonilnik *jTDS JDBC*, s katerim je mogoča povezava z podatkovno bazo *MS SQL*.

Dodatek B

Zagon aplikacije

Znotraj *Eclipse IDE* se aplikacija zaganja kot javanska aplikacija. Z desnim klikom kliknite na projekt, *Run As* in nato *Run Configurations....* Dvakrat kliknite na *Java application.* Spremenite ime profila in kot glavni razred izberite *lui.warehouseplanning.Loader.* V zavihku *Arguments* dodajte želene argumente.

Aplikacijo v arhiva JAR poganjamo preko ukazne vrstice. Primer ukaza je naslednji:

```
> java {jar warehouse-planning-<verzija>.jar <argumenti>
```

Argumente, ki jih uporablja aplikacija, lahko pridobimo z ukazom *-h.* Argumenti, ki jih aplikacija uporablja so:

```
Uporaba warehouse-planning [-h|--help] -S|G|I [parametri]

  [-h|--help]
        Prikaže to sporočilo
  -S|G|I
        Način izvajanja: (S)imulacija, (G)rafika, (I)nteraktivni

 Način simulacije:
 [-S] [-d <dni>] [-o <naročil>] [-i <blaga>] -s <simulator> -m
 <model> [--seed <seme>] planer1 planer2 ... planerN
```

```
Omogoča samodejno testiranje različnih planerjev.

[-S]
      Izvajanje simulacije
[-d <dni>]
      Število dni izvajanja simulacije (privzeto: 10)
[-o <naročil>]
      Število naročil na dan simulacije (privzeto: 4)
[-i <blaga>]
      Količina blaga na naročilo (privzeto: 10)
-s <simulator>
      Pot do simulacijske datoteke
-m <model>
      Pot do datoteke modela skladišča
[--seed <seme>]
      Če je argument prisoten, nastavi seme za generiranje
      naključnih števil na podano vrednost
planer1 planer2 ... planerN
      Planerji, ki jih bo uporabljal sistem. Vsak planer ima
      podane parametre ločene z vejico znotraj oglatih oklepajev.
      Primer: SimplePlanner[evaluator=DAYS_IN_WAREHOUSE,
      eval_props={weight1:0.5}]

Grafični način:
[-G] -s <simulator> -m <model> [--seed <seme>] planer1 planer2 ...
planerN

Omogoča vizualizacijo izvajanja plana dela, kot ga proizvede
planer.

[-G]
      Grafična vizualizacija
```

```
 -s <simulator>
       Pot do simulacijske datoteke
 -m <model>
       Pot do datoteke modela skladišča
 [--seed <seme>]
       Če je argument prisoten, nastavi seme za generiranje
       naključnih števil na podano vrednost
 planer1 planer2 ... planerN
       Planerji, ki jih bo uporabljal sistem. Vsak planer ima
       podane parametre ločene z vejico znotraj oglatih oklepajev.
       Primer: SimplePlanner[evaluator=DAYS_IN_WAREHOUSE,
       eval_props={weight1:0.5}]

Interaktivni način:

Omogoča priklop na podatkovno bazo in vzpostavitev komunikacije z
zunanjim s Hydra@Warehouse.

[-I] [(-c|--config) <config>]
 [-I]
       Interaktivni način
 [(-c|--config) <config>]
       Pot do konfiguracijske datoteke
```

Dodatek C

Uporabljene zunanje knjižnice

Aplikacija uporablja spodaj naštete zunanje knjižnice. Vse uporabljene knjižnice upravljalnik projektov *Maven* samodejno prenese ob izgradnji aplikacije.

Jsap 2.1 je knjižnica, ki omogoča enostavno izgradnjo razčlenjevalnika za parametre programa, preverjanje strukturne pravilnosti in ustreznosti argumentov, ter prevajanje parametrov v ustrezne objekte, ki jih sistem potrebuje za inicializacijo. Na voljo pod licenco *LGPL*.

Miglayout 3.7.4 je knjižnica, ki definira ureditev grafičnih elementov v okolju *Swing*. Omogoča enostavno definiranje umestitve elementov in všečno razporeditev. Precej lažja in enostavnejša za uporabo kot so sistemski upravljavci razporeditve. Na voljo pod licenco *BSD*.

Log4j 1.2.16 je knjižnice za logiranje stanja sistema. Z datoteko *src/main/resources/log4j.properties* določimo nivo logiranja in način izpisa. Na voljo pod licenco *The Apache Software License 2.0*.

Slf4j-api 1.6.4 in **slf4j-log4j12** sta knjižnici za logiranje stanja sistema. Aplikacija neposredno uporablja le *slf4j-api*, ki je le API za logiranje. *Slf4j-log4j12* je implementacija te specifikacije, ki pa je le ovojna knjižnica za *log4j*, ki je dejanska knjižnica za logiranje. Razloga za uporabo te sheme sta večja fleksibilnost, ker lahko uporabimo katerikoli loger brez posegov v kodo, in pospešitev izvajanja kode, ker se združevanje nizov izvede le v primeru, če

nivo logiranja dopušča izpis niza. Na voljo pod licenco *The Apache Software License 2.0*.

Javax.persistence 2.0.3 in **eclipselink 2.2.0** sta knjižnici, ki definirata in implementirata JPA. Knjižnici abstrahirata dostope do podatkovne baze in skrbita za objekten pogled na podatke v bazi. Ker implementacija ni vezana na specifično podatkovno bazo, deluje na katerikoli podatkovni bazi, za katero je implementiran *JDBC Driver*. Na voljo pod licenco *Eclipse Public License 1.0*.

Commons-io 2.1 je knjižnica, ki implementira najpogostejše operacije pri manipulaciji z datotekami in tokovi. Uporabljena za manipulacijo imen datotek za shranjevanje in nalaganje modela iz datotek. Na voljo pod licenco *The Apache Software License 2.0*.

Joda-time 2.0 je knjižnica, ki implementira objektno definicijo časovnih komponent, kot so ure in datumi, ter njihovo manipulacijo. Odpravi marsikatero pomanjkljivost standardne javanske implementacije *java.util.Date*. Poenostavi in pohitri manipulacijo časovnih objektov. Na voljo pod licenco *The Apache Software License 2.0*.

Xstream 1.2.2 je knjižnica za serializacijo javanskih objektov v XML format. Uporabljena je za serializacijo naročil in korakov plana pred vnosom v podatkovno bazo. Na voljo pod licenco *BSD*.

Kxml 2.2.2 je knjižnica s hitro implementacijo XML zapisovalca, ki ga uporablja *Xstream*. Na voljo pod licenco *Public Domain*.

Jtds 1.2.4 je knjižnica, ki implementira *JDBC Driver* za podatkovno bazo *MS SQL*. Za razliko od uradne Microsoftove implementacije je odprtokodna in brez številnih hroščev. Na voljo pod licenco *LGPL*.

Junit 4.10 je knjižnica, ki omogoča enostavno testiranje kode. Ni del produkcijskega sistema, uporablja se le kot del razvojnega okolja. Na voljo pod licenco *Common Public License 1.0* .

Dodatek D

Formati datotek

Parametri datotek bodo opisani z naslednjimi simboli:

<*ime_parametra*:*tip*>	ime parametra in njegov tip
<*ime_parametra*:v_1|v_2>	ime parametra z možnimi vrednostmi v_1 in v_2
−||−	poljubno število definicij iz prejšnje vrstice
...	poljubno število ponovitev zadnjega parametra

Možni tipi parametrov so naslednji:

int	pozitivno celo število
zint	nenegativno celo število
float	pozitivno decimalno število
zfloat	nenegativno decimalno število
percentage	decimalno število med 0 in 1
word	beseda, sestavljena iz številk, črk in podčrtajev
date	datum v formatu dd.MM.yyyy
uuid	enkratno 128 bitno število

Format datoteke *.model*:

```
<ID_skladišča:uuid>
STATES
<št_stanj:int> <tip:NONE|FULL> <čas_prehoda_med_stanji:zfloat>
VERTICES
<ID:int> <ID_stanja:zint[1]> (<širina:int> <dolžina:int>)
```

```
-||-
EDGES
<ID:int> <ID_začetnega_križišča:int> <ID_končnega_križišča:int>
  <ID_stanja:int[1]> (<širina:int> <dolžina:int> <usmerjenost:UP|DOWN|
  LEFT|RIGHT>)
-||-
RACKS
<ID:int> <pozicija:word[2]> [<ID_mikrolokacije:word[3]>:<ID_cone:zint[4]>
  (<višina_police:int>),...[5]] (<širina:int> <dolžina:int> <višina:int>
  <stran_ceste:LEFT|RIGHT>)
-||-
RACK_GROUPS
<ID:int> [<ID_regala>,...] <ime_omejitve:word>
-||-
EXITS
<ID:int> <pozicija:word[2]> <tip:STORE|REMOVE> <kapaciteta:zint>
-||-
```

Format datoteke *.items*:

```
<ID_blaga:word> TYPE:<tip:word[6]> RACK_LOC_ID:
  <ID_police:word[3]> QUANTITY:<količina:float> IN_WAREHOUSE:
  <datum_prihoda:date> EXPIRE:<rok_porabe:date> WEIGHT:
  <teža_na_enoto:zfloat> [<parameter:word>:<vrednost:word>,...[7]]
-||-
```

Format datoteke *.paths*:

```
<začetek_poti:word[2]> <konec_poti:word[2]> [<ID_ceste:int>,...[8]]
  <dolžina_poti:float> <ID_stanja:zint[1]>
-||-
```

Format simulacijske datoteke:

```
MAXED_FILLED: <polnost_polic:percentage>
ORDERED: <urejenost:percentage[9]>
MEAN: <povprečno_število_izdelkov_na_polici:zfloat>
```

```
DEVIATION: <standardna_deviacija_od_povprečja:zfloat>
WORKERS
<ID:int> <tip:word[10]> MV_SPEED:<hitrost_pobiralca:float>
  POS:<pozicija:word[2]> RPR_DATA:(<širina:int>,<dolžina:int>)
  HL_SPEED:<hitrost_spuščanja_vilice:float>[11]
  HR_SPEED:<hitrost_dviganja_vilice:float>[11]
  CON_DOC:<število_predelov:int>[12]
  MAX_ITEMS:<maksimalna_količina_blaga:int>[12]
  RACK_LVL:<najvišja_polica_s_katere_pobira_pobiralec>[12]
-||-
ITEMS
<ID_blaga> WEIGHT:<teža_na_enoto:zfloat>
  EXPIRATION:<št_dni_pred_iztekom_roka_porabe:int>
  DISTRIBUTION:<distribucija_blaga_v_skladišču:percentage>[13]
ORDERS[14]
NEW_DAY[15]
Order[16]
ID: <ID_naročila:word>
Type: <tip_naročila:word>
Exit ID: <ID_izhoda:int>
<ID_izdelka> Quantity: <količina:float>
-||-
```

[1] če je ID stanja 0, potem objekt pripada privzetemu stanju

[2] format je E-<ID_ceste:int>[<odmik_od_začetnega_križišča:int>] ali V-<ID_križišča:int>

[3] če ID police ni celo število, je potrebno pretvoriti ID v celoštevilski ID z implementacijo vmesnika *IDTransformer*

[4] argument je opcijski, če ni prisoten je privzeta cona 0

[5] ID-ji polic si sledijo od najnižje do najvišje police

[6] sistem prepozna le tip SIMPLE_ITEM

[7] dodatni parametri so opcijski, [] definira definicijo blaga brez dodatnih parametrov

[8] v oglatih oklepajih sledijo po vrsti ID-ji vseh cest preko katerih poteka pot (tudi če poteka le po delu ceste)

[9] 0 je popolnoma kaotično skladišče, 1 popolnoma urejeno

[10] tipa pobiralca sta *FORKLIFT* ali *CART*

[11] potreben argument le pri pobiralcu tipa *FORKLIFT*

[12] potreben argument le pri pobiralcu tipa *CART*

[13] če se skupna distribucija ne sešteje v ena, bo sistem izvedel normalizacijo

[14] naročila so neobvezni del parametrov simulacije

[15] *NEW_DAY* označuje začetek naročil za naslednji dan, število dni je poljubno

[16] znotraj enega dneva je lahko poljubno število naročil