УНИВЕРЗИТЕТ У БЕОГРАДУ

ФАКУЛТЕТ ОРГАНИЗАЦИОНИХ НАУКА

Душан С. Савић

РАЗВОЈ СОФТВЕРА ЗАСНОВАН НА МОДЕЛУ СЛУЧАЈЕВА КОРИШЋЕЊА И MDD ПРИСТУПУ

докторска дисертација

Београд, 2016.

UNIVERSITY OF BELGRADE
FACULTY OF ORGANIZATIONAL SCIENCES

Dušan S. Savić

MODEL AND USE CASE DRIVEN SOFTWARE DEVELOPMENT PROCESS

Doctoral Dissertation

Belgrade, 2016

Ментор: **др Владан Девеџић, редовни професор**

Факултет организационих наука, Београд

____

Чланови комисије: **др Синиша Влајић, ванредни професор**

Факултет организационих наука, Београд

____

др Саша Д. Лазаревић, ванредни професор

Факултет организационих наука, Београд

____

др Драган Бојић, ванредни професор

Електротехнички факултет, Београд

____

PhD Alberto Manuel Rodrigues da Silva, associate professor

Department of Computer Science and Engineering of IST /UTL, Portugal

____

Датум одбране: _______________

Дисертацију посвећујем супрузи Марини и деци Софији, Јовану и Андреју

ПРЕДГОВОР

Управо сам погледао у сат... Сетио сам се нечег... Ево скоро па ћу постати пунолетан, да овде у Београду. Те 1998. године дошао сам у Београд из малог места, чувеног по вину... али мој тата је правио сјајну ракију... дуњевачу... Брат ме је сачекао, лако је када имате старијег брата... Али мени је брат много више од тога...

Захваљујем се **мами Миланки, тати Слободану** и **брату Христу** који су ми пружили могућност да нађем свој пут.

Али ништа није слутило да ћу остати у Београду... дошао сам из провинције... и ништа нисам знао... неки кажу да сам знао само да играм фудбал... Али онда сам на ФОН-у упознао професора Синишу Влајића... и од тада до данас траје наше искрено пријатељсво и поштовање, а надам се да ће потрајати још много година.

Захваљујем се професору **Синиши Влајићу** што ми је пружио прилику да будем део једног тима.

Да...тим... то смо ми чланови лабораторије за софтверско инжењерство. Захваљујем се професору **Саши Лазаревићу**, **Војиславу Станојевићу**, **Илији Антовићу**, **Милошу Милићу** на безрезервној помоћи свих ових година, а нарочито у периоду израде ове докторске дисертације.

Захваљујем се и **свим професорима, садашњим и бившим (и професорима који нажалост више нису са нама), асистентима, свим запосленима, студентима (и садашњим и бившим)** који су ме од првог дана прихватили као члана породице. Посебно велико хвала једном ...Eric Clapton-у са ФОН-а који не свира, које не пева, али обожава музику...

Да... породица.... за мене чаробна...смирујућа... увек на првом месту... пуна разумевања....

Захваљујем се супрузи **Марини** на разумевању и подршци коју ми свакодневно пружа..., а највише се захваљујем нашој деци **Софији**, **Јовану** и **Андреју** који су тати увек давали додатну снагу када му је била најпотребнија.

РАЗВОЈ СОФТВЕРА ЗАСНОВАН НА МОДЕЛУ СЛУЧАЈЕВА КОРИШЋЕЊА И MDD ПРИСТУПУ

РЕЗИМЕ

У докторској дисертацији је разматран проблем интеграције случајева коришћења у моделом вођени развој софтвера и предложена оригиналана *Silab-UCMDDM* метода.

Предложена *Silab-UCMDDM* метода истиче важност и неопходност коришћења 3 међусобно конзистентна и комплементарна модела: а) модела случајева коришћења, б) доменског модела и ц) модела прелаза стања. У дисертацији је идентификована директна веза између ова три модела која се пре свега огледа у томе да спецификација акција случаја коришћења треба да се ослања на доменски модел, док се предуслови и постуслови за извршење случајева коришћења дефинишу у моделу прелаза стања.

Silab-UCMDM метода користи две стратегије у развоју софтвера: а) стратегију засновану на случајевима коришћења (***Use Case Driven Development***) и б) стратегију засновану на ***MDD (Model Driven Development)*** приступу. Спецификација захтева у оквиру *Silab-UCMDM* методе омогућена је преко сопственог доменски специфичног језика (*UCDSL*).

Имплементација предложеног *UCDSL* језика извршена је преко *JetBrains MPS* алата за метапрограмирање *(JetBrains MPS metaprogramming system)*. *UCDSL* језик је интегрисан у оквиру SILAB-MDDTOOLSET алата који се може користити као додатак *(plugin)* за окружења као што су MPS и IntelliJ IDEA.

Евалуација предложене *Silab-UCMDD* методе урађена је на три начина:

1) Компаративном анализом предложене методе у односу на постојеће методе.

2) Приказом и анализом студијског примера који је развијен предложеном методом.

3) Анализом резултата теста у коме су учествовали студенти који су оцењивали предложену методу и *UCDSL* језик за спецификацију и валидацију захтева.

Silab-UCMDDM метода део је свеобухватног *Silab-MDD* приступа. У оквиру *Silab-MDD* приступа дефинисан је и начин интеграције *Структурне систем анализе,* којом се описује функционалност пословног система, са фазама прикупљања захтева и анализе у развоју софтвера. У том смислу креирани су сопствени доменски специфични језици помоћу којих се могу описати *дијаграми токова података (DFDDSL)* и *речник података (DataDDSL)*.

Правци будућег истраживања биће усмерени на унапређењу предложеног алата у смислу креирање посебне *Silab-MDD workbranch* платформе.

Кључне речи: развој вођен моделима (MDD), случајеви коришћења, доменско-специфични језик (DSL), спецификација захтева, трансформације

Научна област: Рачунарске науке

Ужа научна област: Софтверско инжењерство

УДК број: 0.004.4

MODEL AND USE CASE DRIVEN SOFTWARE DEVELOPMENT PROCESS

ABSTRACT

The thesis discusses the problem of integration of the Use Cases in the Model driven software development and proposes an original Silab-UCMDDM method.

The Silab-UCMDDM method emphasizes the importance and necessity of using 3 mutually consistent and complementary models: a) the use case model, b) the domain model and c) the state machine model. The thesis identifies a direct link between these three models which are primarily reflected in the fact that the specification of use case actions should be based on the domain model, while the preconditions and postconditions for executing use cases should be defined in the state machine model. This model state machine model is used for a clear and precise definition of use cases.

The Silab-UCMDM method uses two strategies in software development: a) a strategy based on the use cases (Use Case Driven Development) and b) a strategy based on MDD (Model Driven Development) approach. Requirements specification within Silab-UCMDM method is enabled via its own domain specific language (UCDSL).

The proposed UCDSL language was performed using the JetBrains MPS tool for metaprogramming (metaprogramming system JetBrains MPS). UCDSL language is integrated within the SILAB-MDDTOOLSET tool that can be used as an add-on (plugin) for environments such as MPS and IntelliJ IDEA.

The proposed Silab-UCMDD method was evaluated in three different ways:

1) By performing the comparative analysis of the proposed method and other existing methods.
2) By presenting and analyzing the case study that has been developed using proposed method.
3) By analyzing the results of the testing with students who evaluated the proposed method and UCDSL language for requirements specification and validation.

The Silab-UCMDDM method is a part of a comprehensive Silab-MDD approach. Silab-MDD approach defines the way of integration of the Structured System Analysis method, which describes the functionalities of the business system. In addition Silab-MDD defines the phases of requirements gathering and analysis in the software development lifecycle. Therefore, the Silab-MDD approach contains its own domain specific languages for specifying the Data Flow diagrams (DFDDSL) and the Data Dictionary (DataDDSL).

Further research will be focused on improving the proposed tools in terms of creating special Silab-MDD workbench platform.

САДРЖАЈ

Предговор 5
РАЗВОЈ СОФТВЕРА ЗАСНОВАН НА МОДЕЛУ СЛУЧАЈЕВА КОРИШЋЕЊА И MDD ПРИСТУПУ 6
Резиме 6
MODEL AND USE CASE DRIVEN SOFTWARE DEVELOPMENT PROCESS 8
Abstract 8
Поглавље 1. Увод 4
1.1. Предмет и циљ истраживања 4
1.2. Полазне хипотезе 8
1.3. Структура рада 8
Поглавље 2. Преглед области истраживања 12
2.1. Инжењерство софтверских захтева 12
2.2. Моделом вођени развој софтвера 16
2.2.1. Генеративни моделом-вођени развој софтвера 17
2.2.2. Интерпретативни моделом вођени развој софтвера 20
2.3. Случајеви коришћења 22
2.3.1. Сценарио случаја коришћења 23
2.3.2. Шаблони за спецификацију случајева коришћења 25
2.3.3. Случајеви коришћења 2.0 26
2.4. Преглед релевантних истраживања 28
2.4.1. Интеграција софтверских захтева у моделом вођени развој софтевра 28
2.4.2. Интегрисање случајева коришћења у MDD приступ 33
2.4.3. Патерни за спецификацију случајева коришћења 34
2.4.4. Трансформација модела захтева у моделе анализе 42
Поглавље 3. Silab-MDD приступ 46
3.1 Упрошћена Ларманова метода развоја софтвера 46
3.2 Трансформације између модела Silab-MDD приступа 51
3.3 О SILAB пројекту 64
Поглавље 4. Silab-UCMDM метода 66
4.1. Активности SILAB-UCMDDM методе 67
4.2. UCDSL – језик за спецификацију доменског модела, модела случаја коришћења и модела прелаза стања 71
4.2.1. Спецификација доменског модела помоћу UCDSL језика 71
4.2.2 Спецификација модела случаја коришћења помоћу UCDSL језика 79
4.2.3. Спецификација модела прелаза стања помоћу UCDSL језика 102
4.3. UCAppNDSL - језик за спецификацију прототипа апликације 105

4.4. DFDDSL- језик за спецификацију дијаграма тока података 109
4.5. DataDDSL - језик за специфицију речника података 119
4.5.1. Апстрактна синтакса 120
4.5.2. Конкретна синтакса 123
Поглавље 5. Silab-MDDToolSet алат 124
5.1. JetBrains Mps као алат за метапрограмирање 124
5.2. SILAB-MDDTOOLSET додатак за MPS 127
Поглавље 6.Евалуација 132
6.1. Анализа постојећих приступа и Silab-UCMDDM методе 132
6.2. Студијски пример 140
6.2.1. Вербални опис система 140
6.2.2. Спецификација модела 143
6.3. Пилот-тест евалуација 150
6.3.1. Пилот-тест задатак 151
6.3.2. Анализа Пилот-тест експеримента 152
Поглавље 7. Закључак 161
7.1. Остварени доприноси 164
7.2. Правци будућих истраживања 165
Поглавље 8. Литература 166
Поглавље 9. Списак слика 178
Поглавље 10. Списак табела 181
Поглавље 11. Додаци 182
Додатак А. Дефинисање кључних термина 182
Софтвер 182
Софтверски процес 182
Софтверско инжењерство 182
Захтев 183
Документ спецификације захтева 186
Модел 187
Моделовање 188
Метамодел 188
Мета-метамодел 189
Језик за моделовање 191
Синтакса језика за моделовање 191
Семантика језика за моделовање 192
Класификација језика за моделовање 192
Језици за спецификацију захтева 193
Трансформације модела 193

Типови трансформације модела ... 193
Додатак Б. Процес утврђивања софтверских захтева ... 195
Додатак В. Случајеви коришћења: Историјат и значајни аутори ... 199
Додатак Г. Шаблони за спецификацију случајева коришћења ... 202
Cockburn- ов шаблон за спецификацију случаја коришћења ... 202
RUP шаблон за спецификацију случаја коришћења ... 206
Биографија аутора ... 207
Изјава о ауторству ... 209
Изјава о истоветности штампане и електронске верзије докторског рада ... 210
Изјава о коришћењу ... 211

ПОГЛАВЉЕ 1. УВОД

У уводном делу описани су проблем, предмет и циљеви докторске дисертације. У складу са тиме дефинисане су опште и посебне хипозезе. На крају поглавља дата је структура докторске дисертације по поглављима.

1.1. ПРЕДМЕТ И ЦИЉ ИСТРАЖИВАЊА

Развој софтвера вођен моделом ***(Model Driven Development - MDD)*** [FRANKEL, D., (2002)] представља један од најсавременијих приступа у развоју софтвера. Према овом приступу предлаже се коришћење модела у свим фазама развоја софтвера. Дакле, идеја која овим приступом треба да се спроведе јесте коришћење модела директно у развоју софтвера, па се развој софтвера може посматрати кроз развој скупа модела и њихових трансформација. Један од основних циљева који се жели постићи јесте повећање продуктивности и скраћење времена потребно за имплементацију софтвера. Примена модела у развоју софтвера може позитивно да утиче на комплетан процес развоја софтвера јер омогућава лакше разумевање проблема односно система који се посматра. Основни принцип на коме се заснива моделом вођени развој софтвера гласи: *„Све је модел“* [BÉZIVIN, J. (2004)].

У почетним фазама развоја софтвера (фази прикупљања корисничких захтева) захтеви се обично описују у форми текста. То отежава трансформацију захтева у одговарајуће моделе анализе.

Различите методе које су засноване на принципима моделом вођеног развоја софтвера као на пример *Model Driven Architecture (MDA)* углавном акценат стављају на: а) трансфомацију *Platform Independent Model* (*PIM* модела) у *Platform Specific Model* (*PSM* модел) и б) трансформацију *PSM* модела у програмски код. Дефинисање одговарајућег *Computer Independent Modela* (модела захтева) и његова трансформација у одговарајући *PIM* модел углавном није посматрана као део целокупног процеса развоја софтвера. Модел захтева према [FRANKEL, D., (2002)] представља категорију логичких модела којим се специфицирају захтеви које треба да испуни будући софтверски систем. Ови модели се обично креирају на основу знања о пословном систему па су због тога блиски експертима из домена проблема. Према самој дефиницији развоја

софтвера вођеног моделом, идеалан развој софтвера би подразумевао креирање модела захтева и аутоматско генерисање софтверског система на основу модела. Међутим, ово често није могуће, а није ни једноставно јер постоји велики јаз између модела захтева и софтверског система.

У литератури се могу наћи појединачни покушаји трансформације модела захтева у моделе анализе, али не постоји добро дефинисана упутство о примени таквих решења у целокупни процес развоја софтвера [ZHANG L. & LANG W., (2008)]. Два су кључна проблема идентификована у моделом вођеном развоју софтвера:

1) Проблем дефинисања модела захтева (који се у *MDA* приступу може користити као *CIM* модел) и његове трансформације у модел анализе (који се у *MDA* приступу користи као *PIM* модел). Захтеви се најчешће описују у текстуалној форми услед чега се појављују следећи проблеми: а) захтеви нису између себе конзистентни; б) захтеви нису потпуни; в) захтеви нису јасни; г) тако описани захтеви нису погодни за трансформацију у различите моделе анализе; д) тешко је успоставити следљивост захтева кроз све фазе развоја софтвера. У случају када се захтеви описују у текстуалној форми аутори [LONIEWSKI et al., (2011)], предлажу коришћење *Aboott* хеуристике, при чему се захтеви ручно *(manual)* преводе у одговарајуће модел анализе [BRUEGGE, B. & DUTOIT, AH. (2004)], [LARMAN, C., (2004)]. *Loniewski* са својом групом аутора наглашава да је услед нејасне дефиниције *CIM* и *PIM* модела веома тешко приметити комплетан *MDA* приступ у развоју софтвера који управо почиње од дефинисања *CIM* модела [LONIEWSKI et al., (2011)]. Иако је највећа пажња у *MDA* приступу усмерена на трансформацију *PIM* модела у *CIM* модел, а посебно *CIM* модела у програмски код, постоји и покушаји трансформације *CIM* у *PIM* модел [JAMSHIDI, P. et al., (2009)] , [KHERRAF, S. et al., (2008)].

2) Проблем непостојања одговарајућег приступа који ће омогућити интеграцију процеса утврђивања софтверских захтева у комплетан моделом вођени развој софтвера. Иако је фаза прикупљања захтева једна од кључних фаза у процесу развоја софтвера, *Loniewski* са својом групом је показао да не постоји систематизован процес развоја софтвера који укључује ову фазу развоја софтвера у комплетан процес развоја софтвера

заснован на моделима. У последњих неколико година како у академској заједници тако и у индустријској пракси доста пажње је посвећено примени различитих приступа у моделовању захтева и њиховој интеграцији у моделом вођени развој софтвера . Тако су на пример, *Nicolas* и *Toval* у свом раду дали преглед постојећих приступа који се могу наћи у литератури, а који се односе на генерисање текстуалне спецификације захтева на основу модела [NICOLAS, J. & TOVAL, A (2009)]. У свом раду [G. LONIEWSKI, G. at al, 2010] је са групом аутора дао преглед и извршио анализу радова који су се бавили применом техника инжењеринга софтверских захтева у моделом вођени развој софтвера. Циљ овог истраживања је био да са утврди на који начин активности процеса утврђивања софтверских захтева могу бити интегрисане у моделом вођени развој софтвера и на који начин оне могу бити аутоматизоване. Већина традиционалних *MDD* приступа је вођена анализом *(analist-driven)* што означава да дефинисање иницијалног (почетног) *MDD* модела не представља део *MDD* приступа, већ овај модел настаје као резултат анализе претходно обављене спецификације захтева [ZIKRA et al., (2011)]. У свом раду *Zikra* је са групом аутора на основу спроведеног истраживања извршио анализу постојећих приступа који интегришу процес утврђивања захтева у моделом вођени развој софтвера. И поред тога што постоји уска веза између модела (уопштено) и захтева, аутори су дошли до закључка да мали број метода које су по свом карактеру моделом вођени интегришу модел захтева у комплетан процес развоја софтвера.

Са друге стране, методе развоја софтвера које су засноване на случајевима коришћења, предлажу коришћење случајева коришћења кроз све фазе развоја софтвера. У пракси је опште прихваћено да случајеви коришћења представљају веома важну технику за спецификацију функционалних захтева система. Случајеви коришћења су своју популарност између осталог добили што су кратки, добро структурирани, лаки за читање и што се најчешће документују природним језиком. Са друге стране, коришћење говорног језика само по себи доноси проблеме јер по својој природи говорни језик може да буде двосмислен. Како би се избегли проблеми у вези са случајевима коришћења потребно је дефинисати јасна и комплетна упутства, препоруке, стандарде које треба пратити и обрасце

које треба користити како би се ови проблеми смањили или потпуно елиминисали [FORBES, M. (2009)]. На тај начин се може утицати на квалитет спецификације захтева и на побољшање канала комуникације између различитих учесника у развоју софтвера. Интеграција случајева коришћења у *MDD* захтева детаљну и прецизну спецификацију случајаве коришћења, пре свега у делу који се односи на спецификацију акција сценарија случаја коришћења, као и предуслова и постуслова случајева коришћења.

Креирање доменског модела (најчешће описаног преко дијаграма класа или модела објеката и веза) на основу функционалних захтева једна је од активности која се јавља у скоро свим методама развоја софтера. У методама развоја софтвера које користе стратегију засновану на случајевима коришћења (а посебно објектно-оријентисаним методама), случајеви коришћења играју битну улогу за идентификовање класа и метода. Обично се у тим приступима случајеви коришћења користе заједно са осталим *UML* моделима [I. JACOBSON et al., 1992], [I. JACOBSON et al., 1999], [LARMAN, C. (2002)], али и поред тога не постоји добро утемељена техника која омогућава једноставну трансформацију модела класа на основу модела случајева коришћења [ANDA, B. et al. (2005)]. Предмет истраживања у овом раду јесте интегрисање случајева коришћења у моделом вођени развој софтевра.

Циљ овог рада јесте дефинисање методе која ће омогућити интеграцију случајева коришћења у моделом вођени развој софтвера и креирање одговарајућег језика за спецификацију случајева коришћења.

1.2. ПОЛАЗНЕ ХИПОТЕЗЕ

На основу анализе доступне литературе, на основу дефинисаног предмета и циља истраживања може се поставити општа хипотеза:

- Могуће је дефинисати методу која ће случајеве коришћења интегрисати у моделом вођени развој софтвера.

Поред опште хипотезе, могу се поставити и следеће посебне хипотезе:

- Могуће је дефинисати језик за ригорозну и детаљну спецификацију случајева коришћења.
- Могуће је специфицирати модел случајева коришћења у складу са доменским моделом.
- Могуће је на основу модела случаја коришћења дефинисати модел прототипа корисничке апликације.

1.3. СТРУКТУРА РАДА

Докторска дисертација је организована у 11 поглавља на следећи начин:

Прво поглавље представља увод у истраживање у којем је укратко описан проблем истраживања, предмет, циљеви, почетне хипотезе и дат је кратак опис садржаја докторске дисертације по поглављима.

Друго поглавље се односи на преглед стања из научне области истраживања. У том контексту, у овом поглављу, дат је приказ тренутно једног од најсавременијих приступа у *развоју софтвера који је вођен моделом* (***Model Driven Development - MDD***). У основи развоја софтвера помоћу *MDD* приступа су модели и њихова трансформација у програмски код. Основни циљ који се жели постићи овим приступом јесте да се повећа продуктивност и скрати време потребно за имплементацију и одржавање софтвера. Такође, у овом поглављу, описани су *случајеви коришћења* (***use-cases***) као једна од најзаступљенијих техника за спецификацију софтверских захтева. Случајеви коришћења су постали популарни јер су добро структурирани и описују се једноставним реченицама природног језика којима се прецизно дефинише редослед интеракције између корисника и софтверског система. Природни језици потенцијално могу да направе

проблеме при дефинисању случајева коришћења, због њихове двосмислености и непрецизности ако се пажљиво и јасно не користе. Наведени проблеми се могу избећи или свести на најмању могућу меру ако се дефинишу јасна упутства, препоруке, обрасци и стандарди код описивања случајева коришћења. Тиме се значајно може утицати на квалитет спецификације захтева и побољшање комуникације између различитих учесника у развоју софтвера.

Ово поглавље садржи и преглед релевантних истраживања која су се бавила: а) проблемом спецификације захтева засноване на случајевима коришћења и креирања других производа (артифаката) на основу случајева коришћења и б) проблемом интеграције процеса спецификације захтева и *MDD-a.* Интеграција случајева коришћења у *MDD* приступ захтева детаљну и прецизну спецификацију случајаве коришћења, пре свега у делу који се односи на спецификацију акција сценарија случаја коришћења, као и предуслова и постуслова случајева коришћења.

У **трећем поглављу** дат је приказ *Silab-MDD* (***Silab Model Driven Development***) приступа у развоју софтвера. Овај приступ је заснован на упрошћеној Лармановој методи развоја софтвера [Vlajić, S. (2015)]. *Silab-MDD* прати основне принципе моделом вођеног развоја софтвера. Артифакти *Silab-MDD* приступа јесту модели који се специфицирају преко посебно развијених доменско-специфичних језика који су објашњени у шестом погглављу.

Према овом приступу најважнији артифакт у развоју софтвера јесте модел. Модели се најчешће аутоматизованим трансформацијама преводе у друге моделе, семантички обогаћују појединим детаљима, да би се на крају аутоматски на основу њих генерисао програмски код за дату циљану имплементациону платформу. У овом поглављу дат је приказ основних трансформација које су подржане у оквиру *Silab-MDD* приступа.

У **четвртом поглављу** дат је приказ *Silab-UCMDM* методе за спецификацију захтева која користи две стратегије у развоју софтвера: а) стратегију засновану на случајевима коришћења (***Use Case Driven Development***) и б) стратегију засновану на ***MDD (Model Driven Development)*** приступу. Спецификација захтева у оквиру *Silab-UCMDM* методе омогућена је преко

посебно сопственог доменски специфичног језика (*UCDSL*). Помоћу *UCDSL* се описују три модела:

1. Доменски модел (***Domain Model - DM***) који представља поједностављену верзију UML дијаграма класа.
2. Модел случајева коришћења (***Use Case Model*** - ***UCM***) који служи за дефинисање и спецификацију случајева коришћења.
3. Модел прелаза стања (***State Transition Model*** - ***STM***) који служи за дефинисање дијаграма прелаза стања за сваки доменски објекат и дефинисање скупа случајева коришћења који се могу извршити над објектом у сваком од дефинисаних стања.

Наведени модели су међусобно конзистентни, што значи да се током ажурирања неког од модела непрекидно проверавају и усаглашавају концепти сва три модела.

У **петом поглављу** поглављу дат је приказ језика и алата који је развијен у оквиру ***JetBrains MPS*** алата, који се може користити као додатак *(plugin)* за *MPS*, као додатак за ***IntelliJ IDEA*** развојно окружење или самостално као одвојен алат.

У шестом поглављу извршена је евалуација предложене *Silab-UCMDD* методе на три различита начина:

1) Компаративном анализом предложене методе у односу на постојеће методе.
2) Приказом и анализом студијског примера који је развијен предложеном методом.
3) Анализом резултата пилот-тест евалуације, у којој су учествовали студенти завршне године основних академских студија Факултета организационих наука, смера за Информационе системе и технологије који су након експерименталног коришћења алата оцењивали предложени приступ, језик и алат..

Седмо поглавље овог рада је закључак у којем се сумира све што је урађено у истраживању, уз општи осврт на проблем и предмет истраживања, циљеве, хипотезе, постигнуте резултате и доприносе, као и резултате евалуације. На самом крају су дата и нека размишљања о будућим правцима истраживања у смислу развоја и примене предложеног приступа.

Осмо поглавље садржи приказ литературе која је коришћена у изради ове докторске дисертације.

Девето поглавље садржи листу слика које су приказане у дисертацији.

Десто поглавље садржи листу табела које су приказане у дисертацији.

Једанаесто поглавље садржи приказ најважнијих појмова који су коришћени у докторској дисертацији, као и нека шира објашњења за поједине концепте.

ПОГЛАВЉЕ 2. ПРЕГЛЕД ОБЛАСТИ ИСТРАЖИВАЊА

2.1. ИНЖЕЊЕРСТВО СОФТВЕРСКИХ ЗАХТЕВА

Софтверско инжењерство *(software engineering)* представља инжењерску дисциплину која је усмерена на систематичан приступ у развоју софтвера.

Софтверско инжењерство као инжњерска дисциплина почиње да се развија 60-тих година прошлог века као одговор на *„софтверску кризу“.* Тај период је познат по томе што велики број пројеката није успешно реализован, најчешће због прекорачења времена потребног за израду пројекта, али и трошкова који су неретко прбацивали буџет. Тада је увиђено да постојеће методе у развоју софтвера нису давале очекиване резултате и да је потребно развити нове методе које би осигурале да развој софтвера постане предвидљивији и ефикаснији. Од тада па до данас софтверско инжењерство се интензивно развија и представља грану која још увек није достигла своју зрелост.

Почетком 20. века *Software Engineering Coordinating Committee* група је почела са радом на SWEBOK (The Software Engineering Body of Knowledge) пројекту [SWEBOK, (2004)]. SWEBOK пројекат је у непрекидном развоју и његова последња верзија 3 је урађена 2014. године [SWEBOK - V3, (2010)]. Сврха SWEBOK пројекта јесте да дефинише софтверско инжењерство као научну дисциплину, која промовише конзистентан поглед на софтверско инжењерство широм света, да разјасни место и постави границу софтверског инжењерства у односу на друге научне дисциплине (као што су рачунарска наука, пројектни менаџмент, рачунарско инжењерство,...,итд.), као и да обезбеди основу за сертификацију и лиценцирање софтверских инжењера.

У литератури се могу наћи различите дефиниције *софтверског инжењерства*. Једна од најчешће цитираних дефиниција **софтверског инжењерства** јесте дефиниција коју је дао професор *Fritz Bauer* у извештају који је усвојен на NATO конференцији која је одржана 1969.године: *“Софтверско инжењерство је успостављање и коришћење здравих инжењерских принципима у циљу добијања економски оправданог софтвер који је поуздан и ради ефикасно на реалним машинама.”* [NAUR, P. & RANDELL, B. (1969)].

Софтверско инжењерство се према SWEBOK-u (верзији од 2004.године) дели на десет области знања *(knowledge area)* и то:

1. Софтверски захтеви *(Software requirements).*
2. Пројектовање софтвера *(Software design).*
3. Конструкција софтвера *(Software construction).*
4. Тестирања софтвера *(Software testing).*
5. Алати и методе софтверског инжењерства *(Sofware engineering tools and methods).*
6. Квалитет софтвера *(Software quality).*
7. Одржавање софтвера *(Software maintenance).*
8. Управљање софтверским конфигурацијама *(Software Configuration Managment).*
9. Управљања софтверским инжењерством *(Software configuartion Managment).*
10. Инжењерство софтверског процеса *(Software Engineering Process).*

Промене које су настале у верзији 3 SWEBOK пројекта обухватају ажурирање тема које су се налазиле у оквиру SWEBOK пројекта из 2004 године, укључивање нових тема у већ постојеће области софтверског инжењерства које су постале опште прихваћене од 2004 доа данас, избацивање тема које више нису релевантне, боља интеграција са сродним дисциплина и додавање пет нових области .

Нове области софтверског инжењерства су:

1. Основе рачунарства *(Computing Foundations)*
2. Основе инжењерства *(Engineering Foundations)*
3. Основе математике *(Mathematical Foundations)*
4. Економија софтверског инжењерства *(Software Engineering Economics)*
5. Професионална пракса софверског инжењерства *(Software Engineering Professional Practice)*

На основу приказа дефинисаних области софтверског инжењерства може се закључити да софтверско инжењерство представља инжењерску дисциплину која се бави проучавањем свих аспеката производње софтвера.

Према новој верзији SWEBOK-a, области софтверског инжењерства су организоване у две групе и то:

1. **Области знања које карактеришу праксу софтерског инжењерства** *(Knowledge Areas Characterizing the Practice of Software Engineering)* у које спадају све области софтверског инжњењерстава које су дефинисане у SWEBOK водичу од 2004.године као и нова област Професионална пракса софверског инжењерства.
2. **Области знања које се односе на образовне захтеве за софтверским инжењерсвтом** *(Knowledge Areas Characterizing the Educational Requirements of Software Engineering)* у које спадају следеће области: Економија софтверског инжењерства, Основе рачунарства, Основе математике и Основе инжењерства.

Термин **инжењерство захтева** *(requirements engineering)* први пут је уведен од стране *Alford-a* приликом дефинисања SREM *(Software Requirements Engineering Method)* методе још далеке 1977. године. Од тог тренутка, овај термин почиње активно да се користи у литератури и пракси, и све чешће мења до тада активни термин анализа система *(systems analysis).* На самом почетку, овај термин је коришћен у контексту описа две кључне активности животног циклуса развоја софтвера: *анализе захтева* и *спецификације захтева*. Почетком 90-тих година прошлог века, овај термин постаје опште прихваћен термин који се односи на један од кључних процеса у животном циклусу развоја софтвера. Данас се термин *инжењерство захтева* користи како би означио различите вештине, процесе, методе, технике и алате који се користе у анализи и спецификацији захтева. У овој секцији биће дат приказ кључних термина, дефиниција и питања која су релевантна за ову докторску дисертацију а односе се на инжењерство захтева.

Инжењерство захтева се може посматрати и као дисциплина и као процес. Стога ће термин инжењерство захтева бити коришћен у контексту дисциплине, док ће се у контексту процеса користити термин утврђивање софтверских захтева.

Као дисциплина *инжењерство захтева* представља *"систематизован и дисциплинован приступ који се састоји од скупа практичних техника за*

откривања и управљање захтевима током животног циксклуса развоја софтвера" [WIEGERS, K. (2003]. Треба нагласити да инжењеринг захтева има мултидисциплинаран карактер јер је у вези са неколико других инжењерских дисциплина.

Процес утврђивања захтева представља итеративан и инкраменталан процес у коме најчешће учествују велики број различитих заинтересованих страна *(stakeholders)* чији је основни циљ да осигура:

- да се сви релевантни захтеви експлицитно искажу и разумеју
- да постоји довољан ниво сагласности свих заинтересованих страна о дефинисаним захтевима
- да су сви захтеви документовани и специфицирани у складу са дефинисаним правилима за документовање/спецификацију захтева и да су ови захтеви евидентирани у одговарајућем формату .

Различити аутори дефинишу различите активности у процесу утврђивања софтверских захтева. Тако на пример *Pohl* је идентификовао три кључне активности у оквиру процеса утврђивања софтверских захтева: 1) **активност откривања захтева** *(Elicitation)*, 2) **активност документовања захтева** *(Documentation)*. и 3) **активност преговарања** *(Negotiation)* [POHL, K., (2010)], док је *Somerville* идентификовао четири кључне активности: 1) **израда студије изводљивости**, 2) **откривање и анализа захтева**, 3) **спецификација захтева** и 4) в**алидација захтева** [SOMERVILLE, I., (2010)].

У *Поглавље 11. Додаци*, у секцији *Додатак Б. Процес утврђивања софтверских захтева* дат је детаљан приказ ових активности.

2.2. МОДЕЛОМ ВОЂЕНИ РАЗВОЈ СОФТЕВРА

Термин моделом вођени развој софтвера (*Model-Driven Development, MDD*) данас је широко у употреби, у литератури и пракси. Међутим, поред овог термина постоји још неколико термина који се користе у истом или сличном контексту, да означе систематичну примену модела у развоју софтвера. Неки од тих термина су: Моделом-вођени инжењеринг (*Model-Driven Engineering, MDE*), Моделом-вођен развој софтвера (*Model Driven Software Development, MDSD*) [SELIC, B. (2008)], Моделом-заснован инжењеринг (*Model-Based Engineering, MBE*), Моделом-заснован развој (*Model-Based Development, MBD*), као и Моделом-вођена архитектура (*Model Driven Architecture, MDA*). Као последица великог броја термина данас имамо већи број дефиниција за сваки од поменутих термина. Највећи проблем свакако јесте што не постоји јасна разлика (граница) између ових термина, јер различити аутори на различите начине дефинишу ове термине. Изузетак представља термин MDA који представља термин који је дефинисан и заштићен од стране OMG групе. Како софтверско инжењерство представља инжењерску дисциплину онда и термини који се користе у овој области морају да имају јасно и прецизно значење. У овој дистертацији ће се у остатку текста бити коришћен термин *моделом вођени развој софтевра* или краће *MDD*.

Термин моделом вођени инжењеринг (*Model Driven Engineering*) уведен је од стране *Stuart Kent-a* [KENT, S., (2002)]. Овај термин је тада уведен да означи једну идеју о систематичном коришћењу модела у развоју софтвера, односно креирању софтвера који је у потпуности вођен моделом. Модел тако преузима кључну улогу у развоју софтвера. Он се користи у свим фазама развоја софтвера као основа за анализу и спецификацију захтева, али и као основа за генерисање програмског кода. Тако модел са једне стране представља кориснички захтев, али и програмски код са друге стране.

Термин моделом вођени развој (*Model-driven Development*) се користи у контексту систематске примене концепта апстракције односно модела кроз све фазе развоја софтвера [KLEPPE et al., (2003)]. MDD приступ се према [STANDISH_GROUP] дефинише као *„итеративни приступ у развоју софтвера у коме модели представљају извор извршавања програма било интерпретацијом тог модела било генерисањем програмског кода на основу модела“*. Моделом

вођени развој софтвера представља приступ који се заснива на аутоматском генерисању софтверског производа (апликације) на основу модела [SELIC, 2003].

У развоју софтвера који је вођен моделима могу се уочити два различита приступа. Први приступ предлаже трансформацију модела са вишег нивоа апстракције на моделе на нижем нивоу апстракције и познат је као **генеративни моделом вођени развој софтвера**. За разлику од првог приступа, према другом приступу модели се интерпретирају у времену извршења од стране виртуелне машине *(execution engines)*. Овај други приступ је познат под називом **интерпретативни моделом вођени развој софтвера**.

2.2.1. ГЕНЕРАТИВНИ МОДЕЛОМ-ВОЂЕНИ РАЗВОЈ СОФТВЕРА

Генеративни моделом-вођени развој софтвера заснива се на трансформацији модела који се налазе на вишем нивоу апстракције на моделе на нижем нивоу апстракције и аутоматском генерисању програмског кода који може касније бити ручно мењан. Према овом приступу модел се пре свега користи за опис проблема. Типичан сценарио у развоју софтвера применом генеративног модела се састоји из неколико корака: 1) дефинисање одговарајућег мета-модела или коришћење постојећих мета-модела, 2) креирање модела који је „у складу“са дефинисаним мета-моделом, 3) креирање нових или коришћење постојећих правила трансформације, 4) креирање одговарајућих шаблона за генерисање извршивог програмског кода, 5) трансформацију модела у програмски код.

MDA представља најпознатију реализацију овог приступа и *de facto* стандард у овом приступу. Идеја о дефинисању оквира за развој софтвера у коме централно место заузима модел на високом нивоу апстракције покренута је од стране OMG групе 2001.године. MDA полази од добро познате и утемељене идеја да је потребно одвојити спецификацију система од његове имплементације. MDA разликује три модела система:

1. **Рачунарски независтан модел** (*Computation Independent Model - CIM)* представља модел система које је усмерен на окружење система и захтеве система, док су детаљи структуре система и начин процесирања скривени или недефинисани. CIM модел се још назива и модел домена *(domain model)*, модел анализе *(analysis model)* или пословни модел *(business*

model). Модел домена је дефинисан на такав начин да је разумљив доменским експертима који се баве доменом за који се дефинише овај модел. Претпоставља се да је примарни корисник овог модела доменски експерт који нема знања о моделима и артифактима који ће се користити за реализацију фунционалности захтева који су дефинисани у овом моделу. CIM модел има значајну улогу у превазилажењу јаза који постоји између доменских експерата и захтева система са једне стране и експерата за пројектовање и конструкцију софтвера са друге стране [MILLER, J. & MUKERJI, J. (2003)].

2. **Платформски независан модел** (*Platform Independent Model - PIM)* представља модел система који је независан од платформе. PIM модел описује (обезбеђује) спецификацију структуре и понашања система без навођења техничих детаља. Овај модел представља модел анализе система. Модел анализе описује које функционалности систем треба да обезбеди (које функционалности) и има за циљ да буде јасан и недвосмислен, коректан и конзистентан. У објектно-оријентисаном развоју софтвера обично се овај модел креира на основу спецификације корисничких захтева, и приказује преко различитих UML дијаграма. Поред UML-а могу се користити и други језици за моделовање.

3. **Платформски специфичан модел** *(Platform Specific Model - PSM)* представља модел система који проширује PIM модел додајући детаље који су специфични за софтверску (програмски језик, оперативни систем) и/или хардверску платформу. Овај модел представља модел пројектовања.

Поред модела, значајно место у овом приступи припада и трансформацијама које моделе на вишем нивоу апстракције трансформишу у моделе на нижем нивоау апстракције: одговарајући CIM модели се трансформишу у PIM моделе, PIM модели у PSM моделе, док се на основу PSM модела генерише програмски код. Шире гледано, трансформације се могу дефинисати између било која два MDA модела, а не само модела на различитом нивоу апстракције. Платформа се дефинише као скуп подсистема и технологија које обезђује кохерентан скуп функционалности преко интерфејса, где свака апликација која је подржана од стране те платформе може да користи те функционалности без знања

о детаљима како је та функционалност имплементирана од стране платформе [OMG]. Модерне извршиве платформе хијерархијски су приказане ниже на слици (Слика 1).

Слика 1. Хијерархијски приказ модерних извршивих платформи

MDA се базира на низу спецификација познатих под називом OMG стандарди: Meta Object Facility (MOF)[1], Unified Modeling Languange (UML)[2], Object Constraint Language (OCL)[3], XML Metadata Interchange (XMI), Common Warehouse Metamodel (CWM)[4] итд.

Поред MDA као једног од представника генеративног моделом-вођеног развоја софтвера који се заснива на генерисању програмског кода углавном на основу UML модела, постоје и други представници овог правца. Програмски код се може генерисати на основу спецификације која може бити писана у неком од текстуалних или графичких доменски специфичних језика.

---

[1] OMG-MOF. Object Management Group - Meta Object Facility (MOF) Core Specification, v2.4.2.; 2014, веб адреса: http://www.omg.org/mof/ .

[2] OMG-UML. United Modeling Language Infrastructure Specification, Version 2.4.1; 2011, веб адреса: http://www.uml.org/

[3] OMG-OCL. Object Constraint Language (OCL), v2.4; 2014. веб адреса: http://www.omg.org/spec/OCL/

[4] OMG-CWM: Object Management Group – Common warehouse metamodel (CWM); 2003, веб адреса: http://www.omg.org/spec/cwm/ .

2.2.2. ИНТЕРПРЕТАТИВНИ МОДЕЛОМ ВОЂЕНИ РАЗВОЈ СОФТВЕРА

Уколико се интерпретативни моделом-вођени развој софтвера посматра у контексту MDA, тада се скуп PIM модела директно интерпретира уместо да се трансформише у PSM моделе и програмски код. Основна карактеристика овог приступа јесте да се модели који се креирају директно извршавају. Један од најпознатијих представника овог приступа јесте извршиви UML (Executable UML) [MILICEV, D., (2009)], [MELLOR, S.J. & BALCER, M.J. (2002)]. Предност овог приступа се огледа у томе што се промене у моделу одмах виде, па систем може брзо на тај начин да се мења и прилагоди потребама крајњих корисника. Предности генеративног приступе је што се брже извршава и има бољу превенцију грешака [MEIJLER. T. et al. (2010)].

Поред модела, језика и трансформација у моделом вођеном развоју софтвера значајно место заузимају и алати за мета-моделовање. Ови алати се користе како за дефинисање модела, тако и за дефинисања мета-модела односно језика за мета-моделовање. Ниже су дати [WEB-MoLa][5]:

- *Алати који прате генеративни моделом вођени развој софтвера* како што су:
    - RUX Tool (http://www.homeria.com/rux-tool),
    - WebRatio (http://www.webratio.com),
    - WebML (http://www.webml.org),
    - EMF (http://www.eclipse.org/modeling/emf),
    - OpenEDGE (http://web.progress.com),
    - GMF (http://www.eclipse.org/modeling/gmp),
    - AndroMDA (http://www.andromda.org),
    - PathMATE (http://www.pathfindermda.com),
    - IBM Rational Rhapsody (http:/ibm.com/software/awdtools/rhapsody),
    - iQgen (http://www.innoq.com/iqgen) ,
    - OpenMDX (http://www.openmdx.org),
    - smartGENERATOR (http://www.bitplan.com),

[5] [WEB-MoLa] http://modeling-languages.com/

  - Mendix (http://www.mendix.com),
  - MetaEdit+ (http://www.metacase.com)
- *Алати који прате интерпретативни моделом вођени развој софтвера* као што су:
  - AlphaSimple (http://alphasimple.com),
  - Executable UML – fUML (http://www.omg.org/spec/FUML),
  - Alf (http://www.omg.org/spec/ALF/Current)
- *Алати који прате хибридни моделом вођен развој софтвера* као што су:
  - Abstract Solutions Tools (http://www.kc.com/PRODUCTS)
  - Bridge Point (http://www.mentor.com/products/sm)
  - OOA Tool (http://ooatool.com/OOATool.html)

2.3. СЛУЧАЈЕВИ КОРИШЋЕЊА

У пракси је опште прихваћено да случајеви коришћења представљају веома важну технику за спецификацију функционалних захтева система. Случајеви коришћења су своју популарност између осталог добили што су кратки, добро структурирани, лаки за читање и што се најчешће документују природним језиком. Са друге стране, коришћење говорног језика само по себи доноси проблеме јер по својој природи говорни језик може да буде двосмислен. Како би се избегли проблеми у вези са случајевима коришћења потребно је дефинисати јасна и комплетна упутства, препоруке, стандарде које треба пратити и обрасце које треба користити како би се ови проблеми смањили или потпуно елиминисали [FORBES, M. (2009)]. На тај начин се може утицати на квалитет спецификације захтева и на побољшање канала комуникације између различитих учесника у развоју софтвера.

Случајеви коришћења се најчешће користе и посматрају у контексту јединственог језика за моделовање и поред тога што су се појавили пре њега. Случајеви коришћења представљају технику за откривање и спецификацију функционалних захтева система засновану на сценарију, па се као такви користи у оквиру различитих метода развој софтвера као што су Ларманова метода развоја софтвера [LARMAN, C. (2002)], [LARMAN, C. (2005)], Јединствени процес (*Unified Process)* [KRUCHTEN, P. (2004)] и *Iconix Process* [ROSENBERG, D. (2004)].

Случај коришћења се дефинише као *„секвенца трансакција у систему чији је задатак да донесе неку мерљиву вредност актору система"* [JACOBSON, I. et al. (1995)], али и као *„спецификација секвенци акција (укључујући и варијације алтернативне акције) који систем може да изврши у интеракцији са акторима“* [WEB-UML (2011)].

Cockburn на основу анализе постојећих дефиниција случаја коришћења датих од стране различитих експерата (18 различитих дефиниција) даје своју дефиницију случаја коришћења на следећи начин: „*Случај коришћења је скуп могућих секвенци интеракција између система који се посматра и његових корисника (или актора), која је усмерена ка одређеном циљу. Колекција случајева коришћења треба да дефинише сва понашања система која се односе на акторе и да их увери да ће њихови циљеви бити задовољени. Свако понашање система*

које је ирелевантно за актора не би требало да буду укључено у случајеве коришћења“ [COCKBURN, A. (1995)].

Случајеви коришћења представљају груписан скуп сценарија (једног главног сценарија, и/или више алтернативних сценарија, и/или више сценарија изузетка). У складу са тим, *Rumbaugh* са групом аутора случај коришћења дефинише као *„спецификацију секвенце акција, укључујучи алтернативне акције и акције које доводе до грешке, коју систем или подсистем могу да изврше у интеракцији са спољним актором у циљу постизања циља “* [RUMBAUGH, J. E. et al. (2005)]

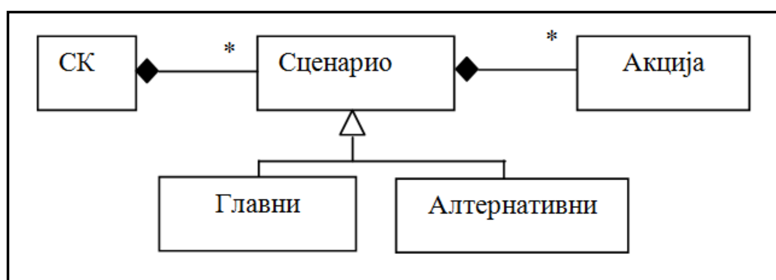
Једну од општијих дефиниција случаја коришћења дао је *Sommerville* који каже да *„случај коришћења представља технику за откривање корисничких захтева која је заснована на сценарију“* [I.SOMMERVILLE (2006)].

У поглављу *Додатак В. Случајеви коришћења: Историјат и значајни аутори* дат је преглед аутора који су најзаслужнији за широку примену случајева коришћења као технике за спецификацију корисничких захтева у оквиру различитих метода развоја софтвера.

2.3.1. СЦЕНАРИО СЛУЧАЈА КОРИШЋЕЊА

Сценарио случаја коришћења описује једно жељено коришћење система од стране актора, односно система из перспективе корисника. Стога, сценарио чине акције којима се дефинише интеракција између актора и система. Сценарио представља један могући пут кроз случај коришћења, па стога и извршење једног сценарија случаја коришћења не укључује извршење свих акција случаја коришћења.

Случај коришћења има један главни и један или више алтернативних сценарија. Сценарио је описан преко: а) секвенце акција и б) интеракција између актора и система. СК се састоји из главног и алтернативних сценарија (Слика 2) .



Slika 2. Однос случаја коришћења и сценарија

Неки аутори, поред ова два типа сценарија дефинишу и сценарио изузетка као посебни тип сценарија случаја коришћења. Овим сценариом се дефинише на који начин систем реагује на грешке које се десе у главном сценарију, алтернативном или у неком другом сценарију изузетка [I.SOMMERVILLE (2006)]. *Alexander* разликује 4 типа сценарија: 1) нормални односно главни сценарио *(Normal Case Scenario)*, 2) алтернативни сценарио *(Alternative Case Scenario),* 3) Сценарио изузетка *(Exception Cases)* и 4) "шта-ако" сценарио *(What-If Scenarios)* [ALEXANDER, I. & MAIDEN, N. (2004)].

Интеракција између корисника и система се може описати на различите начине, у различитој форми односно облику. *Rebecca Wirfs-Brock* разликује три форме за опис ове интеракције:

- Прва форма је у облику приче *(narrative form).* Основна карактеристика ове форме јесте да се интеракција описује у форми слободног текста, обично у једном параграфу. На овај начин се истичу намере корисника приликом извршења случаја коришћења. Интеракција у овој форми се специфицира на високом нивоу апстракције без укључивања детаља. Приликом спецификације интеракције користе се термини из домена проблема.

- Друга форма је у облику сценариа *(scenario form).* Сценарио се састоји од скупа корака, при чему се сваки корак представља декларативан исказ без гранања.

- Трећа форма је форма у облику дијалога *(conversation form).*

2.3.2. ШАБЛОНИ ЗА СПЕЦИФИКАЦИЈУ СЛУЧАЈЕВА КОРИШЋЕЊА

На популарност случајева коришћења као технике за спецификацију захтева доста су утицали аутори односно стручњаци из праксе који су писали о практичној примени случајева коришћења на релним пројектима. Тако су различити аутори развили сопствене шаблоне односно стилове за спецификацију захтева путем случајева коришћења. Потреба за дефинисањем шаблона настала је услед потребе да се осигура комплетност спецификације случаја коришћења, али и да се на јединствени и конзистентан начин прикажу случајеви коришћења различитим заинтересованим странама које учествују у пројекту. У пракси два шаблона за спецификацију случајева коришћења су се издвојила као најчешће коришћена и то: 1) *Cockburn-ов* шаблон [COCKBURN, A. (2001)] и 2) шаблон дефинисан у оквиру Јединственог процеса развоја софтвера) [JACOBSON, I. et al.(1999)]. Поред ова два шаблона значајно место заузимају и шаблони (стилови) које су дефинисали:

- *Alistair Cockburn* и *Ivar Jacobsson* који су предложили такозвани контекстуални опис случаја коришћења кога карактерише усмереност ка циљу који корисник жели да оствари. Овим приступом се предлаже одвајање спецификације пословних правила и корисничког интерфејса од спецификације случаја коришћења, али уз задржавање контекстуалне везе између њих. Такође, према овом приступу тежи се спецификацији свих могућих сценарија у оквиру једног случаја коришћења.

- Према приступу кога заступа *Martin Fowler* случај коришћења се специфицира кроз један сценарио без разматрања алтернативних сценарија и сценарија која могу настати услед грешака у извршењу акција основног или неког другог алтернативног сценарија. Алтернативна сценарија се према овом приступу издвајају и специфицирају као посебни случајеви коришћења. Основни недостатак оваквог приступа у спецификацији случаја коришћења јесте велика робусност и велика редунданса.

- Према приступу који заступају *Kurt Bittner* и *Ian Spencer* спецификација случајева коришћења подразумева детаљну и свеобухватну спецификацију случаја коришћења која укључује спецификацију свих сценарија са елементима пословних правила и прототипом корисничког интерфејса.

Према *Cockburn-у* ниједан шаблон за спецификацију случаја коришћења није погрешан, само је у појединим ситуацијама више или мање прикладан [COCKBURN, A. (2000)].

У поглављу *Додатак Г. Шаблони за спецификацију случајева коришћења* дат је приказ два најпопуларнија шаблона за спецификацију случајева коришћења.

2.3.3. СЛУЧАЈЕВИ КОРИШЋЕЊА 2.0

Ivar Jacobsson, творац случајева коришћења и један од водећих стручњака у области примене случајева коришћења заједно са групом аутроа 2011. године издао је „*Приручник за успешан рад са случајевима коришћења*“ *(Use Case 2.0 – The Guide to succeeding with Use Cases*[6]*)*. У овом приручнику *Ivar Jacobsson* са групом аутора даје приказ 6 основниих принципа којих се треба придржавати за успешну примену случајева коришћења:

1) Приче требају бити једноставне *(„Keep it simple by telling stories“).* Приповедање је један од најједноставнијих и најефикаснијих начина да се пренесе знање са једне особе на другу. То је један од најбољих начина комуникације којим се утврђује шта систем треба да ради и омогућава свим заинтересованим странама да се фокусирају на исти циљ. Према овом принципу случајеве коришћења (који представљају једну причу) треба писати тако да буду јасни, разумљиви *(actionable)* и да могу бити тестирани *(testable)*.
2) Разумети велику слику (*„Understand the big picture“*). Приликом развоја софтверског система много је важно упознати се са комплетним системом, упознати систем у целини, односно јасно дефинисати границе система. Уколико границе система нису јасно дефинисане тешко је одредити потребно време за реализацију софтверског система, тешко је утврдити шта припада систему, а ште не, као и ко и које користи има од таквог система. Због тога је потребно разумети систем у целини (*„Understand the big picture“*) и за приказ границе система користити UML дијаграм случајева коришћења.

[6] https://www.ivarjacobson.com/publications/white-papers/use-case-ebook

3) Усмеравање пажње на вредност („*Focus on value“).* Према овом принципу при специфкацији случајева коришћења пажња треба бити усмерена на вредност коју корисник добија коришћењем система. Овај принцип истиче да је усмеравање пажње на то како ће систем бити коришћен од стране корисника система значајније него дефинисање скупа функција које систем треба обезбеди.

4) Развој софтверског система у деловима („*Build the system in slices“).* Развој софтверског система може бити дуг процес. Један од чинилаца који утиче на дужину развоја софтверског система свакако представљају сложеност функционалних и нефункционалних захтева. Често се у прошлости, а неретко и данас, прави грешка да се комплетан развој софтверског система изврши кроз један пролазак, односно да се најпре дефинишу сви захтеви, а након тога крене у њихову имплементацију. Према овом принципу, систем треба развијати кроз делове *(slices)* при чему након завршетка сваког дела корисник добија софтверски систем који за њега има јасну вредност (корисник зна јасно шта је софтверским системом добио, односно која је његова вредност, шта је то што му софтвер пружа).

5) Испорука софтверског система у инкрементима *(„Deliver the system in increments“).* Овај принцип је директно повезан са предходним принципом, а односи се на испорука софтверског система након завршетка сваког дела *(slices)*.

6) Прилагодити случајаве коришћења тако да задовоље потребе тимова *(„Adapt to meet the team’s needs“)*. У развоју софтвера треба водити рачуна о специфичности сваког система за који се креира софтвер и захтевима који постављају различите заинтересоване стране. Ово утиче да захтеви буду дефинисани различитим стиловима, са различитим нивоом детаља, а што захтева да случајеви коришћења буду дефинисани тако да испуњавају тренутне потребе различитих тимова.

2.4. ПРЕГЛЕД РЕЛЕВАНТНИХ ИСТРАЖИВАЊА

2.4.1. ИНТЕГРАЦИЈА СОФТВЕРСКИХ ЗАХТЕВА У МОДЕЛОМ ВОЂЕНИ РАЗВОЈ СОФТЕВРА

Примена модела у развоју софтвера може позитивно да утиче на комплетан процес развоја софтвера јер омогућава лакше разумевање проблема односно система који се посматра. Основни принцип на коме се заснива моделом-вођени развој софтвера гласи: све је модел [BEZIVIN, J. (2004)] . Дакле, сви артифакти који настају у процесу развоја софтвера представљају модел. Стога, овај приступ се заснива:

- на подизању нивоа апстракције,
- на примени доменски специфичних језика у развоју софтвера,
- на примени техника за трансформацију модела и
- аутоматском генерисању програмског кода.

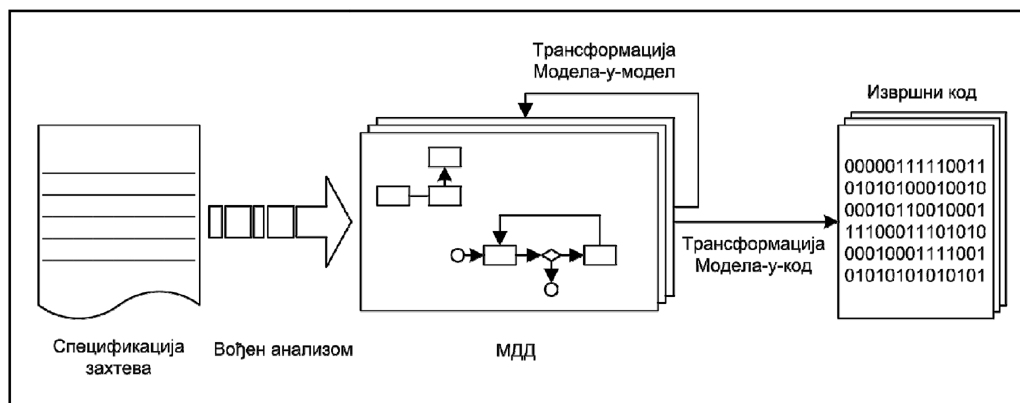
У последњих неколико година како у академској заједници тако и у индустријској пракси доста пажње је посвећено примени различитих приступа у моделовању захтева и њиховој интеграцији у моделом вођени развој софтвера. Ниже је дат преглед радова који су се односили на решавање проблема интеграције процеса утврђивања захтева у моделом вођени развој софтвера.

У свом раду *Loniewski* је са групом аутора [LONIEWSKI et al., 2010] дао преглед и извршио анализу радова који су се бавили применом техника инжењеринга софтверских захтева у моделом вођени развој софтвера. Циљ овог истраживања је био да се утврди на који начин активности процеса утврђивања софтверских захтева могу бити интегрисане у моделом вођени развој софтвера и на који начин оне могу бити аутоматизоване. У ову анализу на основу селектованих 877 радова који су објављени у последњих 10 година изабрано је 65 радова на основу којих се базира ово истраживање. Према *Loniewski-у* [LONIEWSKI et al., 2010] у литератури се до тада нису могли наћи радови који приказују преглед постојећих приступа у примени модела захтева у моделом-вођеном развоју софтвера. Са друге стране, познато је било истраживање о анализи постојећих приступа који користе моделе за генерисање текстуалних захтева. Комплетан

списак радова који је обухваћен овом анализом доступан је на линку: *www.dsic.upv.es/~einsfran/review-remdd.htm.*

Након анализе резултата истраживања аутори су извукли следеће закључке: 1) модели нису тако често коришћени као што су очекивали у фази прикупљања захтева у MDD приступу (само 64%), што је било супротно њиховим очекивањима, 2) спецификација захтева природним језиком је веома важна, 3) након спецификације захтева њихова следљивост није добро дефинисана, 4) не постоји потпуни приступи (добро документовани и потврђени у пракси) који интегришу процес утврђивања захтева у моделом-вођени развој софтвера. Такође, аутори су нагласили да не постоји много емпиријских студија које потврђују користи интегрисања техника инжењеринга захтева у моделом вођени развој софтвера.

Већина традиционалних MDD приступа је вођена анализом *(analist-driven)* што означава да дефинисање иницијалног (почетног) MDD модела не представља део MDD приступа, већ овај модел настаје као резултат анализе претходно обављене спецификације захтева [ZIKRA, J. et al., (2011)].



Слика 3. MDD процес [ZIKRA, J. et al., (2011)].

У свом раду *Zikra* је са групом аутора на основу спроведеног истраживања извршио анализу постојећих приступа који интегришу процес утврђивања захтева у моделом вођени развој софтвера. И поред тога што постоји уска веза између модела и захтева аутори су дошли до закључка да мали број метода које су по свом карактеру моделом вођене, интегришу модел захтева у комплетан процес развоја софтвера. На основу анализе аутори су дефинисали скуп својстава о

којима треба водити рачуна приликом интеграције процеса утврђивања корисничких захтева у моделом вођени развој софтвера. Могући приступи у интеграцији процеса утврђивања захтева са *MDD-ом* су подељени су у три групе:

1. У прву групу спадају приступи који на основу текстуалне спецификације која је дефинисана у форми природног текста (у форми приче) техникама процесирања природног (говорног) језика *(Natural Language Processing, NLP) генеришу иницијални MDD модел*.

2. У другу групу спадају приступи код којих се *модел захтева посматра као иницијални MDD модел.* За креирање иницијалног MDD модела креирају се одговарајућа упутства. У оквиру ове групе аутор разликује три варијанте:

   a) *Приступи на ивици интеграције (Edge integration approaches).У* приступима овог типа дефинишу се препоруке и правила како се на основу тесктуално дефинисаних захтева креира иницијални MDD модел без улажења у детаље како се касније тај модел може користити у моделом вођеном развоју софтвера. За креирање иницијалног модела највише приступа користи UML, језике сличне њему као што је SysML, чиме наглашавају да се тиме олакашава интеграција са MDD приступима који се заснивају на UML-у *(UML-based MDD approaches). Soares* и *Vrancken* [SOARES, M. S. & VRANCKEN, J. L. M. (2008)] предлажу креирање модел захтева интеграцијом SysML-а модела и модела случајева коришћења. *Jørgensen et al.* предлажу извршиве случајеве коришћења *(Executable Use Cases)* који се могу користити током свих фаза MDD процеса [JØRGENSEN, J. B. et al., (2009)]. *Lmendros-Jiménez* и *Iribarne* [ALMENDROS-JIMÉNEZ, J. M. & IRIBARNE, L. (2004)] на основу случајева коришћења генеришу UML дијаграм активности који се касније користи за генерисање графичког корисничког интерфејса.

   b) *Приступи засновани на парцијалној интеграцији* (*Partial integration approaches)* обезбеђују упутства за креирање модела захтева и његовој трансформацији у иницијални MDD модел. На основу резултата истраживања, аутор је дошао до закључка да се за парцијалну интеграцију најчешће користе: а) i* модели [ALENCAR, F. et al., (2009)], [LUCENA, M. et al., (2009)], [MARTÍNEZ, A. et al. (2003)], [MAZÓN, J.N. et al.,

(2007)], б) случајеви коришћења [CYSNEIROS, L. & do PRADO LEITE, J.C.S. (2004)], [DEBNATH, N. et al. (2008)], [FATWANTO, A. & BOUGHTON, C. (2008)] и ц) UML [BIFFL, S. et al. (2007)], [GUELFI, N. & PERROUIN, G. (2007)], [KOCH, N. et al. (2006)], [SÁNCHEZ, P. et al. (2010)]. Аутори поред тога истичу *OO-Method* [PASTOR, O. et al. (2001)] који представља један од репрезентативних приступа овог типа који користи модел *„анализе комуникације“* [PASTOR, O. et al. (2011)] . *Kalnins* је са групом аутора развио језик за спецификацију захтева *(Requirements Specification Language)* на основу кога се применом дефинисаних трансформација креира UML дијаграм класа и секвенцни дијаграм. Секвенцни дијаграми се касније користе за генерисање програмског кода. Све трансформације су имплементиране коришћењам MOLA језика за трансформације. [KALNINS, A. et al., (2010)].

c) **Приступи засновани на тоталној интеграцији** *(Total integration approaches)***,** приступи овог типа нуде решење у виду комплетне интеграције модела пословног процеса у моделом вођени развој софтвера [SILVA, A.R. et al. (2007)], [NAVARRO, E. (2007)].

3. У трећу групу спадају приступи који користе концепт следљивости *(traceability)* како би означили везу између два модела. Дакле, ови приступи користе концепт везе *(trace)* како би креирали следљивост између модела захтева и иницијалног MDD модел. Треба нагласити да у моделом вођеном развоју софтвера ова веза не означава само везу која приказује да је један модел у вези са другим (следи га или се односи на њега) већ ова веза представља и семантичку везу између два модела.

У свом раду *Assar* [ASSAR, S. (2012)] је извршио анализу радова који су објављени на MoDRE конференцији у периоду од 2011. до 2013. године (*MoDRE 2011, MoDRE 2012 и MoDRE 2013*), а који се односе на примену доменски специфичних језика, модела и трансформација у процесу утврђивања софтверских захтева (интеграцији MDE и RE). Анализа је обухватала свих 29 радова без претходне селекције и имала је за циљ да одговори на три кључна питања:

1. Којом групом проблема су се аутори бавили у својим радовима (*research issue targeted)*. На основу сличних истраживања [LONIEWSKI et al., 2010], [ZIKRA et al., 2011] и на основу анализе радова који су се бавили истим проблемом [NUSEIBEH, B. & EASTERBROOK, S. (2000)], [CHENG, B. H. C. & ATLEE, J. M. (2007)] *Assar* је креирао 6 група односно класа проблема и анализирао радове према тим групама :

    a. проблеми која се односе на процес откривања захтева као прву активност у процесу утврђивања софтверских захтева *(Elicitation)*

    b. проблеми која се односе на начин представљања захтева односно дефинисања погодне нотације за спецификацију захтева *(Representation).*

    c. проблеми који се односе на верификацију и валидацију захтева *(Verification and validation)*

    d. проблеми који се односе на следљивост *(Traceability)*

    e. проблеми који се односе на креирање иницијалног, односно почетног модела *(Derivation)*

    f. проблеми који се односе на интегрални процес утврђивања софтверских захтева

2. Које технике, алате, методе су аутори развијали и користили за решавање претходно поменутих проблема и које се означили као доприносе у својим радовима *(research contribution)*? Ове доприносе *Assar* је класификовао у 6 група:

    a. дефинисање новог језика (нотације) за спецификацију захтева

    b. дефинисање алата који у потпуности или делимично подржава предложени приступ

    c. предлог нове методе или новог приступа

    d. дефинисање технике за трансформацију модела

    e. дефинисање нове или прилагођавање постојећих техника за успостављање следљивости између различитих артифаката

   f. анализу и критички став према постојећим приступима који постоје у теорији и/или пракси.

3. На који начин је вршена евалуација у предложеним приступима *(evaluation method)*? Према овом критеријуму радови су класификовани према томе да ли је евалуација вршена: а) на основу примене предложеног приступа у пракси, у компанијама које се баве развојем софтвера, 2) на основу контролисаног експеримента и 3) на основу неке илустрације, односно неког студијског примера.

На основу спроведене анализе *Assar* је извео неколико закључака:

- У односу на типове захтева (функционалне и не-функционалне) од 29 радова која су анализирана, њих 21 се односио на функционалне захтеве, 5 на не-функционалне захтеве, док су се у 3 рада аутори бавили проблемима који су се односили и на функционалне и на не-функционалне захтева. Углавном су аутори писали о проблемима који су се односили на пословне апликације (16), док се 5 радова се односило на интегрисане *(embedded)* системе.

- Највећи број радова се односи на предлагања нових језика за моделовање (*RDAL, AoUCM , TADL2, RSL, RSL-IL, MoCKRE, SySML-sec*).

2.4.2. ИНТЕГРИСАЊЕ СЛУЧАЈЕВА КОРИШЋЕЊА У MDD ПРИСТУП

Hoffmann и *Lichter* су у свом раду *„Towards the Integration of UML – and textual Use Case Modeling“* [HOFFMANN, A. N. V. & LICHTER, H. (2009)] нагласили да се у процесу развоја софтвера који је заснован на UML-у, случајеви коришћења најпре идентификују и специфицирају путем UML дијаграма случаја коришћења, након чега следи њихова текстуална спецификација. Сходно томе, модел случајева коришћења се састоји из два дела: први, који дефинише случајеве коришћења и њихове везе, и други који чини текстуална спецификација случајева коришћења. У једном таквом развоју софтвера, потребно је обезбедити конзистентност између UML модела и текстуалног описа случајева коришћења. Како би овај захтев био испуњен потребно је увести одређени ниво формализма у опису акција случаја коришћења. Због тога, ова група аутора предлаже увођење новог формата за текстуални опис случајева коришћења, који са једне стране

треба да буде разумљив и читљив, а са друге стране треба да обезбеди конзистентност UML модела и текстуалне спецификације случаја коришћења. Предложени приступ од стране ове групе аутора је познат као *„flow-oriented“* и при спецификација акција добрим делом се ослања на препоруке предложене од стране *Bittner* и *Spence* [BITTNER, K. & SPENCE, I. (2004)]. Предложен приступ је подржан одговарајућим NaUTiluS алатом *(Narrative Use Case Description Toolkit for Evaluation and Simulation)* који се састоји од скупа додатака *(plugin)* који су део ViPER платформе [WEB-ViPER].

Drazan и *Mencl* су предложили приступ који се заснива на примени рестриктивног природног језика *(constrained natural language)* за детаљну текстуалну спецификацију случајева коришћења на основу које је могуће аутоматски или полу-аутоматски генерисати различите моделе, најчешће применом различитих техника парсирања текста [DRAZAN, J. & MENCL, V. (2007)].

Some у свом раду *„A Meta-model for Textaul Use Case Description“* дефинише апстрактна синтаксу за текстуалну презентацију случаја коришћења која проширује UML метамодел [SOME, S. (2009)]. Аутор констатује да су одређени елементи (формализми) UML језика као што су акције, формално дефинисане преко метамодела (односно одговарајућих метакласа). Са друге стране, UML није формално дефинисао метамодел за текстуални опис случаја коришћења, иако текстуална нотација представља основну нотација за опис случаја коришћења. Стога, аутор дефинише метамодел за описивање интеракције између корисника система и система.

2.4.3. ПАТЕРНИ ЗА СПЕЦИФИКАЦИЈУ СЛУЧАЈЕВА КОРИШЋЕЊА

Случајеви коришћења и патерни као тема у области софтверског инжењерства заузимају значајно место већ пуних 20 година [ISSA, A. & ALALI, A.(2011)], [DIAZ, I. et al. (2008)]. Први радови у којима се помиње термин патерн су радови *Christopher Alexander-a* који овај термин користи у контексту пројектовања архитектуре у грађевинарству [CHRISTOPHER, A. et al. (1977)]. *Christopher* је патерне уочио на основу структуре градова и грађевинских објеката. Термине које је користио *Christopher* у објашњавању патерна, као што су на пример силе *(forces),* патерн језици *(pattern languages)* широко су у примени при дефинисању и објашњењу патерна.

Патерн у најопштијем смислу представља решења неког проблема, у неком контексту, који се може поново користити за решавање нових, сличних проблема. Уколико је контекст у коме се неки проблем појављује познат, тада се могу користити иста правила која су раније примењивана за решавање неког ранијег проблема.

Langlands у свом раду *„Inside the Oval: Use-Case Content Patterns“* [LANGLANDS, M. (2010)] даје приказ 10 патерна за спецификацију случајева коришћења. Сам назив рада *"испод елипсе односно испод случаја коришћења"* симболично означава да се аутор у овом раду бави проналажењем патерна који се односе на садржај онога што се налази унутар случаја коришћења, дакле, пажњу усмерава на унутрашњу структуру случаја коришћења, односно на спецификацију акција. Стога, ови патерни представљају шаблон за спецификацију акција сценарија случајева коришћења Приликом дефинисања патерна *Langlands* прати широко прихваћени стил за опис патерна: дефинисање контекста, проблема и решења за дати проблем у дефинисаном контексту. *Langlands* полази од чињенице да се у пословним информационим системима случајеви коришћења заправо своде на извршавање једне или више CRUD операције над пословним доменским ентитетима. Патерне које је идентификовао *Langlands* су:

1) Патерн *PROPERTY LIST* одговара *„fully-dressed“*патерну [COCKBURN, A. (2001)] и *„State 6“* патерну [BITTNER, K. & SPENCE, I. (2003)]. За разлику од већине других патерна које је *Langlands* дефинисао овај патерн се не користи за дефинисање шаблона (темплејта) за спецификацију сценарија случаја коришћења, већ за спецификацију својства пословног (доменског) објекта. Својство се у контексту овог патерна користи да означи било који атомски податак који корисник у интеракцији са системом уноси или мења, или који систем приказује кориснику. При спецификацији оних случајева коршћења који користе овај патерн дефинишу се својства пословног-доменског објекта битна за конкретни случај коришћења, и дефинишу се карактеристите тог својства. Карактеристике својства треба дефинисати тако да буду јасна и комлетна, а у исто време да се осигура њихова конзистентност у смислу да се специфицирају на јединствени начин. Она треба да пруже

довољно информација за друге учеснике у развоју софтвера као што су пројектанти корисничког интерфејса, програмери, тестери.

Спецификација пословног-доменског објекта се може приказати табеларно тако да се у редовима наводе својства објекта, а у колонама карактеристике тих својстава. *Langlands* наводи које су основне карактеристике које треба дефинисати за свако својство:

a) тип податка при чему је потребно дефинисати да ли је вредност која се уноси простог типа; на пример да ли је целобројна вредност или је вредност датумског типа, или је структурирани тип података као што је на пример адреса. Такође, треба дефинисати да ли се вредност која се уноси бира из неке унапред дефинисане листе вредности, или представља референцу на неки објекат који постоји у систему.

b) да ли је вредност која се уноси за одређено својство обавезна вредност или није, да ли је могуће ту вредност мењати или не.

c) правила валидације које важе за својство.

Langlands сматра да на овај начин специфицирана односно дефинисана својства пословног-доменског ентитета не представљају коначну и потпуну спецификацију, али представљају добру полазну основу за њен даљи развој. Приликом спецификације својстава потребно је дефинисати и њихово значење. Према [COCKBURN, A. (2001)], [ADOLPH, S. et al. (2003)] значење својстава треба дефинисати одвојено у речнику појмова *(glossary)*. Друго решење, за које се и *Langlands* залаже, али које није у потпуности развијено, јесте референцирање свих својстава на одговарајући пословни доменски модел *(Business Domain Model)* [BITTNER, K. & SPENCE, I. (2003)]. Стога, приликом спецификације случаја коришћења потребно је извући сва својства пословног-доменског објекта која се користе у акцијама сценарија случаја коришћења и специфицирати их на начин како је то дефинисано према PROPERTY LIST патерну. Спецификација својства треба бити изван акција сценарија случаја коришћења и за свако својство треба дефинисати име, одговарајући тип податка и правила валидације, али и друге

информације уколико су потребне и значајне. Из акција случаја коришћења потребно је се референцирати на овако дефинисана својства.

2) Патерн *BASIC CREATOR* се користи као образац односно шаблон при спецификацију случаја коришћења када корисник жели да сачува одређени пословни-доменски објекат. Приликом креирања и чувања пословног-доменског објекта треба водити рачуна да су сва правила која се односе на валидацију података задовољена и треба осигурати да такав објекат већ не постоји у систему. Стога приликом чувања објекта потребно је унети мимимални скуп података који се односе на пословни-доменски објекат на основу којих је могуће, са једне стране утврдити да такав објекат не постоји у систему, а са друге стране проверити да ли су сва валидациона правила и правила интегритета задовољена.

3) Патерн *VIEWER / UPDATER* се користи као образац при спецификацији оних случајева коришћења када корисник жели да: а) погледа текуће податке који се чувају о објекту и/или б) измени текуће податке. Дакле, овај патерн се користи код случајева коришћења ажурирања вредности атрибута за одређени објектат и/или прегледа постојећих вредности које се чувају о објекту. Према овом патерну текући подаци о објекту се могу погледати и/или изменити само уколико је пословни-доменски објекат претходно идентификован. Пре извршења овог случаја коришћења потребно је пронаћи и изабрати објекат (чији се подаци желе изменити) преко одређеног случаја коришћења. Према овом патерну претрага пословних-доменских објеката не треба да буде део овог случаја коришћења, већ треба да претходи овом случају коришћења.

4) Патерн *SIMPLE VIEWER / UPDATER*. Пословни-доменски објекти могу имати различиту структуру. *Langlands* прави разлику између једноставних и сложених објеката. Једноставни су они објекти који имају релативно мали број атрибута и који немају или ако имају везе, са другим пословним-доменским објектима, код којих је пресликавање 0..1 или 1..1. Сложени објекти могу да имају већи број атрибута и обично ка другим пословним-доменским објектима имају пресликавање

0..* или 1..*. SIMPLE VIEWER / UPDATER патерн се као образац користи при спецификацији оних случајева коришћења који се извршавају над једноставним пословним-доменским објектима.

5) Патерн *MULTIPLE SELECTABLE PATHS.* се користи као образац при спецификацији случајева коришћења који се односе на преглед и/или измену сложеног пословног-доменског објеката. Сложени објекат може да садржи релативно велики број атрибута (често су атрибути код ових објеката организовани по групама односно логичким целинама, при чему се унос и измена тих атрибута најчешће врши по овим групама). Такође, сложени објекат најчешће има једну или више веза са пресликавањем 0..* или 1..* ка другим пословним-доменским објектима. Према овом патерну, информације које корисник жели да види или мења о сложеном објекту се деле у групе односно логичке целине и за сваку логичку целину података се креира главни сценарио. Стога се главни сценарио за преглед и/или измену сложеног објекта састоји од више могућих главних сценарија који се могу изабрати у зависности од тога која група података се жели погледати и/или мењати.

6) Патерн *CRITERIA SEARCH*. У пословним информационим системима велики број случајева коришћења се заснива на проналажењу постојећих пословних-доменских објеката у систему и измени текућих података над објектима. Поставља се питање на који начин корисник врши интеракцију са системом како би пронашао жељени објеката. *Langlands* види решење у *CRITERIA SEARCH* патерну. *Langlands* разликује најмање два начина који се користе за претрагу: 1) према првом начину претрага се врши на основу вредности које се задаје за унапред дефинисана својстава објекта, 2) према другом начину претрага се врши на основу вредности која се слободно уноси (најчешће је то неки стринг у комбинацији са џокер знацима) при чему се не наводе веза унетог текста са доменским објектом или неким својством доменског објекта. Само је први начин претраге дефинисан овим патерном.

7) Патерн *SELECTOR.* Често се у оквиру неке од акција случаја коришћења од корисника захтева да изврши селекцију (избор) једног

или више објеката истог типа који постоје у систему. Поставља се питање на који начин корисник врши овај избор. Решење је да се у оквиру ове акције позове случај коришћења којим се дефинише начин избора односно селекције објеката. Дакле, случај коришћења којим се специфицира избор једног или више објеката одређеног типа се описује преко *SELECTOR* патерна. Стога, случај коришћења који је означен као *SELECTOR* се не позива директно од стране корисника, већ из неке конкретне акције неког другог случаја коришћења и који као резултат враћа један или више изабраних објеката. Стога, случај коришћења који је означен као *SELECTOR* повезан је са позивајућим случајем коришћења UML везом *include*. Листа објеката из које корисник врши селекцију може бити различита у погледу броја елемената. Тако на пример, листа може да садржи ограничени, мали број елемената при чему кориник селекцију може да врши директно, на пример из падајуће листе. Са друге стране број елемената листе може бити толико велики да је практично немогуће вршити селекцију из падајуће листе, већ је за проналазак елемената потребно дефинисати одређени критеријуме претраге.

8) Патерн *DESTRUCTOR* се користи као образац при спецификацији случајева коришћења који се односе на брисање пословног-доменског објеката
9) Патерн *KNOWN OBJECT.* Овај патерн се користи при спецификацији случаја коришћења када се над неким доменским објектом који је идентификован (пронађен) из неког другог случаја коришћења потребно извршити неку од операција на пример операцију брисања, измене или прегледа текућих података. KNOWN OBJECT се назива објекат који је идентификован у неком ранијем случају коришћења (на пример неком случају коришћења претраге) и који је прослеђен другом случају коришћења (на пример случају коришћења у коме ће се текући подаци о пословном-доменском објекту мењати). KNOWN OBJECT објекат се још назива *context* објекат.
10) Патерн *OBJECT MANAGER Langlands* користи како би описао на који начин се дефинисани патерни могу користити интегрисано.

Cruz је у свом раду *„A Pattern Language for Use Case Modeling“* [CRUZ, A.M. (2014)] дефинисао и објаснио патерне случаја коришћења за спецификацију пословних информационих система. Предложени патерни се могу користити заједно са стандардним UML-ом како би се олакшало разумевање комплетног модела случаја коришћења.

Cruz је мишљења да модел случаја коришћења треба креирати у складу са доменским моделом тако да постоји јасна веза између модела случаја коришћења и доменских класа и њених операција. Модел случајева коришћења на тај начин допуњује доменски модел са:

- идентификованим функцијама које систем треба да обезбеди (CRUD операције),
- идентификованим операцијама навигације између доменских објеката за које су дефинисани случајеви коришћења,
- идентификованим акторима система (дефинисање функција система којима сваки од актора може да приступи).

Cruz дефинише два типа случаја коришћења [CRUZ, A.M.R. & FARIA, J.P. (2009)], [CRUZ, A.M.R. & FARIA, J.P. (2010)]:

- независни случајеви коришћења *(Independent use cases)* представљају случајеве коришћења који могу бити инстанцирани директно, па стога постоји директна веза између актора и случајева коришћења овог типа
- зависни случајеви коришћења *(Dependent use cases)* који могу бити инстанцирани само из другог случаја коришћења (изворног (source)) јер они зависе од контекста тог случаја коришћења. Ови случајеви коришћења проширују или су укључени у извршење изворног случаја коришћења.

Cruz је груписао патерне у зависности од структуре доменског објекта за који је дефинисан случај коришћења у следеће групе:

- Обрада ентити инстанци *(Manage an entity instance)*
- Обрада зависних ентити инстанци *(Manage dependent related entity instances)*
- Обрада независних повезаних ентити инстанци *(Manage independent related entity instances)*

- Обрада зависних повезаних ентити инстанци *(Manage dependent related entity collections)*
- Обрада независних повезаних колекција ентити инстанци *(Manage independent related entity collections)*.

Diaz са групом аутора у раду *„Specification pattern for use cases“* [DIAZ, I. et al.(2008)] предлаже општи формат и смернице за документовање случајева коришћења, а у циљу да се ублаже проблеми које са собом носи спецификација случајева коришћења природним језиком који по својој природи може бити двосмислен, непрецизан и нејасан. Због тога су, аутори фокусирани на лингвистичку анализу садржаја случаја коришћења односно анализу структуре и композиције реченица.

2.4.4. ТРАНСФОРМАЦИЈА МОДЕЛА ЗАХТЕВА У МОДЕЛЕ АНАЛИЗЕ

Креирање доменског модела (најчешће описаног преко дијаграма класа или модела објеката и веза) на основу функционалних захтева је једна од активности која се јавља у скоро свим методама развоја софтера. У методама развоја софтвера које користе стратегију засновану на случајевима коришћења (а посебно објектно-оријентисаним методама), случајеви коришћења играју битну улогу за идентификовање класа и метода. Обично се у тим приступима случајеви коришћења користе заједно са осталим UML моделима [I. JACOBSON et al., 1992], [I. JACOBSON et al., 1999], [LARMAN, C. (2002)], али и поред тога не постоји добро утемељена техника која омогућава једноставну трансформацију модела класа на основу модела случајева коришћења [ANDA, B. et al. (2005)].

Anda и група аутора су у раду *„Investigating the Role of Use Cases in the Construction of Class Diagrams“* објавили резултате до којих су дошли извођењем серије експеримената ради утврђивања улоге који случајеви коришћења имају у креирању доменског модела [ANDA, B. et al., (2005)]. Наиме, ови аутори су анализирали две технихе за откривање доменског модела. Прву технику су назвали *„derivation“* техника која је заснована на директном откривању доменског модела из случајева коришћења, док су другу технику назвали *„validation“* техника, према којој се доменски модел креира независно од случајева коришћења, најчешће на основу граматичке анализе текстуалне спецификације захтева, а случајеви коришћења се касније користе ради валидације доменског модела и његовог побољшања [MEYER, B. (1999)]. Први експеримент представљао је пилот пројекат [SYVERSEN, E. et al. (2003)] у коме је учествовало 26 ученика који је имао за циљ поређење ове две технике, а одмах затим су креирали и два експеримента: први експеримент са узорком од 53 студента, и други експеримент који су чинили професионални програмери (њих 22). Резултати до којих су дошли могу се сумирати на следећи начин:

a) Експеримент 1 је показао да се применом технике валидације долази до комплетнијег доменског модела

b) Експеримент 2 је показао да не постоји разлика између ове две технике у погледу комплексности

c) *„Derivation"* техника доводи до нешто боље структуре класа у доменском моделу, али ова разлика није значајна у експерименту 2.

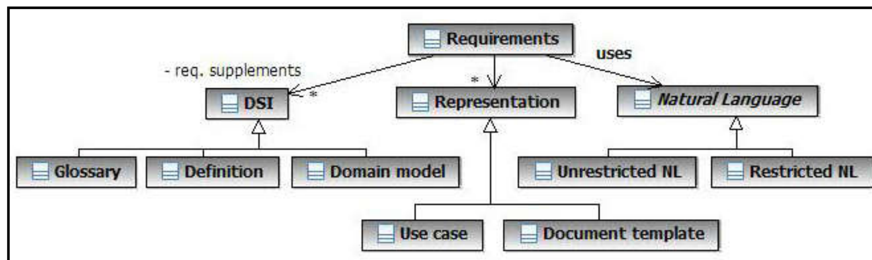
d) У погледу потребног време за креирање доменског модела не постоји разлика између ове две технике.

Yue са групом аутора је спровео истраживање које је обухватило систематичан преглед литературе која се односи на постојеће приступе у трансформацији корисничких захтева у модел анализе и резултате објавили у виду извештаја [YUE, T. et al. (2009)] и научно истраживачког рада [YUE, T. et al. (2011)]. Циљ овог истраживања је био да се направи систематичан преглед постојеће литературе која се односи на трансформацију текстуалних захтева у модел анализе, дефинисање отворених питања и препорука за будуће правце истраживања. Ово истраживање је обухватило анализу 20 студија (16 приступа) који су селектовани након детаљно планиране процедуре за селекцију радова који су у периоду од 1996 до 2008 објављени у часописима, конференцијама и књигама из области софтверског инжењерство. С обзиром да се захтеви могу специфицирати: а) у форми текста (природним језиком), б) у форми случајева коришћења, ц) у форми унапред дефинисаних образаца (темплејта) или ц) применом формалних језика (који су овом анализом искључени) креиран је концептуални оквир како би сви селектовани приступи били синтетизовани и како би били упоредиви. Овај концептуални оквир садржи заједничке концепте и таксономију која је неопходна за анализу, синтезу и поређење идентификованих приступа (студија). Тако на пример таксономија која се односи на захтеве се састоји од три концепта који чине уређену тројку и то (**Error! Reference source not found.**):

1) **Доменски Специфичне Информације** (*Domain Specification Information – DSI)* - Приликом креирања модела анализе на основу захтева користе се неки додатни документи у коме је описана терминологија која је коришћена у спецификацији захтева.

2) **Приказивање** (*Representation*) - Захтеви се могу приказати преко случајева коришћења или коришћењем шаблон документа.

3) **Природни језик** (*Natural Language)* – Захтеви се могу описати преко природних језика који могу имати или немати рестрикцију.



Слика 4. Yue таксономија за систематичан преглед литературе [YUE, T. et al. (2009)]

Овако дефинисана уређена тројка *(DSI, Representation, Natural Language*) представља патерн по коме су сви приступи класификовани. На пример, уређена тројка *(None, Use case, Yes)* означава приступ који не захтева DSI, користи случајева коришћења као технику за спецификацију захтева, а том приликом се користи рестриктивни природни језик за спецификацију случајева коришћења. За наше истраживање од посебног су интереса приступи који користе случајеве коришћења за спецификацију захтева. У оквиру овог истраживања идентификоване су следећи приступи са уређеном тројком:

a) ***(None, Use cases, No)*** [I. DIAZ et al. (2005)], [L. M. G. FEIJS, (2000)], [INSFRÁN, E. (2002)]
b) ***(None, Use cases, Yes)*** [M. ŚMIAŁEK et al. (2007)]
c) **(*Glossary*, *Use cases*, *Yes*)** [LIU, D. (2003)], [K. SUBRAMANIAM et al. (2004)]
d) ***(Domain model, Use cases, Yes)*** [SAMARASINGHE, N. & SOMÉ, S. (2005)], [S. SOMÉ, 2006]

На основу спроведеног истраживања *Yue* са ауторима закључује следеће:

i. У моделом вођеном развоју софтвера трансформација модела захтева у модел анализе један је од најбитнијих корака. Али и поред тога што постоји значај број истраживања на ову тему, још увек не постоји практично, потпуно аутоматизовано решење.
ii. Пожељно је не користити додатне информације из неког другог модела, а уколико се користе пожељно је да то буде из речника *(glosarry)*
iii. Случај коришћења као технику треба користити јер је је једна од најчешће коришћених техника за спецификацију захетва

iv. Рестриктиван природни језик се може користити у спецификацији захтева како би аутоматска трансформација била лакше подржана

v. У складу са дефинисаним патерном (уређеном тројком) који се односи на начин представљања корисничких захтева препорука је користити неки од наведених уређених тројки *(None, Use cases, No)*, *(None, Use cases, Yes)*, *(Glossary, Use cases, Yes)* и (Glossary, Use cases, No).

ПОГЛАВЉЕ 3. SILAB-MDD ПРИСТУП

Silab-MDD приступ (метода) се користи у развоју софтверских система. Овај приступ је заснован на упрошћеној Лармановој методи[7] развоја софтвера [Vlajic, S. (2015)]. **Silab-MDD** приступ описује моделе (добијене у различитим фазама развоја софтвера) и трансформације између модела.

У оквиру **SILAB-MDD** приступа дефинисани су *сопствени доменски специфични језици:*

a) ***Use Case Domain Specific Language - UCDSL*** за спецификацију модела случаја коришћења, доменског модела и модела прелаза стања.
b) ***Data-Flow Diagram Domain Specific Language - DFDDSL*** за спецификацију дијаграма тока података и
c) ***Data Dictionary Doman Specific Language - DataDDSL*** за спецификацију речника податка.

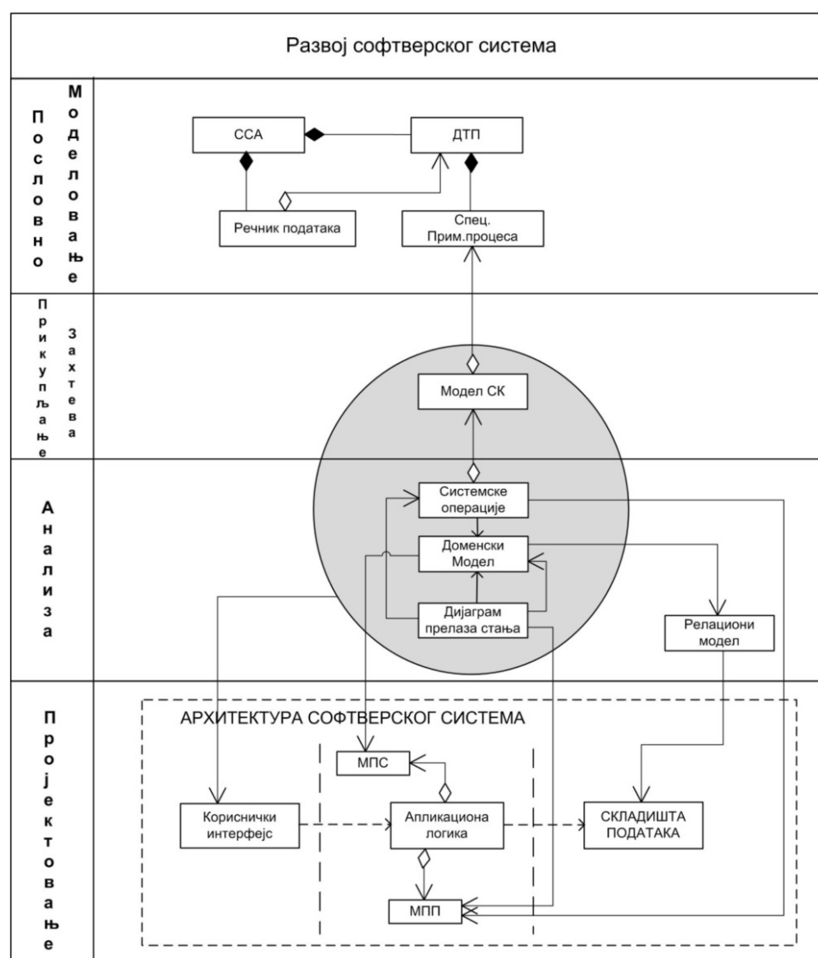
3.1 УПРОШЋЕНА ЛАРМАНОВА МЕТОДА РАЗВОЈА СОФТВЕРА

Упрошћена Ларманова метода развоја софтвера састоји се из следећих фаза (Слика 5):

a) пословно моделирање (пословна анализа)
b) прикупљање захтева
c) анализа
d) пројектовање
e) имплементација и
f) тестирање

[7] Упрошћена Ларманова метода развоја софтвера се примењује од 2004. године на Факултету организационих наука на неколико предмета *Катедре за софтверско инжењерство,* који се баве пројектовањем, имплементацијом и развојем софтверских система. Коришћењем ове методе је урађено на десетине завршних и мастер радова на ФОН-у. Ова метода је упрошћена јер разматра а) фазе развоја софтвера, б) моделе односно артифакте фаза развоја софтвера и ц) зависности између артифаката. Она не узима у обзир оне аспекте развоја софтвера који се односе на моделовање пословних процеса који прате фазе развоја софтвера. Поред тога прва фаза Ларманове методе *пословно моделирање* је реализована преко пословних случаја коришћења, док је упрошћене Ларманове методе уместо тога дата Структурна систем анализа.

Фаза пословног моделирања се описује преко *Структурне Систем Анализе* која представља методу за функционалну спецификацију пословног система. Структурна систем анализа се описује помоћу *дијаграма токова података)* и *речника података*. На основу дијаграма тока података могуће је направити *речник података* и дати прецизну спецификацију основних (примитивних) процеса система.



Слика 5. Упошћена Ларманова метода

Фаза прикупљања захтева, се описује преко *модела случаја коришћења* који се добија на основу спецификације основних процеса. У фази анализе се описује логичка структура и понашање софтверског система односно ***пословна логика*** софтверског система. Структура софтверског система се описује преко *концептуалног (доменског)* и *релационог модела* док се понашање софтверског система описује помоћу *секвенцних дијаграма* и *системских операција*. У фази пројектовања се описује *архитектура* софтверског система која је *тронивојска* и

састоји се од: а) *корисничког интерфејса* б) *апликационе логике* и ц) *складишта података*. У фази имплементације се праве *имплементационе компоненте* које се реализују у некој од постојећих технологија (*Јава, .NET*,...). У фази тестирања, свака од компоненти се тестира тако што се за сваку од њих праве *тест случајеви*, *тест процедуре* и *тест компоненте*.

5.2 Сопствени доменски специфични језици Silab-MDD приступа

SilabMDD приступ се може посматрати као проширење Упрошћене Ларамановe методе развоја софтвера јер поред стандардних фаза развоја софтвера, *SilabMDD* приступ разматра и трансформације између модела добијених у различитим фазама развоја софтвера. *SilabMDD* приступ користи две стратегије развоја софтвера:

а) Развој софтвера зснован на случајевима коришћења (*Use Case Driven Development*),
б) Моделом вођен развој софтвера (*Model Driven Development*)

Један од начина да се подржи моделом вођени развој софтвера јесте коришћење доменско-специфичних језика чији је један од представника **SILAB-MDD** приступ. Он користи два доменско-специфична језика за спецификацију два кључна артифакта активности пословне анализе: а) **DFDDSL** језик за спецификацију дијаграма тока података и б) **DataDDSL** језика за спецификацију токова података и складишта података[8] и **UCDSL** доменско-специфични језик за спецификацију случајева коришћења за исказивање доменског модела, модела прелаза стања и случајева коришћења активности прикупљања захтева и анализе.

Коришћем језика за спецификацију модела у великој мери се аутоматизује развој софтвера што свакако позитивно утиче и на продуктивост и на флексибилност развоја софтвера. Трансформацијама се један модел преводи у други модел који се тада семантички обогаћује (описују се детаљи који су релеванти за тако формирани нови модел) и као такав поново трансформише у нови модел или програмски код.

[8] Овај језик се свакако може користити независно од DFDDSL језика јер се њиме у општем смислу могу описивати документа у пословним информационим системима

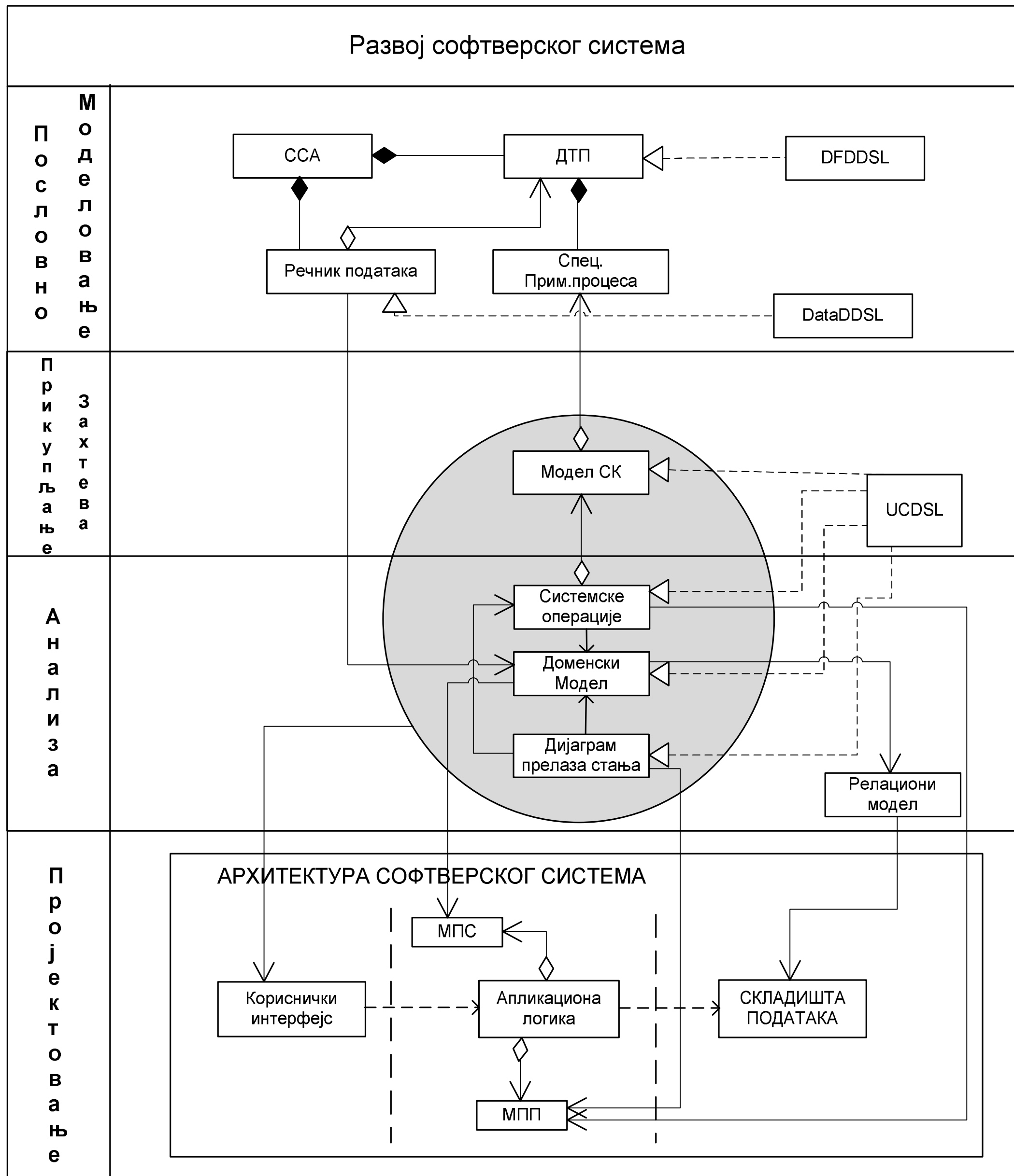
Сви доменско-специфични језици који се користе у **SILAB-MDD** приступу развијени су коришећњем *JetBrains Meta Programing Systems*[9] алата. Ниже на слици (Слика 6) Слика 108 приказана је улога ових језика у предложеном приступу.

Имајући у виду да развијени језици користе текстуалну конкретну синтаксу, а да је за приказ појединих модела адекватнија графичка синтакса (нотација) у плану је да се у оквиру **SILAB-MDD** приступа креирају одговарајуће трансформације у моделе који су погодни за графичку презентацију (пре свега мислимо на одговарајуће UML моделе). Тако се на пример за приказ доменског модела може користити UML дијаграм класа, или за приказ модела промене стање UML дијаграма прелаза стања.

Између осталог, овај недостатак је могуће надокнадити креирањем посебних библиотека за цртање графичких концепата и њихова интеграција у оквиру алата. Тако је на пример у оквиру Лабораторије за софтверско инжењерство овог лета прокренут пројекат за интерпретацију текстуалне спецификације хијерархијског дијаграма тока података и његове визуелне презентације у складу са концептима дијаграм тока података дефинисаним у оригиналној ССА методи[10]. На тај начин спецификација дијаграма тока података ће се вршити на исти начин (коришћењем **DFDDSL** језика), али ће приказ самог модела дијаграм тока података бити подржан и текстуалном и графичком нотацијом. Креиране моделе тако је могуће извести у XMI формат како би се могли користити и у другим алатима.

[9] https://www.jetbrains.com/mps/

[10] У реализацију ових пројеката активно су укључени и студенти мастер студија смера Софтверско инжерерство и рачунарске науке Факултета организационих наука

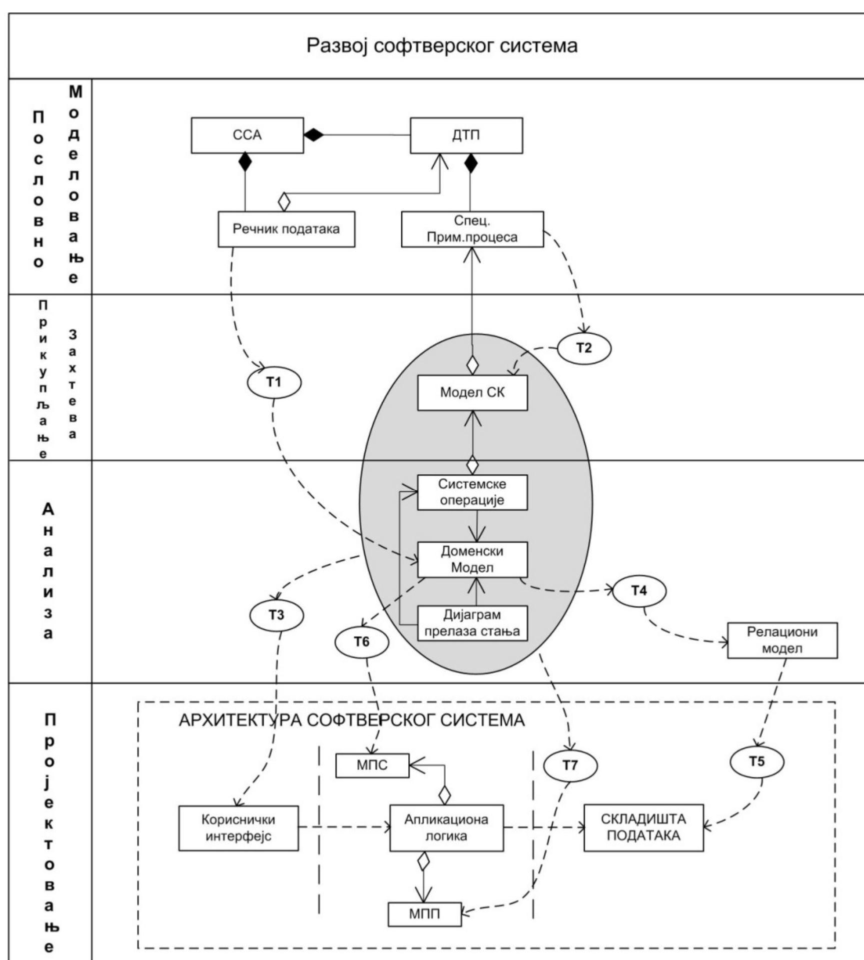


Слика 6. Концептуални приказ *SilabMDD* приступа.

3.2 ТРАНСФОРМАЦИЈЕ ИЗМЕЂУ МОДЕЛА SILAB-MDD ПРИСТУПА

У овом поглављу биће представљене трансформације које чине саставни део *SilabMDD* приступа (Слика 7):

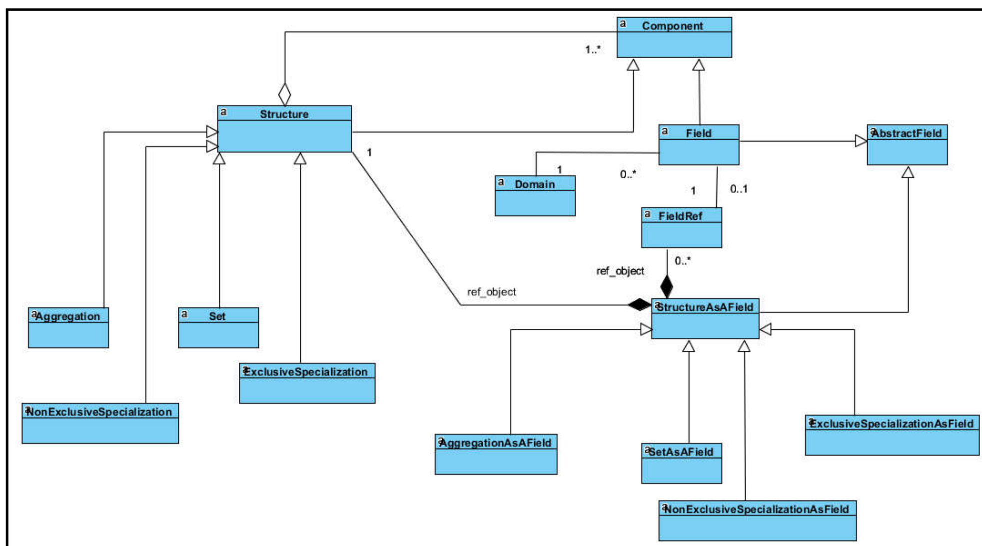
- Т1: Трансформација из модела речника података у доменски модел.
- Т2: Трансформација из модела дијаграма тока података у модел случајева коришћења.
- Т3: Трансформација из модела захтева у модел корисничког интерфејса.
- Т4: Трансформације из доменског модела у релациони модел.
- Т5: Трансформације из релационог модела у DDL наредбе.
- Т6: Трансформације из доменског модела у модел пословне структуре.
- Т7: Трансформације из модела захтева у модел понашања пројектовања.



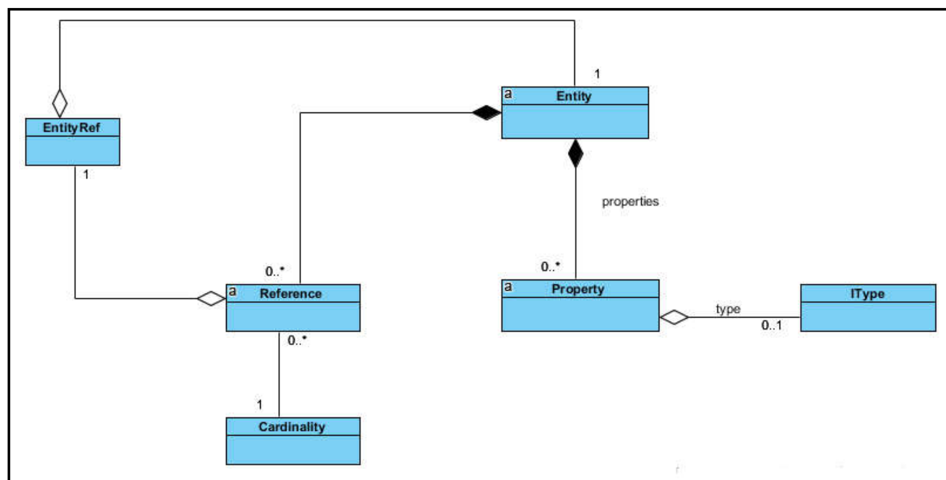
Слика 7. Улога трансформација у SILABMDD приступу

Т1: Дефинисање трансформације из речника података у доменски модел

На слици (Слика 8) приказани су основни концепи метамодела речника података, док су на слици (Слика 9) приказани основни концепти метамодела доменског модела. Детаљан приказ ова два метамодела дат је у наредном поглављу, али је овде дат њихов приказа ради лакшег разумевања дефинисане трансформације.



Слика 8. Основни концети метамодела речника података



Слика 9. Део метамодел доменског модела

Ниже је на логичком нивоу описана трансформација која на основу модела речника података креира одговарајући доменски модел пратећи дефинисана правила:

1) Structure концепт из речника података се преводи у концепт Entity из доменском модела
2) Концепт Field из речника подаак се трансформише у концепт Property из доменског модела
3) Концепт Domain из речника података се трансформише у конкретни концепт IType из доменског модела
4) Апстрактни концепт StructureAsAField из речника података се трансформише у конкретни концепт Reference из доменског модела

T2: Дефинисање трансформације из модела дијаграма тока података у модел случајева коришћења

Као што је предходно речено, ССА се заснива на функционалној декомпозицији система. Једна од основних карактеристика ове методе се огледа у могућности хијерархијског описа процеса. За сваки процес, односно функцију обраде се дефинише дијаграм тока података. Улога дијаграма тока података јесте идентификовање свих процеса у систему који на основу улазних токова токова података, врше обраду и генеришу излазне токове. Декомпоновање процеса се врши до нивоа примитивних процеса. Примитивни процеси представљају атомске процесе који се даље не декомпонују. Уколико процес има само један улазни и један излазни ток сигурно се може сматрати да је примитиван.

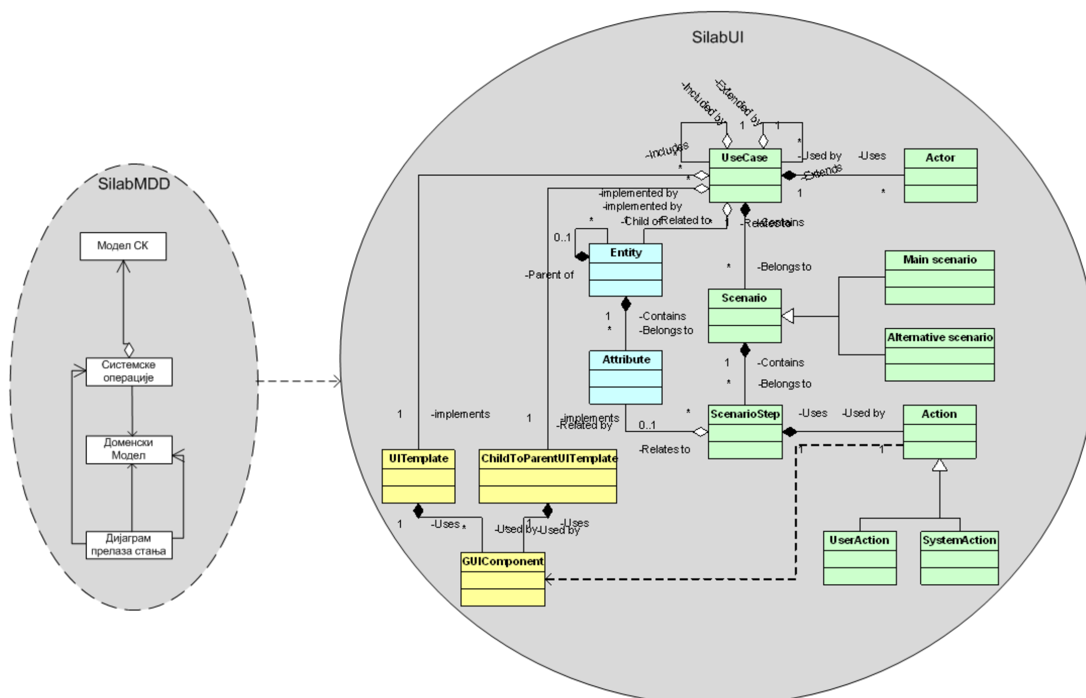
У циљу дефинисања трансформације која на основу модела дијаграма тока података креира модел случаја коришћења (не и спецификацију случаја коришћења) потребно је за сваки примитивни процес дефинисати једно примарно складиште над којим се врши обрада. Трансформација примитивних процеса у модел случајева коришћења се извршава за сваки примитивни процес који се налази на моделу дијаграма тока података на следећи начин:

a) за сваки примитивни процес који је везан за примарно складиште података улазним током креира се случај коришћења за претрагу (тип случаја коришћења *search*)
b) за сваки примитивни процес који је везан за примарно складиште података излазним током креирају се два случаја коришћења: случај коришћења за креирање (тип случаја коришћења *create)* и случај коришћења за ажурирање (тип случаја коришћења *manage*)

Т3: Дефинисање трансформације из модела захтева у модел корисничког интерфејса

На основу модела захтева, могуће је дефинисати трансформацију у модел корисничког интерфејса. Сама дефиниција модела корисничког интерфејса била је предмет посебног истраживања названог SilabUI приступ [ANTOVIĆ, I. (2015)] [ANTOVIĆ, I. et al. (2012)] које је спроведено у Лабораторији за софтверско инжењерство, а које је имало за циљ да аутоматизује процес развоја корисничког интерфејса на основу улазне спецификације базиране на спецификацији случајева коришћења. SilabUI приступ обухвата мета-моделом прецизно дефинисану улазну спецификацију чију основу чине концепти везани за модел случајева коришћења, а која се може проширити концептима који прецизније специфицирају елементе корисничког интерфејса који ће бити генерисан. Поред тога, дефинисана су правила трансформације, која су имплементирана на тај начин да се као резултат добија програмски код корисничког интерфејса у одабраној имплементационој технологији. Уз наведено, развијени су и алати за текстуалну, графичку и визуелну обраду улазне спецификације и генерисање корисничког интерфејса. Дакле, коришћењем SilabUI пројекта могуће је директно генерисање корисничког интерфејса на бази основне спецификације коју чини спецификација случајева коришћења. Резултат генерисања је програмски код корисничког интерфејса који ће садржати подразумеване карактеристике (шаблон корисничког интерфејса, коришћене графичке компоненте и сл.), тј. карактеристике које су трансформацијом предвиђене уколико се у улазној спецификацији нису прецизиране информације везане за кориснички интерфејс. Уколико добијени интерфејс није одговарајући, могуће је проширити улазну спецификацију информацијама које директно описују елементе корисничког интерфејса и на тај начин прецизније одговорити на захтеве корисника.

Како се улазна спецификација SilabUI приступа ослања на случајеве коришћења, извршена је интеграција ова два приступа на тај начин што је дефинисана трансформација модела захтева SilabMDD приступа у модел улазне спецификације SilabUI приступа. Ниже су приказани концепти SilabMDD приступа који се трансформишу у концепте SilabUI приступа.



Слика 10. Пресликавање концепата модела захтева у модел корисничког интерфејса

Трансформације између модела ова два приступа претходи дефинисање модела навигације у SilabMDD приступу који дефинише распоред случајева коришћења на будућем корисничком интерфејсу.

У табели 1. приказани су концепти између којих се успостављану релације приликом трансформације модела захтева SilabMDD приступа у модел улазне спецификације SilabUI приступа.

Табела 1. Трансформација концепата SilabMDD приступа у концепте SilabUI приступа

Концепт SilabMDD приступа	Концепт SilabUI приступа
Модел СК	UseCase, Actor, Scenario, MainScenario, AlternativeScenario, ScenarioStep
Доменски модел	Entity, Attribute
Системске операције	Action, UserAction, SystemAction

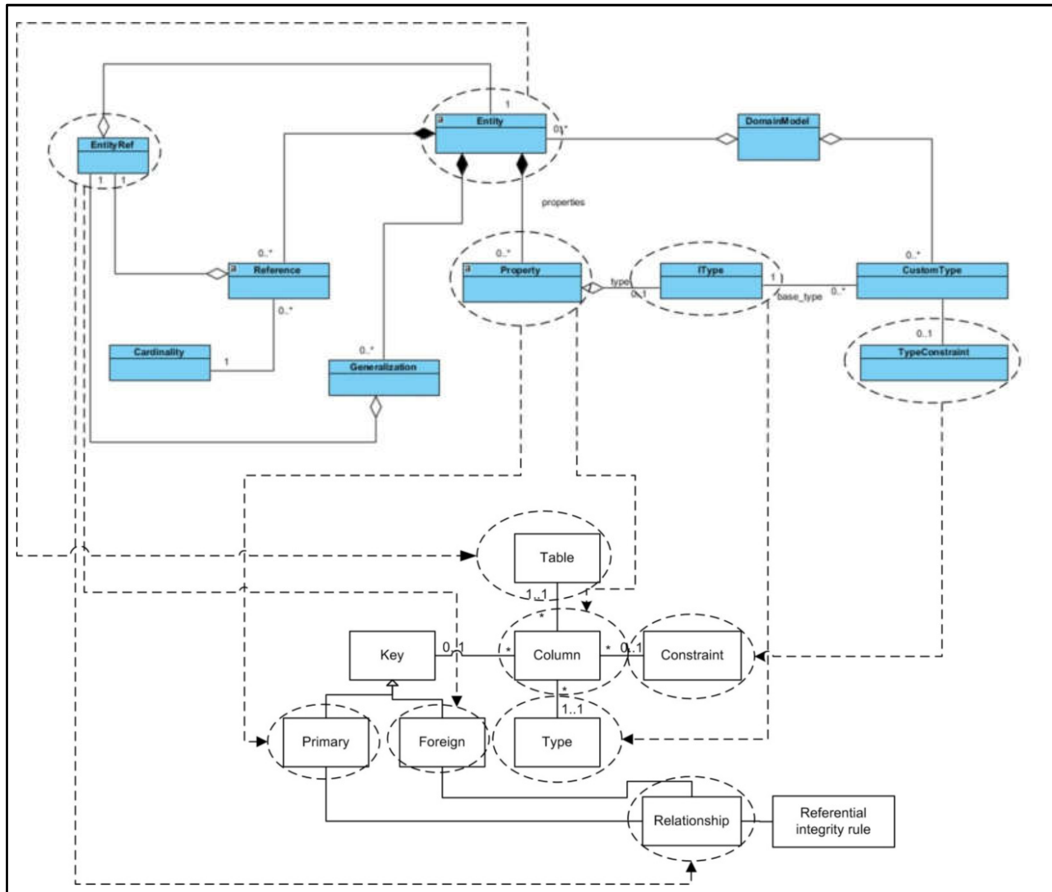
T4: Дефинисање трансформације из доменског модела у релациони модел

Под релационим моделом подразумевамо модел релација (табела) који је независан од конкретног система за управљање базом података. Основни концепти овог модела су релација, колона, примарни кључ, спољни кључ и веза која се остварује преко вредности примарног и спољњег кључа. У претходном делу овог поглавља објашњена је трансформација из модела речника података у доменски модел. Наставак на ову трансформацију представља трансформација доменског модела у релациони модел. Из самог описа основних концепта може се уочити да је трансформација доменског модела захтева у релациони модел могуће извршити једнозначно. Једино што би на први поглед могло да представља проблем јесте дефиниција примарних и спољних кључева. У ту сврху у моделу захтева приликом дефинисања атрибута доменске класе потребно је дефинисати да ли тај атрибут представља идентификатор. Моделом захтева предвиђено је специфицирање једног идентификатора за релацију. Уколико се више атрибута означи као идентификатор то ће представљати сложени идентификатор. Овако дефинисани идентификатор представља кандидат за примарни кључ релације. Што се тиче спољних кључева они се могу препознати из референтних атрибута односно атрибута који представљају везе између доменских концепата односно ентитета. Атрибут релације који представља спољни кључ биће идентификатор референтног доменског концепта. Релациони модел поред ових информација требао би да садржи и информације о ограничењима на вредности атрибута као и правила референцијалног интегритета. У пројекту *SilabBiz* бавимо се овим детаљима. Резултат овог пројекта требало би да буде *SilabBizDSL* језик који ће омогућити додавање детаља потребних за креирање ових ограничења. Из тог разлога ова проблематика није детаљније разматрана.

Као што је предочено, информације које су потребне за креирање релационог модела се налазе дефинисане у доменском моделу, па је стога могуће извршити трансформацију доменског у релациони модел. У наставку је представљен коначан скуп правила за трансформацију доменског модела захтева у релациони модел:

a) за сваки ентитет из доменског модела креира се одговарајућа табела. Концепт *Entity* из модела захтева се преводи у концепт *Table* из модела пројектовања структуре.
b) сваки атрибут *Entity* класе постаје колона табеле. Тип колоне је одређен типом ентити атрибута. Концепт *Property* из модела захтева се преводи у концепт *Column* из модела пројектовања структуре. Концепт *IType* из модела захтева се преводи у концепт *Type* из модела пројектовања структуре.
c) идентификатор *Entity* класе постаје примарни кључ табеле. Сви концепти *Property* класе који су означени као идентификатори преводе се у *Primary* концепт из модела пројектовања структуре.
d) на основу сваке релације креира се веза ка табели која одговара Entity класи која учествује у релацији. Ова веза се у релационом моделу остварује преко спољног кључа релације. Сви *EntityRef* конепти из модела захтева се преводе у концепт *Relationship* модела структуре пројектовања. Атрибут којим се остварује веза између два Entity концепта постаје постаје *ForeignKey* у моделу структуре пројектовања.
e) свака веза генерализације утиче на примарни кључ изведеног *Entity* концепта. Идентификатор основног *Entity* концепта постаје примарни кључ изведеног *Entity* концепта.
f) Ограничења дефинисана у доменском моделу пресликавају се у ограничења дефинисана за колону релациног модела. Коцепт *TypeConstraint* из модела захтева пресликава се у *Constraint* концепт модла структуре пројектовања.
g) Правила референцијалног интегритета нису део модела захтева па је неопходно додати информације о истим.

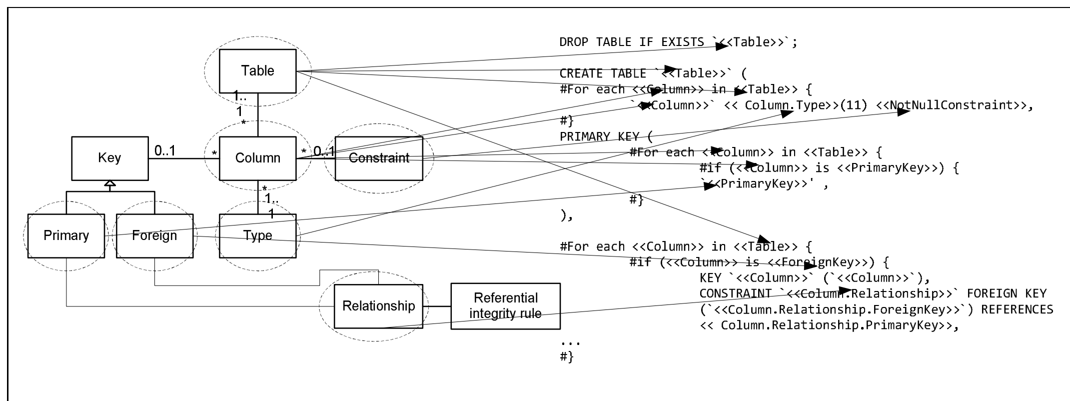
У наставку је приказанп пресликавање коцепата мета-модела захтева за дефинисање доменских класа у концепте мета-модела за дефинисање релационог модела (Слика 11).



Слика 11. Пресликавање концепата метамодела доменског модела у концепте релационог модела

T5: Дефинисање трансформације из релационог модела у DDL наредбе

Релациони модел састоји се од табела које могу бити међусобно повезане. Свака табела садржи колоне које су одређеног типа, као и ограничења која се дефинишу над колонама. С друге стране, колоне могу бити део примарног или спољног кључа. На основу модела релације за конкретни систем за управљање подацима могуће је креирати DDL наредбе. Креирање ове трансформације је могуће урадити уз додатно обогаћивање релационог модела информацијама специфичним за конкретан систем за управљање базом података. Такође, многи алати за рад са базама података омогућавају креирање/измену релационог модела коришћењем чаробњака и аутоматску трансформацију извршених измена у одговарајуће DDL наредбе. Овај тип трансформације представља трансформацију из модела у текст па није било могуће представити мета модел *DDL* наредбе. Уместо тога у наставку је дат пример обрасца помоћу кога је могуће извршити директну трансформацију концепта (*Table*, *Column*, *Key*…) из релационог модела у ***DDL*** наредбе.



Слика 12. Пресликавање концепата метамодела релационог модела у DDL наредбе

T6: Дефинисање трансформације из доменског модела у модел структуре пројектовања

Модел структуре пројектовања у објектно-оријентисаним језицима, као што је на пример Јава, представљају одговарајуће класе у којима су елементи структуре описани атрибутима. Дакле, атрибути класе дефинишу стање неког објеката, док се понашање објекта дефинише операцијама. У последње време тренд је да се подаци о класама (мета-подаци) чувају у оквиру саме класе коришћењем анотација. Мета-подаци доменских класа се највише користе од стране перзитентних оквира али се могу користити и у друге сврхе. Раније су се

ове информације углавном чувале у одговарајућим конфигурационим датотекама (најчешће *xml* датотекама). У контексту креирања доменских класа пројектовања битно је истаћи да доменске класе пројектовања могу садржати ограничења на вредност атрибута што је у великој мери подржано перзистентним оквирима. Ово је до те мере постало присутно да су анотације за дефинисање ограничења на вредности атрибута укључене у *de facto* стандарде везане за перзистентност података. Поред овога у фази пројектовања могуће је мењати апликациони доменски модел у складу са принципима пројектовања. Тако је доменски модел могуће обогатити класама које ће обезбедити коришћење неких генеричких механизама или пак искористити ранија решења која су се добро показала у пракси. Како је структура доменског модела (пословни доменски модел) дефинисана у моделу захтева, могуће је извршити трансформацију у модел пословне логике (апликациони доменски модел). Заправо, модел захтева садржи довољно богату семантику како би омогућио трансформацију у модел структуре пројектовања. У великом броју случајева трансформација из доменског модела у модел структуре пројектовања је директан и врло једноставан.

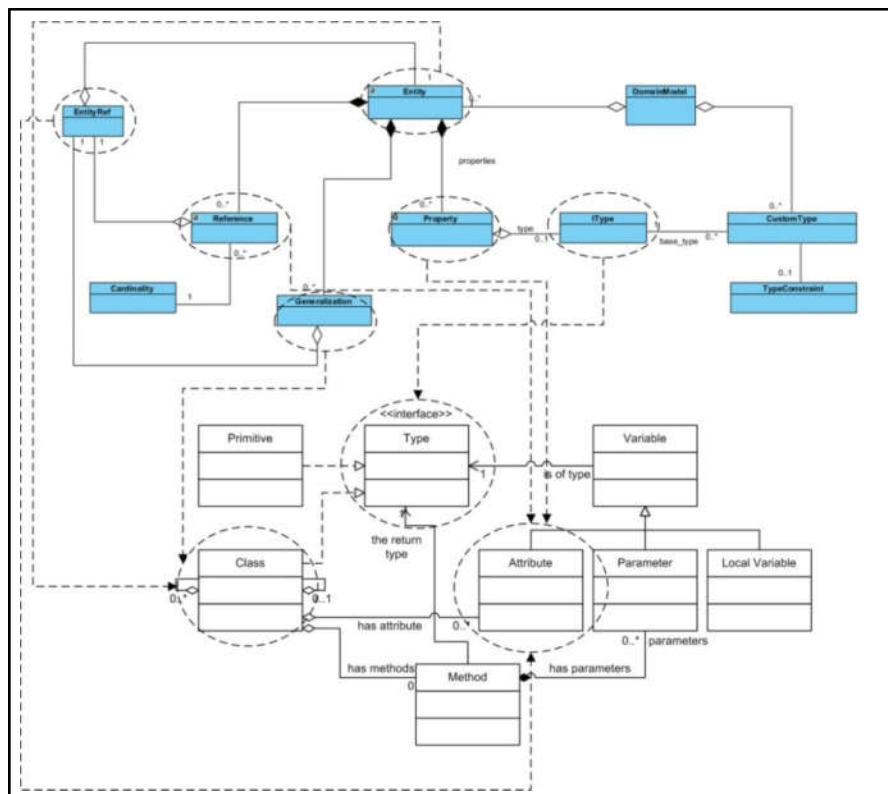
Према свему наведеном изводе се правила која морају да се поштују приликом трансформације:

a) За сваки ентитет из доменског модела креира се одговарајућа класа. Концепт *Entity* из модела захтева се преводи у концепт *Class* из модела структуре пројектовања.
b) Сваки атрибут ентити класе постаје атрибут класе пројектовања структуре. Концепт *Property* из модела захтева се преводи у концепт *Attribute* из модела структуре пројектовања.
c) У моделу захтева су дефинисани типови података за сваки од атрибута па се исти могу пресликати у одговарајуће типове модела структуре пројектовања. Концепт *IType* из модела захтева се преводи у концепт *Тип* из модела структуре пројектовања.
d) Свака веза између два ентитета прелази у атрибут референтног типа (референцира се на класу која одговара ентитету који учествује у релацији у доменском моделу (target елемент)). У овом контексту неопходно је истаћи да је у моделу захтева могуће дефинисати навигацију за сваку везу између два ентитета. Ово се остварује тако

што се референтни атрибут поставља у класу у којој се посматрана веза остварује. Треба нагласити да је у случају двосмерне везе потребно дефинисати атрибуте у обе доменске класе које формирају везу. У зависности од кардиналности пресликавања зависиће и тип референтног атрибута, па ће у случају када је горња граница кардиналности више(*) атрибут представљати колекцију објеката одговарајуће класе. Концепти ***Reference** и **EntityRef*** из модела захтева се преводе у концепт ***Attribute*** из модела структуре пројектовања.

е) Веза генерализициjе из доменског модела прелази у везу генерализације у објектном моделу. Концепт ***Generalization*** из модела захтева се преводи у концепт ***Class*** из модела структуре пројектовања.

На слици (Слика 13) је приказано пресликавање концепата модела захтева *SilabMDD* приступа везаног за дефинисање доменског модела у концепте модела структуре пројектовања. Ради једноставности извршена је апстракција појединих делова модела.



Слика 13. Пресликавање концепата метамодела доменског модела у концепте модела структуре пројектовања

Т7: Дефинисање трансформације из модела захтева у модел понашања пројектовања

Модел понашања пројектовања представља модел уговора о системским операцијама. За сваки уговор о системској операцији дефинише се: а) назив уговора, б) назив системске операције, в) улазни параметри за системску операцију, г) предуслов за извршење системске операције и д) пост услов системске операције. Ове информације се налазе у моделу захтева и применом ове трансформације из модела захтева могуће је креирати овако дефинисани модел понашања пословног система на следећи начин:

a) За сваку системску операцију, идентификовану у случају коришћења, креира се уговор. У оквиру модела захтева који користи SilabMDD приступ ове операције дефинисане су у оквиру доменског модела, односно доменске класе над којом се операција извршава. Тако дефинисане операције се придружују одговарајућој акцији (request block) у случају коришћења.
b) На основу контекста у случају коришћења у ком је позвана системска операција могу се идентификовати улазни параметри системске операције. То може бити доменски концепт који је означен као примарни концепт случаја коришћења или неки други концепт који је у релацији са њим.
c) За сваки случај коришћења дефинише се модел прелаза стања примарног објекта. Поред спецификације могућих стања специфицирају се и операције које преводе објекат из једног у друго стање. Из овог модела може се утврдити у којем стању се доменски објекат мора наћи како би се извршила системска операција. Тако дефинисано стање(а) доменског објекта представља предуслов за извршење системске операције.
d) На основу модела прелаза стања утврђује се стање у које доменски објекат прелази након извршења системске операције. То стање доменског објекта представља пост услов за системску операцију.

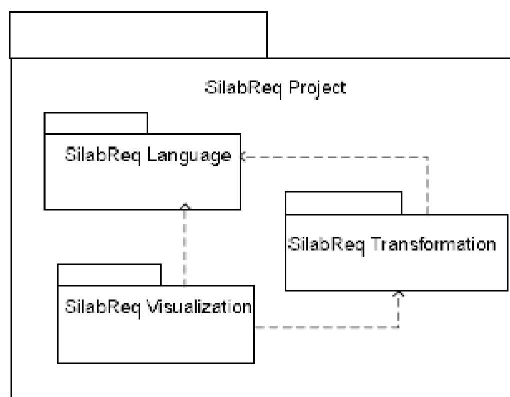
Модел пројектовања понашања је семантички богатији од предложеног модела захтева из чега произилази да се предложеном трансформацијом добија само део информација. Ово имплицира да је неопходно обогатити добијени модел

додатним информацијама како би исти могао да се трансформише у извршиви програмски код. У ту сврху у оквиру Silab пројекта развија се *SilabBiz* доменски језик за детаљнију спецификацију системских операцију. Овим језиком поред дефинисања ограничења која морају бити задовољена специфицира се и алгоритам извршавања системских операција. У оквиру *SilabBiz* пројекта се бавимо овом проблематиком па она овде неће бити детаљније разматрана.

3.3 О SILAB ПРОЈЕКТУ

У оквиру Лабораторије за софтверско инжењерство, Факултета организационих наука почетком 2007. године инициран је и покренут **Silab** пројекат. Основни циљ овог пројекта је био да се стандардизује, униформише и формализује начин спецификације корисничких захтева на основу који би се омогућило аутоматско генерисање различитих компоненти софтверског система.

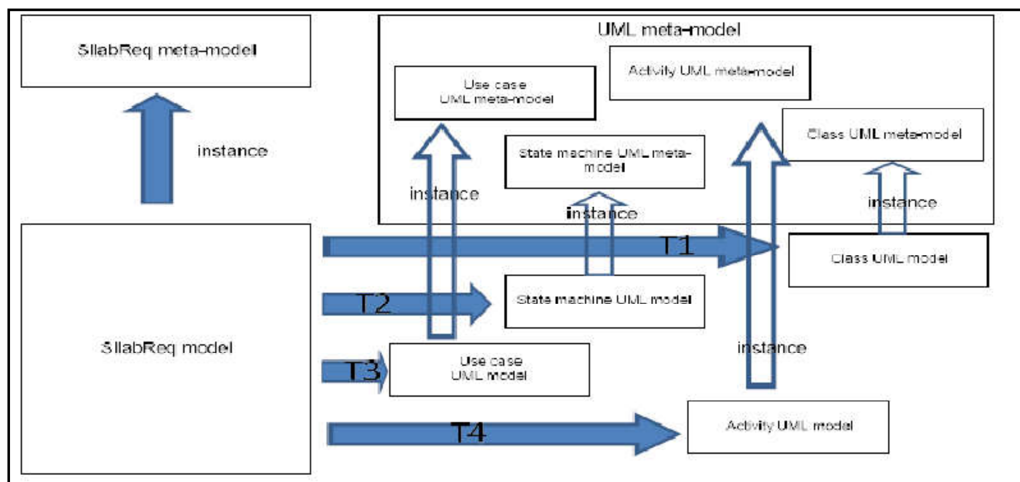
У почетку је овај пројекат био подељен на два главна подпројекта **SilabReq** и **SilabUI** пројекат која су се развијала одвојено један од другог. **SilabReq** пројекат се бавио проблемима који су се односили на начин спецификације и презентације корисничких захтева и њихове трансформације у различите UML моделе како би се олакшао процес анализе и осигурала валидација и конзистентност софтверских захтева. Главни део овог пројекта је чинио **SilabReq** [D.SAVIĆ et al. (2011)], [D.SAVIĆ et al. (2014)], [D.SAVIĆ et al. (2012)] [D.SAVIĆ et al. (2010)] доменски специфичан језик за спецификацију функционалних корисничких захтева у облику случајева коришћења. Са друге стране, **SilabUI** пројекат је разматрао значај односно утицај одређених елемената софтверских захтева и доменског модела на кориснички интерфејс, како би се омогућило аутоматско генерисање корисничког интерфејса на основу спецификације случајева коришћења и доменског модела [I.ANTOVIĆ et al. (2012)]. Ниже на слици дат је приказ **SilabReq** пројекта.



Слика 14. SilabReq пројекат

SilabReq пројекат је укључивао три кључне компоненте и то: језик за спецификацију корисничких захтева ***(SilabReq Language)***, компоненту за трансформације ***(SilabReq Transformation)*** и компоненту за визуелну

приказивање креираних модела *(**SilabReq Visualization**)*. У оквиру **SilabReq Language** компоненте налази се имплементиран **SilabReq** контролисан природни језик, док се у оквиру ***SilabReq Transformation*** компоненте налази скуп трансформација којима се **SilabReq** модел трансформише у одговарајући UML модел. На слици (Слика 15) дат је приказ ових трансформација, док је детаљан приказ **SilabReq** пројеката и одговарајућих трансформација дат у радовима [D.SAVIĆ (2014) at all.], [D.SAVIĆ (2015) at all.].



Слика 15. SilabReq трансформације

Све трансформације су имплементиране коришћењам Kermeta [Web-Kermeta] императивног језика за мета-моделовање који је у потпуности компатибилан са OMG Essential Meta-Object Facility (EMOF) мета-моделом [Web-OMG-UML] и Ecore мета-моделом над којим је креиран Eclipse-ов оквир за моделовање (Eclipse Modeling Framework EMF). **SilabReq Visualization** компонента је одговорна за визуелно представљање софтверских захтева преко UML дијаграма класа, UML дијаграма секвенци, UML дијаграма активности и UML дијаграма прелаза стања.

Када су оба пројекта (**SilabReq** ***и*** **SilabUI**) достигли одређени ниво зрелости, ова два пројекта почињу да се користе на интегралан начин тако да одређени резултати из **SilabReq** пројекта почињу да се користе као улаз за **SilabUI** пројекат. Овакав интегрални приступ први пут је коришћен за реализацију Kostmod 4.0 пројекта за потребе Министарства одбране Краљевине Норвешке [Kostmod 4.0 (2009)].

ПОГЛАВЉЕ 4. SILAB-UCMDM МЕТОДА

У овом поглављу дат је приказ *Silab-UCMDM* методе за спецификацију захтева која је заснована на моделу случајева коришћења. Спецификација захтева у оквиру *Silab-UCMDM* методе омогућена је преко посебно сопственог доменски специфичног језика (*UCDSL*). Помоћу UCDSL се описују три модела:

4. Доменски модел (***Domain Model - DM***) који представља поједностављену верзију UML дијаграма класа.
5. Модел случајева коришћења (***Use Case Model*** - ***UCM***) који служи за дефинисање и спецификацију случајева коришћења.
6. Модел прелаза стања (***State Transition Model*** - ***STM***) који служи за дефинисање дијаграма прелаза стања за сваки доменски објекат и дефинисање скупа случајева коришћења који се могу извршити над објектом у сваком од дефинисаних стања.

Наведени модели су међусобно конзистентни, што значи да се током ажурирања неког од модела непрекидно проверавају и усаглашавају концепти сва три модела. **SILAB-UCMDDM** метода користи две стратегије у развоју софтвера:

a) Стратегију засновану на случајевима коришћења. Случајеви коришћења се користе за спецификацију функционалних захтева система. Стратегија заснована на случајевима коришћења подразумева коришћење случајева коришћења кроз све фазе развоја софтвера.
b) Стратегију засновану на моделима и трансформацијама модела. Модел представља упрошћену, али субјективну слику једног дела или комплетног система, најчешће неког аспекта система (система који већ постоји или кога треба креирати на основу модела) посматран из угла посматрача. Због тога је за разумевање комплетног система најчешће потребно креирати више модела. Модел се увек изражава ограниченим скупом концепата. Ови концепти дефинисани су у одговарајућем језику за моделовање преко одговарајућег метамодела. Дакле, језик за моделовање представља вештачки језик који се користи за спецификацију знања и информација о систему на конзистентан и структурирани начин пратећи правила која су дефинисана преко метамодела. Моделовање представља процес креирања

модела система. Сам начин креирања модела система се дефинише конкретном методом моделовања која дефинише један могући начин моделовања система. Једна комплетна метода за моделовање система се састоји од језика за моделовање и поступка за моделовање. Поступак за моделовање представља упутство за коришћење језика како би се модел креирао на исправан начин. Дакле, моделовање се може дефинисати као процес који на основу улаза, који представља систем који се посматра, применом методе за моделовање наведеног процеса и језика за моделовање креира модел као излаз из наведеног процеса .

У том контексту *Silab-UCMDM* представља методу за спецификацију захтева система. У првом делу овог поглавља дат је приказ активности *Silab-UCMDM* методе, док је у наставку поглавља дата формална спецификација *UCDSL* језика.

4.1. АКТИВНОСТИ SILAB-UCMDDM МЕТОДЕ

Silab-UCMDM метода се дефинише помоћу:

a) Активности,
b) редоследа извршења активности и
c) дефинисања улазних и излазних докумената (артифаката) за сваку активност.

На слици (Слика 16) приказане су основне активности *Silab-UCMDM* методе.

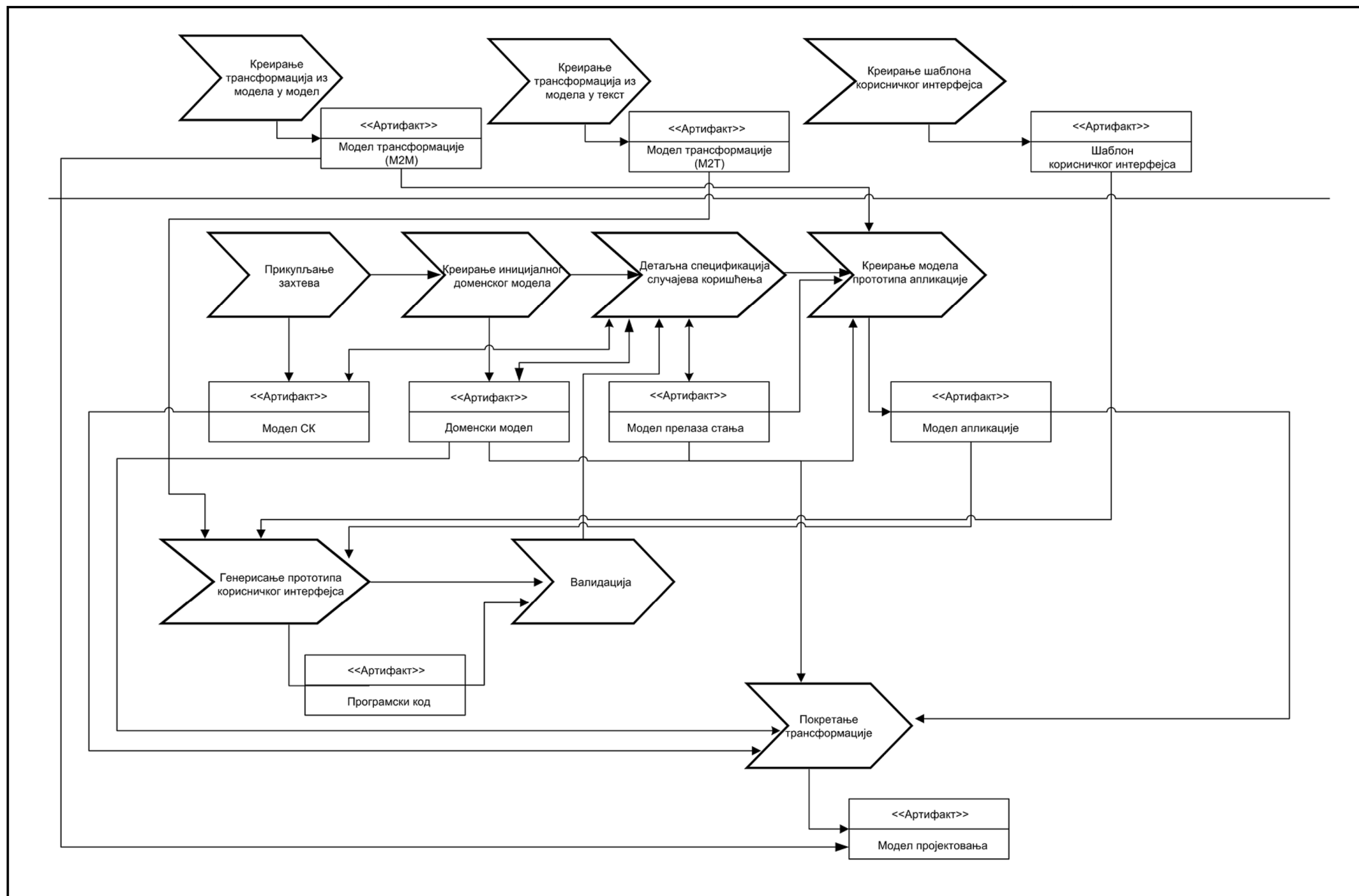
Прикупљање захтева представља прву активност предложене *Silab-UCMDM* методе. Током ове фазе врши се идентификација и попис свих случајева коришћења.

Након ове активности, прелази се у активност идентификовања иницијалног доменског модела. Дефинисање иницијалног доменског модела подразумева дефинисање основних доменских класа. Спецификација својстава и веза између доменских класа је опциона и она се врши упоредо са детаљном спецификацијом случајева коришћења (активност која следи).

Дефинисање иницијалног доменског модела и модела случајева коришћења представља предуслов за детаљну спецификацију случајева коришћења. У оквиру ове активности сваки идентификовани случај коришћења се детаљно специфицира према унапред дефинисаном обрасцу (темплејту). Током ове активности континуирано се проверава ускађеност доменског модела са моделом случаја коришћења. Зависност између доменског модела и модела случајева коришћења огледа се у томе што се акције случајева коришћења специфицирају у складу са доменским моделом односно концептима дефинисаним у доменском моделу.

Поред тога, у оквиру ове активности креира се модел прелаза стања. Моделом прелаза стања дефинишу се предуслови за извршење случајева коришћења и идентификују системске операције. За свако стање у коме се може наћи објекат доменске класе дефинише се најмање један случај коришћења. Извршењем случаја коришћења, односно извршењем неке од системских операција у оквиру случаја коришћења доменски објекат се преводи из једно у друго стање (могуће је да објекат остане у истом стању). Стога се почетни доменски модел којим се описује само структура система проширује системским операцијама којим се дефинише њихово понашање. Идентификовањем системских операција током спецификације случајева коришћења врши се континуирано усклашивање доменског модела и модела прелаза стања. Дакле, током ових активности континуирано се врши усклађивање три модела: доменског модела, модела случајева коришћења и модела прелаза стања.

Након усклађивања ова три модела креира се модел прототипа апликације. Овај модел се користи за генерисање прототипа апликације, који се као такав може користити за валидацију корисничких захтева. Овим моделом се дефинише модел навигације кроз апликацију тако што се идентификовани случајеви коришћења повезују са одговарајућим формама које представљају једну могућу реализацију случаја коришћења. Дефинисањем корисничког интерфејса за сваку екранску форму врши се његова интеграција са проширеним моделом корисничког интерфејса који је део SILAB-UI пројекта [ANTOVIĆ, I. et al. (2012)], [I.ANTOVIĆ (2015)].



Слика 16. Активности *Silab-UCMDM* методе

Након генерисања прототипа корисничког интерфејса и валидације захтева, једна итерација *Silab-UCMDM* методе се завршава трансформцијом модела захтева у модел пројектовања (у претходном поглављу дат је приказ ових трансформација). Треба нагласити да *Silab-UCMDM* метода користи итеративно-инкраментални модел, што значи да се активности током једне итерације могу више пута понављати (из једне активности се по потреби може вратити на неку од претходних активности).

Активности *Креирање трансформација из модела у модел*, *Креирање трансформација из модела у текст* и *Креирање шаблона корисничког интерфејса* представљају активности чији се излази користе као основа за имплементацију моделом вођеног развоја софтвера.

4.2. UCDSL – ЈЕЗИК ЗА СПЕЦИФИКАЦИЈУ ДОМЕНСКОГ МОДЕЛА, МОДЕЛА СЛУЧАЈА КОРИШЋЕЊА И МОДЕЛА ПРЕЛАЗА СТАЊА

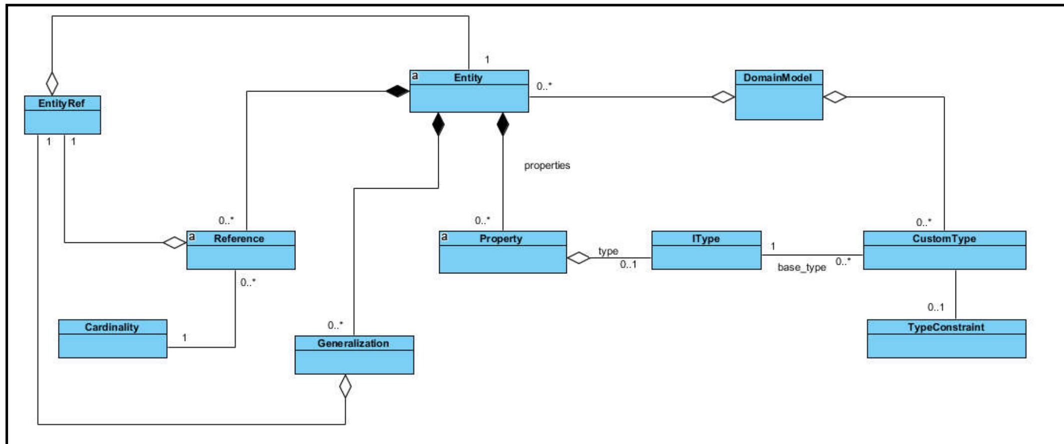
У делу који следи дата је формална спецификација предложеног UCDSL доменски-специфичног језика. Апстрактна синтакса језика је приказана преко UML дијаграма класа и/или *jetbrains.mps.baseLanguage.structure* језика (мета-језика), док је спецификација конкретне синтаксе приказана преко *jetbrains.mps.base.editor* језика (мета-језика) са конкретним примером спецификације (исказивања) модела преко тако дефинисане синтаксе.

4.2.1. СПЕЦИФИКАЦИЈА ДОМЕНСКОГ МОДЕЛА ПОМОЋУ UCDSL ЈЕЗИКА

Спецификација доменског модела у оквиру **Silab-UCMDM** методе посебно је значајна из разлога што се спецификација случаја коришћења у потпуности ослања на доменски модел. Наиме, концепти из доменског модела као што су доменске класе, атрибути, везе између доменских класа, ограничења и операције се користе при спецификацији случајева коришћења. Доменски модел који се дефинише (исказује) преко **UCDSL** језика представља упрошћену верзију UML дијаграма класа.

АПСТРАКТНА СИНТАКСА

`DomainModel` представља основни `(root)` концепт који се користи за дефинисања доменског модела. Ниже на слици (Слика 17) приказани су основни концепти **UCDSL** метамодела за дефинисање доменског модела (у даљем тексту ће се користити термин DOM метамодел за део **UCDSL** метамодела који се односи на доменски модел).



Слика 17. Основни концепти DOM метамодела

Дефинисање доменског модела подразумева:

a) дефинисање назива доменског модела
b) дефинисање и спецификацију доменских класа односно ентитета из домена проблема
c) дефинисање корисничких (*custom*) типова података.

Дефинисање доменских класа омогућено је `Entity` концептом. Овим концептом се дефинише елемент односно концепт који постоји у доменском моделу. `Entity` концепт из DOM метамодела одговара `Class` концепту из UML метамодела.

Дефинисање `Entity` концепта подразумева дефинисање назива концепта, својстава, веза ка другим доменским концептима, ограничења и операција. Дефинисање операција и ограничења се врши приликом спецификације случајева коришћења.

Концепт `Property` се дефинише навођењем:

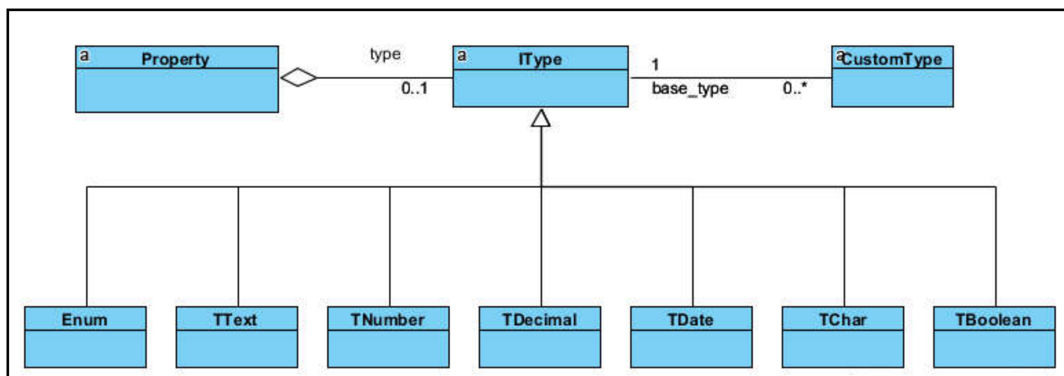
- *Назива својства*. Свако својство мора да има назив. У оквиру једне `Entity` класе сва својства се међусобно разликују на основу назива.
- *Типа својства*. Тип својства може бити један од постојећих простих типова података, кориснички дефинисани тип или енумерисани тип података.

- *Јединствености.* Означавањем да ли својство има карактеристику јединствености, односно да ли се објекти доменске класе јединствено идентификују на основу вредности дефинисаног својства.

У DOM метамоделу дефинисани су следеће основни (примитивни) типови података:

a) целобројни тип (`TNumber` концепт)
b) датумски тип (`TDate` концепт)
c) реални тип (`TDecimal` концепт)
d) текстуални тип (`TText` концепт)
e) знаковни тип (`TChar` концепт)
f) логички тип (`TBoolean` концепт).

Сваки тип података дефинисан је преко одговарајућег концепта у DOM метамоделу (Слика 18).



Слика 18. Концепти `Property` и `Itype` дефинисани преко *jetbrains.mps.baseLanguage.structure* језика

Поред унапред дефинисаних типова података у језику је омогућено и дефинисање корисничких типова података. Дефинисање корисничких типова података подразумева дефинисање назива корисничког типа, дефинисање основног типа, на основу кога се креира нови тип, и ограничења. Ограничења се дефинишу коришћењем природног текста или дефинисањем регуларног израза.

`Entity` концепт поред својстава садржи и везе ка другим `Entity` концептима. За разлику од UML метамодела у коме постоји више различитих типова веза између класа, у DOM метамоделу дефинисане су само веза

асоцијације и веза генерализације. Ове везе су дефинисане преко одговарајућих концепата `Association` и `Generalization`.

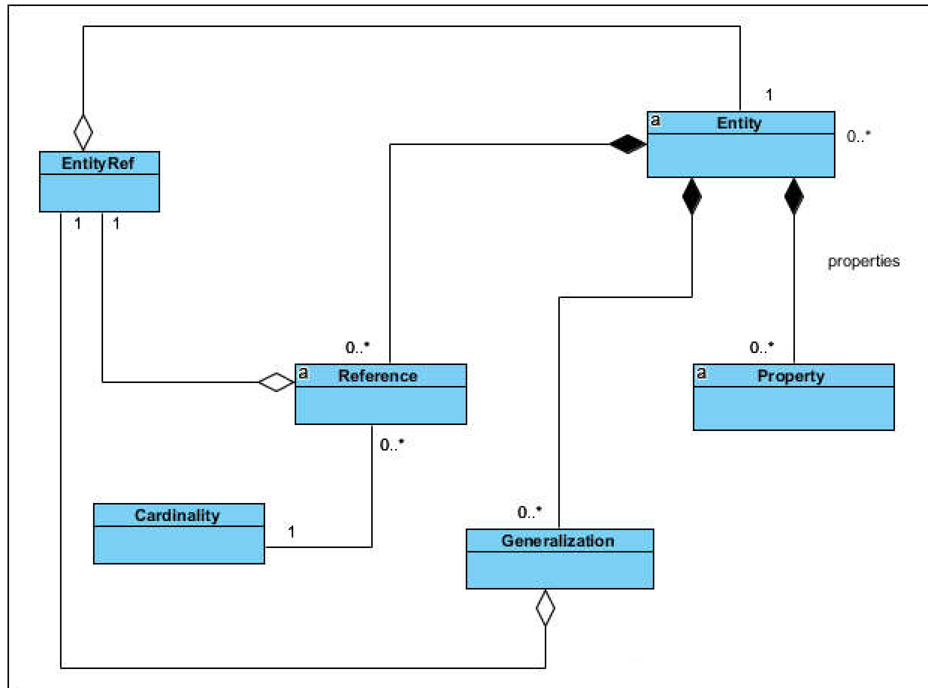
Веза се у DOM моделу увек моделује као бинарна (веза између два ентитета различитог типа). Спецификација унарне и тернарне везе у језику није директно омогућена, али се обе везе моделују као бинарне. Унарна веза се преводи у бинарну, тако што назив унарне везе (односно улога) прелази у нову посебну `Entity` класу која наслеђује `Entity` класу за коју се формира веза, након чега се дефинише бинарна веза између класе за коју се формира веза и посебне `Entity` класе. На пример: Уколико је дата унарна веза у којој један *Zaposleni* (класа за коју се формира веза) може да има једног надређеног (али може и више у зависности од модела), који је из реда запослених, тада ова веза прелази у бинарну везу *ZaposleniNadredjeni* (нова посебна класа), при чему *Nadredjeni* представља `Entity` класу која наслеђује класу *Zaposleni*.

Веза асоцијације се у доменском моделу увек почетно моделује у једном смеру (унидирекциона, *target-sorce*). Моделовање везе у оба смера (бидирекциона, *target-sorce* и *source-target*) углавном је одређено начином интеракције актора са системом, па се моделовање везе у оба смера врши у складу са спецификацијом случајева коришћења. Спецификација везе асоцијације стога захтева:

- Дефинисање назива асоцијације. Назив асоцијације треба дефинисати у складу са улогом коју `Entity` класе имају у вези.
- Дефинисање кардиналности ка `Entity` класи која учествује у вези. Кардиналност везе дефинише минимални и максимални број објеката `Entity` класе са којима један објекат неке класе може да буде у вези. Минимални број увек представља конкретан цео број (на пример 0..1, …, итд), док максимални број може да представља конкретан цео број или неодређени цео број (* - када максимални број није одређен).
- Дефинисање да ли асоцијација представља **`isPartOf`** асоцијацију или не. **`PartOf`** асоцијација се користи код дефинисања везе јаке зависности (UML веза композиције). Ова веза означава да класа која учестује у вези егзистенцијално зависи од класе за коју се дефинише асоцијација. Ова информација се касније користи и приликом дефинисања случајева

коришћења јер за објекте који учествују у асоцијацији није могуће дефинисати све типове случајева коришћења.

На слици (Слика 19) је приказан DOM метамодел који се односи на дефинисање веза између доменских класа.

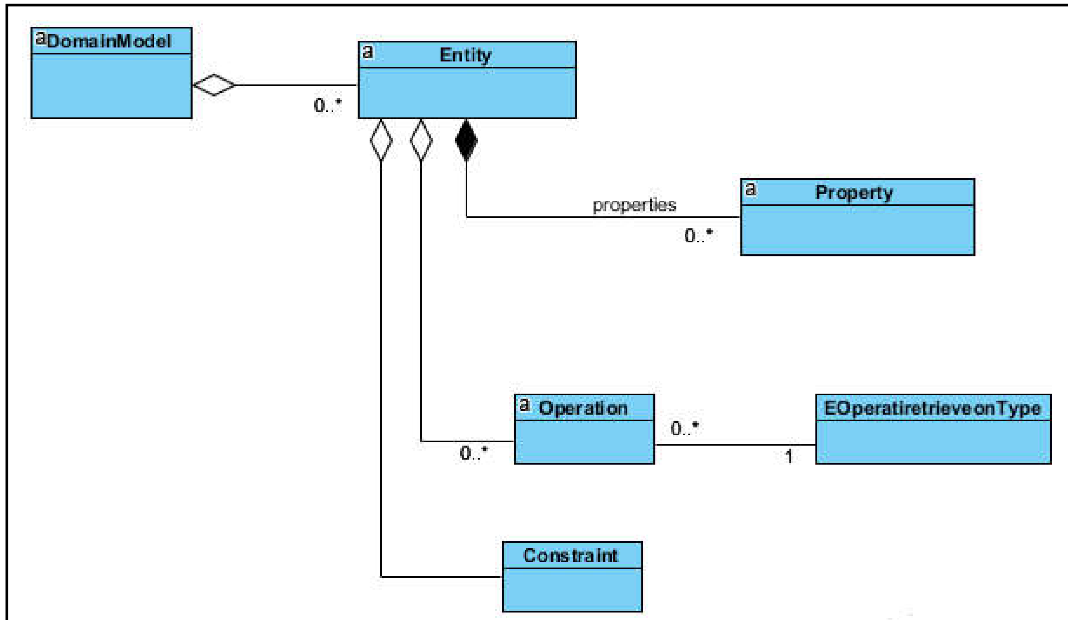


Слика 19. DOM метамодел: Концепт `Reference` и концепт `Generalization`

Концепт `Constraint` се користи за дефинисање ограничења, док се концептом `Operation` дефинишу операције. Дефинисање операције подразумева дефинисање назива операције и типа операција. Свака операција припада једном од унапред дефинисаних типова:

a) `save` – Регистровање новог објекта у систему.
b) `update` – Измена објекта који се налази у систему.
c) `delete` – Брисање објекта из система.
d) `retrieve` – Претрага објеката у систему.

На слици (Слика 20) је дат приказ метамодела за ова два концепта.

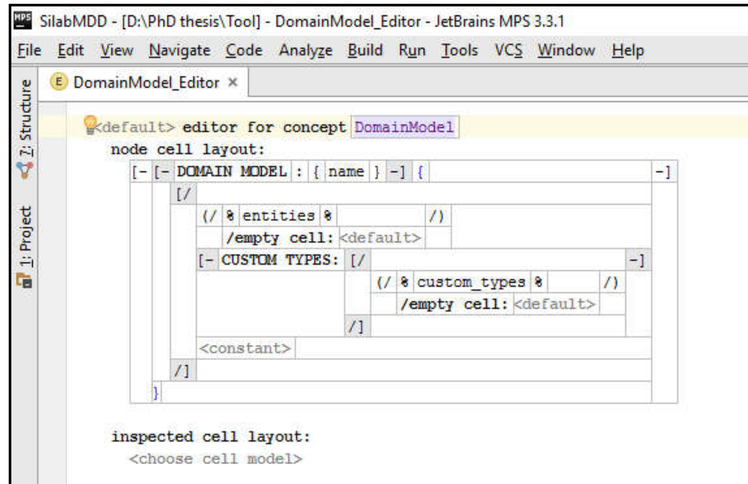


Слика 20. Концепти `Operation` и `Constraint`

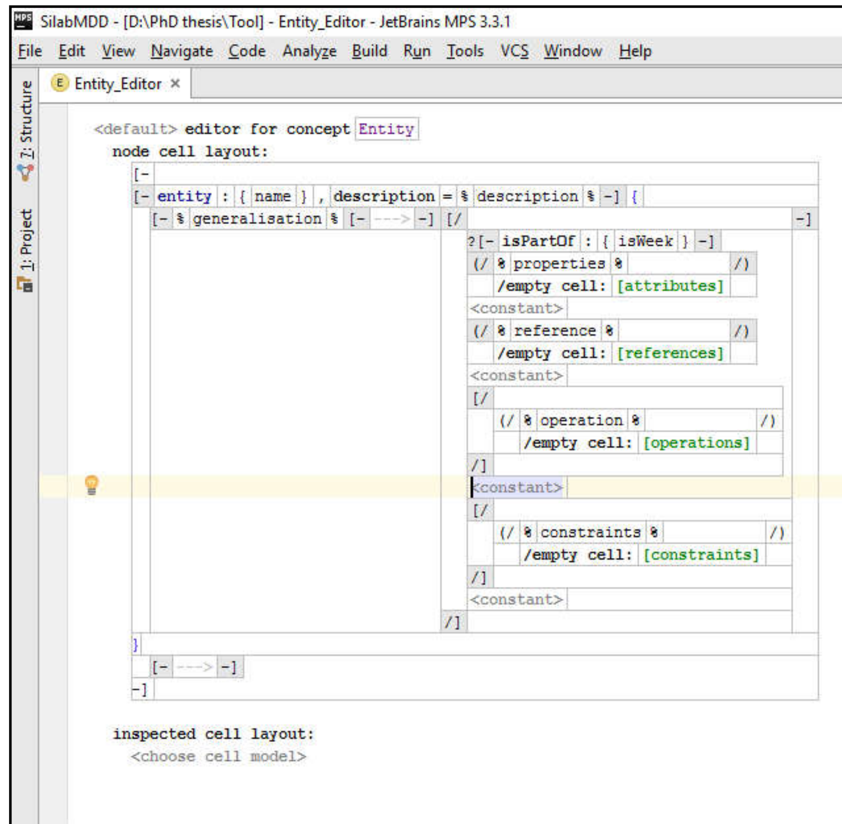
КОНКРЕТНА СИНТАКСА

Поред апстрактне синтаксе, дефинисање језика подразумева и дефинисање одговарајуће конкретне синтаксе и семантике језика. Конкретном синтаксом језика дефинише се начин представљања (приказивања) концепта који су дефинисани апстрактном синтаксом. Семантика језика је у оквиру *JetBrans MPS* алата извршена коришћењем *jetbrains.mps.lang.behavior*, *jetbrains.mps.lang.constraint*, *jetbrains.mps.lang.tupesystem* и *jetbrains.mps.lang.actions* језика.

На слици (Слика 21) је дат приказ начина представљања `DomainModel` концепта и `Entity` концепта (Слика 22) из UCDSL метамодела.



Слика 21. `DomainModel` концепт дефинисан преко *jetbrains.mps.lang.editor* језика



Слика 22. Entity концепт дефинисан преко *jetbrains.mps.lang.editor* језика

На слици (Слика 23) дат је пример спецификације *Invoice* и *Invoice item* доменских објекта исказаних преко UCDSL језика. Концепти *operation* и *constraints* идентификују се током спецификације случајева коришћења.

SilabMDD - [D:\PhD thesis\Tool] - ...\solutions\silad.dsl.MODEL\models\silab\dsl\SilabMDD\model.mps\DOModel - Billing system - JetBrains MPS 3.3.1

File Edit View Navigate Code Analyze Build Run Tools VCS Window Help

DOModel - Billing system

```
}

entity : invoice , description = "Document created from our syste" {
  <no generalisation>  attribute : invoice_id_created_by , type = ENUM can be: [ "system generate" OR "select from list reserved number" ] , identificator = false , description = <no de
                       attribute : id , type = NUMBER , identificator = true , description = " "
                       attribute : date_created , type = DATE , identificator = false , description = <no desciption>
                       attribute : total_value , type = DECIMAL , identificator = false , description = <no desciption>
                       attribute : vat_total_value , type = DECIMAL , identificator = false , description = <no desciption>
                       attribute : discount , type = DECIMAL , identificator = false , description = <no desciption>
                       attribute : reserved_invoice_id , type = NUMBER , identificator = false , description = <no desciption>
                       attribute : reserved_date , type = DATE , identificator = false , description = <no desciption>

                       reference : customer , targetEntity = customer , isPartOf = false , cardinality ( min = " 1 " , max = " 1 " )
                       reference : invoice_items , targetEntity = invoice_items , isPartOf = true , cardinality ( min = " 0 " , max = " * " )
                       reference : reserved_invoice , targetEntity = reserved_invoice , isPartOf = false , cardinality ( min = " 0 " , max = " 1 " )

                       operation : save_invoice , type = save , description = "Save new invoice in the system"
                       operation : save_invoice_and_sent_to_approve , type = save , description = "Save and send to approve"
                       operation : update_invoice , type = update , description = "Update invoice"
                       operation : sent_to_approve , type = update , description = "Send to approve"
                       operation : update_and_sent_to_approve , type = save , description = "Update invoice and send to approve"
                       operation : approve_invoce , type = update , description = "Approve invoice"
                       operation : reject_invoice , type = update , description = "Reject invoice"
                       operation : search_invoice_by_id , type = retrieve , description = "Search invoice by ID"
                       operation : search_invoice_advanced , type = retrieve , description = "Search invoice by criteria"
                       operation : send_invoice , type = update , description = "Send invoice to customer"

                       constraint : constr-generated_new_invoice_by_system , description = "System generate new invoice number"
                       constraint : constr-user_select_from_list , description = "New invoice number is selected from list"
                       constraint : constr-customer-VIP , description = "Customer for which invoice is created is VIP"

}

entity : invoice_items , description = <no description> {
```

Console 3: Usages 0: Messages 9: Version Control Event Log 2: Inspector

SilabMDD - [D:\PhD thesis\Tool] - ...\solutions\silad.dsl.MODEL\models\silab\dsl\SilabMDD\model.mps\DOModel - Billing system - JetBrains MPS 3.3.1

File Edit View Navigate Code Analyze Build Run Tools VCS Window Help

DOModel - Billing system

```
                       constraint : constr-user_select_from_list , description = "New invoice number is selected from list"
                       constraint : constr-customer-VIP , description = "Customer for which invoice is created is VIP"

}

entity : invoice_items , description = "Invoice item" {
  <no generalisation>  attribute : item_no , type = NUMBER , identificator = false , description = " "
                       attribute : quantity , type = NUMBER , identificator = false , description = <no desciption>
                       attribute : unitary_value , type = DECIMAL , identificator = false , description = <no desciption>
                       attribute : item_value , type = DECIMAL , identificator = false , description = <no desciption>
                       attribute : vat_item_value , type = DECIMAL , identificator = false , description = <no desciption>

                       reference : product_on_invoice_item , targetEntity = product , isPartOf = false , cardinality ( min = " 1 " , max = " 1 " )

                       [operations]

                       [constraints]

}
```

Слика 23. Пример спецификације доменског објекта *Invoice*

4.2.2 СПЕЦИФИКАЦИЈА МОДЕЛА СЛУЧАЈА КОРИШЋЕЊА ПОМОЋУ UCDSL ЈЕЗИКА

Спецификација случајева коришћења у оквиру **Silab-UCMDM** методе се заснива на спецификацији три модела: **доменског модела**, **модела случаја коришћења** и **модела прелаза стања**. У претходном делу овог поглавља дат је приказ основних концепата језика који се користе за дефинисање доменског модела. У овом делу поглавља биће дат приказ основних концепата језика који се користе за дефинисање модела случајева коришћења, док су у секцији 6.2.3. овог поглавља приказани основни концепти језика који се користе за дефинисање модела прелаза стања.

Дефинисања модела случајева коришћења (**UCM-модел**) подразумева:

a) дефинисање актора система,
b) дефинисање случајева коришћења и
c) спецификацију дефинисаних случајева коришћења.

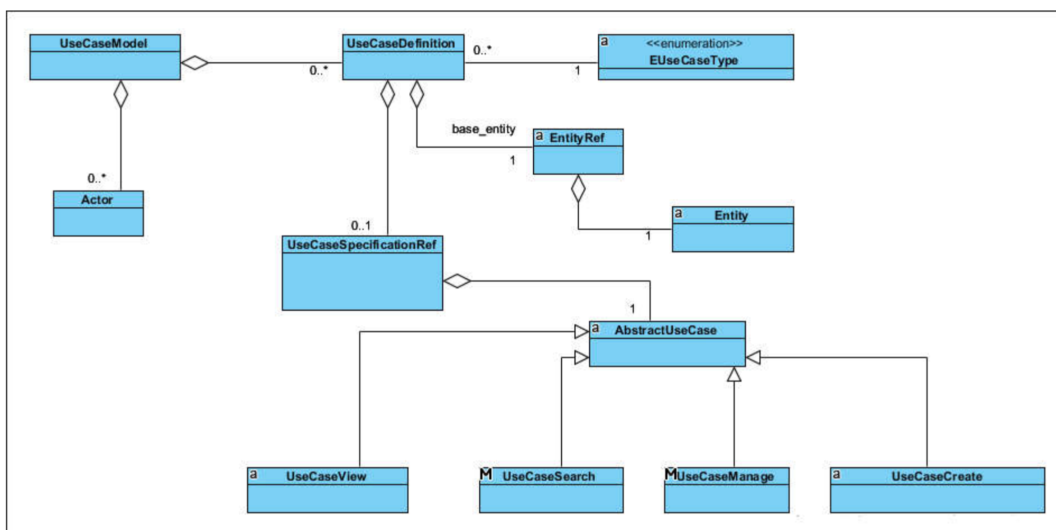
`UseCaseModel` представља главни `(root)` концепт која се користи за дефинисања модела случаја коришћења.

Дефинисање модела случаја коришћења подразумева:

a) *Дефинисање назива модела*. Назив модела представља назив система који се моделује.
b) *Дефинисање листе актора*. Спецификација корисничких захтева преко случајева коришћења подразумева откривање и дефинисање листе актора. За сваког актора се дефинише назив односно улога и кратак опис.
c) *Дефинисање листе случаја коришћења*. Спецификацији случајева коришћења претходи њихово идентификовање односно откривање. Без обзира на начин на који су случајеви коришћења идентификовани (да ли на основу примитивних процеса CCA, или на основу вербалног описа система или на неки трећи начин), пре саме спецификације потребно их је формално дефинисати. Дефинисање једног случаја коришћења подразумева дефинисање назива случаја коришћења, дефинисање актора (једног или више

актора), дефинисање доменске класе над којом се извршава конкретан случај коришћења и дефинисање типа случаја коришћења.

На слици (Слика 24) дат је метамодел за овај део **UCDSL** језика .



Слика 24. Концепт `UseCaseModel`

Један случај коришћења може бити коришћен од стране два или више актора. Дефинисање актора у случају коришћења подразумева референцирање на једног или више актора из дефинисане листе актора. Уколико не постоји разлика у начину коришћења система за посматрани случај коришћења од стране ових актора, тада је довољно специфицирати један случај коришћења за све акторе за тај случај коришћења.

Дефинисање доменске класе над којом се извршава конкретан случај коришћења захтева да доменска класа претходно буде идентификована у доменском моделу. Дефинисање доменске класе над којом се случај коришћења извршава битан је из разлога утврђивања конзистентности доменског модела и модела случаја коришћења. У том смислу, у самом језику дефинисана је веза референцирања; из случаја коришћења врши се референцирање на доменску класу за коју се дефинише случај коришћења.

Дефинисање случаја коришћења подразумева дефинисање типа случаја коришћења. У оквиру **UCDSL** језика сваки случаја коришћења припада неком од

унапред дефинисаних типова случаја коришћења. Један случај коришћења може да припада само једном типу. Као што је наглашено у раду *„A requirements description meta-model for use cases“* [NAKATANI et al. (2001)] основне операције које преовлађују у пословним информационим системима јесу операције чувања *(storage),* претраге *(retrieval),* ажурирања *(updating)*, брисања *(deleting)* и операције које се односе на извештавање. Слично, у оквиру Silab-UCMDM методе у најширем смислу идентификовани су следећи типови случајева коришћења: *create*, *search*, *manage*, *delete*, *update* и *report*. И поред тога што је језиком омогућена спецификација случајева коришћења типа *update* и *delete*, треба имати на уму препоруку која се односи на спецификацију случајева коришћења овог типа, а која се огледа у следећем: спецификацију случајева коришћења типа *delete* и *update* треба посматрати као специфичан случај *manage* типа случаја коришћења, па није грешка (чак је и препорука) да се случајеви коришћења овог типа означе као *manage*. Један од основних разлога због чега треба следити ову препоруку јесте да се смањи број случајева коришћења које је потребно специфицирати у процесу спецификације корисничких захтева, а да се том приликом ништа не изгуби. Тако се најчешће за један доменски објекат дефинише само један случај коришћења типа *manage*, а затим се у том случају коришћења дефинишу све операције које се могу извршити над тим објектом (на пример операција брисања, операција ажурирања и слично...).

Након дефинисања случаја коришћења врши се његова спецификација. Дефинисање сценарија случаја коришћења у великој мери је одређено типом случаја коришћења. Концептом `EUseCaseType` дефинисани су типови случајева коришћења. Сваки случај коришћења независно од типа коме припада садржи назив, тип и везу ка доменској класи за коју се специфицира случај коришћења. Због тога је у **UCDSL** мета-моделу уведен апстрактни концепт `UseCaseAbstract.` Како би се омогућила спецификација различитих типова случајева коришћења (`create`, `search`, `manage`, `update` или `delete`) за сваки од наведених типова случајева коришћења уведени су конкретни концепти. Спецификација случајева коришћења типа `create` омогућена је увођењем концепта `CreateUseCase`, док је спецификација случајева коришћења типа `search` омогућена увођењем концепта `SearchUseCase`. За случајеве коришћења типа `manage` креиран је концепт `ManageUseCase`, док је за случај коришћења типа delete креиран `DeleteUseCase`

концепт. Сваки од конкретних концепата за спецификацију случајева коришћења наслеђује апстракни `UseCaseAbstract` концепт (Слика 24).

`AbstractUseCase` концептом дефинише се:

a) *Тип случаја коришћења*. Сваки тип случаја коришћења припада само једном дефинисаном типу
b) *Опис случаја коришћења.* Сваки случај коришћења може да садржи кратак опис
c) *Листа актора.* Један случај коришћења може да изврши један или више актора.
d) *Доменска класа.* Доменска класа за коју се дефинише случај коришћења. Један случај коришћења се увек извршава над једном доменском класом коју називамо главна (односно примарна). Примарна доменска класа претходно мора бити дефинисана у доменском моделу.

Сценарио случаја коришћења се састоји од скупа корака унутар којих су дефинисане одговарајуће акције актора и система. Ови кораци су у циљу формалније спецификације груписани унутар такозваних блокова, тако да спецификација сценарија случаја коришћења заправо представља спецификацију ових блокова. У зависности од типа интеракције између актора и система у **UCDSL** језику дефинисани су следећи типови блокова:

- Блок за унос/измену података о доменском ентитету.
- Блок за приказ података о доменском ентитету.
- Блок за претрагу доменског ентитета.
- Блок за дефинисање системских операција.

4.2.2.1. СПЕЦИФИКАЦИЈА СЛУЧАЈА КОРИШЋЕЊА ТИПА CREATE

АПСТРАКТНА СИНТАКСА

Концепт `UseCaseCreate` се користи за спецификацију случајева коришћења типа `create`. Спецификација случаја коришћења типа `create` подразумева дефинисање главног сценарија случаја коришћења и алтернативног сценарија (односно сценарија грешке). Концепт `UseCaseCreateFlow` се користи за дефинисање главног сценарија, док се сценарио грешке дефинише преко `ExceptionalFlow` концепта.

У оквиру спецификације случаја коришћења овог типа могуће је дефинисати и именована сценарија за спецификацију акција у којима актор припрема податке за извршење системске операције. Концепт `UseCaseCreateSubFlow` се користи за дефинисање ових именованих сценарија. Именована сценарија се користе код спецификација акција које се извршавају под одређеним условима (дефинисаним ограничењима). Услов под којим се извршава нека акција дефинише се преко ограничења, док се сценарио извршења у случају испуњења услова дефинисаног ограничењем описује преко именованог сценарија (у примеру (Слика 29) при спецификацији случаја коришћења *UC-create-invoice*, дефинисано је ограничење *VIP customer* и именовани сценарио *VIP customer*).

Сценарио случаја коришћења чине два блока:

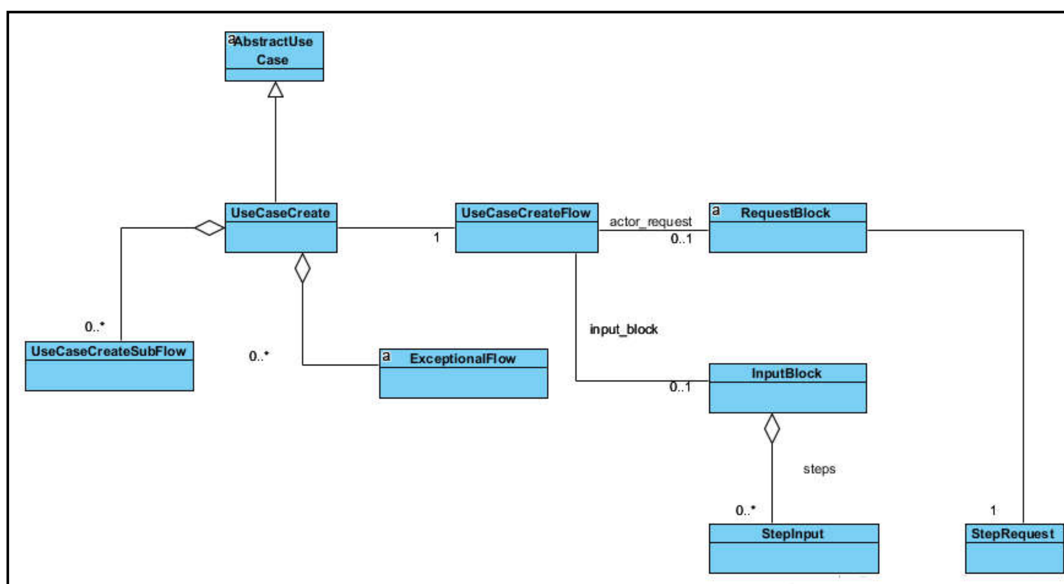
1) Улазни блок који је дефинисан преко `InputBlock` концепта. У оквиру овог блока груписане су акције актора и система којима се припрема улаз за извршење системске операције.
2) Блок за спецификацију системских операција који је дефинисан преко `RequestBlock` концепта. У оквиру овог блока дефинише се скуп системских операција које актор може да позове у конкретном случају коришћења (у оквиру блока може бити дефинисана једна или више системских операција).

У оквиру блока за унос података (дефинисан преко `InputBlock` концепта) дефинишу се кораци (сваки корак се дефинише преко концепта `StepInput`) у оквиру којих се спецификирају улазни подаци актора и система. Улазни подаци актора и система представљају улаз за извршење системске операције. Подаци

које уноси актор и систем се дефинишу у пару, што значи да након спецификације података које уноси актор, могуће је дефинисати податке које аутоматски поставља систем. То је чест случај у пословним информационим системима. У том смислу неколико варијанти овог корака се могу наћи у спецификацији случаја коришћења типа `create`:

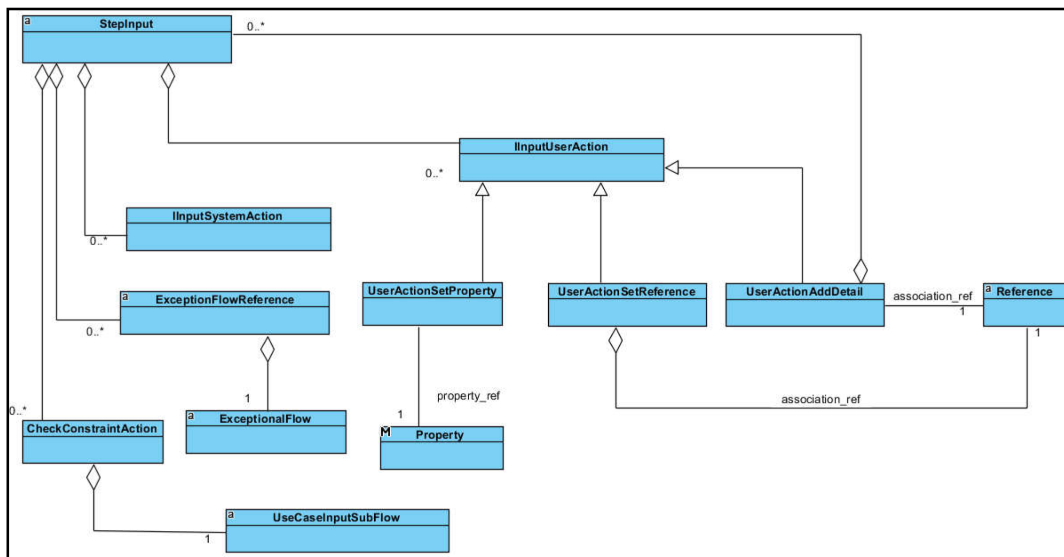
a) Корак у коме актор уноси податке на које систем не реагује.
b) Корак у коме актор уноси податке на који систем реагује тако што поставља одређене вредности аутоматски.
c) Корак у коме систем аутоматски поставља одређене податке (у односу на претходну акцију, нема непосредног уноса података од стране актора)
d) Корак у коме актор уноси податке на основу чега се проверавају одређена ограничења (једно или више) и у зависности од тога дефинише даљи ток извршења сценарија
e) Корак у коме се проверавају одређена ограничења (једно или више) и у зависности од тога дефинише даљи ток извршења сценарија.

На слици (Слика 25) је дат приказ једног дела метамодела који се односи на `UseCaseCreate` концепт.



Слика 25. Концепт `UseCaseCreate`

У оквиру корака за спецификацију улазних података (`StepInput` концепт) акцију припреме података за извршење системске операције може извршити актор или систем. На слици (Слика 26) је дат приказ једног дела метамодела који се односи на `StepInput` концепт.



Слика 26. Концепт `Stepinput`

У том смислу дефинисане су акције које извршава актор (дефинисане су концептом `IInputUserAction`) и акције које извршава систем (дефинисане су концептом `IInputSystemAction`). У зависности од тога да ли актор или систем уноси податке који се односе на својство, на везу која је јака (UML веза композиције) и везу која је слаба (UML веза асоцијације или агрегације) дефинисане су одговарајуће акције:

а) Акција SET-PROPERTY (дефинисана `UserActionSetProperty` и `SystemActionSetProperty` концептом). Овом акцијом специфицира се вредност коју корисник уноси, а која се односи на неко својство неке доменске класе. Својство за које корисник уноси вредност мора бити дефинисано у оквиру доменске класе над којом се конкретни случај коришћења извршава или у некој другој класи са којом ова доменска класа (над којом се случај коришћења извршава) остварује везу.

На пример: Уколико у доменском моделу постоји класа *Производ*, корисник овом акцијом специфицира вредности које се уносе за

својства: *назив производа*, *опис*, *датум производње*, *рок важења производа* и слично.

- Акција SET-REFERENCE (дефинисане `UserActionSelectReference` и `SystemActionSetReference` концептом). Овом акцијом специфицира се веза коју корисник креира између два објекта конкретних доменских класа. У зависности од типа везе (само за асоцијације које нису означене као `partOf`) и кардиналности везе која постоји између две доменске класе корисник може да изабере један (ако је кардиналност 0..1) или више објеката (ако је кардиналност 0..*).

  На пример: Уколико у доменском моделу постоји класа *Производ*, корисник овом акцијом специфицира вредности које корисник уноси за: *произвођача* (за конкретан производ треба изабрати једног произвођача ако се у систему чувају подаци о произвођачима), јединицу мере и слично. Састав неког производа се може специфицирати акцијом PUT (тада се састав производа уноси као текст), или акцијом SET када се детаљно врши избор свих сировина које улазе у састав производа.
- Акција ADD-DETAIL (дефинисана `UserActionAddDetail` и `SystemActionAddDetail` концептом). Овом акцијом специфицира се скуп вредности које корисник уноси о објекту доменске класе која са другом класом остварује везу која је дефинисана преко везе асоцијације типа `partOf`.

  На пример: Уколико у доменском моделу постоји класа `Racun`, она са класом `StavkaRacuna` успоставља везу аосцијације типа `partOf` (UML веза композиције). Акцијом ADD-DETAIL групишу се подаци које корисник уноси при уносу ставке рачуна. Обично је за везу асоцијације типа `partOf` дефинисана кардиналност (0..*) или (1..*), мада постоје и случајеви када је кардиналност (0..1) или (1).
- Акција провере ограничења (дефинисана преко `CheckConstraintAction` концепта). У акцијама овог типа врши се провера испуњености дефинисаног ограничења и дефинишу следећи кораци извршења случаја коришћења (преко именованог сценарија) било да је ограничење задовољено или не.

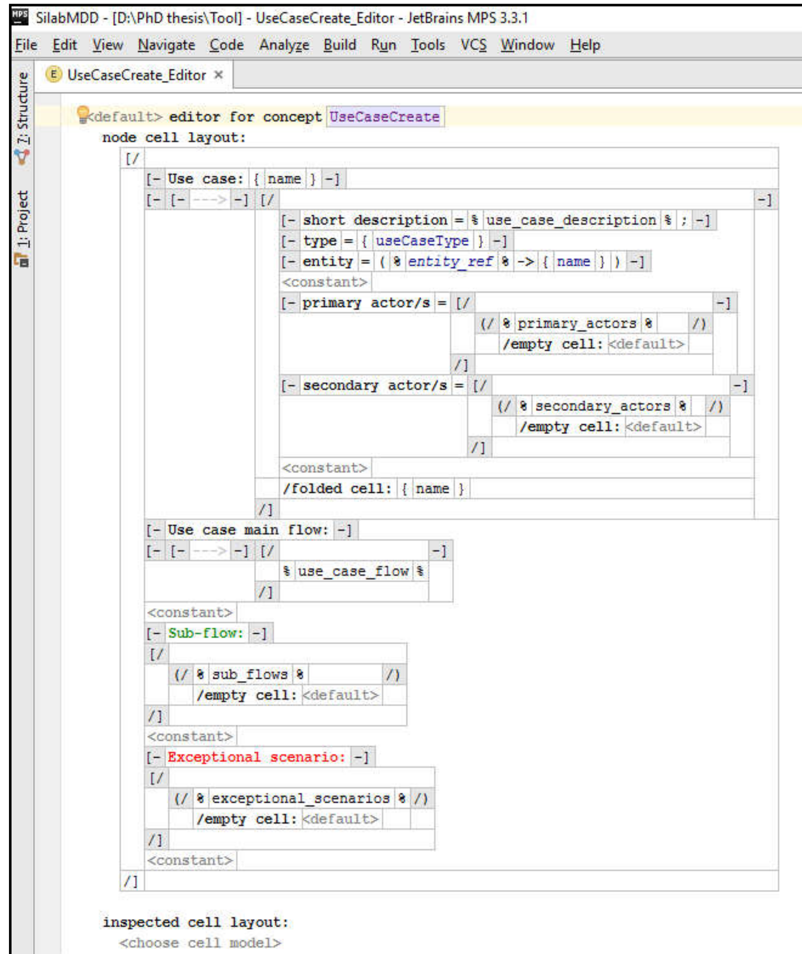
На пример: У систему електронске писарнице приликом завођења новог предмета, нови број предмета може да се добије на два начина: 1) нови број се бира из листе резервисаних бројева или, 2) нови број се аутоматски додељује од стране система. У зависности од тога на који начин се нови број додељује предмету, зависи и даља интеракција корисника са системом. Слично, у систему електронске евиденције правних лица, уколико је правно лице ПДВ обвезник тада се за то правно лице приликом регистрације уноси један скуп додатних података, који се не уноси приликом регистрације правног лица које није ПДВ обвезник. За овакве и сличне примере у **UCDCL** језику користи се овај тип акције.

За разлику од програмских језика опште намене у оквиру којих постоје алгоритамске структуре (линијске, разгранате и цикличне) у **UCDCL** језику оне нису експлицитно дефинисане. Заправо, акција дефинисања ограничења на вредност представља замену за разгранату структуру. Циклична структура не постоји у **UCDCL** језику из разлога што је унос у доброј мери дефинисан доменским моделом, тако да се на основу кардиналности везе између две класе може утврдити да ли се нека акција извршава само једном или више пута.

Једна од основних разлика у спецификацији акција које извршава актор од акција које извршава систем јесте у томе што је приликом спецификације акција које извршава систем потребно дефинисати начин на који систем поставља одређену вредност.

КОНКРЕТНА СИНТАКСА

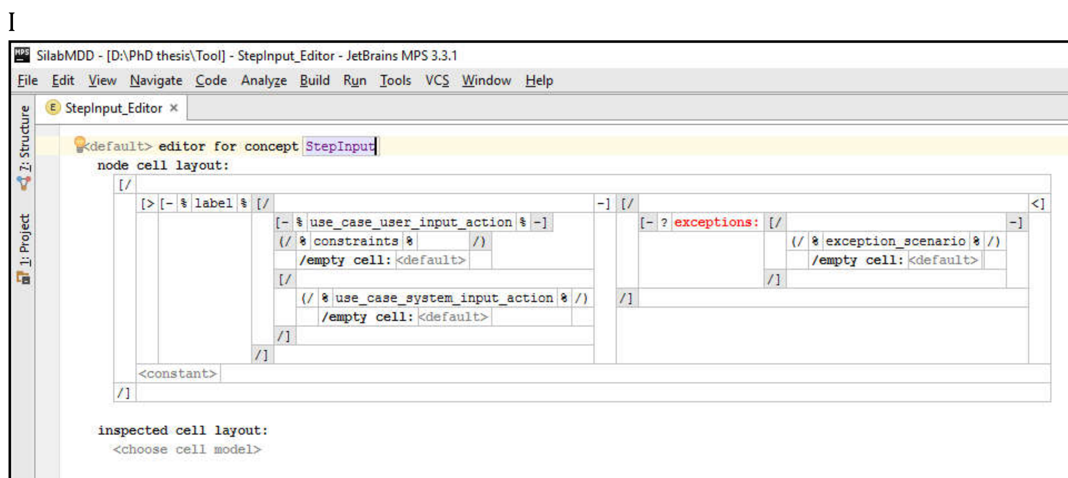
Ниже на слици (Слика 27) дат је приказ реализације конкретне синтаксе за `UseCaseCreate` концепт дефинисан преко *jetbrains.mps.lang.editor* језика.



Слика 27. `UseCaseCreate` концепт дефинисан преко *jetbrains.mps.lang.editor* језика

На слици (Слика 28) дат је приказ реализације конкретне синтаксе за `StepInput` концепт дефинисан преко *jetbrains.mps.lang.editor* језика.

I



Слика 28. `StepInput` концепт дефинисан преко *jetbrains.mps.lang.editor* језика

На слици (Слика 29) дат је пример спецификације *UC-create-invoice* исказан преко UCDSL језика.

```
Use case: UC-create-invoice
  short description = <no use_case_description> ;
  type = create-entity
  entity = invoice

  primary actor/s = << ... >>
  secondary actor/s = << ... >>

Use case main flow:
  "1." [actor SET-PROPERTY invoice_id_created_by                                                      ]<< ... >>
       [check-constraint: "New invoice number is selected from list" sub-flow: VIP customer           ]
       [check-constraint: "System generate new invoice number" sub-flow: system create new invoice number]
       [<< ... >>                                                                                      ]

  "2." [actor SELECT-REFERENCE {                  ]exceptions: customer_not_exist
       [  customer ( customer )                   ]
       [  cardinality ( min = " 1 " , max = " 1 " )]
       [}                                         ]
       [<< ... >>                                 ]
       [<< ... >>                                 ]

  "3." [actor ADD-DETAIL {               ]<< ... >>
       [        ...                      ]
       [      }                          ]
       [<< ... >>                        ]
       [system SET-PROPERTY vat_total_value]
       [  ( "SUM vat item value" )       ]
       [system SET-PROPERTY total_value  ]
       [  ( "SUM vat value" )            ]

  "4." [<no use_case_user_input_action>                                                   ]<< ... >>
       [check-constraint: "Customer for which invoice is created is VIP" sub-flow: VIP customer]
       [<< ... >>                                                                          ]

  "5." [case "Save new invoice in the system" : actor send request to system to execute save_invoice operation]
       [|                                                                                                      ]

Sub-flow:
# system create new invoice number
Step: <no step_ref>
  << ... >>
# VIP customer
Step: "4."
  "4.1." [actor SET-PROPERTY discount]<< ... >>
         [<< ... >>                  ]
         [<< ... >>                  ]


Exceptional scenario:
# customer_not_exist description = "Customer does not exist" terminate use case
```

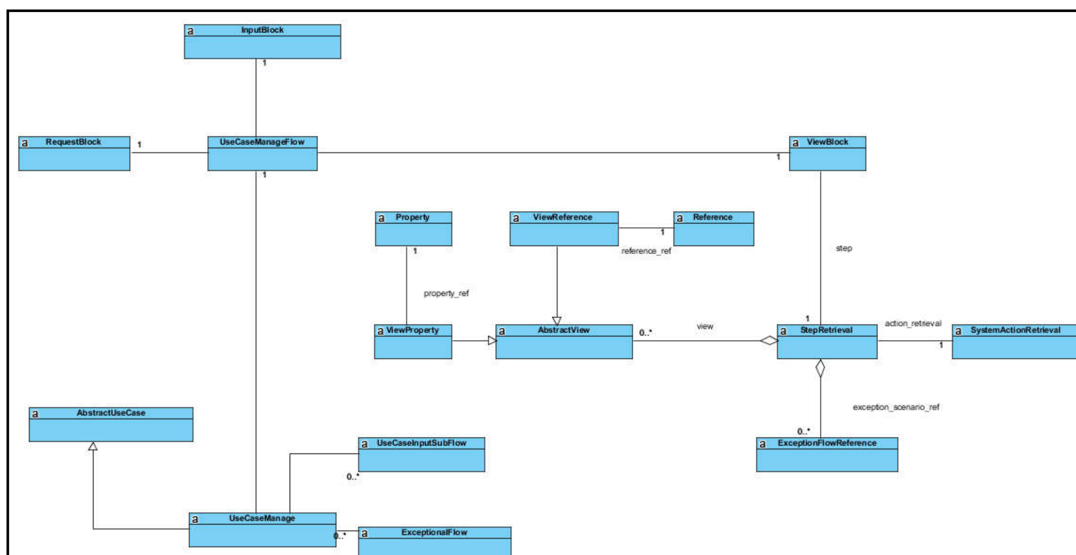
Слика 29. Спецификација *UC-create-invoice* случаја коришћења

4.2.2.2. СПЕЦИФИКАЦИЈА СЛУЧАЈА КОРИШЋЕЊА ТИПА MANAGE

АПСТРАКТНА СИНТАКСА

Концепт `UseCaseManage` се користи за спецификацију случајева коришћења типа `manage.` Спецификација случаја коришћења типа `manage` подразумева дефинисање главног сценарија случаја коришћења и сценарија грешке. Концепт `UseCaseManageFlow` се користи за дефинисање главног сценарија, док се сценарио грешке дефинише преко `ExceptionalFlow` концепта. Слично као и код спецификације случаја коришћења типа `create` могуће је дефинисати и именована сценарија за спецификацију акција у којима актор припрема податке за извршење системске операције.

Спецификација сценарија за случај коришћења типа `manage` подразумева дефинисање блока који служи за приказ доменског ентитета који се мења, спецификацију акција за измену постојећих података и спецификацију блока у коме су дефинисане системске операције које актор може да позове у посматраном случају коришћења. На слици (Слика 30) дат је приказ метамодела за `UseCaseManage` концепт.



Слика 30. UCDSL метамодел: Концепт `UseCaseManage`

КОНКРЕТНА СИНТАКСА

На слици (Слика 31) дат је приказ реализације конкретне синтаксе за `UseCaseManage` концепт дефинисан преко *jetbrains.mps.lang.editor* језика.

```
SilabMDD - [D:\PhD thesis\Tool] - UseCaseManage_Editor - JetBrains MPS 3.3.1
File Edit View Navigate Code Analyze Build Run Tools VCS Window Help
UseCaseManage_Editor ×

<default> editor for concept UseCaseManage
  node cell layout:
    [/
      [- Use case: { name } -]
      [- [- ---> -] [/
                      [- short description = % use_case_description % ; -]
                      [- type = { useCaseType } -]
                      [- entity = ( % entity_ref % -> { name } ) -]
                      <constant>
                      [- primary actor/s = [/
                                             (/ % primary_actors %    /)
                                               /empty cell: <default>
                                           /] -]
                      [- secondary actor/s = [/
                                               (/ % secondary_actors %    /)
                                                 /empty cell: <default>
                                             /] -]
                      <constant>
                    /folded cell: { name }
                    /] -]
      [- Use case main flow: -]
      [- [- ---> -] [/
                      % manage_flow %
                    /] -]
      <constant>
      [- Sub-flow: -]
      [/
        (/ % sub_flows %    /)
          /empty cell: <default>
      /]
      <constant>
      [- Exceptional scenario: -]
      [/
        (/ % exceptional_scenarios % /)
          /empty cell: <default>
      /]
      <constant>
    /]

  inspected cell layout:
    <choose cell model>
```

Слика 31. `UseCaseManageFlow` концепт дефинисан преко *jetbrains.mps.lang.editor* језика

На слици (Слика 32) дат је пример спецификације случаја коришћења исказан преко UCDSL језика, док је на слици (Слика 33) дат приказ корака за спецификацију акција додавања ставки рачуна (на претходној слици тај детаљ је сакривен (*folding* опција, знак + у самом едитору)).

```
Use case: UC-manage-invoice
  short description = "Update invoice" ;
  type = manage-entity
  entity = invoice

  primary actor/s = user-operater
  secondary actor/s = << ... >>

Use case main flow:
  "1." system RETRIEVAL-entity invoice                                   exception scenario: invoice_not_exist
       system DISPLAY-ENTITIES with the following details:
         id
         date_created
         reference customer customer_id
                            customer_name
                            reference customer_address street_name
                                                       door_number
                                                       postal_code
                                                       locality
                                                       country
         reference invoice_items item_no
                                 reference product_on_invoice_item product_id
                                                                   product_name
                                                                   reference product_vat_category vat_category_value
                                 unitary_value
                                 quantity
                                 item_value
                                 vat_item_value
         total_value
         vat_total_value

  "2." actor ADD-DETAIL {                      << ... >>
              ...
            }
       << ... >>
       system SET-PROPERTY vat_total_value
         ( "SUM vat item value" )
       system SET-PROPERTY total_value
         ( "SUM vat value" )

  "3." case "Update invoice" : actor send request to system to execute update_invoice operation

       case "Update invoice and send to approve" : actor send request to system to execute update_and_sent_to_approve operation

Sub-flow:
<< ... >>

Exceptional scenario:
# invoice_not_exist description = "Invoice not exist" terminate use case

Console   3: Usages   0: Messages   9: Version Control   Node Explorer
```

Слика 32. Спецификација случаја коришћења измена рачуна *(UC-manage-invoice)*

```
"2." actor ADD-DETAIL {
         invoice_items
         cardinality ( min = " 0 " , max = " * " )
         USE-ENTITIES: invoice_items
                       product
                                      "2.1" <no use_case_user_input_action> << ... >>
                                            << ... >>
                                            system SET-PROPERTY item_no
                                              ( "automaticaly +1" )

                                      "2.2" actor SELECT-REFERENCE {                                   << ... >>
                                              product_on_invoice_item ( product )
                                              cardinality ( min = " 1 " , max = " 1 " )
                                            }
                                            << ... >>
                                            system SET-PROPERTY unitary_value
                                              ( as: unitary_value = [ product_on_invoice_item . product_unitary_value ] )

                                      "2.3" actor SET-PROPERTY quantity                                                             <<
                                            << ... >>
                                            system SET-PROPERTY item_value
                                              ( as: item_value = quantity * unitary_value )
                                            system SET-PROPERTY vat_item_value
                                              ( as: vat_item_value = quantity * unitary_value * [ product_on_invoice_item . [ product_vat_category . vat_category_value ] ] )
       }
     << ... >>
     system SET-PROPERTY vat_total_value
       ( "SUM vat item value" )
     system SET-PROPERTY total_value
       ( "SUM vat value" )
```

Слика 33. Спецификација корака за додавање стаки рачуна *(ADD-DETAIL invoice_items)*

4.2.2.3. СПЕЦИФИКАЦИЈА СЛУЧАЈА КОРИШЋЕЊА ТИПА UPDATE И DELETE

Случајеви коришћења типа `update` и `delete` се специфицирају на основу шаблона који је дефинисан за случај коришћења типа `manage`. Спецификација ова два типа случаја коришћења се не разликује од спецификације случаја коришћења типа `manage`. Због тога у оквиру језика не постоји посебан концепт за дефинисање ова два типа случаја коришћења, већ се при спецификацији користи шаблон који је дефинисан у оквиру случаја коришћења типа `manage`. Дакле, тип случаја коришћења може да се дефинише као `update` и `delete` како би се нагласио циљ који се жели постићи дефинисаним случајем коришћења.

4.2.2.4. СПЕЦИФИКАЦИЈА СЛУЧАЈА КОРИШЋЕЊА ТИПА SEARCH

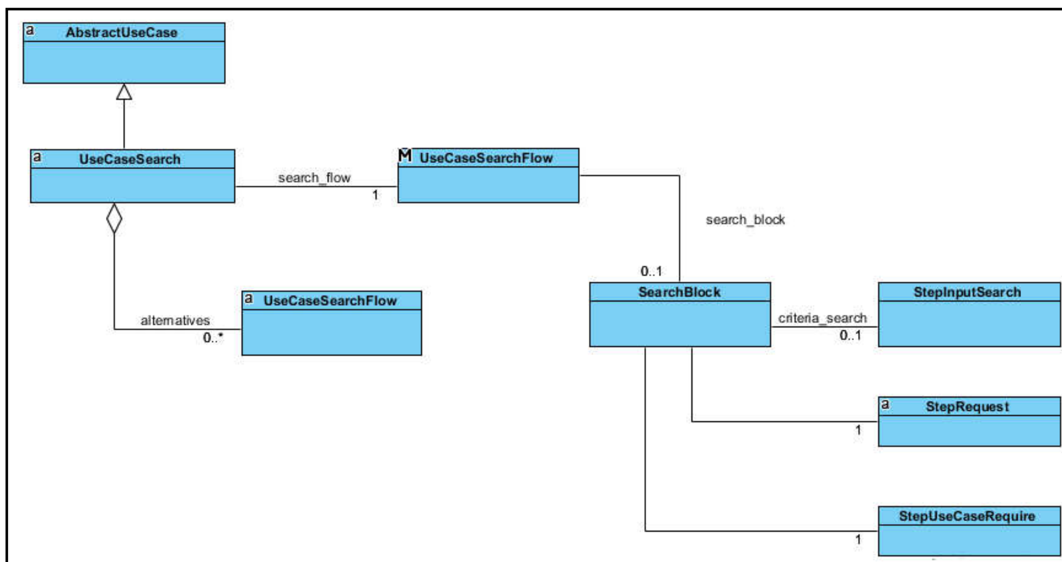
АПСТРАКТНА СИНТАКСА

Спецификција случаја коришћења овог типа омогућена је увођењем `UseCaseSearch` концепта. Акција претраге доменског објекта се дефинише преко `UseCaseSearchFlow` концепта. Приликом спецификације случајева коришћења овог типа претрага објеката се обично може извршити на више начина односно дефинисањем различитих критеријума за претрагу. Најчешће се дефинише подазумевани (*defult*) начин претраге, поред кога се често дефинишу напредни критеријуми за претрагу (*advanced*).

Акције овог типа случаја коришћења груписане су у оквиру блока за претрагу који је дефинисан преко `SearchBlock` концепта. У оквиру блока за претрагу специфицирају се три корака:

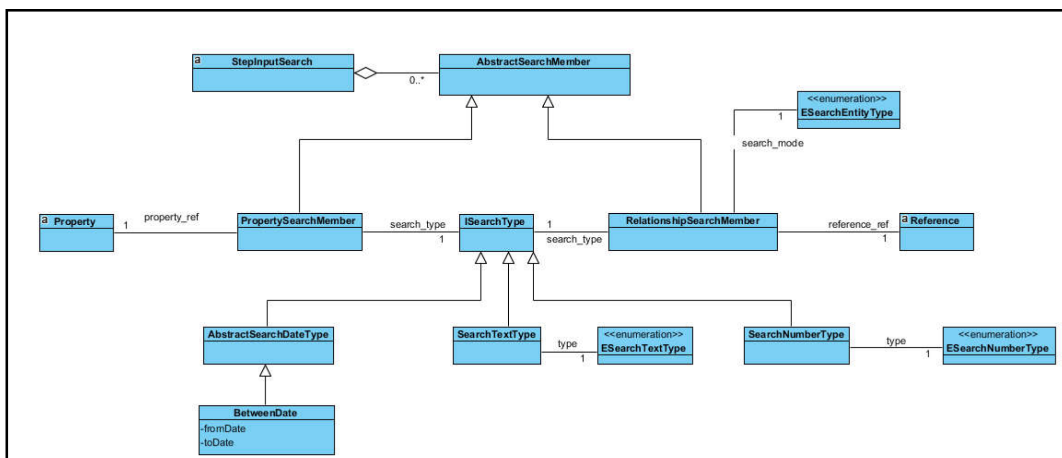
1) Корак дефинисан преко `StepInputSearch` концепта.
2) Корак дефинисан преко `StepRequest` концепта.
3) Корак дефинисан преко `StepUseCaseRequire` концепта.

На слици (Слика 34) дат је приказ **UCDSL** метамодела који се односи на `UseCaseSearch` концепт.



Слика 34. UCM мета-модел: Концепт `UseCaseSearch`

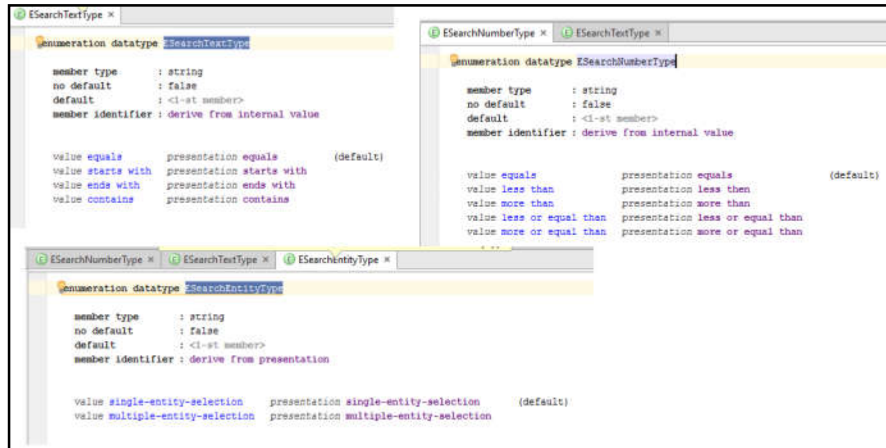
`StepInputSearch` концепт се користи за дефинисање критеријума за претрагу доменског објекта. Претрага доменског објекта је могућа само на основу елемената који су дефинисани у доменском моделу за доменски објекат који се претражује (претрага на основу својства и везе са другим доменским објектима). Ниже на слици (Слика 35) дат је метамодел `StepInputSearch` концепта.



Слика 35. UCM метамодел: Концепт StepInputSearch

У зависности од типа својства зависи и тип претраге. Тако је претрага датумског типа дефинисана `BetweenDate` концептом, претрага својства текстуалног типа дефинисана је `ESearchTextType` концептом, претрага својства нумеричког типа дефинисана је `ESearchNumberType` концептом, док је

спецификације претраге на основу ентитета дефинисана `ESearchEntityType` концептом. На слици (Слика 36) дат је приказ ових концепата.



Слика 36. Енумерације дефинисане преко *jetbrains.mps.baseLanguage.structure* језика

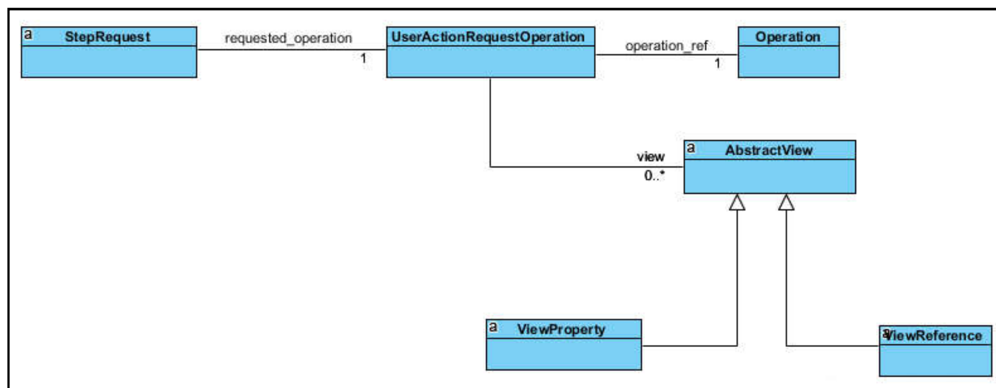
Тако на пример претрага неког својства који је текстуалног типа се може вршити на основу:

a) једнакости *(search type = text : equals),*
b) на основу садржаја *(search type = text : contains),*
c) претрага по тексту који почиње са неким текстом *(search type = text : starts with)* или
d) претрага по тексту који се завршава неким текстом *(search type = text : ends with).*

Претрага својства које је датумског типа увек се дефинише као вредност између две вредности датумског типа.

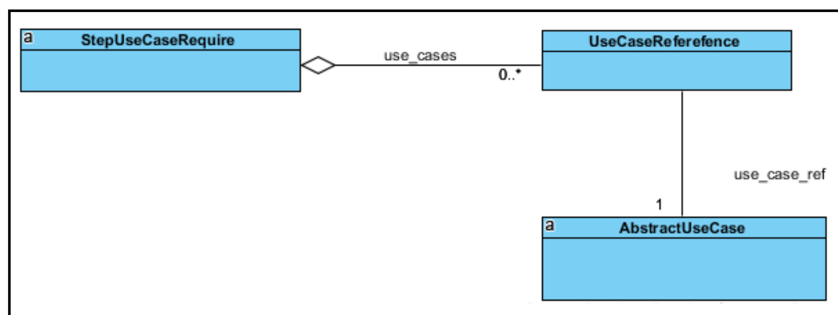
На тај начин се дефинишу параметри на основу којих се врши претрага доменског објекта (својства која се дефинишу) и критеријум за претрагу (на основу једнакости вредности, на основу вредности која почиње неким текстом и слично)

Спецификација `StepRequest` корака подразумева спецификацију акције претраге (акција којом систем враћа кориснику све објекте који задовољавају дефинисани критеријум претраге) и спецификација излаза који систем приказује кориснику након извршења операције претраге. На слици (Слика 37) је дат UCDSL метамодел за `StepRequest` концепт.



Слика 37. UCDSL мета-модел: Концепт `StepRequest`

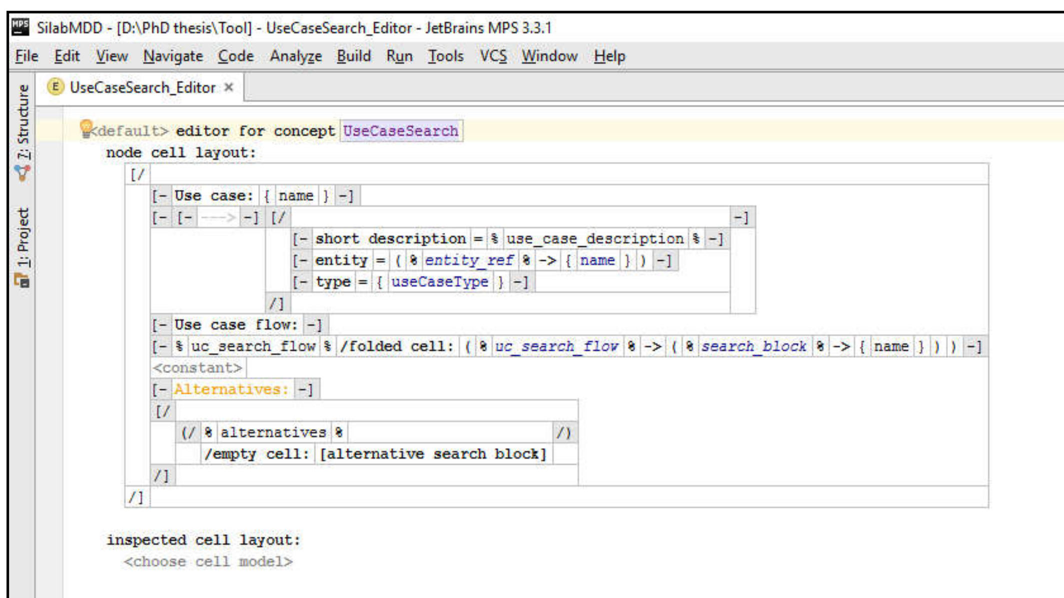
Спецификација `StepUseCaseRequire` корака подразумева спецификацију случајева коришћења којима актор може да приступи из посматраног случаја коришћења. У овом делу могуће је дефинисати само случајеве коришћења који се односе на доменски објекат који се претражује. На слици (Слика 38) је дат UCDSL метамодел за `StepUseCaseRequire` концепт.



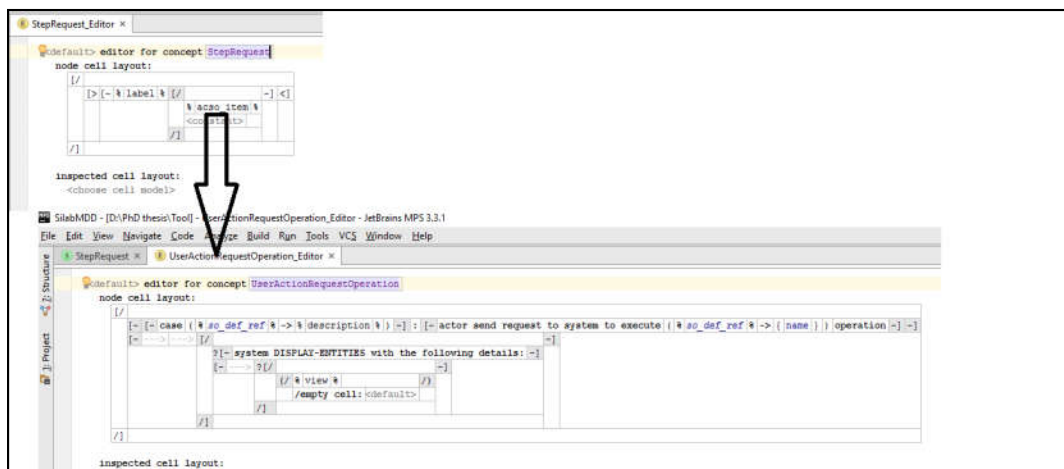
Слика 38. UCDSL метамодел: Концепт `StepUseCaseRequire`

КОНКРЕТНА СИНТАКСА

На слици (Слика 39) дат је приказ конкретне синтаксе за `UseCaseSearch` концепт, а на слици (Слика 40) `StepRequest` концепт је дефинисан преко *jetbrains.mps.lang.editor* језика.



Слика 39. `UseCaseSearch` концепт дефинисан преко *jetbrains.mps.baseLanguage.structure* језика



Слика 40. `StepRequest` концепт дефинисан преко *jetbrains.mps.baseLanguage.structure* језика

Пример спецификације случаја коришћења *UC-search-invoice* приказан је на слици (Слика 41). У овом примеру, претрага се врши на основу јединственог идентификатора или на основу задатог критеријума (*date_created, customer*). На основу спецификације за напредну претрагу (Слика 42) се види да систем треба да пронађе све рачуне (*invoice*) који су креирани између два датума креирана за изабране купце (`search mode= multiple-entity-selection` означава да се претрага може вршити за више купаца) .

Слика 41. Спецификација случаја коришћења *UC-search-invoice (default search)*

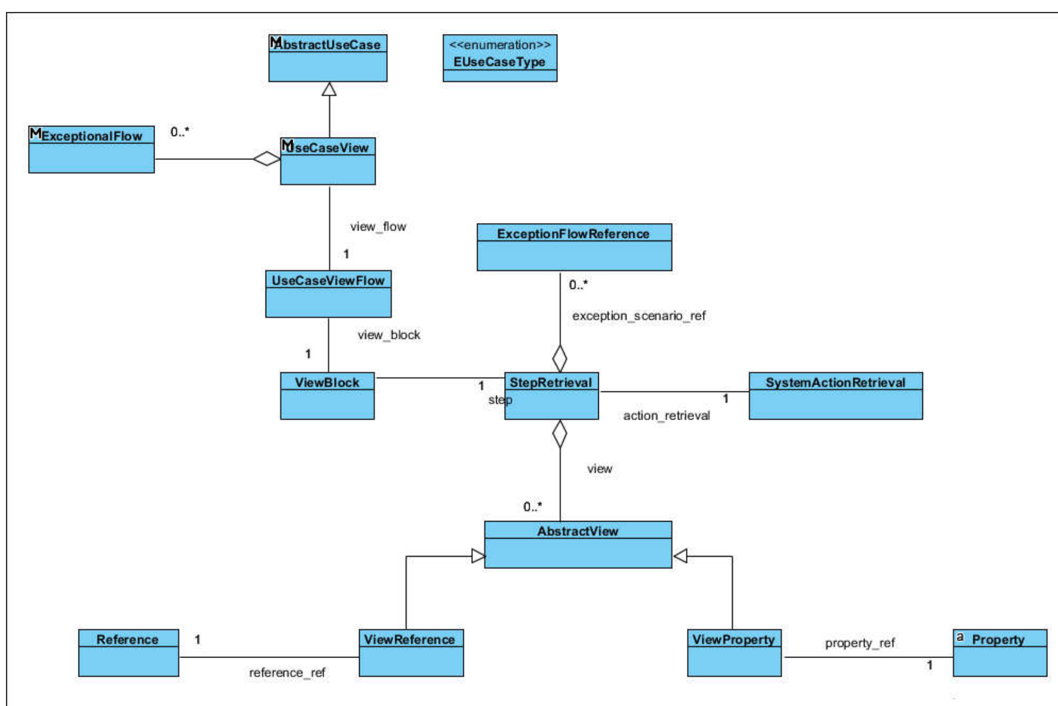
Слика 42. Спецификација случаја коришћења *UC-search-invoice (advanced search)*

4.2.2.5. СПЕЦИФИКАЦИЈА СЛУЧАЈА КОРИШЋЕЊА ТИПА VIEW

АПСТРАКТНА СИНТАКСА

Специфкција случаја коришћења овог типа омогућена је увођењем `UseCaseView` концепта. Спецификација случаја коришћења овог типа подразумева да је објекат који се жели приказати у оквиру овог случаја коришћења, већ претходно идентификован, односно познат [LANGLANDS, M. (2010)]. Према томе, при спецификацији случаја коришћења овог типа се не врши спецификација акција за претрагу и одабир доменске класе чије податке актор жели да види, већ се само врши спецификација својстава доменске класе која се жели приказати актору система.

Спецификација акција у оквиру случаја коришћења овог типа дефинише се унутар блока који је дефинисан преко `ViewBlock` мета-класе. Ниже на слици (Слика 43) дат је приказ UCDSL метамодела за `UseCaseView` концепт.



Слика 43. UCDSL метамодел: Концепт `UseCaseView`

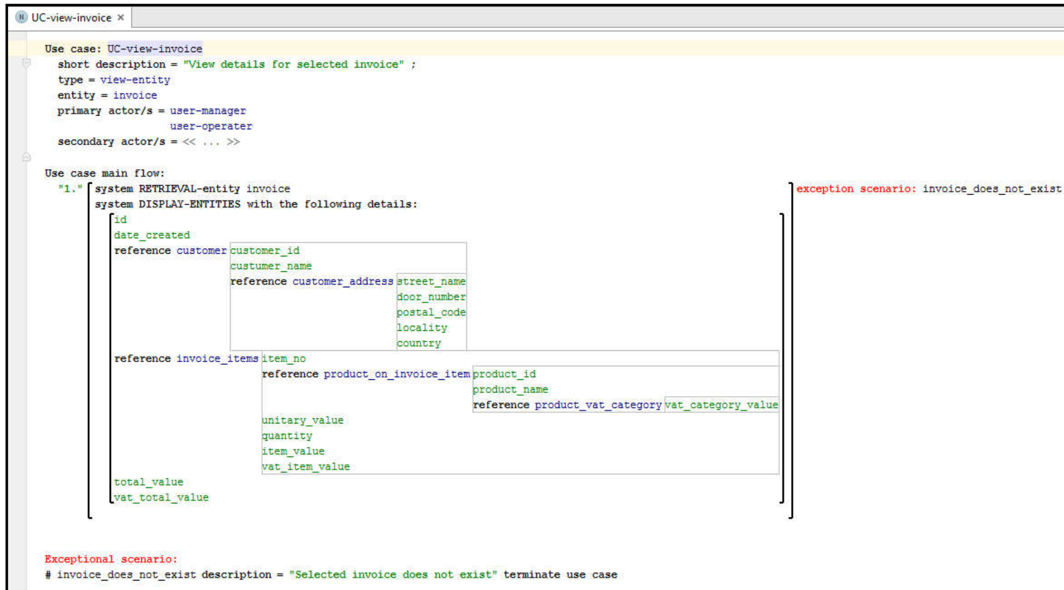
Дефинисање *retrieval* акције система (акција је дефинисана преко `SystemActionRetrieval` концепта) којом систем проналази доменски објекат на основу јединственог идентификатора, подразумева и дефинисање својстава

доменског објекта која се приказују актору система. При спецификацији ових детаља, потребно је дефинисати да ли се приказ односи на приказ својстава доменског објекат или везе. У случају приказа везе коју доменски објекат остварује са другим доменским објектом потребно је дефинисати која се својства објеката који учествује у креирању везе приказују актору система.

КОНКРЕТНА СИНТАКСА

Ниже на слици (Слика 44) је дат приказ `UseCaseView` концепта дефинисаног преко *jetbrains.mps.baseLanguage.structure* језика, док је на слици (Слика 45) приказана спецификација случаја коришћења *UC-view-invoice* у складу са дефинисаном синтаксом.

Слика 44. `UseCaseView` концепт дефинисан преко *jetbrains.mps.lang.editor* језика



Слика 45.Спецификација случаја коришћења *UC-view-invoice*

Приликом спецификације референце неког објекта који се жели приказати, потребно је дефинисати која се тачно својста желе приказати. У примеру приказа рачуна (*UC-view-invoice*), при спецификацији везе *customer* за изабран *invoice,* приказују се својстава *customer_id* и *customer_name.* Приликом спецификације везе *customer_address* приказују се својства *street_name*, *door_number*, *postal_code*, *locality* и *country*.

4.2.3. СПЕЦИФИКАЦИЈА МОДЕЛА ПРЕЛАЗА СТАЊА ПОМОЋУ UCDSL ЈЕЗИКА

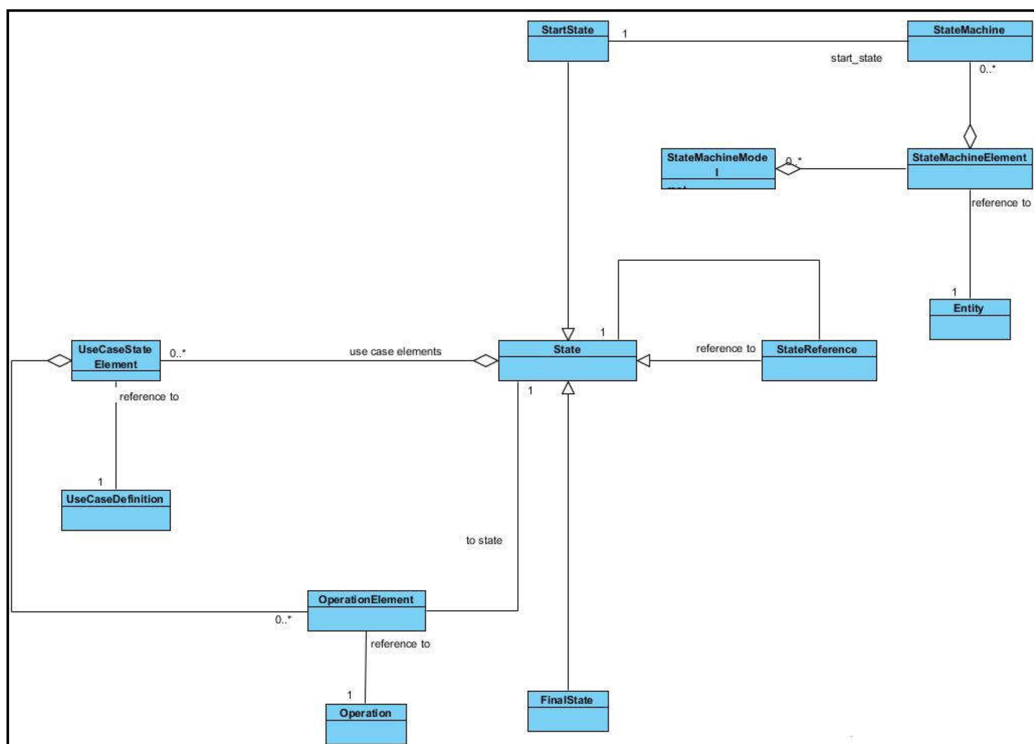
Модел прелаза стања се у оквиру **Silab-UCMDM** методе користи ради прецизног дефинисања услова под којим се одређени случај коришћења извршава, односно за дефинисање предуслова за извршење случаја коришћења. Дефинисањем модела прелаза стања идентификују се и системске операције које су задужене за креирање, ажурирање и брисање објеката из система. Стога дефинисање модела прелаза стања за конкретну доменску класу подразумева:

a) Дефинисање назива модела прелаза стања
b) Дефинисање стања (на пример за доменску класу рачун могуће је дефинисати следећа стања: рачун је креиран, рачун је одобрен, рачун је сторниран, рачун чека на одобреље итд.)
c) Дефинисање скупа случајева коришћења који се могу извшити над доменским објектом у одређеном стању. Повезивањем случаја коришћења са одређеним стањем дефинише се предуслов за извршење тог случаја коришћења. Повезивање стања се врши са случајевима коришћења типа `manage`, `update` и `delete`. У почетном стању најчешће се дефинишу случајеви коришћења типа `create`, док се случајеви коришћења типа `search` најчешће у овом моделу не дефинишу.
d) Дефинисање операција (системских операција) које се могу извршити у оквиру конкретног случаја коришћења.
E) Дефинисања стања у које објекат прелази након извршења дефинисане операције чиме се дефинише пост-услов за извршење системске операције.

АПСТРАКТНА СИНТАКСА

`StateMachineModel` представља главни (`root`) концепт који се користи за дефинисања модела прелаза стања (**STM**odel). Дефинисање модела подразумева дефинисање назива модела и дефинисања елемената модела. Сваки елемент модела се односи на `Entity` концепт који је дефинисан у доменском моделу за који се дефинише модел прелаза стања.

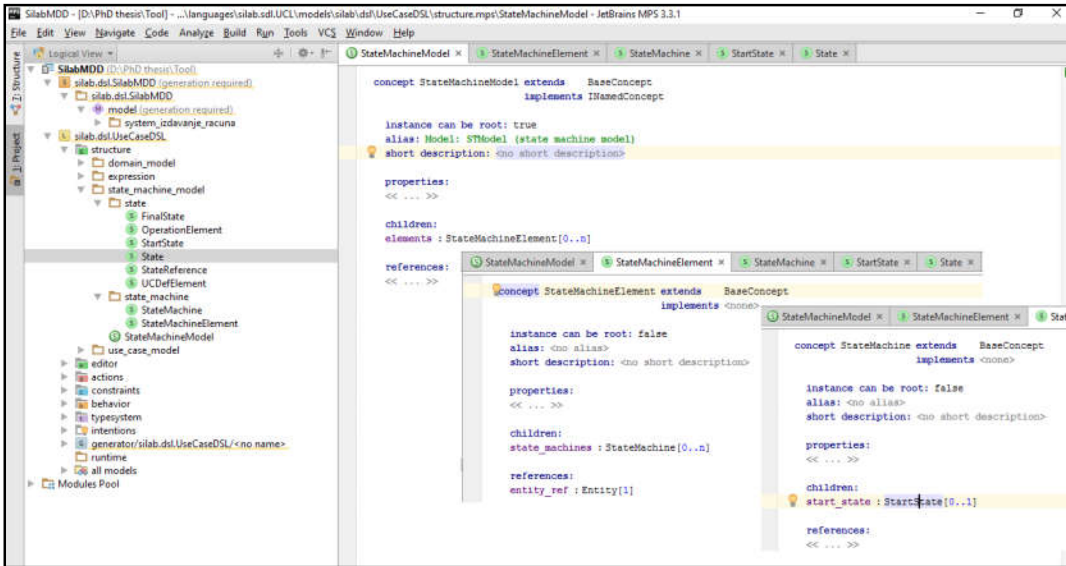
За свако стање дефинише се више `UseCaseStateElement` концепата. `UseCaseStateElement` концептом се дефинише случај коришћења и скуп операција које се могу извршити над доменским објектом током извршења случаја коришћења. Ове операције претходно морају бити дефинисане у доменском моделу. На тај начин се осигурава конзистентност доменског модела, модела случаја коришћења и модела прелаза стања. За сваку операцију се у моделу прелаза стања дефинише стање у које објекат прелази након извршења операције (може се десити да објекат остане у истом стању након извршења неке од операција). Ниже на слици (Слика 46) дат је приказ метамодела за `StateMachine` концепт. Ново стање у које систем прелази након извршења системске операције може бити стање које је већ претходно дефинисано или неко ново стање. Ново стање може бити и крајње стање.



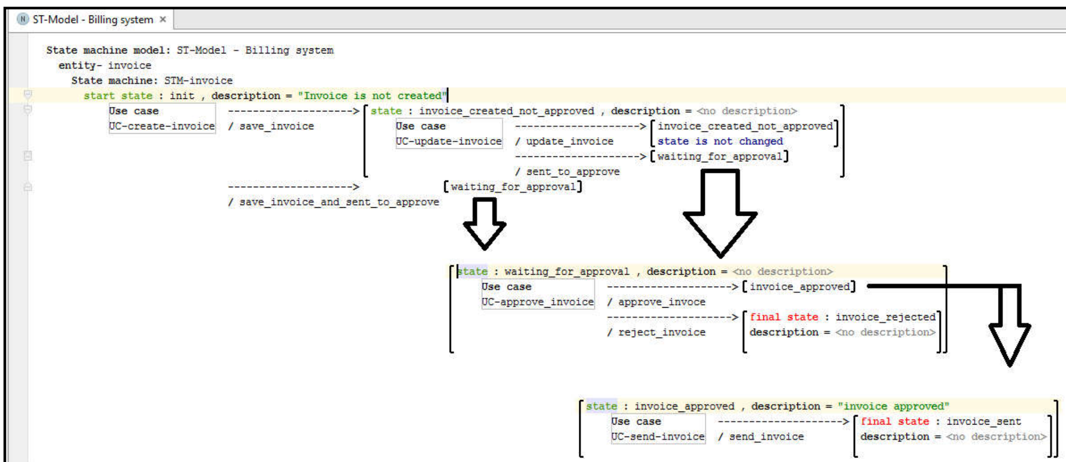
Слика 46. UCDSL метамодел: Концепт `StateMachineModel`

КОНКРЕТНА СИНТАКСА

На слици (Слика 47) дат је приказ `StateMachineModel` концепта дефинисаног преко *jetbrains.mps.baseLanguage.structure* језика, док је на слици (Слика 48) приказана спецификација прелаза стања за концепт рачун (Invoice) исказана дефинисаном синтаксом језика.



Слика 47. `StateMachineModel` концепт дефинисан преко *jetbrains.mps.baseLanguage.structure* језика



Слика 48.Машина прелаза стања за доменску класу Invoice

4.3. UCAPPNDSL - ЈЕЗИК ЗА СПЕЦИФИКАЦИЈУ ПРОТОТИПА АПЛИКАЦИЈЕ

Један од начина валидације корисничких захтева јесте израда прототипа апликације. Израда прототипа апликације подржана је у оквиру *Silab-MDD* приступа на следећи начин:

a) *Израдом модела навигације апликације.*
b) *Дефинисањем шаблона за сваку форму корисничког интерфејса.*

Креирање модела навигације апликације омогућено је преко посебно развијеног *UCAppNDSL* доменски специфичног језика.

Дефинисање шаблона за сваку форму корисничког интерфејса врши се применом модела корисничког интерфејса који је дефинисан у [ANTOVIĆ, I. et al. (2012)] и [ANTOVIĆ, I. (2015)].

Приступ заснован на изради прототипа се заснива на креирању, тестирању, а затим и измени и доради прототипа, све док креирани прототип не задовољи критеријуме на основу којих је могуће креирати комплетан софтверски систем или део система. Овај приступ је погодан у ситуацијама када се током развоја софтверског система очекује велики број интеракција са крајњим корисницима. На тај начин се значајно убрзава процес валидације корисничких захтева.

У литератури и пракси се могу наћи три различита приступа у изради прототипа:

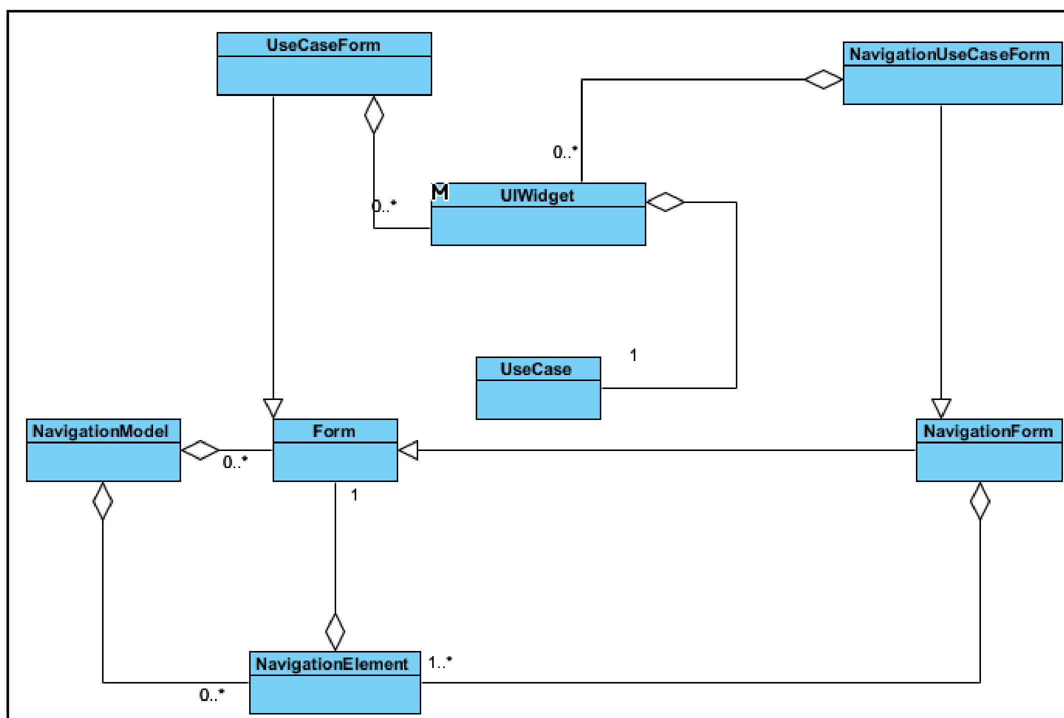
1) Одбацујући или брзи прототип (*throw-away* или rapid *prototyping approach*) који не представља форму коначног решења, односно представља неформално коначно решење. Такво решење није део коначног решења. Обично се користи у ситуацијама када се жели приказати нека идеја и када се жели добити брз одговор од стране корисника, па се као такво користи као приказ одговора на нека питања, након чега се одбацује.
2) Инкраментални приступ (*incremental approach*) представља приступ по коме се развој софтвера заснива на креирању софтверских система коришћењем компоненти које се креирају и интегришу у систем који се развија корак по корак (инкрементално) све док се не развије комплетан систем.

3) Еволутивни приступ (*evolutionary approach*) има за циљ развој комплетног система током низа итерација развоја прототипа. Прототип на тај начин пролази кроз низ побољшања све док на крају не постане коначно решење. Овакав приступ је посебно значајан када сви захтеви нису унапред познати или када су дати нејасно.

У делу који следи дата је апстрактна и конкретна синтакса овог језика.

4.3.1. АПСТРАКТНА СИНТАКСА

`NavigationModel` представља главни (root) концепт која се користи за дефинисање модела апликације (модела навигације апликације). На слици (Слика 49) дат је приказ дела UCAppNDSL метамодела.



Слика 49. Метамодел за `NavigationModel` концепт

Модел навигације чини:

a) Скуп форми и

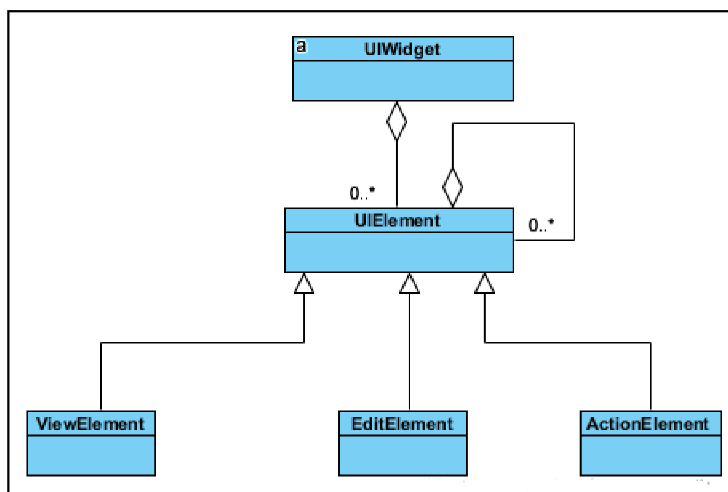
b) Скуп навигационих елемената.

Форма представља апстрактни концепт. Три типа форми се могу дефинисати у овом моделу: форма за навигацију, форма за реализацију случаја коришћења и форма која представља комбинацију претходне две форме.

Форма за навигацију (која је дефинисана преко NavigationForm концепта) представља екранску форму са које је могуће позвати (отворити друге форме). Ова форма стога на себи садржи компоненте навигације које су дефинисане преко NavigationElement концепта. На једној навигационој форми се може наћи више навигационих елемената. Сваки навигациони елемент је везан за једну форму.

Форма за реализацију случаја коришћења дефинисана је преко UseCaseForm концепта. На једној форми у једном тренутку може бити реализовано више случајева коришћења. Због тога се за сваку UseCaseForm форму дефинише UIWidget за који се везује конкретан случај коришћења. На тај начин на једној форми могуће је дефинисати више случајева коришћења.

На слици (Слика 50) приказан је метамодел за UIWidget концепт.

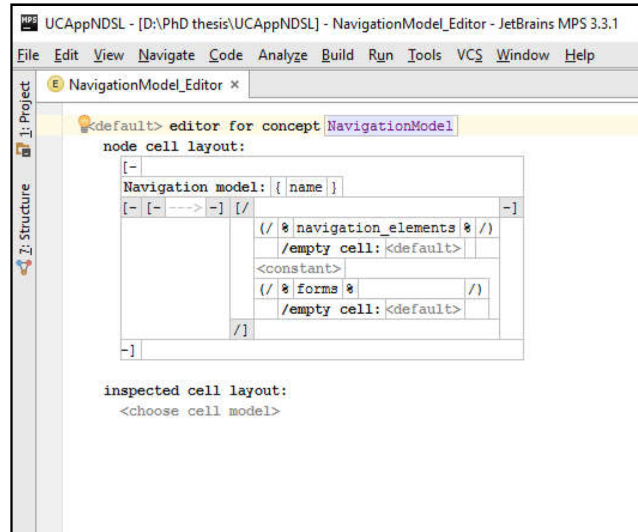


Слика 50. Метамодел UIWidget концепта

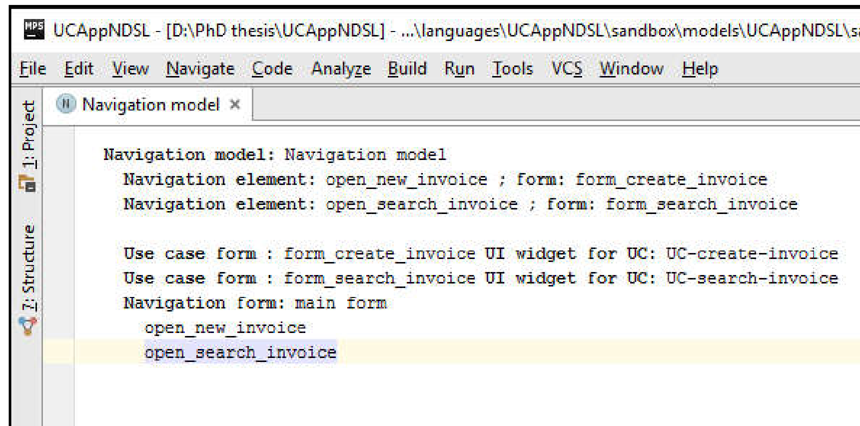
Један UIWidget садржи више UIElement-а који могу бити истог или различитог типа. Елемент за приказ је дефинисан преко ViewElement концепта, елемент за измену преко EditElement концепта, док је елемент за спецификацију акције дефинисан преко ActionElement концепта.

4.3.2. КОНКРЕТНА СИНТАКСА

На слици (Слика 51) дат је приказ `NavigationModel` концепта дефинисаног преко *jetbrains.mps.baseLanguage.structure* језика, док је на слици (Слика 52) приказани модел исказан дефинисаном синтаксом.



Слика 51. NavigationModel концепт дефинисан преко *jetbrains.mps.baseLanguage.structure* језика



Слика 52. Део модела навигације апликације

4.4. DFDDSL- ЈЕЗИК ЗА СПЕЦИФИКАЦИЈУ ДИЈАГРАМА ТОКА ПОДАТАКА

Крајем седамдесетих година прошлог века, прво у Великој Британији, а затим и у Сједињеним Америчким Државама развијена је метода за анализу информационих система која је названа Структурна Системска Анализа (ССА). Најутицајнији аутори ове методе били се *Edward Yourdon* [YOURDON, E. (1989) и *Tom DeMarco* [DeMARCO, T. (1979). Због своје једноставности, али и лакоће примене, ова метода је врло брзо била опште прихваћена за анализу информационих система. Једноставност ССА се огледа у малом броју врло јасних концепата који се користе што је ауторима и био један од циљева. Хијерахијска декомпозиција процеса омогућава рашчлањавање једног сложеног процеса на више мањих под-процеса, чиме се у многоме олакшава анализа система.

Према томе, ССА представља потпуну, тачну, јасну и формалну методу за спецификацију информационог система која се методолошки заснива на функционалној декомпозицији система. Структурну системску анализу чине хијерархијски скуп дијаграма тока података, модел речника података и спецификација примитивних процеса. Једна мало модификована верзија ове ССА методе се већ неколико деценија изучава на Факултету организационих наука чији је утемељивач био професор Бранислав Лазаревић [LAZAREVIĆ, B.et al. (2010)], [BEČEJSKI-VUJAKLIJA, D. (2009)]. У модификованој верзији ССА задржани су концепти који су дефинисани у оригиналној ССА, али су дефинисана строжија правила.

У оквиру *Silab-MDD* приступа развијена су два доменско-специфична језика и то:

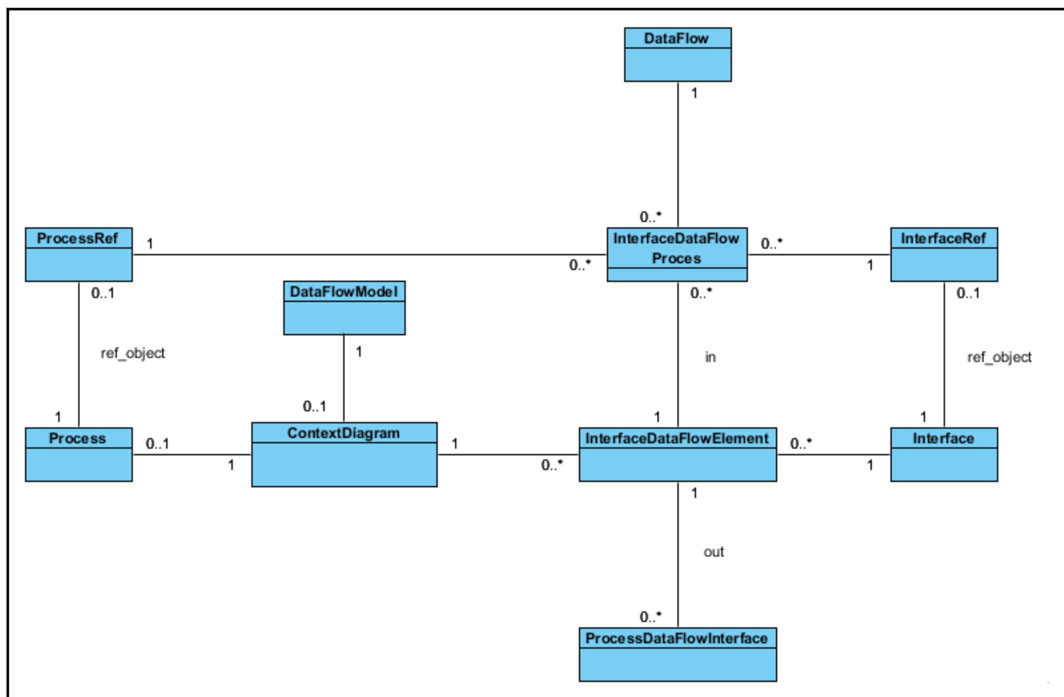
1) **DFDDSL** језика за спецификацију хијерахијског дијаграма тока података.
2) **DataDDSL** језика за спецификацију речника података

У наставку овог поглавља дат је приказ **DFDDSL** доменски-специфичног језика, док је у наредној секцији овог поглавља дат приказ **DataDDSL** доменски-специфичног језика.

4.4.1. АПСТРАКТНА СИНТАКСА

Дијаграм тока података има веома једноставне и јасне графичке концепте. Основни концепти дијаграма тока су: функција (односно процес обраде података), ток података, складишта података и интерфејс (спољни објекат). **DFDDSL** представља доменско-специфични језик са текстуалном конкретном синтаксом, па стога основни концепти дијграма тока података немају графичку презентацију, већ се специфицирају у виду текста. Како би овај недостатак био отклоњен (јер су корисници навикли на графичку нотацију дијаграма тока података) у оквиру Лабораторије за софтверско инжењерство прокренут је пројекат за интерпретацију текстуалне спецификације хијерархијског дијаграма тока података и креирања њене визуелне презентације у складу са концептима дијаграма тока података дефинисаним у оригиналној ССА методи. На тај начин спецификација дијаграма тока података ће се вршити коришћењем **DFDDSL** језика, али ће приказ модела дијаграма тока података бити омогућен и стандардном графичком нотацијом.

Концепт `DataFlowModel` представља *root* концепт **DFDDSL** језика. За дефинисање дијаграма контекста уведен је `ContextDiagram` концепт. За сваки елемент дијаграма тока података (интерфејс, функција односно процес, складиште и ток података) креирани су одговарајући концепти: `Process`, `DataFlow`, `DataStorage` и `Interface` респективно за процес, ток података, складиште и интерфејс. Како би у самом језику подржали правила креирања тока података и правила декомпозиције [LAZAREVIĆ, B. et al., (2010)] у **DFDDSL** језику уведен је концепт референтног објекта који се користи у моделу. Наиме, овим концептом је омогућено референцирање на објекат у моделу који је претходно креиран. Стога су у **DFDDSL** метамоделу уведени концепти `ProcessRef`, `InterfaceRef`, `DataStorageRef` и `DataFlowRef` респективно, за концепте процеса, интерфејса, складишта и тока података. На слици (Слика 53) дат је део DFDDSL метамодела који се односи на `ContextDiagram` концепт.



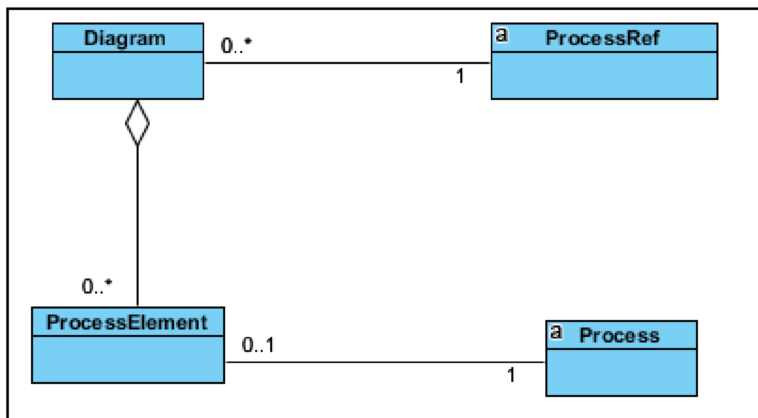
Слика 53. DFDDSL метамодел: Концепт `ContextDiagram`

На дијаграму контекста дефинише се једна фукција и скуп улазних и излазних токова вазаних за ту функцију према спољним објектима (интерфејсима). Број спољних објеката на дијаграму контекста може бити у општем случају произвољан, што зависи од система који се моделује. Спецификација дијаграма контекста преко **DFDDSL** језика подразумева дефинисање онолико `InterfaceDataFlowElement` концепата колико има интерфејса. На слици (Слика 53) се види да на једном дијаграму контекста може бити више `InterfaceDataFlowElement` концепата, а да се један `InterfaceDataFlowElement` концепт односи на један `Interface` концепт.

Дефиниање улазних односно излазних токова на дијаграму контекста омогућено је `InterfaceDataFlowProces` и `ProcessDataFlowInterface` концептима. Посматрано у односу на концепт `Process` за који се дефинише дијаграм контекста, ова два концепта (`InterfaceDataFlowProces` концепт и `ProcessDataFlowInterface` концепт) се користе за исказивање улазних и излазних токова на дијаграму контекста. Тако се концептом `InterfaceDataFlowProces` заправо дефинишу улазни токови, док се концептом `ProcessDataFlowInterface` дефинишу излазни токови. Слично

InterfaceDataFlowProces концепту, дефинисан је и ProcessDataFlowInterface концепт.

Концепт ContextDiagram садржи више дијаграма дефинисаних преко Diagram концепта. Заправо, сваки дијаграм представља дијаграм тока података на одређеном хијерархијском нивоу за конкретан процес који се декомпонује. Ниже на слици (Слика 54) дат је приказ метамодела за Diagram концепт.



Слика 54. DFDDSL метамодел: Концепт Diagram

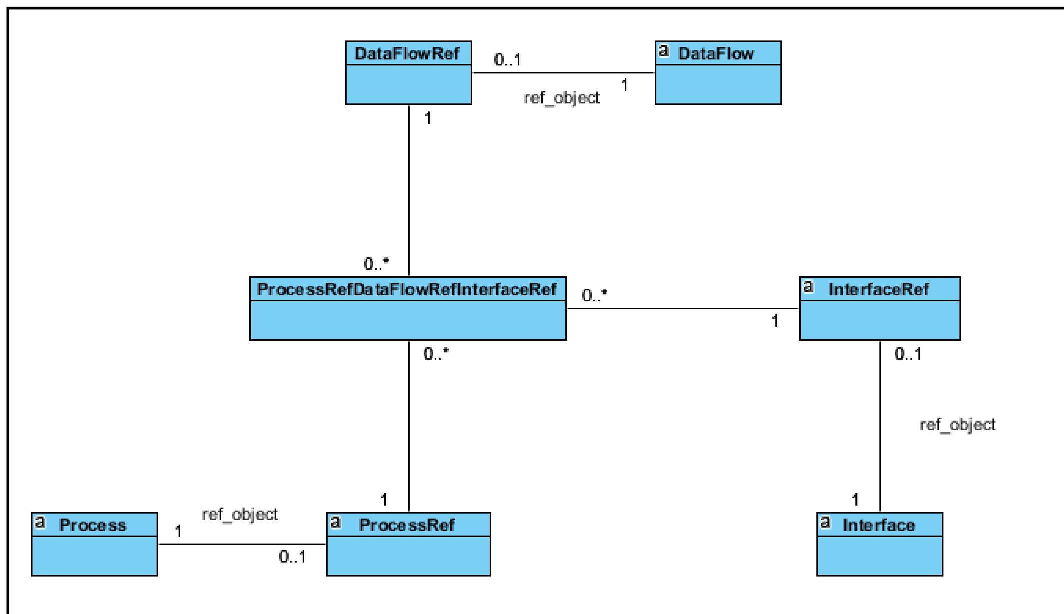
Сваки дијаграм тока података има ознаку која представља ниво декомпозиције, сарджи референцу на процес који се декомпонује, односно за који се креира дијаграм тока података и скуп елемената процеса (дефинисаних преко ProcessElement концепта) за који се дефинишу улазни и излазни токови.

За сваки елемент процеса се дефинишу:

a) Токови који иду ка интерфејсу (излазни токови) који су дефинисани преко ProcessRefDataFlowRefInterfaceRef концепта и од интерфејса (улазни токови) који су дефинисани преко InterfaceRefDataFlowRefProcesRef концепта.
b) Токови који иду од складишта (улазни токови) који су дефинисани преко апстрактног AbstractProcessDataStorage концепта и ка складишту података (излазни токови) који су дефинисани преко акстрактног AbstractDataStorageProcess концепта .

На слици (Слика 55) дат је UCDSL метамодел за ProcessRefDataFlowRefInterfaceRef концепт који се користи за спецификацију

излазних токова из процеса ка интерфејсу. На сличан начин је дефинисан и InterfaceRefDataFlowRefProcesRef концепт.

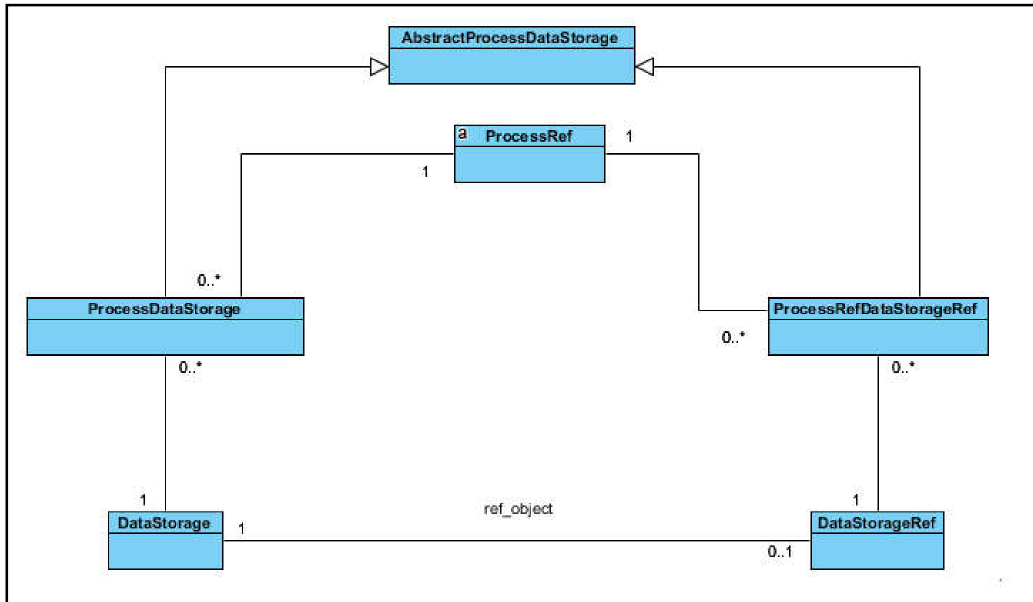


Слика 55. DFDDSL метамодел: Концепт ProcessRefDataFlowRefInterfaceRef

Са слике (Слика 55) се види да креирање инстанце објекта ProcessRefDataFlowRefInterfaceRef концепта на моделу подразумева избор процеса (референцирање на процес који је креиран у моделу, веза са концептом ProcessRef), избор тока података (рефернцирање на ток податак који је креиран у моделу, веза са концептом DataFlowRef) и избор тока података који је такође креиран раније у моделу (веза са концептом InterfaceRef). Избор појединих концепата врши се аутоматизовано у самом језику.

За спецификацију токова од процеса ка складиштима, као што је претходно речено, користе се концепти AbstractProcessDataStorage и AbstractDataStorageProcess. Апстрактни концепти су уведени из разлога што улазни односно излазни ток ка складишту може бити двојак: 1) складиште се први пут уводи у модел, 2) складиште постоји већ у моделу па је потребно извршити само референцирање на њега. Тако су за апстрактни AbstractProcessDataStorage концепт уведени конкретни концепти који га наслеђују: ProcessDataStorage и ProcessRefDataStorageRef.

На слици (Слика 56) дат је метамодел за AbstractProcessDatastorage концепт. На сличан начин је дефинисан и AbstractDataStorageProcess концепт .

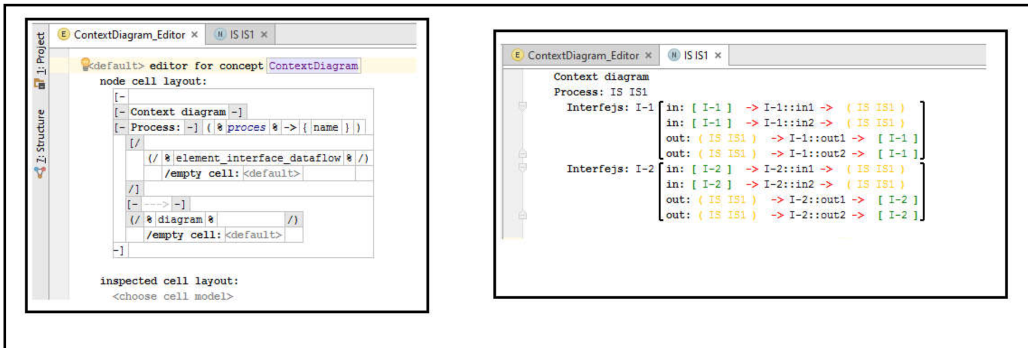


Слика 56. DFDDSL метамодел за концепт `AbstractProcessDatastorage`

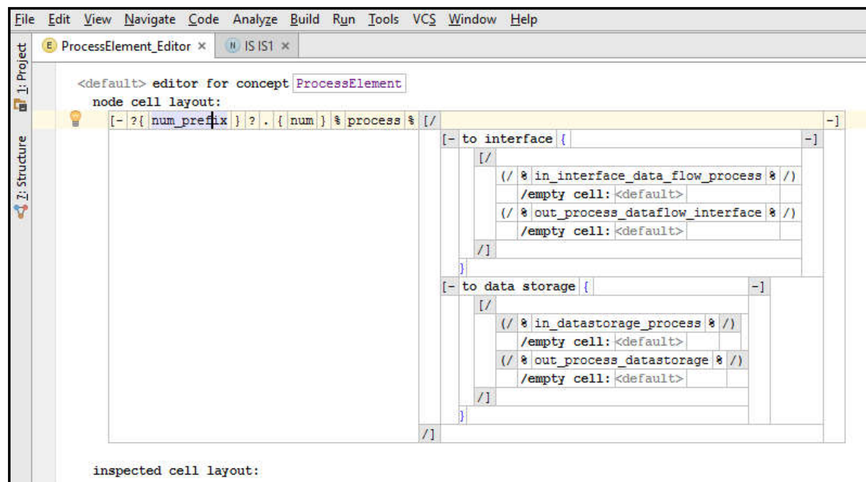
4.4.2. КОНКРЕТНА СИНТАКСА

Структурна системска анализа посматра систем као функцију, стога први корак у креирању дијаграма тока података јесте дефинисање система, односно функције која се моделује.

Ниже на слици (Слика 57) приказан је едитор (слика лево) у коме је дефинисана конкретна синтакса за `ContextDiagram` концепт (дефинисана преко *jetbrains.mps.baseLanguage.structure* језика,) и пример модела дијаграма контекста за хипотетички систем *IS IS1* (слика десно) који је креиран према дефинисаној конкретној синтакси. На дијаграму контекста исказана су два интерфејса: I-1 и I-2, са по два улазна и излазна тока. То је подржано аутоматски и није могуће мењати. На слици (Слика 58) приказан је концепт `ProcessElement` дефинисан преко *jetbrains.mps.lang.editor* језика.

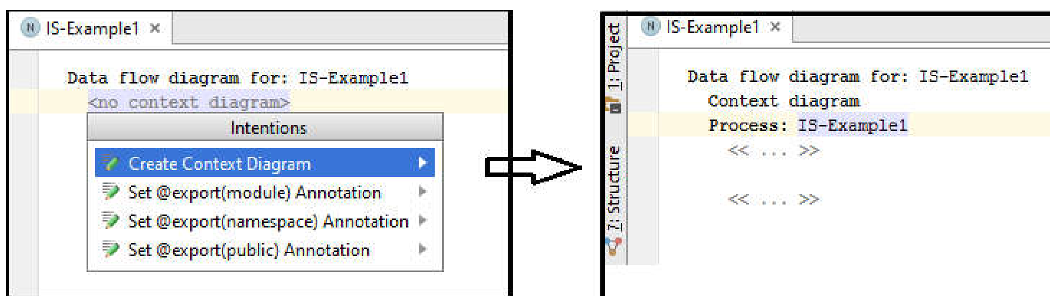


Слика 57. `ContextDiagram` концепт дефинисан преко *jetbrains.mps.lang.editor* језика и пример конкретног модела



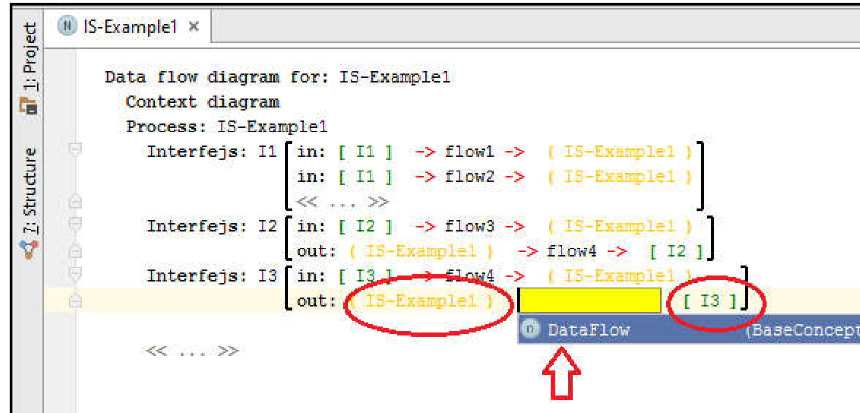
Слика 58. Концепт `ProcessElement` дефинисан преко *jetbrains.mps.lang.editor* језика

DFDDSL језиком креирање дијаграма контекста је могуће извршити преко опције *„Create Context Diagram“* (Слика 59). Назив процеса на дијаграму контекста се генерише аутоматски, а назив процеса је исти као и назив система који се моделује.



Слика 59. Дефинисање дијаграма контекста

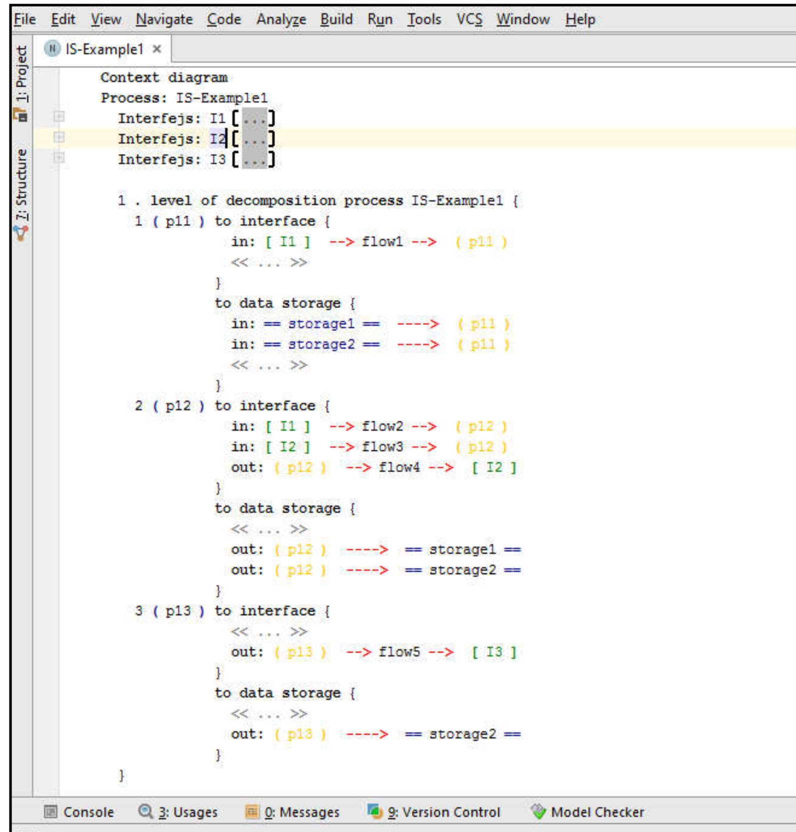
Дефинисање дијаграма контекста подразумева дефинисање интерфејса и одговарајућих излазних и улазних токова. Ниже на слици (Слика 60) је дат приказ спецификације дијаграма контекста исказан преко **DFDDSL** едитора**.**



Слика 60. Спецификација дијаграма контекста

Приликом дефинисања токова који иду од интерфејса до процеса (улазних токова за процес), као и од процеса до интерфејса (излазних токова за процес) потребно је само дефинисати назив тока података. У оквиру самог језика повезивање улазног и излазног тока са одговарајућим интерфејсом и процесом подржано је аутоматски и није могуће мењати.

Након спецификације дијаграма контекста, прелази се на 1. ниво декомпозиције процеса који је дефинисан на дијаграму контекста. На 1.нивоу декомпозиције, процес са дијаграма контекста се декомпонује на процесе (обично од 2 до 7 подпроцеса). За сваки процес који се уведе на дијаграму потребно је дефинисати улазне и излазне токове. Стога се за сваки процес дефинишу токови који иду ка и од (интерфејса и складишта), а у складу са правилима креирања дијаграма тока података дефинисаних у [LAZAREVIĆ, B.et al. (2010)]. Први ниво декомпозиције приказан је на слици (Слика 61).



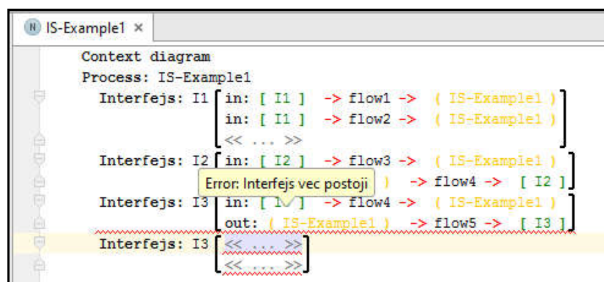
Слика 61. Пример: Декомпозиције процеса IS-Example1

За сваки процес који је идентификован на 1. нивоу декомпозиције специфицирају се улазни и излазни токови ка интерфејсу и улазни и излазни токови ка складишту података. Треба рећи да нове токове ка интерфејсу на 1. нивоу декомпозиције није могуће увести, већ се само користе токови који су дефинисани на дијаграму контекста. Нове токове је могуће дефинисати само ка складишту.

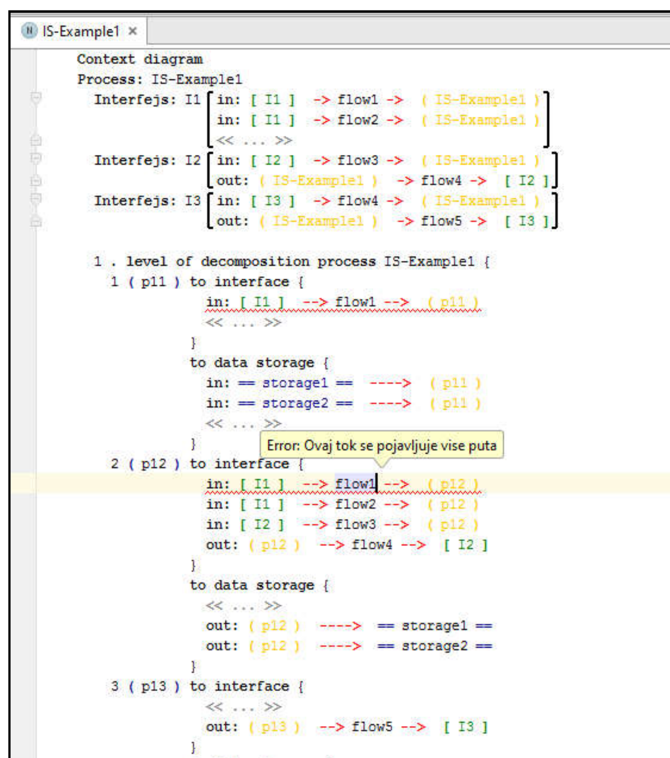
Након креирања 1.нивоа декомпозиције за процесе које је потребно даље декомпоновати креирају се дијаграми 2. нивоа декомпозиције.

У односу на алате као што је на пример Microsoft Visio, који се може користити за креирање дијграма тока података, у оквиру **DFDDSL** језика омогућена је формалнија спецификација токова података у складу са правилима и препорукама за креирање дијаграма тока података [LAZAREVIĆ, B.et al. (2010)]. Тако на пример није могуће дефинисати интерфејс са истим називом који већ постоји (Слика 62). Такође, није могуће дефинисати један исти ток података ка различитим

процесима (Слика 63). Са друге стране, нека правила није могуће нарушити јер су имплементирана у самом језику. Тако се на пример приликом декомпоновања процеса строго води рачуна о балансу токова или о начину повезивања процеса, складишта и интерфејса преко токова (Слика 62) и (Слика 63).



Слика 62. Приказ грешке код дефинисања интерфејса који већ постоји са истим именом



Слика 63. Приказ грешке код дефинисања тока који се већ појављује

4.5. DATADDSL - ЈЕЗИК ЗА СПЕЦИФИКЦИЈУ РЕЧНИКА ПОДАТАКА

Речник података представља модел за структурирани опис података у систему. Њиме се описује садржај и структура података. У оквиру ССА методе речником података се описује структура и садржај свих токова и складишта. Ако складишта и токови имају исту структуру, тада их не треба посебно описивати.

Основни концепти речника података су: а) поље и домен и б) структуре. Поље представља атомску структуру која се даље не докомпонује, и која садржи вредност и представља компоненту неке структуре. Вредност поља дефинисана је доменом. Домен представља скуп дозвољених вредности које поље дефинисано над тим доменом може да узме. Домен може бити унапред дефинисан (предефинисан). Најчешће ови домени одговарају типовима података у програмским језицима (*integer*, *char*, *string*, *boolean* итд.). Семантички домени се дефинишу преко свог имена. Они се креирају на основу неког унапред дефинисаног домена за који се могу дефинисати ограничења на могући скуп вредности.

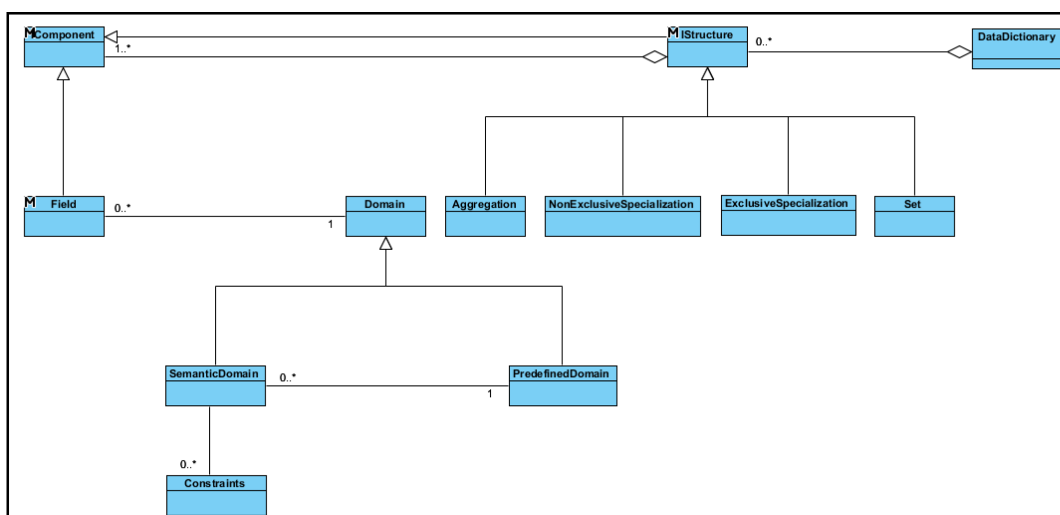
Структура се дефинише као композиција компоненти, где компонента структуре може бити поље или нека друга структура (најчешће део неке структуре). Речник података разликује следеће типове структура:

- **агрегација компоненти**: представља сложену структуру коју чини листа од n компоненти и која се представља унутар шпицастих заграда (<a,b,c>)
- **екслузивна специјализација:** представља унију компоненти која се представља у угластим заградама ([a,b,c]) и која означава да се у структури екслузивно појављује само једна компонента (или a, или b или c).
- **неекслузивна специјализација:** представља унију компоненти која се представља у косим заградама (/a,b,c/) и која означава да се у структури може појавити било само једна, било две или све компоненте.
- **скуп компоненти,** односно скуп више вредности једне компоненте, представља се у витичастим заградама ({a,b,c,d}) и означава да се у одговарајућој структури ова компонента може више пута појавити.

У наставу овог поглавља дат је приказ **DataDDSL** *(DataDictionary Domain Specific Language)* доменски-специфичног језик за спецификацију речника података.

4.5.1. АПСТРАКТНА СИНТАКСА

На основу дефиниције концепата из речника података ниже на слици (Слика 64) су приказани основни концепти метамодела речника података. Концепт `DataDictionary` представља *root* концепт **DataDDSL** језика у оквиру кога се дефинишу одговарајуће структуре.



Слика 64. DFDDSL метамодел: Концепт `DataDictionary`

Основни концепт у метамоделу јесте компонента која је дефинисана преко концепта `IComponent`. Компонента може бити поље (дефинисано концептом `Field`) или структура (дефинисана концептом `IStructure`). Свако поље садржи назив, опис и домен коме припада. Домен (дефинисан концептом `Domain`) може бити предефинисан (концепт `PredefinedDomain`) или семантички (концепт `SemanticDomain`). Предефинисани домени су описани следећим концептима:

- `Number` – користи се за дефинисање целих бројева
- `Date` – користи се за дефинисање датума. Приликом дефинисања поља датумског типа могуће је дефинисати и формат датума
- `Text` – користи се за дефинисање поља која садрже секвенцу знакова

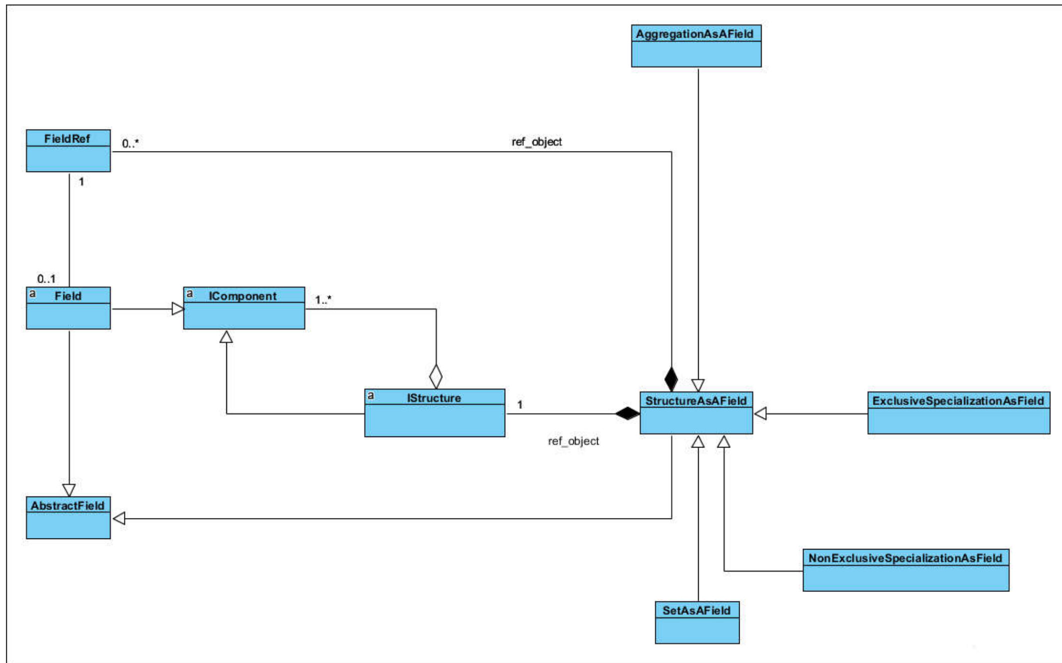
- `Logical` – користи за дефинисање поља логичког типа са скупом дозвољених вредности `true` и `false`
- `Decimal` – користи се за дефинисање поља за опис реалних бројева. Приликом дефинисања поља овог типа потребно је дефинисати број цифара испред зареза и број цифара иза зареза.

Семантички домен се дефинише на основу предефинисаног домена за који се дефинишу ограничења на могући скуп вредности из предефинисаног домена.

Структура представља компоненту која садржи скуп других компоненти које могу бити поља и/или структуре. Због тога је у метамоделу дефинисан апстрактни концепт `IComponent` кога наслеђује конкретни концепт `Field`, и апстрактни концепт `IStructure`. За сваки тип структуре дефинисани су одговарајући концепти који наслеђују апстрактни концепт `IStructure`, па је тако за **агрегацију компоненти** дефинисан `Aggregation` концепт, за **екслузивну специјализацију** дефинисан `ExclusiveSpecialization` концепт, за **неекслузивну специјализацију** дефинисан `NonExclusiveSpecialization` концепт , док је за **скуп компоненти** дефинисан `Set` концепт.

У циљу повећања изражајности **DataDDSL** језика, у коме ће бити исказан речник података за конкретан систем (модел), метамодел речника података са слике (Слика 64) проширен је концептима који на нивоу модела омогућавају рефренцирање на постојеће концепте модела. У том смислу проширење је извршено на концепту `Field` (Слика 65).

У односу на метамодел са слике (Слика 64) уведено је апстракно поље као концепт. Апстракно поље може бити `Field` концепт и `StructureAsAField` концепт. Ниже на слици дат је део метамодела DFDDSL језика који се односи на концепт `Field`. На DFDDSL метамоделу уведен је концепт `StructureAsAField`. Овај концепт се користи при спецификацији конкретног модела, тако да се њиме дефинише поље које садржи више других поља неке структуре која је већ претходно дефинисана.



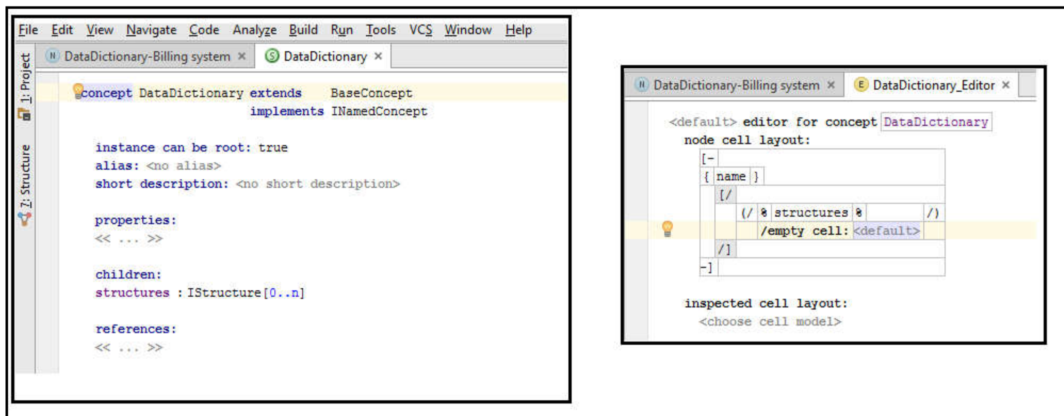
Слика 65. DFDDSL метамодел: Концепт `StructureAsAField`

Концепт `StructureAsAField` представља апстрактан концепт за кога су дефинисани конкретни концепти: `AggregationAsAField`, `ExclusiveSpecializationAsField`, `NonExclusiveSpecializationAsField` и `SetOfComponentsAsAField`. На тај начин у **DataDDSL** језику омогућено је да се у оквиру једне структуре дефинише друга структура која представља структуру од поља предтходно дефинисаних структура. У наставку овог поглавља дати су и неки конкретни примери.

4.5.2. КОНКРЕТНА СИНТАКСА

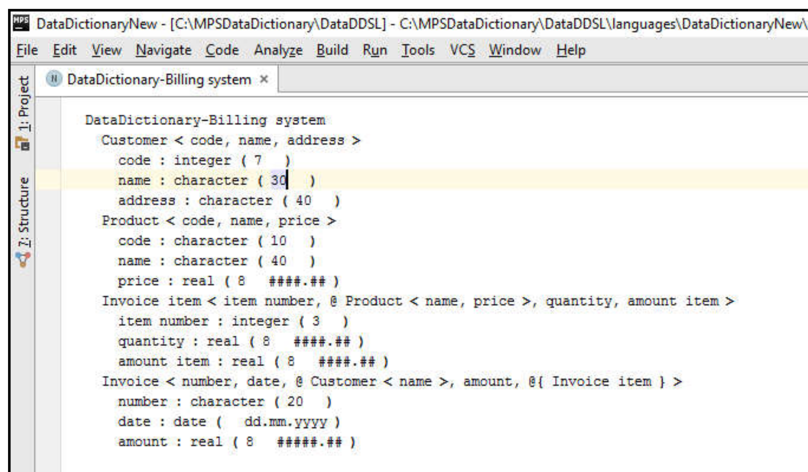
Као и код претходно дефинисаних језика имплементација **DataDDSL** језика је извршена у JetBrains MPS алату, па је дефинисање конкретне синтаксе језика извршено преко *jetbrains.mps.lang.editor* језика.

Дефинисање речника података почиње инстанцирањем `DataDictionary` концепта који представља root концепт у оквиру кога се дефинишу компоненте структуре. Ниже на слици (Слика 66) приказан је `DataDictionary` концепт (*structure* концепт за дефинисање апстрактне синтаксе (лево) и *editor* концепт за дефинисање конкретне синтаксе (десно)).



Слика 66. DataDictionary концепт (*structure* концепт и *editor* концепт)

На слици (Слика 67) дата је спецификација речника података за структуру `Invoice`.



Слика 67. Речник података за структуру Invoice

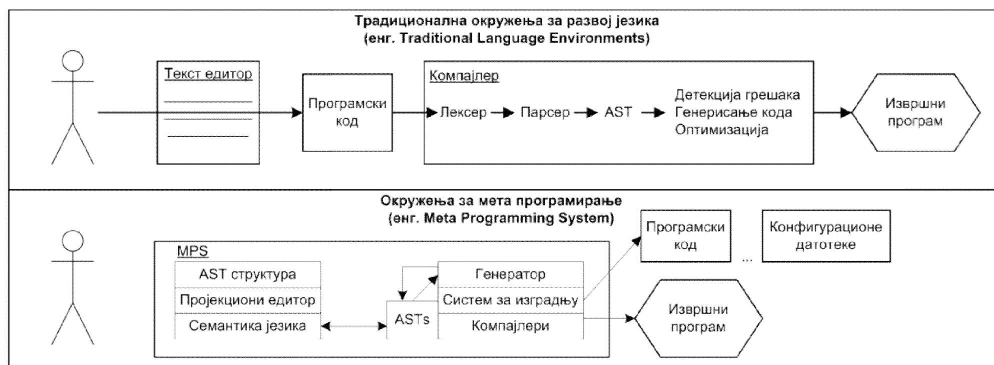
ПОГЛАВЉЕ 5. SILAB-MDDTOOLSET АЛАТ

У претходном делу рада приказани су предложени доменско-специфични језици који служе као основа за реализацију SILAB-MDD приступа. Имплементација ових језика извршена је преко *JetBrains MPS* алата за метапрограмирање *(JetBrains MPS metaprogramming system)*. Овај алат представља окружење *(workbench)* за развој доменско-специфичних језика. У овом поглављу дат је преглед основних карактеристика језика, након чега је приказан развој SILAB-MDDTOOLSET алата који представља додатак *(plugin)* за MPS.

5.1. JETBRAINS MPS КАО АЛАТ ЗА МЕТАПРОГРАМИРАЊЕ

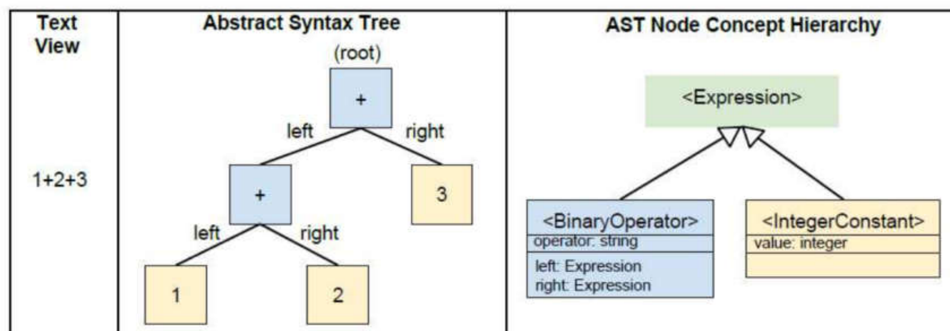
Традиционална развојна окружења користе технологију компајлирања (коришћење лексера и парсера за креирање стабла апстрактне синтаксе *(abstract sintax tree - AST)*) како би утврдили грешке у изворном коду (да ли је изворни код написан у складу са дефинисаном граматиком). Осим за утврђивање грешке, AST се користи за оптимизацију и генерисање извршивог кода. Свако AST стабло садржи један чвор који представља корен који се даље грана.

MPS не користи ову традиционалну парадигму, већ корисник приликом креирања језика директно врши интеракцију са AST-ом преко одговарајућег пројекционог едитора *(projectional editor)*. На тај начин не постоји потреба да се комплетан програм изрази као текст. Тим је омогућено да се при дефинисању језика не користи само текст, већ и симболи, табеле и графици који се пројектују на AST-у. На слици (Слика 68) приказане су ове две парадигме.



Слика 68. Парадигме за развој језика [CAMPAGNE, F. (2015)]

Начин комбиновања чворова приказан је на слици (Слика 69) [CAMPAGNE, F. (2015)].



Слика 69. AST стабло

Дакле, у MPS-у програми су представљени помоћу модела, а не помоћу текста. На овај начин се избегавају проблеми везани за парсирање текста, па се у алату уместо креирање граматика креира метамодел, због чега је овај алат и добио назив *Meta Programming System*. У MPS-у постоји скуп интегрисаних језика који су развијени унутар MPS-а и који могу да се користе и проширују. Једна од значајних предности креирања језика коришћењем ове парадигме јесте могућност проширивања и комбиновања језика што није лако уколико је језик развијен на традиционалан начин. У оквиру MPS-а креиран је посебан језик који се назива *jetbrains.mps.baseLanguage* који представља копију Јава програмског језика верзије 6 и који има скоро идентичан скуп конструкција. Стога MPS најчешће користи Јаву као циљни језик у који се генерише програмски код.

Ниже су наведени најбитнији аспекти који се користе у MPS-у за дефинисање језика. Сваки од ових аспеката представља језик за себе и користи се или за дефинисање синтаксе (апстрактне или конкретне) или семантике језика:

- *Structure аспект* се користи за дефинисање чворова AST стабла који представљају концепте који се могу користити у језику. Сваки концепт може да садржи:
    - *Property* који представља својство које може да има концепт. Свако својство мора да садржи назив и тип.
    - *Children* представља који чворови могу бити деца тог концепта.

- *References* представља показивач односно референцу на неки чвор који постоји у стаблу.
- Један концепт слично као у објектно оријентисаним језицима може да наслеђује други концепт.
- Слично објектно-оријентисаним језицима и у *Structure* аспекту постоји концепт интерфејс.

Језик који се користи за представљање овог аспекта је *jetbrains.mps.lang.structure.*

- *Едитор аспект* се дефинише за концепт језика и служи за дефинисање начина приказивања концепта у језику. Представљен је преко посебног *jetbrains.mps.lang.editor* језика. Едитор садржи ћелије које могу да садрже друге ћелије, текст или неку компоненту. За сваку компоненту поред стандардног (уобичајеног) едитора могуће је дефинисати и едиторе компоненти. Ово омогућава да се за једну компоненту дефинише више едитора.

- *Аспект акције (Actions)* се користи уколико се при измени неког чвора желе обезбедити фунције аутоматског комплетирања (довршавања) израза (на пример корисник уноси део текста, а језик аутоматски комплетира цео израз). Дакле, акција се дефинише за одређени концепт. Акције се могу извршити безусловно или под одређеним (дефинисаним) условима.

- *Аспект понашања (Behavior)* се користи када се неком концепту жели додати понашање. Понашање се описује методом која се дефинише за концепт. У оквиру овог аспекта могуће је дефинисати виртуалне и не-виртуалне методе, статичке и не-статичке методе и конструкторе. За дефинисање овог аспекта коришћен је *jetbrains.mps.lang.behavior* језик.

- *Аспект за дефинисање ограничења (Constraints)* је дефинисан посебним језиком *jetbrains.mps.lang.constraints* и користи се за дефинисање ограничења веза између чворова, и дефинисање ограничења на скупу дозвољених вредности за атрибуте, референце и децу.

- *Intentions аспект* је дефинисан језиком *jetbrains.mps.lang.intentions* и користи се за дефинисање различитих трансформација AST стабла. Ове трансформације активира корисник на захтев, а дефинишу се за одређени контекст (за одређену компоненту под којим условима може да се изврши)
- *Генератор аспект (Generator)* се користи за трансформацију модела написаних у једном језику у програмски код у неком другом програмскокм језику.

5.2. SILAB-MDDTOOLSET ДОДАТАК ЗА MPS

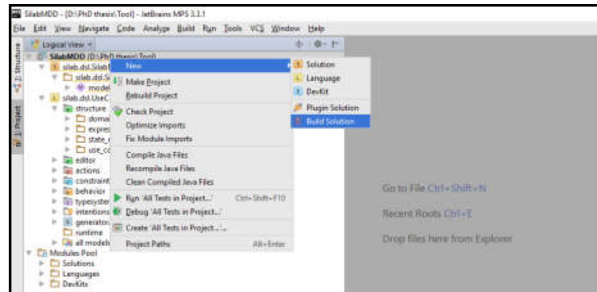
Тренутно се SILAB-MDDTOOLSET може користити као додатак *(plugin)* за окружења као што су MPS и IntelliJ IDEA. У плану је креирање посебног алата на бази MPS-а који је прилагођен *(customized)* само да подржи SILABMDD приступ. SILAB-MDDTOOLSET тренутно чине 3 додатка: 1) додатак којим се омогућава текстуална спецификација дијаграма тока података (овај додатак користи DFDDSL језик), 2) додатак којим се омогућава спецификација речника података (овај додатак користи DataDDSL језик) и 3) додатак којим се омогућава спецификација случајева коришћења (овај додатак користи UCDSL језик).

Корисници MPS-а овај додатак могу користити за спецификацију модела унутар пројекта који се креирају у оквиру MPS развојног окружења. Корисници IntelliJ IDEA развојног окружења имају једну могућност више, а то је да након креирања одговарајућих модела исте користе унутар самог окружења. Тако на пример идентификоване системске операције (дефинисане у моделу захтева) се могу повезати директно са класом или методом у програмском коду који реализује (имплементира) идентификовану функционалност дефинисаних системских операција. На тај начин алат омогућава праћење следљивости *(traceability)* захтева од места њихове идентификације, спецификације и њихове имплементације.

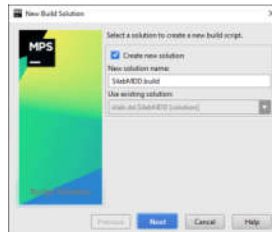
Први корак за креирање додатка за развијени доменско-специфични језик јесте креирање *build* скрипта. За креирање *build* скрипта у оквиру MPS-а користе се *jetbrans.mps.build* и *jetbrans.mps.build.mps* посебно развијени језици. Build скрипт се креира за пројекат. Из менија који је приказан на слици (Слика 108) бира

се опција *Build Solution*. MPS има добро подржану инфраструктуру за креирање додатака. Након креирања *build* скрипта и његовог конфигурисања уз помоћ *ant* алата за копајлирање и паковање *(build)*, додатак се пакује као *zip* архива која се може интегрисати (инсталирати) у развојно окружење...

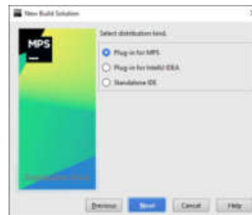
Да би се креирао *build* скрипт, потребно је у кораку 1 креирати *Build Solution* (Слика 70) и у кораку 2 креирати пројекат (Слика 71). У кораку 3 бира се окружење за које се креира додатак (Слика 72), док се у кораку 4 бирају компоненте језика које треба да буду укључене у додатак (Слика 73).



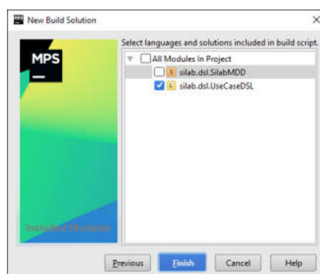
Слика 70. Креирање *Build Solution-a* (корак 1.)



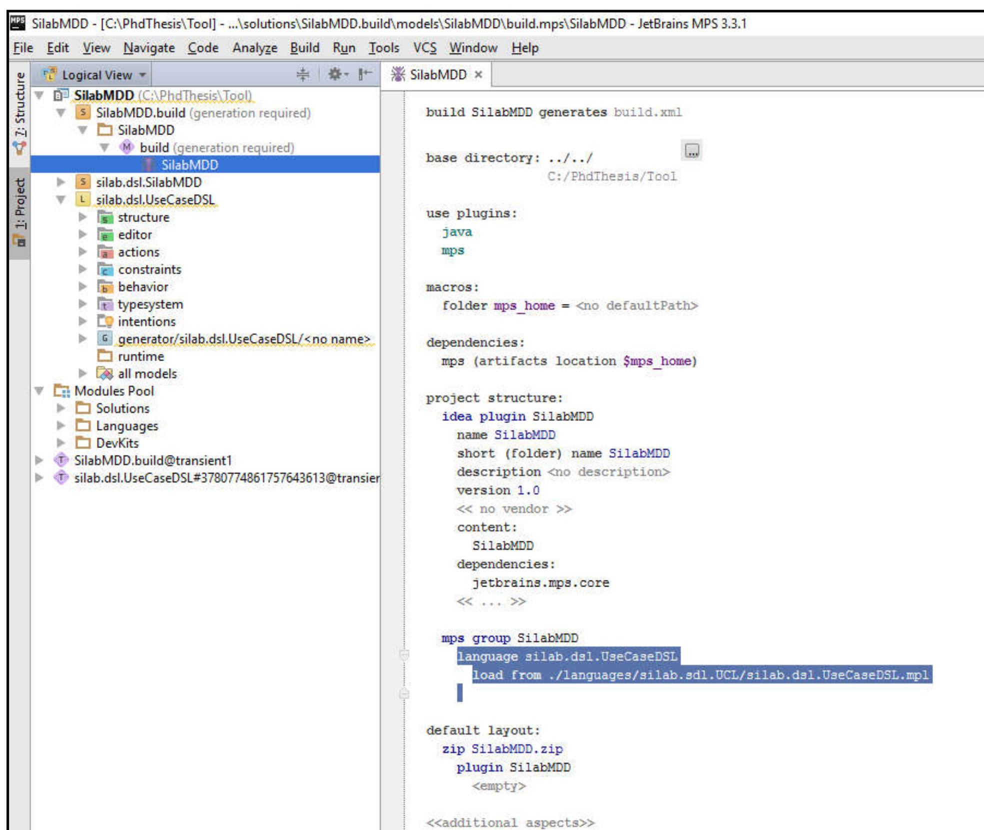
Слика 71. Креирање пројекта (корак 2.)



Слика 72. Избор окружења за које се прави додатак (корак 3.)

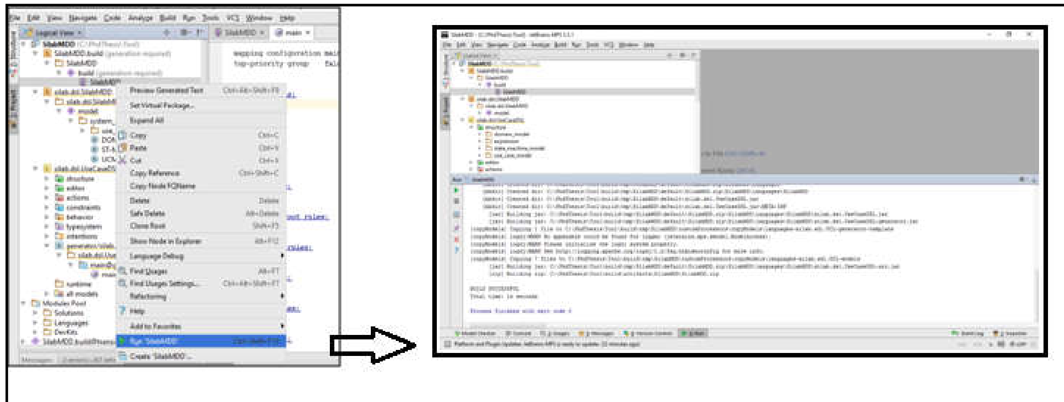


Слика 73. Избор компоненти из пројекта које улазе у додатак (корак 4.)



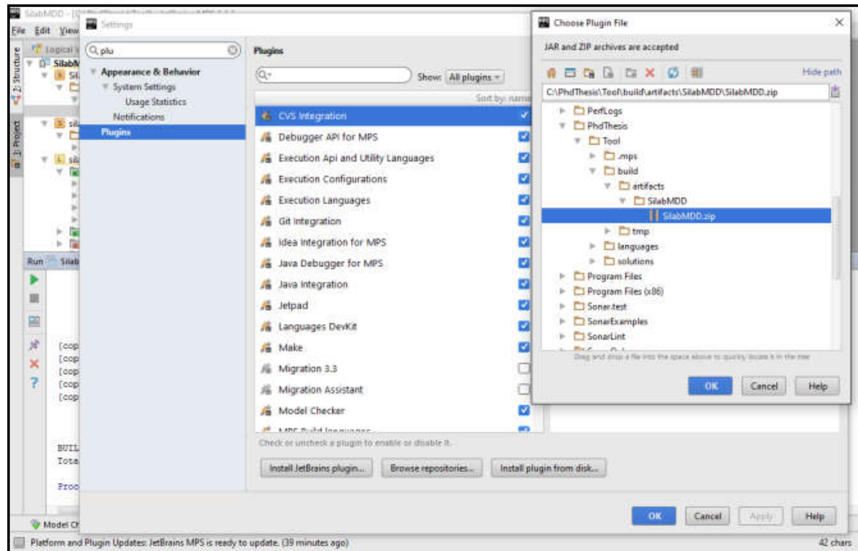
Слика 74. Изглед build скипта

MPS као резултат извршења ова три корака креира *build* скрипт (Слика 74). У оквиру скрипта потребно је дефинисати параметре како би скрипт могао да се изврши. Неки од тих параметара су: локације инсталације MPS-а на диску, локације на којој се налазе додаци који укључују (уколико их има) и локације језика за који се креира додатак. Скрипт се покреће опцијом *Run* (Слика 75). Уколико је скрипт добро конфигурисан и уколико нема грешака у модулу језика за који се креира додатак, додатак се пакује у *zip* архиву након чега је спреман да се укључи (инсталира) унутар одговарајућег развојног окружења (Слика 76).



Слика 75. Покретање *build* скрипта

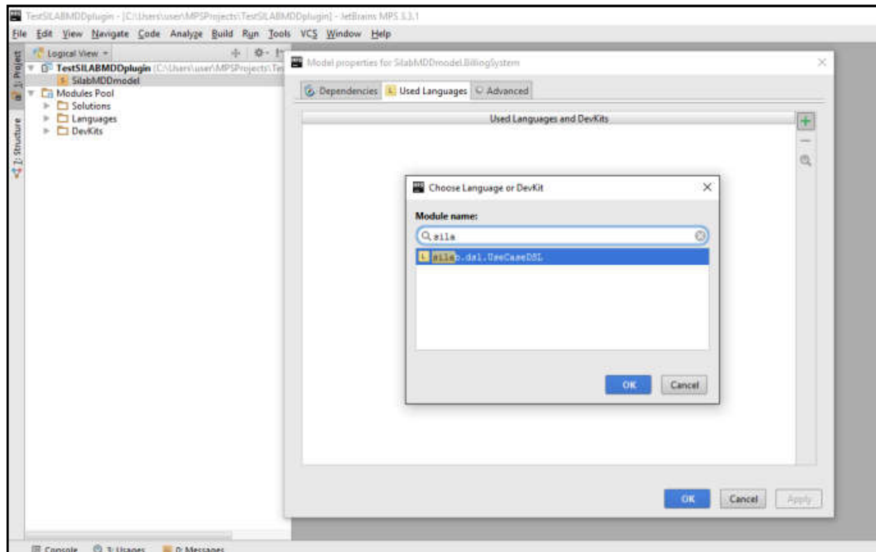
Пре коришћења додатка, додатак је потребно инсталирати. Инсталација новокреираног додатка се врши на исти начин као и инсталација било ког другог додатка (Слика 76).



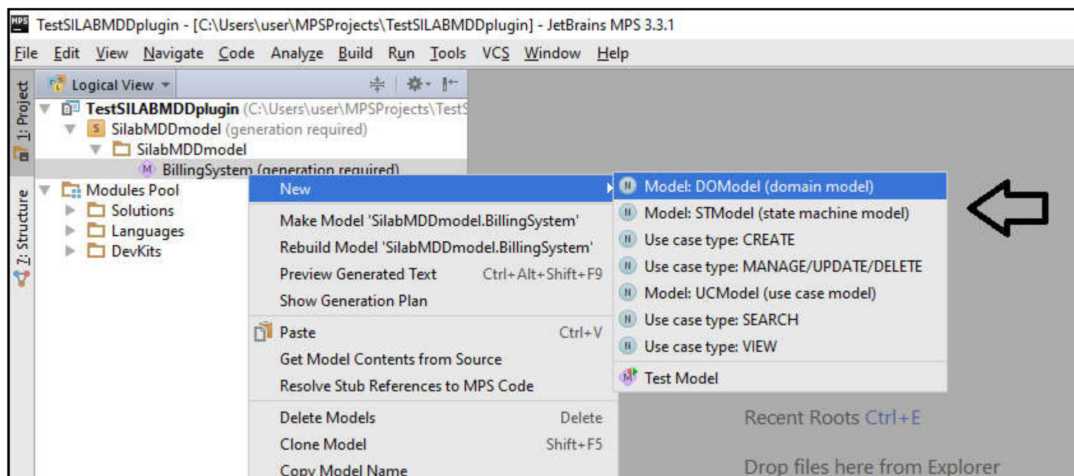
Слика 76. Инсталација *plugin-a*

Након инсталације додатка, исти се може користити за креирање модела. Ниже су приказани кораци које треба следити при креирању модела:

1) Крерати нови пројекат. Креирати *Solution* пројекат (само се у оквиру *Solution* пројекта креирају модели) и у пројекат укључити језике који ће се користити при креирању модела (у нашем случају се укључује UseCaseDSL језик, Слика 77).
2) Креирати одговарајући модел дефинисан у оквиру језика (Слика 78).



Слика 77. Креирање пројекта који користи SilabMDD *plugin*



Слика 78. Креирање модела преко SILAB-MDDTOOLSET

ПОГЛАВЉЕ 6.ЕВАЛУАЦИЈА

У овом поглављу извршена је евалуација предложене *Silab-UCMDD* методе на три различита начина:

a) Анализом предложеног приступа у односу на постојеће приступе.
b) Приказом студијског примера информационог система за издавање рачуна.
c) Анализом резултата пилот-тест евалуације, у којој су учествовали студенти завршне године основних академских студија Факултета организационих наука, смера за Информационе системе и технологије. Они су након експерименталног коришћења алата оцењивали предложени приступ, језик и алат.

6.1. АНАЛИЗА ПОСТОЈЕЋИХ ПРИСТУПА И SILAB-UCMDDM МЕТОДЕ

Као што је у поглављу 4 објашњено, креирање доменског модела (најчешће описаног преко дијаграма класа или модела објеката и веза) на основу функционалних захтева је једна од активности која се јавља у скоро свим методама развоја софтвера. У методама развоја софтвера које користе стратегију засновану на случајевима коришћења, случајеви коришћења играју битну улогу у креирању модела анализе. Моделом анализе описује се структура и понашање система.

Један од основних принципа MDD-а јесте креирање и трансформација модела у свим фазама развоја софтвера. У фази прикупљања захтева, захтеви се најчешће специфицирају у форми природног текста. Ова форма није погодна за аутоматску трансформацију захтева у моделе анализе (моделе структуре и моделе понашања).

У фази прикупљања захтева, препорука је да се модел случајева коришћења и доменски модел користе заједно. *Fortuna* са ауторима у свом раду [FORTUNA, M.H et al. (2008)] наглашава да између ова два модела постоји велики јаз. Даље, он констатује да је ниво аутоматизације трансформације из модела случаја коришћења у доменски модел низак и да у великој мери зависи од интерпретације онога ко моделује систем *(modeler)*. Као примере наводи приступе [BIDDLE, R. et al. (2002)], [LIANG, Y., (2003)], [SUBRAMANIAM, K. (2004)] који предлажу генерисање

доменског модела на основу модела случајева коришћења и приступе [GLINZ, M. (2000)], [KÖSTERS, G. et al. (2001)] који се заснивају на верификацији конзистентности између ова два модела. Један од проблема који је идентификован у овим приступима јесте то што се у овим приступима углавном користи нека форма лингвистичке анализе спецификације случајева коришћења како би се на основу ње добио доменски модел.

Захтеве је у општем случају могуће документовати у форми природног текста, модела или комбиновано. Предмет истраживања ове докторске дисеретације се односи на спецификацију захтева преко случајева коришћења и њихову интеграцију у моделом вођени развој софтвера. Због тога су при компаративној анализи постојећих приступа са предложеним приступом разматрани само они приступи који користе случајеве коришћења као технику за спецификацију захтева. Како је једна од кључних активности метода које су засноване на случајевима коришћења активност која се односи на дефинисање модела анализе на основу случајева коришћења, у компаративној анализи су укључени приступи који предлажу генерисање или валидацију доменског модела на основу модела случајева коришћења.

На основу свеобухватне анализе која је спроведена током истраживања, а у складу са препорукама за систематичан преглед анализе [KITCHENHAM, B.A. et al. (2009)] која је укључила и терцијарну анализу *(tertiary study),* идентификовани су следећи приступи:

1. *Somé* [SOMÉ S. (2003)] је предложио *UCEd* алат који на основу случајева коришћења који се специфицирају у форми рестриктивног природног језика и доменског модела, креира дијаграм прелаза стања. Аутори су предложили и метамодел за текстуалну спецификацију случајева коришћења. Трансформација је подржана алатом и извршава се аутоматски.

2. *Feijs* [FEIJS, L. (2000)] је предложио приступ у коме се на основу случајева коришћења генерише модел понашања система дефинисан дијаграмом секвенци порука *(Message Sequence Charts)*. Према овом приступу сваки корак (исказује се једном реченицом) случаја

коришћења се класификује у три категорије: корак који садржи информације *(information sentences),* корак који се односи на акцију *(action sentences)* и корак који садржи информацију о статусу извршења захтеване операције *(action sentences).* У циљу аутоматизације ове трансформације аутори користе различите форме лексичке анализе.

3. *Subramaniam* са групом аутора [SUBRAMANIAM, K. et al. (2004)] је предложио алат који се назива *Use-Case Driven Development Assistant (UCDA)* који се користи за креирање дијаграма класа, спецификацију случајева коришћења и трансформацију случајева коришћења у модел анализе којим се описује структура система (дијаграм класа). Овај алат користи слободно доступан парсер који је имплементиран у *Python-у.* Пре трансформације модела случајева коришћења у доменски модел, модел случаја коришћења се трансформише у два međумодела, након чега се применом дефинисаних трансформационих правила креира дијаграм класа.

4. *Diaz* је са групом аутора [DIAZ. I. Et al. (2005)] предложио приступ код кога се на основу случајева коришћења креира модел интеракције. Аутори су предложили приступ код кога се секвенцни дијаграми креирају тако што се модел случајева коришћења најпре трансформише у међумодел из кога се уз помоћ синтаксне анализе креирају секвенцни дијаграми.

5. *Insfrán* је са групом аутора [INSFRÁN E. et al. (2002)] предложио приступ у коме се на основу случајева коришћења креира модел анализе описан преко дијаграма класа и дијаграма секвенци. Предложена трансформација није аутоматизована и извршава се ручно.

6. *Śmiałek* [ŚMIAŁEK M. et al. (2007)] је предложио приступ односно методу која је названа ReDSeeDS и која у потпуности прати моделом вођени развој софтвера. У оквиру овог приступа користе се следећи језици:

    a) Језик за спецификацију захтева *(Requirements Specification Language – RSL).*

b) Језик за спецификацију модела пројектовања *(Software Development Specification Language - SDSL)*

c) Језик за трансформацију језика *(MOdel Transformation Language - MOLA)* се користи за дефинисање правила трансформације.

7. *Project-IT* [SILVA, A. (2006)] представља пројекат чији је циљ био да креира комплетно развојно окружење које ће интегрисати активности развоја софтвера: а) спецификацију захтева, б) анализу, в) пројектовање и г) имплементацију (у контексту аутоматског генерисања програмског кода на основу модела), али и активности која се односи на управљање софтверским пројектом. Једна од компоненти овог пројекта јесте *ProjectIT-Requirement* која је задужена за активности које се односе на спецификацију захтева и анализу. Циљ овог пројекта јесте да развије текстуални језик за спецификацију и документовање захтева како би тако дефинисани језик био погодан за коришћење у моделом вођеном развоју софтвера.

8. *Cabral* и *Sampaio* су предложили приступ који на основу модела случаја коришћења генерише формалну спецификацију [CABRAL, G., & SAMPAIO, A. (2008)]. Случајеви коришћења се специфицирају преко контролисаног природног језика (енглеског језика) након чега се трансформишу у CSP процесну алгебру. Аутори су креирали додатак *(plugin)* за *Word* за спецификацију захтева у складу са дефинисаном граматиком. Формална спецификација се може користити за генерисање тест случајева, али није погодна (читљива) за све учеснике *(stakeholders)*.

9. *Tao Yue* са групом аутора је предложио приступ заснован на рестриктивном моделовању случајева коришћења (*Restricted Use Case Modeling – RUCM)* [YUE, T. et al. (2010)]. Према предложеном приступу дефинисан је нови образац (темплејт) који се заснива на добро-дефинисаним рестрикционим правилима како би се избегла двосмисленост и омогућила аутоматска анализа. Алат који је предложен назива се ZEN-RUCM.

Критеријуми на основу којих је компаративна анализа извршена дефинисани су на основу предмета истраживања докторске дисертације. У складу са тим, ови приступи треба да одговоре на следећа питања:

1) Који циљни модел анализе се користи приликом трансформације модела случајева коришћења у доменски модел?
2) Да ли је циљним моделом анализе обухваћена и структура и понашање система?
3) Да ли је ова трансформација модела случаја коришћења у модел анализе аутоматизована или не?
4) Да ли се у предложеним приступима користе алати као подршка за спецификацију случајаве коришћења и њихову трансформацију?
5) Да ли су у процесу трансформације дефинисана правила трансформације и да ли се при трансформацији користе додатни или међумодели?

Табела 2. Компаративна анализа приступа (спецификација, модел анализе)

	Спецификација захтева	**Модел анализе**	
Приступи	**Подржана алатом**	**Модел структуре**	**Модел понашања**
Somé	ДА	НЕ (мора претходно да буде дефинисан)	ДА (дијаграм прелаза стања)
Feijs	непознато	НЕ	MSC
Subramaniam	ДА	ДА (дијаграм класа)	НЕ
Diaz	непознато	НЕ	да (дијаграм секвенци)
Insfrán	ДА	ДА (дијаграм класа)	да (дијаграм секвенци)
Śmiałek	ДА	ДА (дијаграм класа)	ДА (дијаграм активности и дијаграм секвенци)
Project-IT	ДА	ДА (дијаграм класа)	ДА
Cabral u Sampaio	ДА (*plugin* за *word*)	НЕ	CSP процесна алгебра
Tao Yue et al.	ДА	ДА (дијаграм класа)	ДА
Silab-UCMDDM	**ДА**	**ДА (текстуална синтакса)**	**ДА (специфично дефинисан)**

Табела 3. Компаративна анализа приступа (трансформација)

	Трансформација			
Приступи	**Аутоматизована**	**Описана правила трансформације**	**Коришћење међумодела**	**Подржана алатом**
Somé	ДА	НЕ	НЕ	ДА
Feijs	ДА	ДА	ДА	ДА
Subramaniam	ДА	ДА	ДА	Прототип алата
Diaz	ДА	НЕ	НЕ	ДА
Insfrán	НЕ	нема трансформације	НЕ	НЕ
Śmiałek	ДА (велики број корака)	да	НЕ	ДА

	Трансформација			
Приступи	**Аутоматизована**	**Описана правила трансформације**	**Коришћење међумодела**	**Подржана алатом**
Project-IT	ДА	ДА	НЕ	ДА
Cabral u Sampaio	ДА	НЕ	НЕ	ДА
Tao Yue et al.	ДА	ДА	НЕ	ДА
Silab-UCMDDM	ДА (интеграција модела)	ДА (интеграција модела)	НЕ	ДА (интеграција модела)

Поређењем постојећих приступа са предложеном *Silab-UCMDDM* методом може се закључити следеће:

a) У односу на постојеће приступе у *Silab-UCMDDM* методи не постоји потреба за дефинисањем трансформација из модела случајева коришћења у доменски модел јер се током спецификације случајева коришћења врши константно усклађивање модела случаја коришћења и доменског модела. У односу на посматране приступе сличан приступ користи *Somé*, коме је за трансформацију у дијаграм стања неопходно претходно дефинисање доменског модела. Сличан приступ заступа и *Śmiałek*, код кога се до дијаграма класа долази детаљном анализом сваког корака случаја коришћења, за шта се користи речник појмова *(glossary)*.

b) *Silab-UCMDDM* метода за приказ модела структуре користи текстуалну синтаксу, што тренутно може бити недостатак. У оквиру *Silab-UCMDDM* методе у плану је да се развију трансформације у UML моделе структуре и понашања.

c) *Silab-UCMDDM* метода користи специфичан модел понашања који је погодан за генерисање других артифаката у предложеној методи. Овај модел је сличан моделу уговора дефинисаном у [LARMAN, C. (2002)].

d) Слично приступу који предлаже *Somé,* спецификација захтева у оквиру *Silab-UCMDDM* методе подржана је језиком који је формално дефинисан метамоделом.

e) *Silab-UCMDDM* метода у потпуности подржава моделом вођени процес спецификације захтева (слично *Project-IT* приступу и који заступа *Śmiałek*).

f) *Silab-UCMDDM* метода представља једини приступ који комбинује три модела (доменски модел, модел случаја коришћења и модел прелаза стања) и користи их на конзистентан начин.

6.2. СТУДИЈСКИ ПРИМЕР

У овом делу рада извршена је евалуација предложене методе и језика за спецификацију случајева коришћења на студијском примеру информационог система за издавања рачуна (*Billing system*). Према предложеној методи, спецификација захтева подразумева дефинисање три усклађена и компламентарна модела: доменског модела, модела случајева коришћења и модела прелаза стања, док је за генерисање апликације (корисничког интерфејса и модела навигације) потребно креирање додатног модела апликације.

Пре приказа спецификације одговарајућих модела, укратко је дат вербални опис система.

6.2.1. ВЕРБАЛНИ ОПИС СИСТЕМА

Информациони систем за издавање рачуна корисницима система треба да омогући администрацију података о купцима *(customers),* производима *(products)* и рачунима *(invoices).*

Корисник система представља неког ко има кориснички налог и који је повезан са једним или више корисничких профила. У систему постоји неколико кључних корисничких профила:

- профил **корисник-администратор**. Администратор је одговоран за администрацију корисника и корисничких профила. Да би корисник могао да приступи систему, потребно је да претходно буде регистрован. Систем треба да омогући регистровање новог корисника у систему уколико корисник не постоји, или ажурирање података о кориснику уколико је корисник већ регистрован у систему. Током процеса регистрације корисника, за сваког корисника потребно је унети следеће податаке: име, презиме, *e-mail*, адреса и корисничко име. Корисничка шифра се аутоматски генерише од стране система пре него што се корисник евидентира у систему. Систем након тога треба кориснику на *e-mail* да пошаље информације о начину, односно линку преко кога корисник приступа систему. Пре првог приступа систему нопходно је да корисник измени шифру која је генерисана од стране система.

Да би приступио систему, корисник мора бити повезан са неким од унапред дефинисаних корисничких профила. Систем треба да омогући администратору управљање корисничким профилима и додељивање корисничких профила корисницима.

Корисник приступа систему наводећи податке са свог профила (корисничко име и корисничка шифра). Уколико је корисник заборавио своје податке за приступ систему, систем треба да омогући кориснику да промени корисничку шифру шаљући кориснику на e-mail линк на коме може да постави своју нову шифру.

- профил **корисник-оператер**. Корисник са овим профилом одговоран је за администрацију купаца, производа и рачуна. Систем треба да омогући кориснику-оператеру да креира и мења основне податке о купцима. За сваког купца у систему се чувају основне информације (назив, порески идентификациони број, матични број, лого, адреса, подаци о жиро рачуну) и додатне информације које се односе на контакт информације о одговорним лицима, као и информације о томе да ли је купац ВИП или не. Уколико је купац ВИП, тада се том купцу приликом креирања рачуна обрачунава одређени попуст. Износ попуста се може мењати током времена и пре свега зависи од политике компаније.

  Кориснику-оператеру систем треба да омогући администрацију производа. За сваки производ осим основних информација (назив, опис и цена) потребно је чувати информације о томе којој категорији пореза производ припада. Један производ може да припада само једној категорији пореза. Кориснику-оператеру систем треба да омогући управљање овим категоријама.

  За креирање рачуна одговорни су и корисник-оператер и корисник-менаџер. Систем треба да омогући кориснику-оператеру креирање рачуна, њихову измену пре него што се рачун пошаље купцу. Пре самог слања рачуна купцу, рачун је потребно одобрити од стране једног корисника-менаџера. Стога корисник-оператер, пре слања рачуна

купцу, рачун шаље кориснику-менаџеру на одобравање. Уколико рачун буде одобрен, тада се исти штампа, шаље купцу поштом и електронски. Такође, у сваком тренутку корисник може да погледа у ком статусу се налази сваки од рачуна који је креирао, као и да види да ли је рачун плаћен или не.

Систем треба аутоматски да пошаље „узбуну“ за све рачуне који су послати купцима, а који нису плаћени у року од 30 дана од дана када су послати. Поред тога, систем треба да омогући аутоматско архивирање свих плаћених или одбијених рачуна који су старији од једне године.

- профил **корисник-менаџер**. Корисник-менаџер представља једну врсту супервизора који је одговоран за надгледање комплетног процеса издавања рачуна, одобравања и праћења процеса плаћања рачуна. Корисник-менаџер сваки рачун може да одбије, прихвати или врати на измену. Такође, систем треба да омогући кориснику-оператеру креирање рачуна и аутоматско слање истог рачуна купцу. Систем треба овом кориснику да омогући и различите врсте извештаја са агрегираним подацима о купцима, рачунима и производима.

Такође, систем треба да омогући периодично слање (на пример сваког месеца) свих издатих и плаћених рачуна *SAP-Acconting* екстерном систему. Систем за издавања рачуна у овом случају захтеване податке шаље преко изложеног веб сервиса *SAP-Acconting* екстерног система.

Систем за праћење рачуна квартално Порталу пореске институције треба да достави информације о свим издатим рачунима. Ово подразумева, генерисање, односно експорт ових информација у *Excel* шаблон документ дефинисан од стране ове институције. Исти документ се поред портала може по потреби доставити овој институцији и електронски на дефинисану *e-mail* адресу.

6.2.2. СПЕЦИФИКАЦИЈА МОДЕЛА

На основу анализе вербалног модела према UCMDDM методи креирани су следећи модели:

1) Доменски модел
2) Модел случајева коришћења
3) Модел прелаза стања

За аутоматско генерисање апликације дефинисан је и модел апликације. Ниже је дат приказ ових модела.

6.2.2.1. ДОМЕНСКИ МОДЕЛ ЗА СТУДИЈСКИ ПРИМЕР ИЗДАВАЊЕ РАЧУНА

На слици (Слика 79) приказана је спецификација доменског објекта *user*.

```
UC-create-invoice × | UC-search-invoice × | ST-Model - Billing system × | UCModel - Billing sustem × | DOModel - Billing system ×

DOMAIN MODEL : DOModel - Billing system {
  entity : user , description = <no description> {
    <no generalisation>  attribute : unique_code , type = TEXT , identificator = true , description = <no desciption>
                         attribute : first_name , type = TEXT , identificator = false , description = <no desciption>
                         attribute : last_name , type = TEXT , identificator = false , description = <no desciption>
                         attribute : email , type = EMAIL , identificator = false , description = <no desciption>
                         attribute : address , type = TEXT , identificator = false , description = <no desciption>
                         attribute : user_name , type = TEXT , identificator = false , description = <no desciption>

                         [references]

                         operation : register_new_user_and_sen_email , type = save , description = "register new user in the system"

                         [constraints]

  }
```

Слика 79.Спецификација доменског објекта *user*

На слици (Слика 80) приказана је спецификација доменског објекта *customer*.

```
entity : customer , description = <no description> {
  <no generalisation>  attribute : customer_id , type = NUMBER , identificator = true , description = <no desciption>
                       attribute : customer_name , type = TEXT , identificator = false , description = <no desciption>

                       reference : customer_address , targetEntity = address , isPartOf = false , <no cardinality>

                       operation : save_customer , type = save , description = <no description>

                       [constraints]

}
```

Слика 80. Спецификација доменског објекта *customer*

На слици (Слика 81) приказана је спецификација доменских објеката *address* и *product*.

```
entity : address , description = <no description> {
  <no generalisation>   attribute : street_name , type = TEXT , identificator = false , description = <no desciption>
                        attribute : door_number , type = TEXT , identificator = false , description = "cen be 1A or 2B"
                        attribute : locality , type = TEXT , identificator = false , description = <no desciption>
                        attribute : postal_code , type = NUMBER , identificator = false , description = <no desciption>
                        attribute : country , type = TEXT , identificator = false , description = <no desciption>

                        [references]

                        [operations]

                        [constraints]

}

entity : product , description = <no description> {
  <no generalisation>   attribute : product_id , type = TEXT , identificator = true , description = ""
                        attribute : product_name , type = TEXT , identificator = false , description = <no desciption>
                        attribute : product_description , type = TEXT , identificator = false , description = <no desciption>
                        attribute : product_unitary_value , type = DECIMAL , identificator = false , description = <no desciption>

                        reference : product_vat_category , targetEntity = vat_category , isPartOf = false , cardinality ( min = " 1 " , max = " 1 " )

                        [operations]

                        [constraints]

}
```

Слика 81. Спецификација доменског објекта *address* и *product*

На слици (Слика 82) приказана је спецификација доменских објеката *vat_category*.

```
entity : vat_category , description = <no description> {
  <no generalisation>   attribute : id , type = TEXT , identificator = true , description = <no desciption>
                        attribute : vat_category_value , type = DECIMAL , identificator = false , description = <no desciption>

                        [references]

                        [operations]

                        [constraints]

}
```

Слика 82. Спецификација доменског објекта *vat_category*

На слици (Слика 83) приказана је спецификација доменских објеката *invoice* и *invoice_items*.

```
entity : invoice , description = "Document created from our syste" {
  <no generalisation>  attribute : invoice_id_created_by , type = ENUM can be: [ "system generate" OR "select from list reserved number" ] , identificator = false ,
                       attribute : id , type = NUMBER , identificator = true , description = ""
                       attribute : date_created , type = DATE , identificator = false , description = <no desciption>
                       attribute : total_value , type = DECIMAL , identificator = false , description = <no desciption>
                       attribute : vat_total_value , type = DECIMAL , identificator = false , description = <no desciption>
                       attribute : discount , type = DECIMAL , identificator = false , description = <no desciption>
                       attribute : reserved_invoice_id , type = NUMBER , identificator = false , description = <no desciption>
                       attribute : reserved_date , type = DATE , identificator = false , description = <no desciption>

                       reference : customer , targetEntity = customer , isPartOf = false , cardinality ( min = " 1 " , max = " 1 " )
                       reference : invoice_items , targetEntity = invoice_items , isPartOf = true , cardinality ( min = " 0 " , max = " * " )
                       reference : reserved_invoice , targetEntity = reserved_invoice , isPartOf = false , cardinality ( min = " 0 " , max = " 1 " )

                       operation : save_invoice , type = save , description = "Save new invoice in the system"
                       operation : save_invoice_and_sent_to_approve , type = save , description = "Save and send to approve"
                       operation : update_invoice , type = update , description = "Update invoice"
                       operation : sent_to_approve , type = update , description = "Send to approve"
                       operation : update_and_sent_to_approve , type = save , description = "Update invoice and send to approve"
                       operation : approve_invoce , type = update , description = "Approve invoice"
                       operation : reject_invoice , type = update , description = "Reject invoice"
                       operation : search_invoice_by_id , type = retrieve , description = "Search invoice by ID"
                       operation : search_invoice_advanced , type = retrieve , description = "Search invoice by criteria"
                       operation : send_invoice , type = update , description = "Send invoice to customer"

                       constraint : constr-generated_new_invoice_by_system , description = "System generate new invoice number"
                       constraint : constr-user_select_from_list , description = "New invoice number is selected from list"
                       constraint : constr-customer-VIP , description = "Customer for which invoice is created is VIP"

}

entity : invoice_items , description = "Invoice item" {
  <no generalisation>  attribute : item_no , type = NUMBER , identificator = false , description = ""
                       attribute : quantity , type = NUMBER , identificator = false , description = <no desciption>
                       attribute : unitary_value , type = DECIMAL , identificator = false , description = <no desciption>
                       attribute : item_value , type = DECIMAL , identificator = false , description = <no desciption>
                       attribute : vat_item_value , type = DECIMAL , identificator = false , description = <no desciption>

                       reference : product_on_invoice_item , targetEntity = product , isPartOf = false , cardinality ( min = " 1 " , max = " 1 " )

                       [operations]

                       [constraints]

}
```

Слика 83. Спецификација доменског објекта *invoice* и *invoice_items*

6.2.2.2. МОДЕЛ СЛУЧАЈЕВА КОРИШЋЕЊА ЗА СТУДИЈСКИ ПРИМЕР ИЗДАВАЊЕ РАЧУНА

На слици (Слика 84) је приказан део модела случаја за студијски пример *Издавање рачуна.*

```
SilabMDD - [C:\PhdThesis\Tool] - C:\PhdThesis\Tool\solutions\silad.dsl.MODEL\models\silab\dsl\SilabMDD\model.mps\UCModel - Billing sustem - JetBrains MPS 3.3.1
File Edit View Navigate Code Analyze Build Run Tools VCS Window Help
UC-create-invoice × | UC-search-invoice × | UC-register-new-user × | UCModel - Billing sustem × | DOModel - Billing system ×

Use case model: UCModel - Billing sustem
Actors:
  actor ( user-manager , description = <no description> )
  actor ( user-operater , description = <no description> )
  actor ( user-administrator , description = "Adimistrator of the system" )

Use cases:
  (UC) UC-register-new-user base-entity = user , actors = [user-administrator] , use case specification = UC-register-new-user
  (UC) UC-create-invoice base-entity = invoice , actors = [user-manager] , use case specification = UC-create-invoice
  (UC) UC-create-invoice-operatera base-entity = invoice , actors = [user-operater] , use case specification = UC-create-invoice
  (UC) UC-send-invoice base-entity = invoice , actors = [user-operater] , use case specification = [no use case specification]
  (UC) UC-reject_invoice base-entity = invoice , actors = [user-manager] , use case specification = [no use case specification]
  (UC) UC-approve_invoice base-entity = invoice , actors = [user-manager] , use case specification = [no use case specification]
  (UC) UC-update-invoice base-entity = invoice , actors = [user-operater] , use case specification = [no use case specification]
  (UC) UC-create-customer base-entity = customer , actors = [user-manager] , use case specification = [no use case specification]
```

Слика 84. Модел случајева коришћења

На слици (Слика 85) дат је приказ спецификације случаја коришћења *UC-register-new-user.*

```
SilabMDD - [C:\PhdThesis\Tool] - C:\PhdThesis\Tool\solutions\silad.dsl.MODEL\models\silab\dsl\SilabMDD\model.mps\UC-register-new-user - JetBrains MPS 3.3.1
File Edit View Navigate Code Analyze Build Run Tools VCS Window Help
UC-register-new-user × | ST-Model - Billing system × | UCModel - Billing sustem × | DOModel - Billing system ×

Use case: UC-register-new-user
  short description = "register new user in the system" ;
  type = create-entity
  entity = user

  primary actor/s = user-administrator
  secondary actor/s = << ... >>

Use case main flow:
  "1." [actor SET-PROPERTY first_name] << ... >>
       [<< ... >>                     ]
       [<< ... >>                     ]

  "2." [actor SET-PROPERTY last_name] << ... >>
       [<< ... >>                    ]
       [<< ... >>                    ]

  "3." [actor SET-PROPERTY email] << ... >>
       [<< ... >>                ]
       [<< ... >>                ]

  "4." [actor SET-PROPERTY address] << ... >>
       [<< ... >>                  ]
       [<< ... >>                  ]

  "5." [actor SET-PROPERTY user_name   ] << ... >>
       [<< ... >>                      ]
       [system SET-PROPERTY unique_code]
       [  <no details>                 ]

  "6." [case "register new user in the system" : actor send request to system to execute register_new_user_and_sen_email operation]

Sub-flow:
  << ... >>

Exceptional scenario:
  << ... >>
```

Слика 85.Спецификација случаја коришћења *UC-register-new-user*

На слици (Слика 86) дат је приказ спецификације случаја коришћења *UC-create-invoice.*

```
UC-create-invoice ×   ST-Model - Billing system ×   UCModel - Billing sustem ×   DOModel - Billing system ×

Use case: UC-create-invoice
  short description = <no use_case_description> ;
  type = create-entity
  entity = invoice

  primary actor/s = << ... >>
  secondary actor/s = << ... >>

Use case main flow:
  "1." [actor SET-PROPERTY invoice_id_created_by                                                    ]<< ... >>
       [check-constraint: "New invoice number is selected from list" sub-flow: VIP customer           ]
       [check-constraint: "System generate new invoice number" sub-flow: system create new invoice number]
       [<< ... >>                                                                                    ]

  "2." [actor SELECT-REFERENCE {                   ]exceptions: customer_not_exist
       [  customer ( customer )                     ]
       [  cardinality ( min = " 1 " , max = " 1 " ) ]
       [}                                           ]
       [<< ... >>                                   ]
       [<< ... >>                                   ]

  "3." [actor ADD-DETAIL {              ]<< ... >>
       [        ...                     ]
       [      }                         ]
       [<< ... >>                       ]
       [system SET-PROPERTY vat_total_value]
       [  ( "SUM vat item value" )      ]
       [system SET-PROPERTY total_value ]
       [  ( "SUM vat value" )           ]

  "4." [<no use_case_user_input_action>                                              ]<< ... >>
       [check-constraint: "Customer for which invoice is created is VIP" sub-flow: VIP customer]
       [<< ... >>                                                                    ]

  "5." [case "Save new invoice in the system" : actor send request to system to execute save_invoice operation]
       [|                                                                                                     ]

Sub-flow:
# system create new invoice number
Step: <no step_ref>
  << ... >>
# VIP customer
Step: "4."
  "4.1." [actor SET-PROPERTY discount]<< ... >>
         [<< ... >>                  ]
         [<< ... >>                  ]

Exceptional scenario:
# customer_not_exist description = "Customer does not exist" terminate use case

Console   3: Usages   0: Messages   9: Version Control
```

Слика 86. Спецификација случајева коришћења *UC-create-invoice*

На слици (Слика 87) дат је приказ спецификације корака 3 за случај коришћења *UC-create-invoice* .

Слика 87. Спецификације акција додавања ставки на рачун

На слици (Слика 88) дат је приказ спецификације случаја коришћења *UC-search-invoice.*

Слика 88. Спецификација случајева коришћења *UC-search-invoice*

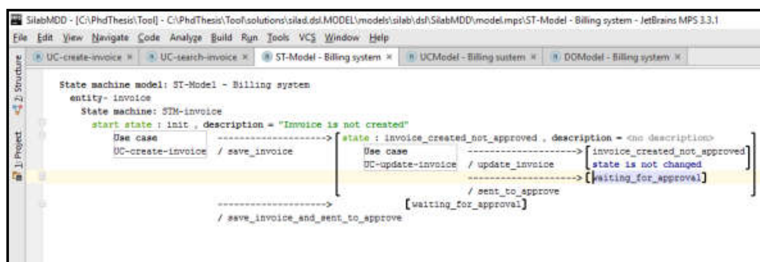
На слици (Слика 89) дат је приказ спецификације случаја коришћења *UC-view-invoice.*

```
UC-create-invoice × | UC-search-invoice × | UC-view-invoice × | ST-Model - Billing system × | UCModel - Billing sustem × | DOModel - Billing system ×

Use case: UC-view-invoice
  short description = "View details for selected invoice" ;
  type = view-entity
  entity = invoice
  primary actor/s = user-manager
                    user-operater
  secondary actor/s = << ... >>

Use case main flow:
  "1." [system RETRIEVAL-entity invoice                                          ] exception scenario: invoice_does_not_exist
       system DISPLAY-ENTITIES with the following details:
        [id
         date_created
         reference customer customer_id
                            customer_name
                            reference customer_address street_name
                                                       door_number
                                                       postal_code
                                                       locality
                                                       country
         reference invoice_items item_no
                                 reference product_on_invoice_item product_id
                                                                   product_name
                                                                   reference product_vat_category vat_category_value
                                 unitary_value
                                 quantity
                                 item_value
                                 vat_item_value
         total_value
         vat_total_value]

Exceptional scenario:
# invoice_does_not_exist description = "Selected invoice does not exist" terminate use case
```

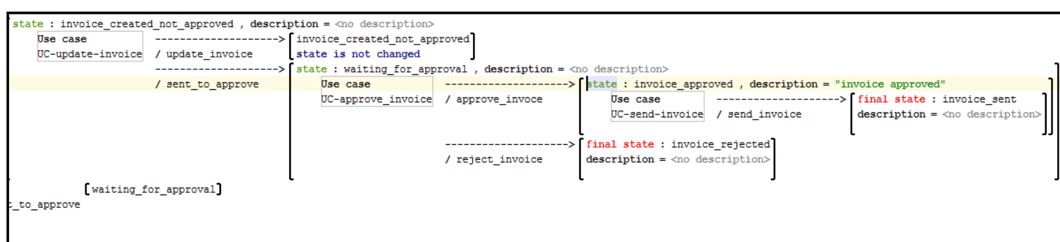
Слика 89. Спецификација случајева коришћења *UC-view-invoice*

6.2.2.4. МОДЕЛ ПРЕЛАЗА СТАЊА ЗА ДОМЕНСКИ ОБЈЕКАТ РАЧУНА

На слици (Слика 90) и слици (Слика 91) дат је приказ дијаграма прелаза стања за документ рачун.



Слика 90. Модел прелаза стања за рачун



Слика 91. Модел прелаза стања за рачун[11]

6.3. ПИЛОТ-ТЕСТ ЕВАЛУАЦИЈА

У циљу боље евалуације предложене Silab-UCMDD методе, UCDSL језика и Silab-MDDToolSet алата у оквиру Лабораторије за софтеврско инжењерство, Факултета организационих наука, Универзитета у Београду током маја 2016. године спроведено је њихово пилот-тестирање. Учесници овог пројекта били су студенти завршне (4. године) основних студија Факултета организационих наука који нису били укључени у развој пројекта, а који су имали задатак да открију потенцијалне грешке у алату или да открију ограничења у *UCDSL* језику. У овом пројекту је учествовало укупно 24 студента.

Овај пилот-тест пројекат је спроведен под следећим условима:

- Пилот-тестирање је вршено у рачунарској сали Факултета организационих наука (контролисано окружење).
- Студенти су радили задатак који је описан у делу 8.3.1.

[11] Слика (Слика 91) представља део слике (Слика 90) које су раздвојене како би модел могао да се прикаже на страници

- Требало је да студенти ураде задатак без претходног знања о методи, језику и алату.
- Студенти су задатак радили на рачунарима на којима је инсталиран Windows оперативни систем, *Јава 8* и *JetBrains MPS* верзија 3.3.1.

Пре него што су студенти кренули да раде задатак који им је додељен, студентима је у 15 минута изложена кратка презентација о Silab-UCMDD методи, UCDSL језика и Silab-MDDToolSet алату. Кроз кратко упутство које им је дато, студентима је демонстрирано на који начин се врши спецификација захтева у Silab-MDDToolSet алату помоћу UCDSL језика. Пример који је студентима демонстриран односио се на спецификацију захтева за студијски пример изадавање рачуна (8.3.1). Након кратке демонстрације, од студената је захтевано да приступе изради задатка који им је додељен. По завршетку израде задатка студенти су попунили анкету чији су резултати приказани ниже у делу 8.3.2 овог поглавља. Просечно време за спецификацију захтева за постављени задатак је било 43 минута.

Валидација *Silab-MDDM* методе и *UseCaseDSL* језика за спецификацију случајева коришћења имала је за циљ да се на основу запажања (импресије) корисника, који нису фамилијарни са предложеном методом и језиком, открију грешке и потенцијални недостаци у самој методи и језику. За ову сврху коришћен је једноставан студијски пример који се односи на креирање поруџбина за набавку производа од потенцијалних понуђача (8.3.1).

6.3.1. ПИЛОТ-ТЕСТ ЗАДАТАК

Информациони систем за креирање поруџбина треба корисницима овог система да омогући креирање поруџбина од добављача који су регистровани у систему. Различити добављачи нуде своје каталоге производа који се евидентирају у систему из којих се касније по потреби врши поручивање жељених производа.

Добављачи приступају систему преко свог корисничког налога. За сваког добављача може бити креирано више корисничких налога. За сваки кориснички налог дефинишу се следећи подаци: име и презиме корисника, *e-mail* корисника, корисничко име и корисничка шифра. Систем треба да омогући креирање новог

добављача, претрагу и измену постојећих података. Креирање добављача и отварање нових корисничких налога врши корисник-администратор.

Добављачи достављају своје каталоге у електронском формату путем веб портала. Систем треба да омогући добављачима унос нових каталога, преглед и измену постојећих католога. Измена каталога је могућа само за каталоге који су тренутно активни. Унос каталога се врши само за оне производе који су регистровани у систему, што значи да приликом уноса каталога за сваки производ који добављачи нуде у својим каталозима потребно је изабрати производ из система који одговара производу који се налази у каталогу. За један производ који постоји регистрован у систему, један добављач може да понуди више различитих производа. За сваки производ који се налази у каталогу се чувају следеће информације: назив производа који је регистрован у систему, назив производа добављача, опис производа, јединична цена и пореска стопа.

Креирање поруџбине врши референт за набавку. Креирање поруџбине се врши увек за једног добављача. Приликом креирања поруџбине, за сваки производ који се поручује потребно је навести из ког каталога и под којом шифром у том каталогу је заведен дати производ, имајући у виду да ће се поруџбина према добављачима вршити искључиво према шифрама добављача. Пре самог поручивања, свака поруџбина мора бити одобрена од стране директора набавке. Директор набавке сваку поруџбину може да прихвати, одбије или измени и тако измењену прихвати.

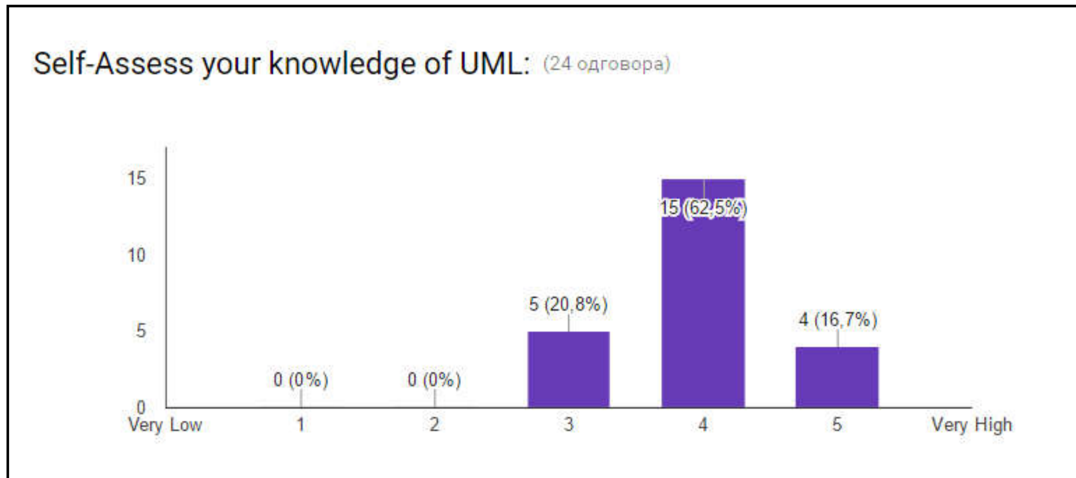
6.3.2. АНАЛИЗА ПИЛОТ-ТЕСТ ЕКСПЕРИМЕНТА

Након спроведеног експеримента, студенти су били у обавези да попуне упитник [WEB-PILOT-TEST]. На готово сва питања (изузев питања које се односило на полну припадност) испитаници су одговоре уносили на скали од 1 до 5.

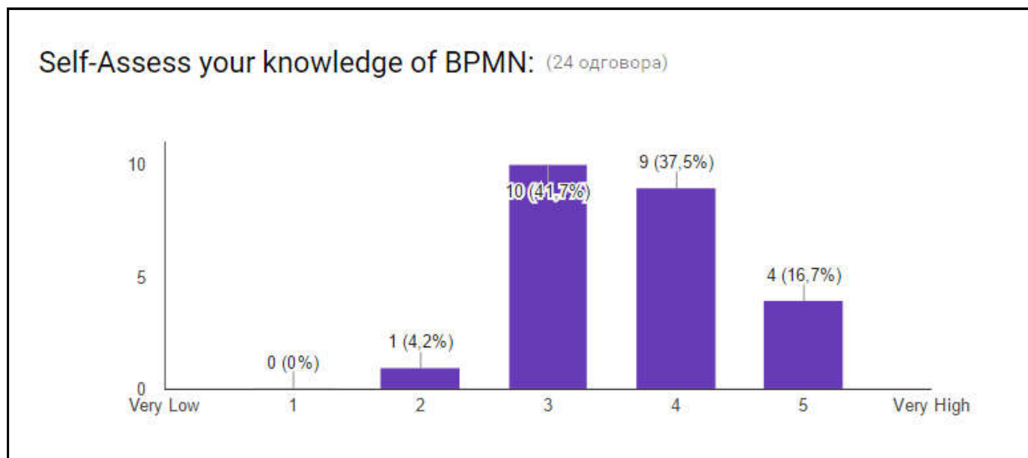
Као што је претходно речено, попуњавању анкете приступило је 24 испитаника (62,5% је било испитаника мушког пола, док је 37,5% било испитаника женског пола).

Упитник је садржао питања која су груписана у 4 групе. Након првог питања које се односило на полну припадност, следећа два питања, која чине прву

групу питања, имала су за циљ да открију колико добро испитаници познају UML и BPMN. Испитаници су сами себе оценили оценама у распону од 1 до 5 (1 је најнижа оцена, док је 5 највиша оцена), а резултати су приказани ниже на слици (Слика 92) и слици (Слика 93).



Слика 92.Анализа оцена познавања UML-а

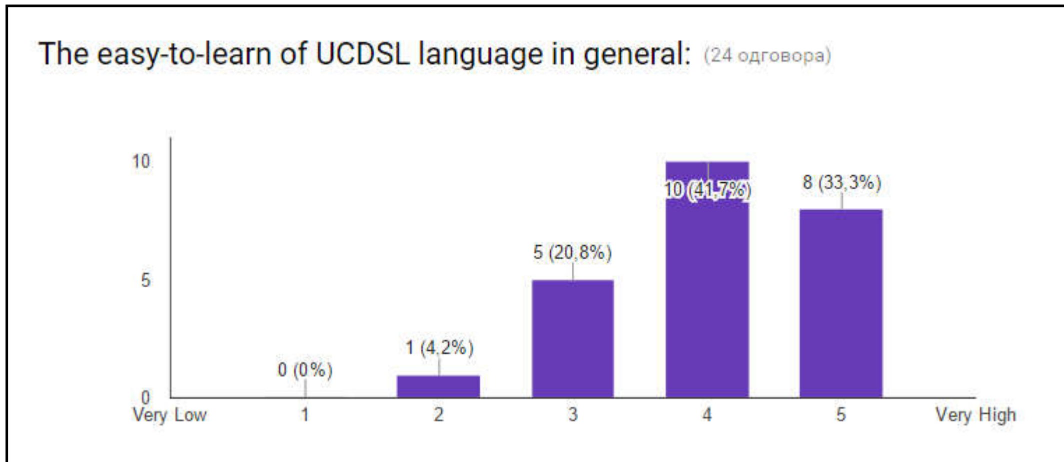


Слика 93. Анализа оцена познавања BPMN-а

На основу резултата може се закључити да студенти сматрају да нешто боље познају UML него BPMN. Остале 3 групе питања су била питања која су се односила на:

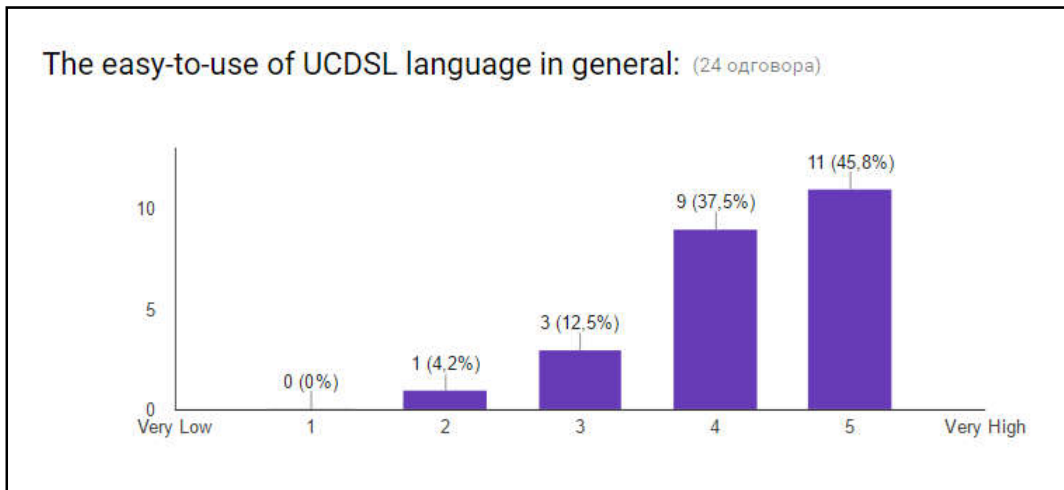
1) **Лакоћу учења и коришћења *UCDSL* језика.**

На слици (Слика 94) приказани су одговори на питање: *Оценом од 1 до 5 оцените једноставност учења UCDSL језика уопштено*[12].



Слика 94. Једноставност учења *UCDSL* језика

На слици (Слика 95) приказани су одговори на питање: *Оценом од 1 до 5 оцените једноставност коришћења UCDSL језика уопштено*[13].



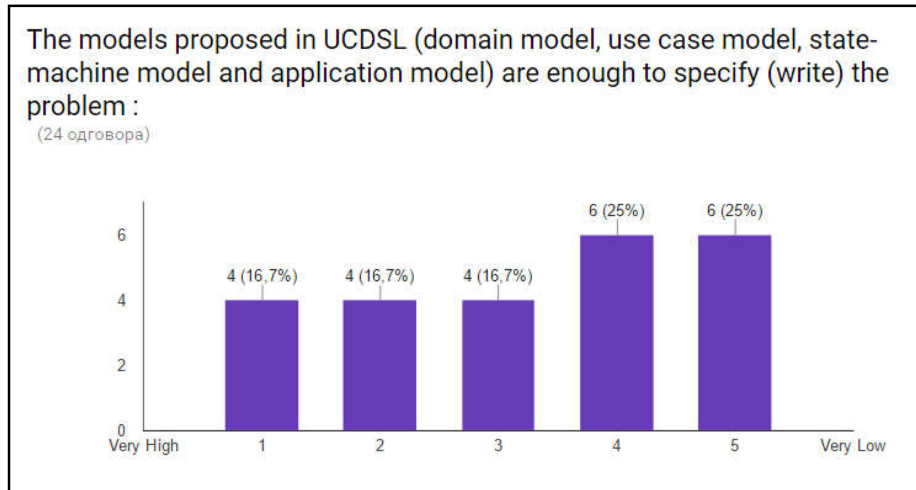
Слика 95. Једноставност коришћења *UCDSL* језика

2) Лакоћу и могућност коришћења предложене методе.

На слици (Слика 96) приказани су одговори на питање: *Оценом од 1 до 5 оцените да ли су модели које су предложени UCMDDM методом*

[12] оригинално питање: *The easy-to-learn of UCDSL language in general*
[13] оригинално питање: *The easy-to-use of UCDSL language in general*

(доменски модел, модел случајева коришћења, модел прелаза стања) довољни да се специфицира проблем.[14]

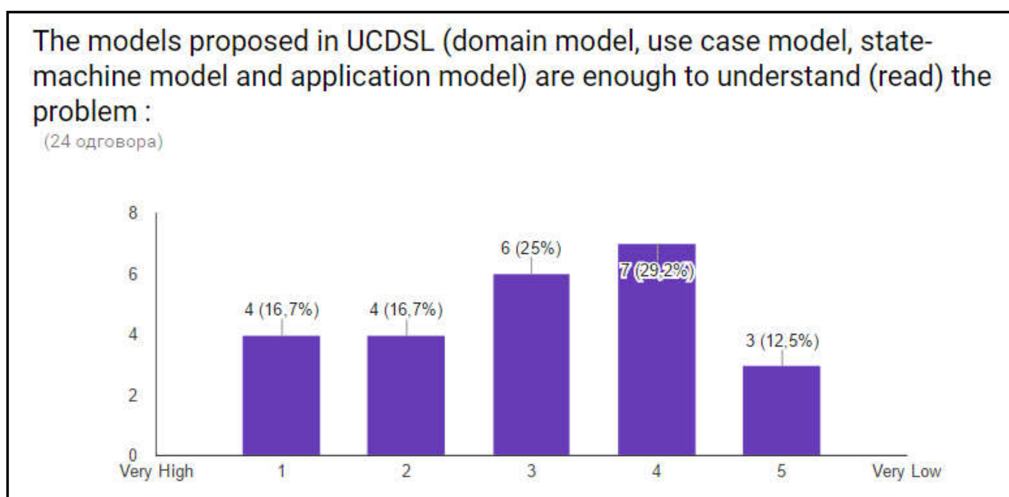


Слика 96. Модели *UCMDDM* методе довољни да се специфицира проблем

На слици (Слика 97) приказани су одговори на питање: *Оценом од 1 до 5 оцените да ли су модели које су предложени UCMDDM методом (доменски модел, модел случајева коришћења, модел прелаза стања) довољни да се разуме проблем.*[15]

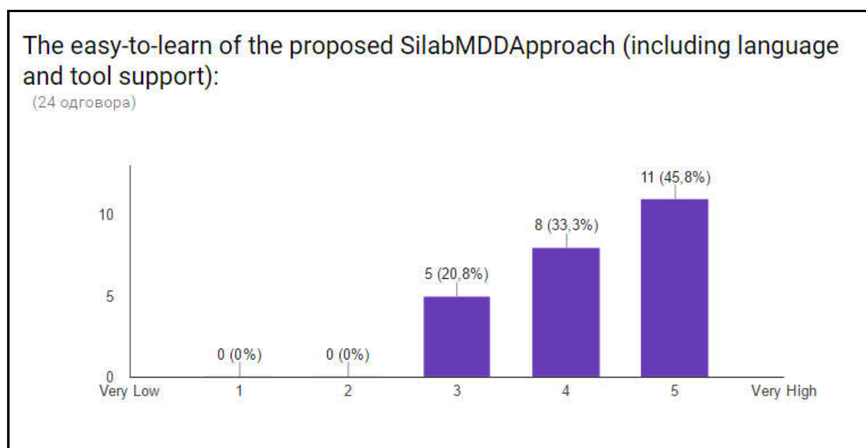
[14] оригинално питање: *The models proposed in UCDSL (domain model, use case model, state-machine model) are enough to specify (write) the problem*

[15] оригинално питање: *The models proposed in UCDSL (domain model, use case model, state-machine model) are enough to understand (read) the problem*



Слика 97. Модели *UCMDDM* методе довољни да се специфицира проблем

На слици (Слика 98) приказани су одговори на питање: *Оценом од 1 до 5 оцените колико је једноставно учење Silab-MDD приступа (језика и алата).*[16]

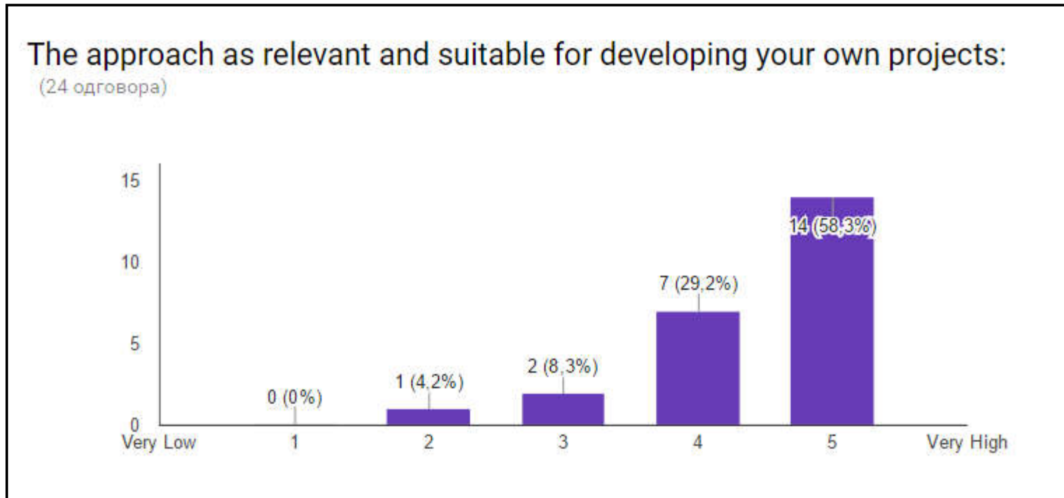


Слика 98. Једноставност учења предложеног Silab-MDD приступа (језика и алата)

На слици (Слика 99) приказани су одговори на питање: *Оценом од 1 до 5 оцените колико је предложени приступ релевантан и одговарајући за креирање ваших пројеката*[17].

[16] оригинално питање: *The easy-to-learn of the proposed SilabMDDApproach (including language and tool support)*

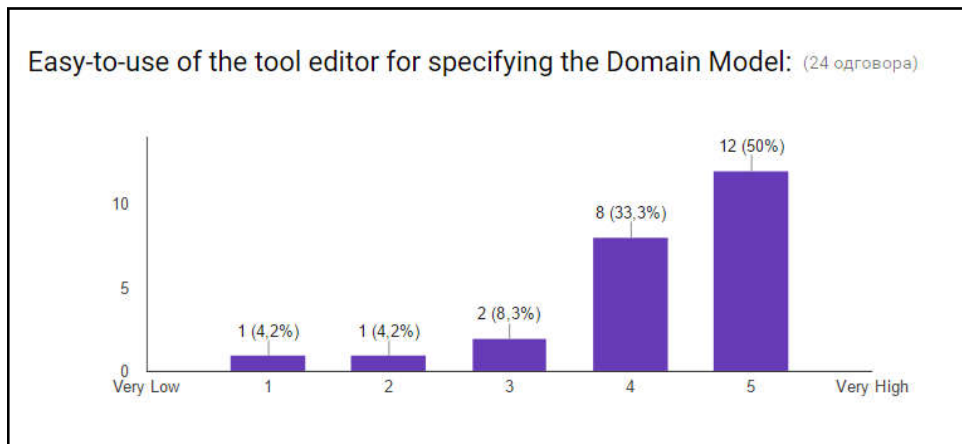
[17] оригинално питање: *The approach as relevant and suitable for developing your own projects.*



Слика 99. Релевантност приступа за сопстеве пројекте

3) **Лакоћу коришћења самог алата односно едитора за спецификацију предложених модела** (доменског модела, модела случаја коришћења, модела прелаза стања и модела апликације).

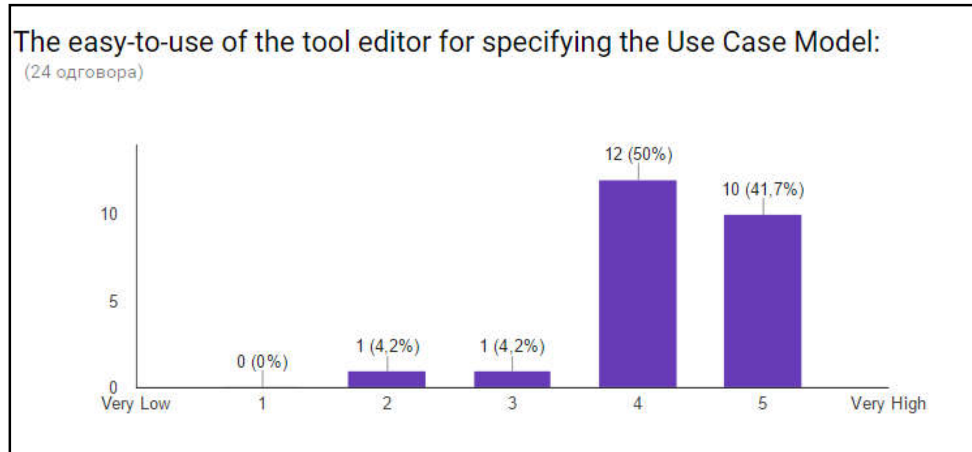
   На слици (Слика 100) приказани су одговори на питање: *Оценом од 1 до 5 оцените лакоћу коришћења едитора за спецификацију доменског модела*[18]



Слика 100. Лакоћа коришћења едитора за спецификацију доменског модела

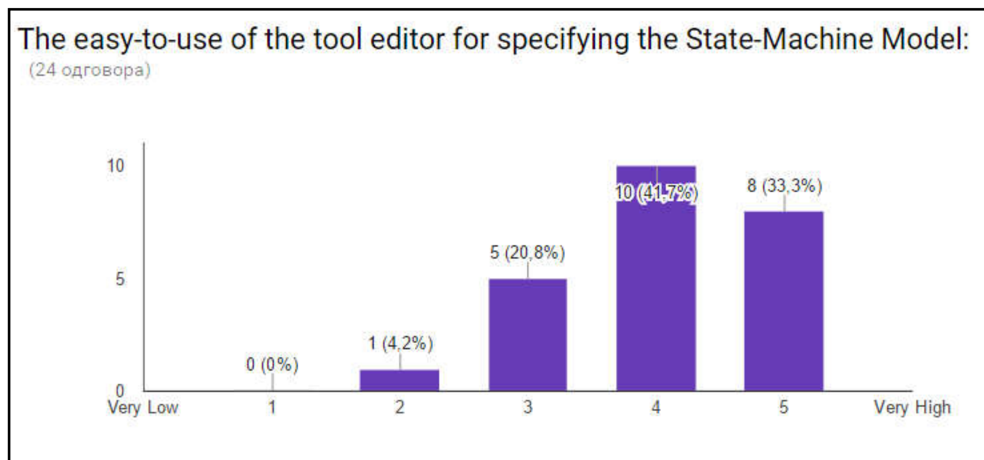
На слици (Слика 101) приказани су одговори на питање: *Оценом од 1 до 5 оцените лакоћу коришћења едитора за спецификацију модела случаја коришћења*[19]

[18] оригинално питање: *Easy-to-use of the tool editor for specifying the Domain Model.*



Слика 101. Лакоћа коришћења едитора за спецификацију модела случаја коришћења

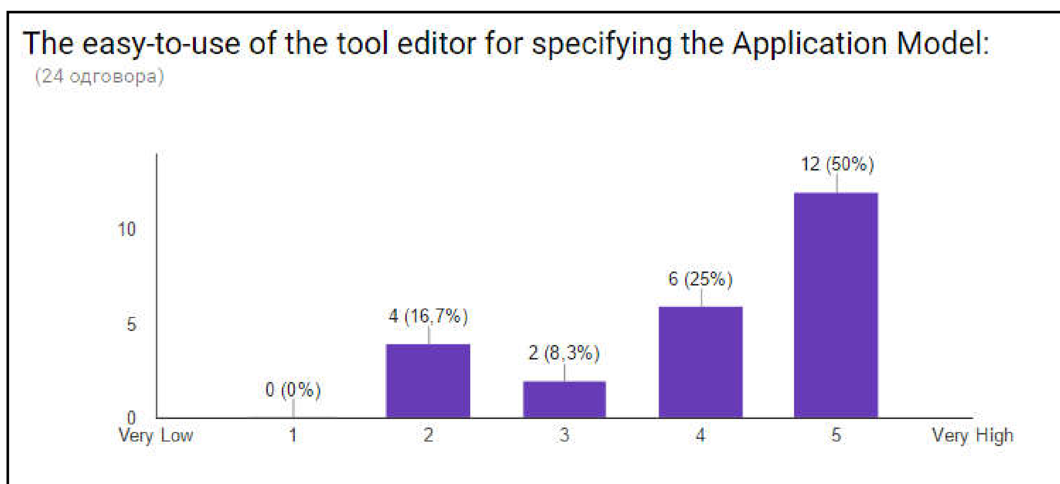
На слици (Слика 102) приказани су одговори на питање: *Оценом од 1 до 5 оцените лакоћу коришћења едитора за спецификацију модела прелаза стања.*



Слика 102. Лакоћа коришћења едитора за спецификацију модела прелаза стања

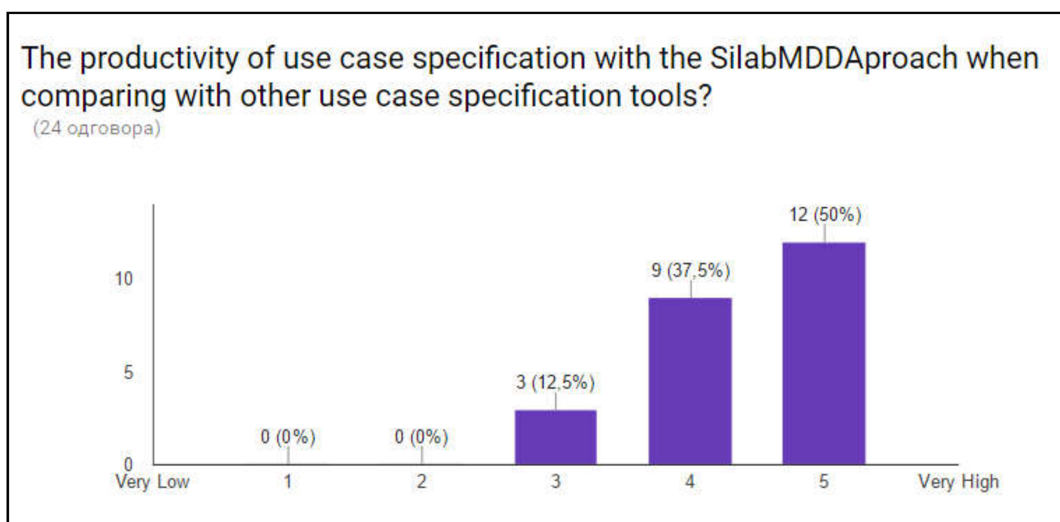
На слици (Слика 103) приказани су одговори на питање: *Оценом од 1 до 5 оцените лакоћу коришћења едитора за спецификацију прототипа апликације*[20].

[19] оригинално питање: *Easy-to-use of the tool editor for specifying the Use case Model.*
[20] оригинално питање: *Easy-to-use of the tool editor for specifying the Application Model.*



Слика 103. Лакоћа коришћења едитора за спецификацију прототипа апликације

На слици (Слика 104) приказани су одговори на питање: *Оценом од 1 до 5 оцените продуктивност предложеног Silab-MDD приступа у односу на постојеће алате за спецификацију захтева*[21].



Слика 104. Продуктивност предложеног *Silab-MDD* приступа у односу на постојеће алате за спецификацију захтева

Подаци који су добијени анкетирањем студената обрађени су статистички у програмском пакету SPSS (верзија 24). На сликама (Слика 105, Слика 106 и Слика 107) приказана је табела фреквенција респективно за групу питања 2, групу питања 3 и групу питања 1.

[21] оригинално питање: *The productivity of use case specification with the SilabMDD Aproach when comparing with other use case specification tools?*

Frequency Table

Друга група питања		Frequency	Percent	Cumulative Percent
Valid	1	8	8.3	8.3
	2	9	9.4	17.7
	3	17	17.7	35.4
	4	28	29.2	64.6
	5	34	35.4	100.0
	Total	96	100.0	

Слика 105. Табела фреквенција за групу питања 2

16	Frequency Table				
17	Трећа група питања				
18			Frequency	Percent	Cumulative Percent
19	Valid	1	1	.8	.8
20		2	7	5.8	6.7
21		3	13	10.8	17.5
22		4	45	37.5	55.0
23		5	54	45.0	100.0
24		Total	120	100.0	

Слика 106. Табела фреквенција за групу питања 3

26	Frequency Table				
27	Прва група питања				
28			Frequency	Percent	Cumulative Percent
29	Valid	2	2	4.2	4.2
30		3	8	16.7	20.8
31		4	19	39.6	60.4
32		5	19	39.6	100.0
33		Total	48	100.0	

Слика 107. Табела фреквенција за групу питања 1

Ако анализирамо укупне резултате испитивања, можемо приметити да одговори испитаних говоре у прилог предложеног приступа. Може се закључити да је предложени приступ заснован на одговарајућим моделима, да је алат једноставан за учење и коришћење, као и да се коришћењем овог приступа повећава продуктивност у процесу развоја софтвера.

ПОГЛАВЉЕ 7. ЗАКЉУЧАК

Развој софтвера вођен моделом представља један од најсавременијих приступа у развоју софтвера. Према овом приступу предлаже се коришћење модела у свим фазама развоја софтвера. У моделом вођеном развоју софтвера модели постају јако битан производ (артифакт) у коме се фокус у развоју софтвера помера са програмског кода на моделе. Модели више нису само неформалне скице у служби комуникације, објашњења, приказа идеје, већ представљају формалну спецификацију како проблема, тако и решења. Према овом приступу, најважнији артифакт у развоју софтвера јесте модел. Модели се најчешће аутоматизованим трансформацијама преводе у друге моделе, семантички обогаћују појединим детаљима, да би се на крају аутоматски на основу њих генерисао програмски код .

У почетним фазама развоја софтвера захтеви се обично описују у форми текста, што отежава трансформацију тако дефинисаних захтева у одговарајуће моделе анализе и интеграцију захтева у моделом вођени развој софтевра.

Методе развоја софтвера које су засноване на случајевима коришћења предлажу коришћење случајева коришћења кроз све фазе развоја софтвера. Случајеви коришћења су своју популарност између осталог добили зато што су кратки, добро структурирани, лаки за читање и што се најчешће документују природним језиком. Како би се избегли проблеми у вези са случајевима коришћења, потребно је дефинисати јасна и комплетна упутства, препоруке, стандарде које треба пратити и обрасце које треба користити како би се ови проблеми смањили или потпуно елиминисали.

Интеграција случајева коришћења у *MDD* захтева детаљну и прецизну спецификацију случајаве коришћења, пре свега у делу који се односи на спецификацију акција сценарија случајева коришћења, као и предуслова и постуслова случајева коришћења.

Креирање доменског модела (најчешће описаног преко дијаграма класа или модела објеката и веза) на основу функционалних захтева је једна од активности која се јавља у скоро свим методама развоја софтера. У методама развоја софтвера које користе стратегију засновану на случајевима коришћења (а посебно објектно-

оријентисаним методама), случајеви коришћења играју битну улогу за идентификовање класа и метода. Обично се у тим приступима случајеви коришћења користе заједно са осталим UML моделима, али и поред тога не постоји добро утемељена техника која омогућава једноставну трансформацију модела класа на основу модела случајева коришћења .

У оквиру докторске дисертације предложен је *Silab-MDD* приступ у развоју софтвера који интегрише модел случајева коришћења са *MDD* приступом. На тај начин полазни модел *MDD-а* постаје *модел случајева коришћења*. Савремене агилне методе развоја софтвера, као што су *Јединствени процес развоја софтвера* и *Лармановa метода развоја софтвера,* засноване су на случајевима коришћења (***Use Case Driven Development***). Наведене методе објашњавају везу случајева коришћења са другим моделима који се добијају током развоја софтвера. Међутим, оне не говоре о аутоматској трансформацији модела током развоја софтвера. У овој докторској дисертацији наведени *Silab-MDD* приступ повезује Ларманову методу развоја софтвера која користи приступ у развоју софтвера вођен случајевима коришћење са *MDD* приступом, што представља оригинални допринос кандидата у области софтверског инжењерства.

У оквиру *Silab-MDD* приступа развијена је *Silab-UCMDM* метода помоћу које се прецизно дефинише поступак прикупљања захтева. Предложена *Silab-UCMDDM* метода истиче важност и неопходност коришћења 3 међусобно конзистентна и комплементарна модела: а) модела случајева коришћења, б) доменског модела и ц) модела прелаза стања. У дисертацији је идентификована директна веза између ова три модела која се пре свега огледа у томе да спецификација акција случајева коришћења треба да се ослања на доменски модел, док се предуслови и постуслови за извршење случајева коришћења дефинишу у моделу прелаза стања. Овај модел прелаза стања се користи за јасно и прецизно дефинисање случајева коришћења. То представља основу предложене *Silab-UCMDDM методе*.

Спецификација захтева у оквиру *Silab-UCMDM* методе омогућена је преко сопственог доменски специфичног језика (*UCDSL*). Помоћу UCDSL се описују три модела:

1. Доменски модел (***Domain Model - DM***) који представља поједностављену верзију UML дијаграма класа.
2. Модел случајева коришћења (***Use Case Model*** - ***UCM***) који служи за дефинисање и спецификацију случајева коришћења.
3. Модел прелаза стања (***State Transition Model*** - ***STM***) који служи за дефинисање дијаграма прелаза стања за сваки доменски објекат и дефинисање скупа случајева коришћења који се могу извршити над објектом у сваком од дефинисаних стања.

Наведени модели су међусобно конзистентни, што значи да се током ажурирања неког од модела непрекидно проверавају и усаглашавају концепти сва три модела. У том контексту *Silab-UCMDM* представља методу за спецификацију захтева система.

У оквиру *Silab-MDD* приступа дефинисан је и начин интеграције Структурне систем анализе, којом се описује функционалност пословног система, са фазама прикупљања захтева и анализе у развоју софтвера. У том смислу креирани су сопствени доменски специфични језици помоћу којих се могу описати дијаграми токова података (DFDDSL) и речник података (DataDDSL).

Имплементација свих предложених доменско специфичних језика извршена је преко *JetBrains MPS* алата за метапрограмирање (*JetBrains MPS metaprogramming system*). Овај алат представља окружење (workbench) за развој доменско-специфичних језика.

Евалуација предложене *Silab-UCMDD* методе урађена је на три различита начина:

- Компаративном анализом предложене методе у односу на постојеће методе.
- Приказом и анализом студијског примера који је развијен предложеном методом.
- Анализом резултата теста у коме су учествовали студенти који су оцењивали предложену методу и *UCDSL* језик за спецификацију и валидацију захтева.

Дисертацијом је доказана неопходност детаљне спецификације случајева коришћења при њиховом интегрисању у *MDD* приступ.

7.1. ОСТВАРЕНИ ДОПРИНОСИ

Главни доприноси ове докторске дисертације су:

1) Израда сопственог *Silab-MDD* приступа који је успео да повеже Ларманову методу развоја софтвера која користи *UCDD* приступ у развоју софтвера са *MDD* приступом.
2) Дефинисање сопствене *Silab-UCMDM* методе којом се прецизно дефинише поступак прикупљања захтева. Ова метода побољшава процеса развоја софтвера јер омогућава непрекидну конзистентност између модела случајева коришћења, доменског модела и модела прелаза стања.
3) Израда сопственог доменски специфичног језика *UCDSL* помоћу кога се могу описати модел случајева коришћења, доменски модел и модел прелаза стања.
4) Израда сопствених доменски специфичних језика *DFDDSL* и *DataDDSL* помоћу којих се могу описати дијаграми токова података и речник података.

Поред тога, у контексту унапређења постојећег стања у области софтверског инжењерства:

1) Направљен је детаљан преглед области (радова и приступа) који се односе на интеграцију софтверских захтева и *MDD-a.*
2) Направљен је детаљан преглед области (радова и приступа) који се односе на прецизну и детаљну спецификацију случајева коришћења.
3) Креиран је алат у коме су интегрисани развијени доменско-специфични језици.

7.2. ПРАВЦИ БУДУЋИХ ИСТРАЖИВАЊА

Имајући у виду резултате који су постигнути у овој докторској дисертацији, правци будућих истраживања биће усмерени ка:

- Унапређењу предложене методе увођењем нових и побољшавањем постојећих доменско-специфичних језика и трансформација.
- Унапређењу предложеног алата у смислу креирање посебне платформе (*Silab-MDD workbench)* засноване на *MPS-у*.
- Примени предложеног решења у пракси, како би се у потпуности потврдила ваљаност решења.

На крају, изражава се нада да ће овај рад допринети да се у будућности више користе случајеви коришћења у интегрисаном моделом вођеном развоју софтвера.

ПОГЛАВЉЕ 8. ЛИТЕРАТУРА

[ABBOTT, RJ., (1986)] R. J. Abbott, *An Integrated Approach to Software Development*, John Wiley, New York, 1986.

[ADOLPH, S. et al. (2003)] S. Adolph, P. Bramble, A. Cockburn, A. Pols, *Patterns for Effective Use Cases*, Addison-Wesley, ISBN: 0-201-72184-8, 2003.

[ALENCAR, F. et al., (2009)] F. Alencar, B. Marín, G. Giachetti, O. Pastor, J. Castro, J. H. Pimentel, *From i* Requirements Models to Conceptual Models of a Model Driven Development Process,* In: PoEM 2009, LNBIP, vol. 39, pp. 99–114. Springer Berlin Heidelberg, 2009.

[ALEXANDER, I. & MAIDEN, N. (2004)] I. Alexander, N. Maiden, *Scenarios, Stories, Use Cases through the Systems Development LifeCycle,* John Wiley and Sons, 2004.

[ALMENDROS-JIMÉNEZ, J. M. & IRIBARNE, L. (2004)] J. M. Almendros-Jiménez, L. Iribarne, *Describing Use Cases with Activity Charts*, In U. K. Wiil (Eds.), Metainformatics, MIS 2004, LNCS 3511 (pp. 141– 159). Salzburg, Austria: Springer, 2004.

[AMBLER, S.W. (2003)] S. W. Ambler, *Agile Model Driven Development Is Good Enough*, IEEE Software, Vol.20, No. 5, (http://www.agilemodeling.com/shared/mda.pdf) (pp. 71 – 73), 2003.

[ANDA, B. et al., (2005)] B. Anda, D. I. K. Sjøberg, *Investigating the Role of Use Cases in the Construction of Class Diagrams*, Empirical Software Engineering 07/2005; 10(3):285-309. DOI:10.1007/s10664-005-1289-3, 2005.

[ANTOVIĆ, I., et al. (2012)] I. Antović, S. Vlajić, D. Savić, M. Milić, V. Stanojević, *Model and software tool for automatic generation of user interface based on use case and data model*, IET Software, The Institution of Engineering and Technology, United Kingdom, 2012.

[ASSAR, S. (2012)] S. Assar, *Model Driven Requirements Engineering*, Mapping the Field and Beyond, MoDRE, 2012.

[BEČEJSKI-VUJAKLIJA, D. (2009)] Prof. Dr. Dragana Bečejski-Vujaklija, *Uvod u Informacione Sisteme*, Fakultet Organizacionih Nauka, Beograd, 2009.

[BERRY, D.M., (1992)] D. M. Berry, *Academic Legitimacy of the Software Engineering Discipline*, Technical Report CMU/SEI-92-TR- 34, Software Engineering Institute, Carnegie Mellon University, 1992.

[BÉZIVIN & GERBÉ, (2001)] J. Bézivin, O. Gerbé, *Towards a precise definition of the OMG/MDA framework*, Proceedings of the 16th IEEE international conference on Automated software engineering, (p. 273). USA, 2001.

[BEZIVIN, J. (2004)] J. Bezivin, *In Search of a Basic Principle for Model Driven Engineering*, UPGRADE - The European Journal for the Informatics Professional, 5(2), (pp. 21-24), 2004.

[BIFFL, S. et al. (2007)] S. Biffl, R. Mordinyi, A. Schatten, *A Model-Driven Architecture Approach Using Explicit Stakeholder Quality Requirement Models for Building Dependable Information Systems*, In Fifth International Workshop on Software Quality. WoSQ'07: ICSE Workshops. doi: 10.1109/WOSQ.2007.1, 2007.

[BITTNER, K. & SPENCE, I. (2003)] K. Bittner, I. Spence, *Use case Modelling*, Addison-Wesley, ISBN 0-201-70913-9, 2003.

[BOEHM, B. W. et al. (2000)] B. W. Boehm, C. Abts, A. Winsor Brown et al., *Software Cost Estimation with COCOMO II*, Prentice Hall PTR, Upper Saddle River, NJ, 2000.

[BOETTGER, K. et al. (2003)] K. Boettger, R. Schwitter, D. Moll`a, and D.Richards, *Reconciling Use Cases via Controlled Language and GraphicalModels*, Web-Knowledge Management and Decision Support, Lecture Notes in Computer Science, Vol. 2543, (pp. 115-128), Springer Verlag, Heidelberg, Germany, 2003.

[BRUEGGE, B. & DUTOIT, AH. (2004)] B. Bruegge, A. H. Dutoit, *Object-oriented software engineering using UML, patterns, and Java*, 2nd edn. Prentice Hall, 2004

[BEZIVIN, J. (2005)] J. Bezivin, *On the unification power of models*, Software & Systems Modeling, 4(2), (pp. 171-188), 2005.

[BIDDLE, R. et al. (2002)] R. Biddle, J. Noble, and E. Tempero, *From Essential Use Cases to Objects*, 1st Intl. Conf. on Usage-Centered, Task-Centered, and Performance-Centered Design (forUse 2002), Ampersand Press, Rowley, MA, 2002, pp. 1-23.

[CAMPAGNE, F. (2015)]F.Campagne, *The MPS Language Workbench*, Fabien Campagne, 2015

[CABRAL, G., & SAMPAIO, A. (2008)] G. Cabral, A. Sampaio, *Formal specification generation from requirement documents*, Electron. Notes Theor. Comput. Sci. 195 (2008) 171–188

[CARROLL, J.M. (1995)] J. M. Carroll, *Scenario-Based Design*, John Wiley&Sons, New York, 1995.

[CHENG, B. H. C. & ATLEE, J. M. (2007)] B. H. C. Cheng, J. M. Atlee, *Research Directions in Requirements Engineering*, in Proceedings Future of Software Engineering (FOSE’07), (pp. 285‑303), 2007.

[CHRISTOPHER, A et al. (1977)] A. CHRISTOPHER, S. ISHIKAWA, M. SILVERSTEIN, M. JACOBSON, I. FIKSDAHL-KING, S. ANGEL, *A Pattern Language*, Oxford University Press, New York, 1977.

[COCKBURN, A. et al. (2002)] S. Adolph, P. Bramble, A. Cockburn, A. Pols, *Patterns for Effective Use Cases*, The Agile Software Development Series, 2002.

[COCKBURN, A. (1997/1)] A. Cockburn, *Goals and Use Cases*, JOOP 10(5), (pp. 35-40), 1997.

[COCKBURN, A. (1997/2)] A. Cockburn, *Using Goal-Based Use Cases*, JOOP 10(7), (pp. 56-62), 1997.

[COCKBURN, A. (2001)] A. Cockburn, *Writing Effective Use Cases*, Addison-Wesley, ISBN 0-201-70225-8, 2001.

[COCKBURN, A. (2003)] A. Cockburn, *Alistair Cockburn April 2002*. http://alistair.cockburn.us/Use+cases%2c+ten+years+later, [преузето: јун, 2016]

[COPLIEN, J.O. (1996)] J. O. Coplien. *Software Patterns*, ISBN 978-1-884842-50-4, 1996.

[CRUZ, A. & FARIA, J.P. (2009)] A. Cruz, J. P. Faria, *Automatic Generation of User Interface Models and Prototypes from Domain and Use Case Models*, In Proceedings of the International Conference on Software Engineering and Data Technologies volume 1, (pp. 169-176), Sofia, Bulgaria, July 2009, INSTICC - Institute for Systems and Technologies of Information, Control and Communication, INSTICC Press, 2009.

[CRUZ, A.M. (2014)] A. M. Cruz, *A pattern language for use case modeling*, Model-Driven Engineering and Software Development (MODELSWARD), 2014 2nd International Conference, Lisbon, 2014.

[CRUZ, A.M.R. & FARIA, J.P. (2010)] A. M. R. Cruz, J. P. Faria, *A Metamodel-based Approach For Automatic User Interface Generation*, In Proceedings of the 13th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (Models 2010), Part 1, LNCS 6394, (pp. 256-270), Oslo, Norway, Springer-Verlag Berlin Heidelberg, 2010.

[CYSNEIROS, L. & do PRADO LEITE J.C.S. (2004)] L. Cysneiros, J. C. S. do Prado Leite, *Nonfunctional Requirements: from Elicitation to Conceptual Models*, IEEE Transactions on Software Engineering 30(5), (pp. 328–350), May 2004.

[DAVIS, A. (1993)] A. M. Davis, *Software requirements - objects, functions, and states*, Prentice Hall international editions, Prentice Hall, ISBN 978-0-13-562174-5, (pp. I-XXVIII), 1993.

[DeMARCO, T. (1979)] T. DeMarco, *Structured analysis and system specification*, Yourdon Press, 1989.

[DEBNATH, N. et al. (2008)] N. Debnath, M. C. Leonardi, M. V. Mauco, G. Montejano, D. Riesco, *Improving Model Driven Architecture with Requirements Models*, In: Fifth International Conference on Information Technology: New Generations, ITNG 2008., (pp. 21–26), 2008.

[DIAZ, I. et al. (2005)] I. Diaz, O. Pastor, A. Matteo, *Modeling Interactions using Role-Driven Patterns*, Proc. IEEE International Conference on Requirements Engineering, (pp. 209-220), 2005.

[DIAZ, I. et al.(2008)] I. Diaz, F. Losavio, A. L. Matteo, O. Pastor, *Specification pattern for use cases*, Information & Management 41(8), (pp. 961–975) 2008.

[DRAZAN, J. & MENCL, V. (2007)] J. Drazan, V. Mencl, *Improved processing of textual use cases: Deriving behavior specifications*, In Proceedings of SOFSEM 2007 LNCS 4362, (pp. 856-86), Harrachov, Czech Republic, January 20 - 26, 2007.

[ELBENDAK, M. et al., 2011] M. Elbendak, P. Vickers, N. Rossiter, *Parsed use case descriptions as a basis for object-oriented class model generation*, J. SYST. SOFTW. 84, 7 (July 2011), 1209-1223. DOI=10.1016/j.jss.2011.02.025 http://dx.doi.org/10.1016/j.jss.2011.02.025, 2011.

[ENGELEN et al., (2010)] L. Engelen, M. van den Brand, *Integrating textual and graphical modelling languages*, Electron. Notes Theor. Comput. Sci., vol. 253, (pp. 105–120), September 2010. [Online]. Available: http://dx.doi.org/10.1016/j.entcs.2010.08.035, 2010.

[EVANS, E. (2004)] E. Evans, *Domain Driven Design*, Addison-Wesley, ISBN 0-321-12521-5, 2004.

[FANTECHI, A. et al. (2003)] A. Fantechi, S. Gnesi, G. Lami, A. Maccari, *Application of Linguistic Techniques for Use Case Analysis*, Requirements Engineering Journal Vol.8, Issue 3, (pp 161-170), Springer-Verlag, 2003.

[FATWANTO, A. & BOUGHTON, C. (2008)] A. Fatwanto, C. Boughton, *Analysis, Specification and Modeling of Functional Requirements for Translative Model-Driven Development*, In International Symposium on Knowledge Acquisition and Modeling, KAM'08 (pp. 859–863). Suzhou, China, 2008.

[FEIJS, L. M. G. (2000)] L. M. G. Feijs, *Natural language and message sequence chart representation of use cases*, Information and Software Technology, vol. 42 (9), (pp. 633-647), 2000.

[FIRESMITH, D.C. (1994)] D. C. Firesmith, *Modeling the Dynamic Behavior of Systems*, *Mechanisms and Classes with Scenarios,* Report on Object Analysis and Design, (pp. 1, 2. 32-36&47), 1994.

[FORTUNA, M.H et al. (2008)] M.H.Fortuna, C.M.L. Werner, M.R.S. Borges, *Info Cases: Integrating Use Cases and Domain Models,* RE'08. Barcelona, Spain

[FORBES, M. (2009)] M. Forbes, *Use Case Survey. Towards Adopting Enterprise Standards for Use Cases,* 2009.

[FOWLER, M. & SCOTT, K. (1997)] M. Fowler, K. Scott, *UML distilled - applying the standard object modeling language*, Addison-Wesley-Longman, ISBN 978-0-201-32563-8, (pp. I-XVIII, 1-179), 1997.

[FRANKEL, D., (2002)] D. Frankel, *Model Driven Architecture: Applying MDA to Enterprise Computing*, John Wiley & Sons, 2002.

[GEORGIADES, G.M. & ANDREOU, A.S. (2013)] M. G. Georgiades, A. S. Andreou, *Patterns for Use Case Context and Content*, ICSR 2013, (pp. 267-282), 13th International Conference on Software Reuse, ICSR 2013, Pisa, Italy, June 18-20, 2013.

[GLINZ, M. (1995)] M. Glinz, *An Integrated Formal Model of Scenarios Based on Statecharts*, In Schäfer, W. and Botella, P. (eds.): Software Engineering – ESEC '95. Proceedings of the 5th European Software Engineering Conference, Sitges, Spain. LNCS 989, Berlin, etc.: Springer, (pp. 254-271), 1995.

[GLINZ, M. (2000)] M. Glinz, *Improving the Quality of Requirements with Scenarios*, Second World Congress on Software Quality, (pp. 55-60), Yokohama, Japan, 2000.

[GOLDSMITH, R.F. (2004)] R. F. Goldsmith, *Discovering Real Business Requirements for Software Project Success*, Artech House Publishers, Boston, London, 2004.

[GUELFI, N. & PERROUIN, G. (2007)] N. Guelfi, G. Perrouin, *A Flexible Requirements Analysis Approach for Software Product Lines*, In: REFSQ 2007, LNCS, vol. 4542, (pp. 78–92) Springer Berlin / Heidelberg, 2007.

[HEUMANN, J. (2008)] J. Heumann, *Tips for writing good use cases*, Requirements Evangelist, IBM Rational Software, 2008.

[HOFFMANN, A. N. V. & LICHTER, H. (2009)] A. N. V. Hoffmann, H. Lichter, *Towards the integration of uml- and textual use case modeling*, Journal of Object Technology, 8(3), (pp. 85–100), http://www.jot.fm/issues/issue_2009_05/article3/, 2009.

[HSIA. P. et al. (1994)] P. Hsia, J. Samuel, J. Gao, D. Kung, *Formal Approach to Scenario Analysis*, IEEE Software 1 1 , 2, (pp. 33-41), 1994.

[JACOBSON et al. (1999)] I. Jacobson, G. Booch, J. E. Rumbaugh, *The unified software development process - the complete guide to the unified process from the original designers*, Addison-Wesley object technology series, Addison-Wesley 1999, ISBN 978-0-201-57169-1, (pp. I-XXIX, 1-463), 1999.

[IEEE- Glosary (1990)] IEEE- Glosary (1990), *IEEE Standard Glossary of Software Engineering Terminology (ANSI)*, [1-55937-067-X] [SH13748-NYF], 1990.

[INSFRÁN, E. et al. (2002)] E. Insfrán, O. Pastor, R. Wieringa, *Requirements Engineering-Based Conceptual Modeling*, Requirements Engineering, vol. 7 (2), (pp. 61–72), 2002.

[ISSA, A. & ALALI, A.(2011)] A. Issa, A. AlAli, *Automated Requirements Engineering: Use Case Patterns Driven Approach*, IET-Software, IET, vol. 5 (3), (pp. 287–303), 2011.

[JAMSHIDI, P. et al., (2009)] P. Jamshidi, S. Khoshnevis, R. Teimourzadegan, A. Nikravesh, and F. Shams, *Toward Automatic Transformation of Enterprise Business Model to Service Model.* In: PESOS '09: Proceedings of the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems, pp. 70-74, IEEE CS, Washington, DC, USA (2009)

[JACOBSON, I. et al. (1995)] I. Jacobson, M. Ericsson, A. Jacobson, *The Object Advantage: Business Process Reengineering With Object Technology*, Addison-Wesley, Reading, Massachusetts, 1995.

[JACOBSON, I. et al. (1992)] I. Jacobson, M. Christerson, P. Jonsson, G. Övergaard, *Object-Oriented Software Engineering – A Use Case Driven Approach*, Reading, Mass., etc.: Addison- Wesley, 1992.

[JACOBSON, I. et al.(1999)] I. Jacobson, G. Booch, J. Rumbaugh, *The Unified Software development Process*, Addison-Wesley, 1999.

[JAMSHIDI, P. et al., (2009)] P. Jamshidi, S. Khoshnevis, R. Teimourzadegan, A. Nikravesh, F. Shams, *Toward Automatic Transformation of Enterprise Business Model to Service Model*, In: PESOS '09: Proceedings of the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems, (pp. 70-74), IEEE CS, Washington, DC, USA, 2009.

[JØRGENSEN, J. B. et al., (2009)] J. B. Jørgensen, S. Tjell, J. M. Fernandes, *Formal Requirements Modeling with Executable Use Cases and Coloured Petri Nets*, Innovations in Systems and Software Engineering, vol. 5(1), (pp. 13–25), 2009.

[KÖSTERS, G. et al. (2001)] G. Kösters, H-W. Six and W. Winter, *Coupling Use Cases and Class Models as a Means for Validation and Verification of Requirements Specifications,* Requirements Engineering Journal 6, Springer London, 2001. pp. 3-17.

[KALNINS, A. et al., (2010)] A. Kalnins, E. Kalnina, E. Celms, A. Sostaks, *From Requirements to Code in a Model Driven Way*, In J. Grundspenkis, M. Kirikova, Y. Manolpoulos, L. Novickis (Eds.), Advances in Databases and Information Systems, LNCS 5968, (pp. 161–168), Riga, Latvia: Springer, 2010.

[KENT, S., (2002)] S. Kent, *Model Driven Engineering. Lecture Notes In Computer Science*, 2335, (pp. 286 – 298), 2002.

[KHERRAF, S. et al., (2008)] S. Kherraf, E. Lefebvre, W.Suryn, *Transformation from CIM to PIM Using Patterns and Archetypes*, aswec, (pp.338-346), 19th Australian Conference on Software Engineering (aswec 2008), 2008

[KITCHENHAM, B.A. et al. (2009)] B.A. Kitchenham, O.P. Brereton, D. Budgen, M. Turner, J. Bailey, S. Linkman, *Systematic literature reviews in software engineering – a systematic literature review*, Inform Software Technol, vol. 51 (1) (pp. 7 – 15), 2009.

[KOCH, N. et al. (2006)] N. Koch, G. Zhang, M. J. Escalona, *Model Transformations from Requirements to Web System Design*, In D. Wolber, N. Calder, C. H. Brooks, & A. Ginige (Eds.), International Conference on Web Engineering (ICWE '06) (pp. 281–288). Palo Alto, CA: ACM, 2006.

[Kostmod 4.0 (2009)] *Kostmod 4.0* (2009), http://rapporter.ffi.no/rapporter/2009/01002.pdf, преузето: јун, 2016.

[KOTONYA, (2000)] G. Kotonya, *Requirements Engineering: Processes and Techniques*, John Wiley and Sons, 2000.

[KRUCHTEN, P. (2004)] P. Kruchten, *The rational unified process: an introduction*, Addison-Wesley Professional, 2004.

[KÜHNE,T. (2006)] T. Kühne, *Matters of (Meta-) Modeling*, Software & Systems Modeling, vol. 5(4), (pp. 369-385), 2006.

[LANGLANDS, M. (2010)] M. LANGLANDS, *Inside The Oval: Use-Case Content Patterns*, Technical report, Planet Project, 2010. Accessed on 2015.

http://planetproject.wikidot.com/use-case-content-patterns

[LARMAN, C. (2002)] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*, Englewood Cliffs, NJ: Prentice-Hall, 2002.

[LARMAN, C. (2004)] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, Third Edition, Prentice Hall, 2004.

[LAZAREVIĆ, B.et al. (2010)] B. Lazarević, Z. Marjanović, N. Aničić, S. Babarogić, *Baze podataka*, Beograd: Fakultet organizacionih nauka, 2010.

[LIEBERMAN, H. et al. (2006)] H. Lieberman, F. Paternò, V. Wulf, Eds., *End-User Development*, Human Computer Interaction Series, Springer, New York, NY, USA, 2006.

[LIU, D. (2003)] D. Liu, *Automating Transition from Use Cases to Class Model*, Thesis, University of Calgary, Department of Electrical and Computer Engineering, 2003.

[LONIEWSKI et al., (2010)] G. Loniewski, E. Insfran, S. Abrahão, *A Systematic Review of the Use of Requirements Engineering Techniques in Model-Driven Development*, in Model Driven Engineering Languages and Systems, D. C. Petriu, N. Rouquette, and Ø. Haugen, eds. Berlin/Heidelberg: Springer, (pp. 213‑227), 2010.

[LONIEWSKI et al., (2011)] G. Loniewski, A. Armesto, and E. Insfran, *Incorporating Model-Driven Techniques into Requirements Engineering for the Service-Oriented Development Process*, In: ME'11: Proceedings of the 2011 Conference on Method Engineering, vol. 351, (pp. 102-107), Springer Boston, 2011.

[LUCENA, M. et al., (2009)] M. Lucena, J. Castro, C. Silva, F. Alencar, E. Santos, J. Pimentel, *A Model Transformation Approach to Derive Architectural Models from Goal-Oriented Requirements Models*, In: OTM 2009 Workshops, LNCS, vol. 5872, (pp. 370–380), Springer Berlin / Heidelberg, 2009.

[MARTÍNEZ, A. et al., (2003)] A. Martínez, J. Castro, O. Pastor, H. Estrada, *Closing the Gap between Organizational Modeling and Information System Modeling*, In: Requirements Engineering (WER2003),The Sixth Workshop, 2003.

[MAZÓN, J.N. et al., (2007)] J. N. Mazón, J. Pardillo, J. Trujillo, *A Model-Driven Goal-Oriented Requirement Engineering Approach for Data Warehouses*, In: ER Workshops, (pp. 255–264), 2007.

[MEIJLER. T. et al. (2010)] T. Meijler, J. Nytun, A. Prinz, and H. Wortmann, *Supporting fine-grained generative model-driven evolution*, Software and Systems Modeling, vol. 9, (pp. 403–424), 10.1007/s10270-009-0144-1. [Online]. Available: http://dx.doi.org/10.1007/s10270-009-0144-1 http://modeling-languages.com/, 2010.

[MENS et al. (2005)] T. Mens, K. Czarnecki, P. Gorp, *Discussion -- A Taxonomy of Model Transformations*, Dagstuhl Seminar on Language Engineering for Model-Driven Software, Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), 2005.

[MEYER, B. (1999)] B. Meyer, *Object-Oriented Software Construction,* Saddle River, NJ: Prentice-Hall, 1999.

[MELLOR, S.J. & BALCER, M.J. (2002)] S. J. Mellor, M. J. Balcer, *Executable UML: A Foundation for Model-Driven Architecture*, Addison-Wesley, 2002. ISBN 0-201-74804-5

[MILLER, J. & MUKERJI, J. (2003)] J. Miller, J. Mukerji, *MDA Guide*, Object Management Group (OMG), Inc, Version 1.0.1, 2003.

[Milicev, D., (2009)] D. Milicev, *Model-Driven Development with Executable UML*, Wiley/Wrox, ISBN 9780470481639, 816 pages, 1999.

[MOF, (2006)] Object Management Group, *Meta-Object Facility (MOF),* Version 2.0, http://www.omg.org/technology/documents/formal/mof.htm, 2006.

[NAVARRO, E. (2007)] E.Navarro, *Architecture Traced from Requirements applying a Unified Methodology*, PhD thesis, Computing Systems Department, UCLM (2007).

[NAKATANI, T. et al. (2001)] T. Nakatani, T. Urai, S. Ohmura, T. Tamai, *A requirements description meta-model for use cases*, Eighth AsiaPacific Software Engineering Conf. (APSEC'01), 2001.

[NAUR, P. & RANDELL, B. (1969)] P. Naur, B. Randell, *Software Engineering*, Report on a conference sponsored by the NATO Science Committee, 1969.

[NICOLAS, J. & TOVAL, A (2009)] J. Nicolas, A. Toval, *On the generation of requirements speci cations from software engineering models: A systematic literature review*, Inf. Softw. Technol, vol. 51(9), (pp. 1291-1307), 2009.

[NICOLAS, J. & TOVAL, A (2009)] B. H. C. Cheng, J. M. Atlee, *Research directions in requirements engineering*, in: Future of Software Eng. (FOSE'07), Minneapolis, USA, 2007.

[NUSEIBEH, B. & EASTERBROOK, S. (2000)] B. Nuseibeh and S. Easterbrook, *Requirements engineering: a roadmap*, in Proceedings of the conference of the Future of Software Engineering, (pp. 35–46), 2000.

[OVERGAARD, G. & PALMKVIST, K. (2004)] G. Overgaard, K. Palmkvist, *Use Cases: Patterns and Blueprints*, Publisher: Addison-Wesley Professional, 2004.

[OMG-MDA] Object Management Group – *MDA (Model Driven Architecture) Guide Version 1.0.1*; 2001 Available at http://www.omg.org/mda/

[OMG-MOF] Object Management Group - *Meta Object Facility (MOF) Core Specification*, v2.4.2.; 2014 Available at: http://www.omg.org/mof/ .

[OMG-UML] United Modeling Language Infrastructure Specification, Version 2.4.1; 2011 Available at http://www.uml.org/

[PASTOR, O. et al. (2011)] O. Pastor, M. Ruiz, S. España, *From Requirements to Code: A Full ModelDriven Development Perspective*, In M. J. Escalona, J. Cordeiro, &

B. Shishkov (Eds.), Software and Data Technologies, ICSOFT 2011, CCIS 303, (pp. 56–70). Seville, Spain: Springer, 2011.

[PASTOR, R. (2006)] R. Pastor, *Model Composition: Definition of Model Composition Properties*. University of York, 2006.

[PASTOR, O. et al. (2001)] O. Pastor, J. Gómez, E. Insfrán, V. Pelechano, *The OO-Method Approach for Information Systems Modeling: from Object-Oriented Conceptual Modeling to Automated Programming*, Information Systems 26(7), 507–534 (2001).

[POHL, K. (2010)] K. Pohl, *Requirements Engineering - Fundamentals, Principles, and Techniques*, Springer 2010, ISBN 978-3-642-12577-5, (pp. I-XVII, 1-813), 2010.

[PRESSMAN, R.S. (1994)] R. S. Pressman, *Software Engineering - A Practitioner's Approach*, McGraw-Hill, 1994.

[REBECCA, J.W. (1999)] J. W. Rebecca, *The Art of Designing Meaningful Conversations*, Reprinted from the Feb 1994 issue of The Smalltalk Report Vol. 3 (5), 1999.

[RAISTRICK, C. et al. (2004)] C. Raistrick, P. Francis, J. Wright, C. Carter, I. Wilkie, *Model Driven Architecture with Executable UML*, Cambridge Univerity Press, 2004. ISBN 0-521-53771-1

[RIVERO, J. M. et al., (2011)] J. M. Rivero,G. Rossi, J. Grigera, E. R. Luna, A. Navarro, *From Interface Mockups to Web Application Models*, In A. Bouguettaya, M. Hauswirth, & L. Liu (Eds.), Web Information System Engineering (WISE 2011), LNSC 6997, (pp. 257–264). Sydney, Australia: Springer Berlin Heidelberg, 2011.

[ROLLAND, C. & ACHOU, C.B. (1998)] C. Rolland, C. B. Achour, *Guiding the construction of textual use case specifications*, Data Knowl. Eng. J. 25 (1–2), (pp. 125–160), 1998.

[ROLLAND, C. et al. (1998)] C. Rolland, C. Souveyet, C. Ben Achour, Guiding *goal modelling using scenarios*, IEEE Transactions on Software Engineering, Special Issue on Scenario Management, Vol. 24 (12), 1998.

[ROSENBERG, D. (2004)] D. Rosenberg, M. Stephens, M. Collins-Cope, *Agile Development with ICONIX Process*, Apress. (ISBN 1590594649), 2015.

[RUMBAUGH, J. E. et al. (2005)] J. E. Rumbaugh, I. Jacobson, G. Booch, *The unified modeling language reference manuel - covers UML 2.0*, Second Edition, Addison Wesley object technology series, Addison-Wesley, ISBN 978-0-321-24562-5, (pp. I-XX, 1-721), 2005.

[SAMARASINGHE, N. & SOMÉ, S. (2005)] N. Samarasinghe, S. Somé, *Generating a Domain Model from a Use Case Model*, Proc. Intelligent and Adaptive Systems and Software Engineering, 2005.

[SÁNCHEZ, P. et al. (2010)] P. Sánchez, A. Moreira, L. Fuentes, J. Araújo, J. Magno, *Model-driven Development of Early Aspects*, Information and Software Technology, vol. 52(3), (pp. 249–273), 2010.

[SARAIVA & SILVA, (2010)] J. Saraiva, A. R. Silva, *A Reference Model for the Analysis and Comparison of MDE Approaches for Web-Application Development*, Journal of Software Engineering and Applications, vol. 3(5), (pp. 419-425), 2010.

[SAVIĆ, D, et al. (2011)] D. Savić, I. Antović, S. Vlajić, V. Stanojević, M. Milić, *Language for Use Case Specification*, Conference Publications of 34th Annual IEEE Software Engineering Workshop, IEEE Computer Society, (pp. 19-26), ISSN: 1550-6215, ISBN: 978-1-4673-0245-6, Limerick, Ireland, 20-21 June 2011, DOI: 10.1109/SEW.2011.9, 2011.

[Savić, D. (2010)] D.Savić, Specifikacija slučaja korišćewa preko SilabReq izvvršivog jezika, Mr teza, FON 2010.

[SAVIĆ, D, et al. (2012)] D. Savic, A. R. da Silva, S. Vlajic, S. Lazarevic, V. Stanojevic, I. Antovic, M. Milic, *Use Case Specification at Different Levels of Abstraction*, QUATIC 2012, (pp. 187-192), 2012.

[SAVIĆ, D, et al. (2014)] D. Savić, S.Vlajić, M. Despotović-Zrakić, From Software Specification to Cloud Model (Chapter in book) High Performance and Cloud Computing in Scientific Research and Education, (pp. 1-350). Hershey,PA: IGI Global. doi:10.4018/978-1-4666-5784-7, 2014.

[SAVIĆ, D, et al. (2015)] D. Savic, S. Vlajić, S. Lazarević, I. Antović, V. Stanojević, M. Milić, A. R. da Silva, *Use case specification using the Silabreq domain specific language*, Computing and Informatics, vol. 34, 2015.

[SAVIĆ, D, et al. (2015)] D. Savic, S. Vlajić, S. Lazarević, I. Antović, V. Stanojević, M. Milić, A. R. da Silva, *SilabMDD - Model Driven Approach*, in Proceedings of ICIST'2015 Conference, 2015

[SCAFFIDI, C. , et al. (2005)] C. Scaffidi, M. Shaw, B. Myers*, Estimating the numbers of end users and end user programmers,* in Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '05), (pp. 207–214), Dallas, Tex, USA, September 2005.

[SEIDEWITZ, E. (2003)] E. Seidewitz, *What Do Models Mean?*, OMG document ad/03-03-31, Object Management Group, 2003.

[SELIC, B. (2008)] B. Selic, *Personal reflections on automation, programming culture, and model-based software engineering*, Automated Software Engineering, vol. 15(3-4), (pp. 379-391), 2008.

[SELIC, B.(2003)] B. Selic, *The Pragmatics of Model-Driven Development*, IEEE Software, vol. 20, (pp. 19–25), 2003.

[SILVA, A.R. et al. (2007)] A. R. da Silva, J. Saraiva, D. Ferreira, R. Silva, C. Videira, *Integration of RE and MDD Paradigms: The ProjectIT Approach and Tools*, Software, IET, vol. 1(6), (pp. 294–314), 2007.

[SILVA, A.R., et al. (2015)] A. R. da Silva, D. Savić, S. Vlajić, I. Antović, S. Lazarević, V. Stanojević, M. Milić, *A Pattern Language for Use Cases Specification*, in Proceedings of EuroPLOP’2015, final version, ACM, 2015.

[SILVA, A. (2006)] A.Silva, C.Videira, J.Saraiva, D.Ferreira and R.Silva, *The ProjectITStudio, an integrated environment for the development of information systems*, In Proc. of the 2nd Int. Conference of Innovative Views of .NET Technologies (IVNET'06), pages 85–103. Sociedade Brasileira de Computação and Microsoft.

[SILVA,A.R., (2015)] A. R. da Silva, *Model-driven engineering: A survey supported by the unified conceptual model*, Computer Languages, Systems & Structures, 2015.

[SLONNEGER, K. et al. (1995)] K. Slonneger, K. Slonneger, B. Kurtz, *Formal Syntax and Semantics of Programming Languages*, Addison-Wesley, 1995.

[ŚMIAŁEK, M. et al., 2007] M. Śmiałek, J. Bojarski, W. Nowakowski, A. Ambroziewicz, T. Straszak, *Complementary Use Case Scenario Representations Based on Domain Vocabularies*, Proc. MoDELS, 2007.

[SOARES, M. S. & VRANCKEN, J. L. M. (2008)] M. S. Soares, J. L. M. Vrancken, *Model-Driven User Requirements Specification Using SysML*, Journal of Software, vol. 3(6), (pp. 57–68), 2008.

[SOME, S. (2009)] S. Some, *A Meta-model for Textaul Use Case Description*, Journal of Object Techology, Zurich, 2009.

[SOMÉ, S. (2006)] S. Somé, *Supporting use case based requirements engineering*, Information and Software Technology, vol. 48 (1), (pp. 43-58), 2006.

[SOMMERVILLE, I. (2015)] I. Sommerville, *Software Engineering*, 10th edition, Addison Wesley Longman Publishing Co. Inc, Boston, 2015.

[STANDISH_GROUP] *Standish group*, http://www.standishgroup.com/, [преузето: јун 2016.]

[STEINBERG, D., et al. (2009)] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks, *Eclipse EMF*, Modeling framework, 2nd editionAddison-Wesley, 2009.

[SUBRAMANIAM. K. et al. (2004)] K. Subramaniam, D. Liu, B. H. Far, A. Eberlein, *UCDA: Use Case Driven Development Assistant Tool for Class Model Generation*, Proc. SEKE'04, 2004.

[SWEBOK- V3, (2010)] *SWEBOK - V3*, https://www.computer.org/web/swebok/v3, [преузето: јун 2016.]

[SWEBOK, 2004] IEEE Computer Society Professional Practices Committee. *SWEBOK®, Guide to the Software Engineering Body of Knowledge*. s.l. : The Institute of Electrical and Electronics Engineers, Inc., 2004.

[SYVERSEN, E. et al. (2003)] E. Syversen, B. Anda, D. I. K. Sjoberg, *An evaluation of applying use cases to construct design versus validate design*, In Hawaii International Conference on System Sciences (HICSS-36), Big Island, Hawaii, January 6Y9, 2003.

[VLAJIC, skripta 2015] S.Vlajic, http://silab.fon.bg.ac.rs/wp-content/uploads/2016/05/ProjektovanjeSoftveraNovaSkripta2015.pdf, преузето: јун 2016

[WANDERLEY, F. et al. (2012)] F. Wanderley, D. S. da Silveira, J. Araujo, M. Lencastre, *Generating Feature Model from Creative Requirements Using Model Driven Design*, In Proceedings of the 16th International Software Product Line Conference, vol. 2, (pp. 18–25). IEEE, 2012.

[WEB-ITU] Internation Telecommuication Union: *Recommendation Z.120 (02/11) Message Sequence Chart (MSC),* http://www.itu.int/rec/T-REC-Z.120, [преузето: јун 2016.]

[WEB-ViPER] *ViPER project site*, http://www.viper.sc, [преузето: јун 2016.]

[WEIDENHAUPT, K. et al. (1998)] K. Weidenhaupt, K. Pohl, M. Jarke, P. Haumer, *Scenarios in System Development: Current Practice*, IEEE Software, vol 15 (2) (pp. 34-45), 1998.

[WIEGERS, K. (2003)] K. Wiegers, *Software Requirements*, Microsoft Press, 2nd edition, ISBN 978-0735618794, 2003.

[WINKLER, S. & PILGRIM J. (2011)] S. Winkler, J. Pilgrim, *A survey of traceability in requirements engineering and model-driven development*, SOFTW. SYST. MODEL, vol 9 (4) (pp. 529-565), DOI=10.1007/s10270-009-0145-0, http://dx.doi.org/10.1007/s10270-009-0145-0, 2010.

[YUE, T. et al. (2009)] T. Yue, http://squall.sce.carleton.ca/pubs/tech_report/TR_SCE-09-03.pdf, 2009.

[YUE, T. et al. (2010)] T. Yue, L. C. Briand and Y. Labiche, *Automatically Deriving a UML Analysis Model from a Use Case Model*, Simula Research Laboratory, Technical Report 2010-15, 2010.

[YUE, T. et al. (2011)] T. Yue, L. C. Briand, Y. Labiche, *A systematic review of transformation approaches between user requirements and analysis models*, Requir. Eng., vol. 16 (2), (pp. 75-99), 2011.

[YOURDON, E. (1989)] E. Yourdon,, *Modern structured analysis*, Englewood Cliffs: Yourdon Press, Vol. 191.1989.

[ZELINKA, L. & VRANIC, V. (2009)] L. Zelinka, V. Vranic, *A Configurable UML Based Use Case Modeling Metamodel*, In: ECBS-EERC, (pp.1-8), 2009.

[ZHANG L. & LANG W., (2008)] L. Zhang, W. Jiang, *Transforming Business Requirements into BPEL: A MDA-Based Approach to Web Application Development*, In: WSCS'08: IEEE Int. Workshop on Semantic Computing and Systems, (pp. 61-66), 2008.

[ZIKRA, J. et al., (2011)] I. Zikra, J. Stirna, J. Zdravkovic, *Analyzing the Integration Between Requirements and Models in Model Driven Development*, in Enterprise, Business-Process and Information Systems Modeling, vol. 81, T. Halpin, S. Nurcan, J. Krogstie, P. Soffer, E. Proper, R. Schmidt, and I. Bider, eds., Berlin/Heidelberg: Springer, (pp. 342‑356), 2011.

ПОГЛАВЉЕ 9. СПИСАК СЛИКА

Слика 1. Хијерархијски приказ модерних извршивих платформи 19
Слика 2. Однос случаја коришћења и сценарија 24
Слика 3. MDD процес [ZIKRA, J. et al., (2011)]. 29
Слика 4. Yue таксономија за систематичан преглед литературе [YUE, T. et al. (2009)] 44
Слика 5. Упошћена Лармановa метода 47
Слика 6. Концептуални приказ *SilabMDD* приступа. 50
Слика 7. Улога трансформација у SILABMDD приступу 51
Слика 8. Основни концети метамодела речника података 52
Слика 9. Део метамодел доменског модела 52
Слика 10. Пресликавање концепата модела захтева у модел корисничког интерфејса 55
Слика 11. Пресликавање концепата метамодела доменског модела у концепте релационог модела 58
Слика 12. Пресликавање концепата метамодела релационог модела у DDL наредбе 59
Слика 13. Пресликавање концепата метамодела доменског модела у концепте модела структуре пројектовања 61
Слика 14. SilabReq пројекат 64
Слика 15. SilabReq трансформације 65
Слика 16. Активности *Silab-UCMDM* методе 69
Слика 17. Основни концепти DOM метамодела 72
Слика 18. Концепти `Property` и `Itype` дефинисани преко *jetbrains.mps.baseLanguage.structure* језика 73
Слика 19. DOM метамодел: Концепт `Reference` и концепт `Generalization` 75
Слика 20. Концепти `Operation` и `Constraint` 76
Слика 21. `DomainModel` концепт дефинисан преко *jetbrains.mps.lang.editor* језика 77
Слика 22. Entity концепт дефинисан преко *jetbrains.mps.lang.editor* језика 77
Слика 23. Пример спецификације доменског објекта *Invoice* 78
Слика 24. Концепт `UseCaseModel` 80
Слика 25. Концепт `UseCaseCreate` 84
Слика 26. Концепт `Stepinput` 85
Слика 27. `UseCaseCreate` концепт дефинисан преко *jetbrains.mps.lang.editor* језика 88
Слика 28. `StepInput` концепт дефинисан преко *jetbrains.mps.lang.editor* језика 88
Слика 29. Спецификација *UC-create-invoice* случаја коришћења 89
Слика 30. UCDSL метамодел: Концепт `UseCaseManage` 90
Слика 31. `UseCaseManageFlow` концепт дефинисан преко *jetbrains.mps.lang.editor* језика 91
Слика 32. Спецификација случаја коришћења измена рачуна *(UC-manage-invoice)* 92
Слика 33. Спецификација корака за додавање стаки рачуна *(ADD-DETAIL invoice_items)* 92
Слика 34. UCM мета-модел: Концепт `UseCaseSearch` 94
Слика 35. UCM метамодел: Концепт StepInputSearch 94
Слика 36. Енумерације дефинисане преко *jetbrains.mps.baseLanguage.structure* језика ..95
Слика 37. UCDSL мета-модел: Концепт `StepRequest` 96
Слика 38. UCDSL метамодел: Концепт `StepUseCaseRequire` 96
Слика 39. `UseCaseSearch` концепт дефинисан преко *jetbrains.mps.baseLanguage.structure* језика 97

Слика 40. StepRequest концепт дефинисан преко *jetbrains.mps.baseLanguage.structure* језика........97
Слика 41. Спецификација случаја коришћења *UC-search-invoice (default search)*........98
Слика 42. Спецификација случаја коришћења *UC-search-invoice (advanced search)*........98
Слика 43. UCDSL метамодел: Концепт UseCaseView........99
Слика 44. UseCaseView концепт дефинисан преко *jetbrains.mps.lang.editor* језика........100
Слика 45.Спецификација случаја коришћења *UC-view-invoice*........101
Слика 46. UCDSL метамодел: Концепт StateMachineModel........103
Слика 47. StateMachineModel концепт дефинисан преко *jetbrains.mps.baseLanguage.structure* језика........104
Слика 48.Машина прелаза стања за доменску класу Invoice........104
Слика 49. Метамодел за NavigationModel концепт........106
Слика 50. Метамодел UIWidget концепта........107
Слика 51. NavigationModel концепт дефинисан преко *jetbrains.mps.baseLanguage.structure* језика........108
Слика 52. Део модела навигације апликације........108
Слика 53. DFDDSL метамодел: Концепт ContextDiagram........111
Слика 54. DFDDSL метамодел: Концепт Diagram........112
Слика 55. DFDDSL метамодел: Концепт ProcessRefDataFlowRefInterfaceRef........113
Слика 56. DFDDSL метамодел за концепт AbstractProcessDatastorage........114
Слика 57. ContextDiagram концепт дефинисан преко *jetbrains.mps.lang.editor* језика и пример конкретног модела........115
Слика 58. Концепт ProcessElement дефинисан преко *jetbrains.mps.lang.editor* језика115
Слика 59. Дефинисање дијаграма контекста........115
Слика 60. Спецификација дијаграма контекста........116
Слика 61. Пример: Декомпозиције процеса IS-Example1........117
Слика 62. Приказ грешке код дефинисања интерфејса који већ постоји са истим именом........118
Слика 63. Приказ грешке код дефинисања тока који се већ појављује........118
Слика 64. DFDDSL метамодел: Концепт DataDictionary........120
Слика 65. DFDDSL метамодел: Концепт StructureAsAField........122
Слика 66. DataDictionary концепт (*structure* концепт и *editor* концепт)........123
Слика 67. Речник података за структуру Invoice........123
Слика 68. Парадигме за развој језика [CAMPAGNE, F. (2015)]........124
Слика 69. AST стабло........125
Слика 70. Креирање *Build Solution-a* (корак 1.)........128
Слика 71. Креирање пројекта (корак 2.)........128
Слика 72. Избор окружења за које се прави додатак (корак 3.)........128
Слика 73. Избор компоненти из пројекта које улазе у додатак (корак 4.)........129
Слика 74. Изглед build скипта........129
Слика 75. Покретање *build* скрипта........130
Слика 76. Инсталација *plugin-a*........130
Слика 77. Креирање пројекта који користи SilabMDD *plugin*........131
Слика 78. Креирање модела преко SILAB-MDDTOOLSET........131
Слика 79.Спецификација доменског објекта *user*........143
Слика 80. Спецификација доменског објекта *customer*........143
Слика 81. Спецификација доменског објекта *address* и *product*........144
Слика 82. Спецификација доменског објекта *vat_category*........144
Слика 83. Спецификација доменског објекта *invoice* и *invoice_items*........145
Слика 84. Модел случајева коришћења........146
Слика 85.Спецификација случаја коришћења *UC-register-new-user*........146

Слика 86. Спецификација случајева коришћења *UC-create-invoice* 147
Слика 87. Спецификације акција додавања ставки на рачун 148
Слика 88. Спецификација случајева коришћења *UC-search-invoice* 148
Слика 89. Спецификација случајева коришћења *UC-view-invoice* 149
Слика 90. Модел прелаза стања за рачун 150
Слика 91. Модел прелаза стања за рачун 150
Слика 92.Анализа оцена познавања UML-а 153
Слика 93. Анализа оцена познавања BPMN-а 153
Слика 94. Једноставност учења *UCDSL* језика 154
Слика 95. Једноставност коришћења *UCDSL* језика 154
Слика 96. Модели *UCMDDM* методе довољни да се специфицира проблем 155
Слика 97. Модели *UCMDDM* методе довољни да се специфицира проблем 156
Слика 98. Једноставност учења предложеног Silab-MDD приступа (језика и алата) 156
Слика 99. Релевантност приступа за сопстеве пројекте 157
Слика 100. Лакоћа коришћења едитора за спецификацију доменског модела 157
Слика 101. Лакоћа коришћења едитора за спецификацију модела случаја коришћења 158
Слика 102. Лакоћа коришћења едитора за спецификацију модела прелаза стања 158
Слика 103. Лакоћа коришћења едитора за спецификацију прототипа апликације 159
Слика 104. Продуктивност предложеног *Silab-MDD* приступа у односу на постојеће алате за спецификацију захтева 159
Слика 105. Табела фреквенција за групу питања 2 160
Слика 106. Табела фреквенција за групу питања 3 160
Слика 107. Табела фреквенција за групу питања 1 160
Слика 108. Класификација корисничких захтева према *Wiegers*-у [WIEGERS, K. (2003)] 186
Слика 109. Однос између модела, метамодела и језика за моделовање 189
Слика 110. Општа 4-нивојска архитектура модела 190
Слика 111. Активности процеса утврђивања захтева према *Sommerville-у* 198

ПОГЛАВЉЕ 10. СПИСАК ТАБЕЛА

Табела 1. Трансформација концепата SilabMDD приступа у концепте SilabUI приступа....55
Табела 2. Компаративна анализа приступа (спецификација, модел анализе)....137
Табела 3. Компаративна анализа приступа (трансформација)....137
Табела 4. *Cockburn-ov* шаблон за опис случаја коришћења....203
Табела 5. RUP-ов шаблон за опис случаја коришћења....206

ПОГЛАВЉЕ 11. ДОДАЦИ

ДОДАТАК А. ДЕФИНИСАЊЕ КЉУЧНИХ ТЕРМИНА

СОФТВЕР

Софтвер *(software)* представља програм или скуп програма са пратећом документацијом и скупом конфигурационих датотека који омогућавају да се ови програми извршавају коректно на очекивани начин [SOMMERVILLE I., (2011)]. Софтвер настаје као резултат извршења неког софтверског процеса који је покренут унутар неког софтверског пројекта.

СОФТВЕРСКИ ПРОЦЕС

Софтверски процес *(software process)* се може дефинисати као скуп акција које се извршавају у одређеном редоследу ради креирања односно производње софтвера које треба да одговара потребама корисника софтвера. Стога, један од битних критеријума које софтвер треба да задовољи јесте да одговара потребама корисника софтвера. Крајњи циљ софтверског процеса није само креирање софтвера који одговара потребама крајњих корисника, већ и креирање вискоко квалитетног софтвера са минимално уложеним напорима и са минимално утрошеним временом. Да би се на овај начин креирао софтвер, потребан је систематичан приступ. Управо се софтверско инжењерство бави систематичним приступом у развоју софтвера.

СОФТВЕРСКО ИНЖЕЊЕРСТВО

Дефиниција **софтверског инжењерства** која се може наћи у IEEE стандардном речнику наглашава природу развоја софтвера као инжењерске дисциплине: *"Софтверско инжењерство представља примену систематичног, дисциплинарног и мерљивог приступа у развоју и одржавању софтвера, што заправо представља примену инжењерства у софтверу"* [IEEE- Glosary, (1990)].

Свеобухватну дефиницију *софтверског инжењерства* дао је *Berry* који наглашава неколико њених различитих аспеката: *"Софтверско инжењерство представља форму инжењерства која примењује систематичан и дисциплинован приступ у развоју и одржавању софтвера, чији су резултати*

мерљиви и која поштује добре принципе рачунарске науке, пројектовања, инжењеринга, менаџмента, математике, психологије, социологије и других дисциплина које су неопходне за економично стварање, развој и одржавање поузданог софтвера ” [BERRY, D.M., (1992)]].

Pressman дефинише *софтверско инжењерство* као *“дисциплину која интегрише процес развоја софтвера са: а) методама које се користе за анализу, пројектовање и тестирање, б) техникама за управљање које се користе за контролу и праћење софтерских пројеката и ц) алатима који се користе као подршка процесима, методама и техникама“* [PRESSMAN, R.S. (1994)]

ЗАХТЕВ

Термин *захтев* различито се дефинише од стране различитих аутора. Често због тога зна да настане велика конфузију јер се термин *захтев* често ближе одређује терминима као што су: *пословни захтеви (business requirement), кориснички захтеви (user requirement), системски захтеви (system requirement), функционални захтеви (functional requirement), софтверски захтеви (software requirement), технички захтеви (technical requirement), ограничења (constraint) или својства (feature).* Границе односно разлике између ових термина често нису јасно дефинисане, па се један исти термин захтева користи у различитим значењима у зависности од аутора. Ниже су дате неке од најчешће коришћених дефиниција овог термина.

Дефиниција захтева

Према SWEBOK пројекту, софтверски захтев се дефинише као *„...својства које софтвер треба да садржи у циљу решавања неког реалног проблема“* [SWEBOK- V3, (2010)] .

Kotonya [KOTONYA, (2000)] је рекао да софтверски захтеви *„изражавају потребе и ограничења дефинисана на софтверском производу која доприносе решењу проблема у стварном свету“.*

IEEE дефинише *захтеве* као *„(1) услов и могућност која је потребна кориснику да би решио проблем или остварио циљ; (2) услов или могућност коју систем или компонента система мора да задовољи односно поседује да би се*

задовољио дефинисан уговор, стандард, спецификацију или било који други захтевани документ; (3) документована репрезентација услова или могућности дефинисаних као код (1) и (2)" [IEEE- Glosary (1990)].

Према *Sommerville-у* софтверски захтеви представљају „спецификацију сервиса које систем треба да обезбеди" [SOMMERVILLE, I. (2015)]. Сваки сервис који систем обезбеђује има одређену вредност за корисника система. Пружањем сервиса систем помаже кориснику да реши одређени проблем. Софтверским захтевима се не ретко дефинишу и ограничења која постоје унутар система.

Класификација захтева

Слично као и приликом дефинисања захтева, тако и приликом његовог класификовања, различити аутори на различите начине класификују захтеве. *Sommerville* захтеве дели на [SOMMERVILLE, I. (2015)]:

a) **Корисничке захтеве**

Кориснички захтеви се дефинишу на високом нивоу апстракције, обично једноставном формом природног (говорног) језика, избегавајући стручне термине, кога често прате једноставне табеле, графикони и интуитивни дијаграми. Њима се специфицирају сервиси односно функције које систем треба да обезбеди. Приликом дефинисања корисничких захтева треба избећи коришћење формалних нотација.

b) **Системске захтеве**

Системски захтеви се са друге стране дефинишу као кориснички захтеви нижег нивоа апстракције. Дакле, они представљају проширену верзију корисничких захтева. Системским захтевима се даје детаљна функционална спецификације софтверског система. Јасно, прецизно дефинисани системски захтеви служе као полазна тачка за развој софтвера. Стога се они, дефинишу односно специфицирају преко структурираног природног језика. Структурираном нотацијом дефинишу се обрасци (template) за дефинисања захтева. Потреба за дефинисањем софтверских захтева на различитим нивоима апстракције је уследила услед великог број корисника система.

Karl Wiegers захтеве дели на [WIEGERS, K. (2003)]:

a) **Функционалне захтеве**

Функционалним захтевима се дефинише шта систем треба да ради.

b) **Не-функционалне захтеве**

Не-функционалним захтевима се дефинише како систем треба да одговори на функционалне захтеве.

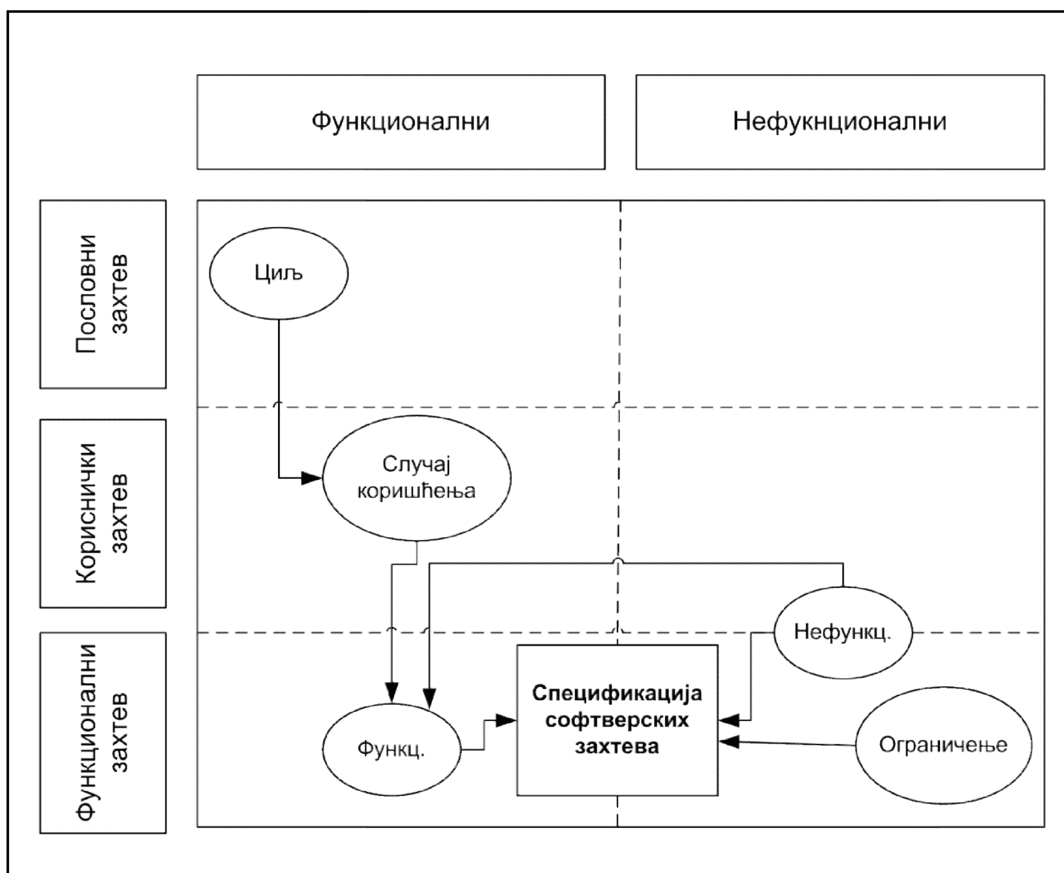
Такође, *Wiegers* захтеве дели на:

a) **Пословне захтеве** *(Business requirements).* Пословним захтевима се дефинише зашто треба покренути и реализовати неки софтверски пројекат односно због чега треба развити софтверски систем. Овим захтевима се дефинише које су то користи које се очекују од софтверског система. *Wiegers* специфицира ове захтеве у документу који се назива *Vision* односно *Scope* документ.

b) **Корисничке захтеве** *(User requirements).* Овим захтевима се дефинише шта је то што корисник очекује од софтверског система, односно које су то функционалности будућег софтверског система из угла корисника система. Њима се специфицирају информације **о томе "шта"** софтверски систем треба да омогући кориснику система. *Случајеви коришћења, сценарија, корисничке приче*, *табела догађаја и одговора (event-response table)* представљају технике које се могу користити за спецификацију корисничких захтева [ALEXANDER, I. & MAIDEN, N. (2004). Случајеви коришћења се специфицирају у посебном документу (документ спецификације случајева коришћења), а неки аналитичари укључују и спецификацију случајева коришћења у документ спецификације софтверских захтева.

c) **Функционалне захтеве** *(Functional requirements)* Ови захтеви садрже такође информације о томе "**шта"** програмер треба да уради како би кориснички захтеви били испуњени. Ови захтеви су познати и као захтеви понашања *(behavioral requirements)* и често знају да се специфицирају традиционалним *"shall"* изразима. Ови захтеви се специфицирају у документу спецификације софтверских захтева *(software requirements specification)*. Овај документ користе и анлитичари за комуникацију са програмерима, тестерима и другим учесницима у пројекту.

Структурирани приказ ових захтева дат је на слици Слика 108.



Слика 108. Класификација корисничких захтева према *Wiegers*-у [WIEGERS, K. (2003)]

Пословни захтеви дефинисани у форми циља који се жели остварити реализују се преко скупа корисничких захтева специфицираних у форми случајева коришћења. На основу случајева коришћења и пословних правила дефинишу се функционални захтеви софтверског система.

ДОКУМЕНТ СПЕЦИФИКАЦИЈЕ ЗАХТЕВА

Документовани захтеви према [POHL, K. (2010)] представљају основу за остале активности процеса развоја софтвера па стога имају велики значај за сваки софтверски пројекат. *Pohl* прави разлику између документованих захтева и специфицираних захтева. Током различитих активности процеса утврђивања захтева, захтеви могу да се документују у различитим форматима, често неформално и на неструктурирани начин. Ово се нарочито односи на документовање захтева током активности процеса откривања захтева.

Специфицирани захтеви представљају посебну врсту документованих захтева, јер сви захтеви који се документују не морају и да се специфицирају.

Језик за спецификацију захтева

Концептуални модели који се користе за спецификацију захтева се називају модели захтева *(requirements model).* Ови модели се креирају уз помоћ језика за спецификацију и моделовање захтева. Ови језици се међусобно разликују у погледу нотације, али и нивоа формализма. Шире посматрано, ови језици се могу поделити у следеће групе односно категорије: природни језици *(unconstraint natural language),* контролисани природни језици *(controlled natural language)*, графички језици за моделовање *(graphical modeling language),* псеудокод *(pseudoceode)* и формални језици *(formal languages).*

МОДЕЛ

Модел представља упрошћену слику или представу о систему који се посматра. Он се креира како би помогао у анализи система кога представља [BÉZIVIN & GERBÉ, (2001)], [KÜHNE, T., (2006)], [SARAIVA & SILVA, (2010)]. Имајући у виду да модел представља поједностављену слику система или неког дела система, за његово разумевање често је потребно креирати више модела. За модел се каже да *„представља апстракцију система који се посматра (односно "Universe of Discourse") који већ постоји или треба да постоји у будућности"* [POHL, K. (2010)] [SILVA, A.R., (2015)]. Такође, модел се дефинише и као: *„скуп исказа о систему који се посматра"*[SEIDEWITZ, E. (2003)]. *Kühne* сматра да је *„модел апстракција система која омогићава предвиђања или доношење закључака на основу њега"* [KÜHNE, 2006]. Поред тога, *„модел представља представу о систему који занемарује одређена својства система која нису битна са одређене тачке гледишта"* [SELIC, B., (2003)]. *Silva* каже: *„Модел као систем нам омогућава да нешто представимо или да дамо одговоре на нека питања без потребе да разматрамо директно систем за који је креиран модел"* [SILVA, 2015]. Модели се могу класификовати на физичке и концептуалне [POHL, K. (2010)]. Концептуални модели се обично документују графички. Сваки модел карактеришу следећа својства: 1) представља слику система који се посматра тако да је могуће на основу модела идентификовати оригинални систем који модел представља, 2) смањује детаље, представља упрошћену слику система који се посматра и 3)

прагматичан је у смислу да је креиран са унапред дефинисаном наменом. Концептуални модели се креирају уз помоћ специфичних језика за моделовање који имају прецизно дефинисану синтаксу и семнатику . У том контексту, према *Kleppe-у* [KLEPPE, et al. (2003)] под моделом се подразумева *систем или део система специфициран преко јасно дефинисаног језика који може бити аутоматски интерпретиран од стране рачунара*. Модели се могу поделити на дескриптивне *(descriptive)* моделе и прескриптивне *(prescriptive)* моделе. За разлику од дескриптивних модела који се најчешће користе за концептуални приказ, разумевање проблема и анализу, прескипривни модели су формални, конзистентни и комплетни и користе се углавном за конструкцију софтверских система и представљају основу за развој софтвера који је вођен моделом (*моделом вођени развој софтвера*). Формално дефинисани модели су засновани на мета-моделима.

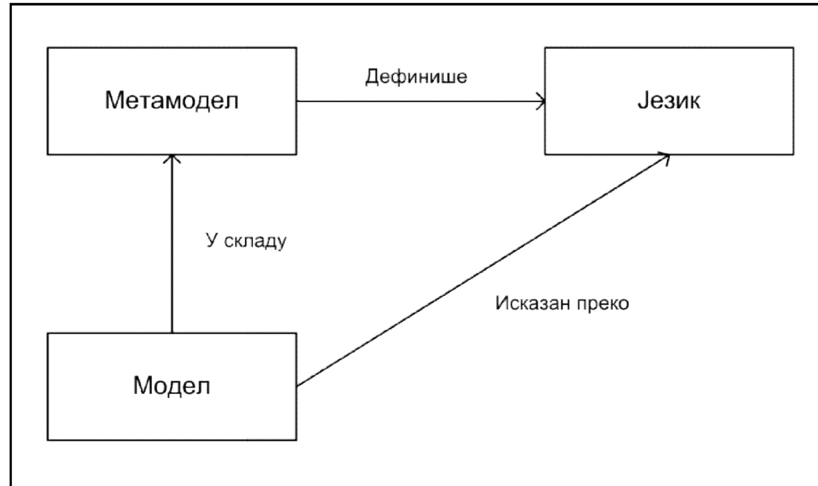
МОДЕЛОВАЊЕ

Моделовање представља процес креирања и/или измене модела. Моделовање представља начин да се размисли о свим питањима пре кодирања, јер нам модел омогућава да размишљамо на вишем нивоу апстракције [AMBLER, 2003]. Методама за моделовање се дефинише и описује поступак и начин креирања модела. Поред методе, за опис конкретног система потребно је да постоји и одговарајући алата за опис система, односно одговарајући (адекватан) скуп концепата како би модел који креирамо одговарао систему који представља. Алат за моделовање система заправо представља језик за моделовање система који дефинише скуп концепата који се користе за опис система. Поред самог језика потребно је дефинисати и упутства и правила његовог коришћења. Према томе, једну методу за моделовање чини језик за моделовање, упутство за коришћење језика и поступак за моделовање.

МЕТАМОДЕЛ

Мета-модел представља модел модела. Однос између модела и метамодела, као и метамодела и мета-метамодела представља се везом *„у складу са“(conforms to).* Веза *„у складу са“* означава да је модел креиран *„у складу са“* својим метамоделом, слично као што је и метамодел креиран *„у складу са“* својим мета-метамоделом.

За формалну спецификацију језика за моделовање користе се метамодели. У складу са тиме постоји веза између ова три концепта приказана на слици (Слика 109) [KUHNE, T. (2006)], [SILVA, A.R. (2015)].

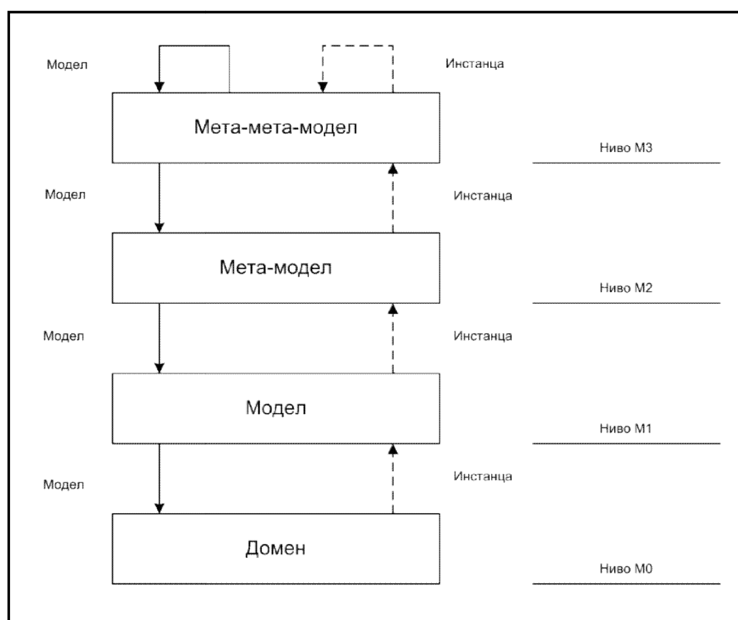


Слика 109. Однос између модела, метамодела и језика за моделовање

МЕТА-МЕТАМОДЕЛ

Метамодел се дефинише преко мета-метамодела, па концепти у метамоделу представљају инстанцу концепата са мета-метамодела. Мета-метамодел је комплетан, обиман и довољно изражајан *(expressive)* да опише себе (као на пример *Meta Object Facility (MOF)*).

OMG група је у оквиру MDA стандарда (*Model Driven Architecture*) дефинисала четири апстрактна нивоа на коме се могу наћи модели (познат као 4-нивојска архитектура модела). На сваком ниво се налазе модели који описују одговарајуће моделе са нижег нивоа, а у исто време представљају појављивања (инстанце) модела са вишег нивоа. На слици (Слика 110) дата је 4-нивојска архитектура модела.



Слика 110. Општа 4-нивојска архитектура модела

Ова 4-нивојска архитектура мета-модела уведена је од стране OMG групе и користи се за дефинисање фамилије OMG стандардизованих језика. На M0 слоју ове 4-нивојске архитектура налазе се објекти реалног света. У контексту UML-а овде се налазе на пример објекти класе Особа као што су на пример Јован, Димитрије, Софија, Андреј,..., итд. Објекти који се налазе на M0 слоју се описују односно моделују на слоју M1. M1 предсатвља модел модела који се налази на M0 слоју. У контексту UML-а на овом слоју се налазе концепти као што су Особа, Град, Место. Модели на M2 слоју представљају модел модела који се налазе на M1 слоју. На слоју M2 се налазе мета-модели за дефинисање OMG језика као што су *Unified Modeling Language* (UML), *Common Warehouse Metamodel* (CWM) и *XML Metadata Interchange* (XMI). Тако на пример концепт `Class`, представља апстракцију концепата (Osoba) који се налазе на M1 слоју. На врху ове 4-нивојске архитектуре (на слоју M3) налази се Meta-Object Facility (MOF) језик који се користи за спецификацију и конструкцију других OMG језика (на пример UML и MOF). MOF дефинише и оквир за управљање мета-подацима и скуп сервиса за потреба развоја алата за моделовање и подршку интероперабилности између њих [MOF, 2006].

ЈЕЗИК ЗА МОДЕЛОВАЊЕ

За креирање концепуалних модела користе се језици за моделовање. Ови језици имају прецизно дефинисану синтаксу (апстрактну и конкретну), семантику и прагматику. Језик за моделовање као и програмски језици представља вештачки *(artificial technical)* језик који се користи за спецификацију знања и информације о систему, на конзистентан и структурирани начин пратећи правила која су дефинисана преко мета-модела. Алати се користе за дефинисање и спецификацију мета-модела, па се због тога и процес дефинисања језика назива мета-моделовање. Изражајност и природа језика за моделовања зависи од начина како је дефинисан мета-модел. Језици за моделовањање на основу начина приказа модела односно на основу типа конкретне синтаксе језика се могу поделити на графичке и текстуалне. И поред тога што важи израз да једна слика може да представља хиљаду речи постоји мишљење да ови језици нису супериорни у односу на језике са текстуалном конкретном синтаксом [ENGELEN et al., (2010)]. За дефинисање језика за моделовање користе се одговарајући алати.

СИНТАКСА ЈЕЗИКА ЗА МОДЕЛОВАЊЕ

Синтаксом програмског језика дефинише се скуп дозвољених симбола и израза који се могу користити за креирање валидних програмских конструкција тог језика. Дефинисање синтаксе језика обухвата дефинисање конкретне и апстрактне синтаксе језика.

Апстрактна синтакса језика описује концепте језика, њихове везе, независно од начина њихове презентације. Са друге стране, конкретна синтакса обезбеђује презентацију концепата језика дефинисаних преко апстрактне синтаксе и омогућава њихово коришћење у креирању језичких израза. Дакле, синтакса се бави искључиво формом и структуром симбола у језику без било каквог размишљања о њиховом значењу. Семантика програмског језика дефинише право значење концепата који се у језику појављују.

Један од начина дефинисање програмских језика јесте коришћење контекстно-слободне граматике. Ова граматика се најчешће описује преко BNF (*Backus–Naur Form*) или EBNF (*Extended Backus–Naur Form*) форме. Како се BNF и EBNF користе за дефинисање других језика, односно њима се дефинише

(исказује) метамодел језика који се дефинише, онда BNF и EBNF можемо дефинисати као формални мета језик за представљање граматика других језика (програмских језика).

СЕМАНТИКА ЈЕЗИКА ЗА МОДЕЛОВАЊЕ

Дакле, апстракном синтаксом језика за моделовање се дефинишу концепти који се могу користити током моделовања. Конкретном синтаксом се дефинише презентација концепата који су дефинисани апстрактном синтаксом. Конкретна синтакса односно нотација је текстуална и графичка, али може бити и комбинована. Поред синтаксе језика, за сваки језик потребно је дефинисати и семантичка правила помоћу којих се дефинишу исправна коришћења сваког од концепата дефинисаних апстрактном синтаксом односно мета-моделом. Семантиком се дефинише скуп правила којим се одређује значење програма написаног у неком језику (на пример програмском језику или језику за моделовање) у недвосмисленом значењу [SLONNEGER, K. et al. (1995).

КЛАСИФИКАЦИЈА ЈЕЗИКА ЗА МОДЕЛОВАЊЕ

Језици за моделовање се могу класификовати на језике опште намене (*General Purpose Modeling Language*) и доменски специфичне језике (*Domain Specific (Modeling) Language*). Карактеристика језика за моделовање опште намене јесте да садрже велики број концепата и имају широку примену у моделовању различитих система и аспеката система (структуре, понашања). Најпознатији представници ове групе језика јесу UML и SysML. Са друге стране, доменски специфични језици су креиране за специфични домен од стране домен експерта и мањег су обима (имају мањи број концепата и језичких конструкција), па су стога лакши за разумевање и валидацију.

ЈЕЗИЦИ ЗА СПЕЦИФИКАЦИЈУ ЗАХТЕВА

Концептуални модели који се користе за спецификацију захтева се називају модели захтева *(requirements model).* Ови модели се креирају уз помоћ језика за спецификацију и моделовање захтева. Ови језици се међусобно разликују у погледу нотације, али и нивоу формализма. Шире посматрано, ови језици се могу поделити у следеће групе односно категорије: природни језици *(unconstrained natural language)*, контролисани природни језици *(controlled natural language)*, графички језици за моделовање *(graphical modeling language),* псеудокод *(pseudoceode)* и формални језици *(formal languages).*

ТРАНСФОРМАЦИЈЕ МОДЕЛА

У моделом вођеном развоју софтвера поред модела важну улогу имају и трансформације модела. Најчешће се под трансформацијом сматра превођење једног модела у други, односно на основу једног улазног модела се креира један излазни модел. Према [MENS et. al. (2005)] могуће је на основу више улазних модела трансформацијом добити један или више излазних модела. Стога трансформацијама се на основу улазног модела (један или више) креира излазни модел (један или више), а у складу са дефинисаним правилама трансформације. Трансформације које комбинују више улазних модела на основу којих се креира излазни модел се називају трансформације композиције модела *(model composition)* јер се њима не трансформише већ креира нови модел као композиција других модела [PASTOR, R. (2006)]. Правилима трансформације се дефинише како се концепт из улазног модела трансформише у концепт из излазног модела.

ТИПОВИ ТРАНСФОРМАЦИЈЕ МОДЕЛА

Трансформације се могу поделити на ендогене *(endogenous)* и егзогене *(exogenous).* Ендогене трансформације представљају трансформације код којих и улазни и излазни модел одговарају истом метамоделу. Егзогене трансформације представљају трансформације када улазни и излазни модел немају исти мета-модел. Поред тога, могу се разликовати *модел-у-модел (model2model)* и *модел-у-текст (model2text)* трансформације. Ова разлика се пре свега огледа у природи трансформације. *Модел-у-модел* тип трансформације користи концепт

пресликавања између мета-модела приликом трансформације, док се у *модел-у-текст* трансформацијама најчешће користе шаблони (темплејти). Већина *модел-у-текст* су егзогене трансформације. Осим ове класификације, трансформације можемо поделити и на хоризонталне и вертикалне. Хоризонталне трансформације су трансформације у којима се улазни и излазни модели налазе на истом нивоу. Са друге стране, вертикалне трансформације врше трансформацију модела са вишег нивоа апстракције на нижи ниво апстракције.

ДОДАТАК Б. ПРОЦЕС УТВРЂИВАЊА СОФТВЕРСКИХ ЗАХТЕВА

Према *Pohl*-у три су кључне активности у оквиру процеса утврђивања софтверских захтева [POHL, K., (2010)]:

1. **Активност откривања захтева** *(Elicitation)* има за циљ да идентификује све изворе захтева и да омогући да се током ове активности захтеви боље разумеју. Извори захтева могу бити различити, почев од различитих заинтересованих страна, преко различитих докумената, правилника и прописа па све до неких ранијих или актуелних софтверских ситема. Веома често пре почетка активности откривања захтева нису сви извори захтева унапред познати, па је утолико и већи значај ове активности.

2. **Активност документовања захтева** *(Documentation).* У центру ове активности јесте дефинисање и спецификација откривених захтева у складу са дефинисаним правилима. У оквиру ове активности може се уочити следећи скуп правила:

   a) *Општа правила документовања* су правила која се односе на све информације које се током процеса инжењеринга захтева документују. То су на на пример правила за дефинисање одговарајућег заглавља *(header),* али и за дефинисање основних информација које се односе на систем управљања документима као што су дефинисање аутора, историје верзије докумената и слично.

   b) *Посебна правила документовања* која се односе и примењују на документ захтева, *(requirements dokument)* која су специфична за сваку фазу процеса инжењеринга захтева. Овде се најчешће дефинишу правила која се односе на избор одговарајућег образца односно шаблона *(template)* који треба користити за документовање захтева. Основна намена ових правила јесте да се осигура и побољша квалитет дефинисања захтева за касније активности процеса инжењеринга захтева.

   c) *Правила спецификације* представљају правила која се односе на дефинисања документа спецификације захтева. Ова правила треба да осигурају високи квалитет специфицираних захтева јер

они треба да буду основа за касније фазе развоја софтвера. У ову групу на пример спадају правила која се односе на коришћење одговарајујућих синтаксних патерна *(syntatic requirements patterns)* или одговарајућег језика за спецификацију захтева *(requirements specification language)*

3. **Активност преговарања** *(Negotiation).* Имајући у виду да број заинтересованих страна који су укључени у процес инжењеринга захтева може бити различити, често се дешава да они имају различите погледе на систем који се развија. Неретко захтеви који се дефинишу од стране различитих учесника могу бити контрадикторни, па је циљ ове активности да се захтеви погледају из различитих углова и открију и разреше контрадикторности уколико оне постоје.

Поред ова три кључне активности *Pohl* истиче и две активности која се одвијају ортогонално и које значајно утичу на квалитет процеса инжењеринга захтева:

1. Валидација *(Validation)* према *Pohl*-у подразумева: а) валидацију докумената захтева *(requirements artifacts),* б) валидацију кључних активности и ц) валидацију контекста система *(validation of consideration of the system context).*

2. Управљање *(Management)* према *Pohl*-у подразумева: а) управљање документима дефинисаних захтева, б) управљање кључним активностима и ц) посматрање контекста система који подразумева праћење промена које настају а које су важне и утичу на систем.

Somerville дефинише четири кључне активности процеса утврђивања захтева [SOMERVILLE, I., (2010)]:

1. **Израда студије изводљивости.** Пословни системи се састоји од скупа пословних процеса. Потреба за управљањем овим пословним процесима и њихова аутоматизација захтева увођење софтверских система. Дакле, нови захтеви за увођењем софтверских система најчешће долазе услед потребе да се одређени процеси унутар пословних система аутоматизују и учине ефикаснијим. Стога, пре него што се крене у реализацију новог софтверског система потребно је дати опис ових пословних процеса, опис

захтева на које систем треба да одговори, као и опис начина на који софтверски систем треба да подржи пословне процесе. Управо опис пословних процеса и система који се жели увести представља улаз за израду студије изводљивости. Студија изводљивости треба да одговори на питања као што су: а) да ли нови систем може да се интегрише са постојећим системом (уколико већ постоји софтверски систем), б) да ли систем може да се реализује имајући у виду ограничења која су дефинисана у виду цене коштања и времена које је предвиђено за увођење овог софтверског система. Ова студија као резултат обезбеђује извештај који садржи препоруке (било позитивне или негативне) у вези са реализацијом и увођењем софтверског система.

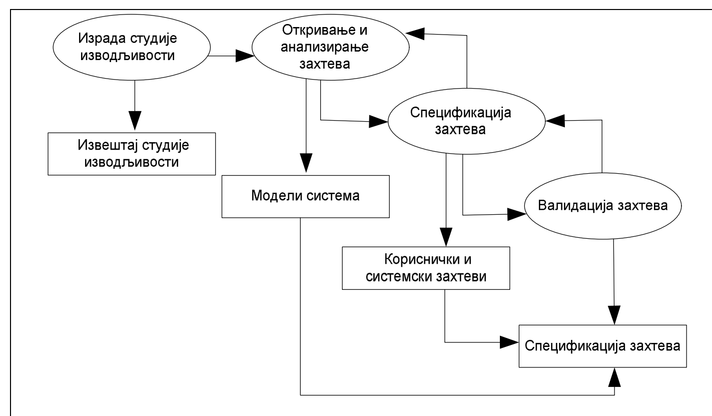
2. **Откривање и анализа захтева.** У оквиру активности откривања и анализе захтева врши се идентификација и анализа захтева. Како би се захтеви идентификовали потребно је у оквиру ове активности укључити све учеснике који су на директни или индиректни начин повезани са пословним процесима за које се дефинишу захтеви. У оквиру ове активности могу се уочити четири подактивности: откривање захтева, класификација и организација захтева, дефинисање приоритета у откриваним захтевима и документовање захтева. Ову активност *Sommerville* приказује као итеративни процес у коме се наведене подактивности извршавају циклично. Свака итерација почиње откривањем захтева и завршава се са документовањем захтева.

3. **Спецификација захтева.** Откривене захтеве потребно је на конзистентан и прегледан начин документовати. Системски, функционални захтеви се обично документују преко случајева коришћења или преко *табеле догађаја и одговора (event-response table)* [WIEGERS, K. (2003)]. Спецификацијом функционалних захтева јасно се дефинише домен модела (атрибути и релације између концепата који чине домен) али и дефинишу операције на које систем треба да одговори. Комплетна спецификација захтева обухвата спецификацију како функционалних тако и не-функционалних захтева. Комплетна спецификација захтева која садржи опис свих софтверских захтева које треба реализовати документује се у посебном документу спецификације захетва *(Software Requirements*

Specification). Данас постоји много различитих шаблона који дефинишу изглед овог документа. Један од шаблона дефинисан је као стандард IEEE/ANSI 830-1998. Шаблоне и стандарде за спецификацију софтверских захтева треба схватити више као препоруку и водич за спецификацију захтева, пре него као правило, јер информације које треба да садржи овај документ највише зависе од карактеристика самог система који се посматра [SOMMERVILLE, I. (2015)].

4. **Валидација захтева.** У оквиру ове активности потребно је утврдити да ли дефинисани захтеви одговарају захтевима наручиоца софтверског система. Најчешће коришћене технике за валидацију захтева су: израда прототипова и тестирање захтева.

Претходно поменуте активности процеса инжењеринга захтева приказане су ниже на Слика 111 [SOMMERVILLE, I. (2015)].



Слика 111. Активности процеса утврђивања захтева према *Sommerville-у*

Иако су ове активности на слици приказане линеарно, оне се не извршавају у једној итерацији. У пракси се ове активности међусобно преклапају и спецификација захтева настаје као резултат итеративног проласка кроз ове активности.

ДОДАТАК В. СЛУЧАЈЕВИ КОРИШЋЕЊА: ИСТОРИЈАТ И ЗНАЧАЈНИ АУТОРИ

Ivar Jacobson је увео појам случајева коришћења у процесу развоја софтвера и дао значење и прву дефиницију овог термина. Један је од познате тројке - „три амигоса“ (*James Rumbaugh*, *Grady Booch* и *Ivar Jacobson*) који су креирали UML. Почетком 80-тих година прошлог века *Jacobson* је написао први рад који је оригинално назван на шведском *„Användningsfall“* што у преводу значи случај коришћења *(use case).* У овом раду *Jacobson* је на основу искуства са првог пројекта у коме је користио објектно-оријентисани приступ *(„AXE Project“[22])* истакао значај спецификације интеракције између корисника и система у види акција. Први пут идеја о случајевима коришћена је јавно објављена у раду „*Object-oriented development in an industrial environment“* [JACOBSON, I. (1987)]. Нешто касније *Jacobson* је са групом аутора објавио књигу *"Object-Oriented Software Engineering: A Use Case Driven Approach"* [JACOBSON, I. et al. (1992)] у којој је детаљно објашњена улога случајева коришћења у процесу развоја софтвера. Неколико година касније са групом аутора објавио је књигу под називом *“The Unified Software Development Process - The Complete Guide to the Unified Process from the Original Designers”* [JACOBSON, I. et al. (1999)] у којој је детаљно објашњена улога коју случајеви коришћења имају у оквиру методе *Јединственог процеса развоја софтвера*.

Значајан допринос у примени случајева коришћења и промовисању улоге коју они имају у процесу развоја софтвера дали су следећи аутори:

- *Rebecca Wirfs-Brock* у свом раду „*The Art of Designing Meaningful Conversations*“ [REBECCA, J.W. (1999)] уводи и предлаже коришћење оквира за спецификацију случајева коришћења и објашњава како се случајеви коришћења, интеракција систем/корисник и објектно-оријентисано пројектовање уклапају заједно. Она је била инспирана књигом *„Software Requirements Objects, Functions and States“* [DAVIS, A. (1993)] аутора *Davis-a* који у књизи објашњава дилему познату као „шта према како“ *(what versus how)*. Према њему је тврдња, да су захтеви "шта" а не "како", много

[22] У оквиру овог пројекта *Jacobsson* је креирао нови тип дијаграма који је назвао дијаграм секвенци *(sequence diagram),* а који је коришћен за приказ интеракција између објеката.

поједностављена и није довољна да би смо развијали система онакав какав желимо.

- *Kurt Bittner* и *Ian Spence* са књигом „*Use Case Modeling*" [K. BITTNER, K. & SPENCE, I (2004)] су у великој мери утицали на подизање популарности ове технике за спецификацију захтева. На основу чињенице да се до тада класичним, конвенционалним начином дефинисања захтева декларативно (*„shall“* изразима) не може описати динамика система, они истичу да се преко случајева коришћења може дефинисати понашање система које ће бити разумљиво свим учесницима у пројекту.

- *Alistair Cockburn* као специјани саветник у Централној Банци Норвешке, крајем 90-тих, објавио је у часопису *„Journal of Object-Oriented Programming“* два рада: „*Goals and Use Cases“* [COCKBURN, A. (1997/1)] и *„Using Goal-Based Use Cases“* [COCKBURN, A. (1997/2)] у коме предлаже да акције случаја коришћења треба да буду усмерене ка циљу који треба остварити извршењем случаја коришћења. Ови радови су послужили многима за увођење технике случајева коришћења у многе компаније. Посебно су значајне и његове књиге *„Writing Effective Use Cases“* [COCKBURN, A. (2000)] и *„Patterns for Effective Use Cases“* [COCKBURN, A. et al. (2002)].

- *Gunnar Overgaard* један од коаутора књиге *„Object-oriented software engineering - a use case driven approach“*, поред радова по којима је познат по нешто формалнијем приступ у спецификацији случајева коришћења [G. OVERGAARD & K. PALMKVIST (1998)], [G. OVERGAARD (1999)], [G.OVERGAARD (2000)], познат је по књизи *„Use Cases: Patterns and Blueprints“* [OVERGAARD, G. & PALMKVIST, K. (2004)] у којој предлеже препоруке које се односе на стил писања случаја коришћења. Поред тога он дефинише и патерне за спецификацију случаја коришћења, али и примере добре праксе за организовање и поновно коришћење случајева коришћења.

- *Steve Adolph* са групом аутора у књизи *“Patterns for Effective Use Cases (The Agile Software Development Series)“* [COCKBURN, A. et al. (2002)] дефинише каталог од 31 патерна класификованих у две групе: 1) патерне структуре и 2) патерне процеса.

- Према *Martin Fowler-у* случајеви коришћења представљају технику за организовање и откривање захтева и као такви најбољи су уколико су кратки и читљиви. И поред тога што активно користи случајеве коришћења, он сматра да је тешко да одреди најбоље начине за дефинисање и коришћење случајева коришћења [FOWLER, M. & SCOTT, K. (1997)]. *Fowler* заступа тврдњу да *"непостоји стандардан начин за писање контекста случаја коришћења и да треба користити различите формате у различитим случајевима"* [M. FOWLER & K. SCOTT (1997)].

ДОДАТАК Г. ШАБЛОНИ ЗА СПЕЦИФИКАЦИЈУ СЛУЧАЈЕВА КОРИШЋЕЊА

COCKBURN- ОВ ШАБЛОН ЗА СПЕЦИФИКАЦИЈУ СЛУЧАЈА КОРИШЋЕЊА

Према *Cockburn-у* спецификацију случаја коршћења не треба компликовати чињеницама које се односе на детаљну интеракцију корисника са системом, пословним правилима, детаљима везаним за кориснички интерфејс и слично. Према *Cockburn-у* случај коришћења представља само облик односно форму писања која се може користити у различитим ситуацијама. С обзиром на различитост ситуација у којима се могу користити, примењују се различити стилови и шаблони (темплејти) за документовање случајева коришћења. Имајући у виду управо ове различите ситуације за који се случајеви коришћења могу користити, *Cockburn* је дефинисао различите типове случајева коришћења, који се међусобно разликују према карактеристикама које је назвао димензије *(dimensions)*.

Према *Cockburn-у* стил који треба пратити приликом документовања случаја коришћења јесу реченице које прате следећу форму: "*У одређеном времену z, актор y ради x са актором w са подацима v*". Глагол треба користити за означавање неке активности, па акције у којима се приказује интеракције треба писати у форми: *"Актор1 глагол актор2 са одређеним подацима."*

Основне карактеристике *Cockburn* стила су: 1) случај коришћења се документује у форми приче у једној колони 2) акције сценарија случаја коришћења су нумерисане, 3) не користе се условне реченице (не користе се IF-THEN искази), 4) алтернативна сценарија се такођу нумеришу комбинацијом бројева и слова (на пример 2.а или 2.б).

Cockburn дели шаблоне за спецификацију случајева коришћења према нивоу детаља и степену формалности [COCKBURN, A. (2003)]. *Cockburn* стога први шаблон који дефинише је потпуно описни *(fully dressed)* шаблон за спецификацију случајева коришћења. У овом шаблону потребно је да сви елементи буду специфицирани: примарни и секундарни актори, други актори заинтересовани за случај коришћења, предуслови, постуслови, иницијатори случаја коришћења *(trigger)*, опсег случаја коришћења, ниво случаја коришћења, потребни услови, довољни услови, главни сценарио и проширења главног сценарија. Поред

дефинисања ових елемената *Cockburn* истиче да је могуће користити и мањи скуп елементата случајева коришћења у зависности од потребе. У табели (Табела 4) је дат приказ *Cockburn-ov* шаблона за спецификацију случаја коришћења у потпуно описном облику:

Табела 4. *Cockburn-ov* шаблон за опис случаја коришћења

Елемент	**Опис елемента**
Број (Number)	Број случаја коришћења
Име (Name)	Треба да означи циљ случаја коришћења у форми кратког глаголског израза у активу
Циљ у контексту случаја коришћења (Goal in Context)	Уколико постоји потреба дефинисати шири опис циља случаја коришћења
Опсег (Scope)	Дефинисати опсег система који се пројектује
Ниво (Level)	Могући нивои случаја коришћења: *summary, primary task, subfunction*
Предуслов (Precondition)	Опис стања система које мора бити испуњено пре покретања случаја коришћења
Успешно извршење случаја коришћења (Success End Condition)	Опис стања система након успешног извршења случаја коришћења
Грешка у извршењу случаја коришћења (Failed End Condition)	Опис стања система након завршетка случаја коришћења који је проузроковала нека грешка у извршењу случаја коришћења
Окидач (Trigger)	Акција која окида извршење случаја коришћења
Основни сценарио (Main scenario)	Опис акција главног сценарија <корак #> <опис акције>
Алтернативни сценарио	Опис акција које се дешавају у случају да се

Елемент	**Опис елемента**
(Extension)	одређена акција главног сценарија не изврши успешно
Варијације (Sub-Variation)	Варијације на одређену акцију главног сценарија
Приоритет (Priority)	Дефинисање приорите који овај случај коришћења има за систем односно ораганизацију
Перформансе (Performance Target)	Очекивано време за извршење случаја коришћења
Учесталост (Frequency)	Дефинисање учесталости извршења случаја коришћења
Случајеви коришћења који укључују овај случај (Superordinate Use Case)	Називи случајева коришћења
Случајеви коришћења који користе овај случај коришћења (Subordinate Use Case)	Називи случајева коришћења
Остали корисници (Secondary Actors)	Листа корисника система
Отворена питања (Open Issues)	Опционо, листа питања на које овај случај коришћења „чека одговор“
Датум (Due Date)	Датум

Поред овог шаблона, *Cockburn* истиче и неформални *(casual)* шаблон. Основни елементи овог шаблона су наслов, примарни актор, опсег случаја коришћења, ниво односно степен сложености функционалности која је описана случајем коришћења и вербалним описом тока извршења сценарија случаја

коришћења. *Cockburn* сматра да уколико постоји потреба да се дефинише формалан модел за спецификацију случаја коришћења треба поћи од OCCAM језика кога је дефинисао *Tony Hoare*.

RUP ШАБЛОН ЗА СПЕЦИФИКАЦИЈУ СЛУЧАЈА КОРИШЋЕЊА

У табели (Табела 5) је дат приказ RUP шаблона за спецификацију случаја коришћења са кратким описом основних елемената који чине овај шаблон.

Табела 5. RUP-ов шаблон за опис случаја коришћења

Елемент	**Опис елемента**
Име (Name)	Кратко име случаја коришћења
Кратак опис (Brief Description)	Кратак опис случаја коришћења који треба да садржи опис и намене случаја коришћења
Ток догађаја (Flow of Events)	Текстуални опис интеракција између корисника и система. Овај опис треба да буде разумљив свим заинтересованим странама.
Посебни захтеви (Special Requirements)	Текстуални опис који садржи не-функционалне захтеве за текући случај коришћења, а који су битни за даљи развој и имплементацију софтверског система.
Предуслов (Precondition)	Опис предуслова који треба бити испуњен пре него што почне извршење случаја коришћења
Постуслов (Postcondition)	Опис стања система након извршења случаја коришћења
Тачка проширења (Extension point)	Дефинисање места у случају коришћења (тачка проширења) на којима извршење неког другог случаја коришћења проширује извршење текућег случаја коришћења

БИОГРАФИЈА АУТОРА

Душан Савић је рођен 02.10.1979. године у Александровцу Жупском. У Александровцу је завршио основну школу и гимназију природно-математичког смера. Учествовао је на многим такмичењима, а од резултата се издваја 2. место на Републичком такмичењу из физике за ученике седмог разреда основних школа у Србији. Године 1998. уписује *Факултет организационих наука*, *Универзитет у Београду*, на коме је 2004. године и дипломирао. На ФОН-у 2004.године уписује последипломске студије где је 2010. године успешно одбранио магистарски рад под насловом „***Спецификација случаја коришћења преко SilabReq извршивог језика***" у области софтверског инжењерства, под менторством др Синише Влајића, доцента ФОН-а.

Од 2002. године ангажован је на извођењу лабораторијских вежби на ФОН-у на предметима *Пројектовање програма* и *Принципи програмирања*, а од 2005. године и запослен као асистент приправник на ***Катедри за Софтверско инжењерство*** и ангажован на предметима на основним студијама: *Пројектовање софтвера*, *Софтверски патерни*, *Софтверски процес и еволуција софтвера*, *Конструкција софтвера* и *Тестирање софтвера*. Такође, ангажован је и на предметима мастер академских студија студијског програма *Софтверско инжењерство и рачунарске науке*.

Као члан *Катедре за софтверско инжењерство* учествовао је у извођењу значајних пројеката међу којима су:

- *Примена рачунарске технике у експерименталној физици чврстог стања* - Министарство за науку и технолошки развој, број 174031, Београд, 2011-2016.
- *Пројекат KOSTMOD* - Министарства одбране Краљевине Норвешке и Министарства одбране Републике Србије, Београд, 2006-2010.
- *Идејни пројекат информационог система е-аукцијске јавне набавке* - Министарства за телекомуникације и информатичко друштво Републике Србије, Београд, 2007-2008.
- *Модернизација информационог система за здравствене установе*, IQ-net, Београд, 2006.

Поред тога учествовао је и у пројектима:

- *Менаџмент јавних набавки*, Републички фонд за здравствено осигурање, Београд, 2014.
- *Фото модул, вести и архива*, Политика новине и магазини, Београд, 2014.
- *РФЗО апотеке*, Републички фонд за здравствено осигурање, Београд, 2013.
- *Развој информационог подсистема за кадровску евиденцију*, ДЗ Вождовац, Београд, 2007-2009.
- *Развој апликације за вођење књиге улазних фактура*, ДЗ Вождовац, Београд, 2007-2008.
- *Развој здравственог информационог система за колоректални скрининг*, Европска агенција за реконструкцију (ЕПОС), Београд, 2006-2007.
- *Развој информационог система за новинску агенцију Бета*, Београд 2005.
- *Развој дела пословног информационог система фирме „Перихард инжењеринг“*, Београд 2004-2006

Од 2013. до 2015. године био је ангажован као консултант *Републичког фонда за здравствено осигурање.*

Октобра 2010. године одржао је гостујуће предавање на Математичком факултету под насловом *„Extended Software Architecture based on Security Patterns“.*

Ожењен је и са супругом *Марином* има троје деце: ћерку *Софију* (9 година) и два сина: *Јована* (7 година) и *Андреја* (18 месеци).

Изјава о ауторству

Име и презиме аутора Душан Савић

Број индекса 522/2012

Изјављујем

да је докторска дисертација под насловом

РАЗВОЈ СОФТВЕРА ЗАСНОВАН НА МОДЕЛУ СЛУЧАЈЕВА КОРИШЋЕЊА И MDD ПРИСТУПУ

- резултат сопственог истраживачког рада;
- да дисертација у целини ни у деловима није била предложена за стицање друге дипломе према студијским програмима других високошколских установа;
- да су резултати коректно наведени и
- да нисам кршио ауторска права и користио интелектуалну својину других лица.

Потпис аутора

У Београду, 15.07.2016.

____

Изјава о истоветности штампане и електронске верзије докторског рада

Име и презиме аутора Душан Савић

Број индекса 522/2012

Студијски програм Информациони системи

Наслов рада РАЗВОЈ СОФТВЕРА ЗАСНОВАН НА МОДЕЛУ СЛУЧАЈЕВА КОРИШЋЕЊА И MDD ПРИСТУПУ

Ментор проф. др Владан Девеџић

Изјављујем да је штампана верзија мог докторског рада истоветна електронској верзији коју сам предао ради похрањена у **Дигиталном репозиторијуму Универзитета у Београду.**

Дозвољавам да се објаве моји лични подаци везани за добијање академског назива доктора наука, као што су име и презиме, година и место рођења и датум одбране рада.

Ови лични подаци могу се објавити на мрежним страницама дигиталне библиотеке, у електронском каталогу и у публикацијама Универзитета у Београду.

Потпис аутора

У Београду, 15.07.2016.

____

Изјава о коришћењу

Овлашћујем Универзитетску библиотеку „Светозар Марковић“ да у Дигитални репозиторијум Универзитета у Београду унесе моју докторску дисертацију под насловом:

РАЗВОЈ СОФТВЕРА ЗАСНОВАН НА МОДЕЛУ
СЛУЧАЈЕВА КОРИШЋЕЊА И MDD ПРИСТУПУ

која је моје ауторско дело.

Дисертацију са свим прилозима предао сам у електронском формату погодном за трајно архивирање.

Моју докторску дисертацију похрањену у Дигиталном репозиторијуму Универзитета у Београду и доступну у отвореном приступу могу да користе сви који поштују одредбе садржане у одабраном типу лиценце Креативне заједнице (Creative Commons) за коју сам се одлучио.

1. Ауторство (CC BY)

2. Ауторство – некомерцијално (CC BY-NC)

3. Ауторство – некомерцијално – без прерада (CC BY-NC-ND)

4. Ауторство – некомерцијално – делити под истим условима (CC BY-NC-SA)

5. Ауторство – без прерада (CC BY-ND)

6. Ауторство – делити под истим условима (CC BY-SA)

(Молимо да заокружите само једну од шест понуђених лиценци.

Кратак опис лиценци је саставни део ове изјаве).

Потпис аутора

У Београду, 15.07.2016.