Универзитет у Београду
Факултет организационих наука

Милош Ж. Милић

Побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера

Докторска дисертација

Београд, 2017.

University of Belgrade
Faculty of Organizational Sciences

Miloš Ž. Milić

Improvement of object-oriented software systems by applying software quality standards

Doctoral Dissertation

Belgrade, 2017.

Ментор:

Др Синиша Влајић, ванредни професор
Универзитет у Београду
Факултет организационих наука

_____

Чланови комисије:

Др Саша Лазаревић, ванредни професор
Универзитет у Београду
Факултет организационих наука

_____

Др Драган Бојић, ванредни професор
Универзитет у Београду
Електротехнички факултет

_____

Датум одбране: _____

Захвалница

Породица је основа свега: у породици живот започиње, у породици се живот одвија, због породице живот има смисла. У том смислу бих желео да се захвалим родитељима, Вери и Живојину, и брату Марку за сву љубав, усмеравања и подршку коју су ми пружили у животу и школовању. Такође бих желео да се захвалим комплетној широј породици за сву помоћ и подршку у току досадашњег школовања које, уколико се у обзир узму и докторске студије, траје преко двадесет година. Сваки мој успех је уједно и ваш успех и хвала вам на томе.

Велику захвалност дугујем ментору, професору Синиши Влајићу, за све савете које ми је давао од почетка студија до данашњих дана. Ова докторска дисертација представља резултат дугогодишњег рада и сигуран сам да се кроз мој рад виде његова усмеравања, савети и критике који су допринели да докторска дисертација има овакав облик. Наша сарадња и поштовање траје преко десет година и надам се да ћемо успешно сарађивати и у будућности.

Велику захвалност дугујем колегама Илији Антовићу, Душану Савићу, Војиславу Станојевићу и Саши Лазаревићу за заједничке године рада и савете које су ми дали, посебно у времену писања докторске дисертације. Заједно смо учествовали на великом броју пројеката у оквиру којих смо се сусретали са различитим инжењерским проблемима и решењима што ми је помогло да проширим границе свога знања. Надам се да ћемо, као тим, у будућности сарађивати на великом броју научно-стручних пројеката.

Хтео бих да се захвалим професорки Јелени Јовановић која ми је помогла у процесу припреме рада за објављивање у часопису. Њене критике и савети су били драгоцени и у докторску дисертацију су уграђени резултати које ми је она помогла да обликујем. Јелена је један од најбољих истраживача кога сам имао прилике да упознам и надам се да ћемо сарађивати и у будућности.

На предлог ментора својевремено сам се заинтересовао за квалитет софтвера, стандарде квалитета софтвера, атрибуте квалитета софтвера, софтверске метрике и алате за статичку анализу квалитета софтвера. Стога бих хтео да се захвалим професорки Драгани Бечејски-Вујаклија која ми је за потребе израде мастер рада обезбедила приступ до ISO/IEC стандарда из области софтверског инжењерства. Посебно јој се захваљујем што ми је обезбедила приступ до стандарда серије ISO/IEC 25000 непосредно након његовог изласка.

Велику захвалност дугујем уваженом колеги Радиславу Чарапићу, софтверском инжењеру са преко петнаест година искуства у индустрији развоја софтвера. Његова експертиза обухвата велики број софтверских система различите сложености који се примењују у различитим индустријским гранама. У процесу израде докторске дисертације често смо имали прилику да размењујемо мишљења: том приликом сам добијао савете, критике и повратне информације које се односе на могућност примене научних резултата у индустрији што је за сваког истраживача веома значајно. Волео бих да и у будућности размењујемо мишљења и сарађујемо на занимљивим пројектима.

Хтео бих да се захвалим професорки Мирјани Чоловић-Меснер која ми стално помаже око припреме радова. Њено искуство и савети су заиста драгоцени, увек је лепо и корисно сарађивати са њом.

Велику захвалност дугујем студентима треће и четврте године Факултета организационих наука који су учествовали у процесу реализације студије случаја. Захваљујући њима дошао сам до веома значајних опажања и резултата који се односе на примену стандарда квалитета софтвера у процесу развоја објектно-оријентисаних софтверских система. Студенти су доборовољно и анонимно учествовали у анкети и/или студији случаја за време колоквијума и испитних рокова због чега им се посебно захваљујем.

У Београду, 15.09.2017. год.

Резиме

Предмет истраживања докторске дисертације је могућност побољшања објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера.

У истраживању је дат преглед различитих стандарда квалитета софтвера из области софтверског инжењерства, софтверских метрика и алата за статичку анализу квалитета софтвера који се оперативно користе у процесу евалуације квалитета софтвера.

У докторској дисертацији су идентификовани механизми побољшања објектно-оријентисаних софтверских система (општи принципи пројектовања софтвера, принципи објектно-оријентисаног пројектовања софтвера, стратегије пројектовања софтвера, патерни пројектовања софтвера и методе развоја софтвера) и успостављена је њихова веза са стандардима квалитета софтвера.

У раду је развијена оригинална SilabQOSS (енг. Silab Quality Method for Object-oriented Software Systems) метода за побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера. Предложена метода користи стандарде квалитета софтвера, софтверске метрике и алате за статичку анализу квалитета софтвера. Такође, метода користи претходно поменуте механизме за побољшање објектно-оријентисаних софтверских система. Посматрана метода је подржана софтверским алатом SilabMetrics који се може користити самостално или се интегрисати са NetBeans окружењем за развој софтвера. На основу извршене евалуације закључено је да се применом SilabQOSS методе и SilabMetrics алата за статичку анализу квалитета софтвера омогућава развој софтверских система који су стабилнији, једноставнији за развој, одржавање и даљу надоградњу.

Кључне речи: објектно-оријентисани системи, стандарди квалитета софтвера, модели квалитета софтвера, софтверске метрике, механизми побољшања објектно-оријентисаних софтверских система

Научна област: Рачунарске науке

Ужа научна област: Софтверско инжењерство

УДК број: 004.4

Abstract

The research subject of the doctoral dissertation is the possibility of improvement of object-oriented software systems by applying software quality standards.

The research provides an overview of different software engineering quality standards, software metrics, and tools for static analysis of software quality which are operatively used in the software quality evaluation process.

The doctoral dissertation identifies different mechanisms for improving object-oriented software systems (i.e. general principles of software design, principles of object-oriented software design, software design strategies, design patterns, and software development methods) as well as their relation with the software quality standards.

The doctoral dissertation introduces original SilabQOSS (Silab Quality Method for Object-oriented Software Systems) method for improving object-oriented software systems using software quality standards. The proposed method incorporates software quality standards, software metrics, and tools for static analysis of software quality. In addition, the method uses mechanisms for improving object-oriented software systems, as already mentioned. The method is supported by the SilabMetrics software tool that can be used independently or integrated with the NetBeans software development environment. The evaluation results confirm that the SilabQOSS method and the SilabMetrics software quality tool enable the development of software systems which are more stable, easier to develop, maintain, and upgrade.

Keywords: object-oriented software systems, software quality standards, software quality models, software metrics, mechanisms for improving object-oriented software systems

Scientific field: Computer Science

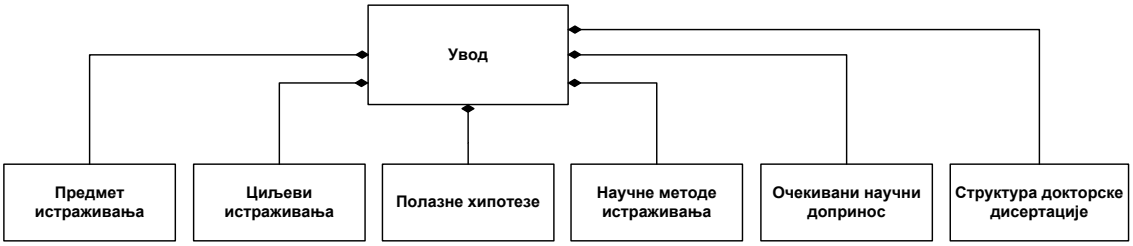
Scientific subfield: Software Engineering

UDK number: 004.4

Садржај

1. Увод

У уводном поглављу даје се опис предмета истраживања и циљева истраживања докторске дисертације. Такође, у оквиру уводног поглавља дефинишу се полазне хипотезе, научне методе истраживања и очекивани научни допринос. На крају поглавља даје се структура докторске дисертације по поглављима. Концептуални приказ поглавља дат је на наредној слици (Слика 1).



Слика 1. Концептуални приказ уводног поглавља

1.1. Предмет истраживања

Предмет истраживања докторске дисертације је развој SilabQOSS (енг. Silab Qualty Method for Object Oriented Systems) методе за побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера.

Софтверски системи су нашли широку примену у данашњем пословном окружењу. У претходним деценијама је порасла потреба за квалитетом софтвера. Стога се може рећи да осигурање квалитета софтверског производа и процеса његовог развоја представља неминовност и потребу сваке софтверске компаније. Мерењем различитих карактеристика посматраног софтверског производа и процеса његовог развоја могу се предузети акције које би требале да доведу до повећања нивоа квалитета и поузданости софтвера. Да би се осигурао одговарајући квалитет потребно је извршити правилну спецификацију и евалуацију квалитета софтвера. Ово се може остварити дефинисањем одговарајућих атрибута квалитета софтвера [Bansiya02], узимајући у обзир сврху примене софтвера. У том смислу се

атрибути квалитета могу користити за дефинисање нивоа квалитета софтверског производа, али и за дефинисање квалитета процеса развоја софтвера. Због тога ће у докторској дисертацији бити дат преглед стандарда квалитета софтвера и софтверских метрика које се могу применити у процесу развоја и побољшања објектно-оријентисаних софтверских система. У процесу израде докторске дисертације ће бити развијен сопствени алат који је заснован на стандардима квалитета софтвера и интегрисан са NetBeans развојним окружењем.

Евалуацију квалитета софтвера могуће је извршити применом стандарда квалитета софтвера које дефинишу Међународна организација за стандардизацију и Међународна електротехничка комисија. Стандарди квалитета ISO/IEC 25000 [ISO25000], ISO/IEC 14598 [ISO14598] и ISO/IEC 9126 [ISO9126] су посебно значајни за софтверско инжењерство. Стандард квалитета ISO/IEC 14598 дефинише поступак за евалуацију квалитета софтверског производа при чему се као модел квалитета користи ISO/IEC 9126 стандард. Стандард квалитета ISO/IEC 9126 дефинише моделе квалитета софтвера, атрибуте квалитета софтвера, као и софтверске метрике за сваки атрибут квалитета [Schackmann09]. Ове метрике се користе за мерење квалитета софтверског система. Нова серија стандарда ISO/IEC 25000 треба да замени и прошири ISO/IEC 9126 и ISO/IEC 14598 стандарде [Bevan05].

У процесу развоја софтвера примењују се различите методе развоја софтвера (Јединствени процес развоја софтвера, Ларманова метода развоја софтвера, Екстремно програмирање итд.) које су засноване на различитим моделима развоја софтвера (Модел водопада, Итеративно-инкрементални модел, Спирални модел итд.), које користе различите стратегије развоја софтвера (засноване на случајевима коришћења, моделима, тестовима итд.). У фази пројектовања софтвера користе се општи и објектно-оријентисани принципи пројектовања софтвера, стратегије пројектовања софтвера и патерни пројектовања софтвера. У том смислу ће у докторској дисертацији бити дефинисане и објашњене методе развоја софтвера, општи и објектно-

оријентисани принципи пројектовања софтвера, патерни пројектовања софтвера, стратегије пројектовања софтвера и њихова веза са стандардима квалитета софтвера.

Имајући у виду све изнесено може се закључити да развој софтвера представља скуп сложених активности којима је потребно управљати на одговарајући начин [Offutt02]. У том смислу је потребно истражити:

- утицај стандарда, метода и алата квалитета софтвера на процес развоја објектно-оријентисаних софтверских система и
- везу метода развоја софтвера, општих и објектно-оријентисаних принципа пројектовања софтвера, патерна пројектовања софтвера и стратегија пројектовања софтвера са стандардима квалитета софтвера.

Докторска дисертација је усмерена на развој сопствене методе за побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера и алата за статичку анализу квалитета софтвера.

1.2. Циљеви истраживања

Циљеви докторске дисертације су вишеструки. Циљ је, најпре, да се истражи постојећа литература везана за стандарде квалитета софтвера и да се укаже на могућности примене стандарда квалитета софтвера и софтверских метрика у процесу развоја објектно-оријентисаних софтверских система. У том смислу ће бити дат преглед алата за статичку анализу квалитета софтвера и извршена њихова компаративна анализа.

Наредни циљ докторске дисертације је развој сопственог алата заснованог на стандардима квалитета софтвера и софтверским метрикама који се може применити у процесу развоја и побољшања објектно-оријентисаних софтверских система.

С обзиром на сложеност процеса развоја софтвера, циљ докторске дисертације је да укаже на методе развоја софтвера, опште и објектно-оријентисане принципе пројектовања софтвера, патерне пројектовања софтвера, стратегије пројектовања софтвера и њихову везу са стандардима квалитета софтвера у циљу побољшања објектно-оријентисаних софтверских система.

Све заједно би требало да резултира идентификовању и развоју нове методе за побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера, што је и примарни циљ докторске дисертације.

1.3. Полазне хипотезе

На основу анализе доступне литературе и постављеног предмета и циља истраживања може се поставити општа хипотеза као и посебне хипотезе докторске дисертације, које ће у раду бити потврђене или оповргнуте.

Општа хипотеза:

- Објектно-оријентисане софтверске системе могуће је унапредити коришћењем стандарда квалитета софтвера.

Посебне хипотезе:

- Стандарди квалитета софтвера могу се применити у процесу развоја и побољшања објектно-оријентисаних софтверских система.
- Софтверске метрике се могу повезати са општим и објектно-оријентисаним принципима пројектовања софтвера, патернима пројектовања софтвера и стратегијама и методама развоја софтвера.
- Могуће је развити сопствени алат за статичку анализу квалитета софтвера који се може применити у процесу развоја и побољшања објектно-оријентисаних софтверских система.

- Сопствени алат за статичку анализу квалитета софтвера могуће је уградити у NetBeans развојно окружење.
- Могуће је развити сопствену методу за побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера и алата за статичку анализу квалитета софтвера.

1.4. Научне методе истраживања

Да би се успешно реализовала идеја истраживања и потврдиле (или оповргле) постављене хипотезе, у раду ће бити коришћен основни метод истраживања који се базира на постојећим теоријским резултатима и експерименталном раду. Под тим се подразумева сакупљање и проучавање доступне литературе, њена анализа и систематизација, све са циљем да се покаже оправданост и корисност нове методе побољшања објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера.

У раду ће се користити следеће методе истраживања: анализа-синтеза, индукција-дедукција, конкретизација-генерализација, метода моделовања, компаративна метода, метода студије случаја, статистичка метода.

1.5. Очекивани научни допринос

Допринос који се жели остварити овом докторском дисертацијом се огледа у систематизацији постојећих знања које се односе на стандарде квалитета софтвера и развоју сопствене методе за побољшање објектно-оријентисаних софтверских система, коришћењем стандарда квалитета софтвера и сопственог алата за статичку анализу квалитета софтвера.

Сходно томе, као очекивани резултати овог рада могу се навести:

- Приказ стандарда квалитета софтвера који се могу се применити у процесу развоја и побољшања објектно-оријентисаних софтверских система.
- Преглед и класификација софтверских метрика.

- Преглед и анализа постојећих алата за статичку анализу квалитета софтвера.
- Дефинисање и објашњење општих и објектно-оријентисаних принципа пројектовања софтвера, патерна пројектовања софтвера и стратегија и метода развоја софтвера.
- Развој сопственог алата за статичку анализу квалитета софтвера који се може применити у процесу развоја и побољшања објектно-оријентисаних софтверских система.
- Повезивање сопственог алата за статичку анализу квалитета софтвера и NetBeans развојног окружења.
- Развој сопствене методе за побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера и алата за статичку анализу квалитета софтвера.

1.6. Структура докторске дисертације

Докторска дисертација састоји се од дванаест поглавља која су организована на следећи начин:

У уводном поглављу дефинисани су предмет, проблем и циљеви истраживања. На основу тога су дефинисане хипотезе истраживања и научне методе истраживања.

У другом поглављу су описани објектно-оријентисани софтверски системи. Најпре су приказани основни концепти објектно-оријентисаних софтверских система: класа и њене чланице, наслеђивање, полиморфизам, апстрактна класа и интерфејс. Затим се даје приказ објектно-оријентисаних технологија и окружења за развој објектно-оријентисаних софтверских система.

У трећем поглављу су описани стандарди квалитета софтвера. У том смислу је дат преглед различитих дефиниција квалитета софтвера, као и преглед ISO/IEC 9126, ISO/IEC 14598 и ISO/IEC 25000 стандарда квалитета софтвера

који се могу користити у процесу евалуације квалитета софтверских система.

У четвртом поглављу описане су софтверске метрике. Најпре је дат преглед карактеристика које софтверске метрике као квантитативни показатељи требају да задовоље, као и веза софтверских метрика са стандардима квалитета софтвера. У оквиру посебног одељка дат је преглед различитих класификација софтверских метрика. На крају је дат опис често коришћених софтверских метрика за модел пројектовања софтвера [Chidamber94]: *Циклична сложеност*, *Сложеност пондерисаних метода*, *Број пондерисаних метода класе*, *Број одговора класе*, *Недостатак кохезивности метода у класи*, *Повезаност објеката*, *Дубина стабла наслеђивања*, *Број подкласа*, *Број наредби у методи*, као и *Метрика стабилности софтвера* [Martin06][Vlajic17].

У петом поглављу дат је преглед и анализа постојећих алата за евалуацију квалитета софтвера. У оквиру поглавља разматра се веза алата за статичку анализу квалитета софтвера са стандардима квалитета софтвера. Затим се даје преглед Swat4J [Milic07], SonarQube [SonarQube], FindBugs [FindBugs] и SilabMetrics [Milic17] алата за статичку анализу квалитета софтвера који се оперативно користе у процесу евалуације квалитета софтверских система. Након прегледа алата дефинисани критеријуми на основу којих је извршена њихова компаративна анализа.

У шестом поглављу описани су механизми побољшања објектно-оријентисаних софтверских система [Martin96][Vlajic14]: општи принципи пројектовања софтвера, принципи објектно-оријентисаног пројектовања софтвера, стратегије пројектовања софтвера, патерни пројектовања софтвера и методе развоја софтвера.

У седмом поглављу је описана сопствена SilabQOSS метода за побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера. Предложена метода користи стандарде квалитета

софтвера, софтверске метрике и алате за статичку анализу квалитета софтвера. Такође, метода користи механизме за побољшање објектно-оријентисаних софтверских система. У оквиру поглавља дат је опис методе, као и преглед активности и артифаката методе.

У осмом поглављу извршена је евалуација дефинисане SilabQOSS методе за побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера. У процесу евалуације користи се метода студије случаја. У том смислу је спроведен експеримент са две групе студената како би се проверила могућност практичне примене дефинисане методе. Затим је извршена анализа резултата примене SilabQOSS методе на студију случаја. На крају поглавља дат је преглед ограничења извршене евалуације.

У деветом поглављу дата су закључна разматрања и дефинисани могући правци даљег истраживања.

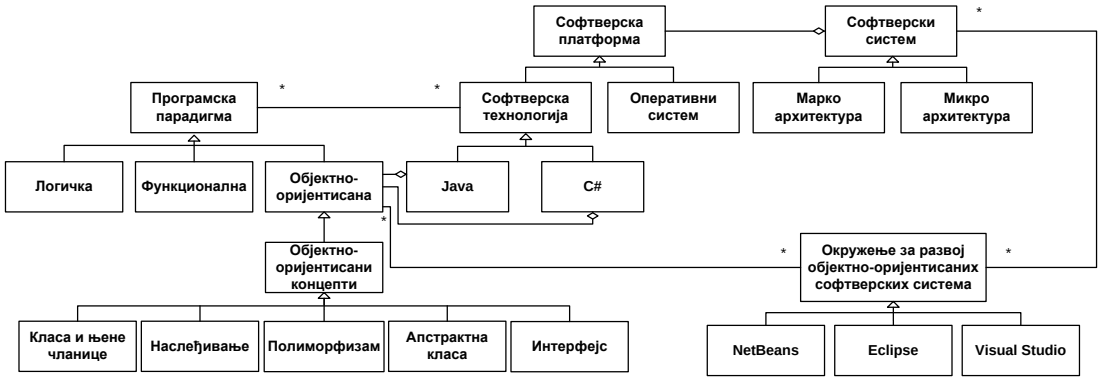
У десетом поглављу дат је преглед коришћене литературе у процесу израде докторске дисертације.

У једанаестом поглављу дат је попис слика у докторској дисертацији.

У дванаестом поглављу дат је попис табела у докторској дисертацији.

2. Објектно-оријентисани софтверски системи

У процесу развоја софтвера могуће је користити различите програмске парадигме (нпр. објектно-оријентисану, функционалну, логичку итд.). Објектно-оријентисана парадигма и објектно-оријентисани софтверски системи се често користе у процесу развоја софтвера и представљају индустријски стандард који подржавају највеће софтверске компаније. Због тога се у овом поглављу даје приказ објектно-оријентисаних софтверских система. Најпре ће бити објашњени основни објектно-оријентисани концепти који се користе у процесу развоја софтвера. Након тога ће бити приказане објектно-оријентисане софтверске технологије. На крају поглавља даје се приказ окружења за развој објектно-оријентисаних софтверских система. Концептуални приказ поглавља дат је на наредној слици (Слика 2).



Слика 2. Концептуални приказ објектно-оријентисаних софтверских система

2.1. Објектно-оријентисани концепти

У овом одељку даје се опис основних објектно-оријентисаних концепата: класе и њених чланица, наслеђивања, полиморфизма, компатибилности објектних типова, касног повезивања, апстрактних класа и интерфејса. Важно је напоменути да су посматрани објектно-оријентисани концепти независни од конкретне софтверске технологије, тј. имплементирани су у сваком језику који подржава објектно-оријентисану парадигму. Посматрани концепти представљају неопходну основу за развој генеричких компоненти

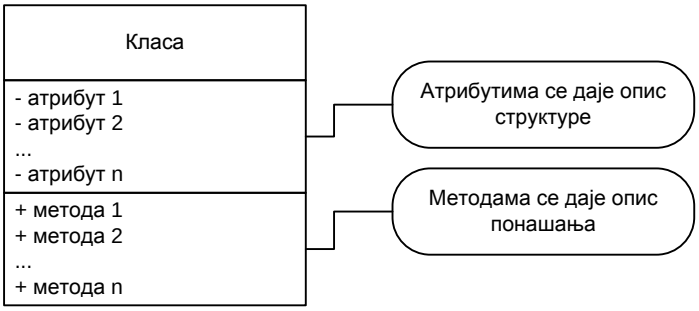
објектно-оријентисаних софтверских система чиме се омогућава развој поузданог софтвера [Meyer88].

2.1.1. Класа и њене чланице

Класа представља основни објектно-оријентисани концепт који се користи у процесу развоја софтвера. Постоје различите дефиниције класе:

- "Класа је објектно-оријентисани концепт који обухвата скуп података и процедура које обрађују податке." [Pressman10]
- "Класа представља програмски ентитет у објектно-оријентисаном програму који садржи комбинацију функција и података." [ISO24765]
- "Класа представља апстрактни тип података који је делимично или у потпуности имплементиран." [Meyer88]
- "Класа представља скуп објеката који деле заједничку структуру и заједничко понашање." [Booch06]
- "Класа представља апстрактну представу скупа објеката који имају исте особине." [Vlajic08]

Из наведених дефиниција се може закључити да се класом дефинише неки ентитет у оквиру кога се учаурују подаци (тј. структура) и понашање (тј. функције) о посматраном скупу објеката [Rumbaugh91]. Структура класе описује се атрибутима, док се понашање описује методама [Meyer88]. Атрибути и методе класе се једним именом називају чланице класе [Vlajic08]. Класа се може представити коришћењем UML дијаграма класа, што је и приказано на наредној слици (Слика 3).

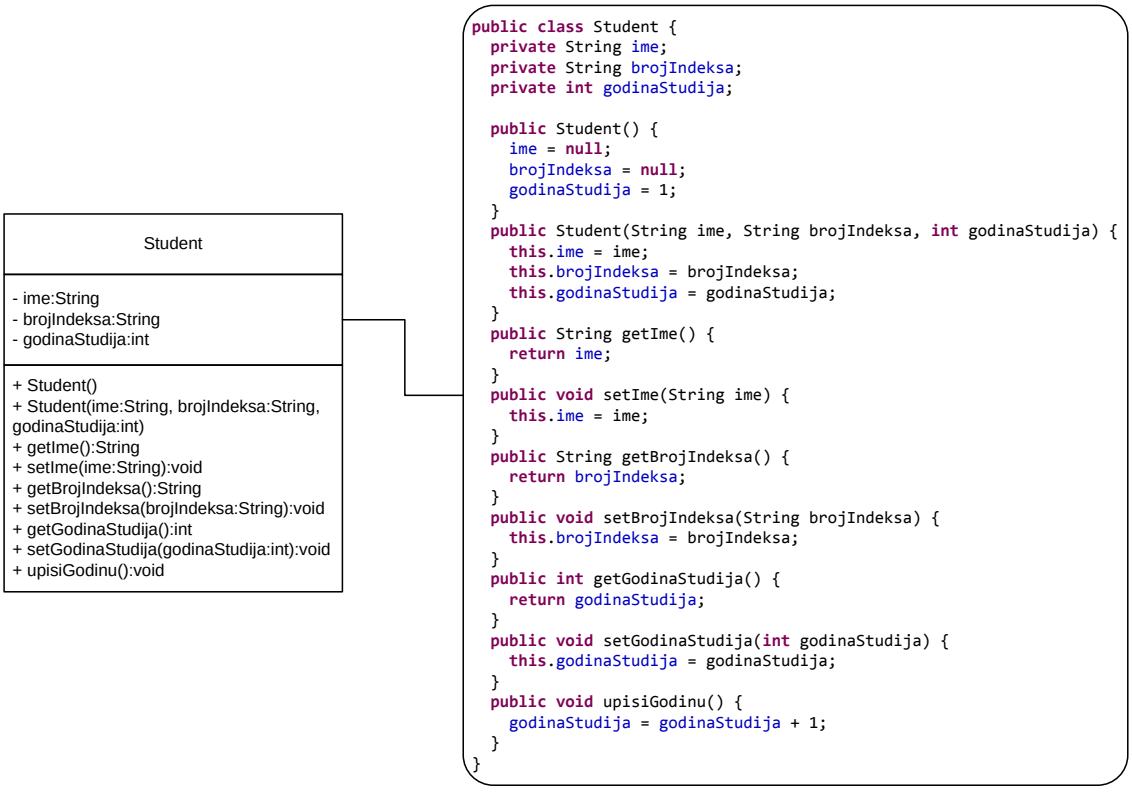


Слика 3. Концептуални приказ класе коришћењем UML дијаграма класа

С друге стране, објекат представља примерак (инстанцу), тј. једно конкретно појављивање посматране класе.

Размотримо пример који је приказан на наредној слици (Слика 4). Са слике се уочава класа Student која садржи атрибут *ime* које је типа String, атрибут *brojIndeksa* који је типа String и атрибут *godinaStudija* који је типа int.

Посматрана класа садржи два конструктора: *Student()* и *Student(String ime, String brojIndeksa, int godinaStudija)*. Конструктор представља методу која се зове исто као класа и служи за иницијализацију објеката класе. Први приказани конструктор назива се непараметарски конструктор и његова је карактеристика да свим објектима класе, приликом иницијализације, додељује исте почетне вредности. Други приказани конструктор назива се параметарски конструктор и његова је карактеристика да приликом иницијализације објектима класе додељује почетне вредности у зависности од параметара.



Слика 4. Приказ класе Student која садржи атрибуте и методе

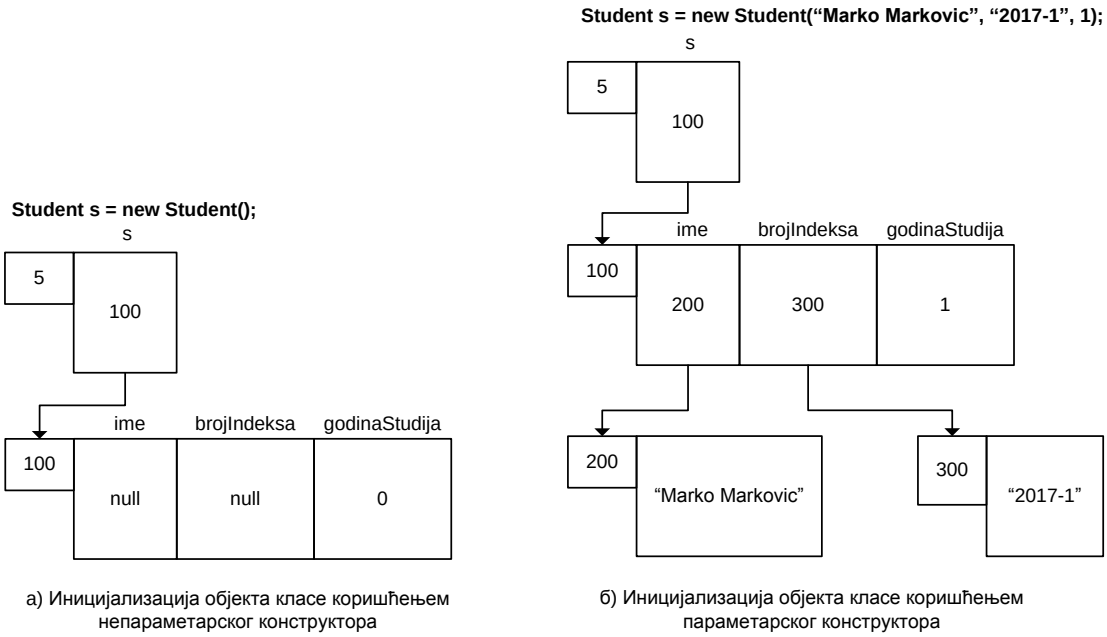
Такође, из примера се уочава да класа садржи методе *getIme()*, *getBrojIndeksa()* и *getGodinaStudija()* које као повратну вредност враћају име, број индекса и годину студија студента, респективно. Поред тога, уочавају се и методе *setIme()*, set*BrojIndeksa()* и set*GodinaStudija()* које постављају вредност за име, број индекса и годину студија студента у зависности од прослеђеног параметра. Посматрана класа такође садржи методу *upisiGodinu()* која повећава годину студија студента. Као што се примећује, метода класе састоји се од следећих целина:

- Назив методе - на основу назива се врши позив посматране методе.
- Листа параметара (тј. аргумената) методе - посматрана метода може имати више параметара који су међусобно одвојени зарезом. Поред тога, метода не мора имати параметре.
- Тип повратне вредности методе - повратна вредност методе може бити било ког типа (или типа *void* уколико метода нема повратну вредност). Све методе осим конструктора имају тип повратне вредности.
- Тело методе - у оквиру тела методе даје се конкретна имплементација посматране методе. Уколико метода враћа повратну вредност неког типа, у телу методе се наводи кључна реч *return* и врши враћање одговарајуће вредности, у складу са специфицираним типом повратне вредности.

Називом методе и листом параметара одређује се потпис методе. Потпис методе мора бити јединствен, тј. може постојати више метода које се исто зову али се разликују по броју и/или типу параметара [Vlajico8].

Иницијализација објекта класе врши позивом одговарајућег конструктора, коришћењем кључне речи *new*. На тај начин се у меморији, на одређеној меморијској адреси, резервише меморијски простор за објекат класе. Садржај објектне променљиве је референца на креирани објекат, а не сам објекат [Vlajico8], што је и приказано на наредној слици (Слика 5). У том

смислу се за промењљиву *s* каже се да је референтног типа: њен садржај може да буде референца на било који објекат класе *Student*, односно класе која је изведена из класе *Student*.



а) Иницијализација објекта класе коришћењем непараметарског конструктора

б) Иницијализација објекта класе коришћењем параметарског конструктора

Слика 5. Иницијализација објекта класе

На основу приказаног се може закључити да се атрибутима класе дефинише структура. С другим стране, методама се дефинише понашање и врши непосредна промена вредности атрибута.

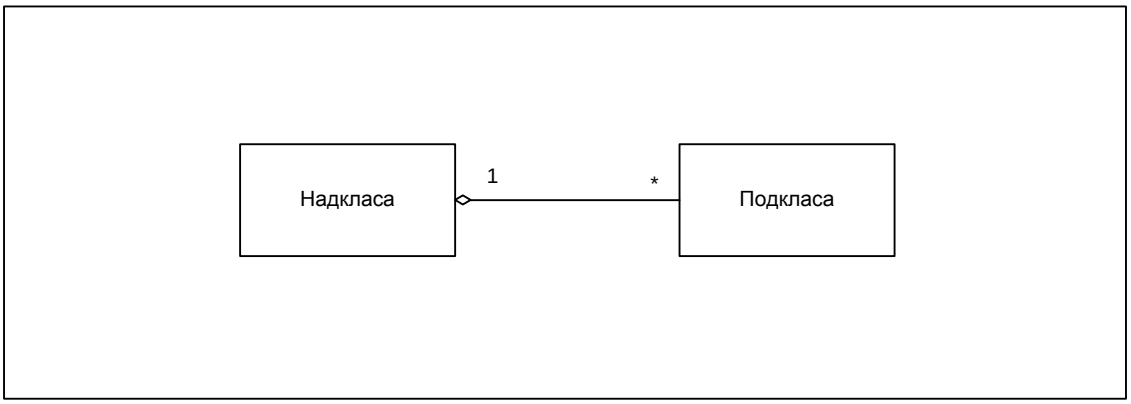
Сваки софтверски систем састоји се од већег броја класа које су на одређени начин логички организоване и користе се за решавање одређеног проблема.

2.1.2. Наслеђивање

Наслеђивање представља један од најважнијих концепата у објектно-оријентисаном програмирању. Применом овог концепта омогућава се дефинисање класе која је изведена из основне класе и на тај начин наслеђује атрибуте и методе основне класе. Основна класа назива се надкласа (енг. superclass), док се изведена класа назива подкласа (енг. subclass) [Vlajico8]. Применом концепта наслеђивања омогућава се прављење хијерархије класа у циљу моделовања неког домена и/или решавања одређеног проблема. На

тај начин добија се скуп класа које су међусобно повезане и логички организоване. Такође, концепт наслеђивања представља кључни концепт у процесу проширивања компоненти и поновног коришћења компоненти [Meyer88].

Једна надкласа може имати више подкласа. С друге стране, једна подкласа изведена је из једне надкласе[1], што је и приказано на наредној слици (Слика 6).



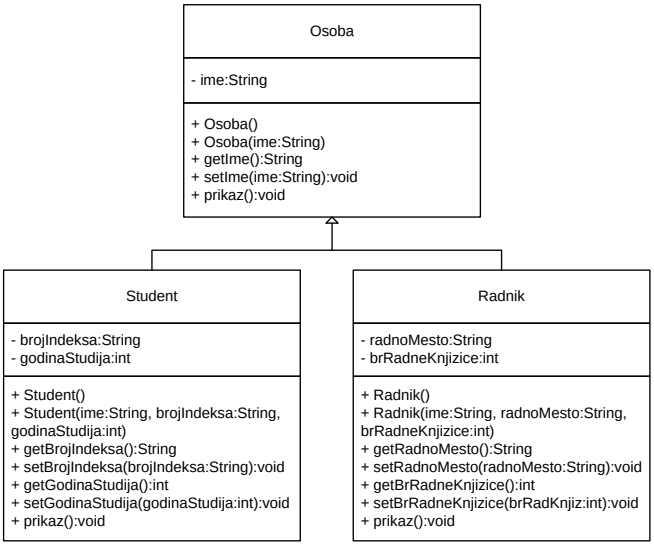
Слика 6. Однос између надкласе и подкласе

Применом концепта наслеђивања изведена класа наслеђује структуру и понашање основне класе. Међутим, то не ограничава подкласу да промени структуру или понашање (у смислу додавања нових атрибута, метода и измене понашања постојећих метода). С друге стране, уколико дође до измене у надкласи, све промене се преносе и на подкласу. У том контексту се може рећи да се промене у надкласи пропагирају и на подкласе [Pressman10]. Међутим, промене у подкласама се не пропагирају на надкласе (подкласа је изведена из надкласе).

Размотримо пример који је приказан на наредној слици (Слика 7). Са слике се уочава класа Osoba са својим атрибутима и методама. Из класе Osoba

[1] Савремени објектно-оријентисани програмски језици као што су Java и C# не подржавају концепт вишеструког наслеђивања класа (енг. multiple class inheritance). Уместо примене вишеструког наслеђивања класа препоручује се коришћење интерфејса што ће бити објашњено у наставку рада.

изведене су класе Student и Radnik које садрже атрибуте и методе основне класе и додају нове атрибуте и методе. У том смислу се за класу Osoba каже да је надкласа или основна класа, док се за класе Student и Radnik каже да су подкласе или изведене класе.



Слика 7. Примена концепта наслеђивања - из основне класе су изведене две подкласе

Концепт наслеђивања је веома важан због тога што се на тај начин омогућава примена концепата полиморфизма, компатибилности објектних типова и касног повезивања који ће бити објашњени у наставку.

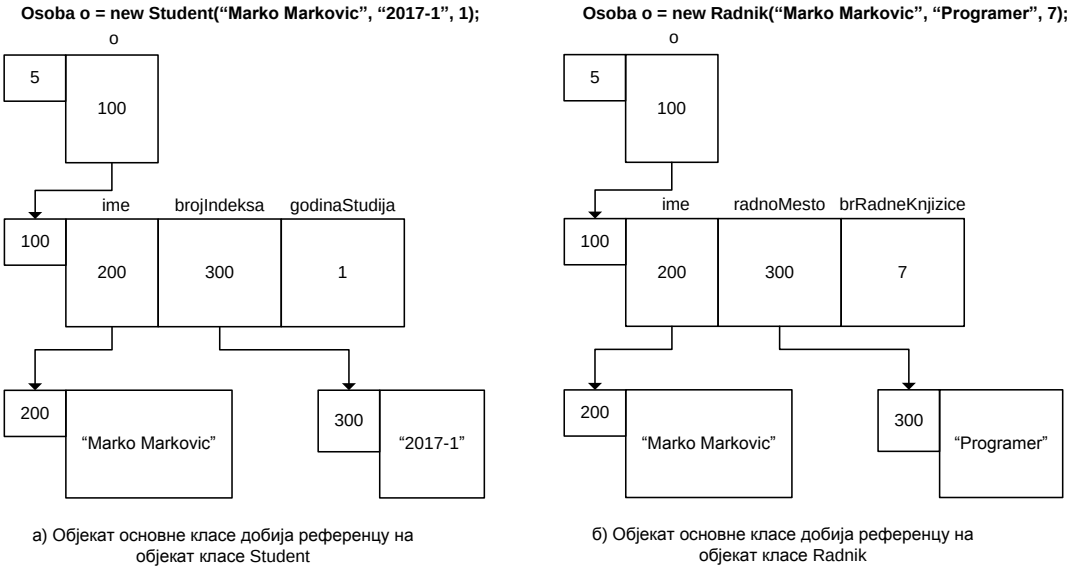
2.1.3. Полиморфизам

У контексту објектно-оријентисаног развоја софтвера полиморфизам (енг. polymorphism) представља концепт којим се дефинише нешто што може имати више облика [Vlajico8].

Раније је напоменуто да у оквиру једне класе може постојати више метода које се исто зову али се разликују по броју и/или типу параметара. Такве методе називају се преклопљене методе (енг. overloaded methods). Преклопљене методе представљају један од облика полиморфизма [Vlajico8]. Уколико размотримо пример приказан на претходној слици (Слика 7), примећује се да постоји преклапање конструктора класа (конструктор такође представља методу, па је и то облик полиморфизма).

С друге стране, са исте слике се уочава да је у класи Osoba дефинисана метода *prikaz()* која приказује податке о конкретној особи. У изведеним класама Student и Radnik извршено је редефинисање поменуте методе тако да се приказују подаци о конкретном студенту и раднику, респективно. Другим речима, примећује се да у основној класи и изведеним класама посматрана метода има исти потпис али се разликује њена имплементација. Овакве методе називају се прекривене методе (енг. overridden methods).

Са концептом полиморфизма повезан је и концепт компатибилности објектних типова. Под концептом компатибилности објектних типова подразумева се могућност да објекат основне класе може да добије референцу било ког објекта изведене класе. Размотримо пример који је дат на наредној слици (Слика 8). С обзиром да су из класе Osoba изведене класе Student и Radnik, објекат класе Osoba (у нашем примеру променљива o) може добити референцу на објекте подкласа. Другим речима, с обзиром да подкласе наслеђују атрибуте и методе надкласа, објекат надкласе може добити референцу на објекат подкласе. С друге стране, објекти подкласа могу имати и додатне чланице (атрибуте и методе), па објекат подкласе не може добити референцу на објекат надкласе.

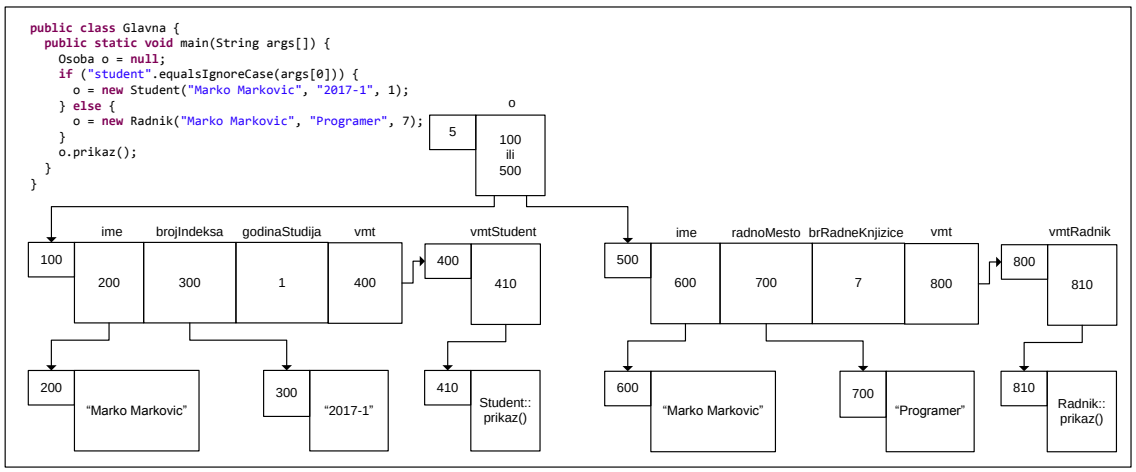


а) Објекат основне класе добија референцу на објекат класе Student

б) Објекат основне класе добија референцу на објекат класе Radnik

Слика 8. Пример компатибилности објектних типова - објекат основне класе добија референцу на различите објекте подкласа

Концептом касног повезивања (енг. late binding) омогућава се повезивање метода са програмом у времену извршавања програма [Vlajico8]. На тај начин је могуће у времену извршавања програма одредити методу коју је потребно позвати, што је и приказано на наредној слици (Слика 9). Примећује се да сваки објекат поседује атрибут *vmt* чији садржај представља адресу табеле метода посматране класе, при чему табела метода садржи низ показивача на адресе метода класе [Vlajic14b]. У примеру се, у зависности од улазних аргумената командне линије, врши иницијализација објекта класе Student или Radnik, а затим се позива метода *prikaz()*. Због тога се у време компајлирања програма наредба *o.prikaz()* трансформише у облик *o.vmt[0]*. Другим речима, у времену компајлирања програма није познато који ће се објекат креирати и чија ће се метода *prikaz()* позвати, већ се то одређује у времену извршавања програма (због тога је садржај референтне промењиве *o* адреса 100 или 500, у зависности од објекта класе који је креиран и додељен промењивој *o*).



Слика 9. Пример касног повезивања у времену извршавања програма

Концепт наслеђивања, заједно са концептима компатибилности објектних типова и касног повезивања, омогућава јак полиморфизам [Vlajico8]. Посматрани концепти су веома значајни зато што омогућавају израду генеричких метода које се веома често користе у процесу објектно-оријентисаног развоја софтвера. Генеричким методама се дефинише опште

понашање, које се може применити за све објекте подкласа, чиме се остварује поновна употреба компоненти (енг. reusability) у посматраном софтверском систему.

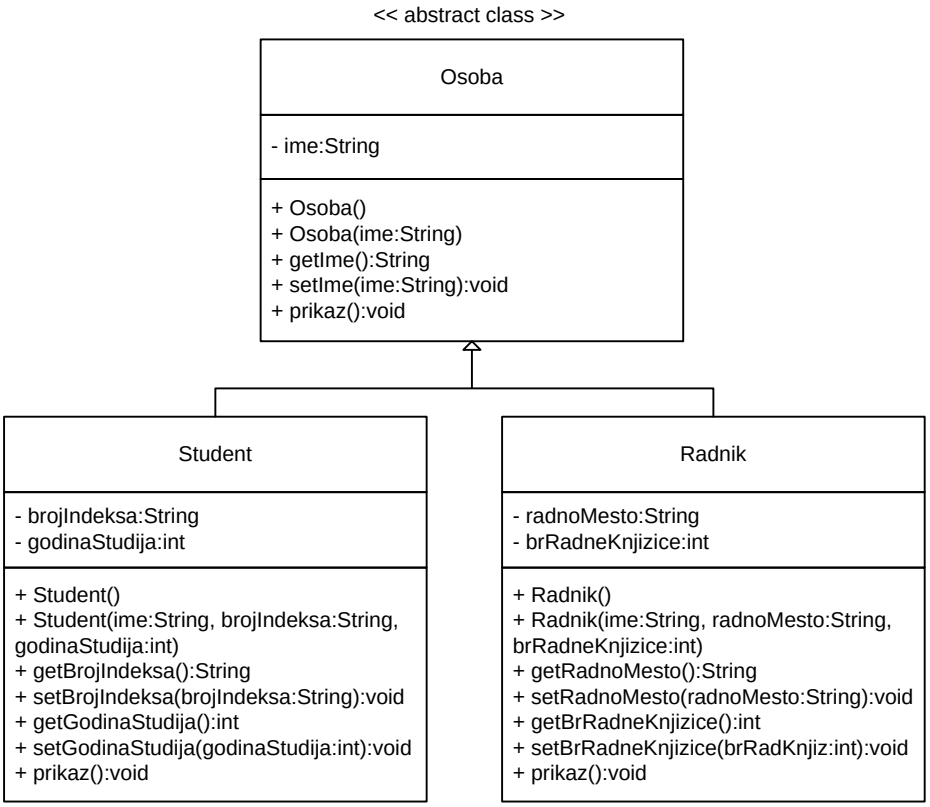
2.1.4. Апстрактна класа

У претходном одељку је дефинисан концепт класе и објашњено је да класа садржи структуру која се дефинише путем атрибута и понашање које се дефинише путем метода. С друге стране, у процесу објектно-оријентисаног развоја софтвера некада није позната имплементација сваке методе. У том смислу, уколико понашање методе није познато, метода се може прогласити апстрактном. Класа која садржи барем једну апстрактну методу такође мора бити апстрактна[2]. У том случају класа која наслеђује апстрактну класу треба да имплементира апстрактну методу. На тај начин се одговорност за имплементацију апстрактне методе преноси на подкласу. Уколико ни подкласа не изврши имплементацију апстрактне методе и она мора бити апстрактна. Посматрани поступак се понавља, све док се не дође до конкретне класе која има имплементацију свих метода (тј. нема апстрактне методе). Међутим, класа може бити апстрактна и уколико нема ни једну апстрактну методу.

Заједничка карактеристика апстрактних класа је да оне не могу имати појављивања (не може се направити нови објекат апстрактне класе). С друге стране, објекат апстрактне класе може добити референцу на било који објекат конкретне класе која је изведена из апстрактне класе, што је и приказано на наредној слици (Слика 10). Са слике се уочава апстрактна класа Osoba из које су изведене класе Student и Radnik. С обзиром да је класа Osoba апстрактна, она се не може инстанцирати, али зато може добити референцу на објекат подкласе и позивати методе подкласе (у

[2] Објектно-оријентисани програмски језици Java и C# за дефинисање апстрактне класе/методе користе кључну реч abstract.

примеру је добијена референца на објекат класе Student и извршен је позив његове методе *prikaz()*).




```
Osoba o = null;
// Apstraktna klasa se ne moze instancirati
// o = new Osoba();
// Apstraktna klasa moze da dobije referencu na objekat podklase
o = new Student("Marko Markovic", "2017-1", 1);
o.prikaz();
```

Слика 10. Пример апстрактне класе из које су изведене две конкретне класе

Такође, важно је напоменути да апстрактне класе могу имати конструкторе. Штавише, конструктори у класама никако не смеју бити апстрактни. Приликом иницијализације подкласа, конструктори се позивају према хијерархији, од надкласа ка подкласама. У том смислу би апстрактни конструктор онемогућио креирање објекта посматране класе.

Апстрактне класе представљају веома значајан и често коришћен концепт у процесу објектно-оријентисаног развоја софтвера. Њиховом применом омогућава се пренос одговорности за реализацију одређеног понашања на

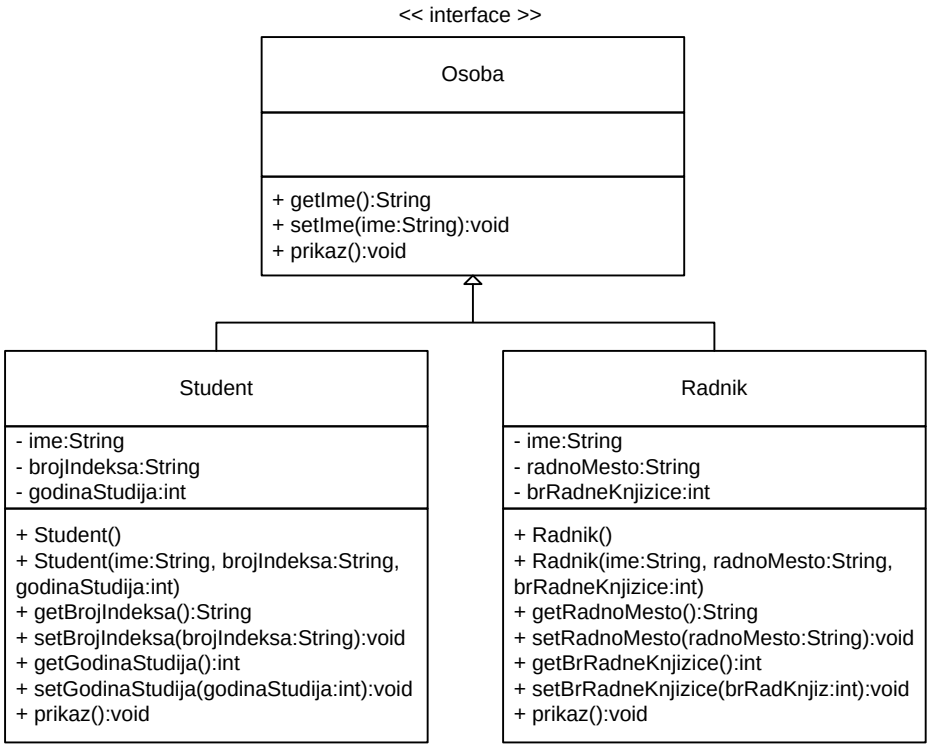
подкласе, чиме се могу дефинисати различите имплементације које се односе на посматрано понашање.

2.1.5. Интерфејс

Интерфејс је објектно-оријентисани концепт којим се даје спецификација метода без њихове имплементације. Другим речима, интерфејсом се даје спецификација метода (тј. даје се потпис метода), при чему се одговорност за њихову имплементацију преноси на класе које имплементирају посматрани интерфејс. На тај начин омогућава се одвајање спецификације од имплементације [Vlajico8]. У том смислу се може рећи да су све методе у интерфејсу апстрактне.

У претходном одељку објашњен је концепт наслеђивања. У том смислу, савремени објектно-оријентисани програмски језици као што су Java и C# не дозвољавају вишеструко наслеђивање класа (свака класа може имати само једну надкласу). С друге стране, једна класа може да имплементира већи број интерфејса, чиме се решава поменути проблем. Такође, важно је напоменути да један интерфејс може да наследи више интерфејса. На тај начин се омогућава прављење читаве хијерархије интерфејса и класа у посматраном софтверском систему.

Слично као апстрактне класе, интерфејси не могу имати појављивања (не може се направити нови објекат интерфејса). С друге стране, објекат интерфејса може добити референцу на било који објекат конкретне класе која је извршила имплементацију посматраног интерфејса, што је и приказано на наредној слици (Слика 11). Са слике се уочава интерфејс Osoba кога непосредно имплементирају класе Student и Radnik. С обзиром да је Osoba интерфејс, она се не може инстанцирати, али зато може добити референцу на објекат који је извршио његову имплементацију и позивати методе подкласе (у примеру је добијена референца на објекат класе Student и извршен је позив његове методе *prikaz()*).




```
Osoba o = null;
// Interfejs se ne moze instancirati
// o = new Osoba();
// Interfejs moze da dobije referencu na objekat koji je izvrsio
njegovu implementaciju
o = new Student("Marko Markovic", "2017-1", 1);
o.prikaz();
```

Слика 11. Пример интерфејса кога имплементирају две конкретне класе

Концепт интерфејса је веома значајан и често коришћен у процесу објектно-оријентисаног развоја софтвера. Он се користи заједно са концептима наслеђивања, полиморфизма, компатибилности објектних типова и апстрактних класа. У том смислу, у процесу објектно-оријентисаног развоја софтвера препоручује се програмирање ка интерфејсима а не ка имплементацији, што представља један од најважнијих принципа објектно-оријентисаног пројектовања софтвера [Gamma94][Martin96][Vlajic14]. На тај начин врши се одвајање спецификације од имплементације компоненти и реализује слаба повезаност између компоненти софтверског система, што последично олакшава даљи развој и одржавање посматраног софтверског система.

2.2. Објектно-оријентисане технологије

У претходном одељку описани су основни објектно-оријентисани концепти: класа и њене чланице, наслеђивање, полиморфизам, компатибилност објектних типова, касно повезивање, апстрактна класа и интерфејс. Између приказаних концепата и објектно-оријентисаних технологија постоји непосредна веза. Наиме, заједничка карактеристика приказаних концепата је да су они подржани од стране свих објектно-оријентисаних технологија. Због тога се у овом одељку даје опис често коришћених објектно-оријентисаних технологија Java и C#. У одељку се, такође, даје упоредни приказ реализације претходно приказаних објектно-оријентисаних концепата у програмским језицима Java и C#.

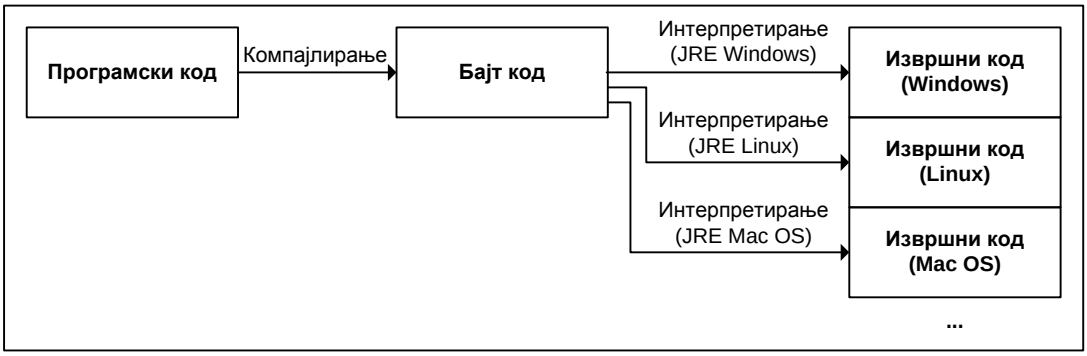
2.2.1. Java

Java је објектно-оријентисана софтверска технологија коју је иницијално пројектовала и имплементирала компанија Sun Microsystems. Поменуту компанију је купила компанија Oracle, тако да она води процес развоја, усавршавања и одржавања Java технологије.

Важно је напоменути да се Java може посматрати као платформа и као објектно-оријентисана софтверска технологија [Vlajic08]. У том смислу постоје различита издања Java платформе (стандардно издање, пословно издање и микро издање). Такође, постоје и различите библиотеке компоненти независних произвођача (нпр. библиотеке компоненти за израду веб сервиса, библиотеке компоненти за пословно извештавање, библиотеке компоненти којима се реализује перзистентност и др.) које се ослањају на Java платформу. Поред тога, постоје и различити програмски језици који се могу извршавати на Java платформи (нпр. програмски језици Scala, Clojure, Groovy, Kotlin итд.).

С друге стране, у процесу развоја софтвера користи се програмски језик Java који подржава објектно-оријентисане концепте.

Једна од најважнијих карактеристика посматране технологије је њена платформска неутралност [Vlajic16b]. Наиме, једном написан програм у програмском језику Java се може извршити на било којем оперативном систему, под условом да за тај оперативни систем постоји одговарајуће окружење за извршавање програма[3] (енг. Java Runtime Environment - JRE). У том смислу се програмски код компајлира у бајт-код који представља платформски независан код. Затим се, путем окружења за извршавање програма, врши интерпретација бајт-код наредби у наредбе које су специфичне за конкретан оперативни систем, што је и приказано на наредној слици (Слика 12).



Слика 12. Концептуални приказ извршавања Java програма

На тај начин се омогућава конструкција софтвера који је инваријантан у односу на оперативни систем. Другим речима, једном написан програм могуће је извршити на различитим оперативним системима, што у значајној мери олакшава процес миграције у случају да се појави потреба за променом оперативног система.

Платформска неутралност, различита издања и велики број библиотека независних произвођача су у великој мери допринели популарности посматране софтверске технологије.

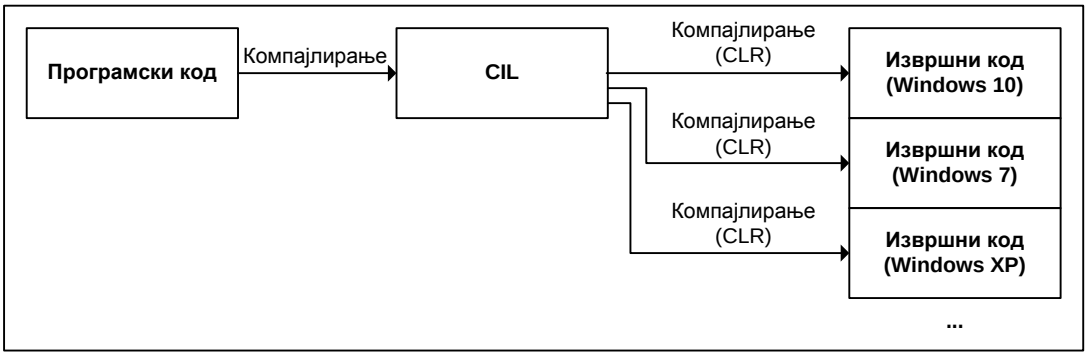
[3] У ранијим верзијама програмског језика за извршно окружење се користио термин Java виртуелна машина (енг. Java Virtual Machine - JVM).

2.2.2. C#

C# је објектно-оријентисана софтверска технологија коју је пројектовала и имплементирала компанија Microsoft, која и води процес развоја, усавршавања и одржавања посматране технологије.

Једна од најважнијих карактеристика посматране технологије је њена платформска неутралност [Vlajic16b]. Наиме, једном написан програм у програмском језику C# се може извршити на различитим оперативним системима. У том смислу се програмски код компајлира у CIL међујезик (енг. Common Intermediate Language - CIL) који представља платформски независан језик.

Сам процес компајлирања се извршава у два корака. Најпре се програмски код компајлира у CIL међујезик а затим се, путем извршног окружења (енг. Common Language Runtime - CLR) врши компајлирање програма за конкретан оперативни систем, што је и приказано на наредној слици (Слика 13).



Слика 13. Концептуални приказ извршавања C# програма

Раније је напоменуто да се програм написан у програмског језика Java компајлира у бајт-код, након чега се коришћењем извршног окружења врши његова интерпретација за конкретан оперативни систем. С друге стране, програм написан у програмском језику C# се компајлира у извршни код за посматрани оперативни систем, што значи да није потребно његово интерпретирање. У том смислу је могуће остваривање бољих перформанси у

процесу извршавања програма који су имплементирани коришћењем C# софтверске технологије [Eaddy01][Singer03].

Поред тога, C# технологија је интегрисана са осталим софтверским технологијама компаније Microsoft (нпр. са Microsoft Team Foundation Server системом за управљање програмским кодом, Microsoft Visual Studio окружењем за развој софтвера итд.) и подржана је од стране независних произвођача софтвера који развијају различите компоненте и развојна окружења. Због тога је стекла велику популарност и примену у процесу развоја софтвера.

2.2.3. Упоредни приказ реализације објектно-оријентисаних концепата у софтверским технологијама Java и C#

У овом одељку даје се упоредни приказ реализације објектно-оријентисаних концепата у софтверским технологијама Java и C#. У том смислу се најпре даје пример, а затим се приказује имплементација у поменутим технологијама.

1. Дефиниција класе и њених чланица

Нека је дата класа Operacija која као атрибуте има два цела броја. Поред тога, класа садржи параметарски и непараметарски конструктор, као и методе за узимање и постављање вредности атрибута (енг. getter and setter methods). Упоредни приказ дефиниције посматране класе и њених чланица приказан је у наредној табели (Табела 1).

Табела 1. Упоредни приказ дефиниције класе и њених чланица

Java	C#

`public class Operacija {`
`  protected int x;`
`  protected int y;`
`  public Operacija() {`
`  x = 0;`
`  y = 0;`
`  }`
`  public Operacija(int x, int y) {`
`  this.x = x;`
`  this.y = y;`
`  }`
`  public int getX() {`
	`public class Operacija`
	`{`
	`  protected int x { get; set; }`
	`  protected int y { get; set; }`
	`  public Operacija()`
	`  {`
	`  x = 0;`
	`  y = 0;`
	`  }`
	`  public Operacija(int x, int y)`
	`  {`
	`  this.x = x;`

Java	C#

`    return x;`
`  }`
`  public void setX(int x) {`
`    this.x = x;`
`  }`
`  public int getY() {`
`    return y;`
`  }`
`  public void setY(int y) {`
`    this.y = y;`
`  }`
`}`
	`    this.y = y;`
	`  }`
	`}`

2. Примена концепта наслеђивања

За класу Operacija потребно је дефинисати методу *operacija()*. Дефинисати подкласе Sabiranje и Oduzimanje. У подкласама извршити прекривање методе *operacija()*. На тај начин се показује примена концепта наслеђивања, што је и приказано у наредној табели (Табела 2).

Табела 2. Упоредни приказ примене концепта наслеђивања

Java	C#

`public class Operacija {`
 	
`  // Atributi, konstruktori i metode su isti`
 	
`  public int operacija() {`
`    return 0;`
`  }`
`}`
`class Sabiranje extends Operacija {`
`  @Override`
`  public int operacija() {`
`    return x + y;`
`  }`
`}`
`class Oduzimanje extends Operacija {`
`  @Override`
`  public int operacija() {`
`    return x - y;`
`  }`
`}`
	`public class Operacija`
	`{`
	`  // Atributi, konstruktori i metode su isti`
	
	`  public virtual int operacija()`
	`  {`
	`    return 0;`
	`  }`
	`}`
	`class Sabiranje : Operacija`
	`{`
	`  public override int operacija()`
	`  {`
	`    return x + y;`
	`  }`
	`}`
	`class Oduzimanje : Operacija`
	`{`
	`  public override int operacija()`
	`  {`
	`    return x - y;`
	`  }`
	`}`

3. Примена концепта апстрактне класе

За класу Operacija потребно је дефинисати апстрактну методу *operacija()*. Дефинисати подкласе Sabiranje и Oduzimanje. У подкласама дати

имплементацију апстрактне методе. На тај начин се показује примена концепта апстрактне класе, што је и приказано у наредној табели (Табела 3).

Табела 3. Упоредни приказ примене концепта апстрактне класе

Java	C#

`public abstract class Operacija {`
 	
`  // Atributi, konstruktori i metode su isti`
 	
`  public abstract int operacija();`
`}`
`class Sabiranje extends Operacija {`
`  @Override`
`  public int operacija() {`
`  return x + y;`
`  }`
`}`
`class Oduzimanje extends Operacija {`
`  @Override`
`  public int operacija() {`
`  return x - y;`
`  }`
`}`
	`public abstract class Operacija`
	`{`
	`  // Atributi, konstruktori i metode su isti`
	
	`  public abstract int operacija();`
	`}`
	`class Sabiranje : Operacija`
	`{`
	`  public override int operacija()`
	`  {`
	`  return x + y;`
	`  }`
	`}`
	`class Oduzimanje : Operacija`
	`{`
	`  public override int operacija()`
	`  {`
	`  return x - y;`
	`  }`
	`}`

4. Примена концепата полиморфизма, компатибилности објектних типова и касног повезивања

У зависности од аргумента командне линије извршити иницијализацију одговарајуће класе (Sabiranje или Oduzimanje) и позвати методу *operacija()*. На тај начин се показује примена концепата полиморфизма, компатибилности објектних типова и касног повезивања, што је и приказано у наредној табели (Табела 4).

Табела 4. Упоредни приказ примене концепата полиморфизма, компатибилности објектних типова и касног повезивања

Java	C#

`public abstract class Operacija {`
 	
`  // Atributi, konstruktori i metode su isti`
 	
`  public static void main(String[] args) {`
`  Operacija o = null;`
`  if ("Sabiranje".equalsIgnoreCase(args[0])) {`
`  o = new Sabiranje();`
`  } else {`
`  o = new Oduzimanje();`
`  }`
`  o.setX(10);`
	`public abstract class Operacija`
	`{`
	`  // Atributi, konstruktori i metode su isti`
	
	`  static void Main(string[] args)`
	`  {`
	`  Operacija o = null;`
	`  if ("Sabiranje".Equals(args[0],`
	`StringComparison.OrdinalIgnoreCase))`
	`  {`
	`  o = new Sabiranje();`
	`  }`

Java	C#

o.setX(9);
System.out.println("Rezultat: " +
o.operacija()); }
}
	else
	{
	o = new Oduzimanje();
	}
	o.x = 10;
	o.y = 9;
	Console.WriteLine("Rezultat: " +
	o.operacija());
	}
	}

5. Примена концепта интерфејса

Нека је дат интерфејс Operacija који садржи методу *operacija()*. У класама Sabiranje и Oduzimanje извршити имплементацију методе посматраног интерфејса. На тај начин се показује примена концепта имплементације интерфејса, што је и приказано у наредној табели (Табела 5).

Табела 5. Упоредни приказ примене концепта имплементације интерфејса

Java	C#

public interface Operacija {
int operacija(int x, int y);
}
 	
class Sabiranje implements Operacija {
@Override
public int operacija(int x, int y) {
return x + y;
}
}
 	
class Oduzimanje implements Operacija {
@Override
public int operacija(int x, int y) {
return x - y;
}
}
	public interface Operacija
	{
	int operacija(int x, int y);
	}
	class Sabiranje : Operacija
	{
	public int operacija(int x, int y)
	{
	return x + y;
	}
	}
	class Oduzimanje : Operacija
	{
	public int operacija(int x, int y)
	{
	return x - y;
	}
	}

2.3. Окружења за развој објектно-оријентисаних софтверских система

Уколико се разматра процес развоја софтвера, сам процес имплементације је могуће извршити коришћењем текст едитора (нпр. коришћењем текст едитора Notepad, Notepad++, Sublime Text, Atom и др.). Међутим, у ту сврху се често користе окружења за развој софтверских система која у значајној

мери олакшавају процес развоја софтвера. Због тога се у овом одељку даје опис NetBeans, Eclipse и Visual Studio окружења за развој објектно-оријентисаних софтверских система.

2.3.1. NetBeans

NetBeans окружење за развој софтверских система је производ компаније Oracle и подржава објектно-оријентисане концепте програмског језика Java.

Окружење подржава развој софтверских система за различита издања Java платформе (тј. за стандардно издање, пословно издање и микро издање). Такође, у оквиру окружења су интегрисане библиотеке класа независних произвођача (нпр. библиотеке Spring и Hibernate су подразумевано доступне у оквиру развојног окружења).

NetBeans окружење у значајној мери олакшава процес развоја софтвера. У том смислу, окружење има подршку за комплетирање програмског кода и у процесу писања програма врши проверу синтаксе. У случају настанка грешке окружење подвлачи програмски код који садржи грешку. На тај начин се омогућава уочавање грешака пре компајлирања програма. Поред тога, окружење даје предлог како насталу грешку треба исправити. Окружење подржава компајлирање, линковање и извршавање програма, као и процес отклањања грешака у програму (енг. debugging).

У оквиру окружења је интегрисана документација која се односи на класе и методе објектно-оријентисаног језика Java. С обзиром на велики број класа и метода које садржи програмски језик Java, документација у значајној мери олакшава развој посматраног софтверског система.

Поред тога, NetBeans окружење подржава израду додатака (енг. plugins) чиме је могуће проширити скуп функционалности које су подржане од стране посматраног окружења.

2.3.2. Eclipse

Eclipse окружење за развој софтверских система је производ Eclipse фондације и подржава објектно-оријентисане концепте програмског језика Java.

Eclipse подржава развој софтвера за стандардно, пословно и микро издање Java платформе. С друге стране, окружење подржава израду додатака (енг. plugins) чиме се обезбеђује подршка за различите софтверске технологије. У том смислу окружење може подржати и друге објектно-оријентисане софтверске технологије (нпр. програмски језик PHP).

Слично као и NetBeans окружење, Eclipse окружење олакшава процес развоја софтвера тако што подржава комплетирање програмског кода. Приликом израде програма окружење врши проверу синтаксе, а у случају појаве грешке окружење подвлачи програмски код који садржи грешку. Поред тога, окружење садржи брзе исправке (енг. quick fixes) којима се даје предлог како насталу грешку треба исправити. Eclipse такође омогућава компајлирање, линковање, извршавање програма, као и процес отклањања грешака у програму, чиме се у значајној мери олакшава процес развоја софтвера.

2.3.3. Visual Studio

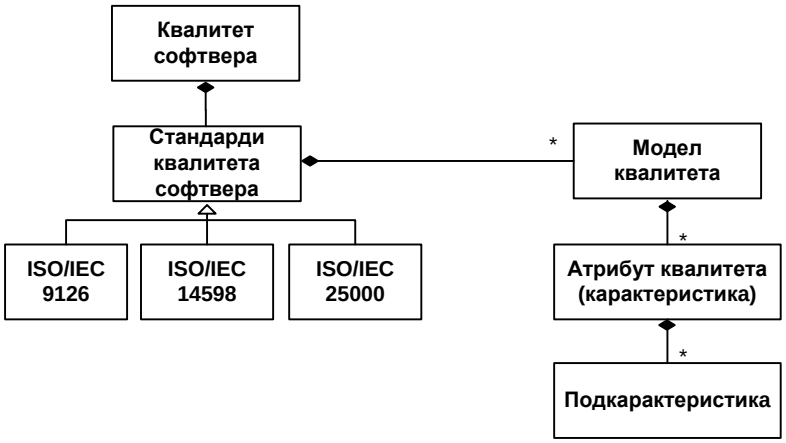
Visual Studio окружење за развој софтверских система је производ компаније Microsoft и подржава објектно-оријентисане концепте програмског језика C#.

Посматрано окружење представља најбоље окружење за развој софтверских система који су засновани на .NET технологији. У том смислу окружење подржава и друге програмске језике компаније Microsoft (нпр. подржани су програмски језици Visual Basic, Visual C++, F# итд.). Поред тога, окружење пружа подршку за интеграцију са другим технологијама компаније Microsoft (нпр. подржана је интеграција са Microsoft SQL Server базом података, интеграција са Microsoft Team Foundation Server системом за управљање програмским кодом итд.).

Слично као NetBeans и Eclipse окружења, Visual Studio окружење у значајној мери олакшава процес развоја софтвера. У том смислу је обезбеђена подршка за комплетирање програмског кода и провера синтаксе у процесу писања програма. Уколико у процесу писања програма настане грешка окружење подвлачи програмски код који садржи грешку. Поред тога, окружење даје предлог како насталу грешку треба исправити. Visual Studio окружење такође подржава компајлирање, линковање и извршавање програма, као и отклањање грешака у програму.

3. Стандарди квалитета софтвера

У оквиру овог поглавља биће разматрани стандарди квалитета софтвера. Најпре ће бити дата дефиниција квалитета софтвера а затим представљени стандарди квалитета софтвера који се примењују у процесу развоја софтвера. У том смислу биће описани стандарди Међународне организације за стандардизацију и Међународне електротехничке комисије ISO/IEC 9126, ISO/IEC 14598 и ISO/IEC 25000 којима се дефинише квалитет софтвера. Поменути стандарди квалитета софтвера се могу користити за дефинисање и евалуацију квалитета сваког софтверског система. Стандардом квалитета софтвера дефинишу се модели квалитета софтвера. С друге стране, моделом квалитета софтвера дефинишу се атрибути квалитета софтвера (тј. карактеристике квалитета), док свака карактеристика може бити декомпонована на више подкарактеристика. На наредној слици (Слика 14) дат је концептуални преглед поглавља.



Слика 14. Стандарди квалитета софтвера

3.1. Дефиниција квалитета софтвера

Као што је раније напоменуто, процес индустријске производње је веома сложен па је, у циљу побољшања производних резултата, развијен велики број метода индустријске производње које решавају различите проблеме. У

том смислу је потребно извршити добру организацију процеса производње како би се обезбедила израда производа који немају дефекте, односно како би се број дефеката свео на најмању могућу меру [Kruchten04] [Sommerville04]. Дефектни производ свакако представља проблем: уколико се дефект уочи потребно је уложити додатне ресурсе (људске, материјалне, новчане и др.) за исправку уочених неусаглашености. Такође, веома је важан тренутак (или фаза процеса производње) када се неусаглашеност уочи, као и број производа који садрже неусаглашености. Уколико се дефект уочи на крају процеса производње, ресурси потребни за његово отклањање могу бити велики. Све ово може представљати значајан трошак. Из тих разлога се може закључити да је, између осталог, потребно повећати ниво квалитета процеса производње и самих производа. Управљање квалитетом и контрола квалитета представљају значајан део сваког процеса производње. У најопштијем смислу, квалитет се може дефинисати као "усаглашеност са захтевима" [Gaffney81], односно као "усаглашеност производа са детаљном спецификацијом производа" [Crosby80].

С друге стране, квалитет можемо посматрати и са стране корисника (купца) производа. Производ свакако мора бити функционалан и квалитетно израђен. У том смислу, производ мора омогућити кориснику да задовољи одређене потребе. Међутим, веома важан аспект који корисник узима у обзир је и сама цена производа: он може бити веома функционалан, квалитетно израђен, али корисник можда не буде спреман да то плати. Стога је потребно направити производ који садржи вредност коју је корисник спреман да плати.

Сличан начин размишљања се може применити и у процесу развоја софтвера. Данашњи софтверски системи су веома сложени, састоје се од великог броја модула, компоненти и у себи интегришу различите софтверске технологије. При томе, софтверски системи еволуирају с временом па је потребно њихово одржавање и измена како би се прилагодили захтевима корисника. Сходно томе, у сам процес развоја

софтвера потребно је уградити стандарде и процедуре како би се овим променама управљало на ефикасан начин. Чланови пројектног тима који развијају софтвер се мењају (због преласка на другу позицију у компанији, преласка у други развојни тим, преласка у други компанију итд.) што намеће потребу да се процес развоја софтвера учини независним у односу на промене у развојном тиму. Другим речима, потребно је развити софтверски систем који је у складу са дефинисаним моделом квалитета софтвера.

Постоје различите дефиниције квалитета софтвера:

- "Квалитет софтвера значи усаглашеност са захтевима" [Letouzey12b]
- "Квалитет софтвера се може дефинисати као способност софтверског производа да задовољи захтеване и подразумеване потребе у одређеним условима" [ISO25010]
- "Квалитет софтвера се може дефинисати као тоталитет атрибута квалитета софтверског производа који детерминишу његову могућност да задовољи захтеване и подразумеване потребе приликом примене у одређеним условима" [ISO9126]
- "Квалитет софтвера представља степен у којем систем, компонента или процес задовољава постављене захтеве, потребе или очекивања корисника" [Galin04]
- С друге стране, Pressman у књизи [Pressman10] даје следећу дефиницију квалитета софтвера: "Квалитет софтвера представља ефикасан софтверски процес чији је резултат користан софтверски производ који пружа мерљиве вредности за оне који га производе и оне који га користе". Ова дефиниција је веома добра зато што квалитет посматра као саставни део процеса развоја софтвера. Развој софтвера је веома сложен процес који укључује велики број модела, стратегија, метода и активности. Свака метода развоја има своје предности и недостатке и даје боље резултате у развоју одређених софтверских система. Међутим, све методе развоја софтвера за циљ

имају стварање високо-квалитетног софтвера. У том смислу се може поставити питање шта је заправо квалитетан софтвер? Такође, примећује се да је процес развоја усмерен ка креирању корисног софтверског производа и да се вредности које софтвер додаје могу измерити. Ове вредности се посматрају из две перспективе: перспективе пројектног тима који непосредно учествује у процесу развоја софтвера, односно перспективе корисника који непосредно користе софтвер. На тај начин се добија софтверски систем који је функционалан, поуздан, добро пројектован, сигуран, тестиран и једноставан за одржавање.

Међутим, ове карактеристике софтвера је веома тешко дефинисати и измерити кроз сам процес развоја софтвера: процес је усмерен на извршавање одређених активности, примену метода развоја софтвера и стратегија развоја софтвера које за резултат стварају одређену верзију, инкремент софтвера. Исправност тог софтвера се може утврдити различитим методама: инспекцијом кода, писањем тестова, тестирањем софтвера од стране развојног тима, извршавањем софтвера уз коришћење алата за отклањање грешака (енг. debugger), непосредним тестирањем софтвера од стране корисника, израдом прототипова [Liu10]. На тај начин се врши провера да ли је софтвер имплементиран у складу са функционалним захтевима корисника, односно врши се провера семантичке исправности софтверског система. Предуслов за проверу семантичке исправности софтвера је синтаксна исправност софтвера. Синтаксна исправност софтвера се може проверити приликом компајлирања програма, коришћењем компајлера и алата за изградњу софтверског система. У том процесу врши се лексичка анализа софтвера, предпроцесирање програмског кода, парсирање програма, семантичка анализа програмског кода, оптимизација програмског кода и друге активности потребне за процес изградње софтвера. Уколико је програм синтаксно неисправан, алати за изградњу ће приликом покушаја превођења приказати грешку и прекинути процес изградње извршног

програма. Приликом престанка процеса изградње извршног програма алати приказују насталу грешку, са детаљним описом насталог проблема и места у програмском коду који је довео до проблема. Алати често садрже и упутство како треба разрешити настали проблем тако да је синтаксне грешке могуће на једноставан начин исправити. С друге стране, семантичке грешке је много теже уочити: оне се не могу уочити изградњом извршног софтвера већ је неопходно применити неку од раније поменутих метода. Регресионо тестирање значајно помаже у процесу исправљања семантичких грешака програма. На тај начин је приликом измене постојеће или додавања нове функционалности могуће на једноставан начин ефикасно извршити све тестове и проверити да ли настале промене утичу на остатак софтверског система. Алати за тестирање дају детаљне извештаје о насталим проблемима што значајно може да помогне у отклањању насталих грешака.

Међутим, захтеви као што су сигурност, поузданост и једноставност за одржавање се не могу проверити на овај начин. Ови захтеви представљају нефункционалне карактеристике софтверског система, односно атрибуте квалитета софтверског система [Gorton11][Vlajic15] па се, најчешће, дефинишу посебни стандарди и процедуре на основу којих се врши њихова провера. У том смислу се може рећи да квалитет софтвера у значајној мери зависи од нефункционалних захтева [Sommerville11][Vlajic15].

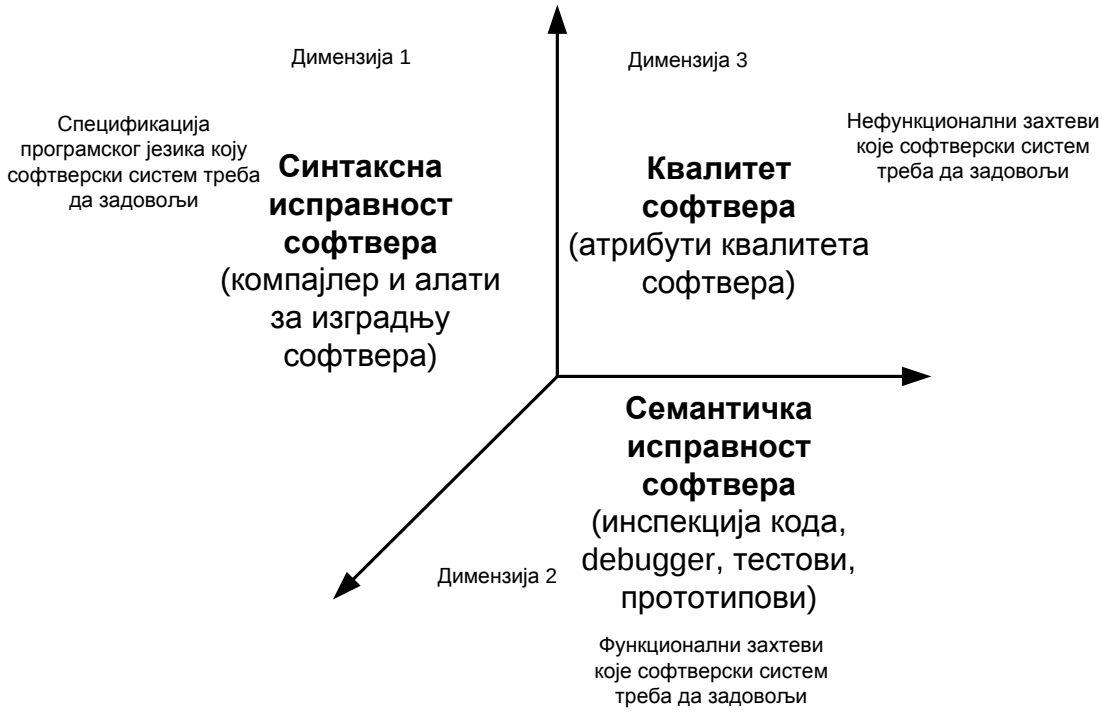
Boehm је дефинисао следеће атрибуте квалитета софтвера: безбедност (енг. safety), разумљивост (енг. understandability), преносивост (енг. portability), сигурност (енг. security), могућност тестирања (енг. testability), корисност (енг. usability), поузданост (енг. reliability), адаптивност (енг. adaptability), могућност поновног коришћења (енг. reusability), еластичност (енг. resilience), модуларност (енг. modularity), ефикасност (енг. efficiency), робусност (енг. robustness), сложеност (енг. complexity) и могућност учења (енг. learnability) [Boehm76][Boehm78][Sommerville11]. Ово је једна од могућих класификација. Различити стандарди квалитета софтвера дефинишу различите атрибуте квалитета софтвера. Као што се може

приметити постоји велики број атрибута квалитета софтвера који су комплементарни и допуњују се међусобно. Међутим, веома је тешко направити софтверски систем код кога ће сваки атрибут квалитета имати максималне вредности [Sommerville11]. Стога је веома важно да се у процесу развоја софтвера одреде најважнији атрибути квалитета софтвера за софтверски систем који се развија, као и да се дефинишу прихватљиве вредности за сваки атрибут квалитета софтвера.

На основу изложеног можемо рећи да у процесу развоја софтвера разликујемо три димензије процеса развоја софтвера:

- **Синтаксну исправност софтвера** - Синтаксна исправност софтвера представља основу за развој софтверског система и условљена је спецификацијом програмског језика у којем се пројектује, имплементира и тестира софтверски систем. Синтаксна исправност софтвера проверава се од стране компајлера и алата за изградњу извршног софтверског система.
- **Семантичка исправност софтвера** - Семантичка исправност софтвера проверава се инспекцијом кода, тестовима, израдом прототипова и другим методама. Семантичком анализом софтвера врши се провера функционалних захтева.
- **Квалитет софтвера** - Анализом квалитета софтвера врши се провера нефункционалних захтева. У том смислу потребно је извршити спецификацију атрибута квалитета које софтверски систем треба да задовољи.

Димензије процеса развоја софтвера приказане су на наредној слици (Слика 15).



Слика 15. Димензије процеса развоја софтвера

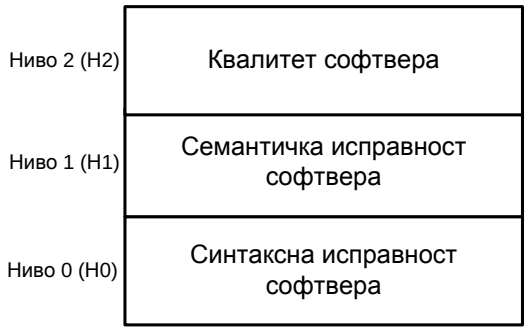
Може се закључити да се сваки софтверски систем карактерише синтаксном исправношћу софтвера, семантичком исправношћу софтвера и квалитетом софтвера. Наведене димензије развоја софтвера су комплементарне и могуће их је представити и на други начин, по нивоима:

- Синтаксна исправност софтвера се налази на нултом нивоу (ниво Н0). Синтаксна исправности софтвера представља темељ на којем се заснива развој софтверског система. Синтаксно исправан софтвер не значи да је он и семантички исправан, као ни да је он квалитетан, али је, с друге стране, синтаксна исправност **неопходан услов за даљу анализу (за анализу семантичке исправности и анализу квалитета софтвера)**.
- На нивоу изнад (ниво Н1) налази се семантичка исправност софтвера. Уколико је софтвер синтаксно исправан могуће је извршити проверу семантичке исправности софтвера. Семантичка исправност

представља **неопходан услов за даљу анализу (за анализу квалитета софтвера)**.

- Уколико је софтвер синтаксно и семантички исправан, може се проверити његов квалитет. Због тога се квалитет софтвера налази на другом нивоу (ниво H2), изнад нивоа H1. Квалитет софтвера се може проверити и без провере семантичке исправности, али је у том случају непознато да ли је софтверски систем семантички исправан.

Нивои процеса развоја софтвера приказани су на наредној слици (Слика 16).

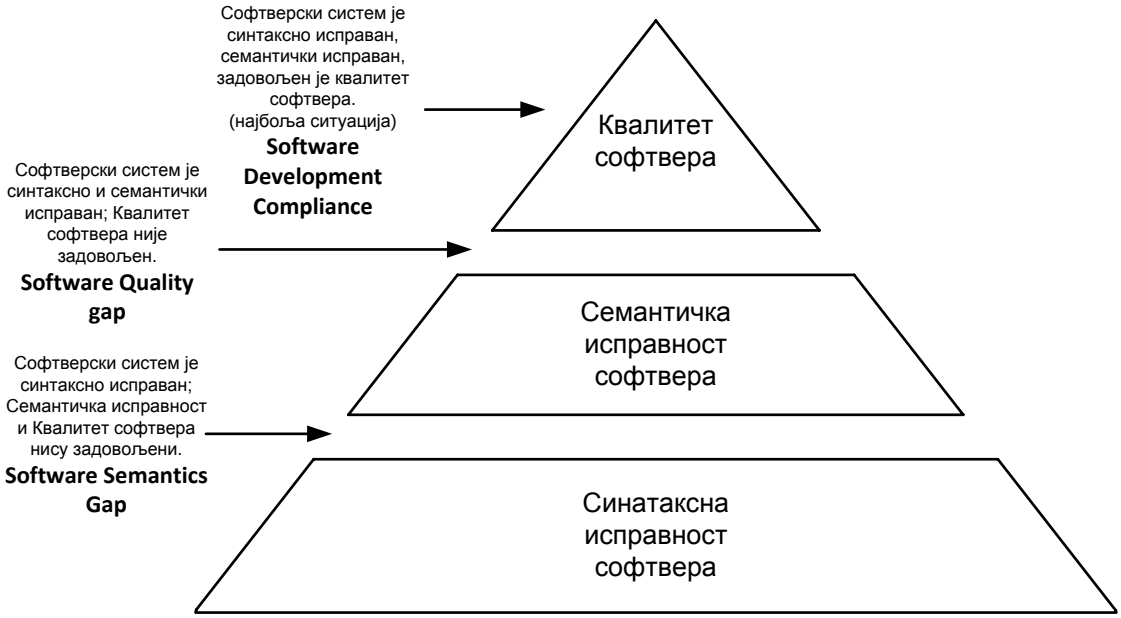


Слика 16. Нивои процеса развоја софтвера

У процесу развоја софтвера постоје три могуће ситуације усклађености синтаксне и семантичке исправности софтвера са квалитетом софтвера (енг. Software Development Compliance):

- Софтверски систем је синтаксно и семантички исправан, задовољен је квалитет софтвера. У том смислу је постигнута **усклађеност нивоа процеса развоја софтвера (енг. Software Development Compliance)**.
- Софтверски систем је синтаксно и семантички исправан. Квалитет софтвера није задовољен. То значи да постоји **раскорак у смислу квалитета софтвера (енг. Software Quality Gap)**.
- Софтверски систем је синтаксно исправан. Семантичка исправност и квалитет софтвера нису задовољени. То значи да постоји **раскорак у смислу семантичке исправности софтвера (енг. Software Semantics Gap)**.

На наредној слици (Слика 17) приказана је усклађеност нивоа процеса развоја софтвера.



Слика 17. Усклађеност нивоа процеса развоја софтвера

Квалитет софтвера је препознат као важан концепт у пракси и у научним истраживањима. Стога је у оквиру водича кроз области знања софтверског инжењерства (енг. Software Engineering Body of Knowledge - SWEBOK), дефинисана посебна област знања која се бави проучавањем квалитета софтвера [Bourque14].

У претходним деценијама је порасла потреба за квалитетом софтвера. Примена софтверских система у данашњем пословању је веома разноврсна, па се стога може рећи да је његово исправно функционисање од велике важности за пословни успех. Зато је веома важно да се приликом развоја софтвера већ у фази планирања дефинише специфично значење квалитета софтвера [Alsultanny09], како би се правовремено уочили неусаглашености у процесу развоја софтвера [Jiango8]. Софтверски стандарди и модели квалитета софтвера имају веома важну улогу у софтверском инжењерству [Schneidewindo2].

С друге стране, веома важан аспект у изради софтверског производа је цена његовог развоја. Узимајући у обзир чињеницу да развој софтвера представља сложен и структуриран процес, који се састоји из скупа активности у којима учествује велики број софтверских инжењера, јасно је да од времена и напора које је потребно уложити у израду софтвера у великој мери зависи и његова цена [Antovic12]. Све ово утиче на цену развоја софтвера али и на цену финалног софтверског производа. Примена стандарда квалитета софтвера и модела квалитета софтвера у процесу развоја софтвера у значајној мери утиче на смањење цене софтверског производа.

Осигурање квалитета софтвера представља потребу сваке софтверске компаније. У крајњем случају дефект у софтверском систему може изазвати велике проблеме, нпр. повећање трошкова одржавања софтвера, што може довести до незадовољства крајњих корисника. Једна од метода коју користе многе организације је придржавање стандарда у писању програмског кода што обезбеђује униформност и прати најбоље праксе писања програмског кода како би се, у складу са дефинисаним моделом квалитета, максимизирали атрибути квалитета софтверског система.

Квалитет софтвера се може проверити статичком и динамичком анализом квалитета софтвера. Статичка анализа омогућава проверу квалитета софтвера без непосредног покретања самог софтвера. С друге стране, динамичка анализа усмерена је на анализу програма који се непосредно извршава и обухвата анализу перформанси (нпр. време потребно за извршавање одређене операције, рачунарске ресурсе који су потребни за извршавање одређене операције: заузеће меморије, број активних нити, заузеће процесора и сл.). Динамичка и статичка анализа су комплементарне технике за анализу квалитета софтвера [Ball99]. С обзиром да је за динамичку анализу неопходно покретање софтвера, а да се у раду разматра квалитет софтвера у контексту побољшања објектно-оријентисаних софтверских система и промовише стална инспекција и стално побољшање

свих делова софтвера (који не морају бити извршиви), фокус ће бити усмерен на статичку анализу квалитета софтвера. Коришћењем алата за анализу квалитета софтвера омогућава се брже проналажење грешака и дефеката у софтверу, као и њихово исправљање у раним фазама процеса развоја софтвера, када их је много лакше и јефтиније исправити [Johnson13].

Међународна организација за стандардизацију (енг. International Organization for Standardization - ISO) и Међународна електротехничка комисија (енг. International Electrotechnical Commission - IEC) представљају међународне организације чије су активности усмерене ка дефинисању међународних стандарда у различитим областима. Стандарди ISO/IEC 25000 [ISO25000], ISO/IEC 14598 [ISO14598] и ISO/IEC 9126 [ISO9126] су посебно значајни за софтверско инжењерство. ISO/IEC 14598 дефинише општи оквир за евалуацију софтверског производа коришћењем модела квалитета који је дефинисан ISO/IEC 9126 стандардом [Bevan06]. Стандард ISO/IEC 9126 дефинише атрибуте квалитета софтвера, као и софтверске метрике за сваки атрибут квалитета [Schackmann09]. Ове метрике се користе за мерење квалитета софтверског система. Нова серија стандарда ISO/IEC 25000 треба да замени и прошири ISO/IEC 9126 и ISO/IEC 14598 стандарде [Bevan05]. У том смислу врши се ревизија постојећих стандарда и постепено повлачење старих верзија. На пример, уколико се разматра стандард квалитета софтвера ISO/IEC 14598, може се приметити да су активна само два његова дела, док су остали делови повучени из употребе.

Сваки стандард квалитета софтвера састоји се из више делова. У наставку овог рада биће представљени стандарди ISO/IEC 9126, ISO/IEC 14598 и ISO/IEC 25000.

3.2. ISO/IEC 9126 стандард квалитета софтвера

ISO/IEC 9126 стандард квалитета софтвера усмерен је ка дефинисању квалитета софтверског производа. Овај стандард се може применити за

утврђивање карактеристика сваког софтверског производа. Стандард се састоји из четири дела [ISO9126]:

- ISO/IEC 9126-1 Software engineering - Product quality - Part 1: Quality model - овај део ISO/IEC 9126 стандарда квалитета софтвера дефинише модел квалитета софтвера. Моделом квалитета софтвера дефинишу се атрибути квалитета софтвера (уобичајени термин који се користи у стандардима је: карактеристике квалитета). Свака карактеристика квалитета садржи већи број подкарактеристика квалитета. У том смислу стандард дефинише карактеристике/подкарактеристике које се могу користити за одређивање интерног и екстерног квалитета софтвера, односно употребног квалитета софтвера.
- ISO/IEC 9126-2 Software engineering - Product quality - Part 2: External metrics - Дефинише екстерне софтверске метрике (софтверске метрике које се користе за утврђивање екстерног квалитета софтвера). Овај део стандарда користи се у комбинацији са ISO/IEC 9126-1 стандардом.
- ISO/IEC 9126-3 Software engineering - Product quality - Part 3: Internal metrics - Дефинише интерне софтверске метрике (софтверске метрике које се користе за утврђивање интерног квалитета софтвера). Овај део стандарда користи се у комбинацији са ISO/IEC 9126-1 стандардом.
- ISO/IEC 9126-4 Software engineering - Product quality - Part 4: Quality in use metrics - Дефинише употребни квалитет (или квалитет у употреби) софтверских метрика. Овај део стандарда користи се у комбинацији са свим претходно наведеним деловима стандарда (ISO/IEC 9126-1 стандардом, ISO/IEC 9126-2 стандардом и ISO/IEC 9126-3 стандардом).

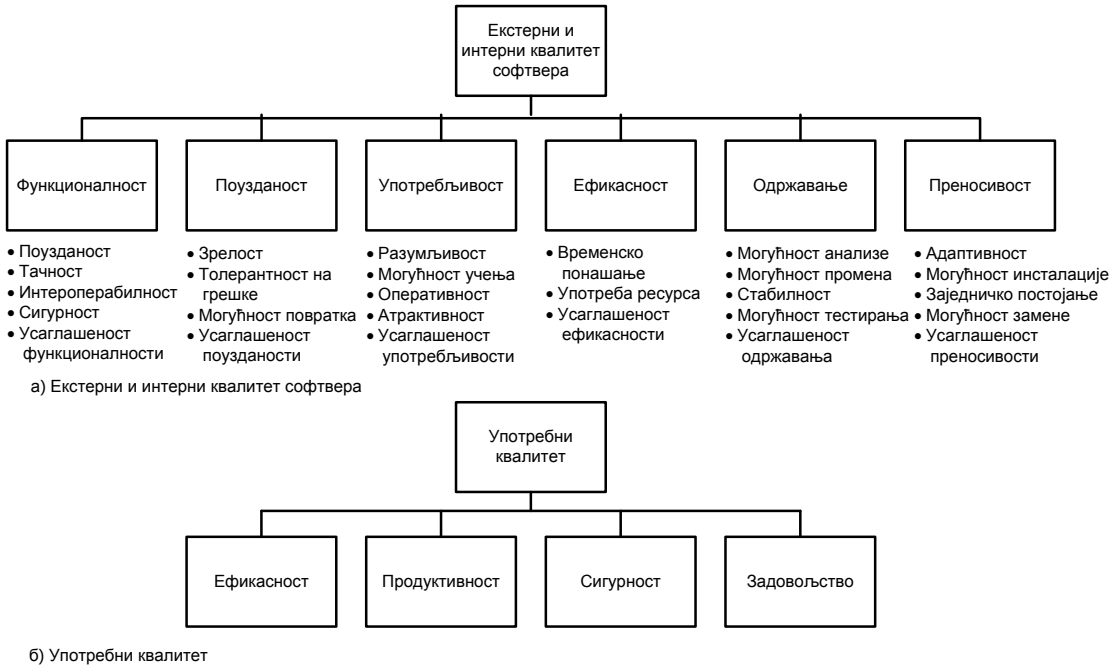
У складу с деловима стандарда, у наставку ће бити приказан модел квалитета софтвера према ISO/IEC 9126-1 стандарду. Након тога ће бити

дефинисан екстерни квалитет софтвера и екстерне софтверске метрике према ISO/IEC 9126-2 стандарду квалитета софтвера. На сличан начин ће бити дефинисан интерни квалитет софтвера и интерне софтверске метрике према ISO/IEC 9126-3 стандарду квалитета, као и употребни квалитет софтвера и софтверске метрике употребног квалитета према ISO/IEC 9126-4 стандарду квалитета софтвера.

3.2.1. Модел квалитета софтвера према ISO/IEC 9126 стандарду

Модел квалитета софтвера представља основу за утврђивање квалитета софтверског производа. У том смислу, у оквиру модела квалитета софтвера извршена је спецификација атрибута (односно карактеристика) квалитета софтвера које софтверски систем треба да задовољи. Свака карактеристика квалитета се у наставку разлаже на саставне делове, односно подкарактеристике квалитета. Подкарактеристике квалитета описују дати атрибут квалитета софтвера: оне су комплементарне и детерминишу посматрани атрибут квалитета софтвера. Спецификацијом карактеристика и подкарактеристика квалитета софтвера идентификују се кључни атрибути квалитета софтвера [Pressman10].

На основу дефинисаног модела квалитета софтвера могуће је извршити евалуацију интерног квалитета софтвера, екстерног квалитета софтвера и употребног квалитета софтвера. Међутим, моделом квалитета се не дефинише начин евалуације квалитета софтвера. Другим речима, моделом се не врши даља декомпозиција подкарактеристика квалитета софтвера [ISO9126v1]. Због тога је потребно одредити начин евалуације квалитета софтвера, што је дефинисано у оквиру других делова стандарда. На наредној слици (Слика 18) приказан је модел квалитета софтвера који је дефинисан ISO/IEC 9126 стандардом квалитета софтвера.



Слика 18. Модел квалитета софтвера према ISO/IEC 9126 стандарду [ISO9126v1]

Са слике се уочавају следећи атрибути квалитета софтвера [ISO9126v1]:

- Функционалност - Означава степен у којем софтверски систем задовољава исказане потребе [Pressman10]. Функционалност је одређена следећим подкарактеристикама: Поузданост, Тачност, Интероперабилност, Сигурност и Усаглашеност функционалности.
- Поузданост - У оквиру овог атрибута квалитета дефинише се временски период у којем је софтверски систем доступан за коришћење [Pressman10]. Поузданост је одређена следећим подкарактеристикама: Зрелост, Толерантност на грешке, Могућност повратка и Усаглашеност поузданости.
- Употребљивост - Употребљивост софтверског система се може посматрати из различитих перспектива (нпр. из перспективе крајњих корисника система, менаџера и инжењера за развој софтвера) [Abrano3]. Може се дефинисати као способност да корисник разуме, научи и користи софтверски систем [ISO9126v1]. Употребљивост је одређена следећим подкарактеристикама: Разумљивост, Могућност учења, Оперативност, Атрактивност и Усаглашеност употребљивости.

- Ефикасност - Приликом рада софтверски систем користи одређене ресурсе (нпр. време, меморију, процесор и др.). Ефикасност подразумева оптимално коришћење расположивих системских ресурса [Pressman10]. Ефикасност је одређена следећим подкарактеристикама: Временско понашање, Употреба ресурса и Усаглашеност ефикасности.
- Одржавање - У процесу развоја софтвера примењују се методе развоја софтвера коју су одређене моделом, стратегијом и активностима процеса развоја. Као резултат процеса развоја добија се софтверски систем кога је потребно одржавати, односно на једноставан начин прилагодити насталим променама у окружењу и захтевима корисника. Одржавање је одређено следећим подкарактеристикама: Могућност анализе, Могућност промена, Стабилност, Могућност тестирања и Усаглашеност одржавања.
- Преносивост - Као резултат процеса развоја добија се софтверски производ који се извршава у одређеном окружењу. Преносивост дефинише способност преноса софтверског система из једног окружења у друго окружење [ISO9126v1]. Преносивост је одређена следећим подкарактеристикама: Адаптивност, Могућност инсталације, Заједничко постојање, Могућност замене и Усаглашеност преносивости.

Из изложеног се може закључити да је модел квалитета софтвера сложен концепт који садржи велики број атрибута (тј. карактеристика) квалитета софтвера [Henningsson02]. Такође се може приметити да свака карактеристика поседује подкарактеристику која се односи на усаглашеност (функционалности, поузданости, употребљивости, ефикасности, одржавања и преносивости). У том смислу се дефинише способност софтверског производа да се придржава стандарда, конвенција, закона и других правила која се непосредно односе на посматрану карактеристику [Abran03] [ISO9126v1]. Дефинисане карактеристике су применљиве на сваку врсту

софтвера и на тај начин обезбеђују конзистентну терминологију за дефинисање квалитета софтверског производа, као и радни оквир (енг. framework) за специфицирање захтева за квалитетом софтвера.

С друге стране, за употребни квалитет је дефинисан посебан модел квалитета који садржи следеће карактеристике: Ефективност, Продуктивност, Сигурност и Задовољство [ISO9126v1][Seffaho6]. За разлику од интерног и екстерног модела квалитета употребни квалитет је дефинисан само на нивоу атрибута квалитета (тј. карактеристика). Употребни квалитет условљен је екстерним квалитетом софтвера који је, с друге стране, условљен интерним квалитетом софтвера [ISO9126v1][Lethbridge05].

3.2.2. Екстерне софтверске метрике према ISO/IEC 9126 стандарду

У претходном одељку представљен је ISO/IEC 9126-1 стандард који дефинише модел квалитета софтвера. Моделом квалитета специфицирано је шест атрибута квалитета који се могу користити за мерење екстерног и интерног квалитета софтвера. С друге стране, моделом квалитета се не даје спецификација начина на који треба извршити мерење вредности - он само дефинише карактеристике квалитета и разрађује их до нивоа подкарактеристика.

Екстерне софтверске метрике користе се за мерење екстерног квалитета софтвера. У том смислу ISO/IEC 9126-2 стандард треба схватити као надоградњу на модел квалитета софтвера који је дефинисан ISO/IEC 9126-1 стандардом. Екстерне софтверске метрике треба користити у комбинацији с моделом квалитета софтвера: оне специфицирају начин утврђивања нивоа екстерног квалитета.

У оквиру овог дела стандарда даје се опис софтверских метрика, при чему се метрике везују за карактеристике/подкарактеристике квалитета софтверског система. Такође, наводи се и примери примене метрика у току процеса развоја софтвера [ISO9126v2]. У том смислу су за сваки атрибут

квалитета софтверског система дефинисане метрике: дат је назив метрике, описана њена сврха, начин коришћења, формула, интерпретација измерених вредности, коришћена мерна скала, тип мере, као и корисници који користе резултате мерења.

Софтверске метрике ће детаљније бити описане у посебном одељку које се бави мерењима и софтверским метрикама.

3.2.3. Интерне софтверске метрике према ISO/IEC 9126 стандарду

Као што је раније напоменуто, стандардом ISO/IEC 9126-1 је прописан модел квалитета. У том смислу уочене су карактеристике квалитета софтвера које се могу користити за утврђивање екстерног и интерног квалитета софтвера. Моделом квалитета није дата спецификација начина на који треба извршити мерење вредности - он само дефинише карактеристике квалитета софтвера, при чему су за сваку карактеристику дефинисане подкарактеристике квалитета софтвера.

Интерним софтверским метрикама се мери интерни квалитет софтвера. У том смислу ISO/IEC 9126-3 део стандарда се наслања на модел квалитета софтвера који је дефинисан ISO/IEC 9126-1 делом стандарда. Оне су, пре свега, усмерене ка процесу развоја софтвера и омогућавају мерење унутрашњих особина квалитета софтверског производа [ISO9126v3]. Узимајући у обзир дефиницију екстерних софтверских метрика из претходног одељка може се рећи да постоји непосредан однос између интерних и екстерних софтверских метрика. Интерним софтверским метрикама се кроз процес развоја софтвера обезбеђују предуслови за задовољење екстерних софтверских метрика [ISO9126v3]. Интерне и екстерне софтверске метрике су комплементарне. Интерним софтверским метрикама се, такође, обезбеђују предуслови за задовољење употребног квалитета софтвера [ISO9126v3]. Интерне софтверске метрике су приказане на исти начин као и у оквиру ISO/IEC 9126-2 делу стандарда који описује екстерне софтверске метрике.

Софтверске метрике ће детаљније бити описане у посебном одељку који се бави мерењима и софтверским метрикама.

3.2.4. Метрике употребног квалитета према ISO/IEC 9126 стандарду

У оквиру ISO/IEC 9126-1 стандарда дефинисан је модел употребног квалитета софтвера који садржи следеће карактеристике: Ефективност, Продуктивност, Сигурност и Задовољство. Дефинисаним моделом квалитета се не врши даља разрада карактеристика, већ се у оквиру ISO/IEC 9126-4 стандарда дефинишу софтверске метрике за утврђивање употребног квалитата софтвера.

Софтверске метрике употребног квалитета су приказане на исти начин као и у оквиру ISO/IEC 9126-2 делу стандарда који описује екстерне софтверске метрике, односно ISO/IEC 9126-3 делу стандарда који описује интерне софтверске метрике.

На основу изложеног може се закључити да је неопходан услов за задовољење употребног квалитета софтвера да екстерни квалитет софтвера буде на одговарајућем нивоу. С друге стране, неопходан услов за задовољење екстерног квалитета софтвера је да интерни квалитет софтвера буде на одговарајућем нивоу. На наредној слици (Слика 19) је приказан софтверски систем и његови нивои квалитета:

- Модел екстерног и интерног квалитета софтвера, односно модел употребног квалитета софтвера представља темељ за изградњу квалитета софтвера. Ови модели дефинисани су ISO/IEC 9126-1 стандардом квалитета софтвера.
- На нивоу о (Но) налази се интерни квалитет софтверског система. Њиме се утврђују интерне карактеристике софтверског производа. Интерне софтверске метрике за утврђивање интерног квалитета софтвера дефинисане су ISO/IEC 9126-3 стандардом квалитета софтвера. Интерни квалитет софтвера представља **неопходан услов за утврђивање екстерног квалитета софтвера**.

- На нивоу 1 (H1) налази се екстерни квалитет софтверског система. Њиме се утврђују екстерне карактеристике софтверског производа. Екстерне софтверске метрике за утврђивање екстерног квалитета софтвера дефинисане су ISO/IEC 9126-2 стандардом квалитета софтвера. Екстерни квалитет софтвера представља **неопходан услов за утврђивање употребног квалитета софтвера**.
- На нивоу 2 (H2) налази се употребни квалитет софтверског система. На тај начин се мери употребљивост софтверског система. Софтверске метрике за утврђивање употребног квалитета софтвера дефинисане су ISO/IEC 9126-4 стандардом квалитета софтвера. Да би употребни квалитет софтвера био задовољен морају бити задовољени интерни квалитет софтвера и екстерни квалитет софтвера.

Слика 19. Нивои квалитета софтвера према ISO/IEC 9126 стандарду квалитета софтвера

Стога се може рећи да су интерни квалитет, екстерни квалитет и употребни квалитет софтвера повезани, међусобно условљени, комплементатни и да чине једну целину. На тај начин се обезбеђује свеобухватан поглед на квалитет софтверског система, из различитих перспектива.

3.3. ISO/IEC 14598 стандард квалитета софтвера

У претходном одељку приказан је ISO/IEC 9126 стандард квалитета софтвера. Овим стандардом дефинисан је модел квалитета за утврђивање екстерног

квалитета софтвера и интерног квалитета софтвера. Стандардом су дефинисане карактеристике квалитета софтвера (тј. атрибути квалитета софтвера), као и подкарактеристике квалитета софтвера које су непосредно повезане с атрибутима квалитета софтвера. У том смислу, за сваки атрибут квалитета дефинисане су софтверске метрике за мерење екстерног квалитета и интерног квалитета софтвера. Стандард такође специфицира модел употребног квалитета софтвера, као и софтверске метрике за мерење употребног квалитета софтвера.

Модел екстерног и интерног квалитета софтвера дефинисан је ISO/IEC 9126-1 делом стандарда квалитета. Овај део стандарда такође дефинише употребни квалитет софтвера. Екстерне софтверске метрике дефинисане су ISO/IEC 9126-2 делом стандарда квалитета, док су интерне софтверске метрике дефинисане ISO/IEC 9126-3 делом стандарда квалитета. Софтверске метрике употребног квалитета дефинисане су ISO/IEC 9126-4 делом стандарда квалитета софтвера.

Стандардом ISO/IEC 14598 дефинише се начин евалуације квалитета софтвера при чему се, као основа за евалуацију квалитета, користи ISO/IEC 9126 стандард квалитета софтвера. Овај стандард се може применити за евалуацију квалитета сваког софтверског производа. У том смислу овај стандард обухвата следеће делове [ISO14598v1]:

- ISO/IEC 14598-1:1999 Information technology - Software product evaluation - Part 1: General overview - У оквиру првог дела стандарда дат је уопштени опис стандарда квалитета, описани су његови саставни делови и веза с другим стандардима.
- ISO/IEC 14598-2:2000 Software engineering - Product evaluation - Part 2: Planning and management - У оквиру другог дела стандарда дата су упутства и смернице које се користе у процесу планирања и управљања евалуацијом квалитета софтвера.

- ISO/IEC 14598-3:2000 Software engineering - Product evaluation - Part 3: Process for developers - У оквиру трећег дела стандарда описан је поступак евалуације намењен инжењерима за развој софтвера (пројектантима).
- ISO/IEC 14598-4:1999 Software engineering - Product evaluation - Part 4: Process for acquirers - У оквиру четвртог дела стандарда описан је поступак евалуације намењен онима који врше набавку софтвера.
- ISO/IEC 14598-5:1998 Information technology - Software product evaluation - Part 5: Process for evaluators - У оквиру петог дела стандарда описан је поступак евалуације намењен независним оцењивачима.
- ISO/IEC 14598-6:2001 Software engineering - Product evaluation - Part 6: Documentation of evaluation modules - У оквиру шестог дела стандарда дата су упутства и смернице за документовање модула евалуације.

У складу с тиме у наставку ће бити приказани делови ISO/IEC 14598 стандарда за евалуацију квалитета софтвера.

3.3.1. Општа упутства за примену стандарда (ISO/IEC 14598-1)

У оквиру ISO/IEC 14598-1 дела стандарда квалитета дата су општа упутства за примену стандарда. Описани су делови ISO/IEC 14598 стандарда (стандард садржи шест делова који су наведени у претходном одељку), дефинисани су термини и напомене у вези с применом стандарда.

У том смислу се може рећи да стандард дефинише смернице за евалуацију софтверског производа са различитих нивоа [ISO14598v1]:

- Кроз процес непосредног развоја или унапређења софтверског производа, што је дефинисано ISO/IEC 14598-3 делом стандарда квалитета.
- Кроз процес набавке софтверског производа, што је дефинисано ISO/IEC 14598-4 делом стандарда квалитета.

- Кроз процес евалуације софтверског производа од стране независних оцењивача, што је дефинисано ISO/IEC 14598-5 делом стандарда квалитета.

Стандард ISO/IEC 14598 је у непосредној вези са стандардом ISO/IEC 9126. Наиме, приликом евалуације софтвера користи се ISO/IEC 9126 стандард квалитета софтвера. Овим стандардом је дефинисан модел екстерног, интерног и употребног квалитета (у оквиру ISO/IEC 9126-1 дела стандарда), односно дефинисане су карактеристике и подкарактеристике квалитета софтвера. Стандардом су такође дефинисане екстерне софтверске метрике (ISO/IEC 9126-2 део стандарда), интерне софтверске метрике (ISO/IEC 9126-3 део стандарда) и софтверске метрике за мерење употребног квалитета (ISO/IEC 9126-4 део стандарда).

У оквиру ISO/IEC 14598-1 дела стандарда дате су смернице за евалуацију квалитета софтвера. У том смислу наводи се процес евалуације квалитета софтвера који обухвата следеће фазе [ISO14598v1]:

1. Дефинисање захтева за евалуацију квалитета: одређује се сврха евалуације квалитета, дефинишу се типови производа и специфицира модел квалитета софтвера. За спецификацију модела квалитета софтвера користи се ISO/IEC 9126-1 стандард квалитета.
2. Спецификација евалуације квалитета: врши се спецификација критеријума за евалуацију и избор софтверских метрика. Приликом избора софтверских метрика користи се ISO/IEC 9126-2 стандард (за избор екстерних софтверских метрика) и ISO/IEC 9126-3 стандард (за избор интерних софтверских метрика).
3. Пројектовање евалуације квалитета: врши се израда плана евалуације квалитета софтвера, у складу с претходно дефинисаним захтевом и спецификацијом евалуације квалитета софтвера.
4. Извршавање евалуације квалитета: врши се непосредно спровођење претходно дефинисаног плана; извршавају се мерења и врши њихово

упоређивање с дефинисаним критеријумима након чега се врши оцена добијених резултата.

3.3.2. Планирање и управљање (ISO/IEC 14598-2)

У претходном одељку описан је уводни део ISO/IEC 14598-1 стандарда квалитета у оквиру кога су, поред самог описа стандарда, наведене и фазе у процесу евалуације квалитета. У том смислу су у оквиру ISO/IEC 14598-2 дела стандарда квалитета дате смернице за планирање и управљање процесом евалуације квалитета софтвера [ISO14598v2].

Овај део стандарда намењен је:

- корисницима који управљају технологијом за евалуацију,
- корисницима који пружају подршку евалуацији софтвера и
- корисницима који управљају организацијама за развој софтвера.

У том смислу се може рећи да ISO/IEC 14598-2 део стандарда квалитета софтвера обезбеђује смернице за евалуацију квалитета софтвера како на нивоу појединачног пројекта тако и на нивоу организације.

Стандардом се наводе концепти управљања евалуацијом, као и захтеви и препоруке за управљање процесом евалуације квалитета софтвера. На тај начин се дају смернице за спровођење процеса евалуације квалитета софтвера. Битно је напоменути да стандард предвиђа и прикупљање повратних информација о процесу евалуације квалитета што последично доводи до побољшања квалитета процеса евалуације.

3.3.3. Поступак евалуације при процесу развоја софтвера (ISO/IEC 14598-3)

У оквиру ISO/IEC 14598-3 дела стандарда описан је поступак евалуације намењен пројектантима који непосредно раде на развоју софтвера. У том смислу разматрају се активности евалуације квалитета софтвера у процесу самог развоја софтвера. На тај начин дефинишу се смернице за евалуацију

квалитета софтвера што се може применити приликом развоја новог или унапређења постојећег софтверског производа [ISO14598v3].

Овај део стандарда намењен је менаџерима пројекта, пројектантима софтвера, инжењерима који се баве осигурањем и контролом квалитета софтвера и инжењерима који одржавају софтвер [ISO14598v3]. Другим речима, стандард се примењује на нивоу пројекта и треба га користити у комбинацији с ISO/IEC 14598-1 (који даје опште смернице за примену стандарда евалуације квалитета софтвера), ISO/IEC 14598-2 (који обезбеђује упутства за евалуацију квалитета софтвера на нивоу пројекта) и ISO/IEC 14598-6 (који је усмерен на документовање модула евалуације квалитета).

Стандардом су описани концепти евалуације квалитета софтвера: потребе корисника, екстерни атрибути квалитета, интерни атрибути квалитета, индикатори квалитета и процес евалуације квалитета [ISO14598v3]. Такође, описани су и захтеви за процес евалуације квалитета софтвера. У том смислу се врши разрада фаза процеса евалуације квалитета софтвера која је дата ISO/IEC 14598-1 делом стандарда квалитета.

Значај ISO/IEC 14598-3 дела стандарда је што евалуацију квалитета софтвера посматра из перспективе оних који непосредно учествују у процесу развоја софтвера. На тај начин се омогућава да евалуација квалитета софтвера буде саставни део процеса развоја софтвера: омогућава се евалуација квалитета сваког инкремента у процесу развоја софтвера, као и евалуација финалног софтверског производа, у складу с процесом његовог развоја.

3.3.4. Поступак евалуације при процесу набавке софтвера (ISO/IEC 14598-4)

Поступак дефинисан ISO/IEC 14598-3 делом стандарда се може применити уколико се софтвер непосредно развија у посматраној организацији. Међутим, организација до софтвера може доћи и на други начин, нпр. набавком софтверског производа. У том случају је, такође, потребно извршити евалуацију квалитета производа и тада се примењује ISO/IEC

14598-4 део стандарда квалитета којим се дефинише поступак евалуације у процесу набавке.

Стандард се може успешно применити приликом набавке готових софтверских производа, софтверских производа које треба прилагодити организацији, односно софтверских производа које треба изменити [ISO14598v4]. Стандард се примењује у комбинацији с ISO/IEC 14598-1 (који даје опште смернице за примену стандарда евалуације квалитета софтвера), ISO/IEC 14598-2 (који обезбеђује упутства за евалуацију квалитета софтвера на нивоу организације) и ISO/IEC 14598-6 (који је усмерен на документовање модула евалуације квалитета). При томе, стандард користи модел квалитета који је дефинисан ISO/IEC 9126-1 стандардом.

У том смислу најпре се даје уопштена разматрања која се односе на поступак евалуације квалитета софтвера при процесу набавке софтвера. Затим се врши разрада фаза које су дефинисане ISO/IEC 14598-1 стандардом квалитета. Најпре се даје спецификација евалуације квалитета приликом набавке готових софтверских производа. Након тога се даје спецификација евалуације квалитета софтверских производа које треба прилагодити организацији, односно софтверских производа које треба изменити.

Овај део стандарда намењен је менаџерима пројекта, систем инжењерима, особљу које се бави развојем и одржавањем софтверских система, као и крајњим корисницима који планирају да набаве софтверски производ, односно добављачима софтверских производа [ISO14598v4].

Значај ISO/IEC 14598-4 стандарда је што дефинише поступак евалуације при процесу набавке софтвера. У том смислу се може рећи да су стандарди ISO/IEC 14598-3 и ISO/IEC 14598-4 комплементарни: први дефинише поступак евалуације квалитета при процесу развоја софтвера док други дефинише поступак евалуације квалитета при процесу набавке софтвера. На тај начин се омогућава поглед на поступак евалуације из различитих перспектива: из перспективе организације која непосредно развија софтвер

и спроводи активности животног циклуса развоја софтвера, односно организације која планира да набави софтверски производ.

3.3.5. Поступак за оцењиваче (ISO/IEC 14598-5)

Стандард ISO/IEC 14598-3 се може користити у поступку евалуације квалитета уколико се софтверски систем непосредно развија (или унапређује) у оквиру организације. С друге стране, уколико се врши набавка софтверског система, у поступку евалуације квалитета се може користити стандард ISO/IEC 14598-4. Ови стандарди усмерени су на евалуацију квалитета софтвера од стране запослених у организацији која развија, унапређује и набавља софтвер. Међутим, понекад је потребно извршити независну евалуацију квалитета (нпр. за потребе трећег лица или од стране трећег лица). У ту сврху се у процесу евалуације квалитета користи стандард ISO/IEC 14598-5 [ISO14598v5].

Стандард се примењује у комбинацији с ISO/IEC 14598-1 (који даје опште смернице за примену стандарда евалуације квалитета софтвера) и ISO/IEC 14598-6 (који је усмерен на документовање модула евалуације квалитета). При томе, стандард користи модел квалитета који је дефинисан ISO/IEC 9126-1 стандардом.

Стандард најпре даје уопштена разматрања која се односе на поступак евалуације квалитета софтвера од стране оцењивача. У том смислу се разматрају полазна основа и карактеристике процеса евалуације од стране оцењивача: поновљивост, репродуктивност, непристрасност и објективност. Затим се врши разрада фаза које су дефинисане ISO/IEC 14598-1 стандардом квалитета. На крају се наводи нормативни шаблон за израду извештаја оцењивача о извршеној евалуацији квалитета софтвера.

Овај део стандарда намењен је лабораторијама које се баве евалуацијом квалитета софтвера, добављачима софтвера, организацијама које врше набавку софтвера, корисницима софтвера и сертификационим телима [ISO14598v5].

Значај ISO/IEC 14598-5 стандарда је што дефинише поступак евалуације квалитета од стране независних оцењивача. У том смислу се може рећи да је овај стандард комплементаран са ISO/IEC 14598-3 и ISO/IEC 14598-4 стандардима. Наиме, ISO/IEC 14598-3 и ISO/IEC 14598-4 стандарди посматрају поступак евалуације квалитета софтвера од стране организације која непосредно учествује у процесу развоја, унапређења или набавке софтверског производа, док стандард ISO/IEC 14598-5 посматра поступак евалуације квалитета софтвера од стране независних оцењивача.

3.3.6. Документација модула евалуације (ISO/IEC 14598-6)

У претходним одељцима објашњени су делови ISO/IEC 14598 стандарда за евалуацију квалитета. Примећује се да постоји више делова стандарда који дају поглед на софтверски систем из различитих перспектива. Такође, овај стандард је непосредно повезан с другим стандардима, пре свега с ISO/IEC 9126 стандардом квалитета. У том смислу неопходно је направити документацију за поступак евалуације квалитета софтвера што је дефинисано ISO/IEC 14598-6 делом стандарда квалитета [ISO14598v6]. Увидом у ISO/IEC 14598 серију стандарда може се приметити да је сваки део стандарда непосредно повезан с ISO/IEC 14598-6 стандардом.

Документација модула евалуације има шест делова [ISO14598v6]:

- EM0 - Представља увод у модул евалуације квалитета
- EM1 - Дефинише примењивост модула евалуације квалитета
- EM2 - У оквиру овог дела наводе се референце које су значајне за поступак евалуације квалитета
- EM3 - Садржи дефиниције које се користе у процесу евалуације квалитета
- EM4 - У оквиру овог дела наводи се спецификација улаза у процес евалуације
- EM5 - Садржи упутства о начину тумачења измерених вредности

Поред тога, постоји и необавезни анекс EMA који садржи упутства везана за примену модула евалуације [ISO14598v6].

3.4. ISO/IEC 25000 стандард квалитета софтвера

У претходним одељцима приказани су ISO/IEC 9126 и ISO/IEC 14598 стандарди квалитета софтвера. Стандардом ISO/IEC 9126 дефинисан је модел екстерног, интерног и употребног квалитета (у оквиру ISO/IEC 9126-1 дела стандарда), чиме су одређене карактеристике и подкарактеристике квалитета софтверског система. На основу тога се дефинишу софтверске метрике за мерење екстерног квалитета (у оквиру ISO/IEC 9126-2 дела стандарда), интерног квалитета (у оквиру ISO/IEC 9126-3 дела стандарда) и употребног квалитета (у оквиру ISO/IEC 9126-4 дела стандарда). На основу тога је закључено да су интерни квалитет, екстерни квалитет и употребни квалитет софтвера повезани, међусобно условљени, комплементарни и да чине једну целину. Стога се може рећи да се ISO/IEC 9126 стандардом квалитета обезбеђује свеобухватан поглед на квалитет софтверског система, из различитих перспектива. Наредна табела (Табела 6) даје сумарни преглед делова ISO/IEC 9126 стандарда квалитета софтвера.

Табела 6. Делови ISO/IEC 9126 стандарда квалитета софтвера

Карактеристика	**Делови ISO/IEC 9126 стандарда квалитета софтвера**			
	ISO/IEC 9126-1	**ISO/IEC 9126-2**	**ISO/IEC 9126-3**	**ISO/IEC 9126-4**
Намена	Дефинише модел екстерног, интерног и употребног квалитета (атрибуте квалитета софтвера).	Дефинише екстерне софтверске метрике за мерење екстерног квалитета софтвера.	Дефинише интерне софтверске метрике за мерење интерног квалитета софтвера.	Дефинише софтверске метрике употребног квалитета.

Карактеристика	Делови ISO/IEC 9126 стандарда квалитета софтвера			
	ISO/IEC 9126-1	ISO/IEC 9126-2	ISO/IEC 9126-3	ISO/IEC 9126-4
Веза са другим стандардима	ISO/IEC 9126 (остали делови), ISO/IEC 14598 (сви делови)	ISO/IEC 9126-1, ISO/IEC 14598-2	ISO/IEC 9126-1, ISO/IEC 14598-2	ISO/IEC 9126-1

С друге стране, стандардом ISO/IEC 14598 дефинисан је начин евалуације квалитета софтвера. У том смислу су дефинисане фазе које се користе у поступку евалуације (дефинисане су ISO/IEC 14598-1 делом стандарда), смернице за планирање и управљање у поступку евалуације (дефинисане су ISO/IEC 14598-2 делом стандарда), поступак евалуације при процесу развоја софтвера (дефинисан је ISO/IEC 14598-3 делом стандарда), поступак евалуације при процесу набавке софтвера (дефинисан је ISO/IEC 14598-4 делом стандарда), поступак евалуације за независне оцењиваче (дефинисан је ISO/IEC 14598-5 делом стандарда), као и документација модула у поступку евалуације (дефинисана је у оквиру ISO/IEC 14598-6 дела стандарда). Стандард ISO/IEC 14598 обезбеђује поглед на поступак евалуације квалитета софтвера из различитих перспектива: организације која развија (или унапређује) софтверски систем, организације која набавља софтверски систем и организације која спроводи/захтева независан поступак евалуације квалитета софтвера. Наредна табела (Табела 7) даје сумарни преглед делова ISO/IEC 14598 стандарда квалитета софтвера.

Табела 7. Делови ISO/IEC 14598 стандарда квалитета софтвера

Карактеристика	Делови ISO/IEC 14598 стандарда квалитета софтвера					
	ISO/IEC 14598-1	ISO/IEC 14598-2	ISO/IEC 14598-3	ISO/IEC 14598-4	ISO/IEC 14598-5	ISO/IEC 14598-6

Карактеристика	**Делови ISO/IEC 14598 стандарда квалитета софтвера**					
	ISO/IEC 14598-1	**ISO/IEC 14598-2**	**ISO/IEC 14598-3**	**ISO/IEC 14598-4**	**ISO/IEC 14598-5**	**ISO/IEC 14598-6**
Намена	Уводне напомене, фазе у поступку евалуације квалитета софтвера.	Дефинише смернице за планирање и управљање у поступку евалуације квалитета софтвера.	Дефинише поступак евалуације квалитета при процесу развоја (или унапређења) софтвера.	Дефинише поступак евалуације квалитета при процесу набавке софтвера.	Дефинише поступак евалуације квалитета софтвера од стране независних оцењивача.	Дефинише документа модула у поступку евалуације квалитета софтвера.
Веза са другим стандардима	ISO/IEC 9126-1, ISO/IEC 14598 (остали делови)	ISO/IEC 9126-1, 9126-2, 9126-3, ISO/IEC 14598-1, 14598-5, 14598-6	ISO/IEC 9126-1, ISO/IEC 14598-1, 14598-2, 14598-6	ISO/IEC 9126-1, ISO/IEC 14598-1, 14598-5	ISO/IEC 9126, ISO/IEC 14598-1, 14598-6	ISO/IEC 9126-1, ISO/IEC 14598 (остали делови)

Стандард ISO/IEC 14598 је у непосредној вези са стандардом ISO/IEC 9126. Наиме, приликом евалуације квалитета софтвера користи се ISO/IEC 9126 стандард квалитета софтвера. Другим речима, стандард ISO/IEC 14598 у поступку евалуације користи модел квалитета и софтверске метрике ISO/IEC 9126 стандарда за евалуацију квалитета софтвера. У том смислу се може рећи да су наведени стандарди комплементарни и да на јединствен начин посматрају квалитет софтвера. Ови стандарди се заједно примењују у поступку евалуације квалитета софтвера.

На тај начин успостављени су стандарди којима се експлицитно мери ниво квалитета софтверског система, независно од тога да ли је у питању процес развоја новог (или унапређења постојећег) софтверског система или процес

набавке софтверског система. Наведени стандарди имају следеће карактеристике [ISO25000]:

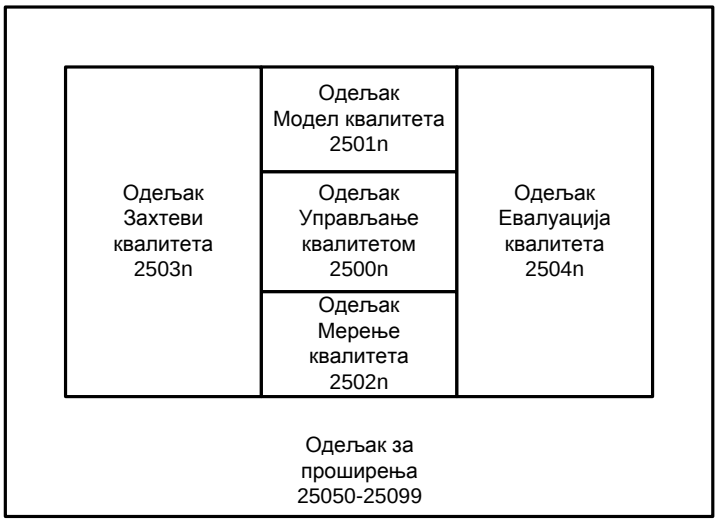
- Стандарди имају заједнички норматив
- Представљају скуп комплементарних стандарда који се користе приликом евалуације квалитета софтвера
- Независни животни циклус развоја стандарда квалитета довео је до њихове неконзистентности

У том смислу су Међународна организација за стандардизацију и Међународна електротехничка комисија приступиле развоју новог стандарда серије ISO/IEC 25000 [ISO25000] који обједињује и замењује стандарде ISO/IEC 9126 и ISO/IEC 14598. Стандардом се дефинишу захтеви за квалитет и евалуацију система и софтвера (енг. Systems and software Quality Requirements and Evaluation - SQuaRE). Стандард обухвата следеће одељке [ISO25000]:

- ISO/IEC 2500n - одељак Управљање квалитетом (енг. Quality Management Division): Дефинише скуп стандарда који се односе на управљање квалитетом. У оквиру овог одељка наводе се упутства за примену стандарда. Стандарди ове групе дефинишу основу и користе се у оквиру осталих стандарда.
- ISO/IEC 2501n - одељак Модел квалитета (енг. Quality Model Division): Дефинише скуп стандарда који се односе на модел квалитета.
- ISO/IEC 2502n - одељак Мерење квалитета (енг. Quality Measurement Division): Дефинише скуп стандарда који се односе на мерење квалитета.
- ISO/IEC 2503n - одељак Захтеви квалитета (енг. Quality Requirements Division): Дефинише скуп стандарда који се односе на захтеве квалитета.

- ISO/IEC 2504n - одељак Евалуација квалитета (енг. Quality Evaluation Division): Дефинише скуп стандарда који се односе на евалуацију квалитета.
- ISO/IEC 25050-25099 - одељак за проширења (енг. Extension Division): Представља резервисан простор за стандарде и техничке извештаје који су проширења ISO/IEC 25000 серије стандарда и који се могу користити у комбинацији са постојећим стандардима.

Структура ISO/IEC 25000 стандарда квалитета софтвера приказана је на наредној слици (Слика 20).



Слика 20. Структура ISO/IEC 25000 стандарда квалитета [ISO25000]

У наставку рада дат је приказ сваког одељка ISO/IEC 25000 стандарда квалитета софтвера. Најпре ће бити приказан одељак Управљање квалитетом. У оквиру њега је дефинисан речник појмова и дат је преглед целе серије стандарда, тако да представља полазну основу за проучавање других одељака стандарда. Након тога ће бити приказан одељак Модел квалитета у оквиру којег се даје спецификација модела квалитета софтверског производа и употребног квалитета. Одељак Мерење квалитета биће приказан након одељка Модел квалитета. Он садржи референтни модел мерења и упутство за примену модела квалитета, као и софтверске метрике квалитета софтверског производа (које се користе у процесу

евалуације екстерног и интерног квалитета софтвера), односно софтверске метрике употребног квалитета (које се користе у процесу евалуације употребног квалитета софтвера). Након тога ће бити приказан одељак Захтеви квалитета који је усмерен на спецификацију захтева квалитета софтверског производа. На крају ће бити дат приказ одељка Евалуација квалитета који дефинише фазе процеса евалуације квалитета софтвера.

3.4.1. Одељак Управљање квалитетом (ISO/IEC 2500n)

Одељак за управљање квалитетом садржи упутство за примену стандарда. Ово упутство дефинисано је ISO/IEC 25000 стандардом квалитета софтвера. У оквиру њега даје се увод у ISO/IEC 25000 серију стандарда, описује се структура стандарда и дају термини и дефиниције које се користе у оквиру осталих стандарда. Такође се приказује веза овог стандарда са другим стандардима. Посебан део стандарда даје преглед ISO/IEC 9126 и ISO/IEC 14598 стандарда квалитета софтвера, као и везу ових стандарда квалитета са ISO/IEC 25000 стандардом квалитета софтвера [ISO25000].

Одељак такође садржи и ISO/IEC 25001 стандард који се односи на планирање и управљање захтевима и евалуацијом квалитета софтвера. Стандардом су описани концепти управљања евалуацијом квалитета софтвера (с посебним нагласком на одговорности групе која врши евалуацију квалитета), као и препоруке које се односе на спецификацију захтева и евалуацију квалитета софтвера [ISO25001].

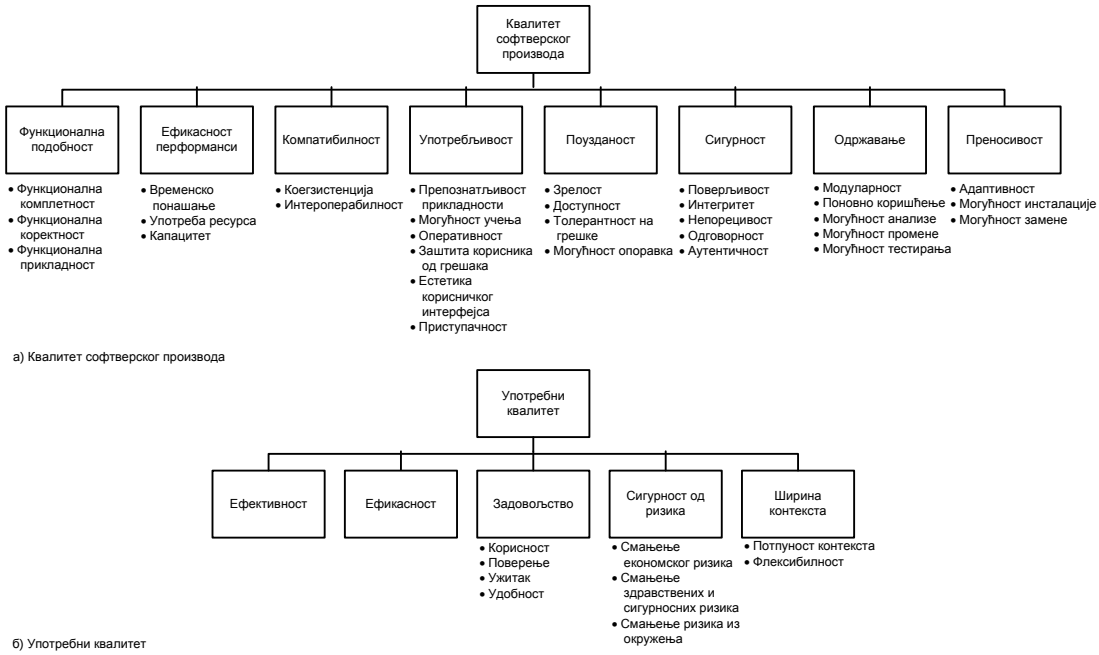
Стандарди из ове групе представљају основни градивни елемент за друге стандарде квалитета из серије ISO/IEC 25000.

3.4.2. Одељак Модел квалитета (ISO/IEC 2501n)

У оквиру одељка Модел квалитета дефинисан је стандард ISO/IEC 25010 који представља основу за утврђивање квалитета софтверског производа [ISO25010]. Слично као код ISO/IEC 9126 стандарда квалитета софтвера, стандардом ISO/IEC 25010 су дефинисане карактеристика квалитета софтвера које софтверски систем треба да задовољи. Свака карактеристика

квалитета је разложена на подкарактеристике квалитета, при чему подкарактеристике квалитета описују дати атрибут квалитета софтвера.

У оквиру стандарда дефинисана су два модела квалитета: модел квалитета софтверског производа и модел употребног квалитета. На основу дефинисаних модела квалитета софтвера могуће је извршити евалуацију интерног квалитета софтвера, екстерног квалитета софтвера и употребног квалитета софтвера. Начин евалуације квалитета софтвера дефинисан је у оквиру других делова стандарда. На наредној слици (Слика 21) приказани су модели квалитета софтвера који су дефинисани ISO/IEC 25010 стандардом квалитета софтвера.



Слика 21. Модел квалитета софтвера према ISO/IEC 25010 стандарду [ISO25010]

Примећује се другачији назив модела у односу на ISO/IEC 9126 стандард квалитета: у оквиру ISO/IEC 9126-1 стандарда модел има назив "Екстерни и интерни квалитет" док се у стандарду ISO/IEC 25010 модел назива "Квалитет софтверског производа". Без обзира на назив, модел је намењен за мерење екстерног и интерног квалитета софтверског производа [ISO25010]. Такође, примећује се да су нове карактеристике квалитета софтвера *Сигурност* и *Компатибилност*. Промењене су и подкарактеристике квалитета софтвера

како би се на прецизнији начин дефинисале карактеристике квалитета софтверског производа.

Модел употребног квалитета је такође промењен. Наиме, додата је нова карактеристика *Ширина контекста* која је усмерена ка дефинисању степена у којем се систем може ефикасно, ефективно, сигурно и са задовољством користити [ISO25010]. У оквиру стандарда ISO/IEC 9126-1 модел употребног квалитета је дефинисао само карактеристике и није се бавио њиховом даљом декомпозицијом. С друге стране, модел употребног квалитета у стандарду ISO/IEC 25010 разрађује карактеристике до нивоа подкарактеристика употребног квалитета.

У оквиру овог одељка налази се и стандард ISO/IEC 25012 којим се дефинише модел квалитета података [ISO25012].

3.4.3. Одељак Мерење квалитета (ISO/IEC 2502n)

У оквиру одељка Мерење квалитета налазе се стандарди квалитета којима се експлицитно дефинишу софтверске метрике за мерење екстерног квалитета софтвера, интерног квалитета софтвера и употребног квалитета софтвера. У том смислу одељак обухвата следеће стандарде:

- ISO/IEC 25020:2007 - У оквиру стандарда ISO/IEC 25020 дат је референтни модел мерења и упутство за примену модела квалитета који су дефинисани стандардима одељка ISO/IEC 2501n (одељак садржи модел квалитета софтверског производа, модел употребног квалитета и модел квалитета података) [ISO25020].
- ISO/IEC 25021:2012 - У оквиру стандарда ISO/IEC 25021:2012 дефинисан је почетни скуп елемената за мерење квалитета софтверског производа. Стандард садржи и упутства за њихов развој и верификацију [ISO25021].
- ISO/IEC DIS 25022 - Овај стандард је непосредно повезан са моделом употребног квалитета софтвера који је дефинисан ISO/IEC 25010

стандардом и специфицира софтверске метрике за евалуацију употребног квалитета софтвера. Стандард је у фази израде.

- ISO/IEC DIS 25023 - Овај стандард је непосредно повезан са моделом квалитета софтверског производа који је одређен ISO/IEC 25010 стандардом и садржи софтверске метрике за евалуацију екстерног и интерног квалитета софтвера. Стандард је у фази израде.
- ISO/IEC DIS 25024 Овај стандард је непосредно повезан са моделом квалитета података који је одређен ISO/IEC 25010 стандардом и дефинише софтверске метрике за евалуацију квалитета података. Стандард је у фази израде.

Одељак Мерење квалитета (ISO/IEC 2502n) је веома значајан и у непосредној је вези са одељком Модел квалитета (ISO/IEC 2501n). Стандарди одељка ISO/IEC 2502n разрађују моделе квалитета који су дефинисани одељком ISO/IEC 2501n, тј. дефинишу начин евалуације квалитета софтвера.

Такође се може рећи да постоји аналогија са ISO/IEC 9126 стандардом квалитета који у оквиру ISO/IEC 9126-1 дела стандарда дефинише моделе квалитета софтвера, док у оквиру ISO/IEC 9126-2, ISO/IEC 9126-3 и ISO/IEC 9126-4 делова стандарда дефинише софтверске метрике које се користе за евалуацију квалитета софтвера, у складу са дефинисаним моделима квалитета.

Важно је напоменути да стандарди одељка ISO/IEC 2502n нису у потпуности имплементирани (неки стандарди су још увек у фази израде). Међутим, ови стандарди свакако чине једну целину која дефинише моделе квалитета софтвера и софтверске метрике чиме се стварају предуслови за свеобухватну евалуацију квалитета софтвера.

3.4.4. Одељак Захтеви квалитета (ISO/IEC 2503n)

Данашњи софтверски системи су веома сложени и представљају резултат рада великог броја експерата. Развој великих софтверских система захтева доста ресурса (време, новац, радна снага итд.) па је стога потребно да се од

почетних фаза процеса развоја софтвера дефинишу захтеви које се односе на квалитет софтвера. У оквиру ISO/IEC 2503n одељка дефинисан је стандард квалитета ISO/IEC 25030 који је усмерен на спецификацију захтева за квалитетом софтвера [ISO25030].

У стандарду се разматрају модели квалитета софтвера који су дефинисани ISO/IEC 9126 стандардом и ISO/IEC 25010 стандардом, мере квалитета софтвера које су дефинисане ISO/IEC 25020 стандардом, као и њихова веза са захтевима за квалитетом софтвера. Узимајући у обзир да су моделима квалитета софтвера обухваћени екстерни квалитет, интерни квалитет и употребни квалитет софтвера, постоје следећи захтеви квалитета [ISO25030]:

- Захтеви употребног квалитета софтвера,
- Захтеви екстерног квалитета софтвера,
- Захтеви интерног квалитета софтвера.

Стандардом се, такође, дају препоруке за спецификацију захтева квалитета софтвера.

3.4.5. Одељак Евалуација квалитета (ISO/IEC 2504n)

У оквиру одељка Евалуација квалитета дефинисан је ISO/IEC 25040 стандард који дефинише фазе евалуације квалитета софтвера. Стандард се може користити приликом развоја новог (или унапређења постојећег) софтверског производа, приликом набавке производа или приликом независне евалуације од стране оцењивача. Одељак такође садржи ISO/IEC 25041 стандард који дефинише упутство за примену стандарда [ISO25041]. У том смислу се може рећи да стандарди овог одељка практично замењују ISO/IEC 14598-3, ISO/IEC 14598-4 и ISO/IEC 14598-5 стандарде квалитета софтвера.

Стандардом се дефинише процес евалуације квалитета софтвера који обухвата следеће фазе [ISO25040]:

1. Дефинисање захтева за евалуацију квалитета: одређује се сврха евалуације квалитета, дефинишу захтеви квалитета софтвера и одређује софтверски производ чију евалуацију квалитета треба извршити. Смернице за дефинисање захтева квалитета дате су стандардом ISO/IEC 25030, при чему као основу за спецификацију захтева квалитета треба користити модел квалитета који је дефинисан стандардом ISO/IEC 25010.
2. Спецификација евалуације квалитета: врши се спецификација мера квалитета и дефинисање критеријума одлучивања за процес евалуације. Приликом избора мера квалитета користе се стандарди из одељка ISO/IEC 2502n (у оквиру њега су дефинисани стандарди који садрже референтни модел мерења као и софтверске метрике за мерење употребног квалитета софтвера и квалитета софтверског производа).
3. Пројектовање евалуације квалитета: врши се израда плана евалуације квалитета софтвера, у складу са претходно утврђеним захтевима за евалуацију квалитета и спецификацијом евалуације квалитета софтвера.
4. Извршавање евалуације квалитета: врши се непосредно спровођење претходно дефинисаног плана евалуације; извршавају се мерења и врши примена претходно дефинисаних критеријума одлучивања за процес евалуације квалитета софтвера.
5. Завршетак евалуације квалитета: врши се преглед резултата евалуације квалитета и израда извештаја о извршеној евалуацији. На крају се организацији даје повратна информација о извршеној евалуацији квалитета софтвера.

Може се приметити да су прве четири фазе процеса евалуације сличне као и фазе процеса евалуације које су дефинисане ISO/IEC 14598-1 стандардом квалитета софтвера. Међутим, у оквиру ISO/IEC 25040 стандарда је експлицитно додата нова, пета фаза, која се односи на завршетак процеса

евалуације квалитета, израду извештаја и информисање организације о извршеној евалуацији квалитета софтвера. У том контексту се може рећи да је процес евалуације квалитета софтвера дефинисан стандардом ISO/IEC 25040 комплетнији и представља логички заокружену целину.

У наредној табели (Табела 8) дат је компаративни приказ ISO/IEC 9126, ISO/IEC 14598 и ISO/IEC 25000 стандарда квалитета софтвера.

Табела 8. Упоредни приказ ISO/IEC 9126, ISO/IEC 14598 и ISO/IEC 25000 стандарда квалитета софтвера

Карактеристика	**ISO/IEC 9126**	**ISO/IEC 14598**	**ISO/IEC 25000**
Намена стандарда	Обезбеђује свеобухватан поглед на квалитет софтверског система, из различитих перспектива.	Обезбеђује свеобухватни поглед на поступак евалуације квалитета софтвера из различитих перспектива.	Обезбеђује свеобухватан поглед на квалитет и поступак евалуације квалитета софтверског система, из различитих перспектива.
Делови стандарда	ISO/IEC 9126-1 (модел квалитета софтвера), ISO/IEC 9126-2 (екстерне софтверске метрике),	ISO/IEC 14598-1 (општа упутства), ISO/IEC 14598-2 (планирање и управљање), ISO/IEC 14598-3 (евалуација при	ISO/IEC 2500n (управљање квалитетом), ISO/IEC 2501n (модел квалитета), ISO/IEC 2502n (мерење

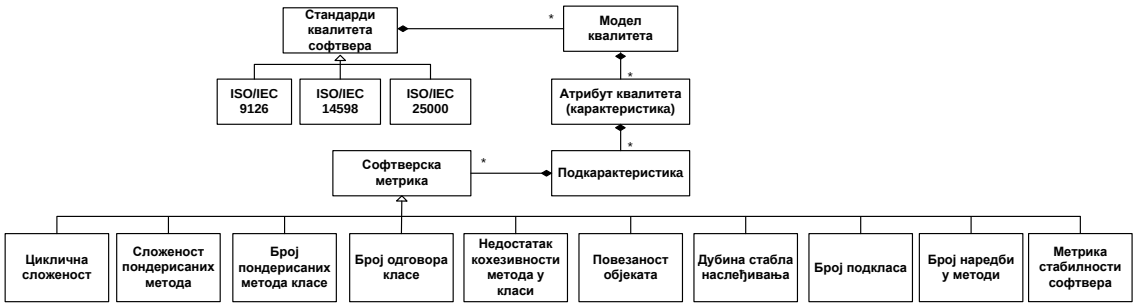
Карактеристика	**ISO/IEC 9126**	**ISO/IEC 14598**	**ISO/IEC 25000**
	ISO/IEC 9126-3 (интерне софтверске метрике), ISO/IEC 9126-4 (метрике употребног квалитета)	процесу развоју), ISO/IEC 14598-4 (евалуација при процесу набавке), ISO/IEC 14598-5 (поступак за оцењиваче), ISO/IEC 14598-6 (документација модула евалуације)	квалитета), ISO/IEC 2503n (захтеви квалитета), ISO/IEC 2504n (евалуација квалитета), ISO/IEC 25050-25099 (одељак за проширења)
Напомене у вези стандарда	- Комплементаран са стандардом ISO/IEC 14598 - Независни животни циклус развоја ISO/IEC 9126 и ISO/IEC 14598 довео је до њихове неконзистентности - Стандард ISO/IEC 9126 се налази у фази повлачења	- Комплементаран са стандардом ISO/IEC 9126 - Независни животни циклус развоја ISO/IEC 14598 и ISO/IEC 9126 довео је до њихове неконзистентности - Стандард ISO/IEC 14598 се налази у фази повлачења	- Обједињује, замењује и проширује стандарде квалитета софтвера ISO/IEC 9126 и ISO/IEC 14598 - Стандард ISO/IEC 25000 се налази у фази развоја

4. Софтверске метрике

У оквиру претходног поглавља представљени су стандарди квалитета софтвера. Стандардима квалитета софтвера даје се спецификација модела квалитета софтвера (један стандард квалитета може специфицирати један или више модела квалитета софтвера), тј. даје се спецификација карактеристика (или атрибута) и подкарактеристика квалитета софтвера.

За оперативно мерење карактеристика и подкарактеристика квалитета софтвера непосредно се користе софтверске метрике. С друге стране, на основу нивоа квалитета карактеристика и подкарактеристика може се утврдити ниво квалитета целог софтверског система. Другим речима, софтверским метрикама се посредно мери ниво квалитета целог софтверског система. Због тога је ово поглавље посвећено софтверским метрикама.

Најпре ће бити представљен значај мерења и софтверских метрика. У том смислу ће бити изложене различите дефиниције софтверских метрика. Након тога ће бити представљене најважније карактеристике које софтверске метрике као квантитативне мере требају да задовоље, као и веза софтверских метрика са стандардима квалитета софтвера. У оквиру посебног одељка биће дат преглед различитих класификација софтверских метрика. На крају поглавља ће бити дат опис често коришћених софтверских метрика за модел пројектовања софтвера: циклична сложеност, сложеност пондерисаних метода, број пондерисаних метода класе, број одговора класе, недостатак кохезивности метода у класи, повезаност објеката, дубина стабла наслеђивања, број подкласа, број наредби у методи и метрика стабилности софтвера. Концептуални приказ поглавља дат је на наредној слици (Слика 22).



Слика 22. Софтверске метрике и стандарди квалитета софтвера

4.1. Мерења и софтверске метрике

У најранијим периодима развоја цивилизације човек је, како би задовољио своје потребе, морао да се прилагоди природи или је природу прилагођавао својим потребама. У том смислу морао је да разуме законе и процесе који се дешавају у окружењу и пронађе начин да их формално искаже. Један од начина за формално исказивање законитости је коришћење математичких формализама. Математичким формализмима се, на високом нивоу апстракције, могу описати различите појаве у окружењу (нпр. у физици, медицини, грађевини, машинству, архитектури, електротехници). Разумевање окружења и формално исказивање проблема представља полазну основу за примену поступка мерења како би се извршила анализа проблема и предузеле активности за њихово отклањање. На тај начин се стварају предуслови за напредак у процесу развоја друштва и стварање бољих услова за живот текућих и нових генерација.

У том смислу се може рећи да мерења имају практичну примену у многим областима. Мерења се, с великим успехом, примењују и у софтверском инжењерству [Fentonoo]. У најопштијем случају мерење се може дефинисати као "поступак којим се квантитативно исказује вредност неког атрибута софтверског производа или процеса" [Pressmanıo]. Из дефиниције се примећује да се нагласак ставља на квантитативном утврђивању вредности. Мерењем се стварају предуслови за даљу анализу што треба да допринесе развоју и унапређењу посматраног софтверског производа и процеса развоја софтвера [Kitchenham96].

Са поступком мерења непосредно је повезана софтверска метрика. Софтверска метрика се, на одређени начин, односи на појединачну меру [Pressman10]. Постоје различите дефиниције софтверске метрике:

- "Метрика представља квантитативну меру до које систем, компонента или процес поседује посматрани атрибут." [ISO24765]
- "Метрика представља утврђен метод мерења и мерну скалу." [ISO14598v1]
- "Софтверска метрика је карактеристика софтверског система, документације система или процеса развоја система која се може објективно измерити." [Sommerville11]

Из наведених дефиниција се примећује да је метрика усмерена на атрибуте посматраног софтверског производа или софтверског процеса, да је формализована и квантитативно оријентисана. Због тога се софтверске метрике могу успешно применити у евалуацији квалитета софтвера и процеса животног циклуса развоја софтвера.

У наставку рада биће описане карактеристике софтверских метрика. С обзиром да се софтверске метрике могу користити у поступку евалуације квалитета софтвера биће објашњена њихова веза са стандардима квалитета софтвера. На крају поглавља дат је приказ често коришћених софтверских метрика.

4.2. Карактеристике софтверских метрика

Анализом софтверских метрика могу се добити информације о квалитету софтверског производа. У том смислу је могуће уочити неусаглашености са дефинисаним моделом квалитета софтвера и могу се предузети акције како би се усаглашености елиминисале. Међутим, да би се софтверска метрика могла користити она треба да задовољи одређене карактеристике [Pressman10]:

- Софтверска метрика треба да буде формализована и да поседује математичка својства.
- Софтверским метрикама се, као што је речено, мере карактеристике софтверског система. Уколико се посматрана карактеристика софтверског система повећава или смањује софтверска метрика би требала да прати промене карактеристике софтверског система (да се, у складу са променама, повећава или смањује).
- Софтверска метрика би, пре практичне примене, треба да буде емпиријски потврђена.

Поред тога, на основу раније поменутих дефиниција софтверских метрика, можемо рећи да софтверска метрика треба да буде објективна и да самим тим доприносе рационалном доношењу одлука о квалитету софтверског производа и софтверског процеса.

Софтверска метрика, такође, треба да буде проверљива и предвидљива. Уколико се софтверска метрика користи за мерење неке карактеристике посматраног софтверског система, понављање мерења вредности софтверске метрике за посматрану верзију софтверског система (понављање мерења без измена у софтверском систему) треба да резултира истом нумеричком вредношћу.

Софтверска метрика треба да буде независна од програмског језика у оквиру којег се врши имплементација софтверског система, величине софтверског система, врсте софтверског система, домена проблема [Pressman10], као и од примењеног процеса животног циклуса развоја софтвера.

Велики број софтверских метрика задовољава поменуте карактеристике. Међутим, неке софтверске метрике не задовољавају све поменуте карактеристике [Pressman10]. Штавише, у литератури се могу пронаћи критике софтверских метрика (нпр. постоје критике, расправе и покушаји унапређења софтверске метрике *Недостатак кохезивности метода у класи* које се могу пронаћи у радовима [Hitz96][Mayer99]).

При процесу утврђивања вредности софтверских метрика могу се користити различите мерне скале. У том смислу разликују се следеће скале мерења [Bourque14] [ISO24765]:

- Номинална скала (енг. Nominal Scale),
- Ординална скала (енг. Ordinal Scale),
- Интервална скала (енг. Interval Scale),
- Скала односа (енг. Ratio Scale).

Свака скала омогућава да се резултати мерења искажу на одговарајући начин. Различите софтверске метрике користе различите мерне скале и на тај начин омогућавају анализу квалитета софтверског система.

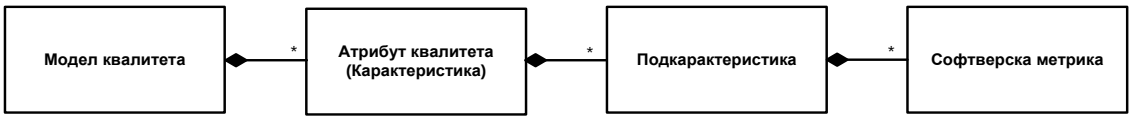
4.3. Софтверске метрике у стандардима квалитета софтвера

У оквиру претходног поглавља описани су ISO/IEC 9126, ISO/IEC 14598 и ISO/IEC 25000 стандарди квалитета софтвера. Стандардима се на свеобухватан начин разматра квалитет и евалуација квалитета софтверског производа:

- Стандардом ISO/IEC 9126 дефинисан је модел екстерног, интерног и употребног квалитета, чиме су одређене карактеристике и подкарактеристике квалитета софтверског система, као и софтверске метрике за мерење екстерног, интерног и употребног квалитета софтвера.
- Стандардом ISO/IEC 14598 дефинисан је начин евалуације квалитета софтвера: дефинисане су фазе које се користе у поступку евалуације, смернице за планирање и управљање у поступку евалуације као и поступак и документација евалуације квалитета софтвера.
- Наведени стандарди су комплементарни и могу се користити у процесу евалуације квалитета софтверског производа. Међутим, њихов независан развој довео је до њихове неконзистентности па је

развијен нови стандард серије ISO/IEC 25000 који обједињује и замењује стандарде ISO/IEC 9126 и ISO/IEC 14598.

У најопштијем смислу се може рећи да се софтверским стандардом дефинише модел квалитета софтвера (нпр. ISO/IEC 9126-1 модел екстерног и интерног квалитета, ISO/IEC 9126-1 модел употребног квалитета, ISO/IEC 25010 модел квалитета софтверског производа, ISO/IEC 25010 модел употребног квалитета). Модел квалитета садржи атрибуте квалитета софтвера (или карактеристике квалитета), при чему се свака карактеристика квалитета даље декомпонује на подкарактеристике квалитета. За подкарактеристике квалитета се непосредно везују софтверске метрике што је и приказано на наредној слици (Слика 23).



Слика 23. Веза између модела квалитета софтвера, карактеристика, подкарактеристика и софтверских метрика [Milic17]

Примећује се да се софтверским метрикама врши непосредно рачунање вредности подкарактеристика квалитета. Другим речима, софтверске метрике се оперативно користе за утврђивање задовољења подкарактеристика квалитета. С друге стране, на основу подкарактеристика је могуће извршити оцену атрибута квалитета посматраног софтверског система. Другим речима, коришћењем софтверских метрика се посредно врши оцена квалитета софтверског система. У оквиру наредне табеле (Табела 9) дат је приказ стандарда којима се остварује веза између модела квалитета софтвера и софтверских метрика.

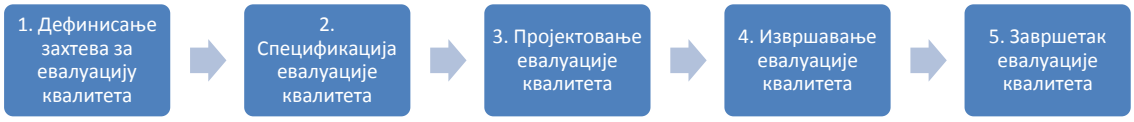
Табела 9. Стандарди којима се остварује веза модела квалитета софтвера и софтверских метрика

Стандард	**Модел квалитета (атрибути квалитета и подкарактеристике)**	**Софтверске метрике**
ISO/IEC 9126	ISO/IEC 9126-1 (Модел екстерног и интерног квалитета)	ISO/IEC 9126-2 (Екстерне софтверске метрике), ISO/IEC 9126-3 (Интерне софтверске метрике)
	ISO/IEC 9126-1 (Модел употребног квалитета)	ISO/IEC 9126-4 (Софтверске метрике употребног квалитета)
ISO/IEC 25000	ISO/IEC 25010 (Модел квалитета софтверског производа)	ISO/IEC DIS 25023 (софтверске метрике квалитета софтверског производа)
	ISO/IEC 25010 (Модел употребног квалитета)	ISO/IEC DIS 25022 (Софтверске метрике употребног квалитета)
	ISO/IEC 25012 (Модел квалитета података)	ISO/IEC DIS 25024 (Софтверске метрике квалитета података)

На основу табеле се може закључити да стандард ISO/IEC 9126 дефинише два модела квалитета (модел екстерног и интерног квалитета и модел употребног квалитета). С друге стране, стандард ISO/IEC 25000 такође

дефинише три модела квалитета (модел квалитета софтверског производа, модел употребног квалитета и модел квалитета података). За сваки модел квалитета су у оквиру посебних стандарда дефинисане софтверске метрике које се оперативно користе за утврђивање квалитета посматраног софтверског производа.

С друге стране, потребно је дефинисати поступак евалуације квалитета софтвера како би се утврђивање квалитета обавило на систематичан и организован начин. На тај начин се формализује приступ процесу евалуације квалитета софтвера. Стандарди ISO/IEC 14598 (у оквиру ISO/IEC 14598-1 стандарда) и ISO/IEC 25000 (у оквиру ISO/IEC 25040 стандарда) дефинишу фазе поступка евалуације квалитета софтвера при чему се фазе незнатно разликују. На наредној слици (Слика 24) приказане су фазе поступка евалуације квалитета софтвера према ISO/IEC 25040 стандарду.



Слика 24. Поступак евалуације квалитета софтвера према ISO/IEC 25040 стандарду [ISO25040]

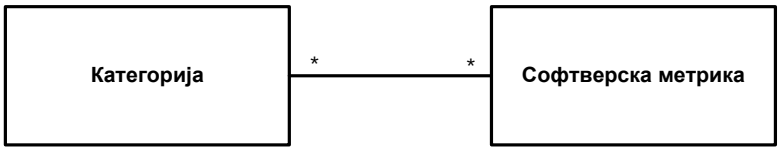
Поступак евалуације обухвата читав низ фаза (дефинисање захтева за евалуацију, спецификација евалуације квалитета, пројектовање евалуације квалитета) који претходе непосредној евалуацији квалитета софтвера. На крају се врши преглед резултата евалуације и израда одговарајућих извештаја о извршеној евалуацији квалитета софтверског производа.

На основу изложеног се може закључити да су софтверске метрике веома важан чинилац за утврђивање квалитета софтверског система. Софтверским метрикама се мере различити атрибути квалитета софтверског система. С друге стране, софтверске метрике се дефинишу у зависности од погледа на софтверски систем: постоје софтверске метрике за мерење интерног квалитета софтвера, софтверске метрике за мерење екстерног квалитета

софтвера, софтверске метрике за мерење квалитета софтверског производа као и софтверске метрике за мерење употребног квалитета софтвера. Поред спецификације модела квалитета софтвера и софтверских метрика важно је дефинисати поступак евалуације квалитета софтвера којим се на организован и систематичан начин омогућава примена дефинисаног модела квалитета и софтверских метрика.

4.4. Класификација софтверских метрика

Постоје различите класификације софтверских метрика. У том смислу се могу дефинисати категорије којима припадају софтверске метрике. Једна категорија може обухватити више софтверских метрика. С друге стране, једна софтверска метрика се може наћи у оквиру више категорија, што је и приказано на наредној слици (Слика 25). Штавише, једна софтверска метрика се може наћи у оквиру различитих класификација.



Слика 25. Класификација софтверских метрика: однос између категорије и софтверске метрике

Према [Kano02] софтверске метрике се, у најопштијем случају, могу класификовати у три категорије:

- Метрике производа (енг. Product Metrics) - У оквиру ове категорије налазе се софтверске метрике које су усмерене на мерење карактеристика посматраног софтверског система.
- Метрике процеса (енг. Process Metrics) - Ова категорија обухвата софтверске метрике које су усмерене на процес животног циклуса развоја софтвера, као и метрике које се користе у процесу одржавања софтвера. Метрике производа и метрике процеса у значајној мери утичу на активности процеса развоја софтвера [Abreu96].

- Метрике пројекта (енг. Project Metrics) - У оквиру ове категорије налазе се метрике које описују карактеристике и извршавање пројекта развоја софтвера. [Kan02]

Стандард квалитета софтвера ISO/IEC 9126 дефинише три категорије софтверских метрика [ISO9126v2] [ISO9126v3] [ISO9126v4]:

- Интерне софтверске метрике - У оквиру ове категорије налазе се софтверске метрике за мерење интерног квалитета софтвера.
- Екстерне софтверске метрике - Ова категорија садржи софтверске метрике за мерење екстерног квалитета софтвера.
- Софтверске метрике употребног квалитета - У оквиру ове категорије налазе се софтверске метрике за утврђивање употребног квалитета посматраног софтверског производа.

С друге стране, стандард квалитета софтвера ISO/IEC 25000 даје нешто другачију класификацију у односу на ISO/IEC 9126 стандард квалитета. Овај стандард дефинише следеће категорије софтверских метрика [ISO25020]:

- Метрике квалитета софтверског производа - Обухвата софтверске метрике којима се утврђује квалитет софтверског производа (екстерни и интерни квалитет софтвера).
- Метрике употребног квалитета - У оквиру ове категорије налазе се софтверске метрике за мерење употребног квалитета софтвера.
- Метрике квалитета података - У оквиру ове категорије налазе се софтверске метрике за утврђивање квалитета података.

У оквиру [Pressman10] је дата следећа класификација софтверских метрика:

- Метрике за модел захтева - У оквиру ове категорије налазе се софтверске метрике које се односе на модел захтева.
- Метрике за модел пројектовања - Ова категорија обухвата софтверске метрике које се односе на пројектовање софтверског система и

обухвата метрике архитектуре софтверског система и објектно-оријентисане софтверске метрике. Објектно-оријентисани софтверски системи се састоје од класа који међусобно сарађују. Класа представља основни градивни елемент објектно-оријентисаних софтверских система па се у оквиру ове категорије могу наћи софтверске метрике које су оријентисане ка класама и чију су класификацију дали различити аутори [Chidamber94] [Harrison98] [Lorenz94]. У наставку рада биће описане софтверске метрике из класификације Chidamber-a [Chidamber94].

- Метрике за веб апликације - У оквиру ове категорије налазе се софтверске метрике које су усмерене на кориснички интерфејс, садржај веб страница и навигацију унутар веб апликација.
- Метрике програмског кода - Представљају специјализоване софтверске метрике које су усмерене на програмски код. Њима се, на пример, разматра број линија програмског кода, број линија коментара, број пакета у пројекту и сл. На тај начин је могуће посматрати квалитет софтверског система из различитих перспектива (нпр. на нивоу пројекта, на нивоу пакета у пројекту, на нивоу класе у пакету итд.).
- Метрике за тестирање - Ова категорија обухвата софтверске метрике које се односе на процес тестирања софтвера. У том смислу се разматрају метрике које се односе предвиђање потребног броја тестова за сваку од компоненти, као и на одређивање покривености програмског кода тестовима (енг. test coverage).
- Метрике за одржавање - Претходно наведене софтверске метрике се могу користити и у процесу развоја софтвера и у процесу одржавања софтвера. С друге стране, у оквиру ове категорије налазе се софтверске метрике које су експлицитно усмерене на одржавање софтверског система, односно на разматрање извршених промена у модулима и њиховог утицаја на остатак софтверског система.

Може се приметити да постоје различите класификације софтверских метрика [Briandoo]. Међутим, примећује се да су софтверске метрике усмерене на различите димензије процеса развоја софтвера и да омогућавају сагледавање квалитета софтверског производа из различитих перспектива. У оквиру наредне табеле (Табела 10) дат је сумарни преглед класификација софтверских метрика.

Табела 10. Сумарни преглед класификација софтверских метрика

Извор	**Класификација**
Kan, S. H. (2002). Metrics and models in software quality engineering. Addison-Wesley Longman Publishing Co., Inc..	Метрике производа, Метрике процеса, Метрике пројекта
ISO/IEC 9126 стандард квалитета софтвера	Интерне софтверске метрике, Екстерне софтверске метрике, Софтверске метрике употребног квалитета
ISO/IEC 25000 стандард квалитета софтвера	Метрике квалитета софтверског производа, Метрике употребног квалитета, Метрике квалитета података
Pressman, R. S. (2010). Software engineering: a practitioner's approach. Palgrave Macmillan.	Метрике за модел захтева, Метрике за модел пројектовања, Метрике за веб апликације, Метрике програмског кода, Метрике за тестирање, Метрике за одржавање

4.5. Опис софтверских метрика

У наставку је дат опис често коришћених софтверских метрика:

- Циклична сложеност (енг. Cyclomatic Complexity - CC),
- Сложеност пондерисаних метода (енг. Weighted Method Complexity - WMC2),
- Број пондерисаних метода класе (енг. Number of Weighted Methods - WMC1),

- Број одговора класе (енг. Response for Class - RFC),
- Недостатак кохезивности метода у класи (енг. Lack of Cohesion in Methods - LCOM),
- Повезаност објеката (енг. Coupling Between Objects - CBO),
- Дубина стабла наслеђивања (енг. Depth of Inheritance Tree - DIT),
- Број подкласа (енг. Number of Children - NOC),
- Број наредби у методи (енг. Number of statements in method),
- Метрика стабилности софтвера (енг. Stability Software Metric).

Свака софтверска метрика биће описана на следећи начин:

- Најпре ће бити дата дефиниција посматране софтверске метрике.
- Затим ће, на основу дефиниције, бити дефинисан математички модел посматране софтверске метрике и биће приказан конкретан пример.
- Узимајући у обзир дефиницију и математички модел биће дата интерпретација вредности посматране софтверске метрике.

Ове софтверске метрике имају велику практичну примену и подржане су од стране већине алата који се баве статичком анализом програмског кода. Још један разлог за избор ових метрика је њихова објектно-оријентисана усмереност и примена у развоју објектно-оријентисаних софтверских система, што представља фокус нашег рада.

4.5.1. Циклична сложеност

Метрика Циклична сложеност је, такође, усмерена на класу и методе класе. Циклична сложеност представља сложеност примењеног алгоритма у методи. Свака метода класе садржи неки алгоритам у оквиру којег се примењују алгоритамске структуре, заједно са операторима и наредбама посматраног програмског језика, тј. свака метода класе је усмерена на реализацију неке функционалности.

Математички модел:

Циклична сложеност софтверског система се мери рачунањем броја тачака одлучивања или условних исказа посматраног програмског језика [Milico7]. Другим речима, метрика узима у обзир бинарне операторе "&&", "||", "&", "|", као и тернарни оператор ":?". Метрика *Циклична сложеност* такође узима у обзир наредбе "if", "for", "while..do", "do..while", "switch", "case", "return", "throw" и "catch" посматраног програмског језика.

Другим речима, циклична сложеност се рачуна на следећи начин:

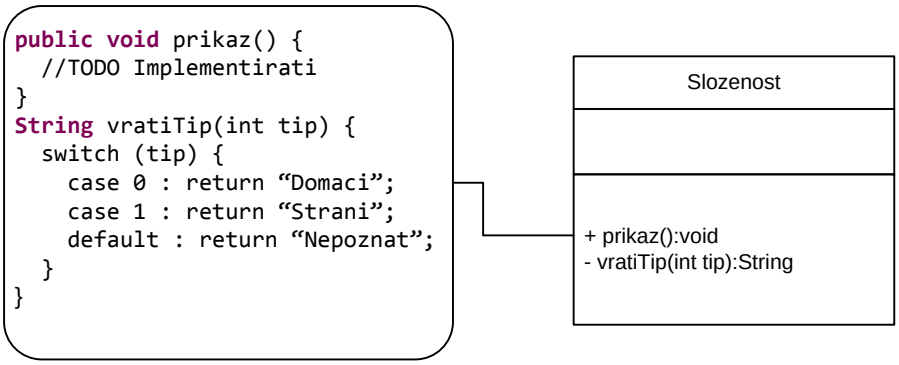
Циклична сложеност = P + 1,

При чему је:

P - Број предиката или број услова или број бинарних оператора,

1 - Улазна путања функције.

Размотримо пример који је приказан на наредној слици (Слика 26). Са слике се уочава класа Slozenost која садржи две методе: *prikaz* (чија је циклична сложеност C1) и *vratiTip* (чија је циклична сложеност C2). С обзиром да метода *prikaz()* није имплементирана, она не садржи предикате, услове и бинарне операторе. У том смислу ће њена циклична сложеност ће бити C1 = 1. С друге стране, метода *vratiTip()* је имплементирана и враћа тип у зависности параметра. С обзиром да се у оквиру метода извршава гранање, њена циклична сложеност ће бити једнака C2 = 4 (у методи, практично, постоје три услова - две case наредбе и једна default наредба).



Слика 26. Сложеност пондерисаних метода и циклична сложеност

Интерпретација:

Циклична сложеност сваке методе је већа или једнака броју 1. Уколико функција не садржи предикате, услове и бинарне чворове - њена циклична сложеност ће бити 1. С друге стране, постојање предиката, услова и бинарних чворова доводи до повећања цикличне сложености посматране методе (у том случају ће почетна циклична сложеност бити увећана у зависности од број предиката, услова или бинарних чворова).

Идеално би било да циклична сложеност сваке методе буде 1. Висока циклична сложеност представља индикацију да је примењени алгоритам у методи сложен (да садржи вишеструко гранање, вишеструку обраду изузетака, вишеструко испитивање услова, који се могу извршавати у петљи, што додатно повећава цикличну сложеност), што у значајној отежава разумевање, тестирање и одржавање методе. Висока циклична сложеност методе је индикација да би посматрану методу класе требало организовати на другачији начин (нпр. поделити посматрану методу на две или више метода, у зависности од проблема).

Иако постоје критике да метрика не садржи експлицитни модел на коме се заснива циклична сложеност [Shepperd88], чињеница је да је циклична сложеност често примењивана метрика која је нашла велику практичну примену и која је подржана од великог броја алата који се баве статичком анализом програмског кода. Метрика се, између осталог, може користи и у

процесу тестирања софтверских система [McCabe89][Watson96]. Она се такође користи као мера сложености за рачунање вредности метрике Сложеност пондерисаних метода (описана је у оквиру претходног одељка).

4.5.2. Сложеност пондерисаних метода

Објектно-оријентисани софтверски системи садрже велики број класа које су логички организоване у пакете. Класама се, заправо, реализују функционалности софтверског система. Свака класа садржи атрибуте и методе класе које се једним именом називају чланице класе:

- Атрибутима класе дефинисана је структура класе
- Методама класе дефинисано је понашање класе

Метрика *Сложеност пондерисаних метода* се дефинише као сума сложености метода класе. При томе, остављена је могућност избора метрика које се користе као "сложеност методе". Свакој методи класе се додељује одређена сложеност (тј. тежински коефицијент), па сума сложености метода представља збир тих сложености.

Математички модел:

Претпоставимо да имамо класу C која садржи методе M1, M2, ..., Mn (наведене методе представљају саставни део класе C). Сваком методом дефинисано је одређено понашање, тј. свака метода је усмерена на решавање одређеног проблема. У том случају се може рећи да свака метода има своју сложеност C1, C2, ..., Cn. На основу тога се сложеност пондерисаних метода може израчунати на следећи начин:

Сложеност пондерисаних метода (WMC_2) = C1 + C2 + ... + Cn,

При чему је Ci, i=1..n, сложеност i-те методе класе.

Као што се може приметити, свакој методи је додељен одређени пондер који представља сложеност методе, при чему није експлицитно наведено шта је

то сложеност методе. Најчешће се као "мера сложености" методе користи метрика *Циклична сложеност* (која је описана у претходном одељку).

Размотримо пример који је приказан на претходној слици (Слика 26). Са слике се уочава класа Slozenost која садржи две методе: *prikaz* (чија је циклична сложеност C1) и *vratiTip* (чија је циклична сложеност C2). С обзиром да метода *prikaz()* није имплементирана, њена циклична сложеност ће бити једнака C1 = 1. Метода *vratiTip()* је имплементирана и враћа тип у зависности параметра, тако да ће њена циклична сложеност бити једнака C2 = 4. У том смислу ће вредност метрике Сложеност пондерисаних метода бити једнака:

$WMC_2 = C1 + C2 = 1 + 4 = 5$.

Интерпретација:

Сложеност пондерисаних метода је већа или једнака броју 1. Уколико се као мера сложености користи метрика *Циклична сложеност*, идеално би било да циклична сложеност сваке методе буде једнака 1. У том случају би сложеност пондерисаних метода била једнака броју метода класе. Уколико класа има високу сложеност пондерисаних метода требало би извршити рефакторисање (рефакторисати класу/методу на две или више класа/метода, респективно) [Milico7].

4.5.3. Број пондерисаних метода класе

У оквиру претходног одељка објашњена је софтверска метрика *Сложеност пондерисаних метода*. Софтверска метрика *Број пондерисаних метода класе* је непосредно повезана са објашњеном софтверском метриком и усмерена је на одређивање броја пондерисаних метода у класи.

Математички модел:

Нека је дата класа C која садржи методе M1, M2, ..., Mn. Нека је сложеност сваке методе C1, C2, ..., Cn. На основу тога се број пондерисаних метода класе може израчунати на следећи начин:

Број пондерисаних метода класе (WMC_1) = n,

При чему n представља број пондерисаних метода класе.

У том смислу разматра однос метрика *Број пондерисаних метода класе* и *Сложеност пондерисаних метода*, тј:

$$WMC_{odn} = \frac{\text{Број пондерисаних метода класе}}{\text{Сложеност пондерсаних метода}}$$

Размотримо пример који је приказан на претходној слици (Слика 26). Као што је раније напоменуто, посматрана класа има две пондерисане методе па ће вредност метрике *Број пондерисаних метода класе* бити 2. С друге стране, сложеност пондерисаних метода класе износи 5. Због тога ће вредност метрике WMC_{odn} бити:

WMC_{odn} = 2 / 5 = 0.4.

Интерпретација:

Вредност WMC_{odn} је мања или једнака броју 1. Као што је претходно објашњено, Сложеност пондерисаних метода је већа или једнака броју 1. Идеално би било када би WMC_{odn} имао вредност 1, односно идеално би било да је број пондерисаних метода класе једнак сложености пондерисаних метода. С друге стране, уколико је сложеност пондерисаних метода већа од броја пондерисаних метода класе, класа може бити тешка за одржавање.

4.5.4. Број одговора класе

У развоју објектно-оријентисаних софтверских система објекти комуницирају разменом порука. Та комуникација се остварује позивом одговарајуће методе класе. У том смислу се може рећи да одређена порука може бити побуда која ће довести до тога да се посматрани објекат понаша

на одређени начин. Та побуда се реализује позивом методе посматраног објекта класе. У том контексту се методе посматране класе могу дефинисати као одговор на одређене поруке [Chidamber94].

Математички модел:

С обзиром на дефиницију, објектно-оријентисана софтверска метрика *Број одговора класе* се може израчунати на следећи начин:

Број одговора класе = |RS|,

При чему је:

RS - Скуп одговора класе, односно скуп свих метода које могу бити позване као одговор на поруку објекта класе

Даљом разрадом формуле добија се следеће:

Број одговора класе = |RS| = M + R,

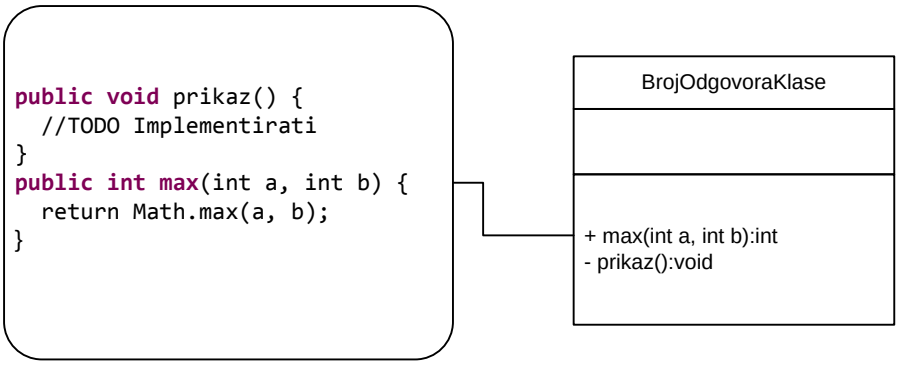
При чему је:

M - Број метода у посматраној класи које могу бити позване као одговор класе,

R - Укупан број других метода које се позивају од стране метода посматране класе (ове методе се позивају од стране метода које су дефинисане у M).

Размотримо пример који је приказан на наредној слици (Слика 27). Са слике се уочава класа BrojOdgovoraKlase која садржи две методе: *prikaz()* и *max()*. У том смислу је M = 2. С друге стране, из методе *max()* се позива метода класе Math, па ће због тога R = 1. Због тога је вредност метрике број одговора класе:

Број одговора класе = M + R = 2 + 1 = 3.



Слика 27. Број одговора класе

Интерпретација:

Број одговора класе представља скуп метода које могу бити позване као одговор на неку побуду. Уколико је тај скуп метода велики - класа је сложенија. Велики број одговора класе такође повећава сложеност тестирања и уочавања грешака у посматраној класи [Chidamber94]. У том случају би требало размислити о другачијем организовању класе и њених метода тј. потребно је да се број одговора посматране класе смањи.

4.5.5. Недостатак кохезивности метода у класи

Као што је раније напоменуто, објектно-оријентисани софтверски системи садрже велики број класа којима се реализују функционалности софтверског система. Свака класа садржи атрибуте класе (којима се дефинише структура класе) и методе класе (којима се дефинише понашање класе). У том смислу се може рећи да је сваки објекат одређен структуром и понашањем. Обично се структури не приступа директно, већ се за то користе методе. Другим речима, структура класе се мења коришћењем понашања - у зависности од одређеног понашања долази до промене структуре класе.

На основу изложеног се може закључити да су методе класе и атрибути класе непосредно повезани тј. да методе класе мењају вредности атрибутима класе. За класу се каже да је кохезивна уколико има методе које се извршавају над истим скупом атрибута. Потребно је да кохезија унутар класе буде висока, односно да недостатак кохезије у класи буде што мањи.

Математички модел:

Нека је дата класа C која садржи методе M1, M2,..., Mn. Нека је $\{I_j\}$ скуп свих атрибута који се користе од стране методе Mj. С обзиром да класа C има n атрибута постојаће n оваквих скупова $\{I_1\}$, $\{I_2\}$,..., $\{I_n\}$. С обзиром на дефиницију, објектно-оријентисана софтверска метрика *Недостатак кохезивности метода у класи* се може дефинисати на следећи начин [Chidamber94]:

Недостатак кохезивности метода у класи (LCOM) = |P| - |Q|, уколико је |P| > |Q|, а у супротном је Недостатак кохезивности метода у класи = 0

При чему је:

$P = \{(I_i, I_j) | I_i \cap I_j = \varnothing\}$ - број празних пресека скупова (не постоје заједнички атрибути који се користе од стране метода M_i и M_j),

$Q = \{(I_i, I_j) | I_i \cap I_j \neq \varnothing\}$ - број не-празних пресека скупова (постоје заједнички атрибути који се користе од стране метода M_i и M_j)

Поред тога, постоје још два побољшања метрике *Недостатак кохезивности метода у класи* (у литератури су побољшања софтверске метрике позната под називом LCOM2 и LCOM3) [Henderson-Sellers96]:

Недостатак кохезивности метода у класи 2 (LCOM2) = 1 - sum(mA) / (m*a),

Недостатак кохезивности метода у класи 3 (LCOM3) = (m-sum(mA)/a) / (m-1), при чему је:

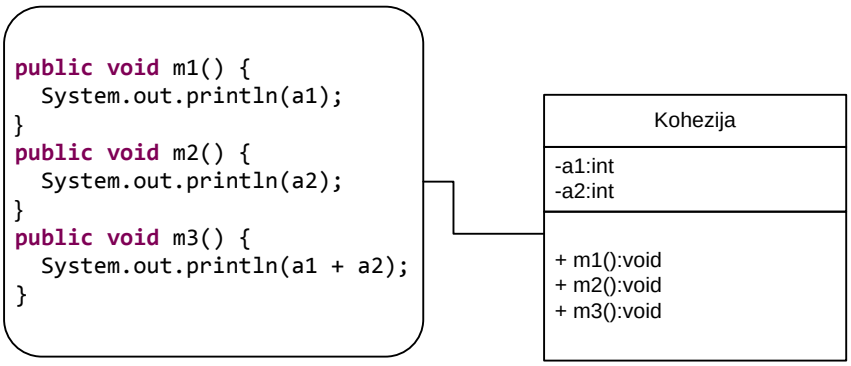
m - Број метода у класи,

a - Број атрибута у класи,

mA - Број метода које приступају атрибуту A,

sum(mA) - Сума свих mA вредности унутар посматране класе.

Размотримо пример који је приказан на наредној слици (Слика 28). Са слике се уочава класа Kohezija која има два атрибута (атрибути a1 и a2) и три методе (методе *m1*, *m2* и *m3*). При томе, метода *m1()* приступа атрибуту a1, метода *m2()* приступа атрибуту a2, док метода *m3()* приступа атрибутима a1 и a2.



Слика 28. Недостатак кохезивности метода у класи

У наредној табели (Табела 11) дат је упоредни приказ вредности метрика LCOM, LCOM2 и LCOM3 за посматрани пример.

Табела 11. Упоредни приказ вредности метрика LCOM, LCOM2 и LCOM3

Софтверска метрика	**Напомена**
LCOM = |P| - |Q|, уколико је |P| > |Q|, иначе је 0 LCOM = 0	I_{m1} = {a1}, I_{m2} = {a2}, I_{m3} = {a1, a2} P = 1, Q = 2
LCOM2 = 1 - sum(mA) / (m*a) LCOM2= 1 - 4 / (3*2) = 1 - 0.67 = 0.33	m=3, a=2, mA(a1)=2, mA(a2)=2, sum(mA)=4
LCOM3 = (m-sum(mA)/a) / (m-1) LCOM3 = (3 - 4/2) / (3-1) = 1 / 2 = 0.5	

Интерпретација:

Недостатак кохезивности метода у класи је број који је већи или једнак 0. Потребно је да свака класа има висок ниво кохезије [Chidamber94] [Pressman10] па се стога може рећи да је пожељно да недостатак кохезије буде што мањи. Идеално би било да недостатак кохезивности метода у класи не постоји (да је вредност броја празних пресека скупова једнака 0, при чему је вредност броја не-празних пресека скупова максимална), што би значило да свака метода приступа сваком атрибуту класе. Висок недостатак кохезивности метода у класи је индикација да посматрану класу треба рефакторисати (поделити је на две или више класа).

Уз наведено, метрика *Недостатак кохезивности метода у класи* има бројне критике и покушаје унапређења [Henderson-Sellers96][Hitz96][Mayer99]. Тако се, поред оригиналне верзије за рачунање вредности софтверске метрике, у литератури могу пронаћи различите софтверске метрике за утврђивање кохезије (или недостатка кохезије) метода у класи [Badri04].

4.5.6. Повезаност објеката

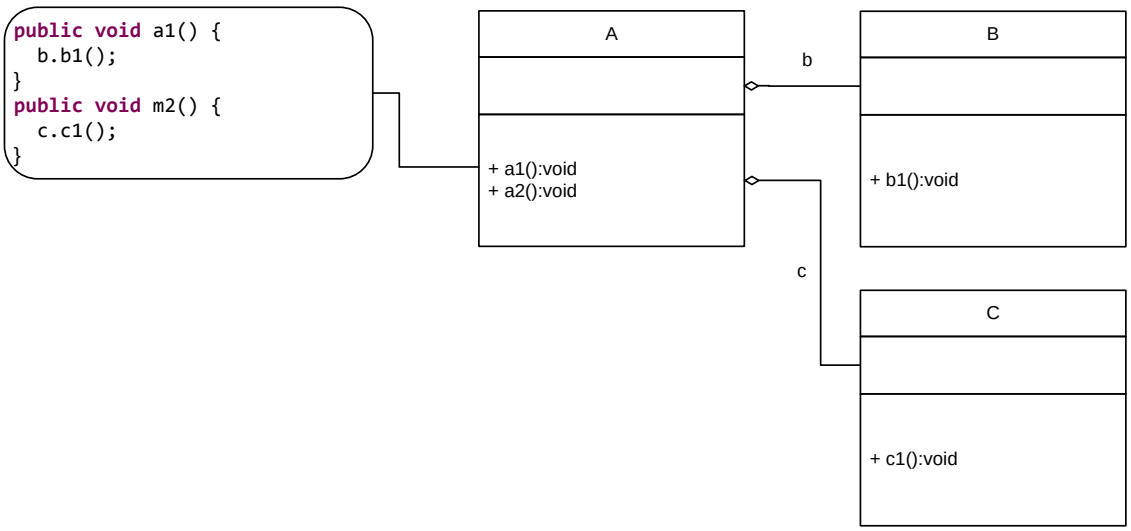
У претходном одељку описана је софтверска метрика *Недостатак кохезивности метода у класи*. Приликом објектно-оријентисаног развоја потребно је да кохезија класе буде на високом нивоу, односно да недостатак кохезије класе буде на ниском нивоу. Из изложеног се може закључити да је кохезија класе усмерена на класу, методе посматране класе и атрибуте посматране класе, односно кохезија разматра однос чланица једне класе.

С друге стране, повезаност објеката је усмерена на однос између класа (разматра се однос између класа, њихових метода и њихових атрибута). Стога се може рећи да су две класе међусобно повезане уколико методе једне класе користе атрибуте или методе друге класе.

Математички модел:

С обзиром на дефиницију, вредност метрике *Повезаност објеката* посматране класе се рачуна тако што се непосредно изброји број других класа са којима је посматрана класа повезана (чије атрибуте и методе користи) [Chidamber94].

Размотримо пример који је приказан на наредној слици (Слика 29). Са слике се уочавају класе A, B и C, при чему класа A као атрибуте садржи објекте класе B и C. Класа B има методу *b1()*, класа C има методу *c1()*, док класа A има методе *a1()* и *a2()* из којих се позивају методе класа B и C, респективно. С обзиром да методе класе A користе методе класа B и C, може се рећи да је повезаност објеката у класи A једнака 2. С друге стране, класа B не користи атрибуте и методе других класа, па је због тога повезаност објеката класе B једнака нули. Такође, класа C не користи атрибуте и методе других класа, па је због тога повезаност објеката класе C једнака нули.



Слика 29. Повезаност објеката

Интерпретација:

Потребно је да повезаност објеката буде на ниском нивоу. Висока повезаност класа смањује ниво модуларног пројектовања и онемогућава поновно коришћење класа [Chidamber94]. У том смислу је потребно да се повезаност класа сведе на најмању могућу меру (идеално би било да се

повезаност класа остварује на нивоу интерфејса). У том случају је посматрана класа "више независна" па се лакше може поновно користити. Поред тога, висока повезаност објеката отежава измену класе, као и тестирање класе након извршених измена [Pressman10].

С обзиром на дефиницију метрике *Повезаност објеката* и на дефиницију метрике *Недостатак кохезивности у класи*, можемо рећи да приликом развоја објектно-оријентисаних софтверских система треба тежити да кохезија класе буде на високом нивоу, односно да повезаност класа треба бити на ниском нивоу. На тај начин се зависности између класа своде на минимум уз истовремено повећавање једноставности измене, одржавања, тестирања и могућности поновног коришћења компоненти посматраног софтверског система.

4.5.7. Дубина стабла наслеђивања

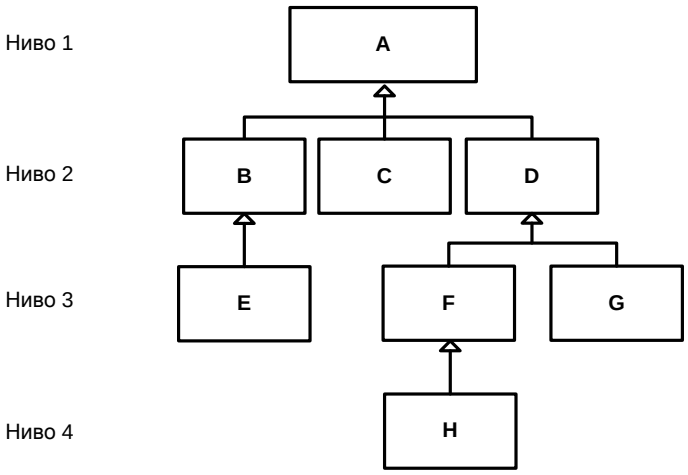
Метрика *Дубина стабла наслеђивања* је објектно-оријентисана софтверска метрика која је усмерена на класу и одређивање њене позиције у хијерархији наслеђивања. У том смислу се дубина класе у хијерархији наслеђивања може дефинисати као максимални број нивоа од посматраног чвора до корена стабла [Chidamber94].

Математички модел:

С обзиром на дефиницију, софтверска метрика *Дубина стабла наслеђивања* се рачуна непосредним бројањем предака посматране класе у хијерархији наслеђивања. Уколико је посматрана класа изведена из више класа (ово је могуће уколико програмски језик подржава вишеструко наслеђивање, па је у том случају могуће да једна класа има више надкласа), рачунање се посматра за све путање и дубина стабла наслеђивања једнака је највећем број нивоа од посматране класе до корена стабла.

На наредној слици (Слика 30) дат је пример класа и њихове хијерархије наслеђивања. На врху хијерархије налази се класа A, при чему су из ње

изведене класе B, C и D. Из класе D је изведена класа E, док класа E има два потомка: класу F и класу G. На дну хијерархије је класа H која је изведена из класе F.



Слика 30. Пример класа и њихове хијерархије наслеђивања

На основу слике и дефиниције софтверске метрике можемо израчунати дубину стабла наслеђивања за сваку класу:

- С обзиром да је класа A на врху хијерархије (она није изведена из других класа), њена дубина стабла наслеђивања је 1.
- Класе B, C, и D су непосредни потомци класе A па је дубина стабла наслеђивања за класе B, C и D једнака 2.
- Класа E је изведена из класе B. Њена дубина стабла наслеђивања је 3.
- С друге стране, на истом нивоу се налазе класе F и G које су изведене из класе D. Њихова дубина стабла наслеђивања је такође 3.
- На дну хијерархије налази се класа H која је непосредни потомак класе F. Њена дубина стабла наслеђивања је 4.

Интерпретација:

Потребно је да дубина стабла наслеђивања буде што мања. Уколико је дубина класе наслеђивања велика то значи да је она изведена из већег броја класа па самим тим може садржати и већи број наслеђених метода које имају неко понашање. Штавише, наведене методе се у поткласама могу

прекрити (енг. Method Overriding) чиме се мења понашање које је дефинисано у оквиру надређене класе. У том случају је теже предвидети понашање класе, што доводи то тежег разумевања и одржавања посматраног софтверског система [Chidamber94]. С друге стране, уколико је хијерархија класа добро пројектована, може се повећати могућност поновног коришћења метода [Chidamber94].

4.5.8. Број подкласа

Метрика *Број подкласа* је објектно-оријентисана софтверска метрика која је усмерена на класу и одређивање броја њених непосредних потомака у хијерархији наслеђивања [Chidamber94].

Математички модел:

С обзиром на дефиницију, софтверска метрика *Број подкласа* се рачуна бројањем непосредних потомака посматране класе у хијерархији класа. За разлику од софтверске метрике *Дубина стабла наслеђивања* која проналази максималну путању од посматране класе до корена стабла ова софтверска метрика у обзир узима само непосредне потомке посматране класе (класе које су директно изведене из посматране класе).

На основу дијаграма класа датог у претходном примеру (Слика 30) и дефиниције софтверске метрике можемо израчунати број подкласа за сваку класу:

- Класа A налази се на врху хијерархије, при чему су из ње непосредно изведене три класе (B, C и D). Број подкласа класе A је 3.
- Из класе B је изведена једна класа E па је број подкласа класе B једнак 1. Класа C нема непосредних потомака па је број подкласа класе C једнак 0. Из класе D су изведене класе F и G па је број подкласа класе D једнак 2.
- Класа E нема непосредних потомака па је број подкласа класе E једнак 0. Из класе F изведена је класа H па је број подкласа класе F једнак 1.

С друге стране, класа G нема непосредних потомака па је њен број подкласа једнак 0.

- Класа H нема непосредних потомака па је број подкласа класе H једнак 0.

Софтверска метрика *Број подкласа* је комплементарна софтверској метрици *Дубина стабла наслеђивања*. Ове софтверске метрике усмерене су на концепт наслеђивања у посматраном објектно-оријентисаном програмском језику. Метрика Дубина стабла наслеђивања је усмерена на утврђивање броја класа (у смислу хијерархије наслеђивања) које су коришћене како би била формирана посматрана класа, док је метрика *Број подкласа* усмерена на утврђивање броја класа које су непосредно формиране од стране посматране класе (у смислу хијерархије наслеђивања). Ове софтверске метрике се најчешће користе у пару. Наредна табела (Табела 12) даје сумарни преглед вредности софтверских метрика *Број подкласа* и *Дубина стабла наслеђивања* за пример са претходне слике (Слика 30).

Табела 12. Упоредни приказ вредности софтверских метрика Дубина стабла наслеђивања и Број подкласа

Класа	**Софтверска метрика**	
	Дубина стабла наслеђивања (DIT)	**Број подкласа (NOC)**
	Максимални број нивоа од посматраног чвора (тј. класе) до корена стабла.	Број непосредних потомака класе у хијерархији наслеђивања.
Класа A	DIT(A) = 1	NOC(A)= 3
Класа B	DIT(B) = 2	NOC(B) = 1
Класа C	DIT(C) = 2	NOC(C) = 0

Класа	**Софтверска метрика**	
	Дубина стабла наслеђивања (DIT)	**Број подкласа (NOC)**
	Максимални број нивоа од посматраног чвора (тј. класе) до корена стабла.	Број непосредних потомака класе у хијерархији наслеђивања.
Класа D	DIT(D) = 2	NOC(D) = 2
Класа E	DIT(E) = 3	NOC(E) = 0
Класа F	DIT(F) = 3	NOC(F) = 1
Класа G	DIT(G) = 3	NOC(G) = 0
Класа H	DIT(H) =4	NOC(H) = 0

Интерпретација:

Велики број подкласа повећава могућност поновног коришћења класе. С друге стране, велики број подкласа може довести до неодговарајуће апстракције родитељске класе (уколико посматрана класа има велики број непосредно изведених класа може доћи до грешке приликом креирања објекта класе).

4.5.9. Број наредби у методи

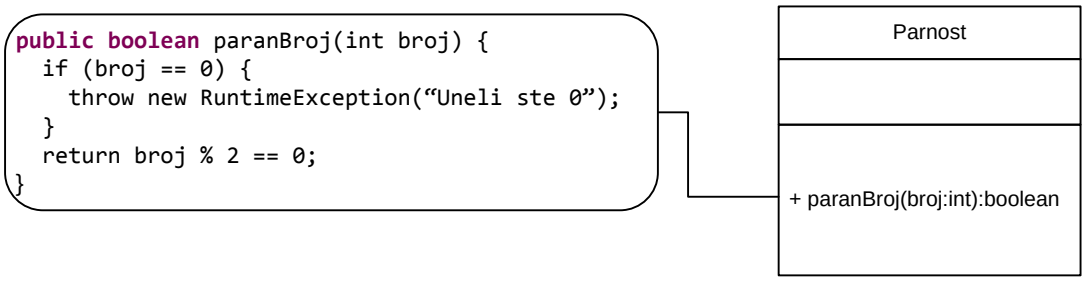
Уколико разматрамо процес објектно-оријентисаног развоја софтвера, методама се реализује понашање софтверског система. Метрика *Број наредби у методи* је софтверска метрика која је усмерена на методу класе и одређивање непосредног броја наредби које се позивају унутар посматране методе.

Математички модел:

С обзиром на дефиницију, софтверска метрика *Број наредби у методи* се рачуна бројањем наредби унутар посматране методе. Под наредбом се сматра if-then-else конструкција, switch/case конструкција, return наредба, затим коришћење петљи (for петља, foreach петља, петља са предусловом, петља са постусловом), позив друге методе итд.

Размотримо пример који је приказан на наредној слици (Слика 31). Са слике се уочава класа Parnost која садржи методу *paranBroj()*. Унутар посматране методе врши се провера да ли је прослеђени број паран или непаран на следећи начин:

- Број нула није ни паран ни непаран број. Због тога се, у том случају, генерише изузетак.
- Број је паран уколико је остатак приликом дељења посматраног броја са бројем 2 једнак нули. У том случају метода враћа логичку вредност true. У супротном је број непаран и метода враћа логичку вредност false.



Слика 31. Број наредби у методи

У посматраној методи се уочавају три наредбе: if, throw и return, па ће због тога вредност метрике *Број наредби у методи* бити 3.

Интерпретација:

Број наредби у методи је већи или једнак нули. Потребно је да број наредби у методи буде што је могуће мањи. Превелики број наредби може довести до отежаног разумевања и одржавања посматране методе. У том случају треба размотрити могућност да се посматрана метода подели на више метода.

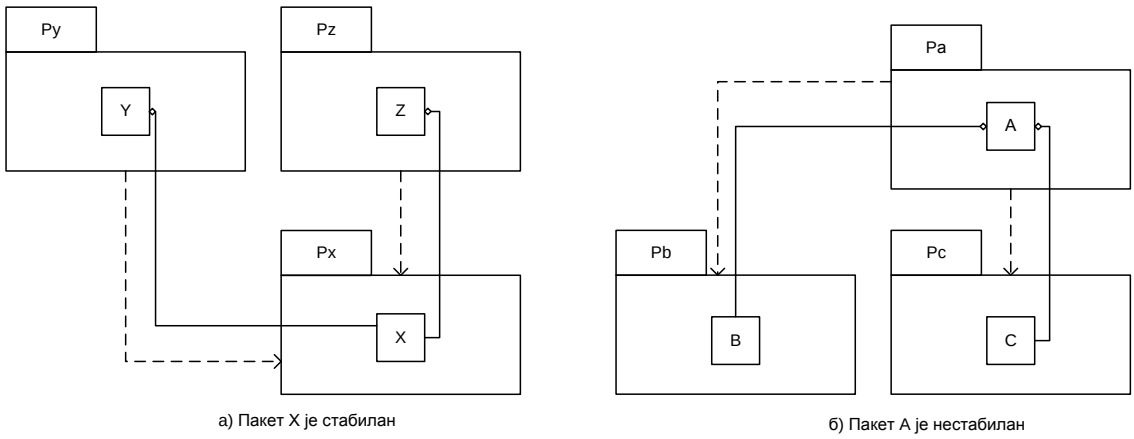
4.5.10. Метрика стабилности софтвера

Према ISO/IEC 9126 стандарду квалитета, стабилност софтвера се може дефинисати као осетљивост система на промене [ISO9126]. Метрику стабилности софтвера дефинисао је Роберт Мартин [Martin06]. Ова софтверска метрика разматра стабилност софтвера у контексту односа између пакета који постоје у посматраном софтверском систему [Martin06].

Математички модел:

Размотримо пример који је приказан на наредној слици (Слика 32). Претпоставимо да софтверски систем садржи пакете Py, Pz и Px који садрже класе Y, Z и X, респективно. При томе, класе Y и Z за реализацију својих функционалности користе класу X. За пакет Px се каже да је стабилан зато што не зависи од других пакета [Martin06].

С друге стране, на слици је дат приказ софтверског система са пакетима Pa, Pb и Pc који садрже класе A, B и C, респективно. При томе, класа A за реализацију својих функционалности користи класе B и C. За пакет Pa се каже да је нестабилан зато што зависи од других пакета [Martin06].



Слика 32. Стабилност и нестабилност пакета [Martin06]

У том смислу је могуће дефинисати метрику нестабилности софтвера [Martin06]:

I = Ce / (Ca + Ce),

при чему је:

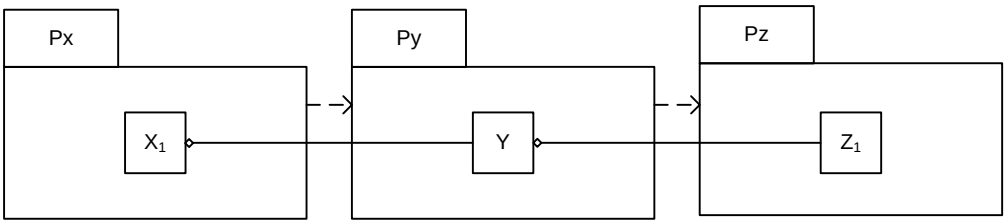
I - Нестабилност софтвера,

Ce - Одлазна повезаност (енг. efferent coupling), тј. број класа унутар пакета које зависе од класа изван пакета,

Ca - Долазна повезаност (енг. afferent coupling), тј. број класа изван пакета које зависе од класа унутар пакета.

4.5.10.1. Побољшање Метрике стабилности софтвера Роберта Мартина

Метрика стабилности софтвера не узима у обзир ситуацију када не постоји одлазна или долазна повезаност пакета [Vlajic17]. Размотримо пример који је приказан на наредној слици (Слика 33).

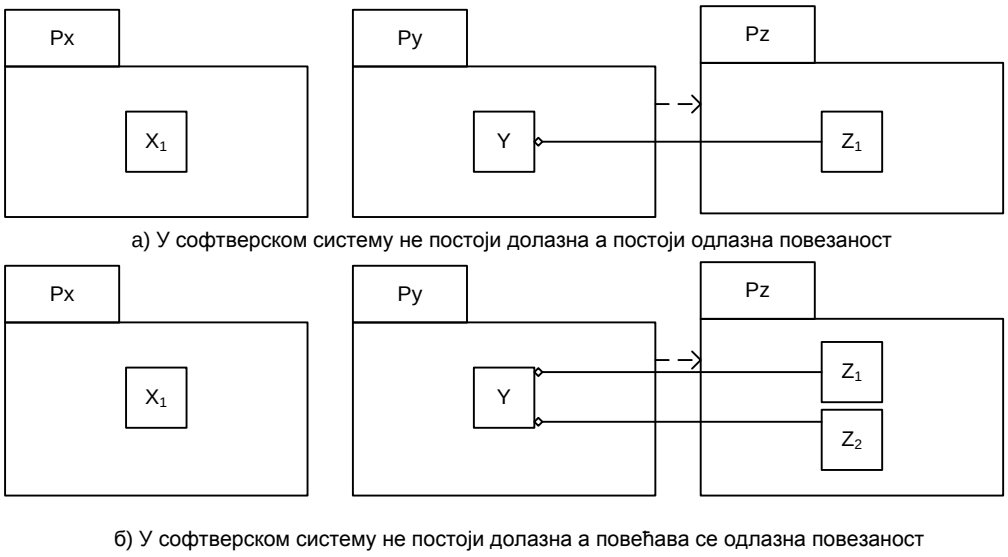


Слика 33. Софтверски систем који садржи одлазну и долазну повезаност пакета [Vlajic17]

Са слике се уочавају пакети Px, Py и Pz који садрже класе X1, Y и Z1, респективно. Другим речима, пакет Px зависи од пакета Py, док пакет Py зависи од пакета Pz. Због тога је нестабилност пакета Py: I = 1/(1+1) = 1/2 = 0.5, с обзиром да је Ce=1, Ca=1. Уколико се у пакетима додају нове класе које такође зависе од класа из других пакета, долази и до промене вредности метрике стабилности софтвера.

С друге стране, проблем настаје уколико не постоји долазна а постоји одлазна повезаност. Размотримо пример који је дат на наредној слици (Слика 34). Са слике се уочавају пакети Px, Py и Pz који садрже класе X1, Y и Z1, респективно. При томе, пакет Px не зависи ни од једног пакета док пакет Py зависи од пакета Pz. У том смислу у софтверском систему не постоји

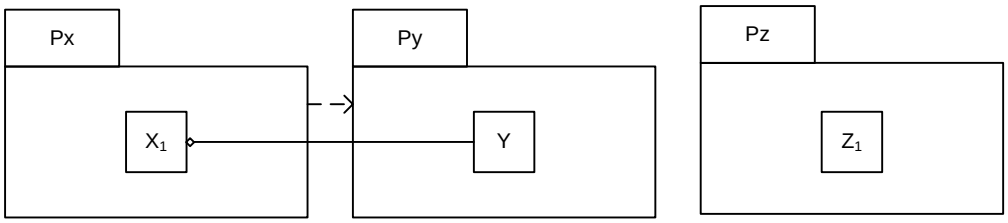
долазна повезаност а постоји одлазна повезаност, па је нестабилност посматраног пакета Py: I = 1/(0+1) = 1/1 = 1, с обзиром да је Ca = 0, Ce = 1. Уколико се у пакету Pz уведе нова класа Z2, вредност метрике нестабилност пакета Py је: I = 2/(0+2) = 2/2 = 1, с обзиром да је Ca = 0, Ce = 2.



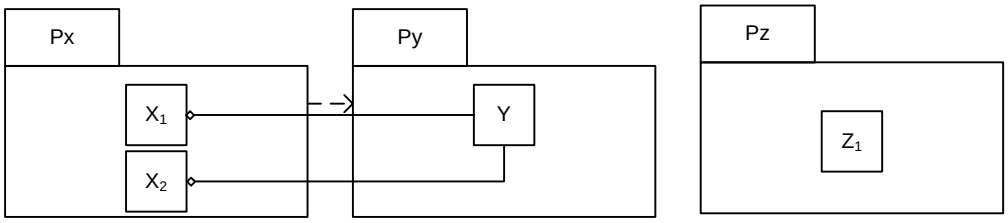
Слика 34. Релативизација одлазне повезаности када је долазна повезаност једнака нули [Vlajic17]

Другим речима, приликом додавања нове одлазне повезаности не долази до промене нестабилности пакета (нестабилност пакета ће увек имати вредност 1). ***У том смислу долази до релативизације одлазне повезаности када је долазна повезаност једнака нули*** [Vlajic17].

Проблем такође настаје уколико не постоји одлазна а постоји долазна повезаност. Размотримо пример који је дат на наредној слици (Слика 35). Са слике се уочавају пакети Px, Py и Pz који садрже класе X1, Y и Z1, респективно. При томе, пакет Px зависи од пакета Py, док пакет Pz не зависи од других пакета. У том смислу у софтверском систему не постоји одлазна повезаност а постоји долазна повезаност, па је нестабилност пакета Py: I = 0/(1+0) = 0/1 = 0, с обзиром да је Ca = 1, Ce = 0. Уколико се у пакету Px уведе нова класа X2, вредност метрике нестабилност пакета Py је: I = 0/(2+0) = 0/2 = 0, с обзиром да је Ca = 2, Ce = 0.



а) У софтверском систему не постоји одлазна а постоји долазна повезаност



б) У софтверском систему не постоји одлазна а повећава се долазна повезаност

Слика 35. Релативизација долазне повезаности када је одлазна повезаност једнака нули [Vlajic17]

Другим речима, приликом додавања нове долазне повезаности не долази до промене нестабилности пакета (нестабилност пакета ће увек имати вредност 0). ***У том смислу долази до релативизације долазне повезаности када је одлазна повезаност једнака нули*** [Vlajic17].

У том смислу се разматра стабилност софтвера Ir(δ1, δ2) приликом релативизације Ce (када је Ca = 0), односно приликом релативизације Ca (када је Ce = 0), што је и приказано у наредној табели (Табела 13) [Vlajic17].

Табела 13. Проблем релативизације одлазне и долазне повезаности [Vlajic17]

Ca	**Ce**	**Ir(δ1, δ2) = (Ce+ δ1) / (Ca + Ce + δ2)**
0	n	→1
0	…	…
0	2	Ir1 (δ1, δ2) = a
0	1	Ir2 (δ1, δ2) = b
1	0	Ir3 (δ1, δ2) = c
2	0	Ir4 (δ1, δ2) = d
…	0	…
m	0	→0

Другим речима, метрика стабилности се може представити путем следећег система:

$Ir1\ (\delta 1, \delta 2) = a,$

$Ir2\ (\delta 1, \delta 2) = b,$

$Ir3\ (\delta 1, \delta 2) = c,$

$Ir4\ (\delta 1, \delta 2) = d$, при чему је $1 > a > b > c > d > 0$.

Уколико се у обзир узму вредности Ca и Ce из претходне табеле (Табела 13), добија се систем од четири једначине са две непознате:

$(2 + \delta 1) / (2 + \delta 2) = a,$

$(1 + \delta 1) / (1 + \delta 2) = b,$

$\delta 1 / (1 + \delta 2) = c,$

$\delta 1 / (2 + \delta 2) = d.$

Посматрани систем у општем случају нема решења. Међутим, применом методе најмањих квадрата може се увести појам решења посматраног система[4]. У том смислу, уколико се примени метода најмањих квадрата, при чему је, на пример, a = 0.6, b=0.55, c=0.45 и d=0.4, посматрани систем се своди на облик [Vlajic17]:

$2\delta 1 - \delta 2 = 0,$

$-2\delta 1 - 1.025\delta 2 = 0.205.$

Решења посматраног система су $\delta 1 = 4.1$, $\delta 2 = 8.2$. Другим речима, метрика се може дефинисати на следећи начин [Vlajic17]:

$Ir = (Ce + 4.1) / (Ca + Ce + 8.2)$ или

[4] Ово је једно од могућих решења посматраног система.

Ir = (Ce + δ1) / (Ca + Ce + 2δ1) за δ1 = 4.1. Уколико се уведе замена:

Ce' = Ce + δ1,

Ca' = Ca + δ1, тада је:

Ir = Ce'/(Ca'+Ce'), при чему је Ce' = Ce + δ1, Ca' = Ca + δ1, за δ1 = 4.1.

Наведена софтверска метрика представља побољшање метрике стабилности софтвера[5] с обзиром да узима у обзир релативизацију долазне повезаности (када је одлазна повезаност једнака нули), односно одлазне повезаности (када је долазна повезаност једнака нули) [Vlajic17].

У том смислу, уколико разматрамо стабилност софтверског система у којем постоји одлазна повезаност а не постоји долазна повезаност (представљен је на слици Слика 34), метрика нестабилности ће имати вредност Ir = 5.1 / 9.2 ≈ 0.55 [Vlajic17]. С друге стране, уколико се дода нова одлазна повезаност, метрика ће имати вредност Ir = 6.1 / 10.2 ≈ 0.6. Другим речима, додавањем нове одлазне повезаности (при чему долазна повезаност не постоји) долази до повећања нестабилности посматраног софтверског система.

Такође, уколико разматрамо стабилност софтверског система у којем постоји долазна повезаност а не постоји одлазна повезаност (представљен је на слици Слика 35), метрика нестабилности ће имати вредност Ir = 4.1 / 9.2 ≈ 0.44 [Vlajic17]. С друге стране, уколико се дода нова долазна повезаност, метрика ће имати вредност Ir = 4.1 / 10.2 ≈ 0.4. Другим речима, додавањем нове долазне повезаности (при чему одлазна повезаност не постоји) долази до смањења нестабилности посматраног софтверског система.

[5] У наставку рада ћемо под метриком стабилности софтвера подразумевати побољшану метрику стабилности софтвера I_r.

Интерпретација:

Нестабилност софтвера узима вредности $0 \leq I \leq 1$. Боље је када нестабилност софтвера тежи нули. У том смислу класе унутар пакета не зависе од класа из других пакета, па је посматрани пакет стабилан и једноставан за одржавање. Уколико је нестабилност софтвера већа од нуле, то значи да класе из неког пакет зависе од класа из других пакета, што последично може да доведе до отежаног одржавања посматраног софтверског система.

4.5.11. Веза софтверских метрика са принципима, стратегијама и патернима пројектовања софтвера

У оквиру претходног одељка представљене су различите софтверске метрике. На основу изложеног се може закључити да су оне усмерене на објектно-оријентисане софтверске системе и да се заједно користе у процесу евалуације квалитета софтвера. Посматране софтверске метрике су комплементарне и међусобно се допуњују.

Фокус докторске дисертације је дефинисање концептуалног скупа елемената за побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера. У том смислу, у оквиру докторске дисертације разматрају се софтверске метрике и њихова веза са различитим механизмима са побољшање објектно-оријентисаних софтверских система: општим принципима пројектовања софтвера, принципима објектно-оријентисаног пројектовања софтвера, стратегијама пројектовања софтвера и патернима пројектовања софтвера.

С друге стране, софтверске метрике се оперативно користе за мерење нивоа атрибута квалитета софтвера. Због тога је могуће успоставити везу између софтверских метрика и атрибута квалитета софтвера. Атрибути квалитета софтвера су повезани са моделима квалитета софтвера и стандардима квалитета софтвера, па у том смислу постоји посредна веза између софтверских метрика и модела и стандарда квалитета софтвера. Такође, алати за статичку анализу квалитета софтвера подржавају софтверске

метрике На наредној слици (Слика 36) дат је концептуални приказ повезаности софтверских метрика са принципима, стратегијама и патернима пројектовања софтвера, атрибутима квалитета софтвера и алатима за статичку анализу квалитета софтвера.

Слика 36. Концептуални приказ повезаности софтверских метрика са принципима, стратегијама и патернима пројектовања софтвера, атрибутима квалитета софтвера и алатима за статичку анализу квалитета софтвера

Претходно описане софтверске метрике је могуће повезати са општим принципима пројектовања софтвера. Ови принципи представљају основне градивне елементе који се користе у процесу пројектовања софтвера и биће објашњени у Поглављу 6. У наредној табели (Табела 14) дата веза софтверских метрика са општим принципима пројектовања софтвера.

Табела 14. Веза софтверских метрика са општим принципима пројектовања софтвера

Софтверска метрика	**Општи принцип пројектовања софтвера**
Сложеност пондерисаних метода	Декомпозиција и модуларизација
Број пондерисаних метода класе	Декомпозиција и модуларизација
Циклична сложеност	Декомпозиција и модуларизација
Број одговора класе	Апстракција, Кохезија и повезаност
Недостатак кохезивности метода у	Апстракција, Кохезија и повезаност,

Софтверска метрика	**Општи принцип пројектовања софтвера**
класи	Декомпозиција и модуларизација, Учаурење и сакривање информација
Повезаност објеката	Апстракција, Кохезија и повезаност, Декомпозиција и модуларизација
Дубина стабла наслеђивања	Апстракција, Одвајање интерфејса и имплементације
Број подкласа	Апстракција, Одвајање интерфејса и имплементације
Број наредби у методи	Декомпозиција и модуларизација
Метрика стабилности софтвера	Декомпозиција и модуларизација

С друге стране, посматране софтверске метрике је могуће повезати са принципима објектно-оријентисаног пројектовања софтвера који се користе приликом објектно-оријентисаног програмирања и пројектовања класа. (биће објашњени у Поглављу 6.). У наредној табели (Табела 15) дата веза софтверских метрика са принципима објектно-оријентисаног пројектовања софтвера.

Табела 15. Веза софтверских метрика са принципима објектно-оријентисаног пројектовања софтвера

Софтверска метрика	**Принцип објектно-оријентисаног пројектовања софтвера**
Сложеност пондерисаних метода	Принцип отворено-затворено
Број пондерисаних метода класе	Принцип отворено-затворено
Циклична сложеност	Принцип отворено-затворено
Број одговора класе	Принцип једне одговорности, Лисков принцип замене
Недостатак кохезивности метода у класи	Принцип једне одговорности, Лисков принцип замене, Принцип издвајања интерфејса

Софтверска метрика	**Принцип објектно-оријентисаног пројектовања софтвера**
Повезаност објеката	Принцип једне одговорности, Лисков принцип замене, Принцип инверзије зависности, Принцип додавања зависности, Принцип издвајања интерфејса
Дубина стабла наслеђивања	Принцип отворено-затворено, Лисков принцип замене, Принцип инверзије зависности, Принцип додавања зависности, Принцип издвајања интерфејса
Број подкласа	Принцип отворено-затворено, Лисков принцип замене, Принцип инверзије зависности, Принцип додавања зависности, Принцип издвајања интерфејса
Број наредби у методи	Принцип отворено-затворено
Метрика стабилности софтвера	Принцип инверзије зависности, Принцип додавања зависности

Стратегијама пројектовања софтвера дефинише се стратешки приступ процесу пројектовања софтвера. Описане софтверске метрике је могуће повезати и са стратегијама пројектовања софтвера (биће објашњено у Поглављу 6.), што је и приказано у наредној табели (Табела 16).

Табела 16. Веза софтверских метрика са стратегијама пројектовања софтвера

Софтверска метрика	**Стратегије пројектовања софтвера**
Сложеност пондерисаних метода	Подели и победи, С врха на доле, Одоздо на горе
Број пондерисаних метода класе	Подели и победи, С врха на доле, Одоздо на горе

Софтверска метрика	**Стратегије пројектовања софтвера**
Циклична сложеност	Подели и победи, С врха на доле, Одоздо на горе
Недостатак кохезивности метода у класи	Подели и победи, С врха на доле, Одоздо на горе
Повезаност објеката	Подели и победи
Број одговора класе	Подели и победи
Број наредби у методи	Подели и победи, С врха на доле, Одоздо на горе
Дубина стабла наслеђивања	Подели и победи
Број подкласа	Подели и победи
Метрика стабилности софтвера	Подели и победи

Описане софтверске метрике је, такође, могуће повезати са патернима пројектовања софтвера (биће објашњено у Поглављу 6.), што је и приказано у наредној табели (Табела 17).

Табела 17. Веза софтверских метрика са патернима пројектовања софтвера

Софтверска метрика	**Патерни пројектовања софтвера**
Сложеност пондерисаних метода	Општи облик патерна пројектовања софтвера
Број пондерисаних метода класе	
Циклична сложеност	
Број одговора класе	
Број наредби у методи	
Метрика стабилности софтвера	
Повезаност објеката	
Дубина стабла наслеђивања	
Број подкласа	

4.5.12. Веза софтверских метрика са атрибутима квалитета софтвера

Софтверске метрике је могуће повезати и са атрибутима квалитета софтвера. Оне се оперативно користе у процесу утврђивања вредности атрибута квалитета софтвера. У наредној табели (Табела 18) дат је приказ веза софтверских метрика са атрибутима квалитета софтвера, узимајући у обзир стандарде квалитета софтвера који су описани у Поглављу 3.

Табела 18. Веза софтверских метрика са атрибутима квалитета софтвера

Софтверска метрика	**ISO/IEC 9126-1 (Модел интерног квалитета софтвера)**	**ISO/IEC 25010 (Модел квалитета софтверског производа)**
Сложеност пондерисаних метода	Функционалност, Одржавање	Функционална подобност, Одржавање
Број пондерисаних метода класе	Функционалност, Одржавање	Функционална подобност, Одржавање
Циклична сложеност	Функционалност, Одржавање	Функционална подобност, Одржавање
Број одговора класе	Функционалност, Одржавање	Функционална подобност, Одржавање
Недостатак кохезивности метода у класи	Функционалност, Одржавање	Функционална подобност, Одржавање
Повезаност објеката	Функционалност, Одржавање	Функционална подобност, Одржавање
Дубина стабла наслеђивања	Функционалност, Употребљивост, Одржавање	Функционална подобност, Употребљивост, Одржавање
Број подкласа	Функционалност, Употребљивост, Одржавање	Функционална подобност, Употребљивост, Одржавање

Софтверска метрика	**ISO/IEC 9126-1 (Модел интерног квалитета софтвера)**	**ISO/IEC 25010 (Модел квалитета софтверског производа)**
Број наредби у методи	Функционалност, Одржавање	Функционална подобност, Одржавање
Метрика стабилности софтвера	Функционалност, Одржавање	Функционална подобност, Одржавање

4.5.13. Веза софтверских метрика са алатима за статичку анализу квалитета софтвера

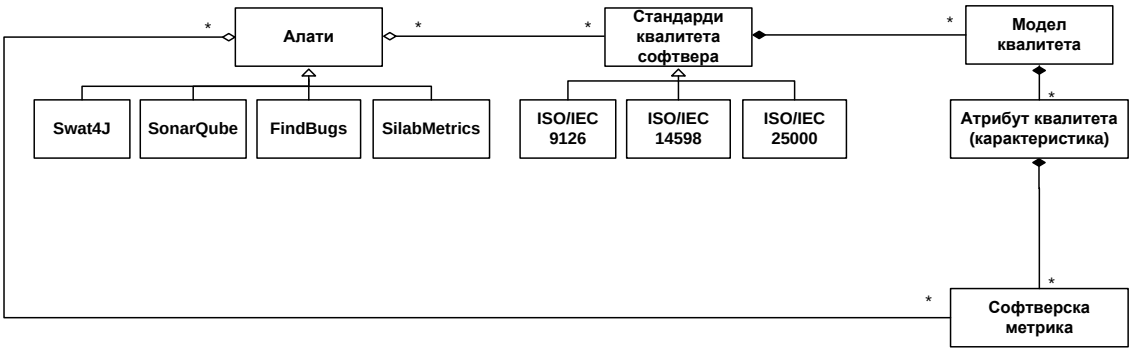
Софтверске метрике је могуће повезати и са алатима за статичку анализу квалитета софтвера. У наредном поглављу биће описани Swat4J, SonarQube, FindBugs и SilabMetrics алати за статичку анализу квалитета софтвера. Ови алати се оперативно користе у процесу евалуације квалитета софтвера. Стога је у наредној табели (Табела 19) дата веза приказаних софтверских метрика са поменутим алатима за статичку анализу квалитета софтвера.

Табела 19. Веза софтверских метрика са алатима за статичку анализу квалитета софтвера

Софтверска метрика	**Алат за статичку анализу квалитета софтвера**
Сложеност пондерисаних метода	Swat4J, SonarQube, SilabMetrics
Број пондерисаних метода класе	Swat4J
Циклична сложеност	Swat4J, SonarQube, SilabMetrics
Број одговора класе	Swat4J
Недостатак кохезивности метода у класи	Swat4J
Повезаност објеката	Swat4J, SonarQube, SilabMetrics
Дубина стабла наслеђивања	Swat4J, SonarQube, SilabMetrics
Број подкласа	Swat4J, SonarQube, SilabMetrics
Број наредби у методи	Swat4J
Метрика стабилности софтвера	SilabMetrics

5. Преглед и анализа постојећих алата за статичку анализу квалитета софтвера

У оквиру овог поглавља биће представљени и анализирани алати за статичку анализу квалитета софтвера. Алати могу бити формално засновани на стандардима квалитета софтвера. У том смислу ће најпре бити разматрана веза алата за статичку анализу квалитета софтвера са стандардима квалитета софтвера. Затим ће бити дат преглед Swat4J, SonarQube и FindBugs алата за статичку анализу квалитета софтвера који се оперативно користе у процесу евалуације квалитета софтверских система. Затим ће бити приказан SilabMetrics алат за статичку анализу квалитета софтвера који је развијен на Факултету организационих наука. Након прегледа алата биће дефинисани критеријуми на основу којих ће бити извршена компаративна анализа алата за статичку анализу квалитета софтверских система. Концептуални приказ поглавља дат је на наредној слици (Слика 37).



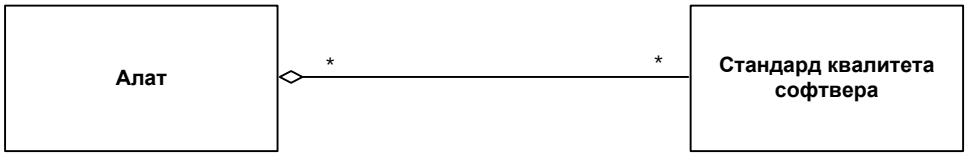
Слика 37. Алати за статичку анализу квалитета софтвера

5.1. Значај алата за статичку анализу квалитета софтвера и њихова веза са стандардима квалитета софтвера

У оквиру претходних поглавља описани су ISO/IEC 9126, ISO/IEC 14598, ISO/IEC 25000 стандарди квалитета софтвера. Стандардима квалитета софтвера дефинишу се модели квалитета софтвера. Затим се за сваки модел квалитета софтвера наводе атрибути квалитета, док су посебним деловима стандарда дефинисане софтверске метрике које се користе за утврђивање

вредности атрибута квалитета. Атрибути квалитета могу се користити за дефинисање нивоа квалитета софтверског производа, али и за дефинисање квалитета процеса развоја софтвера [Liu10].

Алати за статичку анализу квалитета софтвера су у непосредној вези са стандардима квалитета софтвера. Наиме, алати за статичку анализу квалитета софтвера могу обезбедити подршку за неки стандард квалитета софтвера (алат може да имплементира пуну подршку за један или више стандарда квалитета софтвера; такође, алат може бити заснован на принципима неког стандарда квалитета софтвера). С друге стране, један стандард квалитета софтвера може бити подржан од стране више алата за статичку анализу квалитета софтвера, што је и приказано на наредној слици (Слика 38).



Слика 38. Однос између алата за статичку анализу квалитета софтвера и стандарда квалитета софтвера

Стандардом квалитета софтвера дефинише се модел квалитета софтвера (нпр. ISO/IEC 9126-1 модел употребног квалитета, ISO/IEC 9126-1 модел екстерног и интерног квалитета, ISO/IEC 25010 модел употребног квалитета, ISO/IEC 25010 модел квалитета софтверског производа итд.). Моделом квалитета софтвера специфицирају се атрибути квалитета софтверског система, при чему се свака карактеристика квалитета (у зависности од примењеног стандарда квалитета) може даље декомпоновати на подкарактеристике квалитета. За оперативно утврђивање задовољења атрибута квалитета софтвера користе се софтверске метрике које су усмерене на објективно мерење атрибута квалитета софтвера и процеса његовог развоја [Sommerville11]. С друге стране, на основу атрибута квалитета је могуће извршити евалуацију квалитета софтверског система. Другим речима, коришћењем софтверских метрика се посредно врши оцена

квалитета софтверског система. Веза између алата за статичку анализу квалитета софтверског система, стандарда квалитета софтвера, модела квалитета софтвера и софтверских метрика приказана је на наредној слици (Слика 39).

Слика 39. Веза између алата за статичку анализу квалитета софтвера, стандарда квалитета софтвера, модела квалитета софтвера и софтверских метрика

Као што је већ напоменуто, алати за статичку анализу квалитета софтвера се оперативно користе за утврђивање нивоа квалитета посматраног софтверског система. У том смислу је значај заснованости алата на стандардима квалитета софтвера вишеструк. Најпре, једна од најважнијих карактеристика процеса развоја софтвера је производња квалитетног софтвера који је у складу са дефинисаним захтевима. Коришћењем алата за статичку анализу квалитета софтвера инжењери за развој софтвера, развојни тим и менаџмент имају добар увид у квалитет софтвера, боље се разумеју и лакше сарађују. У том смислу се може рећи да стандарди квалитета софтвера представљају нешто око чега су сагласни и окупљени сви учесници у процесу развоја софтвера.

Коришћењем алата који су засновани на стандардима квалитета софтвера је, такође, омогућено брзо увођење нових софтверских инжењера у процес развоја софтвера: софтверски инжењери су усмерени да пишу програмски код који је у складу са дефинисаним моделом квалитета. Због тога је

квалитет софтвера препознат као веома важна карактеристика процеса развоја софтвера.

Примена алата за статичку анализу квалитета софтвера у процесу развоја софтвера помаже у креирању софтверског система који је сагласан са дефинисаним моделом квалитета софтвера тј. у креирању софтверског система чији атрибути квалитета имају високе вредности. Поред тога, омогућено је уочавање и отклањање неусаглашености са дефинисаним моделом квалитета у раним фазама процеса развоја софтвера, када је грешке много лакше уочити и исправити. Алати се најчешће интегришу са окружењима за развој софтвера што значајно олакшава уочавање и исправљање грешака.

Усаглашеност процеса развоја софтверског система са дефинисаним моделом квалитета последично доводи до креирања софтверског система чији су атрибути квалитета задовољени. На тај начин се добија софтверски систем који је ефикасан, добро пројектован, који се лако одржава и може се поново користити. Ово су неки од атрибута квалитета који могу бити дефинисани моделом квалитета софтвера.

Као што је раније напоменуто, стандардима квалитета софтвера дефинише се модел квалитета који се најчешће може користити за евалуацију квалитета сваког софтверског система. С друге стране, предефинисани модел квалитета је оперативно подржан алатима за статичку анализу квалитета софтвера. На тај начин је успостављена непосредна веза између стандарда квалитета софтвера и алата за статичку анализу квалитета софтвера. Међутим, софтверски системи који се развијају разликују се по многим критеријумима:

- Програмским парадигмама које се користе у процесу развоја софтверског система,
- Програмским језицима у оквиру којих се врши имплементација софтверског система,

- Софтверским технологијама и оквирима (енг. framework) који се користе у процесу развоја софтвера.
- Развојним окружењима која се користе у процесу развоја софтвера,
- Примењеним стратегијама, моделима и методама процеса животног циклуса развоја софтвера,
- Артифактима који настају у процесу развоја софтвера,
- Врсти софтверског система,
- Намени софтверског система,
- Домену проблема који софтверски систем решава,
- Корисницима софтверског система итд.

У том смислу предефинисани модел квалитета не мора бити одговарајући за евалуацију квалитета сваког софтверског система. За посматрани софтверски систем поједини атрибути квалитета могу бити значајнији у односу на друге атрибуте квалитета. Такође, неки атрибути квалитета не морају уопште имати утицај на квалитет посматраног софтверског система. Стога се алатима за статичку анализу квалитета софтвера може извршити измена предефинисаног модела квалитета софтвера (у смислу измене атрибута квалитета софтвера, граничних вредности софтверских метрика, везе атрибута квалитета софтвера са софтверским метрикама, као и других релевантних параметара који се односе на посматрани модел квалитета софтвера).

Уколико алати подржавају промену параметара предефинисаног модела квалитета, могуће је вршити постоптималну анализу квалитета софтверског система. У том смислу је могуће утврдити како промена појединачног параметра (или групе параметара) модела квалитета утиче на квалитет софтверског система у целини. Такође је могуће утврдити у којим границама се појединачни параметар (или група параметара) може кретати а да квалитет софтверског система остане задовољен. Другим речима, могуће је правити баланс у смислу задовољења атрибута квалитета софтвера

узимајући у обзир значај атрибута квалитета за посматрани софтверски систем, односно узимајући у обзир специфично значење квалитета[6].

На основу изложеног се може рећи да су стандарди квалитета софтвера непосредно повезани са алатима за статичку анализу квалитета софтвера. Алати за статичку анализу квалитета софтвера могу бити засновани на стандардима квалитета софтвера и оперативно се користе у процесу развоја посматраног софтверског система. Они се, најчешће, користе у комбинацији, комплементарни су (међусобно се допуњују) и њихово заједничко коришћење доприноси повећању нивоа квалитета софтверског система који се развија.

Алати за статичку анализу квалитета софтвера и стандарди квалитета софтвера су нашли велику примену у процесу развоја софтвера. Због тога ће у наставку овог поглавља најпре бити представљени Swat4J, SonarQube, FindBugs и SilabMetrics алати за статичку анализу програмског кода који могу бити засновани на неком стандарду квалитета софтвера. Након тога ће бити дефинисани критеријуми њиховог поређења и на основу њих ће бити извршена компаративна анализа алата.

5.2. Преглед алата за статичку анализу квалитета софтвера

Као што је раније напоменуто, алати за статичку анализу квалитета софтвера су оперативно усмерени на евалуацију квалитета посматраног софтверског система. Ови алати формално могу бити засновани на неком стандарду квалитета софтвера. С обзиром да у развоју стандарда квалитета софтвера учествују различите заинтересоване стране (академска заједница, индустрија, удружења и друге интересне групе), као и да се стандарди квалитета софтвера могу с великим успехом примењивати у различитим индустријским гранама, заснованост алата на неком стандарду квалитета

[6] Под специфичним значењем квалитета подразумева се одређивање вредности софтверских метрика и атрибута квалитета у складу са потребама софтверског система чији се развој врши.

софтвера је веома пожељна карактеристика. На тај начин се повећава могућност практичне примене и усвајања посматраних стандарда квалитета софтвера и алата за статичку анализу квалитета у процесу евалуације квалитета софтверских система. Њиховим коришћењем се непосредно утиче на повећање нивоа квалитета софтверског система.

У наставку поглавља биће представљени Swat4J, SonarQube, FindBugs и SilabMetrics алати за статичку анализу квалитета софтвера који могу бити засновани на неком стандарду квалитета софтвера. Софтверски пакет Swat4J представља алат за статичку анализу квалитета софтверског система који је реализован у програмском језику Java. Овај алат је заснован на ISO/IEC 9126 стандарду квалитета софтвера. За разлику од њега, софтверским пакетом SonarQube се може вршити евалуација квалитета софтверских система који су реализовани у различитим програмским језицима (нпр. у програмским језицима Java, C#, C/C++, Cobol итд.). Он је, такође, заснован на ISO/IEC 9126 стандарду квалитета софтвера. Софтверски пакет FindBugs, за разлику од Swat4J и SonarQube алата, не садржи модел квалитета који је експлицитно заснован на неком стандарду квалитета софтвера. Он је нашао велику практичну примену у евалуацији квалитета софтверских система који су реализовани у програмском језику Java. Софтверски пакет SilabMetrics је алат за статичку анализу софтверског система који је заснован на SonarQube алату и подржава ISO/IEC 9126 стандард квалитета софтвера. На основу изнесеног се може закључити да су посматрани алати за статичку анализу оперативно усмерени ка евалуацији квалитета софтверског производа.

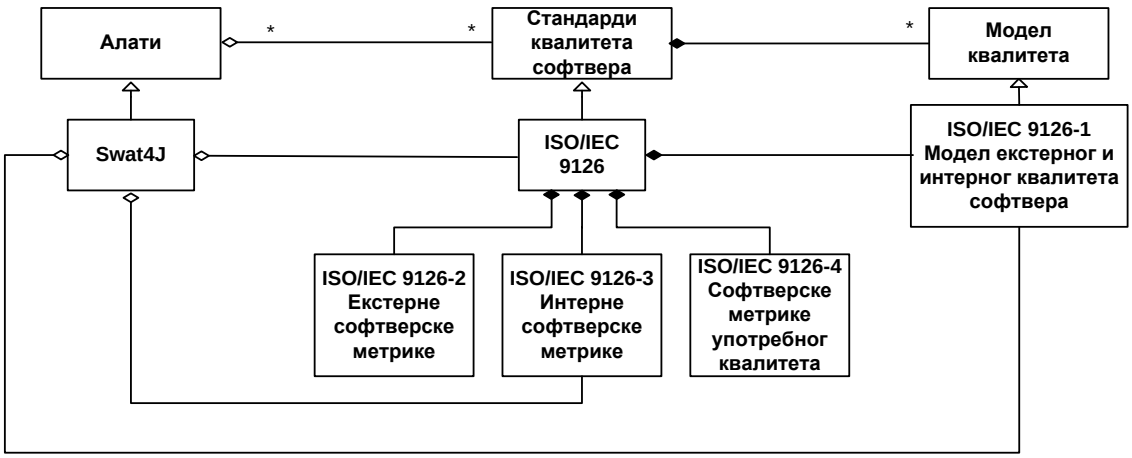
За сваки од поменутих алата ће најпре бити представљене најважније карактеристике и начин рада посматраног алата. Затим ће бити објашњена његова веза са стандардима квалитета софтвера (уколико је алат формално заснован на неком стандарду квалитета). У том смислу биће извршен приказ модела квалитета и софтверских метрика које су подржане од страна алата за статичку анализу квалитета софтвера. Преглед алата за статичку анализу квалитета софтвера представља полазну основу за извршавање

компаративне анализе алата. Због тога су у наставку поглавља описане најважније карактеристике Swat4J, SonarQube, FindBugs и SilabMetrics алата за статичку анализу квалитета софтвера.

5.2.1. Swat4J

Swat4J је софтверски алат за статичку анализу програмског кода софтверског система који је написан у програмском језику Java. У том смислу алат се може користити у процесу развоја софтвера, за анализу квалитета посматраног софтверског производа.

Swat4J је заснован на ISO/IEC 9126 стандарду квалитета софтвера. Његов модел квалитета формиран је на основу модела екстерног и интерног квалитета софтвера који је дефинисан ISO/IEC 9126-1 стандардом квалитета. Такође, Swat4J подржава интерне софтверске метрике које су дефинисане ISO/IEC 9126-3 стандардом квалитета софтвера [Milico7], што је и приказано на наредној слици (Слика 40). У том смислу се може рећи да су алатом подржане софтверске метрике које се непосредно односе на софтверски производ.



Слика 40. Модел квалитета софтвера у Swat4J алату

Модел квалитета који је подржан овим алатом дефинише следеће атрибуте квалитета софтвера [Milico7]:

- Тестирање (енг. Testability),

- Квалитет пројектовања (енг. Design Quality),
- Перформансе (енг. Performance),
- Разумљивост (енг. Understandability),
- Одржавање (енг. Maintainability),
- Поновно коришћење (енг. Reusability).

Према предефинисаном моделу квалитета алат садржи преко 30 софтверских метрика и преко 100 правила најбољих пракси у писању програмског кода која се примењују у индустрији [Lavalo8]. У оквиру софтверског пакета метрике су класификоване у следеће категорије [Milico7]:

- Објектно-оријентисане софтверске метрике - Софтверске метрике ове категорије усмерене су на мерење квалитета објектно-оријентисаног пројектовања, односно на утврђивање веза и зависности које су успостављене између објеката посматраног софтверског система. У том смислу се у овој групи налазе софтверске метрике које су описане у оквиру претходног одељка а односе се на објектно-оријентисане концепте. У ову категорију, између осталих, спадају метрике *Сложеност пондерисаних метода*, *Недостатак кохезивности метода у класи*, *Повезаност објеката*, *Дубина стабла наслеђивања* и *Број подкласа*.
- Метрике за одређивање индекса сложености - У оквиру ове категорије налазе се софтверске метрике које су усмерене на одређивање индекса сложености посматраног софтверског система или компоненти посматраног софтверског система. У оквиру ове категорије налазе се софтверске метрике као што су *Циклична сложеност* (такође објашњена у оквиру претходног поглавља) и *Халстедова сложеност*.
- Метрике за одређивање индекса одржавања - У оквиру ове категорије налази се софтверска метрика *Индекс одржавања* која је усмерена на

утврђивање једноставности одржавања посматраног софтверског система.

- Метрике које се односе на програмски код - Софтверске метрике ове категорије усмерене су на програмски код и омогућавају специфичан увид у квалитет програмског кода посматраног софтверског система. Оне се, најчешће, користе у комбинацији са објектно-оријентисаним софтверским метрикама и софтверским метрикама за одређивање индекса сложености. С обзиром на њихову специфичност, оне се могу посматрати на различитим нивоима: на нивоу методе, на нивоу класе, на нивоу датотеке и на нивоу пројекта (у том смислу се, на пример, може посматрати повезаност објеката на нивоу класе, на нивоу датотеке и на нивоу целог пројекта). Ове метрике су веома специјализоване и омогућавају увид у квалитет посматраног софтверског система.

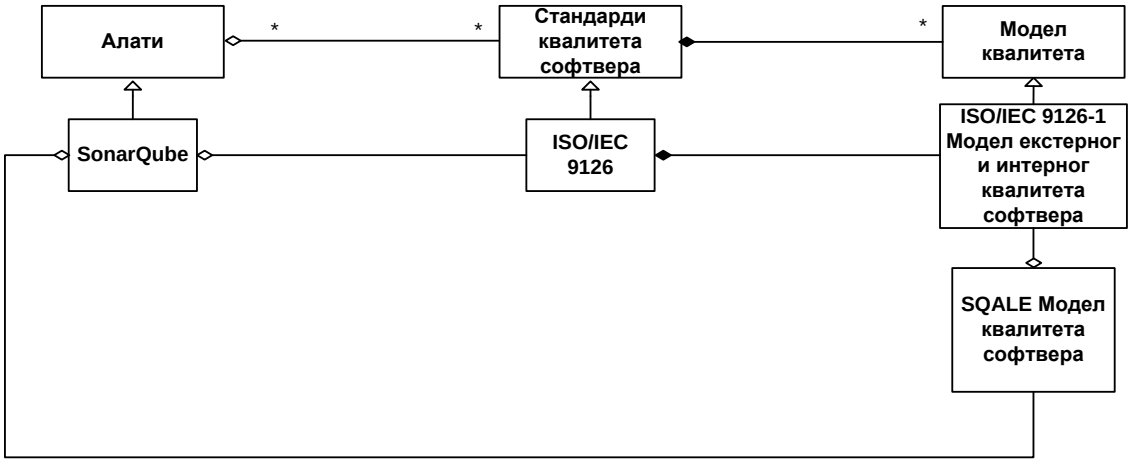
На основу вредности софтверских метрика према предефинисаном моделу квалитета врши се оцена атрибута квалитета софтверског система и даје графички приказ квалитета софтверског система. У том смислу даје се приказ очекиваних и остварених вредности атрибута квалитета па је, у складу са тиме, могуће извршити даљу појединачну анализу атрибута квалитета посматраног софтверског система.

Као што је већ напоменуто, алат Swat4J садржи предефинисани модел квалитета који је формиран на основу ISO/IEC 9126-1 и ISO/IEC 9126-3 стандарда квалитета софтвера. Поред тога, софтверски пакет Swat4J омогућава конфигурисање модела квалитета (у смислу атрибута квалитета софтверског система, софтверских метрика и веза атрибута квалитета софтверског система са софтверским метрикама). На тај начин се омогућава дефинисање специфичног значења квалитета софтвера за посматрани софтверски производ.

5.2.2. SonarQube

SonarQube [SonarQube] је алат за статичку анализу програмског кода софтверског система. Алат подржава велики број програмских језика (нпр. програмске језике Java, C#, C/C++, Cobol итд.). Поред тога алат се може проширити укључивањем додатака (енг. plugins) чиме се могу добити додатне функционалности. SonarQube представља софтвер отвореног кода што у значајној мери доприноси његовој популарности у заједници инжењера који се баве развојем софтвера.

SonarQube је заснован на ISO/IEC 9126 стандарду квалитета софтвера. Наиме, алат садржи SQALE модел квалитета софтвера [Letouzey12] [Letouzey12b]. Овај модел је формиран на основу модела екстерног и интерног квалитета софтвера који је дефинисан ISO/IEC 9126-1 стандардом квалитета што је и приказано на наредној слици (Слика 41).



Слика 41. Модел квалитета софтвера у SonarQube алату

Међутим, SQALE модел квалитета се у извесној мери разликује од модела екстерног и интерног квалитета који је дефинисан ISO/IEC 9126-1 стандардом. Наиме, SQALE модел не укључује све карактеристике и подкарактеристике из ISO/IEC 9126-1 модела. У том смислу се може рећи да модел квалитета који је подржан SonarQube алатом дефинише следеће атрибуте квалитета [Letouzey12]:

- Тестирање (енг. Testability),
- Поузданост (енг. Reliability),
- Промењљивост (енг. Changeability),
- Ефикасност (енг. Efficiency),
- Сигурност (енг. Security),
- Одржавање (енг. Maintainability),
- Преносивост (енг. Portability),
- Поновно коришћење (енг. Reusability).

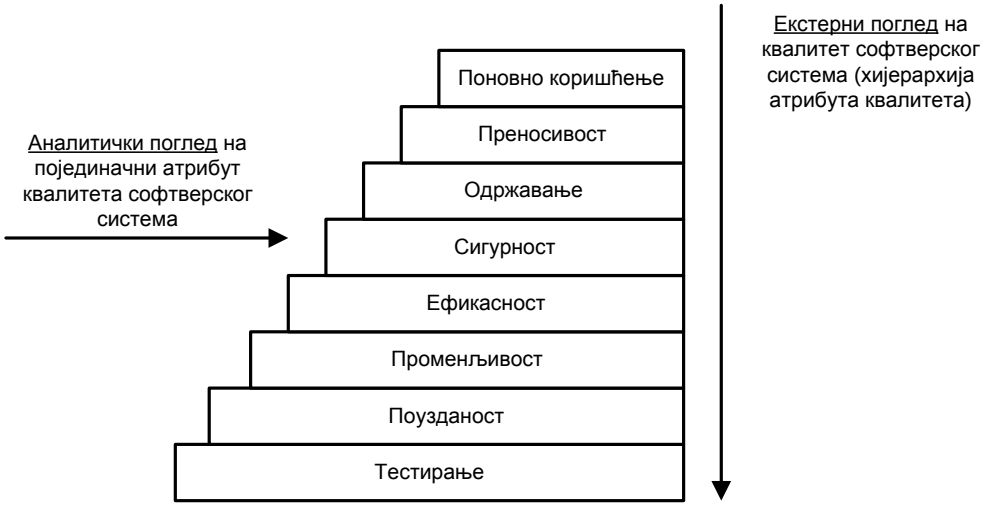
За сваки атрибут квалитета везано је више подкарактеристика. С друге стране, за једну подкарактеристику везано је више софтверских метрика које се у оквиру овог алата називају правила. Правилом се дефинише поступак мерења неке вредности али се такође дефинишу и граничне вредности за посматрано правило. Уколико измерена вредност није у оквиру граничних вредности правило је нарушено. За разлику од модела квалитета који је дефинисан ISO/IEC 9126-1 стандардом квалитета у оквиру кога се сви атрибути квалитета налазе на истом нивоу, SonarQube алат и SQALE модел атрибуте квалитета организују према хијерархији [Letouzey12b] као што је и приказано на наредној слици (Слика 42). У том смислу се модел квалитета софтвера може посматрати кроз две димензије:

- Хоризонтално

  На овај начин се врши посматрање појединачног атрибута квалитета посматраног софтверског система. У том смислу је омогућен аналитички поглед на појединачне атрибуте квалитета софтверског система [Letouzey12b].
- Вертикално

  На овај начин се врши посматрање свих атрибута квалитета посматраног софтверског система. У том смислу омогућен је екстерни поглед на атрибуте квалитета софтверског система (у контексту хијерархије атрибута квалитета софтверског система), тј. посматрање

софтверског система као целине [Letouzey12b]. Екстерни поглед је сумарни поглед на атрибуте квалитета посматраног софтверског система.



Слика 42. Хијерархија атрибута квалитета у оквиру модела квалитета SonarQube алата [Letouzey12b]

Као што се може приметити атрибути су организовани хијерархијски, према нивоу значајности. У том смислу атрибути квалитета софтвера на вишем нивоу хијерархије зависе од атрибута квалитета софтвера на нижем нивоу хијерархије. Уколико посматрани атрибут квалитета није у складу са дефинисаним вредностима нарушени су и сви атрибути квалитета на вишим нивоима хијерархије који зависе од њега. С друге стране, предуслов за задовољење посматраног атрибута квалитета је да су задовољени сви атрибути квалитета на нижим нивоима хијерархије.

На првом нивоу хијерархије се налази атрибут квалитета *Тестирање*. Ово је најзначајнији атрибут квалитета: уколико је он нарушен нарушени су сви остали атрибути квалитета. Изнад атрибута квалитета *Тестирање* налазе се следећи атрибути: *Поузданост*, *Променљивост*, *Ефикасност*, *Сигурност*, *Одржавање*, *Преносивост* и *Поновно коришћење*, респективно. На пример, задовољење атрибута квалитета *Поновно коришћење* није од значаја уколико је софтверски систем тешко одржавати (незадовољен је атрибут квалитета

Одржавање) или тестирати (незадовољен је атрибут квалитета *Тестирање*), односно уколико су незадовољени атрибути квалитета софтвера који се налазе на нижим нивоима хијерархије. Стога је веома важно да сви атрибути квалитета који су део хијерархије буду задовољени.

Алат је усмерен на рачунање техничког дуга (енг. Technical Debt), односно времена које је потребно за исправку неусаглашености које постоје у посматраном софтверском систему. Технички дуг представља суму трошкова исправљања (енг. Remediation Cost) у смислу времена (или новчаних средстава) који су потребни да би се настале неусаглашености елиминисале. Неусаглашености представљају нарушена правила која су дефинисана у оквиру модела квалитета софтвера. На основу појединачних трошкова санације могуће је израчунати различите индексе квалитета (за сваки појединачни атрибут квалитета софтверског система, као и глобални индекс квалитета софтверског система) и различите индикаторе квалитета (одређивање ранга пројекта у смислу квалитета софтвера, као и израда пирамида квалитета) [Letouzey12b]. На тај начин се обезбеђује детаљно извештавање о квалитету посматраног софтверског система.

Предефинисани модел квалитета садржи преко 100 правила. За свако правило дефинисана је гранична вредност и ниво значајности посматраног правила, као и трошкови исправљања тог правила.

SonarQube алат се може интегрисати са развојним окружењима што у значајној мери олакшава његову примену. Уз то алат подржава интеграцију са алатима за управљање софтверским пројектима (нпр. Apache Maven, Apache Ant, JIRA итд.), као и алатима за управљање верзијама програмског кода (нпр. SVN, Git, Mercurial итд.). Стога се може рећи да се алат може интегрисати у процес развоја софтвера. Такође је дозвољено писање додатака чиме се на једноставан начин могу креирати нова проширења која је могуће користити приликом статичке анализе квалитета софтвера. Штавише, алат подржава интеграцију са другим алатима за статичку

анализу квалитета софтвера (у себи интегрише алате као што су FindBugs, PMD и CheckStyle) тако да се у том смислу може посматрати као платформа за статичку анализу квалитета софтвера.

Овај софтверски пакет такође дозвољава измену предефинисаног модела квалитета софтвера. У том смислу се може извршити измена предефинисаног модела квалитета и на неком нивоу хијерархије се може поставити други атрибут квалитета (у односу на предефинисани модел), при чему се и даље поштује правило да атрибут квалитета који се налази на нижем нивоу хијерархије утиче на све атрибуте квалитета који се налазе на вишим нивоима хијерархије.

С обзиром да се у оквиру SQALE модела квалитета велика важност придаје атрибуту квалитета *Тестирање* (налази се на првом нивоу хијерархије у моделу квалитета и од њега зависе сви остали атрибути квалитета софтвера), може се рећи да су SQALE модел квалитета и SonarQube алат погодни за примену у методама развоја софтвера које користе стратегију вођену тестовима. У том смислу намеће се употреба модела квалитета у оквиру агилних метода развоја софтвера али се, с великим успехом, може користити и у оквиру других метода развоја софтвера [Letouzey12b].

5.2.3. FindBugs

FindBugs [FindBugs] је софтверски алат за статичку анализу програмског кода софтверског система који је написан у програмском језику Java. За разлику од раније поменутих алата који непосредно анализирају изворни програмски код, FindBugs алат врши анализу компајлираног Java програмског кода који се назива бајткод (енг. Java Bytecode). С обзиром да је бајткод платформски независан (може се интерпретирати на било којем оперативном систему под условом да постоји одговарајућа Java виртуелна машина) статичку анализу софтверског система је могуће извршити без постојања програмског кода, на било којој платформи.

Алат се може проширити укључивањем додатака (енг. plugins) чиме се могу добити додатне функционалности. FindBugs представља софтвер отвореног кода и реализован је као академски пројекат у оквиру Универзитета Мериленд. Користи се за статичку анализу софтвера у оквиру великог броја пројеката отвореног кода и финансијски је подржан од стране великих софтверских компанија и институција (нпр. компанија Google, Национална фондација за науку и Универзитет Мериленд дају подршку овом пројекту) што у значајној мери доприноси његовој популарности у заједници инжењера који се баве развојем софтвера.

За разлику од претходно поменутих алата FindBugs не садржи модел квалитета који је експлицитно заснован на неком стандарду. Он је, пре свега, усмерен на оперативно откривање грешака у програмском коду па су у том смислу дефинисани обрасци грешака (енг. Bug Patterns). Сваки образац грешке припада једној категорији, при чему једна категорија садржи више образаца грешака. С друге стране, алат садржи већи број категорија што је и приказано на наредној слици (Слика 43).

Слика 43. Модел квалитета софтвера у FindBugs алату

Алат садржи преко 400 образаца грешака које може да уочи. Детаљан списак и опис образаца грешака може се пронаћи на [FindBugs]. Они су разврстани у следеће категорије:

- Лоша пракса (енг. Bad Practice),
- Тачност (енг. Correctness),
- Сумњив код (енг. Dodgy code),
- Експерименталност (енг. Experimental),
- Интернационализација (енг. Internationalization),

- Рањивост при злонамерном коду (енг. Malicious code vulnerability),
- Вишенитна тачност (енг. Multithreaded correctness),
- Перформансе (енг. Performance),
- Сигурност (енг. Security).

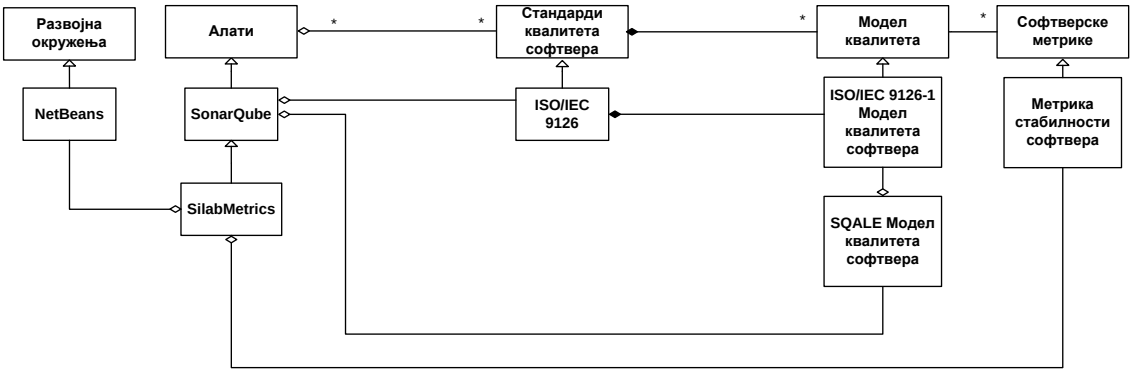
FindBugs алат користи синтаксно поклапање програмског кода и анализу тока програма као технике за уочавање проблема [Rutar04]. У том смислу се може рећи да је алат усмерен на уочавање идиома који представљају потенцијалне грешке у програмском коду посматраног софтверског система [Hovemeyer04]. Другим речима, алат је усмерен на оперативно одређење квалитета софтверског производа (садржи метрике софтверског производа које су усмерене на програмски код).

Поред тога алат се успешно интегрише са многим развојним окружењима. Уз то алат подржава интеграцију са алатима за управљање софтверским пројектима (Apache Maven и Apache Ant). На тај начин је омогућена његова примена у процесу развоја софтвера: може се извршити статичка анализа програмског кода нове функционалности (или функционалности која се мења) док је у фази израде. Другим речима, омогућава се анализа функционалности пре него што постану саставни део софтверског система који је видљив целом развојном тиму. Такође је дозвољено писање додатака чиме се на једноставан начин могу креирати нови обрасци грешака који ће бити саставни део модела квалитета софтвера.

5.2.4. SilabMetrics

SilabMetrics [Milic17] је алат за статичку анализу програмског кода софтверског система који је имплементиран у програмском језику Java. Алат је развијен у оквиру Лабораторије за софтверско инжењерство Факултета организационих наука (Универзитет у Београду). Алат је првенствено намењен учењу и упознавању основних концепата који се односе на стандарде квалитета софтвера, моделе квалитета софтвера и софтверске метрике.

SilabMetrics алат је развијен као надоградња програмског пакета SonarQube и заснован је на ISO/IEC 9126 стандарду квалитета софтвера [Milic17]. Као што је раније поменуто, SonarQube алат садржи SQALE модел квалитета софтвера који је заснован на ISO/IEC 9126 моделу квалитета софтвера [Letouzey12][Letouzey12b]. SilabMetrics алат у позадини користи SonarQube алат за статичку анализу квалитета софтвера. Поред тога, у оквиру алата је имплементирана подршка за метрику стабилности софтвера[7], као и интеграција са NetBeans развојним окружењем, што је и приказано на наредној слици (Слика 44).



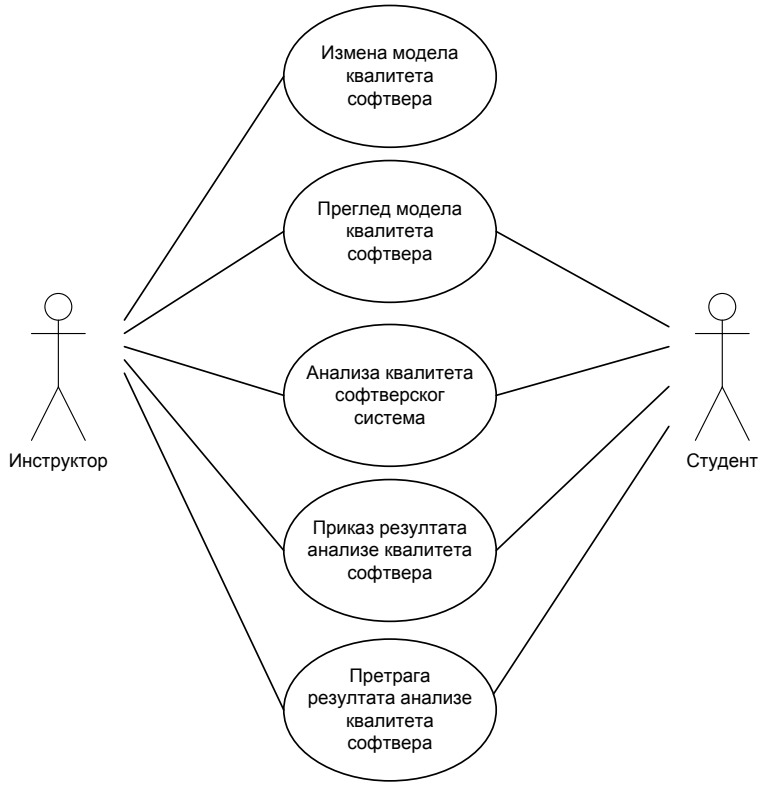
Слика 44. Модел квалитета софтвера у SilabMetrics алату

С обзиром на непосредну повезаност са SonarQube софтверским пакетом, SilabMetrics дефинише следеће атрибуте квалитета софтвера [Milic17]:

- Променљивост (енг. Changeability),
- Ефикасност (енг. Efficiency),
- Одржавање (енг. Maintainability),
- Преносивост (енг. Portability),
- Поузданост (енг. Reliability),
- Поновно коришћење (енг. Reusability),
- Сигурност (енг. Security),
- Тестирање (енг. Testability).

[7] Метрика стабилности софтвера објашњена је у Поглављу 4.

Првенствени корисници алата су инструктори и студенти. Коришћењем алата инструктори могу да анализирају програмски код студената и на тај начин прате процес учења [Milic17]. С друге стране, студенти могу да користе алат како би анализирали програмски код и на тај начин добили повратне информације о његовом квалитету [Milic17]. На наредној слици (Слика 45) приказан је скуп функција које подржава SilabMetrics алат. SilabMetrics алат омогућава прилагођавање предефинисаног модела квалитета софтвера. На тај начин могуће је дефинисати специфично знање квалитета софтвера како би се остварили циљеви учења [Milic17]. На пример, уколико се разматра атрибут квалитета Одржавање, могуће је конфигурисати софтверске метрике које се односе на повезаност објеката, дубину стабла наслеђивања и метрику стабилности софтвера, док се остале софтверске метрике могу искључити. Променом програмског кода софтверског система долази и до промене вредности посматраних метрика, што последично доводи и до лакшег одржавања и надоградње софтверског система. Са слике се може уочити да могућност измене модела квалитета имају само инструктори. С друге стране, студенти и инструктори имају могућност прегледа модела квалитета софтвера, анализе квалитета софтверског система, приказа резултата анализе квалитета софтвера и претраге резултата анализе квалитета софтвера. Важно је напоменути да исте функције подржава и SonarQube алат.



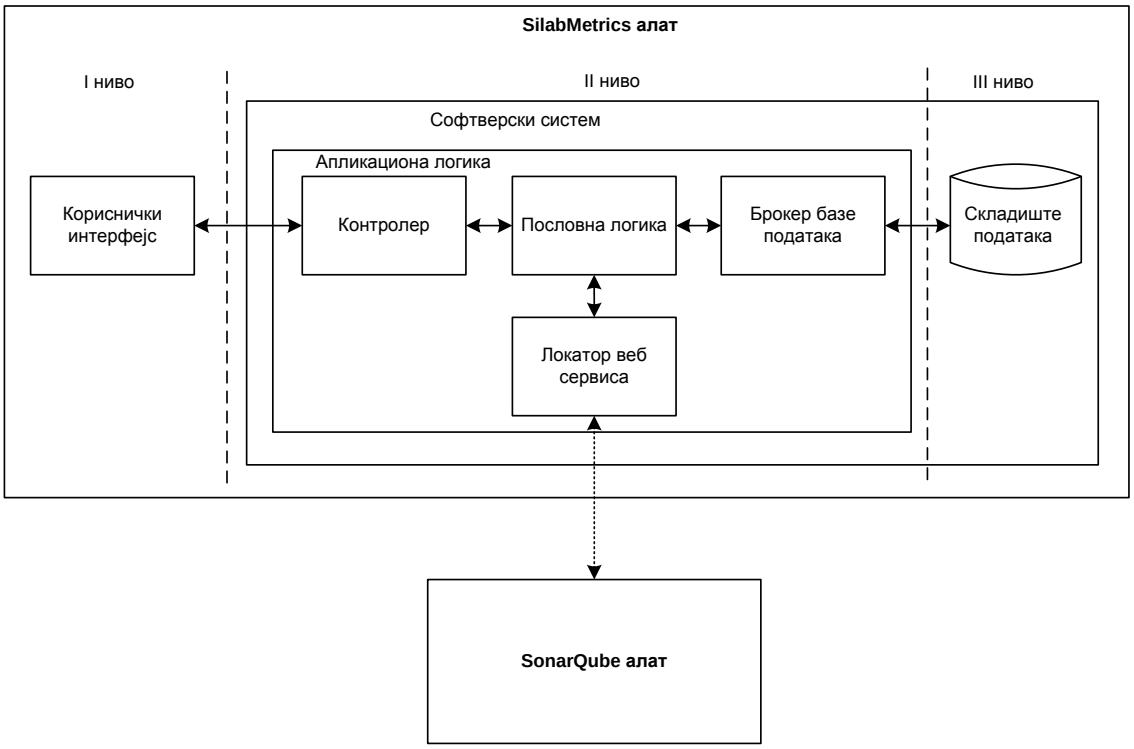
Слика 45. Функције које подржава алат SilabMetrics

На наредној слици (Слика 46) приказана је архитектура SilabMetrics алата. Примећује се да алат развијен коришћењем тронивојске архитектуре:

- Ниво корисничког интерфејса који омогућава преглед модела квалитета софтвера, измену модела квалитета софтвера, као и анализу квалитета софтверског система.
- Ниво апликационе логике садржи контролер, пословну логику којом се реализује понашање софтверског система и брокер базе података. Контролер од корисничког интерфејса прихвата захтев за извршавање одређене операције (нпр. преглед модела квалитета софтвера, измена модела квалитета софтвера, измена софтверске метрике и сл.) и прослеђује га до остатка софтверског система. Такође, Контролер је задужен да резултате извршења операције врати до корисничког интерфејса. У оквиру пословне логике имплементирана је метрика стабилности софтвера, као и позив брокера базе података. Такође, путем пословне логике, преко

компоненте *Локатор веб сервиса*, врши се непосредна интеграција са SonarQube алатом за статичку анализу квалитета софтвера. Путем брокера базе података реализоване су операције које се односе на чување, измену, брисање и селекцију података о моделу квалитета софтвера и анализираним пројектима.

- Ниво складишта података садржи структуру система. У оквиру складишта података чувају се подаци о анализираним пројектима, као и подаци која се односе на модел квалитета софтвера (тј. на атрибуте квалитета софтвера и софтверске метрике).



Слика 46. Архитектура SilabMetrics алата за статичку анализу квалитета софтвера

5.2.4.1. Интеграција SilabMetrics алата са NetBeans развојним окружењем

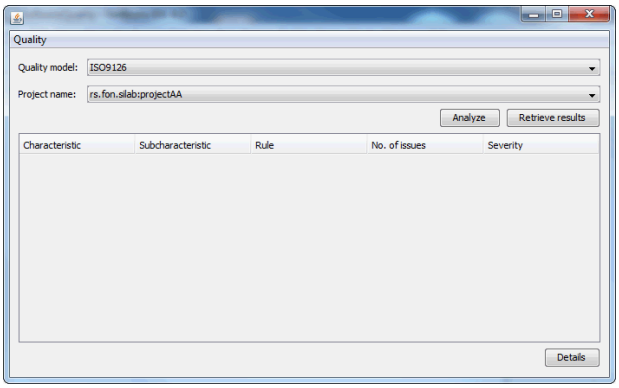
SilabMetrics алат је интегрисан са NetBeans окружењем за развој софтвера. На тај начин је омогућена брза анализа и исправљање уочених недостатака у

процесу развоја софтвера: инструктори и студенти су усмерени да пишу програмски код који је у складу са дефинисаним моделом квалитета. Након извршене анализе SilabMetrics приказује детаљни опис насталог проблема, уз сугестије како посматрани проблем треба исправити. Дуплим кликом на посматрани проблем отвара се специфична линија у SilabMetrics окружењу за развој софтвера, након чега се врши непосредна исправка проблема [Milic17].

У наставку је дат поступак интеграције са NetBeans развојним окружењем. У том смислу се даје редослед извршења корака и графички приказ извршења сваког корака путем слике.

Корак 1. Одабрати модел квалитета и пројекат

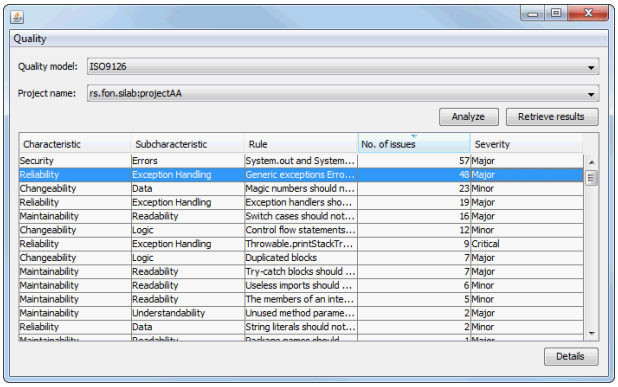
На почетку је потребно изабрати модел квалитета и пројекат чију је статичку анализу потребно извршити, као што је и приказано на наредној слици (Слика 47).



Слика 47. Интеграција SilabMetrics алата са NetBeans развојним окружењем - Избор модела квалитета и пројекта

Корак 2. Извршити статичку анализу квалитета софтвера

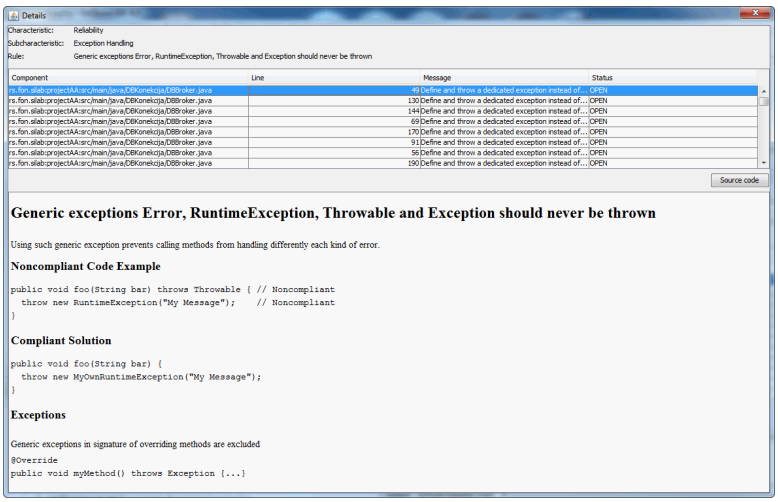
Статичка анализа квалитета софтвера извршава се кликом на дугме **Analyze**. Тада се позива SonaQube алат који анализира изабрани пројекат. Приказ резултата статичке анализе реализује се кликом на дугме **Retrieve Results**, као што је и приказано на наредној слици (Слика 48).



Слика 48. Интеграција SilabMetrics алата са NetBeans развојним окружењем - Извршење статичке анализе квалитета софтвера

Корак 3. Погледати опис насталог проблема

На основу извршене статичке анализе квалитета приказује се опис свих проблема које је потребно исправити. Детаљније информације о проблему добијају се дуплим кликом на проблем у табели или селекцијом реда у табели и кликом на дугме **Details**. Тада се отвара дијалог у оквиру кога се приказује детаљни опис изабраног проблема, уз сугестије како посматрани проблем треба исправити, што је и приказано на наредној слици (Слика 49).

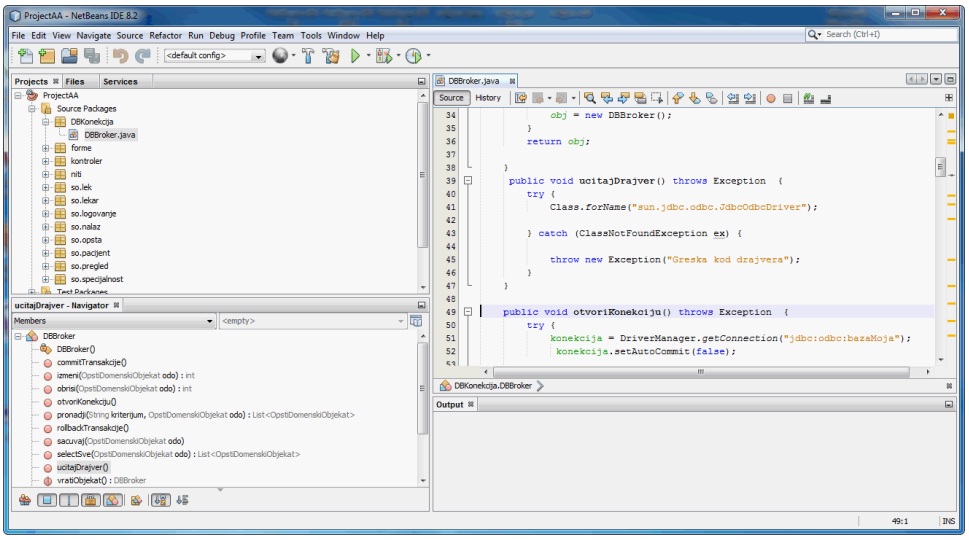


Слика 49. Интеграција SilabMetrics алата са NetBeans развојним окружењем - Преглед описа насталог проблема

Корак 4. Исправити проблем у NetBeans развојном окружењу

Приликом прегледа описа насталог проблема приказује се класа и број реда у којем је посматрани проблем настао. На основу тога се дуплим кликом на

ред у табели у NetBeans развојном окружењу отвара класа и позиционира на изабрану линију кода, што је и приказано на наредној слици (Слика 50). На тај начин је могућа исправка насталог проблема.



Слика 50. Интеграција SilabMetrics алата са NetBeans развојним окружењем - Исправка проблема

5.3. Компаративна анализа алата за статичку анализу квалитета софтвера

У оквиру претходних одељака приказани су алати за статичку анализу програмског кода. У том смислу представљени су Swat4J, SonarQube, FindBugs и SilabMetrics алати и њихове најважније карактеристике. Примећује се да алати имају различит приступ анализи квалитета софтверског система: разликују се подржани модели квалитета и софтверске метрике, као и сам начин рада алата. На пример, алати Swat4J, SonarQube и SilabMetrics врше непосредну анализу програмског кода док је FindBugs алат усмерен на анализу бајткода. У случају да је програмски код доступан могу се користити сви поменути алати. С друге стране, уколико програмски код није доступан, алати Swat4J, SonarQube и SilabMetrics се не могу користити.

У наставку рада даје се упоредни приказ алата за статичку анализу квалитета софтверског система. У том смислу је важно дефинисати

критеријуме поређења посматраних алата. У научној литератури се могу наћи различите компаративне анализе алата за статичку анализу квалитета софтвера [Emanuelsson08][Gomes09][Hofer10]. У сваком раду дефинишу се релеватни критеријуми за анализу:

- У раду [Hofer10] разматра се анализа великог броја алата који су усмерени на различите програмске језике при чему се користе следећи критеријуми поређења: инсталација, конфигурација, подршка, извештаји, грешке и подршка управљању пројектом.
- У раду [Emanuelsson08] разматра се анализа три алата за статичку анализу софтверских система који су написани у језицима C и C++. Као главни критеријум поређења користе се функционалности које су подржане алатима као што су подршка за програмске језике, софтверске метрике, инкрементална анализа, проширивост и др. При томе, ова студија је специфична зато што се односи на компанију Ericsson и њена искуства у примени ових алата.
- У раду [Gomes09] разматра се већи број алата који су усмерени на различите програмске језике, при чему се као критеријум упоредне анализе користи постојећи студијски пример који садржи дефекте. У том контексту се разматра употреба ових алата и резултата које они дају.

Пажљивом анализом доступне научне литературе, узимајући у обзир досадашње резултате истраживања који се односе на стандарде квалитета софтвера и алате за статичку анализу квалитета софтвера, дефинисали смо следеће критеријуме поређења:

- заснованост модела квалитета на стандардима квалитета софтвера,
- могућност конфигурације модела квалитета софтвера,
- подржане софтверске метрике,
- могућност конфигурације софтверских метрика,
- подршка за различите софтверске технологије,

- неопходност постојања изворног програмског кода,
- интеграција са другим алатима за статичку анализу квалитета софтвера,
- интеграција са алатима за управљање софтверским пројектима,
- интеграција са развојним окружењима,
- могућност израде додатака (енг. plugins),
- могућност програмског приступа резултатима анализе.

У наставку рада ће сваки критеријум поређења бити образложен. Након тога ће бити дат приказ наведених алата у контексту посматраног критеријума. На крају ће бити дат сумарни приказ упоредне анализе алата.

5.3.1. Заснованост модела квалитета на стандардима квалитета софтвера

Стандардима квалитета софтвера дефинише се специфично значење квалитета софтвера [Alsultanny09]. У оквиру ранијих поглавља приказани су ISO/IEC 9126, ISO/IEC 14598 и ISO/IEC 25000 стандарди квалитета софтвера. Стандардом ISO/IEC 9126 дефинишу се модели квалитета софтвера и софтверске метрике које одговарају тим моделима док се стандардом ISO/IEC 14598 дефинише поступак евалуације квалитета софтвера. У том смислу се може рећи да су поменути стандарди компатибилни и да се међусобно допуњују. Стандарди серије ISO/IEC 25000 проширују (и замењују) поменуте стандарде: стандарди су организовани у одељке и на свеобухватан начин посматрају квалитет софтвера. Поједини стандарди из серије ISO/IEC 25000 су још увек у фази израде. При томе, у оквиру стандарда је дефинисан посебан одељак који ће садржати техничке извештаје и проширења ISO/IEC 25000 стандарда квалитета софтвера.

С друге стране, стандарди се могу односити и на најбоље праксе у писању програмског кода. На пример, компанија Oracle има своје препоруке како треба писати програмски код у програмском језику Java. Исто то дефинише и компанија Microsoft за своју фамилију .NET програмских језика. Такође и

поједине компаније могу имати своје интерне стандарде које се односе на квалитет софтверског система.

Значај стандарда квалитета софтвера је вишеструк. Најпре, најважнија карактеристика је производња квалитетног софтвера који је у складу са дефинисаним захтевима. У том смислу инжењери за развој софтвера, развојни тим и менаџмент имају добар увид у квалитет софтвера, боље се разумеју и лакше сарађују. Коришћењем алата који су засновани на стандардима квалитета омогућено је брзо увођење нових софтверских инжењера у процес развоја софтвера: софтверски инжењери су усмерени да пишу програмски код који је у складу са дефинисаним моделом квалитета. С друге стране, сви заинтересовани корисници су у могућности да искажу своје потребе на начин који је разумљив свим учесницима у процесу развоја софтвера. Стога се може рећи да стандарди квалитета софтвера представљају медијатор, нешто око чега се слажу сви који су заинтересовани за посматрани софтверски производ.

Софтверски пакет Swat4J је заснован на ISO/IEC 9126 стандарду квалитета софтвера. Његов модел квалитета формиран је на основу модела екстерног и интерног квалитета софтвера који је дефинисан ISO/IEC 9126-1 стандардом квалитета. У том смислу су дефинисани атрибути квалитета софтвера: тестирање, квалитет пројектовања, перформансе, разумљивост, одржавање и поновно коришћење. Алат подржава интерне софтверске метрике које су дефинисане ISO/IEC 9126-3 стандардом квалитета софтвера. Један атрибут квалитета софтвера садржи више софтверских метрика, при чему једна софтверска метрика може бити повезана са више атрибута квалитета софтвера.

Софтверски пакет SonarQube је такође заснован на ISO/IEC 9126 стандарду квалитета софтвера. Као што је раније објашњено он садржи SQALE модел квалитета софтвера који је креиран на основу модела екстерног и интерног квалитета софтвера (дефинисан је у оквиру ISO/IEC

9126-1 стандарда квалитета). У том смислу алат дефинише следеће атрибуте квалитета: тестирање, поузданост, променљивост, ефикасност, сигурност, одржавање, преносивост и поновно коришћење. Ови атрибути су организовани хијерархијски, према нивоу значајности (ово је велика разлика у односу на модел квалитета који је дефинисан ISO/IEC 9126-1 стандардом квалитета код кога се сви атрибути квалитета налазе на истом нивоу). Стога атрибути квалитета на вишем нивоу хијерархије зависе од атрибута квалитета на нижем нивоу хијерархије: уколико је неки атрибут квалитета незадовољен нарушени су сви атрибути квалитета који се налазе на вишим нивоима хијерархије. За сваки атрибут квалитета везано је више подкарактеристика. С друге стране, за једну подкарактеристику везано је више софтверских метрика која се у оквиру овог алата називају правила.

Софтверски пакет FindBugs не садржи модел квалитета који је експлицитно заснован на неком стандарду. Он је, пре свега, усмерен на оперативно откривање грешака у програмском коду па су у том смислу дефинисани обрасци грешака. Сваки образац грешке припада једној категорији, при чему једна категорија садржи више образаца грешака. Софтверски пакет обухвата следеће категорије: лоша пракса, тачност, сумњив код, експерименталност, интернационализација, рањивост при злонамерном коду, вишенитна тачност, перформансе и сигурност.

Софтверски пакет SilabMetrics је такође заснован на ISO/IEC 9126 стандарду квалитета софтвера. Као што је раније објашњено он је заснован на SonarQube алату за статичку анализу квалитета софтвера који садржи SQALE модел квалитета софтвера. SilabMetrics алат садржи следеће атрибуте квалитета софтвера: променљивост, ефикасност, одржавање, преносивост, поузданост, поновно коришћење, сигурност и тестирање.

У наредној табели (Табела 20) дат је компаративни приказ алата по критеријуму *Заснованост модела квалитета на стандардима квалитета софтвера*.

Табела 20. Компаративни приказ алата по критеријуму Заснованост модела квалитета на стандардима квалитета софтвера

Критеријум	**Swat4J**	**SonarQube**	**FindBugs**	**SilabMetrics**
Заснованост модела квалитета на стандардима квалитета софтвера	Да (ISO/IEC 9126-1 модел екстерног и интерног квалитета, ISO/IEC 9126-3 интерне софтверске метрике)	Да (SQALE модел квалитета који је заснован на ISO/IEC 9126-1 моделу екстерног и интерног квалитета)	Не садржи модел квалитета који је експлицитно заснован на неком стандарду	Да (заснован на алату SonarQube који садржи SQALE модел квалитета и подржава ISO/IEC 9126-1 модел екстерног и интерног квалитета)

5.3.2. Могућност конфигурације модела квалитета софтвера

У оквиру претходног одељка приказан је значај стандарда квалитета софтвера. Коришћење стандарда квалитета софтвера у процесу развоја софтвера помаже у стварању софтверског производа који је у складу са дефинисаним моделом квалитета софтвера. Уколико је посматрани софтверски систем развијен у складу са дефинисаним моделом квалитета добија се софтверски систем који је ефикасан, добро пројектован, који се лако одржава и може се поново користити. Ово су неки од атрибута квалитета који могу бити дефинисани моделом квалитета софтвера.

Међутим, нису сви софтверски системи исти: они се разликују у величини, домену проблема који се њиховим коришћењем решава, коришћеним технологијама у процесу развоја софтвера итд. Такође, нису ни све

компаније које се баве развојем софтверских система исте. На крају, разликују се и непосредни корисници посматраног софтверског система.

У том смислу се може рећи да предефинисани модел квалитета не мора бити одговарајући за развој сваког софтверског система. Неки атрибути квалитета могу бити значајнији у односу на друге атрибуте квалитета. С друге стране, неки атрибути квалитета не морају уопште бити од значаја за посматрани софтверски систем. Стога је потребно дефинисати специфично значење квалитета за софтверски производ који се развија. На тај начин се квалитет софтвера може прилагодити специфичним захтевима који се односе на софтверски производ, компанијама које развијају софтвер или корисницима софтвера. С друге стране, променом параметара модела квалитета могуће је вршити постоптималну анализу квалитета софтверског система. На тај начин је могуће утврдити како промена неког параметра модела квалитета утиче на квалитет софтверског система. Другим речима, могуће је правити баланс у смислу задовољења атрибута квалитета софтвера узимајући у обзир специфично знање квалитета.

Софтверски пакет Swat4J омогућава подешавање модела квалитета софтвера. У оквиру алата је могуће дефинисати везу софтверских метрика са атрибутима квалитета софтвера. На тај начин се може дефинисати специфично значење квалитета софтвера. Променом параметара модела квалитета добијају се другачије оцене атрибута квалитета посматраног софтверског система. С друге стране, алат не дозвољава изостављање неких атрибута квалитета софтвера.

Софтверски пакет SonarQube такође дозвољава подешавање модела квалитета софтвера. Према предефинисаном моделу квалитета на првом нивоу хијерархије налази се атрибут квалитета *Тестирање*. Сви остали атрибути зависе од овог атрибута квалитета. За сваки атрибут квалитета везано је више подкарактеристика, док је за сваку подкарактеристику везано више правила. У том смислу је могуће извршити измену предефинисаног

модела квалитета и на првом (или неком другом) нивоу хијерархије поставити неки други атрибут квалитета. Међутим, и даље се поштује правило да атрибут квалитета на нижем нивоу хијерархије утиче на све атрибуте квалитета који се налазе на вишим нивоима хијерархије. Ово је, заправо, измена SQALE модела квалитета који се користи у оквиру SonarQube софтверског пакета. Штавише, SQALE модел квалитета је јавно доступан, као отворени пројекат: може се преузети комплетна документација и дати другачија организација хијерархије атрибута квалитета, дефинисати другачије карактеристике и другачија правила [Letouzey12].

Софтверски пакет FindBugs није експлицитно заснован на неком стандарду али такође садржи сопствени модел квалитета. У том смислу су дефинисане различите категорије, при чему свака категорија садржи више образаца грешака. Овај софтверски пакет такође подржава измену предефинисаног модела квалитета. Аутори су спровели истраживање међу корисницима алата и дошли су до занимљивих запажања. Наиме, 55% испитаника је рекло да не врши никаква подешавања овог алата, тј. да користи предефинисана подешавања, без додатног филтрирања образаца грешака [Ayewah08].

Софтверски пакет SilabMetrics је заснован на SonarQube алату и садржи предефинисани SQALE модел квалитета софтвера. С обзиром да су првенствени корисници SilabMetrics алата инструктори и студенти, алат дозвољава подешавање модела квалитета софтвера како би се на бољи начин стекла знања и вештине које се односе на стандарде квалитета софтвера, моделе квалитета софтвера и софтверске метрике.

У наредној табели (Табела 21) дат је компаративни приказ алата по критеријуму *Могућност конфигурације модела квалитета*.

Табела 21. Компаративни приказ алата по критеријуму Могућност конфигурације модела квалитета софтвера

Критеријум	Swat4J	SonarQube	FindBugs	SilabMetrics
Могућност конфигурације модела квалитета	Да (у ограниченој мери)	Да	Да	Да

5.3.3. Подржане софтверске метрике

Софтверске метрике имају веома значајну улогу у оквиру стандарда квалитета софтвера, односно модела квалитета софтвера. Наиме, стандардима квалитета софтвера дефинишу се модели квалитета софтвера (атрибути квалитета и њихове подкарактеристике). Задовољење атрибута квалитета мери се софтверским метрикама.

Софтверска метрика треба да задовољи следећа својства: математички је формализована, емпиријски потврђена, објективна је, проверљива је, предвидљива је, независна је од програмског језика, величине софтверског система и домена проблема [Pressman10]. Софтверским метрикама се оперативно мере атрибути квалитета посматраног софтверског система: софтверска метрика доводи до промене атрибута квалитета софтверског система са којим је повезана (у складу са повећавањем вредности софтверске метрике посматрани атрибут квалитета софтверског система се повећава или смањује). На тај начин се могу рационално донети одлуке о квалитету софтверског система.

У оквиру Поглавља 4 дате су дефиниције и објашњења често коришћених софтверских метрика:

- Сложеност пондерисаних метода,
- Број пондерисаних метода класе,

- Циклична сложеност,
- Број одговора класе,
- Недостатак кохезивности метода у класи,
- Повезаност објеката,
- Дубина стабла наслеђивања,
- Број подкласа,
- Број наредби у методи,
- Метрика стабилности софтвера.

Приказане софтверске метрике су усмерене на објектно-оријентисане софтверске системе и заједно се користе у процесу евалуације квалитета софтвера. Оне су комплементарне (међусобно се допуњују). У наредној табели (Табела 22) дат је компаративни приказ алата по критеријуму *Подржане софтверске метрике*.

Табела 22. Компаративни приказ алата по критеријуму Подржане софтверске метрике

Критеријум	**Swat4J**	**SonarQube**	**FindBugs**	**SilabMetrics**
Подржане софтверске метрике	Сложеност пондерисаних метода, Број пондерисаних метода класе, Циклична сложеност, Број одговора класе, Недостатак кохезивности метода у класи, Повезаност објеката, Дубина стабла наслеђивања, Број подкласа,	Сложеност пондерисаних метода, Циклична сложеност, Повезаност објеката, Дубина стабла наслеђивања, Број подкласа	-	Сложеност пондерисаних метода, Циклична сложеност, Повезаност објеката, Дубина стабла наслеђивања, Број подкласа, Метрика стабилности софтвера

Критеријум	Swat4J	SonarQube	FindBugs	SilabMetrics
	Број наредби у методи			

5.3.4. Могућност конфигурације софтверских метрика

Раније је напоменуто да се софтверским метрикама оперативно мере атрибути квалитета посматраног софтверског система. Поред тога, софтверска метрика може имати и додатна својства (нпр. ниво значајности софтверске метрике, граничне вредности софтверске метрике итд.). У том смислу је могућност конфигурације појединачних софтверских метрика веома значајно. Променом параметара софтверске метрике могуће је извршити постоптималну анализу (утврдити како промена неке софтверске метрике утиче на појединачни атрибут квалитета, односно на квалитет посматраног софтверског система). На овај начин се може дефинисати специфично значење квалитета софтвера.

Софтверски пакет Swat4J омогућава конфигурацију софтверских метрика. У том смислу могуће је дефинисати везу софтверских метрика са атрибутима квалитета софтвера (један атрибут квалитета софтвера повезан је са више софтверских метрика, док једна софтверска метрика може бити повезана са више атрибута квалитета софтвера). Могуће је извршити конфигурацију граничних вредности софтверске метрике за сваки атрибут квалитета софтвера. У складу са извршеним изменама могуће је добити другачије вредности за атрибуте квалитета посматраног софтверског система.

Софтверски пакет SonarQube омогућава конфигурацију софтверских метрика (у оквиру алата користи се термин *правила*). У том смислу је могуће дефинисати везу правила са карактеристикама и подкарактеристикама квалитета софтвера, као и појединачно подешавање правила. Може се

променити подразумевана вредност посматраног правила као и ниво значајности посматраног правила. На тај начин се може дефинисати специфично значење квалитета софтвера.

Софтверски пакет FindBugs омогућава конфигурацију софтверских метрика (у оквиру пакета се користи термин *обрасци грешке*) додавањем филтера. Једном дефинисани филтери се могу сачувати и користити у оквиру других пројеката. Филтерима се може утицати на предефинисани модел квалитета софтвера.

Софтверски пакет SilabMetrics омогућава конфигурацију софтверских метрика. У том смислу могуће је дефинисати везу између софтверских метрика и различитих атрибута квалитета софтвера. На тај начин инструктори и студенти могу да стекну знања и вештине које се односе на стандарде квалитета софтвера, моделе квалитета софтвера и софтверске метрике.

У наредној табели (Табела 23) дат је компаративни приказ алата по критеријуму *Могућност конфигурације софтверских метрика*.

Табела 23. Компаративни приказ алата по критеријуму Могућност конфигурације софтверских метрика

Критеријум	**Swat4J**	**SonarQube**	**FindBugs**	**SilabMetrics**
Могућност конфигурације софтверских метрика	Да	Да	Да	Да

5.3.5. Подршка за различите софтверске технологије

Процес развоја софтвера је сложен: он обухвата различите моделе, методе, стратегије и активности које се непосредно користе у процесу развоју софтвера. Као резултат развоја добија се производ, софтверски систем који

је реализован у некој софтверској технологији (или у више софтверских технологија). Свака софтверска технологија има своје предности и недостатке: на пример, велика предност Java технологије је њена платформска независност па се у том смислу једном написан програм може извршавати на различитим оперативним системима под условом да за посматрани оперативни систем постоји одговарајућа Java виртуелна машина. На тај начин се обезбеђује преносивост посматраног софтверског система из једног оперативног окружења у друго оперативно окружење. Због тога је веома важан избор софтверске технологије за имплементацију софтверског система.

У том контексту се могу посматрати и алати за статичку анализу програмског кода. Уколико се компанија бави развојем софтверских система коришћењем једне софтверске технологије, у процесу развоја се може користити алат за статичку анализу квалитета који подржава ту софтверску технологију. Међутим, уколико се у процесу развоја користи више софтверских технологија алат за статичку анализу квалитета би требао да подржава више софтверских технологија. Овај приступ има вишеструке предности. Најпре, на тај начин се рационалније користе ресурси компаније (време, новац и други значајни ресурси компаније). Другим речима, није потребно ново учење и прилагођавање алату за статичку анализу квалитета софтвера: исти софтверски пакет се користи за анализу квалитета различитих софтверских пројеката који могу бити имплементирани у различитим софтверским технологијама.

Софтверски пакет Swat4J је усмерен на анализу квалитета софтверских система који су имплементирани у програмском језику Java. Он ни на који начин не подржава друге софтверске технологије.

Софтверски пакет SonarQube подржава велики број софтверских технологија. С обзиром на архитектуру која у основи омогућава израду проширења (енг. plugin) алат има могућност да подржи различите

програмске језике. У том смислу је кроз софтверски пакет могуће извршити статичку анализу квалитета софтверских система који су реализовани различитим софтверским технологијама: Java, C#, C/C++, PHP, Objective-C, Visual Basic 6, VB.NET и Python. Поред тога алат има подршку и за PL/SQL језик компаније Oracle. Важно је напоменути да се подршка за софтверске технологије остварује кроз проширења од којих су нека комерцијална. Међутим, софтверски пакет има добро дефинисану архитектуру и пружа подршку за преко 20 софтверских технологија [SonarQube].

Софтверски пакет FindBugs је усмерен на анализу квалитета софтверских система који су реализовани коришћењем програмског језика Java. Кроз софтверски пакет нису подржане друге софтверске технологије.

Софтверски пакет SilabMetrics је усмерен на анализу квалитета софтверских система који су имплементирани у програмском језику Java [Milic17]. Он не подржава друге софтверске технологије. С друге стране, он је интегрисан са SonarQube алатом за статичку анализу квалитета софтвера, тако да је могуће путем веб сервиса омогућити подршку за друге софтверске технологије.

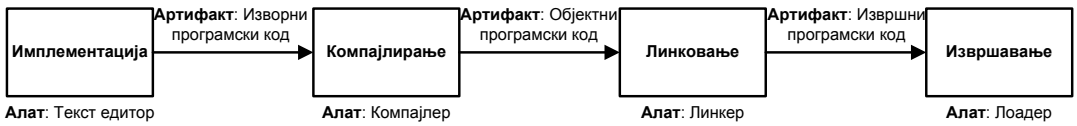
У наредној табели (Табела 24) дат је компаративни приказ алата по критеријуму *Подршка за различите софтверске технологије*.

Табела 24. Компаративни приказ алата по критеријуму Подршка за различите софтверске технологије

Критеријум	**Swat4J**	**SonarQube**	**FindBugs**	**SilabMetrics**
Подршка за различите софтверске технологије	Не (подршка само за Java софтверску технологију)	Да (подршка за преко 20 софтверских технологија)	Не (подршка само за Java софтверску технологију)	Не (подршка само за Java софтверску технологију)

5.3.6. Неопходност постојања изворног програмског кода

Изворни програмски код (енг. Source Code) представља артифакт који настаје у фази имплементације софтверског система у конкретној софтверској технологији. Изворни програмски код се уређује у неком текст едитору. Међутим, за извршавање програма није неопходан изворни програмски код. У општем случају се изворни програмски код поступком који се назива компајлирање преводи у објектни програмски код (енг. Object Code) који, у ствари, представља машински програмски код. Након тога се поступком који се назива линковање врши повезивање израђеног објектног кода са другим објектним кодовима (нпр. библиотеком коришћених класа које су уграђене у програмски језик) у једну целину. На тај начин се добија извршни код програма који је могуће покренути на конкретном оперативном систему. Уопштени поступак израде софтверског система приказан је на наредној слици: приказани су алати и артифакти који се користе у процесу развоја софтвера.



Слика 51. Уопштени поступак израде софтверског система

На слици (Слика 51) је приказан уопштени поступак израде: различити програмски језици могу то урадити на другачије начине. На пример, као резултат изградње Java програма не добија се извршни програмски код већ бајткод који је платформски независан. Он се може интерпретирати на конкретном оперативном систему под условом да за конкретан оперативни систем постоји Java виртуелна машина која наредбе из бајткода преводи у наредбе конкретног оперативног система [Vlajico8].

Из изложеног се може закључити да се програмски код посматраног софтверског система може наћи у више појавних облика тако да се у оквиру статичке анализе квалитета не мора користити изворни програмски код.

Некада изворни програмски код није доступан: нпр. уколико је софтверски систем купљен, уколико је набављена библиотека класа, уколико су набављене софтверске компоненте. У том случају се поставља питање извршења анализе квалитета софтвера. Неки алати омогућавају анализу квалитета софтверских система и уколико изворни програмски код није доступан.

С друге стране, боље је уколико је изворни програмски код доступан. Он се свакако може превести у други појавни облик. Такође, доступност изворног програмског кода у значајној мери олакшава интеграцију са алатима за статичку анализу квалитета софтвера: могућ је прецизан приказ линија програмског кода у којима постоје проблеми и може се омогућити ефикаснија исправка уочених грешака.

Софтверски пакет Swat4J усмерен је анализу изворног програмског кода. Постојање изворног програмског кода представља неопходан услов за статичку анализу квалитета посматраног софтверског система.

Софтверски пакет SonarQube је такође усмерен на анализу изворног програмског кода. Он захтева постојање изворног програмског кода како би се могла извршити статичка анализа квалитета посматраног софтверског система.

Софтверски пакет FindBugs омогућава статичку анализу квалитета посматраног софтверског система при чему се не захтева постојање изворног програмског кода. Као што је раније напоменуто овај алат се користи за анализу софтверских система који су имплементирани у програмском језику Java. Другим речима, софтверски пакет у поступку анализе користи бајткод. Он се може ефикасно користити уколико изворни програмски код посматраног софтверског система није доступан.

Софтверски пакет SilabMetrics је усмерен на анализу изворног програмског кода. У том смислу је неопходно постојање изворног

програмског кода како би се могла извршити статичка анализа квалитета посматраног софтверског система.

У наредној табели (Табела 25) дат је компаративни приказ алата по критеријуму *Неопходност постојања изворног програмског кода*.

Табела 25. Компаративни приказ алата по критеријуму Неопходност постојања изворног програмског кода

Критеријум	**Swat4J**	**SonarQube**	**FindBugs**	**SilabMetrics**
Неопходност постојања изворног програмског кода	Да	Да	Не	Да

5.3.7. Интеграција са другим алатима за статичку анализу квалитета софтвера

На тржишту постоји велики број алата за статичку анализу квалитета софтвера као што су: Coverity, PMD, CheckStyle, JLint и др. [Emanuelsson08][Gomes09][Hofer10]. Сваки алат има има неке карактеристике које га издвајају па може бити погодан за статичку анализу квалитета посматраног софтверског система.

С друге стране, неки софтверски пакети омогућавају интеграцију са другим софтверским пакетима за статичку анализу квалитета. Под интеграцијом се подразумева једноставно укључивање другог алата у оквиру посматраног алата. Предности овог приступа су свакако вишеструке: софтверски пакет постаје комплетнији, омогућава већи број софтверских метрика, другачије моделе квалитета софтвера и сл. Другим речима, интеграцијом са другим алатима за статичку анализу квалитета софтвера се могу проширити

функционалности и надокнадити ограничења које поседује посматрани алат.

На тај начин се, такође, могу ефикасније користити расположиви ресурси (време, новац и други значајни ресурси компаније). Другим речима, није потребно ново учење и прилагођавање алату за статичку анализу квалитета софтвера као ни куповина нових софтверских пакета: исти алат се може користити за анализу квалитета посматраног софтверског система при чему је могуће користити већи скуп софтверских метрика, правила, модела квалитета и сл.

Софтверски пакет Swat4J не подржава интеграцију са другим алатима за статичку анализу квалитета софтвера. Штавише, он је реализован као проширење које користи Eclipse развојну платформу па се мора користити у оквиру Eclipse интегрисаног развојног окружења.

Софтверски пакет SonarQube подржава интеграцију са другим алатима за статичку анализу квалитета софтвера. С обзиром на архитектуру која у основи омогућава израду проширења софтверски пакет има могућност да подржи различите алате за статичку анализу квалитета софтвера. У том смислу је извршена интеграција са алатима FindBugs, PMD и CheckStyle [SonarQube]. На тај начин је значајно проширен број правила која су обухваћена овим софтверским пакетом.

Софтверски пакет FindBugs не подржава интеграцију са другим алатима за статичку анализу квалитета софтвера. Међутим, он се успешно може укључити у оквиру других алата за статичку анализу квалитета софтвера (у том смислу је извршена интеграција са SonarQube софтверским пакетом).

Софтверски пакет SilabMetrics подржава интеграцију са SonarQube алатом за статичку анализу квалитета софтвера. На тај начин је у оквиру SilabMetrics алата омогућено коришћење функција SonarQube алата које су изложене путем веб сервиса.

У наредној табели (Табела 26) дат је компаративни приказ алата по критеријуму *Интеграција са другим алатима за статичку анализу квалитета софтвера*.

Табела 26. Компаративни приказ алата по критеријуму Интеграција са другим алатима за статичку анализу квалитета софтвера

Критеријум	**Swat4J**	**SonarQube**	**FindBugs**	**SilabMetrics**
Интеграција са другим алатима за статичку анализу квалитета софтвера	Не	Да	Не (може се укључити у оквиру других алата)	Да (интеграција са SonarQube алатом)

5.3.8. Интеграција са алатима за управљање софтверским пројектима

Софтверски системи могу бити веома сложени: састоје се од великог броја модула, реализовани су у различитим софтверским технологијама, користе велики број готових библиотека, имају велики број линија програмског кода и сл. Ово су само неке могуће карактеристике сложених софтверских система. У том смислу је неопходно ефикасно управљање процесом развоја софтверског пројекта па су развијени посебни алати који се могу користити у ову сврху.

Алати за управљање софтверским пројектом обезбеђују више функционалности. Најважнија функционалност је свакако изградња верзије (енг. build) софтверског система. У том процесу је могуће аутоматизовати извршење свих тестова: на тај начин се приликом додавања нове или измене постојеће функционалности могу покренути сви тестови како би се проверило да ли верзија софтвера задовољава све претходно написане

тестове. Овај поступак се зове регресионо тестирање. Алати такође омогућавају и генерисање документације као и детаљно извештавање о процесу изградње верзије система.

Аналогно се може посматрати и интеграција алата за статичку анализу квалитета софтвера са алатима за управљање софтверским пројектима. Уколико је интеграција могућа, у процесу изградње софтвера је могуће извршити статичку анализу квалитета посматраног софтверског система и на основу тога добити извештаје о нарушавању метрика и атрибута квалитета софтверског система. Овај поступак се може извршити приликом додавања нове или измене постојеће функционалности па је, у складу са извештајем о анализи квалитета посматраног софтверског система, могуће предузети акције како би се неусаглашености елиминисале.

Софтверски пакет Swat4J не подржава интеграцију са алатима за управљање софтверским пројектима. Он се искључиво користи у оквиру Eclipse развојног окружења.

Софтверски пакет SonarQube подржава интеграцију са алатима за управљање софтверским пројектима. Он се може интегрисати са алатима за управљање софтверским пројектима (нпр. Apache Maven, Apache Ant и JIRA), као и алатима за управљање верзијама програмског кода (нпр. SVN, Git и Mercurial) [SonarQube]. С обзиром на свеобухватну подршку може се рећи да се алат може успешно интегрисати у процес развоја софтвера.

Софтверски пакет FindBugs подржава интеграцију са алатима за управљање софтверским пројектима. Он се може успешно интегрисати са Apache Maven и Apache Ant алатима за управљање софтверским пројектима [FindBugs]. Он се, такође, може успешно интегрисати у процес развоја софтвера.

Софтверски пакет SilabMetrics не подржава интеграцију са алатима за управљање софтверским пројектима. Он се користи као самостални алат за

статичку анализу квалитета софтвера или као алат за статичку анализу који се интегрише са NetBeans развојним окружењем.

У наредној табели (Табела 27) дат је компаративни приказ алата по критеријуму *Интеграција са алатима за управљање софтверским пројектима*.

Табела 27. Компаративни приказ алата по критеријуму Интеграција са алатима за управљање софтверским пројектима

Критеријум	**Swat4J**	**SonarQube**	**FindBugs**	**SilabMetrics**
Интеграција са алатима за управљање софтверским пројектима	Не	Да (Алати за управљање софтверским пројектима - Apache Maven, Apache Ant и JIRA, Алати за управљање верзијама - SVN, Git и Mercurial)	Да (Алати за управљање софтверским пројектима - Apache Maven и Apache Ant)	Не

5.3.9. Интеграција са развојним окружењима

Сваки софтверски систем се, у општем случају, може користити самостално, као независан алат. С друге стране, могућност интеграције са другим алатима је веома значајна карактеристика посматраног алата. На тај начин се пружају различите могућности коришћења посматраног алата.

У оквиру претходног одељка објашњен је критеријум *Интеграција са алатима за управљање софтверским пројектима* и предности које та врста

интеграције доноси. С друге стране, може се посматрати и интеграција алата за статичку анализу квалитета софтвера са развојним окружењима. Раније је напоменуто да се у процесу развоја софтвера користе едитори текста у оквиру којих се непосредно пише програмски код. Уместо текст едитора могу се користити интегрисана развојна окружења која пружају значајно шири скуп функционалности.

Интеграција алата за статичку анализу квалитета софтвера са развојним окружењима доноси вишеструке предности. Најпре, инжењер који се бави развојем софтвера не мора да учи да користи нови алат: он све послове обавља из истог развојног окружења које већ користи и на које је навикао. У том смислу се елиминише време које је потребно за пребацивање из развојног окружења у алат за статичку анализу квалитета софтвера и обрнуто. С друге стране, на овај начин се повећава продуктивност инжењера за развој софтвера: приликом додавања нове или измене постојеће функционалности инжењер је у могућности да из развојног окружења изврши статичку анализу квалитета софтверског система и провери да ли постоје неке неусаглашености (у смислу нарушавања софтверских метрика или атрибута квалитета посматраног софтверског система). С обзиром да инжењер већ користи развојно окружење, он може на једноставан начин да изврши исправку уочених неусаглашености.

Софтверски пакет Swat4J подржава интеграцију са окружењима за развој софтвера. Он се може користити у оквиру Eclipse развојног окружења. Треба нагласити да је ово једини начин за коришћење Swat4J пакета - он се не може користити као самостална апликација нити се може користити из других развојних окружења.

Софтверски пакет SonarQube подржава интеграцију са развојним окружењима. Он се може користити у оквиру Eclipse и IntelliJ IDEA развојних окружења. Треба напоменути да се овај софтверски пакет може користити и као самостални алат, односно може се интегрисати са алатима

за управљање софтверским пројектима. У том смислу се може рећи да постоји више сценарија коришћења посматраног софтверског пакета која задовољавају различите потребе.

Софтверски пакет FindBugs подржава интеграцију са развојним окружењима. У том смислу алат подржава интеграцију са Eclipse развојним окружењем. Путем независних додатака омогућена је и интеграција са NetBeans и IntelliJ IDEA развојним окружењима. Овај софтверски пакет се такође може користити и као самостални алат, односно може се интегрисати са алатима за управљање софтверским пројектима. Другим речима, алат пружа различита сценарија коришћења.

Софтверски пакет SilabMetrics подржава интеграцију са окружењима за развој софтвера. У том смислу омогућена је интеграција алата са NetBeans окружењем за развој софтвера. На тај начин се, након извршене анализе посматраног софтверског система, може извршити брза исправка уочених неусаглашености са моделом квалитета софтвера.

У наредној табели (Табела 28) дат је компаративни приказ алата по критеријуму *Интеграција са развојним окружењима*.

Табела 28. Компаративни приказ алата по критеријуму Интеграција са развојним окружењима

Критеријум	**Swat4J**	**SonarQube**	**FindBugs**	**SilabMetrics**
Интеграција са развојним окружењима	Да (Eclipse развојно окружење)	Да (Eclipse и IntelliJ IDEA развојна окружења)	Да (Eclipse, IntelliJ IDEA и NetBeans развојна окружења)	Да (NetBeans развојно окружење)

5.3.10. Могућност израде додатака

У оквиру претходних одељака дискутоване су различите могућности интеграције алата:

- Интеграција алата за статичку анализу квалитета софтвера **са другим алатима за статичку анализу квалитета софтвера**
- Интеграција алата за статичку анализу квалитета софтвера **са алатима за управљање софтверским пројектима**
- Интеграција алата за статичку анализу квалитета софтвера **са развојним окружењима**.

У том смислу разматране су предности које доносе интеграције: софтверски пакет постаје комплетнији, могу се надокнадити ограничења која постоје у посматраном софтверском алату, ефикасније се могу користити расположиви ресурси (време, новац и други значајни ресурси којима компанија располаже), инжењери који се баве развојем софтвера могу бити продуктивнији, процес изградње софтвера се може аутоматизовати и др.

Међутим, понекад поменуте врсте интеграције нису довољне. Наиме, понекад је потребно проширити посматрани софтверски пакет са новим функционалностима које је потребно имплементирати. У том смислу је потребно да алат омогући израду додатака. Израдом додатака додају се нове функционалности у алату за статичку анализу квалитета софтвера: нове метрике, нова правила, нови модел квалитета и слично.

Да би израда додатака била могућа алат мора да подржава архитектуру која је заснована на софтверским компонентама, тј. мора да обезбеди механизме за проширење алата, за израду нових компоненти, при чему архитектура софтверског система треба остати иста. У објектно-оријентисаним програмским језицима овај механизам се остварује дефинисањем интерфејса преко којих се могу додавати нове функционалности при чему се архитектура софтверског система не мења. Једном написано проширење је

могуће поново користити: може се надограђивати, дати на коришћење другим организацијама и слично.

Могућност израде додатака је пожељна карактеристика софтверских пакета за статичку анализу квалитета софтвера. На тај начин се остварује висок ниво поновног коришћења посматраног софтверског пакета.

Софтверски пакет Swat4J не обезбеђује могућност израде додатака. Штавише, он је написан као додатак за Eclipse развојно окружење. У том смислу он може да користи функционалности које су обезбеђене од стране Eclipse развојне платформе али не постоји експлицитна подршка за израду додатака.

Софтверски пакет SonarQube обезбеђује могућност израде додатака. Његова архитектура је добро осмишљена и постоји имплементиран велики број додатака: за интеграцију са различитим софтверским технологијама (подржано је преко 20 софтверских технологија), различитим алатима за управљање софтверским пројектима (нпр. Apache Maven, Apache Ant, JIRA), различитим алатима за управљање верзијама (нпр. SVN, Git, Mercurial). Поред тога, омогућене су додатне софтверске метрике, као и различити алати за визуализацију и извештавање. Детаљан списак доступних проширења може се пронаћи у [SonarQube].

Софтверски пакет FindBugs обезбеђује могућност израде додатака. Алат се може проширити укључивањем или израдом додатака (енг. plugins) чиме се могу добити додатне функционалности. Његова архитектура дозвољава израду проширења кроз дефинисање нових образаца грешака што доводи до проширења предефинисаног модела квалитета. С друге стране, у оквиру алата су интегрисана предефинисана проширења (нпр. за чување резултата у оквиру cloud-а). Детаљан списак доступних проширења се може пронаћи у [FindBugs].

Софтверски пакет SilabMetrics не обезбеђује могућност израде додатака. Штавише, он у позадини користи SonarQube алат за статичку анализу квалитета софтвера. У том смислу он може да користи функционалности које су обезбеђене од стране SonarQube алата али не постоји експлицитна подршка за израду додатака.

У наредној табели (Табела 29) дат је компаративни приказ алата по критеријуму *Могућност израде додатака*.

Табела 29. Компаративни приказ алата по критеријуму Могућност израде додатка

Критеријум	**Swat4J**	**SonarQube**	**FindBugs**	**SilabMetrics**
Могућност израде додатака	Не	Да	Да	Не

5.3.11. Могућност програмског приступа резултатима анализе

Алатима за статичку анализу квалитета омогућава се анализа посматраног софтверског система без непосредног покретања програма. Након извршене анализе квалитета посматраног софтверског система потребно је креирати извештаје који дају синтетички и аналитички преглед резултата анализе. На основу резултата анализе могу се предузети акције како би се нарушене вредности софтверских метрика или нарушене вредности атрибута квалитета софтвера исправиле.

С друге стране, понекад је потребна даља анализа која се може заснивати на резултатима анализе посматраног алата. Један од примера је учитавање резултата анализе у складишта података и њихова даља анализа за потребе пословног извештавања (енг. Business intelligence - BI), у циљу ефикасног управљања пројектима компаније [Letouzey12b]. Такође, компаније могу користити резултате анализе у свом процесу развоја софтвера [Ayewah08].

На поменуте начине се стварају предуслови за унапређење квалитета софтвера и процеса развоја софтвера.

Постоје различити механизми како се ова подршка може реализовати:

- Алати за статичку анализу квалитета софтвера могу да генеришу извештаје у неком формату (нпр. у текстуалном формату, XML формату, CSV формату и сл.). Након тога је могуће извршити програмску обраду (парсирање) генерисаних извештаја.
- Алати за статичку анализу квалитета софтвера могу да омогуће директан приступ бази података у оквиру које су складиштени резултати анализе квалитета посматраног софтверског система.
- Алати за статичку анализу квалитета софтвера могу да омогуће удаљени позив метода које враћају резултате анализе у неком облику.
- Алати за статичку анализу квалитета софтвера могу да омогуће приступ резултатима анализе коришћењем веб сервиса који може да трансформише резултате анализе у различите формате (нпр. у текстуалном формату, XML формату, CSV формату, JSON формату и сл.).

Могућност програмског приступа резултатима анализе је пожељна особина алата за статичку анализу квалитета софтвера.

Софтверски пакет Swat4J обезбеђује могућност програмског приступа резултатима статичке анализе квалитета софтвера. У том смислу могуће је генерисати различите аналитичке извештаје који се могу даље обрађивати.

Софтверски пакет SonarQube обезбеђује могућност програмског приступа резултатима статичке анализе квалитета софтвера. У том смислу могућ је директан приступ бази података у оквиру које су складиштени резултата анализе. Међутим, овај приступ није пожељан зато што може довести до нарушавања интегритета података. С друге стране, алат обезбеђује приступ резултатима анализе путем веб сервиса што представља препоручени

приступ који треба користити. Веб сервис обезбеђује различите извештаје који могу бити представљени у различитим форматима.

Софтверски пакет FindBugs обезбеђује могућност програмског приступа резултатима статичке анализе квалитета софтвера. У том смислу доступни су различити аналитички извештаји који се могу даље обрађивати.

Софтверски пакет SilabMetrics не обезбеђује могућност програмског приступа резултатима статичке анализе квалитета софтвера.

У наредној табели (Табела 30) дат је компаративни приказ алата по критеријуму *Могућност програмског приступа резултатима анализе*.

Табела 30. Компаративни приказ алата по критеријуму Могућност програмског приступа резултатима анализе

Критеријум	**Swat4J**	**SonarQube**	**FindBugs**	**SilabMetrics**
Могућност програмског приступа резултатима анализе	Да	Да	Да	Не

5.3.12. Сажетак компаративне анализе алата за статичку анализу квалитета софтвера

У оквиру претходних одељака приказана је компаративна анализа Swat4J, SonarQube, FindBugs и SilabMetrics алата за статичку анализу квалитета софтвера. Сви алати усмерени су на анализу квалитета софтверског система при чему се приступи алата анализи разликују. У раду су дефинисани и образложени различити критеријуми за компаративну анализу алата:

- Заснованост модела квалитета на стандардима квалитета софтвера,
- Могућност конфигурације модела квалитета софтвера,

- Подржане софтверске метрике,
- Могућност конфигурације софтверских метрика,
- Подршка за различите софтверске технологије,
- Неопходност постојања изворног програмског кода,
- Интеграција са другим алатима за статичку анализу квалитета софтвера,
- Интеграција са алатима за управљање софтверским пројектима,
- Интеграција са развојним окружењима,
- Могућност израде додатака (енг. plugins),
- Могућност програмског приступа резултатима анализе.

На основу извршене компаративне анализе може се рећи да сваки алат има своје предности и недостатке и погодан је за примену у одређеним ситуацијама (уколико је, на пример, неопходно да алат садржи модел квалитета који је заснован на неком стандарду квалитета софтвера могу се користити Swat4J, SonarQube и SilabMetrics алати; с друге стране, уколико је потребно вршити анализу квалитета софтверског система чији програмски код није доступан може се користити FindBugs алат итд.). С друге стране, примена алата за статичку анализу квалитета софтвера у процесу развоја софтвера директно доприноси повећању нивоа квалитета посматраног софтверског система.

У наредној табели (Табела 31) дат је интегрални приказ поређења алата по свим критеријумима.

Табела 31. Интегрални приказ поређења алата по свим критеријумима

Критеријум	**Swat4J**	**SonarQube**	**FindBugs**	**SilabMetrics**
Заснованост модела квалитета на стандардима	Да (ISO/IEC 9126-1 модел екстерног и интерног	Да (SQALE модел квалитета који је	Не садржи модел квалитета који је	Да (заснован на алату SonarQube који садржи

Критеријум	**Swat4J**	**SonarQube**	**FindBugs**	**SilabMetrics**
квалитета софтвера	квалитета, ISO/IEC 9126-3 интерне софтверске метрике)	заснован на ISO/IEC 9126-1 моделу екстерног и интерног квалитета)	експлицитно заснован на неком стандарду	SQALE модел квалитета и подржава ISO/IEC 9126-1 модел екстерног и интерног квалитета)
Могућност конфигурације модела квалитета	Да (у ограниченој мери)	Да	Да	Да
Подржане софтверске метрике	Сложеност пондерисаних метода, Број пондерисаних метода класе, Циклична сложеност, Број одговора класе, Недостатак кохезивности метода у класи, Повезаност објеката, Дубина стабла наслеђивања, Број подкласа, Број наредби у	Сложеност пондерисаних метода, Циклична сложеност, Повезаност објеката, Дубина стабла наслеђивања, Број подкласа	-	Сложеност пондерисаних метода, Циклична сложеност, Повезаност објеката, Дубина стабла наслеђивања, Број подкласа, Метрика стабилности софтвера

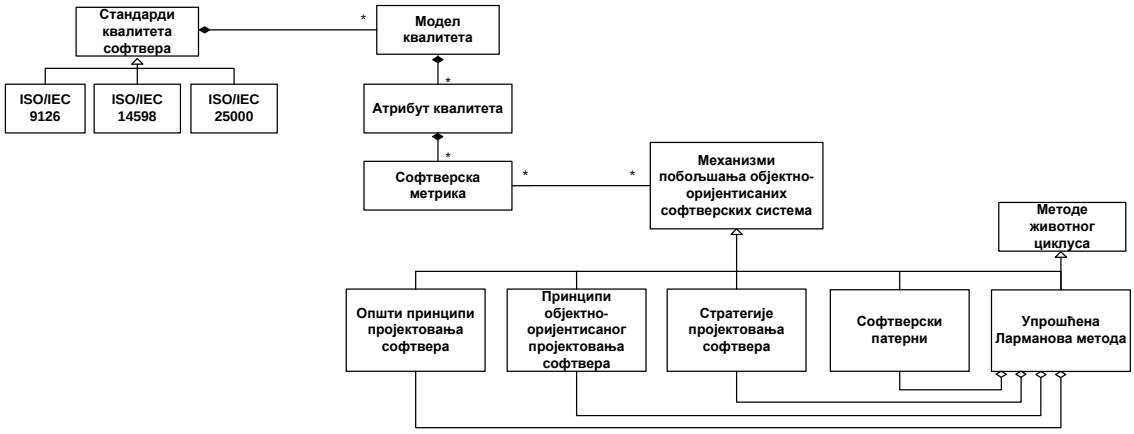
Критеријум	**Swat4J**	**SonarQube**	**FindBugs**	**SilabMetrics**
	методи			
Могућност конфигурације софтверских метрика	Да	Да	Да	Да
Подршка за различите софтверске технологије	Не (подршка само за Java софтверску технологију)	Да (подршка за преко 20 софтверских технологија)	Не (подршка само за Java софтверску технологију)	Не (подршка само за Java софтверску технологију)
Неопходност постојања изворног програмског кода	Да	Да	Не	Да
Интеграција са другим алатима за статичку анализу квалитета софтвера	Не	Да	Не (може се укључити у оквиру других алата)	Да (интеграција са SonarQube алатом)
Интеграција са алатима за управљање софтверским	Не	Да (Алати за управљање	Да (Алати за управљање	Не

Критеријум	**Swat4J**	**SonarQube**	**FindBugs**	**SilabMetrics**
пројектима		софтверским пројектима - Apache Maven, Apache Ant и JIRA, Алати за управљање верзијама - SVN, Git и Mercurial)	софтверским пројектима - Apache Maven и Apache Ant)	
Интеграција са развојним окружењима	Да (Eclipse развојно окружење)	Да (Eclipse и IntelliJ IDEA развојна окружења)	Да (Eclipse, IntelliJ IDEA и NetBeans развојна окружења)	Да (NetBeans развојно окружење)
Могућност израде додатака	Не	Да	Да	Не
Могућност програмског приступа резултатима анализе	Да	Да	Да	Не

6. Механизми побољшања објектно-оријентисаних софтверских система

У другом поглављу представљени су објектно-оријентисани софтверски системи и њихови најважнији концепти: класа и њене чланице (атрибути и методе), интерфејси, апстрактне класе, наслеђивање, полиморфизам. Објектно-оријентисани софтверски системи су нашли велику практичну примену у процесу животног циклуса развоја софтвера. Због тога се у оквиру овог поглавља разматрају механизми побољшања објектно-оријентисаних софтверских система и њихове везе са софтверским метрикама. У том смислу, најпре се даје преглед општих принципа пројектовања софтвера који се могу применити у процесу развоја свих софтверских система. Затим се даје преглед принципа објектно-оријентисаног развоја софтвера, као и стратегија пројектовања софтвера. Као значајни механизам побољшања објектно-оријентисаних софтверских система идентификовани су софтверски патерни па се због тога, у посебном одељку, даје њихов приказ. На крају, као механизам побољшања објектно-оријентисаних система идентификоване су методе развоја софтвера.

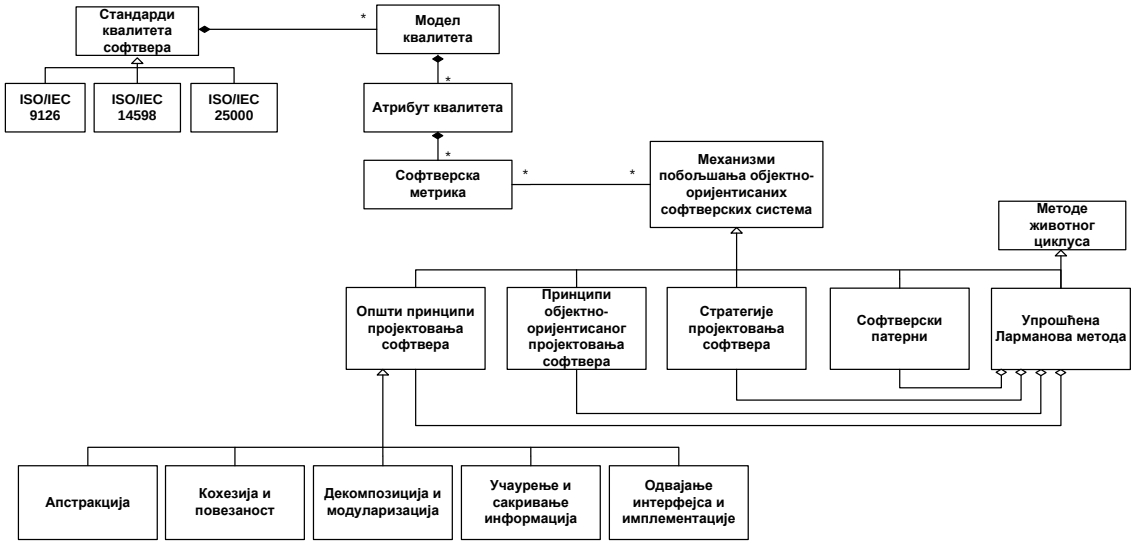
Концептуални преглед поглавља дат је на наредној слици (Слика 52).



Слика 52. Приказ механизама побољшања објектно-оријентисаних софтверских система

6.1. Општи принципи пројектовања софтвера

Општи принципи пројектовања софтвера представљају основне принципе који се користе у процесу пројектовања софтвера. Ови принципи могу се користити приликом пројектовања свих софтверског система. У том смислу разликују се следећи принципи пројектовања софтвера: апстракција, кохезија и повезаност, декомпозиција и модуларизација, учаурење и сакривање информација, и одвајање интерфејса и имплементације, као што је приказано на наредној слици (Слика 53) [Vlajic14].



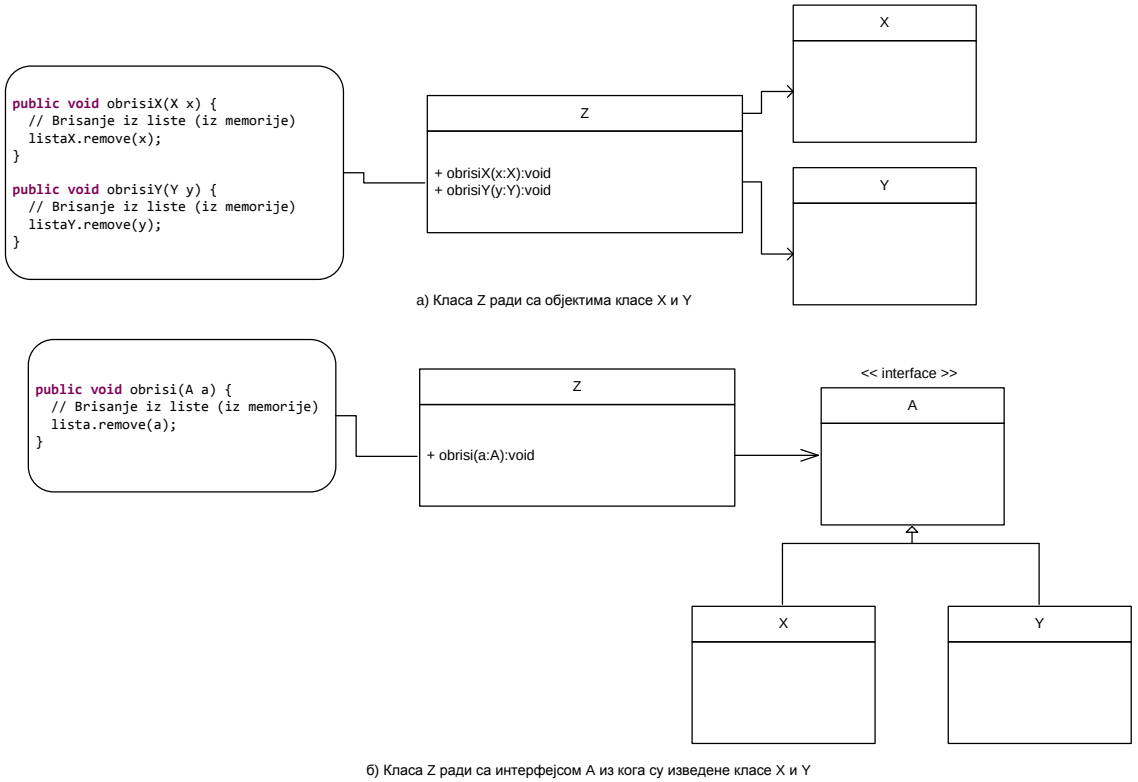
Слика 53. Општи принципи пројектовања софтвера

6.1.1. Апстракција

Апстракција је принцип пројектовања софтвера и усмерена је на постепено и контролисано увођење детаља који су од значаја за посматрани софтверски систем. Применом механизма апстракције издвајају се опште информације, како би се нагласила суштина неке појаве, док се детаљи о самој појави избегавају [Vlajic14].

Размотримо пример који је приказан на наредној слици (Слика 54а). Са слике се уочава класа Z која је непосредно повезана са класама X и Y. Због тога ће метрика *Повезаност објеката* имати вредност 2, док ће метрика *Број одговора класе* имати вредност 3. Такође, класа Z садржи методе *obrisiX()* и *obrisiY()* којима се омогућава брисање објеката класа X и Y, респективно. С

обзиром да методе раде над различитим скупом атрибута, метрика *Недостатак кохезивности метода у класи* имаће вредност 0.5. У оквиру посматраног примера није коришћен концепт наслеђивања па ће метрика *Дубина стабла наслеђивања* имати вредност 1, док ће метрика *Број подкласа* имати вредност 0.



Слика 54. Примена принципа апстракције

Уколико се у примеру уведе интерфејс A који је апстракција од класа X и Y тада се мењају наведене метрике (Слика 54б). Са слике се уочава да је класа Z повезана са интерфејсом A из кога су изведене класе X и Y. У том смислу је смањена повезаност објеката (постоји веза само са интерфејсом A), па ће због тога метрика *Повезаност објеката* имати вредност 1, док ће метрика *Број одговора класе* имати вредност 2. С друге стране, у оквиру класе постоји само једна метода која ради над свим атрибутима класе, па ће метрика *Недостатак кохезивности метода у класи* имати вредност 0. С обзиром да се у посматраном примеру користи концепт наслеђивања, интерфејс A има две имплементације па ће због тога метрика *Број подкласа* имати вредност

2. С друге стране, класе X и Y имплементирају интерфејс A па ће због тога те класе за метрику *Дубина стабла наслеђивања* имати вредност 2. У наредној табели (Табела 32) дат је упоредни приказ вредности посматраних метрика приликом примене принципа апстракције.

Табела 32. Упоредни приказ посматраних метрика приликом примене принципа апстракције

Софтверска метрика	**Принцип апстракције није примењен**			**Принцип апстракције је примењен**			
	Z	**X**	**Y**	**Z**	**A**	**X**	**Y**
Повезаност објеката	2	0	0	1	0	0	0
Број одговора класе	3	0	0	2	0	0	0
Недостатак кохезивности метода у класи	0.5	0	0	0	0	0	0
Дубина стабла наслеђивања	1	1	1	1	1	2	2
Број подкласа	0	0	0	0	2	0	0

На основу изложеног се може закључити да се применом принципа апстракције смањује повезаност између објеката и број одговора посматране класе. Такође, применом принципа апстракције омогућава се пројектовање класа које имају високу кохезију. С друге стране, применом принципа апстракције повећава се дубина стабла наслеђивања и број подкласа у посматраном софтверском систему. У том смислу се може рећи да се општи принцип пројектовања софтвера *Апстракција* може повезати са софтверским метрикама, што је и приказано у наредној табели (Табела 33).

Табела 33. Веза општег принципа пројектовања софтвера Апстракција са софтверским метрикама

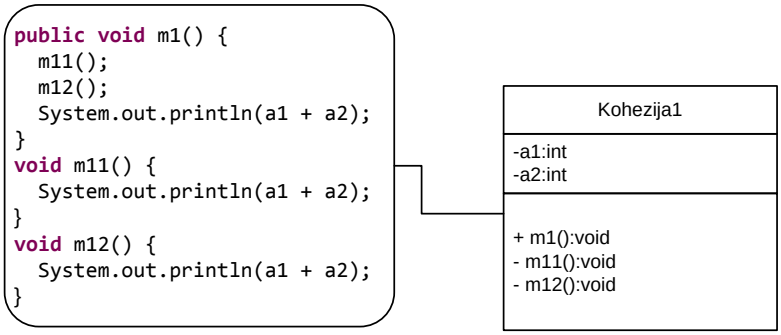
Општи принцип пројектовања софтвера	**Веза са софтверским метрикама**
Апстракција	- Повезаност објеката - Број одговора класе

Општи принцип пројектовања софтвера	**Веза са софтверским метрикама**
	- Недостатак кохезивности метода у класи - Дубина стабла наслеђивања - Број подкласа

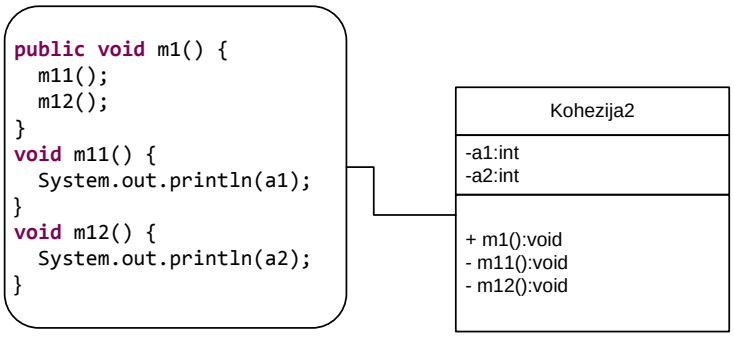
6.1.2. Кохезија и повезаност

Кохезија је принцип пројектовања софтвера који је усмерен на класу. У том смислу све чланице посматране класе (сви атрибути и методе које посматрана класа садржи) требају да буду у функцији посматране класе. Стога се може рећи да класа треба да садржи (тј. да учаури) само чланице које су неопходне за остваривање непосредне функције посматране класе или софтверске компоненте чији је класа део [Pressman10]. Другим речима, кохезија класе треба да буде висока. С друге стране, повезаност је усмерена на везу која се успоставља између различитих класа (тј. на однос између класа) [Pressman10]. У том смислу се разматра однос између класа, њихових метода и њихових атрибута па се може рећи да су две класе међусобно повезане уколико методе једне класе користе атрибуте или методе друге класе. Уколико се ниво повезаности (комуникације и сарадње) између класа повећа, повећава се и сложеност софтверског система [Pressman10]. Стога се може рећи да повезаност класа треба смањити и свести на најмању могућу меру, тј. да софтверски систем треба садржати класе које су независне или слабо повезане (идеално би било да се повезаност класа може остварити на нивоу интерфејса) [Vlajic14]. На основу изложеног се може закључити да приликом развоја софтверских система треба тежити да кохезија класе буде висока, док повезаност класа треба бити ниска [Rosenberg97]. На тај начин се зависност класа своди на најмању могућу меру уз истовремено повећавање једноставности измене, одржавања, тестирања и могућности поновног коришћења компоненти посматраног софтверског система.

Са наведеним принципом пројектовања софтвера непосредно је повезана софтверска метрика *Недостатак кохезивности метода у класи* која је директно усмерена на мерење нивоа недостатка кохезије. Размотримо пример који је дат на наредној слици (Слика 55). На слици је приказана класа Kohezija1 која садржи методу *m1* (из које се позивају методе *m11* и *m12*). Такође, класа Kohezija1 садржи атрибуте a1 и a2, при чему свака метода посматране класе приступа свим атрибутима класе. Из изложеног се може уочити непосредна повезаност метода са атрибутима унутар класе [Chidamber94]. С друге стране, може се уочити да су све методе посматране класе у функцији класе. Због тога се може закључити да је посматрана класа високо кохезивна, односно да не постоји недостатак кохезивности метода у класи (вредност метрике *Недостатак кохезивности метода у класи* биће нула).



а) Класа са високом кохезијом



б) Класа са недостатком кохезије

Слика 55. Недостатак кохезивности метода у класи

С друге стране, на слици је приказана класа Kohezija2 која садржи методу *m1* (из које се позивају методе *m11* и *m12*). Такође, класа Kohezija2 садржи атрибуте a1 и a2, при чему метода *m1()* не приступа атрибутима класе, метода *m11()* приступа атрибуту a1, док метода *m12()* приступа атрибуту a2. Другим речима, уочава се да методе класе раде над различитим скупом атрибута. Због тога се може закључити да је кохезија класе нарушена, тј. да постоји недостатак кохезивности метода у класи[8] (метрика *Недостатак кохезивности метода у класи* имаће вредност 1).

У наредној табели (Табела 34) дат је упоредни приказ вредности метрике *Недостатак кохезивности метода у класи* за посматрани пример.

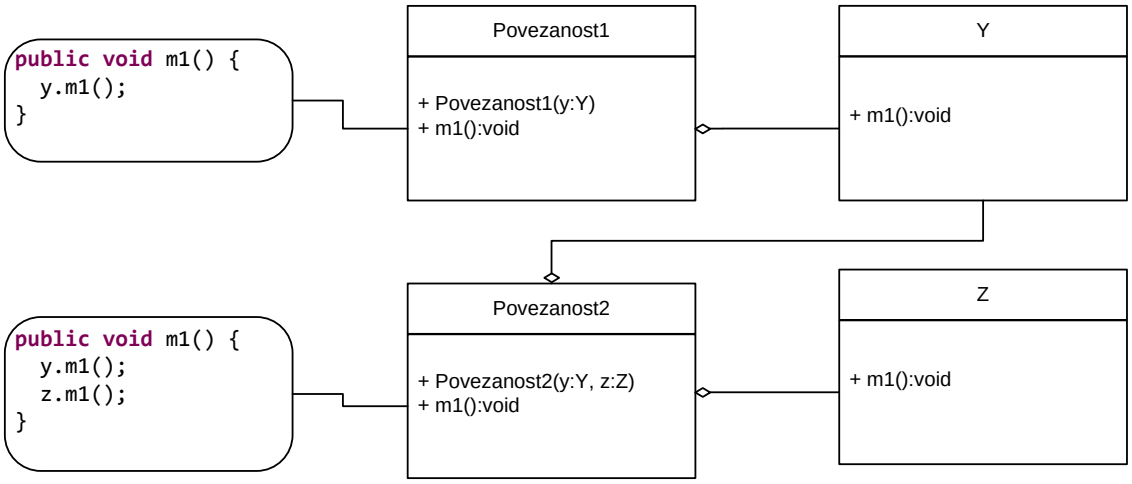
Табела 34. Упоредни приказ вредности метрике Недостатак кохезивности метода у класи за класе Kohezija1 и Kohezija1

Софтверска метрика	**Класа Kohezija1**	**Класа Kohezija2**
Недостатак кохезивности метода у класи	0	1

На основу изложеног се може закључити да уколико методе раде над различитим скупом атрибута долази до нарушавања кохезије, односно до повећања недостатка кохезивности метода у класи, што није пожељна особина у процесу развоја софтвера. У том смислу се посматрани принцип пројектовања софтвера може повезати са софтверском метриком *Недостатак кохезивности метода у класи*.

Такође, са наведеним принципом пројектовања софтвера непосредно су повезане софтверске метрике *Повезаност објеката* и *Број одговора класе* које су усмерене на мерење повезаности између класа. Размотримо пример који је дат на наредној слици (Слика 56).

[8] За мерење недостатка кохезивности метода у класи коришћена је софтверска метрика LCOM3.



Слика 56. Повезаност објеката

У наведеном примеру уочава се класа Povezanost1 која као атрибут има објекат класе Y. Из методе *m1()* класе Povezanost1 позива се метода *m1()* класе Y. Другим речима, уочава се повезаност између класа Povezanost1 и Y, тако да уколико разматрамо класу Povezanost1, вредност метрике *Повезаност објеката* износи 1. С обзиром да класа Povezanost1 дефинише методу *m1()* из које се позива метода *m1()* класе Y, вредност метрике *Број одговора класе износи* 2.

Са слике се, такође, уочава класа Povezanost2, као и веза класе Povezanost2 са класама Y и Z (класа Povezanost2 као атрибуте има објекте класа Y и Z). Из методе *m1()* класе Povezanost2 позивају се метода *m1()* класе Y и метода *m1()* класе Z. Због тога, уколико разматрамо класу Povezanost2, вредност метрике *Повезаност објеката* износи 2. Другим речима, класа Povezanost2 је непосредно повезана са две друге класе (тј. са класама Y и Z) што последично доводи до повећања вредности метрике *Повезаност објеката*. С обзиром да класа Povezanost2 дефинише методу *m1()* из које се позивају метода *m1()* класе Y и метода *m1()* класе Z, вредност метрике *Број одговора класе износи* 3.

У наредној табели (Табела 35) дат је упоредни приказ вредности метрике Повезаност објеката и Број одговора класе за посматрани пример.

Табела 35. Упоредни приказ вредности метрике Повезаност објеката и Број одговора класе за класе Povezanost1 и Povezanost2

Софтверска метрика	**Класа Povezanost1**	**Класа Povezanost2**
Повезаност објеката	1	2
Број одговора класе	2	3

Можемо закључити да уколико су класе повезане са већим бројем других класа долази и до повећања вредности метрика *Повезаност објеката* и *Број одговора класе*. У том смислу се посматрани принцип пројектовања софтвера може повезати са софтверским метрикама *Повезаност објеката* и *Број одговора класе*.

Из изложеног се може закључити да се општи принцип пројектовања софтвера *Кохезија и повезаност* може повезати са софтверским метрикама, што је и приказано у наредној табели (Табела 36). У том смислу се може рећи да су наведене софтверске метрике у функцији реализације наведеног принципа пројектовања софтвера.

Табела 36. Веза општег принципа пројектовања софтвера Кохезија и повезаност са софтверским метрикама

Општи принцип пројектовања софтвера	**Веза са софтверским метрикама**
Кохезија и повезаност	- Недостатак кохезивности метода у класи - Повезаност објеката - Број одговора класе

6.1.3. Декомпозиција и модуларизација

Декомпозиција представља принцип пројектовања софтвера који је усмерен на поделу софтверског система на више модула. Стога се може рећи да као резултат процеса декомпозиције софтверског система настаје

модуларизација софтверског система [Vlajic14]. У том смислу Влајић разматра декомпозицију кроз три аспекта пројектовања софтвера:

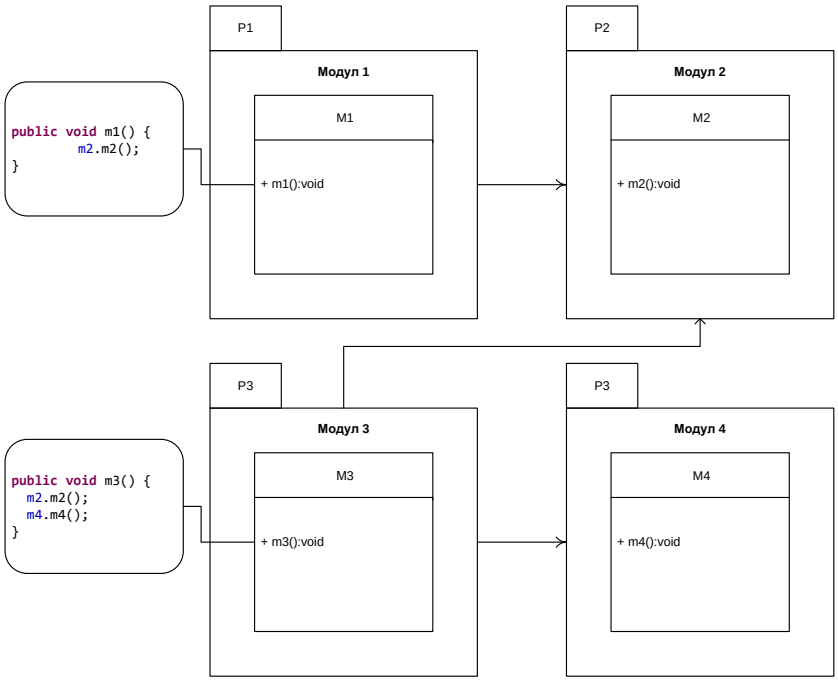
- Декомпозицију код прикупљања захтева (почетни кориснички захтев се декомпонује на скуп захтева који се могу представити на одговарајући начин, нпр. моделом случајева коришћења, при чему се сваки од тих захтева може даље декомпоновати).
- Декомпозицију код пројектовања софтвера (у том смислу се архитектура софтверског система може декомпоновати на више нивоа, при чему се, опет, сваки ниво може даље декомпоновати). На пример, тронивојска архитектура се може декомпоновати на ниво корисничког интерфејса, ниво апликационе логике и ниво складишта података. С друге стране, апликациона логика се даље може декомпоновати на контролер апликационе логике, пословну логику и брокер базе података.
- Декомпозицију функција (уколико је нека функција сложена може се декомпоновати у више функција; наведене функције се могу независно имплементирати а затим интегрисати у једну целину чиме се реализује почетна функција).

Декомпозиција и модуларизација код прикупљања захтева, пројектовања софтвера и функција представљају веома важне и често коришћене принципе пројектовања софтвера чиме се остварује поновно коришћење и проширивање модула софтверског система [Meyer88]. На тај начин се почетни проблем може поделити (декомпоновати) на једноставније потпроблеме (модуле) који се могу независно реализовати и који ће бити у функцији реализације посматраног софтверског система.

Приликом декомпозиције система у модуле, потребно је да модули имају јаку кохезију [Vlajic14], тј. потребно је да све чланице посматраног модула (класе, пакети и сл.) буду у функцији посматраног модула. Другим речима, модул треба да садржи само оне чланице које су неопходне за реализацију

функције модула. Такође, потребно је да модули буду слабо повезани [Vlajic14]. То значи да је потребно зависност између модула свести на најмању могућу меру.

Размотримо пример који је приказан на наредној слици (Слика 57). Са примера се може уочити класе M1, M2, M3 и M4 које припадају модулима Модул 1, Модул 2, Модул 3 и Модул 4 и налазе се у пакетима P1, P2, P3 и P4, респективно. Такође, може се уочити да су модули међусобно повезани тако што се из методе *m1()* првог модула позива метода *m2()* другог модула, док се из методе *m3()* трећег модула позивају метода *m2()* другог модула и метода *m4()* четвртог модула. Због успостављене зависности између модула, вредност софтверске метрике *Повезаност објеката* за модул M1 ће износити 1, док ће вредност исте метрике за модул M3 износити 2. Другим речима, како се зависност између модула повећава долази и до повећања вредности софтверске метрике *Повезаност објеката*. На основу изложеног се закључује да се општи принцип *Декомпозиција и модуларизација* непосредно може повезати са софтверском метриком *Повезаност објеката*.



Слика 57. Повезаност модула приликом декомпозиције

Са слике се, такође, уочава успостављена зависност између пакета, па се може применити Метрика Стабилности софтвера. С обзиром да модул M1 има једну одлазну повезаност, при чему долазна повезаност не постоји, вредност метрике стабилности модула износи Ir = 5.1 / 9.2 ≈ 0.55. С друге стране, модул M3 има две одлазне повезаности, при чему долазна повезаност не постоји, тако да вредност метрике стабилности модула износи Ir = 6.1 / 10.2 ≈ 0.60. Другим речима, како се зависност између модула повећава долази и до повећања вредности нестабилности пакета. На основу изложеног се закључује да се општи принцип *Декомпозиција и модуларизација* непосредно може повезати са софтверском метриком *Метрика стабилности софтвера*. У наредној табели (Табела 37) дат је упоредни приказ вредности софтверских метрика *Повезаност објеката* и *Метрика стабилности софтвера* за модуле M1 и M3.

Табела 37. Вредност софтверских метрика за модуле M1 и M3

Софтверска метрика	**Модул M1**	**Модул M3**
Повезаност објеката	1	2
Метрика стабилности софтвера	0.55	0.60

У том смислу изводимо закључак да се принцип пројектовања софтвера *Декомпозиција и модуларизација* може повезати са софтверским метрикама *Повезаност објеката* и *Метрика стабилности софтвера*. Уколико се приликом декомпозиције и модуларизације успостави зависност између модула долази до повећања вредности софтверских метрика *Повезаност објеката* и *Метрика стабилности софтвера*. Због тога би зависност модула требало свести на најмању могућу меру, чиме се последично смањује повезаност модула и повећава стабилност посматраног софтверског система.

С друге стране, приликом декомпозиције функција, почетна функција се може декомпоновати на више функција. Размотримо пример који је

приказан на наредној слици (Слика 58). Посматрани пример за интервал бројева [donja, gornja] проналази и приказује просте бројеве.

```
public void prostiBrojevi(int donja, int gornja) {
  if (donja > gornja || donja <= 0) {
    throw new RuntimeException("Granice nisu dobro unesene!");
  }
  for (int i = donja; i <= gornja; i++) {
    boolean prost = true;
    for (int j = 2; j <= i / 2; j++) {
      if (i % j == 0) {
        prost = false;
        break;
      }
    }
    System.out.println("Broj " + i + " je prost: " + prost);
  }
}
```

Dekompozicija1

+prostiBrojevi(int donja, int gornja):void

```
public void prostiBrojevi(int donja, int gornja) {
  proveraGranica(donja, gornja);
  for (int i = donja; i <= gornja; i++) {
    System.out.println("Broj " + i + " je prost: " + prostBroj(i));
  }
}

void proveraGranica(int donja, int gornja) {
  if (donja > gornja || donja <= 0) {
    throw new RuntimeException("Granice nisu dobro unesene!");
  }
}

boolean prostBroj(int broj) {
  for (int i = 2; i <= broj / 2; j++) {
    if (broj % i == 0) {
      return false;
    }
  }
  return true;
}
```

Dekompozicija2

+prostiBrojevi(int donja, int gornja):void
-proveraGranica(int donja, int gornja):void
-prostBroj(int broj):boolean

Слика 58. Декомпозиција функција

Са слике се уочава класа Dekompozicija1 која садржи методу *prostiBrojevi()*. Метода је имплементирана тако да се у њој директно врши провера да ли су границе добро унесене (уколико нису, генерише се изузетак), а затим се за сваки број из посматраног интервала врши провера да ли је прост. Другим речима, у оквиру класе Dekompozicija1 није извршена декомпозиција функција већ се сва програмска логика за решавање проблема налази у функцији *prostiBrojevi()*.

С друге стране, са слике се такође уочава класа Dekompozicija2 која садржи методе *prostiBrojevi()*, *proveraGranica()* и *prostBroj()*. Из методе *prostiBrojevi()* се позивају методе *proveraGranica()* и *prostBroj()* које су задужене за проверу граница и проверу да ли је посматрани број прост, респективно. Другим

речима, методе *proveraGranica()* и *prostBroj()* су у функцији реализације методе *prostiBrojevi()*.

У наредној табели (Табела 38) дат је приказ вредности метрика *Циклична сложеност*, *Сложеност пондерисаних метода*, *Број пондерисаних метода класе* и *Број наредби у методи* за класе Dekompozicija1 и Dekompozicija2. Из табеле се уочава да се приликом декомпозиције значајно смањују вредности метрика *Циклична сложеност*, *Сложеност пондерисаних метода* и *Број наредби у методи*, уз истовремено повећање вредности метрике *Број пондерисаних метода*.

Табела 38. Вредност софтверских метрика за класе Dekompozicija1 и Dekompozicija2

Софтверска метрика	**Dekompozicija 1**	**Dekompozicija 2**
Циклична сложеност	6	2
Сложеност пондерисаних метода	6	2
Број пондерисаних метода класе	1	3
Број наредби у методи	9	3

На тај начин је, након извршене декомпозиције, почетна функција непосредно повезана са функцијама у циљу решавања одређеног проблема чиме се остварује висока кохезија међу функцијама (у конкретном примеру ће вредност софтверске метрике *Недостатак кохезивности метода у класи* бити нула). На основу наведеног се закључује да се општи принцип пројектовања софтвера *Декомпозиција и модуларизација* непосредно може повезати и са софтверским метрикама *Сложеност пондерисаних метода*, *Број пондерисаних метода класе, Циклична сложеност, Недостатак кохезивности метода у класи* и *Број наредби у методи*.

Из изложеног се може закључити да се општи принцип пројектовања софтвера *Декомпозиција и модуларизација* може повезати са софтверским метрикама, што је и приказано у наредној табели (Табела 39). У том смислу

се може рећи да су наведене софтверске метрике у функцији реализације наведеног принципа пројектовања софтвера.

Табела 39. Веза општег принципа пројектовања софтвера Декомпозиција и модуларизација са софтверским метрикама

Општи принцип пројектовања софтвера	**Веза са софтверским метрикама**
Декомпозиција и модуларизација	- Сложеност пондерисаних метода - Број пондерисаних метода класе - Циклична сложеност - Недостатак кохезивности метода у класи - Повезаност објеката - Број наредби у методи - Метрика стабилности софтвера

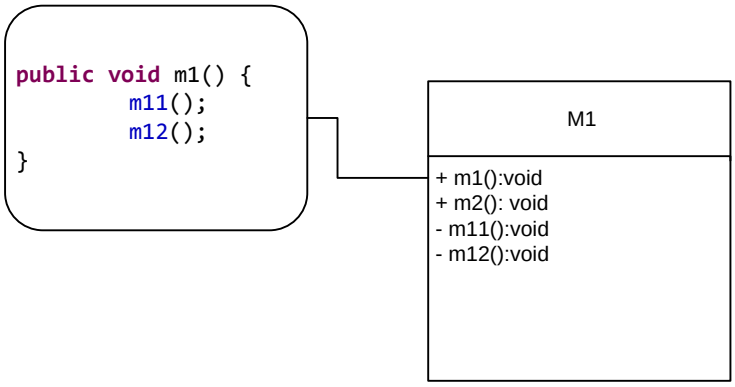
6.1.4. Учаурење и сакривање информација

Раније је објашњен принцип пројектовања софтвера који се назива *Кохезија* и који је усмерен на структуру и понашање класе. Према том принципу, кохезија класе треба бити висока, тј. све чланице посматране класе (сви атрибути и методе које посматрана класа садржи) требају да буду у функцији те класе. Другим речима, класа треба да садржи (тј. учаури) само чланице класе које су неопходне како би се остварила непосредна функција посматране класе или софтверске компоненте чији је посматрана класа саставни део.

Учаурење представља процес у оквиру којег се раздвајају особине модула у односу на остатак софтверског система [Vlajic14]. На тај начин се врши класификација особина софтверског система према начину приступа. У том смислу се разликују особине модула (нпр. класе) које су јавне за друге модуле и особине модула које су сакривене од других модула система. Особине модула које су јавне могу бити коришћене од стране других делова

софтверског система (њима је могуће непосредно приступити). С друге стране, особине модула које нису јавне не могу бити коришћене од стране других модула, тј. наведене особине остају скривене од других делова софтверског система (њима није могуће непосредно приступити). Јавне особине се стављају на располагање другим деловима софтверског система: преко јавних особина се, посредно, може приступити особинама модула које су сакривене од других делова софтверског система. У том смислу се може рећи да сакривање информација настаје као непосредна последица процеса учаурења [Vlajic14].

Размотримо пример који је приказан на наредној слици (Слика 59). Могу се уочити две јавне методе *m1()* и *m2()*, као и две приватне методе *m11()* и *m12()*. Такође, из методе *m1()* се позивају методе *m11()* и *m12()*. Другим речима, може се уочити да се кориснику на располагање стављају само јавне функције. Преко јавне методе m1 се посредно приступа до остатка система, тј. до метода *m11()* и *m12()*, али је начин имплементације метода *m11()* и *m12()* сакривен од корисника (он им не може приступити).



Слика 59. Учаурење и сакривање информација

Са слике се уочава да су методе *m11()* и *m12()* у функцији реализације методе *m1()*. С друге стране, методе *m1()* и *m2()* су методе којима се јавно излажу остатку софтверског система и оне су у функцији реализације класе M1. Другим речима, класа M1 има високу кохезију. У том смислу се са наведеним принципом пројектовања софтвера може повезати софтверска метрика

Недостатак кохезивности метода у класи, што је и приказано у наредној табели (Табела 40).

Табела 40. Веза општег принципа пројектовања софтвера Учаурење и сакривање информација са софтверским метрикама

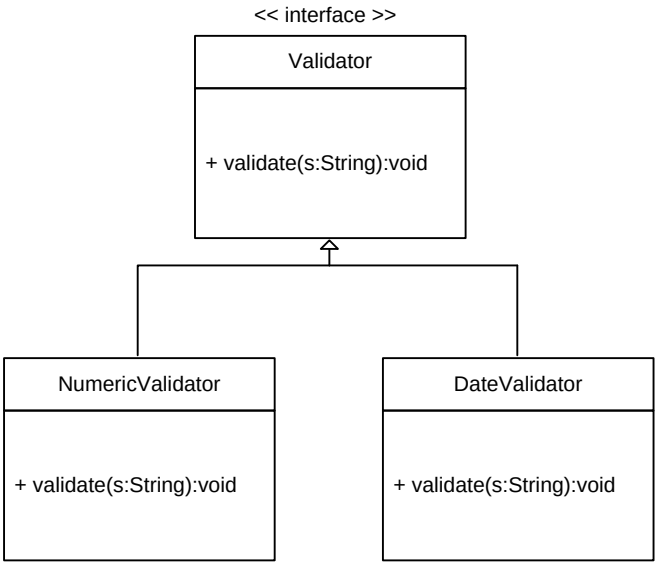
Општи принцип пројектовања софтвера	**Веза са софтверским метрикама**
Учаурење и сакривање информација	- Недостатак кохезивности метода у класи

6.1.5. Одвајање интерфејса и имплементације

Интерфејсом се даје спецификација функција посматраног модула. У процесу развоја софтвера често се јавља потреба да се спецификација одвоји од имплементације. На тај начин корисник посматраног софтверског система (нпр. инжењер за развој софтвера, други део софтверског система, други софтверски систем итд.) може приступити детаљима софтверског система али без неопходног познавања детаља о посматраном софтверском систему. У том смислу се софтверском систему не приступа директно, већ посредно, путем интерфејса. На тај начин детаљи имплементације софтверског система остају сакривени [Meyer88]. С друге стране, одвајање интерфејса од имплементације доноси још једну значајну предност: у процесу развоја софтвера је на тај начин могуће направити различите имплементације за посматрану спецификацију (тј. интерфејс). За реализацију наведених циљева могу се користити програмски концепти који се ослањају на интерфејсе. У том смислу се интерфејс одваја од имплементације софтверског система и излаже се кориснику, при чему корисник не зна како је интерфејс имплементиран [Vlajic14].

Размотримо пример који је приказан на наредној слици (Слика 60). Са слике се уочава интерфејс Validator који садржи методу *validate()* и има две класе које је реализују (NumericValidator и DateValidator). Другим речима,

интерфејсом се даје спецификација валидације, док су конкретне имплементације дате у класама које имплементирају наведени интерфејс.



Слика 60. Одвајање интерфејса од имплементације

С обзиром на примену концепта наслеђивања, са овим принципом пројектовања софтвера се може повезати софтверска метрика *Дубина стабла наслеђивања*. Она је усмерена на класу и одређивање њене позиције у хијерархији класа [Chidamber94]. Са посматраним принципом пројектовања софтвера се може повезати и софтверска метрика *Број подкласа*. Она је усмерена на класу одређивање броја њених непосредних потомака у хијерархији [Chidamber94]. Наредна табела (Табела 41) даје упоредни приказ вредности софтверских метрика *Дубина стабла наслеђивања* и *Број подкласа* за посматрани пример.

Табела 41. Упоредни приказ вредности софтверских метрика Дубина стабла наслеђивања и Број подкласа за посматрани пример

Класа	**Софтверска метрика**	
	Дубина стабла наслеђивања	**Број подкласа**
Validator	1	2
NumericValidator	2	0

Класа	**Софтверска метрика**	
	Дубина стабла наслеђивања	**Број подкласа**
DateValidator	2	0

Из изложеног се може закључити да се општи принцип пројектовања софтвера *Одвајање интерфејса и имплементације* може повезати са софтверским метрикама, што је и приказано у наредној табели (Табела 42). У том смислу се може рећи да су наведене софтверске метрике у функцији реализације наведеног принципа пројектовања софтвера.

Табела 42. Веза општег принципа пројектовања софтвера Одвајање интерфејса и имплементације са софтверским метрикама

Општи принцип пројектовања софтвера	**Веза са софтверским метрикама**
Одвајање интерфејса и имплементације	- Дубина стабла наслеђивања - Број подкласа

Принцип пројектовања софтвера *Одвајање интерфејса и имплементације* је такође повезан са принципом пројектовања софтвера *Учаурење и сакривање информација*. Ови принципи пројектовања софтвера су комплементарни, међусобно се допуњују и користе се заједно у процесу развоја софтвера.

У наредној табели (Табела 43) даје се сумарни приказ општих принципа пројектовања софтвера и њихових веза са софтверским метрикама.

Табела 43. Сумарни приказ општих принципа пројектовања софтвера и њихових веза са софтверским метрикама

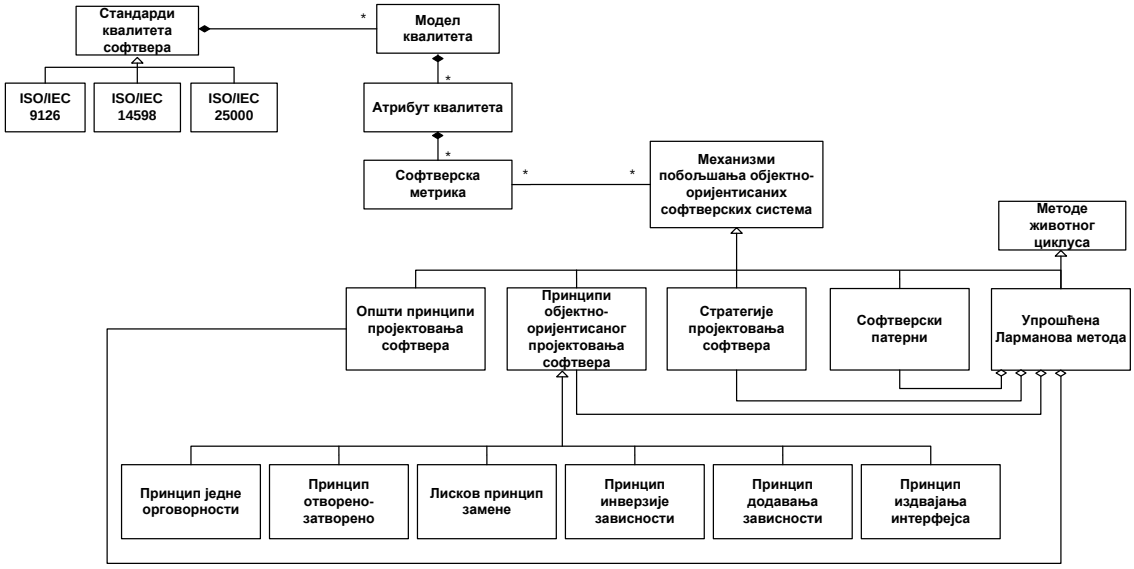
Општи принцип пројектовања софтвера	**Веза са софтверским метрикама**
Апстракција	- Повезаност објеката - Број одговора класе

Општи принцип пројектовања софтвера	**Веза са софтверским метрикама**
	- Недостатак кохезивности метода у класи - Дубина стабла наслеђивања - Број подкласа
Кохезија и повезаност	- Недостатак кохезивности метода у класи - Повезаност објеката - Број одговора класе
Декомпозиција и модуларизација	- Сложеност пондерисаних метода - Број пондерисаних метода класе - Циклична сложеност - Недостатак кохезивности метода у класи - Повезаност објеката - Број наредби у методи - Метрика стабилности софтвера
Учаурење и сакривање информација	- Недостатак кохезивности метода у класи
Одвајање интерфејса и имплементације	- Дубина стабла наслеђивања - Број подкласа

6.2. Принципи објектно-оријентисаног пројектовања софтвера

Раније је напоменуто да објектно-оријентисано пројектовање представља у пракси често коришћен приступ пројектовању софтверских система. Принципи објектно-оријентисаног пројектовања представљају основне принципе који се користе приликом објектно-оријентисаног програмирања и пројектовања класа. У том смислу се разликују *принцип једне одговорности* (енг. Single responsibility principle), *принцип отворено-затворено* (енг. Open-closed principle), *Лисков принцип замене* (енг. Liskov substitution principle), *принцип издвајања интерфејса* (енг. Interface segregation principle), *принцип инверзије зависности* (енг. Dependency

inversion principle) и *принцип додавања зависности* (енг. Dependency injection principle) [Martin96] [Vlajic14]. Наведени принципи се у литератури могу пронаћи под акронимом SOLID[9]. На наредној слици (Слика 61) приказани су принципи објектно-оријентисаног пројектовања софтвера.



Слика 61. Принципи објектно-оријентисаног пројектовања софтвера

6.2.1. Принцип једне одговорности

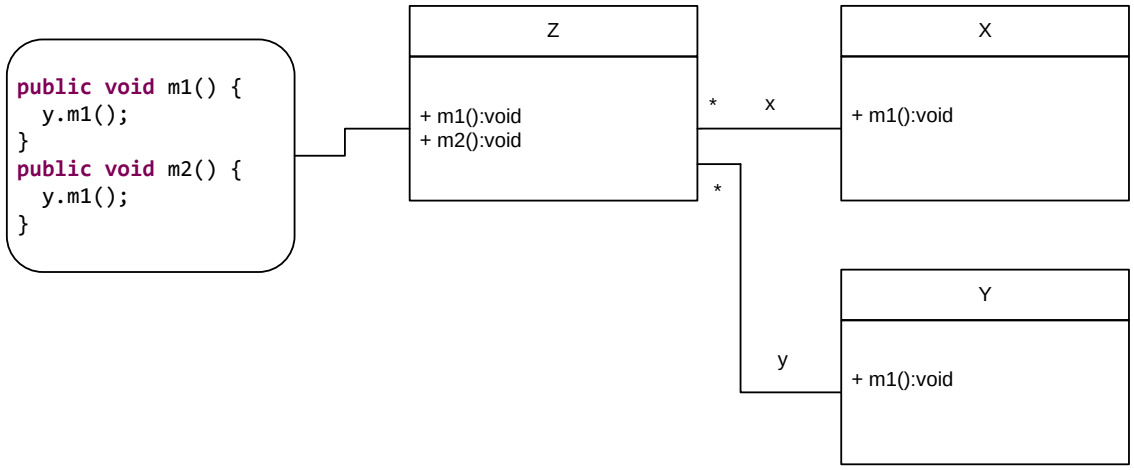
Принцип једне одговорности усмерен је на одговорност класе у софтверском систему. Према овом принципу класа треба да има једну одговорност у посматраном софтверском систему (уколико постоји више одговорности посматране класе постоји и више разлога за измену посматране класе). Стога је неопходно да све чланице посматране класе (атрибути и методе класе) требају да буду у функцији реализације те одговорности. На тај начин се остварује висока кохезија посматране класе. У том смислу се може рећи да је принцип једне одговорности непосредно повезан са принципом којим се промовише висока кохезија класе (објашњен је у оквиру одељка *Општи принципи пројектовања софтвера*). С друге

[9] Акроним SOLID је настао комбинацијом почетних слова сваког принципа (**S**ingle responsibility principle, **O**pen-closed principle, **L**iskov substitution principle, **I**nterface segregation principle и **D**ependency inversion/injection principle).

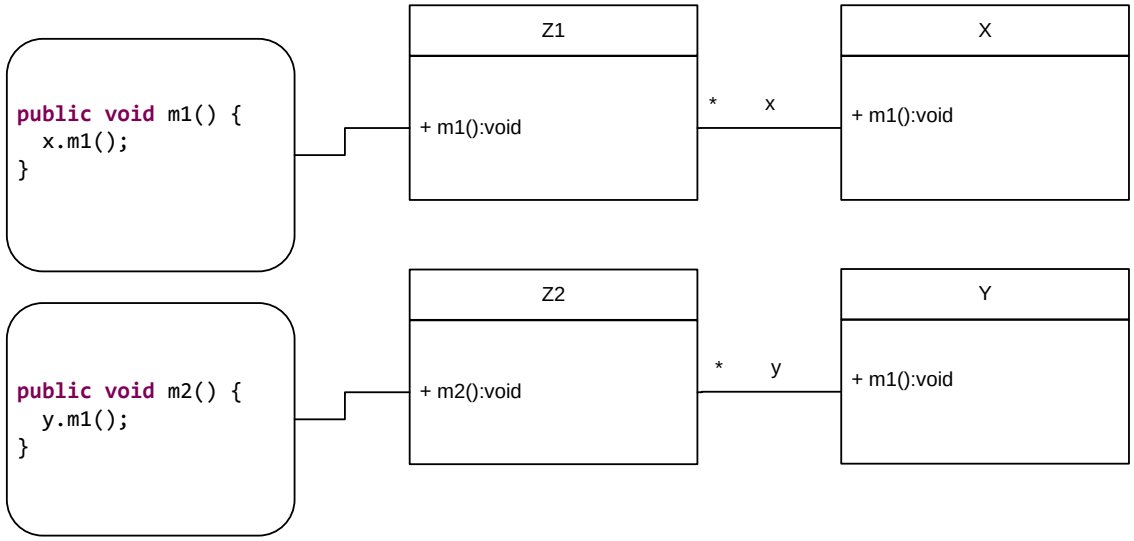
стране, уколико посматрана класа има више одговорности треба размотрити могућност поделе посматране класе на више класа, при чему би свака класа имала тачно једну одговорност [Martin96].

Размотримо пример који је приказан на наредној слици (Слика 62а). Са слике се уочава класа Z која је повезана са класама X и Y. Уколико разматрамо класу Z, вредност метрике *Повезаност објеката* имаће вредност 2. Такође, класа Z садржи методу *m1()* из које се позива метода *m1()* класе X, као и методу *m2()* из које се позива метода *m1()* класе Y. Другим речима, методе посматране класе се извршавају над различитим скупом атрибута, па ће због тога вредност метрике *Недостатак кохезивности метода у класи* износити 0.5. С обзиром на структуру класе Z, вредност метрике *Број одговора класе* износи 4.

Другим речима, посматрана класа има две одговорности, што последично отежава њен даљи развој и одржавање. Посматрани проблем се може решити поделом класе Z на две класе, што је такође приказано на слици (Слика 62б). Са слике се уочава класа Z1 која је повезана са класом X и класа Z2 која је повезана са класом Y. Уколико разматрамо класу Z1, вредност метрике *Повезаност објеката* имаће вредност 1. Уколико разматрамо класу Z2, вредност метрике *Повезаност објеката* ће такође имати вредност 1. Из методе *m1()* класе Z1 позива се метода *m1()* класе X, док се из методе *m2()* класе Z2 позива метода *m1()* класе Y. У том смислу се методе класе Z1 и методе класе Z2 извршавају над истим скупом атрибута, па ће због тога вредност метрике *Недостатак кохезивности метода у класи* износити 0. С обзиром на структуру класе Z1, вредност метрике *Број одговора класе* износи 2. Узимајући у обзир структуру класе Z2, вредност метрике *Број одговора класе* износи 2. Другим речима, класа Z1 има једну одговорност и класа Z2 има једну одговорност.



а) Класа Z има две одговорности



б) Класе Z1 и Z2 имају по једну одговорност

Слика 62. Принцип једне одговорности

У наредној табели (Табела 44) дат је упоредни приказ вредности метрика *Повезаност објеката*, *Недостатак кохезивности метода у класи* и *Број одговора класе* за посматрани пример.

Табела 44. Упоредни приказ вредности софтверских метрика за класе Z, Z1 и Z2

Софтверска метрика	**Z (Класа има две одговорности)**	**Z1 (Класа има једну одговорност)**	**Z2 (Класа има једну одговорност)**
Повезаност објеката	2	1	1

Софтверска метрика	**Z (Класа има две одговорности)**	**Z1 (Класа има једну одговорност)**	**Z2 (Класа има једну одговорност)**
Недостатак кохезивности метода у класи	0.5	0	0
Број одговора класе	4	2	2

Из упоредног приказа се може закључити да уколико посматрана класа има више одговорности долази до повећања вредности метрика *Повезаност објеката*, *Недостатак кохезивности метода у класи* и *Број одговора класе*. С друге стране, уколико посматрана класа има само једну одговорност, вредности посматраних метрика се смањују.

На основу изложеног се може закључити да се принцип једне одговорности може повезати са софтверском метриком *Недостатак кохезивности метода у класи*. Ова софтверска метрика усмерена је на класу, тј. на атрибуте и методе посматране класе. Потребно је да недостатак кохезије буде мали (идеално је да не постоји недостатак кохезивности метода у класи). Принцип једне одговорности се, такође, може повезати и са софтверским метрикама *Повезаност објеката* и *Број одговора класе*. Ове софтверске метрике усмерене су на однос између класа и потребно је да имају што мању вредност. У наредној табели (Табела 45) приказана је веза принципа једне одговорности са софтверским метрикама.

Табела 45. Веза принципа објектно-оријентисаног пројектовања софтвера Принцип једне одговорности са софтверским метрикама

Принцип објектно-оријентисаног пројектовања софтвера	**Веза са софтверским метрикама**
Принцип једне одговорности	- Недостатак кохезивности метода у класи - Повезаност објеката

Принцип објектно-оријентисаног пројектовања софтвера	**Веза са софтверским метрикама**
	- Број одговора класе

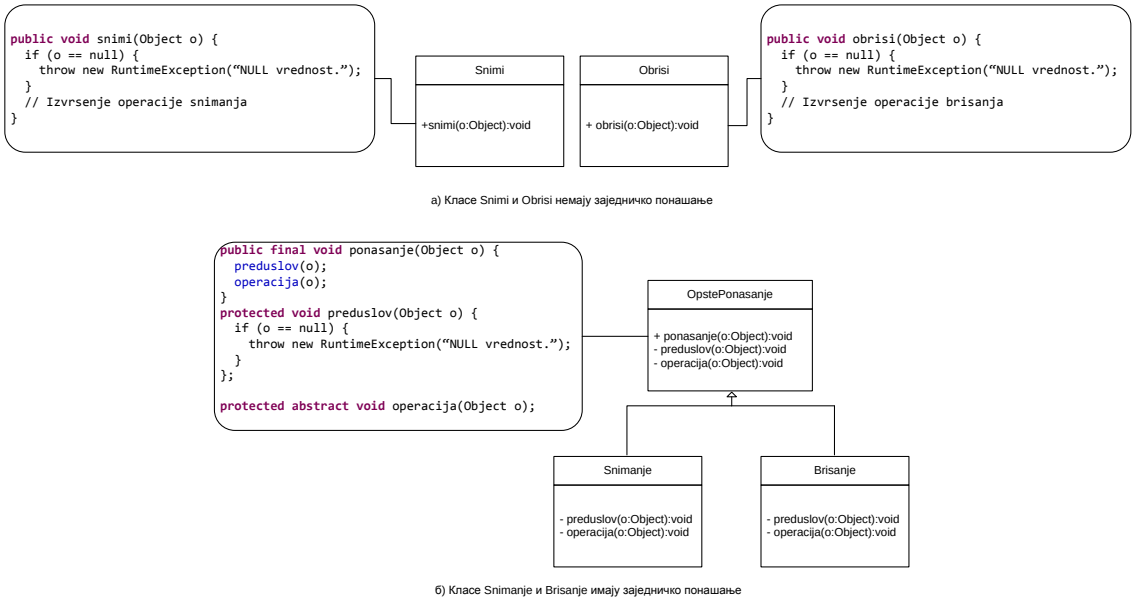
6.2.2. Принцип отворено-затворено

Ентитети (нпр. класе, модули, функције итд.) у процесу објектно-оријентисаног развоја софтвера требају да буду отворени за проширења али и затворени за модификацију [Martin96][Vlajic14]. Другим речима, у процесу развоја софтвера треба пројектовати класе, модуле и функције који се неће мењати. С друге стране, уколико је потребно извршити измене у софтверском систему, нова функционалност се додаје проширивањем постојећих ентитета а не њиховом изменом [Martin96]. У том смислу се додавање нове функционалности не врши изменом постојећег програмског кода већ непосредним додавањем новог програмског кода (део софтверског система који је већ написан се не мења). На тај начин се пројектује софтверски систем који је једноставан за одржавање и проширивање [Martin96].

Размотримо пример који је приказан на наредној слици (Слика 63а). Са слике се уочавају класе Snimi и Obrisi које су задужене за извршење операција снимања и брисања неког објекта, респективно. При томе, приликом извршења операције врши се провера предуслова да ли је снимање или брисање објеката дозвољено. Може се закључити да је имплементација посматраних операција слична, иако се поједини кораци у имплементацији разликују. Међутим, посматране класе немају заједничко понашање.

Са слике (Слика 63б) се такође уочава класа OpstePonasanje чија је одговорност да дефинише алгоритам извршења неке операције (у

конкретном примеру се разматра операција чувања објекта и операција брисања објекта). Алгоритам извршења дефинисан је методом *ponasanje()* и састоји се од два корака: најпре се врши провера предуслова (користи се метода *preduslov()*), а затим се непосредно извршава сама операција (користи се метода *operacija()*). Са слике се, такође, уочавају класе Snimanje и Brisanje чије су одговорности снимање и брисање података о неком објекту, респективно. Другим речима, класама Snimanje и Brisanje је дозвољено да промене неке кораке алгоритма (тј. начин провере предуслова и/или операције), при чему сама промена алгоритма није дозвољена, што представља суштину објектно-оријентисаног принципа пројектовања отворено-затворено.



Слика 63. Принцип отворено-затворено

Са наведеним принципом објектно-оријентисаног пројектовања софтвера може се повезати софтверска метрика *Дубина стабла наслеђивања*. У том смислу је могуће дефинисати позицију сваке класе у хијерархији наслеђивања. У конкретном примеру, класа OpstePonasanje се налази на дубини 1, док се изведене класе Snimanje и Brisanje налазе на дубини 2. С друге стране, класе Snimi и Obrisi се налазе на дубини 0.

Са принципом се, такође, може повезати софтверска метрика *Број подкласа*. С обзиром да се поједини кораци алгоритма могу редефинисати, могуће је одредити број потомака сваке посматране класе у хијерархији класа. У том смислу класа OpstePonasanje има два потомка којима се реализују наведене операције, док класе Snimanje и Brisanje немају потомке. С друге стране, класе Snimi и Obrisi немају потомке.

Са принципом отворено-затворено се могу повезати и софтверске метрике *Сложеност пондерисаних метода, Број пондерисаних метода, Број наредби у методи* и *Циклична сложеност*. С обзиром да се одговорност за реализацију појединих корака алгоритма преноси на подкласе, долази до смањења сложености метода у класи у којој је дефинисано опште понашање, што последично доводи до лакшег одржавања свих класа у хијерархији наслеђивања. У наредној табели (Табела 46) дат је упоредни приказ посматраних метрика за класе Snimi, Obrisi, OpstePonasanje, Snimanje и Brisanje.

Табела 46. Упоредни приказ вредности метрика за класе Snimi, Obrisi, OpstePonasanje, Snimanje и Brisanje

Софтверска метрика	**Класе немају заједничко понашање**		**Класе имају заједничко понашање (примењен је принцип Отворено-затворено)**		
	Snimi	**Obrisi**	**OpstePonasanje**	**Snimanje**	**Brisanje**
Дубина стабла наслеђивања	1	1	1	2	2
Број подкласа	0	0	2	0	0
Сложеност пондерисаних метода	2	2	4	1	1
Број пондерисаних метода	1	1	3	1	1
Број наредби у методи	2	2	2	0	0
Циклична сложеност	2	2	2	1	1

На основу приказаног се може приметити да се након примене принципа Отворено-затворено повећавају вредности софтверских метрика *Дубина стабла наслеђивања* и *Број подкласа*. С друге стране, уколико се разматрају метрике *Сложеност пондерисаних метода, Број пондерисаних метода, Број наредби у методи* и *Циклична сложеност*, примећује се да су њихове вредности у класи OpstePonasanje исте или се повећавају, али се зато њихове вредности смањују у класама Snimanje и Brisanje.

Из изложеног се може закључити да се принцип отворено-затворено може повезати са софтверским метрикама, што је и приказано у наредној табели (Табела 47). У том смислу се може рећи да су наведене софтверске метрике у функцији реализације наведеног принципа пројектовања софтвера.

Табела 47. Веза принципа објектно-оријентисаног пројектовања софтвера Принцип отворено-затворено са софтверским метрикама

Принцип објектно-оријентисаног пројектовања софтвера	**Веза са софтверским метрикама**
Принцип отворено-затворено	- Дубина стабла наслеђивања - Број подкласа - Сложеност пондерисаних метода - Број пондерисаних метода - Број наредби у методи - Циклична сложеност

Са принципом отворено-затворено је непосредно повезан Лисков принцип замене. Ови принципи се користе заједно у процесу објектно-оријентисаног развоја софтвера.

6.2.3. Лисков принцип замене

Лисков принцип замене је веома често коришћен принцип у развоју објектно-оријентисаних софтверских система. Објектно-оријентисани

софтверски системи састоје се од класа које непосредно сарађују. У класама су, путем атрибута и метода, учаурени структура и понашање софтверског система, респективно. Сарадња између класа одвија се непосредним позивом чланица (атрибута и метода) једне класе из чланица (метода) друге класе. У том смислу функције које користе показиваче или референце ка основним класама (тј. надкласама) морају бити у могућности да користе и објекте изведених класа (тј. подкласа) [Martin96]. Другим речима, подкласе требају бити замењиве са њиховим надкласама [Vlajic14].

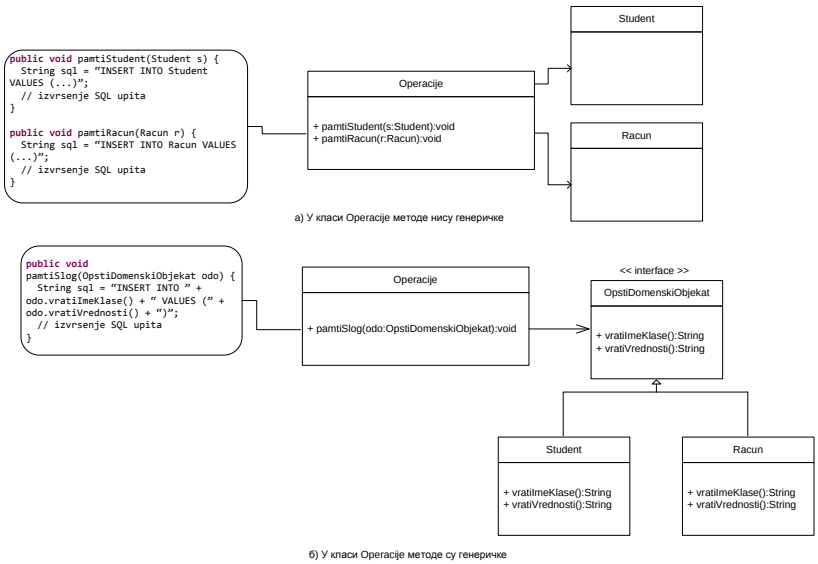
За реализацију овог принципа примењују се концепти објектно-оријентисаног развоја софтвера: наслеђивање и полиморфизам. Применом овог принципа омогућава се пројектовање компоненти софтверског система које се могу поново користити у процесу развоја софтвера (на пример, могуће је пројектовање метода којима ће се дефинисати генеричко понашање - у том смислу се пројектује понашање које садржи референцу ка надкласи; понашање које је дефинисано над објектом надкласе важи и за све изведене подкласе - методе пројектоване на овај начин се често називају генеричке методе).

С друге стране, у процесу објектно-оријентисаног развоја софтвера се не препоручује зависност од конкретних класа: у том смислу би за сваку нову класу било потребно дефинисање нове зависности што у значајној мери отежава одржавање и проширивање посматраног софтверског система. Поред тога, у том случају би такође био нарушен принцип отворено-затворено према којем модули у процесу објектно-оријентисаног развоја софтвера требају да буду отворени за проширења али и затворени за модификацију (додавањем нове зависности се, практично, врши нова измена посматраног модула, што није у складу са принципом отворено-затворено) [Martin96][Vlajic14]. Другим речима, Лисков принцип замене је непосредно повезан са принципом отворено-затворено: они су комплементарни, међусобно се допуњују и заједно се користе у процесу објектно-оријентисаног развоја софтверских система. Применом ових

принципа пројектује се софтверски систем који је једноставан за одржавање и проширивање [Martin96].

Размотримо пример који је приказан на наредној слици (Слика 64а). Са слике се уочава класа Operacije која дефинише методе *pamtiStudent()* и *pamtiRacun()* чиме се омогућава памћење података о студенту и рачуну, респективно. У том смислу, уколико би се у софтверском систему додао нови објекат кога треба запамтити, у класи Operacije је потребно написати нову методу чиме се омогућава памћење тог објекта. Класа Operacije је директно зависна од конкретних класа. Другим речима, методе у класи Operacije нису генеричке.

Размотримо и пример који је приказан на наредној слици (Слика 64б). Са слике се уочава класа Operacije која дефинише методу *pamtiSlog()* која као параметар има интерфејс OpstiDomenskiObjekat. Ова метода дефинише генеричко понашање за чување свих објеката који задовољавају услов да имплементирају интерфејс OpstiDomenskiObjekat (у конкретном примеру то су класе Student и Racun). Другим речима, уколико је у систему потребно додати нови објекат, није потребно написати нову методу за чување, већ је потребно да се имплементира интерфејс OpstiDomenskiObjekat.



Слика 64. Лисков принцип замене

У наредној табели (Табела 48) дат је упоредни приказ вредности софтверских метрика *Дубина стабла наслеђивања*, *Број подкласа, Повезаност објеката, Број одговора класе* и *Недостатак кохезивности метода у класи* за посматрани пример.

Табела 48. Упоредни приказ вредности метрика за класе Operacije, Student, Racun и OpstiDomenskiObjekat

Софтверска метрика	У класи Operacije методе нису генеричке		У класи Operacije методе су генеричке (примењен је Лисков принцип замене)		
	Operacije	**Student, Racun**	**Operacije**	**Opsti Domenski Objekat**	**Student, Racun**
Повезаност објеката	2	0	1	0	0
Недостатак кохезивности метода у класи	0.5	0	0	0	0
Број одговора класе	4	0	3	2	2

С обзиром да се користи концепт наслеђивања, са посматраним принципом објектно-оријентисаног пројектовања софтвера може се повезати софтверска метрика *Дубина стабла наслеђивања*. У том смислу је могуће дефинисати позицију сваке класе у хијерархији наслеђивања. Са принципом се, такође, може повезати софтверска метрика *Број подкласа*, чиме се омогућава одређивање броја потомака сваке посматране класе у хијерархији класа. На основу приказаних вредности софтверских метрика закључује се да се применом *Лисковог приниципа замене*, вредности ових метрика повећавају.

На основу вредности софтверских метрика закључује се да се са наведеним принципом могу повезати и софтверске метрике *Повезаност објеката* и *Број одговора класе*. С обзиром да се генеричком методом дефинише опште понашање, све специфичности се смештају у класе које имплементирају посматрани интерфејс. На тај начин долази до смањења повезаности

објеката и броја одговора класе у којој је дефинисано генеричко понашање, што последично доводи до лакшег одржавања софтверског система.

Поред тога, све методе класе у којој се дефинише генеричко понашање су у функцији класе и у процесу имплементације раде искључиво са надкласом, чиме се постиже висока кохезија класе која садржи генеричко понашање. У том смислу се посматрани принцип може повезати са софтверском метриком *Недостатак кохезивности метода у класи*. На основу приказаних вредности софтверских метрика закључује се да се применом *Лисковог приниципа замене* вредност метрике *Недостатак кохезивности метода у класи* смањује.

Из изложеног се може закључити да се принцип објектно-оријентисаног пројектовања софтвера Лисков принцип замене може повезати са софтверским метрикама, што је и приказано у наредној табели (Табела 49). У том смислу се може рећи да су наведене софтверске метрике у функцији реализације наведеног принципа пројектовања софтвера.

Табела 49. Веза принципа објектно-оријентисаног пројектовања софтвера Лисков принцип замене са софтверским метрикама

Принцип објектно-оријентисаног пројектовања софтвера	**Веза са софтверским метрикама**
Лисков принцип замене	- Дубина стабла наслеђивања - Број подкласа - Повезаност објеката - Број одговора класе - Недостатак кохезивности метода у класи

6.2.4. Принцип инверзије зависности

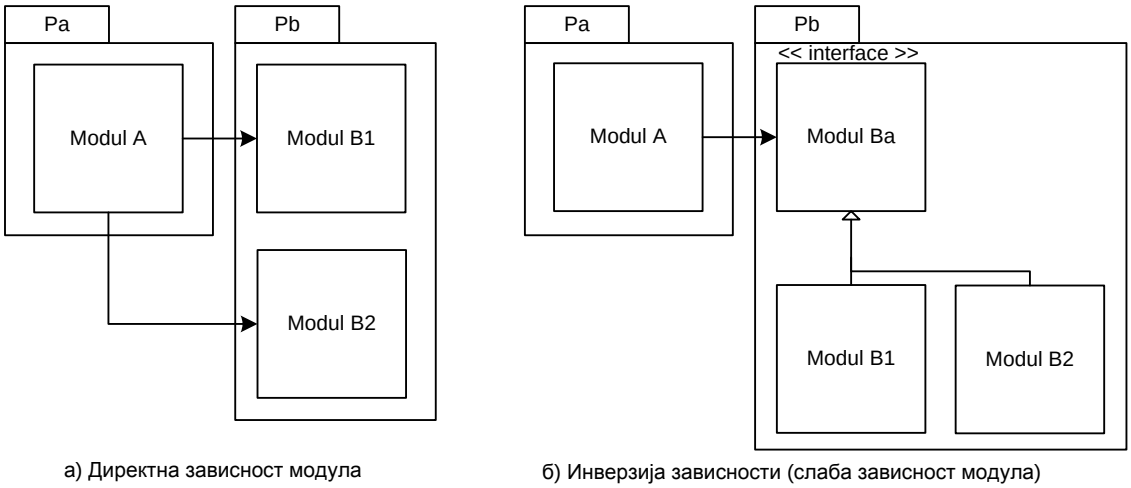
У процесу објектно-оријентисаног развоја софтвера креирају се модули који међусобно сарађују (у смислу да један модул позива операције другог

модула). Посматрани модули могу бити на различитим нивоима хијерархије. У том смислу модули вишег нивоа не треба да зависе од модула нижег нивоа: и модули вишег нивоа и модули нижег нивоа требају бити зависни од апстракције [Martin96]. С друге стране, апстракција не треба да зависи од детаља већ детаљи требају бити зависни од апстракције [Martin96]. Другим речима, применом принципа инверзије зависности избегава се директна зависност између модула већ се та зависност остварује посредно, путем апстракције (најбоље би било да се зависност између модула остварује на нивоу интерфејса). Уколико би постојала директна зависност између модула (у смислу њихове комуникације и сарадње) повећала би се и сложеност софтверског система што би последично отежало надоградњу и одржавање посматраног софтверског система. На пример, у случају постојања директне зависности између модула, промена или додавање новог модула нижег нивоа захтевала би промену и у оквиру модула вишег нивоа: у том случају би повезаност модула била висока. Због тога се, као што је раније напоменуто, у процесу развоја софтвера тежи да се повезаност модула сведе на најмању могућу меру. У том контексту је потребно елиминисати директну зависност између модула што се остварује увођењем још једног нивоа индирекције који заправо представља апстрактни модул. У том смислу се врши раскидање директне зависности али последично долази до успостављања нових веза [Vlajic14]:

- Успоставља се зависност између модула високог нивоа и апстрактног модула.
- Успоставља се зависност између модула ниског нивоа и апстрактног модула.

На тај начин се елиминише директна зависност између модула високог нивоа и модула ниског нивоа - она се остварује посредно, коришћењем апстрактног модула (тј. коришћењем апстракције). Другим речима, модул високог нивоа непосредно комуницира са апстрактним модулом, док

апстрактни модул непосредно комуницира са конкретним модулом на нижем нивоу хијерархије. Апстрактни модул представља посредника у комуникацији. Апстрактни модул се може реализовати на различите начине а најчешће се реализује коришћењем интерфејса и апстрактних класа. На тај начин се смањује директна повезаност између модула при чему је истовремено повећава могућност повезивања модула на вишем нивоу апстракције са различитим конкретним модулима на нижем нивоу апстракције (на пример, у времену извршавања програма је могуће извршити повезивање са различитим модулима на нижем нивоу хијерархије). На наредној слици (Слика 65) дат је приказ директне зависности модула Modul A од модула Modul B1 и Modul B2, као и завиности модула Modul A од апстрактног модула Modul Ba. Претпоставка је да се Modul A налази у пакету Pa док се модули Modul B1, Modul B2 и Modul Ba налазе у пакету Pb.



Слика 65. Принцип инверзије зависности

У наредној табели (Табела 50) дат је приказ вредности софтверских метрика Повезаност објеката, Дубина стабла наслеђивања и Број подкласа за посматрани пример.

Табела 50. Упоредни приказ вредности метрика за модуле A, B1, B2 и Ba

Софтверска метрика	**Директна зависност модула**			**Инверзија зависности модула (слаба зависност модула)**			
	A	**B1**	**B2**	**A**	**Ba**	**B1**	**B2**
Повезаност објеката	2	0	0	1	0	0	0
Дубина стабла наслеђивања	1	1	1	1	1	2	2
Број подкласа	0	0	0	0	2	0	0

С обзиром на избегавање директне зависности каже се да се применом принципа инверзије зависности смањује повезаност објеката [Vlajic14]. На тај начин се омогућава једноставно поновно коришћење модула који се налазе на вишим нивоу хијерархије - измена модула на вишем нивоу хијерархије може довести до измене модула на нижем нивоу хијерархије; измена модула на нижем нивоу хијерархије не доводи до измене модула на вишем нивоу хијерархије [Martin96]. На основу тога се може закључити да се овај принцип објектно-оријентисаног развоја софтвера може повезати са софтверском метриком *Повезаност објеката*.

С обзиром на примену концепта наслеђивања, принцип инверзије зависности се може повезати и са софтверским метрикама *Дубина стабла наслеђивања* којом се може одредити позиција посматраног модула у хијерархији наслеђивања. С друге стране, принцип се може повезати и са софтверском метриком *Број подкласа* којом се може одредити број конкретних имплементација посматраног модула.

У наредној табели (Табела 51) дат је приказ вредности метрике *Метрика стабилности софтвера* за пакет Pa. Примећује се да код директне зависности модула пакет има две одлазне повезаности а нема долазне повезаности. С друге стране, код инверзије зависности модула пакет има једну одлазну повезаност а нема долазних повезаности. На основу

изложеног се закључује да се приликом инверзије зависности модула нестабилност посматраног пакета смањује.

Табела 51. Упоредни приказ вредности Метрике стабилности софтвера за пакет Pa

Софтверска метрика	**Директна зависност модула**	**Инверзија зависности модула**
	Ig пакета Pa	**Ig пакета Pa**
Метрика стабилности софтвера	0.60	0.55

Из изложеног се може закључити да се принцип објектно-оријентисаног пројектовања софтвера Принцип инверзије зависности може повезати са софтверским метрикама, што је и приказано у наредној табели (Табела 52). У том смислу се може рећи да су наведене софтверске метрике у функцији реализације наведеног принципа пројектовања софтвера.

Табела 52. Веза принципа објектно-оријентисаног пројектовања софтвера Принцип инверзије зависности са софтверским метрикама

Принцип објектно-оријентисаног пројектовања софтвера	**Веза са софтверским метрикама**
Принцип инверзије зависности	- Повезаност објеката - Дубина стабла наслеђивања - Број подкласа - Метрика стабилности софтвера

На крају, потребно је напоменути да се начин на који се врши непосредно повезивање модула на вишем нивоу хијерархије са модулом на нижем нивоу хијерархије дефинише принципом додавања зависности који ће бити објашњен у наставку. Применом овог принципа инверзије зависности омогућава се креирање компоненти софтверског система које се могу

поново користити што последично олакшава одржавање и надоградњу објектно-оријентисаног софтверског система [Martin96].

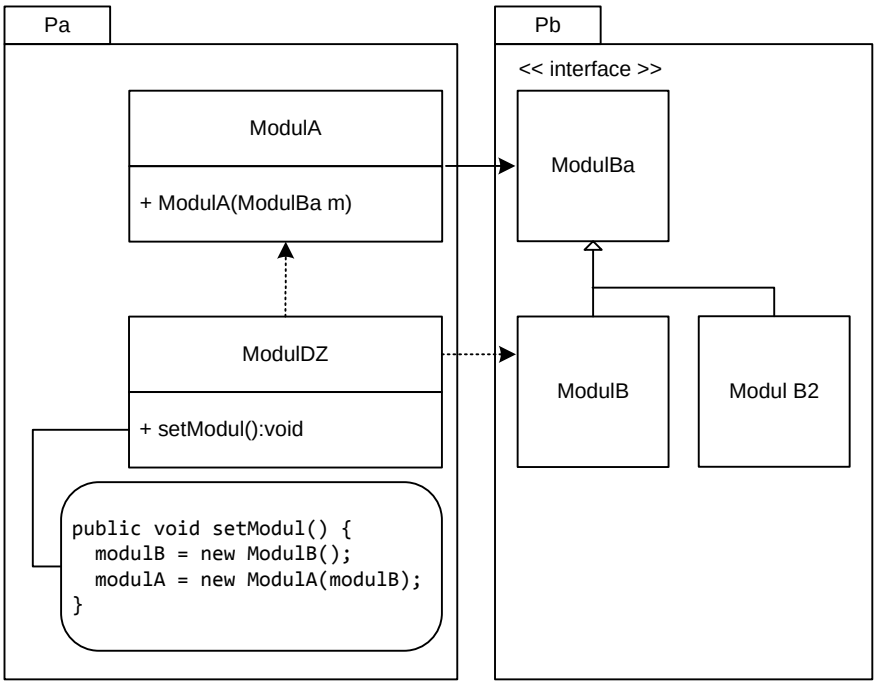
6.2.5. Принцип додавања зависности

На основу принципа инверзије зависности може се закључити да се у процесу објектно-оријентисаног развоја софтвера избегава директна повезаност и зависност између модула на вишем нивоу хијерархије и модула на нижем нивоу хијерархије. Посматрана зависност се остварује посредно, путем апстракције (тј. апстрактног модула). Међутим, поставља се питање саме реализације зависности, тј. начина на који се врши непосредно повезивање модула на вишем нивоу хијерархије са модулом на нижем нивоу хијерархије. Према принципу додавања зависности, зависност између две компоненте објектно-оријентисаног софтверског система се конкретно успоставља у време извршавања програма, коришћењем неке треће компоненте [Vlajic14]. Та трећа компонента је задужена да, коришћењем одговарајућег механизма, успостави конкретну зависност између модула на вишем нивоу хијерархије и конкретног модула на нижем нивоу хијерархије. Постоје различити механизми за успостављање зависности између модула [Vlajic14]:

- успостављање зависности путем конструктора (енг. Constructor injection),
- успостављање зависности путем методе (енг. Method injection) и
- успостављање зависности путем поља (енг. Field injection).

Сваки од наведених механизама је специфичан али се суштина огледа у томе да се у оквиру модула вишег нивоа хијерархије уводи референца на апстрактни модул па се путем конструктора, методе или поља омогућава дефинисање зависности. При томе, спецификација конкретних компоненти између којих се успоставља зависност се дефинише на посебан начин (нпр. коришћењем програмске логике софтверског система, коришћењем XML датотека, коришћењем текстуалних датотека, коришћењем датотека које

чувају уређене парове кључ-вредност, коришћењем анотација над класом, методама класе и/или атрибутима класе итд.). У времену извршавања програма врши се учитавање конкретно специфицираних зависности (из програмске логике софтверског система, датотека или анотација) и користи се механизам рефлексије чиме се омогућава непосредно успостављање специфицираних зависности, што је и приказано на наредној слици (Слика 66). На слици је приказана зависност модула Modul A од апстрактног модула Modul Ba. Из модула Modul Ba су изведени конкретни модули Modul B и Modul B2. На тај начин је реализована инверзија зависности, као што је и објашњено у оквиру принципа инверзије зависности. Са слике се, такође, уочава модул Modul DZ који садржи методу *setModul()*. У оквиру ове методе се врши иницијализација модула Modul B, а затим и иницијализација модула Modul A и повезивање модула Modul A са модулом Modul B. Другим речима, путем модула Modul DZ извршено је додавање зависности између модула Modul A и Modul B.



Слика 66. Принцип додавања зависности

Принцип инверзије зависности и принцип додавања зависности представљају основне градивне елементе на којем се заснивају многе

објектно-оријентисане технологије и оквири (енг. frameworks) који се користе у процесу развоја софтвера. Штавише, њиховом применом се промовише принцип "програмирања ка интерфејсима а не ка имплементацији" што представља један од најважнијих принципа објектно-оријентисаног пројектовања који је дефинисан у оквиру књиге "Design patterns: elements of reusable object-oriented software" [Gamma94]. Другим речима, инжењери су у процесу објектно-оријентисаног развоја софтвера усмерени да пројектују софтверске системе чији су модули слабо повезани, тј. да пројектују модуле који зависе од апстракције а не од имплементације (уместо директне зависности између модула промовише се приступ посредног успостављања зависности између модула, путем апстракције).

На основу изложеног се може закључити да се принцип додавања зависности може повезати са истим софтверским метрикама као и принцип инверзије зависности, што је и приказано у наредној табели (Табела 53).

Табела 53. Веза принципа објектно-оријентисаног пројектовања софтвера Принцип додавања зависности са софтверским метрикама

Принцип објектно-оријентисаног пројектовања софтвера	**Веза са софтверским метрикама**
Принцип додавања зависности	- Повезаност објеката - Дубина стабла наслеђивања - Број подкласа - Метрика стабилности софтвера

6.2.6. Принцип издвајања интерфејса

Као што је напоменуто у другом поглављу, интерфејсом се даје спецификација функција посматраног модула. У процесу развоја софтвера често се спецификација одваја од имплементације. У том смислу се детаљима софтверског система не приступа директно већ посредно, путем

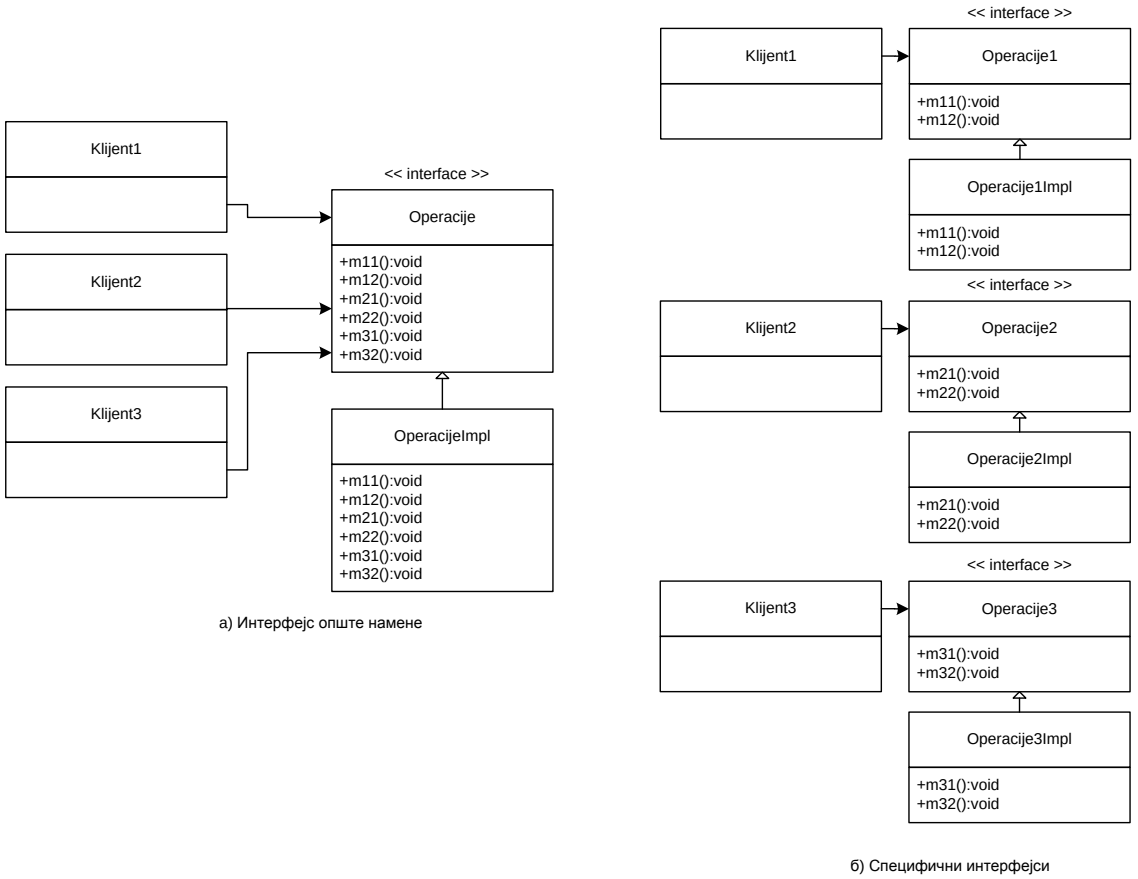
интерфејса. На тај начин детаљи имплементације софтверског система остају сакривени од корисника софтверског система (кориснику се коришћењем интерфејса излаже само спецификација функција софтверског система, при чему корисник не зна како су посматране функције имплементиране). Због тога је веома важно да се на правилан начин изврши спецификација функција модула посматраног софтверског система у смислу дефинисања самих интерфејса. Стога клијенти не требају бити зависни од интерфејса које не користе [Martin96]. Другим речима, боље је да у софтверском систему постоји више специфичних интерфејса који се користе од стране клијената него један интерфејс опште намене [Vlajic14].

Размотримо пример који приказан на наредној слици (Слика 67). Уочава се интерфејс Operacije који дефинише шест метода. Претпоставимо да се методе користе за реализацију одређених функционалности:

- Методама *m11()* и *m12()* се реализује функционалност Ф1.
- Методама *m21()* и *m22()* се реализује функционалност Ф2.
- Методама *m31()* и *m32()* се реализује функционалност Ф3.

При томе, претпоставимо да имамо три клијента:

- Клијент Klijent1 жели да приступи функционалности Ф1,
- Клијент Klijent2 жели да приступи функционалности Ф2,
- Клијент Klijent3 жели да приступи функционалности Ф3.



Слика 67. Принцип издвајања интерфејса

Уколико би постојао само један интерфејс опште намене он би садржао све функционалности (у примеру Ф1, Ф2 и Ф3) које можда неће бити коришћене од стране клијената (или ће бити коришћене од стране само неких клијената) - на тај начин се клијенту излажу функције које можда није потребно да види. С друге стране, у процесу објектно-оријентисаног развоја софтвера треба тежити да свака класа садржи само једну одговорност, тј. не би требао да постоји више од једног разлога за измену посматране класе (објашњено је кроз принцип једне одговорности). Такође, у процесу објектно-оријентисаног развоја софтвера класа треба да буде високо кохезивна и слабо повезана (објашњено је кроз принцип кохезија и повезаност), тј. потребно је да све чланице (атрибути и методе) посматране класе буду у функцији реализације саме класе, при чему ће се повезаност са другим класама свести на најмању могућу меру (најбоље би било да се

повезаност класа остварује на нивоу интерфејса) [Martin96][Vlajic14]. Уколико би постојао само један интерфејс опште намене повећава се могућност нарушавања принципа једне одговорности, принципа високе кохезије и принципа слабе повезаности (класа може имати више одговорности, односно може имати ниску кохезију и високу повезаност, што нису пожељне карактеристике објектно-оријентисаног развоја софтвера) [Martin96]. На основу тога може се рећи да се овај принцип може повезати са софтверским метрикама *Недостатак кохезивности метода у класи* и *Повезаност објеката*.

У том смислу могуће је извршити поделу посматраног интерфејса на интерфејсе Операције1 (садржи методе *m11* и *m12*), Операције2 (садржи методе *m21* и *m22*) и Операције3 (садржи методе *m31* и *m32*). Другим речима, интерфејсом Операције1 дефинишу се методе које су неопходне за реализацију функционалности Ф1, интерфејсом Операције2 дефинишу се методе које су неопходне за реализацију функционалности Ф2, интерфејсом Операције3 дефинишу се методе које су неопходне за реализацију функционалности Ф3. У том смислу је извршено одвајање интерфејса и сваки код њих има одговорност која је усмерена на посматрану функционалност.

Поред наведеног, одвајање спецификације од имплементације омогућава да се у процесу објектно-оријентисаног развоја софтвера направе различите имплементације за посматрану спецификацију (тј. интерфејс). Уколико би постојао само један интерфејс опште намене израда имплементације би захтевала реализацију свих функционалности које су дефинисане интерфејсом што може бити веома захтеван задатак који ће захтевати додатне ресурсе (нпр. време потребно за развој, време потребно за тестирање, инжењере који учествују у процесу развоја, новац итд.). Због тога би било добро да се интерфејс опште намене подели на више специфичних интерфејса који ће се излагати корисницима посматраног софтверског система. У том случају се приликом измене или додавања нове

функционалности у софтверском систему мења само интерфејс клијента који треба да садржи посматрану функционалност док се интерфејси осталих клијената не мењају [Vlajic14]. На овај начин се вишеструко олакшава надоградња и одржавање посматраног софтверског система (измене се врше само на једном месту док остатак софтверског система функционише на потпуно исти начин, без потребе за изменама). У том смислу се локализује место у објектно-оријентисаном софтверском систему на којем је потребно извршити измене. Због тога се овај принцип може повезати и са софтверским метрикама *Дубина стабла наслеђивања* и *Број подкласа*.

У наредној табели (Табела 54) дат је упоредни приказ вредности посматраних метрика. Примећује се да се вредност метрике Недостатак кохезивности метода у класи смањује, док вредности метрика Повезаност објеката, Дубина стабла наслеђивања и Број подкласа имају исте вредности без обзира да ли се користи интерфејс опште намене или специфични интерфејси.

Табела 54. Упоредни приказ вредности метрика за Принцип издвајања интерфејса

Софтверска метрика	**Интерфејс опште намене**		**Специфични интерфејси**	
	Operacije	**OperacijeImpl**	**Operacije1, Operacije2, Operacije3**	**Operacije1Impl, Operacije2Impl, Operacije3Impl**
Недостатак кохезивности метода у класи	0	1	0	0
Повезаност објеката	1	1	1	1
Дубина стабла наслеђивања	1	2	1	2
Број подкласа	1	0	1	0

На основу изложеног се може закључити да се принцип издвајања интерфејса може повезати са софтверским метрикама, што је и приказано у наредној табели (Табела 55).

Табела 55. Веза принципа објектно-оријентисаног пројектовања софтвера Принцип издвајања интерфејса са софтверским метрикама

Принцип објектно-оријентисаног пројектовања софтвера	**Веза са софтверским метрикама**
Принцип издвајања интерфејса	- Недостатак кохезивности метода у класи - Повезаност објеката - Дубина стабла наслеђивања - Број подкласа

У овом одељку представљени су принципи објектно-оријентисаног пројектовања софтвера. На основу датих објашњења се може закључити да се принципи објектно-оријентисаног пројектовања софтвера могу повезати са софтверским метрикама. Наредна табела (Табела 56) даје сумарни приказ принципа објектно-оријентисаног пројектовања софтвера и њихових веза са софтверским метрикама.

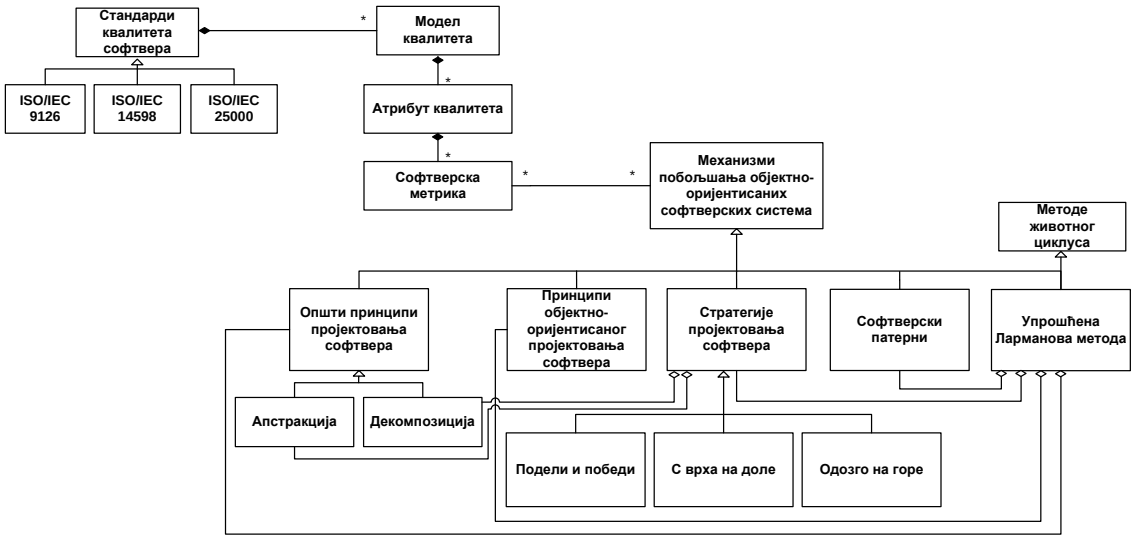
Табела 56. Сумарни приказ објектно-оријентисаних принципа пројектовања софтвера и њихових веза са софтверским метрикама

Принцип објектно-оријентисаног пројектовања софтвера	**Веза са софтверским метрикама**
Принцип једне одговорности	- Недостатак кохезивности метода у класи - Повезаност објеката - Број одговора класе
Принцип отворено-затворено	- Дубина стабла наслеђивања - Број подкласа

Принцип објектно-оријентисаног пројектовања софтвера	**Веза са софтверским метрикама**
	- Сложеност пондерисаних метода - Број пондерисаних метода - Број наредби у методи - Циклична сложеност
Лисков принцип замене	- Дубина стабла наслеђивања - Број подкласа - Повезаност објеката - Број одговора класе - Недостатак кохезивности метода у класи
Принцип инверзије зависности	- Повезаност објеката - Дубина стабла наслеђивања - Број подкласа - Метрика стабилности софтвера
Принцип додавања зависности	- Повезаност објеката - Дубина стабла наслеђивања - Број подкласа - Метрика стабилности софтвера
Принцип издвајања интерфејса	- Недостатак кохезивности метода у класи - Повезаност објеката - Дубина стабла наслеђивања - Број подкласа

6.3. Стратегије пројектовања софтвера

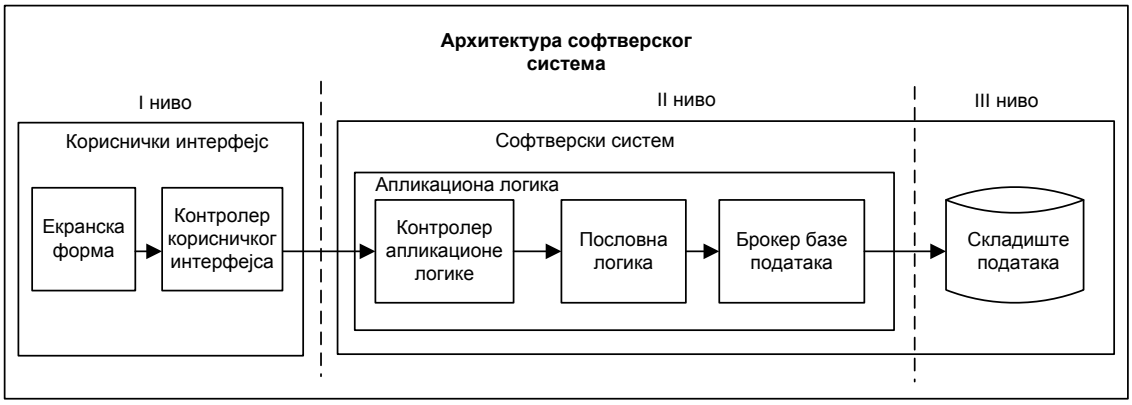
Стратегијама пројектовања софтвера се дефинише стратешки приступ процесу пројектовања софтвера. У том смислу разликују се стратегије *подели и победи*, *с врха на доле* и *одозго на горе* [Vlajic14]. На наредној слици (Слика 68) приказане су стратегије пројектовања софтвера.



Слика 68. Стратегије пројектовања софтвера

6.3.1. Подели и победи

Стратегија *Подели и победи* је заснована на принципу декомпозиције, тј. на принципу поделе почетног проблема на више потпроблема [Vlajic14]. При томе, као резултат процеса декомпозиције настаје модуларизација проблема. Ова стратегија промовише дељење почетног проблема на мање проблеме који се могу независно решавати. Потпреблеми се, такође, могу даље декомпоновати, док се не дође до елементарних потпроблема који се на једноставан начин могу решити. На пример, тронивојска архитектура се може декомпоновати на ниво корисничког интерфејса, ниво апликационе логике и ниво складишта података, при чему се сваки ниво може даље декомпоновати на саставне елементе, што је и приказано на наредној слици (Слика 69). Решавањем мањих проблема се заправо решава почетни проблем - они су у функцији решавања почетног проблема.



Слика 69. Примена стратегије Подели и победи

Стратегија Подели и победи заснована је на два општа принципа пројектовања софтвера: апстракцији и декомпозицији. Сходно томе, ова стратегија је повезана са софтверским метрикама са којима су повезани наведени принципи: Сложеност пондерисаних метода, Број пондерисаних метода класе, Циклична сложеност, Недостатак кохезивности метода у класи, Повезаност објеката, Број одговора класе, Број наредби у методи, Дубина стабла наслеђивања, Број подкласа и Метрика стабилности софтвера.

6.3.2. С врха на доле

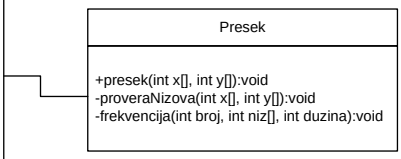
Стратегија *С врха на доле* (енг. top-down) је усмерена на декомпозицију функција [Vlajic14]. Према тој стратегији почетна функција се декомпонује на више подфункција које се могу независно решавати. На крају се све функције интегришу у једну целину чиме се заправо реализује почетна функција. У том смислу се може рећи да су декомпоноване подфункције усмерене на реализацију почетне функције (нпр. уколико разматрамо неку сложену методу која је део софтверског система, може се извршити њена декомпозиција на већи број мањих метода које се такође могу декомпоновати и које су у функцији реализације почетне методе). Стратегија *С врха на доле* је непосредно повезана са стратегијом *Подели и победи* - она дефинише начин на који треба поделити почетни проблем

(почетни проблем се на почетку декомпонује на потпроблеме који се могу независно решавати и у функцији су реализације почетног проблема).

Размотримо пример који је приказан на наредној слици (Слика 70). Са слике се уочава функција *presek()* која приказује пресек два низа целих бројева. Она је имплементирана на следећи начин:

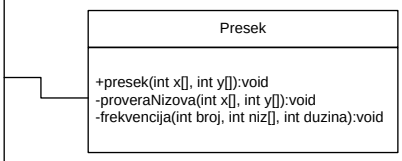
1. Најпре се даје имплементација почетне функције *presek()*. Приликом израде функције се уочава да је потребно извршити проверу да ли су низови празни. Такође, потребно је израчунати фреквенцију појављивања неког елемента у низу. Због тога се почетна функција *presek()* декомпонује на функције *proveraNizova()* и *frekvencija()*, што је и приказано на слици (Слика 70а).
2. Након тога се подфункције разрађују. У том смислу врши се имплементација функције *proveraNizova()* која проверава да ли су низови празни, односно имплементација функције *frekvencija()* која рачуна број појављивања неког елемента у низу. Другим речима, почетна функција декомпонована је на подфункције које извршавају свој посао. На крају су посматране функције интегрисане у функцију *presek()* чиме се реализује приказ пресека два низа целих бројева, што је и приказано на слици (Слика 70б).

```
public void presek(int x[], int y[]) {
  proveraNizova(x, y);
  for (int i = 0; i < x.length; i++) {
    if (frekvencija(x[i], y, y.length) > 0 && frekvencija(x[i], x, i) == 0) {
      System.out.println("Element " + x[i] + " je deo preseka nizova x i y.");
    }
  }
}
void proveraNizova(int x[], int y[]) {
  //TODO Potrebno je implementirati funkciju
}
int frekvencija(int broj, int niz[], int duzina) {
  //TODO Potrebno je implementirati funkciju
  return 0;
}
```



а) Стратегија С врха на доле – почетна функција presek() се декомпонује на две подфункције

```
public void presek(int x[], int y[]) {
  proveraNizova(x, y);
  for (int i = 0; i < x.length; i++) {
    if (frekvencija(x[i], y, y.length) > 0 && frekvencija(x[i], x, i) == 0) {
      System.out.println("Element " + x[i] + " je deo preseka nizova x i y.");
    }
  }
}
void proveraNizova(int x[], int y[]) {
  if (x.length == 0 || y.length == 0) {
    throw new RuntimeException("Niz je prazan!");
  }
}
int frekvencija(int broj, int niz[], int duzina) {
  int f = 0;
  for (int i = 0; i < duzina; i++) {
    if (niz[i] == broj) {
      f++;
    }
  }
  return f;
}
```



б) Стратегија С врха на доле – имплементација и интеграција подфункција са почетном функцијом presek()

Слика 70. Примена стратегије С врха на доле

Уколико се не би применила стратегија *С врха на доле* постојала би само једна метода која би садржала комплетан програмски код. У том смислу би посматрана метода била сложена за одржавање и имала би високе вредности софтверских метрика *Циклична сложеност*, *Сложеност пондерисаних метода*, *Број пондерисаних метода класе* и *Број наредби у методи*. С друге стране, применом стратегије *С врха на доле* вредности наведених метрика се смањују. Такође, применом стратегије *С врха на доле* остварује се висока кохезија међу функцијама (у конкретном примеру ће вредност софтверске метрике *Недостатак кохезивности метода у класи* бити нула).

Стратегија С врха на доле заснована је на два општа принципа пројектовања софтвера: апстракцији и декомпозицији. Сходно томе, ова стратегија је повезана са софтверским метрикама са којима су повезани наведени принципи: Сложеност пондерисаних метода, Број пондерисаних метода класе, Циклична сложеност, Недостатак кохезивности метода у класи, Повезаност објеката, Број одговора класе, Број наредби у методи, Дубина стабла наслеђивања, Број подкласа и Метрика стабилности софтвера.

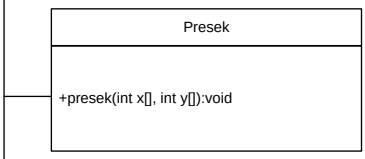
6.3.3. Одоздо на горе

Стратегија *Одоздо на горе* (енг. bottom-up) је заснована на принципу генерализације [Vlajic14]. У том смислу ова стратегија промовише израду почетне функције а затим, уколико је посматрана функција сложена, уочавање једне или више логичких целина које се проглашавају за функције [Vlajic14]. Стратегија *Одоздо на горе* је непосредно повезана са стратегијом *Подели и победи* - она дефинише начин на који треба поделити почетни проблем (почетни проблем се на почетку решава а затим се примењује принцип генерализације којим се одређене логичке целине решења проблема проглашавају за функције; у том смислу се може рећи да су оне такође у функцији реализације почетног проблема).

Размотримо пример који је приказан на наредној слици (Слика 71а). Са слике се уочава функција *presek()* која приказује пресек два низа целих бројева. У функцији се најпре проверава да ли су низови празни, а затим се за сваки елемент првог низа проверава да ли је већ обрађен. Уколико посматрани елемент није обрађен, проверава се постојање тог елемента у другом низу, док се у супротном поступак понавља за остале елементе низа првог низа. У оквиру посматраног примера постоји само једна метода која садржи комплетан програмски код. У том смислу је метода сложена за одржавање и има високе вредности софтверских метрика *Циклична сложеност*, *Сложеност пондерисаних метода*, *Број пондерисаних метода класе* и *Број наредби у методи*.

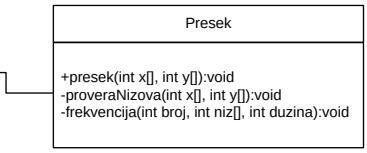
С друге стране, када је уочено да је почетна функција сложена, могуће је уочити више логичких целина и прогласити их за функције. У том смислу су уочене функције *frekvencija()* и *proveraNizova()*, као што је и приказано на слици (Слика 71б). Наведене функције се затим позивају из почетне функције. Другим речима, применом стратегије *Одоздо на горе* вредности софтверских метрика *Циклична сложеност*, *Сложеност пондерисаних метода*, *Број пондерисаних метода класе* и *Број наредби у методи* се смањују. Такође, применом стратегије *Одоздо на горе* остварује се висока кохезија међу функцијама (у конкретном примеру ће вредност софтверске метрике *Недостатак кохезивности метода у класи* бити нула).

```java
public void presek(int x[], int y[]) {
  if (x.length == 0 || y.length == 0) {
    throw new RuntimeException("Niz je prazan!");
  }
  for (int i = 0; i < x.length; i++) {
    int obradjen = 0;
    for (int j = 0; j < i; j++) {
      if (x[i] == x[j]) {
        obradjen = 1;
        break;
      }
    }
    if (obradjen == 0) {
      for (int j = 0; j < y.length; j++) {
        if (x[i] == y[j]) {
          System.out.println("Element " + x[i] + " je deo preseka nizova x i y.");
          break;
        }
      }
    }
  }
}
```



а) Није примењена стратегија Одоздо на горе

```java
public void presek(int x[], int y[]) {
  proveraNizova(x, y);
  for (int i = 0; i < x.length; i++) {
    if (frekvencija(x[i], y, y.length) > 0 && frekvencija(x[i], x, i) == 0) {
      System.out.println("Element " + x[i] + " je deo preseka nizova x i y.");
    }
  }
}
int frekvencija(int broj, int niz[], int duzina) {
  int f = 0;
  for (int i = 0; i < duzina; i++) {
    if (niz[i] == broj) {
      f++;
    }
  }
  return f;
}
void proveraNizova(int x[], int y[]) {
  if (x.length == 0 || y.length == 0) {
    throw new RuntimeException("Niz je prazan!");
  }
}
```



б) Примењена је стратегија Одоздо на горе

Слика 71. Примена стратегије Одоздо на горе

Стратегија Одоздо на горе заснована је на два општа принципа пројектовања софтвера: апстракцији и декомпозицији. Сходно томе, ова стратегија је повезана са софтверским метрикама са којима су повезани наведени принципи: Сложеност пондерисаних метода, Број пондерисаних метода класе, Циклична сложеност, Недостатак кохезивности метода у класи, Повезаност објеката, Број одговора класе, Број наредби у методи, Дубина стабла наслеђивања, Број подкласа и Метрика стабилности софтвера.

Из изложеног се може закључити да су све три стратегије пројектовања софтвера засноване на два општа принципа пројектовања софтвера: апстракцији и декомпозицији. Сходно томе изводи се закључак да су све три стратегије пројектовања софтвера повезане са истим софтверским метрикама. У наредној табели (Табела 57) дат је сумарни приказ веза софтверских метрика са стратегијама пројектовања софтвера.

Табела 57. Приказ веза стратегија пројектовања софтвера са софтверским метрикама

Стратегија пројектовања софтвера	**Веза са софтверским метрикама**
Подели и победи, С врха на доле, Одоздо на горе	- Сложеност пондерисаних метода - Број пондерисаних метода класе - Циклична сложеност - Недостатак кохезивности метода у класи - Повезаност објеката - Број одговора класе - Број наредби у методи - Дубина стабла наслеђивања - Број подкласа - Метрика стабилности софтвера

6.4. Патерни пројектовања софтвера

Корени патерна налазе се у радовима Кристофера Александера и односе се на пројектовање архитектуре у грађевинарству [Vlajic14b]. У том смислу, Александер је да следећу дефиницију патерна: "Сваки патерн је троделно правило, које успоставља релацију између неког проблема, његовог решења и њиховог контекста. Патерн је у исто време и ствар, која се дешава у стварности, и правило које говори када и како се креира наведена ствар" [Alexander79]. Другим речима, патерном се разматра проблем, који се може понављати на различите начине, као и решење које се може користити за решавање сличних проблема [Vlajic14b].

Крајем осамдесетих година патерни почињу да се примењују у софтверском инжењерству. У том смислу се издваја дефиниција Ричарда Габријела који је софтверски патерн дефинисао као "троделно правило које успоставља релацију између неког контекста, неког система сила који се појављују у том контексту (проблем) и софтверске конфигурације која омогућава тим силама да успоставе одговарајуће односе (решење)" [Gabriel96]. У том смислу се под контекстом мисли на софтверски систем са својим ограничењима, док се под силама подразумевају елементи тог софтверског система и њихови међусобни односи [Vlajic14b].

Постоје различите класификације патерна [Vlajic14b]:

- Тронивојски патерни - У оквиру ове класификације патерни се посматрају са различитих нивоа апстракције. У том смислу разликују се идиоми (енг. idioms), патерни пројектовања (енг. design patterns) и оквири (енг. frameworks):
    - Идиоми представљају патерне најнижег нивоа апстракције који зависе од специфичне имплементационе технологије, нпр. од програмског језика.
    - Патерни пројектовања представљају патерне средњег нивоа апстракције који су независни од имплементационе

технологије. Патернима пројектовања реализује се микроархитектура неког софтверског система. Патерни пројектовања добијају велику пажњу и афирмацију појавом књиге "Design patterns: elements of reusable object-oriented software" [Gamma94]. У оквиру наведене књиге разматрају се патерни пројектовања софтвера за које се често користи скраћеница GoF[10] патерни пројектовања софтвера.

- Оквири представљају патерне највишег нивоа апстракције и пројектовани су тако да обезбеђују опште решење које конкретне апликације користе и проширују својим специфичностима. Један оквир може да садржи друге патерне различитих нивоа апстракције (идиоме, патерне пројектовања и/или друге оквире).

- Анти-патерни - Представљају патерне који су се у пракси показали као лоши па њихова примена не решава посматрани проблем или делимично решава посматрани проблем.
- Мета-патерни - Представљају генерализацију кључних структура многих GoF патерна пројектовања. Наведене структуре представљају блокове (мета-патерне из којих се могу направити други патерни).

Патерни пројектовања[11] у најопштијем смислу представљају решење неког проблема у неком контексту, при чему се та решења могу користити за решавање других проблема [Gamma94]. У том смислу се може рећи да се патернима пројектовања формирају најбоље праксе које треба применити у

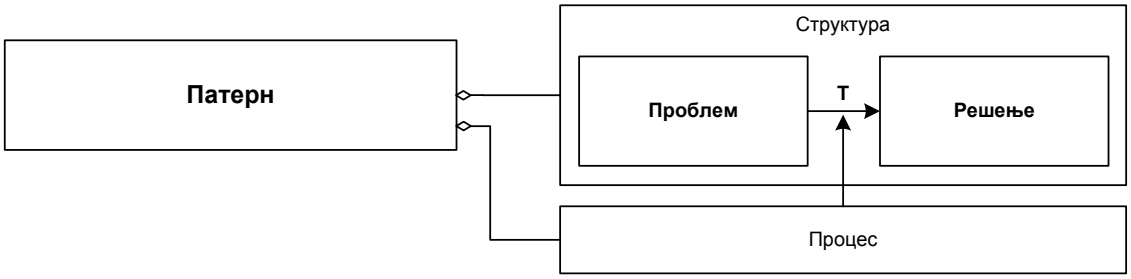
[10] Скраћеница GoF настала је на основу назива "четворочлана банда" (енг. Gang of Four). Под тим именом су познати аутори који су написали књигу "Design patterns: elements of reusable object-oriented software".

[11] Патернима пројектовања реализована је микроархитектура неког софтверског система [Vlajic14b]. Важно је напоменути да постоје и патерни који се користе у другим фазама процеса развоја софтвера (нпр. патерни захтева, патерни анализе и имплементациони патерни - идиоми) [Vlajic14b]. С обзиром на непосредну повезаност патерна пројектовања софтвера са општим принципима пројектовања софтвера, стратегијама пројектовања софтвера и принципима објектно-оријентисаног пројектовања софтвера, у овој докторској дисертацији ће фокус бити усмерен ка патернима пројектовања софтвера.

процесу развоја софтвера. Заједничка карактеристика свих патерна пројектовања је да олакшавају развој нових софтверских система и да помажу у одржавању и надоградњи постојећих софтверских система. Због тога су патерни пројектовања софтвера идентификовани као механизам за побољшање објектно-оријентисаних софтверских система.

6.4.1. Општи облик GoF патерна пројектовања

На основу приказаних дефиниција у претходном одељку може се закључити да патерн има два важна дела: проблем и решење. Такође, с обзиром да се применом патерна даје решење посматраног проблема које се може применити више пута код других, различитих проблема, патерни имају особину поновне употребљивости [Vlajic14b]. С обзиром да патерн представља "ствар и правило за грађење ствари" [Coplien96], може се закључити да је патерн у исто време и структура и процес, што је и приказано на наредној слици (Слика 72).

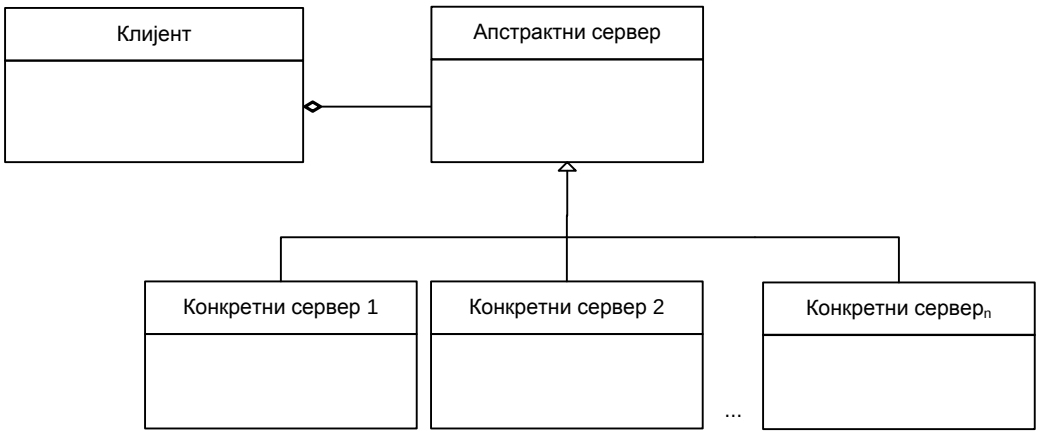


Слика 72. Патерн као структура и процес [Vlajic14b]

Са слике се уочава да патерн садржи структуру проблема и структуру решења. С друге стране, уочава се да је патерн и процес у оквиру кога се трансформацијом (T) креира структура решења из структуре проблема: Структура проблема $\xrightarrow{T}$ Структура решења [Vlajic14b].

У академским истраживањима и инжењерској пракси најцитиранија књига која се бави патернима пројектовања је "Design patterns: elements of reusable object-oriented software" [Gamma94]. У оквиру књиге дефинисана су 23 патерна пројектовања која су подељена у три групе: креациони патерни,

патерни структуре и патерни понашања. За сваки патерн дефинисан је контекст, проблем који се разматра и решење посматраног проблема коришћењем патерна. С друге стране, код 20 патерна може се уочити општа структура решења GoF патерна пројектовања [Vlajic16], што је и приказано на наредној слици (Слика 73).



Слика 73. Општа структура решења GoF патерна пројектовања [Vlajic16]

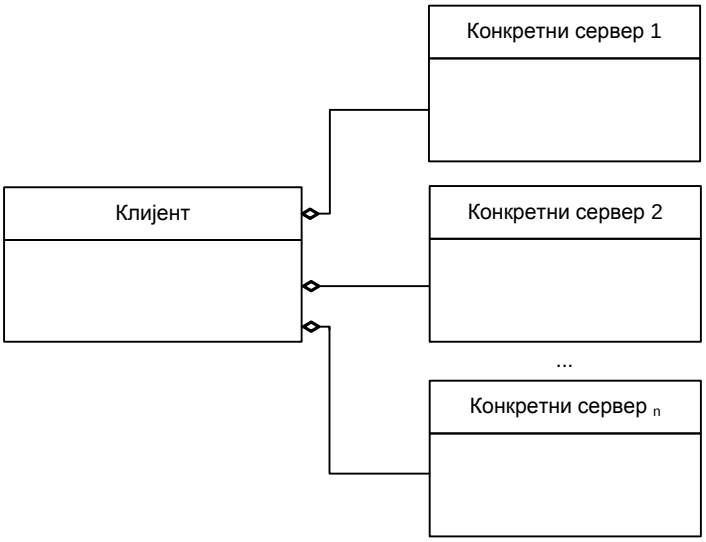
Општа структура решења GoF патерна представља уређену тројку (Клијент, Апстрактни сервер, Конкретни сервер), при чему може постојати n конкретних сервера [Vlajic16]. У том смислу, општа структура решења софтверског патерна се састоји из следећих делова:

- **Клијент** - Клијент користи функционалности које му излаже апстрактни сервер. Клијент је, преко апстрактног сервера, посредно повезан са конкретним серверима. Клијент може бити реализован као конкретна или апстрактна класа.
- **Апстрактни сервер** - Апстрактни сервер излаже одређене апстрактне функционалности. У општој структури решења софтверског патерна постоји један апстрактни сервер и он може имати више конкретних реализација (тј. конкретних сервера). Апстрактни сервер може бити реализован као интерфејс или као конкретна/апстрактна класа.

- **Конкретни сервер** - Конкретни сервер је изведен из апстрактног сервера и представља имплементацију конкретне функционалности. У општој структури решења софтверског патерна може постојати више конкретних сервера, чиме се даје различита реализација посматраних апстрактних функционалности. Конкретни сервер је реализован као конкретна класа.

У контексту објектно-оријентисаног развоја софтвера, из опште структуре решења софтверског патерна може се закључити да се користе концепти наслеђивања и касног повезивања. Између клијента и апстрактног сервера је могуће успоставити везу агрегације или зависности. С друге стране, између апстрактног и конкретног сервера је веза наслеђивања или реализације, док директна веза између клијента и конкретног сервера не постоји (она је индиректна и успоставља се преко апстрактног сервера) [Vlajic14b]. Другим речима, клијент је у време компајлирања програма повезан са апстрактним сервером и нема знања о конкретном серверу. С друге стране, у време извршавања програма клијент се повезује са конкретним сервером [Vlajic14b]. На тај начин се клијент може повезати са различитим реализацијама посматраних функционалности, тј. омогућава се једноставна замена конкретног сервера без потребе за изменом клијента. Тиме се реализује поновно коришћење компоненти, што и представља суштину патерна пројектовања [Vlajic14b].

С друге стране, општа структура проблема GoF патерна пројектовања је уређени пар (Клијент, Конкретни сервер), при чему може постојати n конкретних сервера [Vlajic16]. У том смислу се може дефинисати структура проблема GoF патерна пројектовања која је приказана на наредној слици (Слика 74).



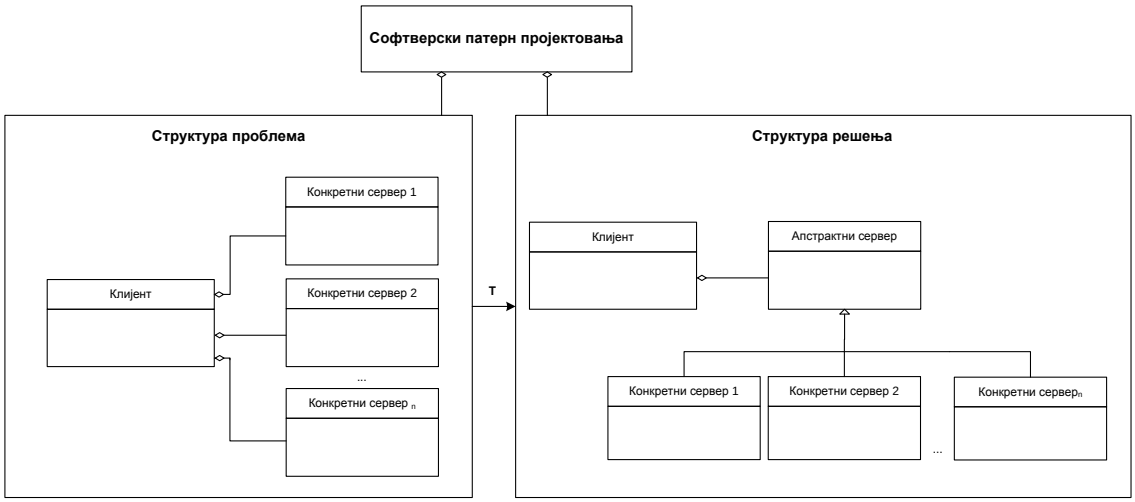
Слика 74. Структура проблема GoF патерна пројектовања [Vlajic16]

Уочава се да се структура проблема софтверског патерна састоји из следећих делова:

- **Клијент** - Клијент користи функционалности које му излаже конкретни сервер. Клијент је директно повезан са конкретним сервером.
- **Конкретни сервер** - Конкретни сервер је представља имплементацију конкретне функционалности. Може постојати више конкретних сервера.

У контексту објектно-оријентисаног развоја софтвера, из структуре проблема се може закључити да је у времену компајлирања програма клијент повезан са конкретним сервером (тј. има знања о конкретном серверу). Другим речима, уколико је потребно додавање или промена конкретног сервера, неопходна је и измена самог клијента, што доводи до отежаног одржавања и надоградње посматраног софтверског система [Vlajic14b].

На основу изложеног може се закључити да се општи облик GoF патерна пројектовања може представити коришћењем структуре проблема и структуре решења, као што је и приказано на наредној слици (Слика 75).

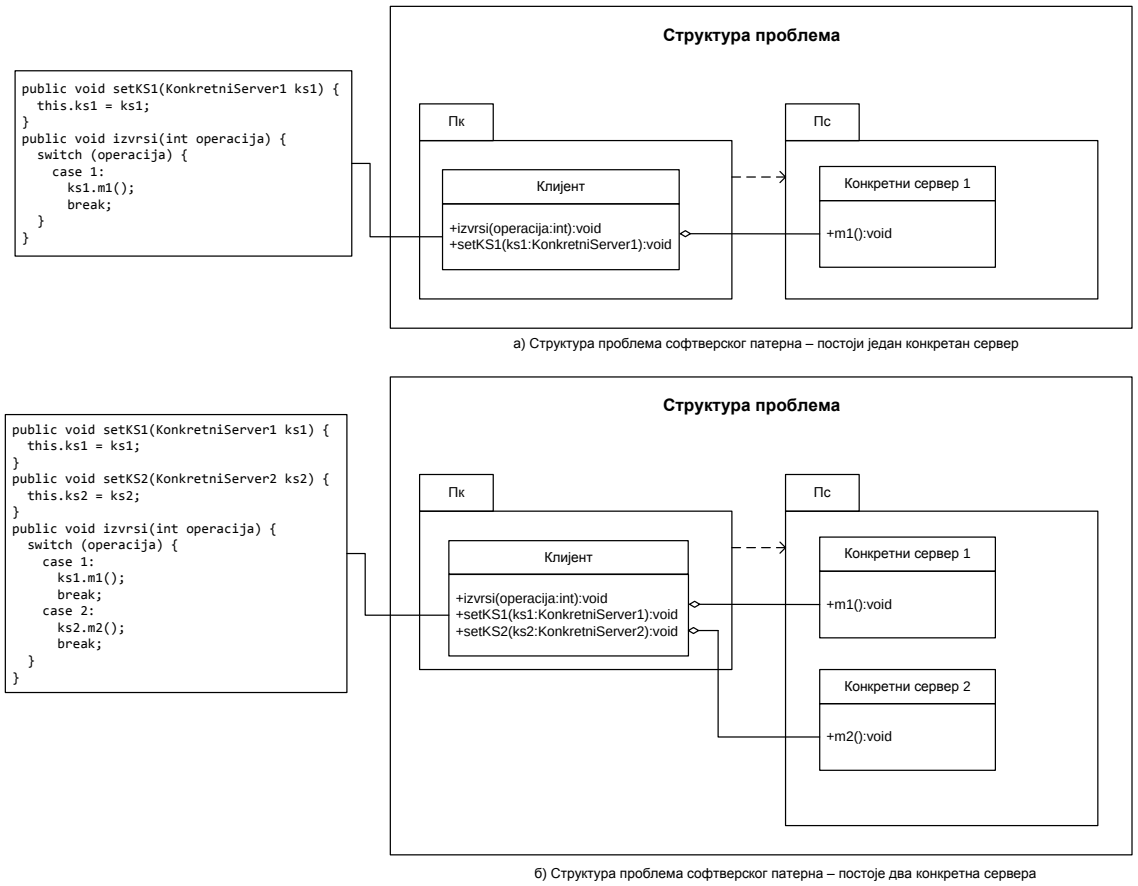


Слика 75. Општи облик GoF патерна пројектовања [Vlajic16]

Другим речима, општи облик GoF патерна пројектовања садржи структуру проблема, структуру решења, и трансформацију структуре проблема у структуру решења [Vlajic16]. У наставку рада разматра се веза општег облика GoF патерна пројектовања са софтверским метрикама.

6.4.2. Веза општег облика GoF патерна пројектовања са софтверским метрикама

Размотримо општи облик структуре проблема софтверског патерна. Нека је дата структура која је приказана на наредној слици (Слика 76). Претпоставимо да се класа Клијент налази у пакету Пк, док се класе које представљају конкретне сервере налазе у пакету Пс.



Слика 76. Софтверски патерн пројектовања - пример структуре проблема

Уколико посматрамо пакет Пк, са слике (Слика 76а) се уочава да постоји веза са једним конкретним сервером. Другим речима, постоји једна одлазна повезаност, а не постоји долазна повезаност. Размотримо стабилност пакета Пк путем побољшане метрике стабилности Ir[12]. Уколико постоји једна одлазна повезаност, а не постоји долазна повезаност (Ce=1, Ca=0), нестабилност посматраног пакета Пк износи: Ir = 5.1 / 9.2 ≈ 0.55. С друге стране, уколико разматрамо пакет Пк, са слике (Слика 76б) се уочава да постоји веза са два конкретна сервера. Другим речима, постоје две одлазне повезаности (Ce=2), а не постоји долазна повезаност (Ca=0). У том случају нестабилност посматраног пакета Пк износи: Ir = 6.1 / 10.2 ≈ 0.60. У наредној

[12] С обзиром да метрика стабилности I врши релативизацију одлазне повезаности када не постоји долазна повезаност, стабилност се разматра коришћењем побољшане верзије метрике стабилности Ir [Vlajic17].

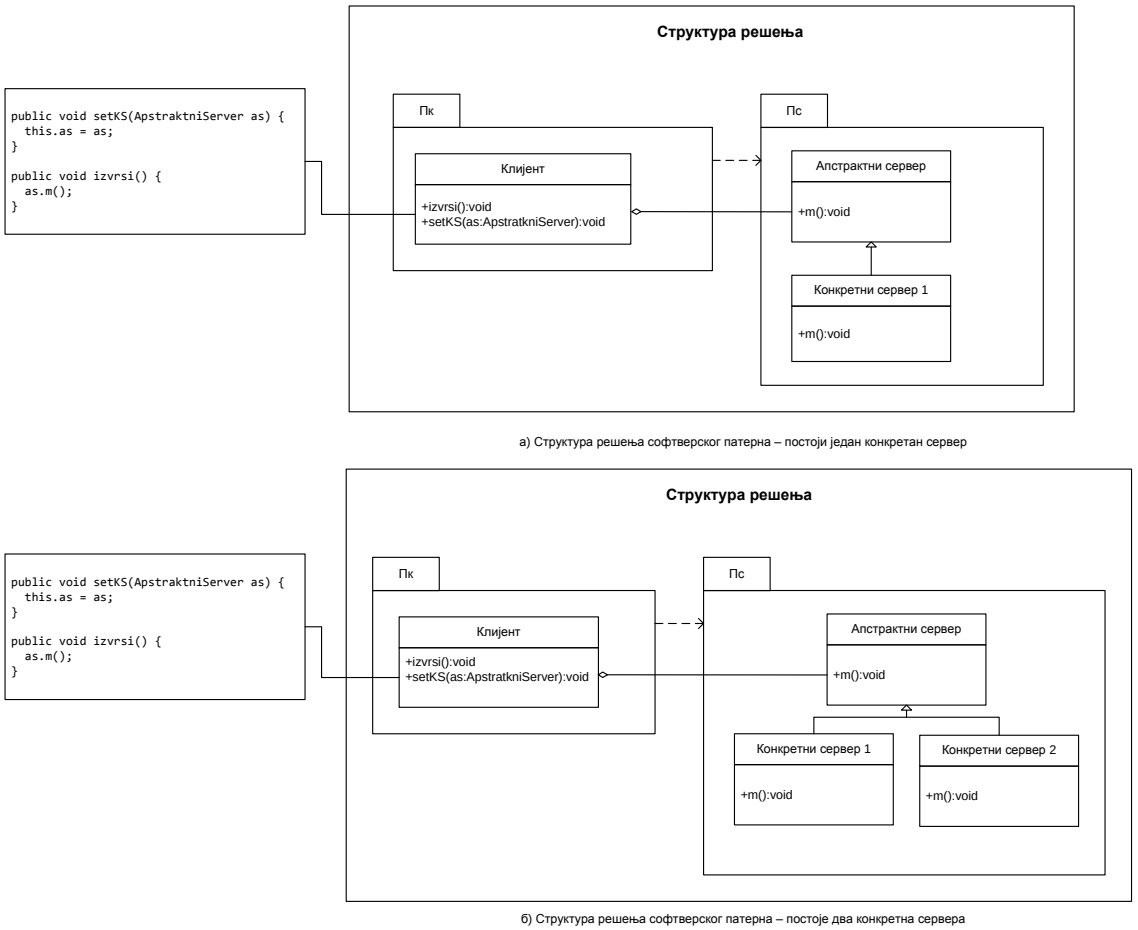
табели (Табела 58) дат је приказ промене вредности побољшане метрике стабилности са повећањем броја конкретних сервера.

Табела 58. Структура проблема софтверског патерна - промена вредности побољшане метрике стабилности са повећањем броја конкретних сервера

Број конкретних сервера	**Са**	**Се**	**Ir пакета Пк**
1	0	1	0.55
2	0	2	0.60
3	0	3	0.63
…	…	…	…
n - 1	0	n - 1	≈ 1
n	0	n	1

На основу приказаног се може закључити да се повећавањем броја конкретних сервера повећава нестабилност посматраног пакета Пк. **У том смислу се може рећи да се у структури проблема софтверског патерна успоставља директна зависност између клијента и конкретних сервера, што последично повећава нестабилност посматраног софтверског система**.

Размотримо општи облик структуре решења софтверског патерна. Нека је дата структура која је приказана на наредној слици (Слика 77). Претпоставимо да се класа Клијент налази у пакету Пк, док се класе Апстрактни сервер и класе које представљају конкретне сервере налазе у пакету Пс.



Слика 77. Софтверски патерн пројектовања - пример структуре решења

Уколико посматрамо пакет Пк, са слике (Слика 77а) се уочава да постоји веза са апстрактним сервером а не постоји веза са Конкретним сервером 1 (веза са Конкретним сервером 1 се остварује посредно, коришћењем апстрактног сервера). Другим речима, постоји једна одлазна повезаност, а не постоји долазна повезаност. Размотримо стабилност пакета Пк путем побољшане метрике стабилности Ir. Уколико постоји једна одлазна повезаност, а не постоји долазна повезаност (Ce=1, Ca=0), нестабилност посматраног пакета Пк износи: Ir = 5.1 / 9.2 ≈ 0.55. С друге стране, уколико разматрамо пакет Пк, са слике Слика 77б се уочава да постоји веза са апстрактним сервером а не постоји веза са Конкретним сервером 1 и Конкретним сервером 2 (веза са Конкретним сервером 1 и Конкретним сервером 2 се остварује посредно, коришћењем апстрактног сервера). Другим речима, постоји једна одлазна повезаности (Ce=1), а не постоји

долазна повезаност (Ca=0). У том случају нестабилност посматраног пакета Пк износи: Ir = 5.1 / 9.2 ≈ 0.55. У наредној табели (Табела 59) дат је приказ промене вредности побољшане метрике стабилности са повећањем броја конкретних сервера.

Табела 59. Структура решења софтверског патерна - промена вредности побољшане метрике стабилности са повећањем броја конкретних сервера

Број конкретних сервера	Ca	Ce	Ir пакета Пк
1	0	1	0.55
2	0	1	0.55
3	0	1	0.55
…	…	…	…
n - 1	0	1	0.55
n	0	1	0.55

На основу приказаног се може закључити да се повећавањем броја конкретних сервера не повећава нестабилност посматраног пакета Пк (она износи 0.55). **У том смислу се може рећи да се у структури решења софтверског патерна путем апстрактног сервера успоставља посредна зависност између клијента и конкретних сервера, што не повећава нестабилност посматраног софтверског система, тј. нестабилност система се не мења додавањем нових конкретних сервера**.

У наредној табели (Табела 60) дат је упоредни приказ вредности метрике стабилности софтвера за структуру проблема и структуру решења.

Табела 60. Упоредни приказ вредности метрике стабилности софтвера за структуру проблема и структуру решења

Број конкретних сервера	Ir (структура проблема)	Ir (структура решења)
1	0.55	0.55
2	0.60	0.55

3	0.63	0.55
…	…	…
n - 1	≈1	0.55
n	1	0.55

На основу изложеног се може закључити да се општи облик софтверског патерна пројектовања може повезати са Метриком стабилности софтвера. Применом општег облика структуре решења софтверског патерна омогућава се једноставно додавање конкретних сервера, што не утиче на клијента, тј. не доводи до нестабилности посматраног решења.

Софтверске патерне пројектовања могуће је повезати и са софтверском метриком Повезаност објеката. Размотримо општи облик структуре проблема софтверског патерна који је приказан на претходној слици (Слика 76). Уколико разматрамо приказане примере, може се закључити да постоји директна повезаност клијента са конкретним серверима. Другим речима, додавањем новог конкретног сервера, повећава се повезаност објеката. У наредној табели (Табела 61) дат је приказ промене вредности софтверске метрике Повезаност објеката са повећањем броја конкретних сервера.

Табела 61. Структура проблема софтверског патерна - промена вредности метрике Повезаност објеката са повећањем броја конкретних сервера

Број конкретних сервера	**Повезаност објеката**
1	1
2	2
3	3
…	…
n - 1	n - 1
n	n

Размотримо општи облик структуре решења софтверског патерна који је приказан на претходној слици (Слика 77). Уколико разматрамо приказане

примере, може се закључити да постоји директна повезаност клијената са апстрактним сервером, док не постоји директна повезаност клијента са конкретним серверима (ова повезаност је посредна и реализује се преко апстрактног сервера). Другим речима, додавањем новог конкретног сервера, не долази до повећања повезаности објеката. У наредној табели (Табела 62) дат је приказ промене вредности софтверске метрике Повезаност објеката са повећањем броја конкретних сервера.

Табела 62. Структура решења софтверског патерна - промена вредности метрике Повезаност објеката са повећањем броја конкретних сервера

Број конкретних сервера	Повезаност објеката
1	1
2	1
3	1
…	…
n - 1	1
n	1

На основу приказаног се може закључити да се повећавањем броја конкретних сервера не повећава вредност софтверске метрике Повезаност објеката. **У том смислу се може рећи да се у структури решења софтверског патерна путем апстрактног сервера успоставља посредна зависност између клијента и конкретних сервера, што не повећава повезаност објеката у посматраном софтверском систему, тј. повезаност објеката се не мења додавањем нових конкретних сервера**.

У наредној табели (Табела 63) дат је упоредни приказ вредности метрике Повезаност објеката за структуру проблема и структуру решења.

Табела 63. Упоредни приказ вредности метрике Повезаност објеката за структуру проблема и структуру решења

Број конкретних сервера	Повезаност објеката (структура проблема)	Повезаност објеката (структура решења)

Број конкретних сервера	Повезаност објеката (структура проблема)	Повезаност објеката (структура решења)
1	1	1
2	2	1
3	3	1
…	…	…
n - 1	n - 1	1
n	n	1

На основу изложеног се може закључити да се општи облик софтверског патерна пројектовања може повезати са софтверском метриком Повезаност објеката. Применом општег облика структуре решења софтверског патерна омогућава се једноставно додавање конкретних сервера, што не утиче на клијента, тј. не доводи до повећања повезаности објеката у посматраном софтверском систему.

Софтверске патерне пројектовања могуће је повезати и са софтверским метрикама Дубина стабла наслеђивања и Број подкласа. Размотримо општи облик структуре проблема софтверског патерна који је приказан на претходној слици (Слика 76). Уколико разматрамо приказане примере, може се закључити да постоји директна повезаност клијента са конкретним серверима. Другим речима, у структури проблема се не примењује концепт наслеђивања, па су вредности софтверских метрика Дубина стабла наслеђивања = 1, Број подкласа = 0. У наредној табели (Табела 64) дат је приказ промене вредности софтверских метрика Дубина стабла наслеђивања и Број подкласа са повећањем броја конкретних сервера.

Табела 64. Структура проблема софтверског патерна - промена вредности метрика Дубина стабла наслеђивања и Број подкласа са повећањем броја конкретних сервера

Број конкретних сервера	Дубина стабла наслеђивања	Број подкласа
1	1	0

Број конкретних сервера	**Дубина стабла наслеђивања**	**Број подкласа**
2	1	0
3	1	0
…	…	…
n - 1	1	0
n	1	0

На основу приказаног се може закључити да повећавањем броја конкретних сервера не долази до повећања вредности метрика Дубина стабла наслеђивања и Број подкласа. **У том смислу се може рећи да се у структури проблема софтверског патерна успоставља директна зависност између клијента и конкретних сервера, што последично не доводи до повећања вредности метрика Дубина стабла наслеђивања и Број подкласа у посматраном софтверском систему**.

Размотримо општи облик структуре решења софтверског патерна који је приказан на претходној слици (Слика 77). Уколико разматрамо приказане примере, може се закључити да постоји повезаност клијената са апстрактним сервером, док не постоји директна повезаност клијента са конкретним серверима (ова повезаност је посредна и реализује се преко апстрактног сервера). Примећује се да се у структури решења користи концепт наслеђивања и да су конкретни сервери изведени из апстрактног сервера. Другим речима, у структури решења се примењује концепт наслеђивања, па су вредности софтверских метрика Дубина стабла наслеђивања = 2, Број подкласа > 0. У наредној табели (Табела 65) дат је приказ промене вредности софтверских метрика Дубина стабла наслеђивања и Број подкласа са повећањем броја конкретних сервера.

Табела 65. Структура решења софтверског патерна - промена вредности метрика Дубина стабла наслеђивања и Број подкласа са повећањем броја конкретних сервера

Број конкретних сервера	Дубина стабла наслеђивања	Број подкласа
1	2	1
2	2	2
3	2	3
…	…	…
n - 1	2	n - 1
n	2	n

На основу приказаног се може закључити да додавањем апстрактног сервера и повећавањем броја конкретних сервера долази до повећања вредности метрика Дубина стабла наслеђивања и Број подкласа. **У том смислу се може рећи да се у структури решења софтверског патерна успоставља посредна зависност између клијента и конкретних сервера, што последично доводи до повећања вредности метрика Дубина стабла наслеђивања и Број подкласа у посматраном софтверском систему**.

У наредној табели (Табела 66) дат је упоредни приказ вредности метрика Дубина стабла наслеђивања и Број подкласа за структуру проблема и структуру решења.

Табела 66. Упоредни приказ вредности метрика Дубина стабла наслеђивања и Број подкласа за структуру проблема и структуру решења

Број конкретних сервера	Дубина стабла наслеђивања		Број подкласа	
	Структура проблема	Структура решења	Структура проблема	Структура решења
1	1	2	0	1
2	1	2	0	2
3	1	2	0	3
…	…	…	…	…

Број конкретних сервера	**Дубина стабла наслеђивања**		**Број подкласа**	
	Структура проблема	**Структура решења**	**Структура проблема**	**Структура решења**
n - 1	1	2	0	n - 1
n	1	2	0	n

На основу изложеног се може закључити да се општи облик софтверског патерна пројектовања може повезати са софтверским метрикама Дубина стабла наслеђивања и Број подкласа. Применом општег облика структуре решења софтверског патерна омогућава се једноставно додавање конкретних сервера, што не утиче на клијента, али доводи до повећања дубине стабла наслеђивања и броја подкласа у посматраном софтверском систему.

Софтверске патерне пројектовања могуће је повезати и са софтверским метрикама Циклична сложеност, Сложеност пондерисаних метода, Број пондерисаних метода, Број наредби у методи и Број одговора класе. Размотримо општи облик структуре проблема софтверског патерна који је приказан на претходној слици (Слика 76). Уколико разматрамо приказане примере, може се закључити да постоји директна повезаност клијента са конкретним серверима. У том смислу, у структури проблема се за сваки конкретни сервер дефинише метода за постављање конкретног сервера, као и метода *izvrsi()* која је задужена за позив метода које дефинишу конкретни сервери. У наредној табели (Табела 67) дат је приказ промене вредности софтверских метрика Циклична сложеност, Сложеност пондерисаних метода, Број пондерисаних метода, Број наредби у методи и Број одговора класе са повећањем броја конкретних сервера.

Табела 67. Структура проблема софтверског патерна - промена вредности посматраних метрика са повећањем броја конкретних сервера

Број конкретних сервера	Циклична сложеност	Сложеност пондерисаних метода	Број пондерисаних метода	Број наредби у методи	Број одговора класе
1	3	4	2	3	3
2	4	6	3	5	5
3	5	8	4	7	7
…	…	…	…	…	…
n - 1	n + 1	2n + 1	n	2n	2n
n	n + 2	2n + 2	n + 1	2n + 1	2n + 1

На основу приказаног се може закључити да повећавањем броја конкретних сервера долази до повећања вредности метрика Циклична сложеност, Сложеност пондерисаних метода, Број пондерисаних метода, Број наредби у методи и Број одговора класе. **У том смислу се може рећи да се у структури проблема софтверског патерна успоставља директна зависност између клијента и конкретних сервера, што последично доводи до повећања вредности метрика Циклична сложеност, Сложеност пондерисаних метода, Број пондерисаних метода, Број наредби у методи и Број одговора класе у посматраном софтверском систему**.

Размотримо општи облик структуре решења софтверског патерна који је приказан на претходној слици (Слика 77). Уколико разматрамо приказане примере, може се закључити да постоји повезаност клијената са апстрактним сервером, док не постоји директна повезаност клијента са конкретним серверима (ова повезаност је посредна и реализује се преко апстрактног сервера). Примећује се да се у структури решења користи концепт наслеђивања и да су конкретни сервери изведени из апстрактног сервера. У том смислу, у структури решења се дефинише само једна метода за постављање конкретног сервера, као и метода *izvrsi()* која је задужена за

позив метода које дефинишу конкретни сервери. У наредној табели (Табела 68) дат је приказ промене вредности софтверских метрика Циклична сложеност, Сложеност пондерисаних метода, Број пондерисаних метода, Број наредби у методи и Број одговора класе са повећањем броја конкретних сервера.

Табела 68. Структура решења софтверског патерна - промена вредности посматраних метрика са повећањем броја конкретних сервера

Број конкретних сервера	Циклична сложеност	Сложеност пондерисаних метода	Број пондерисаних метода	Број наредби у методи	Број одговора класе
1	1	2	2	1	3
2	1	2	2	1	3
3	1	2	2	1	3
…	…	…	…	…	…
n - 1	1	2	2	1	3
n	1	2	2	1	3

На основу приказаног се може закључити да додавањем апстрактног сервера и повећавањем броја конкретних сервера не долази до повећања вредности метрика Циклична сложеност, Сложеност пондерисаних метода, Број пондерисаних метода, Број наредби у методи и Број одговора класе. **У том смислу се може рећи да се у структури решења софтверског патерна успоставља посредна зависност између клијента и конкретних сервера, што последично не доводи до повећања вредности метрика Циклична сложеност, Сложеност пондерисаних метода, Број пондерисаних метода, Број наредби у методи и Број одговора класе у посматраном софтверском систему**.

У наредној табели (Табела 69) дат је упоредни приказ вредности метрика Циклична сложеност (за посматрану метрику се, ради прегледности, користи ознака M1), Сложеност пондерисаних метода (користи се ознака

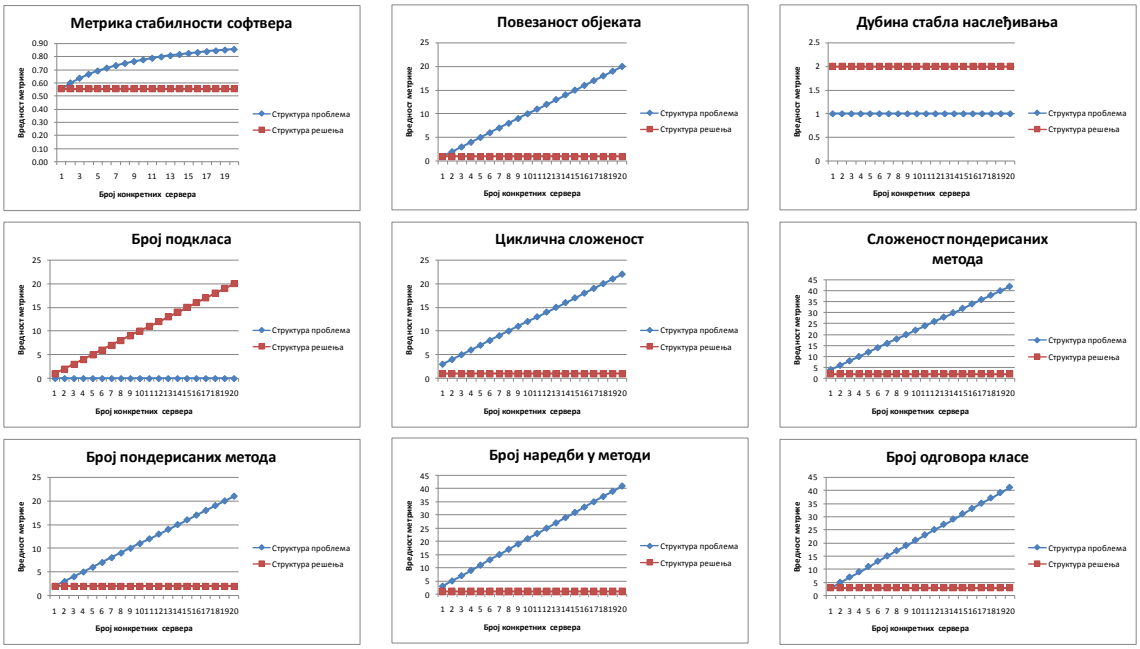
M2), Број пондерисаних метода (користи се ознака M3), Број наредби у методи (користи се ознака M4) и Број одговора класе (користи се ознака M5) за структуру проблема и структуру решења.

Табела 69. Упоредни приказ вредности промене посматраних метрика за структуру проблема и структуру решења

Број конкретних сервера	Структура проблема					Структура решења				
	M1	M2	M3	M4	M5	M1	M2	M3	M4	M5
1	3	4	2	3	3	1	2	2	1	3
2	4	6	3	5	5	1	2	2	1	3
3	5	8	4	7	7	1	2	2	1	3
...	...	...	...	...	...	...	...	...	...	...
n - 1	n+1	2n+1	n	2n	2n	1	2	2	1	3
n	n+2	2n+2	n+1	2n+1	2n+1	1	2	2	1	3

На основу изложеног се може закључити да се општи облик софтверског патерна пројектовања може повезати са софтверским метрикама Циклична сложеност, Сложеност пондерисаних метода, Број пондерисаних метода, Број наредби у методи и Број одговора класе. Применом општег облика структуре решења софтверског патерна омогућава се једноставно додавање конкретних сервера, што не утиче на клијента, тј. не доводи до повећања вредности посматраних софтверских метрика.

Уочене везе општег облика патерна пројектовања софтвера могу се представити и графички, што је и приказано на наредној слици (Слика 78). Ради једноставности број конкретних сервера ограничен је на 20. За сваку софтверску метрику даје се вредност структуре проблема и структуре решења.



Слика 78. Графички приказ веза општег облика патерна пројектовања софтвера са софтверским метрикама

Уколико у општем облику патерна пројектовања софтвера посматрамо структуру проблема софтверска метрика може имати три појавна облика:

- нелинеарна функција,
- линеарна функција и
- константа.

С друге стране, уколико у општем облику патерна пројектовања софтвера посматрамо структуру решења софтверска метрика може имати два појавна облика:

- линеарна функција и
- константа.

У наредној табели (Табела 70) дат је приказ појавног облика сваке софтверске метрике за структуру проблема и структуру решења.

Табела 70. Приказ појавног облика софтверских метрика у структури проблема и структури решења

Софтверска метрика	Патерн пројектовања софтвера

	Структура проблема	**Структура решења**
Метрика стабилности софтвера	Нелинеарна функција	Константа
Повезаност објеката	Линеарна функција	Константа
Дубина стабла наслеђивања	Константа	Константа
Број подкласа	Константа	Линеарна функција
Циклична сложеност	Линеарна функција	Константа
Сложеност пондерисаних метода	Линеарна функција	Константа
Број пондерисаних метода	Линеарна функција	Константа
Број наредби у методи	Линеарна функција	Константа
Број одговора класе	Линеарна функција	Константа

Из изложеног се може закључити следеће:

- Метрика стабилности софтвера у структури проблема има вредност нелинеарне функције. С друге стране, у структури решења посматрана метрика има вредност константе. Боља вредност посматране метрике је остварена у структури решења.
- Метрике *Повезаност објеката*, *Циклична сложеност*, *Сложеност пондерисаних метода*, *Број пондерисаних метода*, *Број наредби у методи* и *Број одговора класе* у структури проблема имају вредности линеарних функција. С друге стране, у структури решења посматране метрике имају вредности константе. Боље вредности посматраних метрика су остварене у структури решења.
- Метрика *Дубина стабла наслеђивања* има константну вредност у структури проблема и структури решења. Боља вредност посматране метрике је остварена у структури проблема.
- Метрика *Број подкласа* у структури проблема има константну вредност. С друге стране, у структури решења посматрана метрика има вредност линеарне функције. Боља вредност посматране метрике је остварена у структури проблема.

Узимајући у обзир претходно описано, може се закључити да се софтверски патерни пројектовања могу повезати са софтверским метрикама. У наредној табели (Табела 71) дат је приказ веза софтверских патерна пројектовања са софтверским метрикама.

Табела 71. Веза софтверских патерна пројектовања са софтверским метрикама

Софтверски патерн пројектовања	**Софтверске метрике**
Општи облик софтверског патерна пројектовања	- Метрика стабилности софтвера - Повезаност објеката - Дубина стабла наслеђивања - Број подкласа - Циклична сложеност - Сложеност пондерисаних метода - Број пондерисаних метода - Број наредби у методи - Број одговора класе

С обзиром на њихову објектно-оријентисану усмереност и непосредну повезаност са софтверским метрикама, можемо рећи да софтверски патерни пројектовања представљају важан механизам побољшања објектно-оријентисаних софтверских система. Коришћењем софтверских патерна омогућава се развој софтверских система који су једноставни за одржавање и надоградњу.

6.5. Методе развоја софтвера

Методом развоја софтвера описује се процес развоја софтвера. На тај начин примењују се најбоље праксе у процесу развоја софтвера. Методе развоја софтвера могу да користе претходно описане опште принципе пројектовања софтвера, принципе објектно-оријентисаног пројектовања софтвера, стратегије пројектовања софтвера и патерне пројектовања софтвера. Због тога су методе развоја софтвера идентификоване као важан механизам за побољшање објектно-оријентисаних софтверских система.

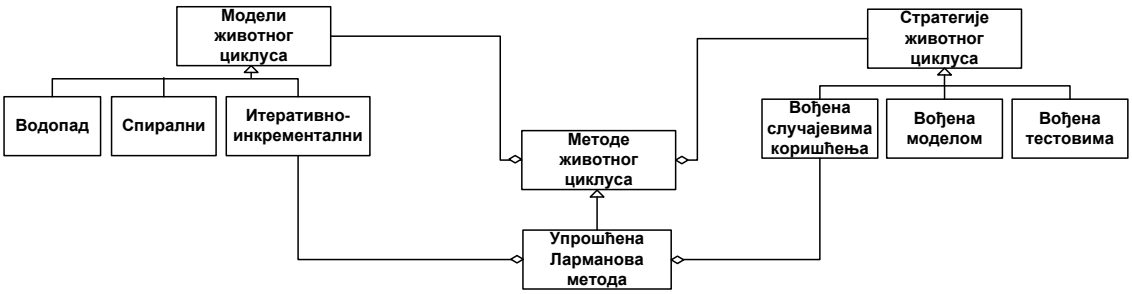
У наставку поглавља даје се опис упрошћене Ларманове методе развоја софтвера. Затим ће на примеру упрошћене Ларманове методе развоја софтвера бити уочена веза метода развоја софтвера са општим принципима пројектовања софтвера, принципима објектно-оријентисаног пројектовања софтвера, стратегијама пројектовања софтвера и патернима пројектовања софтвера.

6.5.1. Упрошћена Ларманова метода развоја софтвера

Упрошћена Ларманова метода развоја софтвера настала је модификацијом Ларманове методе развоја софтвера [Larman05][Vlajic15]. Фокус ове методе је на објектно-оријентисаној анализи и пројектовању софтверског система, уз примену патерна пројектовања софтвера.

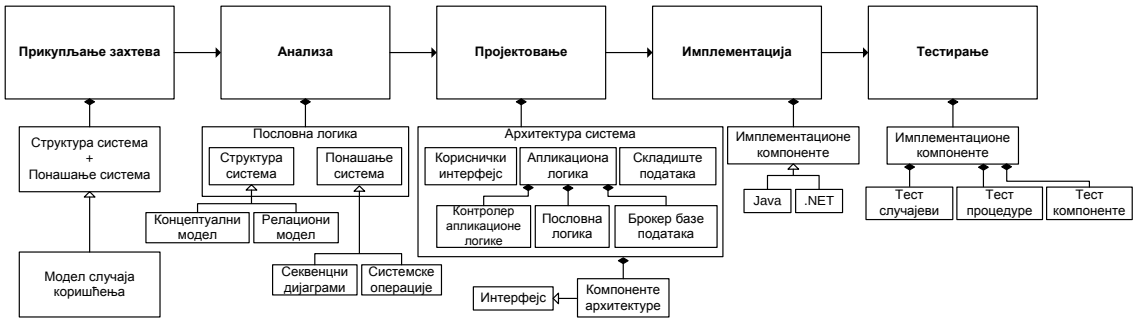
У оквиру упрошћене Ларманове методе развоја користе се различити UML дијаграми у свим фазама процеса развоја софтвера. У том смислу нагласак се ставља на објекте и њихову међусобну сарадњу [Larman05][Vlajic15].

Упрошћена Ларманова метода развоја софтвера користи итеративно-инкрементални модел развоја софтвера и стратегију вођену случајевима коришћења, што је и приказано на наредној слици (Слика 79).



Слика 79. Упрошћена Ларманова метода развоја софтвера

Упрошћена Ларманова метода обухвата следеће фазе: прикупљање захтева, анализа, пројектовање, имплементација и тестирање, што је и приказано на наредној слици (Слика 80).



Слика 80. Фазе и артифакти упрошћене Ларманове методе развоја софтвера [Vlajic15]

У наставку рада ће бити описане фазе упрошћене Ларманове методе развоја софтвера [Vlajic15].

У фази прикупљања захтева дефинишу се захтеви које софтверски систем треба да задовољи [Vlajic15]. Захтеви се описују случајевима коришћења система од стране актора. Случајеви коришћења у себи садрже елементе структуре и понашања софтверског система. Они представљају често коришћену технику за спецификацију корисничких захтева [Cockburn00] [Jacobson93]. Случајеви коришћења се могу представити на различите начине [Cockburn00], али је препорука да се за сваки случај коришћења наводе учесници, актори, предуслови и постуслови који морају бити задовољени, основни сценарио и алтернативна сценарија [Pressman10][Vlajic15]. Случајевима коришћења се описује интеракција између актора и система. Сви случајеви коришћења део су модела случајева коришћења и могу се представити путем UML дијаграма случајева коришћења. На тај начин се омогућава сагледавање функција система[13].

У фази анализе дефинише се логичка структура и понашање софтверског система. Логичка структура се описује помоћу концептуалног и релационог модела, док се понашање описује помоћу секвенцних дијаграма и системских операција [Vlajic15]. Сваки случај коришћења описује

[13] На основу једног случаја коришћења могуће је уочити више функција (системских операција) система. Стога се може рећи да случајеви коришћења представљају жељене (молекулске) функције система, док системске операције представљају основне (атомске) функције система [Vlajic15].

функционалност која се извршава над концептуалним моделом (или делом концептуалног модела). Због тога је веома важно да се у почетној фази развоја софтверског система случајеви коришћења правилно идентификују. Концептуални модел (тј. структура) софтверског система може се представити коришћењем UML дијаграма класа, док се понашање софтверског система може представити коришћењем UML дијаграма секвенци. На основу системских дијаграма секвенци уочавају се системске операције које је потребно пројектовати.

У фази пројектовања даје се спецификација архитектуре софтверског система. У том смислу врши се пројектовање корисничког интерфејса, апликационе логике и складишта података. Апликациона логика се даље декомпонује на контролер апликационе логике, пословну логику и брокер базе података. У оквиру ове фазе врши се израда различитих UML дијаграма интеракције на основу којих се описује начин сарадње објеката у циљу испуњења захтева, као и дијаграма класа на основу којих се описује структура софтверског система.

На основу артифакта из фазе пројектовања у фази имплементације израђују се имплементационе компоненте коришћењем неког програмског језика који подржава објектно-оријентисане програмске концепте (нпр. програмски језици Java или C#).

Фаза тестирања подразумева израду одговарајућих тестова за сваку имплементациону компоненту. У том смислу праве се тест случајеви, тест процедуре и тест компоненте.

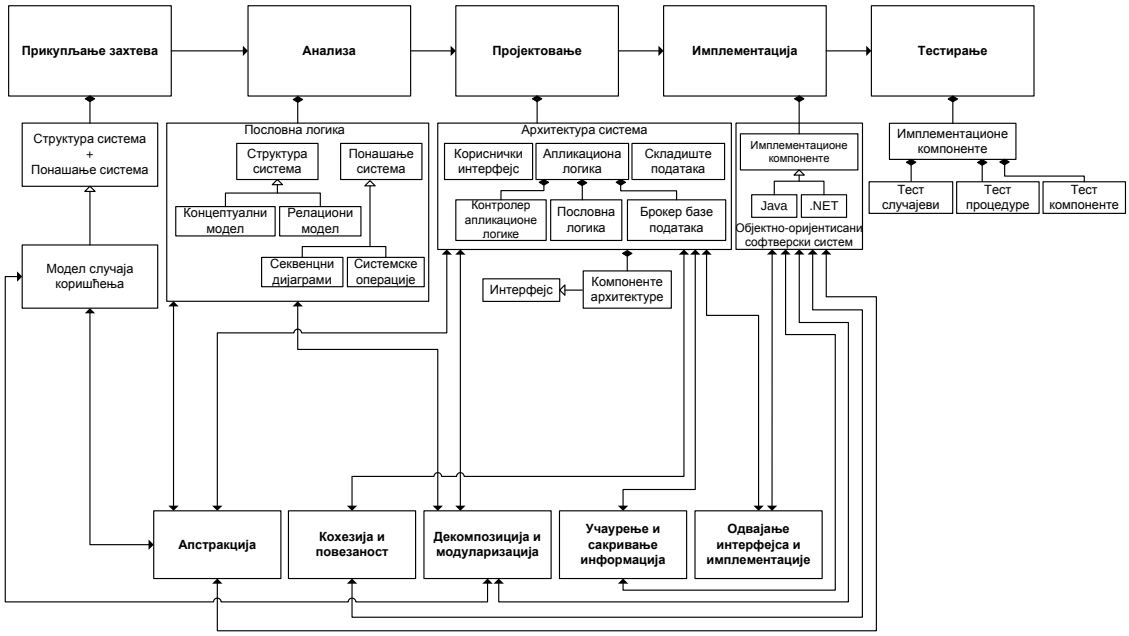
Велика предност упрошћене Ларманове методе развоја софтвера је итеративно-инкрементални приступ процесу развоја софтвера. У том смислу резултати претходне итерације представљају улаз у наредну итерацију тако да даља анализа и пројектовање доводе до унапређења и побољшања софтверског система. Упрошћена Ларманова метода користи објектно-оријентисани приступ и вођена је случајевима коришћења

6.5.2. Веза упрошћене Ларманове метода развоја софтвера са механизмима за побољшање објектно-оријентисаних софтверских система

У наставку рада разматра се веза упрошћене Ларманове методе развоја софтвера са општим принципима пројектовања софтвера, принципима објектно-оријентисаног пројектовања софтвера, стратегијама пројектовања софтвера и патернима пројектовања софтвера.

6.5.2.1. Веза упрошћене Ларманове методе развоја софтвера са општим принципима пројектовања софтвера

Општи принципи пројектовања софтвера представљају основне принципе који се користе у процесу пројектовања сваког софтверског система. На наредној слици (Слика 81) приказана је веза упрошћене Ларманове методе са општим принципима пројектовања софтвера.



Слика 81. Веза упрошћене Ларманове методе са општим принципима пројектовања софтвера

У наставку је дат опис веза упрошћене Ларманове методе са општим принципима пројектовања софтвера.

Веза упрошћене Ларманове методе са општим принципом пројектовања софтвера Апстракција

Апстракција је принцип пројектовања софтвера и усмерена је на постепено и контролисано увођење детаља који су од значаја за софтверски систем. Применом механизма апстракције издвајају се опште информације о посматраном софтверском систему док се детаљи избегавају.

У фази прикупљања захтева функционални захтеви описују се преко модела случаја коришћења. Посматрани модел у себи садржи елементе структуре и понашања софтверског система. Моделом случаја коришћења приказују се актори, случајеви коришћења и везе које се успостављају између њих. Другим речима, на основу модела случаја коришћења могуће је сагледати структуру и понашање софтверског система на високом нивоу апстракције.

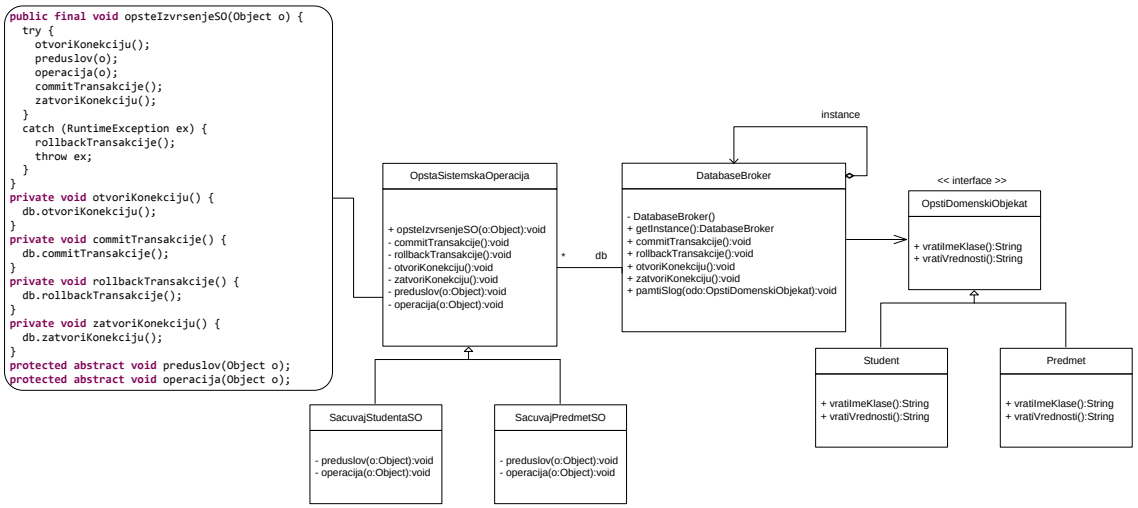
С друге стране, у фази анализе дефинише се логичка структура и понашање софтверског система (пословна логика). У том смислу се структура описује путем концептуалног и релационог модела, док се понашање описује секвенцним дијаграмима и системским операцијама.

У фази пројектовања се врши издвајање компоненти архитектуре софтверског система (кориснички интерфејс, апликациона логика и складишта података), што такође представља примену механизма апстракције. У том смислу је могуће успоставити везу упрошћене Ларманове методе са општим принципом пројектовања софтвера Апстракција.

Веза упрошћене Ларманове методе са општим принципом пројектовања софтвера Кохезија и повезаност

Кохезија је принцип пројектовања софтвера који је усмерен на класу. Стога се може рећи да класа треба да садржи (тј. да учаури) само чланице (атрибуте и методе) које су неопходне за остваривање непосредне функције посматране класе.

У фази пројектовања, у оквиру пословне логике, дефинише се понашање софтверског система путем системских операција. Размотримо пример који је приказан на наредној слици (Слика 82). На слици је приказан дијаграм класа софтверског система, при чему је акценат стављен на структуру класа OpstaSistemskaOperacija, DatabaseBroker и OpstiDomenskiObjekat.



Слика 82. Пример опште системске операције

Класа OpstaSistemskaOperacija је високо кохезивна. Са дијаграма класа се уочава да она као атрибут садржи објекат класе DatabaseBroker. Све методе посматране класе раде са овим објектом, па можемо рећи да се методе класе извршавају над истим скупом атрибута. Другим речима, све чланице посматране класе су у функцији саме класе. У том смислу је класа за опште извршење системских операција високо кохезивна [Vlajic14].

С друге стране, са истог дијаграма класа се примећују класе SacuvajStudentaSO и SacuvajPredmetSO. Ове класе су задужене за реализацију системских операција. Размотримо однос између посматраних класа. У том смислу се не примећује постојање повезаности између посматраних класа (нпр. да методе једне класе користе атрибуте или методе друге класе). Другим речима, на основу дијаграма класа може се закључити да су посматране класе слабо повезане [Vlajic14]. Штавише, приликом примене упрошћене Ларманове методе развоја софтвера системске

операције се пројектују као независне [Vlajic15]. У фази имплементације се врши имплементација претходно дефинисаних компоненти у конкретном програмском језику. На основу изложеног се може закључити да упрошћена Ларманова метода има везу са принципом Кохезија и повезаност.

Веза упрошћене Ларманове методе са општим принципом пројектовања софтвера Декомпозиција и модуларизација

Декомпозиција представља принцип пројектовања софтвера који је усмерен на поделу софтверског система на више модула, чиме настаје модуларизација софтверског система. Упрошћена Ларманова метода се може повезати и са принципом Декомпозиција и модуларизација [Vlajic14]:

- Могуће је извршити декомпозицију код прикупљања захтева. Наиме, почетни кориснички захтев се декомпонује на конкретне захтеве који могу бити представљени путем модела случаја коришћења и конкретних случајева коришћења. На основу сваког посматраног случаја коришћења могуће је уочити једну или више системских операција.
- У фази анализе врши се декомпозиција пословне логике на структуру система и понашање система. У том смислу се структура декомпонује на концептуални и релациони модел, док се понашање декомпонује на секвенцне дијаграме и системске операције.
- С друге стране, могуће је извршити декомпоновање архитектуре софтверског система. На тај начин, уколико се разматра фаза пројектовања, долази се до тронивојске архитектуре која укључује кориснички интерфејс, апликациону логику и складиште података. Апликациону логику је могуће даље декомпоновати на контролер апликационе логике, пословну логику и брокер базе података.
- У оквиру упрошћене Ларманове методе врши се декомпозиција функција. Уколико разматрамо претходно поменуту класу OpstaSistemskaOperacija, примећује се да је почетна функција

opsteIzvrsenjeSO() декомпонована на помоћне функције које се користе у поступку реализације посматране функционалности. На тај начин извршена је декомпозиција функција.

Веза упрошћене Ларманове методе са општим принципом пројектовања софтвера Учаурење и сакривање информација

У процесу развоја софтвера треба пројектовати класе које садрже (тј. учаурују) само чланице класе које су неопходне како би се остварила непосредна функција посматране класе или софтверске компоненте. На тај начин се врши класификација особина софтверског система према начину приступа. Другим речима, потребно је разликовати особине класе које су јавне и особине класе које су сакривене од других модула система.

У том смислу, уколико разматрамо фазу пројектовања и претходно поменуту класу OpstaSistemskaOperacija, може се уочити да су поједине методе класе приватне и не може им се приступити ван класе. Неким методама посматране класе се може приступити само из подкласа, а методи *opsteIzvrsenjeSO()* је могуће приступити из сваке класе. На основу тога се изводи закључак да упрошћена Ларманова метода има везу са принципом Учаурење и сакривање информација.

Веза упрошћене Ларманове методе са општим принципом пројектовања софтвера Одвајање интерфејса и имплементације

Уколико се разматра фаза пројектовања, долази се до тронивојске архитектуре при чему су нивои архитектуре дефинисани путем скупа класа које представљају компоненте архитектуре софтверског система [Vlajic14]. Компоненте су дефинисане преко интерфејса чиме се врши одвајање интерфејса од имплементације. У том смислу, уколико се разматра претходно поменута класа DatabaseBroker (приказана на слици Слика 82), примећује се да она ради са интерфејсом OpstiDomenskiObjekat, а не са имплементацијама тог интерфејса. Интерфејсом се даје спецификација

функција посматраног модула, при чему детаљи имплементације остају сакривени. Такође, на тај начин је могуће направити различите имплементације посматраног интерфејса (у нашем примеру то су класе Student и Predmet). У том смислу, упрошћену Ларманову методу развоја софтвера могуће је повезати и са принципом одвајања интерфејса од имплементације.

С обзиром да се у фази имплементације врши имплементација претходно дефинисаних компоненти у конкретном програмском језику (нпр. Java или C#), ова фаза се може повезати са свим општим принципима пројектовања софтвера.

У наредној табели (Табела 72) дат је сумарни приказ веза упрошћене Ларманове методе развоја софтвера са општим принципима пројектовања софтвера.

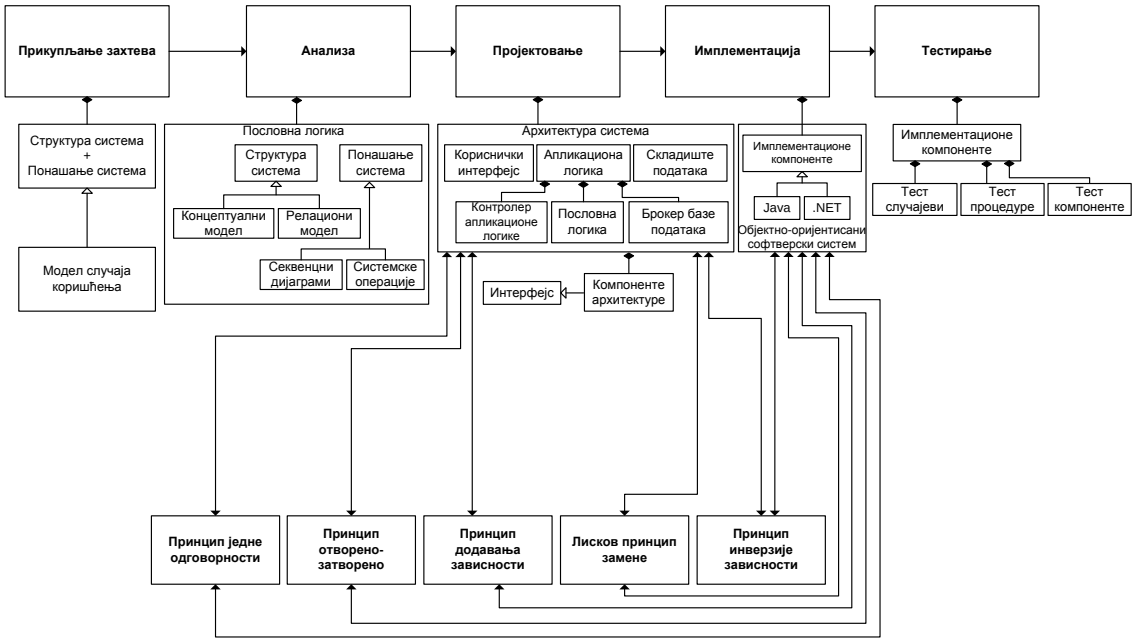
Табела 72. Веза упрошћене Ларманове методе развоја софтвера са општим принципима пројектовања софтвера

Метода развоја софтвера	**Веза са општим принципима пројектовања софтвера**
Упрошћена Ларманова метода развоја софтвера	Апстракција, Кохезија и повезаност, Декомпозиција и модуларизација, Учаурење и сакривање информација, Одвајање интерфејса и имплементације

6.5.2.2. Веза упрошћене Ларманове методе развоја софтвера са принципима објектно-оријентисаног пројектовања софтвера

Принципи објектно-оријентисаног пројектовања представљају основне принципе који се користе приликом објектно-оријентисаног програмирања и пројектовања класа. С обзиром да упрошћена Ларманова метода користи објектно-оријентисани приступ могуће је успоставити везу методе са принципима објектно-оријентисаног пројектовања софтвера. Веза

упрошћене Ларманове методе са принципима објектно-оријентисаног пројектовања софтвера је приказана на наредној слици (Слика 83).



Слика 83. Веза упрошћене Ларманове методе са принципима објектно-оријентисаног пројектовања софтвера

У наставку је дат опис веза упрошћене Ларманове методе са принципима објектно-оријентисаног пројектовања софтвера.

Веза упрошћене Ларманове методе са принципом једне одговорности

Принцип једне одговорности усмерен је на одговорност класе у софтверском систему. Према овом принципу класа треба да има једну одговорност у посматраном софтверском систему

Размотримо пример који је приказан на претходној слици (Слика 82). Са слике се уочавају системске операције SacuvajStudentaSO и SacuvajPredmetSO које чувају податке о студенту и предмету. Са слике се, такође, уочава класа DatabaseBroker која је задужена да извршава операције

над базом података[14]. Са слике се, такође, уочава да су све чланице посматраних класа (атрибути и методе класа) у функцији тих класа. У том смислу, свака од поменутих класа има тачно једну одговорност па се упрошћена Ларманова метода развоја софтвера може повезати са Принципом једне одговорности.

Веза упрошћене Ларманове методе са принципом отворено-затворено

У процесу објектно-оријентисаног развоја софтвера ентитети (нпр. класе, модули, функције итд.) требају да буду отворени за проширења али и затворени за модификацију.

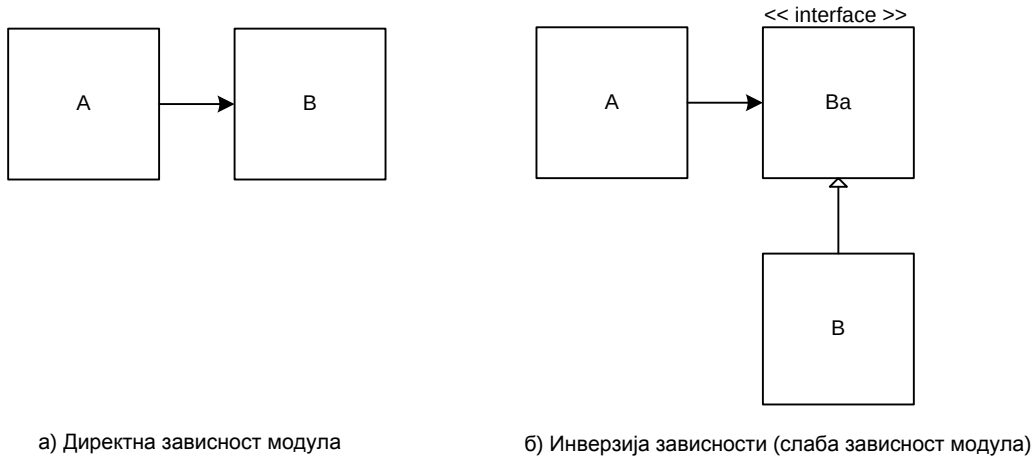
Уколико размотримо пример који је приказан на претходној слици (Слика 82), уочава се да општа класа за извршење системске операције (класа OpstaSistemskaOperacija) има генеричку методу *opsteIzvrsenjeSO()* која је затворена за промене на вишем нивоу апстракције, али је и отворена за промене на нижем нивоу апстракције. Преко ње је дефинисано заједничко понашање за класе SacuvajStudentaSO и SacuvajPredmetSO. У том смислу метода *opsteIzvrsenjeSO()* дефинише скелет алгоритма који се не може променити и увек подразумева исти редослед корака. С друге стране, класама SacuvajStudentaSO и SacuvajPredmetSO је дозвољено да промене поједине кораке тог алгоритма.

Веза упрошћене Ларманове методе са принципом инверзије зависности

Применом принципа инверзије зависности избегава се директна зависност између модула већ се та зависност остварује посредно, путем апстракције (најбоље би било да се зависност између модула остварује на нивоу интерфејса). У општем случају, уколико имамо два модула A и B између којих постоји директна зависност, применом принципа инверзије

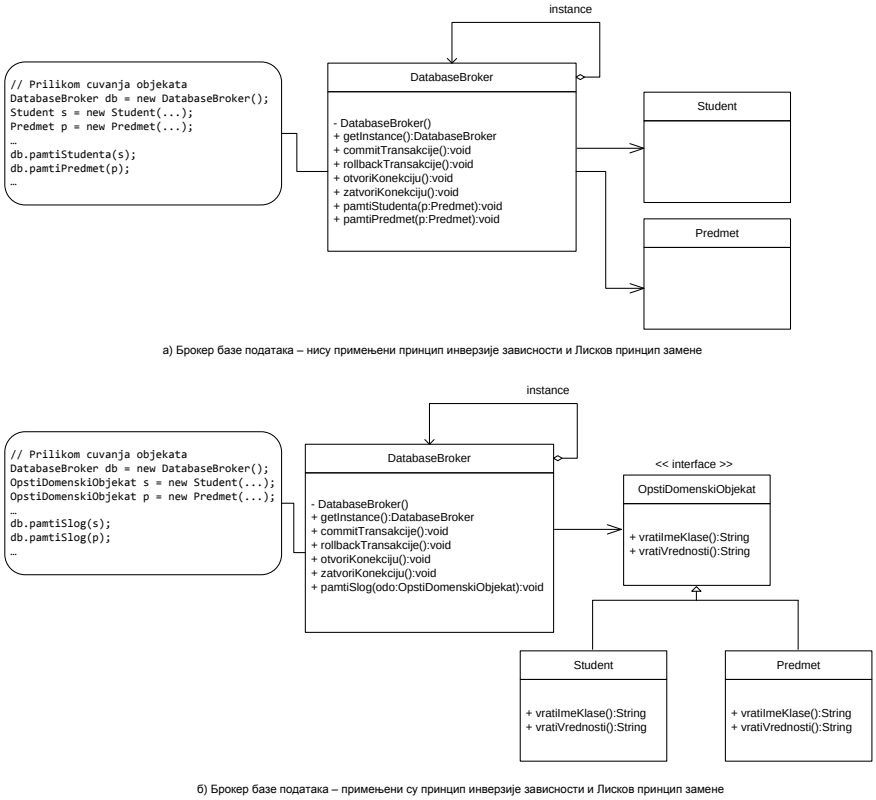
[14] На дијаграму класа је дат пример за операцију чувања. На сличан начин се могу пројектовати и остале операције (селекција, измена и брисање).

зависности уводи се модул Ba из кога је изведен модул B. На тај начин се врши инверзија зависности, што је и приказано на наредној слици (Слика 84).



Слика 84. Принцип инверзије зависности - општи случај

Размотримо пример који је приказан на наредној слици (Слика 85).



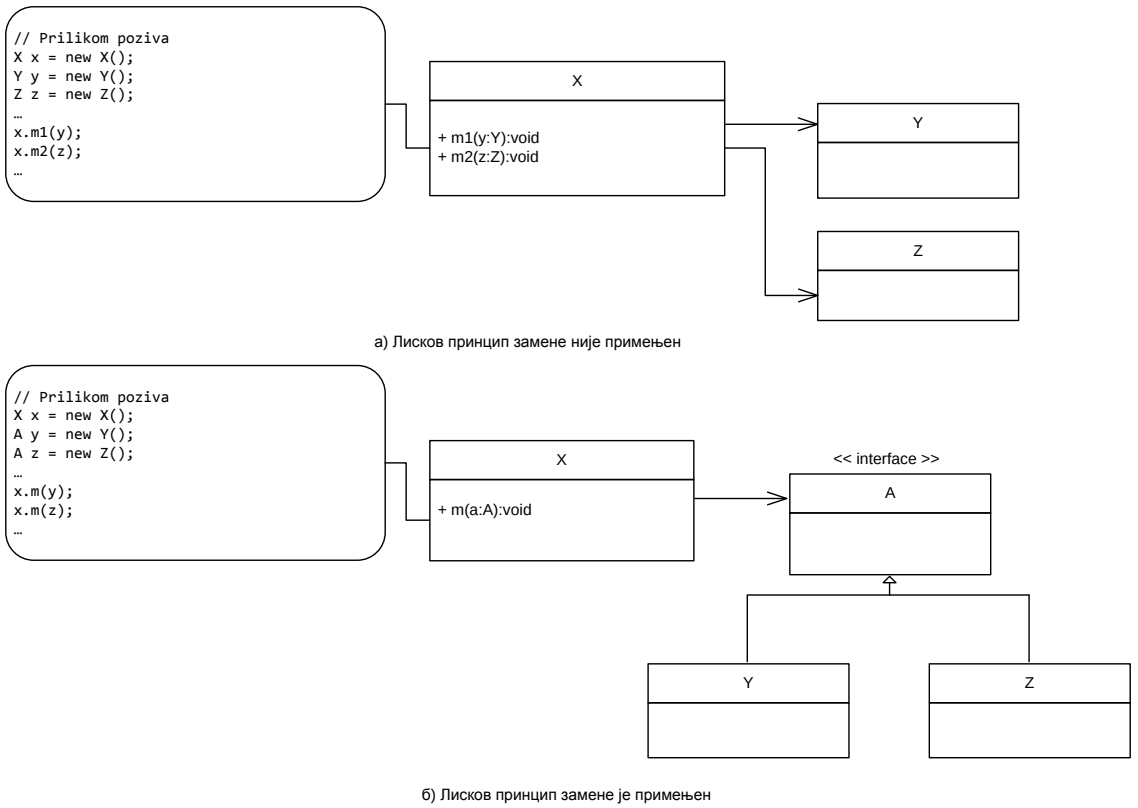
Слика 85. Примена принципа инверзије зависности и Лисковог принципа замене

У фази пројектовања врши се пројектовање брокера базе података који је задужен за непосредну комуникацију са складиштем података. Уколико размотримо пример који је приказан на слици (Слика 85а), примећује се директна повезаност класе DatabaseBroker са доменским класама Student и Predmet. Другим речима, за сваку доменску класу се у брокеру базе података дефинишу посебне методе за памћење објекта. На тај начин је успостављена директна зависност између класа и додавање нове доменске класе захтевало би додавање нове методе у класи DatabaseBroker.

С друге стране, уколико размотримо пример који је приказан на слици (Слика 85б) може се уочити да су методе класе DatabaseBroker генеричке и омогућавају чување било ког објекта који је изведен из интерфејса OpstiDomenskiObjekat. Другим речима, методе посматране класе користе интерфејс OpstiDomenskiObjekat па су самим тим у могућности да користе и објекте изведених класа које имплементирају тај интерфејс. У конкретном примеру из посматраног интерфејса су изведене доменске класе Student и Predmet. Сходно томе, класа DatabaseBroker може да извршава операције чувања објеката класа Student и Predmet. У ту сврху користе се објектно-оријентисани концепти који се називају наслеђивање и полиморфизам. На тај начин је примењен Принцип инверзије зависности.

Веза упрошћене Ларманове методе са Лисковим принципом замене

Према Лисковом принципу замене функције које користе показиваче или референце ка основним класама (тј. надкласама) морају бити у могућности да користе и објекте изведених класа (тј. подкласа) [Martin96]. Општи случај примене Лисковог принципа замене приказан је на наредној слици (Слика 86).



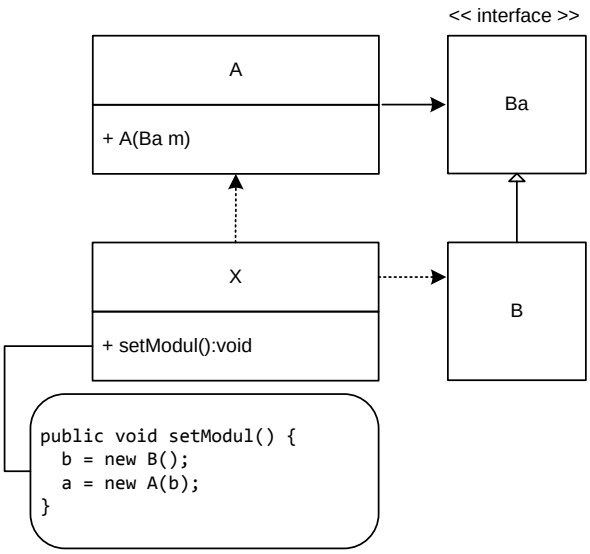
Слика 86. Лисков принцип замене - општи случај

Раније је напоменуто да су у примеру са слике (Слика 85б) методе класе DatabaseBroker генеричке и омогућавају чување било ког објекта који је изведен из интерфејса OpstiDomenskiObjekat. У конкретном примеру из посматраног интерфејса су изведене доменске класе Student и Predmet што практично значи да класа DatabaseBroker може да извршава операције чувања над поменутим доменским класама. Другим речима, интерфејс OpstiDomenskiObjekat је заменљив са класама Student и Predmet. На тај начин је примењен Лисков принцип замене.

Веза упрошћене Ларманове методе са принципом додавања зависности

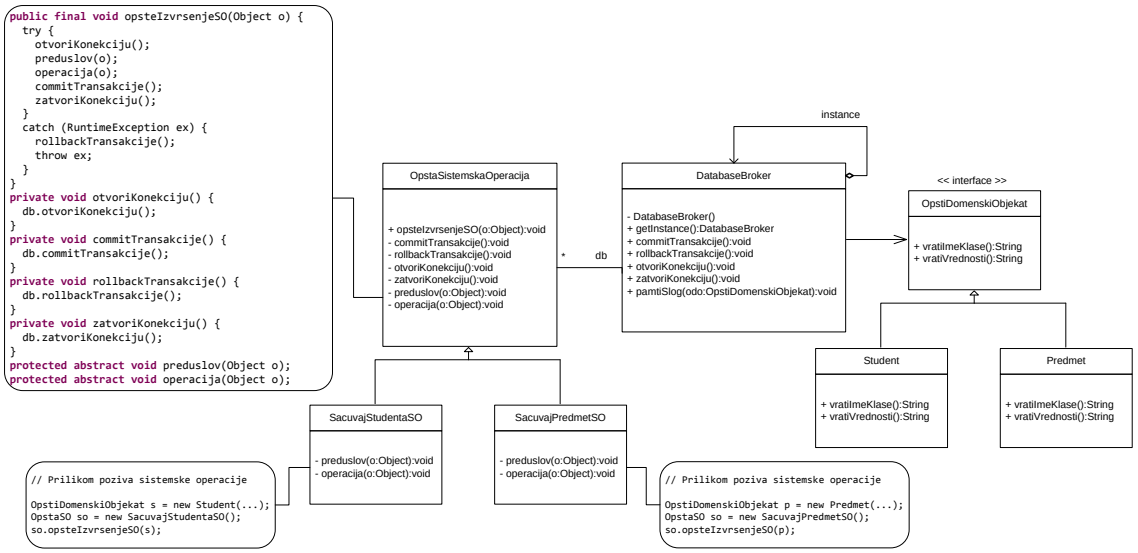
Према принципу додавања зависности, зависност између два модула објектно-оријентисаног софтверског система се конкретно успоставља у време извршавања програма. Зависност се успоставља коришћењем неког трећег модула, што је и приказано на наредној слици (Слика 87). Са слике се

уочавају два модула (модул A и модул B, при чему модул B представља реализацију интерфејса Ba) између којих је потребно успоставити зависност. Успостављање зависности између модула реализовано је преко модула X.



Слика 87. Принцип додавања зависности - општи случај

Уколико размотримо пример који је приказан на наредној слици (Слика 88) може се закључити да су методе класе DatabaseBroker генеричке и да раде са интерфејсом OpstiDomenskiObjekat. С друге стране, класе Student и Predmet имплементирају поменути интерфејс и повезивање брокера базе података са конкретним доменским класама реализује се преко класе OpstaSistemskaOperacija која је саставни део пословне логике посматраног софтверског система. У том смислу се упрошћена Ларманова метода развоја софтвера може повезати са принципом додавања зависности.



Слика 88. Примена принципа додавања зависности

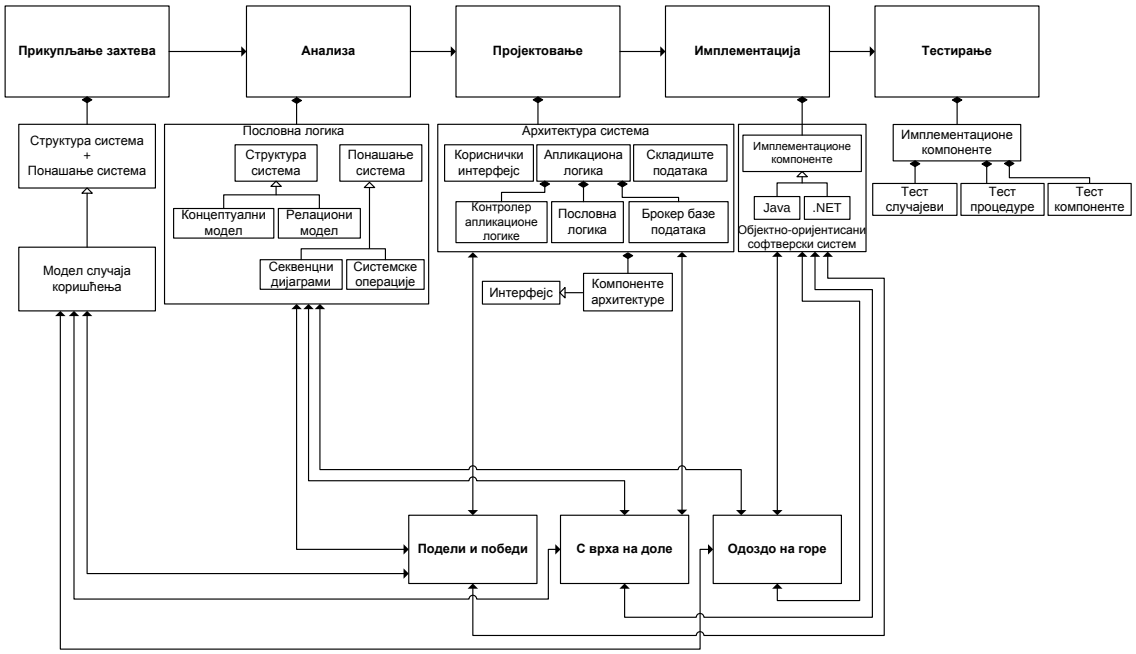
С обзиром да се у фази имплементације врши имплементација претходно дефинисаних компоненти у конкретном програмском језику (нпр. Java или C#), ова фаза се може повезати са свим претходно поменутим принципима објектно-оријентисаног пројектовања софтвера. У наредној табели (Табела 73) дат је сумарни приказ веза упрошћене Ларманове методе развоја софтвера са принципима објектно-оријентисаног пројектовања софтвера.

Табела 73. Веза упрошћене Ларманове методе развоја софтвера са принципима објектно-оријентисаног пројектовања софтвера

Метода развоја софтвера	**Веза са принципима објектно-оријентисаног пројектовања софтвера**
Упрошћена Ларманова метода развоја софтвера	Принцип једне одговорности, Принцип отворено-затворено, Лисков принцип замене, Принцип инверзије зависности, Принцип додавања зависности

6.5.2.3. Веза упрошћене Ларманове методе развоја софтвера са стратегијама пројектовања софтвера

Стратегијама пројектовања софтвера дефинише се стратешки приступ процесу пројектовања софтвера. На наредној слици (Слика 89) приказана је веза упрошћене Ларманове методе са стратегијама пројектовања софтвера.



Слика 89. Веза упрошћене Ларманове методе са стратегијама пројектовања софтвера

У наставку је дат опис веза упрошћене Ларманове методе са стратегијама пројектовања софтвера.

Веза са стратегијом Подели и победи

Стратегија *Подели и победи* је заснована на принципу декомпозиције, тј. на принципу поделе почетног проблема на више потпроблема. У том смислу се промовише дељење почетног проблема на мање проблеме који се могу независно решавати. Потпреблеми се, такође, могу даље декомпоновати, док се не дође до елементарних потпроблема који се на једноставан начин могу решити.

Уколико разматрамо процес развоја софтвера, у фази прикупљања захтева почетни захтев се представља моделом случајева коришћења који се састоји

од више случаја коришћења. Они у себи садрже елементе структуре и понашања софтверског система.

У фази анализе врши се раздвајање структуре и понашања софтверског система. У том смислу се врши дефинисање логичке структуре и понашања софтверског система. Логичка структура се описује путем концептуалног и релационог модела, док је понашање описује путем секвенцних дијаграма и системских операција.

У фази пројектовања дефинише се архитектура софтверског система. Архитектура софтверског система обухвата кориснички интерфејс, апликациону логику и складиште података. С друге стране, апликациону логику је могуће поделити на контролер апликационе логике, пословну логику и брокер базе података, као што је и приказано на претходној слици (Слика 89). На тај начин примењена је стратегија Подели и победи.

Веза са стратегијом С врха на доле

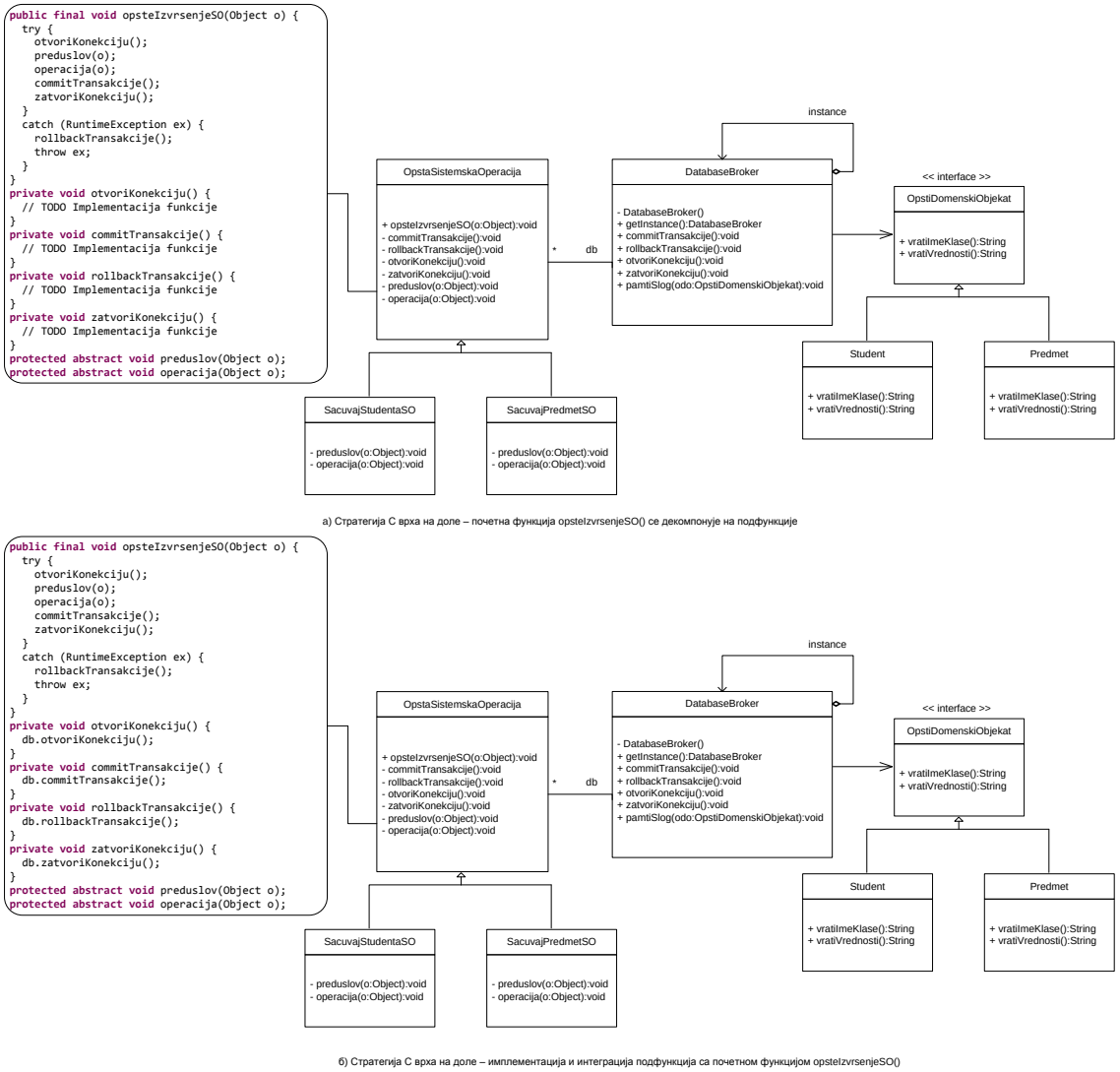
Стратегија С врха на доле односи се на декомпозицију функција. У том смислу се врши декомпозиција функција на више подфункција које се могу независно решавати.

Уколико разматрамо фазу пројектовања, у оквиру пословне логике пројектују се системске операције. Уколико размотримо класу OpstaSistemskaOperacija која је приказана на наредној слици (Слика 90), може се закључити да је почетна функција *opsteIzvrsenjeSO()* декомпонована на следећи начин:

1. Најпре се даје имплементација почетне функције *opsteIzvrsenjeSO()*. Приликом израде функције уочава се да је потребно извршити отварање конекције, проверу предуслова, извршење конкретне системске операције, потврду/поништење трансакције и затварање конекције. Због тога се почетна функција декомпонује на функције

otvoriKonekciju(), *preduslov()*, *operacija()*, *commitTransakcije()*, *rollbackTransakcije()* и *zatvoriKonekciju()*.

2. Након тога се подфункције разрађују. У том смислу се врши имплементација посматраних функција. Другим речима, посматране подфункције су интегрисане у функцију *opsteIzvrsenjeSO()* чиме се реализује опште извршење системске операције. На тај начин је примењена стратегија C врха на доле.



а) Стратегија C врха на доле – почетна функција opsteIzvrsenjeSO() се декомпонује на подфункције

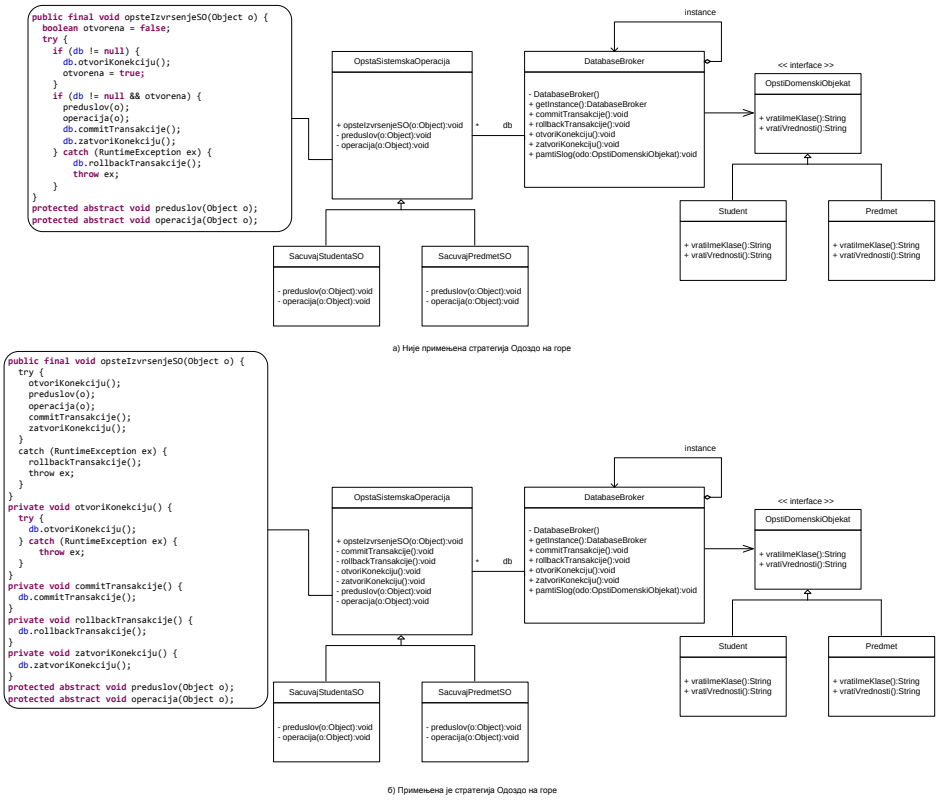
б) Стратегија C врха на доле – имплементација и интеграција подфункција са почетном функцијом opsteIzvrsenjeSO()

Слика 90. Примена стратегије C врха на доле у процесу пројектовања системских операција

Веза са стратегијом Одоздо на горе

Стратегија *Одоздо на горе* користи принцип генерализације. У том смислу се врши израда почетне функције а затим, уколико је посматрана функција сложена, уочавање једне или више логичких целина које се проглашавају за функције.

Размотримо пример који је приказан на наредној слици (Слика 91а). Са слике се уочава функција *opsteIzvrsenjeSO()* којом се дефинише опште извршење системске операције. У оквиру посматраног примера постоји само једна метода која садржи комплетан програмски код. У том смислу је метода сложена за одржавање па је могуће уочити више логичких целина и прогласити их за функције. Због тога су уочене функције *otvoriKonekciju()*, *commitTransakcije()*, *rollbackTransakcije()* и *zatvoriKonekciju()*, као што је и приказано на слици (Слика 91б). Наведене функције се затим позивају из почетне функције чиме се реализује стратегија Одоздо на горе.



Слика 91. Примена стратегије Одоздо на горе у процесу пројектовања системских операција

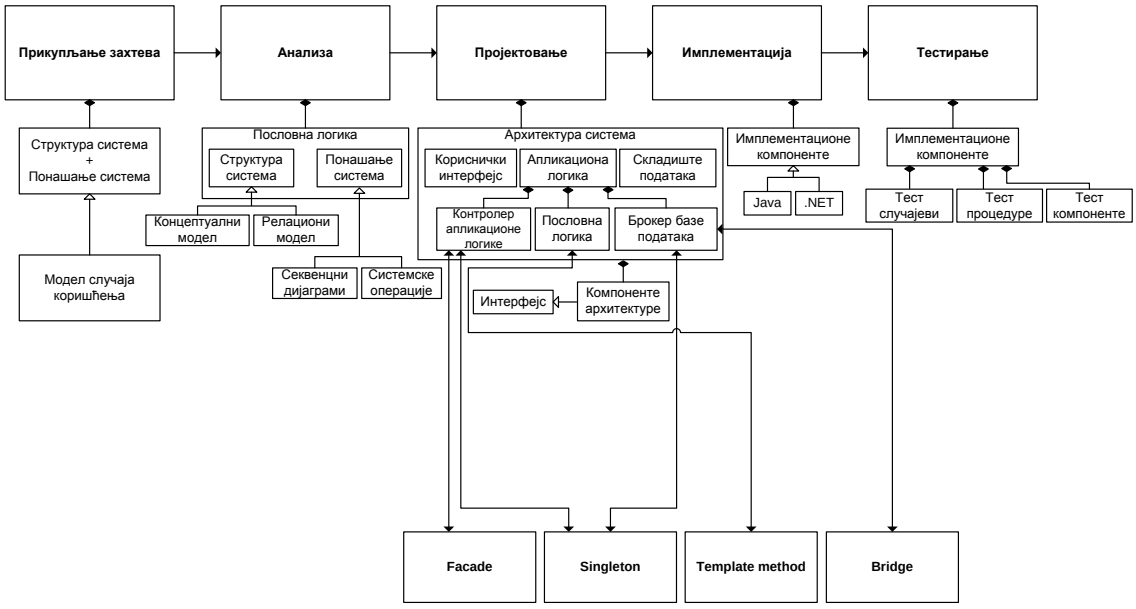
У наредној табели (Табела 74) дат је сумарни приказ веза упрошћене Ларманове методе развоја софтвера са стратегијама пројектовања софтвера.

Табела 74. Веза упрошћене Ларманове методе развоја софтвера са стратегијама пројектовања софтвера

Метода развоја софтвера	**Веза са стратегијама пројектовања софтвера**
Упрошћена Ларманова метода развоја софтвера	Подели и победи, С врха на доле, Одоздо на горе

6.5.2.4. Веза упрошћене Ларманове методе развоја софтвера са патернима пројектовања софтвера

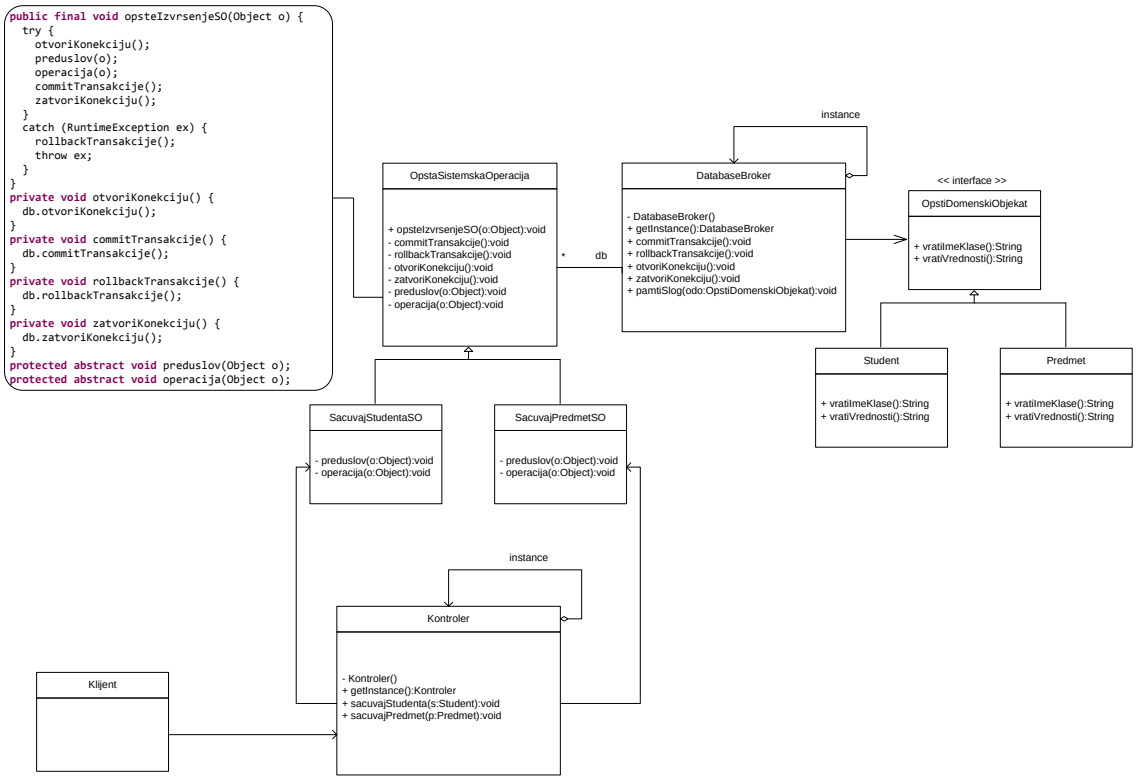
Патерни пројектовања софтвера у најопштијем смислу представљају решење неког проблема у неком контексту, при чему се та решења могу користити за решавање других проблема [Gamma94][Vlajic14b]. На наредној слици (Слика 92) приказана је веза упрошћене Ларманове методе са патернима пројектовања софтвера.



Слика 92. Веза упрошћене Ларманове методе са патернима пројектовања софтвера

У наставку је дат опис веза упрошћене Ларманове методе са патернима пројектовања софтвера. С обзиром да је за разматрање веза са патернима

пројектовања софтвера потребан приказ структуре и понашања софтверског система, размотрићемо пример који је приказан на наредној слици (Слика 93).



Слика 93. Примењени софтверски патерни у имплементацији софтверског система

Веза упрошћене Ларманове методе са Singleton патерном пројектовања софтвера

Из приказаног дијаграма класе се уочава да класе Kontroler и DatabaseBroker имају само једно појављивање. Посматране класе имају приватне конструкторе и јавне методе *getInstance()* којима се обезбеђује креирање само једног објекта посматраних класа. У том смислу примењен је софтверски патерн који се назива Singleton. Софтверски патерн Singleton обезбеђује само једно креирање појављивања неког објекта и глобални приступ до њега [Gamma94][Vlajic14b].

Веза упрошћене Ларманове методе са Facade патерном пројектовања софтвера

Са посматраног дијаграма класа се такође уочава да клијент остатку система приступа посредно, путем класе Kontroler. Контролер је, у том смислу, задужен да прихвати захтев од клијента и проследи га до остатка система, тј. до конкретне системске операције која треба да изврши посматрани захтев. Након извршења системске операције Контролер је задужен да резултат извршења врати назад клијенту. У том смислу је за реализацију контролера примењен софтверски патерн Facade. Софтверски патерн Facade обезбеђује јединствен интерфејс за скуп интерфејса неког подсистема, чиме се обезбеђује лакше коришћење посматраног подсистема [Gamma94][Vlajic14b].

Веза упрошћене Ларманове методе са Template method патерном пројектовања софтвера

У посматраном софтверском систему је могуће уочити класу OpstaSistemskaOperacija која дефинише скелет алгоритма за извршавање системских операције преко методе *opsteIzvrsenjeSO()*. Сам алгоритам се не може променити и увек подразумева исти редослед корака (дефинисан је методом *opsteIzvrsenjeSO*). С друге стране, класама SacuvajStudentaSO и SacuvajPredmetSO је дозвољено да промене поједине кораке тог алгоритма. На тај начин је примењен софтверски патерн Template method. Софтверски патерн Template method дефинише скелет алгоритма у операцији, при чему препушта извршење појединих корака операција подкласама [Gamma94][Vlajic14b].

Веза упрошћене Ларманове методе са Bridge патерном пројектовања софтвера

Са дијаграма класа се уочава класа DatabaseBroker која је задужена да извршава операције базе података. Она је непосредно повезана са интерфејсом OpstiDomenskiObjekat и омогућава перзистенцију објеката који имплементирају посматрани интерфејс (у примеру су то класе Student и Predmet). На тај начин примењен је софтверски патерн Bridge. Софтверски

патерн Bridge одваја апстракцију од њене имплементације, чиме се оне могу мењати независно [Gamma94][Vlajic14b].

У наредној табели (Табела 75) дат је сумарни приказ веза упрошћене Ларманове методе развоја софтвера са патернима пројектовања софтвера.

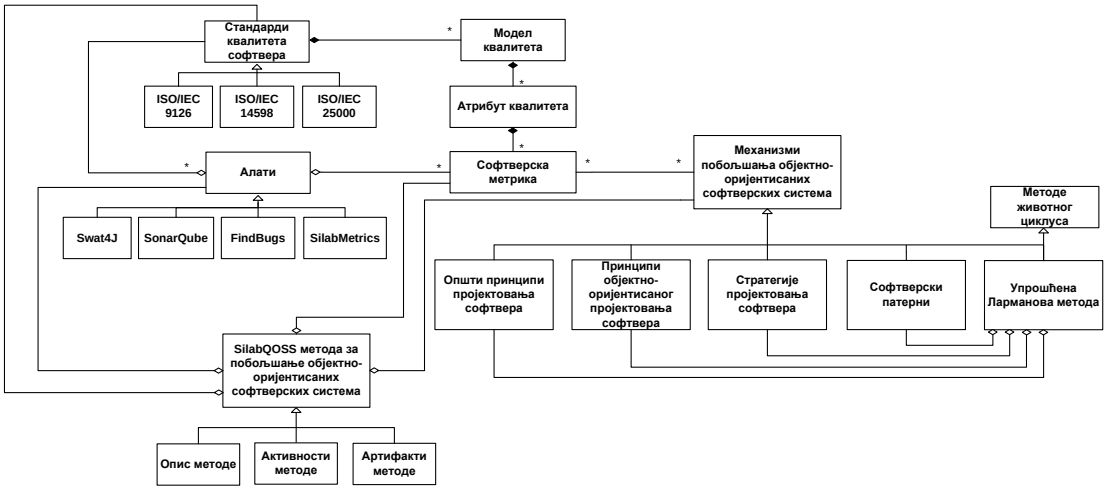
Табела 75. Веза упрошћене Ларманове методе развоја софтвера са патернима пројектовања софтвера

Метода развоја софтвера	**Веза са патернима пројектовања софтвера**
Упрошћена Ларманова метода развоја софтвера	Singleton, Facade, Template method, Bridge

На основу свега изложеног може се закључити да се упрошћена Ларманова метода може повезати са механизмима са побољшање објектно-оријентисаних софтверских система. У том смислу, у оквиру посматране методе примењене су најбоље праксе које се односе на процес развоја софтвера. Због тога можемо рећи да и примена упрошћене Ларманове методе развоја софтвера представља један од механизама за побољшање објектно-оријентисаних софтверских система.

7. SilabQOSS метода за побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера

У оквиру овог поглавља биће представљена сопствена SilabQOSS (енг. Silab Quality Method for Object-oriented Software Systems) метода за побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера. Предложена метода користи стандарде квалитета софтвера, софтверске метрике и алате за статичку анализу квалитета софтвера. Такође, метода користи механизме за побољшање објектно-оријентисаних софтверских система. Због тога се у наставку најпре даје опис предложене методе. Затим ће бити дефинисане активности и артифакти предложене методе. На наредној слици (Слика 94) дат је концептуални приказ SilabQOSS методе за побољшање објектно-оријентисаних софтверских система.



Слика 94. Концептуални приказ SilabQOSS методе за побољшање објектно-оријентисаних софтверских система

7.1. Опис предложене методе

SilabQOSS метода за побољшање објектно-оријентисаних софтверских система развијена је у оквиру Лабораторије за софтверско инжењерство Факултета организационих наука (Универзитет у Београду). Метода је

усмерена на објектно-оријентисане софтверске системе[15] који су нашли велику практичну примену у процесу животног циклуса развоја софтвера.

Карактеристика методе огледа се у примени стандарда квалитета софтвера[16] кроз све њене активности. Стандардима квалитета софтвера даје се спецификација модела квалитета софтвера, тј. даје се спецификација атрибута квалитета софтвера. За оперативно мерење атрибута квалитета софтвера непосредно се користе софтверске метрике[17]. С друге стране, на основу атрибута квалитета могуће је утврдити ниво квалитета читавог софтверског система. Другим речима, софтверским метрикама се посредно мери ниво квалитета читавог софтверског система.

Стандарди квалитета софтвера развијени су од стране различитих међународних организација и у њиховом развоју учествују различите заинтересоване стране (нпр. индустрија, академска заједница, истраживачке групе и појединци). С друге стране, стандарди квалитета софтвера и софтверске метрике су подржане од стране различитих алата за статичку анализу квалитета софтвера[18]. Због тога метода разматра стандарде квалитета софтвера, софтверске метрике и алате за статичку анализу квалитета софтвера у контексту побољшања објектно-оријентисаних софтверских система.

С обзиром на објектно-оријентисану усмереност, метода обухвата различите механизме побољшања објектно-оријентисаних софтверских система[19]:

- Општи принципи пројектовања софтвера - Представљају основне принципе који се могу користити приликом пројектовања сваког софтверског система. Идентификовани су следећи општи принципи

[15] Објектно-оријентисани софтверски системи описани су у Поглављу 2.
[16] Стандарди квалитета софтвера описани су у Поглављу 3.
[17] Софтверске метрике су описане у Поглављу 4.
[18] Алати за статичку анализу квалитета софтвера описани су у Поглављу 5.
[19] Механизми побољшања објектно-оријентисаних софтверских система описани су у Поглављу 6.

пројектовања софтвера: кохезија и повезаност, декомпозиција и модуларизација, учаурење и сакривање информација, одвајање интерфејса и имплементације, и довољност, комплетност и једноставност.

- Принципи објектно-оријентисаног пројектовања софтвера - Представљају принципе који се користе приликом објектно-оријентисаног програмирања и пројектовања класа. Као принципи објектно-оријентисаног пројектовања идентификовани су принцип једне одговорности, принцип отворено-затворено, Лисков принцип замене, принцип инверзије зависности, принцип додавања зависности и принцип издвајања интерфејса.
- Стратегије пројектовања софтвера - Стратегијама пројектовања софтвера се дефинише стратешки приступ процесу пројектовања софтвера. Идентификоване су следеће стратегије пројектовања софтвера: подели и победи, с врха на доле, одоздо на горе и итеративно-инкрементални приступ.
- Патерни пројектовања софтвера - Представљају решење неког проблема у неком контексту, при чему се та решења могу користити за решавање других проблема [Gamma94][Vlajic14b]. Идентификовани су следећи патерни пројектовања софтвера: патерни за креирање објекта, структурни патерни и патерни понашања.
- Методе развоја софтвера - Методом развоја софтвера описује се процес развоја софтвера. Методе развоја софтвера могу да користе претходно поменуте принципе, стратегије и патерне пројектовања софтвера. У том смислу се предлаже коришћење упрошћене Ларманове методе развоја софтвера која користи објектно-оријентисани приступ процесу развоја софтвера и софтверске патерне.

Као што је раније напоменуто, стандардима квалитета софтвера дефинишу се модели и атрибути квалитета софтвера, док се за оперативно мерење

атрибута квалитета софтвера користе софтверске метрике. Другим речима, стандарди квалитета, модели квалитета софтвера, атрибути квалитета софтвера и софтверске метрике представљају контекст у процесу развоја објектно-оријентисаних софтверских система. У оквиру тог контекста разматрају се неусаглашености софтверског система са стандардима квалитета софтвера која се, уз коришћење механизама побољшања објектно-оријентисаних софтверских система, трансформишу у усаглашености софтверског система са стандардима квалитета софтвера, што је и приказано на наредној слици (Слика 95).

Слика 95. Механизми побољшања објектно-оријентисаних софтверских система у контексту стандарда квалитета софтвера

Примена стандарда квалитета софтвера, софтверских метрика, алата за статичку анализу квалаитета софтвера и механизама за побољшање објектно-оријентисаних софтверских система повећава ниво квалитета посматраних софтверских система. На тај начин се омогућава развој софтверских система који су у складу са дефинисаним моделом квалитета, софтверским метрикама, принципима, стратегијама и патернима пројектовања софтвера. Другим речима, њиховим коришћењем се омогућава брз увид у квалитет програмског кода. Инжењери за развој софтвера су у том смислу усмерени да пишу програмски код који је једноставан, квалитетан и лако разумљив свим учесницима у процесу развоја софтвера. На тај начин је омогућено брзо уклапање инжењера за развој софтвера у

процес развоја софтвера, уз поштовање правила која се односе на квалитет софтверског система који се непосредно развија.

SilabQOSS метода промовише интензивну употребу алата за статичку анализу квалитета софтвера. Метода је подржана од стране Swat4J, SonarQube алата, као и SilabMetrics алата који је заснован на програмском пакету SonarQube [Milic17]. Алати за статичку анализу квалитета софтвера најчешће садрже предефинисани модел квалитета софтвера. Међутим, с обзиром на различитост софтверских система који се непосредно развијају, алати дозвољавају конфигурацију и измену предефинисаног модела квалитета софтвера (у смислу атрибута квалитета софтвера, софтверских метрика и веза атрибута квалитета софтвера са софтверским метрикама). На тај начин се омогућава дефинисање специфичног значења квалитета у односу на софтверски систем који се непосредно развија. Ове измене у моделу квалитета омогућавају учесницима у процесу развоја софтвера усвајање знања која настају као непосредна последица различитих "шта-ако" анализа квалитета софтвера (нпр. шта ће се десити уколико се у софтверском систему вредност одређеног параметра квалитета софтвера повећа/смањи; у којим границама се могу кретати одређени параметри квалитета софтвера; како промена вредности неког параметра квалитета софтвера утиче на остале параметре и сл.). Ове анализе, такође, упућују учеснике у процесу развоја софтвера да критички сагледају процес развоја софтвера у контексту задовољења стандарда квалитета софтвера, као и да размишљају како би евентуалне неусаглашености са моделом квалитета софтвера требало разрешити. На тај начин се могу практично применити различита знања о процесу развоја софтвера.

С друге стране, поменути алати за статичку анализу квалитета софтвера се могу интегрисати са окружењима за развој софтвера. На пример, SilabMetrics алат за статичку анализу квалитета се може интегрисати са NetBeans развојним окружењем, SonarQube алат се може интегрисати са Eclipse и IntelliJ IDEA развојним окружењима итд. У том смислу је могуће

брзо уочавање грешака које настају у процесу развоја софтвера. Уколико су алати интегрисани са окружењем за развој софтвера неусаглашеност са моделом квалитета развоја софтвера се на одговарајући начин приказује (нпр. модел квалитета посматраног софтверског система се може графички приказати, програмски код се може обележити другачијом бојом, линије програмског кода могу бити подвучене и сл.). Поред тога, алати могу дати опис нарушавања модела квалитета, са референцама ка другим релевантним документима (нпр. стандардима квалитета софтвера, спецификацијама, интернет ресурсима итд.) где се могу пронаћи додатне информације о нарушеном моделу квалитета, односно нарушеној софтверској метрици. На тај начин се омогућава бољи увид у модел квалитета софтвера, софтверске метрике и проблеме које софтверске метрике решавају. Опис насталог проблема најчешће садржи примере усаглашеног и неусаглашеног програмског кода, што у значајној мери доприноси процесу усвајања знања која се односе на модел квалитета софтвера, софтверске метрике и софтверске технологије које се примењују у процесу развоја софтвера. Окружења за развој софтвера могу дати сугестије које се односе на исправку уочене неусаглашености са дефинисаним моделом квалитета софтвера. У том смислу окружења за развој софтвера могу понудити брза решења за исправку неусаглашеног програмског кода (енг. quick fixes) којима се, уз коришћење најбољих пракси, проблем (програмски код који садржи нарушавања стандарда квалитета софтвера) аутоматски трансформише у решење проблема (програмски код који је усаглашен са стандардима квалитета софтвера). Тако се стичу непосредна, практична сазнања како настали проблем треба разрешити. На тај начин се уочене грешке у процесу развоја софтвера могу брзо и лако исправити.

У наставку поглавља биће описане активности које чине саставни део SilabQOSS методе за побољшање објектно-оријентисаних софтверских система. Затим ће бити приказани артифакти који настају и који се користе у оквиру предложене методе.

7.2. Активности предложене методе

У претходном одељку описана је SilabQOSS метода за побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера. Метода разматра стандарде квалитета софтвера, софтверске метрике и алате за статичку анализу квалитета софтвера у контексту побољшања објектно-оријентисаних софтверских система. При томе, идентификовани су различити механизми за побољшање објектно-оријентисаних софтверских система који се могу користити у процесу развоја софтвера.

На основу изложеног могуће је дефинисати и активности SilabQOSS методе за побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера:

1. **Упознавање са објектно-оријентисаним концептима који се користе у процесу развоја софтверских система**

   Концепти објектно-оријентисаног развоја представљају основне градивне елементе који су саставни део објектно-оријентисаних софтверских система. Због тога је њихово познавање идентификовано као прва активност предложене методе. У том смислу потребно је познавање класа и њихових чланица (атрибута и метода), интерфејса и апстрактних класа, наслеђивања, полиморфизма, компатибилности објектних типова и касног повезивања.

   С обзиром да је ово прва активност предложене методе, не постоје предуслови за њено извршавање.
2. **Упознавање са стандардима и моделима за евалуацију квалитета софтвера**

   Стандардима квалитета софтвера даје се спецификација модела квалитета софтвера. С друге стране, модел квалитета софтвера садржи атрибуте квалитета софтвера чијом се оценом одређује квалитет посматраног софтверског система. Такође, атрибути

квалитета софтвера повезани су са софтверским метрикама. У том контексту познавање стандарда квалитета софтвера и модела квалитета софтвера представља неопходан услов за извршавање евалуације квалитета софтверских система.

За ову активност предложене методе не постоје предуслови за њено извршавање.

3. **Упознавање са софтверским метрикама за евалуацију квалитета софтвера**

   Софтверске метрике се оперативно користе у процесу евалуације квалитета софтвера. Оне представљају формализоване мере којима се врши оцена атрибута квалитета софтвера. С друге стране, на основу атрибута квалитета софтвера могуће је извршити оцену квалитета читавог софтверског система.

   У том смислу је потребно добро познавање математичког модела софтверске метрике, мерне скале софтверске метрике, као и добро разумевање и тумачење (тј. интерпретација) вредности које софтверска метрика мери. Софтверске метрике су директно повезане са алатима за статичку анализу квалитета софтвера и са механизмима за побољшање објектно-оријентисаних софтверских система.

   С обзиром на директну повезаност софтверских метрика са моделом квалитета софтвера, предуслов за извршавање ове активности је познавање стандарда и модела квалитета софтвера.

4. **Упознавање са алатима за статичку анализу квалитета софтвера**

   Алати за статичку анализу квалитета софтвера се непосредно користе за евалуацију квалитета софтверског система. Алати могу обезбедити подршку за неки стандард квалитета софтвера или могу бити засновани на принципима неког стандарда квалитета софтвера. С друге стране, за оперативно мерење нивоа квалитета софтвера алати користе софтверске метрике. Због тога је предуслов за извршавање

ове активности познавање софтверских метрика за евалуацију квалитета софтвера.

5. **Упознавање са механизмима за побољшање објектно-оријентисаних софтверских система**

   Механизми за побољшање објектно-оријентисаних софтверских система представљају градивне елементе који се користе у процесу развоја софтвера. Као механизми побољшања објектно-оријентисаних софтверских система идентификовани су принципи, стратегије и патерни пројектовања софтвера. Такође, као механизам побољшања идентификоване су и методе развоја софтвера које се ослањају на принципе, стратегије и патерне пројектовања софтвера. Механизми побољшања представљају најбоље праксе које се користе у процесу објектно-оријентисаног развоја софтвера.

   Посматрани механизми су, такође, непосредно повезани са софтверским метрикама: њиховом применом се долази до повећања или смањења вредности посматраних софтверских метрика. С друге стране, применом механизама побољшања објектно-оријентисаних софтверских система добијају се софтверски системи који су једноставни за развој, надоградњу и одржавање.

   Приликом извршавања ове активности користе се објектно-оријентисани концепти: класа и њене чланице, наслеђивање, апстрактна класа, интерфејс, полиморфизам, компатибилност објектних типова и касно повезивање. Због тога познавање објектно-оријентисаних концепата представља предуслов за извршавање ове активности.

6. **Примена механизма побољшања објектно-оријентисаних софтверских система**

   У оквиру претходне активности је објашњено да применом механизама побољшања објектно-оријентисаних софтверских система последично долази до промена вредности посматраних софтверских

метрика. У том смислу се у процесу развоја софтвера примењују ови механизми како би атрибути квалитета софтвера и софтверске метрике били задовољени. У зависности од софтверског система који се развија могуће је применити један или више механизама побољшања софтверских система. Након њихове примене могуће је спровести статичку анализу и извршити евалуацију квалитета посматраног софтверског система.

Познавање механизама побољшања објектно-оријентисаних софтверских система представља предуслов за извршавање посматране активности.

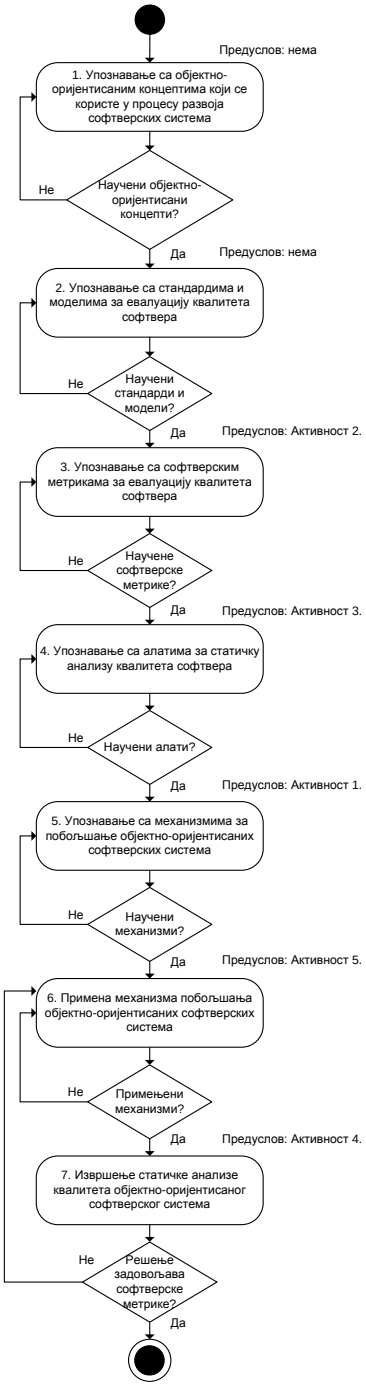
7. **Извршење статичке анализе квалитета објектно-оријентисаног софтверског система**

   У оквиру ове активности користе се алати за статичку анализу квалитета софтвера који су засновани на стандардима квалитета софтвера и софтверским метрикама.

   Алати за статичку анализу оперативно утврђују вредности софтверских метрика и мере задовољење атрибута квалитета посматраног софтверског система. Другим речима, након извршене анализе алати на одговарајући начин приказују извештаје о вредностима метрика и задовољењу атрибута квалитета софтвера. На основу извршене статичке анализе могуће је оценити квалитет посматраног софтверског система. Уколико су софтверске метрике и атрибути квалитета у задовољавајућим границама, активности методе се завршавају, а у супротном се морају предузети одговарајуће акције у циљу побољшања квалитета посматраног софтверског система. У том смислу се прелази на претходну активност, тј. поново се спроводи активност примене механизама побољшања објектно-оријентисаних софтверских система.

   Познавање алата за статичку анализу квалитета софтвера је предуслов за извршавање посматране активности.

На наредној слици (Слика 96) приказане су активности SilabQOSS методе за побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера. Приказ активности SilabQOSS методе реализован је коришћењем UML дијаграма активности.

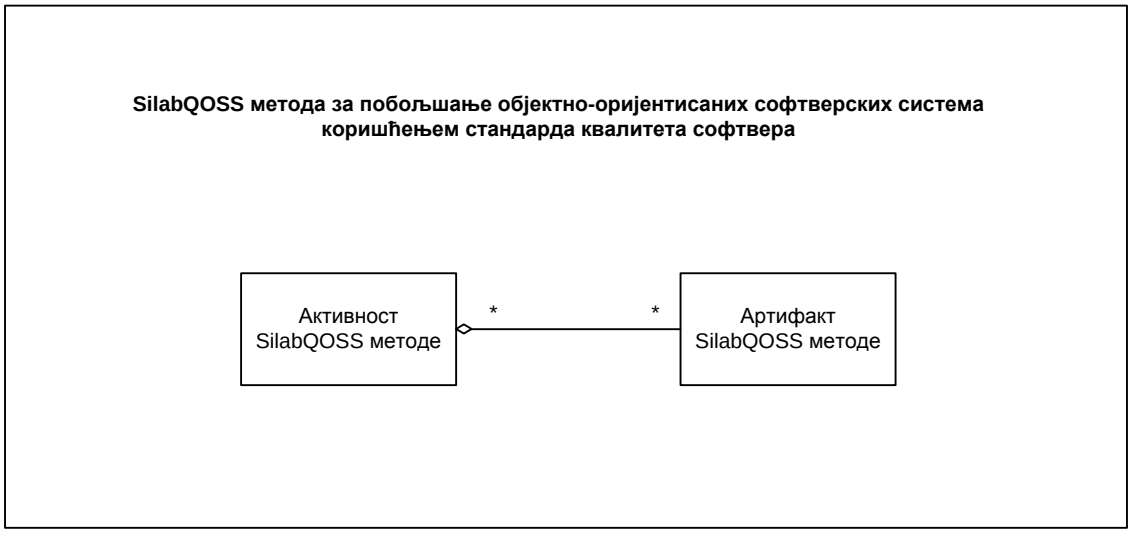


Слика 96. Активности SilabQOSS методе за побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера

У наставку поглавља ће бити приказани артифакти који настају и који се користе у оквиру предложене методе.

7.3. Артифакти предложене методе

Са активностима SilabQOSS методе за побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера непосредно су повезани артифакти предложене методе. Наиме, артифакти методе су потребни за реализацију посматране активности методе или настају у поступку извршавања посматране активности методе. У том смислу је могуће успоставити везу између активности и артифаката предложене методе. Једна активност може бити непосредно повезана са више артифаката. С друге стране, један артифакт методе може бити непосредно повезан са више активности предложене методе. Однос између артифаката и активности SilabQOSS методе за побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера приказан је на наредној слици (Слика 97).



Слика 97. Однос између артифаката и активности SilabQOSS методе за побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера

На основу изложеног се може закључити да су артифакти методе непосредно повезани са активностима методе. Због тога се у наставку најпре наводе активности, а затим се даје објашњење који су артифакти повезани са

посматраном активношћу. Узимајући у обзир опис и активности SilabQOSS методе за побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера, као и везу артифаката са активностима, разликују се следећи артифакти предложене методе:

Активност 1. Упознавање са објектно-оријентисаним концептима који се користе у процесу развоја софтверских система

У оквиру ове активности врши се упознавање са основним концептима објектно-оријентисаног развоја софтвера. У том смислу, као артифакти ове активности идентификоване су класе и њихове чланице (атрибути и методе), интерфејси, апстрактне класе, полиморфизам, компатибилност објектних типова, касно повезивање. Ови артифакти представљају градивне блокове који се користе у процесу развоја објектно-оријентисаних софтверских система тако да њихово познавање представља неопходан услов за даљи развој софтверског система

Активност 2. Упознавање са стандардима и моделима за евалуацију квалитета софтвера

У оквиру ове активности врши се селекција стандарда квалитета софтвера који ће се користити у процесу евалуације квалитета. На тај начин се даје спецификација модела квалитета софтвера, односно атрибута квалитета софтвера који представљају саставни део модела квалитета софтвера.

У том смислу, као артифакт ове активности идентификован је ISO/IEC 9126 стандард квалитета софтвера. Овај стандард квалитета софтвера усмерен је ка дефинисању квалитета софтверског производа. У оквиру овог стандарда идентификован је ISO/IEC 9126-1 модел квалитета софтвера који представља саставни део поменутог стандарда и представља артифакт посматране активности.

С друге стране, као артифакт ове активности идентификован је ISO/IEC 25000 стандард квалитета софтвера. Овим стандардом се дефинишу захтеви

за квалитет и евалуацију система и софтвера. У оквиру овог стандарда идентификован је ISO/IEC 25010 модел квалитета софтверског производа. Посматрани модел представља саставни део поменутог стандарда и представља артифакт посматране активности.

Активност 3. Упознавање са софтверским метрикама за евалуацију квалитета софтвера

Софтверске метрике представљају квантитативне мере којима се врши оцена атрибута квалитета софтвера. С друге стране, на основу оцене атрибута квалитета софтвера могуће је оценити квалитет посматраног софтверског система. Као артифакти ове активности идентификоване су софтверске метрике којима се врши евалуација квалитета софтверског производа:

- *Циклична сложеност* - Усмерена је на методе које представљају саставни део класе. Методама се дефинише понашање и овом метриком се рачуна сложеност примењеног алгоритма у посматраној методи.
- *Сложеност пондерисаних метода* - Ова метрика је такође усмерена на методе класе. Метрика *Сложеност пондерисаних метода* се дефинише као сума сложености метода класе. При томе, најчешће се као мера сложености метода класе користи метрика *Циклична сложеност*. У том смислу су поменуте метрике непосредно повезане и користе се заједно у процесу евалуације квалитета софтвера.
- *Број пондерисаних метода класе* - Метрика је усмерена на рачунање броја пондерисаних метода у посматраној класи. Користи се заједно са метрикама *Циклична сложеност* и *Сложеност пондерисаних метода*.
- *Број одговора класе* - Метрика је усмерена на класу и дефинише се као скуп одговора класе, односно као број свих метода које могу бити позване као одговор на поруку објекта класе.

- *Недостатак кохезивности метода у класи* - Кохезија је усмерена на атрибуте и методе класе. За класу се каже да је кохезивна уколико се њене методе извршавају над истим скупом атрибута. Посматрана метрика мери недостатак кохезивности метода у класи. У том смислу је потребно да кохезија унутар класе буде висока, односно да недостатак кохезивности метода у класи буде низак.
- *Повезаност објеката* - Метрика је усмерена на однос између класа (разматра се однос између посматраних класа, њихових метода и њихових атрибута). У том смислу се може рећи да су класе међусобно повезане уколико методе једне класе користе атрибуте или методе друге класе.
- Дубина стабла наслеђивања - Ова софтверска метрика је усмерена на класу и на одређивање њене позиције у хијерархији наслеђивања, односно на одређивање максималног броја предака посматране класе у хијерархији наслеђивања.
- Број подкласа - Ова софтверска метрика усмерена је на класу и одређивање броја њених непосредних потомака. Посматрана софтверска метрика се користи заједно са метриком *Дубина стабла наслеђивања*.
- Број наредби у методи - Ова софтверска метрика усмерена је на методе класе, односно на одређивање броја наредби у посматраној методи класе.
- Метрика стабилности софтвера - Ова софтверска метрика разматра стабилност у контексту односа између пакета посматраног софтверског система.

Активност 4. Упознавање са алатима за статичку анализу квалитета софтвера

Алати за статичку анализу квалитета софтвера засновани су на стандардима квалитета софтвера и користе софтверске метрике у процесу евалуације

квалитета софтвера. Као артифакти ове активности идентификовани су SilabMetrics, SonarQube и Swat4J алати за статичку анализу квалитета софтвера. Сви посматрани алати подржавају ISO/IEC 9126 стандард квалитета софтвера и усмерени су на евалуацију квалитета софтверског производа. Такође, посматрани алати имају подршку за софтверске метрике *Циклична сложеност*, *Сложеност пондерисаних метода*, *Број пондерисаних метода класе*, *Број одговора класе*, *Недостатак кохезивности метода у класи*, *Повезаност објеката*, *Дубина стабла наслеђивања*, *Број подкласа*, *Број наредби у методи* и *Метрика стабилности софтвера*.

Активност 5. Упознавање са механизмима за побољшање објектно-оријентисаних софтверских система

Механизми за побољшање објектно-оријентисаних софтверских система представљају градивне елементе који се користе у процесу пројектовања софтвера. Идентификовани су следећи механизми за побољшање објектно-оријентисаних софтверских система:

- Општи принципи пројектовања софтвера - Представљају основне принципе пројектовања. Њихова заједничка карактеристика је да се могу користити приликом пројектовања свих софтверских система. Идентификовани су следећи општи принципи пројектовања софтвера: апстракција, кохезија и повезаност, декомпозиција и модуларизација, учаурење и сакривање информација и одвајање интерфејса и имплементације.
- Принципи објектно-оријентисаног пројектовања софтвера - Користе се приликом објектно-оријентисаног програмирања и пројектовања. Идентификовани су следећи принципи објектно-оријентисаног пројектовања: принцип једне одговорности, принцип отворено-затворено, Лисков принцип замене, принцип инверзије зависности, принцип додавања зависности и принцип издвајања интерфејса.

- Стратегије пројектовања софтвера - Одређују стратешки приступ процесу пројектовања софтвера. Уочене су следеће стратегије пројектовања софтвера: подели и победи, с врха на доле и одоздо на горе.
- Патерни пројектовања софтвера - Одређују решење неког проблема у неком контексту. Идентификовани су следећи патерни пројектовања софтвера: патерни за креирање објекта, структурни патерни и патерни понашања [Gamma94][Vlajic14b].
- Методе развоја софтвера - Методом развоја софтвера описује се процес развоја софтвера. У том смислу је као метода развоја софтвера коришћена упрошћена Ларманова метода развоја софтвера која се ослања на претходно поменуте принципе, стратегије и патерне пројектовања софтвера.

Механизми побољшања објектно-оријентисаних софтверских система користе објектно-оријентисане концепте: класе и њихове чланице (атрибути и методе), интерфејсе, апстрактне класе, полиморфизам, компатибилност објектних типова и касно повезивање метода. Због тога су објектно-оријентисани концепти идентификовани као артифакти посматране активности.

Активност 6. Примена механизма побољшања објектно-оријентисаних софтверских система

У оквиру ове активности врши се непосредна примена механизама побољшања на посматраном објектно-оријентисаном софтверском систему. У том смислу сви претходно поменути механизми побољшања објектно-оријентисаних софтверских система (принципи, стратегије, патерни пројектовања софтвера и методе развоја софтвера) представљају артифакте ове активности. За реализацију ове активности непосредно се користе објектно-оријентисане концепти: класе и њихове чланице (атрибути и методе), интерфејсе, апстрактне класе, полиморфизам, компатибилност

објектних типова и касно повезивање метода. Због тога су и објектно-оријентисани концепти идентификовани као артифакти посматране активности.

С друге стране, као резултат примене механизама побољшања објектно-оријентисаних софтверских система добија се програмски код посматраног софтверског система који такође представља артифакт ове активности.

Активност 7. Извршење статичке анализе квалитета објектно-оријентисаног софтверског система

Ова активност подразумева употребу алата за статичку анализу квалитета софтвера. У том смислу су идентификовани артифакти SilabMetrics, SonarQube и Swat4J.

Алати су засновани на стандардима квалитета софтвера и софтверским метрикама. У том смислу су као артифакти посматране активности идентификовани ISO/IEC 9126 и ISO/IEC 25000 стандарди квалитета софтвера, односно ISO/IEC 9126-1 и ISO/IEC 25010 модели квалитета софтвера.

Такође, као артифакти активности идентификоване су и софтверске метрике *Циклична сложеност*, *Сложеност пондерисаних метода*, *Број пондерисаних метода класе*, *Број одговора класе*, *Недостатак кохезивности метода у класи*, *Повезаност објеката*, *Дубина стабла наслеђивања*, *Број подкласа*, *Број наредби у методи* и *Метрика стабилности софтвера*.

Софтверске метрике се оперативно користе за евалуацију квалитета посматраног софтверског система. У том смислу је као артифакт активности идентификован програмски код софтверског система. На основу извршене статичке анализе добијају се вредности софтверских метрика на основу чега је могуће даље доношење одлука о квалитету посматраног софтверског система.

На основу изложеног се види непосредна повезаност активности и артифаката предложене методе. У наредној табели (Табела 76) дат је упоредни преглед активности и артифаката SilabQOSS методе за побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера.

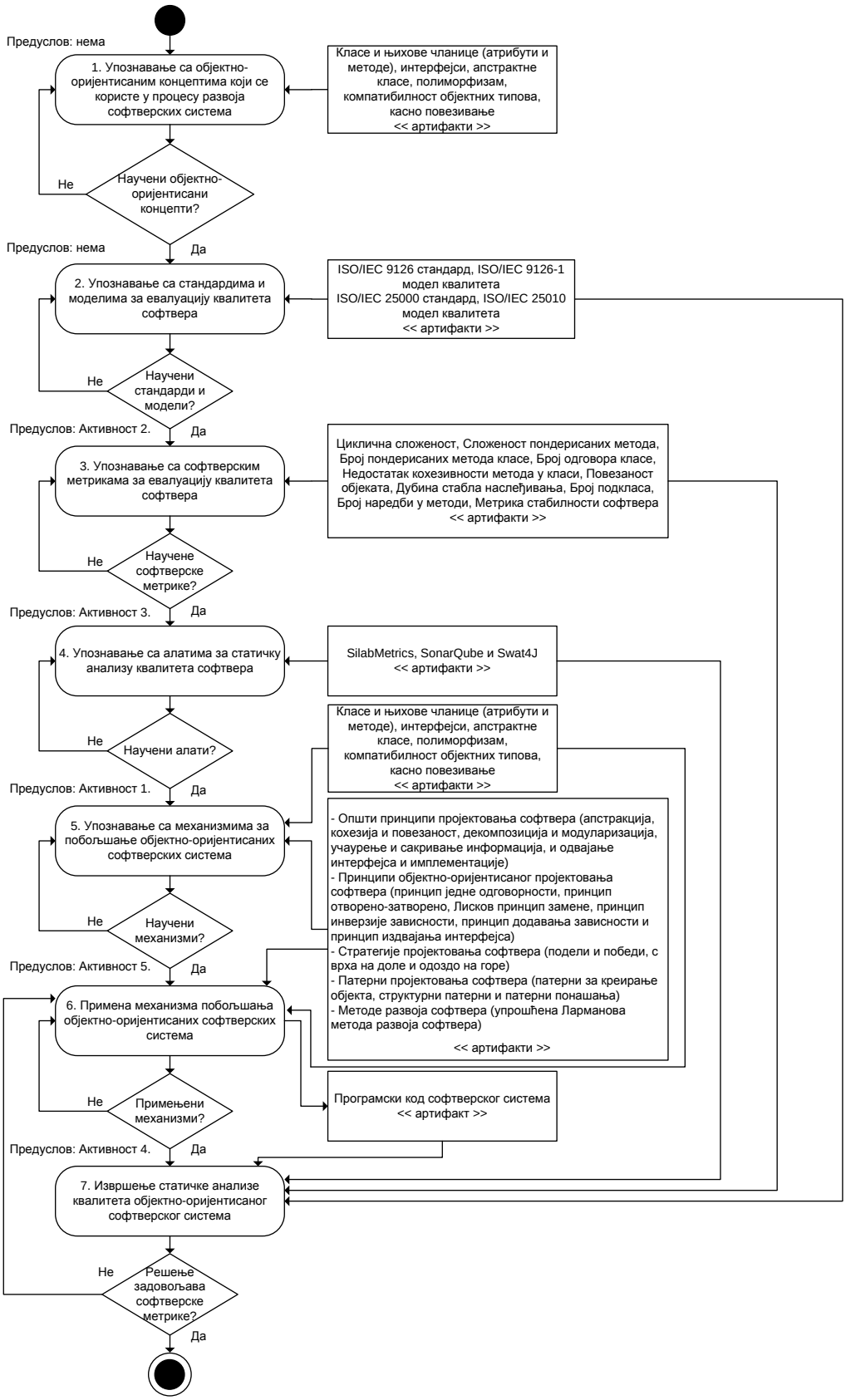
Табела 76. Упоредни преглед активности и артифаката SilabQOSS методе за побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера

Активност	**Артифакти**
1. Упознавање са објектно-оријентисаним концептима који се користе у процесу развоја софтверских система	Класе и њихове чланице (атрибути и методе), интерфејси, апстрактне класе, полиморфизам, компатибилност објектних типова, касно повезивање
2. Упознавање са стандардима и моделима за евалуацију квалитета софтвера	- ISO/IEC 9126 стандард, ISO/IEC 9126-1 модел квалитета - ISO/IEC 25000 стандард, ISO/IEC 25010 модел квалитета
3. Упознавање са софтверским метрикама за евалуацију квалитета софтвера	Софтверске метрике *Циклична сложеност*, *Сложеност пондерисаних метода*, *Број пондерисаних метода класе*, *Број одговора класе*, *Недостатак кохезивности метода у класи*, *Повезаност објеката*, *Дубина стабла наслеђивања*, *Број подкласа*, *Број наредби у методи* и *Метрика стабилности софтвера*
4. Упознавање са алатима за статичку анализу квалитета софтвера	SilabMetrics, SonarQube и Swat4J
5. Упознавање са механизмима за побољшање објектно-оријентисаних софтверских система	- Објектно-оријентисани концепти (класе и њихове чланице, интерфејси, апстрактне класе, полиморфизам, компатибилност објектних типова, касно повезивање) - Општи принципи пројектовања софтвера (апстракција, кохезија и повезаност, декомпозиција и

Активност	**Артифакти**
	модуларизација, учаурење и сакривање информација, и одвајање интерфејса и имплементације) - Принципи објектно-оријентисаног пројектовања софтвера (принцип једне одговорности, принцип отворено-затворено, Лисков принцип замене, принцип инверзије зависности, принцип додавања зависности и принцип издвајања интерфејса) - Стратегије пројектовања софтвера (подели и победи, с врха на доле и одоздо на горе) - Патерни пројектовања софтвера (патерни за креирање објекта, структурни патерни и патерни понашања) - Методе развоја софтвера (упрошћена Ларманова метода развоја софтвера)
6. Примена механизма побољшања објектно-оријентисаних софтверских система	- Објектно-оријентисани концепти (класе и њихове чланице, интерфејси, апстрактне класе, полиморфизам, компатибилност објектних типова, касно повезивање) - Општи принципи пројектовања софтвера (апстракција, кохезија и повезаност, декомпозиција и модуларизација, учаурење и сакривање информација, и одвајање интерфејса и имплементације) - Принципи објектно-оријентисаног пројектовања софтвера (принцип једне одговорности, принцип отворено-затворено, Лисков принцип замене, принцип инверзије зависности, принцип додавања зависности и принцип издвајања интерфејса) - Стратегије пројектовања софтвера (подели и победи, с врха на доле и одоздо на горе) - Патерни пројектовања софтвера (патерни за креирање објекта, структурни патерни и патерни понашања)

Активност	**Артифакти**
	- Методе развоја софтвера (упрошћена Ларманова метода развоја софтвера) - Програмски код софтверског система
7. Извршење статичке анализе квалитета објектно-оријентисаног софтверског система	- Алати SilabMetrics, SonarQube и Swat4J - ISO/IEC 9126 стандард, ISO/IEC 9126-1 модел квалитета, - ISO/IEC 25000 стандард, ISO/IEC 25010 модел квалитета - Софтверске метрике *Циклична сложеност*, *Сложеност пондерисаних метода*, *Број пондерисаних метода класе*, *Број одговора класе*, *Недостатак кохезивности метода у класи*, *Повезаност објеката*, *Дубина стабла наслеђивања*, *Број подкласа*, *Број наредби у методи* и *Метрика стабилности софтвера* - Програмски код софтверског система

На наредној слици (Слика 98) дат је графички приказ активности и артифаката SilabQOSS методе за побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера. Приказ активности и артифаката SilabQOSS методе реализован је коришћењем UML дијаграма активности.



Слика 98. Графички приказ активности и артифаката SilabQOSS методе за побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера

Из изложеног се може закључити да су активности и артифакти предложене методе повезани. У ток контексту, за реализацију једне активности се користи више артифаката. С друге стране, један артифакт може бити повезан са више активности.

7.4. Лин приступ у процесу примене SilabQOSS методе за побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера

Развојем индустрије створила се потреба за другачијим методама и приступима у производњи, а у циљу повећања нивоа квалитета и континуалног побољшања процеса производње. Лин представља приступ настао у производњи који заступа становиште да је најбитнији циљ производње стварање вредности за крајњег купца. Свако трошење средстава за било који други циљ самим тим представља расипање (енг. waste) и треба га елиминисати [Womack10]. Са аспекта купца, вредност представља било коју акцију или процес који је купац спреман да плати (купац свакако није спреман да плати губитке и проблеме који су настали у процесу производње). Лин приступ је у значајној мери утицао на постојеће приступе и данас се, поред примене у производњи, с великим успехом примењује у администрацији, пружању услугама, образовању, софтверском инжењерству итд.

Данас су доминантне и у пракси често примењиване агилне методе развоја софтвера (нпр. Extreme Programming, Feature Driven Development, Crystal Clear Methods, Scrum, Lean развој софтвера итд.) [Abrahamsson02]. Лин приступ представља покушај трансфера позитивних искустава из класичне индустрије у индустрију развоја софтвера. Он је првенствено усмерен на тим који непосредно ради на развоју софтвера и промовише развој софтвера у више мањих итерација (најчешће од једне до четири недеље).

Основни принцип који се примењује у Лин развоју софтвера јесте елиминација расипања (отпада). Аутори Marry и Tom Poppendieck у књизи

[Poppendieck03] дефинишу седам типова расипања у процесу развоја софтвера и упоређују их са расипањима у процесу производње, што је и приказано у наредној табели (Табела 77).

Табела 77. Седам врста расипања у производњи и седам врста расипања у софтверском инжењерству [Poppendieck03]

Седам врста расипања у производњи	**Седам врста расипања у софтверском инжењерству**
Инвентар	Делимично завршени послови
Додатна обрада	Додатни процеси
Хиперпродукција	Додатне функционалности
Транспорт	Пребацивање са задатка на задатак
Чекање	Чекање
Покрети	Покрети
Дефекти	Дефекти

Из приказаног се може закључити да отпад у лин производњи софтвера представља све оно што не доноси вредност производу на начин на који вредност софтвера доживљава крајњи корисник:

- Делимично завршени послови односе се на недовршене послове у процесу развоја софтвера. У том смислу тим који развија софтвер не може бити сигуран да ли ће софтвер радити како треба.
- Додатни процеси односе се на процесе који троше расположиве ресурсе и повећавају време одговора на постављене захтеве. У том смислу би тим који развија софтвер требао да се сконцентрише на процесе који доносе вредност финалном софтверском производу.
- Додатне функционалности односе се на додавање функционалности у софтвер, при чему посматране функционалности нису тражене. С обзиром да свака линија програмског кода повећава величину софтверског система и представља потенцијалну тачку у којој може

настати грешка, додатне функционалности представљају велико расипање.

- Пребацивање са задатка на задатак односи се на распоређивање запослених на више пројеката. Уколико запослени припада већем броју тимова (или ради на више пројеката) потребно је више времена како би се пребацио са једног на други задатак.
- Чекање се односи на кашњење у процесу развоја софтвера (нпр. кашњење у почетку пројекта, кашњење у тестирању и сл.). Свако кашњење повећава време потребно за завршетак развоја посматраног софтверског система.
- Покрети се односе на акције које је потребно извршити како би се обавио додељени задатак (нпр. уколико члан тима за развој софтвера има питање, колико покрета му је потребно да би дошао до одговора).
- Дефекти се односе на грешке у софтверском производу. Они могу имати значајан утицај на развој и одржавање посматраног софтверског система. Због тога је важно њихово рано уочавање и елиминисање.

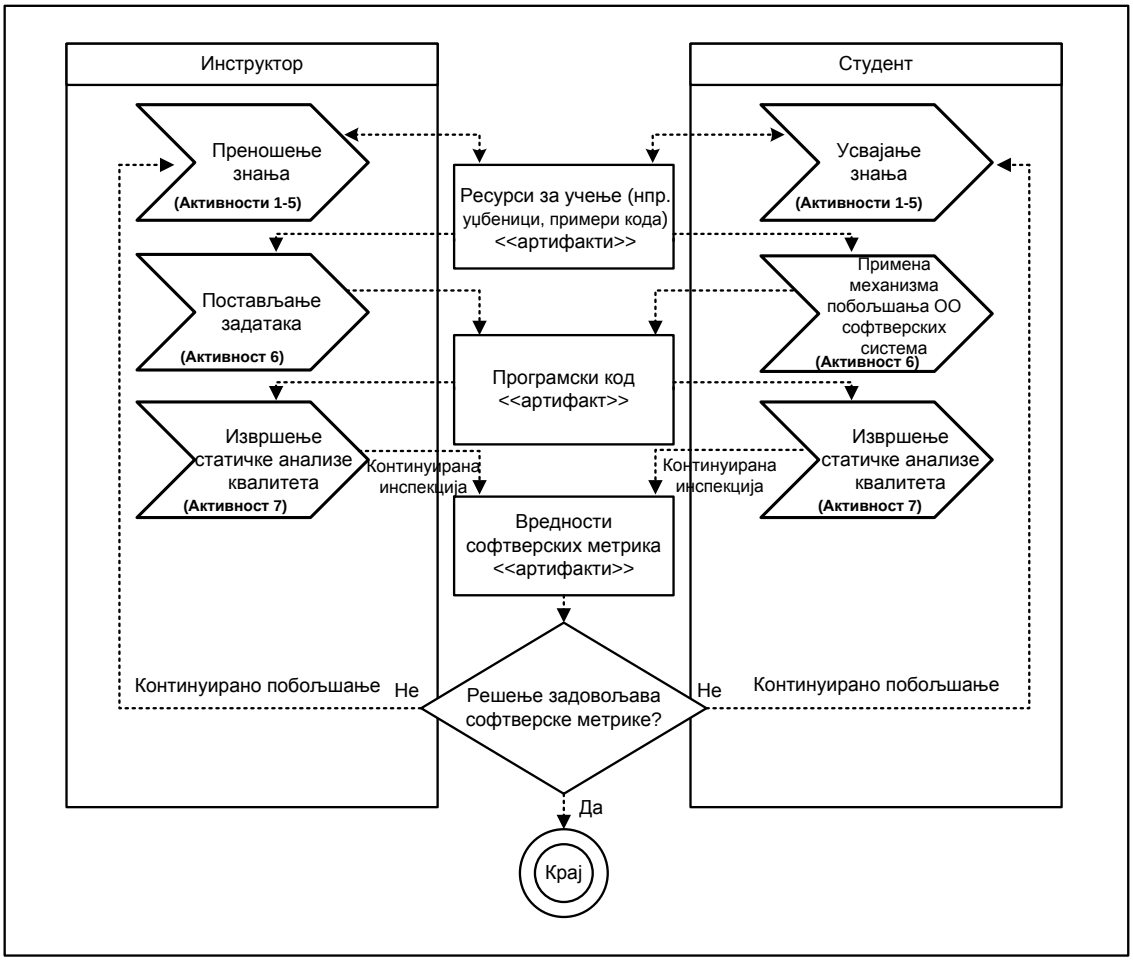
Јасно је да нпр. "Делимично завршени послови" или "Дефекти" не представљају вредност софтвера за крајњег корисника (та расипања мора да отклони тим који развија софтвер). У том смислу, први корак у елиминисању отпада је његово уочавање, што представља предуслов за развој алата и метода за његово отклањање [Poppendieck03]. Лин приступ у развоју софтвера базира се на сталном побољшању (енг. continious improvement) људи, процеса и технологије, као и предузимању мера како би се та побољшања остварила. На тај начин омогућава се исправљање уочених неусаглашености у раним фазама развоја софтвера: уколико се дефекат открије раније потребно је мање времена, напора, и мањи је трошак његове исправке.

Уколико се разматра SilabQOSS метода, претпоставка је да су у процесу примене методе учесници у потпуности посвећени побољшању посматраног објектно-оријентисаног софтверског система, тако да не постоје расипања "Додатни процеси" и "Пребацивање са задатка на задатак". С обзиром да су активности и артифакти SilabQOSS методе прецизно дефинисани, не постоје ни расипања "Додатне функционалности" и "Чекање". У том контексту, SilabQOSS метода за побољшање објектно-оријентисаних софтверских система у обзир узима расипања "Делимично завршени послови", "Покрети" и "Дефекти" као губитке које треба елиминисати.

С обзиром на активности SilabQOSS методе, пренос и усвајање знања који се односе на Активности 1-5 (упознавање са објектно-оријентисаним концептима, стандардима и моделима квалитета софтвера, софтверским метрикама, алатима за статичку анализу квалитета софтвера и механизмима побољшања објектно-оријентисаних софтверских система) могуће је извршити уз помоћ инструктора. У том смислу разликују се две улоге: инструктор и студент.

Другим речима, инструктор може водити процес примене SilabQOSS методе тако што ће преносити знање студенту и постављати му задатке који се односе на примену механизма побољшања објектно-оријентисаних софтверских система (дефинисано је кроз Активност 6.). С друге стране, инструктор и студент могу вршити статичку анализу квалитета објектно-оријентисаног софтверског система (дефинисано је кроз Активност 7.). На тај начин сви учесници у процесу примене SilabQOSS методе могу добити повратну информацију о квалитету објектно-оријентисаног софтверског система на основу чега је могуће предузети акције у циљу побољшања истог. Посматрани поступак се може представити коришћењем перспективе улога/задаци [Silva07], у оквиру које се задаци извршавају од стране два учесника: инструктора и студента. У том смислу је могуће уочити различите задатке које врше учесници, везу задатака са активностима SilabQOSS методе, као и различите артифакте који се користе у процесу реализације

посматраног задатка. Последично се, кроз континуирану инспекцију и континуирано побољшање, обезбеђује побољшање квалитета објектно-оријентисаног софтверског система, што је и приказано на наредној слици (Слика 99).



Слика 99. Лин приступ у процесу примене SilabQOSS методе за побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера [Milic17]

Раније је напоменуто да је први корак у елиминисању расипања његово уочавање. С обзиром да се под расипањем сматрају "Делимично завршени послови", "Покрети" и "Дефекти", примена Лин приступа у оквиру SilabQOSS методе за побољшање објектно-оријентисаних софтверских система укључује следеће концепте [Milic17]:

- Континуирану инспекцију - Односи се на извршење статичке анализе квалитета софтвера, али и на инспекцију читавог процеса преноса и

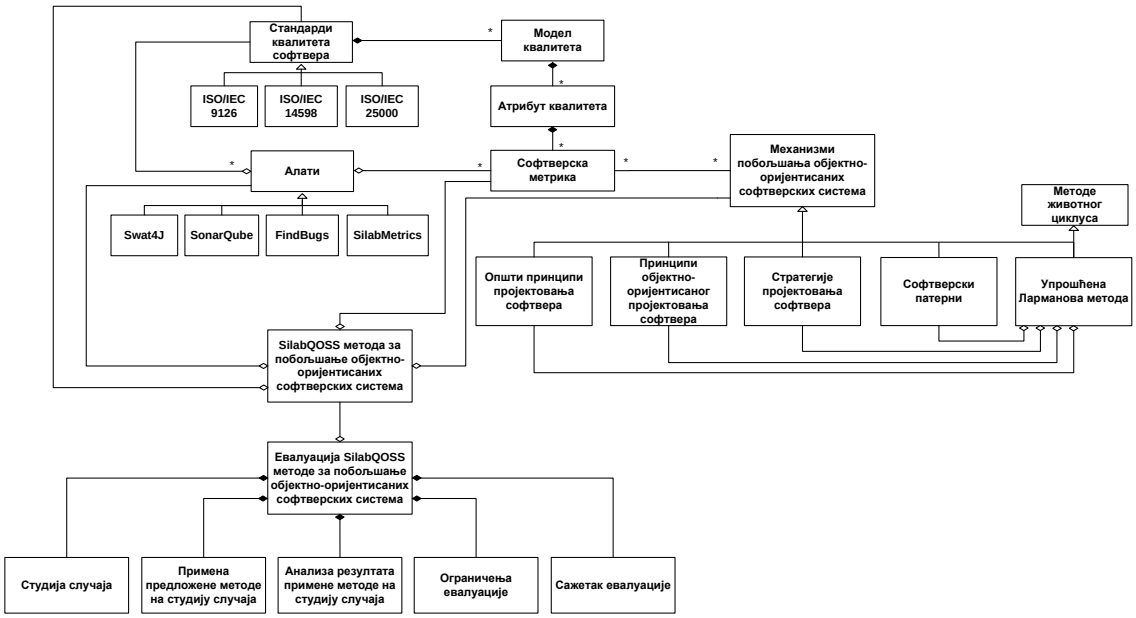
усвајања знања. У том смислу је могуће извршити анализу квалитета објектно-оријентисаног софтверског система, уочити недостатке у квалитету софтверског система и процесу преноса и усвајања знања, и креирати основу за побољшање истих.

- Континуирано побољшање - Односи се на примену механизма побољшања објектно-оријентисаних софтверских система, али и на побољшање читавог процеса преноса и усвајања знања. Након извршења статичке анализе квалитета добијају се вредности софтверских метрика. Уколико посматрано решење не задовољава софтверске метрике добија се повратна информација на основу чега је могуће побољшати процес преноса и усвајања знања. С друге стране, на основу вредности метрика могуће је извршити исправке (у смислу примене механизма побољшања објектно-оријентисаних софтверских система) и добити нови програмски код објектно-оријентисаног софтверског система. Ново решење је такође могуће анализирати и побољшавати, у циљу остваривања жељеног нивоа квалитета софтвера.

У наредном поглављу врши се евалуација предложене SilabQOSS методе за побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера.

8. Евалуација предложене SilabQOSS методе за побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера

У оквиру овог поглавља извршена је евалуација предложене SilabQOSS методе за побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера. У том смислу, у првом делу поглавља биће представљен опис студије случаја која ће бити реализована. У другом делу поглавља биће објашњена примена SilabQOSS методе на студију случаја, док ће у трећем делу бити дата анализа резултата примене SilabQOSS методе на студију случаја. У четвртом делу поглавља биће идентификована ограничења евалуације SilabQOSS методе, док се у петом делу поглавља даје сажетак евалуације. Концептуални приказ поглавља дат је на наредној слици (Слика 100).



Слика 100. Концептуални приказ евалуације SilabQOSS методе за побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера

На почетку евалуације дефинисана су кључна истраживачка питања:

1. **Да ли се стандарди и модели квалитета софтвера могу применити у процесу побољшања објектно-оријентисаних софтверских система?**

   Стандарди квалитета софтвера усмерени су на евалуацију квалитета софтверског система, са циљем стварања високо-квалитетног софтвера. Другим речима, стандарди квалитета представљају основу на којој се заснива процес евалуације квалитета софтверског система. Стандардом квалитета софтвера дефинишу се модели квалитета софтвера (један или више модела квалитета софтвера), у смислу карактеристика и подкарактеристика квалитета софтвера (тј. атрибута квалитета софтвера).

   Као што је напоменуто у Поглављу 7, SilabQOSS метода посматра процес побољшања објектно-оријентисаних софтверских система у контексту стандарда квалитета софтвера. У том смислу, предлаже се коришћење ISO/IEC 9126 и ISO/IEC 25000 стандарда квалитета софтвера.

2. **Да ли се софтверске метрике могу применити у процесу побољшања објектно-оријентисаних софтверских система?**

   Са атрибутима квалитета софтвера непосредно су повезане софтверске метрике које мере ниво квалитета посматраног атрибута квалитета, на основу чега је могуће извршити оцену квалитета софтверског система.

   У оквиру SilabQOSS методе идентификоване су следеће софтверске метрике:

   - Циклична сложеност,
   - Сложеност пондерисаних метода,
   - Број пондерисаних метода класе,
   - Број одговора класе,
   - Недостатак кохезивности метода у класи,
   - Повезаност објеката,

- Дубина стабла наслеђивања,
- Број подкласа,
- Број наредби у методи и
- Метрика стабилности софтвера.

На основу извршене статичке анализе квалитета софтвера добијају се вредности софтверских метрика. Другим речима, добијају се повратне информације о квалитету посматраног објектно-оријентисаног софтверског система на основу чега је могуће извести закључке о квалитету посматраног софтверског система и предузети акције у циљу побољшања квалитета.

3. **Да ли се алати за статичку анализу квалитета софтвера могу применити у процесу побољшања објектно-оријентисаних софтверских система?**

   Алати за статичку анализу квалитета софтвера могу бити засновани на неком стандарду квалитета софтвера. У том смислу се може рећи да они обезбеђују практичну примену стандарда квалитета софтвера у процесу евалуације квалитета. Коришћењем алата за статичку анализу квалитета омогућен је брз увид и анализа квалитета софтверског система, уз приказ релевантних информација о квалитету софтверског система.

   У Поглављу 5 је разматрана могућност примене алата за статичку анализу квалитета софтвера у процесу развоја софтвера. Закључено је да њихова примена може донети вишеструке предности свим учесницима у процесу развоја софтвера, што ће последично допринети да софтверски системи који се развијају буду сагласни са дефинисаним нивоом квалитета софтвера. На основу тога се може закључити да су алати за статичку анализу квалитета комплементарни са стандардима квалитета софтвера. У том смислу, у оквиру SilabQOSS методе предлаже се коришћење SilabMetrics, SonarQube и Swat4J алата за статичку анализу квалитета софтвера.

4. **Да ли примена механизама побољшања објектно-оријентисаних софтверских система доводи до побољшања квалитета посматраног софтверског система?**

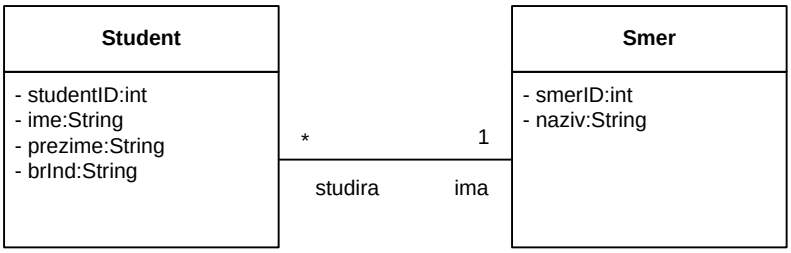
   На основу извршене статичке анализе квалитета могуће је уочити неусаглашености са дефинисаним моделом квалитета софтвера и применити механизме побољшања објектно-оријентисаних система. У оквиру SilabQOSS методе као механизми побољшања идентификовани су принципи, стратегије и патерни пројектовања софтвера. Такође, као механизам побољшања идентификоване су и методе развоја софтвера које се ослањају на принципе, стратегије и патерне пројектовања софтвера.

   Ови механизми су непосредно повезани са софтверским метрикама: њиховом применом долази до повећања или смањења вредности посматраних софтверских метрика. У том смислу, након примене механизама побољшања објектно-оријентисаних софтверских система могуће је спровести статичку анализу и извршити евалуацију квалитета посматраног софтверског система.

На основу изложеног се може приметити да су истраживачка питања усмерена на процес побољшања објектно-оријентисаних софтверских система, стандарде квалитета софтвера, софтверске метрике, алате за статичку анализу квалитета софтвера и механизме побољшања објектно-оријентисаних софтверских система. То су заправо градивни елементи SilabQOSS методе чија се евалуација треба извршити.

8.1. Студија случаја

У циљу евалуације SilabQOSS методе за побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера користи се студија случаја. У том смислу, на почетку је дефинисан концептуални (доменски) модел који се користи у процесу примене студије случаја. Концептуални модел приказан је на наредној слици (Слика 101).



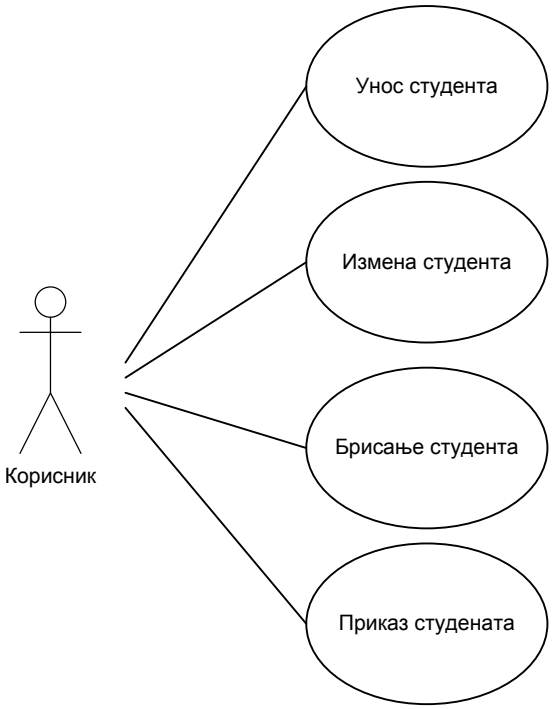
Слика 101. Концептуални (доменски) модел за реализацију студије случаја

Студија случаја подразумева материјализацију и дематеријализацију посматраних доменских објеката у оквиру релационог система за управљање базом података. Због тога се посматрани доменски модел преводи у следећи релациони модел:

Smer(SmerID, Naziv)

Student(StudentID, Ime, Prezime, BrInd, *SmerID*)

Преко концептуалног (доменског) модела и релационог модела дефинисана је структура софтверског система. С друге стране, понашање софтверског система дефинисано је преко случајева коришћења које треба реализовати. На наредној слици (Слика 102) дат је приказ модела случајева коришћења. На њему су приказани актори, случајеви коришћења и односи између њих. На основу приказаног уочава се да у систему постоји један актор (његов име је Корисник), као и да постоји четири случаја коришћења (Унос студента, Измена студента, Брисање студента и Приказ студената) који су међусобно независни.



Слика 102. Модел случајева коришћења за реализацију студије случаја

У процесу реализације посматраних захтева користи се објектно-оријентисани програмски језик Java, при чему се екранске форме реализацију коришћењем Java Swing технологије. Другим речима, студија случаја подразумева израду стандардне (десктоп) апликације.

8.2. Примена предложене методе на студију случаја

На основу дефинисаног концептуалног модела, релационог модела и случајева коришћења реализује се студија случаја. У том смислу, у наставку је дат опис учесника, процедуре и инструмената који су коришћени у процесу реализације студије случаја.

8.2.1. Учесници

У процесу реализације студије случаја учествовало је двадесет студената основних академских студија Факултета организационих наука (Универзитет у Београду) у школској 2016/17. години. Студенти су били подељени у две групе: експерименталну групу (10 студената) и контролну групу (10 студената).

Експерименталну групу чинили су студенти четврте године који су одслушали курс Пројектовање софтвера. У том смислу се може рећи да су студенти експерименталне групе упознати са објектно-оријентисаним концептима, упрошћеном Ларамановом методом развоја софтвера и софтверским патернима који се користе у процесу пројектовања софтвера.

Контролну групу су чинили студенти треће године који су упознати објектно-оријентисаним концептима у програмском језику Java.

Студенти су имали различите просечне оцене које су остварили у досадашњем школовању. Студенти су се добровољно пријавили да учествују у истраживању. На почетку истраживања студенти су информисани о циљевима истраживања, истраживачким питањима на које покушавамо на одговоримо, као и о процедури и инструментима који се користе у процесу реализације студије случаја.

8.2.2. Процедура

Обе групе студената имале су традиционалне лекције (предавања и лабораторијске вежбе). На лабораторијским вежбама су обрађени основни и напредни објектно-оријентисани концепти у програмском језику Java: класе и методе, наслеђивање, апстрактне класе, интерфејси, графички кориснички интерфејс и рад са базом података.

С друге стране, студенти експерименталне групе су имали додатно предавање у оквиру кога су упознати са SilabQOSS методом за побољшање објектно-оријентисаних система коришћењем стандарда квалитета софтвера. У том смислу студенти су упознати са SilabQOSS методом, као и са активностима и артифактима посматране методе.

Студенти експерименталне и контролне групе су имплементирали претходно описане случајеве коришћења на основу чега је извршена статичка анализа квалитета софтвера. Уз то, студенти експерименталне групе су попунили упитник који се односио на примену SilabQOSS методе за

побољшање објектно-оријентисаних система коришћењем стандарда квалитета софтвера. Приказ питања упитника дат је у наредном одељку.

У процесу развоја софтвера студенти су користили NetBeans развојно окружење и MySQL систем за управљање базом података. Сва потребна објашњења и инструкције студенти су добили од стране инструктора.

8.2.3. Инструменти

У процесу статичке анализе квалитета софтвера коришћени су SilabMetrics, SonarQube и Swat4J алати. У оквиру статичке анализе коришћене су следеће софтверске метрике: Циклична сложеност, Сложеност пондерисаних метода, Број пондерисаних метода класе, Број одговора класе, Недостатак кохезивности метода у класи, Повезаност објеката, Дубина стабла наслеђивања, Број подкласа, Број наредби у методи и Метрика стабилности софтвера. Резултати статичке анализе квалитета софтвера дати су у наредном одељку.

Такође, као техника за прикупљање података коришћен је упитник. Упитник је базиран на петостепеној Ликертовој скали са вредностима "*1 - потпуно се не слажем*" до "*5 - потпуно се слажем*". Упитник су попуњавали студенти експерименталне групе који су у процесу развоја софтвера применили SilabQOSS методу за побољшање објектно-оријентисаних система коришћењем стандарда квалитета софтвера. Приказ питања упитника дат је у наредној табели (Табела 78).

Табела 78. Питања упитника који је попунила експериментална група

РБ	**Питање**
1.	Да ли SilabQOSS метода доприноси бољем разумевању објектно-оријентисаних концепата?
2.	Да ли SilabQOSS метода доприноси бољем разумевању стандарда и модела квалитета софтвера?
3.	Да ли SilabQOSS метода доприноси бољем разумевању софтверских метрика?

РБ	Питање
4.	Да ли SilabQOSS метода доприноси бољем разумевању алата за статичку анализу квалитета софтвера?
5.	Да ли SilabQOSS метода доприноси бољем разумевању механизама побољшања објектно-оријентисаних софтверских система?
6.	Да ли интеграција алата за статичку анализу квалитета софтвера и развојног окружења побољшава процес развоја софтвера?
7.	Да ли коришћење алата за статичку анализу квалитета софтвера омогућава брже и лакше проналажење грешака у софтверском систему?
8.	Да ли се знање стечено на овај начин може искористити у процесу развоја других софтверских система?

Питања су логички организована у три групе:

- Прва група питања (Питања 1-5) усмерена су на SilabQOSS методу и њене везе са објектно-оријентисаним концептима, стандардима и моделима квалитета софтвера, софтверским метрикама, алатима за статичку анализу квалитета софтвера и механизмима побољшања објектно-оријентисаних софтверских система.
- Друга група питања (Питања 6-7) усмерена су на примену алата за статичку анализу квалитета софтвера у процесу развоја софтвера.
- Трећа група питања (Питање 8) усмерена је разматрање могућности примене стечених знања у процесу развоја других софтверских система.

Резултати упитника и анализа резултата упитника дати су у наредном одељку.

8.3. Анализа резултата примене предложене методе

Сви студенти који су учествовали у истраживању успешно су извршили све постављене задатке. У наставку одељка најпре се даје приказ и анализа

резултата статичке анализе квалитета софтвера. Након тога се даје приказ и анализа резултата упитника.

8.3.1. Приказ и анализа резултата статичке анализе квалитета софтвера

У наредној табели (Табела 79) дат је приказ вредности софтверских метрика за пројекте студената експерименталне групе. У табели су, такође, приказане аритметичка средина и стандардна девијација вредности за сваку софтверску метрику.

Табела 79. Вредности софтверских метрика за пројекте студената експерименталне групе

Метрика	**Студенти експерименталне групе**											
	C1	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**	**C8**	**C9**	**C10**	$\bar{x}$	**σ**
Циклична сложеност	1.19	1.17	1.32	1.23	1.26	1.19	1.19	1.18	1.20	1.24	1.22	0.05
Сложеност пондерисаних метода	6.94	7.00	8.24	7.35	7.22	7.12	6.58	6.89	6.89	7.00	7.12	0.44
Број пондерисаних метода класе	5.53	5.67	5.89	5.67	5.42	5.67	5.25	5.53	5.42	5.35	5.54	0.19
Број одговора класе	32.00	31.00	32.35	30.88	31.78	33.47	28.26	30.17	30.61	31.11	31.16	1.40
Недостатак кохезивности метода у класи	0.51	0.49	0.62	0.50	0.51	0.66	0.50	0.50	0.51	0.51	0.53	0.06
Повезаност објеката	4.15	4.23	4.38	4.38	4.46	4.23	4.08	4.31	4.46	4.46	4.31	0.14
Дубина стабла наслеђивања	1.64	1.64	1.64	1.64	1.64	1.64	1.64	1.64	1.64	1.64	1.64	0.00
Број подкласа	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.00
Број наредби у методи	2.50	2.51	2.96	2.62	2.79	2.55	2.57	2.50	2.52	2.64	2.62	0.15
Метрика стабилности	0.47	0.47	0.47	0.47	0.47	0.47	0.47	0.47	0.47	0.47	0.47	0.00

Метрика	Студенти експерименталне групе											
	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	$\bar{x}$	σ
софтвера												

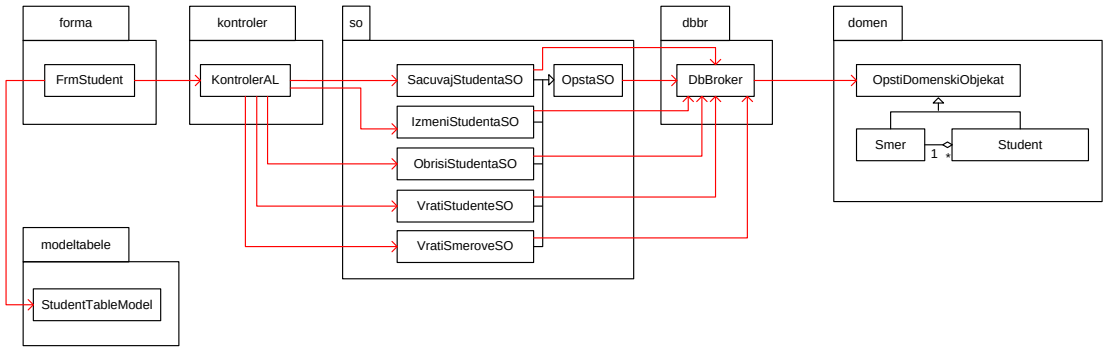
У наредној табели (Табела 80) дат је приказ вредности софтверских метрика за пројекте студената контролне групе. У табели су, такође, приказане аритметичка средина и стандардна девијација вредности за сваку софтверску метрику.

Табела 80. Вредности софтверских метрика за пројекте студената контролне групе

Метрика	Студенти контролне групе											
	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	$\bar{x}$	σ
Циклична сложеност	1.21	1.48	1.32	1.21	1.56	1.35	1.25	1.35	1.41	1.43	1.36	0.12
Сложеност пондерисаних метода	9.45	7.67	8.55	9.45	9.09	8.11	9.23	8.72	8.73	9.40	8.84	0.60
Број пондерисаних метода класе	7.82	5.17	6.45	7.82	5.82	6.60	6.62	7.12	5.54	7.25	6.62	0.91
Број одговора класе	50.09	43.25	49.55	50.09	47.82	49.35	51.03	46.75	46.35	47.30	48.16	2.34
Недостатак кохезивности метода у класи	0.51	0.54	0.55	0.51	0.51	0.52	0.53	0.51	0.53	0.54	0.52	0.02
Повезаност објеката	6.14	7.00	6.43	6.14	5.86	6.35	6.31	5.97	6.39	6.46	6.31	0.32
Дубина стабла наслеђивања	1.14	1.17	1.29	1.14	1.29	1.25	1.17	1.27	1.13	1.29	1.21	0.07
Број подкласа	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Број наредби у методи	3.02	3.65	3.77	3.02	4.42	3.55	3.68	4.05	3.16	3.51	3.58	0.44
Метрика	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.00

Метрика	**Студенти контролне групе**											
	C1	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**	**C8**	**C9**	**C10**	$\bar{x}$	**σ**
стабилности софтвера												

На наредној слици (Слика 103) дат је приказ архитектуре софтверског система који су применили студенти експерименталне групе[20]. Архитектура софтверског система дефинисана је упрошћеном Ларамановом методом развоја софтвера која представља саставни део SilabQOSS методе за побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера.

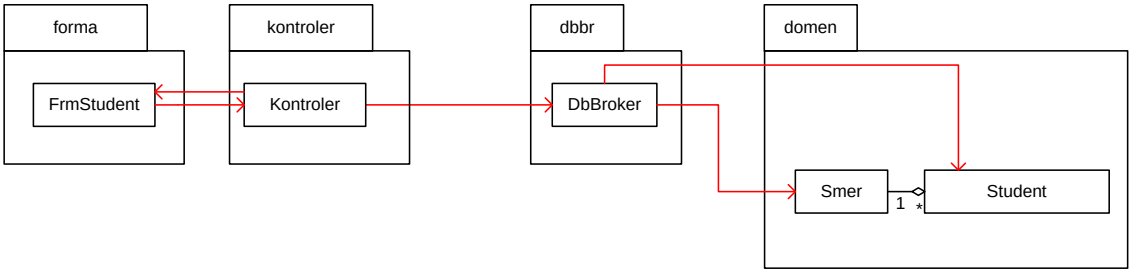


Слика 103. Архитектура софтверског система - експериментална група

С друге стране, студенти контролне групе су процес развоја софтвера реализовали без примене неке одређене методе: они су имали могућност да сами дефинишу архитектуру коју ће применити у процесу развоја софтвера.

[20] Увидом у програмски код закључено је да су студенти експерименталне групе у процесу имплементације давали различите називе класама и пакетима. У том смислу је на слици дат уопштени приказ архитектуре који су применили сви студенти експерименталне групе.

На наредној слици (Слика 104) дат је уопштени приказ архитектуре софтверског система коју су применили студенти контролне групе[21].



Слика 104. Архитектура софтверског система - контролна група

На основу изложеног се може закључити да се архитектуре софтверских система експерименталне и контролне групе разликују. Најпре, студенти експерименталне групе су сваку системску операцију имплементирали као посебну класу, док су студенти контролне групе системске операције имплементирали у оквиру метода контролера. Такође, студенти експерименталне групе су пројектовали генеричке методе брокера базе података, док су студенти контролне групе пројектовали специфичне методе брокера базе података.

У наставку рада даје се анализа софтверских метрика: Циклична сложеност, Сложеност пондерисаних метода, Број пондерисаних метода класе, Број одговора класе, Недостатак кохезивности метода у класи, Повезаност објеката, Дубина стабла наслеђивања, Број подкласа, Број наредби у методи и Метрика стабилности софтвера. За сваку метрику даје се анализа добијених вредности, као и промена вредности метрике у случају повећања броја системских операција и доменских класа[22].

[21] Увидом у програмски код закључено је да су студенти контролне групе у процесу имплементације давали различите називе класама и пакетима. У том смислу је на слици дат уопштени приказ архитектуре који су применили сви студенти контролне групе.
[22] Претпоставка је да су системске операције и доменске класе које се додају сличне сложености као постојеће системске операције и доменске класе.

8.3.1.1. Анализа софтверске метрике Циклична сложеност

Софтверска метрика Циклична сложеност усмерена је на класу и њене методе. У том смислу циклична сложеност представља сложеност примењеног алгоритма у методи. Метрика анализира операторе и наредбе посматраног програмског језика на основу чега се одређује сложеност посматране методе. Циклична сложеност сваке функције је већа или једнака 1.

Просечна циклична сложеност метода у експерименталној групи износи 1.22 ± 0.05. С друге стране, просечна циклична сложеност метода у контролној групи износи 1.36 ± 0.12. Из изложеног се закључује да методе нису сложене, при чему је просечна циклична сложеност метода у контролној групи већа у односу на просечну цикличну сложеност метода у експерименталној групи.

На основу приказаних архитектура софтверских система у експерименталној и контролној групи и анализе метрике Циклична сложеност можемо закључити да повећање броја доменских класа у софтверском систему не доводи до повећања вредности посматране метрике у експерименталној и контролној групи. Доменске класе садрже само методе за постављање и узимање вредности атрибута чија циклична сложеност износи 1, што представља најмању могућу вредност.

С друге стране, повећање броја системских операција у софтверском систему такође не доводи до повећања вредности посматране метрике у експерименталној и контролној групи (претпоставка је да су системске операција сличне сложености као постојеће операције).

8.3.1.2. Анализа софтверске метрике Сложеност пондерисаних метода

Софтверска метрика Сложеност пондерисаних метода усмерена је на класу и њене методе. У том смислу сложеност пондерисаних метода представља суму сложености метода класе. Као мера сложености методе класе користи се метрика Циклична сложеност.

Просечна сложеност пондерисаних метода у експерименталној групи износи: WMC2 = 7.12 ± 0.44.

Посматрана вредност се не може тумачити самостално већ се у обзир мора узети и вредност метрике Број пондерисаних метода класе. У том смислу, просечан број пондерисаних метода класе у експерименталној групи износи: WMC1 = 5.54 ± 0.19.

У том смислу, уколико се у експерименталној групи посматра однос метрика Број пондерисаних метода класе и Сложеност пондерисаних метода добија се следеће:

$$WMC_{odn} = \frac{\text{Број пондерисаних метода класе класе}}{\text{Сложеност пондерисаних метода}} = \frac{WMC1}{WMC2} = \frac{5.54}{7.12} = 0.78.$$

С друге стране, просечна сложеност пондерисаних метода у контролној групи износи: WMC2 = 8.84 ± 0.60. Број пондерисаних метода класе у контролној групи износи WMC1 = 6.62 ± 0.91. Уколико се размотри однос посматраних метрика у контролној групи добија се следеће:

$$WMC_{odn} = \frac{\text{Број пондерисаних метода класе класе}}{\text{Сложеност пондерисаних метода}} = \frac{WMC1}{WMC2} = \frac{6.62}{8.84} = 074.$$

С обзиром да је однос WMC_{odn} мањи или једнак 1, може се закључити да методе класе у експерименталној и контролној групи нису сложене. Другим речима, примењени алгоритми у методама су једноставни за разумевање, надоградњу и одржавање. С друге стране, примећује се већа просечна сложеност пондерисаних метода у контролној групи у односу просечну сложеност пондерисаних метода у експерименталној групи.

На основу приказаних архитектура софтверских система у експерименталној и контролној групи и анализе метрике Сложеност пондерисаних метода можемо закључити да повећање броја доменских класа у софтверском систему не доводи до повећања вредности посматране метрике у експерименталној и контролној групи. Доменске класе садрже само методе за постављање и узимање вредности атрибута чија циклична сложеност

износи 1, па ће сложеност пондерисаних метода бити једнака збиру ових цикличних сложености.

С друге стране, повећање броја системских операција у софтверском систему такође не доводи до повећања вредности посматране метрике у експерименталној и контролној групи (претпоставка је да су системске операција сличне сложености као постојеће операције).

8.3.1.3. Анализа софтверске метрике Број пондерисаних метода класе

Софтверска метрика Број пондерисаних метода класе је, такође, усмерена на класу и њене методе. Метрика Број пондерисаних метода класе се рачуна непосредним бројањем метода у класи.

Просечан број пондерисаних метода класе у експерименталној групи износи: WMC1 = 5.54 ± 0.19. С друге стране, просечан број пондерисаних метода класе у контролној групи износи: WMC1 = 6.62 ± 0.91.

На основу вредности посматране метрике се може извести закључак да класе немају превелик број метода. Стога се може рећи да су класе једноставне за разумевање и одржавање. С друге стране, посматрана метрика се користи заједно са метриком Сложеност пондерисаних метода. У том смислу се, такође, може извести закључак да методе класе у експерименталној и контролној групи нису сложене (образложење је дато у оквиру претходног одељка). Такође се примећује да је просечан број пондерисаних метода класе у контролној групи већи у односу на просечан број пондерисаних метода класе у експерименталној групи.

Уколико претпоставимо да се у софтверском систему повећава број доменских класа може се закључити да ће и у експерименталној и у контролној групи порасти број пондерисаних метода класе, при чему ће број пондерисаних метода у експерименталној групи бити већи од броја пондерисаних метода у контролној групи. Наиме, у експерименталној и

контролној групи у оквиру доменских класа постоје методе за постављање и узимање вредности атрибута (енг. getter and setter methods). Поред тога, у оквиру доменских класа експерименталне групе постоје и додатне методе које су дефинсане интерфејсом OpstiDomenskiObjekat, што последично повећава број пондерисаних метода класе.

С друге стране, повећање броја системских операција довело би до повећања вредности метрике Број пондерисаних метода класе у контролној групи, док се вредност посматране метрике у експерименталној групи у том случају не мења. Наиме, у експерименталној групи је број пондерисаних метода класе дефинисан класом OpstaSO. С друге стране, у контролној групи се за нове системске операције у оквиру контролера морају додати нове методе што последично повећава број пондерисаних метода класе.

8.3.1.4. Анализа софтверске метрике Број одговора класе

Софтверска метрика Број одговора класе усмерена је на класу. Метриком Број одговора класе дефинише се скуп одговора класе, односно скуп свих метода које могу бити позване као одговор на поруку објекта класе.

Просечан број одговора класе у експерименталној групи износи 31.16 ± 1.40. С друге стране, просечан број одговора класе у контролној групи износи 48.16 ± 2.34. С обзиром да је студијски пример који су студенти имплементирали мали (садржи четири случаја коришћења који се извршавају на два доменска објекта) може се рећи да је у експерименталној и контролној групи број одговора класе велики. Такође се примећује да је број одговора класе у контролној групи већи од броја одговора класе у експерименталној групи.

Анализом програмског кода експерименталне групе долази се до закључка да је број одговора класе велики зато што у оквиру класа које представљају доменске објекте (у посматраном студијском примеру то су класе Student и Smer) постоји велики број јавних метода:

- непараметарски конструктори класе,
- параметарски конструктори класе,
- методе за узимање вредности атрибута (енг. Getter methods),
- методе за постављање вредности атрибута (енг. Setter methods)[23],
- методе интерфејса OpstiDomenskiObjekat који доменске класе имплементирају,
- метода *toString()* која је дефинисана у оквиру класе Object[24].

Посматране методе се позивају из других класа (нпр. из метода брокера базе података, из контролера апликационе логике, из системских операција, са екранских форми и сл.). Уколико ове методе не би постојале број одговора класе би био значајно мањи[25].

С друге стране, анализом програмског кода контролне групе долази се до истог закључка као и за експерименталну групу. Поред тога, студенти контролне групе су за сваки атрибут форме дефинисали додатне методе за постављање и узимање вредности атрибута, за сваку графичку компоненту, што је додатно повећало број одговора класе. Уколико ове методе не би постојале број одговора класе би био на нивоу броја одговора класе експерименталне групе.

На основу изложеног је могуће извести веома важан закључак: у процесу развоја софтвера велику пажњу треба посветити доменским класама, у

[23] Методе за узимање и постављање вредности атрибута се дефинишу за сваки атрибут класе појединачно. Посматране методе се користе у пару за приступ атрибуту класе. Другим речима, уколико би посматрана класа имала пет атрибута, у њој би постојало десет метода за узимање и постављање вредности атрибута.

[24] Посматрана метода *toString()* представља методу надкласе. У оквиру доменских класа извршено је прекривање посматране методе (енг. method overriding). Поред тога, могуће је извршити прекривање и других метода (нпр. прекривање метода *equals()* и *hashCode()*). У посматраном студијском примеру извршено је прекривање методе *toString()* док прекривање других метода није извршено.

[25] Искључивање посматраних метода из статичке анализе квалитета софтвера није могуће зато што се оне позивају из остатка софтверског система. Њиховим брисањем програмски код се не може компајлирати па се самим тим не може извршити статичка анализа квалитета посматраних софтверских система.

смислу правилне спецификације доменских класа, атрибута доменских класа и односа који се успостављају између доменских класа.

Другим речима, доменске класе представљају суштину објектно-оријентисане анализе и на њима се заснива даљи развој софтверског система. Правилна спецификација доменских класа представља неопходан услов за даље пројектовање, имплементацију и тестирање софтверског система.

Другим речима, уколико претпоставимо да се у софтверском систему повећава број доменских класа може се закључити да ће и у експерименталној и у контролној групи порасти број одговора класе (додавање нових класа и метода свакако доводи до повећања броја одговора класе), при чему ће број одговора класе у контролној групи бити већи од броја одговора класе у експерименталној групи. У контролној групи су методе брокера базе података специфичне, док су у експерименталној групи методе брокера базе података генеричке. У том случају се приликом додавања нових доменских класа у контролној групи врши додавање нових специфичних метода брокера базе података, док се број генеричких метода брокера базе података у експерименталној групи не мења.

С друге стране, додавање нових системских операција довело би до пораста вредности посматране метрике и у експерименталној и у контролној групи. На тај начин се уводи ново понашање у софтверском систему што последично повећава број одговора класе.

8.3.1.5. Анализа софтверске метрике Недостатак кохезивности метода у класи

Софтверска метрика Недостатак кохезивности метода у класи усмерена је на класу и атрибуте класе. Кохезија представља меру којом се утврђује повезаност атрибута и метода у оквиру класе. У том смислу се за класу каже да је кохезивна ако се методе класе извршавају над истим скупом атрибута. Посматрана метрика мери недостатак кохезије метода у класи.

Просечан недостатак кохезивности метода експерименталне групе износи 0.53 ± 0.06. С друге стране, просечан недостатак кохезивности метода контролне групе износи 0.52 ± 0.02. С обзиром да је недостатак кохезивности метода у класи већи или једнак од 0, на основу вредности се може закључити да у посматраним софтверским системима постоји недостатак кохезивности метода у класи. С друге стране, може се закључити да у посматраним софтверским системима не постоји висок недостатак кохезивности метода у класи[26]. Такође се може закључити да је недостатак кохезивности метода у класи у експерименталној групи већи од недостатка кохезивности метода у класи у контролној групи. Студенти контролне групе су имали већи број метода који се извршавао над различитим скупом атрибута (нпр. у класи која је задужена за рад са базом података и у класи која представља екранску форму постоји већи број метода које раде над различитим скупом атрибута), што је последично довело до веће вредности метрике Недостатак кохезивности метода у класи.

Другим речима, повећањем броја доменских класа дошло би до повећања недостатка кохезивности у експерименталној и контролној групи, при чему би недостатак кохезивности био већи у контролној групи.

С друге стране, повећање броја системских операција не утиче на недостатак кохезивности метода у експерименталној и контролној групи.

8.3.1.6. Анализа софтверске метрике Повезаност објеката

Софтверска метрика Повезаност објеката усмерена је на однос између класа. У том смислу у оквиру метрике се посматра однос између класа, њихових метода и њихових атрибута: две класе су међусобно повезане уколико се у оквиру једне класе користе методе и/или атрибути друге класе.

[26] Под високим недостатком кохезивности метода у класи сматра се свака вредност која је већа или једнака 1 [Henderson-Sellers96].

Просечна повезаност објеката у експерименталној групи износи 4.31 ± 0.14. С друге стране, просечна повезаност објеката у контролној групи износи 6.31 ± 0.32. Може се закључити да повезаност објеката није велика. Такође, закључује се и да је просечна повезаност објеката у контролној групи већа у односу на просечну повезаност објеката у експерименталној групи. Студенти контролне групе правили су директне зависности између класа (класа која је задужена за рад са базом података директно је повезана са свим доменским класама), што је последично довело до веће вредности метрике Повезаност објеката.

С друге стране, повећање броја системских операција довело би до повећања повезаности објеката у експерименталној групи. Наиме, у експерименталној групи се за сваку системску операцију прави посебна класа и врши њен позив из контролера, па додавање нових системских операција доводи до повећања повезаности објеката. При томе, додавање нових системских операција не повећава вредност метрике Повезаност објеката у контролној групи. У контролној групи су системске операције имплементиране у методама контролера, што значи да се додавањем нових системских операција не успостављају нове везе између објеката.

8.3.1.7. Анализа софтверске метрике Дубина стабла наслеђивања

Софтверска метрика Дубина стабла наслеђивања је усмерена на класу и одређивање њене позиције у хијерархији наслеђивања. Другим речима, метриком се одређује максимални број нивоа од посматраног чвора до корена стабла у хијерархији наслеђивања класа.

Просечна дубина стабла наслеђивања у експерименталној групи износи 1.64 ± 0.00. С друге стране, просечна дубина стабла наслеђивања у контролној групи износи 1.21 ± 0.07. С обзиром да је дубина стабла наслеђивања већа или једнака 0, може се рећи да класе не наслеђују превелики број класа што последично олакшава разумевање и даљи развој софтверског система.

Из изложеног се, такође, може приметити да у експерименталној групи за посматрану метрику не постоји стандардна девијација. Ово је последица примене предложене архитектуре софтверског система која је дефинисана упрошћеном Ларамановом методом развоја софтвера. Такође се примећује да је просечна дубина стабла наслеђивања у експерименталној групи већа у односу на просечну дубину стабла наслеђивања у контролној групи.

Другим речима, студенти експерименталне групе су у оквиру брокера базе података пројектовали генеричке методе чиме је омогућена материјализација и дематеријализација свих објеката који имплементирају интерфејс OpstiDomenskiObjekat. У том смислу додавање нових доменских класа повећава дубину стабла наслеђивања у експерименталној групи. С друге стране, додавање нових доменских класа не утиче на дубину стабла наслеђивања у контролној групи.

Такође, студенти експерименталне групе су у оквиру класе OpstaSO дефинисали опште извршење системске операције, док су у подкласама дефинисали специфично извршење сваке системске операције. С друге стране, студенти контролне групе су системске операције имплементирали у оквиру метода контролера. Због тога је вредност метрике Дубина стабла наслеђивања у експерименталној групи већа у односу на вредност исте метрике у контролној групи. Другим речима, додавањем нових системских операција повећава се дубина стабла наслеђивања у експерименталној групи. С друге стране, додавање нових системских операција не утиче на дубину стабла наслеђивања у контролној групи.

8.3.1.8. Анализа софтверске метрике Број подкласа

Софтверска метрика Број подкласа је усмерена на класе. Метриком Број подкласа одређује се број непосредних потомака посматране класе у хијерархији наслеђивања.

Просечан број подкласа у експерименталној групи износи 0.50 ± 0.00. С друге стране, просечан број подкласа у контролној групи износи 0.00 ± 0.00.

С обзиром да је број подкласа већи или једнак 0, може се рећи да класе немају превелики број подкласа што последично олакшава разумевање и даљи развој софтверског система. Из изложеног се, такође, може приметити да у експерименталној групи за посматрану метрику не постоји стандардна девијација. Ово је последица примене предложене архитектуре софтверског система која је дефинисана упрошћеном Ларамановом методом развоја софтвера.

Студенти експерименталне групе су приликом пројектовања доменских класа и генеричких метода брокера базе података имали две имплементације интерфејса OpstiDomenskiObjekat (то су класе Student и Predmet). Због тога би додавање нових доменских класа повећало број подкласа у експерименталној групи. С друге стране, додавање нових доменских класа не утиче на број подкласа контролној групи.

Такође, студенти експерименталне групе су дефинисали пет класа за извршење системских операција (то су класе SacuvajStudentaSO, IzmeniStudentaSO, ObrisiStudentaSO, VratiStudenteSO и VratiSmeroveSO) које наслеђују класу OpstaSO. С друге стране, студенти контролне групе су системске операције имплементирали у оквиру метода контролера. Због тога је вредност метрике Број подкласа у експерименталној групи већа у односу на вредност исте метрике у контролној групи. Другим речима, додавањем нових системских операција повећава се број подкласа у експерименталној групи. С друге стране, додавање нових системских операција не утиче на број подкласа у контролној групи. Штавише, примећује се да у контролној групи подкласе не постоје, што значи да у процесу развоја софтвера није примењиван концепт наслеђивања.

8.3.1.9. Анализа софтверске метрике Број наредби у методи

Софтверска метрика Број наредби у методи је усмерена на методу класе и одређивање непосредног броја наредби које се позивају унутар посматране методе.

Просечан број наредби у методи у експерименталној групи износи 2.62 ± 0.15. С друге стране, просечан број наредби у методи у контролној групи износи 3.58 ± 0.44. У том смислу је могуће закључити да методе класа у експерименталној и контролној групи не садрже велики број наредби. На тај начин је омогућено лако одржавање и надоградња метода. Такође се примећује да је просечан број наредби у контролној групи већи од просечног броја наредби у експерименталној групи.

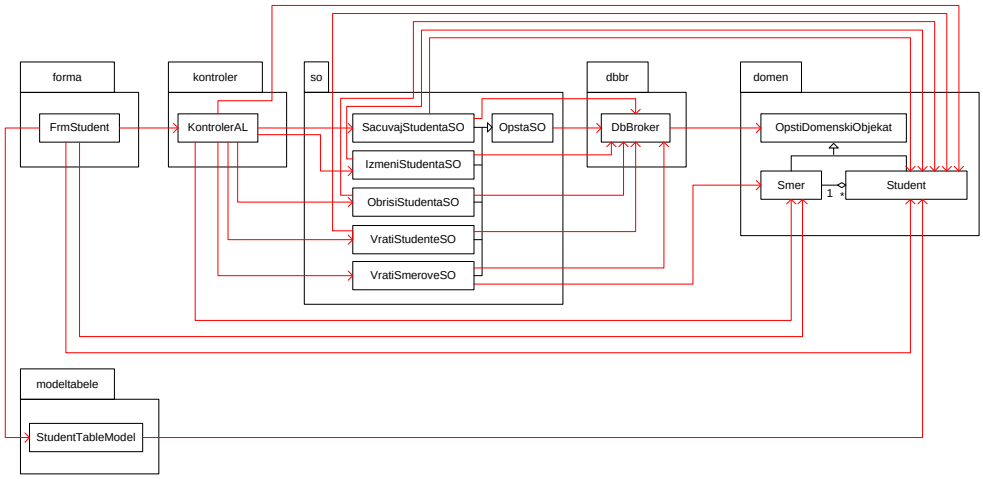
Уколико би у софтверском систему дошло до повећања броја доменских класа у експерименталној и у контролној групи не би дошло до повећања вредности посматране метрике. Штавише, за сваки атрибут доменске класе дефинишу се методе за постављање и узимање вредности атрибута, што последично доводи до смањења просечног броја наредби у методи.

Студенти контролне групе су у оквиру метода контролера вршили имплементацију системских операција, што је довело до понављања програмског кода (тј. до понављања метода за учитавање управљачког програма, за успостављање везе са базом података, за потврду и/или поништење трансакције, за затварање конекције) , што је последично довело до повећања броја наредби у методи. С друге стране, студенти експерименталне групе су у оквиру класе OpstaSO дефинисали опште извршење системске операције, док су у подкласама дефинисали специфично извршење сваке системске операције, чиме је избегнуто претходно објашњено понављање програмског кода које постоји код контролне групе. Због тога је просечан број наредби у експерименталној групи мањи у односу на просечан број наредби у контролној групи.

8.3.1.10. Анализа софтверске метрике Метрика стабилности софтвера

Метриком стабилности софтвера[27] разматра се однос између пакета који постоје у посматраном софтверском систему.

Просечна вредност метрике стабилности софтвера у експерименталној износи 0.47 ± 0.00. С обзиром да метрика стабилности софтвера узима вредности између 0 и 1, може се рећи да у софтверским системима студената експерименталне групе постоји нестабилност. Из изложеног се, такође, може приметити да за посматрану метрику не постоји стандардна девијација. Када је експериментална група у питању, ово је последица примене предложене архитектуре софтверског система која је дефинисана упрошћеном Ларамановом методом развоја софтвера. С обзиром на примењену архитектуру софтверског система, однос између пакета и класа софтверског система експерименталне групе је приказан на наредној слици (Слика 105).



Слика 105. Однос између пакета и класа софтверског система - експериментална група

[27] Анализа софтверских система извршена је коришћењем побољшане верзије метрике стабилности софтвера [Vlajic17]. Наиме, у Поглављу 4. је објашњено да метрика стабилности I врши релативизацију одлазне повезаности када не постоји долазна повезаност, односно релативизацију долазне повезаности када не постоји одлазна повезаност. Због тога се стабилност софтверских система разматра коришћењем побољшане верзије метрике стабилности I_r.

У том смислу могуће је одредити одлазне и долазне повезаности за сваки пакет, на основу чега се може израчунати вредност Метрике стабилности софтвера експерименталне групе, што је и приказано у наредној табели (Табела 81).

Табела 81. Метрика стабилности софтвера - експериментална група

Пакет	**C_a**	**C_e**	**I_r**
domen	11	0	0.21
dbbr	6	1	0.34
modeltabele	1	1	0.50
so	5	11	0.62
forma	0	4	0.66
kontroler	1	7	0.69

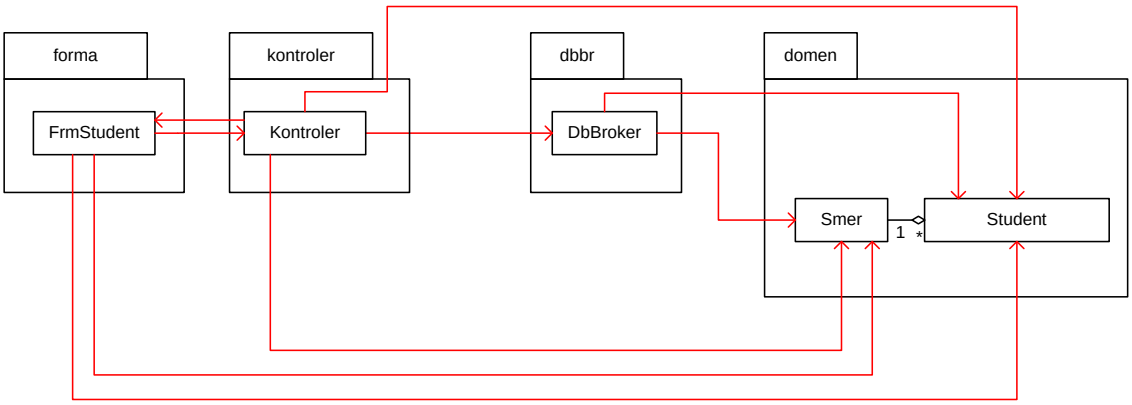
На основу извршеног долазимо до закључка стабилности пакета у експерименталној групи:

- Најстабилнији пакет је пакет domen (његова нестабилност износи 0.21). Он има 11 долазних повезаности, при чему нема одлазних повезаности. Он је једини пакет у софтверском систему који нема зависности од других пакета, већ други пакети зависе од њега.
- Пакет dbbr у којем се налази брокер базе података је такође веома стабилан (његова нестабилност износи 0.34). Наиме, посматрани пакет има 6 долазних повезаности, при чему има само једну одлазну повезаност. Долазне повезаности односе се на везу са системским операцијама, док се одлазна повезаност односи на везу са интерфејсом OpstiDomenskiObjekat из пакета domen.
- Пакет modeltabele има само једну долазну повезаност и једну одлазну повезаност. Због тога је његова нестабилност 0.5. Долазна повезаност односи се на везу са формом, док се одлазна повезаност односи на везу са доменском класом.

- Пакет so има нестабилност 0.62. Посматрани пакет има 11 одлазних повезаности (шест одлазних повезаности се односе на везе са брокером базе података док се пет одлазних повезаности односе на везе са доменским класама). Такође, посматрани пакет има 5 долазних повезаности које се односе на везу са контролером апликационе логике (из контролера апликационе логике врши се позив системских операција).
- Пакет forma има нестабилност 0.66. Посматрани пакет има 4 одлазне повезаности (једна одлазна повезаност односи се на везу са контролером, једна одлазна повезаност односи се на везу са моделом табеле и две одлазне повезаности односе не везе са доменским класама). С друге стране, посматрани пакет нема долазних повезаности[28].
- Пакет kontroler има нестабилност 0.69. Посматрани пакет има 7 одлазних повезаности (пет одлазних повезаности се односе на везе са системским операцијама док се две одлазне повезаности односе на везе са доменским класама). Такође, посматрани пакет има 1 долазну повезаност која се односи на везу са формом.

С друге стране, вредност метрике стабилности софтвера у контролној групи износи 0.50 ± 0.00. С обзиром да метрика стабилности софтвера узима вредности између 0 и 1, може се рећи да у софтверским системима студената контролне групе постоји нестабилност. Из изложеног се, такође, може приметити да за посматрану метрику не постоји стандардна девијација. С обзиром на примењену архитектуру софтверског система, однос између пакета и класа софтверског система контролне групе је приказан на наредној слици (Слика 106).

[28] Посматрани пакет може имати долазних повезаности уколико би се, на пример, посматрана форма позивала са главне форме или са неке друге форме. Постојање долазних повезаности би последично утицало на вредност метрике стабилности софтвера. Међутим, у датом студијском примеру не постоје долазне повезаности.



Слика 106. Однос између пакета и класа софтверског система - контролна група

У том смислу могуће је одредити одлазне и долазне повезаности за сваки пакет, на основу чега се може израчунати вредност Метрике стабилности софтвера контролне групе, што је и приказано у наредној табели (Табела 82).

Табела 82. Метрика стабилности софтвера - контролна група

Пакет	**C_a**	**C_e**	**I_r**
domen	6	0	0.29
dbbr	1	2	0.54
kontroler	1	4	0.61
forma	1	3	0.58

На основу извршеног долазимо до закључка стабилности пакета у контролној групи:

- Најстабилнији пакет је пакет domen (његова нестабилност износи 0.29). Он има 6 долазних повезаности, при чему нема одлазних повезаности. Он је једини пакет у софтверском систему који нема зависности од других пакета, већ други пакети зависе од њега. Његова нестабилност је већа у односу на нестабилност пакета domen у експерименталној групи која је износила 0.21.

- Пакет dbbr у којем се налази брокер базе података има нестабилност 0.54. Наиме, посматрани пакет има 1 долазну повезаност, при чему има две одлазне повезаности. Долазна повезаност односи се на везу са контролером, док се одлазне повезаности односе на везу са доменским класама Student и Smer. Уколико би се повећао број доменских класа, брокер базе података би морао имати везе са сваком од класа што би последично још више повећало његову нестабилност. У том смислу се примећује да је његова нестабилност већа у односу на нестабилност dbbr пакета у експерименталној групи која је износила 0.34.
- Пакет kontroler има нестабилност 0.61. Посматрани пакет има једну долазну повезаност која се односи на везу са формом, као и четири одлазне повезаности (једна одлазна повезаност се односи на везу са формом, једна одлазна повезаност се односи на везу са брокером базе података, две одлазне повезаности се односе на везу са доменским класама Student и Smer). У том смислу се примећује да је његова нестабилност мања у односу на нестабилност пакета kontroler у експерименталној групи која је износила 0.69.
- Пакет forma има нестабилност 0.58. Посматрани пакет има једну долазну повезаност која се односи на везу са контролером, као и три одлазне повезаности (једна одлазна повезаност се односи на везу са контролером, две одлазне повезаности се односе на везу са доменским класама Student и Smer). У том смислу се примећује да је његова нестабилност мања у односу на нестабилност пакета forma у експерименталној групи која је износила 0.66.

На основу приказаних архитектура софтверских система експерименталне и контролне групе и анализе стабилности појединачних пакета могуће је извести опште закључке који се односе на Метрику стабилности софтвера. Другим речима, уколико претпоставимо да се у софтверском систему

повећава број доменских класа могуће је извести следеће закључке о Метрици стабилности софтвера:

- Уколико би се повећао број доменских класа у софтверском систему, брокер базе података у контролној групи би морао имати везе са сваком од доменских класа што би последично повећало број одлазних повезаности и његову нестабилност. С друге стране, додавањем нових доменских класа брокер базе података у експерименталној групи и даље има само једну одлазну повезаност са интерфејсом OpstiDomenskiObjekat. Другим речима, додавањем нових доменских класа у експерименталној групи не доводи до промене нестабилности брокера базе података, тј. посматрани пакет остаје стабилан.
- Повећање броја доменских класа у експерименталној и контролној групи последично доводи и до повећања нестабилности пакета који садржи екранске форме.
- Повећање броја доменских класа у експерименталној и контролној групи доводи и до повећања нестабилности пакета који садржи контролер апликационе логике.

С друге стране, уколико претпоставимо да се у софтверском систему повећава број системских операција могуће је извести следеће закључке о Метрици стабилности софтвера:

- Повећањем броја системских операција брокер базе података у експерименталној групи би морао имати везе са сваком системском операцијом што би последично повећало број долазних повезаности и смањило његову нестабилност. С друге стране, у контролној групи су системске операције имплементиране у оквиру метода контролера апликационе логике, па стога повећање броја системских операција не доводи до промене нестабилности брокера базе података.

- Повећањем броја системских операција контролер апликационе логике у експерименталној групи би морао имати везе са сваком од системских операција што би последично повећало број одлазних повезаности и његову нестабилност. С друге стране, у контролној групи су системске операције имплементиране у оквиру метода контролера апликационе логике, па стога повећање броја системских операција не доводи до промене његове нестабилности.
- Повећање броја системских операција у експерименталној и контролној групи не доводи до промене нестабилности пакета који садржи екранске форме.

У наредној табели (Табела 83) дат је упоредни приказ промена вредности софтверских метрика у експерименталној и контролној групи у зависности од повећања броја доменских класа и системских операција.

Табела 83. Упоредни приказ промена вредности софтверских метрика у експерименталној и контролној групи

Софтверска метрика	**Повећање броја доменских класа**		**Повећање броја системских операција**	
	Експериментална група (Е)	**Контролна група (К)**	**Експериментална група (Е)**	**Контролна група (К)**
Циклична сложеност	Не мења се	Не мења се	Не мења се	Не мења се
Сложеност пондерисаних метода	Не мења се	Не мења се	Не мења се	Не мења се
Број пондерисаних метода класе	Повећање, веће него у (К) групи	Повећање, мање него у (Е) групи	Не мења се	Повећање
Број одговора класе	Повећање, мање него у (К) групи	Повећање, веће него у (Е) групи	Повећање	Повећање
Недостатак кохезивности метода у	Повећање, мање него у (К) групи	Повећање, веће него у (Е) групи	Не мења се	Не мења се

Софтверска метрика	**Повећање броја доменских класа**		**Повећање броја системских операција**	
	Експериментална група (Е)	**Контролна група (К)**	**Експериментална група (Е)**	**Контролна група (К)**
класи				
Повезаност објеката	Повећање, мање него у (К) групи	Повећање, веће него у (Е) групи	Не мења се	Не мења се
Дубина стабла наслеђивања	Повећање	Не мења се	Повећање	Не мења се
Број подкласа	Повећање	Не мења се	Повећање	Не мења се
Број наредби у методи	Смањење	Смањење	Не мења се	Повећање
Метрика стабилности софтвера	- Не утиче на нестабилност брокера базе података - Повећава нестабилност екранских форми - Повећава нестабилност контролера апликационе логике	- Повећава нестабилност брокера базе података - Повећава нестабилност екранских форми - Повећава нестабилност контролера апликационе логике	- Смањује нестабилност брокера базе података - Не утиче на нестабилност екранских форми - Повећава нестабилност контролера апликационе логике	- Не утиче на нестабилност брокера базе података - Не утиче на нестабилност екранских форми - Не утиче на нестабилност контролера апликационе логике

На основу приказаних архитектура софтверских система експерименталне и контролне групе и анализе софтверских метрика експерименталне и контролне групе уочавају се кључне разлике у софтверским системима:

- Студенти експерименталне групе су све системске операције имплементирали као посебне класе које су наслеђивале класу за извршење системске операције (у њој је било дефинисано опште

понашање, док су класе за извршење системских операција садржале специфична понашања). С друге стране, студенти контролне групе су системске операције имплементирали у оквиру метода контролера, са доста понављања програмског кода.

- Студенти експерименталне групе су пројектовали генеричке методе брокера базе података, док су студенти контролне групе пројектовали специфичне методе брокера базе података.

Архитектура коју су применили студенти експерименталне групе је сложенија у односу на архитектуру софтверског система коју су применили студенти контролне групе. С друге стране, студенти експерименталне групе су применили генеричка, општа и стабилнија решења која им омогућавају лакше одржавање и надоградњу софтверских система.

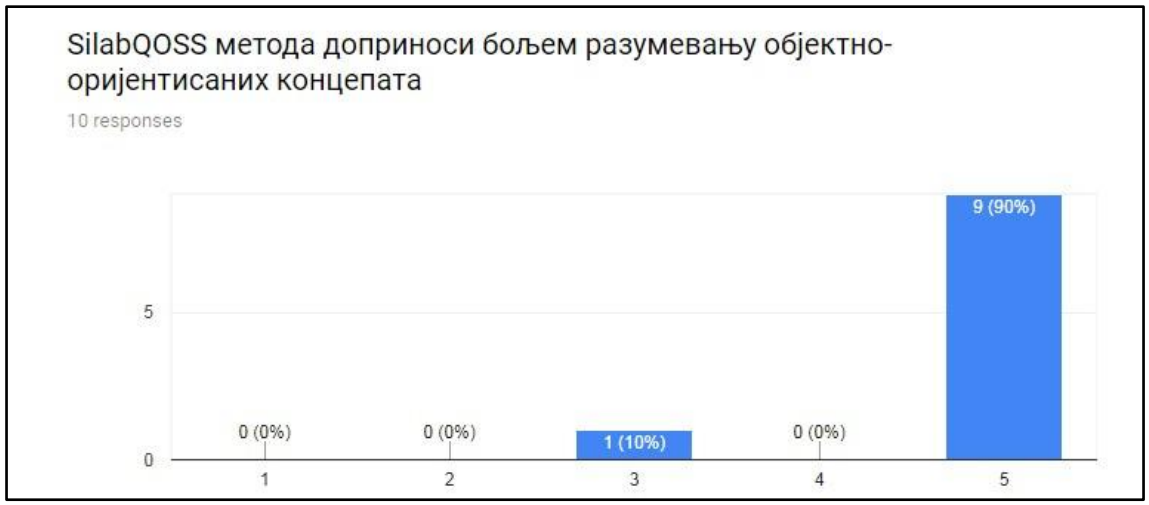
8.3.2. Приказ и анализа резултата упитника

Студенти експерименталне групе су попуњавали упитник који се односи евалуацију SilabQOSS методе за побољшање објектно-оријентисаних система коришћењем стандарда квалитета софтвера. У наредној табели (Табела 84) приказани су одговори студената на постављена питања. У табели су, такође, приказане аритметичка средина и стандардна девијација вредности за свако питање.

Табела 84. Одговори студената експерименталне групе на питања из упитника

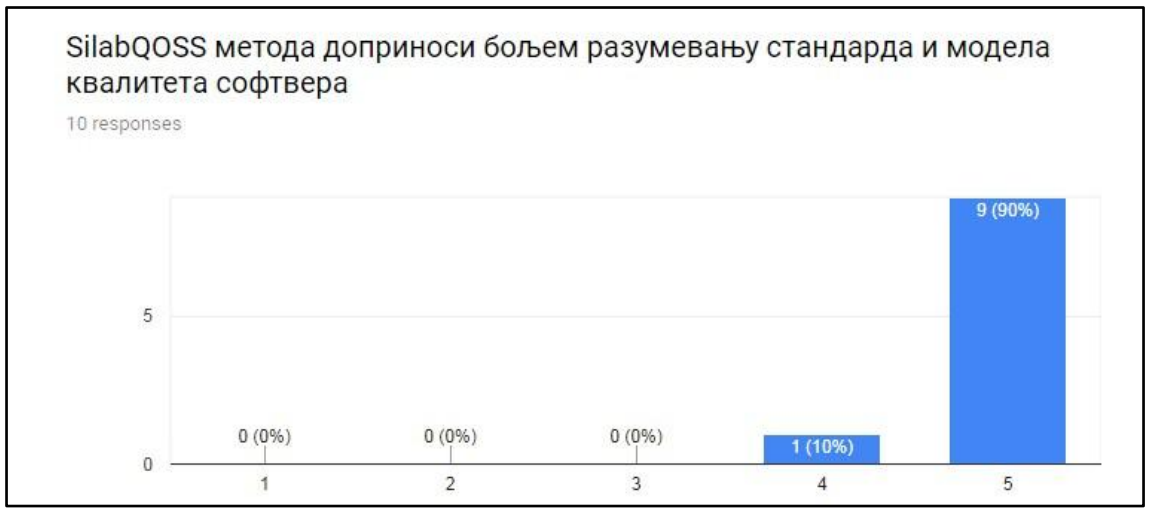
Питање	**Студенти експерименталне групе**											
	C1	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**	**C8**	**C9**	**C10**	$\bar{x}$	**σ**
1. Да ли SilabQOSS метода доприноси бољем разумевању објектно-оријентисаних концепата?	5	5	5	5	5	3	5	5	5	5	4.80	0.63
2. Да ли SilabQOSS метода доприноси бољем разумевању стандарда и модела квалитета софтвера?	5	5	5	5	5	5	5	4	5	5	4.90	0.32
3. Да ли SilabQOSS метода доприноси бољем разумевању софтверских метрика?	5	5	4	5	5	4	5	5	5	5	4.80	0.42
4. Да ли SilabQOSS метода доприноси бољем разумевању алата за статичку анализу квалитета софтвера?	5	5	5	5	5	3	5	5	4	5	4.70	0.67
5. Да ли SilabQOSS метода доприноси бољем разумевању механизама побољшања објектно-оријентисаних софтверских система?	5	5	5	5	5	3	5	5	5	5	4.80	0.63
6. Да ли интеграција алата за статичку анализу квалитета софтвера и развојног окружења побољшава процес развоја софтвера?	4	5	4	5	5	5	5	5	5	5	4.80	0.42
7. Да ли коришћење алата за статичку анализу квалитета софтвера омогућава брже и лакше проналажење грешака у софтверском систему?	5	5	5	5	5	4	5	5	4	5	4.80	0.42
8. Да ли се знање стечено на овај начин се може искористити у процесу развоја других софтверских система?	5	5	5	5	5	5	5	5	5	5	5.00	0.00

На наредној слици (Слика 107) приказана је расподела одговора студената експерименталне групе на питање "*1. Да ли SilabQOSS метода доприноси бољем разумевању објектно-оријентисаних концепата?*". На основу приказаног закључује се да је девет студената (90%) на постављено питање одговорило са оценом 5 (тј. "у потпуности се слажем"). С друге стране, један студент (10%) је одговорио са оценом 3 (тј. "неутрално"). У том контексту се може закључити да су студенти, уз један неутралан одговор, сагласни да посматрана метода доприноси бољем разумевању објектно-оријентисаних концепата.



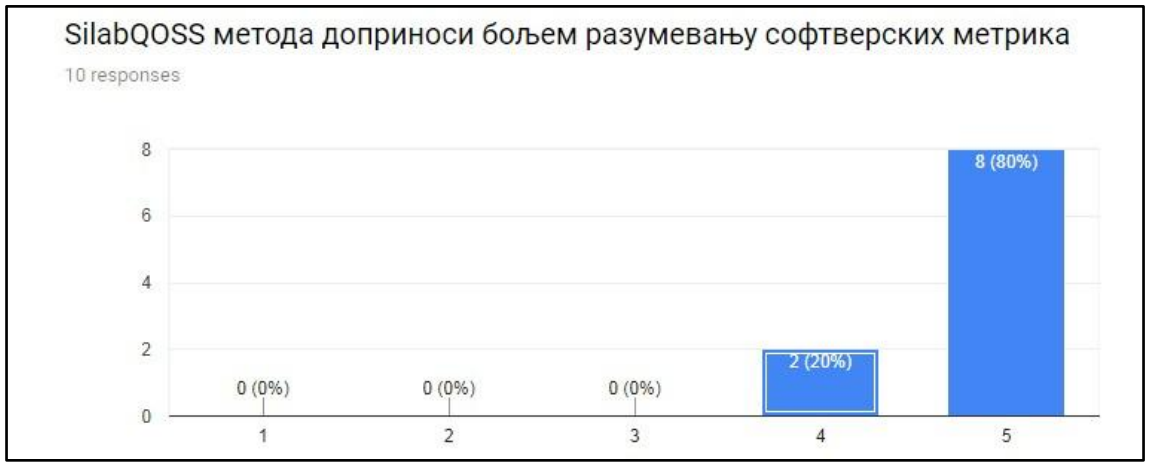
Слика 107. Расподела одговора студената експерименталне групе на прво питање

На наредној слици (Слика 108) приказана је расподела одговора студената експерименталне групе на питање "*2. Да ли SilabQOSS метода доприноси бољем разумевању стандарда и модела квалитета софтвера?*". На основу приказаног закључује се да је девет студената (90%) на постављено питање одговорило са оценом 5 (тј. "у потпуности се слажем"). С друге стране, један студент (10%) је одговорио са оценом 4 (тј. "слажем се"). У том контексту се може закључити да су студенти сагласни да посматрана метода доприноси бољем разумевању стандарда и модела квалитета софтвера.



Слика 108. Расподела одговора студената експерименталне групе на друго питање

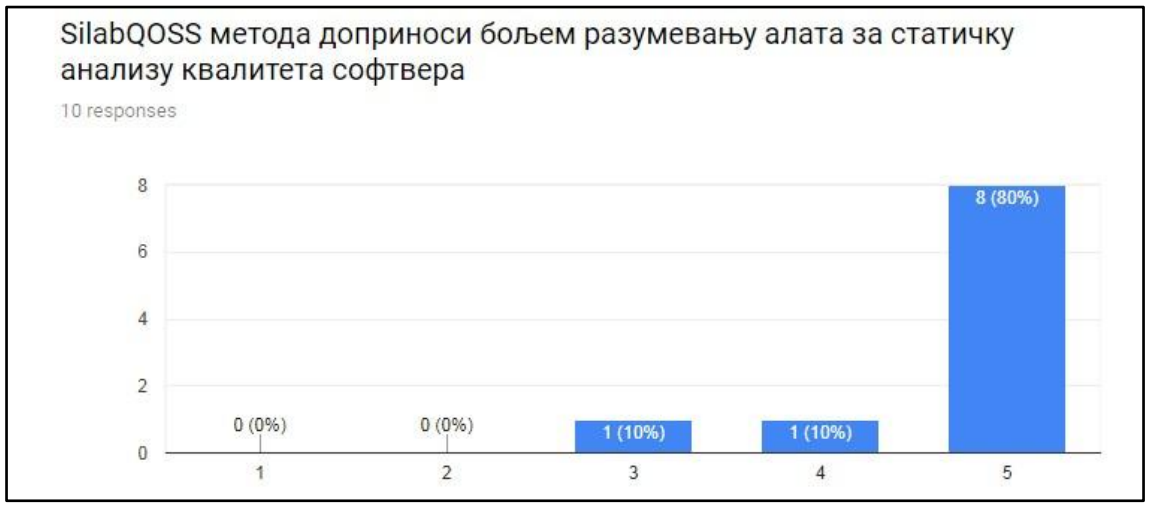
На наредној слици (Слика 109) приказана је расподела одговора студената експерименталне групе на питање "*3. Да ли SilabQOSS метода доприноси бољем разумевању софтверских метрика?*". На основу приказаног закључује се да је осам студената (80%) на постављено питање одговорило са оценом 5 (тј. "у потпуности се слажем"). С друге стране, два студента (20%) су одговорила са оценом 4 (тј. "слажем се"). У том контексту се може закључити да су студенти сагласни да посматрана метода доприноси бољем разумевању софтверских метрика.



Слика 109. Расподела одговора студената експерименталне групе на треће питање

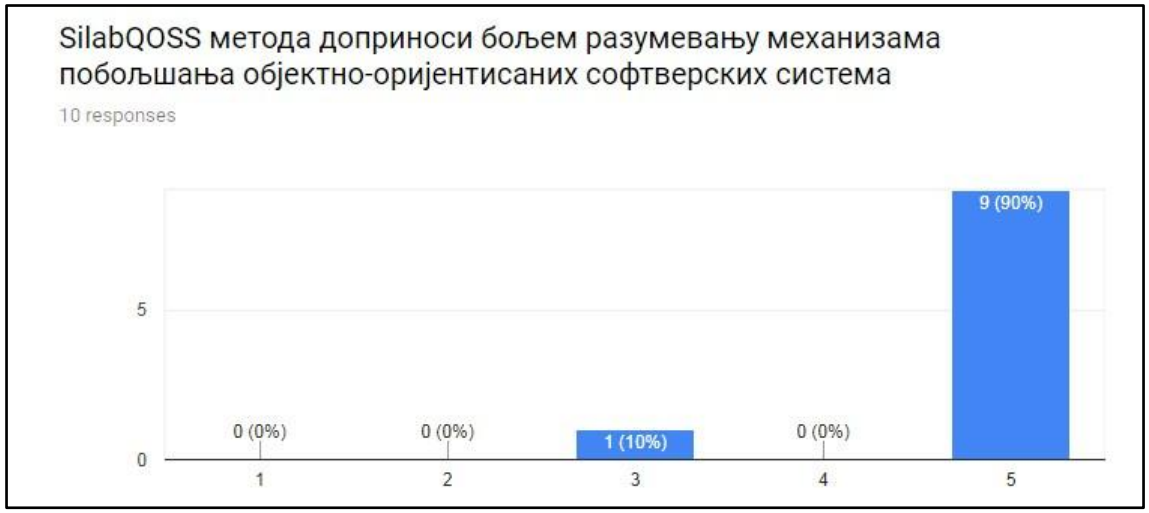
На наредној слици (Слика 110) приказана је расподела одговора студената експерименталне групе на питање "*4. Да ли SilabQOSS метода доприноси*

бољем разумевању алата за статичку анализу квалитета софтвера?". На основу приказаног закључује се да је осам студената (80%) на постављено питање одговорило са оценом 5 (тј. "у потпуности се слажем"). С друге стране, један студент (10%) је одговорио са оценом 4 (тј. "слажем се") и један студент (10%) је одговорио са оценом 3 (тј. "неутрално"). У том контексту се може закључити да су студенти, уз један неутралан одговор, сагласни да посматрана метода доприноси бољем разумевању алата за статичку анализу квалитета софтвера.



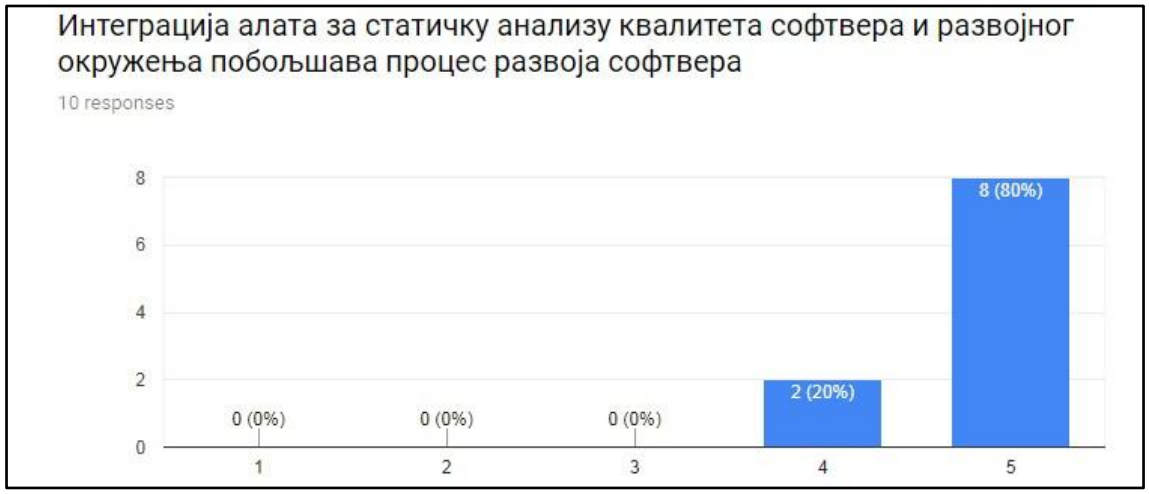
Слика 110. Расподела одговора студената експерименталне групе на четврто питање

На наредној слици (Слика 111) приказана је расподела одговора студената експерименталне групе на питање "5. *Да ли SilabQOSS метода доприноси бољем разумевању механизама побољшања објектно-оријентисаних софтверских система?*". На основу приказаног закључује се да је девет студената (90%) на постављено питање одговорило са оценом 5 (тј. "у потпуности се слажем"). С друге стране, један студент (10%) је одговорио са оценом 3 (тј. "неутрално"). У том контексту се може закључити да су студенти, уз један неутралан одговор, сагласни да посматрана метода доприноси бољем разумевању механизама побољшања објектно-оријентисаних софтверских система.



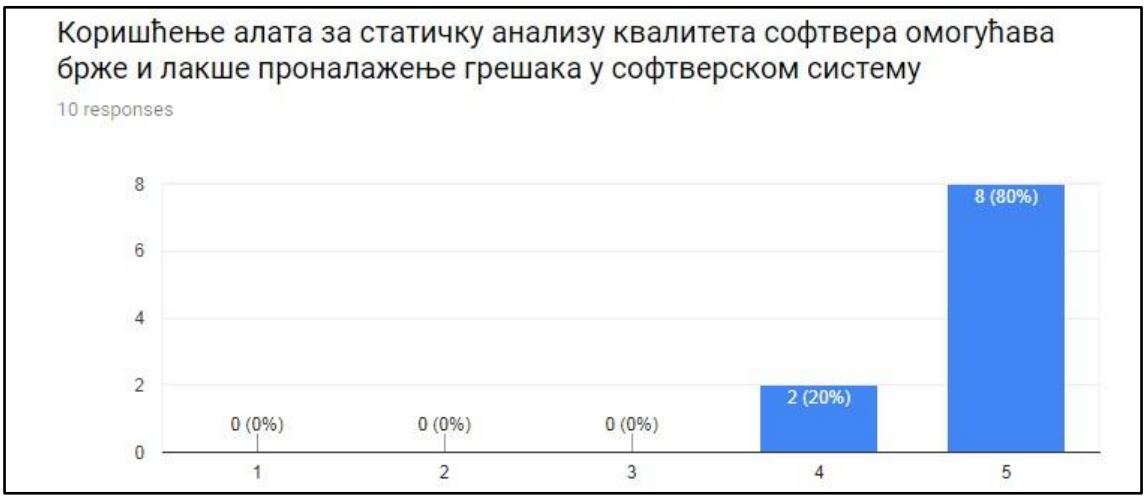
Слика 111. Расподела одговора студената експерименталне групе на пето питање

На наредној слици (Слика 112) приказана је расподела одговора студената експерименталне групе на питање "*6. Да ли интеграција алата за статичку анализу квалитета софтвера и развојног окружења побољшава процес развоја софтвера?*". На основу приказаног закључује се да је осам студената (80%) на постављено питање одговорило са оценом 5 (тј. "у потпуности се слажем"). С друге стране, два студента (20%) су одговорила са оценом 4 (тј. "слажем се"). У том контексту се може закључити да су студенти сагласни да интеграција алата за статичку анализу квалитета софтвера и развојног окружења побољшава процес развоја софтвера.



Слика 112. Расподела одговора студената експерименталне групе на шесто питање

На наредној слици (Слика 113) приказана је расподела одговора студената експерименталне групе на питање "*7. Да ли коришћење алата за статичку анализу квалитета софтвера омогућава брже и лакше проналажење грешака у софтверском систему?*". На основу приказаног закључује се да је осам студената (80%) на постављено питање одговорило са оценом 5 (тј. "у потпуности се слажем"). С друге стране, два студента (20%) су одговорила са оценом 4 (тј. "слажем се"). У том контексту се може закључити да су студенти сагласни да коришћење алата за статичку анализу квалитета софтвера омогућава брже и лакше проналажење грешака у софтверском систему.



Слика 113. Расподела одговора студената експерименталне групе на седмо питање

На наредној слици (Слика 114) приказана је расподела одговора студената експерименталне групе на питање "*8. Да ли се знање стечено на овај начин може искористити у процесу развоја других софтверских система?*". На основу приказаног закључује се да је десет студената (100%) на постављено питање одговорило са оценом 5 (тј. "у потпуности се слажем"). У том контексту се може закључити да су студенти сагласни да се знање стечено на овај начин може искористити у процесу развоја других софтверских система.

Слика 114. Расподела одговора студената експерименталне групе на осмо питање

8.4. Ограничења евалуације

У оквиру претходних одељака Поглавља 8. извршена је евалуација и анализа резултата евалуације SilabQOSS методе за побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера. У наставку се наводе ограничења евалуације:

- **У процесу евалуације учествовао је мали број испитаника**

  У реализацији студије случаја учествовало је двадесет студената. Они су били подељени у две групе: експерименталну групу (10 студената) и контролну групу (10 студената). У том смислу се може рећи да је број учесника мали. У циљу добијања квалитетнијих резултата потребно је да експериментална и контролна група обухвате већи број учесника. Другим речима, потребно је повећати узорак како би се добили што релевантнији резултати евалуације.

- **У процесу евалуације учествовали су само студенти**

  У претходној ставки напоменуто је да су у реализацији студије случаја учествовали студенти. Учешће студената у истраживању је свакако добро, али је такође потребно да се у процес истраживања укључе и инжењери различитих специјалности (нпр. софтвер инжењери, тестери, програмери, софтвер архитекте, инжењери који се баве осигурањем квалитета софтвера итд.), са различитим искуством (нпр.

јуниори, медиори, сениори итд.), у циљу добијања свеобухватних резултата.

- **Доменски модел и релациони модел су били једноставни**

  Примењени доменски модел је био једноставан. Наиме, посматрани модел је садржао само два доменска концепта (то су концепти Smer и Student) између којих је постојала једна веза асоцијације типа "један-према-више". Самим тим је и релациони модел био једноставан (постојале су две табеле, Smer и Student, при чему је у табели Student постојао спољни кључ табеле Smer).

  У софтверском систему доменски модел може бити значајно сложенији. У том смислу у софтверском систему може постојати велики број доменских концепата између којих је могуће успоставити различите везе. Последично би и релациони модел био сложенији. Другим речима, потребно је допунити доменски модел и релациони модел како би се добили што релевантнији резултати евалуације.

- **Имплементациони захтеви су били једноставни**

  У реализацији студије случаја вршена је имплементација четири операције (унос студента, измена студента, брисање студента и приказ студената). Посматране операције извршавале су се над доменским моделом који је садржао два концепта. У оквиру имплементираних операција није било додатних захтева који се односе на валидацију података и проверу ограничења над атрибутима, проверу предуслова за извршење операције, проверу постуслова извршења операције итд. Другим речима, имплементациони захтеви су се односили на реализацију CRUD[29] операција над базом података. С друге стране, у софтверском систему

[29] Акронимом CRUD означавају се основне операције над слоговима базе података: креирање (енг. **C**reate), селекција (енг. **R**ead), ажурирање (енг. **U**pdate) и брисање (енг. **D**elete).

може постојати већи број имплементационих захтева који могу бити веома сложени.

У том смислу, потребно је допунити постављене захтеве како би се обухватио већи број операција различите тежине за имплементацију. На тај начин се могу добити релевантнији резултати статичке анализе квалитета софтверских система.

8.5. Сажетак евалуације

У претходним одељцима извршена је евалуација предложене SilabQOSS методе за побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера. У том смислу је дефинисана студија случаја у чијој реализацији су учествовали студенти који су били подељени у две групе: експерименталну и контролну групу. Студенти експерименталне групе су у процесу имплементације користили SilabQOSS методу, док су студенти контролне групе процес имплементације спроводили без примене неке одређене методе. У том смислу се разликују архитектуре софтверских система експерименталне и контролне групе разликују:

- Студенти експерименталне групе су сваку системску операцију имплементирали као посебну класу, док су студенти контролне групе системске операције имплементирали у оквиру метода контролера.
- Студенти експерименталне групе су пројектовали генеричке методе брокера базе података, док су студенти контролне групе пројектовали специфичне методе брокера базе података.

Архитектура коју су применили студенти експерименталне групе је сложенија у односу на архитектуру софтверског система коју су применили студенти контролне групе. С друге стране, студенти експерименталне групе су применили генеричка решења која им омогућавају лакше одржавање и надоградњу софтверског система.

На основу извршене статичке анализе квалитета софтвера експерименталне и контролне групе могу се извести следећи закључци:

- Приликом примене SilabQOSS методе остварује се нижа вредност софтверских метрика Циклична сложеност, Сложеност пондерисаних метода, Број пондерисаних метода класе, Повезаност објеката, Број наредби у методи, Недостатак кохезивности метода у класи и Метрике стабилности софтвера у односу на вредности истих метрика у контролној групи. У том смислу се закључује да је у експерименталној групи примењени алгоритам у методама једноставнији за разумевање, да број метода у класи није велики, као и да су класе и пакети стабилнији, једноставнији за одржавање и даљу надоградњу.
- Приликом примене SilabQOSS методе остварује се нижа вредност метрике Број одговора класе у односу на вредност исте метрике у контролној групи. Међутим, узимајући у обзир да је студијски пример који су студенти имплементирали мали, може се рећи да посматрана метрика има високу вредност. Висока вредност је постигнута због великог броја јавних метода у доменским класама. У том контексту је изведен важан закључак: у процесу развоја софтвера велику пажњу треба посветити доменским класама, у смислу правилне спецификације доменских класа, атрибута доменских класа и односа који се успостављају између доменских класа.
- У посматраним софтверским системима експерименталне и контролне групе не постоји велики недостатак кохезивности метода у класи. Већи недостатак кохезивности постоји у софтверским системима контролне групе.
- Приликом примене SilabQOSS методе остварује се виша вредност метрике Дубина стабла наслеђивања и Број подкласа у односу на вредности истих метрика у контролној групи. Ово је последица

архитектуре која дефинисана упрошћеном Ларамановом методом развоја софтвера која представља саставни део SilabQOSS методе.

На основу извршене анализе метрике стабилности софтвера могу се извести следећи закључци о примени SilabQOSS методе:

- Најстабилнији пакет у софтверском систему је пакет који садржи доменске класе. Доменским класама се дефинише структура софтверског система. Другим речима, доменске класе немају зависности од других класа. С друге стране, остале класе у софтверском систему зависе од доменских класа.
- Стабилан пакет у софтверском систему је и пакет који садржи генерички брокер базе података. Другим речима, брокер базе података има само једну зависност од интерфејса. С друге стране, све системске операције у софтверском систему зависе од брокера базе података.
- Нестабилан пакет у софтверском систему је пакет који садржи системске операције. Системским операцијама реализује се понашање софтверског система. Системске операције зависе од брокера базе података и од доменских класа. С друге стране, системске операције се позивају путем контролера апликационе логике.
- Нестабилан пакет у софтверском систему је пакет који садржи форме.
- Нестабилан пакет у софтверском систему је пакет који садржи контролер апликационе логике. Контролер апликационе логике представља фасаду ка остатку софтверског система. Он је задужен да од форме (или неке друге компоненте[30]) прихвати захтев за извршавање системске операције, проследи захтев до конкретне системске операције, прихвати резултат извршења системске

[30] Друга компонента може бити сервлет, веб сервис или нека друга технологија за удаљени позив метода (нпр. Corba, Socket, RMI, EJB итд.).

операције и врати га назад форми (или другој компоненти која је позвала контролер).

С друге стране, на основу извршене анализе резултата упитника могу се извести следећи закључци о примени SilabQOSS методе:

- Студенти су, уз један неутралан одговор, сагласни да SilabQOSS метода доприноси бољем разумевању објектно-оријентисаних концепата.
- Студенти су сагласни да SilabQOSS метода доприноси бољем разумевању стандарда и модела квалитета софтвера.
- Студенти су сагласни да SilabQOSS метода доприноси бољем разумевању софтверских метрика.
- Студенти су, уз један неутралан одговор, сагласни да SilabQOSS метода доприноси бољем разумевању алата за статичку анализу квалитета софтвера.
- Студенти су, уз један неутралан одговор, сагласни да SilabQOSS метода доприноси бољем разумевању механизама побољшања објектно-оријентисаних софтверских система.
- Студенти су сагласни да интеграција алата за статичку анализу квалитета софтвера и развојног окружења побољшава процес развоја софтвера.
- Студенти су сагласни да коришћење алата за статичку анализу квалитета софтвера омогућава брже и лакше проналажење грешака у софтверском систему.
- Студенти сагласни да се знање стечено на овај начин може искористити у процесу развоја других софтверских система.

У оквиру поглавља су, такође, идентификована ограничења евалуације:

- У процесу евалуације учествовао је мали број испитаника.
- У процесу евалуације учествовали су само студенти.

- Доменски модел и релациони модел су били једноставни.
- Имплементациони захтеви су били једноставни.

9. Закључак

Савремени софтверски системи су веома сложени: састоје се од великог броја модула и компоненти које могу бити имплементиране коришћењем различитих софтверских технологија. При томе, софтверски системи еволуирају с временом у смислу развоја нових и одржавања постојећих функционалности како би се прилагодили захтевима корисника.

У том смислу сложеност савремених софтверских система намеће потребу за добром организацијом процеса развоја софтвера. Сходно томе, у сам процес развоја софтвера потребно је инкорпорирати стандарде и процедуре како би се посматраним променама управљало на ефикасан и ефективан начин.

Другим речима, у процесу развоја софтвера потребно је применити стандарде квалитета софтвера. Због тога је у оквиру докторске дисертације дат приказ ISO/IEC 9126, ISO/IEC 14598 и ISO/IEC 25000 стандарда квалитета софтвера који се могу користити за дефинисање и евалуацију квалитета сваког софтверског система. Из изложеног се може закључити да се стандардом квалитета софтвера даје спецификација једног или више модела квалитета софтвера. С друге стране, моделом квалитета софтвера даје се спецификација атрибута квалитета софтвера (тј. карактеристика квалитета), док свака карактеристика може бити декомпонована на више подкарактеристика. С обзиром да су стандарди квалитета усмерени на интерни, екстерни и употребни квалитет софтвера и њихову евалуацију, може се закључити да стандарди квалитета софтвера обезбеђују свеобухватан поглед на квалитет софтверског система, из различитих перспектива.

Са атрибутима квалитета софтвера непосредно су повезане софтверске метрике. Оне се користе за оперативно мерење атрибута квалитета софтвера (тј. карактеристика и подкарактеристика квалитета софтвера). Софтверским метрикама се, такође, мери ниво квалитета целог софтверског система (на основу нивоа квалитета карактеристика и подкарактеристика

може се утврдити ниво квалитета целог софтверског система). У оквиру докторске дисертације представљене су најважније карактеристике софтверских метрика. Закључено је да су софтверске метрике формализоване мере које поседују математичка својства и прате промену атрибута квалитета софтвера који мере. У оквиру докторске дисертације дат је опис често коришћених софтверских метрика за модел пројектовања софтвера: *Циклична сложеност*, *Сложеност пондерисаних метода*, *Број пондерисаних метода класе*, *Број одговора класе*, *Недостатак кохезивности метода у класи*, *Повезаност објеката*, *Дубина стабла наслеђивања*, *Број подкласа*, *Број наредби у методи* и *Метрика стабилности софтвера*. Посебно је важно истаћи да је у оквиру докторске дисертације дато оригинално побољшање метрике *Метрика стабилности софтвера* с обзиром да посматрана метрика врши релативизацију долазне повезаности када не постоји одлазна повезаност, односно релативизацију одлазне повезаности када не постоји долазна повезаност.

У оквиру докторске дисертације разматрана је и веза алата за статичку анализу квалитета софтвера са стандардима квалитета софтвера. У том смислу је у дисертацији дат преглед Swat4J, SonarQube и FindBugs алата за статичку анализу квалитета софтвера који се оперативно користе у процесу евалуације квалитета софтверских система. У оквиру рада дат је и опис SilabMetrics алата за статичку анализу квалитета софтвера који је развијен у оквиру Лабораторије за софтверско инжењерство Факултета организационих наука (Универзитет у Београду). Алат је првенствено намењен учењу и упознавању основних концепата који се односе на стандарде квалитета софтвера, моделе квалитета софтвера и софтверске метрике и интегрисан је са NetBeans окружењем за развој софтвера. Након прегледа алата дефинисани су критеријуми на основу којих је извршена њихова компаративна анализа. На основу извршене компаративне анализе може се закључити да сваки алат има своје предности и недостатке и погодан је за примену у одређеним ситуацијама (уколико је, на пример,

неопходно да алат садржи модел квалитета који је заснован на неком стандарду квалитета софтвера могу се користити Swat4J, SonarQube и SilabMetrics алати; с друге стране, уколико је потребно вршити анализу квалитета софтверског система чији програмски код није доступан може се користити FindBugs алат итд.). С друге стране, на основу описа алата и извршене компаративне анализе се може закључити да примена алата за статичку анализу квалитета софтвера у процесу развоја софтвера директно доприноси повећању нивоа квалитета посматраног софтверског система.

С обзиром да су објектно-оријентисани софтверски системи нашли велику практичну примену у процесу животног циклуса развоја софтвера у оквиру докторске дисертације разматрани су различити механизми побољшања објектно-оријентисаних софтверских система и њихове везе са софтверским метрикама. Идентификовани су следећи механизми побољшања објектно-оријентисаних софтверских система:

- Општи принципи пројектовања софтвера - представљају основне принципе који се користе у процесу пројектовања софтвера. Ови принципи могу се користити приликом пројектовања свих софтверског система. У оквиру докторске дисертације разматрани су општи принципи пројектовања софтвера *апстракција*, *кохезија и повезаност*, *декомпозиција и модуларизација*, *учаурење и сакривање информација*, и *одвајање интерфејса и имплементације*, као и њихове везе са софтверским метрикама.
- Принципи објектно-оријентисаног пројектовања софтвера - представљају основне принципе који се користе приликом објектно-оријентисаног програмирања и пројектовања класа. У оквиру докторске дисертације разматрани су *принцип једне одговорности*, *принцип отворено-затворено*, *Лисков принцип замене*, *принцип издвајања интерфејса*, *принцип инверзије зависности* и *принцип додавања зависности*, као и њихове везе са софтверским метрикама.

- Стратегије пројектовања софтвера - дефинишу стратешки приступ процесу пројектовања софтвера. У оквиру докторске дисертације разматране су стратегије *подели и победи*, *с врха на доле* и *одозго на горе* и њихове везе са софтверским метрикама.
- Патерни пројектовања софтвера - дефинишу решења неког проблема у неком контексту, при чему се посматрана решења могу користити за решавање сличних проблема. У оквиру докторске дисертације разматран је општи облик софтверског патерна пројектовања и његове везе са софтверским метрикама.
- Методе развоја софтвера - методама развоја софтвера описује се процес развоја софтвера. Оне могу да користе опште принципе пројектовања софтвера, принципе објектно-оријентисаног пројектовања софтвера, стратегије пројектовања софтвера и патерне пројектовања софтвера. У оквиру докторске дисертације разматрана је упрошћена Ларманова метода развоја софтвера и њене везе са општим принципима пројектовања софтвера, принципима објектно-оријентисаног пројектовања софтвера, стратегијама пројектовања софтвера и патернима пројектовања софтвера.

У оквиру докторске дисертације развијена је сопствена SilabQOSS (енг. Silab Qualty Method for Object Oriented Systems) метода за побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера. Посматрана метода користи стандарде квалитета софтвера, софтверске метрике, алате за статичку анализу квалаитета софтвера и механизаме за побољшање објектно-оријентисаних софтверских система. У том смислу су идентификоване различите активности и различити артифакти SilabQOSS методе за побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера. На основу описа, активности и артифаката SilabQOSS методе може се извести закључак да су активности и артифакти предложене методе повезани. У том

контексту, за реализацију једне активности се користи више артифаката. С друге стране, један артифакт може бити повезан са више активности.

Како би се проверила оправданост примене предложене методе у оквиру докторске дисертације је извршена њена евалуација применом методе студије случаја. У том смислу је дефинисана студија случаја у чијој реализацији су учествовали студенти Факултета организационих наука (Универзитет у Београду) који су били подељени у две групе: експерименталну и контролну групу. Студенти експерименталне групе су у процесу имплементације користили SilabQOSS методу, док су студенти контролне групе процес имплементације спроводили без примене неке одређене методе. На основу извршене анализе се може закључити да је архитектура коју су применили студенти експерименталне групе сложенија у односу на архитектуру софтверског система коју су применили студенти контролне групе. С друге стране, студенти експерименталне групе су у процесу развоја софтвера применили генеричка решења што последично олакшава одржавање и надоградњу софтверског система.

С друге стране, на основу извршене анализе резултата упитника може се закључити да су студенти сагласни да примена SilabQOSS методе доприноси бољем разумевању објектно-оријентисаних концепата, стандарда и модела квалитета софтвера, софтверских метрика, алата за статичку анализу квалитета софтвера и механизама побољшања објектно-оријентисаних софтверских система. Поред тога, може се закључити да су студенти сагласни да се знање стечено на овај начин може искористити у процесу развоја других софтверских система.

На основу изложеног се може закључити да су у оквиру докторске дисертације остварени следећи научни доприноси:

- Дат је преглед стандарда квалитета софтвера који се могу се применити у процесу развоја и побољшања објектно-оријентисаних софтверских система.

- Извршена је класификација софтверских метрика и дат је преглед често коришћених објектно-оријентисаних софтверских метрика.
- Извршен је преглед и компаративна анализа постојећих алата за статичку анализу квалитета софтвера.
- Развијен је сопствени SilabMetrics алат за статичку анализу квалитета софтвера који се може применити у процесу развоја и побољшања објектно-оријентисаних софтверских система.
- Извршена је интеграција сопственог алата за статичку анализу квалитета софтвера и NetBeans развојног окружења.
- Дефинисани су и објашњени механизми побољшања објектно-оријентисаних софтверских система: општи принципи пројектовања софтвера, принципи објектно-оријентисаног пројектовања софтвера, стратегије пројектовања софтвера, патерни пројектовања софтвера и методе развоја софтвера.
- Успостављена је веза између механизама побољшања објектно-оријентисаних софтверских система и софтверских метрика.
- Развијена је сопствена SilabQOSS метода за побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера.

У оквиру докторске дисертације потврђене су следеће посебне хипотезе:

- Стандарди квалитета софтвера могу се применити у процесу развоја и побољшања објектно-оријентисаних софтверских система.
- Софтверске метрике се могу повезати са општим и објектно-оријентисаним принципима пројектовања софтвера, патернима пројектовања софтвера и стратегијама и методама развоја софтвера.
- Могуће је развити сопствени алат за статичку анализу квалитета софтвера који се може применити у процесу развоја и побољшања објектно-оријентисаних софтверских система.

- Сопствени алат за статичку анализу квалитета софтвера могуће је уградити у NetBeans развојно окружење.
- Могуће је развити сопствену методу за побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера и алата за статичку анализу квалитета софтвера.

С обзиром да су све посебне хипотезе успешно потврђене, потврђена је и општа хипотеза:

- Објектно-оријентисане софтверске системе могуће је унапредити коришћењем стандарда квалитета софтвера.

Евентуални будући правци истраживања могли би се односити на:

- Даљу евалуацију SilabQOSS методе за побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера у циљу добијања свеобухватнијих резултата. У том смислу могуће је извршити следећа унапређења у процесу евалуације:
    - Повећање броја учесника у процесу евалуације
    - Укључивање у процес евалуације инжењера различитих специјалности (нпр. софтвер инжењера, тестера, програмера, софтвер архитеката, инжењера који се баве осигурањем квалитета софтвера итд.), са различитим искуством (нпр. јуниори, медиори, сениори итд.)
    - Реализацију процеса евалуације која укључује сложенији доменски и релациони модел
    - Реализацију процеса евалуације која укључује већи број операција различите тежине за имплементацију
- Развој методе за побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера која се може

применити и у почетним фазама процеса развоја софтвера (нпр. у фази прикупљања захтева и фази анализе)

- Проширење SilabMetrics алата за статичку анализу квалитета софтвера које ће обезбедити подршку за друге објектно-оријентисане софтверске технологије (нпр. C#, PHP, Python итд.)
- Интеграцију SilabMetrics алата за статичку анализу квалитета софтвера са другим развојним окружењима која поседују архитектуру засновану на проширењима (енг. plugins)
- Развој метода и алата за статичку анализу квалитета софтвера који ће подржавати друге програмске парадигме (нпр. функционалну, логичку итд.)

У оквиру докторске дисертације разматрани су концепти који су веома важни за софтверско инжењерство као научну дисциплину. С друге стране, приказани концепти су веома важни и за софтверско инжењерство као инжењерску дисциплину (тј. важни су у струци). Стога сматрамо да остварени резултати представљају добру основу за даља научна и стручна истраживања како би софтверски системи постали стабилнији, једноставнији за развој, одржавање и даљу надоградњу.

10. Литература

[Abrahamsson02] Abrahamsson, P. (2002). Agile Software Development Methods: Review and Analysis (VTT publications).

[Abran03] Abran, A., Khelifi, A., Suryn, W., & Seffah, A. (2003). Usability meanings and interpretations in ISO standards. Software Quality Journal, 11(4), 325-338.

[Abreu96] e Abreu, F. B., & Melo, W. (1996, March). Evaluating the impact of object-oriented design on software quality. In Software Metrics Symposium, 1996., Proceedings of the 3rd International (pp. 90-99). IEEE.

[Alexander79] Alexander, C. (1979). The timeless way of building (Vol. 1). New York: Oxford University Press.

[Alsultanny09] Alsultanny, Y. A., & Wohaishi, A. M. (2009, December). Requirements of Software Quality Assurance Model. In Environmental and Computer Science, 2009. ICECS'09. Second International Conference on (pp. 19-23). IEEE.

[Antovic12] Antović, I., Vlajić, S., Milić, M., Savić, D., & Stanojević, V. (2012). Model and software tool for automatic generation of user interface based on use case and data model. IET Software, 6(6), 559-573.

[Ayewah08] Ayewah, N., Hovemeyer, D., Morgenthaler, J. D., Penix, J., & Pugh, W. (2008). Using static analysis to find bugs. Software, IEEE, 25(5), 22-29.

[Badri04] Badri, L., & Badri, M. (2004). A Proposal of a new class cohesion criterion: an empirical study. Journal of Object Technology, 3(4), 145-159.

[Ball99] Ball, T. (1999). The concept of dynamic analysis. In Software Engineering—ESEC/FSE'99 (pp. 216-234). Springer Berlin Heidelberg.

[Bansiya02] Bansiya, J., & Davis, C. G. (2002). A hierarchical model for object-oriented design quality assessment. IEEE Transactions on software engineering, 28(1), 4-17.

[Bevan06] Bevan, N. (2006). International standards for HCI. Encyclopedia of human computer interaction, 362.

[Boehm76] Boehm, B. W., Brown, J. R., & Lipow, M. (1976, October). Quantitative evaluation of software quality. In Proceedings of the 2nd international conference on Software engineering (pp. 592-605). IEEE Computer Society Press.

[Boehm78] Boehm, B. W., Brown, J. R., & Kaspar, H. (1978). Characteristics of software quality.

[Booch06] Booch, G. (2006). Object oriented analysis & design with application. Pearson Education.

[Bourque14] Bourque, P., & Fairley, R. E. (Eds.) (2014). Guide to the Software Engineering Body of Knowledge (SWEBOK (R)): Version 3.0. IEEE Computer Society Press.

[Briand00] Briand, L. C., Wüst, J., Daly, J. W., & Porter, D. V. (2000). Exploring the relationships between design measures and software quality in object-oriented systems. Journal of systems and software, 51(3), 245-273.

[Chidamber94] Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. Software Engineering, IEEE Transactions on, 20(6), 476-493.

[Cockburn00] Cockburn, A. (2000). Writing effective use cases. Addison-Wesley Longman Publishing Co., Inc.

[Coplien96] Coplien, J. O. (1996). Software patterns. Bell Labs, The Hillside Group.

[Crosby80] Crosby, P. B. (1980). Quality is free. Signet Book.

[Eaddy01] Eaddy, M. (2001). C# versus Java. Dr. Dobb's Journal, 26(2), 74-78.

[Emanuelsson08] Emanuelsson, P., & Nilsson, U. (2008). A comparative study of industrial static analysis tools. Electronic notes in theoretical computer science, 217, 5-21.

[Fenton00] Fenton, N. E., & Neil, M. (2000, May). Software metrics: roadmap. In Proceedings of the Conference on the Future of Software Engineering (pp. 357-370). ACM.

[FindBugs] FindBugs, http://findbugs.sourceforge.net

[Gabriel96] Gabriel, R. P. (1996). Patterns of software (Vol. 62). New York: Oxford University Press.

[Gaffney81] Gaffney Jr, J. E. (1981, January). Metrics in software quality assurance. In Proceedings of the ACM'81 conference (pp. 126-130). ACM.

[Galin04] Galin, D. (2004). Software quality assurance: from theory to implementation. Pearson Education.

[Gamma94] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design patterns: elements of reusable object-oriented software. Pearson Education.

[Gomes09] Gomes, I., Morgado, P., Gomes, T., & Moreira, R. (2009). An overview on the static code analysis approach in software

development. Faculdade de Engenharia da Universidade do Porto, Portugal.

[Gorton11] Gorton, I. (2011). Software quality attributes. In Essential Software Architecture (pp. 23-38). Springer Berlin Heidelberg.

[Harrison98] Harrison, R., Counsell, S. J., & Nithi, R. V. (1998). An evaluation of the MOOD set of object-oriented software metrics. Software Engineering, IEEE Transactions on, 24(6), 491-496.

[Henderson-Sellers96] Henderson-Sellers, B. (1996). Object-oriented metrics: measures of complexity. Prentice-Hall

[Henningsson02] Henningsson, K., & Wohlin, C. (2002, October). Understanding the relations between software quality attributes - a survey approach. In Proceedings 12th International Conference for Software Quality.

[Hitz96] Hitz, M., & Montazeri, B. (1996). Chidamber and Kemerer's metrics suite: a measurement theory perspective. Software Engineering, IEEE Transactions on, 22(4), 267-271.

[Hofer10] Hofer, T. (2010). Evaluating Static Source Code Analysis Tools (Doctoral dissertation, École Polytechnique Fédérale de Lausanne).

[Hovemeyer04] Hovemeyer, D., & Pugh, W. (2004). Finding bugs is easy. ACM Sigplan Notices, 39(12), 92-106.

[ISO14598] ISO/IEC 14598, Information technology -- Software product evaluation, http://www.iso.org

[ISO14598v1] ISO/IEC 14598-1:1999 Information technology -- Software product evaluation -- Part 1: General overview

[ISO14598v2] ISO/IEC 14598-2:2000 Software engineering -- Product evaluation -- Part 2: Planning and management, http://www.iso.org

[ISO14598v3] ISO/IEC 14598-3:2000 Software engineering -- Product evaluation -- Part 3: Process for developers, http://www.iso.org

[ISO14598v4] ISO/IEC 14598-4 Software engineering -- Product evaluation -- Part 4: Process for acquirers, http://www.iso.org

[ISO14598v5] ISO/IEC 14598-5:1998 Information technology -- Software product evaluation -- Part 5: Process for evaluators, http://www.iso.org

[ISO14598v6] ISO/IEC 14598-6:2001 Software engineering -- Product evaluation -- Part 6: Documentation of evaluation modules, http://www.iso.org

[ISO24765] ISO/IEC/IEEE 24765:2010 Systems and software engineering -- Vocabulary, http://www.iso.org

[ISO25000] ISO/IEC 25000, Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- Guide to SQuaRE, http://www.iso.org

[ISO25001] ISO/IEC 25001:2014 Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- Planning and management, http://www.iso.org

[ISO25010] ISO/IEC 25010:2011 Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- System and software quality models, http://www.iso.org

[ISO25012] ISO/IEC 25012:2008 Software engineering -- Software product Quality Requirements and Evaluation (SQuaRE) -- Data quality model, http://www.iso.org

[ISO25020] ISO/IEC 25020:2007 Software engineering -- Software product Quality Requirements and Evaluation (SQuaRE) -- Measurement reference model and guide, http://www.iso.org

[ISO25021] ISO/IEC 25021:2012 Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- Quality measure elements, http://www.iso.org

[ISO25030] ISO/IEC 25030:2007 Software engineering -- Software product Quality Requirements and Evaluation (SQuaRE) -- Quality requirements, http://www.iso.org

[ISO25040] ISO/IEC 25040:2011 Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- Evaluation process, http://www.iso.org

[ISO25041] ISO/IEC 25041:2012 Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- Evaluation guide for developers, acquirers and independent evaluators, http://www.iso.org

[ISO9126] ISO/IEC 9126, Software Engineering - Product Quality, http://www.iso.org

[ISO9126v1] ISO/IEC 9126-1:2001 Software engineering -- Product quality -- Part 1: Quality model, http://www.iso.org

[ISO9126v2] ISO/IEC TR 9126-2:2003 Software engineering -- Product quality -- Part 2: External metrics, http://www.iso.org

[ISO9126v3] ISO/IEC TR 9126-3:2003 Software engineering -- Product quality -- Part 3: Internal metrics, http://www.iso.org

[ISO9126v4] ISO/IEC TR 9126-4:2004 Software engineering -- Product quality -- Part 4: Quality in use metrics, http://www.iso.org

[Jacobson93] Jacobson, I., Christerson, M., Jonsson, P., & Övergaard, G. (1992). Object-oriented software engineering: a use case driven approach.

[Jiang08] Jiang, Y., Cuki, B., Menzies, T., & Bartlow, N. (2008, May). Comparing design and code metrics for software quality prediction. In Proceedings of the 4th international workshop on Predictor models in software engineering (pp. 11-18). ACM.

[Johnson13] Johnson, B., Song, Y., Murphy-Hill, E., & Bowdidge, R. (2013). Why don't software developers use static analysis tools to find bugs?. In Software Engineering (ICSE), 2013 35th International Conference on (pp. 672-681). IEEE.

[Kan02] Kan, S. H. (2002). Metrics and models in software quality engineering. Addison-Wesley Longman Publishing Co., Inc.

[Kitchenham96] Kitchenham, B., & Pfleeger, S. L. (1996). Software quality: the elusive target [special issues section]. IEEE software, 13(1), 12-21.

[Kruchten04] Kruchten, P. (2004). The rational unified process: an introduction. Addison-Wesley Professional.

[Larman05] Larman, C. (2005). Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development, Third Edition. Pearson Education.

[Laval08] Laval, J., Bergel, A., & Ducasse, S. (2008, October). Assessing the Quality of your Software with MoQam. In FAMOOSr 2008-2nd Workshop on FAMIX and Moose in Reengineering.

[Lethbridge05] Lethbridge, T. C., & Laganiere, R. (2005). Object-oriented software engineering. New York: McGraw-Hill.

[Letouzey12] Letouzey, J. L., & Ilkiewicz, M. (2012). Managing technical debt with the sqale method. IEEE software, (6), 44-51.

[Letouzey12b] Letouzey, J. L. (2012, June). The SQALE method for evaluating technical debt. In Proceedings of the Third International Workshop on Managing Technical Debt (pp. 31-36). IEEE Press.

[Liu10] Liu, Y., Khoshgoftaar, T. M., & Seliya, N. (2010). Evolutionary optimization of software quality modeling with multiple repositories. Software Engineering, IEEE Transactions on, 36(6), 852-864.

[Lorenz94] Lorenz, M., & Kidd, J. (1994). Object-oriented software metrics: a practical guide. Prentice-Hall, Inc..

[Martin06] Martin, R. C., & Martin, M. (2006). Agile principles, patterns, and practices in C#, Pearson Education

[Martin96] Martin, R. C. (1996). The Principles of OOD. http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

[Mayer99] Mayer, T., & Hall, T. (1999). A critical analysis of current OO design metrics. Software Quality Journal, 8(2), 97-110.

[McCabe89] McCabe, T. J., & Butler, C. W. (1989). Design complexity measurement and testing. Communications of the ACM, 32(12), 1415-1425.

[Meyer88] Meyer, B. (1988). Object-oriented software construction (Vol. 2, pp. 331-410). New York: Prentice hall.

[Milic07] Milić M. (2007). Komparativna analiza Spring i EJB tehnologije (master rad). Fakultet organizacionih nauka, Beograd, 2007.

[Milic17] Milić, M., Vlajić, S., Antović, I., Savić, D., Stanojević, V. & Lazarević, S. (2017). Software Quality Standards and Lean Approach in Teaching and Learning Programming, International Journal of Engineering Education, 33(4), 1345-1360.

[Offutt02] Offutt, J. (2002). Quality attributes of web software applications. IEEE software, 19(2), 25-32.

[Poppendieck03] Poppendieck, M., & Poppendieck, T. (2003). Lean software development: an agile toolkit. Addison-Wesley Professional.

[Pressman10] Pressman, R. S. (2010). Software engineering: a practitioner's approach. Palgrave Macmillan.

[Rosenberg97] Rosenberg, L. H., & Hyatt, L. E. (1997). Software quality metrics for object-oriented environments. Crosstalk journal, 10(4), 1-6.

[Rumbaugh91] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., & Lorensen, W. E. (1991). Object-oriented modeling and design (Vol. 199, No. 1). Englewood Cliffs, NJ: Prentice-hall.

[Rutar04] Rutar, N., Almazan, C. B., & Foster, J. S. (2004, November). A comparison of bug finding tools for Java. In Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on (pp. 245-256). IEEE.

[Schackmann09] Schackmann, H., Jansen, M., & Lichter, H. (2009). Tool support for user-defined quality assessment models. Proc. of MetriKon, 9.

[Schneidewind02] Schneidewind, N. F. (2002). Body of knowledge for software quality measurement. Computer, 35(2), 77-83.

[Seffah06] Seffah, A., Donyaee, M., Kline, R. B., & Padda, H. K. (2006). Usability measurement and metrics: A consolidated model. Software Quality Journal, 14(2), 159-178.

[Shepperd88] Shepperd, M. (1988). A critique of cyclomatic complexity as a software metric. Software Engineering Journal, 3(2), 30-36.

[Silva07] da Silva, A. R., Saraiva, J., Ferreira, D., Silva, R., & Videira, C. (2007). Integration of RE and MDE paradigms: the ProjectIT

approach and tools. IET software, 1(6), 294-314.

[Singer03] Singer, J. (2003, June). JVM versus CLR: a comparative study. In Proceedings of the 2nd international conference on Principles and practice of programming in Java (pp. 167-169). Computer Science Press, Inc.

[Sommerville04] Sommerville I. (2004). Computer Science Handbook, In Tucker, A. B. (Ed.) Chapman & Hall/CRC, 2004.

[Sommerville11] Sommerville, I. (2011). Software Engineering, Ninth Edition. Addison-Wesley.

[SonarQube] SonarQube, http://www.sonarqube.org

[Vlajic08] Влајић С., Савић Д., Станојевић В., Антовић И., Милић М. (2008). Пројектовање софтвера - Напредне Јава технологије, Златни Пресек, Београд, 2008.

[Vlajic14] Vlajic, S. (2014). Softverski proces (skripta), Univerzitet u Beogradu - Fakultet organizacionih nauka, http://silab.fon.bg.ac.rs, Beograd, 2014.

[Vlajic14b] Влајић С. (2014). Софтверски патерни, Златни Пресек, Београд, 2014.

[Vlajic15] Vlajic, S. (2015). Projektovanje softvera (skripta), Univerzitet u Beogradu - Fakultet organizacionih nauka, http://silab.fon.bg.ac.rs, Beograd, 2015.

[Vlajic16] Vlajić, S., Stanojević, V., Savić, D., Milić, M., Antović, I., & Lazarević, S. (2016). The General Form of GoF Design Patterns. World of Computer Science & Information Technology Journal, 6(2).

[Vlajic16b] Vlajić, S., Antović, I., Savić, D. (2016). General concept of the platform independency model. In Proceedings of the Symorg 2016, pp. 1253-1262.

[Vlajic17] Vlajić, S., Milić, M., Stanojević, V., Savić, D., Antović, I., Lazarević, S. & Lazović, R. (2017). Improving Robert C. Martin's Stability software metric, https://www.researchgate.net/publication/303820266_Improving_Robert_C_Martin%27s_Stability_software_metric

[Watson96] Watson, A. H., McCabe, T. J., & Wallace, D. R. (1996). Structured testing: A testing methodology using the cyclomatic complexity metric. NIST special Publication, 500(235), 1-114.

[Womack10] Womack, J. P., & Jones, D. T. (2010). Lean thinking: banish waste and create wealth in your corporation. Simon and Schuster.

11. Попис слика

12. Попис табела

Биографија

Милош Милић је рођен 26. марта 1983. године у Пожаревцу, Република Србија, где је завршио основну и средњу техничку школу. Факултет организационих наука Универзитета у Београду уписао је 2002. године. Дипломирао је на Факултету организационих наука Универзитета у Београду (смер Информациони системи и технологије) 2006. године са просечном оценом 9.47 током студија. Награђиван је од стране Факултета организационих наука и Министарства просвете и науке за постигнуте резултате у току основних студија. Дипломске академске (мастер) студије је завршио је 2007. године са постигнутом просечном оценом 10 током студија.

Од 2004. године до 2008. године ради као лаборант на Факултету организационих наука на предметима који се баве информационим системима и софтверским инжењерством. На Факултету организационих наука је у последњих неколико година држао неколико основних и виших курсева Јаве који су високо оцењени од стране полазника тих курсева. Завршио је курсеве Java IA-32 и Java IA-34 на Електротехничком факултету Универзитета у Београду, CSM (Certified Scrum Master) курс, NetBeans platform training, DSL Summer School. Задњих година се изузетно истакао у раду на међународном пројекту КОСТМОД у развоју софтверског система, који је рађен на Факултету организационих наука у оквиру Лабораторије за софтверско инжењерство. Такође, учествовао је и на пројекту анализе и унапређења пословних процеса у области „Листе чекања“ и доступности јавних података здравствених установа и њиховог повезивања, као и изради софтверских апликација за подршку доступности јавних података са сајтова здравствених установа и посебно повезивања сајтова апотекарских установа, који су рађени за потребе Републичког фонда за здравствено осигурање (РФЗО). Запослен је на Факултету организационих наука.

Списак објављених радова:

1. **Милић Милош**, Влајић Синиша, Антовић Илија, Станојевић Војислав, Савић Душан: Софтверске метрике као техника за евалуацију и побољшање квалитета софтвера, SymOrg 2008, Београд, 2008.

2. Илија Антовић, Душан Савић, Војислав Станојевић, **Милош Милић**, Синиша Влајић : Алати и методе софтверског инжењеринга по swebok пројекту, XIV конференција YU INFO, Копаоник, 2008.

3. A. Purić, **M. Milić**, I. Antović, V. Stanojević, D. Savić: A contribution to defining and determining software quality, International Conference on Education and New Learning Technologies, Barcelona, Spain, 2010.

4. А. Николић, И. Антовић, С. Влајић, **М. Милић**, Д. Савић, С. Д. Лазаревић, Компаративна анализа Hibernate и EJB технологије, Фестивал информатичких достигнућа – ИНФОФЕСТ, Будва, 2011.

5. Dušan Savic, Ilija Antović, Siniša Vlajić, Vojislav Stanojević, **Miloš Milić**, Language for Use Case Specification, 34th Annual IEEE Software Engineering Workshop, SEW 2011, Limerick, Ireland, 2011.

6. V. Stanojević, S. Vlajić, **M. Milić**, M. Ognjanović, Guidelines for framework Development process, Central and Eastern European Software Engineering Conference Russia, CEE-SECR 2011, Moscow, Russia, 2011.

7. Ристин Небојша, Влајић Синиша, Антовић Илија, **Милић Милош**, Станојевић Војислав, Компаративна анализа Java и .NET web сервиса, Info M 2011, vol. 10, br. 40, 2011.

8. **M. Milić**, S. Vlajić, I. Antović, A metric-based evaluation of Spring framework and EJB technologies, In Proceedings of The 13th International Symposium SymOrg 2012, Zlatibor 2012.

9. I Antović, S Vlajić, **M Milić**, D Savić, V Stanojević, Model and software tool for automatic generation of user interface based on use case and data model, IET software, 2012.

10. D. Savić, A. R. da Silva, S. Vlajić, S. Lazarević, V. Stanojević, I. Antović, M. Milić, Use Case Specification at Different Levels of Abstraction, QUATIC 2013, 8th International Conference on the Quality of Information and Communications Technology, Portugal, Lisabon 2012.

11. **Milić, M.**, Vlajić, S., Lazarević S., Improving Serbian healthcare system with "Find a medicine", In Proceedings of The 14th International Symposium SymOrg 2014, ISBN: 978-86-7680-295-1, Zlatibor, June 6 - 10, 2014

12. Dušan Savic, Alberto Rodrigues da Silva, Siniša Vlajić, Saša Lazarević, Ilija Antović,Vojislav Stanojević, **Miloš Milić**, Preliminary experience using JetBrains MPS to implement a requirements specification language, in Proceedings of QUATIC'2014 Conference, 2014 , IEEE Computer Society

13. **Milić, M.**, Antović, I., Savić, D., Stanojević, V., Vlajić, S., Povećanje transparentnosti podataka od javnog značaja i unapređivanje zdravstvenog sistema Srbije korišćenjem aplikacije "Pronađi lek", časopis Info-M, god. 14, sv. 51, str. , 26-32, Beograd, 2014.

14. Goran Sekulić, Ilija Antović, Siniša Vlajić, Saša Lazarević, Dušan Savić, Vojislav Stanojević, **Miloš Milić**, Conceptual Model of Software Architecture in Instruction Java Web Frameworks, International Journal of Engineering Education, Vol. 31, No. 1(A), 2015.

15. Stanojević V., Lazarević S., **Milić M.**, Comparative Analysis of UML Modeling Tools with Focus on Business Logic Specification, SYMORG 2016, Zlatibor, pp. 1272-1280, 2016.

16. Vlajić, S., Stanojević, V., Savić, D., **Milić, M.**, Antović, I., Lazarević, S., The General Form of GoF Design Patterns, World of Computer Science & Information Technology Journal, Vol. 6, No. 2, 2016.

17. **Milić, M.**, Vlajić, S., Antović, I., Savić, D., Stanojević, V., Lazarević, S., Software Quality Standards and Lean Approach in Teaching and Learning Programming, International Journal of Engineering Education, Vol. 33, No. 4, 2017.

Изјава о ауторству

Име и презиме аутора Милош Милић

Број индекса 7/2009

Изјављујем

да је докторска дисертација под насловом

Побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера

- 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/ла ауторска права и користио/ла интелектуалну својину других лица.

Потпис аутора

У Београду, 15.09.2017.

Изјава о истоветности штампане и електронске верзије докторског рада

Име и презиме аутора Милош Милић

Број индекса 7/2009

Студијски програм Информациони системи и менаџмент

Наслов рада Побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера

Ментор Проф. др Синиша Влајић

Изјављујем да је штампана верзија мог докторског рада истоветна електронској верзији коју сам предао/ла 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, 15.09.2017.

Изјава о коришћењу

Овлашћујем Универзитетску библиотеку „Светозар Марковић“ да у Дигитални репозиторијум Универзитета у Београду унесе моју докторску дисертацију под насловом:

Побољшање објектно-оријентисаних софтверских система коришћењем стандарда квалитета софтвера

која је моје ауторско дело.

Дисертацију са свим прилозима предао/ла 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/ла.

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, 15.09.2017.