UDC: 004.514 Info M str. 22-29

КОМПАРАТИВНА АНАЛИЗА КВАЛИТЕТА ПРОГРАМСКОГ КÔДА ДОБИЈЕНОГ РАЗВОЈЕМ ВОЂЕНИМ ТЕСТОМ И КОНВЕНЦИОНАЛНОМ МЕТОДОМ РАЗВОЈА
COMPARATIVE ANALYSIS OF THE PROGRAMMING CODE QUALITY DEVELOPED USING TEST DRIVEN DEVELOPMENT AND CONVENTIONAL METHOD

Сања Костић, Саша Д. Лазаревић

РЕЗИМЕ: У овом раду је паралелно приказан развој и имплементација клијент-сервер апликације методом развоја вођеног тестом и конвенционалном методом у којој се процес писања тестова обавља након написаног кôда. Користећи исту спецификацију, апликације су имплементиране на различите начине. Развој апликације методом развоја вођеног тестом почиње од креирања тестова корисничког интерфејса, а након тога следе интеграциони и јединични тестови све док се писањем кôда не дође до стања да су сви тестови успешно извршени. У свим фазама писања тестова примењена је техника "тест-имплементација-рефакторисање". Коришћењем статичких софтверских метрика извршена је анализа и поређење квалитета кôда добијеног применом обе методе. Најзначајнији аспект рада јесте утицај тестова на структуру кôда и стабилност система, док сама покривеност кôда има мању улогу. Рад обухвата и студије случаја организоване од стране софтверских компанија и приказује добијене резултате на тему мерења ефективности развоја вођеног тестом.

KLJUČNE REČI: тестирање, развој вођен тестом, квалитет кôда, софтверске метрике, имплементација, анализа

ABSTRACT: The development and implementation of the client-server application using test driven development and the conventional method in which test are written before the code are presented in parallel in this paper. Applications are implemented in different ways using the same specification. Test driven development of the application starts from the development of the user interface tests, followed by integration and unit tests development until the written code ensures that all tests have passed. Technique "test-code-refactor" is applied in all phases of tests writing process. An analysis and comparison of the code quality for both applications is done using static software metrics. The most important aspect of the paper is the impact of tests on code structure and system stability, while the code coverage has a minor role. Case studies organized by software companies are also included and the results of measuring the effectiveness of test driven development are presented.

KEY WORDS: testing, test driven development, code quality, software metrics, implementation, analysis

1. УВОД

Идеја да је потребно знати како нешто треба да ради пре креирања њега самог није се први пут јавила у софтверском инжењерству. Она је примењива у разним областима попут аутомобилске индустрије и грађевинарства, где је потребно пре развоја производа дефинисати какве ће каракертистике морати да има и каква ће све оптерећења морати да издржи. Писање тестова пре самог кôда је први пут представљен као револуционарни приступ развоју софтвера 2002. године када је амерички софтверски инжењер Кент Бек *(Kent Beck)* објавио књигу под називом "Тестом вођени развој – на основу примера"[1] и представио концепт развоја вођеног тестом.

У процесу развоја софтвера значај тестова је велики. Постојање тестова гарантује да одређени део програмског кôда ради баш онако како је и очекивано. Помоћу савремених алата извршавање тестова траје јако кратко и увек је могуће имати увид у тренутно стање система. Тестови не представљају само покривеност програмског кôда, већ могу добити и нову димензију у процесу развоја софтвера. Управо на томе се заснива идеја развоја вођеног тестом (*Test Driven Development*) – писањем тестова пре имплементације метода може се постићи потпуно разумевање корисничких захтева и истовремено бољи дизајн и структура кôда. "Водећи се овим закључцима, развој вођен тестом значајно мења начин развоја софтвера и унапређује квалитет система, тачније поузданост и флексибилност да се одговори на нове захтеве" [3].

Догађа се често да се развој вођен тестом погрешно протумачи. "Он јесте инверзија стандардног развоја у коме се тестови пишу на крају имплементације, али свакако не занемарује дизајн софтвера." [2] Често се повлачи паралела између конвенционалног развоја софтвера и развоја вођеног тестовима. Питање која од ове две методе развоја даје боље резултате у погледу структуре и квалитета кôда је често постављано. Неминовно је да ће тестом вођен развој имати већу покривеност кôда, међутим уколико систем има лош дизајн, покривеност кôда неће пуно значити. С обзиром да се развој вођен тестом заснива на понављању малог развојног циклуса и да се рефакторисање кôда посматра као могућност након сваког написаног дела кôда, ова метода може пружити велике могућности за писање чистог и квалитетног кôда. Може се навести велики број разлога за и против коришћења ове методе, али је сигурно да уколико се примењује на правилан начин може имати велики утицај на квалитет софтвера. Циљ овог истраживања је доћи до одговора на питања попут: Која од наведене две методе развоја, када се користе у реалним системима, даје боље резултате у погледу стабилности система? Помоћу које методе се лакше уводе нове функционалности у систем или модификују постојеће? Помоћу које методе се правилно развије већи број случајева коришћења? Да ли тестабилност апликације има утицаја на сам квалитет?

[1] **Beck, Kent**. *Test Driven Development: By Example*. Addison-Wesley Professional, 2002

2. СОФТВЕРСКЕ МЕТРИКЕ

Како би се дошло до мерљивих резултата по питању комплексности кôда и међузависности класа коришћене су софтверске метрике. "Софтверске метрике представљају квантитативне методе за оцену квалитета софтвера"[15]. Коришћењем софтверских метрика попут цикличне комплексности, куповања, недостатка кохезије итд. може се доћи до процене квалитета кôда са аспекта дизајна и структуре.

2.1 *Циклична комплексност:*

Једна од најчешће коришћених статичких метрика која директно указује на сложеност кôда јесте циклична комплексност. "Настала је 1976. године од стране Мекејба (*Thomas J. McCabe*), и представља укупан број логичких одлучивања у оквиру софтверског модула. Софтверски модул је функција или потпрограм који има једну улазну и једну излазну тачку." [4]

Може се рећи да је циклична комплексност једнака броју линеарно независних путања у одређеном делу кôда. Логичка структура кôда се графички може приказати усмереним графом. Сходно томе, циклична комплексност за израчунавање комплексности кôда користи граф контроле тока, који се састоји из чворова и грана. Чворовима се представљају најмање групе команди у програму, док је усмереним гранама приказан пренос контроле извршавања између два чвора. Свака могућа путања кроз кôд одговара једној путањи од улазног до излазног чвора у графу контроле тока. За израчунавање цикличне комплексности користи се следећа формула [5]:

$$CC = E - N + 2P$$

(***E*** = број грана у графу, ***N*** = број чворова у графу, ***P*** = број повезаних компоненти)

Бројне студије истраживале су међузависност цикличне комплексности и броја дефеката у функцији или методи. "У многима од њих дошло се до закључка да је веза између ова два параметра јака, односно да функције или методе са већим индексом цикличне комплексности имају тенденцију да садрже и већи број дефеката. Међутим не може се са сигурношћу тврдити и обрнуто тј. да ће смањењем цикличне комплексности доћи и до смањења броја дефеката у систему."[4]

2.2 *Куповање*

"Куповање (спојеност) је мера везаности класе са другим класама, којом се утврђује колико је јако класа повезана са другим класама (зависна од њих). Класа са високом (*high coupling*) или снажном (*strong coupling*) повезаношћу зависи од других класа."[6]

Еферентно куповање мери међузависност класа. Уколико се мери на нивоу класе онда представља број класа од којих посматрана класа зависи. "Класе са вишим еферентним куповањем ће испољити резултат промена или дефеката насталих у другим класама."[7] "Еферентно куповање пакета дефинише број класа у посматраном пакету које зависе од класа у другим пакетима."[8]

Аферентно куповање означава број класа које зависе од посматране класе. "Класе са високим аферентним куповањем ће своје измене испољити на другим класама."[7] Висока вредност овог параметра указује на јако стабилну компоненту. "Аферентно куповање може се измерити и на нивоу пакета. У том случају показује осетљивост преосталих пакета на промене у анализираном пакету." [8]

2.3 *Нестабилност*

Нестабилност је метрика која се користи за мерење релативне осетљивости класе на промене. По дефиницији једнака је односу еферентног и укупног куповања (броју одлазних зависности према броју укупних зависности). Вредност овог параметра креће се у границама између 0 и 1. Уколико класа или пакет има вредност нестабилности 0, то значи да је класа потпуно стабилна, односно не зависи ни од чега и све зависи од ње. Нестабилност једнака јединици означава потпуну нестабилност класе.

2.4 *Недостака кохезије метода*

"Кохезија је мера којом се утврђује колико су методе унутар класе међусобно повезане. Класе треба изградити тако да имају већу кохезију." [6] Класа са високом кохезијом извршава једну функцију, док класа са ниском кохезијом извршава две или више неповезаних функција. "Недостатак кохезије значи да је класа одговорна да обезбеди више различитих понашања која нису између себе повезана. Таква класа је тешка за одржавање и надоградњу." [6] Вредности се крећу између 0 и 1. Ниска вредност овог параметра (ближа нули) идентификује кохезивну класу. Уколико је вредност ближа јединици, то означава класу са недостаком кохезије у методама и директно указује на то да је класу потребно поделити у више мањих и независних подкласа.

2.5 *Дубина наслеђивања*

Дубина наслеђивања је метрика која се рачуна на нивоу класе и показује максималну дужину од корена стабла до чвора стабла које представља дата класа. Анализа ове метрике има смисла уколико у апликацији постоји вишеструко наслеђивање. "Дубина наслеђивања показује колико родитељских класа могу потенцијално утицати на анализирану класу." [9] Дубока стабла се обично везују за сложен дизајн. Што је класа дубље у хијерархији, наследиће више метода и атрибута, па ће се самим тим и њена комплексност повећати. Позитивна страна је та што ће се постојеће методе поново употребити и то добро утиче на перформансе система.

2.6 *Индекс специјализације*

Индекс специјализације је метрика која утврђује степен у ком подкласа мења (преклапа) понашање родитељских класа (надкласа). "Уколико класа мења понашање великог

броја метода своје надкласе, то може указивати на потенцијално неодговарајућу апстракцију. Подкласе би требало да надограде понашање које је дефинисано у надкласи коју наслеђују додавањем нових метода, а не да у потпуности редефинишу или бришу постојеће понашање. Индекс специјализације указује на квалитет уведене апстракције." [10]

2.7 *Број метода класе*

Укупан број метода**,** као што и само име каже, дефинише укупан број метода по класи. Свака класа треба имати одређени број метода који не сме бити превелики. "Препоручене вредности овог параметра су између 3 и 7. Вредност већа од седам може указати на потребу за даљом објектно-оријентисаном декомпозицијом. Вредност мања од 2 показује да то није класа већ конструкција података." [11]

Број преклопљених метода показује укупан број метода подкласе у којима је промењено понашање дефинисано у надкласи. "Превелики број преклопљених метода указује на велику разлику у односу на родитељску класу и тада наслеђивање има све мање смисла" [11]

2.8 *Број линија кôда*

Укупан број линија кôда представља квантитативну меру физичких линија кôда. У укупан збир улазе све линије кôда које имају неко значење. Празне линије или закоментарисан кôд неће бити урачунати.

Број линија кôда метода представља број физичких линија активног кôда који се налази унутар метода. "Дужина методе се користи при процени разумљивости, одрживости и могућности поновног коришћења кôда. Препоручен број линија кôда метода зависи од програмског језика који се користи."[10]

3. СТУДИЈСКИ ПРИМЕР

Потребно је направити апликацију за бележење резервација соба гостију хотела описану следећим корисничким захтевима:

Свака резервација има статус који мора бити видљив клијенту. Када клијент покрене систем резервација је у статусу „нова". Клијент уноси датуме у оквиру којих жели да резервише собу, као и број особа које ће боравити у изабраној соби. Након унетих података, клијент добија информацију да је резервација одобрена или одбијена. У случају да је резервација одобрена добија се информација о доступној соби у том периоду, као и цени резервације. Клијент може да одабере начин на који жели да плати резервацију. Једна могућност је готовинско плаћање, при чему цена остаје непромењена. Други начин је плаћање на рате. Клијент може да одабере на колико рата жели да уплати резервацију, при чему се са повећањем броја рата повећава и укупна цена резервације. Уколико је резервација успешно сачувана, она добија статус "резервисана". Резервацију је могуће платити. У случају готовинског плаћања резервација се плаћа у потпуности и преостали дуг клијента је 0 РСД. Статус постаје "плаћена" .У случају плаћања на рате, клијент може да плати одређени износ за одређени датум, при чему укупан збир свих уплата мора бити једнак цени резервације. Могуће је уплатити онолико пута колико је рата изабрано при креирању резервације.

3.1 *Развој апликације конвенционалном методом развоја*

Прва апликација биће развијена стандардном метдом развоја. То подразумева писање кôда, а након имплементиране функционалноти и писање тестова који ће проверавати креиране методе (*test-last*). С обзиром да се у овом раду акценат не ставља на покривеност кôда тестовима, за прву апликацију неће бити креирани тестови након имплементације функционалности.

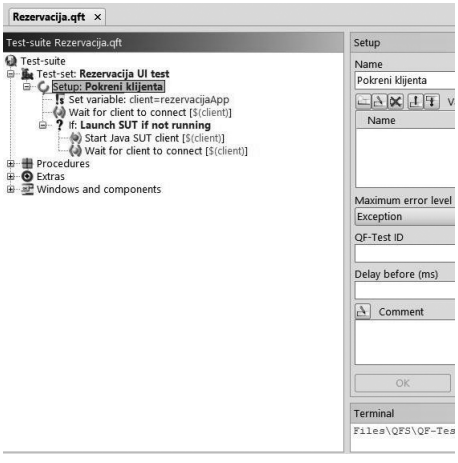
На основу дефинисане спецификације серверска страна биће имплементирана тако да обавља све потребне операције над доменским објектима. Класа *RezervacijaPomocnik* је нова класа коју ће инстанцирати нит *RezrvacijaServis* и користиће се за имплементацију конкретних метода. Ове методе биће позиване од стране нити у зависности од параметра трансферног објекта који се добија преко улазног тока са клијентске стране

3.2 *Развој апликације методом тестом вођеног развоја*

На основу спецификације започеће развој апликације коришћењем методе развоја вођеног тестом. Имплементација ће почети од креирања тестова корисничког интерфејса, а настављена са јединичним тестирањем. Принцип "тест-кôд-рефакторисање" биће примењен на све функционалности.

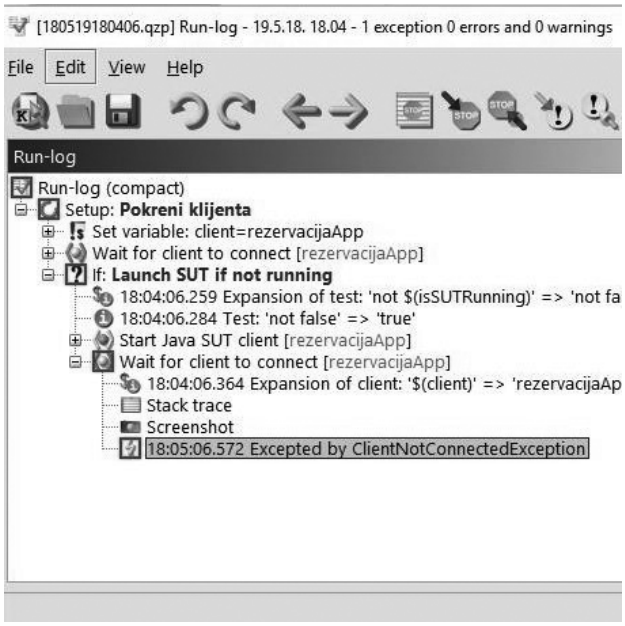
3.2.1 Имплементација корисничког интерфејса

Тест - Прво ће бити креиран тест графичког корисничког интерфејса помоћу алата *QF-test*. За почетак је потребно покренути клијентски део апликације. У те сврхе биће креиран сет тестова (*test-set*) који ће садржати све тестове неопходне за адекватан рад ове апликације. Покретање клијента се врши у делу "*setup*"



(Слика 1)

Ово је први тест који је креиран за потребе апликације и он осигурава да се покретање апликације увек успешно извршава. С обзиром да графички кориснички интерфејс апликације још увек није креиран, након покретања овог теста очекивано је понашање да тест буде неуспешан.



(Слика 2)

Након покретања теста, добијена је грешка која наводи да клијент није успешно конектован. У делу "*Wait for client to connect*" одређено је да се на повезивање клијента чека укупно 60 секунди. Уколико се клијент из било ког разлога не конектује у том временском интервалу, наведена грешка ће бити приказана. У овом конкретном случају кôд за клијентски део апликације још увек није развијен па је добијена грешка и очекивана.

Кôд - У клијентском пројекту додат је нов пакет под називом "*ui*" и у њему креирана нова класа *GlavnaForma* која наслеђује *JFrame.*

```
public class GlavnaForma extends JFrame {
    private JPanel contentPane;

    public static void main(String[] args) {
        EventQueue.invokeLater(new Runnable() {
            public void run() {
                try {
                    GlavnaForma frame = new GlavnaForma();
                    frame.setVisible(true);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        });
    }
}
```

Након ове имплементације поновним покретањем креираног теста добија се следећи резултат:

(Слика 3)

Рефакторисање- У методи тестом вођеног развоја након сваког написаног дела кôда потребно је одрадити рефакторизацију уколико се уочи било каква могућност за унапређење кôда. У овом случају није потребно радити рефакторисати написан кôд па се овај корак може занемарити.

Следећи овај пример, све компоненте графичког корисничког интерфејса развијене су на исти начин. Добијени тестови гарантују да све компоненте корисничког интерфејса раде на одговарајући начин и усклађено.

3.2.2 Имплементација серверске логике

Тест - Свака резервација која има најмање једну доступну собу може се сматрати одобреном. Провера статуса резервације биће издвојена у посебну методу у оквиру класе *RezervacijaPomocnik.*

```
public Rezervacija proveriRezervaciju (Rezervacija rezervacija){

        return null;

}
```

Ова метода прима резервацију као улазни параметар и враћаће исту резервацију са измењеним подацима. Водећи се принципима ТДД-а, биће потребно креирати тест који проверава ситуацију када резервација има статус "одобрена". У делу *given* креира се нова резервације са статусом "нова" и са бројем особа 7. С обзиром да соба са капацитетом за 7 особа постоји и да нема ни једне друге резервације, очекивано је да новокреирана резервација буде одобрена.

```
@Test
public void proveriRezervacijuTESTproveriStatusOdobreneRezerva-
cije () throws ParseException {
        // Given
   Rezervacija rez = new Rezervacija("nova",
                new Date(sdf.parse("10.10.2018").getTime()),
       new Date(sdf.parse("20.10.2018").getTime()), 7);
   // When
   rez = sut.proveriRezervaciju(rez);
   // Then
   assertThat(rez.getStatus(), equalTo("odobrena"));
   assertThat(rez.getSoba(), equalTo(s));
}
```

Након позива методе која се тестира, у делу *then* проверава се статус резервације, а потом и да ли се одговарајућа соба исправно додаје на саму резервацију. Покретањем овог теста резултат ће бити негативан с обзиром да метода која се тестира још увек нема одговарајућу имплементацију.

Кôд - Метода *proveriRezervaciju (Rezervacija rezervacija)* биће имплементирана тако да најпре позива методу *proveriOdgovarajuceSobe(),*а затим уколико добијена листа није празна мења се статус резервације и резервацији се додељује прва соба из листе.

```
public Rezervacija proveriRezervaciju (Rezervacija rezervacija){
        ArrayList<Soba> listaSoba = pronadjiOdgovarajuceSobe(re
zervacija);
        if (listaSoba.size() > 0) {
                rezervacija.setSoba(listaSoba.get(0));
                rezervacija.setStatus("odobrena");
        }
        return rezervacija;
}
```

Софтверска метрика	Ниво	Стандардни развој		Развој вођен тестом		Препоручене вредности
		укупно	просек	укупно	просек	
Циклична комплексност	пројекат сервер	13	3.6	6	2.2	**< 10**
	пројекат клијент	3	1.52	3	1.47	**< 10**
Ефферентно куповање	пакет *servis*	2	-	2	-	**< 20**
Аферентно куповање	пакет *servis*	1	-	4	-	**< 500**
Нестабилност	пакет *servis*	0.667	-	0.333	-	**0**
Недостатак кохезије	пројекат сервер	0.8	0.367	0.786	0.189	**0**
Дубина наслеђивања	пројекат сервер	2	-	3	-	**< 5**
	пројекат клијент	6	-	6	-	**< 5**
Укупан број метода	пројекат сервер	13	3.25	28	3.5	-
	Rezervacija Servis	7	-	15	-	**3-7**
Број метода преклапања	пројекат сервер	2	-	2	-	**< 5**
Индекс специјализац.	пројекат сервер	-	0.321	-	0.142	**< 0.15**
	Rezervacija Servis	0.286	-	0.076	-	**< 0.15**
Број линија кода	пројекат сервер	259	-	325	-	-
Број линија кода метода	пројекат сервер	140	-	167	-	-

(Слика 4)

Рефакторисање- У овом случају нема простора за додатним побољшањем кôда. Процес рефакторизације може се односити и на имплементиран кôд методе и на тест.

На овај начин имплементиране су све методе апликације. Коришћени су параметризовани тестови, а кôд је рефакторисан сваки пут када је било потребе. У процесу рефакторизације примењен је *decorator* патерн на део кôда који рачуна цену резервације и *builder* патерн у параметризованом тесту.

4. АНАЛИЗА

За анализу кôда коришћено је укупно 11 метрика. У следећој табели упоредно су приказани добијени резултати по свакој метрици у обе апликације. У колони "укупно" приказане су максимано достигнуте вредности на новоу пројекта/класе, док колона "просек" садржи просечне вредности параметара.

Софтверска метрика	Ниво	Стандардни развој		Развој вођен тестом		Препоручене вредности
		укупно	просек	укупно	просек	
Циклична комплексност	пројекат сервер	13	3.6	6	2.2	**< 10**
	пројекат клијент	3	1.52	3	1.47	**< 10**
Ефферентно куповање	пакет *servis*	2	-	2	-	**< 20**
Аферентно куповање	пакет *servis*	1	-	4	-	**< 500**
Нестабилност	пакет *servis*	0.667	-	0.333	-	**0**
Недостатак кохезије	пројекат сервер	0.8	0.367	0.786	0.189	**0**
Дубина наслеђивања	пројекат сервер	2	-	3	-	**< 5**
	пројекат клијент	6	-	6	-	**< 5**
Укупан број метода	пројекат сервер	13	3.25	28	3.5	-
	Rezervacija Servis	7	-	15	-	**3-7**
Број метода преклапања	пројекат сервер	2	-	2	-	**< 5**
Индекс специјализац.	пројекат сервер	-	0.321	-	0.142	**< 0.15**
	Rezervacija Servis	0.286	-	0.076	-	**< 0.15**
Број линија кода	пројекат сервер	259	-	325	-	-
Број линија кода метода	пројекат сервер	140	-	167	-	-

(Слика5)

Вредности означене црвеном бојом показују да су граничне вредност метрика прећене. Прва таква ситуација је вредност цикличне комплексности у апликацији развијеној конвенционалним развојем. Овај број указује на то да су методе писане тако да задовоље функционалну целину, и самим тим садрже велики број одлучивања. Супротно томе, у апликацији развијеној принципима тестом вођеног развоја, свака метода морала је имати тест пре саме имплементације, па је то довело до већег броја мањих метода. Због тога циклична комплексност друге апликације износи 6. Управо из истог разлога се значајно разликују вредности метрике која проверава укупан број метода. У апликацији развијеној стандарном методом развоја, овај параметар на нивоу пројекта износи 13, док је у другој тај број више него дупло већи – 28. Међутим, уколико се провери класа са највећим бројем метода, долази се до резултата да у првој апликацији класа *RezervacijaServis* има 7 метода, док у другој има укупно 15. Ова вредност драстично прелази дефинисане границе од 3-7 метода по класи. Разлог ове ви-

соке вредности јесте потреба тестирања саме класе која је уједно нит. Све линије кôда које комуницирају са класом *RezervacijaPomocnik* издвојене су у посебне методе како би се тестирало да ли долази до позива одговарајућих метода током извршавања нити.

Поређењем вредности за нестабилност пакета јасно се уочава да апликација развијена конвенционалним развојем има дуплло већи резултат. Ова метрика директно зависи од аферентног и еферентног купловања, па је самим тим што је већа вредност афретентног купловања у другој апликацији постигнута и већа стабилност.

Максимални недостатак кохезије метода постигнут у обе апликације има приближно једнаке вредности од око 0.8. Ово означава да постоји класа са јако ниском кохезијом метода. Поређењем просечне вредности недостатка кохезије на нивоу целог пројекта, долази се до вредности од 0.367 за апликацију развијену стандардним развојем и 0.189 за апликацију развијену ТДД-ем. Може се закључити да друга апликација има просечно већу кохезију класа у односу на прву иако и даље има простора за побољшањем овог параметра.

Обе апликације у клијентском пројекту достижу дубину наслеђивања 6 и то услед коришћења класа које наслеђују *JDialog* који је већ у хијерерхији *swing* компоненти. Што се серверске стране тиче, максимална дубина наслеђивања износи 2 и 3 што спада у оквире препоручених вредности.

Индекс специјализације проверен је на нивоу класе *RezervacijaServis,* као и на нивоу серверског пројекта. *RezervacijaServis* показује значајно другачије вредности ове метрике поредећи обе апликације. У првој она износи 0.286, што је свакако изнад препоручене границе од 0.15. У апликацији развијеној тестом вођеним развојем индекс специјализације износи 0.076, због већег броја метода дефинисаних у овој класи. Ситуација је слична и на нивоу пројекта, с тим да су обе вредности незнатно повећане.

Применом софтверских метрика на ове две апликације добијени су резултати који указују на то да се коришћењем развоја вођеног тестом значајно може утицати на квалитет кôда. Вредности метрика показују да је апликација развијена ТДД-ем стабилнија, да има мање купловање и већу кохезију, као и то да су методе мање комплексне. Што се недостатака тиче, показано је да развој вођен тестом доводи до већег броја линија кôда, како укупних тако и линија кôда метода, а такође и до већег броја метода у комплекснијим класама.

Покривеност кôда тестовима није централна тема овог рада, али свакако вреди напоменути да тестом вођени развој, уколико се примењује на правилан начин, осигурава да свако понашање методе има тест који ће у сваком тренутку гарантовати да метода ради управо онако како је и очекивано да ради. У овој апликацији пакет који саджи тестове има укупно 392 линије кôда, од чега је 174 линија кôда метода. Ове вредности превазилазе бројеве линија кôда активних класа. Највећи бенефити постојања тестова уочени су у кораку рефакторизације кôда. Иако је рефакторисање стандардни део тестом вођеног развоја, постојање тестова осигурава да измена није нарушила текућу као ни постојеће фунцкионалности.

5. СЛИЧНА ЕМПИРИЈСКА ИСТРАЖИВАЊА

Коришћењем софтверских метрика попут цикличне комплексности, купловања, недостатка кохезије итд. може се доћи до процене квалитета кôда са аспекта дизајна и структуре. Међутим, ове метрике неће дати одговор на питање квалитета софтвера у погледу поузданости, ефикасности, лакоће коришћења, одрживости итд. Уколико се посматра број дефеката као мера квалитета софтвера, софтверске метрике, такође, неће дати резултате. У развоју софтвера регресионо тестирање даје одговор на питање да ли је имлементацијом нових функционалности или модификацијом постојећег кôда уведен нови дефект у систем. Избором софтверских метрика за проверу квалитета кôда не може се проверити број укупних дефеката нити број новонасталих.

У наставку текста биће дат опис три емпиријска истраживања и резултата који су у њима постигнути. Сви експерименти рађени су у више група, од којих је једна развијала софтвер коришћењем развоја вођеног тестом, а друга коришћењем традиционалног начина развоја.

5.1 *Компаративана студија случаја утицаја развоја вођеног тестом на дизајн програма и покривеност тестовима*

Истраживање је рађено је у Финској 2007. године у оквиру техничког истраживачког центра Финске. [12] Основни циљ компаративне емпиријске евалуације тестом вођеног развоја био је да се истражи утицај ТДД-а на дизајн софтвера и на покривеност кôда тестовима. Истраживање је обављено над три пројеката развоја софтвера. За два пројекта коришћен је итеративан традиционалан приступ развоју, док је за последњи коришћен тестом вођени развој. За поређење резултата коришене су стандардне софтверске метрике објектно-оријентисаног дизајна.

Резултати купловања показали су да класе пројекта развијеног тестом вођеним развојем имају мање купловање, али такође је уочено да пројекат развијен ТДД-ем обухвата највећи распон вредности купловања. Међутим, сви пројекти имали су јако ниске вредности купловања, па се не може са сигурношћу рећи да је трећи пројекат имао најбоље резултате искључиво због технике развоја.

Недостатак кохезије показао је другачије резултате, односно трећи пројекат имао је најмању кохезију метода. Уочено је да су резултати најконзистентнији у трећем пројекту што указује да лоша кохезија није случајност. Без обзира да ли је до овог резултата дошло услед коришћења тестом вођеног развоја или мањка искуства програмера, сигурно је доказано да ТДД сам по себи не гарантује високу кохезију.

Остале метрике нису показале значајне разлике између пројеката. Покривеност кôда тестовима мерена је коришћењем покривености метода (*method coverage)*, покривености исказа (*statement coverage)* и покривености грана (*branch coverage).* Сва три параметра имала су значајно

боље вредности у пројекту на ком се користио развој вођен тестом. Важно је нагласити да се коришћењем ТДД-а постигла значајно боља покривеност кôда тестовима а да то није утицало на продуктивност програмера и да је производ био испоручен на време. [12]

5.2 *Процена ефикасности тестом вођеног развоја*

Истраживање на тему ефикасности ТДД-а обављено је у *Microsoft* компанији, у оквиру две дивизије: *Windows* - пројекат А и *MSN* - пројекат Б. [13] Оба пројекта су била анализирана након завршетка, а програмери током развоја софтвера нису били упознати са тим да ће постигнути резултати бити мерени. Квалитет кôда провераван је на основу густине дефеката, односно броја дефеката у сваких 1,000 линија кôда.

У оба пројекта се показало да је кôд настао развојем вођеним тестом имао значајно мање дефеката. Тимови који су радили развој апликације стандардном методом остварили су 2.6 пута већи број дефеката на пројекту А и чак 4.2 пута већи број дефеката на пројекту Б у односу на тимове који су користили ТДД. Међутим, за оба пројеката тимовима је било потребно више времена за развој уколико су користили развој вођени тестом.

У овом истраживању проверена је ефикасност тестом вођеног развоја у погледу квалитета софтвера и продуктивности (времену потребном за развој). [13]

5.3 *Утицај примене развоја вођеног тестом: Студија случаја*

Истраживање је спроведено 2007. године на Универзитету у Мисисипију. [14] Формиране су две групе од по 9 студената, који су били су обучени за примену обе методе развоја софтвера, конвенционалне и ТДД-а.

Након завршетка пројеката проверени су резлтати следећих параметара: број написаних случајева тестирања, број дефеката пронађен од стране тима за осигурање квалитета покретањем јединичних тестова након сто су јединични тестови завршени од стране програмера, број дефеката пронађених након интеграционог тестирања, број дефеката након тестова прихватања, и на крају број потрошених сати по члану тима.

Добијени резултати показали су да развој вођен тестом производи значајно мање дефеката у поређењу са стандардним развојем. Број дефеката у јединичном тестирању се значајно разликује у ове две методе, док је број дефеката у интеграционом тестирању приближно једнак. На основу приложених резултата дошло се до закључка да су квалитет софтвера и продуктивност програмера значајно побољшани коришћењем ТДД-а. Сматра се да је развој вођен тестом допринео продуктивности јер су функционалности недвосмислено биле имплементиране. Уједно, више времена је уторшено на дораду кôда него у конвенционалном развоју.

6. ЗАКЉУЧАК

Један од аспеката овог рада јесте приближавање методе развоја вођеног тестом читаоцу. Детаљно је описан процес писања тестова пре имплементације на конкретном примеру. Објашњено је на који начин се може имплементирати апликација од креирања корисничког интерфејса до извршавања операција над доменским објектима. Приказани су различити начини тестирања и њихов утицај на дизајн софтвера. Главну улогу у развоју апликације ТДД-ем имао је корак рефакторизације који се обавља на крају сваког развојног циклуса. Он приморава програмера на анализу написаног кôда и евентуално побољшање при чему осигурава да ни једна до тада имплементирана функционалност неће бити нарушена додатним изменама.

Квалитет софтвера може се дефинисати на више начина. Један до њих јесте квалитет кôда у смислу структуре и дизајна. Како би се добили мерљиви резултати за поређење апликација коришћење су софтверске метрике. Применом софтверских метрика добили су се резултати везани за комплексност и међузависност класа у апликацијама. Значајна разлика уочена је приликом поређења вредности цикличне комплексности. Апликација развијена конвенционалном методом имала је дупло већу комплексност у односу на апликацију у којој је примењен ТДД, док је истовремено имала дупло мањи број линија кôда. Показано је да тестом вођени развој доприноси мањем купловању и већој кохезији класа. На тај начин приказан је утицај различитих врста тестова на квалитет и дизајн кôда.

Паралелном анализом апликација показано је да апликација развијена тестом вођеним развојем има боље вредности метрика у односу на апликацију развијену стандарним развојем. Добијен је тестабилнији кôд, мање комплексан и са бољом структуром. Неминовна је чињеница да примена развоја вођеног тестом захтева више времена, али уједно даје и више добробити.

Развој вођен тестом се практично ретко примењује, најчешће због недостатка времена или неправилно усвојене методе, па је сврха овог рада и приближавање ТДД-а на начин да су користи од примене развоја вођеног тестом знатно веће од напора који је потребан да се овај метод правилно усвоји. Сматрам да ће се развој вођен тестом у будућности све више примењивати и да ће значајно доприности тестирању и квалитету софтвера.

7. ЛИТЕРАТУРА

[1] **Beck, Kent**. *Test Driven Development: By Example*. Addison-Wesley Professional, 2002.

[2] **Farcic, Victor and Garcia, Alex.** *Test-Driven Java Development.* Birmingham : Packt Publishing Ltd., 2015.

[3] **Freeman, Steve and Pryce, Nat.** *Growing Object-Oriented Software, Guided by Test.* Boston : Pearson Education, Inc., 2010.

[4] **Thomas J. McCabe, Arthur H. Watson.** McCabe Software. [августа 2018.год. приступано] http://www.mccabe.com/pdf/mccabe-nist235r.pdf.

[5] TutorialsPoint. *TutorialsPoint / Cyclomatic Complexity.* [августа 2018.год. приступано] https://www.tutorialspoint.com/software_testing_dictionary/cyclomatic_complexity.html
[6] **Влајић, Синиша.** Софтверски процес (скрипта), Београд, 2016.
[7] **Erickson, Aaron.** Using Metrics to Find Out if Your Code Base Will Stand the Test of Time. *http://www.informit.com/articles/article.aspx?p=1561879&seqNum=3.* [августа 2018.год. приступано]
[8] Object-oriented metrics by Robert Martin. *https://www.future-processing.pl/blog/object-oriented-metrics-by-robert-martin/.* [августа 2018.год. приступано]
[9] **Frederick T. Sheldon, Kshamta Jerath, Hong Chung.** Metrics for maintainability of class inheritance hierarchies. *https://www.csm.ornl.gov/~sheldon/public/Sheldon-smr249.pdf.* [августа 2018.год. приступано]
[10] **Rodriguez, Daniel / Harrison, Rachel.** An Overview of Object-Oriented Design Metrics. *http://www.cc.uah.es/drg/b/RodHarRama00.English.pdf.* [августа 2018.год. приступано] 11. Objecteering Metrics User Guide. *http://support.objecteering.com/objecteering6.1/help/us/metrics/toc.htm* [августа 2018.год. приступано]
[12] **Siniaalto, Maria and Abrahamsson, Pekka.** A Comparative Case Study on the Impact of Test-Driven Development on Program Design and Test Coverage. *https://arxiv.org/ftp/arxiv/papers/1711/1711.05082.pdf. 2007* [септембра 2018.год. приступано]
[13] **Thirumalesh Bhat, Nachiappan Nagappan.** Evaluating the Efficacy of Test-Driven Development: Industrial Case Studies. *http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.187.9671&rep=rep1&type=pdf* [септембра 2018.год. приступано]
[14] **Yenduri, Sumanth and L.A. Perkins.** Impact of Using Test-Driven Development: A Case Study. *https://www.researchgate.net/publication/221610913_Impact_of_Using_Test-Driven_Development_A_Case_Study* 2007 [септембра 2018.год. приступано]
[15] **Mrinal Kanti Debbarma, Swapan Debbarma, Nikhil Debbarma, Kunal Chakma, Anupam Jamatia.** A Review and Analysis of Software Complexity Metrics in Structural Testing, http://www.ijcce.org/papers/154-K271.pdf, 2013 [септембра 2018.год. приступано]

Сања Костић, Универзитет у Београду - Факултет организационих наука, студент мастер студија
Контакт: kostic.n.sanja@gmail.com
Област интересовања: софтверско инжењерство, тестирање софтвера

Проф. др. Саша Д. Лазаревић, Факултет организационих наука, Београд
Контакт: slazar@fon.rs
Област интересовања: софтверско инжењерство, информациони системи, базе података, системи за управљање документацијом, .NET платформ

infoM

UPUTSTVO ZA PRIPREMU RADA

1. Tekst pripremiti kao Word dokument, A4, u kodnom rasporedu 1250 latinica ili 1251 ćirilica, na srpskom jeziku, bez slika. Preporučen obim – oko 10 strana, single prored, font 11.
2. Naslov, abstakt (100-250 reči) i ključne reči (3-10) dati na srpskom i engleskom jeziku.
3. Jedino formatiranje teksta je normal, bold, italic i bolditalic, VELIKA i mala slova (tekst se naknadno prelama).
4. Mesta gde treba ubaciti slike, naglasiti u tekstu (Slika1...)
5. Slike pripremiti odvojeno, VAN teksta, imenovati ih kao u tekstu, radi identifikacije, u sledećim formatima: rasterske slike: jpg, tif, psd, u rezoluciji 300 dpi 1:1 (fotografije, ekranski prikazi i sl.), vektorske slike – cdr, ai, fh,eps (šeme i grafikoni).
6. Autor(i) treba da obavezno priloži svoju fotografiju (jpg oko 50 Kb), navede instituciju u kojoj radi, kontakt i 2-4 oblasti kojima se bavi.
7. Maksimalni broj autora po jednom radu je 5.

Redakcija časopisa Info M