UDC: 004.4'416:004.42.045 INFO M: str. 46-54

TEST DRIVEN DEVELOPMENT УЗОРИ И РЕФАКТОРИСАЊЕ ТЕСТНОГ КОДА
TEST DRIVEN DEVELOPMENT PATTERNS AND TEST CODE REFACTORING

Слободан Мирковић - ФОН Београд, Саша Д. Лазаревић - ФОН Београд

РЕЗИМЕ: Test driven development (TDD) је процес развоја софтвера настао као део Екстремног програмирања, да би га касније усвојиле све агилне методологије. Главна тачка у агилним методологијама је софтвер који ради, а TDD је процес који је фокусиран на производњу поузданог софтвера који се лако одржава. TDD мења улогу коју тестирање има у процесу развоја софтвера и не служи више само да открије грешке у систему, већ помаже пројектном тиму да боље разуме функционалности потребне кориснику. TDD процес се заснива на понављању кратког развојног циклуса у коме се кориснички захтеви претварају у тестове, затим се пише минимална количина кода потребна да ти тестови прођу и на крају се кôд рафакторише. У овом раду се говори о узорима који се примењују у TDD-у као начинима за структуирање кôда тако да он буде прилагођенији за тестирање. Такође се наводе узори који се користе за писање јединичних тестова и узори зелене фазе који нам помажу да решавањем тестова дођемо до одговарајуће имплементације захтева. Затим се даје преглед test double објеката који се користе у тестовима као замена за објекте који ће се користити у продукционој верзији.
КЉУЧНЕ РЕЧИ: Test driven development, рефакторисање, јединични тестови, mock објекти
ABSTRACT: Test driven development (TDD) is a software development process created as part of Extreme Programming, but has since been adopted by all of the other Agile methods. A central point in agile methods is working software and TDD is a process that is focused on producing reliable and maintainable software. TDD changes the role that testing has in the software development process and is not longer used only to detect errors in the system, but also helps the project team to better understand the functionality needed by the user. TDD process is based on the repetition of a short development cycle in which user requirements are transformed into tests, then write the minimum amount of code to pass that test, and at the end code is refactored. This paper discusses the patterns that are used in TDD as a way of structuring the code so that it is adapted for testing. Also referred to patterns that are used for writing unit tests and the green phase patterns that help us to, by making tests pass, get appropriate implementation of the requirements. It then provides an overview of the test double objects used in the tests as a substitute for the objects that will be used in production.
KEY WORDS: Test driven development, refactoring, unit tests, mock objects

1. УВОД

Када би Test driven development (TDD) процес морали да опишемо једном реченицом, она би гласила: „Само пишите кôд како би поправили тестове који падају“. Прво пишемо тест, затим кôд као решење за тест. На крају тражимо најбоље могуће пројектно решење за кôд који већ имамо, ослањајући се на постојеће тестове као осигурање да функционалност неће бити нарушена. Овај начин креирања софтвера подстиче квалитетно пројектовање, производи кôд који се лако тестира, и спречава програмере да погреше током креирања система на основу лоше претпоставке [1]. Све ово се постиже једноставним поступком вођења пројекта помоћу тестова у сваком кораку, који нас воде ка завршној имплементацији.

TDD мења улогу који тестирање има у процесу развоја софтвера и оспорава претпоставке о томе чему тестирање служи. Тестирање не служи више само да открије грешке у систему пре него што корисник почне да га употребљава, већ помаже пројектном тиму да боље разуме које су функционалности потребне кориснику и да омогући њихову поуздану испоруку кориснику [3]. Из овога можемо да закључимо да TDD радикално мења начин на који развијамо софтвер и битно побољшава квалитет система који креирамо, као и његову поузданост и флексибилност приликом појаве нових захтева. TDD је у широкој употреби код агилних метода развоја софтвера. Представља део Екстремног програмирања (XP), и често се користи у Scrum методи [4].

TDD процес није први применио тестове за нешто више од саме верификације исправности софтвера, и раније су неки програмери писали тестове пре кода. Али, данас овај начин развоја софтвера има име - TDD. Међутим, TDD и даље представља нову технику развоја софтвера [2]. Као што је нека данашња роба у прошлости претстављала луксуз, тако и технике програмирања и пројектовања често почињу као луксуз неколико искусних практичара, а касније постају прихваћене од стране великог броја пројектаната неколико година касније, када су пионири доказали и обликовали технику. Ако програмери усвоје TDD као професионалну дисциплину, писаће десетине тестова сваког дана, стотине тестова сваке недеље и хиљаде тестова сваке године [6]. Сви ови тестови биће им на дохват руке да их покрену сваки пут када направе неку измену у коду.

2. ПРОЦЕС TDD-А

Процес TDD-а који подразумева писање тестова, писање кода и рефакторисање може допринети расту система кроз додавање тестова за нове функционалности у постојећу инфраструктуру. Али као почети рад на првој функционалности, пре него што креирамо ову инфраструктуру. Тестови

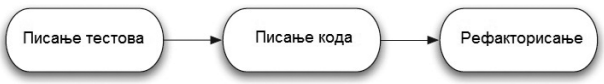
прихватљивости морају тестирати цео систем како би нам дали повратну информацију о спољашњим интерфејсима система, што значи да морамо имплементирати циклус компајлирања, инсталације и тестирања система. А то представља велики део посла и пре него што можемо да почнемо са тестирањем.

Дилема код писања првог теста прихватљивости је то да је тешко креирати тестове и функционалност система који се тестира истовремено. Промена на једној страни проузрокује промену на другој, а праћење грешака је тешко када се архитектура, тестови и продукциони кôд стално мењају. Да би решили овај проблем код креирања прве функционалности делимо га на два мања проблема. Прво, нађемо начин како да креирамо, испоручимо и тестирамо скелет апликације, а онда да искористимо ту структуру да напишемо тестове прихватљивости за прву значајну функционалност [5].

„Скелет апликације“ представља имплементацију најмањег дела функционалности система који се може аутоматски креирати, испоручити и тестирати с краја на крај. Он треба да садржи само толико главних компоненти, кумуникационих механизама и да буде аутоматизован колико је потребно да се почне рад на првој функционалности. Скелет апликације правимо да буде једноставан и лак за разумевање, како би се концентрисали на инфраструктуру [6]. На пример, ако креирамо интернет апликацију која узима податке из базе података, наш скелет ће да садржи интернет страницу са пољима која се налазе у бази података.

Креирање нове функционалности започињемо писањем теста прихватљивости који пада да би демонстрирали да систем не садржи функционалност коју пишемо и да пратимо напредак ка завршетку функционалности. За писање теста прихватљивости користимо само терминилогију из домена апликације, а не из слојева испод (као што су базе података или web сервиси). Ово нам помаже да разумемо сврху система без претпоставки о имплементацији и компликовању тестова са детаљима везаним за технологије које би се користиле. То такође штити наше тестове прихватљивости од промена које би могле настати на техничкој инфраструктури система. Писањем таквих тестова пре писања кода помаже нам да разјаснимо шта желимо да постигнемо. Прецизно исказивање захтева на начин да се могу аутоматски проверавати помаже откривању имплицитних претпоставки.

2.1. *TDD циклус*



Слика 1 – *TDD циклус*

TDD је техника пројековања и програмирања која се у суштини заснива на једноставном правилу: писање продукционог кода само за тестове који падају. TDD мења до сада уобичајни пројектовање-кодирање-тестирање поредак активности у оно што се назива TDD циклус: тестирање-кодирање-рефакторисање. TDD циклус се састоји од три фазе:

Црвена фаза - У којој се захтеви претварају у тестове (који падају и одатле назив црвена). Сви захтеви који се могу предвидети за један подсистем (функционалност), могу се поделити на низ задатака, који када се обаве, почетни захтев је испуњен. У TDD-у захтев се, уместо у задатке, дели на низ тестова. Када сви тестови за захтев прођу он је испуњен. Превођење захтева у тестове има предност у односу на разлагање захтева на задатке јер се са задацима може лако изгубити из вида крајњи циљ, а то је испуњен захтев. Такође, задаци нам само дају представу шта треба урадити, без конкретне дефиниције када је задатак испуњен. При креирању тестова разлагањем корисничких захтева ти тестови морају да имају две битне особине: атомност и изолацију. Ово нам говори да су добри тестови мали, фокусирани и атомски део жељене функционалности и да морају бити изоловани од других тестова како резултат теста не би зависио од претходно извршених тестова.

Зелена фаза - Ако тестови не пролазе то значи да се налазимо у црвеној фази. Да би дошли до зелене фазе потребно је да напишемо само онолико кода колико је довољно да нови тест прође, и притом да не изавове пад неког од постојећих тестова.

Фаза рафакторисања - Без обзира да ли примењују TDD или не програмери увек имају потребу да поправљају њихов кôд, што се у суштини и догађа приликом рефакторисања. Предност рефакторисања у TDD-у је што постоји скуп јединичних тестова који проверавају да ли је ваш кôд и даље у складу са пословним захтевима и после измена. Програмер треба да периодично прегледа програмски кôд и скуп тестова и да тражи могућности за њихово побољшање рефакторисањем. Рефакторисање треба извршавати тако што ће се правити мале промене, а потом користити јединичне тестове за проверу да ли рефакторисани кôд и даље функционише. Док год тестови пролазе кôд је исправан.

Веома је битно праћења реда извршавања ових фаза. Када се прати овај низ корака (прикупљање захтева, писање тестова, писање кода) обезбеђује се постојање тестова за кôд који се пише, и то да се пише само кôд за тестове који постоје. Када се заврши један TDD циклус почиње следећи јер се појављују нови захтеви и функционалности, који се преводе у тестове. Потпун скуп тестова представља очекивану функционалност апликације.

3. РЕФАКТОРИСАЊЕ КОДА

Ниједан програмски кôд није савршен. Увек постоји нешто што се може боље урадити, мада то у тренутку креирања и не примећујемо. Када гледамо кôд који смо написали раније, можемо наћи мале делове кода које желимо да изменимо да би га учинили бржим, боље усклађеним са потребама посла или лакшим за одржавање.

Рефакторисање представља мењање интерне имплементације класа или метода са циљем да се кôд учини читљивијим и лакшим за одржавање. Рефакторисање такође смањује укупну комплексност кода а при то не мења спољње понашање класа или метода**.** Ове промене

могу бити једноставне, као што је промена имена методи или променљивој, па до комплексних ствари као што је измештање методе из једне класе у другу или дељење једне класе на више класа. Рефакторисање омогућава непрекидну измену и унапређење кода [4].

3.1. *SOLID принципи*

SOLID представља скраћеницу за пет принципа објектно орјентисаног програмирања и пројектовања, које је увео Роберт Мартин 2000. Године [2]. Ови принципи се примењују заједно у намери да се направи систем који се лако одржава и надограђује. SOLID принципи представљају једну врсту упуства за програмере у пројектовању нових апликација и рефакторисању постојећег програмског кода.

Принцип једног задужења (енг. Single Responsibility Principle) говори да свака класа треба да има једно задужење, и то задужење мора бити енкапсулирано у класи. Сви сервиси класе морају бити уско повезани са тим задужењем. Роберт Мартин дефинише задужење као „разлог за промену“, и закључује да класа или метод треба да има један и само један разлог за промену [2].

Отворен/Затворен принцип (енг. Open/Close Principle) је концепт који говори да софтвер (методе, класе итд.) мора бити отворен за надоградњу, али затворен за модификацију [2]. Овај принцип је сличан принципу наслеђивања у објектно орјентисаном програмирању. Сврха овог принципа је да програмер користи основне класе или креира основне класе које ће неки други програмер да користи. Корисници основних класа не могу да их мењају. Ово је битно јер се други корисници ослањају на функционалности коју обезбеђује основна класа. Ако би се дозволило кориснику да промени основну класу то би изазвало ланчану реакцију не само на нивоу једне апликације.

Оно што је наслеђивање за Отворен/Затворен принцип то је полиморфизам за **Лисков принцип замене** (енг. Liskov Substitution Principle). Лисков принцип замене говори да се објекат у апликацији може заменити објектом надкласе а да притом не наруши конзистентност апликације [2].

Интерфејс се може описати као уговор који се дефинише у коду и који класа мора поштовати ако жели да га имплементира. То значи да класа мора да имплементира све методе које су дефинисане у интерфејсу. Начин на који ће имплементирати методе зависи од класе, али мора поштовати дефиницију из интерфејса. **Принцип издвајања интерфејса** (енг. Interface Segregation Principle) говори да клијент не сме бити приморан да се ослони на интерфејс уколико га не користи [2].

Dependency Inversion принцип говори да кôд треба да буде зависан од абстракција а не од конкретних имплементација [2]. Класе се ослањају на друге класе како би оне за њих извршиле неки посао. Овај концепт чини да систем буде флексибилнији. Ако је класама битно да компоненте подржавају одређени интерфејс а не конкретни тип, онда се може лако променити функционалност ових сервиса ниског нивоа са минималним утицајем на остатак система. Ова зависност између компонената се може моковати (енг. mock) ради лакшег тестирања. Али у једном тренутку мора се обезбедити конкретна имплементација ових сервиса ниског нивоа како би класа могла да извршава свој посао. Ово се постиже помоћу Dependency Injection патерна.

3.2. *Узори у рефакторисању*

Током развоја софтвера програмери наилазе на групу проблема који се често појављују у коду. Ови проблеми током времена формирају групу опште познатих и широко распрострањених решења. Ова решења се називају узори и у даљем тексту наводимо неке који се најчешће користе у TDD-у.

Издвајање метода је једно он најчешћих рефакторисања која се примењује. Метод који је сувише дугачак или кôд коме треба коментар да би му намена постала јасна се рефакторише тако што се део кода претвара у засебан метод [4].

Издвајање класе - Класа би требало да буде сажета апстракција, која управља са неколико јасних одговорности. У пракси класа расте, јер јој се додају нове операције и подаци. Такве класе имају велики број метода и података, и сувише је велика да би се лако разумела. У оваквим случајевима треба применити издвајање класе, тако што се направи нова класа која од постојеће преузима одређена поља и операције [4].

Енкапсулирање поља - Један од основних начела објектно орјентисаног програмирања је енкапсулација, или скривање. Оно каже да подаци никада не требају бити јавно доступни. Када су подаци јавни други објекти могу да приступају подацима и да их мењају, без знања објекта који их поседује. Овим се одвајају подаци од понашања [4].

Замена услова полиморфизмом - Сврха полиморфизма је у томе што нам омогућава да избегнемо писање експлицитног услова када имамо објекте чије се понашање мења зависно од њиховог типа. Као последица тога у објектно орјентисаном програмирању има мање switch наредби које се мењају према типу и if-else израза које за услов користе назив типа. Највећа предност у употреби полиморфизма је ако се исти скуп услова јавља на пуно места у програму. Тако да када желите да додате нови тип, морате да пронађете сва места где се услови налазе и да их измените. Али ако користите поткласе, једноставно направите нову потклasу и обезбедите одговарајуће методе [4].

Промена имена метода, променљивих и класа - Приликом рефакторисања кода од сложеног процеса добијамо већи број малих метода. Да би разумели шта те методе раде, важно је њихово правилно именовање. Методи морају бити тако именовани да њихово име одражава њихову намену. Имена метода, променљивих и класа би требало да буду јасна, промишљена, написана једноставним и свакодневним језиком. Не би требало користити скраћенице које нису

уобичајне у компанији или у развоју софтвера уопште. Дуга дескриптивна имена су пожељнија од кратких на основу којих није лако открити намену метода, поља или класе [2].

4. УЗОРИ У TDD-У

У овом делу рада се бавимо узорима који нам помажу да лакше креирамо софтвер користећи TDD [6]. На почетку наводимо узоре чијом употребом, приликом пројектовања софтвера, добијамо кôд који се лакше тестира. Затим наводимо узоре које користимо приликом креирања јединичних тестова. После њих говоримо о узорима које користимо током зелене фазе TDD циклуса. На крају описујемо узоре за креирање test double објеката.

4.1. *Узори за креирање кода који се лакше тестира*

Када креирамо софтвер вођен тестовима, ми доносимо одлуке везане за то како ће софтвер бити пројектован. Писање тестова пре кода помаже томе да он буде лакши за тестирање, али би било корисно када би знали који то узори у пројектовању чине да наш кôд буде лакши за тестирање а који не.

Бирати агрегацију пре наслеђивања - Један начин за структуирање кôда тако да он буде прилагођенији за тестирање, представља креирање комплексних објеката од мањих изолованих функционалности. У објектно орјентисаним језицима то се може постићи наслеђивањем или агрегацијом. Рад са хијерархијом наслеђивања може бити нарочито тежак када је потребно инстанцирати објекат у тесту. На пример, ако морамо да обезбедимо валидан параметар за конструктор наткласе иако нас само интересује понашање поткласе. Агрегација представља начин креирања објеката који пружа сложене функционалности комбиновањем групе мање сложених објеката. Тако добијени објекат делегира посао својим компонентама уместо да позива методе своје наткласе. У суштити, агрегација се базира на подели одговорности на нивоу објеката уместо статичке поделе на нивоу класе. Агрегацијом се постиже већа флексибилност пројектовања у погледу поновне употребе функционалности других класа [3].

Избегавати static методе и Singleton узор - У зависности како се користе static методе или Singleton узор у коду који тестирамо, може бити веома компликовано заменити њихову функционалност са test double објектима који имитирају ту функционалност. Позиви static метода су тешки за имитацију јер је позив циљаној класи уграђен у кôд. Слично, тешко је извршити замену Singleton класе имитацијом имплементације јер се она обично позива преко static getInstance() методе. Уобичајени начин на који програмери користе Singleton класе односи се на карактеристику узора да обезбеђује једну тачку приступа. Мада је добро имати једну тачку приступа, то може постати проблематично ако се приступ тој тачки налази на много места у коду [3].

Изоловање зависности - Да би се олакшала замена зависности са њиховим имитацијама за тестирање (енг. test doubles), битно је прво изоловати зависности на начин који нам омогућава да најлакше извршимо замену. Места на којима једну функционалност можемо заменити другом се називају спојеви. Спојеви су места на којима можемо заменити део кода са другим кодом током периода тестирања, а да при томе не мењамо кôд који се тестира. Спој је по дефиницији праћен са једном или више тачака за искоришћење споја, који представљају начине за његову употребу. Изоловање зависности представља битан узор за креирање кода који се лакше тестира и који је лакши за одржавање. А наручито је користан у случајевима када изолација омогућава лакше тестирање тако што се зависност премести, па уместо да продукциони кôд набавља објекте од којих је зависан, објекте инјектује неки екстерни кôд [3].

Инјектовање зависности - Инјектовање зависности је приступ за структуирање кода на начин да замењује директне зависности са индиректним зависностима, другим речима замењује методу за узимање вредности са методом за постављање вредности. Када се користи инјектовање зависности тестни кôд постаје јаснији него раније, јер се у тестовима могу лако конфигурисати објекти потребни за поставку теста. У пракси за инјектовање зависности се користи inversion of control (IoC) контејнер који обавља посао за нас. Овај контејнер се бави везивањем зависности за имплементацију, и углавном користи централизовани метод за конфигурацију (обично се користи XML фајл) [3].

4.2. *Узори јединичних тестова*

Јединични тестови су интегрални део TDD-а, и њих делимо у две групе: узоре за писање верификације и узоре за структуирање и креирање поставке теста.

4.2.1 *Узори верификације*

Верификација представља суштину теста, јер је тест безначајан уколико ништа не верификује. **Верификација стања резултата** (енг. Resulting State Assertion) је један од најчешћих начина за извршавање верификације у јединичним тестовима. Идеја је да се на објекту који се тестира изврши одређена функција и да се после изврши верификација да ли интерно стање објекта одговара нашим очекивањима. Верификацију стања користимо када нас занима завршно стање објекта који се тестира, а не како смо дошли до објекта. Тај ограничен поглед на систем који се тестира нам омогућује да енкапсулирамо његову имплементацију [6].

Верификација може некада бити безуспешна ако објекат који се тестира није иницијализован као што се очекивало. Када се у тесту појави неочекивани проблем, то може изазвати грешку у тесту и тест ће пасти. **Узор заштите верификације** (енг. Guard Assertion) служи за проверу претпоставки о поставци теста пре позивања функционалности коју желимо да тестирамо [6].

Понекад је потребно да у тестовима радимо с кодом над којим немамо потпуну контрулу. Ово наручито представља проблем ако тест има припрему над којом нема контрулу. Решење је да се не верификује апсолутно стање када се позове тестирана функционалност, већ да се тестира разлика, делта, између почетног и крајњег стања. Овај узор се назива **делта верификација** [6].

Понекад количина кода потребна за верификацују далеко превазилази количину кода потребну за поставку теста и позивање функционалности објекта који се тестира. Када се ово догоди потребно је издвојити методу у којој ће се налазити верификација, тако да се добија мала метода која енкапсулира комплексну верификациону логику која се може позивати из тест методе. Ова верификација се назива **прилагођена верификација** (енг. Custom Assertion) [6].

4.2.2. *Узори поставке теста*

Поставка представља битан део теста, и често има сложену конструкцију. Поставке тестова могу бити обимне у којима је потребно креирати велики број објеката. Комплексан и обиман кôд представља проблем било да је продукциони или тест кôд. Тако да се током година појавило доста узора који нам помажу да правилно структуирамо поставку теста.

Већина објеката који се налазе у припреми теста су ентитетски објекти, и они представљају неку појаву у пословном домену. Ови типови објеката имају велики број атрибута и већина кода у припреми теста се бави додељивањем вредности овим атрибутима, иако велика већина од њих није од значаја за тест. **Параметризовани метод креирања објекта** (енг. Parameterized Creation Method) скрива небитне атрибуте од методе за припрему теста издвајањем креирања објеката у засебну методу, која узима промењљиве атрибуте као аргументе и додељује константне или насумичне вредности небитним атрибутима. Такође, параметризовани метод креирања објекта метода може да води рачуна о додељивању вредности небитним атрибутима који имају јединствене захтеве [6].

У почетку, можемо доста постићи рефакторисањем тест класа тако да користе методе за креирање објеката уместо да се кôд понавља. Тако да ће доћи до тога да понављамо методе за креирање објеката у више тест класа. Следећи корак је премештање поновљеног кода у методе за креирање објеката у издвојену помоћну класу. **Узор мајка објекат** (енг. Object Mother) описује баш такав скуп метода за креирање објеката. Мајка објекат узор може бити реализован као софистицирана фабрика објеката која обезбеђује приступ комплетном скупу објетака који престављају валидне доменске објекте, укључујући инстанце са различитим стањем [6]. Мајка објекат може да обезбеди и методе за измену задатих доменских објеката, на пример, повезивање објеката између себе, уклањање везе, или премештање објекта у жељено стање. Поред тога што смањује понављање у тестном коду, Мајка објекат узор, олакшава приступ потребним објектима програмерима који су тек почињу да пишу јединичне тестове, и то их охрабрује да пишу још више тестова.

Сврха постојања метода за враћање на почетно стање у платформама за јединично тестирање је да изврши сва потребна брисања после извршавања тест методе. На пример, могу се брисати објекти из базе података или генерисани фајлови из фајл система. Тамо где је логика за враћање на почетно стање комплексна или постоји велики број објеката које треба обрисати, тестови постају претрпани и може се дестити да се не избрише неки од објеката. То може изазвати проблеме које је тешко дебаговати и наћи извор проблема. **Узор за аутоматско враћање на почетно стање** решава овај проблем тако што енкапсулира логику за враћање на почетно стање у засебну класу. У узору за аутоматско враћање на почетно стање поставка теста не само да креира потребне објекте за тест већ их и региструје у регистар тестних објеката. Регистар представља колекцију референци ка регистрованим објектима и одговоран је за брисање објеката [6].

4.3. *Узори зелене фазе*

Као што постоји моного начина да се тест напише, постоји и много начина да се нађе решење за њега. Кент Бек у својој књизи Test-Driven Development by Example наводи три начина за превазилажење зелене фазе TDD циклуса:

Имитација решења - Понекад после кретирање теста који пада, немамо идеју како да напишемо исправан кôд који ће бити решење тог теста. У таквим случајевима, можемо да прибегнемо имитацији одређене функционалности како би добили тест који пролази [3]. Имитација решења може бити враћање константе како би се задовољили тренутни захтеви из теста. Наш примарни циљ је да дођемо у конзистентно стање тако што ће сви тестови пролазити, а имитација је боље решење него оставити тест да пада дужи период времена. Предност имитације решења што мало времена проводимо у решавању теста, па ако у даљем раду откријемо да тест није добро написан биће нам драго што у његово решавање нисмо инвестирали много времена.

Триангулација - Ако не знате како да решите комплексни алгоритам, требало би да напишете више тестних случајева. Често је лакше одредити понашање алгоритма испитујући неколико тестних случајева него само једног. Триангулација је техника која омогућава напредак према исправној имплементацији када још немамо јасан концепт како да имплементирамо одређену финкционалност [3].

Очигледна имплементација - У већини случајева неопходан корак који морамо да направимо како би тест прошао је очигледан. У том случају можемо наставити развој са оним што сматрамо за очигледну имплементацију, правећи већи корак него уобичајени када користимо триангулацију или имитацију функционалности долазимо до крајњег циља [3]. Пример за очигледну имплементацију могу бити узори који се стално појављују.

4.4. *Test double објекти*

Најчешћи проблеми који се јављају приликом тестирања класа или објеката у вези су са другим објектима с којима

они сарађују или од којих зависе. На пример, када је потребно проследити објекту који се тестира, инстанцу објекта од кога он зависи чије је инстанцирање тешко изводљиво у тесту. Као решење овог проблема користимо test double објекте. Test double објекти су објекти који замењују праву инстанцу објекта у тесту тако да објекат који их користи не примећује разлику. Они се обично извршавају брже од правих инстанци објеката а време потребно за њихово креирање је краће. Постоји неколико типова test double објеката:

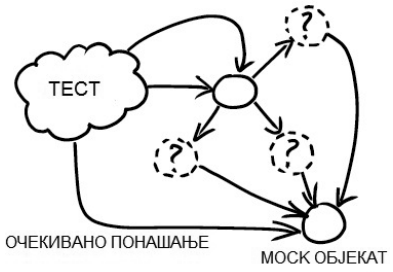
Dummy објекат представља најједноставнији тип test double објеката. Он не садржи никакву имплементацију и у већини случајева се користи да би био прослеђен као аргумент, и тако се елиминише потреба за креирањем правог објекта [5].

Test stub креирамо тако што прво имплементирамо тест верзију интерфејса од кога објекат који се тестира зависи. Ова имплементација је конфигурисана тако да одговара на позиве од стране објекта који се тестира са унапред дефинисаним вредностима или изузецима. Test Stub можемо користити као контролну тачку која нам омогућава да контролишемо понашање објекта који се тестира приликом прослеђивања различитих улазних вредности [5].

Spy је објекат који бележи информације о интеракцији коју има са објектом који се тестира, тако да би те информације биле доступне за верификацију самог теста. Примена Spy објекта је једноставан начин за имплементацију верификације понашања [5].

Fake објекат представља много једноставнију имплементацију функционалности коју обезбеђује компонента од које је зависан објекат који се тестира. Fake објекат се користи када објекат који се тестира зависи од друге компоненте која је недоступна или процес тестирања чини компликованим или спорим, а тесту је потребно сложеније понашање него коришћењем Test Stub или Mock објекта. Fake објекат можемо користити да замењује базу података или web сервис [5].

Mock објекти се могу користити да симулирају комплексно понашање које користи објекат који се тестира. Mock објекти су комплексни и као такве не креирамо их ми сами већ користимо библиотеке mock објеката. Тестови креирани коришћењем mock објеката се разликују од традиционалних тестова, јер очекивано понашање mock објекта мора да буде одређено пре позива објекта који се тестира. Ово чини тестове тежим за креирање и разумевање. Када се позове током извршавања објекта који се тестира, mock објекат упоређује улазне аргументе са очекиваним аргументима и ако се не поклапају тест пада. Да би користили mock објекте морамо бити у могућности да предвидимо већину или све аргументе позива метода пре него што позовемо објекат који се тестира [5].



Слика 2 – *Употреба mock објеката*

5. СТУДИЈСКИ ПРИМЕР: УПОТРЕБА TDD-А

На следећем примеру користимо TDD за развој методе која, приликом креирања новог корисника, проверава да ли су вредности његових атрибута валидне. Кориснички захтеви које треба испунити приликом креирања методе су:

- Корисничко име мора бити веће од 6, а мање од 12 карактера.
- Шифра мора бити већа од 8, а мања од 14 карактера.
- Шифра мора садржати макар једно велико слово и макар један број.
- Два корисника са истим корисничким именом не могу да постоје.

Рад почињемо тако што креирамо тест класу у којој ће се налазити сви тестови за методу која се креира. А затим креирамо прву тест методу за један од корисничких захтева:

```
[TestFixture]
class KorisnikServiceTest
{
    [Test]
    public void korisnicko_ime_manje_od_6_karaktera_vraca_false()
    {
        KorisnikService korisnikService = new KorisnikService();
        string korisickoIme = "abcd";
        string sifra = "Password2";
        bool rezultat = korisnikService.Provera (korisickoIme, sifra);   //
poziv metode koja se testira

        Assert.IsFalse(rezultat);  // verifikacija rezultata
    }
}
```

У овом тренутку тест неће проћи, нити ће се компајлирати јер класа KorisnikService и метода Provera не постоје. Да би се кôд компајлирао потребно је креирати потребну класу, а методу имплементирати на следећи начин како би прво имали тест који пада.

```
public class KorisnikService
{
    public bool Provera(string korisnickoIme, string sifra)
    {
        throw new NotImplementedException();
    }
}
```

Када имамо тест који пада прелазимо у зелену фазу, у којој имплементирамо методу на најједноставнији начин како би тест прошао.

```
public bool Provera(string korisnickoIme, string sifra)
    {
      return false;
    }
```

Када имамо тест који пролази прелазимо на фазу рефакторисања. После првог теста рефакторисање није потребно тако да настављамо развој писањем теста за други кориснички захтев:

```
    [Test]
    public void korisnicko_ime_vece_od_6_karaktera_vraca_true()
    {
      KorisnikService korisnikService = new KorisnikService();
      string korisnickoIme = "abcdefg";
      string sifra = "Password2";
      bool rezultat = korisnikService.Provera(korisickoIme, sifra);

      Assert.IsTrue(rezultat);
    }
```

Опет примењујемо најједноставније решење тако да имплементација методе изгледа овако:

```
public bool Provera(string korisnickoIme, string sifra)
    {
      if (korisnickoIme.Length < 6)
        return false;
      else
        return true;
    }
```

Следећи корак је рефакторисање у коме можемо да издвојимо инстанцирање класе KorisnikService у SetUp методу која се извршава пре извршавања сваког теста.

```
  [TestFixture]
  class KorisnikServiceTest
  {
    KorisnikService korisnikService;

    [SetUp]
    public void SetUp()   // metoda koja se izvrsava pre svakog testa
    {
      korisnikService = new KorisnikService();
    }
}
```

После рефакторисања вршимо регресивно тестирање како би смо проверили да ли кôд има исту функционалност после измена.

На исти начин креирамо тестове за корисничке захтеве у којем се захтева да корисничко име буде мање од 12 карактера, шифра мања од 8 а већа од 14 карактера, да шифра садржи макар један број и велико слово. И затим вршимо рефакторисање тако што у посебне методе издвајамо кôд који проверава валидност корисничког имена и шифре. Тако да имплементација изгледа овако:

```
    private bool ProveraKorisnickogImena(string korisnickoIme)
    {
      if (korisnickoIme.Length < 6 || korisnickoIme.Length > 12)
        return false;
      else
        return true;
    }

    private bool ProveraSifre(string sifra)
    {
      if (sifra.Length < 8 || sifra.Length > 14)
        return false;
      else if (Regex.IsMatch(sifra, @"\d") == false)
        return false;
      else if (sifra.Any(c => char.IsUpper(c)) == false)
        return false;
      else
        return true;
    }

    public bool Provera(string korisnickoIme, string sifra)
    {
      if (!ProveraKorisnickogImena(korisnickoIme))
        return false;
      else if (!ProveraSifre(sifra))
        return false;
      else
        return true;
    }
```

Следећи кориснички захтев проверава да ли корисник са тим корисничким именом већ постоји у складишту података. И за то се користи метода VratiKorisnikaZaID класе KorisnikDL, чије ћемо понашање представити mock објектом. Тако да кôд за следећи тест изгледа овако:

```
 [Test]
 [TestCase("abcd2012", "Password2", false)]
 [TestCase("abcd2013", "Password2", true)]
 public void ako_korisnicko_ime_vec_postoji
(string korisickoIme, string sifra,
bool ocekivano)
 {
   // konfigurisanje mock objekata
    var korisnikDl = new Mock<KorisnikDL>();
    korisnikDl.Setup(p => p.VratiKorisnikaZaID("abcd2012"))
          .Returns(new Korisnik("abcd2012", "Password2"));
    korisnikDl.Setup(p => p.VratiKorisnikaZaID("abcd2013"))
          .Returns((Korisnik)null);
    korisnikService.korisnikDLProp = korisnikDl.Object;

    bool rezultat = korisnikService.Provera(korisickoIme, sifra);
    Assert.AreEqual(ocekivano, rezultat);
 }
```

Решење претходног теста представља следећи кôд:

```
public bool Provera(string korisnickoIme, string sifra)
    {
      if (!ProveraKorisnickogImena(korisnickoIme))
        return false;
      else if (!ProveraSifre(sifra))
        return false;
      else if (korisnikDL.VratiKorisnikaZaID(korisnickoIme) != null)
        return false;
      else
        return true;
    }
```

Међутим, покретањем осталих тестова видећемо да они падају, јер у њима нисмо користили mock објекат за конфигурацију методе VratiKorisnikaZaID. Тако да је потребно креирање, конфигурисање и инсталацију mock објекта преместити у SetUp методу:

```
[TestFixture]
   class KorisnikServiceTest
   {
      KorisnikService korisnikService;

      [SetUp]
      public void SetUp()
      {
         korisnikService = new KorisnikService();
         var korisnikDl = new Mock<KorisnikDL>();
         korisnikDl.Setup(p => p.VratiKorisnikaZaID("abcd2012"))
            .Returns(new Korisnik("abcd2012", "Password2"));
         korisnikDl.Setup(p => p.VratiKorisnikaZaID("abcd2013"))
            .Returns((Korisnik)null);
         korisnikService.korisnikDLProp = korisnikDl.Object;

      }
}
```

Када завршимо премештање у SetUp методу покрећемо све тестове, и проверавамо да ли после измене сви тестови пролазе. Ако се сви тестови успешно изврше завршили смо имплементацију функционалности.

6. ПРЕДНОСТИ И НЕДОСТАЦИ TDD-А

Када се TDD представља пројектантима, програмерима и пројект менаџерима који га пре нису користили, дочекају га са неповерењем. На први поглед креирање кода делује као дуг и компликован процес. Али TDD доноси и следеће користи [2, 3 ,4 ,7]:

- TDD осигурава квалитетан кôд од почетка. Програмери су подстакнути да пишу само онолико кода потребног да тест прође и тако испуни захтев. Ако метод има мање линија кода, логично је да је мања вероватноћа да кôд има грешку.
- TDD осигурава висок степен сличности између кода и пословних захтева. Ако су захтеви написани као тестови, а тестови пролазе, може се високим степеном сигурности рећи да кôд испуњава потребе посла.
- TDD подстиче креирање једноставнијих и уско орјентисаних библиотека. TDD мења начин развоја софтвера зато што програмер креира интерфејс за библиотеку а такође је њен први корисник. Ово програмеру даје нову перспективу у погледу како интерфејс треба да буде написан, и одмах зна да ли интерфејс има смисла.
- TDD подстиче комуникацију са пословним корисницима. За креирање тестова, потребна је колаборација програмера и пословног корисника. На овај начин програмер може да разуме све комбинације улазних и излазних параметара, и да помогне кориснику да разуме какав систем креира.
- TDD избацује кôд који се не користи из система. Већина програмера креира апликације у којима дизајнира интерфејсе и креира методе на основу онога шта може да се догоди у будућности. Ово доводи до система који садржи велики део кода који се не користи. Овај кôд је скуп, јер се троше ресурси за његово писање, а иако кôд не ради ништа он се мора одржавати. Такође, кôд изгледа претрпан, тако да одвлачи пажњу са кода који ради.
- TDD обезбеђује уграђено регресивно тестирање. Како се мења систем и ваш кôд, увек ћете имати пакет тестова које сте креирали да сутрашње промене не наруше данашњу функционалност.
- TDD излази на крај са грешкама које се понављају. Често се дешава да програмер поправи грешку која му је пријављена од стране тестера, али се после одређеног времена та грешка опет појави. У TDD-у чим се пријави грешка, креира се нови тест који открива ту грешку. Када тест прође, и настави да пролази програмер је сигуран да је грешка отклоњена.

Поред ових предности TDD има и следеће недостатке [1, 3 ,4 ,7]:

- Јединичне тестове у TDD-у креирају програмери који креирају и продукциони кôд који они тестирају. Па ако програмер погрешно интерпретира захтеве може се десити да и тестови и продукциони кôд буду неисправни.
- TDD је тешко применити у ситуацијама где је потребно тестирати пуну функционалност система, као што је употреба корисничког интерфејса, када програм ради са подацима из база података, или је зависан од мрежне конфигурације. Зато TDD подстиче програмере да такве системе креирају са што мање кода а да спољни свет представе помоћу fake или mock објеката.
- Ниво покривености и сами тестови који су креирани током TDD циклуса не могу се лако исправити или поново креирати у каснијим фазама развоја. Тако да ови оргинални тестови постају све битнији како време пролази. Ако лоша архитектура или лоше пројектовање доведу до тога да измене које се касније изврше изазову падање великог броја постојећих тестова, битно је да се они индивидуално поправљају. Једноставно брисање или брзоплето мењање могу довести до неприметних празнина у покривености кода тестовима.

7. ЗАКЉУЧАК

Употребом TDD методологије за развој софтвера значајно се смањује број дефеката у коду, побољшава продуктивност програмера и добија софтвер бољег квалитета. Такође, када користе TDD програмери имају јасан увид у то како напредују у извршавању њихових задатака и боље разумеју постављене захтеве [7]. У TDD-у свакодневним покретањем интеграционих тестова постижемо то да се

потенцијални проблеми појаве много раније него када се кôд интегрише у продукционој фази. Међутим, многим програмерима представља проблем прелазак на начин размишљања који се користи у TDD-у и многи се суочавају са потешкоћама у усвајању TDD методологије [7].

Даљи правци развоја су истраживање како се TDD примењује у различитим типовима апликација. На пример, како се TDD примењује у web апликацијама, пре свега у ASP.NET MVC (Model View Controller) технологији. Затим како се помоћу TDD-а тестирају: апликације за приступ складиштима података, апликације које користе нити и апликација асинхроног кода.

У даље правце развоја може се навести и следеће:

- Анализирање Behaviour-Driven Development-а, као технике настале из TDD-а, и у чему се оне разликују.
- Развој аутоматизованих алата за рефакторисање.
- Обука пројектног тима да користи TDD и његове алате.

ЛИТЕРАТУРА

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

[2] James Bender, Jeff McWherter: Professional Test-Driven Development with C#, Developing Real World Applications with TDD; Wiley Publishing, 2011.

[3] Lasse Koskela: Test Driven: Practical TDD and acceptance TDD for Java developers; Manning Publications, 2008.

[4] Martin Fowler: Рефакторисање, побољшање дизајна постојећег кода; CET, 2003.

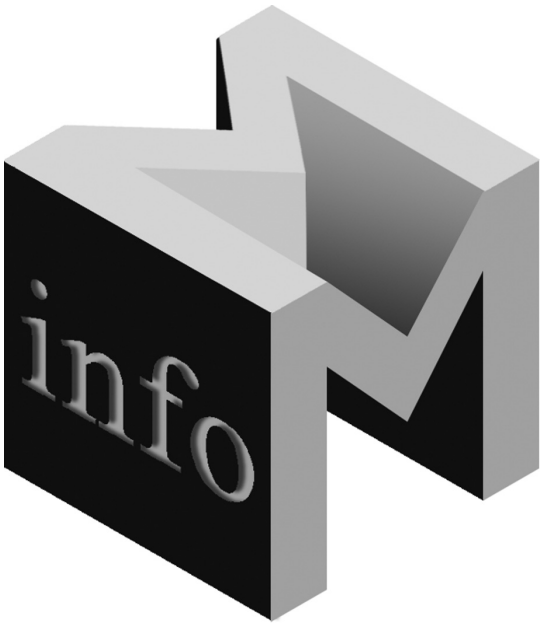
[5] http://msdn.microsoft.com/en-us/magazine/cc163358.aspx

[6] Gerard Meszaros: xUnit Test Patterns, Refactoring Test Code; Addison Wesley, 2007.

[7] B. George, L. Williams; An Initial Investigation of Test- Driven Development in Industry, http://collaboration.csc.ncsu.edu/laurie/Papers/TDDpaperv8.pdf

Слободан Мирковић, студент мастер студија, Факултет организационих наука Универзитета у Београду.
Контакт: mirkovic.sloba@gmail.com
Области интересовања: софтверско инжењерство, тестирање софтвера, базе података

Др Саша Д. Лазаревић, Катедра за софтверско инжењерство, Факултет организационих наука, Универзитет у Београду.
Контакт: sasa.lazarevic@fon.bg.ac.rs
Области интересовања: конструкција софтвера, тестирање софтвера, квалитет софтвера, развој информационих система, структуре података и алгоритми, базе података, системи за управљање документацијом, .NET платформа



CIP – Каталогизација у публикацији Народна библиотека Србије, Београд 659.25

INFO M : časopis za informacionu tehnologiju i multimedijalne sisteme = journal of information technology and multimedia systems / glavni i odgovorni urednik Dejan Simić.

– Štampano izd. – God. 1, br. 1 (2002) – . – Beograd : Fakultet organizacionih nauka, 2002 – (Stara Pazova : SAVPO). – 30 cm

Tromesečno. – Je nastavak: Info Science = ISSN 1450-6254. – Drugo izdanje na drugom medijumu: Info M (CD-ROM izd.) = ISSN 1451-4435

ISSN 1451-4397 = Info M (Štampano izd.) COBISS.SR-ID 105690636