др Синиша Влајић, Бојан Томић

ПРАКТИЧНО ТЕСТИРАЊЕ ПРОГРАМСКИХ КЛАСА ЗАСНОВАНО НА ПРОВЕРИ ОГРАНИЧЕЊА

Кратак преглед рада

У овом раду је представљен један практичан поступак за функционално тестирање програмских класа који је првенствено намењен ученицима, студентима, истраживачима и свима који се не баве тестирањем професионално. На почетку рада се налази увод који описује важност тестирања програма уопште. Рад се наставља дефиницијом тестирања програма и класификацијом тестова. Предмет и циљеви рада су описани у наредном поглављу. Централни део рада садржи детаљан опис предложеног поступка, његове структуре, редоследа корака и показатеља успешности, као и кратку напомену о практичној примени. На крају рада се налази закључак у коме је описано шта је постигнуто и који су могући даљи правци истраживања.

Увод

Развој рачунара и рачунарских технологија је довео до великих промена у свим сферама живота. Управо највећи утицај овог развоја се може видети у области науке. Научна истраживања, велика или мала, зависе од рачунара у многим својим фазама: прикупљање података, моделирање, симулација, обрада података, потврда или оповргавање хипотеза, формирање извештаја итд. Због тога, честа је ситуација да управо научници и истраживачи развијају или мењају програме који су им потребни за обављање истраживања.

Међутим, ту постоји један велики проблем. Област тестирања и контроле квалитета програма уопште није на одговарајући начин регулисана. Чак је и у домену пословних апликација иста ситуација. Стандарди постоје (нпр. ISO 9126 [4]) али нису прихваћени, а исто је и са поступцима и регулативама који се тичу квалитета програма. Последица је та да се програми контролишу и тестирају нередовно, несистематично и без икаквог плана па често имају слабе перформансе, слабу поузданост, не пружају тражену функционалност, тешки су за одржавање и надоградњу или су компликовани за употребу.

Али, не мора да буде тако. Тестирање програма може да помогне да се неки од ових проблема превазиђу. Додатно време које је утрошено на тестирање ће свакако бити враћено због смањеног времена потребног да се отклоне грешке или повећају перформансе.

Ако се прихвати чињеница да је тестирање програма неопходно, отвара се питање како би то требало радити. Да би резултати тестирања били добри, тестирање мора да се врши организовано и систематично. Другим речима, потребан је један добар поступак за тестирање којим би се одредиле фазе тестирања и њихов редослед, предмет и структура тестова, њихов број, редослед писања и критеријуми који се односе на покривеност тестовима.

Овакви поступци већ постоје, али су често превише компликовани за коришћење, неинтуитивни или једноставно неефикасни. Иако је њихова компликованост можда оправдана када се ради о пословним апликацијама, ови поступци су тешко применљиви у осталим случајевима. Да би био прихваћен, поступак за тестирање би морао бити систематичан, али пре свега практичан. У овом раду се описује један такав поступак.

Тестирање програма

Тестирање програма је део контроле квалитета програма - процес који има за циљ осигурање потребног нивоа квалитета програма. Неке од дефиниција тестирања програма су:

„Тестирање програма (Software testing) се састоји у динамичкој провери понашања програма извођењем коначног броја тестова (који су на одговарајући начин изабрани од бесконачно много могућих тестова) и упоређивањем са очекиваним понашањем програма.“ [1]

„Тестирање програма (Software testing) је процес чијим извршавањем се настоји утврдити коректност, потпуност, безбедност и квалитет развијеног програма. То је техничка истрага коју врше заинтересована лица са циљем утврђивања информација које приказују однос нивоа постигнутог квалитета производа у односу на жељени квалитет.“ [2]

Квалитет програма се састоји из две димензије: ефективност и ефикасност. *Ефективност* се односи на то да ли су кориснички захтеви на одговарајући начин формулисани (да ли захтеви описују оно што је кориснику потребно). Процес провере ефективности програма зове се *валидација*. *Ефикасност*, са друге стране, подразумева да се захтевима описана функционалност пружа на квалитетан начин (програм ради брзо, поуздано, нема грешке, лак је за одржавање и надоградњу и сл.). Процес провере ефикасности програма зове се *верификација*.

Тестирање програма је верификаторно средство. *Предмет* тестирања је ефикасност програма (а не ефективност). Тестирање не може да обезбеди да програм ради оно што би требало јер се тестови пишу у складу са корисничким захтевом а, ако је захтев лоше написан, и програм ће бити лош у складу са тим. Ни сви аспекти ефикасности се не могу обезбедити тестирањем. Пример за то је једноставност одржавања и надоградње. Једини начин да се то обезбеди је коришћење *узора* (*software pattern*) при пројектовању програма ([6], [7]). Оно што тестирање може да обезбеди је да програм квалитетно (брзо и без грешке) ради оно што је описано у захтевима.

Два појма се често појављују у контексту тестирања програма: програмска мана и програмска грешка. *Програмска мана* (*software fault*) је неки недостатак

који програм поседује. *Програмска грешка* (*software failure*) настаје када се програмска мана манифестује. Другим речима, сваки програм има мане, али се само неке од њих испољавају у виду грешака, а многе не, па остану неоткривене. Да би се мана испољила, потребно је да се део програма који садржиману изврши. Мане које остану сакривене, могу да почну да се манифестују као грешке ако се промени софтверска или хардверска платформа на којој се извршава програм.

Тестирање програма има за *задатак* да открије што више програмских мана на тај начин што ће, извршавањем и провером извршавања што више делова програма, довести до њиховог манифестовања у виду грешака. Циљ је да се, откривањем и отклањањем мана, створи одређени ниво сигурности да програм ради оно што је описано захтевима.

Сами тестови се могу класификовати на више начина, али је најпопуларнија класификација према врсти захтева на који се односе. Према овом критеријуму, тестови се деле на:

1. *Тестове за проверу функционалности* - којима се испитује да ли програм пружа функционалност која је описана захтевима. Значи, овим тестовима се не проверава брзина рада програма или безбедност већ само да ли програм (или неки његов део) без грешке ради оно што је описано у функционалним захтевима. Ови тестови се даље деле према величини дела програма на који се односе на: појединачне тестове (*unit test* - односе се на појединачне класе), интегративне тестове (тестирање више класа које се међусобно позивају у раду), системске тестове (тестирање целог програма од стране програмера) и тестове прихватљивости (*acceptance test* – тестирање целог програма од стране крајњег корисника).
2. *Тестове за проверу нефункционалних карактеристика* - испитује се да ли су нефункционални захтеви испуњени. Нефункционални захтеви се најчешће тичу брзине рада или скалабилности програма (могућност опслуживања већег броја корисника истовремено задовољавајућом брзином), али се могу тицати и: безбедности програма (security), интероперабилности, компатибилности и других нефункционалних карактеристика. Према томе, ови тестови се даље деле на: тестове оптерећења, тестове преоптерећења, тестове безбедности итд.

Поред ове класификације, постоји и класификација тестова према критеријуму познавања кода који се тестира. Према овом критеријуму тестови се деле на [3]:

1. *Тестове засноване на принципу „црне кутије“* (*black-box*) - ови тестови се пишу тако као да се не познаје структура кода који се тестира. Другим речима, код се посматра као црна кутија - за дате улазе се тестирају очекивани излази, док се занемарује како код интерно функционише. Када се, на пример, тестирају класе тестирају се само њихове јавно доступне методе. Предност овог приступа се састоји у томе да промене интерне структуре метода не утичу на ефективност тестова. Мана овог принципа је у томе што се, због „непознавања“ унутрашње структуре кода, може пропустити тестирање неког дела кода.

2. *Тестове засноване на принципу „беле кутије“* (*white-box* - негде се помиње и као приступ *„стаклене кутије“* тј. *glass-box*) - тестови се пишу у складу са структуром кода који се тестира. Овај принцип је обрнут у односу на принцип црне кутије. Тестира се све: јавни, приватни и заштићен код. Тестирају се и вредности приватних променљивих, и алгоритми по којима функционишу методе. Предност овог приступа је у томе што се као метрика може користити покривеност тестовима. *Покривеност тестовима* (*test coverage*) је показатељ који указује на то који је проценат изворног кода покривен бар неким тестом. Што је ова вредност виша, то је и бољи резултат тестирања. Мана овог приступа је то што је велики део тестова неупотребљив чим се код који се тестира промени.
3. *Тестове засноване на принципу „сиве кутије“* (*grey-box*) - принцип „сиве кутије“ је негде између претходна два принципа. Идеја је у томе да се тестови пишу као да се користи принцип црне кутије (тестирају се искључиво јавно доступне методе), али да се користи знање о функционисању кода који се тестира. На овај начин се остварују предности и избегавају мане претходна два принципа: тестови остају валидни и ако се промени неки део кода, а код је боље истестиран јер се тестови пишу за ситуације за које се највише сумња да ће представљати проблем.

Проблем са којим се често сређу ученици, истраживачи, научници или студенти је тај да у тренутку када је потребно да тестирају програм који су развили нису баш сигурни како да тоураде. Тада постају актуелна следећа питања:

- Одакле би требало почети са тестирањем?
- Шта би све требало да се провери тестовима?
- Колико тестова би требало написати?
- Којим редоследом би требало писати тестове?
- Које класе (или методе) се морају тестирати, а које не?

У недостатку јасног плана, неки ће кренути са писањем тестова на основу интуиције и познавања кода, други ће се ослонити на своје искуство, али је чињеница да су ови приступи несистематични, и не пружају одговарајуће резултате. Оно што је потребно је управо један практичан поступак за тестирање. Овим поступком би се дефинисали кораци у процесу писања тестова, њихова међузависност и принципи које би требало испоштовати.

Предмет и циљ рада

Предмет овог рада је један *практичан поступак за функционално тестирање програмских класа.* У недостатку оваквог поступка, ученици, студенти, истраживачи и сви који се не баве професионално тестирањем програма су препуштени сами себи, јер су поступци који су углавном заступљени: компликовани, неинтуитивни или једноставно нису систематични. С обзиром на то да је, у принципу, релативно једноставно тестирати перформансе, безбедност и друге нефункционалне карактеристике програма, поступак је усмерен ка функционалном тестирању. У последње време програмирање се најчешће обавља коришћењем

објектно оријентисаних језика па је, ради једноставности примене, за предмет поступка узета основна градивна јединица - програмска класа.

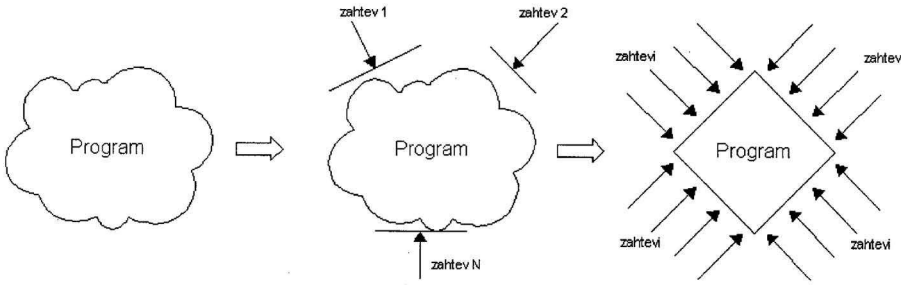
Циљ рада је да се пружи поступак за функционално тестирање класа који би био:

- *прилагођен ученицима, студентима, истраживачима и свима који се не баве тестирањем професионално;*
- *интуитиван и једноставан за примену;*
- *релативно независан од конкретних алата за тестирање или технологија;*
- *систематичан и комплетан;*
- *погодан за аутоматизацију - у смислу генерисања и извршавања тестова.*

Поступак за функционално тестирање програмских класа

Према претходно наведеној дефиницији, функционалним тестовима се врши провера функционалности коју пружа програм. Та функционалност би требало да буде у складу са функционалним захтевима - који описују шта би програм требало да ради. Пример функционалног захтева је: „програм би требало да омогућава израчунавање тачног износа плате на основу унапред дефинисаног система бодовања“. Поставља се питање: како на систематичан и организован начин претворити ове функционалне захтеве у скуп одговарајућих тестова?

Ако се крене од тога да захтеви, у ствари, одређују програм, може се видети следеће. У почетку, када захтеви још не постоје, програм може да буде било какав и да ради било шта. Када се неки од захтева одреде, програм већ постаје благо ограничен у вези са тим шта ради и како то ради. Дефинисањем свих захтева, програм постаје једна јасно ограничена целина. Границе програма чине управо ти захтеви (слика 1).



(а) Захтеви још нису дефинисани (б) Захтеви се дефинишу (в) Захтеви су дефинисани

Сл. 1. Формирање граница програма на основу захтева

Функционални захтеви чине само део граница па се може рећи следеће: *функционални захтеви су, у ствари, ограничења над логичком структуром и понашањем програма.* При томе, сваки захтев се може посматрати као једно огра-

ничење. У контексту тестирања, може се рећи да је за потпуно тестирање програма потребно проверити да ли систем задовољава сва ограничења. У овом приступу су идентификоване три врсте ограничења која настају на основу функционалних захтева:

1. *Ограничења над атрибутима (вредносна ограничења)* - утичу на могуће вредности атрибута система.
2. *Ограничења над асоцијацијама (структурна ограничења)* - односе се на асоцијације (везе) између елемената система, њихов референцијални интегритет и кардиналности.
3. *Ограничења над методама (ограничења понашања)* - односе се на методе система и утичу на његово понашање.

Када се ова ограничења пренесу на ниво појединачних елемената програма - конкретних класа, може се закључити да управо она дефинишу сваку класу тј. њене атрибуте, асоцијације и методе (понашање). Наравно, на сваку класу се односи само одређени број ограничења, а не сва. После даље разраде, добија се комплетна класификација ограничења над класом која настају на основу функционалних захтева:

1. *Ограничења над атрибутима класе (вредносна ограничења)*
   1.1. *Ограничења типа атрибута* - дефинишу тип вредности које атрибут неке класе може да има, нпр. атрибут „`matični_broj`“ класе „`Čovek`“ може да има само вредности „`int`“ типа.
   1.2. *Ограничења дозвољеног скупа вредности атрибута* - распон вредности за неки конкретан атрибут, нпр. атрибут „`matični_broj`“ класе „`Čovek`“ мора да буде број од тачно тринаест цифара при чему вредност последње цифре зависи од претходних, а првих седам цифара дефинише неки датум. Вредност овог поља не сме да се изостави нити може да буде мања од нуле.
   1.3. *Ограничења на међузависности вредности атрибута једне класе* - ова ограничења се односе на зависност вредности неког конкретног атрибута у односу на вредности атрибута исте класе. Постоје две врсте ових ограничења:
   1.3.1. *Ограничења на вредности истог атрибута у односу на друга појављивања исте класе* - ово се првенствено односи на идентификаторе тј. примарне кључеве („`PRIMARY KEY`“) и јединствене („`UNIQUE`“) вредности. На пример, поље „`matični_broj`“ мора да има јединствену вредност у сваком појављивању класе „`Čovek`“ јер представља идентификатор и може да се користи као примарни кључ.
   1.3.2. *Ограничења на вредности различитих атрибута у оквиру истог појављивања класе*, нпр. атрибути „`prodajni_kurs`“, „`srednji_kurs`“ и „`kupovni_kurs`“ класе „`Valuta`“ (средњи курс је увек између ова два).
   1.4. *Ограничења на међузависности вредности атрибута више класа* - ова ограничења се односе на вредности атрибута које зависе од вредности атрибута неких других класа. На пример, поље „`ukupan_iznos`“ класе „`Račun`“ је сума поља „`iznos_stavke`“ класе „`Stavka_računa`“.

2. *Ограничења над асоцијацијама класе (структурна ограничења)*
   2.1. *Ограничења кардиналности веза* - свака асоцијација између две класе има своју доњу и горњу кардиналност. Ове две вредности одређују минималан и максималан број објеката једне класе који могу да буду у вези са једним објектом друге класе и обрнуто. На пример, класа „`Račun`“ може да буде у вези са једним или више појављивања класе „`Stavka_računa`“ (сваки рачун има једну или више ставки), док је једно појављивање класе „`stavka_računa`“ увек у вези са само једним појављивањем класе „`račun`“ (ставка увек припада само једном конкретном рачуну).
   2.2. *Ограничења референцијалног интегритета* - ова ограничења се односе на ситуацију када се једно од два појављивања класе која су у вези обрише или измени. У контексту релационих база података, ово су „`ON UPDATE`“ и „`ON DELETE`“ ограничења. Практичан пример је следећи: ако се промене подаци појављивања класе „`proizvod`“, потребно је променити референце свих појављивања класе „`stavka_računa`“ које садрже тај производ на ново појављивање.
3. *Ограничења над методама класе (ограничења понашања)*
   3.1. *Ограничења типа вредности параметара метода* - ова ограничења се односе на типове параметара метода. На пример, метода „`unesi_račun`“ може да садржи само један параметар - појављивање класе „`Račun`“.
   3.2. *Ограничења скупа вредности параметара метода* - ограничење скупа вредности које се могу унети као параметри метода. Ако, на пример, метода „`unesi_račun`“ садржи само један параметар типа „`Račun`“, није дозвољено да овај параметар има „`null`“ вредност.
   3.3. *Ограничења понашања под условом да су прва два типа ограничења задовољена* - ова ограничења се односе на понашање методе у ситуацији да су унети параметри у оквиру претходних ограничења. Нека метода „`unesi_račun`“ садржи само један улазни параметар типа „`Račun`“. Ако се метода позове са одговарајућом вредности параметра, она би требало да (ограничења понашања су):
      - ако рачун не постоји у складишту података, метода га уноси
      - ако рачун већ постоји у складишту, метода обавештава корисника о томе.

Поступак за функционално тестирање класа који се предлаже у овом раду се састоји у провери ових ограничења за сваку класу и то у редоследу који је дат у класификацији. Другим речима, тестови се пишу тако да провере да ли су сва ограничења над конкретном класом заиста и имплементирана у оквиру ње.

Извор података о ограничењима је сама документација програма - било да је у питању обичан текстуални опис или су то детаљни UML дијаграми [5]. Прве две врсте ограничења се могу извести из текстуалних описа, концептуалног модела, релационог модела или дијаграма класа, док се трећа врста ограничења такође може извести из текстуалних описа, али и случајева коришћења, секвенцијалних дијаграма или дијаграма сарадње.

Редослед тестирања класа - препорука је да се класе тестирају у редоследу имплементације и то паралелно са имплементацијом. То значи следеће: написати (имплементирати) класу, одмах је тестирати па тек онда прећи на имплементацију следеће класе. На тај начин се обезбеђује сигурност да оно што је већ имплементирано функционише без грешке и избегава се ситуација да је програм потребно тестирати у целини одједном.

Садржај и број тестова за конкретну класу зависи од ограничења које та класа имплементира. Битно је напоменути да свака класа имплементира само нека од ограничења и да је тестирање тих ограничења битно само за ту класу. У принципу, *сваки атрибут класе је одређен са једним или више ограничења атрибута, свака асоцијација са једним или више ограничења асоцијација, а свака метода са једним или више ограничења метода. Тестирање конкретне класе се састоји у провери ограничења за сваки атрибут, асоцијацију и методу те класе.* На тај начин се остварује систематичност јер ништа не остаје непроверено. Провера ограничења се може вршити на два начина. Први начин је *покушај нарушавања ограничења* - покуша се унос неке недозвољене вредности, типа, остваривање недозвољене асоцијације, кардиналности асоцијације итд. Други начин је *провера спровођења ограничења* - провера да ли се метода заиста извршила у складу са ограничењем понашања, да ли се брисањем или изменом неког објекта заиста освежавају везе других објеката према њему итд. Још једна напомена је то да се при тестирању користи принцип „*црне кутије*“ тј. тестирање се врши позивањем само јавно доступних метода класе.

Показатељ за процену квалитета спровођења поступка се може назвати *степен покривености ограничења тестовима.* Овај показатељ представља процентуално изражен однос броја ограничења класе која су проверена тестовима и укупног броја ограничења (за дату класу). Што је показатељ већи, то је спроведено тестирање квалитетније. Наравно, када су сва ограничења проверена, степен покривености је 100% и тестирање је завршено. Овај показатељ се може користити на нивоу класе, али и на нивоу целог програма.

$$SPOT = \frac{N_t}{N} * 100\%$$
$$N = N_{at} + N_{as} + N_m$$

где је:

$SPOT$ - степен покривености ограничења тестовима,

N_t - укупан број ограничења класе која су проверена тестовима,

N - укупан број ограничења за дату класу,

N_{at} - број ограничења над атрибутима класе (вредносна ограничења),

N_{as} - број ограничења над асоцијацијама класе (структурна ограничења),

N_m - број ограничења над методама класе (ограничења понашања).

Иако наизглед сложен, овај поступак се веома лако користи. То портврђује и низ практичних примера који се могу видети у [7]. Примери су написани у складу са поступком, а односе се на тестирање Java програма. Као што поступак налаже, креће се од једноставнијих тестова којима се проверавају ограничења вредности.

То је у складу са редоследом имплементације класа, па је цео процес логичан и једноставан: имплементира се класа или неки њен део, па се тестира (тестирају се ограничења која би та класа требало да садржи). После се прелази на тестирање ограничења асоцијација, да би се завршило са тестирањем ограничења понашања. Да би се указало на једноставност аутоматизације извршавања овако написаних тестова, све време се користи JUnit - који је практично стандард међу бесплатним алатима за тестирање Java програма [8].

Закључак

Поступак описан у овом раду испуњава све задате циљеве. У питању је поступак који је првенствено намењен научницима, истраживачима, студентима и свима који се не баве тестирањем професионално. Он има карактеристике систематичности и комплетности али задржава једноставност у примени и интуитивност који су неопходни да би био прихваћен. За његову примену нису потребни никакви посебни алати нити познавање специјалних техника, већ је довољно имати само пар дијаграма или описа система који се тестира.

Неки од могућих даљих праваца страживања су: истраживање начина за аутоматско генерисање тестова на основу овог поступка и прављење генератора тестова на основу поступка.

Литература

[1] Guide to the Software Engineering Body of Knowledge, IEEE SWEBOK, 2004., доступно у електронској форми на интернет адреси: `www.computer.org/portal/cms_docs_ieeecs/ieeecs/education/certification/Swebok_2004.pdf`

[2] Software testing, *Wikipedia the free encyclopedia*, доступно у електронској форми на интернет адреси: `http://en.wikipedia.org/wiki/Software_testing`

[3] Expert One-on-One J2EE Design and Development, Rod Johnson, Wrox press, 2003., ISBN 0764543857.

[4] ISO 9126 standard, International Standards Organisation (ISO), доступно у електронској форми на адреси: `http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=22749& ICS1=35& ICS2=80& ICS3=`

[5] UML ukratko, Martin Fowler, Mikro knjiga, prevod 3. izdanja, 2004., ISBN 86-7555-239-4.

[6] Projektovanje programa (SKRIPTA), Siniša Vlajić (FON), sopstveno izdanje, 2004., Beograd.

[7] Design patterns, Erich Gamma; Richard Helm; Ralph Johnson; John Vlissides, AddisonWesley, 1999.

[8] Testiranje Java programa korišćenjem JUnit alata: Praktikum sa dodatnim objašnjenjima i postupcima za NetBeans i Eclipse razvojna okruženja za Javu, Bojan Tomić, Zlatni presek, 2007., Beograd, ISBN 978-86-86887-01-6.

[9] JUnit - бесплатан алат за тестирање Java програма, јавно доступан на интернет адреси: `www.junit.org`

Факултет организационих наука, Универзитет у Београду, Јове Илића 154, 11000 Београд
E-mail: `vlajic@fon.bg.ac.yu, bishop@drenik.net`