UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Robert Koritnik

METODE ZA TESTIRANJE PROGRAMSKIH APLIKACIJ IN NJIHOVA TRŽNA RAZŠIRJENOST

DIPLOMSKO DELO
NA UNIVERZITETNEM ŠTUDIJU

Mentor: prof. dr. Dušan Kodek

LJUBLJANA, 2010

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 01674/2010
Datum: 05.04.2010

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **ROBERT KORITNIK**

Naslov: **METODE ZA TESTIRANJE PROGRAMSKIH APLIKACIJ IN NJIHOVA TRŽNA RAZŠIRJENOST**

METHODS FOR SOFTWARE APPLICATIONS TESTING AND THEIR MARKET PENETRATION

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Napake v programih in na njih zasnovanih aplikacijah so neizogibni sestavni del razvoja programske opreme. Od vsega začetka je zato obstajalo tudi testiranje, ki se je do danes razvilo v vrsto različnih pristopov in testnih orodij. Čeprav je sistematična in pravilna uporaba teh orodij pogoj za kvaliteto programskih aplikacij, je njihova uporaba v resničnih razvojnih okoljih pogosto neustrezna. Za ugotovitev današnjega stanja naredite raziskavo, iz katere bo razvidno, katera testna orodja in v kolikšni meri, se uporabljajo v podjetjih, ki razvijajo aplikacije. Naredite pregled najpogosteje uporabljanih testnih orodij in metod ter opišite njihove prednosti in slabosti. Njihovo uporabo ilustrirajte s primeri spletnih in namiznih aplikacij v tehnologiji ASP.NET MVC in v programskem jeziku C#.

Mentor:

prof. dr. Dušan Kodek

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan:

prof. dr. Nikolaj Zimic

Izjava o avtorstvu

Spodaj podpisani Robert Koritnik,
z vpisno številko 24930483,
sem avtor diplomskega dela z naslovom:

Metode za testiranje programskih aplikacij in njihova tržna razširjenost.

S svojim podpisom zagotavljam, da

- sem diplomsko delo izdelal samostojno pod mentorstvom
  prof. dr. Dušana Kodeka

- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela

- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki »Dela FRI«

V Ljubljani, dne 21. 10. 2010 Robert Koritnik

Zahvala

Hvala mentorju prof. dr. Dušanu Kodeku za vse nasvete in vodenje pri izdelavi diplomskega dela. Brez njegovega kritičnega pogleda bi bilo neustrezno in na mnogo nižji kvalitetni ravni.

Hvala celotni družini, ki me je vedno podpirala in spodbujala.

Posebej bi se rad zahvalil moji ženi Jerneji za vso spodbudo, podporo, predvsem pa obilno potrpljenje. Brez njenega truda doma bi izdelava diplomskega dela trajala mnogo dlje, kot je sicer. Hvala ti.

Kazalo

Seznam uporabljenih kratic

AAA	(angl.) Arrange–Act–Assert – vzorec zaporedja funkcionalnosti pri pisanju testnih funkcij npr. pri testiranju enot
AJAX	(angl.) Asynchronous Javascript And XML – skupek tehnologij na uporabniški strani (v spletnem brskalniku), ki omogočajo izvedbo interaktivnih spletnih strani oziroma spletnih aplikacijskih rešitev
ASP.NET	(angl.) Active Server Pages .Net – microsoftova tehnološka naslednica tehnologije ASP, ki služi razvoju dinamičnih spletnih strani, spletnih aplikacij in servisov
CSS	(angl.) Cascading Style Sheets – oblikovna semantika spletnih strani, ki definira vizualno podobo posameznih HTML elementov
DI	(angl.) Dependency Injection – vstavljanje odvisnosti oziroma odvisnih razdruženih funkcionalnih enot
HTML	(angl.) HyperText Markup Language – jezik spletnih strani svetovnega spleta
HTTP	(angl.) HyperText Transfer Protocol – temeljni komunikacijski podatkovni protokol svetovnega spleta
IOC	(angl.) Inversion Of Control – princip obrnjenega nadzora, ki se uporablja pri razdruževanju funkcionalnih enot
JAVA EE	(angl.) Java Enterprise Edition – tehnologija za razvoj spletnih aplikacij z uporabo jezika Java
JSON	(angl.) JavaScript Object Notation – enostaven in berljiv tekstovni format namenjen podatkovni izmenjavi najpogosteje uporabljen v skriptnem jeziku Javascript
MVC	(angl.) Model–View–Controller – arhitekturni vzorec razvoja aplikacijskih rešitev pri čemer so osnovne funkcionalne enote razdeljene v funkcionalnosti poslovne logike (controller), podatkov (model) in logike uporabniškega vmesnika (view)
PHP	(angl.) Personal Home Page – popularna odprtokodna tehnologija za razvoj spletnih aplikacij; današnja kratica ima pomen Hypertext Preprocessor
POCO	(angl.) Plain Old Class Object – zelo koheziven razred, katerega primarna naloga je hranjenje podatkov določene entitete; najpogosteje se uporablja za izmenjavo podatkov med posameznimi plastmi aplikacijske rešitve
REST	(angl.) REpresentational State Transfer – arhitekturni vzorec za distribuirane spletne aplikacije
SOC	(angl.) Separation Of Concerns – ločevanje logičnih funkcionalnih enot na posamezne medsebojno navidez neodvisne dele
TDD	(angl.) Test Driven Development – metodologija razvoja aplikacijskih rešitev, pri kateri testiranje predstavlja poglavitni del razvojnega cikla
UI	(angl.) User Interface – uporabniški vmesnik
WATIX	(angl.) Web Application Testing In »X« – orodje za testiranje spletnih aplikacij, pri čemer X lahko zamenjamo z R (Ruby), N (.Net), J (Java) itd.
XHTML	(angl.) eXtensible HyperText Markup Language – XML standardiziran zapis HTML
XML	(angl.) eXtensible Markup Language – tekstovni standard berljivega in zelo prilagodljivega formata za izmenjavo podatkov

Povzetek

Razvoj aplikacijskih rešitev že dolgo ni več sestavljen le iz pisanja programov in razhroščevanja programskih napak. V profesionalnih okoljih razvojnih skupin je danes v proces razvoja vključeno tudi zelo sistematično testiranje različnih aspektov rešitve, katerega rezultati so merljivi in sledljivi. Cilj sodobnega testiranja zato ni več le iskanje in odpravljanje programskih napak, temveč tudi njihovo nadaljnje preprečevanje, ki tako izboljša kvaliteto končne aplikacijske rešitve kot tudi njeno lažje vzdrževanje.

Vsebina diplomskega dela je osredotočena na področje testiranja aplikacijskih rešitev s poudarkom na njihovi avtomatizaciji. Našteta so najpogosteje uporabljana sodobna testna orodja ter knjižnice, čemu so namenjena in pri nekaterih tudi njihov specifičen način uporabe s primeri kratkih programov.

Za uspešno izvajanje avtomatiziranega testiranja je potrebno primerno prilagoditi pisanje programov. Razloženi so takšni vzorci, ki izboljšajo testnost in omogočajo avtomatizacijo. Naštete so testne metode in najpomembnejši testni nivoji ter katere je možno avtomatizirati in na kakšen način. Največji poudarek je na testiranju enot.

Kvantitativni podatki v vsebini diplomskega dela se nanašajo na rezultate posebej za ta namen opravljene raziskave. V okviru raziskave je bila opravljena analiza uporabe sistematičnega testiranja aplikacijskih rešitev v realnih profesionalnih razvojnih skupinah.

Ključne besede

testiranje aplikacij, testnost, testne metode, avtomatizacija testiranja

Abstract

In the pursuit of quality, software development teams are adopting new and innovative software testing techniques which allow them to deliver better, faster and more reliable applications to their clients. Indeed, structured software testing allows development teams to track and measure software quality in a number of areas including code conformance and business process rules regularity as well as meet non-functional requirements such as scalability and security. Thus software testing has transcended its original purpose of finding basic code errors and moved on to provide future bug prevention mechanisms that enhance an application's quality, stability and maintainability.

This thesis focuses on the growing discipline of software testing with a particular emphasis on automated testing. The most frequently used testing tools and techniques are described along with their common usage. Code snippets and explanations are provided where relevant.

To successfully incorporate automated testing, software development teams have to adjust their coding patterns. This thesis explains the changes that need to be made to enhance software testability and allow for test automation. Test methods, as well as test levels, are explained in detail and an indication is given as to which methods may be automated and how this can be achieved. Special emphasis has been given to unit testing.

Specifically for the purpose of this thesis, I have performed an original research into the practices of professional software development teams with regard to their use of systematic software testing techniques. Results of this original research analysis make up the bulk of the quantitative data presented herein.

Keywords

software testing, testability, test methods, test automation

1 Uvod

Odkar obstaja razvoj aplikacijskih rešitev so obstajale napake v programih in napačno razviti poslovni procesi. Zato je od vsega začetka obstajalo tudi testiranje. V takšni ali drugačni obliki. Ker so bile sprva aplikacije trivialne narave, je bilo takšno tudi njihovo testiranje – predvsem razhroščevanje programov. Premosorazmerno z naraščanjem kompleksnosti aplikacijskih rešitev je postajalo kompleksnejše tudi njihovo testiranje. Danes je s pravilnim pristopom sistematično urejeno, sledljivo in merljivo. Poleg najosnovnejše in najstarejše oblike testiranja v obliki razhroščevanja napak v programih danes sistematično testiranje poizkuša preprečevati nadaljnje napake in zagotavlja stabilno delovanje obstoječih funkcionalnosti kljub morebitnemu dograjevanju novih poslovnih procesov.

Bistvo testiranja je torej zagotavljanje pravilne implementacije aplikacijskih zahtev in preprečevanje napak ter neregularnosti v programih in vprogramiranih poslovnih procesih. Zato s sistematičnim testiranjem poleg stabilnejše aplikacije posredno dobimo tudi neke vrste zagotovilo, da so poslovni procesi regularni v okviru predhodne in sprotno komunicirane specifikacije delovanja poslovnih procesov. To pomeni, da je aplikacija lahko dobra le v tolikšni meri, kolikor je dobra specifikacija in njeno razumevanje.

Kljub vsemu se je potrebno zavedati dejstva, da še tako dobra specifikacija in sistematično celostno testiranje ne zagotavljata stoodstotno regularno delujoče aplikacije. Zagotavljata le mnogo zanesljivejšo in boljšo rešitev kakor bi bila razvita brez ali le z ad-hoc testiranjem. Popolnih aplikacijskih rešitev predvsem zaradi človeškega faktorja ni in ne more biti.

Eden izmed večjih vzrokov slabe kvalitete kompleksnih aplikacijskih rešitev je v nesistematičnem in površnem izvajanju razvojnega procesa, ki je sestavljen iz razvoja, testiranja, umestitve v končno okolje in vzdrževanja. V sodobnih razvojni metodologijah kot je TDD (angl. *Test Driven Development*) se ta proces izvaja iterativno, saj s tem omogoča mnogo cenejše odpravljanje programskih napak.

V današnjem času razvojne skupine najpogosteje razvijajo spletne aplikacijske rešitve. S primernim načrtovanjem in izbiro razvojnih tehnologij lahko močno izboljšamo testnost aplikacij ter posredno omogočimo avtomatizirano testiranje. Zaradi izboljšane prilagojenosti testnosti je izbira tehnologije Asp.net MVC več kot smotrna, saj močno olajša izvajanje avtomatiziranega testiranja. Vzporedno s to tehnologije je potem izbira sodobnega objektno orientiranega programskega jezika C# trivialna.

1.1 Raziskava uporabe sistematičnega testiranja aplikacijskih rešitev

V okviru diplomskega dela sem opravil raziskavo uporabe sistematičnega testiranja aplikacijskih rešitev v okolju profesionalnih razvojnih skupin. Raziskavo sem opravil v septembru 2010.

Kadar so kjerkoli v vsebini tega diplomskega dela navedeni kvantitativni podatki (običajno odstotne vrednosti) in njihov vir ni posebej naveden, so le-ti rezultat analize te raziskave.

Rezultati raziskave so potrdili moje razvojne izkušnje in predvidevanja, da se testiranje večinoma izvaja neustrezno, pogostokrat pa sploh ne. Glede na to, da je razvoj aplikacijskih rešitev iz finančnega stališča draga storitev, bi bila za podjetja, katerih primarna ali sekundarna dejavnost je razvoj aplikacijskih rešitev, smotrna uvedba sistematičnega testiranja. Moje lastne izkušnje kažejo, da je predaja projektov pogosto izvedena prezgodaj, zato so končne aplikacijske rešitve nestabilne in vsaj delno neustrezne. Prezgodaj predvsem zato, ker so projekti zaradi cenovnih okvirov optimistično načrtovani, zaradi česar razvojne skupine ne posvetijo dovolj časa testiranju in odpravi programskih napak.

Raziskava je pokazala, da skoraj polovica razvojnih skupin (49,2%) nima posebnega člana, katerega primarno delo je testiranje razvijanih aplikacijskih rešitev – testerja. Le dobra petina (20,6%) razvojnih skupin ima testerje, ki izvajajo testiranje na sistematičen način, torej je merljiv in sledljiv. Pri kompleksnejših aplikacijskih rešitvah je takšna oseba nepogrešljiva za izvedbo kvalitetne in stabilne aplikacijske rešitve.

Pri testiranju aplikacijskih rešitev je pomemben prihranek virov tudi vključitev avtomatiziranega testiranja v proces razvoja. S pomočjo avtomatizacije lahko razvojne skupine hitro in sprotno nadzorujejo kvaliteto in stabilnost rešitve. Žal rezultati raziskave zopet kažejo, da je tudi uvedba avtomatizacije dokaj nizka. Med razvojnimi skupinami, ki razvijajo spletne aplikacije, jih kar polovica (50%) ne izvaja avtomatiziranega testiranja. Pri ostalih skupinah je ta odstotek nekoliko nižji (36%) vendar še vedno visok.

Celotna analiza rezultatov raziskave se nahaja v poglavju B.2 na strani 43.

1.2 Cilji

Glede na rezultate opravljene raziskave in posredno dejstvo, da se sistematično testiranje aplikacijskih rešitev v profesionalnih razvojnih okoljih premalo ali neustrezno izvaja, je glavni cilj diplomskega dela praktični prikaz primernega testiranja aplikacijskih rešitev. V vsebinski okvir so zajeti vzorci in razvojne prakse, ki pripomorejo k učinkovitejšemu testiranju. Poseben poudarek je na avtomatiziranem testiranju enot, ki pri razvoju kompleksnejših aplikacijskih rešitev dolgoročno prihrani čas in omogoča lažji ter stabilnejši razvoj. Predvsem je to pomembno za tiste razvojne skupine, v katerih je več razvijalcev (lahko že dva v kolikor njuna komunikacija ni ustrezna).

Posredno bi področje testiranja rad praktično približal vsem članom razvojnih skupin (razvijalcem, vodstvenemu kadru ter vsem, ki so vključeni v proces razvoja), kateri menijo, da je testiranje aplikacijskih rešitev predvsem potratnost virov in h končni kvaliteti aplikacijske rešitve ne doprinese toliko, kolikor virov (človeških, časovnih in finančnih) potroši. Skratka vsem tistim, ki menijo, da je testiranje predvsem strošek brez prave vrednosti.

2 Sodobne razvojne tehnologije in testna orodja

Danes imajo razvijalci na voljo več razvojnih tehnologij in platform kot kdajkoli prej. Z vse večjo informatizacijo, hitrejšo povezanostjo preko svetovnega spleta ter vedno novimi in zmogljivejšimi pametnimi napravami ta trend še dodatno narašča. Izbira primernih tehnologij je zato vse prej kot lahka. Pri tem imajo pomembno vlogo različni dejavniki, njihova pomembnost pa je drugačna za vsak projekt ali razvojno skupino:

- podpora primerni pratformi – npr. za namizne, spletne ali mobilne aplikacije;
- preizkušenost tehnologije – starost oziroma trajanje prisotnosti na trgu;
- razvojna podpora – plačljiva ali brezplačna v obliki številčne in žive spletne skupnosti;
- sodobnost – podpora sodobnim aplikacijskim vzorcem in razvojnim prostopom;
- enostavnost uvedbe avtomatiziraneag testiranja;
- splošna razširjenost uporabe ipd.

Za vsebino tega diplomskega dela je najpomembnejši dejavnik prav enostavnost uvedbe in izvajanja avtomatiziranega testiranja.

2.1 Razvojne tehnologije

V okviru raziskave, ki sem jo opravil, so bili anketiranci vprašani o platformah za katere razvijajo aplikacijske rešitve ter katere tehnologije in jezike pri tem uporabljajo. Zaradi vsesplošne dostopnosti in posredno velikega tržnega potenciala ter enostavnosti posodabljanja ter nameščanja, je pričakovano največje število razvojnih skupin usmerjenih v razvoj spletnih aplikacijskih rešitev. Najpogostejše platforme za razvoj spletnih rešitev so:

- Asp.net WebForms – (ASP angl. *Active Server Pages*) microsoftova najbolj razširjena in preizkušena platforma, ki je na trgu že desetletje (trenutna različica 4.0). Omogoča hitro izdelavo spletnih aplikacij. Avtomatizirano testiranje je brez namerne uvedbe posebnih aplikacijskih vzorcev oteženo. Zaradi abstrakcije dogodkovnega modela je implementacija kompleksnih procesov uporabniškega vmesnika dokaj zapletena.
- Asp.net MVC – (MVC angl. *Model–View–Controller*) sodobna microsoftova knjižnica (na trgu približno 2 leti in pol – trenutna uradna različica 2.0, v izdelavi že 3.0), ki je osnovana na platformi Asp.net. Ena izmed osnovnih zahtev pri izdelavi knjižnice je bila izboljšanje testnosti, zato je implementacija avtomatiziranih testov zelo poenostavljena.
- PHP – preizkušena in najbolj razširjena odprtokodna platforma, ki je na trgu že 15 let (trenutna različica 5.3). Z uporabo primernih knjižnic in s prilagojenim pisanjem programov, je avtomatizacija testiranja mogoča.

- Silverlight – napredna microsoftova tehnologija za izdelavo bogatih (angl. *rich*) spletnih in tudi povezanih namiznih aplikacijskih rešitev (trenutna različica 4.0). Z uporabo primernih knjižnic in prilagojenim pisanjem programov, je avtomatizirano testiranje omogočeno. Uporaba na odjemalcih je mogoča le z namestitvijo knjižnice (slabost).
- Java EE – zelo preizkušena tehnologija za izdelavo spletnih aplikacijskih rešitev pogosto v uporabi v finančnem sektorju (trenutna različica 6). Z uporabo primernih knjižnic je avtomatizirano testiranje omogočeno.

Opravljena raziskava je pokazala, da razvojne skupine, ki ne razvijajo spletnih aplikacij, več ali manj razvijajo namizne aplikacije (92%) in sicer največ za okolje Microsoft Windows. Od teh jih 81% razvija v .net okolju z microsoftovimi tehnologijami. Za implementacijo avtomatiziranega testiranja je priporočeno, da te razvojne skupine uporabljajo aplikacijske vzorce in knjižnice, ki jim omogočajo avtomatizacijo testiranja.

Ker je razvoj spletnih rešitev bolj razširjen bi se od vseh tehnologij rad osredotočil na tisto, ki že v osnovi brez uporabe dodatnih knjižnic omogoča najboljšo testnost (začetek poglavja 3 na strani 13). To je tehnologija oziroma skeletna knjižnica (angl. *framework*) Asp.net MVC, ki privzeto izvaja ločevanje funkcionalnih enot (podrobnosti v poglavju 3.1 na strani 13).

Opravljena raziskava je pokazala, da je najpogosteje uporabljani programski jezik C# in sicer ga uporablja oziroma razume kar 81% vseh razvijalcev. Če upoštevamo še sorodne jezike, ki uporabljajo sintakso jezika C (Java, C++, Objective C, Javascript) je takšnih razvijalcev glede na opravljeno raziskavo kar 95,2%. Zaradi tega so primeri programov v vsebini diplomskega dela napisani v tem jeziku saj so tako razumljivi kar najširšemu krogu razvijalcev.

2.2 Skeletna knjižnica Asp.net MVC

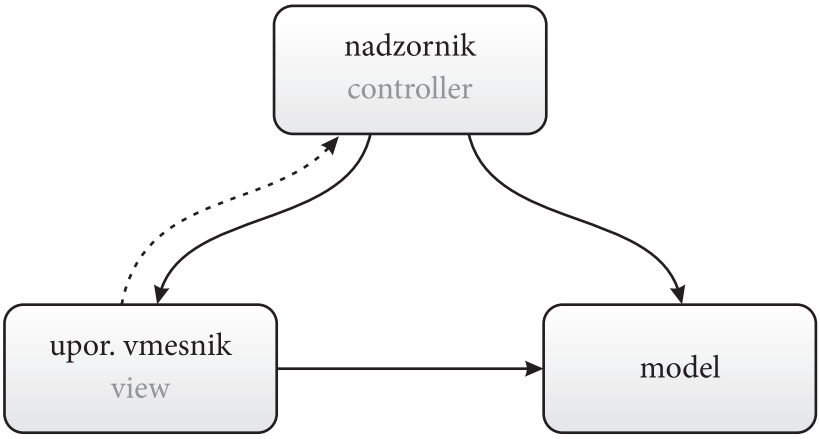
V diplomskem delu so za lažje razumevanje vsebine ponekod navedeni primeri programov v programskem jeziku C#. Povzeti so iz spletne aplikacijske rešitve za upravljanje opomnikov in nalog, ki sem jo napisal za potrebe tega diplomskega dela. Ker sem zaradi podpore avtomatiziranega testiranja želel izpostaviti skeletno knjižnico Asp.net MVC, je bila aplikacija razvita v tej tehnologiji in z uporabo te knjižnice.

Prva različica knjižnice je bila uradno izdana 13. marca 2009, zato jo uvrščamo med sodobne knjižnice za razvoj spletnih rešitev. Pomembnejši povod njenega nastanka je bil zapolnitev vrzeli podpore sodobnim arhitekturnim vzorcem s povečano testnostjo med microsoftovimi tehnologijami. Knjižnica Asp.net MVC s svojo zgradbo in delovanjem zahteva razvoj aplikacij v katerih se privzeto izvaja ločevanje funkcionalnih enot (poglavje 3.1 na strani 13) zato močno poenostavlja implementacijo avtomatiziranega testiranja.

Poleg tega ima skeletna knjižnica Asp.net MVC še mnogo drugih prednosti:

- popoln nadzor nad HTML elementi posameznih pogledov uporabniškega vmesnika;
- enostavna integracija Javascript knjižnic na odjemalski strani;

- podpora REST (angl. *REpresentational State Transfer* [22]) in sodobnim pristopom razvoja spletnih rešitev z uporabo Ajax (angl. *Asynchronous Javascript And XML*) klicev;
- zelo dobra prilagoditev prenosnemu HTTP protokolu, ki v svoji osnovi ne podpira hranjenaja in prenosov stanj (angl. *stateless protocol*).

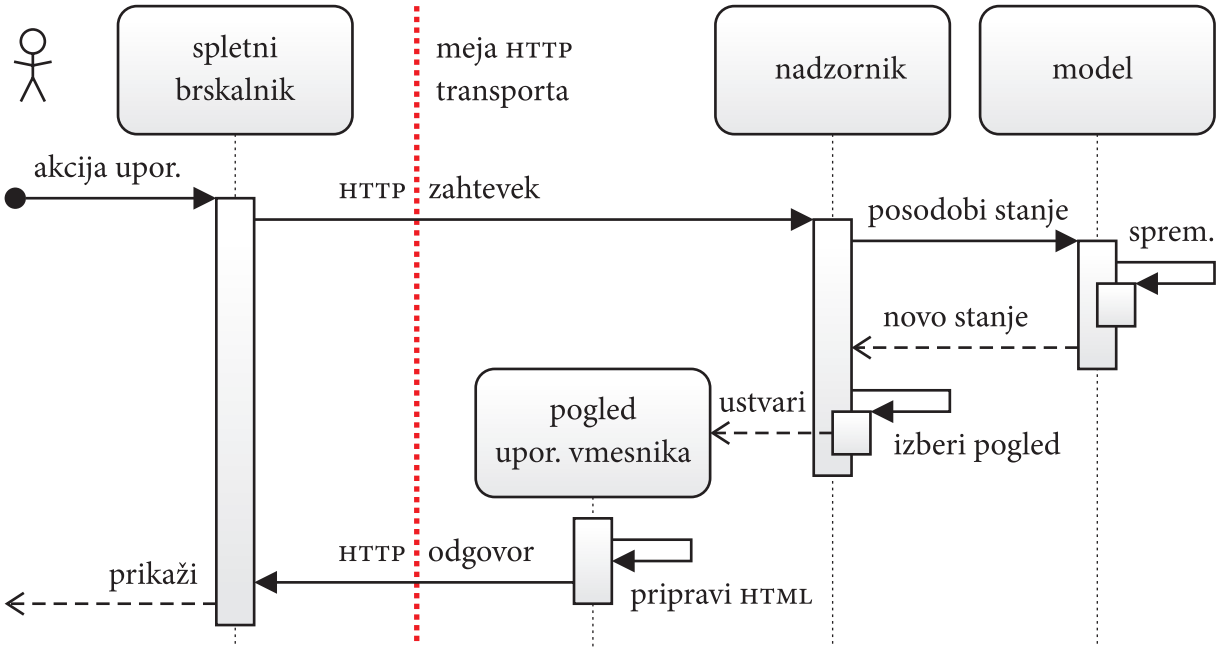


Slika 1. Konceptualna zgradba aplikacijskega razvojnega vzorca MVC, kakršen je implementiran v okviru skeletne knjižnice Asp.net MVC. Sklenjena črta prikazuje neposredno povezanost dveh funkcionalnih enot, prekinjena pa posredno.

2.2.1 Proces delovanja skeletne knjižnice Asp.net MVC

Delovanje procesa konceptualnega modela skeletne knjižnice Asp.net MVC je popolnoma prilagojeno HTTP transportnemu protokolu. Proces ob vsaki uporabniški zahtevi na spletni strežnik poteka na sledeč poenostavljen način:

1. uporabnik v okviru HTML pogleda uporabniškega vmesnika v brskaniku izvede določeno akcijo, ki povzroči pričetek HTTP zahtevka npr. klikne na gumb Ustvari nov opomnik;
2. nadzornik prejme podatke HTTP zahtevka; interno delovanje skeletne knjižnice Asp.net MVC omogoča avtomatično pretvorbo poslanih podatkov v točno določen objekt razreda iz aplikacijskega modela (ni pa nujno) in tudi njegovo preverjanje regularnega stanja v kolikor ga razred tega objekta definira;
3. nadzornik izvede neposreden klic v model, ki je odvisen od uporabnikove akcije npr. pokliče funkcijo za ustvarjanje novega opomnika;
4. model izvede klicano funkcinalnost npr. shrani nov opomnik v podatkovno zbirko;
5. nadzornik glede na uporabnikovo akcijo in podatke vrnjene s strani modela izbere določen HTML pogled uporabniškega vmesnika;
6. nadzornik posreduje podatke modela novo izbranemu pogledu v uporabo;
7. pogled uporabniškega vmesnika pretvori posredovane podatke modela v HTML vsebino primerno za prikazovanje v brskalniku in jo posreduje nazaj brskalniku na uporabnikovi strani kot HTTP odgovor.



Slika 2. UML sekvenčni diagram delovanja procesa skeletne knjižnice Asp.net MVC. Leva stran meje HTTP transporta predstavlja uporabniško stran, desna pa strežniško.

2.3 Testna orodja in knjižnice

Pri pisanju in izvajanju posameznih testov razvojne skupine običajno uporabljajo različna testna orodja in knjižnice. Nekatera izmed njih so brezplačna in dosegljiva preko spleta, druga pa plačljiva, ker pogosto nudijo obsežnejše funkcionalnosti.

Programske knjižnice uporabljajo predvsem razvijalci (ali testerji-razvijalci), da prihranijo razvojni čas. Knjižnice jim hitro in zanesljivo nudijo primerno funkcionalnost pri pisanju avtomatiziranih testov. S tem se zmanjša količina programskih napak v avtomatiziranih testih.

Testna orodja po drugi strani pogosteje uporabljajo testerji (ali pa testerji-razvijalci) z namenom avtomatizacije funkcionalnih testov v okviru uporabniškega vmesnika (metoda testiranja črne škatle je razložena v poglavju 4.1 na strani 23).

Kot so pokazali rezultati opravljene raziskave, se nekatera od teh orodij in knjižnic uporabljajo mnogo pogosteje od drugih. Po pogostosti uporabe si knjižnice in orodja sledijo v sledečem vrstnem redu:

- Visual Studio Unit Test Framework – knjižnica in orodja integrirana v razvojno okolje Microsoft Visual Studio. Orodja omogočajo ustvarjanje testnih projektov in takojšnje izvajanje in sledenje testov znotraj okolja.
- xUnit – eno izmed najbolj razširjenih odprtokodnih orodij za pisanje in izvajanje testov. Poleg orodij za izvajanje avtomatiziranih testov vsebuje tudi knjižnico za njihovo zelo preprosto pisanje. Implementirana je za mnogo različnih okolij oziroma jezikov (predpo-

na X je zato lahko J-Java, N-.net, itd.); v okviru testnih programov tega diplomskega dela je bila uporabljena NUnit različica (spletni naslov: http://www.nunit.org). Poleg testnega orodja in knjižnice je vključeno tudi enostavno orodje za simuliranje funkcionalnih enot (poglavje 5.1.1 na strani 28).

- Moq (spletni naslov: http://code.google.com/p/moq) – popularna specifična knjižnica za .net 3.5 in novejše okolje s pomočjo katere lahko enostavno in hitro izdelujemo simulirane funkcionalne enote z definiranimi tipi brez čarobnih vrednosti (angl. *magic strings* ali *magic values*). Omogoča t.i. tekoči način pisanja programov (angl. *fluent interface*) zaradi česar je uporaba enostavna in časovno izredno učinkovita.
- RhinoMocks (spletni naslov: http://www.ayende.com/projects/rhino-mocks.aspx) – starejša knjižnica za .net okolje za izdelovanje simuliranih funkcijskih enot. Omogoča nekoliko drugačno delovanje od Moq knjižnice.
- WatiX – brezplačno testno orodje namenjeno testiranju spletnih aplikacij. Testi so napisani v skriptnem jeziku za izbrano okolje (končnica X je vezana na okolje, podobno kot pri xUnit knjižnici – N pomeni .net in je dosegljivo na spletnem naslovu: http://watin.sourceforge.net). Omogoča testno metodo črne škatle s simulacijo uporabnika neposredno v okolju brskalnika.
- Selenium (spletni naslov: http://seleniumhq.org) – odprtokodno orodje za testiranje spletnih aplikacijskih rešitev po metodi črne škatle. V primerjavi z orodjem WatiX se razlikuje po načinu testiranja aplikacijske rešitve. Selenium omogoča snemanje izvedbe uporabniškega procesa in kasnejše predvajanje, zato ga z večjo enostavnostjo lahko uporabljajo tudi testerji, ki nimajo nikakršnega razvojnega predznanja. Podpira naknadno prilagajanje in spreminjanje posnetih testnih skript.
- TestComplete (na spletu: http://www.automatedqa.com/products/testcomplete) – komercialno orodje namenjeno testiranju različnih tipov aplikacijskih rešitev. Nudi skriptiranje testov v različnih jezikih kot tudi veliko možnosti glede snemanja in predvajanja uporabnikovega obnašanja.
- HP Unified Functional Testing (staro ime QuickTest Professional) – orodje, ki je del večjega komercialnega produkta za celovito upravljanje kvalitete aplikacijskih rešitev HP Quality Center (splet: http://www.google.com/search?q=hp+quality+center). Orodje za testiranje HP Unified Functional Testing omogoča avtomatizirano izvajanje funkcionalnih in regresivnih testiranj.
- Pex and Moles (splet: http://research.microsoft.com/en-us/projects/pex) – brezplačni napredni orodji za testiranje aplikacijskih rešitev, ki sta integrirani v razvojno okolje Microsoft Visual Studio 2010. Pex služi avtomatičnemu generiranju testov enot in njihovemu izvajanju z visoko testno pokritostjo programov. Moles služi izdelovanju različnih tipov simuliranih funkcionalnih enot.
- TypeMock (spletni naslov: http://www.typemock.com) – komericalna in brezplačna orodja namenjena testiranju aplikacijskih rešitev. TypeMock Isolator komercialno orodje je še posebej prilagojeno izdelovanju simuliranih enot, katerih s klasičnimi knjižnicami kot sta Moq in RhinoMocks ne moremo neposredno simulirati.

2.4 Sprotna integracija

Sprotna integracija je avtomatiziran proces, ki sprotno izvaja preverjanje kvalitete programov aplikacijske rešitve in je zato za avtomatizirano testiranje pomemben in zelo priporočljiv postopek. Proces se izvaja na strežniku, zato je za razvojno skupino nemoteč, saj ne obremenjuje njihovih razvojnih postaj. Prav tako je v tesni povezavi z verzioniranjem oziroma centralizirano hrambo programov. Običajne nastavitve procesa sprotne integracije ob vsaki shranitvi spremembe sprožijo prevajanje celotnih programov izbrane aplikacijske rešitve.

Dva poglavitna notranja postopka sprotne integracije, ki neposredno vplivata na prihranek virov in kvalitetnejše testiranje aplikacijskih rešitev:

- V proces sprotne integracije je s primernimi nastavitvami možno vključiti izvajanje avtomatiziranih testov. Posredno to pomeni, da se prevajanje programov na razvojnih računalnikih hitreje izvaja, saj razvijalcem ni potrebno preverjati avtomatiziranih testov ob vsakem prevajanju. Prav tako odpade napaka človeškega faktorja, saj sprotna integracija preprečuje, da bi razvijalci pozabili sprotno preverjati rezultate avtomatiziranih testov.
- Proces sprotne integracije lahko s primernimi nastavitvami ob uspešnem prevajanju programov in izvedbi avtomatiziranih testov novo nastalo aplikacijsko rešitev namesti v stopenjsko testno okolje (angl. *staging environment*). S tem so nove ali popravljene funkcionalnosti takoj dostopne testerjem in to v okolju, ki je karseda natančna replika produkcijskega oziroma končnega okolja. Namestitveni proces je tako popolnoma transparenten in na takšen način tudi ne zahteva razvojnih virov, rezultati ročnega testiranja pa so zaradi podobnosti okolja relevantnejši.

Za izvajanje postopka sprotne integracije obstaja več različnih orodij. Najpogosteje uporabljano orodje sprotne integracije je odprtokodna rešitev CruiseControl. Različica za .net okolje imenovana CruiseControl.Net oziroma na kratko CCNet je dostopna na spletnem naslovu: http://ccnet.thoughtworks.com.

Od ostalih orodij za sprotno integracijo sta pogosto v uporabi tudi brezplačni orodji TeamCity (spletni naslov: http://www.jetbrains.com/teamcity) ter Hudson (spletni naslov: http://hudson-ci.org). Na tem mestu so ta orodja navedena predvsem zaradi prej omenjenih pomembnih prednosti, ki so v zvezi s sistematičnim testiranjem aplikacijskih rešitev.

3 Testnost programov in tehnike izboljševanja

Testnost je lastnost oziroma sposobnost posameznih logičnih funkcionalnih enot in razredov aplikacijskega modela, da njihove notranje procese delovanja preverjamo z avtomatiziranim procesom [21]. Sposobnost lahko dosežemo in izboljšamo s prilagojevanjem programov.

Testnost je neposredno povezana s kvalitetnim aplikacijskim načrtovanjem objektno-razrednega modela. Njegova dva glavna stebra sta [19]:

- Kohezivnost – merilo sorodnosti funkcionalnosti posameznega razreda, da služijo njegovemu osnovnemu namenu. Sorodne funkcionalnosti so kohezivnejše.
- Razdruženost – lastnost razreda, da je neodvisen od podrobnosti implementacije funkcionalnosti ostalih enot zunaj sebe. Podrobni opis v poglavju 3.3 na strani 15.

Testnost in dobro aplikacijsko načrtovanje sta neposredno povezana tudi z vzdrževanjem. Večja kohezivnost in razdruženost razredov v objektno-razrednem modelu je premosorazmerna z lažjim vzdrževanjem in dograjevanjem aplikacijske rešitve.

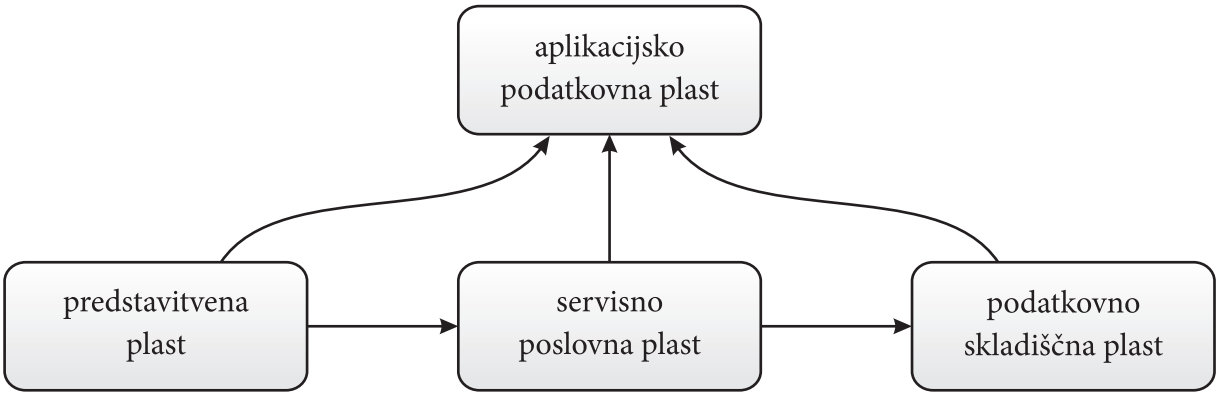
Za izboljšanje testnosti uporabljamo posebne metode in vzorce pisanja programov. Z njihovo pomočjo lahko enostavneje in hitreje uvedemo avtomatizirano testiranje s čimer prihranimo mnogo razvojnih virov.

3.1 Ločevanje logičnih funkcionalnih enot

Konceptualna metoda ločevanja logičnih funkcionalnih enot (angl. *Separation of Concerns* oziroma kratica *SoC*) omogoča boljšo kohezivnost in upravljanje razvoja kompleksnih aplikacijskih rešitev. S to metodo izvajamo dekompozicijo osnovnih kompleksnih problemov v več manjših vendar trivialnejših podproblemov. Ta metoda sloni na predpostavki, da so posamezni podproblemi trivialni in rešljivi. S pravilno dekompozicijo to lahko tudi zagotovimo.

Slika 3 prikazuje zelo pogosto dekompozicijo aplikacijske rešitve v posamezne logične funkcionalne enote. Razvoj posameznih enot je lažji, obseg manjši, njihova medsebojna povezanost pa reši osnovni problem – izvedbo primarnih netrivialnih poslovnih procesov.

Posredno z ločevanjem logičnih enot pridobimo tudi boljšo stopnjo vzdrževanja aplikacijske rešitve, saj se v primeru odpravljanja napak ali dograjevanja novih poslovnih procesov osredotočamo na mnogo manjšo količino programov in funkcionalnosti. Delovanje posameznih delov je zaradi manjšega obsega mnogo razumljivejše, spremembe pa zaradi trivialnih podproblemov predstavljajo manjše tveganje obstoječim že implementiranim funkcionalnostim.



Slika 3. Klasična dekompozicija aplikacijske rešitve v ločene vendar medsebojno povezane logične funkcionalne enote. Smer puščice nakazuje odvisnost.

Dober primer ločevanja logičnih funkcionalnih enot je arhitekturni vzorec MVC (katerega implementira tudi predstavljena skeletna knjižnica Asp.net MVC). Predstavitveno plast aplikacijske rešitve ločuje na tri ločene funkcionalne enote:

- aplikacijsko podatkovni model (na sliki je del zgornje enote),
- uporabniški vmesnik (pogledi) ter
- nadzornike (obe sta del predstavitvene plasti).

Še en dober primer iz razvoja spletnih aplikacijskih rešitev sta jezika HTML in CSS. HTML skrbi za elemente vmesnika torej za njegovo vsebino, CSS pa za izgled in obliko. Dodajanje nove vsebine tako poenostavi njeno oblikovanje. Enako velja tudi v obratni smeri. Sprememba oblikovnih napotkov zapisanih s pomočjo določil CSS se odraža na vseh pripadajočih elementih uporabniškega vmesnika zapisanega v HTML jeziku.

3.2 Princip obrnjenega nadzora

Princip obrnjenega nadzora (angl. *Inversion of control* oziroma kratica *IoC*) je vzorec izvajanja programov pri čemer je potek nadzorne funkcionalnosti obrnjen v primerjavi s sekvenčnim proceduralnim izvajanjem.

Ob podrobnejšem pregledu izvajanja katerekoli aplikacijske rešitve lahko vidimo, da ima vsaka neko centralno-nadzorno funkcionalnost. Ko aplikacijo zaženemo, se običajno prične izvajati prav ta nadzorni del. Kot bomo videli, je izvajanje le-tega lahko različno. Martin Fowler [10] zelo dobro razloži princip obrnjenega nadzora. Za obrazložitev uporablja izvajanje dveh psevdo programov s procesom vnosa in obdelave uporabniških podatkov.

Prvi primer programa ima popoln nadzor nad izvajanjem sekvenčnega procesa. V poenostavljeni kodi jezika C# ga lahko zapišemo na sledeči način:

```
1. void Main()
2. {
3.     WriteLine(“Kako vam je ime?”);
4.     string name = ReadLine();
5.     WriteLine(“Koliko ste stari?”);
6.     string age = ReadLine();
7.     ProcessData(name, age);
8. }
```

Drugi primer programa ima obrnjen nadzor, pri čemer program nima centralno-nadzorne funkcije, zato ne nadzoruje vnosa in obdelave podatkov. Izvajanje programa ni sekvenčno:

```
1. void Main()
2. {
3.     Label lblName = new Label(“Kako vam je ime?”);
4.     Label lblAge = new Label(“Koliko ste stari?”);
5.     TextBox txtName = new TextBox();
6.     TextBox txtAge = new TextBox();
7.     Button btnProcess = new Button {
8.         OnClick += new EventHandler(ProcessData)
9.     };
10.
11.    this.AddControl(lblName);
12.    this.AddControl(txtName);
13.    this.AddControl(lblAge);
14.    this.AddControl(txtAge);
15.    this.AddControl(btnProcess);
16.
17.    this.Run();
18.}
```

Program pripravi funkcionalnosti v obliki dogodkovne funkcije `ProcessData` (za razumevanje programa so njene podrobnosti nepomembne). Izvajanje le-te je odvisno od zunanjega nadzornika, ki jo bo izvajal oziroma klical v skladu z lastnim logičnim procesom.

Med izvajanjem teh dveh programov je torej bistvena razlika. V prvem primeru program kliče sistemske funkcionalnosti za izvajanje svojega procesa. V drugem primeru program le pripravi funkcionalnosti svojega procesa, sistem pa jih nato kliče. Torej: *namesto da ga kličem jaz, naj me raje pokliče on*. Temu pravimo princip obrnjenega nadzora katerega v razvoju aplikacijskih rešitev pogosto srečujemo in uporabljamo.

3.3 Razdruževanje

Od vseh vzorcev izboljševanja testnosti je vzorec razdruževanja (angl. *decoupling*) najpomembnejši in najpogosteje uporabljan. Razdruževanje (praktična uporaba pri obrnjeni odvisnosti [14] angl. *Dependency Inversion* oziroma kratica *DI*) se v razvoju aplikacijskih rešitev uporablja

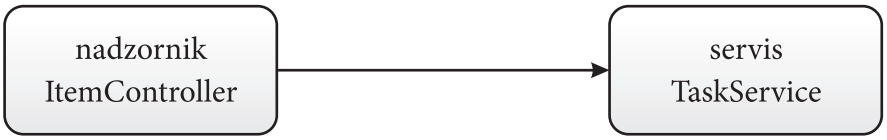
za identifikacijo takšnih enot, ki so navzdol funkcionalno soodvisne. Z razdruževanjem dosežemo, da neposredno niso odvisne druga od druge. Nekatere funkcionalne enote morajo ostati generične, podrobnosti delovanja drugih enot nanje zato ne smejo vplivati.

Uporaba razdruževanja posredno vpliva na zmanjšano tveganje nepravilnega delovanja povezanih enot: kadar eno enoto spremenimo, se z ustrezno razdružitvijo delovanje ostalih enot ne bi smelo odražati v programski napaki. Razdruževanje torej vnaša transparentnost uporabe funkcionalnih enot.

3.3.1 Primer odvisnih enot

Takšen neprimeren način programiranja je prisoten predvsem v nesistematičnem razvoju brez avtomatizacije testov, ki ga najpogosteje srečamo pri manjših internih aplikacijskih rešitvah z nizko verjetnostjo uporabe na širšem trgu. Z razširitvami funkcionalnosti in spremembami programi postanejo težji za vzdrževanje, vsaka dodatna razširitev zaradi notranjih popolnoma odvisnih funkcionalnih enot običajno predstavlja večjo količino razvojnega dela.

Slika 4 prikazuje diagram odvisnosti med dvema funcionalnima enotama. Za lažje razumevanje razdruževanja je uporabljen modificiran primer iz aplikacije. Nadzornik `ItemController` je odvisen od servisa nalog `TaskService`. Smer puščice na sliki nakazuje odvisnosti enot. Servis za upravljanje z nalogami je od nadzornika popolnoma neodvisen. Nadzornik je odvisen od njega, saj je zadolžen za vpisovanje novih nalog, kakor tudi za njihovo branje, zato uporablja njegovo funkcionalnost.



Slika 4. Primer medsebojno odvisnih enot v aplikaciji za upravljanje opomnikov in nalog.

Servis je zaradi neodvisnosti od nadzornika mnogo lažje ponovno uporabiti, saj ne kliče nadzornikovih funkcionalnosti navzgor (uporablja le morebitne funcionalnosti podenot). Največjo težavo pri tem modelu predstavlja nadzornik, ki je izven konteksta dela z nalogami povsem neuporaben. Če bi želeli omogočiti branje opomnikov (ki so nalogam zelo podobni) bi morali spremeniti program nadzornika.

Odvisna oblika programa nadzornika in servisa za branje nalog bi v jeziku C# lahko izgledala takole (zaradi lažjega razumevanja so izpuščeni ostali deli):

```
1. // service implementation
2. public class TaskService
3. {
4.     public IList<Item> GetItems()
5.     {
6.         return TaskRepository.GetItemsFromDataStore();
7.     }
8. }
9.
10.// tightly coupled controller implementation
11.public class ItemController
12.{
13.    public ActionResult GetItems()
14.    {
15.        TaskService service = new TaskService();
16.        IList<Item> items = service.GetItems();
17.        return View(items);
18.    }
19.}
```

3.3.2 Primer razdruženih transparentnih enot

Za izboljšanje testnosti aplikacijskih rešitev moramo funkcionalne enote razdruževati. Nadzornikovo delovanje iz prejšnjega primera je možno razdružiti od detajlov delovanja odvisnega servisa. Pri tem uvedemo abstrakcijo v obliki vmesnika (angl. *interface*), ki določa funkcionalnosti delovanja servisa. Vmesnik predstavlja neke vrste dogovor oziroma protokol po katerem dve razdruženi enoti lahko razumljivo komunicirata. V objektno orientiranih jezikih kakršen je C# lahko to razumemo kot razredno dedovanje, pogosteje pa kot implementacijo vmesnika.

Slika 5. Razredni diagram v objektno orientiranem razdruženem modelu spletne aplikacijske rešitve. Diagram prikazuje pogostejšo vrsto razdruževanja z uporabo vmesnikov.

Nadzornik v zgornjem diagramu ni več neposredno odvisen od podrobnosti delovanja servisa. Za uporabo servisa je med njima vmesnik `IItemService`, ki skrbi za razumljivo komunikacijo med obema razdruženima enotama. Za pravilno delovanje nadzornika je servis popolnoma transparenten. S takšno razdružitvijo lahko funkcionalnosti nadzornika uporabimo tudi za drugačne tipe servisov. Pogoj za to je, da servisi implementirajo vmesnik `IItemService`.

Razdruženi enoti sta medsebojno postali neodvisni do te mere, da notranje spremembe (ali pa zamenjava katere od enot) ne vpliva na delovanje druge razdružene enote. Vzdrževanje in nadgrajevanje takšnih enot je močno olajšano.

```
1. // interface definition
2. public interface IItemService
3. {
4.     IList<Item> GetItems();
5. }
6.
7. // service implementation
8. public class TaskService : IItemService
9. {
10.    public IList<Item> GetItems()
11.    {
12.        return TaskRepository.GetItemsFromDataStore();
13.    }
14.}
15.
16.// decoupled controller implementation
17.public class ItemController
18.{
19.    // service property of interface type
20.    public IItemService Service { get; private set; }
21.
22.    // constructor
23.    public ItemController(IItemService service)
24.    {
25.        this.Service = service;
26.    }
27.
28.    // get items action
29.    public ActionResult Items()
30.    {
31.        IList<Item> items = this.Service.GetItems();
32.        return View(items);
33.    }
34.}
```

Navedeni program prikazuje implementacijo razdruževanja enot nadzornika in servisa. Takšen vzorec oziroma tehnika pisanja programov se zelo pogosto uporablja za omogočanje avtomatiziranega testiranja enot (poglavje 5.1 na strani 27).

3.3.3 Kaj storiti, ko razdruževanje ni mogoče

V določenih primerih razdruževanje funkcionalnih enot ni mogoče. Mogoče je, kadar uporabljamo funkcionalne enote, ki implementirajo vmesnike, so abstraktne ali pa imajo virtualne funkcije, katere je z dedovanjem možno nadomestiti z novimi. V nasprotnem primeru je razdruževanje s klasičnimi pristopi nemogoče.

Eden izmed načinov pristopa k temu problemu je pisanje ovojnih (angl. *wrapper class*) razredov, v katere vstavimo delovanje nespremenljivih funkcionalnih enot. Primer bi bila funkcionalna enota, ki znotraj procesa uporablja statični objekt razreda `System.IO.File`. Ta statični

razred ne implementira nobenega vmesnika. Za izvedbo razdruževanja napišemo ovojni razred, ki služi končni implementaciji, za potrebe testov pa lahko vstavimo odvisnost s simuliranim ovojnim razredom. Primer izvedbe in uporabe ovojnega razreda, ki uporablja prej omenjeni razred za delo z datotekami:

```
1. // wrapper interface
2. public interface IFileChecker
3. {
4.     bool FileExists(string fileName);
5. }
6.
7. // wrapper interface implementation class
8. public class FileChecker : IFileChecker
9. {
10.    public bool FileExists(string fileName)
11.    {
12.        return System.IO.File.Exists(fileName);
13.    }
14.}
15.
16.// class that uses wrapper
17.public class FileOperations
18.{
19.    private IFileChecker Checker { get; private set; }
20.
21.    // dependency injection constructor
22.    public FileOperations(IFileChecker checker)
23.    {
24.        this.Checker = checker;
25.    }
26.
27.    // default constructor
28.    public FileOperations()
29.        : this(new FileChecker())
30.    {
31.        // does nothing
32.    }
33.}
```

Za potrebe preprostih nerazdružljivih funkcionalnosti kot na primer uporaba `DateTime.Now()` pisanje ovojnih razredov ni smotrno. V takšnih primerih lahko uporabljamo napredne testne knjižnice kot sta Moles (iz orodja Pex and Moles) ali TypeMock Isolator, ki zmoreta manipulacijo tudi takšnih funkcionalnosti za katere je pisanje ovojnih razredov nesmotrno.

3.4 Vstavljanje odvisnosti

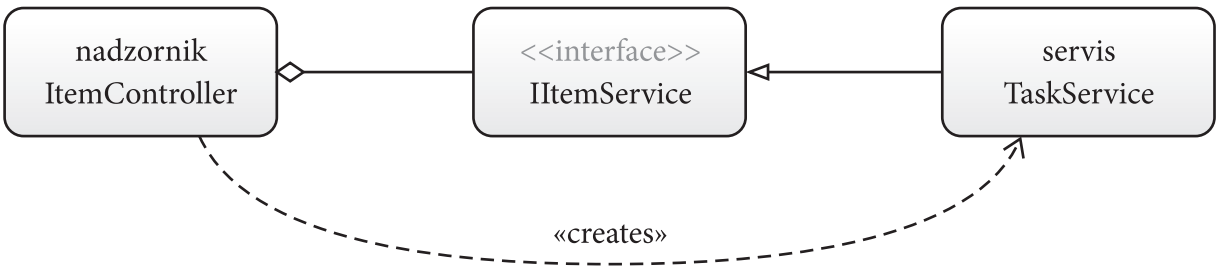
Vstavljanje odvisnosti (angl. *Dependency Injection*) se uporablja v primeru izvajanja funkcionalnosti razdruženih funkcionalnih enot, kot je bilo to ravnokar opisano in prikazano. Ne glede na

razdruženost posameznih funkcionalnih enot in njihovo neodvisnost od podrobnosti delovanja, potrebujemo na nek način poklicati funkcionalnost razdruženo odvisne enote. Nadzornik kljub vsemu potrebuje klic servisa, ki mu bo zagotovil (ali pa shranil) podatke nalog ali opomnikov. V okviru avtomatiziranega testiranja je vstavljanje odvisnosti tesno povezano s simuliranimi funkcionalnimi enotami (poglavje 5.1.1 na strani 28).

3.4.1 Ročno vstavljanje odvisnosti

V primeru enostavnih funkcionalnih enot lahko odvisnosti vstavljamo ročno s pomočjo primernega konstruktorja. V prejšnjem primeru prikazanega programa je imel nadzornik le en konstruktor, kateremu je potrebno podati instanco končnega servisa, katerega bo tekom delovanja uporabljal. Običajno razredom takšnih funkcionalnih enot dodamo še privzeti konstruktor brez parametrov, ki sam določi končni servis, ki naj bi se uporabljal. Predvsem je to pomembno pri testiranju aplikacijskih rešitev, ki so narejene z Asp.net MVC tehnologijo. Knjižnica v ozadju namreč sama instancira nove objekte nadzornikov zato nad parametri konstruktorja nimamo vpliva. Primer spremenjenega nadzornika s privzetim kostruktorjem:

```
1. public class ItemController
2. {
3.     public IItemService Service { get; private set; }
4.
5.     // dependency injection constructor
6.     public ItemController(IItemService service)
7.     {
8.         this.Service = service;
9.     }
10.
11.    // default constructor
12.    public ItemController()
13.        : this(new TaskService())
14.    {
15.        // does nothing
16.    }
17.}
```



Slika 6. Diagram delovanja ročnega vstavljanja odvisnosti s pomočjo konstruktorja.

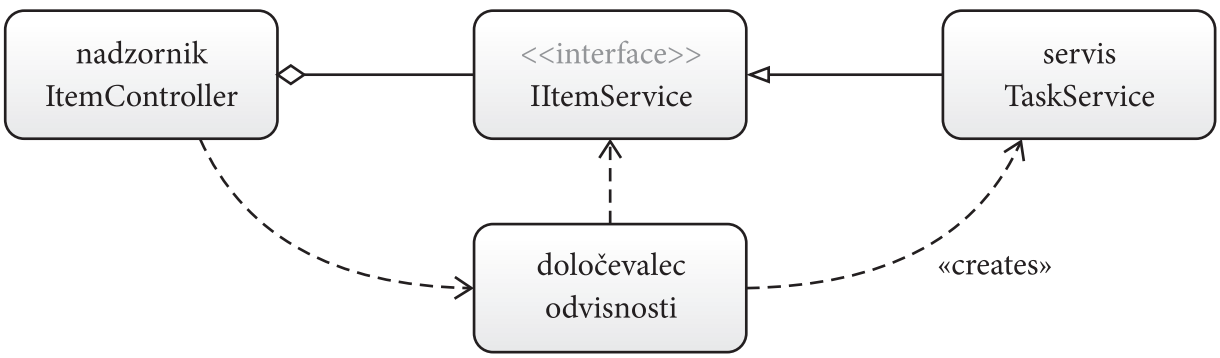
3.4.2 Določevalec odvisnosti funkcionalnih enot

V prejšnjem primeru je bilo prikazano vstavljanje odvisnosti preko konstruktorja razreda, ki za potrebe razredov z nizko stopnjo odvisnosti deluje povsem zadovoljivo (zadovoljivo je tudi s stališča vzdrževanja in porabe razvojnih virov). V kompleksnih aplikacijskih rešitvah so medsebojne odvisnosti med razredi lahko mnogo bolj zapletene in razvejane. Posamezni razred je lahko navzdol posredno odvisen od velikega števila funkcionalnih enot. Primer takšnega razreda je instanciranje hipotetičnega servisa, ki nadzoruje logistiko pošiljanja paketov:

```
1. // instatiate shipping service
2. IShippingService service = new ShippingService(
3.     new ProductLocator(),
4.     new PricingService(),
5.     new InventoryService(),
6.     new TrackingRepository(
7.         new ConfigProvider()
8.     ),
9.     new Logger(
10.        new EmailLogger(
11.            new ConfigProvider()
12.        )
13.    )
14.);
```

V tem primeru je število razdruženih soodvisnih enot veliko, prav tako je prisotno gnezdenje. V primeru avtomatiziranega testiranja kompleksnih rešitev niso takšne soodvisnosti nič neobičajnega, zato v takšnih primerih ne uporabljamo ročnega vstavljanja odvisnosti, pač pa uporabimo določevalca odvisnosti (angl. *Dependency Injection Container* ali *IoC Container*). Primer uporabe takšnega določevalca kaže ekvivalenten program prejšnjemu:

```
1. // instatiate shipping service
2. IShippingService service = IoCContainer.Resolve<IShippingService>();
```



Slika 7. Diagram delovanja vstavljanja odvisnosti z uporabo določevalca odvisnosti.

Določevalec odvisnosti običajno potrebuje zunanje nastavitve, ki povezujejo vmesnike s končnimi razredi, katere mora instancirati. Za produkcijski program navedemo določevalcu določen nabor nastavitev, za potrebe avtomatiziranega testiranja pa drug nabor (običajno so to simulirane funkcionalne enote).

4 Testne metode

Za metodo testiranja se uporablja analogije škatle pri čemer kot takšno vidimo aplikacijsko rešitev. Obstajata dva poglavitna pogleda na škatlo – aplikacijsko rešitev – ter še tretji, ki je kombinacija obeh:

- črna škatla,
- bela škatla ter
- kombinirana siva škatla.

4.1 Črna škatla

Testiranje črne škatle pomeni, da o notranjem delovanju testirane aplikacijske rešitve ne vemo ničesar. Zato preverjamo funkcionalnosti preko vmesnika. S postopkom testiranja črne škatle torej preverjamo, da določeni vhodi v škatlo (vhodni podatki) rezultirajo v pričakovanih izhodih (rezultatih) iz nje. Ker je notranje delovanje oziroma implementacija aplikacijske rešitve pri tej metodi neznana, v tem primeru zato preverjamo skladnost z začetno specifikacijo rešitve (regularnost poslovnih procesov). Torej preverjamo delovanje na višjem funkcionalnem nivoju pri čemer podrobnosti delovanja niso znane.

Primer testiranja črne škatle je t.i. ad-hoc testiranje, kjer tester preizkuša delovanje posameznih delov brez testnega načrta ter preverja njihove izhode. Slabost ad-hoc testiranja je predvsem zelo nizka (ali nikakršna) sledljivost in merljivost uspešnosti testov. Prav tako je nemogoče ugotavljati kvaliteto testiranja – koliko funkcionalnosti je test pokril (testna pokritost), ali je preverjanje izvajano za pravilne in nepravilne vhode ipd.

Testiranje črne škatle je kljub vsemu možno izvajati sistematično. V ta namen je na voljo več različnih testnih pristopov oziroma strategij (naštete so le najpogostejše):

- Raziskovalno testiranje (angl. *exploratory testing*) – testiranje pri katerem je aplikacijska rešitev popolna črna škatla, kjer o njej ne vemo popolnoma ničesar. Testerju ni na voljo niti specifikacija z začetnimi pogoji, ki jih mora končna rešitev izpolnjevati. Metoda raziskovalnega testiranja je podrobneje opisana v poglavju 4.1.1 na strani 24.
- Dvostopenjsko testiranje s testiranjem za uspeh (angl. *test to pass*) in nato testiranjem za neuspeh (angl. *test to fail*) – dvostopenjska metoda pri kateri najprej površinsko testiramo delovanje z regularnimi vhodi, torej v grobem pregledujemo po pričakovanjih regularno delovanje posameznih delov. Druga faza je testiranje za neuspeh, ko podrobno testiramo vsako funkcionalnost z napačnimi vhodi in preverjamo pokritost in regularnost delovanja v takšnih primerih.
- Ekvivalentna porazdelitev (angl. *equivalence partitioning*) – vhodne podatke lahko porazdelimo v logična ekvivalentna območja, pri čemer predpostavljamo, da je test tipične vrednosti posameznega območja enak za katerokoli vrednost v tem območju. Območja,

ki bi jih lahko določili za funkcionalnost zmanjševanje vhodne vrednosti za 1 (angl. *decrement*) bi bila $1 \ldots 2^{16}$, $(2^{16}+1) \ldots 2^{32}$ itd. V kolikor je rezultat testa posemezne operacije znotraj izbranega območja uspešen, je z veliko verjetnostjo rezultat pravilen tudi za vse ostale vrednosti znotraj istega območja. Takšna strategija omogoča testiranje velikega števila vhodov, kljub temu da je število testov relativno majhno.

- Analiza mejnih vrednosti (angl. *boundary value analysis*) – običajno se programske napake dogajajo pri mejnih vrednostih vhodnih podatkov, zato testiramo delovanje z opazovanjem obnašanja pri teh vrednostih. Primer bi bil vnos izredno dolgega imena v polje za vnos imena in priimka. Analiza mejnih vrednosti je povezana z ekvivalentno porazdelitvijo, saj z njo lahko določimo primerne pričakovane mejne vrednosti.
- Diagram prehajanja stanj – vsak poslovni proces lahko vizualno opišemo s pomočjo diagrama stanj. Vsak prehod je odvisen od določenega pogoja (npr. pritisk tipke, vnos določenega podatka, določen čas ipd.), zato lahko pravilne prehode teh stanj preverjamo s primernim vstavljanjem pogojev, ki takšne prehode sprožajo. Ta strategija pomaga testirati pravilnost implementacije poslovnih procesov in prehodov notranjih stanj.

4.1.1 Raziskovalno testiranje

Takšno testno strategijo bi lahko imenovali tudi »črna škatla brez navodil za uporabo«. Uporabljamo jo v primerih, ko tester nima specifikacije oziroma začetnih zahtev aplikacijske rešitve. Torej ne pozna niti implementiranih poslovnih procesov. V tem primeru je sistematičnost težje uvesti, vendar je z metodo raziskovalnega testiranja tudi to mogoče.

Z raziskovalnim testiranjem preverjamo aplikacijsko rešitev kot specifikacijo samo. To metodo bi lahko opisali kot naraščajoče sprotno testiranje. S pregledovanjem posameznih funkcionalnosti aplikacijske rešitve si tester beleži funkcionalnosti in s tem posredno izdeluje specifikacijo. Preverja možne vhode in pregleduje izhode. Za boljšo kvaliteto takšnega testiranja je primeren bolj izkušen tester, ki bo s temeljitostjo zagotovil primerno testno pokritost.

Takšno testiranje sicer ne more biti tako popolno kot s specifikacijo, saj tester ne more vedeti, katere funkcionalnosti oziroma poslovni procesi manjkajo. Testiranje je osredotočeno le na implementirane funkcionalnosti uporabniškega vmesnika in poslovne procese, katere preko vmesnika lahko uporabljamo. Vendar takšen pristop kljub vsemu rezultira v sledljivem in merljivem testiranju.

4.2 Bela škatla

Od metode testiranja črne škatle se ta razlikuje v tem, da so testerju znani in tudi dostopni notranji algoritmi in delovanje aplikacijske rešitve. Rezultat tega védenja so podrobnejši testni načrti, ki testirajo regularnost internega delovanja programov. Večinoma v testih po metodi

bele škatle tester potrebuje tudi razvojno znanje, zato te teste lahko opravljajo le razvijalci. Pri teh testih se uporabljajo tehnike izboljšanja testnosti, kot so bile prikazane v poglavju 3.

Pri testiranje bele škatle razvijalec napiše testne programe, ki enote testirajo pravilnost na več različnih načinov in sicer:

- za določene vhode testira pravilno delovanje razvejitev znotraj programa,
- izvajanje zank,
- klice oziroma uporabo nižjih funkcionalnosti,
- upravljanje s pričakovanimi napakami ipd.

Pri tem se najpogosteje uporabljajo sledeče strategije in testiranja:

- testiranje pravilne uporabe povezanih programskih vmesnikov API;
- programska pokritost – razvijalec napiše teste, ki pokrivajo vse poti znotraj posamezne funkcije in sicer na način, da je vsaka vrstica programa vsaj enkrat klicana oziroma izvajana;
- vstavljanje napak (angl. *fault injection*) – testiranje tistih poti znotraj funkcij, ki upravljajo z napakami, ki se sicer tekom normalnega delovanja ne bi zgodile, torej testiramo skrajnosti delovanja robustnih funkcij;
- statično testiranje programske kode – takšno testiranje najlažje in najučinkoviteje izvajamo z orodji, ki pregledujejo različne aspekte izvorne kode od kompleksnosti in učinkovitosti posameznih funkcij do konsistentnost oblike, kar močno olajša dograjevanje in vzdrževanje programov.

Tesno v povezavi s testiranjem bele škatle je faktor testne pokritosti programov (angl. *code coverage*), ki pove koliko funkcionalnosti aplikacijske rešitve je pokrite s testi. Faktor je pomemben predvsem v avtomatiziranih testih, kjer samo testiranje ni nadzorovano s strani testerja, ampak odvisno od razvijalecev testov. Cilj je seveda 100% testna pokritost, vendar se v praksi izkaže, da takšna pokritost ni vedno tudi smotrna. Kljub vsemu stremimo k čimvečji pokritosti. Aplikacijska rešitev z majhnim številom 100% uspešnih avtomatiziranih testov je lahko neprimerno slabše kvalitete od druge rešitve, ki ima večje število testov, ki pa niso v celoti uspešni. Torej je faktor med pokritostjo in uspešnostjo testov tisti, ki vodi do stabilnih in pravilno delujočih aplikacijskih rešitev.

4.3 Siva škatla

Metoda testiranja sive škatle je kombinacije prejšnjih metod črne in bele škatle. Testiranje aplikacijske rešitve poteka na nivoju uporabnika oziroma uporabniškega vmesnika (enako kot v primeru črne škatle), testni procesi pa so prilagojeni védenju notranjega delovanja notranjih funkcionalnosti (bela škatla).

Ustrezen primer testiranja sive škatle bi bilo testiranje delovanja posameznega procesa uporabniškega vmesnika s povezavo na podatkovno bazo, kjer tester posega neposredno v podatke

podatkovne baze. V tabele dodaja, popravlja ali briše podatke, s katerimi lahko ustvarja pogoje testiranja posameznih razvejitev poslovnega procesa aplikacijske rešitve. Pri takšnem testiranju moramo zelo dobro poznati in upoštevati sorodne oziroma kohezivne funkcionalnosti. Določenemu zapisu v podatkovni bazi na primer ne smemo prirediti vrednosti, katerih ni mogoče določiti preko aplikacijskega vmesnika ali notranjega delovanja poslovnega procesa, ker v nasprotnem primeru delujemo z neregularnimi podatki, katerih proces v ozadju ni sposoben obdelati oziroma uporabiti.

Metodo sive škatle pogosto uporabljajo razvijalci v procesu razhroščevanja aplikacijskih rešitev. Pri tem neposredno manipulirajo s podatki v podatkovni bazi ali pa neposredno v spremenljivkah znotraj funkcionalnosti s pomočjo razhroščevalnih orodij ipd.

5 Testni nivoji

Testni nivoji povezujejo različna testiranja, ki jih vključujemo v različnih fazah razvojnega procesa. Nivoje lahko opredelimo tudi po obsežnosti dosega testiranja v okviru posameznega testa: od najožjega (najplitvejšega) do najširšega (najglobljega).

5.1 Testiranje enot

Testiranje enot (angl. *unit testing*) je prvi nivo programskega testiranja bele škatle, pri čemer testiramo pravilno delovanje najmanjše programske funkcionalne enote. Enota je najmanjši del funkcionalnosti in v objektno orientiranem modelu običajno predstavlja posamezno funkcijo razreda. Testiranje enot je primerjava realnih rezultatov klica funkcije s predvidenimi glede na dane vhodne podatke. Predvideni rezultati so določeni z zahtevami procesa delovanja posamezne enote.

Za namene testiranja je potrebno enoto povsem ločiti od podrobnosti delovanja ostalih povezanih funkcionalnih enot. Za dosego tega pogoja je izvajanje razdruževanja najlažji, naučinkovitejši in zato tudi najpogostejši način (razdruževanje je bilo razloženo v poglavju 3.3 na strani 15). V posameznih testih enot je zato potrebno vstaviti odvisnost razdružene funkcionalne enote, ki implementira točno določeno funkcionalnost (vmesnik), vendar pri tem ne uporabljamo pravih enot, saj bi potem testirali skupek večih enot hkrati in ne le delovanja posamezne enote. V ta namen uporabljamo simulirane funkcionalne enote, ki se obnašajo kot prava produkcijska enota (simulirane enote so razložene v poglavju 5.1.1 na strani 28).

Testiranje enot je osnovno načelo testno vodenega razvoja (angl. *Test driven development* ali kratica *TDD*). Proces razvoja vsake posamezne funkcionalnosti se prične z razvojem testov in šele nato implementacije končne funkcionalnosti enote.

Tekom razvoja aplikacijske rešitve lahko spoznamo, da mora biti delovanje določene funkcionalne enote nekoliko drugačno od do sedaj predvidenega. Temu primerno je potrebno prilagoditi tudi testiranje te enote. Nekateri razvijalci namreč menijo, da se testi enot tekom razvoja ne smejo spremeniti, vendar temu ni tako. Če upoštevamo dejstvo, da je edina nespremenljiva stvar v razvoju aplikacijskih rešitev ta, da se bodo zahteve in rešitev tekom razvoja spreminjale, potem se lažje zavedamo tudi dejstva, da se bodo zaradi tega posledično spreminjala tudi testiranja enot.

Pisanje testov enote zagotavlja tudi pravilno uporabo razdruženih enot. Zaradi tega je potrebno biti pri spremembah testov še posebej previden. V kolikor se notranji proces enote spremeni in prodobi dodatna notranja stanja (kar pomeni, da so obstoječi testi enote uspešni), potem je potrebno napisati le še dodatne teste, ki zagotavljajo uspešno testiranje novih prehodov in dodatnih stanj. Kadar pa se proces temeljito spremeni, je potrebno spreminjati tudi obstoječe teste enote, ki zadoščajo novim izhodom oziroma rezultatom testirane enote.

Vsak test enote je običajno sestavljen iz treh delov (poznani po kratici AAA):

- urejanje (angl. *arrange*) – v tem delu pripravimo instance simuliranih razdruženih enot, katere v izvajalnem delu vstavimo v konstruktor testirane funkcionalne enote;
- izvajanje (angl. *act*) – vstavljanje simuliranih odvisnih enot in izvajanje testa s klicem testirane funkcije;
- potrjevanje (angl. *assert*) – preverjanje pravilnosti izvedenega klica, torej primerjava pravih rezultatov s pričakovanimi in preverjanje klicanja posameznih funkcionalnosti simuliranih enot.

5.1.1 Simuliranje funkcionalnih enot

Pri testiranju posamezne enote se pojavi težava vstavljanja odvisnih razdruženih enot. Ker testiramo samo eno enoto, vstavljanje razdruženih odvisnih enot, ki so del končnega programa, ni pravilno. Potem testiranje ne bi obsegalo le ene same enote, ampak tudi vseh razdruženih enot zraven in njihovega delovanja.

Zato v testiranju enot uporabljamo simulirane funkcionalne enote (angl. *mock*), ki poleg zagotavljanja funkcionalnosti omogočajo tudi preverjanje izvajanja posameznih klicev.

Simulirane enote lahko pripravimo ročno s pisanjem razredov, ki implementirajo povezovalne vmesnike. Dodamo jim le potrebno funkcionalnost, ki bi v določenem testu primerno simulirala realno enoto. Primer programa, za katerega moramo napisati test:

```
1. public class ItemController
2. {
3.     private IItemService Service { get; private set; }
4.
5.     public ItemController(IItemService service)
6.     {
7.         this.Service = service;
8.     }
9.
10.    public ItemController()
11.        : this(new TaskService())
12.    {
13.    }
14.
15.    // this method needs a unit test
16.    public ActionResult Items()
17.    {
18.        IList<Item> items = this.Service.GetItems();
19.        return View(items);
20.    }
21.}
```

Ker testiranje enot testira najmanjše funkcionalne enote, je zato potrebno napisati teste funkcije `Items()`, ki predstavlja takšno enoto (vrstica 15–20). Test mora preveriti, da funkcija kot rezultat klica vrne objekt razrednega tipa `ViewResult`.

Pisanje ročnih simuliranih enot največkrat ni smotrno zato ponavadi uporabljamo knjižnice, s pomočjo katerih lahko hitro ustvarimo primerno simulirano enoto. Sledeči testni program enote uporablja funkcionalnosti knjižnice za avtomatizirano testiranje NUnit in knjižnico za izdelavo simuliranih funkcionalnih enot Moq.

```
1. [TextFixture]
2. public class ItemControllerTest
3. {
4.     [Test]
5.     // unit test using Moq library
6.     public void ItemsTest_ReturnsViewResultType()
7.     {
8.         // arrange
9.         // prepare service mock
10.        IItemService mock = new Mock<IItemService>();
11.        mock.Setup<IList<Item>>(svc => svc.GetItems())
12.            .Returns(() => new List<Item>());
13.
14.        // act
15.        // inject mock service object
16.        ItemController controller = new ItemController(mock.Object);
17.        ActionResult result = controller.Items();
18.
19.        // assert
20.        Assert.IsInstanceOf<ViewResult>(result);
21.    }
22.}
```

5.1.2 Slabosti testiranja enot

Testiranje enot ni popolnoma trivialen proces. Pri pisanju testov so razvijalci soočeni s sledečimi izzivi:

- Kvaliteta testov enot – dobra je lahko le toliko, kot je dober razvijalec pri predvidevanju možnih izhodnih rezultatov glede na dane vhodne podatke ter kolikor truda je za pisanje le-teh razvijalec pripravljen vložiti.
- Količina testov enot – povečevanje števila testov ni povezano s kvaliteto aplikacijske rešitve. Več testov ne pomeni nujno tudi večje pokritosti izvajanih programov.
- Novi programi in stari testi – medtem ko so testi enot dodajani in popravljani pred in med razvojem novih funkcionalnosti, se le-ti redkeje spreminjajo kasneje, ko je funkcionalnost v delujočem stanju. Če kasneje razvijalec doda nova notranja stanja testne enote in ne napiše dodatnih testov, novi programi s starimi testi ne bodo pokriti.

- Skriti integracijski testi – testiranje enot bi moralo potekati v izoliranem okolju z uporabo simuliranih funkcionalnih enot, vendar temu ni vedno tako, zato lahko testiranje enot znotraj uporablja detajle pravih razdruženih funkcionalnih enot.

5.1.3 Parametrizirano testiranje enot

V izogib slabe kvalitete testov enot (prva alineja prej opisanih slabosti) obstaja poseben pristop pisanja parametriziranih testov enot z uporabo orodij kot je Pex. Pri pisanju parametriziranih testov enote namesto testne funkcije brez parametrov napišemo takšno, ki sprejme tudi parametre vhodnih podatkov testirane enote. Orodje Pex ob izvajanju inteligentno izbira takšne vrednosti, ki so za proces delovanja enote odločilni. Pri tem upošteva pogojne razvejitve v programih in s tem doseže zelo dobro testno pokritost programov, kar je eden izmed ciljev testiranja enot. S tem se močno zmanjša slabost prve alineje, ki govori o slabi kvaliteti testov enot.

```
1. [PexMethod]
2. public void DummyTest(int value)
3. {
4.     // assume
5.     PexAssume.IsTrue(value > 0);
6.
7.     // arrange
8.     // no mocks here in this simple test
9.
10.    // act
11.    InvalidCalculator calc = new InvalidCalculator();
12.    int result = calc.ThrowExceptionWhen123(value);
13.
14.    // assert
15.    PexAssert.AreEqual(result, value);
16.}
```

Prikazana enostavna koda uporablja hipotetični objekt razreda `InvalidCalculator`, ki vrne napako, kadar mu posredujemo vrednost 123. Pex bo ob izvajanju tega parametriziranega testa prikazal vrednost 123, pri kateri je test neuspešen. Dobra lastnost orodja je ta, da vhodnih vrednosti ne izbira naključno, zato so takšni testi lahko zares relevantni.

5.2 Integracijsko testiranje

V okviru testiranja enot so bile posamezne enote programov testirane v izoliranem okolju. Vsi klici v razdružene funkcionalne enote so potekali preko simuliranih funkcionalnih enot. Prav tako v okviru testiranja enot ni mogoče (ali pa izredno težko, zamudno in nesmotrno)

testirati enot, ki so neposredno povezane na zunanje vire (podatkovna baza, datotečni sistemi ipd.). Testiranje teh enot je zato vključeno v nivo integracijskega testiranja.

Integracijsko testiranje (angl. *integration testing*) je programsko testiranje bele škatle in odstranjuje simulirane enote iz programov testov enot, zato testira delovanje enot v integriranem načinu. Pri tem testira vse nivoje, razen tistih, ki so vezani na zunanje sisteme, katerih delovanja ni mogoče nadzorovati (zunanji spletni servisi, plačljive storitve ipd.).

Za izvajanje integracijskega testiranja obstaja več pristopov:

- od zgoraj navzol – postopna integracija enot od zgoraj proti dnu;
- od spodaj navzgor –postopna integracija enot od spodaj proti vrhu;
- kombinacija prejšnjih dveh prostopov imenovana tudi sendvič pristop;
- pristop velikega poka (angl. *big bang*) – celotna integracija naenkrat.

Tudi pri teh testih želimo ostati v mejah smotrnosti in učinkovitosti, zato pisanje posebnih testov v celoti običajno ni priporočljivo. Najlažji sta tako izvedbi od spodaj navzgor ali pa pristop velikega poka.

Pri pristopu velikega poka uporabimo obstoječe teste enote in iz njih odstranimo vse simulirane funkcionalne enote. Nadomestimo jih s končnimi produkcijskimi enotami. Takšne teste nato lahko avtomatizirano izvajamo na enak način kot avtomatizirana testiranja enot. Pri določenih testih je potrebno zagotoviti začetne pogoje (na primer primerne in količinsko zadostne podatke v bazi podatkov), kar lahko storimo v kodi, ali pa z zunanjo skripto, katero izvedemo pred testiranjem. Zanesljivejši način je znotraj kode, saj moramo sicer zagotoviti pravilni redosled izvajanja integracijskih testov, kar pa običajno zaradi popravkov prinaša slabe rezultate in pogosta nepravilna testiranja (ter posredno potratnost virov).

Drugi možni način je s pristopom od spodaj navzgor. Pri tem pristopu pričnemo s testiranjem najnižjih funkcionalnih enot, ki uporabljajo zunanje vire in na nivoju testiranja enot niso bile vključene v testiranje. Ko se testiranje najnižjih enot uspešno zaključi dodamo v integracijske teste enote nad njimi ter ponovimo postopek. To nadaljujemo toliko časa, dokler ne pridemo do vrha oziroma do tistih funkcionalnih enot, ki so izvor vseh funkcionalnosti. Pri razvoju Asp.net MVC aplikacijskih rešitev so to razredi in funkcionalnosti nadzornikov.

Izvajanje integracijskih testov od zgoraj navzdol se redkokdaj uporablja, saj zahteva več dela v smislu pisanja testnih programov. V urejevalnem delu testa moramo uporabljati tako prave kot tudi simulirane enote. Pri vsaki iteraciji (pomikanje navzdol) moramo napisati vedno večje in večje število testov, zato je ta pristop redek in se običajno zaradi potratnosti virov ne uporablja. Izvajanje je priporočeno in smotrno pri zelo tveganih projektih, saj s tem pristopom odkrijemo tiste veje programov, ki nimajo pokritih koncev in jih zato s pristopom od spodaj navzgor ne moremo pokriti.

Pri sendvič pristopu lahko izvajamo do določenega nivoja eno testiranje, potem pa obrnemo in pričnemo z drugim, katerega izvedemo do istega nivoja. Ta pristop uporabimo, kadar namesto popolnega testiranja od spodaj navzgor temeljito testiramo le skrajni spodnji del in nivo nad njim (da je izvedena integracija med njima), potem pa pričnemo s pristopom od zgoraj in integriramo do predzadnjega nivoja.

5.3 Regresivno testiranje

Regresivno testiranje (angl. *regression testing*) je lahko kombinacija testiranja črne škatle (preko uporabniškega vmesnika) in bele škatle (neposredno testiranje programov). Regresivno testiranje se uporablja za odkrivanje že odpravljenih programskih napak, ki so se ponovno pojavile. Izkušnje kažejo, da je ob dokončno implementiranih funkcionalnostih aplikacijske rešitve pojavljanje novih in starih že odstranjenih programskih napak dokaj pogost pojav [23]. Zaradi tega se ob pojavu nove programske napake shrani test (najbolje avtomatiziran), ki takšno napako dokazuje. Ko je napaka odpravljena, je testiranje uspešno, vendar ga ob vsaki namestitvi v testno okolje ponovno preverjamo, če se z novimi popravki v programih stara napaka slučajno ni ponovno pojavila.

Pogostost izvajanja regresivnega testiranja je odvisna od posamezne razvojne skupine. V primeru avtomatizacije se običajno izvaja enkrat dnevno skupaj z dnevnim prevajanjem. Pri ročnem pristopu se te teste izvaja redkeje, saj je proces časovno mnogo potratnejši.

Iz zapisanega sledi, da je nivo regresivnega testiranja zelo pomemben v obdobju vzdrževanja aplikacijske rešitve. Že odpravljene programske napake se običajno ponovno pojavijo med refaktoriranjem delujočih funkcionalnih enot, saj pri tem razvijalci pozabijo implementirati procesne podrobnosti. Testi regresivnega testiranja opozarjajo prav na tiste posebnosti poslovnih procesov funkcionalne enote, ki so na prvi pogled neočitne.

5.4 Dimno testiranje

Dimno testiranje (angl. *smoke testing*) je posebna vrsta testiranja črne škatle (ne testiramo več programov neposredno) pri kateri preverjamo površinsko delovanje najpomembnejših procesov integrirane aplikacijske rešitve. S hitrim pregledom osnovnih funkcionalnosti delovanja rešitve ugotavljamo, če je podrobno testiranje sploh smiselno.

Dimno testiranje se običajno izvaja po vsakem glavnem dnevnem prevajanju programov, ko se rešitev namesti v testno okolje. V kolikor dimni test pokaže ključne težave jih razvijalci z najvišjo prioriteto odpravijo, ter ponovijo postopek dnevnega prevajanja z namestitvijo.

Postopek dimnega testiranja se najpogosteje izvaja ročno s strani enega člana razvojne skupine. Postopek je mogoče tudi avtomatizirati. Pri tem se uporablja orodja, ki omogočajo snemanje uporabniškega procesa in kasnejše ponovno predvajanje (na primer orodje Selenium). Avtomatizacija je smotrna pri obsežnejši aplikacijski rešitvi ali pri splošno nestabilni rešitvi, kjer je neuspešnost dimnega testiranja pogostejša.

5.5 Sistemsko testiranje

Sistemsko testiranje preverja skladnost aplikacijske rešitve z njenimi nefunkcionalnimi zahtevami. Specifikacija lahko na primer opredeljuje določene procesno-hitrostne zahteve, ki se v okviru teh testiranj preverijo. V okvir sistemskega testiranja spadajo [20]:

- testiranje posebnih zahtev uporabniškega vmesnika – uporaba asinhrone komunikacije, uporabnost aplikacije za uporabnike z določeno telesno hibo ipd.;
- uporabnostno testiranje (angl. *usability testing*) – običajno se to testiranje izvaja neposredno s končnimi uporabniki;
- testiranje povezljivosti (angl. *compatibility testing*) – to testiranje je še posebej prisotno pri spletnih aplikacijah saj se testira povezljivost in pravilno delovanje v različnih brskalnikih;
- testiranje upravljanja z napakami – beleženje napak, primerno obveščanje uporabnika, uporaba transakcij na nivoju podatkovne baze ipd.;
- stresno/obremenilno testiranje – testiranje visoke obremenljivosti, delovanje v okolju z velikimi količinami podatkov;
- varnostno testiranje – preverjanje zagotavljanja primerne zaščite;
- testiranje razširljivosti (angl. *scalability testing*) – testiranje sposobnosti prilagoditve aplikacijske rešitve na povečan obseg dela oziroma procesiranja brez posegov v programe;
- namestitveno testiranje (angl. *deployment testing*) – testiranje namestitvenega postopka v različnih okoljih z različnimi pogoji;
- ipd.

Nekatera od omenjenih testiranj je zelo priporočljivo avtomatizirati, saj so le tako lahko učinkovita in merljiva. Drugih ne moremo izvajati drugače kot ročno in sicer jih lahko izvaja le oseba, ki ima primerna znanja in veščine za preverjanje regularnosti.

5.6 Sprejemljivostno testiranje

Sprejemljivostno testiranje (angl. *acceptance testing*) je metoda črne škatle, saj ne testiramo rešitve programsko, ampak iz uporabniškega vidika. Nivo sprejemljivostnega testiranja se izvaja na koncu razvoja aplikacijske rešitve. V iterativni razvojni metodologiji se del teh testiranj lahko izvaja po vsaki iteraciji in sicer za tiste funkcionalnosti, ki so bile tekom iteracije implementirane. Lahko je del sistemskega testiranja, ki preverja skladnost rešitve z zahtevami.

Ko je funkcionalnost v celoti implementirana, se testira njeno sprejemljivost v smislu testiranja uporabniških zgodb (angl. *user story*), ki opredeljujejo procese delovanja aplikacijske rešitve. Uporabniške zgodbe so običajno dogovorjene med razvojno skupino in naročniki, saj razvijalci le tako lahko pridobijo primerno poznavanje področja, katerega procese naj aplikacijska rešitev implementira.

Uporabniške zgodbe običajno opredelujejo uporabniške funkcionalno usmerjene procese v grobem delovanju in se ne spuščajo v podrobnosti implementacije, saj le ta že lahko meji na testno metodo bele škatle. Uporabniške zgodbe zato vključujejo različne aktivnosti, ki jih uporabniki preko vmesnika lahko izvajajo, ne pa podrobnosti izvedbe, kako naj se jih izvaja. V primeru spletnih aplikacijskih rešitev se zato ne opredeljuje (razen, kadar je za to posebna zahteva), da mora biti določen proces implementiran s pomočjo asinhronih klicev strežnika. Tehnološke podrobnosti uporabniških zgodb lahko ohromijo njihovo implementacijo, saj po nepotrebnem povečujejo kompleksnost izvedbe.

Sprejemljivostno testiranje je lahko delno avtomatizirano, vendar zopet le z orodji, ki omogočajo snemanje in kasnejše predvajanje uporabniškega procesa v uporabniškem vmesniku.

6 Sklepne ugotovitve

Za uspešno implementacijo sistematičnega testiranja aplikacijskih rešitev je potrebno v okviru razvojne skupine zadostiti sledečim pogojem:

- poznavanje razvojnih vzorcev s katerimi izboljšujemo testnost aplikacijskih rešitev,
- vključevanje testerjev v razvojne skupine, ki imajo testna znanja s pomočjo katerih lahko izvajajo merljivo in sledljivo testiranje,
- upoštevanje razvojne kulture pisanja kvalitetnih in zadostnih testov enot,
- poznavanje orodij in knjižnic s katerim si pri pisanju in izvajanju testov pomagamo in prihranimo čas ter
- uporaba centraliziranega sistema za upravljanje s programskimi napakami.

Da med testiranjem po nepotrebnem razvojne skupine ne tratijo razvojnih virov, je potrebno ves čas preverjati smotrnost pisanja posameznih testov. Pogojno jih ni potrebno pisati za tiste funkcijske enote, ki zadoščajo vsaj trem pogojem:

1. notranji proces enote je trivialen – nima razvejitev;
2. enota nima vhodnih podatkov ali jih ne spreminja;
3. enota nima izhodnih podatkov ali ne manipulira z njimi;
4. enota opravlja le klice v nižje enote.

Pri takšnih enotah običajno pisanje testov zahteva mnogo več časa kot pa končna implementacija enote. Zaradi trivialnosti delovanja je površina nastanka programskih napak skrajno majhna. V tem primeru je smotrnost pisanja testov odvisna predvsem od ostalih dejavnikov tveganja (npr. pogostost in količina spreminjanja aplikacijske rešitve tekom razvoja).

Prav tako bi poudaril, da za zagotavljanje sistematičnega testiranja aplikacijskih rešitev ni potrebno izvajati celotnega nabora testnih nivojev, kot so opisani v tem diplomskem delu. Obvezna je avtomatizirana implementacija predvsem kvalitetnih testov enot. Zelo priporočeno je tudi izvajanje regresivnih testov. Če je le mogoče v avtomatiziranem načinu.

Predvsem pa je pomembna osveščenost in spremljanje razvoja testnih pristopov, vzorcev in orodij, saj nam le-ta pogosto močno olajšajo testiranje in prihranijo drage razvojne vire.

Dodatek A Terminološki slovar

DESTRUKTOR posebna funkcija razreda, ki se izvede ob njegovem brisanju iz pomnilnika. V njej se običajno sprosti vse povezane enote (npr. dolgotrajno odprte datoteke, odprte transportne poti ipd.).

ENTITETA pomenski razred določenega tipa, ki s svojim imenom posredno razlaga notranje procese, možnosti delovanja in stanja. V razvojni terminologiji se izraz entiteta običajno uporablja v povezavi s podatki v podatkovni bazi.

IMPLEMENTACIJA (1) proces pisanja programov, ki služi izvajanju določenega poslovnega procesa (2) *implementacija vmesnika* zaključen program, ki izpolnjuje funkcionalne zahteve razrednega vmesnika.

INSTANCA objekt določenega tipa razreda v pomnilniku, ki ima stanje, njegove funkcionalnosti so dostopne klicateljem.

INSTANCIRANJE proces ustvarjanja novega objekta točno določenega tipa razreda (običajno s klicem →konstruktorja), ko se v pomnilniku vzpostavi njegovo začetno stanje in njegove funkcionalnosti postanejo dostopne klicateljem – v pomnilniku se ustvari →instanca objekta tipa določenega razreda.

KONSTRUKTOR posebna funkcija razreda, ki je klicana ob →instanciranju objekta, torej ob njegovem nastanku. Kadar ima →razred več možnih začetnih stanj, se to običajno odraža v večjem številu različnih konstruktorjev od katerih vsak vzpostavi drugačno začetno stanje objekta.

PARAMETER podatek v obliki instance objekta ali vrednosti točno določenega vrednostnega tipa ali tipa →razreda, ki se posreduje funkciji ob njenem klicu.

RAZRED zaključena celota programa, ki enovito določa stanje, spremembe stanj in druge funkcionalnosti (obnašanje), ki so povezane z njenim notranjim procesom.

RAZŠIRLJIVOST prilagoditvena sposobnost aplikacijske rešitve na večji obseg procesiranja ali količine dostopov. Aplikacijska rešitev je razširljiva, kadar lahko povečan obseg kontroliramo z zunanjimi prilagoditvami (recimo dodajanje procesne moči s porazdeljenimi strežniki) torej brez posegov v same programe aplikacije.

Dodatek B Raziskava

Za potrebe tega diplomskega dela je bila izvedena tudi raziskava uporabe sistematičnega testiranja v profesionalnem okolju razvojnih skupin. Sestavljen je bil vprašalnik namenjen podjetjem in posameznikom, ki se ukvarjajo z razvojem programske opreme. Objavljen je bil na svetovnem spletu, tako da so nanj lahko odgovarjali ljudje s celega sveta.

B.1 Sestava vprašalnika

Vprašanja so bila razdeljena v več medsebojno odvisnih sklopov, tako da so posamezniki glede na odgovore v določenih sklopih odgovarjali naprej le na pripadajoča vprašanja. Na primer, če je nekdo v začetku izbral, da razvijajo v podjetju spletne aplikacijske rešitve, je kasneje odgovarjal na vprašanja povezana s spletnim razvojem ipd.

B.1.1 Vstopni filter

Najprej so bili anketiranci vprašani, če se oni oziroma podjetje v katerem delajo ukvarja z razvojem programske opreme. V kolikor so odgovorili z »ne«, so bili nemudoma preusmerjeni na zaključek vprašalnika, saj njihovi odgovori ne bi bili merodajni. Torej je vstopni filter segregiral primerne anketirance.

B.1.2 Prvi sklop – informacije o posamezniku in podjetju

1. Država v kateri živi in dela anketiranec: prosto vpisno polje v katerega so anketiranci lahko vnesli poljubno državo.
2. Okvirna starost (v obliki starostnih območij): manj kot 20 let, 20–30, 30–40, 40-50 ali več kot 50 let.
3. Posplošeno delovno mesto: razvijalec, tester, vodja razvoja, vodja splošno ali drugo, kamor so anketiranci lahko vpisali poljubno delovno mesto.
4. Koliko časa se profesionalno ukvarjate z razvojem (vi ali vaše podjetje): manj kot 3 leta, 3–5, 5–10, 10–15 ali več kot 15 let.
5. Kako ocenjujete svoje razvojno znanje: ocena od 1 (osnovno) do 5 (ekspertno).
6. Kako ocenjujete razvojne sposobnosti vašega podjetja: ocena od 1 (osnovno) do 5 (profesionalno).
7. Velikost vaše razvojne skupine: 1, 2–4, 5–7, 8–10, 11–20, 21–50 ali več kot 50.

8. Velikost podjetja v številu zaposlenih: manj kot 5, 5–20, 20–50, 50–100, 100–500, 500–2000 ali več kot 2000.
9. Za kateri sektor razvijate programsko opremo: Finančni sektor in zavarovalništvo, it in komunikacije, sektor dela s končnimi uporabniki (klicni centri, servisi ipd.), inženiring, vodstveni sektor, zdravstvo in farmacija, energetika, pravni sektor, javni sektor, prodaja, oglaševanje in mediji ter drugo, kamor so lahko vpisali dodatne sektorje.
10. Katero vrsto aplikacij razvijate največ: spletne aplikacije, namizne aplikacije, modilne rešitve, vgradne rešitve (angl. *embedded*).
11. Katere baze podatkov uporabljate: Microsoft SQL Server, MySql, PostgreSql, SQLite, CouchDB, VistaDB, Oracle, db2, Google Gears in drugo, kamor so anketiranci lahko vpisali karkoli drugega.

Na podlagi 10. odgovora so bili anketiranci preusmerjeni ali na sklop vprašanj razvoja spletnih aplikacijskih rešitev ali pa namiznih/mobilnih/vgradnih rešitev.

B.1.3 Drugi sklop – razvoj spletnih rešitev

1. Za katere spletne tehnologije razvijate aplikacijske rešitve: Asp.net WebForms, Asp.net MVC, PHP, Ruby on Rails, Flash, Silverlight, Java EE, ColdFusion in drugo, kamor je bilo možno vpisati karkoli drugega.
2. Za katero od izbranih tehnologij najbolj oziroma najpogosteje: enake možnosti kot pri prvem vprašanju, le da so anketiranci lahko izbrali en sam odgovor.
3. Katere programske jezike uporabljate: C#, VB.net, Javascript, HTML/XHTML, CSS, C++, Java, PHP, Ruby, Perl, Python in drugo, kamor so lahko vpisali poljubne jezike.

B.1.4 Drugi sklop – razvoj namiznih/mobilnih/vgradnih rešitev

1. Za katere tehnologije razvijate namizne/mobilne/vgradne rešitve: Microsoft Windows, Apple OS X, Linux, Adobe Air, Silverlight, Microsoft Windows Mobile, Apple iOS, posebne vgradne in drugo, kamor so lahko vpisali poljubne tehnologije.
2. Za katero od izbranih tehnologij najbolj oziroma najpogosteje: enake možnosti kot pri prvem vprašanju, le da so anketiranci lahko izbrali en sam odgovor.
3. Katere programske jezike uporabljate: C#, VB.net, C++, Java, Python, Delphi, Perl, Assembler in drugo, kamor so lahko vpisali poljubne programske jezike.

B.1.5 Tretji sklop – testiranje aplikacijskih rešitev

1. Ali so testerji, katerih delo obsega samo testiranje, del vaše razvojne skupine: da in izvajajo sistematično testiranje; da, vendar izvajajo le ad-hoc testiranje; ne, nimamo testerjev.
2. Ali imate pri vaših projektih dokument s testnim načrtom: ocena podrobnosti takšnega dokumenta od 1 (nimamo dokumenta) do 5 (zelo podrobno opredeljen dokument).
3. Ali veste, kaj je avtomatiziran test: da, ker jih pišemo ves čas; da, vendar jih pišemo le občasno; da, vendar jih ne pišemo; ne, ne vem.
4. Ali veste kakšna je razlika med integracijskim testom in testom enot: da; zdi se mi da vem, vendar nisem popolnoma prepričan; ne vem.
5. Kakšno je vaše mnenje o testiranju enot (možnaizbira več odgovorov): ne vem kaj je test enot; testi enot so uporabni, vendar jih ne pišemo; pišemo jih, ko naša produkcijska koda že deluje; pišemo jih preden napišemo produkcijsko kodo; moramo jih pisati, vendar menim, da potratijo preveč virov; zahtevajo veliko dodatnega dela, vendar so zelo uporabni; radi bi prepričali vodstvo, ki meni da so predragi; prosto polje, kamor so lahko vpisali poljubno mnenje.
6. Katere vrste testiranj uporabljate v vaši razvojni skupini: razhroščevanje programov, ad-hoc testiranje (brez načrta, sledenja in meril), testiranje enot, integracijsko testiranje, regresijsko testiranje, sprejemljivostno testiranje, dimno testiranje, uporabno testiranje, stresno testiranje, varnostno testiranje ter drugo, kamor so lahko vpisali poljubne vrste testiranj.
7. Katere vrste testov avtomatizirate: nobenih, teste enot, integracijske teste, regresijske teste, sprejemljivostne teste, dimne teste, stresne teste ter drugo, kamor so lahko vpisali poljubne teste.
8. Katera testna orodja in knjižnice pri testiranju uporabljate (pri knjižnicah s črko X pomeni, da je to lahko katerakoli implementacija knjižnice prirejena določenemu okolju ali jeziku): nobenih, xUnit, Microsoft Visual Sudio Testing Framework, xMock, Moq, RhinoMock, Mockito, Pex and Moles, Selenium, WatiX, HP Quick Test Professional, Autilt, TestComplete in drugo kamor so lahko vpisali poljubna testna orodja.
9. Ali testirate tudi skripte podatkovnih baz: da; ne, ker nismo vedeli, da je to sploh možno; ne testiramo.
10. Ali imate kakršnokoli rešitev sledenja programskih napak: da in jo tudi uporabljamo; da, vendar je sploh ne ali pa napačno uporabljamo; ne, takšne rešitve nimamo.

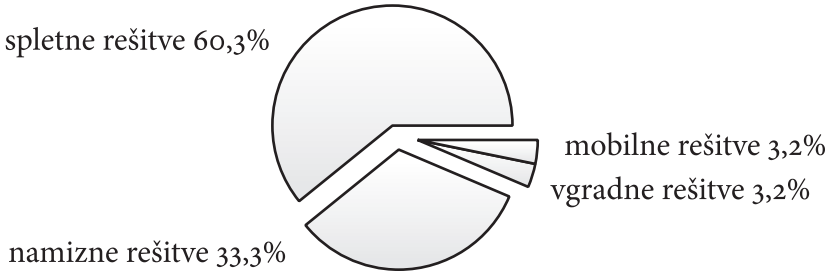
V kolikor so anketiranci pri 10. vprašanju odgovorili s prvim ali drugim odgovorom, so bili preusmerjeni na četrti sklop vprašanj povezanih s sledenjemoziroma upravljanjem programskih napak.

B.1.6 Četrti sklop – sledenje programskih napak

1. Ali popravljate znane programske napake preden nadaljujete z implementacijo novih funkcionalnosti: da, kolikor je to le mogoče; ne nujno, vendar lahko kakšno odpravimo; ne, napake odrpavljamo šele, ko so funkcionalnosti implementirane; drugo, kamor so lahko vpisali poljuben odgovor.
2. Katere rešitve za sledenje programskih napak uporabljate: Microsoft Office Excel razpredelnice, Google Docs razpredelnice, Bugzilla, Trac, Jira, FogBugs, Mantis, Microsoft Team Foundation Server (TFS), BugTracker.net, Redmine, navadno tablo oziroma t.i. kanban tablo ter drugo, kamor so lahko vpisali poljubno rešitev.
3. Katero od izbranih rešitev uporabljate najbolj: enaki možni odgovori, kot pri prejšnjem vprašanju, le da so lahko izbrali le enega.
4. Kako zadovoljni ste z enostavnostjo uporabe vaše rešitve: ocena enostavnosti od 1 (zapletena) do 5 (zelo enostavna).
5. Kako časovno potratna je rešitev, ki jo uporabljate: ocena od 1 (majhna) do 5 (velika).
6. Kakšne so vgrajene možnosti poročanja vaše rešitve: ocena od 1 (brez možnosti poročanja) do 5 (mnogo možnosti poročanja).
7. Kako zadovoljni ste z vgrajenimi možnostmi obveščanja o napakah: ocena od 1 (ni obvestil) do 5 (zelo dobre možnosti obveščanja).
8. Kakšne so možnosti podatkovnih prilagoditev ter prilagoditev uporabniškega vmesnika vašemu razvojnemu procesu: ocena od 1 (majhne) do 5 (zelo prilagodljiva rešitev).
9. Kako primerna se vam zdi rešitev za vašo razvojno skupino: ocena od 1 (neprimerna) do 5 (zelo primerna).
10. Kakšne so vgrajene možnosti procesa sledenja programskih napak: ocena od 1 (ni procesa) do 5 (zelo dobre).
11. Kakšna je količina podatkov, ki jo morate v sistem vnašati: ocena od 1 (premajhna) do 5 (preveč podatkov) pri čemer vrednost 3 pomeni ravno pravšnjo količino podatkov.
12. Kakšna so možnosti dostopnosti do rešitve od koderkoli oziroma s čimerkoli: ocena od 1 (slabe) do 5 (zelo dobre).
13. Kakšne so možnosti dodajanja vtičnikov (angl. *plug-in*) ali dodatkov (angl. *add-in*) v rešitev, ki jo uporabljate: ocena od 1 (ni teh možnosti) do 5 (zelo dobre).
14. Kakšno je vaše splošno zadovoljstvo z rešitvijo sledenja programskih napak: ocena od 1 (slabo) do 5 (zelo zadovoljen).
15. Ali bi karkoli spremenili glede vaše rešitve sledenja programskih napak: prosto polje, kamor so anketiranci lahko vpisali kakršnekoli spremembe.

B.2 Rezultati raziskave

Vprašalnik je skupno izpolnilo 66 oseb. Trije odzivi nanj so neuporabni, saj so jih izpolnjevale osebe, ki se ne ukvarjajo z razvojem programske opreme in tudi ne delajo v podjetju, ki se s tem ukvarja. Torej je skupno 63 uporabnih izpolnjenih vprašalnikov.



Slika 8. Porazdelitev razvoja glede na primarni tip aplikacijskih rešitev.

Porazdelitev anketirancev glede na delovno mesto, ki ga opravljajo:

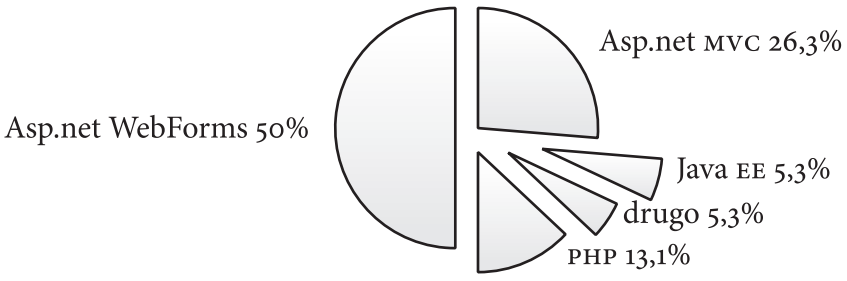
- razvijalec aplikacijskih rešitev: 68,2% (43 oseb);
- tester: 1,6% (1 oseba);
- ostalo (vodstveni kader, konzultant ipd.): 30,2% (19 oseb).

Geografsko so anketiranci iz sledečih držav:

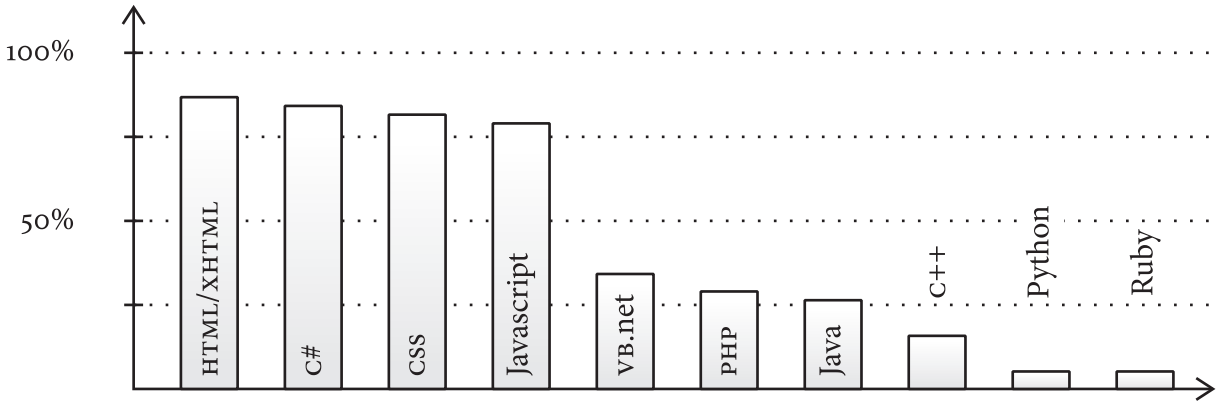
- Slovenija: 46%;
- Združene države Amerike: 23,8%;
- Velika Britanija: 17,5%;
- 1 oseba iz Kanade, Grčije, Danske, Mehike, Nizozemske in Srbije (1,6%);
- 2 osebi nista navedli države.

B.2.1 Spletne aplikacijske rešitve

Od 38 oseb, ki razvijajo spletne aplikacijske rešitve, jih največ razvija v tehnologiji Asp.net (skupno kar 76,3%). Podrobnejšo porazdelitev kaže slika.



Slika 9. Porazdelitev razvoja glede na vrsto spletne tehnologije.

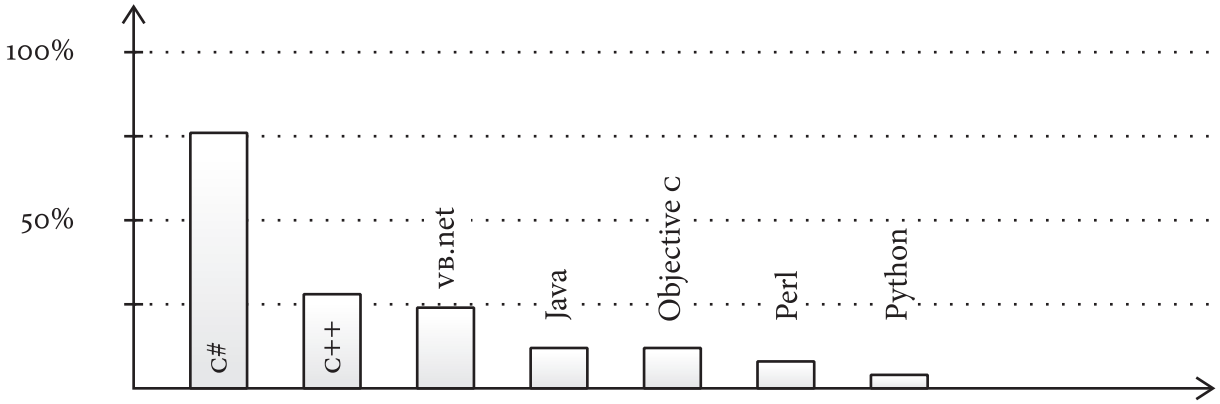


Slika 10. Uporaba programskih jezikov in tehnologij razvijalcev spletnih aplikacijskih rešitev.

B.2.2 Namizne, mobilne in vgradne aplikacijske rešitve

Skupno 25 oseb ne razvija spletnih aplikacij. Pri tem 92% (23 oseb) razvija namizne aplikacije, 4% (1 oseba) mobilne aplikacije in 4% (1 oseba) vgradne aplikacije. Od 23 oseb, ki razvijajo namizne aplikacijske rešitve jih:

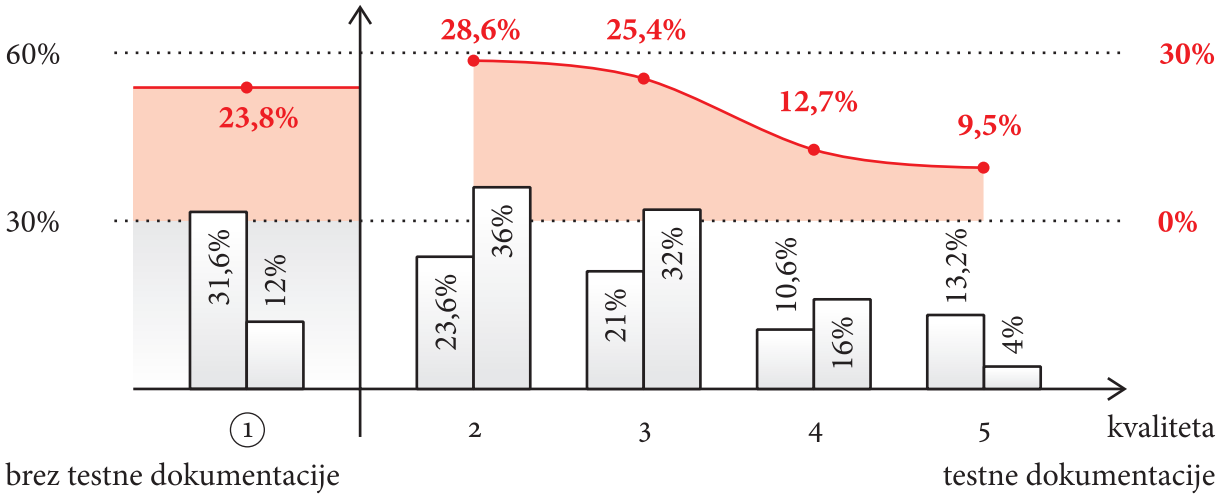
- 96% razvija za operacijski sistem Microsoft Windows,
- 4% razvija za operacijski sistem Apple iOS (ena oseba).



Slika 11. Uporaba primarnih programskih jezikov razvijalcev namiznih aplikacijskih rešitev.

B.2.3 Kvaliteta testnega načrta

Če najprej pogledamo splošno sistematizacijo testiranja aplikacijskih rešitev, katera se prične s testnim načrtom, lahko ugotovimo, da je skoraj ¼ vseh projektov brez vsakršnega testnega načrta, kar je strah vzbujajoče dejstvo. Pri spletnih aplikacijskih rešitvah to velja za skoraj ⅓ vseh projektov, kar prinaša mnogo prostora za izboljšanje razvojnega procesa.



Slika 12. Diagram kvalitete testnega načrta. Levi stolpci kažejo vrednosti spletnih aplikacijskih rešitev, desni pa namiznih, mobilnih in vgradnih aplikacijskih rešitev skupaj. Rdeči del diagrama so skupne kumulative za vse tipe rešitev.

Slika 12 prikazuje kvaliteto testnega načrta, kakor ga smatrajo vprašane osebe v raziskavi. Kakor je razvidno iz diagrama, je skupno zelo majhen odstotek (9,5%) projektov oziroma razvojnih skupin, ki imajo zelo kvaliteten testni načrt. Velika razlika tudi obstaja med spletnimi (13,2%) in nespletnimi rešitvami (4%).

Zelo zanimiva ugotovitev je tudi povprečna kvaliteta testnega načrta, ki znaša skupno in primerjalno (vrednost v točkah od 2 do 5):

- skupna povprečna kvaliteta: 3,04;
- povprečna kvaliteta za spletne rešitve: 3,19;
- povprečna kvaliteta za namizne/mobilne/vgradne rešitve: 2,86.

Posredno to pomeni, da kadar imajo spletne aplikacijske rešitve testni načrt, je ta v povprečju kvalitetnejši, kot pri nespletnih rešitvah. Žal je dejstvo takšno, da je spletnih rešitev brez testnega načrta skoraj 3-krat več kot namiznih. In to kljub dejstvu, da potrebujejo razvijalci spletnih aplikacijskih rešitev več znanj v obliki različnih tehnologij in jezikov (Slika 10 in Slika 11). To predstavlja večje tveganje programskih napak kakor pri namiznih aplikacijah, kjer razvijalci lahko večino stvari napišejo z uporabo enega samega jezika in tehnologije okrog njega.

B.2.4 Vključevanje testerjev v razvojne skupine

Kljub vpeljevanju sistematizacije testiranja podjetja danes ne upoštevajo prednosti ročnega testiranja aplikacijskih rešitev, ki ga izvajajo namenski testerji. Na vprašanje o tem ali imajo v razvojni skupini tudi testerje, katerih izključno delo je testiranje aplikacijskih rešitev v razvoju, so bili odgovori sledeči:

- skoraj polovica (49,2%) anketiranih je odgovorila, da takšnih ljudi v svojih razvojnih skupinah nimajo;
- slaba tretjina (30,2%) razvojnih skupin ima tudi testerje, ki pa žal izvajajo več ali manj le ad-hoc testiranje;
- dobra petina (20,6%) anketiranih pravi, da imajo testerje, ki opravljajo svoje delo na način, kot bi ga bilo potrebno opravljati na vseh projektih v profesionalnem razvojnem okolju: sistematično, sledljivo in merljivo.

Iz odgovorov v raziskavi je razvidno, da imajo vsi projekti s sistematičnimi testerji tudi testni načrt, katerega povprečna ocena kvalitete znaša kar 3,92 točke.

Na podlagi odgovorov bi rad opozoril na še eno dejstvo, ki kaže na površno poznavanje področja sistematičnega testiranja. Obstajajo namreč kontradiktorni projekti, kjer obstaja kvaliteten testni načrt, testerjev pa ni ali pa izvajajo kvečjemu ad-hoc testiranje. Torej proces izvedbe projekta pravilno zastavijo, izvajajo pa žal ne. V številkah to pomeni takole:

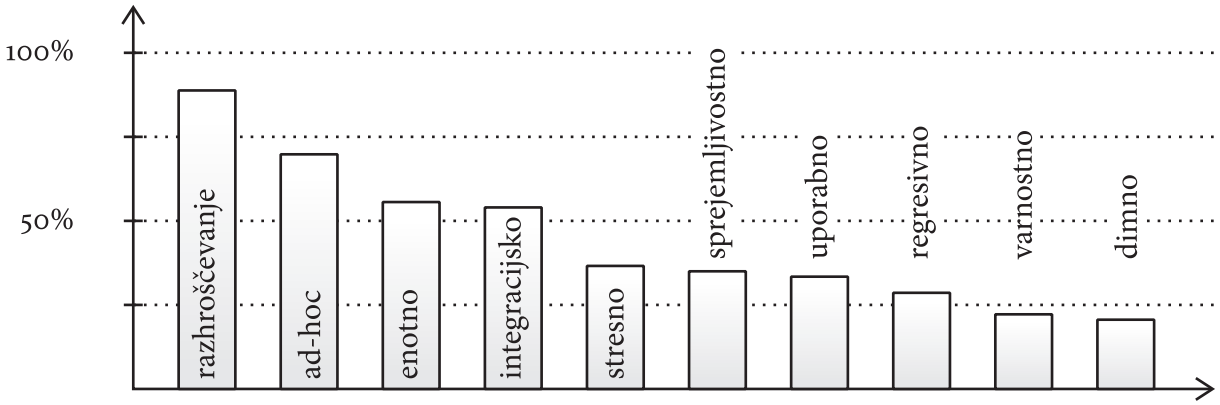
- kvaliteta načrta vsaj 3 točke: 28,6% projektov z neustreznim testiranjem;
- kvaliteta načrta vsaj 4 točke: 9,5% projektov z neustreznim testiranjem;
- kvaliteta načrta 5 točk: 1 projekt oz. 1,6% z neustreznim testiranjem.

Zgornje vrednosti zajemajo neustrezno testiranje, vendar če pri teh projektih upoštevamo tudi avtomatizirane teste, lahko ugotovimo, da je nekaj teh projektov popolnoma neustrezno testiranih. Torej rezultati brez posebnega testerja kot člana razvojne skupine, kot tudi brez avtomatiziranih testov:

- kvaliteta vsaj 3 točke: 9,5% projektov s popolnoma neustreznim testiranjem;
- kvaliteta vsaj 4 točke: 1,6% oziroma en projekt;
- kvaliteta načrta 5 točk: takšnih projektov ni torej 0%.

B.2.5 Vrste testiranj

V vprašalniku je bilo vprašanje katere vrste testov na projektih izvajajo posamezne razvojne skupine. Po pričakovanjih so najpogostejša testiranja razhroščevanje programov in nesistematično ad-hoc testiranje. Sicer pa je uporaba testiranj takšna, kot kaže diagram.

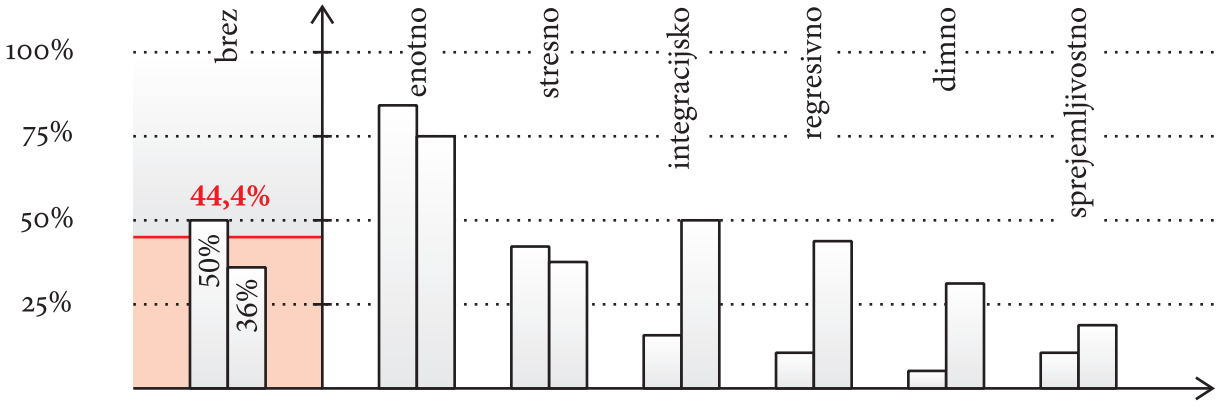


Slika 13. Pogostost uporabe posameznih tipov testiranj pri razvoju aplikacijskih rešitev.

Na vprašanje o testiranju skript podatkovne baze jih je le 23,8% odgovorilo, da testirajo tudi te skripte, 15,9% jih je odgovorilo, da te vrste testiranja ne poznajo, preostalih 60,3% pa jih je odgovorilo, da teh testov ne izvajajo.

B.2.6 Avtomatizirano testiranje

Od vseh odgovorov vprašalnika v raziskavi je skupno 44,4% takšnih razvojnih skupin, kjer ne izvajajo avtomatiziranega testiranja, kar posledično pomeni, da je razvoj teh projektov dražji, kot bi bil sicer z uvedbo avtomatizacije, če je le-ta mogoča. Če upoštevamo le spletne aplikacijske rešitve je ta odstotek višji in sicer kar 50%.



Slika 14. Diagram uporabe avtomtiziranega testiranja aplikacijskih rešitev. Levi stolpci so vezani na spletne aplikacijske rešitve, desni na namizne. Rdeča črta na levem delu diagrama prikazuje skupno kumulativo.

Zgornji diagram na levi strani ordinatne osi prikazuje odstotek razvojnih skupin, kjer ne izvajajo avtomatiziranega testiranja aplikacijskih rešitev in zato predstavlja odstotno vrednost vseh

odgovorov vprašalnika glede na tip aplikacijske rešitve. Rdeča črta in odstotna vrednost ponazarjata skupno kumulativo ne glede na tip.

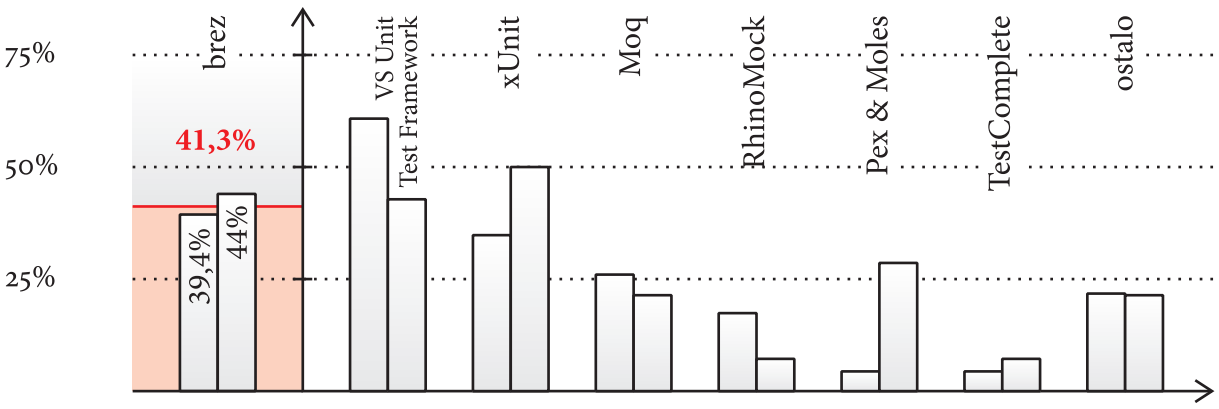
Desna stran ordinatne osi prikazuje odstotne vrednosti avtomatizacije testiranj, vendar ne vključuje razvojnih skupin, kjer avtomatizacije ne izvajajo. Torej je prikazana porazdelitev avtomatizacije testiranj tistih razvojnih skupin, kjer avtomatizacijo uporabljajo.

Iz prikazanih podatkov je razvidno, da se pri razvoju spletnih aplikacijskih rešitev v veliki meri avtomatizira predvsem testiranje enot in stresno testiranje. Ostali tipi so redkeje avtomatizirani. Pri namiznih rešitvah je ta porazdeljenost drugačna, saj se stresno testiranje avtomatizira redkeje od integracijskega in regresijskega. Rezultat ni presenetljiv, saj je uporabniški vmesnik namizne aplikacije v interakciji z enim samim uporabnikom. Zato je bolj pomembno integracijsko testiranje, kjer so testirane skupne procesne funkcionalnosti (na primer baza podatkov). Kadar je namizna aplikacijska rešitev povezana na skupne procesne funkcionalne enote, je kljub temu obseg uporabnikov mnogo manjši, kakor pri široko dostopnih spletnih rešitvah, kjer je število uporabnikov lahko neprimerljivo večje. Temu lahko pripišemo manjšo potrebo po stresnem testiranju namiznih aplikacij.

Slika 14 prav tako prikazuje avtomatizacijo le spletnih ter namiznih rešitev. Mobilne in vgradne niso vključene. Osebe, ki so odgovarjale na vprašalnik in so izbrale primarni razvoj mobilnih ali vgradnih rešitev, ne izvajajo avtomatiziranega testiranja.

B.2.7 Uporaba testnih orodij in knjižnic

Določenih testov ne moremo avtomatizirati povsem brez orodij. Še posebej je nemogoče avtomatizirati testiranja, pri katerih je potrebno simulirati uporabnikovo interakcijo z uporabniškim vmesnikom. V ta namen se uporabljajo različna orodja, ki lahko posnamejo interakcijo in jo kasneje ponovijo s predvajanjem. Kjer orodja niso nujno potrebna je smotrna uporaba knjižnic, ki močno olajšajo in skrajšajo pisanje testov. Sledeči diagram prikazuje pogostost uporabe testnih orodij in knjižnic.



Slika 15. Uporaba knjižnic in orodij pri avtomatiziranem testiranju aplikacijskih rešitev. Levi stolpci prikazujejo spletne desni pa namizne razvojne skupine.

Diagram prikazuje najpogosteje uporabljana orodja in knjižnice, ki so razvojnim skupinam v pomoč pri avtomatiziranem testiranju. Podobno kot na prejšnjem diagramu tudi tu leva stran ordinate povzema kumulative z upoštevanjem razvojih skupin, ki orodij in knjižnic ne uporabljajo, desna pa zajema porazdelitev uporabe samo tistih razvojnih skupin, ki pri avtomatizaciji uporabljajo orodja in knjižnice.

Tudi pri tem vprašanju se je izkazalo, da orodja in knjižnice niso v uporabi pri razvojnih skupinah, ki razvijajo mobilne ali vgradne aplikacijske rešitve. Zato diagram prikazuje segregacijo le na spletne in namizne.

Zadnji par stolpcev diagrama je vezan na vsa ostala orodja in knjižnice, med katerimi so anketiranci izbrali ali vpisali sledeča:

- Selenium,
- WatiX,
- Squish – komercialni izdelek za testiranje različnih tipov aplikacijskih rešitev in omogoča snemanje/predvajanje za različne tipe aplikacij (tudi mobilne),
- Boost Test – knjižnica za testiranje C++ aplikacijskih rešitev,
- HP Quick Test Professional ter
- Visual T# – knjižnica za pisanje testov, ki vpeljuje svoj testni jezik in je integrirana v .net okolje.

B.2.8 Razhroščevanje

V vprašalniku je bilo tudi nekaj vprašanj v zvezi z razhroščevanjem. Vprašanja so bila o orodjih, ki jih za upravljanje programskih napak razvojne skupine uporabljajo ter tudi nekaj splošnih vprašanj o vključevanju razhroščevanja v razvojni proces.

Na vprašanje o tem ali razvojne skupine uporabljajo kakršnokoli orodje za upravljanje s programskimi napakami, so bili odgovori sledeči:

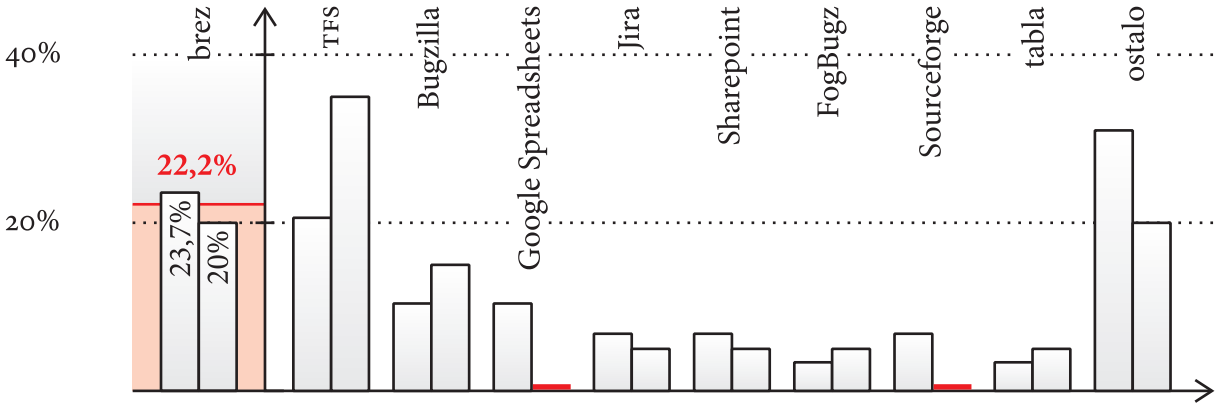
- 22,2% jih je odgovorilo, da programskih napak sploh ne sledijo, kar je zelo alarmanten podatek, ki z veliko verjetnostjo na koncu rezultira v slabih aplikacijskih rešitvah. Vzrok zanj pa ni nujno nepoznavanje ali izogibanje temu delu. Glede na lastne pretekle izkušnje lahko podam svoje subjektivno mnenje: vzrok za takšen rezultat je pogosto optimističen načrt izvedbe aplikacijske rešitve, ki ima močan vpliv na neuspeh razvojnih skupin kasneje – žal je vzrok na koncu pogosto povsem drugače prikazan, kot pa je v resnici.
- 20,6% anketirancev je odgovorilo, da sicer imajo orodje za upravljanje s programskimi napakami, vendar ga ne uporabljajo na pravi način ali pa sploh ne.
- preostalih 57,1% jih je odgovorilo, da orodje za upravljanje s programskimi napakami učinkovito in pogosto uporabljajo.

Kot se je izkazalo, razvojne skupine uporabljajo zelo različna orodja za upravljanje s programskimi napakami. Glede na to, da več kot ¾ spletnih razvojnih skupin in velika večina namiznih

razvija v .net okolju je pričakovano najbolj razširjeno orodje Microsoft Team Foundation Server (na kratko s kratico TFS), ki nudi celovitejše upravljanje razvoja aplikacijskih rešitev:

- hramba programske kode – verzioniranje, razvejitve ipd.;
- zbirka podatkov – programske napake, razvojne naloge, ocenjevalne naloge itd.;
- poročanje – zelo širok nabor poročil v zvezi z razvojem rešitve;
- sprotna integracija s prevarjanjem programov in izvajanjem avtomatiziranih testov;
- integracija z razvojnim orodjem Microsoft Visual Studio Team System.

Kljub tesni integraciji TFS v razvojno okolje, katerega glede na opravljeno raziskavo uporablja večina razvojnih skupin, pa orodje ne izstopa tako izrazito, kot bi lahko. Glede na lastne izkušnje s tem orodjem je morda vzrok v nestabilnem delovanju sprotne integracije in preobširnosti celotnega orodja. Z nepopolnim poznavanjem možnosti, nas lahko uporaba TFS ohromi namesto podpira, saj delo z njim postane časovno prepotratna naloga.



Slika 16. Najpogosteje uporabljane rešitve za upravljanje s programskimi napakami. Levi stolpci prikazujejo spletne razvojne skupine desni pa namizne/mobilne/vgradne. Rdeča črta na levem delu diagrama prikazuje skupno kumulativo razvojnih skupin brez rešitev za upravljanje s programskimi napakami.

Podobno kot pri zadnjih dveh diagramih tudi tukaj levi del prikazuje skupne kumulative razvojnih skupin, ki ne uporabljajo nobenega orodja za upravljanje programskih napak. Desno od ordinatne osi so prikazane vrednosti le za tiste skupine, ki uporabljajo določene rešitve za sledenje programskih napak.

Kakšno je tveganje, če programskih napak ne upravljamo? Mnogo razvijalcev meni, da si bo očitne napake zapomnilo in odpravilo kasneje. Žal se običajno zgodi ena od dveh stvari: ali nanje pozabijo, ker se vmes pojavi še več novih, ali pa jih zaradi preobilice dela (razvoja novih funkcionalnosti) nikoli ne uspejo odpraviti. In ker napake niso nikjer vnešene, je zelo težko ocenjevati stabilnost, regularnost in kvaliteto končnega izdelka oziroma aplikacijske rešitve.

Na vprašanje, če odpravljajo programske napake preden se lotijo novih funkcionalnosti, je kar 63,3% anketirancev odgovorilo pritrdilno. Nadaljnjih 26,5% jih meni, da sicer programske napake niso prioritetne, kljub vsemu pa jih kar nekaj odpravijo. Zadnjih 10,2% razvojnih skupin

programskih napak ne odpravlja, dokler niso razvite vse funkcionalnosti. Odpravljanje hroščev je najcenejše, če jih odpravimo nemudoma, ko se pojavijo. Ta desetina razvojnih skupin žal potroši mnogo virov (razvojnih, časovnih in finančnih) za odpravljanje programskih napak.

B.2.9 Sklepne ugotovitve raziskave

Kateri so torej dejavniki, zaradi katerih lahko smatramo, da izbrana razvojna skupina izvaja kvaliteten, učinkovit in sistematičen razvoj? Nekaj teh dejavnikov je bilo vključenih v okvir vprašalnika raziskave in so sledeči:

- testerji morajo biti del razvojne skupine,
- razvojna skupina izvaja tudi avtomatizirano testiranje,
- upravljanje s programskimi napakami je del razvojnega procesa,
- odpravljanje programskih napak je prioritetna naloga,
- projekt ima testni načrt.

Z upoštevanjem teh začetnih pogojev dobimo sledeče rezultate, ki izpostavljajo razvojne skupine, ki so ali kvalitetne ali pa na zelo dobri poti, da s primernimi nadgraditvami razvojnega procesa takšne postanejo:

- skupno je takšnih razvojnih skupin le 14,3%;
- kvaliteta testnih načrtov v povprečju znaša 3,4 točke;
- 55,6% je spletnih in 44,4% namiznih razvojnih skupin (0% mobilnih/vgradnih);
- od tega jih ⅓ uporablja Team Foundation Server za nadzor razvoja.

Če želimo ugotoviti kolikšen odstotek razvojnih skupin je že kvalitetnih in katerih razvojni proces je na visoki ravni, ki potrebuje morda malenkostno uravnavanje, potem moramo pri zgornjih začetnih pogojih upoštevati dve dodatni predpostavki:

- testerji izvajajo sistematično testiranje in ne le ad-hoc metode ter
- razvojna skupina ves čas razvoja učinkovito uporablja orodje za upravljanje s programskimi napakami.

Ob upoštevanju teh dveh dodatnih pogojev k začetnim, dobimo sledeče rezultate, ki so vezani na najboljše razvojne skupine iz raziskave:

- 7,9% razvojnih skupin izvaja kvaliteten, učinkovit in sistematičen razvoj;
- kvaliteta njihovih testnih načrtov ima v povprečju vrednost 3,7 točke;
- 60% je spletnih razvojnih skupin in 40% namiznih.

Razvojnih skupin, ki izvajajo proces razvoja aplikacijskih rešitev na popolnoma neprimeren način je le 6,3%. Kljub vsemu pa ta odstotek ni zanemarljiv. Od teh je ena razvojna skupina, ki izdeluje vgradne aplikacijske rešitve, katerih razvojni proces je zagotovo zelo drugačen od klasičnega. V primeru, da razvijajo kvantitativno dovolj rešitev, bi z dodatnim testerjem in orodjem

za upravljanje s programskimi napakami lahko mnogo pridobili. Konec koncev je tester običajno cenejši od razvijalca, zato je pametno ročno testiranje iz razvijalcev prenesti na testerje.

Preostanek in večinski delež razvojnih skupin je nekje vmes med slabimi in relativno dobrimi, kar nakazuje na primeren rezultat raziskave, saj so kvalitativno stvari običajno porazdeljene po gaussovi krivulji.

Literatura in spletni viri

[1] Myers, Glenford J. *The art of software testing: revisited and updated by Tom Badgett and Todd M. Thomas with Corey Sandler.* 2. izdaja. Hoboken, New Jersey, USA: John Wiley & Sons Inc., 2004. ISBN 0-471-46912-2

[2] Patton, Ron. *Software Testing*. 2. izdaja. Indianapolis, USA: Sams Publishing, 2006. ISBN 0-672-32798-8

[3] Everett, Gerald D. in McLoud, Jr., Raymond. *Software testing: testing across the entire software development lifecycle*. Hoboken, New Jersey, USA: John Wiley & Sons Inc., 2007. ISBN 978-0-471-79371-7

[4] Martin, Robert C. *Clean Code: A handbook of agile software craftsmanship*. New Jersey, USA: Prentice Hall, Inc., 2008. ISBN 978-0-13-235088-4

[5] Shore, James in Warden, Shane. *The art of agile development*. Sebastopol, California, USA: O'Riley Media, Inc., 2007. ISBN 978-0-596-52767-9

[6] Burnstein, Ilene. *Practical software testing: a process-oriented approach*. Chicago, Illinois, USA: Springer-Verlag, News York, Inc., 2003. ISBN 0-387-95131-8

[7] Galloway, Jon et al. *Professional Asp.net MVC 2.* Indianapolis, Indiana, USA: Wiley Publishing, Inc., 2010. ISBN 978-0-470-64318-1

[8] Ibrahim, Emad. *Asp.net MVC 1.0 Test Driven Development: Problem Design Solution.* Indianapolis, Indiana, USA: Wiley Publishing, Inc., 2009. ISBN 978-0-470-44762-8

[9] Gelperin, David in Hetzel, William C. *The growth of software testing.* Communications of the ACM. *junij 1988, let. 31, št. 6, str. 687 – 695*

[10] Fowler, Martin. *Inversion of control.* (online) 26. junij 2005.
Dostopno na naslovu: http://martinfowler.com/bliki/InversionOfControl.html

[11] Wikipedia. *Software testing.* (online)
Dostopno na naslovu: http://en.wikipedia.org/wiki/Software_testing

[12] Ronney, Paula. *Microsoft CEO: The 80/20 rule applies to bugs, not just features.* (online: ChannelWeb, 3. oktober 2002) *Dostopno na naslovu:* http://www.crn.com/news/security/18821726/microsofts-ceo-80-20-rule-applies-to-bugs-not-just-features.htm

[13] Wikipedia. *Model–View–Controller.* (online) *Dostopno na naslovu:* http://en.wikipedia.org/wiki/Model–View–Controller

[14] Martin, Robert C. *The Dependency Inversion Principle*. C++ Report. *maj 1996*

[15] Fowler, Martin. *Inversion of Control Containers and the Dependency Injection pattern.* (online) 23. januar 2004. *Dostopno na naslovu:* http://martinfowler.com/articles/injection.html

[16] Karem, Cem et al. *Lessons learned in software testing: a context driven approach.* Hoboken, New Jersey, USA: John Wiley & Sons Inc., 2002. ISBN 978-0-471-08112-8

[17] IEEE Standards Board. *IEEE Standard for Software Unit Testing: An American National Standard.* The Institute of Electrical and Electronics Engeneers, Inc., 1986

[18] McDonald, Mark et al. *The practical guide to defect prevention: Techniques to meed the demand for more reliable software.* Redmond, Washington, USA: Microsoft Press, a division of Microsoft Corporation, 2007. ISBN 978-0-735-62253-1

[19] McConnell, Steve. *Code Complete: A Practical Handbook of Software Construction.* 2. izdaja, Redmond, Washington, USA: Microsoft Press, a division of Microsoft Corporation, 2004. ISBN 978-0-735-61967-8

[20] Wikipedia. *System testing.* (online) *Dostopno na naslovu:* http://en.wikipedia.org/wiki/System_testing

[21] Wikipedia. *Software testability.* (online) *Dostopno na naslovu:* http://en.wikipedia.org/wiki/Software_testability

[22] Wikipedia. *Representational State Transfer.* (online) *Dostopno na naslovu:* http://en.wikipedia.org/wiki/Representational_State_Transfer

[23] Wikipedia. *Regression Testing.* (online) *Dostopno na naslovu:* http://en.wikipedia.org/wiki/Regression_testing