UNIVERZA V LJUBLJANI
EKONOMSKA FAKULTETA

MAGISTRSKO DELO

IZBIRA PROGRAMSKEGA ORODJA ZA AVTOMATIZACIJO TESTIRANJA PROGRAMSKE OPREME V IZBRANEM PODJETJU

Ljubljana, september 2015 BLAŽ BRENČIČ

IZJAVA O AVTORSTVU

Podpisani Blaž Brenčič, študent Ekonomske fakultete Univerze v Ljubljani, izjavljam, da sem avtor magistrskega dela z naslovom Izbira programskega orodja za avtomatizacijo testiranja programske opreme v izbranem podjetju, pripravljenega v sodelovanju s svetovalcem prof. dr. Tomažem Turkom.

Izrecno izjavljam, da v skladu z določili Zakona o avtorski in sorodnih pravicah (Ur. l. RS, št. 21/1995 s spremembami) dovolim objavo magistrskega dela na fakultetnih spletnih straneh.

S svojim podpisom zagotavljam, da:

- je predloženo besedilo rezultat izključno mojega lastnega raziskovalnega dela;
- je predloženo besedilo jezikovno korektno in tehnično pripravljeno v skladu z Navodili za izdelavo zaključnih nalog Ekonomske fakultete Univerze v Ljubljani, kar pomeni, da sem
    - poskrbel, da so dela in mnenja drugih avtorjev oziroma avtoric, ki jih uporabljam v magistrskem delu, citirana oziroma navedena v skladu z Navodili za izdelavo zaključnih nalog Ekonomske fakultete Univerze v Ljubljani, in
    - pridobil vsa dovoljenja za uporabo avtorskih del, ki so v celoti (v pisni ali grafični obliki) uporabljena v tekstu, in sem to v besedilu tudi jasno zapisal;
- se zavedam, da je plagiatorstvo – predstavljanje tujih del (v pisni ali grafični obliki) kot mojih lastnih – kaznivo po Zakonu o avtorski in sorodnih pravicah (Ur. l. RS, št. 55/2008 s spremembami);
- se zavedam posledic, ki bi jih na osnovi predloženega magistrskega dela dokazano plagiatorstvo lahko predstavljalo za moj status na Ekonomski fakulteti Univerze v Ljubljani v skladu z relevantnim pravilnikom.

V Ljubljani, dne 12.09.2015 Podpis avtorja:

KAZALO

KAZALO SLIK

KAZALO TABEL

UVOD

Programske rešitve postajajo vedno bolj kompleksne, vzporedno s tem se povečuje tudi potreba po ustreznem zagotavljanju kakovosti programske opreme.

V začetnih fazah razvoja računalništva so programsko opremo večinoma testirali kar programerji sami, z naraščajočo kompleksnostjo sistemov in zahtevnostjo testiranja se je pojavila potreba po specializiranih strokovnjakih, testerjih programske opreme. Ob tem je testiranje sprva potekalo ročno, kasneje so se, z naraščajočo kompleksnostjo tako programske opreme kot tudi samega testiranja, pojavila orodja za avtomatizacijo, ki ne nadomeščajo ročnega testiranja, temveč ga »razširijo«, saj omogočajo, da se v krajšem času izvede več opravil, s čimer je človek razbremenjen monotonih in ponavljajočih nalog.

Programske opreme za avtomatiziranje testiranja programske opreme je trenutno na trgu zelo veliko. Gledano dolgoročno je naložba v nakup takšne strojne opreme praktično zanemarljiva, a kljub temu lahko postane problematična ali vprašljiva, če se kasneje ugotovi, da je bila neprimerna. Pri vpeljavi orodja za avtomatizacijo testiranja (v nadaljevanju AT) je največji strošek predvsem čas tistih, ki skrbijo za vpeljavo AT, saj bi ga lahko porabili za druge stvari.

Izbira orodja za AT je zgolj en korak v procesu izboljševanja testiranja programske opreme oz. njenega razvoja, ob čemer pot do uspeha ne vodi naravnost. Veliko različnih projektov uvajanja AT je pokazalo, da je ta uspeh lahko, in je bil dosežen, ne glede na razlike v področjih, programskem okolju in ciklih razvoja programske opreme. Prav tako ni preprostih odgovorov na vprašanje, kaj lahko pripelje do neuspeha pri uvajanju AT, obstajajo samo določene skupne točke (Graham & Fewster, 2012, str. 1; Weinberg, 2008 , str. 154).

Bozhkova (2014., 12. november) trdi, da je za uspeh uvedbe avtomatizacije testiranja v podjetje potrebno sodelovanje različnih skupin zaposlenih, predvsem je ključno sodelovanje razvijalcev programske opreme in testerjev, ki so zadolženi za avtomatizacijo. V kolikor programska oprema ni primerna za avtomatizacijo testiranja, bo projekt uvedbe težko dosegel takšno stopnjo uspešnosti, kot bi jo, če bi bila programska oprema bolje prilagojena avtomatizaciji. Po raziskavi Bozhkove so na drugem mestu realistična pričakovanja (vrhnjega) managementa.

V magistrski nalogi bom proces izbire orodja za AT predstavil na primeru izbranega podjetja, v katerem razvijajo različne programske rešitve v oblaku za logistične procese, od vnosa naročil do odpreme pošiljk, njihovega sledenja, optimizacije logističnih poti in podpore ravnanju s kontejnerji. Veliko rešitev za stranke podjetja je narejenih *po meri*. Zaradi programske opreme v oblaku lahko spremembe v kodi za eno stranko vplivajo na vse stranke, zato je zagotavljanje ustrezne kakovosti zelo pomembno in hkrati tudi izjemno zahtevno. Podjetje želi z uvedbo AT produkte bolje prilagoditi AT ter hkrati narediti testiranje učinkovitejše in hitrejše.

Namen magistrske naloge je izbira programskega orodja za avtomatizacijo testiranja programske opreme v izbranem podjetju na podlagi značilnosti in načina razvoja programske opreme v podjetju.

Magistrska naloga lahko služi kot pomoč tako ostalim podjetjem, ki se znajdejo pred zahtevno izbiro orodja, kot tistim, ki jih področje zanima bolj z raziskovalnega vidika.

Cilji magistrskega dela so:

- Na podlagi relevantne literature postaviti jasno definicijo programske opreme in AT.
- Predstavitev trenutnega stanja razvoja in testiranja programske opreme v izbranem podjetju.
- Predstavitev procesa uvajanja AT.
- Predstavitev kriterijev in funkcionalnih zahtev za izbiro orodja.
- Primerjava orodij za AT, ki se ponujajo podjetju kot ena izmed možnih rešitev.
- Izvesti primerjavo med orodji in izbrati najbolj primernega.

Pri pisanju magistrske naloge bom uporabil predvsem dve metodi dela, in sicer pregled literature in študijo primera. Tako se bom v prvem, bolj teoretičnem delu, opiral predvsem na analizo sekundarnih virov. V drugem, bolj praktičnem delu, bo glavna metoda dela predvsem študija primera izbranega podjetja, s katero bom prikazal proces uvajanja programske rešitve za AT. Pri analizi študije primera bom uporabil tako znanje, pridobljeno tekom študija, kot praktične izkušnje, pridobljene z delom v razvoju programskih rešitev.

Magistrsko delo bo sestavljeno iz 5 glavnih vsebinskih poglavij. Prvi del bo teoretičen in bo zajemal dve poglavji; v prvem bom predstavil testiranje programske opreme, v drugem teoretični vidik avtomatizacije testiranja programske opreme. Ti poglavji bosta temelj za nadaljnje razumevanje besedila. Preostala poglavja bodo bolj praktično usmerjena, in sicer bom v tretjem poglavju predstavil izbrano podjetje, trenuten proces razvoja in testiranja programske opreme. Temu bo sledila predstavitev nekaterih orodij za avtomatizacijo testiranja programske opreme, med katerimi se podjetje odloča, v zadnjem poglavju bom opisal primerjavo med temi orodji. Temu bo sledil zaključek magistrske naloge, v katerem bom povzel glavne ugotovitve.

1 TESTIRANJE PROGRAMSKE OPREME

V prvem poglavju bom predstavil teoretične vidike testiranja programske opreme. Najprej bom na kratko definiral samo testiranje programskih orodij in predstavil njihovo zgodovino. V naslednjih podpoglavjih bom naredil teoretični prerez, od metod testiranja, nivojev, vrst, do modelov testiranja programske opreme. Ta poglavja bodo služila za boljšo predstavo naslednjemu teoretičnemu poglavju in tudi samemu razumevanju problematike magistrskega dela.

1.1 Definicija testiranja programske opreme

Islovar (b.l.) **testiranje oz. preizkušanje** opredeljuje kot preverjanje ustreznosti izdelka ali storitev, lahko gre tudi za preizkušanje znanja. Le-to sicer razloži testiranje, vendar je definicija preveč groba za pričujoče besedilo.

ISTQB (2011, str. 13), mednarodna neprofitna organizacija, ki skrbi za certificiranje kompetenc v testiranju programske opreme, trdi, da je preizkušanje programske opreme več kot le golo izvajanje testov, to je le ena izmed aktivnosti. Aktivnosti testiranja se dogajajo pred in po sami izvedbi testa, in sicer vključujejo planiranje in kontrolo, izbiro pogojev za testiranje, načrtovanje in izvedbo testnih scenarijev (»testov«), preverjanje rezultatov, evaluacijo izhodnih kriterijev, izdelavo poročil o sistemu, ki je testiran, in samem procesu testiranja in zaključevanje oz. dopolnjevanje zaključnih aktivnosti, ko je sam proces testiranja končan. Testiranje vključuje tudi izdelavo dokumentov in statičnih analiz ter lahko zasleduje naslednje cilje; iskanje napak, pridobivanje zaupanja o ravni kvalitete, zagotavljanje informacij za odločanje in preprečevanje napak.

Po mnenju omenjene organizacije obstaja 7 principov testiranja programske opreme (ISTQB, 2011, str. 14):

- **Testiranje prikazuje prisotnost napak**: testiranje lahko pokaže prisotnost napak v programski opremi, ne more prikazati njihove odsotnosti. Tudi v primeru, da nobena napaka ni odkrita, to ne pomeni, da napak ni.
- **Izčrpno testiranje ni mogoče**: nemogoče je testirati vse mogoče scenarije in kombinacije. Predvsem to ni smiselno za preproste, trivialne primere. Pred samo izvedbo testiranja je potrebno preučiti glavna tveganja in izdelati načrt, kaj ima višjo prioriteto testiranja.
- **Zgodnje testiranje**: za odkritje čim večjega števila napak je potrebno testiranje začeti opravljati čim prej, usmerjeno naj bo na določene cilje.
- **Grozdenje napak**: testiranje naj bo usmerjeno na področja, za katera se pričakuje največja gostota napak.
- **Paradoks pesticidov**: večkratno ponavljanje istih testnih scenarijev po navadi pomeni, da sčasoma to testiranje ne bo odkrilo nobenih novih napak. Testne scenarije je potrebno redno posodabljati in prilagajati glede na spremembe v programski rešitvi.
- **Testiranje je odvisno od vsebine ali konteksta**: testiranje je izvedeno različno v različnih kontekstih. Varnostno tvegana programska oprema se testira drugače kot npr. spletna trgovina.
- **Zmotno prepričanje o odsotnosti napak**: odkrivanje napak ni smiselno, če sistem ne izpolnjuje pričakovanj in potreb in je neuporaben.

Kvaliteto lahko opišemo z naslednjimi atributi (Kaner, Bach & Pettichord, 2002, str. 60):

- dostopnost,

- zmogljivost,
- združljivost,
- sočasnost,
- združljivost s standardi,
- učinkovitost,
- možnost nameščanja in brisanja,
- lokalizacija,
- vzdržljivost,
- učinkovitost,
- prenosnost,
- obnovljivost,
- zanesljivost,
- skalabilnost,
- varnost,
- podpora,
- pripravljenost na testiranje,
- uporabnost.

Glavni povzročitelji napak so lahko eni izmed naslednjih dejavnikov (Kaner et al., 2002, str. 61–62):

- **Nove stvari**: nove funkcionalnosti lahko ne delujejo.
- **Nove tehnologije**: novi koncepti vodijo do novih napak.
- **Nov(i) trg(i)**: novi kupci bodo produkt videli in uporabljali drugače.
- **Krivulja učenja**: pojavijo se napake, ki so posledica neznanja.
- **Pozne spremembe**: hitre spremembe, zaposleni, ki so nemotivirani in prisiljeni delati hitro, kar vodi do napak.
- **Spremenjene stvari**: spremembe lahko negativno vplivajo na staro kodo.
- **Hitro delo**: nekatere naloge ali projekti trpijo kronično podhranjenost virov, posledično trpijo vsi vidiki kvalitete.
- **Slab dizajn ali implementacija, ki jo je težko vzdrževati**: odločitve glede internega dizajna kode slednjo naredijo tako težko za vzdrževanje, da vsi novi popravki konstantno povzročajo nove probleme.
- **Utrujeni programerji**: dolgotrajno delo, ki poteka več tednov ali celo mesecev, kar negativno vpliva na učinkovitost in povzroča napake.
- **Drugi problemi s kadri**: problemi s pitjem alkohola, zdravstveni problemi, smrt v družini, osebni problemi med programerji itd.
- »**Ni izumljeno tukaj**«: zunanje komponente lahko povzročijo probleme.
- »**Ni v proračunu**«: naloge s premajhnim proračunom so lahko izpeljane slabo.
- **Nejasnost**: nejasne zahteve v specifikaciji ali drugih dokumentih lahko vodijo do nepravilnih ali konfliktnih implementacij.

- **Konfliktne zahteve**: nejasnost pogosto skrije konflikt, rezultat je izguba vrednosti za določeno osebo.
- **»Evolucionarne« spremembe zahtev**: ljudje se zavedajo, kaj hočejo, ko je produkt končan.
- **Kompleksnost**: kompleksna koda je lahko hroščata.
- **Hroščatost**: funkcionalnost z veliko poznanimi hrošči ima lahko tudi veliko nepoznanih.
- **Odvisnost**: napake lahko povzročijo druge napake.
- **Nezmožnost testiranja**: riziko počasnega in ne dovolj učinkovitega testiranja.
- **Premalo testiranja enot** (opisano podrobneje v nadaljevanju magistrske naloge): programerji najdejo in odpravijo večino napak. Ubiranje bližnjic je lahko nevarno.
- **Premalo sistemskega testiranja**: programska oprema, ki ni bila dovolj testirana, ima lahko napake.
- **Zanašanje na prejšnje testne strategije, ki so preveč ozko usmerjene**: regresijski in funkcionalni testi lahko povzročijo »zbirko« hroščev, ki »preživijo« več novih verzij programske opreme.
- **Šibka testna orodja**: če orodje ne omogoča odkrivanja določenega tipa napak, je večja verjetnost, da bodo take napake ostale spregledane.
- **Nezmožnost popravljanja**: riziko, da hrošča ni mogoče odpraviti.
- **Napake, vezane na specifični programski jezik.**

1.2 Kratek pregled zgodovine testiranja programske opreme

Charles Babbage je leta 1822 naredil diferencialni stroj, zaradi česar Meerts in Graham (2015) to leto navajata kot čas, v katerem se je prvič pojavilo testiranje programske opreme. Meerts in Graham (2015) postavita kot začetka testiranje programske opreme v leto 1822, ko je Charles Babbage naredil diferencialni stroj. Za magistrsko nalogo bolj zanimiv je čas pred oz. med 2. svetovno vojno, ko so se pojavili temelji današnjega računalništva. Proti koncu 2. svetovne vojne je John von Neumann definiral logičen načrt računalnika, ki uporablja pomnilnik za shranjevanje podatkov. Ta arhitektura se po izumitelju imenuje Von Neumannova arhitektura, na kateri še danes temelji večina sodobnih računalnikov. 6 let kasneje, leta 1951, Joseph M. Juran, ki velja za očeta managementa kakovosti, definira kvaliteto kot »primernost za uporabo« in definira 3 procese managementa kakovosti: načrtovanje, nadzor in izboljševanje kvalitete. Že istega leta Armand Vallin Feigenbaum opredeli kvaliteto kot kupčevo odločenost. Produkt naj bi zagotavljal aktualne in pričakovane potrebe kupca.

Teoretičnim vpogledom o kakovosti sledijo leta, ko se je pojavil prvi množični računalnik in več knjig o programiranju programskih rešitev. Naslednje prelomno leto za testiranje programske opreme je tako leto 1958, ko Gerald M. Weinberg vzpostavi prvo testno ekipo za vesoljski program Merkur. 10 let kasneje, leta 1968, poročilo organizacije NATO vključuje poglavje o zagotavljanju kvalitete (ang. *quality assurance*). Leto kasneje Edsger Dijkstra izreče znano trditev o tem, da testiranje lahko pokaže prisotnost hroščev, nikoli

njihove odsotnosti. To leto nekako nakazuje naslednjo prelomno točko v testiranju programske opreme, do katere je prišlo leta 1970, ko je Winston Royce opisal model razvoja programske opreme in ga poimenoval slap (ang. *waterfall*). Ta model bo podrobneje predstavljen v nadaljevanju. Tu velja omeniti le, da model opisuje razvoj v korakih oz. Sekvencah, Royce sam je hkrati opisal tudi njegove pomanjkljivosti (Meerts & Graham, 2015).

Pomembnih odkritij je bilo v naslednjih letih veliko. V nadaljevanju bodo omenjena tista, ki so bolj povezana z magistrsko nalogo. Navajanje vseh zaradi omejitve strani magistrskega dela nima smisla. Leta 1978 je William Howden definiral funkcijsko testiranje. Leto kasneje je Glendfor Myers izdal knjigo *Umetnost testiranja programske opreme* (ang. *The Art of Software Testing*), ki je bila prva izdana knjiga, namenjena izključno preizkušanju programskih rešitev. Leta 1984 je bila v Združenih državah Amerike organizirana prva konferenca na svetu, ki je bila posvečena izključno testiranju programske opreme. Eno leto kasneje je bil izdan *AutoTester*, prvi program za testiranje drugih programov. Izdalo ga je podjetje Linde Hayes. Naslednje leto je Paul E. Rook predstavil V-model testiranja. Leta 1988 je Cem Kaner prvi na svetu opisal raziskovalno testiranje (ang. *exploratory testing*). Istega leta je Kaner izdal knjigo *Testiranje programske opreme* (ang. *Testing Computer Software*), ki postavi nove standarde v testiranju programske opreme. Naslednje leto je podjetje Mercury izdalo prvo programsko orodje *LoadRunner* za preizkušanje zmogljivosti programske opreme (Meerts & Graham, 2015).

Leta 1990 je Boris Beizer opisal taksonomijo oz. klasifikacijo programskih hroščev. Tri leta kasneje sta bila opisana dva modela, ki sta zanimiva tudi za pričujoče magistrsko delo. Jeff Sutherland, John Scumniotales in Jeff McKenna tega leta opišejo *scrum* razvojni model, Paul Herzlich W-model. Jakob Nielsen in Robert L. Mack eno leto kasneje opišeta metode za preizkušanje uporabnosti, kar je prvo delo o tem področju. Leta 1996 James Bach predstavi hevristično strategijo testiranja. Leta 1998 se zgodi prvo evropsko certificiranje v Evropi, ki sta ga vodila Dorothy Graham in Mark Fewster. Leta 1999 Cem Kaner, James Bach, Brian Marick in Bret Pettichord ustanovijo kontekstno usmerjeno šolo testiranja na principu, da vrednost vsake prakse temelji na njenem kontekstu. V istem letu Dorothy Graham in Mark Fewster izdata knjigo *Avtomatizacija testiranja programske opreme* (ang. *Software Test Automation*), ki postane z razlago problemov, strategij in taktik avtomatizacije nekakšno klasično delo tega področja (Meerts & Graham, 2015).

Ob začetku novega tisočletja, leta 2001, je predstavljen manifest o agilnih metodah razvoja programske opreme in njenih 12 temeljnih principih. *The International Software Testing Qualifications Board* (ISTQB) je bil ustanovljen v Edinburghu leto kasneje. Istega leta Kent Beck predstavi testno voden razvoj (ang. *test driven development*), model, v katerem se najprej napiše testni scenarij, na podlagi katerega se izvede razvoj same programske rešitve. Takrat je IBM izdal tudi Rational Functional Tester, eno izmed prvih orodijj za avtomatizacijo testiranja. Leta 2003 je Chris Evan predstavil domensko voden razvoj (ang. *domain driven developmen*t), model namenjen testiranju kompleksnih poslovnih domen.

2004 je Jason Huggins razvil Selenium, odprtokodno rešitev za avtomatizacijo testiranja programske opreme. V letu 2007 organizacija ISO predstavi standard 29119, ki nadomesti več dotedanjih standardov za testiranje programske opreme. V 2008 Leo van der Aalst izumi izraz testiranje programskih rešitev kot storitev (ang. *Software Testing as a Service*). Leto kasneje Mike Cohn objavi članek o piramidi AT, v katerem zagovarja svojo teorijo, da mora avtomatizacija potekati na treh nivojih: enoti, storitvi in uporabniškem vmesniku. Leta 2011 Alberto Savoia na konferenci podjetja Google izjavi, da je klasično testiranje programske opreme mrtvo, saj večina novoustanovljenih podjetij stremi k stalnemu izdajanju nove programske opreme in nenehnemu razvoju (Meerts & Graham, 2015).

1.3 Metode testiranja programske opreme

V nadaljevanju bom po načelu delitve na črne, sive in bele skrinjice na kratko opisal metode testiranja ter predstavil nekaj prednosti in pomanjkljivosti le-teh.

1.3.1 Testiranje »črna skrinjica«

Testiranje po metodi »črne skrinjice« (ang. *black box testing*) je vsako testiranje, v katerem preizkuševalec ne pozna arhitekture programske opreme in nima dostopa do njene izvorne kode. Po navadi preizkuševalec preko uporabniškega vmesnika vnaša vhodne podatke, ki kot rezultate »proizvedejo« izhodne podatke. Preizkuševalec ne ve ne kje in ne kako so se vhodni podatki predelali v izhodne. Glavne prednosti te metode so, da je zelo primerna za testiranje večjih segmentov kode, da dostop do kode ni potreben, da jasno loči vlogo razvijalca programske opreme in njenega uporabnika in da lahko večje število testerjev programsko opremo testira, brez da bi potrebovali veliko znanja o njeni implementaciji, programskih jezikih ali operacijskih sistemih. Slabosti tega pristopa so, da omogoča zgolj omejeno testiranje programske opreme, saj so izvedeni le določeni, težko izvedljivi testni scenariji. Negativna lastnost je tudi, da imajo testerji o aplikaciji, ki jo testirajo, največkrat le omejeno znanje, ob čemer gre največkrat zgolj za slepo preizkušanje, pri katerem tester določenih segmentov kode, ki so bolj »odporni« na napake, ne more testirati.(Tutorials Point, b.l.; Software Testing Fundamentals, 2011b).

1.3.2 Testiranje »bela skrinjica«

V nasprotju z metodo testiranja, opisano v prejšnjem podpoglavju, je testiranje po metodi »bele skrinjice« (ang. *white box testing*) podrobno preiskovanje kode in interne logike programske rešitve. Da bi tester lahko testiral programsko rešitev po tej metodi, mora tako poznati njeno »notranjost«. Z vpogledom v njeno izvorno kodo tester ugotavlja, kateri del kode povzroča napake. Glavne prednosti te metode so, da zaradi poznavanja »drobovja« programske rešitve tester lahko lažje ugotovi, katere podatke potrebuje za testiranje, pomaga pri optimizaciji kode, odvečne vrstice kode so lahko odstranjene, kar lahko vpliva na boljše delovanje. Zaradi poznavanja kode je mogoče doseči maksimalno mogoče pokritje kode s testnimi scenariji. Glavna slabost tega pristopa so stroški testerja s primernimi izkušnjami. Včasih je nemogoče pregledati vse dele kode in odkriti vse skrite

napake. Ta metoda testiranja je zahtevna tudi z vidika vzdrževanja, saj zahteva specializirana orodja za analizo kode in njeno razhroščevanje (Tutorials Point, b.l.; Software Testing Fundamentals, 2011b).

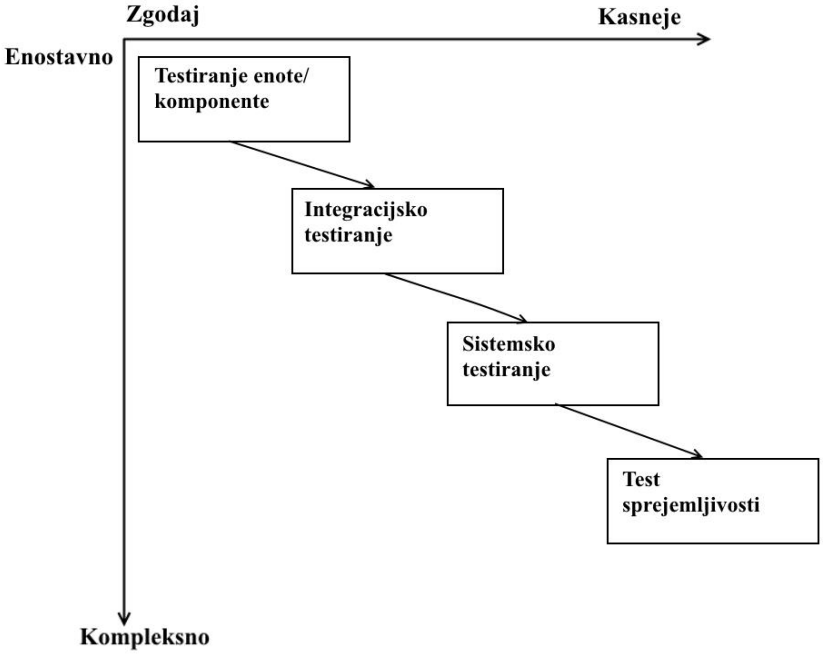
1.3.3 Testiranja »siva skrinjica«

Ta metoda testiranja je nekakšna kombinacija prej opisanih. Po metodi »sive skrinjice« (ang. *grey box testing*) se programsko opremo testira z omejenim poznavanjem notranjega delovanja programske opreme. Več kot vemo o programski rešitvi, bolj uspešno bo naše testiranje. Dobro poznavanje domene sistema za testerja vedno pomeni prednosti pred testerjem, ki tega domenskega znanja nima. Po tej metodi ima tester dostop do načrtov aplikacije in podatkovne baze, s čimer lahko naredi učinkovitejše testne scenarije. Glavna prednost te metode je, da združuje pozitivne lastnosti obeh prej opisanih metod. Testerji se ne zanašajo na izvorno kodo, ampak na funkcionalne specifikacije in definicije vmesnikov. Kljub omejeni količini podatkov, ki so na voljo, tester lahko napiše dobre testne scenarije, še zlasti na področju komunikacijskih protokolov in testiranja podatkovnih tipov. Test je izveden na podlagi »videnja« uporabnika in ne načrtovalca programske opreme. Slabost tega pristopa je, da dostop do izvorne kode ni mogoč, zato ni mogoče opraviti njenega pregleda. Testi so lahko nepotrebni, če jih je načrtovalec sistema izvedel že sam. Testiranje vsakega možnega vhodnega toka podatkov je nemogoče, zato veliko testnih scenarijev ni izvedenih. (Tutorials Point, b.l.; Software Testing Fundamentals, 2011b).

1.4 Nivoji testiranja programske opreme

V tem podpoglavju bodo predstavljeni nivoji testiranja programske opreme.

Slika 1: Nivoji testiranja



Vir: U. Eriksson, Differences between the different levels of tests, ReQtest, 2014.

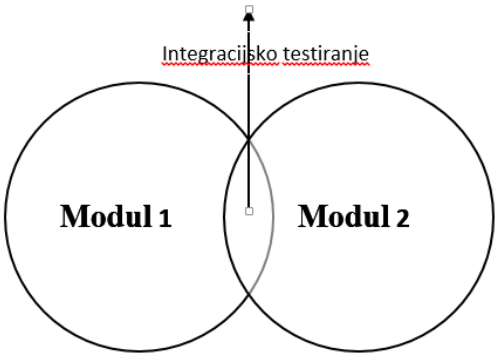
1.4.1 Testiranje enote

Pri testiranju enote oz. komponente (ang. *unit testing*) gre večinoma za uporabo metode bele skrinjice. Namen testiranja komponent je preveriti delovanje vsake individualne komponente programske rešitve, tako da se jo izolira in s testiranjem preverja, ali izpolnjuje zahteve in zaželeno funkcionalnost. Enota se lahko nanaša na funkcijo, program ali proceduro. To testiranje se v glavnem izvaja v začetnih fazah razvoja programske rešitve, izvajajo ga sami razvijalci kode. Glavna prednost testiranja na nivojih je, da se teste lahko izvede ob vsaki spremembi kode, kar privede do hitrejšega odkrivanja napak in njihovega odpravljanja. Slabost tega testiranja je, da ga v glavnem izvajajo samo razvijalci oz. ljudje z dovolj velikimi izkušnjami v programiranju ter da so testi napisani tako, da ustrezajo programerjevi implementaciji in ne nujno specifikaciji. Določene pomanjkljivosti odpravlja »prijateljsko testiranje« (ang. *buddy testing*), v katerem preizkuševalec napiše testni scenarij, preden se razvoj prične. Prednost tega pristopa je, da se poveča sodelovanje med različnimi vrstami zaposlenih in poveča izmenjava znanj in izkušenj med njimi (Eriksson, 2014.; Pearson, 2013; Software Testing Fundamentals, 2011b; Oladimeji, b.l.).

1.4.2 Integracijsko testiranje

Namen **integracijskega testiranja** je testiranje različnih delov sistema v kombinaciji ali skupinah, tako da se lahko preveri, kako delujejo skupaj. S testiranjem enot v skupinah se lahko odkrijejo napake v njihovem medsebojnemu delovanju. Ne glede na to, kako učinkovito deluje posamezna enota, lahko pride, če le-ta ni pravilno integrirana s preostalimi, do negativnega vpliva na celotni sistem. Obstaja več metod za to testiranje, izbira je odvisna predvsem od tega, kako so posamezne enote zgrajene. Dva izmed glavnih pristopov sta »od spodaj navzgor« (ang. *bottom - up*) in »od zgoraj navzdol« (ang. *up - bottom*). V prvem primeru se najprej izvaja testiranje višje ležečih kombinacij v vedno bolj kompleksnih scenarijih. V drugem primeru se testiranje začne s kompleksnimi scenariji in se konča s kasnejšim preizkušanjem enostavnejših delov. To testiranje izvajajo tako razvijalec programskih rešitev kot tudi preizkuševalci. Testiranje se po navadi začne, ko je končana specifikacija programske rešitve in se nadaljuje čez celoten projekt. Poleg omenjenih poznamo še testiranje »veliki pok« (ang. *Big Bang integration testing*) (Eriksson, 2014.; Pearson, 2013; ISTQB Exam Certification, b.l.; Software Testing Fundamentals, 2011b; Oladimeji, b.l.).

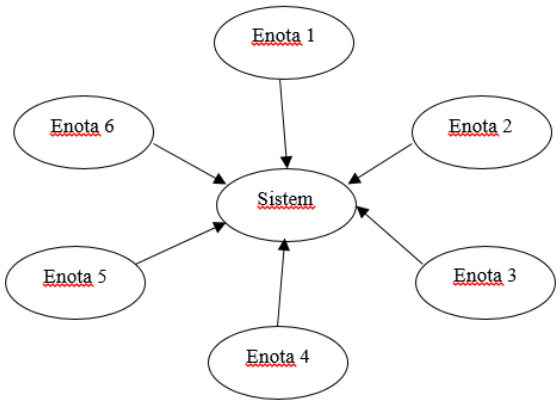
Slika 2: Integracijsko testiranje



Vir: ISTQB Exam Certification (b.l.), What is integration testing?, b.l.

Pri testiranju »veliki pok« se vse komponente ali moduli sistema združijo naenkrat, nato se vse testira kot celota. Prednost tega pristopa je, da so vsi posamezni deli končani, preden se začne testiranje. Slabost je, da se zahteva velik vložek časa in da je zaradi pozne integracije težko odkriti vzroke napak. Ta pristop je prikazan na sliki številka 3 (ISTQB Exam Certification, b.l.).

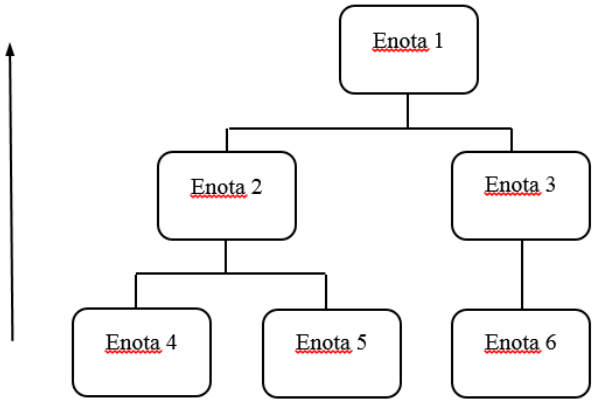
Slika 3: Integracijsko testiranje – »veliki pok«



Vir: ISTQB Exam Certification (b.l.), What is integration testing?, b.l.

Spodnja slika (slika št. 4) prikazuje **pristop »od zgoraj navzdol«.** Testiranje pri tem pristopu poteka od zgornjih »plasti« programske rešitve, prične se pri uporabniškem vmesniku oz. glavnem menuju. Prednost tega pristopa je, da je testiran produkt zelo konsistenten, saj je to testiranje izvedeno v okolju, ki je zelo podobno realnosti. Slabost tega pristopa je, da je osnovna funkcionalnost testirana proti koncu razvojnega cikla (ISTQB Exam Certification, b.l.).

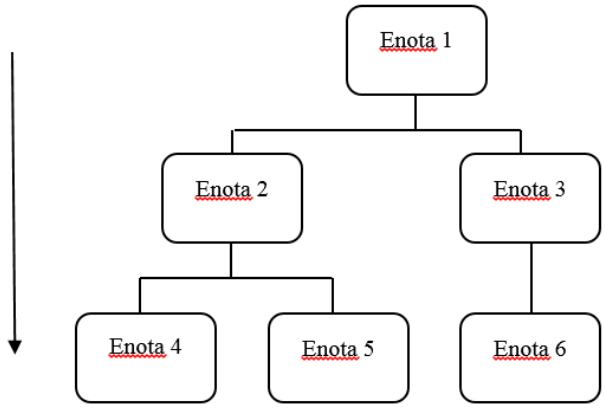
Slika 4: Integracijsko testiranje – »od spodaj navzgor«



Vir: ISTQB Exam Certification (b.l.), What is integration testing?, b.l.

Naslednji pristop, **pristop komponentnega testiranja**, je prikazan na sliki številka 5 in je ravno obraten prej opisanemu, Gre za pristop »od spodaj navzgor«, njegova največja prednost je, da testiranje in razvoj lahko potekata hkrati, slabost, da lahko glavne probleme vmesnikov zaznamo šele na koncu cikla in da je potrebno narediti testne gonilnike za vse nivoje, razen čisto zgornjega (ISTQB Exam Certification, b.l.).

Slika 5: Integracijsko testiranje – »od spodaj navzdol«



Vir: ISTQB Exam Certification (b.l.), What is integration testing?, b.l.

Zadnji pristop komponentnega testiranja je imenovan inkrementalno testiranje. Koda posameznih razvijalcev se vključuje korak za korakom Vsakemu koraku sledi testiranje. Prednost tega pristopa je, da se hibe odkrije prej in se tudi lažje ugotovi, v katerem delu kode je izvor napake. Slabost pristopa je, da je časovno zamuden. (ISTQB Exam Certification, b.l.).

1.4.3 Sistemsko testiranje

Sistemsko testiranje je proces testiranja končanega in popolnoma integriranega strojnega in programskega sistema oz. produkta z namenom preverjanja, ali sistem izpolnjuje podane zahteve oz. specifikacije. Glavni fokus tega testiranja je evaluacija poslovnih in funkcijskih zahtev ter zahtev končnega uporabnika. Gre za testiranje po metodi črne skrinjice, v katerem je zunanje delovanje programske opreme ocenjeno s pomočjo dokumentacije z zahtevami in popolnoma temelji na pogledu končnega uporabnika. Za ta tip testiranja ni potrebno poznavanje interne strukture ali kode. Testiranje se začne, ko se konča integracijsko testiranje. Na kratko povedano bi lahko zapisali, da je sistemsko testiranje dejansko zbirka testov, katerih namen je preučiti delovanje celotnega sistema ali produkta. Sistemsko testiranje vključuje testiranje funkcijskih in nefunkcijskih področij (Guru99, b.l.; Software Testing Class, 2012; Software Testing Help, 2015a).

Sistemsko testiranje se v glavnem osredotoča na (Software Testing Help, 2015a):

- zunanje vmesnike,
- kompleksne funkcionalnosti,
- varnost,
- sposobnost obnovitve,
- zmogljivost,
- gladkost uporabniškega vmesnika,
- dokumentacijo,
- uporabnost,
- obremenitev.

Razlogi, zakaj je sistemsko testiranje pomembno (Software Testing Class, 2012):

- v ciklu razvoja programske opreme je to prvo testiranje, kjer se sistem oz. programsko rešitev testira kot celoto,
- v temu koraku testiranja se preveri, ali sistem izpolnjuje funkcionalne zahteve ali ne,
- sistemsko testiranja omogoča validacijo in verifikacijo arhitekture programske rešitve in poslovnih zahtev,
- testiranje poteka v okolju, ki spominja na produkcijsko okolje, v katerega bo programska oprema na koncu nameščena.

Zaželeno je, da testni načrt vedno vključuje tudi to vrsto testiranja. Za testiranje sistema kot celote morajo biti podane jasne in razumljive zahteve in pričakovanja. Tester mora razumeti uporabo programske rešitve v realnem času. Orodja zunanjih proizvajalcev, različne verzije operacijskih sistemov ipd., lahko vplivajo na funkcionalnost, zmogljivost, varnost, obnovljivost ali možnost za namestitev sistema. Za to je potrebna jasna predstava, kako se bo programska rešitev oz. sistem uporabljal in kakšne težave lahko nastanejo v realnem svetu. Jasna in posodobljena dokumentacija lahko zmanjša številna

nerazumevanja, vprašanja in predvidevanja, ki se lahko pojavijo pri testerju (Software Testing Help, 2015a).

Sistemskega testiranja se je možno lotiti v naslednjih 6 korakih (Software Testing Class, 2012):

1. **Izdelava načrta sistemskega testiranja**: ta korak je eden od pomembnejših. Točke, ki so pokrite v tem načrtu, se lahko razlikujejo tako od organizacije do organizacije kot tudi od projektnega načrta, testne strategije in glavnega testnega načrta. Ne glede na te razlike ta načrt po navadi vključuje: cilje in naloge, obseg, kritična fokusna področja, testni izdelek, testno strategijo, časovni načrt, vhodne in izhodne kriterije, kriterije za prekinitev in nadaljevanje, testno okolje, vloge in zadolžitve ter slovar.
2. **Izdelava testnih scenarijev**: v tej fazi se izdela testne scenarije in primere uporabe. V tem koraku se običajno izbero eno izmed vrst testiranja programske opreme, ki bodo podrobneje opisane v nadaljevanju, v poglavju 1.4. Testni scenarij po navadi vključuje unikatno identifikacijsko šifro, kratek naslov, opis, kako testirati, testne podatke, pričakovan rezultat, in dejanski rezultat.
3. **Priprava testnih podatkov, ki se bodo uporabili za testiranje.**
4. **Izvedba avtomatiziranega testnega scenarija oz. skripte.**
5. **Izvedba ročnega testnega scenarija in posodobitev testnega scenarija v orodju za management testnih scenarijev, če je kateri v uporabi.**
6. **Izdelava poročil o hroščih, preverjanje hroščev in regresijsko testiranje.**
7. **Ponovitev testnega cikla, če je potrebno.**

1.4.4 Uporabniški test sprejemljivosti

Test sprejemljivosti (ang. *user acceptance testing*) ali beta testiranje ali testiranje končnega uporabnika, se običajno šteje kot zadnja faza v stopnji razvoja programske opreme, faza pred končno namestitvijo oz. distribucijo na produkcijskem sistemu. Testiranje poteka na sistemu, ki je najbolj podoben produkcijskemu. Izvajajo ga končni uporabniki, ki bodo produkt tudi uporabljali, ali predstavniki poslovnih uporabnikov. Namen tega testiranja je ugotoviti, ali sistem lahko podpira poslovne in uporabniške scenarije ter zagotavlja, da je sistem primeren in pravilen za poslovno uporabo. Končni uporabniki oz. njihovi predstavniki imajo možnost, da uporabljajo programsko opremo in preverijo, ali vse deluje, tako kot bi moralo, ali so bile kakšne funkcionalnosti spregledane, nepravilno ali sploh ne sporočene. Pri temu testiranju je ključen uporabnik (Bordo, b.l.; Rice, b.l.; Setter, b.l.).

Pogoji za pričetek oz. vhodni kriteriji za test sprejemljivosti so (guru99, b.l.):

- poslovne zahteve morajo biti na voljo,
- koda programske rešitev je končana,
- testi enot, integracijsko in sistemsko testiranje so končani,
- odkritih ni nobenih kritičnih hroščev, ki bi preprečevali začetek testiranja,

- samo »kozmetične« napake so sprejemljive pred pričetkom testiranja,
- regresijsko testiranje je bilo opravljeno in ni odkrilo nobenih večjih napak,
- vse (kritične) odkrite napake pred začetkom testiranja so odpravljene,
- matrika za sledenje testiranja zahtev je končana,
- okolje za testiranje je pripravljeno,
- testni tim je obvestil uporabnike, da lahko pričnejo s testiranjem.

Rice (b.l.) navaja, da so za uspešno izvedbo testiranje sprejemljivosti pomembni naslednji dejavniki:

- **Razumevanje, da se to testiranje izvaja v najbolj neprimernem času v projektu**: večinoma na koncu projekta, saj je to čas, ko je celotni sistem »sestavljen« ali naložen. Pred koncem projekta lahko uporabniki testirajo določene dele, vendar ne celote. Konec projekta je najslabši čas za odkrivanje in odpravljanje napak. Strošek vsakega odkritega in odpravljenega hrošča je 10-krat večji, kot bi bil, če bi bil ta hrošč odkrit že času ustvarjanja zahtev. To je zaradi t. i. efekta »zorenja«, saj je popravek morda potreben tudi v drugih delih sistema. Rešitev za te probleme je zgodnejše vključevanje uporabnikov v projekt, najbolje, da v času, ko sporočajo svoje zahteve, že izdelajo kriterije sprejemljivosti in sodelujejo pri pregledu in nadzoru zahtev.
- **Test naj bo izveden na realnih pogojih, ne uporabniških zahtevah**: obstajata dve »strani« testiranja, verifikacija in validacija. Verifikacija je testiranje na podlagi specifikacij in (funkcionalnih) zahtev, validacija je testiranje na podlagi operativnih pogojev iz realnosti. Slednje je potrebno, ker imajo zahteve lahko napake. V določenih primerih zahteve niso na voljo, na primer, ko je programsko opremo razvil zunanji ponudnik.
- **Razumevanje uporabnikov**: beta testiranje ni aktivnost, ki bi jo bilo mogoče uporabiti za vse uporabnike enako. Nekateri uporabniki nimajo motivacije, časa ali znanja, da bi lahko izvedli primerne teste. Ena od možnih rešitev je profiliranje končnih uporabnikov po njihovih motivacijah in zmožnostih.
- **Vključevanje uporabnikov**: nekatere organizacije izvajajo to vrsto testiranja z »nadomestnimi« uporabniki, ki prevzamejo vlogo končnih uporabnikov, čeprav to niso. Tveganje tega pristopa je, da bodo dejanski končni uporabniki ob splavitvi programske rešitve našli probleme, ki jih »nadomestni« uporabniki niso upoštevali. Tveganja je mogoče zmanjšati, tako da se z uporabniki izvede kratke ocenjevalne seje. Celovite ocenjevalne seje, ki vključujejo tudi podrobnejši pregled zahtev, so še boljše. Dobro je imeti tudi zasilni načrt za primer, če so odkrite kakšne napake. V primeru, da uporabniki ne želijo ali ne morejo sodelovati pri testiranju, je to situacijo potrebno omeniti kot tveganje v poročilu o statusu projekta.
- **Prilagajanje intenzivnosti testiranja relativnim tveganjem in sposobnostim uporabnikov**: vsak projekt ne zahteva izčrpnega testiranja. Za projekte, ki vključujejo velika denarna sredstva ali lahko vplivajo na varnost ljudi, je nujna izčrpna validacija. Uporabniki bodo lahko spraševali o koristnosti takšnega testiranja, vendar če se na to pogleda s perspektive (operativnih) tveganj, so čas in sredstva, namenjena

učinkovitemu testiranju, učinkovito porabljeni viri. Da bi testiranja prilagodili tveganjem, je potrebno narediti oceno tveganj, ki mora biti kvantificirana in dokumentirana. Ocena tveganja naj vključuje, kateri sistem in poslovna področja so najbolj izpostavljena tveganjem. To omogoča alokacijo testnih virov na način, s katerim bo imelo testiranje največji učinek.

- **Testiranje naj se načrtuje vnaprej**: obstajajo 3 ravni planiranja testiranja. **Strateški nivo**, ki specificira, kaj naj bo izvedeno v testiranju. Opiše višji nivo smeri in ciljev testiranja, ne opiše, »kako« bo testiranje izvedeno. **Taktičen ali logističen nivo** je tudi višjenivojski opis, ki opiše, kako bo testiranje izvedeno. To je dejansko projektni načrt za testiranje in je po navadi napisan, tako da se usmeri na eno fazo testiranja, kot je npr. integracijsko, sistemsko ali test sprejemljivosti. **Testni primer** ali **skripta za testni nivo** v podrobnostih opiše aktivnosti, ki bodo izvedene, pričakovane rezultate in procedure za izvedbo testiranja. Nekateri ljudje raje izberejo manj formalne in naključne pristope k testiranju, ki lahko najdejo napake, vendar bodo po navadi spregledali tiste, ki so najbolj skrite in pogosto najbolj težavne. Naslednji problem pomanjkanja načrtovanja je, da se nikoli ne ve, kdaj je test končan. Podroben testni načrt omogoči, da ena oseba naredi načrt, več oseb izvede testni scenarij. Podroben testni scenarij omogoči tudi, da se test izvede večkrat. Planiranje testiranja je v končni fazi potrebno prilagoditi tveganjem.
- **Pregled testnih načrtov**: testni načrti imajo lahko napake, tako kot kateri koli drug tip projektne dokumentacije. Načrti za test sprejemljivosti so lahko pregledani s strani ekipe za test sprejemljivosti, testne ekipe, projektnega vodje ali katere koli druge osebo, ki ima znanje o testiranju projekta.

1.5 Vrste testiranj programske opreme

Obstaja več vrst testiranj programske opreme. V nadaljevanju bom naštel samo tiste, ki so za pričujočo magistrsko delo najbolj zanimivi. Za nekaj vrst testiranj je težko reči, da gre npr. zgolj za testiranje po metodi črne skrinjice. Prav tako bom izpustil vrste testiranj, ki so povezane s specifično platformo npr. testiranje za mobilne telefone. Nekatere od (pod)vrst so bile podrobneje predstavljene že v prejšnjih poglavjih.

1.5.1 Funkcionalno testiranje

Testiranje je izvedeno na podlagi funkcionalnih zahtev oz. specifikacije. Testni scenariji so napisani na tak način, da se preveri, ali programska rešitev deluje tako, kot je predvideno. Po navadi se to testiranje izvaja ob koncu razvijalskega cikla, vendar se lahko, kar je veliko bolje, začne izvajati prej. Funkcionalno testiranje preverja, kako dobro sistem izvaja funkcije, kot so ukazi uporabnika, manipulacija podatkov, iskanje, poslovne procese, uporabniške zaslonske maske in integracijo. Preverja »zunanji« del funkcije in operacije, ki tečejo v ozadju, kot je npr. varnost in kako posodobitve vplivajo na delovanje sistema (TestingBrain, b.l.; Software Testing Fundamentals, 2011).

1.5.2 Testiranje zmogljivosti

Namen te vrste testiranja je ugotoviti odzivnosti in stabilnost sistema pod določeno obremenitvijo. Obstaja več podvrst tega testiranja (TestingBrain, b.l.; Software Testing Fundamentals, 2011):

- **Obremenitveno testiranje** (ang. *load testing*): s tem testiranjem se ugotavlja, kako se sistem obnaša ob povečani obremenitvi. Testiranje se običajno izvaja z veliko količino vhodnih podatkov, in sicer ob največji obremenitvi, ki jo sistem lahko prenese. Namen po navadi ni zlomiti sistema, temveč ga imeti na visoki frekvenci delovanja. Uporablja se specializirana orodja za avtomatizacijo ali skriptne programske jezike za izdelavo primernih podatkov.
- **Stresno testiranje** (ang. *stress testing*): namen je ugotoviti, kako se sistem obnaša ob povečani obremenitvi, ki je bodisi enaka ali večja od največje predvidene. Razlika z obremenitvenim testiranjem je, da se pri obremenitvenem testiranju skuša ugotoviti, kako se sistem obnaša ob predvideni obremenitvi, medtem ko se tu skuša ugotoviti, kako se sistem obnaša, ko je ta meja presežena.
- **Vzdržljivostno testiranje** (ang. *endurance testing*): je tip testiranja, pri katerem je sistem dlje časa pod večjo obremenitvijo.
- **Koničasto testiranje** (ang. *spike testing*): gre za vrsto testiranja, pri katerem se obremenitev nenadoma poveča.

1.5.3 Varnostno testiranje

Varnostno testiranje je vrsta testiranja, katere namen je ugotoviti, kako dobro je sistem in njegovi viri zaščiten pred nepooblaščenim dostopom, krajo podatkov, poškodbo kode, zlorabami in kako se je pred temi zlorabami sposoben sam zavarovati. Za to testiranje so potrebne sofisticirane tehnike (TestingBrain, b.l.; Software Testing Fundamentals, 2011).

Obstajajo 4 glavna področja varnostnega testiranja (TestingBrain, b.l.; Software Testing Fundamentals, 2011):

- **Omrežna varnost** (ang. *network security*): gre za iskanje varnostnih pomanjkljivosti v omrežni infrastrukturi (sistemski viri in varnostne politike).
- **Varnost programske opreme sistema**: ocenjevanje varnostnih pomanjkljivosti v različni programski opremi, kot so operacijski sistem, baza podatkov in druge programske rešitve, od katerih je odvisno programsko orodje.
- **Varnost na strani klienta**: zagotavljanje, da brskalnik ali drugo podobno orodje klienta ne more biti zlorabljeno.
- **Varnost na strani strežnika**: zagotavljanje, da so strežniška koda in njegove tehnologije dovolj robustne, da odbijejo morebitni napad.

Obstaja praktično neomejeno število poti, po katerih je možno vdreti v določen sistem. Varnostno testiranje ni edino ali najboljše merilo za to, kako varna je programska rešitev.

Zelo priporočeno je, da je del standardnega razvijalskega procesa, saj je na svetu ogromno ljudi, ki bi radi vdrli v kak sistem, ali zaradi denarnih koristi ali zgolj za zabavo (TestingBrain, b.l.; Software Testing Fundamentals, 2011).

1.5.4 Dimno testiranje

Dimno testiranje (ang. *smoke testing*) ali preizkus zaupanja je dobilo ime po preizkušanju kontejnerjev in cevi z dimom, s katerimi so preverili, ali so kje kakšna puščanja. Namen tega testiranja je ugotoviti, ali osnovne in kritične funkcije delujejo. Testira se večino pomembnih funkcionalnosti, vendar nobene podrobneje. Na podlagi tega testiranja se je mogoče odločiti, ali je graditev dovolj stabilna za nadaljnjo, bolj podrobno testiranje oz., ali je to sploh mogoče. S tem testiranjem se lahko ugotovi morebitne integracijske probleme ali velike hrošče in zagotavlja določeno raven zaupanja, da spremembe v kodi programske opreme niso bistveno vplivale na njeno delovanje. Sčasoma, ko programska oprema postaja zrelejša in se njene funkcionalnosti povečujejo, je potrebno razširiti nabor testnih scenarijev za dimno testiranje (TestingBrain, b.l.; Software Testing Fundamentals, 2011).

1.5.5 Raziskovalno testiranje

Raziskovalno testiranje (ang. *exploratory testing*) se izvaja brez kakršnega koli testnega načrta ali testnega scenarija. Običajno se izvaja z namenom, da se preizkuševalec seznani s sistemom (kot trening ali ga izvaja že izkušeni tester, ki odlično pozna sistem in opravlja npr. hitre teste) (TestingBrain, b.l.; Software Testing Fundamentals, 2011).

1.5.6 Regresijsko testiranje

Regresijsko testiranje (ang. *regression testing*) je vrsta testiranja, ki skuša zagotoviti, da spremembe v kodi programske opreme niso vplivale na njeno delovanje. Spremembe v kodi običajno nastanejo kot posledica dodajanja novih funkcionalnosti ali odpravljanja (starih) napak. Pri regresijskem testiranju po navadi izvajamo enak nabor testnih scenarijev, ki se po potrebi razširi z vsakim novim razvojnim ciklom glede na pomembnost funkcionalnosti. To testiranje tako poskuša zmanjšati 2 tveganji, in sicer lahko sprememba, ki naj bi odpravila hrošča, ne deluje ali nova sprememba prinese novega hrošča oz. ponovno omogoči starega. Regresijsko testiranje je mogoče opraviti na katerem koli testnem nivoju, običajno je najprimernejše sistemsko testiranje. Zaradi ponavljajočega izvajanja enakih testnih scenarijev in pomanjkanja virov je to eden izmed najprimernejših kandidatov za avtomatizacijo testiranja (TestingBrain, b.l.; Software Testing Fundamentals, 2011).

Regresijski pristopi se razlikujejo glede na njihov fokus (TestingBrain, b.l.; Software Testing Fundamentals, 2011):

- **Regresija hroščev**: ponovno testiranja hrošča, ki naj bi bil odpravljen.

- **Regresija starih popravkov**: ponovno testiranje nekaterih starih hroščev, da se preveri, ali so se vrnili (programska oprema je regresirala v slabo stanje).
- **Splošno funkcijsko testiranje**: splošno testiranja sistema, ki vključuje področja, ki so delovala prej, da se prepriča, ali zadnje spremembe niso negativno vplivale na delovanje (to je po navadi del AT).
- **Testiranje konverzije ali »prehoda«**: programska oprema je »prenešena« na novo platformo. Del regresijskega testnega cikla se izvede, da se preveri, ali je bil ta prenos uspešen. Tu je fokus po navadi usmerjen bolj na novo platformo kot na spremenjeno staro kodo.
- **Konfiguracijsko testiranje**: programska oprema teče na novi napravi ali operacijskem sistemu. To je podobno konverzijskemu testiranju, le da tu koda ni bila spremenjena, spremenjena je zunanja naprava, s pomočjo katere uporabnik uporablja programsko rešitev.
- **Lokalizacijsko testiranje**: programska rešitev je prilagojena za uporabo v različnih jezikih ali sledi različnim kulturnim pravilom. To testiranje vključuje nabor tako starih testnih scenarijev kot tudi novih, ki so prilagojeni jeziku.
- Dimno testiranje: podrobneje opisano v podpoglavju 1.5.4.

1.5.7 Testiranje uporabnosti

Testiranje uporabnosti (ang. *usabilitiy testing*) ali testiranje prijaznosti do uporabnika je vrsta testiranja, ki se izvaja s perspektive končnega uporabnika z namenom preverjanja, ali je sistem mogoče enostavno uporabljati. To je proces sodelovanja s končnimi uporabniki neposredno in posredno z namenom ugotavljanja, kako uporabnik vidi programsko rešitev in kako jo uporablja. Proces odkrije področja težavnosti za uporabnike kot področja, ki so narejena dobro. Cilj tovrstnega testiranja je omejitev in zmanjšanje težavnosti za končne uporabnike ter izboljšanje dobrih plati z namenom zagotavljanja največje možne uporabnosti. V idealnem primeru bi to testiranje vključevalo tako neposreden kot posreden (opazovanje obnašanja) uporabnikov odziv in ko je to mogoče tudi računalniško podprt odziv. Slednji je pogosto, če ne vedno, izpuščen iz tega procesa. Računalniško podprt odziv je lahko štoparica, ki meri čas, ki ga uporabnik porabi za upravljanje s pogovornimi okni in kako pogosto se pojavljajo določeni dogodki, kot so sporočila o napakah in sporočila za pomoč. To pogosto vključuje zanemarljive posege v kodo programske rešitve, ki lahko prinesejo velikansko povračilo na investicijo. Glede na ugotovljene izsledke bi testiranje uporabnosti, kot rezultat prineslo izboljšavo uporabnosti programske rešitve. Te spremembe bi morale biti neposredno povezane z realnostjo, hkrati bi se za to čim pogosteje delalo ustrezno dokumentacijo, ki bi olajšala podobne spremembe v prihodnosti (TestingBrain, b.l.; Software Testing Fundamentals, 2011).

1.5.8 Domensko testiranje

Domensko testiranje (ang. *domain testing*) je najbolj pogosto opisana vrsta testiranja. Glavna poanta tega testiranja je, da se veliko število testnih scenarijev razdeli v

podmnožice, ki so si na nek način podobne, nato se za testiranje uporabi predstavnika vsake od podmnožic (TestingBrain, b.l.; Software Testing Fundamentals, 2011).

1.5.9 Testiranje scenarijev

Testiranje na podlagi scenarijev je realistično, kredibilno in za poslovne deležnike motivacijsko, hkrati pomeni tudi izziv za programsko rešitev ter je enostavno za ocenjevanje s strani testerjev. Zagotavljajo tako kombinacijo funkcij in spremenljivk, ki ima nek pomen, kot umetno kombinacijo, ki je značilnejša za domensko ali kombinatorno testiranje. Takšne scenarije običajno priskrbijo končni, to so poslovni uporabniki programske rešitve. Kljub temu da je programska rešitev dobro testirana in interno testiranje ne pokaže nobenih (kritičnih) napak, lahko takšno testiranje odkrije napake, ki so bolj povezane s samim procesom uporabe (npr. določene funkcije ne delujejo, kot je bilo načrtovano) (TestingBrain, b.l.; Software Testing Fundamentals, 2011).

1.5.10 Alfa testiranje

To testiranje se običajno izvaja na mestu, kjer se razvija programska oprema. Večinoma ga izvajajo zaposleni v podjetju, lahko tudi določeni končni uporabniki. Razvijalci programske opreme opazujejo vedenje alfa testerjev in beležijo morebitne napake, ki jih skušajo odpraviti pred naslednjo fazo testiranja, to je pred beta testiranjem, ki bo opisano v naslednjem podpoglavju. Alfa testiranje se izvaja malo pred koncem razvojnega cikla programske rešitve. Manjše spremembe v programski kodi so v tej fazi še možne. Skupina, ki izvaja te teste, je običajno neodvisna od razvijalcev programske opreme, po navadi je to interni testni tim. Testiranje se izvaja preden je programska rešitev na voljo splošni javnosti oz. končnim uporabnikom. V prvi fazi alfa testiranja testiranje izvajajo zaposleni razvijalci v podjetju. Uporabljajo specializirana programska orodja za razhroščevanje programske kode ali strojno podprte razhroščevalnike. Cilj je, da hrošče najdejo čim hitreje. V naslednji fazi testiranje izvaja interni tim za zagotavljanje kakovosti oz. testerji, izvaja se v okolju, ki mora biti podobno končnemu.(TestingBrain, b.l.; Software Testing Fundamentals, 2011).

1.5.11 Beta testiranje

V grobem bi lahko trdili, da je **beta testiranje** enako uporabniškemu testu sprejemljivosti, zato podrobnejšega opisa beta testiranja v tem poglavju ne bo (TestingBrain, b.l.; Software Testing Fundamentals, 2011).

1.6 Modeli razvoja in testiranja programske opreme

V tem podpoglavju bom na kratko opisal najbolj pogoste modele razvoja programske opreme oz. njenega testiranja. V določenih primerih sem vire pridobil na spletnih podstraneh, v virih sem navedel le začetno stran (npr.: Gharhai, b.l. in ISTQB Exam Certification, b.l.).

1.6.1 Slapovni model

Gre za t. i. »klasični model«, ki je zelo pogosto razširjen zaradi »zgodovinskih« razlogov. To je linearni in zaporedni model razvoja in testiranja programske opreme z zelo majhno povratno zanko, ki se največkrat zgodi v primeru, ko so pri testiranju odkrite določene napake. Ta model ima različne cilje za vsako fazo cikla razvoja programske rešitve, zaradi česar še najbolj spominja na vodo v slapu (ang. *watefrall*), ki pada preko roba na nižje ležeča mesta, iz katerih se ne more več vrniti nazaj na višje lege. Podobno je pri tem modelu. Ko se enkrat ena faza zaključi, ni več vrnitve v prejšnjo fazo. Glavne prednosti tega modela so, da omogoča lažji nadzor s strani managementa in omogoča lažjo delitev na oddelke. Časovni načrt s skrajnimi roki je lahko postavljen za vsako fazo posebej, produkt se izdeluje skozi razvojni cikel kot po tekočem traku ter je teoretično lahko na voljo v predvidenem času. Slabost tega pristopa je, da ne omogoča veliko refleksije in revizije. Ob vstopu programske rešitve v testno fazo je zelo težko iti nazaj in popraviti napake, storjene v fazi koncepta (Barber, 2007; Narula, b.l.).

Faze tega modela so (Narula, b.l.):

- načrtovanje projekta,
- definiranje zahtev,
- načrtovanje,
- razvoj,
- integracija in testiranje,
- namestitev/potrditev,
- vzdrževanje.

Po mnenju Barberja (2007) se ta model prostovoljno zelo redko uporablja. Pogosto je »stranska posledica« kakšnega logističnega izziva, pri katerem testerji niso mogli uporabljati programske opreme ali sodelovati z razvijalci. Ta model je občasno primeren takrat, ko »upamo, da bo programska rešitev delovala«, kot je na primer nameščanje popravka na produkcijsko programsko rešitev.

1.6.2 V-model

V-model pomeni validacijo in verifikacijo. Je nadgradnja slapovnega modela, opisanega v prejšnjem podpoglavju. Vsaka faza v razvojnem ciklu je verificirana, preden se začne naslednja faza. Pri tem modelu se testiranje eksplicitno začne že na začetku, ko so napisane glavne zahteve. Testiranje torej zajema tudi pregled in ocenjevanje, t. i. statično testiranje. To pomaga odkrivati napake zgodaj v razvojnemu ciklu in zmanjšuje možnost njihovega pojavljanja kasneje. Vsak nivo razvojnega cikla ima ustrezen testni plan. V času, ko je določena faza v razvoju, se vzporedno pripravlja testni načrt, ki je namenjen testiranju te faze. V pripravi testnih načrtov so lahko definirani tudi pričakovani rezultati in vstopni in izhodni kriteriji. Razvoj programske rešitve poteka podobno kot pri slapovnem modelu, od

zgoraj navzdol, medtem ko testiranje poteka od spodaj navzgor. Tako se tvori črka »V« (Ghahrai, b.l.).

Narula (b.l.) navaja faze V modela:

- **Analiza zahtev**: v tej fazi so zbrane zahteve uporabnikov o sistemu, ki bo razvit v naslednjih fazah. Ta faza se vrti predvsem okoli definiranja tega, kako naj bi sistem deloval v idealnem primeru. Običajno se te podatke pridobiva z intervjuvanjem uporabnikov. Na podlagi tega se izdela dokument z zahtevami, ki po navadi vključuje funkcijske in fizične zahteve, če so le-te potrebne, tudi vmesnike, zmogljivost, podatke ter zahteve glede varnosti, kot jih pričakuje uporabnik. Uporabniki v tej fazi pripravijo testne scenarije za uporabniški test sprejemljivosti.
- **Načrt sistema**: sistemski inženirji analizirajo dokument z uporabniškimi zahtevami. Ugotovijo, kakšne so možnosti in tehnike za implementiranje uporabniških zahtev. V primeru, da katera od zahtev ni izvedljiva, se o tem obvesti uporabnika. V tej fazi se izdela dokument o specifikacijah sistema, ki služi kot nekakšen načrt sistema in vključuje splošne informacije o organizaciji sistema, strukturi menijev, podatkovni strukturi ipd. Vključuje lahko tudi primere poslovnih scenarijev uporabe, primere programskih oken, poročil za lažje predstavo in razumevanje. Dokument za sistemsko testiranje je pripravljen v tej fazi.
- **Visokonivojski načrt**: v tej fazi se naredi arhitektura sistema, ki vključuje seznam modulov, njihov kratek opis, njihove odnose na nivoju vmesnikov, odvisnost, tabele v podatkovni bazi, arhitekturne diagrame, podrobnosti o tehnologiji. V tej fazi se izvede integracijsko testiranje.
- **Spodnjenivojski načrt**: načrtovan sistem je razdeljen na manjše enote ali module. Vsak od njih je podrobneje razložen, tako da programerji lahko pričnejo s kodiranjem programske rešitve. Ta načrt vsebuje podrobne opise funkcionalne logike, psevdokodo, tabele v podatkovni bazi z vsemi elementi, kot so tipi in velikost, vse podrobnosti o vmesnikih, vse težave glede odvisnosti, sporočila o napakah uporabnikom, celoten pregled vhodnih in izhodnih podatkov za module. Testiranje enot je razvito v tej fazi.

Prednosti tega modela so (Ghahrai, b.l., ISTQB Exam Certification, b.l.):

- lahek in enostaven za uporabo,
- vsaka faza ima svoje specifične produkte ali rezultate,
- večja možnost uspeha kot pri slapovnem modelu, ker so testni načrti narejeni prej, s čimer se prihrani čas,
- proaktivno iskanje napak - te so najdene v zgodnejših fazah,
- dobro deluje pri manjših projektih, kjer so zahteve dobro razumljene,
- izogibanje pomikanju težav po »toku« navzdol,
- izraba virov je velika.

Slabosti tega modela so (Ghahrai, b.l., ISTQB Exam Certification, b.l.):

- zelo rigiden in nefleksibilen,
- programska oprema je razvita v fazi implementacije, tako da niso razviti nobeni zgodnji prototipi,
- v primeru sprememb na polovici razvojnega cikla je potrebno posodobiti dokument z zahtevami in testno dokumentacijo.

1.6.3 Inkrementalni model

V tem modelu se načrtovanje, kodiranje in testiranje programske rešitve dogaja inkrementalno, z vsako novo graditvijo se namreč doda nekaj novih funkcionalnosti. To se dogaja, dokler produkt ni končan in dokler niso izpolnjene vse zahteve. Le-te so razdeljene v različne graditve, tako da do končanja obstaja več razvojnih ciklov. Gre za zaporedje kombinacije več slapovnih modelov z iterativno filozofijo prototipiranja. Razvojni cikli so razdeljeni na manjše, lažje upravljane module. Vsak modul gre čez faze izdelave zahtev, načrtovanja, implementacije in testiranja. Delujoča verzija programske rešitev je izdelana med prvo graditvijo, vsak naslednji cikel nato doda nove funkcionalnosti že obstoječim. Vsaka nova graditev je predstavljena stranki, ko je končana. To omogoča delno uporabo nedokončane programske rešitve in se izgone daljšemu času razvoja. Model zmanjša »šok«, ki se pojavi ob uvedbi popolnoma novega sistema. Problemi pri tem modelu so, da mora biti vsaka nova graditev integrirana s prejšnjo, prav tako razdelitev produkta na manjše graditve ni lahka naloga. V primeru, da je premalo graditev in je vsaka polna napak, to lahko privede do »gradi in popravljaj« pristopa. V primeru, da je teh graditev preveč, je z novo graditvijo vsakič dodano premalo uporabnosti (Ghahrai, b.l.; ISTQB Exam Certification, b.l.).

Ghahrai (b.l.) trdi, da so prednosti tega modela naslednje:

- ustvari delujočo programsko rešitev hitro in zgodaj v programskem razvojnemu ciklu,
- bolj fleksibilen, saj so stroški zmanjšanja obsega in zahtev manjši,
- testiranje in razhroščevanje je lažje med manjšo iteracijo,
- lažje je upravljati s tveganji, ker so lahko odpravljeni med njihovo iteracijo,
- vsaka iteracija je lahko upravljan končni cilj.

Slabosti tega pristopa so (ISTQB Exam Certification, b.l.):

- potrebuje dobro planiranje in načrt,
- potrebuje jasno in popolno definicijo celotnega sistema, preden je ta lahko razbita na manjše dele,
- celotni stroški so večji kot pri slapovnem modelu.

Ta model se običajno uporablja (ISTQB Exam Certification, b.l.):

- takrat, ko so zahteve celotnega sistema jasno definirane in razumljene,
- glavne zahteve morajo biti definirane (kljub temu se določene lahko razvijejo s časom),

- produkt mora biti hitro na trgu,
- uporablja se nova tehnologija,
- viri s potrebnimi znanji niso dosegljivi,
- obstaja nekaj tveganih funkcionalnosti in ciljev.

1.6.4 Model hitrega razvoja programskih rešitev

Model hitrega razvoja programskih rešitev (ang. *rapid application development*, v nadaljevanju RAD) je vrsta inkrementalnega razvoja programskih rešitev, ki je bil omogočen z napredkom v razvijalskih orodjih. Komponente ali funkcije programske rešitve se razvijajo vzporedno, kot da bi šlo za manjše projekte. Razvojni cikli imajo natančno določen časovni okvir. Ko so končani, so »sestavljeni« v prototip. To omogoča, da uporabnik hitro dobi delno končan produkt, ki ga lahko vidi in preizkusi ter poda svoje mnenje. Na ta način se lahko odkrije tudi veliko napak. Po teoriji je preizkušanje »produkcijskega« sistema boljše kot prebiranje in analiziranje dokumentacije. Razvojni cikli, ki temeljijo na RAD modelu, so prinesli manj zavrnitev programskih rešitev, ko so te enkrat v produkciji. To vsebuje tudi tveganje, saj ta model pogosto prinese dramatična preseganja zastavljenih rokov in načrtovanih stroškov. Uporabnik ima občutek, da je večino njegovih zahtev mogoče enostavno spremeniti, zato lahko pride do začaranega kroga, ko uporabnik nenehno podaja svoje zahteve, razvijalci jih implementirajo, zato je najbolj primeren za uporabo, ko je potrebno izdelati produkt, ki ga je lahko modulirati v dveh ali treh mesecih. Potreben je tudi dovolj velik proračun, hkrati mora biti na voljo dovolj veliko število načrtovalcev. Poleg tega so potrebni tudi viri z dovolj poslovnega znanja. Ta model se po navadi uporablja, ko je potrebno sistem izdelati hitro (Ghahrai, b.l.; ISTQB Exam Certification, b.l.).

Faze tega razvojnega cikla so (ISTQB Exam Certification, b.l.):

- **Poslovno modeliranje**: identificiran je tok informacij med različnimi poslovnimi funkcijami.
- **Podatkovno modeliranje**: informacije, zbrane v poslovnem modeliranju, se uporabijo, da se definira podatkovne objekte, ki so potrebne za poslovni vidik.
- **Procesno modeliranje**: podatkovni objekti iz faze podatkovnega modeliranja so spremenjeni, da se doseže poslovni tok informacij, ki je potreben za dosego specifičnega poslovnega cilja.
- **Generiranje programske rešitve**: avtomatizirana orodja se uporabljajo, da spremenijo procesne modele v kodo in dejanski sistem.
- **Testiranje**: testiranje novih komponent in vseh vmesnikov.

Glavne prednosti tega modela so (ISTQB Exam Certification, b.l.):

- zmanjšan čas razvoja,
- povečana ponovna uporaba komponent,
- mogoč je hiter začetni pregled programske rešitve,

- vzpodbuja povratni odziv uporabnika,
- integracija od začetka zmanjša veliko integracijskih problemov.

Glavne slabosti tega modela so (ISTQB Exam Certification, b.l.):

- odvisen je od močnih timov in posameznikovih sposobnosti identificiranja poslovnih zahtev,
- samo sistemi, ki so lahko modulirani, so lahko narejeni s tem modelom,
- zahteva zelo izkušene razvijalce in načrtovalce,
- velika odvisnost od sposobnosti modeliranja,
- ni primeren za projekte z majhnimi proračunom, ker so stroški modeliranja in avtomatiziranega generiranja kode zelo visoki.

1.6.5 Agilni model

Tudi ta model je podoben inkrementalnemu pristopu. Programske rešitve se razvijajo v inkrementalnih in hitrih ciklih. Rezultat tega je več manjših inkrementalnih graditev, vsaka graditev gradi na prejšnji in se jo natančno testira, da se zagotavlja kakovost. Model se uporablja za programske rešitve, za katere je potreben hiter čas razvoja, in ko je potrebno vpeljati nove spremembe. Te so zaradi frekvence posameznih inkrementacij mogoče zelo hitro. Za implementacijo nove spremembe razvijalci običajno potrebujejo le nekaj dni ali ur, zato je spremembe mogoče tudi zelo hitro odstraniti. Pri tem modelu ni potrebnnega veliko načrtovanja, da se prične s projektom. Agilni pristop predvideva, da se zahteve uporabnikov v realnosti nenehno spreminjajo. O spremembah in funkcionalnostih se je mogoče pogovoriti z uporabniki in na podlagi njihovega odziva te spremembe odstraniti ali spremeniti. Razvijalci programske rešitve in poslovni deležniki podjetja imajo na voljo več svobode glede časa, kot če bi bil sistem razvit v bolj rigidnem zaporednem modelu. Pomembne odločitve se lahko pusti za kasnejše faze projekta, ko so na voljo novi podatki, kar pomeni, da se projekt lahko nadaljuje brez nevarnosti, da bi se nenadoma ustavil. Svoboda, ki jo omogoča ta model, je zelo pomembna (Ghahrai, b.l.; ISTQB Exam Certification, b.l.).

Ta model bolj ali manj odstrani vnaprej predviden tok testnega cikla, saj je mogoče med različnimi testnimi aktivnostmi menjati, kadar koli je to v korist projekta. Na primer tester, ki med preverjanjem rezultatov njegovega testiranja ugotovi, da ima testni scenarij pomanjkljivosti. Ta tester se tako »vrne« v fazo načrtovanja in ustvarjanja testnih scenarijev. V primeru slapovnega modela ali iterativnega toka bi moral počakati na naslednji testni cikel. Agilni pristop se lahko uporabi kot samostojni model ali komplementarni sistem obstoječemu. Tester lahko v iterativnem modelu določeno obdobje testira agilno, skupaj z razvijalcem, ko iščeta in odpravljata probleme v določeni funkcionalnosti. Pristop je bolj pogost, kot se predvideva, saj pogosto poteka v glavi testerja, ne glede na izbrani model testiranja, po katerem testira programsko opremo. Ta pristop ni preveč priljubljen med managementom in poslovnimi deležniki, saj jim ne daje zaupanja, da testerji testirajo dovolj premišljeno in organizirano (Barber, 2007).

1.6.6 Iterativni model

Ta model ne skuša začeti s celotno specifikacijo zahtev. Namesto tega se razvoj začne z implementiranjem zgolj dela zahtev. Razviti del programske opreme se nato pregleda, na podlagi tega se lahko identificira naslednje zahteve. Proces se nato ponavlja, z vsakim ciklom se naredi nova verzija programske rešitev. Ta proces sestoji iz faze zahtev, v kateri se zbere in analizira zahteve. Cilj iteracije je v končni fazi izdelati kompletno in končno specifikacijo zahtev. V naslednji fazi, fazi načrtovanja, je izdelan načrt programske rešitve, ki naj bi izpolnila zahteve. To je lahko nov načrt ali nadgradnja obstoječega. V fazi impetementacije in testiranja poteka pisanje kode, integracija in testiranje. V zadnji fazi, fazi ocenjevanja, se ovrednoti programsko rešitev, zahteve in predlagane spremembe glede na zahteve. Za vsak cikel modela je potrebna odločitev, ali naj se izdelana programska rešitev uniči ali obdrži in ali naj ta služi kot začetna točka (občasno se to opisuje kot inkrementalno prototipiranje) za naslednji cikel. Na koncu se doseže točka, ko so zahteve izpolnjene in je programska rešitev lahko izdana ali je potrebno pričeti z razvojem od začetka. Ključ za uspeh tega modela je strogo ocenjevanje zahtev in verifikacija ter testiranje vsake verzije. Prve tri faze izhajajo iz zaporednega V-modela ali slapovnega modela. Vsak cikel modela zahteva testiranje na vseh nivojih. Tako kot se programska rešitev razvija skozi cikle, tako se mora ponavljati in razširiti njeno testiranje (Ghahrai, b.l.;).

Iteracije testiranja se lahko zgodijo pred prvo graditvijo programske rešitve. Lahko jih je več med eno graditvijo. Drugače kot iteracija kodiranja, se iteracija testiranja lahko kadar koli prekliče in vrne v »raziskovalni način«. Četudi se iteracija programiranja lahko prekine in ponovno kadar koli začne, lahko to ogrozi celoten projekt. V-model in spiralni model so vsi »derivati« tega modela. Ta model dobro deluje na projektih, kjer se programska oprema razvija v vnaprej planiranih, predvidenih inkrementih ali kjer se razvija v tako hitrih in nepredvidljivih ciklih, da je neproduktivno, da testerji načrtujejo na podlagi načrtovanih izdaj programske opreme (Barber, 2007).

Prednosti tega modela so (Ghahrai, b.l.; ISTQB Exam Certification, b.l.):

- delujoča programska rešitev je narejena hitro in zgodaj v ciklu razvoja,
- po tem modelu se na začetku lahko naredi le grob načrt programske rešitve, preden se začne dejansko kodiranje.
- sčasoma se programsko rešitev izboljšuje s spreminjanjem in dodajanjem novih funkcionalnosti glede na zahteve in pridobljene informacije,
- lažje testiranje in razhroščevanje pri manjših iteracijah,
- produkt se gradi in izboljšuje po korakih, zato se lahko napake lovi pri zgodnjih korakih, s čimer se zmanjša možnost prenašanja napak v kasnejše korake,
- lažje je odkriti in nadzorovati posamezna tveganja zaradi manjših iteracij,
- vsaka iteracija je lažje nadzorljiv mejnik,
- manj časa je porabljenega za dokumentiranje in več za načrtovanje,

- s predstavljanjem skic in načrtov je lažje dobiti bolj relevanten odziv uporabnikov.

Slabosti tega modela so (Ghahrai, b.l.; ISTQB Exam Certification, b.l.):

- vsaka faza iteracije je zaprta celota, ki se ne prekriva z drugimi fazami,
- lahko nastanejo problemi s sistemsko arhitekturo, saj ta ni znana na začetku, kar lahko predstavlja ogromne stroške.

1.6.7 Spiralni model

Spiralni model je podoben inkrementalnem modelu, večji poudarek je na analizi tveganj. Začne se s »prehodom« skozi standardni slapovni model, pri čimer se uporabi del zahtev, da se razvije robusten prototip. Po začetni fazi se cikel ponovno »zavrti«, pri čemer se doda več funkcionalnosti in naredi nov prototip. To se nadaljuje, tako da prototip z novo iteracijo postaja vedno večji in večji. Teorija pravi, da je zbirka zahtev hierarhična po naravi, dodatna funkcionalnost je grajena na prejšnjih poskusih. Ta sistem je odlična izbira za sisteme, kjer je celoten problem dobro definiran že na začetku, to so na primer programske rešitve za modeliranje in simulacije. Poslovne programske rešitve, ki temeljijo na bazah podatkov, ne izkoristijo te prednosti, saj je večina funkcij v bazi podatkov neodvisna druga od druge, čeprav lahko uporabljajo skupne podatke. Ta model poteka v 4 fazah. V prvi fazi, fazi planiranja, se zbere zahteve in izdela specifikacije. V fazi analize tveganj se začne proces identificiranja tveganj in alternativnih rešitev. Prototip je izdelan na koncu te faze. V kolikor je odkrito kakšno tveganje med analizo tveganj, se predlaga alternativa rešitev, ki se jo tudi implementira. V fazi »inženiringa« se programsko rešitev razvije. Na koncu je na vrsti testiranje. V fazi evaluacije stranka oceni izdelek do te faze, nato sledi naslednji cikel spirale (Ghahrai, b.l.; ISTQB Exam Certification, b.l.).

Glavne prednosti tega modela so (ISTQB Exam Certification, b.l.):

- velika količina analize tveganj, s čimer je povečana možnost izogibanja le-tem,
- dobra izbira za velike in kritične projekte,
- velik nadzor nad dokumentacijo in potrjevanjem,
- dodatne funkcionalnosti so lahko dodane kasneje,
- programska rešitev je izdelana zgodaj v ciklu razvoja.

Glavne slabosti tega modela so (ISTQB Exam Certification, b.l.):

- lahko gre za drag model,
- analiza tveganj zahteva zelo specifična znanja,
- uspešnost projekta je zelo odvisna od faze analize tveganj.
- ne deluje v redu za manjše projekte.

2 AVTOMATIZACIJA TESTIRANJA PROGRAMSKE OPREME

2.1 Opredelitev avtomatizacije testiranja programske opreme

ISlovar (b.l.) opredeljuje AT kot testiranje, pri katerem se rezultati ugotavljajo z različnimi pripomočki ali programi.

ISTQB (2014) opredeljuje AT kot eno ali več od naslednjih nalog, ki so uporaba posebnih programskih orodij za nadzor in vzpostavljanje predpogojev za testiranje, izvajanje testov in primerjavo dejanskih rezultatov s pričakovanimi. Glavni poudarek je, da je programska oprema, ki se uporablja za testiranje, ločena od sistema, ki je testiran. AT pomaga pri konsistentnem in ponavljajočem poganjanju večjega števila testnih primerov, vendar je to več kot zgolj mehanizem za poganjanje testne zbirke brez interakcije človeka. Je proces načrtovanja testne opreme, ki vključuje programsko opremo, dokumentacijo, testne primere, testno okolje in podatke.

ISTQB (2014) navaja, da obstajajo naslednji pristopi za testiranje sistema, ki so:

- testiranje z javnimi vmesniki do razredov, modulov ali knjižnic,
- testiranje preko uporabniških vmesnikov sistema,
- testiranje preko omrežnih protokolov.

Seetaram (2011, 17. september) trdi, da je za testiranje večjih in kompleksnejših sistemov nujno uporabiti AT, saj ročno testiranje težko »pokrije« celoten sistem. Nad testnimi ekipami se tako vrši vedno večji pritisk, da je potrebno testirati čim več v čim krajšem času. Za dobro upravljanje kvalitete je potrebno celovito testiranje sistema. AT je tako postal sestavni del razvojnega cikla programske opreme.

2.2 Pristopi k avtomatiziranju testiranja programske opreme

Obstaja več pristopov ali metod k avtomatizaciji testiranja. Določeni pristopi so primerni samo za določene vrste testiranja npr. testiranje uporabniškega vmesnika. Na kratko bom opisal najbolj pogoste.

2.2.1 »Posnemi in predvajaj«

Gre za eno izmed najstarejših metod testiranja, saj je v uporabi že desetletja. Zaradi tega je pogosto opisana kot zastarela in nezdružljiva z modernimi metodami razvoja programske opreme, kot sta agilni pristop in testno usmerjen razvoj (ang. *test driven development*). Večina sodobnih metodologij za razvoj programske opreme zahteva pisanje testnih primerov oz. skript, ki temeljijo na poslovni logiki. To včasih ni izvedljivo, saj je npr. poslovna logika lahko »prepletena« z uporabniškim vmesnikom. Ob tem je lahko nerazumljiva poslovnim uporabnikom ali je neizvedljiva s tehničnega vidika oz. je z vidika porabljenega časa in denarja preveč zahtevna. Ta metoda temelji na dveh korakih. Prvi korak je »posnemi«, v katerem uporabnik z določeno komponento »posname« svoje aktivnosti v sistemu, ki jih nato v naslednjem koraku, »predvajaj«, programsko orodje za

avtomatizacijo testiranja izvede samodejno in nato dobljene rezultate primerja s pričakovanimi oz. prejšnjimi (Vaněk, 2010).

Glavne prednosti te metode so (Hayes, b.l.; Vaněk, 2010):

- zahteva najmanj časa od vseh metod za izobraževanje in namestitev - krivulja učenja je relativno kratka tudi za uporabnike, ki so tehnično manj vešči.
- testi ne rabijo biti izdelani vnaprej, kar pomeni, da se lahko definirajo ob uporabi, s čimer se izkušenim testerjem omogoča, da prispevajo k testnemu procesu na »ad hoc« osnovi.
- omogoča shranjevanje podrobne zgodovine izvedbe za revizije, saj je v primeru napak jasno, pri katerem koraku je prišlo do napake in kako priti do tega koraka.
- je odlična za starejše sisteme in ko je ročno pisanje skript preveč zahtevno s stroškovnega in časovnega vidika.
- v primerih, ko ni na voljo dovolj usposobljenih ljudi.
- lahko se uporabi za testiranje sistema, ne glede na programski jezik, ki se uporablja za uporabniški vmesnik.

Pomanjkljivosti te metode so (Hayes, b.l.; Vaněk, 2010):

- Testi morajo biti ročno »posneti«, kar je praktično enako, kot če bi se izvajali ročno, prihranki časa, razen v primeru večih ponovitev, niso tako veliki.
- Programska rešitev, ki je testirana, mora biti stabilna in mora že obstajati. Tako ni na voljo veliko priložnosti za zgodnje ugotavljanje napak. Vsak test, ki odkrije napako, bo moral biti verjetno ponovno izveden, da se ohrani pravilen rezultat.
- Možnost ponavljanja in izpuščanja je visoka. Vsak tester se bo sam odločil, kaj je najprimerneje testirati, kar lahko pomeni, da se bodo določene funkcionalnosti ponavljale, nekatere bodo povsem izpuščene. Potrebno je načrtovati sledljivost testnih skript, da se ve, kaj je bilo testirano in kaj ne.
- Testi morajo biti kombinirani. Potrebno je razmisliti o konvencijah poimenovanja in standardih za pisanje skript, da se po pomoti skripte ne »povozi«, ali komplikacij, ko se jih skuša izvesti kot zbirko.
- Odsotnost vzdrževanja, ki vpliva na uporabnost skript, tudi če so izvedene večkrat. Vhodni in izhodni podatki so zakodirani v skripte, tako da lahko tudi manjši posegi v programski rešitvi pripeljejo do sprememb velikega števila skript.
- Kratko »življenje« skript. Kljub temu da po navadi ni potrebno poznati nobenega programskega jezika, je to znanje potrebno ob izvajanju sprememb skript, zaradi česar je včasih preprosteje skripto ponovno »posneti«.
- Vse odločitve, kaj se zgodi v naslednjem koraku, se narejene s strani izdelovalcev skript. Ta odsotnost logike za sprejemanje odločitev v skriptah lahko pomeni, da bodo vse naslednje ponovitve testov padle zaradi neprimernega konteksta.

2.2.2 Podatkovno usmerjen (razvoj) pristop

Razlika med **podatkovno usmerjenim** (ang. *data driven*) **testiranjem** in »posnemi in predvajaj« je, da so pri drugem vhodni in izhodni podatki fiksni, medtem ko se pri pristopu, ki je opisan v nadaljevanju, spreminjajo. To je doseženo, tako da se test najprej izvede ročno, nato se zamenja ulovljene vhodne in pričakovane izhodne podatke s spremenljivkami, katerim ustrezajoči podatki so shranjeni v podatkovni zbirki, ki je neodvisna od testne skripte. Sekvenca akcije ostane fiksna in shranjena v testni skripti. Ta pristop podpira večina orodij, ki podpirajo katerega od skriptnih jezikov in uporabo variabilnih podatkov, vendar lahko niso mogoča s »posnemi in predvajaj« orodji. Da so testni primeri definirani kot podatkovni zapisi, ki so obdelani z zunanjimi skriptami, morajo biti poznani vsi podatkovni elementi, povezani z vsakim procesom. To naj bi zagotavljal načrt programske rešitve. Priporočljivo je tudi, da se sledi naslednji strukturi. Podatkovno usmerjena skripta naj sledi principu ena skripta en proces. Vsaka skripta je vezana na eno procesno sekvenco, vendar bo podpirala več testnih primerov. Sekvenca korakov, ki vnese ali procesira podatke, vključuje veliko testnih scenarijev, ki so povezani na posamezni element podatkov ali korakov sekvenc. Izbrati je potrebno zaporedje korakov, ki zahtevajo konsistentno podatkovno zbirko za vsako iteracijo, in skripte ustrezno poimenovati. Naslednji pomembni korak strukture je, da je en podatkovni zapis en testni primer. Identifikator testnega primera naj bo shranjen v podatkovnem zapisu. To omogoča, da ena skripta procesira več testnih scenarijev in ustrezno zapisuje rezultate posameznega. Naslednji pomembni dejavnik je, da je programska rešitev kar precej intenzivna v smislu obdelave podatkov. Enaki koraki so izvedeni večkrat z različnimi podatki. Zadnji dejavnik je konsistentno obnašanje. Enaki koraki se izvajajo z majhnimi razlikami med njimi. V primeru, da vrednost enega polja lahko privede do popolnoma druge procesne poti, se zahtevnost logike, da se procesira en testni primer, poveča eksponentno (Hayes, b.l.).

Hayes (b.l.). navaja glavne prednosti tega pristopa kot:

- Testni primeri ali scenariji so lahko izdelani vnaprej, še preden je programska rešitev končana ali dovolj stabilna. To vključuje tako pripravo vhodnih kot izhodnih podatkov. Samo končna skripta je lahko razvita do konca, tako razvita je tudi programska rešitev.
- Fleksibilnost pri ustvarjanju testnih scenarijev. Vhodni in izhodni podatki so lahko shranjeni v različnih oblikah, kot preglednice, dokumenti, podatkovne baze ali druga poznana tehnologija, da so kasneje lahko uporabljeni s skripto. Poznavanje in včasih tudi uporaba za izdelavo testnih scenarijev ni nujno potrebna.
- Ena skripta se lahko uporabi za veliko testnih scenarijev, le-ti so lahko dodani kasneje, brez spreminjanja skripte.
- S tem, ko ni potrebno ponavljati sekvenc akcij ali logike, so znižani stroški vzdrževanja za vsak testni primer. V primeru, da se koraki za določeno akcijo spremenijo (npr. dodajanje uporabniškega računa), je to potrebno spremeniti samo na enem mestu.

Hayes (b.l.) kot glavne slabosti tega pristopa našteje:

- Zahteva tehnično znanje. Da skripte lahko procesirajo spreminjajočo zbirko podatkov, mora biti vsaj eden od testerjev dovolj usposobljen za uporabo testnega orodja in poznati koncept variabilnih vrednosti, kako implementirati zunanje datoteke in logiko programiranja, kot so zanke in »če« stavki.
- Podobno kot v prejšnjem primeru bodo podatki testnih scenarijev zahtevali nekoga z znanjem v kreiranju in uporabljanju testnih datotek. Velika količina podatkovnih elementov v vsakem testnem scenariju lahko pripelje do velike količine podatkov, ki jo je težko uporabljati, kar lahko zahteva dobro poznavanje ustvarjanja in manipuliranja makrojev, preglednic, podatkovnih obrazcev, dokumentnih procesov in prenašanja podatkov v datoteke, ki so kompatibilne s testnim orodjem.

2.2.3 Testno usmerjen razvoj

Testno usmerjen razvoj (ang. *test driven development*) je povezan z nastankom in razmahom popularnosti agilnih metod razvoja programskih rešitev, konceptom ekstremnega programiranja ter zahtevami trga po čim hitrejši splovitvi izdelka in njegovem posodabljanju. Cilj tega pristopa je imeti popolno testno »pokritje« sistema. Vsaka vrstica kode in zahtevane sistemske funkcionalnosti morajo biti pokrite s testnim scenarijem, le-ti so napisani, preden se začne razvoj programske opreme. To naj bi programerje spodbujalu k temu, da ostanejo bolj fokusirani na svoje naloge. S tem pristopom sta povezana dva principa. Vsaka vrstica kode je lahko napisana samo, če je pokrita z ustreznim avtomatskim testom, pri čemer ne sme prihajati do podvajanja. Ta pristop lahko poveča zaupanje, a hkrati ustvari občutek lažne varnosti. Pred začetkom uporabe je potrebno dobro premisliti, ali je to pravi pristop k projektu ali organizaciji, in pridobiti podporo vseh udeležencev. S podedovanimi sistemi in programskimi rešitvami, ki nimajo ogrodja za test enot, bo ta pristop težko uporabiti. Lahko je najhitrejša in najcenejša izbira za kvaliteten produkt pri novem projektu, ki ima dober načrt (Hill, 2015, 23. februar; Vaněk, 2010).

Glavne prednosti tega pristopa so (Hill, S., 2015, 23. februar; Levison, M., 2008, 14. oktober):

- Pisanje testnih scenarijev najprej zahteva, da se dobro ve, kaj želimo od programske rešitve.
- Krajši cikel povratnih informacij.
- Ustvari podrobnejšo specifikacijo.
- Lahko pripelje do enostavne, elegantne in modularne kode. Koda je napisana zgolj glede na zahteve testov. Ta pristop sili razvijalce, da pišejo manjše kodne razrede za posamezno stvar.
- Lahko omogoči, da so problemi odkriti prej. Manj časa se porabi v razhroščevalniku kode.
- Testni scenariji so lahko kot »živa« dokumentacija in omogočijo lažje razumevanje kode.
- Lahko pospeši razvoj na dolgi rok.
- Vzdrževanje in refaktoriranje kode je lažje.

- Razvijalcem omogoča, da razmišljajo z vidika uporabnika.
- Načrt programske rešitve se spreminja glede na razumevanje projekta.
- S tem, ko nas sili k razmisleku glede namena in specifikacije kode, izboljša kvaliteto in zmanjša hrošče, poenostavi kode (veliko problemov izvira iz kompleksnosti) in zagotavlja, da spremembe in nova koda ne »zlomijo« pričakovanj obstoječe kode.

Glavne slabosti tega pristopa so (Hill, S., 2015, 23. februar; Levison, M., 2008, 14. oktober):

- Te metode se je težko naučiti brez zunanje pomoči. V prvih nekaj mesecih je pričakovan upad produktivnosti.
- Usmerjenost na čim enostavnejši trenutni načrt brez razmišljanja vnaprej lahko pripelje do veliko refaktoriranja kode kasneje.
- Težko je narediti testne scenarije samo za najbolj nujne stvari in se izogibati nepotrebnim stvarem.
- Vzdrževanje nabora testnih scenarijev zahteva čas in trud in mora biti ustrezno načrtovano.
- Če se načrt hitro spreminja, potem je potrebno spreminjati testne scenarije . Veliko časa je tako lahko porabljenega za pisanje testnih scenarijev za funkcionalnosti, ki se jih hitro znebimo.

2.2.4 Razvoj, usmerjen na podlagi ključne besede

V tem podpoglavju opisan **pristop, usmerjen na podlagi ključne besede** (ang. *keyword driven development*), je izpeljan s pomočjo prej opisanega podatkovno usmerjenega pristopa. Pristop zahteva razvoj ključnih besed in podatkovnih tabel, le-te so neodvisne od programskega orodja za avtomatizacijo, ki jih izvaja, in skript za avtomatizacijo, ki »poganjajo« testni sistem. Gre za zaporedje operacij, ki so poimenovane s ključnimi besedami in simulirajo dejanja uporabnika. Pristop je sestavljen iz dveh glavnih faz, faze načrtovanja in faze implementacije. V prvi fazi se ustvari tabele, ki vsebujejo akcije (ključne besede), vhodne podatke in pričakovane rezultate, vse v enem zapisu. Tabela vsebuje tudi vse dodatne informacije, ki so potrebne za vhodne podatke sistema in preizkusne rezultate, ki služijo za preverjanje statusa komponent ali sistema v celoti. V naslednji fazi, fazi implementacije, se ustvari skripte, ki interpretirajo tabele s ključnimi besedami, vnesejo vhodne podatke in preverijo pričakovane rezultate. Kritični dejavniki uspeha te metode so, da je potrebno popolnoma ločiti razvoj testnih scenarijev (polnjenje tabel s ključnimi besedami) in ogrodje za avtomatizacijo. Oboje zahteva različna znanja. Pomembneje je, da so testni scenariji neodvisni od spodaj ležeče tehnologije ogrodja ali sistema in da se prepreči vpliv testnih scenarijev na integracijo ogrodja ali sistema. Naslednji pomembni dejavnik je, da morajo imeti testni scenariji jasen in diferenciran obseg in da se od njega ne odmikajo. Bolj uporabno je ustvariti več manjših testnih scenarijev, ki pokrivajo samo eno funkcionalnost. Zadnji kritični dejavnik je, da mora biti izbran ustrezen nivo abstrakcije. V določenih primerih je pomembneje napisati testne

scenarije za višje ležeče nivoje, kot je poslovna logika, v določenih primerih za nižje ležeče, kot je uporabniški vmesnik. Pomembno je, da testno ogrodje omogoča dovolj veliko fleksibilnost. Po velikih začetnih investicijah ta pristop privede do zelo robustnih testnih scenarijev, ki se lahko uporabijo tudi za regresijske teste, ko so dodane nove funkcionalnosti ali po spremembah v uporabniškem vmesniku (BSD Mag, 2014, 6. november; Vaněk, 2010).

Glavne prednosti te metode so (Vaněk, 2010; Zylberman, A.& Shotten, A., b.l.):

- Podatki v tabelah s ključnimi besedami so lahko avtomatično pretvorjeni v navaden tekst, ki je nato lahko sestavni del specifikacije programske opreme za poslovne uporabnike.
- Ločevanje vlog pri ustvarjanju testnih scenarijev. Po tem, ko testerji ustvarijo slovar ključnih besed, lahko začnejo izdelovati načrte testiranj brez kakršnega koli poznavanja testnega ogrodja, skriptnega jezika in do določne mere tudi podrobnosti sistema.
- Testerji lahko pričnejo z ustvarjanjem testnih scenarijev in tabel brez razvite programske rešitve, če so na voljo zahteve na višjih nivojih.
- Izboljša možnosti, da bo ista skripta uporabljena večkrat. V določenih primerih je dovolj zgolj ena skripta.
- S podrobnejšim opisom, kako izvršiti ključne besede, lahko te teste izvaja praktično vsak tester, metoda je uporabna tako za ročno kot avtomatizirano testiranje.
- Omogoča enostaven pregled nad vsemi operacijami v obliki tabel ali preglednic. Vsi objekti, povezani z vmesniki, so shranjeni v zunanjih datotekah, kar zmanjšuje možnost izgube podatkov.
- Testne scenarije je mogoče enostavno dodajati, spreminjati ali brisati.
- V primeru zamenjave testnega orodja ne zahteva veliko sprememb v kodi.

Glavne slabosti te metode so (Vaněk, 2010; Zylberman et al., b.l.):

- Začetna faza porabi veliko časa, še zlasti, če se zahteve pogosto spreminjajo.
- Tester bo potreboval znanje skriptnega jezika orodja za avtomatizacijo, če želi ustvariti funkcije in zmogljivosti po meri, ki jih potrebuje za testiranje določene programske rešitve.
- Vsaj v začetku lahko zahteva učenje posebnih oblik in ključnih besed za ustvarjanje zmogljivosti po meri. To učenje je potrebno vzeti v zakup ob načrtovanju testnih aktivnosti.

2.2.5 Razvoj, usmerjen na podlagi obnašanja

Ta pristop, ki se imenuje **razvoj, usmerjen na podlagi obnašanja** (ang. *behavior driven development),* je močno povezan s testno usmerjenim razvojem in nekako predstavlja drugo generacijo razvojnih in testnih metodologij. To besedno zvezo je skoval Dan North. Pristop je usmerjen bolj na »obnašanje« programske opreme kot k njeni tehnični implementaciji. Testni scenariji so napisani v naravnem jeziku, ki je bolj razumljiv tudi

ljudem, ki nimajo tehničnega znanja. Poslovnim analitikom in managementu omogoča, da aktivno sodeluje v procesu izdelovanja testnih scenarijev in procesu ocenjevanja. Vzpodbuja razvijalce programske opreme, da uporabljajo njihov naravni jezik v povezavi z domensko usmerjenim jezikom. Na ta način se zmanjšajo razlike pri »prevajanju« med tehničnim jezikom (na podlagi katerega je napisana koda) in domenskim jezikom (uporabljajo ga poslovni uporabniki, uporabniki, projektni management, itd.). Prvi princip tega pristopa je vzpostavitev (in dokumentiranje) ciljev različnih deležnikov, da se vzpostavi vizija sistemske implementacije. V naslednjem koraku je potrebno narediti osnutek funkcionalnosti na podlagi teh ciljev. Nato je potrebno vključiti vse deležnike v celoten razvoj sistema. Temu sledi korak oz. princip, ki na podlagi primerov opiše delovanje sistema ali zbirke komponent. Na podlagi teh primerov je potrebno izdelati avtomatizirane teste, da se zagotovi hiter povratni odziv in regresijsko testiranje. Naslednji princip govori o uporabi pravilnih besed za lažje razumevanje testnih scenarijev. Tako je priporočljivo uporabljati »moral bi« in »ne bi smel« kot »testiraj«, kar privede do boljšega razumevanja pričakovanega obnašanja. Podobno govori naslednji princip o uporabi besede »zagotavlja« za opis neposredne odgovornosti sistema. Kritično je razlikovanje med zaželenimi rezultati komponent (ali njihove kode) in stranskega učinka (od druge komponente ali njene kode). Zadnji princip govori o uporabljanju prototipnih komponent kot nadomestkov za dele sistema, ki še niso bili razviti. Opisani pristop je primeren predvsem za projekte, ki se začnejo bolj ali manj od začetka. Pristop je še bolj usmerjen na poslovni vidik kot testno usmerjen pristop, zato je pomembo, da so vsi deležniki, udeleženi v procesu, pripravljeni ta pristop sprejeti. Za poslovno orientirane uporabnike in deležnike organizacije ta metodologija prinaša boljši pregled v celoten cikel razvoja in jim omogoča, da v njem sodelujejo in ga upravljajo (North, b.l.; Madaan, 2015, 10. marec; Vaněk, 2010).

Glavne prednosti tega pristopa so po mnenju Madaan (2015, 10. marec) naslednji:

- Izboljšano sodelovanje. Vzpodbuja produktne skupine, poslovne analitike, testne time in razvijalce, da skušajo najti soglasje, in vzpostavlja platformo za zmanjševanje razlik zaradi različnih pogledov. Tako zagotavlja, da imajo vsi udeleženci enaka pričakovanja. To privede do dobrih kriterijev sprejemljivosti.
- Omogoča enostavno recenziranje in pridobivanje povrtanih informacij. Nobene razvijalske sposobnosti niso potrebe za ustvarjanje testnih scenarijev, saj so ti napisani v razumljivem jeziku. Poslovni analitik lahko aktivno sodeluje pri pregledovanju avtomatiziranih testnih scenarijev in daje povratne informacije o tem, kako jih izboljšati.
- Pomaga k večjemu osredotočenju na potrebe uporabnikov in pričakovanem obnašanju sistema kot prevelikemu osredotočanju na implementacijo testiranja.
- Večji donosi na investicije. Ugotovljeno je bilo, da obnašanje sistema ostane dlje časa kot tehnična implementacija. Tako je na podlagi tega pristopa lažje spremeniti testne scenarije, s čimer so večje tudi povratne koristi. Na voljo je tudi veliko odprtokodnih rešitev.

- Zaradi lažje večkratne uporabe istega testnega koraka omogoča lažje vzdrževanje testnih scenarijev, kar zmanjšuje stroške njihovega vzdrževanja.

Po mnenju Madaan (2015, 10. marec) so glavne slabosti tega pristopa naslednje:

- Potrebna znanja za uporabo tega pristopa morda niso na voljo v takšnem obsegu kot za ostale pristope.
- Določeni testni scenariji, ki jih napišejo testerji, poslovni uporabniki, uporabniki in poslovni analitiki niso dovolj dobri za avtomatizacijo. Lahko je potrebno dodatno delo, da se jih spremeni, tako da so primerni za avtomatizacijo.
- Podporna skupnost je zaradi mladosti tega pristopa še relativno mlada.

2.3 Pogoji za uvedbo avtomatizacije testiranja programske opreme

Pred uvedbo avtomatizacije testiranja morajo biti izpolnjeni določeni pogoji. Ti so (AbeachA, b.l.; Brodie, b.l.; F(x) Solutions, 2013, 18. maj; Schaefer, b.l.; Warwick, b.l.):

- **Jasna pričakovanja**: zelo pomembno je, da vemo, kaj in zakaj želimo avtomatizirati ter kakšna je poslovna vrednost rezultata. Ta pričakovanja morajo biti jasno predstavljena vsem deležnikom v projektu uvedbe avtomatizacije testiranja, še zlasti vrhnjemu managementu. Postaviti je potrebno določene (časovne) cilje in stalno meriti njihov napredek.
- **Predanost (vrhnjega) managementa**: management se mora zavedati, da je potreben določen začetni vložek, tako finančni kot časovni, ki se na dolgi rok načeloma obrestuje.
- **Strategija ali načrt implementacije**: to predstavlja (grobi) osnutek implementacije, časovni načrt, potek vpeljave orodja, glavna tveganja, kako se izogniti tveganjem teroceno izpolnjenih pogojev.
- **Stabilno testno okolje**: brez stabilnega testnega okolja so skripte neuporabne ali je njihov rezultat napačen. Poleg tega se zmanjšuje sam časovni okvir, v katerem se izvajajo skripte, tako da se izvajajo manjkrat.
- **Identificiranje, kaj (je lahko) bo avtomatizirano**: določenih stvari se ne da avtomatizirati ali jih je bolje narediti ročno. Ugotoviti je potrebno, kaj je lahko ali mora biti spremenjeno v programski rešitvi, ki je predmet testiranja, da bo testiranje bolj učinkovito ali lažje izvedeno. Vsa orodja za testiranje grafičnega uporabniškega vmesnika temeljijo na zmožnosti, da zaznajo in enolično identificirajo vsako pogovorno okno ali objekt. To je v končni fazi odvisno tudi od tega, kako je narejena naša programska rešitev.
- **Dodeljeni in dovolj usposobljeni strokovnjaki**: zaposleni potrebujejo primerna znanja in sposobnosti za sodelovanje v projektu. AT programskih rešitev je po navadi bolj tehnično usmerjen kot ročno testiranje. Pomembno je tudi, da imajo zaposleni, ki so dodeljeni projektu, dovolj zanimanja za to področje, pri tem je dosti bolj kot kvantiteta zaposlenih pomembna kvaliteta. Izkušen tester bo lahko enak ali boljši

rezultat dosegel v bistveno krajšem času in z manj dodatnega dela, kot je ukvarjanje z birokracijo. Uporaba testerjev, ki so prej izvajali le ročna testiranja, se po navadi izkaže kot neuspeh. Rezultat je po navadi omejen, ne široko uporabljan in ne najbolj zanesljiv testni sistem. Le redki ljudje so na delovnem mestu sposobni učinkovitega izvajanja več kot ene kompleksne naloge, in če že, te po navadi niso narejene enako dobro. Ena od nalog bo tako trpela, saj se vsak posameznik po navadi odloči za tisto, ki je zanj bolj zanimiva ali ob reševanju katere se bolje počuti. Pisanje testnega programa, ki je zanesljiv in se ga da vzdrževati, stalno zagotavlja natančne rezultate in učinkovito testira posamezno funkcionalnost, po navadi zahteva več truda kot ročno testiranje, zato bo ena izmed obeh aktivnosti trpela.

- Dobre prakse testiranja: to je eden izmed ključnih pogojev za uspeh avtomatizacije. Testna skripta je dobra samo toliko, kot je dober testni scenarij. Testnih scenarijev, ki so bili pisani za ročno izvajanje, pogosto ni mogoče uporabiti neposredno za avtomatizirano izvajanje.
- Čas trajanja projekta: na začetku se večina časa porabi za kreiranje testnih skript in kasneje za njihovo optimizacijo. Testna skripta lahko pokrije investicijo časa zgolj z večkratnim izvajanjem.

2.4 Ključni dejavniki uspeha pri uvajanju avtomatizacije testiranja programske opreme

Avtorji in organizacije navajajo veliko različnih ključnih dejavnikov uspeha pri uvajanju orodja za AT. Ta se lahko od projekta do projekta implementacije razlikujejo, saj ni možno po istem kopitu implementirati enakega orodja v različnih organizacijah. Do teh razlik prihaja zaradi različnih dejavnikov, kot so velikost organizacij, vrste programske opreme, za katero želijo narediti AT, (pred)znanja in izkušenj, virov, časovnih rokov in pričakovanj.

Podjetje Telerik je opravilo raziskavo o avtomatizaciji testiranja uporabniškega vmesnika med 985 različnimi informacijskimi strokovnjaki o njihovih izkušnjah pri implementaciji orodja. Večina (40,00 %) strokovnjakov je svoje znanje programiranja ocenila kot najvišjo stopnjo na lestvici, to je »izkušen«. Četrtina vseh vprašanih je bila neposredno udeležena v pilotni projekt avtomatizacije, večina (63,00 %) nikoli udeležena. 39,00 % vprašanih je zaposlenih v podjetju z med 1 in 50 zaposlenih, 26,00 % v podjetju z med 50 do 500 zaposlenih in 35 % v podjetju z 500 do 1000 zaposlenimi. Velikost testne ekipe je bila pri večini (57,00 %) med 1 in 10. Večina vprašanih (37,00 %) dela v računalniškem in informacijskem sektorju. V teh podjetjih je skoraj polovica (45,00 %) teste izvajala samo ročno, tretjina (35,00 %) je imela že avtomatizirane. Velika večina (49,00 % in 43,00 %) je želela avtomatizirati programske rešitve, izdelane v tehnologiji Javasript in HTML 5. Glavne ugotovitve raziskave so, da zgodnje vključevanje razvijalcev pri pripravi sistema na avtomatizacijo, bistveno pripomore k uspešnosti projekta tudi na dolgi rok, saj je večina podjetji, ki je to storila, imela uspešno izvedene projekte avtomatizacije. Pilotni projekti avtomatizacije odločijo o prihodnosti avtomatizacije v podjetjih. Večina podjetij, ki ne

uspe, ne poskusi ponovno. Strokovnjaki s programskimi izkušnjami so bili najbolj zaslužni, da so projekt, ki ne bi uspel, spremenili v uspelega. K uspešnosti avtomatizacije so prispevali naslednji glavni faktorji - razvijalski tim je priredil programsko rešitev avtomatizaciji, management je imel realistične zahteve in komunikacija med vodjem projekta avtomatizacije in managementa je bila učinkovita (Bozhkova, 2014, 12. november).

Po mnenju organizacije ISTQB (2014, str. 13–14) so najpomembnejši dejavniki pri implementiranju programske opreme za AT naslednji:

- **Dobra arhitektura testne avtomatizacije**: arhitektura avtomatizacije testiranja je arhitektura programske opreme orodja za delo in naj bo tako tudi obravnavana. Jasno mora biti, katere zahteve naj ta arhitektura podpira, in še pomembneje, katere nefunkcionalne zahteve so ključne. Kot pri drugih projektih programske opreme vse nefunkcionalne zahteve ne morejo biti izpolnjene med avtomatiziranim testiranjem, najbolj pomembne bi morale biti jasno identificirane pred razvojem. Arhitektura mora biti zasnovana, tako da jih lahko podpre, ter zasnovana za vzdrževanje, zmogljivost in sposobnost učenja. Modro je vključiti ljudi, ki poznajo arhitekturo sistema, ki bo testiran.
- **Sistem, ki je predmet testiranja, mora biti zasnovan, tako da ga je možno testirati**. Potrebne so spremembe tega sistema, tako da podpira AT in nove vrste testiranja, ki so izvedene na njem. To je ključ do uspeha avtomatizacije. V primeru testiranja uporabniškega vmesnika je pomembno, da se karseda loči interakcije v uporabniškem vmesniku in podatke od izgleda grafičnega vmesnika. V primeru testiranja programskih vmesnikov (ang. *API*) to lahko pomeni, da je več razredov, modulov in vmesnikov komandne vrstice »izpostavljenih« kot javnih, tako da so lahko testirani. Zmožnost testiranja sistema mora biti predvidena v načrtu sistema, ki je predmet testiranja, da se lahko omogoči avtomatizirano testiranje.
- **Deli sistema, ki jih je mogoče testirati preko avtomatizacije, naj bodo v prvi ciljni skupini**. V splošnem uspešnost testne avtomatizacije leži v tem, kako enostavno je implementirati skripte za avtomatizacijo. S tem ciljem v mislih in zato da se zagotovi ustrezno dokazilo koncepta, mora inženir testne avtomatizacije prepoznati module ali komponente sistema, ki jih je enostavno testirati z avtomatizacijo in s projektom začeti tam.
- **Vzpostavitev praktične in konsistentne strategije, ki odgovarja na vprašanja glede sposobnosti vzdrževanja in izboljšav ter konsistentnosti**. Lahko, da ni možno uporabiti avtomatizacije na enak način za nove in stare dele sistema. Pri ustvarjanju strategije je potrebno premisliti stroške, koristi in tveganja pri uporabi avtomatizacije za različne dele kode. Konsistentnost testiranja pomeni, da je testiranje uporabniških in programskih vmesnikov z avtomatiziranimi skriptami in preverljivost konsistentnosti rezultatov možno kadar koli.
- **Ustvariti ogrodje za AT, ki je lahko namenjeno za uporabo ter dokumentiranje in ga je mogoče vzdrževati.** Za to je potrebno narediti naslednje:

- Implementirati zmogljivosti za poročanja, ki zagotavljajo informacije deležnikom o kvaliteti sistema, kot so, ali je bil test izveden uspešno ali ne, informacije o napakah, ali test ni bil zagnan, statistični podatki.
- Omogoča enostavno odpravljanje težav. Ogrodje za avtomatizacijo mora omogočati enostaven način za odkrivanje in odpravljanje napak testov, ki lahko padejo zaradi težav v sistemu, ki je testiran, napak v testih ali testnem okolju. Da bi lahko odkrili izvir napak neuspešnih testov, mora sistem za avtomatizacijo testiranja omogočati, da se lahko izbere določen, manjši del testov in se jih zažene. Omogočena mora biti tudi analiza njihovih izvajanj.
- Ustrezno testno okolje. Testna orodja so odvisna od konsistentnosti testnega okolja. Za AT je nujno imeti posebno okolje. V primeru, da ni nadzora nad testnim okoljem in podatki, zahtevana vzpostavitev za teste ne bo dosegala zahtev za testna izvajanja in je verjetno, da bo kot rezultat proizvedla napačne podatke o izvajanju.
- Dokumentiranje avtomatiziranih testov. Cilji avtomatizacije morajo biti jasni, katere dele sistema testiranja, do katere stopnje, in kateri atributi bodo testirani (funkcionalni ali nefunkcionalni).
- Sledenje avtomatiziranim testom. Pomembno je, da orodje omogoča sledenje, ki omogoča upravitelju, da lahko sledi podatkom o korakih, ki so bili izvedeni.
- Enostavno vzdrževanje. V idealnem primeru je vzdrževanje avtomatiziranih testov enostavno, tako da le-to ne zahteva večjega truda, vloženega v avtomatizacijo. Za dosego tega je možno testne scenarije analizirati, jih spreminjati in razširiti.
- Testni scenariji morajo biti ažurni. Testni primeri ali celotne zbirke so lahko neuspešne zaradi napak, ki so odkrite v sistemu, ki je predmet testiranja, ali zaradi spremenjenih zahtev glede omenjenega sistema.
- Načrtovati za namestitev: testne skripte je lahko namestiti, spremeniti in odstraniti.
- Spremljanje in obnovitev sistema, ki je predmet testiranja v praksi za neprestano izvajanje testnih scenarijev ali zbirke, mora biti sistem, kjer je predmet testiranja nenehno nadzorovan. V primeru kritične napake se mora biti orodje za avtomatizacijo sposobno ustaviti in nadaljevati z naslednjim testnim primerom.

Ni nenavadno, da ima sistem za avtomatizacijo ravno toliko ali več vrstic kode kot sistem, ki je predmet testiranja, zato je zelo pomembno, da ga je mogoče vzdrževati. To kodo je po navadi težje vzdrževati kot samo kodo sistema, in sicer zaradi različnih orodij, ki se uporabljajo, različnih tipov preverjanj in različnih artefaktov, ki morajo biti vzdrževani, kot so npr. vhodni podatki in testna poročila. Več faktorjev uspeha, kot je ali še bo izpolnjenih, večja je verjetnost, da bo projekt avtomatizacije uspel. Vsi faktorji niso pomembni in so v praksi težko izpolnjeni. Pred začetkom testiranja je potrebno analizirati možnosti uspeha z analiziranjem faktorjev, ki že obstajajo in tistih, ki še manjkajo. Ko je avtomatizacija v uporabi, je potrebno raziskati, kateri predmeti manjkajo in kateri potrebujejo še dodatno

delo. Morda je smiselno delo na manjkajočih faktorjih pred samim začetkom projekta, vendar je mogoče začeti tudi brez njih. V vsakem primeru morajo imeti ljudje, zadolženi za avtomatizacijo, v glavi tveganja, povezana s pristopom. Ni zagotovljene poti, kako se odločiti, ali začeti s projektom ali delati na izboljševanju manjkajočih faktorjev. To je odvisno od vsebine projekta in okolja (ISTQB, 2014, str. 14).

2.5 Koristi in nevarnosti avtomatizacije testiranja programske opreme

2.5.1 Koristi avtomatizacije testiranja programske opreme

Glavne koristi avtomatizacije testiranja programske opreme so naslednje (Base36; 2013, 19. marec; Ferrari, 2013, 19. September; ISTQB, 2014, str. 12; Seetaram, 2011, 17. september):

- Več testov se izvede na eno graditev.
- Možno je izvajati teste, ki jih ročno ni mogoče (v realnem času, vzporedno, na daljavo), določena varnostna testiranja ter testiranja zmogljivosti.
- Testi so lahko bolj kompleksni.
- Testi se izvajajo hitreje in učinkovitejše. Začetna namestitev orodja lahko zahteva nekaj časa, prav tako vzpostavitev avtomatiziranih testov. Kasneje se te teste lahko ponovno večkrat uporabi (za isto graditev), kar je še zlasti uporabno za regresijske teste. To omogoča tudi hitrejšo splovitev programske opreme na trg.
- Testi so bolj odporni na napake osebe, ki jih izvaja.
- Bolj uspešna in učinkovita uporaba testerjev. Avtomatizirani testerji se lahko samodejno izvajajo in pošiljajo poročila. Človeški poseg ni potreben. Testerji lahko ta čas porabijo za druge naloge, za katere prej niso imeli časa.
- Boljše sodelovanje testerjev z razvijalci.
- Izboljšana zanesljivost sistema.
- Izboljšana kvaliteta testov in zanesljivost rezultatov. Avtomatizirana skripta bo izvedena vedno enako in rezultati bodo vedno zabeleženi.
- Je lahko stroškovno učinkovita. Orodja za avtomatizacijo so lahko draga na kratek rok, prihranek je možen na dolgi rok. Ne samo zato ker lahko izvedejo več opravil kot človek, temveč tudi zato ker lahko hitreje najdejo napake, kar zaposlenim omogoča, da se nanje hitro odzovejo in jih popravijo.
- Je bolj zanimivo. Ročno izpolnjevanje vedno istih obrazcev in vnašanje enakih podatkov postane monotono. Vzpostavitev testnih scenarijev zahteva večinoma kodiranje in razmislek, ki skrbi, da ostanejo tehnično najbolj izpopolnjeni strokovnjaki zavezani delu. To predstavlja tudi več izziva testerjem, ki bodo primorani bolje medsebojno sodelovati in izpopolnjevati svoje tehnično znanje.
- Teste lahko vidi kdor koli. Medtem ko tester izvaja teste ročno, je težko videti rezultate testiranj, ki jih izvaja. Pri AT si je mogoče te rezultate ogledati med izvajanjem testa ali na koncu. To izboljša sodelovanje med različnimi oddelki in pripomore k boljšemu končnemu izdelku.

2.5.2 Nevarnosti avtomatizacije testiranja programske opreme

Glavne nevarnosti avtomatizacije testiranja programske opreme so naslednje (Base36; 2013, 19. marec; ISTQB, 2014, str. 13; Seetaram, 2011, 17. september):

- Vsi ročni testi ne morejo biti avtomatizirani.
- Lahko preveri samo rezultate, ki jih lahko interpretira stroj (računalnik).
- Lahko preveri samo tiste aktualne rezultate, ki jih lahko primerja programsko orodje za avtomatizacijo. Orodje ne more testirati stvari, povezanih z izgledom, kot je barva ali velikost pisave.
- Orodja za avtomatizacijo lahko stanejo veliko denarja, zato je dobra izbira pomembna. Napačna izbira orodja lahko pomeni velike stroške za organizacijo in posledično »ubije« projekt avtomatiziranja.
- Orodja še vedno lahko porabijo veliko časa, še zlasti pisanje in vzdrževanje avtomatiziranih skript. V primeru, da je teh skript veliko, to lahko pomeni nekaj dodatnih ur izvajanja.

3 OBRAVNAVANO PODJETJE

V tem poglavju bom predstavil podjetje, ki je v procesu izbire orodja za AT. Zaradi zaupnosti višine investicije v izbiro orodja imena ne bom razkril. Gre za srednje veliko mednarodno podjetje, ki ima več kot 100 zaposlenih. Podjetje razvija programsko opremo v oblaku za pokrivanje logističnih procesov podjetij iz različnih industrijskih in storitvenih panog. Kot sem omenil že v uvodu, ta programska oprema omogoča vnos naročil, odpremo naročil z različnimi vrstami prevoza, sledenje pošiljkam, optimizacijo poti, izdajo računov, vodenje evidence o kontejnerjih in njihovi vsebini ter upravljanje glavnih podatkov. Programska oprema, ki jo razvijajo, je sestavljena iz modulov. Moduli so si tako po starosti in izgledu kot tudi po tehnologiji dokaj različni. Vsak modul se prilagaja procesom in zahtevam stranke, tako da gre za programsko opremo »po meri«.

Podjetje je v večinski lasti velikega mednarodnega podjetja z dolgoletno tradicijo in več tisoč zaposlenimi. Matično podjetje obravnavanemu podjetju zagotavlja tudi določeno informacijsko infrastrukturo, kar pomeni, da se mora v določenih primerih podjetje prilagajati matičnemu, kot so na primer procesi dodeljevanja administratorskih pravic zaposlenim podjetja, ki lahko namestijo nova programska orodja le z uporabniškim računom, ki nima dostopa do interneta. To pomeni tudi, da matično podjetje postavlja pravila glede tega, katere programske rešitve so lahko nameščene. To vpliva tudi na v nadaljevanju obravnavna programska orodja, ki morajo slediti tem smernicam. Izjeme so možne, vendar zahtevajo kar nekaj procedur pred samo vpeljavo.

3.1 Opis trenutnega stanja procesa razvoja in testiranja programske opreme v podjetju

Podjetje za razvoj programske opreme uporablja »klasični« V-model. V grobem bi delo podjetja lahko razdelili na razvoj programske opreme za projekte in razvoj sprememb, ki jih naročajo obstoječe stranke, ki so po velikosti vsa večja od podjetja, ki razvija rešitve zanje. Projekti po navadi trajajo vsaj nekaj izdaj programske opreme, lahko tudi več kot 1 leto. Na leto je trenutno v podjetju 5 izdaj posodobitve programske opreme, ki so začrtana na koncu preteklega leta. Razvoj sprememb vključuje razvoj manjših sprememb, kot so spremembe v obstoječi funkcionalnosti ali dodajanje nove. Poleg teh dveh obstaja še razvoj produkta, ki vključuje funkcionalnosti, ki jih uporabljajo vse obstoječe stranke (npr. posodobitve glede varnosti) ali funkcionalnost, ki jo podjetje lahko prodaja obstoječim in novim strankam. V primeru projektov se projektni vodja, management podjetja in predstavniki stranke na začetku projekta dogovorijo o njegovem obsegu, finančnih vidikih, potencialnih novih funkcionalnostih, ki bodo razvite oz. prilagojene podjetju. V primeru manjših sprememb stranka stopi v stik z oskrbnikom ključnih strank ali projektnim managerjem. V obeh primerih je rezultat strankinih zahtev izdelana poslovna specifikacija zahtev. Projektni manager v večini primerov stopi v stik s produktnim managerjem ali drugim specialistom, ki mu svetuje o izvedljivosti zahteve in izdela določene popravke, če je to potrebno. Rezultat je izdelana poslovna specifikacija, ki je poslana stranki v odobritev. V kolikor se stranka ne strinja s specifikacijo ali z določenimi deli le-te, sledi več iteracij, kjer se v specifikaciji dela popravke toliko časa, dokler obe strani ne dosežeta dogovora. Zaradi naštetega lahko ta korak traja dlje časa, kot je predvideno na začetku, kar ima lahko negativne posledice za nadaljevanje procesa.

V naslednji, produktni, fazi manager izdela tehnično specifikacijo, ki vsebuje podatke o podatkovnih tabelah, funkcionalnosti, ki jo je potrebno spremeniti ali se jo lahko uporabi, naredi grobe napotke razvijalcem, naredi oceno časa dela produktnega managerja, razvijalca in testerja, da opravijo njihove naloge. Vsi ti podatki se vnesejo v kartični sistem, Atlassian JIRA.

Razvijalec na podlagi tehnične specifikacije in napotkov produktnega managerja razvije zahtevo. V primeru večjih posegov oz. sprememb v kodi izdela dokumentacijo v internem dokumentacijskem sistemu. Programer izdela tufi kratke napotke testerju o tem, kako testirati razvito funkcionalnost. Ko programer zaključi s svojim delom, je na vrsti testiranje. V primeru večjih projektov se pred začetkom testiranja izdela testne strategije, matrike sledljivosti in testne načrte. V primeru sprememb ali hroščev tester običajno, če mu čas to dopušča, izdela testni načrt. V kolikor tega časa ni, se napiše zgolj kratke informacije o testiranju, predvsem o korakih. Ob odkritju napake lahko zavrne kartico razvijalca in napiše poročilo o hrošču. Programer mora nato poročilo o hrošču analizirati in ga odpraviti ali dobiti dodatne informacije od testerja.

V primeru projektov se po zaključku celotnega testiranja za določeno izdajo oz. graditev obvesti stranko, ki nato prične s testiranjem sprejemljivosti, pri čemer ji lahko pomagajo

praktično vse vrste zaposlenih v prej opisanem postopku razvoja in testiranja, le programerji bolj redko. To običajno zahteva pošiljanje podatkov o testiranju, postopkih testiranja, odkritih in odpravljenih hroščih ter morebitnih specifikacij. V primeru, da bi stranka želela za delovanje uporabljati svoje obstoječe programske rešitve, je potrebno pred testiranjem sprejemljivosti opraviti integracijsko testiranje. Po končanem testiranju sprejemljivosti stranka bodisi zavrne bodisi sprejme novo funkcionalnost. Testiranje sprejemljivosti mora biti končano pred zadnjo graditvijo, ki je običajno v sredo, v tednu, ko je nova izdaja (vedno ob nedeljah zaradi čim manjšega vpliva na motenje delovanje produkcijskega sistema). V kolikor stranka zavrne novo funkcionalnost, jo je potrebno v sistemu onemogočiti ali popolnoma odstraniti.

Produkcijske hrošče in probleme se odpravlja z vročimi popravki. Z njimi se lahko naknadno uvede tudi manjše spremembe, v kolikor jih ni bilo mogoče končati do nove izdaje.

3.2 Projekt avtomatizacije testiranja programske opreme

Projekt avtomatizacije testiranja se je v podjetju začel z izdelavo strategije, njen okvirni načrt je pripravil vodja testnega oddelka. Vodja oddelka in člani so si nato med seboj razdelili poglavja strategije. Vsak je tako prispeval določen del k dopolnitvi strategije. Glavni namen strategije je odgovoriti na vprašanje, kaj je njen glavni cilj, katera so glavna tveganja in kako jih zmanjšati ter podati konkretne korake za implementacijo. Cilj strategije je bil tudi izdelati približno »enotno« mnenje o tem, kaj sploh je avtomatizacija testiranja. Po končanju strategije je sledila prva iteracija s članom vrhnjega managementa, produktnim managerjem in predstavnikom razvijalcev Namen je bil predstavitev strategije prej omenjenim ter pridobitev »zunanjega« mnenja. Po prvi iteraciji se je pripravil osnutek za naslednjo iteracijo, ki je zajemal predvsem vprašanja, ki so ostala neodgovorjena v prvi iteraciji. V drugi iteraciji s podobno zasedbo se je te odgovore nato predstavilo akcijski skupini in izdelalo načrte za naslednje faze. Tej fazi je sledila vzpostavitev projektne skupine, ki je zasnovana in deluje po enakih princip kot ostale projekte skupine v podjetju, zadolžene za (interni) razvoj. Na prvem sestanku projektne skupine za avtomatizacijo testiranja je bil na pobudo vrhnjega managementa sprejet dogovor, da se organizira dve delavnici. Na prvi bi podjetje obiskal svetovalec, ki bi projektni skupini predstavil svoja mnenja, poglede in izkušnje ter ji svetoval pri izbiri orodja. Na drugi, dlje časa trajajoči delavnici, bi se nato določena orodja preizkusilo. V času pisanja te magistrske naloge se ni izvedla še nobena delavnica.

Še pred temi delavnicami se je v podjetju pričelo s pilotnim projektom. Ob vsaki novi izdaji programske rešitve se je ob nedeljah izvajalo dimne teste, pri čemer je vsaj eden od testerjev moral biti prisoten v podjetju. To testiranje zahteva veliko zbranosti, saj se izvaja na produkciji. Občasno prihaja tudi do zamud pri pripravi sistema na novo verzijo, kar pomeni, da je tester v podjetju v nedeljo vsaj nekaj ur, za samo testiranje porabi dobro uro, odvisno od njegove hitrosti in morebitnih težav, ki so odkrite. V podjetju so se zato odločili, da se za pilotni projekt avtomatizacije uporabi prav dimne teste, ki jih je bilo v

času pisanja magistrske naloge6. Za to avtomatizacijo se je uporabilo orodje Selenium IDE, ki je dodatek za brskalnik Firefox. Omogoča »posnemi in predvajaj« vrste kreiranja testov, tako da je skripte za avtomatizacijo možno narediti hitro. Orodje ima tudi določene pomanjkljivosti, saj ni najbolj primerno za vzdrževanje. Obenem se je odkrilo tudi določene pomanjkljivosti programske rešitve, ki je testirana, saj so bili določeni moduli slabo ali sploh neprilagojeni avtomatizaciji. V večini primerov predvsem zato ker določeni elementi na spletnih straneh nimajo unikatnih imen, kar naredi prepoznavo elementov za orodje za avtomatizacijo težje. Poleg tega je testno okolje, ki se uporablja za regresijsko testiranje (kjer je bil pilotni projekt najprej izveden) dokaj počasno, kar je potrebno upoštevati pri izdelavi testnih skript, da ne prihaja do neuspešnih izvedb določenih korakov prepoznave elementov zaradi predolgega čakanja, da se element pojavi na strani.

Izkušnja, ki sem jo doživel v enem izmed prejšnjih podjetji pri vpeljavi orodja za avtomatizacijo določenih poslovnih procesov, je ravno obratna. Takrat se je izbralo eno orodje, brez da bi testirali tudi preostala, prav tako ni bilo izdelane nobene formalne strategije. Po kakšnem letu uporabe se je orodje pričelo opuščati, saj je avtomatizacija pokrila le četrtino potreb.

3.2.1 Razlogi za avtomatizacijo testiranja programske opreme v podjetju

Nekateri razlogi za avtomatizacijo so bili posredno navedeni že v podpoglavju 3.1. Podjetje za testiranje uporablja V-model, ki je precej rigiden. V primeru zamud v vsaki fazi se te zamude prenašajo po »verigi« navzdol, pri čemer imajo testerji manj časa za testiranje in izdelavo testnih scenarijev, kar se posredno odraža tudi v končni kvaliteti. Pred vsako novo izdajo programske rešitve je dva in pol tedna t. i. »zamrznjene kode«, ko naj bi se v kodi lahko izvajalo samo popravke sprememb funkcionalnosti in ne dodajalo novih. V praksi je to obdobje pogosto kršeno zaradi zamud v razvoju in prejšnjih faz razvojnega cikla. Regresijsko testiranje se večinoma izvaja ročno. Glede na to, da je trenutno na leto 5 novih izdaj programske opreme, to pomeni, da je izvedenih vsaj toliko regresijskih testiranj. Večina testov se izvaja ročno, kar za testerje sčasoma postane dolgočasno. Prav tako se s povečevanjem novih funkcionalnosti in strank povečuje število regresijskih testov, ki jih je trenutno okrog dvesto. Testna oddelka sta omejena tako s številom zaposlenih kot tudi s časom. Avtomatizacija regresijskih testov bi tako testerje razbremenila izvajanja vedno istih testov in jim omogočila, da ta čas porabijo bodisi za testiranje novih funkcionalnosti bodisi za kreiranje novih testnih skript.

Podjetje si je za cilj za prvo obdobje (do novembra 2015) zastavilo avtomatizirati vsaj 25 % regresijskih testov. Poleg samega prihranka časa testerjev to pomeni tudi prihranek stroškov testiranja. Obenem lahko avtomatizirano testiranje izboljša ugled podjetja v očeh (potencialnih) strank kot modernega podjetja, ki sledi tehnološkim smernicam in ki ima potrebne zmogljivosti za testiranje. Stroški nedelovanja sistema bi bili lahko za stranke namreč že v primeru kratkega izpada ogromni. Poleg tega je podjetje zavezano k približno 99,9 % dostopnosti.

Določene vrste testiranja se trenutno izvajajo bolj redko. Testiranje zmogljivosti večinoma opravljajo sistemski administratorji, del teh testiranj bi se lahko prenesel na testne oddelke, oz. bi se lahko ustrezno razširil. Ročno testiranje je manj odporno na napake, poleg tega zahteva več časa za samo izvedbo kot za pripravo poročil.

3.2.2 Ocena vpliva uvedbe avtomatizacije testiranja programske opreme na trenuten proces testiranja programske opreme v podjetju

V poglavju 3.2.1 sem navedel nekaj razlogov za avtomatizacijo testiranja v podjetju, iz česar je mogoče sklepati nekaj možnih vplivov uvedbe avtomatizacije testiranja. Poleg prihrankov časa, stroškov testiranja ter razbremenitve monotonih testiranj, je vsaj v določenih primerih pričakovati, da bo uvedba avtomatizacije prinesla dodatne pozitivne učinke. Testerji v podjetju skupaj ne sodelujejo zelo pogosto, saj je po navadi vsak dodeljen določenemu projektu ali strankam in tako niti ni velikega timskega sodelovanja. Projekt avtomatizacije bi tako bil kot nek skupen projekt, ki bi izboljšal tako sodelovanje v testnih oddelkih kot tudi sodelovanje z razvijalci. Ta projekt bi delo testerjev naredil tudi bolj zanimivo, saj bi se morali soočiti z novimi izzivi, ki so bolj tehnološki, kar bi zahtevalo nova usposabljanja in določena nova znanja oz. uporabo obstoječih, ki trenutno niso v uporabi. To bi torej pozitivno vplivalo na motivacijo testerjev tako z vidika trenutnih zaposlitev kot tudi nadaljnjega kariernega razvoja. Vse več podjetij se namreč odloča za uvedbo avtomatizacije testiranja programske opreme ali išče izkušene strokovnjake s tega področja.

V podjetju se pojavljajo tudi ideje, da bi podjetje skušalo ponovno uvesti agilnejše pristope k razvoju programske opreme. V preteklosti so že izvedli en poskus, ki se je končal neuspešno, saj je bilo težko nadzorovati, kaj je že bilo razvito, in se držati dogovorjenih rokov s strankami. Večina strank podjetja so namreč velika podjetja, ki so dokaj rigidna in stremijo k sledenju svojim rokom in prav tako zahtevajo, da se jim podjetje v določeni meri prilagaja. Agilnejših pristopov k razvoju programske opreme se je skoraj nemogoče lotiti brez avtomatiziranega testiranja, še zlasti v primeru tega podjetja, ki svoje rešitve prilagaja strankam. Posledično bi hitrejše in bolj pogoste spremembe v kodi zahtevale še dodatna testiranja, ki bi jih bilo ročno težko zagotoviti, razen v primeru, da podjetje bistveno poveča število zaposlenih.

4 PREDSTAVITEV PREDLAGANIH PROGRAMSKIH REŠITEV ZA AVTOMATIZACIJO PROGRAMSKE OPREME V PODJETJU

V sklop priprave strategije je sodilo tudi zbiranje podatkov o potencialno zanimivih orodjih, ki bi jih podjetje lahko uporabljalo. V podjetju bodo najverjetneje vzpostavili ogrodje za avtomatizacijo, ki bo vključevalo različna orodja. Ker je le-teh ogromno, sem se odločil, da bom najprimernejše iskal med tistimi, ki podpirajo testiranje preko grafičnega uporabniškega vmesnika in so namenjeni spletnim stranem. V posvetovanju s člani testnega oddelka sem izdelal tudi izbor orodij za podjetje, ki se je na koncu skrčil na

približno 10 testnih orodij. V drugi fazi izbora sem poiskal 5 orodij, ki bodo opisana v nadaljevanju. Višina investicije v orodja je skrivnost, vendar podjetje ne želi kupovati najdražjih, kot so npr. orodja podjetij HP in IBM, in stremi k uporabi odprtokodnih rešitev ali rešitev manjših podjetij.

Podjetje ima razvito tudi mobilno aplikacijo za sistem Android, ki je namenjena predvsem sledenju pošiljkam. Pomemben del testiranja so tudi spletni vmesniki in javno dostopni programski razredi za prenos podatkov. Mobilna aplikacija predstavlja zelo majhen del portfelja podjetja in za testiranje ni pomembna, saj se redkeje razvija. Spletni oz. programski vmesniki predstavljajo pomembnejši del, vendar jih nisem upošteval v nadaljevanju, saj podjetje za njihovo testiranje uporablja svoje rešitve oz. se odloča za specializirana orodja.

V nadaljevanju sem nekatere vire črpal iz večih (pod)strani organizacij ali podjetij. V virih sem navedel le začetne strani posameznega orodja.

4.1 Selenium (Web Driver in Selenium IDE)

Selenium (b.l.). Selenium WebDriver. Najdeno 12. aprila 2015 na spletnem naslovu http://docs.seleniumhq.org/docs/03_webdriver.jsp

Gre za enega bolj priljubljenih (odprtokodnih) orodij za AT, ki je na trgu že nekaj časa. Zaradi odprtokodnosti in velike fleksibilnosti se je hitro razširil. Dejansko bi težko govorili o orodju kot takem, pač gre bolj za zbirko različnih knjižnic in dodatkov za brskalnike, kot je že prej omenjeni Selenium IDE. Slednji sicer poseduje grafični uporabniški vmesnik, ki ga je težko primerjati z orodji, opisanimi v nadaljevanju. V tem podpoglavju se bom posvetil predvsem Web Driverju, knjižnici za avtomatizacijo testiranja.

Začetek projekta Selenium sega v leto 2004, ko je Jason Huggins med ročnim testiranjem interne programske rešitve v podjetju, v katerem je delal, ugotovil, da bi te teste lahko avtomatiziral. Razvil je Javascript knjižnico, ki je omogočala interakcije s spletnimi stranmi in avtomatično izvajanje testov v različnih brskalnikih. Ta knjižnica je postala temelj Selenium Cora, ki je temelj Selenium Remote Controla (v nadaljevanju RC) in Selenium IDE-a. Selenium RC je bil prelomen, saj pred njim nobeno drugo orodje ni omogočalo nadzora brskalnika z različnimi programskimi jeziki. Kljub vsej zmogljivosti je imel Selenium RC tudi slabosti, saj je temeljil na Javascript knjižnici, ki jo je bilo zaradi varnostnih omejitev, ki so jih brskalniki postavili temu skriptnemu jeziku, vedno težje uporabljati. Spletne aplikacije so napredovale tudi zaradi novih tehnologij v brskalnikih, kar je naredilo te omejitve še hujše za Selenium RC. Leta 2006 je Simon Stewart delal na projektu v podjetju Google, ki ga je poimenoval WebDriver. Google je veliko uporabljal Selenium, testerji podjetja so se morali soočati z njegovimi omejitvami, zaradi česar je Stewart hotel razviti testno orodje, ki bi se sporazumevalo z brskalnikom v »naravnem« jeziku brskalnika in operacijskega sistema in se na ta način izogniti omejitvam Javascripta. Začetek projekta Web Driverja je torej nameraval zaobiti glavne omejitve Seleniuma. Leto

2008 je prineslo združitev Seleniuma, ki je imel ogromno podporo skupnosti in komercialno podporo podjetij, ki so plačevala za posebne storitve uporabniške podpore, WebDriver je Seleniumu prinašal tehnologijo prihodnosti (b.l.).

Najnovejša verzija knjižnice je Selenium 2, ki podpira bolj objektno orientiran pristop in obenem omogoča združljivost za nazaj. Vključuje vse dobre strani Seleniuma in WebDriverja. Kot sem že omenil, gre za programsko knjižnico, ki jo je mogoče uporabiti z različnimi programskimi jeziki. S Seleniumom IDE je mogoče »posneti« delovanje uporabnika v brskalniku in ga nato izvoziti v enega izmed programskih jezikov, ki ga Selenium podpira. To so Java, C#, Phyton, Pearl, Ruby in PHP. Te skripte je nato mogoče uvoziti v poljubno razvijalsko okolje, ki mora imeti dodane ustrezne knjižnice za uporabo WebDriverja in jih nato zagnati preko razvijalskega orodja. To torej pomeni, da je mogoče WebDriver integrirati z obstoječimi programskimi razredi in vmesniki. Za avtomatizacijo testov v brskalniku je potrebno imeti knjižnice za ta brskalnik. Selenium WebDriver podpira vse najbolj razširjene brskalnike, Internet Explorer, Google Chrome, Firefox, Opero in Safari, Selenium IDE zgolj Firefox. Selenium WebDriver nima uporabniškega vmesnika kot takega, možno ga je le integrirati v orodje po meri. Možnosti za uporabo so praktično neomejene, obenem knjižnica, ko se prenese iz spletne strani, nima praktično ničesar. Najprej je potrebno ustrezno namestiti določene ostale knjižnice in pripraviti računalnik za delo s sistemi za verzioniranje kode. V nadaljevanju je potrebno ročno narediti tako strukturo kot nastaviti poročila. Kljub brezplačnosti je potrebno vložiti kar nekaj časa, da se sistem vzpostavi tako, kot je potrebno. Na voljo je velika skupnost uporabnikov, ki je največja med vsemi opisanimi orodji. Obstaja tudi Selenium-Grid, ki omogoča vzporedno testiranje ter izvajanje skript v več okoljih. Sponzorji projekta nudijo podjetjem plačljivo podporo v obliki treningov, pomoči uporabnikom in orodij, ki temeljijo na Selenium WebDriver tehnologiji ipd. (Selenium, b.l.).

4.2 Borland (Micro Focus) Silk Test

Silk Test je produkt ameriškega podjetja Borland (Micro Focus), ki razvija različna orodja za pomoč pri razvoju programskih rešitev, od programskih orodij, namenjenih delu z zahtevami strank in specifikacijami, do orodij za podporo testiranju, ki zajemajo podporo upravljanju s testnimi scenariji, orodij za testiranje zmogljivosti in orodij za podporo avtomatizaciji. V zadnji sklop spadajo 4 orodja, DevPartner, Silk Central, Silk Mobile in Silk Test. Kljub temu da so za izbrano podjetje zanimiva vsa, sem za nadaljevanje izbral samo Silk Test. DevPartner je orodje, namenjeno analizi in odpravljanju napak v kodi, Silk Central testiranju spletnih aplikacij v oblaku ter Silk Mobile testiranju mobilnih programskih rešitev. Silk Test podpira testiranje spletnih in celovitih programskih rešitev. Po zagotovilih podjetja je mogoče ustvarjanje testnih scenarijev, brez pisanja kakršne koli kode, kar se v praksi pogosto izkaže za marketinški trik. Podobno kot Selenium WebDriver omogoča povezljivost z orodji za razvoj programskih rešitev, kot sta Microsoft Visual Studio in Eclipse. Slednjo rešitev izbrano podjetje uporablja za razvoj programskih rešitev. To omogoča pisanje skript za avtomatizacijo v programskem jeziku »po izbiri«. Podpira

avtomatizacijo v naslednjih spletnih brskalnikih, Google Chrome, Microsoft Internet Explorer in Mozilla Firefox. Glavno hrbtenico orodja predstavlja testiranje na podlagi ključnih besed, ki je bilo podrobneje opisano v poglavju 2.2.4, kar omogoča enostavno vzdrževanje in ustvarjanje novih testnih scenarijev, za kar se načeloma ne potrebuje nobenega znanja programskega jezika in ga lahko izvaja tudi višji management. V primeru, da so bile narejene spremembe v procesu prijave v spletno stran, kot je sprememba identifikatorja polja za vnos geslo, se to spremembo vnese v eno ključno besedo npr. »Prijavi se«, ki se nato avtomatično »preslika« povsod, kjer je ta beseda uporabljena, kar omogoča »spremembe na enem mestu«. Hkrati to pomeni, da je potrebno na začetku nastaviti smiseln načrt teh ključnih besed, da ne prihaja do nepotrebnega podvajanja funkcionalnosti. Testiranje na podlagi ključnih besed loči tudi testne scenarije od same implementacije skripte. Omogoča tudi t. i. agnostično testiranje, ki podpira prepoznavanje slik in besedila (Borland, b.l.).

4.3 Ranorex

Ranorex je programsko orodje istoimenskega podjetja Ranorex s sedežem v avstrijskem Gradcu, podružnico ima tudi v Združenih državah Amerike. Podjetje ima več kot 1700 strank, ki delujejo v 60 državah. Med večjimi in bolj znanimi strankami podjejta so Yahoo!, General Electrics, Philips in Bank of America. Podpira testiranje spletnih, mobilnih in namiznih programskih rešitev. Od spletnih brskalnikov podpira Google Chrome, Mozilla Firefox, Microsoft Internet Explorer in Apple Safari. Možno je »posneti« akcije uporabnika v enem spletnem brskalniku in nato isti testni scenarij izvesti v vseh preostalih brskalnikih, ki jih Ranorex podpira. Med podprtimi spletnimi tehnologijami so HTML 5, Adobe Flash, Microsoft Silverlight, AJAX, Java, in Javascript. Vključuje .NET knjižnico, ki omogoča enostavno integracijo z Microsoftovim orodjem Visual Studio in izvozom testnih skript v programska jezika C# in VB.NET (Ranorex Gmbh, b.l.).

Podobno kot programsko orodje Borland Silk Test podpira testiranje na podlagi ključnih besed. Sama funkcionalnost je v osnovi zelo podobna tisti pri orodju Silk Test, tako da je tu ne bom ponovno opisoval. Poleg tega podpira tudi podatkovno usmerjeno testiranje, ki sem ga podrobneje opisal v podpoglavju 2.2.2. Za vire je možno uporabiti Microsoft Excelove preglednice, CSV datoteke in SQL baze podatkov. Uporabiti je možno tudi spremenljivke, ki so shranjene v testnih scenarijih, narejenih z metodo »posnemi in predvajaj«, ali repozitorjih orodja. Kot sem navedel že pri opisu podatkovno usmerjenega testiranja, so glavne prednosti tega pristopa predvsem v tem, da testne skripte ostanejo nespremenjene, spreminjajo se namreč vhodni podatki, posledično tudi izhodni. To omogoča ločevanje logike testnih skript od vira podatkov (Ranorex Gmbh, b.l.).

4.4 SmartBear TestComplete

SmartBear je ameriško podjetje s sedežem v Bostonu, ki ima podružnice na Irskem, Švedskem in v Rusiji. Zaposlenih ima več kot 250 ljudi. V svojem portfelju ima tudi brezplačna orodja (npr. SoapUI), zato ni presenetljiv podatek, da orodja podjetja uporablja

več kot 3 milijone strokovnjakov s področja razvoja programskih rešitev v 194 državah sveta in več kot 25 000 organizacijah. Med večjimi strankami podjetja so Google, Intel, Apple, Microsoft, Walmart, General Electrics, J.P. Morgan, Disney in mnoga druga. Portfelj podjetja tvorijo orodja za podporo razvoju programskih rešitev, kot so orodja za optimizacijo, sodelovanje, testiranje in spremljanje zmogljivosti. Na voljo so tudi 4 brezplačna orodja in 2 odprtokodna. SoapUI je predstavnik orodij, ki spada v obe skupini in je tudi resen kandidat izbranega podjetja za testiranje programskih in spletnih vmesnikov. Orodje dosega veliko popularnost v skupnosti testerjev in razvijalcev programskih rešitev (SmartBear Software, b.l.).

TestComplete je platforma, ki sem jo izbral kot eno izmed potencialnih možnih rešitev za izbrano podjetje. Podpira testiranje spletnih, namiznih in mobilnih programskih rešitev. Od spletnih brskalnikov ponuja podporo Google Chrome, Microsoft Internet Explorer, Opera in Mozilla Firefox. Od spletnih tehnologij podpira HTML 5, Javascript, Ajax, Adobe Flash. Naslednja zanimivost je tudi, da podpira integracijo s Selenium WebDriver orodjem. Omogoča izvoz testnih skript v Phyton, VBScript, JavaScript, C++ Script, C# Script ali DelphiScript. Deluje na objektnem nivoju, kar pomeni, da poleg samih akcij uporabnika oz. testerja »posname« tudi ime objekta in njegove lastnosti ter akcije objekta in prepozna najbolj pogoste knjižnice za uporabniški vmesnik in specifične programske rešitve, ki so predmet testiranja ter uporablja enaka imena oken in kontro, kot so jim jih dodelili razvijalci. Poleg tega podpira tudi podatkovno usmerjen pristop (podpoglavje 2.2.2) (SmartBear Software, b.l.).

4.5 Froglogic Squish GUI Tester

Zadnje orodje, ki ga bom opisal, razvija podjetje froglogic GmbH, s sedežem v Hamburgu v Nemčiji. Po podatkih, ki so na voljo na spletni strani, ima podjetje manj kot 30 zaposlenih. Kljub majhnosti produkte podjetja uporablja več kot 3000 podjetij, med njimi velika podjetja, kot so Airbus, Google, Disney, General Electric, Boeing, Pfizer in Siemens (froglogic GmbH, b.l.).

Squish GUI Tester je orodje, ki je za pričujočo magistrsko nalogo najbolj zanimivo. Podpira AT namiznih, mobilnih, spletnih in vgrajenih programskih rešitev. Je edino programsko orodje med obravnavanimi, ki podpira vse glavne spletne brskalnike, torej Microsoft Internet Explorer, Google Chrome, Mozilla Firefox, Apple Safari in Opero. Med podprtimi tehnologijami so HTML 5, Javascript, AJAX in druga. Poleg standardnega pristopa, ki ga ponujajo vsa našteta orodja (Selenium s spletnim vtičnikom Selenium IDE) , to je »posnemi in predvajaj«, podpira tudi podatkovno usmerjeno testiranje in testiranje na podlagi obnašanja (podpoglavje 2.2.5). Slednje omogoča izdelavo testnih scenarijev tudi tistim uporabnikom, ki niso vešči programiranja. Podpira jezik Gherkin, ki je človeku lahko berljiv in je de facto standard na področju testiranja na podlagi obnašanj. To tudi jasno loči testno logiko od testne implementacije (froglogic GmbH, b.l.).

Uporabniški vmesnik opisanega orodja temelji na odprtokodni programski rešitvi, Eclipse. Omogoča izvoz testnih skript v programske oz. skriptne jezike, Phyton, JavaScript, Perl, Ruby in Tcl.

5 PRIMERJAVA ORODIJ

Obstaja več načinov, kako izbrati programsko orodje za določeno podjetje. Pri tem gre predvsem za vprašanje, kako zmanjšati subjektivnost pri izbiri. Vsak posameznik, ki izbira orodje, se ne more popolnoma znebiti svoje pristranskosti, in ga tako ocenjuje predvsem na podlagi osebnih preferenc in želja ter notranjih motivov. Izbrano orodje je morda lahko dobro za podjetje, ne pa nujno tudi za posameznika, ki želi imeti drugo orodje zaradi večjih kariernih zmožnosti. V enem izmed podjetij, v katerem sem delal kot avtor te magistrske naloge, so se izbire orodja po prvem neuspehu lotili tako, da so to nalogo zaupali drugemu oddelku, saj naj bi s tem dosegli večjo nepristranskost. Slabost tega pristopa je, da drugi oddelek ni glavni uporabnik tega orodja in ima tako morda manjši interes za njegovo izbiro. Večji problem je, da lahko izbrano orodje sproži »upor« v oddelku, ki naj bi bil njegov glavni uporabnik. Pri izbiri orodja gre tako predvsem za kompromis med željami različnih uporabnikov in časom, dobro ga je mogoče dobro spoznati šele po vsaj nekajtedenski uporabi, kar zahteva čas in stroške, ki si jih večina podjetij noče naložiti.

Večina organizacij izbira orodja predvsem na podlagi cene in pričakovanj, za katere želijo, da jih orodje izpolni. Metode za analizo in izbiro orodij segajo vse od osnovne intuicije do štetja zahtev, ki jih orodje izpolnjuje, ali kombinacij obojega. Izbira in ocenjevanje orodij mora biti narejena na konsistenten, kvantitativen način, da je lahko učinkovita. Z uporabo formalne metode je možno uporabiti mešane kriterije, ki pripeljejo do kohezivne odločitve, ki ne sme temeljiti samo na tehničnih, intuitivnih in političnih faktorjih (Bandor, 2006, str. 7).

Organizacija FedSolutions (2015, 7. julij) trdi, da podjetja kupujejo novo tehnologijo za zagotavljanje naslednjih ciljev: povečanje produktivnosti, naslavljanje operativnih izzivov in prihranek denarja. Kljub temu lahko nova tehnologija brez ustrezne strategije negativno vpliva na trud, ki je bil vložen v dosego ciljev. Še huje, izbira napačnega orodja lahko pomeni velike stroške vzdrževanja na dolgi rok, izogibanje uporabi s strani uporabnikov in nakup dodatnih programskih rešitev, ki pokrijejo nastale vrzeli v funkcionalnosti.

V prejšnjem odstavku omenjena organizacija navaja 10 kriterijev za izbiro orodja (FedSolutions, 2015, 7. julij):

- Vedeti moramo**, zakaj sploh potrebujemo programsko rešitev**.
- **Prepoznati je potrebno prioritete**: potrebno je razumeti zmožnosti posameznih orodij in kako lahko te pozitivno vplivajo na organizacijo. Analizirati je potrebno želje in jih rangirati po pomembnosti, katere morajo biti nujno izpolnjene in katere so samo »če je možno«.

- **Kritičnost za poslovanje**: V primeru, da programska rešitev ni dosegljiva, to lahko čez nekaj časa privede do škode za poslovanje. Tak primer so napake v programskih rešitvah za finance. Kritično za poslovanje, je, da za zaposleni nemoteno izvajajo njihove zadolžitve in odgovornosti. Zadnja stvar, ki jo podjetje potrebuje. je nedostopnost programske rešitve, ki lahko resno ogrozi poslovanje. To je potrebno razumeti, ko se ocenjuje kredibilnost proizvajalca programske opreme, odvisnost produkta od drugih tehnologij in podporo uporabnikom.
- **Kredibilnost in dolgoživost proizvajalca opreme**: pred nakupom je smiselno raziskati zgodovino proizvajalca, izvedeti, kako dolgo je na trgu in kakšen je njegov ugled med strankami.
- **Zanesljivost programske opreme**: potrebno je ugotoviti, ali obstajajo kakšne nepravilnosti v programski opremi, ki lahko ogrozijo njeno delovanje, ter koliko časa je potrebnega, da se odpravi napake. Smiselno je preizkusiti tudi odzivnost proizvajalca na naš klic in poizvedbe, preiskati spletne strani in forume, da se dobi realnejšo sliko o programski rešitvi. Bivše stranke podjetja se, če je to možno, kontaktira in vpraša, zakaj so zamenjali programsko rešitev.
- **Operativna integracija**: odgovoriti je potrebno na vprašanje, kako enostavno se izbrana rešitev integrira v obstoječe okolje, ali izboljša procese in prihrani čas, kdo bo uporabljal programsko rešitev, kje, kako, ipd. Narediti je potrebno načrt o tem, kdo bo uporabljal programsko orodje in kako ga implementirati v organizacijo. Nujno je potrebno ugotoviti, ali bo programska rešitev izpolnila zahteve.
- **Model podpore**: kako deluje tehnična podpora v primeru, da zaposleni odkrijejo napako, ali proizvajalec produkta zagotavlja pomoč samo po elektronski pošti ali tudi po telefonu, kakšen je odzivni časa, ali so na voljo potrebne informacije, ali obstajajo kakšni skriti stroški.
- **Fleksibilnost za rast**: ali bo programska rešitev še vedno primerna v primeru, da podjetje raste in bo to povečalo stroške njene uporabe. Lahko se pojavijo problemi z licencami, ko bi produkt uporabljali novi zaposleni.
- **Cena**: cena programske rešitev naj se odraža v funkcionalnosti in podpori, ki je na voljo.
- **Kako se bo merila donosnost na investicijo**: odgovoriti je potrebno na vprašanje, ali bo nova programska rešitev zamenjala obstoječe in ali se bo izboljšala kvaliteta dela ali produkta.

Stone (2013, 7. junij) navaja 6 korakov za objektivno izbiro orodja:

- **Načrtovanje in iniciacija**: potrebno je pridobiti poglobljeno razumevanje poslovanja in faktorjev, ki vodijo odločitev. Pregledati je potrebno poslovne in funkcionalne zahteve na visokem nivoju in pridobiti odzive iz poslovnih območij, kot so finance, informacijska tehnologija, prodaja in marketing, skrb za stranke, pogodbe in nabave.
- **Izdelati dolg seznam potencialnih dobaviteljev programskih rešitev**: ta seznam temelji na osnovnem razumevanju potreb. Uporabi se dolg seznam funkcionalnosti, ki jih imajo vodilna podjetja na področju, iz katerega je programska rešitev.

- **Obdelava zahtev**: pregledati je potrebno zahteve vsakega poslovnega območja in ostale zahteve, kot so arhitekturne in komercialne. Priredi se seznam iz prejšnje faze. Dogovori se o prioritetah zahtev na podlagi metode MoSCoW (ang. *Must, Should, Could, Won't*) – »mora imeti«, »naj bi imel«, »bi lahko imel«, »ne bo imel«. Najmanjšo možno funkcionalnost in kritično funkcionalnost je potrebno dati v kategorijo »mora imeti«.
- **Izdelava krajšega seznama**: na podlagi »mora imeti« kriterija se hitro izdela krajši seznam z največ od 2 do 4 orodji, ki imajo največji potencial, da se dobro prilagodijo na dolgi rok. Ta seznam se nato uporabi v naslednji fazi.
- **Ocenjevanje programskih rešitev z matriko za ocenjevanje na podlagi »mora imeti« in »naj bi imel«**: v primeru, da je to možno, se organizira predstavitev proizvajalca programske opreme. Proizvajalcu opreme se predstavi poslovne zahteve.
- **Izbira orodja, pogajanje in pogodba**: uporabi naj se vse kriterije in izkušnje ter kvalitativne vpoglede, kot so kvaliteta predlogov, demonstracije in pripravljenost na izpolnjevanje poslovnih potreb. Uskladiti se je potrebno glede najvišje izbire in izpogajati pogodbo, ki deluje za poslovanje organizacije, v kolikor je to možno. Začeti je potrebno priprave na implementacijo.

5.1 Kriteriji za izbiro orodja

Za primerjavo bom izdelal matriko, katere namen je na podlagi tehtanih povprečnih uteži pridobiti objektivnejšo sliko za izbiro. Matrika je predstavljena v naslednjem podpoglavju, 5.2, v katerem so opisani rezultati primerjave.

Vsak kriterij, ki sem ga ocenjeval, ima lahko oceno od 0 (najnižja ocena) do 10 (najvišja ocena).

Pri določenih kriterijih seveda obstaja nevarnost subjektivnega ocenjevanja. Več kot je teh kriterijev, nižja je možnost subjektivnosti. Za realnejšo oceno bi enak postopek moralo opraviti več ocenjevalcev.

Vse ocene sem zaokroževal na 0,5.

5.1.1 Cena in stroški vzdrževanja

Podjetje sicer načeloma nima limita za nakup orodja, saj je pomembnejša njegova kvaliteta, vendar sem v primerjavo kljub temu vključil tudi ta kriterij. Razlogov za to je več. Eden je ta, da je to eden od kriterijev, za katerega je najlažje pridobiti kvantitativne podatke. Naslednji razlog je, da napačna izbira, kot sem navedel v začetku v poglavju 5, lahko prinese veliko stroškov in zmanjša možnosti za ponovni zagon projekta. Finančna sredstva je smiselneje porabiti za druge stvari, kot je trening zaposlenih, nakup drugih orodij, najem svetovalca ipd.

Stroški, ki so pri nakupu orodij pogosto skriti, jih je težko izračunati ali se nanje preprosto pozabi, so stroški vzdrževanja. Težko je namreč izračunati, koliko sredstev bo zlasti na

dolgi rok zahtevalo nameščanje, usposabljanje ljudi in vzdrževanje orodja. Pojavijo se lahko nepredvideni dogodki, kot sta daljše nameščanje orodja na sistem podjetja ali zahtevno usposabljanje. Največji strošek je ob tem predvsem čas, ki ga bodo testerji namenili avtomatizaciji in bi bil lahko porabljen za druge stvari.

V kolikor ima podjetje dovolj podatkov in kazalnikov, se lahko izračuna tudi donosnost na investicijo.

Pri tem kriteriju sem ocenjeval nakupno ceno in stroške licenčnin in vzdrževanj za 5 let za 5 zaposlenih (5 poimenskih licenc) in 1 licenco za strežnik (za 5 let). Kjer ni bilo razvidno, koliko so stroški za 1 strežnik, sem računal, kot da je še en dodatni uporabnik, torej skupno 6.

Tabela 1: Cena in stroški vzdrževanja

Orodje	Licenca	Strežnik	Podpora	Podpora (strežnik)	Skupaj	Končna ocena
Selenium	0 €	0 €	0 €	0 €	0 €	10
Borland[1]	1800 €	0 €	378€ / leto	378€ / leto	19.872 €	3
Ranorex[2]	1990 €	690 €	390 €	149 €	19.036 €	3
SmartBear[3]	889 € + 889 €	889 € + 889 €	889 € + 889 € (2 leti)	889 € + 889 € (2 leti)	26.690 €	2
froglogic[4]	2.400 € (1 leto podpore)	2.400 € (1 leto podpore)	800 € (2 leti)	800 € (2 leti)	24.000 €	2

Vir: Lastni izračun

5.1.2 Združljivost s sistemom, ki je predmet testiranja.

Obravnavano podjetje spletne strani razvija v programskem jeziku Java. Za dinamične prikaze uporablja Javascript in AJAX. Uradno podprta brskalnika sta Microsoft Internet Explorer in Mozilla Firefox. Kljub temu nekaj uporabnikov uporablja tudi drugi brskalnike, med katerimi je najbolj pogost Google Chrome. Po statistki W3Schools (b.l.), organizacije za spletne standarde, je glede na junijske rezultate daleč na prvem mestu Google Chrome (64,8 %), ki mu sledi Mozilla Firefox (21,3 %), Microsoft Internet

[1] Podjetje Borland na svoji strani nima objavljenih cen produktov. Cene, ki sem jih našel na spletu, se razlikujejo, zato sem kontaktiral podjetje. V času pisanja magistrske naloge še nisem prejel odgovora. Cene sem iz tega razloga vzel s tiste spletne strani, ki je bila najbolj ažurna. Vir: Software Quality Lab. Najdeno 23. Avgusta na spletnem naslovu http://www.software-quality-lab.com/fileadmin/files/download/Studien/SWQL-UI-TestAutomationStudy.pdf

[2] Vir: *Ranorex*. Najdeno 17. avgusta na spletnem naslovu http://www.ranorex.com/purchase/buy-now.html

[3] Vir: *SmartBear*. Najdeno 17. avgusta na spletnem naslovu http://smartbear.com/product/testcomplete/pricing/

[4] Podatki niso na voljo na spletni strani podjetja, zato sem poizvedbo poslal preko elektronske pošte. Podjetje nima verzije za strežnik.

Explorer (7,1 %), Apple Safari (3,8 %) in Opera (1,8 %). Podjetje sicer skuša razširiti krog podprtih brskalnikov. Orodje mora torej podpirati vsaj Internet Explorer in Firefox, dobro je, če tudi Chrome.

Ta ocena je sestavljena iz 2 delov: 50 % ocene je orodje dobilo za brskalnik, 50 % za lokatorje, ki bodo predstavljeni v nadaljevanju. Brskalnikov je ravno 5, vendar sta Internet Explorer in Firefox pomembnejša za podjetje, zaradi česar imata večji ponder, vsak 15 %, preostali po 10 % oz. 5 %.

Uteži brskalnikov v odstotkih (50 % je skupaj) so torej:

- Internet Explorer: 15 %.
- Firefox: 15 %.
- Chrome 10 %.
- Safari 5 %.
- Opera 5 %.

Ocene orodij glede na brskalnike:

- Selenium (Web Driver in Selenium IDE): Opera ni podprta. Skupaj 45 %.
- Borland (Micro Focus) Silk Test: Safari in Opera nista podprta. Skupaj 40 %.
- Ranorex: Opera ni podprta. Skupaj 45 %.
- SmartBear TestComplete: Safari ni podprt. Skupaj 45 %.
- froglogic Squish: Vsi. Skupaj 50 %.

Kritični element avtomatizacije testiranja so t. i. lokatorji, ki prepoznajo element oz. njegovo lokacijo glede na njegove značilnosti. Orodja jih imenujejo različno, npr. strategija iskanja, iskalnik ali identifikator kontrole. Ključni koncept je enak za vsa orodja in poimenovanja: kako skripte orodja najdejo stvari na spletni strani, s katerimi morajo interaktirati. Pri tem se zanašajo na strukturo strani ter na lastnosti elementa, včasih kar na oboje skupaj. Slabi lokatorji povzročajo probleme, saj se v primeru, da se spremeni lokacija elementa na strani, testi ne izvajajo uspešno. Ravno tako se testi se izvedejo neuspešno, kadar se v uporabniškem vmesniku pojavi nov element, ki spremeni podatke (npr. nova vrstica v tabeli, ki bere podatke iz baze), testi so neuspešni zaradi dinamične generiranosti lokatorjev. Zelo pomembno je, kako so lokatorji shranjeni. Nujno je, da obstaja centralizirana definicija lokatorjev, prav tako se je potrebno izogniti njihovemu podvajanju. (Holmes, 2014, 4. december).

Obstaja več vrst lokatorjev, ki se po navadi ločijo glede na način oz. vrsto elementa, ki ga iščejo (Holmes, 2014, 4. december):

- **Identifikator** (ang. *ID*, v nadaljevanju ID): je vrednost atributa HTML elementa. Ni nujno, da je ID nastavljen za vsak element posebej, čeprav je po standardih HTML to zaželeno. Prednost tega je, da ima vsak element svoj unikaten ID. Skripte lahko najdejo

ta element ne glede na spremembe na strani ter ne glede na to, ali se spremeni lokacija elementa, ki ga iščemo. Vsak brskalnik ID obdela zelo hitro. Ne glede na vse prednosti to lahko naredi izdelavo testnih scenarijev težko in ne tako robustno, še zlasti v primerih, ko so ustvarjeni dinamično bodisi zaradi platforme bodisi zaradi kontrol ali razvijalcev.

- **Prekrivni slogi** (ISlovar, b.l.) (ang. *Cascading Style Sheet*, v nadaljevanju CSS): uporablja se jih za definiranje stilov na strani. Tako lahko več HTML elementov uporablja enak stil ali CSS razred, kar olajša izdelavo grafičnega vmesnika strani. V določenih primerih se lahko zgodi, da ima kateri od elementov spletne strani unikaten CSS stil. Njihove atribute se lahko uporablja za iskanje. Prav tako se njihov način zapisa lahko uporablja za iskanje drugih elementov, tudi takih, ki nimajo nastavljenega CSS. Brskalniki jih obdelajo skoraj tako hitro kot ID, ki so zelo fleksibilni in jih je relativno enostavno brati. Na žalost lahko iskanje poteka samo navzdol po strukturi strani.
- **XPath**: morda gre za eno izmed najbolj slabo razumljenih in uporabljenih, a tudi najzmogljivejših tehnologij, ki omogoča prepoznavo vrednosti elementov, navigacijo gor, čez in dol po strukturi strani in za ustvarjanje zelo zmogljivih strategij za lokatorje. Zahteva veliko časa za vzdrževanje. Je najpočasnejši lokator, še zlasti za Internet Explorer. Glavni problem tega je, da se začne v »korenini« HTML strani in uporablja fiksne indekse za prepoznavo specifičnih elementov iz skupine elementov, kar pomeni, da skoraj vsaka sprememba v spletni strani povzroči, da lokator ne bo več deloval in bo izvedba testna neuspešna. XPath je tako ob pravilni in previdni uporabi lahko zmogljivo orodje, v nasprotnem primeru lahko povzroči veliko težav.
- **Vsebina besedila**: to je uporabno zlasti v prikazih tabel in vrstičnem izpisu podatkov, saj včasih ne želimo iskati le po fiksni lokaciji oz. indeksu elementa, pač po vsebini, ki se smiselno prilagaja (npr., če želimo testirati tabelo s podatki uporabnikov na spletni strani je velika verjetnost, da se bo njihov vrstni red spreminjal. Zato je smiselneje iskati po sami vsebini, npr. določeno uporabniško ime uporabnika).
- **Ostali atributi**: iskanje po slikah, spletnih povezavah in imenskih atributih.

Zelo pomembni so tudi asinhroni klici ali t. i. AJAX tehnologije. Namesto da bi brskalnik naložil celotno stran, ponovno naloži samo del strani. To prihrani določene resurse in ne »lomi« uporabniške izkušnje. Določeni elementi lahko v tem času izginejo iz strani. Ena od možnosti je, da se fiksno določi, koliko časa naj skripta čaka na določen element, boljša rešitev je, da se uporabi pogojna čakanja, ki trajajo, dokler določen element ni prisoten na spletni strani. To naredi skripto optimalnejšo in razumljivejšo ostalim, saj se lahko lažje ugotovi, zakaj skripta čaka (Holmes, 2014, 4. december).

Regularni izraz (ang. *regular expression*, v nadaljevanju RegEx) je poseben načina zapisa iskalnega vzorca, ki je nekakšna naprednejša verzija uporabe nadomestnih znakov. Lahko se uporablja tako za iskanje kot za preverjanje vnesenih podatkov, npr., ali je vneseni elektronski naslov res elektronski naslov (Regular-Expressions.info, 2015, 17. avgust).

Ocene orodij za avtomatizacijo glede na vrste lokatorjev, ki jih podpirajo

- Selenium (Web Driver in Selenium IDE): ID, ime razreda, ime značke, ime, spletno povezavo, del spletne povezave, CSS, XPath, JavaScript[5]. Vseh načinov: 9.
- Borland (Micro Focus) Silk Test: ID, XPath, značka, slika, besedilo[6]. Vseh načinov: 5.
- Ranorex: XPath, slike, RegEx, Javascript[7]. Skupaj: 4.
- S bmartBear TestComplete: vse, ki jih tudi Selenium, slike[8]. Skupaj: 10.
- froglogic Squish: Skupaj: ID, ime razreda, XPath, CSS, Javascript, besedilo, spletno povezavo, del spletne povezave, ime značke, slike[9]. Skupaj: 10.

Pri tem velja omeniti, da določena orodja, kot je na primer Ranorex, omogočajo razširitev načinov iskanja preko funkcij, ki jih uporabnik sprogramira sam oz. jih poišče med primeri podjetja. XPath lahko najde tudi CSS elemente. Bolj merodajno bi bilo primerjanje zmogljivosti orodij in njihovih lokatorjev na različnih praktičnih primerih, kar presega obseg te magistrske naloge.

Tehtano povprečje lokatorjev je 50 %. SmartBear TestComplete ima lokatorjev ravno 10, kar torej znese 50 % in končno oceno 5.

Tabela 2: Ocena združljivosti orodja s sistemom

Orodje	Brskalniki v %	Lokatorji	Lokatorji v %	Skupaj v %	Skupaj
Selenium	45 %	9	45 %	90 %	9
Borland	40 %	5	25 %	65 %	6.5
Ranorex	45 %	4	20 %	65 %	6.5
SmartBear	45 %	10	50 %	95 %	9.5
froglogic	50 %	10	50 %	100 %	10

Vir: Lastni izračun

5.1.3 Integracija s kartičnim sistemom JIRA

Kot sem omenil že v prejšnjih poglavjih, podjetje uporablja kartični sistem podjetja Atlassian, JIRA, ki je nekakšen krvožilni sistem podjetja. Brez ustrezne kartice se naj ne bi zgodila nobena sprememba v kodi. Poleg tega stranke podjetja uporabljajo ta sistem za

[5] Vir: *Selenium HQ*. Najdeno 17. avgusta na spletnem naslovu http://docs.seleniumhq.org/docs/03_webdriver.jsp
[6] Vir: *Micro Focus*. Najdeno 17. avgusta na spletnem naslovu http://documentation.microfocus.com/help/index.jsp?topic=%2Fcom.borland.silktest.classic.doc%2FSTCLASSIC-A055AD74-DYNAMIC-OBJECT-RECOGNITION.html
[7] Vir: *Ranorex*. Najdeno 17. avgusta na spletnem naslovu http://www.ranorex.com/support.html
[8] Vir: *SmartBear*. Najdeno 17. avgusta na spletnem naslovu http://smartbear.com/product/testcomplete/web-module/overview/
[9] Vir: *froglogic*. Najdeno 18. avgusta na spletnem naslovu http://doc.froglogic.com/squish/latest/rgs-webconvenience.html

delo s podporo. V njem so shranjeni vsi testni scenariji, zahteve strank in določene specifikacije. Uporablja se tudi za interno delo, kot je kreiranje kartic za nove graditve sistema, dodeljevanje pravic uporabnikom, sporočanje težav in drugo delo, ki je povezano s sistemskimi administratorji in računalniškimi tehnikami.

Ob vpeljavi avtomatizacije testiranja je smiselno, da bi avtomatizirano orodje samodejno prenašalo podatke o izvajanju testnih scenarijev v sistem JIRA in kreiralo poročila o hroščih. V pilotni fazi je še bolje, da bi programsko orodje izdelalo osnutek, ki bi ga nato potrdil uporabnik orodja. V primeru napačne konfiguracije ali veliko napak se namreč lahko zgodi, da bi bil sistem JIRA »zasut« z množico kartic.

Integracija s sistemom JIRA je možna na več načinov, bodisi preko vnosne maske orodja za avtomatizacijo (kar je neposredna integracija) bodisi preko programskih oz. spletnih vmesnikov. Določne nastavitve, kot so ime strežnika, uporabniškega imena in gesla, je potrebno vedno nastaviti ročno. Pri spletnih vmesnikih je po navadi potreben klic razreda, ki je javno dostopen. Pri nastavljanju programskih vmesnikov je običajno potrebno uporabiti spletne knjižnice ali jih ročno sprogramirati. Možno je tudi pripraviti izvoz poročil iz enega orodja v obliki, ki je berljiv v drugem orodju.

Integracijo sem tako ocenjeval na naslednje načine; v kolikor jo orodje podpira preko grafičnega vmesnika ob inštalaciji, brez programiranja, je dobilo orodje 10. V kolikor je podpora mogoča preko spletnih ali programskih vmesnikov, je orodje lahko dobilo največ 6, glede na to, kako zahtevna je implementacija in koliko kodiranja potrebuje. V primeru, da ni razvidno, ali je to mogoče, je ocena 0.

- **Selenium (Web Driver in Selenium IDE):** preko JIRA programskih vmesnikov. Možnosti za to integracijo je več, možna je tudi integracija knjižnice JIRA[10]. Končna ocena: 6.
- **Borland (Micro Focus) Silk Test**: mogoče le z nakupom orodja Silk Central[11]. Končna ocena: 0.
- **Ranorex**: za integracijo obstaja uporabniški vmesnik, ki zahteva nekaj znanja kodiranja[12]. Končna ocena: 8.
- **SmartBear TestComplete**: Omogoča integracijo preko grafičnega vmesnika. Možno je klikniti na gumb »Vnesi hrošč v JIRA«. Ni jasno razvidno, ali je to mogoče tudi avtomatično[13]. Od vseh orodij deluje ta integracija najbolje. Končna ocena: 9.
- **froglogic Squish**: uporablja orodje, ki pretvarja XML datotečna poročila v kartice v JIRA. To je možno tudi samodejno. Zahteva nekaj nastavitev in programiranja preko ukazne vrstice.[14] Končna ocena: 9.

[10] Vir: več spletnih strani najdenih preko spletnega brskalnika Google.
[11] Vir: *Borland*. Najdeno 18. avgusta na spletnem naslovu http://www.borland.com/Products/Software-Testing/Automated-Testing/Silk-Test/Integration
[12] Vir: *Ranorex*. Najdeno 18. avgusta na spletnem naslovu http://www.ranorex.com/blog/integrating-ranorex-test-cases-into-jira
[13] Vir: *SmartBear*. Najdeno 18. avgusta na spletnem naslovu https://support.smartbear.com/viewarticle/76131/

5.1.4 Enostavnost za uporabo

To je eden od kriterijev, ki je najbolj subjektiven, hkrati ga je, vsaj na začetku, težko ocenjevati. Uporabniki se med seboj razlikujemo tako po izkušnjah pri uporabi (podobnih) programskih orodjih kot tudi po sposobnostih privajanja oz. učenja novih orodij, zahtevnosti, mnenjih o videzu in osebnih preferencah. Obstajajo določeni industrijski standardi, ki se jih ne držijo vsa podjetja enako ali skušajo celo uveljaviti svoja.

Orodja za avtomatizacijo testiranja so si po funkcionalnosti dokaj podobna, razlikujejo se v tem, kolikšen nivo znanja programiranja potrebuje uporabnik, da jih lahko upravlja, ter v tem, kako hitro je uporabnik »prisiljen« zapustiti »snemalna« orodja in se lotiti kodiranja. Po izkušnjah, ki jih imam z orodji za avtomatizacijo poslovnih procesov, je potrebno relativno hitro uporabiti vsaj skriptni programski jezik. Po navadi je to v primerih, ko želimo avtomatizirati procese, ki zajemajo dinamične vsebine in podatke.

Za realnejšo oceno uporabniškega orodja je potrebno orodje preizkušati na daljši rok z večimi uporabniki, ki nato ocenijo orodja vsak zase, zatem se napravi diskusija. Glede na prej opisane razlike med uporabniki kot tudi glede na njihove lastne interese, se nato skuša izdelati realnejšo sliko orodja. Nenazadnje so pomembna tudi pričakovanja, ki jih imamo od orodja.

To oceno sem razdelil na 2 dela. 70 % končne ocene predstavlja ocenjevanje »posnemi in predvajaj« funkcionalnosti. Tu sem skušal oceniti tako nivo znanja programiranja, ki ga uporabnik potrebuje, kot tudi izgled. 30 % končne ocene sestavljajo pristopi k avtomatizaciji, ki jih orodje podpira. Ta ne zahtevajo specializiranega orodja, da bi izvedli določen pristop, vendar lahko orodja poenostavijo začetek pri uvajanju pristopa oz. ga zapletejo. Gre bolj za to, kako je organiziran proces razvoja in testiranja v podjetju. Podjetje lahko to doseže z drugimi orodji. Pristopi, ki jih orodja najpogosteje pokrivajo, so pristopi usmerjeni na podlagi podatkov, ključnih besed ali obnašanja.

Ocena orodja »posnemi in predvajaj«:

- **Selenium (Web Driver in Selenium IDE)**: Web Driver nima nobenega uporabniškega vmesnika. Selenium IDE je vtičnik za FireFox, kar omogoča enostavno namestitev in uporabo, vendar je žal na voljo samo za ta brskalnik. Sam snemalnik je dokaj enostaven, saj omogoča le osnovne funkcionalnosti, kot so »posnemi in predvajaj«, shrani, izvozi v različne programske jezike in razhroščevanje. Določenega koraka ni mogoče onemogočiti. Omogoča samoizpolnjevanje ukazov[15]. Glavne odlike so tako enostavna namestitev, velika množica ukazov ali akcij s kratkimi razlagami in primeri, ter razširljivost. Glavna slabost je, da je omejen samo na en brskalnik. Web Driver nima uporabniškega vmesnika in zahteva znanje programiranja. Na voljo so tudi

[14] Vir: *froglogic*. Najdeno 18. avgusta na spletnem naslovu http://doc.froglogic.com/squish/5.1/rg-cmdline.html#rgc-xml2jira
[15] Vir: *SeleniumHQ*. Najdeno 18. avgusta na spletnem naslovu http://www.seleniumhq.org/projects/ide/

ogrodja, ki naredijo kreiranje skript enostavnejše. Posnete skripte se lahko uvozi v praktično katero koli razvojno orodje. Končna ocena: 6.

- **Borland (Micro Focus) Silk Test**: skripto je mogoče posneti enkrat in jo uporabiti v vseh brskalnikih, ki jo podpira. Uporabniški vmesnik se lahko prilagodi poslovni vlogi uporabnika. Lahko se integrira z Eclipse in Visual Studio razvojnim okoljem. Podpira kreiranje delovnih tokov.[16] Vmesnik deluje »prečiščen« in enostaven za uporabo. Končna ocena: 8.
- **Ranorex**: Enostaven vmesnik. Ob napaki se samodejno naredi zaslonska slika brskalnika. Podatki se samodejno prenesejo tudi v skriptno »shrambo«. Snemalno orodje lahko deluje kot samodejno orodje ali kot del celotnega paketa. Za dostop do bližnjic je mogoče uporabiti »vroče tipke«. Skripte je možno izvoziti v C# in VB.NET, ki sta za izbrano podjetje manj primerna. Vse skripte, ki jih orodje generira, so narejene kot zagonske (.exe) datoteke, kar je vprašljivo glede na varnostno politiko podjetja[17]. Zaradi slednjih 2 pomanjkljivosti je končna ocena nižja: 7.
- **SmartBear TestComplete**: eno skripto lahko posnamemo v enem brskalniku in teče na vseh. Samodejno naredi zaslonske slike ob snemanju, ki jih samodejno posodablja v kolikor ugotovi, da so nastale kakšne spremembe. Uporablja objektni pristop za robustnejše snemanje, pri čemer ne posname samo lastnostni objekta, ampak tudi njegove akcije. Omogoča postavitev kontrolnih točk, ki lahko med izvajanjem testa izvedejo različna primerjanja, kot je npr. primerjava 2 slik. Omogoča izvoz skript v različne programske jezike. Ne podpira ne Eclipse razvojnega orodja ne Jave[18]. Končna ocena: 8.
- **froglogic Squish**: uporabniški vmesnik je dejansko vtičnik za Eclipse. Ne omogoča izvoza v Javo. Omogoča postavljanja točk za preverjanje.[19] Končna ocena: 6.

Pristopi k avtomatizaciji:

- **Selenium (Web Driver in Selenium IDE)**: sam po sebi ne podpira nobenega pristopa. S konfiguracijskimi datotekami, nastavitvami in odprtokodnimi razširitvami ga je možno pripraviti za podporo vsem 3 zgoraj naštetim (vir). Končna ocena: 8.
- **Borland (Micro Focus) Silk Test**: Podpira samo testiranje na podlagi ključnih besed. Končna ocena: 3.
- **Ranorex**: Samo podatkovno usmerjeno. Končna ocena: 3
- **SmartBear TestComplete**: podatkovno usmerjeno in po ključnih besedah. Končna ocena: 6.
- **froglogic Squish**: obnašanje in podatkovno usmerjeno. Končna ocena 6.

[16] Vir: *Borland.* Najdeno 18. avgusta na spletnem naslovu http://www.borland.com/Products/Software-Testing/Automated-Testing/Silk-Test/Features
[17] Vir: *Ranorex*. Najdeno 18. avgusta na spletnem naslovu http://www.ranorex.com/test-automation-tools.html#c6437
[18] Vir: *SmartBear*. Najdeno 18. avgusta na spletnem naslovu http://smartbear.com/product/testcomplete/web-module/overview/
[19] Vir: *froglogic*. Najdeno 18. avgusta na spletnem naslovu http://doc.froglogic.com/squish/latest/

Katere pristope posamezno orodje podpira, sem zapisal pri opisih orodij.

Tabela 3: Ocena enostavnosti za uporabo

Orodje	Snemalnik	Snemalnik v % (70 %)	Pristopi	Pristopi v % (30 %)	Skupaj v %	Ocena
Selenium	6	42 %	8	24 %	66 %	6.5
Borland	8	56 %	3	9 %	65 %	6.5
Ranorex	7	49 %	3	9 %	58 %	6
SmartBear	8	56 %	6	18 %	74 %	7.5
froglogic	6	42 %	6	18 %	60 %	6

Vir: Lastni izračun

5.1.5 Vzdrževanje skript

Vzdrževanje skript je kriterij, ki ga je predvsem na začetku, pred izbiro orodja, verjetno najtežje ocenjevati, in kriterij, ki se ga pogosto zanemarja. Na dolgi rok ravno vzdrževanje testnih skript lahko pomeni razliko v uspešnosti in učinkovitosti projekta avtomatizacije testiranja. V začetnih uporabah orodij se najprej ukvarjamo z namestitvijo in učenjem osnovnih funkcij. Vprašanje vzdrževanja se pojavi šele kasneje, ko z orodjem ustvarimo že kar nekaj skript in ugotovimo, da vzdrževanje zahteva vedno več časa. Podobno sem opazil sam pri delu z orodjem za avtomatizacijo testiranja poslovnih procesov, Network Automation AutoMate[20], kjer je vzdrževanje sčasoma postajalo vedno težje in je zahtevalo vedno več časa.

Dobro vzdrževanje bi moralo omogočiti enostavno onemogočenje skript, ki se jih ne potrebuje več. Ta status bi moral biti jasno razviden, najbolje kar s spremenjeno ikono skripte ali česa podobnega v grafičnem vmesniku. Večina orodij omogoča izvoz skript, ki jih je nato mogoče prenesti na disk ali sistem verzioniranja kode (ali repozitorij), kjer to strukturo lahko oblikujemo po svoje. To je zlasti uporabno v primerih, kjer je repozitorij obravnavanega podjetja že dolgo uveljavljen in je integriran v sistem JIRA. Iz njega je možno videti, kdo je zadnji naredil spremembe, kje v datotekah so se te spremembe izvedle, možno je navesti tudi komentarje sprememb. V kolikor ima orodje že vgrajen tak repozitorij, je veliko boljše. Slabe zmožnosti orodja za vzdrževanje to lahko naredijo še težje in obratno, odvisno je tudi od nas samih, kako organiziramo skripte. Ta premislek velja narediti na začetku, pred uporabo orodja, in določiti pravila, ki bi se jih morajo držati vsi uporabniki orodja, saj je le v tem primeru mogoče dosegati boljše rezultate.

Skripte je potrebno, v kolikor to orodje ne dela samo, optimizirati. V idealnem primeru skripte uporabljajo skupne razrede, ki se jih spreminja samo na enem mestu. To bistveno zmanjša čas, potreben za spremembe.

[20] http://www.networkautomation.com/automate/automate/

- **Selenium (Web Driver in Selenium IDE):** ne IDE ne Web Driver ne vključujeta nikakršnega repozitorija. Pri obeh smo bolj ali manj prepuščeni sami sebi, kako in kam bomo shranjevali skripte ter kaj bomo z njimi naredili. Velika svoboda je hkrati prednost za bolj izkušene uporabnike orodja in slabost za manj izkušene. V primeru izbranega podjetja to razumem kot prednost, saj lahko podjetje uporabi popolnoma svoj pristop k pisanju in vzdrževanju skript, ki bo podjetju najbolj odgovarjala. Glavna slabost je, ker ni, razen razvijalskega orodja, nobenega grafičnega vmesnika za pregled skript. Končna ocena: 6.
- **Borland (Micro Focus) Silk Test**: s podporo testiranja na temelju ključnih besed uporabe ene skripte za vse brskalnike ter poljubnemu razvojnemu orodju, kot je Eclipse, je vzdrževanje relativno enostavno. Več informacij iz virov, ki so na voljo, nisem mogel pridobiti. Končna ocena: 7.
- **Ranorex**: Podpira testiranje na temelju ključnih besed, vključuje lasten repozitorij, v katerem shranjuje tudi vse objekte, ki so ustvarjeni med snemanjem in jih je možno uporabiti večkrat, brez ponovnega »snemanja«. Končna ocena: 8.
- **SmartBear TestComplete**: podpira snemanje na temelju ključnih besed, integracijo z zunanjimi repozitorji in repozitorij objektov. Končna ocena: 8.
- **froglogic Squish**: podpira testno usmerjen pristop na podlagi obnašanja, uporablja t. i. načrt objektov uporabniškega vmesnika in uporabniški vmesnik na temelju Eclipse. Končna ocena: 7.

5.1.6 Integracija z razvojnim okoljem

Obravnavano podjetje razvija spletne programske rešitve v programskem jeziku Java. To je glavni programski jezik podjetja, v katerem je razvitih največ funkcionalnosti. Glavno razvojno orodje je Eclipse, odprtokodna rešitev. Določeni razvijalci uporabljajo tudi NetBeans orodje, ki je prav tako odprtokodna rešitev. V Java programskem jeziku so napisane vse funkcije, ki se izvajajo na strežnikih. Za prikaz dinamičnih vsebin in funkcij se uporabljata Javascrtipt in AJAX. Management podjetja je odprt glede programskega jezika, ki naj bi se uporabljal za avtomatizacijo. Sam sem mnenja, da glede na to, da ima velika večina programerjev izkušnje v Javi, imajo prednost orodja, ki podpirajo izvoz v ta programski jezik in razvijalsko orodje Eclipse. Nekdo, ki je vešč objektnega programskega jezika, se bo lažje naučil podobnega kot popoln začetnik. Alternativa je tudi C#, ki je Javi zelo podoben, saj je bil razvit kot nekakšno ogrodje zanjo. Primeren je tudi Javascript, vendar ima podjetje manj funkcij, spisanih v njem. Najvišje ocene sem tako dal orodjem, ki podpirajo Javo, Eclipse in Javascript. V primeru C# nižjo, pri ostalih še manj. Pri integraciji z razvijalskim orodjem je pomembno tudi, kako dobro se integrira z repozitorijem kode. Vsa orodja podpirajo izvoz skript, zato tega nisem posebej ocenjeval.

- **Selenium (Web Driver in Selenium IDE):** podpira Javo, Javascript in se integrira kot knjižnica v Eclipse. Končna ocena: 10.

- **Borland (Micro Focus) Silk Test**: Java in Eclipse.[21] Končna ocena: 8.
- **Ranorex**: Visual Stduio, C#.[22] Končna ocena: 3.
- **SmartBear TestComplete**: Eclipse in Javascript.[23] Končna ocena: 6.
- **froglogic Squish**: Eclipse in Javascript.[24] Končna ocena: 6.

5.1.7 Dokumentacija, viri za trening in izobraževanje

Tu sem oceno razdelil na 3 dele. 40 % predstavlja število zadetkov v brskalniku Google, kar lahko pokaže tudi, kako »popularno« je orodje. 40 % predstavlja urejenost uradne spletne strani podjetja, kako lahko je najti določne informacije, kako so te prikazane, ali podjetje nudi kakšna dodatna izobraževanja in kakšni so te stroški ter 20 % število zadetkov v spletni knjigarni Amazon.

Prvi in zadnji kriterij sta bolj kvantitativna, ocenjevanje dokumentacije je bolj subjektivno. Kljub temu večina ljudi v primeru iskanja dodatnih informacij najprej obišče spletno stran podjetja, ki je lahko tudi edini, bolj urejen, vir informacij.

Število zadetkov:

- **Selenium (Web Driver in Selenium IDE**): 823.000 za Web Driver. 499.00 za IDE. Iskanje zgolj po besedi »Selenium test«, brez upoštevanja »seleniuma« kot kemijskega elementa vrne več kot 17 milijonov zadetkov. Končna ocena: 10.
- **Borland (Micro Focus) Silk Test**: 74.000. Končna ocena: 1.
- **Ranorex**: 189.000. Končna ocena: 2.
- **SmartBear TestComplete**: 123.000. Končna ocena: 2.
- **froglogic Squish**: 72.100. Končna ocena: 1.

Ocena spletne strani in dokumentacije:

- **Selenium (Web Driver in Selenium IDE)**: pregledna, s praktičnimi primeri. Manjkajo slike in videovsebine. Za iskalnik se uporablja Google. Končna ocena: 6.
- **Borland (Micro Focus) Silk Test**: Glavni problem je predvsem, da so spletne strani razdrobljene v različnih formatih, med partnersko spletno stranjo podjetja MicroFocus in spletnimi stranmi Borland. Z nekaj truda se najde dostop do spletne vsebine, spletne treninge in dokumentacijo, ki ni najbolj posodobljena. Končna ocena: 6.
- **Ranorex**: najbolje urejeno med vsemi orodji, pregledne spletne strani z enostavno navigacijo, veliko slikami, praktičnimi primeri, video vsebinami in pogosto zastavljenimi vprašanji. Končna ocena: 10.

[21] Vir: *Borland*. Najdeno 20. avgusta na spletnem naslovu http://www.borland.com/Products/Software-Testing/Automated-Testing/Silk-Test/Features/Cross-browser-testing-made-easy
[22] Vir: *Ranorex*. Najdeno 20. avgusta na spletnem naslovu http://www.ranorex.com/products/ui-testing-with-net-and-visual-studio.html
[23] Vir: *SmartBear*. Najdeno 20. avgusta na spletnem naslovu http://www.ranorex.com/products/ui-testing-with-net-and-visual-studio.html
[24] Vir: *froglogic*. Najdeno 18. avgusta na spletnem naslovu http://www.froglogic.com/squish/gui-testing/features/index.php?id=varietyOfScriptingLanguages.html

- **SmartBear TestComplete**: pregledne spletne strani, slike in videovsebine. Dostop do glavnih vsebin sicer zahteva registracijo, ima tudi forum, ki ni najbolj pregleden. Končna ocena: 8.
- **froglogic Squish**: video vsebine, slike, praktični primeri in blog. Podjetje uporablja kot glavno barvo zeleno in njene odtenkev, kar naredi določne vsebine rahlo nepregledne. Končna ocena: 7.

Knjigarna Amazon (razdelek knjige):

- **Selenium (Web Driver in Selenium IDE):** 3.206. Končna ocena: 10.
- **Borland (Micro Focus) Silk Test**: 0. Končna ocena: 0.
- **Ranorex**: 4. Končna ocena: 1.
- **SmartBear TestComplete**: 1. Končna ocena: 1.
- **froglogic Squish**: 0. Končna ocena: 0.

Tabela 4: Ocena dokumentacije

Orodje	Google	Google v % (40 %)	Splet	v % (40 %)	Amazon	v % (20%)	Skupaj v %	Ocena
Selenium	10	40 %	6	24 %	10	20 %	84 %	8.5
Borland	1	4 %	6	24 %	0	0 %	28 %	3
Ranorex	2	8 %	10	40 %	1	2 %	50 %	5
SmartBear	2	8 %	8	32 %	1	2 %	42 %	4
froglogic	1	4 %	7	28 %	0	0 %	32 %	3

Vir: Lastni izračun

5.1.8 Uporabniška podpora

V tem podpoglavju sem ocenjeval podporo, ki jo uporabnikom nudijo podjetja. Ta lahko zajema različne načine komuniciranja z uporabniki, kot sta telefon in elektronska pošta, ter časovna dostopnost. Omejena je lahko na nekaj ur na leto, odvisno od pogodbe, ali je neomejena. Nadalje je tu možnost dodatnih treningov in izobraževanj in certificiranja za uporabnike. Uporabniki lahko pomoč dobijo tudi preko spletnih forumov ali od drugih uporabnikov. Delno se prekriva s prejšnjim podpoglavjem, ker lahko ob primeru enostavne dostopnosti do informacije in dobro organizirane dokumentacije uporabnik sploh ne rabi kontaktirati klicnega centra.

- **Selenium (Web Driver in Selenium IDE):** ima spletni forum, ki uporablja Google groups, ki ni najbolj pregledna rešitev in klepetalnico preko IRC kanala. Na voljo je tudi kartični sistem za vnos težav. Pri tem je potrebno poudariti, da so vsi ti načini pomoči brezplačni in neprofesionalni, gre za podporo skupnosti, ki je zaradi popularnosti orodja velika. Vprašljiv je seveda odziv na težavo, saj uporabnikov ne

veže nobenega pogodba. Na voljo je tudi plačljiva pomoč, o kateri nisem pridobil podatkov.[25] Končna ocena: 6.

- **Borland (Micro Focus) Silk Test**: na voljo so različne vrste pomoči, ki so razdeljene v razrede, platinast, zlat, srebrni in »razširjen«, ki je namenjen starejšim produktom podjetja. Prva 2 nudita pomoč 24 ur na dan, vse dni v letu. Vsi nudijo 60 % vseh nadgradenj, obveščanje o novostih po elektronski pošti in kartični sistem za vnos in spremljanje težav. Podatkov o cenah ni. Prav tako ne, ali je kateri od teh paketov na voljo že ob nakupu določene programske opreme. Klicne številke so na voljo za več kot 40 držav, vključno s tisto, v kateri se nahaja obravnavno podjetje. Na voljo je tudi forum in spletni trening, namenjen uporabnikom.[26] Končna ocena: 10.
- **Ranorex**: na voljo je pregledni forum, trening in certificiranje uporabnikov. Telefonske podpore ni, le 2 klicni številki podjetij v Avstriji in ZDA. V primeru težav je potrebno uporabiti forum ali poslati preko elektronske pošte oz. vnosne maske na spletni strani.[27] Končna ocena: 7.
- **SmartBear TestComplete**: Na voljo je spletni forum, ki ni najbolj pregleden in dostop do pomoči preko prijave v spletni programski rešitvi in oddaje podatkov. Uporabnik je nato kontaktiran preko elektronske pošte. Na voljo je več telefonskih številk, vendar ni podatkov o dosegljivosti.[28] Končna ocena: 8.
- **froglogic Squish**: foruma ni. Pomoč je dostopna preko elektronske pošte. Na voljo so tudi »spletni sestanki« in pomoč po telefonu. Oboje spada pod »premium« storitve in je plačljivo po uri. Na voljo je le ena telefonska številka, v Nemčiji, preostale so povezane s prodajo.[29] Končna ocena: 6.

5.1.9 Izdelava poročil

Poročila o testiranju so eden izmed najpomembnejših produktov dela preizkuševalca programske opreme. Z njimi sporoča testerjem ter ostalim udeležencem v razvoju programske opreme, kot so razvijalci in managerji, pomembne informacije o produktu, ki je testiran in tudi o samem procesu testiranju. Na podlagi teh informacij ostali udeleženci sprejmejo odločitve, katerih kvaliteta je delno odvisna od kvalitet samih poročil.

Testerji si s poročilom gradijo svoj ugled. V kolikor bo poročilo zahtevalo preveč dodatnega truda in časa (po mnenju razvijalca nepotrebnega) za razvijalca, se bo ta morda pričel »izogibati« testerja. Slabo napisana poročila lahko v managerju vzbudijo občutek, da tester napihuje majhne stvari, da otežuje druge ter da ni vložil zadosti časa v raziskavo problema. Dolžnost testerja bi zato morala biti, da hrošče sporoča na način, ki je natančen in lahko razumljiv. Tester mora postati nekakšen zagovornik odprave hrošča, poročilo je

[25] Vir: *SeleniumHQ*. Najdeno 20. avgusta na spletnem naslovu http://www.seleniumhq.org/support/
[26] Vir: *Borland*. Najdeno 20. avgusta na spletnem naslovu http://www.borland.com/Products/Software-Testing/Automated-Testing/Silk-Test/Features/Cross-browser-testing-made-easy in http://www.borland.com/Support
[27] Vir: *Ranorex*. Najdeno 20. avgusta na spletnem naslovu http://www.ranorex.com/support.html
[28] Vir: *SmartBear*. Najdeno 20. avgusta na spletnem naslovu http://www.ranorex.com/support.html
[29] Vir: *froglogic*. Najdeno 20. avgusta na spletnem naslovu http://www.froglogic.com/services/#Support_Services

njegovo »prodajno« orodje, s katerim prepriča ostale udeležence, da je napaka vredna popravila in dodatno vloženega časa in truda. To lahko doseže, tako da navede prednosti, ki jih bo ta naloga prinesla, ter da pove, kako pomembno je to za stroške podjetja, kot je npr. navedba stroškov odprave podobnega hrošča v preteklosti (Kaner et al., 2002, str. 65–67).

Za pripravo dobrega poročila je lahko potrebnega veliko časa in truda, kar se s prakso praviloma izboljšuje. Izdelava poročil postaja pomembnejša s časom, ko imamo več avtomatiziranih skript in pogosteje, ko jih zaganjamo. V idealnem primeru na podlagi enega poročila izvemo vse pomembne podatke o zadnjem izvajanju testnih poročil, koliko je bilo uspešnih in neuspešnih ter katere so najbolj kritične napake. Poročila se lahko razlikujejo glede na to, kdo jih bere, tako da bi morala obstajati možnost več različnih vrst poročil. Poročila naj bo lahko izvoziti in poslati po elektronski pošti.

- **Selenium (Web Driver in Selenium IDE)**: privzetih nima nobenih poročil. To je možno nastaviti preko konfiguracije datoteke. Na koncu naredi HTML datoteko, ki jo lahko odpremo v spletnem brskalniku ali jo ročno pošljemo komu drugemu. Preko programskih vmesnikov oz. kodiranja je mogoče narediti več možnosti. Končna ocena: 5.
- **Borland (Micro Focus) Silk Test**: Ni podatkov. Končna ocena: 0.
- **Ranorex**: poročila so izdelana v datotečnem formatu XML. V primeru napak ob zagonu se samodejno izdelajo zaslonske slike za lažje kreiranje poročil o hroščih. Izdelava poročil je neposredno povezana z modulom za analizo spletnih strani in repozitorijem objektov. Poročilo prikazuje vse podatke o določenem testnem ciklu. S klikom na posamezni testni primer je mogoče dobiti podrobnejše informacije o posameznem testnem scenarij, kot so podrobnejše informacije o napakah in statusu. Nastaviti je mogoče različne nivoje pomembnosti informacij poročila tako na nivoju skupka testa kot na posameznem testu. Manj pomembne napake se lahko izključi iz poročila. Nastaviti je mogoče tako informacije, ki bodo v poročilu, kot samo obliko poročila. Preko kode je mogoče nastaviti lastne nivoje poročanja.[30] Končna ocena: 8.
- **SmartBear TestComplete**: podatki so lahko izvoženi v formah HTML, XML, MHT. Poročilom je možno dodati poljubno besedilo in vsebino HTML. Ob napaki se avtomatično naredi zaslonska slika spletne strani. Podatki o izvajanju testov se lahko avtomatično izvozijo v sistem JIRA in nekatera ostala orodja.[31] Končna ocena: 9.
- **froglogic Squish**: testni rezultati so prikazani v grafičnem vmesniku orodja. Z uporabo lastnih kontrol je mogoče izdelati poljubna sporočila, ki se jih lahko shrani v

[30] Vir: *Ranorex*. Najdeno 21. avgusta na spletnem naslovu http://www.ranorex.com/test-automation-tools.html in http://www.ranorex.com/support/user-guide-20/lesson-8-reporting.html
[31] Vir: *SmartBear*. Najdeno 21. avgusta na spletnem naslovu http://smartbear.com/product/testcomplete/features/test-reporting/

podatkovno bazo oz. izvozi v datoteke HTML. Rezultate je mogoče tudi neposredno izvoziti v datoteke XML.[32] Končna ocena: 6.

5.1.10 Drugi podprti tipi testov

Vsa opisana orodja podpirajo testiranje spletnih programskih rešitev in funkcionalno testiranje. Zanimalo me je, ali podpirajo še kakšne druge vrste testiranja, kot je npr. testiranje zmogljivosti in varnostno testiranje. Nekatera orodja podpirajo tudi testiranje mobilnih, vgrajenih ali namiznih programskih rešitev, vendar to za obravnavano podjetja ne prinaša bistvenih koristi, zato jih nisem ocenjeval. To podpoglavje se delno prekriva s podpoglavjem 5.1.4.

Končne ocene so:

- **Selenium (Web Driver in Selenium IDE):** podpira podatkovno usmerjeno testiranje in regresijske teste, z nekaj programiranja oz. konfiguriranja tudi testiranje na podlagi ključnih besed in obnašanja ter načeloma tudi testiranje zmogljivosti.[33] Končna ocena: 7.
- **Borland (Micro Focus) Silk Test**: testiranje na podlagi ključnih besed, regresijsko testiranje.[34] Končna ocena: 6.
- **Ranorex**: testiranje na podlagi ključnih besed in podatkov, regresijsko testiranje.[35] Končna ocena: 7.
- **SmartBear TestComplete**: testiranje na podlagi ključnih besed in podatkov, regresijsko, testiranje na ravni enote (in podpora zunanjim ogrodjem), obremenitveno testiranje, ročno, testiranje bele skrinjice.[36] Končna ocena: 10.
- **froglogic Squish**: testiranje na podlagi obnašanja in podatkov, regresijsko.[37] Končna ocena: 7.

5.2 Rezultati primerjave orodij

Ocene, ki sem jih postavil v prejšnjih podpoglavjih, sem nato vnesel v matriko, da bi dobil končno oceno. Posameznim ocenam sem glede na pogovore v podjetju, lastne izkušnje in teorijo, opisano na začetku tega poglavja, dodelil statistične uteži. Vse izračune sem izvedel s pomočjo programskega orodja Microsoft Excel 2013.

---

[32] Vir: *froglogic*. Najdeno 20. avgusta na spletnem naslovu http://doc.froglogic.com/squish/latest/tutorials-tk.html#test-results-tk.1

[33] Vir: *SeleniumHQ*. Najdeno 23. avgusta na spletnem naslovu http://www.seleniumhq.org/docs/06_test_design_considerations.jsp

[34] Vir: *Borland*. Najdeno 23. avgusta na spletnem naslovu http://www.borland.com/Products/Software-Testing/Automated-Testing/Silk-Test/Features/Complete-test-automation

[35] Vir: *Ranorex*. Najdeno 23. avgusta na spletnem naslovu http://www.ranorex.com/windows-desktop-test-automation.html

[36] Vir: *SmartBear*. Najdeno 23. avgusta na spletnem naslovu http://smartbear.com/product/testcomplete/features/test-reporting/

[37] Vir: *froglogic*. Najdeno 23. avgusta na spletnem naslovu http://www.froglogic.com/squish/gui-testing/features/index.php

Tabela 5: Končni izračun

Kriterij	Utež v %	Selenium	Borland	Ranorex	SmartBear	froglogic
Združljivost s sistemom, ki je predmet testiranja	20,00%	9,0	5,0	4,0	10,0	10,0
Cena in stroški	15,00%	10,0	3,0	3,0	1,0	2,0
Integracija s sistemom JIRA	15,00%	6,0	0,0	8,0	9,0	9,0
Enostavnost za uporabo	10,00%	6,5	6,5	6,0	7,5	6,0
Vzdrževanje skript	10,00%	6,0	7,0	8,0	8,0	7,0
Integracija z razvijalskim okoljem	10,00%	10,0	8,0	3,0	6,0	6,0
Dokumentacija, viri za trening	5,00%	8,5	3,0	5,0	4,0	3,0
Uporabniška podpora	5,00%	6,0	10,0	7,0	8,0	6,0
Izdelava poročil	5,00%	5,0	0,0	8,0	9,0	6,0
Drugi podprti tipi testov	5,00%	7,0	6,0	7,0	10,0	7,0
SKUPAJ	100,00%	**8**	**4,5**	**5,5**	**7**	**6,5**

Vir: Lastni izračun

Kot je razvidno iz tabele, je končni vrstni red ocenjevanih orodij naslednji:

1. **Selenium (Web Driver in Selenium IDE)**. Končna ocena: 8.
2. **SmartBear TestComplete**. Končna ocena: 7.
3. **froglogic Squish**. Končna ocena: 6,5.
4. **Ranorex**. Končna ocena: 5,5.
5. **Borland (Micro Focus) Silk Test**. Končna ocena: 4,5.

Iz tabele in zgornjih rezultatov je razvidno, da je najbolje ocenjeno orodje Selenium. Največja prednost tega orodja, poleg brezplačnosti, je velika skupnost uporabnikov, ki razvijajo orodje in skrbijo za široko dokumentacijo. Orodje ima tako praktično skoraj vse, kar imajo ostala komercialna orodja. Glavna slabost je, da samo orodje brez programiranja in ročnega nastavljanja skoraj ni uporabno. Določen del »brezplačnosti« se tako izgubi ob nastavljanju orodja. Prav tako ni primerno za ljudi, ki nimajo programerskih in tehničnih znanj.

Drugo orodje na preizkušnji je TestComplete podjetja SmartBear. Glavna pomanjkljivost orodja je cena, saj je najdražje orodje na testu. Po drugi strani podpira veliko lokatorjev, ima dobro integracijo s sistemom JIRA, enostavno izdelavo poročil in podpira veliko različnih vrst testov.

Tretje orodje, froglogic Squish, je produkt majhnega podjetja s 30 zaposlenimi, zato določene stvari morda niso tako dodelane kot pri večjih komercialnih orodjih.

Dokumentacija bi lahko bila boljša, predvsem spletna strani rabi boljšo predstavitev funkcionalnosti, ki jih orodje podpira.

Ranorex je zanimivo orodje, ki ni najbolj združljivo z obravnavanim podjetjem. V nasprotnem primeru bi bila ocena verjetno višja. Orodje je preveč usmerjeno na Microsfotovo .NET okolje. Pohvalna je organizacija dokumentacije in pomoči, ki je med najboljšimi na testu.

Zadnje orodje je Borland TestCOmplete. Podobno kot pri froglogic Squis se zdi, da bi podjetje lahko naredilo več za predstavitev funkcionalnosti, ki jih orodje podpira. Ne podpira integracije s sistemom JIRA in brez nakupa dodatne programske opreme ne podpira izdelave poročil.

SKLEP

Na podlagi izbranega podjetja sem predstavil enega izmed načinov izbire orodja za avtomatizacijo testiranja. Podjetje se je za implementacijo takšnega orodja odločilo, ker ročno testiranje predvsem za regresijsko testiranje ni bilo več dovolj učinkovito in ker dolgoročno za zaposlene testerje postaja vse manj zanimivo. Izpostaviti velja še stroškovni vidik, podjetje namreč pričakuje, da bo z investicijami v orodje prihranilo stroške testiranja.

Kot sem omenil že v navajanju avtorjev v uvodu, Graham et al., Weinberg, pri uvajanju takšnega orodja ni enotnega načina ali poti za vsa podjetja. Vsako podjetje je poseben primer. Uspeh in neuspeh se lahko zgodita pri vseh, možnosti za eno in drugo se da izboljšati oz. zmanjšati glede na to, kako se podjetje loti implementacije.

V obravnavanem podjetju so se lotili avtomatizacije rahlo bolj konzervativno in počasi, najprej z izdelavo strategije, s katero so želeli tako najvišjemu managementu kot preostalim zaposlenim v podjetju pokazati, kaj sploh je avtomatizacija testiranja. To zavedanje je pomembno, saj avtomatizacija testiranja ne nadomešča, kvečjemu ga lahko izboljša. V podjetju so si za cilj postavili avtomatizacijo vsaj 25,00 % regresijskih testov do konca leta.

Podjetje je že začelo s pilotnim projektom avtomatizacije, kjer so izbrali orodje Selenium IDE prav zaradi cene in zaradi nekaj zaposlenih v podjetju, ki so z njim že imeli izkušnje. Sedaj so pred odločitvijo, kako izbrati orodje.

Sam sem se izbire orodja lotil, tako da sem jih izbral 5, ki sem jih nato podrobneje analiziral in ocenjeval njihove funkcionalnosti, ob tem sem za izračun končne ocene uporabljal statistične uteži, in sicer glede na pomembnosti posameznega kriterija. Ta način ocenjevanja je mogoče preskočiti in kar takoj pričeti s preizkušanjem vseh 5 orodij. Kljub temu to zahteva še več časa, poleg tega manjka pregled funkcionalnosti in način, kako orodja sploh rangirati.

Podjetju bi tako priporočil, da pilotni projekt nadaljuje z dvema ali tremi orodji. To sta lahko prva 2 na seznamu, Selenium Web Driver in SmartBear Test Complete. Na podlagi tega, kako se orodje izkaže v praksi, je možno postaviti objektivnejšo oceno. Nevarnost je, da samo preizkušanje orodij zahteva preveč časa, s čimer se s končno odločitvijo samo odlaša.

Kljub temu da je izbrano orodje eden izmed pomembnejših dejavnikov za uspešnosti projekta avtomatizacije, ni edini. Niti najboljše orodje ne more rešiti vseh zahtev in problemov, še zlasti ne tistih, ki so povezani s sistemom, ki je predmet testiranja, če sistem za to ni primeren. Omejitve trenutnega stanja sistema so bile vidne v času izdelave dimnih testov, saj vsi moduli, ki jih podjetje razvija, ne glede na starost oziroma tehnologijo, niso primerni za avtomatizacijo. Za izboljšanje tega je potreben čas in investiranje sredstev. Kot sem omenil že v uvodu in podrobneje v poglavju 2.3., »Ključni dejavniki uspeha

avtomatizacije«, izkušnje ostalih podjetij pri uvajanju avtomatizacije kažejo, da je pomembno dobro sodelovanje med testerji in razvijalci programskih rešitev. Pri tem lahko odigra pomembno vlogo tudi management podjetja, ki lahko vzpodbuja in olajša sodelovanje.

Izbira orodja je tako le eden izmed korakov v procesu avtomatizacije testiranja in izboljšanja učinkovitosti testiranja kot izboljšanja kvalitete končnega produkta. Kriteriji, ki sem jih ocenjeval v izbiri, npr. vzdrževanje skript, bodo postajali vedno pomembnejši. Tako bo tudi bolje prišlo do izraza, kako uspešen je bil sam projekt vpeljale avtomatizacije. Obravnavano podjetje ima z vpeljavo avtomatizacije možnost, da testiranje in nenazadnje tudi razvoj programskih rešitev popelje na novo raven, hkrati je avtomatizacija testiranja tudi eden od pogojev za vpeljavo agilnejših metod razvoja.

LITERATURA IN VIRI

1. AbeachA (b.l.). *Automation*. Najdeno 24. avgusta na spletnem naslovu http://www.abeacha.com/abeacha_automation.html

2. Avtomatizirano e-testiranje. (b.l.) V iSlovarju. Najdeno 3. aprila 2015 na spletni strani http://www.islovar.org/izpisclanka.asp?id=9675

3. Bandor, M.S. (2006). Quantitative Methods for Software Selection and Evaluation. Najdeno 12. aprila 2015 na spletnem naslovu http://www.sei.cmu.edu/reports/06tn026.pdf

4. Barber, S. (2007). The A-B-C's of software testing models. *SearchSoftwareQuality*. Najdeno 26. julija na spletnem naslovu http://searchsoftwarequality.techtarget.com/tip/The-A-B-Cs-of-software-testing-models

5. Base36 (2013, 19. marec). *Automated vs. Manual Testing: The Pros and Cons of Each*. Najdeno 9. avgusta na spletnem naslovu http://www.base36.com/2013/03/automated-vs-manual-testing-the-pros-and-cons-of-each/

6. Bordo, V. (b.l.). Overview of User Acceptance Testing (UAT) for Business Analysts (BAs). *Developmentor*. Najdeno 18. julija na spletnem naslovu https://www.develop.com/useracceptancetests

7. Borland (b.l.). Automated Testing. Najdeno 12. aprila 2015 na spletnem naslovu http://www.borland.com/Products/Software-Testing/Automated-Testing

8. Bozhkova, A. (2014, 12. november). The Unexpected Truth About UI Test Automation Pilot Projects. *Telerik*. Najdeno 17. aprila 2015 na spletnem naslovu http://developer.telerik.com/featured/unexpected-truth-ui-test-automation-pilot-projects/

9. Brodie, J. (b.l.). When to automate your testing. *Amsphere Limited*. Najdeno 24. avgusta na spletnem naslovu http://www.amsphere.com/insights/when-automate-your-testing

10. BSD Mag (2014, 6. november). Pros & Cons of Keyword Driven Testing. Najdeno 08. avgusta na spletnem naslovu http://bsdmag.org/pros-cons-of-keyword-driven-testing/

11. Eriksson, U. (2014, 24. september). Differences between the different levels of tests. *ReQtest*. Najdeno 15. junija na spletnem naslovu http://reqtest.com/testing-blog/differences-between-different-test-levels/

12. FedSolutions (2015, 7. julij). 10 Criteria for Selecting Business Software. Najdeno 12. aprila 2015 na spletnem naslovu http://www.fedsolutions.com/2012/10/it-support-10-criteria-for-selecting-business-software/

13. Ferrari, R. (2013, 19. september). Test Automation – Advantages & Limitations. *Olenick.* Najdeno 12. avgusta na spletnem naslovu http://www.olenick.com/advantages-of-automation-for-mobile-testing/

14. Froglogic (b.l.). *Squish Test Automation Suite*. Najdeno 12. aprila 2015 na spletnem naslovu http://www.froglogic.com/index.php

15. F(x) Solutions (2013, 18. maj). Prerequisites for Test Automation. Najdeno 24. avgusta na spletnem naslovu http://www.fofxsolutions.com/2013/05/18/prerequisites-for-test-automation/

16. Ghahrai, A. (b.l.). SDLC Models Archive. *Testing Excellence*. Najdeno 26. julija na spletnem naslovu http://www.testingexcellence.com/category/sdlc-models/

17. Graham, D., & Fewster, M. (2012). Experiences of test automation: case studies of software test automation (2nd ed.). New Jersey: Pearson Education Inc.

18. Guru99. (b.l.). *What is User Acceptance Testing (UAT)?*. Najdeno 18. julija na spletnem naslovu http://www.guru99.com/user-acceptance-testing.html

19. Guru99. (b.l.). *What is System Testing?*. Najdeno 12. julija na spletnem naslovu http://www.guru99.com/system-testing.html

20. Hayes, L.G. (b.l.). The Automated Testing Handbook. *STP*. Najdeno 2. avgusta na spletnem naslovu http://www.softwaretestpro.com/itemassets/4772/automatedtestinghandbook.pdf

21. Hill, S. (2015, 23. februar). The pros and cons of Test-Driven Development. *LeanTesting*. Najdeno 8. avgusta na spletnem naslovu https://leantesting.com/resources/test-driven-development/

22. Holmes, J. (2014, 4. december). Practical UI Test Automation – Locators and Asynchronous Loading. *simple talk.* Najdeno 17. avgusta na spletni strani https://www.simple-talk.com/dotnet/asp.net/practical-ui-test-automation-–-locators-and-asynchronous-loading/

23. ISTQB (2011). Foundation Level Syllabus. Najdeno 7. aprila 2015 na spletnem naslovu http://www.istqb.org/downloads/finish/16/15.html

24. ISTQB (2014). Expert Level Syllabus Test Automation – Engineering. Najdeno 7. aprila 2015 na spletnem naslovu http://www.istqb.org/downloads/finish/18/146.html

25. ISTQB Exam Certification (b.l.). What are the Software Development Models? Najdeno 26. julija 2015 na spletnem naslovu http://istqbexamcertification.com/what-are-the-software-development-models/

26. ISTQB Exam Certification (b.l.). What is integration testing? Njdeno 26. junija 2015 na spletnem naslovu http://istqbexamcertification.com/what-is-integration-testing/

27. Kaner, C., Bach, J., & Pettichord, B. (2002). Lessons learned in software testing: a context driven approach. New York: Wiley Computer Publishing.

28. Levison, M. (2008, 14. oktober). ADVANTAGES OF TDD. *Agile Pain Relief*. Najdeno 08. avgusta na spletnem naslovu https://agilepainrelief.com/notesfromatooluser/2008/10/advantages-of-tdd.html#.VcYeB7Sd7Pk

29. Madaan, D. (2015, 10. marec). BEHAVIOUR DRIVEN TESTING – AN INTRODUCTION. *Women Testers*. Najdeno 9. avgusta spletnem naslovu http://www.womentesters.com/behaviour-driven-testing-an-introduction/

30. Meerts, J., & Graham, D. (2015). The History of Software Testing. Najdeno 25. maja na spletnem naslovu http://www.testingreferences.com/testinghistory.php

31. Narula, S. (b.l.). Test Life Cycle / Software Testing models (manual testing). *Siliconindia*. Najdeno 26. julija na spletnem naslovu http://www.siliconindia.com/online-courses/tutorials/Test-Life-Cycle--Software-Testing-modelsmanual-testing-id-44.html

32. North, D. (b.l.). INTRODUCING BDD. *Dan North & Associates Ltd*. Najdeno 9. Avgusta na spletnem naslovu http://dannorth.net/introducing-bdd/

33. Oladimeji, P. (b.l.). Levels of Testing. *University of Wales Swansea Computer Science Department*. Najdeno 19. junija na spletnem naslovu http://www.cs.swan.ac.uk/~csmarkus/CS339/presentations/20061202_Oladimeji_Levels_of_Testing.pdf

34. Pearson, L. (2013, 31. julij). The Four Levels of Software Testing. *Segue Technologies Inc*. Najdeno 13. maja 2015 na spletnem naslovu http://www.seguetech.com/blog/2013/07/31/four-levels-software-testing

35. Prekrivni slogi (b.l.). V iSlovarju. Najdeno 17. avgusta 2015 na spletni strani http://www.islovar.org/izpisclanka.asp?id=7069&oznaci=1

36. Quality First Software GmbH (b.l.). Overview QF-Test. Najdeno 13. aprila 2015 na spletnem naslovu http://www.qfs.de/en/qftest/index.html

37. Ranorex Gmbh (b.l.). Why Use Ranorex Test Automation Tools. Najdeno 14. aprila 2015 na spletnem naslovu http://www.ranorex.com/test-automation-tools.html

38. Regular-Expressions.info (2015, 17. avgust). *Welcome to Regular-Expressions.info*. Najdeno 18. avgusta 2015 na spletni strani http://www.regular-expressions.info

39. Rice, R. (b.l.). Keys to Successful User Acceptance Testing. *Rice Consulting Services, Inc.*. Najdeno 18. julija na spletnem naslovu http://www.riceconsulting.com/home/index.php/User-Acceptance-Testing/keys-to-successful-user-acceptance-testing.html

40. Schaefer, H. (b.l.). Risk Based Testing, Strategies for Prioritizing Tests against Deadlines. *Methods & Tools.* Najdeno 24. avgusta na spletnem naslovu http://www.methodsandtools.com/archive/archive.php?id=31

41. Selenium (b.l.). Selenium WebDriver. Najdeno 12. aprila 2015 na spletnem naslovu http://docs.seleniumhq.org/docs/03_webdriver.jsp

42. Seetaram (2011, 17. september). Fundamental Concepts of Test Automation. *Selftechy*. Najdeno 29. aprila 2015 na spletnem naslovu http://selftechy.com/2011/09/17/fundamental-concepts-of-test-automation

43. Setter, M. (b.l.). USER ACCEPTANCE TESTING – HOW TO DO IT RIGHT!. Usersnap GmbH. Najdeno 18. julija na spletnem naslovu http://usersnap.com/blog/user-acceptance-testing-right/

44. SmartBear Software (b.l.). TestComplete Platform. Najdeno 13. aprila na spletnem naslovu http://smartbear.com/product/testcomplete/overview/

45. Software Testing Class (2012). System Testing: What? Why? & How?. Najdeno 1. julija na spletnem naslovu http://www.softwaretestingclass.com/system-testing-what-why-how/

46. Software Testing Fundamentals (2011a). Software Testing Levels. Najdeno 15. maja na spletnem naslovu http://softwaretestingfundamentals.com/software-testing-levels/

47. Software Testing Fundamentals (2011b). Software Testing Methods. Najdeno 15. maja na spletnem naslovu http://softwaretestingfundamentals.com/software-testing-methods/

48. Software Testing Fundamentals (2014, 28. junij). Types. Najdeno 15. maja na spletnem naslovu http://softwaretestingfundamentals.com/category/types/

49. Software Testing Help (2015a, 26. junij). A Beginner's Guide to System Testing. Najdeno 15. julija 2015 na spletnem naslovu http://www.softwaretestinghelp.com/system-testing/

50. Software Testing Help (2015b, 26. junij). Automate Testing – How to Choose the Best Automation Testing Tool. Najdeno 15. julija 2015 na spletnem naslovu http://www.softwaretestinghelp.com/choosing-automation-tool-for-your-organization/

51. Software Testing Tools (b.l.). Software Test Automation Tools. Najdeno 24. aprila na spletnem naslovu http://www.testingtools.com/test-automation/

52. Stone, J. (2013, 7. junij). Criteria for Objective Software and Solution Selection. Revenue Architects. Najdeno 28. aprila na spletnem naslovu http://www.revenuearchitects.com/blog/2013/06/07/criteria-for-objective-software-and-solution-selection/

53. TestingBrain (b.l.). Types of Software Testing. Najdeno 19. julija na spletnem naslovu http://www.testingbrain.com/tutorials/types-of-software-testing.html

54. Testiranje. (b.l.) V iSlovarju. Najdeno 2. aprila 2015 na spletni strani http://www.islovar.org/izpisclanka.asp?id= 7388

55. The Software Testing Club (2010). *A Tester Is For Life Not Just for Christmas*. Najdeno 22. aprila na spletnem naslovu http://www.ministryoftesting.com/wp-content/uploads/2010/12/atesterisforlife.pdf

56. Tutorials Point (b.l.). Software Testing – Methods. Najdeno 22. maja na spletnem naslovu http://www.tutorialspoint.com/software_testing/software_testing_methods.htm

57. Vaněk, L. (2010). Automated test methods. Najdeno 16. aprila na spletnem naslovu http://www.fit.vutbr.cz/study/courses/TJD/public/0910TJD-Vanek.pdf

58. Warwick, S. (b.l.). Test Automation as a Development Requirement. *Odin Technology Ltd*. Najdeno 24. avgusta na spletnem naslovu http://www.odintech.com/Downloads_Resources/White_Papers/TestAutomationAsDevelopmentRequirement.pdf

59. W3Schools (b.l.). *Browser Statistics*. Najdeno 17. avgusta na spletnem naslovu http://www.w3schools.com/browsers/browsers_stats.asp

60. Weinberg, G. M. (2008). Perfect software and other illusions about software testing. New York: Dorset House Publishing.

61. Zylberman, A.& Shotten, A. (b.l.). Test Language -Introduction to Keyword Driven Testing. *Methods & Tools*. Najdeno 8. avgusta na spletnem naslovu http://www.methodsandtools.com/archive/archive.php?id=108