UNIVERZA V LJUBLJANI
EKONOMSKA FAKULTETA

MAGISTRSKO DELO

OCENA PRIMERNOSTI ORODJA ZA AVTOMATIZACIJO TESTIRANJ GRAFIČNIH UPORABNIŠKIH VMESNIKOV V PODJETJU BANKART

Ljubljana, maj 2019 BLAŽ FIOLIĆ

IZJAVA O AVTORSTVU

Podpisani Blaž Fiolić, študent Ekonomske fakultete Univerze v Ljubljani, avtor predloženega dela z naslovom Ocena primernosti orodja za avtomatizacijo testiranj grafičnih uporabniških vmesnikov v podjetju Bankart, pripravljenega v sodelovanju s svetovalcem red. prof. dr. Tomažem Turkom,

I Z J A V L J A M ,

1. da sem predloženo delo pripravil samostojno;

2. da je tiskana oblika predloženega dela istovetna njegovi elektronski obliki;

3. da je besedilo predloženega dela 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 oziroma navajam v besedilu, citirana oziroma povzeta v skladu z Navodili za izdelavo zaključnih nalog Ekonomske fakultete Univerze v Ljubljani;

4. da se zavedam, da je plagiatorstvo – predstavljanje tujih del (v pisni ali grafični obliki) kot mojih lastnih – kaznivo po Kazenskem zakoniku Republike Slovenije;

5. da se zavedam posledic, ki bi jih na osnovi predloženega dela dokazano plagiatorstvo lahko predstavljalo za moj status na Ekonomski fakulteti Univerze v Ljubljani v skladu z relevantnim pravilnikom;

6. da sem pridobi vsa potrebna dovoljenja za uporabo podatkov in avtorskih del v predloženem delu in jih v njem jasno označil;

7. da sem pri pripravi predloženega dela ravnal v skladu z etičnimi načeli in, kjer je to potrebno, za raziskavo pridobil soglasje etične komisije;

8. da soglašam, da se elektronska oblika predloženega dela uporabi za preverjanje podobnosti vsebine z drugimi deli s programsko opremo za preverjanje podobnosti vsebine, ki je povezana s študijskim informacijskim sistemom članice;

9. da na Univerzo v Ljubljani neodplačno, neizključno, prostorsko in časovno neomejeno prenašam pravico shranitve predloženega dela v elektronski obliki, pravico reproduciranja ter pravico dajanja predloženega dela na voljo javnosti na svetovnem spletu preko Repozitorija Univerze v Ljubljani;

10. da hkrati z objavo predloženega dela dovoljujem objavo svojih osebnih podatkov, ki so navedeni v njem in v tej izjavi.

V Ljubljani, dne 29.5.2019 Podpis študenta: _____

KAZALO

KAZALO TABEL

KAZALO SLIK

KAZALO PRILOG

SEZNAM KRATIC

ang. – angleško

API – (ang. Application Programming Interface); aplikacijski programski vmesnik

DOM – dokumentno-objektni model

GUI – (ang. Graphical User Interface); grafični uporabniški vmesnik

UVOD

Iz akademskega in poslovnega sveta se poroča, da predstavljajo stroški testiranj enega glavnih izzivov na področju razvoja aplikacij in lahko predstavljajo vse do 60 % celotnih stroškov razvoja ter so redko nižji od 20 % (Ellims, Bridges & Ince, 2006; Hailpern & Santhanam, 2002; Ericson, Subotic & Ursing, 1997). Hitra in agilna okolja postajajo smernice industrije razvoja aplikacij, kjer je poudarek na kontinuirani integraciji, razvoju in prenosu do strank (Holmström Olsson, Alahyari & Bosch, 2012). Pri takem trendu se pojavi težnja po hitrejšem in pogostejšem testiranju ter poročanju o kakovosti aplikacij, kar je lahko dober argument za potrebo po vse večji avtomatizaciji testiranj.

Testiranje aplikacij je proces odkrivanja latentnih napak v delovanju aplikacij in je kritična faza v življenjskem ciklu razvoja aplikacij (Kumar & Mishra, 2016). Po odkritju napak v povprečju 7 % odpravljenih napak povzroči nove napake, zato je dobro ponovno testirati ostale funkcije aplikacij (Binder, 2011). Iz lastne prakse bi lahko rekel, da zgodnje zaznave napak in pospešitev različnih faz testiranja lahko privedejo do pospešitve celotnega procesa razvoja aplikacij. Kasneje, ko se napake odkrijejo, je potrebno več časa in s tem posledično stroškov za njihovo odpravo, težje je odkriti razlog za napako in večja je možnost, da bo treba popraviti kodo tudi na drugih mestih in ponovno testirati. Pozna odkrivanja napak vodijo v zamude projektov in če so napake odkrite po lansiranju produkta, se poleg stroškov popravil naredi tudi škoda dobrega imena podjetja. Torej ne glede na časovno potratnost in zahteve po virih je testiranje nekaj, česar ne moremo ignorirati. Vsak novo razvit ali spremenjen produkt mora čez temeljito testiranje, da se zagotovi kakovost produkta (Jalote, 2012, str. 465). Ker se podjetja, ki so v industriji razvoja aplikacij, srečujejo z internacionalno konkurenco in želijo zmanjšati stroške in časovne roke (Ramler & Wolfmaier, 2006), je lahko faza testiranj dobra priložnost za optimizacijo, da se doseže konkurenčna prednost.

Kot rezultat je danes na voljo veliko orodij, ki omogočajo avtomatizacijo testiranj, in podjetij, ki ponujajo storitve za izboljšavo testiranj. Vendar preden slepo vložimo v rešitev, moramo vedeti, da je avtomatizirano testiranje kritičen proces, in moramo najprej analizirati, katere funkcionalnosti v aplikacijah ga potrebujejo in so zanj primerne, preden podjetje nadaljuje z avtomatizacijo testiranj (Garousi & Mäntylä, 2016). Razlog je tudi v tem, da vsebuje avtomatizacija testiranj visoke začetne investicijske stroške v obliki analize potreb, analize orodja, grajenja skript, pridobitve orodja, izobraževanja itd. (Bertolino, 2007). Po uvedbi orodja pa želimo izvedeti, ali je bila investicija v avtomatizacijo testiranj upravičena, kar pa lahko storimo, če lahko identificiramo dejavnike, ki prinesejo koristi ali slabosti ter so merljivi (Bannister, 2004, str. 11). Da to dosežemo, je treba razumeti, kako poteka razvoj aplikacij in kakšno vlogo v njem igra testiranje ter kakšna so pričakovanja. Ko identificiramo dejavnike, jim moramo dodati količinsko vrednost, saj je po besedah Barnnisterja (2004, str. 11) vsaka ocena vrednosti več ali manj neuporabna, če je ne moremo izraziti s količinsko vrednostjo.

Kakovost aplikacij in časovni roki projektov so eni izmed glavnih problemov, ki se jih želi rešiti z avtomatizacijo testiranj grafičnih uporabniških vmesnikov (ang. Graphical User Interface, v nadaljevanju GUI). Problema sta velikega pomena še posebej pri razvoju aplikacij v primeru kontinuirane integracije (Fitzgerald & Stol, 2014), kjer se manjše spremembe pogosto implementirajo v aplikacijo, kar zahteva veliko ponavljajočih se testov. Optimizacija testnega procesa v primerih kontinuirane integracije je tako izrednega pomena, ker ima lahko velik vpliv na nove verzije aplikacij. Če želimo optimizirati testni proces in oceniti primernost le-tega, je treba analizirati, kateri so dejavniki, ki vplivajo na čas, stroške in kakovost produkta, ki nastajajo iz naslova avtomatizacije testiranj GUI.

Opisana problematika se je pojavila v podjetju Bankart. Aplikacije, ki jih podjetje razvija, se nenehno dopolnjujejo z novimi funkcionalnostmi ali pa spreminjajo obstoječe. V večini primerov te spremembe vplivajo na že obstoječo kodo in je tako treba že za manjše spremembe testirati celotno aplikacijo. Z rastjo aplikacij ta testiranja postajajo časovno zahtevna in zaradi ročnega dela je vse večja možnost, da se napake v kodi ne odkrijejo in prenesejo vse do produkcije. Rešitev je podjetje videlo v avtomatiziranem orodju za testiranje GUI, vendar po slabem letu dni ni bilo čisto jasnih dejavnikov, ki bi investicijo v orodje upravičili in dali odgovor na vprašanje, ali licenco za orodje podaljšati.

Namen in cilji

Namen magistrskega dela je preveriti, ali je bila uvedba orodja za avtomatizacijo testiranj grafičnih uporabniških vmesnikov v podjetju Bankart upravičena.

Ob tem se bo zasledovalo naslednje cilje:

- cilj 1: identifikacija dejavnikov, pomembnih za oceno upravičenosti uvedbe orodja za avtomatizacijo testiranj GUI;
- cilj 2: ovrednotenje identificiranih dejavnikov;
- cilj 3: analiza stanja pred uvedbo in po uvedbi orodja.

Za doseganje ciljev je treba:

1. narediti pregled literature na temo razvoja aplikacij, testiranja aplikacij in kako vrednotiti naložbe v avtomatizacijo testiranj GUI;
2. predstaviti uporabo orodja in sprememb, ki jih je prineslo;
3. izvesti intervju z uporabniki;
4. definirati dejavnike, ki vplivajo na stroške in koristi, vezane na avtomatizacijo testiranj GUI;
5. ovrednotiti definirane dejavnike;

6. izvesti primerjavo pred in po implementaciji orodja.

Magistrsko delo bo tako lahko dodalo znanje na podlagi prakse o stroških, povezanih z avtomatizacijo testiranj GUI, prepoznavo dejavnikov, ki stroške opravičujejo ali večajo, in kako take vrste avtomatizacija ustreza v današnjih hitrih agilnih razvojnih okoljih.

Struktura naloge

Magistrsko delo se vsebinsko razdeli na dva dela. Prvi del se začne s proučevanjem literature o razvoju aplikacij, kakšna je pri tem vloga testiranj, kakšne so poročane slabosti in koristi ročnega in avtomatiziranega testiranja ter kako te koristi meriti.

V drugem delu se predstavi študijo primera uvedbe avtomatiziranega orodja za testiranje GUI v podjetju Bankart, prikaže rezultate, definira dejavnike in opravi oceno upravičenosti uvedbe orodja na podlagi rezultatov in definiranih dejavnikov. Bolj natančno, najprej se predstavi potreba po avtomatizaciji testiranj GUI v danem primeru s predstavitvijo podjetja, orodja, zakaj je prišlo do odločitve o vpeljavi orodja in kako je potekal proces vpeljave in uporabe orodja. Sledijo predstavitev rezultatov pridobljenih iz meritev, pregled literature in intervjujev. Za tem so na vrsti analiza pridobljenih podatkov, definiranje in vrednotenje dejavnikov. Na koncu se naredi primerjava začetnega in končnega stanja ter ocena upravičenosti vpeljave orodja. Za zaključek se podajo ugotovitve in sklep.

Metodologija

Poudarek magistrskega dela je na dejavnikih, ki imajo vpliv na stroške in koristi pri avtomatizaciji testiranj GUI. Definiranje in merjenje dejavnikov sta razdeljena na dva dela. En del predstavlja merjenje dejavnika prihranjenega časa. Meri se porabljen čas za implementacijo in uporabo avtomatiziranega testiranja GUI ter primerja z ročnim testiranjem GUI. Empirični podatki se nabirajo z uporabo implementiranega orodja v obdobju enega leta. Metodologija meritev je natančneje predstavljena v točki 2.6. Drugi del predstavlja prepoznavo ostalih dejavnikov poleg časa in določanje njihove pomembnosti in vrednosti. To se je doseglo tako, da se je na podlagi pregleda literature, pričakovanj podjetja in opravljenih intervjujev definiralo dejavnike. Za odgovor na raziskovalno vprašanje se je primerjalo stanje pred uvedbo orodja in se ga primerjalo s stanjem po uvedbi orodja, kjer se upoštevajo definirani dejavniki. Sledi natančnejša predstava metodologije intervjujev, analize podatkov in sinteze podatkov.

Intervju

Uporabljen je bil delno strukturiran intervju. Študija je raziskovalne narave, tako je izbrani tip intervjuja primeren za zbiranje poglobljenih in kakovostnih informacij, s tem ko dovoljuje dodatna vmesna vprašanja v primerih, ko vprašani pove nekaj, kar nam je posebej zanimivo (Robson, 2002, str. 270). S sledenjem navodil, kot jih podajata Runeson in Höst (2009), je intervju razdeljen na štiri teme. Intervju se je začel z odprtim vprašanjem,

nadaljeval z bolj specifičnimi in formuliranimi vprašanji in nato ponovno zaključil z odprtim vprašanjem, kar oblikuje model peščene ure (Runeson & Höst, 2009). Teme so:

- **priprava in izkušnje**: vprašanim se predstavi namen, cilj in potek intervjuja. Vprašanja so se nanašala na predhodne izkušnje in trenutne vloge v avtomatiziranem testiranju GUI;

- **pregled avtomatiziranega testiranja GUI**: vprašanja so se nanašala na celoten proces in aktivnosti pri uporabi avtomatiziranega testiranja GUI in uporabi orodja;

- **koristi in slabosti avtomatizacije testiranja GUI:** v tem delu so vprašani razložili natančno, kako uporabljajo orodje in izvajajo avtomatizirano testiranje GUI, izpostavili dobro prakso, ki prinese dodatne koristi in težave, s katerimi so se srečevali. Poleg testiranja so razložili tudi, kako so vzdrževali skripte in kakšne so bile težave. Koristim in slabostim, ki so jih izpostavili, so dodelili tudi pomembnost;

- **predstavitev koristi in slabosti iz pregleda literature:** v zadnjem delu se združi literaturo s prakso. Ugotovitve iz pregleda literature, ki se navezujejo na iskane dejavnike, so bile predstavljene v tabelarnem formatu vprašanim, na katere so podali svoja mnenja (ali se strinjajo in stopnja pomembnosti).

Prve tri teme so mišljene kot prvi del intervjuja in zadnja tema kot drugi del. Cilj je bil dobiti mnenja uporabnikov glede različnih delov avtomatizacije testiranja GUI in kasneje združiti ugotovitve pregleda literature s subjektivnim mnenjem uporabnikov. S pridobljenimi informacijami so se lahko določile količinske vrednosti dejavnikom.

Analiza podatkov

Obstaja nekaj kvalitativnih pristopov, ki so raznovrstni in kompleksni in eden od njih je tematska analiza (Braun & Clarke, 2006). Je najbolj razširjena metoda analize za kvalitativne raziskave (Vaismoradi, Turunen & Bondas, 2013). Boyatzis (1998, str. 4) definira tematsko analizo kot proces, ki se ga lahko uporabi skoraj z vsako kvalitativno metodo in omogoča transformacijo kvalitativnih informacij v kvantitativne podatke. Tematska analiza je primerna za vprašanja, ki se navezujejo na poglede ljudi in njihove izkušnje (Vaismoradi, Turunen & Bondas, 2013). Tematska analiza poskuša zbrati mnenja in interpretirati nekaj pomembnega o podatkih v povezavi z raziskovalnim vprašanjem ter predstaviti vzorce in pomen znotraj sklopa podatkov (Braun & Clarke, 2006). Za analizo podatkov je uporabljena tematska analiza šestih korakov, ki sta jo predlagala Braun in Clarke (2006) za analizo kvalitativnih podatkov. V nadaljevanju je vsak korak tematske analize širše opisan.

Seznanjenje s podatki: prvi korak v procesu tematske analize je, da spoznamo podatke, ki so na voljo. Bird (2005) opiše ta korak kot ključen korak analize podatkov znotraj interpretativne kvalitativne metodologije. Ključnega pomena je, da se poglobimo v vsebino, preden pridemo do naslednjega koraka, ker oblikuje podlago, na kateri stoji preostanek

analize (Braun & Clarke, 2006). Vaismoradi, Turunen in Bondas (2013) pravijo, da ne glede na to, ali je cilj podrobna analiza, iskanje tem ali je teoretično voden, se je pomembno seznaniti z vsemi aspekti vsebine. Ko se vsi intervjuji opravijo, se temeljito prebere vse zapise in označi pomembne misli vprašanih. Ko se prvi korak zaključi, sledi korak ustvarjanja kod, kjer so označenim mislim dodeljene kode, ki dodajo pomen, kaj podatki predstavljajo.

Generiranje inicialnih kod: Boyatzis (1998, str. 31) definira kodo kot osnovni element ali segment surovih podatkov, ki prinese pomen fenomenu. Ko se spoznamo s podatki, se ustvari spisek kod. V tem koraku so pregledani vsi zapiski intervjujev in označenim podatkom so dodane kode. Pri pripravi kod se upošteva, da se ustvarijo kode za čim več možnih tem in da se označi podatke, ki razložijo kodo.

Iskanje tem: v tem koraku se osredotočenost analize preusmeri na širši pogled tem z razvrščanjem v potencialne teme. Bistvo koraka leži v tem, kako so lahko različne kode kombinirane, da se oblikuje tema (Braun & Clarke, 2006). Da se to doseže, se kode na ustvarjenem seznamu kod postavi v tabelo. Nato se vsako kodo analizira, da se identificira, ali so kode pomensko med sabo povezane oziroma si podobne. Tako se oblikujejo kategorije kod.

Pregled in izboljšava tem: ko imamo sklop kandidatnih tem, se ta korak začne z dopolnjevanjem tem, ki so na voljo. Braun in Clarke (2006) pravita, da obstajata dve stopnji tega koraka. V prvi stopnji se pregleda izvlečke kodnih podatkov, v drugi stopnji pa se validirajo teme z izvlečki podatkov.

Definiranje in poimenovanje tem: v tem koraku se pridobljene teme in pripadajoče kode definira in doda oznake s ponavljajočim se pregledovanjem kodnih podatkov (Bird, 2005). Podatki iz pregleda literature in intervjujev so podlaga za določanje pravilnih oznak kodam in temam. Premisli se, ali vsaka tema pripomore in ustreza k širšemu raziskovalnemu izidu.

Izdelava končnih tem: v zadnjem koraku so za vsako temo pridobljeni podporni podatki in opisane ugotovitve vsake teme.

Sinteza podatkov

Ko se zbere podatke s pregledom literature, sledi sinteza pridobljenih podatkov z uporabnikovimi mnenji (Dyba, Kitchenham & Jogensen, 2005). Pregled literature mora pridobiti podatke iz raziskav, ki koristijo uporabnikom in jim lahko služijo kot vodilo za omogočanje in podpiranje dobrih odločitev (Badampudi & Wohlin, 2016). Za doseganje tega se je uporabila Bayesianova metoda sinteze, ki jo predlagata Badampudi in Wohlin (2016).

Bayesianov pristop sintetizira podatke in ponudi sklepanja z vključevanjem znanja in izkušenj uporabnikov (Badampudi & Wohlin, 2016). Splošno povedano, Bayesianov pristop

je uporaben za sintezo podatkov in prenašanje znanja do uporabnikov z integriranjem njihovih subjektivnih mnenj v podatkovni sintezi. V nadaljevanju so širše opisani koraki Bayesianove sinteze.

Predhodna verjetnost: pomeni odstotek vprašanih, ki so omenili, da je koda veljavna in pomembna. V prvi fazi intervjuja vprašani povejo osebne izkušnje in mnenja o avtomatiziranem testiranju GUI brez podpornih informacij iz pregleda literature. S tematsko analizo se analizirajo njihovi odgovori ter generirajo kode in potencialne teme oz. kategorije za dejavnike, ki vplivajo na upravičenost implementacije orodja za avtomatizacijo testiranj GUI. Predhodna verjetnost, da je koda veljavna v kontekstu, se pridobi z združenjem pridobljenih informacij za vsako kodo.

Verjetnost: pomeni odstotek raziskav, ki poročajo določen faktor kot veljaven in pomemben. Verjetnost predstavlja, kar je že znano, npr. predstavitev raziskovalnih dokazov iz literature (Badampudi & Wohlin, 2016). Podatki so pridobljeni iz pregleda literature na temo koristi in slabosti avtomatiziranega testiranja GUI. Pridobljeni faktorji so razdeljeni v kategorije, ki so bile ustvarjene prek tematske analize v prejšnjih korakih in po potrebi dodane nove kategorije.

Posteriorna verjetnost: pomeni odstotek vprašanih, ki so trdili, da je faktor veljaven v kontekstu. Pregled literature se opravi pred izvedenimi intervjuji, da se pridobi mnenja uporabnikov za podatke iz literature. V drugi fazi intervjuja so vprašanim predstavljeni faktorji, pridobljeni iz raziskovalnih del v tabelarni obliki. Vprašani podajo svoja mnenja glede veljavnosti in pomembnosti vsakega faktorja v njihovem kontekstu. Te informacije se uporabi za izračun posteriorne verjetnosti.

1 PREGLED LITERATURE

1.1 Razvoj aplikacij

1.1.1 Faze razvoja aplikacij

Procesi, ki se izvajajo med razvojem in evolucijo programske opreme, postajajo vse bolj kompleksni in zahtevajo veliko resursov. Vključujejo ljudi, ki izvajajo aktivnosti, katerih primarni namen je ustvarjanje kakovostnih aplikacij v skladu s postavljenimi zahtevami. Tako morajo biti razvijalci sposobni in imeti veliko znanja, kar pa za podjetje pomeni drage ure dela. Zato ima projektni vodja pomembno vlogo pri planiranju razvoja in ocenjevanju, koliko človeških dni dela bo potrebno za razvoj, ker vsaka nekonsistentnost lahko generira povečane stroške. Pri tem pa mu lahko pomaga upoštevanje formalno oblikovanih faz in metodologij razvoja programske opreme (Thulasee Krishna, Sreekanth, Perumal & Rajesh Kumar Reddy, 2012).

Razvoj programske opreme se deli na sedem faz, kjer dobijo določene faze večjo pozornost glede na vrsto projekta (Despa, 2004). V nadaljevanju so povzete omenjene faze:

- **raziskave**: faza, v kateri lastnik projekta, vodja projekta in projektna skupina zbirajo in si izmenjujejo informacije. V fazi raziskave bo lastnik projekta poskušal najti ljudi ali podjetja s podobnimi cilji in dokumentirati način, kako so oni delovali pri izpolnjevanju svojih ciljev. Vodja projekta je odgovoren za sprejemanje zahtev od lastnika projekta, njihovo vrednotenje in posredovanje zahtev projektni skupini kot tehnične specifikacije. Projektna skupina je odgovorna za ovrednotenje zahtev s tehničnega vidika, raziskavo konceptualnih okvirjev, aplikacijskih programskih vmesnikov (v nadaljevanju API), knjižnic in infrastrukturo, ki bodo potrebni za izdelavo aplikacije;

- **planiranje**: faza, v kateri so vsi elementi nastavljeni za razvoj aplikacije. Začne se z določanjem generalnega toka aplikacije. Na podlagi zahtevane funkcionalnosti je zasnovana struktura baze podatkov. Vodja projekta se mora odločiti o najprimernejši metodologiji upravljanja in o ustreznem delovnem protokolu za projekt in skupaj s projektno skupino izbrati tehnologijo, ki bo uporabljena za razvoj aplikacije;

- **dizajn**: faza, v kateri se ustvari postavitev aplikacije. Odvisno od narave aplikacije lahko dizajni segajo od enostavnih in funkcionalno usmerjenih do kompleksnih in artistično usmerjenih. Faza se lahko prekriva s fazo planiranja in razvoja. Na tej stopnji ponavadi lastnik projekta prihaja do novih zahtev, ki jih je treba vključiti v raziskave in planiranja;

- **razvoj**: faza, v kateri se napiše koda in se aplikacija dejansko zgradi. Razvojna faza se začne z vzpostavitvijo razvojnega in testnega okolja. Pomemben vidik faze razvoja je spremljanje napredka. Vodja projekta mora določiti dejanski napredek in ga oceniti glede na začetno načrtovanje in nenehno poročati lastniku projekta o splošnem napredku;

- **testiranje**: faza, v kateri se odkrivajo in odpravljajo napake v programiranju in dizajnu. Programske napake so scenariji, kjer se aplikacija zruši ali obnaša drugače, kot načrtovano. Napake v dizajnu so neskladnosti med zahtevo lastnika projekta in kar je projektna skupina implementirala;

- **namestitev**: faza, v kateri je aplikacija nameščena v produkcijskem okolju. Namestitvena faza nastopi pred začetkom uporabe aplikacije. Ko je aplikacija nameščena, gre ponovno skozi celoten cikel testiranj. Ko je testiranje končano, se aplikaciji doda vsebina;

- **vzdrževanje**: faza, ki zajema razvoj aplikacije po namestitvi aplikacije in je tudi odgovorna za zagotavljanje, da aplikacija deluje v okviru načrtovanih parametrov. Zagotavljanje pravilnega delovanja aplikacije se doseže s spremljavo dnevnikov napak. Pomemben del faze vzdrževanja je sistematično testiranje funkcionalnosti za napake, ki v testni fazi niso bile ugotovljene, ali za težave, ki niso prikazane v

dnevnikih napak. Faza vzdrževanja predstavlja tudi priložnost dodajanja novih funkcionalnosti aplikaciji.

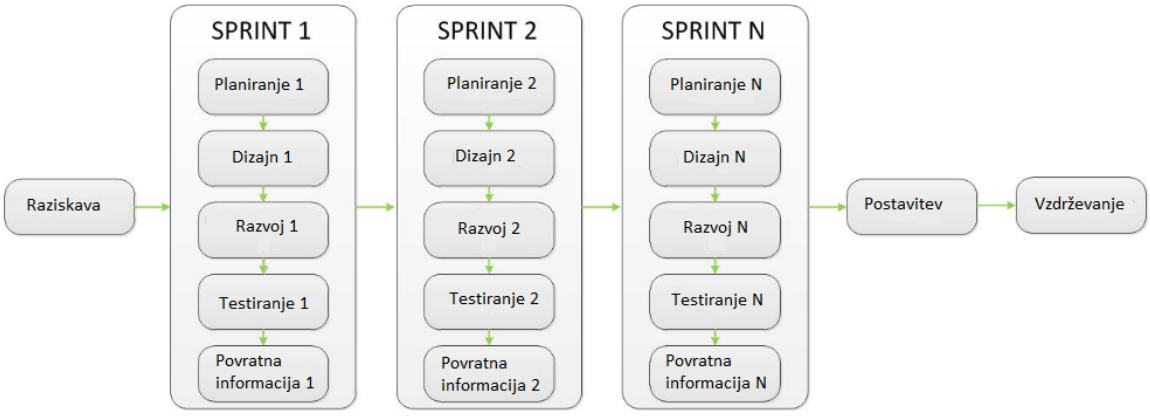
S predstavljenimi fazami se na splošno strinja skupnost za razvoj aplikacij in veljajo kot steber vsakega projekta razvoja aplikacij. Odvisno od metodologije razvoja aplikacij pa so lahko drugače poimenovane, se prekrivajo, imajo spremenjen vrstni red, lahko pa tudi manjkajo.

1.1.2 Metodologije razvoja aplikacij

Obstajajo mnoge metodologije razvoja in izbira prave lahko pomeni veliko razliko pri doseganju uspešnih rezultatov, ko se jih meri v obliki stroškov, doseganju končnih rokov, zadovoljstvu strank in robustnosti aplikacije (Young, 2013). V nadaljevanju so naštete različne metodologije razvoja, ki jih omenjajo številni avtorji (Young, 2013; Despa, 2004; Thulasee Krishna, Sreekanth, Perumal & Rajesh Kumar Reddy, 2012):

- slapovni model (ang. Waterfall),
- model V,
- Scrum,
- model hitrega razvoja aplikacij (ang. Rapid application development),
- ekstremno programiranje (ang. Extreme programming),
- metoda razvoja dinamičnih sistemov (ang. Dynamic systems development method),
- spiralna metoda (ang. Spiral),
- vitka metoda (ang. Lean),
- kristalne metode (ang. Crystal methods),
- inkrementalna metoda (ang. Incremental).

Slika 1: Model sprinta



Vir: Prirejeno po (Despa, 2004).

Natančneje je v nadaljevanju predstavljena metodologija Scrum, ker je uporabljena v našem primeru. Metodologija Scrum je namenjena skupinam od tri do devet članov, ki svoje delo razdelijo na več manjših nalog, ki jih je mogoče dokončati v časovnih intervalih, imenovanih

sprint (30 dni ali manj, najpogosteje dva tedna), kot prikazano v sliki 1. Scrum sestavlja ponavljajoč se cikel več faz, in sicer groominga, ki nastopi pred začetkom novega sprinta, kjer se definirajo aktualne naloge za določen projekt in se jim določi prioriteta. Sledi mu planiranje, kjer se posameznikom določi naloge, ki jih bodo naredili v naslednjem sprintu. Napredek nalog se sproti spremlja s 15-minutnimi dnevnimi sestanki, imenovanimi daily. Za zaključek sprinta se opravi še retrospektiva, kjer se preveri in predstavi, kaj je bilo narejeno v zadnjem sprintu.

1.2 Testiranje aplikacij

1.2.1 Definicija testiranja

Odkar so bile zgrajene prve aplikacije, obstaja testiranje aplikacij, saj je bilo treba preveriti, ali je aplikacija funkcionalna, preden se jo sprosti v produkcijo. Testiranje aplikacij zagotavlja zanesljivost, kar povečuje zaupanje kupcev (Kassab, 2016).

Aplikacije morajo biti testirane, če želijo ustvariti zaupanje, da bodo pravilno delovale v namenjenem okolju. Testiranje aplikacij mora biti učinkovito pri odkrivanju napak in hkrati biti hitro ter povzročati nizke stroške (Fewster & Graham, 1999, str. 3).

Testiranje odkrije primere, kjer aplikacija ne doseže dogovorjenih specifikacij. Na podlagi te definicije je katera koli aktivnost, ki odkrije kakršno koli neskladje s specifikacijami, lahko označena kot testiranje. V tem kontekstu spadajo pod testiranje aktivnosti npr. pregled dizajna, pregled kode in analiza izvirne kode, čeprav se ne izvršuje koda pri iskanju napak. Seveda je pa izvrševanje kode z izvedbo specifičnih testnih primerov, ki ciljajo na določeno funkcionalnost aplikacije, velik del testiranja (Hailpern & Santhanam, 2002).

Testiranje je kritični element pri zagotavljanju kakovosti aplikacij in predstavlja glavni pregled specifikacij, dizajna in kode (Sharma, 2014). Sharma (2014) izpostavi, da testiranje aplikacij ne zagotovi samo pravilnega delovanja produkta, ampak tudi, pod katerimi pogoji aplikacija ne deluje. Različni tipi aplikacij imajo različne zahteve in različne končne uporabnike, tako je treba razviti aplikacijo, ki bo ustrezala njenim končnim uporabnikom, njeni ciljni publiki, njenim kupcem in njenim interesnim skupinam. Testiranje aplikacij je proces, s katerim poskušamo zagotoviti to ustreznost (Sharma, 2014). Način testiranja se razdeli na dve vrsti, in sicer ročno testiranje in avtomatizirano testiranje (Sharma, 2014; Patidar, Sharma & Dave, 2017; Rathi & Mehra, 2015; Hewlett-Packard Development Company, L.P., 2010).

1.2.2 Aktivnosti testiranja

Opravljanje aktivnosti testiranja različnih organizacij se lahko razlikuje (Fewster & Graham, 1999, str. 13). Fewster in Graham (1999, str. 13) opišeta osnovne aktivnosti, ki se v večini primerov izvedejo in si sledijo po naslednjem zaporedju:

1. **identifikacija testnih pogojev:** ugotavljanje, kaj je lahko testirano, in postavljanje prioritet pogojem;

2. **dizajniranje testnih primerov:** določanje, kaj bo testirano, kako bo testirano, kakšni so pričakovani rezultati in morebitne ostale informacije, potrebne za test;

3. **grajenje testnih primerov:** priprava testnih skript, testnih vhodov, testnih podatkov in pričakovanih rezultatov;

4. **izvedba testnih primerov:** aplikacijo pod testom se uporabi po navodilih testnih primerov;

5. **primerjava dobljenih rezultatov s pričakovanimi rezultati:** dejanski izid vsakega testa je treba pregledati, če je aplikacija pod testom delovala pravilno.

1.2.3 Ročno testiranje

Ročno testiranje je tehnika testiranja, pri kateri tester ročno pripravi in izvede testni primer, da identificira napake v aplikaciji. Je najbolj rigorozna in stara metoda testiranja aplikacij (Maurya & Kumar, 2012).

S časom je tehnologija napredovala in aplikacije so postale bolj zapletene, proces ročnega testiranja pa je ostal večinoma nespremenjen. Še vedno porabi veliko resursov in predstavlja ponavljajoče se monotono delo z ročnim vpisovanjem velike količine podatkov, kar lahko povzroča večje možnosti napak, ki vodijo v spregledane napake in poročanje lažnih napak (Hewlett-Packard Development Company, L.P., 2010). Tako Sharma (2014) pravi, da mora biti tester potrpežljiv, pozoren, kreativen, inovativen, odprtih misli in spreten.

1.2.4 Prednosti in slabosti ročnih testiranj

Prednosti ročnega testiranja so, da ni potrebnih kompleksnih priprav, grajenja skript ali kodiranja. Lahko ga opravi uporabnik, ki nima veliko tehničnega znanja ali posebnih orodij. Ročno testiranje je lahko opravljeno v kateri koli fazi v življenjskem ciklu aplikacije (Hewlett-Packard Development Company, L.P., 2010).

Slabosti, s katerimi se strinjajo različni avtorji (Sharma, 2014; Patidar, Sharma & Dave 2017; Hewlett-Packard Development Company, L.P., 2010; Mohan Doss Gandhi & Pillai, 2014):

- velika poraba časa,
- velika investicija v ljudi,
- zaradi človeškega faktorja ni velike zanesljivosti,
- zaradi časovnih ovir ni mogoče temeljito testirati vseh funkcij aplikacije, preden se prenese aplikacija v produkcijo.

Naštete slabosti tudi sam vidim v praksi, ko pridobijo testerji vse več pritiska zaradi krajšanja časovnih rokov, kar posledično povzroči manjšo natančnost in pozornost pri testiranju. Težko je opraviti teste v različnih okoljih, brskalnikih in operacijskih sistemih. Zato se osredotoča testiranje na najbolj pogoste primere in ignorira robne. Problem se tudi pojavi, ko se napake zaznajo in jih tester težko ponovi in komunicira razvijalcu, ker ne ve točno, kaj se je zgodilo oziroma kakšno je bilo točno zaporedje, ki je sprožilo napako.

1.2.5 Definicija avtomatizacije testiranj

Mohan Doss Gandhi in Pillai (2014) definirata avtomatizacijo kot uporabo strategij, orodij in artefaktov, ki zmanjšujejo potrebo po ročnem delu ali interakciji v ponavljajočih se ali odvečnih nalogah. Sharma (2014) pravi, da avtomatizacija testiranja aplikacij vključuje razvoj testnih skript, ki uporabljajo skriptne jezike, kot so Python, JavaScriptali ali TCL, tako da lahko teste izvedejo računalniki z minimalno človeško pomočjo. Vendar meni, da je cilj avtomatizacije zmanjšati število testov, ki jih je treba opraviti ročno, in ne v celoti izključiti ročnega testiranja.

Za boljše razumevanje, kako avtomatizacija testiranja vpliva na proces testiranja, so v nadaljevanju predstavljene aktivnosti avtomatiziranega testiranja (Garousi & Mäntylä, 2016):

- **dizajn testnih scenarijev**: priprava seznama testnih scenarijev ali testnih zahtev, da se zadosti kriterij pokritosti in ostalih ciljev;
- **priprava testnih skript**: dokumentiranje testnih primerov;
- **izvedba testa**: poganjanje testnih primerov na aplikaciji, ki je pod testom, in zapisovanje rezultatov;
- **ocena testov**: ocenjevanje rezultatov testiranj (uspešnih ali neuspešnih), imenovano tudi testne razsodbe;
- **poročanje rezultatov testiranj**: poročanje testnih razsodb in napak razvijalcem;
- **upravljanje testov in ostalih testnih aktivnosti**: vključuje aktivnosti, kot so planiranje, kontroliranje, spremljanje in ocene napora. Ostale testne aktivnosti vključujejo minimiziranje testnih sklopov in izbiro regresijskih testov.

Fewster in Graham (1999) pravita, da sta prvi dve aktivnosti testiranja, identifikacija in dizajn testnih primerov, intelektualne narave, medtem ko sta izvajanje testnih primerov in primerjava rezultatov bolj administrativne narave. Slednji aktivnosti sta bolj delovno intenzivni in se ju večkrat ponovi, zato sta bolj primerni za avtomatizacijo.

1.2.6 Definicija testiranj grafičnih uporabniških vmesnikov

Avtomatizirano testiranje aplikacij je preferirano za aplikacije, ki temeljijo na GUI, saj se opravljanje testiranj zgodi večkrat (Maruthi Prasad & Krishna Kishore, 2015). GUI je programski vmesnik, ki izkoristi grafične sposobnosti računalnika, da olajša uporabo aplikacij (Isabella & Retna, 2012). Najpomembnejši del aplikacij, ki se uporablja v današnjem času, je GUI (Xie & Memon, 2006). Testiranje GUI Isabella in Retna (2012) definirata kot proces testiranja aplikacijskega GUI, če dosega zapisane specifikacije in če aplikacija funkcionalno deluje pravilno. Maruthi Prasad in Krishna Kishore (2015) pravita, da testiranje GUI vključuje, kako aplikacija obravnava dogodke v aplikaciji, povzročene s tipkovnico ali miško, kako različne komponente GUI, kot so menijske vrstice, orodne vrstice, pogovorna okna, gumbi, urejanje polj itd., reagirajo na uporabnikove akcije in ali jih aplikacija izvaja, kot je pričakovano. Testiranje GUI je proces testiranja aplikacijskih vizualnih elementov za zagotavljanje, da so slike in ostali objekti vidni uporabniku in delujejo kot pričakovano (Aebersold, 2018).

Pri aplikacijah GUI je treba opraviti več testov, ker je možno ogromno različnih načinov interakcij z GUI, kar vodi do velikega števila stanj GUI (povezana težava je določiti obseg testnih primerov), veliko število možnih stanj GUI pa povzroči veliko število permutacij vhodov, ki jih je treba upoštevati. Preverjanje stanja GUI ni enostavno, saj je težko določiti, katere objekte in lastnosti je treba preveriti (Martinez Perez, Luis Mateo Navarro & Sevilla Ruiz, 2016).

1.2.7 Prednosti in slabosti avtomatizacije testiranj

V nadaljevanju so predstavljene prednosti in slabosti avtomatizacije testiranj, ki so se identificirale v različnih literarnih delih.

Prednosti avtomatizacije testiranj:

- **izboljšana kakovost produkta:** kakovost, mišljena kot manj poročanih napak v aplikaciji, ki je bila izdana v produkcijo (Fewster & Graham, 1999; Karhu, Repo, Taipale & Smolander, 2009; Malekzadeh & Ainon, 2010; Tan & Edwards, 2008);

- **pokritost testa:** višja pokritost kode je dosežena skozi avtomatizacijo (Alshraideh, 2008; Burnim & Sen, 2008; Coelho in drugi, 2007; Fewster & Graham, 1999; Karhu, Repo, Taipale & Smolander, 2009; Leitner, Ciupa, Meyer & Howard, 2007; Malekzadeh & Ainon, 2010; Rathi & Mehra, 2015; Saglietti & Pinte, 2010; Sharma, 2014; Tan & Edwards, 2008);

- **krajši čas testiranja:** čas, porabljen za testiranje, npr. sposobnost izvedbe več testov v določenem času (Berner, Weber & Keller, 2005; du Bousquet & Zuanon, 1999; Fewster & Graham, 1999; Karhu, Repo, Taipale & Smolander, 2009; Rathi & Mehra, 2015; Tan & Edwards, 2008; Wissink & Amaro, 2006);

- **zanesljivost:** avtomatizirano testiranje je bolj zanesljivo, ker ko se testi ponavljajo, tester z ročnim testiranjem ne ponovi testov vsakič identično, tako prihaja do drugačnih izidov (Fewster & Graham, 1999; Patidar, Sharma & Dave, 2017; Rathi & Mehra, 2015; Sharma, 2014);

- **višje zaupanje:** povečano zaupanje v kakovost sistema, npr. kot ga dojemajo razvijalci (Coelho, in drugi, 2007; Fewster & Graham, 1999; Haugset & Hanssen, 2008);

- **izboljšana motivacija:** ker testerji ne izvajajo več monotonih, ponavljajočih se testov, se jim izboljša delovna motivacija (Fewster & Graham, 1999; (Hewlett-Packard Development Company, L.P., 2010);

- **ponovna uporabnost testov:** ko so testi ustvarjeni z vzdrževanjem v mislih, so lahko pogosto ponovno izvedeni tudi na različnih verzijah aplikacij. Visoka stopnja ponavljanja testnih primerov vodi do prednosti, ki jih ena sama izvedba testa ne (Alshraideh, 2008; Coelho, in drugi, 2007; Dallal, 2009; Fecko & Lott, 2002; Fewster & Graham, 1999; Karhu, Repo, Taipale & Smolander, 2009; Leitner, Ciupa, Meyer & Howard, 2007; Malekzadeh & Ainon, 2010; Rathi & Mehra, 2015; Sharma, 2014; Tan & Edwards, 2008);

- **manj človeškega truda:** avtomatizacija testiranja zmanjša človeški trud za testiranje, ki se ga lahko porabi za druge aktivnosti (Berner, Weber & Keller, 2005; Dallal, 2009; du Bousquet & Zuanon, 1999; Fewster & Graham, 1999; Leitner, Ciupa, Meyer & Howard, 2007; Rathi & Mehra, 2015; Wissink & Amaro, 2006);

- **zmanjšanje stroškov:** z visoko stopnjo avtomatizacije se prihranijo stroški (Alshraideh, 2008; Bashir & Banuri, 2008; Berner, Weber & Keller, 2005; Dallal, 2009; Fewster & Graham, 1999; Patidar, Sharma & Dave, 2017; Rathi & Mehra, 2015; Shan & Zhu, 2009; Sharma , 2014);

- **testi, ki jih ročno testiranje ne zmore:** nekateri testi so ročno neizvedljivi, npr. testi zmogljivosti in stres testi (Berner, Weber & Keller, 2005; Fewster & Graham, 1999; Ramler & Wolfmaier, 2006).

Poleg naštetih prednosti Melton (2015) predstavi še skrite prednosti, ki jih lahko prinese avtomatizacija testiranj:

- **zaznava subtilnih napak:** eno ključnih koristi avtomatiziranega testiranja je, da je malo inkrementalnih stroškov v ponavljanju testov. Veliko napak se zgodi z nizko frekvenco, kar pomeni, da je visoka verjetnost, da se je z enim testom ne bo odkrilo. Z avtomatiziranimi testi se lahko testi ponovijo večkrat, da se zazna tudi te napake;

- **prikaz, kje se lahko test izboljša:** dobri avtomatizirani testi beležijo čas vsega. Uporabno je vedeti, koliko časa traja posamezen test, vendar je še posebej dobro vedeti, koliko časa porabijo posamezni koraki. Tako se lahko hitreje optimizira testni proces ali pa tudi vidi, kateri aspekt produkta pod testom se lahko izboljša;

- **paralelno delovanje skript:** medtem ko lahko ročno izvajamo le en test naenkrat, lahko računalnik opravi več testov hkrati. Z dobro arhitekturo avtomatiziranih testov se lahko požene več testov hkrati;

- **poročila:** poleg avtomatiziranega testa se lahko generirajo avtomatizirana poročila o uspešno ali neuspešno opravljenih testih;

- **regresijski testi brez predhodnega dokumentiranja:** lahko se zažene različne regresijske teste brez predhodnega dokumentiranja korakov testa. Avtomatsko izdelana poročila po testu lahko služijo kot dokument, kaj je bilo testirano.

Slabosti avtomatizacije testiranj:

- **avtomatizacija ne more nadomestiti ročnega testiranja:** vseh testiranj se ne more z lahkoto avtomatizirati. Za nekatere testne primere tudi ni smiselno uporabiti avtomatiziranih testov, kot so nekateri primeri testov GUI (Berner, Weber & Keller, 2005; Karhu, Repo, Taipale & Smolander, 2009; Pettichord, 1999; Ramler & Wolfmaier, 2006);

- **visoki začetni stroški:** stroški nakupa licenc oziroma orodja, priprava okolja za uporabo orodja, izobraževanja uporabnikov in ustvarjanje prvih skript (Bertolino, 2007; Fewster & Graham, 1999; Jalote, 2012; Karhu, Repo, Taipale & Smolander, 2009; Liu, 2000; Rathi & Mehra, 2015);

- **težko vzdrževanje testnih skript:** spremembe v tehnologiji in evolucija aplikacij vodijo do težkega vzdrževanja testnih skript (Berner, Weber & Keller, 2005; Fewster & Graham, 1999; Karhu, Repo, Taipale & Smolander, 2009; Liu, 2000; Pettichord, 1999; Ramler & Wolfmaier, 2006);

- **proces avtomatizacije testiranja potrebuje čas za zrelost:** ustvarjanje infrastrukture in testov za avtomatizacijo potrebuje čas, torej da avtomatizacija in pripadajoče prednosti pridejo do izraza, je potreben čas (Bashir & Banuri, 2008);

- **napačna pričakovanja:** organizacije imajo nepraktična pričakovanja glede avtomatizacije testiranja aplikacij, kjer je pogosto cilj prihraniti čim več stroškov, npr. porabljanje truda na neproduktivnih testnih aktivnostih (Berner, Weber & Keller, 2005; Fewster & Graham, 1999; Pettichord, 1999);

- **potreba po ljudeh z znanji:** za oblikovanje testnih primerov in vzdrževanje testnih skript je potrebno znanje, kar pomeni, da so tehnične zahteve za testerje višje kot pri ročnem testiranju (Fecko & Lott, 2002; Pettichord, 1999).

1.2.8 Tehnike testiranj

Tradicionalno se lahko tehnike testiranja aplikacij razdelijo na dva dela, in sicer testiranje črne škatle (ang. Black box testing) in testiranje bele škatle (ang. White box testing). Testiranje črne škatle se imenuje tudi funkcionalno testiranje, ki je funkcionalna tehnika testiranja, kjer se izdelajo testni primeri na podlagi informacij specifikacij. S testiranjem črne škatle tester naj ne bi imel oziroma nima dostopa do same izvorne kode. Pri testiranju črne škatle se ne obremenjujemo z internimi mehanizmi sistema, ampak je fokus izrecno na generiranih izhodih v odziv na izbrane vhode in pogoje izvrševanja. Testerju so znane le informacije o vhodih, ki grejo v črno škatlo in vejo, kaj pričakovati, da se pošlje nazaj (Liu & Kuan Tan, 2009).

Testiranje bele škatle se imenuje tudi strukturno testiranje (ang. Structural testing) ali testiranje steklene škatle (ang. Glass box testing), ki je strukturna tehnika testiranja, kjer se izdelajo testni primeri na podlagi informacij iz izvorne kode. Tester bele škatle, ki je ponavadi razvijalec kode, ve, kako koda izgleda, in napiše testne primere z izvrševanjem metod z določenimi parametri. Pri testiranju bele škatle nas zanimajo interni mehanizmi sistema in je fokus na toku podatkov programa (Liu & Kuan Tan, 2009). Testiranje bele škatle je večinoma uporabljeno za odkrivanje napak v logiki programske kode, za odpravo napak v kodi, imenovano razhroščevanje (ang. Debugging), za iskanje naključnih tipografskih napak in odkrivanje napačnih programskih domnev (Nidhra & Dondeti, 2012).

Testiranje aplikacije je vpleteno v vsako stopnjo življenjskega cikla aplikacije, vendar način, kako je testiranje opravljeno na vsaki stopnji razvoja aplikacije, je različen in ima drugačne cilje (Nidhra & Dondeti, 2012).

Testiranje enot je testiranje, ki bazira na kodi in ga izvede razvijalec. Namen je testirati vsako individualno enoto posebej (Jorgensen, 2002). Enota je najmanjši element, ki se lahko testira v aplikacijah (Burnstein, 2003).

Integracijsko testiranje validira, da dve ali več enot ali druge integracije delujejo skupaj pravilno, in se nagiba k fokusu na povezave, ki so specificirane na stopnji dizajna komponent (Jorgensen, 2002).

Sistemsko testiranje prikaže, da celoten sistem deluje v okolju, ki je podobno produkcijskemu, da se vidijo poslovne funkcije, ki so bile specificirane v dizajnu (Jorgensen, 2002). Testira se funkcionalne in nefunkcionalne zahteve sistema, kot so stres testi, robustnostni testi, varnostni testi itd. (Pawar, 2015).

Potrditveni test opravi poslovni lastnik z namenom, da stranka sama testira, ali produkt dosega njihove poslovne pogoje in pričakovanja (Jorgensen, 2002).

Regresijski test je testiranje aplikacije po narejenih spremembah. Namen testiranja je zagotoviti zanesljivost vsake nove izdaje aplikacije v produkcijo in da spremembe niso prinesle novih napak pri delovanju sistema (Jorgensen, 2002).

Funkcijski test je opravljen za dokončano aplikacijo (Jorgensen, 2002). Obstajajo številni kriteriji definiranja funkcijskih testov. Če začnemo z vhodno domeno programa, so lahko testni primeri izpeljani naključno z generiranjem direktno iz specifikacij (Duran & Ntafos, 1984) ali pa sistematično s formaliziranjem procesa izpeljave testnih primerov v elementarne korake (Pezze & Young, 2007). Pezze in Young (2007) predstavita še operacijsko testiranje, ki temelji na ideji, da bodo nekatere funkcionalnosti pogosteje uporabljene kot ostale, tako si zaslužijo več truda pri testiranju. Operacijski profil je zgrajen z dodeljevanjem verjetnosti pojavljanja k funkcionalnostim. Natančnost operacijskega profila je odvisna od natančnosti informacij o tem, kako bo sistem uporabljen.

Dimni test (ang. Smoke test) je osnovni test, ki preveri osnovne funkcionalnosti aplikacije. Namen je, da je izvedba testa hitra in se preveri pravilno delovanje ključnih funkcij. Dimni testi so lahko uporabni takoj po dodajanju ali spremembi komponente, da se preveri, ali se lahko nadaljuje z bolj specifičnimi testi ali pa po prenosu aplikacije iz enega okolja v drugega (Pittet, 2018).

1.3 Vrednotenje investicije v informacijsko tehnologijo

Dobro vodene investicije v IT, ki so pazljivo izbrane in se osredotočajo na doseganje poslovnih ali ciljnih potreb, imajo lahko pozitiven vpliv na delovanje organizacije. Slabe investicije, tiste, ki so neustrezno upravičene ali pri katerih je slabo upravljanje s stroški, tveganji in koristi, lahko ovirajo in tudi preprečijo delovanje organizacije (Gunasekaran, Love, Rahimi & Miele, 2001).

Tradicionalni finančni modeli so lahko za vrednotenje investicij v IT problematični pri količinski opredelitvi stroškov, ker je težje oceniti nematerialne koristi in preveriti možne finančne rezultate iz teh koristi (Lin, Graham & McDermid, 2005). Tako vrednotenje IT-naložbe pogosto ne uspe. Po drugi strani pa je vrednotenje naložb v IT mogoče storiti z upoštevanjem stroškov izvajanja in analizo splošnih koristi sistemov (Goodhue, Wybo & Kirsch, 1992). Poleg tega je bila poslovna vrednost IT prikazana kot sposobnost IT, ki povečuje tržno in finančno uspešnost ter s tem splošno konkurenčnost organizacije (Nakata, Zhu & Kraimer, 2008). Zato je pomembno poudariti, da ocena naložb v IT ni povezana le s finančnimi rezultati, ampak tudi z večrazsežnostnimi vidiki (Soh & Markus, 1995), ki odločevalcem omogočajo, da pri odločanju upoštevajo več vidikov. Nazadnje, in kar je najpomembneje, popolnejša slika poslovne vrednosti IT se lahko zagotovi, če objektivni ukrepi izražajo tako neotipljive (ne neposredno merljive) kot otipljive (neposredno merljive) koristi (Tallon, Kraemer & Gurbaxani, 2000).

Vrednotenje naložb v IT je problematična tema. Čeprav so številne organizacije trdile, da uporabljajo finančne meritve kot metodo vrednotenja, so različne literature, ki so poudarile njene omejitve, trdile, da finančne meritve niso primerne za merjenje naložb v IT (Lefley, 2013).

Različni avtorji so poskušali določiti metodo ocenjevanja za naložbe v IT. Na podlagi raziskave (Powell, 1992) se metode ocenjevanja naložb v IT lahko razvrstijo v dve glavni kategoriji: objektivne metode in subjektivne metode. Prva se nanaša na tradicionalne metode, katerih namen je količinsko opredeliti sistemske vhode in izhode, da bi pritrdili vrednosti na predmete. Po drugi strani pa subjektivne metode vrednotijo vrednost naložb v IT iz odnosov in mnenj uporabnikov in sistemskih graditeljev. Na podoben način sta Bannister in Remenyi (2000) razvrstila metodo ocenjevanja naložb v IT, ki temelji na tem, kako vrednost prinaša naložbeno odločitev. Metode so razvrščene v tri skupine: temeljna, kompozitna in meta metoda. Temeljna metoda je niz meritev, ki poskušajo razvrstiti niz značilnosti naložbe v eno meritev; na primer tehnike določanja kapitalskega proračuna, kot sta donosnost naložb in notranja stopnja donosa. Kompozitna metoda je kombinacija temeljnih meritev za pridobitev splošne slike o vrednosti naložbe; na primer uravnotežen kazalnik (Kaplan & Norton, 1993), pristop portfelja naložb (Ward, 1994) in enostavne metode ocenjevanja večvrstnih lastnosti (Goodwin & Wright, 1998). Meta metoda poskuša izbrati optimalni niz ukrepov za kontekst ali niz okoliščin; na primer primerjavo med različnimi naložbami v IT skozi čas.

Nasprotno pa raziskave Gomeza in Pathera (2012) kažejo, da so bile prejšnje literature o metodah ocenjevanja naložb v IT preveč osredotočene na skrb za učinkovito rabo tehnologij in pojasnile učinkovitost na različne načine, od sorazmerno preprostih računovodskih ukrepov do zapletenih večdimenzionalnih uravnoteženih kazalnikov. Trdita, da je vrednotenje naložb v IT odvisno od preoblikovanja tehnologij, v katerih se danes osredotoča na neotipljive vidike poslovnih koristi, vključno z zvestobo in izboljšanjem blagovne znamke.

Bannister (2004, str. 81–83) izpostavi tudi nevidne stroške, ki se pojavljajo v IT in jih je težko najti ter jim postaviti vrednost. Vendar to ne pomeni, da so tovrstni stroški kaj manj resnični. Avtor našteje sledeče povzročitelje nevidnih stroškov:

- **krivulja učenja** predstavlja porabljen čas za seznanjanje s sistemom. Med tem časom lahko pride do padca produktivnosti vse do 50 % normalne produktivnosti. Pojavljajo se tudi napake pri delu, kar lahko še poveča stroške. Najboljši način zmanjševanja stroškov krivulje učenja je dobro izobraževanje in podpora;

- **neučinkovita raba sistema** lahko povzroči različne nevidne stroške. Lahko so manjši elementi, ki jih posamezniki ne znajo uporabljati učinkovito, vendar ko to počne več uporabnikov skozi daljše obdobje, ima lahko velik vpliv. Tu je prav tako ključnega pomena izobraževanje;

- **neučinkoviti sistemi** so pogost vir skritih stroškov. Pogost razlog za to je slab dizajn, ker se delovanje sistema ne ujema z načinom, kako uporabnik ali oddelek deluje. Pomembno je, da se pregledajo procesi in tok delovanja;

- **slabo delovanje sistema** vodi do nevidnih stroškov, ki so lahko kdaj tudi dobro vidni. Eden od razlogov, da so ti stroški težko zaznani, je, da so sestavljeni iz velikega števila manjših zamud. Poleg izgube produktivnosti zaradi zamudnega delovanja prihaja do nepotrebnih nadur, uporabnikove frustracije in slabih storitev za stranke. Sisteme, ki slabo delujejo, je bolje zamenjati in ne poskušati iztržiti vrednosti iz njih;

- **neprimerni sistemi** so uporaba produkta za neprimerne namene. Problemi se pogosto pojavljajo, ker uporabniki ne poznajo alternative. Podoben primer je, ko uporabniki poizkušajo rešiti vse probleme z istim orodjem. Da se to prepreči, so potrebne periodične revizije aplikacij, ki se uporabljajo;

- **prekomerne specifikacije** lahko povzročajo nepotrebne stroške. Sistemi, ki so premočni ali vsebujejo več funkcij, kot jih je potrebno za delovanje, pogosto nepotrebno trošijo čas in denar;

- **nepotrebna podvajanja** trošijo čas in denar z opravljanjem istega dvakrat ali celo večkrat. Pogosto se dogaja zato, ker ljudje niso seznanjeni, da je bilo nekaj že opravljeno. Rešitev se lahko najde v boljši komunikaciji;

- **nestandardni sistemi** povzročajo stroške, ko je treba prenesti datoteke ali podatke med sistemi in zaradi dodatnih izobraževanj zaposlenih. Standardizacija ima svoje slabosti, vendar prihrani denar;

- **nepravilna raba sistema** se zgodi, ko zaposleni uporabljajo sisteme za delo, ki ni mišljeno za njihovo delovno mesto. Primer je uporaba orodja za risanje zaposlenega, ki ni grafični oblikovalec.

Čeprav številne različne literature ponujajo različne metode in meritve z različnimi usmeritvami za reševanje težav pri vrednotenju naložb v IT, nobena od teh tehnik ne velja za formalno metodo vrednotenja za naložbe v IT. Anandarajan in Wen (1999) sta izjavila, da so raziskovalci poskušali razviti evalvacijske ukrepe za preverjanje učinkovitosti IT in nekatere od teh meritev, čeprav imajo akademsko vrednost, imajo težave z ezoterijo in težko operacionalizacijo. Tako formalne metode vrednotenja, ki jih obravnava večina industrij in organizacij, še vedno temeljijo na finančnih meritvah, kot so donosnost naložb, neto sedanja vrednost in notranja stopnja donosa. Iz tega izhajam, da je najprej treba razumeti koristi, ki jih prinese IT.

1.3.1 Koristi investicije v informacijsko tehnologijo

Interes v vrednost, ki jo prinese IT, je velik zaradi splošne percepcije, da zadnjih 20 let večina vlaganj v IT ni prinesla veliko merljivih koristi. Čeprav je ta percepcija lahko v veliko primerih zavedljiva, je interes v vrednosti, ki jih prinese IT, zdrav za IT (Bannister, 2004, str. 10).

Za definiranje dejavnikov, ki vplivajo na ocenjevanje investicijske upravičenosti avtomatizacije testiranj GUI, moramo razumeti koristi, ki se jih pričakuje od IT-investicije. Bannister (2004, str. 11–13) predstavi spisek pomembnih koristi s poslovnega vidika, ki naj bi jih prinesel IT:

- **zmanjšanje stroškov**: IT lahko zmanjša stroške na več načinov, npr. z zmanjšanjem števila zaposlenih ali zmanjšanjem volumna papirologije;

- **prestrukturiranje stroškov**: lahko je velikega pomena v mnogih organizacijah. Stroški so lahko prestrukturirani s premikom stroškov iz dela v kapital, spreminjanjem ravnovesja med proizvodnimi stroški in spreminjanjem dodeljevanja režijskih stroškov;

- **izboljšana učinkovitost**: IT lahko izboljša učinkovitost na veliko načinov, kot na primer z ukinitvijo ali zamenjavo neproduktivnega dela, izboljšanjem komunikacije med zaposlenimi, izboljšanjem natančnosti pri delu in dizajniranjem boljših produktov. Izboljšanje učinkovitosti ni nujno, da privede do nižanja stroškov;

- **izboljšanje trenutnih storitev strankam**: primeri so krajši čas naročanja, hitrejši odzivi na povpraševanje, manj dokumentacije, boljše poprodajne storitve in boljša komunikacija;

- **nudenje novih produktov in storitev strankam**: primeri so spletna naročila, 24-urna dostava, neposredna spremljava zalog z avtomatičnim ponovnim naročanjem, nudenje informacij o produktih ali trgu, spletni dostop do računov in tako dalje;

- **pridobivanje konkurenčne prednosti z diferenciacijo**: večina upravičenosti za IT je danes zaradi pridobivanja konkurenčne prednosti. Primer je nudenje krajšega dostavnega časa, dodajanje vrednosti produktu z minimalnimi stroški, nudenje strankam spletni dostop do informacij in uporabo IT za obojestransko koristne zaveze;

- **boljše odločanje**: to se lahko doseže z zaznavanjem najbolj profitabilnih produktov ali strank, uporabo različnih orodij, kot so analitična orodja in orodja za modeliranje odločitev;

- **izboljšanje morale zaposlenih in s tem produktivnosti**: morala zaposlenih in produktivnost se lahko izboljšata z nudenjem boljših informacij, nudenjem več

informacij, oblikovanjem boljšega delovnega okolja in odstranjevanjem dolgočasnih in ponavljajočih se nalog;

- **pomoč pri privabljanju in zadrževanju boljših zaposlenih**: dobri IT-sistemi lahko pomagajo pri privabljanju dobrih delavcev z nudenjem atraktivnih delovnih pogojev;

- **izboljšanje prikaza organizacijske strukture**: primer so večja javna prisotnost, pisma po meri, uporaba lojalnostnega programa in druge tehnologije kartic;

- **boljši vpogled v izvedbe in operacije**: s padanjem stroškov si lahko tudi manjše organizacije privoščijo orodja za modeliranje in analiziranje poslovanja. Primeri koristi so hitrejše poročanje, velike analitične sposobnosti, dostop do zunanjih informacij in modeliranje simulacij;

- **izboljšana kakovost produktov**: primeri dodajanja vrednosti produktom s pomočjo IT so dizajniranje s pomočjo računalnikov, avtomatizacija pregleda kakovosti, bolj natančne meritve itd.;

- **izboljšane komunikacije**: obseg današnjega poslovanja je podprt z IT-komunikacijami, ki nudijo ogromno širino, hitrost in globino komunikacijskih možnosti.

1.3.2 Način merjenja koristi avtomatizacije testiranj

Avtomatizacija testiranj ni vedno potrebna, primerna ali stroškovno učinkovita. V primerih, ko izvajamo odločitve na podlagi pričakovane donosnosti naložb, nas lahko analize usmerijo, kje nam lahko avtomatizacija testiranj prinese koristi. Ti donosi so najbolje izračunani s primerjavo stroškov in prihodkov, doseženih skozi avtomatizacijo testiranj v primerjavi z ročnim testiranjem. Dobre odločitve so lahko sprejete o uporabi avtomatizacije za izboljšavo testiranj, z identifikacijo in oceno stroškov in koristi avtomatizacije testiranj. To tudi pomaga pri identifikaciji, na katere faktorje se moramo osredotočati, da najbolje izkoristimo investicijo. Kadar je avtomatizacija potrebna zaradi pogodbe ali zaradi tehničnih ovir, izračun donosnosti naložbe lahko ne bo koristen. Neoprijemljivi faktorji lahko predstavljajo velik del donosnosti in tako izračun ne bo predstavljal prave vrednosti avtomatizacije. Ko želimo razumeti vrednost avtomatizacije testiranj, obstajajo številni pristopi. Da se izognemo napakam, napačnim pričakovanj in razočaranj z avtomatizacijo, je treba pazljivo oceniti relevantne faktorje za vsako testno situacijo in določiti, če je prisoten strošek ali če kaj pridobimo z avtomatizacijo, ter oceniti količino le-tega. Te faktorje se nato primerja med avtomatiziranim in ročnim testiranjem na enakem področju ali pa se agregira stroške in koristi skozi niz ročnih in avtomatiziranih testov (Hoffman, 1999).

Silva, Abreu in Jino (2009) so predstavili pristop, ki sloni na učinkovitosti ekipe, in razvili model ocene truda za oceno porabljenega truda pri izvrševanju ročnih testov. Pred razvojem modela so avtorji izvedli nekaj korakov, ki razložijo pristop. Koraki vključujejo analizo

podatkov truda, formulirane hipoteze in akumulirano učinkovitost. Akumulirana učinkovitost je metrika truda, ki predstavlja vsoto vseh korakov, ki jih je izvršila ekipa za vse testne primere, deljeno z vsoto časa, porabljenega za izvršene teste, ki jih lahko uvrstimo pod te korake. Model je naslednji:

$$T = (1 + r) * \frac{S}{WE_A} \quad (1)$$

T predstavlja oceno truda izvajanja v časovnih enotah, r predstavlja relativni koren povprečne kvadratne napake, S predstavlja število korakov v testnih primerih in WEA predstavlja povprečno ponderirano aritmetično sredino (Silva, Abreu & Jino, 2009).

Z vidika avtomatiziranih testov pa so Almeida, Abreu in Moraes (2009) predstavili metodo, imenovano ponderirana metoda Nageswaran (ang. Weighted Nageswaran method), za oceno truda, ki temelji na primerih uporabe. Metoda razdeli primere uporabe na dva tipa, tiste, ki imajo normalen tok, in ostale, ki imajo izjemen tok. Primeri, ki so izjeme, so del normalnih primerov. Metoda najprej identificira akterje, nato jih ponderira v skladu z njihovimi interakcijami s sistemom in se pridobi vrednost, imenovana neprilagojen akterski ponder (ang. Unadjusted Actor weight (UAW)), ki predstavlja vsoto pondrov vseh akterjev. Kot drugo dodeli ponder vsakemu primeru uporabe po naslednji formuli:

$$P_T = P_N * N + P_E * E \quad (2)$$

PT predstavlja ponder primera uporabe, PN predstavlja ponder normalnega primera, PE predstavlja ponder izjemnega primera, N se nanaša na število normalnih primerov in E na število izjemnih primerov. Vsota vseh se imenuje neprilagojen ponder primera uporabe (ang. Unadjusted Use Case Weight (UUCW)) (Almeida, Abreu & Moraes, 2009).

Kot tretje upošteva tehnične in okoljske faktorje, katerim dodeli vrednost glede na njihovo razpoložljivost in stopnjo pomembnosti, jih pomnoži in se dobi vrednost, imenovana tehnično-okoljski faktor (ang. Technical Environment Factor (TEF)). Kot četrto se pridobi končna vrednost, imenovana prilagojena točka primera uporabe (ang. Adjusted Use Case Point (AUCP)), z naslednjo formulo:

$$AUCP = UUCP * [0, 65 + (0, 01 * TEF)] \quad (3)$$

Kjer je UUCP enak seštevku UAW in UUCW.

Končni trud se pridobi z naslednjo formulo:

$$Trud = AUCP * \left[\frac{Človeška\ ura}{Točka\ primera\ uporabe}\right] \quad (4)$$

Aranha in Borba (2007) sta govorila o trudu, porabljenem na avtomatizaciji testov. Avtorja razložita, da lahko testni inženirji določijo, kateri testi bi se morali avtomatizirati z upoštevanjem truda, ki je potreben za avtomatizacijo in izvedbo vsakega testa.

Gulechha (2013) predstavi formulo za izračun testne učinkovitosti pri identificiranju napak.

$$Testna\ učinkovitost = \left[\frac{DT}{DT+DU} * 100\right]\% \quad (5)$$

DT predstavlja število odkritih napak, DU pa število napak, odkritih s strani uporabnikov v produkciji.

Gulechha (2013) predstavi še formulo za izračun produktivnosti pisanja skript, ki jo lahko uporabimo za spremljanje krivulje učenja.

$$Produktivnost\ grajenja\ skript = \left[\frac{\sum Opravljeni\ koraki}{Trud\ (število\ ur)}\right] Korakov/uro \quad (6)$$

Kjer opravljeni koraki pomeni:

- število klikov na podatke, ki se osvežijo,
- število vhodnih parametrov,
- število dodanih točk preverjanja.

Avtomatizacija testiranj prinese oprijemljive in neoprijemljive koristi (TechArcis Solutions, Inc., 2018). Oprijemljive koristi so vrste koristi, ki jih je mogoče izmeriti z objektivnim, količinskim in pogosto finančnim ukrepom in nasprotujejo neoprijemljivim koristim, ki jih je mogoče le subjektivno presojati (Ward & Daniel, 2006, str. 20). V primeru avtomatizacije testiranj so oprijemljive koristi lahko povezane s pridobljenim časom, neoprijemljive koristi pa so lahko sestavljene iz hitrejših povratnih informacij s strani ljudi in hitrejšega odkrivanja napak v razvojnem ciklu (TechArcis Solutions, Inc., 2018).

Neoprijemljive koristi ostajajo vprašanje za vse finančne meritve. Različne literature so trdile, da vse finančne meritve niso uspele zajeti neoprijemljivih koristi, ki jih je ustvarila naložba (Antine, Eph & Stray, 1998). Oliver, Barrick in Janicki (2006) navajajo več razlogov, zakaj izzivi kažejo na neoprijemljive koristi pri meritvah vrednotenja. Ti izzivi vključujejo ugotavljanje neoprijemljivih koristi naložbe, razvijanje standardnih meritev za te koristi, vključitev teh koristi v finančne meritve, odkup s poslovnimi enotami na zahtevane ugodnosti in vrednotenje po projektu. Gomez in Pather (2012) sta dodatno povzela vprašanje zajemanja neoprijemljivih koristi, ki jih je težko količinsko opredeliti in jim težko postaviti finančno vrednost. Kot rezultat številne organizacije posvečajo malo pozornosti k neoprijemljivim koristim pri odločitvah za investicijo (Lin, Graham & McDermid, 2005). Bannister (2004, str. 81) pravi, da je v praksi edini način učinkovito izmeriti neoprijemljive koristi tako, da se vpraša tiste, ki koristi dobijo, in oni sami določijo njihovo vrednost.

1.3.3 Značilnosti vrednotenja investicij v avtomatizacijo testiranj

Največja prednost avtomatskega testiranja so kratek čas trajanja testov in nizki stroški. V primerjavi z ročnim testiranjem na stroške in koristi avtomatiziranega testiranja v glavnem vplivajo sledeči dejavniki (Cui & Wang, 2015):

- **testna orodja**: avtomatsko testiranje mora imeti orodja za podporo. Nekatera orodja za avtomatsko testiranje in orodja za razvoj skript zahtevajo licence. Stroške je mogoče oceniti s stroški nakupa, pomnoženimi z amortizacijsko stopnjo;

- **usposabljanje**: za avtomatsko testiranje potrebujejo testerji dodatno znanje. Testerji morajo sprejeti usposabljanje za orodja in standardizacijo razvoja. Stroške je mogoče oceniti s časom usposabljanja (enota: dan osebe), pomnoženim s plačilom časa enote;

- **oblikovanje testov in razvijanje primerov**: pri avtomatiziranem testnem procesu orodja igrajo le podporno vlogo, glavno vlogo pa ima dizajn testnega okvira in testnih primerov. Standard razvoja neposredno določa, ali je avtomatizirano testiranje uspešno. Stroške je mogoče oceniti glede na porabljen čas za dizajn (enota: ura), pomnožen s plačilom časa enote;

- **testiranje**: potek testa je postopek vodenja testnih primerov in analiza rezultatov testa. Stroške je mogoče oceniti s časom izvajanja (enota: ura za krog), pomnoženim s frekvenco testov, pomnoženo s plačilom časa enote;

- **vzdrževanje skript**: ko izide nova različica aplikacije, se lahko funkcije in vmesniki aplikacije spremenijo. Zato je treba testne skripte prilagoditi novi različici. Stroške je mogoče oceniti z vsoto števila sprememb v aplikaciji, pomnoženo s stroški dizajna primerov.

Sypolt (2015) predstavi ključne indikatorje koristi avtomatizacije testiranj v primerih kontinuirane integracije, ki pokažejo stanje projekta in območja za izboljšavo:

- čas, potreben za izvedbo testa,
- količina uspešnih in neuspešnih testov,
- čas, potreben od razvoja določene kode in prenosa do produkcije.

Pri podjetju TechArcis Solutions, Inc. (2018) so izpostavili sledeče ključne indikatorje koristi avtomatizacije testiranj:

- pokritost testiranja,
- čas, pridobljen z vsakim ciklom regresijskih testov,
- število najdenih napak na cikel regresijskih testov,
- stroški priprave avtomatiziranih testov.

Problem klasične kalkulacije donosnosti naložbe je, da ne moremo primerjati ročnega testiranja z avtomatiziranim testiranjem, ker ne moremo opraviti enake količine ročnih in avtomatiziranih testov. Prava vrednost donosnosti naložbe avtomatizacije so koristi, ki izhajajo iz tega tipa testiranj in to je lahko skrajšan čas do prehoda v produkcijo, povečana učinkovitost testov in povečana uspešnost testov. Ne smemo pa pozabiti na stroške avtomatizacije, primarno iz vzdrževanja skript (TechArcis Solutions, Inc., 2018).

2 UVEDBA ORODJA SAHI V PODJETJU BANKART

2.1 Predstavitev podjetja

Leta 1997 se je večina slovenskih bank odločila ustanoviti družbo za procesiranje plačilnih instrumentov, ki je začela redno poslovati 1. 4. 1998. Družba Bankart je bila ustanovljena zaradi zniževanja operativnih in razvojnih stroškov ter poenotenja na področju samopostrežnega in kartičnega poslovanja.

Temeljno poslanstvo Bankarta je zagotoviti zanesljivo, varno in stroškovno učinkovito obdelavo transakcij z različnimi bančnimi plačilnimi instrumenti. Poslanstvo družbe je, da s skrbnim razvojem, gradnjo in vzdrževanjem ustreznega informacijskega okolja omogoči vsem odjemalcem nemoteno in kakovostno uporabo storitev Bankarta.

Vizija Bankarta je usmerjena v iskanje novih priložnosti in izzivov na območju jugovzhodne Evrope; organizirati in usposobiti družbo tako, da bo v celoti kos izzivom, ki jih prinašajo tržne zakonitosti in sodobno konkurenčno okolje.

Cilj družbe je s ponudbo visokokakovostnih in tehnološko najzahtevnejših storitev utrditi položaj vodilnega slovenskega procesnega centra in postati eden najkakovostnejših mednarodnih procesnih centrov v regiji.

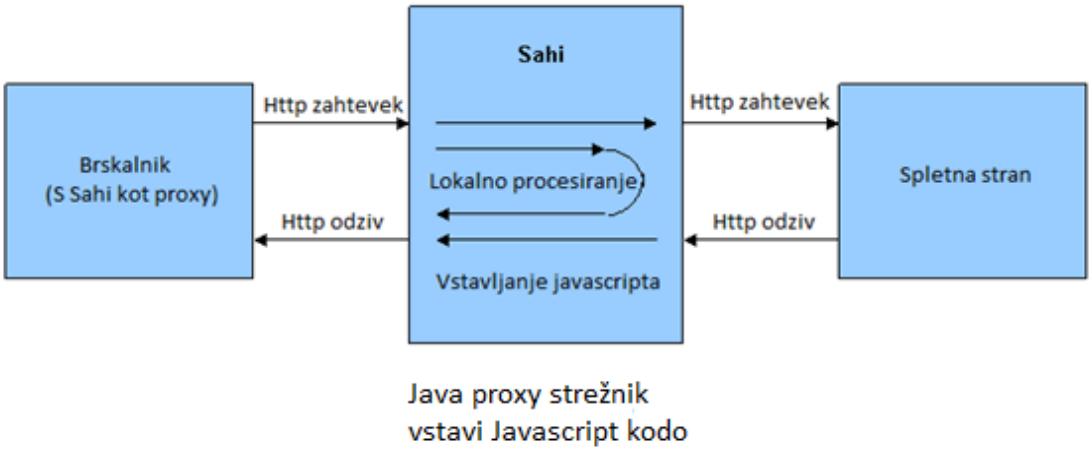
Poslovne usmeritve temeljijo na nenehnem razvoju in tehnični dovršenosti, upoštevanju želja in potreb bank in drugih finančnih institucij, implementiranju razvojnih trendov na področju sodobnega plačilnega prometa in implementiranju zahtev mednarodnih kartičnih organizacij. Osnovni cilj sta nadaljnja kontinuirana rast in razvoj, kar Bankart dosega prek zastavljenih dolgoročnih ciljev:

- razvijanje novih dejavnosti in s tem izpolnjevanje celovite ponudbe storitev,
- stalna racionalizacija poslovanja in povečevanje učinkovitosti,
- nadaljnji razvoj celovitega informacijskega sistema in lastne komunikacijske infrastrukture,
- sprotno prilagajanje kadrovske in organizacijske strukture razmeram, ki jih narekujejo trg in nove tehnologije,
- nenehna širitev obsega poslovanja.

2.2 Predstavitev orodja

Orodje Sahi je bilo prvotno odprtokodno (ang. Open source) orodje za testiranje, osredotočeno na avtomatizacijo prihajajočih spletnih tehnologij, ki je postalo orodje, usmerjeno v testerje. Razvito je bilo v jezikih Java in Javascript ter gostovano na SoundForgu. Ima zmožnost snemanja in ponovnega predvajanja katere koli spletne aplikacije na katerem koli brskalniku in na katerem koli operacijskem sistemu. Nekatere od zmožnosti Sahija so kontrole znotraj brskalnika, inteligentni snemalec, tekstovne skripte, podpora AJAX, dobra, že vgrajena poročila itd. Testiranje spletnih aplikacij je lahko problematično zaradi različnih brskalnikov, ker ima vsak svoje posebnosti. Kot prikazano na sliki 2, Sahi deluje kot približek (ang. Proxy), strežnik, ki prestreže promet iz spletnega brskalnika. Sahi nato vstavi upravljavce dogodkov Javascript na spletno stran, kar omogoča snemanje in ponavljanje dogodkov v brskalniku.

Slika 2: Arhitektura delovanja orodja Sahi



Vir: http://sahipro.com/docs/introduction/architecture.html.

Tako je Sahi z uporabo strežnika proxy neodvisen od brskalnika, ki se uporablja. Pogoj je, da brskalnik uporablja jezik Javascript in dovoljuje postavljanje strežnika proxy. V tabeli 1 so našteti podprti brskalniki.

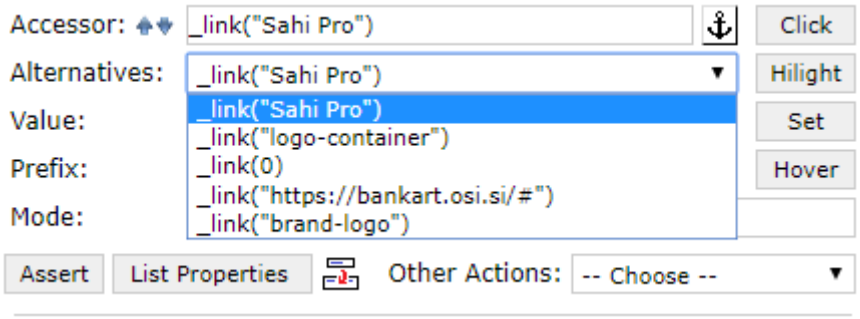
Tabela 1: Podprti brskalniki

Brskalnik	Verzija
Internet Explorer	6+
Mozilla Firefox	2+
Google Chrome	6+
Safari	5+
Opera	9+
PhantomJS	1+
Microsoft Edge	25+

Vir: http://sahipro.com/docs/introduction/index.html#Sahi%20Pro.

Za identifikacijo elementov Sahi uporablja lastne ovijalke okoli dokumentno objektnega modela (v nadaljevanju DOM) JavaScripta. Kot vidno na sliki 3, API-ji orodja Sahi uporabljajo različne atribute DOM elementa za identifikacijo. Atributi, ki jih Sahi išče, so lahko vrednost (ang. Value), ime (ang. Name), ID, css, indeks itd.

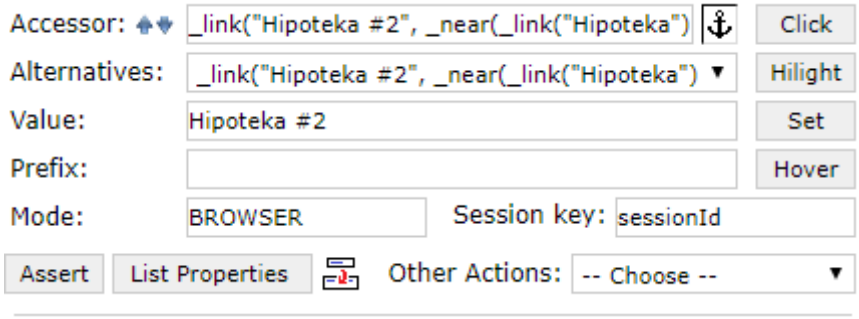
Slika 3: Primer možnih atributov za identifikacijo elementa



Vir: Lastno delo.

Sahi uporablja tudi relacijske API-je, kot so blizu (ang. near), v (ang. in), pod (ang. under), levo od (ang. leftOf), desno od (ang. rightOf) itd., da identificirajo en element glede na relacijo z drugim. Uporaba je prikazana na sliki 4. API-ji so normalizirani tako, da delujejo v vseh brskalnikih.

Slika 4: Uporaba relacijskega API »near« za identifikacijo elementa



Vir: Lastno delo.

Sahi Script, privzeti jezik, ki ga uporablja Sahi, je razširitev jezika Javascript. Sahi Script uporablja iste konstrukte Javascripta, vendar dodaja sposobnost za učinkovito interakcijo z brskalnikom in izvaja tudi sistemske akcije, kot so branje iz podatkovnega sistema, dostop do podatkovne baze, klic Java itd.

Sahijev Javascript se izvaja v Rhino Javascript interpretatorju, ki deluje znotraj Sahijevega proxyja. Samo relevantne posamezne izjave se pošljejo v brskalnik za izvedbo, preostalo izvajanje pa se dogaja v Rhinu znotraj Sahijevega proxyja.

Rhino je Javascript runtime, ki se izvaja znotraj virtualnega stroja Java (ang. Java Virtual Machine). To doda skriptam Sahi veliko moči, saj lahko neposredno skličejo Java kodo v skriptah.

2.3 Odločitev o uvedbi orodja

Različne ekipe znotraj Bankarta imajo skrbništvo nad različnimi aplikacijami. Te aplikacije zahtevajo kontinuiran proces razvoja in posledično tudi redno testiranje. V okviru ekipe B2B, katere sem tudi sam član, je skrbništvo več spletnih aplikacij. Ekipa deluje po metodologiji scrum razvoja aplikacij. Pred uvedbo orodja je v ekipi B2B testiranje aplikacij okvirno potekalo tako, da je tester najprej zapisal scenarije testov, ki jih želi narediti v aplikaciji. Ko so bili scenariji zapisani, je opravil testiranje aplikacije po zapisanih scenarijih in si sproti zapisoval, če je prišlo kje do napake pri izvedbi. Ta proces je vzel veliko časa, ko so se daljši scenariji začeli ponavljati, zaradi vedno novih popravkov in dodajanja funkcij v aplikaciji, hkrati je postajala tudi vse večja možnost, da so se napake spregledale in prenesle vse do produkcije. Samo hranjenje zapisov že opravljenih testov in brskanje po njih je tudi predstavljalo zmedo. Podobne težave so imele tudi ostale ekipe.

Tako se je sprejela odločitev, da se v nadaljevanju s pomočjo orodij izboljša in optimizira proces testiranja. Sama odločitev je bila skladna z namenom podjetja, ki je zniževanje stroškov, in po pregledu literature je to ena izmed značilnosti orodja za avtomatizacijo in splošno investiranja v IT.

Pričakovanja od orodja se je pridobilo iz internega dokumenta podjetja, ki je služil kot predlog za vpeljavo orodja. Pričakovanja so bila sledeča:

- prihranek v času,
- višja kakovost testov,
- boljši pregled nad opravljenimi testi,
- boljša komunikacija z razvijalci,
- ponovna uporabnost testov.

2.4 Izbira orodja

Izbira orodja je potekala tako, da smo preiskali po internetu orodja za avtomatizacijo testiranj GUI, ki so na voljo. Tako smo največkrat odkrili orodja Sahi in Selenium, ki sta prikazana v tabeli 2, ter naredili primerjavo, da ugotovimo, katero orodje je najbolj primerno za nas.

Po analizi obeh orodij smo prišli do odločitve, da je orodje Sahi v našem primeru primernejše, ker je za nas pomembno, da lahko testiramo na različnih brskalnikih, in da je orodje bolj primerno za testerje, ki nimajo veliko znanja o programiranju.

Tabela 2: Primerjava orodij Selenium in Sahi

	Selenium	Sahi
Podprt brskalnik	Firefox	Vsi
Pojavna okna	Ima težave pri snemanju	Lahko snema
Okvir	Uporablja XPath za identifikacijo elementov, če ID ali naziv ni na voljo	Uporablja različne algoritme za identifikacijo elementov za lahko prepoznavo
Programski jezik	Java, Ruby, Perl, Python, C# in še veliko več	Sahi Script, Java, Ruby
Težavnost uporabe	Enostavno začeti z uporabo v roku petih minut	Nekoliko težji začetek uporabe, ker je potrebna namestitev orodja
	Potrebno je znanje programiranja	Večina avtomatizacije je lahko dosežena s funkcijami in variabilami. Ima vgrajene API-je za večino kompleksnejših nalog
	Potrebuje vstavljanja tako imenovanega čakanja, da lahko AJAX deluje	V večini primerov ne potrebuje čakanja
	Podpira paralelno delovanje skript	Podpira paralelno delovanje skript
Poročila	V realnem času prikazuje korake. Dnevnik korakov se lahko shrani, če se namesti dodatek.	V realnem času prikazuje korake. Dnevnik vseh skript, ki so bile kdaj izvršene, in pregleda vse korake in se lahko primerja skripte med sabo

Vir: Povzeto po Kaizen testing (2012).

2.5 Implementacija orodja

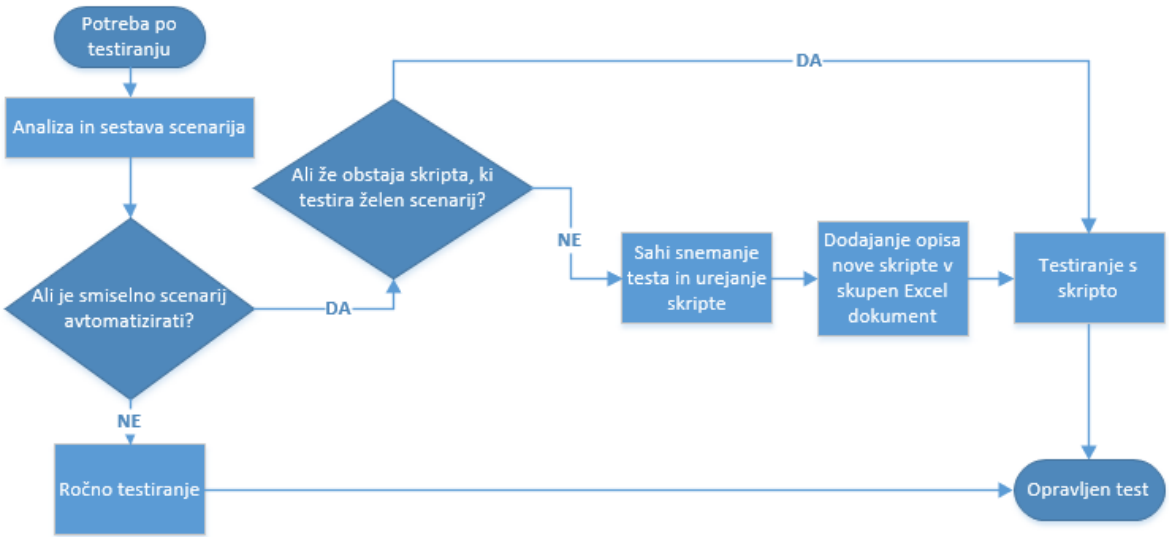
Implementacija orodja se je začela z udeležbo na izobraževalni delavnici pri izbranem podjetju, ki služil kot podpora za orodje. Na delavnici je bilo predstavljeno osnovno delovanje in namen orodja, kako se orodje namesti, kako se ustvari in spreminja skripte ter kako se namesti potrebne certifikate. Tisti, ki smo se izobraževanja udeležili, smo pridobljeno znanje prenesli do ostalih bodočih uporabnikov. Preden smo se tega lotili, je vsak analiziral, kako najbolje uporabiti orodje in kako ga implementirati v obstoječ proces delovanja.

Bankart je pridobil tri licence Sahi pro, kar pomeni, da lahko trije uporabniki istočasno uporabljajo orodje. Za upravljanje z licencami je bil vzpostavljen strežnik Sahi licence virtualni, na katerega so bile nameščene ustrezne komponente. Strežnik je služil tudi kot baza za skupen dostop do skript in poročil.

Projekti avtomatizacije testiranj lahko uporabijo tehniko snemaj in ponovi (ang. capture-reply), tehniko skriptiranja (ang. scripting) ali pa kombinacijo obeh za ustvarjanje testov (Boehmer & Patterson, 2001). Prvi poskusi avtomatizacije testiranj pogosto slonijo na uporabi tehnike snemaj in ponovi, da se vidi, ali avtomatizacija testiranj ustreza v določenem primeru in splošno deluje ta pristop koristen, vendar je na dolgi rok ta pristop pogosto neuspešen in je zato opuščen (Allott, 1999).

V ekipi B2B sem namestil in pripravil orodje za uporabo svojim sodelavcem in jim predstavil nov proces testiranja in praktično uporabo orodja. Novi predlagani proces je viden na sliki 5 in je sledeč: uporabnik najprej analizira in pripravi scenarij, ki bi ga želel testirati. Drugi korak je, da se uporabnik vpraša, ali obstaja potreba po avtomatizaciji, če je ni, se ročno testira. Če je potreba po avtomatizaciji, se preveri, ali že obstaja skripta, ki testira želen scenarij. Če obstaja, se skripto predvaja z orodjem, če še ne obstaja, se sledi kombinirani tehniki ustvarjanja testov, in sicer snemanje testa z orodjem in nato urejanje skripte. Ko je skripta shranjena, je treba dodati še njen opis v skupni Excelov dokument. Zadnji korak je testiranje s skripto in opravljen test.

Slika 5: Proces testiranja



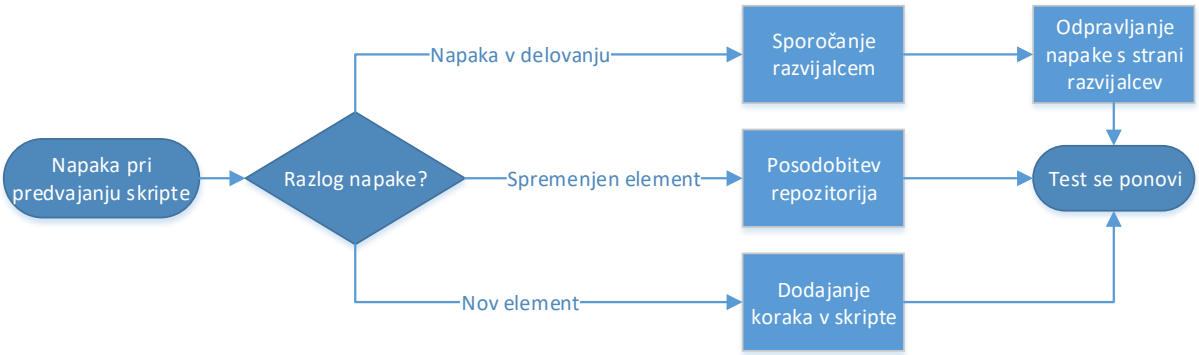
Vir: Lastno delo.

Za boljšo organizacijo se je ustvaril Excelov dokument na skupni lokaciji, ki vsebuje opise obstoječih skript. Namen je, da se razume, kaj testira posamezna skripta. Vse nove skripte se dodajajo na isti način v dokument, ločeno po zavihkih po aplikacijah.

Z implementiranjem orodja v proces testiranja se je pojavil tudi nov proces preverjanja napak, prikazan na sliki 6. Proces je sledeč: ko pride do napake pri predvajanju skripte, se pogleda v zapisnik orodja, kjer je razvidno, kje je prišlo do napake. Lahko je napaka v delovanju aplikacije in se napako komunicira razvijalcem, lahko je sprememba v aplikaciji in je treba posodobiti skripto. Če se je spremenil obstoječi element v aplikaciji, ki ga skripte že uporabljajo, se posodobi element v repozitoriju. Tako se prenesejo spremembe na vse

skripte s tem elementom. Če je razlog za napako nov element, dodamo korak v skripte, kjer je potrebno.

Slika 6: Proces odprave napak pri izvedbi skripte



Vir: Lastno delo.

2.6 Uporaba orodja

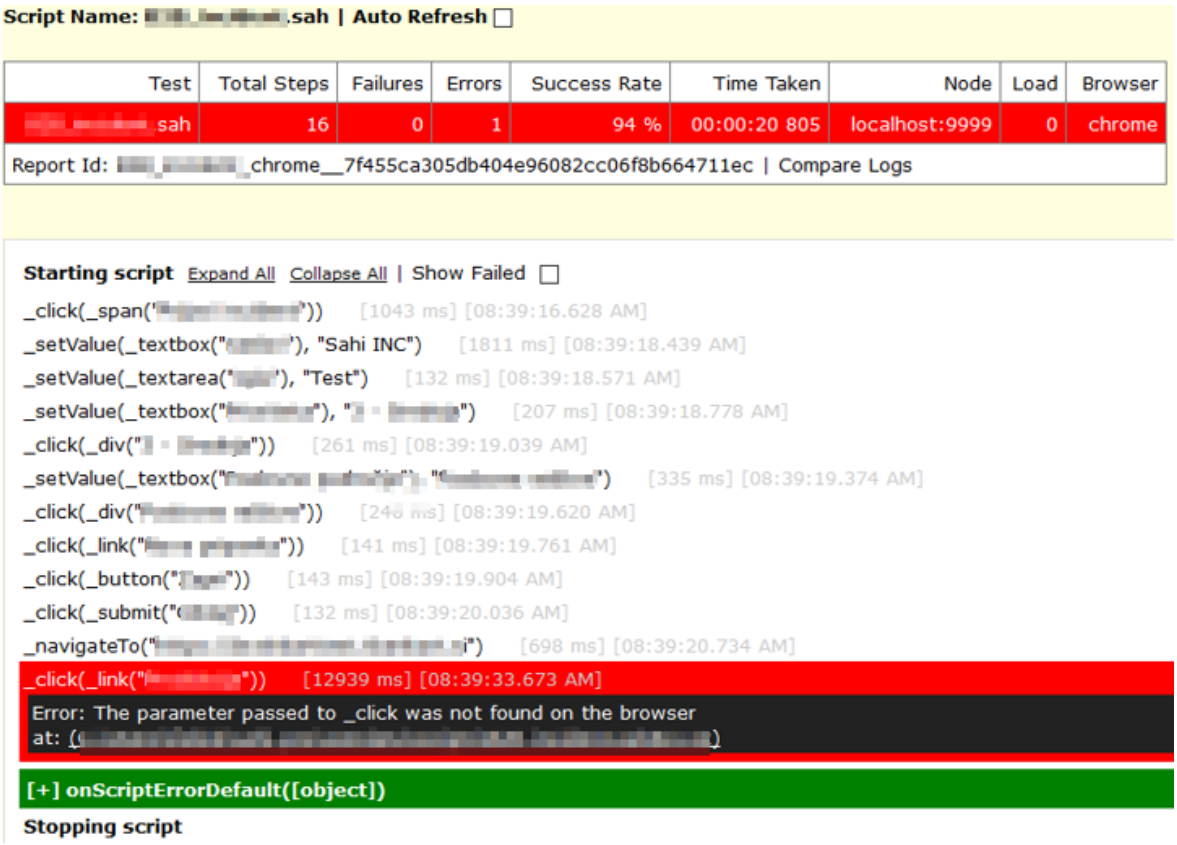
V obdobju opazovanja se je orodje uporabljalo na šestih aplikacijah. Za potrebe upravičevanja implementacije orodja se je določilo, da se bo natančneje merila uporaba na eni aplikaciji, ker se ni želelo izgubiti preveč časa, da bi si vsi uporabniki natančno spremljali uporabo, bo pa njihovo mnenje za nadaljnjo uporabo upoštevano. Naloga merjenja uporabe je bila dodeljena moji ekipi.

Sledilo se je navodilom novega procesa, prikazanega na sliki 5. Najprej se je analiziralo, kateri so scenariji oziroma testi, ki so se največ ponavljali in jih je možno avtomatizirati. Izbrane scenarije se je z orodjem posnelo in dopolnilo v urejevalniku skript. Skripte je bilo treba po snemanju urejati zato, ker je snemanje odvisno od tega, kako je aplikacija zgrajena, kar pomeni, da je v določenih primerih en klik na element v aplikaciji predstavljal dva elementa v kodi in je posledično skripta zabeležila dva klika, kjer mora biti en. S časom se je izkazalo, da je snemanje korakov najbolje za postavitev začetnega ogrodja skripte in nato ročno dopolniti skripto. V primerih, ko ne znamo narediti določenega koraka v skripti, ima orodje povezavo do bogate dokumentacije s prikazanimi primeri, kako napisati želeni korak v skripti. V redkih primerih, ko nismo našli odgovora v omenjeni dokumentaciji, smo pregledali odgovore na internetnem forumu orodja, vendar tam pogosto nismo našli odgovora, ker skupnost ni tako velika, in smo v takih primerih kontaktirali podjetje, ki nam služi za podporo pri orodju. Ko so bile skripte dokončane, smo jih vnesli s potrebnim opisom v skupni dokument.

Testiranje je splošno potekalo tako, da ko se je v aplikacijo implementirala nova funkcija ali pa naredila sprememba že obstoječe, se je najprej pognalo skripte, ki so opravile dimni test. Po dimnem testu se je testiranje nadaljevalo dinamično z ročnim in avtomatiziranim testiranjem. V praksi se je večinoma začelo z ročnimi testi funkcije pod testom in ko smo prišli do koraka, ki je avtomatiziran, smo pognali skripto ter nato nadaljevali z ročnimi testi. Sproti smo bili pozorni, če se pojavi možnost avtomatizacije. Avtomatizirani testi niso

testirali samo toka aplikacije, ampak tudi pravilne vrednosti, prisotnost in vidnost elementov. Če je skripta javila napako, smo poleg tega, da smo spremljali že med delovanjem, kaj dela skripta po aplikaciji GUI, pogledali v dnevnik in videli, kaj je napako povzročilo, kot se vidi na sliki 7.

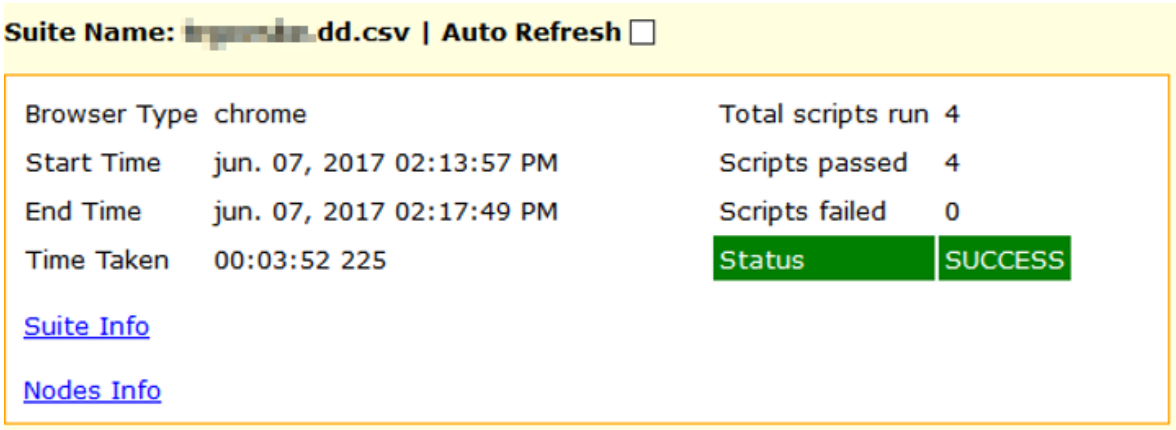
Slika 7: Primer napake v dnevniku



Vir: Lastno delo.

Proženje več skript paralelno hkrati je bilo časovno zelo učinkovito, vendar je bilo z vidika testerja v primeru napake težje razumeti, kaj je povzročilo napako, ker ni možno spremljati več zaslonov hkrati in čeprav lahko vidimo v dnevniku, na katerem koraku se je zgodila napaka, je lahko napaka posledica prejšnjih korakov, kar pa lažje ugotovimo, če spremljamo delovanje testa. Proženje več skript hkrati, je prikazano na sliki 8.

Slika 8: Primer proženja štirih skript hkrati



Vir: Lastno delo.

Problem, ki se je pojavil, je bil pri uporabi več brskalnikov, ker zaradi varnostnih nastavitev v podjetju ni mogoče spremeniti proxy nastavitev vseh brskalnikov. Tako smo omejeni na Chrome in Internet Explorer, kar pa je za nas zadostno z vidika testiranj, ker večina naših strank uporablja prav ta dva brskalnika.

Za potrebe merjenja uporabe orodja se je beležilo:

- **čas gradnje skripte** pomeni čas, porabljen, da se je določena skripta dokončala. To pomeni, da se je pri gradnji skript spremljal čas, porabljen od začetka snemanja testa, vključno s testiranjem skripte, če deluje kot želeno, dokler ni bil vnesen opis skripte v skupen Excelov dokument, kar je pomenilo, da je skripta pripravljena za uporabo;

- **produktivnost grajenja skript** pomeni čas, izračunan po formuli Gulechha (2013) za izračun produktivnosti grajenja skript, predstavljeno v točki 1.3.2;

- **čas trajanja skripte** pomeni čas, ki ga skripta porabi, da opravi test. Čas je pridobljen iz dnevnika opravljenih testov in predstavlja približno povprečje;

- **čas ročnega testiranja** pomeni čas, ki se porabi v primeru, da bi ročno testirali vse korake, ki jih opravi skripta. Čas se je pridobil, tako da smo dvakrat opravili ročni test in merili čas ter zapisali približno povprečje;

- **število ponovitev** pomeni, kolikokrat v celotnem obdobju merjenja se je uporabila skripta. Število ponovitev je bilo pridobljeno iz dnevnika opravljenih testov, kjer niso bili upoštevani testi, opravljeni v sklopu grajenja in vzdrževanja skripte;

- **čas vzdrževanja** pomeni skupen čas, ki se je porabil za popravljanje ali dodajanje korakov v skripti. Čas je bil pridobljen tako, da je tisti, ki je skripto vzdrževal, zapisal okviren porabljen čas v skupno datoteko, kjer smo spremljali vzdrževanje.

Porodila se je tudi ideja, da se meri tudi čas, ki se porabi za odpravo napak, da bi s tem primerjali, kako hitro se odpravijo napake, če komunikacija z razvijalci izhaja iz avtomatiziranega testa ali iz ročnega testa. Za to se nismo odločili, ker smo ocenili, da ne moremo teh dveh časov primerjati med seboj, ker jih je težko meriti in zaradi unikatnosti primerov. Upoštevalo se bo mnenje razvijalcev in testerjev, kako je orodje vplivalo na komunikacijo.

3 PREGLED REZULTATOV

3.1 Rezultati uporabe orodja

V nadaljevanju so predstavljene tabele meritev, kjer so v tabeli 3 prikazane meritve v obdobju prvih šestih mesecev, v tabeli 4 pa meritve drugih šestih mesecev. Meritve smo razdelili v dve obdobji zato, ker smo po prvih šestih mesecih opazili, da so bile skripte neoptimalno zgrajene in jih je smiselno ponovno zgraditi za lažjo nadaljnjo uporabo. S tem smo porabili nekaj časa, vendar smo bili mnenja, da se proces izboljša in olajša. Pridobili smo tudi dobro sliko krivulje učenja gradnje skript, ki je prikazana na sliki 9.

Skripte so poimenovane s kodami skript, sestavljenih iz črke in številke, kjer skripte z isto črko testirajo enake funkcije aplikacije pod drugačnimi pogoji. Točna imena skript so maskirana zaradi poslovne skrivnosti, saj so imena sestavljena tako, da razkrijejo veliko informacij o funkciji aplikacije, ki jo skripta testira. Časovne meritve so bile zabeležene z zaokroženim povprečnim časom in so prikazane v formatu ure:minute:sekunde oziroma minute:sekunde.

Tabela 3: Meritve prvih šestih mesecev

Koda skripte	Čas gradnje skripte	Število korakov v skripti	Čas trajanja skripte	Čas ročnega testiranja	Število ponovitev
A1	4:00:00	56	1:10	3:10	188
A2	1:00:00	46	0:50	2:00	122
A3	15:00	50	1:00	2:20	68
A4	15:00	50	1:10	3:10	27
A5	15:00	50	1:10	3:10	27
A6	15:00	50	1:10	3:10	54
B1	10:00	11	0:30	1:00	211
B2	10:00	13	0:15	0:50	58
C1	3:20:00	57	0:50	3:00	254
D1	5:40:00	93	2:50	6:00	34
E1	20:00	37	2:10	2:50	55
E2	10:00	14	0:40	0:50	55

se nadaljuje

nadaljevanje

Koda skripte	Čas gradnje skripte	Število korakov v skripti	Čas trajanja skripte	Čas ročnega testiranja	Število ponovitev
E3	10:00	14	0:40	0:50	55
F1	1:00:00	60	2:20	3:30	31
F2	20:00	60	2:20	3:30	31
F3	15:00	55	2:10	3:10	48
F4	15:00	55	2:10	3:10	31
G1	4:30.00	203	3:55	12:00	19
G2	1:15:00	226	3:00	15:00	11
H1	30:00	120	2:50	7:30	62
H2	15:00	101	1:40	5:00	44
H3	15:00	61	1:20	3:00	31

Vir: Lastno delo.

Tabela 4: Meritve drugih šestih mesecev

Koda skripte	Čas gradnje skripte	Število korakov v skripti	Čas trajanja skripte	Čas ročnega testiranja	Število ponovitev
A1	25:00	85	1:30	3:10	97
A2	7:00	76	1:10	2:00	21
A3	7:00	80	1:10	2:20	18
A4	7:00	80	1:20	3:10	10
A5	7:00	80	1:20	3:10	10
A6	7:00	80	1:20	3:10	10
B1	5:00	25	0:30	1:00	103
B2	5:00	26	0:15	0:50	10
C1	20:00	81	0:50	3:00	132
D1	35:00	120	2:50	6:00	5
E1	15:00	38	2:10	2:50	72
E2	5:00	14	0:40	0:50	72
E3	5:00	14	0:40	0:50	72
F1	30:00	66	2:30	3:30	83
F2	10:00	67	2:30	3:30	42
F3	10:00	62	2:20	3:10	42
F4	10:00	62	2:20	3:10	42
G1	25:00	274	4:10	12:00	2
G2	10:00	258	3:00	15:00	3
H1	5:00	161	3:10	7:30	11
H2	5:00	137	2:10	5:00	11
H3	5:00	80	1:20	3:00	11
I1	25:00	45	1:00	4:20	19
I2	5:00	45	1:00	4:20	18
I3	5:00	45	1:00	4:20	18
J1	20:00	62	2:15	6:40	11

Vir: Lastno delo.

Najbolj pogosto uporabljene skripte predstavljajo dimne teste. V mislih je treba imeti, ali bi bili dimni testi tako obsežni in pogosti v primeru ročnih testov. V tem primeru bi verjetno testirali le funkcijo, ki je pod testom, in naredili dimni test le, če bi bili mnenja, da je velika verjetnost, da funkcija pod testom vpliva na drugo kodo.

Vidimo, da se je v primerjavi med tabelo 3 in tabelo 4 povečal čas trajanja nekaterih skript. Razlog je v tem, da smo zaradi večjega znanja o gradnji skript nastavili dodatna preverjanja znotraj skript za kakovostnejše teste. Na drugi strani se je čas vzdrževanja zmanjšal s 26 ur na 9 ur in 20 minut zaradi bolj optimalno zgrajenih skript, ki so omogočile lažje spreminjanje skript. Vendar je to lahko posledica unikatnih sprememb v aplikaciji, lahko zato, ker so bile skripte do takrat že tolikokrat popravljene, da je težje prihajalo do napak, in ker je bil razvoj na aplikaciji v prvi polovici bolj intenziven, kar lahko vidimo iz števila ponovitev, kjer je bilo število pognanih skript v prvi polovici 1516, v drugi polovici pa 945.

Slika 9 prikazuje krivuljo učenja gradnje skript skozi čas, kjer se vidi, da se produktivnost grajenja skript zmerno povečuje. Bilo je izračunano, da je bila povprečna produktivnost prve polovice meritev 2,4 koraka na minuto, v drugi polovici pa 6,2 koraka na minuto pod predpostavko, da je gradnja skript na dan 13. 11. 2017 izvzeta iz izračuna, ker gre za izjemo, kjer se je večino skript kopiralo, in ne predstavlja realnega časa grajenja skript.

Slika 9: Krivulja učenja grajenja skript v opazovanem obdobju

Vir: Lastno delo.

3.2 Rezultati pregleda literature

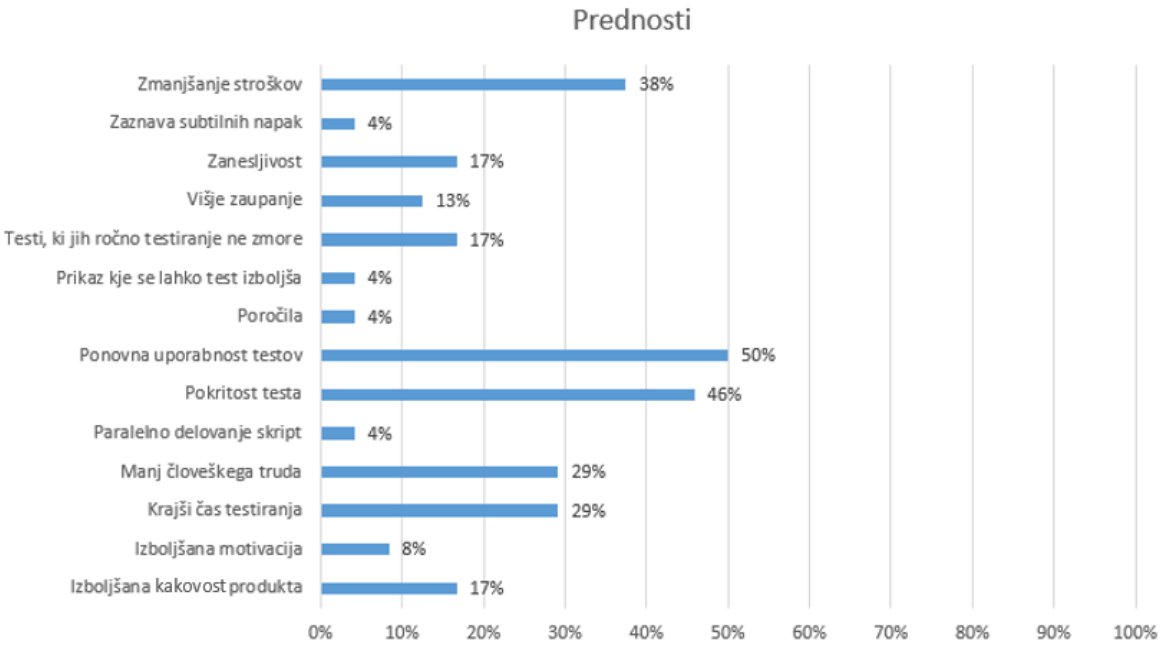
Opravljen je bil pregled literature, kot navedeno v uvodu in metodologiji dela. Pregled se začne z osnovami razvoja in testiranja aplikacij ter prehaja do podrobnosti o prednostih in slabostih avtomatiziranega testiranja ter načinu opravljanja meritev.

Na podlagi pregleda literature so v nadaljevanju predstavljeni dejavniki prednosti in slabosti avtomatizacije testiranja GUI. Izbrani so dejavniki, ki so bili najbolj primerni za naš izbrani primer. Število virov v primerjavi z vsakim izbranim dejavnikom je bilo izračunano s pregledom, kolikokrat je bil določen dejavnik omenjen v literaturi. Vključeni so bili tudi

dejavniki, ki so bili omenjeni splošno za avtomatizacijo testiranj, vendar so lahko upoštevani tudi pri avtomatizaciji testiranj GUI.

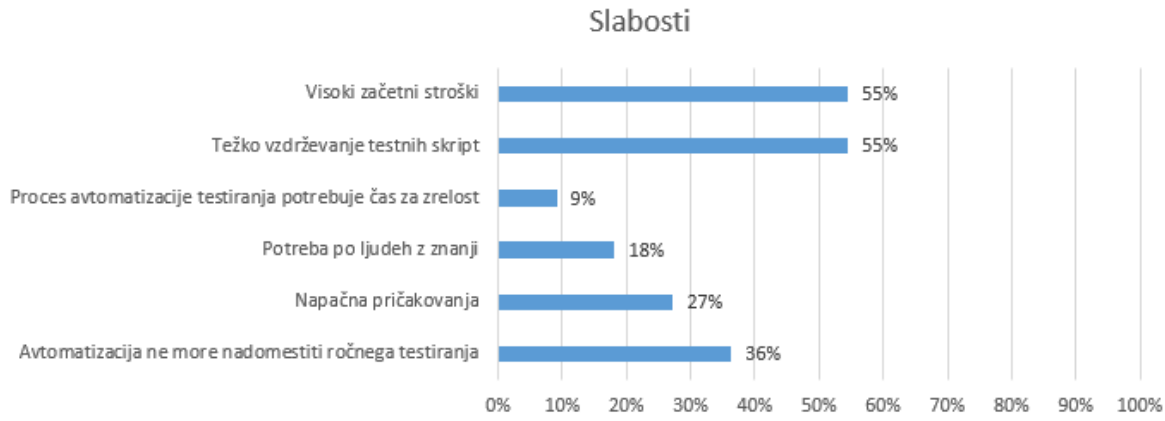
Pregled dejavnikov prednosti, ki predstavljajo njihove frekvence omemb v pregledani literaturi, je prikazan na sliki 10, pregled dejavnikov slabosti pa na sliki 11.

Slika 10: Frekvence omemb dejavnikov prednosti v literaturi



Vir: Lastno delo.

Slika 11: Frekvence omemb dejavnikov slabosti v literaturi



Vir: Lastno delo.

S pregledom literature je bilo identificiranih 14 dejavnikov, ki predstavljajo prednosti, in šest dejavnikov, ki predstavljajo slabosti. Zmanjšanje stroškov, ponovna uporabnost testov in pokritost testa so najbolj pogosti dejavniki prednosti glede na frekvenco omemb v literaturi. Visoki začetni stroški in težko vzdrževanje testnih skript pa sta najbolj pogosta

dejavnika slabosti glede na frekvenco omemb v literaturi. V obeh primerih je frekvenca omemb dejavnikov relativno nizka, kot bi se za določene dejavnike pričakovalo, ker se je v večini literature osredotočalo na posamezen dejavnik in so bili ostali redko omenjeni. Pri sintezi podatkov se dejavniki uvrstijo še v kategorije, pridobljene s tematsko analizo.

3.3 Rezultati intervjujev

3.3.1 Rezultati prvega dela

Za pridobivanje podatkov se je opravilo šest intervjujev. Seznam vprašanih je predstavljen v tabeli 5.

Tabela 5: Seznam sodelujočih v intervjujih

Vprašani	Število aplikacij, ki jih testira	Leta izkušenj s testiranjem	Naziv
Vprašani 1	1	1	Poslovni analitik
Vprašani 2	1	4	Poslovni analitik
Vprašani 3	2	6	Poslovni analitik
Vprašani 4	4	5	Sistemski inženir
Vprašani 5	1	2	Poslovni analitik
Vprašani 6	2	7	Sistemski inženir

Vir: Lastno delo.

Za analizo podatkov intervjujev je bila uporabljena šeststopenjska tematska analiza, ki sta jo podala Braun in Clarke (2006). V nadaljevanju so predstavljeni rezultati vsakega koraka.

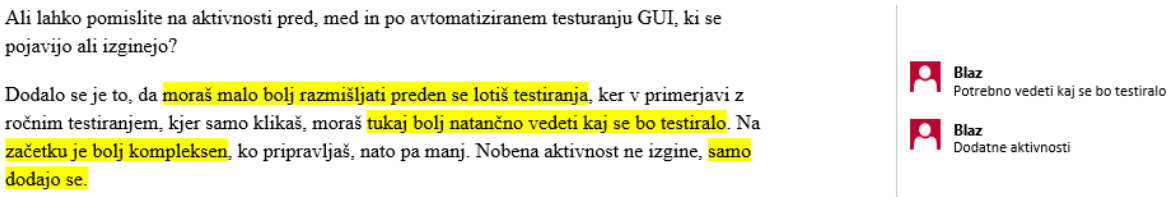
Seznanjanje s podatki

Kljub temu da sam delujem na izbranem področju in sem seznanjen s podatki, sem temeljito večkrat prebral zapiske, pridobljene z intervjuji, in označil pomembne zapise za dodatno seznanjanje z vsebino. Tako sem lažje večkrat pregledal zapise in videl stvari z drugega vidika.

Generiranje inicialnih kod

Prvotni rezultati, povezani z identifikacijo dejavnikov prednosti in slabosti avtomatizacije testiranj GUI, so temeljili na drugem koraku tematske analize. V tem koraku se je ustvaril niz prvotnih kod, tako da se je prebralo in označilo pomembne zapise, ki izražajo prednosti ali slabosti. Primer tega procesa je prikazan na sliki 12, kjer so se označile pomembne informacije, podane s strani enega izmed vprašanih uporabnikov, in so se pripisale kode, ko je bilo potrebno.

Slika 12: Primer generiranja inicialnih kod



Vir: Lastno delo.

Nekateri primeri kod so *dodatne aktivnosti*, *kakovostnejši testi*, *priprava na test* in *naprednejši testi potrebujejo znanje*. V celoti se je pridobilo 23 prvotnih kod in so prikazane v tabeli 6.

Tabela 6: Inicialne kode

Številka kode	Ime kode	Številka kode	Ime kode
1	Priprava na test	13	Paralelno testiranje
2	Manj nadležno testiranje	14	Instalacija je kompleksna
3	Implementacija porabi veliko časa	15	Skripto lahko vedno poganjaš
4	Ni toliko napak v končnem produktu	16	Naloge se uvrsti v sprint nalogo
5	Natančno vedeti, kaj se bo testiralo	17	Produkcija poroča manj napak
6	Dodatne aktivnosti	18	Daljši cikel razvoja
7	Stabilnejša aplikacija	19	Zahteva dobro vzpostavljen proces testiranja
8	Dobro za ponavljajoče testiranje	20	Naprednejši testi potrebujejo znanje
9	Samodejno shranjevanje opravljenih testov	21	Zapisano, kje vse so bile napake
10	Razbije monotonost ponavljajočih se testov	22	Več opravljenih testov
11	Hitrejše testiranje	23	Uporabe se je treba navaditi
12	Ročno testiranje še prisotno		

Vir: Lastno delo.

Iskanje tem

V drugem koraku se je organiziralo kode v kategorije, da se je nizu kod z različnimi atributi podalo skupen pomen. Temeljilo je na lastnem razumevanju obravnavane študije. Najprej se je razdelilo na kode, ki predstavljajo prednosti, in kode, ki predstavljajo slabosti. Nato so se razdelile v kategorije, ki po lastni presoji najbolje razložijo kode, in sicer:

- **povezano s testno aktivnostjo**: kategorija se nanaša na karakteristike testnih primerov in relevantnih aktivnosti. Primer: kodi »hitrejše testiranje« in »opravljanje

več testov hkrati« sta testni aktivnosti, ki predstavljata prednosti in sta uvrščeni v kategorijo »testne aktivnosti«. V to kategorijo je bilo na ta način uvrščenih sedem kod;

- **povezano s testnim orodjem**: kakovost, podpora in izbira primernega orodja za avtomatizirano testiranje GUI igrajo pomembno vlogo pri procesu avtomatizacije. Tako vključuje kode, kot sta »implementacija porabi veliko časa« in »naprednejši testi potrebujejo znanje«. V kategorijo »povezano s testnim orodjem« je bilo uvrščenih šest kod;

- **povezano z aplikacijo**: kategorija pojasni elemente, ki izhajajo iz aplikacije pod testom. Našlo se je tri kode, ki so bile uvrščene v to kategorijo, kot je »stabilnejša aplikacija«;

- **povezano s človeškimi viri in organizacijo**: tudi človeške in organizacijske odločitve vplivajo na proces avtomatizacije. Pod to kategorijo je bilo uvrščenih šest kod, kot sta »naloge se uvrsti v sprint naloge« in »uporabe se je treba navaditi«;

- **križanje**: pri analizi podatkov je bila opažena ena koda, ki se lahko uvrsti v več kot eno kategorijo. Tako se je oblikovala kategorija križanje. »Paralelno testiranje« je primer kode, ki je bila uvrščena v to kategorijo, saj se jo lahko uvrsti v »povezano s testno aktivnostjo« in »povezano s testnim orodjem«.

Tabela 7: Razvrstitev prvotnih kod v kategorije

Prednosti	Povezano s testno aktivnostjo	Manj nadležno testiranje
		Hitrejše testiranje
		Več opravljenih testov
	Povezano s testnim orodjem	Dobro za ponavljajoče se testiranje
		Samodejno shranjevanje opravljenih testov
		Skripto lahko vedno poganjaš
		Zapisano, kje vse so bile napake
	Povezano z aplikacijo	Manj napak v končnem produktu
		Stabilnejša aplikacija
		Zadovoljnejši končni uporabniki
	Povezano s človeškimi viri in organizacijo	Razbije monotonost ponavljajočih se testov
		Naloge se uvrsti v sprint nalogo
	Križanje	Paralelno testiranje
Slabosti	Povezano s testno aktivnostjo	Priprava na test
		Natančno vedeti, kaj se bo testiralo
		Dodatne aktivnosti
		Ročno testiranje še prisotno

se nadaljuje

nadaljevanje

Slabosti	Povezano s testnim orodjem	Implementacija porabi veliko časa
		Instalacija je kompleksna
	Povezano s človeškimi viri in organizacijo	Daljši cikel razvoja
		Zahteva dobro vzpostavljen proces testiranja
		Naprednejši testi potrebujejo znanje
		Uporabe se je treba navaditi

Vir: Lastno delo.

Tabela 7 prikazuje rezultate tretjega koraka v procesu. Tako je bilo 23 inicialnih kod uvrščenih v pet kategorij, razdeljenih na prednosti in slabosti. Teh pet kategorij predstavlja naše teme. Vse kategorije so kodirane po barvah za lažje prikazovanje in berljivost.

Pregled in izboljšava tem

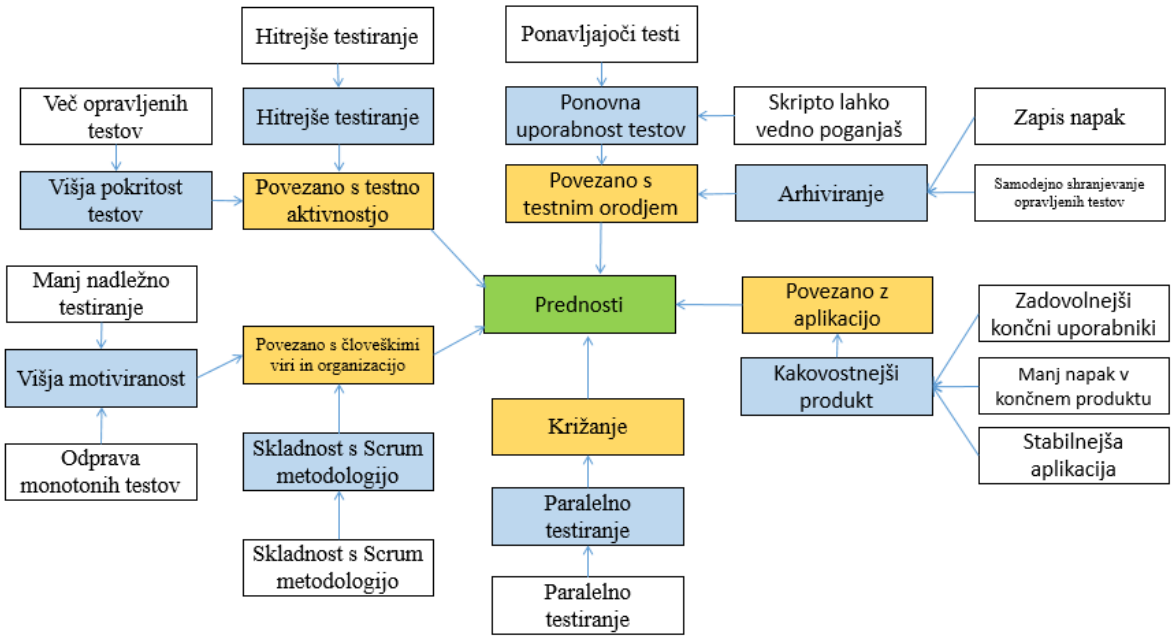
Četrti korak vključuje pregled pridobljenih kod in njihovo izpopolnjevanje. Po pregledu so bile kode predefinirane za lažje razumevanje in enostavnost. Na primer koda »naloge se uvrsti v sprint nalogo« se je predefinirala v »skladnost z metodologijo Scrum« in koda »razbije monotonost ponavljajočih se testov« se je predefinirala v »odprava monotonih testov«. Podobno je bilo predefiniranih še nekaj kod.

Da se je zagotovila konsistentnost procesa analize in v izogib prehitri oziroma nedokončani analizi podatkov, so se podatki ponovno analizirali po določenem času, da se je lažje videlo drugi vidik, s svežim pogledom, na primer: koda »manj nadležno testiranje« je bila prvotno uvrščena pod »povezano s testno aktivnostjo«, po ponovnem pregledu pa se je ocenilo, da jo je bolj smiselno uvrstiti pod »povezano s človeškimi viri in organizacijo«, saj izhaja bolj iz samega uporabnika kot testnega procesa.

Definiranje in poimenovanje tem

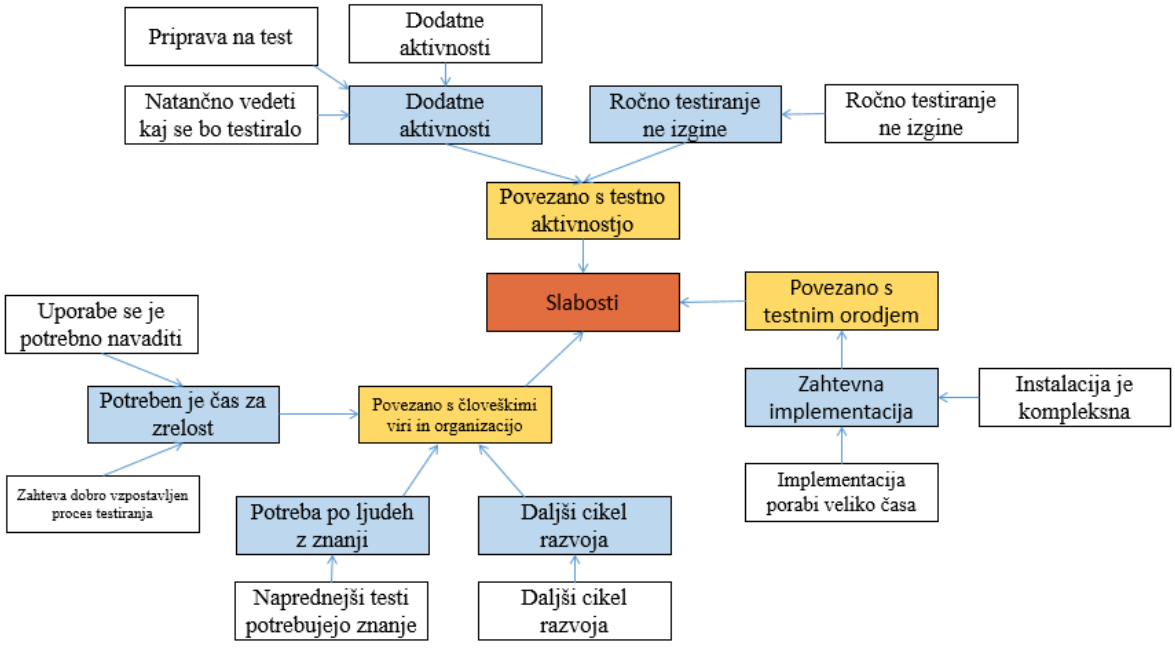
V temu koraku so se razdelile kode v oznake, da pridobijo ideje, ki se razvijajo iz kod, pomen. Z intervjuji in pregledom literature so se pridobile oznake za kode, nat primer: pri primerjanju kode »več opravljenih testov« in podobnih omemb v literaturi se je zaradi ocenjene povezave dodelilo labelo »višja pokritost testov«. Drugi primer je, ko sta se kodi »ponavljajoči se testi« in »skripto lahko vedno poganjaš« združili pod isto oznako »ponovna uporabnost testov«. Na enak način se je tudi ostalim kodam dodelilo oznake. Teme niso potrebovale nobenega dodatnega definiranja in poimenovanja, ker se je ocenilo, da imajo zadosten pomen. Slika 13 predstavlja oznake, ki so se definirale med prednosti, slika 14 pa predstavlja oznake, ki so se definirale med slabosti. Za lažjo preglednost so uporabljene različne barve, kjer so definirane oznake obarvane modro.

Slika 13: Definiranje oznak prednosti



Vir: Lastno delo.

Slika 14: Definiranje oznak slabosti



Vir: Lastno delo.

Izdelava končnih tem

Podporni podatki in končne teme so predstavljeni v točki 4.1.

3.3.2 Rezultati drugega dela

Za validacijo in uporabo ugotovitev, ki temeljijo na raziskavah iz literature, treba le-te integrirati z mnenji in izkušnjami uporabnikov, ki imajo konkretno znanje o specifičnih primerih in pogojih, s katerimi so se srečali. Pri vključevanju uporabnikov v proces učenja, kjer se kombinira dokaze iz literature z njihovim znanjem in izkušnjami, se jim omogoči, da lahko naredijo informirane odločitve, kot da le sledijo predlaganim smernicam avtomatiziranega testiranja GUI iz literature.

Tabela 8: Rezultati mnenj vprašanih

Faktor	**V1**	**V2**	**V3**	**V4**	**V5**	**V6**	**Konsenz**
Izboljšana kakovost produkta	B,5	B,5	B,4	A,5	D,5	A,5	B,5
Pokritost testa	A,3	A,4	A,4	A,3	A,4	A,5	A,4
Krajši čas testiranja	B,5	A,4	B,5	B,3	B,4	B,5	B,4
Zanesljivost	A,4	B,4	A,5	B,2	B,4	B,3	B,4
Višje zaupanje	B,3	B,4	A,4	C,2	C,3	B,3	B,3
Izboljšana motivacija	A,2	C,2	D,3	C,2	C,1	C,2	C,2
Ponovna uporabnost testov	A,5	A,5	A,5	B,4	A,3	A,5	A,5
Manj človeškega truda	A,4	A,5	A,5	A,5	C,3	A,5	A,5
Zmanjšanje stroškov	E,3	C,2	A,5	C,2	C,3	C,2	C,3
Testi, ki jih ročno testiranje ne zmore	C,2	C,1	D,2	C,2	C,1	B,3	C,2
Zaznava subtilnih napak	A,4	B,3	A,5	B,3	B,4	B,4	B,4
Prikaz, kje se lahko test izboljša	A,3	B,4	A,5	B,4	D,2	B,4	B,4
Paralelno delovanje skript	A,2	A,4	D,4	A,5	A,3	A,3	A,3
Poročila	A,4	A,5	A,4	B,2	D,3	A,4	A,4
Avtomatizacija ne more nadomestiti ročnega testiranja	A,2	A,3	B,3	C,2	A,3	A,2	A,2
Visoki začetni stroški	A,2	A,4	A,4	B,4	A,2	A,3	A,3
Težko vzdrževanje testnih skript	D,4	C,4	C,4	C,2	A,4	C,5	C,4
Proces avtomatizacije testiranja potrebuje čas za zrelost	A,3	B,5	A,4	B,4	B,3	A,4	A,4
Napačna pričakovanja	B,3	B,3	B,2	B,3	C,4	D,3	B,3
Potreba po ljudeh z znanji	A,3	B,3	B,3	A,4	B,2	B,3	B,3

A = faktor je veljaven B = faktor je delno veljaven C = faktor je generalno veljaven, vendar ne v tem kontekstu D = faktor ni veljaven E = nesigurnost

Vir: Lastno delo.

Rezultati pregleda literature so se uporabnikom predstavili pred kategoriziranjem s tematsko analizo. Faktorji so bili predstavljeni vprašanim in so podali svoja mnenja. Njihova mnenja so bila nato zapisana v tabelarnem formatu. Tako so vprašani podali informacije, ki se nanašajo na prednosti in slabosti avtomatiziranega testiranja GUI, s katerimi se je pridobilo veljavnost vsakega faktorja in njegovo pomembnost. Vprašani so podali različna mnenja, nato sta bila določena njihovo skupno povprečno mnenje in pomembnost, kar je predstavljeno v tabeli 8. Za dejavnike, ki so se identificirali v tematski analizi in v literaturi

niso bili omenjeni, se je vprašane, ki so jih omenili, vprašalo še za njihovo pomembnost, kar je predstavljeno v tabeli 9. Za lažjo preglednost so z zeleno barvo označeni faktorji, ki se nanašajo na prednosti, in z rdečo faktorji, ki se nanašajo na slabosti. Stolpci prikazujejo rezultate vprašanih in pa ocenjen konsenz njihovih mnenj.

Tabela 9: Rezultati mnenj vprašanih za faktorje, ki jih literatura ne omenja

Faktor	**V1**	**V2**	**V3**	**V4**	**V5**	**V6**	**Konsenz**
Skladnost z metodologijo Scrum	3	4	3	3	3	2	3
Dodatne aktivnosti	4	4	3	4	4	3	4
Zahtevna implementacija	2	2	3	3	2	2	2
Daljši cikel razvoja	/	/	/	4	/	4	4

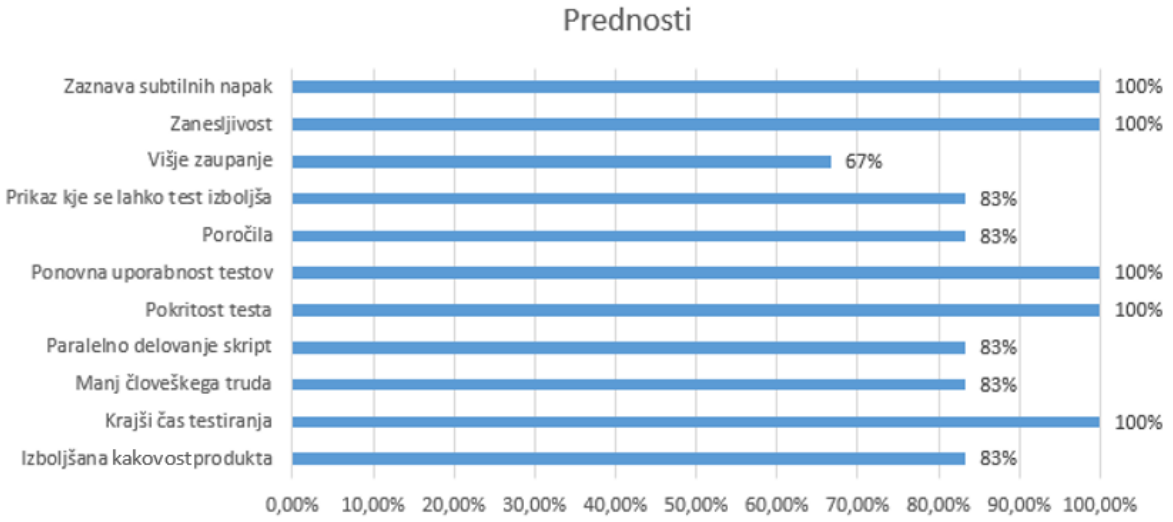
Vir: Lastno delo.

Vrednost A pomeni, da je faktor prisoten v veliki meri, na primer: faktor »pokritost testa« je označen z vrednostjo A, ker je večino vprašanih povedalo, da je bilo znatno povečanje v številu opravljenih testov in tudi povečano število elementov, ki se je v aplikacijah testiralo z uporabo orodja. Faktor »prikaz, kje se lahko test izboljša« je označen z vrednostjo B, ker so bili odgovori vprašanih mešani in je tako faktor delno veljaven. Vrednost C pomeni, da so vprašani mnenja, da faktor drži splošno v avtomatizaciji testiranj GUI, vendar ne v njihovem kontekstu. Dober primer vrednosti C je faktor »testi, ki jih ročno testiranje ne zmore«, kjer so vprašani trdili, da je z orodjem možno avtomatizirati stres teste, vendar orodja niso uporabljali za te namene. Vrednost D pomeni, da faktor ne drži, medtem ko vrednost E pomeni, da vprašani ne more reči, ali faktor drži ali ne, ker v njihovem primeru ni bil prisoten. Posamezniki so na določene faktorje podali mnenja v vrednosti D ali E, vendar noben faktor ni bil v konsenzu označen s tema vrednostma. Tako bi se lahko reklo, da so bili iz pregleda literature dobro izbrani faktorji, je bilo pa nekaj faktorjev, ki v danem kontekstu niso veljavni. Primer vrednosti C, ki je presenetil, je faktor »zmanjšanje stroškov«, kjer so bili vprašani mnenja, da se stroški sicer zmanjšajo, vendar niso relevantni v njihovem kontekstu, ker se bo za testiranje porabilo ravno toliko stroškov, samo da se bo testiralo več oziroma druge zadeve.

Poleg veljavnosti faktorjev je predstavljena še pomembnost faktorjev, kjer so vprašani glede na lastno mnenje ocenili, kako pomemben se jim zdi posamezen faktor. Glede na mnenja vprašanih se je pomembnost označila v numerični obliki od 1 do 5, kjer 1 pomeni ni pomembno in 5 pomeni zelo pomembno, na primer: faktor »manj človeškega truda« je pridobil pomembnost 5, kar pomeni, da je zmanjšanje truda zelo pomemben faktor in velja za eno izmed osnov, ki se pričakuje pri procesu avtomatiziranega testiranja GUI. Faktor »potreba po ljudeh z znanji« je pridobil oceno 3, kar pomeni, da je srednje pomembno, koliko znanja vsebuje tester, saj se s časom nauči in je tudi odvisno od same aplikacije. Kot skoraj nepomemben faktor je z vrednostjo 2 označen faktor »izboljšana motivacija«, ker so bili vprašani mnenja, da četudi faktor ni prisoten, bo testiranje večinoma enako potekalo.

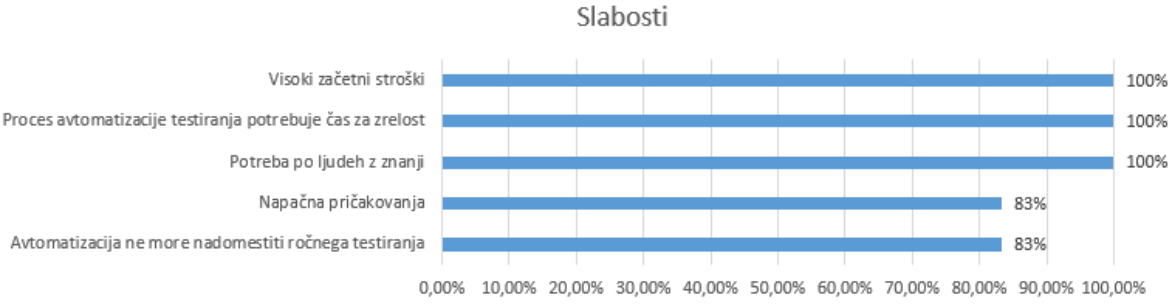
Faktorji, ki so dobili vrednost A ali B, so vključeni pri izračunu stopnje veljavnosti, medtem ko so C, D in E izvzete iz izračuna. Tako se štiri faktorje ni vključilo, ker so bili po konsenzu vprašani mnenja, da faktorji ne spadajo v njihov kontekst. To pomeni, da se vsi faktorji prednosti in slabosti avtomatiziranega testiranja iz literature ne zrcalijo tudi na avtomatizirano testiranje GUI v danem primeru. Stopnja veljavnosti za vsak faktor prednosti in slabosti iz literature je izračunana na podlagi mnenj vprašanih in je prikazana na slikah 15 in 16. Verjetnost je izračunana kot odstotek števila vseh vprašanih, ki so bili mnenja, da je določen faktor veljaven ali delno veljaven.

Slika 15: Posteriorne vrednosti dejavnikov prednosti



Vir: Lastno delo.

Slika 16: Posteriorne vrednosti dejavnikov slabosti



Vir: Lastno delo.

Uporabljena je bila Bayesianova metoda sinteze, kjer se je sintetiziralo rezultate intervjujev in pregleda literature, kar je privedlo do koristne interpretacije rezultatov. V naslednjem poglavju je metoda natančneje predstavljena.

4 ANALIZA IN OCENA

4.1 Definiranje dejavnikov

4.1.1 Sinteza podatkov

Kot omenjeno, je bil uporabljen Bayesianov pristop sinteze raziskovalnih dokazov in uporabe znanja in izkušenj uporabnikov. Iz predhodnih subjektivnih mnenj uporabnikov se je generiralo dejavnike koristi in slabosti, ki so bili kategorizirani v skupne opisne teme (kategorije). Mnenja vsakega uporabnika so bila združena za predhodno verjetnost vsakega dejavnika, ki predstavlja njihovo stopnjo veljavnosti v kontekstu. Predhodna verjetnost je bila izračunana kot odstotek vprašanih uporabnikov, ki so omenili dejavnik kot veljaven v njihovem kontekstu. Primer: dejavnik »arhiviranje« ima predhodno verjetnost v vrednosti 33 %, ker sta dva od šestih vprašanih uporabnikov omenila dejavnik kot veljavno prednost pri avtomatiziranem testiranju GUI. Kot naslednje so bili izbrani relevantni dejavniki iz pregleda literature in bili v sklopu sinteze podatkov kategorizirani v enake teme, kot so bile pridobljene z intervjuji, nekaj pa jih je bilo tudi dodanih. Verjetnost vsakega dejavnika je bila izračunana kot odstotek števila študij, ki so ga omenile kot veljavnega. Verjetnost dejavnika »manj človeškega truda« je 29-odstotna, ker je sedem od 24 študij poročalo o omenjenem dejavniku kot veljavnem. Za konec je bila izračunana še posteriorna verjetnost z združenjem mnenj uporabnikov in dejavnikov iz literature. Delež uporabnikov, ki so bili mnenja, da je dejavnik, poročan v literaturi, veljaven, predstavlja posteriorno verjetnost. Dejavnik »visoki začetni stroški« s posteriorno verjetnostjo 100 % pomeni, da je bilo vseh šest vprašanih uporabnikov mnenja, da je omenjeni dejavnik iz literature veljaven. Predhodna verjetnost, verjetnost in posteriorna verjetnost za vsak dejavnik so predstavljene v tabeli 10.

Tabela 10: Rezultati sinteze podatkov

Kategorija	Oznaka	Predhodna verjetnost	Verjetnost	Posteriorna verjetnost
Prednosti				
Povezano s testno aktivnostjo	Hitrejše testiranje	100 %	29 %	100 %
	Višja pokritost testov	33 %	46 %	100 %
Povezano s testno aktivnostjo	Zanesljivost	x	17 %	100 %
	Prikaz, kje se lahko test izboljša	x	4 %	83 %
	Testi, ki jih ročno testiranje ne zmore	x	17 %	17 %
	Zaznava subtilnih napak	x	4 %	100 %
Povezano s testnim orodjem	Ponovna uporabnost testov	50 %	50 %	100 %
	Arhiviranje	33 %	4 %	83 %

se nadaljuje

nadaljevanje

Kategorija	Oznaka	Predhodna verjetnost	Verjetnost	Posteriorna verjetnost
Prednosti				
Povezano z aplikacijo	Kakovostnejši produkt	83 %	17 %	83 %
Povezano z aplikacijo	Višje zaupanje	x	13 %	67 %
Povezano s človeškimi viri in organizacijo	Višja motiviranost	17 %	8 %	17 %
Povezano s človeškimi viri in organizacijo	Manj človeškega truda	x	29 %	83 %
Povezano s človeškimi viri in organizacijo	Skladnost z metodologijo Scrum	100 %	x	100 %
Križanje	Paralelno testiranje	17 %	4 %	83 %
Križanje	Zmanjšanje stroškov	x	38 %	17 %
Slabosti				
Povezano s testno aktivnostjo	Ročno testiranje ne izgine	83 %	36 %	83 %
Povezano s testno aktivnostjo	Dodatne aktivnosti	100 %	x	100 %
Povezano s testnim orodjem	Zahtevna implementacija	100 %	x	100 %
Povezano s človeškimi viri in organizacijo	Potreben čas za zrelost	50 %	9 %	100 %
	Potreba po ljudeh z znanji	100 %	18 %	100 %
Povezano s človeškimi viri in organizacijo	Težko vzdrževanje testnih skript	x	55 %	17 %
	Napačna pričakovanja	x	27 %	83 %
Povezano s človeškimi viri in organizacijo	Daljši cikel razvoja	33 %	x	33 %
Križanje	Visoki začetni stroški	x	55 %	100 %

: Dejavniki, omenjeni pri pregledu literature in pri intervjuju uporabnikov

: Dejavniki, omenjeni pri pregledu literature

: Dejavniki, omenjeni pri intervjuju uporabnikov

Vir: Lastno delo.

Iz tabele 10 je razvidno, da je pri večini dejavnikov posteriorna verjetnost zelo visoka. To pomeni, da se vprašani uporabniki s predstavljenimi dejavniki iz pregleda literature strinjajo, da so prisotni in relevantni v njihovem kontekstu. Najdeni so bili štirje dejavniki z nizko

posteriorno verjetnostjo, kar pomeni, da jih vprašani niso prepoznali za prisotne oziroma veljavne v njihovem kontekstu. Nekaj novih dejavnikov, ki jih vprašani prvotno niso sami omenili, pa je bilo dodanih, kot je »zaznava subtilnih napak«, ki je v literaturi sicer redko poročana, vendar so se vsi vprašani strinjali, da je dejavnik prisoten in relevanten v njihovem kontekstu. Nekateri dejavniki so bili omenjeni s strani vprašanih in prav tako omenjeni v literaturi, literatura je tako le potrdila njihova mnenja. S strani vprašanih so bili omenjeni štirje dejavniki, o katerih literatura ni poročala, kar je lahko posledica specifičnosti dejavnikov ter načina in obsega izbire literature.

4.1.2 Analiza podatkov

Cilj te točke je analizirati rezultate prejšnjih točk in definirati iskane dejavnike. Predstavlja opise dejavnikov prednosti in slabosti avtomatiziranega testiranja GUI.

Prednosti

Med sintezo podatkov se je identificiralo 15 dejavnikov, ki predstavljajo prednosti avtomatiziranega testiranja GUI. Od 15 dejavnikov je bilo sedem pridobljenih tako iz mnenj uporabnikov kot iz literature, en bil identificiran izključno s strani uporabnikov in sedem izključno iz literature. Med dejavniki so bili trije označeni s strani uporabnikov za neveljavne v njihovem kontekstu. Največ dejavnikov izhaja iz kategorije »povezano s testno aktivnostjo«, ostali dejavniki so enakomerno porazdeljeni med ostale kategorije. Identificirani dejavniki prednosti pokrijejo štiri od petih dejavnikov, naštetih v točki 2.3, katerih prisotnost in veljavnost sta se pričakovali pred implementacijo orodja. Dejavnik »boljša komunikacija z razvijalci« ni bil omenjen v opravljeni raziskavi in je tako označen za neveljavnega.

Ugotovljeni relevantni dejavniki:

- **hitrejše testiranje:** v primerjavi z ročnim testiranjem so avtomatizirani testi GUI opravljeni hitreje. Kot pravi eden izmed vprašanih: »*Ko je skripta enkrat postavljena, so testi očitno hitrejši.*« To so potrdile tudi opravljene meritve orodja v točki 3.1;
- **višja pokritost testov:** eden izmed vprašanih je trdil: »*Lahko se naredi dober test z veliko naključnimi vnosi in preverjanja le-tega.«* Obseg elementov, ki se jih lahko v določenem času preveri, se z avtomatizacijo znatno poveča, saj kot že v prejšnjem dejavniku omenjeno, se zaradi večje hitrosti testiranja hkrati lahko preverja tudi pravilnost elementov in ne samo funkcionalnosti;
- **zanesljivost:** s potrebo po večkratnem opravljanju istih testov se z ročnim testiranjem težko vsakič identično ponovi teste. To težavo avtomatizirano testiranje GUI odpravi, kar je eden od vprašanih potrdil s sledečim mnenjem: »*Strinjam se, da je zanesljivost večja, saj to opazim pri testih, ki jih še nisem avtomatiziral in sem jim moral večkrat opraviti. Pri teh testih se je vedno pojavil dvom, če sem vse poklikal isto kot v preteklih primerih.*«;

- **prikaz, kje se lahko test izboljša:** z beleženjem korakov in časa, ki jih porabijo, se lahko testi optimizirajo. Eden vprašani temu še doda: »*Ko že večkrat poženeš en test, lahko vidiš, v katerih korakih so napake najbolj pogoste, in preurediš test tako, da najprej testira tiste korake.*«;
- **zaznava subtilnih napak:** napake se lahko pojavijo samo pod določenimi pogoji in se pri nizki količini testov ne opazijo. Ker avtomatizirano testiranje GUI omogoča veliko število testov v kratkem času, se lahko zazna tudi te napake. Kot je povedal eden vprašani: »*Pri enem testu sem večkrat pognal skripto in je naključno generirala črke za vnos v eno polje in je enkrat vstavila šumnik, kar je pokazalo napako, da šumniki ne delujejo pravilno. Sam tega verjetno ne bi zaznal, ker vedno napišem besede, kot je test ali kaj podobnega.«;*
- **ponovna uporabnost testov:** ko so skripte testov enkrat zgrajene, se lahko iste teste vedno ponavlja;
- **arhiviranje:** vsi opravljeni testi se z orodjem za avtomatizacijo testiranj GUI zapisujejo v dnevnik, kar uporabnikom lahko koristi za boljšo preglednost. »*S tem, ko se vsi testi zapišejo, grem lahko za nazaj pogledat, katere teste sem že naredil in katere še moram.*«;
- **kakovostnejši produkt:** poroča se manj napak iz produkcije, kot trdi tudi vprašani: »*Aplikacija ima na koncu manj napak in je bolj stabilna.*«;
- **višje zaupanje:** pomeni višje zaupanje, da aplikacija deluje pravilno, ker opravljeni testi niso poročali napak;
- **manj človeškega truda:** dejavnik se nanaša na izvajanje testov in ne upošteva grajenja in vzdrževanja skript, ker so bili vprašani mnenja, da lahko slednja povečata človeški trud, vendar ne v njihovem kontekstu. Ob tej predpostavki se porabi manj človeškega truda, ker pri avtomatiziranem testiranju GUI ni treba razmišljati, kaj vse je treba preklikati v testih, in nato teste še ročno narediti, treba je le pognati skripto in počakati na konec;
- **skladnost z metodologijo Scrum:** kot pravi vprašani: »*Grajenje skript, ki so že vnaprej poznane, se uvrsti v sprint nalogo in tako omogoči lažji pregled, da se ne pozabi na obveznost in zato nameni čas. Rekel bi, da je dobro v primerih, ko se odlaša z grajenjem skript in te tako prisili, da jo zgradiš.*«;
- **paralelno testiranje:** možnost vzporednega izvajanja več testov.

Slabosti

Med sintezo podatkov se je identificiralo devet dejavnikov, ki predstavljajo slabosti avtomatiziranega testiranja GUI. Od devetih dejavnikov so bili trije pridobljeni tako iz mnenj uporabnikov kot iz literature, trije so bili identificirani izključno s strani uporabnikov in trije izključno iz literature. Med dejavniki je bil eden označen s strani uporabnikov za neveljavne v njihovem kontekstu. Največ dejavnikov izhaja iz kategorije »povezano s človeškimi viri in organizacijo«, ostali dejavniki so enakomerno porazdeljeni med ostale kategorije. V kategorijo »povezano z aplikacijo« ni bil uvrščen noben dejavnik.

Ugotovljeni relevantni dejavniki:

- **ročno testiranje ne izgine:** zaradi ogromnega števila možnih vhodov in scenarijev, ki se jih težko predvidi in s tem težko avtomatizira, je pri testiranju GUI velikokrat primernejše opraviti teste ročno;
- **dodatne aktivnosti:** z implementacijo avtomatiziranega testiranja GUI se pojavijo dodatne aktivnosti v procesu testiranja, kot sta priprava in vzdrževanje testnih skript;
- **zahtevna implementacija:** namestitev orodja, kar lahko povzroča tehnične težave. Kar je manj odvisno od specifičnega orodja, je sprememba procesa testiranja;
- **potreben čas za zrelost:** večina vprašanih je trdila, da lahko koristiš več prednosti, ko se naučiš uporabe orodja in ko dobro vpelješ avtomatizirano testiranje GUI v svoj proces testiranja. Vprašana uporabnica je dejala: »*Potrebno je imeti dobro vzpostavljen proces avtomatiziranega testiranja GUI, v tem primeru bi rekla, da ti lahko zelo pomaga, če ga pa nimaš, pa težko dobiš koristi.*« Krivulja učenja gradnje testnih skript, predstavljena v točki 3.1, nam je tudi pokazala, da s časom uporabniki vse hitreje gradijo testne skripte in s tem znižajo porabljen čas za vzdrževanje in posledično stroške, hkrati pa so testne skripte tudi kakovostneje zgrajene in pozitivno vplivajo na druge dejavnike prednosti;
- **potreba po ljudeh z znanji:** eden izmed vprašanih je dejal: »*Moraš malo bolj razmišljati, preden se lotiš testiranja, ker v primerjavi z ročnim testiranjem, kjer samo klikaš, moraš tukaj bolj natančno vedeti, kaj se bo testiralo.*« Slednje pomeni, da mora imeti tester znanje, kako opraviti dobre teste. Vprašani so še trdili, da je za naprednejše teste potrebno tehnično znanje;
- **napačna pričakovanja:** napačno pričakovanje uporabnikov, da z avtomatizacijo testiranj izgine potreba po razmišljanju in da bo orodje dovolj pametno, da bo večino dela opravilo samo. Še vedno je prisotna analiza, kaj se bo testiralo, in priprava orodja, da pravilno testira aplikacijo;
- **daljši cikel razvoja:** sistemska inženirja sta izpostavila, da se z večjo pripravo na testiranje cikel razvoja podaljša;
- **visoki začetni stroški:** potrebna je finančna investicija v nakup orodja ali licenc, čas, ki se ga porabi za pripravo orodja in skript, ter čas, ki se porabi za izobraževanje uporabnikov.

4.2 Vrednotenje dejavnikov

V nadaljevanju je dejavnikom, definiranim v točki 4.1, dodeljena vrednost glede na njihovo ocenjeno pomembnost, ki je bila pridobljena z intervjuji in prikazana v točki 3.3.2. Dejavniki, ki so se upoštevali pri oceni upravičenosti, morajo imeti vrednost konsenza pomembnosti 4 ali 5. Za lažjo preglednost so v tabeli 11 z zeleno barvo označeni faktorji, ki se nanašajo na prednosti, in z rdečo faktorji, ki se nanašajo na slabosti.

Tabela 11: Končne vrednosti dejavnikov

Dejavnik	Pomembnost	Veljavnost	Vrednost
Kakovostnejši produkt	5	83 %	1,66
Višja pokritost testov	4	100 %	1
Hitrejše testiranje	4	100 %	1
Zanesljivost	4	100 %	1
Ponovna uporabnost testov	5	100 %	2
Manj človeškega truda	5	83 %	1,66
Zaznava subtilnih napak	4	100 %	1
Prikaz, kje se lahko test izboljša	4	83 %	0,83
Arhiviranje	4	83 %	0,83
Proces avtomatizacije testiranja potrebuje čas za zrelost	4	100 %	1
Daljši cikel razvoja	4	33 %	0,33
Dodatne aktivnosti	4	100 %	1

Vir: Lastno delo.

Vrednost v tabeli 11 je izračunana po sledeči enačbi: pomembnost 5 šteje za dve točki, pomembnost 4 šteje za eno točko. Točka iz pomembnosti se nato pomnoži s procentualno vrednostjo veljavnosti. Primer: dejavnik »arhiviranje« ima pomembnost 4, kar prinese eno točko, pomnoženo s 83 %, izračun prinese 0,83 točke vrednosti.

4.3 Primerjava stanj pred in po uvedbi orodja

V tej točki se primerja čas, ki se je prihranil z avtomatiziranim testiranjem GUI na podlagi opravljenih meritev v točki 3.1. Pomnožilo se je število ponovitev s časom trajanja ročnih testov in nato še s časom trajanja skript. Razlika med rezultatoma predstavlja prihranjen čas. Nato se je seštel čas, porabljen za vzdrževanje, in čas, ki predstavlja stroške nakupa licenc, stroške izobraževanj in stroške implementacije. Seštevek se je nato odštel izračunanemu prihranjenemu času. Rezultat prikazuje, koliko stroška investicije se je povrnilo s prihranjenim časom.

Uporaba orodja za avtomatizacijo testiranj GUI je v prvih šestih mesecih prinesla sledeče rezultate: v primerjavi z opravljanjem enakih testov ročno se je v povprečju z orodjem prihranilo eno minuto in 42 sekund na izveden testni primer, kjer je bilo povprečno opravljenih 51 korakov. V tem obdobju je to naneslo na 42 ur in 48 minut prihranjenega časa za 1516 opravljenih testov. V obdobju drugih šest mesecev se je prihranjen čas na izveden testni primer znižal na eno minuto in 18 sekund, vendar se je povprečno število korakov zvišalo na 58. V drugem obdobju se je prihranilo 21 ur za 945 opravljenih testov. Skupno je bilo prihranjenih 63 ur in 48 minut. Skupnemu prihranjenemu času je treba odšteti čas 35 ur in 20 minut, ki je bil porabljen za vzdrževanje skript, kot je bilo omenjeno že v točki 3.1, kar nanese na 27 ur in 28 minut čistega prihranjenega časa. Začetni stroški, izraženi v človeških urah dela, so se delili na šest delov, saj so bile meritve opravljene samo za enega uporabnika in znašajo 138 ur dela. Za končni izračun se je prihranjenemu času odštelo čas, izražen kot

strošek začetnega nakupa, izobraževanj in implementacije orodja, kar pomeni, da je po obdobju enega leta v tem primeru uvedba orodja še vedno kazala na strošek v višini 110 ur in 32 minut človeškega dela.

Sledi primerjava dejavnikov, kjer se sešteje vrednosti dejavnikov prednosti in odšteje dejavnike slabosti. Razlika prikaže, ali je avtomatizacija testiranj pozitivno ali negativno vplivala na proces testiranja in kar je z njim povezano in v kakšni meri.

To je narejeno z enostavno enačbo, in sicer seštevku vrednosti dejavnikov prednosti v tabeli 11 se je odštelo seštevek vrednosti dejavnikov slabosti, kar je naneslo na pozitivno vrednost 8,65.

4.4 Ocena upravičenosti uvedbe orodja

V točki 4.3 se je izračunalo oprijemljive vrednosti, vendar je za oceno upravičenosti uvedbe orodja treba upoštevati še neoprijemljive koristi in slabosti. S sledenjem misli Bannisterja (2004, str. 81), da je neoprijemljive koristi težko izračunati, se je vprašalo tiste, ki koristi dobijo, in to vzelo za merilo, ali je bila uvedba upravičena. Da so se neoprijemljivi dejavniki lahko količinsko izrazili, se je ocena upravičenosti določila po sledečem izračunu: namen vpeljave orodja za avtomatizacijo testiranj GUI je bil doseganje pričakovanih dejavnikov, predstavljenih v točki 2.3. Interno je bilo dogovorjeno, da se smatra uvedba orodja kot upravičena, če se bo dosegalo vsaj tri naštete dejavnike oziroma dejavnike, ki se bodo izkazali kot njihov ekvivalent. Iz tega izhaja, da celoten strošek investicije predstavlja negativnih šest točk vrednosti, vsak posamezen dejavnik pa dve pozitivni točki vrednosti. Če bo izračun teh vrednosti nič ali pozitiven, pomeni, da je bila uvedba orodja upravičena.

Za natančnejšo prepoznavo dejavnikov in njihovih vrednosti se je izvedlo tematsko analizo, kjer se je prepoznalo večje število dejavnikov, tako pozitivnih kot negativnih, nekateri so bili tudi enaki oziroma podobni prvotnim pričakovanim dejavnikom. Njihova vrednost je bila natančneje določena v primerjavi s prvotnimi dejavniki, ki so bili določeni le glede na občutek določenih zaposlenih, ki so uvedbo orodja predlagali. Tako se je za izračun uporabilo vrednost 8,65, pridobljeno v točki 4.3, in ji odštelo vrednost stroškov. Vrednost stroškov se je še procentualno zmanjšalo za izračunan prihranjen čas, kar pomeni na 80 % vrednosti. Končni izračun je pomenil, da se je vrednosti 8,65 odštelo 4,8 vrednosti stroškov, kar je naneslo na 3,85 točke vrednosti. Z visoko pozitivno vrednostjo je ocenjeno, da je bila uvedba orodja za avtomatizacijo testiranj GUI upravičena.

SKLEP

V magistrskem delu je bil narejen pregled literature, ki služi k razumevanju tematike in poda predloge za ocenjevanje upravičenosti uvedbe avtomatiziranega testiranja GUI. Pregled literature je bil uporabljen tudi za opravljeno tematsko analizo, ki je skupaj z meritvami uporabe implementiranega orodja služila za izračun končne upravičenosti uvedbe orodja za avtomatizirano testiranje GUI.

Z vidika izračuna je bila uvedba orodja zaradi pozitivne končne vrednosti upravičena. Definirani dejavniki so pokrili vse prvotne pričakovane dejavnike, razen dejavnika »boljša komunikacija z razvijalci«, ki v analizi ni bil zaznan kot veljaven. Kljub visokim začetnim stroškom je doseganje prednosti tisto, kar v danem primeru upraviči uvedbo orodja.

Z analizo se je ugotovilo, da proces avtomatiziranega testiranja GUI potrebuje čas za zrelost. S krivuljo učenja je bilo dokazano, da se stroški grajenja testnih skript s časom manjšajo. Prav tako se zaradi večanja znanja manjša čas, porabljen za vzdrževanje skript. Tako se lahko pričakuje, da se s časom stroški v celoti upravičijo v obliki prihranjenega časa in ne na račun slabše kakovosti produkta. Iz meritev je bilo tudi vidno, da daljši kot so posamezni testi, hitreje se stroški upravičijo.

Za natančnejše informacije o zmanjševanju stroškov bi bilo treba opravljati meritve dlje časa in na večjem obsegu, poleg tega bi bilo treba upoštevati še meritve iz naslova manj poročanih napak iz produkcijskega okolja in prihranjenega časa popravil ter prihranke iz naslova kakovostnejšega produkta in dobrega imena produkta. To so meritve, ki se v danem primeru niso natančneje upoštevale. Izmerjeno je bilo, kateri dejavniki so prisotni in pomembni. Poleg dejavnikov prednosti pa so se definirali tudi dejavniki slabosti, ki se jih pred uvedbo orodja ni pričakovalo.

Za zaključek, avtomatizacija testiranj GUI se dobro uvrsti v proces razvoja aplikacij, aktivnosti testiranj in njegovo organizacijo, kar je dokazano z dejavnikom »skladnost z metodologijo Scrum«. Pokrijejo se pomanjkljivosti ročnega testiranja, hkrati pa ročnemu testiranju ni v napoto. Uvedba orodja za avtomatizacijo testiranj GUI je priporočljiva, saj se pozitivno dotakne dejavnikov, kot so kakovost, čas in stroški. Kako hitro se pridobi koristi, je pa tako kot pri večini uvedb informacijske tehnologije odvisno od zavzetosti vseh udeležencev.

LITERATURA IN VIRI

1. Aebersold, K. (2018). *Why and When to Conduct GUI Testing*. Pridobljeno 22. junij 2018 iz https://smartbear.com/learn/automated-testing/why-gui-testing.
2. Allott, S. K. (1999). Automate your tests - you won't regress it! *Proceedings of Pacific N.W. Software Quality Conference* (str. 132–154). Oregon: Seventeenth Annual Pacific Northwest Software Quality Conference.
3. Almeida, E. R., Abreu, B. T. & Moraes, R. (2009). An Alternative Approach to Test Effort Estimation Based on Use Cases. *2nd International Conference on Software Testing Verification and Validation (ICST)*(str. 279–288). Denver: IEEE.
4. Alshraideh, M. (2008). A complete automation of unit testing for javascript programs. *Journal of Computer Science, 4*(12), 1012–1019.
5. Anandarajan, A. H. & Wen, J. (1999). Evaluation of information technology investment. *Management Decision, 37*(4), 329–339.

6. Antine, J., Eph, S. & Stray, A. (1998). Financial appraisal and the IS / IT investment decision-making process. *Journal of Information Technology, 13*(1), 3–14.
7. Aranha, E. & Borba, P. (2007). Test effort estimation models based on test specifications. *Testing: Academic and Industrial Conference - Practice and Research Techniques* (str. 67–71). Windsor: IEEE.
8. Badampudi, D. & Wohlin, C. (2016). Bayesian synthesis for knowledge translation in software engineering: Method and illustration. *Software Engineering and Advanced Applications (SEAA)* (str. 148–156). Limassol: IEEE.
9. Bannister, F. (2004). *Purchasing and Financial Management of Information Technology*. Oxford: Elsevier.
10. Bannister, F. & Remenyi, D. (2000). Act of faith: instinct, value and IT investment decisions. *Journal of Information Technology, 15*(3), 231–241.
11. Bertolino, A. (2007). Software testing research: Achievements, challenges, dreams. *Future of Software Engineering* (str. 85–103). Minneaposlis: IEEE.
12. Bashir, M. F. & Banuri, S. H. (2008). Automated model based software test data generation system. *Proceedings of the 4th International Conference on Emerging Technologies* (str. 275–279). Rawalpindi: IEEE.
13. Berner, S., Weber, R. & Keller, R. K. (2005). Observations and lessons learned from automated testing. *Proceedings of the 27th International Conference on Software Engineering* (str. 571–579). Saint Louis: IEEE.
14. Binder, R. V. (2011). *What's my Testing ROI?*. Pridobljeno 2. junija 2018 iz http://robertvbinder.com/wp-content/uploads/sites/4/2011/06/Whats-My-Testing-ROI1.pdf.
15. Bird, C. M. (2005). How i stopped dreading and learned to love transcription. *Qualitative inquiry, 11*(2), 226–248.
16. Boehmer, B. & Patterson, B. (2001). Software test automation: Planning and infrastructure for success. *Proceedings of the STAR East 2001 Conference*. Orange Park: Software Quality Engineering (SQE).
17. Boyatzis, R. E. (1998). *Transforming qualitative information: Thematic analysis and code development*. Thousand Oaks: Sage Publications.
18. Braun, V. & Clarke, V. (2006). Using thematic analysis in psychology. *Qualitative research in psychology, 3*(2), 77–101.
19. Burnim, J. & Sen, K. (2008). Heuristics for scalable dynamic test generation. *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering* (str. 443–446). L'Aquila: IEEE.
20. Burnstein, I. (2003). *Practical software testing*. Springer - New York: Verlag New York, Inc.
21. Coelho, R., Cirilo, E., Kulesza, U., von Staa, A., Rashid, A. & de Lucena, C. P. (2007). Jat: A test automation framework for multi-agent systems. *Proceedings of the 23rd IEEE International Conference on Software Maintenance* (str. 425–434). Paris: IEEE.
22. Cui, M. & Wang, C. (2015). Cost-Benefit Evaluation Model for Automated Testing Based on Test Case Prioritization. *Journal of Software Engineering, 9*(4), 808–817.

23. Dallal, J. A. (2009). Automation of object-oriented framework application testing. *Proceedings of the 5th IEEE GCC Conference and Exhibition* (str. 1–5). Kuwait City: IEEE.
24. Despa, M. L. (2004). Comparative study on software development methodologies. *Database Systems Journal, 5*(3). Pridobljeno 10. junija 2018 iz http://dbjournal.ro/archive/17/17_4.pdf.
25. du Bousquet, L. & Zuanon, N. (1999). An overview of lutess: A specification-based tool for testing synchronous software. *Proceedings of the 14th Conference on Automated Software Engineering* (str. 208–215). Cocoa Beach: IEEE.
26. Duran, J. W. & Ntafos, S. C. (1984). An evaluation of random testing. *IEEE Transactions on Software Engineering, 10*(7), 438–444.
27. Dyba, T., Kitchenham, B. A. & Jogensen, M. (2005). Evidence-based software engineering for practitioners. *IEEE software, 22*(1), 58–65.
28. Ellims, M., Bridges, J. & Ince, D. C. (2006). The economics of unit testing. *Empirical Software Engineering, 11*(1), 5–31.
29. Ericson, T., Subotic, A. & Ursing, S. (1997). TIM - A test improvement model. *Software Testing Verification and Reliability, 7*(4), 229–246.
30. Fecko, M. A. & Lott, C. M. (2002). Lessons learned from automating tests for an operations support system. *Software: Practice and Experience, 32*(15), 1485–1506.
31. Fewster, M. & Graham, D. (1999). *Software test automation: effective use of test execution tools*. Harlow: Addison-Wesley.
32. Fitzgerald, B. & Stol, K.-J. (2014). Continuous software engineering and beyond: trends and challenges. *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering* (str. 1–9). Hyderabad: ACM.
33. Garousi, V. & Mäntylä, M. V. (2016). When and what to automate in software testing? A multi-vocal literature review. *Information and Software Technology, 76*, 92–117.
34. Gomez, R. & Pather, S. (2012). ICT Evaluation: Are We Asking the Right Questions? *The Electronic Journal of Information Systems in Developing Countries, 50*(5), 1–14.
35. Goodhue, L. D. Wybo, D. M. & Kirsch, J. L. (1992). The Impact of Data Integration on the Costs and Benefits of Information Systems. *MIS Quarterly, 16*(3), 293–311.
36. Goodwin, P. & Wright, G. (1998). *Decision Analysis for Management Judgment*. Chichester: Wiley.
37. Gulechha, L. (2013). *Software Testing Metrics*. Pridobljeno 20. maja 2018 iz https://www.agileconnection.com/sites/default/files/article/file/2013/XDD14789filelistfilename1_0.doc.
38. Gunasekaran, A., Love, P. E., Rahimi, F. & Miele, R. (2001). A model for investment justification in information technology projects. *International Journal of Information Management, 21*(5), 349–364.
39. Hailpern, B. & Santhanam, P. (2002). Software debugging, testing, and verification. *IBM Systems Journal, 41*(1), 4–12.
40. Haugset, B. & Hanssen, G. K. (2008). Automated acceptance testing: A literature review and an industrial case study. *Proceedings of the Agile Development Conference* (str. 27–38). Toronto: IEEE.

41. Hewlett-Packard Development Company, L.P. (2010). *How to increase efficiency of manual testing*. Pridobljeno 15. maja 2018 iz http://static.ziftsolutions.com/files/8ac4eec43be1b4a1013c73158c8a1f3c.pdf.
42. Hoffman, D. (1999). *Cost benefits analysis of test automation*. Pridobljeno 6. junija 2018 iz https://www.agileconnection.com/sites/default/files/article/file/2014/Cost-Benefit%20Analysis%20of%20Test%20Automation.pdf.
43. Holmström Olsson, H., Alahyari, H. & Bosch, J. (2012). Climbing the" Stairway to Heaven" A multiple-case study exploring barriers in the transition from agile development towards continuous deployment of software. *38th Euromicro Conference on Software Engineering and Advanced Applications* (str. 392–399). Cesme: IEEE.
44. Isabella, A. & Retna, E. (2012). Study paper on test case generation for GUI based testing. *International Journal of Software Engineering & App lications (IJSEA), 3*(1), 139–147.
45. Jalote, P. (2012). *An integrated approach to software engineering*. Boston: Springer Science & Business Media.
46. Jorgensen, P. (2002). *Software testing: a craftman's approach*. New York: CRC Press.
47. Kaplan, R. S. & Norton, D. P. (1993). Putting the Balanced Scorecard to Work. *Harvard Business Review*, *71*(5), 134–147.
48. Karhu, K., Repo, T., Taipale, O. & Smolander, K. (2009). Empirical observations on software testing automation. *Proceedings of the Second International Conference on Software Testing Verification and Validation (ICST 2009)* (str. 201–209). Denver: IEEE.
49. Kassab, M. (2016). Software Testing Practices in Industry: The State of the Practice. *IEEE Software*.
50. Kumar, D. & Mishra, K. K. (2016). The Impacts of Test Automation on Software's Cost, Quality and Time to Market. *Procedia Computer Science, 79*, 8–15.
51. Lefley, F. (2013). The appraisal of ICT and non-ICT capital projects. *International Journal of Managing Projects in Business, 6*(3), 505–533.
52. Leitner, A., Ciupa, I., Meyer, B. & Howard, M. (2007). Reconciling manual and automated testing: The autotest experience. *Proceedings of the 40th Hawaii International International Conference on Systems Science* (str. 261). Waikoloa: IEEE.
53. Lin, C., Graham, P. & McDermid, D. (2005). IS/IT Investment Evaluation and Benefits Realization Issues in Australia. *Journal of Research and Practice in Information Technology, 37*(3), 235–251.
54. Liu, C. (2000). Platform-independent and tool-neutral test descriptions for automated software testing. *Proceedings of the 22nd International Conference on Software Engineering* (str. 713–715). Limerick: IEEE.
55. Liu, H. & Kuan Tan, H. B. (2009). Covering code behavior on input validation in functional testing. *Information and Software Technology, 51*(2), 546–553.
56. Malekzadeh, M. & Ainon, R. (2010). An automatic test case generator for testing safety-critical software systems. *Proceedings of the 2nd International Conference on Computer and Automation Engineering (ICCAE)* (str. 136–167). Singapore: IEEE.
57. Martinez Perez, G., Luis Mateo Navarro, P. & Sevilla Ruiz, D. (2016). Automated GUI Testing Validation guided by Annotated Use Cases. *Gesellschaft für Informatik* (str. 154–252). Lübeck: GI Jahrestagung.
58. Maruthi Prasad, K. V. & Krishna Kishore, J. (2015). Role of Automated GUI Testing in Secure Software Development of Indian Spacecraft Ground Software,

GEOSCHEMACS. *International Journal of Science and Research (IJSR), 4*(8), 570–573.
59. Maurya, V. N. & Kumar, R. (2012). Analytical Study on Manual vs. Automated Testing Using with Simplistic Cost Model. *International Journal of Electronics and Electrical Engineering, 2*(1), 24–35.
60. Melton, R. J. (2015). The Hidden Benefits of Automated Testing. *29th Aerospace Testing Seminar*. Pridobljeno 18. junija 2018 iz https://cosmosrb.com/assets/The_Hidden_Benefits_of_Automated_Testing.pdf.
61. Mohan Doss Gandhi, G. & Pillai, A. S. (2014). Challenges in GUI Test Automation. *International Journal of Computer Theory and Engineering, 6*(2), 192–195.
62. Nakata, C., Zhu, Z. & Kraimer, L. M. (2008). The Complex Contribution of Information Technology Capability to Business Performance. *Journal of Managerial issues, 20*(4), 485–506.
63. Nidhra, S. & Dondeti, J. (2012). Black box and White box Testing Techniques - A Literature Review. *International Journal of Embedded Systems and Appli cations (IJESA), 2*(2), 29–50.
64. Oliver, A., Barrick, J. & Janicki, T. N. (2006). Difficulties in Quantifying IT Projects with Intangible Benefits: A Survey of Local IT Professionals. *The Proceedings of the Conference on Information Systems Applied Research, 2*, 1–12.
65. Patidar, R., Sharma, A. & Dave, R. (2017). Survey on Manual and Automation Testing strategies and Tools for a Software Application. nternational Journal of Advanced *Research in Computer Science and Software Engineering, 7*(4), 283–292.
66. Pawar, S. (2015). *System Testing in Software Testing.* Pridobljeno 22. junija 2018 iz http:// http://softwaretestingbooks.com/system-testing.
67. Pettichord, B. (1999). Seven steps to test automation success. *Proceedings of the STAR West Conference* (str. 136–167). San Jose: Star West.
68. Pezze, M. & Young, M. (2007). *Software Testing and Analysis: Process, Principles and Techniques*. New York: John Wiley & Sons Inc.
69. Pittet, S. (2018). *The different types of software testing*. Pridobljeno 22. junija 2018 iz https://www.atlassian.com/continuous-delivery/different-types-of-software-testing
70. Powell, P. (1992). Information Technology Evaluation: Is It Different? *The Journal of the Operational Research Society, 43*(1), 29–42.
71. Ramler, R. & Wolfmaier, K. (2006). Economic Perspectives in Test Automation: Balancing Automated and Manual Testing with Opportunity Cost. *Proceedings of the 2006 international workshop on Automation of software test* (str. 85–91). Shanghai: ACM.
72. Rathi, P. & Mehra, V. (2015). Analysis of Automation and Manual Testing Using Software Testing Tool. *International Journal of Innovations & Advancement in Computer Science IJIACS*, 4, 710–713.
73. Robson, C. (2002). *Real World Research: A Resource for Social Scientists and Practitioner-Researchers*. Oxford: Blackwell Publishing.
74. Runeson, P. & Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering, 14*(2), 131.
75. Saglietti, F. & Pinte, F. (2010). Automated unit and integration testing for component-based software systems. *Proceedings of the International Workshop on Security and Dependability for Resource Constrained Embedded Systems* (str. 51–56). Vienna: ACM.

76. Shan, L. & Zhu, H. (2009). Generating structurally complex test cases by data mutation: A case study of testing an automated modelling tool. *Computing Journal, 52*(5), 571–588.
77. Sharma , R. M. (2014). Quantitative Analysis of Automation and Manual Testing. *International Journal of Engineering and Innovative Technology (IJEIT), 4(*1), 252–257.
78. Silva, D. G., Abreu, B. T. & Jino, M. (2009). A Simple Approach for Estimation of Execution Effort of Functional Test cases. *2nd International Conference on Software Testing Verification and Validation (ICST)* (str. 289–298). Denver: IEEE.
79. Soh, C. & Markus,, M. (1995). How IT creates Business Value: A Process Theory Synthesis. *Sixteenth International Conference on Information Systems* (str. 29–41). Amsterdam: ICIS
80. Sypolt, G. (2015, 3. junij). *Test Automation KPIs*. Pridobljeno 22. junija 2018 iz https://saucelabs.com/blog/test-automation-kpis.
81. Tallon, P. P., Kraemer, K. & Gurbaxani, V. (2000). Executives' perceptions of the business value of information technology: a process - oriented approach. *Journal of Management Information Systems, 16*(4), 145–173.
82. Tan, R. P. & Edwards, S. (2008). Evaluating Automated Unit Testing in Sulu. *Proceedings of the First International Conference on Software Testing, Verification and Validation* (str. 62–71). Lillehammer: IEEE.
83. TechArcis Solutions, Inc. (2018, 17. junij). *Metrics and Measures to find the True ROI of Test Automation*. Pridobljeno 22. junija 2018 iz https://www.techarcis.com/metrics-and-measures-to-find-the-true-roi-of-test-automation.
84. Thulasee Krishna, S., Sreekanth, S., Perumal, K. & Rajesh Kumar Reddy, K. (2012). Explore 10 Different Types of Software Development Process Models. *International Journal of Computer Science and Information Technologies, 3*(4), 4580–4584.
85. Vaismoradi, M., Turunen, H. & Bondas, T. (2013). Content analysis and thematic analysis: Implications for conducting a qualitative descriptive study. *Nursing & health sciences,15*(3), 398–405.
86. Ward, J. (1994). *A portfolio approach to evaluating information systems investments and setting priorities*. Boston: Springer, Boston, MA.
87. Ward, J. & Daniel, E. (2006). *Benefits Management. Delivering Value form IS & IT Investments*. Chichester: Wiley.
88. Wissink, T. & Amaro, C. (2006). Successful test automation for software maintenance. *Proceedings of the 22nd IEEE International Conference on Software Maintenance* (str. 265–266). Philadelphia: IEEE.
89. Xie, Q. & Memon, A. M. (2006). Model-based testing of community driven open-source GUI applications. *International Conference on Software Maintenance (ICSM)* (str. 145–154). Philadelphia: IEEE.
90. Young, D. C. (2013). *Software Development Methodologies*. Pridobljeno 15. maja 2018 iz https://www.asc.edu/sites/default/files/org_sections/HPC/documents/sw_devel_methods.pdf.

PRILOGE

Priloga 1: Vprašalnik intervjujev

Faza 1

Predstavitev

Predstavim se in vprašanemu predstavim namen, cilj in strukturo intervjuja.

Namen in cilji

Pišem magistersko nalogo, katere namen je ugotoviti upravičenost implementacije orodja za avtomatizacijo testiranj GUI. Cilj je identificirati dejavnike, ki vplivajo na upravičenost in v kakšni meri. Z intervjujem želim pridobiti vaše mnenje in perspektivo na avtomatizacijo testiranj GUI.

Kako se bodo podatki intervjuja uporablili

Vaši odgovori bodo zaupne narave, ki bodo veliko pripomorli k raziskavi in ugotovitvam.

Snemanje

Vprašam za dovoljenje za snemanje intervjuja, da se zagotovi pravilno beleženje.

Predstavitvena vprašanja:

Katere aplikacije testirate z orodjem?

Koliko let izkušenj imate s testiranjem?

Kakšno je vaše delovno mesto?

Pregled avtomatiziranega testiranja GUI:

Prosim opišite vaš celoten proces izvajanja avtomatiziranega testiranja GUI.

Kaj ste pričakovali od orodja?

Opišite težave, ki je orodje prinašalo, če jih je.

Kako bi sprejeli, če ne bi imeli več orodja na voljo za uporabo?

Prednosti in slabosti avtomatizacije testiranja GUI:

Vam je vzela implementacija orodja veliko časa in truda?

a. Ali je bilo potrebno veliko izobraževanja oz. tehničnega znanja?

Ko dobite zahtevo za testiranje ali se takoj lotite testiranja z orodjem?

a. Kako avtomatizirano testiranje GUI deluje v Scrum metodologiji?

Ali lahko pomislite na aktivnosti pred, med in po avtomatiziranem testuranju GUI, ki so pojavile ali izginile z uvedbo orodja?

Kako je test GUI z orodjem vplival na komunikacijo z ostalimi testerji, z razvijalci?

Vam je predstavlja grajenje skript veliko napora?

a. S kakšnimi težavami ste se srečevali?

b. Ali so te težave pojavljale pri vseh testih?

Za katere teste menite da je avtomatizirano testiranje GUI najbolj primerno in zakaj?

Kje vidite prednost, ki vam jo avtomatizirano tesiranje GUI omogoča?

Kje vidite slabosti avtomatiziranega tesiranja GUI?

Faza 2

Predstavitev prednosti in slabosti iz pregleda literature:

Vprašanemu so bile predstavljene prednosti in slabosti avtomatiziranega testiranja iz literature.

Našteval bom prednosti in slabosti avtomatiziranega testiranja, ki jih navajajo različni avtorji iz literature. Prosil bi, da poveste vaše mnenje glede veljavnosti posameznega faktorja v vašem primeru in v kakšni meri menite da je pomemben.

Predstavitev prednosti in slabosti iz prve faze intervjuja, ki jih v pregledu literature ni:

Za dejavnike, ki so se indetificirali v tematski analizi in jih v literaturi ni bilo omenjenih, se je vprašane, ki so jih omenili, vprašalo še za njihovo pomembnost.

Sledečo prednost oz. slabost ste v prvi fazi intervjuja sami omenili, prosim če poveste v kakšni meri menite da je pomembena.