Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Matej Lenarčič

Testiranje modulov pri inačicah spletne aplikacije

diplomsko delo

visokošolski strokovni študijski program prve stopnje računalništvo in informatika

Mentor: izr. prof. dr. Marko Robnik-Šikonja

Ljubljana 2013

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00416/2013
Datum: 05.04.2013

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

Kandidat: **MATEJ LENARČIČ**

Naslov: **TESTIRANJE MODULOV PRI INAČICAH SPLETNE APLIKACIJE**
UNIT TESTING OF WEB APPLICATION VARIANTS

Vrsta naloge: Diplomsko delo visokošolskega strokovnega študija prve stopnje

Tematika naloge:

Mnoge spletne aplikacije nastopajo v številnih inačicah, ki so prilagojene posameznim tehnološkim platformam, funkcionalnim zahtevam uporabnikov ali tipom uporabnikov. Inačicam je potrebno prilagoditi tudi testiranje modulov aplikacije.
Preglejte področje testiranja modulov in opišite dobre rešitve za testiranje inačic. Izdelajte prototip sistema za testiranje bančne aplikacije, ga ovrednotite in na primerih prikažite njegovo delovanje. Vaš pristop primerjajte z obstoječimi rešitvami na tem področju.

Mentor: izr. prof. dr. Marko Robnik Šikonja

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI 2

Dekan: prof. dr. Nikolaj Zimic

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Matej Lenarčič, z vpisno številko **63070133**, sem avtor diplomskega dela z naslovom:

Testiranje modulov pri inačicah spletne aplikacije

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom izr. prof. dr. Marka Robnika Šikonje,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 22. julija 2013 Podpis avtorja:

Zahvalil bi se rad profesorju dr. Marku Robniku Šikonji za mentorstvo pri diplomski nalogi. Hvala tudi vsem, ki so na kakršenkoli način pripomogli k izdelavi diplomske naloge.

Družini in punci.

Kazalo

Povzetek

V diplomskem delu smo predstavili proces testiranja programske opreme, ki je eden ključnih dejavnikov pri zagotavljanju kvalitetnega izdelka. Opisali smo nekaj metod in orodij, s katerimi si pomagamo pri testiranju programske opreme. Naredili smo primerjavo med orodji, ki so najpogosteje uporabljana. Podrobneje smo opisali testiranje modulov, pri orodjih pa smo največ pozornosti posvetili programu SoapUI in različici Pro, ki podpira podatkovno odvisne teste. Za potrebe testiranja bančne aplikacije smo razvili razširitev, ki omogoča testiranje inačic spletnih aplikacij. Opisali smo komponente razširitve, način dela in najpogostejše primere uporabe. Navedli smo primer podatkov in strukturo datotek, ki jih uporabljamo. Primerjali smo našo rešitev za variantno testiranje s SoapUI Pro in predlagali izboljšave.

Abstract

We present the software testing process which is one of the main components of quality in software products. We describe some methods and tools for software testing and compare the most used testing tools. We detaily describe module testing, and program SoapUI with version Pro, which supports data driven testing. For the purpose of testing web banking application, we develope an extension, which allows variant tesing of web applications. We describe components of solution, work process, and the most frequent usage patterns. We present examples of data and data structures used in our solution. We compare our solution for variant testing with SoapUI Pro and sugest certain improvements.

Poglavje 1

Uvod

Pri razvoju programske opreme se vse več pozornosti namenja testiranju. Tudi proces razvoja programske opreme je prilagojen tako, da testiranje vpeljemo že v začetni fazi razvoja. Temeljito testiranje je pogoj za kakovosten izdelek. Pri razvoju spletnih aplikacij se večkrat srečamo s problemom, ko želimo testirati posamezno funkcionalnost za različne trge, različne tipe uporabnikov ali pa želimo preveriti funkcionalnost z različnimi nabori podatkov. Spletni servis nam lahko vrača podatke, ki jih uporabniki z različnimi privilegiji vidijo različno. Po drugi strani, če aplikacijo testiramo z obsežnejšim naborom podatkov, smo lahko bolj prepričani, da smo res temeljito preverili delovanje. Ta problematika je pogosta pri razvoju bančnih aplikacij, kjer banke uporabljajo isto funkcionalnost, ta pa se razlikuje pri podatkih v zahtevkih in odgovorih. Če hočemo testiranje avtomatizirati, ugotovimo, da je potrebno poiskati rešitev, s katero se bomo izognili pisanju specifičnih testov in tako na enostaven način avtomatizirali testiranje.

Variantno testiranje spletnih aplikacij sodi na področje testiranja spletnih storitev in se ne razlikuje bistveno glede pristopov in tehnik testiranja v primerjavi s testiranjem programske opreme na splošno. Lahko gre za testiranje različnih variant, ki se pojavljajo pri uporabi aplikacij, ali pa preprosto za testiranje aplikacije z različnimi nabori podatkov, ki povzročijo različne odzive na aplikacije in različne rezultate.

Pri razvoju aplikacije za elektronsko bančništvo smo želeli poenostaviti testiranje, saj se je nabor bank začel širiti, prilagajanje in pisanje novih testov pa terja ogromno časa. Za testiranje smo uporabljali SoapUI, ki nima možnosti podatkovno odvisnih testov. Za zgled smo vzeli SoapUI Pro, ki jih podpira, odločili pa smo se, da izdelamo svojo rešitev, ki bo še bolj ustrezala našim potrebam.

Najprej bomo opredelili razvoj programske opreme in opisali, kako poteka proces, po katerem razvijamo programsko opremo. Pogledali bomo najpogostejše metode, s katerimi si pomagamo pri razvoju programske opreme, ter kakšne so prednosti in slabosti vsake izmed njih. Opisali bomo testiranje na splošno ter naredili kratek pregled skozi zgodovino. Opisali bomo namen in pomen temeljitega in kakovostnega testiranja ter njegov vpliv na programske izdelke. Spoznali bomo, da je danes večina razvoja programske opreme že v začetnih fazah tesno povezana s testiranjem. Opisali bomo različne pristope, tehnike in metode testiranja. Naredili bomo pregled dveh orodij, ki služita testiranju aplikacij oz. njihovih funkcionalnosti. Podrobneje bomo opisali orodje SoapUI, čemu služi in na kakšen način se uporablja. Opisali bomo problem, na katerega smo naleteli pri razvoju bančnih aplikacij, in rešitev, ki smo jo razvili. Predstavili bomo vse dele rešitve, knjižnico, skripte, način uporabe in hranjenja podatkov.

Našo rešitev bomo primerjali s programom SoapUI Pro, ki je plačljiva različica programa SoapUI in ponuja podatkovno odvisne teste. V zadnjem poglavju bomo pregledali pomanjkljivosti naše rešitve ter preučili možnosti za izboljšave.

Poglavje 2

Testiranje

V tem poglavju bomo podrobneje predstavili proces razvoja programske opreme in opisali nekaj osnovnih pristopov. Predstavili bomo zgodovino testiranja in definirali pojem testiranja. Opisali bomo razliko med statičnim in dinamičnim testiranjem, ki sta del verifikacije in validacije. Spoznali bomo stopnje testiranja in nazadnje še najpogostejše metode testiranja programske opreme.

2.1 Razvoj programske opreme

Industrijo programske opreme zanima predvsem zanesljivost programske opreme. Pri tem obstajajo tri osnovne smernice: cena, čas in kvaliteta. Programska oprema mora biti proizvedena za sprejemljivo ceno, v sprejemljivem času in biti mora kvalitetna. Ti trije parametri usmerjajo in določajo projekt. Poslovni trendi določajo čim krajši čas dobave produkta. Za izdelek to pomeni, da je narejen čim hitreje in v določenem času. Večja kot je produktivnost razvijalca, manjša so tveganja pri času dobave in posledično pri ceni. Poleg teh dveh faktorjev je pomembna tudi sama kvaliteta izdelka. Področje zagotavljanja kakovosti programske opreme lahko razdelimo v šest podskupin: funkcionalnost, zanesljivost, uporabnost, učinkovitost, vzdrževanje in prenosljivost. Kljub pomembni vlogi vsakega od faktorjev pa je najpomemb-

nejši dejavnik zanesljivost. Poleg tega ima pomembno vlogo še vzdrževanje, ki nastopi, ko je izdelek dostavljen stranki. Z uporabo izdelka lahko cena vzdrževanja krepko preseže ceno razvoja. Zaradi visokih stroškov je zaželena programska oprema, ki jo je lahko vzdrževati[1].

2.2 Proces razvoja programske opreme

Proces je izvajanje zaporedja korakov, da dosežemo zastavljen cilj[30]. Pri razvoju programske opreme je cilj zadostiti potrebam uporabnikov. Za dosego visoko kakovostnih produktov in doseganje čim višje produktivnosti potrebujemo optimalne procese. Razlikovati moramo specifikacijo procesa od samega procesa. Proces zajema izvajanje več dejanj, specifikacija procesa pa je opis postopka, ki mu sledimo v določenem projektu, da dosežemo cilj. Proces je največkrat določen na višji ravni kot zaporedje faz. Zaporedje korakov za vsako fazo imenujemo podproces glavnega procesa[1].

2.3 Modeli razvoja programske opreme

Modeli razvoja programske opreme se osredotočajo na aktivnosti, povezane z razvojem programske opreme, na primer načrtovanje, programiranje in testiranje. Procesni model določa glavni proces, ki je razdeljen na več faz, zaporedje, po katerem si faze sledijo, in ostale omejitve in pogoje pri posameznih fazah. Z drugimi besedami, procesni model določa smernice za razvoj primernega procesa na projektu. Zaradi pomembnosti razvojnega procesa so bili vpeljani različni modeli. Predstavljamo nekaj najpogostejših[1].

2.3.1 Slapovni model

Najenostavnejši procesni model je slapovni model, ki predvideva, da si faze sledijo v linearnem zaporedju. Projekt se v tem modelu začne z analizo izvedljivosti, sledita analiza zahtev in planiranje projekta. Ko je analiza zahtev

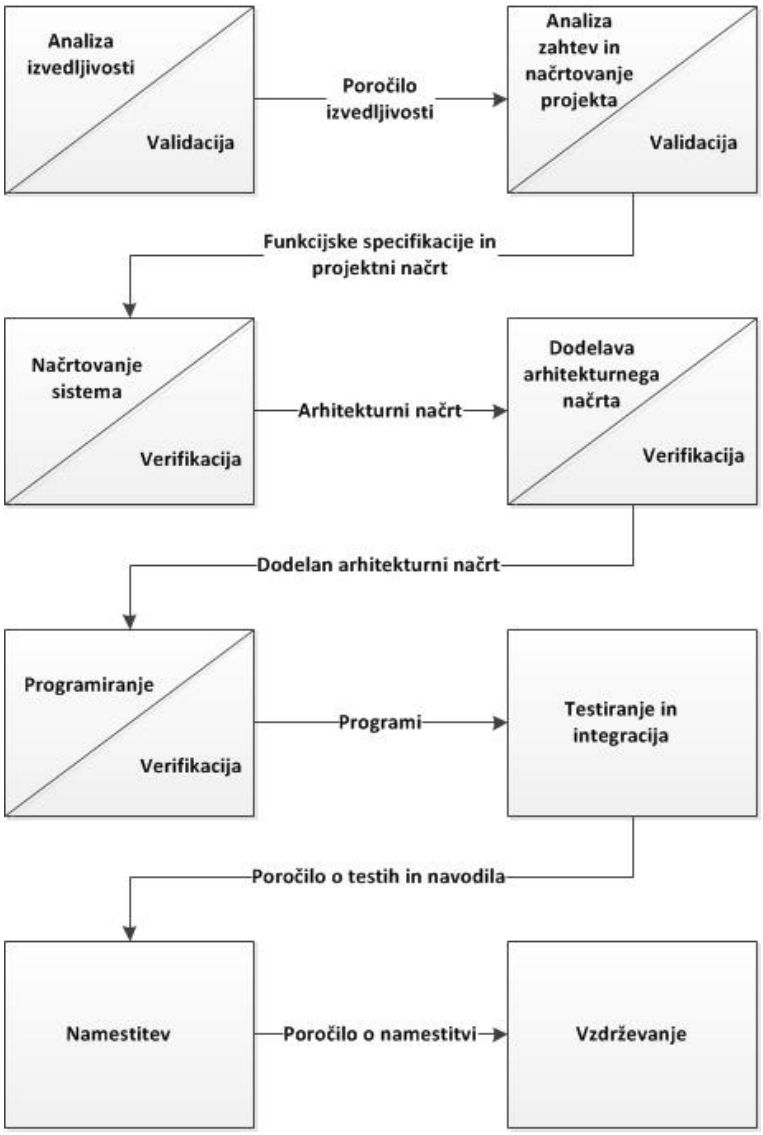
dokončana, se začne načrtovanje oblike, arhitekture itn. Tej fazi sledi kodiranje in nato testiranje ter dostava naročniku. Delovanje je prikazano na sliki 2.1. Ena od glavnih prednosti slapovnega modela je njegova preprostost. To je konceptualno preprost model, ki deli celotno nalogo izgradnje programske opreme v ločene faze, pri čemer vsaka faza predstavlja zaključeno nalogo. Čeprav je slapovni model precej uporabljan, pa ima nekatere omejitve. Model predvideva, da strojne zahteve ostanejo enake skozi ves projekt, kar praktično ni mogoče pri snovanju sistemov, ki jih lahko razvijamo tudi več let. Ena od pomankljivosti je tudi, da je izdelek dostavljen šele na koncu projekta. Tako naročnik ne ve, kaj točno lahko pričakuje. Kljub vsemu pa je slapovni model najbolj uporabljan in morda celo najbolj učinkovit proces[1].

2.3.2 Prototipni model

Ideja prototipnega razvojnega procesa je v nasprotju s prvo omejitvijo slapovnega modela, to je zamrznitev zahtev pred načrtovanjem oz. kodiranjem. Razvoj prototipa je podvržen načrtovanju, kodiranju in testiranju, pri tem pa ni nujno, da je vsaka faza končana v celoti. Pri uporabi tega sistema naročnik dobi občutek, kako se bo sistem obnašal, kar lahko pripomore k izdelavi celovitejšega sistema. Prav tako je metoda učinkovita pri demonstraciji izvedljivosti določenega pristopa. Prototipni model je primeren za projekte, kjer so zahteve težko določljive. Je tudi odličen za zmanjševanje nekaterih tveganj med razvojem[1].

2.3.3 Iterativni razvoj

Iterativni razvoj poskuša združiti prednosti prototipnega in slapovnega modela in se pri tem izogiba omejitvam slapovnega modela. Osnovna ideja je, da programsko opremo razvijamo po korakih, pri čemer na vsakem koraku dodamo novo funkcionalnost, dokler ne dobimo popolnega sistema. Iterativni razvoj postaja vse bolj zaželen. Razlogov za to je več. Stranke brez povra-



Slika 2.1: Slapovni model.

tne informacije ne želijo investirati preveč. Iterativni razvoj ponuja prav to – po vsaki iteraciji je izdelek dostavljen strankam in zato tveganje manjše. Naslednji razlog za tak razvoj je nepredvidljivost sistema. Nikoli namreč ne moremo predvideti vseh zahtev, zato je posodabljanje izdelka nujno. Vsaka iteracija zagotavlja delujoč sistem, iz katerega dobimo povratne informacije za naslednjo iteracijo. Vsi naslednji modeli, ki jih bomo opisali, uporabljajo iterativni pristop[1].

2.3.4 Proces RUP (ang. Rational Unified Process)

Gre za iterativni procesni model, ki je bil najprej zasnovan za objektno orientiran razvoj. RUP deli razvoj programske opreme v več ciklov, od katerih vsak zagotavlja popolnoma delujoč sistem. Vsak cikel se izvaja kot ločen projekt, katerega cilj je dostava dodatnih funkcionalnosti glede na obstoječ sistem (narejen v prejšnjih ciklih). RUP ponuja fleksibilen procesni model, ki sledi iterativnemu pristopu ne le na najvišjem nivoju (s cikli), temveč tudi v fazah znotraj ciklov[1].

2.3.5 Timeboxing model

Za pohitritev razvoja vpeljemo vzporednost med iteracijami. To pomeni, da novo iteracijo začnemo, preden končamo trenutno. S tem teče razvoj nove izdaje vzporedno s trenutno. Na ta način se zmanjša čas, potreben za izdelavo posamezne iteracije. Da to dosežemo, je potrebno pravilno strukturirati iteracije, usklajene morajo biti tudi ekipe, zadolžene za razvoj. Timeboxing je primeren za projekte, ki zahtevajo razvoj mnogih funkcionalnosti v relativno kratkem času na zanesljivi arhitekturi in z uporabo zanesljivih tehnologij[1].

2.3.6 Ekstremno programiranje in agilno načrtovanje

Agilni razvoj se je pojavil leta 1990, kot posledica dokumentno in birokratsko usmerjenih procesov, točneje slapovnega modela. Agilni pristopi temeljijo na skupnih načelih:

- delujoča programska oprema je ključni pokazatelj napredka v projektu,
- za čim hitrejši napredek v projektu mora biti programska oprema razvita in dostavljena hitro in v manjših korakih,
- medsebojna komunikacija ima prednost pred dokumentacijo,
- povratna informacija in vpletenost stranke sta potrebni za razvoj kvalitetne programske opreme,
- preprosta zasnova sistema, ki se razvija in dopolnjuje skozi čas, je boljša od izdelanega sistema za ravnanje z vsemi možnimi scenariji,
- datum dobave je določen s strani pooblaščenih ekip (in ni narekovan).

Najbolj priljubljen in znan pristop med agilnimi metodami je ekstremno programiranje (ang. extreme programming). Ekstremno programiranje in ostale agilne metode so primerne za situacije, kjer sta obseg in hitrost sprememb zahtev visoka in kjer je veliko tveganj. Zaradi zanašanja na močno komunikacijo med člani ekip je metoda učinkovita, ko je ekipa skupaj in šteje do pribl. 20 članov. Ker predvideva vključenost stranke v razvoj in v načrtovanje dobavnih rokov, je ključnega pomena pripravljenost stranke na sodelovanje skozi celoten razvoj, tudi kot del razvojne ekipe[1].

2.4 Zgodovina testiranja

Razdelitev med testiranjem in razhroščevanjem je prvotno predstavil Glenford J. Myers leta 1979. Čeprav je bila njegova pozornost usmerjena na razdelitev testov, je ponazorila željo takratne programske industrije po ločitvi temeljne razvijalske funkcije, kot je recimo razhroščevanje, od na primer verifikacije[5].

2.5 Testiranje

Testiranje je proces, pri katerem preverimo razlike med funkcijskimi specifikacijami in našim izdelkom. Testiranje ocenjuje kakovost izdelka in preverja ustreznost funkcij oz. njihovo pravilno implementacijo. Testiranje programske opreme je del procesa razvoja. Z drugimi besedami, testiranje programske opreme je postopek preverjanja in potrjevanja oz. verifikacije in validacije[6]. Osnovni namen testiranja je dokazati prisotnost ene ali več napak, ne pa lokacije napak[3]. Testiranje lahko vpeljemo v katerikoli fazi razvoja. Največkrat se testiranje vpelje, ko so zahteve jasno določene in je kodiranje končano. V agilnem pristopu je večina testiranja opravljena že v tej fazi[5].

2.6 Statično in dinamično testiranje

Ločimo dve vrsti testiranja, statično in dinamično, zanju poznamo več programskih orodij. Statično testiranje predstavljajo npr. preverjanje kode (ang. code review), predstavitev izdelka (ang. software walkthrough) in pregled izdelka (ang. software inspections), medtem ko med dinamično testiranje sodi izvajanje dejanske kode z naborom testnih primerov oz. scenarijev. Statično testiranje v praksi na žalost vse prevečkrat izpustimo. Dinamično testiranje pride na vrsto, ko je program na voljo za uporabo, z izvajanjem pa moramo začeti, preden je program dokončan. Tipičen način, da to dosežemo, je uporaba navideznih objektov ali pa zagon iz okolja razhroščevalnika. Statično testiranje je del verifikacije, dinamično testiranje pa je del validacije. Skupaj pomagata zagotavljati kakovost programske opreme[5].

2.6.1 Navidezni objekt

V objektno orientiranem programiranju so navidezni objekti simulirani, tako da posnemajo delovanje resničnega objekta na kontroliran način. Pri testiranju modulov lahko navidezni objekti simulirajo (prave) kompleksne objekte.

Uporabimo jih, ko je pravi modul nepraktično ali nemogoče vključiti v testiranje. Če ima objekt katero od naslednjih lasnosti, ga včasih nadomestimo z navideznim objektom:

- rezultat vsebuje podatke, ki jih ne moremo določiti (trenutni čas ...),
- vsebuje stanja, ki jih je nemogoče kreirati ali reproducirati (napaka omrežja ...),
- je počasen (baza, ki se mora postaviti ...),
- vsebuje podatke in metode, potrebne samo za testiranje, ne pa tudi za nadaljnjo uporabo.

Navidezni objekti uporabljajo enak vmesnik kot pravi objekt, ki ga posnemajo[26].

2.7 Verifikacija in validacija

Verifikacija je proces, pri katerem preverjamo, če izdelek zadostuje zahtevam, določenim v predhodni fazi, oz. če se obnaša v skladu s specifikacijo. Za verifikacijo lahko rečemo, da je sinonim za vse vrste preverjanj. O validaciji govorimo, ko programsko opremo na koncu njenega razvoja vrednotimo z namenom, da ugotovimo skladnost z zahtevami[6].

2.8 Metode testiranja programske opreme

2.8.1 Testiranje po metodi »bele škatle«

Testiranje po metodi bele škatle je tehnika, pri kateri se osredotočimo na notranjo strukturo programa[6]. Za kreiranje testnih primerov je potrebno poznavanje sistema in programerske izkušnje. Sistem testiranja po metodi bele škatle lahko vpeljemo pri testiranju modulov, integracije in sistema, najpogosteje na nivoju modulov. S to metodo preverimo poti znotraj modulov,

povezave med moduli pri integraciji in povezave med podsistemi pri sistemskem testiranju. Čeprav s to metodo odkrijemo veliko napak in problemov, pa lahko spregledamo dele, ki niso integrirani, ali manjkajoče zahteve[13].

2.8.2 Testiranje po metodi »črne škatle«

Testiranje po metodi črne škatle (lahko tudi funkcijsko testiranje) ignorira zgradbo sistema in se osredotoča na rezultate, pridobljene z različnimi vhodnimi parametri[6]. Testiranje po metodi črne škatle izvajamo na skoraj vseh nivojih testiranja programske opreme: modulov, integracije, sistema in prevzema[14].

2.8.3 Testiranje po metodi »sive škatle«

Je kombinacija testiranja po metodi bele in črne škatle. Pri tem testiranju preizkuševalec delno pozna delovanje programa, ki ga testira. Koncept testiranja po tej metodi je preprost: če preizkuševalec pozna način delovanja programa, lahko pripravi boljše testne primere, četudi jih potem preverja po metodi črne škatle[15].

2.9 Stopnje testiranja

2.9.1 Testiranje modulov

Pri testiranju modulov preverjamo, ali so posamezni sklopi izvorne kode, posamezni moduli ali več programskih modulov skupaj s povezanimi kontrolnimi podatki in uporabljane procedure primerne za nadaljno uporabo. Vsak modul izdelka je testiran tako, da se preveri njegova pravilna implementacija in pričakovano delovanje. Testiranje se ponavadi izvaja v sami kodi[7]. Razlogi za tako testiranje so trije. Prvič, testiranje modulov je način upravljanja s skupnimi elementi, ki se testirajo, saj je pozornost osredotočena na manjše enote programa. Drugič, testiranje modulov poenostavi razhroščevanje, saj

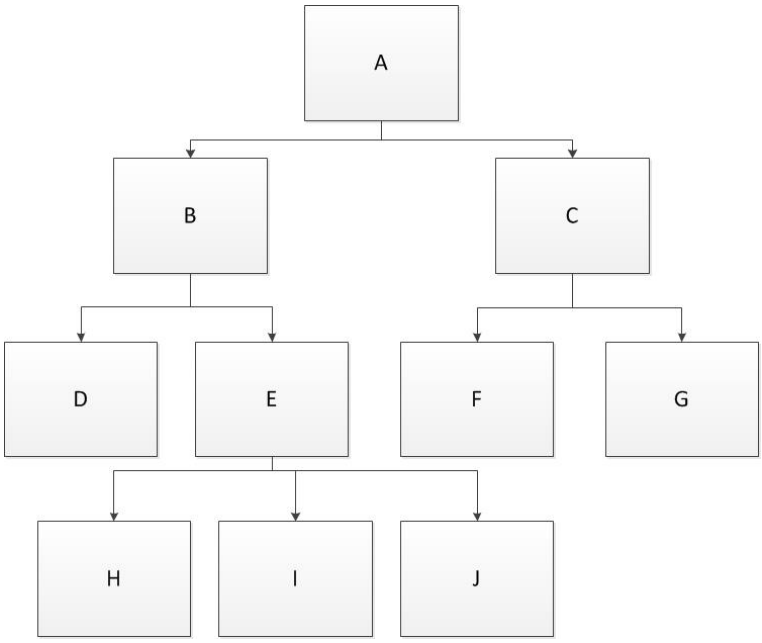
je najdena napaka locirana v testiranem modulu. In nazadnje, testiranje modulov uvaja vzporednost v proces testiranja programa, saj omogoča sočasno testiranje več modulov. Namen testiranja modulov je pokazati, da je modul v nasprotju s specifikacijami[2].

Pri izvajanju procesa testiranja modulov sta ključna dva vidika: oblikovanje učinkovitega sklopa testnih primerov in način, kako so moduli združeni, da tvorijo delujoč program.

Za oblikovanje testnih primerov potrebujemo dva tipa informacij: specifikacije modula in izvorno kodo modula. Specifikacije definirajo pričakovane vhodne in izhodne vrednosti ter funkcije. Testiranje modulov v veliki meri poteka po metodi bele škatle. Ker s testiranjem po tej metodi lažje testiramo dele programa in ker v poznejših fazah testiranja preverjamo druge tipe napak, izdelek največkrat preverimo z eno ali več opcijami testiranja po metodi bele škatle in nato testne primere dopolnimo s testiranjem po metodi črne škatle.

Drugi vidik, združevanje modulov, je pomemben, ker vpliva na obliko, v kateri so testni primeri za modul napisani, vrste testnih orodij, ki jih lahko uporabimo, vrstni red, v katerem so moduli programirani in testirani, stroške testnih primerov in stroške odpravljanja napak. Pri tem imamo na voljo dva pristopa, postopnega (ang. incremental) in ločenega (ang. nonincremental). Pri prvem pristopu pred testiranjem novega modula združimo modul z že testiranimi moduli in nato izvedemo testiranje, pri ločenem pa testiramo vsak modul zase in nato vse združimo v program. Program na sliki 2.2 uporabimo kot primer. Pravokotniki predstavljajo deset modulov v programu. Črte, ki povezujejo module, predstavljajo odvisnost med njimi. Na primer: modul A kliče modula B in C, modul B kliče D in E, modul C kliče F in G in tako naprej. Pri ločenem pristopu torej izvedemo test na vsakem od 10 modulov kot samostojni enoti. Na koncu module združimo v program.

Glede na to, da gre pri postopnem pristopu za združevanje več modulov, obstajata dva pristopa: od zgoraj navzdol (ang. top-down) in od spodaj navzgor (ang. bottom-up). Strategija od zgoraj navzdol vzame za začetek



Slika 2.2: Primer programske strukture.

najvišji oz. začetni modul v programu, naslednji modul pa testiramo tako, da je vsaj eden od nadrejenih modulov testiran pred tem. Na primeru s slike 2.2 začnemo z modulom A, modula J pa ne smemo testirati, dokler ne preverimo modulov B in E.

Nasprotno poteka testiranje od spodaj navzgor, kjer začnemo z najnižjimi moduli (tistimi, ki ne kličejo ostalih) in nadaljujemo z nadrejenimi. Pri tem velja pravilo, da so vsi podrejeni moduli testirani pred nadrejenim. Začnemo torej z enim ali vsemi moduli D, H, I, J, F in K, ki jih lahko testiramo bodisi vzporedno bodisi zaporedno. Tudi tukaj velja, da preden testiramo modul C, preverimo modula F in G[1].

2.9.2 Integracijsko testiranje

Namen integracjiskega testiranja je preveriti vmesnike med komponenetami v programu[6]. To testiranje razkrije napake v vmesniku in interakcijah med integriranimi moduli. Pri testiranju integracije se posamezne programske

enote testirajo v skupinah[4].

2.9.3 Testiranje funkcionalnosti

Testiranje funkcionalnosti je način testiranja po metodi črne škatle. Testni primeri so sestavljeni na podlagi funkcijskih specifikacij ter preverjajo, ali se program obnaša v skladu s pričakovanji[8].

2.9.4 Sistemsko testiranje

Sistemsko testiranje programske opreme poteka tako, da preverimo, če celoten integriran sistem zadošča potrebam. Sistemsko testiranje temelji na metodi črne škatle in ne zahteva poznavanja programske logike ali kode. Sistemsko testiranje za vhodne parametre praviloma vzame vse komponente, ki so uspešno prestale integracijsko testiranje, in programsko opremo, integrirano s strojno opremo[9].

2.9.5 Stres testi

Pri stres testih sistem namerno intenzivno obremenimo, da preverimo stabilnost v kritičnih situacijah. Gre za testiranje onkraj meja zmogljivosti, pogosto do točke, kjer se sistem sesuje, z namenom opazovanja obnašanja pod takimi pogoji[10].

2.9.6 Prevzemno testiranje

Pri prevzemnem testiranju so testi lahko opravljeni s strani končnih uporabnikov, strank ali odjemalcev, ki se odločajo, ali hočejo to opremo ali ne[4]. Ločimo več tipov prevzemnega testiranja: uporabniško prevzemno testiranje, operacijsko prevzemno testiranje, pogodbeno in ureditveno prevzemno testiranje ter alpha in beta testiranje[11].

2.9.7 Regresijsko testiranje

Regresijsko testiranje služi odkritju še neodkritih napak v obstoječi funkcionalnosti oz. v še nerazvitih delih sistema, po spremembah kot so izboljšave, spremembe konfiguracije ali (sistemske) poti v sistemu[12].

Poglavje 3

Rešitve

Za podporo testiranju programske opreme poznamo več programskih rešitev. Opisali bomo nekaj najpogostejših orodij, s katerimi si pomagamo pri testiranju. Podrobneje si bomo pogledali JUnit in SoapUI. Prvega kot primer orodja za testiranje modulov, drugega pa kot primer testiranja funkcionalnosti. Predstavili bomo način dela s SoapUI, strukturo testov in njegove funkcije. Na koncu poglavja sledi primerjava vseh opisanih rešitev.

3.1 Orodja za testiranje

3.1.1 Apache JMeter

Apache JMeter je namizna aplikacija, namenjena testiranju funkcionalnosti in merjenju zmogljivosti. Najprej je bila izdelana za testiranje spletnih aplikacij, vendar se je razširila še na druge testne funkcije. Aplikacijo lahko uporabljamo za testiranje zmogljivosti statičnih in dinamičnih virov. Lahko jo uporabimo za simulacijo preobremenitve strežnika, omrežja ali objekta, za testiranje moči in zanesljivosti, ali pa za analizo celotnega delovanja pod različnimi pogoji[16].

3.1.2 Ranorex

Ranorex je grafični uprabniški vmesnik za testiranje aplikacij. Sam po sebi nima vgrajenega skriptnega jezika, lahko pa si pomagamo z uporabo funkcij iz programskih jezikov, kot sta C# in VB.NET, ki jih vzamemo kot osnovo in nadgradimo s funkcionalnostmi, ki jih ponuja Ranorex[20].

3.1.3 Solex

Solex je brezplačno odprtokodno orodje za testiranje spletnih aplikacij, narejeno kot vtičnik za Eclipse IDE. Ponuja funkcije za simulacijo seje, prilagojene za različne parametre. Solex se obnaša kot HTTP posrednik in posnema vse zahteve in odgovore med spletnim klientom in spletnim strežnikom. Ponovitev scenarija temelji na pošiljanju posnetega (lahko tudi spremenjenega) zahtevka na strežnik in preverjanju odgovora[17].

3.1.4 SoapSonar

SoapSonar je orodje za testiranje servisov in aplikacijskega vmesnika. Ponuja preprost grafični vmesnik, s katerim izvajamo funkcijske, performančne, skladnostne in varnostne teste[19].

3.1.5 Parasoft SOAtest

Parasoft SOAtest poenostavlja kompleksno testiranje, ki je pomembno za poslovno kritične aplikacijske vmesnike, oblake in sestavljene aplikacije. SOAtest pomaga zagotavljati varno, zanesljivo in skladno poslovno aplikacijo z intuitivnim vmesnikom za ustvarjanje, vzdrževanje in izvajanje končnih scenarijev[18].

3.2 JUnit

JUnit je ogrodje, namenjeno testiranju modulov za programski jezik java. Pomemben je pri testno orientiranem razvoju programske opreme[21]. Je del xUnit arhitekture, ki je ogrodje za testiranje modulov[22].

JUnit testi potekajo v javanskih razredih. V JUnitu uporabljamo oznake, s katerimi definiramo, kaj metoda predstavlja, kdaj naj se izvede, ali pa test ignoriramo.

3.2.1 Način uporabe

Test modula je koda, ki jo napiše programer, in izvede neko funkcionalnost v kodi. Testi modulov izvajajo manjše sklope kode (npr. preverjanje metode, razreda ...). Delež kode, ki jo testiramo s temi testi, se imenuje pokritost testov (ang. test coverage). Bolj kot je testiranje pokrito, lažje in bolj varno bomo razvijali izdelek in odpravljali napake, ne da bi spremenili obstoječo funkcionalnost. Ponavadi se testi modulov nahajajo v svojem projektu ali v svoji mapi, da se izognemo mešanju kode[23].

Vse testne metode uporabljajo anotacijo @Test. Podatke praviloma preverjamo s stavki, s katerimi preverimo, ali je podatek pravilen ali ne.

V testnem razredu, ki ga ustvarimo, dodamo test, s katerim želimo preveriti na primer funkcijo za množenje. Predpostavimo, da funkcija pričakuje kot parametra dve števili, ki ju zmnoži. Naša testna metoda izgleda tako:

```
@Test
  public void testMnozi() {
    MojRazred operacije = new MojRazred ();
    assertEquals("10 x 5 je 50", 50, operacije.mnozi
        (10, 5));
  }
```

3.3 SoapUI

SoapUI je brezplačna odprtokodna rešitev za funkcijsko testiranje. Omogoča preprosto kreiranje in izvrševanje testov, regresijsko testiranje in nalaganje testov. V testnem okolju omogoča popolno pokritost in podpira vse standardne protokole in tehnologije[24]. Narejen je izključno za platformo java in uporablja uporabniški vmesnik Swing[25]. V SoapUI je vgrajen skriptni jezik Groovy.

Groovy je objektno orientiran programski jezik, ki deluje na platformi java. Je dinamičen skriptni jezik. Groovy uporablja podobno sintakso kot java in večina kode, napisane v javi, je sintaktično veljavna tudi v Groovyju[27].

SoapUI je na voljo v dveh različicah, brezplačni in plačljivi, ki poleg podpore vsebuje še dodatne funkcionalnosti. Ena izmed njih je podpora podatkovno odvisnim testom, ki so osnova za variantno testiranje, saj v zunanjih datotekah hranimo različne nabore podatkov.

3.3.1 Struktura testov v SoapUI

SoapUi podpira več delovnih okolij (ang. workspace). V vsakem imamo lahko definirane različne projekte, ki jih uporabljamo za testiranje. V posameznem delovnem okolju imamo definirane projekte, ki vsebujejo teste za eno različico aplikacije (na primer za nek trg, za različne uporabnike ...).

Projekt je sestavljen iz več testnih zbirk (ang. TestSuite), ki so zbirke testnih primerov (ang. TestCase) in služijo za združevanje funkcionalnih testov v logične enote. Število testnih zbirk ni omejeno.

Testni primeri so zbirke testnih korakov (ang. TestStep), namenjenih testiranju specifičnih delov servisa. Možno jih je definirati več in uporabljati v povezavi s kompleksnejšimi scenariji.

Testni koraki so osnovni gradniki testnih scenarijev. Dodani so na testne primere in se uporabljajo za izvajanje in kontrolo poteka izvajanja ter validacijo rezultatov testiranega servisa[28].

3.3.2 Naslavljanje in uporaba parametrov

Podatki v SoapUI so shranjeni na različnih mestih v programu, v globalnih nastavitvah, na nivoju projekta, testnih zbirk, testnih primerov ali pa testnih korakov. Do njih dostopamo na različne načine. V zahtevkih in odgovorih do parametrov dostopamo z notacijo:

```
${#Lokacija, kjer se nahaja parameter#ime parametra}
```

Pri pisanju skript naslavljamo spremenljivke glede na lokacijo, kjer se nahajajo:

```
testRunner.testCase.getPropertyValue( " ime parametra "
    )
testRunner.testCase.testSuite.getPropertyValue(" ime
    parametra " )
testRunner.testCase.testSuite.project.getPropertyValue
    (" ime parametra " )
com.eviware.soapui.SoapUI.globalProperties.
    getPropertyValue( " ime parametra " )
```

Na podoben način spremenljivkam določimo in spreminjamo vrednosti. To uporabimo, ko želimo shraniti nek parameter, ki smo ga dobili v odgovoru za nadaljnjo uporabo ali obdelavo. Primer je številka seje, saj tako ne potrebujemo ponovne prijave v sistem, ampak uporabljamo ta parameter v zahtevkih (dokler seja ne poteče). Če se parameter spremeni, ga na enak način posodobimo.

3.3.3 Podatkovno odvisni testi

Podatkovno odvisno testiranje je shranjevanje testnih podatkov (vhod, pričakovan odgovor, zahtevek itn.) v zunanji pomnilnik (podatkovna baza, datoteke Excel, XML datoteka itn.) in njihova uporaba pri poganjanju testov.

SoapUI Pro vsebuje testni korak, s katerim prenesemo testne podatke v program, ti pa se dalje lahko uporabijo v zahtevkih, preverjanjih, skriptah

itn. Vključimo lahko še zanko, ki izvede prejšnje testne korake za vsak nabor testnih podatkov[29].

3.4 Primerjava opisanih rešitev

Rešitve, ki smo jih opisali, se razlikujejo po funkcijah, ki jih ponujajo, ceni, možnosti integracije z drugimi izdelki itn. V tabeli bomo predstavili osnovne značilnosti.

Orodje	**Integracija z ostalimi produkti**	**Podpora podatkovno odvisnim testom**	**Cena**
Apache JMeter	Ne	Ne	Odprtokodni program
Ranorex	Ne	Ne	389€ - 1480 €
Solex	Eclipse	Ne	Odprtokodni program
SoapSonar	Ne	Samo izdaji server in professional	Izdaja Personal brezplačna, ostale izdaje od 799$ dalje
Parasoft SOAtest	Eclipse	Ne	Od 3995$ dalje
JUnit	Eclipse, NetBeans	Ne	Odprtokodni program
SoapUI	Intellij, Eclipse, NetBeans	Samo izdaja SoapUI Pro	Izdaja SoapUI brezplačna, SoapUI Pro od 273 € dalje

Tabela 3.1: Primerjava orodij. Kot vir smo uporabili spletne strani [16-20, 22, 24].

Poglavje 4

SoapUI za variantno testiranje bančne aplikacije

Predstavili bomo našo rešitev, ki deluje kot razširitev za SoapUI in omogoča podatkovno odvisne teste. Opisali bomo osnovne funkcionalnosti in naredili pregled komponent, potrebnih za delovanje. Podrobneje bomo predstavili način uporabe in potek delovanja.

4.1 Opis problema

Spletna aplikacija, ki jo razvijamo, je spletni vmesnik, namenjen bančnim operacijam. Preko spletnih servisov se v programu izpisujejo podatki o računih, transakcijah, uporabnikih itn., ki jih program dobi iz podatkovne baze. Ti podatki se razlikujejo med bankami in trgi. Banke podpirajo različne funkcionalnosti, imajo druge omejitve, med trgi se razlikujejo valute, delovni dnevi itn.

Če želimo poganjati teste, ki so bančno specifični, ugotovimo, da se med različnimi bankami razlikujejo obvezni podatki, ki jih pošiljamo v testnih zahtevkih. Z istim testom namreč želimo preveriti pravilno delovanje servisov, ki so specifični za posamezno banko. Pri tem se razlikujeta le nabor podatkov, ki jih posredujemo, in nabor podatkov, ki jih dobimo kot odgovor.

Kot zgled smo vzeli SoapUI Pro, ki omogoča podatkovno odvisne teste, ker se podatki nahajajo v zunanjih datotekah, SoapUI pa jih vstavi v program, da lahko upravljamo z njimi.

Razvili smo knjižnico, s pomočjo katere beremo podatke iz zunanje datoteke, za shranjevanje podatkov pa smo uporabili program Excel. Potrebno je bilo definirati strukturo map, saj na podlagi tega program ve, kje se nahajajo podatki in kam jih je potrebno vstaviti.

4.2 Podatki

Podatki, ki jih hranimo v zunanjih datotekah, so kompleksni zahtevki in odgovori, ki vsebujejo več parametrov, ti pa se lahko med bankami razlikujejo po številu in vsebini. S tem, ko shranimo te podatke v zunanje datoteke, imamo možnost ne le spremeniti podatke glede na banko, ampak lahko hranimo zahtevke in odgovore z različnimi parametri. V datoteki hranimo le dele zahtevkov in odgovorov, ker je osnovna oblika določena in se ne spreminja med bankami. Celoten zatevek sestavljajo še številka seje, podatek o banki, uporabniku, operaciji, ki jo izvajamo ... Primer zahtevka, ki ga hranimo v zunanjih datotekah:

```
<api-payorder >
    <debtorAccount>${#Project#client1Account1}</
        debtorAccount>
    <creditorName>${#Project#creditorName}</creditorName
        >
    <creditorAccount>${#Project#postdCreditor1Account1
        }</creditorAccount>
    <amount>${amount1}</amount>
    <currencyName>${defaultCurrency}</currencyName>
    <creditorBankName>ime_banke</creditorBankName>
    <purposeCode>A2</purposeCode>
</api-payorder >
```

V zahtevku se nahaja več spremenljivk, ki jih SoapUI zamenja z dejanskimi vrednostimi. Te vrednosti predstavljajo globalne nastavitve, v tem primeru številko računa izdajatelja in prejemnika, ime prejemnika, znesek in valuto. Zahtevek za drugo banko bi vseboval drugo ime banke, drugo kodo namena in več ali manj parametrov.

4.3 Komponente razširitve za SoapUI

4.3.1 Knjižnica

Knjižnica je namenjena branju podatkov iz programa Excel. Struktura dokumenta, v katerem se nahajajo podatki, je določena. Vsebina se nahaja na posameznih zavihkih v datoteki. Vsak stolpec predstavlja eno spremenljivko, v prvi vrstici je zapisano ime, ostale vrstice pa predstavljajo vrednosti spremenljivk. Funkcija, ki vrača nabor podatkov, dobi kot vhodna parametra ime zunanje datoteke in ime zavihka. Parametra se določita glede na lokacijo testnega primera, v katerem želimo izvajati podatkovno odvisne teste.

Ime datoteke, kjer se nahajajo podatki glede na ime delovnega okolja, ime projekta in ime testne zbirke, določimo z ukazom.

```
def sFileName = context.get('${projectDir}') + "/" +
    com.eviware.soapui.SoapUI.globalProperties.
    getPropertyValue("systemid") + "/projects/" +
    testRunner.testCase.testSuite.project.name + "/" +
    testRunner.testCase.testSuite.name + ".xls"
```

Ime zavihka določimo na podlagi testnega primera, v katerem se nahajamo.

```
def sSheet = testRunner.testCase.name
```

V metodi ReadCSV iz imena zavihka, na katerem se nahajajo podatki, določimo številko zavihka.

Sprehodimo se skozi vrstice in stolpce na zavihku in dokler obstajajo podatki, jih vpišemo v tabelo. Tako nastane tabela podatkov, identična datoteki.

4.3.2 Skripte

V SoapUI je vgrajen skriptni jezik. S pomočjo skript lahko preberemo podatke iz Excela, tako da kličemo metodo iz knjižnice, izvajamo teste v zanki ... Za izvajanje različnih operacij smo definirali različne skripte, ki se kličejo na različnih mestih v SoapUI in za različne operacije. Obstajaja pet skript: skripta za branje globalnih nastavitev, skripta za branje podatkov, ki jih vstavljamo v zahtevke ali odgovore, skripta za zanko, ki v primeru, da imamo definiranih več vrednosti parametrov, izvaja enega za drugim, skripta za brisanje vseh podatkov, ki smo jih uvozili, in skripta, s katero zunanjo XML datoteko stisnemo in zakodiramo z base64 algoritmom. Skripte so definirane v zunanjih datotekah in se kličejo na več mestih v programu SoapUI. Lokacija skript je razvidna s slike 4.1.

V skripti preverimo, če se v zunanji datoteki nahajajo podatki. V primeru, da se, na testnem primeru nastavimo parametre iz datoteke. Pri tem ime v prvi vrstici datoteke predstavlja ime parametra, spremenljivka iStartRow pa definira, v kateri vrstici se nahaja vrednost. V primeru, da imamo nastavljeno zanko, se spremenljivka iStartRow povečuje, dokler obstajajo podatki. V ostalih primerih se prebere vrednost iz druge vrstice datoteke.

4.4 Struktura map

Za pravilno delovanje skript je potrebno upoštevati direktorijsko strukturo. Na podlagi strukture program ve, katero datoteko s podatki mora poiskati in katere spremenljivke naložiti v delovno okolje. Na začetku definiramo mapo, v kateri bomo hranili podatke za eno banko. Znotraj nje definiramo še dve mapi. Poimenujemo ju projects in users. V prvi se nahajajo specifični podatki za banko, v drugi pa definiramo različne uporabnike in njihove nastavitve, s katerimi poganjamo teste. Poleg teh map definiramo še globalne lastnosti, specifične za banko. Definiramo jih v Excel datoteki, ki jo poimenujemo globalProperties. V mapi projects se nahajajo mape, poimenovane po projektih, ki jih uvozimo v SoapUI. V teh mapah definiramo Excel da-

toteke, ki vsebujejo podatke za bančno specifične teste. Če ne želimo, da se nekateri testi izvedejo, jih zapišemo v datoteko ignoreList. Prikazana je struktura map, potrebna za pravilno delovanje podatkovno odvisnih testov.

- <ime banke>
  - Mapa »projects«
    - <ime projekta>
      - <testna zbirka>.xls
      - ...
      - ...
      - ignoreList.xls
    - ...
    - ...
    - ...
  - Mapa »users«
    - <uporabnik>
      - properties.xls
    - ...
    - ...
    - ...
  - globalProperties.xls
- scriptForBase64Encoding.groovy
- scriptForLoop.groovy
- scriptForProjectSetup.groovy
- scriptForReadingFromFile.groovy

4.4.1 Struktura podatkov

Podatki so shranjeni v posameznih datotekah Excel in se nahajajo na več zavihkih. Vsak zavihek je poimenovan po testnem primeru, za katerega vsebuje podatke. V zavihku so podatki, potrebni za izvajanje vseh korakov za dani primer. V prvi vrstici je ime spremenljivke, v naslednjih pa definiramo enega ali več naborov podatkov. Ena vrstica ponavadi predstavlja en zahtevek.

4.4.2 Wildcards

Pogosto se zgodi, da vseh podatkov v odgovorih ne moremo preveriti. SoapUI v ta namen dovoljuje, da namesto podatka uporabimo zvezdico, s katero povemo, da je vrednost parametra lahko poljubna.

4.5 Integracija v SoapUI

Predstavili bomo, kako vključimo razširitev v SoapUI, strukturo datotek, skript in knjižnic. Opisali bomo funkcije, ki so na voljo, in nekaj najpogostejših primerov uporabe.

4.5.1 Konfiguracija SoapUI

Da lahko pričnemo rešitev uporabljati, je potrebna integracija v SoapUI. Dodati je potrebno knjižnico za branje iz Excel datotek in Apache knjižnico, ki jo le-ta uporablja. V globalne nastavitve dodamo še dve vrednosti, in sicer: ime uporabnika, s katerim želimo poganjati teste, in parameter, s katerim vklopimo ali izklopimo nalaganje iz zunanjih datotek.

Skripte, ki so shranjene v zunanjih datotekah, pokličemo v SoapUI s skripto. Pri nalaganju globalnih nastavitev to storimo v nastavitveni skripti, ko poženemo projekt. Kjer želimo vstavljati podatke iz zunanjih datotek, kreiramo skripto in vanjo vključimo klic.

Delovni prostor (ang. Workspace) v SoapUI poimenujemo po imenu banke, na katero se navezujejo testi.

4.5.2 Nalaganje bančno specifičnih nastavitev

Na začetku projekta, ki ga želimo izvajati, je potrebno nastaviti uporabniško specifične nastavitve (uporabniško ime, geslo, certifikat ...) in globalne nastavitve, ki veljajo za banko, definirano v imenu delovnega prostora (končna točka, naslov servisov, parametri za delo s podatkovno bazo ...). V SoapUI vključimo klic skripte v Setup script, ki se nahaja v zavihku projekta.

```
def su = new File(context.get('${projectDir}') + "/
    scriptForProjectSetup.groovy")
run(su)
```

4.5.3 Konfiguracija testnih primerov in testnih zbirk

Pri različnih bankah se ne izvajajo vsi testi, ki se nahajajo v projektu. V ta namen lahko definiramo seznam testov (ignoreList), v katerem povemo, da naj se deli projekta ne izvajajo. Pri branju nastavitev za projekt se ti deli projekta onemogočijo.

4.5.4 Implementacija branja iz zunanjih datotek v SoapUI

Na mestu, kjer želimo uporabiti podatke iz zunanjih datotek, ustvarimo Groovy skripto in v njej pokličemo skripto, ki bo prebrala podatke iz ustreznega projekta in jih shranila na nivo testnega primera. Od tam jih vstavljamo na poljubna mesta v korake testnega primera.

```
def su = new File(context.get('${projectDir}') + "/
    scriptForReadingFromFile.groovy")
run(su)
```

4.5.5 Izvajanje testov v zanki

Če želimo pognati test z večjim naborom podatkov, ustvarimo skripto, ki poskrbi, da se test izvede z vsemi nabori podatkov, ki so definirani v zunanji datoteki. Za test dodamo novo skripto, v kateri pokličemo skripto, ki bo izvajala test(e) v zanki.

```
def su = new File(context.get('${projectDir}') + "/
    scriptForLoop.groovy")
run(su)
```

4.5.6 Stiskanje in kodiranje datoteke

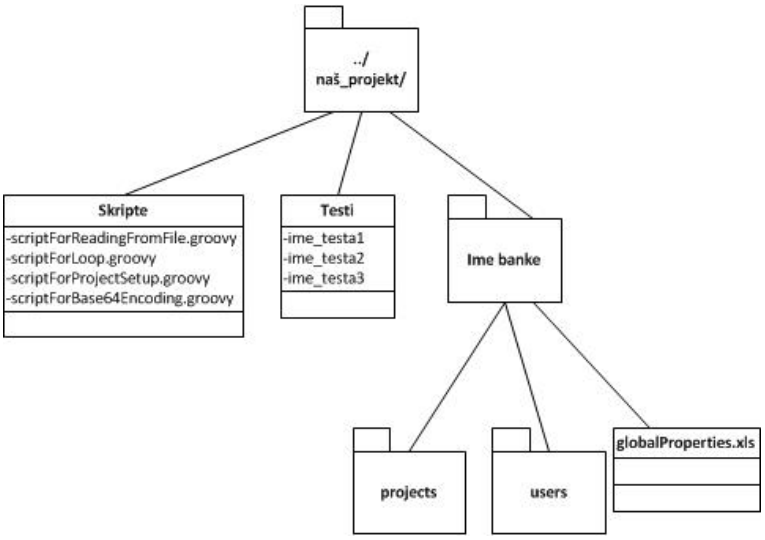
V nekaterih testih uporabljamo uvoz datotek v aplikacijo, v tem primeru uporabimo skripto za stiskanje in kodiranje. V skripti pokličemo drugo skripto, ki bo datoteko pripravila za nadaljnjo uporabo.

```
def su = new File(context.get('${projectDir}') + "/
    scriptForBase64Encoding.groovy")
run(su)
```

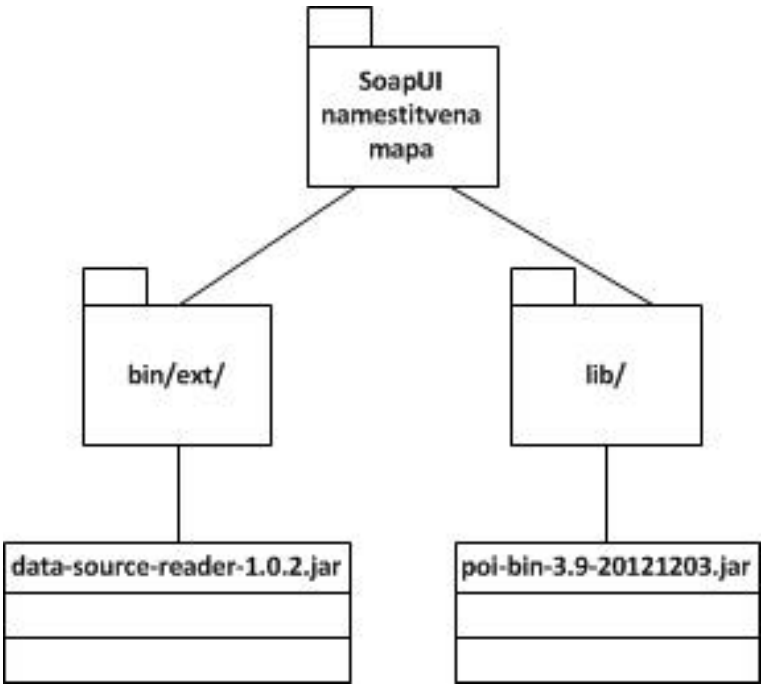
4.6 Shema

Na sliki 4.1 je prikazana struktura map in lokacije skript, testov in zunanjih datotek, v katerih hranimo podatke, potrebne za delovanje našega programa. Na nivoju, kjer se nahajajo testi, dodamo vse skripte, ki jih bomo potrebovali. Za vsako banko ustvarimo svojo mapo, ki jo ustrezno poimenujemo. Struktura map je opisana v poglavju 4.4.

Za delovanje razširitve je potrebno dodati knjižnico Apache POI, to je aplikacijski programski vmesnik za delo z dokumenti Microsoft in knjižnico, ki smo jo napisali za branje iz zunanjih Excel datotek. Ti dve knjižnici vključimo v mapo, kjer je nameščen program SoapUI.



Slika 4.1: Struktura datotek, potrebnih za podatkovno odvisne teste.



Slika 4.2: Struktura datotek, potrebnih za delovanje razširitve.

4.7 Uporaba razširitve za variantno testiranje

Predstavili bomo nekaj primerov uporabe rešitve pri dodajanju banke, testa, kodiranju datoteke in uporabo zanke.

4.7.1 Dodajanje banke

Ko želimo v proces testiranja vključiti novo banko, najprej ustvarimo potrebne mape in pripravimo datoteke, v katerih se nahajajo bančno in uporabniško specifični podatki ter podatki, ki jih bomo potrebovali pri testiranju. V programu kreiramo novo okolje, poimenovano po banki, v katerem bomo hranili specifične teste. V projekt vstavimo skripto, kot je opisano v poglavju 4.5.2. Ta bo pri izvajanju naložila vse bančno specifične nastavitve. Če so podatki v zunanjih datotekah pravilno pripravljeni, lahko poženemo teste.

4.7.2 Dodajanje bančno specifičnega testa

Pri dodajanju novega bančno specifičnega testa pripravimo podatke v zunanjih datotekah. V projektu, kamor želimo dodati test, se premaknemo v testno zbirko in testni primer, oz. jih ustvarimo. Na začetku testnega primera dodamo skripto za branje podatkov iz zunanjih datotek, kot je opisano v 4.5.4.

4.7.3 Kodiranje datoteke in uporaba zanke

Aplikacija podpira nalaganje računov, pripravljenih v XML obliki, v bazo. Dokument mora imeti določeno obliko in mora zadostiti predvidenim omejitvam podatkov. Pri pošiljanju se datoteka stisne in zakodira z base64 algoritmom. Testna datoteka lahko vsebuje enake anotacije, kot jih uporabljamo v SoapUI, ti se pred stiskanjem zamenjajo s podatki iz SoapUI. Na ta način lahko iste datoteke – račune uporabimo za različne banke (spremeni se npr.: ime izdajatelja, valuta ...).

Zanko uporabimo, ko želimo testirati z različnimi nabori podatkov. Zanka se izvaja, dokler v datotekah še obstajajo podatki. Če želimo na primer v bazo uvoziti več računov, v datoteki navedemo poti, kjer se nahajajo. V SoapUI najprej vstavimo skripto za branje podatkov (opisano v poglavju 4.5.4). Pred testni primer za uvoz računov dodamo skripto za kodiranje datoteke (opisano v poglavju 4.5.6), ta bo račun stisnila in zakodirala. Za testom dodamo zanko (opisano v poglavju 4.5.5), ki vsakič poveča številko vrstice, v kateri se podatki nahajajo.

Poglavje 5

Primerjava naše rešitve s SoapUI Pro

Trenutno je SoapUI najbolj razširjena in najbolj popolna rešitev za testiranje spletnih storitev. SoapUI Pro ponuja tudi podatkovno odvisne teste, kar omogoča testiranje različnih variant. Na voljo imamo več načinov hranjenja podatkov v zunanjih datotekah. SoapUI Pro tako podpira hranjenje podatkov v Excel datotekah, XML-jih, tabeli, ko jo definiramo v programu, direktorijih, do podatkov pa lahko dostopamo tudi s povezavo do baze. Pri vsaki možnosti se pojavijo še dodatni parametri, ki jih lahko definiramo, na primer v kateri datoteki, na katerem zavihku, od katere vrstice naprej se nahajajo podatki, shranjeni v Excel datotekah. Ko imamo parametre definirane, SoapUI prebere vsebino in pripravi tabelo s podatki, ki se nahajajo v zunanjih datotekah.

Pri vstavljanju podatkov v zahtevke in odgovore ni potrebno definirati, kateri parameter bi želeli vstavili preko anotacij, saj to storimo s pomočjo menija, ki ga ponuja SoapUI Pro.

Če želimo vstaviti zanko, ki bo teste izvajala vsakič z drugim naborom podatkov, to storimo iz menija, kjer definiramo podatke, ki jih vstavljamo, in teste za izvajanje v zanki.

Predpostavimo, da so podatki definirani v Excel datoteki. V SoapUI Pro

na začetek testnega primera dodamo testni korak podatkovni vir (ang. Data Source), v katerem definiramo prej omenjene parametre. Imena teh parametrov določimo v programu, ki nam nato na podlagi vseh podatkov prikaže vsebino zunanje datoteke v preprosti tabeli. Odpremo testni korak in desno kliknemo na mestu, kjer želimo dodati zunanji parameter. Izberemo podatek, ki naj se zamenja v skladu z vsebino iz zunanje datoteke. V primerjavi s tem je naša rešitev kompleksnejša za uporabo in ima tudi več omejitev (npr. v prvi vrstici se mora nahajati ime parametra).

Naša razširitev je namenjena testiranju točno določene aplikacije, zato ima tudi svoje prednosti. Tako je izdelava testov bolj enostavna, saj imamo postavljena določena pravila (ime zavihka v Excelu predstavlja ime testne zbirke, ime datoteke predstavlja ime datoteke ...). Na ta način je struktura zunanjih datotek in njihova uporaba poenostavljena in preprečuje, da bi uporabnik v eno datoteko vpisal podatke za nepovezane teste ali vse podatke hranil v eni sami datoteki, kar bi otežilo vzdrževanje pri razširjanju aplikacije. Razširitev na podlagi imena delovnega okolja izbere in v program naloži bančno specifične podatke. Tako na enostaven način zamenjamo banko za testiranje in uporabnika (kreiramo lahko uporabnike z različnimi pravicami, podatki, omejitvami ...). Te prednosti izkoristimo pri avtomatizaciji testiranja, pomagajo nam tudi pri ročnem testiranju.

Prednost je tudi definicija podatkov v zunanjih datotekah. Naša rešitev omogoča, da podatke definiramo v Excelu in jih poljubno razširjamo (dodajamo in odvzemamo parametre ...), kar pa ne velja za SoapUI Pro, v katerem moramo število parametrov definirati v programu. Naša rešitev je za testiranje bančne aplikacije zelo uporabna, saj ustreza točno določenim zahtevam. Splošno gledano pa so druge rešitve na trgu bolj vsestranske, saj niso pogojene s strukturo in imeni datotek.

Zaradi določenih zahtev in cenovno ugodnejše rešitve je bila naša rešitev za variantno testiranje spletne aplikacij smiselna, saj nam ponuja več prednosti.

Poglavje 6

Zaključek

Izdelali smo rešitev za variantno testiranje, ki zadošča potrebam testiranja bančne aplikacije. Razširitev za SoapUI sestavlja knjižnica in več skript, od katerih ima vsaka svojo funkcionalnost. Knjižnica je razvita v javi in omogoča branje podatkov iz zunanjih datotek. Skripte, ki so napisane v skriptnem jeziku Groovy, omogočajo izvajanje podatkovno odvisnih testov, izvajanje testov v zanki z različnimi nabori podatkov in kodiranje datotek. Določili smo strukturo hranjenja datotek, ki jo je potrebno upoštevati za pravilno delovanje.

Naša rešitev za variantno testiranje bančne aplikacije je dosegla cilj, ki smo si ga zadali, to so podatkovno odvisni testi in preprosto preklapljanje med testiranjem različnih bank z različnimi uporabniki. S knjižnico, s pomočjo katere beremo podatke iz zunanjih datotek in skriptami, ki služijo za delovanje le-te, smo dobili možnost testiranja spletne aplikacije, ne le za različne inačice, ampak tudi možnost preverjanja funkcionalnosti s širšim naborom podatkov.

Razširitev za SoapUI vsebuje tudi nekatere pomanjkljivosti. V Excel datotekah se morajo podatki nahajati v tekstovnem formatu, drugače program ne zna prebrati vseh podatkov. Program ne zna delati s podatki, ki so shranjeni v Excel dokumentih s končnico xlsx.

Pri uporabi znaka zvezdica moramo biti še posebej previdni. Ker se lahko

namesto nje nahaja karkoli, se lahko zgodi, da v parametru podatka ni oz. je prazen. V tem primeru bo preverjanje uspelo, kar je napačno. V primeru, da imamo definiranih več zank, je potrebno vsako skripto poimenovati drugače, da program ve, katero zanko in podatke mora prebrati.

Imamo tudi nekaj idej za izboljšave. Pri dodajanju testnih korakov ponuja SoapUI nabor različnih komponent, s katerimi oblikujemo teste. Če želimo vstaviti našo rešitev, je potrebno najprej kreirati Groovy skripto in v njej definirati skripto, ki se bo izvajala. Lažje bi bilo, če bi imeli že v naboru možnosti vključeno našo rešitev, ki bi jo lahko dodali samo s klikom in pri tem definirali še dodatne parametre (npr. ime zunanje datoteke, na katerem zavihku se nahajajo podatki za test, v kateri vrstici se nahajajo podatki ...). Podatke bi lahko nalagali preko menija, kjer bi se pojavili parametri, ki so na voljo. Izvajanje v zanki bi lahko prilagodili tako, da bi definirali prvi in zadnji test, ki se izvedeta.

Literatura

[1] Pankaj Jalote, A Concise Introduction to Software Engineering, British Library Cataloguing in Publication Data, 2008.

[2] Glenford J. Myers, The art of software testing, Second edition, John Wiley & Sons, Hoboken, 2004.

[3] Kaner, Cem, Falk, Jack and Nguyen, Hung Quoc, Testing Computer Software, 2nd Ed. New York, John Wiley and Sons, Inc., 1999.

[4] Tomaž Dogša, Verifikacija in validacija programske opreme, Univerza v Mariboru,Tehniška fakulteta, Maribor, 1993.

[5] Software testing. Wikipedija, http://en.wikipedia.org/wiki/ Software_testing (zadnji obisk: maj 2013).

[6] What is software testing? What are the different types of testing? Code project, http://www.codeproject.com/Tips/351122/What-is-software-testing-What-are-the-different-ty (zadnji obisk: maj 2013).

[7] Unit testing. Wikipedia, https://en.wikipedia.org/wiki/Unit_testing (zadnji obisk: maj 2013).

[8] Functional testing. Wikipedia, http://en.wikipedia.org/wiki/ Functional_testing (zadnji obisk: maj 2013).

[9] System testing. Wikipedia, http://en.wikipedia.org/wiki/System_testing (zadnji obisk: maj 2013).

[10] Stress testing. Wikipedia, http://en.wikipedia.org/wiki/Stress_testing (zadnji obisk: maj 2013).

[11] Acceptance testing. Wikipedia, http://en.wikipedia.org/wiki/ Acceptance_testing (zadnji obisk: maj 2013).

[12] Regression testing. Wikipedia, http://en.wikipedia.org/wiki/ Regression_testing (zadnji obisk: maj 2013).

[13] White-box testing. Wikipedia, http://en.wikipedia.org/wiki/White-box_testing (zadnji obisk: maj 2013).

[14] Black-box testing. Wikipedia, http://en.wikipedia.org/wiki/Black-box_testing (zadnji obisk: maj 2013).

[15] What is gray/grey box testing? Dostopno na: http://www.robdavispe.com/free2/What-02210-is-gray-grey-box-testing.html (zadnji obisk: maj 2013).

[16] Apache JMeter. Apache software foundation, http://jmeter.apache.org/ (zadnji obisk: maj 2013).

[17] Solex. Dostopno na: http://solex.sourceforge.net/ (zadnji obisk: maj 2013).

[18] Parasoft SOAtest. Parasoft, http://www.parasoft.com/jsp/products/soatest.jsp?itemId=101 (zadnji obisk: maj 2013).

[19] SoapSonar. CrossCheck, http://www.crosschecknet.com/products/ soapsonar.php (zadnji obisk: maj 2013).

[20] Ranorex. Wikipedia, http://en.wikipedia.org/wiki/Ranorex (zadnji obisk: maj 2013).

[21] Junit. Wikipedia, http://en.wikipedia.org/wiki/JUnit (zadnji obisk: maj 2013).

[22] JUnit. Dostopno na: http://junit.org/ (zadnji obisk: maj 2013).

[23] JUnit – Tutorial. Dostopno na: http://www.vogella.com/articles/JUnit/article.html (zadnji obisk: maj 2013).

[24] What is SoapUI? SoapUI, http://www.soapui.org/About-SoapUI/what-is-soapui.html (zadnji obisk: maj 2013).

[25] SoapUI. Wikipedia, http://en.wikipedia.org/wiki/SoapUI (zadnji obisk: maj 2013).

[26] Mock object. Wikipedia, http://en.wikipedia.org/wiki/Mock_object (zadnji obisk: maj 2013).

[27] Groovy. Wikipedia, http://en.wikipedia.org/wiki/ Groovy_(programming_language) (zadnji obisk: maj 2013).

[28] Structuring and running tests. SoapUI, http://www.soapui.org/Functional-Testing/structuring-and-running-tests.html (zadnji obisk: maj 2013).

[29] Functional tests. SoapUI, http://www.soapui.org/Data-Driven-Testing/functional-tests.html (zadnji obisk: maj 2013).

[30] IEEE. IEEE standard gloassary of software engineering terminology. Tehnical report, 1990.