View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by ePrints.FRI

UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Dejan Veternik

Avtomatizacija testiranja funkcionalnosti z orodjem Selenium WebDriver

DIPLOMSKO DELO

UNIVERZITETNI ŠTUDIJSKI PROGRAM PRVE STOPNJE RAČUNALNIŠTVO IN INFORMATIKA

Ljubljana, 2014

UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Dejan Veternik

Avtomatizacija testiranja funkcionalnosti z orodjem Selenium WebDriver

DIPLOMSKO DELO

UNIVERZITETNI ŠTUDIJSKI PROGRAM PRVE STOPNJE RAČUNALNIŠTVO IN INFORMATIKA

MENTOR: izr. prof. dr. Viljan Mahnič

Ljubljana, 2014

To delo je ponujeno pod licenco *Creative Commons Priznanje avtorstva-Deljenje pod enakimi pogoji 2.5 Slovenija* (ali novejšo različico). To pomeni, da se tako besedilo, slike, grafi in druge sestavine dela kot tudi rezultati diplomskega dela lahko prosto distribuirajo, reproducirajo, uporabljajo, priobčujejo javnosti in predelujejo, pod pogojem, da se jasno in vidno navede avtorja in naslov tega dela in da se v primeru spremembe, preoblikovanja ali uporabe tega dela v svojem delu, lahko distribuira predelava le pod licenco, ki je enaka tej. Podrobnosti licence so dostopne na spletni strani creativecommons.si ali na Inštitutu za intelektualno lastnino, Streliška 1, 1000 Ljubljana.

Izvorna koda diplomskega dela, njeni rezultati in v ta namen razvita programska oprema je ponujena pod licenco *GNU General Public License*, različica 3 (ali novejša). To pomeni, da se lahko prosto distribuira in/ali predeluje pod njenimi pogoji. Podrobnosti licence so dostopne na spletni strani http://www.gnu.org/licenses.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Proučite značilnosti testno vodenega razvoja programske opreme, ki ga propagirajo agilne metode, in prikažite, kako se ta način testiranja vklaplja v metodo Scrum. Na podlagi tega predlagajte postopek avtomatskega testiranja spletnih aplikacij z orodjem Selenium WebDriver. Opišite možnosti, ki jih za testiranje funkcionalnosti nudi to orodje, in prikažite uporabo predlaganega postopka na konkretnem primeru spletne aplikacije GemaLogic. Na podlagi pridobljenih izkušenj analizirajte prednosti in pomanjkljivosti avtomatskega testiranja v primerjavi z ročnim.

IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Spodaj podpisani Dejan Veternik, z vpisno številko **63020174**, sem avtor diplomskega dela z naslovom:

Avtomatizacija testiranja funkcionalnosti z orodjem Selenium WebDriver

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom izr. prof. dr. Viljana Mahniča
- 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 na svetovnem spletu preko univerzitetnega spletnega arhiva

V Ljubljani, dne 15. avgusta 2014 Podpis avtorja:

Zahvaljujem se mentorju izr. prof. dr. Viljanu Mahniču za skrbno vodenje in pomoč pri izdelavi diplomske naloge. Zahvala gre sodelavcem podjetja Solvera Lynx d.d., ki so omogočili izdelavo projekta in bili vedno na voljo za pomoč. Posebna zahvala gre punci Bini, ki me je v najtežjih trenutkih študija spodbujala in mi stala ob strani. Na koncu študijske poti bi se zahvalil tudi družini za podporo in potrpljenje v času celotnega študija.

Kazalo

Povzetek

Abstract

Poglavje 1 Uvod ... **1**

Poglavje 2 Scrum in zagotavljanje kakovosti ... **3**

2.1 Scrum ... 4

2.2 Avtomatizacija testiranja ... 5

2.2.1 Nujnost avtomatizacije ... 6

2.2.2 Kaj avtomatizirati ... 7

2.3 Umestitev testiranja v Scrum ... 8

2.3.1 Nočna gradnja ... 9

2.3.2 Predizdaja (Release Candidate) ... 9

Poglavje 3 Avtomatizacija testiranja z orodjem Selenium WebDriver ... **11**

3.1 Splošno o testiranju funkcionalnosti ... 11

3.2 Splošno o orodju Selenium WebDriver ... 12

3.3 Interakcija z brskalnikom ... 13

3.4 Dostop do spletnih elementov ... 13

3.4.1 Izbiranje po identifikaciji ali imenu ... 14

3.4.2 Izbiranje s pomočjo CSS ... 15

3.4.3 XPath izbirnik ... 16

3.4.4 Ostali podprti izbirniki ... 17

3.5 Zagotavljanje preprostosti in preglednosti kode ... 17

3.5.1 Načrtovalski vzorec PageObjects ... 18

3.5.2 Razred PageFactory ... 18

3.6 Selenium WebDriver in Ajax klici ... 19

3.6.1 Implicitno čakanje ... 19
3.6.2 Eksplicitno čakanje ... 20
3.7 Selenium WebDriver in ogrodje JUnit ... 20
3.7.1 Uporaba anotacij ... 21
3.7.2 Preverjanje rezultatov ... 22
3.7.3 Podpora v ostalih aplikacijah ... 22
3.7.4 Možnost parametriziranih testov ... 24
3.8 Testiranje na različnih brskalnikih ... 24
3.8.1 InternetExplorerDriver ... 25
3.8.2 ChromeDriver ... 26
3.8.3 FirefoxDriver ... 27
3.8.4 HtmlUnitDriver ... 28
3.8.5 RemoteWebDriver ... 29
Poglavje 4 Sistem za testiranje funkcionalnosti aplikacije GemaLogic ... 31
4.1 O aplikaciji GemaLogic ... 31
4.1.1 Časovnica razvoja GemaLogic po metodologiji Scrum ... 32
4.2 Testno okolje ... 33
4.2.1 Delovna postaja ... 33
4.2.2 Strežnik za izvajanje testnih primerov ... 34
4.2.3 Strežnik za sprotno integracijo ... 34
4.2.4 Shramba Subversion ... 34
4.3 Avtomatizacija testnih primerov ... 35
4.3.1 Tabela indentifikatorjev ... 35
4.3.2 Oblika implementacije v Javi ... 36
4.3.3 Glavni razred ... 37
4.3.4 Prijava v aplikacijo ... 38
4.3.5 Brisanje rezultatov prejšnjih testiranj ... 39
4.3.6 Testiranje grafičnega prikaza ... 41
4.3.7 Testiranje tabelaričnega prikaza ... 42

4.4 Samodejno zaganjanje testnih primerov ... 43
4.4.1 Avtomatsko testiranje nočne gradnje ... 44
4.4.2 Avtomatsko testiranje predizdaje ... 48
Poglavje 5 Ročno in avtomatsko testiranje v praksi ... 51
5.1 Priprava dokumentacije ... 51
5.2 Načrtovanje in programiranje ... 51
5.3 Izvrševanje testov ... 52
5.4 Urejanje testnih primerov ... 52
Poglavje 6 Sklepne ugotovitve ... 53

Seznam uporabljenih kratic

kratica	angleško	slovensko
CI	continious integration	sprotna integracija
HTML	hyper text markup language	jezik za označevanje nadbesedila
DOM	document object model	dokumentno objektni model
XML	extensible markup language	razširljiv označevalni jezik
AJAX	asynchronous javascript and XML	asinhroni javascript in XML
CSS	cascading style sheets	kaskadne stilske predloge
COM	component object model	komponentni objektni model
JSON	javascript object notation	javascript objektna notacija
FTP	file transfer protocol	protokol za prenos datotek
XHTML	XML hyper text markup language	jezik za označevanje nadbesedila XML

Povzetek

Danes se vse več podjetij odloča za razvoj programske opreme po agilnih metodologijah. Zagotavljanje kakovosti zaradi visokih zahtev in kratkih rokov zahteva avtomatizacijo procesa testiranja na vseh ravneh. Avtomatizacija testiranja funkcionalnosti za spletne aplikacije zaradi počasnejšega izvajanja in krhkosti testov lahko predstavlja izziv. Selenium WebDriver je eno vodilnih orodij za avtomatizacijo testiranja funkcionalnosti spletnih aplikacij. V tem diplomskem delu bo bralec spoznal koncept delovanja orodja, njegove dobre in slabe strani in se naučil njegove uporabe. Dejanska uporaba predlaganih postopkov je prikazana v drugem delu naloge. Opisuje celovito rešitev za samodejno testiranje predizdaje in nočnih gradenj kompleksnejše spletne aplikacije. Delo vsebuje tudi krajše poglavje, ki je namenjeno primerjavi ročnega in avtomatskega izvajanja testov na podlagi pridobljenih izkušenj.

Ključne besede: selenium, webdriver, scrum, testiranje funkcionalnosti, avtomatizacija

Abstract

More and more companies are deciding to develop software using agile methods. Because of high demands and tight deadlines, automating the testing process at all levels is required to provide enough quality. Due to time consuming implementation and fragility of tests, automation of functional tests (for web applications) can be a challenge. Selenium WebDriver is one of the leading tools for automating testing web applications. In this thesis, the reader will become acquainted with the concept of the tool, its pros and cons and will learn how to use it. The second part of the thesis describes the actual solution of automated testing. It shows the reader how to automate testing of nightly builds and release candidate of a complex web application. The thesis also contains a short chapter, which is intended for comparison between manual and automated testing on the basis of the experiences gained.

Keywords: selenium, webdriver, scrum, functional testing, automation

Poglavje 1 Uvod

Trg danes zahteva visoko kakovost programske opreme, ki je dostavljena ob pravem času in hkrati deluje tako, da izpolnjuje vse naročnikove zahteve. Ker so aplikacije istočasno postajale (in še vedno postajajo) vedno bolj kompleksne, je zagotovitev teh zahtev pri uporabi tradicionalnih metodologij razvoja vedno težja. Zato so se razvile nove – agilne metodologije. Te so se izkazale za učinkovitejše.

Agilne metodologije razvoja programske opreme temeljijo na prilagodljivosti. Kot že samo ime pove je za doseg te lastnosti potrebno čim bolj pogosto (v ciklih) izdajati preizkušene in delujoče komponente programske opreme. S tem pridobimo zelo koristne povratne informacije Obenem zmanjšamo tveganje, da bo naročnik z izdelkom nezadovoljen, saj se morebitne pomanjkljivosti odpravljajo sproti.

Da je programska oprema, ki jo izdamo na koncu vsakega cikla, preizkušena in delujoča, je potrebno zagotoviti z zadostno količino testov. Testiranja je potrebno v vsakem ciklu agilnega razvoja čim večkrat ponoviti, hkrati pa njihovo število narašča s kompleksnostjo aplikacije. Zato je njihova avtomatizacija nujna. Avtomatski testi se izvajajo na več ravneh. Prva raven so testi enot, ki so zelo hitri in se izvajajo ob vsaki spremembi aplikacije. Zadnja raven so testi za testiranje funkcionalnosti, ki zelo veliko prispevajo k zadovoljstvu naročnika, saj v končni fazi lahko rečemo, da simulirajo uporabnikovo interakcijo z aplikacijo. Zato so zelo občutljivi na kakršnekoli spremembe uporabniškega vmesnika. Ta problem še posebej pride do izraza pri testiranju spletnih aplikacij, saj se spreminjajo bolj pogosto, obenem pa je njihov prikaz celo odvisen od brskalnika. Implementacija, vzdrževanje in samo izvajanje takih avtomatskih testov, je precej časovno potratno, kar pa predstavlja svojevrsten izziv pri umestitvi v danes najbolj priljubljeno agilno metodologijo Scrum.

V tem diplomskem delu, kot raziskavi z določenim namenom, si bomo v drugem poglavju najprej ogledali osnove agilnih metod in Scruma, zakaj je pri uporabi agilnih metod avtomatizacija testov nujna in kako lahko uvrstimo testiranje funkcionalnosti v sam cikel.

V tretjem poglavju bomo pod drobnogled vzeli danes najbolj priljubljeno orodje za avtomatizacijo testov za testiranje funkcionalnosti spletnih aplikacij Selenium WebDriver. Bralec bo dobil podrobno predstavo, kako orodje deluje, kako se uporablja in česa je zmožno.

Kako se lotiti implementacije avtomatskih testov za kompleksnejšo spletno aplikacijo GemaLogic, razne programerske prijeme in kako omogočiti avtomatsko izvajanje testov, si bomo na resničnem primeru ogledali v četrtem poglavju.

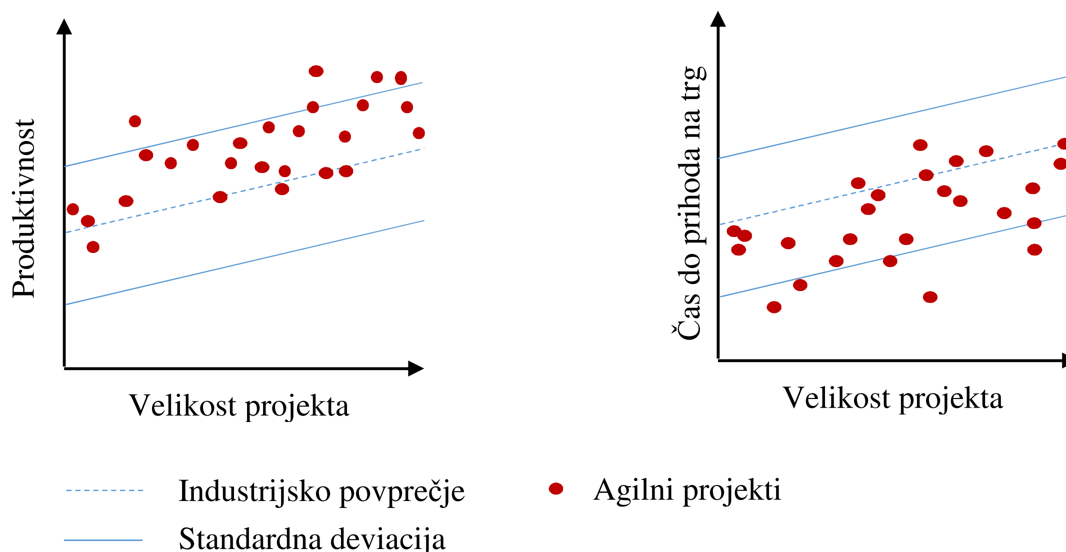
Za konec bomo preverili doprinos avtomatizacije na našem primeru in omenili nekaj možnosti izboljšav.

Poglavje 2 Scrum in zagotavljanje kakovosti

Danes vse več podjetij prisega na agilne metode razvoja programske opreme, ker jim te prinašajo:

- večjo produktivnost in nižje stroške,
- izboljšano angažiranost zaposlenih in več zadovoljstva pri delu,
- krajši čas realizacije; od razvoja do izdaje produkta na trg,
- višjo kakovost in
- večje zadovoljstvo vpletenih [2].

Omenjene točke so podprte s številnimi strokovnimi raziskavami. Zanimivejša je raziskava, ki jo je leta 2008 v partnerstvu s QSM Associates objavil Michael Mah [21]. Združenje QSM Associates je več kot 15 let zbiralo podatke o produktivnosti, kakovosti in drugih lastnostih pri razvoju različnih projektov. Michael je primerjal 26 agilnih projektov z bazo QSMA, ki je vsebovala 7500 tradicionalnih projektov. Izsledki govorijo v prid agilnim metodam. Na sliki 2.1 sta prikazana grafa, ki glede na povprečje dokazujeta boljšo produktivnost in hitrejše lansiranje na trg.

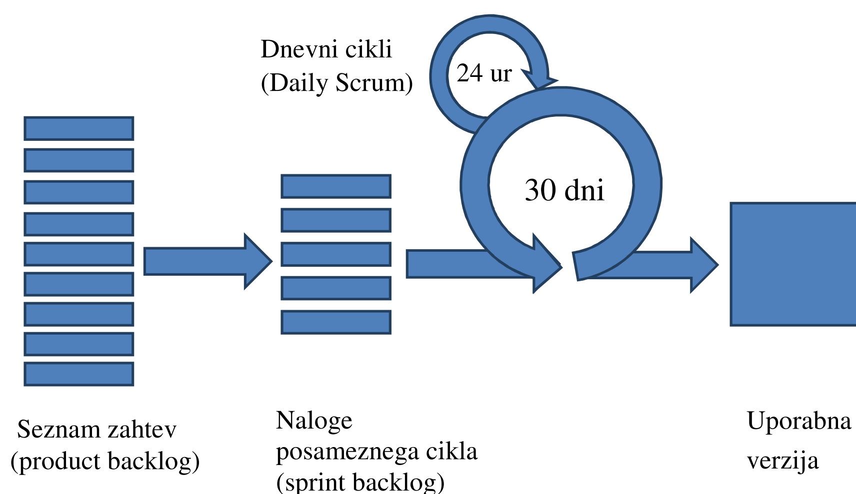


Slika 2.1: Iz levega grafa je razvidno, da so agilne ekipe produktivnejše od industrijskega povprečja, medtem ko desni graf prikazuje krajši čas do izdaje produkta na trg (povzeto po [21]).

V času pisanja tega diplomskega dela, je daleč najbolj priljubljena agilna metoda vodenja razvoja programske opreme Scrum.

2.1 Scrum

Scrum je lahko ogrodje, ki je ustvarjeno za pomoč majhnim skupinam programerjev pri razvoju kompleksnih aplikacij [8]. Da lahko zagotovimo agilnost, je potrebno delo ekipe razdeliti na kratke iteracije, ki se imenujejo Sprint. Na sliki 2.2 je prikazan običajen potek razvoja programske opreme po tej metodologiji.



Slika 2.2: Prikaz običajnega cikla razvoja programske opreme po metodologiji Scrum (povzeto po [9]).

Pred vsako iteracijo je potrebno opraviti sestanek, kjer se definira seznam zahtev, parametrov, argumentov, dokumentacije in načrtovanja, ki jih je potrebno uresničiti za izdajo novega izdelka. Zahteve na tem seznamu so razvrščene po prioriteti. Iz njih se tvori naloge, ki se bodo izvajale v prihajajočem Sprintu. Sprint običajno traja 30 dni, med njegovim izvajanjem pa se vsak dan ob točno določenem času opravi kratek dnevni sestanek, ki se imenuje dnevni Scrum. Namen tega sestanka je razprava o tekočem delu znotraj programerske ekipe. Na koncu vsakega cikla (Sprinta) je na voljo uporaben izdelek.

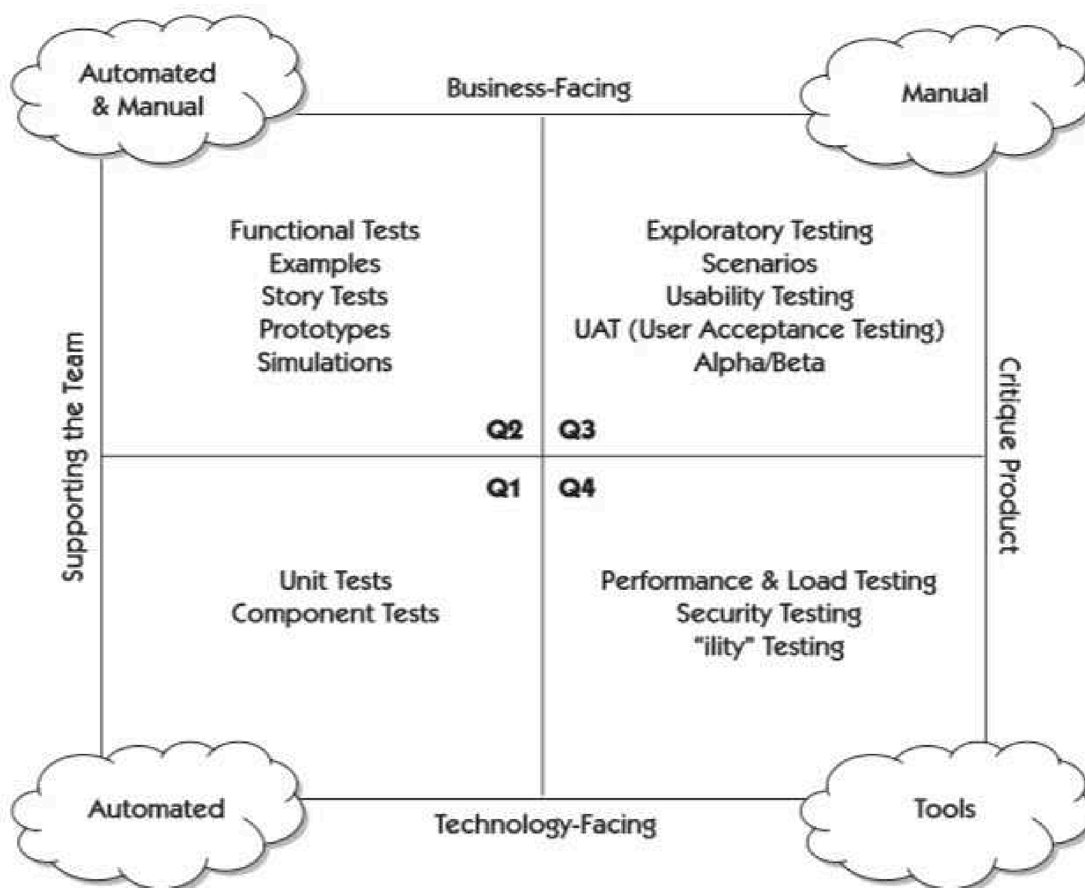
Scrum v osnovni različici ne pokrije vseh procesov razvoja programske opreme. Zato ga veliko organizacij dopolni z dodatnimi procesi, da se ustvari bolj celovito izvajanje [9]. V naslednjih

poglavjih si bomo ogledali primer dopolnitve s procesom obsežnejšega avtomatskega testiranja.

2.2 Avtomatizacija testiranja

Dolžina iteracije (Sprinta) je kompromis [7]. Ima velik vpliv na to, kako dobro se Scrum obnese v podjetju [10]. Kratke iteracije omogočajo boljšo agilnost, dolge iteracije pa več časa za dosego ciljev in manjši odstotek porabljenega časa za sestanke za izdelavo plana iteracije.

Scrum predpisuje fiksno dolžino iteracije, ki se je s tekom Sprinta nikakor ne sme spreminjati. Datum izdaje je tako že vnaprej znan. Seveda morajo biti takrat doseženi vsi cilji, izdaja pa testirana do te mere, da je primerna za produkcijsko namestitev. Pri takem tempu se lahko zgodi, da se testna ekipa izkaže kot ozko grlo. V izogib temu problemu je avtomatizacija testiranja nujna. Na sliki 2.3 so prikazani ti. kvadranti agilnega testiranja, ki prikazujejo katerih tipov testov se morajo lotiti testerji pri agilnem razvoju programske opreme, katera področja pokrivajo in kaj je potrebno avtomatizirati.



Slika 2.3: Kvadranti agilnega testiranja [3].

2.2.1 Nujnost avtomatizacije

Eden izmed najboljših razlogov za avtomatizacijo je ta, da je preprosto ročno testiranje časovno preveč potratno. Časovna potratnost ročnega testiranja se izkaže posebno v primeru testiranja funkcionalnosti, ko se večina testov odvija na nivoju uporabniškega vmesnika. Ob uporabi metodologije Scrum za razvoj programske opreme, je potrebno testiranje ponoviti po vsaki iteraciji. Zato pogosto govorimo tudi o ciklu testiranja. Pri vsakem ciklu testiranja je potrebno opraviti vse teste. Kakršnokoli iskanje bližnjic se lahko drago obrestuje. Aplikacija, ki jo testiramo, se sproti razvija in postaja z vsako iteracijo obsežnejša in kompleksnejša. Potrebno je zagotoviti dobro pokritost funkcionalnosti s testi, zato sorazmerno z obsežnostjo aplikacije narašča tudi število avtomatskih testov. Z ročnim testiranjem v času, ki ga imamo na razpolago, ni mogoče izvesti vseh scenarijev, kar pa lahko pripelje do tega, da spregledamo pomembno pomanjkljivost.

Pokritost kode s testi pa ni edini razlog, zaradi katerega tester pri ročnem testiranju lahko spregleda napake. Zanesljivost ročnega testiranja je močno izpostavljena človeškim faktorjem. Zaradi ročnega ponavljanja vedno enakih scenarijev se lahko pojavi monotonost, ki vodi do preskakovanja testnih primerov. S tem dejanjem se močno poveča možnost, da se spregleda tudi očitnejše napake. Drugi človeški razlog, zaradi katerega lahko prihaja do napak pri testiranju je preutrujenost, ki pa se ji izognemo z avtomatizacijo. Če se velik del testov opravi avtomatsko, testerju ostane dovolj energije, da zbrano opravi ročni del testiranja.

Programer lahko s spreminjanjem obstoječe ali dodajanjem nove kode zlahka pokvari funkcionalnost, ki je že pravilno delovala. Z regresijskim testiranjem to lahko preprečimo, saj je ostala koda že pokrita z avtomatskimi testi, ki se bodo samodejno sprožili ob potrditvi (angl. commit) kode v shrambo (angl. repository). Takrat je programerju veliko lažje odpraviti napako kot pa kasneje, saj se še dobro zaveda s čim jo je povzročil. Na tej točki je smiselno omeniti tudi testno voden razvoj, ki prinaša številne prednosti, kot so preprost inkrementalen razvoj (angl. incremental development). Razvoj poteka v kratkih iteracijah. Glavna prednost tega je, da imamo skoraj vsak trenutek na voljo delujoč program, čeprav še nima implementiranih vseh funkcionalnosti [5].

Vlaganja v avtomatizacijo so nujna. Poleg začetnega vložka, je vložek potreben tudi kasneje. Avtomatske teste moramo ves čas prilagajati spremembam aplikacije, ki se testira. Če niso ažurni, ne morejo služiti svojemu namenu, ko pa mu enkrat služijo, se stroški zaradi hitrega odkrivanja napak hitro povrnejo. Avtomatizacija testiranja je steber prilagajanja hitremu razvoju.

2.2.2 Kaj avtomatizirati

Očitno je, da večkrat kot se določen proces ponavlja, večje bodo prednosti avtomatizacije, kar nazorno pokažejo primerjave ročnega in avtomatiziranega testiranja.
Gotovo morajo biti avtomatizirani testi enot, ki so hitri, preprosti in se izvajajo najpogosteje.
Prav tako je pametno avtomatizirati teste, ki testirajo mehanizme s katerimi se aplikacija povezuje z drugimi aplikacijami in okoljem. Tukaj so mišljeni programski vmesniki, spletni servisi, vtičnice… Taki testi so zelo učinkoviti, hkrati pa je tudi njihova implementacija zelo preprosta.
S stresnimi testi običajno testiramo meje naše aplikacije – poskušamo ustvariti ekstremne pogoje, kar pa v večini primerov ročno sploh ni mogoče izvesti. Implementacija kode, ki na različne načine obremeni aplikacijo, je enostavna.

Priporočljiva je tudi avtomatizacija testov, ki temeljijo na primerjavi nečesa, saj je avtomatsko primerjanje veliko lažje in hitrejše kot ročno. V to skupino spadajo tudi testi, pri katerih se rezultat primerja z v naprej (v datoteke) shranjenimi vrednostmi. Ta pristop imenujemo tudi testiranje z zlatimi datotekami (angl. golden files testing), ki se pri testiranju funkcionalnosti zelo pogosto uporablja.

Več premisleka je potrebnega pri testih pri katerih interakcija z aplikacijo poteka preko uporabniškega vmesnika. Ta vrsta testov ima za podjetje običajno najnižjo donosnost naložbe [3]. V primerjavi z drugimi tipi testov, ki smo jih že omenili, so počasnejši hkrati pa tudi zelo krhki, saj lahko padejo že po majhni spremembi uporabniškega vmesnika. Zato zahtevajo veliko pozornosti razvijalca testov. Vseh testov ni mogoče avtomatizirati, ali pa se jih enostavno ni racionalno. Da lažje določimo, kaj bomo avtomatizirali, si lahko pomagamo tako, da odgovorimo na nekaj osnovnih vprašanj o testih in njihovem izvajanju. V nadaljevanju je naštetih nekaj primerov takšnih vprašanj:

- Kolikokrat se bo testni primer izvajal?
- Kakšna je težavnost implementacije?
- Koliko časa bomo prihranili?
- Bodo pričakovani rezultati konsistentni?
- Se testna procedura lahko preprosto ročno ponovi?

Donosnost naložbe pri tem tipu testov lahko povečamo s pravilno izbiro orodja in pravilno implementacijo, ki omogoča preprosto vzdrževanje testov [3].

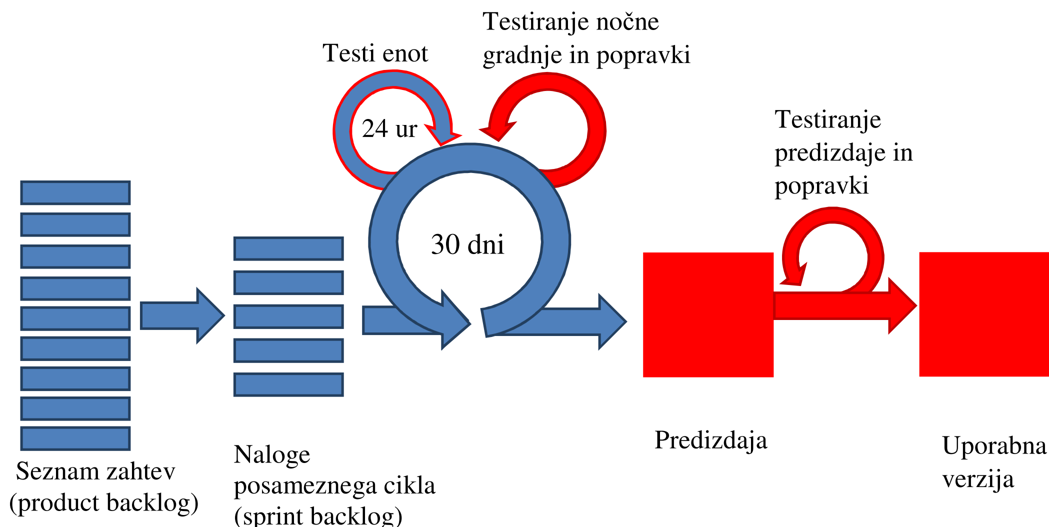
2.3 Umestitev testiranja v Scrum

Testi komponent in testi enot so lahko v celoti avtomatizirani. Testni primeri so kratki in se izvajajo zelo hitro (nekaj milisekund). Zato ne predstavljajo problemov pri umestitvi v iteracijo Scruma, saj se lahko v okviru sprotne integracije zaganjajo ob vsaki potrditvi kode v shrambo. Težava se pojavi pri izvajanju testiranja funkcionalnosti in raznih simulacij (kvadrant Q2 na sliki 2.3). Ti testi so prepočasni, da bi se zaganjali ob vsaki potrditvi kode, obstajajo pa tudi potrebe, ki jih sploh ni mogoče avtomatizirati. Torej je te teste potrebno zaganjati tako, da niso ozko grlo pri izvajanju Sprinta:

1. izvajajo se vsak dan (ponoči), kot testiranje nočne gradnje (angl. nightly build),

2. ročni in polavtomatski (za uspešno izvedbo potrebno posredovanje testerja) se izvajajo med Sprintoma oziroma v tako imenovanem času priprave nove verzije. Testira se predizdaja (angl. nightly build).

Na sliki 2.4. je dopolnjen diagram iz poglavja 1.1.1. Dodatno je označeno izvajanje testov enot, nočne gradnje in predizdaje.



Slika 2.4: Dopolnjena skica razvoja programske opreme po metodologiji Scrum. Sliki 2.1 smo dodali še cikle za izvajanje testov enot (ob vsaki potrditvi kode), nočne gradnje in testiranje predizdaje.

V tej diplomski nalogi se bomo osredotočili le na avtomatizacijo testiranja funkcionalnosti, ki pokrije tudi nekatere druge vrste testov, kot sta stresno testiranje in testiranje zmogljivosti. Preden pa si ogledamo priljubljeno orodje, ki to omogoča, na kratko osvežimo, kaj je to nočna gradnja in kaj predizdaja.

2.3.1 Nočna gradnja

Pri razvoju programske opreme imenujemo nevtralna gradnja gradnjo aplikacije, ki odseva trenutno stanje izvorne kode, ki jo razvijalci shranjujejo v sistem za nadzor različic. Nevtralna gradnja je izvedena na sistemu, ki ne vsebuje orodij, knjižnic, ali katere druge programske opreme, ki je bila uporabljena za izdelavo aplikacije [13]. S tem se prepričamo, da aplikacija lahko deluje samostojno, brez napak, ki jih v razvojnem okolju ni možno zaznati in je neodvisna od strojne opreme. Te vrste gradnja se običajno izvaja v času, ko v pisarnah ni prisotnih oseb (ponoči) zato jo pogosto imenujemo tudi nočna gradnja.

2.3.2 Predizdaja (Release Candidate)

Predizdaja ni namenjena splošni rabi; predstavlja potencialnega kandidata za končno izdajo. Vsebuje vse funkcionalne in nefunkcionalne zahteve, ki so bile izbrane za implementacijo v času Sprinta. Novih funkcionalnosti se tej gradnji do konca cikla zanesljivo ne bo dodajalo, potrebno pa je obsežnejše testiranje. V kolikor se v predizdaji po koncu testiranja najdejo napake, se jim po internem pogovoru določi prioriteto. Če je prioriteta najvišja (angl. blocker), je potrebno kodo popraviti in pripraviti novo predizdajo. Ta cikel se ponavlja, dokler ne dosežemo izdaje brez napak z najvišjim nivojem prioritete.

Poglavje 3 Avtomatizacija testiranja z orodjem Selenium WebDriver

V tem poglavju si bomo podrobneje ogledali orodje Selenium WebDriver [22] in kakšne možnosti nam to orodje nudi. Bralec bo po koncu tega poglavja znal uporabiti orodje in implementirati osnovne avtomatske teste za testiranje funkcionalnosti.

Kot smo že omenili, je Selenium WebDriver orodje za avtomatizacijo testov za testiranje funkcionalnosti. Zato si bomo najprej ogledali nekaj osnov o testiranju funkcionalnosti. Sledi arhitektura orodja Selenium WebDriver in prelet brskalnikov, ki jih je orodje sposobno avtomatizirati. Najbolj priljubljenim brskalnikom bomo namenili več pozornosti. Nato si bomo ogledali, kako lahko dostopamo do spletnih elementov s pomočjo izbirnikov in katero vrsto je priporočeno uporabiti v različnih situacijah. Sledilo bo poglavje o ogrodju JUnit, ki se primarno uporablja za testiranje enot, a z združitvijo z orodjem Selenium WebDriver postane uporabno tudi pri testiranju funkcionalnosti. Poglavje bomo zaključili z načini predstavitve rezultatov.

Orodje podpira pisanje testnih primerov v več različnih programskih jezikih. V tej diplomski nalogi bomo uporabljali izključno programski jezik Java.

3.1 Splošno o testiranju funkcionalnosti

Testiranje funkcionalnosti je proces zagotavljanja kakovosti, katerega testni primeri temeljijo na specifikaciji, napisani iz uporabniške perspektive. To so testi, ki potrdijo, da sistem deluje tako, kot od njega pričakujejo uporabniki [12]. Običajno jih ne zanima, kako dobro je spisana koda v ozadju. Svoje mnenje o aplikaciji si ustvarijo na podlagi drugih lastnosti, kot sta uporabnost, stabilnost in intuitivnost. Zato so zelo pomembni, saj z dobro izvedenimi testi za testiranje funkcionalnosti zagotovimo ravno te lastnosti. Za zagotavljanje stabilnosti je potrebno testirati čim več scenarijev – tudi take, ki nam lahko na prvi pogled sploh ne delujejo logično.
Testiranje funkcionalnosti se deli na dve kategoriji:

- pozitivno testiranje funkcionalnosti in
- negativno testiranje funkcionalnosti.

Pri pozitivnem testiranju funkcionalnosti preverjamo funkcionalnost aplikacije s pravilnimi scenariji in preverjamo, če so rezultati enaki pričakovanim, medtem ko negativno testiranje funkcionalnosti vključuje tudi kombinacije, ki so neveljavne in velikokrat tudi presegajo namen delovanja naše aplikacije.

3.2 Splošno o orodju Selenium WebDriver

Selenium WebDriver je orodje za avtomatsko testiranje spletnih aplikacij. Orodje je nastalo z združitvijo dveh nekdaj najbolj priljubljenih orodij za testiranje spletnih aplikacij – orodij Selenium in WebDriver. Ustvarjalci so združili preprostost in intuitivnost orodja Selenium z dodelanim objektno orientiranim programskim vmesnikom orodja WebDriver. Novonastalo orodje Selenium WebDriver je večkrat poimenovano tudi kot Selenium 2.

Za razliko od predhodnika – orodja Selenium 1.0, ki avtomatizira brskalnik s pomočjo knjižnic JavaScript, WebDriver uporablja brskalnikov API vmesnik. Vmesniki se od brskalnika do brskalnika razlikujejo, zato nekakšna univerzalna metoda za krmiljenje vseh vrst brskalnikov ne obstaja. Za vsak brskalnik potrebujemo ti. gonilnik brskalnika, preko katerega se lahko »pogovarjamo« z njim. Medtem, ko je po eni strani to slaba lastnost WebDriverja, je po drugi strani to dobro, saj tako vsak brskalnik lahko nadziramo optimalno, ker je gonilnik optimiziran točno za ta brskalnik.

Celoten sistem WebDriver je sestavljen iz štirih različnih sekcij [1]:

- **WebDriver API :** to je tisti del, na katerega metode se sklicuje uporabnik med pisanjem testnih primerov.
- **WebDriver SPI:** del, ki opravlja dekodiranje ukazov in elementov, ki jih programer naslavlja v svoji kodi.
- **Protokol JSON Wire:** omogoča komunikacijo med strežnikom ali brskalnikom in aplikacijo, kjer se izvaja koda (testni primeri).
- **Brskalnik/Selenium strežnik:** dekodira JSON objekte, ki jih dobi preko JSON Wire protokola in glede na dobljene ukaze usmerja brskalnik.

Na tem mestu naj omenimo, da je teste mogoče enostavno posneti in nato tudi predvajati. To omogoča posebna razširitev za brskalnik Firefox, ki se imenuje Selenium IDE. Medtem, ko se orodje dobro izkaže pri enostavnih statičnih straneh, ima pri kompleksnejših dinamičnih

spletnih aplikacijah precej težav. Zaradi teme diplomskega dela nas zanima samo programerska implementacija testov, zato se v podroben opis orodja Selenium IDE ne bomo podajali.

3.3 Interakcija z brskalnikom

Selenium WebDriver sprejema ukaze in jih posreduje brskalniku [11]. Ta mehanizem je implementiran preko tako imenovanega gonilnika brskalnika, ki pošilja ukaze brskalniku in od njega nazaj pridobiva rezultate [11], kot je prikazano na sliki 3.1. Navadno gonilnik brskalnika dejansko zažene brskalnik in do njega dostopa na način, ki je specifičen za vsak brskalnik. Izjema je gonilnik brskalnika HtmlUnit, ki si ga bomo podrobneje ogledali v poglavju *Testiranje na različnih brskalnikih*.

Slika 3.1: Komunikacija z brskalnikom preko gonilnika brskalnika.

3.4 Dostop do spletnih elementov

Uporabniški vmesnik, je sestavljen iz različnih spletnih elementov. Kot spletne elemente si predstavljamo gumbe, povezave, tekst, slike,… Skratka vse, kar je možno zapisati z značkami (angl. tags) v kodi HTML. Pri pisanju avtomatskih testov, ki simulirajo interakcijo uporabnika, mora programer orodju za avtomatizacijo povedati, do katerega spletnega elementa v danem trenutku želi dostopati. To dosežemo z uporabo posebnih izrazov, ki jih imenujemo izbirniki. Selenium WebDriver podpira uporabo več različnih tipov izbirnikov. Iz uporabniškega vidika med seboj razlikujejo predvsem po uporabnosti in hitrosti dostopa do elementov. Za razvijalca avtomatskih testov sta tudi zelo pomembni preglednost in sledljivost. Lepo formirani izbirniki močno pomagajo pri razumevanju kode.

Spodobi se, da si najprej ogledamo mehanizem za interakcijo z elementi iz tehničnega vidika. Za lociranje elementov z orodjem Selenium WebDriver se uporabljata metodi `findElement()` in `findElements()`, ki jih priskrbita razreda `WebDriver` in `WebElement` [1]. Metodi kot parameter sprejmeta objekt tipa `By`, preko katerega izberemo vrsto izbirnika. V kolikor je element najden, metoda `findElement()` vrne nov objekt tipa `WebElement`, metoda `findElements()` pa seznam (tip `List<WebElement>`). Razred

`WebElement` je abstraktnega tipa `interface` in vsebuje metode za interakcijo z elementom, kot so na primer `click()`, `sendKeys()` in `getText()`. V kolikor noben element uporabljenemu izbirniku ne ustreza, metoda `findElement()` sproži izjemo `noSuchElementException`, metoda `findElements()` pa vrne prazen seznam.

Pravilen izbor tipa izbirnika je pri dinamičnih aplikacijah zelo pomemben in včasih se od razvijalca testov zahteva sprejemanje kompromisov med preglednostjo in uporabnostjo. Na primer, izbiranje po identifikaciji je najbolj enostaven in pregleden način, hkrati pa je pri dinamičnih aplikacijah zaradi samodejnega poimenovanja elementov skoraj neuporaben. Zato se je potrebno poslužiti nekoliko bolj kompleksnih izbirnikov, kot sta CSS in XPath.

Za boljše razumevanje si bomo njihovo uporabo ogledali na enostavnem elementu HTML, ki predstavlja gumb in ga v kodi definiramo, kot je prikazano v izvorni kodi 3.1.

```
<input type="button" id="prijava">
```

Izvorna koda 3.1: Definicija HTML gumba `prijava`.

3.4.1 Izbiranje po identifikaciji ali imenu

Ta tip izbirnikov je najbolj enostaven za uporabo, istočasno pa je koda preglednejša, saj je za izbiro elementa dovolj, da je znan njegov enoličen identifikator (atribut `id`) ali enolično ime (atribut `name`). Če so identifikatorji oziroma imena pametno dodeljena, je iz kode moč takoj ugotoviti za kateri element gre. Programerju avtomatskih testnih primerov ni potrebno tvoriti daljših izrazov, kakršne običajno tvorimo pri izbirniku XPath. Ker ima gumb iz zgornjega primera lepo določen atribut `id` (identifikator), za klik nanj lahko zapišemo:

```
driver.findElement(by.id("prijava")).click();
```

Izvorna koda 3.2: Implementacija izbire spletnega elementa s pomočjo izbirnika XPath in klika na izbran element.

Žal se pri dinamičnih aplikacijah izkaže, da je ta vrsta izbiranja večinoma nemogoča, saj so elementi poimenovani avtomatsko in se lahko celo spreminjajo ob vsakem nalaganju strani. Avtomatsko generirane identifikatorje prepoznamo po obliki `j_idXYZ`, kjer XYZ predstavljajo števke od 0 do 9, ki se spreminjajo. V takih primerih je potrebno izbrati drug tip

izbirnika, ki omogoča dostop do elementa ne glede na njegov identifikator. Taka primera izbirnikov sta izbirnika CSS in XPath.

3.4.2 Izbiranje s pomočjo CSS

CSS (angl. Cascading Style Sheets) je jezik, s katerim razvijalec določi izgled (stil) poljubne HTML strani. Stili so v svoji definiciji poimenovani; na njih pa se sklicujejo spletni elementi v kodi HTML. Orodje Selenium prav to lastnost izkorišča za dostop do elementov.

Izbiranje s pomočjo CSS je priporočljiv pristop. Največji prednosti izbiranja preko CSS sta njegova hitrost in enako delovanje v vseh brskalnikih. Za razliko od izbirnika XPath, tukaj obstaja samo ena implementacija, ki ustreza standardu CSS3.

Poglavje je namenoma poimenovano kot *Izbiranje s pomočjo CSS* in ne *Izbirnik CSS*, saj CSS že sam po sebi vsebuje mehanizme, ki se imenujejo *izbirniki CSS*. Zaradi enakega imena, izbirnikov CSS ne smemo mešati s Seleniumovimi izbirniki CSS o katerih govori to poglavje. V dinamičnih aplikacijah so najbolj pogosto uporabljeni izbirniki CSS tisti, ki opisujejo razmerja med staršem in otrokom. Tabela 3.1 prikazuje nekaj teh primerov.

Izbirnik CSS	**Opis**
:last-child/:last-child	Vrne prvega/zadnjega otroka nekega elementa.
:nth-child(n)	Vrne n-tega otroka nekega elementa.
:nth-of-type(n)	Vrne n-ti element nekega tipa.
:preceding-sibling	Vrne naslednje vozlišče (element).

Tabela 3.1: Seznam najbolj pogosto uporabljenih izbirnikov CSS.

Naj opozorimo še, da je za izvajanje testov, ki uporabljajo izbiranje s pomočjo CSS, potrebno uporabljati dovolj nov brskalnik. Starejši brskalniki imajo namreč težave s prepoznavo nekaterih izbirnikov tega tipa.

Za klik na gumb iz zgornjega primera z uporabo izbiranja s pomočjo CSS lahko zapišemo, kot je prikazano v izvorni kodi 3.3.

```
driver.findElement(By.cssSelector("input#prijava")).click();
```

Izvorna koda 3.3: Implementacija izbire spletnega elementa s pomočjo izbirnika CSS in klika na izbran element.

Kljub vsem preverjeno dobrim lastnostim tega tipa izbiranja, pri dinamičnih aplikacijah velikokrat potrebujemo več fleksibilnosti pri tvorbi samega izraza za izbiranje. Takrat programer lahko uporabi izbirnike XPath.

3.4.3 XPath izbirnik

XPath (XML path) je poizvedbeni jezik za izbiranje vozlišč XML dokumenta [4]. Ker je HTML lahko implementiran kot XML (XHTML), lahko uporabniki orodja Selenium Webdriver ta močan jezik uporabijo za dostop do elementov spletne aplikacije, ki se testira. V naslednjih poglavjih bomo videli, da je izbirnik XPath pri testiranju dinamičnih aplikacij zaradi samodejno generiranih identifikatorjev zelo uporaben, po drugi strani pa je zelo občutljiv na spremembe v XHTML kodi in ob nepremišljeni uporabi lahko povzroča veliko težav.

Lep primer dobre prakse, pri katerem je potrebna uporaba XPath izbirnika, je enostaven tabelaričen prikaz, ki vsebuje podatke, ki jih dobimo z SQL poizvedbo. Takrat navadno ni znano v naprej, koliko vrstic bo vrnila SQL poizvedba. Za dostop do določene vrednosti v tabeli z izbirnikom XPath zadostuje že, da nam je znano ime tabele znotraj XHTML (ali kateri izmed unikatnih atributov). Do celic potem dostopamo enostavno preko indeksov (številka vrstice, številka stolpca). Na primeru izvorne kode 3.4 si lahko ogledamo izraz za dostop do celice v prvi vrstici in petem stolpcu tabele `foo`.

```
//table[@id='foo']/tr[1]/td[5]
```

Izvorna koda 3.4: Primer zapisa izbirnika XPath za dostop do prve vrstice in petega stolpca tabele `foo`.

Prednost izbirnika XPath so tudi vgrajene funkcije, ki nam dostop do elementov še dodatno olajšajo. Nekaj najbolj uporabnih vgrajenih funkcij si bralec lahko ogleda v tabeli 3.2.

Funkcija	Opis
Last()	Vrne zadnji element v množici elementov, ki jo vrne izbirnik.
Contains()	Preverja ali niz vsebuje podan podniz.
Matches()	Preverja ali se niz ujema s podanim regularnim izrazom.
Text()	Vrne tekst, ki se nahaja znotraj izbranega elementa.
StripSpace()	Odstrani odvečne presledke.

Tabela 3.2: Seznam funkcij XPath, ki se pri testiranju dinamičnih spletnih aplikacij izkažejo kot zelo uporabne.

Primer dostopa do našega gumba z izbirnikom XPath in ukaz za klik nanj je prikazan z izvorno kodo 3.5.

```
driver.findElement(By.xpath("//input[@id='prijava']")).click();
```

Izvorna koda 3.5: Izbira spletnega elementa s pomočjo izbirnika XPath in klika na izbran element.

Kritični dejavniki in največja slabost izbirnika XPath so različne implementacije v brskalnikih. Zaradi velikih razlik med njimi, lahko enak izbirnik na prvem brskalniku deluje brez težav, medtem ko ga kateri drugi brskalnik sploh ne prepozna. Še večje težave se pojavijo pri brskalniku Internet Explorer, ki do različice 9 nima domorodne podpore XPath. Zato so bili razvijalci orodja Selenium WebDriver primorani uporabiti rešitev z uporabo posebne knjižnice JavaScript, kar pa se pozna na zmogljivosti.

3.4.4 Ostali podprti izbirniki

Zgoraj smo omenili tri različne načine izbiranja spletnih elementov, ki so najbolj pogostokrat uporabljeni. Na razpolago imamo še nekaj drugih načinov:

- izbiranje po razredu (preverja se atribut `class`),
- izbiranje po znački (angl. tag),
- izbiranje po tekstu povezave in po delnem tekstu povezave.

Zaradi njihove omejene uporabnosti pri testiranju kompleksnejših dinamičnih aplikacij, se ne bomo spuščali v detajle. Izbiranje po tekstu povezave - HTML značka `a`, ali delnemu tekstu povezave je smiselno iz vidika enostavnosti. Iz kode je takoj vidno, na katero povezavo se sklicujemo. Slaba stran je, da deluje le na povezavah. Izbiranje po imenu razreda deluje tako, da se preverja atribut `class`. Tako kot tudi izbiranje po imenu značke pride v poštev predvsem takrat, ko želimo elemente prešteti. Žal je običajno pri kompleksnejših aplikacijah elementov, ki ustrezajo znački ali nekemu razredu, enostavno preveč.

3.5 Zagotavljanje preprostosti in preglednosti kode

Dinamičnim spletnim aplikacijam se pogosto spreminja uporabniški vmesnik. To predstavlja težavo pri dostopu do elementov na spletni strani. Programerju ne preostane nobenih drugih možnosti, kot da popravi kodo testnih primerov, kar pa je lahko zelo zamudno. Potrebno je

zagotoviti čim lažje uvajanje sprememb. Istočasno je potrebno kodo ohraniti pregledno in logično. V ta namen se tudi pri razvoju avtomatskih testov uporablja tako imenovane načrtovalske vzorce (angl. design patterns). Daleč najbolj uporaben v primeru spletnega testiranja je načrtovalski vzorec PageObjects.

3.5.1 Načrtovalski vzorec PageObjects

Z uporabo modela PageObject lahko ustvarimo manj krhko kodo in zmanjšamo ali odpravimo ponavljanje testne kode [6]. Ideja pristopa PageObjects je, da vsako stran in vsak element na njej predstavimo kot objekt. Ko se premikamo med stranmi, se sklicujemo na njim pripadajoče objekte v kodi. Ti objekti imajo implementirane metode, preko katerih izvajamo željene akcije. Po uspešni akciji metoda vrne nov objekt. Lep primer ja prijavna stran iz primera 3.6. Sklicujemo se na objekt tipa `LoginPage`, ki predstavlja prijavno stran. Metoda vpiše prijavne podatke in potrdi formo nato pa ob uspešni prijavi vrne nov objekt `WelcomePage`:

```
LoginPage lp = new LoginPage();
WelcomePage wp = lp.login("admin", "password");
```

Izvorna koda 3.6: Prikaz inicializacije objektov pri uporabi PageObjects načrtovalskega vzorca.

Vzorec PageObjects v prvi vrsti zagotavlja večjo preglednost kode, ki jo je posledično veliko lažje urejati. Druga dobra lastnost tega pristopa je, da se izognemo podvajanju kode. Tako je ob spremembi uporabniškega vmesnika lažje popraviti teste, saj so spremembe potrebne le na enem mestu. Zaradi priljubljenosti tega načrtovalskega vzorca so razvijalci orodja Selenium WebDriver implementirali še razred `PageFactory`, ki nam še dodatno olajša delo.

3.5.2 Razred PageFactory

Omenjeni razred je razširitev orodja Selenium WebDriver, ki omogoči lažje delo pri uporabi načrtovalskega vzorca PageObjects. Uporablja se za iskanje in samodejno inicializacijo elementov objekta stran, ali celo objekta stran samega [14].

Razred `PageFactory` lahko uporabimo le, ko pišemo teste v programskem jeziku Java. Koda je z njegovo uporabo bolj berljiva in je lažje z njo opravljati, saj se znebimo klicev `findElement()` za vsak spletni element. Najdemo in inicializiramo ga kar z anotacijo `@FindBy`, kot je prikazano z izvorno kodo 3.7.

```
@FindBy(how = How.ID, using = "foo")
WebElement foo;
```

Izvorna koda 3.7: Primer anotacije pri uporabi razreda `PageFactory`.

Kljub popolnoma pravilni inicializaciji in pravilnemu sklicevanju na elemente pa se pri dinamičnih aplikacijah pogosto zgodi, da element vseeno ni bil najden. Problem se pojavi zaradi klicev Ajax, ki spremenijo DOM (dokumentni objetni model) brez ponovnega nalaganja strani. Kako se lahko temu izognemo, si bomo ogledali v naslednjem podpoglavju.

3.6 Selenium WebDriver in Ajax klici

Ajax (asinhroni JavaScript in XML) je skupina medsebojno povezanih spletnih razvojnih tehnik, uporabljenih za ustvarjanje interaktivnih spletnih aplikacij. S tehnologijo Ajax si lahko spletne aplikacije izmenjujejo podatke s strežnikom asinhrono v ozadju, brez potrebe po ponovnem nalaganju strani [17]. To je tudi razlog, da je včasih sklicevanje na nek spletni element neuspešno. Do problema pride, ker Selenium WebDriver ob izvedbi akcije, ki sproži Ajax klic (običajno traja nek določen čas), ne ve, kdaj je klic končan. Tako se lahko zahteva dostop do elementa, ki v tem času še ni prisoten v DOM. Problemu bi se lahko izognili s preprostim klicem `thread.sleep()`, vendar to ni optimalno, ker tako na element pogosto čakamo dlje, kot bi lahko.

Zato Selenium WebDriver nudi dva mehanizma, ki vnaprej določen čas konstantno preverjata, kdaj bo element na voljo. To sta implicitno čakanje (angl. implicit wait) in eksplicitno čakanje na elemente (angl. explicit wait).

Oba načina opravljata enako funkcijo – generirata `NoSuchElementException` šele, če je element nedosegljiv dlje od nekega v naprej določenega časa.

3.6.1 Implicitno čakanje

Implicitno čakanje inicializiramo na začetku razreda (ko kreiramo object `Driver`) in velja skozi celotno življensko dobo objekta `Driver` [1]. Izvorna koda 3.8 prikazuje inicializacijo s preprostim klicem metode `implicitlyWait()`, ki ji podamo parameter, ki pove koliko časa naj gonilnik čaka na element preden generira izjemo tipa `NoSuchElementException`.

```
driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
```

Izvorna koda 3.8: Inicializacija implicitnega čakanja.

Ta način čakanja na elemente je precej neprilagodljiv. Obnaša se enako za vse elemente/akcije, do katerih dostopamo z gonilnikom, na katerem smo ga inicializirali. Izjemo generira v vsakem primeru, ne glede na razlog, zakaj interakcija z elementom ni možna. Zato so v Selenium WebDriver vgradili tudi mehanizem za ti. eksplicitno čakanje.

3.6.2 Eksplicitno čakanje

Eksplicitno čakanje omogoča različno delovanje za različne elemente ali akcije. Preden dostopamo do nekega elementa, lahko s pomočjo razreda ExpectedCondition natančno povemo, pod katerim pogojem naj se sproži izjema – na primer, ko bo na element možno klikniti (`elementToBeClickable()`), ko bo element izbran (`elementToBeSelected()`), ko bo prisoten poljuben tekst (`textToBePresent()`),...

Primer izvorne kode 3.9 prikazuje, kako zagotovimo eksplicitno čakanje (največ 30 sekund) na element z identifikatorjem `elementId`, dokler nanj ni možno klikniti.

```
public String waitForElement(String elementId) {
    WebDriverWait wait = new WebDriverWait(driver,30);
    WebElement element = wait.until(

    ExpectedConditions.elementToBeClickable(By.id(elementId)));
    return elementId;
}
```

Izvorna koda 3.9: Inicializacija eksplecitnega čakanja.

3.7 Selenium WebDriver in ogrodje JUnit

Prikaz dokončnih rezultatov je zelo pomemben del testiranja, saj dobro poročilo olajša komunikacijo med testerji in razvijalci. Pogostokrat celo rečemo, da končno poročilo testiranja odseva trenutno stanje testirane programske opreme.

Rezultate bi lahko preverjali programsko s primerjalnimi operatorji in enostavnimi stavki `IF`. Toda glede na močna zastonjska ogrodja, ki jih imamo danes na razpolago, bi bilo to

nesmiselno. Ogledali si bomo, kako lahko priljubljeno ogrodje JUnit, ki se v osnovi uporablja za implementacijo testov enot, združimo z orodjem Selenium WebDriver.
Z združitvijo pridobimo:

- preglednejšo kodo testnih primerov z uporabo anotacij,
- enostavno primerjanje rezultatov z uporabo funkcij `assert` ogrodja JUnit,
- podporo v drugih aplikacijah, ki podpirajo zaganjalnik (angl. runner) JUnit,
- možnost parametrizacije testnih primerov.

3.7.1 Uporaba anotacij

Ogrodje JUnit od različice 4 dalje uporablja anotacije, s katerimi programer lahko določi, katere metode naj se izvajajo pred ali po testih, katere predstavljajo testne primere in katere teste želimo preskočiti.

Anotacijo `@BeforeClass` uporabimo za inicializacijo Selenium WebDriverja, saj se z njo označena metoda kliče vedno, preden se začnejo izvajati metode označene z anotacijo `@Test`. V kolikor je kateri od testnih primerov še nedokončan, ga lahko enostavno preskočimo tako, da metodo označimo z anotacijo `@Ignore`. Po izvedenih testih je potrebno zapreti brskalnik in počistiti morebitne pomožne podatke, ki so se generirali med testi. Metodo, ki to opravi, označimo z anotacijo `@AfterClass`.

Tako dobimo predlogo razreda za zaganjanje Selenium testov, ki je prikazana kot izvorna koda 3.10.

```
import org.junit.*;

public class Test_junit {

    @BeforeClass
    public static void setUp() throws Exception {
    // Nastavitve za gonilnik brskalnika in ostale nastavitve
    }

    @Test
    public void test_1 () {
       // Koda testnega primera 1
    }

    @Test
    public void test_2() {
        // Koda testnega primera 2
    }
```

```
    @AfterClass
    public static void stopSelenium() throws Exception {
        // Zaustavitev gonilnika brskalnika
    }
}
```

Izvorna koda 3.10: Predloga za implementacijo testnih primerov Selenium Webdriver v kombinaciji z ogrodjem JUnit.

3.7.2 Preverjanje rezultatov

Za primerjanje rezultatov imamo možnost uporabiti vse metode `assert`, s katerimi razpolaga ogrodje JUnit. Torej: `assertEquals()`, `assertTrue()`, `assertFalse()`, `assertNotNull()`, `assertNull()`, `assertSame()`, `assertNotSame()` in `assertArrayEquals()`.

Zaradi kritičnih dejavnikov uspeha je priporočljivo, da se metodam poleg dejanske in pričakovane vrednosti posreduje še krajši opis napake, zaradi katere je test spodletel. Taka primera stavkov prikazuje izvorna koda 3.11.

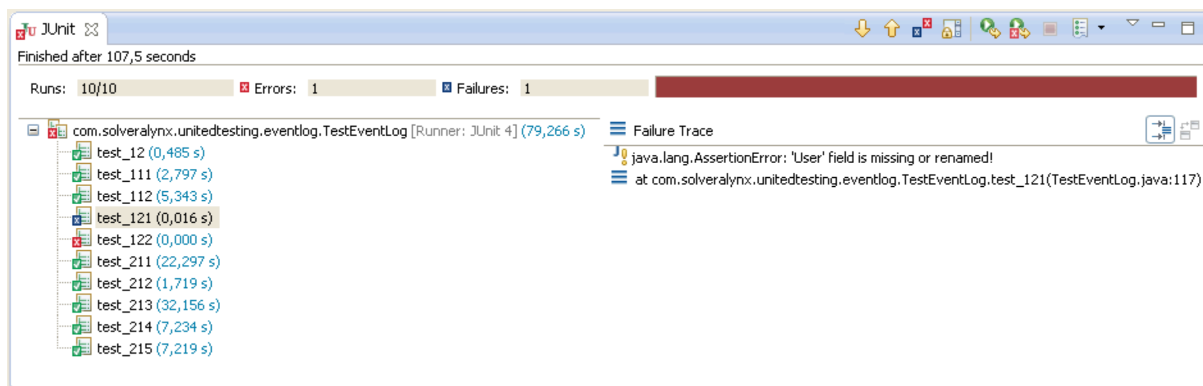
```
assertTrue("Nepričakovano število vrstic tabele! ",
driver.findElements(by.xpath("//table/tr")).size() == 5);

assertEquals("Nepričakovan naslov prijavne strani! ", "Prijava",
driver.getTitle());
```

Izvorna koda 3.11: Primer uporabe funkcij `assert` v kodi.

3.7.3 Podpora v ostalih aplikacijah

Priljubljenost ogrodja JUnit je razlog, da so možnosti kombiniranja z drugimi aplikacijami skoraj neskončne. V tem poglavju bomo omenili tiste, ki so za nas pomembne iz vidika avtomatizacije Selenium WebDriver testov.

Danes imajo že vsa resnejša razvijalska orodja ogrodje JUnit, že integrirano v samem okolju. Na sliki 3.2 je prikazan primer izpisa rezultatov po zagonu Selenium WebDriver testov v orodju Eclipse IDE, ki ga dosežemo brez posebnih dodatkov razvojnemu okolju.



Slika 3.2: Primer izpisa rezultatov »Webdriver + JUnit« testov v razvojnem okolju Eclipse IDE.

Samo dobra integracija v razvojno okolje nam omogoča učinkovito razhroščevanje testnih primerov in lažje dodajanje novih. Poleg tega je uporabna tudi, ko želi tester ob svoji prisotnosti zagnati avtomatske teste ali le del teh. Primer takega prijema si bomo ogledali v poglavju *Testiranje predizdaje*.

JUnit je v obliki opravila podprt tudi v orodju Apache ANT, kateri je nepogrešljiv pri avtomatizaciji. Podroben opis orodja ANT bi presegal okvirje te diplomske naloge, zato bomo predpostavili, da bralec že pozna principe delovanja tega orodja.

ANT skripta nam omogoča izvajanje testov JUnit (in s tem Selenium WebDriver) testov na strežniku za sprotno integracijo, kot sta na primer Bamboo in Hudson. Pogoj je, da strežnik kot vhodni podatek dobi datoteko, v kateri so rezultati izvajanja testov zapisani v dokumentu XML. To omogočimo tako, da ANT opravilu `junit` določimo lastnost `formatter type`, kot je vidno v primeru kode 3.12.

```
<junit>
    …
    <formatter type="xml"/>
    …
</junit>
```

Izvorna koda 3.12: Del ANT kode za izbiro tipa izhoda JUnit rezultatov.

Vse, kar je kasneje potrebno storiti v aplikaciji za sprotno integracijo, je, da ji pokažemo pot do teh datotek XML. Aplikacija sama poskrbi za uporabniku bolj prijazen izpis rezultatov, kot ga vidimo na sliki 3.3.

Summary Tests Changes Artifacts Logs Metadata Issues Actions

Build Result Summary

Details

Trigger This build was manually triggered by Dejan Veternik.
Revision 24091
Completed 22 apr 2014, 11:46:57 PM – 14 seconds ago
Duration 11 minutes
Failing since #902 (Scheduled build – 1 week before)
Labels None

Code Changes

No changes found for this Build.

Comments

No comments on this Build.

Comment

Test Summary 31 tests in total

6 New Failures
9 Existing Failures
0 Fixed

Tests

New Test Failures (6) Expand All Collapse All

Test	View Job	Duration
TestBasicReport test_232	Default Job	13 secs
TestBasicReport test_214	Default Job	29 secs
TestBasicReport test_211	Default Job	1 min

```
junit.framework.AssertionFailedError: Testing table sorting failed!
        at com.solveralynx.unitedtesting.basicreport.TestBasicReport.test_232(TestBasicReport.java:579)

junit.framework.AssertionFailedError: Values in the last column should be greater than those in the first column!
        at com.solveralynx.unitedtesting.basicreport.TestBasicReport.test_214(TestBasicReport.java:473)

junit.framework.AssertionFailedError: 'Grafični prikaz' window is missing!
        at com.solveralynx.unitedtesting.basicreport.TestBasicReport.test_211(TestBasicReport.java:310)
```

Slika 3.3: Aplikacije za sprotno integracijo omogočajo zelo pregledno predstavitev rezultatov.

3.7.4 Možnost parametriziranih testov

Ogrodje JUnit od različice 4 naprej ponuja tudi možnost ti. parametriziranih testov. Omenjamo jo zato, ker se izkaže za zelo uporabno tudi pri testiranju spletnega uporabniškega vmesnika. Parametrizirani testi namreč omogočajo razvijalcu, da se isti test ponovi večkrat z različnimi vrednostmi [15]. Zato jih lahko uporabimo za preverjanje vrednosti tabelaričnih prikazov, visečih menijev, itd. Konec koncev jih lahko uporabimo tudi za preverjanje rezultatov aritmetičnih operacij na različnih podatkih, v primeru, da jih spletna aplikacija izračunava.

3.8 Testiranje na različnih brskalnikih

Programer testnih primerov testni brskalnik izbere že takoj na začetku kode z definicijo objekta tipa WebDriver. To pomeni, da med izvajanjem testov ni mogoče spremeniti brskalnika v katerem se izvaja testiranje. Orodje Selenium WebDriver že samo po sebi podpira izvajanje testov z več različnimi brskalniki, njihov nabor pa se močno razširi, če upoštevamo še

gonilnike, ki so delo tretjih oseb in ne razvijalcev orodja Selenium WebDriver. Poleg več brskalnikov in različnih platform za osebne računalnike, je mogoče izvajanje testov tudi na vseh popularnejših mobilnih sistemih, kot so Android, Windows Phone in iOS. V nadaljevanju bomo opisali samo gonilnike za najpomembnejše brskalnike, ki so našteti v tabeli 3.3.

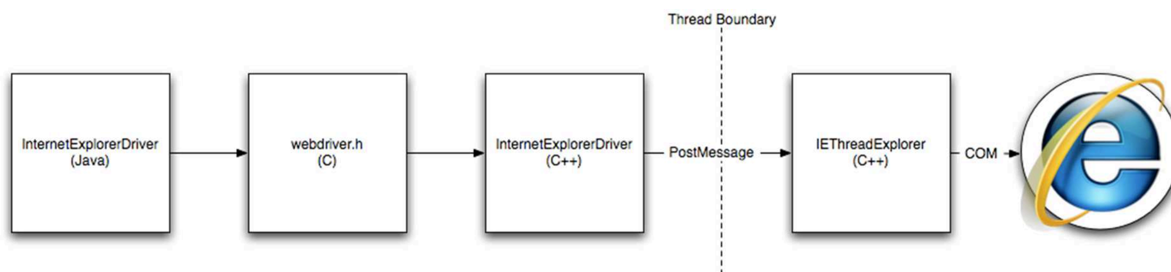
Brskalnik	Razred 'WebDriver'
Google Chrome	ChromeDriver
Microsoft Internet Explorer	InternetExplorerDriver
Mozilla Firefox	FirefoxDriver
HtmlUnit	HtmlUnitDriver
*	RemoteWebDriver

Tabela 3.3: Seznam najbolj pogosto uporabljenih brskalnikov. HtmlUnit in RemoteWebDriver sta označena drugače, ker nista gonilnika »pravih« brskalnikov.

O izboru gonilnika, ki se bo uporabljal za testiranje, odloča programer sam. Navadno se uporabi gonilnik za brskalnik, ki je predpisan v sistemskih zahtevah za uporabo aplikacije. V naslednjih poglavjih si bomo podrobneje ogledali gonilnike za najbolj pogosto uporabljene brskalnike.

3.8.1 InternetExplorerDriver

Spletni brskalnik Internet Explorer zaradi številnih različic in implementacij podpore JavaScript in CSS pri razvijalcih spletnih aplikacij velja kot najbolj problematičen brskalnik. Ni redkost, da se testira samo v tem brskalniku Internet Explorerju. Namreč, zelo velika verjetnost je, da v kolikor aplikacija brezhibno deluje s tem brskalnikom, potem deluje tudi z ostalimi. Slabi lastnosti tega gonilnika sta njegova počasnost in pa seveda omejenost na operacijski sistem Windows. V starejših verzijah orodja Selenium je bilo za izbiro spletnih elementov in interakcijo z njimi potrebno injecirati ti. Sizzle – JS knjižnico za izbiranje spletnih elementov. To je močno vplivalo na hitrost izvajanja testnih primerov. Danes gonilnik brskalnika z Internet Explorerjem komunicira preko dodatne aplikacije imenovane IEDriverServer. IEDriverServer s pomočjo tehnologije COM brskalniku pošilja ukaze v obliki sistemskih domorodnih dogodkov, kot je prikazano na sliki 3.4.



Slika 3.4: Potek interakcije med gonilnikom InternetExplorerDriver in brskalnikom [16].

Tak pristop ima dve pomembni prednosti. Prva je izogib počasni knjižnici JavaScript, druga pa interakcija s spletno aplikacijo s popolnoma enakimi dogodki, kot jih generira tudi sam uporabnik – npr. klik na levi miškin gumb. Z drugimi besedami povedano: spletna aplikacija ne loči, kdaj klika uporabnik in kdaj avtomatski test.

Iz previdnosti se priporoča, da uporabnik med izvajanjem testnih primerov s tem gonilnikom ne uporablja delovne postaje. Saj lahko povzroči, da aktivni brskalnik izgubi fokus. Posledica je, da se zaradi sistemskih domorodnih dogodkov brskalnik ta začne obnašati zelo nepredvidljivo.

Izvorna koda 3.13 prikazuje inicializacijo gonilnika InternetExplorerDriver.

```
System.setProperty("webdriver.ie.driver","D:/iexploredriver.exe");
WebDriver driver = new InternetExplorerDriver();
```

Izvorna koda 3.13: Inicializacija gonilnika InternetExplorerDriver z nastavitvijo poti do iexplorerdriver.exe aplikacije.

3.8.2 ChromeDriver

Glede na mnenja raziskav, ki jih je moč najti v spletu, je Google Chrome brskalnik, ki mu v času pisanja te diplomske naloge najhitreje raste priljubljenost med uporabniki [20]. S tem tudi gonilniku ChromeDriver raste pomembnost. Enako kot InternetExplorerDriver, tudi ChromeDriver za komunikacijo z brskalnikom potrebuje dodatno izvršljivo aplikacijo imenovano `chromedriver.exe`, ki deluje kot most med gonilnikom brskalnika in brskalnikom samim. To aplikacijo razvijajo kar Googlovi razvijalci iz projekta Chromium, kamor spada tudi sam brskalnik Chrome. Posledica tega je zelo dobra podpora in odlične zmogljivosti pri izvajanju testnih primerov.

Gonilnik ChromeDriver ima v primerjavi z ostalimi gonilniki še eno prednost - možnost uporabe na mobilnih napravah s sistemom Android. To nam omogoča testiranje spletne strani ali aplikacije tudi v mobilnem načinu prikaza.

Izvorna koda 3.14. prikazuje inicializacijo gonilnika ChromeDriver s pomočjo podporne izvršljive datoteke `chromedriver.exe`.

```
System.setProperty("webdriver.chrome.driver",
"D:/chromedriver.exe");
WebDriver driver = new ChromeDriver();
```

Izvorna koda 3.14: Inicializacija gonilnika ChromeDriver z nastavitvijo poti do datoteke `chromedriver.exe`.

3.8.3 FirefoxDriver

Gonilnik za brskalnik FireFox je implementiran v obliki Firefox razširitve (angl. Firefox extension), zato zanj ne potrebujemo dodatne izvršljive aplikacije. Poleg hitrosti, je njegova prednost možnost upravljanja z uporabniškim profilom. To nam omogoča veliko večjo kontrolo nad brskalnikom. Pridobimo številne nove možnosti, kot sta uporaba sistemskih domorodnih dogodkov in upravljanje s sistemskimi dialogi, kot je na primer dialog za prenos datotek. Opravljanje s slednjim je prikazano v primeru izvorne kode 3.15, ki prikazuje, kako nastavimo ciljno pot prenosov in samodejno sprožitev prenašanja datotek PDF. Objekt tipa `FirefoxProfile` podamo kot parameter ob inicializaciji samega gonilnika.

```
FirefoxProfile profile = new FirefoxProfile();

profile.setPreference("browser.download.dir",
"/home/dejanv/downloads");

profile.setPreference("browser.download.folderList", 2);

profile.setPreference("browser.helperApps.neverAsk.saveToDisk",
"application/pdf");

WebDriver driver = new FirefoxDriver(profile);
```

Izvorna koda 3.15: Konfiguracija profila za samodejen prenos datotek tipa PDF in inicializacija gonilnika FirefoxDriver.

Brskalnik Firefox nam v sklopu svojega dovršenega sistema razširitev, v največji meri ponuja tudi mnogo uporabnih razvijalskih orodij. Naj omenimo orodje Firebug in njegov dodatek FirePath, brez katerih si težko predstavljamo tvorjenje izrazov XPath in CSS. Poleg prikaza trenutne kode strani in možnosti spreminjanja le-te v realnem času, pridobimo tudi močan preverjalnik izrazov in grafični prikaz spletnih elementov, na katere se izraz (izbirnik) nanaša.

Slika 3.5: Uporabniški vmesnik dodatkov Firebug in FirePath.

3.8.4 HtmlUnitDriver

Ta gonilnik se nekoliko razlikuje od vseh zgoraj omenjenih. HtmlUnitDriver je edini gonilnik, ki za svoje delovanje ne uporablja »pravega« brskalnika, ampak brskalnik HtmlUnit. Glavna posebnost brskalnika HtmlUnit je, da je v celoti implementiran v programskem jeziku Java in

je brez uporabniškega vmesnika. Brskalnik HtmlUnit modelira HTML dokumente in nudi API preko katerega odpiramo strani, izpolnjujemo obrazce, klikamo povezave, itd... [18] Gonilnik je zaradi omenjene zasnove najhitrejši in je uporaben pri testiranju funkcionalnosti pri katerih ni pomemben tip brskalnika ampak hitrost izvedbe testov. Omogoča tudi posnemanje delovanja drugih brskalnikov.

Inicializiramo ga po standardni metodi, konstruktorju HtmlUnitDriver pa po potrebi posredujemo parameter za omogočanje JavaScript podpore oziroma emulacije drugih brskalnikov, kot je prikazano z izvorno kodo 3.16.

```
WebDriver driver = new
HtmlUnitDriver(BrowserVersion.INTERNET_EXPLORER_11);
```

Izvorna koda 3.16: Inicializacija gonilnika HtmlUnitDriver za brskalnik HtmlUnit z emulacijo Internet Explorerja različice 11.

3.8.5 RemoteWebDriver

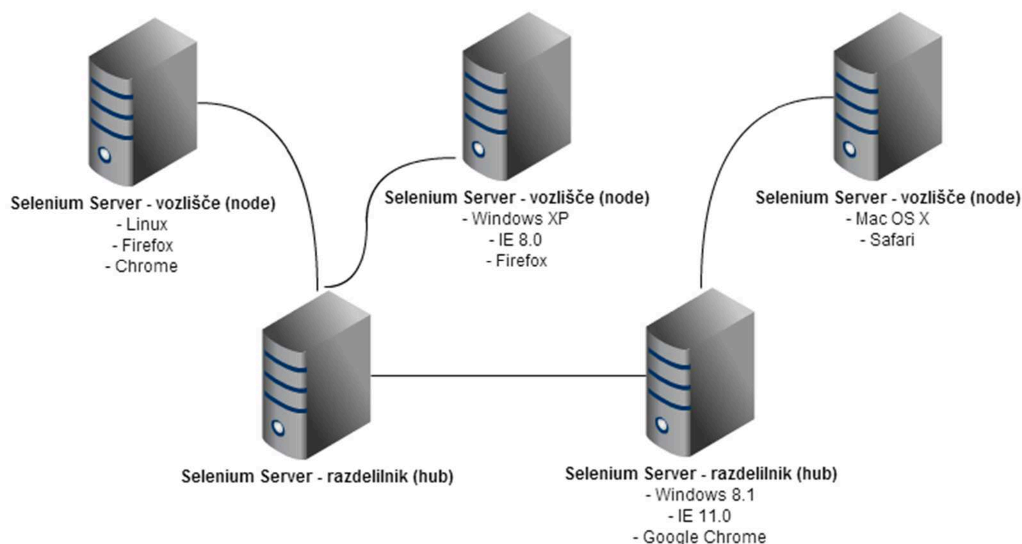
Kot smo na kratko že omenili na začetku poglavja o podprtih brskalnikih, je RemoteWebDriver poseben gonilnik, katerega namen je podpora porazdeljenemu testiranju. Ideja je, da različne platforme z različnimi konfiguracijami povežemo v mrežo. Eno ali več vozlišč v tej mreži ima vlogo razdelilnika (angl. hub), na katerega se sklicujemo iz naše kode. Pred klicem s pomočjo razreda `DesiredCapabilities` najprej določimo naše zahteve, kot sta tip platforme in tip brskalnika. Kot vidimo iz izvorne kode 3.15, sami inicializaciji gonilnika kot parameter podamo tudi naslov strežnika in številko vrat, kjer se nahaja razdelilnik. Razdelilnik nato samodejno aktivira vozlišče, ki popolnoma ustreza željenim zahtevam.

```
DesiredCapabilities caps = DesiredCapabilities.firefox();
capability.setPlatform(Platform.LINUX);
WebDriver driver = new RemoteWebDriver(new
URL("http://server:30035/wd/hub"), caps);
```

Izvorna koda 3.15: Inicializacija gonilnika RemoteWebDriver, ko želimo teste zaganjati na brskalniku Firefox in platformi Linux. Potrebno je podati tudi naslov in številko vrat strežnika, kjer se nahaja Selenium Server v vlogi razdelilnika.

Vsi ukazi brskalniku se preusmerijo na to vozlišče, kjer se po zagonu brskalnika prične izvajanje testov. Za delovanje na vsakem vozlišču potrebujemo nameščeno in zagnano aplikacijo Selenium Server, ki ji ob zagonu določimo vlogo (razdelilnik ali navadno vozlišče), platformo in kateri brskalniki so na razpolago na trenutnem vozlišču. Vsako vozlišče nato še registriramo na poljuben razdelilnik, tako da Selenium Server poženemo z naslednjimi parametri:

```
java -jar selenium-server.jar -role node  -hub
http://url_razdelilnika:vrata/grid/register
```



Slika 3.8: Primer porazdelitve testnih postaj z različnimi testnimi okolji.

Naj omenimo še, da v primeru številčnejših nastavitev lahko ustvarimo tekstovno datoteko, kjer te nastavitve zapišemo v formatu JSON.

Poglavje 4 Sistem za testiranje funkcionalnosti aplikacije GemaLogic

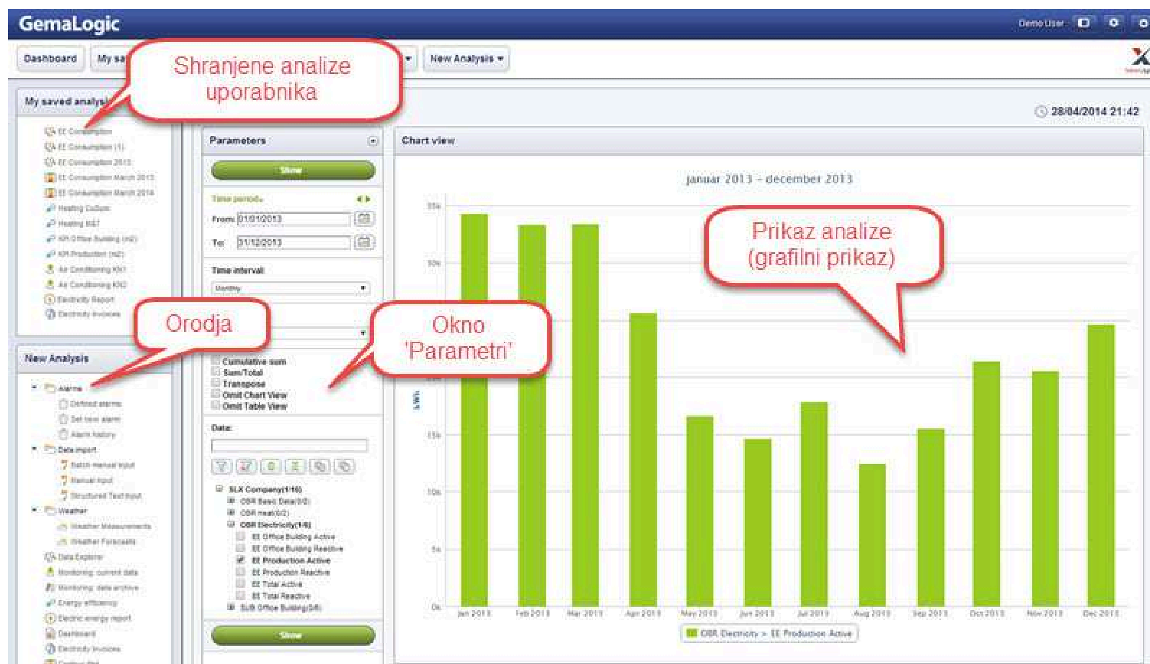
Kar smo teoretično povzeli v prejšnjih poglavjih, bomo v tem poglavju opisali na praktičnem primeru aplikacije GemaLogic. Podrobneje si bomo ogledali implementacijo avtomatskih testov »Splošni pregled«, ki je eno izmed glavnih orodij omenjene aplikacije. Prikazali bomo način za samodejno pripravo in testiranje nočne gradnje in potek testiranja predizdaje.

Za boljše razumevanje pristopa si bomo najprej na kratko ogledali aplikacijo GemaLogic in časovnico razvoja te aplikacije po metodologiji Scrum.

4.1 O aplikaciji GemaLogic

S strani porabnikov energije, se kot ena od posledic odpiranja trga z energijo, dobaviteljev energije ter izvajalcev ostalih energetskih storitev, pojavljajo vedno večje zahteve po izboljšanem nadzoru rabe energentov in zato potreba po ažurnih podatkih o njihovi porabi. Praktično uporaben odgovor na to je učinkovit informacijski sistem za podporo energetskemu managementu električne energije, zemeljskega plina, vode, toplote ter komprimiranega zraka... GemaLogic je podatkovni in aplikativni strežnik, ki predstavlja jedro informacijskega sistema za energetski management. Strežnik zbira podatke z obstoječih merilnih mest in obstoječih informacijskih sistemov in jih pretvarja v uporabne informacije za spremljanje porabe in energentov.

Spletni uporabniki do teh informacij dostopajo in jih pregledujejo preko spletnega brskalnika. Po uspešni prijavi v aplikacijo imajo možnost izbrati eno izmed več orodij za prikaz različnih poročil. Parametre poročila, kot so časovno obdobje in željeni podatki uporabnik lahko izbere v posebnem oknu imenovanem 'Parametri'. Takšno poročilo imenujemo analiza. Vsak uporabnik lahko shrani množico analiz, da mu ni potrebno vedno znova nastavljati parametrov.

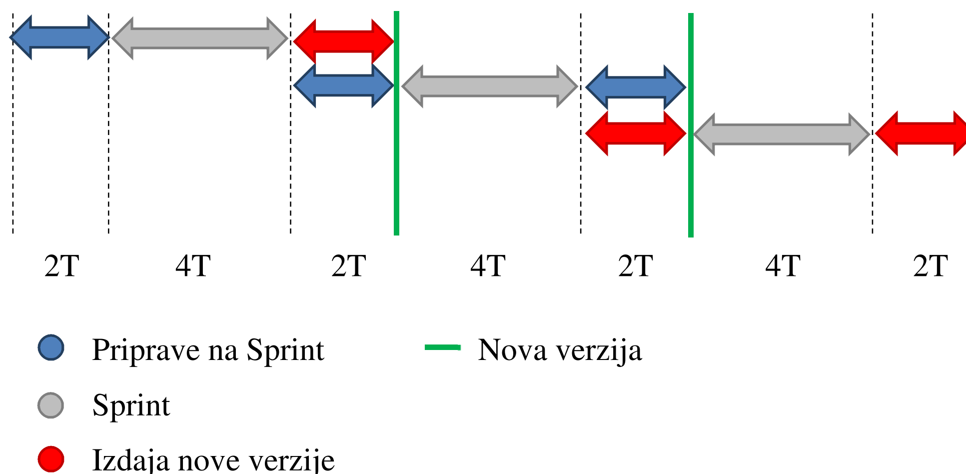


Slika 4.1: Uporabniški vmesnik aplikacije GemaLogic.

Na sliki 4.1 je prikazan uporabniški vmesnik aplikacije Gemalogic. Iz tehničnega vidika, je aplikacija napisana v programskem jeziku Java in teče znotraj aplikacijskega strežnika JBoss. Vsa koda se nahaja v shrambi Subversion. Gradnja nove verzije se izvede na strežniku za sprotno integracijo, kjer teče programska oprema Bamboo podjetja Atlassian.

4.1.1 Časovnica razvoja GemaLogic po metodologiji Scrum

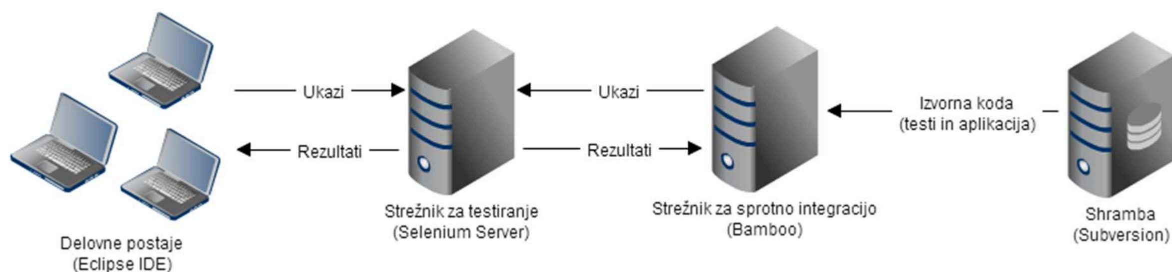
GemaLogic Sprint traja 4 tedne, 2 tedna pa istočasno tečejo priprave na naslednji Sprint in izdaja nove verzije. Torej stabilno aplikacijo nameščamo v produkcijo vsakih 6 tednov. Časovnica je grafično predstavljena na sliki 4.2. Izdaja nove verzije zavzema testiranje predizdaje in popravilo hroščev, ki so bili najdeni pri tem testiranju. Vsaka verzija, ki je kot predizdaja zavrnjena, se označi kot neprimerna za produkcijsko namestitev. Popravki se implementirajo v novo vejo (angl. branch) iz katere se zgradi nova predizdaja.



Slika 4.2: Primer časovnice iteracij z dolžino Sprinta 4 tedne.

4.2 Testno okolje

Za učinkovito zagotovitev samodejnega izvajanja testov, tako za nočno gradnjo, kot tudi za predizdajo, smo delovno postajo, strežnik za izvajanje testov, strežnik za sprotno integracijo in strežnik s shrambo kode povezali v celovit sistem za testiranje, kot je prikazan na sliki 4.3.



Slika 4.3: Skica testnega okolja za avtomatsko testiranje aplikacije GemaLogic.

V naslednjih poglavjih si bomo podrobneje ogledali naloge vseh komponent testnega sistema.

4.2.1 Delovna postaja

Funkcionalnost delovnih postaj vključuje zmogljivost, nemoteno delovanje in enostavno vzdrževanje. Na delovni postaji je naloženo okolje za razvoj, avtomatizacijo in popravljanje testnih primerov. S samostojnim odjemalcem ali z razširitvijo za razvojno okolje, kodo testnih primerov shranjujemo v shrambo – v našem primeru Subversion.

Določen scenarij na delovni postaji lahko ponovimo in ga za boljšo predstavo delimo z razvijalcem.

4.2.2 Strežnik za izvajanje testnih primerov

V testne namene se Aplikacija GemaLogic splavi na pred-nastavljen aplikacijski strežnik JBoss, ki je nameščen kot servis. To nam omogoča, da aplikacija teče v ozadju brez aktivne uporabniške seje. Prav tako se aplikacijski strežnik brez težav samodejno zažene v kolikor pride do nepričakovanega ponovnega zagona sistema.
Druga pomembna aplikacija, ki teče na tem strežniku je Selenium Server. Teče v vlogi navadnega vozlišča. Tudi ta zaradi prej omenjenih razlogov teče kot servis.

Na strežniku za izvajanje testnih primerov so prisotni tudi zelo uporabni dnevniki, ki jih piše aplikacija GemaLogic med testiranjem. V večini primerov so sledi sklada (angl. stacktrace), ki se zapišejo v dnevnike ob napaki, pomemben del poročila, ki ga tester posreduje razvijalcu. Zaradi lažjega in hitrejšega dostopa do dnevnikov iz delovne postaje smo na direktoriju, kjer se nahajajo dnevniki, omogočili skupno rabo.

4.2.3 Strežnik za sprotno integracijo

Nočna gradnja in njeno testiranje navadno potekata brez prisotnosti človeka. Zato potrebujemo mehanizem, ki samodejno:

- pripravi nočno gradnjo ob v naprej določenem času,
- jo splavi na strežnik za testiranje,
- poskrbi za samodejno izvajanje testov,
- shrani rezultate in jih po možnosti posreduje odgovorni osebi.

Kako je to moč izvesti, s pomočjo strežnika za sprotno integracijo Bamboo, si bomo podrobneje ogledali v poglavju *Nočna gradnja in splavitev*.

4.2.4 Shramba Subversion

Celotna izvorna koda aplikacije GemaLogic se skupaj z zgodovino in dnevniki sprememb nahaja na samostojnem strežniku v shrambi Subversion. Strežnik za sprotno integracijo pred vsako gradnjo iz te shrambe samodejno prenese kodo aplikacije in testnih primerov.

4.3 Avtomatizacija testnih primerov

Projekt avtomatizacije testov mora potekati točno tako, kot drugi projekti razvoja programske opreme [19]. Ko smo prepričani, da so testni primeri dobro izbrani, pravilni in čim bolj pokrivajo vse funkcionalnosti, se lahko lotimo implementacije v razvojnem okolju. Kot pri vsakemu razvijalskem projektu je potrebno zagotoviti učinkovito upravljanje s kodo in sledenje spremembam, saj se teste s časom dopolnjuje, njihovo število in količina kode pa zato lahko močno narasteta. Kodo je zato potrebno shraniti v shrambo, ki pa bo, kot bomo videli, prišla prav tudi pri samodejnem prevajanju testov za testiranje nočne gradnje. Prav tako bodo tudi v avtomatizaciji testnih primerov hrošči [19]. Zato je potrebno vsak avtomatiziran testni primer preveriti, če je rezultat res tak, kot si ga želimo. Lahko se zgodi, da bo rezultat nadaljnjega testa odvisen od prejšnjega rezultata. V takih primerih je potrebno to predpostaviti (včasih tudi več scenarijev) in zagotoviti tekoče izvajanje kode.

V namen lažjega spreminjanja kode v bodoče, je pametno, da najprej določimo obliko implementacije – če in kateri razvijalski vzorec bomo uporabili in kako bomo spisali razrede.

4.3.1 Tabela indentifikatorjev

Grafični del aplikacije GemaLogic je implementiran s pomočjo ogrodja JavaServer Faces (https://javaserverfaces.java.net/), ki samodejno generira identifikatorje spletnim elementom, če jih ne določi programer eksplicitno. Hitra rešitev problema, ki se je izkazala za uspešno, je enostavna tabela identifikatorjev, ki jo sproti (ob razvoju) dopolnjujejo razvijalci sami. Zato se mora nahajati nekje v internem omrežju, v obliki, ki je dostopna vsem in omogoča čim hitrejše urejanje. Tabela ne vsebuje vseh spletnih elementov, ampak samo tiste, ki so za testerja oporni. Ta jo redno dopolnjuje med samim razvojem avtomatskih testov razvijalec pa po dodelitvi identifikatorja elementu tega vpiše v stolpec »Določen identifikator«.

#	Opis elementa	Izrisan XHTML	HTML Tag	Atribut	Določena identifikacija
1.	Gumb 'Prijava' na prijavni strani.	login.xhtml	input	name	loginForm:loginButton
2.	Povezava 'Odjava'.	info.xhtml	a	id	infoForm:logoutLink

Tabela 4.1.: Del tabele identifikatorjev, ki jo dopolnjujeta tako tester, kot tudi razvijalec.

Seveda je za to rešitev potreben dogovor in določena mera discipline, saj razvijalec dodatno izpolnjevanje tabelo med programiranjem vidi kot dodatno breme.

4.3.2 Oblika implementacije v Javi

Oblika implementacije je močno odvisna od aplikacije, ki jo testiramo, zato smo aplikacijo razdelili na več logičnih delov. Vsak tak logični del aplikacije predstavlja svoj javanski razred ali ločen nabor testnih primerov. Oblika implementacije odvisna tudi od tehnologij, ki jih uporabljamo. Na primer v našem primeru smo orodje Selenium WebDriver uporabili skupaj z ogrodjem JUnit. Torej je bilo potrebno v mislih imeti tudi to, da se bodo testi zaganjali z zaganjalnikom JUnit.

Pri implementaciji testov za testiranje funkcionalnosti aplikacije GemaLogic smo si pri razdelitvi na več logičnih delov pomagali z naslednjimi dejstvi:

- potrebujemo glavni razred, ki bo imel tako obliko, kot smo jo opisali v poglavju *Uporaba anotacij orodja JUnit*;
- prva stran je prijavna stran, torej potrebujemo razred `Login`, ki bo zagotavljal prijavo v aplikacijo;
- pred kakršno koli uporabo se je potrebno prijaviti kot veljaven uporabnik, zato se bo objekt razreda `Login` klical vedno preden bomo pričeli s testiranjem;
- po prijavi v aplikacijo ima uporabnik na voljo več različnih orodij za prikaz informacij, torej bomo za vsako orodje pripravili svoj nabor testnih primerov (ločena koda). Vsak glavni razred bomo zaradi boljše preglednosti poimenovali kot »Test« + angleško ime orodja. Primer: `TestBasicReport.java;`
- orodje navadno prikaže informacije v obliki grafa in tabele. Informacije so odvisne od parametrov, ki jih izbere uporabnik v posebnem oknu. Tako lahko za testiranje grafa implementiramo razred `CheckChart` in za testiranje funkcionalnosti tabel `CheckTable`. Kateri testni parametri bodo izbrani, se ve vnaprej, zato se tudi ve, katere rezultate pričakujemo;
- za učinkovito testiranje funkcionalnosti je potrebno shranjevanje podatkov v testno bazo. S posebnimi metodami bo potrebno zagotoviti izbris teh podatkov iz baze pred naslednjim zagonom testov.

Z upoštevanjem teh dejstev, smo kodo razdelili na več javanskih razredov. Razredi za orodje »Splošni pregled« so skupaj z njihovimi nalogami našteti v tabeli 4.2.

Razred	Naloge
`TestBasicReport.java`	• glavni razred, ki ga zaženemo z JUnit zaganjalnikom • vsebuje metode za preverjanje prisotnosti elementov na strani • poskrbi za inicializacijo Selenium Webdriverja in sistema za pisanje dodatnih dnevnikov (log4j) • kliče metode razredov `CheckTable` in `CheckChart` • vsebuje metodo za brisanje rezultatov prejšnjih izvajanj testov
`Login.java`	• poskrbi za prijavo v aplikacijo
`CheckTable.java`	• vsebuje metode za testiranje funkcij tabelaričnega prikaza
`CheckChart.java`	• vsebuje metode za testiranje funkcij grafičnega prikaza
`Utils.java`	• vsebuje druge pomožne metode (tvorjenje raznih nizov)

Tabela 4.2: Seznam razredov za implementacijo avtomatskih testov za testiranje orodja Splošni pregled. Vsa orodja aplikacije GemaLogic so si po zasnovi podobna, zato se tudi oblike imlementacij testnih primerov za ostala orodja ne razlikujejo veliko.

V naslednjih poglavjih si bomo podrobneje ogledali nekatere dele izvorne kode teh razredov in izpostavili nekaj rešitev.

4.3.3 Glavni razred

Kot smo že na kratko omenili, smo zaradi preglednosti glavni razred poimenovali kar po orodju, ki ga testiramo. Koda glavnega razreda za testiranje orodja Splošni pregled, se zato nahaja v razredu `TestBasicReport`. Implementirali smo ga po zgledu iz poglavja *Uporaba anotacij orodja JUnit*. Glavni razred se tako prične z metodo `setUp()`, ki smo jo izkoristili za inicializacije in pripravo testnega okolja. Na tem mestu smo poskrbeli tudi za inicializacijo sistema za pisanje dnevnikov Log4J (http://logging.apache.org/log4j/2.x/) in izbrali brskalnik v katerem se bodo poganjali testi (inicializacija objekta `WebDriver`). Dodatno smo maksimirali okno brskalnika, saj se s tem skoraj izognemo manjši pomanjkljivosti orodja Selenium, da bi klikali na drsnike brskalnika namesto na željen spletni element. Problem se sicer sliši, kot obroben in dokaj nepomemben, a tester lahko za njegovo odkritje porabi ogromno časa.

```
@BeforeClass
public static void setUp() throws Exception  {
    ConsoleAppender ca = new ConsoleAppender();
    ca.setWriter(new OutputStreamWriter(System.out));
    ca.setLayout(new PatternLayout("%-5p [%t]: %m%n"));
    log.addAppender(ca);

    log.setLevel(Level.INFO);
    DesiredCapabilities caps =
    DesiredCapabilities.internetExplorer();

    driver = new RemoteWebDriver(new
             URL("http://192.168.100.80:30035/wd/hub"), caps);

    driver.get("http:// 192.168.100.80:30030/gema");

    driver.manage().window().maximize();

    Login.loginToPage(driver);

    deleteTestAnalysis();

    driver.findElement(
           By.xpath("//a[@title='Splošni pregled']")).click();
}
```

Izvorna koda 4.1: Metoda `setUp()`, ki se izvrši pred vsakim izvajanjem testov.

V metodi `setUp()` se izvede tudi klic metode razreda `Login` (ogledali si ga bomo v naslednjem podpoglavju), ki poskrbi za prijavo v aplikacijo. Preden odpremo orodje »Splošni pregled«, je potrebno poklicati metodo `deleteTestAnalysis()`, ki počisti morebitne ostanke prejšnjih izvajanj testov.

4.3.4 Prijava v aplikacijo

Pri pisanju kode za samodejno prijavo v aplikacijo, je bilo potrebno upoštevati, da se prijavna stran vedno prikaže v jeziku, ki je kot privzeti nastavljen v brskalniku, s katerim uporabnik pregleduje stran. Zato smo uporabili izbirnike, ki so neodvisni od prikazanega teksta.

V kolikor bi namesto izbirnika `By.id("loginForm:submitLink")` uporabili `By.xpath(//input[text()='Prijava'])`, bi ob angleškem prikazu strani prišlo do izjeme `ElementNotFoundException`, saj spletni element gumba ne bi bil najden. Druga stvar, ki jo je bilo potrebno zagotoviti, je prijava s pravilnim uporabnikom. Možno je, da je pri prejšnjem izvajanju testov prišlo do nepričakovane napake in brskalnik ni bil pravilno zaprt. Takrat se zaradi mehanizma enkratne prijave (angl. single sign-on) prijavna stran ne prikaže,

ker je seja še veljavna. V tem primeru se je potrebno najprej odjaviti nato pa ponovno prijaviti. To logiko smo enostavno implementirali tako, da lovimo izjemo `NoSuchElementException`. Do te pride, če spletni elementi za prijavo niso najdeni. V stavku `catch` pokličemo metodo za odjavo `logout()` in nato ponovimo postopek prijave.

```
public static void loginToPage(WebDriver driver){
    try{
        driver.findElement(By.id("username")).sendKeys("admin");
        driver.findElement(By.id("password")).sendKeys("admin");
        driver.findElement(By.id("submitLink")).click();
    }catch (SeleniumException e){
        log.warn("Logout required. Trying to logout first.",e);
        logout(driver);

    driver.findElement(By.id("username")).sendKeys("admin");
    driver.findElement(By.id("password")).sendKeys("admin");
    driver.findElement(By.xpath("//span[text()='Prijava']"))
    .click();
  }
}
```

Izvorna koda 4.2: Metoda `loginToPage()`, ki se nahaja v razredu `Login`.

4.3.5 Brisanje rezultatov prejšnjih testiranj

Rezultati nekaterih testnih primerov se zaradi same zasnove aplikacije GemaLogic shranjujejo v njeno bazo. Zato je bilo potrebno zagotoviti, da se pred ponovnim izvajanjem testov povrne prvotno stanje – pobrišejo sledi predhodnega izvajanja istih testov.

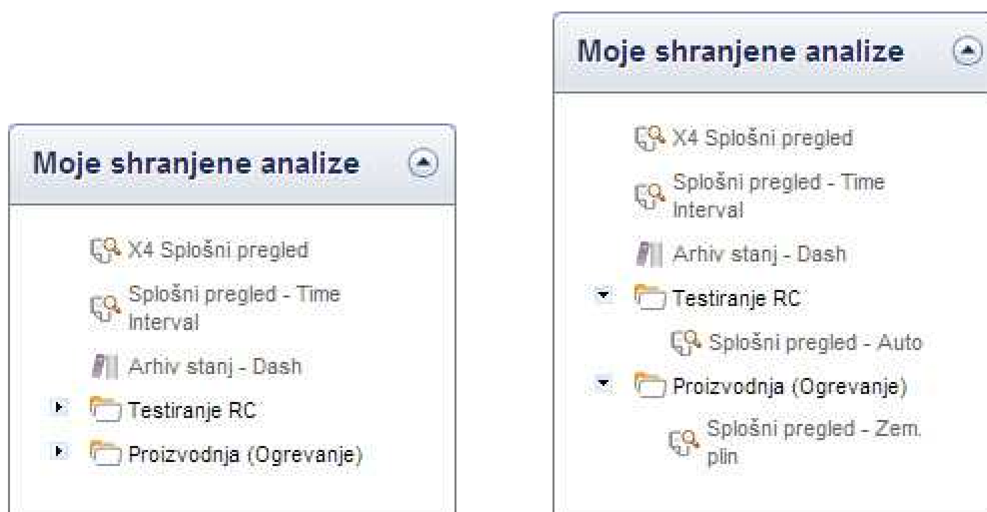
Metodo za brisanje rezultatov prejšnjih testov lahko poženemo:

1. na koncu izvajanja testov (kličemo jo iz metode z notacijo `@AfterClass`);
2. na začetku, prede sploh začnemo izvajati teste (kličemo jo iz metode z notacijo `@BeforeClass - SetUp()` metode.

Sami smo uporabili slednjo možnost, saj se v prvem primeru lahko zgodi, da se izvajanje testov nepričakovano prekine in se metoda za brisanje rezultatov prejšnjih testov sploh ne izvede.

V opisu aplikacije GemaLogic smo omenili, da je pogosto uporabljene analize možno shraniti za kasnejšo ponovno uporabo. Pri testiranju orodja »Splošni pregled« smo to zmožnost izkoristili za hitrejše izvajanje testov. S tem odpade potreba po vedno vnovičnem nastavljanju vseh parametrov analize. Seveda bi te shranjene analize lahko vplivale na rezultate nekaterih testnih primerov v trenutnem izvajanju. Zato je potrebno pred vsakim izvajanje preveriti če

obstajajo in jih odstraniti. V ta namen smo implementirali metodo `deleteTestAnalysis()`, ki poišče te analize in jih odstrani. Dostop do shranjenih analiz je mogoč samo preko drevesne strukture (implementirane s tehnologijo JavaScript), ki je prikazana na sliki 4.4. Najprej je potrebno s klikom na ikono puščice ali na ime starša razširiti drevo. Šele nato lahko analizo odpremo in odstranimo. Analize, ki se shranjujejo med izvajanjem testov, se namenoma vedno poimenuje z besedo »Auto«, ki sledi imenu orodja in pomišljaju. Tako lahko identificiramo analize, ki so bile shranjene med prejšnjim izvajanjem testov. Vse testne analize se tudi shranjujejo v mapo »Testiranje RC«.



Slika 4.4: Primer s tehnologijo JavaScript implementiranega drevesa shranjenih analiz. Leva slika prikazuje drevo z zaprtima mapama »Testiranje RC« in »Proizvodnja (Ogrevanje)«. Analizi »Splošni pregled – Auto« in »Splošni pregled – Zem. Plin« obstajata v DOM, a jih zaradi zaprtih map ni možno klikniti. Slika na desni prikazuje razširjeno drevo.

V kolikor drevo ni razširjeno, povezava do analize že obstaja v DOM, ampak je skrita in je ni možno klikniti. Zato je potrebno v primeru, da povezavo lociramo z metodo `findElements()` najprej z metodo `isDisplayed()` preveriti, če je klik na povezavo mogoč. Če ni, najprej razširimo mapo »Testiranje RC« in šele za tem v zanki odpremo in pobrišemo vsako analizo posebej.

```
public static void deleteTestAnalysis(){
    int count = 0;
    log.info("Checking for analyses from previous test runs...");


    if(driver.findElements(By.xpath("//td/a[contains(text(),
        'Splošni pregled - Auto')]")).size() > 0){
```

```
if(!driver.findElement(By.xpath("//td/a[contains(text(),
            'Splošni pregled - Auto')]")).isDisplayed()){

                driver.findElement(By.xpath(
                 "//td/span[text()='Testiranje RC']")).click();
        }
    }


    while(driver.findElements(By.xpath("//td/a[contains(text(),
            'Splošni pregled - Auto')]")).size() > 0){

          count++;
         driver.findElement(By.xpath("//td/a[contains(text(),
          'Splošni pregled - Auto')]")).click();

          driver.findElement(By.xpath("//span[text()='Briši']"))
           .click();

          try{
               driver.switchTo().alert().accept();
          }catch (Exception e){
              Log.error("Unable to confirm deletion!);
          }

    if (count != 0)
         log.info(count + " old analys(i/e)s deleted.");
}
```

Izvorna koda 4.3: Metoda `deleteTestAnalysis()`.

Brisanje analize je vedno potrebno potrditi s pomočjo dialoga, ki se prikaže ob kliku na gumb »Briši«. Iz kode 4.3 je razvidno, kako izberemo dialog in ga potrdimo z metodo `accept()`.

4.3.6 Testiranje grafičnega prikaza

Grafični prikaz podatkov je v aplikaciji GemaLogic v celoti realiziran s pomočjo tehnologije JavaScript, kar otežuje dostop do elementov grafa z izbirniki. Zato smo bili primorani v tem primeru uporabili drugačen pristop. S pomočjo metode `getPageSource()` v spremenljivko `pageSource` shranimo kar celotno kodo HTML trenutno prikazane strani. Testni primeri nato pravilnost prikaza preverjajo tako, da preverijo prisotnost pričakovane kode HTML v spremenljivki `pageSource`. Pričakovano kodo na podlagi prejetih parametrov generirajo in vračajo dodatne v ta namen implementirane metode. Izvorna koda 4.4 prikazuje preverjanje zaslonskega namiga (angl. tooltip) na grafu. Dodatna metoda `parseTooltipFromDate()` na podlagi vrednosti iz tabelaričnega prikaza generira kodo HTML ali JSON, katere prisotnost

v kodi strani (spremenljivka `pageSource`) se nato preverja z metodo `verifyChartTooltip()`.

```
pageSource = driver.getPageSource();
String tmpStr = "";

tmpStr = Utils.parseTooltipFromDate(driver.findElement(By.xpath(
         "//table[@class='rich- table']/tbody/tr/th")).getText(),
         "Podatek 1 > Podatek 2",
          driver.findElement(
          By.xpath("//td[@id='table:0:0']/span")).getText(),
          0);
```

Izvorna koda 4.4: Metoda `parseTooltipFromDate()` za parametre dobi vrednosti iz tabelaričnega prikaza na spletni strani in generira del HTML kode, ki mora biti vsebovan v spremenljivki `pageSource`.

Razred `ChartCheck` vsebuje še druge metode za preverjanje grafa, kot sta na primer `verifyChartMeasurementUnits()` za preverjanje enot na oseh in `verifyChartLegend()` za preverjanje legende pod grafom. Vse te metode zaradi zgoraj omenjenega pristopa delujejo na podoben način kot je prikazan v izvorni kodi 4.5.

```
public boolean verifyChartTooltip(String pageSource,
                                  String tooltipString){

    return (pageSource.contains(tooltipString));
}
```

Izvorna koda 4.5: Primer metode za preverjanje grafičnega prikaza.

4.3.7 Testiranje tabelaričnega prikaza

Tudi testiranja tabelaričnega prikaza smo se lotiku tako, da smo najprej sestavili spisek lastnosti in funkcionalnosti le-tega. V aplikaciji GemaLogic so to:

- pričakovano število stolpcev v testni analizi;
- možnost razdelitve tabele na več strani (paginacija);
- pričakovano število vrstic na stran;
- pomikanje po straneh;
- možnost sortiranja tabele glede na poljuben stolpec;

- možnost prikaza vsote v zadnji vrstici ali stolpcu tabele;
- prevračanje tabele.

Testne primere, ki testirajo naštete ključne točke smo implementirali v razred `CheckTable` kot funkcije, ki vračajo njihovo resničnost (`true/false`). Funkcije se tako, kot pri grafičnem prikazu, kličejo iz glavnega razrede preko JUnit metod `assert`. Izvorna koda 4.6 prikazuje primer klica metode za sortiranje tabele po decimalnih vrednostih, ki jih vsebuje.

```
assertTrue("Testing table sorting failed!",
TableCheck.checkSorting(driver));
```

Izvorna koda 4.6: Klic metode, ki preveri delovanje sortiranja tabele in vrne rezultat tipa `boolean`.

Klik na glavo stolpca sproži sortiranje celotne tabele. Pri tem ukazu se pošlje klic Ajax na strežnik, ki vrne največje ali najmanjše vrednosti (odvisno od smeri sortiranja). Vrednosti, ki jo pričakujemo ne poznamo. Zato ni možno uporabiti mehanizma za implicitno ali eksplicitno čakanje na zaključek sortiranja tabele. Tako smo čakanje na spletne elemente realizirali kar s klicem metode `wait()`, ki čakanje zagotovi s klicem metode `Thread.sleep()`.

```
public static void wait(int ms){
    try{
        Thread.sleep(ms);
    } catch (InterruptedException e) {}
}
```

Izvorna koda 4.7: Potratna alternativa mehanizmu za implicitno in eksplicitno čakanje na elemente. Žal, obstajajo primeri, ko je to edina rešitev.

4.4 Samodejno zaganjanje testnih primerov

Zgoraj opisano izvorno kodo testov lahko uporabimo za testiranje tako nočne gradnje, kot predizdaje. Pri testiranju predizdaje teste navadno zaganjamo preko razvojnega okolja (Eclipse IDE), saj je tester ob izvajanju prisoten. Drugače je pri zaganjanju za testiranje nočne gradnje. Tukaj je bilo dela nekoliko več, saj je potrebno samodejno sprožiti prevajanje in izvajanje testnih primerov. V naslednjih podpoglavjih si bomo podrobneje ogledali kako.

4.4.1 Avtomatsko testiranje nočne gradnje

Testiranje nočne gradnje se izvede v času, ko človek ni prisoten. Zato je potrebno postopek testiranja avtomatizirati v celoti. Torej vsakega izmed naslednjih korakov:

1. nastaviti urnik izvrševanja in avtomatsko izvrševanje,
2. prenos kode aplikacije iz shrambe,
3. prenos kode testov iz shrambe,
4. prevajanje kode,
5. prenos artefaktov na testni strežnik,
6. odstranitev starih artefaktov (in po potrebi zaustavitev aplikacijskega strežnika),
7. splavitev novih artefaktov in dvig aplikacijskega strežnika,
8. prevajanje kode testov,
9. inicializacija orodja Selenium WebDriver,
10. izvršitev testov,
11. priprava poročila testiranja.

Samodejno proženje izvajanja testnih primerov smo dosegli s kombinacijo močnega in priljubljenega orodja za avtomatizacijo gradenj ANT in aplikacije za sprotno integracijo Bamboo.

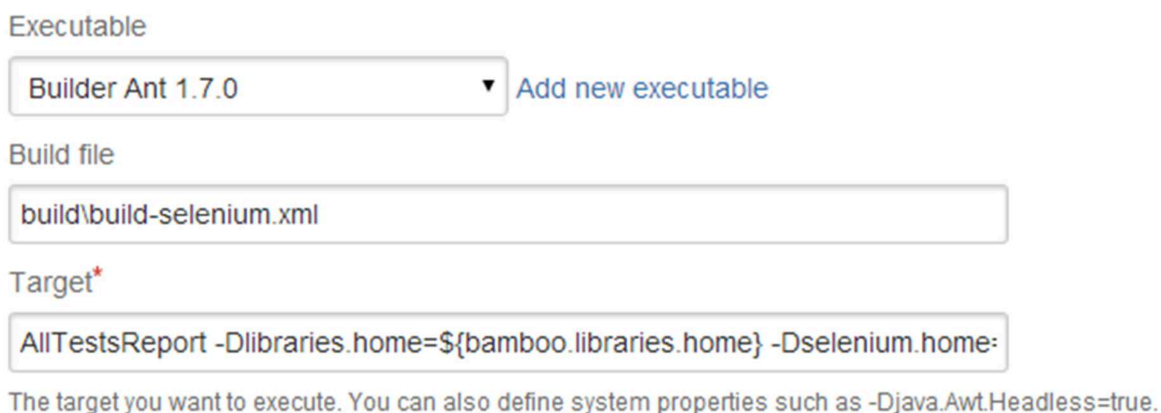
Bamboo omogoča definicijo ti. načrtov (angl. plans), s katerimi določimo, katera opravila se bodo izvajala in kdaj se bodo prožila. Realizacija prvih dveh zgoraj naštetih korakov je bila zato trivialna. Na sliki 4.5 je prikazan primer nastavitev prožilca tipa `Single daily build`, ki sproži gradnjo vsako noč ob 3:30.

Slika 4.5: Nastavitve prožilca nočne gradnje v aplikaciji Bamboo.

Skripta ANT se tako, kot tudi kodi testov in aplikacije, ki jo testiramo, nahaja v shrambi Subversion. Zato je samoumevno, da smo kot prvo opravilo v načrtu določili prenos vse kode na lokalni strežnik. Podpora za komunikacijo s shrambo Subversion je integrirana v Bamboo in ne predstavlja težave.

Vse ostale točke smo realizirali s pomočjo orodja ANT. Z aplikacijo Bamboo smo ga povezali tako, da smo ji posredovali izbrano verzijo knjižnice ANT, ki jo želimo uporabiti, lokalno pot do skripte ANT in ime cilja (angl. target) iz te skripte, ki ga želimo zagnati. Pri klicu cilja smo priredili tudi vrednosti dodatnim spremenljivkam (na sliki 4.6 `libraries.home`, in `selenium.home`), ki se uporabljajo v skripti. Obvezno na tem mestu posredujemo gesla in podobne občutljive podatke, saj sam zapis v skripto ni priporočljiv.

Glavni cilj `AllTestsReport` se nahaja v datoteki `build-selenium.xml`, ki se skupaj s vso kodo prenese iz shrambe. Kako to povemo aplikaciji Bamboo, je prikazano na sliki 4.6.



Slika 4.6: Nastavitve za izvajanje ANT opravila v aplikaciji Bamboo.

V skripti ANT smo implementirali več ciljev (za vsako GemaLogic orodje svoj cilj), saj to omogoča, da lahko poženemo teste samo za določeno orodje. Glavni cilj `AllTestsReport` požene vse teste za vsa orodja. Seveda je pomemben tudi vrstni red izvajanja ciljev – na primer, preden pričnemo izvajati teste, jih je potrebno najprej prevesti. To smo trivialno zagotovili z atributom `depends`, ki ga določimo cilju. Izvorna koda 4.8 prikazuje definicijo cilja `AllTestsReport`. Iz nje je razvidno, kako pred izvajanjem testov za več orodij pokličemo prenos iz shrambe in prevajanje testov. Določili smo tudi direktorij, kamor se bodo zapisali rezultati in kako bodo poimenovani.

```
<target name="AllTestsReport" depends="svn-checkout-selenium-
tests, init-selenium, build-selenium, TestBasicReport,
TestArchive, TestMonitoring, TestMT">
```

```
    <junitreport todir="seleniumresults">
        <fileset dir="seleniumresults">
            <include name="TEST-*.xml"/>
        </fileset>
        <report format="frames" todir="seleniumresults"/>
    </junitreport>
</target>
```

Izvorna koda 4.8: Skripta ANT za zagon JUnit testov in izpis rezultatov v datoteko XML.

Ko imamo teste prevedene, pred njihovim zagonom splavimo še aplikacijo GemaLogic na oddaljeni testni strežnik. Na prvi pogled to ne predstavlja nekega izziva, saj bi lahko uporabili storitev FTP, ali pa kar orodje za splavitev, ki je vgrajeno v sam Bamboo. Seveda pa je pred splavitvijo potrebno popraviti konfiguracijo aplikacij, počistiti stare datoteke, ali celo zaustaviti in ponovno zagnati aplikacijski strežnik. Zgoraj omenjeni možnosti nam tega ne omogočata, zato smo se odločili za opravili `scp` in `sshexec`, ki sta že vgrajeni v orodje ANT. `Sshexec` omogoča zaganjanje ukazov na oddaljenem strežniku, `scp` pa prenašanje datotek. Opravili omogočata tudi kriptirane prenosi preko omrežja, kar nam odgovarja, saj se strežnik za testiranje in strežnik za sprotno integracijo nahajata na fizično različnih lokacijah. Uporaba opravil je prikazana v izvorni kodi 4.9.

```
<scp port="${remote.deploy.port}"
     todir="${remote.deploy.userid}:${remote.deploy.password}@
            ${remote.deploy.server}:" trust="true" verbose="true">

    <fileset dir="${deploy.dir}/${build.tag}"/>
</scp>

<sshexec host="${remote.deploy.server}"
         port="${remote.deploy.port}"
         username="${remote.deploy.userid}"
         password="${remote.deploy.password}"
          command="./remote-deploy.bat"
         trust="true"/>
```

Izvorna koda 4.9: Skripta ANT za prenos datotek na testni strežnik in zagon skripte za splavitev.

Da smo se izognili večjemu številu ukazov preko opravila `sshexec`, smo na testnem strežniku pripravili skripto (`.bat` za Windows, `.sh` za Linux), ki poskrbi za pravilno konfiguracijo in morebitno opravljanje aplikacijskega strežnika. Ta skripta nam omogoči, da je za splavitev

preko opravila `sshexec` dovolj le en ukaz, kot je razvidno iz izvorne kode 4.9. Del skripte .bat, ki se proži lokalno na strežniku za testiranje prikazuje koda 4.10.

```
echo Stopping GprsBoxServer and JBossAS services...
net stop JBossAS

echo Removing old version and temp files...
rd /q/s C:\jboss\server\default\tmp
del /q C:\jboss\server\default\deploy\gema.ear

echo Moving new files to JBoss Deploy folder...
move gema.ear C:\jboss\server\default\deploy

echo Restarting GprsBoxServer and JBossAS services...
net start JBossAS
echo Finished deploying new build.
```

Izvorna koda 4.10: Del skripte .bat, ki splavi artefakt »gema.ear« v aplikacijski strežnik JBoss.

Za preglednejši prikaz rezultatov na spletnem vmesniku Bambooja, je bilo le-temu potrebno povedati, da bodo testi generirali rezultate in posredovati pot do datotek XML z rezultati, kot prikazuje slika 4.7.

Where should Bamboo look for the test result files?

☑ The build will produce test results.

If checked, the build will fail if no tests are found. Test output must be in JUnit XML format.

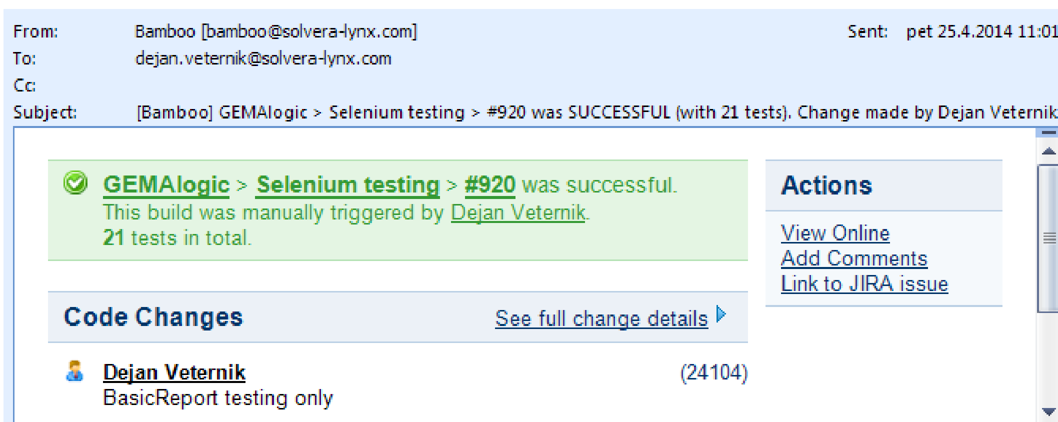
Specify custom results directories

**/seleniumresults/TEST-*.xml

Where does the build place generated test results?
this is a comma separated list of test result directories. You can also use Ant style patterns such as **/test-reports/*.xml

Slika 4.7: Nastavitve za preglednejši prikaz rezultatov na spletnem vmesniku Bamboo.

Poročilo vsebuje tako spremembe, ki so bile od zadnjega prenosa iz shrambe storjene na kodi, kot tudi rezultate testov. Bamboo ponuja tudi možnost pošiljanja poročila na določene elektronske naslove. Primer poročila je prikazan na sliki 4.8.



Slika 4.8: Sporočilo prejeto po elektronski pošti, ki ga pošlje strežnik za sprotno integracijo.

4.4.2 Avtomatsko testiranje predizdaje

Artefakte predizdaje aplikacije GemaLogic pripravijo razvijalci (lahko tudi s pomočjo aplikacije Bamboo). Naloga testerja je, da aplikacijo splavi na enak način in v okolju, ki je čim bolj podobno produkcijskemu. Splavitev za razliko od nočne gradnje v celoti poteka ročno. S tem preverimo še ostalo funkcionalnost, ki je nismo pokrili z avtomatskimi testi v nočni gradnji. Istočasno lahko spremljamo dnevnike aplikacijskega strežnika, da ne prihaja do napak.

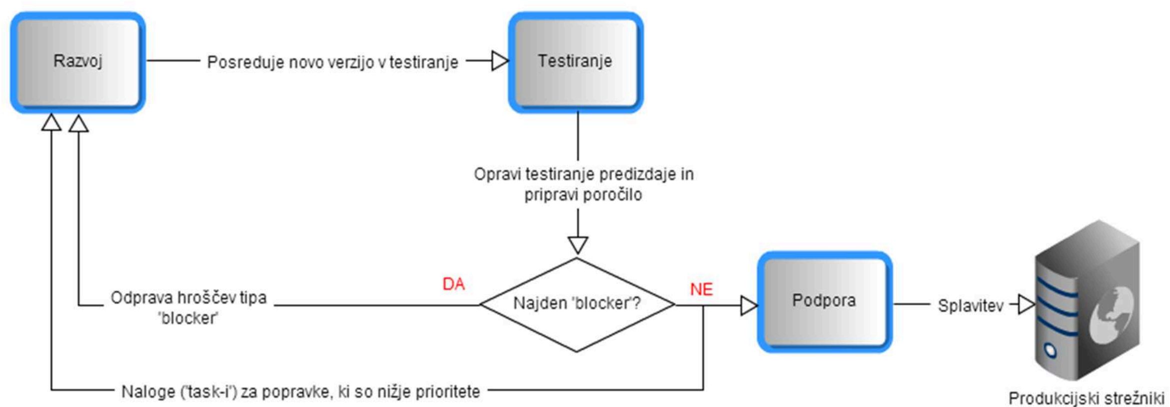
Testiranje predizdaje je obsežnejše od testiranja nočne gradnje. Ker vseh testov ni možno avtomatizirati, je potrebno pri testiranju predizdaje izvesti tudi ročni del testov.

V ta namen testiranje predizdaje izvajamo na delovni postaji osebe, ki je odgovorna za testiranje. Ker se uporablja gonilnik brskalnika tipa `RemoteWebDriver` – torej testi tečejo na testnem strežniku, lahko istočasno izvajamo še ročne teste, saj z našim klikanjem ne motimo interakcije Selenium WebDriverja z brskalnikom. V kolikor pride do kakršnihkoli nepričakovanih rezultatov, lahko hitro odreagiramo in če je potrebno ročno ponovimo scenarij, ki je povzročil težavo.

Kot smo videli v poglavju *3.7.3 Podpora v ostalih aplikacijah*, je izpis rezultatov s pomočjo JUnit zaganjalnika dovolj pregleden. Za dodatne izpise, ki omogočajo bolj podrobno spremljanje izvajanja testov, smo uporabili knjižnico Log4J.

V primeru, da pride do napake (najdemo hrošča) je dobro, da čim prej obvestimo odgovornega razvijalca, da prične z odpravljanjem problema.

Ko je testiranje zaključeno, se po odpravi napak lahko zgradi nova predizdaja, celoten cikel pa se ponavlja, dokler predizdaja ne vsebuje nobenega hrošča višje prioritete. Diagram celotnega poteka prikazuje slika 4.9.



Slika 4.9: Diagram testnega cikla pri testiranju predizdaje.

Poglavje 5 Ročno in avtomatsko testiranje v praksi

V tem poglavju si bomo ogledali, kakšen je doprinos avtomatizacije testiranja funkcionalnosti z orodjem Selenium WebDriver v primerjavi z ročnim testiranjem. Vsako stopnjo izvajanja testiranja bomo na kratko opisali in časovno primerjali obe varianti testiranja. Podatki, ki bodo predstavljeni izhajajo iz dejanskih meritev pri pisanju in izvajanju avtomatskih testov za sistem GemaLogic.

5.1 Priprava dokumentacije

Dokumentacija vseh testnih primerov je prvi korak, ki ga je potrebno storiti preden se sploh lotimo kakršnegakoli testiranja. Predstavljamo si jo lahko kot nekakšen temelj, tako v primeru ročnega testiranja, kot tudi avtomatskega testiranja. Dobro napisan dokumentacija za vsak testni primer vsebuje natančno opisan postopek in pričakovan rezultat. Pri dokumentiranju testnih primerov za aplikacijo GemaLogic smo za boljše razumevanje v opis postopka in rezultata po potrebi celo vstavljali slikovno gradivo. Vsak opis testnega primera mora biti tudi dodobra preverjen. Zato za pisanje dokumentacije lahko porabimo ogromno časa. Po naših izkušnjah je en človek porabil 5 uro za dokumentiranje 20 testnih primerov. Ta korak je potreben pri obeh variantah testiranja.

5.2 Načrtovanje in programiranje

Ob pregledno urejeni dokumentaciji lahko že pričnemo z ročnimi testi, medtem ko pri avtomatskem načinu izvajanja testov najprej naredimo načrt, kako se avtomatizacije testiranja lotiti na pravi način. Šele po tem se lahko lotimo s programiranja avtomatskih testov. Hitrost avtomatizacije testov je odvisna od kompleksnosti aplikacije, ki se testira. Če je aplikacija dovolj enostavna, je možno teste posneti z orodjem Selenium IDE. Takrat je čas avtomatizacije nekoliko daljši od časa trajanja enega cikla ročnega testiranja. V našem primeru orodje Selenium IDE zaradi kompleksnosti aplikacije GemaLogic ni bilo uporabno, zato so bili časi avtomatizacije precej daljši. En človek je za pripravo okolja in avtomatizacijo 30 testov za testiranje funkcionalnosti porabil 6 ur. Torej za 150 testov (kar v praksi ni veliko) smo porabili kar 30 ur. Kot smo že omenili, pri ročnem izvajanju testov tega koraka ni.

5.3 Izvrševanje testov

Sledijo testiranja. Večkrat se ročno izvrševanje testov ponovi, bolj je časovno zahtevno in s tem tudi dražje v primerjavi z avtomatskim. Razlike v časih so ogromne. Iz izkušenj pri testiranju aplikacije GemaLogic je množico testnih primerov v praksi en človek ročno izvršil v enem delovnem dnevu. Ista množica testnih primerov se je v avtomatsko izvršila v slabih 20 minutah, čeprav so bili testi zagnani z uporabo najpočasnejšega brskalnika, ki smo ga v času merjenja imeli na voljo (Internet Explorer 8). Časovne ocene tega koraka so ključen dokaz, zakaj se avtomatizacija splača. Zaradi velike hitrosti izvrševanja testov je sploh možno vsakodnevno testiranje nočne gradnje. Namreč, če bi se testiranje nočne gradnje opravljalo ročno, bi se tester praktično vsak dan ukvarjal le s tem. Kot smo že omenili, vedno obstaja odstotek testnih primerov, ki jih ni mogoče avtomatizirati. Teh primerov ni mogoče vključiti v nočno gradnjo, ker se mora v celoti izvesti samodejno. Potrebno pa jih je tako pri ročnem testiranju, kot tudi pri avtomatskem izvajanju testov, ročno izvesti na vsaki predizdaji.

5.4 Urejanje testnih primerov

V primeru, da je potrebno test popraviti zaradi spremembe uporabniškega vmesnika ali funkcionalnosti, je seveda časovno bolj potratno popravljanje avtomatskih testov. Poleg tega, da je potrebno popraviti dokumentacijo (tudi pri ročnem testiranju jo je potrebno), je potrebno še dodatno popraviti kodo avtomatskih testov in popravke dobro preveriti. Zato je urejanje in popravljanje avtomatskih testov časovno zelo potratno, še vedno pa ne izniči časa, ki ga vsak cikel testiranja prihranimo pri izvrševanju testov. V spodnji tabeli so zbrane ključne ocene, ki smo jih podali v tem poglavju.

	Dokumentiranje testnih primerov	**Programiranje testnih primerov**	**Izvrševanje testnih primerov**	**Urejanje testnih primerov**
Ročno testiranje	DA	NE	20 testov/uro	Dokumentacija
Avtomatsko testiranje	DA	DA (5 testov/uro, brez uporabe Selenium IDE)	8 testov/min	Dokumentacija + koda

Tabela 5.1: Primerjava potrebnih opravil in hitrosti izvajanja med ročnim in avtomatskim testiranje

Poglavje 6 Sklepne ugotovitve

Z vpeljavo nočnih gradenj in vsakodnevnega testiranja le-teh v proces razvoja, se kakovost programske opreme zelo zviša. Hkrati se skrajša čas testiranja predizdaje, saj so bil velik del testiranja funkcionalnosti že opravljeni s testiranjem nočnih gradenj. To nam posledično daje možnost, da postopoma skrajšamo čas med Sprinti.

Orodje Selenium WebDriver sicer (še) ni idealno za testiranje kompleksnejših aplikacij. Šele med daljšo uporabo v tem diplomskem delu opisanega testnega sistema so se pokazale nekatere pomanjkljivosti. Zgodilo se je že, da test ni bil uspešen, ker spletni element ni bil najden, čeprav je obstajal in bil viden v brskalniku. Zaznati je možno tudi nekoliko drugačno obnašanje na različnih delovnih postajah. Na primer, testi so na razvojni delovni postaji delovali brez problema, ko pa jih je zagnala aplikacija Bamboo pa so se pojavile težave s šumniki v imenih spletnih elementov. Potrebna je bila tudi optimizacija časovnih omejitev (angl. timeouts) pri čakanju na spletne elemente. Zanesljivost se lahko nekoliko dodatno zmanjša tudi zaradi porazdeljenosti testnega sistema na več naprav. Če v času izvajanja testov pade povezava s strežnikom Selenium Server, bodo posledično padli tudi testi. Bamboo jih bo ponovno zagnal šele naslednji dan. Kljub opisanim pomanjkljivostim orodje Selenium WebDriver še vedno prinaša dovolj koristi, da ga je smiselno uporabiti pri avtomatizaciji testiranja funkcionalnosti.

Prostora za izboljšave je ogromno tako na strani samega orodja Selenium WebDriver, kot tudi na strani opisanega testnega sistema. Na strani testnega sistema bi po potrebi lahko testiranje razširili na več brskalnikov. Kateri brskalnik želimo uporabiti, bi lahko izbrali preko dodatne spremenljivke ali parametra pri zagonu testov. Aplikacija Bamboo se dobro povezuje z drugimi Atlassianovimi produkti, kot sta Confluence in Jira. Zato bi bilo smiselno razmisliti celo o samodejnem ustvarjanju opravil razvojni ekipi glede na rezultate testiranja.

Literatura

[1] David Burns, *Selenium 2 Testing Tools Beginner's Guide*, Packt Publishing Ltd, 2012.

[2] Mike Cohn, *Succeeding with Agile Software Development Using Scrum*, Pearson Education, Inc 2010.

[3] Lisa Crispin, *Agile Testing A Practical Guide for Testers and Agile Teams*, Pearson Education, Inc., 2009.

[4] Unmesh Gundecha, *Selenium Testing Tools Cookbook*, Packt Publishing Ltd, 2012.

[5] Thomas Hammell, R. Gold, *Test Driven Development: A J2EE Example*, Apress, 2005

[6] Roy de Kleijn, *Learning Selenium*, Leanpu

[7] Henrik Kniberg, *Scrum and XP from the Tranches*, C4Media Inc, 2007.

[8] Chris Sims, Hillary Louise Johnson, *Scrum: a Breathtakingly Brief and Agile Inroduction*, Dymaxicon, 2012.

Spletni viri

[9] (2014) *Wikipedia: Scrum (software development)*. Dostopno na: http://en.wikipedia.org/wiki/Scrum_(software_development)

[10] (2012) *Scrum Best Practices: Choosing the Right Sprint Length*. Dostopno na: http://www.samuelcclemens.com/2012/12/scrum-best-practices-choosing-the-right-sprint-length/

[11] (2014) *Wikipedia: Selenium (software)*. Dostopno na: http://en.wikipedia.org/wiki/Selenium_(software)

[12] (2007) *Unit Testing vs Functional Tests.* Dostopno na: http://www.softwaretestingtricks.com/2007/01/unit-testing-versus-functional-tests.html

[13] *(2014) What is a neutral build.* Dostopno na: http://www.wisegeek.com/what-is-a-neutral-build.htm

[14] (2012) *Selenium PageObjects and PageFactory.* Dostopno na: http://relevantcodes.com/pageobjects-and-pagefactory-design-patterns-in-selenium/

[15] (2014) *JUnit Tutorial, poglavje JUnit – Parameterized Test*. Dostopno na: http://www.tutorialspoint.com/junit/junit_parameterized_test.htm

[16] (2013) *How the InternetExplorerDriver Works*. Dostopno na: https://code.google.com/p/selenium/wiki/InternetExplorerDriverInternals

[17] (2014) *Wikipedia: Ajax (programiranje).* Dostopno na: http://sl.wikipedia.org/wiki/Ajax_(programiranje)

[18] (2014) *HtmlUnit.* Dostopno na: http://htmlunit.sourceforge.net/

[19] (2001) *Seven steps to automation success.* Dostopno na: https://www.prismnet.com/~wazmo/papers/seven_steps

[20] (2014) *Wikipedia: Usage share of web browsers*. Dostopno na: http://en.wikipedia.org/wiki/Usage_share_of_web_browsers

[21] *(2008) The Agile Impact Report, Proven Performance Metrics from the Agile Enterprise.* Dostopno na: http://www.rallydev.com/sites/default/files/Agile_Impact_Report.pdf

[22] *Selenium WebDriver*. Dostopno na: http://docs.seleniumhq.org/projects/webdriver/