UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Gorazd Kovačič

Avtomatsko vizualno testiranje spletnih strani

DIPLOMSKO DELO
NA UNIVERZITETNEM ŠTUDIJU

Mentor: izr. prof. dr. Viljan Mahnič

Ljubljana, 2015

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Avtomatsko vizualno testiranje spletnih strani

Tematika dela:

Opredelite vlogo vizualnega testiranja spletnih strani in opišite probleme, ki se pri tem pojavljajo. Na podlagi tega definirajte potrebne korake za izvajanje avtomatiziranih regresijskih testov in opišite orodja, ki se za to uporabljajo. Podrobno analizirajte dva pristopa: testiranje na osnovi primerjave bitnih slik in testiranje na osnovi podatkov o postavitvi posameznih elementov ter predlagajte kombinacijo, ki bo združevala prednosti enega in drugega.

Mentor: izr. prof. dr. Viljan Mahnič

IZJAVA O AVTORSTVU

diplomskega dela

Spodaj podpisani, **Gorazd Kovačič**,

sem avtor diplomskega dela z naslovom:

Avtomatsko vizualno testiranje spletnih strani.

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 v zbirki »Dela FRI«.

V Ljubljani, dne 10.6.2015 Podpis avtorja: _____

Zahvala

Zahvaljujem se svoji družini za podporo, spodbudo in razumevanje. Prav tako se zahvaljujem svojemu mentorju za strokovno pomoč in vodenje.

Kazalo

Seznam uporabljenih kratic in simbolov

TDD (*Test-Driven Development*) – testno usmerjeni razvoj

BDD (*Behaviour-Driven Development*) – vedenjsko usmerjeni razvoj

UAT (*User Acceptance Tests*) – testi sprejemljivosti

HTML (*Hyper Text Markup Language*) – označevalni jezik za oblikovanje večpredstavnostnih dokumentov

CSS (*Cascading Style Sheets*) – prekrivni slogi

RWD (*Responsive Web Design*) – odzivna zasnova spletnih strani

SPA (*Single Page Application*) – enostranska aplikacija; spletna aplikacija, ki je po značilnostih podobna namiznim aplikacijam

DOM (*Document Object Model*) – model za predstavitev objektov dokumenta

URL (*Uniform Resource Locator*) – internetni naslov, na katerem je vsebina

XPath (*XML Path Language*) – poizvedovalni jezik za XML-dokumente

Povzetek

Naloga podaja pregled avtomatskega vizualnega testiranja spletnih strani s prikazom uporabe orodij, ki so trenutno na voljo, in problemov, ki se pri postavitvi takšnega testiranja pojavljajo. Na podlagi tega definira postopek testiranja, ki za avtomatsko ocenjevanje izgleda spletnih strani kombinira dva pristopa: primerjavo bitnih slik in primerjavo postavitve elementov.

V začetnem delu je izpostavljena pomembnost testno usmerjenega razvoja in predstavljena ločnica med testi, ki so v podporo ekipi, ter testi, ki so namenjeni kritiki produkta. Ta meja se z avtomatskim vizualnim testiranjem spletnih strani mehča. Sledi opis postopka avtomatskega vizualnega testiranja, ki se deli na korake vzpostavitve stanja brskalnika, zajema izgleda, ocenitve izgleda in vrnitve rezultata o uspešnosti. Pri tem so poudarjeni problemi zaradi množice naprav, operacijskih sistemov in resolucij, s katerimi se dostopa do spletnih strani, kar je pomembna ovira za ročno preverjanje. Izgled na različnih resolucijah je treba preverjati zaradi uporabe odzivne zasnove spletnih strani. Zaradi razlik v delovanju pa je treba izvajati teste na različnih brskalnikih in operacijskih sistemih.

V nadaljevanju je prikazan pregled orodij in storitev, ki so na voljo za izvedbo vizualnega testiranja. Na primeru spletne strani iz zavarovalniške aplikacije sta podrobneje analizirana dva pristopa: regresijsko testiranje s primerjavo bitnih slik in regresijsko testiranje s primerjavo postavitve elementov. Ker ne prvi ne drugi sam zase ne dajeta dovolj točnih rezultatov, je v nalogi predlagan kombiniran pristop, ki zadovoljuje vse zahteve avtomatskega vizualnega testiranja spletnih strani.

Ključne besede: testno usmerjeni razvoj, avtomatsko testiranje, razvoj spletnih strani, vizualno testiranje

Abstract

This thesis reveals an overview of automated visual web testing and displays currently available tools usages, as well as problems occurring in setting up such environment. Based on that, test procedure for visual web estimation is defined with two approaches: bitmap image comparison and elements layout comparison.

The importance of test-oriented development is emphasized in the initial part of the thesis and further, a boundary between the tests to support the team and the tests to critique the product is presented. This boundary is softening with automated visual web testing. Then, there is a description of automated visual testing, which is divided into the following steps: webpage state establishment in the browser, capture of the layout, evaluation of the layout, and return of the report. With that, a peculiarity in a multitude of devices, operating systems and resolutions that enable access to the websites is highlighted which is a significant obstacle for manual verification. The appearance on different resolutions must be validated due to the use of responsive web design. Because of the behavior differences, tests must be executed in different browsers and operating systems.

Further, there is an overview of tools and services available for visual testing. On insurance web application example an analysis of two approaches are presented: regression bitmap image comparison testing and regression element layout comparison testing. Because neither satisfies requirements, combined approach is recommended that satisfies all requirements of automated visual web testing.

Keywords: Test-Driven Development, Automated Testing, Web Development, Visual Testing

1 Uvod

Razvoj spletnih strani je vedno bolj dinamičen. Naprav, ki imajo dostop do spleta, je vedno več. Danes je možno brskati po spletu že prek ročne ure ali televizije. Programska oprema je z razvojem vedno bolj kompleksna in njeno obvladovanje je nujno. Pri tem nam pomagajo agilne metodologije razvoja, ki umeščajo testiranje v svoj center. Testiranje ni več samo postopek pred namestitvijo programske opreme v produkcijsko okolje, ampak tudi orodje za razvoj opreme. Določena programska oprema se že sprotno dostavlja v produkcijsko okolje (*continuous delivery*).

Pri spletnih straneh poznamo več različnih faktorjev kot običajno, ki lahko vplivajo na delovanje programske opreme. Postavitev avtomatskega testiranja za spletne strani je z napredki v strojni in programski opremi nujna. Narava izvajanja po eni strani prinaša nestabilnosti glede okolja, po drugi strani pa je arhitektura zastavljena na način, ki omogoča enostavne pristope k testiranju. Avtomatsko vizualno testiranje je eno izmed njih.

Avtomatsko vizualno testiranje je vrednotenje ustreznosti izgleda uporabniškega vmesnika. Pri spletnih straneh uporabimo posnetek strani na različnih brskalnikih, napravah in operacijskih sistemih, saj se izgled lahko razlikuje namenoma, tj. z uporabo odzivne zasnove spletnih strani, ali pa pomotoma zaradi nepričakovanega obnašanja spletne strani. Posnetek lahko zajamemo kot bitno sliko in jo primerjamo z želeno, kar uporabljamo pri regresijskem testiranju, ali pa ga zajamemo z dodatnimi podatki in preverjamo postavitev posameznih elementov glede na specifikacijo testa. Z avtomatizacijo vizualnega testiranja želimo tovrstne teste uporabiti v standardnem procesu testno usmerjenega razvoja in omogočati sprotno dostavo programske opreme.

Cilji naloge so opredelitev avtomatskega vizualnega testiranja spletnih strani, pregled možnosti izvajanja v zvezi s tem s pregledom orodij in storitev ter izvedba na konkretnem primeru.

V drugem poglavju tako predstavimo testno usmerjeni razvoj programske opreme, navedemo razloge za njegovo izvajanje in ga primerjamo s tradicionalnimi pristopi k razvoju. Prav tako predstavimo kvadrant agilnega testiranja, v katerega umeščamo avtomatsko vizualno testiranje.

V tretjem poglavju orišemo posebnosti pri razvoju spletnih strani in prikažemo, za kako raznolika okolja moramo razvijati spletno stran. Drugi del poglavja namenjamo postopkom avtomatskega vizualnega testiranja. Pri tem upoštevamo možnosti, ki so danes na voljo.

V četrtem poglavju, glede na opisane postopke v tretjem poglavju, naredimo pregled orodij in storitev, ki so na voljo na trgu. V zadnjem, petem, poglavju izvedemo eksperimenta, kjer

preverimo obnašanje primerjave bitnih posnetkov ter primerjave postavitve. Na primeru pa izvedemo postavitev avtomatskega vizualnega testiranja. Pri tem upoštevamo spoznanja o avtomatskem vizualnem testiranju spletni strani iz eksperimenta.

2 Testiranje programske opreme

2.1 Metode razvoja programske opreme

Tradicionalne metodologije razvoja programske opreme se izkazujejo za zastarele in prehod v agilne metodologije že nekaj časa prevladuje. Potreba po hitri in iterativni dostavi programske opreme je že standardna ter v interesu tako naročnika kot tudi razvijalca. V interesu naročnika sta dobrobit, ki jo dobi z novo programsko opremo in čimprejšnja možnost oblikovanja oz. preoblikovanja zahtev na podlagi videnih rezultatov, kar predstavlja zelo dober način za zgodnje preverjanje razvoja. V interesu razvijalca pa sta čimprejšnje preverjanje rezultatov dela in sposobnost spreminjanja zasnove, kjer se predpostavke spreminjajo na podlagi novih zahtev. To pomeni, da se programska oprema ne samo gradi, ampak tudi nadgrajuje in s tem tudi spreminja. Ravno prilagajanje spremembam je tisti del, ki onemogoča tradicionalne pristope.

Tradicionalne metodologije predvidevajo, da je razvoj programske opreme ponovljiv proces v obliki proizvodnega procesa, kar pa ne drži [4]. Pri oblikovanju programske opreme igra pomembno vlogo testno usmerjeni razvoj (*Test-Driven Development* oz. TDD). Testiranje programske opreme je sicer prisotno v vseh metodologijah. Se pa razlikuje glede na to, kdo jih izvaja, kdaj se izvajajo, kako se izvajajo in kaj je predmet testiranja. Pri obravnavi razvoja programske opreme kot proizvodnega procesa je testiranje nekaj, kar se zgodi ob koncu, kjer se preverja ustreznost izdelka. Na to pa lahko gledamo kot poskus vgraditve kakovosti v izdelek, namesto da bi v proces izdelave vgradili kakovost [6].

Testiranje in s tem zagotavljanje kakovosti ima v agilnih metodah osrednjo vlogo. Agilne metode razvoja programske opreme že v definiciji predvidevajo naslednje [4]:

- obsežno zbirko avtomatskih testov,
- koda brez testa ne sme priti v produkcijsko okolje,
- najprej se napišejo testi in
- testi narekujejo, kaj je treba razviti.

Potreba po testiranju se še posebej izkaže takrat, ko je v razvoj programske opreme vključena večja skupina in je treba usklajevati ter preverjati njihovo delo sproti in hitro. Sprotno in hitro testiranje omogoča zgodnje odkrivanje napak ter tako znižuje stroške njihovega odpravljanja. Tako je avtomatizacija zbirke testov pomemben element izvajanja testiranja. Ob t. i. agilni revoluciji je bilo ugotovljeno, da je uspešnih projektov zgolj 16 %. Značilnost uspešnih projektov pa so bile nizka cena, kratko trajanje in majhne ekipe, kar pomeni, da gre za projekte z nizko kompleksnostjo [4]. Potreba po testiranju izhaja tudi iz potrebe po dokazovanju

delovanja razvite ali popravljene programske opreme vsem zainteresiranim strankam. Prav tako pa povečuje nivo zaupanja in zmanjšuje stres [5]. Rezultati testiranja so dokaz o delovanju programske opreme. Testiranje služi tudi zanesljivemu preurejanju (*refactoring*) kode, kjer so neuspešno izvedeni testi vodilo o uspešnosti preurejanja. Spreminjanje programske opreme je pri iterativnih ciklih pogosto in ravno zato je pomembno programsko opremo ne samo graditi, ampak tudi spreminjati.

2.2 Testno usmerjeni razvoj

Za testiranje programske opreme sta potrebna disciplina in vnaprejšnje planiranje testiranja. Testno usmerjeni razvoj predvideva pristop, da to postane rutinsko opravilo, ki ne izvaja samo zagotavljanja kakovosti, ampak tudi pomaga pri samem razvoju.

Testno usmerjeni razvoj je razvoj, kjer veljajo naslednja pravila [5]:

1. Najprej se napišejo testi.
2. Testi se zaženejo tako, da so neuspešni.
3. Razvijamo vsebino, dokler testi niso uspešni.

Omenjeni vrstni red ima precej posledic. Med prvim korakom pisanja testov je treba že dobro razmisliti, kakšen problemski prostor se rešuje, kakšni so robni pogoji in kakšni pričakovani rezultati. Tega koraka ne moremo izvesti, dokler vsebina in obseg nalog nista popolnoma jasna.

Drugi korak je pomemben, ker dokazuje, da je lahko test tudi neuspešen. Ta je sicer najbolj enostaven, ni pa nujno trivialen, saj je treba programsko kodo v tem koraku tudi prevesti. Ta korak je torej namenjen pisanju potrebne infrastrukture in zagotavlja, da je test lahko tudi neuspešen. Če bi bil to zadnji korak, bi obstajala možnost, da je test napisan slabo ali presplošno. Najbolj enostaven primer takšnega testa je test, ki je vedno uspešen.

Tretji korak je tisti, ki bi ga izvedli tudi v primeru, če testov ne bi bilo. Ključno je, da je ta korak zadnji, saj ob kakovostnem prvem koraku uspešni testi že zagotavljajo pravilno delovanje.

Vsi trije koraki skupaj pa dajejo razvijalcu programske opreme boljši občutek o napredku, o stvareh, ki jih je še treba narediti, in ob vseh uspešnih testih signal o opravljenem delu.

Vse naštete posledice pa imajo pozitiven vpliv tako na razvijalca kot tudi na sam projekt.

2.3 Vrste testiranja

Agilni pristopi k razvoju programske opreme in ekstremno programiranje narekujejo pogosto spreminjanje ter pogosto dostavo delujočih verzij programske opreme. Pri kompleksnih sistemih je ročno testiranje dolgotrajno in preverjanje rezultatov kompleksno. Pri tem testno usmerjeni razvoj igra ključno vlogo, saj so, ob koncu razvoja ali popravljanja obstoječe funkcionalnosti, testni primeri pripravljeni na avtomatsko izvedbo.

Avtomatizacija izvajanja testov je zelo pomembna, saj je testov tudi za na videz majhno funkcionalnost lahko veliko. V večjih paketih programske opreme pa je nujno, da so funkcionalnosti testirane v največji možni meri. Pri pogosti dostavi in vpletenosti velikega števila razvijalcev testiranje ne bi bilo možno brez avtomatizacije, saj bi trajalo veliko časa, vključen pa bi bil človeški faktor. Pomembno pri tem pa je, da metodologija omogoča rast tako kompleksnosti programske opreme kot tudi števila razvijalcev in nam s tem omogoča izvedbo vedno bolj zahtevnih projektov.

Poznamo različne vrste testov, ki preverjajo različne poglede na programsko opreme. Po načinu izvajanja testov poznamo:

- ročne in
- avtomatske teste.

Ročno testiranje je testiranje, ki ga izvaja uporabnik programske opreme z izvajanjem akcij. V takšnih primerih imamo ročno napisane scenarije, ki jih preverjamo. Takšno testiranje je dolgotrajno in vsebuje človeški faktor. Pri kratkih ciklih razvoja je težko konsistentno izvajati oziroma ponavljati testiranje. Težko je tudi povečevati količino testiranja, saj smo omejeni na čas, ki ga uporabniki za scenarije potrebujejo. Običajno se ročno izvajajo raziskovalni testi, testi sprejemljivosti, testi uporabnosti in scenariji. Ročno se izvajajo tudi testi, ki jih ni možno avtomatizirati iz različnih razlogov, kot so: drago okolje za postavitev, drag razvoj avtomatskih testov, pomanjkljivi algoritmi, pomanjkljiva orodja itd. Avtomatske teste je možno izvajati pogosto – tudi večkrat dnevno oz. ob vsaki spremembi. To pomembno vpliva na dinamiko razvoja in nivo zaupanja v razvoj. Ravno avtomatizacija nudi možnost obvladovanja večjih razvojnih ekip in večjih kompleksnosti programske opreme. Nekaterih testov ni možno izvajati avtomatsko zaradi različnih razlogov (drago okolje za postavitev, drag razvoj avtomatskih testov, pomanjkljivi algoritmi, pomanjkljiva orodja itd.).

Po obsegu poznamo [9]:

- teste enot,
- integracijske teste,
- sistemske teste in
- teste sprejemljivosti (*User Acceptance Tests* oz. UAT).

Testi enot so najbolj pogosti in najbolj enostavni. Predstavljajo teste samostojnih neodvisnih enot, ki predstavljajo zaključeno celoto. To je lahko enostavna funkcija ali pa npr. v objektnem programiranju objekt ali komponenta. Glede na delovanje imamo lahko pripravljenih množico testov za enoto, ki preveri pravilnost reševanja problemskega prostora, ki ga pokriva. Potrebne značilnosti testov enot so hitrost izvajanja in preverjanje majhne zaokrožene celote funkcionalnosti. Morebitne odvisnosti drugih enot je treba izvzeti iz testiranja. Hitrost je v

praksi pomembna, ker je teh testov običajno veliko in jih je treba izvajati pogosto – ob vsaki spremembi kode. V primeru napake pri testu enote je precej enostavno ugotoviti, kje je problem, kar je dobra osnova za povezovanje enot v večje sklope.

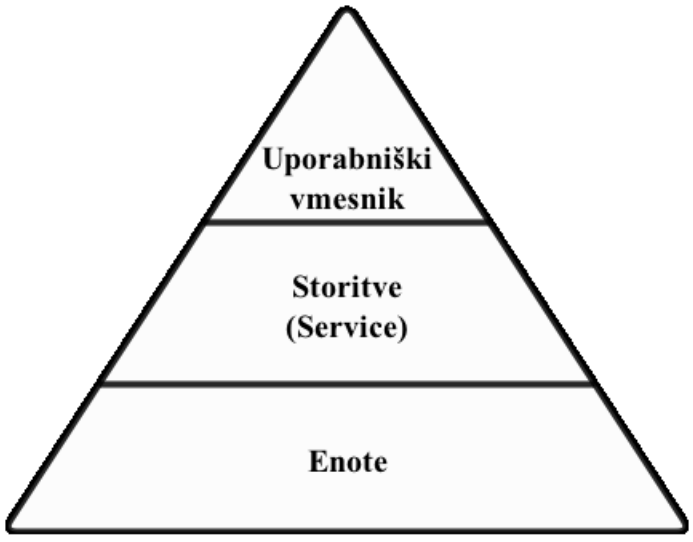
Integracijski testi povezujejo več enot skupaj prek vmesnikov in preverjajo povezano delovanje ter sodelovanje teh enot. Po obsegu so zahtevnejši in kompleksnejši. V primeru napake je v integracijskem testiranju težje ugotoviti, od kod napaka izhaja. Če so povezane enote dobro testirane, lahko hitreje sklepamo o napaki v integraciji enot. Integracijske teste je možno sestavljati takoj, ko je smiselno, in ni treba čakati, da so vse enote razvite. Pri tem poznamo več tehnik, kot so inkrementalna, od zgoraj navzdol (*Top down*), od spodaj navzgor (*Bottom up*) in njihove različne kombinacije [9].

Sistemski oz. sistemski integracijski testi so različica integracijskih testov, kjer so vključene vse enote, ki so v celoti povezane med seboj. Testira se torej sistem kot celoto.

Do sedaj naštete vrste testov so tesno povezane z načinom razvoja opreme, uporabljenimi tehnologijami in vzorci. Testi sprejemljivosti so po obsegu in predmetu testiranja drugačni ter so bližji uporabnikom, saj se testira končni rezultat dela. Običajno so to zahteve na višjem nivoju, ki naj bi jasno opredelila pričakovanja uporabnika s funkcionalnega, performančnega ter z vizualnega vidika. Tovrstne teste je treba posredovati razvojni ekipi vnaprej, da se lahko preveri njihova ustreznost pred dostavo.

2.4 Obseg avtomatskega testiranja

Teste izvajajo vsi, ki so vpleteni v razvoj programske opreme, in tudi njeni uporabniki. Ker imajo zahteve in razvoj iterativno naravo, je avtomatizacija pomemben del testiranja.



Slika 1 Piramida avtomatskega testiranja (*The Test Automation Pyramid*) [6]

Dodatno je pomembno, da se testi izvajajo hitro in tako nudijo hitro povratno informacijo. Po konceptu piramide avtomatskega testiranja [6] se na različnih nivojih programske opreme izvaja različno število avtomatiziranih testov.

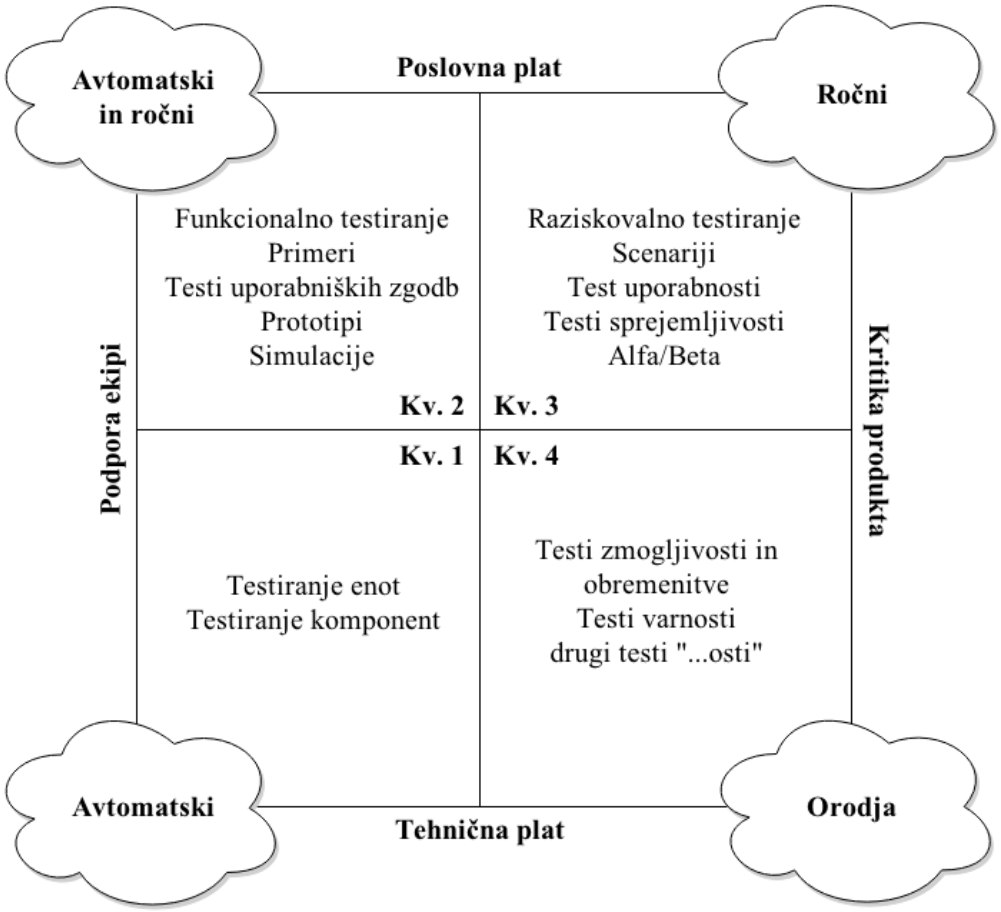
Testiranje enot je glede na število testov največje. Ker se testirajo majhne funkcionalnosti v izoliranem okolju, je tovrstne teste enostavno avtomatizirati.

Avtomatizacija testiranja uporabniškega vmesnika je po drugi strani zelo kompleksna. Izdelava testov s snemalniki je običajno enostavna, vendar so ob spremembah slabo izdelani testi neuspešni in zahtevajo ponovno izdelavo. Ta krhkost testov pa prinaša negativne posledice, kot so: višja cena, nezadovoljstvo zaradi ponavljajočega se dela in pa tudi nevarnost opustitve avtomatizacije tega dela testiranja. Za takšne teste obstaja nevarnost, da so sami sebi namen in ne služijo testiranju. Ob postavitvi avtomatskega testa obstaja tudi problem obsega testiranja. Kadar se testi pripravijo kot zaporedje korakov, ki bi jih naredil uporabnik, se testira celotna aplikacija s poslovno logiko in integracijo. Takšni testi so lahko počasni in s tem tudi nepravočasni za potrebe sprotnega preverjanja rezultatov. V primeru neuspešnega testa je napake zaradi velikega obsega takšnih testov tudi težje odkriti in odpraviti.

Zaradi teh razlogov je dobro načrtovati testiranje uporabniškega vmesnika ločeno od preostanka programske opreme in vključiti ločeno testiranje storitev [6], ki jih uporabniški vmesnik uporablja. To je mogoče samo, kadar je program načrtovan na takšno delitev odgovornosti. Kadar je testiranje del procesa izdelave opreme, pa je to tudi možno takrat zagotoviti. Po avtorju [6] storitveni del *»ni omejen na storitveno orientirano arhitekturo«,* ampak predstavlja poslovne dele aplikacije kot storitve, ki jih uporabniški vmesnik lahko uporablja za izvajanje. Kadar se storitveni del testira ločeno z vsemi potrebnimi robnimi pogoji, je testiranje uporabniškega vmesnika precej enostavnejše, saj se funkcionalno testirajo pravilnosti povezanosti grafičnih elementov s klici pravilnih storitev in vizualno njihov izgled ter postavitev. Temu lahko rečemo test povrhnjice kože (*SubcutaneousTest*) [8]. Zaradi tega pa je po piramidi avtomatskega testiranja testov uporabniškega vmesnika malo.

2.5 Ostale vrste testiranja

Naštete vrste testov niso vse, ki obstajajo. Kot že prej omenjeno, teste delimo po tem, kako se izvajajo, po obsegu, pogostosti izvajanja, kdo jih izvaja in kaj od njih pričakujemo. V delu [7] sta avtorja razdelila problem agilnega testiranja v 4 kvadrante.



Slika 2 Kvadranti agilnega testiranja [7]

Po vodoravni dimenziji delimo teste na tiste, ki podpirajo razvojno ekipo, in tiste, ki so namenjeni kritiki produkta. Po navpični dimenziji pa delimo teste na tiste, ki so namenjeni tehničnemu delu, in tiste, ki so namenjeni poslovnemu delu produkta.

Testi, ki podpirajo razvojno ekipo, imajo s tehnične plati teste enot in teste komponent, ki jih sicer lahko štejemo za enote, s poslovne plati pa funkcionalno testiranje, primere uporab, uporabniške zgodbe, prototipe in simulacije.

Testi, ki kritizirajo produkt, pa so po tehnični plati testi zmogljivosti in obremenitev, varnosti, zanesljivosti, razširljivosti ter drugi [3]. Po poslovni plati pa razlikujemo raziskovalno testiranje, scenarije, teste uporabnosti in sprejemljivosti ter alfa in beta teste.

Vse vrste testov se med seboj zelo razlikujejo po svojem namenu in obsegu. Delitev, ki je predstavljena, pa zelo dokončno določa pripadnosti posameznih vrst testiranja. Tema je zelo aktualna, vendar že obstajajo predlogi po spremembi vodoravne osi, saj delitev v nekaterih primerih ni potrebna. Primeri teh sprememb so testi zmogljivosti kot testi, ki so v podporo ekipi in hkrati še vedno služijo za kritiko produkta, in funkcionalni testi, ki so v svojem bistvu testi

sprejemljivosti, če jih v enaki obliki vključimo v avtomatsko testiranje in so s tem na voljo za podporo ekipi [2].

Iz tega sledi, da če se najde formalni način opisa testa in obstajajo orodja za njihovo avtomatizacijo, lahko združimo levo in desno stran diagrama v nekem obsegu. S tem primerom se ukvarja vedenjsko usmerjeni razvoj (*behavior-driven development* oz. BDD). To je posebna vrsta testno usmerjenega razvoja, ki je po namenu bližje poslovnemu delu kvadranta in po izvedbi bližje tehnični plati. V zvezi s poimenovanjem BDD je sicer precej nejasnosti, ki so opisane v [1] in kjer avtor izbere popolnoma novo ime 'specifikacija s primerom' (*Specification by Example*). Gre za prakse, ki omogočajo izvedbo tovrstnega testiranja.

3 Avtomatsko vizualno testiranje spletnih strani

3.1 Vizualizacija spletnih strani

Spletna aplikacija je arhitekturno sestavljena iz strežniškega dela (*server-side*) in dela za odjemalca (*client-side*). Del za odjemalca se izvaja v brskalniku. Brskalnik pošlje zahtevo za izbrani vir na strežniški del aplikacije, ta pa ga servira. Komunikacija poteka po protokolu HTTP, ki je namenjen izmenjavi besedilnih podatkov brez stanja. Kadar odjemalec zahteva spletno stran, jo brskalnik prikaže. Pri tem lahko zahteva dodatne vire, ki so potrebni za prikaz strani.

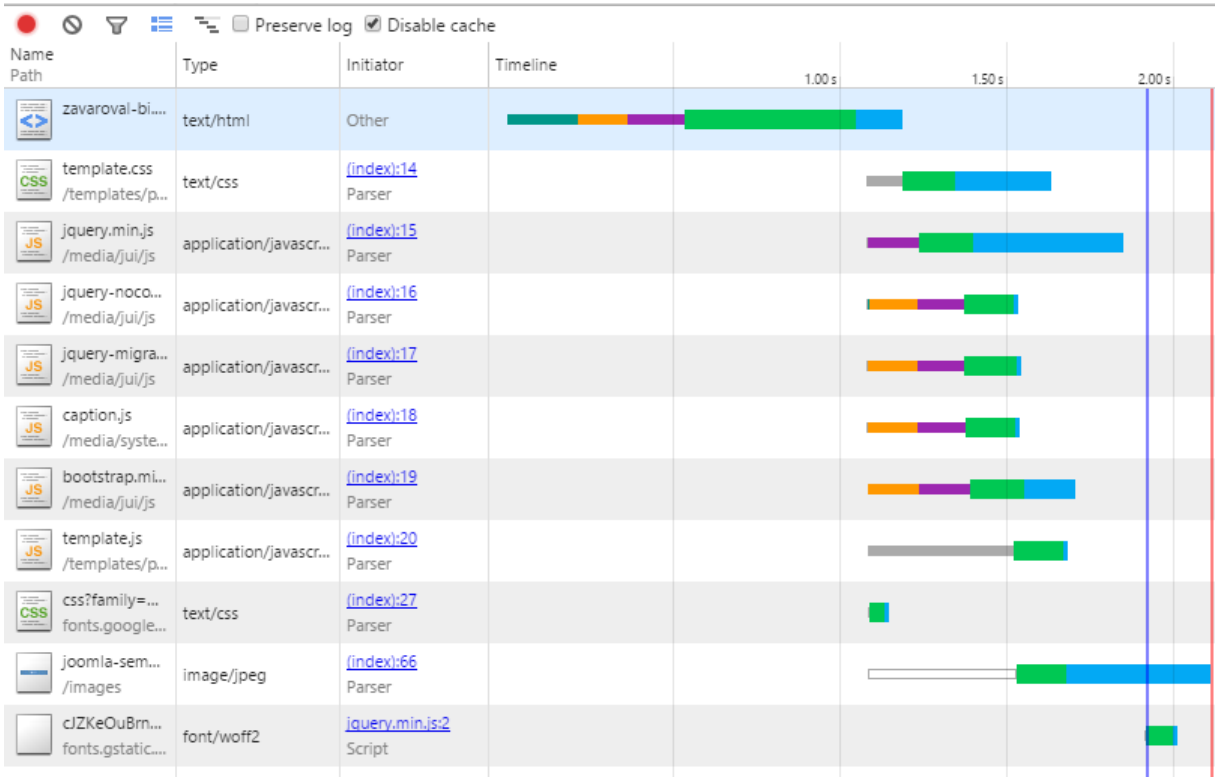
Spletna stran je oblikovana z označevalnim jezikom HTML (*Hyper Text Markup Language*), ki ga je definirala organizacija W3C (*World Wide Web Consortium*). Dodatni viri, na katere se spletna stran lahko navezuje, so:

- slikovni viri,
- CSS-viri
- viri Javascript in
- drugi viri.

S slikovnimi viri je možno besedilo obogatiti s slikovnim gradivom, ki ga umestimo na želeno mesto. Z viri CSS (*Cascading Style Sheet*) obogatimo spletne dokumente z oblikovnimi stili. Z viri Javascript omogočimo izvajanje kode v brskalniku in s tem interaktivno uporabniško izkušnjo.

Obstajajo tudi druge vrste virov, ki jih brskalnik lahko vizualizira, kot so npr. pisave, vektorske slike, video vsebine itd. Kombinacija različnih vrst virov danes omogoča širok nabor spletnih strani z značilnostmi namiznih aplikacij.

Brskalniki omogočajo vizualizacijo komunikacije z zunanjim svetom, ki je namenjena predvsem razvijalcem. Slika 3 prikazuje primer časovnice komunikacije med brskalnikom in strežnikom, kjer je razvidno, da je bila izvedena prva zahteva. Na podlagi rezultata pa je bila izvedena še množica drugih zahtev (stolpec *Initiator*). Iz stolpca *Type* pa je razvidno, za kakšno vrsto vira gre.



Slika 3 Prikaz zahtev brskalnika na primeru

Opisi virov so predmet standardizacije za to namenjenih organizacij. Jezik Javascript je tako standardizirala organizacija Ecma International in je voden pod šifro ECMA-262 z imenom ECMAScript [12]. HTML in CSS-viri pa je standardizirala organizacija W3C, ki izdaja priporočila namesto specifikacij [37]. Standardizacija je zapleten in dolgotrajen postopek z namenom trajnosti specifikacije. V primeru W3C gre dokument skozi vrsto različnih faz, kot so: osnutek, kandidat priporočila, predlog priporočila in na koncu priporočilo W3C [14].

Zaradi dolgotrajnosti postopkov in interesa uporabnikov razvijalci uporabljajo dokumente, ki še niso v končni fazi priporočila W3C.

Brskalnik si strukturne podatke naloži v t. i. DOM (*Document Object Model*), ki predstavlja reprezentacijo elementov dokumenta za vizualizacijo. Prek Javascripta je možno z DOM-podatki tudi manipulirati. To omogoča dinamičnost vsebine strani.

3.2 Razvoj spletnih strani

Z arhitekturnega vidika ločimo dva tipa spletnih strani. Prvi je generirana stran na strežniku. Vsebina je lahko statična ali pa je na podlagi zahteve generirana v celoti. To omogočajo tehnologije, kot so: PHP, Ruby On Rails, ASP.NET, JSP in druge. Značilnost te arhitekture je priprava HTML-strani v celoti. Drugi tip je generirana stran na brskalniku s pomočjo Javascripta. Temu rečemo SPA (*Single Page Application*). Obstaja tudi mešani tip, kjer se ne generira celotna stran, ampak samo del strani.

Značilnost prvega tipa spletnih strani je sposobnost prikaza strani na brskalniku brez izvajanja Javascripta. Gre za prenos celotne strani v celoti za vsako zahtevo in izvajanje zahtevnega procesiranja strani na strežniku. Takšne strani običajno uporabljajo izmenjavo podatkov s strani do strani prek dodatnih parametrov z namenom, da bi uporabnik dobil občutek stanja. Arhitektura sicer stanja ne predvideva in ga oponaša prek dodatnih parametrov seje in podatkov s strani. Z vidika vizualizacije je po prenosu virov in izrisu vsebine strani stanje v brskalniku stabilno in s tem pripravljeno na testiranje.

SPA-arhitektura pa za izvajanje uporablja Javascript in statične vire s strežnika. Stanje se vodi v brskalniku. Z manipulacijo DOM-podatkov strani se izvajajo spremembe vsebine strani. S stališča izvajanja je arhitektura bolj zahtevna za napravo in brskalnik, na katerem se izvaja. Zaradi vedno boljših naprav in napredkov v razvoju brskalnikov je ta možnost vedno bolj privlačna, saj nudi nepovezano delovanje, gostovanje v domorodnih (*native*) aplikacijah naprav in primerljivo izkušnjo namiznim aplikacijam. Takšne strani prinašajo nove izzive pri razvoju spletnih strani. Razvoj strani zahteva veliko kode Javascript, ki pa temu ni bil nikoli namenjen. Da bi prešli te težave, se pojavljajo orodja, ki generirajo Javascript iz varnejših jezikov, ki statično preverjajo tipe. Primeri so TypeScript, WebSharper, Clojure.

Posebnost SPA-aplikacij z vidika vizualizacije je manipulacija vsebine in oblike strani prek Javascripta. Pri tem je v splošnem težko ugotoviti, kdaj je vizualno stanje stabilno in s tem pripravljeno na testiranje.

3.3 Naprave in brskalniki

Spletne strani se izvajajo v brskalnikih. Brskalniki so namenski programi, ki si jih uporabnik namesti na svoj operacijski sistem oz. napravo in predstavljajo stik s spletnimi aplikacijami. Slabost takšnega okolja z vidika razvoja spletnih strani je v tem, da med razvojem ne poznamo izvajalnega okolja, to pa se lahko skozi čas tudi spreminja z novimi napravami in različicami. Po drugi strani pa je spletni brskalnik na voljo na vedno več napravah z internetno povezavo. Med takšne naprave štejemo pametne telefone, tablice, televizije, igralne konzole itd.

Prikaz spletne strani in njena uporabnost morata biti prilagojena lastnostim posameznih naprav. Uporaba mobilnih naprav se povečuje in takšen trend je pričakovati tudi naprej [10]. Iskalnik Google kot odziv na takšen trend že izloča strani iz rezultatov iskanja, ki niso prijazne mobilnim napravam pri iskanju na takšnih napravah [28].

Pri načrtovanju vizualnega testiranja imamo tako širok nabor naprav, ki jih lahko vključimo v testiranje. Pri tem si lahko pomagamo s statistikami, ki so javno dostopne, ali pa vodimo statistiko za svojo stran. Glede na to, da se osredotočamo na spletne strani, do katerih dostopamo prek brskalnikov, je potrebna precejšnja previdnost pri zanašanju na statistike uporabe. Teh je na internetu na voljo veliko. Da bi bolje razumeli, kaj se trenutno dogaja, je treba razumeti, kako do takšnih podatkov sploh lahko pridemo.

Brskalnik ob zahtevi na strežnik posreduje podatke, ki lahko strežnikom omogočajo različno ravnanje. Tako lahko strežnik takšne podatke zbira in obdeluje. Spletne strani imajo v sebi vključene tudi namenske elemente, ki izbrane podatke posreduje spletnim storitvam za potrebe analiz. Pri interpretaciji takšnih podatkov je treba biti previden, saj veljajo za druge strani in ne za tisto, ki jo razvijamo.

Za potrebe splošnega pregleda prikazujemo podatke podjetja StatCounter, ki nudi storitev analiz dostopov do internetnih strani, ki uporabljajo omenjeno storitev. Hkrati pa nudi javno dostopne splošne podatke po različnih dimenzijah.

Podatki, ki bi nas lahko zanimali za načrtovanje testiranja, so:

- uporaba brskalnikov,
- uporaba naprav in
- uporaba resolucij na mobilnih napravah.

Prikazujemo podatke v obdobju od januarja do vključno aprila 2015 za Slovenijo. Tabela 1 kaže, da prevladuje dostop do spletnih strani z namiznimi računalniki. Vendar pa se za dostop do spletnih strani v več kot 15 % uporabljajo mobilne naprave in tablice. V nadaljnjih poizvedbah to upoštevamo in jih ločimo na namizne ter mobilne naprave in tablice skupaj.

Vrsta naprave	**%**
Namizna	84,51
Mobilna	10,77
Tablica	4,70
Igralna konzola	0,03

Tabela 1 Uporaba vrst naprav za dostop do spletnih strani v Sloveniji [34].

Tabela 2 kaže na raznolikost resolucij, uporabljenih pri mobilnih napravah in tablicah. Nekatere resolucije so skladne glede na pokončno oz. ležečo uporabo naprave. Tabela 3 prikazuje uporabo brskalnikov na teh napravah. Tabela 4 pa prikazuje podatke o uporabi brskalnikov na namiznih računalnikih.

Resolucija naprave	%
360 x 640 (640 x 360)	21,4 (2,84)
768 x 1024	15,8
1280 x 800 (800 x 1280)	8,82 (1,58)
320 x 568	6,56
480 x 800 (800 x 480)	5,44 (0,8)
720 x 1280	3,99
320 x 480	3,25
320 x 534	3,16

Tabela 2 Najbolj pogoste resolucije mobilnih naprav in tablic v Sloveniji [34].

Brskalnik	%
Chrome	42,63
Safari	26,26
Android	24,87

Tabela 3 Najpogostejši brskalniki na mobilnih napravah in tablicah v Sloveniji [34].

Brskalnik	%
Chrome	47,06
Firefox	30,66
IE	16,73
Safari	3,27
Opera	1,67

Tabela 4 Najpogostejši brskalniki na namiznih računalnikih v Sloveniji [34].

Ti podatki so lahko samo okvirno vodilo. Za svojo spletno stran je treba voditi svojo statistiko. Z orodji, kot sta StatCounter ali Google Analytics, si lahko pri tem pomagamo. Iz prikazanih podatkov lahko kljub temu sklepamo, da je raznolikost prisotna in je ne smemo zanemariti niti pri razvoju niti pri vizualnem testiranju spletnih strani. Glede na ciljne oz. dejanske uporabnike lahko razvijamo in vizualno testiramo spletne strani s konfiguracijami, ki jih uporabniki imajo. Ker želimo, da je naša stran funkcionalna in ima privlačen izgled na vseh napravah (namiznih računalnikih, pametnih telefonih, tablicah), se moramo temu prilagoditi.

Pri razvoju spletnih strani lahko izdelamo ločene aplikacije, prilagojene konfiguracijam, kar zahteva dodaten razvoj in vzdrževanje, ali pa prilagodimo spletno stran z uporabo odzivne zasnove spletnih strani (*Responsive Web Design* oz. RWD). Z uporabo odzivne zasnove se DOM-podatki strani prilagajajo glede na različne karakteristike naprave, brskalnika in načine upravljanja. Tako lahko na manjšem ekranu prikažemo podatke horizontalno, namesto vertikalno. Določene elemente lahko privzeto skrijemo oziroma jih dodatno prikažemo. RWD se ureja prek CSS-poizvedb glede na lastnosti naprave. Poizvedbe podpirajo lastnosti, kot so:

dimenzije naprave, orientacija, razmerje prikaza (*aspect ratio*) in drugi. CSS-poizvedbe so opisane v W3C-priporočilu [13].

Izgled strani na različnih napravah se lahko kljub uporabi RWD razlikuje iz različnih razlogov. Dokler se izgled strani ne preveri z različnimi brskalniki na različnih resolucijah, ne moremo biti zagotovo prepričani, da je ustrezen. Pri vizualnem testiranju si pomagamo z avtomatizacijo izvajanja testov na izbranih konfiguracijah. Pri tem lahko uporabljamo prave brskalnike in tudi brezoblične brskalnike (*headless browser*, brskalnik brez grafičnega uporabniškega vmesnika), ki so po funkcionalnosti zelo podobni pravim, so hitrejši v izvajanju in primernejši za uporabo pri avtomatskem testiranju [23]. Tovrstni brskalniki slonijo na istih tehnologijah kot pravi brskalniki. Primer takšnega brskalnika je PhantomJS [29].

Pri razvoju in avtomatskem vizualnem testiranju spletni strani je treba sprejeti odločitve o konfiguracijah, na katerih želimo razvijati in preverjati spletno stran. Pri tem je treba upoštevati, da se okolje spreminja z npr. novimi različicami brskalnikov.

3.4 Postopek testiranja

Po kvadrantu agilnega testiranja, opisanega v 2.5, bi lahko vizualno testiranje umestili v kvadrant 3, ki je namenjen kritiki produkta s poslovne plati. Po tej delitvi bi to lahko bili testi uporabnosti in tudi sprejemljivosti. Glede na namen konkretne aplikacije je izgled lahko bolj ali manj pomemben. Pri vizualnem testiranju želimo preveriti možnosti avtomatizacije teh testov. Z avtomatizacijo testov približamo vizualno testiranje h kvadrantu 2 in s tem podporo ekipi.

3.4.1 Namen

Z vizualnim testiranjem preverjamo, ali je uporabniški vmesnik videti, kot bi bilo treba. Takšna definicija je zelo splošna in jo je treba opredeliti. Avtomatizacija je pomembna zaradi velikega števila spremenljivk v takšnih aplikacijah po izgledu, kjer je ročno testiranje pri hitrih iteracijah razvoja programske opreme praktično nemogoče. Posebnosti pri izvajanju spletnih strani nas silijo v redno preverjanje izgleda zaradi zunanjih dejavnikov pri izvajanju, kot so na primer nove različice brskalnikov in nove naprave.

3.4.2 Predmet testiranja

Testira se lahko izgled celotne strani ali pa posameznih elementov strani. Zaradi hitrosti izvajanja je dobro, da se vsebina testiranja omeji na vizualni del, kolikor je možno. Kadar govorimo o testiranju celotne strani, govorimo o funkcionalnem testiranju in hkrati o testih sprejemljivosti. Kadar pa pripravimo strani za potrebe testiranja posameznih gradnikov, govorimo o testih enot oz. komponent. S testi enot lahko vizualno testiranje izvedemo na posameznih gradnikih in nato integracijsko še na celotni strani. S tem lahko vizualno testiranje spletnih strani umestimo v TDD kot enakovreden element.

3.4.3 Zahteve

Postopek vizualnega testiranja mora biti načrtovan tako, da ne moremo dobiti lažno uspešnih rezultatov in da je lažnih neuspešnih rezultatov čim manj. Posledica lažnih uspešnih rezultatov je prepozno zaznavanje napak. Poročila o tem prejmemo v najslabšem primeru prek pritožb uporabnikov ali še huje prek zmanjšane uporabe strani. Lažni neuspešni rezultati pa vodijo v dodatno nepotrebno delo ekipe, ki mora ročno pregledovati in potrjevati poročila o neuspešnih testih in s tem vnaša nezaupanje v same teste. Ravno nezaupanje v teste pa lahko povzroči kasnejši izklop ali njihovo ignoriranje.

3.5 Koraki vizualnega testiranja

Vizualno testiranje spletnih strani mora izvesti naslednje korake:

1) Vzpostavitev stanja strani v brskalniku.
2) Zajem izgleda strani ali dela strani.
3) Ocenitev izgleda.
4) Vrnitev rezultata ustreznosti oz. neustreznosti izgleda.

Prvi korak je vzpostavitev želenega stanja strani. Običajno je dovolj že vpis naslova v brskalnik. Glede na spletno stran pa so lahko potrebni še dodatni koraki prek interakcije na strani (npr. izpolnitev potrebnega podatka ali klik na gumb). Drugi korak pa je dejanski zajem posnetka, kjer imamo različne možnosti. Zajem je zelo povezan s tretjim korakom ocenitve izgleda, ki jo lahko izvajamo s pomočjo regresijskega testiranja ali opisa zahtevanega izgleda. Primerjava je glede na drugi korak možna s primerjavo bitne slike ali s primerjavo prek strukturiranega opisa. Zadnji korak je sporočanje rezultata testiranja, kjer je možno sporočiti ne samo ustreznost, ampak tudi dejansko primerjavo slik v primeru regresijskega testiranja.

Postopek želimo v največji možni meri avtomatizirati.

3.5.1 Vzpostavitev stanja strani v brskalniku

Pri vzpostavitvi stanja strani v brskalniku gre za zagon brskalnika s potrebnimi parametri. Med njimi štejemo vzpostavitev nastavitev, kot so omogočanje izvajanja Javascripta, vzpostavitev zahtevane resolucije brskalnika in drugi parametri, ki lahko vplivajo na delovanje.

V brskalnik vpišemo naslov strani, ki jo testiramo. Če se je med razvojem strani upoštevala potreba po testiranju uporabniškega vmesnika, je postopek končan, saj lahko naslov s parametri že pripravi stran v ustrezno stanje. Če pa je treba za vzpostavitev stanja še izvajati akcije, pa je te treba izvesti.

Nekatere strani imajo lahko elemente, ki se pogosto spreminjajo in jih je težko testirati. Takšni elementi nedeterminizma so lahko med drugim oglasi, izpis trenutnega časa ali podatki glede na geografsko lokacijo. Takšne elemente je za potrebe učinkovitega testiranja treba izločiti iz

preverjanja, da bi bili testi lahko učinkoviti. Z izločevanjem takšnih elementov zmanjšujemo delež lažnih neuspešnih rezultatov.

Izločanje lahko dosežemo na dva različna načina. Prvi način je prilagoditev strani za potrebe izvajanja testov. To je možno doseči z dodatnim nivojem abstrakcij, ki za potrebe testiranja uporablja predpripravljene podatke, ki so iz testa v test isti. Tu je treba paziti, da ne pride do prevelike poškodbe zasnove aplikacije (*Design Damage*). Drugi način je označitev izbranih delov strani in dopolnitev izgleda z atributom vidnosti prek CSS-razreda.

Pri vzpostavitvi stanja na brskalniku je treba biti pozoren na to, da vzpostavitev stanja ni nujno takojšnja. Na to lahko vplivajo:

- viri, ki jih mora brskalnik prenesti s strežnika,
- skripte, ki se morajo izvesti za vzpostavitev dokončnega stanja in
- čas, ki ga brskalnik potrebuje, da izriše vsebino na ekran.

Vse tri točke so zelo pomembne pri obravnavi hitrosti spletne strani, ki pa niso predmet dela. Z vidika vizualnega testiranja je pomembno, da vemo, kdaj je stanje vzpostavljeno.

3.5.2 Zajem izgleda strani

Zajem izgleda strani lahko izvedemo s posnetkom strani na izbranih nastavitvah. Enostaven zajem predvideva posnetek z bitno sliko celotne strani ali posameznih elementov. Predmet zajema je odvisen od vsebine posameznega testa. Poleg bitnega posnetka lahko zajamemo tudi DOM-podatke, kot so lokacija in velikost posameznih elementov na sliki.

3.5.3 Ocenitev izgleda

Po zajemu izgleda strani je možno izvesti oceno izgleda. Neregresijsko avtomatsko ocenjevanje iz bitne slike je zelo težko, lahko pa si pomagamo s primerjavo z drugim posnetkom. Iz DOM-podatkov je možno preverjati lokacijo in velikost posameznega elementa ali pa relativno postavitev glede na drugi element.

Primerjavo posnetka lahko izvedemo s posnetkom:

- zadnjega uspešnega testa,
- iz drugega naslova ali
- izbrane referenčne konfiguracije.

V prvem in drugem primeru govorimo o regresijskem testiranju. V prvem primeru imamo posnetek zadnjega uspešnega testa shranjen. V drugem primeru izvedemo dodatni zajem ločene strani, ki ne vključuje sprememb, ki jih preverjamo. Razlika med njima je tudi v tem, da se v drugem primeru ne upošteva razlik v delovanju brskalnik in konfiguracije. S takšnim načinom je tudi enostavnejše menjati konfiguracije, ki so predmet testiranja. Po drugi strani pa zahteva dodaten napor pri zagotavljanju delovanja starejših različic spletnih strani. Pri tretjem primeru

gre za zaupanje v določeno referenčno konfiguracijo, ki jo uporabimo kot osnovo za primerjavo z ostalimi. V tem primeru gre za primerjavo med konfiguracijami.

Primerjava bitnega posnetka je enostavna, vendar postopki izrisovanja strani z uporabo grafične procesne enote in drugi sistemski dejavniki lahko povzročijo razlike na posameznih točkah posnetka. Razlike se pojavljajo zaradi različnih nastavitev operacijskih sistemov, grafičnih procesnih enotah in njihovih gonilnikih. Razlike nastanejo z mehčanjem (*antialiasing*), razlikami v svetlosti barv, zamiki pik pri izrisovanju pod nivojem resolucije (*subpixel shift*) ipd.

Težave je možno rešiti z izvajanjem testov na identičnih okoljih, izklopom dejavnikov, ki prinašajo razlike, s sprejemanjem manjših odstopanj pri ocenitvi izgleda ali z uporabo boljših algoritmov. Vpeljava dovoljenih odstopanj lahko ublaži lažne neuspešne rezultate, ki so rezultat sistemskih razlik, vendar omogoči tudi lažne uspešne, če bi dovolili prevelika odstopanja. To bi dovolili na podlagi majhnega zamika elementa, ki je glede na celotno stran velik. Primerjava bitnega posnetka bi zahtevala nastavitev večjega odstopanja, ki pa bi na majhnem elementu omogočala lažne uspešne rezultate.

Preverjanje postavitve elementov pride v poštev pri preverjanju postavitve strani (*page layout*), kjer je lahko, glede na različne resolucije brskalnika, postavitev strani drugačna. To dosežemo z uporabo odzivne zasnove spletnih strani (*Responsive Web Design*).

3.5.4 Vrnitev rezultata ustreznosti oz. neustreznosti izgleda

V primeru neuspešnega testa je treba rezultat javiti. Za razvijalca, ki mora napako oz. regresijo odpraviti, je nujno, da dobi na voljo čim boljši opis napake. Pri vizualnem testiranju uporabimo posnetek, ki ga lahko opremimo z lokacijo napake. V primeru regresijskega testiranja lahko pripravimo tri posnetke: originalnega, trenutnega in primerjavo obeh. Na tak način je napako lahko najti in s tem hitreje odpraviti.

3.6 Vzpostavitev avtomatskega testiranja

Pri avtomatizaciji vizualnega testiranja je treba določiti parametre, na katerih se testi izvajajo. Parametri so lahko med drugim seznam naprav, operacijskih sistemov, brskalnikov in različne verzije zadnjih dveh. Vzpostavitev in vzdrževanje velikega nabora okolij sta lahko težka ter draga.

Z vidika razvijalcev spletnih strani bi bilo idealno, da bi lahko teste izvedel hitro in med razvojem. Vendar bi, tudi če bi bilo to možno, še vedno morali imeti na voljo vse naprave, kar bi lahko bilo drago. Pri tem razvijalec lahko uporablja posnemovalnike dejanskih naprav ali nabor parametrov za potrebe razvoja zmanjša na praktično raven. Celovito testiranje pa se izvede na testnih strežnikih ob določeni periodi oz. ob spremembah. Takšno testiranje je lahko del zvezne integracije spletne strani.

V primeru regresijskega testiranja se posnetek zadnjega uspešnega testa shrani v repozitorij testov, kjer je posnetek na voljo za primerjavo. Ta posnetek je skupaj s postopkom zajema slike del testa. V primeru inicialnega posnetka je test neuspešen, zato sta potrebna ročni pregled posnetka in potrditev odgovorne osebe. S potrditvijo se posnetek shrani v repozitorij in testiranje se ponovi. S tem test postane uspešen. Kljub temu, da je v takšnem primeru potreben ročni poseg, je zajem posnetka še vedno avtomatiziran. Z ustreznim sistemom vrnitve rezultata testov pa je ročni del poenostavljen.

Orisali smo postopek avtomatskega vizualnega testiranja spletnih strani. Z vidika TDD sledimo zahtevam, saj za tovrstno testiranje velja, da najprej napišemo test, ki je neuspešen. To naredimo tako, da vključimo izbrano stran v primerjavo. Pri regresijskem testiranju moramo prvotno ročno preveriti izgled strani in ga potrditi. Pri tem nas testi ne vodijo do končnega rezultata. To bi lahko popravili tako, da bi imeli izgled strani pripravljen vnaprej. V primeru preverjanja postavitve elementov pa lahko teste napišemo kot običajno in sledimo ciklom razvoja po TDD-metodologiji.

3.7 Možne aplikacije vizualnega testiranja

Testiranje izgleda ne zadeva samo razvijalcev spletnih strani, ampak tudi druge vpletene v delovanje strani. Po vlogah pri delovanju strani se lahko preverjajo različne pogledi na ustrezen izgled:

- Razvijalci brskalnikov prek regresije preverjajo spremembe v delovanju različic brskalnikov [11].
- Razvijalci spletnih strani ali posameznih delov spletnih strani lahko preverjajo izgled strani na različnih konfiguracijah brskalnikov, naprav in resolucij.
- Oddelek za zagotavljanje kakovosti lahko redno preverja izgled strani glede na zahteve, ki jih ima.
- Skrbnik spletne strani pa lahko redno preverja izgled strani na najnovejših brskalnikih in napravah.
- Podjetja, ki nudijo spletno oglaševanje, lahko preverjajo, da so njihovi oglasi vidni na pravih mestih in v pravi obliki. V primeru nizke hitrosti vzpostavljanja stanja strani lahko preverjajo, ali so oglasi vidni že vnaprej.

Kadar smo v okolju brez izvajanja avtomatskega vizualnega testiranja, imamo lahko koristi z njegovo delno izvedbo. Že postavitev prvih dveh korakov lahko omogoči precejšnji prihranek časa, saj ni treba izvajati scenarijev ročno. Samo korake od ocenitve izgleda naprej je treba izvesti ročno. Na takšen način je možno vizualno testiranje vpeljati v več ločenih korakih.

3.8 Izvedba

Z vidika izvedbe sta problematična koraka vzpostavitve stanja strani v brskalniku in zajem izgleda strani, saj sta povezana s komunikacijo z brskalnikom. Ostali koraki se lahko izvajajo izven brskalnika v okolju, ki nam najbolj ustreza.

Za potrebe vzpostavljanja stanja strani v brskalniku in zajema posnetka potrebujemo orodje za upravljanje brskalnika. V ta namen ima organizacija W3C v fazi osnutka vmesnik WebDriver [36]. Osnovni namen vmesnika je omogočanje razvijalcem pisanje testov, ki vodijo brskalnik. Pred tem je bilo možno upravljati brskalnik:

- s spremembo strani ali
- s posebno kodo, ki je upravljala posamezen brskalnik.

Pri spremembi strani gre za vključitev dodatnih virov Javascript, ki izvajajo operacije na dejanski strani, ki je predmet testiranja. Takšen način je podvržen omejitvam, ki jih lahko Javascript izvaja v brskalniku.

Pri posebni kodi pa je slabost v tem, da je koda zmožna upravljati samo točno določen brskalnik ali celo točno določeno različico brskalnika.

Ravno standardizacija komunikacije z brskalnikom prinaša možnost poenotenega testiranja v brskalnikih na enak način, kot bi to delali ročno, ne glede na brskalnik, na katerem izvajamo teste.

Pri zajemu izgleda strani predvideva vmesnik WebDriver zajem bitne slike. Po drugi strani pa je za dostop do DOM-podatkov potreben poseben klic, ki prenese vsebino v tekstovni obliki.

Prek dodatne vključitve virov Javascript je zajem bitne slike neizvedljiv, saj takšne funkcionalnosti ni. Dostop do DOM-podatkov pa je poenostavljen, saj se skripta izvaja na brskalniku in je s tem analiza lažja.

Za posamezni brskalnik vedno obstaja tudi specifična možnost implementacije komunikacije. Tovrstna rešitev je problematična zaradi neenotne kode za vsak brskalnik posebej. Vmesnik WebDriver opravlja ravno to nalogo, saj v ozadju implementira komunikacijo z vsakim brskalnikom posebej.

4 Pregled in primerjava orodij ter storitev

Na voljo imamo množico knjižnic, ogrodij in celovitih rešitev, ki naslavljajo problematiko avtomatskega vizualnega testiranja spletnih strani s podporo različnih korakov. V tem poglavju bomo opisali to podporo z vidika definicije postopka testiranja in dodatne značilnosti, ki so za to delo pomembne.

Najprej bomo pregledali orodja za upravljanje z brskalniki. V drugem podpoglavju bomo našteli orodja, ki omogočajo vizualno testiranje spletnih strani. V tretjem podpoglavju bomo pregledali komercialne storitve v oblaku, ki nudijo širok nabor konfiguracij.

4.1 Upravljanje brskalnika

Upravljanje brskalnika je nujni element vizualnega testiranja za izvedbo naslednjih korakov:

- vzpostavitev stanja strani v brskalniku in
- zajem izgleda strani.

Zaradi velikega števila možnih konfiguracij je standardizacija upravljanja korak, ki nudi rešitve na dolgi rok.

4.1.1 Selenium

Selenium [26], [33] je odprtokodni projekt s široko skupnostjo. Je osrednje orodje avtomatskega testiranja spletnih aplikacij. Orodje je namenjeno upravljanju brskalnikov. Napisano je v Javi, kar zagotavlja izvajanje na vseh operacijskih sistemih, ki jih ta podpira. Aplikacijski vmesnik, ki je na voljo v Javi, ima implementacije svojega vmesnika tudi v drugih jezikih:

- MS .NET,
- Python,
- JavaScript in
- Ruby.

Izvajanje testov v vseh teh tehnologijah je možno brez dodatnih omejitev. Tako uporabnik lahko uporabi testna ogrodja, ki mu najbolje ustrezajo.

Upravljanje brskalnikov zagotavlja vmesnik WebDriver. Priloga A navaja seznam metod, ki jih vmesnik predvideva, in orisuje, kako izvedemo zajem bitne slike ter DOM-podatkov.

Podpora brskalnikom je široka in je odvisna od implementacij WebDriverja za posamezen brskalnik. Tabela 5 našteva seznam trenutno podprtih brskalnikov in operacijskih sistemov. V

primeru Androida in iOS-a gre za implementacijo WebDriverja, ki ga je možno upravljati, hkrati pa tudi preverjati aplikacije na teh sistemih. To kaže na splošnost vmesnika WebDriver.

Brskalnik oz. operacijski sistem
Google Chrome
Internet Explorer
Firefox
Safari
Opera
HtmlUnit
phantomjs
Android
iOS

Tabela 5 Seznam brskalnikov z implementacijo vmesnika WebDriver

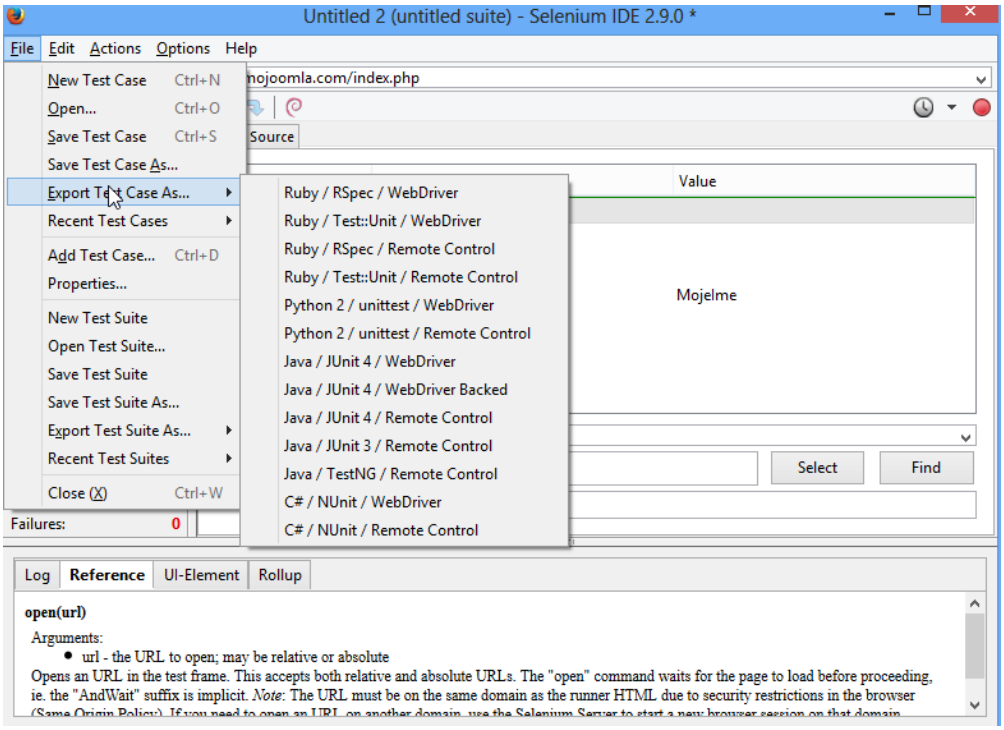
Konkretno je vmesnik WebDriver za upravljanje z operacijskimi sistemi omogočen z ločenimi projekti, ki to nudijo:

- Appium [15],
- Selendroid [32] in
- ios driver [27].

Appium nudi vmesnik do operacijskih sistemov Android in iOS, medtem ko Selendriod nudi dostop samo do Andrioida, ios driver samo do iOS-a.

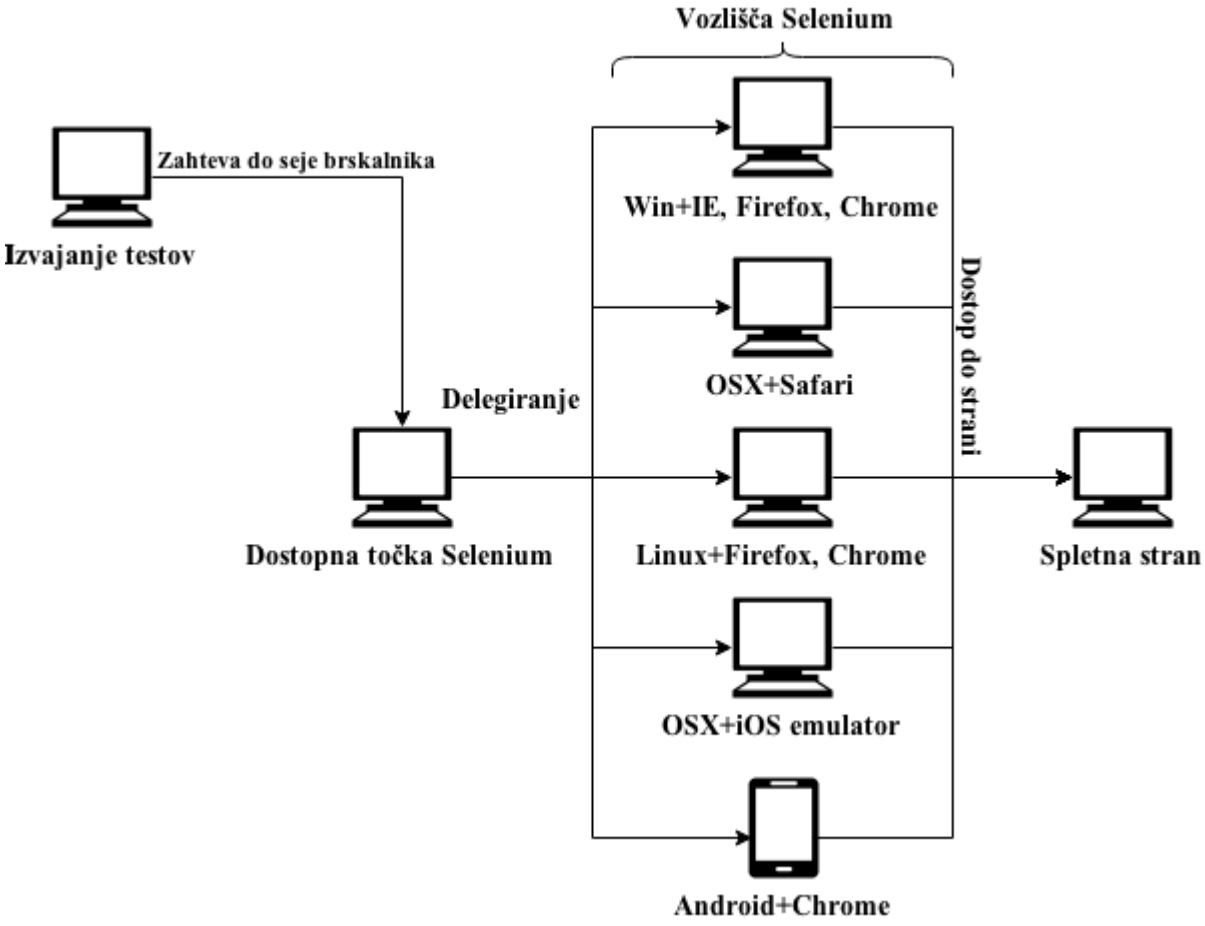
Trenutna različica Selenium 2 je različica, ki uporablja vmesnik WebDriver. Različica 1 oz. RC izvaja upravljanje brskalnika z vključitvijo dodatnih virov Javascript. Poleg tega vsebuje Selenium še dodatna orodja, s katerimi se ogrodje dopolnjuje in olajšuje delo z njim.

Selenium IDE je vtičnik za brskalnik Firefox, ki omogoča generiranje scenarijev prek uporabe brskalnika, njihovo ponavljanje in izvoze Selenium skript v različne kombinacije jezikov ter testnih ogrodij, kot kaže Slika 4. Gre za prototipe testnih scenarijev, ki lahko bistveno pohitrijo razvoj testov. Te specifikacije testov je sicer treba dopolniti, vendar uporaba takšnega orodja olajša delo.



Slika 4 Izgled Selenium IDE, različica 2.9.0, s prikazom možnih izvozov

Selenium Grid omogoča povezavo več vozlišč Selenium (*nodes*) v skupno dostopno točko (*hub*). Odjemalec tako dostopa do dostopne točke. Dostopna točka pa ureja komunikacijo do vozlišč, kot kaže Slika 5. To orodje nudi enostavno skalabilnost glede na potrebe testnega okolja, enotno točko dostopa do vseh konfiguracij in možnost vzporednega testiranja. Zadnja točka je pomembna zaradi počasnosti izvajanja testov na pravih brskalnikih. S tem lahko občutno zmanjšamo trajanje izvajanja testov, kadar izvajamo teste na več brskalnikih. Testi dostopajo do vozlišč tako, da pri zahtevi navedejo parametre, ki so pomembni: brskalnik, različica, operacijski sistem. Dostopna točka pa določi, katera vozlišča temu ustrezajo.



Slika 5 Primer arhitekture pri uporabi Selenium Grid

4.1.2 CasperJS

CasperJS [19] je ogrodje za upravljanje z brezobličnima brskalnikoma PhantomJS in SlimmerJS. Omogoča običajno navigacijo in manipulacijo s stranjo, kot smo jo opisali pri Seleniumu. CasperJS je napisan v jeziku Javascript in se izvaja v brskalniku, kjer se izvaja test. Dokler PhantomJS ni podpiral vmesnika WebDriver, je bil to edini način za izvajanje avtomatskega testiranja na brezobličnih brskalnikih, ki so primerni ravno zaradi svoje hitrosti in enake osnove kot pravi brskalniki. PhantomJS tako temelji na WebKit, ki je osnova za Safari. SlimmerJS pa temelji na Gecko, ki je osnova za Firefox. Prav tako omogoča zajem bitnih slik. Dostop do DOM-podatkov omogoča na enak način kot Selenium.

4.1.3 Watir

Watir [35] je odprtokodna knjižnica, napisana v jeziku Ruby, in omogoča upravljanje brskalnikov. Watir implementira svoj vmesnik za upravljanje. Obstajata Watir Classic, ki omogoča upravljanje brskalnika Internet Explorer, in Watir WebDriver, ki implementira vmesnik WebDriver ter s tem omogoča dostop do drugih brskalnikov.

4.1.4 Sahi

Sahi [17] je ogrodje, napisano v Javi, in je namenjeno upravljanju z brskalniki. Obstajata odprtokodna različica Sahi OS in komercialna Sahi Pro.

Deluje tako, da postavi posrednika (*proxy*) med testno ogrodje in brskalnik. Pri tem posrednik dopolni stran z dodatnimi viri Javascript, ki skrbijo za upravljanje brskalnika. Ko bo vmesnik WebDriver v fazi priporočila, bodo pripravili tudi podporo temu.

Vsebuje orodje Sahi Pro Controller, ki je namenjeno snemanju in izvajanju testov. Orodje služi začetni postavitvi testov in deluje v vseh brskalnikih ter operacijskih sistemih. Za identifikacijo elementov strani uporablja vmesnik izražanje, kot so blizu, pod, nad itd. Pri Sahi trdijo, da je takšen način enostavnejši in hitrejši kot prek poizvedb XPath.

Testni scenariji se pišejo v jeziku Sahi Script, ki je v bistvu Javascript, ki se izvaja v procesu ogrodja in omogoča klice ogrodja Java. Pri izvajanju scenarijev ni potrebne posebne kode za čakanje na vzpostavitev stanja.

Sahi podpira izvedbo posnetka strani, vendar je zaradi izvajanja v brskalniku prek Javascripta metoda nekakšna izjema v delovanju. Pred zajemom je treba brskalnik postaviti v ospredje in izvesti posnetek z drugim orodjem. Tako se ne izvede posnetek strani, ampak kar celega brskalnika z vsemi elementi. Metoda ni podprta v Sahi OS.

4.2 Vizualno testiranje

Orodja za vizualno testiranje izvedejo naslednja koraka:

- ocenitev izgleda in
- vrnitev rezultata.

Običajno nudijo dostop prek orodij za upravljanje brskalnikov tudi do prvih dveh korakov tako, da so orodja s pogleda vizualnega testiranja spletnih strani celovita. Običajno je možno s konfiguracijo rezultatov te prilagoditi potrebam strežnikom za zvezno integracijo (*continuous integration*). S tem se omogoči tudi korak avtomatizacije vizualnega testiranja.

4.2.1 Wraith

Wraith [18] je orodje, namenjeno vizualnemu testiranju spletnih strani. Napisano je v jeziku Ruby. Pripravila ga je razvojna ekipa spletne strani bbc-news, ki je s pomočjo tega orodja izvedla prepis kode strani v celoti.

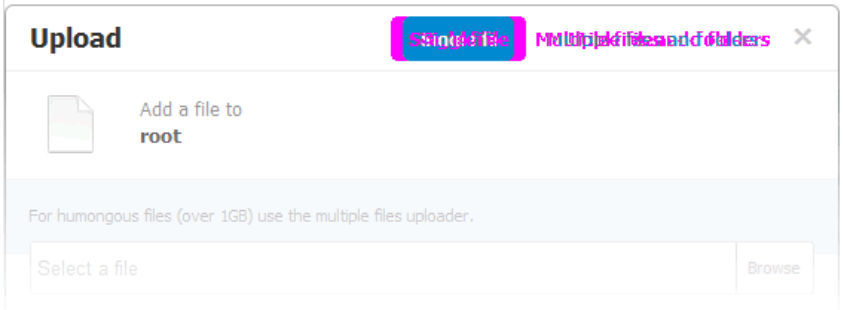
Wraith se povezuje z brskalnikoma PhantomJS in SlimmerJS prek orodja za upravljanje brskalnikov CasperJS. Za primerjavo posnetkov pa uporablja program ImageMagick [25].

Omogoča primerjavo posnetkov med dvema naslovoma ali primerjavo s shranjenim posnetkom. Slika 6 prikazuje primer prikaza razlik, kjer so razlike poudarjene z modro barvo.

Slika 6 Primer prikaza razlik v orodju Wraith [18]

4.2.2 PhantomCSS

PhantomCSS [24] je orodje, napisano v Javascriptu, ki za svoje izvajanje uporablja orodje za upravljanje brezobličnih brskalnikov CasperJS. Za primerjavo posnetkov uporablja knjižnico Javascript Resemble.js [30]. PhantomCSS se izvaja v brskalniku, na katerem se izvaja test. Prikazuje primer prikaza razlik, kjer rožnata barva označuje razlike.

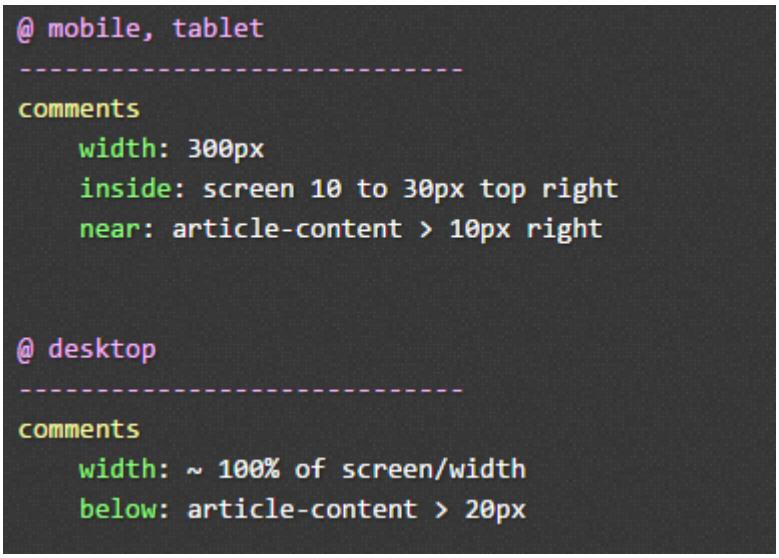


Slika 7 Primer prikaza razlik v orodju PhantomCSS

4.2.3 Galen Framework

Ogrodje Galen [22] je pri vizualnem testiranju šlo korak dlje. Poleg primerjave posnetkov strani omogoča tudi testiranje postavitve elementov glede na druge elemente in v ta namen definira poseben format testnih specifikacij, ki omogočajo preverjanje izgleda strani.

Napisano je v Javi in uporablja vmesnik Selenium za upravljanje brskalnikov. Ogrodje predvideva poizvedbe elementov in zajem posnetkov. Z dodatkom Galen Pages pa je možno upravljati brskalnik prek Seleniuma. Ker Galen uporablja Selenium, je možno brskalnike upravljati prek Selenium Grida.



Slika 8 Primer specifikacije za Galen Framework [22]

Slika 8 prikazuje primer specifikacije, ki je ločena na dva dela. Prvi del se izvaja na mobilnih napravah, drugi pa na namiznih. Pri tem pravi, da morajo elementi, ki jih pozna pod imenom *comments,* imeti naslednje lastnosti.

Za mobilne naprave:

- Širina: 300 pik
- Znotraj ekrana z razmikom od 10 do 30 pik od zgoraj in z desne strani
- Desno od elementa *article-content* več kot 10 pik

Za namizne naprave:

- Širina približno 100 % ekrana
- Podelementa *article-content* več kot 20 pik

S takšnim izražanjem opisujemo lastnosti elementov in odnosne med njimi, ki se nam zdijo pomembni za izgled. Ločimo jih lahko za različne postavitve, ki jih predvidevamo.

4.2.4 AppliTools Eyes

Podjetje AppliTools nudi orodje AppliTools Eyes [16], ki se ukvarja z vizualno primerjavo strani. Orodje je možno uporabljati ročno prek svojega brskalnika, tako da v brskalnik naložimo vtičnik, s katerim potem dostopamo do storitev primerjave. Trenutno je vtičnik na voljo samo za brskalnik Chrome. Orodje deluje tako, da naredimo posnetek na izbrani resoluciji in ga

označimo za referenčnega. Po naslednjem zagonu pa nudi primerjavo posnetkov glede na različne elemente, to so:

- celotna stran,
- postavitev strani in
- vsebina strani.

```
public static void Main(string[] args)
{
    IWebDriver driver = new FirefoxDriver();

    // This is your api key, make sure you use it in all your tests.
    var eyes = new Eyes();
    eyes.ApiKey = "[redacted]";

    try
    {
        // Start visual testing with browser viewport set to 1024x768.
        // Make sure to use the returned driver from this point on.
        driver = eyes.Open(driver, "Applitools", "Test Web Page", new Size(1024, 768));

        driver.Navigate().GoToUrl("http://www.applitools.com");

        // Visual validation point #1
        eyes.CheckWindow("Main Page");
```

Slika 9 Primer integracije z orodjem AppliTools Eyes prek Seleniuma

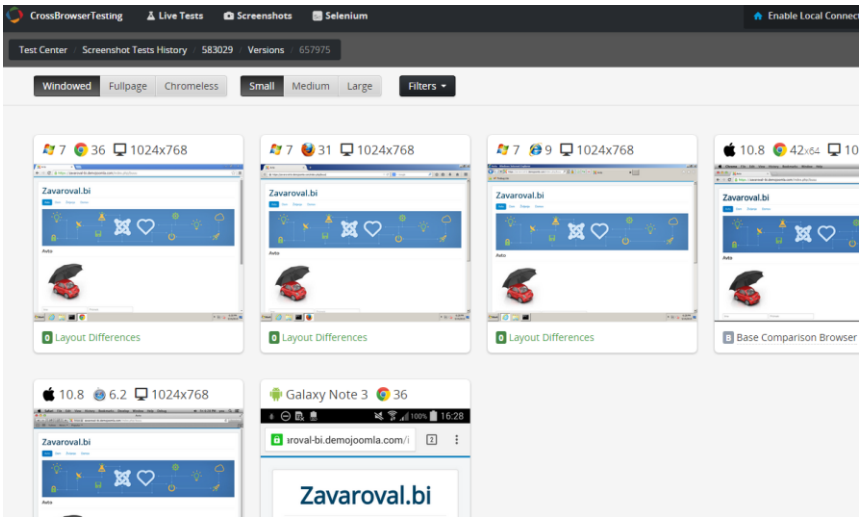
Poleg ročnega preverjanja nudi podjetje tudi integracijo z različnimi jeziki in ogrodji, ki jih uporabimo za povezovanje na njihov vmesnik za testiranje. Postopek deluje tako, da za različna ogrodja nudijo knjižnice za uporabo, ki jih potem uporabimo. V primeru testov Selenium kreiramo instanco WebDriver kot običajno in jo nato podamo knjižnici ApplTools Eyes, kot to prikazuje Slika 9.

4.3 Storitve uporabe konfiguracij

Z vidika vizualnega testiranja je izvedba testov na pravih konfiguracijah smiselna, vendar zaradi cene postavitve in vzdrževanja takšnega okolja težko dostopna. V takšni situaciji lahko uporabimo komercialne rešitve podjetij, ki nudijo že pripravljena okolja.

Pri storitvah uporabe konfiguracij gre za nudenje dostopa v različnih oblikah do brskalnikov na različnih napravah in simulatorjih. Oblike dostopov so splošne in običajno zajemajo naslednje storitve, ki so pomembne za izvajanje vizualnega testiranja spletnih strani:

- zajem izgleda strani (Slika 10 prikazuje primer izvedbe zajema),
- primerjava izgleda strani glede na izbrano konfiguracijo,
- ročno upravljanje brskalnika in
- dostop do Selenium Grida.



Slika 10 Primerjava izgleda strani na crossbrowsertesting.com

Vse storitve nudijo na velikem številu konfiguracij in nudijo orodja za dostop do lokalnega okolja tako, da je možno dostopati do strani v notranjem omrežju. S tem je možno storitve uporabljati tudi v razvojnih fazah, ko strani niso javno dostopne.

Primeri takšnih storitev so:

- Cross Browser Testing [20],
- Browser Stack [21] in
- Sauce Labs [31].

Tabela 6 prikazuje primerjavo teh storitev.

	Cross Browser Testing	Browser Stack	Sauce Labs
Število konfiguracij	1000+	700+	500+
Dostop do pravih mobilnih naprav	Da	Ne	Ne
Omogoča dostop do notranjega omrežja	Da	Da	Da
Ročno testiranje	Da	Da	Da
Podpira Selenium Grid	Da	Da	Da

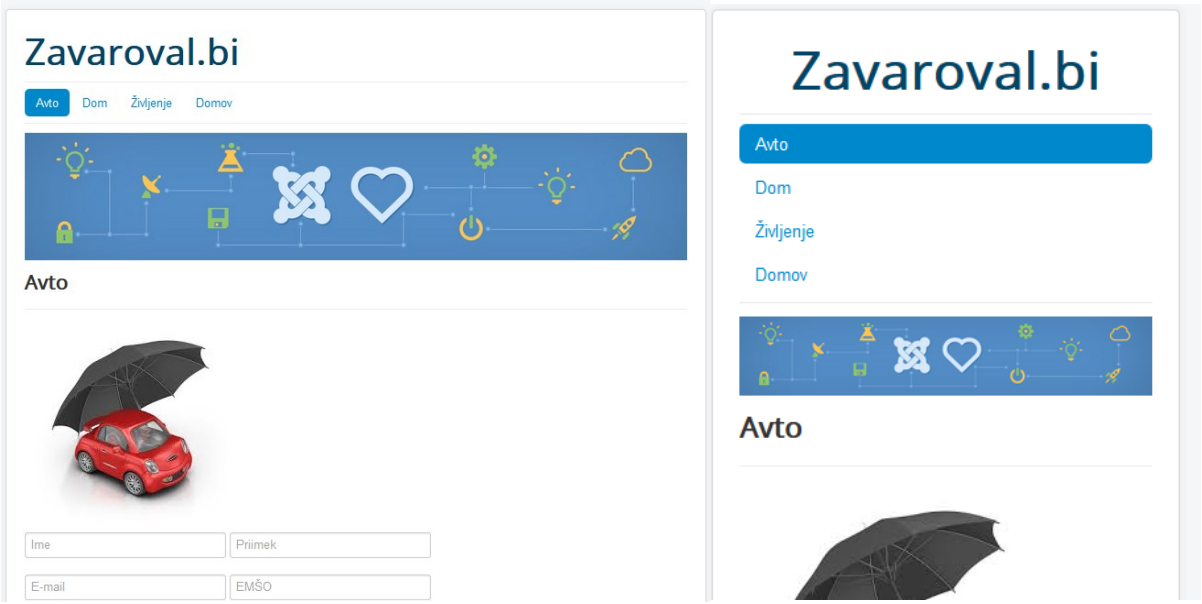
Tabela 6 Primerjava storitev konfiguracij

Poleg omenjenih storitev nudijo nekatera podjetja tudi orodja za vizualno primerjavo. Primerjava se izvaja na podlagi izbrane referenčne konfiguracije. Na ostalih konfiguracijah pa prikažejo morebitne razlike. CrossBrowserTesting tako nudi prikaz primerjave izgleda strani glede na referenčno konfiguracijo. S tem je možno zaznavati razlike v izgledu med različnimi konfiguracijami.

5 Izvedba na primeru

Za spletno stran, pripravljeno za potrebe tega dela, želimo izvesti regresijsko testiranje izgleda celotne strani. Pri tem bomo preverjali, da so elementi urejeni glede na resolucijo naprave in da je osnovna postavitev narejena po zahtevah. Poleg tega bomo preverjali posamezne elemente strani ločeno s primerjavo bitne slike.

Slika 11 prikazuje stran, ki jo preverjamo, na različnih resolucijah. Gre za prodajno stran zavarovalniških produktov. Sestavljena je iz glave, navigacije in osrednje strani za sklepanje posameznih tipov zavarovanj. Na manjši resoluciji se glava poravna sredinsko. Navigacija pa se iz vodoravne ureditve preuredi v navpično. Izvedli bomo primerjavo za sklepanje avtomobilske zavarovalne police.



Slika 11 Izgled strani za izvedbo na primeru

5.1 Določitev orodja in metod

Pri izbiri orodij smo sledili opisom iz poglavja 4. Glede na to, da želimo preverjati tudi postavitev elementov, smo se odločili za *Galen Framework (Galen)*, ki to omogoča. *Galen* omogoča poleg preverjanja postavitev tudi zajem in primerjavo bitnih posnetkov.

Pred izdelavo specifikacij se na eksperimentu prepričamo, kako se obnašajo primerjave:

- med različnimi brskalniki na isti strani,
- med različnimi operacijskimi sistemi na isti strani,
- pri majhni spremembi velikega elementa na istem okolju,

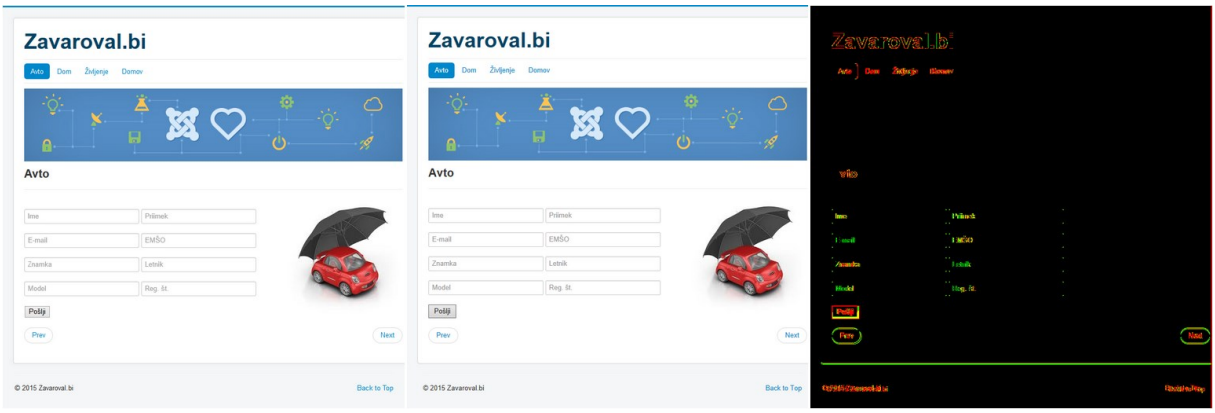
- pri veliki spremembi majhnega elementa.

Obe vrsti eksperimentov bomo smiselno izvedli na primerjavi bitnih posnetkov in preverjanju postavitve. Na podlagi rezultatov bomo izbrali metode primerjave in določili, kakšna odstopanja bomo pri tem dovoljevali.

5.2 Obnašanje primerjave bitnih slik

5.2.1 Primerjava med različnimi brskalniki

Najprej izvedemo primerjavo bitnega posnetka. Slika 12 prikazuje primerjavo posnetka iste strani med brskalnikoma Firefox (levi del) in Internet Explorer (srednji del). Na pogled enaki vizualizaciji imata pri primerjavi na nivoju pik veliko razlik. Poročilo primerjave pravi, da gre za skoraj 2 % odstopanje.



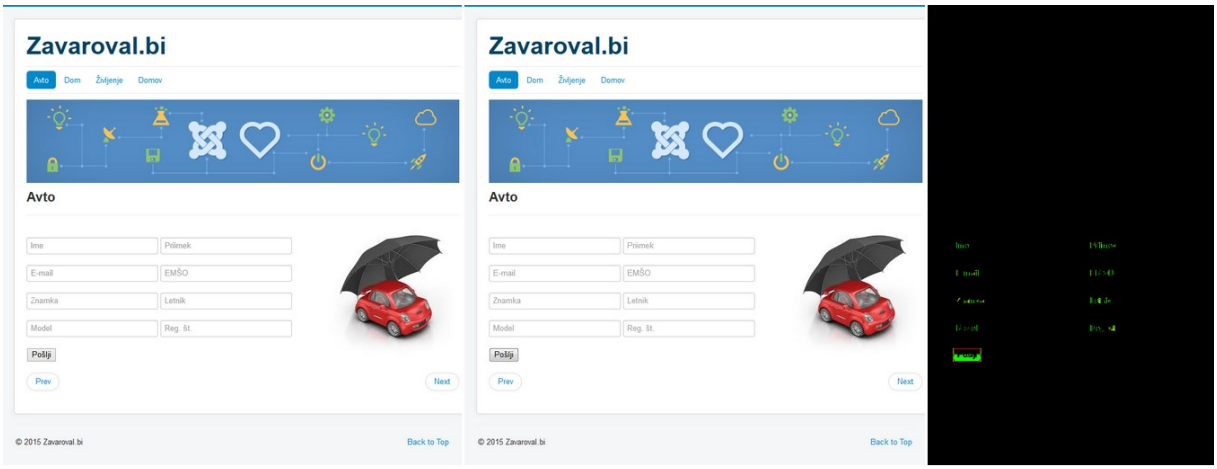
Slika 12 Primerjava posnetka med brskalnikoma Internet Explorer in Firefox

Razlike so v vseh elementih, razen v bitnih slikovnih virih. Napaka se od zgoraj navzdol tudi sešteva. Presenetljiva je tudi razlika, označena na skrajnem desnem robu, ki označuje razliko v velikosti posnetka.

Primerjava v takšni obliki je neuporabna za avtomatizacijo primerjave, saj je že na prikazanem primeru enostavne strani razlika razpršena po celotni strani. Tu je treba poseči po zahtevnejših algoritmih, ki bi upoštevali posamezne elemente, in na njih izvajati primerjavo skupaj z odstopanjem. Izvajanje vizualne primerjave bomo glede na rezultate izvajali za vsak brskalnik posebej.

5.2.2 Primerjava med različnimi operacijskimi sistemi

Slika 13 prikazuje primerjavo posnetka na brskalniku Firefox na drugem sistemu. Levo je Windows 8, desno pa Windows 7. Rezultati pri tem so boljši (0,2 %), saj se razlike pojavljajo samo na vnosnih elementih. Brskalniki se pri teh elementih zanašajo na sistemske nastavitve, ki se v tem primeru razlikujejo.



Slika 13 Primerjava izgleda celotne strani na istem brskalniku

Kljub vsemu je treba biti pozoren, kadar bi izvajali primerjavo med različnimi sistemi.

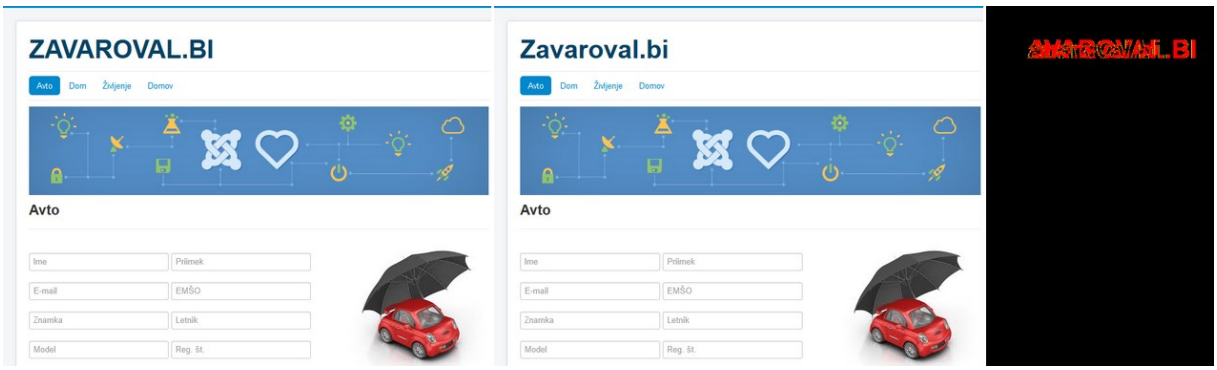
5.2.3 Primerjava pri majhnem premiku velikega elementa

Slika 14 prikazuje primerjavo posnetka po premiku slike za 5 pik. Iz poročila je razvidno, da ta premik povzroči 2 % razliko glede na celotno stran, kar je nesorazmerno z velikostjo napake. Človeško oko takšne razlike na pogled sploh ne opazi.

Slika 14 Primerjava posnetka premika slike

5.2.4 Primerjava pri veliki spremembi majhnega elementa

Slika 15 prikazuje primerjavo posnetka, kjer smo spremenili zapis naslov v velike črke. Napaka je za človeško oko zelo opazna. Poročilo primerjave pa pokaže, da gre zgolj za 0,8 % razliko.



Slika 15 Primerjava posnetka spremembe črk naslova

5.3 Obnašanje primerjave postavitve

Preverjanje postavitve strani je lahko primernejše za nekatere teste, ker je bližje definiciji izgleda strani. Pri tem se osredotočimo na podatke o lokaciji in velikosti posameznih elementov. Za potrebe splošne primerjave moramo določiti oceno razlike glede na razlike v podatkih. Oceno določimo z naslednjimi zahtevami.

1) Ločeno želimo spremljati spremembo lokacije in velikosti elementa. Če gre za premik elementa, ki ima enako velikost, želimo upoštevati samo premik.
2) Ker so elementi znotraj drugih, ne želimo, da se napaka sešteva. To pomeni, da premika nadelementa ne želimo šteti tudi za podelemente.
3) Majhne spremembe na velikih elementih želimo manj izpostaviti kot majhne spremembe na majhnih elementih. To pomeni, da mora ocena upoštevati spremembo relativno na velikost elementa.
4) Vsako dimenzijo želimo preverjati posebej, saj npr. sprememba v širini na podolgovatem elementu predstavlja drugačno spremembo kot pri višini.
5) Spremembo na manjši strani želimo bolj izpostaviti, kot na večji.

Določimo oceno, kjer so:

- x_i, y_i, h_i, w_i – koordinati x in y ter višina in širina originalnega elementa i
- dx_i, dy_i, dh_i, dw_i – absolutne spremembe koordinat x in y ter višine in širine elementa i
- s_i – ocena primerjave elementa i
- h, w – višina in širina originalne strani
- dh, dw – absolutni spremembi višine in širine strani
- s – ocena primerjave strani

$$s_i = dx_i \frac{x_i}{w^2} + dy_i \frac{y_i}{h^2} + dw_i \frac{w_i}{w^2} + dh_i \frac{h_i}{h^2}$$

Enačba 1 Ocena primerjave postavitve elementa

$$s = \frac{dw}{w} + \frac{dh}{h} + \sum_{i=0}^{n} s_i$$

Enačba 2 Ocena primerjave postavitve strani

Enačba 1 prikazuje oceno primerjave elementa, ki ustreza zgornjim pogojem. Vrednost ocene elementa je med 0 in 1 dokler ni razlike v velikostih celotne strani. Enačba 2 prikazuje definicijo ocene strani, kjer se vsi elementi seštejejo. Doda pa se še ocena razlike velikost strani.

Z ogrodjem Galen pripravimo testno specifikacijo, kjer samo navedemo elemente in zaženemo generiranje poročila. Slika 16 prikazuje primer takšnega poročila. Z izbiro elementov *content* in *nav* se v desnem delu elementi označijo vizualno, v spodnjem delu pa so predlogi dopolnitve specifikacije iz katere razberemo za vse elemente glede na element *content*.

Slika 16 Poročilo testne specifikacije

Obdelane podatke zabeležimo v testno specifikacijo, ki je osnova za izvedbo testov. Slika 17 prikazuje izsek te specifikacije.

```
header
    inside: content  21px top, 21px left, 21px right, 593px bottom
nav
    inside: content  79px top, 21px left, 21px right, 537px bottom
image-sep
    inside: content  135px top, 21px left, 21px right, 388px bottom
title
    inside: content  285px top, 21px left, 21px right, 353px bottom
in-name
    inside: content  364px top, 21px left, 525px right, 270px bottom
in-mail
    inside: content  410px top, 21px left, 525px right, 224px bottom
in-mark
    inside: content  456px top, 21px left, 525px right, 178px bottom
```

Slika 17 Izsek testne specifikacije za primerjavo postavitve

Podatki testne specifikacije so namenoma preveč specifični za namene poskusa. V praksi se ne bi poslužili tako natančne specifikacije ampak bi dopuščali napako in preverjali elemente in odnose, ki so za postavitev strani pomembni.

5.3.1 Primerjava postavitve med različnimi brskalniki

Izvedba testa je enaka kot pri primerjavi bitne slike. Primerjava zazna razlike, ki dobijo oceno 0,012456. Tabela 7 prikazuje obdelane podatke za izvedbo ocene v formatu (x, y, w, h). Največja razlika se pojavi na elementu *in-send*, ki ima razliko v širini 3 pike.

Element	Dimenzije specifikacije	Dimenzije brskalnika Firefox
header	(21, 21, 724, 48)	(21, 21, 724, 48)
nav	(21, 79, 724, 46)	(21, 79, 724, 46)
image-sep	(21, 135, 724, 139)	(21, 135, 724, 139)
title	(21, 285, 724, 24)	(21, 286, 724, 24)
in-name	(21, 364, 220, 28)	(21, 365, 220, 28)
in-surname	(244, 364, 220, 28)	(245, 365, 220, 28)
image-car	(514, 337, 225, 224)	(515, 338, 225, 224)
in-mail	(21, 410, 220, 28)	(21, 411, 220, 28)
in-personalreg	(244, 410, 220, 28)	(245, 411, 220, 28)
in-mark	(21, 456, 220, 28)	(21, 457, 220, 28)
in-year	(244, 456, 220, 28)	(245, 457, 220, 28)
in-model	(21, 502, 220, 28)	(21, 503, 220, 28)
in-reg	(244, 502, 220, 28)	(245, 503, 220, 28)
in-send	(21, 548, 53, 26)	(21, 549, 50, 24)
prev	(21, 592, 57, 30)	(21, 591, 55, 30)
next	(688, 592, 57, 30)	(688, 591, 57, 30)

Tabela 7 Podatki postavitve elementov na različnih brskalnikih

5.3.2 Primerjava med različnimi operacijskimi sistemi

Izvedba testa je enaka kot pri primerjavi bitne slike. Primerjava ne zazna razlik na postavitvi.

5.3.3 Primerjava pri majhnem premiku velikega elementa

Izvedba spremembe je enaka kot pri primerjavi bitne slike. Pričakovano test zazna spremembo na slikovnem elementu kot premik za 4 pike v horizontali in 1 piko v vertikali. Glede na velikost strani ima ocena vrednost 0,003960.

5.3.4 Primerjava pri veliki spremembi majhnega elementa

Izvedba spremembe je enaka kot pri primerjavi bitne slike. Ker je konkreten element raztegnjen čez celotno širno strani, razlik primerjava ne znana, kar pomeni, da gre za lažen uspešen rezultat z vidika celotne strani. Z vidika primerjave postavitve pa pravilen rezultat. Težava je v tem, da ne preverjamo vsebine elementa, ampak samo njegovo postavitev.

5.4 Primerjava rezultatov in določitev odstopanj ter metod

Na podlagi primerjav naredimo primerjavo rezultatov in določimo smiselna odstopanja. Tabela 8 prikazuje primerjavo, kjer na metodah primerjave bitnega posnetka prikazujemo odstotek razlike glede na celoten posnetek. Pri primerjavi postavitve pa smo določili oceno.

Primerjava metod	Primerjava bitnega posnetka	Primerjava postavitve
Med različnimi brskalniki	2 %	0,012456
Med različnimi operacijskimi sistemi	0,2 %	0
Majhna sprememba velikega elementa	2 %	0,003960
Velika sprememba majhnega elementa	0,8 %	0

Tabela 8 Primerjava metod na scenarijih

Iz rezultatov primerjave bitnega posnetka sledi, da se ne moremo zanašati na primerjave med različnimi konfiguracijami saj so razlike prevelike. V kolikor bi na podlagi rezultatov dovoljevali 2 % odstopanje bi omogočili lažne uspešne rezultate, česar ne dovolimo. Iz zadnjih dveh rezultatov prav tako sledi, da se izračunana napaka odziva nesorazmerno z napako v videzu strani. S tega stališča je primerjava bitnih slik celotne strane neprimerna za primerjavo.

Iz rezultatov primerjave postavitve sledi, da razlika med brskalniki prav tako obstaja, vendar glede na ostale rezultate smiselna, saj z vklopom takšnega odstopanja ne bi omogočili lažnih uspešnih rezultatov. Ker preverjamo zgolj postavitev elementov, metoda ne zaznava sprememb njihove vsebine. Testne specifikacije bi lahko dopolnili tudi s testi barve in besedila elementa. S tem bi povečali obseg testiranja v takšnih testih. Galen omogoča preverjanje barve elementov tako, da naredi spektrogram elementa, preverjanje besedila elementa in preverjanje drugih CSS-lastnosti elementov. Takšne specifikacije bi bile že zelo specifične in bi bile same sebi namen.

Da bi testiranje obvladovali moramo uporabljati teste, za katere menimo, da lahko predvidimo obnašanje. Smiselno je tako izvajati kakršnokoli primerjavo bitne slike, kadar izvajamo testiranje na istem brskalniku in sistemu. To zahteva vodenje posnetkov za vsako takšno konfiguracijo. Slabost takšnega testiranja je v tem, da je treba ročno potrditi spremembe za vsako konfiguracijo posebej. Prednost je v celovitosti testa, saj lahko preverjamo poljubno velik del strani, slabosti pa so glede na dovoljena odstopanja ali pogosti lažni neuspešni rezultati ali možnost lažnih uspešnih rezultatov.

Pri testiranju odzivne zasnove spletne strani je smiselno izvajati teste enot ne glede na konfiguracijo. To je smiselno, ker je takšna zasnova namenjena ravno temu. Pri tem moramo definirati prag dovoljenega odstopanja. Takšno testiranje omogoča določitev odstopanja za

vsak element testiranja. Namesto koordinat postavitve je možno preverjati druge odnose med elementi.

Glede na ugotovljeno je smiselno preverjati teste postavitve na celotni strani z isto testno specifikacijo za vse konfiguracije hkrati. Specifikacije omogočajo testiranje na različnih resolucijah. Tako je potrebno za stran pripraviti samo eno specifikacijo. Ker testi postavitve ne preverjajo vsebine elementov je smiselno na posameznih elementih izvajati primerjavo bitnega posnetka. Ker obravnavamo vsak element posebej, je možno določiti dovoljeno odstopanje za vsakega posebej. Ker gre za manjše elemente, imamo boljši nadzor nad napakami, ki se lahko dogajajo med različnimi brskalniki.

5.5 Izvedba testiranja

Testiranje bomo izvedli na brskalnikih:

- PhantomJS,
- Firefox in
- Internet Explorer.

Za potrebe testiranja med razvojem bomo pripravili skripto, kjer se bodo testi izvajali samo na brskalniku PhantomJS. To je predvsem zaradi hitrosti. Za potrebe temeljitega testa pa bomo izvedli teste tudi na pravem brskalniku.

Teste bomo razdelili na:

- teste enot,
- teste postavitve in
- teste celotne strani.

S testi enot bomo podrobneje preverjali posamezne elemente strani tako, da si bomo poenostavili celovite teste. Teste bomo izvedli s pripravo strani, ki bodo vsebovale samo posamezne elemente. Kjer je smiselno, bomo izvedli tako preverjanje postavitve elementov kot tudi primerjavo izgleda posnetka. S testi postavitve strani bomo preverjali ustreznost postavitve glede na izbrano resolucijo. Pri testih celotne strani pa bomo primerjali posnetek glede na zadnji zagon.

Za specifikacijo testov pozna Galen datoteko .spec, ki je razdeljen na dva dela. V glavi opišemo elemente, ki jih želimo obravnavati, in podatke za njihovo iskanje. V vsebini specifikacije pa navajamo njihove lastnosti in odnose med njimi.

Pripravili smo naslednje specifikacije testov:

- Unit/footer.spec,
- Unit/title.spec,

- Unit/nav.spec,
- Layout.spec in
- Homepage.spec

Prve tri specifikacije so specifikacije enot, naslednja je specifikacija za postavitev, zadnja pa je specifikacija za preverjanje vsebine.

```
=================================================
header              css      header
nav                 css      .navigation
content             css      .container
body                css      .body
=================================================

@ *
---------------
content
    centered horizontally on: header

@ desktop
---------------
nav
    height: ~46px

@ mobile, tablet
---------------
header
    centered horizontally inside: content
    above: nav ~10px

nav
    width: 100% of header/width
    aligned vertically all: header
```

Slika 18 Vsebina specifikacije testa layout.spec

Slika 18 prikazuje vsebino specifikacijo testa layout.spec, kjer na primer preverjamo, da je vsebina poravnana sredinsko glede na glavo.

Za izvedbo specifikacij testov smo pripravili testne sklope (*Test Suites*), ki jih Galen predvideva. V testnih sklopih navedemo podrobnosti izvajanja specifikacij testov, kot kaže Slika 19:

- izvajanje testov (Selenium),
- brskalnik (vhodni parameter),
- resolucija (po tabeli) in
- testna specifikacija.

```
@@ set domain http://localhost/test/
@@ set test layout

@@ parameterized
    | deviceName | tags      | size     |
    | Mobile     | mobile    | 320x600  |
    | Tablet     | tablet    | 640x480  |
    | Desktop    | desktop   | 1024x768 |

@@ groups pagetests
Test ${test} on ${deviceName} device in ${browser} browser and size ${size}
    selenium ${browser} ${domain}${test}.html ${size}
        check "specs/${test}.spec" --include "${tags}"
```

Slika 19 Testni sklop za primer izvajanja testov postavitve strani

Za izvedbo testnega sklopa pa smo pripravili še ukazno datoteko, ki izvede klic, kot kaže Slika 20.

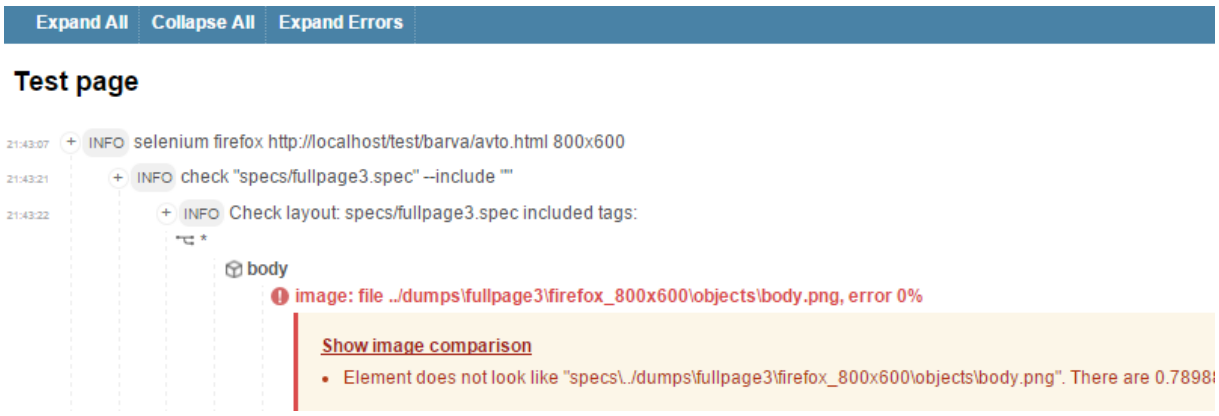
```
call galen.bat test "tests/layout.test" ^
  --htmlreport "reports/layout" ^
  --parallel-suites 4 ^
  -Dbrowser=phantomjs

call galen.bat test "tests/layout.test" ^
  --htmlreport "reports/layout" ^
  --parallel-suites 4 ^
  -Dbrowser=firefox
```

Slika 20 Ukazna datoteka za izvajanje testnega sklopa za postavitev strani

5.6 Rezultati

Prikazali smo primer postavitve avtomatskega vizualnega testiranja s primerom primerjave bitnih posnetkov in testiranja postavitve spletne strani.

V primeru odstopanja ogrodje izdela poročilo o napaki, kjer je možno razbrati, za kakšno napako gre. Poročilo je opisno in opiše, za kakšno napako gre ter v kateri testni specifikaciji. Pri tem izvozi poročilo v HTML-obliki, tako da je možno podatke tudi na enostaven način pregledati. Slika 21 prikazuje tak primer. Testne specifikacije so urejene v drevesno obliko. S klikom na povezavo pa se odpre tudi vizualni prikaz razlike.



Slika 21 Primer poročila o napaki

5.7 Problemi in možne izboljšave

Pri upravljanju brskalnika z vmesnikom WebDriver se med testiranjem kaže nestabilnost v delovanju. To za nekatere brskalnike velja bolj kot druge, kar kaže na konkretno implementacijo vmesnika. Zaradi tega so lahko testi tudi neuspešni, kar zahteva ponovni zagon. V takšnih primerih bi lahko teste izvedli ponovno, kar pa bi podaljšalo čas izvajanja.

Pri izvedbi bitnega posnetka nekateri brskalniki ne upoštevajo definicije vmesnika, da je treba zajeti vsebino celotne strani in ne samo vidnega dela. Ogrodje Galen predvideva rešitev v takšnih primerih z avtomatskim zajemom več slik in z njihovo združitvijo. Komplikacija je nepotrebna in samo uvaja dodatno nestabilnost pri že tako različnih delovanjih. Standard vmesnika WebDriverja (ali kakšnega drugega) je treba pripeljati v fazo priporočila. Razvijalci brskalnikov pa bi morali poskrbeti za ustrezno implementacijo vmesnika. To bi prineslo dodatno stabilnost pri upravljanju brskalnikov.

Pri zajemu DOM-podatkov iz brskalnika se moramo zanašati na lastnosti HTML-elementov za potrebe lociranja. Dopolnitev strani z dodatnimi podatki za potrebe testiranja lahko lociranje elementov olajša. Primer takšne dopolnitve bi bil dodaten atribut na elemente zanimanja, kot je na primer *<element test= "logično ime "/>*. S takšno dopolnitvijo bi si bistveno olajšali delo s testnimi specifikacijami in omogočili kasnejše preurejanje kode, saj se ne bi zanašali na lastnosti, ki so potrebne za izgled strani. Pri tem bi lahko vmesnik dopolnili tako, da bi olajšal pridobitev takšnih podatkov na enostaven način.

Pri primerjavi bitnih slik je največ možnosti za izboljšave. Človeški vid je za nekatere razlike bolj dojemljiv kot za druge. Algoritmi za primerjavo bi se morali zanašati na takšne lastnosti. Pri tem bi si lahko pomagali s podatki postavitve. Konkretnih javno dostopnih implementacij takšnih algoritmov nismo našli. Jih pa nudijo komercialne rešitve, ki postopkov ne opisujejo, ampak samo navajajo, katere probleme so s tem rešili. Poskus ponovitve primerjave pri majhnem premiku na orodju AppliTools Eyes pokaže razliko na enak način. Še vedno pa označuje to kot napako na celotnem elementu.

6 Sklepne ugotovitve

Avtomatsko vizualno testiranje spletnih strani potrebujemo zaradi velikega števila različnih konfiguracij, na katerih morajo spletne strani ustrezno izgledati. Podpora odzivni zasnovi spletnih strani omogoča enotno zasnovo strani za poljubne resolucije. Izgled spletne strani pa je treba preverjati na vseh konfiguracijah.

Pri tem nam pomaga avtomatsko vizualno testiranje spletnih strani, ki prek avtomatskega upravljanja brskalnikov omogoča takšno testiranje. Pri upravljanju brskalnikov je pomembno, da lahko uporabljamo prave brskalnike in da jih lahko upravljamo na enoten način. Pri tem izstopa vmesnik WebDriver, ki to omogoča. S testno specifikacijo moramo izvesti vzpostavitev stanja strani, ki je predmet testiranja in zajem podatkov iz brskalnika. Na voljo imamo bitni posnetek in DOM-podatke strani iz brskalnika.

Orodja za vizualno testiranje z zajetimi podatki preverjajo ustreznost izgleda. Pri tem uporabljajo primerjavo bitnih slik in obravnavo DOM-podatkov. Pokazali smo, da primerjava bitnih slik ni enostavna, saj brskalniki delujejo različno. Te razlike so za oko neopazne in se zaznajo šele s podrobnim pregledom. DOM-podatki nudijo dodatne podatke o strukturi strani, ki jih lahko uporabimo za preverjanje postavitve, ki je pomembna pri odzivni zasnovi spletnih strani.

Običajno so orodja za vizualno testiranje zgolj vmesnik za upravljanje brskalnika in orodjem za primerjavo posnetkov. Ogrodje Galen je to dopolnilo z lastnim jezikom testnih specifikacij postavitev strani. Z njimi je možno preverjati lastnosti izbranih elementov in odnose med njimi. Menimo, da je to novost pri opisu izgleda treba formalizirati in s tem po potrebi dopolniti. S tem bi takšen opis lahko uporabili povsod, kjer je postavitev pomembna, ne samo pri spletnih straneh.

Na primerih smo prikazali razlike, ki jih lahko prinašajo različni sistemi, brskalniki in drugi elementi, ki vplivajo na izgled strani. Pri tem kompenziramo razlike z dovoljenimi odstopanji, ki lahko pri nespametni uporabi omogočajo lažne pozitivne rezultate, kar je za avtomatsko testiranje nesprejemljivo. Ker gre pri spletnih straneh za strukturiran opis izgleda, bi pričakovali boljše ujemanje med brskalniki. Zato na izvedenem primeru razdelimo teste na manjše enote, kjer imamo boljši nadzor nad odstopanji. Menimo, da je uporabljeni pristop delitve testov na teste enot, teste postavitve in celovite teste analogija običajnemu testiranju enot, integracijskemu ter sistemskemu testiranju. Kljub temu je načrtovanje vizualnih testov naloga, ki ni enostavna. Ravno v delu opisa testov ima avtomatsko vizualno testiranje spletnih strani

posebnosti, saj je treba izgled zelo konkretno definirati in napisati omejitve samo tam, kjer so najbolj pomembne.

A Seznam metod vmesnika WebDriver

Vmesnik WebDriver – specifikacija je trenutno v fazi osnutka. Tabela 9 našteva seznam metod, ki jih predvideva.

HTTP metoda	URL	Opis
POST	/session	Nova seja
DELETE	/session/{sessionId}	Brisanje seje
POST	/session/{sessionId}/url	Odpiranje URL-naslova
GET	/session/{sessionId}/url	Trenutni URL-naslov
POST	/session/{sessionId}/back	Pojdi nazaj
POST	/session/{sessionId}/forward	Pojdi naprej
POST	/session/{sessionId}/refresh	Osveži
GET	/session/{sessionId}/title	Trenutni naslov strani
GET	/session/{sessionId}/window_handle	Trenutna ročica okna
GET	/session/{sessionId}/window_handles	Trenutne ročice okna
DELETE	/session/{sessionId}/window_handle	Zapri okno
POST	/session/{sessionId}/window/size	Nastavi velikost okna
GET	/session/{sessionId}/window/size	Trenutna velikost okna
POST	/session/{sessionId}/window/maximize	Razpri okno
POST	/session/{sessionId}/window/fullscreen	Nastavi okno v celozaslonski način
POST	/session/{sessionId}/window	Preklopi na okno
POST	/session/{sessionId}/frame	Preklopi na okvir strani
POST	/session/{sessionId}/frame/parent	Preklopi na nadokvir strani
POST	/session/{sessionId}/element	Najdi element
POST	/session/{sessionId}/elements	Najdi elemente
GET	/session/{sessionId}/element/{elementId}/ displayed	Je element viden
GET	/session/{sessionId}/element/{elementId}/ selected	Je element označen
GET	/session/{sessionId}/element/{elementId}/ attribute/{name}	Atributi elementa

GET	/session/{sessionId}/element/{elementId}/ css/{propertyName}	CSS-lastnosti elementa
GET	/session/{sessionId}/element/{elementId}/text	Besedilo elementa
GET	/session/{sessionId}/element/{elementId}/name	Značka elementa
GET	/session/{sessionId}/element/{elementId}/rect	Okvir elementa
GET	/session/{sessionId}/element/{elementId}/enabl ed	Je element omogočen
POST	/session/{sessionId}/execute	Izvedi skripto
POST	/session/{sessionId}/execute_async	Izvedi skripto asinhrono
GET	/session/{sessionId}/cookie/{name}	Vrni piškotek
POST	/session/{sessionId}/cookie	Dodaj piškotek
DELETE	/session/{sessionId}/cookie/{name}	Briši piškotek
POST	/session/{sessionId}/timeouts	Nastavi časovno omejitev
POST	/session/{sessionId}/actions	Akcije
POST	/session/{sessionId}/element/{elementId}/click	Klikni na element
POST	/session/{sessionId}/element/{elementId}/tap	Tapni element
POST	/session/{sessionId}/element/{elementId}/clear	Počisti element
POST	/session/{sessionId}/element/{elementId}/ sendKeys	Pošlji zaporedje tipk v element
POST	/session/{sessionId}/dismiss_alert	Zavrni opozorilo
POST	/session/{sessionId}/accept_alert	Sprejmi opozorilo
GET	/session/{sessionId}/alert_text	Besedilo opozorila
POST	/session/{sessionId}/alert_text	Pošlji opozorilo
GET	/session/{sessionId}/screenshot	Izvedi posnetek strani

Tabela 9 Seznam vmesnika WebDriver [36]

URL-naslovi imajo v zavitih oklepajih parametre, ki jih pridobimo z drugimi metodami. Tako pridobimo:

- *{sessionId}* s klicem metode *GET /session* in
- *{attributeId}* s klicem metode *GET /session/{sessionId}/element(s).*

Posnetek strani izvedemo s klicem metode *GET /session/{sessionId}/screenshot*. Po specifikaciji mora metoda vrniti posnetek celotne strani, ne samo vidnega dela.

Dostop do DOM-podatkov je možen prek metod z elementi. Poizvedovanje elementov je možno prek identifikatorja elementa, CSS-izbirnika (*CSS Selector*) ali prek XPath-poizvedbe. Za celovito strukturo pa je treba zagnati izvajanje skript in zahtevati podatke prek njih. To omejitev ima WebDriver, ker nastopa kot uporabnik brskalnika in nima direktnega dostopa do podatkov brskalnika.

Slike

Tabele

Enačbe

Literatura

[1] G. Adžić, "Specification by Example: How Successful Teams Deliver the Right Software", Shelter Island, N.Y: Manning Publications, 2011, str. xvii–xix.

[2] G. Adžić, (2013). *Let's break the Agile Testing Quadrants*. Dostopno na: http://gojko.net/2013/10/21/lets-break-the-agile-testing-quadrants/.

[3] T. Alspaugh, (2013). *Kinds of Software Quality (Ilities)*. Dostopno na: http://www.thomasalspaugh.org/pub/fnd/ility.html.

[4] D. Astels, "Test-Driven Development: A Practical Guide", Upper Saddle River, NJ: Prentice Hall, 2003, str. 5–43, 487–495.

[5] K. Beck, "Test-Driven Development: By Example", Boston: Addison-Wesley Professional, 2002, str. 1–14, 121–156.

[6] M. Cohn, "Succeeding with Agile: Software Development Using Scrum", Upper Saddle River, NJ: Addison-Wesley Professional, 2009, str. 307–324.

[7] L. Crispin in J. Gregory, "Agile Testing: A Practical Guide for Testers and Agile Teams", Upper Saddle River, NJ: Addison-Wesley Professional, 2009, str. 97–108.

[8] M. Fowler, (2011). *SubcutaneousTest*. Dostopno na: http://martinfowler.com/bliki/SubcutaneousTest.html.

[9] K. Naik in P. Tripathy, "Software Testing and Quality Assurance: Theory and Practice", Hoboken, N.J: Wiley-Spektrum, 2008, str. 1–30, 158–191.

[10] (2009). *Morgan Stanley Releases The Mobile Internet Report*. Dostopno na: http://www.morganstanley.com/about-us-articles/4659e2f5-ea51-11de-aec2-33992aa82cc2.html.

[11] (2011). *An overview of the QualityBots design.* Dostopno na: https://code.google.com/p/qualitybots/wiki/QualityBotsDesign.

[12] (2011). *Standard ECMA-262*. Dostopno na: http://www.ecma-international.org/publications/standards/Ecma-262.htm.

[13] (2012). *Media Queries - W3C Recommendation 19 June 2012*. Dostopno na: http://www.w3.org/TR/css3-mediaqueries/.

[14] (2014). *World Wide Web Consortium Process Document*. Dostopno na: http://www.w3.org/2014/Process-20140801/#Reports.

[15] (2015). *Appium: Mobile App Automation Made Awesome.* Dostopno na: http://appium.io/.

[16] (2015). *Applitools – Visual Test Automation*. Dostopno na: https://applitools.com/.

[17] (2015). *Automation Testing Tool For Web Applications*. Dostopno na: http://sahipro.com/.

[18] (2015). *BBC-News/wraith · GitHub*. Dostopno na: https://github.com/BBC-News/wraith.

[19] (2015). *CasperJS, a navigation scripting and testing utility for PhantomJS and SlimerJS*. Dostopno na: http://casperjs.org/.

[20] (2015). *Cross Browser Testing. Real mobile devices & browsers!*. Dostopno na: http://crossbrowsertesting.com/.

[21] (2015). *Cross Browser Testing Tool. 300+ Browsers, Mobile, Real IE.* Dostopno na: https://www.browserstack.com/.

[22] (2015). *Galen Framework | Automated testing of responsive design*. Dostopno na: http://galenframework.com/.

[23] (2015). *Headless browser*. Dostopno na: http://en.wikipedia.org/w/index.php?title=Headless_browser&oldid=657210951.

[24] (2015). *Huddle/PhantomCSS · GitHub*. Dostopno na: https://github.com/Huddle/PhantomCSS.

[25] (2015). *ImageMagick: Convert, Edit, Or Compose Bitmap Images*. Dostopno na: http://www.imagemagick.org/script/index.php.

[26] (2015). *Introduction — Selenium Documentation*. Dostopno na: http://www.seleniumhq.org/docs/01_introducing_selenium.jsp#brief-history-of-the-selenium-project.

[27] (2015). *ios-driver documentation*. Dostopno na: https://ios-driver.github.io/ios-driver/.

[28] (2015). *Make Your Website Mobile Friendly, Or Else!*. Dostopno na: http://www.endertechnology.com/blog/make-your-website-mobile-friendly-or-face-extinction-on-google.

[29] (2015). *PhantomJS*. Dostopno na: http://phantomjs.org/.

[30] (2015). *Resemble.js : Image analysis*. Dostopno na: http://huddle.github.io/Resemble.js/.

[31] (2015). *Sauce Labs: Selenium Testing, Mobile Testing, JS Unit Testing and More*. Dostopno na: https://saucelabs.com/.

[32] (2015). *Selendroid: Selenium for Android*. Dostopno na: http://selendroid.io/.

[33] (2015). *Selenium (software)*. Dostopno na: http://en.wikipedia.org/wiki/Selenium_(software).

[34] (2015). *StatCounter Global Stats*. Dostopno na: http://gs.statcounter.com/#browser-SI-monthly-201501-201504-bar.

[35] (2015). *Web Application Testing in Ruby*. Dostopno na: http://watir.com/.

[36] (2015). *WebDriver*. Dostopno na: https://w3c.github.io/webdriver/webdriver-spec.html.

[37] (2015). *World Wide Web Consortium (W3C)*. Dostopno na: http://www.w3.org/.