UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

GREGOR STAMAĆ

ORODJE ZA TESTNO VODEN RAZVOJ JAVASCRIPT APLIKACIJ

DIPLOMSKO DELO NA UNIVERZITETNEM ŠTUDIJU

MENTOR: PROF. DR. SAŠA DIVJAK

LJUBLJANA, 2015

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

Orodje za testno voden razvoj JavaScript aplikacij

Tematika naloge:

Zasnujte orodje za testiranje JavaScript kode. Orodje naj nam pomaga pri testno vodenem razvoju JavaScript aplikacij. Najprej opišite okolje JavaScript in jezik TypeScript, ki je nadgradnja jezika JavaScript. Opišite pravila, prednosti in slabosti testno vodenega razvoja. Opredelite splošno testiranje enot, ki je osnova testno vodenega razvoja in se posvetite vedenjsko vodenemu razvoju. V nadaljevanju predstavite obstoječe rešitve in orodja za testiranje JavaScript. Predstavite tudi zgradbo in delovanje orodja AllGreen, namenjenega testiranju kode JavaScript.

IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Spodaj podpisani Gregor Stamać, z vpisno številko 24950460, sem avtor diplomskega dela z naslovom:

Orodje za testno voden razvoj JavaScript aplikacij

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom prof. dr. Saše Divjaka,
- 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 15. marca 2015 Podpis avtorja:

Posebna zahvala gre mojima staršema Ljubi in Rafaelu, ki sta mi kupila prvi ZX Spectrum, s katerim se je vse začelo, me podpirala in me vsake toliko opomnila, da je pametno stvari dokončati. Brez njiju ne bi bil to kar sem.

Iskrena hvala moji ženi Vanji za spodbudo in podporo, ki mi jo daje in mi dovoli, da se včasih zgubim med ničlami in enicami. Še posebna hvala mojim otrokom Lari, Emi in Eriku, ki mi več energije dajo kot vzamejo in mi vsak dan pokažejo, da sem lahko ponosen na njih.

Hvala tudi osnovnošolski učiteljici Maji Manohin, ki me je usmerjala pri raziskovanju Basica in mi pomagala pri prvih korakih v svetu programiranja.

Zahvaljujem se mentorju prof. dr. Saši Divjaku, da je kljub upokojitvi prevzel mentorstvo pri izdelavi diplomske naloge.

Hvala Andreji Kobal, ki mi je nalogo več kot odlično zlektorirala.

KAZALO

SEZNAM SLIK

SEZNAM UPORABLJENIH KRATIC

Kratica	
API	Programski vmesnik *(angl. application programming interface)* - vmesnik, ki zagotavlja, da ima računalniški program na razpolago funkcije operacijskega sistema ali drugega računalniškega programa.
BDD	Vedenjsko voden razvoj (*angl. behavior driven development*)
CI	Zvezna integracija (*angl. continuous integration*) - samodejno prevajanje izvorne kode z namenom zgodnjega ugotavljanja napak.
CSS	Prekrivni slogi (*angl. cascading style sheet*) - stilna predloga na spletni strani, v kateri je zapisana oblika spletne strani.
DOM	Objektni model dokumenta (*angl. document object model*) – konvencija za predstavitev in interakcijo z objekti v HTML dokumentih.
DRY	»Ne ponavljaj se« (*angl. »don't repeat yourself«*) – princip razvoja programske opreme, kjer stremimo k temu, da se koda ne ponavlja.
ECMA	European Computer Manufacturers Association – Mednarodna organizacija za standarde informacijskih in komunikacijskih sistemov.
HTML	Standardni označevalni jezik za izdelavo spletnih strani (*angl. hypertext markup language*).
IoC	Inverzija kontrole (*angl. inversion of control*)
JSON	Odprto-standardni format, ki omogoča prenos objektov v človeško berljivi obliki (*angl. JavaScript object notation*).
MVVM	Arhitekturni vzorec, ki loči stanje od vedenja programa (*angl. model view viewmodel*).
OWIN	Standard za vmesnik med .NET spletnimi aplikacijami in spletnimi strežniki (*angl. open web interface for .NET*).
TDD	Testno voden razvoj (*angl. test driven development*)
W3C	World Wide Web Consortium – Glavna mednarodna organizacija za standarde spletnih tehnologij.

WPF	Grafični podsistem za izrisovanje uporabniških vmesnikov v .NET aplikacijah za Windows operacijski sistem (*angl. Windows presentation foundation*).
YAGNI	»Ne boš ga potreboval« (*angl. »You aren't gonna need it«*) – princip razvoja programske opreme, ki pravi, da programer ne sme dodati funkcionalnosti, dokler je ne potrebuje.

POVZETEK

Diplomsko delo opisuje implementacijo orodja za testiranje JavaScript kode. Orodje je zasnovano tako, da nam pomaga pri testno vodenem razvoju JavaScript aplikacij. Zato je pomembno, da nam kar najhitreje sporoča rezultate testov. Naloga je razdeljena na štiri dele. V prvem delu naloge je opisano samo JavaScript okolje. Opisana je kratka zgodovina JavaScript jezika, razširjenost in slabosti. V tem delu je opisan tudi TypeScript jezik, ki je nadgradnja JavaScript jezika. V drugem delu naloge je opisan testno voden razvoj. Opisana so pravila, prednosti in slabosti testno vodenega razvoja. V tem delu je opisano splošno testiranje enot, ki je osnova testno vodenega razvoja. Kot nadgradnja pa je opisan tudi vedenjsko voden razvoj. V tretjem delu je opisano testiranje v JavaScript okolju. Predstavljene so obstoječe rešitve in orodja za JavaScript testiranje. V zadnjem delu pa je opisano orodje AllGreen, namenjeno za testiranje JavaScript kode. Predstavljeni so arhitektura, zgradba in delovanje orodja. Za konec pa je opisano še testiranje samega orodja AllGreen.

Ključne besede: testno voden razvoj, TDD, vedenjsko voden razvoj, testiranje enot, JavaScript, TypeScript, zvezna integracija, razvoj programske opreme, .NET, čebulna arhitektura, agilni razvoj programske opreme

ABSTRACT

Thesis describes the implementation of a tool for testing JavaScript code. The tool is designed to help us in test-driven development of JavaScript-based applications. Therefore, it is important to display test results as quickly as possible. The thesis is divided into four parts. First part describes JavaScript environment. It contains a brief history of the JavaScript language, prevalence, strengths and weaknesses. This section also describes TypeScript programming language that is a superset of JavaScript language. Second part of the thesis describes test-driven development. It contains the rules, advantages and disadvantages of test-driven development. It also describes unit testing, which is the basis of test-driven development. It also describes behavior-driven development, as an upgrade of test-driven development. Third part describes the testing in JavaScript environment. The existing solutions and tools for JavaScript testing are introduced. The last part describes AllGreen, a tool for testing JavaScript code. The architecture and inner workings of the tool are presented. For the end testing the AllGreen tool is described.

Keywords: test-driven development, TDD, behavior-driven development, unit testing, JavaScript, TypeScript, continuous integration, software development, .NET, onion architecture, agile software development

1 Uvod

Test-Driven Development (TDD) ali testno voden razvoj je postal ena izmed osnovnih praks modernega agilnega razvoja. Pri tej praksi programerji pišejo teste pred produkcijsko kodo in s tem povečajo kvaliteto programiranja in se osredotočijo na rešitev, ki jo poskušajo realizirati s programom. Kot stranski produkt te prakse pa pridobijo nabor testov, ki omogoča varno preurejanje kode (*angl. refactoring*) in na ta način spremenijo kodo tako, da je bolj berljiva in se jo lažje vzdržuje.

Testno voden razvoj je, kot je sam povedal, »ponovno iznašel« Kent Beck leta 2003 v knjigi Test-Driven Development by Example [1]. Testno voden razvoj je povezan s konceptom Test-First, kot del Extreme Programming metodologije [2], ki se je začela razvijati leta 1999. Vendar pa je sam koncept pisanja testov pred pisanjem produkcijske kode veliko starejši. Že leta 1957 je bil podoben koncept opisan v knjigi Digital Computer Programming [3], leta 1972 pa ga je v knjigi The Humble Programmer [4] opisal celo Edsger W. Dijkstra.

Avtomatsko testiranje kode in celo testno voden razvoj je torej star že več kot 50 let, pa vendar je še danes zelo pomemben. Verjetno še bolj, saj so programi obsežnejši. Testiranje pa je zelo pomembno, če ne celo nujno potrebno, ko uporabljamo dinamične programske jezike in skriptne programske jezike. Dinamični jeziki so še posebno dovzetni za napake programerjev, saj dopuščajo ogromno svobode. Skriptni jeziki pa, ker nimajo prevajalnikov,

zahtevajo od programerjev, da sami odkrivajo in preprečujejo napake, ki bi jih sicer odkrili prevajalniki.

JavaScript je programski jezik, ki spada v obe zgoraj omenjeni skupini programskih jezikov. Poleg tega se skripte, ki so napisane v JavaScript programskem jeziku, izvajajo v različnih okoljih različnih proizvajalcev. Zaradi tega se lahko zgodi, da se ista skripta obnaša drugače v različnih okoljih. Zato je še posebno pomembno, da se JavaScript kodo natančno testira v vseh okoljih, za katera je namenjena.

Obstaja torej potreba po orodju, ki bi omogočalo enostavno in hitro testiranje JavaScript kode v čim večjem obsegu. Zato sem ustvaril AllGreen zaganjalnik JavaScript testov. AllGreen omogoča avtomatsko izvajanje testov na velikem številu brskalnikov. Deluje kot strežnik, na katerega se povežejo brskalniki, ki izvajajo teste, ki jim jih strežnik dostavi.

2 JAVASCRIPT

JavaScript je dinamičen programski jezik, ki ga največkrat uporabljajo v spletnih brskalnikih. Leta 1995 so ga razvili v podjetju Netscape za njihov Netscape Navigator brskalnik. Leta 1996 so ga v brskalnik Internet Explorer dodali tudi v podjetju Microsoft, vendar so uporabili ime JScript zaradi zaščite imena JavaScript. Leta 1996 je podjetje Netscape objavilo, da so oddali JavaScript kot predlog za standardizacijo organizaciji Ecma International. Tako je JavaScript leta 1997 dobil standardizirano verzijo pod imenom ECMAScript. JavaScript ima lastnosti, zaradi katerih velja za jezik več programskih paradigem. Podpira objektno orientirano, imperativno in funkcijsko programiranje. [5]

2.1 ZGODOVINA JAVASCRIPTA

JavaScript je leta 1995 razvil Brendan Eich v podjetju Netscape. Takrat se je ta jezik imenoval LiveScript. Ker pa so v oddelku za marketing hoteli izkoristiti popularnost programskega jezika Java, so se odločili preimenovati LiveScript v JavaScript. Na žalost so začeli JavaScript povezovati z jezikom Java, čeprav nista imela veliko skupnega. Java je bila takrat namenjena tudi za razširitev brskalnikov, in je s tem upočasnila izvajanje brskalnika. Tako se je Jave prijel slab glas, da jo razvijalci uporabljajo za dodajanje neuporabnih dodatkov, ki samo upočasnijo izvajanje brskalnikov. Ker pa so JavaScript povezovali z Javo, se je tudi JavaScripta prijel slab glas. [6]

Leta 1996 je tudi Microsoft dodal podporo za JavaScript v svoj brskalnik Internet Explorer 3.0. V tej verziji brskalnika je Microsoft dodal podporo za dva skriptna jezika, JScript, ki je bil osnovan na JavaScriptu, in VBScript, ki je bil osnovan na Visual Basicu. VBScript je kmalu utonil v pozabo. Microsoft je hotel prekositi Netscape zato je v svojo verzijo JavaScripta dodal veliko novih funkcij, vključno z dostopom do njihove ActiveX tehnologije.

Čeprav je Microsoftov JScript baziral na Netscapovem JavaScriptu, so vseeno obstajale razlike. Razvijalci so to reševali tako, da so v aplikacijah preverjali, na katerem brskalniku se skripta izvaja, in glede na to izvedli primerno skripto. Ko je Microsoft s svojim brskalnikom v tržnem deležu ujel Netscape in se je uporaba Netscapovega JavaScripta in Microsoftovega JScripta izenačila, je bilo potrebno nekaj narediti glede standardizacije JavaScripta.

Tako je podjetje Microsoft leta 1996 predalo JavaScript v standardizacijo organizaciji European Computer Manufacturers Association (ECMA). ECMA je JavaScript formaliziralo v standard ECMAScript. Ob istem času pa je tudi organizacija World Wide Web Consortium (W3C) končala standard Document Object Model (DOM), ki bi ga JavaScript in drugi skriptni jeziki uporabljali za dostop do vsebine spletne strani. Do takrat je imel JavaScript le omejen dostop do te vsebine. Še pred izdajo tega standarda pa sta tako Microsoft kot Netscape že izdala novi verziji svojih brskalnikov, ki sta že imela vsak svoj objektni model. Z novejšimi verzijami sta ta modela opustila in danes se uporablja izključno standardni DOM.

Prva verzija ECMAScripta je bila izdana junija leta 1997. Leto kasneje je bila izdana druga verzija, ki je vsebovala uredniške spremembe, da je ustrezala mednarodnemu standardu ISO/IEC 16262. Leta 1999 je bila izdana verzija 3, kateri je bila dodana podpora za regularne izraze, boljše ravnanje z nizi, nove kontrolne stavke, try/catch ravnanje z napakami, formatiranje numeričnih izhodov in druge izboljšave. Četrta verzija specifikacije je bila zavržena zaradi nestrinjanja glede kompleksnosti jezika in kompatibilnosti s prejšnjimi verzijami. Nekaj funkcij je bilo prenesenih v novejše verzije, nekaj pa je bilo opuščenih. V verziji 5, ki je bila objavljena leta 2009, je bila dodana opcija »strict mode«, ki je omogočala omejen nabor funkcionalnosti, vendar bolj obsežno preverjanje

napak. Poleg tega je bila dodana podpora JSON, »getter« in »setter« konstrukta ter omogočen obsežnejši dostop do strukture objektov (*angl. reflection*). Zadnja objavljena verzija ECMAScript 5.1 iz leta 2011 je popolnoma usklajena z mednarodnim standardom ISO/IEC 16262:2011. [7]

V pripravi sta dve novi verziji ECMAScript standarda. Verzija 6 (imenovana tudi ECMAScript Harmony) naj bi vsebovala veliko sintaktičnih sprememb, ki bi omogočale lažjo izdelavo kompleksnih aplikacij. Objava naj bi bila načrtovana za junij leta 2015. Verzija 7 naj bi nadaljevala reforme, načrtovane za verzijo 6, in izboljšala izolacijo kode, podporo za funkcijsko programiranje, numerično matematiko, alternativo statičnim tipom, prepisovanje operatorjev, nove strukture in drugo.

Večina današnjih brskalnikov podpira ECMAScript verzija 5. Nekateri brskalniki že podpirajo nekatere neuradne funkcije verzije 6. Obstaja pa še odprtokodni programski jezik, imenovan TypeScript, ki je nadgradnja jezika JavaScript. TypeScript doda nekatere funkcije ECMAScripta 6, vendar ga prevajalnik pretvori v ECMAScript 5 ali celo ECMAScript 3.

2.2 RAZŠIRITVE RAZLIČNIH PROIZVAJALCEV

JavaScript uradno upravlja Mozilla Foundation, naslednik podjetja Netscape. Nove funkcije jeziku dodajajo dokaj redno, vendar pa jih ne podpirajo vsi JavaScript pogoni. Nekatere funkcije, ki jih ne podpirajo vsi proizvajalci, so:

- »getter« in »setter« funkcije za dostop do lastnosti objektov
- Pogojni procesorji napak (*angl. catch*)
- Iterator protokol (povzeto iz Pythona)
- Plitki generatorji – korutine (povzeto iz Pythona)
- Izrazi za generiranje seznamov iz drugih seznamov (povzeto iz Pythona)
- Pravilen obseg spremenljivk z uporabo »let« izraza
- Destrukturiranje seznamov (omejena oblika ujemanja vzorcev)
- Zgoščena oblika deklaracije funkcij

2.3 RAZŠIRJENOST

V osnovi je bil jezik JavaScript razvit za uporabo v dinamičnih spletnih straneh in še danes ga največkrat povežemo z spletnimi brskalniki, vendar pa njegova uporaba ni omejena le na to. Nekateri brskalniki ga že uporabljajo za programiranje dodatkov. V zadnjem času je najbolj znana njegova uporaba za programiranje strežniških aplikacij z ogrodjem Node.js. Po nekaterih statistikah naj bi bil JavaScript najbolj razširjen programski jezik na svetu. [8] [9]

Poleg JavaScripta v dinamičnih spletnih straneh in Node.js pa obstaja še mnogo bolj ali manj znanih implementacij njegove uporabe [5].

2.3.1 VDELANI SKRIPTNI JEZIK

- Google's Chrome dodatki, Opera's dodatki, Apple's Safari 5 dodatki, Apple's Dashboard Widgets, Microsoft's Gadgets, Yahoo! Widgets, Google Desktop Gadgets in Serence Klipfolio.
- MongoDB podatkovna baza sprejema povpraševalne izraze, napisane v JavaScriptu. MongoDB in Node.js sta osnovni komponenti MEAN, ki je namenjen za pisanje spletnih aplikacij z uporabo le JavaScripta.
- Adobe Acrobat in Adobe Reader podpirata JavaScript v PDF datotekah.
- Orodja v Adobe Creative Suite (Photoshop, Illustrator, Dreamweaver in InDesign) omogočajo pisanje skript z JavaScriptom.
- OpenOffice.org omogoča uporabo JavaScripta kot skriptnega jezika.
- Interaktivno programje za procesiranje glasbenih signalov Max/MSP omogoča razvijalcem uporabo JavaScript modela. Omogoča bolj natančno kontrolo, kot pa model na osnovi grafičnega vmesnika.
- Apple Logic Pro X omogoča uporabo JavaScripta za ustvarjanje MIDI učinkov.
- ECMAScript je bil del VRML97 standarda za pisanje skriptnih vozlišč VRML datotek.
- Odprtokodno ogrodje Re-Animator omogoča razvoj 2D iger z uporabo JavaScripta in XML formata.

- Unity pogon za igre podpira modificirano verzijo JavaScripta za pisanje skript preko Mono okolja.
- DX Studio (3D pogon) uporablja SpiderMonkey implementacijo JavaScripta za igre in simulacijsko logiko.
- Maxwell Render uporablja pogon za procesiranje skript na podlagi ECMA standarda.
- Google Apps Script v Google Spreadsheetsu in Google Sitesu omogoča pisanje formul po meri, avtomatizacija nalog in interakcijo z ostalimi Google produkti (npr. Gmail).
- Veliko IRC klientov, kot ChatZilla ali XChat, uporablja JavaScript za procesiranje skript.
- SpinetiX produkti uporabljajo SpiderMonkey JavaScript pogon za procesiranje skript v SVG datotekah.

2.3.2 SKRIPTNI POGON

- Microsoft Active Scripting tehnologija podpira JScript kot skriptni jezik.
- Programski jezik Java je vključil paket javax.script v verziji 6. Ta vključuje JavaScript implementacijo na osnovi Mozilla Rhino. Tako lahko Java aplikacije gostijo skripte ki dostopajo do objektov aplikacije, tako kot spletni brskalniki gostijo skripte, ki dostopajo do modela DOM.
- Qt C++ orodje vsebuje QtScript modul, ki interpretira JavaScript, tako kot javax.script paket v jeziku Java.
- JSDB (JavaScript for Databases) je odprtokodna JavaScript lupina za Windows, Mac OS X, Linux in Unix. JSDB nadgrajuje Mozilla JavaScript pogon z objekti za datoteke, podatkovne baze, spletne pošte in omrežje.
- jslibs je odprtokodna JavaScript lupina za Windows in Linux, ki nadgrajuje Mozilla JavaScript pogon. Ima podporo za klicanje funkcij v popularnih knjižnicah (npr. NSPR, SQLite, libTomCrypt, OpenGL, OpenAL in librsvg).
- JavaScript OSA (ali JavaScript for OSA ali JSOSA) proizvajalca Late Night Software je zastonjski paket, alternativa AppleScriptu za Mac OS X. Zasnovan je na Mozilla 1.5 JavaScript implementaciji, z dodatkom objekta MacOS za interakcijo z operacijskim sistemom in eksternimi aplikacijami.

2.3.3 APLIKACIJSKA PLATFORMA

- ActionScript, programski jezik, uporabljen v Adobe Flash, je še ena implementacija ECMAScript standarda.
- Adobe Integrated Runtime je JavaScript izvajalno okolje, ki omogoča izgradnjo namiznih aplikacij.
- CA, Inc.'s AutoShell cross-application okolje za procesiranje skript je zgrajeno na SpiderMonkey JavaScript pogonu.
- GNOME Shell, lupina za GNOME 3 namizno okolje, je privzel JavaScript kot primarni programski jezik.
- Mozilla platforma, ki je osnova za Firefox, Thunderbird in še nekatere spletne brskalnike ter klienta za spletno pošto, uporablja JavaScript za implementacijo grafičnega uporabniškega vmesnika.
- myNFC je ogrodje na osnovi JavaScripta, ki omogoča razvijalcem izgradnjo aplikacij za pametne telefone.
- Qt Quick uporablja JavaScript za označevalni jezik od verzije 4.7 dalje.
- TypeScript je programski jezik, ki temelji na JavaScriptu.
- Ubuntu Touch ponuja JavaScript API za enoten uporabniški vmesnik.
- webOS v SDK uporablja WebKit implementacijo JavaScripta in s tem omogoča razvijalcem razvoj samostojnih aplikacij izključno v JavaScriptu.
- WinJS ponuja posebno funkcionalnost Windows Library for JavaScript v Windows 8, ki omogoča razvoj aplikacij v HTML5 in JavaScriptu.

2.3.4 MIKROKRMILNIKI

Število specifikacij mikrokrmilnikov se je v zadnjem času povečalo, zato je možna uporaba JavaScripta tudi za nadzor nad strojno opremo in vgrajenimi napravami. Trenutno obstajata dve glavni implementaciji:

- Espruino je JavaScript interpreter za mikrokrmilnike z nizko porabo energije.
- Tessel je plošča z mikrokrmilnikom, ki ima vgrajeno brezžično omrežje WiFi.

2.4 JavaScript je dinamičen jezik

Dinamični programski jeziki opravljajo v času izvajanja nekatere funkcije, ki jih statični jeziki v času prevajanja. JavaScript je izredno dinamičen, saj v času izvajanja spreminja tipe, strukturo objektov in celo izvaja izraze v obliki nizov znakov.

V JavaScriptu so tipi povezani z vrednostmi, in ne s spremenljivkami. Tako ima lahko neka spremenljivka najprej numerično vrednost in kasneje vrednost v obliki niza znakov.

Objekti v JavaScriptu so asociativni nizi, tako da so imena lastnosti in metode objektov predstavljeni z nizi znakov. JavaScript podpira dve notaciji za dostop do lastnosti in metod, obj.x ali obj['x']. Lastnosti in metode lahko objektom dodajamo, spreminjamo in odstranjujemo v času izvajanja. Dedovanje je realizirano s prototipi. Nove instance objektov kreiramo tako, da obstoječi objekt kloniramo. Temu objektu lahko dodajamo nove metode in lastnosti.

JavaScript vsebuje tudi funkcijo »eval«, ki omogoča izvajanje izrazov, ki jih v času izvajanja predstavimo v obliki nizov znakov. Tako lahko izvedemo izraze, ki pred izvajanjem še niso bili napisani, in jih dinamično oblikujemo v času izvajanja.

2.5 Slabosti JavaScripta

Ker je JavaScript dinamičen jezik, nam omogoča veliko svobode pri programiranju. Vendar pa nam ta svoboda omogoča tudi veliko napak, ki bi jih v statičnem jeziku odkril prevajalnik. Tako moramo, če želimo narediti kvalitetne programe, sami narediti veliko kontrol, ki bi jih sicer izvedel prevajalnik. Naloge ki bi jih izvedel prevajalnik, so tako odvisne od discipline programerjev. Še posebno je to pomembno, če programiramo ogrodja ali knjižnice, ki jih bodo uporabljali drugi. Imamo sicer še drugo možnost in pustimo uporabnikom naših knjižnic, da sami poskrbijo za te kontrole.

Rezultat je razvojno okolje, kjer se lahko v kodi hitro pojavljajo hrošči zaradi človeške zmotljivosti. Hrošče je težko odkriti in jih odkrijemo šele v času uporabe programa. Zaradi tega morajo biti programerji veliko bolj disciplinirani. Ta pomanjkljivost dinamičnih jezikov vpliva tudi na urnik projektov, saj je potrebno računati na dodatne zamude zaradi neodkritih hroščev.

Zagovorniki dinamičnih jezikov pravijo, da statičnih tipov ne potrebujemo, če imamo teste enot. Poleg tega povedo še, da nekatere rešitve niso možne s statičnimi tipi. [10] Vendar pa je Evan R. Farrer v svoji disertaciji pokazal, da testi enot ne morejo popolnoma nadomestiti statičnih tipov. Primerjal je štiri odprtokodne projekte, napisane v dinamičnem jeziku Python. Vse štiri je pretvoril v statični jezik Haskell in ugotovil, da statični tipi niso bili omejitev. Le pri dveh je bila potrebna manjša sprememba arhitekture, da je dinamične tipe spremenil v statične. Ko je projekte pretvoril, je ugotovil, da je uporaba statičnih tipov odkrila napake, ki jih tudi testi enot niso odkrili. [11]

Nekateri pravijo, da JavaScript ni »pravi« razvojni jezik [12], ker:

- se ne prevede v strojno kodo. Zato lahko vsak, ki dostopa do spletne strani, kjer se neka JavaScript skripta uporablja, to skripto prebere in jo »ukrade«. Še hujši problem pa je, da lahko to kodo spreminja;
- ni pravi objektno orientirani jezik. Zagovorniki sicer pravijo, da lahko s prototipno orientiranim jezikom naredimo vse, kar lahko z jezikom, ki vsebuje razrede in vmesnike. Vendar pa JavaScriptu manjkajo nekatere funkcije objektno orientiranih jezikov: abstrakcija, enkapsulacija, polimorfizem in dedovanje;
- praktično nima tipske varnosti. Zagovorniki sicer pravijo, da je JavaScript funkcijski jezik in je zato dinamičen, vendar to nikakor ni pogoj za funkcijski jezik. »Pravi« funkcijski jeziki, kot sta Haskell in OCaml, so zelo močno tipizirani in ne dovolijo nikakršnega implicitnega pretvarjanja tipov;
- spodbuja abstraktne nivoje. Razlog za toliko abstrakcij, kot jih ima JavaScript (npr. jQuery, Objective-J in GWT), pa so verjetno ravno pomanjkljivosti JavaScripta.

Da je JavaScript pomanjkljiv jezik, lahko zaključimo tudi zato, ker obstaja ogromno pravil, ki nam povejo, kako pravilno uporabljamo JavaScript, da si ne nakopljemo težav. O tem na primer govori velik strokovnjak za JavaScript Dmitry Baranovskiy, ki je odkrival napake v znani JavaScript knjižnici Closure, ki so jo naredili v podjetju Google. [13]

Zaradi vseh teh pomanjkljivosti je veliko proizvajalcev poskušalo nadomestiti JavaScript z nečim boljšim.

Google poskuša JavaScript nadomestiti s svojim programskim jezikom Dart. Google izdaja verzijo svojega brskalnika, v katerem lahko zaganjamo Dart skripte. Poleg tega so naredili tudi prevajalnik, ki Dart skripte pretvori v JavaScript skripte.

CoffeeScript je odprtokodni programski jezik, ki temelji na jezikih Ruby, Python in Haskell. Tudi CoffeeScript je lahko prevesti v JavaScript.

Microsoft pa pomanjkljivosti JavaScripta poskuša rešiti s svojim odprtokodnim jezikom TypeScript. Tudi TypeScript je možno prevesti v JavaScript. Zanimivo je to, da je TypeScript stroga supermnožica JavaScripta, kar pomeni, da je vsaka JavaScript skripta tudi TypeScript skripta.

3 TypeScript

TypeScript [14] so naredili, da bi zadostili potrebam ekip, ki v JavaScriptu gradijo in vzdržujejo velike JavaScript programe. TypeScript pomaga programerjem definirati vmesnike med programskimi komponentami in pridobiti vpogled v obstoječe JavaScript knjižnice. TypeScript tudi pomaga programerjem zmanjšati konflikte pri imenovanju elementov programa, s tem da omogoča organiziranje kode v dinamične module. TypeScriptov opcijski tipni sistem pa omogoča JavaScript programerjem uporabo visokoproduktivnih orodij in praks: statično preverjanje, navigacijo na osnovi simbolov, samodejno dovršitev stavkov in preoblikovanje kode (*angl. refactoring*).

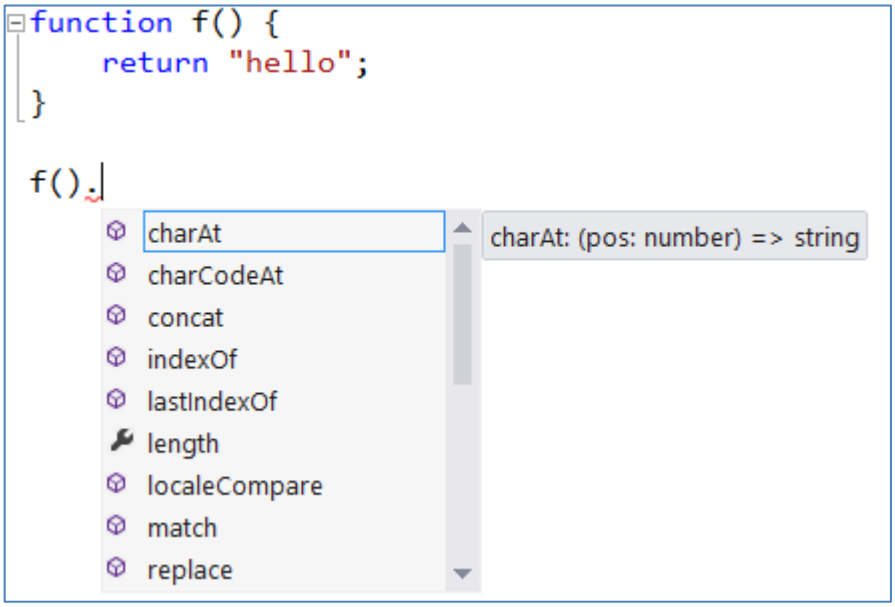
3.1 Sintaksa TypeScripta

TypeScript je sintaktična nadgradnja JavaScripta. TypeScriptova sintaksa je supermnožica sintakse ECMAScripta 5 (ES5). Vsak JavaScriptov program je tudi TypeScriptov program. TypeScriptov prevajalnik izvede le lokalne transformacije na TypeScriptovih programih in ne spremeni vrstnega reda spremenljivk definiranih v TypeScriptu. Rezultat tega je JavaScript izhodna skripta, ki se zelo ujema s TypeScriptovo vhodno skripto. TypeScript ne spreminja imena spremenljivk in s tem omogoča direktno razhroščevanje izhodne JavaScript kode. TypeScript lahko izvozi tudi povezovalne datoteke in s tem omogoča razhroščevanje na nivoju izvorne kode. TypeScript orodja običajno izvozijo JavaScript kodo pri shranjevanju vhodnih datotek.

TypeScript sintaksa vključuje več predlaganih lastnosti ECMAScripta 6 (ES6), vključno z razredi in moduli. Razredi omogočajo programerjem izražanje standardnih objektno orientiranih vzorcev na standardni način. Tako so funkcije kot dedovanje bolj berljive in jih je lažje vzdrževati. Moduli omogočajo programerjem organiziranje njihove kode v komponente in izogibanje konfliktom v poimenovanju. TypeScriptov prevajalnik omogoča generiranje kode modulov tako, da je možno statično ali dinamično nalaganje vsebine modulov.

TypeScript omogoča še JavaScript programerjem sistem za opcijsko označevanje s tipi. To označevanje s tipi je podobno JSDoc komentarjem, ki jih najdemo v sistemu Closure, vendar pa so v TypeScriptu integrirani v sintaksi jezika. Ta integracija naredi kodo bolj berljivo in zmanjša stroške vzdrževanja.

TypeScriptov tipni sistem omogoča programerjem izražanje omejitev JavaScriptovih objektov in uporabo orodij, ki te omejitve uveljavljajo. Da bi zmanjšali potrebo po označevanju tipov, zato da bi bila orodja uporabna, pa TypeScriptov tipni sistem obsežno uporablja tipno sklepanje. Tako lahko recimo TypeScript iz vrednosti, ki jo vrne neka funkcija, ugotovi, kakšnega tipa naj bi bil rezultat te funkcije. Da bi programer izkoristil to tipno sklepanje, lahko uporabi servis TypeScript jezika. Tako lahko na primer urejevalnik za kodiranje uporablja servis jezika TypeScript, da poišče elemente objekta in jih prikaže (slika 1). In to naredi, ne da bi uporabil označevanje s tipi.



Slika 1: Dodatek za urejanje TypeScript kode

Nekaterih tipov pa TypeScript ne more prepoznati, zato je potrebno spremenljivke, parametre in rezultate funkcij označiti s tipi. V TypeScriptu tip parametra izrazimo tako, kot je prikazano na spodnjem primeru:

```
function f(s: string) {
    return s;
}

f({});       // Napaka
f("hello");  // V redu
```

Ta opcijski tip parametra bo TypeScriptu dal vedeti, da mora biti parameter »s« tipa »string«. V notranjosti funkcije »f« lahko orodja prevzamejo, da je »s« tipa »string« in lahko pomagajo programerjem, da jim predlaga funkcije in lastnosti za objekt tipa »string«. TypeScriptov prevajalnik pa bo tudi izpisal napako, če bo programer hotel uporabiti to funkcijo s parametrom, ki ni tipa »string«, kot lahko vidimo v zgornjem primeru. Funkcijo »f« bo TypeScriptov prevajalnik predelal v spodnjo JavaScript kodo:

```
function f(s) {
    return s;
}
```

V izhodni JavaScript kodi so bile vse označbe tipov izbrisane. Na splošno TypeScriptov prevajalnik izbriše celotno informacijo o tipih predno izvozi JavaScript kodo.

3.2 AMBIENTNE DEKLARACIJE

Ambientna deklaracija vpelje spremenljivke v obseg TypeScripta, vendar nič ne vpliva na izhodni JavaScript program. Programerji lahko uporabljajo ambientne deklaracije, da povedo TypeScript prevajalniku, da bo neka druga komponenta priskrbela spremenljivko. TypeScript prevajalnik bo privzeto javil napako, če uporabljamo nedefinirano spremenljivko. Da pa lahko uporabljamo nekatere znane spremenljivke, ki jih priskrbijo brskalniki, lahko uporabimo ambientne deklaracije.

V spodnjem primeru deklariramo objekt »document«, ki ga priskrbijo brskalniki. Ker deklaracija ne nastavi tipa, je privzeto, da je tip »any«. Tip »any« pomeni, da ne moremo nič vedeti o obliki in obnašanju objekta »document«.

```
declare var document;
document.title = "Hello";  // V redu, saj je bil document
deklariran
```

4 TESTNO VODEN RAZVOJ

Testno voden razvoj (*angl. test-driven development - TDD*) ni samo pisanje testne kode pred pisanjem produkcijske kode. Testno voden razvoj je drugačen način razmišljanja.

4.1 ZGODOVINA

Ideja pisanja testne kode pred produkcijsko kodo je že zelo stara. Spodaj so nekateri odlomki iz literature, ki tako ali drugače omenjajo avtomatsko testiranje kode. Kot je razvidno, se ne omenja samo pisanje testne kode pred pisanjem produkcijske kode, ampak celo prepletanje teh dveh opravil. To je zelo pomembno, saj je to eno izmed glavnih vodil testno vodenega razvoja.

»Postopek **preverjanja lahko začnemo še pred pričetkom kodiranja**. Da bi popolnoma potrdili pravilnost odgovorov, je nujno imeti ročno izračunane odgovore, s katerimi lahko preverimo odgovore, ki jih bo kasneje izračunal računalnik. To pomeni, da računalnikov s shranjenim programom nikoli ne uporabljamo za probleme z enostavno rešitvijo. Vedno je prisoten element iteracije, da se izplača. Ročne izračune lahko naredimo kadarkoli v času programiranja. Največkrat pa z računalniki upravljajo računalniški izvedenci, ki pripravijo rešitev kot storitev inženirjem ali znanstvenikom. V tem primeru je priporočljivo, da »stranka« pripravi pravilne odgovore, saj lahko na ta način izpostavimo logične napake in nesporazume med »stranko« in programerjem.« To je odlomek iz knjige Digital Computer Programming iz leta 1957 [3] in je

najstarejši znan opis metode pisanja testne kode pred pisanjem produkcijske kode.

»Programski sistem najbolje zasnujemo tako, da je **testiranje prepleteno z načrtovanjem, namesto da ga uporabimo po načrtovanju**. Skozi ponovitve procesa prepletenega testiranja in načrtovanja model končno postane sam programski sistem.« (Poročilo konference The NATO Software Engineering Conference iz leta 1968 [15]).

»Danes je ustaljena praksa, da najprej naredimo program, potem pa ga testiramo. Testiranje programa je zelo učinkovit način, kako prikazati prisotnost napak, vendar pa je zelo neuspešno pri prikazu odsotnosti le-teh. Edini učinkovit način za bistven dvig ravni zaupanja v program je, da poskrbimo za prepričljiv dokaz o pravilnosti programa.

Vendar pa ni priporočljivo najprej narediti program in potem dokazati njegovo pravilnost, saj bi zahteva po zagotavljanju dokaza le povečala breme ubogega programerja. Nasprotno bi moral programer pustiti, da **dokaz in program nastajata z roko v roki**.« (Odlomek iz knjige The Humble Programmer iz leta 1972 [4]).

»**Načrtovalci implementirajo avtomatizirane teste enot vnaprej in jih izvajajo skozi celoten projekt**.« To je ena izmed 12 osnovnih praks metodologije Extreme Programming [2] iz leta 1999.

4.2 PRAVILA TESTNO VODENEGA RAZVOJA

Kent Beck je v knjigi Test Driven Development: By Example [1] leta 2002 »ponovno odkril« testno voden razvoj. V tej knjigi je Kent Beck definiral dve pravili testno vodenega razvoja:

1. Ne napiši vrstice kode, če ne obstaja spodleteli avtomatiziran test.
2. Odstrani dvojnost v kodi.

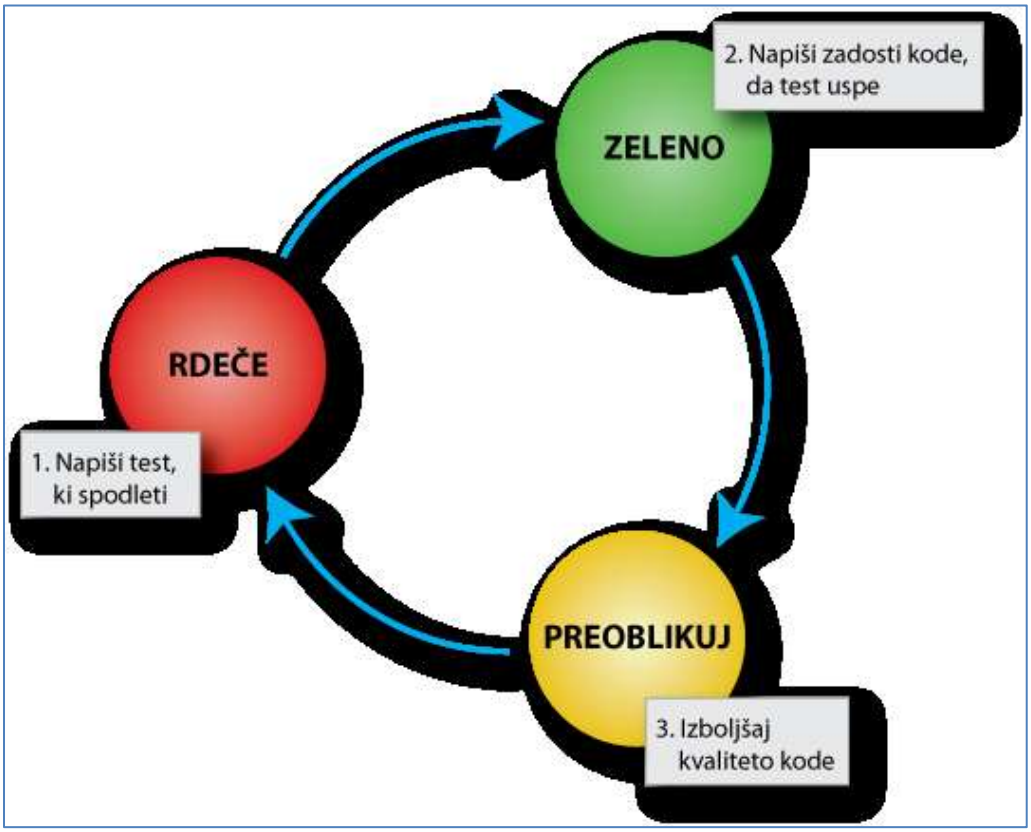
Ti dve preprosti pravili nas prisilita v poseben način razvoja. Nekaj teh tehničnih implikacij je:

1. Sistem načrtujemo organsko, delujoča koda pa nam daje povratno informacijo med odločitvami.
2. Ker je čas med pisanjem testa in pisanjem produkcijske kode kratek, moramo pisati teste sami.
3. Razvojno okolje mora omogočati hiter odziv na majhne spremembe.
4. Sistem mora biti sestavljen iz veliko močno kohezivnih, ohlapnih komponent, da je testiranje lažje.

Pravili testno vodenega razvoja pa določata tudi poseben postopek programiranja. Temu postopku se reče tudi TDD mantra:

1. Rdeče (*angl. red*) – napiši majhen avtomatiziran test, ki spodleti – morda se celo ne prevede.
2. Zeleno (*angl. green*) – napiši zadosti kode (in ne več), da bodo vsi testi uspešni – pri tem ni pomembno, da je koda lepo napisana.
3. Preoblikovanje kode (*angl. refactor*) – izboljšaj kvaliteto kode, odstrani podvojeno kodo in uredi kodo, da jo bo lažje vzdrževati.

TDD mantra je torej »rdeče – zeleno – preoblikuj« (slika 2).



Slika 2: TDD mantra

4.2.1 RDEČE-ZELENO

Testno voden razvoj se začne z avtomatiziranim testom, ki definira željeno spremembo ali novo funkcionalnost. Test mora spodleteti, saj definira funkcionalnost, ki še ni implementirana. Test je pravzaprav avtomatizirana specifikacija kode. Pomembno pravilo je tudi, da mora biti test kar se da majhen, torej naj testira majhen del specifikacije.

Naslednji korak pa je pisanje kode, da bo ta test uspešno izveden. Ko imamo napisanih več testov, je nujno, da vedno uspejo vsi testi, predno nadaljujemo z naslednjim korakom. Tudi tu je pomembno, da napišemo minimalno količino kode, da testi uspejo. Pri tem koraku ne smemo pisati kode, za katero predvidevamo, da jo bomo potrebovali, saj bi s tem napisali kodo, ki ni testirana.

Ker napredujemo z majhnimi koraki, se zavarujemo, da v primeru, ko se s spremembami funkcionalnost programa pokvari, te spremembe niso velike in zato lažje odpravimo napake.

4.2.2 PREOBLIKOVANJE KODE

Rezultat prvih dveh korakov TDD mantre je produkcijska koda, ki je pokrita z avtomatiziranimi testi. Ti testi nam dajo neko varnost, da lahko izvedemo zadnji korak TDD mantre – preoblikovanje kode (*angl. refactoring*). Preoblikovanje kode je majhno iterativno spreminjanje kode, tako da se izboljša interna struktura kode, zunanje obnašanje le-te pa mora ostati enako. Osnovno pravilo preoblikovanja kode je torej, da mora ostati funkcionalnost po preoblikovanju enaka. Ker pa testi testirajo funkcionalnost kode, nam ti testi takoj javijo spremembo funkcionalnosti in posledično tudi napako pri preoblikovanju kode.

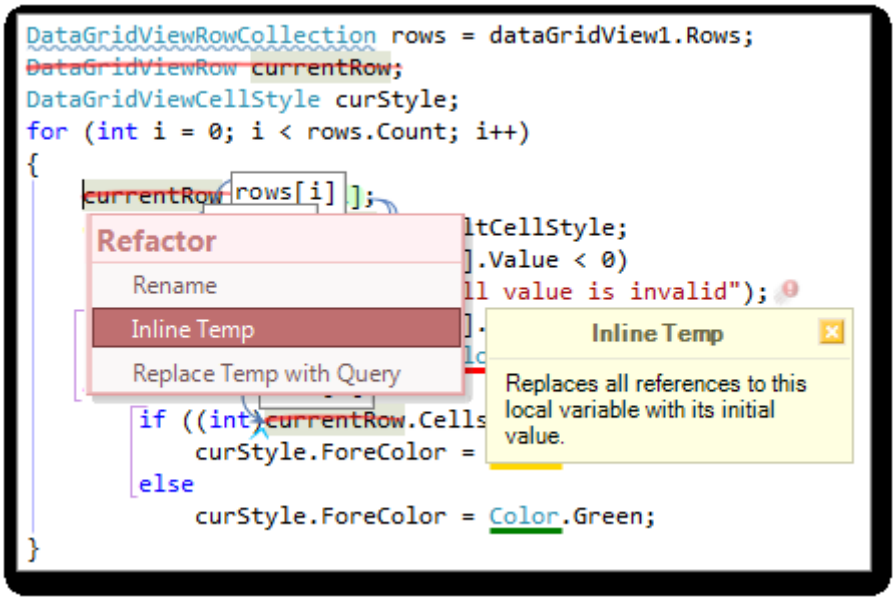
Kot analogijo lahko vzamemo matematični izraz *1*5*4*2*3*. Ta izraz lahko preoblikujemo v *1*2*3*4*5* in nato v *5!*, ne da bi s tem spremenili pomen in rezultat izraza. Vsi trije izrazi pomenijo isto stvar, vendar je vsak naslednji veliko lažje berljiv in razumljiv. Preoblikovanje kode naredi enako s programsko kodo. Postopek preoblikovanja kode nam omogoča ohranjati berljivo kodo in s tem jo tudi lažje vzdržujemo.

Seveda pa je problem, kako zagotoviti, da se pri preoblikovanju kode ne bo spremenila funkcionalnost. To zagotovilo nam dajo testi. Če teh testov ne bi imeli, verjetno ne bi imeli poguma spreminjati kodo – vsaj ne v veliki meri. Kolikokrat se nam je že zgodilo, da smo kakšno staro kodo raje pustili, tako kot je, kot pa da bi jo preuredili.

Moderna orodja za razvoj že vsebujejo nekatere avtomatizirane metode preoblikovanja kode. Eno izmed takih orodij je tudi dodatek za Visual Studio CodeRush proizvajalca DevExpress [16]. V času pisanja te diplome je CodeRush vseboval več kot 200 avtomatiziranih operacij preoblikovanja kode. V nadaljevanju je omenjenih le nekaj.

Vstavi začasno spremenljivko (angl. inline temp)

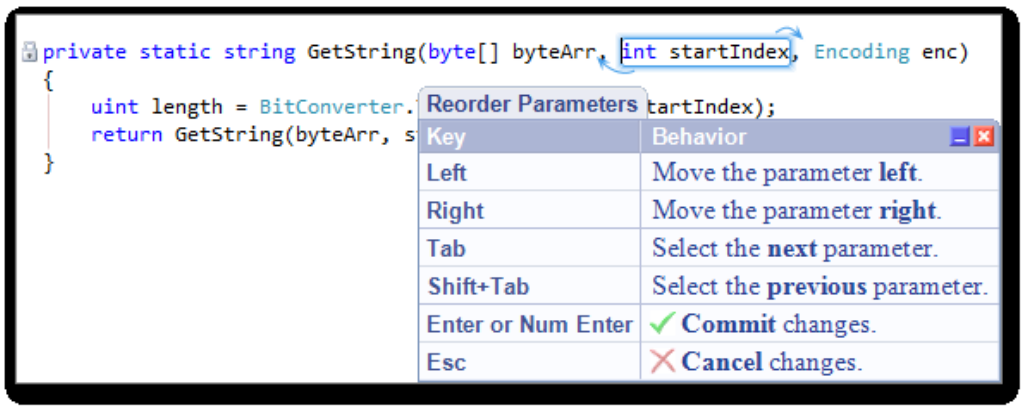
Operacija zamenja reference na neko spremenljivko z vrednostjo te spremenljivke (slika 3).



Slika 3: Preoblikovanje kode - vstavi začasno spremenljivko

Spremeni vrstni red parametrov (angl. reorder parameters)

Operacija spremeni vrstni red parametrov metode in posodobi vse klice te metode (slika 4).



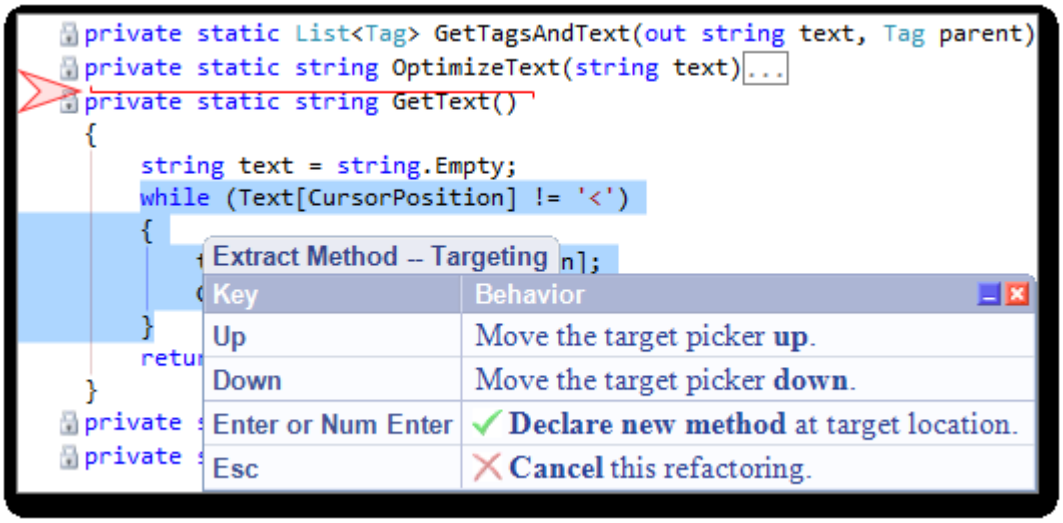
Slika 4: Preoblikovanje kode - spremeni vrstni red parametrov

Preimenuj (angl. rename)

Operacija preimenuje označeno spremenljivko, metodo, funkcijo in posodobi vse reference na ta element.

Izvleci metodo (angl. extract method)

Operacija označeno kodo prestavi v novo metodo in spremeni v klic te metode. Del te operacije je tudi imenovanje nove metode (slika 5).



Slika 5: Preoblikovanje kode - izvleci metodo

4.3 TESTIRANJE ENOT

Pri testno vodenem razvoju uporabljamo metodo testiranja, imenovano testiranje enot (*angl. unit testing*). Enota v tem pogledu je najmanjši del kode, ki se jo da testirati. Pri objektno orientiranem programiranju je ta enota ponavadi razred.

Praviloma mora biti test popolnoma izoliran. To dosežemo tako, da je test neodvisen od drugih testov. Nikakor ni priporočljivo izvajati zaporedja testov. Pogoj za izoliranost je tudi to, da je rezultat testa odvisen le od pravilnosti enote, ki jo testiramo. Ta zadnji pogoj je na prvi pogled težko doseči, saj je vsak razred ponavadi odvisen od drugega razreda. Vseeno pa lahko izoliranost enote dosežemo tako, da dele, od katerih je ta enota odvisna, kontroliramo.

4.3.1 TESTNI DVOJNIKI

Pri testno vodenem razvoju testiramo posamezne enote kode. Pravilo testiranja enot je, da mora biti uspeh testa odvisen le od enote, ki jo testiramo. Zato moramo enoto, ki jo testiramo, izolirati. To pa lahko dosežemo tudi tako, da druge dele, od katerih je enota odvisna, kontroliramo. Ker je enota pri objektno orientiranem programiranju objekt, moramo torej kontrolirati druge objekte. Najlažje to naredimo tako, da te objekte v testu zamenjamo s testnimi dvojniki (*angl. test doubles*). Testni dvojniki simulirajo delovanje resničnih objektov. Testni dvojnik ima enak vmesnik (*angl. interface*) kot resnični objekt, zato ga lahko nadomesti. Predvsem jih uporabljamo takrat, kadar imajo resnični objekti naslednje lastnosti:

- Objekt je preveč kompleksen.
- Objekt ima nedeterministične lastnosti (npr. datum).
- Objekt ima stanja, ki jih je težko simulirati (npr. napaka omrežja).
- Objekt je odvisen od infrastrukture (npr. internetna povezava).
- Objekt dostopa do drugih resničnih sistemov (npr. pošiljanje elektronskega sporočila).
- Kreiranje objekta je prezahtevno (npr. podatkovna baza).
- Objekt še ne obstaja.

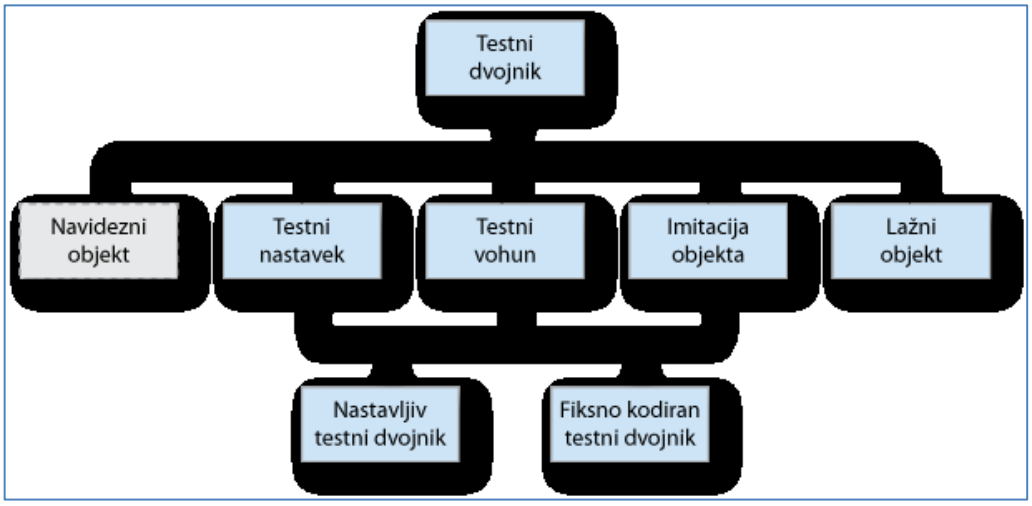
Obstaja več variant testnih dvojnikov (slika 6). Gerard Meszaros jih je definiral na naslednji način [17]:

- Navidezni objekti (*angl. dummy object*) so dvojniki, ki se jih samo posreduje (ponavadi kot parameter), vendar se jih ne uporablja.
- Lažni objekti (*angl. fake object*) imajo delujočo implementacijo, vendar uporabljajo neko bližnjico in so neuporabni za produkcijo (dober primer je podatkovna baza v pomnilniku).
- Testni nastavki (*angl. test stub*) so objekti, ki zamenjajo resnični objekt zato, da razredu, ki ga testiramo, lahko kontroliramo neposredne vhodne podatke. Testni nastavki nam omogočajo, da lahko testiran objekt postavimo v stanje, ki ga v resničnem svetu ne bi nikoli dosegel.

- Testni vohuni (*angl. test spy*) so izboljšana oblika testnega nastavka, ki lahko preverja tudi izhodne podatke testiranega objekta.
- Imitacije objektov (*angl. mock object*) so objekti, ki so programirani s pričakovanimi klici, ki naj bi jih prejeli. Imitacija objekta se razlikuje od testnega vohuna v tem, da testni vohun samo shranjuje klice, medtem ko imitacija objekta še preverja, če je bil klic pravilen. Imitacija objekta lahko poleg pravilnosti klicev preverja tudi vrstni red klicev.

Vedeti pa je potrebno, da se v različni literaturi, še posebno pa na spletu ta imena uporabljajo z različnimi pomeni. Najpogosteje se uporablja »mock« kar za vse testne dvojnike.

Testne dvojnike lahko implementiramo sami, največkrat pa se uporablja kar ogrodje za testne dvojnike (*angl. mock framework*). Obstaja veliko plačljivih in brezplačnih ogrodij za različne programske jezike.



Slika 6: Testni dvojniki

4.3.2 TESTIRAMO REZULTAT, NE IMPLEMENTACIJO

Kot rečeno, je imitacija objekta testni dvojnik, ki ima vnaprej programirane pričakovane klice in celo vrstni red klicev. Vendar pa moramo biti pri preverjanju vrstnega reda klicev zelo previdni. Ločiti je potrebno preverjanje zaporedja klicev, ki je rezultat nekega zunanjega zaporedja dogodkov, ki je kontroliran, in preverjanje zaporedja klicev, ki je rezultat interne strukture (**implementacije**)

enote, ki jo testiramo. Testirati moramo torej, kakšno spremembo (**rezultat**) povzroči testirana enota, na podlagi določenih vhodnih podatkov. Nikakor pa ne smemo testirati, **kako** je do teh rezultatov prišla.

4.4 POSLEDICE TESTNO VODENEGA RAZVOJA

Čeprav na prvi pogled izgleda, da je testno voden razvoj samo način, kako zagotovimo program s čim manj napakami, pa je ta način razvoja veliko več. Kmalu namreč ugotovimo, da so avtomatizirani testi pravzaprav stranski proizvod testno vodenega razvoja. Pomembno je razumeti, da obstaja sicer majhna, a zelo pomembna razlika med pisanjem testov pred pisanjem produkcijske kode in testno vodenim razvojem.

Poglejmo najprej prakso pisanja testov pred pisanjem produkcijske kode. Ta praksa dopušča, da najprej napišemo več testov in kasneje napišemo produkcijsko kodo. Ker je ta čas daljši, nam lahko teste napiše nekdo drug. Ponavadi celo je tako, saj »programerji nismo testerji«. S tem smo si podaljšali čas povratne informacije. Če nam teste piše nekdo drug, pa se zgodi še to, da začnemo pisati kodo, ki ni pokrita s testi, saj ne moremo prekiniti dela vsakič, ko je potrebno, da tester napiše nov test.

Majhno razliko, ki je značilna za testno voden razvoj, predstavljata prvi dve točki TDD mantre. Pozorni moramo biti na besedo »majhen«. Napisati moramo **majhen** avtomatiziran test in **najmanj kode**. Ta dva detajla nam zagotovita, da nimamo kode, ki ne bi bila pokrita s testi, oziroma je te kode kar se da malo. Še bolj pomembno pa je, da ne bomo napisali kode, ki je nismo načrtovali in posledično ne bomo napisali kode, ki je nihče ne potrebuje. Kot se programerjem, ki radi ustvarjamo, velikokrat zgodi, da napišemo nek del programa, za katerega mislimo, da ga bomo nekoč potrebovali, na koncu pa ga nihče ne rabi. Ta praksa nam omogoča, da smo zelo osredotočeni na problem, ki ga je potrebno rešiti, in ne na probleme, za katere predvidevamo, da jih bo nekoč potrebno rešiti.

4.5 ANALIZE TESTNO VODENEGA RAZVOJA

Na žalost je analiza testno vodenega razvoja zelo težka, saj se večina programerjev, ki ga uporabljajo, strinja, da se prednosti pojavijo šele čez nekaj časa. Največ analiz je bilo narejenih na poskusih, ki so jih izvajali z neizkušenimi programerji, neizkušenimi vsaj v smislu testno vodenega razvoja. Nekaj raziskav pa je bilo narejenih z izkušenimi programerji. Eno sta izvedla Boby George in Laurie Williams [18]. Izvedla sta jo s 24 programerji. Vsi so programirali v parih, kar je še ena aktivnost Extreme Programming metodologije [2]. Razdelili so se v dve skupini, ena je programirala po metodi testno vodenega razvoja, druga pa po tradicionalni metodi načrtovanje-razvoj-testiranje-razhroščevanje (*angl. waterfall method*).

Raziskava je pokazala naslednje prednosti testno vodenega razvoja:

- **18%** večja kvaliteta kode,
- **87,5%** razvijalcev poroča o boljšem razumevanju zahtev,
- **95,8%** razvijalcev poroča o manj razhroščevanja,
- **78%** razvijalcev poroča o izboljšanju skupne produktivnosti,
- **50%** razvijalcev ugotavlja zmanjšanje skupnega časa razvoja,
- **92%** razvijalcev ima občutek, da so producirali kvalitetnejšo kodo,
- **79%** razvijalcev verjame, da testno voden razvoj pomaga pri enostavnejšem načrtu.

Raziskava pa je pokazala tudi nekaj slabosti testno vodenega razvoja:

- **16%** daljši čas razvoja,
- **40%** razvijalcev ugotavlja, da je sprejetje testno vodenega razvoja težavno.

Raziskovalca sta zahtevala tudi to, da mora vsaka skupina na koncu oddati tudi avtomatizirane teste. Vendar pa je le ena skupina, ki je programirala po tradicionalni metodi predložila nekaj uporabnih testov. Dodaten čas pri testno vodenem razvoju bi lahko torej pripisali temu.

Pomembno je povedati, da so programerji programirali enostavno aplikacijo, ki je imela približno 200 vrstic kode. Prave prednosti bi se verjetno pokazale pri večji aplikaciji.

4.6 PREDNOSTI TESTNO VODENEGA RAZVOJA

Poleg že omenjenih prednosti testno vodenega razvoja je potrebno omeniti še druge, ki so jih neformalno odkrili nekateri pristaši testno vodenega razvoja [19].

4.6.1 DOBRO TESTIRANA KODA

Za kodo, ki ni bila načrtovana s testiranjem v mislih, je pisanje avtomatiziranih testov zelo težko ali celo nemogoče. Kot že povedano je potrebno testirano enoto v testih izolirati. In če ta enota ne uporablja vmesnikov, za katere bi lahko naredili testne dvojnike jo je praktično nemogoče izolirati. Če pa pišemo teste pred pisanjem kode, tega problema seveda nimamo.

Če se TDD-mantre strogo držimo, bi morali imeti celotno kodo pokrito s testi. Potrebno se je zavedati, da nekaterih stvari enostavno ne moremo testirati. Velikokrat moramo uporabiti aplikacijska ogrodja, ki jih nismo napisali sami. Tudi ta ogrodja so lahko napisana tako, da se jih ne da testirati. Vendar pa bi moralo biti glavno vodilo, da mora programer imeti dober razlog, zakaj nek del kode ni testiran.

4.6.2 ZMANJŠANA UPORABA RAZHROŠČEVALNIKA

Ker nam sami testi prikažejo napake kode, je uporaba razhroščevalnika (*angl. debugger*) skoraj nepotrebna. Veliki zagovornik testno vodenega razvoja Robert C. Martin celo pravi, da bi moral vsak programer uporabo razhroščevalnika jemati kot neuspeh [20].

4.6.3 POVEČANA PRODUKTIVNOST

Večina izkušenih uporabnikov testno vodenega razvoja se strinja, da se produktivnost poveča. To je pokazala celo raziskava, ki so jo naredili Hakan Erdogmus, Maurizio Morisio in Marco Torchianos [21]. To je sicer na prvi pogled nelogično, saj napišemo veliko več testov. Vendar pa se produktivnost poveča zato, ker je razvoj zelo osredotočen. To je razlog, da ne razvijamo funkcionalnosti, ki ni zahtevana.

4.6.4 NAČRTOVANJE

Testno voden razvoj ni samo testiranje pravilnosti kode ampak vodi načrtovanje aplikacije. Verjetno veliko škodo testno vodenemu razvoju dela beseda test v imenu [22]. Res je, da je del procesa pisanje testov, vendar testi niso cilj procesa [23]. Glavni namen procesa je osredotočen razvoj, za kar nam služijo testi. Torej je namen testov, da nam vodijo načrtovanje. S tega vidika lahko rečemo, da so testi pravzaprav specifikacija funkcionalnosti programa.

Nastajajoče načrtovanje (*angl. emergent design*) je praksa, ki z razvojem vsake nove funkcionalnosti dopolnjuje načrt [24]. Ker je pogoj nastajajočega načrtovanja čiščenje kode (*angl. refactoring*), je izvajanje te prakse brez testov praktično nemogoče. Zato sta praksa nastajajočega načrtovanja in testno voden razvoj tesno povezana.

4.6.5 FOKUS

Programer pri testno vodenem razvoju dela majhne korake. Ker je pri vsakem ciklu cilj uspeh testa, je zelo osredotočen na nalogo, ki jo poskuša rešiti. Vsaka vrstica kode mora imeti predhodno napisan test, zato je malo verjetno, da bo napisal kodo, ki je nepotrebna. Rezultat tega je skladnost s principom YAGNI (*angl. »You aren't gonna need it«* ali »Ne boš ga potreboval«). YAGNI princip je eden izmed principov ekstremnega programiranja [2].

4.6.6 CENEJŠI RAZVOJ

Čeprav pri testno vodenem razvoju napišemo več kode, pa je po modelu Müllerja in Padberga [25] celoten čas razvoja krajši. Veliko testov pomaga pri odpravi napak zgodaj v procesu razvoja. Odkrivanje napak zgodaj nam zmanjša potrebo po dolgotrajnem odkrivanju napak kasneje v času razvoja.

4.6.7 MANJ PODVOJENE KODE

Drugo pravilo testno vodenega razvoja nam pravi, da je potrebno odstraniti podvojeno kodo. To sovpada s principom DRY (*angl. »Don't repeat yourself«* ali »Ne ponavljaj se«), ki sta ga definirala Andy Hunt in Dave Thomas [26].

Poznamo več vrst podvojene kode [27]:

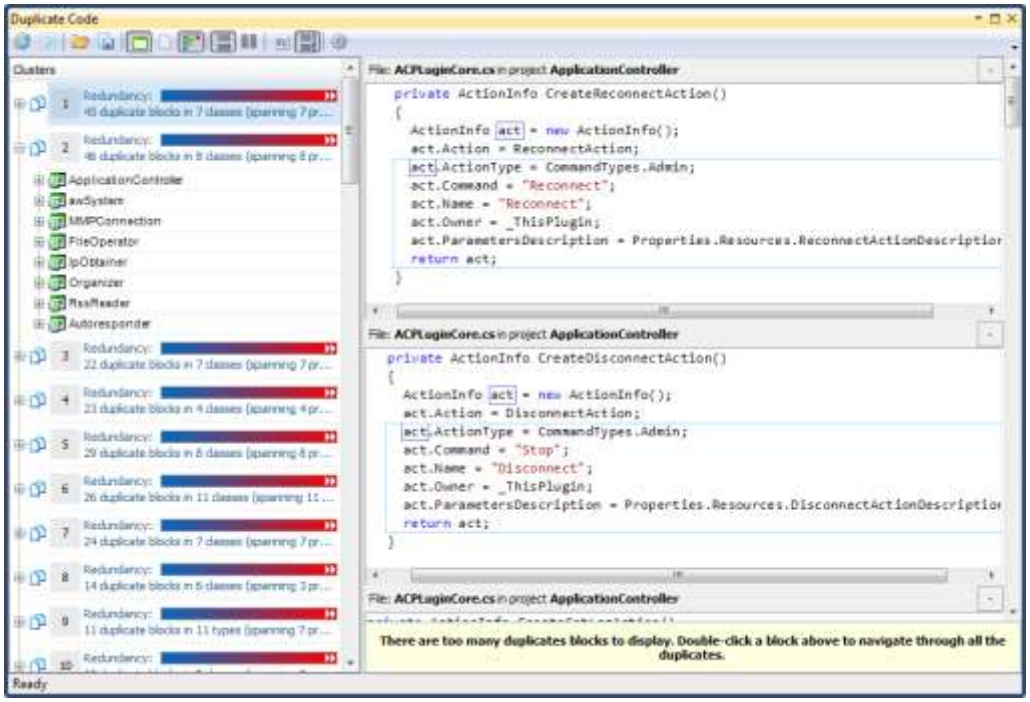
- Podvojen je vsak znak,
- Podvojen je vsak znak, razen praznih znakov in komentarjev,
- Podvojen je vsak člen (*angl. token*),
- Podvojen je vsak člen z manjšimi variacijami (npr. vstavljanje, brisanje, spreminjanje členov),
- Funkcijsko enaka koda.

Podvojena koda lahko povzroči povečane stroške vzdrževanja in povečano verjetnost napak. Če imamo podvojeno kodo in v enem primeru odkrijemo in odpravimo napako, se lahko zgodi, da v drugem primeru te napake ne odkrijemo.

Obstaja več načinov odkrivanja podvojene kode:

- Bakerjev algoritem [28]
- Algoritem Rabin–Karp [29]
- Uporaba abstraktnih sintaktičnih dreves [30]
- Vizualna detekcija podvojene kode [31]

Tudi nekatera moderna razvojna orodja imajo vgrajene detektorje podvojene kode. CodeRush proizvajalca DevExpress vsebuje orodje, ki podvojeno kodo odkrije in jo tudi odpravi (slika 7).



Slika 7: Detektor podvojene kode

4.6.8 ZANOS

Zanos (*angl. flow*) je stanje popolnega osredotočenja in usmerjene pozornosti na izvajanje določene mentalne ali fizične aktivnosti. Je trenutna, subjektivna izkušnja, pri čemer je bistvenega pomena posameznikovo dojemanje naloge, sebe in okolja. V stanju zanosa posameznik v celoti izkorišča svoj potencial, je notranje motiviran za opravljanje naloge, zmanjša se njegova stopnja samozavedanja in občutek za čas [32].

Večina programerjev je že doživela zanos. Ponavadi zanos doživimo takrat, ko nimamo zunanjih motenj. Zato programerji govorijo o zanosu, ki so ga doživeli pozno zvečer ali ponoči. Vendar pa lahko zanos doživimo tudi takrat, ko imamo zunanje motnje, če le te niso prepogoste. Največji problem pri motnjah je ta, da je čas, ko ponovno vstopimo v stanje zanosa, daljši. Zato Adam Tornhill govori o načinih, kako po motnji čim hitreje ponovno vstopimo v stanje zanosa [33]. In prav testno voden razvoj je eden izmed teh načinov. Tornhill pravi, da je priporočljivo pri delu pustiti za seboj sled. Ko nadaljujemo delo, lahko po tej sledi hitro ugotovimo, kje smo ga končali. Ker pri testno vodenem razvoju

napredujemo test za testom, je to prav ta sled, o kateri govori Tornhill. Priporočljivo je celo, da vedno končamo delo s testom, ki se ne izvede. Tako bomo kasneje takoj videli, kje smo končali delo.

4.7 Pomanjkljivosti TDD

4.7.1 Pokritost kode s testi

Testno voden razvoj ne izvaja zadosti testiranja v situacijah, kjer je potrebno celotno funkcionalno testiranje. Primeri, kjer testiranje enot ni mogoče, zajemajo uporabniške vmesnike, delo s podatkovnimi bazami in delo z omrežjem. V takih primerih je priporočljivo, da v take module damo najmanjšo količino kode. S tem večamo količino kode, ki je testabilna.

4.7.2 Teste piše razvijalec

Največkrat teste v testno vodenem razvoju piše isti razvijalec, ki napiše tudi kodo. To pa pomeni, da lahko razvijalec izpusti pomembno lastnost funkcije programa, tako v testih kot v kodi. Če na primer razvijalec ne ugotovi, da je potrebno preveriti neke določene vhodne parametre, jih verjetno tudi v kodi ne bo upošteval. Zato lahko pride do lažnega občutka varnosti, saj bodo testi uspešno izvedeni. Zato je zelo priporočljivo pri testno vodenem razvoju uporabiti še eno metodo ekstremnega programiranja, imenovano programiranje v parih (*angl. pair programming*) [2].

4.7.3 Pomanjkanje drugega testiranja

Rezultat testno vodenega programiranja je veliko število testov enot. To pa nam lahko da lažen občutek varnosti in pravilnosti celotnega programa. Pri testih enot se je potrebno zavedati, da nam le-ti testirajo enote v izoliranosti. To je zahteva, zato da pri neuspešnem testu hitro ugotovimo, kje se nahaja napaka.

Vendar pa ta zahteva povzroči, da so povezave med posameznimi enotami slabo testirane. Zato je nujno implementirati še druge oblike testiranja, kot so integracijski testi [34], testi komponent [35], sistemski testi [36], testi sprejemljivosti [37] itd.

4.7.4 VZDRŽEVANJE TESTOV

Testi enot postanejo del vzdrževanja projekta ali produkta. Slabo napisani testi lahko povzročijo še večje težave pri vzdrževanju programa. Če je teh testov veliko, obstaja nevarnost, da jih začnemo ignorirati in lahko se med te ignorirane teste skrije tudi tak, ki ga ne smemo ignorirati. Enako se lahko zgodi z opozorili prevajalnika. Ponavadi so opozorila neškodljiva za pravilnost kode, zato jih začnemo ignorirati. Vendar pa se občasno pojavi tudi kakšno opozorilo, ki je nevarno za pravilnost kode, in če ga ne odpravimo sproti, se nam tako pomembno opozorilo lahko izmuzne. Zato se je nujno zavedati, da so testi enakovredni kodi, ki jo testirajo, in je potrebno vložiti enako količino truda, da ostanejo lahko vzdržljivi.

4.7.5 PISANJE TESTOV VZAME ČAS

Pisanje in vzdrževanje testov vzame čas. Ker testno voden razvoj velja tudi pri kasnejših spremembah programa, je potrebno najprej spremeniti teste in nato kodo. Če testi niso napisani dobro ali je arhitektura programa slaba, je potrebno pri spremembah spremeniti veliko testov. Če se zgodi to, je potrebno teste popraviti enega po enega. Če bi se odločili, da bomo teste onemogočili ali jih pavšalno spremenili, se lahko zgodi, da povzročimo luknje v pokritju kode s testi. Zato je še posebno pomembno pravilo, da test preverja eno stvar.

4.7.6 PRETIRANO TESTIRANJE LAHKO POVZROČI NEPRIMERNO ARHITEKTURO

Pisanje kode, ki je testabilna, je včasih težko ali celo nemogoče. Ker testno voden razvoj zagovarja testiranje vsakega dela kode, se lahko zgodi, da postane

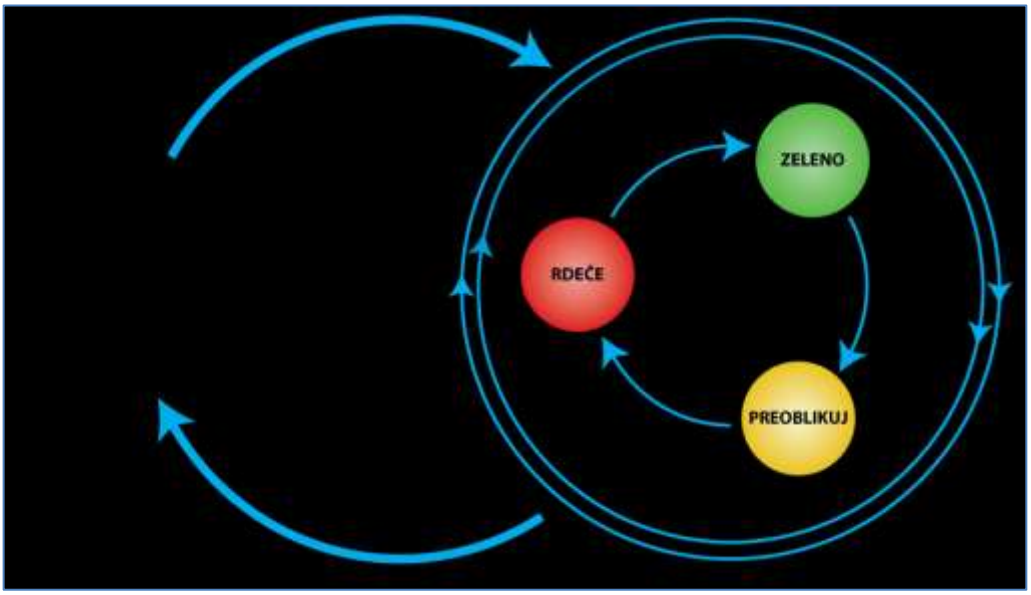
koda precej izrojena in povzroči neprimerno arhitekturo programa. Zato je potrebno uporabljati zdrav razum pri upoštevanju tega pravila. Na žalost ne obstaja veliko priporočil, ki bi nam pomagala pri teh odločitvah, pomaga nam le praksa.

4.8 VEDENJSKO VODEN RAZVOJ

Največji problem testno vodenega razvoja je verjetno beseda »test« v njegovem imenu. Ko ljudje, ki niso testerji, slišijo besedo »test«, se takoj odzovejo, da oni s tem nimajo nič. Še posebno to velja za netehnične ljudi, ki se ne ukvarjajo s programiranjem. Seveda pa bistvo testno vodenega razvoja ni testiranje.

Da bi ta problem omilil je Dan North pretvoril testno voden razvoj v vedenjsko voden razvoj (*angl. behavior driven development – BDD*). Postavil je nekaj pravil, ki bi testno voden razvoj izboljšala in ga na ta način približala tudi netehničnim ljudem. [38]

Testno voden razvoj, kot rečeno, testira enote. Da bi razvoj približali poslovnim ljudem, v vedenjsko vodenem razvoju pišemo teste sprejemljivosti (slika 8). V teh testih na višjem nivoju napišemo, kaj je potrebno, da bo uporabnik neko funkcionalnost sprejel kot končano.



Slika 8: Vedenjsko voden razvoj

Dan North je definiral nekaj pravil, ki naj bi odpravila večino problemov s testno vodenim razvojem.

4.8.1 IMENA TESTNIH METOD NAJ BODO STAVKI

Na ta način lahko razvijalci ustvarijo vsaj nekaj dokumentacije. Ker pa so stavki pisani v jeziku domene poslovnih ljudi, so proizvedeni dokumenti razumljivi tudi njim.

4.8.2 ENOSTAVNA STAVČNA PREDLOGA FOKUSIRA TESTE

Enostavno pravilo, da se imena testov začnejo z besedama »naj bi«(*angl. »should«*), nas prisili, da so testi narejeni le za trenuten razred. Torej nam pomaga, da se fokusiramo. Če poskušamo napisati test, katerega ime ne sodi v to predlogo, nam to nakazuje, da ta funkcionalnost sodi drugam.

4.8.3 Jasno izraženo ime testa nam pomaga, ko test ne uspe

Če nam ime testa pove, kaj naj bi testiral, nam to lahko pomaga pri odpravi napake. Če test ne uspe, je verjetno razlog eden izmed naslednjih:

- V kodo je bil vnesen hrošč. Rešitev: Odstranitev hrošča.
- Vedenje, ki ga test testira se je prestavilo drugam. Rešitev: Test je potrebno prestaviti na pravo mesto.
- Vedenje ni več pravilno, predpostavka sistema se je spremenila. Rešitev: Odstranitev testa.

To zadnje se v agilnih projektih dogaja, saj se zahteve spreminjajo, ko se veča razumevanje domene. Na žalost pa se začetniki zelo bojijo brisati teste. Ker uporabljamo izraz »naj bi«, ki je bolj mil, kot pa recimo »mora«, nam je lažje izzvati predpostavke vedenja, ki ga testiramo.

4.8.4 »Vedenje« je bolj uporabna beseda kot »test«

Dan North je uvidel, da se večina nejasnosti glede testno vodenega razvoja vedno vrača k besedi »test«. Saj so testi del testno vodenega razvoja, vendar pa nas lahko premami v lažen občutek varnosti, če testi ne testirajo vedenja sistema.

4.8.5 Ugotovi naslednje najpomembnejše vedenje

Če v vedenjsko voden razvoj uvedemo še en koncept, poslovna vrednost, lahko zahteve, ki še niso bile implementirane, razvrstimo po poslovni vrednosti. Tisto, ki ima največjo poslovno vrednost, bomo naslednjo implementirali. Tako si olajšamo vprašanje, na katerega je dostikrat težko odgovoriti: »kje začeti«.

4.8.6 BDD UVEDE JEZIK ZA VSE VKLJUČENE V ANALIZI

Ponavadi se v agilnem razvoju uporablja spodnja predloga za definiranje uporabniških zgodb (*angl. user story* [39]):

```
Kot [X]
hočem  [Y]
da bom lahko [Z]
```

```
As a [X]
I want [Y]
So that [Z]
```

kjer je Y neka zahteva, Z neka korist in X oseba, ki bo imela korist, ko bo zahteva realizirana. Prednost te predloge je, da nas prisili definirati korist. Če korist ni jasna, hitro začnemo uporabljati izraze, kot so »ker jo potrebujem«. To nam olajša opustitev nekih zahtev, ki očitno nimajo velike poslovne vrednosti.

Vedenje neke uporabniške zgodbe pa je pravzaprav kriterij za sprejemljivost. Tako je Dan North definiral še predlogo za zajem kriterija sprejemljivosti.

```
Če imamo nek začeten
kontekst,
Ko se zgodi nek dogodek,
Potem mora biti zagotovljen
rezultat.
```

```
Given some initial context
(the givens),
When an event occurs,
Then ensure some outcomes.
```

Na ta način definiramo različne scenarije neke uporabniške zgodbe.

4.8.7 KRITERIJ SPREJEMLJIVOSTI NAJ BO IZVRŠLJIV

Deli scenarija so dovolj majhni, da jih lahko predstavimo neposredno v kodi. Na ta način dobimo teste sprejemljivosti, ki jih lahko kadarkoli izvršimo in s tem preverimo, da sistem še vedno deluje tako, kot smo ga definirali.

4.9 ZVEZNA INTEGRACIJA

Rezultat testno vodenega razvoja ali vedenjsko vodenega razvoja je množica avtomatiziranih testov, torej testov, ki jih lahko izvršimo brez nadzora ljudi. Zato so ti testi idealen dodatek postopku zvezne integracije. Lahko bi celo rekli, da so glavna komponenta zvezne integracije (*angl. continuous integration – CI*).

Zvezna integracija je praksa v programiranju, kjer kodo različnih programerjev združimo, prevedemo in preverimo večkrat dnevno. Prvi jo je poimenoval Grady Booch v Boochovi metodi [40], vendar on ni predvidel integracije večkrat dnevno. Današnjo definicijo je predstavil Kent Beck kot del ekstremnega programiranja [2].

Namen zvezne integracije je preprečiti tako imenovani »integracijski pekel«(*angl. »integration hell«*). Ta se lahko zgodi, ko več razvijalcev dela spremembe na istem delu kode. Če je teh sprememb veliko, je potrebno veliko dela, da se te spremembe integrirajo v produkcijsko kodo.

V začetku je bil namen prakse preveriti kodo programerja na njegovem računalniku in jo šele potem izročiti na glavni repozitorij. Danes pa se prevajanje in preverjanje večinoma izvaja na prevajalnih strežnikih. Današnja praksa je taka, da programerji večkrat dnevno (oziroma vsaj enkrat) izročijo svoje spremembe na glavni repozitorij, takoj za tem pa se na prevajalnem strežniku avtomatsko izvrši prevajanje in preverjanje.

4.10 TESTIRANJE V JAVASCRIPTU

Kot rečeno, ima JavaScript veliko pomanjkljivosti, zaradi katerih lahko programer vnese v kodo hrošče, ki lahko ostanejo neodkriti, dokler nekdo te kode ne uporabi. Daljši ko je čas od nastanka hrošča do njegovega odkritja, večja je poslovna škoda [41]. Zato je skoraj nujno potrebno testirati JavaScript kodo in s tem hrošče odkriti čim prej.

Včasih smo spletne aplikacije oziroma spletne strani testirali tako, da smo napisali malo kode, osvežili spletno stran in ročno preverili, če je vse tako, kot mora biti. Ker obstaja veliko število brskalnikov, je to precej zamudno delo. Tudi če se omejimo le na glavne spletne brskalnike, je vseeno potrebno preveriti, če naša aplikacija deluje v vseh pomembnih verzijah teh brskalnikov.

Seveda tako testiranje ni realno izvedljivo. Zato je potrebno testiranje avtomatizirati. Obstaja ogromno orodij, ki nam to omogočajo.

Danes se JavaScript teste izvaja avtomatsko v spletnih brskalnikih in simulacijah brskalnikov.

5 OBSTOJEČE REŠITVE ZA JAVASCRIPT TESTIRANJE

Za testiranje JavaScript kode potrebujemo vsaj dve orodji: ogrodje za pisanje testov in izvajalnik testov. Vsako ogrodje že vsebuje tudi izvajalnik, vendar pa obstaja več izvajalnikov za posamezna ogrodja.

Ko začnemo z vsaj malo bolj zahtevnimi testi, hitro ugotovimo, da bomo potrebovali tudi knjižnico za pisanje testnih dvojnikov. Obstaja jih veliko, nekatera ogrodja za pisanje testov pa že vsebujejo lastne knjižnice za pisanje testnih dvojnikov.

5.1 OGRODJA ZA PISANJE TESTOV

Avtomatsko testiranje se je začelo z orodjem za SmallTalk, imenovanim Sunit, in z orodjem za testiranje v Javi JUnit. Pri obeh je imel ključno vlogo že omenjeni Kent Beck. Za popularizacijo vzorca testno vodenega razvoja, ki ga je definiral Kent Beck, je največ pripomogel JUnit, in zato se uporablja naziv xUnit za vsa ogrodja za testiranje, ki so grajena na osnovi JUnit ogrodja.

Drugi tip ogrodij pa ima za osnovo vedenjsko voden razvoj. V teh ogrodjih pišemo teste sprejemljivosti.

5.2 IZVAJALNIKI TESTOV

JavaScript testiranje se malo razlikuje od testiranja v drugih programskih jezikih, saj se skripte izvajajo v spletnih brskalnikih. Zato je potrebno teste izvesti v brskalniku ali pa simulirati brskalnik s tako imenovanim brezglavim brskalnikom. Zato lahko izvajalnike testov razdelimo v dve skupini.

Poleg te značilnosti pa se izvajalniki testov v JavaScriptu ločijo tudi po tem, da lahko izvajajo asinhrone teste.

5.3 BREZGLAVI BRSKALNIKI

Brezglavi brskalniki (*angl. headless browser*) so brskalniki, ki nimajo grafičnega vmesnika. Tak brskalnik dostopa do spletnih vsebin, vendar jih ne prikaže ljudem. Namenjen je, za posredovanje vsebin drugim računalniškim programom. Ker nima grafičnega vmesnika, je idealen za zaganjanje JavaScript testov [42].

5.4 KNJIŽNICE ZA PISANJE TESTNIH DVOJNIKOV

Ker je JavaScript dinamičen jezik, je pisanje enostavnih testnih dvojnikov praktično že vgrajeno v jezik sam. Da pa si olajšamo pisanje testnih dvojnikov, ki tudi preverjajo klice funkcij (testni vohuni, testni nastavki in imitacije objektov), imamo na voljo več knjižnic za pisanje testnih dvojnikov. V glavnem se razlikujejo po tem, da preverjajo klice po opravljenem testu ali pa moramo predhodno nastaviti pričakovane klice.

5.5 OBSTOJEČA ORODJA ZA TESTIRANJE JAVASCRIPT KODE

5.5.1 JSUNIT

To je prvo ogrodje za testiranje JavaScripta. Namenjen je za pisanje xUnit stila testov, vendar se ga danes ne vzdržuje več aktivno.

Teste lahko izvaja le v brskalniku.

5.5.2 QUNIT

Uporabljamo ga v jQuery projektu, vendar pa ga lahko uporabljamo za testiranje katerekoli JavaScript kode. Tudi QUnit je ogrodje za pisanje xUnit stila testov. Metode za preverjanje v QUnit ogrodju sledijo specifikaciji CommonJS [43]. QUnit je originalno razvil John Resig kot del jQuery knjižnice. Leta 2008 pa so ga ločili iz jQuery knjižnice in ustvarili samostojen projekt, imenovan QUnit. Originalna verzija je sicer uporabljala jQuery za interakcijo z DOM, vendar so jo leta 2009 predelali in jo ustvarili popolnoma samostojno in neodvisno od jQuery knjižnice.

Teste lahko izvaja tako v brskalniku kot v brezglavem brskalniku.

5.5.3 YUI TEST

YUI Test je ogrodje, ki je del knjižnice YUI in ga je ustvarilo podjetje Yahoo. Čeprav je del YUI knjižnice, ga lahko uporabljamo samostojno za testiranje katerekoli JavaScript kode. Kot sami pravijo, YUI Test ni neposredna pretvorba JUnit ogrodja, vendar uporablja nekatere karakteristike JUnita in nUnita, zato ga lahko uvrstimo med xUnit tip ogrodja. Ogrodje nUnit je implementacija xUnit ogrodja za .Net okolje, napisana v C#.

Teste lahko izvaja tako v brskalniku kot v brezglavem brskalniku.

5.5.4 MOCHA

Je dokaj novo ogrodje, saj se je razvoj začel leta 2011. Podpira tako xUnit stil pisanja testov kot tudi BDD stil. Zasnovano je izredno modularno in razširljivo, zato obstaja veliko dodatkov.

Teste lahko izvaja tako v brskalniku kot v brezglavem brskalniku.

5.5.5 JASMINE

Prva verzija je bila izdana leta 2010. Glavno vodilo Jasmine ogrodja je, da morajo biti testi enostavni za branje. Sintaksa je podobna sintaksi RSpec, ki je ogrodje za pisanje testov stila BDD v programskem jeziku Ruby. Vsebuje tudi svojo knjižnico za pisanje testnih dvojnikov.

Teste lahko izvaja tako v brskalniku kot v brezglavem brskalniku.

5.5.6 BUSTER.JS

Je ogrodje za testiranje JavaScript kode, ki se lahko izvaja v brskalniku in v brezglavem brskalniku. Teste zna izvajati tudi v Node.js okolju. Zelo je fleksibilen in ima javen API za skoraj vse. Lahko celo zamenjamo sintakso pisanja testov. Je odprtokoden projekt, tako ima poleg avtorjev Augusta Lilleaasa in Christiana Johansna veliko drugih, ki so prispevali kodo. Del projekta je tudi več uporabnih knjižnic, ki jih lahko uporabljamo ločeno od ogrodja Buster.js.

Trenutno je še v verziji beta, torej lahko vsebuje napake. Imajo pa že velike načrte za naprej:

- Izvajanje testov neposredno na sistemu BrowserStack.
- Zaganjalnik testov, ki ve, kateri testi so bili predhodno neuspešni, in tako lahko izvede le te teste.
- Prekinitvene točke za teste, ki vas vržejo naravnost v brskalnikovo orodje za razhroščevanje.

5.5.7 JsMockito

JsMockito je knjižnica, ki jo uporabljamo izključno za pisanje testnih dvojnikov. Pričakovane operacije preverimo po izvedbi testne kode. Odlikuje jo lahka integracija z drugimi ogrodji in podpira simuliranje metod objektov in funkcij. Ima pa slabost, da je odvisna od knjižnice jsHamcrest, ki je namenjena ugotavljanju enakosti parametrov.

5.5.8 jqMock

Knjižnico za pisanje testnih dvojnikov jqMock uporabljamo izključno z ogrodjem jqUnit, ovojem za QUnit ogrodje. Pričakovane operacije je potrebno predhodno definirati.

5.5.9 Sinon.JS

Je samostojna knjižnica za pisanje testnih dvojnikov, ki deluje z vsakim testnim ogrodjem. Ustvaril jo je Christian Johansen, ki je tudi avtor knjige Test-Driven JavaScript Development.

5.5.10 PhantomJS

PhantomJS je brezglavi brskalnik. Zgrajen je na osnovi WebKit tehnologije, ki je osnova za brskalnik Safari, in je bila osnova za brskalnik Chrome [44]. Je hiter in ima vgrajeno podporo za različne spletne standarde: DOM, CSS, JSON, Canvas in SVG [45].

5.5.11 Node.JS

Node.js [46] je odprtokodna platforma, zgrajena na osnovi Googlovega V8 izvajalnika JavaScript kode. Node.js lahko izvajamo na več operacijskih sistemih: Windows, Linux, OS X in FreeBSD.

Namenjena je enostavni gradnji nadgradljivih mrežnih aplikacij. Uporablja asinhroni dogodkovni model brez blokiranja, zaradi katerega je lahkotna in učinkovita. Uporabna je za podatkovno intenzivne aplikacije v realnem času, ki se izvajajo na porazdeljenih napravah.

Node.js ima vgrajeno knjižnico, s katero se lahko aplikacije predstavijo kot spletni strežnik, brez pomoči aplikacij, kot sta Apache HTTP Server ali IIS. Vse bolj je popularna kot strežniška platforma in jo uporabljajo velika podjetja: Groupon, SAP, LinkedIn, Microsoft, Yahoo!, Walmart, Rakuten in PayPal.

Uporabnost je še posebno izboljšalo orodje npm (Node.js Package Manager), s katerim osnovni Node.js inštalaciji dodajamo pakete, ki so na voljo v spletnem npm registru.

5.5.12 TESTEM

Je samostojen zaganjalnik testov, ki podpira različna ogrodja. Vgrajeno ima podporo za Jasmine, QUnit, Mocha in Buster.js. Ima pa možnost razširitve s pomočjo adapterjev za testna ogrodja. Teste lahko zaganja v vseh popularnih brskalnikih ter v PhantomJS in Node.js. Zato sta možna dva različna načina uporabe: kot pomoč pri testno (ali vedenjsko) vodenem razvoju ter kot del zvezne integracije. Ker je narejen na osnovi Node.js, je podprt na Windows, OS X in Linux operacijskih sistemih. Zasnovan je modularno, zato obstaja veliko razširitev za različna testna ogrodja, generatorje poročil in prožilnike.

5.5.13 KARMA

Karma je samostojen zaganjalnik testov, ki je zgrajen na platformi Node.js. Nastal je kot del projekta AngularJS, ki je trenutno najbolj uspešna platforma za gradnjo spletnih aplikacij.

Glavni cilj Karme je priskrbeti razvijalcem produktivno testno okolje. Torej okolje, kjer ni potrebno ogromno konfiguriranja, ampak lahko razvijalci pišejo kodo in dobijo hiter odziv njihovih testov. Saj jih hiter odziv naredi produktivne in kreativne. [47]

Karma je orodje, ki zažene testno kodo na vseh priključenih brskalnikih. Rezultati testov so zbrani in prikazani v orodni vrstici, tako da lahko vidimo, v katerem brskalniku se nek test je ali ni uspešno izvedel. Rezultate iz brskalnika lahko Karma pridobi tako, da:

- se brskalnik priključi na Karma strežnik ročno,
- Karma sama zažene brskalnike, ki jih predhodno nastavimo.

Karma tudi nadzira vse datoteke, ki jih nastavimo v nastavitvah in zažene vse teste, tako da sporoči priključenim brskalnikom, da je potrebno zagnati teste. Brskalnik nato teste izvede in sporoči Karma strežniku rezultate.

5.5.14 CHUTZPAH

Chutzpah [48] je testni zaganjalnik, ki omogoča izvrševanje testov z uporabo orodij QUnit, Jasmine, Mocha, CoffeeScript in TypeScript. Uporablja PhantomJS brezglavi brskalnik za izvrševanje testov. Zanimivo je, da je integriran v Microsoftov Visual Studio.

5.5.15 TESTSWARM

TestSwarm omogoča distribuirano zvezno integracijo za JavaScript. Naredil ga je John Resig, kot osnovno orodje za testiranje enot jQuery JavaScript knjižnice.

Osnovni cilj TestSwarma je poenostavitev kompliciranega in dolgotrajnega procesa izvajanja JavaScript testov v več spletnih brskalnikih. To dosega tako, da priskrbi orodja, potrebna za potek dela zvezne integracije JavaScript projekta. [49]

5.5.16 BrowserStack

BrowserStack [50] je sistem za testiranje JavaScript kode v oblaku. Preko njegovih strežnikov lahko našo JavaScript kodo testiramo na različnih platformah in spletnih brskalnikih.

Podprte platforme:

- Apple iOS
- Google Android
- Opera Mobile
- Windows XP
- Windows 7 (32 bit in 64 bit)
- Windows 8
- Windows 8.1
- Mac OS X Lion
- Mac OS X Mavericks
- Mac OS X Mountain Lion
- Mac OS X Snow Leopard
- Mac OS X Yosemite Beta

Podprti brskalniki (vsi niso na voljo na vseh platformah):

- Mozilla Firefox
- Google Chrome
- Internet Explorer
- Opera
- Safari

6 KAJ JE ALLGREEN

Ko sem se začel ukvarjati s testno vodenim razvojem v JavaScriptu, sem začel uporabljati nekatera že opisana orodja. Na koncu sem pristal na uporabi ogrodja Jasmine za pisanje testov in zaganjalnika Karma za avtomatsko izvrševanje testov. Jasmine sem dodal še dodatek Jasmine-jQuery [51] za preverjanje jQuery operacij. S tem sem lahko uspešno pisal JavaScript teste.

Karma pa ima po mojem mnenju pomanjkljivost pri izpisovanju rezultatov. Rezultate izpisuje v orodno vrstico, kar je pri veliki količini testov zelo nepregledno. Prva ideja, ki mi je prišla na misel, je bila, da bi napisal svoj generator poročil, saj ima Karma možnost razširitev. Vendar tudi to ni bilo tako preprosto, kot se sliši. Problem je, da Karma veliko podatkov, ki jih sprejme od testnega ogrodja, ne posreduje naprej generatorju poročil.

Drugo pomanjkljivost pa sem ugotovil, ko sem začel programirati v TypeScriptu. Ker TypeScript generira JavaScript, je možno za testiranje uporabiti vsa orodja za testiranje JavaScripta. Vendar pa je zelo zamudno ugotavljanje, kje v skripti TypeScripta je prišlo do napake, saj Karma ne uporablja povezovalnih datotek, s katerimi bi lokacijo v generirani JavaScript kodi pretvorila v lokacijo v TypeScript kodi.

Tako sem dobil navdih, da začnem pisati svoj zaganjalnik testov, ki sem ga poimenoval AllGreen. Konec koncev je cilj, ki si ga vsi želimo, da bi bili vsi rezultati testov obarvani zeleno.

7 ARHITEKTURA SISTEMA ALLGREEN

Aplikacija je zgrajena iz dveh delov: strežnika in odjemalca. Odjemalcev, ki se priključijo na strežnik, je lahko več. Strežnik in odjemalec med seboj dvosmerno komunicirata po HTTP protokolu. Večinoma se odjemalci izvajajo na istem računalniku kot strežnik, vendar ni nujno tako. Odjemalec se lahko izvaja celo na mobilnih napravah, kot so pametni telefoni in tablični računalniki.

Na arhitekturo in izbiro tehnologij so vplivali naslednji dejavniki:

- Hitrost,
- Zanesljivost,
- Testi naj se izvajajo tam, kjer se bo izvajala produkcijska koda,
- Uporaba pri testno vodenem razvoju,
- Možnost razširitev in zamenjave delov sistema,
- Delno tudi uporaba pri razvoju z orodjem Visual Studio.

Strežnik je napisan v programskem jeziku C# z uporabo .Net tehnologije. Odjemalec, ki se izvaja v brskalniku, pa je napisan z uporabo TypeScript programskega jezika.

7.1 ČEBULNA ARHITEKTURA

Pri razvoju strežnika in razvoju odjemalca sem uporabil t.i. čebulno arhitekturo (*angl. onion architecture*) [52]. Koncept je znan pod več imeni: čista arhitektura (*angl. clean architecture*) [53], heksagonalna arhitektura (*angl. hexagonal architecture*) [54], DCI [55] in BCE [56].

Tradicionalna večplastna arhitektura [57] je sestavljena iz več plasti, ki imajo svoje odgovornosti (slika 9).

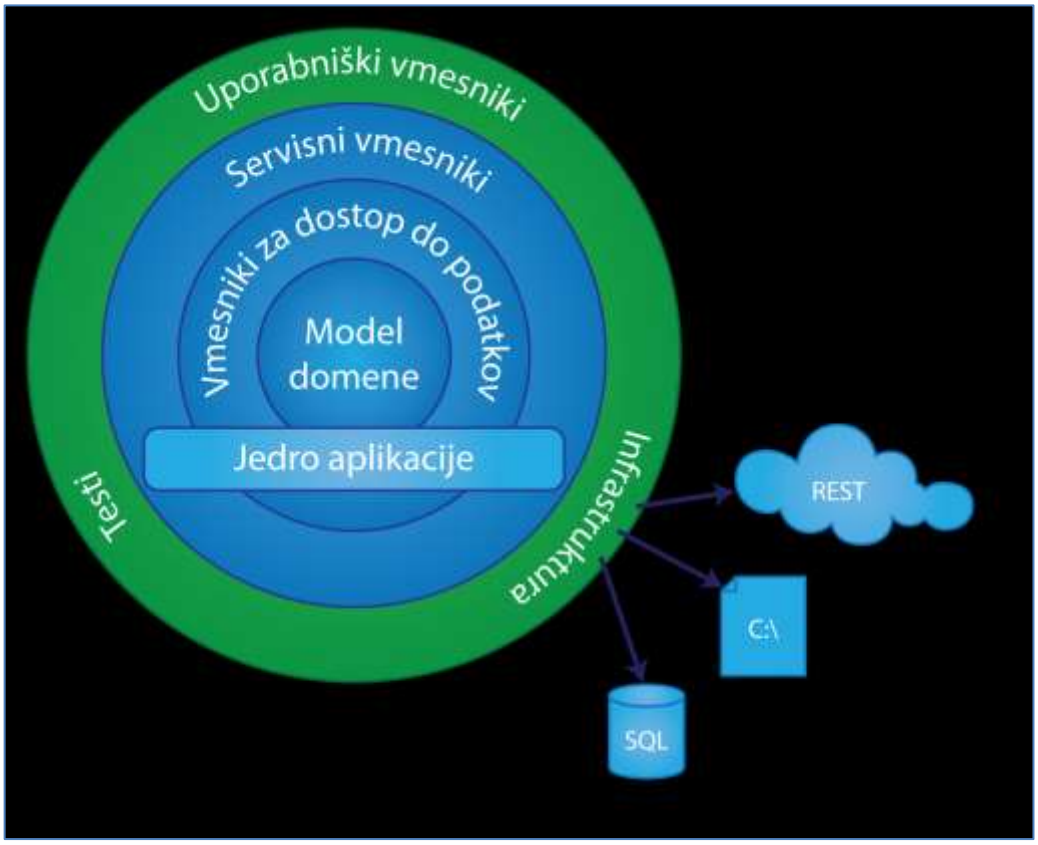
Slika 9: Tradicionalna večplastna arhitektura

Najpogostejše plasti (od vrha do dna) so:

- predstavitvena plast (*angl. presentation layer*)
- aplikacijska plast (*angl. application layer*)
- plast poslovne logike (*angl. business logic layer*)
- plast za dostop do podatkov (*angl. data access layer*)

Pri tem je vsaka plast odvisna od plasti pod njo. To pomeni, da je ponavadi potrebno plast spremeniti, če se spremeni karkoli v katerikoli plasti pod njo. Tu pa nastopi problem. Najnižja plast je plast za dostop do podatkov, kar pomeni, da so vse plasti odvisne od nje. Način dostopa do podatkov se zgodovinsko spreminja vsake tri leta. Torej bi morali aplikacijo popolnoma spremeniti vsake tri leta, če bi hoteli uporabljati najsodobnejšo tehnologijo.

Rešitev ponuja čebulna arhitektura (slika 10). Pri tej arhitekturi plasti niso naložene ena na drugo, ampak so razporejene v koncentrične kroge. Odvisnost plasti je usmerjena proti centru krogov, torej je plast, ki je bolj zunaj, odvisna od plasti v notranjosti.



Slika 10: Čebulna arhitektura

V center torej postavimo plast, ki se najmanj spreminja – model domene. V drugi plasti od sredine so vmesniki za dostop do podatkov. V naslednji so servisni vmesniki. Te tri plasti sestavljajo jedro aplikacije. V plasti, ki je najbolj zunaj, pa so uporabniški vmesniki, infrastruktura in povezava s podatkovno bazo. To nam omogoča, da lahko zamenjamo tehnologijo grafičnega vmesnika ali podatkovno bazo, ne da bi s tem vplivali na ostale dele sistema.

Čebulna arhitektura se močno zanaša na princip inverzije odvisnosti (*angl. dependency inversion principle* [58]). Plast, ki je bolj proti centru, vsebuje le vmesnike, preko katerih jo lahko plast, ki je bolj zunaj, uporablja. Zato potrebuje implementacije teh vmesnikov. Te implementacije pa pridejo iz zunanjih plasti.

Na primer servisna plast potrebuje implementacije vmesnikov za dostop do baze. Vmesniki so v drugi plasti, implementacije pa so v zunanji plasti, kjer je podatkovna baza. Za rešitev tega problema se ponavadi uporablja injiciranje odvisnosti (*angl. dependency injection* [59]).

7.2 STREŽNIK (*SERVER*)

Strežnik je glavni del sistema. Ima podatke o vseh datotekah v projektu, o vseh priključenih odjemalcih in o statusu testov. Zadolžen je, da:

- komunicira z odjemalci,
- sporoča rezultate testov razvijalcu,
- posreduje spletne datoteke,
- nadzira spremembe opazovanih datotek in glede na to sproži izvajanje testov.

Večinoma se strežnik izvaja na istem računalniku, kot razvojnik razvija program, ki ga testira. Vendar pa se lahko strežnik izvaja na drugem računalniku, le dostop do datotek mora imeti.

Strežnik je sestavljen iz petih komponent, to so:

- spletni strežnik
- vozlišče za izvajalce testov
- oddajnik za izvajalce testov
- poročevalec rezultatov
- nadzornik datotek

Vse komponente so popolnoma ali delno zamenljive.

7.2.1 SPLETNI STREŽNIK (*WEBSERVER*)

Spletni strežnik je zadolžen za dostavo vseh statičnih podatkov odjemalcem. Statični podatki so:

- JavaScript in HTML datoteke za odjemalčev zaganjalnik
- izvorna koda ogrodja za testiranje
- izvorna koda testov
- izvorna koda, ki jo testiramo.

Trenutna implementacija spletnega strežnika je zgrajena na osnovi OWIN tehnologije. Vendar pa je arhitektura sistema zasnovana tako, da je možna tudi druga implementacija spletnega strežnika.

7.2.2 VOZLIŠČE ZA IZVAJALCE TESTOV (*RUNNERHUB*)

Vozlišče za izvajalce testov je odgovorno za dvosmerno komunikacijo z odjemalci. Sporočila, ki jih sprejema, so:

- Reset – ponastavitev izvajalca testov
- Started - začetek izvajanja testov
- SpecUpdated - test končan
- Finished - vsi testi končani
- Register - registriraj izvajalca testov

Zadolženo pa je tudi za posredovanje sprejetih sporočil poročevalcu rezultatov.

Dvosmerna komunikacija je realizirana z uporabo SignalR tehnologije.

7.2.3 ODDAJNIK ZA IZVAJALCE TESTOV (*RUNNERBROADCASTER*)

Oddajnik za izvajalce testov je zadolžen za pošiljanje sporočil vsem odjemalcem. Vsem odjemalcem pošlje sporočilo, ko je potrebno ponovno prenesti in izvesti teste.

Trenutno je to tudi edino sporočilo, ki ga pošlje odjemalcem. Vendar pa bi lahko pošiljal tudi druga sporočila, npr. stanje drugih izvajalcev ali razlog za zagon testov.

7.2.4 Poročevalec rezultatov (*Reporter*)

Poročevalec rezultatov je zadolžen, da rezultate, ki jih prejme od vozlišča za izvajalce testov, na nek uporaben način posreduje programerju. Lahko jih prikaže na ekran, zapiše v neko datoteko, pošlje po elektronski pošti, posreduje na neko spletno mesto in še marsikaj drugega si lahko zamislimo.

Trenutna implementacija omogoča prikazovanje v konzoli ali prikazovanje v organizirani drevesni strukturi. Prikaz v konzoli je izredno nepregleden, tako kot pri orodju Karma, prikaz v oknu z uporabo drevesa pa je veliko boljši. Ta vmesnik je bolj podrobno opisan kasneje.

7.2.5 Nadzornik datotek (*FileWatcher*)

Nadzornik datotek je zadolžen za opazovanje datotek. Ko opazi, da je prišlo do spremembe neke datoteke, to sporoči oddajniku za izvajalce testov. Oddajnik nato sporoči vsem izvajalcem, naj ponovno izvedejo vse teste.

Nadzornik datotek je neobvezna komponenta, saj lahko zagon testov sprožimo tudi na druge načine. Teste lahko sprožimo na primer s pritiskom na gumb na grafičnem vmesniku ali s tem, ko ponovno naložimo spletno stran na odjemalcu. Že ob prvi registraciji odjemalca na strežnik se zaženejo testi.

Trenutna implementacija opazuje datotečni sistem. Odzove se, ko shranimo datoteko, naredimo novo, jo zbrišemo ali preimenujemo.

Arhitektura sistema pa je zasnovana tako, da lahko nadzornika implementiramo tudi na druge načine. Lahko si zamislimo nadzornika datotek, ki opazuje sistem za nadzor različic (npr. Git). Zalo uporaben nadzornik bi bil tudi tak, ki bi se povezal z Visual Studiom in od njega sprejemal sporočila o spremembah datotek.

7.2.6 UPORABLJENE TEHNOLOGIJE

Pri implementaciji strežnika sem v osnovi uporabil ASP.NET [60] tehnologijo podjetja Microsoft. Spletni strežnik je implementiran z uporabo OWIN tehnologije, ki omogoča enostavno menjavo strežnika za gostovanje. Za dvosmerno komunikacijo z odjemalcem sem uporabil tehnologijo SignalR.

Testiranje je realizirano z uporabo Microsoftovega testnega ogrodja MSTest [61]. Za lažje preverjanje rezultatov sem uporabil knjižnico Fluent Assertions. Za pisanje testnih dvojnikov sem uporabil knjižnico Moq.

OWIN in projekt Katana [62]

OWIN (Open Web Interface for .NET) je odprt standard, ki definira vmesnik med .NET spletno aplikacijo in spletnim strežnikom. Pred tehnologijo OWIN je bila Microsoftova ASP.NET tehnologija zasnovana na osnovi spletnega strežnika IIS in aplikacij ni bilo mogoče enostavno poganjati na drugih spletnih strežnikih. Namen OWIN-a pa je razdvojiti povezavo med ASP.NET aplikacijami in IIS strežnikom z definicijo standardnega vmesnika. Prav tako je mogoče razviti nova spletna aplikacijska ogrodja kot alternativo ASP.NET.

Projekt Katana je množica OWIN komponent, ki jih je zgradil Microsoft.

OWIN pa poleg ločitve med aplikacijskim ogrodjem in spletnim strežnikom omogoča tudi veriženje komponent. Tako lahko spletno ogrodje komunicira z OWIN-om, ne da bi vedelo, ali komunicira neposredno s spletnim strežnikom ali z več nivoji nad njim. Tako se lahko infrastruktura kot na primer preverjanje identitete loči v svoj modul, izloči iz aplikacijske kode in uporabi v več aplikacijah.

SignalR [63]

ASP.NET SignalR je knjižnica za ASP.NET, ki poenostavi komunikacijo med strežnikom in odjemalcem v realnem času. Omogoča, da lahko strežnik posreduje sporočila priključenim odjemalcem takoj, ko se zgodijo.

Uporablja WebSocket, ki je del nove HTML5 API specifikacije, ki omogoča dvosmerno komunikacijo med spletnim strežnikom in spletnim brskalnikom. Ko

WebSocket ni na voljo, pa SignalR uporabi druge tehnike in tehnologije. To se vse dogaja skrito razvijalcu, tako ostane aplikacijska koda enaka.

Caliburn.Micro [64]

To je ogrodje, ki omogoča enostavno gradnjo aplikacij v XAML [65] platformah. Poenostavi nam gradnjo aplikacij z uporabo MVVM [66] arhitekture grafičnega vmesnika.

TinyIoC [67]

Kontejner za inverzijo kontrole (*angl. inversion of control (IoC) container*) je komponenta, ki omogoča uporabo principa inverzije odvisnosti. TinyIoC je enostavna knjižnica, ki implementira tako komponento. Čeprav je uporaba enostavna, pa omogoča veliko.

Inverzija kontrole je zgradba programa, kjer se kontrola obrne, če gledamo klice med generično (jo lahko večkrat uporabimo) in specializirano kodo. V tradicionalni zgradbi specializirana koda kliče metode v generični kodi (npr. knjižnice ali ogrodja). Pri inverziji kontrole pa generična koda kliče metode v specializirani kodi.

7.3 ODJEMALEC (*CLIENT*)

Odjemalec je del sistema, na katerem se izvajajo testi. Največkrat je to spletni brskalnik na računalniku, tabličnem računalniku ali pametnem telefonu. Večinoma se izvaja na istem računalniku kot strežnik, vendar pa se lahko izvaja kjerkoli, le dostop do strežnika preko HTTP protokola mora imeti.

Istočasno je lahko na strežnik priključenih več odjemalcev.

Odjemalec je sestavljen iz petih komponent, to so:

- App –JavaScript aplikacija
- Hub - vozlišče za komunikacijo s strežnikom
- Reporter - poročevalec rezultatov

- Adapter - adapter za povezavo s testnim ogrodjem
- Izvorna koda testov in programa, ki ga testiramo.

7.3.1 JavaScript aplikacija (*App*)

JavaScript aplikacija je odgovorna za povezavo vseh delov odjemalca. Ko se odjemalec poveže s strežnikom, se aplikacija inicializira in se izvaja, dokler uporabnik odjemalca (brskalnik) ne ugasne. Ostali deli odjemalca se v aplikacijo registrirajo in jih zato lahko zamenjamo z drugimi implementacijami.

7.3.2 Vozlišče za komunikacijo s strežnikom (*Hub*)

Vozlišče za komunikacijo s strežnikom je odgovorno za dvosmerno komunikacijo s strežnikom. Za komunikacijo se uporablja SignalR JS Client tehnologija. Iz strežnika sprejema sporočila za zagon testov, ki jih posreduje aplikaciji. Od adapterja za testno ogrodje pa sprejema sporočila o stanju izvajanja testov, ki jih posreduje strežniku.

7.3.3 Poročevalec stanja (*Reporter*)

Poročevalec stanja je namenjen prikazu stanja na brskalniku. Sestavljen je iz dveh poročevalcev: poročevalca stanja strežnika (*ServerReporter*) in poročevalca stanja izvajanja testov (*RunnerReporter*).

Poročevalec stanja strežnika prejema stanje od vozlišča za komunikacijo s strežnikom. Možna stanja so:

- Connected – ko se vozlišče uspešno priključi na strežnik,
- Disconnected – ko se vozlišče odklopi s strežnika,
- Error – ko pride do napake pri komunikaciji s strežnikom,
- Reconnecting – ko se vozlišče poskuša samodejno ponovno priključiti na strežnik,
- Reconnected – ko se vozlišče uspešno ponovno priključi na strežnik.

Poročevalec stanja izvajanja testov prejema sporočila od adapterja za testno ogrodje. Sporočila, ki jih prejme, so:

- Reset – ponastavitev izvajanja testov
- Started – začetek izvajanja testov
- SpecUpdated – zaključek posameznega testa
- Finished – zaključek izvajanja testov

7.3.4 TESTNO OGRODJE IN ADAPTER ZA TESTNO OGRODJE (*ADAPTER*)

Testno ogrodje ni del AllGreen aplikacije. Povezava s testnim ogrodjem je realizirana z adapterjem. Zaradi tega je možno implementirati povezavo s poljubnim testnim ogrodjem, ki se lahko izvaja v brskalniku. Trenutno je podprto testno ogrodje Jasmin.

Adapter stanja in rezultate izvajanja testov posreduje poročevalcu stanja izvajanja testov in vozlišču za komunikacijo s strežnikom. Zato imata enak vmesnik in adapter ne razlikuje med njima. Tako se vozlišče za komunikacijo s strežnikom v adapter registrira kot poročevalec. Adapter enostavno posreduje stanja in rezultate vsem prijavljenim poročevalcem.

7.3.5 IZVORNA KODA TESTOV IN PROGRAMA, KI GA TESTIRAMO

Ko odjemalec od spletnega strežnika zahteva datoteke za zagon testov, strežnik oblikuje seznam teh datotek in ga posreduje odjemalcu. Odjemalec nato od strežnika zahteva vse datoteke s tega seznama. Tako strežnik nadzoruje, katere teste in katero kodo bo odjemalec testiral.

7.3.6 UPORABLJENE TEHNOLOGIJE

Pri implementaciji odjemalca sem uporabil standardne spletne tehnologije HTML in CSS. Za lažjo definicijo CSS stilskih podlog sem uporabil orodje LESS.

Dinamični del je bil sprogramiran z uporabo programskega jezika in orodja TypeScript.

Za dvosmerno komunikacijo s strežnikom sem uporabil tehnologijo SignalR JS Client. SignalR JS Client temelji na ogrodju jQuery. jQuery sem uporabil tudi sam za manipulacijo DOM modela.

Testiranje implementacije je bilo realizirano z uporabo ogrodja Jasmine in dodatka Jasmine-jQuery, ki je namenjeno testiranju jQuery aplikacij.

jQuery

jQuery je JavaScript knjižnica, ki olajša pisanje skript za dinamične HTML strani. Trenutno ga uporablja približno 60% vseh internetnih strani, kar pomeni, da je najbolj uporabljana JavaScript knjižnica. V osnovi je jQuery knjižnica za manipulacijo DOM modela. Vendar pa omogoča lahko programiranje animacij, efektov, krmiljenje dogodkov, asinhronih nalog in marsikaj drugega. Omogoča tudi enostavno dodajanje razširitev.

Verjetno največja prednost in razlog za popularizacijo pa je transparentna podpora več različnih verzij brskalnikov. Pred jQuery knjižnico so morali programerji pisati različne skripte za brskalnike različnih proizvajalcev. Z jQuery knjižnico pa lahko pišemo skripte, ki delujejo na vseh (testiranih) brskalnikih. [68]

SignalR JS Client

To je JavaScript SignalR odjemalec, ki omogoča dvosmerno komunikacijo s strežnikom.

CSS in LESS

CSS so kaskadne stilske podloge (angl. cascading style sheets), predstavljene v obliki preprostega slogovnega jezika, ki skrbi za predstavitev spletnih strani. Z njimi določimo stil HTML oz. XHTML elementov v smislu pravil, kako naj se ti prikažejo na strani. Določamo lahko barve, velikosti, odmike, poravnave, obrobe, pozicije in vrsto drugih atributov, prav tako pa lahko nadziramo aktivnosti, ki jih uporabnik nad elementi strani izvaja (npr. prekritje

povezave z miško). Podloge so bile razvite z namenom konsistentnega načina podajanja informacij o stilu spletnim dokumentom. [69]

LESS je dinamičen slogovni jezik, ki močno olajša pisanje CSS podlog. Je nadgradnja jezika CSS, kar pomeni, da je vsaka CSS podloga tudi LESS podloga. Jeziku CSS doda spremenljivke, gnezdenje, dedovanje, operacije in funkcije. [70]

8 DELOVANJE

Delovanje sistema je razdeljeno na tri faze:

- Zagon strežnika
- Registracija odjemalcev
- Izvajanje testov na odjemalcu in prikaz rezultatov

8.1 ZAGON STREŽNIKA

V prvi fazi je potrebno zagnati strežnik, ki skrbi za seznam odjemalcev in dostavo datotek, potrebnih za izvajanje testov. Strežnik je ponavadi na razvijalčevem računalniku, tako da lahko teste izvaja sproti, med spreminjanjem kode.

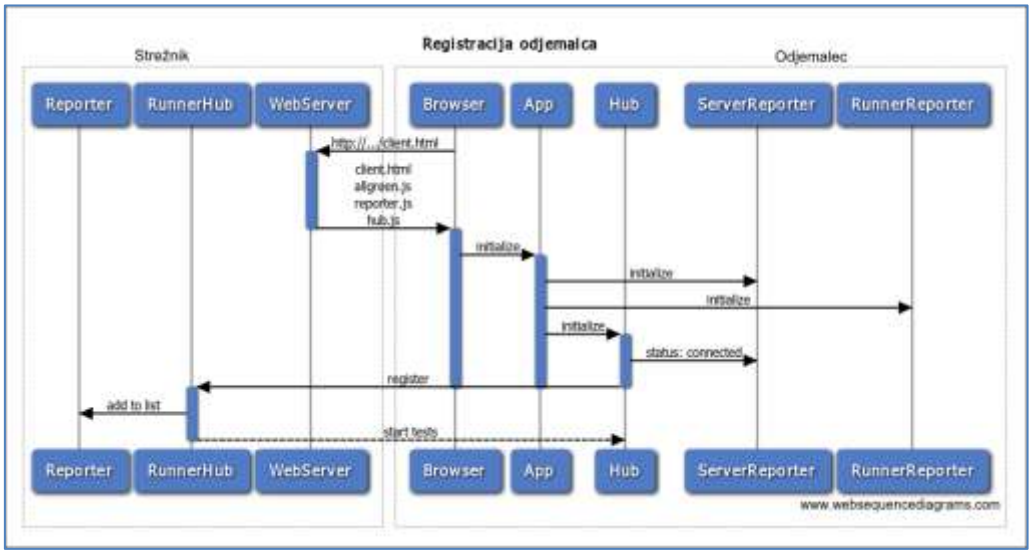
8.2 REGISTRACIJA ODJEMALCEV

Ko je strežnik zagnan, se lahko nanj registrirajo odjemalci (slika 11). Odjemalci so lahko tudi na razvijalčevem računalniku ali pa drugje, le dostop preko HTTP protokola morajo imeti omogočen.

Registracija se prične s strani odjemalca, ki zahteva privzeto datoteko (client.html). Strežnik mu poleg te datoteke dostavi še JavaScript datoteke, v katerih je odjemalčeva aplikacija. Ko odjemalec te datoteke dobi, se avtomatsko

zažene inicializacija aplikacije. Na koncu inicializacije aplikacije odjemalec pošlje zahtevo za registracijo na strežnik s pomočjo SignalR knjižnice. Odjemalec se na strežniku doda v seznam odjemalcev. Na koncu strežnik še pošlje zahtevo odjemalcu za zagon testov.

Odjemalec se s seznama odstrani, ko strežnik zazna odklop odjemalca. Nekateri brskalniki to sporočijo takoj, ko internetno stran zapremo, nekateri pa potrebujejo več časa.



Slika 11: Sekvenčni diagram registracije odjemalca

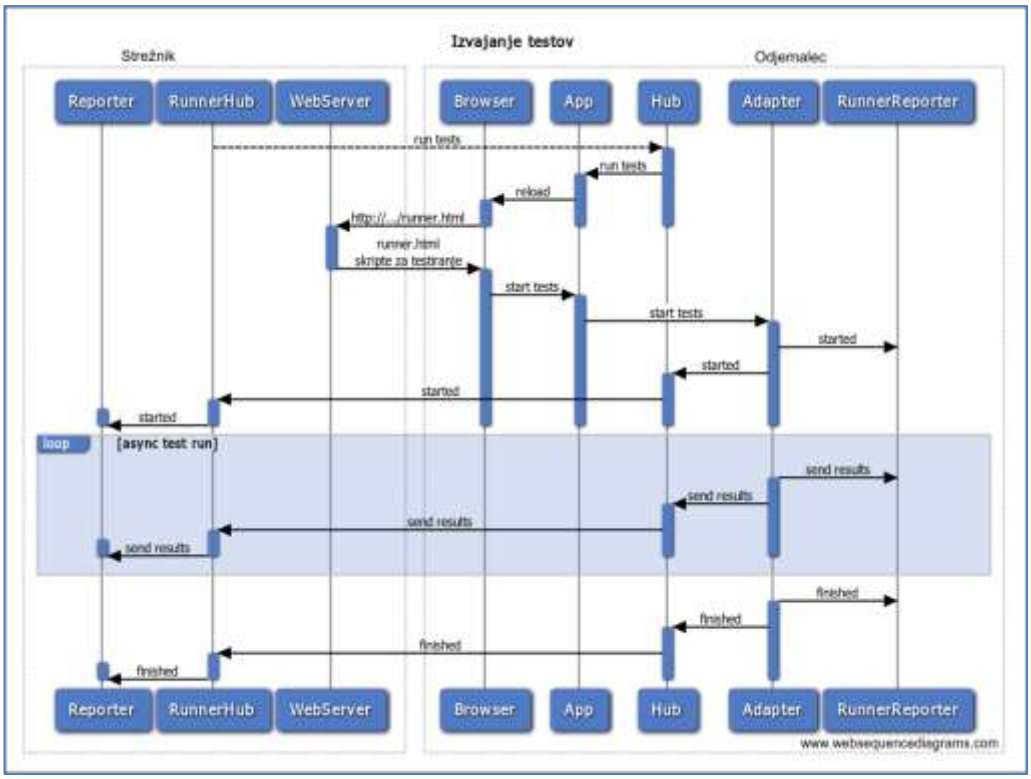
8.3 IZVAJANJE TESTOV NA ODJEMALCU

Registriran odjemalec lahko izvaja teste (slika 12). Zahteva za izvajanje testov vedno izvira iz strežnika. Kot prej omenjeno, pride do te zahteve na koncu registracije odjemalca. Vendar pa lahko zahtevo za zagon testov strežnik pošlje tudi iz drugih razlogov. V trenutni implementaciji so ti razlogi naslednji:

- Registracija odjemalca,
- Uporabnik pritisne gumb na grafičnem vmesniku,
- Opazovane datoteke so bile spremenjene.

Izvajanje testov se torej začne tako, da strežnik pošlje zahtevo odjemalcu. Odjemalec nato ponovno naloži dokument runner.html, ki se nahaja v IFRAME elementu v dokumentu client.html. Ker je v IFRAME elementu, jo je možno ponovno naložiti, ne da bi ponovno naložili celoten dokument. To pa je pomembno zato, ker bi s tem, ko bi ponovno naložili glavni dokument, povzročili ponovno registracijo odjemalca. Ker so vse datoteke za izvajanje testov v IFRAME elementu, nam ponovni zagon omogoča popolnoma čisto okolje za izvajanje testov brez vplivov predhodnega izvajanja.

Dokument runner.html vsebuje tudi povezave do vseh datotek, ki so potrebne za izvajanje testov. Katere so te datoteke, je del nastavitev strežnika. Ko se naložijo vse datoteke, se avtomatsko zažene izvajanje testov. Testi se izvajajo s pomočjo adapterja za testno ogrodje. Adapter sproti pošilja stanje in rezultat testov strežniku s pomočjo SignalR knjižnice. Hkrati pa rezultate testov tudi izpisuje, tako da so vidni v brskalniku.

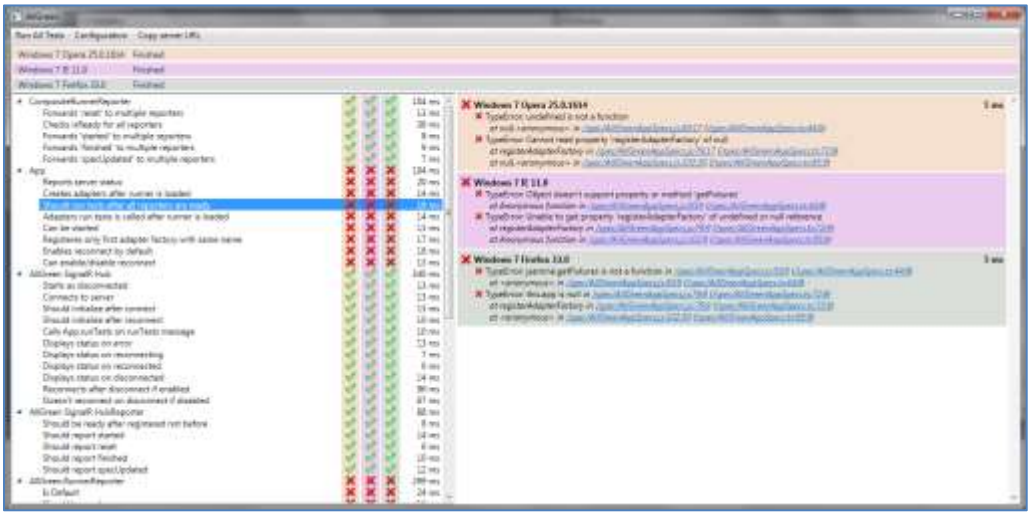


Slika 12: Sekvenčni diagram izvajanja testov

8.4 GRAFIČNI VMESNIK

Grafični vmesnik (slika 13) je .NET aplikacija na osnovi WPF tehnologije [71]. Razdeljena je na tri glavne dele:

- Seznam in stanje registriranih odjemalcev (zgoraj)
- Seznam in rezultat testov (levo)
- Podrobnosti izvajanja izbranega testa (desno)



Slika 13: Grafični vmesnik aplikacije AllGreen

8.4.1 SEZNAM IN STANJE REGISTRIRANIH ODJEMALCEV

Seznam registriranih odjemalcev je na vrhu grafičnega vmesnika. Aplikacija uporablja UAParser [72], da pretvori User Agent [73] niz v prikazan naziv ter tip brskalnika in operacijskega sistema. Poleg naziva se nahaja še trenutna operacija, ki se izvaja na odjemalcu. Vsak odjemalec je obarvan z naključno izbrano barvo, ki se uporablja za lažje povezovanje odjemalca z rezultatom testa na tem odjemalcu.

8.4.2 SEZNAM IN REZULTAT TESTOV

Na levi strani grafičnega vmesnika je seznam testov. Testi so razvrščeni v drevesni strukturi, kakršna je definirana v izvorni datoteki testov. Tako so testi

združeni v skupine. Poleg testa so rezultati izvajanja na posameznem odjemalcu. Rezultat je obarvan z enako barvo, kot je obarvan odjemalec v seznamu odjemalcev. Na koncu je še skupna dolžina izvajanja testa na vseh odjemalcih.

8.4.3 PODROBNOSTI IZVAJANJA IZBRANEGA TESTA

Ko test v seznamu izberemo, se na desni strani prikažejo podrobnosti izvajanja tega testa. Tudi tu so podrobnosti obarvane z enakimi barvami, kot so obarvani odjemalci. Podrobnosti izvajanja testa se bolj ali manj razlikujejo glede na to, na katerem odjemalcu je bil test izveden. Večinoma pa lahko v podrobnostih vidimo dolžino izvajanja testa, napako in sklad klicev (*angl. stack trace*), v katerem lahko najdemo lokacijo napake v izvorni kodi.

JavaScript nam pri lociranju napake povzroča manjše nevšečnosti. Večinoma zaradi prihranka pri prenosu JavaScript datotek uporabljamo t.i. pomanjševalnik datotek (*angl. minifier*) [74]. Ta nam zmanjša izvorno JavaScript datoteko s tem, da odstrani nepotrebne presledke, vso kodo strne v eno vrstico, zamenja imena spremenljivk z najkrajšimi možnimi imeni, itd. S tem ko prihranimo prostor, pa izgubimo preglednost kode. Še dodaten problem pa nastane z uporabo TypeScript prevajalnika. Tako imamo izvorno kodo, napisano v TypeScript jeziku, prevedeno v JavaScript jezik in na koncu to prevedeno v pomanjšano verzijo JavaScripta. Ko se zgodi napaka, pa nam brskalnik sporoči lokacijo napake v tej zadnji pomanjšani verziji našega programa.

Vsa ta orodja, ki spreminjajo JavaScript kodo, nam poleg pretvorjene kode izdelajo tudi povezovalne datoteke (*angl. JavaScript source map* [75]). V teh datotekah je seznam povezav med lokacijami v izvorni datoteki in lokacijami v pretvorjeni datoteki. Tako lahko lokacijo napake v pomanjšani datoteki pretvorimo v lokacijo napake v TypeScript datoteki.

Lokacije napak v izvorni in pretvorjeni datoteki so prikazane v grafičnem vmesniku. Lokacije so predstavljene kot povezave, s katerimi lahko odpremo datoteko v urejevalniku besedil, ki ga nastavimo v nastavitvah.

9 TESTIRANJE ORODJA ALLGREEN

Orodje AllGreen je bilo razvito z uporabo testno vodenega razvoja že od samega začetka. Tako je bilo napisanih veliko testov enot za izvorno kodo na strežniku in izvorno kodo na odjemalcu. Napisanih je bilo 142 testov za kodo na strežniku in 58 testov za kodo na odjemalcu.

9.1 TESTIRANJE IZVORNE KODE ZA STREŽNIK

Za strežnik je bila uporabljena .NET tehnologija in C# programski jezik. Zato je bila tudi za teste uporabljena ista tehnologija. Testi so bili napisani s pomočjo orodja Visual Studio Unit Testing Framework, ki je sestavni del Visual Studio okolja. Za bolj pregledno pisanje testov oz. preverjanje rezultatov v testih je bila uporabljena knjižnica Fluent Assertions. Za pisanje testnih dvojnikov je bila uporabljena knjižnica Moq.

9.1.1 VISUAL STUDIO UNIT TESTING FRAMEWORK

Visual Studio Unit Testing Framework [76] je nabor orodij, ki nam omogoča pisanje in izvajanje testov v okolju .NET. Je sestavni del okolja Visual Studio od verzije 2005 naprej. Ker se za izvajanje teh testov največkrat uporablja orodje za orodno vrstico MSTest, se ogrodje največkrat imenuje kar MSTest. V novejše verzije okolja Visual Studio (verzija 2012 in naprej) je bilo dodano novo orodje za

izvajanje testov v orodni vrstici imenovano VSTest.Console. Microsoft priporoča uporabo tega namesto MSTest orodja.

9.1.2 Fluent Assertions

Fluent Assertions [77] je nabor .NET razširitvenih metod, ki omogočajo bolj naravno pisanje pričakovanega izida testov. Omogočajo tudi enostavno prestrezanje in preverjanje dogodkov objektov.

Namesto da napišemo naslednji pričakovan izid:

```
Assert.AreEqual("ABC", result);
```

s pomočjo Fluent Assertions napišemo:

```
result.Should().Be("ABC");
```

Že na prvi pogled je bolj jasno, kaj pravzaprav testiramo.

Poleg tega je v primeru, da test ne uspe, tudi izpisan opis napake bolj natančen. V prvem primeru bo izpis naslednji:

```
Assert.AreEqual failed. Expected:<ABC>. Actual:<ABCDEF>.
```

V primeru Fluent Assertions pa bo izpis takšen:

```
Expected string to be "ABC" with a length of 3, but "ABCDEF" has
a length of 6.
```

Še večja uporabnost pa se izkaže pri preverjanju objektnih grafov in seznamov objektov, saj Fluent Assertions omogoča enostavno in fleksibilno preverjanje. Pri vgrajenem preverjanju pa bi se morali sami sprehoditi po objektnem grafu in preveriti, ali sta objekta in njune lastnosti pričakovane.

9.1.3 Moq

Moq [78] je .NET knjižnica za testne dvojnike, od začetka razvita tako, da popolnoma uporablja .NET Linq izrazna drevesa (*angl. expression trees*) in

lambda izraze [79]. Podpira testne dvojnike za vmesnike in razrede. Programski vmesnik je izredno enostaven.

9.2 TESTIRANJE IZVORNE KODE ZA ODJEMALEC

Ker je bila za odjemalec uporabljena tehnologija JavaScript, se tudi orodja za testiranje popolnoma razlikujejo od tistih za testiranje kode za strežnik. Za pisanje testov je bila uporabljena knjižnica Jasmine, z dodatkom Jasmine-jQuery. Jasmine knjižnica že vsebuje knjižnico za pisanje testnih dvojnikov, tako da ni bila uporabljena nobena druga knjižnica.

Za izvajanje testov sem uporabil Karma orodje. Med razvojem sem upal, da bom za izvajanje testov lahko uporabil kar orodje AllGreen, ko bo implementacija zadostna. Vendar pa je v okolju JavaScript izredno težko ločiti posamezne module. Tako bi v sistemu obstajala dva modula z enakim imenom: modul, ki ga testiramo, in modul, ki izvaja teste. Rešitev bi bila ta, da bi pred izvajanjem spremenil ime enega izmed njiju (torej bi orodje pred izvajanjem spremenilo izvorno kodo), vendar se to ime nahaja v več datotekah (datoteke se lahko celo dinamično nalagajo) in bi bila taka implementacija izredno zahtevna. Zato sem to idejo na koncu opustil.

10 SKLEPNE UGOTOVITVE

Cilj diplomskega dela je bil izdelati orodje, ki bi olajšalo testno voden razvoj JavaScript aplikacij. Osnovna naloga orodja naj bi bila izvajanje testov, vendar pa je bil cilj narediti orodje tudi prijazno uporabniku. Čeprav kar nekaj takih orodij že obstaja, vsem tem orodjem primanjkuje prav ta komponenta – prijaznost do uporabnika. Ker je uporabnik orodja programer, je orodje prijazno do uporabnika takrat, ko mu omogoča kar najhitrejše lociranje napake. Zato je bilo to vodilo glavno pri razvoju orodja AllGreen in upam, da je bil ta cilj dosežen.

Kljub temu pa je možnih še veliko izboljšav orodja AllGreen. Nekaj idej, ki so mi padle na pamet:

- Integracija v okolje Visual Studio,
- Dodati možnost avtomatskega zagona brskalnikov,
- Boljša integracija s CI orodji,
- Možnost izvajanja omejenega nabora testov, ki zajemajo le zadnje spremembe izvorne kode,
- Prikaz pokritosti izvorne kode s testi.

Za konec velja še omeniti, da sem pri razvoju uporabljal git [80] orodje za upravljanje z verzijami in GitHub [81] spletni portal za gostovanje git repozitorijev. Orodje AllGreen je odprtokodna aplikacija in je dosegljiva na naslovu https://github.com/gstamac/AllGreen.

LITERATURA

[1] K. Beck, Test-Driven Development by Example, Addison Wesley, 2002.

[2] K. Beck, Extreme Programming Explained: Embrace Change, Addison-Wesley, 1999.

[3] D. McCracken, Digital Computer Programming, John Wiley & Sons, Inc., 1957.

[4] E. W. Dijkstra, The Humble Programmer, ACM Turing Lecture, 1972.

[5] „Javascript," Wikipedia, 2014. [Elektronski]. Dostopno na: http://en.wikipedia.org/wiki/JavaScript.

[6] S. Suehring, JavaScript Step by Step, Third Edition, Sebastopol: O'Reilly Media, Inc., 2013.

[7] „ECMAScript," Wikipedia, 2014. [Elektronski]. Dostopno na: http://en.wikipedia.org/wiki/ECMAScript.

[8] S. O'Grady, „The RedMonk Programming Language Rankings: January 2013," 2013. [Elektronski]. Dostopno na: http://redmonk.com/sogrady/2013/02/28/language-rankings-1-13/.

[9] „Programming Trends," Dodgy Coder, 2014. [Elektronski]. Dostopno na: http://www.dodgycoder.net/p/programming-trends.html.

[10] E. R. Farrer, „Unit testing isn't enough. You need static typing too.," 2012. [Elektronski]. Dostopno na: http://evanfarrer.blogspot.com/2012/06/unit-

testing-isnt-enough-you-need.html.

[11] E. R. Farrer, „A Quantitative Analysis Of Whether A Quantitative Analysis Of Whether Checking For Error Detection,“ 2011.

[12] D. Arno, „Why JavaScript is a toy language,“ 2010. [Elektronski]. Dostopno na: http://www.davidarno.org/2010/05/18/why-javascript-is-a-toy-language/.

[13] D. B. i. K. Yank, „Google Closure: How not to write JavaScript,“ 2014. [Elektronski]. Dostopno na: http://www.sitepoint.com/google-closure-how-not-to-write-javascript/.

[14] Microsoft, „TypeScript Language Specification,“ 2014. [Elektronski]. Dostopno na: http://go.microsoft.com/fwlink/?LinkId=267121.

[15] NATO, „The Nato Software Engineering Conference,“ Garmisch, 1968.

[16] „DevExpress.com,“ 2014. [Elektronski]. Dostopno na: http://www.devexpress.com.

[17] G. Meszaros, xUnit Test Patterns: Refactoring Test Code, 2007.

[18] L. W. Boby George, „An initial investigation of test driven development in industry,“ 2003.

[19] „TDD,“ Wikipedia, 2014. [Elektronski]. Dostopno na: http://en.wikipedia.org/wiki/Test-driven_development.

[20] R. C. Martin, „Debuggers are a wasteful Timesink,“ 2003. [Elektronski]. Dostopno na: http://www.artima.com/weblogs/viewpost.jsp?thread=23476.

[21] M. M. M. T. Hakan Erdogmus, „On the Effectiveness of the Test-First Approach to Programming,“ *IEEE Transactions on Software Engineering,* 2005.

[22] J. Atwood, „Podcast #52,“ 2009. [Elektronski]. Dostopno na: http://blog.stackoverflow.com/2009/05/podcast-52/.

[23] R. C. Martin, 2009. [Elektronski]. Dostopno na: https://twitter.com/unclebobmartin/status/1548729270.

[24] „Emergent Design,“ Wikipedia, 2014. [Elektronski]. Dostopno na: http://en.wikipedia.org/wiki/Emergent_Design.

[25] F. P. Matthias M. Müller, „About the Return on Investment of Test-Driven Development,“ 2012.

[26] D. T. Andy Hunt, The Pragmatic Programmer, 1999.

[27] „Duplicate code,“ Wikipedia, 2014. [Elektronski]. Dostopno na: http://en.wikipedia.org/wiki/Duplicate_code.

[28] B. S. Baker, „A Program for Identifying Duplicated Code,“ 1992.

[29] M. O. R. Richard M. Karp, „ Efficient randomized pattern-matching algorithms,“ 1987.

[30] „Abstract syntax tree,“ Wikipedia, 2014. [Elektronski]. Dostopno na: http://en.wikipedia.org/wiki/Abstract_syntax_tree.

[31] S. D. Matthias Rieger, „Visual Detection of Duplicated Code“.

[32] „Zanos,“ Wikipedia, 2014. [Elektronski]. Dostopno na: http://sl.wikipedia.org/wiki/Zanos.

[33] A. Tornhill, Interviewee, *Psychology in Programming with Adam Tornhill.* [Intervju]. 2014.

[34] „Integration testing,“ Wikipedia, 2014. [Elektronski]. Dostopno na: http://en.wikipedia.org/wiki/Integration_testing.

[35] „What is component testing,“ Wikipedia, 2014. [Elektronski]. Dostopno na:

http://istqbexamcertification.com/what-is-component-testing/.

[36] „System testing,“ Wikipedia, 2014. [Elektronski]. Dostopno na: http://en.wikipedia.org/wiki/System_testing.

[37] „Acceptance testing,“ Wikipedia, 2014. [Elektronski]. Dostopno na: http://en.wikipedia.org/wiki/Acceptance_testing.

[38] D. North, „Introducing BDD,“ [Elektronski]. Dostopno na: http://dannorth.net/introducing-bdd/.

[39] „User story,“ Wikipedia, 2014. [Elektronski]. Dostopno na: http://en.wikipedia.org/wiki/User_story.

[40] G. Booch, Object-oriented Analysis and Design with Applications, 1993.

[41] B. W. i. P. N. P. Boehm, „Understanding and Controlling Software Costs,“ *IEEE Transactions on Software Engineering,* 1988.

[42] „Headless browser,“ Wikipedia, 2014. [Elektronski]. Dostopno na: http://en.wikipedia.org/wiki/Headless_browser.

[43] „CommonJS,“ Wikipedia, 2014. [Elektronski]. Dostopno na: http://en.wikipedia.org/wiki/CommonJS.

[44] „WebKit,“ Wikipedia, 2014. [Elektronski]. Dostopno na: http://en.wikipedia.org/wiki/WebKit.

[45] A. Hidayat, „PhantomJS,“ 2014. [Elektronski]. Dostopno na: http://phantomjs.org/.

[46] „Node.js,“ Joyent, Inc., 2014. [Elektronski]. Dostopno na: http://nodejs.org/.

[47] „Karma,“ [Elektronski]. Dostopno na: http://karma-runner.github.io.

[48] „Chutzpah,“ 2014. [Elektronski]. Dostopno na: https://github.com/mmanela/chutzpah.

[49] „TestSwarm,“ [Elektronski]. Dostopno na: https://github.com/jquery/testswarm/wiki.

[50] „BrowserStack,“ BrowserStack, 2014. [Elektronski]. Dostopno na: http://www.browserstack.com.

[51] „Jasmine-jQuery,“ [Elektronski]. Dostopno na: https://github.com/velesin/jasmine-jquery.

[52] J. Palermo, „Onion Architecture,“ 2008. [Elektronski]. Dostopno na: http://jeffreypalermo.com/blog/the-onion-architecture-part-1/.

[53] R. C. Martin, „Clean Architecture,“ 8th Light, Inc., 2014. [Elektronski]. Dostopno na: http://blog.8thlight.com/uncle-bob/2012/08/13/the-clean-architecture.html.

[54] A. Cockburn, „Hexagonal Architecture,“ 2008. [Elektronski]. Dostopno na: http://alistair.cockburn.us/Hexagonal+architecture.

[55] J. O. C. i. G. Bjørnvig, Lean Architecture: for Agile Software Development, Wiley, 2010.

[56] I. Jacobson, Object Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley Professional, 1992.

[57] „Multilayered Architecture,“ Wikipedia, 2014. [Elektronski]. Dostopno na: http://en.wikipedia.org/wiki/Multilayered_architecture.

[58] „Dependency inversion principle,“ Wikipedia, 2014. [Elektronski]. Dostopno na: http://en.wikipedia.org/wiki/Dependency_inversion_principle.

[59] „Dependency injection,“ Wikipedia, 2014. [Elektronski]. Dostopno na: http://en.wikipedia.org/wiki/Dependency_injection.

[60] „ASP.NET,“ Wikipedia, 2014. [Elektronski]. Dostopno na: http://en.wikipedia.org/wiki/ASP.NET.

[61] „MSTest,“ Wikipedia, [Elektronski]. Dostopno na: http://en.wikipedia.org/wiki/MSTest.

[62] „Open Web Interface for .NET,“ Wikipedia, 2014. [Elektronski]. Dostopno na: http://en.wikipedia.org/wiki/Open_Web_Interface_for_.NET.

[63] „ASP.NET SignalR,“ [Elektronski]. Dostopno na: http://signalr.net/.

[64] „Caliburn.Micro,“ [Elektronski]. Dostopno na: https://github.com/Caliburn-Micro/Caliburn.Micro.

[65] „XAML Overview (WPF),“ Microsoft, 2014. [Elektronski]. Dostopno na: https://msdn.microsoft.com/en-us/library/ms752059%28v=vs.110%29.aspx.

[66] „Model View ViewModel,“ Wikipedia, 2014. [Elektronski]. Dostopno na: http://en.wikipedia.org/wiki/Model_View_ViewModel.

[67] S. Robbins, „TinyIoC,“ [Elektronski]. Dostopno na: https://github.com/grumpydev/TinyIoC.

[68] „jQuery,“ Wikipedia, 2014. [Elektronski]. Dostopno na: http://en.wikipedia.org/wiki/JQuery.

[69] „CSS,“ Wikipedia, 2014. [Elektronski]. Dostopno na: http://sl.wikipedia.org/wiki/CSS.

[70] „Less,“ Wikipedia, 2014. [Elektronski]. Dostopno na: http://en.wikipedia.org/wiki/Less_%28stylesheet_language%29.

[71] „Windows Presentation Foundation,“ Microsoft, 2015. [Elektronski]. Dostopno na: https://msdn.microsoft.com/en-us/library/ms754130%28v=vs.110%29.aspx.

[72] T. Langel, „UA Parser,“ 2014. [Elektronski]. Dostopno na: https://github.com/tobie/ua-parser.

[73] „User Agent,“ Wikipedia, 2014. [Elektronski]. Dostopno na: http://en.wikipedia.org/wiki/User_agent.

[74] „Minification,“ Wikipedia, 2014. [Elektronski]. Dostopno na: http://en.wikipedia.org/wiki/Minification_%28programming%29.

[75] R. Seddon, „Introduction to JavaScript Source Maps,“ 2012. [Elektronski]. Dostopno na: http://www.html5rocks.com/en/tutorials/developertools/sourcemaps/.

[76] „Visual Studio Unit Testing Framework,“ Wikipedia, 2014. [Elektronski]. Dostopno na: http://en.wikipedia.org/wiki/Visual_Studio_Unit_Testing_Framework.

[77] D. Doomen, „Fluent Assertions,“ [Elektronski]. Dostopno na: http://www.fluentassertions.com/.

[78] „Moq,“ [Elektronski]. Dostopno na: https://github.com/Moq/moq4.

[79] „LINQ,“ Wikipedia, 2014. [Elektronski]. Dostopno na: http://en.wikipedia.org/wiki/Language_Integrated_Query.

[80] „Git,“ [Elektronski]. Dostopno na: http://www.git-scm.com/.

[81] „GitHub,“ [Elektronski]. Dostopno na: https://github.com/.

[82] Y. G. Y. Yuan, „Count Matrix Based Code Clone Detection,“ 2011.