Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Matic Jurglič

# Testiranje programske opreme z uporabo ogrodja Ruby on Rails

DIPLOMSKA NALOGA
NA UNIVERZITETNEM ŠTUDIJU

doc. dr. Damjan Vavpotič
MENTOR

Ljubljana, 2014

# IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Spodaj podpisani izjavljam, da sem avtor dela, da slednje ne vsebuje materiala, ki bi ga kdorkoli predhodno že objavil ali oddal v obravnavo za pridobitev naziva na univerzi ali drugem visokošolskem zavodu, razen v primerih kjer so navedeni viri.
S svojim podpisom zagotavljam, da:

- sem delo izdelal samostojno pod mentorstvom Damjana Vavpotiča,
- so elektronska oblika dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko in
- soglašam z javno objavo elektronske oblike dela v zbirki "Dela FRI".

— Matic Jurglič, Ljubljana, julij 2014.

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Matic Jurglič

# Testiranje programske opreme z uporabo ogrodja Ruby on Rails

## POVZETEK

V svetu modernih spletnih aplikacij in odprtokodnih tehnologijah je čedalje pogosteje moč opaziti uporabo testno usmerjenega razvoja programske opreme, pri katerem razvijalci pišejo teste za izvorno kodo. Glavna prednost investiranja v primerno napisane teste so lažje odkrivanje napak, učinkovitejši razvojni proces in posledično višja stopnja kvalitete produkta.

Diplomsko delo opisuje splošne tehnike testiranja in se osredotoča na uporabo v ogrodju Ruby on Rails, za katerim stoji kultura odprtokodne skupnosti, ki močno poudarja pomembnost testiranja programske opreme. Predstavljena je vpeljava testno usmerjenega razvoja v projektu iz resničnega življenja, razložen je postopek avtomatizacije testov in njihov pomen pri vpeljavi zvezne integracije.

S primeri iz realnih situacij je prikazano, kako se je vložek v testiranje obrestoval in preprečil marsikatero napako, s katero bi v produkcijskem okolju lahko nastala poslovna škoda in izguba uporabnikov aplikacije.

**Kljucne besede:** testiranje programske opreme, Ruby on Rails, test enote, integracijski test, TDD, BDD, graditev, avtomatizacija, zvezna integracija

University of Ljubljana
Faculty of Computer and Information Science

Matic Jurglič

**Software Testing using Ruby on Rails framework**

# ABSTRACT

In the world of modern web applications and open source technologies there is a rising in popularity of using test driven development methodologies in software development. The main advantages of writing tests are easier error discovery, more effective development process, and consequently higher product quality.

This thesis describes common testing techniques and focuses on usage in Ruby on Rails framework, which has a vibrant open source community with a culture that strongly emphasizes the importance of testing. Thesis describes the introduction of test driven development into a real life project, explains the test automation procedure and how it associates with continuous integration.

With real life scenario cases we demonstrated how writing tests proved to be worthwhile and how testing prevented multiple errors which might cause business damage and loss of users in production environment.

**Key words:** software testing, Ruby on Rails, unit test, integration test, TDD, BDD, build, automatization, continuous integration

# ZAHVALA

*Zahvaljujem se družini za podporo in prijateljem za vzpodbudo.*

— Matic Jurglič, Ljubljana, julij 2014.

# KAZALO

**Povzetek** i

**Abstract** iii

**Zahvala** v

**1 Uvod** 1

**2 Testiranje programske opreme** 3

2.1 Cilji testiranja . . . 3

2.2 Prednosti testiranja v tehnologiji Ruby On Rails . . . 4

2.2.1 Načelo minimalne konfiguracije . . . 4

2.2.2 Ločeno testno okolje . . . 4

2.2.3 Avtomatsko generiranje začetnih testov . . . 4

2.3 Testno usmerjen razvoj (TDD) . . . 6

2.3.1 Prednosti . . . 7

2.3.2 Slabosti . . . 7

2.4 Vedenjsko usmerjeni razvoj (BDD) . . . 7

2.4.1 Primerjava s TDD . . . 7

2.4.2 Prednosti . . . 8

2.4.3 Slabosti . . . 8

2.4.4 Primer BDD v Rails . . . 8

2.5 Pogoste vrste testov . . . 9

2.5.1 Testi enot . . . 11

2.5.2 Integracijski testi . . . 11

2.5.3 Sistemski testi . . . 11

2.5.4 Sprejemni testi . . . 12
2.5.5 Funkcijski testi . . . 12
2.5.6 Regresijski testi . . . 13
2.5.7 Testi grafičnega uporabniškega vmesnika . . . 13
2.6 Metrike testne pokritosti . . . 13

**3 Orodja za testiranje v tehnologiji Ruby on Rails 15**
3.1 MiniTest . . . 15
3.2 RSpec . . . 16
3.3 Capybara . . . 17
3.3.1 JavaScript gonilniki . . . 18
3.3.2 FactoryGirl . . . 18
3.4 Cucumber . . . 20
3.4.1 Značilnost . . . 20
3.4.2 Scenarij . . . 20
3.4.3 Primer značilnosti . . . 21

**4 Vloga testov pri zvezni integraciji in zvezni postavitvi 23**
4.1 Pomen . . . 23
4.2 Koraki in odgovornosti . . . 24
4.3 Zvezna postavitev . . . 24
4.4 Rešitve za Rails aplikacije . . . 25

**5 Uvedba testno usmerjenega razvoja v realnem projektu 27**
5.1 Testno usmerjen razvoj sistema za procesiranje . . . 27
5.2 Testno usmerjen razvoj uporabniške spletne aplikacije . . . 32
5.2.1 Testi modelov . . . 32
5.2.2 Testi lastnosti . . . 35
5.2.3 Vedenjsko usmerjen razvoj . . . 37
5.2.4 Zvezna integracija in postavitev . . . 41
5.2.5 Uporabljene storitve . . . 41
5.2.6 Avtomatska integracija aplikacije . . . 42

**6 Metrike testne pokritosti in scenariji v realnem projektu** **45**

6.1 Testna pokritost . . . . . 46

6.1.1 Rezultati . . . . . 46

6.1.2 Komentar . . . . . 46

6.2 Prikaz koristi testiranja v realnih scenarijih . . . . . 47

6.2.1 Refaktoriranje . . . . . 47

6.2.2 Pomankljivo napisan test . . . . . 48

6.2.3 Sprememba konfiguracije . . . . . 52

**7 Zaključek** **57**

**Literatura** **59**

# 1 Uvod

Že kar nekaj časa je v svetu programske opreme moč opaziti vzpon agilnih metodologij, kjer je testiranje oz. pisanje testov eden izmed ključnih temeljev učinkovitega razvoja aplikacij. Testiranje programske opreme je proces izvrševanja določenega procesa ali sistema z namenom iskanja napak. Večina modernih razvijalcev se poslužuje pisanja testov za programsko kodo, ki jo sproducira in se strinja z idejo, da testi prinašajo veliko korist, še posebno pri razvoju spletnih aplikacij. To je moč opaziti v raznih odprtokodnih projektih (npr. vtičniki za Ruby on Rails, imenovani *gems*), kjer avtorji pogosto dodajo nabor testov oz. je to celo zaželjeno in potrjuje kredibilnost ter stopnjo zaupanja v projekt.

Kljub temu pa imajo različni razvijalci programske opreme na različnih projektih in tehnologijah tudi različne poglede na pisanje testov. Nekateri so pristaši doseganja čim višje stopnje testne pokritosti, drugi pa se zadovoljijo zgolj s testi za najbolj ključne funkcionalnosti ali pa jih sploh ne pišejo.

Včasih se je potrebno vprašati o smiselnosti pisanja testov, saj je le-to odvisno od tipa in obsežnosti projekta, števila programerjev, njihove razvojne kulture in izkušenosti

itd. Če gre recimo za izdelavo prototipa, ki mora biti narejen v najkrajšem možnem času, ali pa za (znanstveno) eksperimentalni projekt, kjer smer nadaljnega razvoja ni povsem jasna, potem se ukvarjati s testi ni preveč smiselno, saj se specifikacije hitro spreminjajo in napisani testi hitro postanejo neuporabni. Po drugi strani pa imamo lahko jasno določen projekt, kjer se v večini primerov izkaže, da se investicija v pisanje testov nedvomno splača.

Razvili smo spletno aplikacijo v obliki storitve (*ang. SaaS - Software As A Service*), uporabili pa smo tehnologijo Ruby on Rails (skrajšano: Rails [1] ). Odprtokodno ogrodje Rails je povzročilo pravo malo revolucijo na področju testno usmerjenega razvoja, saj s svojimi orodji omogoča izredno enostavno pisanje testov. Zato smo se odločili, da pri razvoju aplikacije uporabimo testno usmerjeni razvoj, ki se je izkazal za pravilno odločitev, saj nam je med razvojem odkril številne nepravilnosti in nam posledično omogočil, da preprečimo številne napake, ki bi se lahko zgodile pri razvoju različnih funkcionalnosti aplikacije.

V diplomskem delu so opisana osnovna načela in koncepti testiranja programske opreme, kako se ti koncepti v praksi uporabljajo v ogrodju Rails in primeri uporabe v svoji spletni aplikaciji. Opisano je tudi, kako se testni nabor avtomatizira in izvede vsakič, ko dodamo kodo v skupni repozitorij - zvezna integracija in zvezna postavitev. Prikazani in analizirani so tudi konkretni primeri, ko nam je avtomatizirano testiranje odkrilo napake v aplikaciji, zaradi katerih bi imeli težave v produkcijskem okolju.

# 2 Testiranje programske opreme

## 2.1 Cilji testiranja

Glavni cilj testiranja programske opreme je doseči določeno stopnjo potrditve, da programska oprema deluje na način, kot je definirano v specifikaciji. Ta cilj na splošno ne pomeni, da s testiranjem odkrijemo čisto vsako možno napako, ampak da pravočasno odkrijemo situacije, ki bi lahko imele negativen učinek na uporabnost aplikacije, vzdrževanje programske opreme in na končnega uporabnika.

Med najbolj pogoste cilje testiranja programske opreme štejemo:

- Odkrivanje in preprečevanje napak
- Visoka stopnja kvalitete produkta
- Zadovoljstvo uporabnika
- Učinkovit razvojni proces
- Minimiziranje stroška vzdrževanja

Testiranje programske opreme nikoli ne more identificirati vseh napak v aplikaciji. Namesto tega poskuša stanje programske opreme primerjati s stanji ali obnašanji, katere smatramo za pravilne - to so lahko specifikacije, mehanizmi, sorodne aplikacije, prejšnje verzije istega produkta, logična pričakovanja, uporabniška pričakovanja, standardi ali drugi kriteriji.

## 2.2 Prednosti testiranja v tehnologiji Ruby On Rails

Ruby on Rails je uveljavljeno in preverjeno moderno odprtokodno ogrodje za izdelavo spletnih aplikacij, ki temelji na programskem jeziku Ruby. Vključuje celoten sklad za izdelavo aplikacij, ki komunicirajo s strežnikom in podatkovno bazo.

Ogrodje Rails je bilo že od vsega začetka zgrajeno v duhu pomembnosti testiranja. Za ogrodjem stoji močna odprtokodna skupnost, ki vzdržuje množico orodij, ki omogočajo učinkovito pisanje raznovrstnih testov.

### 2.2.1 Načelo minimalne konfiguracije

Rails močno sledi paradigmi razvoja programske opreme imenovani *Convention over Configuration*. Omogoča hiter razvoj ali prototipiranje aplikacije, kjer je potreba po različnih konfiguracijskih datotekah in pravilih minimizirana. Pri tem je potrebno spoznati pravila pomenovanja datotek, razredov in direktorijev v ogrodju Rails, vendar pa lahko za posebne primere definiramo tudi svoja pravila.

### 2.2.2 Ločeno testno okolje

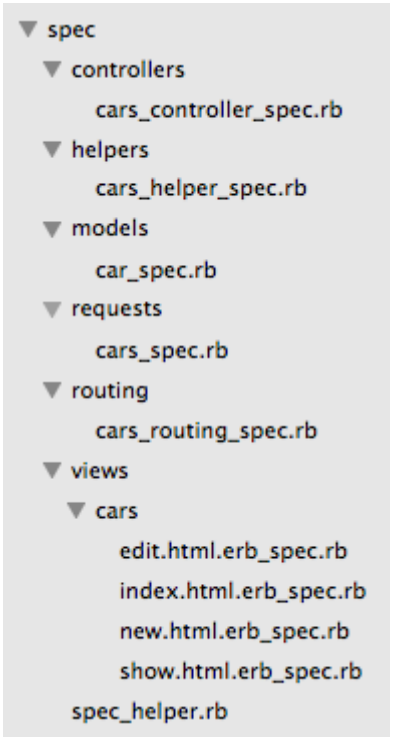
Privzeto ima vsaka Rails aplikacija tri okolja: razvojno, testno in produkcijsko. Za vsakega definiramo svojo podatkovno bazo. Podatkovna baza, v celoti posvečena testiranju, nam omogoča vzpostavitev in interakcijo s testnimi podatki v izolaciji. To pomeni, da se naši testi ne dotikajo podatkov v razvojnem ali produkcijskem okolju.

### 2.2.3 Avtomatsko generiranje začetnih testov

Rails ponuja priročne ukaze, s katerimi si olajšamo in pohitrimo tipična programerska dela, kot so avtomatično generiranje ogrodja kontrolerjev, modelov in pogledov (ang: *scaffolding*). Če uporabimo omenjene ukaze, nam Rails z orodjem *RSpec* avtomatsko generira tudi pripadajoče prazne teste, ki spadajo zraven.

Za primer avtomatsko generirajmo vse potrebne datoteke za pregled, dodajanje in urejanje avtomobilov z lastnostmi izdelovalec, model, končna hitrost:

```
rails generate scaffold Car manufacturer:string model:string top_speed:integer
```




**Slika 2.1** Prikaz strukture testnih datotek, ki nam jih generira Rails

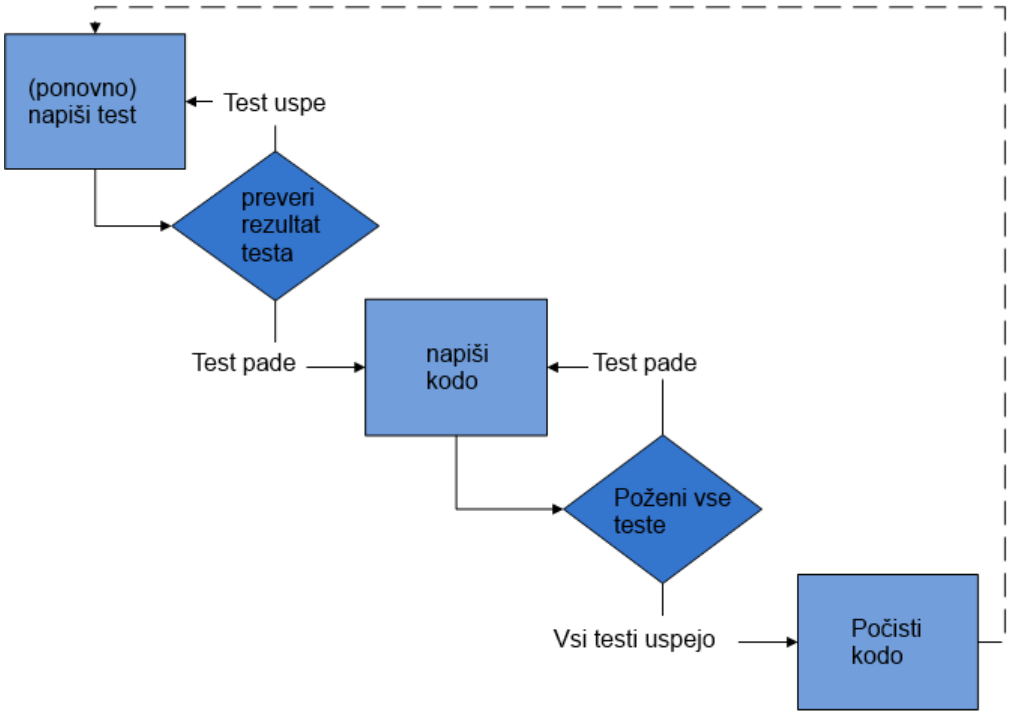
Ukaz kreira naslednje:

- Kontroler, model in pripadajoče poglede za vsako izmed akcij (index, show, new, edit)
- Migracijo, ki sproži kreiranje tabele *cars* (ponazarja enega izmed primerov konvencije: ime tabele v podatkovni bazi je množinska oblika imena modela)
- Kreira URL poti (ang. route)

- Za vsako izmed zgornjih stvari generira prazna ogrodja testov, ki se avtomatsko dodajo v testni nabor in jih dopolnemo v skladu z uporabniško specifikacijo

## 2.3 Testno usmerjen razvoj (TDD)

Testno usmerjen razvoj (ang. kratica TDD - *Test Driven Development*) je evolucijski proces razvoja programske opreme, ki temelji na ponavljanju zelo kratkega razvojnega cikla: najprej razvijalec napiše (sprva padajoč) test, ki definira izboljšavo ali novo funkcionalnost aplikacije, nato pa napiše minimalno količino kode, ki povzroči, da omenjeni test uspe. Nato kodo refaktorira, da postane v skladu s standardi [3].




**Slika 2.2** TDD diagram

Kaj je primarni cilj testno usmerjenega razvoja? Po eni strani je cilj tovrstnega razvoja specifikacija in ne validacija. Drugače povedano, TDD je način razmišljanja o funkcionalnostih in arhitekturi še preden napišemo funkcionalno kodo (kar pomeni, da je TDD zelo pomemben za agilne metodologije testnega razvoja). Po drugi strani pa je TDD programerska tehnika in njen cilj je napisati lepo kodo, ki deluje [4].

### 2.3.1 Prednosti

Z uporabo TDD je potreba po razhroščevanju programske kode praviloma redka. TDD zviša produktivnost, saj se ekipe osredotočajo na kratke testne iteracije. Uporaba kode, preden je dejansko napisana, nam pomaga ustvariti enostavnejše vmesnike in boljšo strukturo, kar vodi do višje stopnje produktivnosti skozi čas.

Testno usmerjen razvoj nam zagotovi dodaten sloj varnosti. Če teste pišemo hkrati ali pa šele za funkcionalno kodo, ki jo želimo testirati, tvegamo, da testiramo napačno stvar. Če sprva napišemo test, potem napisana koda ne more vplivati nanj in je manj možnosti, da nehote napišemo test, ki uspe ob sicer napačnem rezultatu.

### 2.3.2 Slabosti

Za razvijalce, ki jim TDD ni poznan, je ponavadi potrebno nekaj časa, da se ga ustrezno navadijo. Na začetku je ponavadi moč opaziti upočasnjen razvoj zaradi usmerjanja truda v pisanje testov. Potrebna je predvsem kulturna sprememba na stopnjah posameznih razvijalcev, saj TDD vpliva na pristop, pisanje in oblikovanje arhitekture programske opreme.

Vzdrževanje TDD načina razvoja je včasih naporno, še posebno pri situacijah, kjer do končnega roka ne ostane več veliko časa, zato mora za načeli TDD stati celotna ekipa ljudi, vključenih v projekt.

## 2.4 Vedenjsko usmerjeni razvoj (BDD)

Vedenjsko usmerjeni razvoj (ang. kratica BDD - *Behaviour Driven Development*) je izpopolnitev praks izhajajočih iz TDD. BDD je podmnožica TDD, vendar z malo drugačno miselnostjo: cilj je testiranje sistema, namesto določenega kosa kode. Razlika je subtilna, učinek pa velik.

### 2.4.1 Primerjava s TDD

Poglejmo si primer poimenovanja istega testa, prvič na TDD način, drugič na BDD način:

1. "Ko se kliče post akcija na blog kontrolerju, le-ta shrani objavo in jo preda naslovni strani v pogled, kjer se kliče index akcija".

2. "Ko na blog dodam objavo, se le-ta pokaže na naslovni strani."

Oba stavka opisujeta isto kodo, vendar BDD stil izraža nek določen namen in ni vezan na implementacijo, medtem ko TDD opisuje, kaj počne koda. Glavna korist BDD je nedvomno prej omenjena nevezanost na implementacijo. Testi postanejo manj krhki in refaktoriranje lažje [5].

### 2.4.2 Prednosti

BDD je narejen z namenom eliminirati težave, ki jih lahko povzroči TDD. Če pogledamo zgornja opisa obeh testov, lahko opazimo, da je razlika zgolj v besedah. BDD uporablja bolj ekspresiven način - tako, da se bere kot stavek. BDD testi so bolj fokusirani na lastnosti, in ne na dejanske rezultate.

Zmožnost branja testa kot naravni stavek povzroči kognitivni preskok v našem razmišljanju. Če lahko tekoče beremo naše teste, jih bomo naravno tudi bolje napisali (bolj celovito in razumljivo) [6].

BDD pomeni tudi obliko komunikacije med naročnikom projekta in njegovimi izvajalci (razvijalci). Dobro definirane uporabniške zgodbe v specifikaciji se lahko dobesedno prevedejo v definicije korakov BDD razvoja.

### 2.4.3 Slabosti

Za preproste aplikacije z malo poslovne logike je BDD razvoj običajno precej nepotreben oz. ne prinaša dovolj koristi, da bi lahko upravičili njegovo uporabo.

V primerih, ko specifikacijo piše naročnik, se lahko zgodi, da ne zna dovolj dobro napisati BDD uporabniških zgodb in jih morajo razvijalci dopolnjevati in popravljati. Še ena slabost je lahko ta, da je včasih naročnika težko prepričati, da vsakodnevno prek BDD sodeluje z razvijalci (preverja različne specifikacije, uporabniške zgodbe,...).

### 2.4.4 Primer BDD v Rails

Primer BDD testa v okolju Rails z orodjem *Cucumber* [7]:

```
Feature: Addition
  In order to get the result
  As a cashier
  I want to see the sum of two numbers
```

```
  Scenario: Add two numbers
    Given I have entered 50 into the calculator
    And I have entered 70 into the calculator
    When I press add
    Then the result should be 120 on the screen
```

Ko imamo enkrat test napisan, definiramo še definicije korakov, ki se s testom povežejo prek regularnih izrazov:

```
Given /I have entered (\d+) into the calculator/ do |n|
  @calc.push n.to_i
end

When /I press (\w+)/ do |op|
  @result = @calc.send op
end

Then /the result should be (.*) on the screen/ do |result|
  @result.should == result.to_f
end
```

## 2.5 Pogoste vrste testov

V realnosti obstaja v svetu razvoja programske opreme ogromno (več kot 100) vrst testov in načinov testiranja. V tej diplomski nalogi se bomo osredotočili na tiste najbolj najbolj pogoste, ki jih večinoma lahko uporabimo pri skoraj vsakem tipu aplikacije.

Zakaj obstaja toliko tipov testiranja programske opreme? Kvaliteta programske opreme se meri v obliki šestih faktorjev kvalitete (funkcionalnost, zanesljivost, učinkovitost, uporabnost, vzdrževalnost in prenosljivost). Vsaka izmed obstoječih vrst testiranja je narejena z namenom preizkušanja programske opreme v okviru prej naštetih faktorjev kvalitete. S hitrim višanjem kompleksnosti aplikacij (ogrodja, programski jeziki, čedalje več uporabnikov interneta, nove platforme, tehnologije) je zraslo tudi število vrst testiranja. Skladno z večjim številom tipa testiranj je rasla tudi potreba po več orodjih za

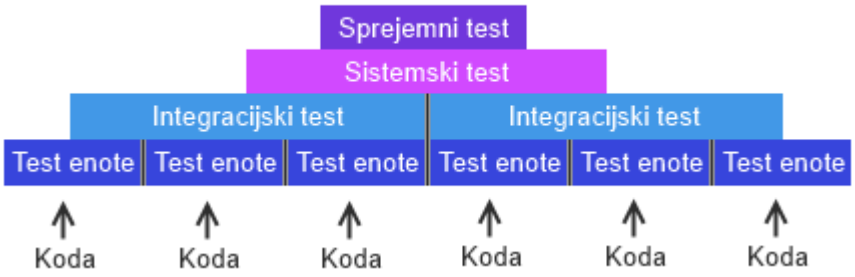
testiranje.

Na najvišjem nivoju lahko teste ločimo na dva dela, in sicer *White-Box* ter *Black-Box* metodi testiranja (princip bele in črne škatle). Prva metoda testira notranjo strukturo aplikacije, testni primeri so narejeni glede na notranjo sestavo sistema in avtor testov mora imeti globoko razumevanje o izvorni kodi. Nasprotna metoda pa je princip črne škatle, ki testira funkcionalnost programske opreme zgolj preko vhodnih in izstopnih točk sistema kot enote.


**Slika 2.3** Princip črne škatle

Pri Black-Box testiranju specifično znanje o kodi/interni strukturi ponavadi ni potrebno. Izvajalec testiranja se zaveda, *kaj* naj bi aplikacija počela, ampak ne ve, *kako* to počne. Pod to metodo testiranja štejemo tudi ročno testiranje, kjer tester prevzame vlogo končnega uporabnika in testira aplikacijo brez uporabe skript in orodij. Pri tem se poslužujejo testnih načrtov, primerov in testnih scenarijev.

Najbolj pogosto uporabljani testi so **testi enot, integracijski, sistemski in sprejemni testi**. V nadaljevanju so poleg še nekaterih drugih vrst testov opisane njihove lastnosti in razmerja.




**Slika 2.4** Diagram razmerij najbolj pogostih testov programske opreme

### 2.5.1 Testi enot

Test enote (ang. *Unit Test*) je najbolj osnoven test, ki ga lahko napiše programer, testiranje enot pa je metoda, s katero testiramo individualne dele kode, modulov in procedur. Za omenjeno enoto pa lahko smatramo najmanjši del aplikacije, ki ga lahko testiramo. Pisanje tovrstnih testov zahteva podrobno znanje o notranjem delovanju kode.

V proceduralnem programiranju enota lahko predstavlja celotni modul, medtem ko je v objektno orientiranem programiranju enota ponavadi kar celoten vmesnik, razred, lahko pa je tudi posamezna metoda [2].

Ponavadi se te enote povezujejo, kličejo med seboj in so odvisne druga od druge. Testiranje pravilne povezave med enotami ni naloga testov enot, zato lahko v teh primerih uporabimo preproste objekte ali nastavke (ang. *fakes, doubles, stubs, mocks,...)*, ki oponašajo nek drug objekt. Metodam teh objektov lahko nastavimo fiksen rezultat z namenom kontroliranih vhodov v enote, ki jih testiramo. V Rails je tovrstno testiranje še posebno enostavno z vtičnikom *RSpec*, kar je demonstrirano v praktičnem delu diplome.

### 2.5.2 Integracijski testi

Integracijski testi so včasih poimenovani tudi kot testi komponent. Integracijsko testiranje je faza v testiranju programske opreme, kjer se kombinirajo različne komponente in moduli aplikacije ter se testirajo kot skupina. To testiranje logično sledi testiranju enot. Moduli, ki so bili testirani v fazi testiranja enot, se sestavijo v večji agregat, nad katerim izvedemo test.

Namen integracijskih testov je potrditi funkcionalne, učinkovne in zanesljivostne specifikacijske potrebe skupin enot. Te skupine so testirane prek njihovih vhodov (princip črne škatle).

V Rails se integracijsko testiranje večinoma uporablja za testiranje interakcije med poljubnim številom kontrolerjev. Tovrstne teste uporabimo za testiranje delovnih tokov v aplikacij (primer: "prijavi se kot prodajalec in preveri zalogo artiklov").

### 2.5.3 Sistemski testi

Sistemsko testiranje je prva stopnja testiranja, kjer se sistem s kombiniranjem združenih delov aplikacije testira kot celota. Namen sistemskega testiranja je analizirati vedenje celega sistema in preveriti, ali ustreza uporabniškim specifikacijam.

Pri klasičnem razvoju programske opreme so sistemski testi zadnja faza testiranja, ki jo izvede ekipa za razvoj programske opreme. Sledijo jim sprejemni testi, kjer imajo glavno vlogo naročnik in uporabniško osebje [8]. Pri agilnih metodologijah razvoja pa je meja med sistemskim ter sprejemnim testom zabrisana in sistemske teste redko omenjamo. Njihov pomen se prenese in združi skupaj s sprejemnimi testi, kjer velja princip testiranja uporabniških zgodb.

### 2.5.4 Sprejemni testi

Tudi sprejemne teste lahko ločimo na dva tipa. Klasični sprejemni testi nastopijo po sistemskem testiranju in vključujejo test celega sistema (recimo uporaba spletne strani preko spletnega brskalnika), kjer naročnik in končni uporabniki preverjajo, ali funkcionalnost aplikacije izpolnjuje specifikacijo.

Pri agilnem razvoju programske opreme (kjer se ogrodje Rails tudi največkrat uporablja), pa sprejemni testi običajno pomenijo testno implementacijo uporabniških zgodb, ustvarjenih med razvojem s komunikacijo z naročnikom. Če test upe, pomeni, da programska oprema ustreza potrebam naročnika in uporabniško zgodbo lahko smatramo kot končano. Sprejemni testi so v bistvu izvršljiva specifikacija, napisana v domensko specifičnem jeziku v izbranem programskem jeziku.

### 2.5.5 Funkcijski testi

Funkcionalno testiranje je proces zagotavljanja kakovosti in je vrsta *testiranja črne škatle* [1], katerega testni primeri temeljijo na specifikaciji komponente, ki je predmet testiranja. Funkcije se testirajo tako, da jim pošljemo vhodne parametre in preverimo njihove izhodne parametre.

V Rails s funkcijskimi testi testiramo individualne kontrolerje. Kontrolerji prevzamejo prihajajoče spletne zahteve in se nanje odzovejo z generiranim pogledom.

V funkcijski test v ogrodju Rails ponavadi vključimo teste sledečih stvari:

- je bila spletna zahteva uspešna?
- je bil uporabnik preusmerjen na pravilno stran?
- je bil uporabnik uspešno prijavljen?

[1] Testiranje programske opreme z različnimi parametri na skupnem vhodu in validiranje rezultatov s pričakovanim izhodom

- je bil v odzivu uporabljen pravi objekt?
- je bilo uporabniku prikazano pravo sporočilo?

### 2.5.6 Regresijski testi

Regresijsko testiranje je tip testiranja programske opreme, katerega namen je odkriti nove napake (ali *regresije*) po tem, ko je bil spremenjen obstoječ del sistema, npr. dodelave, prilagoditve, spremembe konfiguracije,...

Namen tega je zagotoviti, da omenjene spremembe ne prinesejo novih napak. Glavni cilj je ugotoviti, ali sprememba v enem delu vpliva na druge dele sistema. Navadno metode regresijskega testiranja pomenijo ponovno poganjanje nabora testov in preverjanje, ali se je programsko vedenje spremenilo ali pa so se pojavile napake [9].

Tudi pri Rails aplikacijah je regresijski test mišljen predvsem proces, ko implementiramo kaj novega ali naredimo spremembo ter poženemo množico testov, z namenom odkrivanja napak. Testi enot so avtomatsko tudi regresijski testi in to je ena izmed njihovih največjih prednosti.

### 2.5.7 Testi grafičnega uporabniškega vmesnika

Večina programske opreme ima grafični vmesnik, ki uporabniku ponuja razne izbire, poglede in akcije. Za človeškega testerja je praktično nemogoče, da vedno znova pokrije vse testne primere z isto stopnjo natančnosti. Zato se uporabljajo avtomatski testi grafičnega uporabniškega vmesnika in tudi to je namenjeno zagotavljanju izpolnitve specifikacije. Testi testirajo, kako se aplikacija odziva na klike, pritisnjene gumbe na tipkovnici in kako razne komponente, kot so menuji, orodne vrstice, dialogi, gumbi, polja za urejanje, kontrole seznama, slike, forme, reagirajo na uporabnikove akcije [10].

Pri spletnih aplikacijah imamo za GUI teste na voljo sisteme, ki avtomatizirajo prave spletne brskalnike (Selenium [11]) in brezglavne brskalnike[2] (PhantomJS [12]). Ti simulirajo uporabniško interakcijo na spletni strani, v testu pa lahko preverimo rezultat.

## 2.6 Metrike testne pokritosti

Ena izmed najbolj popularnih metrik za merjenje kakovosti testov programske kode je testna pokritost. Metrike testne pokritosti so dober način za identifikacijo vrstic kode,

[2]Brezglavni brskalnik je spletni brskalnik brez grafičnega vmesnika. Lahko dostopa do spletnih strani, izriše vsebino, vendar je ne prikaže.

ki jih testi ne pokrivajo. To nam pride prav npr. pri identifikaciji robnih primerov, za katere v prvi fazi razvoja nismo napisali testov.

Ker moramo za generiranje te metrike pognati izvajanje testov, temu pravimo tudi dinamična analiza (nasprotno od statične analize programske izvorne kode, kjer se preverja samo izvorna koda).

Metrike testne pokritosti lahko izračunamo na tri načine:

1. C0 pokritost - Odstotek vrstic kode, ki so bile izvršene pri izvajanju testov
2. C1 pokritost - Odstotek vej, katerim je sledilo izvajanje testov vsaj enkrat
3. C2 pokritost - Odstotek unikatnih poti skozi izvorno kodo, katerim je sledilo izvajanje testov

V programskem jeziku Ruby je najbolj popularna metoda C0. Nizek rezultat je opozorilo za slabo testno pokritost, vendar pa visoka testna pokritost še ne zagotavlja, da so naši testi tudi dobro in temeljito napisani. Namreč teoretično bi lahko 100 odstotno pokritost zagotovili že z enim testom, v katerem pravzaprav ne bi preverjali ničesar [25].

Analiza testne pokritosti je procesorsko in spominsko zelo potraten proces, ki znatno upočasni graditev (v sistemu za zvezno integracijo), zato generiranje metrik testne pokritosti uvrstimo v fazo po uspešni izvedbi testov [28].

# 3 Orodja za testiranje v tehnologiji Ruby on Rails

Pred pojavom ogrodja Rails je bilo avtomatsko testiranje redko del spletnega razvoja. Spletno aplikacijo so testirali uporabniki in (mogoče) QA[1] ekipa za zagotavljanje kakovosti.

Med ostalimi ogrodji za spletni razvoj Rails posebej izstopa, saj ima testno ogrodje že vgrajeno. To je odsev avtorjeve[2] zavezanosti k metodologiji prej opisanega testno vodenega razvoja (TDD). Rails je uvedel disciplino testno vodenega razvoja in jo predstavil širši skupnosti spletnih razvijalcev. TDD je torej zelo značilen za Rails in od vsakega izkušenega Rails razvijalca se pričakuje, da ga obvlada [13].

## 3.1 MiniTest

MiniTest je minimalistično in hitro ogrodje za TDD in BDD testiranje, ter je vključeno v standardne knjižnice programskega jezika Ruby novejših verzij - od 1.9 dalje (v času pisanja je aktualna verzija 2.1). MiniTest je nadomestil starejše ogrodje Test::Unit in je vzratno kompatibilen s testi, napisanimi v starejšem ogrodju.

[1] Quality Assurance
[2] David Heinemeier Hansson

Primer funkcionalnega testa z uporabo ogrodja MiniTest:

```
require "minitest/autorun"
require_relative "../lib/person"

describe Person do

  describe "when name is empty" do
    it "is not valid" do
      person = Person.new
      person.wont_be :valid?
    end
  end

  describe "when name is not empty" do
    it "is valid" do
      person = Person.new("Yukihiro", "Matsumoto")
      person.must_be :valid?
    end

    it "has a full name" do
      person = Person.new("Yukihiro", "Matsumoto")
      person.full_name.must_equal "Yukihiro Matsumoto"
    end
  end
end
```

## 3.2 RSpec

Tudi RSpec je odprtokodno knjižnica za avtomatsko testiranje v jeziku Ruby, vendar ni standardno vključena in je popularna alternativa MiniTest ogrodju. V primerjavi z Minitest ponuja mnogo več funkcij in prvorazrednih objektov, ki vključujejo podporo za delo s koncepti testiranja (negacije, samo-opisovanje), medtem ko MiniTest uporablja zgolj enostavne metode za preverjanje testnih primerov. Vključuje tudi bogate metapo-

datke, deljene kontekste in svoj *domensko specifični jezik*[3], ki omogoča bolj pripovedno, jezikovno naravno pisanje testov.

Spodaj je primer boljše ekspresivnosti v primerjavi z MiniTest. V testu pričakujemo, da *create* akcija kontrolerja ustvari nov zapis:

```
# RSpec
expect {
  post :create, {post: valid_attributes}
}.to change(Post, :count).by 1

# MiniTest
assert_difference 'Post.count', 1 do
  post :create, {:post => valid_attributes}
end
```

RSpec je zaradi svojih ekspresivnih lastnosti najbolj popularno testno ogrodje v Rails [14] in v povezavi s Capybaro (opisano v nadaljevanju) tudi najbolj pogosto uporabljena rešitev. Tudi vsi prikazani primeri testov v nadaljevanju so narejeni za poganjanje v ogrodju RSpec.

## 3.3 Capybara

Je orodje za integracijsko testiranje spletnih aplikacij. Simulira uporabnikovo interakcijo s spletno stranjo. Ponuja enostaven domensko specifični jezik, ki omogoča visoko ekspresivnost pri pisanju in branju testov [15].

Oglejmo si primer integracijskega testa, kjer testiramo funkcijo avtentikacije določenega uporabnika. S preprosto sintakso, podobno naravnemu jeziku, test v spletno formo vnese uporabniško ime, geslo, klikne gumb za prijavo in preveri, ali je bila prijava uspešna:

```
feature "Signing in" do
  background do
    User.make(:email => 'user@example.com', :password => 'caplin')
  end
```

[3] ang. Domain Specific Language - DSL, je računalniški jezik, specializiran za določeno aplikacijsko domeno. Omogoča višjo abstrakcijo znotraj določenega programskega jezika.

```
scenario "Signing in with correct credentials" do
  visit '/sessions/new'
  within("#session") do
    fill_in 'Login', :with => 'user@example.com'
    fill_in 'Password', :with => 'caplin'
  end
  click_link 'Sign in'
  expect(page).to have_content 'Success'
end
```

### 3.3.1 JavaScript gonilniki

V spletnih aplikacijah različne akcije in elementi pogosto delujejo s pomočjo JavaScripta. V ta namen Capybara omogoča delo z JavaScript gonilniki, ki jih moramo posebej definirati. Najbolj popularen je Selenium, ki uporablja brskalnikovo okno in dejansko lahko v realnem času opazujemo, kako se izvajajo test in njegove akcije. Vendar je Selenium včasih neprikladen zaradi svoje počasnosti, namreč operacije izrisa v grafičnem vmesniku so počasne. Alternativa je WebKit ogrodje, ki poganja Chrome, Safari in večino brskalnikov na mobilnih napravah. Če ga uporabimo za JavaScript gonilnik pri Capybari, dobimo hitrost brezglavnega brskalnika in moč polnega, realnega JavaScript interpreterja [16].

Še en popularen gonilnik je Poltergeist. Le-ta integrira Capybaro s PhantomJS [12], ki je brezglavni WebKit brskalnik z JavaScript programskim vmesnikom. Je hiter in ima domorodno podporo za različne web standarde: ravnanje z objektnim modelom dokumenta, CSS selektorji, JSON, Canvas, SVG.

### 3.3.2 FactoryGirl

Temelj testiranja so testni podatki in ko poženemo funkcijske, integracijske in ostale teste, moramo zagotovoiti objekte, ki nosijo testne podatke, nad katerimi operirajo testi naših specifikacij.

Rails privzeto nudi podporo definiranja testnih podatkov z imenom *Fixtures*, ki so v bistvu .yml datoteke, v katerih definiramo atribute objektov:

```
article:
  title: Welcome to Rails!
  body: Hello world!
  category: about
```

Problem s Fixtures je ta, da so to zunanje odvisnosti, zaradi katerih testi lahko postanejo krhki in bolj dovzetni za napake. Predstavljajo fiksne, neprilagodljive zapise, ki se naložijo v testno podatkovno bazo.

Pametnejše nadomestilo za Fixtures so t.i. tovarne (ang. Factories [4], ki nam omogočijo predloge za veljavne in ponovno uporabljive objekte. V primerjavi z Fixtures nam omogočijo več možnosti pri instanciranju objektov in nam zagotovijo, da kot rezultat dobimo veljaven objekt.)

V Rails okolju je za kreiranje objektov testnih podatkov najbolj popularen vtičnik FactoryGirl [17].

Primer factory metode, ki kreira objekt za entiteto uporabnika:

```
FactoryGirl.define do
  factory :user do
    first_name "John"
    last_name "Doe"
    admin false
  end
```

V testu ga instanciramo z enostavnim klicem:

```
user = create(:user)
```

[4] Kreacijski vzorec programiranja, kjer se za kreacijo objektov brez da bi točno specificirali razred, uporabljajo factory metode

## 3.4 Cucumber

Cucumber je vtičnik za programski jezik Ruby (tudi za Javo, .NET), ki omogoča programsko razvojnim ekipam, da v golem besedilu opišejo, kako naj bi se programska oprema obnašala. Besedilo je napisano v poslovnem domensko specifičnem jeziku in služi kot dokumentacija, avtomatizirani testi in pomoč pri razvoju - vse to, združeno v enem formatu. Posebna prednost tega je, da lahko opise (teste) funkcionalnosti pišejo in razumejo tudi netehnični ljudje. Cucumber postavi učinkovito povezavo med poslovneži in razvojniki.

Pomembno je vedeti, da Cucumber ni nadomestilo za MiniTest ali RSpec in ni nizkonivojsko testno ogrodje. Igra pa ključno vlogo v vedenjsko vodenem razvoju (BDD). Cucumber je skupaj z RSpec celovita rešitev za izvedbo opisov vedenja programske opreme na zelo razumljiv način: tako, da jih razumejo vsi vpleteni.

### 3.4.1 Značilnost

Pri opisih Cucumber uporablja format značilnosti (ang. Feature). V definiciji značilnosti uporabimo besedilni opis uporabniške zgodbe:

```
Kot <vloga>
hocem <znacilnost>
da bi lahko <poslovna vrednost>
```

Ta format se osredotoča na naslednja tri pomembna vprašanja:

- Kdo uporablja sistem?
- Kaj počne?
- Kaj hoče s tem doseči?

### 3.4.2 Scenarij

Značilnost je definirana z enim ali več scenariji. Scenarij je sekvenca korakov, ki določa pot značilnosti. Scenarij je sestavljen iz treh delov (korakov):

1. Dano je: (ang. Given) - vzpostavi se začetno stanje oz. kontekst za scenarij

2. Ko: (ang. When) - predmet značilnosti, akcija in vedenje, na katerega se osredotočamo

3. Nato: (ang. Then) - preveri končne pogoje (če se je prejšnja stopnja izvedla pravilno)

Splošna forma scenarija je:

```
Scenarij: <opis>
  <korak 1>
       ...
  <korak n>
```

### 3.4.3 Primer značilnosti

```
Feature: coder starts game
  As a code-breaker
  I want to start a game
  So that I can break the code
  Scenario: start game
    Given I am not yet playing
    When I start a new game
    Then the game should say Welcome to CodeBreaker
    And the game should say Enter guess:
```

Ko poženemo ta test, nam Cucumber avtomatsko pripravi predlogo in ogrodje za nedefinirane korake, ki jih moramo izpolniti:

```
Given /^I am not yet playing$/ do
  pending
end
When /^I start a new game$/ do
  pending
```

```
end
Then /^the game should say Welcome to CodeBreaker$/ do
  pending
end
Then /^the game should say Enter guess:$/ do
  pending
end
```

Osnovna struktura je ključna beseda (Given, When, Then), ki ji sledi regularni izraz in blok Ruby kode. Regularni izraz služi za identifikacijo implementacije tega koraka. Ko je preko teh izrazov najdena implementacija za ta korak, se pod-nizi v opisu uporabijo kot argumenti za blok kode.

Primeri implementacij korakov iz opisa:

```
When /^I start a new game$/ do
    @messages = @game.start
end
```

```
Then /^the game should say &ldquo;(.*)&rdquo;$/ do |message|
  @messages.should include(message)
end
```

# 4 Vloga testov pri zvezni integraciji in zvezni postavitvi

Zvezna integracija je razvojna praksa, kjer razvijalci programske opreme večkrat na dan integrirajo svojo kodo v skupni repozitorij. Vsak tak prispevek sproži proces, imenovan graditev[1](ang: build), kateremu sledi preverjanje konfiguracij, poganjanje nabora testov in ostali mehanizmi za zgodnje odkrivanje napak.

## 4.1 Pomen

Če se poslužujemo pogostih integracij sprememb, močno zmanjšamo potreben čas za izsledite napak. To je hkrati tudi glavni pomen zvezne integracije, ki je zelo značilna za stabilne in uspešne programske projekte. Druge prednosti vključujejo:

- Zmanjšuje čas, potreben za dolge in tvegane integracije
- Zvišuje zavedanje o stanju projekta in posledično izboljša komunikacijo in zaupanje v projekt

[1] Pretvorba kode v konstrukte

- S pomočjo primerno napisanih testov hitro ujame napake in omogoči hitrejši razvoj novih specifikacij ali prilagoditve obstoječih

Slabost pa je čas, potreben za vzpostavitev tovrstnega sistema pri večjih projektih, še posebej, če se tega ne lotimo na začetku razvoja programskega projekta.

## 4.2 Koraki in odgovornosti

V okviru zvezne integracije je potrebno upoštevati naslednje korake:

1. Razvijalci iz skupnega repozitorija naredijo lokalno kopijo in si jo pogosto osvežujejo
2. Ko razvijalec konča z nekim delom, doda prispevek v repozitorij
3. Strežnik, na katerem teče sistem za zvezno integracijo, nadzira repozitorij (ali določeno vejo repozitorija) in ob spremembah sproži graditev in teste
4. Sporočanje o stanju trenutne graditve

Vsak sodelujoči pri tehničnem delu razvoja programske opreme je odgovoren za spoštovanje naslednjih pravil:

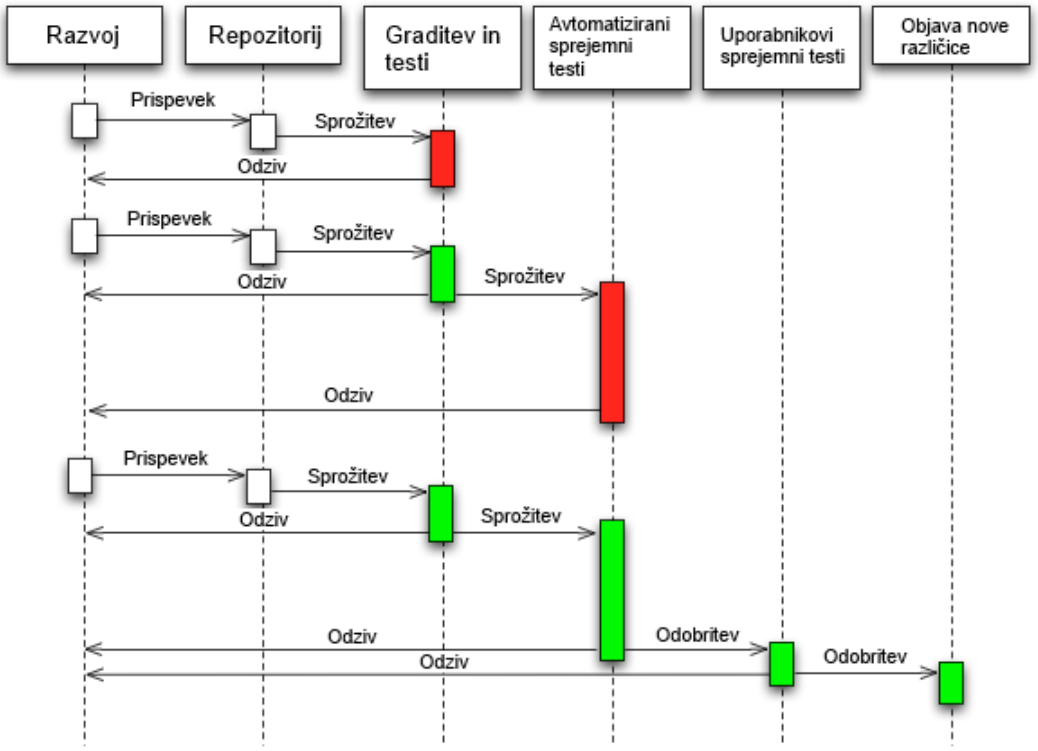
1. Pogosto prispevaj v repozitorij in pogosto sinhroniziraj lokalne kopije
2. Ne prispevaj nedelujoče kode
3. Ne prispevaj netestirane kode
4. Ne prispevaj kode, ko je graditev na strežniku v spodletelem stanju
5. Ko prispevaš v repozitorij, vedno preveri, ali je graditev na strežniku uspela

## 4.3 Zvezna postavitev

Zvezno postavitev (ang. Continuous deployment) lahko smatramo kot podaljšek zvezne integracije. Je sistem z avtomatiziranimi mehanizmi, ki z vsako uspešno integracijo (uspešna graditev) posodobi aplikacijo v določenem živem okolju (staging[2], produkcija).

Glavna prednost uporabe sistema za zvezno postavitev je zmanjšanje časa med fazo razvoja in fazo uporabe končnega uporabnika.

[2]Okolje, čimbolj podobno produkcijskem, kjer lahko testerji preverijo aplikacijo




**Slika 4.1** Diagram poteka cikla razvoja programske opreme s sistemom zvezne integracije in postavitve

## 4.4 Rešitve za Rails aplikacije

V Rails svetu obstaja množica odprtokodnih rešitev za zvezno integracijo (in postavitev), ki si jih lahko postavimo na lastni strežnik. Najbolj popularni [19] v času pisanja so:

- Travis CI
- Cruisecontrol
- Integrity
- Jenkins
- Big Tuna

Bolj pogosto pa Rails razvojne ekipe posežejo po rešitvah, ki jih različni ponudniki ponujajo kot gostovano storitev, saj je z njimi najmanj preglavic. Prednost takih rešitev

je v zelo kratkem času vzpostavitve, minimalni konfuguraciji, razširljivosti z API vmesniki, avtomatski povezavi z gostovanimi repozitoriji (GitHub, Bitbucket) in aplikacijskimi platformami (Heroku). Slabost pa lahko nekateri najdejo v ceni in v tem, da moramo tovrstnim storitvam odobriti bralni dostop do repozitorija, kar lahko privede do kraje kode in občutljivih podatkov pri hekerskih napadih ipd.

Trenutno so ene izmed najbolj popularnih gostovanih storitev za zvezno integracijo/postavitev naslednje:

- CircleCI
- Codeship
- Semaphore
- tddium

# 5 Uvedba testno usmerjenega razvoja v realnem projektu

Predmet testiranja je lastno razvita spletna aplikacija *SiteStalker*, zgrajena v tehnologiji Ruby on Rails. Namenjena je uporabnikom, ki želijo spremljati spletne strani, katerih URL naslove so vnesli in biti obveščeni takoj, ko se na njih pojavi kakšna nova hiperpovezava, ki vključuje podane ključne besede. Delovni tok je sledeč:

1. Uporabnik vpiše naslov spletnih strani, na katerih želi spremljati nove povezave
2. Uporabnik vpiše ključne besede, ki jih morajo vsebovati naslovi teh povezav
3. Sistem periodično procesira vnešene spletne strani in uporabniku prikaže rezultate ter pošlilja obvestila

Narejena je iz dveh neodvisnih delov, ki si delita podatkovno bazo - iz uporabniškega vmesnika in sistema za procesiranje.

## 5.1 Testno usmerjen razvoj sistema za procesiranje

Razviti moramo modul, ki bo prebral vsebino (HTML kodo) spletne strani, izluščil povezave v njej (hiperlinke) in nam jih vrnil v slovarju oblike (ključ: naslov povezave,

vrednost: URL povezave).

Napišimo prazen modul:

```
module Crawler
  class << self
  end
end
```

Nadaljujemo tako, da razmislimo, kaj moramo implementirati v modulu. To je - branje izvorne kode poljubne spletne strani. Bistvo testno usmerjenega razvoja je, da sprva napišemo padajoči **test enote**. V testu pričakujemo, da ima modul Crawler metodo read_source, ki kot argument sprejme URL spletne strani in nam v nizu vrne HTML izvorno kodo. Test uspe, če metoda, ki jo testiramo, vrne niz, ki vsebuje html in body značko, kar je značilno za HTML dokument.

```
describe Crawler do
  describe "extracting links from website" do
    describe "reading source code" do
      it "reads source html" do
        source = Crawler.read_source("http://google.com")
        source.should include("<html")
        source.should include("<body")
      end
    end
  end
end
```

Poženemo test in kot pričakovano, le-ta ne uspe, saj nismo v modulu še ničesar implementirali:

```
Failures:

  1) Crawler extracting links from website reading source code reads source html
     Failure/Error: source = Crawler.read_source("http://google.com")
     NoMethodError:
       undefined method `read_source' for Crawler:Module
     # ./spec/module/crawler\_spec.rb:26:in `block (4 levels) in <top (required)>
```

Naslednji korak je, da v modulu implementiramo metodo, za katero smo napisali test:

```
module Crawler
  class << self
    def read_source(url)
      Nokogiri::HTML(open(url, :allow_redirections => :all)).inner_html
    end
  end
end
```

Zopet poženemo test, ki tokrat uspe:

```
Crawler
  extracting links from website
    reading source code
      reads source html

Finished in 0.7212 seconds
1 example, 0 failures
```

V modulu manjka še funkcionalnost za ekstrakcijo hiperlinkov iz HTML izvorne kode. Sprva napišimo padajoči test enote, ki od našega modula pričakuje metodo extract_links,

ki za argument sprejme URL in HTML izvorno kodo spletne strani. Rezultat mora biti seznam slovarjev, ki imajo za ključ naslov hiperlinka, za vrednost pa njegov URL naslov. Za namen testa je vnaprej pripravljena, da lahko preverjamo pravilnost rezultata.

```
describe "extracting links" do
  let(:fake_url) { "http://example.com" }
  let(:fake_source) { "
<!DOCTYPE html>
<html>
  <head>
    <title>Test</title>
  </head>
<body>
  <h2>Test</h2>
  <a href="http://google.com">Google</a>
  <a href="http://testlink1.com/some/path">test link1</a>
  <a href="relative_link1">relative link1</a>
  <a href="/relative_link2">relative link2</a>
</body>
</html>" }
  let(:expected_links) {[ { "Google" => "http://google.com" },
                          { "test link1" => "http://testlink1.com/some/path" },
                          { "relative link1" => "#{fake_url}/relative_link1" },
                          { "relative link2" => "#{fake_url}/relative_link2" }
                          ]}

  it "extracts links from source" do
    links = Crawler.extract_links(fake_url, fake_source)
    links.should == expected_links
   end
 end
```

Tudi tokrat test pade, saj dejanske funkcionalnosti še nismo implementirali v modulu. Napišimo funkcionalno kodo:

```
def extract_links(url, source)
  links = []
  doc = Nokogiri::HTML(source)
  doc.css('a').each do |link|
    link_title = link.children.text
    link_url = link['href']
    if !link_title.blank? and !link_url.blank?
      link_url = fix_url url, link_url
      if url_valid? link_url and title_valid? link_title
        links << {link_title => link_url}
    end
    links
end
```

Še enkrat sprožimo izvajanje testov enot in opazujemo rezultat:

```
Crawler
  extracting links from website
    extracting links
      extracts links from source
    reading source code
      reads source html

Finished in 0.30504 seconds
2 examples, 0 failures
```

Vidimo lahko, da je novi test, skupaj s prvim, ki smo ga dodali, uspel. To nam da potrditev, da na novo napisana funkcionalnost deluje in da uvedba novih funkcionalnosti ni prinesla napak v obstoječi kodi, s tem pa smo tudi kodo pokrili s testi, ki nam bodo v prihodnosti služili kot varnostna varovalka. To je bistvo testno usmerjenega razvoja in

v takem duhu je bil razvit tudi preostali del modula za procesiranje (validacije URL-jev, naslovov, shranjevanje rezultatov v bazo,...).

## 5.2 Testno usmerjen razvoj uporabniške spletne aplikacije

### 5.2.1 Testi modelov

Modeli v spletni aplikaciji predstavljajo razrede, ki so predstavitev podatkovne relacije (tabele v podatkovni bazi).

Predmet testiranja je spletna aplikacija, namenjena registriranim uporabnikom, zato je uporabniški model prvi model, ki ga zgradimo.

Preden se lotimo pisanja kode, napišimo padajoči test uporabniškega modela, ki se bo odzival na klic tipičnih lastnosti in metod, kot so uporabniško ime, e-naslov, geslo, avtentikacija, itd.:

```
describe User do

  before do
    @user = User.new(username: "Example User", email: "user@example.com",
                     password: "foobar", password_confirmation: "foobar")
  end

  subject { @user }

  it { should respond_to(:username) }
  it { should respond_to(:email) }
  it { should respond_to(:password_digest) }
  it { should respond_to(:password) }
  it { should respond_to(:password_confirmation) }
  it { should respond_to(:authenticate) }
  it { should be_valid }

end
```

Ko se prepričamo, da test pade, se lotimo pisanja funkcionalne kode za uporabniški model. Kreiramo tabelo v podatkovni bazi, definiramo potrebne metode in avtentikacijsko metodo. Dodajmo še nekaj testov, s katerimi specificiramo sledeče osnovne stvari:

1. E-naslov mora biti unikaten na ravni sistema
2. E-naslov mora imeti veljaven format
3. Geslo ne sme biti prekratko
4. Metoda za avtentikacijo mora delovati

---

```
describe "when email address is already taken" do
    before do
      user_with_same_email = @user.dup
      user_with_same_email.email = @user.email.upcase
      user_with_same_email.save
    end

    it { should_not be_valid }
  end

  describe "when email format is invalid" do
    it "should be invalid" do
      addresses = %w[user@foo,com user_at_foo.org example.user@foo.
                     foo@bar_baz.com foo@bar+baz.com]
      addresses.each do |invalid_address|
        @user.email = invalid_address
        expect(@user).not_to be_valid
      end
    end
  end

  describe "when email format is valid" do
    it "should be valid" do
      addresses = %w[user@foo.COM A_US-ER@f.b.org frst.lst@foo.jp a+b@baz.cn]
```

```
    addresses.each do |valid_address|
      @user.email = valid_address
      expect(@user).to be_valid
    end
  end
end

describe "with a password that's too short" do
  before { @user.password = @user.password_confirmation = "a" * 5 }
  it { should be_invalid }
end

describe "return value of authenticate method" do
  before { @user.save }
  let(:found_user) { User.find_by_email(@user.email) }

  describe "with valid password" do
    it { should eq found_user.authenticate(@user.password) }
  end

  describe "with invalid password" do
    let(:user_for_invalid_password) { found_user.authenticate("invalid") }

    it { should_not eq user_for_invalid_password }
    specify { expect(user_for_invalid_password).to be_false }
  end
end
```

Enako kot pri prejšnjih postopkih, dopolnemo funkcionalno kodo do te mere, da vsi testi uspejo. Na tak način tudi nadaljujemo z izdelavo preostalih modelov, ki jih potrebujemo v aplikaciji (modeli za URL-je, ključne besede, skupine, rezultate, naročnine in plačila).

Velika prednost testno usmerjenega razvoja modelov je odlična testna pokritost že na podatkovnem nivoju aplikacije. Pomaga nam v večji meri zagotoviti pravilnost delova-

nja podatkovnega sistema in CRUD[1] operacij ter lažje, bolj modularno lociranje izvora napak, ko se le-te pojavijo.

### 5.2.2 Testi lastnosti

V ogrodju Rails se testi na najvišjem nivoju imenujejo testi lastnosti (ang. feature tests). So nekakšna mešanica **sprejemnih in integracijskih testov** ter so namenjeni testiranju povezanih kosov funkcionalnosti skozi aplikacijo. Tovrstni testi svoj testni tok vedno začnejo uporabniškem vmesniku. Pri tem uporabimo prej omenjeno Capybaro, saj nam s svojo ekspresivnostjo in jezikom omogoča resnično preprosto testiranje uporabniških zgodb na najvišjem nivoju.

Napišimo test za lastnost avtentikacije uporabnika. Avtentikacijska forma predstavlja vstopno točko v aplikacijo in je eden izmed ključnih elementov za testiranje.

Napišimo teste, ki zagotavljajo naslednje lastnosti procesa avtentikacije:

1. Zavrnjen dostop pri neveljavnem uporabniškem imenu ali geslu
2. Če registrirani uporabnik nima aktivnega plačila, mu mora sistem ponuditi možnost izbire naročnine in plačila
3. Če ima uporabnik aktivno plačilo, ga sistem spusti v uporabniški nadzorni prostor

---

```
describe "Authentication" do

  subject { page }

  describe "signin page" do
    before { visit root_path }

    it { should have_text("Sign up") }
    it { should have_text("Login") }
  end

  describe "signin" do
```

[1] Create, Read, Update, Delete

```
before { visit root_path }

describe "with invalid information" do
  before { click_on "Do it!" }

  it { should have_text("Login") }
  it { should have_text("Wrong login") }

  describe "after visiting another page" do
    before { click_link "SiteStalker" }
    it { should_not have_text("Wrong login") }
  end
end

describe "with valid information" do
  let(:plan1) { Plan.create(keyword_limit: 1, url_limit: 1, price: 100,
      name: "test plan1", frequency_period: 5*60)}
  let(:plan2) { Plan.create(keyword_limit: 1, url_limit: 1, price: 100,
      name: "test plan2", frequency_period: 5*60)}
  let(:plan3) { Plan.create(keyword_limit: 1, url_limit: 1, price: 100,
      name: "test plan3", frequency_period: 5*60)}
  let(:user) { User.create(username: "Example User", email:
      "user@example.com", password: "foobar",
  password_confirmation: "foobar", plan_id: plan1.id) }

  describe "with no active payment" do
    before do
      user.reload; plan2.reload; plan3.reload
      fill_in "session_email", with: "user@example.com"
      fill_in "session_password", with: "foobar"
      click_button "Do it!"
    end

    it "should offer payment plans" do
      page.should have_text "You have to make a payment to start using The
          SiteStalker."
```

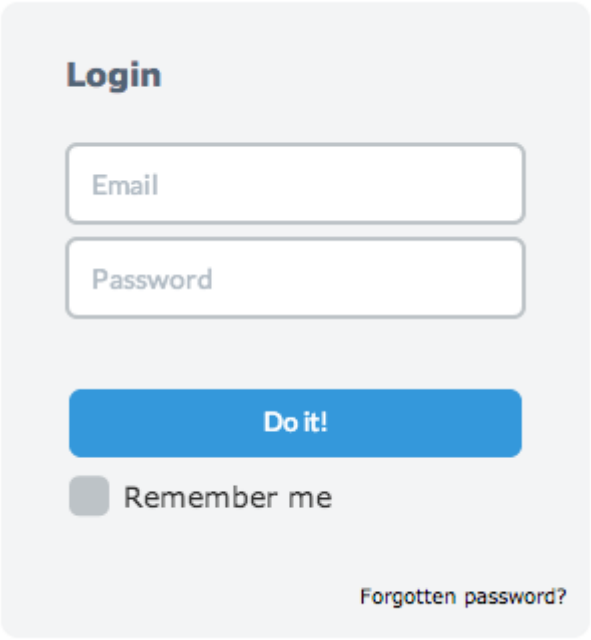
```
      end
    end

    describe "with active payment" do
      let(:payment) { Payment.create(user_id: user.id, plan_id: user.plan.id,
          valid_until: DateTime.now + 1.month) }
      before do
        user.reload; plan2.reload; plan3.reload; payment.reload
        fill_in "session_email", with: "user@example.com"
        fill_in "session_password", with: "foobar"
        click_button "Do it!"
      end

      it "should enter into application" do
        page.should have_text "Thanks for joining! For starters, add a group."
      end
    end
  end
 end
end
```

Testi lastnosti simulirajo dejanske, realistične delovne tokove končnega uporabnika. Služijo za potrditev uporabniških zgodb (navigiranje skozi aplikacijo, delo s formami, itd).

### 5.2.3 Vedenjsko usmerjen razvoj

Vedenjsko usmerjen razvoj (ang. kratica BDD) je izpeljanka testno usmerjenega razvoja, ki se osredotoča na testiranje lastnosti, ne rezultatov. Tovrstno testiranje je v Rails okolju zelo popularno, omogoča pa ga vtičnik Cucumber. Tudi BDD pristop je bil uporabljen v omenjeni spletni aplikaciji, ki je predmet našega testiranja.

Aplikacija je potrebovala funkcionalnost za resetiranje (pozabljenega) uporabniškega gesla. V datoteko *password_management.feature* zapišemo naslednji tekst:




**Slika 5.1** Avtentikacijska forma, ki je predmet testiranja zgornjega testa

```
Feature: Password management
  Scenario: Forgot password
    Given a registered user
    When I ask for a password reset
    Then I should see a page which says password reset link was sent to me
    And an email with a password reset link should be sent
```

Tako smo na zelo ekspresiven način in v naravnem jeziku zapisali, kakšno lastnost želimo od sistema. Tovrstno lastnost lahko z lahkoto napiše tudi oseba brez tehničnih znanj.

V terminalu poženemo ukaz *rake cucumber* in opazimo sledeči izpis:

```
You can implement step definitions for undefined steps with these snippets:

Given(/^a registered user$/) do
  pending # express the regexp above with the code you wish you had
end

When(/^I ask for a password reset$/) do
  pending # express the regexp above with the code you wish you had
end

Then(/^I should see a page which says password reset link was sent to me$/) do
  pending # express the regexp above with the code you wish you had
end

Then(/^an email with a password reset link should be sent$/) do
  pending # express the regexp above with the code you wish you had
end
```

Cucumber nam je izpisal korake, ki jih moramo dopolniti. Za opise uporabniške interakcije na spletni formi uporabljamo Capybara sintakso:

```
Given(/^a registered user$/) do
  @plan = Plan.create(keyword_limit: 1, url_limit: 1, price: 100)
  @user = User.create(username: "Example User", email: "user@example.com",
      password: "foobar",
     password_confirmation: "foobar", plan_id: @plan.id)
end

When(/^I ask for a password reset$/) do
  visit "/forgotten_passwords"
  fill_in "email", with: "user@example.com"
  click_button "Reset Password"
end
```

```
Then(/^I should see a page which says password reset link was sent to me$/) do
  page.should have_text "An email with password reset link was sent to
      #{@user.email}"
end

Then(/^an email with a password reset link should be sent$/) do
  ActionMailer::Base.deliveries.last.subject.should == "Forgotten password
      request"
  ActionMailer::Base.deliveries.last.decode_body.should have_link("Reset your
      forgotten password.", { :href =>
      "http://sitestalker.net/forgotten_passwords/#{@user.reload.reset_token}" })
end
```

Kodo dopolnjujemo toliko časa, dokler nam test z ukazom *rake cucumber* ne uspe:

```
Scenario: Forgot password
    Given a registered user
    When I ask for a password reset
    Then I should see a page which says password reset link was sent to me
    And an email with a password reset link should be sent

1 scenario (1 passed)
4 steps (4 passed)
```

Na enak način so s pomočjo BDD napisani še sprejemni testi za preostale funkcionalne lastnosti aplikacije (dodajanje skupin, URL-jev, ključnih besed,...).

Za aplikacijo je bilo napisanih 67 testov (testi enot, funkcijski testi, integracijski testi) in 5 scenarijev (BDD uporabniških zgodb), sestavljenih iz skupno 28 korakov.

### 5.2.4 Zvezna integracija in postavitev

Koda projekta je shranjena v Git repozitoriju[2], ki gostuje na spletni storitvi Bitbucket [20], ki ponuja brezplačne privatne repozitorije.

Namen Git repozitorija je upravljanje z datotekami projekta, ki se spreminjajo skozi čas in nam omogoči preklapljanje verzij datotek, dela na različnih vejah projekta in pregled zgodovine.

Ideja zvezne integracije je v tem, da se ob vsaki spremembi repozitorija sproži avtomatska graditev projekta. V Rails okolju je graditev sestavljena iz naslednjih korakov:

1. Osvežitev najnovejše kode iz repozitorija v lokalno kopijo
2. Namestitev vseh gemov določenih verzij (Ruby vtičniki) v sistem (ali ukaz za njihovo uporabo, če so že nameščeni)
3. Priprava podatkovne baze, izvajanje migracij
4. Poganjanje testov

V našem primeru je zvezna postavitev logično nadaljevanje zvezne integracije. Če se graditev v zvezni integraciji izvede uspešno, sledijo še naslednji koraki:

1. Osvežitev najnovejše kode iz določene veje repozitorija in prenos na na ciljni strežnik, platformo v oblaku,...
2. Izvedba skript za postavitev
3. Priprava podatkovne baze, izvajanje migracij

### 5.2.5 Uporabljene storitve

#### Bitbucket

Bitbucket [20] je spletna storitev za projekte, ki uporabljajo Git ali Mercurial sisteme za kontrolo različic. Je alternativa GitHubu, vendar s to razliko, da ponuja brezplačne privatne repozitorije.

[2]Distribuiran nadzor verzij programske opreme in sistem za upravljanje izvorne kode, ki ga je razvil Linus Torvalds

#### Heroku

Heroku [21] je aplikacijska platforma v oblaku (PaaS[3]), ki podpira več popularnih programskih jezikov. Omogoča izvajanje in postavitev dinamičnih spletnih aplikacij, podpira ogromno dodatkov, podatkovnih sktruktur, skaliranje procesov itd. Osredotoča se na to, da razvijalci svoj čas porabijo na svoji aplikaciji in ne na upravljanju strežnikov, postavitvah, sistemskih operacijah in skaliranju.

Za gostovanje na Heroku smo se odločili, ker je ta platforma v oblaku za tehnologijo Ruby on Rails zelo uveljavljena in popularna, pa tudi sama je narejena v Ruby on Rails.

#### Codeship

Codeship.io [22] je gostovana storitev za avtomatsko zvezno integracijo in postavitev spletnih aplikacij. Omogoča natančen in enostaven pregled celotnega cikla integracije, od detekcije spremembe repozitorija, graditve, poganjanja testov do postavitve na aplikacijski platformi.

Izbrali smo jo, ker omogoča izredno enostavno povezavo z Bitbucket in Heroku ter je brezplačna za 50 graditev na mesec.

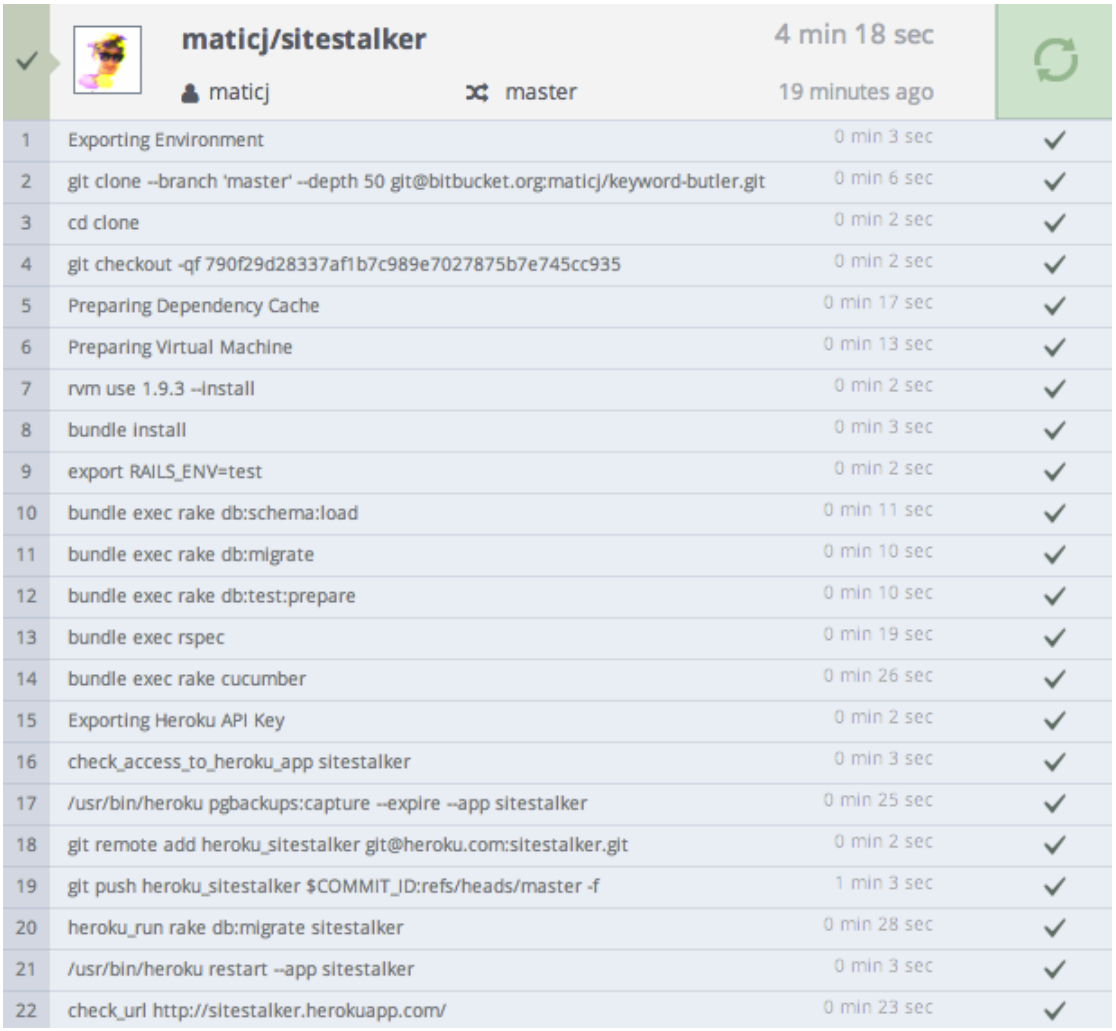
### 5.2.6 Avtomatska integracija aplikacije

Avtomatska zvezna integracija in postavitev aplikacije je izvedena s pomočjo prej omenjenih storitev. Takoj, ko Bitbucket repozitorij prepozna najmanjšo spremembo, sproži POST zahtevo na storitev Codeship. Slednja začne graditev, ki jo lahko v živo spremljamo v Codeship vmesniku.

Strnjen opis dogajanja v okolju storitve Codeship:

1. Osveževanje lokalne kopije repozitorija
2. Priprava Ruby okolja in izbira verzije
3. Osveževanje Gemov (Ruby vtičnikov)
4. Priprava testnega Rails okolja in testne podatkovne baze
5. Poganjanje testov

[3] Platform As A Service

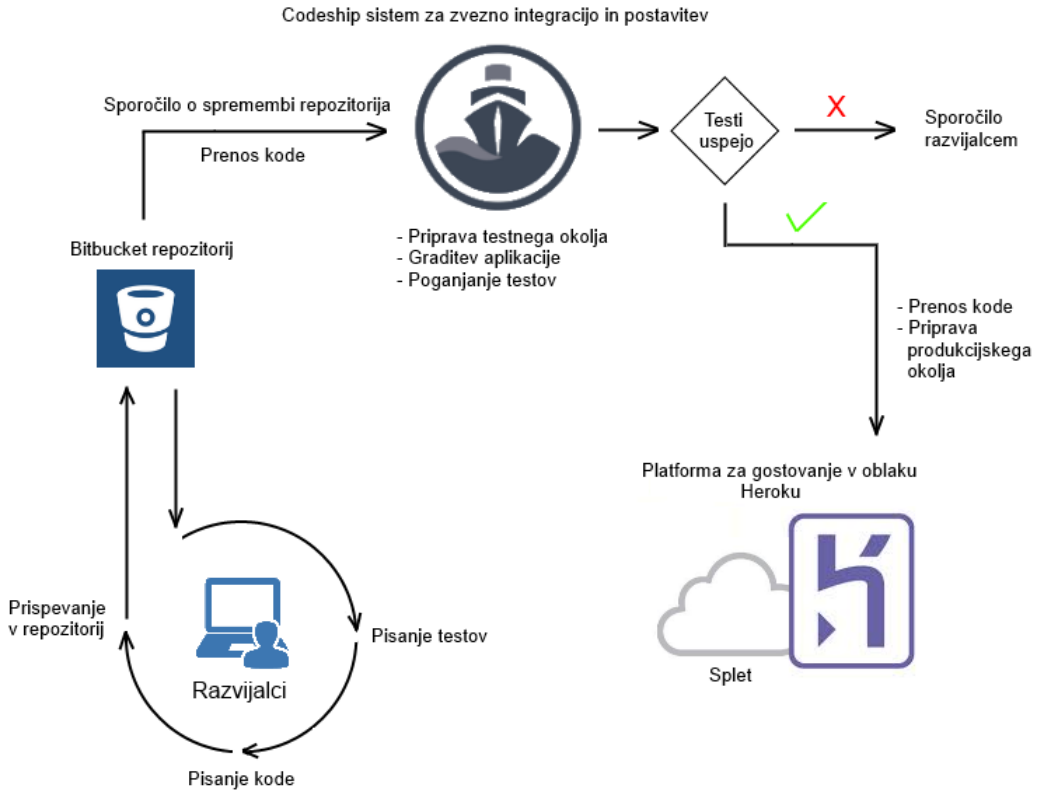


| | maticj/sitestalker | 4 min 18 sec | |
|---|---|---|---|
| | maticj master | 19 minutes ago | |
| 1 | Exporting Environment | 0 min 3 sec | ✓ |
| 2 | git clone --branch 'master' --depth 50 git@bitbucket.org:maticj/keyword-butler.git | 0 min 6 sec | ✓ |
| 3 | cd clone | 0 min 2 sec | ✓ |
| 4 | git checkout -qf 790f29d28337af1b7c989e7027875b7e745cc935 | 0 min 2 sec | ✓ |
| 5 | Preparing Dependency Cache | 0 min 17 sec | ✓ |
| 6 | Preparing Virtual Machine | 0 min 13 sec | ✓ |
| 7 | rvm use 1.9.3 --install | 0 min 2 sec | ✓ |
| 8 | bundle install | 0 min 3 sec | ✓ |
| 9 | export RAILS_ENV=test | 0 min 2 sec | ✓ |
| 10 | bundle exec rake db:schema:load | 0 min 11 sec | ✓ |
| 11 | bundle exec rake db:migrate | 0 min 10 sec | ✓ |
| 12 | bundle exec rake db:test:prepare | 0 min 10 sec | ✓ |
| 13 | bundle exec rspec | 0 min 19 sec | ✓ |
| 14 | bundle exec rake cucumber | 0 min 26 sec | ✓ |
| 15 | Exporting Heroku API Key | 0 min 2 sec | ✓ |
| 16 | check_access_to_heroku_app sitestalker | 0 min 3 sec | ✓ |
| 17 | /usr/bin/heroku pgbackups:capture --expire --app sitestalker | 0 min 25 sec | ✓ |
| 18 | git remote add heroku_sitestalker git@heroku.com:sitestalker.git | 0 min 2 sec | ✓ |
| 19 | git push heroku_sitestalker $COMMIT_ID:refs/heads/master -f | 1 min 3 sec | ✓ |
| 20 | heroku_run rake db:migrate sitestalker | 0 min 28 sec | ✓ |
| 21 | /usr/bin/heroku restart --app sitestalker | 0 min 3 sec | ✓ |
| 22 | check_url http://sitestalker.herokuapp.com/ | 0 min 23 sec | ✓ |

**Slika 5.2** Zvezna integracija in postavitev aplikacije na Codeship

6. Postavitev na platformo v oblaku Heroku

7. Preverjanje dostopnosti na novo postavljene aplikacije




**Slika 5.3** Vizualizacija uporabljenih orodij

# 6 Metrike testne pokritosti in scenariji v realnem projektu

Zaradi narave jezika Ruby in ogrodja Rails (dinamični programski jezik, hiter razvoj in prototipiranje) se kaj hitro pripeti, da nevede storimo razne tipkarske napake, kakšen razred ali modul postavimo na napačen kraj ali ga kličemo na napačen način, pa se tega zavemo šele, ko nekdo na to napako naleti v produkcijskem okolju. Temu se kot razvijalci seveda skušamo na vsak način izogniti.

Med delom na aplikaciji (izboljševanjem, dodajanjem funkcij, refaktoriranjem), še posebno v zaključnih fazah izdelave, ko je bila aplikacija že na voljo v produkcijskem okolju za testne končne uporabnike, smo mnogokrat dobili potrditev, da se nam je vložek časa za pisanje testov nedvomno izplačal. Za prej omenjene vrste napak so mi testi takoj javili, da je nekaj narobe in identificirali problematično mesto. Tako smo se večkrat uspešno izognili mučnemu razhroščevanju in prebiranju dnevnikov na produkcijskem okolju, kar zna včasih po nepotrebnem vzeti precej časa. S prisotnostjo primerno napisanih testov je tako razvijalec lahko mnogo bolj samozavesten in prepričan v svoje prispevke k projektu.

V nadaljevanju je poleg metrik testne pokritosti opisano nekaj najbolj zanimiv primerov, kjer nam je avtomatizirano testiranje (pri zvezni integraciji in postavitvi) prihranilo

marsikateri sivi las, podan pa je tudi primer, kjer se test zaradi pomankljivosti pri odkrivanju napak ni obnesel.

## 6.1 Testna pokritost

### 6.1.1 Rezultati

Za generiranje metrik testne pokritosti smo uporabili gem SimpleCov [26], ki nam po poganjanju testov generira lično poročilo v obliki HTML datoteke.

Odstotki testne pokritosti našega testnega nabora so sledeči:

1. Vse datoteke: 82.41%
2. Kontrolerji: 53.83%
3. Modeli: 91.92%
4. Mailerji: 54.55%
5. Pomožne metode: 84.55%
6. Knjižnice: 83.33%

### 6.1.2 Komentar

Odstotek testne pokritosti projekta je nadpovprečen, vendar ne ravno najboljši. Najmanjši je pri testih kontrolerjev, kjer so bili testi napisani samo za akcije pri najbolj pomembnih tokovih uporabe. Bolj vzpodbuden odstotek je pri odstotku testne pokritosti modelov, knjižnic in pomožnih metod kjer testi pokrivajo večino pomembnih validacij in akcij.

Kolikšen odstotek testne pokritosti sploh pomeni, da je testna pokritost dobra? Mnenja so deljena. Nekateri so mnenja, da je 100 odstotna testna pokritost naraven stranski učinek pravilnega testnega razvoja. Nekateri drugi pa so pristaši razmišljanja, da zelo visoka testna pokritost daje zgolj lažen občutek izboljšane varnosti in da ne bi nikoli smeli stremeti k 100 odstotni testni pokritosti [25].

Pri ugotavljanju, kakšen odstotek testne pokritosti je pri določenem projektu dovolj dober, moramo uporabiti lastno presojo. Različni projekti z različnimi razvijalci imajo

lahko različne vrednosti za optimalno testno pokritost pri evoluciji projekta (četudi sploh ne spremljajo omenjenih metrik).

## 6.2 Prikaz koristi testiranja v realnih scenarijih

### 6.2.1 Refaktoriranje

V Rails projektu so poti (ang. routes [1]) definirane v datoteki *routes.rb*. Ena izmed njih je definirana na naslednji način:

```
match '/signin', to: 'sessions#new'
```

To navodilo pomeni, da pot /signin vodi k akciji kontrolerja za seje, ki prikaže formo za prijavo uporabnika. Pot /signin smo želeli spremeniti v /login, saj smo želeli večji kontrast med potjo za registracijo uporabnika, ki se glasi /signup.

Spremenimo pot v /login:

```
match '/login', to: 'sessions#new'
```

Pri taki spremembi je potrebno v kodi poiskati vse reference s to potjo in jih zamenjati, da bodo v skladu z novo. Vedoč, da imamo z napisanimi testi aplikacijo dobro pokrito, sprva poženemo teste na lokalnem razvojnem sistemu. Eden izmed testov pade. Pokvarili smo povezavo na prijavno formo, ki se nahaja na strani za registracijsko formo:

```
Failures:

  1) Authentication signup page leading to login site
     Failure/Error: click_link "Log In"
     ActionController::RoutingError:
       No route matches [GET] "/signin"
     # ./spec/requests/authentication_pages_spec.rb:10
```

[1] Prek poti aplikacija prepozna URL-je in jih posreduje v akcije kontrolerjev.

Preko padlega testa lahko takoj identificiramo mesto v kodi, kjer imamo še staro referenco. Dodamo popravek in zopet poženemo teste.

```
Finished in 8.26 seconds
67 examples, 0 failures
```

Napako, ki smo jo pridobili zaradi spremembe, smo odpravili. Popravek prispevamo v repozitorij in sistem za zvezno integracijo in postavitev Codeship na strežnik postavi novo verzijo.

Onemogočena je bila povezava med registracijsko in prijavno stranjo, zato pridobljena napaka za aplikacijo ni bila ravno izjemnega pomena, saj se obrazec za prijavo avtomatsko izriše že na prvi strani aplikacije. V primeru, da bi ta napaka zašla v produkcijsko okolje, poslovne škode verjetno ne bi bilo, a vendar bi kredibilnost in ugled aplikacije zaradi okvarjene povezave pri uporabnikih zagotovo padla.

Pisanje testa za nastalo situacijo je trajalo zanemarljivo količino časa, napaka je bila odkrita takoj, popravek pa trivialen.

Čeprav je po spremembah poti in refaktoriranju logičen naslednji korak posodabljanje starih poti v kodi, lahko ob spodobnem testnem naboru ta korak preskočimo in pustimo testom, da odkrijejo mesta pomankljivosti. Tako smo ob prispevanju sprememb bolj samozavestni in prepričani v pravilnost delovanja aplikacije, privarčujemo pa tudi čas, ki bi ga porabili ob brskanju po kodi, iskajoč reference na spremenjene poti.

### 6.2.2 Pomankljivo napisan test

Kot smo že omenili v prejšnjih poglavjih, je sistem za procesiranje (razčlenjevanje in iskanje novih vsebin) spletnih strani, katerih URL-je vnesejo uporabniki, eden izmed najbolj pomembnih delov aplikacije. Zanj so testi še posebej pomembni, saj je jedrni del aplikacije, ki v bazo redno vnaša novo vsebino.

Eden izmed prvih testnih uporabnikov je po naključju odkril, da sistem v izvorni kodi spletnih strani ne zazna vseh povezav, ki bi jih moral. Po ročnem preverjanju smo odkrili zanimiv problem, in sicer ta, da sistem ne upošteva URL-jev, ki imajo v naslovu

Unicode[2] znake. Primeri takih URL-jev so:

1. http://www.example.com/ä
2. http://en.wikipedia.org/wiki/$\phi$
3. http://ja.wikipedia.org/wiki/未練なく散

Težava je bila v metodi, ki je preverjala pravilnost URL naslova. V primeru, da je spodnja metoda vrnila *true*, se je procesiranje za ta URL izvedlo, v nasprotnem primeru pa ga je preskočilo:

```
def url_valid?(url)
  begin
    uri = URI.parse(URI.encode(url))
    uri.kind_of?(URI::HTTP)
  rescue URI::InvalidURIError
    false
  end
end
```

Za zgornjo metodo je sicer že obstajal test enote, vendar s premajhno testno množico. Le-ta je bila dopolnjena, da je vključevala tudi URL-je z Unicode znaki [24]. Po dopolnitvi testne množice URL-jev ponovno poženemo teste in pričakujemo, da test zgornje metode sedaj pade, saj smo testno množico veljavnih URL naslovov razširili, metode same pa še nismo popravili:

```
Failures:

  1) Crawler strict url validation should return invalid for invalid examples
     Failure/Error: Crawler.url_valid?(url).should == false
       expected: false
            got: true (using ==)
```

[2]Mednarodni kodni standard, kjer je vsaka črka, številka ali simbol predstavljena z numerično vrednostjo, ki velja za različne platforme in aplikacije

```
    # ./spec/module/crawler_spec.rb:35:in `block (4 levels) in <top (required)>'

Finished in 0.84448 seconds
6 examples, 1 failure
```

Odločimo se, da metodo, ki preverja veljavnost URL naslova, napišemo na novo in sicer z uporabo regularnega izraza, ki smo ga napisali za popolno validacijo URL naslovov:

```
class RegularExpressions

  PERFECT_URL_PATTERN = %r{
    \A

    # protocol identifier
    (?:(?:https?|ftp)://)

    # user:pass authentication
    (?:\S+(?::\S*)?@)?

    (?:
      # IP address exclusion
      # private & local networks
      (?!10(?:\.\d{1,3}){3})
      (?!127(?:\.\d{1,3}){3})
      (?!169\.254(?:\.\d{1,3}){2})
      (?!192\.168(?:\.\d{1,3}){2})
      (?!172\.(?:1[6-9]|2\d|3[0-1])(?:\.\d{1,3}){2})

      # IP address dotted notation octets
      # excludes loopback network 0.0.0.0
      # excludes reserved space >= 224.0.0.0
      # excludes network & broacast addresses
      # (first & last IP address of each class)
```

```
      (?:[1-9]\d?|1\d\d|2[01]\d|22[0-3])
      (?:\.(?:1?\d{1,2}|2[0-4]\d|25[0-5])){2}
      (?:\.(?:[1-9]\d?|1\d\d|2[0-4]\d|25[0-4]))
    |
      # host name
      (?:(?:[a-z\u00a1-\uffff0-9]+-?)*[a-z\u00a1-\uffff0-9]+)

      # domain name
      (?:\.(?:[a-z\u00a1-\uffff0-9]+-?)*[a-z\u00a1-\uffff0-9]+)*

      # TLD identifier
      (?:\.(?:[a-z\u00a1-\uffff]{2,}))
    )

    # port number
    (?::\d{2,5})?

    # resource path
    (?:/[^\s]*)?

    \z
  }xi

end

def url_valid?(url)
  (RegularExpressions::PERFECT_URL_PATTERN =~ url) == 0
end
```

Znova poženemo teste in vsi uspejo. Tako smo popravili metodo in njen test, ki zdaj zagotavljata, da sistem pri razčlenjevanju HTML izvorne kode podanih spletnih strani ne zavrže nobenih URL naslovov, ki so v resnici veljavni.

Ta zgodba nazorno demonstrira situacijo, ko nam pomankljivo napisan test lažno zatrjuje, da testirana metoda deluje pravilno. Po principu testno usmerjenega razvoja

v takem primeru vedno popravimo ali dopolnemo test, da reproduciramo napako, ki se pojavlja, da dobimo padajoči test. Nato subjekt testiranja popravljamo toliko časa, dokler test zopet ne uspe.

V tem primeru nam prvotno napisani test ni prinesel koristi, saj je bil pomankljivo napisan, ker smo v fazi načrtovanja pozabili, da imamo v realnosti lahko opravka z omenjenimi URL naslovi s posebnimi znaki in smo posledično za validacijo uporabili napačno orodje.

Tovrstne napake se v realnem, produkcijskem okolju pogosto pojavljajo, saj je v fazi načrtovanja, pisanja kode in testiranja nemogoče predvideti vse možne scenarije in interakcije, s katerimi se aplikacija sooča v realnem svetu. V takih primerih je pomembno, da pri popravljanju napak dopolnemo še test, ki potrdi, da smo napako res odpravili. Tako zagotovimo večjo testno pokritost in postopno ožanje luknje, od koder pridejo napake.

### 6.2.3 Sprememba konfiguracije

Ena izmed najbolj pogostih in pomembnih interakcij med našo aplikacijo in uporabniki poteka prek elektronske pošte. Uporabniki prek nje dobivajo obvestila o novih rezultatih, o dogajanjih na njihovih uporabniških računih itd., zato je to esencialen del aplikacije.

Uporabniško ime in geslo za dostop do SMTP e-poštnega strežnika smo hranili v konfiguracijski datoteki v repozitoriju kar v čisti tekstovni obliki, kar se smatra za slabo prakso [23].

```
config.action_mailer.smtp_settings = {
        address: 'smtp.mandrillapp.com',
        port: 587,
        domain: 'sitestalker.net',
        user_name: username,
        password: password,
        authentication: 'plain',
        enable_starttls_auto: true }
```

Dobra praksa v Rails okolju veleva, da razna gesla, ključe in občutljive avtentikacijske

podatke za zunanje storitve hranimo v okoljskih spremenljivkah sistema[3], ločeno od Rails aplikacije.

Torej, spremenimo konfiguracijski del, da aplikacija prebere uporabniško ime in geslo iz okoljskih spremenljivk:

```
config.action_mailer.smtp_settings = {
       address: 'smtp.mandrillapp.com',
       port: 587,
       domain: 'sitestalker.net',
       user_name: ENV["MANDRILL_USERNAME"],
       password: ENV["MANDRILL_PASSWORD"],
       authentication: 'plain',
       enable_starttls_auto: true }
```

Med testi za našo aplikacijo imamo tudi takega, ki testira pošiljanje e-poštnih sporočil. V tej točki bi moral ta test pasti, saj v okoljske spremenljivke svojega razvojnega sistema nismo vnesli prej omenjenih avtentikacijskih podatkov. Poskusimo:

```
Feature: Password management

  Scenario: Forgot password                                    #
      features/password_management.feature:2
    Given a registered user                                    #
        features/step_definitions/navigation_steps.rb:1
    When I ask for a password reset                            #
        features/step_definitions/navigation_steps.rb:7
      454 4.7.1 <user@example.com>: Relay access denied
       (Net::SMTPServerBusy)
      ./app/controllers/forgotten_passwords_controller.rb:15:in `create'
      ./features/step_definitions/navigation_steps.rb:10:in `/^I ask for a
          password reset$/'
      features/password_management.feature:4:in `When I ask for a password reset'
```

[3]Množica spremenljivk, ki vpliva na tekoče procese

```
 Failing Scenarios:
cucumber features/password_management.feature:2 # Scenario: Forgot password
```

Kot pričakovano, je test padel, saj se brez avtentikacijskih podatkov ne moremo vpisati v zunanjo storitev za razpečevanje e-pošte. V okoljske spremenljivke sistema vnesemo omenjena podatka in znova poženemo teste:

```
rspec
Finished in 8.3 seconds
67 examples, 0 failures

rake cucumber
5 scenarios (5 passed)
28 steps (28 passed)
0m10.348s
```

Vse izgleda v redu in odločimo se, da je delo končano. Popravek prispevamo v glavno vejo repozitorija, da sprožimo graditev in samodejno postavitev nove verzije v produkcijsko okolje. Le nekaj minut zatem dobimo avtomatizirano obvestilo iz sistema za zvezno integracijo in postavitev (Codeship), da je graditev spodletela. Ko odpremo dnevnik zadnje graditve v Codeship, opazimo, da je eden izmed testov padel:

```
Feature: Password management
  Scenario: Forgot password
    Given a registered user
    When I ask for a password reset
      454 4.7.1 <user@example.com>: Relay access denied
       (Net::SMTPServerBusy)
      ./app/controllers/forgotten_passwords_controller.rb:15
      ./features/step_definitions/navigation_steps.rb:10
      features/password_management.feature:4
Failing Scenarios:
cucumber features/password_management.feature:2 # Scenario: Forgot password
5 scenarios (1 failed, 4 passed)
28 steps (1 failed, 2 skipped, 25 passed)
```

**Slika 6.1** Dnevnik graditve aplikacije v sistemu Codeship

Ta napaka nam je znana še od takrat, ko smo spremembo testirali na lokalnem, razvojnem sistemu in nam je test padel, ker nismo nastavili okoljskih spremenljivk. Spomnemo se, da smo pozabili vnesti spremenljivke za avtentikacijo storitve za razpečevanje e-poštnih sporočil tako v okolju zvezne integracije kot tudi v produkcijskem okolju.

Ko smo vnesli manjkajoče okoljske spremenljivke, naslednja graditev uspe z vsemi testi vred in sistem avtomatizirano postavi novo verzijo v produkcijsko okolje. V tem primeru smo demonstrirali korist dvonivojskega testiranja (najprej lokalno, nato v sistemu za zvezno integracijo), saj lahko na tak način odkrijemo napake v situacijah, ko testi lokalno uspejo, v drugem okolju pa zaradi različnih pomankljivosti ne.

Če v tem primeru ne bi imeli avtomatiziranega testiranja, napake (oz. dejstva, da smo pozabili vnesti nove okoljske spremenljivke v testnem in produkcijskem sistemu) ne bi zgodaj odkrili in bi v produkcijsko okolje nevede postavili verzijo, ki sproži izjemo pri pošiljanju elektronskih sporočil uporabnikom.

Čeprav je čas, potreben za popravek napake, zelo kratek, bi nam ta napaka lahko naredila precejšnjo poslovno škodo. Razlog je ta, da bi napako odkrili šele, ko bi nanjo naletel eden ali več uporabnikov. V najboljšem primeru bi naleteli na nezadovoljstvo uporabnikov, v naslabšem (in najbolj realnem) pa prekinitev plačanih uporabniških naročnin. Za pisanje tega testa smo porabili zgolj nekaj minut, kar je zanemarljivo glede na njegovo korist, ki je demonstrirana v tem primeru.

# 7 Zaključek

V diplomskem delu smo opisali potek testno usmerjenega razvoja v teoriji, kot tudi v praksi na realnem projektu, narejenem v tehnologiji Ruby on Rails. Dosegli smo praktično vse cilje, ki jih poskušamo doseči z uvedbo postopkov testiranja programske opreme, t.j. hitro odkrivanje in preprečevanje napak, visoka stopnja kvalitete produkta, predvsem pa učinkovit razvojni proces in minimiziranje stroškov (časa) vzdrževanja. Z avtomatizacijo testov s pomočjo zvezne integracije in postavitve smo naredili izdelek, ki je stabilen, lahek za vzdrževanje in izjemno enostaven za dodajanje novih verzij.

Seveda vedno in povsod obstaja tudi prostor za napredek in izboljšanje. Tudi naš testni proces je možno še izboljšati. Ena izmed izboljšav, ki bi jo lahko uvedli pri razvojnem procesu, je uporaba orodij, ki samodejno poganjajo teste ob zaznavi sprememb datotek v lokalnem razvojnem okolju. V okolju Rails je najbolj popularna rešitev Guard::RSpec [27]. Omogoči nam, da smo o uspehu testov obveščeni že med pisanjem kode in nam posledično privarčuje nekaj časa, ki bi ga izgubili z ročnim poganjanjem testu po tem, ko smo zaključili z urejanjem.

Naslednja stvar, ki bi jo lahko še dodelali, je optimizacija hitrosti izvajanja testov.

V povprečju se testi za našo aplikacijo izvajajo približno 20 sekund, kar sicer ni problematično, vendar bi v primeru rasti aplikacije in s tem števila testov potreben čas lahko hitro narastel na več minut. Ob pogostih in hitrih spremembah nas počasni testi zelo omejujejo. Najbolj pogoste rešitve za take težave vključujejo zmanjšanje število klicev podatkovne baze, izvajanje integracijskih testov zgolj v sistemu za zvezno integracijo in postavitev ter paralelno izvajanje testov na več jedrih procesorja.

# LITERATURA


[1] (2014) David Heinemeier Hansson, Ruby on Rails, mar. 2014. Dostopno na: http://rubyonrails.org

[2] (2013) Wikipedia, Unit Testing, mar. 2014. Dostopno na: http://en.wikipedia.org/wiki/Unit_testing

[3] (2014) Wikipedia, Test driven development, apr. 2014. Dostopno na: http://en.wikipedia.org/wiki/Test-driven_development

[4] Robert C. Martin. *Agile Software Development, Principles, Patterns, and Practices.* Prentice Hall, 2002.

[5] (2009) Garry Shutler, What is BDD (Behaviour Driven Design), apr. 2014. Dostopno na: http://blog.robustsoftware.co.uk/2009/11/what-is-bdd-behaviour-driven-design.html

[6] (2013) Josh Davis, The Difference Between TDD and BDD, apr. 2014. Dostopno na: http://joshldavis.com/2013/05/27/difference-between-tdd-and-bdd

[7] (2013) Aslak Hellesøy, Cucumber, apr. 2014. Dostopno na: http://cukes.info

[8] (2009) QTP Tutorials & Interview Questions, Difference between Acceptance testing & System testing, apr. 2014. Dostopno na: http://qtp.blogspot.com/2009/04/difference-acceptance-system-testing.html

[9] (2013) Wikipedia, Regression Testing, apr. 2013. Dostopno na: http://en.wikipedia.org/wiki/Regression_testing

[10] (2013) AppPerfect, GUI testing, apr. 2014. Dostopno na:
http://www.appperfect.com/products/application-testing/app-test-gui-testing.html

[11] (2013) SeleniumHQ, Selenium Browser Automation, apr. 2014. Dostopno na:
http://docs.seleniumhq.org

[12] (2013) Ariya Hidayat, PhantomJS, apr. 2014. Dostopno na:
http://phantomjs.org

[13] (2013) Daniel Kehoe, What is Ruby on Rails?, apr. 2014. Dostopno na:
http://railsapps.github.io/what-is-ruby-rails.html

[14] (2013) Christoph Olszowka, Unit Test Frameworks, apr. 2014. Dostopno na:
https://www.ruby-toolbox.com/categories/testing_frameworks

[15] (2014) Jonas Nicklas, Acceptance test framework for web applications, maj 2014. Dostopno na:
https://github.com/jnicklas/capybara

[16] (2014) JumpstartLab, JavaScript Testing with Selenium & Capybara-Webkit, maj 2014. Dostopno na:
http://tutorials.jumpstartlab.com/topics/capybara

[17] (2014) ThoughtBot, A library for setting up Ruby objects as test data, maj 2014. Dostopno na:
https://github.com/thoughtbot/factory_girl

[18] (2013) Agile Alliance, Continuous Deployment, maj 2014. Dostopno na:
http://guide.agilealliance.org/guide/cd.html

[19] (2013) The Ruby Toolbox, Continuous Integration, maj 2014. Dostopno na:
https://www.ruby-toolbox.com/categories/continuous_integration

[20] (2014) Atlassian, Atlassian Bitbucket, maj 2014. Dostopno na:
https://bitbucket.org

[21] (2014) Heroku, Heroku, maj 2014. Dostopno na:
https://www.heroku.com

[22] (2014) Codeship, Continuous Deployment made simple, maj 2014. Dostopno na: https://www.codeship.io

[23] (2014) Heroku, Heroku, maj 2014. Dostopno na: https://devcenter.heroku.com/articles/config-vars

[24] (2013) Mathias Bynens, In search of the perfect URL validation regex, maj 2014. Dostopno na: http://mathiasbynens.be/demo/url-regex

[25] (2013) Code Climate, Deciphering Ruby Code Metrics, maj 2014. Dostopno na: http://blog.codeclimate.com/blog/2013/08/07/deciphering-ruby-code-metrics

[26] (2013) Christoph Olszowka, Code coverage for Ruby 1.9, maj 2014. Dostopno na: https://github.com/colszowka/simplecov

[27] (2013) Guard, Guard RSpec allows to automatically and intelligently launch specs when files are modified, maj 2014. Dostopno na: https://github.com/guard/guard-rspec

[28] John Ferguson Smart. *Jenkins: The Definitive Guide*. O'Reilly, 2011.