LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

HIPOTĒŽU VIRZĪTAS IZSTRĀDES IZPĒTE

MAĢISTRA DARBS

Autors: **Marta Lapiņa**
Stud. apl. Nr. ml11055
Darba vadītājs: Dr. dat. Normunds Grūzītis

RĪGA 2017

ANOTĀCIJA

Projektos, kas īsteno spējo izstrādi, ir pieejamas izmantošanai vairākas izstrādes metodes, lai nodrošinātu produkta kvalitāti līdz ar piegādes ātrumu un uzticamību. Darba mērķis ir izpētīt vienu no šādām metodēm, hipotēžu virzītu izstrādi, gan no teorētiskā, gan praktiskā skatu punkta. Nav zināms, cik lielā mērā šī pieeja ir saistīta vai atkarīga no citām “X virzītas izstrādes” pieejām, tādām kā testu virzīta izstrāde un uzvedības virzīta izstrāde, vai vēl kādām citām izstrādes metodēm. Ja sistēmas izstrādātāji veic novērojumus par sistēmu, tie var izvirzīt hipotēzes saistībā ar to. Hipotēžu izmantošana izstrādes procesā norāda uz eksperimenta iesaistīšanu sistēmas iespēju izstrādē. Ja izstrādātāji ir īstenojuši eksperimentu ar jaunu iespēju sistēmā, iegūtie rezultāti noder, lai pieņemtu lēmumu, vai šo iespēju paturēt sistēmā.

ABSTRACT

Projects that use agile development approach have several methods they may use to ensure product quality along with speed and realiability of delivery process. “Exploration of Hypothesis Driven Development” focuses on one of such method, namely, Hypothesis Driven Development, and explores it from both theoretical and practical point of view. It is not yet clear how is HDD connected to other xDD approaches, such as TDD and BDD. Given that developers make an observation about a system, they can form a hypothesis about it. Use of hypothesis in development process signifies experimental approach in new feature development. If development team has conducted an experiment on a newly released system feature, results should be helpful when deciding whether to keep the new feature.

AUTOREFERĀTS

Darba novitāte. Autore izvēlējusies pētīt tēmu, kas tai šķiet maz pazīstama nozarē, vismaz Latvijas mērogā. Autore cer, ka viņas darba rezultāti būs noderīgi kolēģiem, kas meklē veidu, kā panākt produktīvāku izstrādātāju un pasūtītaju sadarbošanos projektā.

Literatūras izpēte. Autore ir pētījusi gan literatūru, gan tiešsaistes avotus, recenzētus un nerecenzētus. Pēdējie pārsvarā izmantoti attēlu ieguvei.

Izklāsta pamatīgums. Darbā aplūkotās problēmas un risinājumi ir aprakstīti gan vispārīgi, gan padziļināti, ar piemēriem. Apjomīgas vai sarežģītas informācijas prezentācijai papildus izmantoti paskaidrojoši attēli.

Paveiktā praktiskā darba apjoms. Autore pati ir veikusi sagatavošanos darba praktiskajai daļai: atradusi projektu, kas piedalītos pētījumā; definējusi priekšnosacījumus HVI realizācijai un pārbaudījusi, vai izvēlētajā projektā tie atrodami; informējusi projekta komandu par to, kas ir HVI un kā tas tiks īstenots; konsultējusi projektu praktiskās daļas laikā un ievākusi komandas atsauksmes par pieredzi HVI izmantošanā.

Rezultātu aprobācija. Autorei neizdevās atrast analogu pētījumu pieejamajos informācijas avotos. Ir atrodami raksti par to, kā īstenot HVI, taču tajos netika izvērsti pastāstīts par konkrētu projektu pieredzi ar šo metodi, arī atsauksmes par šīs metodes izmantošanu netika atrastas. Autores darba mērķis bija uzzināt pēc iespējas daudz par HVI un tās saistību ar citām izstrādes metodēm, un autore uzskata, ka to ir paveikusi.

Darba noformējuma kvalitāte. Darba teksts ir pārlasīts un atrastās ievades, komatu un vārdu locījumu kļūdas ir izlabotas. Autore ir pārliecinājusies, ka darbā nav pareizrakstības kļūdu. Darbā ir izmantota latviešu valodā oficiāli pieņemtā nozares terminoloģija.

Plaģiāta iespējamība. Idejas, formulējumi, attēli utt., kas aizgūti no citiem autoriem, ir atzīmēti ar attiecīgām literatūras atsaucēm.

SATURS

APZĪMĒJUMU SARAKSTS 7

IEVADS 8

1. KODA IZMAIŅU VEIKŠANA, STRĀDĀJOT SPĒJĀS IZSTRĀDES KOMANDĀ 10

1.1. Izmaiņu integrēšana un tās biežuma nozīme 10

1.1.1. Nepārtrauktā integrācija 11

2. TESTU VIRZĪTA IZSTRĀDE UN TĀS LOMA CITU PROGRAMMATŪRAS IZSTRĀDES METOŽU ATTĪSTĪBĀ 13

2.1. Testu virzīta izstrāde, un ieguvumi no tās 15

2.2. Pārpratumi TVI īstenošanā, un to sekas 16

2.3. Izstrādes metodes atvasināšana no TVI 18

2.3.1. Uzvedības virzīta izstrāde 18

3. PROGRAMMINŽENIERIJA AR ZINĀTNISKU PIEEJU JEB HIPOTĒŽU VIRZĪTA IZSTRĀDE 24

3.1. Izstrādes metodes, ko ieteicams ievērot, lai veiksmīgi realizētu HVI 24

3.1.1. Nepārtrauktā piegāde un nepārtrauktā izvietošana 25

3.1.2. A/B testēšana 27

3.1.3. Iespēju pārslēgšana 28

3.1.4. Pārraudzīšana 29

3.2. Hipotēžu virzītas izstrādes norise 29

3.1. Hipotēžu formāts 31

4. PRAKTISKĀ DAĻA – MĒĢINĀJUMS ĪSTENOT HVI IT PROJEKTĀ 33

4.1. Pētāmās sistēmas apraksts 34

4.1.1. Pētāmās sistēmas uzdevumi 34

4.1.2. Izmantotās tehnoloģijas un sistēmas arhitektūra 35

4.1.3. Daži pētāmās sistēmas raksturlielumi 36

4.2. Iemesli, kāpēc pētāmās sistēmas izstrādē neizmanto TVI un UVI 37

4.2.1. Iemesli neizmantot TVI 37

4.2.2. Iemesli neizmantot UVI 37
4.3. HVI priekšnosacījumu izpilde pētāmajā sistēmā 38
4.3.1. Nepārtrauktā integrācija un nepārtrauktā piegāde 38
4.3.2. A/B testēšanas alternatīva 39
4.3.3. Iespēju pārslēgšana 41
4.3.4. Pārraudzīšana 42
4.4. Piemērs jaunas iespējas izstrādei, ievērojot HVI 43
4.4.1. Hipotēzes formulēšana 43
4.4.2. Izstrāde un izlaišana 44
4.4.3. Hipotēzes pārbaude 45
4.5. HVI un tās priekšnosacījumu īstenošanas rezultāti pētāmajā projektā 46
4.5.1. Ieguvumi projektā no nepārtrauktās integrācijas 47
4.5.2. Ieguvumi projektā no nepārtrauktās piegādes 47
4.5.3. Ieguvumi projektā no iespēju pārslēgšanas 47
4.5.4. Ieguvumi projektā no HVI 48
REZULTĀTI 49
IZMANTOTĀ LITERATŪRA UN AVOTI 50
PIELIKUMI 53

APZĪMĒJUMU SARAKSTS

Apzīmējums	Skaidrojums
TVI	Testu virzīta izstrāde. *Test Driven Development*. (angļu val.)
UVI	Uzvedības virzīta izstrāde. *Behaviour Driven Development*. (angļu val.)
HVI	Hipotēžu virzīta izstrāde. *Hypothesis Driven Development*. (angļu val.)

IEVADS

Strādājot IT firmā, kas apkalpo dažādu sfēru pasūtītājus un kurā cilvēki nodarbināti dažāda izmēra IT projektos, darba autore novērojusi, ka vairums projektu tiecas īstenot spējo programmatūras izstrādi. Tās realizācija projektu starpā notiek atšķirīgi. Vairums novēroto projektu priekšroku dod zināmiem spējās izstrādes ietvariem, tādiem kā *Scrum* vai SAFe, ir novēroti pat gadījumi, kad projekts nolemj izmantot pavisam specifisku risinājumu, apvienojot idejas no spējās izstrādes un citām programminženierijas pieejām. Pēc autores domām, eksperimentēt ar programminženierijas pieejām un metodēm nav slikti. Tas paver iespēju uzlabot esošās metodes vai definēt jaunas, efektīvākas.

Laika gaitā bagātinot savu pieredzi IT nozarē, autore ir iepazinusies ar vairākām izstrādes pieejām, kuru priekšnosacījums ir spējās izstrādes īstenošana. Tādas ir, piemēram, testu virzīta izstrāde (TVI) un uzvedības virzīta izstrāde (UVI). Abas metodes paredz, ka izstrāde notiek cikliski, regulāri piegādājot pasūtītajam jaunu produkta versiju, kas ir tipiski spējajai izstrādei. Vēl savā pieredzē autore ir iepazinusies ar izstrādes, piegādes un izlaišanas tehnikām, kas piemērojamas gan spējajai izstrādei, gan tās ietvariem un patapinātajām pieejām, un par kuru derīgumu un vērtību autorei nācies pašai pārliecināties. Te būtu jāpiemin nepārtrauktā integrācija, nepārtrauktā piegāde un nepārtrauktā izvietošana, un varētu atrast vēl citas.

Pirms nedaudz vairāk kā pusgada autore pirmo reizi mūžā sastapās ar apzīmējumu *Hypothesis Driven Development* (angļu val.). Latviskojot tā būtu "hipotēžu virzīta izstrāde" (HVI). Autorei ātri radās interese par iepriekš nedzirdētu un neizmantotu izstrādes pieeju. Radās jautājumi saistībā ar to. Kā to īstenot? Kāds no tās ir ieguvums izstrādātājiem un pasūtītājam? Vai šī metode ir kādā veidā saistīta ar TVI vai UVI? Pēdējam jautājumam pamatā ir metodes nosaukuma atbilstība "X virzīta izstrāde" šablonam.

Autore nolēma izpētīt teorijas un informatīvos materiālus, kas pieejami par HVI un, ja rastos iespēja, pielietot šo metodi projektā, lai to izpētītu arī praksē. Vispirms, nācās saprast, kā HVI tiek īstenota. Internetā pieejamie materiāli un literatūra aprakstīja HVI norisi ļoti kodolīgi. Izdevās atrast aprakstus, kuros minēts, ka HVI pamatā ir cikliska eksperimenta atkārtošana. Citā avotā HVI jēdziens tika minēts līdzās citām izstrādes merodēm, kā A/B testēšana un nepārtrauktā piegāde. Autore nolēma, ka pilnvērtīgai izpratnei par HVI nepieciešams noskaidrot, ar kādām izstrādes metodēm tā visciešāk saistīta. Rezultātā autore noteica četras metodes, kuras, pēc autores domām, vēlams īstenot projektā, lai izdotos veiksmīgi realizēt HVI. Pētot, vai HVI ir saistīta ar TVI un UVI un, ja ir, tad, kādā veidā, tika lasīts liels apjoms tiešsaistes materiālu, kas aptvēra forumus, blogus un publikācijas, kā arī tika izmantoti literatūras avoti un video materiāli. Pirmkārt, autorei nācās izprast TVI un UVI atsevišķi no

HVI. Otrkārt, secināt, cik lielā mērā šīs trīs metodes savā starpā ir saistītas. Darba tapšanas gaitā autore konsultējās ar kolēģiem ar mērķi veikt viedokļu apmaiņu par darba tēmu un rast atbildes uz darba gaitā radušamies jautājumiem.

Gatavojoties darba praktiskajai daļai, autore apjautāja vairākus projektus, cerībā atrast tādu, kas piekristu izmēģināt jaunu izstrādes pieeju. Viens no apjautātajiem projektiem piekrita piedalīties pētījumā. Projekta komanda tika iepazīstināta ar autores atrastajiem teorijas materiāliem par HVI. Tika kopīgi secināts, kurus no HVI priekšnosacījumiem projekts spēj apmierināt un kurus ne. Eksperimentāla HVI izmantošana pētāmajā projektā norisinājās no š.g. februāra vidus līdz aprīļa beigām. Tuvojoties darba izstrādes beigām, autore apspriedās ar projekta komandu, lai uzzinātu izstrādātāju atsauksmes un viedokļus par HVI un tas noderīgumu projektā.

Darba saturs ir sakārtots četrās nodaļās. Pirmajā nodaļā tiek aprakstīta nepieciešamība pēc biežas izmaiņu integrēšanas kopējā komandas darbā, ja tiek īstenota spējā izstrāde. Tiek aprakstīta nepārtrauktās integrācijas metode un paskaidrots, kā tā ir nozīmīgs solis produkta kvalitātes nodrošināšanā un uzlabošanā. Otrā nodaļa veltīta TVI un UVI teorētiskās izpētes rezultātiem. Nodaļā ir aprakstīts, kā šīs metodes tiek īstenotas, kādi ir ieguvumi no to izmantošanas, un ar kādām problēmām var nākties saskarties to īstenošanā. Trešajā nodaļā pastāstīts tas, ko izdevās atrast par HVI procesiem, tiek aprakstīti autores identificētie priekšnosacījumi, kurus ieteiktu īstenot, pirms izmantot HVI. Visbeidzot, ceturtā nodaļa ir veltīta pētījuma praktiskajai daļai. Tiek aprakstīts pētāmais projekts un sistēma, ko tajā izstrādā, HVI priekšnosacījumu izpilde pētāmajā projektā, piemērs vienas izstrādes hipotēzes pārbaudīšanai un praktiskā pētījuma kopsavilkums ar rezultātiem.

1. KODA IZMAIŅU VEIKŠANA, STRĀDĀJOT SPĒJĀS IZSTRĀDES KOMANDĀ

Spējā programmatūras izstrāde nodrošina to, ka pasūtītājam tiek īsā laikā, nepārtraukti piegādāta pasūtītā funkcionalitāte. Laiks starp piegādēm produkcijas vidē var būt, sakot no pāris nedēļām līdz pāris mēnešiem, lai gan priekšroka tiek dota īsākiem pārtraukumiem [1]. Ņemot vērā biežās piegādes, izstrādājamā produkta kodam pastāvīgi ir jābūt pietiekami augstas kvalitātes, pretējā gadījumā jaunas versijas instalācija produkcijas vidē var izraisīt lielu daudzumu problēmu. Nodrošināt šāda līmeņa kvalitāti nav viegls uzdevums.

1.1. Izmaiņu integrēšana un tās biežuma nozīme

Izstrādātāju komandas darbā nepieciešams nepārtraukti sekot tam, lai jaunas izmaiņas kodā nepasliktina kopējo kvalitāti. Tiek pieņemts, ka komanda izmanto kādu no versiju kontroles rīkiem, piemēram, Git. Versiju kontrole nodrošina ne tikai kopēju koda glabāšanas vietu visai komandai, bet arī visu kopējā kodā veikto izmaiņu vēsturi. Izmaiņu vēsture satur dažādu informāciju, sākot no izmaiņas veikšanas laika un tās autora un beidzot ar komentāriem par izmaiņas saturu un nepieciešamības iemeslu. Programmas kods parasti tiek glabāts attālināti, kā arī ir ielādējams izstrādātāju lokālai izmaiņu veikšanai. Kad izstrādātājs vēlas ar to strādāt un veikt tajā izmaiņas, tiek izveidota kopija uz izstrādātāja darbstacijas. Kad izstrādātājs lokāli ir veicis izmaiņu kodā un vēlas to integrēt komandas kopējā kodā, tas tiek darīts ar versiju kontroles rīku. Tiek veikta sapludināšana.

Tikai, testējot kodu, ir iespējams pateikt, vai pēc kārtējās izmaiņas integrēšanas kodā no izejas failiem ir iespējams iegūt izpildāmu kodu un vai tas strādā tā, kā to sagaida pasūtītajs. Pirms veikt sapludināšanu, katra izstrādātāja pienākums ir uzrakstīt savai izmaiņai testus un izpildīt tos uz savas darbstacijas. Tikai tad, ja testi tiek izieti bez kļūdām, jauno izmaiņu drīkst sapludināt ar pārējo kodu. Teorētiski, ar lokālām pārbaudēm vajadzētu pietikt, lai novērstu lielāko daļu kļūdu pirms tās nonāk kopējā koda repozitorijā. Tiesa, ikdienā nevar paļauties uz to, ka katrs izstrādātājs katru reizi, pirms integrēt jaunu izmaiņu, lokāli izpildīs testus, pārliecināsies, ka izmaiņa nekaitē produkta kopējai kvalitātei un salabos problēmas, ja tādas tomēr rodas. Kā arī, rodas jautājums, cik bieži integrēt izmaiņas kopējā kodā? Vai spējās izstrādes laikā izstrādātājam ir prātīgi nedēļu strādāt pie savām izmaiņām lokāli un tikai tad integrēt tās? Vai tādējādi nepalielinās iespējamo problēmu skaits, kuras var rasties, integrēšanas laikā? Piemēram, nesavietojamība ar citu izstrādātāju izmaiņām, no kuras varētu izvairīties, ja

integrēšana notiktu ātrāk un regulārāk. Novēlota integrēšana var palielināt izstrādes laiku, jo izstrādātājiem var nākties novērst lielu skaitu problēmu. Šādos gadījumos tiek tērēts ne tikai laiks un pasūtītāja nauda, bet arī komandas kopējais gars krietni noplok. Sarunās ar nozares pārstāvjiem gan darbā, gan publiskos pasākumos, kā konferences Latvijā un Eiropā, autore ir vairākkārt dzirdējusi kolēģu stāstus par šādiem gadījumiem. Notiek aktīva komunikācija komandas starpā, bet tā kā tā ir novēlota un situācija ir kritiska, komunikācija kļūst agresīva un palielina jau tā augsto stresa līmeni komandā. Šādu situāciju nozarē mēdz saukt par "integrēšanās elli". Lai novērstu problēmu rašanos un nevajadzīgu laika patēriņu to labošanai, ieteicams izmantot izstrādes metodi, kas paredz biežu, tas ir, vismaz reizi dienā, izmaiņu integrēšanu.

1.1.1. Nepārtrauktā integrācija

Praksē autorei ir nācies palīdzēt projektiem īstenot nepārtrauktās integrācijas metodi, kura paredz, ka:

- visi izstrādātāji vismaz reizi dienā integrē visas izmaiņas, ko ir veikuši kodā, kopējā repositorijā;
- pēc katras integrēšanas no kopējā repositorija koda tiek izveidots būvējums un visa tajā esošā funkcionalitāte tiek testēta, turklāt būvējuma veidošanai un testēšanai jānotiek automātiski.

Ir speciāli nepārtrauktajai integrācijai paredzēti rīki, piemēram, *Jenkins* [2], kas nodrošina procesu automatizāciju un rezultāta attēlošanu viegli saprotamā lietotāja saskarnē. Pēc autores domām, Jenkins lietotāja saskarnes ar pareizu konfigurāciju spēj atspoguļo integrācijas procesu pietiekami saprotami, lai to izmantotu procesa raksturošanai ar piemēru. Attēlā 1.1. redzams, kā izskatās šāda saskarne. Koda būvējumi sakārtoti cits virs cita, katra horizontālā rinda attēlo vienu būvējuma ķēdi. Kreisajā pusē, rindas sākumā, redzams būvējuma kārtas numurs. Nereti šis kārtas numurs figurē arī būvējuma versijas nosaukumā. Vertikālās ailes atspoguļo būvējuma ķēdes posmus, tie ietver sevī būvējumu, dažāda līmeņa testēšanu un citus, brīvi izvēlētus soļus, piemēram, kā atsevišķu soli mēdz izdalīt instalēšanu dažādās testa vidēs. Pirmā kolonna atspoguļo soli, kurā izejas faili tiek izgūti no repositorija un no tiem tiek izveidots izpildāms kods. Atkarībā no programmēšanas valodas, šis solis var ietvert arī zema līmeņa testēšanu. Piemēram, valodas *Java* gadījumā, ko izejas failiem tiktu kompilēti izpildāmi *.class* faili, kurus apvienotu JAR vai kāda cita formāta arhīvā. Uzreiz pēc kompilācijas tā paša soļa ietvaros

1.1.att. ***Jenkins*** **lietotāja saskarne** [31]

iegūtie moduļi tiktu notestēti ar vienībtestiem. Ja kompilācijas kļūdu nav un vienībtestēšanas rezultāti ir pozitīvi, tad pirmais solis ir pabeigts un saskarnē tas iekrāsojas zaļš. Veiksmīgi iziets ķēdes posms nozīme, ka var pāriet pie nākamā posma. Pāreja var būt vai nu pilnībā automatizēta, vai arī izstrādātājs var izpildīt nākamo soli no saskarnes, kad pats to vēlas. Ja kompilācija vai testēšana beidzas ar kļūdu, tad attiecīgais ķēdes posms iekrāsojas sarkans un būvējuma ķēde ir apstājusies. Neuzsāktie ķēdes soļi uzrādās gaiši zili.

Jenkins ir plašas iespējas integrēšanai ar citiem rīkiem, piemēram, e-pastu vai komandas kopējo tērzēšanas serveri. Ja visa komanda savā tērzēšanas kanālā redz paziņojumu, ka kārtējā būvējuma testi uzrāda negatīvu rezultātu, izstrādātāji var īsā laikā reaģēt uz problēmu un sākt to labot.

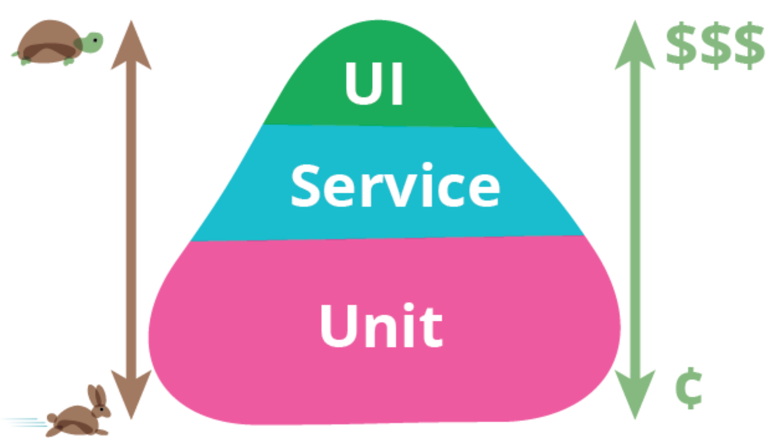
2. TESTU VIRZĪTA IZSTRĀDE UN TĀS LOMA CITU PROGRAMMATŪRAS IZSTRĀDES METOŽU ATTĪSTĪBĀ

Testēšana ir būtisks process kvalitātes nodrošināšanai. Kā panākt, ka programmas kodam ir testu kopa, kas tiek regulāri papildināta un atjaunināta? Kā nodrošināt augstu testu pārklājumu kodam?

Parasti testēšana tiek uztverta kā nepatīkamais, taču vajadzīgais darbs, kas tomēr ir jāpaveic, lai kodu uzskatītu par pabeigtu un gatavu. Testi tiek uzrakstīti pēc funkcionalitātes, ko tie testē. Testu uzrakstīšana sniedz pārliecību, ka uzrakstītā programma strādā tā, kā tai vajadzētu, taču tai no tās varētu būt arī cits ieguvums – tā varētu palīdzēt izstrādātājam strukturēt kodu tā, lai to ir viegli pārklāt ar testiem, respektīvi, lai izmaiņas testos aizņemtu ne vairāk laika, kā izmaiņas sistēmas funkcionalitātē. Vēl viena problēma, kas novērojama praksē - nevēloties veltīt tam daudz laika, priekšroka tiek dota augsta līmeņa, piemēram, lietotāja saskarnes, nevis vienībtestiem. [3] Lai pilnībā pārklātu kodu ar vienībtestiem, parasti ir jāuzraksta liels skaits testpiemēru, kas, savukārt, nozīmē, ka testu rakstīšanai jāparedz arī daudz laika. Tai pat laikā, ja programma jau ir pabeigta, tā kompilējas un to ir iespējams lietot, izstrādātājiem nav vēlmes tērēt laiku, rakstot zema līmeņa testus. Tā vietā priekšroka tiek dota augsta līmeņa testiem, kuri pārbauda konkrētus testa scenārijus. Piemēram, ja ir gatava tīmekļa lietotne, kas jānotestē, tai uzraksta automatizētus lietotāja saskarnes testus, kas simulē lietotāja darbības tīmekļa vietnē. Tādā veidā tiek pārbaudītas atsevišķas funkcionalitātes daļas, taču:

- nav garantijas, ka pilnīgi visa funkcionalitāte dara to, ko tai vajadzētu, ja nav testu, kas to pārbauda;
- gadījumā, ja tiek atrasta kļūda, zema līmeņa testu trūkums traucē ātri atrast problēmas cēloni.

Testēšanas piramīda [4] atspoguļo, kādai vajadzētu izskatīties programmas testu kopai, redzama attēlā 2.1. Shematiski tā parāda, ka zema līmeņa testiem, tas ir, vienībtestiem, ir jāviedo lielāko daļu no programmas testu kopas. Savukārt augsta līmeņa automatizētiem testiem ir jābūt mazā skaitā. Šāda sadalījuma mērķis ir samazināt laiku un resursu patēriņu testu rakstīšanai un uzturēšanai, kā arī izpildei. Zema līmeņa testus var salīdzinoši ātri uzrakstīt (ja koda struktūra to atļauj, piemēram, ja tiek izmantots kāds no koda šabloniem) un rakstīšanas

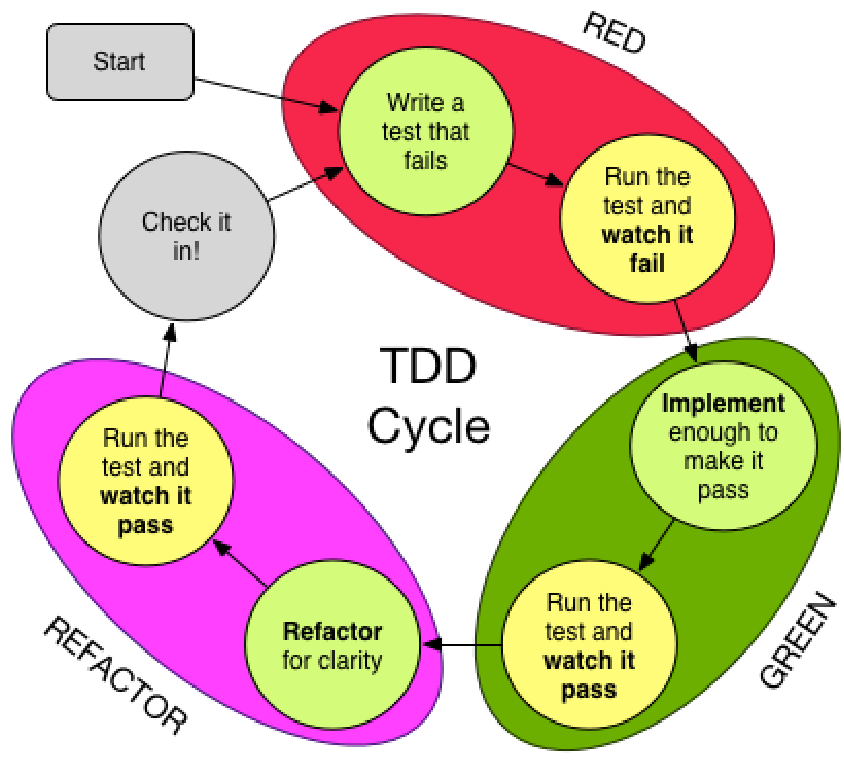


2.1.att. **Testēšanas piramīda** *[4]*

izmaksas ir zemas – tos raksta izstrādātāji paši un tam nav nepieciešams apgūt papildus tehnoloģijas. Turpretī augsta līmeņa testu rakstīšanai, uzturēšanai un automatizēšanai ir nepieciešams definēt testa scenārijus, testu rakstīšanai un izpildei tiek izmantotas tam paredzētas tehnoloģijas, un testu rakstīšana ir kopumā ilgāks process. Arī izpildes laiks šāda veida testiem ir ilgāks. Ja testu kopa pārsvarā sastāv no augsta līmeņa testiem, testu izpilde ir ilgāka nekā gadījumā, ja tā pārsvarā sastāvētu no zema līmeņa testiem. Nepārtrauktās integrācijas gadījumā tas nozīmētu, ka nav iespējams iegūt ātru apstiprinājumu tam, vai kārtējais būvējums ir derīgs vai nē, jo katru reizi būtu jāveic laikietilpīga testēšana. Piemēram, autorei ir nācies strādāt ar projektu, kurā pilna būvējuma ķēde jebkurai izmaiņai aizņem vismaz stundu, un iemesls tam bija liels skaits automatizētu lietotāja saskarnes testu. Šāda situācija ir pretrunā ar nepārtrauktās integrācijas pamatmērķi, proti, ātri noskaidrot, vai jaunākā koda versija ir izvietojama produkcijā vai nav.

2.1. Testu virzīta izstrāde, un ieguvumi no tās

Testu virzīta izstrāde ir programmatūras izstrādes filozofija, kas paredz jaunas sistēmas iespējas pievienošanai nepieciešamo izmaiņu veikšanu kodā sākt ar testu uzrakstīšanu, kuri spētu šīs izmaiņas pārbaudīt. [5] Kad ir zināms, ka nepieciešama jauna funkcionalitāte, ir jābūt skaidram, kas tai jādara. Kad tas ir skaidrs, var izsecināt, kā topošo funkcionalitāti testēt. Attēlā 2.2. parādīts, kā noris izstrāde saskaņā ar TVI. Vispirms izstrādātājs uzraksta testpiemēru, kas pārbauda topošo funkcionalitāti. Lai gan TVI kā metodoloģija attiecināma uz jebkāda veida testēšanu, parasti to piemin attiecībā uz vienībtestiem. Tests tiek izpildīts, lai pārliecinātos, ka



***2.2.att.* Izmaiņas veikšana kodā saskaņā ar TVI** [32]

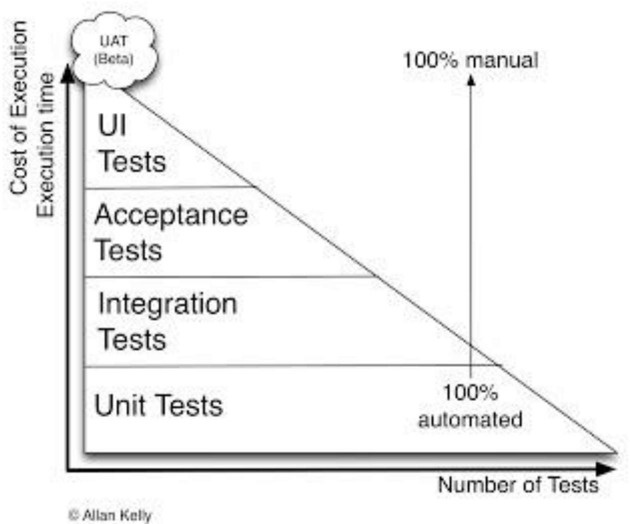
tas pareizi uzrakstīts. Ja tests uzrakstīts pareizi, tam jābeidzas ar kļūdu jeb, kā saka nozarē, tam "jābūt sarkanam", jo vēl nav funkcionalitātes, kas apmierinātu testpiemēra prasības. Kad izstrādātājs ir pārliecinājies, ka testpiemērs rezultātā dod kļūdu, tiek uzrakstīta funkcionalitāte, kas dara to, ko rezultātā sagaida tests. Vēlreiz izpilda testpiemēru, šoreiz tam jāizpildās bez kļūdām jeb, nozares žargonā runājot, "jābūt zaļam". Ja nepieciešams, jauno funkcionalitāti pārstrādā, lai uzlabotu tās lasāmību un struktūru. Ja tiek izmantots kāds no šabloniem, šis ir

īstais brīdis pārliecināties, ka jaunais kods tam atbilst, un uzlabot to, ja tā nav. Pēc pārstrādāšanas vēlreiz izpilda testpiemēru. Ja funkcionalitātē nekas nav sabojāts, tests joprojām būs “zaļš”. Ja tā nav, kodu turpina labot, kamēr tests izpildās pareizi. Beigu beigās jauno izmaiņu integrē. Ar šādu pieeju tiek panākts, ka visa funkcionalitāte ir testēta un dara to, ko no tās sagaida.

Ieguvumus no TVI izmantošanas ir saskatījuši gan izstrādātāji paši, gan pētnieki, kas ir organizējuši eksperimentus, lai pārbaudītu, cik noderīga ir TVI metode. Izstrādātāji atzīst, ka TVI palīdz tiem veidot pārdomātāku koda plānojumu un radīt programmatūru, kurā ir maz kļūdu. [6] [7] [8] *Microsoft*, *IBM* un Ziemeļkarolīnas Universitātes kopīgs pētījums atklāja, ka izstrādātāju komandas, kas seko TVI pieejai, raksta kodu ar 60% - 90% mazāku kļūdu blīvumu nekā komandas, kuras TVI neizmanto, un, pie tam, tērē izstrādei tikai 15% - 35% vairāk laika. [9] Līdzīgus rezultātus uzrāda arī citi pētījumi par TVI efektivitāti, kurus autorei izdevās atrast, TVI palīdz paaugstināt programmatūras kvalitāti, būtiski nepalielinot izstrādes kopējo laiku.

2.2. Pārpratumi TVI īstenošanā, un to sekas

TVI metode ir saņēmusi plašu atzinību, taču tā ir arī kritizēta. It īpaši pēdējo triju gadu laikā izstrādātāji aktīvi dalās ar pieredzi, kas atklāj, gan veiksmīgu, gan neveiksmīgu TVI izmantošanu. Ir izskanējuši pārmetumi, ka TVI izmantošana pat kaitē koda kvalitātei. Tiek pārmests, ka TVI izmantošanas rezultātā tiek pārmērīgi izmantoti koda aizbāžņi, kas padara testu uzturēšanu un izmaiņas tajos par laikietilpīgu procesu, turklāt var radīt draudus testu rezultātu ticamībai. [10] Daži atzīst, ka TVI izstrādes cikls nenodrošina gana kvalitatīvu kodu tā rezultātā. Kad ir panākts, ka testa rezultāts ir “zaļš”, daudzkārt izstrādātāji nepūlas kodu uzlabot pirms to integrēt, un tā rezultātā pasliktinās koda lasāmība un plānojums. Ar mērķi palielināt tādus rādītājus kā testu pārklājums, izpildes ātrums un pozitīvo rezultātu skaits, tiek pieņemti nekvalitatīvi lēmumi attiecībā uz koda plānojumu. Tiek rakstīts kods, ko vēlāk ir sarežģīti uzturēt vai izmainīt. Lielu uzmanību veltot vienībtestēšanai un automatizētai testēšanai kopumā, ir risks atstāt novārtā vai atteikties no manuālās jeb izpētes testēšanas. Attēlā 2.3. parādītā testēšanas piramīda testēšanas veidus iedala sīkāk nekā iepriekš minētā, attēlā 2.1. Šī piramīda iekļauj manuālo lietotāju testēšanu jeb beta-testēšanu kā pēdējo testēšanas soli ceļā uz produkcijas vidi. Attēlā tā shematiski parādīta kā mākonītis piramīdas virsotnē. Beta-testēšanas būtība ir nodot kārtējo versiju gala lietotājiem ar mērķi uzzināt viņu atsauksmes, pirms nogādāt



2.3.att. **Izmaiņas veikšana kodā saskaņā ar TVI [33]**

jauno versiju produkcijā. Ja izstrādātāju komanda, kas izmanto TVI, paļaujas uz to, ir risks, ka beta-testēšana tiek veikta nepietiekamā apjomā vai netiek veikta vispār. Beta-testēšana ir svarīgs atsauksmju avots un ir problēmas, kuras var atrast ar beta-testēšanas palīdzību, bet ne ar zemāka līmeņa testiem. [11]

Nenovērtēt visu veidu testu nozīmīgumu un paļauties, ka ar vienībtestiem pietiek, lai absolūti garantētu kvalitāti; kaitēt koda saprotamībai un pārskatāmībai testu pārklājuma vārdā, neizvērtējot, vai tas ir tā vērts; pārspīlēti izmantot koda aizbāžņus, piemēram, veidot metodes, kurās aizbāžņa metode izsauc cita aizbāžņa metodi – šīs ir kļūdas, ko ir pieļāvuši izstrādātāji, izmantojot TVI metodi. Autore uzskata, ka tās visas ir pamatā cilvēciskas kļūdas, kuru cēlonis ir nepareiza vai atšķirīga metodes interpretācija, un ka tās nenorāda uz trūkumiem pašā TVI filozofijā. Jāatzīst, ka ikdienā programmatūras izstrādes projektos rodas apstākļi, kas apgrūtina TVI īstenošanu vai, kas ir ne mazāk svarīgi, citu pārliecināšana par TVI nozīmi gala produktam. Tādi iemesli ir piemēram, skops tehniskais nodrošinājums testēšanas vajadzībām; īss laika periods, kas tiek atvēlēts izmaiņu veikšanai un nogādāšanai produkcijā; īss laika periods produkta izstrādei; apjomīgs repositorijs, kurā glabājas sen rakstīts, vidējas vai zemas kvalitātes, morāli novecojis kods; kolēģu un/vai pasūtītāja skepse attiecībā uz TVI.

Tā kā katrs projekts ir unikāls, projekta izstrādātāju rokās ir apsvērt un izlemt, kas viņu projektā ir darāms, lai TVI kalpotu par apliecinājumu gala produkta kvalitātei, lai TVI izmanošanas rezultātā nepazeminātos koda kvalitāte, lai tiktu veikta pienācīga apjoma testēšana visos līmeņos, lai testu rakstīšanai un testēšanai patērētais laiks un resursi attaisnotu kvalitātes nodrošinājumu, ko tie sniedz.

2.3. Izstrādes metodes atvasināšana no TVI

Kad izstrādātāji apgalvo, ka izmanto TVI, parasti tas nozīmē, ka saskaņā ar šo metodi tiek rakstīti vienībtesti. Kā tika minēts iepriekšējā nodaļā, nevar paļauties tikai uz viena līmeņa testiem, lai būtu pārliecība par produkta kvalitāti. Programma var būt paredzēta lietošanai ar citām programmām, piemēram, kā maksājumu sistēma. Tā var saņemt datus no citām programmām, apstrādāt tos un sūtīt atpakaļ. Cita veida programmas ir paredzētas lietotājiem. Tās tiek radītas kādam konkrētam mērķim, piemēram, komunikācijai, bankas kontu pārvaldībai, preču pārdošanai utt. Lietotājam ir skaidri zināmas darbības, ko tas vēlas veikt programmā vai un tas zina, kādu rezultātu sagaida. Ar vienībtestu palīdzību var garantēt, ka kods ir tehniski pareizi uzrakstīts, piemēram, ka atsevišķas metodes dara to, ko no tām sagaida. Taču, lai pārbaudītu programmas funkcionalitāti attiecībā pret lietotāja prasībām, nepieciešami augstāka līmeņa testi.

2.3.1. Uzvedības virzīta izstrāde

Lai pierakstītu lietotāja prasības pret sistēmu, spējās izstrādes projektos tiek rakstīti lietotāju stāsti. Tie parasti ir īsi, kodolīgi formulēti teikumi un satur ziņas par to, kādu rezultātu sagaida lietotājs, veicot specifisku darbību sistēmā. Vispārīgi pieņemts lietotāja stāsta formāts parādīts attēlā 2.4. Lietotāja stāsta teikums sastāv no trim daļām, attēlā katra no tām sākas jaunā

As a <type of user>,
I want <to perform some task>
so that I can <achieve some goal/benefit/value>.

2.4.att. **Vispārināts lietotāja stāsta formāts** [34]

rindā. Pirmajā rindā tiek pateikts, kāda veida lietotājs izmanto sistēmu. Otrajā -, kādu darbību lietotājs vēlas veikt sistēmā. Un trešajā rindā tiek pateikts, ko lietotājs ir vēlējies sasniegt, veicot minēto darbību. Attēlā 2.5. apskatīts lietotāja stāsta piemērs. Lietotāja stāsta nosaukums ir “Klients izņem naudu [no bankas automāta].” Pirmajā rindā tiek pateikts, ka lietotājs ir bankas

klients. Otrajā rindā teikts, ka klients vēlas izņemt skaidru naudu no bankas automāta. Trešā rinda atklāj, ka klienta mērķis naudas izņemšanai no automāta ir izvairīties no stāvēšanas rindā pie bankas kases, lai saņemtu naudu. Lietotāju stāstiem būtu jābūt pierakstītiem uz papīra,

User story title: Customer withdraws cash.

As a customer,
I want to withdraw cash from an ATM
So that I don't have to wait in line at the bank.

2.5.att. **Lietotāja stāsta piemērs** *[34]*

ieteicams izmantot neliela izmēra papīra kartītes, lai ieturētu īsu un kodolīgu formulējuma stilu.

Lai pārbaudītu, vai sistēma darbojas atbilstoši tam, kas prasīts lietotāja stāstā, katram lietotāja stāstam tiek formulēti akceptēšanas kritēriji. Tāpat kā lietotāju stāstiem, arī akceptēšanas kritērijiem ir noteikts formāts. Attēlā 2.5. redzamajam lietotāja stāstam atbilstoši akceptēšanas kritēriji parādīti attēlā 2.6. Līdzīgi kā lietotāja stāsts, arī akceptēšanas kritērijs

Acceptance Criterion 1:

Given that the account is creditworthy
And the card is valid
And the dispenser contains cash,
When the customer requests the cash
Then ensure the account is debited
And ensure cash is dispensed
And ensure the card is returned.

Acceptance Criterion 2:

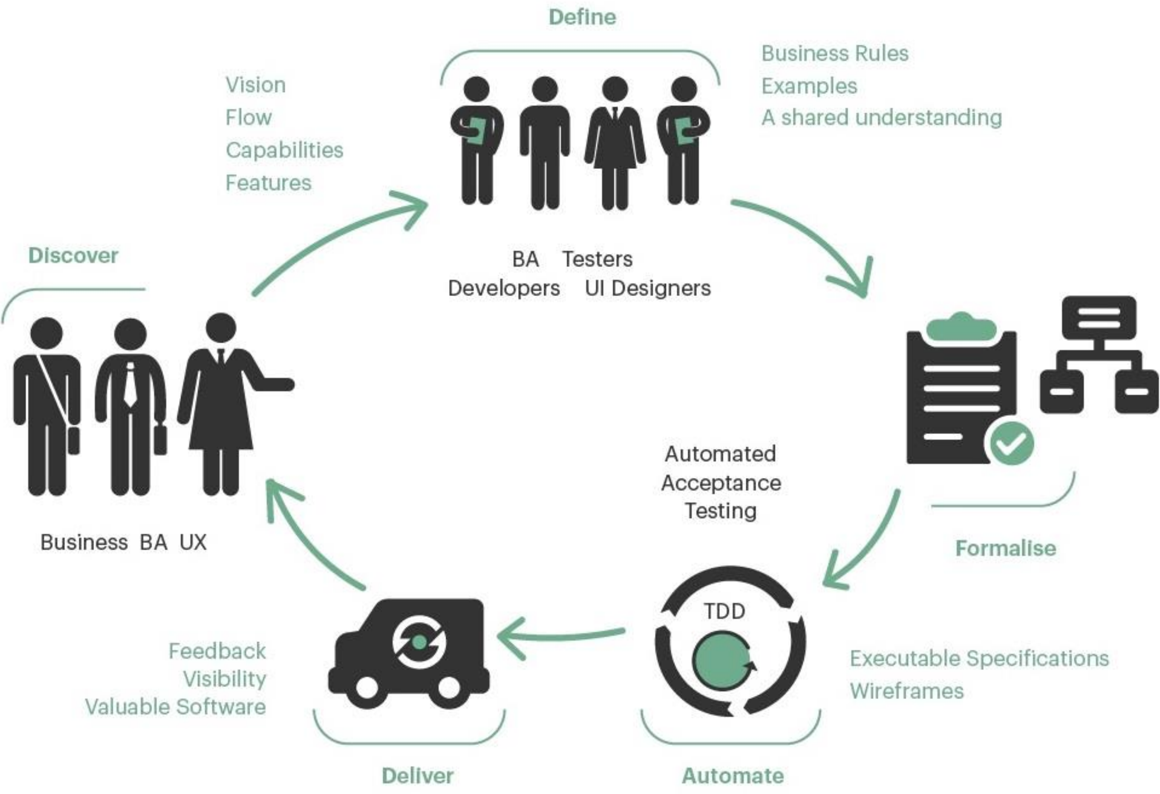
Given that the account is overdrawn
And the card is valid,
When the customer requests the cash
Then ensure the rejection message is displayed
And ensure cash is not dispensed.

2.6.att. **Lietotāja stāsta "Klients izņem naudu [no bankas automāta]" akceptēšanas kritēriji** *[34]*

sastāv no trim daļām. Attēlā redzams, ka šim lietotāja stāstam ir divi akceptēšanas kritēriji. Pirmais kritērijs apskata gadījumu, kad lietotājs ir tiesīgs izņemt naudu no sava bankas konta. Pirmās trīs rindas satur katra vienu nosacījumu, kuram ir jāizpildās, lai lietotājs varētu izņemt

naudu. Pirmajā rindā teikts, ka lietotāja kontā pietiek līdzekļu, otrajā, ka lietotāja bankas karte ir derīga, trešajā, ka bankas automātā ir skaidra nauda. Ceturtā rinda apraksta lietotāja darbību, kas tiek veikta. Šajā gadījumā, lietotājs pieprasa skaidras naudas saņemšanu. Pēdējās trīs rindas apraksta rezultātus, kurus būtu jāsasniedz lietotāja darbības rezultātā. Piektajā rindā teikts, ka no lietotāja konta tiek noskaitīta prasītā summa, sestajā, ka lietotājam tiek izsniegta skaidra nauda, un septītajā, ka lietotājam tiek atdota bankas karte. Otrais kritērijs apskata gadījumu, kad lietotāja kontā nepietiek līdzekļu naudas izņemšanai un lietotājam ir derīga bankas karte.

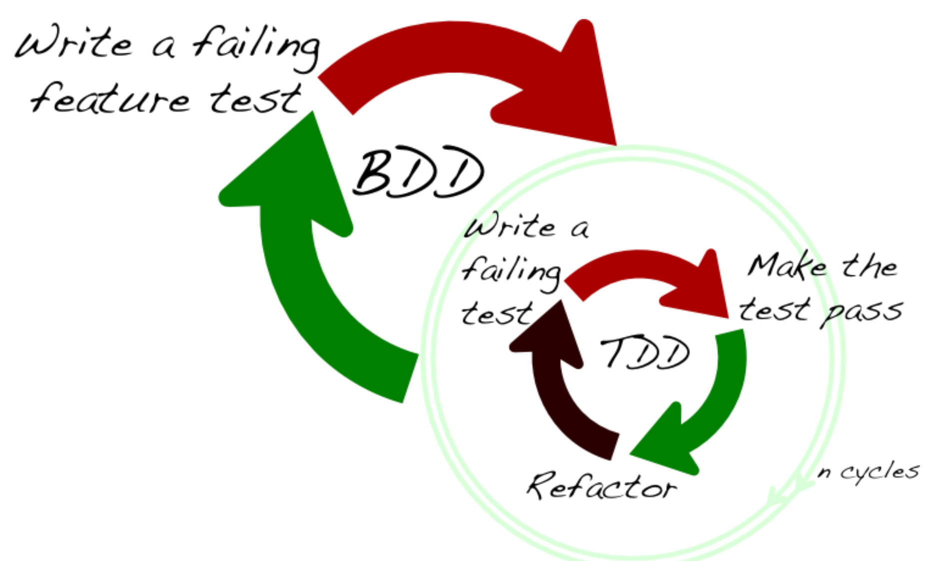
Lietotāja stāstu un akceptēšanas kritēriju ievākšana un formulēšana ir darbietilpīgs process. Ļoti svarīgi ir veltīt pienācīgu uzmanību diskusijām ar pasūtītāju, lai ievāktu pēc iespējas precīzākas prasības. Lietotāju stāstu rakstīšana nav viegls process, it īpaši ja tiek izstrādāta sarežģīta vai apjomīga sistēma. Attēls 2.7. parāda pilnu ciklu, ko iziet izmaiņa kodā,



2.7.att. **Uzvedības virzītas izstrādes procesa aktivitāšu cikls [https://dzone.com/articles/a-day-or-a-sprint-in-the-life-of-a-bdd-team]**

sākot no sarunām ar pasūtītāju un beidzot ar tās piegādi. Vispirms notiek sarunas starp pasūtītāju un piegādātāja puses biznesa analītiķi. Tiek izstrādāta vīzija, ko pasūtītājs galā vēlas redzēt piegādātu. Tiek ieplānots, kādas izmaiņas veicamas, lai pasūtītāja vīziju īstenotu. Tiek uzrakstīti lietotāja stāsti un definēti to akceptēšanas kritēriji. Kad visa vajadzīgā informācija ir apkopota uz papīra vai dokumentu formā, nākamais solis ir automatizētu akcepttestu rakstīšana,

balstoties uz akceptēšanas kritērijiem. Tāpat kā TVI pieejā, uzvedības virzītā izstrādē automatizēti testi tiek rakstīti pirms izmaiņām kodā. Attēls 2.8. rāda, kā TVI cikls iekļaujas UVI. Vispirms tiek uzrakstīts akcepttests, to izpilda, lai saņemtu negatīvu rezultātu. Tālāk notiek nepieciešamās izmaiņas veikšana kodā, sekojot TVI cikla soļiem, kas sīkāk aprakstīti 2.1. nodaļā. Kad ir uzrakstīta vajadzīgā funkcionalitāte un zemāko līmeņu testi nesniedz negatīvu rezultātu, programmu pārbauda ar akcepttestu. Ja tā rezultāts ir "zaļš" jeb pozitīvs,



2.8.att. **TVI cikls, ietverts UVI ciklā** *[35]*

programmu var piegādāt tālāk manuālai testēšanai vai, atsevišķos gadījumos, pirmsprodukcijas vai produkcijas vidē. Ja akcepttests iziet ar negatīvu rezultātu, uzrakstītā funkcionalitāte tiek pārstrādāta, līdz tiek sasniegts pozitīvs rezultāts.

Līdzīgi kā strādājot ar TVI, arī UVI izmantošanā var pieņemt pareizus lēmumus, un var arī kļūdīties. Rezultātā projekts no UVI izmantošanas neiegūst labumu, vai tieši pretēji, UVI ievērošana var kaitēt projektam. Kopumā projektos ir novērotas vairākas bīstamas tendences.

a) Kļūdas izstrādes procesos labot nevis pārskatot procesus, bet gan paļaujoties uz to, ka papildus tehniskais nodrošinājums problēmas atrisinās un ieviešot projektā papildus tehnoloģijas. "Ir jāatmet uzskats, ka pietiek instalēt jaunu rīku, lai visas mūsu problēmas tiktu atrisinātas! [..] mēs esam apmāti ar rīkiem," tā 2013. gada *NDC* konferencē Londonā [12] savas uzstāšanās laikā ir teicis Gojko Adzic, pieredzējis konsultants IT sfērā [13]. Savas uzstāšanās laikā Adzic apstrīd vairākus mītus, saistītus ar UVI. Uzstāšanās video ieraksts ir pieejams tiešsaistē, video nosaukums ir "*BDD: Busting the myths*" [14]. Šī tendence viņa runā tika aprakstīta ar piemēru no reāla projekta. Pārmērīgas testēšanas rezultātā testu kopa izauga līdz

2000 automatizētiem testa piemēriem, kā rezultātā neliela koda izmaiņa aizņēma apmēram divas nedēļas.

b) Uzskatīt, pasūtītāji paši spēs uzrakstīt akcepttestus un rakstīs tos. Pieņēmums, ka pasūtītāja pārstāvji paši rakstīs akcepttestus ir populārākais mīts par UVI. Realitāte rāda, ka pasūtītājs šādā situācijā nesaprot, kāpēc tam būtu jāuzņemas papildu pienākumus projekta laikā un atstāj rūpes par testu kopu paliek projekta testētāju rokās. Nav arī teikts, ka pasūtītājam vajadzētu pašam iesaistīties izstrādes ciklā. UVI procesi paredz aktīvu komunikāciju starp projekta komandu un pasūtītāju, kurai jānodrošina pasūtītāja vēlmju piefiksēšana un izpilde [14].

c) Izpratnes trūkums par dažādu veidu testēšanu. Adzic vārdiem, "mēs saprotam, kas ir vienībtesti, bet mums joprojām bail no visa pārējā." Nesaprotot dažāda veida testu nozīmi, tiek rakstīti augsta līmeņa testpiemēri, kuru uzdevums nav saprotams, jo tie reizē veic vairāku veidu testēšanu – veiktspējas, drošības, integrācijas testēšanu – un ir pārak apjomīgi. Tā dēļ testu izpildes laikā tiek nevajadzīgi pārslogota izpildes vide un izpildes ātrums palielinās [14].

d) Nepilnīga UVI procesu īstenošana, ko radījis izpratnes trūkums par UVI filozofiju. Šajā gadījumā problēma ir tajā, ka projekta komandas uztverē UVI tiek vienādota ar testēšanu. Rodas maldīgs uzskats, ka ar UVI projektā jānodarbojas tikai testētājiem, un UVI metode zaudē savu jēgu [14].

Testēšanas tehnoloģijas, kas tiek izmantotas UVI projektos, var radīt maldinošu iespaidu, ka UVI ir tas pats, kas testēšana. Kā piemēru var minēt *Cucumber* [15] – rīku, ko plaši izmanto akcepttestēšanai, kā arī projektos, kuros īsteno vai cenšas īstenot UVI. *Cucumber* ir

```
# feature/hello_cucumber.feature
Feature: Hello Cucumber
As a product manager
I want our users to be greeted when they visit our site
So that they have a better experience

Scenario: User sees the welcome message
When I go to the homepage
Then I should see the welcome message
```

2.9.att. **Akcepttesta piemēra fragments, paredzēts izpildei ar *Cucumber*** [27]

implementēts 14 programmēšanas valodās, tai skaitā, *Java, Ruby, JavaScript, PHP, C++,* kā arī atbalsta sešus testēšanas ietvarus, tai skaitā *Selenium* un *Ruby on Rails*. Testpiemēri iekļauj sevī lietotāja stāstu (skat. att. 2.9.), kas ir pamatā testējamai izmaiņai, un arī akceptēšanas

kritērijus. Viegli ievērot, ka testpiemēra kodā to formulējums neatšķiras no tā, kāds tas ir uz papīra. Tieši šis faktors, pēc autores domām, ir galvenais cēlonis maldinošai pārliecībai, ka UVI aprobežojas ar testēšanu un ka pārējiem procesiem var nepievērst uzmanību.

Neskatoties uz negatīvu pieredzi UVI projektos un pāragri izdarīta secinājuma, ka UVI nav vērts izmantot, var atrast arī veiksmīgus piemērus šīs metodes izmantošanā. Piemēram, investīciju bankas projekts, kurā izstrādātāji ikdienas darbā saskārās ar daudziem sarežģījumiem un neskaidrībām attiecībā uz funkcionalitāti, kas tiem bija jāizstrādā. Pateicoties komukācijai, ko sevī ietvēra UVI procesi, un piemērotu testēšanas tehnoloģiju izvēlei, ieguvumu no UVI sajuta gan projekta komanda, gan pasūtītājs. Pasūtītāja puses biznesa analītiķiem uzlabojās sapratne par tehniskajām problēmām projektā, savukārt, izstrādātājiem bija skaidra izpratne par pasūtītāja vajadzībām [16]. Vēl viens veiksmīgu UVI gadījumu min G.Adzic savā runā [14]. Projektā akcepttesti tika rakstīti pirms funkcionalitātes, kā to nosaka TVI. Šī projekta īpatnība bija tas, ka tiklīdz testpiemērs deva pozitīvu rezultātu, tas vairs netika izpildīts. Un, pretēji tam, kā to varētu gaidīt, ja pārstāj darbināt testus, regresijas gadījumu skaits bija niecīgs, to praktiski nebija. Zemā regresija tika panākta ar speciālu metodi. Tiklīdz kārtējā izmaiņa veiksmīgi izgāja akcepttestu, tika formulēta hipotēze par to, kādu iespaidu tai jāatstāj produkcijas vidē. Ja hipotēze pierādījās, tika secināts, ka izmaiņa veikta pareizi. Un ja tā, projektā uzskatīja, ka atkārtoti to testēt nav nepieciešams.

3. PROGRAMMINŽENIERIJA AR ZINĀTNISKU PIEEJU JEB HIPOTĒŽU VIRZĪTA IZSTRĀDE

Eksperimenta mērķis ir pārbaudīt pieņēmumu, sauktu arī par hipotēzi, kas izvirzīts saistībā ar pētāmo problēmu. Gatavojoties eksperimentam, tiek formulēta hipotēze. Parasti hipotēze tiek izvirzīta, balsoties uz iepriekšējiem novērojumiem vai ziņām par pētāmo problēmu. Kā arī, gatavojoties tiek izvēlēts neatkarīgais lielums jeb mainīgais, kura vērtība eksperimenta laikā tiek mainīta, un definēti atkarīgie mainīgie, kuru vērtība ir saistīta ar neatkarīgā mainīgā vērtību, tā var mainīties un tiek novērota eksperimenta gaitā. Eksperimenta laikā veiktie novērojumi tiek piefiksēti un analizēti. Balstoties uz veiktajiem novērojumiem, tiek izdarīts secinājums par to, vai hipotēze par pētāmo problēmu ir patiesa vai aplama.

Ir gadījumi, kad hipotēzes pārbaudīšanai nepieciešams eksperimentā iesaistīt cilvēkus un novērot to reakciju uz neatkarīgo mainīgo. Lai varētu labāk veikt salīdzinājumu starp cilvēku stāvokli pirms izmaiņām un pēc tām, eksperimenta objektu grupa tiek sadalīta divās daļās. Pirmā grupa, saukta par eksperimenta grupu, tiek pakļauta izmaiņām. Otra grupa tiek saukta par kontroles grupu un ar izmaiņām netiek pakļauta. Tiek ievākti dati par abām grupām un salīdzināti, lai secinātu, vai eksperimenta mainīgais iespaido cilvēku darbības vai ne.

3.1. Izstrādes metodes, ko ieteicams ievērot, lai veiksmīgi realizētu HVI

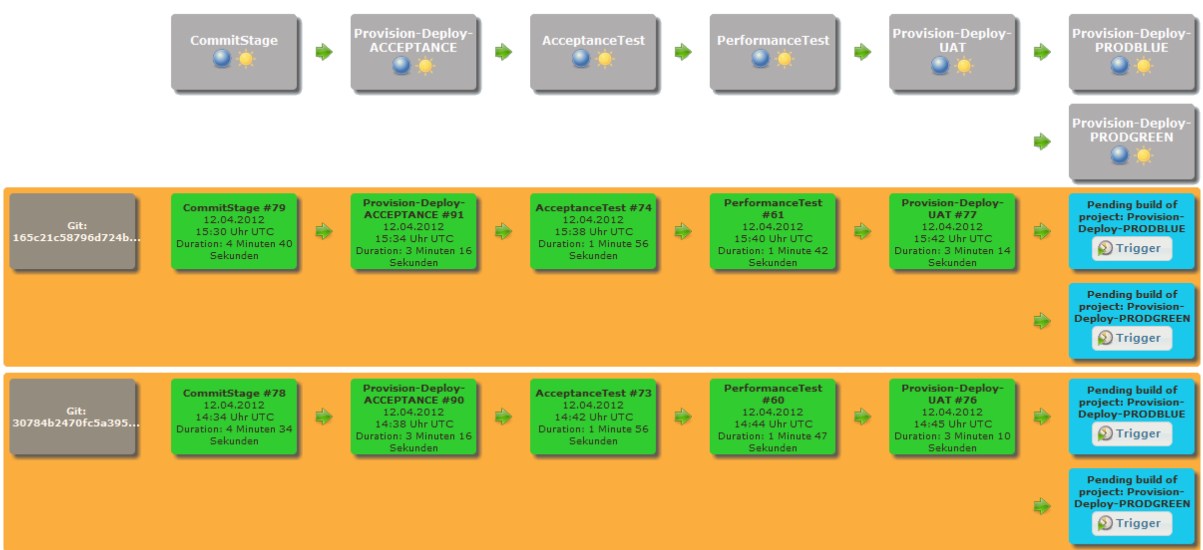
Kad jauna sistēmas funkcionalitāte ir piegādāta produkcijā, kā zināt, vai tā sasniedz mērķi? Tiek pieņemts, ka vairāku līmeņu testēšana ir nodrošinājusi koda tehnisku pareizību, programmas atbilstību pasūtītāja vēlmēm, optimālu veiktspēju un citu kvalitātes kritēriju izpildi. Kad kods ir nonācis produkcijas vidē, pēdējā pārbaude, ar ko programma saskaras, ir lietotāju reakcija. Realitāte rāda, ka izmaiņas sistēmā ar mērķi uzlabot tās darbību tikai trešdaļā gadījumu sniedz gaidīto ieguvumu. Tātad divas trešdaļas izmaiņu, kas tiek veiktas, lai uzlabotu sistēmas, ir veltīgas un neattaisno tām iztērētos resursus. [17] Tā kā projekta resursu parasti nepietiek visu iespējamo uzlabojumu veikšanai, ir jāizvēlas tie, kuri teorētiski nestu pasūtītājam maksimālu ieguvumu. Ja tikai trešā daļa izvēlēto uzlabojumu sevi attaisno, iespējams, ka sistēmā paliek ļoti daudz neīstenota potenciāla. Ir projekti un kompānijas, kas izmanto produkcijas vidi kontrolētu eksperimentu veikšanai. Šo eksperimentu mērķis ir noskaidrot lietotāju reakciju uz izmaiņām un uzlabojumiem sistēmā. Ja lietotāju reakcija rāda, ka tie vairāk atbalsta iepriekšējo sistēmas versiju, izmaiņas tiek atceltas un lietotāji var turpināt lietot versiju, kas tiem patika labāk. Lai spētu efektīvi un ar minimāliem riskiem veikt eksperimentus

produkcijas vidē, projekta komandai izstrāde un piegāde ir jāveic, ievērojot dažas metodes un aktivitātes. Tās ir nepārtrauktā piegāde, nepārtrauktā izvietošana, A/B testēšana, pārraudzīšana un iespēju pārslēgšana.

3.1.1. *Nepārtrauktā piegāde un nepārtrauktā izvietošana*

Nodaļā 1.1.1. tika aprakstīts, kā noris nepārtrauktās integrācijas process. Nepārtrauktās integrācijas process nodrošina buvējuma un testēšanas procesus, katru reizi, kad koda izmaiņa tiek integrēta kopējā repositorijā. Ar atbilstošu tehnoloģiju palīdzību šos procesus var vizualizēt, lai izstrādātājiem būtu jebkurā brīdī redzams, kura koda versija ir izmantojama produkcijā, vai arī kuras izmaiņas rezultātā testēšana dod negatīvu rezultātu.

Kad izstrādes procesi ir organizēti tā, lai katrs būvējumus, kas ir derīgi produkcijas videi, var tikt ātrā, drošā un uzticamā veidā nogādāts produkcijā, var apalvot, ka notiek nepārtrauktā

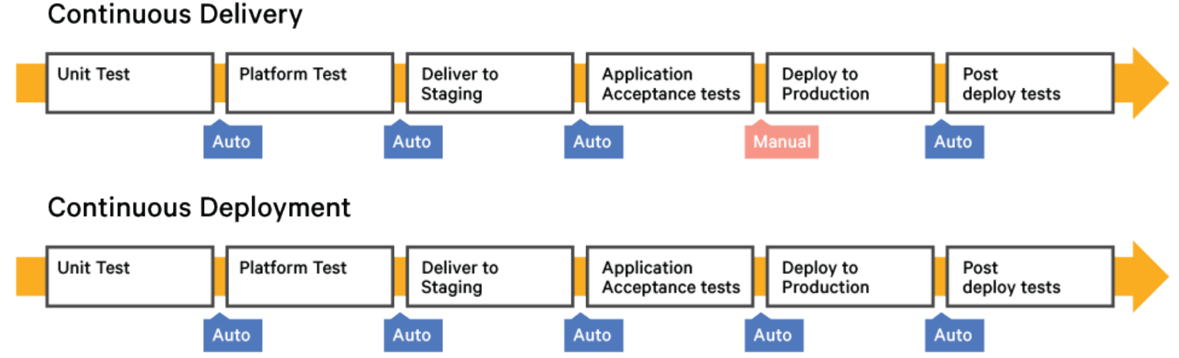


3.1.att. **Nepārtrauktās piegādes ķēde *Jenkins* lietotāja saskarnē** *[36]*

piegāde. Nepārtrauktā piegāde ir vairāku secīgu aktivitāšu apkopojums, kas sevī ietver nepārtraukto integrāciju, kā arī derīgo būvējumu izvietošanu artefaktu repositorijos un testēšanai paredzētās vidēs, un tā sniedz iespēju izvietot būvējumu arī produkcijas vidē. Runājot par tehnoloģisko realizāciju, nepārtrauktās piegādes procesam parasti tiek izmantots tas pats rīks, kas nepārtrauktajai integrācijai. Piemēram, *Jenkins* lieliski tiek ar to galā. Attēlā 3.1. redzama nepārtrauktās piegādes ķēde, kas sākas ar izmaiņas integrēšanu versiju kontroles repositorijā un beidzas ar izvietošanu produkcijas vidē. Attēlā redzamas divu būvējumu ķēdes, abos gadījumos būvējuma un testēšanas soļi ir bijuši veiksmīgi. Redzams arī, ka izvietošana

produkcijā nevienā no abiem gadījumiem nav notikusi, taču ir poga, kura jānospiež, lai izvietošanu uzsāktu. Nepārtrauktajā piegādē visi ķēdes soļi pirms izvietošanas produkcijā var būt automatizēti, taču šai izvietošanai jābūt manuāli izpildāmai darbībai. Vairums projektu piekrīt īstenot nepārtraukto piegādi, jo nepārtrauktā izvietošana pasūtītājam šķiet pārāk riskanta un nepiemērota viņu vajadzībām. Nepārtrauktā izvietošana no nepārtrauktās piegādes atšķiras ar to, ka nepārtrauktās izvietošanas gadījumā katrs būvējums, kas ir derīgs produkcijai, tiek automātiski tur izvietots. Izvietošana produkcijā var notikt desmitiem un pat simtiem reižu dienā. Attēlā 3.2. uzskatāmi parādīta šī atšķirība. Augšējā ķēdē izvietošana ir manuāls solis, savukārt, apakšējā – automātisks.

Pasūtītāji reti piekrīt īstenot nepārtraukto izvietošanu vairāku iemeslu dēļ. Pirmkārt,

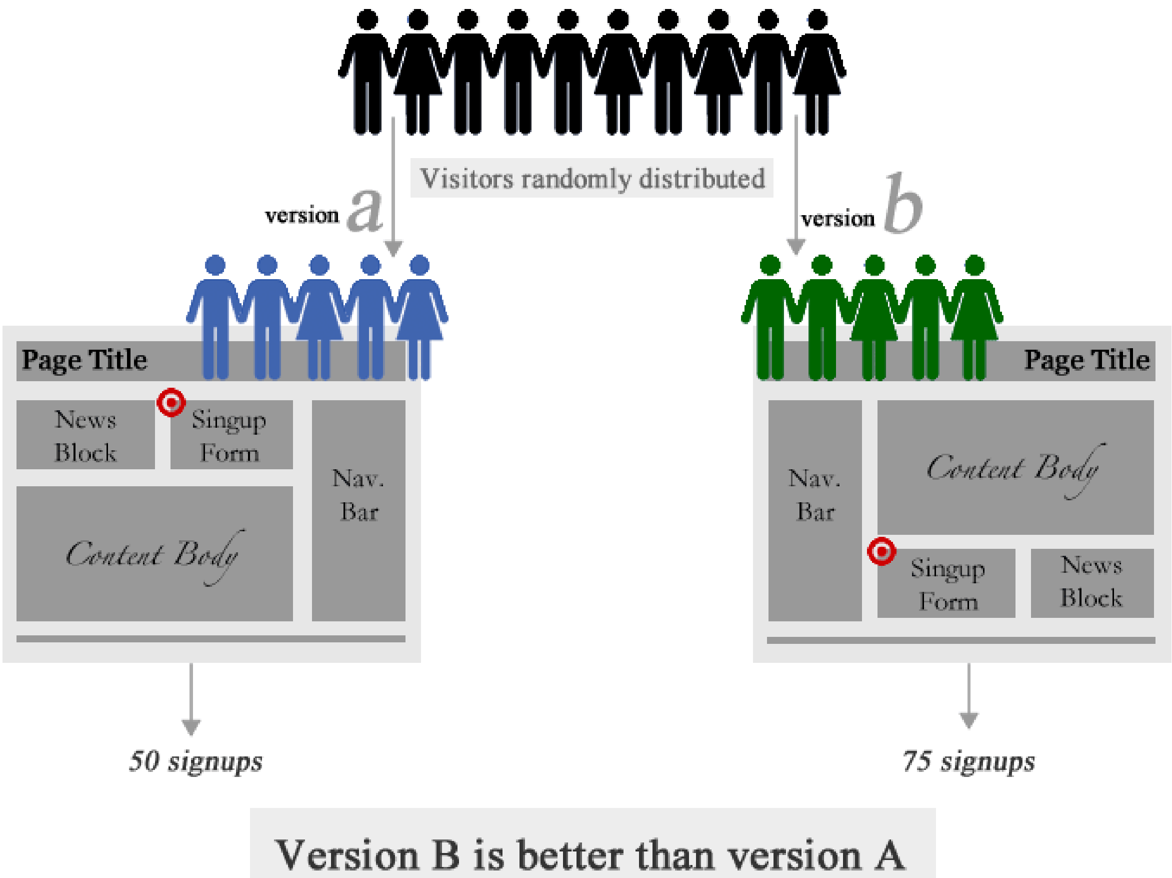


3.2.att. **Nepārtrauktās piegādes ķēde salīdzinājumā ar nepārtrauktās izvietošanas ķēdi** *[37]*

pasūtītājs vēlas kontrolēt, kādas izmaiņas tiek ietvertas kārtējā sistēmas versijā, kas nonāk produkcijā. Nepārtrauktā izvietošana atņem pasūtītājam kontroli pār to, cik izmaiņu un kurā brīdī nonāk produkcijā – tas notiek pastāvīgi. Otrkārt, pasūtītājs saskata risku tajā, ka produkcijā nepārtraukti notiek izmaiņas funkcionalitātē. 2.1.3. nodaļā minēts, kā abas minētās situācijas atrisināt tā, lai pasūtītājs paliek apmierināts un tai pat laikā tiek īstenota nepārtrauktā piegāde. Neskatoties uz atsevišķu pasūtītāju skepsi, nepārtraukto izvietošanu, līdz ar A/B testēšanu, iespēju pārslēgšanu un sistēmas vides pārraudzīšanu, realizē vairākas plaši pazīstamas kompānijas, tādas kā *Facebook*, *Google*, *Linkedin* un *Netflix* [17].

3.1.2. A/B testēšana

A/B testēšana pēc būtības ir divu dažādu sistēmas versiju salīdzināšana, lai noskaidrotu, kura ir labākā. Salīdzināšana notiek produkcijas vidē, vai vidē, kas pēc iespējas līdzīga produkcijai. Sistēmas lietotāji tiek sadalīti divās grupās. Katra no tām izmanto citu sistēmas versiju. Vienu no grupām var uztvert kā eksperimenta grupu un otru – kā kontrolgrupu. Tiklīdz ir pieejami dati par lietotāju uzvedību abās sistēmās, tie tiek salīdzināti, lai noskaidrotu lietotāju reakciju uz izmaiņu sistēmā. Ja reakcija ir pozitīva, izmaiņa tiek ieviesta sistēmā pilnvērtīgi un to saņem visi lietotāji. Ja reakcija ir negatīva, izmaiņa tiek atcelta un visi lietotāji turpina izmantot iepriekšējo sistēmas versiju. Attēlā 3.3. redzams piemērs A/B testēšanas



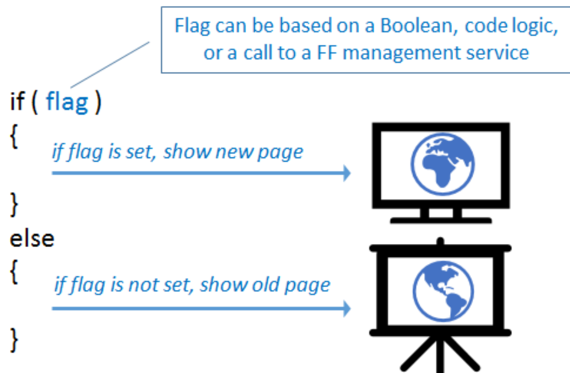
3.3.att. **A/B testēšana, mainot elementu izvietojumu mājaslapā** *[38]*

izmantojumam. Izstrādātāji eksperimentāli vēlas pārbaudīt, kāds elementu izvietojums portālā lietotājiem šķiet vizuāli patīkamāks un aicinošāks. Portāla lietotāji tiek pēc nejaušības principa sadalīti divās grupās. Puse lietotāju redz portālu tādu, kāds tas redzams pa kreisi, versijā A, pieteikšanās forma atrodas tuvāk portāla nosaukumam. Otra puse redz portāla versiju B, kas ir

pa labi, ar pieteikšanās formu tālāk no nosaukuma. Vienādā laika posmā versijā A un B tika reģistrēti attiecīgi 50 un 75 jauni lietotāji. Var izdarīt secinājumu, ka dizains B ir izrādījies labāks nekā dizains A.

3.1.3. Iespēju pārslēgšana

Nodaļā 2.1.1. tika pieminēts, ka pasūtītāji piesardzīgi izturas pret nepārtraukto izvietošanu. Viņiem zūd kontrole pār to, kāda funkcionalitāte konkrētā laika brīdī atrodas produkcijā, un nepārtrauktas izmaiņas produkcijā tiek uztvertas kā papildu risks. Taču, ja tiek īstenota tā sauktā "tumšā izlaišana" (*dark launch*, angļu val.), izmaiņu izvietošana produkcijā var notikt nepārtraukti, kamēr šo izmaiņu izlaišana jeb to padarīšana par lietotājiem pieejamām ir kontrolēts process. Pasūtītajs patur kontroli pār to, kāda funkcionalitāte ir pieejama sistēmas lietotājiem kurā brīdī, un, pateicoties nepārtrauktajai izvietošanai, tās izlaišana vai atsaukšana ir ātrs process.



3.4.att. **Vispārināts zarošanās konstrukcijas izmantojums iespēju pārslēgšanas realizācijā** *[39]*

3.4. attēls vispārināti atspoguļo iespēju pārslēgšanas realizāciju ar zarošanās konstrukciju. Slēdzis šajā gadījumā ir mainīgais ar vērtībām TRUE un FALSE. Ja tā vērtība ir TRUE, iespēja tiek ieslēgta un lietotājs saņem izmaiņas. Ja slēdža vērtība ir FALSE, lietotājs izmaiņas nesaņem.

Interneta veikalu platforma Etsy īsteno nepārtraukto izvietošanu savā produkcijas vidē un jauna funkcionalitātē tiek izlaista ar konfigurācijas slēdžu palīdzību. Sistēmas konfigurācija tiek izvietota produkcijā atsevišķi no sistēmas koda. Konfigurācijā tiek norādītas vērtības, kuras sistēma nolasa un kuras atbilst mainīgajiem sistēmas kodā. Atkarībā no konfigurācijas parametru vērtībām, tiek ieslēgtas vai izslēgtas iespējas sistēmā.

3.1.4. Pārraudzīšana

Sistēmas pārraudzīšana ir process, kas ietver sevī datu ievākšanu par sistēmu, datu apkopošanu un atspoguļošanu cilvēkam viegli uztveramā veidā, anomālu rādījumu atšķiršanu un lietotāja brīdināšanu par anomālijām. Pārraudzīšanas rīki, tādi kā *Zabbix* [18], *Sensu* [19] u.c. automatizē pārraudzīšanas procesu, lai tajā nav jāiesaistās izstrādātājiem.

Ar pārraudzīšanas rīku palīdzību ir iespējams ievākt dažāda veida datus. Piemēram, ja sistēmai ir liels skaits lietotāju vai ja tā paredzētai augstai noslodzei, tiek pārraudzīta sistēmas veiktspēja, mērot laiku, kurā sistēma atbild uz lietotāju pieprasījumiem. Pārraugot lietotāju pieprasījumu skaitu var izdarīt secinājumus par lietotāju aktivitāti sistēmā. Šādi pārraudzīšanas dati mēdz būt noderīgi A/B testēšanā, lai saprastu, kuras sistēmas iespējas lietotājiem šķiet parocīgākas un, līdz ar to, tiek aktīvāk izmantotas.

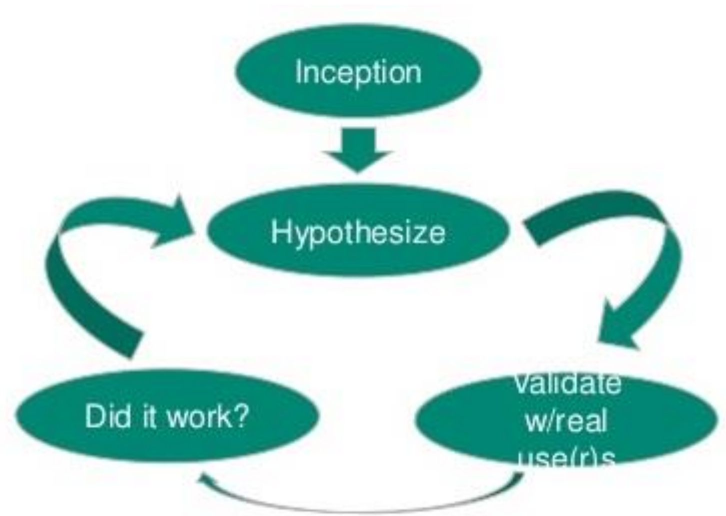
3.2. Hipotēžu virzītas izstrādes norise

Projektā, kas īsteno nodaļā 3.1. aprakstītās metodes, izstrādātājiem ir pieejama plaša informācija par sistēmu, ko tie izstrādā vai uztur. Viņiem ir pieejami pārraudzīšanas dati no produkcijas vides, kā arī no citām. Viņi spēj ātri un droši kontrolēt to, kuras no sistēmas iespējām ir pieejamas lietotājiem, izmantojot iespēju pārslēgšanu, un ar A/B testēšanas palīdzību viņi izdara secinājumus par to, kuras iespējas lietotājiem ir vajadzīgākas. Šādos apstākļos izstrādātāju rīcība ir pietiekami daudz informācijas un iespēju, lai komanda spētu izvirzīt pieņēmumus jeb hipotēzes par to, kā sistēmu iespaidos jaunu iespēju pievienošana vai izmaiņas esošajās, un pārbaudīt savas hipotēzes eksperimentos ar īstiem sistēmas lietotājiem. Citiem vārdiem, projekta komanda ir spējīga īstenot hipotēžu virzītu izstrādi.

Tā kā HVI ir salīdzinoši nesen sākusi gūt popularitāti IT nozares ekspertu vidū, tīmeklī par to ir atrodams mazāk materiālu nekā par TVI vai UVI. Atrodami atsevišķi raksti, pārsvarā to autori ir personības vai kompānijas, kas nodarbojas ar programminženierijas pieeju analīzi, filozofiju IT jomā, tie ir nozares eksperti, kuru viedoklī IT pasaulē ieklausās plaša auditorija. Tāda, piemēram, ir IT konsultēšanas firma *ThoughtWorks* [20] . *ThoughtWorks* ir sapulcējusi IT sfēras autorus, praktiķus un filozofus, kuru mērķis ir atrast veidus, kā uzlabot programminženierijas procesus un metodes. Šo cilvēku skaitā ietilpst IT bestselleru autori, tādi kā Jez Humble [21], kurš sarakstījis programminženierijas labās prakses apkopojumu "*Continuous Delivery*" [22], Martin Fowler, Rebecca Parsons [23] u.c.

2014. gadā *ThoughtWorks* autors Barry O'Reilly ir publicējis aprakstu par to, kā notiek HVI. Ir atrodami arī dažus gadus agrāk publicēti materiāli par šo izstrādes metodi, piemēram,

[24], uz ko O'Reilly arī atsaucas savā rakstā. O'Reilly iezīmē paralēlismu starp zinātniskiem eksperimentiem, ko veicam, vēl būdami skolas solā, un programminženierijas produkta izstrādi projekta komandā. Saskaņā ar *HDD*, jebkura programmatūras produkta izstrādei, jaunas idejas īstenošanai vai pat organizatoriskai izmaiņai ir jāpieiet kā zinātniskam eksperimentam. [25]



3.5.att. **Hipotēžu virzītas izstrādes vispārināts process [https://www.slideshare.net/cote/better-ways-of-developing-software-or-coding-like-a-unicorn-government-edition]**

Tāpat kā TVI un UVI, arī HVI pamatā ir ciklisks process. 3.5. attēlā redzamā shēma vispārināti parāda HVI ciklu. Shēma nesatur informāciju par to, kādas metodes un tehnoloģijas būtu jāizmanto izstrādē, īstenojot HVI. Metode ir abstrakta, un tādēļ realizējama ar projektos ar atšķirīgu tehnoloģiju izmantojumu. Eksperimentu atkārto tik ilgi, kamēr tiek sasniegts gaidāmais rezultāts vai kamēr ideja pierāda sevi kā neīstenojamu. "Galvenais rezultāts, ko sniedz eksperimentālas pieejas izmantošana, ir mērāmi pierādījumi un mācīšanās," tā O'Reilly. Testēšanas uzdevums ir nodrošināt programmas koda kvalitāti; eksperimentēšanas mērķis ir mācīties no eksperimenta rezultāta. Ievērojot HVI pieeju, no izstrādes procesa labumu gūst ne tikai pasūtītājs, kas saņem kvalitatīvu kodu, bet arī izstrādātājs, bagātinot savas zināšanas un attīstot spēju paredzēt izstrādes gaitā sastopamās problēmas un veicamo izmaiņu rezultātus. Zinātniskās pieejas secīgi soļi, ko identificējis O'Reilly:

- veikt novērojumus;
- formulēt hipotēzi;
- izplānot eksperimentu, lai pārbaudītu hipotēzi;
- noteikt, kādi indikatori liecina par eksperimenta veiksmi;
- veikt eksperimentu;
- izvērtēt eksperimenta rezultātus;

- pieņemt vai noraidīt hipotēzi;
- ja nepieciešams, izvirzīt un pārbaudīt jaunu hipotēzi.

O’Reilly norāda, ka izstrādātājam pasūtītāja sfēra parasti ir pilnībā sveša, un ka darbu nezināmā vidē, kura tiek pakāpeniski apgūta un izpētīta, būtu jābalsta hipotēzēs, nevis jāveic pēc stringriem norādījumiem. Autoresprāt, patiesa ir O’Reilly aprakstītā situācija par to, kā “nododot komandai biznesa prasību komplektu, tiek uzturēts pavēļu izpildīšanai līdzīgs darba stils”. Šķiet, ka viens no HVI mērķiem ir sniegt plašāku lemšanas brīvību projekta komandai. Ja komandā ir apvienoti daudzfunkcionāli speciālisti ar plašām tehniskajām zināšanām, ļaujot arī viņiem piedalīties problēmas risinājumu meklēšanā un formulēšanā sniedz iespēju izmantot resursus, kas tiek izniekoti, ja komanda akli seko klienta instrukcijām. [25] Labs veids, kā sākt risinājuma meklēšanu un hipotēzes formulēšanu ir komandas “prāta vētra”. [24]

3.1. Hipotēžu formāts

Attēlos 2.5 un 2.6 redzams testa un prasību modelis, ko izmanto UVI. Šāds modelis veicina komunikāciju starp izstrādātājiem, testētājiem un pasūtītājiem, taču tas nav piemērots eksperimentālai pieejai. Eksperimentālai pieejai būtu nepieciešams aprakstīt precīzi, kādas darbības tiek veiktas, lai sasniegtu vajadzīgo rezultātu. Vajadzīgi konkrēti indikatori, kas norāda, vai plānotais rezultāts sasniegts vai nē. Soļiem un indikatoriem ir jābūt zināmiem pirms testu izpildes. Ja indikatori uzrāda hipotēzes izpildīšanos, projekta komanda var būt droša, ka viņu darbs turpinās vēlamajā virzienā. Attēlā 3.6. ir redzama hipotēzes shēma, ko iesaka izmantot HVI.

3.6. att. **HDD lietotāja stāsta shēma** [25]

Redzams, ka hipotēze sastāv no trim daļām un kopā tās veido aptuveni frāzi “Mēs ticam ka <funkcionalitātes izmaiņas> rezultātā notiks <iznākums>. Mēs zināsim, ka esam sasnieguši pozitīvu iznākumu, kad <mēs novērojam indikatoru>.” Funkcionalitāte tiek izstrādāta, lai pārbaudītu hipotēzi. Tā funkcionalitāte, kas tiek darbināta eksperimenta laikā, sniedz rezultātu. Rezultātam ir jābūt mērāmam, lai varētu saprast, vai hipotēze ir pierādījusies. Mērāms rezultāts varētu būt, piemēram, jaunu lietotāju reģistrāciju skaits portālā nedēļas laikā. Projekta komanda varētu izvirzīt hipotēzi, ka nomainot portālā izmantoto fona krāsu uz dzeltenu, portāls piesaistīs vairāk jaunu lietotāju. Lai pārbaudītu, vai hipotēze pierādās, nedēļu pēc izmaiņas produkcijas tiek mērīts jauno lietotāju reģistrāciju skaits. Ja pēc nedēļas iegūtais skaits ir būtiski lielāks par to, kāds tas vidēji ir citās nedēļās, tad hipotēze ir pierādīta un fona krāsa var palikt dzeltena. Eksperimentiem palīdz, ja testējamajam produktam jau ir apjomīga lietotāju bāze, kā tas ir piemēram *Google* produktiem, taču bieži tā nav. [25] 3.7. attēls rāda vēl vienu piemēru HVI “lietotāju stāstam”.

Business Story

We Believe That increasing the size of hotel images on the booking page

Will Result In improved customer engagement and conversion

We Will Know We Have Succeeded When we see a 5% increase in customers who review hotel images who then proceed to book in 48 hours.

3.7. att. **HDD lietotāju stāsta piemērs** [25]

Hipotēze kopumā: “**Mēs ticam**, ka viesnīcu attēlu palielināšana rezervācijas mājaslapā **rezultēsies** klientu piesaistes un apgrozījuma uzlabojumā. **Mēs zināsim, ka esam to panākuši**, kad redzēsim, ka par 5% piedaudzis to lietotāju skaits, kas apskatās attēlus un veic rezervāciju nākamo 48 stundu laikā.” Jāpiebilst, ka priekšnosacījums ir efektīvas monitoringa sistēmas klātbūtne vidē, kurā notiek eksperiments. Pretējā gadījumā ir problemātiski ievākt rezultātu mērījumus.

4. PRAKTISKĀ DAĻA – MĒĢINĀJUMS ĪSTENOT HVI IT PROJEKTĀ

Projekts, kas būtu vispiemērotākais HVI īstenošanai un kas, pēc autores domām, gūtu maksimālu ieguvumu no HVI izmantošanas, salīdzinājumā ar citiem projektiem, ir tāds, kurā izstrādā lietotājiem paredzētu sistēmu, piemēram sociālā tīmekļa portālu vai Interneta veikalu. Šāds projekts ir piemērots HVI vairāku iemeslu dēļ. Tālāk tiek paskaidrots sīkāk par šiem iemesliem, taču pirms tas tiek darīts, jāpiebilst, ka no projektiem, kurus autore apjautāja, aicinot piedalīties savā pētījumā, tikai viens projekts piekrita aicinājumam. Šis projekts izstrādā un uztur papildu funkcionalitāti elektronisko darījumu apstrādes sistēmai, kuru izmanto citas sistēmas. Sistēmas izmantošanas veids, pēc autores domām, ir jāpatur prātā, kad tiek apskatīta četru HVI priekšnosacījumu (skat. 3. nodaļu) izpilde. Var gadīties, ka ne visi nosacījumi ir izpildāmi vai arī kāds no tiem var būt aizstājams ar alternatīvu, derīgu kādas konkrētas sistēmas gadījumā. Saskaņā ar pasūtītaja un projekta lūgumu un drošības prasībām, konfidenciāla projekta informācija, tāda kā projekta nosaukums, izstrādājamās sistēmas un tās komponenšu nosaukumi, testēšanas un produkcijas vides IP adreses vai dati, kurus apstrādā darījumu sistēma, šajā darbā netiek atklāta. Projekta īstais nosaukums turpmāk darbā tiks aizstāts ar segvārdu Alfa.

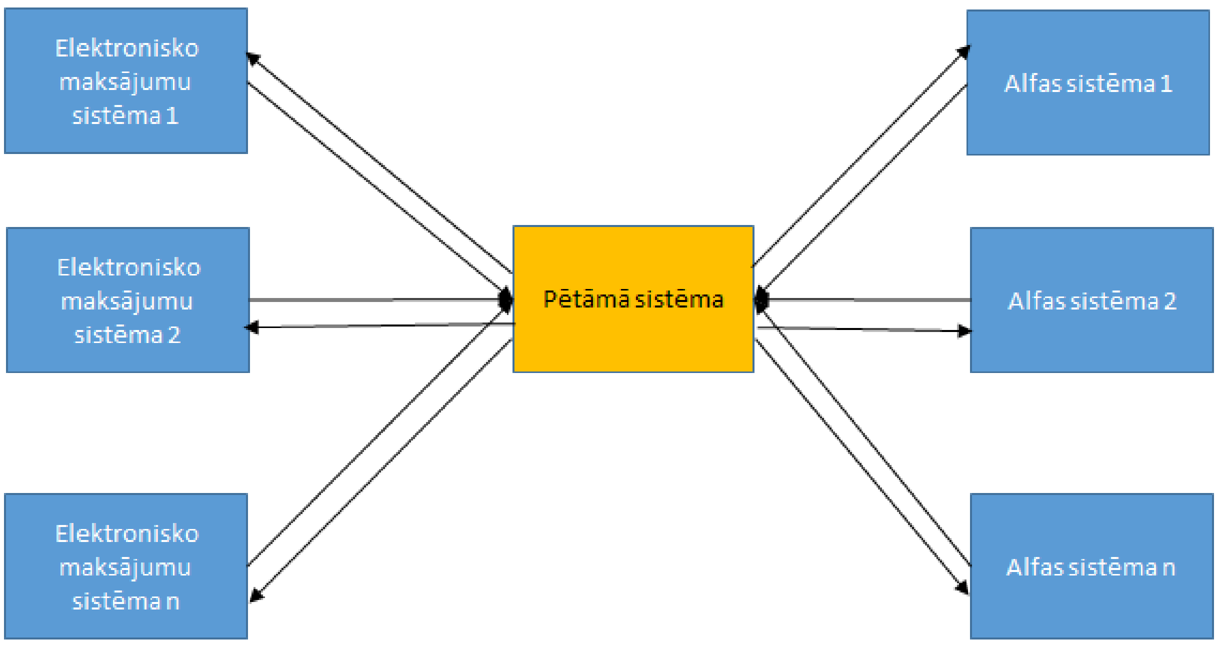
Iepriekš autore minēja, ka saskata iemeslu, kādēļ lietotājiem paredzēta sistēma ir izdevīgāka HVI realizēšanai par cita veida sistēmā. Šis iemesls ir saistīts ar A/B testēšanas procesu. Šī testēšana ir ļoti būtisks solis HVI ciklā, jo pēc būtības ietver sevī eksperimenta norisi (skat. 3. nodaļu). Sistēmas lietotāji, kuri var piedalīties projekta komandas eksperimentos, nonāk tiešā saskarē ar eksperimentālajām sistēmas iespējām. Pretēji tam, sistēmās, ar kurām lietotāji tieši nesaskaras un kuras tiek izmantotas ar citu sistēmu starpniecību, piemēram, Alfas elektronisko darījumu apstrādes sistēmā, eksperimenta funkcionalitāte nenonāk tiešā saskarē ar lietotāju. Šādā sistēmā apstrādājams darījums var tikt uzsākts citā sistēmā, kura nodrošina lietotāja saskarni. Tādējādi lietotājs darījumu sistēmu izmanto netiešā veidā. Taču, ja tiktu izvirzīta hipotēze par izmaiņām darījumu sistēmā un tai censtos veikt A/B testēšanu ar lietotājiem, kas to darbina caur citas sistēmas saskarni, visticamāk, rastos sarežģījumi testēšanas procesā. Turklāt, tiešas saskares trūkums starp lietotājiem un darījumu apstrādes sistēmu, kas rodas no sistēmu savstarpējas pakārtotības autorei liktu šaubīties par A/B testēšanas rezultātu noderīgumu hipotēzes pārbaudei. Vēl autore uzskata, ka būtu tehniski sarežģīti realizēt divu atšķirīgu darījumu sistēmas versiju pieejamību lietotājiem, turklāt tā, lai tās abas veiktu savu darbu vienlīdz uzticami. A/B testēšanas vietā tiek var izmantot *green-blue* izvietošanas metodi ar pirmsprodukcijas testiem, kurus izpilda izstrādātājs izvietošanas brīdī (skat. 4.3.2. nodaļu).

4.1. Pētāmās sistēmas apraksts

Balstoties, uz nozarē pieredzēto, jāsaka, ka Alfa ir vidēji liela izmēra projekts, kopskaitā tajā nodarbināti apmēram trīsdesmit cilvēki. Projekts sadalīts vairākās apakškomandās, no kurām katra ir atbildīga par atsevišķu sistēmu. Visu apakškomandu sistēmas komunicē savā starpā un ir saistītas, taču visu projekta resursu sadalīšana mazākās apakškomandās atvieglo darba organizēšanu. Autore pētīja darbu vienā no Alfas komandām. Pētāmajā komandā ietilpst pieci cilvēki, kas nodarbojas ar savas sistēmas uzturēšanu un jaunas funkcionalitātes izstrādi. Komandas sastāvā ir pieci izstrādātāji dažādās kvalifikācijas pakāpēs. Viņu darba uzdevumos ietilpst komunikācija ar pasūtītāju, jaunas funkcionalitātes realizācijas plānošana, izstrāde, testēšana, piegāde pasūtītājam un izvietošana produkcijas vidē, kā arī padziļināta pārraudzīšana problēmu gadījumā. Izstrādātāju komanda pati ir atbildīga par sistēmai nepieciešamās infrastruktūras konfigurēšanu un drīkst izvēlēties rīkus, kas tam piemērotāki.

4.1.1. Pētāmās sistēmas uzdevumi

Sistēma, ko izstrādā Alfa, nodarbojas ar elektronisko darījumu apstrādi. Konkrēti, tā kalpo kā elektronisko maksājumu vārteja, kura integrēta ar vairākiem elektronisko maksājumu



4.1.att. **Alfas sistēmu un trešo pušu elektronisko maksājumu sistēmu savstarpējās komunikācijas shēma**

pakalpojuma sniedzējiem un kuras mērķis ir nodrošināt vienotu formātu, kādā pieprasījumi tiek sūtīti atšķirīgām elektronisko maksājumu sistēmām un kurš būtu izmantojams ar visām integrētajām sistēmām. 4.1. attēlā vispārināti atspoguļota pieprasījumu plūsma starp pētāmo sistēmu, pārējām Alfas sistēmām un trešo pušu elektronisko maksājumu sistēmām. Pētāmā sistēma atvieglo citu Alfas sistēmu plānošanu un izstrādi, tādēļ, ka tā nodrošina, ka jāapstrādā tikai viena formāta pieprasījumi. Pretējā gadījumā, katrā no pārējām Alfas sistēmām nāktos iekodēt funkcionalitāti N dažāda formāta pieprasījumu apstrādei, kur N ir trešās puses integrāciju skaits. Trešās puses sistēmām savstarpēji atšķiras sūtāmo datu formāts; maksājuma apstrādes procesa plūsma, t.i., maksājuma apstrādei dažādās sistēmās var tikt izmantots dažāds skaits dažāda veida pieprasījumu; komunikācijas protokoli, piemēram, dažas no sistēmām izmanto REST, savukārt, citas – SOAP formāta pieprasījumus. Maksājumu vārtejas izmantošana nodrošina to, ka visas minētās sistēmu atšķirības tiek ņemtas vērā, taču neapgrūtina Alfas sistēmu izstrādi un darbību. Visi pieprasījumi, kas ienāk caur vārteju no trešās puses sistēmām, tiek apstrādāti vienādā veidā un attēloti par viena un tā paša veida objektu. Atkarībā no tā, no kuras maksājumu sistēmas pieprasījums nācis, pieprasījumu saturā var būt maznozīmīgas atšķirības, piemēram, obligāto pieprasījuma lauku skaits un tips.

4.1.2. Izmantotās tehnoloģijas un sistēmas arhitektūra

Iepriekš tika pieminēts, ka saņemtie pieprasījumi tiek attēloti par objektiem, kas norāda, ka pētāmās sistēmas izstrādē tiek izmantota OOP pieeja.

Pārsvarā tiek izmantota *Java* programmēšanas valoda, konkrēti, 8.x versijas. Tiek izmantots arī *Spring* programmēšanas ietvars, konkrēti *Spring Boot* implementācija. Spring Boot nodrošina izpildāmu programmatūras moduļu pakošanu tā, lai minimizētu pūles un laiku, kas jāvelta programmas sāknēšanai. atsauce Pētāmās sistēmas funkcionalitātes nodrošināšanai ir svarīgas dažas *Java* bibliotēkas, tādas kā *jersey-rx-client-java8* un *hystrix-core*. Šīs bibliotēkas tiek izmantotas attiecīgi komunikācijai ar trešās puses maksājumu sistēmām (Alfa izmanto pašu rakstītus klientus asinhronai komunikācijai ar šīm sistēmām) un paralēlās izpildes efektīvai realizācijai, ar Java pavedienu pārvaldību, kas novērš programmas pavedienu dīkstāvi vai ieslēgšanos. Būvējumu rīks, kas satur instrukcijas, lai izejas kodu būvētu, testētu un izvietotu ir *Maven*. Koda kvalitātes pārbaudei tiek izmantots *JaCoCo* spraudnis. JaCoCo analizē vienībtestu un integrācijas testu rezultātus, un viens no galvenajiem rādītajiem, ko tas aprēķina, ir koda pārklājums ar testiem. Sistēmas izstrādē izmanto bibliotēkas un rīkus, kas nodrošina dažādu līmeņu testēšanu. Vienībtesti tiek rakstīti ar *JUnit* ietvaru, savukārt augsta līmeņa testēšanai izmanto testu automatizācijas rīkus *Selenium* un *Gatling*. *Selenium* izpilda

testa scenārijus, kas pārbauda, vai funkcionalitāte darbojas tā, kā tai paredzēts. *Gatling* ir paredzēts sistēmas veiktspējas testēšanai, tas noslogo sistēmu ar dažādu pieprasījumu skaitu un fiksē atbildes laiku.

Testēšanas un pirmsprodukcijas vide ir veidota tā, lai būtu pēc iespējas līdzīga produkcijas videi. Attēls 1. pielikumā atspoguļo, kā ir plānota produkcijas vide, kurā darbojas pētāmā sistēma. Visa Alfas infrastruktūra ir izvietota mākonī. Darba izstrādes laikā tika uzsākta projekta infrastruktūras migrācija starp diviem mākoņpakalpojumu sniedzējiem, un tā joprojām ir progresā. Vēsturiskais pakalpojumu sniedzējs tiek aizstāts ar *Amazon Web Services (AWS)*, un līdz gada beigām ir mērķis pilnībā pāriet uz *AWS* izmantošanu. Infrastruktūras migrācija skar arī pētāmo sistēmu. Attēlā redzams, ka darba rakstīšanas brīdī produkcijas vide ir sadalīta starp diviem mākoņiem. Pa kreisi redzamā daļa atrodas vēsturiskā pakalpojumu sniedzēja datu centros, pa labi redzama *AWS* izvietotā daļa.

No AWS piedāvātājām iespējām pētāmās sistēmas arhitektūra izmanto mākoni ar privāto tīklu jeb *Virtual Private Cloud* (*VPC*). Ir izmantoti dažāda drošības līmeņa tīklu veidi: publiskais apakštīkls, attēlā *Public Subnet C*, un privātais apakštīkls, attēlā *Private Subnet A*. Infrastruktūras mezgli, kas atrodas publiskajā apakštīklā, ir sasniedzami no ārpuses un ir tie, ko aktīvi izmanto produkcijas sistēma. Privātajā apakštīklā esošie mezgli no ārpuses nav sasniedzami un tiek izmantoti testēšanas vajadzībām, lai pārbaudītu jaunu funkcionalitāti produkcijas vidē, pirms tā tiek izlaista. To, vai apakštīkls ir publiski pieejams, vai nav, nosaka sadalītāja konfigurācija. *AWS* daļā ir izmantoti divi sadalītāji, katram apakštīklam viens. Pētāmā sistēma ietver sevī vairāku veidu mezglus: datu bāzi klasteri, mezglus, uz kuriem izvietoti sistēmas koda moduļi un API, kā arī maršrutētājus un sadalītājus. Datu bāze realizēta kā MySQL klasteris ar diviem mezgliem.

4.1.3. Daži pētāmās sistēmas raksturlielumi

Vēl daži sistēmas pirmkoda raksturlielumi, kas palīdz rast kopēju priekšstatu par pētāmo sistēmu un nav minēti iepriekš, redzami tabulā 4.1.

4.1.tabula. **Pētāmās sistēmas *Java* izejas koda rakturlielumi**

Izejas koda raksturlielums	Vērtība
Rindu skaits	177000
Vidējais rindu skaits klasē	140
Klašu skaits	>1250

4.2. Iemesli, kāpēc pētāmās sistēmas izstrādē neizmanto TVI un UVI

Sākot projekta pētījumu, autore vēlējās radīt apstākļus, kuros vienlaikus tiek ievērota gan testu virzīta izstrāde, gan hipotēžu virzīta izstrāde. Autorei šķita, ka TVI izmantošana ir nepieciešama, lai veiksmīgi īstenotu HVI. Tomēr pētāmā projekta komandai bija savi iemesli, kādēļ tā atteicās no TVI vai UVI izmantošanas, un autore komandas lēmumam nepretojās. Jāpiebilst arī, ka pētījuma beigās autore mainīja savu viedokli un secināja, ka 3. nodaļā minētie priekšnosacījumi HVI realizācijai ir svarīgāki nekā TVI īstenošana reizē ar HVI.

4.2.1. Iemesli neizmantot TVI

Alfa projektā TVI nekad nav izmantota, tās vietā izvēlētā tradicionālā izstrādes pieeja, kurā testus raksta pēc tam, kad ir uzrakstīta funkcionalitāte, ko tiem jāpārbauda. Viens no iemesliem, kādēļ komanda atteicās izmantot TVI bija pieradums pie līdzšinējā izstrādes procesa. Izstrādātāju komandas vadītājs izteica uztraukumu, ka pāriešana uz vēl vienu jaunu izstrādes metodi būs izstrādātājiem pārāk apgrūtinoša un noslogojoša. Tika saskatīts risks, ka komandas darbs varētu būtiski palēnināties tā dēļ, un to projekts nebija gatavs pieņemt.

Pēc konsultācijas ar komandas vecāko programmētāju tika atklāts vēl viens iemesls, kura dēļ TVI izmantošana būtu apgrūtinoša projektam. Pašreizējais sistēmas kods un tā testu kopa ir uzrakstīti tā, ka vienkārša izmaiņa kodā izraisa neadekvāta mēroga izmaiņas testpiemēros. Tika atrasts gadījums, kad vienas metodes izmaiņai sistēmas kodā seko izmaiņas piecās testu klasēs. Pirms piemērot TVI, nāktos pārplānot projekta koda daļas un to testus, lai testa plānošana un izmaiņas tajā nebūtu saistīta ar izmaiņām citos testpiemēros. Apsverot nepieciešamos resursus, tika secināts, ka projekts šobrīd nevar to atļauties.

4.2.2. Iemesli neizmantot UVI

Pēc Alfas izstrādātāju uzskatiem, UVI izmantošana ir piemērota programmām ar lietotāja saskarni, tā kā metodei (un spējajai izstrādei kopumā) piesaistītais lietotāju stāstu un to akceptēšanas kritēriju formāts ietver sevī skatu uz sistēmu no lietotāja skatu punkta. Pēc komandas domām, UVI izmantotais testu formāts ir ļoti ērts, lai saprastu ko tieši pārbauda konkrēti testu piemēri un kādēļ varētu rasties negatīvs rezultāts.

Savukārt pētāmajā sistēmā komanda ir nolēmusi uzticēties esošajai testu kopai un formātam, lai nodrošinātu sistēmas koda kvalitāti, un neimplementēt jaunas vai papildu metodes. Esošā testu kopa ietver sevī piecu dažādu veidu testus, kas pārbauda sistēmu dažādos

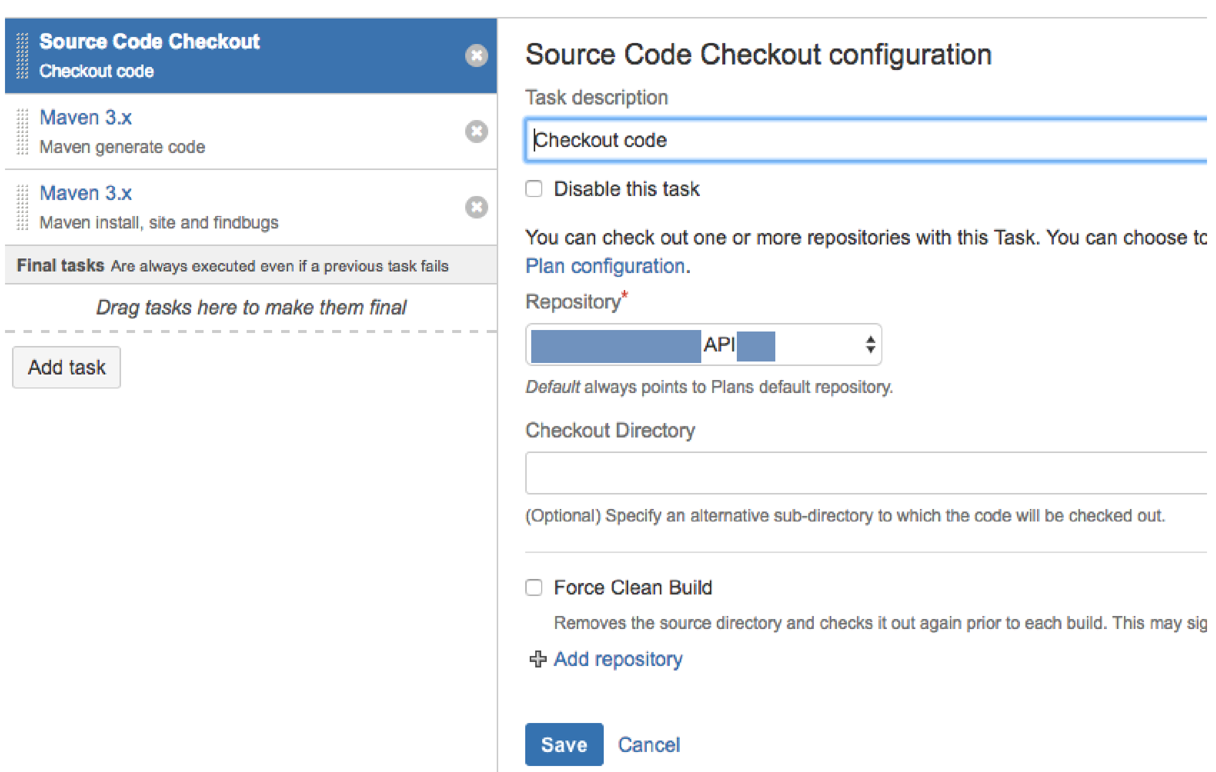
līmeņos: vienībtesti, integrācijas testi, veiktspējas testi, pieejas testi un pirmsprodukcijas testi. Ar vienībtestiem ir pārklāti vairāk kā 90% sistēmas pirmkoda, un pārējie testu veidu nodrošina tuvu 100% pārklājumu sistēmas iespējām.

4.3. HVI priekšnosacījumu izpilde pētāmajā sistēmā

HVI izpildes priekšnosacījumi, kas ir aprakstīti 3. nodaļā, pētāmajā sistēmā ir daļēji realizēti. Vienā no gadījumiem priekšnosacījumu nevar attiecināt uz pētāmo sistēmu, un ir izmantota alternatīva metode.

4.3.1. Nepārtrauktā integrācija un nepārtrauktā piegāde

Pētāmās sistēmas izstrādes process notiek saskaņā ar nepārtrauktās integrācijas principiem. Būvējuma un testēšanas procesu regulāri iziet gan galvenais koda zars, gan katra izstrādātāja individuālie zari. Pirms iepludināt savu lokālo koda versiju kopējā repozitorijā, jebkuram komandas izstrādātājam ir pienākums pārbaudīt, vai viņa kods strādā. Šim nolūkam



4.3.att. **Būvējuma soļa konfigurācija *Bamboo* lietotāja saskarnē**

izstrādātājam savā darbstacijā ir jāizpilda testi. Lokāli izpildām ir visi sistēmas vienībtesti un

integrācijas testi. Ar tiem pietiek, lai pārliecinātos, ka ir iespējams izveidot būvējumu un tas spēj sazināties ar sistēmā izmantoto datu bāzi. Ir pat iespējams pārbaudīt, kā lietotāja lokālā sistēmas versija strādā ar datiem, kas ir identiski produkcijas vides datiem. Šo iespēju nodrošina integrācijas testi, kas sarakstīti tā, lai varētu izmantot gan izstrādātāja lokālo datu bāzi, gan datu bāzi, kas atrodas pirmsprodukcijas vidē. Kad jauna versija tiek iepludināta kopējā kodā, uzreiz tiek automātiski veikts būvējums un testēšana. Katram būvējumam tiek veikta vienībtestēšana un integrācijas testēšana. Autores personīgā pieredze liecina, ka izstrādātāju komandās populārākais rīks nepārtrauktajai integrācijai ir Jenkins, kas jau tika pieminēts 1. nodaļā. Taču Alfa izmanto citu rīku. Iemesls tam ir pasūtītāja lūgums. Izmantotais rīks ir *Bamboo*, to izstrādā *Atlassian*, kas ir autori arī citām projektā izmantotām tehnoloģijām, tādām kā *JIRA* un *Confluence*. Rezultātā *Bamboo* ir viegli integrējams ar pārējiem projekta rīkiem, taču, salīdzinājumā ar *Jenkins*, kam priekšroku būtu devusi darba autore, tam ir savi trūkumi. Piemēram, Bamboo trūkst būvējuma ķēdes vizualizācijas veidā, kā to piedāvā Jenkins (skat. 1.1. attēlu). Pēc autores domām, intuitīvi un ātri uztverama un saprotama saskarne ir ļoti būtisks izstrādātāju rīku aspekts. Attēlā 4.3. parādīts, kā būvējuma process tiek konfigurēts *Bamboo* lietotāja saskarnē. Kreisajā pusē ir definēti trīs būvējuma soļi. Vispirms tiek ielādēti izejas koda faili no versiju kontroles, tas ir *Source Code Checkout* solis. Nākamajā solī tiek kompilēti izpildāmi koda moduļi, un pēdējā solī tiek izpildīti vienībtesti. Veiksmīgas testēšanas rezultātā tiek iegūts izpildāms .jar fails, ko tālāk testēt ar integrācijas testiem. Pēc integrācijas testiem, Bamboo izvieto kodu testēšanas vidē, kur pret to izpilda testus, kas pārbauda, kā sistēma strādā ar trešās puses integrācijām un veiktspējas testus. Pētāmās sistēmas izstrādātāji lielā mērā paļaujas uz vienīb- un integrācijas testiem, lai pārliecinātos, ka kods ir gatavs produkcijas videi. Augstāko līmeņu testi sniedz papildus drošību tam, ka izlaišana būs veiksmīga.

Koda izlaišana produkcijā notiek reizi nedēļā. Izstrādātāji paši nosaka, kad to darīt. Pētāmās sistēmas gadījumā, koda izlaišana produkcijā notiek reizē ar tā izvietošanu. Nepārtrauktā izvietošana nav īstenota tehnisku apstākļu dēļ. Infrastruktūras pārvaldības process nav piemērots tam, lai bieži veiktu izvietošanu. Katru reizi, veicot izvietošanu, infrastruktūras mezgli, uz kuriem atrodas sistēmas API, tiek veidoti no jauna. Process ir pārāk laikietilpīgs, lai tiktu veikts vairākas reizes dienā, netraucējot sistēmas darbībai produkcijā.

4.3.2. A/B testēšanas alternatīva

A/B testēšana pētāmajai sistēmai nav realizējama formā, kāda aprakstīta 3. nodaļā, taču ir piemērota alternatīva testēšanas metode. 3. nodaļā aprakstītais formāts nav iespējams divu iemeslu dēļ. Pirmkārt, kā autore jau izteicās nodaļas sākumā, ir “tehniski sarežģīti realizēt divu

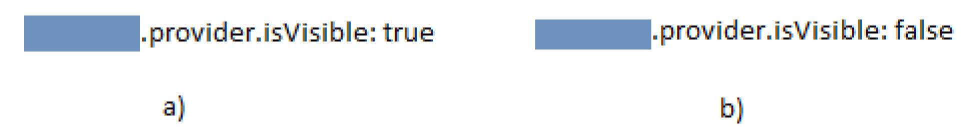
atšķirīgu darījumu sistēmas versiju pieejamību lietotājiem, turklāt tā, lai tās abas veiktu savu darbu vienlīdz uzticami". Apskatot att. 1. pielikumā, redzams, ka pētāmā sistēma ietver sevī datu bāzi. Datu bāzes klasteris sastāv no diviem mezgliem: galvenā jeb *master* mezgla un pakārtotā jeb *slave* mezgla, kas glabā datu kopiju no *master* mezgla. Ja produkcijā tiktu palaistas vienlaicīgi divas Java moduļu versijas, nāktos nodrošināt katrai versijai attiecīgi savu datu bāzes klasteri. Turklāt tas būtu jārealizē tā, lai saturiski dati abos klasteros būtu vienādi, taču spētu atšķiras to formāts vai izkārtojums kolonnās. Kopīgi ar izstrādātāju komandu tika nolemts, ka šādas infrastruktūras nodrošināšana prasa pārāk daudz pūļu, lai tās atmaksātos. Otrkārt, tā kā pētāmo sistēmu izmanto citas sistēmas, nevis lietotāji tieši, eksperimenta grupā un kontroles grupā būtu jādala sistēmas nevis cilvēki. Diskusijās ar izstrādes komandu tika nospriests, ka nav korekti pielietot šādu iedalījumu sistēmām, tika atrasti trīs nopietni iemesli, kādēļ to nedarīt. Pirmkārt, A/B testēšana paredz, ka iekļūšana eksperimenta grupā vai kontrolgrupā notiek pēc nejaušības principa. Tas ir pretrunā ar to, ka no maksājumu apstrādes sistēmām tiek sagaidīta uzticamība un paredzamība. Otrkārt, citas sistēmas izmanto pētāmu sistēmu citādāk nekā to darītu lietotāji. Lietotāju sistēma ir paredzēta specifisku lietotāju mērķu īstenošanai, un arī pieejamā saskarne veidota tā, lai atvieglotu šo procesu. Ja eksperimenta izmaiņas atspoguļojas lietotāju saskarnē un sistēma ir pastāvīgi noslogota, ir ticams, ka drīz pēc izmaiņu izlaišanas tās kāds mēģinās izmantot. Savukārt, ja eksperimenta izmaiņas izmanto cita sistēma, laiks, cik ilgi jāgaida līdz izmantošanai, var ieilgt līdz pat nedēļai. Šādā gadījumā eksperiments notiek ātrāk, ja izstrādātājs manuāli ar testa pieprasījumiem notestē eksperimentālo API daļu. Treškārt, var gadīties, ka, ja tiek pievienota jauna API funkcionalitāte, tā netiek izmantota. Integrēto sistēmu konfigurācijā ir precīzi norādīts, kuras API daļas tiek izmantotas. Atšķirībā no lietotājiem, integrētās sistēmas nespētu pašas atklāt jaunu funkcionalitāti un sākt to izmantot.

Alternatīva metode A/B testēšanai pētāmajā projektā ir *green-blue* izvietošana. Šāda veida izvietošana tiek realizēta *AWS* mākonī, izmantojot nodalījumu starp privāti un publiski pieejamajiem apakštīkliem. Kad jauna sistēmas API versija tiek izlaista produkcijā, tā vispirms tiek izvietota mezglos, kas atrodas privātā apakštīklā. Tai pat laikā produkcijā publiskā apakštīklā turpina darboties esošā produkcijas versija. Privātajā apakštīklā esošie sistēmas mezgli var sūtīt pieprasījumus integrētajām sistēmām un saņemt atbildes, taču citas nevar pirmās nosūtīt pieprasījumu uz privāto apakštīklu. Kad izlaižamā versija izvietota, izstrādātājs izpilda testu kopu, kas pārbauda, kā jaunā sistēma darbojas ar integrētajām sistēmām, lai secinātu, vai jaunā versija ir gatava izlaišanai. Ja jaunā sistēmas versija ir gatava, privātais apakštīkls tiek padarīts atvērts pieprasījumiem no citām sistēmām un produkcijā nonāk jauna API versija. Publiskais apakštīkls, kura darbojās līdzšinējā produkcijas versija, tiek atvienots, taču, ja ir vajadzība, divu stundu laikā šo versiju produkcijā var atjaunot, padarot to atkal

pieejamu. Apakštīkls ar neveiksmīgi izlaisto versiju tādā gadījumā tiek atvienots un produkcija turpina darboties tāpat kā pirms izlaišanas.

4.3.3. Iespēju pārslēgšana

Iespēju pārslēgšana tiek nodrošināta ar Spring programmēšanas ietvara palīdzību. Pētāmās sistēmas kods pirms izvietošanas tiek sapakots izpildāmā JAR arhīvā, kas satur programmatūras serveri *Apache Tomcat*, kurā savukārt ir izvietots sistēmas moduļu kods. Atliek tikai palaist JAR failu, un sistēmas API ir gatavs lietošanai. Bez *Spring Boot* izmantošanas nāktos vispirms uzinstalēt, konfigurēt un palaist *Apache Tomcat* programmatūras serveri vidē, kur paredzēts darbināt sistēmu, tad kopēt sistēmas koda JAR moduļus uz *Tomcat* mapi, no kuras tie būtu palaižami un pasniedzami klientiem. Izmantojot *Spring Boot*, līdz ar maksājumu sistēmas kodu izpildes vidē tiek izvietoti arī konfigurācijas faili, kuros ir norādīta gan pasniedzamo sistēmas moduļu, gan paša *Tomcat* konfigurācija. Konfigurācijas formāts ir *Spring* nodrošināta iespēja, ko sauc par *Spring* profiliem. Konfigurācijas parametri tiek apkopoti profilā, un dažādos profilos konfigurācijas parametram var būt atšķirīgas vērtības.



4.4.att. **Konfigurācijas parametra atšķirīgas vērtības dažādos Spring konfigurācijas profilos; sistēmas iespēja tiek padarīta pieejama - a) vai apslēpta - b).**

Tādējādi sistēmas iespējas var padarīt pieejamas vai apslēpt, balstoties uz to, kurš no konfigurācijas profiliem ir aktivizēts. Attēlā 4.4. redzams piemērs konfigurācijas parametram un, kā tas maina vērtību atkarībā no izvēlētā konfigurācijas profila. Konfigurācijas profils

```
$JAVA_HOME/bin/java  -Dspring.profiles.active=test  -jar  application-name-latest.jar
```

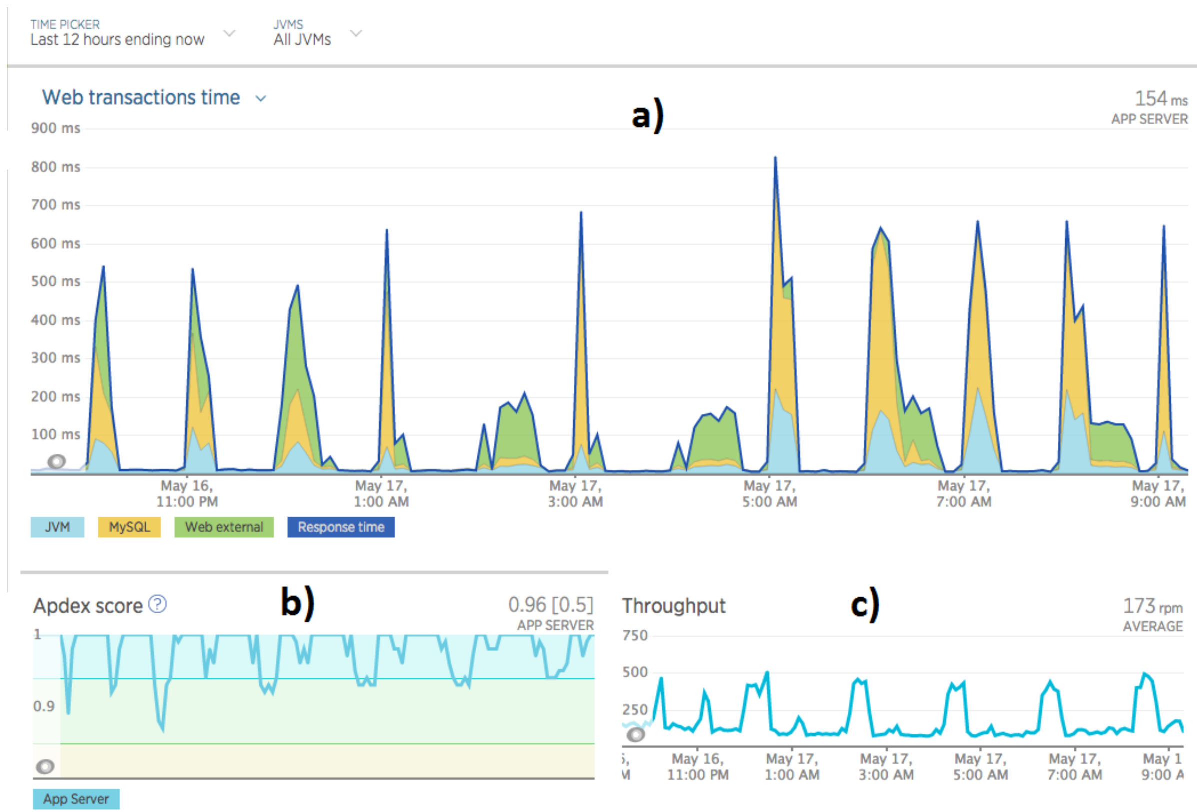
4.5.att. **Konfigurācijas profila norādīšana, palaižot sistēmu**

sistēmai tiek norādīts, palaižot tās JAR moduli, komandas piemērs parādīts attēlā 4.5. Piemērā tiek norādīts testa vides profils, tas tiek padots kā Java komandas papildu parametrs.

Spring profili konfigurācijas pārvaldībai tika sākti izmantot pirms autore uzsāka projekta izpēti, un to sākotnējais mērķis sistēmā ir nodrošināt vieglu veidu, kā pielāgot sistēmas konfigurāciju dažādu integrēto sistēmu vajadzībām. Profils tiek norādīts tikai, izlaižot sistēmu, un netiek mainīts tās darbības laikā. Teorētiski tas ir izdarāms, taču, tā kā projektā nerealizē A/B testēšanu, konfigurāciju maiņa produkcijas darbības laikā nav nepieciešama.

4.3.4. Pārraudzīšana

Pētāmās sistēmas pārraudzīšanas ietvaros tiek ievākti dati par sistēmas kļūdām tās darbības laikā, piemēram, sistēmas kļūdu skaits dotā laikā posmā. Tāpat tiek mērīta sistēmas caurlaide, pētāmas sistēmas izpildes vides veiktspēja un veselīgums, pieprasījumu apstrādes



4.6.att. **Pārraudzīšanas dati par a) maksājumu apstrādes laiku, b) ? un c) pētāmās sistēmas caurlaidi**

laiks un laiks, cik ilgi tiek gaidītas atbildes no citām sistēmām. Projekta izvēlētais

pārraudzīšanas rīks *NewRelic*. Attēlā 4.6. redzams pārraudzīšanas datu vizualizācijas piemērs *NewRelic* lietotāja saskarnē.

Pārraudzīšanas laikā ievāktos datus izstrādātu komanda analizē reizi dienā. Kādam no izstrādātājiem, kurš tajā dienā ir atbildīgs par sistēmas uzturēšanas veikšanu, tiek dots pienākums analizēt pārraudzīšanas datus, lai pārliecinātos par sistēmas veselību.

HVI vajadzībām pārraudzīšanas dati ir izmantojami, lai pārbaudītu hipotēzi nedēļas laikā. Ja hipotēzes pārbaudīšanai nepieciešams ilgāks laiks posms, jāmeklē risinājums kā uzglabāt ilgāku datu vēsturi.

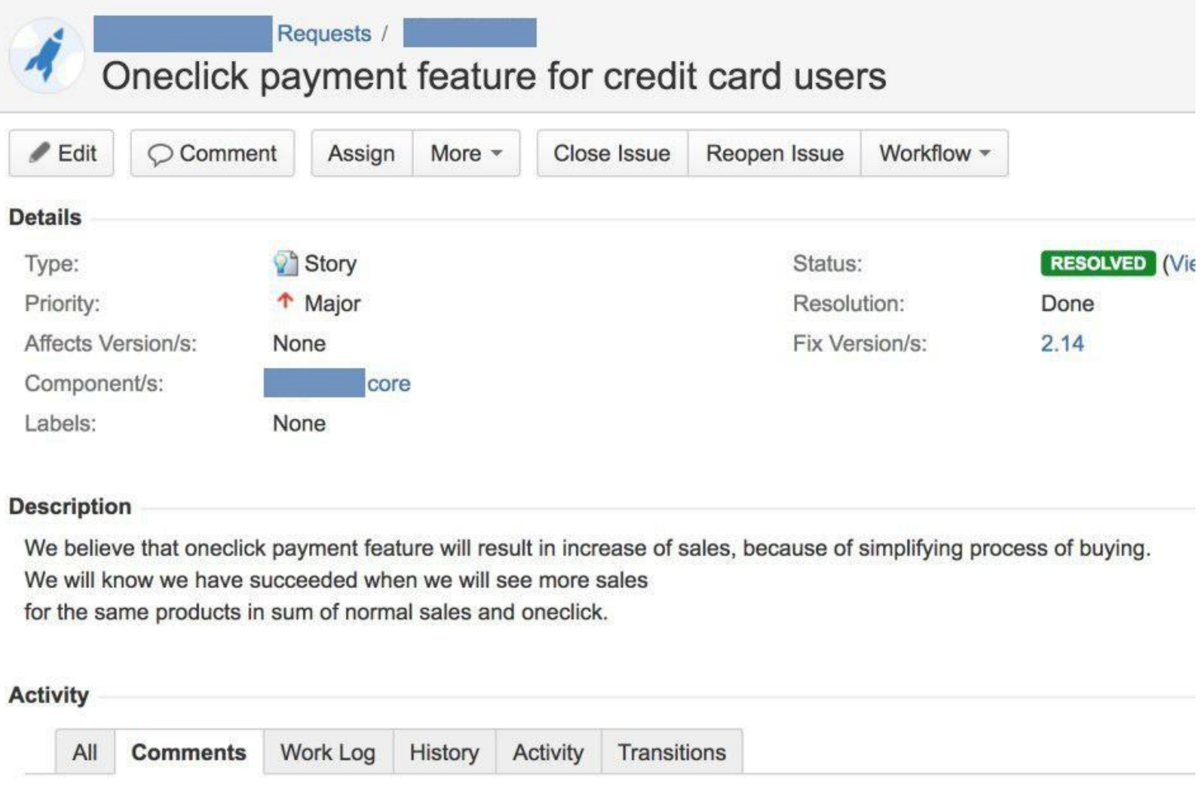
4.4. Piemērs jaunas iespējas izstrādei, ievērojot HVI

Iepriekšējās nodaļās tika aprakstīts, programminženierijas metodes un tehnoloģijas, kas tiek izmantotas pētāmajā projektā. Šajā nodaļā tiek ilustrēts ar piemēru, kā notiek jaunas sistēmas iespējas izstrāde, sākot no hipotēzes formulēšanas un beidzot ar koda izlaišanu produkcijā.

4.4.1. Hipotēzes formulēšana

Viena no sistēmām, ar kuru integrēta pētāmā maksājumu vārtejas sistēma, ir lietotāju portāls, kas piedāvā iespēju pasūtīt preces tiešsaistē un samaksāt par tām elektroniski. Lietotāju sistēma neietilpst Alfa projektā, taču pasūtītājs, ar ko strādā Alfa, sadarbojas arī ar minētās lietotāju sistēmas izstrādātājiem, tādējādi pildot komunikāciju starpnieka lomu starp abu sistēmu izstrādes komandām. Ir iespējas, kuru pilnīgai implementācijai, tās nepieciešams realizēt vairākās sistēmās. Piemēram, maksājums par pasūtīto preci. Lietotāju sistēma nodrošina saskarni, kurā lietotājs ievada maksājuma datus, un nosūta tos apstrādei. Taču šajā lietotāju sistēmā nav funkcionalitātes maksājuma apstrādei, tādēļ maksājuma dati tiek caur maksājumu vārtejas sistēmu sūtīti tālāk sistēmām, kas realizē elektronisku maksājuma izpildi. Kādam Alfas izstrādātājam, kurš pārzināja, kā darbojas abas sistēmas – lietotāju un vārtejas –, radās ideja, ka lietotājiem būtu ērtāk sistēmā norēķināties, ja katru reizi, veicot maksājumu, nevajadzētu ievadīt datus par savu bankas karti. Pasūtītājam tika piedāvāts ieviest iespēju norēķināties par pasūtījumu ar vienu pogas spiedienu. Uzklausot izstrādātāju komandas argumentāciju, pasūtītājs piekrita šādas iespējas nepieciešamībai un nodrošināja, ka lietotāju sistēmā arī tiek uzsākta šādas iespējas implementācija, ieviešot nepieciešamos saskarnes elementus.

Pirms sākt izstrādi, tika formulēta un dokumentēta hipotēze par jaunās iespējas



4.7.att. **Hipotēzes formulējums *JIRA* uzdevuma aprakstā**

rezultātiem (skat. 4.7. att.). Lai arī lietotāji tiešā veidā pētāmo sistēmu neizmanto, hipotēzē tie ir minēti, jo iespējas ieviešanas mērķis ir ieguvums lietotājiem un maksājumu vārteja ir viena no sistēmām, kurā iespēja ir jāimplementē, lai lietotāji to spētu izmantot. Hipotēze tika formulēta atbilstoši formātam, kāds aprakstīts 3. nodaļā. Tā apgalvo, ka iespēja samaksāt par preci ar vienu pogas spiedienu jeb *oneclick* paaugstinās darījumu skaitu, jo tā atvieglos maksājuma procesu lietotājam. Ja hipotēze izrādās patiesa, to varēs zināt, redzot, ka fiksētā laika posmā fiksēta veida preci biežāk pērk, izmantojot *oneclick* iespēju, nevis parasto maksājumu.

4.4.2. Izstrāde un izlaišana

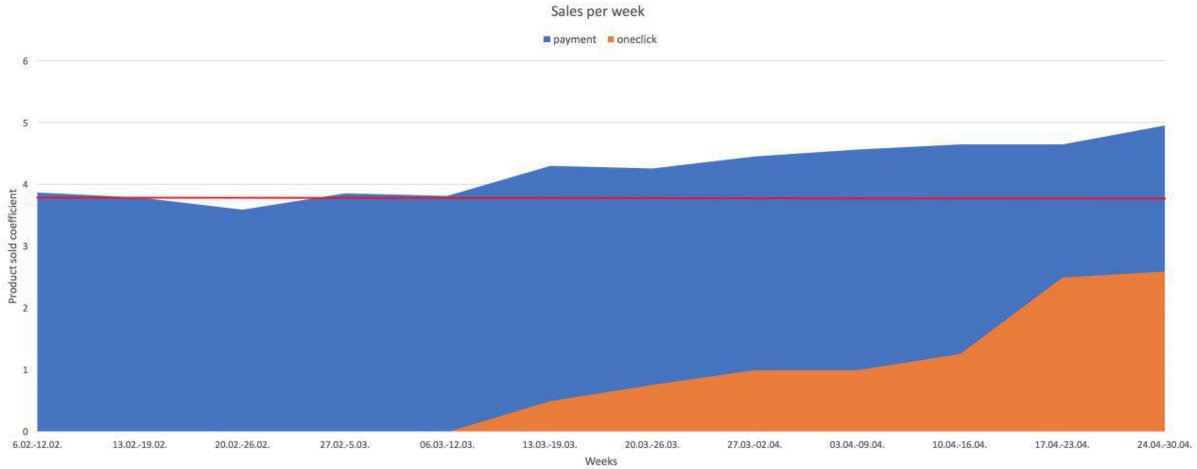
Oneclick maksājumu iespējas implementācija pētāmajā sistēmā norisinājās vienas darba nedēļas garumā. Četras dienas bija nepieciešamas funkcionalitātes un testu uzrakstīšanai. Šajā laikā būvējumu ķēde *Bamboo* tika izieta četras reizes, izstrādātāji savu kodu iepludināja kopējā repozitorijā reizi dienā. Divas no četrām versijām tika izvietotas testēšanas vidē, lai pārbaudītu veiktspēju un regresiju. Piektajā dienā notika jaunās iespējas izvietošana produkcijā. *Green-*

blue izvietošanas laikā veikto testu rezultāti bija pozitīvi, tādēļ jaunā iespēja tika izlaista produkcijā.

Kopumā, no pasūtītāja piekrišanas *oneclick* maksājuma iekļaušanai līdz brīdim, kad šī iespēja tika implementēta un izlaista visās sistēmās, kuras tā skar, pagāja divas darba nedēļas.

4.4.3. Hipotēzes pārbaude

Lai pārbaudītu, vai izstrādātāju izvirzītā hipotēze bija patiesa vai nebija, tika izmantoti dati no pārraudzīšanas sistēmas, kā arī dati no pētāmās sistēmas datu bāzes. Ar pārraudzīšanas sistēmas ievāktajiem datiem nepietika, lai iegūtu nepieciešamo informāciju, tādēļ nācās ņemt papildu datus no sistēmas. Autore apzinās, ka pārraudzīšanas rīku iespējas ir ierobežotas un var nebūt pietiekamas specifisku sistēmas lielumu nolasīšanai, tādēļ nepieciešamība izmantot papildus datu avotu netiek uzskatīta par nopietnu trūkumu. 4.8. attēlā parādīti eksperimenta



4.8.att. **Maksājumu, kas veikti ar *oneclick* metodi (oranžs), īpatsvars, salīdzinājumā ar maksājumiem, kas veikti, izmantojot parasto maksājuma metodi (zils)**

iespējas ieviešanas rezultāti. Grafikā redzamie dati ievākti laika posmā, kas aptver 11 kalendārās nedēļas. Uz horizontālās ass atlikts laiks nedēļās, uz vertikālās – daļa, kādu aizņem katra veida maksājumu skaits, attiecībā pret kopējo maksājumu skaitu. Pirmās četras nedēļas sistēmā bija pieejama tikai parastā maksājumu iespēja, tad tika izlaista *oneclick*. Sarkanā palīglīnija ļauj saskatīt divus novērojumus: a) uzreiz pēc oneclick iespējas ieviešanas pieauga kopējais saņemto maksājumu skaits; b) tuvojoties eksperimenta norises beigām, oneclick maksājumu daļa ir nedaudz lielāka par parasto maksājumu daļu. Pēc novērojumiem grafikā **tika secināts, ka izstrādātāju hipotēze ir pierādījusies.**

Sākotnēji bija paredzēts ietvert datus par īsāku laika posmu un apskatīt 6 nedēļas pēc oneclick izlaišanas, nevis 8. Taču 6 nedēļas beigās sākās straujš oneclick maksājumu skaita pieaugums, tādēļ tika nolemts aptvert ilgāku laika posmu, lai redzētu, cik ilgi straujā augšana turpinātos. Kā izrādījās, straujais kāpums notika laika periodā, kad lietotāju sistēmā bija atlaides lielam skaitam preču. Acumirklī nebija skaidrs, kāpēc tieši atlaižu periodā oneclick maksājumu iespēja kļuva populāra. Apspriežoties ar projekta komandu, autore nonāca pie pieņēmuma, ka atlaižu dēļ lietotāju aktivitāte sistēmā pieauga kopumā. Tā rezultātā vairāk lietotāju pamanīja jauno maksāšanas iespēju un sāka to izmantot.

4.5. HVI un tās priekšnosacījumu īstenošanas rezultāti pētāmajā projektā

Kopsavilkumā, autore uzskata, ka HVI īstenošana plašākā spējās izstrādes ietvarā bija vērtīgs izmēģinājums. Īstenojot šo metodi, izdevās aptvert, ar kādiem izaicinājumiem var nākties saskarties metodes īstenošanā. Lai arī teorijā aprakstīto HVI realizāciju izdevās daļēji sasniegt, pēc autores domām, izvēlētais risinājums sniedza pietiekamas iespējas HVI izpētei. Tabulā 4.2. sniegts pārskats par projektā īstenotajām izstrādes metodēm.

4.2.tabula. **Programminženierijas metožu izmantojums pētāmajā projektā**

Izstrādes metode	Izmantojums projektā
TVI	-
UVI	-
HVI	Jā
Nepārtrauktā integrācija	Jā
Nepārtrauktā piegāde	Jā
Nepārtrauktā izvietošana	-
A/B testēšana	-
Iespēju pārslēgšana	Jā

Pētījuma beigās projekta komandai tika palūgts sniegt savu vērtējumu, kādu ieguvumu projekts ir guvis no 4.2. tabulā minēto metožu izmantošanas. Tika lūgts izteikties gan par tām metodēm, kas ieviestas pirms autore uzsāka savu pētījumu, gan tām, kuras tika ieviestas pētījuma laikā.

4.5.1. Ieguvumi projektā no nepārtrauktās integrācijas

Izstrādātāji atzina, ka automatizēta būvējumu ķēde katrai jaunai sistēmas versijai ar laiku liek vairāk uzmanības pievērst koda kvalitātei un tās nodrošināšanai. Pēc izstrādātāju domām, nepārtrauktā integrācija kopumā ir palīdzējusi uzlabot koda kvalitāti projektā, arī vecākais programmētājs atzina, ka, pārbaudot komandas jaunpienācēju darbu, ir ievērojis, kā ar laiku regulāra izmaiņu integrēšana, būvēšana un testēšana palīdz uzlabot izstrādes ātrumu.

Izmantojot *Bamboo*, lai izvietotu kodu testa vidēs, izvietošanas process ir uzticamāks un ātrāks, nekā, ja to veiktu izstrādātāji manuāli. Tāpat izmaiņu atsaukšana no vides, kur tā izvietota, ir ātrs un uzticams, pēc komandas atzinuma.

4.5.2. Ieguvumi projektā no nepārtrauktās piegādes

Pirms nepārtrauktās piegādes realizācijas, jauna sistēmas versija pasūtītājam tika izlaista ne biežāk kā reizi trijos mēnešos. Pēc tam, kad, pateicoties nepārtrauktajai integrācijai un nepārtrauktajai piegādei, tika nodrošināts ārts un drošs veids kā izvietot sistēmu produkcijā, izlaišanas biežums pieauga, un tās notiek reizi nedēļā. Biežākas izlaišanas rezultātā, tiek īsā laikā iegūtas atsauksmes par jaunām sistēmas iespējām no izstrādātājiem, kas nodarbojas ar integrēto sistēmu uzturēšanu.

Alfas komanda tic, ka ir drošāk izlaist jaunas iespējas biežāk un nelielās porcijās nekā darīt to reti un lielos apjomos, jo samazinās pēc izlaišanas atklāto kļūdu skaits, kas savukārt atvieglo tās prioritizēt labošanai. Šo ieguvumu ir atzinīgi novērtējis arī pasūtītājs. Ar nepārtraukto piegādi, katrā izlaišanā produkcijā nonāk 2-3 jaunas sistēmas iespējas. Agrāk tika izlaistas vairāk kā 20 iespējas reizē..

4.5.3. Ieguvumi projektā no iespēju pārslēgšanas

Kā jau minēts 4.3.3. nodaļā, galvenais ieguvums no iespēju pārslēgšanas pētāmajā projektā ir "pielāgot sistēmas konfigurāciju dažādu integrēto sistēmu vajadzībām". Padarot atsevišķas sistēmas iespējas pieejamas konkrētai integrētajai sistēmai, Alfas komanda ir atradusi veidu, kā integrētās sistēmas izstrādātāji var izmēģināt jaunās Alfas iespējas, pirms tās turpināt implementēt savā sistēmā.

Tāpat komanda atzina, ka *Spring* profilu izmantošana ir drošs un ērts veids, kā pārvaldīt sistēmas iespēju pieejamību. Vairāki izstrādātāji izteica ierosinājumu nākotnē izmantot iespēju pārslēgšanu, lai nošķirtu iespējas izvietošanas procesu no tās izlaišanas. Piemēram, ja,

gatavojoties izmaiņām banku sistēmās, tiek pievienota jauna iespēja un izvietota produkcijā jau laikus. Kad izmaiņas banku sistēmā notiktu, atliktu jauno Alfas iespēju vien “ieslēgt”.

4.5.4. Ieguvumi projektā no HVI

Pirms autore uzsāka savu pētījumu, projekta pasūtītājs bija izteicis vēlmi, lai projekta komanda piedalītos sistēmas uzlabojumu ieteikšanā. Autores pētījuma laikā šo pasūtītāja vēlmi izdevās realizēt, ar projekta komandas palīdzību izpētot iespējas, ko piedāvā dažādas maksājumu sistēmas, un atrodot veidu, kā gūt labumu, pievienojot jaunu iespēju projekta sistēmai.

Hipotēzes formāta izmantošana ierastā lietotāju stāsta formāta vietā, pēc komandas atzinuma, pašiem izstrādātājiem nekādas būtiskas izmaiņas darbā nav radījusi. Tāpat kā lietotāju stāsti, hipotēze palīdz “sakārtot” domas par to, ko tieši ir nepieciešams izstrādāt un kā. Tiek pastiprināts uzskats par izstrādājāmo sistēmu kā vienotu veselumu. Tiesa, izstrādātāju vidū tika izteikts viedoklis, ka hipotēzes formāts varētu kalpot par komunikācijas saskarni starp izstradātājiem un pasūtītāju, lai labāk saprastu citam citu.

Izstrādātāji atzina, ka, strādājot pie iespēju implementācijas un pašiem pārbaudot to izlaišanas rezultātu, pieauga viņu interese par to, kā jaunas iespējas sistēmā ietekmē pasūtītāju. Tika izteikta cerība, ka nākotnē izpratne par sistēmas iespēju ietekmi uz pasūtītāja sfēru palīdzētu izvēlēties no teorētiski iespējamiem sistēmas uzlabojumiem tos, kuri ir vērtīgāki pasūtītājam.

Viens no izstrādātājiem piezīmēja, ka HVI metode šķiet piemērota *start-up* tipa uzņēmumiem vai jaunu produktu izstradē, meklējot tiem atbilstošo nišu.

REZULTĀTI

Darba izstrādes rezultātā autore ir izpētījusi trīs programminženierijas pieejas, TVI, UVI un HVI, kā arī veikusi praktisku pētījumu par HVI izmantošanu IT projektā. Ir iepazītas izpētīto pieeju pamata idejas un procesi, kā arī nozares pārstavju pieredze to izmantošanā. Sākot rakstīt darbu, autore formulēja jautājumus, uz kuriem censties rast atbildes darba izstrādes gaitā. Visi jautājumi bija veltīti HVI, zemāk atrodamas autores atbildes uz tiem.

Kā to īstenot? HVI pamatā ir ciklisks process, kas ietver sevī eksperimenta veikšanu sistēmas produkcijas vidē. Viss sākas ar izstrādātāju novērojumiem par sistēmu. Balstoties uz novērojuma, tiek izvirzīta hipotēze par jaunas iespējas ieviestām pārmaiņām sistēmā. Tad notiek jaunās iespējas izstrāde, testēšana, piegāde un izlaišana produkcijā. Kad izmaiņa izlaista produkcijā, tās lietotājus pēc nejaušības principa sadala eksperimenta un kontroles grupā un novēro reakciju uz jauno sistēmas iespēju. Ja tā ir pozitīva, hipotēze ir pierādīta pierādīta un jaunā iespēja tiek izlaista visiem lietotājiem. Autore identificēja četrus HVI priekšnosacījumus, ko ieteiktu īstenot: nepārtraukto izvietošanu; A/B testēšanu; iespēju pārslēgšanu un sistēmas pārraudzīšanu.

Kāds no tās ir ieguvums izstrādātājiem un pasūtītājam? Praktiskā pētījuma daļas rezultāti rāda, ka hipotēze palīdz "sakārtot" domas par to, ko tieši ir nepieciešams izstrādāt un kā. Pasūtītājs gūst papildu labumu, autores pētījuma gadījumā – peļņu, no tā, ka sistēmas izstrādātāji izsaka piedāvājumus, kā uzlabot sistēmu ar jaunām iespējām un savus piedāvājumus īsteno. Sistēmas izstrādātāji izrāda vēlmi izprast pasūtītaja sfēru un sniegt savu ieguldījumu piegādājamā produkta uzlabošanā.

Vai šī metode ir kādā veidā saistīta ar TVI vai UVI? Pēc visu triju pieeju izpētes nevar noliegt, ka līdzība starp to procesiem pastāv. Tām visām pamatā ir cikls, kas sākas ar sagaidāmā rezultāta vai mērķa definēšanu, kam seko izstrādes process un beigās – pārbaude, vai sākotnēji gaidītais ir sasniegts. Lai arī šīs metodes ir līdzīgas, autore uzskata, ka HVI ir neatkarīga no abām pārējām un ir izmantojama arī atsevišķi. Praktiskājā daļā HVI tika īstenota bez TVI vai UVI, un iegūtie rezultāti sniedz autorei pamatu tā uzskatīt.

IZMANTOTĀ LITERATŪRA UN AVOTI

[1] «Manifesto of Agile Software Development,» [Tiešsaiste]. Available: http://agilemanifesto.org/principles.html. [Piekļūts 26. 03. 2017.].

[2] «Jenkins,» [Tiešsaiste]. Available: https://jenkins.io/. [Piekļūts 27. 03. 2017.].

[3] «The Software Testing Ice Cream Cone,» [Tiešsaiste]. Available: http://saeedgatson.com/the-software-testing-ice-cream-cone/. [Piekļūts 27. 03. 2017.].

[4] «Test Pyramid,» [Tiešsaiste]. Available: https://martinfowler.com/bliki/TestPyramid.html. [Piekļūts 04.01. 2017.].

[5] K. Beck, Test-Driven Development by Example, Addison Wesley - Vaseem, 2003.

[6] «Using TDD to Influence Design,» [Tiešsaiste]. Available: https://www.thoughtworks.com/insights/blog/using-tdd-influence-design. [Piekļūts 05. 04. 2017.].

[7] «StackExchange.com,» [Tiešsaiste]. Available: https://softwareengineering.stackexchange.com/a/59935. [Piekļūts 04. 04. 2017.].

[8] «Why Test Driven Development,» [Tiešsaiste]. Available: http://derekbarber.ca/blog/2012/03/27/why-test-driven-development/. [Piekļūts 05. 04. 2017.].

[9] N. Nagappan, Realizing quality improvement through test driven, 2008.

[10] «Mockists are Dead, Long Live Classicists,» [Tiešsaiste]. Available: https://www.thoughtworks.com/insights/blog/mockists-are-dead-long-live-classicists. [Piekļūts 15. 04. 2017.].

[11] M. Fowler, «Is TDD Dead?,» [Tiešsaiste]. Available: https://martinfowler.com/articles/is-tdd-dead/. [Piekļūts 19. 01. 2017.].

[12] «NDC London,» [Tiešsaiste]. Available: http://ndc-london.com/. [Piekļūts 19. 01. 2017.].

[13] «Gojko Adzic,» [Tiešsaiste]. Available: https://www.linkedin.com/in/gojko. [Piekļūts 19. 01. 2017.].

[14] «Gojko Adzic - BDD: Busting the myths,» [Tiešsaiste]. Available: https://vimeo.com/43612884. [Piekļūts 16. 01. 2017.].

[15] «Getting started with Cucumber,» [Tiešsaiste]. Available: https://cucumber.io/docs. [Piekļūts 18. 01. 2017.].

[16] «3 misconceptions about BDD,» [Tiešsaiste]. Available: https://www.thoughtworks.com/insights/blog/3-misconceptions-about-bdd. [Piekļūts 19. 01. 2017.].

[17] G. Kim, J. Humble un P. Debois, DevOps Handbook, 2015.

[18] «Zabbix,» [Tiešsaiste]. Available: http://www.zabbix.com/. [Piekļūts 04. 04. 2017.].

[19] «SensuApp,» [Tiešsaiste]. Available: https://sensuapp.org/. [Piekļūts 05. 05. 2017.].

[20] «ThoughtWorks,» [Tiešsaiste]. Available: https://www.thoughtworks.com/about-us. [Piekļūts 07. 05. 2017.].

[21] «Jez Humble,» [Tiešsaiste]. Available: https://www.linkedin.com/in/jez-humble-02bb73. [Piekļūts 19. 01. 2017.].

[22] J. Humble, Continous Delivery, 2008.

[23] «ThoughtWorks,» [Tiešsaiste]. Available: https://www.thoughtworks.com/about-us. [Piekļūts 19. 01. 2017.].

[24] J. L. Taylor, «Hypothesis-Driven Development,» 2011. [Tiešsaiste]. Available: http://www.drdobbs.com/architecture-and-design/hypothesis-driven-development/229000656. [Piekļūts 19. 01. 2017.].

[25] B. O'Reilly, «How to Implement Hypothesis-Driven Development,» [Tiešsaiste]. Available: https://www.thoughtworks.com/insights/blog/how-implement-hypothesis-driven-development. [Piekļūts 19. 01. 2017.].

[26] K. Beck, «Why does Kent Beck refer to the "rediscovery" of test-driven development?,» [Tiešsaiste]. Available: https://www.quora.com/Why-does-Kent-Beck-refer-to-the-rediscovery-of-test-driven-development. [Piekļūts 19. 01. 2017.].

[27] «Introduction to Writing Acceptance Tests with Cucumber,» [Tiešsaiste]. Available: https://semaphoreci.com/community/tutorials/introduction-to-writing-acceptance-tests-with-cucumber. [Piekļūts 19. 01. 2017.].

[28] M.Wynne un A.Hellesoy, The Cucumber Book: Behaviour-Driven Development for Testers and Developers, Pragmatic Bookshelf, 2012.

[29] «Using the Agile Testing Quadrants,» [Tiešsaiste]. Available: http://lisacrispin.com/2011/11/08/using-the-agile-testing-quadrants/. [Piekļūts 19. 01. 2017.].

[30] P. Solutions. [Tiešsaiste]. Available: http://www.pathfindersolns.com/wp-content/uploads/2012/05/Effective-TDD-Executive-Summary.pdf. [Piekļūts 19. 01. 2017.].

[31] «Jenkins and Deployment Pipelines,» [Tiešsaiste]. Available: https://trevormquinn.com/2015/01/07/jenkins-and-deployment-pipelines/. [Piekļūts 27. 03. 2017.].

[32] «Test Driven Development,» [Tiešsaiste]. Available: http://www.gargoylesoftware.com/practices/tdd. [Piekļūts 04. 03. 2017.].

[33] «Testing Triangles, Pyramids and Circles,» [Tiešsaiste]. Available: http://allankelly.blogspot.com/2013/05/testing-triangles-pyramids-and-circles.html. [Piekļūts 20. 04. 2017.].

[34] «Agile User Stories,» [Tiešsaiste]. Available: https://www.scrumalliance.org/community/articles/2013/september/agile-user-stories. [Piekļūts 03. 05. 2017.].

[35] «SlideShare,» [Tiešsaiste]. Available: https://image.slidesharecdn.com/entregacontinuaenlapractica-160323115544/95/entrega-continua-en-la-practica-22-638.jpg?cb=1458734300. [Piekļūts 28. 04. 2017.].

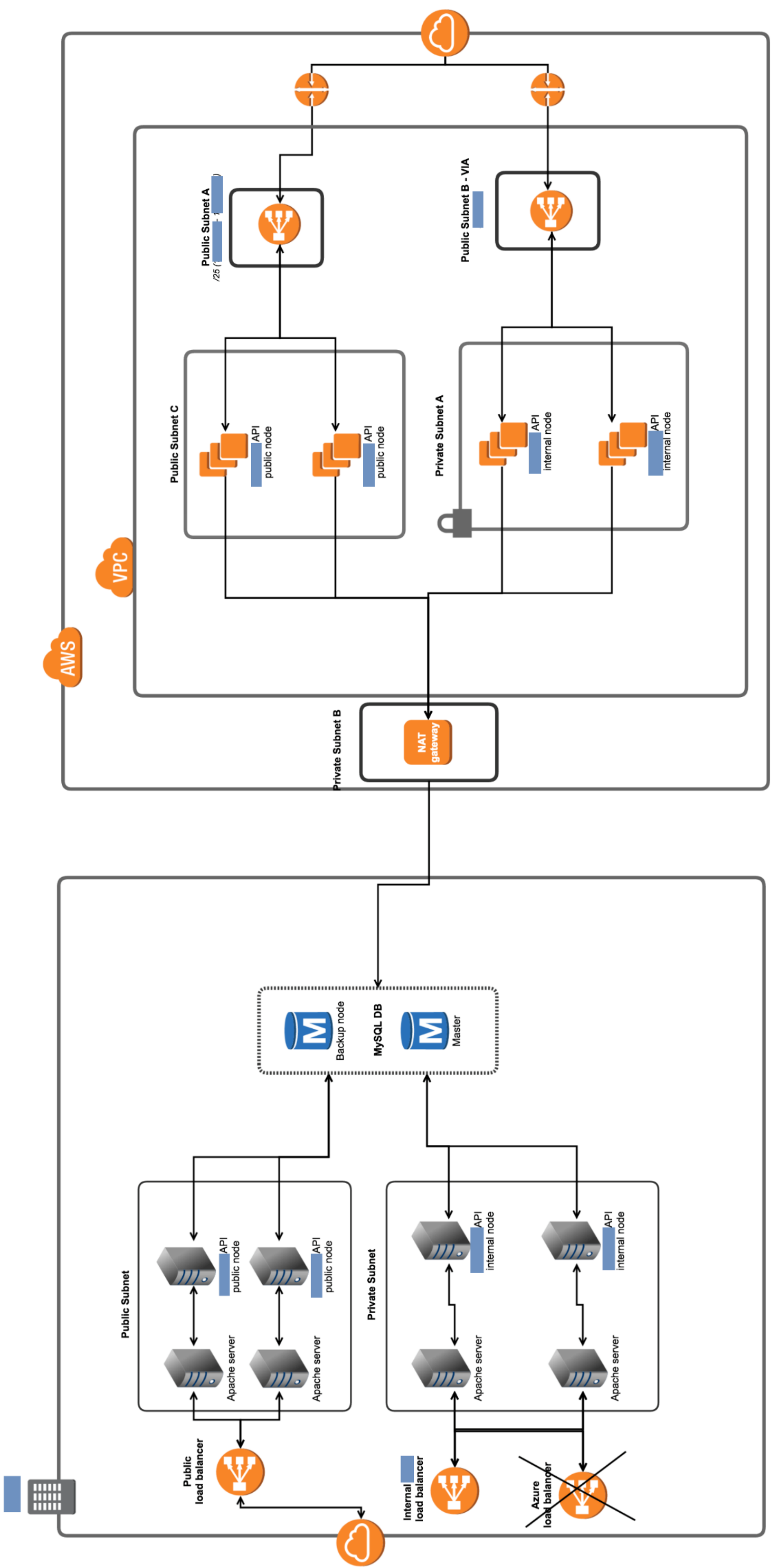
[36] «Continuous Delivery in the Cloud,» [Tiešsaiste]. Available: https://blog.codecentric.de/en/2012/04/continuous-delivery-in-the-cloud-part1-overview/. [Piekļūts 02. 05. 2017.].

[37] «Continuous Delivery VS Continuous Deployment,» [Tiešsaiste]. Available: https://puppet.com/blog/continuous-delivery-vs-continuous-deployment-what-s-diff. [Piekļūts 02. 05. 2017.].

[38] «The Ultimate Guide to A/B Testing,» [Tiešsaiste]. Available: https://www.smashingmagazine.com/2010/06/the-ultimate-guide-to-a-b-testing/. [Piekļūts 04. 04. 2017.].

[39] «How to Implement Feature Flags and AB Testing,» [Tiešsaiste]. Available: https://blogs.msdn.microsoft.com/visualstudioalmrangers/2017/04/04/how-to-implement-feature-flags-and-ab-testing/. [Piekļūts 06. 05. 2017.].

PIELIKUMI



1.att. **Elektronisko darījumu apstrādes sistēmas produkcijas vides sadalījums *Amazon Web Services* un vēl viena mākoņpakalpojumu sniedzēja infrastruktūrā**

Maģistra darbs **“Hipotēžu virzītas izstrādes izpēte”** izstrādāts LU Datorikas fakultātē.

Darba teksta galīgā versija izgatavota **22.05.2017.**

Ar savu parakstu apliecinu, ka pētījums veikts patstāvīgi, izmantoti tikai tajā norādītie informācijas avoti un iesniegtā darba elektroniskā kopija atbilst izdrukai.

Autors:______________________________

(Autora paraksts un datums)

Ar savu parakstu apliecinu, ka esmu lasījis augstāk minēto maģistra darbu un atzīstu to par **p i e m ē r o t u / n e p i e m ē r o t u** (nevajadzīgo svītrot) aizstāvēšanai Latvijas Universitātes datorzinātņu maģistrantūrā.

Darba vadītājs: _______________________________

(Vadītāja paraksts un datums)

Darbs iesniegts **maģistrantūras sekretariātā**__________________________.

(Iesniegšanas datums)

Ar šo es apliecinu, ka darba elektroniskā versija ir augšupielādēta LU informatīvajā sistēmā.

Studju metodiķe:_____________________.

(Metodiķes paraksts)

Recenzents:__________________________________

(Akad.amats, zin.grāds, vārds, uzvārds)

Darbs aizstāvēts maģistra gala pārbaudījumu komisijas sēdē

_______________________________ prot.Nr. _______

(Darba aizstāvēšanas datums)

Komisijas sekretārs:_______________________________

(Sekretāra paraksts)