Latvijas Universitāte
Fizikas un matemātikas fakultāte
Datorikas nodaļa

Tīmekļa programmu automatizēta testēšana

Bakalaura darbs

Autors
Mārtiņš Bruņenieks

Vadītājs
Guntis Arnicāns
Dr. sc. comp.
Datorikas nodaļa, docents,
Latvijas Universitāte

Rīga, 2006.

Anotācija

Tīmekļa aplikācijām kļūstot populārākām, pieaug prasības pret to kvalitāti, un automatizētas testēšanas izmantošana ir viena no metodēm, kā to panākt. Tīmekļa aplikāciju izstrādē pielietotā stratēģija „izlaid ātri, izlaid bieži" izmaksu samazināšanai spiež izvēlēties automatizētu testēšanu, lai nodrošinātu pieprasīto kvalitāti. Šajā darbā apskatītas automatizācijas metodes, to izmantošana izstrādes procesā un arī konkrētu rīku izmantošana un sniegti secinājumi un rekomendācijas.

Šī darba mērķis ir izpētīt iespējas, kādas ir pieejamas tīmekļa aplikāciju testēšanā un apskatīt, ko šajā jomā piedāvā gatavie automatizācijas rīki.

Annotation

As web applications become more and more popular, the quality requirements increase as well. Use of automated testing is one of methods to reach this goal. Most web projects use “release early, release often” strategy. It requires use of automated testing to meet the required quality. In this paper there are described automation methods use of them in development process and given some conclusions and recommendations.

The goal of this paper is to research possibilities that are available for automated testing of web applications and find out what are features provided by third party tools.

Annotatie

Aangezien de Webtoepassingen en meer populairder worden, de verhoging van kwaliteitsvereisten eveneens. Het gebruik van het geautomatiseerde testen is één van methodes om dit doel te bereiken. De meeste Webprojecten gebruiken "versie vroeg, versie vaak" strategie. Het vereist gebruik van het geautomatiseerde testen om de vereiste kwaliteit te ontmoeten. In dit document is er het beschreven gebruik van automatiseringsmethodes van hen in ontwikkelingsproces en gegeven sommige conclusies en aanbevelingen.

Het doel van dit document is aan onderzoekmogelijkheden die beschikbaar voor het geautomatiseerde testen van Webtoepassingen zijn en te weten komen wat eigenschappen die door derdehulpmiddelen zijn worden verstrekt.

Autoreferāts

Strādājot pie automatizētas testēšanas aktivitāšu ieviešanas izstrādes procesā uzņēmumā, kas nodarbojas galvenokārt ar tīmekļa programmu izstrādi, ir radusies sapratne par testēšanas nozīmi produkta kvalitātes nodrošināšanā. Automatizēto testu ieviešanas lielākais ieguvums ir regresijas kļūdu samazināšanas piedāvātā iespēja. Tā ir īpaši nozīmīga izstrādājot pēc iespējas universālāku sistēmu. Šī sistēma tiek veidota kā nepārtrauktās integrācijas process, klientam nepieciešamo modifikāciju ieviešanai izmantojot konfigurēšanas iespējas, nevis jaunu atzaru veidošanu produkta koda bāzē.

Strādājot pie šajā bakalaura darbā apskatītājām tēmām, ir paveiktas šādas lietas :

- No dažādiem avotiem apkopota automatizētās testēšanas teorija
- Aprakstīti tīmekļa aplikāciju testēšanas veidi un iespējas tos praktiski automatizēt
- Noskaidroti iespējamie problemātiskie ievaddati tīmekļa aplikācijām un piedāvāts veids, kā tos iekļaut automatizācijas procesā
- Uzskaitīti automatizētās testēšanas rīki tīmekļa aplikāciju testēšanas vajadzībām
- Aprakstīti konkrēti rīki, kas piedāvā automatizācijas iespējas
- Sniegtas rekomendācijas automatizācijas procesu organizēšanā

Satura rādītājs

IEVADS **7**

1 TESTĒŠANAS PROCESS **8**

1.1 PRODUKTA DZĪVES CIKLS 8
1.2 KAD AUTOMATIZĒT TESTUS? 10

2 TĪMEKĻA APLIKĀCIJU TESTĒŠANA **12**

2.1 TESTĒŠANAS AUTOMATIZĀCIJA 17
2.1.1 Automatizācijas veidi *18*
2.1.2 Automatizētās testēšanas plāns *19*
2.2 UNIVERSĀLĀ TESTĒŠANAS VIDE XUNIT 21
2.3 TESTĒŠANAS VIRZIENI 23
2.3.1 Aplikācijas moduļu vienību testēšana *23*
2.3.2 HTML dokumenta testēšana *24*
2.3.3 Renderētā dokumenta testēšana *24*
2.3.4 Funkcionalitātes testēšana *25*
2.3.5 Tīmekļa programmām specifiskie ievaddati *26*
2.3.6 Scenāriju valodu testēšana *31*
2.3.7 Drošības testēšana *33*

3 TESTĒŠANAS RĪKI **37**

3.1 BITMECHANIC MAXQ 42
3.2 SELENIUM IDE 44
3.3 HTTPUNIT 46
3.4 THOUGHTWORKS SAHI 48
3.5 APPPERFECT WEBTEST 50
3.6 RĪKU KOPSAVILKUMS 52
3.7 SECINĀJUMI UN PRIEKŠLIKUMI 54

NOSLĒGUMS **60**

IZMANTOTĀS LITERATŪRAS SARAKSTS **61**

1. PIELIKUMS – VĀRDNĪCA **65**

2. PIELIKUMS – TESTU SCENĀRIJI **66**

2.1. MAXQ 66
2.2. SELENIUM IDE 67
2.3. HTTPUNIT 68
2.4. TESTINPUT KLASE: 69
2.5. SAHI 70

Ievads

Tīmekļa aplikācijas kļūst arvien populārākas un to kvalitātei būtu jāaug līdzi kvantitātes pieauguma tempiem. Popularitātes galvenais iemesls ir to universālums – tās var izmanot, neatkarīgi no platformas, lai gan joprojām eksistē savietojamības problēmas, kas gan ir atrisināmas.

Tīmekļa aplikāciju izstrāde bieži notiek, izmantojot spējās un reizēm pat ekstrēmas programmatūras izstrādes metodes. Šīs metodes lieliski var izmantot automatizētās testēšanas iespējas, jo jauno iespēju ieviešana notiek bieži un to detalizācija notiek pakāpeniski. Tieši tāpēc iespēja veikt sistēmas funkcionalitātes pārbaudi ar zemām izmaksām pēc katrām izmaiņām ir piedāvājums, no kā ir grūti atteikties. Bieži vien alternatīva ir neveikt testēšanu, jo manuāla problēmu meklēšana izmaksā pārāk dārgi.

Tāpēc šajā darbā apskatīsim automatizēšanas iespējas, vajadzības, metodes un arī konkrētus rīkus.

1. nodaļā apskatīts pats testēšanas process un apsvērtas automatizēšanas iespējas.

2. nodaļā tiek apskatītas konkrētas problēmas, ar kurām nāksies saskarties un kādas lietas būtu jātestē tieši tīmekļa aplikācijām.

3. nodaļā tiek apskatīti piedāvātie rīki automatizācijas veikšanai un aplūkoti arī konkrēti eksemplāri nedaudz sīkāk. Tiek veikta arī realizācijas metožu analīze un autora pieredzes apkopojums un secinājumi, veicot automatizētas testēšanas uzdevumus.

1 Testēšanas process

1.1 Produkta dzīves cikls

No kurienes rodas programmatūra? Tā rodas no idejas kāda cilvēka galvā. Ideja par to, kā radīt kaut ko patiesi nepieciešamu, vai arī vienkārši labi nopelnīt. Ir dažādi piegājieni, kā šo ideju attīstīt.

- Varam likt produkta izstrādātājiem un sistēmu analītiķiem izplānot, kā notiks procesi. Visas idejas tiek dokumentētas specifikāciju veidā, aprakstot sistēmas pamatelementus un funkcionalitāti. Var notikt neliela izpētes tipa programmēšana, lai pārbaudītu, vai šaubīgās idejas strādās. Dokumenti tiek pārlasīti, meklētas iespējās nepilnības. Kad visi papīri ir apstiprināti, sākas pamatstruktūras izstrāde, tiek pievienoti papildus pamata elementi. Šis procesa veids ļauj apzināt izstrādes termiņu un iesaistīt testēšanu daudz vairāk posmos, sākot jau no dokumentācijas.
- Var arī savākt kopā meistarīgus izstrādātājus un likt viņiem veidot produktu bez iepriekšējas plānošanas. Tiek veidots prototips, pievienojas jauni izstrādātāji, kuri pievieno jaunas idejas un vadībai tiek piedāvāts jau uzsākts projekts. Šādi var rasties nepilnības sistēmas uzbūvē, kas nav labojamas bez lielas daļas pamata funkcionalitātes labošanas.

Jebkurā gadījumā, lai turpinātu izstrādi, projektam ir jābūt skaidri definētam.

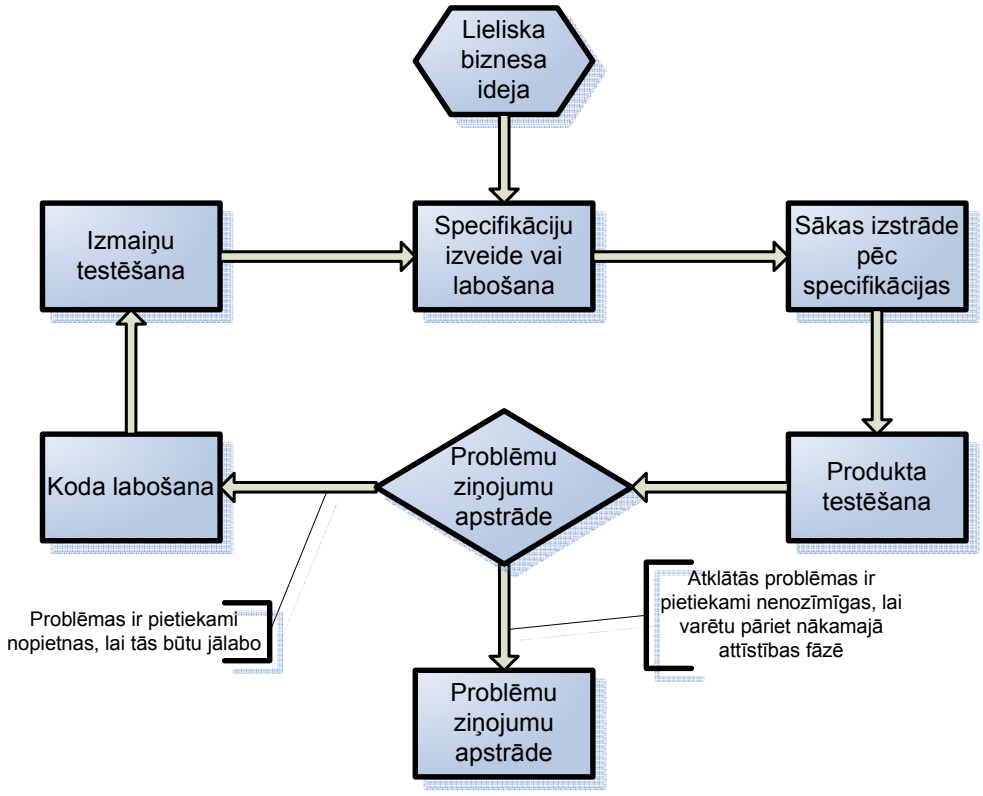
Tīmekļa aplikāciju specifikāciju izveidē nāksies pārklāt ļoti daudzas jomas, lai radītu patiesi labu produkta specifikāciju:

- Dizaina un lietojamības prasības. Tiešsaistes pakalpojumiem ir jābūt vienkāršam dizainam. Klientu zaudēšana tikai nesaprotama dizaina dēļ ir slikts biznesa plāns. Labāk atmest lieku izskata uzlabojumu, ja tas atvieglo lietotāja iespēju iegādāties pārdodamo pakalpojumu.
- Ātrdarbības un mērogojamības problēmas ir nopietns drauds jau piesaistīto klientu zaudēšanā.

- Lietotāju grupu identifikācija. Ir jāpiedāvā tādas lietas, kas piesaista audiences uzmanību.
- Informācijas atjaunošana – cik bieži mainīsies lapas saturs, kad tiks atjaunoti produkti. Tīmekļa spēks ir svaigumā, kvalitātē un stabilitātē, nevis kastītēs sapakotos produktos.

Specifikācijā ir jāparādās visiem šiem sīkumiem. Tas tikai palīdzēs izstrādes un savlaicīgas testēšanas procesā.

Arī pašas specifikācijas ir vērts testēt, jo ātrāk novērsta kļūda ir daudz lētāka. ASV Komercdarbības departamenta pētījums [2] rāda, ka no programmatūras kļūdām radušies zaudējumi ASV ekonomikai nodara 59,5 miljardus dolāru lielus zaudējumus gadā, kurus varētu samazināt par trešdaļu, veicot konkrētus uzlabojumus procesā.



Attēls 1-1 – Produkta dzīves cikls

1.2 Kad automatizēt testus?

Strādājot ar testēšanu, rodas vēlme automatizēt pēc iespējas vairāk testu. Ir labi veikt vienu kārtīgu testu, bet ja nu programmētājs nedaudz izmaina kodu un tas ir vēlreiz jāpārtestē? Ja tas netiek veikts, tā ir testētāja vaina. Tieši tāpēc rodas vēlme automatizēt pēc iespējas vairāk, taču tā ir otra galējība – automatizēt testu, kas izpildīsies vienu reizi vienmēr sanāk dārgāk laika izteiksmē. Ir jāizdomā, kā izlemt, kurus no testiem ir nepieciešamas automatizēt un kuriem tas iznāk nelietderīgi.

Ir skaidrs, ka ir jāautomatizē daļa testu, ir pieeja testēšanas rīkiem, ir neliela pieredze darbā ar tiem. Skaidrs, ka slodzes testēšana daudz efektīvāk izdosies, veicot to automatizēti. Vairāku simtu lietotāju savākšana iekšējas tīmekļa aplikācijas testēšanai nav praktiski realizējams uzdevums. Taču ar citiem testu veidiem tas vairs nav tik vienkārši izlemt, vai tas būs izdevīgi izmaksu ziņā. Tāpēc derētu atbildēt uz Braiena Marika (*Brain Marick*) piedāvātajiem jautājumiem [1]:

- Testa automatizēšana un vienreizēja izpilde izmaksā vairāk. Cik vairāk?
- Automatizētajam testam ir galīgs mūžs, kurā tam ir jāatpelna tā veidošanai veltītais laiks. Vai tas notiks agrāk vai vēlāk? Kādi notikumi to var iespaidot?
- Tā dzīves laikā, cik reāla ir iespēja, ka tas atradīs papildus kļūdas (ārpus tām, kam tas tika rakstīts pirmoreiz)? Kā šis iegumums atsver automatizācijas izmaksas?
- Cik labojumus, ko izraisījušas lielas pārmaiņas kodā pārdzīvos šis konkrētais tests? Ja ir aizdomas par nelielām izmaiņu iespējām, šim testpiemēram ir jābūt ļoti labam, lai izšķirtos par tā automatizēšanu.

Ja šie jautājumi nav pietiekami, tad var ņemt vērā arī citus faktorus.

Runājot par tīmekļa aplikācijām, ir jāņem vērā, ka dažos gadījumos automatizācija ir mazāk izdevīga, kā sākumā šķistu. Šeit gan rezultāti un iztērētais laiks ir atkarīgi no konkrētā testēšanas rīka izvēles.

- Ja automatizācijas scenāriji tiek balstīti uz lietotāja saskarnes elementiem, tad, veidojot ar roku šos scenārijus, tiek zaudēts ļoti daudz laika

- Ja šos scenārijus ģenerē, vienu reizi veicot šīs darbības un veicot to ierakstu, pastāv risks kļūdīties, kas šo ieraksta procesu liktu sākt no sākuma
- Automatizējot testus, ir jāveic arī instrukciju un dokumentācijas veidošana, kas ļautu šos testus izpildīt citiem testētājiem. Automatizētas testēšanas gadījumā darbs ir lielāks, tāpēc jāņem vērā, ka var rasties papildus laika zaudējumi
- Automatizējot testus, tiek atmesta daļa manuālo testu. Ja tā nenotiktu, automatizācijai, kas ir laika ietaupīšanas pasākums, jēgas nebūtu. Diemžēl manuālie testi parasti pārklāj problēmu daudz pilnīgāk, tāpēc ir jāizvērtē, vai kāda testa atmēšana neradīs pārklājuma zudumu kādā aspektā

Runājot par automatizēto testu dzīves cilu, neaizmirsīsim, ka testu īstā vērtība parādās pēc izmaiņām kodā. Ja neņemam vērā stresa vai noslodzes testus, testu kopumu atkārtota izpilde rezultātus nedos. Izmaiņas kodā reizēm būs tik nopietnas, ka tās izsauks testa sabojāšanu. Testu ir vai nu jāaizmirst, vai arī jālabo. Šeit varam pieņemt, ka testa labošana aizņems aptuveni tik pat daudz laika [1], kā aizņēma šī testa izveidošana. Lai arī sākumā tas tā nešķiet, tomēr prakse rāda, ka testa izpētes laiks, neatkarīgi, vai tas ir scenārijs vai ģenerēts ieraksts, aizņems to pašu sagatavošanas laiku, ko aizņēma testa izveide.

Tīmekļa aplikāciju testēšanā ir viens nozīmīgs pluss, kas attiecas uz izmaiņām lietotāja saskarnē. Visa informācija tiek sūtīta un saņemta teksta formātā un nekādas izmaiņas kļūdu paziņojumu izskatā vai ievades lauku pārmaiņas nevar izsaukt grūti novēršamu testu sabojāšanu. Darba virsmas aplikācijas kļūdu paziņojuma nomaiņa no izlecoša dialogloga uz kļūdainā ievadlauka iekrāsošanu sarkanā krāsā var sabojāt testa piemēru un padarīt izmaksu ziņā neefektīvu tā salabošanu. Tomēr parādoties jaunām tehnoloģijām tīmekļa aplikācijās – asinhronai datu pārraide un to attēlošanai ar scenāriju valodu *JavaScript*, var rasties pietiekamas izmaiņas, lai rastos jautājums par testu piemēru labošanas efektivitāti.

2 Tīmekļa aplikāciju testēšana

Izstrādājot tīmekļa aplikācijas, tiek izmantota filozofija, kas liek visas jaunizveidotās iespējas pēc iespējas ātrāk un biežāk ieviest lietošanā. Princips „Izlaist agri, izlaist bieži" [3] tika izmantots Linux sistēmas izstrādes gaitā. Šis piegājiens rada lielu iespējamību, ka parādīsies nopietnas kļūdas, kuras nebūtu iespējamas pie produktu izlaides ar termiņu ierobežojumiem, kad zinot, ka tuvojas izlaides datums, visa izstrāde tiek veltīta stabilitātes uzlabošanai, nevis jaunu iespēju izstrādei. Taču tīmekļa vide ir ļoti dinamiska un iespējas neizlaišana var izmaksāt pārāk dārgi. Šie apstākļi spiež izvēlēties starp vairākiem variantiem. Sistēmu var netestēt un, nezaudējot laiku, nodot pasūtītājam. Tas ir ļoti riskants process. Var veikt automatizētos regresijas testus vai testēt manuāli. Šeit skaidri saskatāma automatizēto testu priekšrocība, kas izpaužas faktā, ka pirms katras no biežajām produkta versijām, tam vajadzētu tikt notestētam.

Tīmekļa aplikācijas atšķiras no ierastajām grafiskās lietotāja saskarnes aplikācijām ar dažādu specifisku pieeju lietām, par kurām nav jāuztraucas standarta Windows, MacOS vai kādā mazāk populārā grafiskā operētājsistēmā.

Testēšanai izdalīsim šādas atsevišķas tīmekļa aplikācijas sastāvdaļas [4]:

1. Vienību testēšana attiecas uz kodu, kas ģenerē lietotāja saskarni vai realizē biznesa funkcijas servera pusē. Principiāli tīmekļa aplikāciju testēšanā tā neatšķiras no testēšanas jebkurai vispārīgi konkrētai aplikācijai vai tās modulim un ir saprotama kā testēšanā vispārzināmā vienību testēšana.
2. Integrācijas testēšana, līdzīgi kā vienību testēšanas gadījumā notiek analoģiski ierastajai integrācijas testēšanai – tiek pārbaudīts, kā individuāli notestētie moduļi darbojas, ja tie ir savienoti ar citiem moduļiem.
3. Dūmu testēšana notiek pēc katra izstrādes posma beigām un tajā tiek pārbaudīts, vai strādā galvenās funkcijas – kaut kur nav redzami „dūmi". Tīmekļa aplikāciju testēšanā būtiski neatšķiras no ierastās definīcijas.
4. Funkcionālā testēšana ir plašāk izmantotais testa veids tīmekļa aplikācijās, jo spēj atklāt procentuāli visvairāk kļūdu, ņemot vērā, ka tīmekļa aplikācijas vairumā ir projekti ar nelielu budžetu, kurā testēšanai tiek atvēlēts mazāk budžeta. Šī testēšana ir jādala divās daļās – lietotāja puses un servera puses testēšanā. Funkcionālā testēšana pārbauda individuālas

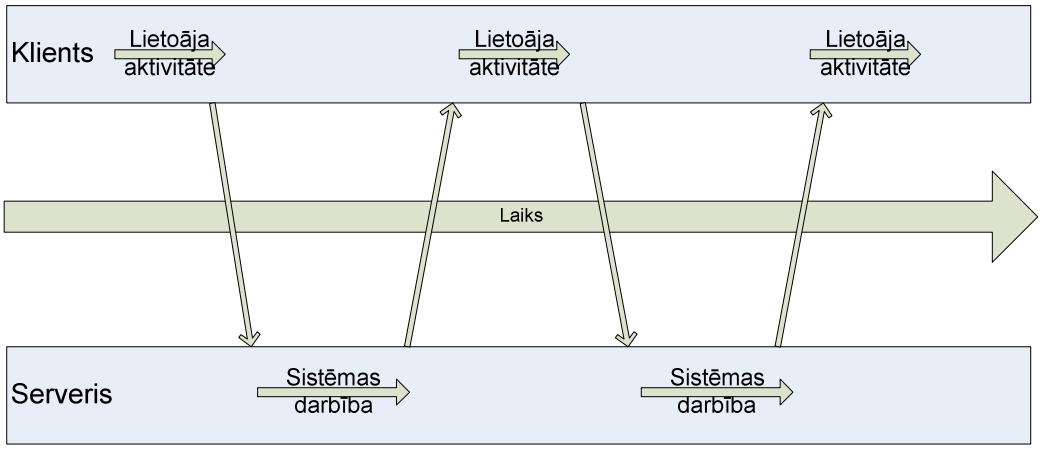
funkcijas no lietotāja skata korektu darbību. Tā var sastāvēt no daudzām sistēmas puses funkcijām, par kuru darbību lietotājam nav ne mazākās nojausmas.

5. Noslodzes testēšana apskata sistēmas uzvedību pie liela lietotāju un pieprasījumu skaita. To var darīt, veicot reālus eksperimentus, lai arī diezgan vienkārši to ir paveikt ar gatavu automatizācijas programmatūru. Šis testēšanas veids ir svarīgs tīmekļa aplikācijām, kas būs brīvi pieejamas plašākam apmeklētāju skaitam, jo norādes uz resursu ievietošana kādā citā populārākā resursā ar lielu lietotāju skaitu vari izraisīt t.s. *Slashdot efektu* [3] - apmeklētāji no otra resursa. Piemēram, ziņa 2006.gada 29. aprīlī par to, ka kompānija Google piedāvās nesen iegādātās 3D modelēšanas rīka Sketchup bezmaksas versiju, izraisīja lapas apmeklējuma pieaugumu (daļēji objektīva informācija [6]) par 1600% [7].
6. Stresa testēšana veic atsevišķu funkciju intensīvu darbināšanu, lai atrastu vājos punktus un sašaurinājumus. Ja noslodzes testēšana vairāk testē lapas funkcionālu darbību pie liela pieprasījumu apjoma, tad stresa testēšana pārbauda atsevišķu funkciju darbību.
7. Ātrdarbības testēšana nosaka izmaksas konkrētām darbībām vai funkcionalitātei. Parasti projekta līgumā tiek norādītas minimālās prasības ātrdarbībai, ko arī pārbauda šī testēšanas metode.
8. Lietojamības testēšana pārbauda, vai tīmekļa aplikāciju varēs izmantot tai paredzētā publikas daļa. Šī testēšana iekļauj gala lietotāja līdzdalību, taču ir lietas, ko var veikt arī automatizēta testēšana.
9. Pieejamības testēšana *accessibility* nozīmē pārbauda, vai lietotāji ar redzes vai dzirdes problēmām varēs izmantot konkrēto lapu. ASV valsts iestāžu tīmekļa lapām ir jāatbilst likumā noteiktām prasībām par lapas pieejamību [8].
10. Uzstādīšanas/novākšanas testēšana pārbauda, cik vienkārši un bez kļūdām ir iespējams uzstādīt sistēmu dažādās vidēs, servera konfigurācijās, kā arī kādas ir sistēmas novākšanas sekas. Tīmekļa programmām šie procesi notiek retāk, taču tāpēc nav mazāk nozīmīgi.
11. Atgūšanas testēšana pārbauda sistēmas spēju ātri atjaunot darbību pēc pašas sistēmas vai aparatūras kļūdas vai avārijas.

12. Savietojamības testēšana pārbauda sistēmas savietojamību gan klienta, gan servera pusē. Ja klienta pusē tas ir servera operētājsistēmas atbalsts, cita programmatūra vai sistēmas arhitektūra, tad klienta pusē ir daudz iespēju problēmām dažādu pārlūku atbalstā dažādiem tīmekļa standartiem, valodu uzstādījumos, teksta attēlošanā no labās uz kreiso pusi (ivrits, arābu rakstība).
13. Servera žurnāla un ziņojumu testēšana iekļauj pārbaudi, vai atkļūdošanas režīmā tajā tiek veikti detalizēti ieraksti par kļūdas dabu un, atkarībā no sistēmas, cita tehniska rakstura informācija. Arī produkcijas režīmā žurnālā būtu jāparādās ieraksti par neveiksmīgām transakcijām un tamlīdzīgām problēmām.
14. Drošības testēšana ir sarežģīts process, kas ir grūti, ja ne neiespējami, pilnībā automatizējams. Tas iekļauj servera programmatūras izpēti, un koda auditu.
15. HTML validācijas testēšana vai rezultāta HTML dokumenta atbilstības pārbaudi norādītajam SGML dokumenta tipam. Dokumenta atbilstība tipa aprakstam var palīdzēt tikt galā ar daļu no savietojamības problēmām un sniedz arī citas priekšrocība. Labi strukturēts HTML dokuments var palielināt lapas vērtējumu interneta meklētājos, kopā ar CSS izmantošanu samazināt dokumenta izmēru un atstāt iespēju nākotnē šos strukturētos datus automatizēti izmainīt.
16. Saišu testēšana veic pārbaudi, vai visas HTML dokumentā ievietotās saites korekti norāda uz citiem tīmekļa resursiem
17. SSL testēšana, ja tas tiek izmantots, testē SSL drošā savienojuma uzvedību.
18. Datu pārraides testēšana veic lapas pārbaudi pie liela (lokālais tīkls) un neliela savienojuma (iezvanpieeja caur ugunsmūri vai starpniekserveri) ātruma.
19. Atmiņas sūču testēšana pārbauda izmantotās atmiņas atbrīvošanu, gan servera pusē, gan tagad jau arī klienta pusē, kur problēmas var rasties, izmantojot *JavaScript*, *ActiveX* un citas klienta puses tehnoloģijas. Šeit gan daļa problēmu rodas pārlūku dēļ kuru problēmas netika risinātas, jo plašāka *JavaScript* pielietošana notiek tikai pēdējo gadu laikā.

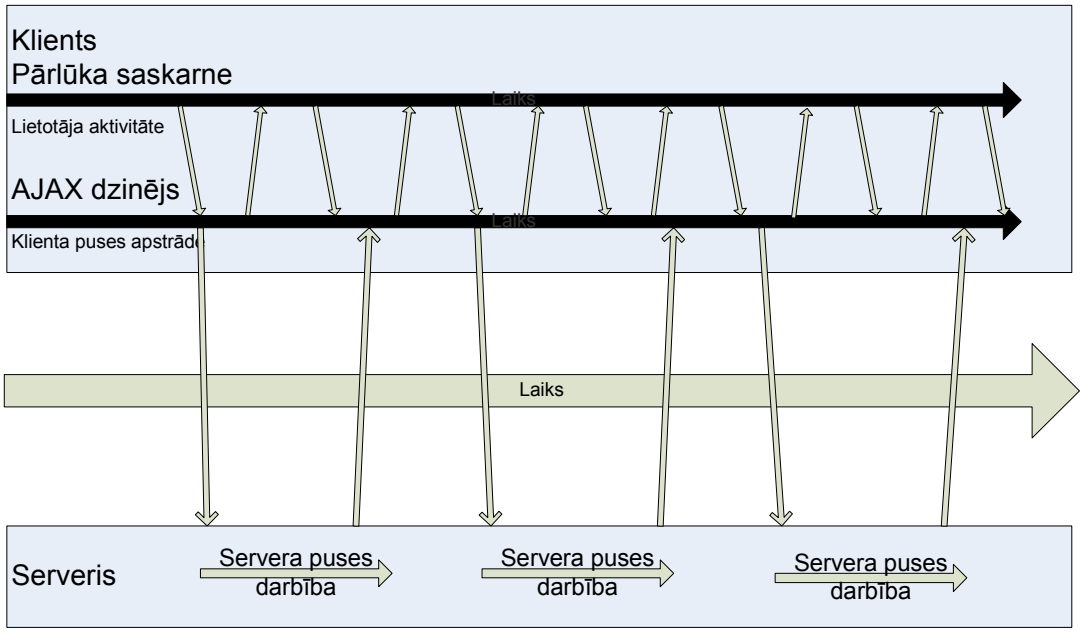
20. Lokalizācijas un internacionalizācijas testēšana nepieciešama tīmekļa aplikācijām, kurās ir paredzēts vairāku valodu atbalsts un pielietojums pasaules mērogā. Šeit varam atsevišķi izdalīt šādus uzdevumus:
    a. Starptautiskā pieejamība nozīmē, ka projekts var tikt lokalizēts visām mērķa valodās, valodās, kuras raksta no labās uz kreiso pusi, valodas, kurās simbola kodēšanai jāizmanto divi baiti, valodas, kurām nepieciešams lielāks minimālais teksta simbola izmērs. Protams, ir jāveic arī paša tulkojuma pārbaude, bet no programmatūras izstrādes viedokļa svarīgāk ir pārbaudīt, vai kādā lokalizētajā versijā valodas īpatnību dēļ nemainās lapas vizuālā struktūra, nenotiek kādas nobīdes vārdu garuma dēļ.
    b. Lokalizācija nozīmē, ka lietotājs visu viņam piedāvāto saskarni redz savā izvēlētajā valodā bez kļūdām
    c. Internacionalizācija nodrošina, ka sistēmā ir iespējams ievadīt visus simbolus, un simbolu kombinācijas un notiek pareiza ievaddatu pārbaude, nenotiek datu bojājumi vai aizvietošana. Iespējamās problēmas ir simbolu kodēšana ar vairākiem baitiem, kas var atsaukties uz ievaddatu pārbaudes funkcijām vai datubāzes lauku garumu.

Tīmekļa programmu datu plūsma no lietotāja uz serveri datu plūsma notiek pa tīkla savienojumu, izmantojot HTTP protokolu. Dati tiek sūtīti nepārprotamā stingri noteiktā protokolā, kas lieliski noder testēšanai. Attēls 2-1 uzskatāmi parāda normālu sistēmas lietošanas plūsmu, lietojot tīmekļa aplikācijas. Lietotājs staigā pa lapu, spiež uz hipersaitēm, iesniedz formas un saņem attiecīgo dokumentu. Šeit paveras iespējas ierakstīt šo datu plūsmu un atskaņot, izpildot automatizācijas testu. Pluss šai metodei ir tāds, ka nav iespējamas nozīmīgas variācijas šajā datu plūsmā, lietojot atšķirīgus lietotāja vides parametrus un konfigurācijas. Mīnuss ir tāds, ka nevaram pārbaudīt sistēmas darbību katrā no šīm konfigurācijām.



Attēls 2-1 – Sinhronais tīmekļa aplikācijas modelis

Alternatīvs tīmekļa aplikāciju darbības veids ir tehnoloģija, ko sauc par AJAX (Asynchronous JavaScript + XML) [9]. Tā izmanot Microsoft ieviesto JavaScript objektu XMLHttpRequest [10], kurš var veikt HTTP pieprasījumu, neizraisot lapas atkārtotu ielādēšanu, kas shēmā (Attēls 2-1) redzama kā pārtraukums lietotāja aktivitātē. Šis process psiholoģiski ir lietotājam traucējošs, tāpēc AJAX tehnoloģija iegūst arvien lielāku popularitāti un tā tiek apsvērta, izstrādājot jaunus projektus.



Attēls 2-2 – Asinhronais (AJAX) tīmekļa aplikācijas modelis

Kā shēmā (Attēls 2-2) redzam, kad ir ielādēta sākotnējā lapa, līdz ar to tiek uzstādīts AJAX dzinējs, kurš veic HTTP pieprasījumus no servera un apstrādā saņemtās atbildes, dinamiski ievietojot lapā saņemtos rezultātus. Lietotājs saņem

daudz patīkamāku sistēmas darbību, izstrādātājam ir nedaudz vairāk darba, jo daļa no tā tiek pārnesta uz klienta pusi. AJAX tehnoloģija ļauj saglabāt lapas funkcionalitāti arī, ja klienta pārlūks neatbalsta JavaScript, tad darbība notiek pēc ierastās shēmas.

Testētājam šī sistēma lielas galvassāpes nerada, un saglabājas arī automatizācijas iespējas, jo tieši tāpat notiek pieprasījumi uz serveri. Iespējams, tiek zaudēta pārskatāmība, taču netiek radīti nopietni šķērši automatizācijas veikšanai.

Otrs variants ir pārbaudīt nevis saņemto HTTP ziņojumu, bet izmantot jau gatavo HTML dokumentu. Šajā gadījumā mēs varam aplūkot darbību konkrētās vidēs, operētājsistēmās un pārlūkos. Tīrā veidā šī metode netiek izmantota, jo daudz ērtāk ir izmantot HTTP statusa kodus, nevis meklēt tos dokumentā.

Praktiski gandrīz netiek izmantota metode iespēja veidot ekrānuzņēmumus no renderētā HTML dokumenta. To mēdz izmantot, tikai, lai apskatītu, kā izskatās sistēma citās, iespējams izstrādātājam nepieejamās konfigurācijās. Taču automatizācijas rīki ar šādu funkcionalitāti netiek veidoti.

2.1 Testēšanas automatizācija

Automatizācija ir lielisks veids, kā nodrošināt, lai programmatūras šodienas versija būtu tik pat kvalitatīva, cik tā bija vakar. Citiem vārdiem, automatizācija nav nekāds brīnumrīks, kas testētājam ļauj aizmirst par savu darbu, bet gan veids, kā izvairīties no regresijas kļūdām un automatizētā veidā veikt liela apjoma dažādu datu ievadi. Kā visi citi testēšanas paņēmieni, automatizācija ir jau jāsāk plānot, programmatūras izstrādes laikā. Tajā brīdī, kad jau ir iespējams rakstīt pirmos automatizācijas scenārijus (izstrādāts stabils API), kļūdas, kuras varētu noķert automatizācijas scenāriji, visdrīzāk jau būs noķertas un izlabotas. Vēl daudzas no tām tiks atrastas, rakstot pašus automatizācijas scenārijus. Izpildot jau gatavos scenārijus, prakse rāda, ka diez vai tiks atklāta vēl kāda jauna kļūda. Jaunas kļūdas varētu tikt atrastas, kad automatizācija tiek palaista ar citādākiem vides uzstādījumiem vai ievadot datus no kāda masīva.

Automatizāciju varam iedalīt divu tipu testu veidos [4]:

1. Konstrukcijas pārbaudes testos (*Build verification tests, eng*), kuri ir automatizēti testi, kuri tiek izpildīti, lai noteiktu, vai ieviestās izmaiņas vai jaunradītais kods ir gatavs testēšanai vai citiem

mērķiem. Tiek pārbaudīta tikai pamata funkcionalitāte. Var izmantot arī pirms koda izmaiņu apstiprināšanas kopējā repozitorijā, lai pārliecinātos, ka nav nevēlamu blakus efektu.

2. Regresijas testēšana pārbauda scenārijus, kas iepriekš ir strādājuši. Šādi var atklāt vai pievienotā funkcionalitāte vai labojums nerada blakus efektus vai nesabojā strādājošu kodu. Šādā veidā atklātās kļūdas sauc par regresijām.

Tātad automatizācijas ieguvums nav uzreiz pamanāms laika nozīmē, jo pietiekami daudz laika ir jāpatērē, lai sagatavotu testa piemērus un scenārijus, kas tiks izpildīti, izmantojot šo testēšanas papildus procesu.

Automatizācija tikai nozīmē vienkāršu testa scenāriju izpildīšanu, kas neprasa cilvēka līdzdalību testēšanas procesā, kas tiek izpildīts simtiem reižu dienā pie katrām nozīmīgākām izmaiņām, ļaujot veltīt testētāja radošo potenciālu sarežģītāku problēmu risināšanai. Tiesa, šis scenārijs katru reizi tiks izpildīts identiski bez jebkādas jaunrades. Tāpēc ir jāsagatavo vairāk testa piemēru, lai pārliecinātos, par pilnīgāku problēmas pārklājumu. Izņēmuma gadījums ir testa piemēri daļēji tiek ģenerēti no koda vai datu ievadam tiek izmantots sagatavots datu masīvs, kas var mainīties. Statistika rāda, ka izpildot regresijas testus, tiek atklāts līdz 30% kļūdu [11], lai gan citi avoti uzrāda arī mazāku skaitu.

2.1.1 Automatizācijas veidi

Automatizācija var notikt dažādos veidos. Varētu iedalīt šādi, ņemot vērā produkta attīstības stadiju:

- Pašā projekta izstrādes sākumā tiek izstrādāts savs API, kuru tad arī var sākt testēt, tiešā veidā veicot metožu/funkciju izsaukumus šajā izstrādes posmā intensīvi testē gan paši izstrādātāji, gan testētāji, kuri gatavo automatizētos testus. Derīga būtu apmaiņa ar gataviem testa piemēriem. Elementāriem tīmekļa projektiem šī fāze var izpalikt, ja tiek izmatots gatavs dzinējs un veikti tikai saskarņu izsaukumi
- Vēlākās produkta fāzes automatizētā testēšana var balstīties jau uz augstākas funkcionalitātes esamību. Tīmekļa vidē šajā fāzē produkts jau var saņemt pieprasījumus no tīmekļa pārlūka un ir pieejami galvenie lietotājs saskarnes elementi. Daudzos risinājumos netiks izmantos pārlūks, bet informācija tiks sūtīta zemākā līmenī un

apstrādātas atbildes un atgrieztie HTTP atbildes kodi. Šāda automatizācija ir ātrāk veicama, bet šādi netiek testēta lietotāja saskarne un elementu izkārtojums. Taču tas jebkurā gadījumā ir cilvēka veiktas testēšanas uzdevums un nav pilnībā automatizējams.

- Citi automatizācijas veidi iekļauj ātrdarbības, mērogojamības un stresa testēšanu. Šeit netiek vērtēta aplikācijas korekta uzvedība vai lietotāja saskarne, bet gan aparatūras spēja apkalpot pieprasījumus un šauro vietu meklēšana kodā. Mērāmie lielumi šeit varētu būt centrālā procesora noslodze, atbildes laiks vai pārsūtīto datu apjoms, kas ļautu spriest, vai aplikācijas uzvedība ir pieļaujamās robežās. Šī automatizācija varētu sastāvēt no dažādiem iespējamiem pieprasījumiem no lietotāja pārlūka un dažādiem parametriem, kas modelētu noteikta lietotāju skaita uzvedību lapā. Šeit varētu mērīt vienlaicīgi iespējamo lietotāju neilgā laika periodā (pīķi) vai ilgākā (pastāvīga pārslodze), kas noteiktu sistēmas spēju izturēt noslodzi, laika vienībās.

2.1.2 Automatizētās testēšanas plāns

Svarīgākā automatizācijas sastāvdaļa ir plāna izveide. Iesākumā būtu jāatbild uz šādiem jautājumiem:

- Kāds ir automatizācijas mērķis?
- Ko tu vēlies darīt?
- Kādu rezultātus tu gribi panākt?

Plānam vajadzētu aptvert gan augsta līmeņa pārskatu, gan specifiskus sīkumus, katra automatizācijas soļa un procesa dokumentāciju. Jāizveido ne tikai darba grafiks, bet arī resursu sadale.

Plānojot vajadzētu ņemt vērā šādas lietas:

- Uz kādām darba stacijām tiks veikti testi? Vai būs specifiski testiem veidota vide, vai nāksies dalīt resursus ar izstrādes vidi.
- Vai testēšanas programmatūra būs centralizēta, vai vajadzēs to uzstādīt uz katras darba stacijas un rūpēties par tās individuālu atjaunošanu?
- Kas rakstīs testus? Vai automatizāciju veiks atsevišķa persona, komanda, vai tos rakstīs individuālais iespējas izstrādātājs? Atšķirība

rodas tur, ka, ja automatizāciju veiks katrs izstrādātājs atsevišķi, visiem būs jāapgūst automatizācijas rīki, bibliotēkas un dokumentācija.

- Kas izpildīs automatizāciju? Vai to darīs katrs atsevišķi, vai tas notiks centralizēti?
- Vai būs pieejamas datubāzes, izejas koda versiju kontrole, cita veida infrastruktūra, kas paredzēta testēšanai?
- Vai testēšanai tiks atvēlēta atsevišķa nodaļa? Ja nē, testu efektivitāti var palielināt, ieviešot dažādību testiem izmantojamo darbstaciju konfigurācijā.

Automatizācijas plānam vajadzētu būt sadalītam diskrētās sadaļās, lai katru no tām varētu izpildīt pēc noteikta izstrādes posma pabeigšanas. Plāna konkrētas detaļas var mainīties, bet galvenos vilcienus vajadzētu noteikt jau sākumā. Nevajadzētu ar automatizāciju sākt nodarboties pārāk ātri, kamēr vēl notiek lielas izmaiņas struktūrā, kas noved pie testu pārrakstīšanas. Taču arī automatizēšanas sākuma nokavēšanai ir negatīva ietekme, jo kādas pārāk vēlu atklātas kļūdas labošanai nepieciešams pārstrādāt lielu daļu sistēmas. Jāsāk būtu ar dūmu jeb konstrukcijas pārbaudes testiem, kas ir automatizējami, kad parādās strādājoša galvenā funkcionalitāte. Turpmāk jau būtu jāveido katras apakšfunkcionalitātes individuālās testēšanas piemēri, kuriem būtu jābūt darba kārtībā neilgi pēc šo iespēju ieviešanas.

2.2 Universālā testēšanas vide xUnit

Daudzu testēšanas rīku pamatā ir pēc xUnit parauga būvēta pamata funkcionalitāte, tāpēc ir vērts to apskatīt atsevišķi. Protams, ir iespējams veidot ļoti vienkāršas konstrukcijas individuāliem testēšanas gadījumiem un neizmantot speciāli veidotās testēšanas vides, taču praksē biežāk izmanto gatavus risinājumus.

xUnit ir dažādu koda bāzētu testēšanas ietvaru (*framework*) kopums, kurš apvieno dažādām objektorientētām programmēšanas valodām radītus testēšanas ietvarus. Pirmo šāda tipa ietvaru uzveidoja Kents Beks (*Kent Beck*) valodai Smalltalk. Tagad šādi ietveri ir izveidoti visām nozīmīgākajām valodām – jUnit Java valodai, cUnit – C valodai, CppUnit – C++ valodai. Šim ietveram ir šādas īpašības [20]:

- Standartizēta kopa ar no valodas neatkarīgiem jēdzieniem un konstrukcijām
- Atbalsts gandrīz jebkurai valodai
- Atvērts standarts
- Visi ietvari atbalsta noteiktas konstrukcijas
    - *TestCase* un *TestSuite* ir identiskas visos risinājumos
    - Pārbaudes konstrukcija *assertEquals(comment, expected, actual)* ir būvēta pēc vienota standarta
- Ir konstrukcija *TestRunner*, kura izpilda testus

xUnit ir vide, kurā var veikt vienību testēšanu, taču uz tā bāzes ir radīti arī diezgan daudzi funkcionālās testēšanas rīki. Populārākā xUnit vide ir Java risinājums jUnit, kuram ir radīti daudzi paplašinājumi – DBUnit (speciāli risinājumiem ar datubāzēm, kas ir arī gandrīz visos tīmekļa aplikāciju projektos), XMLUnit (lai salīdzinātu rezultāta XML failus, kas ne vienmēr ir salīdzināšana pa atsevišķam simbolam), HTMLUnit un HttpUnit (speciāli tīmekļa aplikāciju testēšanai, atšķiras ar to, ka HTMLUnit apstrādā atgriezto HTML dokumentu, bet HttpUnit darbojas ar HTTP protokola datiem), Cactus, kurš paredzēts Java servletiem un atbalsta lietotāju autorizāciju, sesijas un sīkdatnes (*cookies*).

Vienību testēšanas videi būtu jāatbilst šādām prasībām:

- Katram testam jādarbojas neatkarīgi no visām citām testu vienībām
- Kļūdas ir jānosaka un jāziņo testu pa testam
- Ir viegli jāspēj definēt, kurus testus izpildīt

xUnit sastāv no [13]:

- *Test* saskarnes, kuru ir jāievieš visu veidu testu klasēm. Šī brīža realizācijās to dara *TestCase* un *TestSuite*.
- *TestCase* klases, kurā tiek rakstīts ietvars, kurš testēs objektu ar konkrētu testa piemēru kopumu pie noteiktiem apstākļiem, un pēc tam ar *assertTrue*, *assertEquals* vai kādu atvasinājumu konkrētā realizācijā tiek pārbaudīts apgalvojuma patiesums. Šajā klasē ir metodes *setUp()* un *tearDown*, kuras var inicializēt kādu stāvokli pirms testa un pēc tam to novākt.
- *assert* metodes atšķiras no vienkāršiem kļūdu paziņojumiem – nesekmīga *assert* metodes izpilde nozīmē to, ka rezultāti nav gaidītie, nevis notikusi kāda sistēmas kļūda. Veiksmīgas izpildes gadījumā testi aiziet tālāk, neveiksmes gadījumā tie izsaukta izņēmuma situācija. jUnit vidē tās ir *assertTrue*, assert*False, assertEquals, assertNotNull, assertNull, assertSame, assertNotSame.*
- *TestResult* klases, kura saturēs neveiksmīgo un kļūdaino testu rezultātus
- *TestSuite* klases, kura saturēs individuālos *TestCase* objektus vai arī citas *TestSuites* – tā ir *Test* kompozīcija kokveida struktūrā.
- *TestRunner* klases, kura izpilda izveidoto *TestSuite*. Tā ir testēšanas sistēmas saskarne ar tā lietotāju, tā izpilda testu un atgriež rezultātus lietojam. Visus esošos *TestRunner* objektus apvieno *BaseTestRunner*.
- *TestListener*, kas klausās *TestRunner* atgrieztos rezultātus – testa sākumu, beigas un atgrieztos rezultātus

2.3 Testēšanas virzieni

2.3.1 Aplikācijas moduļu vienību testēšana

Jebkura nopietna tīmekļa aplikācija savā pamatā ir būvēta uz objektorientētas programmēšanas principiem, lai arī tiek pietiekami plaši pielietota strukturālā programmēšana. Strukturālo programmas kodu var testēt līdzīgi kā objektorientēto, objektu vietā izmantojot funkciju izsaukumus, taču nozīmīgu kļūdu daudzumu šādi atkalāt nevar. Tāpēc strukturāli programmēto kodu vairāk praksē testē, izmantojot funkcionālo testēšanu, kas tiks aprakstīta nākamajā nodaļā pie lietotāja puses testēšanas.

Vienību testēšana ir procedūra, ko izmanto, lai pārbaudītu, vai konkrētais modulis strādā pareizi. Procedūrā ietilpst testa piemēru rakstīšana visām funkcijām un metodēm, lai pie jebkurām koda izmaiņām varētu radušos problēmu identificēt un labot. Vienību testēšanu parasti veic pats koda autors, izstrādājot savu programmas moduli.

Standartizētā procedūra tradicionālajās izstrādes metodēs nosaka, ka vienību testēšana sastāv no 3 fāzēm un 8 aktivitātēm [14].

1. Testu plānošana
    a. Paņēmienu, resursu un grafika plānošana
    b. Testējamo iespēju noteikšana
    c. Kopējā plāna uzlabošana
2. Testu kopu iegūšana
    a. Testu kopu plānošana un izveidošana
    b. Testa plāna un realizācijas ieviešana
3. Testa vienības novērtēšana
    a. Testa procedūru izpilde
    b. Testu izpildes statistikas konstatēšana
    c. Testa rezultātu novērtēšana

2.3.2 HTML dokumenta testēšana

HTML dokuments ir tīmekļa aplikāciju lietotāja saskarne, tāpēc tā testēšana ir pietiekami nozīmīga.

HTML ir valoda, ar kuru apraksta tīmeklī ievietoto informāciju. Tā atbalsta tekstu, attēlus, multivides objektus, dažādas scenāriju valodas, tabulas, formas, stilu lapas, un protams, hipersaites. Šī brīža pārlūki cenšas atbalstīt HTML standartu 4.01 [14], kurā pēdējās izmaiņas notikušas 1999. gada 24. decembrī. Tomēr katrs pārlūks šo un citas specifikācijas atbalsta nedaudz citādāk.

Vienkāršs piemērs ir izmēģināt atvērt dokumentu, kurā pārklājās dažādu iezīmju kods. HTML pretendē uz strukturēta dokumenta statusu, bet tas satur arī dažus elementus, kas īsti nepakļaujas šādai definīcijai, tāpēc to ir iespējams dažādi attēlot. Populārākā tīmekļa pārlūkprogramma Internet Explorer pieļauj diezgan lielas atkāpes no korektas iezīmju aizvēršanas, taču visam ir savas robežas.

Lai izvairītos no problēmām, ir jāizmanto standartizēts HTML, kas veidots saskaņā ar organizācijas W3C standartu 3.2 vai 4.01. Šī organizācija ir neitrāla organizācija, kas veido arī jaunus standartus. Jo vairāk kods atbildīs standarta, jo mazāk problēmu būs, to izmantojot.

HTML korektuma pārbaude notiek, salīdzinot to ar dokumenta tipa aprakstu, kas ir brīvi pieejams. Ir viegli iedomāties, ka šo pārbaudi var un vajag automatizēt un ir gatavi rīki, kas var pārbaudīt jebkura HTML dokumenta korektumu.

2.3.3 Renderētā dokumenta testēšana

Mūsu izveidotā sistēma var veidot pilnīgi korektu HTML dokumentu, būt funkcionāli pilnīga un nepieļaut kritiskas kļūdas, bet dažādās lietotāja vidēs var izskatīties pilnīgi atšķirīgi. Šeit pie vainas var būt gan atšķirīgā interneta pārlūku pieeja dažādu standartu realizācijā, gan dažādas operētājsistēmas vides īpatnības – fontu uzstādījumi, fontu izmēri, Unicode atbalsts. Tāpēc būtu svarīgi pārbaudīt sistēmas darbību pēc iespējas plašākā iespējamo klienta konfigurāciju klāstā.

Ja sistēmas izskats paliks nemainīgs lielāko tās darbības daļu, tā varētu būt vienreizēja darbība, bet, ja tā savas darbības laikā tiks nemitīgi pilnveidota, pievienojot jaunus grafiskus saskarnes elementus, šī testēšana būs jāveic atkal un atkal.

Ja apskatām pašu testēšanas procesu, šeit nav daudz ko iespējams automatizēt. Automatizēts skripts, kurš varētu izgatavot lapas ekrānšāviņus, nespēs noteikt, vai izmaiņas starp attēliem ir tāpēc, ka kāds ir ievietojis jaunu preses paziņojumu vai ir nobīdījies navigācijas bloks, tāpēc, ka uz zemas izšķirtspējas tam pietrūkst horizontālās ekrāna vietas. Šo pārbaudi ir jāveic cilvēkam. Protams, varētu iedomāties, ka ir iespējams izveidot apmācāmu algoritmu, kurš atšķirs lapas navigācijas daļu no galvenā satura, taču tikpat skaidrs ir, ka ieguldījumi izstrādē neatsvērs ieguvumus no testēšanas.

Tomēr ir lietas, kuras var atvieglot šajā manuālajā pārbaudē – varam veikt simulāciju dažādās vidēs un izgatavot šos ekrānšāviņus, kurus ziņojuma veidā iesniegt testētājā. Ir pieejami gatavi komerciāli risinājumi, piemēram BrowserCam [12], kuri piedāvā veikt pārbaudi uz visdažādākajām lietotāja konfigurācijām. Eksistē arī dažādi kopienas izstrādāti atvēri projekti, kuri piedāvā šo pakalpojumu bez maksas. Šī sistēma balstās uz brīvprātīgām privātpersonām, kuras uzstāda uz saviem datoriem sistēmas klientu, kurš veic daļu no pieprasījumiem rindas kārtībā. Ieguvums šeit ir finansiāls un, iespējams, arī plašāks lietotāja konfigurāciju skaits, taču šī sistēma balstās uz brīvprātības principu un pieprasījumi tiek apstrādāti rindas kārtībā. Problēmas var rasties iekšējiem projektiem, jo rezultāti pārsvarā ir publiski pieejami.

Jebkurā gadījumā šie testi ir veicami ar lielu cilvēka līdzdalību, taču minētie rīki var būt ļoti noderīgi un rezultāti viegli dokumentējami.

2.3.4 Funkcionalitātes testēšana

Iesākumā programmas darbojās teksta režīmā, lietotāji iemācījās komandas un ievadīja tās no klaviatūras. Programmas veica noteiktās darbības un atgrieza paskaidrojošu tekstu vai izvadīja rezultātu kādā failā. Šādu programmu testēšanu automatizēt bija samērā vienkārši. Līdz ar grafisko vižu ieviešanu un logu pārvaldnieku izmantošanu lieta sarežģījās. Lietotājam vairs nebija jāievada teksta komandas, bet jāspiež pogas, jāizvēlas dati no izvēlnēm un citas lietotājam ērtākas darbības. Automatizētiem testēšanas rīkiem tas radīja problēmas, taču ar laiku tās tika apietas, veidojot rīkus, kuri bija piesaistīti konkrētām platformām un operētājsistēmām, un veica izsaukumus, piemēram, pogas nospiešanu, piesaistot sevi izpildāmajām kodam. Lieki piebilst, ka uz vienas platformas veidotie testi nederēja citām platformām. Otrs mīnuss bija tāds, ka izveidotie testi nav labojami – ja ir vajadzība pamainīt ievadāmos datus, ir nepieciešam vēlreiz par jaunu

ierakstīt visu scenāriju. Tāpat izstrādes laikā pie mainīgas lietotāja saskarnes nebija īpaši izdevīgi veidot regresijas testus. Protams, laika gaitā parādījās papildus iespējas, kas atviegloja dažus aspektus [16].

Runājot pat tīmekļa aplikāciju testēšanu, šeit atkrīt daudzas problēmas. Ir divi galvenie veidi, kā mēs varam veikt automatizāciju. Mēs varam izmantot HTML dokumentu vai HTTP pieprasījumu un atbilžu saturu. Patiesībā liela atšķirība šeit nepastāv, jo HTML dokuments ir tikai pārveidots HTTP pieprasījuma ķermenis. Tiek pazaudēta daļa informācijas no HTTP pieprasījuma galvenes, bet tiek izmantota dokumenta struktūra, kas otrā gadījumā ir tikai *String* tipa mainīgais. Abos gadījumos ir savi ieguvumi.

2.3.5 Tīmekļa programmām specifiskie ievaddati

Nebūtu prātīgi automatizācijai likt darīt lietas, kas nedos praktiskus rezultātus. Ir tādi ievaddati, kuri nedos atšķirīgu rezultātus no nedaudz atšķirīgiem rezultātiem, tāpēc ievaddatus varētu sadalīt problēmas izraisošās klasēs.

Strādājot ar tīmekļa aplikāciju veidošanu un testēšanu esmu apzinājis iespējamos problemātiskos ievaddatus, kurus lietotājam ir pilnīgas tiesības ievadīt. Lai saprastu, kādi ievaddati ir bīstami sistēmas darbībai, ir jāizseko datu plūsmai, kādā veidā tā nonāk no lietotāja pārlūka tās galamērķī, parasti datubāzē vai faila ierakstā. Kad esam izsekojuši šo datu plūsmu, jāapskata, kādas tehnoloģijas tiek izmantotas katrā līmenī. Kāds no risinājumiem varētu būt šāds:

1. Lietotāja pārlūks. Pārsvarā lietotāja dati no lietotāja nāk vienotā formātā, taču var būt problēmas ar simbolu kodējuma noteikšanu, kurš var izrādīties neveiksmīgi noteikts un rezultātā tiek saņemti nekorekti dati.
2. Tālāk datus aplikāciju serveris nodod servera programmu videi, kas varētu būt Apache Tomcat (Java), PHP, ASP, Ruby, Perl, pl/SQL. Šeit problēmām nevajadzētu rasties, jo CGI mainīgo saņemšana ir droša operācija un par to rūpējas attiecīgās programmatūras izstrādātāji.
3. Tālāk ir visas iespējas rasties problēmām. Saņemtie dati var tikt ievietoti Oracle, PostgreSQL, MSSQL, MySQL datubāzē, kur problēmas var rasties, ja izstrādātājs neparūpējas par rezervēto simbolu un simbolu virkņu atdalītāju izmantošanu. Ja tiek izmantota pl/SQL integrācija ar Oracle datubāzi, to veic pašas Oracle iebūvētās mainīgo nodošanas

shēmas, taču šī drošība ir iemidzinoša, jo, ja tiek izmantotas iepriekš sagatavotas SQL izteiksmes, kur dinamiski tiek ievietoti parametri, pastāv tādi paši draudi, kā iepriekšminētajos gadījumos.

4. Turpinām otrā virzienā – datu ielasīšana. No datubāzes ielasīt mainīgajos datus izdosies droši un bez problēmām.
5. Taču nākamais solis – nosūtīšana lietotājam uz pārlūku un attēlošana tajā atkal satur draudus. Ja mūsu izvadāmais teksts satur rezervētos HTML simbolus, mēs lietotājam piedāvāsim izkropļotu attēlu, jo pārlūks neatšķirs mūsu mainīgos datus no īstajiem HTML datiem. Turklāt, ieliekot datus <asajās iekavās>, lietotājs šos datus vispār neredzēs.
6. Tomēr daudz bīstamāka ir iespēja mums nemanot ievietot izpildāmu JavaScript kodu, kas var izpildīties, citam lietotājam atverot lapu un, viņam pat nemanot, nozagt sesijas identifikācijas kodu, kas var tikt izmantots, lai iegūtu šī lietotāja konta informāciju.

Tāpēc esmu izveidojis praktiski testējamo ievaddatu sarakstu, kas būtu pakļaujams automatizācijas uzdevumiem.

1. Vispārīga formas iesniegšana testēs pilnīgi legālu lietotāja darbību, kas var notikt jebkuram lietotājam mēģinot aizpildīt datu ievades laukus.
    a. Tukša forma var būt nejauša pogas nospiešana
    b. Korekti aizpildīta forma būs visbiežākais gadījums un būtu nepiedodami atstāt kļūdu šeit.
    c. Aizpildīti obligātie lauki. Šādā gadījumā mēs pārbaudām, vai funkcionāli pareizi tiek veikta pārbaude uz ievadlaukiem. Ja attiecīgi pretī ir datubāzes lauks ar nosacījumu *not null*, ir jāpārliecinās, ka šie lauki tiešām ir aizpildīti.
    d. Nav aizpildīts 1 obligātais lauks, katram laukam atsevišķi – veicot melnās kastes testēšanu, mēs nevaram zināt, kā ir realizēta aizpildīšanas pārbaude, tāpēc ir jāpārbauda katrs atsevišķi.
    e. Gramatikas kļūdas, punkti, koli, izsaukuma zīmes paziņojumos īsti nav automatizējams uzdevums. Tiesa, pareizrakstības pārbaudi veikt automatizēti nebūtu problēmas.
    f. Vai pēc formas iesniegšanas esam pareizā lapā? Šo pārbaudi visbiežāk veic ar kāda konkrēta teikuma meklēšanu rezultāta lapā.

Tas ir paziņojums par veiksmīgu vai neveiksmīgu formas iesniegšanu.

2. Teksta lauki
    a. Teksta laukos ievadīti tukšumi (atstarpes). Saturiski formās ievadīt tukšumus nav īpašas jēgas, tāpēc parasti no ievaddatu galiem tiek nogriezti tukšumi.
    b. Teksta laukos ievadīts viens simbols reti kurā gadījumā būs korekti ievaddati. Pārbaudām, vai ievaddatiem tiek pieprasīts minimālais garums. Praktiski tiek izmantots lietotājvārdu un paroļu laukos.
    c. Ievadītā teksta galos lieki tukšumi ir mēģinājums sasniegt minimālo garumu. Nebūtu labi ļaut šādi apmānīt sistēmu.
    d. Pārāk garš teksts. Vērā varētu ņemt dažādas tehniskas konstantes – mainīgo garumus, datubāzes lauku garumus pēc datu tipa. Šeit noderētu zināšanas par datubāzes struktūru, lai efektīvāk to notestētu. Katrā ziņā, sistēmai vajadzētu nepieļaut šādu ievaddatu aizpildīšanu.
    e. Ievadīti Unicode simboli, kuru glabāšanai izmanto vairākus baitus. Pie reizes pārbaudām arī lapas simbolu kodējumu. Ja veidojam produktu Tālo austrumu tirgum, vienu simbolu var nākties glabāt pat 4 baitos. Šajā gadījumā tas ir jāņem vērā plānojot datu bāzes laukus un uzliekot garuma ierobežojumus HTML dokumenta ievadlaukiem.
    f. Ievadīti dažādi rezervētie simboli HTML (",<,>), SQL (‘,$,#) un citi katram risinājumam specifiski simboli, kuri var izraisīt kļūdas situāciju, ja nav veikti aizsardzības pasākumi. Ja HTML gadījumā var rasties tikai problēmas vizuālajā izskatā, tad SQL gadījumā ar *injekcijas* metodi bieži ir iespējams izpildīt vēl vienu SQL pieprasījumu, kas var radīt nopietnu apdraudējumu sistēmai.
    g. Pārbaudes uz vienkāršu HTML kodu. Ļoti bieži tīmekļa aplikācijās ir atļauta teksta trekināšana, pasvītrošana, sadalīšana rindkopās, kuru veic ar vienkāršu HTML kodu vai izmanto kādu citu teksta iezīmēšanas valodu, kura tiek pārveidota par HTML

h. Pārbaudes ar sarežģītāku HTML kodu. Ja lapas struktūra balstās uz tabulu izmantošanu lapas struktūras veidošanā, tad ļaujot lietotājam ievadīt *</table>,* struktūra var smagi sabojāties. Bīstami tas var būt komentāru sarakstā vai administrācijas daļā, kur formās ievadītie dati tiek rādīti tabulās.

i. Pārbaudes uz *JavaScript* ievadi. Lietotājam nekad nedrīkst ļaut ievadīt šādus koda gabalus, jo tas smagi apdraud sistēmu. Nedrīkstam aizmirst, ka šo kodu var ievadīt dažādos veidos, ne tikai izmantojot vienkāršus <script> tekstus. Rezervētos simbolus var ievadīt, pārveidojot to kodējumu HTML vai URL kodā, bet ielasot no datubāzes un veidojot jaunu dokumentu, tie pārvērtīsies par vajadzīgajām HTML iezīmēm.

3. Ciparu lauki. HTML formās nav speciāla elementa ciparu ievadei, tāpēc šie lauki ir parasti teksta lauki, kuri tikai paredzēti ciparu ievadei. Lai arī ar *JavaScript* palīdzību to var ierobežot, uz to nevar paļauties.
    a. Laukos, kur varētu būt tikai cipari (tālrunis, objektu skaits), mēģinām ievadīt citus simbolus. PHP nav svarīgs ievaddatu tips, jo mainīgajam var mainīt tipu brīvi pēc vajadzības, bet SQL teikumos skaitliskas vērtības parasti neatdala ar simbolu atdalītāju, kas var izraisīt kļūdas situāciju gadījumā, kad tiek ievadīta ne-cipariska simbolu virkne.
    b. Ievadām negatīvus skaitļus, nulli, kur tas nav pieļaujams. Pārbaudām, vai kādā formulā mēs nevaram iegūt neparedžetus rezultātus, piemēram norēķinoties interneta veikalā samaksāt negatīvu summu.
    c. Ievadām ļoti lielus skaitļus, pārsniedzam *Integer* tipa maksimālo konstanti.
    d. Ievadām decimāldaļskaitļus. Tie var neatbilst datu bāzes lauka tipam un vai nu tiks noapaļoti vai radīsies kļūdas situācija.
4. Failu augšupielāde. Jārēķinās, ka dažās automatizācijas sistēmās var būt problemātiska ievade. Taču tā kā joprojām tiek izmantots HTTP protokols, varam sūtīt datus arī šādā formātā.
    a. Iesniedzam tukšu lauku. Šādi veicam pārbaudi, vai sistēma reaģēs uz samērā bieži iespējamu gadījumu.

b. Failu augšupielādes formā ievadām nesakarīgu tekstu. Samērā reti iespējama lietotāja uzvedība, jo reti kurš neizmantos „Browse" pogu, lai izvēlētos failu. Taču jāpārbauda arī sistēmas uzvedība šajā gadījumā.
c. Failu augšupielādes formā izvēlamies lielu failu. Automatizācijas laikā tas nav lietderīgi, taču ir jāapzina iespējas, kādas ir šai sistēmai. Datubāzu binārā lauka izmērs mēdz būt pat mērāms gigabaitos un arī uz diska vietas ierobežojuma nav. Dažās sistēmās gan ir lietotāja ierobežojums uz aizņemto diska vietu.
d. Ja ir jāielādē noteikta tipa failus, tad mēģinām ielādēt cita tipa failus. Varam mēģināt arī nomainīt faila paplašinājumu.

5. Specifiska formāta lauki
    a. Reģistrācijas formā lietotājvārdā izmantojam simbolus, kas nav atrodami uz standarta klaviatūras ar angļu lokāles izkārtojumu. Lietotājvārdu un paroļu politika parasti aizliedz šādu simbolu izmantošanu.
    b. To pašu darām paroles laukā
    c. Paroles laukā ievadām to pašu tekstu, ko lietotājvārda laukā. Pēc līdzīgas shēmas var būt pārbaudāmi citi lauki, kuru saturu nosaka drošības politika
    d. E-pasta laukā ievadam e-pastam neatbilstošus simbolus. Šie ievaddati var izraisīt problēmas vēlāk, kad sistēma mēģinās nosūtīt uz nekorekti ievadīto e-pasta adresi kādu paziņojumu.
    e. Ja tiek prasīts ievadīt datus specifiskā formātā (piemēram, datumu, pasta kodu), ievadām nekorektā formātā, ja to traucē izpildīt JavaScript, atslēdzam to un izpildām. Tiesa, dažas automatizācijas sistēmas paļaujas uz JavaScript pārlūka vadīšanai, tāpēc šis drošības drauds var palikt nenotestēts.
    f. Ja izmantojam ģenerētas bildes ar tekstu (CAPTCHA), kurš jāievada, ievadām to nepareizi. Manuālās testēšanas uzdevums. Ja mēs spēsim izveidot algoritmu, kas atpazīs ģenerēto bildi, būs pienācis laiks uzlabot bildes ģenerēšanas algoritmu, kas atkal liks pilnveidot automatizācijas skriptu.

Kad esam noskaidrojuši, kādi iespējamie lietotāja ievaddati varētu izraisīt problemātisku mūsu tīmekļa aplikācijas darbību, varam apsvērt risinājumu, kā iekļaut šos ievaddatus automatizētajā testēšanas procesā.

Daudzi automatizācijas rīki piedāvā iespēju ievaddatus ņemt no kāda arēja avota – datubāzes, faila vai ģenerēt tos ar skriptu. Zinot šos ievaddatus, ko tikko apskatījām, varam izmantot paraugu (*pattern*, *eng*) veidošanu. Ja izmantojam savu iecienīto programmēšanas valdu, varam vienkāršot procedūru un paši izveidot ievaddatu masīvu, kuru aizpildīt pēc shēmas, lai pārklātu pēc iespējas vairāk potenciāli nederīgo ievaddatu. Protams, ir jāzina, kad apstāties, jo nav vērts automatizēt loģiski ļoti sarežģītus lietotāja ievaddatus, kuri ir atkarīgi no iepriekš ievadītajiem.

2.3.6 Scenāriju valodu testēšana

Populārākā scenāriju valoda, ko izmanto tīmeklī lietotāja pusē un ko atbalsta vairums tīmekļa pārlūku ir *JavaScript*. Tā ir interpretējama valoda un testēšanas mērķi būs:

- Pārbaudīt, vai nav sintaktisku vai citu kļūdu, kas neļautu kodu interpretēt
- Pārbaudīt, vai funkcijas vai JavaScript objekti veic savu korektu darbību
- Pārbaudīt, vai kods strādā uz dažādiem tīmekļa pārlūkiem

Sintakses kļūdas varam pārbaudīt ar dažādiem rīkiem, kas ir automatizējami. Liela daļa no rīkiem, kas automatizē HTML dokumenta korektuma pārbaudi, nodrošina arī iespējamo problēmu meklēšanu *JavaScript* kodā.

Šeit rodas lielākās problēmas ar testēšanu vispār, neņemot vērā pat automatizācijas idejas. Iegūtais kods ir jātestē pašā interneta pārlūkā. Protams, varam izveidot vai izmanto jau gatavu *JavaScript* interpretatoru, bet šāda veida pārbaudes nedos pilnīgu pārliecību par rezultātiem. Tas ir tāpēc, ka diezgan pamatīgi atšķiras konkrēto izstrādātāju atbalsts izstrādātajām standarta specifikācijām. Šeit ir redzami divi risinājumi – testējam scenāriju kodu visos specifikācijā prasītajos vai populārākajos interneta pārlūkos vai automātiski ģenerējam scenāriju valodas kodu ar translatora palīdzību, biznesa loģikas korektumu testējot ar šīs vides testēšanas rīkiem. Tas, protams, ir lietderīgi, ja šī vide ir industrijas standarts ar nopietnām iestrādēm testēšanā.

Ir pieejami dažādi rīku komplekti, kas paredzēti lietotāja saskarnes ģenerēšanai, izmantojot HTML, CSS un *JavaScript*, kā arī XUL un XAML tehnoloģijas. Ir gan komerciālā Microsoft Atlas tehnoloģija, gan atvērtā koda projekts ZK, gan Google piedāvātais bezmaksas risinājums *Google Web Toolkit*. Katrai no šīm sistēmām ir atšķirīga pieeja sistēmas uzbūvei, bet tās vieno ideja, ka lietotāja saskarnes būvēšanai izstrādātājam nav jāpārzina liela apjoma tehnoloģijas un tehniskās nianses dažādos interneta pārlūkos. Visa izstrāde notiek tajā pašā izstrādes vidē (Microsoft gadījumā .NET, citos gadījumos Java), kur top sistēmas funkcionalitāte, izmantojot gatavu bibliotēku lietotāja saskarnes veidošanai.

Šeit tad arī parādās iespējamie ieguvumi un problēmas. Sāksim ar ieguvumiem:

- Izstrādātājam vairs nav jāpārzina konkrētā pārlūka DOM un DHTML atbalsts. Par to ir parūpējušies bibliotēkas autori.
- *JavaScript* nepiedāvā izstrādātājam tik plašas iespējas kā pilnvērtīgas objektorientētas programmēšanas valodas
- JavaScript un klienta puses izstrādei ir vājš iebūvētais atbalsts izstrādes vidēs. Tā kā izstrādi varēs veikt ierastajā pilnvērtīgajā izstrādes vidē, varēs tikt izmantoti visi testēšanas un tās automatizācijas rīki.
- Kļūdas *JavaScript* sintaksē, tipu nesaderība tiks noteikta kompilācijas laikā, nevis darbības laikā.

Negatīvie aspekti savukārt ir šādi:

- Jāpaļaujas uz saskarnes bibliotēkas autoru izstrādāto produktu, kurš var saturēt kļūdas
- Problemātiska integrācijas testēšana – ģenerētais kods visdrīzāk būs nelasāms un problēmas atrašanā tiks zaudēts laiks, kas tiek iegūts daudz efektīvāk veicot vienību testēšanu.
- Tiek pazaudēta procesa caurspīdība. Vienkāršākajos gadījumos tas var nebūt pārāk būtiski, bet izmantojot AJAX tehnoloģijas, ir svarīgi izmantot iespēju apskatīt datu stāvokli konkrētās situācijās.
- Lietotāja saskarne tiek ģenerēta ar to pašu kodu, ko sistēmas funkcionalitāte. Visdrīzāk to raksta viena un tā pati persona. Tīmekļa izstrādes procesā jau sen ir pieņemts nodalīt sistēmas funkcionalitāti no

izskata ar sagatavju palīdzību, kuras var labot cita persona bez programmēšanas zināšanām.

2.3.7 Drošības testēšana

Aplūkosim svarīgākos tīmekļa aplikāciju drošības aspektus un novērtēsim iespējas to testēšanu automatizēt. Izmantojot iegūtās zināšanas par automatizēt tīmekļa programmu iespējām un vājajām vietām, kā arī konkrētu rīku izpētē iegūtās praktiskās nianses, apskatīsim, kā mēs varam pilnībā vai daļēji veikt 10 lielāko drošības problēmu [17] automatizētu testēšanu:

1. Nepārbaudīti lietotāja ievaddati. Tā kā lietotāja ievaddati ir HTTP pieprasījumi (retāk tie ir faili), tad ļaundabīgam lietotājam ir visas iespējas izmainīt šos datus, lai apietu drošības mehānismus. Mūsu uzdevumus ir pārbaudīt, vai izstrādātājs visus datus pirms izmantošanas pārbauda un izfiltrē. 2.3.5 nodaļā apskatīti tīmekļa aplikācijām specifiskie ievaddati ļauj secināt, ka šīs problēmas automatizēta testēšana ir iespējama. Tomēr visdrošākā metode šeit būtu pilnīga koda apskate
2. Nepilnības pieejas kontrolē. Tīmekļa aplikācijām bieži ir vairāklīmeņu pieejas kontrole, kura sniedz pieeju noteiktai funkciju kopai. Bieži vien šī kopa ir diezgan plaša un tāpēc var rasties problēmas pilnībā kontrolēt, vai visās vajadzīgajās vietās tiek pieprasīta autentifikācija. Arī šajā gadījumā visdrošākā pārbaudes metode ir koda apskate un atbilstības salīdzināšana ar pieejas līmeņu dokumentāciju. Automatizācijai šeit paveras plaša iespējas, jo ir iespējam izmantot HTTP pieprasījumu atbildes (statusa kods, teksts HTML dokumentā), lai konstatētu pieejas līmeni.
3. Nepilnīga autentifikācija un sesiju pārvaldība ir vēl viens veids, kā ļaunprātīgais uzbrucējs var izmantot nepilnība un iegūt pieeju viņam neparedzētām funkcijām. Autentifikācija, ja netiek izmantots drošais SSL savienojums, tiek veidota atsevišķi, izmantojot dažādus sesiju veidošanas paņēmienus. Bieži vien tie mēs būt nepilnīgi un lietotājs tiek identificēts tikai pēc sesijas identifikatora vai pieļautas citas drošības mehānisma uzbūves nepilnības.
   Šeit var līdzēt koda apskates un plaša profila uzbrukumu testēšana. Prakse rāda, ka automatizētie rīki uzbrukuma testēšanā ir nepilnīgi

[18] to izmantotā melnās kastes principa dēļ un ir nepieciešama manuālā testēšana, kuras daļu var automatizēt ar speciāliem rīkiem. Standarta funkcionālās testēšanas rīki nepiedāvā šādas iespējas.

4. Starplapu skriptu izpilde (XSS – Cross Site Scripting) ir iespējams, ja kādam lietotājam ir iespējams nosūtīt citam lietotājam ziņu, kurā iekļauts kods skripta veidā, ja netiek veikta pilnīga ievaddatu pārbaude. Šim skriptam ir pieeja visai informācijai, kas glabājas lietotāja sīkdatnēs (cookies*, eng*), kur parasti tiek glabāta arī sesijas informācija un identifikators.
   Automatizētā testēšana šeit var līdzēt kā jau minēts 1. punktā. Kārtējo reizi jāmin, ka iedarbīgākā metode ir koda pārskats, īpaši pievēršot uzmanību scenārijiem, kur lietotāja ievads tiek izvadīts publiskai apskatei.
5. Bufera pārpildīšanās ir patiesi bīstama problēma, kuru gan spēj izmantot tikai augsti kvalificēts uzbrucējs. Izmantojot precīzi izplānotu datu ievadi, uzbrucējs var pārpildīt tīmekļa aplikācijas izpildes steku un likt izpildīt savus koda fragmentus. Visbiežāk tas iespējams dažādās grafiskajās bibliotēkās, kuras izmanto attēlu apstrādei.
   Šajā gadījumā automatizētā testēšana palīdzēt nevar. Palīdzēt var regulāra servera programmatūras atjaunošana un sava koda apskates, kuras var atklāt kādu iespējamu problēmu tīmekļa aplikācijā. Intel jaunākajos procesoros ir iebūvēta tehnoloģija Execute Disable Bit [19], kas aizsargā pret konkrētām bufera pārpildīšanās problēmu klasēm.
6. Injekciju problēmas ļauj uzbrucējam caur tīmekļa aplikāciju nodot komandas operētājsistēmai caur sistēmas izsaukumiem, citām programmām caur čaulas komandām kā arī datubāzu sistēmām caur SQL pieprasījumiem. Vāji aizsargātās sistēmās var ievadīt pat veselus Perl vai Python scenārijus. Apdraudētas ir visas sitēmas, kuras izmanto kāda veida interpretējamu valodu. Tomēr biežāk sastopamā problēma ir SQL injekcijas, kuras ir ļoti vienkārši veidojamas, taču tikpat viegli ir pret tām aizsargāties. Tā ir drošas programmēšanas pamatprincipu skaitā.

Automatizētā testēšana gluži kā minēts 1. un 4. punktā, šeit var līdzēt ar ievaddatu pārbaudes pilnīguma testēšanu.

7. Nepilnīga kļūdu pārvaldība ir veids kā sniegt iebrucējam papildus informāciju par sistēmu. Programmas steka izpildes izsekošana, kļūdu kodi, detalizētas SQL sintakses skaidrojumi ir noderīgi izstrādes laikā, bet ir nopietns drošības drauds produkcijas vidē. Arī nesniedzot pilnas problēmas detaļas, tiek sniegta daļēja informācija par sistēmas uzbūvi, izmantoto programmatūru, kurai var būt konkrētas drošības problēmas ar gataviem uzlaušanas risinājumiem. Pareizais piegājiens ir šādos gadījumos lietotājam parādīt tikai vienkāršu paziņojumu par sistēmas kļūdu un detalizēto informāciju izstrādātājiem sniegt žurnāla failos un standarta izvadē.

   Automatizētā testēšana var palīdzēt arī šajā gadījumā. Nedaudz pārveidojot funkcionālās testēšanas testpiemēra pamatklasi mēs varam tajā iekļaut teksta konstantes, kuras noteikti apzīmēs kļūdas situāciju. Šeit gluži nav universālu risinājumu, bet ir jāizmanto informācija par sistēmā izmantotajām tehnoloģijām un to atgrieztajiem kļūdu paziņojumiem.
8. Nedroša datu glabāšana visvairāk attiecas uz sensitīvās informācijas glabāšanu, taču labi būvētai sistēmai būtu jāglabā visi dati tā, lai pieejami tie būtu tikai attiecīgajām personām. Biezākās problēmas ir kritisko datu glabāšana nešifrētā veidā, nedroša atslēgu, sertifikātu un paroļu glabāšana, vāju algoritmu izvēle un mēģinājumi radīt inovācijas drošības tehnoloģijās.

   Diemžēl šeit automatizācija nelīdzēs, jo pieļautās kļūdas struktūras uzbūvē varēs atklāt tikai rūpīgas koda apskates.
9. Pieejas liegšana (*Denial of Service, eng*) ir smaga problēma tīmekļa aplikāciju drošībā, jo tās ir vāji aizsargātas pret šādām problēmā. Turklāt ir ļoti grūti atšķirt uzbrukumu no vienkārša lapas apmeklējuma, jo IP adresi nevar uzskatīt par pietiekamu identifikācijas līdzekli. Risinājumi šeit ir tikai cīņa ar sekām – serveru pudura izmantošana slodzes sadalīšanai, koda optimizācija, resursu prasīgāko funkciju pārveidošana, kešinga ieviešana, viena lietotāja

izmantoto resursu ierobežošana, bet tas ir tikai īslaicīgs risinājums, jo uzbrucējs var palielināt pieprasījumu skaitu un patērēt visus resursus. Automatizētā testēšana šeit var līdzēt, izmērot slodzi, kādu var izturēt serveri, simulējot lielu pieprasījumu un lietotājus skaitu.

10. Nedroša konfigurāciju pārvaldība ir visai svarīga lieta visas sistēmas drošībā. Tajā ietilpst sistēmas programmatūras atjaunināšana, biežākās problēmas ir failu pieejas tiesības konfigurācija, sistēmas informācijas pieejas liegumi, aizmirsti faili, noklusētie lietotāju konti ar noklusētām parolēm, liekas informācijas izpaušana.

Daļu no šīm problēma var atklāt automatizētie sistēmas drošības skanēšanas rīki. Tiesa, tos var izmantot arī uzbrucējs, tāpēc tas ir jāpaspēj pirms viņa un jālabo atklātās problēmas. Taču šeit galvenā vadlīnija būtu stingras drošības shēmas izveide un tās ievērošana veicot servera konfigurācijas darbus.

3 Testēšanas rīki

Ir pieejami daudzi gatavi rīki, kas paredzēti testēšanas automatizācijai. Ir gan komerciāli risinājumi, gan bezmaksas, gan atvērtā koda produkti. Konkrēta rīka izvēli bieži vien nosaka esošā izstrādes vides konfigurācija, tāpēc katram ir iespējas izvēlēties piemērotāko. Tā kā vēl neviens nav atradis un pierādījis labāko testēšanas metodoloģiju, piedāvāto rīku klāstā ir rīki ar dažādu pielietošanas praksi.

Apskatīsim kādas alternatīvas mums piedāvā dažādi izstrādājumi. Šie rīki ir sakārtoti alfabētiskā secībā.

- **Anteater** ir testēšanas vide, kas balstīta uz Apache Jakarta izstrādāto Ant vidi, kas ir automatizācijas rīks, kas ar XML aprakstītām komandām veic dažādus kompilācijas un izpildāmo moduļu veidošanas uzdevumus. Anteater ir iekļauts šajā procesā. Acīmredzami saskatāmie plusi šeit ir iekļaušanās programmizstrādes procesā, ļaujot testus izpildīt pie katras jaunas programmas versijas veidošanas un veicot regresijas testēšanu. Unikāla ir iespēja gaidīt uz ienākošajiem HTTP paziņojumiem. Īpaši noderīgi tas ir augsta līmeņa risinājumos, kur sistēmas saturiska atbilde notiek tikai pēc procesa izpildes, kas var aizņemt laiku. Saziņai ar serveri tiek izmantoti HTTP ziņojumi ar plašām iespējām atgriezto rezultātu apstrādei ar izplatītākajiem testa apstrādes algoritmiem. [21]
- **Apache HTTP test** ir rīks, kas izmantojams tikai uz Apache HTTP servera un piedāvā noslodzes testēšanas rīku ar svarīgāko ātrdarbības rādītāju mērījumu datiem, un citiem moduļiem, kas izmantojami servera veiktspējas rādītāju novērtēšanai. [22]
- **AppPerfect Unit Tester** ir komerciāls rīks (bezmaksas versija nekomerciāliem mērķiem), kas tiek izmantots Java vienību testēšanas testu ģenerēšanā un izpildīšanā. Tas balstās uz jUnit un HttpUnit testēšanas vidēm. Izveidotie testa piemēri lielā mērā pārklāj koda izpildi un dod iespēju apskatīt izpildes ceļa sazarojumus un vajadzības gadījumā izmainīt tos. Tiek piedāvāta integrācija ar populārākajām izstrādes vidēm. **AppPerfect Functional Tester** ir funkcionālās

testēšanas rīks ar tādiem pašiem licences nosacījumiem un var veikt funkcionālo un regresijas testēšanu. Tas piedāvā ierakstīt darbības scenārija veidā un tās atskaņot, pieļaujot šo ierakstu labošanu. Atbalstīts tiek tikai Internet Explorer 5/6 pārlūks, jo ieraksts tiek veikts, balstoties uz pārlūka notikumu izsaukumiem. [23]

- **Cactus** ir vienkāršs servera puses testēšanas rīks, kas veic servera puses Java koda testēšanu. Tā mērķis ir samazināt ieguldīto darbu, rakstot servera puses testa piemērus. Tas izmanto un paplašina jUnit iespējas. Tas izmanto konteinera stratēģiju, kas nozīmē, ka testi tiek izpildīti savā konteinerā un neietekmē citu testu izpildi. Cactus varētu saukt par vienību integrācijas testēšanas rīku. [24]
- **HtmlUnit** ir veidot kā atvasinājums no HttpUnit, bet izmanto atšķirīgu piegājienu. Tas izmanto nevis HTTP atbildes, bet pašu HTML dokumentu. Abi šie risinājumi ir diezgan līdzīgi un konkrētu izvēli nosaka individuāli kritēriji. [25]
- **HttpUnit** ir funkcionālās testēšanas rīks, kurš neizmanto interneta pārlūku savā darbībā. Tas pats emulē tā darbību, ieskaitot formu iesniegšanu, JavaScript atbalstu, pamata HTTP autentifikāciju, sīkdatnes un lapu pārsūtīšanu. Saņemtās atbildes tas var apstrādāt kā tekstu, XML DOM vai formu, tabulu un saišu kolekcijas. Ir izmantojams kopā ar jUnit, lai veidotu vienotus automatizācijas uzdevumus. [26]
- **JsUnit** ir klienta puses JavaScript vienību testēšanas rīks. Tā ir jUnit realizācija JavaScript valodā ar iebūvētu atbalstu automatizētai izpildei dažādās lietotāja vidēs. [27]
- **JMeter** ir Apache Jakarta izstrādāts grafisks Java rīks ar kuru iespējams veikt funkcionālus slodzes testus un iegūt mērījumus par lapas ātrdarbību. Tas var izpildīt gan lapu, gan failu pieprasījumus, gan datubāzes ierakstu nolasīšanu un piedāvā grafiskus ziņojumus. [28]
- **JSpider** ir rīks, kas var efektīvi veikt tīmekļa aplikācijas lappušu apstaigāšanu. To var izmantot, lai atklātu kļūdainas lapas, izejošās un iekšējās saites, analizētu lapas struktūru, izveidojot lapas karti, lejupielādētu lapu pilnībā un ar pievienojamu spraudņu palīdzību veiktu arī citus noderīgu procesus. Testēšanai noderīgās funkcijas ir

kļūdu paziņojumu meklēšana lapās saišu apstaigāšana, meklējot kļūdas. Tas ņem vērā arī standartus HTTP 1.1 un robots.txt, kas dod instrukcijas tīmekli apstaigājošiem rīkiem [29]

- **JTidy** ir HTML Tidy Java versija, kas paredzēta HTML sintakses pārbaudei un dokumenta struktūras lasāmības uzlabošanai. To var izmantot nekorekta HTML koda labošanai. Tas ir atvērtā koda projekts, kurš piedāvā integrāciju arī ar citiem uz Java balstītiem rīkiem. [30]
- **jUnit** ir xUnit arhitektūras realizācija Java vidē. Tā piedāvā vienību testēšanas atbalstu un gatavie scenāriji ir viegli automatizējami, taču pašu testu veidošanai nav nekāda iebūvēta atbalsta. Taču ļoti dauzi rīki izmanto šo vidi par pamatu, pievienojot testu ieraksta vai ģenerēšanas funkcionalitāti. [31]
- **MaxQ** ir funkcionālās testēšanas rīks, kurš balstās uz HTTP pieprasījumu ierakstu un atskaņošanu. Tiek ģenerēts Python kods, kas ir labojams un papildināms. Ir pieeja Python un Java bibliotēkām. [32]
- **NUnitAsp** ir automatizācijas rīks Microsoft tehnoloģijā ASP.NET veidoto tīmekļa aplikāciju testēšanai tas ir paplašinājums no NUnit, kas ir rīks testēšanai .NET vidē. Šis rīks ir paredzēts izstrādātājiem, lai veiktu sava koda vienību testēšanu un var veikt servera puses loģikas testēšanu. [33]
- **Sahi** ir automatizācijas rīks ar scenāriju ieraksta un atskaņošanas iespēju. Tas ir izstrādāts, izmantojot JavaScript iespējas pārlūkā un tā veidotie testa scenāriji ir rakstīti JavaScript valodā. Ātrdarbības uzlabošanai tas izmanto vairākus pavedienus paralēlai izpildei. Ir iespēja iekļaut to Ant komandu failos, lai to izpildītu vienā kopējā uzdevumā. Tā darbībai nepieciešams atvērts interneta pārlūks. [34]
- **Selenium** ir automatizācijas rīku kopums, kas ir pieejams dažādās konfigurācijās. Tas pilnībā balstās uz JavaScript izmantošanu un dažādie risinājumi tikai piedāvā ērtāku darba vidi. Ir pieejams gan risinājums specifiskiem pārlūkiem, gan servera pusē uzstādāms modulis, kas pieļauj gandrīz jebkuras valodas izmantošanu testu rakstīšanai. Līdzīgi kā Sahi gadījumā testa izpildes laikā ir jābūt atvērtam interneta pārlūkam. [35]

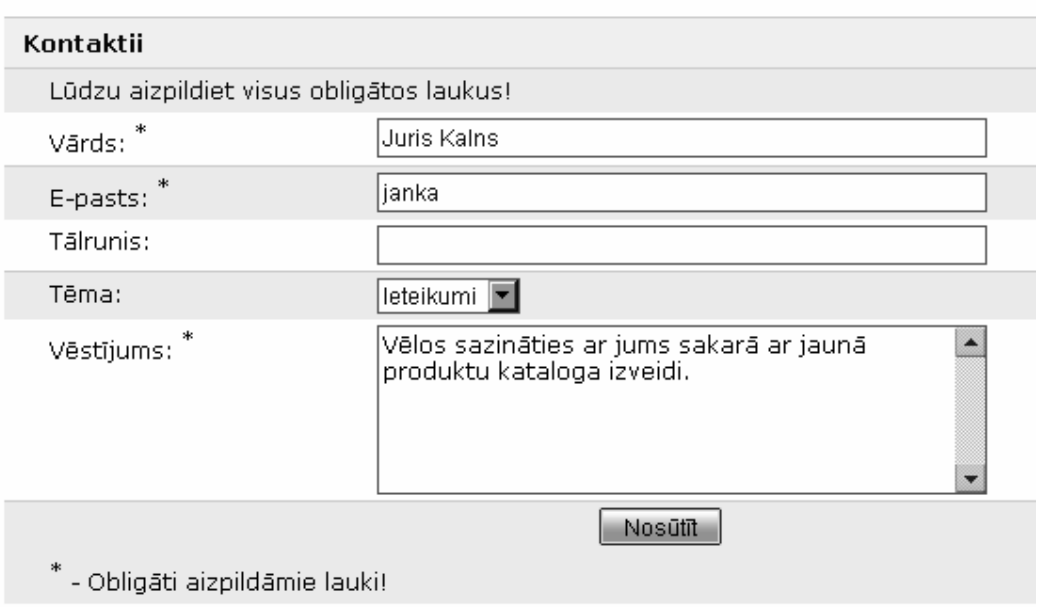
- **Pear PHPUnit2** ir vienību testēšanas ietvars PHP5 valodai. Tas ir objektorientēts un paredzēts izstrādātājam, kurš ievieš vienību testēšanu PHP vidē. [36]
- **TestMaker** ir testu automatizācijas rīks, kas paredzēts funkcionālajai un noslodzes testēšanai. Darbība notiek ar grafisku lietotāja saskarni un testēt var, izmantojot HTTP, HTTPS, SOAP, XML-RPC, SMTP, POP3, IMAP protokolus. Testa skripti tiek veidoti Jython valodā, kas sniedz lielas iesējas to papildināšanā. Testu izpildei tiek izmantots sadalīts klientu tīkls, kas darbojas, netraucējot lietotāju darbību. Tiek piedāvāts arī komerciāls rīka versija ar profesionālu atbalstu, taču arī atvērtā koda risinājums izceļas ar savu augsto kvalitāti [37]
- **utPLSQL** ir PL/SQL videi paredzēts vienību testēšanas rīks, kas piedāvā automatizēt pakotņu, funkciju un procedūru testēšanu. [38]
- **Wattij** ir rīks tīmekļa aplikāciju testēšanai Java vidē, kas radīts uz cita rīka Wattir (paredzēts Ruby videi) bāzes. [39]
- **WebScarab** ir testēšanas ietvars , kurš paredzēts tīmekļa aplikāciju drošības testēšanai. Tas ieraksts pa HTTP un HTTPS protokoliem pārraidītos datus un ļauj tās lietotājam ar tiem pēc tam manipulēt un analizēt. Tas ļauj izstrādātājam atkļūdot citādi grūti pieejams problēmas un drošības speciālistam atklāt iespējamās ievainojamības aplikācijas izstrādes plānojumā vai ieviešanā. Ir pieejami arī dažādi spraudņi, kas paplašina šī rīka funkcionalitāti. [40]

Tagad tuvāk aplūkosim dažus konkrētus testēšanas rīkus. Šo rīku izvēli noteica šādas prasības, kuras tika izveidotas, pamatojoties uz uzņēmumā topošā produkta vajadzībām:

- Rīkam jābūt bezmaksas / atvērtā koda, jo ir mērķis minimizēt testēšanā ieguldītos resursus un nav tāds testētāju kvalifikācijas līmenis, kas prastu izmantot komerciālo rīku papildus iespējas
- Rīkam jābūt paredzētam Java videi vai jābūt universālam
- Ģenerētajiem skriptiem jābūt viegli labojamiem, jaunu skriptu izveidei jābūt vienkāršai. Skriptu valoda ir Java vai arī kāda cita vienkārša valoda.
- Apgūšanas līknei jābūt pēc iespējas lēzenai

Tika apskatīts arī viens komerciāls rīks, lai saprastu, kādi ir ieguvumi, iegādājoties maksas rīku.

Testēšanas rīku praktisko pielietojumu pārbaudīsim uz vienkāršas kontaktu formas [Attēls 3-1], kurai vajadzētu sniegt ieskatu automatizācijas programmas pamata darbībā un automatizācijas scenāriju veidošanā. Formai ir vairāki lauki, kuri ir jāaizpilda, turklāt e-pasta laukam tiek pārbaudīts, vai ievadītais atbilst e-pasta adreses formātam.



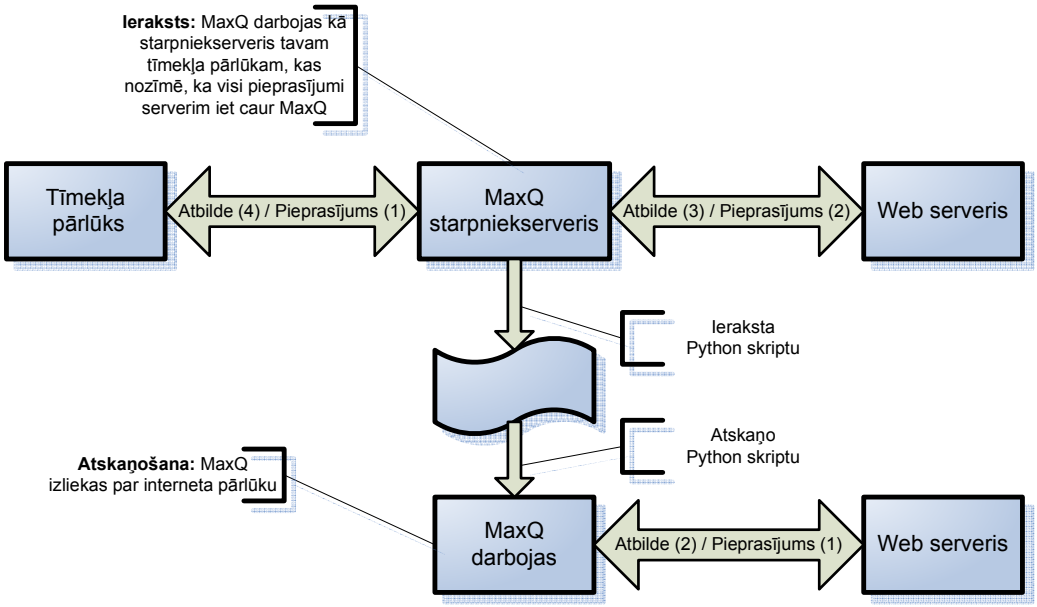
Attēls 3-1 – Kontakforma, kura tika izmantota rīku praktiskai pārbaudei

Izmēģinot šos rīkus, pievērsīsim uzmanību testēšanas piemēru veidošanai, iespējām viegli to labot un modificēt. Svarīga ir iespēja viegli veidot sagataves un netērēt laiku tehniskām detaļām, jo automatizācijas galvenais mērķis ir atvieglot manuālo testēšanu.

2.3.5 nodaļā apskatījām specifiskos ievaddatus pa problēmu klasēm, kuriem vajadzētu pievērst liekāko uzmanību. Būtu labi, ja mēs varētu izpildīt testus uz ievaddatu masīviem, kuri atbilstu šīm ievaddatu klasēm. Ir viegli iedomāties, ka pēc problēmu klasēm un viena derīga ievaddatu piemēra mēs varam izveidot dažādus datus, kuri pārstāvēs katru no iespējamām problēmām. Tāpēc būtu svarīgi izmantot iespēju rakstīt šos skriptus savā ierastajā programmēšanas valodā vai pat izstrādes vidē.

3.1 Bitmechanic MaxQ

Bitmechanic MaxQ ir atvērtā koda funkcionālās testēšanas rīks, kurš rakstīts Java valodā, tāpēc izmantojams uz dažām platformām. Tā darbības princips ir starpniekservera uzstādīšana uz testētāja datora un nosūtīti datu pierakstīšana Jython (valodas Python implementācija, rakstīta Java, iespējams izmantot Java bibliotēkas) scenāriju valodā. Rīks veidots uz jUnit bāzes, tāpēc veidotos testa piemērus varēs darbināt vienā testēšanas uzdevumā ar vienību testēšanai veidotajiem servera puses testa piemēriem. Sistēmas darbības shēmā [Attēls 3-2] redzam, ka skriptus iespējams veidot, izmantojot mūsu veikto darbību pārlūkā ierakstīšanu. Skriptu izpilde iespējama grafiskā lietotāja vidē vai no komandrindas, kā arī, iekļaujot to jUnit TestSuite.



Attēls 3-2 - MaxQ rīka darbības shēma

MaxQ iezīmes:

- Izmanto Python skriptus, kas ir vienkārši un viegli labojami
- Darbojas no komandrindas bez lietotāja palīdzības
- Atbalsta sīkdatnes (*cookies, eng*)
- Strādā aiz starpniekserveriem (*proxy*)

Testu veidošana notiek, uzstādot pārlūkam kā starpniekserveri testētāja datoru un palaižot MaxQ grafisko vidi, izveidojot jaunu testa piemēru un sākot ierakstu.

Aizpildot mūsu kontaktu formu un iesniedzot to, tiek uzģenerēts šāds Python kods, kurš, veidot tā atskaņošanu, veiktu pieprasījumu uz lapu un sagaidītu HTTP statusa kodu 200.

```
          self.msg('Test started')
           params = [
               ('''feedback_name''', '''Juris Kalns'''),
               ('''feedback_email''', '''jurka'''),
               ('''feedback_phone''', '''3245369'''),
               ('''feedback_theme''', '''11'''),
               ('''feedback_text''', '''Vēstījuma teksts'''),
               ('''feedback_submit''', ''''''),]
           self.msg("Testing URL: %s" %
self.replaceURL('''http://test.lv/lv/contacts/index.html?feedback_name=Juris
Kalns&feedback_email=jurka&feedback_phone=3245369&feedback_theme=11&feedback_text=VÄ“stÄ«j
uma teksts&feedback_submit='''))
           url = "http://test.lv/lv/contacts/index.html "
           Validator.validateRequest(self, self.getMethod(), "post", url, params)
           self.post(url, params)
           self.msg("Response code: %s" % self.getResponseCode())
           self.assertEquals("Assert number 1 failed", 200, self.getResponseCode())
           Validator.validateResponse(self, self.getMethod(), url, params)
```

Tas nav pietiekami, lai veiktu pilnīgu funkcionālo testēšanu, jo šāds kods tiks atgriezts pie jebkuras lapas korektas ielādes, tāpēc pievienojam pārbaudei vēl vienu rindiņu:

```
self.assertFalse(self.responseContains('''Lūdzu, ievadiet korektu e-pasta adresi!'''))
```

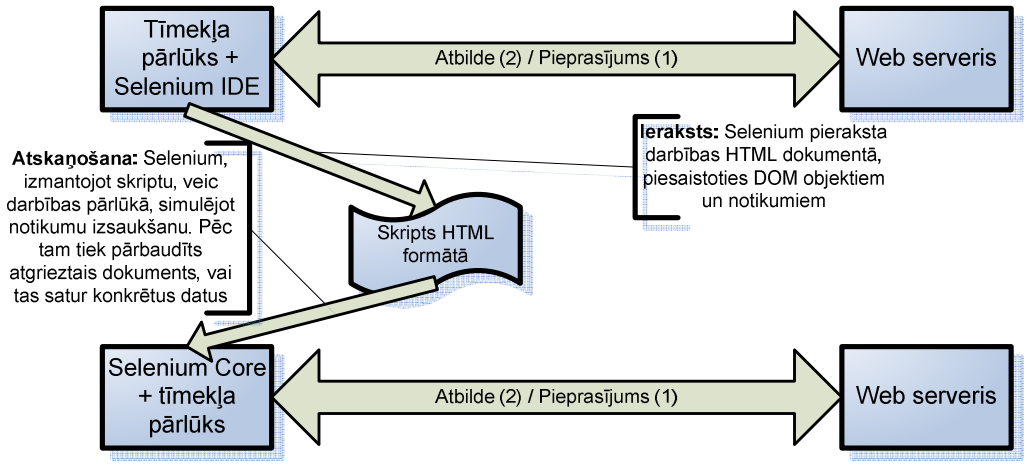
Šī komanda pārbauda, vai rezultātā netika atgriezta lapa ar paziņojumu, ka ievadīta nekorekta e-pasta adrese, kā tas bija mūsu gadījumā.

Redzam, ka komanda `self.post` tiek izpildīta ar parametriem `url` un `params`, kas, ir attiecīgi lapas adrese un datu struktūra ar parametru nosaukumiem un to vērtībām. Viegli iedomāties, ka varam izveidot ievaddatu masīvu un ciklā to izpildīt. Pielikumā 2.1 varam apskatīt vienkāršu papildinājumu ar masīvu un vairākiem ievaddatiem.

Ja aplūkojam šī risinājuma plusus, noteikti jāpiemin tā universālums. HTTP protokola ieraksts ļauj tam darboties dažādās vidēs, ierakstu veikt Windows vidē, bet testu izpildi no Unix servera. Jython valoda dod pieeju visiem Pyhon objektiem, kā arī jebkurai Java bibliotēkai, tāpēc ir ļoti noderīga testētājam / izstrādātajam. Tiesa, Python principu nepārzināšana var būt neliels šķērslis, kas iesākumā jāpārvar, sākot praktisku testēšanas darbu ar šo sistēmu.

3.2 Selenium IDE

Selenium IDE ir funkcionālās testēšanas rīks, kas darbojas tieši pārlūkā, gluži kā lietotājs. Tiek piedāvāts populārāko pārlūku atbalsts uz visām operētājsistēmu platformām. Skripti rakstīti vienai videi darbosies arī citās vidēs. Šis konkrētais risinājums kā Firefox pārlūka paplašinājums. Selenium piedāvā arī moduli, kurš ir uzstādāms uz testējamā servera (Selenium RC), kā arī skriptu kopumu, kurš darbojas uz visiem populārākajiem pārlūkiem un operētājsistēmām ar vienotu testu formātu (Selenium Core). Tas darbojas atsevišķā tīmekļa lapas rāmī ar *JavaScript* palīdzību. Šis skriptu kopums ietilpst arī Selenium IDE un Selenium RC komplektā.



Attēls 3-3 - Selenium darbības shēma

Attēls 3-3 paskaidro Selenium IDE darbību. Jāpiemin, ka Firefox paplašinājums tikai atvieglo automatizācijas skriptu veidošanu, ir iespējams tos veidot arī pašam. Aplūkosim mūsu kontaktu formas iesniegšanas testa ierakstu. Redzam, ka tas ir vienkāršs HTML dokuments. Pilns izejas kods atrodams pielikumā 2.2.

Kontaktformas tests 2		
open	/lv/contacts/	
assertTitle	Kontaktforma	
type	feedback_email	jurka@burka.lv
type	feedback_phone	2348235
select	feedback_theme	label=Ieteikumi
type	feedback_text	Teksts, vēl teksts

Kontaktformas tests 2		
clickAndWait	//input[@value='Nosūtīt']	
assertTitle	Nams24	
type	feedback_name	Juris Krūms
assertTextPresent	Paldies, ka sazinājāties ar mums!	

Arī šoreiz tika izveidots tikai šablons testēšanai, kuru mēs papildinājām ar pēdējo komandu assertTextPresent, kura atgrieztajā dokumentā meklēja šādu tekstu.

Šajā dokumentā, kurš patiesībā ir komandu virkne un tikai lasāmības dēļ tik glabāts HTML, nevis XML formātā, redzam, ka varam izmantot iepriekš redzēto ievaddatu masīvu un ar kādu teksta apstrādes rīku ģenerēt komandas un ievaddatus mūsu problēmu klasēm. Tiesa, šoreiz nevarēsim izmantot ciklu pašā automatizācijas skriptā.

Šīs sistēmas pluss ir vienkāršā testu ieraksta veikšana un semantiski saprotamā skriptu ieraksta valoda, kas gan tomēr nav kāda no programmēšanas valodām, tāpēc ir neizbēgama stāvāka mācīšanās līkne. Tomēr lielāks mīnuss šai sistēmai ir samērā lēnā testu izpilde, jo tiek simulēta reāla pārlūka darbība. Tas varētu būt pieņemami, automatizējot kāda vienkāršāka aplikācijas moduļa darbību, bet tomēr pamatīgi ierobežotu darbu ar pilnu sistēmas moduļu kopuma notestēšanu, tāpēc perspektīvas izmantot šo rīku pie katrām koda izmaiņām nav pārāk spīdošas. Daļēji to varam skaidrot ar to, ka šis rīks ir veidots uz pieņemšanas testēšanas rīka bāzes, kur nav nepieciešama vairākkārtēja regulāra testu kopas izpilde.

3.3 HttpUnit

HttpUnit ir Java vidē veidots funkcionālās testēšanas rīks, kurš var testēt dažādās vidēs veidotas tīmekļa aplikācijas. Tas darbojas līdzīgi kā MaxQ, apstrādājot HTTP atbildes kodus un rezultātu. Taču tam ir arī citas iespējas. Tā automatizācijas skripti ir rakstāmi Java valodā un iekļaujami jUnit testu kopā. Tas spēj daļēji emulēt pārlūka darbību, ieskaitot formu iesniegšanu, JavaScript darbību, vienkāršu HTTP autentifikāciju, sīkdatnes un lapu pārsūtīšanu uz citu adresi. Tas var apstrādāt atgriezto dokumentu kā tekstu, XML DOM vai formu, tabulu un saišu konteinerus.

Šajā gadījumā mums ir jāraksta Java kods un nav testa piemēru ģeneratora. Vienkāršs piemērs, kā izskatās testa izsaukums mūsu kontakstu formai.

```
public void testContactForm() throws Exception {
        WebConversation     conversation = new WebConversation();
        WebRequest  request = new PostMethodWebRequest(
            "http://test.lv/lv/kontakti" );

        WebResponse response = conversation.getResponse( request );
        WebForm contactForm = response.getForms()[0];
        request = contactForm.getRequest();
        request.setParameter( "feedback_name", "Juris Kalns" );
        request.setParameter( "feedback_email", "jurka@mail.lv" );
        request.setParameter( "feedback_phone", "3254325" );
        request.setParameter( "feedback_theme", "11" );
        request.setParameter( "feedback_text", "Ko nu?" );
        request.setParameter( "feedback_submit", "" );
        response = conversation.getResponse( request );
        assertTrue( "Form not accepted",
               response.getText().indexOf("Paldies, ka sazinājāties ar mums!") != -1);
        assertEquals( "Page title", "Kontakti", response.getTitle() );
}
```

Valodas Java izmantošana dod iespēju veidot pašam savus ievaddatu masīvus, viegli ielasīt tos no failiem, datubāzes un citādi brīvi manipulēt. Mēs varam izveidot klasi, kas ģenerē gadījuma testa ievaddatus, lai ar kādu varbūtību atklātu neapredzētu lietotājs uzvedību. Tiesa daudz noderīgāk būtu pārbaudīt lietotāja ievadu ar skaidri apzinātām problēmām. Šeit varam izmantot 2.3.5 nodaļā apskatītos specifiskos ievaddatus tīmekļa programmām. Pielikumā 2.3 var apskatīt klases piemēru, kas veido lietotāja ievadu no teksta vai cipariska ievada un vektora (Java dinamiskā masīva) veidā to atgriezt. Tālāk jau varam cikliski mēģināt izpildīt testa piemērus ar šiem ievaddatiem. Jāņem vērā, ka kļūdas situācijas var izsaukt arī dažādas ievaddatu kombinācijas, kas citreiz strādā pareizi – piemēram, kāds lauks ir atkarīgs no otra – lietotājs izvēlas no Latvijas rajoniem Liepājas rajonu, bet kļūdas pēc viņš pagasta sarakstā var izvēlēties Līvānus. Tāpēc

būtu lietderīgi iekļaut ievaddatu izpildes cilus vienu iekš otra, kas gan dramatiski palielina izpildāmo operāciju skaitu (iekļauto masīvu pārstaigājamo dimensiju reizinājums dos izpildāmo testa gadījumu skaitu).

Šī testa priekšrocības ir salīdzinoši ātrā darbība, kas ļauj to plānot kā regulāri izpildāmu testu kopuma sastāvdaļu. Praksē to varētu veikt kopā ar jUnit tipa vienību testiem pie katrām koda izmaiņām regresiju testēšanai.

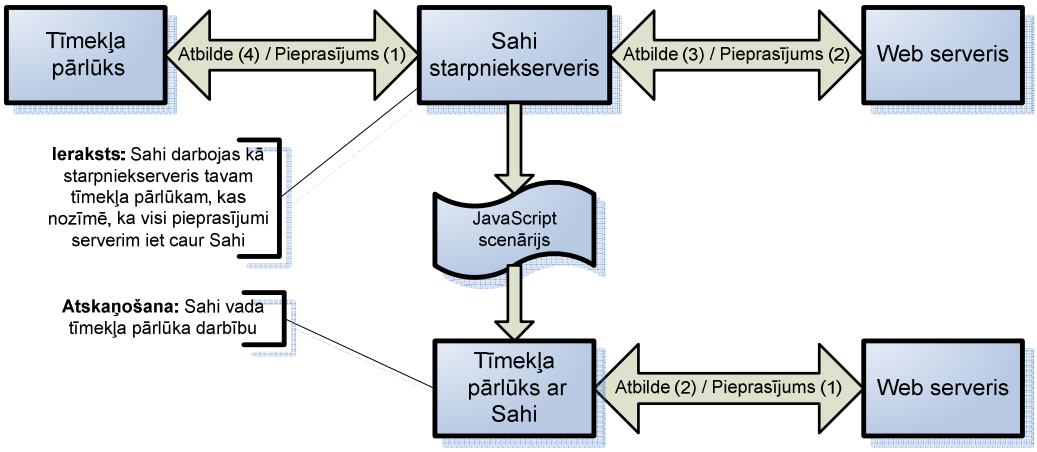
Lielākā šīs testu izstrādes vides problēma ir testu piemēru veidošana. Šeit nav pieejami rīki, kas veiktu kāda lietotājs scenārija darbības ierakstu, tāpēc pašam izstrādātājam ir jāorganizē sava testu koda veidošana, ko varētu loģiski organizēt funkcijās un palīgklasēs, atdalot biežāk lietojamās darbības, piemēram pieslēgšanās sistēmai ar lietotājvārdu un paroli, sīkdatņu veidošanu un citus uzdevumus.

Vēl problēmas mēs varam saskatīt iebūvētā pārlūka simulācijas nepilnībā. Mums ir jāpaļaujas uz to, ka izstrādātāji ir izveidojuši pēc iespējas pilnīgu atbalstu populārākajām konfigurācijām. Par JavaScript atbalstu ir zināms, ka ir pieejams gandrīz pilnīgs JavaScript 1.1 atbalsts, taču ir zināms, ka šodienas pārlūki atbalsta JavaScript valodas 1.4 versiju un tuvāko gadu laikā pamazām notiek JS2 virzienā. Tāpēc ir jārēķinās, ka JavaScript testēšanai būs jāvelta atsevišķa uzmanība un automatizācijai būs jāizmanto kāds no rīkiem, kas vada pārlūka darbību.

Šis rīks vairāk paredzēts izstrādātajiem un visbiežāk to izmantos tās pašas personas, kuras rakstīs vienību testus un arī pašu kodu. Testētājam, kuram nav nepieciešamās programmēšanas iemaņas, ar šo rīku strādāt būs sarežģīti.

3.4 ThoughtWorks Sahi

Sahi ir automatizācijas un testēšanas rīks tīmekļa programmām ar iespēju ierakstīt un atskaņot scenārijus. Tas ir izstrādāts Java un JavaScript un izmanto vienkāršus JavaScript izsaukumus, lai izsauktu notikumus tīmekļa pārlūkā. Veidotie skripti ir vienkārši, iebūvēts atbalsts testa piemēru izpildei pakešu režīmā un paralēlos pavedienos. Atbalsta SSL savienojumu. Līdzīgi kā MaxQ, Sahi darbojas kā starpniekserveris un pārtver pārlūka sūtītās HTTP komandas. Lai piekļūtu lapas elementiem, Sahi ievieto lapā savu kodu, padarot rīku neatkarīgu no tīmekļa aplikācijas.



Attēls 3-4 - Sahi darbības shēma

Sahi atver papildus logu, ar ko tiek kontrolēta skriptu ierakstīšana. Skripti ierakstās failā kā komandu virkne JavaScript sintaksē. Mūs kontaktu formas iesniegšana izpildās ar šādu skriptu, kas tika pārveidots par funkciju, lai veidotu ērtākus izsaukumus. Funkcijas ar pasvītrojumu sākumā ir Sahi definētās un ar tām tiek veikti izsaukumi uz pārlūku.

```
      function submitForm($name, $email, $phone, $theme, $txt, $msg) {
        _setValue(_textbox("feedback_name"), $name);
        _setValue(_textbox("feedback_email"), $email);
        _setValue(_textbox("feedback_phone"), $phone);
        _setSelected(_select("feedback_theme"), $theme);
        _setValue(_textarea("feedback_text"), $txt);
        _click(_submit("Nos\u016bt\u012bt"));
        _assertTrue(_containsText(_byId("col-a"), $msg),"Fail");

      }

      _navigateTo("http://test.lv/lv/contacts/")
      submitForm("Juris Kalns","jurka@mail.lv","4343554","S\u016bdz\u012bbas","Neraadaas
latvieshu burti.","Paldies, ka sazi");
      _navigateTo("http://test.lv/lv/contacts/")
```

```
      submitForm("Juris Kalns","jurkamiallv","4343554","S\u016bdz\u012bbas","Neraadaas
latvieshu burti.","dzu aizpildiet visus oblig");
```

Redzam, ka varam organizēt masīvu pārstaigāšanu, jo šeit testpiemēru izpilde ir vienkārša funkcijas izsaukšana. Sahi ir iebūvēts atbalsts uzdevumu izpildei paralēlos pavedienos, tāpēc dažas konstrukcijas, veidojot ciklus ir jāpārveido.

Šī rīka praktiskais pielietojums ir līdzīgs kā Selenium rīkam – tas veic interneta pārlūka vadību, aizpildot dažādus laukus un spiežot uz saitēm. Praktisks pielietojums šeit saskatāms ne tikai testēšanas automatizācijā, bet arī veicot neikdienišķus (jo testa palaišana nenotiek ļoti ērti) darbus. Arī citi plusi un mīnusi ir līdzīgi kā nosaukti pie iepriekš apskatītā Selenium, taču Sahi izstrādes process vēl ir zemākā līmenī, kas ir manāms dažās nepilnībās. Ja tiek izmantots Firefox pārlūks, ērtāk ir strādāt ar Selenium, bet citādi automatizēto skriptu ierakstīšanu ērtāk ir veikt ar Sahi.

3.5 AppPerfect WebTest

AppPerfect WebTest ir komerciāls rīks, kas veic funkcionālo testēšanu un ar papildus moduļa palīdzību arī noslodzes testēšanu, turklāt var to veikt arī vienlaicīgi. Šī īpašība slodzes testēšanu veic pat labāk, ko reālāk attēlo lietotāja darbību, jo izsauc arī resursus prasīgākas funkcijas.

WebTest ir grafiska lietotāja saskarne un arī testu scenāriju ierakstīšana notiek grafiski. Diemžēl tiek atbalstīts tikai Internet Explorer pārlūks, taču šajā risinājumā ir sarežģīti ieviest citus pārlūkus, jo izsaukumi notiek ar IE specifiskām funkcijām. Paši testi tiek saglabāti kā XML faili, tātad mašīnlasāmi. Arī testu labošana notiek grafiski, aizpildot dažādus lauciņus lietotāja saskarnē. Viena notikuma izsaukums, šajā gadījumā Teksta „Nav labi!" ievadīšana *textarea* HTML elementā aprakstās šādi:

```
<Event ElementTitle="107 101 121 112 114 101 115 115 32 111 110 32 116 101 120 116 97 114
101 97 32 102 101 101 100 98 97 99 107 95 116 101 120 116 32 58 32 78 97 118 32 108 97 98
105 33 " EventTypeName="keypress" ElementName="textarea" EventType="2" Delay="2"
IsIgnore="false" IsExactMatch="false" IsWindowElement="false"
IsRecordedWhileShowingSSL="false" ElementOuterHtml="60 84 69 88 84 65 82 69 65 32 99 108
97 115 115 61 98 105 103 56 32 105 100 61 102 101 101 100 98 97 99 107 95 116 101 120 116
32 110 97 109 101 61 102 101 101 100 98 97 99 107 95 116 101 120 116 32 114 111 119 115 61
55 32 99 111 108 115 61 51 48 62 78 60 47 84 69 88 84 65 82 69 65 62 "
IsBreakPointSet="false" IsReplayIndexBased="false" SourceIndex="79" ImageFile="">
<Attributes>
<Attribute Name="cols" Value="30" Ignore="false" IsParameterize="false" />
<Attribute Name="wrap" Value="soft" Ignore="false" IsParameterize="false" />
<Attribute Name="rows" Value="7" Ignore="false" IsParameterize="false" />
<Attribute Name="name" Value="feedback_text" Ignore="false" IsParameterize="false" />
<Attribute Name="id" Value="feedback_text" Ignore="false" IsParameterize="false" />
<Attribute Name="TextValue" Value="Nav labi!" Ignore="false" IsParameterize="false" />
</Attributes>
</Event>
```

Tomēr testētājam nenāksies saskarties ar šo tehnisko pierakstu, jo visa parametru maiņa notiek grafiski. Var veidot arī kopijas no esošiem uzdevumiem. Salīdzinot ar vienkāršākajiem iepriekš apskatītajiem atvērtā koda produktiem, šeit paveras plašākas iespējas. Tiek piedāvāts automatizēt arī Java appletu un Flash objektu automatizētu testēšanu. Protams, arī dinamisku parametru ievadi no datubāzes, teksta faila vai skripta.

WebTest no iepriekš apskatītajiem rīkiem atšķiras ar daudz profesionālāku lietotāja saskarni, kas acīmredzami ir virzīta uz citu mērķauditoriju kā iepriekš apskatītie rīki. Šis patiešām ir rīks testētājiem, nevis izstrādātajiem, jo, lai veiktu kvalitatīvu automatizācijas scenāriju veidošanu, šeit nav nepieciešamas programmēšanas zināšanas un nav jāiedziļinās nekādos tekstuāla formāta skriptos. Visa darbība notiek lietotājam draudzīgā (subjektīvi) vidē, kur redzama

secīga komandu izpilde ar iespējām nomanīt parametrus, pievienot papildus rezultējošā dokumenta pārbaudes. Šeit gan arī atrodama pietiekami sarežģītu procedūru izveides iespēja, kuras pilnvērtīgai izmantošanai bez dokumentācijas studēšanas neiztikt.

Praktiskā lietošanā šis rīks būtu izmantojams izstrādes procesā, kad ir paredzēti atsevišķi cilvēku resursi testēšanai un šie cilvēki ir kvalificēti manuālās testēšanas speciālisti bez padziļinātām zināšanām programmēšanā. Ja nav šādu procesu un testēšanu veic programmēšanai tuvs personāls, daudz noderīgāks būs zemāka līmeņa rīku komplekts ar iespējām izmantot to pašu valodu, kas tiek izmantota izstrādei. Lietojot šo rīku, var sajust ierobežojumus veidojot sarežģītākus izpildes scenārijus un mēģinot atkalizmantot iepriekš veidotas darbības.

Vēl problēmas šī rīka pielietojumā var rasties dēļ atbalsta tikai vienai lietotāja konfigurāciju grupai – Windows 2000/XP/2003 un Internet Explorer. Kā jau minēts, tīmekļa aplikāciju popularitātes vien no veicinātājiem ir iespējas tās izmanot, aizmirstot par piesaisti izstrādātāja noteiktai platformai.

Šis rīks testu praktiskai izpildei izmanto jau iepriekš par izstrādātājam ne īpaši ērto atzīto pārlūka vadīšanu, darbinot to fonā. Tas pieprasa izstrādātāju veltīt savus izstrādes resursus testēšanai, kā arī jaudīgāku darba staciju. Iespēja darbināt testus uz servera centralizēti kopā ar citu izstrādātāju veidotajiem sniedz priekšrocības arī darba organizēšanā.

3.6 Rīku kopsavilkums

Aplūkojot šos rīkus, ir nepieciešams salīdzināt iespējas, kuras tie piedāvā, tāpēc ieskatam aplūkosim tuvāk apskatīto rīku parametrus. Zem tabulas aplūkojams katra parametra skaidrojums un pamatojums konkrētu vērtību nepieciešamībai.

Rīka nosaukums	**MaxQ**	**Selenium**	**HttpUnit**	**Sahi**	**AppPerfect WebTest**
Izstrādes vide	Java	JavaScript, XUL	Java	Java, JavaScript	Java
Platformas	Windows, Linux, Mac OS	Windows, Linux, Mac OS	Windows, Linux, Mac OS	Windows, Linux, Mac OS	Windows
Vides prasības	Java VM, Interneta pārlūks ar iespēju uzstādīt starpniek-serveri	Interneta pārlūks ar JavaScript atbalstu	Java VM	Java VM	Java VM, Internet Explorer
SSL atbalsts	Nav	Ir	Ir	Ir	Ir
Cookies atbalsts	Ir	Ir	Ir	Ir	Ir
Integrācija	jUnit		jUnit, Cactus, Ant	Ant	Ant
HTTP autentifikācija	Nav	Ir	Ir	Ir	Ir
Pārlūku atbalsts	Visi	Visi, Selenium IDE darbojas tikai uz Firefox	-	Visi	Internet Explorer
Skriptu valoda	Jython	Selenese, Selenium RC atbalsta vairumu populārāk o valodu	Java	JavaScript ar papildinājum iem	XML, labojumi tiek veikti grafiskā vidē
Strādā aiz starpniekserveriem	Jā	Jā	Jā	Jā	Jā
Bez lietotāja darbība	Ir	Ir	Ir	Ir	Jā
Ārējo bibliotēku izmantošana	Jython, pieejamas Java bibliotēkas	Nav	Java bibliotēkas	Nav	Nē
Datubāzes pieeja	Ar bibliotēku	Nav	JDBC	Nav	JDBC

Tabula 3-1 – Apskatīto rīku iespēju salīdzinājums

Tabula 3-1 attēlo dažādu testēšanas automatizācijas rīku sniegtās iespējas.

- Testēšanas rīka izstrādes vide ir nozīmīga gadījumā, ja ir nepieciešams veikt specifiskus rīka uzlabojumus atvērtā koda rīku gadījumā. Izmantojot komerciālus rīkus vēlams, lai tas būtu izstrādāts

līdzīgā vidē, kuras testēšanai tas tiks izmantots, taču tas nav obligāts priekšnosacījums

- Platformas, uz kurām rīku varēs izmantot ir nozīmīgas, lai sistēmas migrācijas laikā rastos mazāk savietojamības problēmu. Turklāt tīmekļa aplikāciju izstrādē viena projekta dalībnieki var izmantot dažādas izstrādes vides, tāpēc būtu ērti, ja visi varētu izmantot vienu testēšanas rīku.
- Vides prasības ir analoģiskas platformu pieejamībai
- SSL atbalsts ir nozīmīgs gadījumā, kad tiek izmantots drošais savienojums vai arī ir paredzami citi projekti, kuros tiks izmantota šī iespēja
- Sīkdatņu (*cookies*) atbalsts ir obligāta prasība, jo reti kura tīmekļa aplikācija iztiks bez lietotāja datu glabāšanas lietotāja pusē.
- Integrācija ar citiem testēšanas un izstrādes rīkiem ir ļoti nozīmīga rīka izvēlē un atvieglo paša testētāja darbību ar tiem.
- HTTP autentifikācija ir nozīmīga, ja tiek izmantots šis autentifikācijas veids. Noderīga iespēja rīkam, bet daudzi tīmekļa projekti bez tā iztiks.
- Pārlūku atbalsts ir nozīmīgs, jo šī brīža procesi liecina, ka populārākā tīmekļa pārlūka MS Internet Explorer tirgus daļa ir samazinājusies pietiekami, lai biznesa interesēs būtu veikt testēšanu dažādās lietotājs vidēs.
- Skriptu valoda ir nozīmīga, lai samazinātu apmācības līkni lietotājam un sniegtu plašākas izvēles iespējas valodas konstrukcijās un bibliotēku pieejā.
- Iespēja strādāt aiz starpniekservera ir nepieciešamība daudzos gadījumos, tāpēc iespējas atbalsts ir nozīmīgs.
- Bez lietotāja darbība ir obligāta prasība automatizācijai, jo iespēja izpildīt testus pie katras koda izmaiņas ir viens no praktiskiem automatizācijas pielietojumiem
- Ārējo bibliotēku pieejamība un pieeja datubāzei ļauj veidot daudz pilnīgākus automatizācijas scenāriju un paveikt vairāk.

3.7 Secinājumi un priekšlikumi

Labs testēšanas rīku komplekts ievērojami atvieglo testētāja ikdienu. Taču nevajag aizmirst testēšanas automatizācijas pamatmērķi – atbrīvot testētāju no garlaicīgiem un atkārtojamiem testiem, lai vairāk laika varētu veltīt sarežģītāku problēmu pētīšanai. Neviens testēšanas rīks, lai cik ērts un praktisks tas būt, nespēs pilnībā aizvietot kvalificētu testētāju. Mēģināšu pielietot savu esošo pieredzi un aplūkoto automatizācijas rīku iespējas, lai izvirzītu un pamatotu dažus priekšlikumus.

Ir svarīgi jau pašā izstrādes procesa sākumā apsvērt automatizācijas nepieciešamību un to, vai vispār ir vajadzība automatizēt testa darbības. Ņemsim par piemēru to pašu kontaktu formu. Šķiet, ka šis objekts ir samērā vienkāršs un neprasīs lielu piepūli to notestēt. Vairumā gadījumu tas tā arī ir – šī forma tiek notestēta, vai tā darbojas un tiek izmantota dažādos projektos kā viens no moduļiem. Tagad iedomājamies, ka kāds klients vēlas kontaktformu ar papildus laukiem. Nebūtu lietderīgi veidot jaunu objektu, vieglāk ir padarīt šo formu konfigurējamu (pieņemot, ka objektu konfigurācijas parametrizējamība sistēmā ir plānota un realizēta).

Šajā brīdī ir jāsāk apsvērt automatizācijas lietderīgumu – iepriekš izveidotais automatizācijas scenārijs ar izmaiņām papildus laukam būtu atvieglojis procesu. Šeit gan ir jāapsver to, kāda ir iespēja sabojāt esošo funkcionalitāti, veicot elementāru funkcionalitātes pievienošanu? Vai atmaksāsies testēšanu automatizēt? Prakse rāda, ka formai izmaiņas bieži nenotiks, kas liek secināt, ka ieguvums ir šaubīgs.

Tagad iedomājamies, ka šī forma tiek veidota ar formu ģenerēšanas sistēmas palīdzību, kur katrs lauks ir konfigurējams. Vai šoreiz būt jāautomatizē testēšana? Apsveram šādus faktus:

- Formu ģenerēšanas sistēma tiks veidota vienreiz ļoti universāla
- Jauniem laiku tipiem un uzvedības scenārijiem būtu jāparādās reti, jo šādu sistēmu mēs veidojām universālu
- Izveidoto formu notestēšanai vajadzīgos scenārijus mēs varēsim veidot pusautomātiski – mēs izveidosim vienu paraugu un pielabosim izmaiņas, kuras ir formu atšķirībās

Redzam, ka veidojot testu automatizāciju, darbs var atmaksāsies. Šeti varētu būt iemesls veidot regresijas testu, jo redzam, ka notestēt šo objektu mēs itin labi varētu automatizēti. Šeit gan jāņem vērā katra rīka īpatnības, veidojot un labojot testa piemērus. HttpUnit gadījumā mums būtu jākopē koda gabali (varētu organizēt funkcijās), MaxQ gadījumā būtu jāpievieno jauni parametri iesnigtajiem datiem. Lai notestētu, būs jāizveido daudz dažādas formu konfigurācijas, būtu tās visas jāizveido un jāievieto dažādās lapās, kas atkal ir papildus konfigurācijas parametri. Līdz nākamajai automatizēto testu izpildes reizei šeit daudz kas varētu būt mainījies, it sevišķi, ja nav pastāvīgas testa vides, kur šo automatizāciju veikt.

Šeit parādās uzturēšanas problēma, jo redzam, ka līdz ar testu komplektu ir jāsaglabā arī testēšanas vide. Ja mēs redzam, ka šāda situācija saglabāsies, tad mēs varēsim izmantot izveidoto skriptu regresiju testēšanai. Šeit saskatāma nepieciešamība pēc īpašas vides testēšanai. Jau sen ir zināms, ka nopietna izstrāde nevar notikt produkcijas vidē, bet ir sarežģīti veikt automatizētu testēšanu izstrādes vidē, kas ir ļoti mainīga ekosistēma. Ja manuālās testēšanas gadījumā mēs, ieraugot izmaiņas sistēmas uzstādījumos, spējam adekvāti reaģēt un vairumā gadījumā apiet problēmu, tad automatizēto testu pielietošanā videi ir jābūt absolūti nemainīgai. Ja tika izlabota drukas kļūda paziņojumā, tests apstāsies ar kļūdas paziņojumu. Mūsu formu ģenerācijas sistēmai ir jāsaglabā visas izveidotās formas, lai mēs varētu veikt regresijas testēšanu vajadzības gadījumā.

Cits gadījums būtu kāda cita forma, kura ir daudz sarežģītāka un darbības notiek vairākos soļos. Varam iedomāties, ka būtu grūti kvalificēti aizpildīt šādas formas vairākas reizes, turklāt piedomājot par to, vai šādi ievaddati pārstāv jaunu problēmas klasi. Pieņemsim, ka mums ir forma ar kuru mēs veidojam jaunu nekustamā īpašuma projektu un ir iespēja tam pievienot vairākas daļas – privātmājās rindu mājas un daudzdzīvokļu namus neierobežotā skaitā.

Ar savu automatizējamu testēšanas rīku mēs veicam aizpildīšanas scenāriju, bet neiedomājamies, ka kļūda varētu notikt tad, ja mēs veicam labojumus kādā jau izveidotā projekta apakšdaļā, ja ir pievienotas vairākas vien tipa apakšdaļas un kādā kļūdaini uzrakstītā SQL vaicājumā gaidītās 1 rindas vietā tiek atgrieztas vairākas. Mūsu automatizācijas scenārijs ir simtiem reižu aizpildījis mūsu sākotnēji iedomāto scenāriju, bet izpildes ceļš nav bijis pilnībā pārklāts. Redzam, ka patiešām viena pati automatizācija nevar nodrošināt pilnību problēmas pārklājumu.

Nākamā problēma ir mācīšanās līkne. Uzsākot automatizācijas procesa ieviešanu testēšanā, pilnīgi noteikti notiks testētāju efektivitātes samazināšanās. Tā kā vienību testu rakstīšana ir izstrādātāja pienākums, jūtams būs arī šis zaudētais laiks. Lai arī grafiska testu izveide varētu likties vienkāršāka, arī šeit neveiksmīga rīka izvēle var aizkavēt ātru apmācību. Visi teksti tiek ierakstīti kādā noteiktā valodā. Ja ar šiem testiem ir jāstrādā izstrādātājam, kas nepārzina šīs valodas īpatnības, uzreiz redzams ieguvums no iespējas rakstīt testus tajā pašā valodā, kur notiek izstrāde.

Testa scenāriju veidošanu vajadzētu uztvert kā jebkuru manuāla uzdevuma automatizēšanas procesa programmēšanu. Tieši tā pat ir prasību definēšanas fāze, plānošanas fāze, koda rakstīšana un testēšanas fāze. Šis pats process ir kā vēl viena manuālās testēšanas aktivitāte, jo gatavs testēšanas scenārijs nozīmē, ka tas izpildās bez kļūdām. Šeit gan parādās cita problēma. Labs testētājs ne vienmēr būs labs automatizācijas testu veidotājs, jo šeit ir nepieciešamas programmēšanas prasmes un zināšanas par izstrādes vidi.

Tīmekļa aplikāciju izstrādē galvenā atšķirība no standarta darbvirsmas aplikācijām ir tā, ka to darbības vide nedarbojas pēc vienota standarta. Jā arī darbvirsmas aplikācijām ir jāņem vērā dažādās operētājsistēmas versijas un vides parametri, taču šīs problēmas ir vieglāk risināmas. Tīmekļa aplikācijas darbojas vidē, kas tām īsti nav paredzētas. HTML iesākumā bija paredzēts dokumentu attēlošanai, HTML formas uzsāka tīmekļa aplikāciju iespēju veidošanu.

Taču ar laiku šīs iespējas nebija pietiekamas – bija nepieciešams elements, kas ļautu izvēlēties datumu no kalendāra, izvēlēties vērtību no skalas ar ritjoslu un citi darbvirsmas aplikācijām ierasti grafiskie lietotāja saskarnes elementi. To visu var panākt ar JavaScript un CSS tehnoloģiju atbalstu, taču šeit sākas problēma. Ne visi interneta pārlūki atbalsta visus specifikācijās norādītos standartus un daži standartus interpretē citādi. Piemēram Internet Explorer kādreiz elementa izmērus atbalstīja līdzīgi kā stāstā par dzelzceļa sliedēm[1]. Kamēr visi pārējie izmēru definēja pēc ārējām dimensijām, IE to darīja pēc iekšējām. Grafiski pieļautās neprecizitātes ir viegli pamanīt un tās nav pieļaujamas gala produktā.

[1] Kad Krievijas impērijā tika veidots dzelzceļu tīkls, tas tika izveidots platāks kā pārējā Eiropā. Oficiālā versija apgalvo, ka tas tika veidots militāru iemeslu dēļ, lai iebrucēji nevarētu izmantot dzelzceļa infrastruktūru. Taču populārs ir stāsts par to, ka Eiropā sliežu ārpusē noņemtais mērs tika izmantots kā iekšējais attālums, būvējot Maskavas – Sanktpēterburgas dzelzceļa līniju.

Praktiskie novērojumi liecina, ka šo pārlūku savstarpējo savietojamību īsti nav iespējams automatizēti testēt, jo visbiežāk tās ir vizuālās atšķirības, kuras praktiski tiek testētas ar manuālās testēšanas metodi. Problēmas var rasties JavaScript dažādā atbalsta gadījumos, kuru var automatizēti atklāt tikai gadījumos, kad pats skripts darbojas konkrētajā problemātiskajā pārlūkā.

Ja tiek izmantoti tikai HTTP dati, nav iespējams noteikt, vai lietotājam bija iespējams ievadīt kādu konkrētu vērtību, vai viņš nespēja to norādīt tāpēc, ka tika izmantota kāda specifiska funkcija, kuru nav ieviesuši visi pārlūku izstrādātāji vai arī tika izmantots pārlūks bez JavaScript iespējām (teksta režīma pārlūks lynx) vai minimālām (mobilo iekārtu dažādi pārlūki) JavaScript atbalsta iespējām.

Veicot automatizācijas skriptu izpildi konkrētā pārlūkā mēs testējam tikai šo konkrēto pārlūka versiju, tāpēc automatizācijas izpilde ir jāveic dažādās vidēs. Tas diemžēl nav iespējams daudzos risinājumos, kuri atbalsta tikai konkrētu pārlūku, kā piemēram, mūsu apskatītais AppPerfect WebTest. Kā rāda statistika [41], populārākais pārlūks ieņem no 60 līdz 90% tirgus daļu ar tieksmi samazināties, tāpēc biznesa prasības spiež atbalstīt pēc iespējas plašāku konfigurāciju kopu.

Skriptu veidošana automatizācijai ir ilglaicīgs ieguldījums – tūlītējs rezultāts iespējams tikai atsevišķos gadījumos, kā jau iepriekš minētajā situācijā ar formu ģenerēšanas sistēmu. Par galveno nosacījumu kļūst iespēja uzturēt šos automatizācijas testus. Ja tas nav iespējams, ir grūti saskatīt ieguvumu no tiem.

Redzot, ka ir risks veltīgi zaudēt laika resursus, ieviešot automatizāciju, vajadzētu sākt ar nelielu uzdevumu veidošanu, lai mācīšanās no kļūdām notiktu, patērējot mazāk resursu. Tieši tāpat vajadzētu iekļaut automatizācijas procesu izstrādes procesā. Cilvēka dabā ir grūti pieņemt jaunus pienākumus, bet automatizētās testēšanai būtu jākļūst par izstrādes aktivitāti, nevis tikai nodarbi, kas notiek blakus izstrādei. Tā kā testētāji pārsvarā izmantos augstākā līmeņa testēšanas rīkus, vienību testēšana ir jāveic izstrādātājiem. Ja testētājs nav kvalificēts rakstīt programmatūras kodu, viņš nevarēs arī veikt testēšanas skriptu izveidi, tāpēc ir jāapsver izstrādātāju iesaistīšanu automatizācijas procesā.

Lai iegūtu ātrus rezultātus un atsvērtu zaudējumus, kas radušies, veicot neveiksmīgus testu veidošanas mēģinājumus, vajag mēģināt saskatīt lielākās manuālās testēšanas problēmas un izmantot iespēju automatizēt tās, pat ja šie testi neiederēsies regresijas testu kopā. Šādu skriptu izmaksas ir zemas, bet ir

iespējams gūt pieredzi ar automatizācijas rīkiem, kas ir svarīgāk par iespējamiem laika zaudējumiem.

Ir grūti sagaidīt rezultātus tūlīt pēc automatizācijas ieviešanas izstrādes procesā. Tas nedrīkst aizstāt manuālo testēšanu, labu plānošanu un kvalitatīvu izstrādi. Pirmajās produkta izlaidēs pat būs novērojama termiņu pārpildīšana, jo šajā posmā vēl tikai tiek veidoti automatizācijas skripti un notiek apmācība, iepazīšanās ar rīkiem, pirmās kļūdas un mācīšanās no tām.

Populārākie automatizācijas rīki piedāvā ieraksta un atskaņošanas iespējas, kas ir lietotājam-testētājam skaidrāk saprotamākas un patīkamākas lietošanā, taču patiesībā tas ir tikai pēdējais automatizācijas solis. Automatizācija ir pieejama arī daudz zemākos koda izstrādes līmeņos, kur kļūdas būtu iespējams atklāt daudz ātrāk, jo programmētājs veicot sava koda vienību testēšanu var daudz labāk saplānot savus testus tā, lai panāktu pilnīgāku koda pārstaigāšanu.

Veicot virspusēju funkcionālo testēšanu, kas patiesībā tāda melnās kastes testēšana vien ir, mēs konkrētos izpildes zaros iesiesim ar mazāku varbūtību. Tomēr pluss šādam testēšanas stilam ir tāds, ka tas precīzāk attēlo iespējamo lietotāja uzvedību lapā un spēs atklāt populārākās kļūdas. Šeit es izmantoju pieņēmumu, ka biznesam daudz svarīgāk ir izlabot kādu pavisam triviālu kļūdu, ko redz 90% aplikācijas lietotāju kā salabot datu pazušanas kļūdu, kas notiek, kad kāds lietotājs veic īpašu darbību kombināciju. Ja tiek izmantots šis testēšanas veids, pārāk vēlu atklāto kļūdu labošana izmaksās ārkārtīgi dārgi.

Tomēr problemātisks ir arī otrs piegājiens galējībā. Labi ir sākt testēt pēc iespējas ātrāk, bet problēmas var rasties uzturot automatizētos testus. Izstrādes sākumā var notikt kardinālas izmaiņas, kas liks pārrakstīt testus. Daļa testu varētu būt arī veltīgi veikti, ja tiek atmesta kāda sistēmas funkcionalitāte, kas sevišķi iespējams gadījumā, kad izstrāde nenotiek pēc stingri izstrādātas specifikācijas.

Ieraksta un atskaņošanas rīkiem ir redzams veiksmīgs pielietojums, ja automatizētie testi nebūs jāuztur un jāizmanto ļoti mainīgā vidē, jo tie tomēr ir ļoti grūti organizējami. Manuāli rakstītie testi parasti ir daudz vienkāršāki un labāk padodas modularizēšanai. Svarīga ir pašu testu rakstīšana – ļoti daudz kļūdu atrod, veidojot pašus testus, to darbināšana regresiju ķeršanas nolūkos meklē cita tipa kļūdas, kuras tomēr ir mazāk sastopamas.

Paša procesa ieviešanā būtu svarīgi, lai būtu persona, kas aktīvi atbalsta un stimulē šos procesus. Šai personai būtu nepieciešamas līdera dotības, pieredze

testēšanā, programmēšanas prasmes, ideālā gadījumā tas varētu būt kāds sistēmu analītiķis. Šīs personas nozīmība ir liela, jo no vadības viedokļa rezultāti neparādās tik strauji, kā plānots. Būtu kļūda norakstīt veikto ieguldījumu automatizācijā, taču reizēm rezultāti ir gaidāmi ilgāk nekā pietiek pacietības investēt resursus.

Šie ir ieteikumi un pārdomas par automatizētās testēšanas iespējām un metodēm, kuri radušies iepazīstoties ar automatizācijas teoriju un veicot konkrētu rīku izmēģināšanu. Tāpat savi novērojumi veikti veicot automatizētas testēšanas aktivitāšu ieviešanu tīmekļa programmatūras izstrādē. Daļa no šiem apgalvojumiem nebalstās uz stingriem faktiem, jo daudzos gadījumos nav iespējams veikt precīzus mērījumus par to, kādus rezultātus dos konkrēta manuāla uzdevuma automatizēšana. Šajos gadījumos rekomendējamā rīcība ir atbildes uz teorijas daļā apskatītajiem kontroljautājumiem. Automatizētā testēšana ir rīks, kas jāprot izmantot. Ieguvumi ir ilgtermiņa, tāpēc šie apsvērumi ir jāņem vērā tikai kā rekomendācija.

Noslēgums

Automatizētai testēšanai ir potenciāli ļoti liela nozīme tīmekļa aplikāciju kvalitātes nodrošināšanā, turklāt veiksmīgi iekļaujoties to izstrādes procesā, kas ir ar uzsvaru uz jauno iespēju pēc iespējas ātru nodošanu lietotājam.

Tomēr automatizētas testēšanas nozīmi tīmekļa aplikāciju testēšanā nevajadzētu pārvērtēt, jo tas nav ārkārtīgi universāls rīks, kas pārklātu visus iespējamos lietotājs ievaddatus un koda izpildes ceļus. Tas ir tikai tik gudrs, cik gudrs ir tā veidotājs un nedarīs neko vairāk kā tam paredzēts. Tāpēc rīku izmantošanai vajadzētu aprobežoties tikai ar uzdevumiem, kuri kavē testētāju nodarboties ar sarežģītāku testēšanas problēmu veikšanu.

Veicot izpētes darbu par pieejamajiem automatizācijas rīkiem, nācās secināt, ka arī tīmekļa programmu testēšanai ir pieejai milzumdaudz komerciālu un bezmaksas rīku. Apskatot atšķirību starp profesionāliem komerciāliem rīkiem un par brīvu pieejamiem, var saskatīt diezgan nopietnu atšķirību. Galvenie ieguvumi, izvēloties maksas rīku ir iespējama laika ietaupīšana, labāki statusa ziņojumi, integrācijas dažādos izstrādes rīkos. Taču komerciālo rīku efektīvai izmantošanai ir nepieciešama pieredze vienkāršu automatizācijas procesu veidošanā, tāpēc aktivitāšu ieviešanu izstrādes procesā vajadzētu uzsākt ar kāda bezmaksas rīka izmantošanu, lai vēlāk varētu skaidrāk apzināties prasības rīkiem, par kuriem varētu arī maksāt naudu.

Darba pēdējā nodaļā pēc rīku apskates ir apkopotas iegūtās atziņas, kuras radās gan izmēģinot rīku piedāvātās iespējas, gan plānojot testēšanas automatizācijas procesu ieviešanu. Mainījās mans viedoklis par ieguvumiem no automatizētās testēšanas un radās sapratne par to, kādas lietas nebūs automatizējamas, kādas nav automatizējas izmaksu dēļ. Vairumā gadījumu automatizācija ir tikai manuālās testēšanas papildinājums.

Šajā darbā tuvāk apskatītie rīki nebūt nav vienīgie iespējamie risinājumi, taču tie labi ieskicē automatizācijas rīku veidošanas tehnisko pieeju. Bieži vien kāda rīka izvēli var noteikt kādas iespējas ērtāka realizācija, tāpēc autora apskatīto rīku kopumam ir tikai rekomendējošs raksturs.

Izmantotās literatūras saraksts

1. Marick Brian / When Should a Test Be Automated?, Testing Foundations, 1998, 1-2. lpp, pieejams Internetā: http://www.testing.com/writings/automate.pdf
2. The Economic Impacts of Inadequate Infrastructure for Software Testing / RTI Health, Social, and Economics Research, 2002, Pieejams internetā: http://www.nist.gov/director/prog-ofc/report02-3.pdf
3. Raymond, Eric S. / The Cathedral & the Bazaar, O'Reilly, 2001, 102-107. lpp
4. Ash Lydia / The Web Testing Companion: The Insider's Guide to Efficient and Effective Tests, John Wiley & Sons, 2003
5. Slashdot effect [Tiešsaiste] / Wikipedia, 12.05.2006, pieejams internetā http://en.wikipedia.org/wiki/Slashdot_effect
6. Alexa.com [Tiešsaiste] / Wikipedia, 02.05.2006, pieejams internetā: http://en.wikipedia.org/wiki/Alexa.com
7. Related info for sketchup.com [Tiešsaiste] 03.05.2006 , pieejams internetā http://alexa.com/data/details/traffic_details?q=&url=sketchup.com
8. Federal Accessibility Standards for Web-based Intranet and Internet Information and Applications [Tiešsaiste], 21.12.2000, pieejams internetā http://www.usability.gov/accessibility/508.html
9. Ajax: A New Approach to Web Applications / Aut. Jesse James Garrett, 18.02.2005, publikācija Adaptive Path, pieejams internetā http://www.adaptivepath.com/publications/essays/archives/000385.php
10. Very Dynamic Web Interfaces / Aut. Drew McLellan, 09.02.2005, publikācija O'Reilly XML.com, pieejams internetā http://www.xml.com/pub/a/2005/02/09/xml-http-request.html
11. How Many Bugs Do Regression Tests Find? [Tiešsaiste] / Aut. Brian Marick, Testing Foundations, 16.06.2004, pieejams internetā http://www.testingcraft.com/regression-test-bugs.html
12. BrowserCam [Tiešsaiste], pieejams internetā, 02.05.2006, http://www.browsercam.com/
13. Massol Vincent / jUnit in action, Manning, 2004, 17-20.lpp
14. IEEE Standard for Software Unit Testing / American National Standards Institute, 1993, 4. lpp
15. HTML 4.01 Specification [Tiešsaiste] / World Wide Web Consortium, 1999, Pieejams internetā: http://www.w3.org/TR/html4/
16. Kanglin Li, Mengqi Wu /Effective GUI Test Automation: Developing an Automated GUI Testing Tool, Sybex, 2004, 23 - 25.lpp

17. The Ten Most Critical Web Application Security Vulnerabilities / The Open Web Application Security Project, 2004. Pieejams internetā: http://www.owasp.org/documentation/topten.html
18. Frankland Jane /Automated Penetration Testing - False Sense of Security [Tiešsaiste], IT Observer , 06.08.2004
19. Shihjong Kuo / Execute Disable Bit Functionality Blocks Malware Code Execution [Tiešsaiste], 2003 /Pieejams internetā: http://www.intel.com/cd/ids/developer/asmo-na/eng/149308.htm
20. Astels David / Test-Driven Development: A Practical Guide, Prentice Hall, 2003, 79-82. lpp
21. Predescu Ovidiu, Turner Jeff /Anteater [Tiešsaiste], 2003, Pieejams internetā: http://aft.sourceforge.net/
22. Apache HTTP Test Project [Tiešsaiste], 2005 / Pieejams internetā: http://httpd.apache.org/test/
23. AppPerfect WebTest / DevTest [Tiešsaiste], 2006, pieejams Internetā: http://www.appperfect.com/products/index.html
24. Jakarta Cactus [Tiešsaiste], 2004, pieejams Internetā: http://jakarta.apache.org/cactus/
25. HtmlUnit introduction [Tiešsaiste], 2006, pieejams internetā: http://htmlunit.sourceforge.net/
26. HttpUnit, 2006 [Tiešsaiste], pieejams internetā: http://httpunit.sourceforge.net/
27. Introduction to JsUnit [Tiešsaiste], 23.03.2006 / pieejams internetā: http://www.jsunit.net/
28. Apache JMeter [Tiešsaiste], 2005, pieejams internetā: http://jakarta.apache.org/jmeter/
29. JSpider [Tiešsaiste], 2003, pieejams internetā: http://j-spider.sourceforge.net/
30. About JTidy [Tiešsaiste], 14.09.2004, pieejams internetā: http://jtidy.sourceforge.net/
31. jUnit overview [Tiešsaiste], 2004, pieejams internetā: http://www.junit.org/
32. Bitmechanic MaxQ [Tiešsaiste], 2005, pieejams internetā: http://maxq.tigris.org/
33. NUnitAsp - ASP.NET unit testing [Tiešsaiste], 09.08.2005, pieejams internetā: http://nunitasp.sourceforge.net/
34. Sahi, Introduction [Tiešsaiste], 16.05.2006 , pieejams internetā: http://sahi.sourceforge.net/
35. OpenQA: Selenium [Tiešsaiste], 2006, piejams internetā: http://www.openqa.org/selenium-ide/
36. PHPUnit2 [Tiešsaiste], 01.05.2006, pieejams internetā: http://pear.php.net/package/PHPUnit2/

37. TestMaker Platform The Test Automation Solution For Service Environments Whitepaper, 2004, pieejams internetā: http://downloads.pushtotest.com/TestMakerPlatform.pdf

38. utPLSQL Project Summary [Tiešsaiste], 10.06.2003, pieejams internetā: http://utplsql.sourceforge.net/

39. Watij - Web Application Testing in Java [Tiešsaiste], 25.05.02006, pieejams internetā: http://watij.xwiki.com/xwiki/bin/view/Main/WebHome

40. OWASP WebScarab Project [Tiešsaiste], 30.05.2006, pieejams internetā: http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project

41. Browser Statistics Month by Month [Tiešsaiste], 26.05.2006 / W3Schools, http://www.w3schools.com/browsers/browsers_stats.asp

Patstāvības apliecinājuma forma

Apliecinājums

Ar šo es apliecinu, ka šodien iesniegto bakalaura darbu es esmu veicis pašrocīgi un esmu izmantojis tikai tajā norādītos palīglīdzekļus.

Rīgā, 2006. gada 5. jūnijā, Paraksts:

/M. Bruņenieks/

1.pielikums – Vārdnīca

Akcepta tests – minimālie kritēriji, lai produkts tiktu pieņemts un varētu pāriet pie nākamās fāzes – cita līmeņa testēšanas, iespēju pievienošanas, piegādes klientiem.

Integrācijas testēšana – tests, kas tiek veikts, kad tiek izveidota kāda jauna funkcionalitāte un ir jānosaka, vai tā strādā (integrējas) esošajā sistēmā.

Lietojamības testēšana – tests, kas nosaka, vai lapas navigācija ir intuitīva un saskarne saprotama.

Melnās kastes testēšana – testu izpildīšana, nepārzinot vai neizmantojot zināšanas par programmas iekšējo uzbūvi. Rezultāts ir tuvāks īstu lietotāju kļūdām un lietošanas stilam.

Pieejamības testēšana – tests, kas nosaka, vai produkts atbilst pieejamības standartiem, kas izstrādāti, lai cilvēki ar veselības traucējumiem varētu izmantot konkrēto produktu.

Pieņemšanas testēšana – tests, kas nosaka, vai ir izpildītas pasūtītāja prasības un var pāriet uz nākamo produkta dzīves cikla fāzi.

Regresijas testēšana – tests, kas nosaka, vai pievienotā funkcionalitāte nesabojā kādu no esošajām funkcijā, kas iepriekš ir stabili strādājusi.

Savietojamības testēšana – testi, kas nosaka, cik labi tīmekļa programmas attēlojas dažādās klienta sistēmas konfigurācijās – interneta pārlūkos, operētājsistēmas valodās, ekrāna izšķirtspējās, tajā skaitā uz mobilajām iekārtām.

Validācijas testēšana – tests, kas nosaka, vai iegūtais dokuments (HTML, XML, WML) atbilst DTD (Dokumenta tipa definīcijai). HTML gadījumā tas var uzlabot savietojamību, XML gadījumā neatbilstošs dokuments nevar tikt izmantots tam paredzētajiem mērķiem.

2.pielikums – Testu scenāriji

2.1. MaxQ

Šajā pielikumā attēlots vienas no automatizācijas programmām MaxQ veidotais automatizācijas scenārijs, kurš trīs reizes ar dažādiem ievaddatiem iesniedz kontaktformu.

```
# Generated by MaxQ [com.bitmechanic.maxq.generator.JythonCodeGenerator]
from PyHttpTestCase import PyHttpTestCase
from com.bitmechanic.maxq import Config
global validatorPkg
if __name__ == 'main':
    validatorPkg = Config.getValidatorPkgName()
# Determine the validator for this testcase.
exec 'from '+validatorPkg+' import Validator'


# definition of test class
class test1(PyHttpTestCase):
    def runTest(self):
        self.msg('Test started')
        params = [
            [('''feedback_name''', '''Juris Kalns'''),
            ('''feedback_email''', '''jurka'''),
            ('''feedback_phone''', ''''''),
            ('''feedback_theme''', '''10'''),
            ('''feedback_text''', '''Ko nu?'''),
            ('''feedback_submit''', ''''''),],
            [('''feedback_name''', '''Juris Kalns'''),
            ('''feedback_email''', '''jurka@mail.lv'''),
            ('''feedback_phone''', ''''''),
            ('''feedback_theme''', '''10'''),
            ('''feedback_text''', '''Ko nu?'''),
            ('''feedback_submit''', ''''''),]
           [('''feedback_name''', '''Juris Kalns'''),
            ('''feedback_email''', '''jurka@mail.lv'''),
            ('''feedback_phone''', ''''''),
            ('''feedback_theme''', '''asdf'''),
            ('''feedback_text''', '''Ko nu?'''),
            ('''feedback_submit''', ''''''),]
        ]
        res = ['''Lūdzu, ievadiet korektu e-pasta adresi''','''Jūsu iesniegtā informācija
ir saņemta''','''Lūdzu, izvēlieties kategoriju''']
        for x in range(len(res))
            self.msg("Testing URL: %s" %
self.replaceURL('''http://test.lv/lv/contacts/'''))
            url = "http://test.lv/lv/contacts/"
            Validator.validateRequest(self, self.getMethod(), "post", url, params)
            self.post(url, params[x])
            self.msg("Response code: %s" % self.getResponseCode())
            self.assertEquals("Assert number 1 failed", 200, self.getResponseCode())
            self.assertTrue(self.responseContains(re[x]))
            Validator.validateResponse(self, self.getMethod(), url, params)


    # ^^^ Insert new recordings here.  (Do not remove this line.)


# Code to load and run the test
if __name__ == 'main':
    test = test1("test1")
    test.Run()
```

2.2. Selenium IDE

Šajā pielikumā attēlots vienas no automatizācijas programmām Selenium IDE veidotais automatizācijas scenārijs veic aizpildītas kontaktformas iesniegšanu.

```
<html>
<head><meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Test2</title></head>
<body>
<table cellpadding="1" cellspacing="1" border="1">
<thead>
<tr><td rowspan="1" colspan="3">test2</td></tr>
</thead><tbody>
<tr><td>open</td><td>/lv/contacts/</td><td></td></tr>
<tr><td>assertTitle</td><td>Nams24</td><td></td></tr>
<tr><td>type</td><td>feedback_email</td><td>jurka@burka.lv</td></tr>
<tr><td>type</td><td>feedback_phone</td><td>2348235</td></tr>
<tr><td>select</td><td>feedback_theme</td><td>label=Ieteikumi</td></tr>
<tr><td>type</td><td>feedback_text</td><td>ASD</td></tr>
<tr><td>clickAndWait</td><td>//input[@value='Nosūtīt']</td><td></td></tr>
<tr><td>assertTitle</td><td>Kontaktforma</td><td></td></tr>
<tr><td>type</td><td>feedback_name</td><td>Juris Krūms</td></tr>
<tr><td>assertTextPresent</td><td>Paldies,       ka       sazinājāties       ar
mums!</td><td></td></tr>
</tbody></table>
</body>
</html>
```

2.3. HttpUnit

Šajā pielikumā attēlots vienai no automatizācijas programmām HttpUnit rakstītais testa scenārijs, kurš veic korektu formas iesniegšanu.

```
public void testContactForm() throws Exception {
        WebConversation     conversation = new WebConversation();
        WebRequest  request = new PostMethodWebRequest(
            "http://test.lv/lv/kontakti" );

        WebResponse response = conversation.getResponse( request );
        WebForm contactForm = response.getForms()[0];
        request = contactForm.getRequest();
        request.setParameter( "feedback_name", "Juris Kalns" );
        request.setParameter( "feedback_email", "jurka@mail.lv" );
        request.setParameter( "feedback_phone", "3254325" );
        request.setParameter( "feedback_theme", "11" );
        request.setParameter( "feedback_text", "Ko nu?" );
        request.setParameter( "feedback_submit", "" );
        response = conversation.getResponse( request );
        assertTrue( "Form not accepted",
                    response.getText().indexOf( "Paldies, ka sazinājāties ar mums!" ) != -1
);

        assertEquals( "Page title", "Kontakti", response.getTitle() );
}
```

2.4. TestInput klase:

Šajā pielikumā attēlota viena vienkārša Java klase, kura varētu tikt izmantota kādā izstrādātāja līmeņa automatizētas testēšanas programmā, lai veidotu ievaddatu masīvus, kuri aptvertu pēc iespējas plašākas tīmekļa programmu ievaddatu problēmas.

```
import java.util.*;

public class TestInput {

  private Vector cases;

  /**
  * konstruktors
  */
  public TestInput() {
    cases = new  Vector();
  }

  /**
  * Atgriež Vector objektu, kurš satur ievaddatu gadījumus String ievaddatiem
  */
  public Vector getCasesVector(String input) {
    cases.add(input);
    cases.add("");
    cases.add(" ");
    cases.add(" " + input);
    cases.add(input + "");
    cases.add(" " + input + "");
    cases.add("a");

    // SQL
    cases.add("'" + input);
    cases.add("$" + input);
    cases.add("#" + input);

    // HTML
    cases.add("\"" + input);
    cases.add("<" + input);
    cases.add(">" + input);
    cases.add("<b>" + input + "</b>");
    cases.add("<h1>" + input + "</h1>");
    cases.add("&lt;b&gt;" + input + "&lt;/b&gt;");

    // JavaScript
    cases.add("\"><script>alert (.Hello.)</script>" + input);
    cases.add("<script>alert ('Hello')</script>" + input);
    cases.add("<Script>alert ('Hello')</script>" + input);
    cases.add("<sCript>alert ('Hello')</script>" + input);
    cases.add("&#60;script&#62;alert('Hello')&#60;&#47;script&#62;" + input);
    cases.add("%22><script%20for=window                                %20event=%22onload()%22>
document.write(%22Hello%22);document.close();</script>Hello%22);document.close();</script>
.write(%22Hello%22);document.close();</script>" + input);

    return cases;
  }

  /**
  * Atgriež Vector objektu, kurš satur ievaddatu gadījumus int ievaddatiem
  */
  public Vector getCasesVector(int input) {
    cases.add(String.valueOf(input * 10^6));
    cases.add(String.valueOf(input * -1));

    // Skaitlis ar komatu
    cases.add(String.valueOf(input * 1d));

    return getCasesVector(String.valueOf(input));
  }
}
```

2.5. Sahi

Šajā pielikumā attēlots vienas no automatizācijas programmām Sahi veidotais automatizācijas scenārijs cikliski veic aizpildītas kontaktformas iesniegšanu. Šis ir modificēts kods, kurš tiek organizēts funkcijās, lai veicinātu atkalizmantojamību.

```
function submitForm($name, $email, $phone, $theme, $txt, $msg) {
	_setValue(_textbox("feedback_name"), $name);
	_setValue(_textbox("feedback_email"), $email);
	_setValue(_textbox("feedback_phone"), $phone);
	_setSelected(_select("feedback_theme"), $theme);
	_setValue(_textarea("feedback_text"), $txt);
	_click(_submit("Nos\u016bt\u012bt"));
	_assertTrue(_containsText(_byId("col-a"), $msg),"Fail");

}


var params = new Array();
params[0] = new Array("Juris
Kalns","jurka@mail.lv","4343554","S\u016bdz\u012bbas","Neraadaas latvieshu
burti.","Paldies, ka sazi");
params[1] = new Array("Juris
Kalns","jurkamiallv","4343554","S\u016bdz\u012bbas","Neraadaas latvieshu burti.","dzu
aizpildiet visus oblig");
params[2] = new Array("","jurkamiallv","4343554","S\u016bdz\u012bbas","Neraadaas latvieshu
burti.","dzu aizpildiet visus oblig");
params[3] = new Array("","jurkamiallv","4343554","S\u016bdz\u012bbas","","dzu aizpildiet
visus oblig");

for (i=0;i<4;i++) {
	_navigateTo("http://nams24.c4.lv/lv/contacts/")
	submitForm(params[i][0],params[i][1],params[i][2],params[i][3],params[i][4],params[
i][5]);
}
```

Šis ir Sahi uzģenerētais JavaScript kods, kurš veic pārlūka vadību:

```
function submitForm($name, $email, $phone, $theme, $txt, $msg) {
sahiAdd("sahi_setValue(sahi_textbox(\"feedback_name\"), "+s_v($name)+");",
"nams_script.sah?n=2")
sahiAdd("sahi_setValue(sahi_textbox(\"feedback_email\"), "+s_v($email)+");",
"nams_script.sah?n=3")
sahiAdd("sahi_setValue(sahi_textbox(\"feedback_phone\"), "+s_v($phone)+");",
"nams_script.sah?n=4")
sahiAdd("sahi_setSelected(sahi_select(\"feedback_theme\"), "+s_v($theme)+");",
"nams_script.sah?n=5")
sahiAdd("sahi_setValue(sahi_textarea(\"feedback_text\"), "+s_v($txt)+");",
"nams_script.sah?n=6")
sahiAdd("sahi_click(sahi_submit(\"Nos\\u016bt\\u012bt\"));", "nams_script.sah?n=7")
sahiAdd("sahi_assertTrue(sahi_containsText(sahi_byId(\"col-a\"),
"+s_v($msg)+"),\"Fail\");", "nams_script.sah?n=8")
}
var params = new Array();
params[0] = new Array("Juris
Kalns","jurka@mail.lv","4343554","S\u016bdz\u012bbas","Neraadaas latvieshu
burti.","Paldies, ka sazi");
params[1] = new Array("Juris
Kalns","jurkamiallv","4343554","S\u016bdz\u012bbas","Neraadaas latvieshu burti.","dzu
aizpildiet visus oblig");
params[2] = new Array("","jurkamiallv","4343554","S\u016bdz\u012bbas","Neraadaas latvieshu
burti.","dzu aizpildiet visus oblig");
params[3] = new Array("","jurkamiallv","4343554","S\u016bdz\u012bbas","","dzu aizpildiet
visus oblig");
for (i=0;i<4;i++) {
sahiAdd("sahi_navigateTo(\"http://nams24.c4.lv/lv/contacts/\")", "nams_script.sah?n=19")
submitForm(params[i][0],params[i][1],params[i][2],params[i][3],params[i][4],params[i][5]);
```

Reģistrācijas lapa

Bakalaura darbs izstrādāts
LU Datorikas nodaļā

Autors:
Fizikas un matemātikas
fakultātes students Mārtiņš Bruņenieks .....
St. apl. Nr. DatZ 020032, 2006. g. .....

Darba vadītājs
Dr. sc. comp. Guntis Arnicāns .....

Recenzents
Dr. habil. sc. comp Audris Kalniņš .....

Darbs iesniegts Datorikas nodaļā 2006. g. .....

Pieņēma sekretāre .....

Aizstāvēts datorzinātņu bakalaura pārbaudījumu komisijas sēdē

2006.g. . . . . .ar atzīmi. . . . . .

Kursa pārbaudījumu komisija . . . . . .