LATVIJAS UNIVERSITĀTE
FIZIKAS UN MATEMĀTIKAS FAKULTĀTE
DATORIKAS NODAĻA

VERSIJU KONTROLE – PROBLĒMAS UN RISINĀJUMI

MAĢISTRA DARBS

Autors: Irina Gerasimenko
Stud. apl. Nr. Prog 020007

Darba vadītāja: Laila Niedrīte
Dr.sc.comp.

RĪGA 2008

Anotācija

Darbs ir veltīts versiju kontroles pārvaldības problēmām. Tas piedāvā ieskatu programmatūras konfigurācijas pārvaldības funkcionalitātei. Kā arī ir dziļi apskatīti versiju kontroles termini. Darbā tiek izanalizēta Transmaster informācija sistēma, tas apakšsistēmu versiju kontroles stāvoklis un identificētas tas versiju kontroles pārvaldības problēmas. Bija izpētītas versiju kontroles eksistējošas zarošanas metodes un piedāvāta sava metode, kas arī bija veiksmīgi ieviestā Transmaster apakšsistēmā.

Abstract

Current work is devoted to version control problems. It suggests overview of software configuration managements' functionality. Also is deeply inspected version control related terminology. In work is analyzed Transmaster information system, its' subsystems' version control state and identified version control management problems. It had researched version control existing branching methods and offered own method, which was successfully implemented into Transmaster subsystem.

Аннотация

Данная работа посвящена проблемам организации управления версиями. Она описывает функциональность управления конфигурационной программатурой. А так же углублённо рассматривает терминологию связанную с управлением версиями. В работе проанализирована информационная система Transmaster, её состояние организации управления версиями и идентифицирует проблемы связанные с управлением версиями. Были рассмотрены существующие методы разделения на ветви и предложен свой способ, который был удачно реализован в Transmaster подсистеме.

Autoreferāts

Autore strādā par programmētāju izstrādes projektā. Šī projekta produkts tiek izstrādāts jau vairāk par 15 gadiem. Transmaster sistēmai uz šo brīdi ir izveidotas 3 versijas. Patreiz autore labo kļūdas Transmaster CMS2 apakšsistēmā un piedalās izstrādē Transmaster CMS3 apakšsistēmā. Abas sistēmas vēsture par objektu modifikācijām tiek uzturēta ar versiju kontroles rīkiem.

Autore savā maģistra darbā ir izpētījusi PKP (SCM) jēdzienu un to visu funkcionalitāti. Darbā ir pārskaitīti un aprakstīti versiju kontroles termini. Tika izanalizēts patreizējais CMS2 un CMS3 stāvoklis un noteiktas problēmas. Darba galvenais mērķis bija piedāvāt risinājumu versiju kontroles zaru glabāšanas metodi, kas arī bija izpildīts. Autore izpētīja esošas zarošanas metodes, salīdzināja tas un piedāvāja savu zarošanas metodi, balstītu uz jau eksistējošam. Šī metode bija veiksmīgi ieviesta projekta versiju kontroles pārvaldībā.

Kā arī darba atvieglošanai autore piedāvā skriptu, kas veic nepieciešamas pārbaudes un zaru sapludināšanu.

Saturs

1 IEVADS 9
2 SCM – PROGRAMMATŪRAS KONFIGURĀCIJU PĀRVALDĪBA 10
2.1 Kāpēc ir vajadzīga PKP? 10
2.2 Kas ir SCM? 11
2.3 PKP galvenā funkcionalitāte 11
2.3.1 Versiju kontrole 12
2.3.2 Konfigurācijas izvēle 14
2.3.3 Laiksakritīga izstrāde 16
2.3.4 Būvējumu pārvaldība 17
2.3.5 Laidiena pārvaldība 17
2.3.6 Darbības apgabala pārvaldība 18
2.3.7 Izmaiņu pārvaldība 18
2.3.8 Integrācija ar citiem rīkiem 20
3 VERSIJU KONTROLE 21
3.1 Termini 21
3.1.1 Repozitorijs 21
3.1.2 Darbības apgabals 22
3.1.3 Revīzija, versija, delta 23
3.1.4 Tags/Iezīme 24
3.1.5 Līdzdalības atbalsts 25
3.1.6 Būvēšana 25
3.1.7 Zari 26
3.1.8 Zaru sapludināšana 27
3.2 Versiju kontroles rīki 27
3.2.1 CVS 28
3.2.2 Subversion 29
4 ESOŠA SITUĀCIJA 32
4.1 IS Card Suite apraksts 32
4.2 Esoša situācija 34
4.3 Patreizējas problēmas 37
4.4 Problēmu risinājums 38
4.4.1 Zarošana uzdevumiem 39
4.4.2 Zarošana komandai 42
4.4.3 Zarošana raksturpazīmēm 43
4.4.4 Zarošana laidieniem 44
4.4.5 Zarošana uzturēšanai 47
4.4.6 Zarošana pēc vajadzības 48
4.4.7 Izvēlēta metode 50
4.4.8 Veco un jaunu zarošanas metožu salīdzinājums 52
4.5 Secinājums 55
5 NOBEIGUMS 57
6 LITERATŪRAS SARAKSTS 58

Termini

Termins latviešu valodā	*Termins angļu valodā*
Atkarības	dependences
Atslēgt	unlock
Būvējums	build
Darbības apgabals	workspace
Ielāps	patch
Ielikšana	check in
Iezīme	label
Izmaiņa	change
Izņemšana	check out
Konfigurācija	configuration
Kompilēšana	compilation
Labojumfails	hotfix
Laidiens	release
Momentuzņēmums	snapshot
Pārbaude	check
Progresīva delta	forward delta
Raksturpazīme	feature
Repozitorijs	repository
Reversīva delta	reverse delta
Revīzija	revision
Sapludināšana	merge
Saskaņot	commit
Slēgt	lock
Tags	tag
Versija	version
Zars	branch

Apzīmējumu saraksts

CMS	Card management system
IP	izmaiņu pieprasījums
IS	informācijas sistēma
KO	konfigurācijas objekts
MPCS	Merchant processing and clearing system
PKP	programmatūras konfigurāciju pārvaldība
RTPS	Real time processing system

1 IEVADS

Paradās jaunie klienti ar savām prasībām, attīstās tehnoloģijas un programmatūra kļūst sarežģītāka. Produkta versiju paralēla izstrāde un uzturēšana aizņem pārāk daudz laika un ir ļoti neērta. Kļūst grūti orientēties liela programmatūras versiju apjomā, labot un pārnest labojumus ir vēl grūtāk un laikietilpīgāk, šo problēmu dēļ uzņēmums nes zaudējumus. Tiek tērēts daudz laika uz vienādām pārbaudēm. Kļūst saprotams, ka ir kaut kas jādara.

Tāpēc šī darba mērķis bija dziļāk izpētīt programmatūras konfigurāciju pārvaldes uzdevumus, identificēt problēmas, saistītas ar to un mēģināt rast risinājumus.

Kursa darbs sastāv no trim daļām.

Pirmā daļā ir uzrakstīts kam ir vajadzīga programmatūras konfigurāciju pārvalde un ir nodefinēts tas jēdziens. Tālāk ir pārskaitītas visas PKP funkcijas, kā arī katra no tām ir aprakstīts kā tas tiek pielietotas.

Otrajā daļā visa uzmanība tiek vērsta versiju kontroles terminoloģijai. Versiju kontrolēs būtība ir jau apskatīta iepriekšējā daļā. Tomēr šajā sadaļā var atrast visus svarīgākus aspektus, kas ir saistīti ar versiju kontroles pārvaldību. Vēl šajā nodaļā tiek aprakstītas populārākie versiju kontroles rīki, kas arī ir izmantoti tālāk izskatāmā sistēmā.

Savukārt trešajā nodaļā tiek aplūkota informācijas sistēma Transmaster, aprakstīta tas izstrādes vēsture, arhitektūra un funkcionalitāte. Tālāk tiek pastāstīts par esošo situāciju šis sistēmas apakšsistēmās versiju kontroles pārvaldībā un ir definētas galvenās problēmas. Ir saprasts, ka galvenā problēma ir zarošanas metodes izvēlē. Kad ir pārāk daudz zaru, turklāt tie veidojas pēc sajauktām zarošanas metodēm, parasto versiju pārvaldības operāciju veikšana ir apgrūtināta. Tāpēc tika izpētītas visas eksistējošas zarošanas metodes un izsecināts kāda no tām varētu atvieglot versiju kontroli CMS3 sistēmā. Beigās ir piedāvāta zarošanas metode, kas arī ir jau ieviesta projekta reālā dzīvē.

2 SCM – PROGRAMMATŪRAS KONFIGURĀCIJU PĀRVALDĪBA

2.1 Kāpēc ir vajadzīga PKP?

Kad programmatūra ir sarežģīta, izstrādes process arī kļūst komplicēts. Kad izstrādē vai uzturēšanā ir iesaistīti daudzi cilvēki ar dažādām lomam ir ļoti svarīgi, lai visiem būtu pieeja pie korektas informācijas. Mazās grupās informāciju var turēt neformāla veidā, tomēr jo lielāka ir grupa, jo paliek sarežģītāk cilvēkiem sazināties un atbilstoši sadarboties. Kad komunicē 2 cilvēki, starp viņiem ir 1 saikne, bet jo vairāk cilvēku, jo lielāks paliek saikņu skaits. Grupai ar n cilvēkiem ir n(n-1)/2 tiešo komunikācijas saikņu. Un paliek pats par sevi saprotams, ka lielākām grupām un ilgtermiņa procesiem, ir nepieciešams efektīvs veids informācijas piekļūšanai.

Programmatūras izstrādes fāze bieži vien notiek ar daudzu izstrādātāju sadarbību. Lai tikt gala ar sarežģītību bija izdomāta programmatūras konfigurāciju pārvalde (PKP). PKP mērķis ir turēt ierakstus par visiem failiem un moduļiem, kas ietilpst programmatūrā un it īpaši visas izmaiņas, kas tika veiktas ar visiem šiem failiem. Papildus tiek atbalstīta arī visa dokumentācija saistīta ar programmatūru.

Programmatūras struktūra izstrādes etapa beigās parasti izskatās savādāk nekā projektēšanas fāzē. Sistēmu izstrādes process ir ļoti sarežģīts un tāpēc PKP vairāk liek uzsvaru uz izstrādes fāzi.

PKP ir programminženierijas disciplīna, kas pārvalda un kontrole projektus un sinhronizētu sadarbību starp dažādiem projekta izstrādātājiem. PKP piedāvā produkta uzturēšanu izstrādes dzīves cikla laikā, metožu un procesu definēšanu, plānu sagatavošanu un rīku izmantošanu, kas palīdz izstrādātājiem un projekta vadītājiem viņu ikdienu darbā izstrādes projektā. PKP ir versts uz programmatūras izstrādes fāzi, jo vairākas PKP aktivitātes ir koncentrētas uz izstrādes fāzi, t.i. kad tiek veidots vai labots programmatūras kods.

2.2 Kas ir SCM?

Eksistē vairākas definīcijas programmatūras konfigurācijas pārvaldībai. Darba autorei piemērotāka šķiet šī definīcija:

Programmatūras konfigurācijas pārvaldība ir process, kas identificē un definē objektus sistēmā, kontrolē šo objektu izmaiņas to visā dzīves ciklā, ieraksta un atskaitās par objektu statusiem un izmaiņu pieprasījumiem un verificē objektu pabeigtību un korektumu [1].

PKP ir divas mērķa grupas ar dažādām vajadzībām: pārvaldība un izstrāde. Vadītājiem vajag kontrolēt un novērtēt produkta izstrādi un it īpaši to laidienus. Esošas PKP sistēmas apmierina vadītāju vajadzības un dod izstrādātājiem lielāku saprašanu par projektu, t.i. paaugstina efektivitāti.

2.3 PKP galvenā funkcionalitāte

Galvenokārt PKP ir paredzēta pašiem izstrādātājiem tāpēc, ka tā apstrādā eksistējošas produkta komponentes, uztur vecas versijas un informāciju par to vēsturi, nodrošina stabilu izstrādes vidi un koordinē vienlaicīgas objektu izmaiņas. Mūsdienas eksistē pietiekami daudz uz PKP mērķiem balstītu rīku un šie rīki parasti nodrošina sekojošo PKP funkcionalitāti [2]:

- versiju kontroli;
- konfigurācijas izvēli;
- laiksakritīga izstrādi;
- būvējumu pārvaldību;
- laidiena pārvaldību;
- darbības apgabala pārvaldību;
- izmaiņu pārvaldību;
- integrāciju ar citiem rīkiem.

2.3.1 Versiju kontrole

Versiju kontrole ir galvenā PKP rīku funkcionalitāte. Parasti domā, ka tas arī ir PKP, kas tomēr nav patiesība.

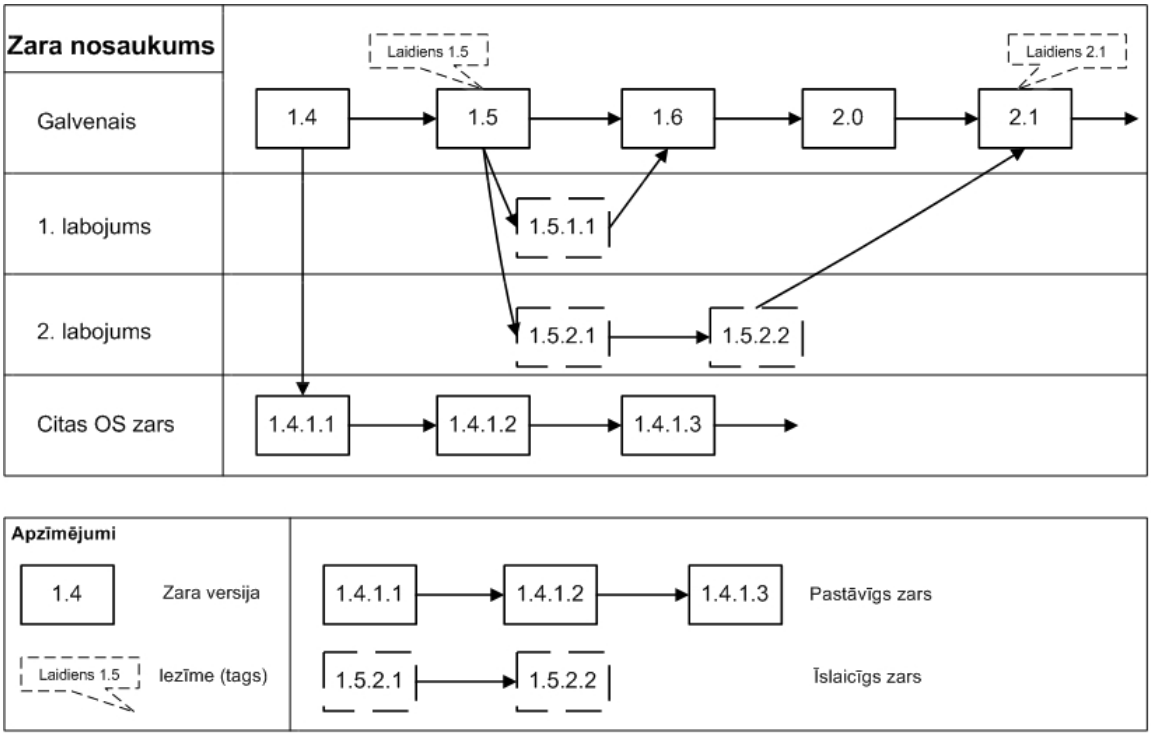
Programmatūras objektu vai dokumentu, kas ir ielikts versiju kontroles rīkā pieņemts saukt par *konfigurācijas objektu* (KO). Parasti KO ir pirmkoda fails, tomēr izpildāmie faili un dokumentācija arī ir KO. PKP rīka galvenā pazīme ir iespēja saglābāt, pārveidot un reģistrēt KO vēsturiskas izmaiņas. Ļoti svarīga versiju īpašība ir tas pastāvība, piemēram, kad versija ir „iesaldēta", tas saturs vairs nevar tikt mainīts, ir jāveido jaunas versijas. Versijas ir gan zariem, gan atsevišķiem objektiem.

Vēl ļoti svarīga PKP rīku īpašība izstrādātājiem ir iespēja sinhronizēt vienlaicīgo darbu vairākiem lietotājiem. Atkarība no rīka šī uzturēšana tiek piedāvāta vairākos veidos, kas ir atkarīgi no sinhronizāciju modeļiem. Pamatmodelis ir „izņemšanas/ielikšanas" modelis, kurā individuālie faili ir glabāti kompaktā formā uz versiju kontroles pamata – *repozitorijā* [3]. Repozitorijs satur tikai vienu galīgu versiju. Atšķirības starp objektu versijām tiek glabāti izmantojot delta algoritmus. Daudzi rīki izmanto lineārus delta algoritmus, kas atgriež rindu atšķirības starp divām versijām. Vēl ir bināri bāzēti delta algoritmi, kas atgriež bināras atšķirības. Galvenā doma delta algoritmu lietošanai ir vietas saglabāšana uz diska.

Šis modelis neļauj mainīt vai lasīt objektus, kamēr tie nav izņemti ārā no repozitorijā. Izņemt ārā nozīme, ka vajadzīga objekta versija ir iekopēta izstrādātāja darbošanās direktorijā un ja ir jāmaina failu, tad tam ir jābūt slēgtam repozitorijā. Tas novērš situāciju, kad vienu un to pašu objektu vienlaicīgi izmaina vairāki cilvēki. Kad objekts ir ielikts atpakaļ, tiek izveidota jauna versija un objekts kļūst atbloķēts. Tieši tāpēc katram objektam ir sava versiju vēsture ar jaunu versiju katrai izņemšanai un ielikšanai.

Objektu versijas var būt organizētas vairākos veidos, piemēram, ja tas tiek organizētas pēc secības, tad to sauc par revīziju, ja tas tiek organizētas kā atsevišķas izstrādes virzieni, tad to sauc par zariem. Zari var tikt apvienoti jaunās versijās, kurām ir divi vai vairāki priekšteči. *1. att.* ir redzams, ka ja notiek kāds apjomīgs labojums vai atsevišķa moduļa izstrāde, ir iespēja izveidot jaunu zaru (attēlā `1. labojums` vai `Citas`

`OS zars`) un labot to atsevišķi, beigās sapludinot to (attēlā zaru `1.5.2.1` sapludina kopā ar `1.6` zara versiju un iegūst jaunu zaru versiju `2.0`).



1. att. **Versiju kontroles piemērs**

Zarus veido dažādo iemeslu dēļ. Galvenie iemesli ir [4]:

- ir gatava jauna laidiena versija – tiek uztaisīts jauns zars, kurā arī turpina programmatūras izstrādi un ir iespēja veikt labojumus klientiem nodotam laidienam;
- paredzēts veikt sarežģītas funkcionalitātes izmaiņas, kas nevar tikt veiktas kopā ar pārējam izmaiņām – tiek izveidots atsevišķs zars, kurā notiek izmaiņu izstrāde, un kad viss ir veiksmīgi pabeigts un notestēts, izmaiņas sapludina kopā ar galveno zaru;
- paredzēts izmantot izstrādājamu programmatūru dažādām operētāj sistēmām, kas būtiski atšķiras – tiek veikti atsevišķi zari un arī izstrāde notiek atsevišķi, izmaiņas netiek sapludinātas kopā.

Dažos versiju kontroles rīkos zars sastāv no daudzām revīzijām[1] un ir papildus zari, kas tika izveidoti uz oriģināla zara bāzes. PKP vadītājiem ir ļoti nozīmīgi izstrādāt stratēģiju zaru veidošanai un apvienošanai.

Versiju rīks var atpazīt revīzijas iekšēji, parasti izmantojot numerācijas metodi vairākas stadijās. Taču lai tas būtu ērtāk lietotājiem, var nosaukt revīziju ar savu tekstu, kas saucas *tags* vai *birka*. Un savukārt rīks var atgriezt versiju, kas ir identificēta ar šo tekstu. Taču ne visiem versiju kontroles rīkiem ir šī funkcionalitāte.

2.3.2 Konfigurācijas izvēle

Failam var būt vairākas versijas un tas, kas patreiz ir jāizmanto, nav vienmēr nepārprotams. Situācija kļūst sarežģītāka apzinoties to, ka IS sastāv no liela failu apjoma un to kombināciju skaits ir pārāk liels. Ikdienas darbā izstrādātājs parasti vēlas faila pēdējo versiju, kas ir mainīts konkrētā zarā. Pārējiem nemainītiem failiem parasti vēlas vecāku versiju, kas strādā stabili un kas ir iekļauta pēdējā programmatūras laidienā. Izstrādei ir jābūt it īpaši elastīgai, lai būtu iespēja mainīt dažādus objektus vienlaicīgi. Piemēram, izmaiņa var prasīt vairāk nekā viena faila modifikācijas. Visās situācijās vajag, lai būtu pieejama atbilstoša failu versiju kopa, kas saucas *konfigurācija.*

Ir jābūt izstrādātam mehānismam kā jāveido konfigurāciju. Šeit ir piedāvāti likumi, kas varētu būt noderīgi:

- pēdēja revīzija izstrādātāja personīgā zarā (failiem, ar kuriem strādā);
- pēdēja revīzija īslaicīgā zarā (failiem, ar kuriem strādā parēji);
- pēdēja revīzija pastāvīgā zarā (failiem, kas atšķiras atkarīgi no produkta);
- izlabota un nosaukta versija (piemēram, pēdējais apakšsistēmas laidiens).

Sistēma, kas ir uzbūvēta izmantojot pēdējo versiju, saucas *daļēji ierobežota* (vai vispārēja) *konfigurācija*, jo pēc iekļautas versijas tas atšķirsies laikā, kad tika ielikta jauna versija. Sistēma, kas bija izveidota savādāk saucas par *ierobežotu konfigurāciju*, kas ir

[1] Revīzija ir programmatūras status uz konkrēto laka momentu, citos rīkos revīzija ir tas pats kā objektu versija. Zīm.1. tas ir apzīmētas ar iezīmēm (piemēram, `Izlaidums 1.5`).

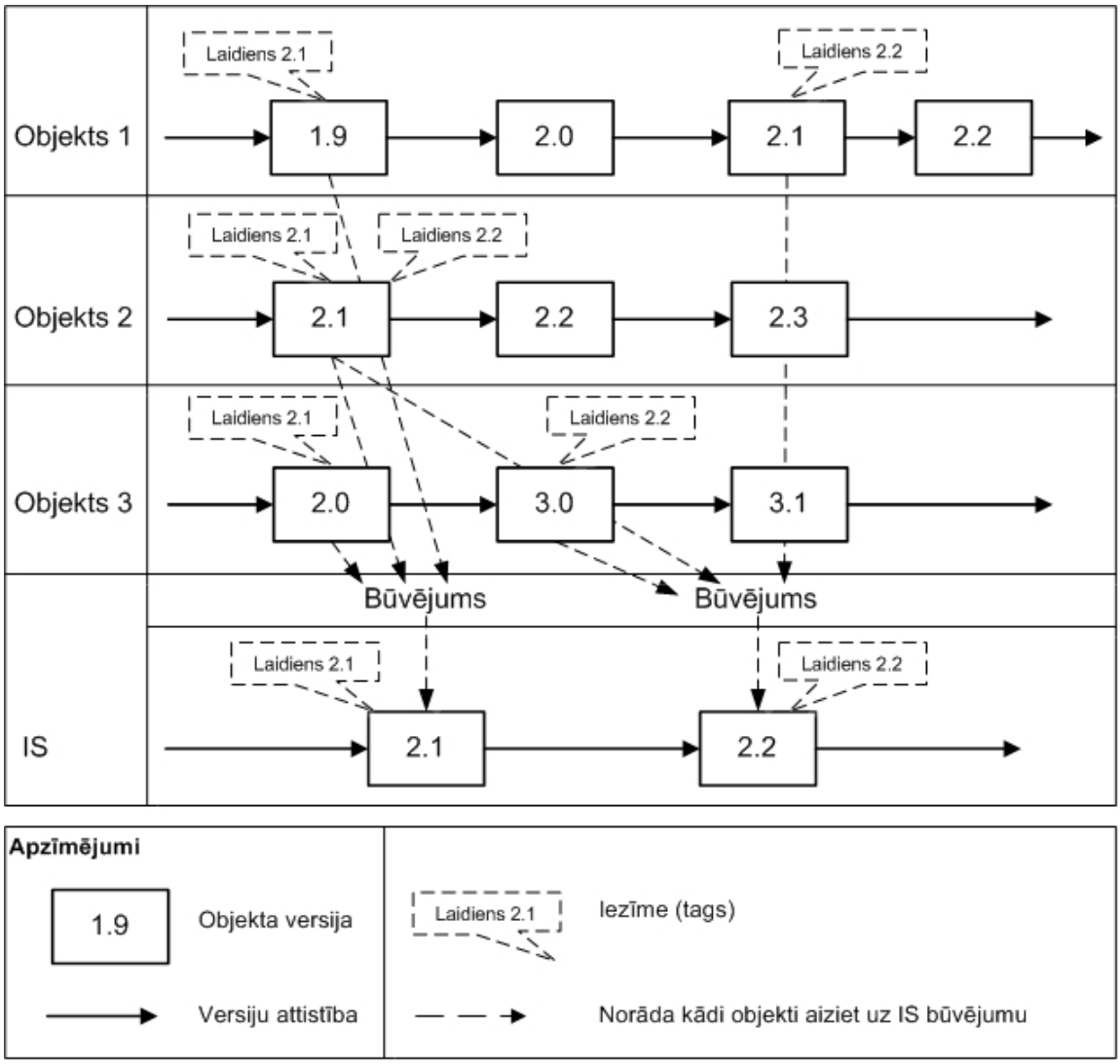
piemērotāka piegādēm, kad versijas visiem iekļautiem failiem ir izlabotas un tāpēc var garantēt, ka vajadzības gadījumā sistēmu varēs atjaunot.

Noteikta ierobežota konfigurācija var veidot bāzlīniju, kas funkcionē kā pamats tālākai izstrādei ar formālu izmaiņu pārvaldību. Tā arī var veidot laidienu, kas ir piegādāts klientam.

Konfigurācijai ir savas versijas un to attīstības vēsture, tāpat kā individuāliem failiem. Lietotāji un klienti redz sistēmas izstrādi lielos soļos, proti, konfigurācijas, laidienus, kas ir izplātītas. Izstrādātāji un vadītāji redz daudz vairāk sistēmas izstrādes posmus, kā arī dalījumus uz zariem un konfigurācijām, katrs ar savu versiju vēsturi.

Versiju nosaukumi (tagi) var tikt izmantoti, lai tiktu galā ar ierobežotas konfigurācijas atlasīšanu, visi faili konfigurācijā ir apzīmēti ar vienādo nosaukumu (piemēram, `Laidiens 2.1`), to var redzēt *2. att*.

Attiecības starp tadam konfigurācijām reti ir atbalstītas ar PKP rīkiem, taču tos jāpārvalda citā veidā, piemēram, ar laidiena dokumentāciju. Protams, var izmantot sistemātiskus nosaukumus, pēc kuriem ir redzamas loģiskas izmaiņas. Piemēram, var izmantot izmaiņu pieprasījuma numurus.



2. att. **Ierobežotu konfigurāciju var veidot apzīmējot ar tagiem visus failus ar vienādam iezīmēm**

2.3.3 Laiksakritīga izstrāde

Viena galvenā priekšrocība PKP sistēmu izmantošanai ir tas, ka izstrādātāji var strādāt vienā projektā laiksakritīgi. Un tas ir priekšrocības vairāku iemeslu dēļ. Piemēram, vairāki izstrādātāji var strādāt laiksakritīgi ar vienu un to pašu failu netraucējot viens otrām, labojot dažādas kļūdas un pēc tam sapludinot savas izmaiņas. Vai arī kāds var labot ar pēdējo laidienu, kamēr cits labo kļūdas iepriekšējā laidienā. Tāpat testētāji var testēt pēdējo stabilu versiju, kamēr izstrādātāji darbojas ar nākamo versiju.

PKP sistēmas nodrošina:

- versiju izvēli, būvējot konfigurācijas dažādam vajadzībām;
- sinhronizācijas modeli laiksakritīgām izmaiņām. Piemēram, slēdzot rediģējamus objektus vai arī ļaut labot objektus jebkurā laika, bet pirms ielikšanas pārbaudīt konfliktus un ja tādi ir, tad sapludināt tos. Eksistē izņemšanas/ielikšanas, kompozīcijas, garo transakciju un izmaiņu kopas modeļi [3].

2.3.4 Būvējumu pārvaldība

Būvējumu pārvaldība dod lietotājam iespēju uzbūvēt produktu vai to daļu (piemēram, komponenti vai bibliotēku). Piemēram, izmantojot būvējuma rīku Make, produkts veidojas automātiski. Tiek savāktas pareizas versijas vajadzīgiem objektiem, tos sakompilē un sasaista pareizā secībā. Make apraksta pirmkoda objektu atkarības būvēšanas laikā un nodrošina, ka atkarīgs pirmkods ir uzbūvēts pareiza secībā.

Tā kā, būvējot lielu sistēmu, tas var aizņemt pārāk daudz laika un nepilnīgs būvēšanas process var iztērēt izstrādātāja laiku, ir ļoti svarīgi atcerēties, ka labāk izmantot tas komponentes, kas pēdējā laikā netika mainītas. It īpaši tas ir aktuāli testēšanai, kad ir jāpārtestē katru mazu izmaiņu. Samazināt laiku var tāds būvēšanas process, kas izmanto nemainītiem objektiem jau iepriekšējā reizē uzbūvētus vienumus.

2.3.5 Laidiena pārvaldība

Identifikāciju un organizāciju visiem piegādājamiem vienumiem (piemēram, izpildāmie objekti, dokumenti un bibliotēkas) iekļautiem IS laidienā sauc par laidiena pārvaldību. Laidiena pārvaldība cieši saistīta ar konfigurācijas izvēli, būvējuma pārvaldību un izmaiņu pārvaldību. Konkrēta izvēlēto objektu konfigurācija tiek izmantota būvējuma procesā un tie objekti, kas ir izveidoti būvējuma procesa laikā ietilpst arī laidiena procesā.

Laidiena pārvaldībai ir divi uzdevumi. Pirmkārt, jāsagatavo piegādājamie vienumi un visa dokumentācija lietotājiem. Tajā ietilpst informācija, ko satur piegādājamie vienumi, kādas ir izveidotas jaunas funkcijas, izmaiņas implementētas no iepriekšēja

laidiena un prasības izpildlaika videi. Otrkārt, ir piedāvāta informācija iekšējai izmantošanai – testēšanai, uzturēšanai vai tālākai izstrādei. Piemēram, ir iespējams uzzināt kurš lietotājs ir izstrādājis kuru versiju kādai komponentei vai kas būs ietekmēts ar nākamam objekta izmaiņām.

2.3.6 Darbības apgabala pārvaldība

Izstrādātāji var strādāt, risinot dažādas problēmas savos darbības apgabalos izolēti no pārējiem un tajā pat laikā viss vienalga tiek kontrolēts ar PKP. Objektu versijas ir izņemtas un īslaicīgi saglabātas darbības apgabalā, ka arī ir kartēšana versiju objektiem repozitorijā un lietotāju failiem un direktorijām darbības apgabalā.

Lai strādāt ar produktu sakumā no repozitorija tiek izņemti visi objekti, nevis tikai tas, kas ir vajadzīgs uz doto brīdi, daži no tiem var būt tikai lasīšana režīmā. Vēlāk pirms konkrēta objekta labošanas ir iespēja dabūt pēdējo versiju tikai nepieciešamiem objektiem vai arī visam produktam kopa.

Kad daži izstrādātāji strādā laiksakritīgi katrs savā darbības apgabalā, ir vajadzīga kontrole starp vienādo objektu dažādām kopijām. Vairāki PKP rīki atbalsta konfliktu atrāšanas funkcionalitāti vienādiem objektiem, kas tika laboti un ielikti repozitorijā laiksakritīgi.

Daži PKP rīki atbalsta kooperatīvo versiju veidošanu, t.i. ir iespēja veidot versijas lokāli savā darbības apgabalā. Tomēr kad objekts ir ielikts atpakaļ repozitorijā, tad tiek ielikta tikai pēdēja versija un pārējas lokālas tiek nodzēstas.

2.3.7 Izmaiņu pārvaldība

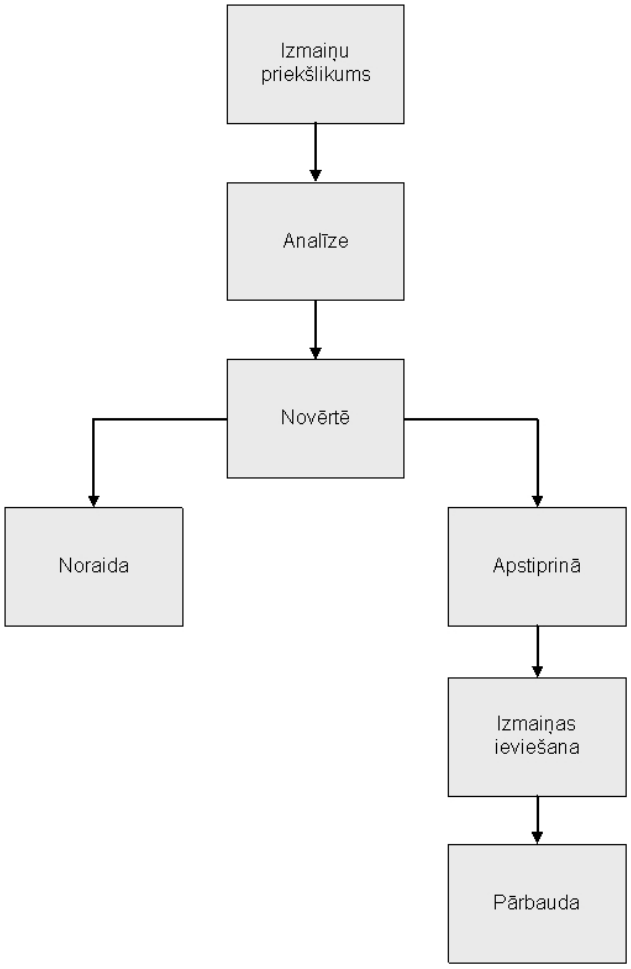
Izmaiņu pārvaldībai ir ieraksts ar visam produkta izmaiņām. Cēlonis izmaiņai var būt atrastas IS kļūdas labošana, izmaiņu pieprasījums vai komponentes uzlabošana. Parasti tas ir atsevišķi rīki, kas nav iekļauti PKP rīkos, piemēram, JIRA, Rational RequisitePro, Lotus QuickPlace utt.

Izmaiņu pārvaldībai ir divi galveni mērķi. Pirmais – sniegt atbalstu izmaiņu veikšanā. Tas iekļauj sevī izmaiņas identifikāciju, analīzi, prioritātes noteikšanu,

plānošanu, lēmumus atteikt vai apstiprināt izmaiņu, komentārus u.c. Otrais – objektu trasējamība, piemēram, ir iespējams atlasīt visas aktīvas un ieviestas izmaiņas.

Izmaiņu pārvaldības process

Sistēmas izmaiņas sauc par *izmaiņu pieprasījumiem* (IP). Kad izmaiņa tika iniciēta, tiek izveidots aktīvs IP pieprasījums, lai izsekot to izmaiņas līdz brīdim, kad pieprasījums ir atrisināts un aizvērts. *3. att.* ir redzams IP pieprasījuma dzīves cikls.

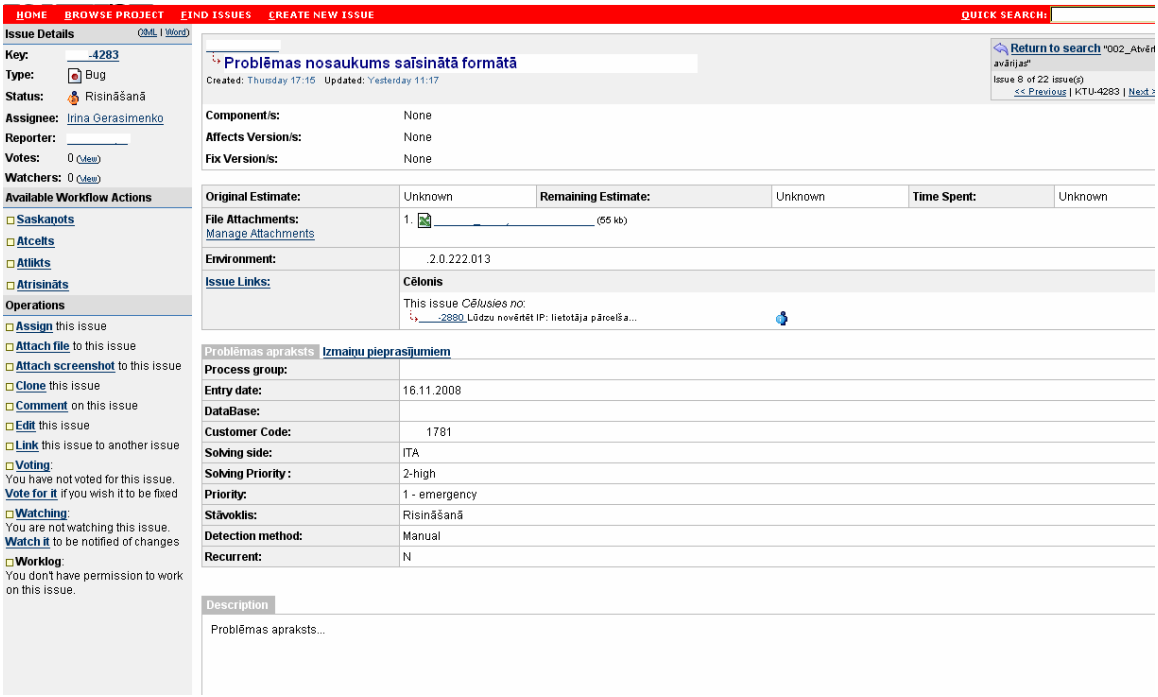


3. att. **IP procesa vienkāršots dzīves cikls**

Trasējamība

Katram IP ir jāredz kādas versijas modificētiem failiem tika izveidotas izpildot pieprasījumu. Arī ir jābūt iespējai dabūt informāciju, kāpēc ir izveidota konkrēta versija konkrētam failam.

Izmaiņu pārvaldība arī dod iespēju redzēt kādas izmaiņas tika iekļautas laidienā, kāds ir pieprasījuma stāvoklis, cik daudz laika ir patērēts uz to, ka arī citu svarīgu informāciju. Piemēram, izmaiņu pārvaldības rīkā JIRA (sk. *4. att.*) var atrast visu nepieciešamo informāciju par konkrētu pieprasījumu.



4. att. **JIRA IP pieprasījuma izskats**

2.3.8 Integrācija ar citiem rīkiem

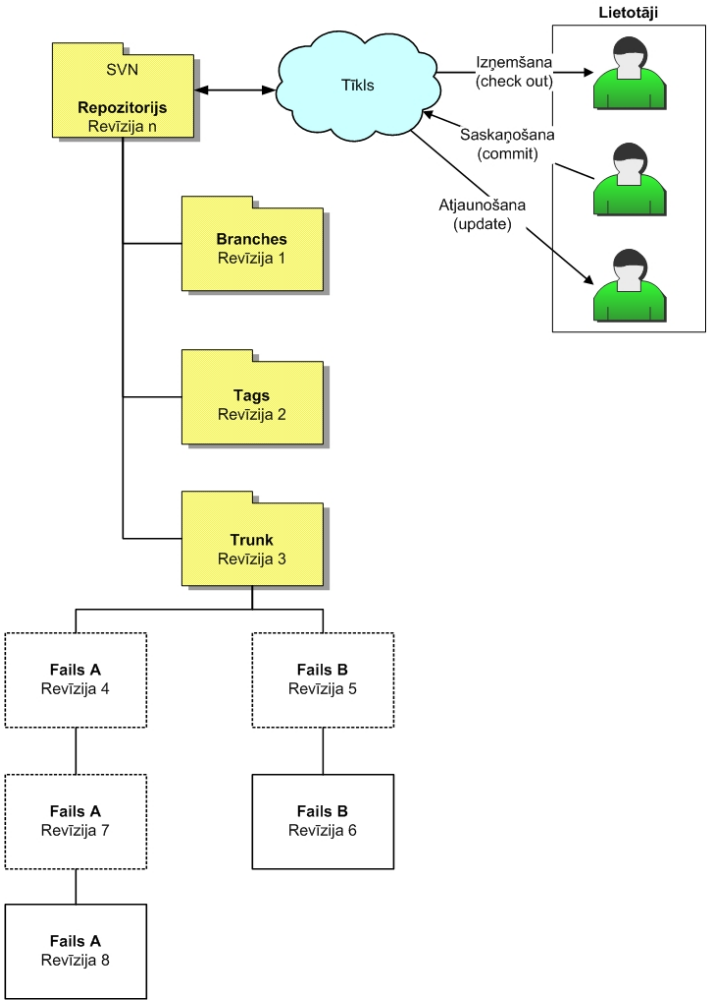
Jaunāki PKP rīki var sadarboties kopā ar citiem izstrādes procesa rīkiem. Bieži dažas funkcijas ir iebūvētas integrētas izstrādes vīdēs (IDE). Piemēram, tādas operācijas kā izņemšana un ielikšana repozitorijā, produkta vai komponentes būvēšana.

3 VERSIJU KONTROLE

3.1 Termini

3.1.1 Repozitorijs

Repozitorijs ir vieta, kur tiek glabāti visi pirmkoda faili, attēli, dokumentācija, konfigurācijas faili, izpildāmie faili, bibliotēkas un visi citi programmatūras objekti, kas ir saistīti ar produktu [6]. Tas viss ir izveidots loģisko failu un mapes hierarhijā. Taču versiju kontroles rīkos repozitorijam ir jāglabā nevis tikai produkta objektus, bet arī to visas versijas. Liels akcents tiek likts uz drošību, jo repozitorijam veiksmīgam būvējumam ir izveidots viens piekļuves punkts. Repozitorija struktūra ir attēlota *5. att*.



5. att. **SVN repozitorija struktūra**

Repozitorijs ir ērts ar to, ka:

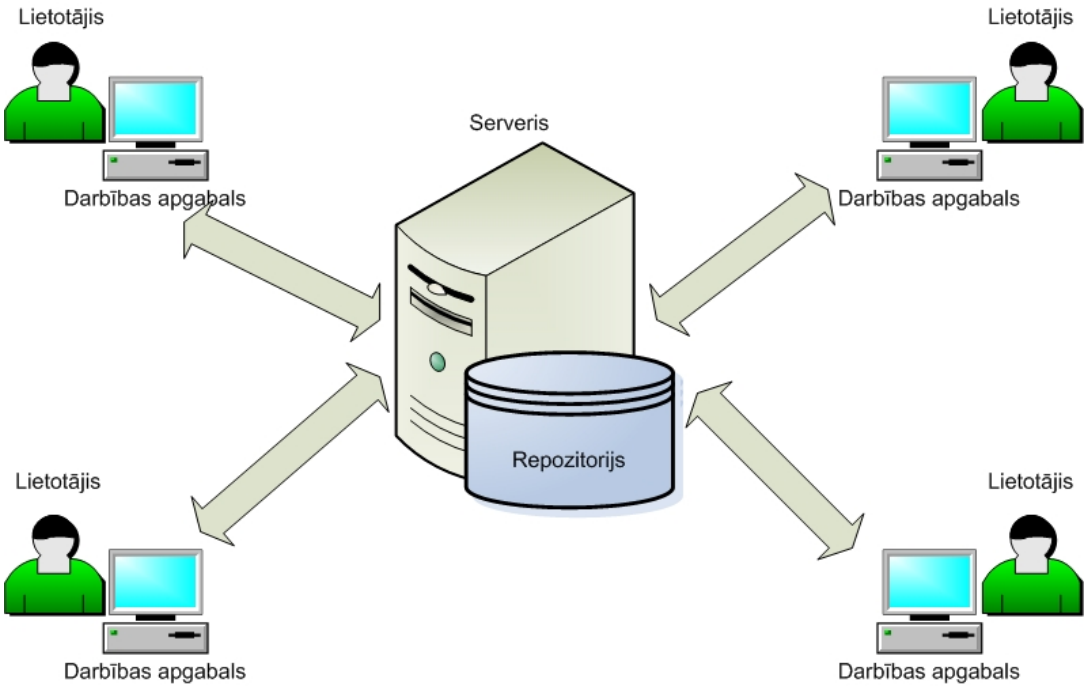
- to ir viegli kontrolēt, dublēt un atjaunot;
- ir stingra piekļuves kontrole katram repozitorija objektam;
- ir iespēja saglābāt logus lietotāja aktivitātei.

Dažādi PKP rīki glāba repozitoriju, izmantojot failus serveru failu sistēmā vai datu bāzē. Repozitorijam ir jāglabā resursu metadatus vai arī datus par resursiem līdzīgi failu vēsturei. Metadati aug kopa ar resursiem, ar to palielinot repozitorija izmērus.

Daudziem PKP rīkiem ir attālināta lietotāja atbalsts. Attālinātie lietotāji var piekļūt repozitorijā caur Internetu.

3.1.2 Darbības apgabals

Darbības apgabals, atšķirībā no repozitorija, atrodas katram izstrādātājam viņu personīgā darbstacijā. Individuālo darbības apgabalu mērķis ir nodrošināt lietotājam vidi, kur viņi var strādāt atsevišķi no repozitorija un izolēti viens no otra. Tāpēc izstrādātāji var strādāt savā darbības apgabala kā grib, netraucējot viens otram. Darbības apgabala ideju var redzēt *6. att*.



6. att. **Darbības apgabali**

Darbības apgabals līdzīgi reporiztorijam var kļūt ļoti liels un bremzēt darbu. Lai tā nenotiek ir biežāk jāsaskaņo izmaiņas ar repozitoriju.

3.1.3 Revīzija, versija, delta

Bieži vien revīzija tiek saprasta kā jauna versija un versija kā izmaiņa. Tāpēc, lai nejaukt šos terminus, ieviesīsim to jēdzienus.

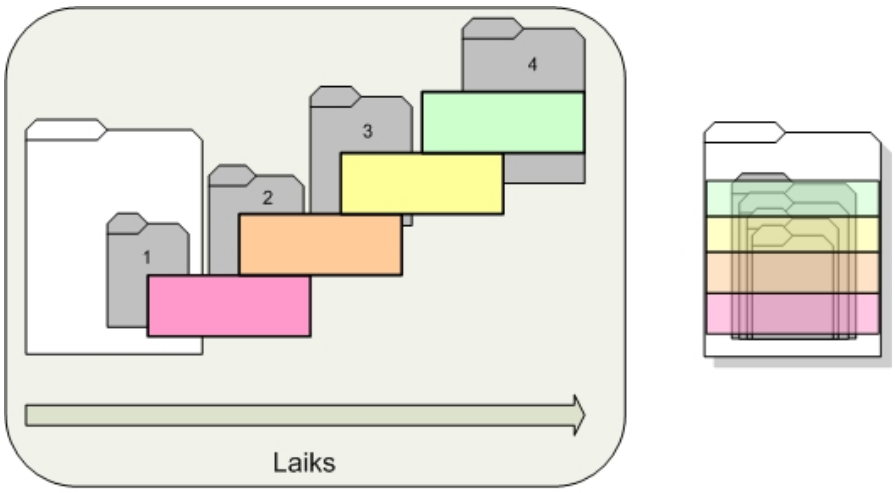
Versijas parāda dažādus objekta vai produkta statusus to dzīves cikla laikā. Tas ir, katru reizi, kad kaut kas ir izmainīts un ielikts repozitorijā, tad tiek izveidota jauna versija. Un *revīzija* ir versiju augoša numuru secība, t.i. 1,2,3, vai 1.1, 1.2, 1.3 utt.

Ir divi revīziju glabāšanas veidi. Viens ir revīziju numurēšana *katram-failam* (per-file), kur katram objektam ir savs revīzijas numurs. Otrs veids izmanto *globālo* revīziju numurēšanu. Pēc katras objekta izmaiņas revīzija tiek piestādīta visai mapei.

Repozitorijam ir iespēja atjaunot objektus no jebkuras revīzijas.

Savukārt *delta* ir atšķirības starp divām versijām. Deltas izmanto izmaiņu identifikācijai, ka arī vietas samazināšanai, glabājot repozitorijā nevis katru reizi izmainīto objektu, bet tikai to deltas.

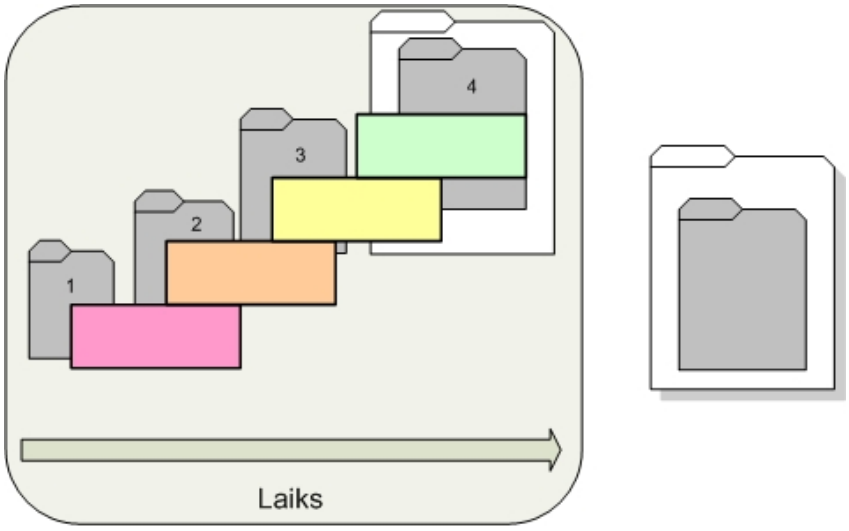
Ir divas deltas glabāšanas metodes. *Progresīvas deltas* metode strādā sekojošā veidā, tiek saglabāta pirmā objekta versija un pēc tam visas izmaiņas tiek saglabātās kā deltas (sk. *7. att.*).



7. att. **Progresīvas deltas metode**

Ar šo metodi viegli un ātri var dabūt vecas versijas, jo tiek ņemta pirmā objektu versija un pieliktas klāt izmaiņas līdz mūs interesējošai revīzijai. Jaunās versijas ar šo metodi dabūt ir grūtāk, t.i. tas notiek lēnāk.

Eksistē arī pretēja metode, tā saucama *reversīva deltas* metode. Šī metode strādā pavisam pretēji, t.i. vienmēr tiek saglabāta jaunā objekta versija, bet iepriekšēja aizvietota ar to deltu (sk. *8. att.*). Tāpēc izmantojot šo metodi ātri var iegūt pēdējo versiju, bet ne vecas.



8. att. **Reversīvas deltas metode**

Tomēr, lai paaugstināt veiktspēju versiju kontroles rīkos tiek pielietotas abas metodes. Tas ir pilnā objekta versija tiek glabāta pēc kāda secīgu deltu daudzuma. Un labojot kādu versiju, tiek pieliktas klāt tuvāk atrodošas deltas.

3.1.4 Tags/Iezīme

Tagu (dažādos avotos tiek saukts par iezīmi) pielietošana ir ļoti svarīga repozitoriju vēstures pārvaldībai. Ar tagu var asociēt kaut kādas izmaiņas saprotamā veidā. Ir iespēja veidot tagus uz esoša repozitoriju stāvokļa vai uz pagātnes stāvokļa.

Tagi ļauj mums veidot reporitorija momentuzņēmumu. Tāpēc dažos projektos, kur tiek izmantoti versiju kontroles rīki ar šādu funkcionalitāti, tagi tiek uzlikti tikai pirms kādām lielām izmaiņām, būvējumiem vai laidieniem. Taču ir rekomendācija likt tagus biežāk, lai varētu atgriezties uz jebkuru objekta iepriekšēju stāvoklī jebkurā brīdi.

3.1.5 Līdzdalības atbalsts

Līdzdalības atbalsts ir ļoti svarīga PKP raksturpazīme, jo programmatūras izstrāde parasti tiek veikta ar cilvēku komandu. Un kad katrs strādā savā darbības apgabalā, ir jābūt sadarbībai, lai mazinātu konfliktus, piemēram, kad ir diviem cilvēkiem ir vajadzīgs labošanai viens objekts.

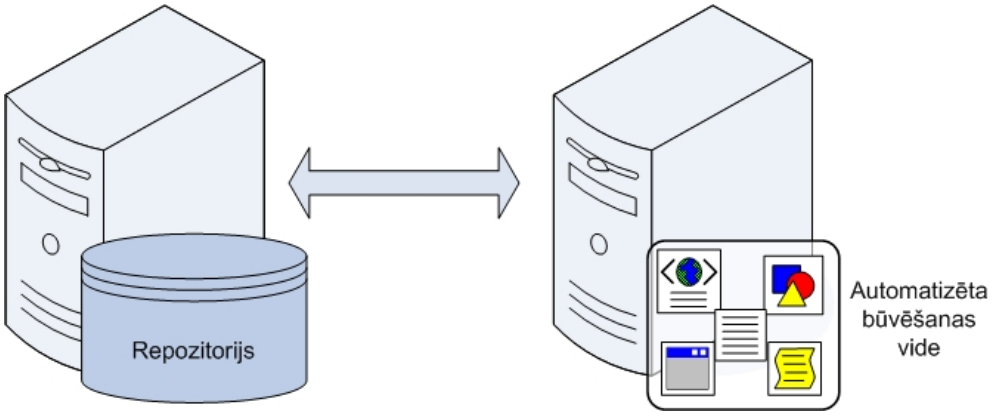
Versiju kontroles sistēmai ir jābūt spējīgai nepieļaut tādas situācijas, kad divi cilvēki labo vienu un to pašu objektu, t.i. neļaut vienam pārrakstīt otra izmaiņas. Šī ir obligāta raksturpazīme, jo bez tas komandai būtu ļoti grūti, ja nu pat neiespējami paveikt kaut kādas izmaiņas.

Lai būtu iespējams līdzdalības atbalsts, versiju kontroles sistēmas izmanto obligātas komandas [7]:

- Izņemšana (Check out);
- Labošana (Modify/Edit);
- Ielikšana (Check in);
- Sapludināšana (Merge);
- Saskaņošana (Commit/Submit).

3.1.6 Būvēšana

Būvēšana ir process, kas transformē pirmkodu bināros izpildāmos failos, apvienojos konfigurācijas objektus, izmantojot būvēšanas skriptus, make failus, kompilatorus un citus rīkus, un veidojot programmatūras versiju. Būvēšana var notikt sakot ar dažiem failiem un beidzot ar tūkstošiem, kas ļoti palēnina procesu. Taču izmantojot tagus un specifiskas konfigurācijas ir iespēja uzlabot un paātrināt būvēšanas procesu, izmantojot objektus, kas netika mainīti pēdējā reizē, kā arī paradās iespēja automatizēt būvēšanas procesu (sk. *9. att.*).



9. att. **Automatizēts būvēšanas process**

Automatizējot būvēšanas procesu ir iespēja dabūt ieguvumus un paaugstināt produktivitāti, jo iespējamas problēmas ir izsekojamas agrāk un regulāri. Šo var panākt izmantojot biežu integritātes būvējumu kopa ar regresu testēšanu. Tas notiek izmantojot automatizētu būvēšanu un testa vides.

Laidiena pārvaldība ir cieši saistīta ar būvēšanas pārvaldību, jo laidiens ir tas, kas ir iegūts pēc produkta būvēšanas. Laidiena pārvaldība ir process, kas palaiž uzbūvētu un notestētu sistēmu produkcijas vidē, ar iespēju padarīt programmatūras lietotni pieejamu gala lietotājam.

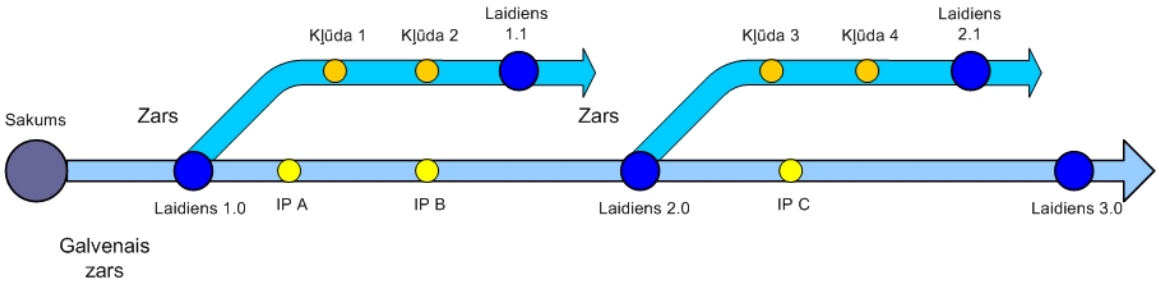
Pēc pirmā laidiena, t.i. programmatūras pirmās versijas, tālākas rūpes sadalās uz vismaz divām daļām – turpinās produkta izstrāde līdz nākamajam laidienam un ir arī jāuztur palaisto versiju. Šiem uzdevumiem ir jāturpinās paralēli un versijas kontroles rīkam ir jābūt tādai iespējai strādāt paralēli.

3.1.7 Zari

Programmatūras izstrādes procesā sakumā tiek izveidota programmēšanas līnija, kurā strādā visi izstrādātāji, t.i. pieliek klāt funkcionalitāti, labo kļūdas, testē. Bet bieži vien lieliem projektiem šī līnija ir pārāk gara, un tad var gadīties, ka pieliekot kaut ko klāt, ir sabojātas jau sen uztaisītas un notestētas lietas. Vai arī dažreiz var redzēt tendenci, ka tiek veidoti kļūdaini būvējumi. Lai izvairīties no šīm un citam vienas programmēšanas līnijas problēmām, tiek izmantoti zari, lai sadalīt programmēšanas līniju uz citām programmēšanas līnijām, veidojot dažas izstrādes ceļus. Katru atsevišķu programmēšanas

līniju sauc par zaru. Parasti eksistē galvenais izstrādes zars un parēji izveidoti zari (piemēram, uzturēšanai vai paralēlai izstrādei).

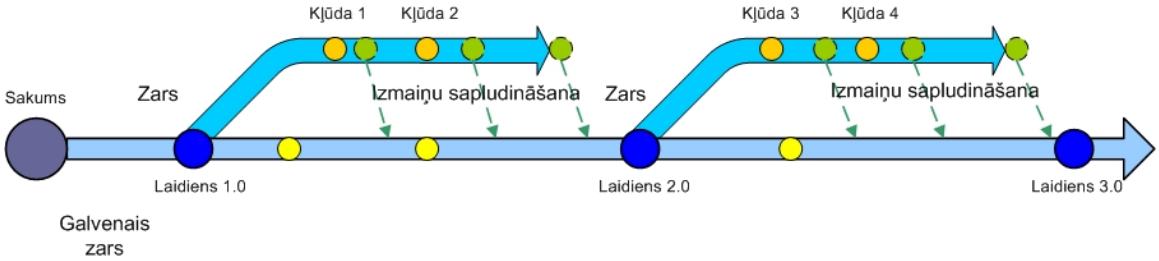
10. att. ir parādīts piemērs kā zarošanas ļauj turpināt programmatūras izstrādi galvenā zarā, kamēr parēji zari tiek uzturēti.



10. att. **Zarošanas piemērs**

3.1.8 Zaru sapludināšana

Sapludināt zarus nozīme pārnest izmaiņas no viena zarā uz otru. Un pēc sapludināšanas vienalga paliek divi zari un viens vai abi tiek arī atjaunināti ar izmaiņām, kas ir veiktas citos zaros. *11. att.* ir attēlots zaru sapludināšanas princips, var redzēt, ka izmaiņas tiek pārnestas no uzturēšanas zara uz galveno.



11. att. **Zaru izmaiņu sapludināšanas piemērs**

Paralēlas izstrādes izmantošana atrisina daudzas problēmas. Gadījumā, kas ir attēlots *11. att.*, bija taisīti uzturēšanas zari pēc katra laidiena. Zarošanas ir labs veids kā organizēt vairāku programmēšanas līniju izstrādi vienlaicīgi. Tas ļauj labāk kontrolēt programmatūras konfigurāciju un pievieno iespēju cieši kontrolēt produkta laidienus.

3.2 Versiju kontroles rīki

Pirmā plaši lietotā versiju kontroles sistēma bija Source Code Control System (SCCS). To izveidoja Marc Rochkind AT&T Bell Labs pētīšanas centrā [8]. Šī sistēma

tika lietota no 1970 gadiem. Nākama bija realizēta SCCS uzlabota versija Revision Control System (RCS). Šī sistēma tika ieviesta 1982 gadā. Mūsdienās ir pieejamas daudzas versiju kontroles sistēmas un šis sistēmas aizvieto citi, vairāk spējīgāki produkti, tādi kā Visual SourceSafe, BitKeeper, Perforce, Vault un atklātā koda sistēmas Arch, CVS un Subversion. Starp atklātā koda programmētājiem CVS kļuva par galveno standartu.

Tā kā šajā darbā tiks izskatītas versiju kontroles problēmas produktam, kas tiek atbalstīts ar CVS un Subversion versiju kontroles rīkiem, tāpēc tie tiks izskatīti detalizētāk.

3.2.1 CVS

CVS ir atklāta koda versiju kontroles sistēma. CVS ir noderīgs gan individuālam izstrādātājam, gan lielai izstrādes komandai, sadalītai uz grupām [9]:

- Klient-servera pieejas metode dod pieeju izstrādātājiem caur Internetu no jebkuras ģeogrāfiskas vietas. Versiju vēsture tiek glabāta centrālā serverī un katrai klientu darbstacijai ir sava kopija ar visiem failiem, ar kuriem viņi var strādāt. Tāpēc, lai veiktu CVS operācija (saskaņošana, atjaunošana u.c.) ir vajadzīgs pieslēgums Internetam, taču lokālām darbībām (esošas versija labošana) ar failiem tas nav nepieciešams.
- CVS ļauj uzturēt vēsturi direktorijas kokam, nevis tikai atsevišķiem failiem.
- CVS atbalsta zarošanu un dod iespēju strādāt ar dažiem zariem paralēli, ka arī nodrošina sapludināšanas mehānismu.
- Ir iespēja uzlikt tagu uz direktoriju koka jebkurā laikā, vēlāk atjaunot failus uz to stāvokli un radīt atšķirības starp tagiem vai revīzijām ar standartu `diff` formātu.
- CVS ir universālas izņemšanas, kas ļauj strādāt ar vienādiem failiem vairākiem lietotājiem vienlaikus.
- CVS ļauj projektu vadītājiem kontrolēt izstrādes procesu:

- `'commitinfo'` konfigurācijas fails atbild par izmaiņu pārbaudi. Piemēram, kad izstrādātājs mēģina saskaņot savas izmaiņas, CVS ar šo skriptu pārbauda vai šis izmaiņas ir jāakceptē.
- `'loginfo'` konfigurācijas fails palaiž skriptu pēc izstrādātāja saskaņotām izmaiņām, piemēram, var sūtīt informāciju citiem izstrādātājiem par ieliktām izmaiņām.
- `'rcsinfo'` konfigurācijas failā nodefinē sagatavi log paziņojumiem saskaņošanas brīdī.
- `'editinfo'` konfigurācijas skriptā var definēt izstrādātāja log ziņojumu pārbaudi. Piemēram, vai tika ierakstīts eksistējošas problēmas kods.

– klienta rīki ir pieejami uz dažādām populārām platformām.

3.2.2 Subversion

Subversion (SVN) atklāta koda versiju kontroles sistēma, kas bija domāta kā CVS aizvietotājs. Bija izlaista 2000. gadā. Principā SVN funkcionalitāte ir līdzīga CVS, taču ir izlabotas dažas CVS problēmas un uzlabota funkcionalitāte **[10]** [11]:

– *Atomu saskaņošana (atomic commits)*

Tad, kad notiek izmaiņu saskaņošana repozitorijā, datu bāžu un versiju kontroles sistēmu pamatprincips ir vai pieņemt izmaiņas, vai arī noraidīt. CVS sistēma to negarantē, jo atomitāte ir garantēta tikai atsevišķi katram failam. Piemēram, izstrādātājs mēģina saskaņot 10 failus un tajā pašā brīdī kāds cits arī sācis saskaņot citus failus, kuru starpā ir kāds pirmās personas fails. Un beigās repozitorijā nonāk 5 faili un uz 6 tiek parādīta kļūda par faila satura konfliktu un nekas tālāk nenotiek, bet arī 5 jau ieliktiem failiem nevar atcelt saskaņošanas procesu.

SVN šo funkcionalitāti nodrošina, jo saskaņošana notiek transakciju veidā un ja kaut kas nenotiek, tad visa transakcija ir atcelta.

– *Failu pārsaukšana un versiju direktorijas*

CVS sistēmai nav tādas iespējas pārsaukt failus un saglabāt versiju vēsturi. Pārsaukt `file1` uz `file2` CVS sistēmā notiek ar sekojošām komandām:

```
1. $ mv file1 file2
2. $ cvs remove file1
3. $ cvs add file2
4. $ cvs commit
```

Taču šādi izveidojas jauns fails bez iepriekšēja faila vēstures.

Šī pati darbība SVN notiek ar komandu `move` un visa vēsture paliek.

```
$ svn move file1 file2
```

Turklāt SVN arī parādījās direktoriju versiju veidošanas iespēja, t.i. revīzijas tiek liktas nevis tikai atsevišķiem failiem, bet arī direktorijam. Tas dod iespēju kopēt ne tikai failus, bet arī direktorijas un to un iekšējo failu vēsture paliek.

– *Zarošana un tagi*

Kad liek tagu CVS repozitorijā, tags tiek pielikts klāt katram failam direktorijā, kurai tiek likts tags. Kad veido jaunu zaru un liek pa virsu tagu, tad pēc tā paša principā tags tiek ielikts visiem failiem. Lieliem repozitoriem šī ir pārāk dārgā operācija.

SVN šo problēmu atrisināja, izslēdzot atšķirības zaru un tagu definīcijām, t.i. tas principā ir viens un tas pats. Tāpēc izmaksas zaru un tagu veidošanai ir līdzīgas. Fiziski nav atšķirību starp tagu un zaru – tags ir produkta kopija noteikta laika momentā, bet nekas vairs netiek mainīts, un zars ir produkta kopija noteikta laika momentā, kura var veikt izmaiņas, neskarot galveno koku.

Papildus, CVS, izņemot failu no zara, pieprasa laiku proporcionālu revīziju skaitam, sakot ar to revīziju, kas norāda uz galveno zaru, papildus revīziju skaits, kas bija veidotas galvenā zarā pēc jauna zara izveides. RCS faili glabā pēdējo revīziju no galvenā zara pilno failu (reversīva deltas metode) un katru reizi, kad ir vajadzīga cita faila versija, tā tiek salikta no iepriekšējo versiju deltām.

Tāpēc `diff` operācijas, zaru pārslēgšana un izņemšana ir apmēram proporcionāla revīziju skaitam, kas tika izveidotas konkrētajā zarā.

– *Klient-servera komunikācija*

Kad fails tiek mainīts lokāli izmantojot CVS un ir nepieciešamība uzzināt atšķirības starp izmainīto kopiju un to, kas ir repozitorijā, tad šis fails tiek nosūtīts uz serveri, tur notiek `diff` operācija un rezultāts tiek atgriezts atpakaļ klientam. Tāpat notiek saskaņošanas, atjaunošanas un sapludināšanas operācijas. Tas nozīme, ka operācijas cena ir proporcionāla lokāli izmainīto failu izmēriem, kas ir vairāk, nekā vienkārši izmaiņu izmērs.

Kad notiek SVN repozitorija atjaunošana, pēdējo revīziju kopija paradās lokāli. Tāpēc, taisot saskaņošanu lokāli izmainītam failam, serverim tiek sūtītas tikai atšķirības starp lokālo pēdējo faila versiju un failu ar izmaiņām, kas noteikti ir izdevīgāk, nekā sūtīt veselo failu.

Papildus pateicoties lokālai repozitorija kopijai ir nepārprotama atšķirība starp servera un lokālam operācijām. Piemēram, atrast kādi faili tika lokāli izmainīti un tas izmaiņas var bez jebkādas piekļuves serverim. Īstenībā SVN operācija cena ir proporcionāla veiktām izmaiņām, nevis faila vai repozitorija izmēriem kā CVS.

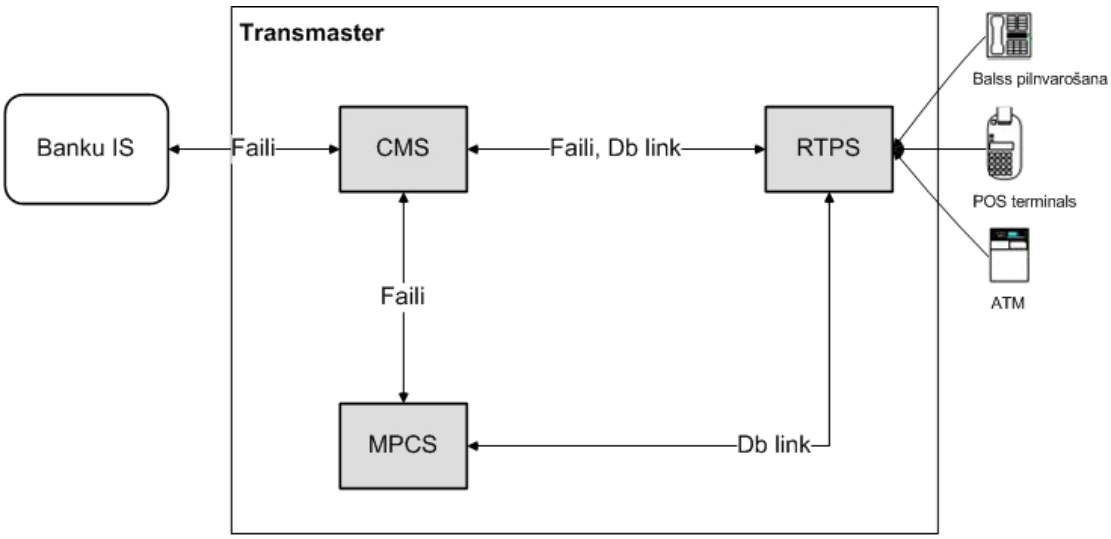
4 ESOŠA SITUĀCIJA

4.1 IS Card Suite apraksts

Versiju kontroles problēmas tiks izskatītas uz Informācijas sistēmas Card Suite (tālāk tekstā Transmaster) piemēra. Šī sistēma ir paredzēta bankas karšu apstrādei un tā ietver sevi sekojošu funkcionalitāti:

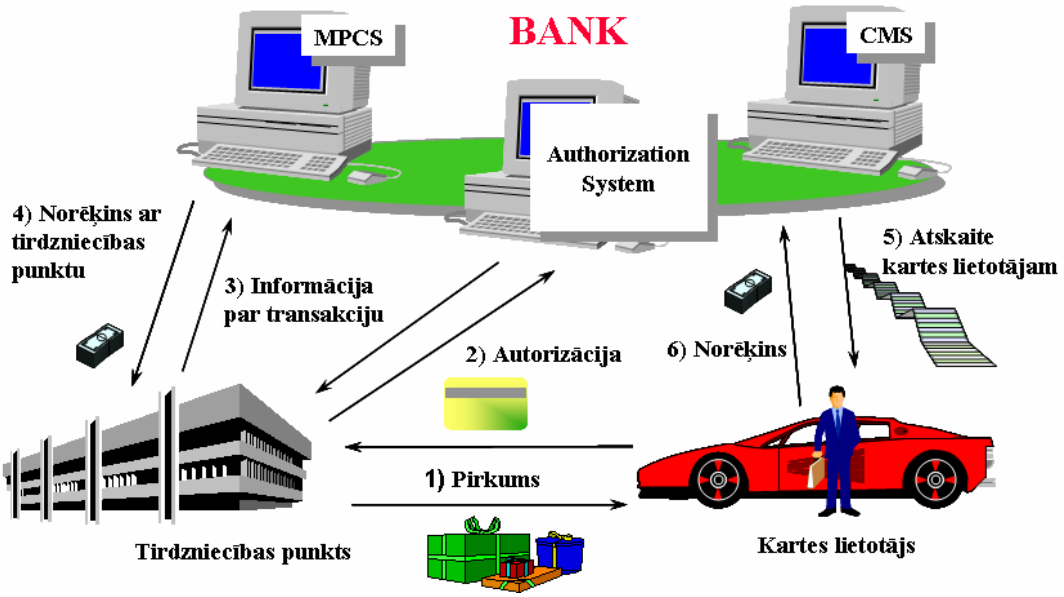
- karšu vadību;
- PIN ģenerēšanu un personalizāciju;
- izdevējautorizāciju;
- kompleksas karšu un maksu kombinācijas;
- elastīgu kontu apstrādi;
- interfeisu ar bankas pamatoperāciju sistēmām;
- 3D Secure shēmas uzturēšana interneta transakcijām.

Patreiz ir izlaistas 3 Transmaster versijas un tiek izstrādāta ceturtā. Šajā darbā tiks izskatītas otrā un trešā versijas. Abas divas sastāv no trim pamat apakšsistēmām (sk. *12. att.*). Un tas ir Card management system (CMS), Merchant processing and clearing system (MPCS) un Real-time processing system (RTPS). Starp šīm un ārējam banku sistēmām nepārtraukti notiek speciālo failu apmaiņa, ka arī tas ir saistītas kopa ar Db link. RTPS nepārtraukti saņem datus no POS termināliem, bankomātiem (ATM) un balss pilnvarošanām. MPCS saņemot datus caur Db link no RTPS, apstrādā tos un pārsūta CMS, kas savukārt nodod tos tālāk citai banku IS.



12. att. **Transmaster**

RTPS strādā tieši ar klienta veiktām naudas darbībām izmantojot kredītkartes vai debetkartes. Piemēram, klients maksājot ar karti veikalā, it kā uzreiz samaksa par pirkumu, tomēr tas nav īsti pareizi (sk. *13. att.*). Šajā brīdī RTPS saņem pieprasījumu par naudas pārskaitījumu, veic pārbaudi cik banka vai var izdot klientam naudas, izveido transakciju un nosūta informāciju MPCS sistēmai, kas jau veic naudas pārskaitījumu tirdzniecības punktam no pieejamiem bankas līdzekļiem. Uzreiz arī tiek nosūtīta informācija CMS sistēmai, kas jau noņem naudu tieši no klienta rēķina un pārskaita to atpakaļ bankai.



13. att. **Transmaster darbības piemērs**

RTPS ir realizēta ar C un Tuxedo valodām, MPCS – Oracle, CMS – Oracle, XSLT, Java. Katra no šim sistēmām ir ļoti specifiskā, taču šajā darbā tiks izskatīta tikai viena sistēma – CMS.

4.2 Esoša situācija

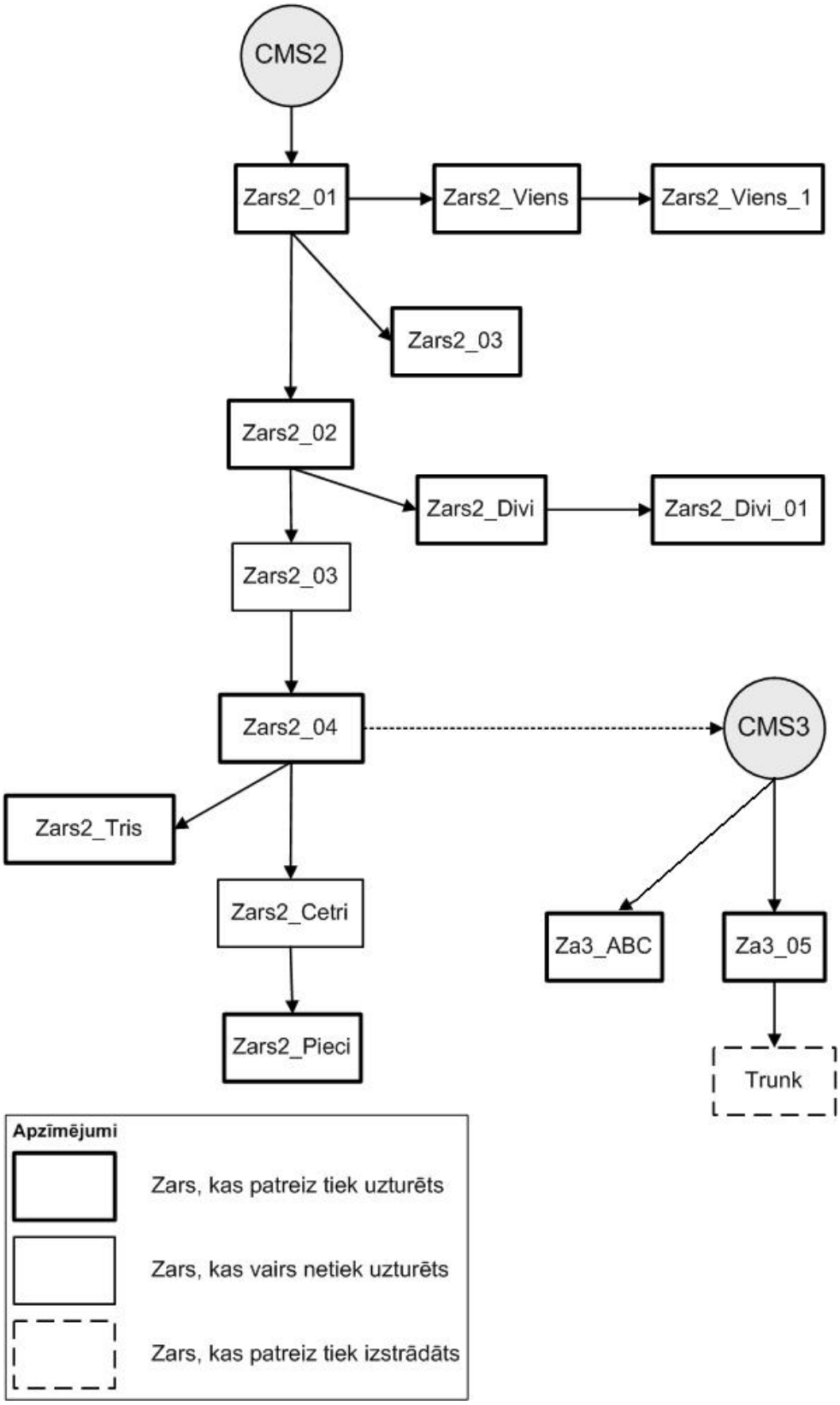
IS Transmaster tiek izstrādāta jau vairāk par 15 gadiem. Ar laiku mainās klientu prasības, tirgus iespējas un piedāvājumi attīstās un dēļ tā arī mainās pati IS.

CMS pirmā versija (KCP vai vēlāk pārsaukta par CMS1) bija realizēta FoxPro valodā. Pēc tam tā tika pārveidota pa jaunam uz Oracle datu bāzes un formām, ievērojami papildinājās funkcionalitāte, t.i. tika izveidota otrā versija (CMS2). Un divu pēdējo gadu laikā tika izveidota jaunā versija (CMS3). CMS3 sastāv no trim tieši saistītam apakšsistēmām – „smagais" un „vieglais" klienti un IssuingWs. Par pamatu CMS3 tika paņemta CMS2, protams, modificēta pamatoties uz klientu prasībām un tas arī ir „smagais" klients. „Vieglais" klients ir „Smaga" klienta vienkāršota versija, tas ir realizēts XUL valodā, ar XML pārskatiem un pieejama caur parasto pārlūkprogrammu. „Smagais" klients ir realizēts Oracle PL/SQL valodā, ar Oracle formām un Oracle report pārskatiem. IssuingWs ir realizēts Java valodā. Katra šī sistēma ir uzturēta versiju kontrolēs rīkos atsevišķi, t.i. 3 atsevišķas sistēmas un 3 atsevišķi repozitoriji.

CMS1 IS ir jau pavisam novecojusi, to nelieto vairs neviena banka un tāpēc tā vispār netiek uzturēta. Patreiz arī CMS2 paliek neaktuāla, tomēr daudzas bankas vēl izmanto to. Daži klienti jau sāka pāriet uz CMS3, tāpēc tagad vienlaicīgi tiek uzturētas gan CMS2, gan CMS3 IS. CMS2 notiek kļūdu labošana un nelielu izmaiņu ieviešana, tomēr kompānija mēģina pārcelt pamat klientus uz CMS3, kas principā visu laiku attīstās, ka arī tiek labotas kļūdas, kas vēl palika pēc CMS2 vai parādījās pa jaunu.

CMS2 un IssuingWs tiek uzturētas ar CVS versiju kontroles rīku un CMS3 tiek uzturēta ar SVN. IssuingWS IS ir relatīvi nelielā sistēma, kas strādā kopā ar CMS3 un tā ir tikai vienā versijā. „Vieglam" klientam patreiz ir tikai 2-as versijas, bet tas jau tiek uzturēts SVN. CMS3 tika izveidota relatīvi nesen un patreiz ir izveidotas tikai 3 versijas. Ar CMS2 ir daudz grūtāk, sistēma nodzīvoja jau ne vienu gadu un pa to laiku bija izveidotas 37 atšķirīgas versijas, no kuram šobrīd tiek uzturētas tikai 10. Katra sistēmas

versija CVS un SVN tiek uzturēta zaru veidā. *14. att*. var redzēt vienkāršoto CMS versiju situāciju uz šo brīdi. Šī nav reālā situācija, tomēr attēla var redzēt situāciju ļoti tuvu esošai. Zaru nosaukumi ir izmainīti, jo CMS2 versijā ir tādi zari, kuru nosaukumi pa tiešo sakrīt ar to banku nosaukumiem, kur speciāli tika taisīts atsevišķais zars. Ka arī, uzskatāmības labad, nav attēloti visi CMS2 37 zari. CMS3 sakumā it kā arī bija uzbūvēti 2 paralēli zari, viens ar bankas nosaukumu, otrs kā sistēmas versijas numurs. Un tagad tiek turpināta izstrāde ar zaru nosaukumiem, kas atbilst versiju numerācijai.



14. att. **CMS2 un CMS3 „Smaga" klienta esošais versiju stāvoklis**

4.3 Patreizējas problēmas

Ja izpētīt *14. att.* detalizētāk, var redzēt, ka zari tiek nosaukti diezgan atšķirīgā veidā un īsti nav saprotama katra zara izveidošanas jēga (vai tas bija veidoti pēc kādam lielām izmaiņām, vai pēc n-daudzuma kļūdu labošanas, vai pēc klientam jauna laidiena nosūtīšanas). Protams, grafā ir redzams kāds zars ir priekštecis pārējiem, taču ne visiem cilvēkiem, kas uzsāk darbu ar kādu no zariem, ir uzreiz saprotams kāds ir šim zaram priekštecis un pēcnācējs, jo šāda pilna CMS2 zarošanas grafa nav. Un tas ir svarīgi zināt, labojot kaut kādu kļūdu, jo nozīmīgi izlabot uzreiz visos zaros, lai pēc tam nav jāatgriežas pie tāda paša rezultāta.

Taču tā zari tika veidoti pamatojoties uz zarošanas modeļa – zars pēc laidiena. Nav īsti saprotams kāpēc daži zari tika nosaukti ar jaunas versijas numuru, bet daži ar klientu (banku) nosaukumiem (*14. att.* tas ir izmainīts uz `Zars2_Viens`, `Zars2_Divi`, utt.). Nav saprotams, jo patiesībā sistēma tika veidota ar dažādu banku licenzēšanu opciju, kas ļauj katrai bankai veidot specifisku risinājumu vienā un tā pašā sistēmā. Taču visticamāk izmaiņas, kas tika veiktas attiecība uz konkrētiem zariem, bija tik lielas, ka bija nolemts izveidot jaunu zaru, lai izstrāde notiktu vienkāršāk. Bet arī pēc to izmaiņu pabeigšanas zari netika sapludināti un turpināja savas dzīves atsevišķi. Toreiz tas visticamāk tiešam bija vieglākais ceļš mērķu sasniegšanai, bet tas nebija pārdomāts, lai arī turpmāk nerastos problēmas.

Izņemot to, ka pirmo reizi izstrādātājam grūti saprast zarošanas vēsturi un paņemt labošanai pareizus zarus, vēl eksistē problēma ar izmaiņu pārnešanu. Tā kā zaru tiešam ir diezgan daudz un principā vairumā tie ir līdzīgi, jo ne vienmēr divos zaros ir visi objekti atšķirīgi, parasti zariem, kas ir bija izdalīti speciāli noteiktiem klientiem, atšķiras tikai daļa no visas funkcionalitātes. Tas nozīme, ka bieži vien labojot kādu lielu vai nelielu kļūdu kādā zarā, ir jāpārnes tas pašas izmaiņas arī citos zaros. Ja tas ir nelielas izmaiņas un jāpārnes tikai vēl vienā zarā, tad principā tas neaizņem pārak daudz laika. Taču, ja izmaiņas ir diezgan lielas un jāpārnes uz vairākiem zariem, tad tas aizņem diezgan lielu laiku gan izstrādātājam, gan pēc tam testētājiem, kas pavisam nav izdevīgi. Jo lielāks zaru skaits, jo grūtāk ir izstrādātājiem labot kļūdas un apjomīgai un sarežģītai sistēmai 10 zari ir diezgan liels skaits.

Vēl ir jāpiemin to, ka produkts ir liels, taču tas tiek pielāgots katram klientam speciāli atbilstoši tā vajadzībām. Klienti nav tikai vienā valstī, tāpēc bieži vien atšķirības sakas ar interfeisa valodu un beidzas ar valsts līkumiem, kas ir jāievēro lietojot konkrēto produktu. Lai šo panākt sistēmā ir ieviests parametrizācijas mehānisms, kas atbild par visu funkcionalitāti. Kā arī kā jau bija minēts ir ievesta licenzēšana, kas atbild par datu apjomiem un produkta pieejamo funkcionalitāti, un katram klientam tiek izdalīta sava licence. Tas ir, ja divos objektos vienīgais kas atšķiras ir kaut kāda parametra vērtības salīdzinājums, tad nedrīkst labot šo daļu balstoties uz atšķirībām. Šādā situācijā ir jāizpēta kāpēc bija pielietots konkrēts parametrs, kādas ir iespējamas vērtības, par ko parametrs atbild un vai tas ir saistīts vēl ar kādu citu. Patreiz neeksistē tāda dokumenta, kur var atrast visiem parametriem saistības, bet ir izmētāts pa visiem funkcionalitātes specifikācijas dokumentiem vai pat ir pieminēts kaut kāda vietā kodā. Tāpēc dažreiz sanāk diezgan daudz laika tērēšanas uz visu šo darbību veikšanu.

Papildus ir jāpiemin to, ka klienta daļa ir uztaisīta ar Oracle formām, kas glabājas binārā formātā. Tas nozīme, ka nevar ar parasto `diff` operāciju vai ar paredzētu tam programmatūru atrast atšķirības divās formās. Ir iespēja dabūt formu teksta formātā un salīdzināt jau to, taču labot vajag vienalga pašā formā, kas ir ļoti neērti un atkal aizņem laiku.

Tātad uz pirmo ieskātu eksistē sekojošas problēmas:

- Liels zaru skaits, kurā ir grūti orientēties un saprast, kuros zaros uzreiz vajag labot kļūdas;
- Izmaiņu pārnešana no viena zara uz citu, kas ir apgrūtināta ar:
  - funkcionalitātes atkarību no parametru iestādījumiem un pielietojumiem;
  - binārs formāts Oracle formām.

4.4 Problēmu risinājums

Esoša CMS2 zaru liela apjoma situācija liecina par to, ka tomēr kaut kas ir jāmaina. Tā kā eksistē vairākas zarošanas metodes, tad šajā darbā ietvaros būs izvelēta metode, kas ir vairāk atbilstoša CMS3 sistēmai. Tā kā pēc *14. att.* var redzēt, ka

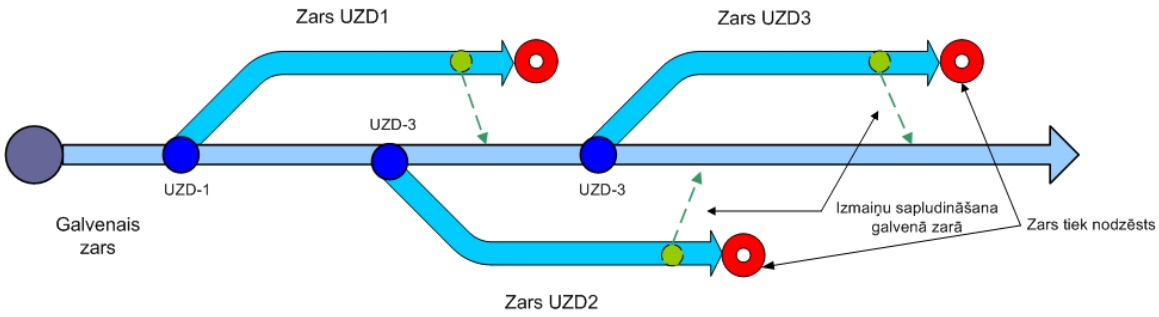
patreizēja CMS3 zarošanas situācija nav tik slikta kā CMS2, tad arī metode būs piemērota CMS3, lai beigās nesanāk tas pats kā ar CMS2. Visi piemēri tiks izskatīti uz CMS2 pamatā, ņemot vērā SVN specifiku.

Pašlaik eksistē sekojošas zarošanas metodes:

- zarošana laidieniem;
- zarošana pēc vajadzības;
- zarošana uzturēšanai;
- zarošana rakstūrpazīmēm;
- zarošana komandai;
- zarošana uzdevumiem.

4.4.1 Zarošana uzdevumiem

Šī metode paredz zarošanu katram programmēšanas uzdevumam, kas ir pieteikts izstrādātājiem (sk. *15. att.*). Tāpēc tikko izstrādātājs uzsāk pildīt kādu uzdevumu (piemēram, `UZD-1`), tad uzreiz izveido jaunu zaru ar strādājošo versiju no galvenā zara. Kad uzdevums ir pabeigts viss tiek notestēts un izmaiņas sapludinātas atpakaļ galvenā zarā, un vēlreiz notestēts jau arī galvenā zarā. Uzdevumu ir diezgan daudz un tāpēc, lai neveidotu milzīgu zaru skaitu, kurā nevar neko saprast, neilgi pēc izmaiņu sapludināšanas zars tiek nodzēsts [12].



15. att. **Zarošana uzdevumiem**

Šīs metodes plusi:

- Galvenā zara versija visu laiku paliek stabilā, jo visas izmaiņas, kas tika veiktas atsevišķos zaros, ir rūpīgi notestētas;
- Kļūdu labošana netraucē veidot strādājošus ielāpus no galvenā zara.

Šīs metodes minusi:

- Ja izstrādes grupa ir liela un uzdevumu ir daudz, tad zaru skaits būs liels;
- Nav jēgas veidot zaru maziem uzdevumiem, kas neietekmē programmatūras kopējo darbību;
- Katram izstrādātājam jāatceras, ka pēc neilgi pēc izmaiņu sapludināšanas, zaru ir jānodzēš;
- Uzdevumu zarus var netīšam sajaukt.

Vai šī metode der CMS3?

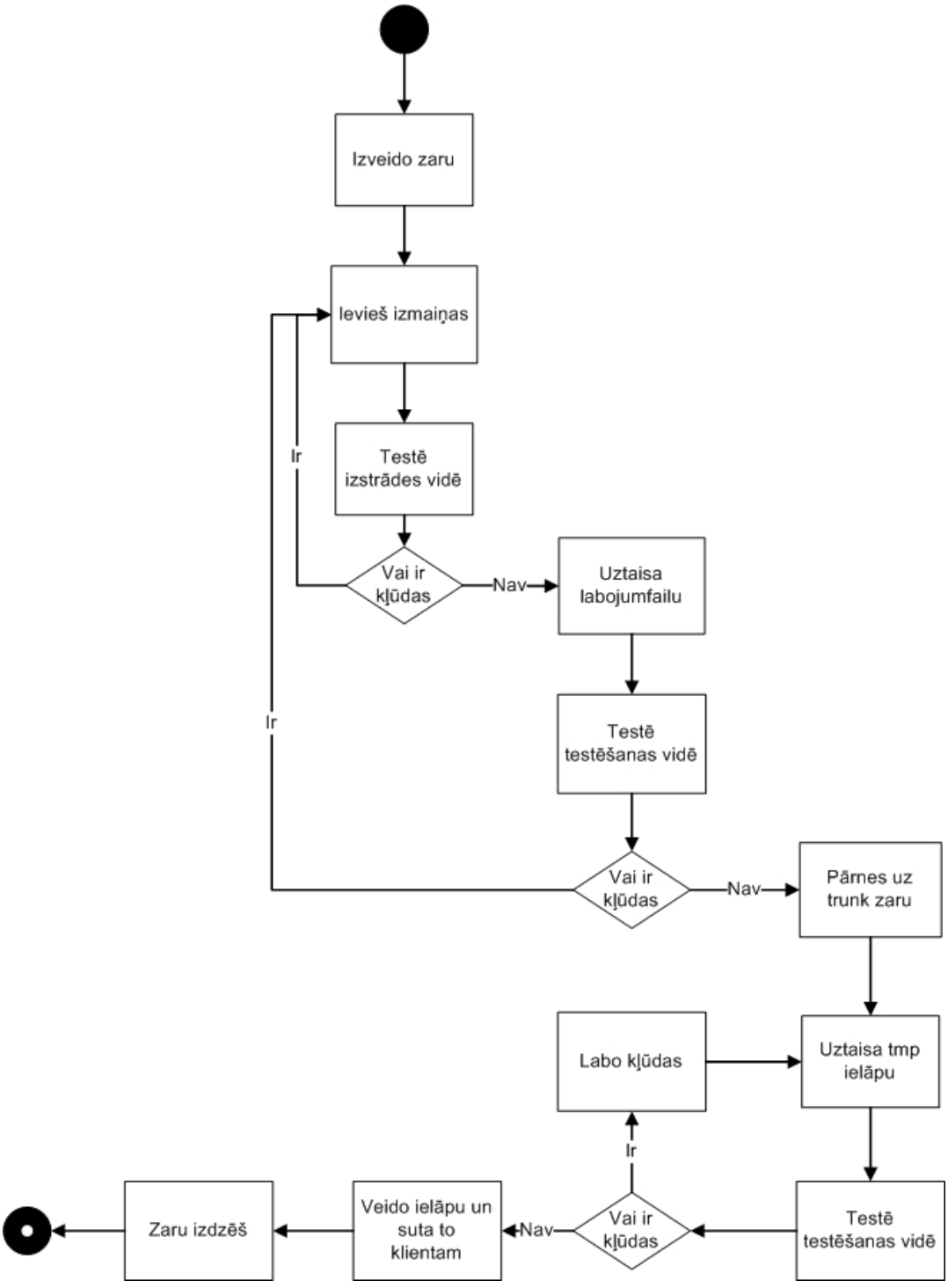
Šī metode vairāk der nelieliem produktiem, ar mazo izstrādātāju skaitu. CMS3 šī metode kā pamat metode neder, jo izstrādes grupa ir diezgan liela, jo kļūdu skaits pārsniedz 2000, jo produkts jau ir diezgan liels un visu laiku attīstoties, rodas jaunas versijas, kuram izdalīt atsevišķus zarus ir lietderīgāk, nekā katram uzdevumam.

Taču šī metode arī var būt ļoti noderīga gadījumos, kad ir jāveic lielas izmaiņas vai labot nopietnu kļūdu. Tad var izveidot atsevišķi zaru un labot tajā, pēc tam pārnesot izmaiņas uz galveno zaru.

Tā kā SVN piedāvā trīs direktorijas zarošanām – Trunk, Branches un Tags, var mēģināt ieviest šo zarošanas metodi vienā no šīm direktorijām. Trunk direktorija atbild par galveno izstrādes zaru, t.i. tāds zars, kurā patreiz notiek jaunas versijas izstrāde. Branch direktorija ir paredzēta pārējiem zariem, kas tika izveidoti no galvenā zara un kuriem var veikt izmaiņas, neskarot galveno koku. Un Tags direktorijā glabājas produkta kopijas noteiktos laika momentos, bet tas vairs netiek mainītas.

Lai būtu vieglāk atsekot visus zarus uzdevumiem, tie tiks veidoti SVN Branches direktorijā. Uzskatāmības labad zari tiks saukti atkarība no uzdevuma jēgas, t.i. IP `za3_task_UZD#` un kļūdas labojumi `za3_bug_UZD#`. Pēc labojumiem jaunā zarā un to notestēšanas, izstrādātājs veido labojumfailu un nosūta to testētājiem, ja viss testi izgāja

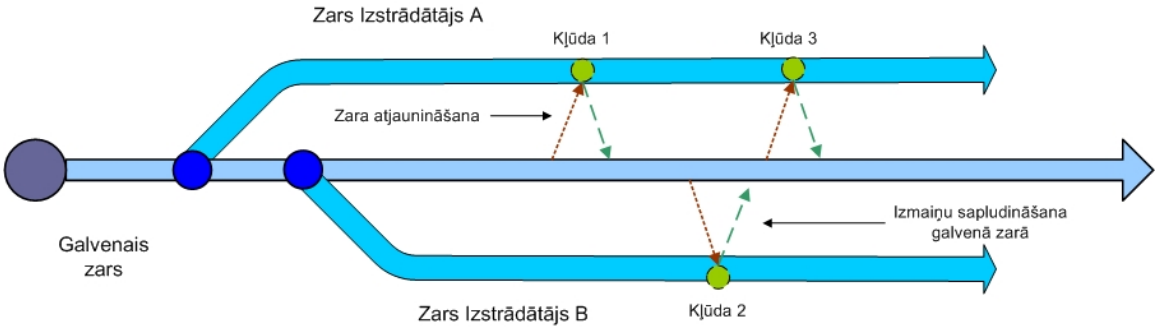
veiksmīgi, izmaiņas tiek pārnestas uz galveno zaru (`trunk`) un, ja arī tur viss strādā pareizi, tad tiek izveidots ielāps, kuru var jau sūtīt klientiem. Tikko izmaiņas ir apstiprinātas uz ielāpa veidošanu, jāizdzēš tā uzdevuma zars (sk. *16. att.*).



16. att. **Zarošana uzdevumiem – uzdevumu pildīšana CMS3 ietvaros**

4.4.2 Zarošana komandai

Šī metode (dažos avotos tiek saukta par zarošanu izstrādātājam) ir līdzīga iepriekšējai, ar vienu atšķirību – zarus veido nevis katram uzdevumam, bet uz katru izstrādātāju vai izstrādātāju grupu un pēc tam katrs veic labojumus sistēmai, bet tikai sev izdalīta zarā (sk. *17. att.*) [12].



17. att. **Zarošana izstrādātājam**

Šīs metodes plusi:

- Salīdzinājumā ar zarošanu uzdevumiem, šai metodei rezultātā ir mazāk zaru un tajos ir vieglāk orientēties;
- Šī metode var ļoti noderēt jauniem darbiniekiem, jo ir iespēja izveidot sev atsevišķu zaru un jau tur apgūt sistēmu.

Šīs metodes minusi:

- Jo ilgāk notiek izstrāde paralēlos zaros, jo grūtāk paliek sinhronizēt izmaiņas starp zariem, tāpēc šī metode prasa, lai jebkura izmaiņa būtu precīzi ievietota galvenā zarā;
- Lieliem projektiem ar lielu darbinieku plūsmi var notikt situācija, kad ir jālabo vecas izmaiņas, bet cilvēks jau sen uzņēmumā nestrādā un kas bija viņa zarā neviens nezin.

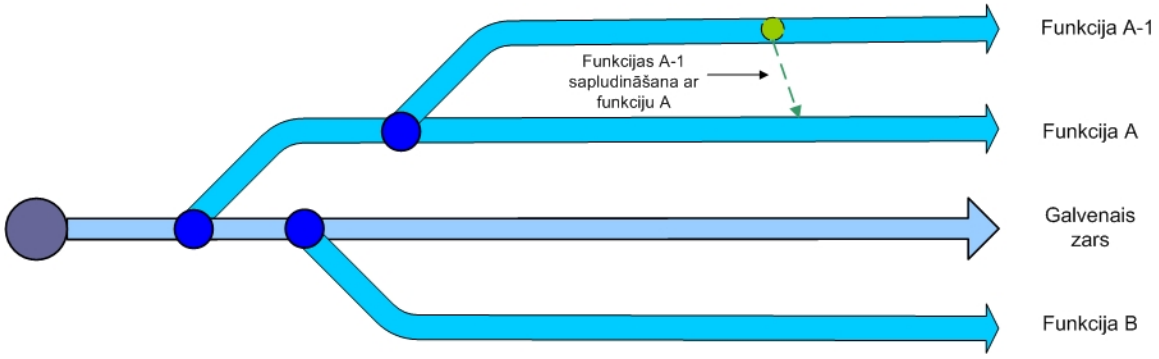
Vai šī metode der CMS3?

Šī metode noteikti nekāda veida nevar būt par pamat metodi CMS3, jo programmatūrai ir lieli apjomi un arī cilvēki nāk un aiziet. Kā arī kompānijai būs atkal

jātērē, jo izmaiņu pārnešana CMS3 ir diezgan apjomīgs darbs. Visticamāk šī metode der maziem vai vidējiem projektiem, kur vēl nav sarežģītas funkcionalitātes apakšā.

4.4.3 Zarošana raksturpazīmēm

Šī ir vairāk hierarhiskā zarošana nekā divas iepriekšējas. Šis metodes jēga ir sadalīt programmatūru uz daļām un katrai daļai izdalīt zaru, kurā notiks šīs daļas izstrāde process. Ar to zaru strādā, kamēr tas koda gabals nekļūst vairāk vai mazāk stabils un pēc tam sapludina to kopā ar galveno produkta zaru (sk.). Tātad zari pamatojas uz programmatūras raksturpazīmēm.



18. att. **Zarošana raksturpazīmēm**

Šīs metodes plusi:

- Šī metode ir ļoti laba programmatūras izstrādes brīdī, jo ir iespēja normāli izstrādāt atsevišķi visas pamat funkcionalitātes daļas;

Šīs metodes minusi:

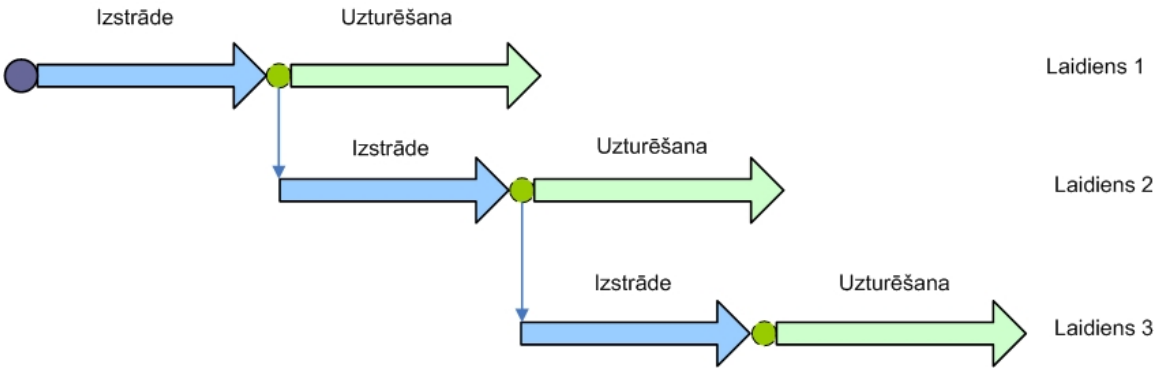
- Uzstūrēšanas projektam šī metode neder, jo galvenā funkcionalitāti ir jau izstrādāta un nav nepieciešamības veidot šādus zarus.

Vai šī metode der CMS3?

CMS sen jau nav izstrādes projekts. Protams, kaut kādas izmaiņas tiek izstrādātas, bet tas nav tik globālas. Jebkurā gadījumā, ja pat izmaiņas, kas jāveic ir tiešam lielās, tad var izmantot zarošanu uzdevumiem, t.i. izveidot jaunu zaru speciāli jaunajam IP.

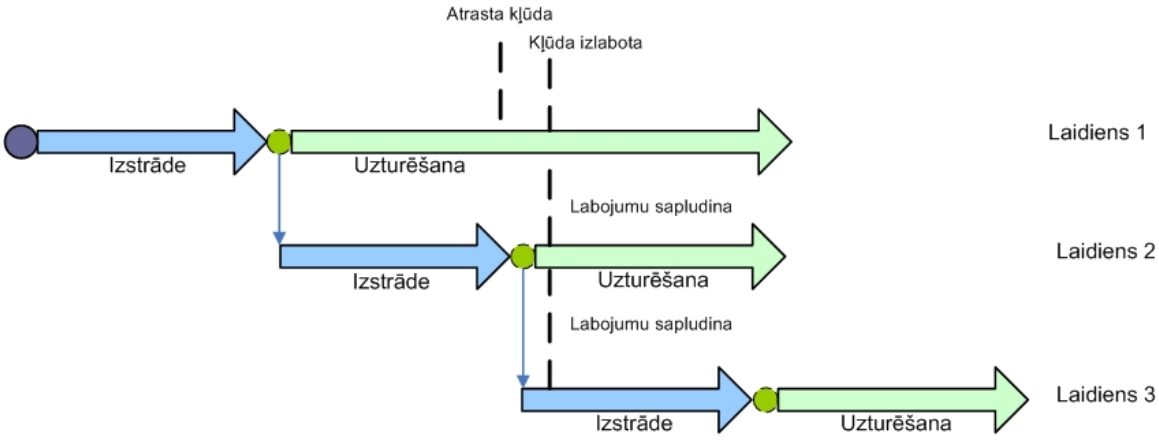
4.4.4 Zarošana laidieniem

Šī metode no pirmā acu skatiena liekas visvienkāršākā saprašanai. Sakumā notiek izstrāde, tikko viss ir notestēts, taisa laidienu (`Laidiens 1`) un sūta to klientam, tālāk šīs zars pāriet uzturēšanas fāzē un tiek izveidots jauns zars nākamai produkta versijai (sk. *19. att.*) [14].



19. att. **Zarošana laidieniem**

Taču viss nav tik vienkārši. Tikko pie klienta ir atrasta kļūda, tā ir jāizlabo tajā zarā, kur tā tika atrasta, taču papildus tam ir jāpārnes tas izmaiņas arī uz pārējiem jaunākiem zariem, lai tā vairs neatkārtojas (sk. *20. att.*).



20. att. **Kļūdas labošana ar laidienu zarošanas metodi**

Šīs metodes plusi:

- Šī ir vienkāršāka zarošanas metode;
- Zarošanas skaits ir atkarīgs no programmatūras versiju skaita, t.i. parasti nav liels;

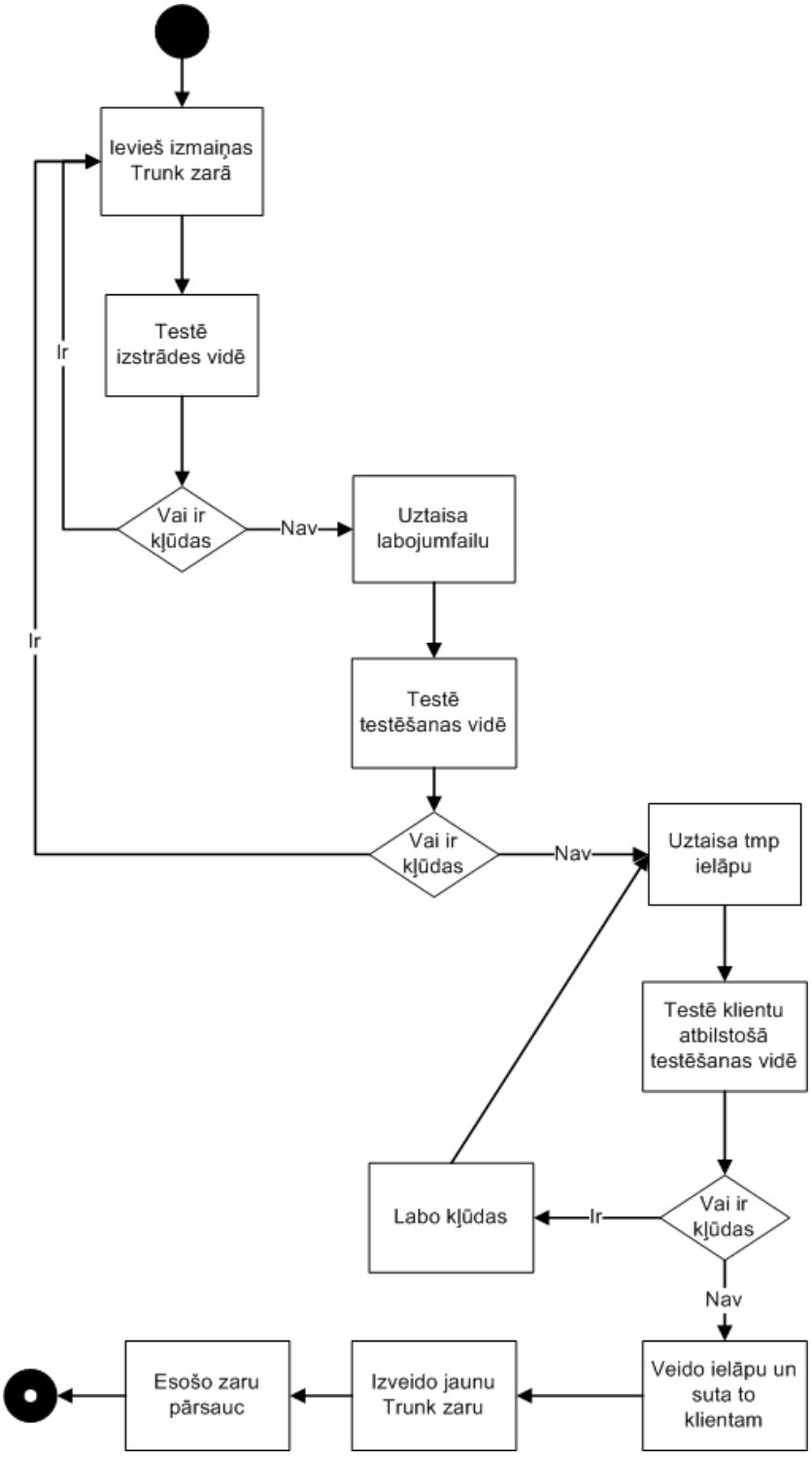
Šīs metodes minusi:

- Jo lielāks ir zaru daudzums, jo vairākiem zariem ir jāpārnes izmaiņas kļūdas gadījumā;
- Ar šo metodi ir grūtāk veidot vairākus laidienus vienlaicīgi, jo būs jādara daudz sapludināšanu ar citiem zariem.

Vai šī metode der CMS3?

Šī metode bija izmantota kā pamat metode CMS2 un arī patreiz ir tāda tendence izmantot šo metodi arī CMS3. Principā var šo metodi izmantot arī tālāk, taču ir viena nianse. Agrāk CMS2 bija pielietota zarošanas uzturēšanai metode, t.i. tika veidoti atsevišķi zari klientiem, kas gribēja stabilu versiju atbilstošu viņu prasībām. Lai nepieļautu arī CMS3 šādu situāciju ir jāveido versijas, kas atbilst klientu nosūtītiem laidieniem. Tas nozīme, ka, kad uztaisa visas izmaiņas nepieciešamas kādam klientam, notestē un izveido jaunu laidienu, tiek izveidots jauns zars, kurā tālāk notiks izstrādes darbi un laidiena zars pariet uzturēšanas fāzē (sk. *21. att.*).

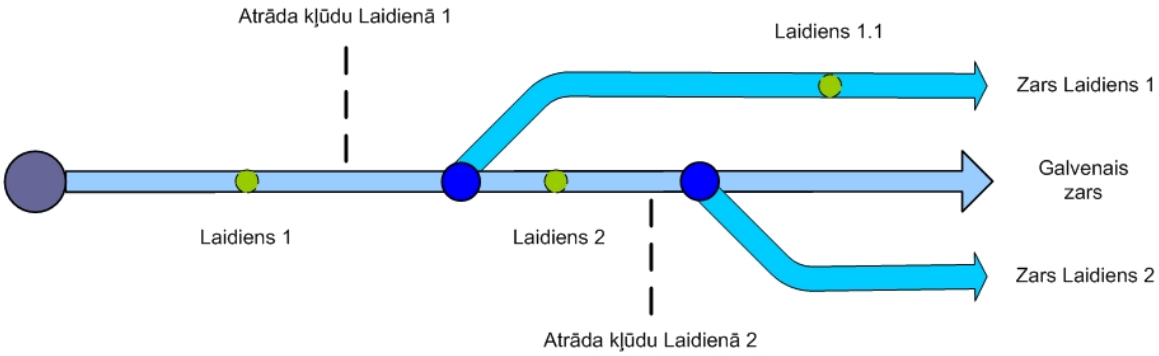
Principā var turpināt izmantot šo metodi, taču, ja iedomāties cik būs šo zaru pēc gada, kad uz CMS3 migrēs vairāki klienti, kas patreiz lieto CMS2, ka arī jaunas versijas pieprasīs klienti, kas jau lieto CMS3, tas sanāk apmēram tāds pats zaru skaits kā CMS2. Šāda rezultāta pavisam negribas, tāpēc izskatīsim vēl vienu zarošanas metodi.



21. att. **Zarošana laidieniem – uzdevumu pildīšana CMS3 ietvaros**

4.4.5 Zarošana uzturēšanai

Šī metode ir ļoti līdzīga zarošanai laidieniem, ar vienīgu atšķirību – jauns zars tiek veidots tikai tad, kad ir nepieciešamība veikt kaut kādas izmaiņas vecajam laidienam (sk. *22. att.*) [13]. Kad visas izmaiņas ir jau paveiktas, tad nav obligāti jāpārnes šis izmaiņas galvenā zarā. Visu laiku tiek attīstīts galvenais zars un ja ir vēl kaut kādas kļūdas laidienu zaros, tad arī tur tas tiek labots. Šī metode ir izdevīga tad, kad ir vajadzība nedaudz pamodificēt iepriekšējus laidienus jauniem klientiem, bet tālāk nepiedāvāt jaunus risinājumus, t.i. to, kas ir jau uz to brīdi galvenā zarā.



22. att. **Zarošana uzturēšanai**

Šīs metodes plusi:

- Ja klients piekrīt saņemt vienu vecāku versiju, papildinātu zem viņa prasībām, tad pielabot tikai vienu zaru un atdot to klientam ir izdevīgi.

Šīs metodes minusi:

- Ja klients pēc savas versijas saņemšana nolemj iegūt jaunāku produkta versiju, tad nāksies pārnest visas izmaiņas uz jaunākiem zariem, kas prasīs ļoti daudz resursus;
- Ja šādi klienti bija vairāki un arī vairāki vēlas sev jaunāko versiju, tad var gadīties, ka būs jālabo diezgan lielu produkta daļu, jo var gadīties lieli konflikti starp diviem tiem klientiem paredzētiem zariem.

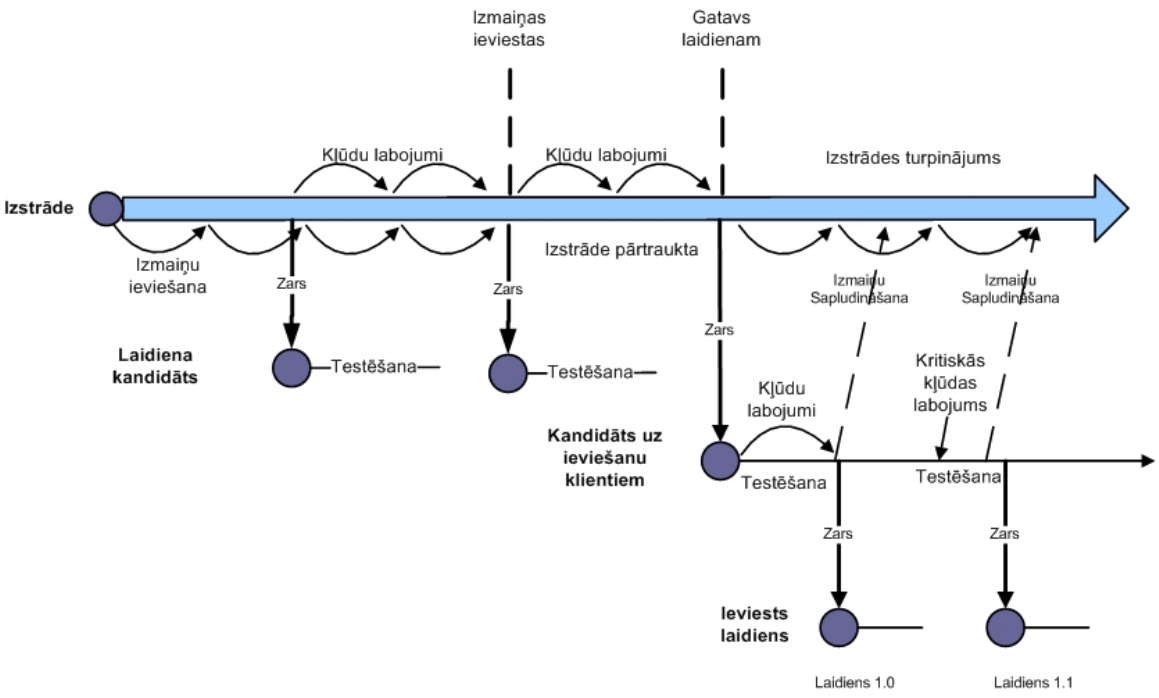
Vai šī metode der CMS3?

Iespējams šī metode daļēji bija izmantota CMS2 zaru veidošanā, jo ir diezgan daudz zaru, kas tika izveidoti speciāli atsevišķiem klientiem. Taču realitāte ir tāda, ka vairums no tiem klientiem gribēja jaunas CMS versijas un nācās vienalga uzturēt viņu zarus, papildus tam veidot migrācijas skriptus uz jaunu CMS3, jo pārveidot galveno zaru CMS2 būtu neracionāli.

Šo praksi atkārtot CMS3 nav vajadzības, jo CMS3 ir uzlabota parametrizācijas un licenzēšanas funkcionalitāte. Ja paradīsies kāds klients, kuram būs vajadzīga stabila CMS3 versija, tad tiks pārtaisīta pēdēja laidiena versija un visas izmaiņas ienestās galvenā zarā. Tas samazinās vēlāko izmaiņu pārnešanu jaunākos zaros.

4.4.6 Zarošana pēc vajadzības

Šīs metodes galvenā doma ir taisīt zarus tikai pēc vajadzības. Tas ir, ja vajag notestēt vai ir jāveido jauns laidiens, tad arī taisa jaunu zaru (sk. *23. att.*). Sakumā izveido atsevišķo zaru testēšanai, kamēr tiek meklētas kļūdas izstrāde turpinās un arī tiek labotas atrastas kļūdas. Pēc tam izveido jaunu zaru atkārtotai testēšanai un pārtrauc izstrādi galvenā zarā, tikai labo atrastas kļūdas. Kad viss strādā stabili, atkal tiek izveidots zars laidienam un turpinās izstrāde galvenā zarā. Laidienā zarā atkal viss tiek pārtestēts uz klienta testēšanas datu bāzes kopijas, atrastas kļūdas tiek labotas un izmaiņas sapludinātas galvenā zarā. Tikko viss ir notestēts uztaisa jaunu zaru un uz tā pamatā izveido laidienu. Zars paliek neskārts, t.i. tas paliks šādā veidā, lai vēlāk var atrast kļūdas izlaistā laidienā. Ja klientā pusē tika atrastas vēl kaut kādas kļūdas, tas tiek labotas zarā, kas bija veidots laidiena testēšanai, un sapludinātas iekš izstrādes galvenā zarā. Tikko izmaiņas ir notestētas, veido laidienu un vēl vienu zaru, kas paliks atkal neskarts.



23. att. **Zarošana pēc vajadzības**

Šīs metodes plusi:

- Ir tikai viens izstrādes zars, tāpēc nav katru reizi jādomā par izmaiņu sapludināšanu citos zaros, ja bija kaut kur atrasta kļūda;
- Izmaiņu sapludināšana ir relatīvi viegla, jo starp zariem nav īpašu atšķirību;
- Vairāku reižu testēšana un uz laiku izstrādes pārtraukums ļauj izveidot stabilu programmatūras versiju.

Šīs metodes minusi:

- Sakumā šī metode liekas nesaprotama.

Vai šī metode der CMS3?

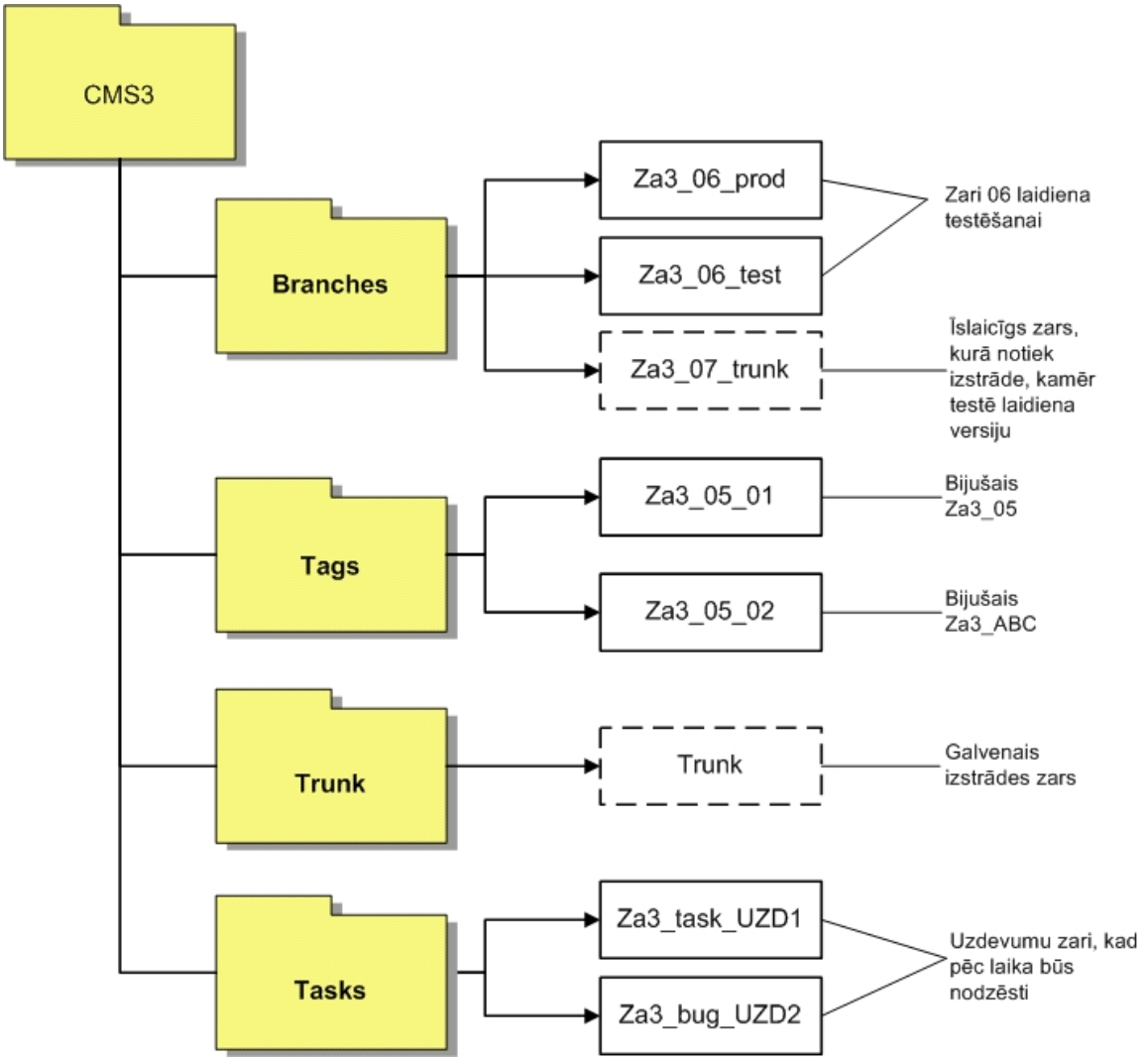
Šī metode noteikti ir piemērota CMS3 izstrādei. Jo visi procesi, ko šī metode iekļauj, notika jau arī agrāk. Visi laidieni (vai versijas) tiek saglabāti neskartā veidā, tāpēc jauniem klientiem ir iespēja piedāvāt uzreiz stabilo versiju.

Diemžēl šī metode neparedz nepārtraukto izstrādi. Kamēr tiek testēta pirms laidiena versija, nevar neko labot. Principā arī šo trūkumu var apiet. Laikā, kad notiek

testēšana var veidot jaunu īslaicīgu zaru, kurā tiks turpināta izstrāde. Tikko laidiens ir notestēts īslaicīgu zaru sapludina kopā ar galveno zaru un jau tur turpina izstrādi.

4.4.7 Izvēlēta metode

CMS3 par zarošanas metodes pamatu ir izvēlēta zarošana pēc vajadzības. Kā arī lieliem uzdevumiem tiek pielietota uzdevumu zarošanas metode. Patreiz zari izskatās kā ir attēlots *24. att.*



24. att. **Zarošana CMS3**

Visi zari tiek saukti pēc noteiktiem līkumiem:

- *Trunk* direktorijā vienmēr atrodas tikai galvenais izstrādes zars ar nosaukumu `Trunk`. Šajā zarā notiek izstrāde un tas nekad netiek dzēsts.
- *Tags* direktorijā atrodas laidienu zari, šos zarus neviens nemaina. Nosaukumi tiek doti `Za3_LaidienaNumurs` formātā. Šie zari nekad netiek dzēsti.
- *Tasks* direktorijā atrodas uzdevumu veikšanas zari. Nosaukumi tiek doti `Za3_bug_UzdevumaNumurs` un `Za3_task_UzdevumaNumurs` formātā. Šie zari tiek dzēsti tad, kad kļūdas risinājums ir apstiprināts.
- *Branches* direktorijā tiek glabāti:
  - īslaicīgie zari izstrādei, kamēr notiek laidiena testēšana. Nosaukumi tiek doti `Za3_LaidinaNumurs_trunk` formātā. Šos zarus dzēš pēc pusgada no tā brīža, kad laidiens bija aizsūtīts klientiem un nebija kritisko kļūdu.
  - zari laidiena testēšanai. Nosaukumi tiek doti Za3`_LaidienaNumurs_test` un `Za3_LaidienaNumurs_prod` formātā. Šos zarus dzēš pēc pusgada no tā brīža, kad laidiens bija aizsūtīts klientiem un nebija kritisko kļūdu.

Īstenībā nekas netiek izdzēsts no SVN repozitorijā, tas vienkārši paliek neredzams, taču vēsture par zaru (vai objektu) glabājas un jebkurā brīdi var atgriezt izdzēsto.

Pateicoties izvelētai zarošanas metodei sapludināt divus zarus palika krietni vieglāk, jo izmaiņas parasti ir niecīgas. Un ja notiek konflikts ar parametru lietošanu, ir viegli atrast kāpēc tieši šādi parametri ir izmantoti objektos. Jo visas izmaiņas, kas ir jāpārnes uz citu zaru ir nesen veidotas un labi nokomentētas. Lai būtu vieglāk atrast dokumentāciju, pēc kuras tika taisītas izmaiņas, JIRA problēmu pārvaldības rīkā tika ieviests jauns lauks, kurā analītiķis ierakstā specifikācijas atrāšanas ceļu. Rezultātā laiks, kas agrāk bija veltīts izmaiņu pārnešanai ir samazinājis.

Patreiz pat dažos gadījumos ir iespējams izmantot SVN iekšējo funkciju `merge`, t.i. veidot izmaiņas automātiski. Tāpēc ir izveidots skripts (sk. 1. Pielikums), kas veic

pārbaudes vai esam pareizā direktorijā, vai abiem zariem ir pēdēja versija, vai objekti nav slēgti, veic SVN `merge` komandu un paziņo par rezultātiem. Tomēr tā kā paļauties uz automātisko sapludināšanu nevar, tāpēc pirms tas palaišanas vajag pārliecināties, ka starp atšķirīgiem objektiem nebūs konfliktu un ka izmaiņu pārnešana neko nesalauzis.

4.4.8 Veco un jaunu zarošanas metožu salīdzinājums

Izskatīsim piemēru uz vecas un jaunas zarošanas metodes pamata.

Situācija

Ir divi klienti `Klients1` un `Klients2`. `Klients2` parādījās vēlāk par `Klients1`, tāpēc zars `za3_klients2` ir `za3_klients1` zara modifikācija. Abiem diviem jau ir piegādātas attiecīgi pēdējas produkta versijas (klientam 2. versija ir jaunāka par versiju klientam 1.). Izstrādes zarā ir atrasta nopietnā kļūda, kuru klienti patreiz nav pamanījuši. Šī kļūda ir izlabota izstrādes zarā un ir saskaņots ar klientiem, ka to labos arī viņu zarā.

Labojums ar veco metodi

Ir 3 zari (`za3_klients1`, `za3_klients2` un `trunk`), kļūda atkārtojusies visos šajos zaros. Lai izlabotu kļūdu izstrādes zarā bija veiktas sekojošas izmaiņas:

koda gabals

```
IF SUBSTR(v_param124, 1, 1) = '0' THEN
  DBMS_OUTPUT.put_line('Izmaiņas tiks saglabātas pēc procesa
pabeigšanas');
…
ELSIF SUBSTR(v_param124, 1, 1) = '1' THEN
  DBMS_OUTPUT.put_line('Izmaiņas veiksmīgi saglabātas');
…
END IF;
```

bija pārveidots par

```
IF v_param124 != 'nn' THEN
  IF SUBSTR(v_param124, 1, 1) = '0' THEN
    DBMS_OUTPUT.put_line('Izmaiņas tiks saglabātas pēc procesa
pabeigšanas');
…
  ELSIF SUBSTR(v_param124, 1, 1) = '1' THEN
    DBMS_OUTPUT.put_line('Izmaiņas veiksmīgi saglabātas');
…
  END IF;
ELSE
  raise_application_error(-20000, 'Aizpildiet paramētru #124!');
```

```
END IF;
```

Zarā `za3_klients1` parametrs 124 sastāvēja tikai no 1 simbola un šīs koda gabals izskatījās sekojoši

```
IF v_param124 = '0' THEN
  DBMS_OUTPUT.put_line('Izmaiņas tiks saglabātas pēc procesa
pabeigšanas');
…
ELSE
  DBMS_OUTPUT.put_line('Izmaiņas veiksmīgi saglabātas');
…
END IF;
```

Tāpēc to izlaboja uz

```
IF v_param124 != 'n' THEN
  IF v_param124 = '0'  THEN
    DBMS_OUTPUT.put_line('Izmaiņas tiks saglabātas pēc procesa
pabeigšanas');
…
  ELSIF v_param124 = '1' THEN
    DBMS_OUTPUT.put_line('Izmaiņas veiksmīgi saglabātas');
…
  END IF;
ELSE
  raise_application_error(-20000, 'Aizpildiet paramētru #124!');
END IF;
```

Zarā `za3_klients2` 124 parametrs jau sastāvēja no 2 simboliem, tāpēc to no

```
IF SUBSTR(v_param124, 1, 1) = '0' THEN
  DBMS_OUTPUT.put_line('Izmaiņas tiks saglabātas pēc procesa
pabeigšanas');
…
ELSE
  DBMS_OUTPUT.put_line('Izmaiņas veiksmīgi saglabātas');
…
END IF;
```

Izlaboja uz

```
IF v_param124 != 'nn' THEN
  IF SUBSTR(v_param124, 1, 1) = '0' THEN
    DBMS_OUTPUT.put_line('Izmaiņas tiks saglabātas pēc procesa
pabeigšanas');
…
  ELSIF SUBSTR(v_param124, 1, 1) = '1' THEN
    DBMS_OUTPUT.put_line('Izmaiņas veiksmīgi saglabātas');
…
  END IF;
ELSE
  raise_application_error(-20000, 'Aizpildiet paramētru #124!');
```

```
END IF;
```

Pēc tam šo visu vēl pārtestēja visos trijos zaros. Tāpēc sanāk, ka bija jālabo 3 zaros, jāskatās kāda bija situācija ar 124 parametru uz laidiena veidošanas brīdi un jāpārtestē. Pēc dota piemēra var redzēt, ka vienā zarā nācās izmainīt kļūdas labojumu, lai tas derētu versijas konfigurācijai. Protams, šīs ir diezgan primitīvs piemērs, kur nekas īpaši nenotiek, taču dzīvē mēdz būt diezgan sarežģītas kļūdas, kuras labošanai nepieciešams krietns laiks.

Labojums ar jauno metodi

Ir viens zars – `trunk` un direktorijā Tags ir divi zari `za3_05_01` un `za3_05_02` (bijušie `za3_klients1` un `za3_klients2`). Kad pārgāja uz jaunu zarošanas metodi tikai pārskatītas visu objektu atšķirības starp bijušiem atsevišķi uzturamiem zariem un katram zaram atbilstošie atšķirības gabali pārveidotas tādā veidā, lai tas strādātu visiem klientiem kādu kļūdu labošanas gadījumā. Tāpēc `trunk` zarā šīs pats koda gabals izskatījās šādi

```
IF SUBSTR(v_param124, 1, 1) = '0' THEN
  DBMS_OUTPUT.put_line('Izmaiņas tiks saglabātas pēc procesa
pabeigšanas');
ELSIF SUBSTR(v_param124, 1, 1) = '1'  THEN
  DBMS_OUTPUT.put_line('Izmaiņas veiksmīgi saglabātas');
END IF;
```

un to izlaboja uz

```
IF SUBSTR(v_param124, 1, 1) != 'n' THEN
  IF SUBSTR(v_param124, 1, 1) = '0' THEN
    DBMS_OUTPUT.put_line('Izmaiņas tiks saglabātas pēc procesa
pabeigšanas');
  ELSIF SUBSTR(v_param124, 1, 1) = '1'  THEN
    DBMS_OUTPUT.put_line('Izmaiņas veiksmīgi saglabātas');
  END IF;
ELSE
  raise_application_error(-20000, 'Aizpildiet paramētru #124!');
END IF;
```

Šīs izmaiņas tika veiktas tikai vienā zarā, nebija nepieciešamības kaut ko meklēt par 124 parametru, taču vienalga šo ir jāpārtestē abu klientu testēšanas vidē.

Rezultāts

Vienas un tās pašas kļūdas labošana vecajai metodei prasītu daudz vairāk laika nekā jaunajai, taču rezultāts abām divām ir vienāds – kļūda ir izlabota gan izstrādes zarā, gan abiem diviem klientiem.

4.5 Secinājums

Pagaidām CMS3 informācijas sistēmai ir tikai 2 klienti, tas nozīme, ka pēc iepriekšējas zarošanas metodes ir jābūt vismaz diviem zariem ar vai nu versijas nosaukumiem, vai ar klientu. Nākamā kvartālā uzsāks vēl viena klienta migrāciju no CMS2 uz CMS3, tas nozīme, ka būtu jau trīs zari. Tikko vēl citi klienti sagribēs sev CMS3, uzreiz pieaugtu zaru skaits. Un ja kādam no klientiem sagribētos kaut kādu funkcionalitāti, kas jau ir citam klientam, tad būtu uzreiz jāpārnes uz citu zaru. Tas prasītu laiku, jo

- ir jāatrod īstas vietas;
- ir jāpārnes atšķirības;
- jānotestē;
- un tikai tad, kad viss ir rūpīgi notestēts, jāveido ielāpu un jāsūta klientam.

Vēl briesmīgāka būtu situācija, kad atrastos kļūda, kas ir visos zaros. Tad papildus iepriekšēji aprakstīts algoritms pareizinātos ar zaru skaitu. Protams, ne visos zaros tai kļūdai ir jābūt, taču jāpārbauda ir visi.

Pats sliktākais variants būtu, ja kļūdā ietilps kaut kas saistīts ar sistēmas parametriem. Tas apgrūtinātu darbu vēl ar parametru vēstures meklēšanu un saprašanu vai būtu pareizi izlabot konkrēto kļūdu visos tajos zaros, iespējams tur parametrs vēl nozīmēja kaut ko pavisam citu vai arī to vērtības bija atšķirīgas. Šīs apstāklis palielinātu darbu vēl uz n-laiku[2].

Taču vairs tas viss vairs nav aktuāls, jo ar jaunas metodes ieviešanu situācija ir izmainījusies. Tagad nav un nebūs katram klientam atsevišķa zara, visas izmaiņas notiek

[2] N-laiks, jo parametru atšķirības varētu būt aprakstītas tikai kādā no specifikācijām, kas bieži vien mēdz atrasties pavisam neparedzamā vietā.

vienā zarā. Nav jāpārnes izmaiņas uz citiem zariem un jāskatās vai vispār vajag tur pārnest. Ir mazāk iespēju neizlabot kļūdu kādam klientam. Protams, kods dēļ programmatūras konfigurācijas modifikācijas ir palicis sarežģītāks. Taču sekojot dokumentācijai arī tas nesastāv problēmu un laiks uz dažādu klientu programmatūras versiju uzturēšanu vienalga netiek tērēts tik pat daudz kā agrāk.

5 NOBEIGUMS

Darba teorētiskajā daļā, kas sastāv no divām pirmām daļām, ir apskatīta PKP galvenā ideja un tas funkcionalitāte. Katra funkcija ir izskatīta atsevišķi ar atbilstošiem piemēriem. Speciāli atsevišķi aplūkota versiju kontroles pārvaldības terminoloģija un piedāvāts apskats divos populārākos rīkos – SVN un CVS.

Darba praktiskā daļā tika aprakstīta autores darbavietā izstrādāta Transmaster informācijas sistēma, aplūkota tas arhitektūra un pārskatīta galvenā biznesa funkcionalitāte. Tur pat ir detalizēti izskatīts CMS2 un CMS3 versiju kontroles pārvaldības esošais stāvoklis pirms darba uzsākšanas. Ir saprasts, ka CMS2 liels apjoms zaru, izveidotu pēc atšķirīgam zarošanas metodēm visu laiku rod problēmas izstrādātājiem un ir jāievieš pastāvīga zarošanas metode, kas tiks pielietots CMS3, kamēr tas stāvoklis nav tik slikts, ka CMS2. Dēļ paralēli uzturamo zaru liela apjoma ir grūti veikt parastas ikdienas operācijas ar versiju uzturēšanu un izstrādi. Tāpēc tālāk darbā ir izpētītas eksistējošas zarošanas metodes, uzskaitītas to stipras un vajās vietas un ir saprasts vai katra no apskatāmām metodēm der CMS3 kā problēmu risinājums.

Beigās ir piedāvāta sava zarošanas metode, kas ir balstīta uz jau eksistējošam, taču pielāgota konkrētai CMS3 sistēmai. Šī metode arī ir veiksmīgi ieviesta CMS3 sistēmas versiju kontroles pārvaldībā. Ir izsecināts, ka pārējas problēmas būtiski samazinās dēļ izvelētas metodes. Tāpēc darba atvieglošanai vēl ir piedāvāts neliels skripts, kas veic nepieciešamas pārbaudes un zaru sapludināšanu.

6 LITERATŪRAS SARAKSTS

[1] B. Appleton, "*Software Configuration Management*" [tiešsaiste]. – [atsauce 15.01.2008.]. Pieejams: http://www.cmcrossroads.com/cgi-bin/cmwiki/edit/CM/MainBradAppleton?topicparent=CM.SoftwareConfigurationManagementhttp://www.cmcrossroads.com/cgi-bin/cmwiki/view/CM/SoftwareConfigurationManagement .

[2] I. Crnkovic, *Implementing and Integraing Product Data Management and Software Configuration Management*, 2003, p. 338.

[3] P. H. Feiler, *Configuration management models in commercial environments*, Software Engineering Institute, 1991, p. 59.

[4] J. Ferguson Smart, "*Merging and branching in Subversion 1.5*", JavaWorld.com, 29. Jan. 2008 [tiešsaiste]. – [atsauce 06.04.2008.]. Pieejams: http://www.javaworld.com/javaworld/jw-01-2008/jw-01-svnmerging.html?page=1

[5] Swedish Standards Institute, *Quality Management—Guidelines for Configuration Management,* ISO 10 007, 1995, p. 24.

[6] A. Serban, *Visual Sourcesafe 2005 Software Configuration Management In Practice*, 2007, p. 400.

[7] D. Thomas, A. Hunt, *Pragmatic Version Control using CVS*, 2004, p. 166.

[8] A. Hawley, *A Manual to the GNU Revision Control System (RCS)*, 2005 [tiešsaite]. – [atsauce 02.05.2008]. Pieejams: https://agave.garden.org/~aaronh/rcs/manual/html/index.html

[9] D. Price, *CVS for new users* [tiešsaiste]. – [atsauce 06.04.2008.] Pieejams: http://ximbiot.com/cvs/cvshome/new_users.html

[10] D. Neary, *Subversion – a better CVS*, 13.09.2005 [tiešsaiste]. – [atsauce 06.04.2008.]. Pieejams: http://www.linux.ie/articles/subversion/

[11] *Subversion (Software)* [tiešsaiste]. – [atsauce 06.04.2008.] Pieejams: http://en.wikipedia.org/wiki/Subversion_(software)

[12] M. Denny, *TFVC: Reducing Source file contention with branches*, 26.01.2006 [tiešsaiste]. – [atsauce 06.04.2008.] Pieejams: http://notgartner.wordpress.com/2006/01/26/tfvc-reducing-source-file-contention-with-branching/

[13] J. Meier, J. Taylor, P. Bansode, A. Mackman, K. Jones, *Chapter 5 – Defining your branching and merging strategy*, 09.2007 [tiešsaiste]. – [atsauce 19.04.2008.] Pieejams: http://msdn.microsoft.com/en-us/library/bb668955.aspx

[14] *Branching Strategies*, VSTS, 5.12.2007 [tiešsaiste]. – [atsauce 19.04.2008.] Pieejams: http://blog.nwcadence.com/2007/12/05/branching-strategies/

[15] M. Ruminer, Recomendations in SCM branching patterns in TFS, 07.09.2007 [tiešsaiste]. – [atsauce 21.04.2008.] Pieejams: http://manicprogrammer.com/cs/blogs/michaelruminer/archive/2006/09/07/88.aspx

[16] C. Walrad, D. Storm, *The importance of branching models in SCM*, 02.2002, p. 31-38.

PIELIKUMI

1. Pielikums

```
LIST_OF_BRANCHES=$@

if [ "$#" -lt "1" ] ; then
      cat <<-EOF
      Usage: $0 BRANCH...
           BRANCH - list of branches

           Script Steals lockg related to BRANCHES,
           merges branches into trunk and
           commits merged changes to repository.

      Example:
           $0 branches/task7 branches/request123
      EOF
      exit
fi

REPOS=http://cvs.konts.lv/repos/TRM/issuing3
#REPOS4SED=`echo $REPOS|sed 's/\//\\\\\\\//g'`

# Check if we are in local copy
LOCATION=`svn info|grep "^URL"|sed 's/URL: //'`
case $LOCATION in
      */trunk )
            # ok
            CURRENT_URL=$REPOS/trunk
            ;;
      */branches/za3_05_01 )
            # ok
            CURRENT_URL=$REPOS/branches/za3_05_01
            ;;
      */branches/za3_05_02 )
            # ok
            CURRENT_URL=$REPOS/branches/za3_05_02
            ;;
      *)
            echo "ERROR: $0 works in trunk, branches/za3_05_01,
branches/za3_05_02 only. You are in $LOCATION." >&2
            exit 1
            ;;
esac


if [ "x$EDITOR" = "x" ] ; then
      EDITOR=vi
      case `uname` in
            "CYGWIN"* )
                  EDITOR=notepad
                  ;;
      esac
fi


# Check if given branches exist
```

```
#branches/task7 r49, branches/task123 r90
LIST_OF_LAST_REV=
ERROR=
for i in $LIST_OF_BRANCHES ; do
      FROM_REVISION=`svn log -q --stop-on-copy --incremental $REPOS/$i|
tail -1| cut -d"|" -f1|sed 's/^r//'|sed 's/ //g'`
      LAST_CHANGED_REV_REPOS=`svn info $REPOS/$i|grep "^Last Changed
Rev:"|cut -d":" -f 2|awk '{print $1}'`
      if [ "x$LAST_CHANGED_REV_REPOS" = "x" ] ; then
            echo "ERROR: Branch $i doesn't exist" >&2
            ERROR=yes
      fi
      if [ "x$LIST_OF_LAST_REV" != "x" ] ; then
            LIST_OF_LAST_REV=$LIST_OF_LAST_REV", "
      fi
      LIST_OF_LAST_REV=$LIST_OF_LAST_REV$i" -r
"$FROM_REVISION":"$LAST_CHANGED_REV_REPOS
done
if [ "x$ERROR" != "x" ] ; then
      exit 1
fi
echo $LIST_OF_LAST_REV

# Root directory - Shell scripts and Makefiles my be changed. client
and server directories should be fresh.
# Check if the copy of client is fresh
LAST_CHANGED_REV_LOCAL=`svn info client |grep "^Last Changed Rev:"|cut
-d":" -f 2`
LAST_CHANGED_REV_REPOS=`svn info $CURRENT_URL/client |grep "^Last
Changed Rev:"|cut -d":" -f 2`
if [ $LAST_CHANGED_REV_LOCAL -ne $LAST_CHANGED_REV_REPOS ] ; then
      echo "ERROR: This is an old copy. Run \"svn update\" or \"svn
update client\"". >&2
      exit 1
fi

# Check if the copy of server is fresh
LAST_CHANGED_REV_LOCAL=`svn info server |grep "^Last Changed Rev:"|cut
-d":" -f 2`
LAST_CHANGED_REV_REPOS=`svn info $CURRENT_URL/server |grep "^Last
Changed Rev:"|cut -d":" -f 2`
if [ $LAST_CHANGED_REV_LOCAL -ne $LAST_CHANGED_REV_REPOS ] ; then
      echo "ERROR: This is an old copy. Run \"svn update\" or \"svn
update server\"". >&2
      exit 1
fi

# Find all lock connected to the list of branches given in the command
line
echo "Steal locks (y/n) ";read RESP
case $RESP in y | Y | yes | YES )
      OLDIFS=$IFS
      IFS='
      '
      echo "Stealing locks ..."
      for i in `svn status -u|grep "^.    O"|awk '{print
substr($0,21)}'|sed 's/\\\\/\\//g'`; do
```

```
            IFS=$OLDIFS
            LOCK_MESSAGE=`svn info $CURRENT_URL/$i | grep "^Locked by
.* in "`
            LOCK_PATH=`echo $LOCK_MESSAGE |awk '/^Locked by (\w+) in
/{print $5}'`
            if [ "x$LOCK_PATH" != "x" ] ; then
                  for j in $LIST_OF_BRANCHES ; do
                        STRIP_HEADING=`echo $j|sed 's/^\/*//'`
                        if [ "$LOCK_PATH" = "$STRIP_HEADING" ] ; then
                              echo  svn lock --force $i -m
"$LOCK_MESSAGE"
                              svn lock --force $i -m "$LOCK_MESSAGE"
                        fi
                  done
            fi
      done
      IFS=$OLDIFS
esac

echo "This is \"merge --dry-run ..\""
for i in $LIST_OF_BRANCHES ; do
      FROM_REVISION=`svn log -q --stop-on-copy --incremental $REPOS/$i|
tail -1| cut -d"|" -f1|sed 's/^r//'|sed 's/ //g'`
      echo "\"svn merge --dry-run -r $FROM_REVISION:HEAD $REPOS/$i\""
      svn merge  --dry-run -r $FROM_REVISION:HEAD $REPOS/$i
done

RESP=y
echo "Start merge (y/n) ";read RESP
case $RESP in y | Y | yes | YES )
      for i in $LIST_OF_BRANCHES ; do
            FROM_REVISION=`svn log -q --stop-on-copy --incremental
$REPOS/$i| tail -1| cut -d"|" -f1|sed 's/^r//'|sed 's/ //g'`
            echo "\"svn merge -r $FROM_REVISION:HEAD $REPOS/$i\""
            svn merge -r $FROM_REVISION:HEAD $REPOS/$i
      done
      ;;
      *)
      ;;
esac

RESP=y
echo "Merged without conflict - No \"C\" prefix in merge output (y/n)
";read RESP
case $RESP in y | Y | yes | YES )
      echo "Generating log message ..."
      echo "Merged from $LIST_OF_BRANCHES" >> merge_message.$$.tmp
      echo $LIST_OF_LAST_REV >> merge_message.$$.tmp
      echo "" >> merge_message.$$.tmp
      echo "--------8<--------- Lines from here will be removed -------
-8<----------">> merge_message.$$.tmp
      svn -q status >> merge_message.$$.tmp
      $EDITOR merge_message.$$.tmp
      awk  '/^--------8<--------- Lines from here will be removed -----
---8<----------$/{exit}{print}' <merge_message.$$.tmp >
merge_message2.$$.tmp && mv merge_message2.$$.tmp merge_message.$$.tmp
      echo "----------------------8<----------------------"
```

```
      cat merge_message.$$.tmp
      echo "----------------------8<----------------------"
      RESP=y
      echo "Commit changes with log message above (y/n)";
      echo "Attention: First line should be \"Merged from
$LIST_OF_BRANCHES\"";
      read RESP
      case $RESP in y | Y | yes | YES )
            svn ci -F merge_message.$$.tmp
            ;;
      esac
      rm -f merge_message.$$.tmp
      ;;
esac
```

Reģistrācijas lapa

Maģistra darbs _____

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

Autors: _____
(Autora paraksts)

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

Darba vadītājs(-ja): _____
(Vadītāja paraksts)

Darbs iesniegts Datorikas nodaļā _____.
(Iesniegšanas datums)
Ar šo es apliecinu, ka darba elektroniskā versija ir augšupielādēta LU informatīvajā sistēmā.

Metodiķe: _____.
(Metodiķes paraksts)

Recenzents: _____
(Recenzenta paraksts)

Darbs aizstāvēts maģistra gala pārbaudījuma komisijas sēdē

_____ prot. Nr. _____, vērtējums _____
(Darba aizstāvēšanas datums)

Komisijas sekretārs: _____
(Sekretāra paraksts)