RĪGAS TEHNISKĀ UNIVERSITĀTE

Datorzinātnes, informācijas tehnoloģijas un enerģētikas fakultāte

Lietišķo datorsistēmu institūts

Gļebs Bondarevs

bakalaura akadēmisko studiju programmas "Datorsistēmas"

students, stud. apl. nr. 211RDB375

Automatizētas dokumentācijas ģenerēšanas ietekmes izpēte uz izstrādes procesu

BAKALAURA DARBS

Zinātniskais vadītājs: Dr.sc.ing., as. profesors

Aleksejs Jurenoks

RĪGA 2024

RĪGAS TEHNISKĀ UNIVERSITĀTE

DATORZINĀTNES, INFORMĀCIJAS TEHNOLOĢIJAS UN ENERĢĒTIKAS FAKULTĀTE

12300 Lietišķo datorsistēmu institūts

Bakalaura darba izpildes lapa

Noslēguma darba autors:

students(-e) Gļebs Bondarevs _____

(paraksts, datums)

Noslēguma darbs ieteikts aizstāvēšanai:

Zinātniskais vadītājs:

Dr.sc.ing. asoc. profesors Aleksejs Jurenoks _____

(paraksts, datums)

ANOTĀCIJA

Bakalaura darba 2. tips: Aktuālo jomas problēmu risinājumi

Atslēgvārdi: Dokumentācija, Automatizācija, Ietekmes izpēte, Izstrādes process.

Mūsdienās, kad tehnoloģijas attīstās ļoti strauji, kad programmatūras izstrādes process kļūst arvien īsāks un programmatūras prasības mainās ļoti ātri, efektīvi un galvenokārt precīzi izstrādāta dokumentācija ir svarīgs faktors kas var ietekmēt uz izstrādes procesu pozitīva veida. Automatizēta dokumentācijas ģenerēšanas risinājumu izmantošana piedāvā iespēju veidot dokumentāciju arvien ātrāk un efektīvāk, salīdzinot ar dokumentāciju, veidotu manuāla veida.

Bakalaura darbs ir veltīts pētījumam par automatizētas dokumentācijas ģenerēšanas ietekmi uz izstrādes procesu. Tajā tiek pētīts, kas ir programmatūras izstrādes process, kas ir dokumentācija, kādas ir tās priekšrocības un trūkumi un kāda ir dokumentācijas loma izstrādes procesā. Tiek aplūkota dokumentācijas funkcionalitāte, programmētāju un atslēgvārdu nozīmē dokumentācijai, apskatīti esošie dokumentācijas veidi un aprakstīti pieejamie dokumentācijas rīki. Bakalaura darba gaitā tiek izstrādāts algoritms automātiskai dokumentācijas izveidei, kas tiek izmantots reālā izstrādes procesā, lai izpētītu un noskaidrotu, kāda ir tā ietekme uz izstrādes procesu. Ietekmes uz izstrādes procesu izpēte balstās uz diviem kritērijiem: dokumentācijas izveides ātrumu un tās kvalitāti. Veicot eksperimentus, kuros tika salīdzināts dokumentācijas izveides process un tās kvalitāte manuālā un automātiskā režīmā, tika secināts, ka automātiskais dokumentācijas izveides risinājums, pirmkārt, ietekmē un uzlabo izstrādes procesu, proti, tas paātrina dokumentācijas izveides un atjaunināšanas procesu un atbrīvo izstrādātājus no nevajadzīga darba. Otrkārt, tas ietekmē dokumentācijas saturam, padarot to strukturētu, kvalitatīvu, efektīvu, kā ari samazinot kļūdu iespējamību.

Darba pamattekstā ir 73 lappuses, 12 attēli, 11 tabulas, 6 pielikumi un 18 izmantotie informācijas avoti.

ABSTRACT

Type 2: Solutions to topical problems

Keywords: Automation, Documentation, Impact study, Development process

Nowadays, when technology is developing very fast, when the software development process is getting shorter and software requirements are changing very fast, efficient and above all accurate documentation is an important factor that can have a positive impact on the development process. The use of automated documentation generation solutions offers the possibility to create documentation in an increasingly faster and more efficient way compared to manual documentation.

The bachelor thesis is dedicated to the study of the impact of automated documentation generation on the development process. It investigates what is the software development process, what is documentation, what are its advantages and disadvantages and what is the role of documentation in the development process. It discusses the functionality of documentation, the meaning of programmers and keywords for documentation, reviews existing documentation types and describes available documentation tools. During the bachelor thesis, an algorithm for automatic documentation creation is developed and used in a real development process in order to investigate and clarify its impact on the development process. The study of the impact on the development process is based on two criteria: the speed of documentation creation and its quality. Experiments comparing the documentation creation process and its quality in manual and automatic mode showed that the automatic documentation creation solution firstly influences and improves the development process, i.e. it speeds up the documentation creation and updating process and frees developers from unnecessary work. Secondly, it has an impact on the content of the documentation, making it structured, high quality, efficient and reducing the likelihood of errors.

The main body of the paper consists of 73 pages, 12 figures, 11 tables, 6 annexes and 18 sources of information.

SATURA RĀDĪTĀJS

IEVADS

Ir zināms, kā mūsdienās visu tehnoloģiju pasaulē strauji attīstās, un dažas tehnoloģijas jau sasniedz savu maksimumu. Tapāt pieaug vajadzība pēc dažādu rīku vai tehnoloģijas automatizāciju, lai paātrinātu darbību un attīstību. Viena no šādiem rīkiem ir automatizētā dokumentācijas ģenerēšana, kuras uzdevums ir automātiski izveidot, tehnisko dokumentāciju, lietošanas instrukcijas dokumentāciju un citu dokumentāciju veidu dažādas darba jomas.

Runājot tieši par IT, dokumentācijai ir svarīga loma gan projektēšanas, gan izstrādes, gan uzturēšanas fāzes. IT uzņēmumos dokumentāciju izmanto ne tikai izstrādājamā produkta koda un citas informācijas dokumentēšanai, bet arī komunikācijai starp komandām un starp dažādiem izstrādes posmiem. Tāpēc uzņēmumiem ir ļoti svarīga dokumentu ģenerēšanas automatizācija, pirmkārt, lai ietaupītu laiku un attīstītu tās efektivitāti. Otrkārt, izmantojot automatizētu dokumentācijas ģenerēšanu, programmētājs var ietaupīt daudz laika uz produktu izstrādi. Un treškārt, izstrādātajam, izmantojot automatizētu dokumentu ģenerēšanu, ir arī iespēja viegli un ļoti ātri veikt rediģēšanas darbības, kas pilnībā atspoguļos visas izmaiņas produktā, kodā un projektā kopumā.

Šajā pētījumā tiks pētīta un aprakstīta automātiskās dokumentācijas ģenerēšanas ietekme uz izstrādes procesu, kas palīdzēs noteikt dokumentācijas nozīmi izstrādes procesā. Pētījuma laikā tiek noteikti vērtēšanas kritēriji veiktā darba izvērtēšanai un izstrādāta dokumentācijas automātiskās ģenerēšanas izstrādes algoritms.

Pirmajā nodaļā ir veikta cita autora pētījuma izpēte un analīze, lai noskaidrotu, kas ir dokumentācija, izstrādes process un kāda ir dokumentācijas loma izstrādes procesā. Turklāt tiek aprakstīti esošie risinājumi un tehnoloģijas dokumentācijas veidošanai, noskaidrotas tās funkcijas, kā arī identificētas tehnoloģijas priekšrocības un trūkumi.

Otrajā nodaļā ir izstrādāts algoritms automātiskai dokumentācijas ģenerēšanai, kas tiks izmantots, lai izpētītu un noskaidrotu šā risinājuma ietekmi uz izstrādes procesu. Turklāt nodaļās ietvaros sīki aprakstīts algoritma izstrādes process un veikta algoritma demonstrācija.

Trešajā nodaļā tiek veikta izstrādātā dokumentācijas ģenerēšanas risinājuma praktiska validācija, lai noteiktu tā ietekmi uz izstrādes procesu. Turklāt, pamatojoties

uz divos eksperimentos iegūtajiem rezultātiem, tiek salīdzinātas manuālā un automātiskā dokumentācijas ģenerēšanas metode, kā arī šīs dokumentācijas saturs, lai noteiktu ietekmi uz izstrādes procesu un noskaidrotu, vai šo risinājumu ir ieteicams izmantot izstrādes procesam.

Pētījuma mērķis

Pētījuma mērķis ir izpētīt un analizēt automatizētas dokumentācijas ģenerēšanas ietekmi uz programmatūras izstrādes procesu, izmantojot izstrādāto algoritmu, lai noteiktu priekšrocības un trūkumus, kas saistīti ar šīs tehnoloģijas izmantošanu, īpaši uzsverot tās ātrumu, efektivitāti un kvalitāti.

Pētījuma uzdevumi:

1. Izpētīt esošos risinājumus un tehnoloģijas, kas paredzētas dokumentācijas ģenerēšanai, un identificēt to funkcijas, pielietojuma jomas un priekšrocības izstrādes procesā.

2. Veikt literatūras apskatu par dokumentācijas nozīmi izstrādes procesā un to, kā automatizētā dokumentācija varētu uzlabot efektivitāti un kvalitāti.

3. Izstrādāt automātiskās dokumentācijas ģenerācijas algoritmu un pielāgot to konkrētai izstrādes vides vajadzībām.

4. Definēt programmatūras dokumentācijas ietekmes salīdzināšanas kritērijus, kas nosaka, kā izmaiņas dokumentācijas izveidošanas procesā ietekmē izstrādes procesa efektivitāti un kvalitāti.

5. Veikt praktisko validāciju, pielietojot izvēlēto dokumentācijas ģenerēšanas risinājumu reālajā izstrādes projektā.

6. Salīdzināt rezultātus pirms un pēc automatizētas dokumentācijas ģenerēšanas risinājuma pielietošanas, izmantojot kvantitatīvus rādītājus, lai noteiktu ietekmi uz izstrādes procesu.

Pamatjēdzieni un definīcijas

Programmatūras koda dokumentācija – ir dokumentācijas veids, kas dokumentē informāciju par programmas kodu, tā komentāriem, funkcijām, klasēm, koda struktūru, atribūtiem, izmantotajām bibliotēkām un citu noderīgu informāciju. Tā ir nepieciešama, lai padarītu to vienkāršāku, ātrāk saprotamu un lietojamu.

Produkta un programmatūras izstrādes dokumentācija – tas ir pilnīgi divi dažādi dokumentācijas veidi. Produkta dokumentācijas ietver sevi ražošanas procesu aprakstu, dizainu, testēšanu, drošības informāciju, kā ari finanšu informācija. Bet ne koda informācija. Savukārt programmatūras dokumentācijā ietver sevi koda, algoritmu, funkciju, metodes un struktūras informāciju.

Automatizēta dokumentācija – tas ir process, kuru izmanto lai veiktu automātisko dokumentāciju par kodu, produktu, sistēmu, izstrādes fāzes un programmatūru. Automatizēta dokumentācija tiek veikta izmantojot dažādus rīkus un programmatūras.

Izstrādes process – tas ir darbību kopums kurš ir tiek veikts lai izveidotu kaut kādu jaunu tehnoloģiju, programmatūru vai produktu. Izstrādes process sastāv no dažādiem posmiem vai fāzēm.

Izstrādes efektivitāte – ir svarīgs kritērijs, pēc kuras var novērtēt izstrādes procesu un secināt kas ir vajadzīgs priekš veiksmīgā projekta pabeigšanai un mērķa sasniegšanai.

Dokumentācijas precizitāte – ir svarīgs vērtēšanas kritērijs kura ir nepieciešamā lai nodrošinātu pareizu dokumentācijas uzrakstīšanu.

Informācijas tehnoloģijas (IT) – ir darbības joma, kas ietver sevi programmatūras, produkta, sistēmas izstrāde, uzturēšana un citas darbības. Mērķis šādai jomai ir efektīvi izmantot tehnoloģiju un tas informāciju.

1. AUTOMATIZĒTA DOKUMENTĀCIJAS ĢENEREŠANA

Tiek pieradīts, ka dokumentācijai mūsdienās ir svarīga nozīme visdažādākajās darba jomās, kas palīdz gan uzņēmumiem , gan tas darbiniekam. IT ir viena no šādām jomām. IT jomā dokumentācija tiek izmantota visos izstrādes posmos, lai nodrošinātu, ka izstrādes process ir drošs, efektīvs un ātrs. Sistēmas vai programmatūras izstrādes procesā izmantotās dokumentācijas piemēri ir PPS dokumentācija, tehniskā dokumentācija, lietotāja rokasgrāmatas dokumentācija, API dokumentācija, klientu rokasgrāmata un dokumentācija programmatūras koda komentēšanai.

1.1. Nodaļas ievads

Šajā sadaļā aprakstīta literatūras meklēšanas procedūra, izmantotie atslēgvārdi un zinātnisko rakstu atlases metodes.

1.1.1. Literatūras meklēšana un tas atlases metode

Analītiskā daļa ietver dažādu zinātnisko rakstu izpēti, analīzi un aprakstu. Veiksmīgam analītiskajam darbam ir nepieciešams atsaukties uz zinātniskiem avotiem, kas palīdz noskaidrot, kas ir automātiskās dokumentācijas izveides process un kādu ietekmi tas var atstāt uz izstrādes procesu.

Zinātnisko rakstu meklēšana un piekļuve tiem tika nodrošināta ar Rīgas Tehniskās universitātes bibliotēkas elektronisko resursu palīdzību. Meklēšanas procesā tika izmantoti tādi elektroniskie resursi kā IEEE Xplore Digital Library, SpringerLink, Web of Science, ScienceDirect, ACM Digital Library un WILEY Online Library.

Zinātnisko rakstu meklēšanas procedūrai tika izmantoti atslēgvārdi, kuri palīdzes atrast nepieciešamos avotus saistīta ar darba tēmu, kā ari atvieglos tas meklēšanas procesu. Tā kā bakalaura darba tēma ir saistīta ar automatizētas dokumentācijas ģenerēšanu un tās ietekmi uz izstrādes procesu, tika izvēlēti pamata atslēgvārdi. Par pamata atslēgvārdiem tika ņemti šādi vārdi: dokumentācija, izstrādes process, kods, pirmkoda dokumentācija, dokumentācijas ģenerēšana, programmatūra, automatizētā dokumentācija un programmatūras dzīves cikls. Atslēgvārdi tika izmantoti meklēšanai dažādās valodās, un dažkārt tie tika arī saistīti kopā, lai atrastu visatbilstošākos rakstus.

Izmantojot elektroniskus resursus un izmantojot atslēgvārdu zinātnisko rakstu meklēšanai bija izvelēti 18 dažādie zinātniskie raksti, kas ir saistīti ar darba tēmu. Rakstu izvēle bija saistīta ar citātu skaitu, žurnāla indeksu un autora pieejamību, kā arī autora vai publicista reputāciju. Dažiem publicētiem rakstām bija kā daļā no liela zinātniskā raksta, kā ari teksta fragmenti no uzticamam grāmatām. Kā piemēru var minēt izlasītu zinātnisku rakstu ar nosakāmu "Automatic Documentation Generation via Source Code Summarization" [7], kas bija paņemts no "IEEE International Conference on Software Engineering". Izvēle bija atkarīga ne tikai no resursa indeksiem, citātu skaitu un autora pieejamību, bet arī no raksta konteksta, kas ir svarīgs punkts pareizai zinātnisko rakstu izvēlei.

1.2. Dokumentācijas un izstrādes procesa izpēte

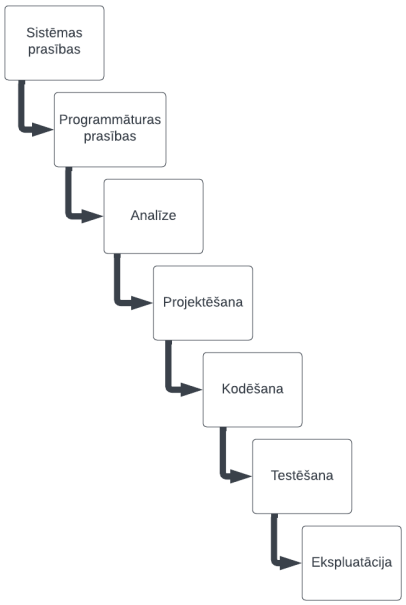
Šajā sadaļā ir detalizēti izpētīti pētnieciskie raksti, lai noteiktu, kas ir dokumentācija, izstrādes process, kāda dokumentācijas ietekmi ir uz izstrādes procesu, kā arī lai noteiktu šāda risinājuma funkcionalitāti, priekšrocības un trūkumus.

1.2.1. Programmatūras izstrādes procesa un dzīves cikla pārskats

Pirms doties uz izpēte, kas ir dokumentācija un kāda ir tās loma, ir svarīgi uzzināt vairāk par programmatūras izstrādes procesu un tā dzīves ciklu, lai būtu skaidrs, kāpēc dokumentācijai ir svarīga loma izstrādes procesā. Šajā sadaļā ir apskatīti un aprakstīti pētnieciskie raksti, kas sniedz informāciju par izstrādes procesa fāzēm un dzīves cikliem, kā arī izstrādes procesa metodēm..

Pamatojoties uz resursu par programmatūras izstrādes procesu [1], tas ir soļu un darbību secība, ko veic programmētājs, lai veiksmīgi izstrādātu programmatūru. Atsaucoties uz pētījuma rakstu [1], programmatūras izstrādes procesa posmi ietver plānošanu, dizaina izstrādi, programmatūras ražošanu, testēšanu un uzturēšanu. Līdz šim ir pierādīts, ka pastāv daudzas produktu izstrādes metodes. Katrs uzņēmums izvēlas piemērotāko izstrādes metožu secību, lai izveidotu programmatūru. Arī, kā secina otrā pētījuma [2] autors, uzņēmumi pielāgo izstrādes dzīves ciklu projekta vajadzībām. Tomēr ir kāda klasiska izstrādes metode, kas tika ieviesta jau ļoti sen un kļuva par pamatu citām pastāvošajām metodēm. Klasiskā metode sastāv no: prasību definēšanas, kas savukārt sastāv no sistēmas prasību definēšanas un programmatūras prasību definēšanas. Pēc prasību definēšanas seko analīze, programmatūras projektēšana un pēc

tam kodēšana. Izstrādes pēdējā posmā ietilpst izstrādātās programmatūras testēšana un ekspluatācija.



1.1. att. Klasiska izstrādes procesa metode [3]

Turklāt, ja runājam tieši par dzīves ciklu, tam ir izstrādāts standarts, kas minēts divos zinātniskajos darbos [2],[4] - ISO/IEC 12207. Vienā no izlasītajiem zinātniskajiem avotiem autors ir uzsvēris izstrādes procesa un tā dokumentēšanas nozīmi. Autors secina, ka daudziem uzņēmumiem ir ļoti svarīgi izstrādāt augstas kvalitātes programmatūru, bet, lai to panāktu, izstrādes procesā ir jābūt labai dokumentācijai [5]. Ar to autors vēlējās uzsvērt, ka dokumentācijai ir liela nozīme mūsdienu izstrādes procesā un ar dokumentācijas palīdzību ir iespējams iegūt augstas kvalitātes produktu. Darbā [3] autors parāda, kā ar dokumentācijas palīdzību var samazināt laiku no izstrādes sākuma līdz programmatūras nodošanai. Tās nozīme, ka kvalitatīvas dokumentācijas izveide, kas satur svarīgu informāciju gan par izstrādes procesu, gan tieši par programmatūras kodu, var ievērojami samazināt izstrādes laiku un paātrināt izstrādes procesu.

Tā kā ir daudz izstrādes metožu un nav skaidrs, kuru metodi uzņēmums izmanto savā izstrādes procesā, nav iespējams teorētiski un precīzi noteikt, cik ilgs ir katrs izstrādes posms un kā automatizētā dokumentācija var uzlabot un paātrināt tā izpildi. Tāpēc turpmāk darbs tiek veikts, lai noskaidrotu, kā dokumentācija ietekmē izstrādes procesu. Šajā nodaļā skaidrots, ka izstrādes procesā ir dažādi izstrādes veidi, kurus uzņēmumi izmanto pēc saviem ieskatiem. Katrs izstrādes veids sastāv no dažādām

izstrādes procesa daļām, kuru izpildes laiks ir atkarīgs no izvēlētās metodes un uzņēmuma vajadzībām. Šajā iedaļā arī izskaidrots, cik daudz dokumentācijas ir nepieciešams izstrādes procesā un kā to ietekmē laba dokumentācija.

1.2.2. Programmatūras dokumentācijas nozīme izstrādes procesā izpēte

Kad izstrādes procesa un tā dzīves cikla pamati ir noskaidroti, ir pamats turpināt pētīt programmatūras dokumentācijas lomu izstrādes procesā. Šāda informācija būs noderīga, lai saprastu, kā dokumentācija teorētiski ietekmē izstrādes procesam, un lai noskaidrotu, ka laba vai slikta dokumentācija var ietekmēt produkta izstrādi un piegādi.

Atsaucoties uz [6] publikāciju, dokumentācijas rakstīšanas process ir viens no svarīgākajiem programmatūras dzīves cikla posmiem. Tās rakstīšana, sagatavošana un uzturēšana ir dārga, un laika gaitā tā bieži noveco. [7]. Dokumentācija laika gaitā noveco, tāpēc ka programmatūra mainās neatkarīgi no izstrādes vai uzturēšanas laika [8],[9]. Dokumentācija ir ideāls papildinājums jebkurai programmatūras sistēmai, lai ieinteresētās puses varētu piekļūt noderīgai informācijai par sistēmu un saistītajām funkcijām [10]. Kā autors atzīmējis pētījumā, dokumentācijas rakstīšana ir sarežģīts uzdevums [7]. Tas ir tāpēc, ka projekta programmētājiem ir jāsniedz zināšanas citiem projektā iesaistītajiem programmētājiem, jāveicina sistēmas attīstība un jāuztur tās dokumentācija. Šī iemesla dēļ pēdējā laikā arvien biežāk tiek mēģināts izveidot standartizētu automātisku dokumentācijas ģeneratoru [8]. Bez automātiskā dokumentācijas ģeneratora lietotājiem vai izstrādātājam ir jāizmanto dažādi palīg rīki, lai iegūtu nepieciešamo inženiertehnisko informāciju un izveidotu labu dokumentāciju [11]. Šādu utilītu vai citu sistēmu izmantošana var novest pie tā, ka dažās dokumentu publicēšanas sistēmās izstrādātājiem ir jāsagatavo dokuments, izmantojot izvēlnes, lai no krātuvēm varētu iegūt nelielus inženiera informācijas avotus vai fragmentus [11].

Lai saprastu, kas ir dokumentu automatizācija, vispirms ir jāzina, kas ir slikta un laba dokumentācija. Atsaucoties uz publikāciju [9] par koda dokumentācijas automatizāciju, varam secināt, ka laba dokumentācija ir tā, kas ir pareizi uzrakstīta, ievērot visus standartus, protokolus un metodes. Tas ir ļoti svarīgi efektīvai un ātrai programmatūras izstrādei. Turklāt, balstoties uz citiem pētījumiem [6], varam iegūt informāciju par to, ka laba dokumentācija var arī samazināt projekta izmaksas. No otras puses, runājot par sliktu programmatūras dokumentāciju un atsaucoties uz pētnieciskajiem darbiem [7], var secināt, ka slikta dokumentācija izraisa strauju

programmatūras sistēmas kvalitātes un lietojamības samazināšanu. Slikta dokumentācija ļoti apgrūtina zināšanu nodošanu, kas ir slikti programmatūras uzturēšanai, kā arī rada nepieciešamību mācīties jauniem komandas locekļiem.

Dokumentācija ir svarīga izstrādes procesa daļa, taču tās manuāla izveide ir dārga, sarežģīta un bieži vien izstrādes procesa laikā noveco, tāpēc ir liela interese par procesa automatizāciju. Labi vai slikti uzrakstītai dokumentācijai ir liela ietekme uz izstrādes procesa daļām un procesa dalībniekiem.

1.2.3. Dokumentācijas pamatprincipu un iespējamu trūkumu izpēte

Šajā sadaļā mēs aplūkosim, kas būtu jāņem vērā, izmantojot automatizētu dokumentācijas ģeneratoru, un kādi tam var būt trūkumi. Šī informācija ir svarīga, lai saprastu, kādam vajadzētu izskatīties automatizācijas rīkam un kam jāpievērš uzmanība.

Kā bija minēts iepriekš un ir aprakstīts zinātniskā raksta [7], programmatūras dokumentācijā nodrošina izstrādes procesu drošu, efektīvu un ātru, kā ari dokumentācijā pastāv, lai palīdzētu programmatūras projektētajiem, izstrādātajiem un uzturētājiem izprast sistēmu un tās gan galvenos, gan sekundāros procesus. Taču procesu automatizācijas ziņā, lai nodrošinātu dokumentācijas izstrādes automatizāciju, ir jāizmanto noteikta veida izstrādes metodes, rīki, kas īsteno šo metodi, un šīs dokumentācijas standartus [11]. Kuru cilvēks dažkārt var palaist garām neuzmanības vai aizmāršības dēļ dokumentācijas sagatavošanas laikā. To var uzskatīt par pirmo noderīgo trūkumu, izmantojot automātisko ģeneratoru dokumentācijas izstrādei. Izstrādājot sistēmu dokumentācijas izveidošanai, tai jābūt drošai, kvalitatīvai un efektīvai, lai ikvienam izstrādē iesaistītajam būtu pilnīga izpratne par izstrādājamo sistēmu un tās funkcionalitāti. Lai efektīvi automatizētu dokumentu izstrādi, ir jāievēro stingrs un detalizēts izstrādes process [11]. Protams, runājot par automatizācijas procesu, nedrīkst aizmirst, ka labu un efektīvu dokumentu automatizāciju nevar izveidot bez noteiktas informācijas.

Nodrošinājāt dokumentu ģenerēšanu drošu, elastīgo, bagātīgu un galvenais automatizēto, var gan izvairīties no dažādam darba pienākumiem programmatūras izstrādātajiem, gan atvieglot izstrādes procesu, gan paātrināt to. Kā ari uzņēmumi var iegūs vairāk pozitīvas ieguvumus izmantojot šādu tehnoloģiju.

Šajā sadaļa dots mazs ieskats par noderīgiem trūkumiem automātiskas ģeneratora izmantošanai. Lai to nodrošinātu, ir jāizmanto noteiktas izstrādes metodes un rīki. Šādai sistēmai jābūt drošai, kvalitatīvai un efektīvai, un bez noteiktas informācijas nav iespējams izstrādāt labu un efektīvu dokumentu automatizācijas rīku.

1.3. Automātiskās dokumentācijas funkcionalitāte un pielāgojamība izpēte

Šī sadaļa ir turpinājums iepriekšējai sadaļai, kurā sīkāk aprakstīts, kas nepieciešams automātiskajam dokumentācijas izveides rīkam, lai lietotāji varētu izveidot dokumentāciju bez iepriekšējā sadaļā minētajiem trūkumiem. Šīs sadaļas nozīme ir tāda, lai turpmākajā darbā būtu skaidrs, kas jāparedz rīka izstrādei.

Elastīgums ir nepieciešams, lai risinātu vienu no galvenajām iepriekš aprakstītajām manuālās dokumentācijas problēmām. Automātiskajam dokumentu ģeneratoram jānodrošina bagātīgs un elastīgs utilītu kopums, ko lietotāji var izmantot, lai no katra izstrādes rīka iegūtu vislabāko pielāgošanas informāciju [11]. Tas ir nepieciešams, lai vienlaicīgi iegūtu informāciju no vairākām dažādām rīku krātuvēm, ģenerētu dokumentus, piemēram, populārām dokumentu publicēšanas sistēmām un vienlaicīgi ģenerētu dažādus dokumentus [11]. Turpinot diskusiju par automatizētu dokumentu ģeneratoru, [11] zinātniskā raksts norāda, ka šim automatizētajam ģeneratoram ir jāizveido pazīstama, konsekventa lietotāja saskarne. Tas jādara, lai to būtu viegli un vienkārši lietot, bez liekām neuzmanības darbībām, kas var radīt problēmas un nevajadzīgas darbības vai atbildību sistēmas izstrādātājiem. Turklāt dokumentu ģeneratoram jānodrošina dokumentu veidnes daudziem valdības, komerciālajiem un starptautiskajiem standartiem [11]. Dažādu līmeņu standartu izveide un ievērošana, rakstot dokumentāciju manuāli, izstrādātājiem var prasīt divas līdz trīs reizes vairāk laika un pūļu salīdzinājumā ar automātisku dokumentācijas ģenerēšanu, ja dokumentu ģeneratorā jau ir pieejami šo standartu veidnes.

Šīs sadaļas beigās varat apkopot par to, kā šajā sadaļā aprakstīta automātiskā dokumentācija, kas nepieciešama, lai izstrādātu labu un piemērotu rīku visam izstrādes posmam. Kā arī sniegt zināšanas par to, kas tieši ir nepieciešams automātiskai dokumentācijai. Izpētot pētnieciskos rakstus, var secināt, ka ir nepieciešams turpināt praktiski pētīt, kas vēl ir nepieciešams, lai izveidotu labu dokumentāciju.

1.4. Programmētāju lomas izpēte uz izstrādes un dokumentēšanas procesā.

Tā kā iepriekšējās sadaļās ir aprakstīta informācija par izstrādes procesu un dokumentāciju, ir svarīgi izpētīt informāciju par programmētājiem, kas ir neatņemama gan izstrādes procesa, gan dokumentācijas izveides un lasīšanas daļa. Šajā sadaļā ir apskatīti zinātniskie avoti un aprakstīts, kā tieši programmētāji ietekmē izstrādes procesu un kā viņi ietekmē dokumentēšanas procesu.

Kopumā izstrādes procesa dokumentācija ir tieši nepieciešama programmētājiem. Kā minēts iepriekš, sistēmu izstrādātāji izmanto dokumentāciju, lai izprastu kodu, kā arī lai uzlabotu sistēmu, izmantojot dokumentāciju [10]. Visa informācija, ko satur tehniskā dokumentācija, ir kods, koda apraksts, algoritmu apraksts, informācija par programmatūras dizainu, saskarnēm un funkcionalitāti [10] [8]. Diemžēl daudziem programmētājiem nav laika un vēlēšanās izveidot un uzturēt dokumentāciju [9]. Turklāt dokumentēšanas process programmētājiem bieži vien ir laikietilpīgs[6]. Kā norādīta viena no pētījumā, "Programmētāji paļaujas uz pirmkoda dokumentāciju, lai ātri saprastu, ko un kā to dara viņu izmantotais kods"[8].

Paļaujoties uz zinātnisko rakstu [10], var secināt, ka programmētāji ir atbildīgi ne tikai par tehniskās dokumentācijas izstrādi, ko viņi izmanto izstrādes laikā, bet arī par lietotāju rokasgrāmatu izveidi. Papildus tam, ka tiek izmantota informācija no viena informācijas avota [10], rokasgrāmatu dokumentācija ir nepieciešama sistēmas vai programmatūras lietotājiem, lai izskaidrotu, kā lietotājiem būtu jāsadarbojas ar sistēmu. Dokumentā ir iekļauta informācija par to, kā darbojas kāda funkcija, soļu secība, kas jāievēro, lai iegūtu vēlamo rezultātu, un tehniskā informācija par sistēmu, kas var būt noderīga, lietojot programmatūru.

Tomēr, tā kā dokumentācija ir ļoti nepieciešama izstrādes procesam un tā izstrādātājiem, tās izveidei ir nepieciešams daudz laika un pūļu. Lai vismaz daļēji atrisinātu šo problēmu, ir ierosinātas dažādas pieejas un rīki, kas sīkāk tiks aprakstīti nākamajā sadaļā, lai palīdzētu izstrādātājiem dokumentēt programmatūru. Tomēr pilnībā automātiskai dokumentācijas izveidei bez cilvēka palīdzības ir nepieciešama dziļa izpratne par programmatūru, ko var iegūt tieši no programmētājiem, un testēšanas laika [10].

Programmētāji ir ļoti atkarīgi no dokumentācijas izstrādes procesa, tāpēc tas ir jāautomatizē, lai programmētāji varētu izmantot savas spējas citos izstrādes posmos.

Kā jau minēts, dokumentācijas rakstīšana nav ļoti sarežģīts programmatūras izstrādes posms, taču tas prasa daudz laika un cilvēkresursu. Nobeigumā jāsecina, ka dokumentācija ir nepieciešama un svarīga ne tikai programmētājiem, bet tā tiek izstrādāta arī cilvēkiem, kas sistēmu izmanto katru dienu. Turpmākajā darbā nepieciešams sīkāk izpētīt, kā slikta un laba dokumentācija ietekmē programmētajiem.

1.5. Dokumentācijas nozīme nepārtrauktā programmatūras izstrādēs izpēte

Lai labāk izprastu, kā darbojas dokumentācija, šajā sadaļā sniegts piemērs par programmatūras izstrādi, izmantojot nepārtrauktas izstrādes pieeju. Tiek piedāvāts paplašināts informācijas kopums, lai parādītu, ka dokumentācijas izveide un izmantošana ir svarīga dažādos izstrādes posmos un dažādām metodēm, kā arī parādītu, kādi citi dokumentācijas veidi tiek izmantoti.

Viens no programmatūras izstrādes paņēmieniem - nepārtraukta programmatūras izstrāde - ir aprakstīts dažādos pētījumos. Pamatojoties uz tiem, var secināt, ka šī tehnika nodrošina efektīvu un ātru programmatūras izstrādi un tās piegādi lietotājiem [12]. Tā nodrošina arī ļoti biežu koda modificēšanu, testēšanu un dokumentēšanu [2]. Nepārtrauktā programmatūras izstrādē (CSD) bieži izmanto neformālu dokumentāciju [13]. Šādu dokumentāciju parasti ir grūti saprast un uzlabot, ja tā ir rakstīta citam izstrādātājam. Šāda dokumentācija ietver, piemēram, skices un citus rasējumus plānošanas laikā [13]. Nepārtrauktā programmatūras izstrādē, ja dokumentācija tiek rakstīta manuāli, tā parasti tiek veidota tikai tad, kad tā ir nepieciešama galaprodukta izveidei [13]. Tas ir tāpēc, ka dokumentācijas rakstīšana manuāli aizņem daudz laika, tāpēc kļūst grūti izveidot dokumentu, ja kods tiek bieži mainīts. Šādas galaprodukta dokumentācijas piemērs ir lietotāja rokasgrāmata, kas aprakstīta vienā no zinātniskiem rakstiem [10]. No otras puses, runājot par šādu dokumentāciju, pētījumā [13] ir aprakstīts, kā vienā dokumentā tiek apspriests par to, ka skices, diagrammas un citi nepieciešamie vizuālie dati netiek saglabāti CSD dokumentā. Tas ir tāpēc, ka šāda izstrādes metode mēdz izdzēst programmatūru, tāpēc nav jēgas dokumentēt kaut ko, kas drīz atkal tiks mainīts. Pēc [13] autora domām, nepārtrauktai programmatūras izstrādei nav jēgas izmantot pirmkoda manuālu dokumentēšanu. Tas ir tāpēc, ka programmatūras kods tiek atjaunināts ļoti bieži un to

ir grūti dokumentēt manuāli, bet automatizācijas rīku izmantošana ļaus dokumentēšanu veikt vismaz biežāk.

Kopumā var secināt, ka ne visas programmatūras izstrādes tehnoloģijas ir piemērotas manuālai dokumentēšanai, jo šādos apstākļos rodas daudz grūtību ar tās izveidi. CSD ar dzīvu piemēru parāda, ka dokumentācijai ir svarīga loma izstrādē, un parāda, ka šādos apstākļos ir svarīgi izmantot automātiskus rīkus, kas to paātrina. Tomēr vēl ir nepieciešams padziļināti izpētīt, kā automātiskā dokumentācija ietekmē izstrādes procesu no praktiskā viedokļa.

1.6. Pirmkoda dokumentācijas izpēte

Dokumentāciju programmatūras izstrādes procesā izmanto dažādos izstrādes posmos, un dažādi dokumentācijas veidi ir atkarīgi no prasībām un darījuma mērķa. Šī sadaļa ir nepieciešama, lai noskaidrotu, kas tā ir, un detalizēti apspriestu vienu izstrādes procesā bieži izmantoto dokumentācijas veidu.

Avota koda dokumentācijā ir svarīgi ierakstīt visu nepieciešamo informāciju par kodu un tā komentārus, lai programmētāji to varētu izmantot izstrādes procesā. Programmētāji parasti vairāk laika pavada, lasot kodu, nekā to rakstot. Avota koda dokumentācija šobrīd ir svarīgs process programmatūras projektu atbalstam un vadībai, secināts vienā no resursiem [6]. Kā minēts iepriekš, programmatūras dokumentācija pastāv, lai palīdzētu izstrādātājiem un lietotājiem saprast sistēmu un tās procesus, tas ir minēts arī divos citos pētījumos [7], [10]. Šādai pirmkoda dokumentācijai vienmēr jābūt kodolīgai, skaidrai un labi strukturētai, lai esošie programmētāji varētu pilnībā saprast tās saturu. Tomēr mūsdienu prakse rāda, ka izstrādātājiem un uzņēmumiem ir dārgi veidot pirmkoda dokumentāciju manuāli, kā rezultātā dokumentācija bieži vien ir nepilnīga un nekvalitatīva [7].

Taču šī pirmkoda dokumentācija ir svarīgs artefakts efektīvai programmatūras izstrādei[9]. Protams, ņemot vērā mūsdienu automatizācijas tendenci, arī pirmkoda dokumentācija ir jāatjaunina. Avota koda dokumentācijas automatizēšana var būt ļoti noderīga, jo dokumentācijas rakstīšana ar rokām vai datora palīdzību bieži vien ir darbietilpīga, prasa daudz resursu un laika[9]. Runājot par to, kas tieši ir pirmkoda dokumentācija, jāsaka, ka tā ir pirmkoda dokumentācija, kas apraksta pirmkoda darbību dažādos veidos, atklātā tekstā dažādās valodās [8]. Kā secināts vienā no resursiem [8]: koda kopsavilkums ir neliels teksta apjoms, apmēram 1-3 teikumi. Bet, protams, tas ir

atkarīgs no situācijas, cik lielam aprakstam jābūt. Šis jautājums tiks pētīts sīkāk, izmantojot esošos rīkus. Koda kopsavilkums izskaidro pirmkoda darbību vai tekstā paskaidro, kā to var izmantot [8]. Avota koda dokumentēšana tiek veikta divējādi: metodes apraksts ir īsa dokumentācija, bet virknes apraksts ir detalizēta dokumentācija [6].

Avota koda dokumentācija ir kodu un tekstu kopums, kas apraksta koda darbību dažādos veidos. Šādai dokumentācijai ir jābūt kodolīgai, īsai, skaidrai un labi strukturētai. Avota koda dokumentācija ir ļoti svarīga, ja tiek izmantots automatizēts rīks, jo, kā tas parasti notiek, izstrādes procesā kods bieži mainās. Turpmākajā praktiskajā darbā aplūkosim, kā tiek veidota pirmkoda dokumentācija un no kā tai sastāvēt.

1.7. API Dokumentācijās izpēte

Programmatūras izstrādes procesā tiek izmantota ne tikai iepriekšējā sadaļā aprakstītā pirmkoda dokumentācija, bet arī tā sauktā API dokumentācija. Aplūkojot šos dokumentācijas veidus, ir svarīgi saprast, kāda ir katra no tiem ietekme uz izstrādes procesu.

Kā apgalvo pētījuma [14] autors, API dokumentācija ir ļoti vienkārša, bet, savukārt, grūti saprotama, nepilnīga un dažkārt neprecīza, turklāt šāda dokumentācija lielākoties ir rakstīta ar roku un parasti tiek sniegta kā vienkāršs teksts. Līdz ar to cilvēki, kas saskaras ar šādu dokumentāciju, saskaras ar problēmām, jo programmētājiem, kas lasa API dokumentāciju, ir pareizi jāizprot viss konteksts, lai vēlāk to varētu izmantot. Turklāt šāda dokumentācija, ja var ticēt rakstam [14], jau eksistē, un tā ir API dokumentācijas veidā, ko var viegli apstrādāt ar datoru palīdzību. Šāda dokumentācija ir rakstīta programmēšanas valodā, kas atbilst visiem standartiem.

1.1. tabula

Dokumentācijas veidu priekšrocības un trūkumi

Dokumentācijas veids	Priekšrocības	Trūkumi
Vienkāršs teksts	Lietotājam lasāms	Grūti apstrādāt ar mašīnām
Manuāli izveidots OpenAPI	Apstrādājams ar mašīnu	Bieža kļūda un lielāka izmaksas
OpenAPI automātiski radīts	Apstrādājams un vispārināms	Lietotājam grūti lasīt
Interaktīva OpenAPI	Apstrādājams un viegli izmantots	Lietotājam grūti lasīt

Šajā tabulā (1.6.) ir apkopoti API dokumentācijas veidi, to priekšrocības un trūkumi. No tabulas var secināt, ka vienkārša teksta dokumentācija būs viegli lasāma cilvēkam vai programmētājam, bet, savukārt, tā būs grūti lasāma un sarežģīta mašīnai. Ja tiek izmantota manuālā OpenAPI dokumentācija, tā ir mašīnlasāma, taču tajā biežāk ir kļūdas, un tas uzņēmumam izmaksā dārgi. Ja tiek izmantota automātiska un interaktīva OpenAPI dokumentācija, tā ir mašīnveidojamā, bet lietotājiem ir grūti lasāma. Var secināt, ka nav konkrēta veida dokumentācijas, ko ieteicams izmantot, viss ir atkarīgs no situācijas un uzņēmuma prasībām.

Noslēdzot šo sadaļu, ir jāpievērš uzmanība API dokumentācijai, lai nodrošinātu, ka programmētāji saņem labu un precīzu dokumentāciju, ar kuru viņi var efektīvi un saprotami strādāt. API dokumentācijas izmantošana sniedz dažādus ieguvumus, tostarp paātrina izstrādes procesu un samazina projekta izmaksas.

1.8. Atslēgvārdu nozīmes izpēte automatizētajā dokumentācijas izveidē

Pēc divu veidu dokumentācijas aprakstīšanas ir lietderīgi aprakstīt dokumentācijas atslēgvārdu nozīmi, lai precīzi saprastu, kā darbojas automātiskā dokumentācijas ģenerēšana. Tas palīdzēs turpmākajā darba ietekmes uz izstrādes procesu analīzē, kā arī rīka izstrādē, jo atslēgvārdu izmantošanai koda analīzē ir svarīga nozīme.

Atslēgvārdi ir arī viena no automatizācijas pielāgošanas dokumentācijas neatņemamām sastāvdaļām. Vienā no pētījumam tika veikta izpēte, lai saprastu, kam tieši programmētāji pievērš uzmanību, lasot dokumentāciju. Izmantojot skatiena izsekošanas sensoru, tika nolemts saprast, kam tieši programmētāji pievērš uzmanību,

lai nākotnē to varētu izmantot avota koda apkopošanā un turpmākajā dokumentācijā [8]. Pētījums parādīja, ka atslēgas vārdi, kurus lasa programmētāji, patiesībā ir atslēgas vārdi, kurus neatkarīga izstrādātāju grupa uzskata par svarīgiem [8]. Izmantojot šo pētījumu, ir iespējams saprast, kuri atslēgvārdi būtu jāizmanto, lai izveidotu automatizācijas dokumentu. Bet, protams, viens no atslēgvārdu atlases veidiem pastāv jau šodien. Vienu no mūsdienīgiem paņēmieniem, kā atlasīt atslēgvārdus no pirmkoda kopuma, ir aprakstījis Haiducs [7], [8]. Šīs tehnikas galvenā ideja ir vektoru telpas modeļa izmantošana, kas, kā uzskata zinātniskā raksta secības autors, ir klasiska tehnika dabiskās valodas izpratnē [8].

Kopumā atslēgvārdi ir ļoti svarīgi izstrādes procesā, turklāt tiem ir liela nozīme arī dokumentēšanas procesa automatizēšanā. Izmantojot ieteiktās metodes, ir iespējams precīzi saprast, uz kādiem vārdiem un lietām programmētājs koncentrējas. Atslēgas vārdu nozīme lielā mērā noteiks arī gaidāmā darba risinājumu.

1.9. Dokumentācijas rīku izpēte

Šajā sadaļā ir aprakstīta informācija par esošajiem dokumentēšanas rīkiem un to izpēte. Svarīgi ir noskaidrot, kādu darbu šajā jomā ir veikuši citi, kas ir jāuzlabo un kāda ir tā ietekme uz izstrādes procesu. Izmantojot šo informāciju, kļūs skaidrs, kam jāpievērš uzmanība, izstrādājot savu rīku.

Izstrādātājiem vai cilvēkiem, kuri mēģinās izveidot automatizētu dokumentu ģeneratoru, lai palīdzētu programmētājiem izstrādes procesā, ir precīzi jāzina, kā darbojas citi dokumentēšanas rīki. Izstrādātājiem precīzi jāzina, kādi programmatūras izstrādes rīki un paņēmieni tiek izmantoti, kādi dokumenti ir jāveido, kādi standarti tiek izmantoti un kāda dokumentu publicēšanas sistēma tiek izmantota [11]. Šīs pētījumos iegūtās zināšanas izstrādātāji var izmantot, lai izveidotu automatizētus dokumentu ģeneratorus. Šīs zināšanas ietver kopas katrai piemērojamai izstrādes rīka, metodes, standartu un dokumentu publicēšanas sistēmas kombinācijai.

Kā minēts iepriekš, dokumentācijas autori bieži izmanto ģenerēšanas rīkus, lai sazinātos ar lasītājiem standartizētā veidā [7]. Kā piemēru var minēt tādus rīkus kā DoxyGen un JavaDoc. Abi šie rīki ir veicinājuši pirmkoda dokumentācijas izstrādi, standartizējot pirmkoda dokumentācijas formātu un noformējumu [7]. Šie divi aprakstītie rīki pieprasa, lai programmētāji rakstītu kopsavilkumus tā, kā tie būtu

rakstāmi, īpaši formatētus metadatus, un viss, ko rīki var izdarīt, ir ievietot informāciju HTML dokumentos [7], [8].

JavaDoc rīks tika sīkāk aprakstīts divos resursos. Tajos apgalvots, ka JavaDoc tagad tiek izmantots kā standarta prakse Java programmēšanas valodu pasaulē [15],[6]. Resursu rakstīšanas laikā autors radīja apgalvojumu - JavaDoc tiešām nevar automātiski ģenerēt avota dokumentāciju pielāgotām metodēm [15],[6]. Taču tas ir pierādīts fakts, tehnoloģijas strauji attīstās, un līdz ar tām attīstās un atjaunināsarī programmatūra. Tāpēc šobrīd nav iespējams secināt, cik tie ir patiesi. Šāds apgalvojums tiks sīkāk aplūkots turpmākajā darbā, lai noskaidrotu, kā ir ar šo rīku. Runājot par otru tajā pašā pētnieciskajā darbā aprakstīto rīku, Doxygen piedāvā līdzīgu funkcionalitāti C un C++ valodu lietotājiem, kuri paļaujas uz to [15]. Abi rīki var arī invertēt kodu, lai izveidotu klašu diagrammas [15]. Protams, lai sniegtu pilnīgu pārskatu par programmatūru, tagad minētie būtu vēl jāpārskata un jāizpēta, lai noskaidrotu pašreizējo stāvokli.

Galvenā problēma, ar ko saskaras automātiskā pirmkoda apkopošana, ir tā, ka nav saskaņota vai skaidri definēta "labas" dokumentācijas standarta [7]. Aprakstīto rīku mērķis ir veikt uzdevumus, kas tiek parādīti konkrētu ietvaru efektivitātē, taču šādi rīki nevar sasniegt cilvēka veidotu kopsavilkumu kvalitātes līmeni [8].

Nav noslēpums, ka mūsdienās ir arī citi jauni rīki, kas var automātiski izveidot programmatūras dokumentāciju un palīdzēt izstrādātājiem ietaupīt laiku un pūles citiem izstrādes procesiem. Viens no šādiem rīkiem vai, labāk sakot, sistēmām ir aprakstīts zinātniskajā rakstā [16]. Bet arī jauna sistēma ir ļoti detalizēti aprakstīta zinātniskajā rakstā ar numuru [6]. Esošie rīki, piemēram, DoxyGen un JavaDoc, kas tika aprakstīti iepriekš, ir ļoti ierobežoti un spēj definēt tikai noteiktu programmēšanas valodu, sev noteiktas metodes, tā uzskata viens no avota autoriem, kurš savukārt piedāvāja savu risinājumu [6]. Darbā [6] ir aprakstīta un piedāvāta jauna sistēma, kas automatizē C pirmkoda dokumentēšanas procesu, dokumentējot katru C programmas rindu. Šāda jauna metode izmanto tā saukto NLP avota koda kompilēšanas metodi [6]. Galvenā atšķirība no JavaDoc un DoxyGen ir formāts. Balstoties uz [6] darbu, viņas ierosinātā sistēma var dokumentēt pirmkodu divos līmeņos: abstraktā un detalizētā. Kā apgalvo Menaka Pushpa Arthur [6], šādas jaunas sistēmas galvenie ieguvumi ir saistīti ar izstrādes procesa dzīves cikla laika un izmaksu samazināšanu.

Šobrīd ir pieejami dažādi rīki, kas palīdz programmētājiem izveidot dokumentāciju, no kuriem var izcelt JavaDoc un DoxyGen. Līdz šim tiem ir viena

galvenā problēma: tiem nav vienotas standartizētas sistēmas, kas būtu piemērota visām izstrādātajām programmām. Lai varētu izmantot šādus rīkus, programmētājiem vai lietotājiem ir jāpārzina tie un dokumentācijas standarti.

1.10. Pašreizējais stāvoklis un tendenču analīze

Runājot par pašreizējo situāciju automātiskās dokumentācijas izveides jomā, saskaņā ar analizētajiem zinātniskajiem rakstiem jau pastāv un tiek izmantoti dažādi automatizācijas rīki, piemēram, DoxyGen un JavaDoc. Tomēr šiem rīkiem joprojām ir vairāki trūkumi. Piemēram, JavaDoc: var strādāt tikai ar Java kodu, ir atkarīgs no koda komentāriem un nevar aprakstīt sarežģītus algoritmus. Līdz ar to JavaDoc rīka izmantošana izstrādes procesā ietekmē to, ka programmētājiem ir jāveic papildu darbības, kā rezultātā tiek zaudēts laiks un enerģija.

Automatizācijas process strauji attīstās, un kļūst pieejami arvien vairāk automatizācijas rīku. Dokumentācijas automatizācija joprojām ir aktuāla un strauji attīstās, parādās jauni rīki, programmatūra un citi zinātniskie pētījumi par šo tēmu. Šobrīd ir sarakstīti vairāki zinātniski avoti par automatizētas dokumentācijas izmantošanu, avoti par jauniem rīkiem, par dažādiem dokumentācijas veidiem un to, kā to automatizēt. Taču ne vienmēr tiek rakstīts un pētītas priekšrocības, trūkumi un ietekme uz izstrādes procesu. Nav pierādīts un praksē novērtēts, cik labi tie palīdz izstrādes procesā un kāda ir to ietekme uz programmētājiem un uzņēmumiem.

Analizējot zinātnisko rakstu, kurā aprakstīta pašreizējā situācija un tendences, varam izdarīt nelielu secinājumu - tēma "Automatizētas dokumentācijas ģenerēšanas ietekmes uz izstrādes procesu izpēte" ir aktuāla arī mūsdienās.

1.11. Secinājumi

Šāda dokumenta galvenais mērķis ir izprast automatizētās dokumentācijas ietekmi uz izstrādātājiem un izstrādes procesu kopumā, kā arī noteikt, kādi rīki vai sistēmas pastāv pašlaik un kādu informāciju un labojumus tās sniedz programmētājiem.

Izlasot, analizējot un aprakstot zinātniskos rakstus, ir iespējams izvirzīt hipotēzi: automātiskā dokumentācijas ģenerēšana pozitīvi ietekmē programmētājus un uzņēmumus, kas izstrādā sistēmu. No zinātniskajiem avotiem ir iespējams noteikt, kā aprakstītā tehnoloģija paātrina izstrādes procesu, atbrīvo programmētājus no nevajadzīgām darbībām un ļauj viņiem izveidot vienotu standarta dokumentu, lai citi

programmētāji precīzi saprastu, kā sistēma darbojas. Tas arī palīdz viņiem iegūt pieredzi koda rakstīšanā.

Galvenā problēma, kas saistīta ar šādu ģeneratoru, ir tā, ka daudzi cilvēki vēl precīzi nesaprot, kā būtu jāveido automātiskās dokumentācijas izveides rīks, lai tas būtu piemērots visām programmām un sistēmām, jo katra sistēma ir unikāla. Otrs trūkums ir saistīts ar šāda ģeneratora, rīka vai sistēmas izstrādi konkrētām darbībām, kas prasa daudz tehnisku zināšanu, zināšanu par izstrādes procesu un, protams, laiku. Var arī atzīmēt, ka pašlaik nav praktiski pierādīts, kā automatizētā dokumentācija ietekmē izstrādes procesu. Nav veikts praktisks pētījums, uz kura pamata varētu noteikt, kādu labumu tas dos programmētājiem, uzņēmumiem un izstrādes procesam kopumā. Turklāt var atzīmēt, ka pētnieciskajos rakstos nav bijusi informācija par to, vai šādi rīki tiek izmantoti uzņēmumos un kas par to tiek rakstīts. Turpmākajā rakstā ir uzskaitīti visi iepriekšējo rakstu trūkumi, kurus ir vērts ņemt vērā, izstrādājot savu algoritmu un padziļināti pētot šo tēmu.

Lai atbildētu uz jautājumu: kāpēc tēma "Automatizētas dokumentācijas ģenerēšanas ietekmes uz izstrādes procesu izpēte" ir aktuāla? Uz šo jautājumu var atbildēt: analizējot visus atlasītos pētnieciskos rakstus, ir skaidrs, ka šobrīd pētījumi par šo tēmu, lai noskaidrotu automatizētās dokumentācijas ģenerēšanas priekšrocības un trūkumus, iepriekš nav veikti. Turklāt vēl nav noskaidrots, cik lielā mērā automatizētā dokumentācijas ģenerēšana paātrina izstrādes procesu ar jau pieejamajiem rīkiem.

Turpmākajā darbā būs jāizpēta, kā automātiskā dokumentācijas ģenerēšana ietekmē programmatūras izstrādes procesu, tā sastāvdaļas, programmētājus un atslēgvārdu lomu dokumentācijas ģenerēšanā. Kā arī noskaidrot, kādas ir šādas tehnoloģijas priekšrocības un trūkumi, jo līdz šim izlasītajos rakstos tas nav identificēts. Tāpat noteikt salīdzināšanas kritērijus, kas tiks izmantoti, lai tieši novērtētu procesa efektivitāti un dokumentācijas kvalitāti, izmantojot izstrādātu algoritmu.

2. RISINĀJUMĀ DAĻA

Risinājuma daļā ir ierosināts un aprakstīts jauns algoritms automātiskai dokumentācijas ģenerēšanai, kas būs nepieciešams, lai izpētītu izstrādes procesu, lai noteiktu automātiskās dokumentācijas ietekmi uz izstrādes procesu. Risinājuma daļas galvenais mērķis ir izstrādāt savu automātiskās dokumentācijas ģenerēšanas algoritmu, pēc kura izstrādes varēs atbildēt uz svarīgiem darba jautājumiem. Iegūtie rezultāti būs nepieciešami, lai noteiktu, kādu ietekmi uz izstrādes procesu atstāj automātiskās dokumentācijas ģenerēšanas rīks vai programmatūra, kā arī kādas priekšrocības un kādus trūkumus tas dod uzņēmumiem un programmētājiem programmatūras izstrādes procesā. Izstrādātais algoritms iespēju robežās demonstrēs, kā notiek koda dokumentēšana un kā algoritms tiek testēts, izmantojot Python valodu.

Kā ir aprakstīts iepriekš, izstrādes procesā tiek izmantoti dažādi dokumentācijas veidi. Ierosinātajā sadaļā ir aplūkota un aprakstīta koda dokumentācija. Tas tiek darīts, izmantojot koda komentārus, kas, savukārt, ir rakstīti īpašā formātā.

Šajā sadaļā ir detalizēti aprakstīts pašu izstrādāts algoritms automātiskai dokumentācijas ģenerēšanai. Ir sniegts algoritma grafiskais attēlojums, un katrs algoritma solis ir sīki aprakstīts. Aprakstā jāiekļauj informācija par to, kas ir nepieciešams šādam algoritmam, kāpēc tas ir nepieciešams, kā tas tiek izveidots un kas tika izmantots, lai veiksmīgi izveidotu šādu algoritmu.

2.1. Rīku izvēles pamatojums priekš algoritma izstrādei

Pirms ķerties pie automātiskās dokumentēšanas algoritma izstrādes un apraksta, ir jāizvēlas optimālā un šim uzdevumam piemērotākā programmēšanas valoda, kas atbilst visiem nepieciešamajiem kritērijiem. Izvēlētā programmēšanas valoda palīdzēs aprakstīt un attēlot katru algoritma soli un tā izstrādes veidu.

2.1.1. Programmēšanas valodas izvēle un tas pamatojums

Šajā nodaļā ir aprakstīts, kā tika izvēlēta programmēšanas valoda, uz kuras pamata tiks izpildīti katra algoritma soļi. Lai izvēlētos piemērotāko programmēšanas valodu šādam uzdevumam, tika atlasīti galvenie kritēriji, kuriem jābūt pieejamiem izvēlētajā programmēšanas valodā. Šie kritēriji ir šādi:

- Teksta apstrāde – izvēloties programmēšanas valodu algoritmu rakstīšanai, ir svarīgi piedāvāt lietotājiem iespēju efektīvi un ātri apstrādāt tekstu, analizēt regulārās izteiksmes vai nodrošināt augsta līmeņa teksta analīzes funkcijas.
- Bibliotēkas pieejamībā - Lai izveidotu automātisku dokumentāciju, izmantojot programmēšanas valodu, ir nepieciešamas papildu bibliotēkas teksta analīzei, failu vai mapju atlasei, dokumentācijas uzturēšanai un citas svarīgas bibliotēkas, kas atvieglo un paātrina algoritma darbību.
- Dokumentā izveidošana – Tā kā plānotā algoritma galvenais mērķis ir izveidot dokumentāciju HTML formātā, uzmanība tiek pievērsta arī šī dokumentācijas formāta atbalstam. Tas nozīmē, ka izvēlētajai programmēšanas valodai jāatbalsta HTML dokumentācijas formāts.
- Automatizācijas uzdevuma izpilde - Izvēlētajai programmēšanas valodai jābūt universālai, spējīgai automatizēt procesus, rīku izstrādi un darbības kopumā. Tas ir nepieciešams, lai automatizētu dokumentāciju un citus procesus.
- Valodas veiktspēja – Lai nodrošinātu efektīvu, ātru un vienkāršu izstrādes procesu, ir jāizvēlas programmēšanas valoda, kas ir pietiekami ātra un efektīva, lai veiksmīgi ģenerētu dokumentāciju lielam koda vai projekta apjomam.
- Izmantošana un resursi pieejamība – Izvēlētajai programmēšanas valodai jābūt viegli lietojamai, un tajā laikā jābūt pieejamiem daudz resursiem un dokumentācijai par tās lietošanu.

Kritēriji tika aprakstīti kā nepieciešamie algoritma veiksmīgai izstrādei, kā arī tā izmantošanai konkrētā uzdevuma risināšanā. Izvērtējot bakalaura darba rakstīšanas laikā populārākās programmēšanas valodas un analizējot tās pēc iepriekš aprakstītajiem kritērijiem, tika izvēlēta viena programmēšanas valodām, kas bija piemērota dotajam uzdevumam. Galvenā programmēšanas valoda dokumentācijas algoritma automātiskai ģenerēšanai un tā demonstrēšanai bija Python, kas atbilda visiem iepriekš aprakstītajiem kritērijiem.

Ja runājam par to, kāpēc prototipēšanai tika izvēlēta tieši šī valoda, varam minēt šādus kritērijus: pirmkārt, Python spēj analizēt un apstrādāt tekstu un izteiksmes, kas ir ļoti svarīgi risinājumu izstrādē; otrkārt, Python ir ļoti plašs bibliotēku saraksts, ko var

izmantot, lai viegli un efektīvi izmantotu funkcijas un metodes programmatūras izstrādē. Tās ietver: dokumenta atlasi, atvēršanu, lasīšanu un izveidi failā vai mapē. Tā kā automātiskai dokumentu izveidei nepieciešams citas programmēšanas valodas atbalsts, Python ir labs valodas variants, jo tajā ir iespēja izmantot citu programmēšanas valodu, piemēram, HTML. Turklāt Python programmēšanas valoda ir ļoti labi piemērota procesu vai darbību automatizēšanai. Šāds kritērijs ir noderīgs un svarīgs šajā dokumentā, jo algoritms ir izstrādāts automatizācijai.

Ne visas pieejamās programmēšanas valodas ir piemērotas šādam uzdevumam. Papildus programmēšanas valodu analīzei Python vide nodrošina milzīgu informācijas apjomu un ļoti plašu dokumentāciju.

2.1.2. Dokumentācijas formāta izvēle un pamatojums

Tā kā automātiskās dokumentācijas ģenerēšanas algoritma galarezultātam ir jābūt koda dokumentācijai, uzdevums bija izvēlēties informācijas nesēju, kurā izvietot šo informāciju. Izanalizējot jau pieejamos dokumentācijas ģenerēšanas rīkus, piemēram, DoxyGen un JavaDoc, kā galarezultāts tika izvēlēts HTML formāts. Šī programmēšanas valoda labi sader ar Python, un HTML formāts ir ļoti efektīvs un viegli lietojams, spriežot pēc esošajiem rīkiem un to sniegtās informācijas.

2.1.1. sadaļā ir sīki aprakstīts viss sākotnējais izstrādes process, tostarp autoru kritēriju izveide, programmēšanas valodas un dokumentācijas formāta izvēle. Katra izvēle tiek pamatota, un tiek izvēlēta valoda un formāts, kas ir optimāls gala rezultātam. Šajā darbā tika izvēlēta Python programmēšanas valoda un HTML dokumentācijas formāts.

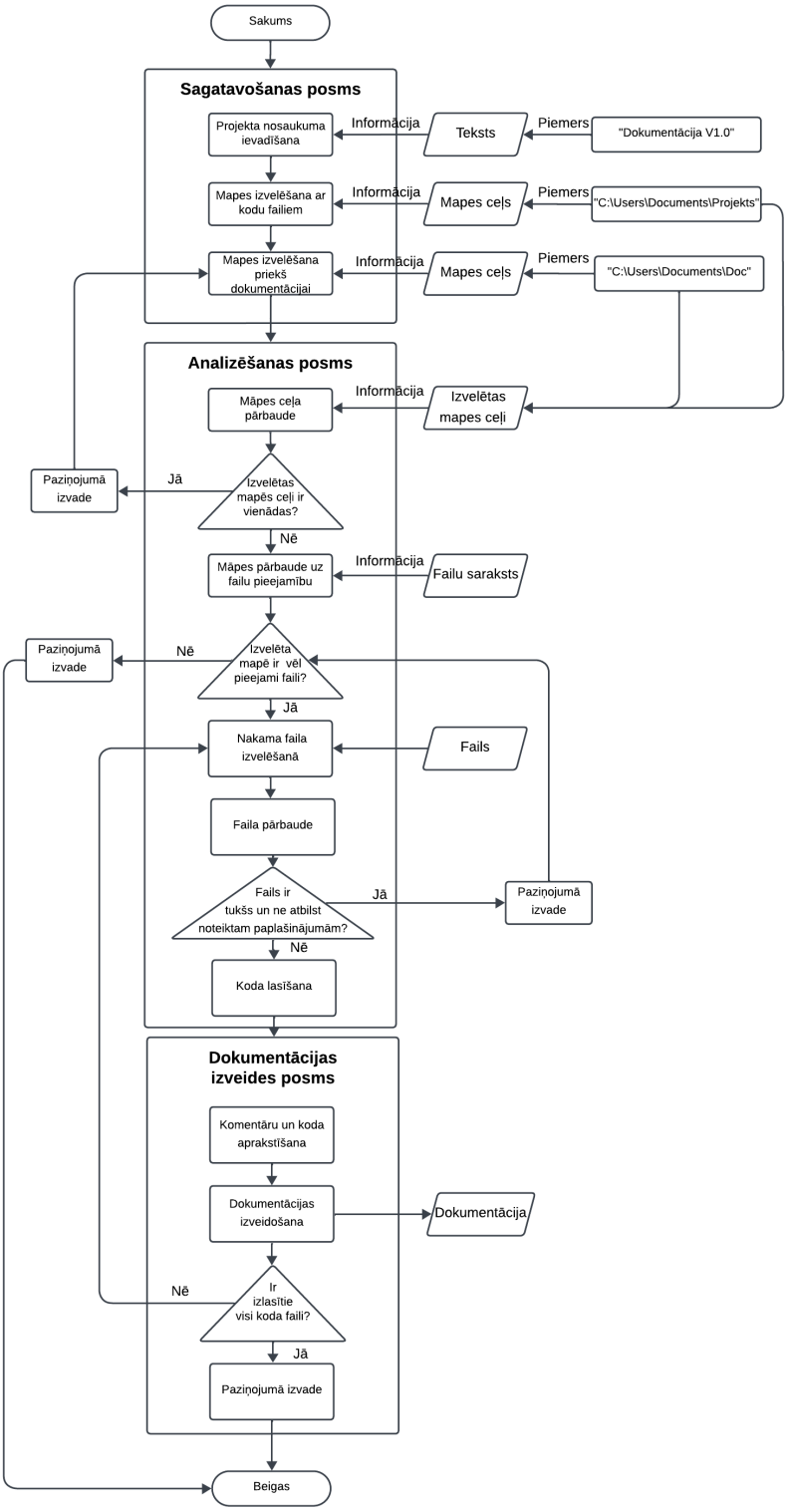
2.2. Algoritma izstrādes fāzes apraksts

Pirms algoritma projektēšanas ir jānoskaidro, kāda veida informācija ir nepieciešama tā darbības laikā, un jānosaka, kas ir nepieciešams, lai veiksmīgi izstrādātu algoritmu, kura uzdevums ir dokumentēt kodu. Šajā sadaļā ir aplūkots, kāda funkcionalitāte būs pieejama izstrādātajā prototipā, kāda veida un tipa dokumentācija ir ģenerēta, kā arī aprakstīts, kā izskatīsies automātiskās dokumentācijas ģenerēšanas process, un sniegta tā shēma.

2.2.1. Dokumentācijas algoritma struktūra

Šādas apakšnodaļas galvenais uzdevums ir izveidot algoritma struktūru, aprakstīt to un grafiski parādīt. Struktūras izveides mērķis ir saprast, kā darbojas automātiskā dokumentācija, kādas galvenās funkcijas tiks izmantotas tehnoloģiskajā procesā, kas nepieciešams dokumentācijas izveidei, un dokumentācijā atspoguļot algoritma veida izveides procesu.

Visu algoritma izstrādes procesu var iedalīt trīs posmos: sagatavošanas posmā, analīzes posmā un dokumentēšanas posmā. Katra grupa ietver dažādus procesus, kas nepieciešami automātiskai dokumentēšanai. Tālāk sniegtajā diagrammā ir parādīta automātiskās dokumentēšanas algoritmu struktūra:



2.1. att. Izstrādāta algoritma darbības posmi

Diagrammā redzams, ka visa algoritma struktūra ir sadalīta 3 galvenajos posmos, no kuriem katram ir savi procesi. Pirmais sagatavošanās posms sastāv no trim galvenajiem soļiem, proti, projekta nosaukuma ievadīšana, piemēram, "Dokumentācija

V1.0", un mapes izvēle. Mapes tiek izvēlētas gan koda failu lasīšanai, gan dokumentācijas glabāšanai. Pēc sagatavošanas posma tiek pārbaudīts ceļš uz izvēlēto mapi, lai noteiktu, vai izvēlētajām mapēm ir vienāds ceļš. Ja ceļi ir vienādi, algoritms lūgs izvēlēties citu mapi dokumentācijas glabāšanai. Nākamajā solī tiek pārbaudīta mape, lai pārliecinātos, vai faili ir pieejami. Ja mape ar kodu failiem ir tukša, algoritms parādīs ziņojumu un pārtrauks visu algoritma darbību. Pēc mapes pārbaudes tiek izvilkts pirmais fails, lai pārbaudītu, vai tas ir tukšs. Ja fails ir tukšs, algoritms vienkārši izlaiž izvilkto failu un izvēlas nākamo failu no mapēm. Tas pats notiek, ja tiek pārbaudīts faila paplašinājums. Pēc tam fails tiek nolasīts, komentēts, identificēts un aprakstīts kodā. Kad algoritms ir nolasījis visu kodu un komentārus no faila, tiek izveidota dokumentācija un saglabāta HTML formātā. HTML dokumentācija tika izveidota katram koda datumam un ievietota noteiktā mapē. Kad viss kods ir izlasīts un dokumentācija ir izveidota, algoritms pabeidz darbu.

Turpmākajās sadaļās katrs algoritma posms ir aprakstīts sīkāk, norādot, kas katram posmam nepieciešams, kāpēc tas ir nepieciešams un kas tika izmantots tā izstrādē. Python programmēšanas valoda tiek izmantota, lai ilustrētu katru izstrādātā algoritma posmu un parādītu, cik iespējams ir izstrādāt un pēc tam izmantot šādu algoritmu.

2.3. Sagatavošanas posms

Algoritms sākas ar sagatavošanas posmu, kas, savukārt, ir sadalīts trīs galvenajos posmos, kuri nepieciešami, lai algoritms varētu efektīvi darboties un dot veiksmīgu rezultātu. Pirmais solis, kas algoritmam nepieciešams, ir ievadīt projekta nosaukumu. Pārējie divi soļi ir saistīti ar mapju atlasi, no kurām katra ir paredzēta citam mērķim.

2.3.1. Projekta nosaukuma ievadīšana

Projekta nosaukuma ievadīšana ir svarīga automātiskajam dokumentēšanas rīkam, jo tā palīdz identificēt dažādus projektus, ja ir vairākas kopijas. Šis nosaukums var kalpot kā īss un vienkāršs projekta dokumentācijas apraksts. Ja izstrādes procesā dokumentācijā kods ir izmantots vairākas reizes, projekta nosaukumu var izmantot arī kā versijas aprakstu, lai noteiktu, kad šāda dokumentācija tika izstrādāta un kādā izstrādes posmā tā bija.

Projekta nosaukumā lietotājam ir manuāli jāievada teksts, izmantojot tastatūru vai citu ievades ierīci. Par projekta nosaukumu var izmantot jebkuru tekstu, ciparus un simbolus. Galvenais, lai lietotājs saprastu, kādam nolūkam tiek veidota dokumentācija. Projekta nosaukuma piemērs ir šāds: "Dokumentācija V1.0".

Lai precīzi parādītu, kā šis solis darbojas, tiek izmantota Python valoda. Ierosināto Python koda fragmentu, kas parādīts praksē, var izmantot, lai to novērstu projekta ievades datos:

```
documentation_name = input("Enter documentation name: ")
```

Projekta nosaukums tiek ievadīts, izmantojot funkciju 'input()', kas ir Python iebūvēta funkcija, kas ļauj ievadīt tekstu, skaitļus un citus datus no konsoles.

2.3.2. Mapēs izvelēšanas process

Pēc projekta nosaukuma ievadīšanas mape tiek atlasīta divos soļos. Pirmajā solī tiek izvēlēta mape, kurā atrodas koda faili, bet otrajā solī tiek izvēlēta mape, kurā tiek glabāta norādītajā mapē izveidotā dokumentācija.

Mapes atlase ir nepieciešama galvenokārt, lai viegli un efektīvi noteiktu ceļu līdz koda failiem. Atlasi nosaka mapju veids, jo dažos gadījumos dokumentācija netiek veidota vienam konkrētam koda failam, bet gan failu kopumam, kas atrodas vienā konkrētā ceļā. Turklāt mapju izvilkšana ir nepieciešama, lai saglabātu dokumentāciju. Lai mapes izvilktu viegli un ātri, tiek izmantots Windows dialoglodziņš, kas palīdz lietotājiem izvilkt vajadzīgo mapi.

Mapes atlases process sākas ar definēto funkciju "main()", kas izsauc papildu funkciju "folder()" ar diviem argumentiem, kas vajadzīgi, lai saglabātu ceļu līdz kodiem un dokumentācijas glabāšanas vietai. Kad vērtība ir iegūta, vērtību kopums tiek izdrukāts kopā ar papildu tekstu.

```
def main():
    folder_file, folder_doc = folder(None,None)
    print(f"Selected folder for codes is: {folder_file}")
    print(f"Selected folder for Documentation saving is: {folder_doc}")
```

Šajā koda fragmentā ir definēta funkcija folder(), kurai ir divi argumenti - first un second. Šī funkcija divreiz izsauc open_folder(), vienu reizi, lai iegūtu pirmo vērtību, un otrreiz, lai iegūtu otro vērtību. Kad vērtība ir iegūta, algoritms ziņo, ka ceļš ir veiksmīgi izvēlēts.

```
.def folder(first,second):
    first = open_folder()
    print("Folder is chosen successfully!")
    second = open_folder()
    print("Folder is chosen successfully!")
    return first, second
```

Funkcija 'open_folder()' ir mapju ieguves procesa galvenais algoritms. Lai funkcija darbotos pareizi un veiksmīgi, tiek izmantota Python bibliotēka tkinter un tās modulis filedialog, lai atvērtu dialogu un izvēlētos mapes atrašanās vietu. Sakumā tiek izveidots jauns objekts, un, izmantojot to, tiek izveidots un parādīts dialogs, lai izvēlētos mapes atrašanās vietu, projekta atrašanās vietu un dokumentācijas atrašanās vietu.

```
import tkinter as tk
from tkinter import filedialog
def open_folder():
    root = tk.Tk()
    root.withdraw()
    data = filedialog.askdirectory(parent=root)
    return data
```

Tālāk dotie koda fragmenti parāda, kā var izstrādāt programmatūru, kas izraksta mapes Windows dialoga formātā, izmantojot 2.2. sadaļā aprakstīto algoritmu. Tālāk aprakstītā procedūra ar mapju izvilkšanu ir pārņemta un līdzīgā veidā izmantota DoxyGen un JavaDoc rīkos.

2.4.Analizēšanas posms

Pēc sagatavošanās posma, kurā tiek ievadīts projekta nosaukums un izvēlēta mape, sākas lielais un galvenais posms, kurā tiek veiktas dažādas iestatītās darbības. Piemēram, mapju, failu, koda pārbaude, tā analīze, dokumentācijas izveide un citi saistītie procesi.

2.4.1. Mapēs pārbaude process

Kad ir atlasītas algoritma darbībai nepieciešamās mapes, pirmais solis ir to testēšana, lai pārliecinātos, ka algoritms darbojas efektīvi un ātri, bez kļūdām.

Vispirms pēc divu vajadzīgo mapju atlases tiek pārbaudīti faili mapē (pirmajā atlasītajā mapē). Tas tiek darīts, lai pārbaudītu, vai mapē ir koda faili, kurus algoritms var analizēt un neradīt kļūdas. Otrā pārbaude tiek veikta, lai pārliecinātos, ka mapju ceļi sakrīt. Tas tiek darīts, lai nodrošinātu, ka netiek traucēta algoritma un koda analīze un ka faili netiek pārrakstīti, kad dokumentācija tiek saglabāta izvēlētajā mapē.

Mapes pārbaudes procesa, jāpārbauda tikai ceļi uz iepriekš izvēlētajām mapēm. Process sākas ar pārbaudi, vai mape ir tukša. Pārbaude tiek veikta, izmantojot bibliotēku "os", kas ļauj manipulēt ar operētājsistēmu. Tā izveido norādītās mapes failu sarakstu un pārbauda, vai mape ir tukša. Ja tā ir tukša, tiek izdots ziņojums un algoritms tiek izbeigts ar sys.exit() palīdzību. Pēc pirmās pārbaudes ceļi tiek pārbaudīti un salīdzināti, izmantojot parasto "if" sintaksi. Ja ceļi sakrīt, algoritms parāda ziņojumu un lūdz izvēlēties citu mapi dokumentācijas glabāšanai. Šo procesu var atkārtot vairākas reizes, līdz otrajai izvēlētajai mapei ir cits ceļš. Pievienotajā koda logā ir redzams Python valodā aprakstītais process:

```
def folder(first,second):
    if not os.listdir(first):
        sys.exit(f"The folder: {first} - is empty.")
    if first == second:
        print("ERROR:")
        print("Chosen folders path are the same! Please choose another folder! ")
        while first == second:
            second = open_folder()
    print("Folder is chosen successfully!")
    return first, second
```

2.4.2. Failu pārbaudes process

Nākamais solis pēc mapju pārbaudes ir divu posmu failu pārbaude. Tā sastāv no faila izmēra un paplašinājuma pārbaudes.

Faila lieluma pārbaude ir nepieciešama, lai izvairītos no nevajadzīga darba, ko var veikt algoritms. Un arī, lai izvairītos no tukšas dokumentācijas izveides gadījumā, ja koda fails ir pilnīgi tukšs. Faila paplašinājuma pārbaude ir svarīga, lai nolasītu tikai noteiktus failu formātus, piemēram, ".py". Tas ir svarīgi, jo projektu mapēm un citiem resursiem var būt dažādi failu paplašinājumi. Šim procesam nav nepieciešams

konfigurēt īpašu pārbaudi. Testā tiek izmantota gatavā "os" bibliotēka, kas darbojas ar vairākiem operētājsistēmas procesiem.

Faila pārbaudi veic funkcija 'extension_check()', kas tiek izsaukta ar 3 atribūtu kopumu. Lai pārbaudes procesā apstrādātu izņēmumus, tika izmantotas konstrukcijas "try" un "except". Pirms failu pārbaudes tika izveidots izvēlētās mapes failu saraksts un pievienots mainīgajam 'file_list'. Tas tiek izmantots, lai to pievienotu cilpai, kuras mērķis ir nolasīt visus failus no mapēm un iegūt tukšos failus. Pirms pārbaudes tiek veiktas papildu darbības, tostarp ceļa rediģēšana (mainīgais 'file') un faila paplašinājuma iegūšana (mainīgais 'extension'). Lai pārbaudītu faila lielumu, tiek izmantota bibliotēka 'os' un funkcija 'getsize()', lai noskaidrotu faila lielumu. Ja fails ir tukšs (vienāds ar 0), no izveidotā saraksta tiek izvēlēts nākamais fails. Paplašinājumā tiek izmantota arī "os" bibliotēka kopā ar funkciju "splitext()". Ja faila paplašinājums atbilst ".py", algoritms izsauc file_read(). Tālāk dotajā koda fragmentā redzams, ka process ir aprakstīts Python programmēšanas valodā:

```
def extension_check(folder_file,folder_doc,documentation_name):
    try:
        file_list = os.listdir(folder_file)
        for i in file_list:
            file = os.path.join(folder_file, i).replace('\\','/')
            extension = os.path.splitext(file)[1]
            if os.path.getsize(file) == 0:
                continue
            if extension == ".py":
                file_read(file,i,folder_doc,documentation_name)
            else:
                print("The file is not a .py file")
    except FileNotFoundError:
        print("The specified file was not found.")
    except Exception as e:
        print("An error occurred:", e)
```

2.4.3. Koda lasīšana process un komentāru pielietojums

Pēc visām 2.4.1. un 2.4.2. apakš sadalās aprakstītajām pārbaudēm seko galvenais koda faila nolasīšanas process. Bet pirms koda lasīšanas ir jānoskaidro galvenie ar to saistītie punkti.

Koda lasīšana un analizēšana ir būtiska, lai veiksmīgi, ātri un efektīvi atrastu vajadzīgo informāciju failā. Tas ietver metodes, funkcijas, parametrus, vērtības, izmantotās bibliotēkas un komentārus, kas savukārt ir aprakstīti noteiktā formātā. Standartizēts komentāru formāts palīdz nodrošināt, ka kods ir pareizi komentēts un dokumentēts, kā arī ka kods un komentāri ir aprakstīti, kas ir svarīgs solis pirms dokumenta izveides.

Priekš paša komentāru veidu izveidošana un aprakstīšana, Python valodai, tiek izmantotā pārzināšanas un izpēte par rīkiem kā JavaDoc un DoxyGen. Pēc tās izpēti ir iegūta zināšanas ka vajadzētu izskatīties tas koda komentāri. Python valoda ir pieejamas dīvā tipa komentārus. Pirmais komentārs ir izmantots lai izveidot viena rinda komentāru, kas sakās ar simbolu: "#". Šādus komentārus algoritms ari ņems vēra un aprakstīs tas gan kā kodā, gan komentāru ietvaros. Otra tipa komentārs ir izmantots tieši priekš dokumentācijai. Tās komentāru ietvaros būs ievietota noteikta informācija izmantojot noteiktu apzīmējumu. Komentārs izskatās šādi:

```
"""
Komentārs
"""
```

Otrā tipa komentārs, kas apraksta konkrētu koda fragmentu, jānovieto tieši pirms tā. Pēc detalizētas koda komentāru izpētes tika sastādīta šāda 2.1. tabula, lai tos apkopotu:

2.1. tabula

Python komentāru veidi un tās apraksts

Komentāra veids	Sintakse	Apraksts
Autors	**$author** (teksts)	Dokumentācijai pievieno tās kodā, failā vai noteikta klases autoru.
Versija	**$version** (versija)	Dokumentācijai pievieno tās koda, failā vai noteikta klases versiju.
Laiks	**$date** (datums)	Dokumentācijai pievieno laiks kad kods, klases, funkcija bija izgatavota.
Klase	**$cl** (klases apraksts)	Dokumentācijai pievieno klases apzīmējumi un aprakstu, pirms tas klases koda.
Metode	**$method** (metodes apraksts)	Dokumentācijai pievieno metodes apzīmējumi un aprakstu, pirms tas metodes koda.
Apraksts	# (apraksts)	Dokumentācijai pievieno papildu aprakstu par failu, klasi, metodi vai funkciju.
Vērtība	**$value** (vertības apraksts)	Dokumentācijai pievieno apzīmējumi un aprakstu par statisko vērtību kodā.
Izņēmums	**$exception** (klases apraksts)	Dokumentācijai pievieno apzīmējumi un aprakstu par izņēmuma klasi.
Parametrs	**$par** (parametra apraksts)	Dokumentācijai pievieno apzīmējumu un aprakstu par izmantotu parametru.
Atgriežamā vērtība	$**return** (atgriežamas vērtības apraksts)	Dokumentācijai pievieno apzīmējumu un aprakstu par atgriežamas vērtības metodes darba laika.

Izmantojot piedāvāto 2.1. tabulu, ir iespējams saprast, uz kuriem komentāriem un to anotācijām ir vērsts izstrādātais algoritms, lai identificētu šos komentārus. Katrs no ierosinātajiem komentāru veidiem ir paredzēts konkrētam uzdevumam. Dokumentācija ir sniegta, aprakstot autoru, versiju un koda rakstīšanas laiku, ja tādi kodā ir. Klases, metodes, vērtības, izņēmumi, parametri un atgriešanas vērtības var tikt izmantotas visā kodā līdz pat konkrētam koda fragmentam. Aprakstītā sintakse palīdz saprast, kā lietot komentārus koda failā. Metožu, funkciju un klašu, bibliotēku

definēšana un aprakstīšana tiek veikta arī bez komentāru palīdzības. Izpildes laikā algoritms izveido to metožu, funkciju un klašu sarakstu, kuras tiek izmantotas kodā.

Zemāk redzamajā koda fragmentā varat redzēt, kā sintakse tiek lietota prakse, lai precīzi redzētu, kā komentāri ir jāizmanto. Šajā nelielajā koda fragmentā divu skaitļu summas aprēķināšanai tiek izmantota viena metode, viena funkcija un viens diapazons, un vienai vērtībai tiek pievienota statiska vērtība. Koda fragmentā ir izmantoti visu veidu komentāri, izņemot $exception.

```
“""
$author Glebs Bondarevs
$version 1.0
$date 29.03.2024
# Programma AddNum ievieš lietojumprogrammu, kas
# vienkārši pievieno divus dotos veselus skaitļus un izdrukas
# izvade uz ekrāna.
"""
"""
$cl AddNum kura ietvaros atrodas metode
"""
class AddNum:
    """
    $method addNum tiek izmantots, lai pievienotu divus veselus skaitļus
    $par numA Ir pirmais metodes parametrs
    $par numB Ir otrais metodes parametrs
    $return int atgriež summu numA un numB.
    """
    def addNum(self, numA, numB):
        return numA + numB
"""
$method main kura uzsauks addNum metodi un pievieno skaitlus pie ta
$value A statiska vertiba kura ir int vertiba 10
"""
def main():
    A = 10
    obj = AddNum()
    sum = obj.addNum(A, 20)
    print("Sum of 10 and 20 is:", sum)
```

2.5. Dokumentācijas izveides posms

Dokumentācijas posms ir šā algoritma pēdējais posms, kura uzdevums ir apkopot visu izlasīto kodu un izveidot par to dokumentāciju. Šajā posmā ietilpst komentāru un koda aprakstīšana un sagatavošana HTML formātam, dokumentācijas izveide un saglabāšana noteiktā vietā.

2.5.1. Komentāru un koda aprakstīšanas process

Komentāri un kodi tiek aprakstīti, izmantojot komentāru tipus, kas jau aprakstīti 2.4.3. sadaļā. Tos var veiksmīgi izmantot, lai izveidotu dokumentāciju, kas satur komentārus par konkrētu parametru, vērtību, metodi, funkciju un citu noderīgu informāciju, kā arī par komentēto kodu.

Aprakstīšanas process ir pēdējais solis pirms dokumentācijas izveides. Tā ir svarīga, jo bez tās nav iespējams izveidot strukturētu, viegli pārskatāmu un informatīvu dokumentāciju, kurā var atrast visu nepieciešamo informāciju par kodu. Lai izstrādātu algoritmu, kura uzdevums būs aprakstīt kodu, ir jāizmanto datne, kas satur kodu, definējot komentāru tipus un izmantojot Python programmēšanas valodas bibliotēku.

Funkcija 'file_read()' ir koda rindu kopums, ko izmanto, lai analizētu un aprakstītu kodu. Funkcija 'file_read()' tiek izsaukta kopā ar 4 nepieciešamajiem atribūtiem koda aprakstam un komentāriem. Pašā algoritma sākumā tiek pārbaudīta faila pieejamība, lai izvairītos no nevajadzīgām kļūdām. Ja šāds fails ir pieejams, programma to atver un pārveido, lai nodrošinātu lasīšanu rindiņa pa rindai. Nākamais solis ir faila nolasīšana pa rindiņām, izmantojot cilpu. Lasot katru faila rindu, tiek pārbaudītas īpašās rakstzīmes, lai noteiktu, kas ir jāapraksta. Ja algoritms sastop trīs pēdiņas rakstzīmes " " " kādā koda rindā, tas izsauc komentāra identifikatoru, kas aprakstīts 2.1. tabulā, un to apraksta. Mainīgie "count", kuros tiek saglabāti skaitļi, ir vajadzīgi tieši, lai noteiktu, kurā apraksta fāzē algoritms atrodas. Tas ir nepieciešams, lai pareizi aprakstītu komentāru, ko algoritms izmantos dokumentācijas izveidei. Procedūra "if in line" definē īpašu komentāru un pēc tam to apraksta, izmantojot HTML. Komentāra apraksts tiek saglabāts mainīgajā 'html'. Dotais kods ir tikai daļa no pilna koda fragments:

```
def file_read(file,i,folder_doc,documentation_name):
    count = 0
    j = 0
```

```
if os.path.isfile(file):
    with (open(file, 'r', encoding='utf-8') as file):
        row = file.readlines()
        html = ''
        metodes_list = ''
        for line in row:
            if """ in line:
                if j > 0:
                    j = 0
                    html += f"</div>"
                count = count + 1
                if count == 1:
                    html += f'<div class=\"example\">'
                if count == 2:
                    count = 0
                    html += f"</div>"
                continue
            if count == 1:
                if '$author' in line:
                    match = re.search(r'\$author\s*(.*)', line)
                    if match:
                        html += "<p><b>Autors:</b> " + match.group(1) + f"</p>\n"
                elif '$version' in line:
                    match = re.search(r'\$version\s*(.*)', line)
                    if match:
                        html += "<p><b>Versija:</b> " + match.group(1) + f"</p>\n"
            if count == 0:
                j += 1
                if j == 1:
                    html += f'<div class=\"example\">'
                html += line
```

Rindu analizēšanas laikā tiek arī pārbaudīts, vai izvēlētā koda rinda ir aprakstīta izmantotajā bibliotēkā. Tas ļauj izvairīties no nevajadzīgām pārbaudēm un noteikt, kuras Python bibliotēkas tika izmantotas piedāvātajā kodā. Meklēšanas un aprakstīšanas process ir līdzīgs komentāru meklēšanas un aprakstīšanas procesam. Mainīgie "imp", kuros tiks glabāti skaitļi, ir tieši nepieciešami, lai noteiktu, kurā

apraksta fāzē atrodas algoritms. Tas ir nepieciešams, lai pareizi aprakstītu algoritma dokumentācijas izveidē izmantotās bibliotēkas. Tālāk dotajā koda fragmentā parādīts bibliotēku meklēšanas process un tukšu virkņu pārbaude:

```
        elif 'import' in line:
            imp = imp + 1
            if imp == 1:
                impr += f"<pre><code>"
            if imp > 0:
                impr += line + f"\n"
            continue
        elif line.strip() == "":
            if imp > 0:
                imp = 0
                impr += f"</code></pre>"
            continue
```

Metožu, funkciju un klašu atrašanai un uzskaitīšanai tiek izmantots atsevišķs algoritms. Metodes un funkcijas tiek atrastas, definējot vārdu 'def' virknē. Ja vārds "def" ir atrasts rindā, algoritms veic nākamo soli, kas ietver satura pārbaudi starp iekavām. Ja starp iekavām tiek atrasts vārds "self", algoritms identificē to kā metodi un pievieno to metožu sarakstam. Pretējā gadījumā tas tiek identificēts kā funkcija un pievienots funkciju sarakstam. Ja virknē ir vārds "class", tas tiek pievienots klašu sarakstam.

```
c = 0
if 'def' in line:
    text = re.findall(r'\((.*?)\)', line)
    for match in text:
        if 'self' in match: c = 1
    match = re.search(r'def\s*(.*)', line)
    if c == 1:
        if match:
            metodes_list += f"<li>" + match.group(1) + "</li>\n"
    else:
        if match: function_list += f"<li>" + match.group(1) + "</li>\n"
elif 'class' in line:
    match = re.search(r'class\s*(.*)', line)
    if match: class_list += f"<li>" + match.group(1) + "</li>\n"
```

2.5.2. Dokumentācijas izveidošana

Pēc tam, kad kods un komentāri ir analizēti un aprakstīti, pēdējais solis ir saistīts ar dokumentācijas izveide. Kā aprakstīts iepriekš, dokumentācijas izveide un saglabāšana tiek veikta HTML formātā..

Bez šī soļa nav iespējams izveidot koda dokumentāciju, ja nav izstrādāts tā algoritms. Tādējādi aprakstītais solis ir ļoti svarīgs veiksmīga gala rezultāta un dokumentācijas izveidei. Lai nodrošinātu kvalitatīvu un standartizētu dokumentācijas izveidi un saglabāšanu, vispirms ir jāizmanto ceļš līdz mapei, kurā dokumentācija tika saglabāta. Sīkāka informācija un algoritms šādam procesam atrodams 2.3.2. iedaļā. Tāpat, lai veiksmīgi izveidotu dokumentāciju, ir nepieciešams projekta nosaukums no 2.3.1. iedaļas, tā koda faila nosaukums, no kura tika izveidota dokumentācija, un koda apraksts un komentāri, kas iegūti, izmantojot 2.5.1. iedaļā aprakstīto algoritmu.

Dokumentācijas izveides process sākas ar dokumentācijas nosaukuma izveidi, kas tiek izveidots, izmantojot koda faila nosaukumu un html paplašinājumu. Šis process ir iespējams Python valodā, izmantojot bibliotēku "os". Kad tas ir izveidots, tiek izsaukts cits Python fails, lai izveidotu pašu html failu. Tā iekšpusē ir mainīgais, kas satur pašu html kodu. Izmantojot mainīgos no iepriekšējiem algoritma izpildījumiem, šī informācija tiek ievietota html kodā. Iepriekš minētajā koda fragmentā ir aprakstīts, kā tiek izsaukts dokumentācijas nosaukums un palaists jauns fails un funkcija:

Main.py kods:

```
def file_read(file,i,folder_doc,documentation_name):
    file_path = os.path.join(folder_doc, i + ".html")
    DocGeneration.DocGeneration(html,metodes_list,function_list,
    class_list,file_path,i,documentation_name,path)
```

Koda fragmentā "DocGeneration.py" redzams, ka dokumentāciju var izveidot, izmantojot html kodu un mainīgos. Vispirms html kodā tiek ievietoti šādi mainīgie:

- {file} – manīga ir saglabāts koda faila nosaukuma kas palīdzes noskaidrot par kuru failu ir izveidota dokumentācija.
- {documentation_name} - manīga ir saglabāta informācija par projekta nosaukumu, kas bija ievadīta lietotājiem.
- {data} - manīga ir saglabāts kodu un komentāru apraksts, kas palīdzes izveidot un strukturēt iegūtas datus.
- {current_date} – mainīga atrodas informācija kad dokumentācija bija izveidota

- {path} – manīga ir saglabāts ceļš līdz koda failam par kuru bija izveidota dokumentācijā
- {metodes_list}{function_list}{class_list} – mainīgos atrodas saraksti, kas ir izmantotas kodā, par metodēm, funkcijām un klasēm.

DocGeneration.py kods:

```
def DocGeneration(data,metodes_list,file_path,file,documentation_name):
    current_date = datetime.now().date()
    html_content = f"""
    <!DOCTYPE html>
    <html lang="en">
    <head>
      <meta charset="UTF-8">
      <title>Python Generated Documentation</title>
    </head>
  <body>
        <h1>{file}</h1>
          <h3>Documentation information:</h3>
          <p><strong>Project name:</strong> {documentation_name}</p>
          <p><strong>Documentation made:</strong> {current_date}</p>
          <p><strong>The documentation was generated from the following file:</strong>{path}</p>
          <h2>Import Statements:</h2> {impr}
          <h2>Methodes list:</h2> <pre><code>{metodes_list}</pre></code>
          <h2>Function list:</h3> <pre><code>{function_list}</pre></code>
          <h2>Class list:</h3> <pre><code>{class_list}</pre></code>
          <p></p>
          <h2>Detailed description:</h3> {data}
        </div>
    </body></html> """
    with open(file_path, "w") as file:
      file.write(html_content)
    print("HTML file generated successfully!")
```

Izmantojot funkciju “with open() as” ir iespēja izveidot dokumentāciju ar sagatavotu dokumentācijas saturu, datumu, projekta nosaukumu un ar citam ievietotam vērtībām html kodā. Pēc dokumentācijas izveidošanai ir izvadīts paziņojums par

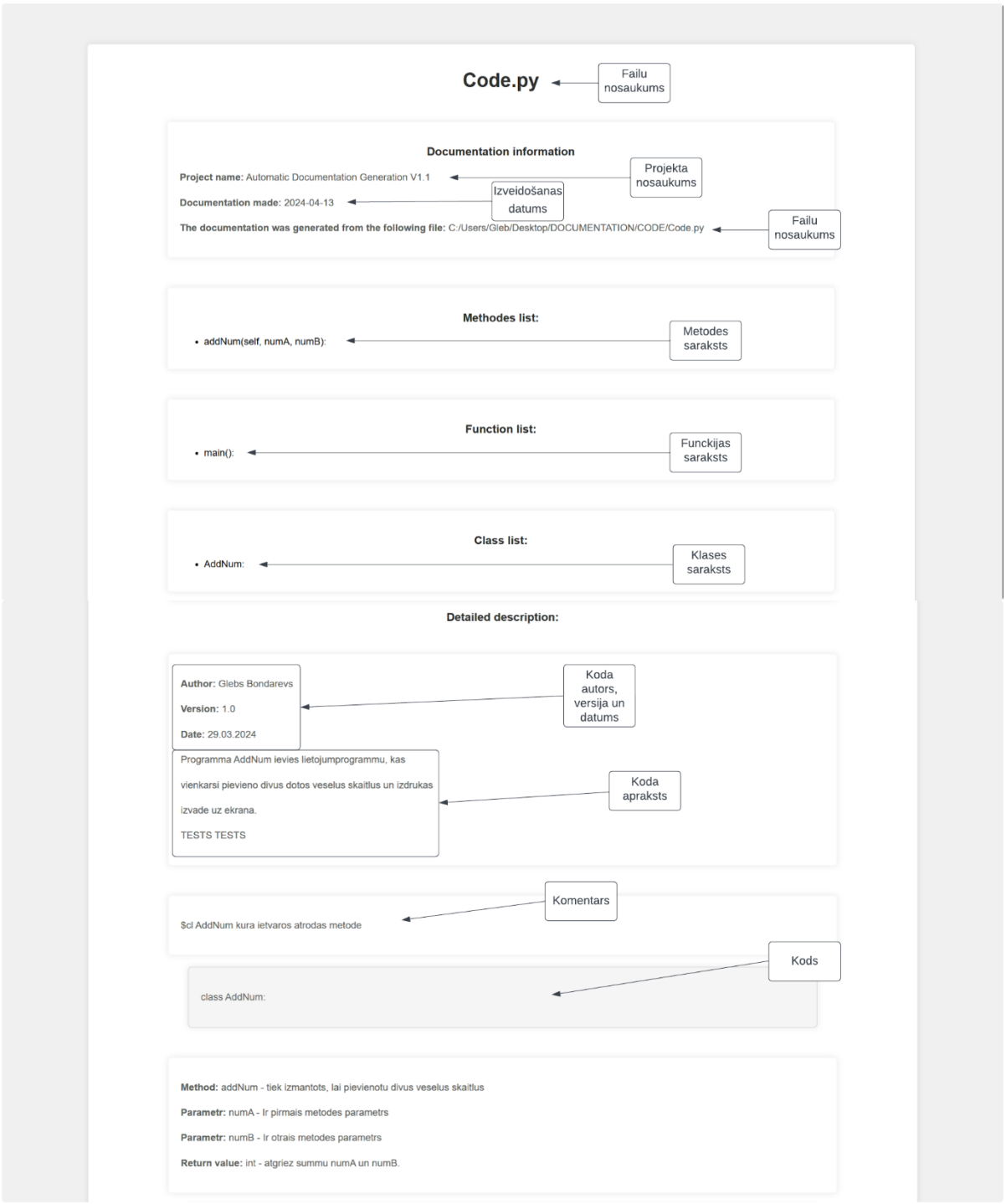
veiksmīgu procesa pabeigšanu. Kad dokumentācija par pirmo koda failu ir izveidotā, tas ņem nākamo koda failu un atgriežas pie procedūras kas bija aprakstīta 2.4.2. sadaļā.

2.6. Izveidotas dokumentācijas demonstrācija

Šajā sadaļā ir demonstrēta izveidotā dokumentācija un tās apraksts. Šīs sadaļas galvenais mērķis ir iepazīties ar kodu dokumentāciju un tās saturu, lai gūtu priekšstatu par to, kā tai vajadzētu izskatīties, kā ari ir pievienots attēls ar dokumentācijas izskatu.

Kā aprakstīts iepriekš, dokumentācija tiek izveidota html formātā un saglabāta izvēlētajā mapē. Dokumentācijas izveidei tiek izmantots 2.4.3. iedaļā aprakstītais koda fragments.

Dokumentācijas saturs ietver informāciju par dokumentētā koda faila nosaukumu, projekta nosaukumu, dokumentācijas izveides datumu, komentāru aprakstu un koda aprakstu. Piedāvātā attēlā redzams, ka komentāri un koda fragmenti ir dokumentēti:



2.2. att. Izveidotas dokumentācijas saturs pēc eksperimentāla faila

3. RISINĀJUMA PĀRBAUDES DAĻA

Šajā nodaļā aprakstīta izstrādātā dokumentācijas ģenerēšanas risinājuma praktiskā validācija reālā izstrādes projektā, izmantojot definētus validācijas kritērijus, lai noteiktu ietekmi uz izstrādes procesu, īpašu uzmanību pievēršot ģenerēšanas ātrumam un dokumentācijas kvalitātei. Tiek veikts manuālās un automātiskās dokumentācijas ģenerēšanas salīdzinājums.

3.1. Risinājuma pārbaudes daļa kritēriju definēšana

Lai noteiktu un novērtētu automatizētās dokumentācijas izveides ietekmi uz izstrādes procesu, tiek definētas divas kritēriju grupas: dokumentācijas izveides un atjaunināšanas laiks un izveidotās dokumentācijas kvalitāte.

3.1.1. Dokumentācijas izveidošanas un atjaunināšanas ātrums

Pirmajā sadaļā ir aprakstīts un apkopots izstrādes process, dokumentācija un automātiskā dokumentācijas ģenerēšana. No apkopotās informācijas var secināt, cik svarīga projektam ir izstrādes procesa kvalitāte un efektivitāte. Turklāt, tā kā ir svarīgi ievērot saskaņoto grafiku un nodot izstrādāto projektu noteiktajos termiņos, termiņu ievērošana ir svarīgs aspekts visā izstrādes procesā. Tāpēc ir jārūpējas par to, lai dokumentācija tiktu sagatavota pēc iespējas ātrāk.

Šis kritērijs ir vērsts uz dokumentācijas izveides un atjaunināšanas ātrumu, lai noteiktu, kura veida dokumentācija tiek izveidota ātrāk. Risinājuma testēšanas daļā vispirms tiek veikts eksperiments, lai noteiktu laiku, kas nepieciešams dokumentācijas izveidei, un pēc tam, pēc projekta koda atjaunināšanas, tiek veikta dokumentācijas atjaunināšanas procedūra, lai noteiktu laiku, kas nepieciešams šā procesa pabeigšanai. Abi eksperimenti tiek veikti, izmantojot manuālu dokumentēšanas metodi un automatizētu dokumentēšanas risinājumu.

Analizējot rezultātus, kas iegūti katram dokumentācijas veidam, piemēram, manuālo metožu sagatavošanas laiku, var izdarīt secinājumus par to ietekmi uz izstrādes procesu. Tādi paši secinājumi tiek izdarīti arī par atjauninājumiem un citiem izstrādes veidiem, lai noteiktu katras metodes ietekmi uz izstrādes procesu. Tiek izdarīti arī secinājumi par metodes ietekmi uz izstrādes procesu un par to, kuru metodi ieteicams izmantot.

3.1.2. Dokumentācijas kvalitāte

Daudzās mūsdienu darba jomās dokumentācija tiek izmantota katru dienu, lai dokumentētu procesus, nodotu zināšanas, sazinātos ar kolēģiem, samazinātu kļūdas un uzlabotu efektivitāti, kvalitāti un produktivitāti. Viena no šādām jomām ir informācijas tehnoloģijas, kur dokumentācija tiek plaši un bieži izmantota, piemēram, izstrādes procesā.

Kvalitatīva dokumentācija ir nepieciešama arī, lai uzlabotu izstrādes darba un procesu efektivitāti, kvalitāti un produktivitāti. Koda dokumentācijai jābūt kvalitatīvai, jo tā sniedz izstrādātājiem svarīgu informāciju par kodu un tā darbību, palīdz jaunajiem izstrādātājiem apgūt projektu un samazina kļūdu skaitu citos izstrādes posmos.

Eksperimenta daļa par dokumentācijas kvalitāti tiek veikta ar manuāli un automātiski izveidotu dokumentāciju. Pamatojoties uz izveidoto dokumentāciju, tiek novērtēta satura precizitāte un detalizētība, satura pilnīgums un dokumentācijas saprotamība. Turpmāk minētie kritēriji palīdz noteikt, kura dokumentācija un tās metode ir kvalitatīvāka. Tika veikta ekspertu aptauja, lai salīdzinātu, kurai dokumentēšanas metodei tiek dota priekšroka izstrādes procesā. Kad būs iegūti aptaujas rezultāti, būs iespējams apkopot un izdarīt secinājumus par veiktajiem eksperimentiem un šī kritērija un dokumentācijas veida ietekmi uz izstrādes procesu.

3.2. Eksperimentālas vides apraksts

Šajā nodaļā ir aprakstīts, kas nepieciešams, lai veiktu eksperimentus 3.1. sadaļā izklāstīto kritēriju pārbaudei. Tajā ir aprakstīts, kas nepieciešams eksperimenta veikšanai, kāda programmēšanas valoda ir nepieciešama projektam, lai varētu piemērot risinājumu, kādam nolūkam projekts tiek izmantots un kur to var atrast.

3.2.1. Prasības eksperimentā datiem

Eksperimenta ietvaros izstrādes projektam jābūt rakstītam vienā konkrētā programmēšanas valodā. Izvēlētajam projektam jābūt pieejamam 1 vai vairākos kodu failos. Tā kā eksperimenta laiks, visticamāk, tiks patērēts manuālai dokumentēšanai, lai ietaupītu laiku citām eksperimenta daļām, dokumentēšanai no projekta tiek izdalīts koda fragments vai vairāki koda faili. Projektam jāizvēlas konkrēta programmēšanas valoda, jo risinājumu var palaist un analizēt tikai vienā programmēšanas valodā.

Pamatojoties uz 3.1. sadaļā definētajiem kritērijiem, tika veikti divi eksperimenti, izmantojot reālu izstrādes projektu. Vispirms tiek veikta manuāla dokumentācijas izveide, pēc tam automātiska dokumentācijas izveide un tad dokumentācijas analīze un novērtēšana. Lai izmērītu dokumentācijas izveides laiku, tiek izmantots vienkāršs hronometrs, un manuālai dokumentācijas izveidei tiek izmantota datorā iebūvēta programmatūra. Šī ir vienkārša pieeja, lai izmērītu laiku, kas vajadzīgs, lai manuāli izveidotu un atjauninātu dokumentāciju. Automātiskai dokumentācijas izveidei un atjaunināšanai tiek izmantota Python "time" bibliotēka. Izmantojot šādu bibliotēku, ir iespējams precīzi noteikt, cik daudz laika nepieciešams dokumentācijas izveidei vai atjaunināšanai. Izstrādātajā risinājumā ir izmantota "time" bibliotēka, un tā ir izmantota šādā veidā:

```
import time
start_time = time.time()
-----------------------------------------
# Dokumentācijas izveidošanas kods
-----------------------------------------
end_time = time.time()
execution_time = end_time - start_time
```

Izmantojot šo pieeju, ir iespējams iegūt datus par laiku, kurā tiek veidota dokumentācija - gan manuāli, gan automātiski. Šādā veidā tiek pētīta dokumentācijas ietekme uz izstrādes procesu.

3.2.2. Programmēšanas valoda izvēle tas apstrādei

3.2.1. sadaļā aprakstīts, ka, lai eksperimenti būtu veiksmīgi izpildīts, izstrādes projekts ir jāraksta konkrētā programmēšanas valodā. Šajā iedaļā ir aplūkots, kura programmēšanas valoda projektam ir jāatbalsta un kāpēc tiek izmantota tieši šī valoda.

Tā kā darba galvenais mērķis ir izpētīt, kā šis risinājums ietekmē izstrādes procesu: nav svarīgi, cik daudz un kāda programmēšanas valoda jāizmanto apstrādei. Gala rezultāts būs dokumentācijas izveide, lai novērtētu šo ietekmi. Tādējādi visa automātiskās dokumentācijas ģenerēšanas algoritma izstrādes pamatā bija Python valodas sintakse. Python tika izvēlēts kā apstrādes programmēšanas valoda automātiskās dokumentācijas ģenerēšanas šādu iemeslu dēļ: pirmkārt, šai valodai ir vienkārša sintakse, tā ir viegli lasāma, tāpēc ar lasīšanas uzdevumu var viegli tikt galā gan algoritms, gan cilvēks, un funkcijas, metodes un citi sintaktiskie elementi ir viegli

identificējami. Otrkārt, Python ir ļoti izplatīta valoda, un nav grūti atrast uzņēmumus, kas izstrādā projektus šajā valodā. Tas ir svarīgi tajā eksperimenta daļā, kur projekta beigās ir jāsagatavo dokumentācija, analīze un secinājumi par projekta darbību. Un, treškārt, izpētot pieejamos rīkus, ir iespējams redzēt, kā Python tiek izmantots dažādos rīkos, kas nozīmē, ka šī valoda ir pieprasītāka un to var droši izmantot.

Tāpēc eksperimenta vajadzībām tiek izmantoti Python valodā rakstīti projekti, lai izstrādātais risinājums varētu analizēt, apstrādāt un izveidot dokumentāciju no projekta koda failiem. Programmēšanas valodas izvēle apstrādei neietekmē rezultātus, jo tiek vērtēts tikai dokumentācijas izveides ātrums un tās kvalitāte.

3.2.3. Struktūras tehniskais nodrošinājums

Automātiskā dokumentācijas ģenerēšanas risinājuma pielietošanai un tas izmantošanas ietekmes izpētīšanā uz izstrādes procesu tiek izmantots projekts, kas ir uzrakstīts Python valodā. Pirms eksperimentu veikšana kura ietvaros tiek veidota dokumentācijās izveidošana gan manuāla, gan automātiskā veida, ir veikta komentāru rediģēšanā kodā, lai tas komentāri atbilsta šādā algoritma darbībai. Komentāru veidi tiek aprakstīti 2.4.3. sadaļā. Projekts un tas izveidotas dokumentācijas tiek publicētas tīmekļa vietnē GitHub. Projekts kurš tiek izmantots abos eksperimenta laika un uzrakstīts Python valoda tiek publicēts: https://github.com/glebsbondarevsRTU/PyAGDoc. Dokumentācija kas ir izveidota pirmā eksperimentā laika un tiek veikta manuāla veida tiek publicēta: https://github.com/glebsbondarevsRTU/Manuala_dokumentacija. Kā ari dokumentācija kas ir izveidota pirmā eksperimentā laika un tiek veikta automātiskā izmantojot iepriekš izstrādātu risinājumu, tiek publicēta: https://github.com/glebsbondarevsRTU/Automatiska_dokumentacija. Dokumentācijas izveidošanas ātruma noteikšana ir veikta izmantojot hronometru un Python bibliotēku, kuri bija aprakstīti 3.2.1. sadaļā. Lai sasniegtu otra eksperimenta rezultātus tiek nodrošināta aptaujas izveidošana, kas satur 10. jautājums par dokumentācijas izveidotas manuāla un automātiskā veida. Aptaujā piedalies eksperti no IT jomas, kuri palīdzes noskaidrot kāda dokumentācijas veids ir precīzāks.

3.3. Izvelēta projekta un sagatavošanas apraksts eksperimenta daļai

Kā aprakstīts 3.2. sadaļā, eksperimentos jāizmanto reāls izstrādes projekts, lai izpētītu automātiskā dokumentācijas izveides risinājuma ietekmi uz izstrādes procesu un noteiktu, kuru dokumentācijas izveides metodi ieteicams izmantot izstrādes procesā, lai uzlabotu vai paātrinātu izstrādes procesu.

Tādējādi risinājuma testa daļai ar uzņēmuma palīdzību tiek atrasts un iegūts projekts. Lai ietaupītu laiku un ievērotu termiņus, kā arī netraucētu citus attīstības procesus uzņēmumā, eksperimentam tiek izvēlēts neliels projekts. Projekts ir rakstīts Python valodā un sastāv no 5 failiem. Izmantotais projekts ir ievietots GitHub timekļa vietnē, saite uz to ir aprakstīta un pievienota 3.2.3. sadaļā. Uz izvēlētā projekta pamata tiek uzsākts darbs pie komentāru pielāgošanas kodā, lai ar automātiskās ģenerēšanas risinājuma palīdzību sagatavotu projektu dokumentēšanai. Tiek veikti arī eksperimenti, izmantojot šo projektu.

Turpmākajās sadaļās ir sīkāk izpētīts un aprakstīts izvēlēts projekts un tas apraksts, kā ari ka komentāri tiek pielāgoti risinājuma vajadzībām pirms eksperimenta veikšanai.

3.3.1. Izvelēta projekta apraksts

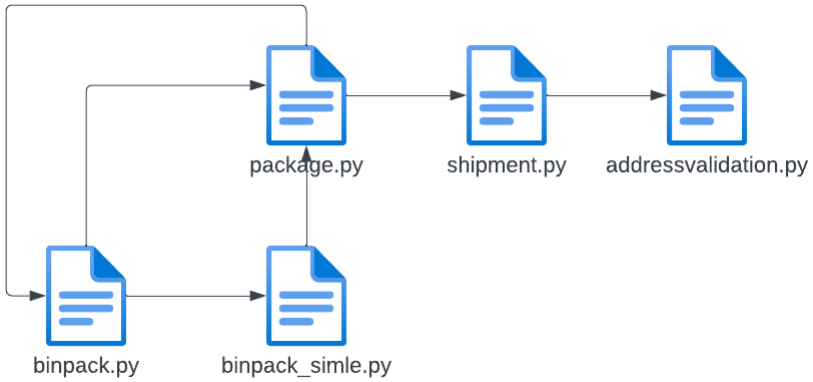
Projekta izvēle ir svarīgs solis risinājuma testa daļā, jo izvēlētais projekts ir pamats manuālajai un automātiskajai dokumentācijai, kā arī diviem svarīgiem eksperimentiem, kuru rezultāti tiek izmantoti secinājumos, lai izpētītu automātiskās dokumentācijas ietekmi uz izstrādes procesu.

Šajā sadaļā ir iekļauts izvēlētā projekta apraksts. Vairāk uzmanības tiek pievērsta failu struktūrai, izmantoto bibliotēku un komentāru pielāgošanas aprakstam. Ir sniegts arī koda fragments no katra projekta faila.

Izvēlētā projekta aprakstu var sākt, aprakstot projekta failus un struktūru. Kā aprakstīts 3.3. sadaļā, izvēlētais projekts sastāv no 5 Python valodā rakstītiem failiem ar paplašinājumu ".py". Visi projekta faili atrodas vienā mapē "Shipping" (sūtījumi) un ir novietoti uz datora darbvirsmas, lai tiem būtu viegli piekļūt. Tātad nākamais uzdevums ir aprakstīt failus, lai noskaidrotu, kāda ir projekta struktūra un cik metodes, klases vai funkcijas tajā tiek izmantotas. Tas ir nepieciešams, lai pēc eksperimenta varētu novērtēt izveidoto dokumentāciju. Tātad izvēlētais projekts sastāv no šādiem Python failiem:

- **“binpack.py”** – Pirmais Python fails no izvēlētā projekta sastāv no 2 funkcijām. Tiek izmantota viena bibliotēkā: time. Failā ir 65 rindas ar kodu un komentārus. Turpmākajā darba ietvaros fails tiek apzīmēts ar – Nr.1.
- **“binpack_simple.py”** – Otrais Python fails no izvēlētā projekta sastāv no 6 funkcijām, 1 metodi un 5 metodēm. Tiek izmantotas 2 bibliotēkas: time un random. Failā ir 251 rindas ar kodu un komentārus. Turpmākajā darba ietvaros fails tiek apzīmēts ar – Nr.2.
- **“shipment.py”** - Trešais Python fails sastāv no 4 klasēm un 27 metodēm kopumā. Tiek izmantotas 2 bibliotēkas: unittest un math. Failā ir 304 rindas ar kodu un komentārus. Turpmākajā darba ietvaros fails tiek apzīmēts ar – Nr.3.
- **“package.py”** - Ceturtais Python fails no izvēlētā projekta sastāv no 2 klasēm, 20 metodēm un 2 funkcijām. Tiek izmantotas 2 bibliotēkas. Failā ir 334 rindas ar kodu un komentārus. Turpmākajā darba ietvaros fails tiek apzīmēts ar – Nr.4.
- **“addressvalidation.py”** – Un piektais Python fails no izvēlētā projekta sastāv no 1 klasē un 5 metodēm un 1 funkcijās. Tiek izmantota viena bibliotēka: unittest. Failā ir 113 rindas ar kodu un komentārus. Turpmākajā darba ietvaros fails tiek apzīmēts ar – Nr.3.

Pēc projekta faila apraksta ir lietderīgi aplūkot arī izmantotā projekta struktūru, kas palīdzēs noskaidrot, kuri koda faili izmanto viens otru, kā arī palīdzēs otrajā eksperimentā, novērtējot koda radīto dokumentāciju. Tātad izvēlētajam projektam ir šāda faila struktūra:



3.1 att. Izvelēta projekta failu struktūra

3.1. attēlā ir parādīta projekta struktūra, kas sastāv no 5 failiem. Galvenais fails ir "binpack.py", kas ir saistīts ar pārējiem diviem projekta failiem. Tiek izmantota 'Package' klase no 'package.py' faila, kā arī funkcijas un metodes no 'binpack_simple.py' faila. Turklāt fails 'binpack_simple.py' ir saistīts ar failu 'package.py', un tajā tiek izmantota Package klase. Faili 'package.py', ko izmanto pārējos divos projekta failos, izmanto papildu funkcijas un metodes no failiem 'binpack.py' un 'shipment.py'. Pēdējā failā 'shipment.py' tiek izmantotas funkcijas, klases un metodes no faila 'addressvalidation.py'.

Projekta apraksts ir nepieciešams, lai noskaidrotu un definētu, kādi faili tiek izmantoti projektā, kādas funkcijas, metodes vai klases tiek izmantotas. Šāds apraksts palīdz noskaidrot un novērtēt gan eksperta manuāli veidotās dokumentācijas izmantošanu, gan risinājumam izveidotās automatizētās dokumentācijas izmantošanu.

3.3.2. Projekta pielāgošanā risinājumam

Pirms pāriet pie eksperimentālās daļas, ir svarīgi pielāgot izmantoto projektu tās prasībām. Parasti dokumentēšanai izmantotajiem rīkiem, piemēram, DoxyGen vai JavaDoc, ir savi komentāru formāti, kas palīdz rīkam identificēt komentārus aprakstošo informāciju. Piemēram, funkcijas, metodes vai klases apraksts, ģenerētā koda autors un datums, kā arī cita nepieciešamā informācija.

Pamatojoties uz darba otrajā nodaļā veikto pētījumu, tika izstrādāts algoritms, kas balstīts uz komentāru veidiem, kuri aprakstīti 2.5.3. nodaļā un 2.5.3. tabulā. Turpmākajā tekstā sniegts mazāk detalizēts apraksts par to, kā tika veikta rediģēšana, un parādīts attēlu izskats pirms un pēc komentāru pielāgošanas.

Tā kā projekta koda failos jau bija labi noformēti komentāri angļu valodā, galvenais uzdevums bija tos nedaudz rediģēt, lai tie atbilstu risinājuma prasībām. Kā piemēru var minēt nelielu koda fragmentu no faila binpack_simple.py. Pirms izmaiņu veikšanas šis koda fragments izskatījās šādi:

```
"""binpack() Function to check if bin is None, if it is empty put standart value """
def binpack(packages, bin=None, iterlimit=5000):
    """
    Packs a list of Package() objects into a number of equal-sized bins.

    Returns a list of bins listing the packages within the bins and a list of packages which can't be
    packed because they are to big."""
    if not bin:
        bin = Package("600x400x400")
    return allpermutations(packages, bin, iterlimit)
```

3.2. att. Komentāru fragments no play.py faila, pirms pielāgošanai

Metodes apraksts sākumā tiek pielāgots, izmantojot aprakstītos komentāru veidus. Metodes nosaukums un apraksts tiek ievietots aiz identifikatora "$method", lai risinājums varētu identificēt metodi un izmantot to aprakstā, kas vēlāk tiks ievietots dokumentācijā. Iepakojuma argumenta apraksts tiek ievietots aiz identifikatora "$par", lai algoritmam būtu vieglāk to atrast. Pēc tam šis apraksts tiek ievietots aiz identifikatora "$return", lai apzīmētu atgriešanas vērtību, kas tiks izmantota tādam pašam mērķim kā iepriekšējā teikumā. Komentāri, kas satur tikai vispārīgu aprakstošu saturu un neatbilst konkrētam funkcijas parametram, netiek ņemti vērā. Visi pārējie komentāri kodā ir jāpielāgo saskaņā ar to pašu principu. Nākamajā attēlā redzams, kā izskatās komentāra fragments pēc pielāgošanas:

```
"""
    $method binpack() Function to check if bin is None, if it is empty put standart value
"""
def binpack(packages, bin=None, iterlimit=5000):
    """
    $par Package Packs a list of objects into a number of equal-sized bins.

    $return Returns a list of bins listing the packages within the
    bins and a list of packages which can't be packed because they are to big.
    """
    if not bin:
        bin = Package("600x400x400")
    return allpermutations(packages, bin, iterlimit)
```

3.3. att. Komentāru fragments no play.py faila, pēc pielāgošanai

Pēc visu failu komentāru iestatīšanas mēs varam pāriet pie testa daļas, kurā tika veikti divi eksperimenti, kas saistīti ar dokumentēšanas ātrumu un dokumentēšanas kvalitāti.

3.4. Eksperimentu daļa

Šajā nodaļā sīki izklāstīts, kā tika veikts katrs eksperiments, kā tika veikti mērījumi, kā tika apkopoti rezultāti un izdarīti secinājumi. Eksperimentālajā daļā tika veikti divi eksperimenti. Pirmajā eksperimentā uzmanība tika pievērsta dokumentācijas izveides laikam, bet otrajā eksperimentā - izveidotās dokumentācijas kvalitātei.

3.4.1. Eksperimenta Nr.1. apraksts

Eksperimenta mērķis ir noskaidrot, cik ilgs laiks ir nepieciešams, lai dokumentāciju izveidotu manuāli, un kādu ietekmi šāds dokumentācijas izmantošanas veids var atstāt uz izstrādes procesu.

Priekš eksperimenta izpildīšanai tiek izmantots izvelēts Python projekts un iebūvētu hronometru Google timekļa vietne. Eksperimentā piedalās 5 informācijas tehnoloģijas komandas specialisti no "Mediterranean Shipping Company" (MSC), kas eksperimentā laika tiek saukti par – ekspertiem. Eksperimenta dalībnieki bija atrasti izmantojot lietotni Microsoft Teams. Eksperimentā piedalās:

- Eksperts numurs 1. – Vecāks informācijas tehnoloģijas specialists, 36 vecumā. Ir neliels priekšstats par izvelēto Python projektu eksperimentam.
- Eksperts numurs 2. – Vecāks informācijas tehnoloģijas specialists, 40 vecumā. Nav priekšstata par izvelēto Python projektu eksperimentam.
- Eksperts numurs 3. – Jaunākais informācijas tehnoloģijas specialists, 26 vecumā. Ir neliels priekšstats par izvelēto Python projektu eksperimentam.
- Eksperts numurs 4. - Jaunākais informācijas tehnoloģijas specialists, 22 vecumā. Nav priekšstata par izvelēto Python projektu eksperimentam.
- Eksperts numurs 5. - Jaunākais informācijas tehnoloģijas specialists, 29 vecumā. Nav priekšstata par izvelēto Python projektu eksperimentam.

Tādējādi eksperimenta laikā piedalījās 2 vecākie eksperti un 3 jaunākie eksperti vecumā no 25 līdz 40 gadiem, kuri palīdzēja sastādīt kodu dokumentāciju. Eksperimenta sekmīgu pabeigšanu un visu prasību izpildi uzraudzīja šī darba autors.

Tā kā manuālajai dokumentācijai nav nepieciešams, lai eksperti to veidotu konkrētā sistēmā, lietojumprogrammā vai datorā, eksperti paši izvēlas, kurā ierīcē un lietojumprogrammā rakstīt dokumentāciju. Katram ekspertam pirms eksperimenta veikšanas tiek sniegta informācija par eksperimentu. Ekspertiem tiek paskaidrots, ka:

1. Jāsagatavo projektu un tās izstrādes vidi, pirms dokumentācijas izveidošanai un laika mērīšanai.
2. Jāieliedz Google iebūvētu hronometru pirms dokumentācijas veidošanai.

3. Jāizmēra katra failā dokumentācijas izveidošanas laiku (Jo faili ir dažādā apjoma).
4. Jāizveido dokumentācijas visam izstrādes failiem.
5. Pēc eksperimenta pabeigšanai jāpieraksta iegūtie rezultāti un jāiesniedz dokumentācijas.

Lai nodrošinātu, ka visi pirmā eksperimenta laikā savāktie dati ir precīzi un galvenokārt ticami, tika izmantota Google laika noteikšanas programma. Tās izmantošanas piemērs ir parādīts 3.4. attēlā:

3.4. att. Mērīšanas lietotne

Eksperimenta laika mērījumi tiek veikti, izmantojot Google mērījumu programmu, un mērījumi katrai dokumentācijai tiek veikti atsevišķi. Tas nozīmē, ka pēc katras dokumentācijas izveides tiek reģistrēts rezultāts un mērījumi tiek sākti no jauna. Kopumā katram ekspertam ir jāizveido 5 dokumentācijas, pa vienai katrai dokumentācijai, un jāveic 5 dažādi mērījumi, kas saistīti ar dokumentāciju izveidi. Eksperimenta beigās jābūt 25 dažādiem mērījumiem un 5 dokumentācijām katram failam, no kurām tikai viena tiek ņemta kvalitātes novērtēšanai.

3.4.2. Eksperimenta Nr.1 rezultātu apkopojums

Eksperimenta norises laikā pēc katras izveidotai dokumentācijai tiek veikti mērījumu un tās rezultātu fiksēšana. Zemāk, piedāvātā tabulā 3.1. tabulā, ir pieejama informācija par iegūtiem rezultātiem eksperimenta laika no 5 ekspertiem. Tas rezultāti ir pierakstīti minūtes, jo sekundes patērētas uz to izstrāde neietekmēs globāli uz izstrādes procesu.

3.1. tabula

Manuālas dokumentācijas izveidošanas eksperimenta rezultāti

Eksperta numurs	Nr.1.	Nr.2.	Nr.3.	Nr.4.	Nr.5.
1	17 min	31 min	40 min	45 min	27 min
2	15 min	30 min	42 min	43 min	25 min
3	20 min	34 min	41 min	50 min	23 min
4	16 min	30 min	37 min	46 min	30 min
5	18 min	39 min	38 min	44 min	29 min

Lai apkopotu rezultātus, aprēķinātas vidējās vērtības katram dokumentācijas izveides laikam. Lai aprēķinātu vidējo dokumentācijas izveides laiku, tiek izmantota programma Excel, tās funkcijas un pirmā eksperimenta dati. Tabulā ir iekļauta informācija par dokumentācijas izveidi kodu failiem un šī faila lielumu. Vidējās vērtības palīdz noskaidrot, cik ilgs laiks nepieciešams, lai izveidotu katra faila dokumentāciju. Šīs vidējās vērtības ir iegūtas un parādītas 3.2. tabulā.

3.2. tabula

Vidēji iegūtie rezultāti eksperimenta laikā

	Nr.1.	Nr.2.	Nr.3.	Nr.4.	Nr.5.
Vidējais laiks	17,2 min	32,8 min	39,6 min	45,6 min	26,8 min
Faila apjoms	65 rindas	251 rindas	304 rindas	334 rindas	113 rindas

Aplūkojot ģenerēto tabulu un tās rezultātus, var secināt, ka manuālās dokumentācijas izveide ir atkarīga no faila lieluma. Šis detalizētais pētījums tiek veikts pirmā eksperimenta beigās.

Ir arī pierādīts, ka projekta kods un saturs ļoti bieži tiek rediģēts un pielāgots prasībām izstrādes posmā, tāpēc dokumentācija ir jāatjaunina ātri un bieži, lai nezaudētu savu atbilstību projektam. Tādējādi pirmajā eksperimentā tiek mērīts arī dokumentācijas atjaunināšanai nepieciešamais laiks.

Pirms eksperimenta veikšanai tiek veikta katra eksperta informēšana par eksperimentu. Tiek paskaidrots, ka ekspertiem ir:

- Jānorediģē jebkuru koda rindu vai jāpievieno jaunu funkciju vai metodi katram projekta failam (bez laika mērīšanai).
- Jāieslēdz Google iebūvētu hronometru pirms dokumentācijas atjaunināšanai.
- Jāizmēra katra failā dokumentācijas atjaunināšanas laiku (Jo faili ir dažādā apjoma).
- Jāatjaunina katru dokumentāciju visam izstrādes failiem, lai tas atbilstu projektam un dokumentācijā būtu aktuālā.
- Pēc eksperimenta pabeigšanai jāpieraksta iegūtie rezultāti un jāiesniedz dokumentācijas.

Eksperimentā piedalās tie paši eksperti, kas bija eksperimenta pirmajā daļā, un tajā tiek izmantota projekta ietvaros izveidotā dokumentācija un faili. Eksperimenta laikā tiek veikti mērījumi un reģistrēti rezultāti pēc katras dokumentācijas atjaunināšanas. Turpmāk piedāvātajā 3.3. tabulā ir apkopoti eksperimenta laikā iegūtie rezultāti no 5 ekspertiem.

3.3. tabula

Manuālas dokumentācijas atjaunināšanas eksperimenta rezultāti

Eksperta numurs	**Nr.1.**	**Nr.2.**	**Nr.3.**	**Nr.4.**	**Nr.5.**
1	3,2 min	3 min	3,5 min	3,1 min	3,6 min
2	4,1 min	4,3 min	4,5 min	4 min	3,9 min
3	6 min	5,5 min	6,1 min	5 min	4,9 min
4	4,2 min	4,4 min	5 min	4,3 min	3,5 min
5	4,5 min	5 min	4,5 min	4,2 min	4,1 min

Pēc atjaunināšanas procedūras dati tiek apkopoti, lai būtu viegli izdarīt secinājumus un veidot grafikus. Izmantojot Excel, tiek izveidota tabula un aprēķināts vidējais dokumentācijas atjaunināšanas laiks. Rezultāti ir apkopoti 3.4. tabulā.

3.4. tabula

Vidēji iegūtie rezultāti eksperimenta laikā

	Nr.1.	Nr.2.	Nr.3.	Nr.4.	Nr.5.
Vidējais laiks	4,4 min	4,5 min	4,7 min	4,1 min	4 min
Faila apjoms	65 rindas	251 rindas	304 rindas	334 rindas	113 rindas

Pirmajā eksperimentā tiek salīdzināti divi dokumentācijas izveides un atjaunināšanas veidi - manuālā un automātiskā. Tā kā dokumentācijas manuāla izveide un atjaunināšana jau ir pētīta un ir iegūti svarīgi rezultāti, ir nepieciešams iegūt arī rezultātus, kas saistīti ar automātiskās dokumentācijas izveides risinājumu izmantošanu šādam izstrādes projektam.

Nākamajā eksperimenta daļā tika veikti 10 mērījumi, kas saistīti ar automātisko dokumentācijas izveidi. Eksperimentā piedalās tādi paši eksperti no eksperimenta pirmās daļas. Priekš laika nomērīšanai ir izmantota Python iebūvētā bibliotēka, kas bija aprakstītā iepriekš, 3.2.1. sadaļā. Laika nomērīšanā sākas ar algoritma palaišana un beidzas ar visu dokumentācijas izveidošanu un tās novietošanu mapē. Tā kā risinājums var izveidot dokumentāciju visiem koda failiem vienā brīdi, tiek veikti 2 mērījumi katram ekspertam, veidojot dokumentācijas. Var izveidot hipotēzi kā rezultāti pēc otra mēģinājumā būs atraks nekā šāds risinājums ir izmantos pirmo reizi. Pirms eksperimenta veikšanai tiek veikta katra eksperta informēšana par eksperimentu. Tiek paskaidrots, ka ekspertiem:

- Nav jārūpējas par laika nomērīšanu (jo algoritms pats izmēra to),
- Pēc algoritma palaišanai ekspertam jāievada nosaukumu priekš izveidotai dokumentācijai.
- Ekspertam jāizvēlas mapi ar projektu un mapi priekš dokumentācijas saglabāšanas vietai
- Jāpieraksta iegūtie mērījumi pēc divām mēģinājumam.

Pēc mērījumu veikšanas rezultāti tiek apkopoti un ievietoti vienā tabulā, izmantojot automātisku dokumentācijas izveides risinājumu. Tās rezultātus apkopojums atrodas 3.5. tabulā:

3.5. tabula

Automātiskas dokumentācijas izveidošanas eksperimenta rezultāti

Eksperta numurs	Dokumentācijas izveidošana pirmais mēģinājums	Dokumentācijas izveidošana otrais mēģinājums
1	40 sekundes	35 sekundes
2	35 sekundes	30 sekundes
3	33 sekundes	31 sekundes
4	45 sekundes	39 sekundes
5	39 sekundes	34 sekundes

Lai apkopotu rezultātus, tika izmantota Excel funkcija, lai aprēķinātu vidējo vērtību katram izmēģinājumam. Iegūtās vērtības ir parādītas 3.6. tabulā.

3.6. tabula

Vidēji iegūtie rezultāti eksperimenta laikā

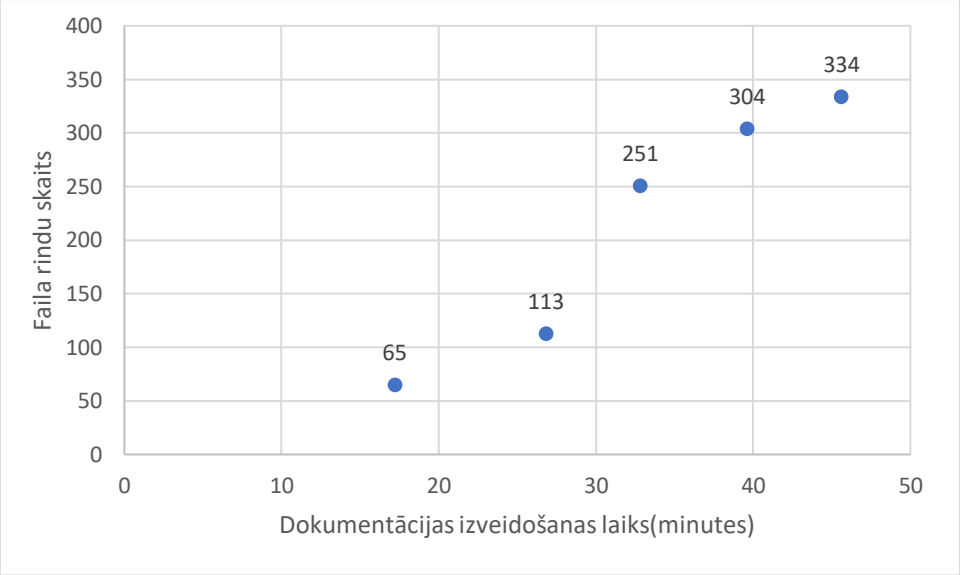
	Dokumentācijas izveidošana pirmais mēģinājums	Dokumentācijas izveidošana otrais mēģinājums
Vidējais laiks	38,4 sekundes	33,8 sekundes

Šajā eksperimenta daļā netiek veikta dokumentācijas atjaunināšana automātiskajā režīmā un laika mērīšana. Tā kā dokumentācijas izveides un atjaunināšanas procedūra izstrādātajā risinājumā tika veikta tādā pašā veidā, nav jēgas vēlreiz veikt tās pašas mērījumu procedūras.

3.4.3. Eksperimenta Nr.1 secinājumi

Eksperimentā numurs 1, pirmā daļā no kopēja eksperimenta, tiek veikta izpēte par to, cik laika specialisti tērē uz manuālas dokumentācijas izveidošanu. Veicot eksperimentu kur piedalījās 5 eksperti bija iegūtas 25 vērtības un tiek aprakstītas 3.1. tabulā. Apskatot piedāvātu tabulu var izveidot secinājumu, ka atkarīgi no izmantota failu tās izveidošanas laiks ir atšķirīgs. Kā ari laiku uz kuru eksperti patērēja izveidojot visas projekta faila dokumentācijas ir ļoti līdzīgas. Tas izveidošanas laiks atrodas starp 150 un 160 minūtēm. Tas nozīme, kā eksperta projekta zināšanas līmenis un pieredze

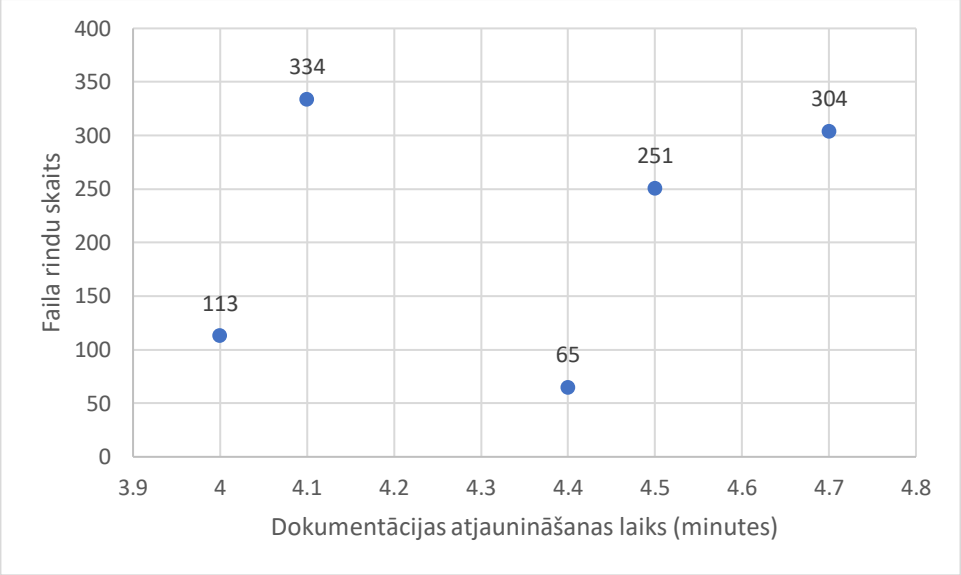
noteiktajā joma ne ietekme uz manuālas izstrādes procesu. Izmantojot iepriekš izveidotu tabulu 3.2., kur ir veikta rezultātu apkopojums un vidēja vērtības aprēķināšanā, tiek izveidotā diagrammā, izmantojot Excel palīdzību. Ar diagrammās palīdzību būs iespēja apskatīt un secināt kāda attiecibā ir faila apjomam un tas dokumentācijas izveidošanas laikam.



3.5. att. Dokumentācijas izveidošanas laika attiecība priekš faila apjomu

Balstoties uz 3.5. attēlu var secināt kā dokumentācijas izveidošanas laiks manuāla veida ir stingri atkarīga no faila rindu apjoma. Eksperimenta laika pirmais fails ar 65 koda rindām bija nodokumentēts vidēji, 5 ekspertiem, par 17,2 minūtēm. 131 koda rindas tiek atrastās 5 faili un tiek nodokumentētas par 26,8 minūtēm. Otrā koda faila tiek nodokumentēti 251 rindas par 32,8 minūtēm. Trešā un ceturtā failā kur koda rindas bija vairāk par 300, tas dokumentācijas izveidošana, ekspertiem, aizņēma vidēji 42,6 minūtes. Tāpēc, jo vairāk kodu rindu failā, jo vairāk laika darbiniekam būs nepieciešams dokumentēšanai. Līdz ar to, var izveidot apkopojumu, ka projekta apjoms stingri ietekme tas dokumentācijas izveidošanas laikam, kā ari liela projekta dokumentēšana aizņems pietiekamu laiku kurš var ietekmēt izstrādes procesa kopumā.

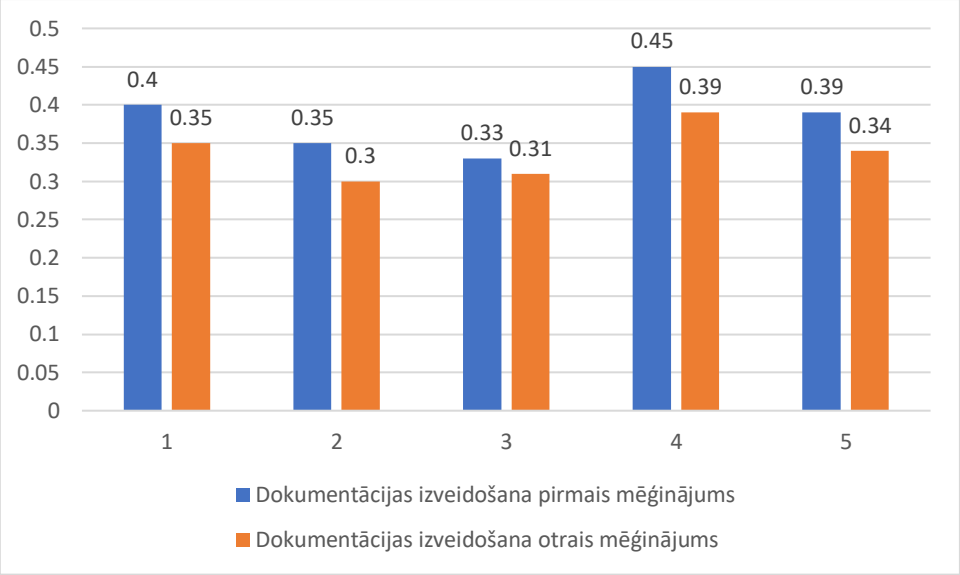
Savukārt, rezultāti, kas iegūti, manuāli atjauninot eksperimenta dokumentāciju, atšķiras no eksperimenta izveides procedūras. Šajā eksperimentā iegūtie rezultāti ir ievietoti 3.3. tabulā, un tālāk tiek aprēķinātas vidējās atjaunināšanas vērtības. Šis vērtību kopums, ņemot vērā faila lielumu, ir aprakstīts 3.4. tabulā. Pamatojoties uz 3.4. tabulu, ir uzzīmēts grafiks, kas parādīts 3.6. attēlā:



3.6. att. Dokumentācijas atjaunināšanas laika attiecība priekš faila apjomu

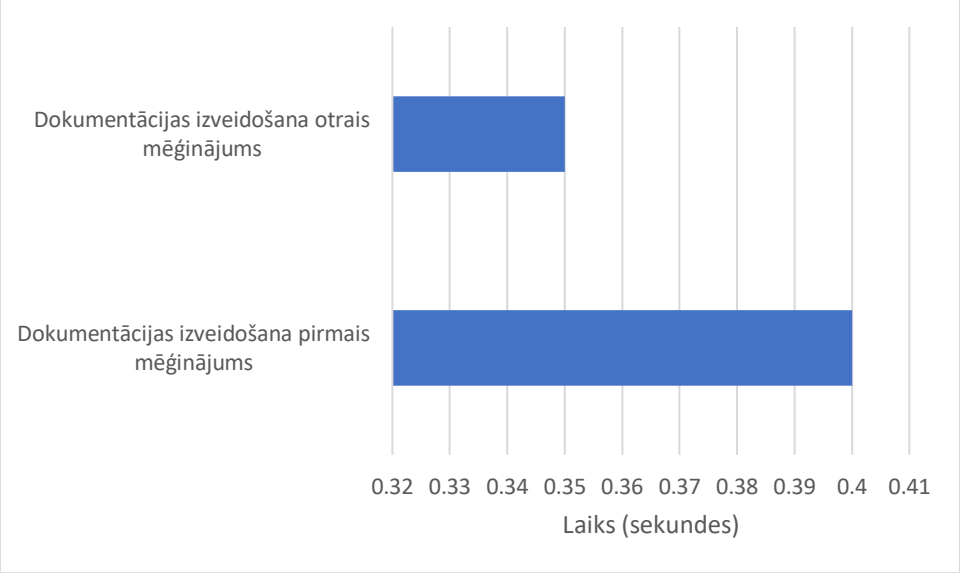
Apskatot izveidotu diagrammu var izveidot secinājumu, ka manuālas dokumentācijas atjaunināšana nav atkarīgā no tas faila apjomā. Tas atjaunināšanas procedūra ir vairāk atkarīga no specialista kurš atjaunina to. Jo balstoties uz iegūtiem vidējam vērtībām, nevar apskatīt rezultātu leģitimitāte. Eksperimenta rezultāti ir ļoti vienādi, tas laiks svārstās no 4 līdz 5 minūtēm. Kā piemēru, tas rezultātu salīdzināšanai var paņemt koda failu Nr.3 un Nr.4. Abiem koda failiem ir gandrīz vienāds rindu skaits. Pirmā gadījuma fails kas sastāv no 334 rindām ir atjaunināts vidēji par 4,1 minūtēm, savukārt fails ar 304 koda rindām bija atjaunināts par 4,7 minūtēm. Tādējādi varam secināt, ka atjaunināšanas procedūras izpildes laiks ir atkarīgs no speciālista un atjaunināšanas jomas, bet ne no projekta apjoma. Turklāt atjaunināšanas izpildes laika nozīme un efektivitāte nav atkarīga no projekta apjoma.

Pirms veikt laika salīdzinājumu starp diviem veidošanas metodēm, ir nepieciešams ari veikt analīze un secinājumu veikšanu priekš automātiskas dokumentācijas izstrādes veidam. Automātiskas dokumentācijas izveidošanas eksperimenta rezultāti tiek novietoti 3.5. tabulā. Tas eksperimenta piedalījās 5 eksperti kuri izveidoja dokumentācijas divas reizes. Tās rezultātu grafiskais attēlojums ir paradīts 3.7. attēlā:



3.7. att. Automātiskā dokumentācijas izveidošanas laiks

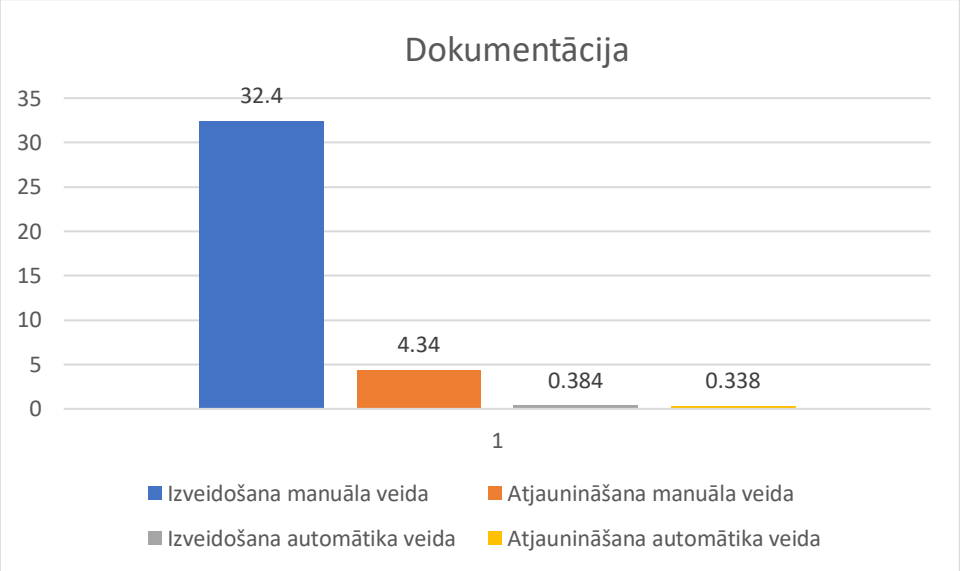
Pamatojoties uz 3.7. attēlu, var secināt, ka dokumentācijas izveides laiks, kad eksperts pirmo reizi saskaras ar izstrādāto risinājumu un nav iepazinies ar dokumentēšanas procedūru, ir nedaudz lielāks nekā otrajā dokumentēšanas izveidošanas mēģinājums. Tas nozīmē, ka izstrādāto dokumentācijas risinājumu eksperti uzskata par skaidru un viegli lietojamu. Pamatojoties uz 3.6. tabulu, tika izveidots arī grafisks attēlojums, lai parādītu atšķirību starp pirmo un otro mēģinājumu. Iegūtais grafiks ir parādīts 3.8. attēlā.



3.8. att. Dokumentācijas atjaunināšanas laika attiecība priekš faila apjomu

Apskatot diagrammu attēlā 3.8. vai secināt kā laika starpība starp pirmo un otru mēģinājumā ir neliela. Līdz ar to šādu rezultātu var uzskatīt par stabilu un precīzu, jo 10 mērījuma laika nebija patērēts uz dokumentācijas izveidošanu mazāk par 30 sekundēm un vairāk par 50 sekundēm. Tā kā dokumentācijas izveidošana un atjaunināšana šādā risinājuma ir veikta vienādi, rezultāti kas ir iegūtas dokumentācijas izveidošanas eksperimenta laika var ari attiekties uz to atjaunināšanas laiku.

Tā kā galvenais darba uzdevums ir saistīts ar manuālas un automātiskas dokumentācijas salīdzinājumu un ietekmi uz izstrādes procesu, turpmākajā diagramma tiek atspoguļota kādu laiku aizņem dokumentācijas izveidošana un atjaunināšana gan manuālā, gan automātiskā veidā.



3.9. att. Dokumentācijas atjaunināšanas laika attiecība priekš faila apjomu

No 3.9. attēla redzams, ka manuālā dokumentācijas izveides un atjaunināšanas metode ir ļoti laikietilpīga salīdzinājumā ar automātisko metodi. Izmantojot primitīvu formulu, tiek aprēķināta ātruma starpība. Dokumentācijas izveides laiks manuālajam veidam ir 84 reizes (32,4 / 0,384 = 84,375) ātrāks nekā automātiskajam veidam. Tā ir ļoti liela atšķirība starp abām metodēm, kas liecina, ka viens veids ir ātrāks par otru. Tādējādi dokumentācijas izveide manuālā veidā nav ieteicamākā un atbilstošākā metode, jo izstrādes procesam ir jābūt ātram, efektīvam, un izstrādes procesam un uzņēmumiem ir svarīgi ievērot termiņus.

Savukārt, atjaunināšanas laiks manuālajā režīmā ir 13 reizes (4,34 / 0,338 = 12,84) ātrāks nekā atjaunināt to izmantojot izstrādāto risinājumu. Tā nav liela atšķirība, salīdzinot ar izveides laiku, taču tā parāda, ka viena metode ir ātrāka par otru.

Pamatojoties uz šiem rezultātiem, varam secināt, ka automātiskās metodes gadījumā atjaunināšanas laiks ir stabils, bet manuālās metodes gadījumā laiks ir tieši atkarīgs no atjaunināmā satura. Tāpēc dokumentācijas atjaunināšanai nav ieteicams izmantot manuālo metodi. Tā vietā labāk ir izmantot jebkuru automātisko dokumentācijas ģeneratoru.

Šie rezultāti ir balstīti uz projektu, kas sastāv tikai no 5 failiem. Izstrādājot un dokumentējot projektu, kas satur vairākus failus un koda rindas, tā izveides un atjaunināšanas laiks būs ilgāks. Tāpēc, ja uzņēmumi ir ieinteresēti samazināt šai izstrādes daļai patērēto laiku, vēlamāk būs izmantot automātisku dokumentācijas ģenerēšanu izstrādes procesa laikā. Izmantojot šādu risinājumu, var ietaupīt speciālistu laiku citām darbībām izstrādes procesā, samazināt uzņēmuma izmaksas, paātrināt izstrādes procesu un samazināt IT speciālistu noslogojumu.

3.4.4. Eksperimentā Nr.2. apraksts.

Eksperimenta mērķis ir noskaidrot, kurš no diviem petitam dokumentācijas izveides veidiem ir labākais, detalizētākais un efektīvākais.

Otrajā eksperimentā tika izmantota dokumentācija, kas uzrakstīta/izveidota, izmantojot manuālo dokumentācijas izveides metodi un ar automātiskā dokumentācijas izveides risinājuma palīdzību. Eksperimentā piedalījās 20 IT speciālisti no dažādiem uzņēmumiem, kas palīdzēja noskaidrot, kāds dokumentācijas veids ir kvalitatīvs. Eksperimenta otrā daļa pamāta bija ekspertu aptauja ar anketu.

3.4.5. Eksperimenta Nr.2. rezultātu apkopojums un rezultāti

Galvenais otra eksperimenta uzdevums ir saistīts ar ekspertu intervēšanu un atsauksmju iegūšanu par sagatavoto dokumentāciju. Iegūtās atsauksmes un rezultāti tiek apkopoti tabulā un tālāk analizēti atbilstoši iegūtajām vērtībām. Tiek izmantota izstrādāta anketa, kas sastāv no jautājumiem par dokumentācijas lietojamību, precizitāti, detalizētību, informācijas iegūšanu un saprotamību. Lai veiktu taisnīgu eksperimentu, anketā atspoguļotie dokumentācijas nosaukumi tiek nosaukti "Dokumentācija 1" - manuālajai dokumentācijai un "Dokumentācija 2" - automātiskajai dokumentācijai. Ekspertu aptaujas rezultāti ir apkopoti un attēloti 3.7. tabulā.

3.7. tabula

Anketēšanas iegūtie rezultāti

	Lietojamība		Precizitāte		Detalizācija		Meklēšana		Saprotamība	
Eksperta Numurs	**Nr.1**	**Nr.2**	**Nr.1**	**Nr.2**	**Nr.1**	**Nr.2**	**Nr.1**	**Nr.2**	**Nr.1**	**Nr.2**
1	4	3	4	5	5	3	3	3	4	3
2	4	4	4	4	4	4	5	5	5	4
3	4	3	5	5	4	2	4	4	3	3
4	3	3	5	5	5	4	3	3	5	3
5	3	4	3	5	4	4	3	3	4	3
6	3	4	3	3	4	3	3	2	3	4
7	4	4	3	3	4	3	3	3	3	3
8	3	5	5	5	3	3	4	4	5	4
9	5	4	3	4	3	3	4	2	5	5
10	5	5	5	5	3	3	4	3	4	4
11	3	2	3	5	4	2	3	1	2	2
12	5	4	3	3	4	3	5	5	5	3
13	5	5	4	5	5	3	4	3	4	4
14	4	4	4	4	5	3	5	5	4	4
15	5	4	4	5	4	4	5	3	5	5
16	4	4	3	5	4	4	3	3	3	3
17	5	3	5	5	4	4	4	4	5	5
18	5	5	3	3	4	4	5	5	4	4
19	5	3	5	4	5	4	5	3	4	4
20	4	3	4	5	4	3	4	4	5	4

Apkopojot aptaujas rezultātus, redzams, ka eksperimentā piedalījās 20 eksperti, rezultāti ir sadalīti 5 kategorijās, katrai no tām ir pievienots dokumentācijas numurs. Dokumentācija zem Nr. 1 ir manuālā dokumentācija, bet dokumentācija zem Nr. 2 ir automatizēta dokumentācija. Vērtības, kas tika iegūti aptaujas laika, svārstās no 1 līdz 5 punktiem. Aplūkojot 3.7. tabulu, ir grūti izdarīt secinājumus no aptaujas rezultātiem. Tāpēc 3.8. tabula tika izveidotā, norādot kopējo iegūto punktu skaitu un vidējās vērtības katrai dokumentācijai un kategorijai.

3.8. tabula

Anketēšanas rezultātu apkopojums

	Lietojamība		Precizitāte		Detalizācija		Meklēšana		Saprotamība	
Dokumentācija	**Nr.1**	**Nr.2**	**Nr.1**	**Nr.2**	**Nr.1**	**Nr.2**	**Nr.1**	**Nr.2**	**Nr.1**	**Nr.2**
Punkti	83	76	78	88	82	66	79	68	82	74
Vidēja vērtība	4,15	3,8	3,9	4,4	4,1	3,3	3,95	3,4	4,1	3,7

Paļaujoties uz izveidotu tabulu 3.8. var izveidot dažus secinājumus par dokumentācijas kvalitāti. Pirmkārt, balstoties uz kategoriju: "Lietojamība", var secināt kā eksperti uzskata ka dokumentācija kas ir izveidota manuāla veida ir nedaudz ērtākā lietošanai nekā dokumentācijai izstrādātai automātiska veida, bet tas starpība nav liela. Tas secina kopēji sasummēti punkti, kas manuālam veidam ir 83 punkti, savukārt automātiskam veidam ir 76 punkti. Kopumā par dokumentācijas lietošanas ērtumu var secināt, kā gan manuālam veidam, gan automātiskam veidam ir vienāds lietošanas ērtums.

Runājot par dokumentācijas precizitāti, balstoties uz iegūtiem rezultātiem, var pateikt kā dokumentācijas kas bija izveidota automātiskā veida tiek uzskatīta par precīzu jo tajā eksperti atrodas vismazāk kļūdas saistītas ar kodu, komentārus un tas pareizrakstību. Tas rezultāti ir 78 pret 88 punkti. Tātad, ja izstrādes procesam ir svarīga precīzā dokumentācijā, tam jāpielieto automātisko dokumentācijas risinājumu.

Salīdzinot abus dokumentācijas veidošanas veidus, detalizācijas līmenis ir augstāks manuālajai dokumentācijai, kas aptaujas laikā ieguva 82 punktus. Šī atšķirība ir pietiekama, lai secinātu, ka detalizācijas līmenis ir labāks manuālajai dokumentācijai, jo tās pamatā ir specializēta valoda un zināšanas par projektu.

Pēdējos kritērijos tas vērtības ir nedaudz atšķirīgi, bet rezultāta, var secināt ka manuālai dokumentācijai ir gan vieglāk atrast nepieciešamu informāciju, gan ir vieglāk saprast visu novietotu informāciju par funkcijām, metodēm un citiem koda dalām. Automātiskā dokumentācijai nav slikts tas meklēšanas ērtums un dokumentācijas saprotamībā, bet tas nedaudz piekāptas manuālam veidam.

Saskaitot visas iegūtos punktus, manuālai un automātiskai dokumentācijai, un sadalot ar tas kategorijas skaitu, ir iegūts kopīgs vērtējums 10 balles sistēmā, kas palīdzes noskaidrot kura no dokumentācijas veidam ir labāka kvalitāte. Aprēķinot šīs vērtības, manuālā dokumentācija saņēma 8 punktus, bet automātiskā dokumentācija -

7 punktus. Rezultātā varam teikt, ka manuālajai dokumentācijai ir kvalitatīvāks saturs, taču atšķirība salīdzinājumā ar automātisko dokumentāciju nav liela.

3.5. Secinājumi

3. nodaļā ir iekļauts praktisks tests, kurā izmantots mans izstrādātais risinājums. Satura nodaļās ir definēti galvenie kritēriji, pēc kuriem tiek vērtēti un salīdzināti manuālie un automātiskie dokumentācijas veidošanas veidi. Katram eksperimentam tika aprakstīta īstenošana un apkopoti rezultāti. Turklāt tika izveidota manuālā tipa dokumentācija un automātiskā tipa dokumentācija. Kopumā tika izveidoti 10 dokumentācijas faili. Katram eksperimentam tika veikta detalizēta izpēte un izdarīti secinājumi par katru no definētajiem kritērijiem.

Pamatojoties uz 3. nodaļā 1. eksperimentā iegūtajiem rezultātiem, var secināt, ka, izmantojot manuālo dokumentācijas izveides un atjaunināšanas metodi, tas aizņem pietiekamu laiku no izstrādes procesam. Izmantojot šādu veidu tas ari var palielināt kļūdu veikšanas varbūtību, kas savukārt pazemināsdokumentācijas kvalitātē. Tas ari var izraisīt atjaunināšanas grūtības izstrādātajiem, kā ari var patērēt daudz cilvēkresursus. Savukārt, izmantojot automātisko veidu, tas var paātrināt dokumentācijas izveidošanas un atjaunināšanas procesu, atvieglot specialista darbu, kā ari var samazināt cilvēkā kļūdu iespējamību. Pēc eksperimenta analīzes, var secināt, ka automātiska dokumentācijas izstrādes veids ir ieteicams izstrādes procesam, jo tas dot vairāk priekšrocības tam.

Pamatojoties uz 3. nodaļā iegūtajiem 2. eksperimenta rezultātiem, var secināt, ka kopējā dokumentācijas kvalitāte ir labākā manuālajā veida. Tā kā, pamatojoties uz 3.8. tabulu, kurā ir ierosināti vērtējumi katram kritērijam, un kopējiem dokumentācijas veidu vērtējumiem, var secināt, ka manuālā režīma dokumentācija ir labāka lietojamības, detalizācijas, meklēšanas un saprotamības ziņā. Savukārt automātiskai dokumentācijai ir labāka dokumentācijas precizitāte. Taču, pamatojoties uz nelielā aprēķinā iegūtajiem punktiem, dokumentācijas kvalitātes atšķirība starp automātisko (7 punkti) un manuālo (8 punkti) dokumentāciju nav būtiska, tāpēc dokumentācijas veida izvēle ir atkarīga no uzņēmuma un izstrādes procesa.

Pamatojoties uz 1. un 2. eksperimenta rezultātiem un secinājumiem, var droši apgalvot, ka automātiskās dokumentācijas izveides risinājums ir ļoti laba alternatīva manuālajam. Tas ir vairākas reizes ātrāks nekā manuālais, un dokumentācijas kvalitāte

ir novērtētā par labu. Tādējādi var secināt, ka automātiskā dokumentācijas izveides risinājuma izmantošana ir ieteicama un tā ietekme uz izstrādes procesu ir pozitīva.

3.6. Risinājuma salīdzinājums ar esošiem risinājumam

Lai pamatotu sevis piedāvātu risinājuma vērtību un praktisko nozīme tam, tiek veikts mana risinājuma salīdzinājums ar eksistējošiem risinājumiem. Mana risinājuma salīdzinājums ir veikts ar rīkiem kas bija jau minēti iepriekš mana darbā. Tas ir DoxyGen un JavaDoc rīki.

Apskatot un izanalizējot abus jau eksistējošos risinājumus, 1. sadaļā, ir izvirzīti 6 galvenie salīdzināšanas kritēriju pēc kuriem ir veikta salīdzināšanā. Salīdzinājumā laika, liela uzmanība ir pievērstā programmēšanas valodām (kuras izmantotas risinājuma), dokumentācijas ģenerēšanai (dokumentācijas formāti), integrācijas iespējam (vides ar kuram integrējas), funkcionalitāte (papildu funkcijas), pielāgošanai un dokumentācijas kvalitātei. Pēc visu risinājumu izpētei tiek veikts apkopojums un tas rezultāti ir novietoti 3.9. tabulā.

3.9. tabula

Risinājuma salīdzinājums

Numurs	**Kritērijs**	**JavaDoc**	**DoxyGen**	**Mans risinājums**
1.	Programmēšanas valodas atbilstībā	Java	C, C++, Java, Python, PHP	Python
2.	Dokumentācijas ģenerēšana	HTML	HTML, LaTeX, RTF, PDF	HTML
3.	Integrācijas iespējas	Ir	Ir	Nav
4.	Funkcionalitāte	Vienkāršas iespējas	Plašas iespējas	Minimāla
5.	Pielāgošanā	Vienkāršā	Plaša	Vienkāršā
6.	Dokumentācijas kvalitāte	Vienkāršā un efektīva	Augsta kvalitāte	Vienkārša un efektīva

Balstoties uz 3.9. tabulu var izveidot salīdzinājumu starp 3 risinājumiem. Pirmā kritērijā ietvaros, bija salīdzinātas valodas, ar kuriem rīks var izveidot dokumentāciju. JavaDoc un mana risinājuma gadījuma ir iespēja izveidot dokumentāciju balstoties tikai uz vienu programmēšanas valodu, Java un Python valodā. Savukārt DoxyGen rīkam ir pieejamas vairāki programmēšanas valodas, kas padaru šo risinājumu par efektīvāko un vairāk izmantojamu.

Otra kritērijā ir vērtēts tas dokumentācijas ģenerēšanas formāts. Mana un JavaDoc risinājuma dokumentāciju ir iespējams izveidot tikai HTML valodā. DoxyGen risinājuma ir vairāki formāti, ieskaitot HTML valodu, līdz ar to tas iedot risinājuma lietotajiem plašāku izvēli un iespējas.

Trešais kritērijs novērtē iespēju integrēties ar citām vidēm. Man izstrādātā risinājuma šādas iespējas nav, taču JavaDoc un DoxyGen ir iespēja integrēties ar dažādām vidēm, piemēram, IDEA un Eclipse. Šī spēja palielina rīka lietderību un efektivitāti reālos izstrādes procesos.

Ceturtā kritērijā ietvaros ir salīdzinātas tas funkcionalitāte. Manam risinājumam ir minimāla funkcionalitāte salīdzinot ar JavaDoc un DoxyGen, kuriem ir iespēja veikt diagrammas, meklēšanas funkcionalitāte, kā ari pieejamas atsauces uz kodu. Mans risinājums var tikai veikt dokumentāciju balstoties uz kodu un tas komentāriem.

Piektā kritērijā ir vērtēts tas pielāgošanas iespējas. Vienkāršās pielāgošanas iespējas piemīt JavaDoc un manam risinājumam. Savukārt DoxyGen rīkam ir plašā pielāgošanā un konfigurācijas iespējas. Tas nozīme kā izmantojot DoxyGen rīku ir iespēja konfigurēt izveidošanas parametrus, stilu un anotācijas, mainīt dokumentācijas izkārtojumu, mainīt stilu dokumentācijai, satura filtrēšana un dažādu valodu atbalsts.

Pēdējā kritērijā ietvaros tiek vērtētas tas dokumentācijas kvalitāte. Kvalitāte esošiem risinājumiem ir balstīta uz atsauksmēm no oficiālā timekļa vietnēm, savukārt mana risinājumā dokumentācijas kvalitāte ir novērtētā otra eksperimentā laika, 3.4.5. sadaļā. DoxyGen dokumentācija ir vērtā par augstu, bet JavaDoc un mans risinājums dokumentācijas kvalitāte ir vērtētā par vienkāršo un efektīvu.

Apkopojot visus iegūtas salīdzināšanas rezultātus un balstoties uz iepriekš izveidotu aprakstu var secināt, ka mans izstrādāts risinājums pēc tās salīdzināšanas kritērijiem ir līdzīgs JavaDoc rīkam. Jo tas programmēšanas valodas, dokumentācijas formāti, kvalitāte un pielāgošanā ir līdzīgās. Savukārt integrācijas iespējas un tas funkcionalitātē ir nedaudz sliktākas nekā JavaDoc rīkam. Bet kopuma manu risinājumu var novērtēt par vērtīgu, tas izmantošana ir iespējama izstrādes procesos noteiktajos

projektos. Kā ari var secināt kā rezultāti kas bija iegūtas eksperimentos laikā izmantojot manu risinājumu var novērtēt par uzticamiem un kvalitatīviem. Tāpēc kā šāds risinājums ir līdzīgs JavaDoc rīkam, kurš uz dotu momentu ir izmantots dažādas uzņēmumos un izstrādes procesus par pamata rīkam dokumentācijas veidošanai.

REZULTĀTI UN SECINĀJUMI

Izstrādājot bakalaura darbu par tēmu "Automatizētas dokumentācijas ģenerēšanas ietekmes izpēte uz izstrādes procesu " tika paveikti sekojoši uzdevumi:

1. Tiek aplūkots programmatūras izstrādes process, tā dzīves cikls un dokumentācijas loma izstrādes procesā.
2. Tika apspriestas dokumentācijas priekšrocības un iespējamie trūkumi.
3. Tika pētīta dokumentācijas funkcionalitāte un pielāgojamība.
4. Tika apspriesta un pētīta programmētāju un atslēgvārdu loma dokumentēšanas procesā.
5. Tika izpētīti esošie risinājumi un tehnoloģijas dokumentācijas izveidei.
6. Risinājuma daļa, tika izstrādātā un aprakstīta dokumentācijas algoritma struktūra, pēc kuras tiek izstrādāts risinājums.
7. Tika izstrādāti komentāru veidi priekš dokumentācijas veidošanas risinājumam, lai izstrādātā risinājuma būtu iespēja identificēt komentārus.
8. Balstoties uz informāciju kas bija iegūta veicot izpēte par pieejamiem dokumentācijas ģenerēšanas rīkiem tiek izstrādāts savs risinājums Python valodā, kas palīdzētu veikt eksperimentus saistība ar dokumentāciju ātrumu un tās kvalitāte.
9. Risinājuma pārbaudes daļa tiek definēti divi pamata kritēriji kuri tiek pārbaudīti eksperimentā laikā.
10. Tika veikta praktiska validācija, pielietojot izstrādātu dokumentācijas ģenerēšanas risinājumu reālajā izstrādes procesā.
11. Tika veikti divi eksperimenti, kas pēc padziļinātas datu izpētei palīdzēja noskaidrot kurai dokumentācijas veidam ir ātrākā izstrāde, dokumentācijas kvalitāte un kuru no veidiem ir ieteicams izmantot izstrādes procesā.
12. Tika izpildīti visi nodefinēti darba uzdevumi un sasniegts darbā izvirzītais mērķis.

Izstrādājot bakalaura darbu tika sasniegti sekojošie rezultāti:

1. Tika pierādīts, ka pateicoties uz 1. eksperimenta rezultātiem un manuālas dokumentācijas izpētei, tas izveidošanas process aizņem pietiekamu laiku no izstrādes procesam un tas slikti ietekme uz izstrādes procesu.
2. Manuālas dokumentācijas izveidošanas laiks ir tieši atkarīgs no projekta saturā un apjoma. Projekts kas ir izmantots validācijā laika sastāva no 5 koda failiem un tas izveidošana ekspertiem aizņēma 162 minūtes. Šādu iegūtu rezultātu var raksturot par ļoti ilgu, kas stingri ietekmes izstrādes procesam un cilvēkam apkārt tam.
3. Paļaujoties uz manuālas dokumentācija atjaunināšanas procesa rezultātiem, atjaunināšana tērē nopietnu laiku no izstrādes procesam, bet salīdzinot to ar izveidošanas laiku rezultāts nav tik slikts.
4. Manuālas dokumentācijas atjaunināšanas laiks ir tieši atkarīgs no atjaunināšanas apjoma. Projekts kas ir izmantots validācijā laika sastāva no 5 koda failiem un tas atjaunināšana eksperimenta laika aizņēma 21,7 minūtēs.
5. Paļaujoties uz automātiskā veida iegūtiem rezultātiem, 1. eksperimenta laika, automātiskā dokumentācija izveidošana un atjaunināšana aizņem gan no eksperta, gan no iestrādes procesa tikai 36,1 sekundes. Šāds rezultāts ir ļoti labs salīdzinot ar manuālas dokumentācijas rezultātu. Dokumentācijas izveidošanas laiks pozitīvi ietekmes izstrādes procesam, paātrinot to, atvieglot specialistu darbību un samazinot cilvēkresursu.
6. Paļaujoties uz abu dokumentācijas veidošanas veidu salīdzinājumu, automātiskais veids ir uzkātots par ieteikumu izstrādes procesam, jo tas ātrums gan izveidošanas procesa, gan atjaunināšanas procesam ir līdzīgs un ātrāks nekā tas izveidotā manuāla veida.
7. Balstoties uz rezultātiem kas bija iegūti 2. eksperimenta laika aptaujas veida var secināt kā dokumentācija kas ir izstrādātā manuāla veida, dokumentācijas kvalitāte ir labāka salīdzinot ar automātisko veidu.
8. Balstoties uz ekspertu anketas atbildēm ir secināts ka manuālā dokumentācija ir labākā detalizācija, ir labākā tas dokumentācijas lietojamības ērtums, informācijas meklēšana un satura saprotamība.

9. Savukārt paļaujoties uz eksperta atbildēm, automātiskā dokumentācija atbilst ļoti precīzam saturam. Dokumentācijas izstrādātā automātiska veida veic vismazāk kļūdas saistība ar tas saturu.
10. Paļaujoties uz kopēju iegūtu vērtējumu par abiem veidiem, dokumentācija kas ir izveidota automātiskā veida ieguva 7 balles no 10, savukārt manuāla dokumentācijas kvalitāte ieguva 8. Balstoties uz šādiem rezultātiem var secināt kā automātiskais veids veido kvalitatīvo dokumentāciju, bet tas ir nedaudz sliktākā par manuālo.

Balstoties uz 1. un 2. eksperimenta rezultātiem var izveidot kopīgo secinājumu. Automātiskas dokumentācijas ģenerēšanas risinājuma pielietošanā izstrādes procesam ir ieteicamā izstrādes procesam un tas uzņēmumiem. Jo šādā veida dokumentācijas izveidošana un atjaunināšana ir ļoti ātra un tas dokumentācijas kvalitāte ir vērtētā ekspertiem par labu. Turklāt šādā risinājumā pielietošanā var paātrināt izstrādes procesu, padarot izstrādes procesu par efektīvu un kvalitatīvu, atvieglot programmētāju darbu, pazeminot izmaksas un cilvēkā resursu pielietošanā.

Turpmāk šo pētījumu var izmantot, kā uzticamu resursu priekš iepazīšanai ar automātiskā dokumentācijas ģenerēšanas pielietošanai izstrādes procesam. Pētījums ari palīdzes noskaidrot uzņēmumiem vai izstrādātajiem, cik svarīgi ir šādā risinājuma pielietošanas izstādes procesam. Kā ari iedos priekšstatu par priekšrocībām un trūkumiem saistība ar manuālo un automātisko dokumentācijas pielietošanu.

Pēc šī darba autores domam, izstrādāt automātisko dokumentācijas ģenerēšanas rīku kas atbilst ātrai dokumentācijas izveidošanai, atjaunināšanai un atbilst ļoti kvalitatīvai dokumentācijai ir iespējams. Kā ari šādā risinājuma pielietošanā ietekme pozitīvi izstrādes procesam un tā izmantošana ir ieteicama izstrādes procesam.

IZMANTOTIE INFORMĀCIJAS AVOTI

[1] C. Yin and W. Zhang, "New Product Development Process Models," 2021 International Conference on E-Commerce and E-Management (ICECEM), Dalian, China, 2021, pp. 240-243, **Pieejams šeit**: https://ieeexplore-ieee-org.resursi.rtu.lv/document/9636948

[2] V. Liubchenko, "Specific Aspects of Software Development Process for AI/ML-based Systems," 2022 IEEE 17th International Conference on Computer Sciences and Information Technologies (CSIT), Lviv, Ukraine, 2022, **Pieejams šeit**: https://ieeexplore-ieee-org.resursi.rtu.lv/document/10000821

[3] Д.Б. Берг, О.М. Зверева, А.Ю. Вишнякова, “Управление жизненным циклом информационных систем”, М-во науки и высшего образования РФ, 2022. – 94, ISBN 978-5-7996-3594-7, **Pieejams šeit**: https://elar.urfu.ru/bitstream/10995/119410/1/978-5-7996-3594-7_2022.pdf

[4] Невлюдов И. Ш., Андрусевич А. А., Евсеев В. В. Анализ жизненного цикла разработки программного обеспечения для корпоративных информационных систем // ВЕЖПТ. 2010. №8 (48). **Pieejams šeit:** https://cyberleninka.ru/article/n/analiz-zhiznennogo-tsikla-razrabotki-programmnogo-obespecheniya-dlya-korporativnyh-informatsionnyh-sistem

[5] J. Klespitz, M. Bíró, L. Kovács, "Aspects of improvement of software development lifecycle management," 2015 16th IEEE International Symposium on Computational Intelligence and Informatics, Budapest, Hungary, 2015, **Pieejams šeit**: https://ieeexplore-ieee-org.resursi.rtu.lv/document/7382943

[6] M.Pushpa Arthuer, “Automatic Source Code Documentation using Code Summarization Technique of NLP” Procedia Computer Science, vol.171, pp. 2522-2531, 2020. **Pieejams šeit:** https://www.sciencedirect.com/science/article/pii/S1877050920312655

[7] P.W.McBurney, "Automatic Documentation Generation via Source Code Summarization," 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Florence, Italy, 2015, pp. 903-906. **Pieejams šeit:** https://ieeexplore-ieee-org.resursi.rtu.lv/document/7203110

[8] P. Rodeghero, "Behavior-Informed Algorithms for Automatic Documentation Generation," 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), Shanghai, China, 2017, pp. 660-664.
Pieejams šeit: https://ieeexplore-ieee-org.resursi.rtu.lv/document/8094477

[9] J.Y.Khan and G.Uddin, "Automatic Code Documentation Generation Using GPT-3" Proceedings of the 37th IEEE/ACM International Conference on automated software engeneering, ase 2023
Pieejams šeit:
https://www.webofscience.com/wos/woscc/full-record/WOS:001062775200129

[10] E. Aghajani et al., "Software Documentation Issues Unveiled," 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 2019, pp. 1199-1210.
Pieejams šeit: https://ieeexplore-ieee-org.resursi.rtu.lv/document/8811931

[11] L. Stein, "Trends in automating document generation," in IEEE Software, vol. 12, no. 5, pp. 116-118, Sept. 1995.
Pieejams šeit: https://ieeexplore-ieee-org.resursi.rtu.lv/document/406781

[12] J.Bosch, "Continuous Software Engineering: An Introduction" in Continuous Software Engineering. Springer, 978-3-319-11283-1, 09 October 2014, **Pieejams šeit:** https://doi.org/10.1007/978-3-319-11283-1_1

[13] T.Theunissen, U.V.Heesch, P.Avgeriou, "A mapping study on documentation in Continuous Software Development" Information and Software Technology, vol.142, 106733, ISSN 0950-5849, 2022. **Pieejams šeit:** https://www.sciencedirect.com/science/article/pii/S095058492100183X

[14] C.González-Mora, C.Barros, I.Garrigós, J.Zubcoff, E.Lloret, J.Mazón, "Improving open data web API documentation through interactivity and natural language generation" Computer Standards & Interfaces, vol.83, 103657, ISSN 0920-5489, 2023. **Pieejams šeit:**
https://www.sciencedirect.com/science/article/pii/S0920548922000344

[15] D. Spinellis, "Code Documentation," in IEEE Software, vol. 27, no.4, pp.18-19, July-Aug. 2010.
Pieejams šeit: https://ieeexplore-ieee-org.resursi.rtu.lv/document/5484109

[16] M.Hermenegildo, “A System for Automatically Generating Documentation for (C)LP Programs” Electronic Notes in Theoretical Computer Science, vol.30, no.4, pp. 289-307, May 2005
Pieejams šeit:
https://www-sciencedirect-com.resursi.rtu.lv/science/article/pii/S1571066105806632

[17] A.Andersen, “Automatic Generation of Technical Documentation” Expert Systems with Applications, vol.3, no.2, pp. 219-227, 1991.
Pieejams šeit:
https://www.sciencedirect.com/science/article/abs/pii/095741749190150D

[18] P.W.McBurney and C.McMilan, “Automatic Documentation Generation via Source Code Summarization of Method Context” June 2014.
Pieejams šeit: https://www3.nd.edu/~cmc/papers/mcburney_icpc_2014.pdf

- Bakalaura darba gaita tiek izmantot dažādi tulkošanas rīki, kas palīdzes uzlabot teksta kvalitāte. Google Translate, pieejams šeit: https://translate.google.com/ un DeepL Translator, kas ir pieejams šeit: https://www.deepl.com/translator

PIELIKUMI

Pielikumi

1. pielikums. “Main.py” kods izstrādātai automātiskas dokumentācijas ģenerēšanas risinājumam.
2. pielikums. “DocGeneration.py” kods izstrādātai automātiskas dokumentācijas ģenerēšanas risinājumam.
3. pielikums. Dokumentācija kas ir izveidota manuāla veida izmantojot “addressvalidation” failu.
4. pielikums. Dokumentācija kas ir izveidota pielietojot izstrādātu risinājumu un izmantojot “addressvalidation” failu.
5. pielikums. Aptauja par dokumentācijas kvalitāte veikta otra eksperimenta laika.
6. pielikums. Aptaujas statistika par dokumentācijas kvalitāte.

1. pielikums

"Main.py" kods izstrādātai automātiskas dokumentācijas ģenerēšanas risinājumam.

```
import DocGeneration
import os
import re
import tkinter as tk
from tkinter import filedialog
import sys
import time
# Function for folder choosing using tkinter
def open_folder():
    root = tk.Tk()
    root.withdraw()
    data = filedialog.askdirectory(parent=root)
    return data
# Method for choosing folders for files and documentation
def folder(first,second):
    # Choose a folder with files with CODES
    first = open_folder()
    print("Folder is chosen successfully!")
    second = open_folder()    # Choose a folder for DOCUMENTATION saving
    # Check if folder with files is empty
    if not os.listdir(first):
        sys.exit(f"The folder: {first} - is empty.")
    # Check if folder path for files is the same as folder path for documentation
    if first == second:
        print("ERROR:")
        print("Chosen folders path are the same! Please choose another folder!")
        # Do while they are not the same folder path
        while first == second:
            second = open_folder()
    print("Folder is chosen successfully!")
    return first, second
def extension_check(folder_file,folder_doc,documentation_name):
```

```
            try:
                file_list = os.listdir(folder_file)             # Add a list of all files from folder_path
                for i in file_list:                             # Go throught each file from file list
                    # replace \\ symbol to / so path would be used correctly in program
                    file = os.path.join(folder_file, i).replace('\\','/')
                    # Check if file has something in it
                    if os.path.getsize(file) == 0:
                        continue
                    extension = os.path.splitext(file)[1]       # Get file extension
                    if extension == ".py":         # Check if extension is what program need
                        file_read(file,i,folder_doc,documentation_name)     #     Open     Method     for
documentation making
                    else:
                        print("The file is not a .py file")
            except FileNotFoundError:
                print("The specified file was not found.")
            except Exception as e:
                print("An error occurred:", e)
        def file_read(file,i,folder_doc,documentation_name):
            # Values which will be used during program work
            count = 0
            j = 0
            imp = 0
            path = file
            ch = 0
            if os.path.isfile(file):  # Check if current file is still enable
                with (open(file, 'r', encoding='utf-8') as file):  # If everything is okay, then open and
read file
                    row = file.readlines()  # read file by lines
                    # Values which are going to be used for code file description
                    html = ''
                    impr = ''
                    metodes_list = ''
                    function_list = ''
                    class_list = ''
                    # For process which read file line by line
                    for line in row:
```

```
                if """ in line:        # Check if in line is available """ symbols to able to comment
comments in it
                    if imp > 0:        # Check if import search are in progress and stop it
                        imp = 0
                        impr += f"</code></pre>"
                    if j > 0:          # Check if code or text search are in progress and stop it
                        j = 0
                        html += f"</code></pre></div>"
                    count = count + 1
                    # Check what phase is comments reading. (just started - 1, finished - 2)
                    if count == 1:
                        html += f'<div>'
                    if count == 2:
                        count = 0
                        html += f"</div>"
                    continue
                # Check if in text is available import text
                elif 'import' in line:
                    imp = imp + 1
                    # Check what phase is import reading. (just started - 1, in progress > 0)
                    if imp == 1:
                        impr += f"<pre><code>"
                    if imp > 0:
                        impr += line + f"\n"
                    continue
                # Check if line is empty
                elif line.strip() == "":
                    if imp > 0:
                        imp = 0
                        impr += f"</code></pre>"
                    continue
                # Else just close import reading and writing
                else:
                    if imp > 0:
                        imp = 0
                        impr += f"</code></pre>"
```

```
                c = 0
                if 'def' in line:
                    text = re.findall(r'\((.*?)\)', line)
                    for match in text:
                        if 'self' in match:
                            c = 1
                    match = re.search(r'def\s*(.*)', line)
                    if c == 1:
                        if match:
                            metodes_list += f"<li>" + match.group(1) + "</li>\n"
                    else:
                        if match:
                            function_list += f"<li>" + match.group(1) + "</li>\n"
                elif 'class' in line:
                    match = re.search(r'class\s*(.*)', line)
                    if match:
                        class_list += f"<li>" + match.group(1) + "</li>\n"
                if count == 1:
                    if '$author' in line:
                        match = re.search(r'\$author\s*(.*)', line)
                        if match:
                            html += "<p><b>Author:</b> " + match.group(1) + f"</p>\n"
                    elif '$version' in line:
                        match = re.search(r'\$version\s*(.*)', line)
                        if match:
                            html += "<p><b>Version:</b> " + match.group(1) + f"</p>\n"
                    elif '$date' in line:
                        match = re.search(r'\$date\s*(.*)', line)
                        if match:
                            html += "<p><b>Date:</b> " + match.group(1) + f"</p>\n"
                    elif '$method' in line:
                        match = re.search(r'\$method\s+(\w+)\s*(.*)', line)
                        if match:
                            html += f"<h3>Method:</h3><p> " + match.group(1) + " - " +
match.group(2) + "</p>\n"
                            #metodes_list += f"<li>" + match.group(1) + "</li>\n"
```

```
                elif '$class' in line:
                    match = re.search(r'\$cl\s+(\w+)\s*(.*)', line)
                    if match:
                        html += "<h2>Class:</h2><p> " + match.group(1) + " - " +
match.group(2) + f"</p>\n"
                elif '$par' in line:
                    match = re.search(r'\$par\s+(\w+)\s*(.*)', line)
                    if match:
                        html += "<p><b>Parameters:</b> " + match.group(1) + " - " +
match.group(2) + f"</p>\n"
                elif '$value' in line:
                    match = re.search(r'\$value\s+(\w+)\s*(.*)', line)
                    if match:
                        html += "<p><b>Value:</b> " + match.group(1) + " - " + match.group(2)
+"</p>\n"
                elif '$return' in line:
                    match = re.search(r'\$return\s+(\w+)\s*(.*)', line)
                    if match:
                        html += "<p><b>Return value:</b> " + match.group(1) + " - " +
match.group(2) + f"</p>\n"
                elif '$exception' in line:
                    match = re.search(r'\$exception\s+(\w+)\s*(.*)', line)
                    if match:
                        html += "<p><b>Exception:</b> " + match.group(1) + " - " +
match.group(2) + f"</p>\n"
                elif '#' in line:
                    match = re.search(r'#\s*(.*)', line)
                    if match:
                        html += "<p>" + match.group(1) + f"</p>\n"
                else:
                    html += "<p>" + line.strip() + f"</p>\n"
            elif count == 0:
                if line.strip().startswith("#"):
                    if j > 0:
                        j = 0
                        html += f"</code></pre></div>"
                    ch += 1
```

```
                    if ch == 1:
                        html += f'<div>'
                    html += f"<pre><code>" + line.strip() + f"</code></pre>\n"
                    continue
                elif ch >= 1:
                    html += f"</div>"
                    ch = 0
                j += 1
                if j == 1:
                    html += f'<div class=\"example\"><p>Code:</p><pre><code>'
                html += line
            file_path = os.path.join(folder_doc, i + ".html")
DocGeneration.DocGeneration(html,metodes_list,function_list,class_list,file_path,impr,i,docume
ntation_name,path) return
    def main(): # Function main() call
        # Documentation name request
        start_time = time.time()
        documentation_name = input("Enter documentation name: ")
        folder_file, folder_doc = folder(None,None)          # Folder choose for files and
documentation
        print(f"Selected folder for codes is: {folder_file}")     # Print chosen folder path for codes
        print(f"Selected folder for Documentation saving is: {folder_doc}")      # Print chosen
folder path for documentation
        # Method which check file extension and starts all program
        extension_check(folder_file,folder_doc,documentation_name)
        end_time = time.time()
        execution_time = end_time - start_time
        print("Execution time:", execution_time, "seconds")
    if __name__ == "__main__":
        main()
```

2. pielikums

“DocGeneration.py” kods izstrādātai automātiskas dokumentācijas ģenerēšanas risinājumam.

```
import os
import Main
from datetime import datetime
def DocGeneration(data,metodes_list,
function_list,class_list,file_path,impr,file,documentation_name,path):
    current_date = datetime.now().date()
    html_content = f"""
    <!DOCTYPE html>
    <html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>Python Generated Documentation</title>
    <style>
        body {{
            font-family: Arial, sans-serif;
            line-height: 1.6;
        }}
        h1, h2, h3 {{
            color: #333;
        }}
        code {{
            background-color: #f4f4f4;
            padding: 2px 4px;
            border-radius: 4px;
        }}
        pre {{
            background-color: #f4f4f4;
            padding: 10px;
            border-radius: 4px;
            overflow-x: auto;
        }}
        .example {{
```

```
            background-color: #e9ecef;
            padding: 10px;
            border-left: 4px solid #007bff;
            margin-bottom: 10px;
        }}
    </style>
    </head>
    <body>
        <h1>{file}</h1>
            <h3>Documentation information:</h3>
            <p><strong>Project name:</strong> {documentation_name}</p>
            <p><strong>Documentation made:</strong> {current_date}</p>
            <p><strong>The documentation was generated from the following file:</strong>{path}</p>
        <h2>Import Statements:</h2>
            {impr}
        <h2>Methodes list:</h2>
            <pre><code>{metodes_list}</pre></code>
        <h2>Function list:</h3>
            <pre><code>{function_list}</pre></code>
        <h2>Class list:</h3>
            <pre><code>{class_list}</pre></code>
        <p></p>
        <h2>Detailed description:</h3>
            {data}
    </div>
    </body>
    </html>
    """
    with open(file_path, "w") as file:
        file.write(html_content)
    print("HTML file generated successfully!")
```

3. pielikums

Izveidota dokumentācija, manuālā veidā, failam “addressvalidation”

Address Validation File

This code check the validity of addresses

Author: MSC

Date: 2016-09-02

Version: package V1.2

Import Statements

```
import unittest
```

Validate Method

The `validate()` method validates an address and returns a possibly corrected address.

```
def validate(adr, servicelevel=1):
```

Paramaters:

- adr - should be a object conforming to the address protocol
- servicelevel - can be an integer with the following values:

Values for `servicelevel`:

- 1 - generic validation, no money/effort should be spend on correction and suggestions
- 2 - TBD

Values for `status`:

- 10invalid - address is for sure non deliverable in this form
- 20troubled - bounced before or is unlikely to be correct - possible alternatives are returned
- 30ok - likely to work
- 41ok - likely to work but was corrected
- 40verified - we are sure it works

Code:

```
adr['land'] = adr['land'].strip()
    adr['plz'] = adr['plz'].strip()
```

Return:

returns (status, message, [corrected addresses and variants])

Code:

```
if adr['land'] != 'DE' and not adr['plz']:
        return ('10invalid', 'Postleitzahl fehlt', [adr])
    if adr['land'] == 'DE' and len(adr.get('plz', '')) != 5:
        return ('10invalid', 'Postleitzahl fehlerhaft', [adr])
    return ('30ok', '', [adr])
```

AddressvalidationTests Class

The `AddressvalidationTests()` class Tests for the address validation facility.

```
class AddressvalidationTests(unittest.TestCase):
```

SetUp method

The `setUp()` method Set up test address base.

```
class AddressvalidationTests(unittest.TestCase):
```

Code:

```
def setUp(self):
        self.address = {'name1': 'HUDORA GmbH',
                        'name2': 'Abt. Cybernetics',
                        'strasse': '13gerwald 13',
                        'land': 'DE',
                        'plz': '42897',
                        'ort': 'Remscheid',
                        'tel': '+49 2191 60912 0',
                        'fax': '+49 2191 60912 50',
                        'mobil': '+49 175 0000xxx',
                        'email': 'nobody@hudora.de'}
```

Test_good_address method

The `test_good_address()` method Test if correct addresses are considered correct.

```
def test_good_address(self):
```

Code:

```
self.assertEqual(validate(self.address)[0], '30ok')
        self.assertEqual(validate(self.address)[1], '')
```

test_missing_zip method

The `test_missing_zip()` method Test if correct addresses are considered correct.

```
def test_missing_zip(self):
```

Code:

```
self.address['plz'] = ''
        self.assertEqual(validate(self.address)[0], '10invalid')
```

test_short_zip method

The `test_short_zip()` method Test if correct addresses are considered correct.

```
def test_short_zip(self):
```

Code:

```
self.address['plz'] = '123'
        self.assertEqual(validate(self.address)[0], '10invalid')
```

test_long_zip method

The `test_long_zip()` method Test if correct addresses are considered correct.

```
def test_long_zip(self):
```

Code:

```
self.address['plz'] = '12345 Rade'
        self.assertEqual(validate(self.address)[0], '10invalid')
```

4. pielikums

Izveidota dokumentācija, pielietojot izstrādātu risinājumu, failam “addressvalidation”

addressvalidation.py

Documentation information:

Project name: Documentation Shipping V1.0 Esk.Nr.3

Documentation made: 2024-05-23

The documentation was generated from the following file:C:/Users/Gleb/Desktop/Shipping/Project/addressvalidation.py

Import Statements:

```
import unittest
```

Methodes list:

```
• setUp(self):
• test_good_address(self):
• test_missing_zip(self):
• test_short_zip(self):
• test_long_zip(self):
```

Function list:

```
• validate(adr, servicelevel=1):
• add(a, b):
```

Class list:

```
• AddressvalidationTests(unittest.TestCase):
```

Detailed description:

```
#!/usr/bin/env python
```

```
# encoding: utf-8
```

addressvalidation.py - check the validity of addresses

Author: MSC

Date: 2016-09-02

Version: package V1.2

Method:

validate - - Validates an address and returns a possibly corrected address.

Code:

```
def validate(adr, servicelevel=1):
```

Parameters: adr - should be a object conforming to the address protocol

Parameters: servicelevel - can be an integer with the following values:

Value: 1 - - generic validation, no money/effort should be spend on correction and suggestions

Value: 2 - - TBD.

status can be:

Code:

```
    adr['land'] = adr['land'].strip()
    adr['plz'] = adr['plz'].strip()
```

Return value: returns - (status, message, [corrected addresses and variants])

Code:

```
    if adr['land'] != 'DE' and not adr['plz']:
        return ('301=valid', 'Postleitzahl fehlt', [adr])
    if adr['land'] == 'DE' and len(adr.get('plz', '')) != 5:
        return ('301=valid', 'Postleitzahl fehlerhaft', [adr])
    return ('30ok', '', [adr])
```

$cl AddressvalidationTests Tests for the address validation facility.

Code:

```
class AddressvalidationTests(unittest.TestCase):
```

Method:

setUp - Set up test address base.

Code:

```
    def setUp(self):
        self.address = {'name1': 'HUDORA GmbH',
                        'name2': '301. Cyberverbis',
                        'strasse': 'Am Sonnenwald 13',
                        'land': 'DE',
                        'plz': '42897',
                        'ort': 'Remscheid',
                        'tel': '+49 2191 60912 0',
                        'fax': '+49 2191 60912 99',
                        'mail2': '+49 175 0000Nox',
                        'email': 'noknok@hudora.de'}
```

Method:

add - - This function takes two numbers as input and returns their sum.

Code:

```
    def add(a, b):
```

Parameters: a - (float or int): The first number.

Parameters: b - (float or int): The second number.

Return value: result - - float or int: The sum of the two input numbers.

Code:

```
        result = a + b
        return result
```

```
# Example usage:
```

Code:

```
        num1 = 5
        num2 = 3
        sum_result = add(num1, num2)
        print(f"The sum of {num1} and {num2} is {sum_result}")
```

Method:

test_good_address - - Test if correct addresses are considered correct.

Code:

```
    def test_good_address(self):
        self.assertEqual(validate(self.address)[0], '30ok')
        self.assertEqual(validate(self.address)[1], '')
```

Method:

test_missing_zip - - Test if correct addresses are considered correct.

Code:

```
    def test_missing_zip(self):
        self.address['plz'] = ''
        self.assertEqual(validate(self.address)[0], '301=valid')
```

Method:

test_short_zip - - Test if correct addresses are considered correct.

Code:

```
    def test_short_zip(self):
        self.address['plz'] = '123'
        self.assertEqual(validate(self.address)[0], '301=valid')
```

Method:

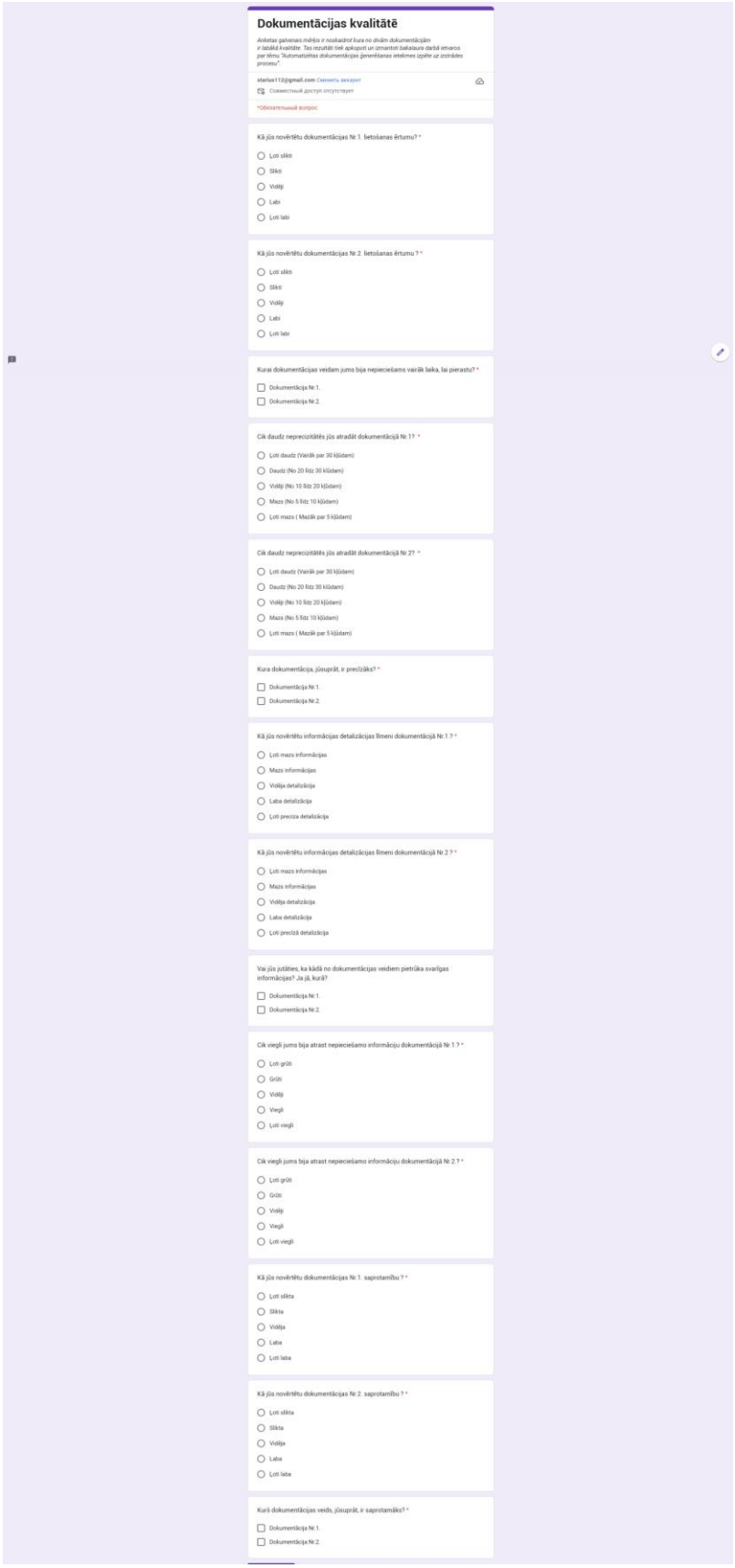
test_long_zip - - Test if correct addresses are considered correct.

Code:

```
    def test_long_zip(self):
        self.address['plz'] = '12345 Rade'
        self.assertEqual(validate(self.address)[0], '301=valid')
if __name__ == '__main__':
    unittest.main()
```

5. pielikums

Aptauja par dokumentācijas kvalitāte



Dokumentācijas kvalitātē

Anketas galvenais mērķis ir noskaidrot kura no divām dokumentācijām ir labākā kvalitāte. Tas rezultāti tiek apkopoti un izmantoti bakalaura darbā ietvaros par tēmu "Automatizētas dokumentācijas ģenerēšanas ieteikmes izpēte uz izstrādes procesu".

starlse112@gmail.com Сменить аккаунт

Совместный доступ отсутствует

*Обязательный вопрос

Kā jūs novērtētu dokumentācijas Nr.1. lietošanas ērtumu? *

- Ļoti slikti
- Slikti
- Vidēji
- Labi
- Ļoti labi

Kā jūs novērtētu dokumentācijas Nr.2. lietošanas ērtumu ? *

- Ļoti slikti
- Slikti
- Vidēji
- Labi
- Ļoti labi

Kurai dokumentācijas veidam jums bija nepieciešams vairāk laika, lai pierastu? *

- Dokumentācija Nr.1.
- Dokumentācija Nr.2.

Cik daudz neprecizitātēs jūs atradāt dokumentācijā Nr.1? *

- Ļoti daudz (Vairāk par 30 kļūdam)
- Daudz (No 20 līdz 30 kļūdam)
- Vidēji (No 10 līdz 20 kļūdam)
- Mazs (No 5 līdz 10 kļūdam)
- Ļoti mazs (Mazāk par 5 kļūdam)

Cik daudz neprecizitātēs jūs atradāt dokumentācijā Nr.2? *

- Ļoti daudz (Vairāk par 30 kļūdam)
- Daudz (No 20 līdz 30 kļūdam)
- Vidēji (No 10 līdz 20 kļūdam)
- Mazs (No 5 līdz 10 kļūdam)
- Ļoti mazs (Mazāk par 5 kļūdam)

Kura dokumentācija, jūsuprāt, ir precīzāka? *

- Dokumentācija Nr.1.
- Dokumentācija Nr.2.

Kā jūs novērtētu informācijas detalizācijas līmeni dokumentācijā Nr.1 ? *

- Ļoti maza informācijas
- Mazs informācijas
- Vidēja detalizācija
- Laba detalizācija
- Ļoti precīza detalizācija

Kā jūs novērtētu informācijas detalizācijas līmeni dokumentācijā Nr.2 ? *

- Ļoti maza informācijas
- Mazs informācijas
- Vidēja detalizācija
- Laba detalizācija
- Ļoti precīzā detalizācija

Vai jūs jutāties, ka kādā no dokumentācijas veidiem pietrūka svarīgas informācijas? Ja jā, kurā?

- Dokumentācija Nr.1.
- Dokumentācija Nr.2.

Cik viegli jums bija atrast nepieciešamo informāciju dokumentācijā Nr.1 ? *

- Ļoti grūti
- Grūti
- Vidēji
- Viegli
- Ļoti viegli

Cik viegli jums bija atrast nepieciešamo informāciju dokumentācijā Nr.2 ? *

- Ļoti grūti
- Grūti
- Vidēji
- Viegli
- Ļoti viegli

Kā jūs novērtētu dokumentācijas Nr.1. saprotamību ? *

- Ļoti slikta
- Slikta
- Vidēja
- Laba
- Ļoti laba

Kā jūs novērtētu dokumentācijas Nr.2. saprotamību ? *

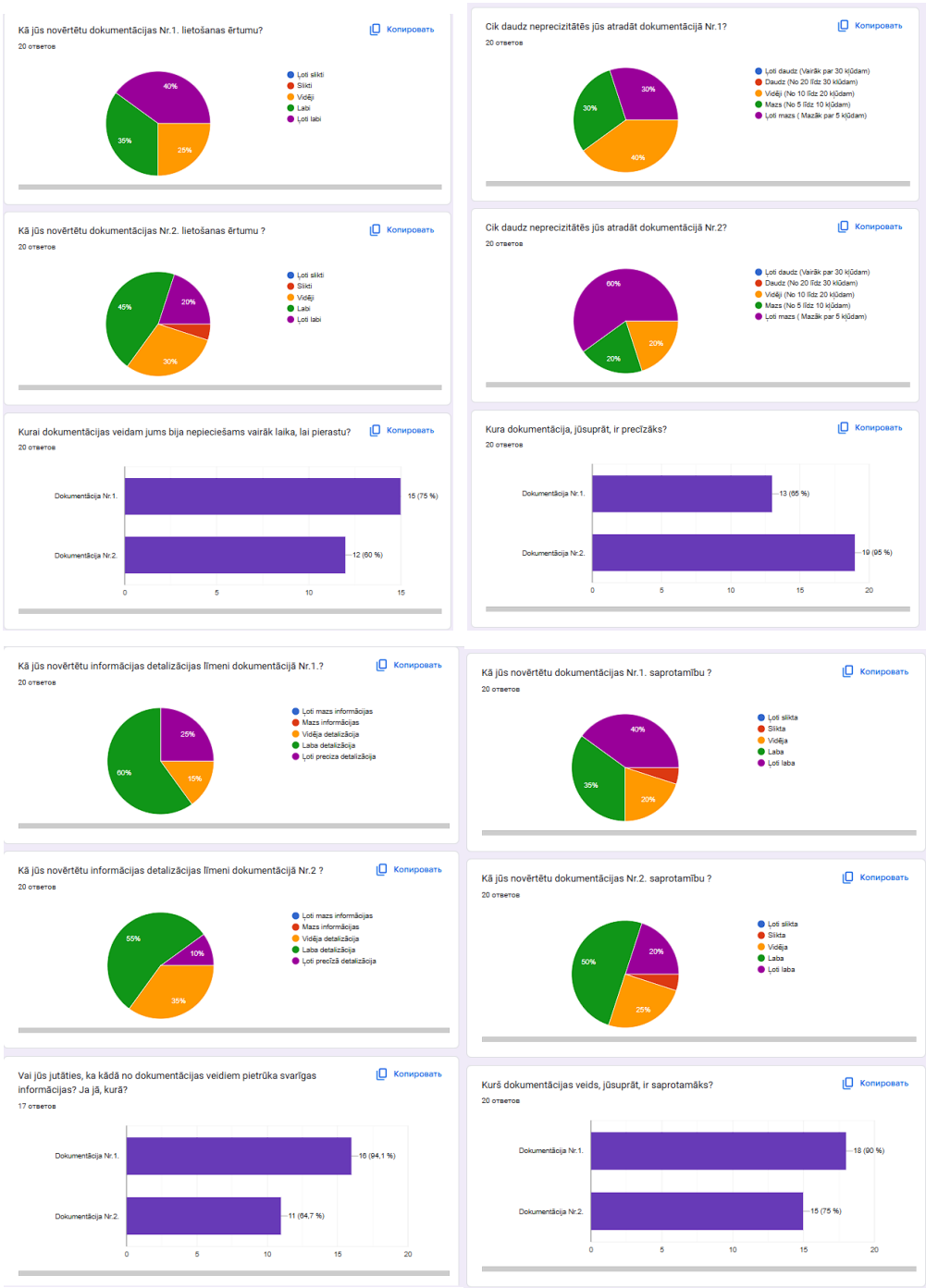
- Ļoti slikta
- Slikta
- Vidēja
- Laba
- Ļoti laba

Kurā dokumentācijas veids, jūsuprāt, ir saprotamāks? *

- Dokumentācija Nr.1.
- Dokumentācija Nr.2.

6. pielikums

Aptaujas statistika par dokumentācijas kvalitāte



Aptaujas statistika par dokumentācijas kvalitāte (turpinājums)