LATVIJAS UNIVERSITĀTE

DATORIKAS FAKULTĀTE

**WEBAPPOS .NET IZSAUKUMU ADAPTERIS UN C# STARPKODA ĢENERATORS**

KVALIFIKĀCIJAS DARBS

Autors:

Artis Pauniņš ap19122

Darba vadītājs: Sergejs Kozlovičs

RĪGA 2022

# ANOTĀCIJA

WebAppOS ir starpplatformu vide, kas ļauj izsaukt kodu, kas atrodas dažādos tīkla mezglos un kas rakstīts dažādās programmēšanas valodās.

Darbā tiek aprakstīti vairāki moduļi, kuru mērķis ir nodrošināt .NET koda izpildi WebAppOS vidē. Darbs satur programmatūras prasību specifikāciju, programmatūras projektējuma aprakstu, kā arī programmkoda un tās testēšanas dokumentāciju.

Izstrādātie moduļi ļaus savienot C# kodu ar kodu citās programmēšanās valodās, ka arī piekļūt tīmekļa atmiņai no C#.

Lai izstrādātu šos moduļus, tiek definēta jauna programmēšanas valoda un tiek izveidots tās kompilators un C# koda ģenerators. Lai nodrošinātu iespēju izsaukt .NET funkcijas no WebAppOS, tiek lietota .NET bibliotēka koda dinamiskai izsaukšanai.

# ABSTRACT

WebAppOS is a cross-platform enviornment, that alows to invoke code, that is stored in different web nodes and that is written in different programming languages.

The work describes multiple modules that aim to execute .NET code in the WebAppOS enviornment. This document contains a description of software specifications, software design as well as the documentation of source code and its testing.

The developed modules will allow us to connect C# code with code in other programming languages as well as to have web memory access from C#.

In order to develop these modules, a new programming language is defined and its compiler and C# code generator are created. In order to enable the posibility to invoke .NET functions from WebAppOS, a .NET library for dynamic code invoking is used.

# SATURA RĀDĪTĀJS

# 

# IEVADS

**Nolūks**

Kvalifikācijas darba nolūks ir aprakstīt programmatūras prasības un pirmkoda dokumentāciju izstrādājamajam adapterim un starpkoda ģeneratoram. Dokumentā tiks detalizēti aprakstītas sistēmas funkcijas un ierobežojumi kā arī ir dokumentēta programmatūras testēšana un konfigurāciju pārvaldība.

Dokuments paredzēts visām ieinteresētajām pusēm, tostarp izstrādātājiem.

**Darbības sfēra**

WebAppOS operētājsistēmas izsaukumu adapteri ir paredzēti, lai tie spētu izsaukt operētājsistēmas funkcijas. Adapterus veido atsevišķi katrai programmēšanas valodai, tādējādi šie adapteri dos iespēju operētājsistēmas izstrādātājiem izstrādāt jaunas un esošās funkcijas vairākās programmēšanas valodās.

Starpkoda ģenerators ir paredzēts, lai tiktu ģenerēts kods, kurš tiks izmantots, lai veidotu programmas WebAppOS videi.

**Saistība ar citiem dokumentiem**

Dokumenta noformējumā ievērotas standarta LVS 68:1996 “Programmatūras prasību specifikācijas ceļvedis” vadlīnijas.

**Pārskats**

Dokuments sastāv no 4 daļām:

1. pirmajā daļā ir ievadinformācija par programmatūru, kas satur dokumenta nolūku, mērķi, darbības sfēru, citu dokumentu izmantošanu dokumenta tapšanas laikā;
2. otrajā daļā tiek dots ieskats, par vispārējo produkta struktūru, tās lietotājiem, prasībām, ierobežojumiem un pieņēmumiem;
3. trešajā nodaļā tiek norādītas funkcionālās un nefunkcionālās prasības izstrādājamai programmai;
4. ceturtajā nodaļā ir programmatūras funkciju un datu struktūru projektējuma apraksts.

# APZĪMĒJUMU SARAKSTS

Atvērts kods jeb atvērta programmatūra – programmatūra, kas tiek izlaista saskaņā ar licenci, kurā autortiesību īpašnieks lietotājiem piešķir tiesības izmantot, pētīt, mainīt un izplatīt programmatūru un tās pirmkodu jebkuram nolūkam.

RAAPI (Repository Access API) – universāla zema līmeņa saskarne, kas nodrošina piekļuvi modeļu repozitorijam (repozitorijs ir datu bāze operatīvajā atmiņā, *in-memory database*).

WAOS C# - programmēšanas valoda, kurā var definēt klases, to atribūtus, metodes un asociācijas. Failiem ar kodu, kas ir rakstīta šajā valoda, ir paplašinājums *waoscs*.

Jaunkods – programmkods, kas rakstīts WAOS C# programmēšanas valodā.

Starpkods – programmkods, kurš tiek ģenerēts pēc jaunkoda veiksmīgas kompilēšanas.

Orģinālkods – programmkods, kurā ir rakstītas metodes, kuras izsauc WebAppOS.

Konteksts – dati, kas glabā informāciju par jaunkoda sastāvdaļu.

Identifikators – simbolu virkne, kas sākas ar burtu vai apakšsvītru un sastāv no burtiem, cipariem un apakšsvītrām.

Primitīvais datu tips – datu tips atribūtam, kurš glabājas RAAPI.

(Metodes) protokols – programmēšanas valoda, kurā tiek izstrādāta metode, kuru izmantos dinamiskai palaišanai WebAppOS vidē.

(Metodes) lokācija – simbolu virkne, kas pasaka koda glabāšanas veidu.

(Metodes) ceļš – ceļš, kas pasaka, kur kodā atrodas metode, piemēram, ja protokols ir *dotnet* un lokācija ir *local* tad ceļš būs formātā *dotnet:local:namespace.classname#methodname,* kur *namespace* ir koda vārdtelpa, *classname* ir klases, kurā metode atrodas, vārds un *methodname* ir metodes vārds. Ceļš ir specifisks konkrētam protokolam un lokācijai.

# 1. VISPĀRĪGS APRAKSTS

## 1.1 Esošā stāvokļa apraksts

Kā jebkurai programmai un operētājsistēmai, arī WebAppOS ir savas funkcijas. Tā kā WebAppOS ir izstrādāts valodā JAVA, tad arī funkcijas ir rakstītas valodā JAVA. Tas ierobežo iespējas izstrādāt operētājsistēmas funkcijas citās programmēšanas valodās.

WebAppOS ir arī funkcijas, kuras ir rakstītas valodā lua. Lai operētājsistēma varētu palaist šīs funkcijas, tad tīmekļa procesorā ir adapteris, kurš ir veidots kādai konkrētai programmēšanas valodai un šie adapteri spēj atrast un izsaukt funkcijas konkrētā programmēšanas valodā. Izņemot JAVA un lua valodas, nav pieejami adapterī citām programmēšanas valodām, piemēram, C# vai Python.

Uzdevums ir izstrādāt tīmekļa procesoram adapteri, kas ļautu operētājsistēmai izsaukt funkcijas dažādās programmēšanas valodās. Papildus tam tiks izstrādāts kompilators WAOS C# programmēšanas valodai un ģenerators, kurš ģenerēs C# valodā rakstītus starpkodus.

## 1.2 Pasūtītājs

Sistēmas pasūtītājs ir Latvijas Universitātes Matemātikas un informātikas institūts.

## 1.3 Produkta perspektīva

WebAppOS apraksts

* Galvenās funkcijas
  + WebAppOS nodrošina iespēju izsaukt kodu, kas ir rakstīts dažādās programmēšanas valodās un kas atrodas dažādos tīkla mezglos
  + WebAppOS nodrošina tīmekļa atmiņas sinhronizāciju starp klientu un serveri.
  + WebAppOS ļauj veidot tīmekļa lietotnes tādā pašā veidā, ka veido darbvirsmas lietotnes, nedomājot par tīklu.
* Saskarnes
  + RAAPI – ļauj piekļūt tīmekļa atmiņai. {https://webappos.org/dev/raapi/}
  + IWebCalls API – ļauj izsaukt funkcijas, kuras rakstītas citās programmēsanas valodās vai kas atrodas dažādos tīkla mezglos.
    - Local – virzienā no WebAppOS uz .NET.
    - Remote – virzienā no .NET uz WebAppOS.
* Nepieciešams tīkla savienojums ar WebAppOS serveri un vēlamā operētājsistēma ir Linux Ubuntu.

Adapteris un kodu ģenerators tiks iekļauti WebAppOS distribūcijā.

## 1.4 Sistēmas lietotāji

Sistēmu izmantos četras lietotāju grupas:

* Administratori – administratoru uzdevums būs WebAppOS uzstādīšana. Par instalācijas datiem tiek uzskatīti divi GitHub repozitoriji – WebAppOS un C# adapteris ar starpkoda ģeneratoru, kas izstrādāts kvalifikācijas darba ietvaros.
* Izstrādātāji – izstrādātāji veidos programmatūru WebAppOS videi izmantojot izstrādāto adapteri un kodu ģeneratoru.
* Gala lietotāji – gala lietotāji izmantos WebAppOS biznesa vajadzībām.
* WebAppOS – sistēma, kuru izmantos iepriekš minētās lietotāju grupas.

**

*1.1. att.* **0. līmeņa DPD diagramma**

## 1.5 Darījumprasības

* Dinamiski palaist funkcijas.
* LINQ atbalsts.
* Programmēšanas valoda jaunkoda veidošanai
* Jaunkodu kompilēšana, kas tiek veikta secīgi, un starpkoda ģenerēšana:
  + Vārdtelpu (namespace) atbalsts starpkodu ģenerēšanai:
    - Vārdtelpas nosaukums ir identifikators.
  + Rezervētie vārdi:
    - class – klašu definēšanai.
    - association – asociāciju definēšanai.
    - Integer – veselo skaitļu definēšanai.
    - String – simbolu virkņu definēšanai.
    - Boolean – Būla vērtību glabāšanai.
    - Real – reālo skaitļu glabāšanai.
    - Void – funkciju, kuras neatgriež vērtību, datu tips.
    - private – privāto lauku glabāšanai.
    - public – publisko lauku glabāšanai.
    - BaseObject – klase “BaseObject”, kas ir virsklase visām definētajām klasēm.
    - \_wm – WebMemory glabāšanai ģenerētajās klasēs.
    - \_wc – RemoteWebCalls glabasanai ģenerētajās klasēs
    - \_object – objekta glabāšanai ģenerētajās klasēs
  + Visiem elementiem glabā rindu, kur elements atrodas programmkodā. Kuru rindu glabā, ir atkarīgs no elementa:
    - Klasēm – kur atrodas klases vārds.
    - Asociācijām – kur atrodas asociācijas definīcija.
    - Laukiem – kur atrodas lauka vārds.
    - Argumentiem – kur atrodas argumenta vārds (ja vārds nav dots, tad tur, kur ir datu tips).
    - Anotācijas – kur sākas anotācija.
  + Kompilatora atmiņā neglabā laukus, kuriem nav doti vārdi, izņemot metožu argumentus. Asociāciju gadījumā neglabā asociācijas, kurām nav abu lomu vārdu
  + Klašu kļūdu pārbaude:
    - Vai eksistē virsklase, no kuras grib mantot?
    - Virsklase nevar būt pamatklase.
    - Katrai klasei ir viena virsklase.
    - Visām klasēm savā starpā ir unikāli vārdi.
    - Klašu definīcijai izmanto atslēgvārdu “class”.
    - Klases vārds nevar būt rezervētais vārds.
  + Lauku (atribūtu un funkciju) kļūdu pārbaude:
    - Laukam ir datu tips “Integer”, “String”, “Boolean” vai “Real”. Metodēm var būt arī “Void”.
    - Lauku vārdi savā starpā ir unikāli, kā arī nedrīkst sakrist ar klases vārdu un asociāciju lomas vārdiem, bet drīkst sakrist ar virsklases vārdu.
    - Lauku vārdi ir identifikatori.
    - Lauka vārds nevar būt rezervētais vārds, ka arī nedrīkst sākties ar “\_constructor\_”.
    - Lauki ir atdalīti ar semikolu.
    - Metožu argumentu vārdi drīkst sakrist ar lauku vārdiem, asociāciju galapunktu vārdiem, kā arī ar klases vārdu.
    - Metožu arguments sastāv no datu tipa un vārda, un visi argumenti savā starpā ir atdalīti ar komatu.
    - Metožu argumentu vārdi savā starpā ir unikāli.
  + Kļūdu pārbaude starp klases un tās virsklašu laukiem (ko drīkst pieļaut):
    - Ja apakšklasē un virsklasē ir metode ar vienu un to pašu vārdu, tad to drīkst pieļaut, ja funkcijām ir identiski datu tipi un argumenti pēc skaita un to datu tipiem. URL drīkst būt dažādi un argumentu vārdi kopīgajās pozīcijās var būt dažādi.
    - Ja apakšklasē un virsklasē ir atribūts ar vienu un to pašu vārdu, tad to drīkst pieļaut, ja atribūtam ir identiski datu tipi.
    - Ja kāds no virsklases laukiem ir privāts, tad to nevajag pārbaudīt.
  + Asociāciju kļūdu pārbaude:
    - Asociācijā starp divām klasēm ir jābūt dažādiem lomu vārdiem skatoties gan no avota klases, gan no mērķa klases, ņemot vērā arī abu klašu virsklases un apakšklases.
    - Avotklasē asociācijas mērķa lomas vārds nedrīkst sakrist ar lauku vārdiem un avotklases vārdu, bet drīkst sakrist ar avotklases virsklases vārdu.
    - Mērķklasē asociācijas avota lomas vārds nedrīkst sakrist ar lauku vārdiem un mērķklases vārdu, bet drīkst sakrist ar mērķklases virsklases vārdu.
    - Associācijas lomu vārds nevar būt rezervētais vārds, ka arī nedrīkst sākties ar “\_constructor\_”.
    - Asociācijas lomu vārdi un klases vārdi ir identifikatori.
  + Ja jaunkodā tiek konstatēta kaut viena kļūda, tad starpkods netiek ģenerēts.

## 1.6 Vispārējie ierobežojumi

* Dažādi OS (Windows 10, Windows 11, Linux Ubuntu, MacOS 10, MacOS 11 ).
* Dažādu procesoru arhitektūras (AMD64, ARM64).
* Java 11 (WebAppOS) un .NET 5.0 (Adapterim un kodu ģeneratoram).

## 1.7 Pieņēmumi un atkarības

* Izmanto trešo pušu kodu ģeneratorus ar atvērtu pirmkoda licenci.
* Uzģenerētajam kodam jādarbojas iepriekš minētajās operētājsistēmās.
* Izstrādātājam jāspēj iemācīties WAOS C# programmēšanas valoda.
* Izstrādātāji var palaist programmas, izmantojot komandrindu.
* Kods WAOS C# programmēšanas valodā ir sintaktiski pareizs, piemēram, argumenti nav atdalīti ar diviem komatiem, bet drīkst pieļaut gadījumus, kad kaut kas nav ievadīts, piemēram, funkcijai nav norādīts ne vārds, ne datu tips. Tādā gadījumā kompilatoram ir jāfiksē to kā kļūdu.
* WAOS C# kodā iekavām ir jābūt pareizi pabeigtām. Tas nozīmē, ka ne tikai tiek ievērota sintakse, bet arī nedrīkst būt tā, ka ir tikai atverošā vai aizveroša iekava. Ir jābūt abām iekavām.
* WAOS C# kodā netiek lietoti C# rezervētie vārdi, piemēram, int, switch, interface, virtual, lai definētu lauku, klašu un asociācijas galapunktu vārdus.

# 2. PROGRAMMATŪRAS PRASĪBU SPECIFIKĀCIJA

## 2.1 Funkcionālās prasības

### 2.1.1 WAOS C# sintakse

#### 2.1.1.1 Bloka sintakse

WAOS C# tiek definēti bloki. Katram blokam ir šāda sintakse:

* *blokaTips blokaĶermenis*

Kur *blokaTips* ir vai nu bloka tips (*class, association*), aizsardzība (*public, private*) vai datu tips (*Integer, String, Boolean, Real, Void*) un *blokaĶermenis* ir vai nu asociācija [skat. 2.1.1.2 nodaļu] vai klase [skat. 2.1.1.3 nodaļu]. Ja bloka tips ir aizsardzība vai datu tips, tad kompilatoram to ir jāuztver kā kļūdu.

#### 2.1.1.2 Asociācijas sintakse

Asociācijai ir šāda sintakse:

* *(avotaVārds:avotaKlase bultas mērķaVārds:mērķaKlase)*

Kur *avotaVārds, avotaKlase, mērķaVards* un *mērķaKlase* ir identifikators, bloka tips (*class, association*), aizsardzība (*public, private*) vai datu tips (*Integer, String, Boolean, Real, Void*). Ja kāds no iepriekš minētajiem mainīgajiem nav identifikators, tad kompilatoram to ir jāuztver kā kļūdu. Asociāciju definēšanai izmanto bloka tipu *association*.

*Bultās* var būt vienā no diviem formātiem: ‘<>-’ vai ‘<->’, kur ir dota vismaz viena svītriņa un svītriņu skaits ir neierobežots. Bultas formātā ‘<->’ nozīmē, ka kompozīcijas nav, bet bultas formātā ‘<>-’ – ka kompozīcija ir.

#### 2.1.1.3 Klases sintakse

Klasei ir šāda sintakse:

* *klasesVārds : virsklasesVārds { lauki }*

Kur *klasesVārds* un *virsklasesVārds* ir identifikators. Klasēm izmanto bloka tipu *class*.

Laukiem ir nepieciešama lauka saturs un semikols [skat. 2.1.1.4 nodaļu].

#### 2.1.1.4 Lauka sintakse

Lauka saturam ir šāda sintakse:

* *anotācija laukaDefinīcija;*

Kur var būt vairākas anotācijas. Anotācijas sintakse tiek aprakstīta nodaļā 2.1.1.5 un lauka definīcijas sintakse – 2.1.1.6 nodaļā.

#### 2.1.1.5 Anotācijas sintakse

Anotācijai ir šāda sintakse:

* *[tips(“vērtība”)]*

Kur *tips* ir identifikators, bloka tips (*class, association*), aizsardzība (*public, private*) vai datu tips (*Integer, String, Boolean, Real, Void*). Ja *tips* nav identifikators, tad kompilatoram to ir jāuztver kā kļūdu.

*Vērtība* ir jebkāda simbolu virkne, kas nav atdalīta ar tukšumiem un kas nesastāv no dubultajām pēdiņām vai iekavām. *Vērtība* var būt, var nebūt starp pēdiņām, bet, ja tā nav starp pēdiņām, tad kompilatoram to ir jāuztver kā kļūdu.

#### 2.1.1.6 Lauka definīcijas sintakse

Lauka definīcijai ir šāda sintakse

* *aizsardzība datuTips vārds ( argumenti )*

Kur *aizsardzība* ir aizsardzība (*public, private*), *datuTips* ir datu tips (*Integer, String, Boolean, Real, Void*) vai bloka tips (*class, association*) un *vārds* ir identifikators. Ja *datuTips* ir bloka tips, tad kompilatoram to ir jāuztver kā kļūdu.

*Argumenti* sastāv no argumentu definīcijām un komatiem. Argumenta definīcijas sintakse ir aprakstīta nodaļā 2.1.1.7.

#### 2.1.1.7 Argumenta definīcijas sintakse

Argumentu definīcijai ir šāda sintakse:

* *datuTips vārds*

Kur *datuTips* ir datu tips (*Integer, String, Boolean, Real, Void*), aizsardzība (*public, private*) vai bloka tips (*class, association*) un *vārds* ir identifikators. Ja *datuTips* ir aizsardzība vai bloka tips, tad kompilatoram to ir jāuztver kā kļūdu.

#### 2.1.1.8 Anotāciju vērtību sintakse atkarībā no tās tipa

Šeit ir aprakstītas anotāciju vērtību sintakses tām anotācijām, kuram tās ir specifiskas.

|  |  |  |
| --- | --- | --- |
| Anotācijas tips | Anotācijas vērtības sintakse | Piezīmes |
| URL | *protokols:lokācija:ceļš* | *protokols* – metodes protokols  *lokācija* – metodes lokācija  *ceļš* – metodes ceļš (katrai programmēsanas valodai tiek definēts citādāk). |

*2.1. tabula.* **Anotāciju vērtību sintakse**

#### 2.1.1.9 URL protokolu un lokāciju uzskaitījums

|  |  |  |
| --- | --- | --- |
| Protokols | Programmēšanas valoda | Piezīmes |
| dotnet | C# un citas valodas, ko atbalsta .NET | Jāizmanto .NET 5. |
| python3 | Python | Versija 3.X |
| staticJava | Java | Tiek atbalstītas tikai statiskas metodes |
| … | … | Varētu nākotnē būt arī citi protokoli |

*2.2. tabula.* **URL protokolu uzskaitījums**

|  |  |
| --- | --- |
| Lokācija | Paskaidrojums |
| local | Kods atrodas lokālā failu sistēmā |
| http | Kods atrodas uz HTTP servera. Pirms palaišanas vajag lejupielādēt. |
| ftp | Kods atrodas uz FTP servera. Pirms palaišanas vajag lejupielādēt vai *mount*. |
| … | Nākotne varētu būt citas lokācijas |

*2.3. tabula.* **URL lokāciju uzskaitījums**

### 2.1.2 Datu struktūru konceptuālais apraksts

#### 2.1.2.1 Kompilatora datu struktūras

Kompilatora objekts ir atbildīgs par visām datu struktūrām un funkcijām, kuras kompilatoram ir jāveic. Kompilatorā tiek glabāta informācija par klasēm, asociācijām un visām kompilēšanas laikā pieļautajām kļūdām. Kļūdas tiek glabātas kā simbolu virknes, savukārt klases un asociācijas – kā objekti.

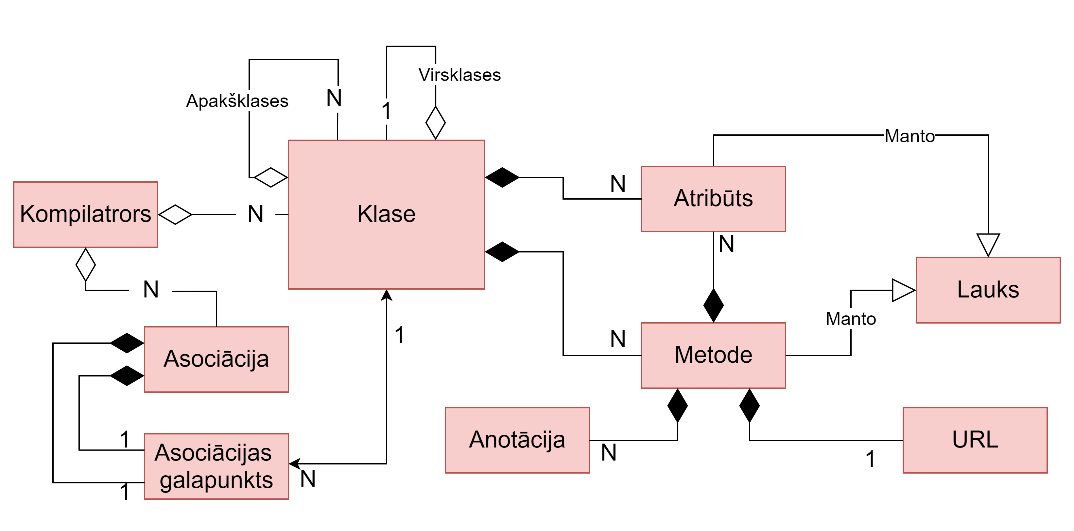
Asociācijām ir nepieciešama informācija par diviem asociācijas galapunktiem – avotu un mērķi. Katram asociācijas galapunktam ir lomas vārds un klase, kurai galapunkts ir piesaistīts. Asociācijā glabā arī to, vai asociācija ir kompozīcija.

Klasēs glabāsies visas metodes, atribūti un asociācijas galapunkti, kuri klasei tiek definēti, kā arī virsklase un apakšklases, ja tādas ir, un klases vārds.

Visi atribūti un metodes tiek uzskatīti par laukiem un laukos glabājas kopīgā informācija par metodēm un atribūtiem, kas iekļauj datu tipu, aizsardzību un vārdu. Metodēm vēl ir nepieciešams glabāt anotācijas, URL, argumentus un atgriežamo vērtību, bet metodēm – tikai iegūstamo vērtību.

Anotācijām ir nepieciešams zināt, kāda tipa anotācija tā ir un kāda vērtība tai tika dota.

URL glabā informāciju par protokolu jeb programmēšanas valodu, kurā ir rakstīts orģinālkods, lokācija, kurā norāda, kur atrodas orģinālkods un metodes ceļš, kurš norāda to, kur orģinālkodā atrodas metode.

**

*2.1. att.* **Kompilatora datu struktūru konceptuālā objektu diagramma**

#### 2.1.2.2 WebMemory datu struktūras

Nodaļā 2.1.8 tiek aprakstīts WebMemory jeb augsta līmeņa RAAPI modulis, kas spēj veikt RAAPI funkcijas tikai augstā līmenī. Gluži kā RAAPI, arī WebMemory ir jāglabā informācija par visām RAAPI klasēm, turpmāk tekstā – WebKlases, tās atribūtiem turpmāk tekstā – WebAtribūti, objektiem, turpmāk tekstā – WebObjekti, un asociācijām, turpmāk tekstā – WebAsociācijas, kuras tajā atrodas. Datu struktūras, kuras tiek izmantotas WebMemory, tiek izmantotas arī RAAPI. Pamatstruktūru arī sauc par WebMemory un tajā ir jāglabā WebKlases.

WebKlase ir vissvarīgākā WebMemory datu struktūra, jo tā ir vienīgā struktūra, kurai ir saistībā ar visām pārējām datu struktūrām. Katrai klasei ir vairāki WebAtribūti, WebObjekti un WebAsociācijas. WebKlasēm var būt arī virsklases un apakšklases.

WebAtribūti glabā datu tipu, vārdu un vērtību. WebAtribūtu vertībam var piekļūt caur WebObjektiem [skat. 2.1.8.7 un 2.1.8.8 nodaļas].

WebAsociācijas galapunkti glabā galapunkta lomas vārdu, avotklasi un mērķklasi.

WebObjekti glabā atsauces uz klasēm, no kurām objekts ir veidots. WebObjekti var būt piesaistīti citiem WebObjektiem caur asociācijām. Lai tos iegūtu, ir nepieciešams pats WebObjekts, kā arī asociācijas galapunkts, jo saistītie objekti var atšķirties ar WebAsociāciju galapunktiem. Tiek lietota RAAPI funkcija *getIteratorForLinkedObjects*, kur jāpadod atsauce uz objektu, kurai jāiegūst saistītie objekti, un lomas vārds asociāciju galapunktam, kuras objektus ir jāiegūst.



*2.2. att.* **WebMemory datu struktūru konceptuālā objektu diagramma**

### 2.1.3 Starpkoda sintakse

#### 2.1.3.1 Klases “BaseObject” sintakse

Klasi “BaseObject” ģenerē šādi:

*using WebAppOS;*

*using System.Collections.Generic;*

*namespace vardtelpasVārds*

*{*

*public class BaseObject*

*{*

*protected static IWebMemory \_wm;*

*protected static IRemoteWebCalls \_wc;*

*public WebObject \_object;*

*protected BaseObject ( IWebMemory wm , IRemoteWebCalls wc )*

*{*

*\_wm = wm;*

*\_wc = wc;*

*}*

*protected BaseObject ( IWebMemory wm , IRemoteWebCalls wc , long rObject )*

*{*

*\_wm = wm;*

*\_wc = wc;*

*}*

*protected BaseObject ( IWebMemory wm )*

*{*

*\_wm = wm;*

*}*

*protected bool checkClass( List<string> attributes , string className )*

*{*

*var c = \_wm.FindClassByName( className );*

*if (c == null)*

*{*

*c = \_wm.CreateClass( className );*

*}*

*Else { return true; }*

*for(int x=0; x<attributes.Count; x+=2)*

*{*

*checkAttribute( attributes[x] , attributes[x+1] , c );*

*}*

*}*

*protected void checkAttribute( string name , string type , WebClass c )*

*{*

*var a = c.FindAttributeByName( attributes[x] );*

*if (a == null)*

*{*

*a = c.CreateAttribute( attributes[x] , attributes[x+1] );*

*}*

*protected void checkAssociationEnd( string sourceName , string targetName , string sourceClass , string targetClass , string Composition)*

*{*

*var cSource = \_wm.FindClassByName( sourceClass );*

*var cTarget = \_wm.FindClassByName( targetClass );*

*var a = cSource.FindTargetAssociationEndByName( targetName );*

*if (a == null)*

*{*

*bool isComposition;*

*if (Composition == “true”) { isComposition = true; }*

*else { isComposition = false; }*

*cSource.CreateAssociationEnd(cTarget,sourceName,targetName,isComposition );*

*}*

*}*

*protected void konstruktors()*

*{*

*List<string> attributes = new() { atribūtuDati };*

*var o = checkClass( attributes , " klasesVārds " );*

*if( o == false )*

*{*

*// SuperClass Check*

*konstruktorsVirsklase( );*

*var c = \_wm.FindClassByName( “klasesVārds” );*

*c.CreateGeneralization( “virsklasesVārds” );*

*// Association classes check*

*List<string> associations = new() { asociācijuDati };*

*konstruktorsAsociācija ( );*

*for ( int x = 0; x < associations.Count; x+=4 )*

*{*

*checkAssociationEnd( associations[x] , associations[x+1] , "klasesVārds" , associations[x+2] , associations[x+3] );*

*}*

*}*

*}*

*}*

*}*

Kur *vārdtelpasVards* ir identifikators.

Katrai klasei ir sava konstruktora funkcija, kuras sintakse ir dzeltenajā krāsā iekrāsotais kods. *Konstruktors* ir rakstīts formātā “\_constructor\_*klasesVārds*”, kur *klasesVārds* ir vārds klasei, kurai konstruktora funkcija ir veidota.

*AtribūtuDati* ir attribūtu dati un *asociācijuDati* ir asociāciju dati.

Atribūtu dati glabājas sarakstā *attributes,* kur ik pa divām simbolu virknēm ir viens atribūts. Pirmais elements ir atribūta vārds, bet otrais ir primitīvais datu tips.

Asociāciju dati glabājas līdzīgi atribūtu datiem, bet ik pa četrām simbolu virknēm ir viena asociācija. Pirmais elements ir avota lomas vārds, otrais – mērķa lomas vārds, trešais – mērķa klases vārds un ceturtais – kompozīcijas patiesumvērtība.

Koda daļā, kas ir iekrāsota dzeltenajā krāsa, ir zarojums “*if(o == false)*”. Šo zarojumu neģenerē, ja klasei nav nevienas asociācijas un virsklases.

*virsklasesVārds* ir virsklases vārds. Koda daļu zem “*//SuperClass check*” neģenerē, ja klasei nav virsklases. *KonstruktorsVirsklase* ir rakstīts formātā ” “\_constructor\_*virsklasesVārds*”.

Koda daļu zem “*//Association classes check*” neģenerē, ja klasei nav asociāciju. Šajā daļā var būt vairākas koda rindiņas *konstruktorsAssociācija(),* kur *konstruktorsAsociācija* ir rakstīts formātā“\_constructor\_*asociācijasKlase*”*,* kur *asociācijasKlase* ir asociācijas galapunkta klases vārds. Šādas rindiņas neģenerē, ja *asociācijasKlase* sakrīt ar *klasesVārds* vai ar kādu no tās virsklasēm.

#### 2.1.3.2 Klases sintakse

Klasēm ir šāda sintakse:

*using WebAppOS;*

*using System.Text.Json;*

*using System;*

*using System.Collections.Generic;*

*namespace vardtelpasVārds*

*{*

*class klasesVārds : virsklasesVārds*

*{*

*}*

*}*

Kur *vārdtelpasVards* ir identifikators, *klasesVārds* ir ģenerējamās klases vārds un *virsklasesVārds* ir ģenerējamās klases virsklases vārds (ja klasei nav virsklases, tad kā virsklasi izmanto klasi “BaseObject”). Iekšā klasē tiek definēti vairāki lauki, kurus var skatīt nākamajās nodaļās.

#### 2.1.3.3 Klases konstruktoru sintakse

Klases konstruktoriem ir šāda sintakse:

*public klasesVārds ( IWebMemory wm , IRemoteWebCalls wc ) : base( wm , wc )*

*{*

*konstruktors();*

*\_object = \_wm.FindClassByName( “klasesVārds” ).CreateObject();*

*}*

*public klasesVārds ( IWebMemory wm, IRemoteWebCalls wc , long rObject ) : base( wm , wc , rObject )*

*{*

*konstruktors();*

*\_object = new( rObject, wm );*

*}*

Kur *klasesVārds* ir ģenerējamās klases vārds un *konstruktors* ir rakstīts formātā “\_constructor\_*klasesVārds*”.

#### 2.1.3.4 Klases atribūtu sintakse

Klases atribūtiem ir šāda sintakse:

*aizsardzība datuTips atribūtaVārds*

*{*

*get { return atgriežamāVērtība; }*

*set { \_object["atribūtaVārds"] = Convert.ToString( value ); }*

*}*

Kur *aizsardzība* ir atribūta aizsardzības tips, *datuTips* ir atribūta datu tips, *atribūtaVārds* ir atribūta vārds un *klasesVārds* ir klases, kura ģenere atribūtu, vārds un *atgriežamāVērtība* ir vērtība, kuru ‘get’ funkcijai ir jāatgriež. Ir šādi varianti atkarībā no atribūta datu tipa:

* *Convert.ToInt64( \_object["atribūtaVārds"] )**,* ja *datuTips* ir ’long’;
* *\_object["atribūtaVārds"] ,* ja *datuTips* ir ’string’;
* *Convert.ToBoolean( \_object["atribūtaVārds"] )* *,* ja *datuTips* ir *’*bool*’;*
* *Convert.ToDouble( \_object["atribūtaVārds"] ) ,* ja *datuTips* ir ’double’;

#### 2.1.3.5 Klases asociāciju galapunktu sintakse

Klases asociācijas galapunktiem ir šāda sintakse

*public List<galapunktaKlase> galapunktaLomasVārds*

*{*

*get*

*{*

*var c = \_wm.FindClassByName(“klasesVārds”);*

*var a = c.FindAssociationEnd (“galapunktaLomasVārds”);*

*var list = \_object.LinkedObjects(a);*

*List<galapunktaKlase> result = new();*

*foreach (var l in list)*

*{*

*result.Add( new galapunktaKlase( \_wm , \_wc , l.GetReference ));*

*}*

*return result;*

*}*

*set*

*{*

*var c = \_wm.FindClassByName(“klasesVārds”);*

*var a = c.FindAssociationEnd (“galapunktaLomasVārds”);*

*var list = value;*

*List<WebObject> result = new();*

*foreach (var l in list)*

*{*

*result.Add( l.\_object );*

*}*

*\_object.LinkObjects(a,result);*

*}*

*}*

Kur *galapunktaKlase* ir asociācijas galapunkta klases vārds, *galapunktaLomasVārds* ir asociācijas galapunkta lomas vārds. *klasesVārds* ir klases, kurā tiek veidots asociācijas galapunkts, vārds.

#### 2.1.3.6 Klases metožu sintakse

Klases metodēm ir šāda sintakse

*aizsardzība datuTips metodesVārds ( argumenti )*

*{*

*string arguments = JsonSerializer.Serialize( new { argumentuVārdi } );*

*string result = \_wc.WebCall( \_wm.GetTDAKernel() , \_object.GetReference , " metodesVārds " , arguments );*

*var json = JsonDocument.Parse(result);*

*JsonElement errorMessage;*

*if (json.RootElement.TryGetProperty("error", out errorMessage) == true)*

*{*

*throw new Exception(errorMessage.GetString());*

*}*

*else*

*{*

*var r = json.RootElement.GetProperty("result");*

*return atgriežamāVērtība;*

*}*

*}*

Kur *aizsardzība* ir metodes aizsardzības tips, *datuTips* ir metodes datu tips, *metodesVārds* ir metodes vārds, *argumenti* ir metodes argumenti, atdalīti ar komatiem, *argumentuVārdi* ir argumentu vārdi bez datu tipiem, atdalīti ar komatiem un *atgriežamāVērtība* ir vērtība, kuru metodei ir jāatgriež. Ir šādi varianti atkarībā no atribūta datu tipa:

* *r.GetInt64(),* ja *datuTips* ir ’long’;
* *r.GetString(),* ja *datuTips* ir ’string’;
* *r.GetBoolean(),* ja *datuTips* ir *’*bool*’*;
* *r.GetDouble(),* ja *datuTips* ir ’double’;

Koda daļu, kas ir iekrāsota dzeltenajā krāsa, neģenerē, ja metodes datu tips ir void.

#### 2.1.3.7 Datu tipi starp jaunkodu un starpkodu

|  |  |
| --- | --- |
| Datu tips jaunkodā (primitīvais datu tips) | Datu tips starpkodā (datu tips) |
| Integer | long |
| String | string |
| Boolean | bool |
| Real | double |
| Void | void |

*2.2. tabula.* **Datu tipu saikne starp jaunkodu un starpkodu**

### 2.1.4 Funkciju sadalījums pa moduļiem/komponentiem

Sistēmu var sadalīt divās daļās

* Kodu ģenerātors, kurš atblid par jaunkoda kompilēšanu un starpkoda ģenerēšanu.
* Adapteris, kuru izmanto funkciju izsaukšanai WebAppOS vidē.

Kodu ģeneratora daļai ir 2 moduļi:

* Kompilatora modulis – atbild par jaunkoda apstrādi jeb kļūdu atrašanu jaunkodā un datu iegūšanu no jaunkoda (klases un tās atribūti, metodes, asociācijas).
* Ģeneratora modulis – atbild par starpkoda ģenerēšanu C# programmēšanas valodā, izmantojot datus, kas iegūti no kompilatora moduļa.

**

*2.3. att.* **1. līmeņa DPD diagramma (koda ģeneratora daļa)**

Adaptera daļai ir 4 moduļi:

* Ģenerētā koda modulis – uzģenerētā pakotne ar C# klasēm, kas savieno C# kodu ar WebMemory moduli. Tajā būs klases, kuras atbilst datiem, kas atrodas WebMemory modulī.
* WebMemory modulis – apstrādā klases, objektus, atribūtus un asociācijas, kuras glabājas šajā modulī. Izmanto RAAPI.
* LocalWebCalls moduilis – atbild par funkciju izsaukšanu, kuras rakstītas C# programmēšanas valodā. Šajā modulī strādā tikai ar dotnet protokolu un *local* kodiem.
* RemoteWebCalls modulis – atbild par funkciju izsaukšanu, kuras rakstītas citās programmēsanas valodās. Izmanto funkciju “*enqueueInnerWebCall*”, kura kvalifikācijas darba ietvaros jau ir dota.

2.2. attēlā ir redzams cikls starp WebAppOS lietotāja grupu, LocalWebCalls moduli, Izstrādātāju kodu un Ģenerētā koda moduli. Šis cikls apraksta metodes izsaukšanu WebAppOS vidē. Ciparu identifikācija ir šāda:

1. Izsaucamās funkcijas dati.
2. Dinamiskā izsaukuma dati (padots, izmantojot refleksiju).
3. Izsaukuma dati, kas padoti C# sintaksē.
4. Izsaukuma rezultāts, kas padota kā C# funkcijas atgriežamā vērtība.
5. Dinamiskā izsaukuma rezultāts (C# atgriežamā vērtība).
6. Izsaucamās funkcijas rezultāts.

Cipari 3’ un 4’ attiecas uz atribūtu un asociāciju galapunktu vērtību iegūšanu/uzstādīšanu, kur 3’ identificējas kā ”WebMemory vaicājuma dati” un 4’ – “WebMemory vaicājuma rezultāts”.

**

*2.4. att.* **1. līmeņa DPD diagramma (adaptera daļa)**

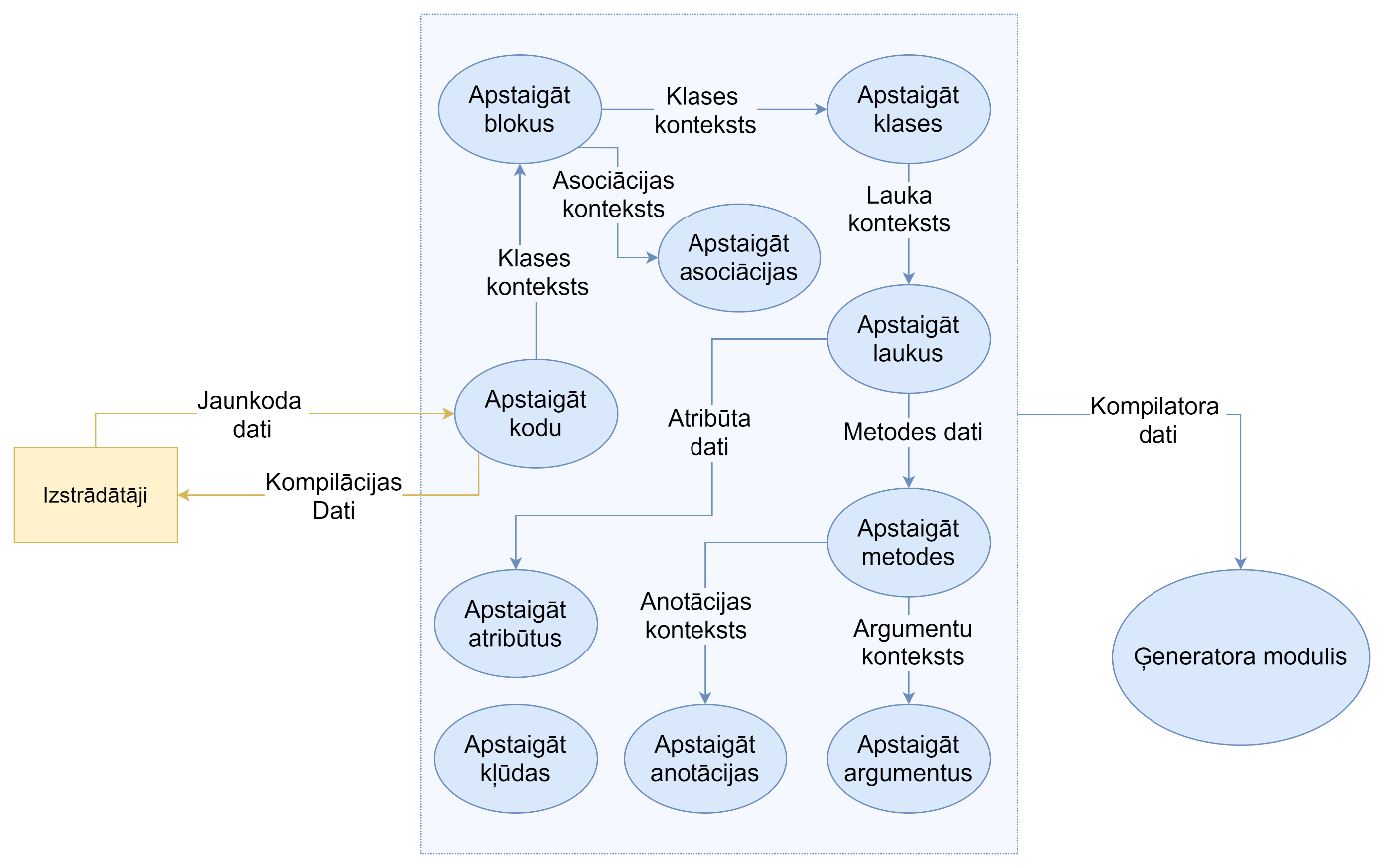
### 2.1.5 Kompilatora modulis

Šo moduli izmantos tikai izstrādātāju grupa, jo tiek padots jaunkods, kurš tiek kompilēts. Uzdevums ir atrast kļūdas jaunkodā. Visas kļūdas (izņemot vārdtelpas kļūdu) tiek saglabātas formātā

* At line *‘rindas numurs’*: *‘kļūdas paziņojums’*.

Pēc koda kompilēšanas, visi dati par klasēm un asociācijām tiek padoti ģeneratora modulim un tas veic starpkoda ģenerēšanu, ja nav atrasta neviena kļūda. Visas funkcijas ir sava starpā saistītas, jo no jaunkoda veido koku [skat. 3.1 nodaļu]. Visam funkcijām kā ievaddati tiek padoti konteksti.

Ja datu struktūrā kāda no īpašībām tiek glabāta kā objekts (neskaitam objektu sarakstus), tad vērtības saglabāšana notiek pēc apstrādes. Ja datu struktūrā kāda no īpasībām tiek glabāta kā primitīvais datu tips, tad vērtību saglabā pirms apstrādes [skat. 3.2.1.1 nodaļu].

* 2.5. att.* **2. līmeņa DPD diagramma kompilatora modulim**

#### 2.1.5.1 Apstaigāt kodu

**Nolūks**

Funkcija nepieciešama, lai veiktu kompilēšanas pamatfunkcijas.

**Ievade**

Funkcijai ir šādi ievaddati:

* Koda konteksts jeb jaunkoda fails;
* Vārdtelpa (namespace).

**Apstrāde**

Tiek veiktas šādas pārbaudes:

* Vai vārdtelpa ir padota pareizā formātā?
* Vai ir atrastas kļūdas pirmkodā?

Uzsākot kompilēšanu tiek izveidoti šādi saraksti:

* Klašu saraksts;
* Kļūdu saraksts;
* Asociāciju saraksts.

**Izvade**

Izvade ir atkarīga no apstrādes:

* Ja vārdtelpa ir nepareizā formātā tad saglabā kļūdas paziņojumu “Vārdtelpa ir nepareizā formātā”.
* Ja kompilēšanas rezultātā ir atrasta kaut viena kļūda, tad tās visas tiek izdrukātas.
* Ja kompilēšanas rezultātā nav atrastas kļūdas, tad notiek starpkoda ģenerēšana.

#### 2.1.5.2 Apstaigāt blokus

**Nolūks**

Funkcija nepieciešama, lai apstaigātu un pārbaudītu koda blokus.

**Ievade**

Funkcijai ir šādi ievaddati:

* Bloka konteksts;

**Apstrāde**

Tiek veiktas šādas pārbaudes:

* Vai blokam ir ievadīts tips?
  + Vai blokam ir pareizs tips (klase vai asociācija)?
* Vai blokam ir ievadīta definīcija?
  + Kāds bloks tiek definēts (klases vai asociācijas)?

**Izvade**

Izvade ir atkarīga no apstrādes:

* Ja blokam ir padots tips, tad tas tiek pārbaudīts.
  + Ja bloka tips nav klase vai asociācija, tad tiek saglabāts kļūdas paziņojums “*’padotais tips’* nav bloka tips. Izmantojiet ()”. Iekavu vietā liek “klasi”, ja ir dots klases ķermenis, “asociāciju”, ja ir dots asociācijas ķermenis un “klasi vai asociāciju”, ja ķermeņa nav.
* Ja bloka tips nav padots, tad saglabājas kļūdas paziņojums “Trūkst atslēgvārda *‘class’*” vai “Trūkst atslēgvārda *‘association’*” atkarībā no tā, vai tiek definēta klase vai asociācija.
* Ja blokam ir definīcija tad pārbauda, vai blokam ir tips.
  + Ja tips ir dots un ir dots pareizs tips, tad skatās, kāda ir tipa un definīcijas kombinācija:
    - Klases tips un klases definīcija – izsaucam funkciju “Apstaigāt klasi” [skat. 2.1.5.4 nodaļu].
    - Asociācijas tips un asociācijas definīcija – izsaucam funkciju “Apstaigāt asociāciju” [skat. 2.1.5.3 nodaļu].
    - Klases tips un asociācijas definīcija – saglabāts kļūdas paziņojums “Klasei ir dota asociācijas definīcija”.
    - Asociācijas tips un klases definīcija – saglabāts kļūdas paziņojums “Asociācijai ir dota klases definīcija”.
  + Ja tips ir dots, bet nav dots pareizs tips, tad kombinācijas netiek skatītas.
  + Pēc tipa pārbaudēm definīcija tiek apstaigāta [skat. 2.1.5.3 un 2.1.5.4 nodaļas].

#### 2.1.5.3 Apstaigāt asociāciju

**Nolūks**

Funkcija nepieciešama, lai apstaigātu un pārbaudītu asociācijas definīciju.

**Ievade**

Funkcijai ir šādi ievaddati:

* Asociācijas konteksts;

**Apstrāde**

Pirms funkcijas uzsākšanas tiek izveidota jaunas asociācijas un tās galapunktu instances.

Tiek veiktas šādas pārbaudes:

* Vai asociācijai ir doti dati par avotu?
  + Vai avotam ir pareizs lomas vārds?
  + Vai avotam ir pareizs klases vārds?
  + Vai klases vārds un lomas vārds ir atdalīti ar kolu?
* Vai asociācijai ir doti dati par mērķi?
  + Vai mērķim ir pareizs lomas vārds?
  + Vai mērķim ir pareizs klases vārds?
  + Vai klases vārds un lomas vārds ir atdalīti ar kolu?
* Vai asociācijai ir bultas?

Vispirms tiek pārbaudīts asociācijas avots un mērķis kopumā, tad tiek pārbaudītas klases un pēc klašu pārbaudēm tiek pārbaudīti lomu vārdi.

**Izvade**

Izvade ir atkarīga no apstrādes:

* Klašu apstrāde:
  + Ja klases vārds nav dots, tad saglabā kļūdas paziņojumu “trūkst klases”, citādi apstaigā klasi.
    - Ja klase vārds sakrīt ar rezervētajiem vārdiem [skat 1.5 nodaļu], tad saglabā kļūdas paziņojumu “klasi nevar saukt par *‘rezervētais vārds’*”.
    - Ja klase ar doto vārdu neeksistē, tad saglabā kļūdas paziņojumu “nav klases *‘klases vārds’*, kuru izmantot kā avota/mērķa klasi”.
  + Ja kļūdu nav, tad klasi saglabā asociācijas avota galapunktā, ja tiek pārbaudīta avotklase vai mērķa galapunktā, ja tiek pārbaudīta mērķklase..
* Lomas vārdu apstrāde:
  + Ja lomas vārds nav dots, tad saglabā kļūdas paziņojumu “trūkst lomas vārda”.
    - Ja lomas vārds sakrīt ar klases vārdu, tad saglabā kļūdas paziņojumu “Asociācijas lomas vārds nevar būt *‘klases vārds’*”
    - Ja lomas vārds sakrīt ar rezervētajiem vārdiem [skat 1.5 nodaļu], tad saglabā kļūdas paziņojumu “lomas vārds nevar būt *‘rezervētais vārds’*”.
    - Ja avota/mērķa lomas vārds sakrīt ar kādu no mērķa/avota klases vai virsklases, vai apakšklases atribūtu, funkciju vai citu avota/mērķu vārdiem, tad saglabā kļūdas paziņojumu “klasē jau eksistē atribūts/funkcija/asociācijas galapunkts ar nosaukumu *‘lomas vārds’*”, norādot rindu, kur padotais vārds jau tiek izmantots. Virsklasēs pievērs uzmanību tikai publiskiem laukiem.
  + Ja kļūdu nav, tad galapunktā saglabā lomas vārdu, tās rindu, asociācijas ID, rindu un galapunktu saglabā asociācijā.
* Ja asociācijas definīcijā nav bultas, tad saglabā kļūdas paziņojumu “Trūkst bultu”. Citādi asociācijā saglabā kompozīcijas patiesumvērtību.
* Ja avota vai mērķa definīcijā starp lomas vārdu un klases vārdu nav kola, tad saglabā kļūdas paziņojumu “Trūkst kola”.
* Apstrādes beigās asociāciju saglabā kompilatorā, ja tajā ir saglabāts vismaz viens galapunkts.

#### 2.1.5.4 Apstaigāt klasi

**Nolūks**

Funkcija nepieciešama, lai apstaigātu un pārbaudītu klases definīciju.

**Ievade**

Funkcijai ir šādi ievaddati:

* Klases konteksts;

**Apstrāde**

Pirms funkcijas uzsākšanas tiek izveidota jaunas klases instance.

Tiek veiktas šādas pārbaudes:

* Vai klasei ir dots pareizs klases vārds?
* Vai klasei ir dots pareizs virsklases vārds?
* Vai klasei ir dots ķermenis?

**Izvade**

Izvade ir atkarīga no apstrādes:

* Klases vārda apstrāde:
  + Ja klases vārds nav dots, tad saglabā kļūdas paziņojumu “trūkst klases vārda”.
  + Ja klases vārds sakrīt ar rezervētajiem vārdiem [skat 1.5 nodaļu], tad saglabā kļūdas paziņojumu “klasi nevar saukt par *‘rezervētais vārds’*”.
  + Ja klase ar doto vārdu eksistē, tad saglabā kļūdas paziņojumu “Klase ar vārdu *‘klases vārds’* jau eksistē”, norādot rindu, kur padotais klases vārds jau tiek izmantots.
* Virsklases apstrāde:
  + Ja virsklases vārds nav dots, bet ir dots kols, tad saglabā kļūdas paziņojumu “trūkst virsklases vārda”.
  + Ja virsklases vārds sakrīt ar rezervētajiem vārdiem [skat 1.5 nodaļu], tad saglabā kļūdas paziņojumu “Virsklasi nevar saukt par *‘rezervētais vārds’*”.
  + Ja virsklase ar doto vārdu neeksistē, tad saglabā kļūdas paziņojumu “Klase ar vārdu *‘klases vārds’* neeksistē”, norādot rindu, kur padotais vārds jau tiek izmantots.
  + Ja virsklases vārds sakrīt ar pamatklases vārdu, tad saglabā kļūdas paziņojumu “Nevar mantot no klases ar tādu pašu vārdu”.
  + Ja kļūdu nav, tad klasē saglabā virsklases instanci.
* Ja klasei nav ķermeņa, tad saglabā kļūdas paziņojumu “Trūkst klases ķermeņa”.
* Ja klasei ir ķermenis, tad izsauc funkciju “Apstaigāt laukus” [skat. 2.1.5.5 nodaļu].
* Apstrādes beigās klasi saglabā kompilatorā, ja tai ir dots vārds.

#### 2.1.5.5 Apstaigāt laukus

**Nolūks**

Funkcija nepieciešama, lai apstaigātu un pārbaudītu klases laukus.

**Ievade**

Funkcijai ir šādi ievaddati:

* Lauku konteksts;

**Apstrāde**

Tiek veiktas šādas pārbaudes:

* Vai laukam ir semikols?
* Vai lauks ir atribūts vai metode?

**Izvade**

Izvade ir atkarīga no apstrādes:

* Ja lauks nebeidzas ar semikolu, tad saglabā kļūdas paziņojumu “Trūkst semikola”.
* Ja laukam ir definēta anotācija un/vai argumenti, tad lauks ir metode un tiek izsaukta funkcija “Apstaigāt metodi” [skat. 2.1.5.7 nodaļu]. Citādi lauks tiek uzskatīts par atribūtu un tiek izsaukta funkcija “Apstaigāt atribūtu” [skat. 2.1.5.6 nodaļu].

#### 2.1.5.6 Apstaigāt atribūtus

**Nolūks**

Funkcija nepieciešama, lai apstaigātu un pārbaudītu klases atribūtus.

**Ievade**

Funkcijai ir šādi ievaddati:

* Lauka konteksts (pirms tam noskaidrots, ka tajā nav anotācijas konteksta un nav definēti argumenti);

**Apstrāde**

Pirms funkcijas uzsākšanas tiek izveidota jauna atribūta instance.

Tiek veiktas šādas pārbaudes:

* Vai atribūtam ir aizsardzība?
* Vai atribūtam ir pareizs datu tips?
* Vai atribūtam ir pareizs vārds?

**Izvade**

Izvade ir atkarīga no apstrādes:

* Ja atribūtam ir aizsardzība, tad atribūtā tā tiek saglabāta. Citādi atribūta aizsardzības vērtība ir *public*.
* Datu tipa apstrāde:
  + Ja datu tips nav dots, tad saglabā kļūdas paziņojumu “atribūtam trūkst datu tipa”.
  + Ja nav dots pareizs datu tips, tad saglabā kļūdas paziņojumu “*’padotais datu tips’* nav derīgs datu tips.”
* Vārda apstrāde
  + Ja vārds nav dots, tad saglabā kļūdas paziņojumu “atribūtam trūkst vārda”.
  + Ja vārds sakrīt ar klases vārdu, tad saglabā kļūdas paziņojumu “*’padotais vārds’* nevar būt klases vārds”.
  + Ja vārds sakrīt ar rezervētajiem vārdiem [skat 1.5 nodaļu], tad saglabā kļūdas paziņojumu “atribūtu nevar saukt par *‘rezervētais vārds’*”.
  + Ja vārds jau ir sastopams klasē kā cita lauka vārds vai kā asociācijas galapunkta lomas vārds, tad saglabā kļūdas paziņojumu “klasē jau eksistē lauks/asociācijas galapunkts ar vārdu *‘padotais vārds’*”, norādot rindu, kur padotais vārds jau tiek izmantots.
* Virsklašu pārbaude:
  + Ja vārds jau ir sastopams kādā no tās virsklasēm kā cita publiska lauka vārds vai kā asociācijas galapunkta lomas vārds, tad saglabā kļūdas paziņojumu “klasē jau eksistē lauks/asociācijas galapunkts ar vārdu *‘padotais vārds’*”, norādot rindu, kur padotais vārds jau tiek izmantots.
  + Ja vārds tiek izmantots virsklasē kā cita publiska atribūta vārds, tad tiek pārbaudīts datu tips virsklases atribūtam. Ja tas nesakrīt, tad saglabā kļūdas paziņojumu “Atribūts *‘atribūta vārds’*, kas eksiste virsklasē *‘virsklases vārds’*, nav ar tādu pašu datu tipu”, norādot rindu, kur virsklasē ir definēts atribūts.
* Ja ne bāzes klasē, ne virsklasēs nav atrasts lauks vai asociācijas galapunkts ar padoto vārdu, tad šis vārds atribūtā tiek saglabāts.
* Apstrādes beigās atribūts tiek saglabāts klasē, ja tam ir dots vārds.

#### 2.1.5.7 Apstaigāt metodes

**Nolūks**

Funkcija nepieciešama, lai apstaigātu un pārbaudītu klases metodes.

**Ievade**

Funkcijai ir šādi ievaddati:

* Lauku konteksts (pirms tam noskaidrots, ka tajā ir anotācijas konteksts un/vai ir definēti argumenti);

**Apstrāde**

Pirms funkcijas uzsākšanas tiek izveidota jaunas metodes instance.

Tiek veiktas šādas pārbaudes:

* Vai metodei ir aizsardzība?
* Vai metodei ir pareizs datu tips?
* Vai metodei ir pareizs vārds?
* Vai metodei ir definētas anotācijas?
* Vai metodei ir definēti argumenti?

Metode tiek apstaigāta sekojošā secībā: argumenti, aizsardzība, datu tips, vārds, anotācijas.

**Izvade**

Izvade ir atkarīga no apstrādes:

* Ja metodei nav argumentu iekavu, tad saglabā kļūdas paziņojumu “Trūkst argumentu definīcijas”, citādi tiek palaista funkcija “Apstaigat argumentus” [skat. 2.1.5.9 nodaļu], ja ir tāda nepieciešamība.
* Ja atribūtam ir aizsardzība, tad atribūtā tā tiek saglabāta. Citādi atribūta aizsardzības vērtība ir *public*.
* Datu tipa apstrāde:
  + Ja datu tips nav dots, tad saglabā kļūdas paziņojumu “atribūtam trūkst datu tipa”.
  + Ja nav dots pareizs datu tips, tad saglabā kļūdas paziņojumu “*’padotais datu tips’* nav derīgs datu tips.”
* Vārda apstrāde
  + Ja vārds nav dots, tad saglabā kļūdas paziņojumu “atribūtam trūkst vārda”.
  + Ja vārds sakrīt ar klases vārdu, tad saglabā kļūdas paziņojumu “*’padotais vārds’* nevar būt klases vārds”.
  + Ja vārds sakrīt ar rezervētajiem vārdiem [skat 1.5 nodaļu], tad saglabā kļūdas paziņojumu “metodi nevar saukt par *‘rezervētais vārds’*”.
  + Ja vārds jau ir sastopams klasē kā cita lauka vārds vai kā asociācijas galapunkta lomas vārds, tad saglabā kļūdas paziņojumu “klasē jau eksistē lauks/asociācijas galapunkts ar vārdu *‘padotais vārds’*”, norādot rindu, kur padotais vārds jau tiek izmantots.
* Virsklašu pārbaude:
  + Ja vārds jau ir sastopams kādā no tās virsklasēm kā cita publiska lauka vārds vai kā asociācijas galapunkta lomas vārds, tad saglabā kļūdas paziņojumu “klasē jau eksistē lauks/asociācijas galapunkts ar vārdu *‘padotais vārds’*”, norādot rindu, kur padotais vārds jau tiek izmantots.
  + Ja vārds jau ir sastopams kādā no tās virsklasēm kā citas publiskas metodes vārds, tad abām metodēm tiek veiktas tālākas pārbaudes:
    - Ja metodēm nesakrīt datu tipi, tad saglabā kļūdas paziņojumu “Metode *‘metodes vārds’*, kas eksiste virsklasē *‘virsklases vārds’*, nav ar tādu pašu datu tipu”, norādot rindu, kur virsklasē ir definēta metode..
    - Ja metodēm nesakrīt argumentu skaits, tad tiek saglabāts kļūdas paziņojums “Virsklases definīcijā metodei ir atšķirīgs argumentu skaits” , norādot rindu, kurā metode ir definēta virsklasē.
    - Ja argumentu skaits ir vienāds, tad katram argumentam, kuras pozīcijas ir vienādas, pārbauda datu tipu. Ja datu tipi nesakrīt, tad tiek saglabāts kļūdas paziņojums “Arguemntam nr. *‘argumenta pozīcija’* nav tāds pats tips kā virsklasē”, norādot rindu, kurā arguments ir definēts virsklasē.
* Ja ne bāzes klasē, ne virsklasēs nav atrasts lauks vai asociācijas galapunkts ar padoto vārdu, tad šis vārds metodē tiek saglabāts.
* Ja metodei ir definētas anotācijas, tad katrai anotācijai izsauc funkciju “Apstaigāt anotācijas” [skat. 2.1.5.8 nodaļu].
* Ja metodei nav definēta URL anotācija, tad saglabā kļūdas paziņojumu “Nav definēts metodes URL”.
* Apstrādes beigās klasē saglabā metodi, ja tai ir dots vārds.

#### 2.1.5.8 Apstaigāt anotācijas

**Nolūks**

Funkcija nepieciešama, lai apstaigātu un pārbaudītu metodes anotācijas.

**Ievade**

Funkcijai ir šādi ievaddati:

* Anotācijas konteksts;

**Apstrāde**

Tiek veiktas šādas pārbaudes:

* Kāds ir anotācijas tips?
* Ja tips ir URL, tad vai vērtība ir pareizajā formātā?
* Vai ir izmantotas iekavas un pēdiņas tam paredzētajās vietās?

Ja anotācijas tips ir URL, tad metodē tiek veidota jauna URL instance, citādāk tiek veidota jaunas anotācijas instance.

**Izvade**

Izvade ir atkarīga no apstrādes:

* Ja trūkst kādas iekavas vai pēdiņas, tad saglabā kļūdas paziņojumu “trūkst iekavas/pēdiņas”.
* Ja anotācijas tips ir URL un metodei vēl nav definēta URL anotācija, tad tiek skatīta anotācijas vērtība.
  + Ja vērtībai nav URL specifiskie dati (protokola un lokācijas), tad tiek saglabāts kļūdas paziņojums “trūkst URL specifisko datu”, citādāk tie tiek skatīti.
    - Ja nav padots protokols/lokācija, tad tiek saglabāts kļūdas paziņojums “protokols/lokācija nav dots/-a”.
    - Ja protokols/lokācija ir padota, tad URL tā tiek saglabāta un tiek veikta atbalsta pārbaude. Ja ir padots neatbalstīts protokols/lokācija, tad tiek saglabāts kļūdas paziņojums “Tika padots neatbalstīts protokols/lokācija”.
  + URL tiek saglabāts metodes ceļš (ja tāds ir padots).
* Ja anotācijas tips ir URL un metodei jau ir definēta URL anotācija, tad saglabā kļūdas paziņojumu “URL jau ir definēts”.
* Ja anotācijas tips nav URL, tad anotācijā tiek saglabāts tips, vērtība (ja tāda ir) un metodē tiek saglabāta pati anotācija.

#### 2.1.5.9 Apstaigāt argumentus

**Nolūks**

Funkcija nepieciešama, lai apstaigātu un pārbaudītu klases laukus.

**Ievade**

Funkcijai ir šādi ievaddati:

* Argumentu konteksts;

**Apstrāde**

Tiek veiktas šādas pārbaudes:

* Vai ir pareiza argumentu un komatu kārtība?
* Vai argumentam ir dots pareizs datu tips?
* Vai argumentam ir dots pareizs vārds?

Programmēšanas valodas gramatika pieļauj to, ka var ievadīt vai nu komatu, vai arī argumentu (datu tips un vārds), tāpēc ir jānoskaidro, kāds elements tiek lasīts un kādam elementam būtu jābūt. Visiem argumentiem ir jābūt atdalītiem ar vienu komatu. Ja metodei tiek definēti argumenti, tad jāsāk un jābeidz ar to, ka ir jābūt argumentam un nevis komatam.

Pirms tiek veikta viena argumenta pārbaude, tai tiek izveidota jauna argumenta instance.

**Izvade**

Izvade ir atkarīga no apstrādes:

* Elementu kārtības pārbaude:
  + Ja pirms argumenta nav komata vai pēdējais argumentu elements ir komats, tad saglabā kļūdas paziņojumu “Trūkst komata”.
  + Ja pirms komata nav argumenta, tad saglabā kļūdas paziņojumu “Tiek sagaidīts arguments”.
  + Ja elements ir arguments, tad tas tiek pārbaudīts.
* Argumenta pārbaude:
  + Ja argumentam nav dots datu tips, tad saglabā kļūdas paziņojumu “Trūkst datu tipa”.
  + Ja argumentam trūkst vārda, tad tiek saglabāts kļūdas paziņojums “Trūkst vārda” un tad rinda, kurā arguments ir definēts, būs rinda, kurā ir definēts tās datu tips.
  + Ja nav dots pareizs datu tips, tad saglabā kļūdas paziņojumu “*’padotais datu tips’* nav derīgs datu tips.”.
  + Ja argumenta vārds sakrīt ar rezervētajiem vārdiem [skat 1.5 nodaļu], tad saglabā kļūdas paziņojumu “Argumentu nevar saukt par *‘rezervētais vārds’*”.
  + Ja argumenta vārds sakrīt ar kāda cita argumenta vārdu, tad saglabā kļūdas paziņojumu “Arguments ar vārdu *‘argumenta vārds’* jau eksistē.” Norādot rindu, kur padotais argumenta vārds jau tiek izmantots.
  + Pēc pārbaudes metodē tiek saglabāts arguments.

#### 2.1.5.10 Apstaigāt kļūdas

**Nolūks**

Funkcija nepieciešama, lai apstaigātu kļūdainos mezglus, kurus izķer ANTLR.

**Ievade**

Funkcijai ir šādi ievaddati:

* Kļūdas mezgls;

**Apstrāde**

* Netiek veiktas nekādas pārbaudes un papildus apstrādes.

**Izvade**

* Tiek saglabāta kļūdas paziņojums “Negaidīts *‘simbols’*”, kur *‘simbols’* ir simbolu virkne, kuru ANTLR uzskata par kļūdainu.

### 2.1.6 Ģeneratora modulis

Pēc koda kompilēšanas, ja nav atrastas kļūdas, tad visi kompilēšanas dati (saraksti ar klasēm, asociācijām un kļūdām) tiek padoti ģenerēšanas modulim, kurš izmanto šos datus, lai ģenerētu visas klases, tās atribūtus, metodes un asociācijas.

* 2.6. att.* **2. līmeņa DPD diagramma ģeneratora modulim**

#### 2.1.6.1 Ģenerēt kodu

**Nolūks**

Funkcija nepieciešama, lai uzsāktu koda ģenerēšanu.

**Ievade**

Funkcijai ir šādi ievaddati:

* Vārdtelpa (namespace).
* Kompilatora dati (klases, asociācijas, kļūdas).

**Apstrāde**

* Netiek veiktas nekādas pārbaudes un papildus apstrādes.

**Izvade**

* Vispirms izsauc funkciju “Ģenerēt klasi *‘BaseObject’*” [skat. 2.1.6.2 nodaļu] un tad katrai klasei izsauc funkciju “Ģenerēt klasi” [skat. 2.1.6.3 nodaļu].

#### 2.1.6.2 Ģenerēt klasi “BaseObject”

**Nolūks**

Funkcija nepieciešama, lai uzģenerētu klasi “BaseObject”, kurā ir klasēm nepieciešamās pamatfunkcijas.

**Ievade**

* Funkcijai nav nepieciešami ievaddati.

**Apstrāde**

* Netiek veiktas nekādas pārbaudes un papildus apstrādes.

**Izvade**

* Izvade ir fails “BaseObject.cs”, kurā ir uzģenerētā klase “BaseObject” [sintaksi skat. 2.1.3.1 nodaļā].

#### 2.1.6.3 Ģenerēt klasi

**Nolūks**

Funkcija nepieciešama, lai ģenerētu klasi.

**Ievade**

Funkcijai ir šādi ievaddati:

* Klases dati

**Apstrāde**

Tiek veiktas šādas pārbaudes:

* Vai klasei ir virsklase?

**Izvade**

Izvade ir atkarīga no apstrādes:

* Ja klasei eksistē virsklase, tad ģenerētajā kodā tā tiek uzģenerēta. Citādi kā virsklasi ģenerē klasi “BaseObject”.
* Sekojošā secībā tiek izsauktas šādas funkcijas:
  + Ģenerēt konstruktoru [skat. 2.1.6.4 nodaļu];
  + Ģenerēt atribūtus [skat. 2.1.6.5 nodaļu];
  + Ģenerēt asociācijas [skat. 2.1.6.6 nodaļu];
  + Ģenerēt metodes [skat. 2.1.6.7 nodaļu];
* Izvade ir fails “*klasesVārds*.cs”, kur *klasesVārds* ir ģenerējamās klases vārds [sintaksi skat. 2.1.3.2 nodaļā].

#### 2.1.6.4 Ģenerēt konstruktoru

**Nolūks**

Funkcija nepieciešama, lai uzģenerētu klasei konstruktoru.

**Ievade**

Funkcijai ir šādi ievaddati:

* Klases vārds

**Apstrāde**

* Netiek veiktas nekādas pārbaudes un papildus apstrādes.

**Izvade**

* Izvades failā tiek ierakstīts konstruktora kods [sintaksi skat. 2.1.3.3 nodaļā].

#### 2.1.6.5 Ģenerēt atribūtus

**Nolūks**

Funkcija nepieciešama, lai uzģenerētu klasei īpasības, ja tādas ir.

**Ievade**

Funkcijai ir šādi ievaddati:

* Atribūtuu dati

**Apstrāde**

Tiek veiktas šādas pārbaudes:

* Vai klasei ir kaut viens atribūts?

**Izvade**

Izvade ir atkarīga no apstrādes:

* Ja klasei nav neviena atribūta, tad nekas netiek apstrādāts un ģenerēts, citādi izvades failā tiek ierakstīts atribūta kods [sintaksi skat. 2.1.3.4 nodaļā].

#### 2.1.6.6 Ģenerēt asociācijas

**Nolūks**

Funkcija nepieciešama, lai uzģenerētu klasei asociācijas galapunktu, ja tādas ir.

**Ievade**

Funkcijai ir šādi ievaddati:

* Asociācijas dati

**Apstrāde**

Tiek veiktas šādas pārbaudes:

* Vai klasei ir kaut viens asociācijas galapunkts?

**Izvade**

Izvade ir atkarīga no apstrādes:

* Ja klasei nav neviena asociācijas galapunkta, tad nekas netiek apstrādāts un ģenerēts, citādi izvades failā tiek ierakstīts asociācijas galapunkta kods [sintaksi skat. 2.1.3.5 nodaļā].

#### 2.1.6.7 Ģenerēt metodes

**Nolūks**

Funkcija nepieciešama, lai uzģenerētu klasei metodes, ja tādas ir.

**Ievade**

Funkcijai ir šādi ievaddati:

* Metodes dati

**Apstrāde**

Tiek veiktas šādas pārbaudes:

* Vai klasē ir kaut viena metode?
* Metodes atgriežamā tipa pārbaude.

**Izvade**

Izvade ir atkarīga no apstrādes:

* Ja klasei nav nevienas metodes, tad nekas netiek apstrādāts un ģenerēts, citādi:
  + Tiek izsaukta funkcija “Ģenerēt argumentus”.
  + Atkarībā no metodes atgriežama tipa, tiek pareizi uzģenerēta atgriežamā vērtība. Ja tips ir void, tad tāda netiek ģenerēta [skat. 2.1.3.6 nodaļā].
  + Izvades failā tiek ierakstīts metodes kods [sintaksi skat. 2.1.3.6 nodaļā]

#### 2.1.6.8 Ģenerēt argumentus

**Nolūks**

Funkcija nepieciešama, lai uzģenerētu klases metodei argumentus, ja tādi ir.

**Ievade**

Funkcijai ir šādi ievaddati:

* Argumentu dati

**Apstrāde**

* Netiek veiktas nekādas pārbaudes un papildus apstrādes.

**Izvade**

* Gadījumā, ja metodei nav neviena argumenta, tad nekas netiek uzģenerēts, citādi tiek uzģenerēti visi metodes argumenti [skat. 2.1.3.6 nodaļā].

### 2.1.7 Ģenerētā koda modulis

Ģenerēšanas rezultātā esam ieguvuši kodu, kuru var izmantot programmu izstrādē WebAppOS videi. Šajā DPD diagrammā redzamo skaitļu 3, 3’, 4 un 4’ identifikācija ir aprakstīta nodaļā 2.1.4, kur 3 un 4 ir daļa no metožu izsaukšanas WebAppOS vidē un 3’ un 4’ ir daļa no atribūtu un asociācijas galapunktu vērtību iegūšanas/uzstādīšanas.

**

*2.7. att.* **2. līmeņa DPD diagramma ģenerētā koda modulim**

#### 2.1.7.1 Pārbaudīt klasi

**Nolūks**

Funkcija nepieciešama, lai pārbaudītu to, vai klase ir saglabāta iekš RAAPI.

**Ievade**

Funkcijai ir šādi ievaddati:

* Klases vārds
* Klases atribūtu dati

**Apstrāde**

Tiek veiktas šādas pārbaudes:

* Vai klase eksistē iekš RAAPI?
* Vai klases atribūti eksistē iekš RAAPI?

**Izvade**

Izvade ir atkarīga no apstrādes:

* Ja klase neeksistē iekš RAAPI tad tā tiek tajā izveidota.
* Ja atribūts neeksistē iekš RAAPI, tad tas tiek tajā izveidots.

#### 2.1.7.2 Pārbaudīt asociācijas

**Nolūks**

Funkcija nepieciešama, lai pārbaudītu to, vai klases asociācijas ir saglabātas iekš RAAPI.

**Ievade**

Funkcijai ir šādi ievaddati:

* Klases vārds (avotklases vārds)
* Klases asociāciju dati

**Apstrāde**

Tiek veiktas šādas pārbaudes:

* Vai asociācijas mērķa klase eksistē iekš RAAPI?
* Vai asociācija eksistē iekš RAAPI?

**Izvade**

Izvade ir atkarīga no apstrādes:

* Ja mērķklase neeksistē iekš RAAPI, tad tā tiek tajā izveidota.
* Ja asociācija neeksistē iekš RAAPI, tad tā tiek tajā izveidots.

#### 2.1.7.3 Iegūt atribūta vērtību

**Nolūks**

Funkcija nepieciešama, lai iegūtu atribūta vērtību no RAAPI.

**Ievade**

Funkcijai nav nepieciešami ievaddati.

**Apstrāde**

Netiek veiktas nekādas apstrādes, izņemot konvertēšana uz pareizo datu tipu, jo iekš RAAPI visas atribūtu vērtības glabājas kā simbolu virknes.

**Izvade**

Izvadē iegūst atribūta vērtību.

#### 2.1.7.4 Uzstādīt atribūta vērtību

**Nolūks**

Funkcija nepieciešama, lai uzstādītu atribūta vērtību.

**Ievade**

Funkcijai ir jāpadod tikai vērtība, kuru vēlas uzstādīt.

**Apstrāde**

Netiek veiktas nekādas apstrādes, izņemot padotās vērtības konvertēšanu uz simbolu virkni, jo iekš RAAPI visas atribūtu vērtības glabājas kā simbolu virknes.

**Izvade**

Izvadē atribūta vērtība tiek saglabāta iekš RAAPI.

#### 2.1.7.5 Iegūt asociācijas vērtību

**Nolūks**

Funkcija nepieciešama, lai iegūtu sarakstu ar asociācijas mērķa klases objektiem.

**Ievade**

Funkcijai nav nepieciešami ievaddati.

**Apstrāde**

No RAAPI iegūstam visus saistītos objektus un no tiem izveidojam jaunu sarakstu, kas sastāv no asociācijas mērķa klases objektiem.

**Izvade**

Izvadē iegūst sarakstu ar mērķklases objektiem.

#### 2.1.7.6 Uzstādīt asociācijas vērtību

**Nolūks**

Funkcija nepieciešama, lai saglabātu asociācijas mērķa klases objektu sarakstu iekš RAAPI.

**Ievade**

Funkcijai ir jāpadod tikai saraksts, kuru vēlas saglabāt.

**Apstrāde**

No ievaddatiem iegūstam visus mērķklases objektus un saglabājam tos iekš RAAPI.

**Izvade**

Izvadē mērķklases objektu saraksts tiek saglabāts iekš RAAPI.

#### 2.1.7.7 Izsaukt metodi

**Nolūks**

Funkcija nepieciešama, lai izsauktu metodi

**Ievade**

Funkcijai ir jāpadod tikai argumenti, kuri metodei ir definēti

**Apstrāde**

Vispirms nepieciešams argumentus saglabāt kā JSON failu, kuru izmanto, lai izsauktu metodi [skat 2.2.10. nodaļu], kuras rezultāta iegūst JSON failu. Tad pārbauda, vai ir radusies kļūda izsaucot šo metodi un ja ir, tad kļūda tiek “mesta”. Citādāk, ja metode nav void metode, tad tiek atgriezta funkcijas vērtība.

**Izvade**

Izvadē tiek atgriezta funkcijas vērtība, ja metode nav void metode.

### 2.1.8 WebMemory modulis

WebMemory ir augsta līmeņa RAAPI. Tās datu struktūras tika aprakstītas nodaļā 2.1.2.2. Šajā nodaļā tiks aprakstītas prasības funkcijām datu struktūru apstrādei. Visas funkcijas tiek veidotas, izmantojot zema līmeņa RAAPI, kurš kvalifikācijas darba ietvaros jau ir dots.

**

*2.8. att.* **2. līmeņa DPD diagramma WebMemory modulim**

#### 2.1.8.1 Izveidot vārdnīcu

**Nolūks**

Funkcijas nepieciešamas, lai izveidotu instanču vārdnīcu. Ir vairākas vārdnīcu izveides funkcijas katra savas vārdnīcas izveidei. Ir šādas instanču vārdnīcas:

* WebObjekta klases (WebKlašu instances)
* WebObjektam saistītie WebObjekti (WebObjektu instances)
* WebKlases (WebKlašu instances)
* WebKlases WebObjekti (WebObjektu instances)
* WebKlases WebAtribūti (WebAtribūtu instances)
* WebKlasē ieejošās WebAsociācijas (WebAsociāciju galapunktu instances)
* No WebKlases izejošās WebAsociācijas (WebAsociāciju galapunktu instances)
* WebKlases virsklases (WebKlašu instances)
* WebKlases apakšklases (WebKlašu instances)

**Ievade**

Funkcijām ir šādi kopīgi ievaddati:

* RAAPI instance
* WebMemory instance

Atkarībā kādu sarakstu jāizveido, tiek padoti šādi ievaddati:

* Objekta atsauce (sarakstiem “Objekta klases” un “Objektam saistītie objekti”).
* Klases atsauce (visiem sarakstiem, izņemot “Objekta klases” , “Objektam saistītie objekti” un “Klases”).

**Apstrāde**

Tiek sagatavota tukša vārdnīca, kur atslēga ir instances reference un vērtība ir instance. No RAAPI tiek iegūts instanču saraksts, kuram iziet cauri, katram elementam iegūst esošu instanci vai izveido jaunu, kuru saglabā vārdnīcā.

**Izvade**

Izvadē tiek atgriezta instanču vārdnīca.

#### 2.1.8.2 Iegūt sarakstu

**Nolūks**

Funkcijas nepieciešamas, lai iegūtu instanču sarakstu. Ir vairākas saraksta iegūšanas funkcijas katra sava saraksta iegūšanai.

2.1.8.1 nodaļā ir uzskaitītas vārdnīcas, kuras tiek veidotas. No šīm vārdnīcām tiek iegūti saraksti.

**Ievade**

Funkcijām nav ievaddatu, izņemot objekta saistīto objektu saraksta iegūšanai, kurai tiek padots:

* Lomas vārds vai asociācijas galapunkts, no kura tiek iegūts lomas vārds

**Apstrāde**

Tiek iegūta nepieciešamā vārdnīca [skat. 2.2.8.1 nodaļu] no kuras tiek izveidots saraksts ar instancēm.

**Izvade**

Izvadē tiek atgriezts instanču saraksts.

#### 2.1.8.3 Izveidot instanci

**Nolūks**

Funkcijas nepieciešamas, lai izveidotu jaunu instanci. Ir vairākas instances izveidošanas funkcijas katra savas klases instances izveidei. Ir iespējams izveidot šādu klašu instances:

* WebKlasei
* WebObjektam
* WebAtribūtam
* WebAsociācijai
* Ģeneralizācijai starp apakšklasi un virsklasi

**Ievade**

Katrai instances izveidošanas funkcijai ir savi ievaddati:

* WebKlases izveidei tiek padots klases vārds
* WebObjekta izveidei nav jāpadod ievaddati
* WebAtribūta izveidei ir jāpadod atribūta vārds un primitīvais tips
* WebAsociācijas izveidei ir jāpadod klases, lomu vārdi un kompozīcijas patiesumvērtība
* Ģeneralizācijas izveidei ir jāpadod virsklases vārds.

**Apstrāde**

Tiek izveidota jauna klases instance, kuru saglabā RAAPI un kura arī tiek atgriezta.

**Izvade**

Izvadē tiek atgriezta instance.

#### 2.1.8.4 Izdzēst instanci

**Nolūks**

Funkcijas nepieciešamas, lai izdzēstu esošu instanci. Ir vairākas instances dzēšanas funkcijas katra savas instances dzēšanai. Ir iespējams izdzēst šādu klašu instances:

* WebKlasi
* WebObjektu
* WebAtribūtu
* WebAsociāciju
* Ģeneralizāciju starp apaksklasi un virsklasi

**Ievade**

Katrai instances izveidošanas funkcijai ir šādi ievaddati:

* Vārds WebKlasei, WebAsociācijai vai WebAtribūtam, atkarība no tā, ko vēlamies dzēst.
* Ģeneralizācijas dzēsanai padodam virsklases vārdu.

**Apstrāde**

RAAPI tiek meklēta instanci ar padoto vārdu un ja tādu atrod, tad tā tiek izdzēsta.

**Izvade**

Netiek veikta datu izvade vai funkcijas vērtību atgriešana.

#### 2.1.8.5 Atrast instanci

**Nolūks**

Funkcijas nepieciešamas, lai atrastu esošu instanci. Ir vairākas instances atrašanas funkcijas katra savas instances atrašanai. Ir iespējams atrast šādu klašu instances:

* WebKlasi
* WebAtribūtu
* WebAsociāciju

**Ievade**

Katrai instances atrašanas funkcijai ir šādi ievaddati:

* Vārds WebKlasei, WebAsociācijai vai WebAtribūtam, atkarība no tā, ko vēlamies atrast.

**Apstrāde**

Vispirms iegūstam sarakstu ar visām instancēm [skat. 2.1.8.1 nodaļu] un iegūstam instances atsauci izmantojot vārdu.

**Izvade**

Izvade ir atkarīga no apstrādes:

* Ja vārdnīcā eksistē elements ar iegūto atsauci, tad atgriežam vārdnicas elementa, kura atslēga sakrīt ar atrasto atsauci, vērtību, citādāk atgriežam null.

#### 2.1.8.6 iegūt apakšklasi

**Nolūks**

Funkcijas nepieciešamas, lai noskaidrotu, vai klase ir apakšklase padotajai klasei:

**Ievade**

Funkcijai tiek padoti šādi ievaddati:

* Virsklases instance

**Apstrāde**

Netiek veikta nekāda apstrāde

**Izvade**

Izvadē tiek atgriezta patiesumvērtība, kas pasaka, vai klase ir apakšklase padotajai klasei.

#### 2.1.8.7 Iegūt atribūta vērtību

**Nolūks**

Funkcijas nepieciešamas, lai uzstādītu objekta atribūtam vertību:

**Ievade**

Funkcijai tiek padoti šādi ievaddati:

* Atribūta vārds

**Apstrāde**

Tiek iegūts saraksts ar objekta klasēm, kurš tiek izskatīts un klasēs tiek meklēts atribūts ar padoto vārdu.

**Izvade**

Izvade ir atkarīga no apstrādes:

* Ja objekta klasēs eksistē atribūts ar padoto vārdu, tad tiek atgriezta šī atribūta vērtība, citādāk atgriež null.

#### 2.1.8.8 Uzstādīt atribūta vērtību

**Nolūks**

Funkcijas nepieciešamas, lai uzstādītu objekta atribūtam vertību:

**Ievade**

Funkcijai tiek padoti šādi ievaddati:

* Atribūta vārds
* Atribūta vērtība

**Apstrāde**

Tiek iegūts saraksts ar objekta klasēm, kurš tiek izskatīts un klasēs tiek meklēts atribūts ar padoto vārdu.

**Izvade**

Izvade ir atkarīga no apstrādes:

* Ja objekta klasēs eksistē atribūts ar padoto vārdu, tad atribūtam tiek piešķirta padotā vērtība.

#### 2.1.8.9 Sasaistīt objektu

**Nolūks**

Funkcijas nepieciešamas, lai sasaistītu divus objektus:

**Ievade**

Funkcijai tiek padoti šādi ievaddati:

* Asociācijas galapunkta lomas vārds vai asociācijas galapunkts, no kura iegūst lomas vārdu
* Objekts, ar kuru vajag sasaistīt

**Apstrāde**

RAAPI tiek izveidota saite starp objektu, kuram izsauc šo funkciju un objektu, kurš tika padots

**Izvade**

Netiek veikta izvade vai vērtību atgriešana

#### 2.1.8.10 Sasaistīt objektus

**Nolūks**

Funkcijas nepieciešamas, lai atjaunotu saistīto objektu sarakstu. Tiek izmantots asociāciju sarakstu vērtības uzstādīšanā [skat. 2.1.7.6 nodaļu].

**Ievade**

Funkcijai tiek padoti šādi ievaddati:

* Asociācijas galapunkts
* Saraksts ar saistītajiem objektiem

**Apstrāde**

Vispirms iegūst pašlaik esošo saistīto objektu sarakstu, kuram iziet cauri un izdzēš visas saites. Tad katram elementam no padotā saraksta izsauc funkciju “Sasaistīt objektu” [skat. 2.1.8.9 nodaļu].

**Izvade**

Netiek veikta izvade vai vērtību atgriešana

### 2.1.9 LocalWebCalls modulis

Local Web Calls modulis ir atbildīgs par funkciju dinamisko izsaukšanu WebAppOS vidē. Šajā modulī tiek izsauktas tikai C# valodā rakstītas funkcijas.

Šajā DPD diagrammā redzamo skaitļu 1, 2, 5 un 6 identifikācija ir aprakstīta nodaļā 2.1.4, kur visi skaitļi ir daļa no metožu izsaukšanas WebAppOS vidē.

**

*2.9. att.* **2. līmeņa DPD diagramma LocalWebCalls modulim**

#### 2.1.9.1 Izsaukt funkciju

**Nolūks**

Funkcija nepieciešama, lai izsauktu C# valodā rakstītu funkciju, kas atrodas lokālā failu sistēmā.

**Ievade**

Funkcijai tiek padoti šādi ievaddati:

* RAAPI
* Atsauce uz objektu
* Metodes URL
* Argumenti (JSON formātā)
* RemoteWebCalls instance [skat. 2.1.10 nodaļu].

**Apstrāde**

Vispirms iegūstam sarakstu ar metodes URL elementiem [skat. 2.1.9.2 nodaļu]. Tad izmantojot šo sarakstu iegūstam klases objektu un to tā iegūstam metodi. Tad tiek veikta pārbaude, vai metode eksistē.

**Izvade**

Izvade ir atkarīga no apstrādes:

* Ja metode, kuru meklēja, neeksistē tad atgriežam JSON simbolu virkni, kas satur elementu *error* ar vērtību “method not found”.
* Ja metode, kuru meklēja, eksistē, tad šī metode tiek izsaukta un tās rezultāts tiek atgriezts.

#### 2.1.9.2 Iegūt metodes datus

**Nolūks**

Funkcija nepieciešama, lai no metodes URL, kas ir viena vesela simbolu virkne, iegūtu atdalītus datus, kas iekļauj protokolu, lokāciju, klases ceļu un metodes vārdu.

**Ievade**

Funkcijai tiek padoti šādi ievaddati:

* Metodes URL

**Apstrāde**

Tā kā ir zināms, ka būs .NET kods, kurš ir lokālajā failu sistēmā, tad metodes URL būs formātā “dotnet:local:namespace.classname#methodname”. Ir jāiegūst saraksts, kas satur šādus elementus: ”dotnet” , “local” , “namespace.classname” un “methodname”, tāpēc URL ir jāsadala šādās daļās un sarakstu jāatgriež.

**Izvade**

Izvadē tiek atgriezts saraksts iepriekš definētajā formātā.

### 2.1.10 RemoteWebCalls modulis

Remote Web Calls modulis ir atbildīgs par funkciju izsaukšanu, izmantojot funkciju “*enqueueInnerWebCall*”, kura kvalifikācijas darba ietvaros jau ir dota. Šajā modulī tiek izsauktas funkcijas, kuras rakstītas dažādās programmēšanas valodās ar dažādām koda lokācijām. Remote Web Calls ir tikai viena funkcija, “Izsaukt funkciju”, kuru izmanto ģenerētais kods.



*2.10. att.* **2. līmeņa DPD diagramma RemoteWebCalls modulim**

#### 2.1.9.1 Izsaukt funkciju

**Nolūks**

Funkcija nepieciešama, lai izsauktu funkciju, izmantojot “*enqueueInnerWebCall*”.

**Ievade**

Funkcijai tiek padoti šādi ievaddati:

* RAAPI
* Atsauce uz objektu
* Metodes vārds
* Argumenti (JSON formātā)

**Apstrāde**

Vienīgais, kas tiek veikts, ir funkcijas “*enqueueInnerWebCall*” izsaukšana, kurai padod atsauci uz objektu, metodes vārdu un argumentus kā JSON simbolu virkni, kuras rezultātā iegūst JSON simbolu virkni.

**Izvade**

Izvade rezultātā tiek atgriezts funkcijas “*enqueueInnerWebCall*” rezultāts.

## 2.2 Nefunkcionālās prasības

* Procesors 2 GHz un 1000 rindiņu jaunkoda failā ir jāspēj kodu kompilēt un ģenerēt 1 sekundē.
* Ja jaunkoda fails nepārsniedz 1000 rindiņu, tad starpkoda faili kopā nepārsniedz 1MB.
* Gramatikas kompilatoram tiek padots fails UTF-8 kodējumā un ģeneratoram faili ir jāveido UTF-8 kodējumā.

# 3. PROGRAMMATŪRAS PROJEKTĒJUMA APRAKSTS

## 3.1 WAOS C# gramatikas projektējums

### 3.1.1 Koda pamatprojektējums

WAOS C# valodā programmkods tiek sadalīts pa vairākiem blokiem. Šajā gadījumā ir divu veidu bloki: klases un asociācijas. Katram blokam ir divas sastāvdaļas:

* Tips, kas apraksta, kāda veida bloks tas ir (Klasēm tiek izmantots atslēgvārds *‘class’* un asociācijām tiek izmantots atslēgvārds *‘association’*).
* Ķermenis, kas apraksta pašu bloku. [skat. 3.1.2 un 3.1.3 nodaļas].

Kodā var būt patvaļīgs skaits bloku, pat var nebūt neviena bloka. Katram blokam ir noteikti jānorāda vai nu tips, vai ķermenis (vismaz viena no sastāvdaļām var trūkt, bet noteikti vienai ir jābūt un ja kādas no sastāvdaļām trūkst, tad kompilatoram to ir jāuztver kā kļūdu).



*3.1. att.* **gramatikas koks**

### 3.1.2 Asociācijas projektējums

Asociācijas ķermenis tiek rakstīts apaļajās iekavās. Tajās tiek norādīta informācija par asociācijas avotu un mērķi, kas ir atdalītas ar bultām. Avots un mērķis sastāv no lomas vārda un klases vārda, kas ir atdalīti ar kolu.

Lomas vārdi, klases vārdi, koli un bultas var netikt ierakstītas, bet kompilatoram to ir jāuztver kā kļūdu.



*3.2. att.* **asociācijas koks**

### 3.1.3 Klases projektējums

Klasei ir noteikti vajadzīgi dati par klases vārdu, tās virsklasi, vai dati par tās sastāvu, kas ir ierakstīts figūriekavās. Vieni no datiem var nebūt, bet kompilatoram to ir jāuztver ka kļūdu.

Ja tiek rakstīti dati par virsklasi, tad noteikti ir jābūt kolam pēc klases vārda.

Klases ķermenis (šajā gadījumā daļa, kas ir rakstīta figūriekavās) sastāv no laukiem (atribūti un metodes), kas ir atdalīti ar semikolu, un katram laukam var definēt vai nu vairākas anotācijas (domātas metodēm), vai sniegt informāciju par pašu lauku (datu tips, vārds, aizsardzība un metodēm argumenti). Noteikti katram laukam tiek definētas anotācijas vai pati definīcija, bet kompilatoram to ir jāuztver kā kļūdu.



*3.3. att.* **klases koks**

### 3.1.4 Anotācijas projektējums

Anotācijas tiek definētas kvadrātiekavās. Tajās tiek ierakstīts anotācijas tips un tam blakus ir apaļas iekavas, kurās pēdiņās tiek rakstīta vērtība. Vērtība var būt jebkāda simbolu virkne, kas nesastāv no dubultajām pēdiņām un iekavām. Šīs simbolu virknes var tik atdalītas ar sekojošiem atdalītājiem: kolu, semikolu, punktu, komatu vai restīti. Ja anotācijas vērtība pirmie divi atdalītāji ir koli, tad teksts pirms otrā kola ar otro kolu ieskaitot tiek uzskatīts par URL anotācijas daļu, kura ir ierakstīta programmēšanas valoda un lokācija.

Drīkst nebūt jebkura no anotācijas sastāvdaļām, kas ir kvadrātiekavās, bet kompilatoram to ir jāuztver kā kļūdu.



*3.4. att.* **anotācijas koks**

### 3.1.5 Lauka definīcijas projektējums

Lauka definīcijā noteikti ir vajadzīgs definēt datu tipu, aizsardzību, lauka vārdu vai argumentus (metodēm). Var nebūt kāds no šiem datiem, bet kompilatoram to ir jāuztver kā kļūdu. Argumentiem ir jābūt definētiem apaļajās iekavās, kurās tiek rakstīts vai nu arguments (datu tips, vārds) vai tiek likts komats.

*3.5. att.* **lauka definīcijas koks**

### 3.1.6 Gramatikas elementu vērtību izklāsts

|  |  |  |
| --- | --- | --- |
| Gramatikas elements | Koks (attēls), kurā atrodas | Vērtību klase/-s |
| Block Type | 3.1 | BLOCKTYPE, PROTECTION, DATATYPE |
| Association Source Name | 3.2 | IDENTIFIER, PROTECTION, DATATYPE, BOCKTYPE |
| Association Source Class | 3.2 | IDENTIFIER, PROTECTION, DATATYPE, BOCKTYPE |
| Association Target Name | 3.2 | IDENTIFIER, PROTECTION, DATATYPE, BOCKTYPE |
| Association Target Class | 3.2 | IDENTIFIER, PROTECTION, DATATYPE, BOCKTYPE |
| Class Name | 3.3 | IDENTIFIER |
| Super Class Name | 3.3 | IDENTIFIER |
| Protocol | 3.4 | IDENTIFIER, PROTECTION, DATATYPE, BOCKTYPE |
| Location | 3.4 | IDENTIFIER, PROTECTION, DATATYPE, BOCKTYPE |
| Annotation Data | 3.4 | ANYTHING, IDENTIFIER, PROTECTION, DATATYPE, BOCKTYPE, |
| Field Protection | 3.5 | PROTECTION |
| Field Data Type | 3.5 | DATATYPE, BLOCKTYPE |
| Field Name | 3.5 | IDENTIFIER |
| Argument Data Type | 3.5 | DATATYPE, BLOCKTYPE, PROTECTION |
| Argument Name | 3.5 | IDENTIFIER |

*3.1. tabula.* **Gramatikas elementu vērtību klases**

|  |  |
| --- | --- |
| Klase | Vērtības |
| BLOCKTYPE | *class*, *association* |
| PROTECTION | *public, private* |
| DATATYPE | *Integer, String, Boolean, Real, Void* |
| IDENTIFIER | identifikators |
| ANYTHING | Simbolu virkne, kas nesastāv no tukšumiem, dubultajām pēdiņām, iekavām, kola, semikola, punkta, komata, restītes, svītriņas un lielāks mazāks zīmēm. |

*3.2. tabula.* **Vērtību klašu vērtības**

## 3.2 Kompilatora projektējums

### 3.2.1 Pamatlietu projektējumi

#### 3.2.1.1 Svarīgo ciklu projektējums

Kompilatorā ļoti bieži veiks sarakstu pārskatu, lai atrastu esošos elementus. Ir iespējami divi gadījumi: elementa pastāvēšana ir kļūdaina vai elementa pastāvēšana ir pareiza. Ja pastāvēšana ir kļūdaina, tad kļūdu saglabā un funkciju beidz. Ja pastāvēšana ir pareiza, tad saglabā datus, beidz cikla darbību, bet funkciju turpina. Kļūdaino pastāvēšanu apzīmēsim ar zilu taisnstūri, bet pareizo pastāvēšanu – ar violetu taisnstūri.



*3.6. att.* **ciklu secību diagramma. Kļūdainā pastāvēšana (zils) un pareizā pastāvēšana (violets)**

#### 3.2.1.2 Datu struktūru projektējums

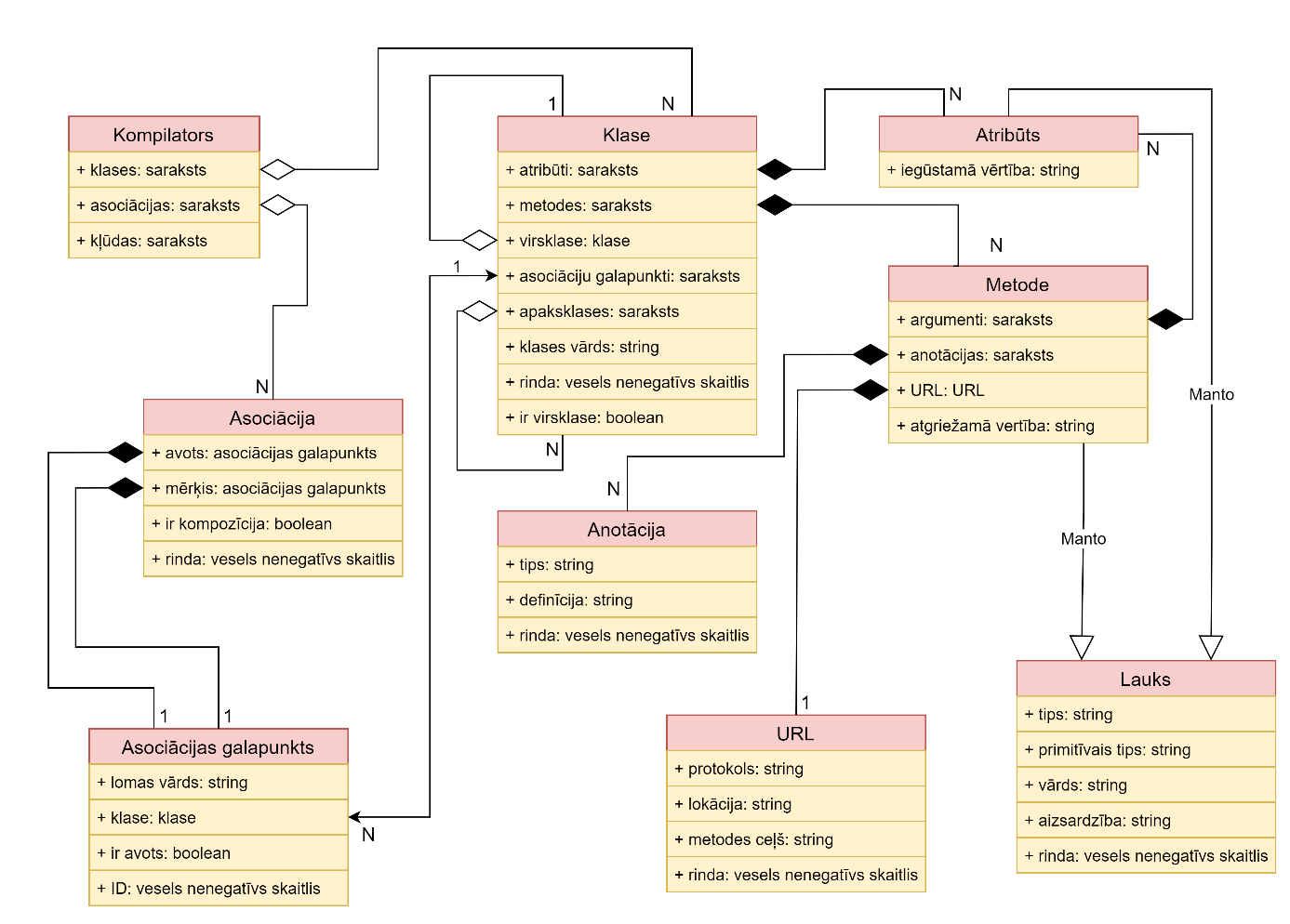
Nodaļā 2.2.2.1 tika aprakstītas datu struktūras, kuras tiek izmantotas kompilatorā. Šajā nodaļā ir izveidota datu struktūru diagramma ar visiem atribūtiem un ir aprakstītas lietas, kuras nav iekļautas iepriekš minētajā nodaļā.

Klasē “Lauks” datu tipu glabā divās īpašībās: *tips* un *primitīvais tips*. Ar *tips* tiek saprasts lauka datu tips C# programmēšanas valodā, bet *primitīvais tips* – datu tips, kādu glabā RAAPI.

Metodēs atribūtu *atgriežamā vērtība* izmantos ģenerators metodes ģenerēšanā. Līdzīgi tiks izmantots Klases ”Atribūts” atribūts *iegūstamā vērtība* atribūtu’get’ funkciju ģenerēšanā.

Ģenerētajās klasēs asociācijas galapunkti tiek ģenerēti kā saraksti, kur sarakstā glabā objektus, kuras tips ir asociācijas galapunkta klase un kuras nosaukums ir asociācijas lomas vārds.

Katrā datu struktūrā glabā rindu kā veselu nenegatīvu skaitli, kuru kompilators izmanto funkcionālo kļūdu meklēšanā, norādot, kur atkārtojas jau izmantoti elementi.

**

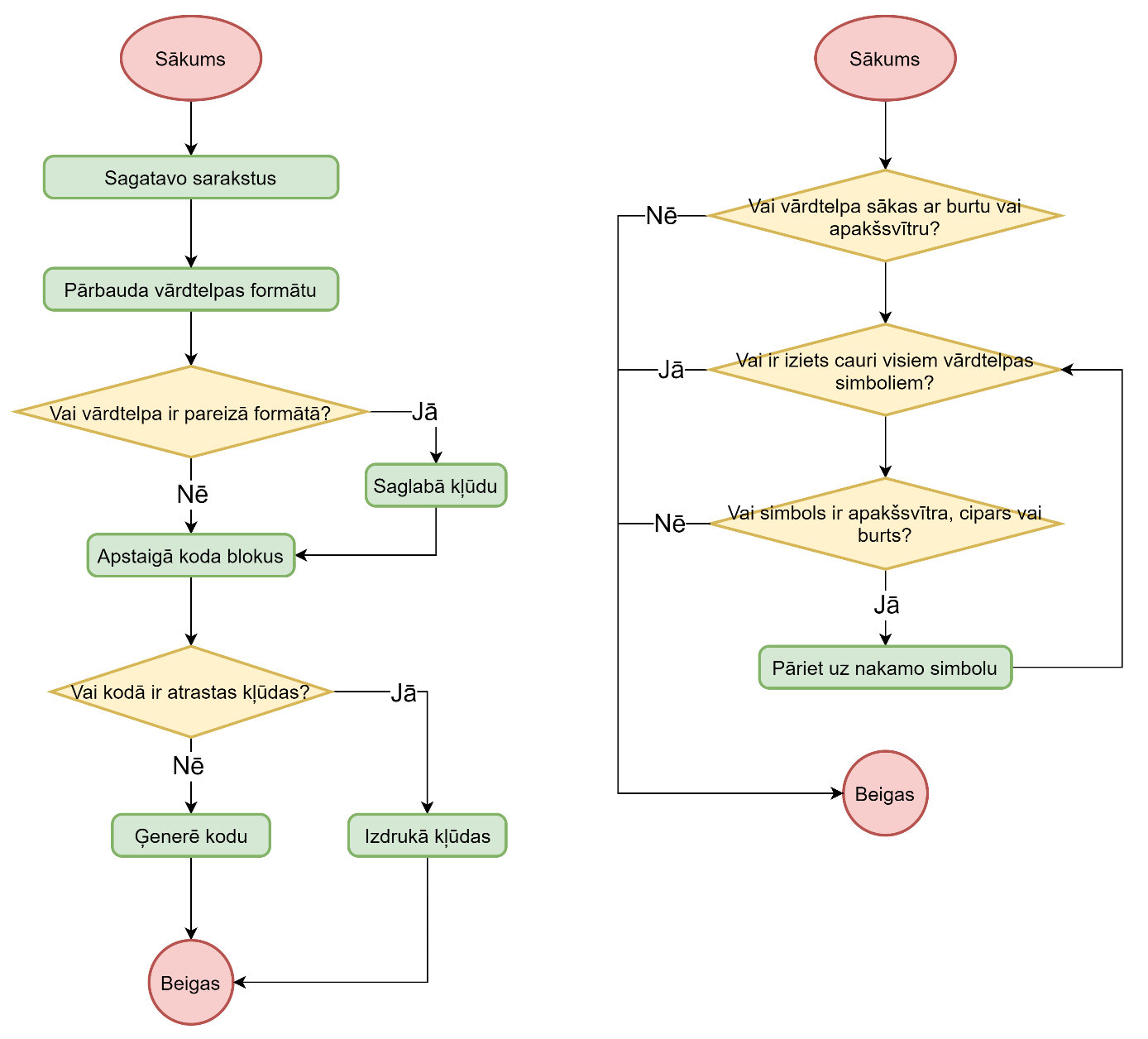
*3.7. att.* **Kompilatora datu struktūru klašu diagramma**

#### 3.2.1.3 Pamatkompilēšanas projektējums

Kompilators kopumā veic trīs funkcijas:

* Pārbauda padotās vārdtelpas formātu [skat 3.8 att. labo shēmu] (vārdtelpa ir pareizā formātā, ja tas ir identifikators)
* Kompilē jaunkodu
* Pārbauda, vai ir kļūdas

Ja pirmo divu funkciju rezultātā nav radušas kļūdas, tad tiek veikta koda ģenerēšana [skat. 3.3 nodaļu]. Citādi tiek izdrukātas visas kļūdas.



*3.8. att.* **Pamatkompilēšanas secību diagramma (pamatfunkcija pa kreisi, vārdtelpas pārbaude pa labi)**

#### 3.2.1.4 Koda bloku kompilēšanas projektējums

Koda bloki sastāv no tipa un ķermeņa [skat. 3.1.1 nodaļu]. Tipam ir jābūt definētam un tam ir jābūt definētam pareizi (ar atslēgas vārdiem *‘class’* vai *‘association’*). Tas pats attiecas arī uz bloka ķermeni. Papildus tam bloka tipam un ķermenim ir jābūt pareizā kombinācijā. Ja izmanto atslēgvārdu *‘class’*,tad ir jābūt definētai klasei un ja izmanto atslēgas vārdu ‘*association*’, tad jābūt definētai asociācijai. Pēc veiktajām pārbaudēm tiek kompilēts bloka ķermenis, ja tāds ir dots, citādi pārejam uz nākamo bloku.



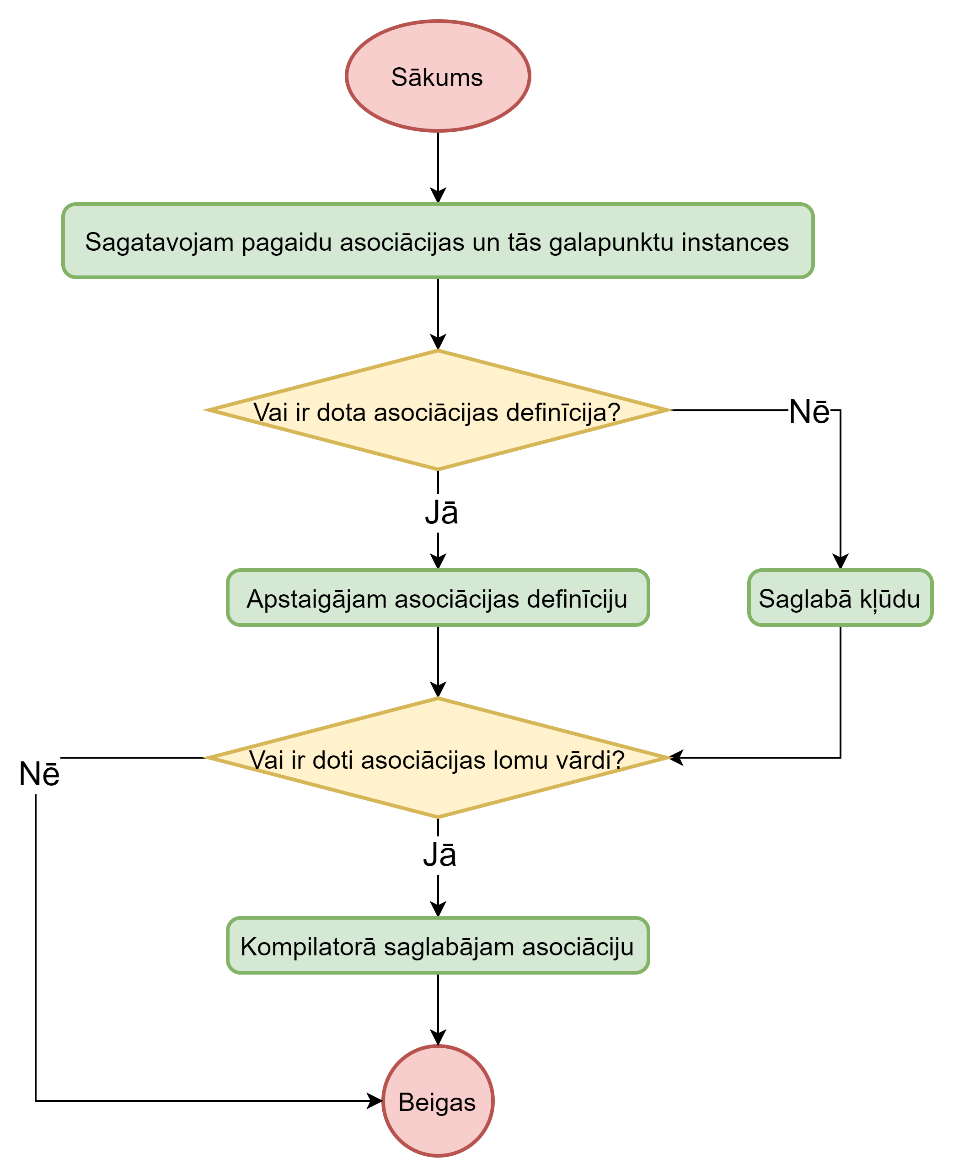
*3.9. att.* **Koda bloku kompilēšanas secību diagramma**

### 3.2.2 Asociācijas kompilēšanas projektējumi

#### 3.2.2.1 Asociācijas pamatkompilēšanas projektējums

Sākumā ir nepieciešams sagatavot instances jaunām asociācijām un to galapunktiem, kuri tiks glabāti kompilatorā. Tad kompilējam asociācijas definīciju [skat. 3.2.2.2 nodaļu], ja tāda ir dota. Ja asociācijā ir dots vismaz viens lomas vārds, tad visi dati tiek saglabāti.

Saglabāšanas procesā asociācijas galapunktiem norādām, kurai asociācijai tie pieder, kā arī to, kurš galapunkts ir avots un kurš nav. Šīs darbības veic, ja galapunktam ir dots lomas vārds. Beigās klasēs un asociācijā saglabājam galapunktus, bet kompilatorā pašu asociāciju.



*3.10. att.* **Asociācijas pamatkompilēšanas secību diagramma**

#### 3.2.2.2 Asociācijas definīcijas kompilēšanas projektējums

Definīcijas kompilēšanā tiek pārbaudītas trīs lietas:

* Vai asociācijai ir definēts avots? Ja tas ir definēts, tad vai lomas vārdu un klasi atdala kols? [skat. 3.1.1 att. galapunkta pārbaudes shēmu]
* Vai ir definēta kompozīcijas esamība?
* Vai asociācijai ir definēts mērķis? Ja tas ir definēts, tad vai lomas vārdu un klasi atdala kols? [skat. 3.1.1 att. galapunkta pārbaudes shēmu]

Kad šīs pārbaudes ir veiktas, tad tiek kompilētas asociācijas klases un, ja ir definēta gan avota, gan mērķa klase, tad arī tiek kompilēti lomu vārdi.

**

*3.11. att.* **Asociācijas definīcijas kompilēšanas secību diagramma (Pamatfunkcija pa kreisi un galapunkta pārbaudes funkcija pa labi)**

#### 3.2.2.3 Asociācijas klases kompilēšanas projektējums

Gadījumā, ja ir definēta klase, tad tā tiek kompilēta. Vispirms jāpārliecinās, ka klases vārds nav kāds no rezervētajiem vārdiem [skat. 1.5. nodaļu]. Ja sakrīt ar kādu no rezervētajiem vārdiem, tad funkcija beidzas ar kļūdas saglabāšanu.

Tad tiek pārbaudītas klases. Ja tiek atrasta padotā klase, tad tās datus saglabā. Datu saglabāšana ir atkarīga no tā, vai pārbauda avotklasi, vai mērķklasi:

* Pārbauda avotklasi – atrasto klasi saglabā asociācijā kā avotklasi un to saglabā arī mērķa galapunktā.
* Pārbauda mērķklasi – atrasto klasi saglabā asociācijā kā mērķklasi un to saglabā arī avota galapunktā.

**

*3.12. att.* **Asociācijas klases kompilēšanas secību diagramma**

#### 3.2.2.4 Asociācijas lomas vārdu kompilēšanas projektējums

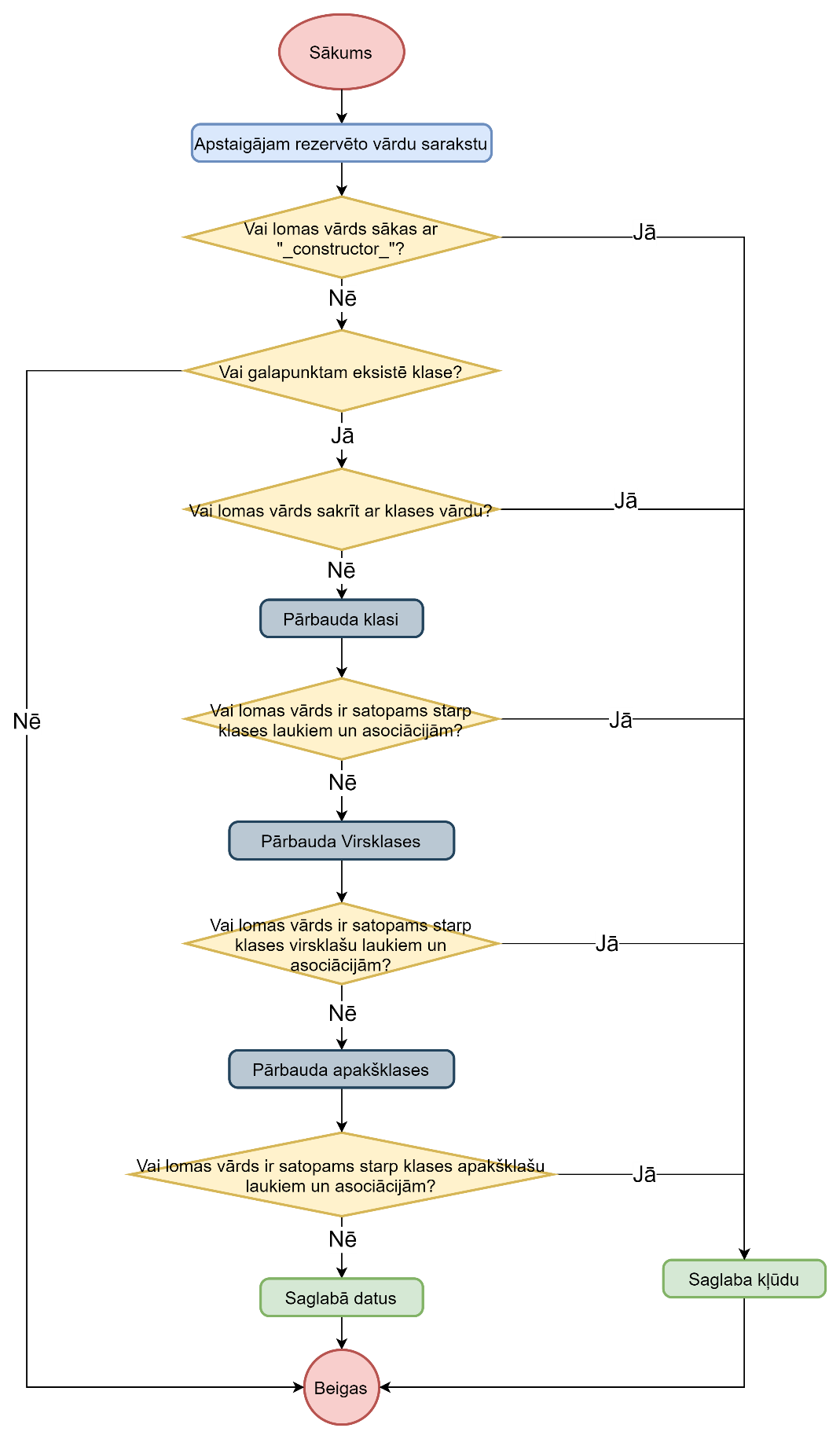
Vispirms jāpārliecinās, ka lomas vārds nesakrīt ar rezervētajiem vārdiem [skat. 1.5 nodaļu] un nesākas ar “\_constructor\_”. Tad jāpārliecinās, ka ir definēta:

* Avota klase, ja tiek skatīts mērķa lomas vārds
* Mērķa klase, ja tiek skatīts avota lomas vārds

Ja nav definēta klase, tad lomas vārda kompilēšana beidzas. Tālāk nedrīkst pieļaut, ka

* Avota lomas vārds sakrīt ar mērķa klases vārdu
* Mērķa lomas vārds sakrīt ar avota klases vārdu

Visbeidzot pārbauda visus klases laukus un asociācijas galapunktus, un pārliecinās, ka kādam no tiem vārds nesakrīt ar padoto lomas vārdu. Tādu pašu lauku pārbaudi veic virsklasei, ja tāda ir definēta un arī apaksklasēm, ja tādas ir.

**

*3.13. att.* **Asociācijas lomas vārdu kompilēšanas pamatfunkcijas secību diagramma**

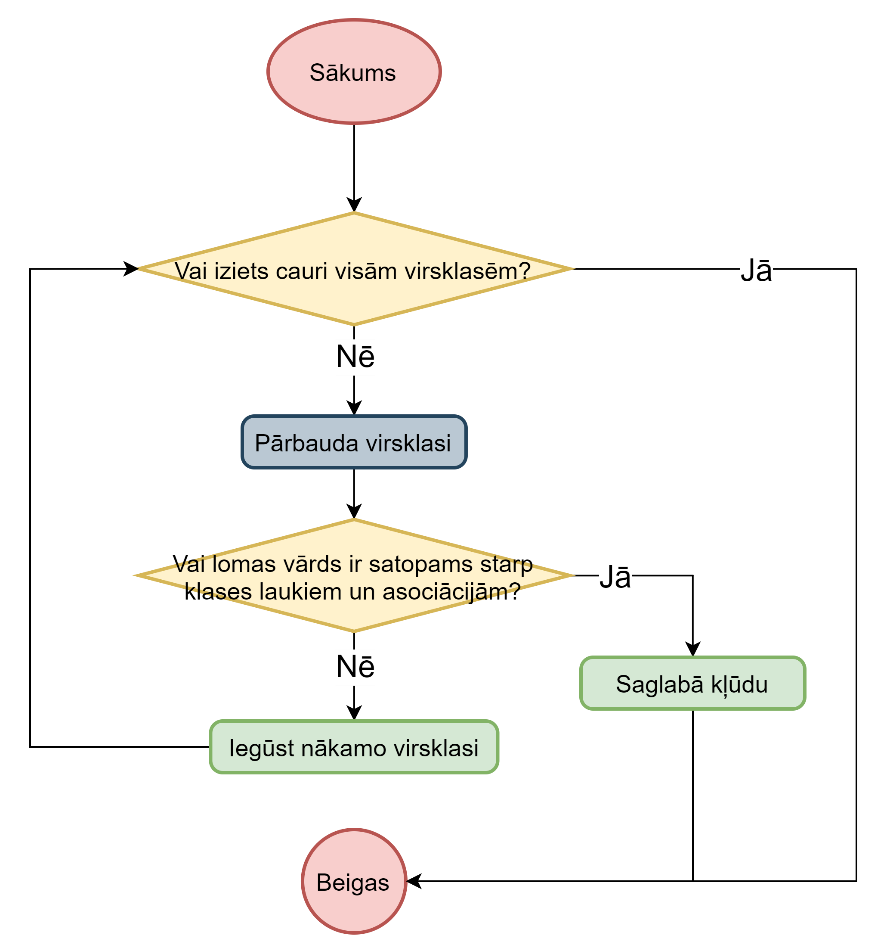
Pamatfunkcijas shēma ir redzama attēlā 3.13.

Klases pārbaudē tikai tiek izskatīti klases lauki un asociācijas [skat. 3.14 att.]. Virsklases pārbaudē vispirms pārliecinās, ka virsklase eksistē [skat. 3.15 att.] un tikai tad veic tās pārbaudi. Pārbaude ir identiska bāzes klases pārbaudei, tikai vēl ir jāņem vērā, ka ir jāskatās virsklašu publiskie lauki.

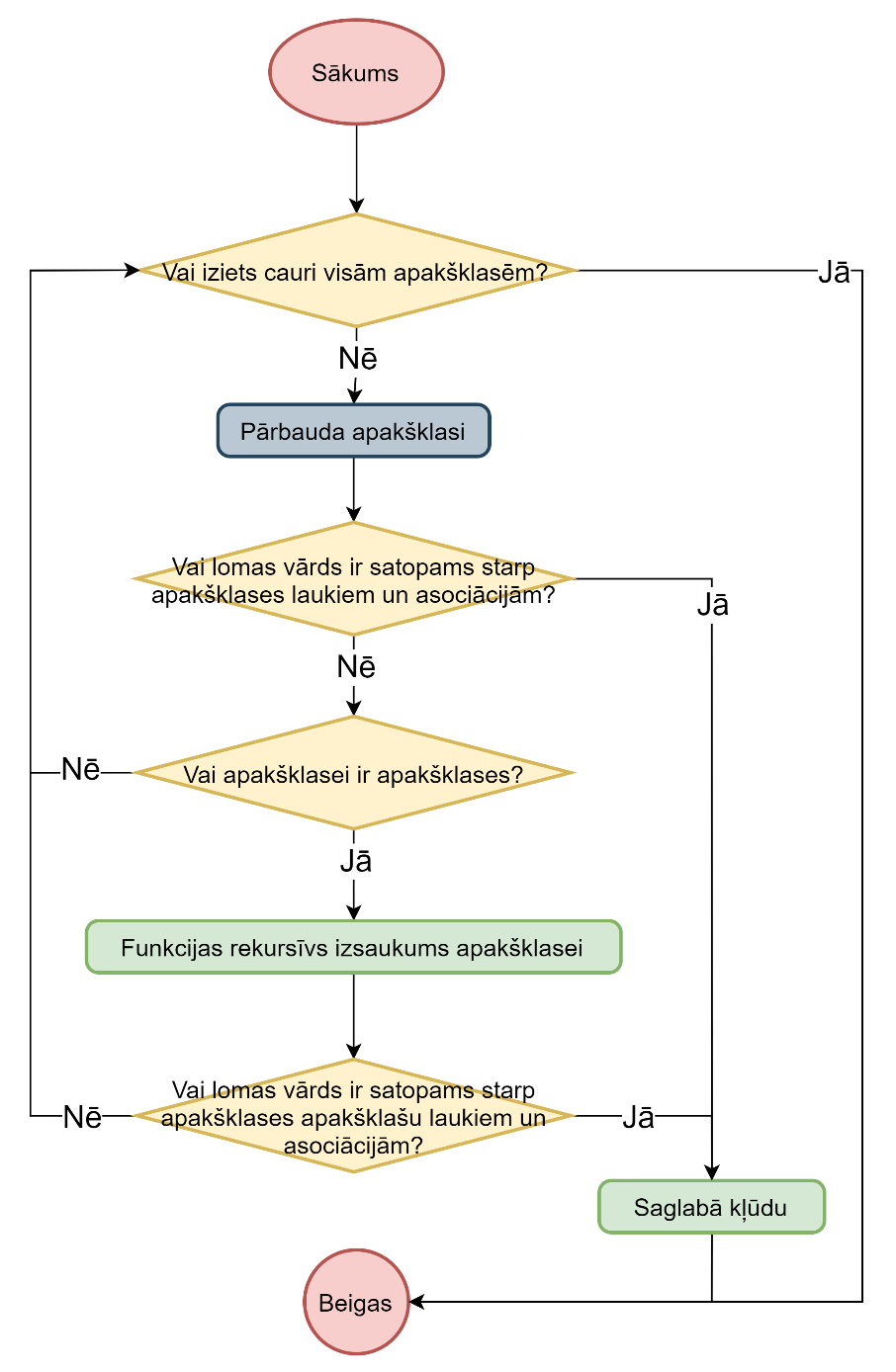
Apakšklašu pārbaudē ir līdzīga koka meklēšanai dziļumā. Tiek izskatīta viena apakšklase un tad skatās, vai šai apakšklasei ir savas apakšklases un ja tādas ir, tad tām iziet cauri. Funkcija ir rekursīva. [skat. 3.16 att.]. Pašas apakšklases pārbaude ir identiska bāzes klases pārbaudei.

Attēls, kurā ir teksts

Apraksts ģenerēts automātiski *3.14. att.* **Asociācijas lomas vārda klases pārbaudes secību diagramma**



*3.15. att.* **Asociācijas lomas vārda virsklases pārbaudes secību diagramma**



*3.16. att.* **Asociācijas lomas vārda apakšklases pārbaudes secību diagramma**

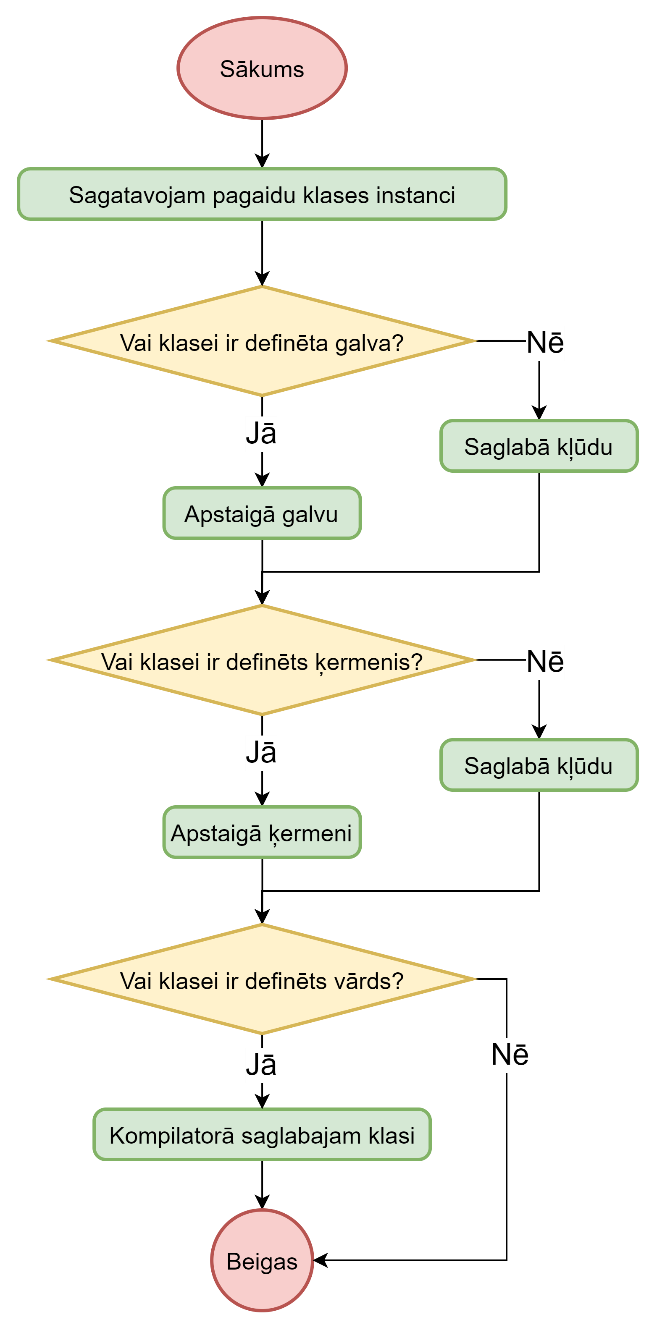
### 3.2.3 Klases kompilēšanas projektējumi

#### 3.2.3.1 Klases pamatkompilēšanas projektējums

Vispirms nepieciešams sagatavot instanci jaunai klasei, kuru glabāsim kompilatorā. Tad tiek veiktas divas pārbaudes:

* Vai klasei ir definēta galva?
* Vai klasei ir definēts ķermenis?

Gadījumā, ja kāda no šīm divām sastāvdaļām ir definēta, tā tiek kompilēta. Pārbaužu un kompilēšanu beigās klasi saglabā kompilatorā, ja tai ir dots vārds.

**

*3.17. att.* **klases pamatkompilēšanas secību diagramma**

#### 3.2.3.2 Klases galvas kompilēšanas projektējums

Galvenais, ko jāpārbauda ir tas, vai klasei ir definēts klases vārds. Ja tāda nav, tad tiek saglabāta kļūda, citādi to kompilē. Ja klases galvā ir definēta virsklase (pēc klases vārda ir kols), tad pārbauda, vai ir dots virsklases vārds. Ja tas nav dots, tad saglabā kļūdu, citādi virsklases vārdu kompilē [skat. 3.2.3.4 nodaļu].

**

*3.18. att.* **klases galvas kompilēšanas secību diagramma. Pamatfunkcija (pa kreisi) un virsklases vārda pārbaudes funkcija (pa labi)**

#### 3.2.3.3 Klases vārda kompilēšanas projektējums

Galvenais, ko jāpārbauda ir tas, vai jau eksistē klase ar padoto vārdu. Sāksim ar to, ka pārbauda padoto klases vārdu starp rezervētajiem vārdiem [skat. 1.5. nodaļu] un tad to pārbauda starp jau apskatīto klašu vārdiem. Ja kaut viens no pārskatītajiem vārdiem sakrīt ar padoto klases vārdu, tad tiek saglabāta kļūda un funkcija beidzas.

**

*3.19. att.* **klases vārda kompilēšanas secību diagramma**

#### 3.2.3.4 Virsklases vārda kompilēšanas projektējums

Virsklases vārda kompilēsana sakrīt ar klases vārda kompilēsanu, bet ir divas atšķirības.

* Tiek pārbaudīts, vai virsklases vārds sakrīt ar bāzes klases vārdu, ko nedrīkst pieļaut.
* Apstaigājot klašu sarakstu, ir jābūt tā, ka klase eksistē, pretēji klases vārda pārbaudei, kur to nedrīkst pieļaut.

**

*3.20. att.* **virsklases vārda kompilēšanas secību diagramma**

### 3.2.3 Klases lauku kompilēšanas projektējumi

#### 3.2.3.1 Lauka sākumkompilēšanas projektējums

Lauka kompilēšanas pamata ir divas funkcijas

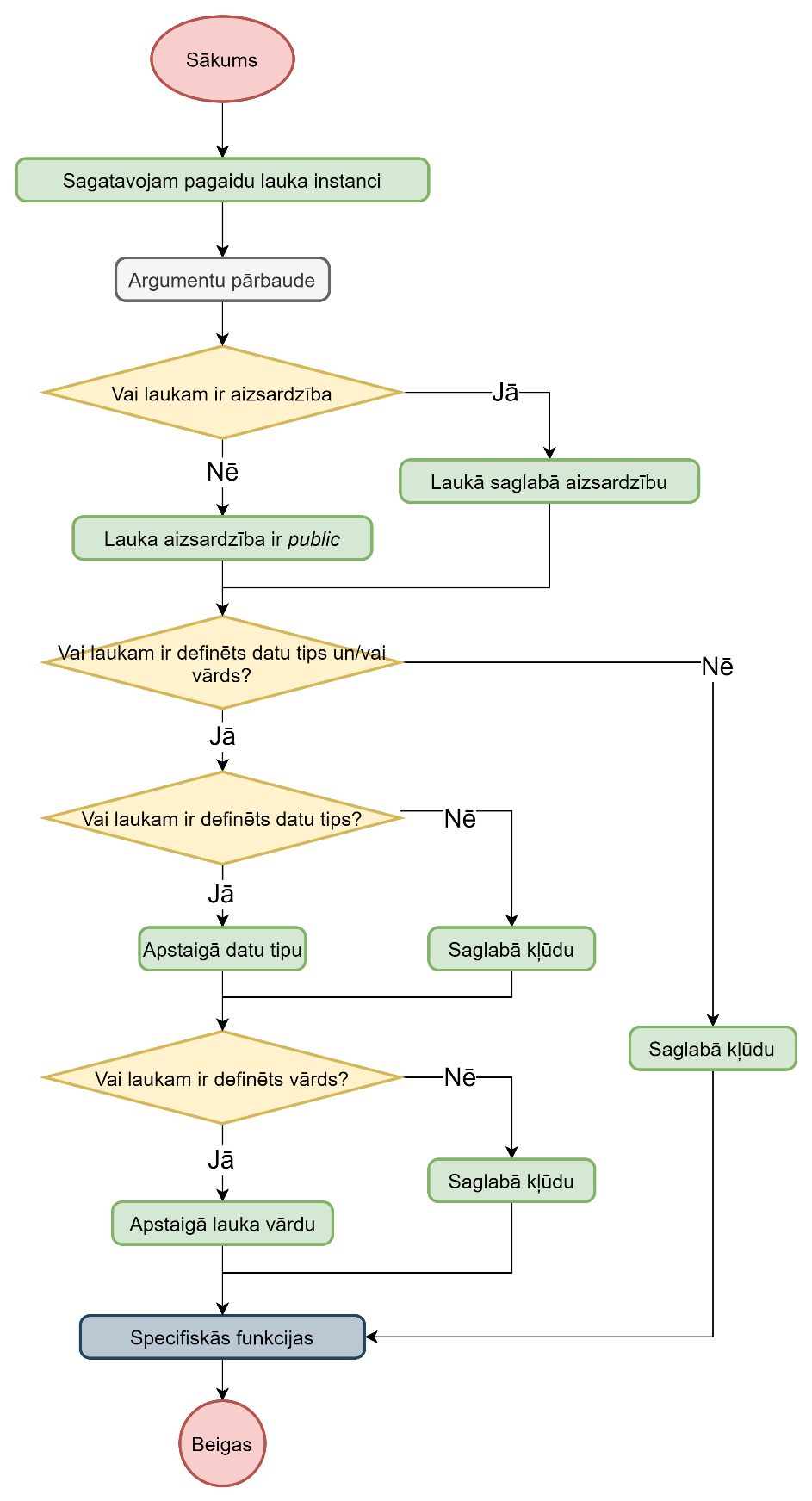
* Noteikt, vai lauks vispār ir definēts. [skat. 3.21 att. kreisā shēma]
* Noteikt, kāds lauks ir definēts. [skat. 3.21 att. labā shēma]

Pirmajā funkcijas daļā vēl ir jāpārliecinās, vai lauks beidzas ar semikolu. Otrajā funkcijas daļā tiek noskaidrots, vai lauks ir metode vai atribūts. Ja laukam ir definēta kaut viena anotācija, vai ir definēti argumenti, tad lauks tiek uzskatīts par metodi, citādāk tas ir atribūts.

*3.21. att.* **lauka sākumkompilēšanas secību diagrammas.**

#### 3.2.3.2 Lauka pamatkompilēšanas projektējums

Šī sadaļa apraksta atribūta un metodes kompilēšanas kopīgos aspektus, kas iekļauj lauka aizsardzības, datu tipa un vārda esamības pārbaudi. 3.22. att. tumši pelēkajā taisnstūri ir ierakstīts “Specifiskās funkcijas”. Šajā daļa funkcijas ir atkarīgas no tā, vai ir definēts atribūts vai metode. Ja tas ir atribūts, tad vienīgais, ko atliek darīt, ir saglabāt klasē atribūta objektu, ja tam ir dots vārds. Par to, kas notiek gadījuma, ja tiek kompilēta metode, lasiet nodaļu 3.2.4.1. Gaiši pelēkais taisnstūris nozīmē, ka ir jāveic argumentu kompilēšana, ja tiek definēta metode. Šīs funkcijas shēma atrodama attēlā 3.26, kuras atrodas nodaļā 3.2.4.1.

**

*3.22. att.* **lauka pamatkompilēšanas secību diagrammas.**

#### 3.2.3.3 Lauku un argumentu datu tipu kompilēšanas projektējums

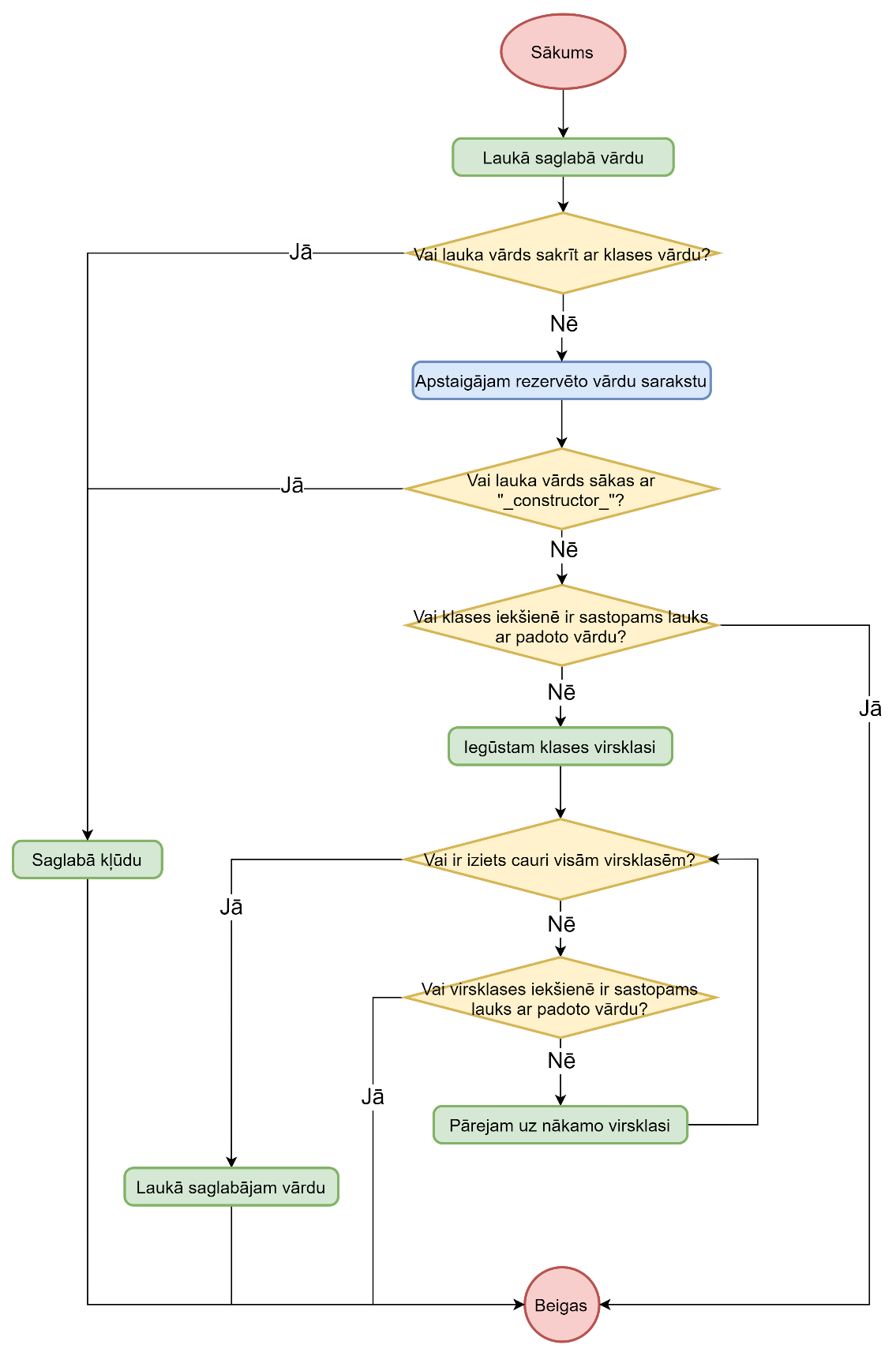
Šajā funkcijā ir tikai jāpārbauda, vai datu tips ir pareizs. Atribūta un argumenta gadījumā pieņemams datu tips ir ‘Integer’, ‘String’, ‘Boolean’ un ‘Real’, bet metodei var arī būt ‘Void’.



*3.23. att.* **lauku un argumentu datu tipu kompilēšanas secību diagrammas.**

#### 3.2.3.4 Lauka vārda kompilēšanas projektējums

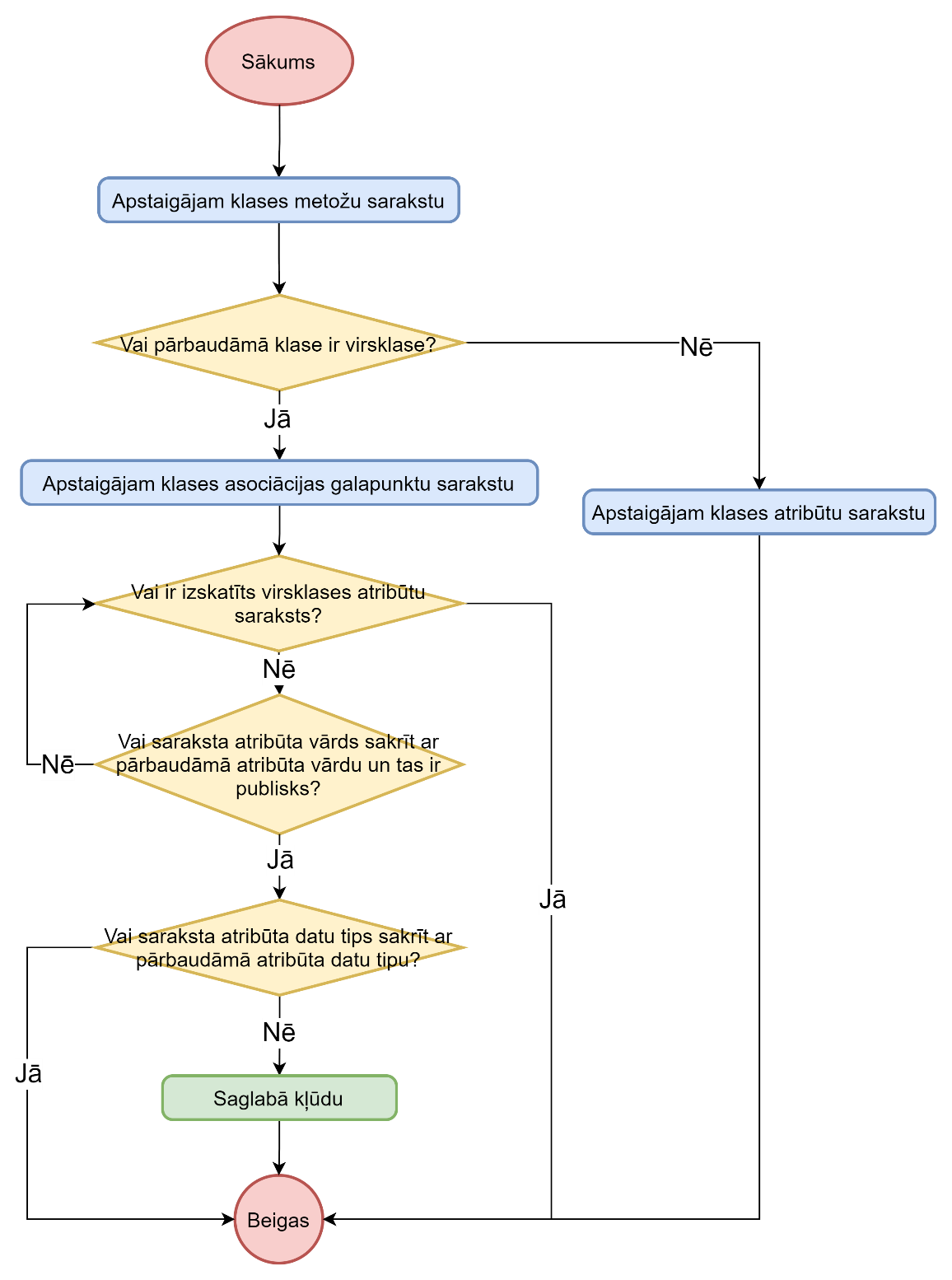
Šī sadaļa apraksta atribūta un metodes vārda kompilēšanas kopīgos aspektus. Vispirms ir jāpārliecinās, ka lauka vārds nesakrīt ar klases vārdu. Tad jāpārliecinās, ka tas nesakrīt ar rezervētajiem vārdiem [skat. 1.5 nodaļu] un ka vārds nesākas ar “\_constructor\_”. Visbeidzot ir jāapskata visi lauki un asociācijas un jāpārliecinās, ka padotais lauka vārds nesakrīt ar kādu no citu lauku un asociāciju vārdiem. 3.24 attēlā ir divi bloki “Vai klases iekšienē ir sastopams lauks ar padoto vārdu?” un “Vai virsklases iekšienē ir sastopams lauks ar padoto vārdu?”. Abos blokos tiek izmantotas vienādas funkcijas, tikai pirmajā blokā padot bāzes klasi un otrajā blokā – virsklasi. Šīs funkcijas ir aprakstītas nodaļās 3.2.3.5 atribūtiem un 3.2.4.2 metodēm. Tiek iziets cauri visām virsklasēm un ja nevienā pārbaudē nav atrasts eksistējošs elements, tad lauka vārds tiek saglabāts.

**

*3.24. att.* **lauka vārdu kompilēšanas secību diagrammas.**

#### 3.2.3.5 Atribūta vārda pārbaudes projektējums

Nodaļā 3.2.3.4 tika aprakstīts lauka vārda kompilēšana. Šī funkcija izskata klases un tās virsklašu laukus un asociāciju galapunktus un skatās, vai padotais atribūta vārds jau tiek izmantots. Vispirms tiek izskatīts metožu saraksts, jo šo sarakstu izskatīšanai nav jāskatās uz to, vai tiek pārbaudīta virsklase, vai bāzes klase, vienīgā piebilde, ka, pārbaudot virsklašu metodes, ir jāskatās, vai tās ir publiskas. Citu atribūtu apskatīšanā bāzes klases gadījumā tikai jānoskaidro tas, vai padotais atribūta vārds jau eksistē. Virsklašu gadījumos pārbaudi veic tikai publiskajiem atribūtiem. Pārbaudes laikā skatās, vai sakrīt datu tipi gan virsklasē, gan bāzes klasē. Asociācijas galapunktus pārbauda tikai virsklasēs, jo asociācijas galapunkti tiek definēti tikai pēc klases definēšanas, tāpēc, bāzes klasē šādu pārbaudi veikt nav nepieciešams.

**

*3.25. att.* **atribūta vārda pārbaudes secību diagrammas.**

### 3.2.4 Klases metožu specifiskās kompilēšanas projektējumi

#### 3.2.4.1 Metodes kompilēšanas specifisko funkciju projektējums

Nodaļā 3.2.3.2 tika pieminēts bloks “Specifiskās funkcijas”. Metodes gadījumā Tiek vēl pārbaudīts, vai ir definētas metodes anotācijas Ja tās ir definētas, tad vēl ir jāpārliecinās, vai metodei ir definēts URL.



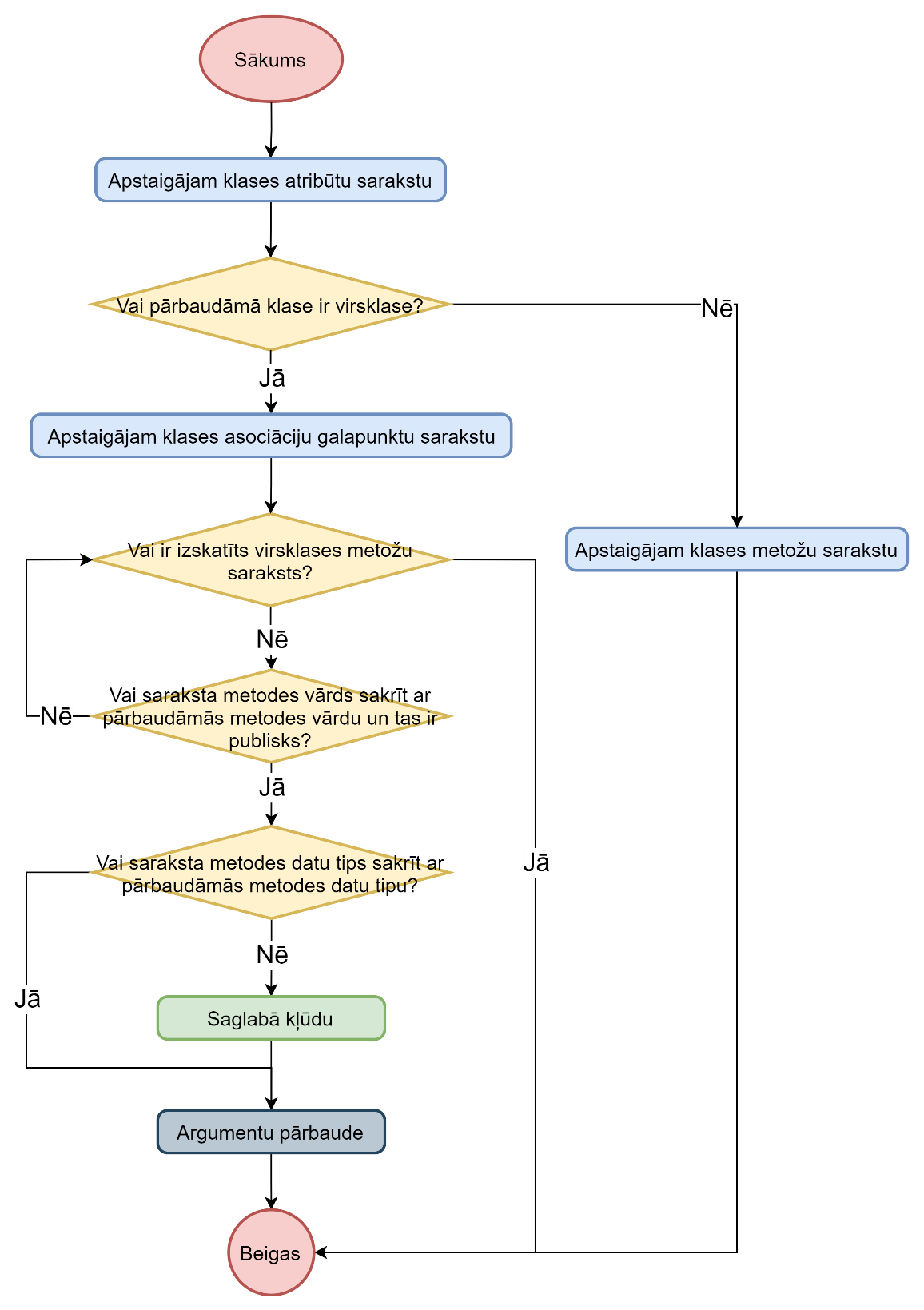
*3.26. att.* **metodes kompilēšanas specifisko funkciju secību diagrammas (Anotāciju pārbaude pa kreisi un argumentu pārbaude pa labi).**

#### 3.2.4.2 Metodes vārda pārbaudes projektējums

Metodes vārda pārbaudes funkcija ir līdzīga atribūta pārbaudes funkcijai [skat. 3.2.3.5 nodaļu], atšķiras tikai ar to, ka, neatkarīgi no bāzes klases un virsklases, atribūtu un asociāciju galapunktu sarakstu pārskats ir vienāds, bet citu metožu saraksta pārskats atšķiras.

Metodes vārda pārbaude virsklasē notiek sekojoši:

1. Atrod metodi ar tādu pašu vārdu.
2. Noskaidro, vai metodēm sakrīt atgriežamais tips.
3. Noskaidro, vai metodēm ir vienāds argumentu skaits [skat. 3.28. att.].
4. Noskaidro, vai i-tais arguments abās metodēs neatšķiras pēc datu tipa un vārda visiem i = {1,2,…,n}, kur n ir argumentu skaits [skat. 3.28. att.].

****

*3.27. att.* **metodes vārda pārbaudes secību diagrammas pamatfunkcijas daļa.**

**

*3.28. att.* **metodes vārda pārbaudes secību diagrammas argumentu pārbaudes daļa.**

#### 3.2.4.3 Argumentu kompilēšanas projektējums

Vispirms ir jānoskaidro, vai metodei argumentu definīcijā ir kaut viens ieraksts. Tad tiek veikta pārbaude, vai tiek skatīts komats vai arguments. Visu laiku ir jāpārliecinās, ka argumentu definīcija sākas un beidzas ar argumentu un katrs arguments tiek atdalīts ar komatu.

******

*3.29. att.* **argumentu kompilēšanas secību diagrammas.**

#### 3.2.4.4 Argumenta kompilēšanas projektējums

Nodaļā 3.2.4.3 tika aprakstīta visas argumenta definīcijas kompilēšana. Šajā nodaļā ir aprakstīta argumenta elementa kompilešana. Funkcija ir līdzīga lauku pamatkompilēsanai [skat. 3.2.3.2 nodaļu] tikai nav jāveic aizsardzības pārbaude un specifiskā funkcija ir argumenta saglabāšana metodē.

****

*3.30. att.* **argumenta kompilēšanas secību diagrammas.**

#### 3.2.4.5 Argumenta vārda kompilēšanas projektējums

Argumenta vārda kompilēšanā ir svarīga tikai divas lietas:

* Argumenta vārds nesakrīt ar rezervētajiem vārdiem [skat 1.5 nodaļu]
* Argumenta vārds neatkārtojas starp citiem argumentiem.

****

*3.31. att.* **argumenta vārda kompilēšanas secību diagrammas.**

### 3.2.5 Metožu anotāciju kompilēšanas projektējumi

#### 3.2.5.1 Anotācijas daļu kompilēšanas projektējums

Kā zināms, anotācijas tiek rakstītas formātā *‘[tips(“vērtība”)]’* [skat. 2.1.1.5 nodaļu]. Šajā formātā redzam trīs daļas, kuras tiek definētas ar atverošo un aizverošo simbolu, un kurā iekšā ir sastāvs:

* *[tips(“vērtība”)],* kur *‘tips(“vērtība”)’* ir sastāvs (“Kompilēt anotāciju”)
* *(“vērtība”),* kur *“vērtība”* ir sastāvs (“Kompilēt anotācijas ķermeni”)
* *“vērtība”,* kur *‘vērtība’* ir sastāvs (“Kompilēt anotācijas definīciju”)

Visām trim daļām tiek pielietota šī funkcija, bet tikai trešajā variantā tiek pārbaudīts atverošais un aizverošais simboli. Funkcijas “Kompilēt anotāciju” beigās tiek pārbaudīts, vai tika skatīta URL anotācija. Ja netika, tad metodes anotāciju sarakstā saglabā parasto anotāciju.

**

*3.32. att.* **anotācijas daļu kompilēšanas secību diagrammas.**

#### 3.2.5.2 Anotācijas satura kompilēšanas projektējums

Anotācijas saturā ir jānoskaidro, vai ir definēts anotācijas tips un vai ir definēts ķermenis. Gadījumā, ja tie ir definēti, tie tiek kompilēti. Ja nav definēts anotācijas tips, tad tas tiek uzskatīts par anotāciju, kas nav URL.

**

*3.33. att.* **anotācijas satura kompilēšanas secību diagrammas.**

#### 3.2.5.3 Anotācijas tipa kompilēšanas projektējums

Šajā funkcija tiek noskaidrots, kāda anotācija tiek definēta. Var būt divi gadījumi:

* Tiek definēts URL
* Tiek definēta cita anotācija

Ja definēts URL, tad jāpārliecinās, vai metodei jau ir definēts URL. Ja tā ir kāda cita asociācija, tad jāpārliecinās, vai padotais anotācijas tips tiek atbalstīts. Šī arī ir funkcija, kura tiek veidotas jaunas anotāciju instances.

*****3.34. att.* **anotācijas tipa kompilēšanas secību diagrammas.**

#### 3.2.5.4 Anotācijas vērtības kompilēšanas projektējums

Šajā funkcijā tiek saglabāta anotācijas vērtība. Ja anotācijas tips nav URL, tad tiek saglabāta visa vērtība. Ja ir URL, tad jāpārbauda, vai URL jau ir definēts un ja nav, tad jāskatās URL specifiskie atribūti [skat. 3.2.5.5 nodaļu] un visu atlikušo daļu saglabā ka metodes ceļu.

****

*3.35. att.* **anotācijas vērtības kompilēšanas secību diagrammas.**

#### 3.2.5.5 URL atribūtu kompilēšanas projektējums

Šajā funkcijā tiek pārbaudīti URL atribūti – protokols un lokācija. Jāpārbauda vai tie ir definēti un ja ir, tad vai ir padoti atbalstīti protokoli/lokācijas.

****

*3.36. att.* **URL atribūtu kompilēšanas secību diagrammas.**

## 3.3 Ģeneratora projektējums

Šajā nodaļa tiek aprakstītas tikai tas funkcijas, kuram ir nepieciešama papildus apstrāde. Funkcijas, kuras tikai ģenerē tekstu, netiek aprakstītas.

### 3.3.1 Pamatģenerēšanas projektējums

Starpkoda ģenerēšana notiek tad un tikai tad, ja kompilēšanas rezultātā nav atrastas kļūdas. [skat. 3.2.1.3 nodaļu]. Katra klase tiek ģenerēta savā failā. Vispirms tiek uzģenerēta klase “BaseObject” [sintaksi skat. 2.1.3.1] un tad tiek uzģenerētas visas pārējās klases, ja tādas ir.



*3.37. att.* **pamatģenerēšanas secību diagrammas.**

### 3.3.2 Klases ģenerēšanas projektējums

Ģenerējot klasi, vispirms ir jānoskaidro tās ģenerējamo virsklasi. Ja pirmkodā tā tika padota, tad to arī izmanto, citādāk kā virsklasi izmanto BaseObject. Kad virsklase tiek iegūta, tad ģenerējam klases galvu un ķermeni [sintaksi skat. 2.1.3.2 nodaļā]. Klases ķermenis tiek ģenerēts šādā secībā.

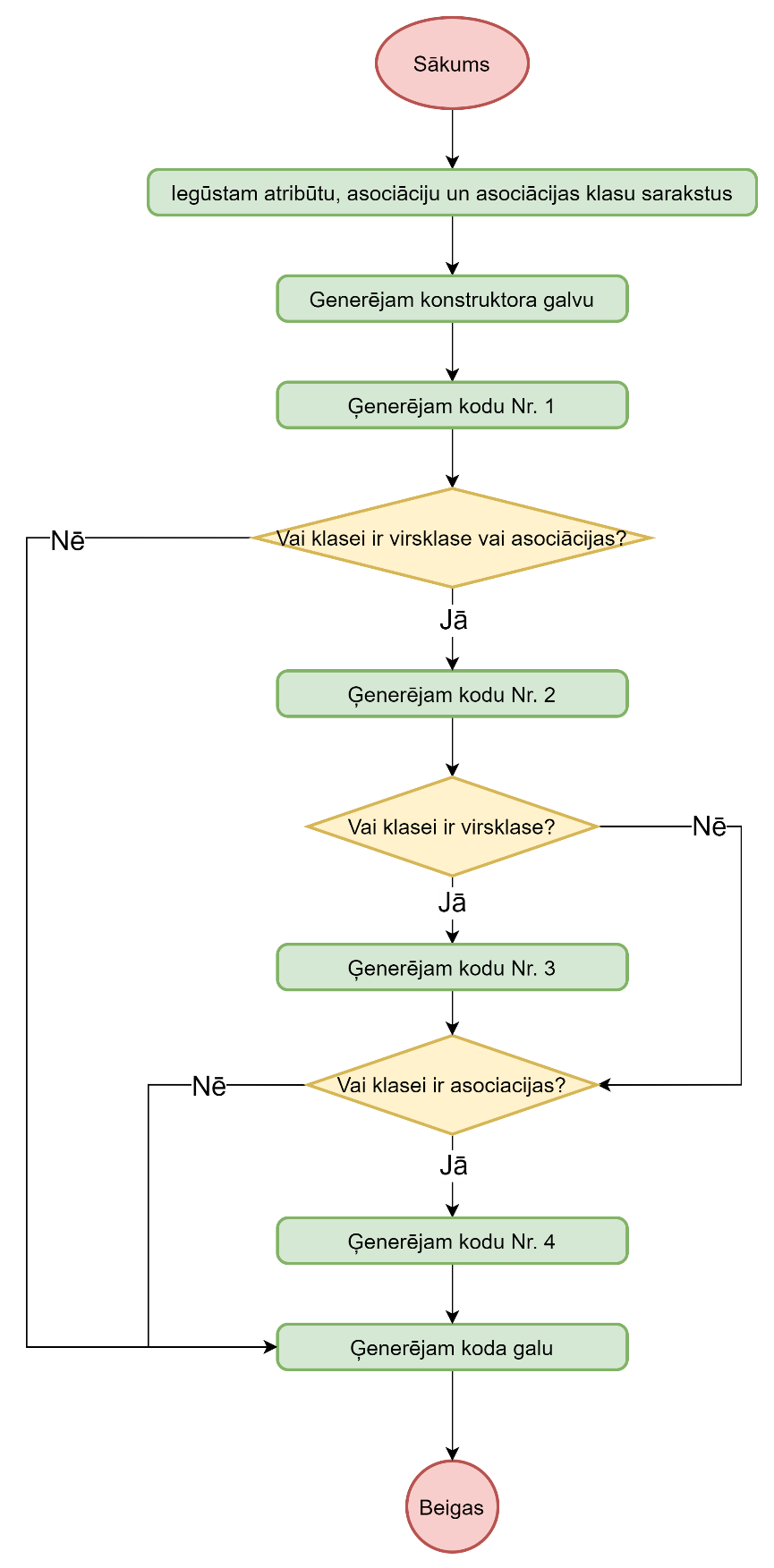
1. Konstruktori
2. Atribūti [skat. 3.3.4 nodaļu]
3. Asociāciju galapunkti [skat. 3.3.5 nodaļu]
4. Metodes [skat. 3.3.6 nodaļu]

**

*3.38. att.* **klases ģenerēšanas secību diagrammas.**

### 3.3.3 Klases konstruktora funkcijas ģenerēšanas projektējums

Šajā nodaļā tiek aprakstīts klases konstruktora funkcija, kura glabājas klasē “BaseObject” [skat. 2.1.3.1 nodaļā esošo dzeltenajā krāsā iekrāsoto kodu].

**

*3.39. att.* **klases konstruktora funkcijas ģenerēšanas secību diagrammas.**

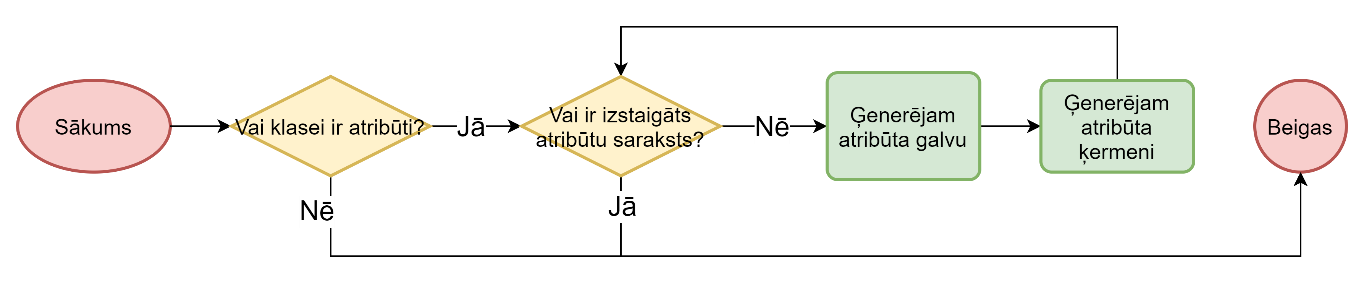
Tabulā 3.3 *klasesVārds* ir klases, kurai funkciju ģenerē, vārds, *attributes* ir atribūtu saraksts, *associations* ir asociāciju saraksts, *virsklasesVārds* ir klases tiesas virsklases vards, *asociācijasKlase* ir klases, ar kuru ir asociācija, vārds.

|  |  |
| --- | --- |
| **Koda gabala nosaukums 3.39. attēlā** | **Kods** |
| Galva | protected void \_constructor\_*klasesVārds*  { |
| Nr. 1 | List<string> attributes = new ( *attributes* );  var o = checkClass( attributes , “*klasesVārds”*); |
| Nr. 2 | if (o == false)  { |
| Nr. 3 | // SuperClass check  \_constructor\_*virsklasesVārds*();  var c = \_wm.FindClassByName(“*klasesVārds*”);  c.CreateGeneralization(“*virsklasesVārds*”); |
| Nr. 4 | List<string> associations = new ( *associations* );  \_constructor\_*asociācijasKlase*();  for(int x=0; x<associations.Count; x+=4)  {  checkAssociationEnd( associations[x] , associations[x+1] , " *klasesVārds*" , associations[x+2] , associations[x+3] );  } |
| Gals | }  }  (Divas aizverošās iekavas tiek ģenerētas tikai tad, ja tiek ģenerēts Nr. 2, citādi ģenerē tikai vienu aizverošo iekavu). |

*3.3. tabula.* **klases konstruktora funkcijas koda daļu identifikācija 3.39. diagrammā.**

### 3.3.4 Klases atribūtu ģenerēšanas projektējums

Ģenerējot klases atribūtus, ir jāpārliecinās, vai klasei tādi ir definēti. Ja ir, tad tie tiek ģenerēti [sintaksi skat. 2.1.3.4 nodaļā] .

* 3.40. att.* **klases atribūtu ģenerēšanas secību diagrammas.**

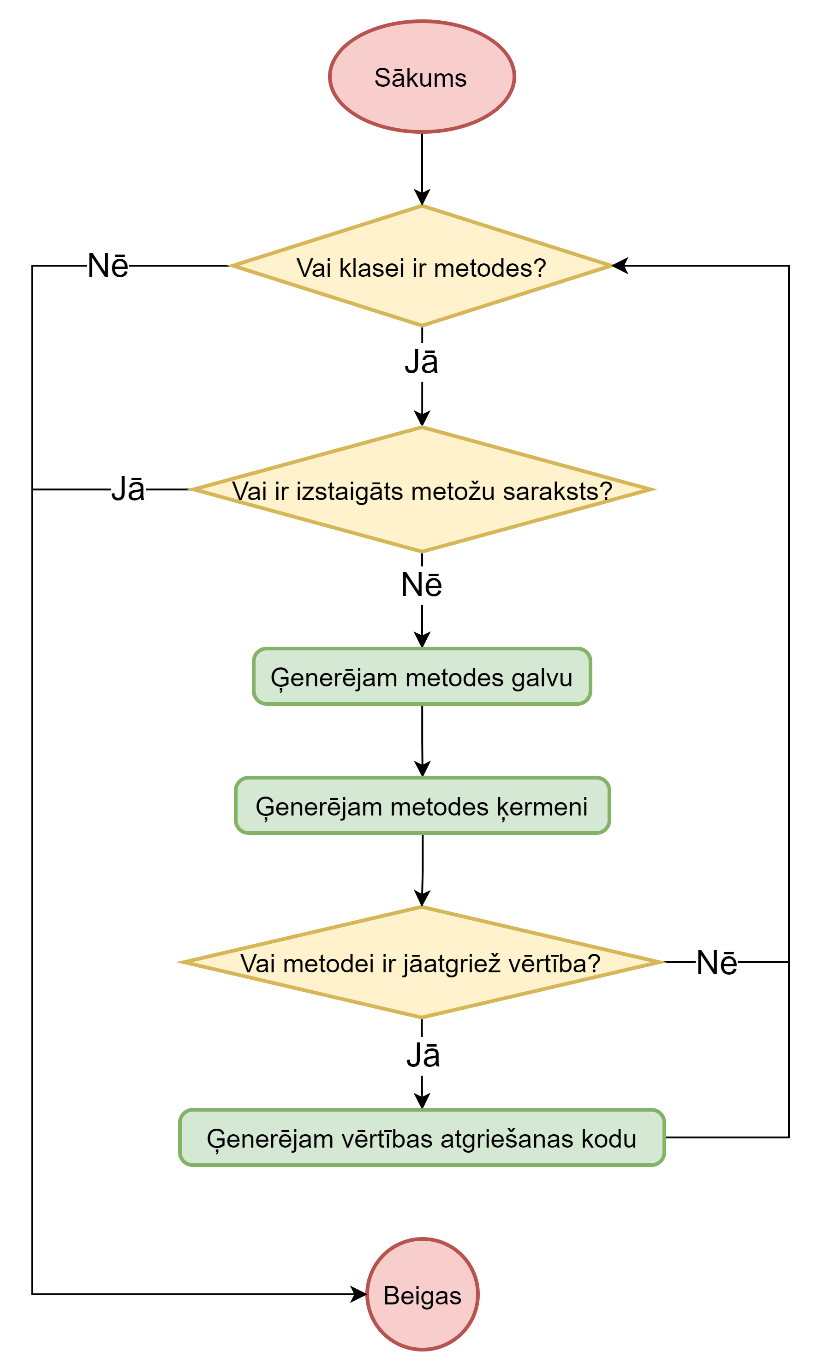
### 3.3.5 Klases asociāciju ģenerēšanas projektējums

Ģenerējot klases asociāciju galapunktus, ir jāpārliecinās, vai klasei ir definētas asociācijas. Ja ir, tad galapunkti tiek ģenerēti [sintaksi skat. 2.1.3.5 nodaļā] .

* 3.41. att.* **klases asociāciju galapunktu ģenerēšanas secību diagrammas.**

### 3.3.6 Klases metožu ģenerēšanas projektējums

Ģenerējot klases metodes, ir jāpārliecinās, vai klasei tādas ir definētas. Ja ir, tad klasei ģenerē metodes [sintaksi skat. 2.1.3.5 nodaļā]. Var gadīties, ka metodei ir datu tips void jeb metodei nav jāatgriež vērtība. Tādā gadījumā nav jāģenerē koda daļa, kura atgriež vērtību.

**

*3.42. att.* **klases metožu ģenerēšanas secību diagrammas.**

## 3.4 Ģenerēto klašu lauku un funkciju projektējums

### 3.4.1 Konstruktora projektējums

Katrai klasei ir trīs konstruktori. Pēc funkcionalitātes konstruktori ir vienādi. Divus no trim konstruktoriem atšķir tas, vai konstruktoriem tiek padota atsauce uz objektu. Ja tāda tiek padota, tad tiek veidots jau esošs objekts. Citādi tiek veidots pavisam jauns objekts. Trešajam konstruktoram padot tikai WebMemory (šajā konstruktorā RAAPI netiek veidots jauns objekts). Šo konstruktoru izmanto virsklašu un asociācijas klašu sagatavei RAAPI. Pirms objekta izveidošanas tiek pārbaudīta klase [skat. 3.4.2 nodaļu] un tās virsklases un asociācijas, ja tādas ir dotas [skat. 3.44 un 3.45 att.].

Attēls, kurā ir teksts

Apraksts ģenerēts automātiski

*3.43. att.* **konstruktora secību diagramma**

Virsklases pārbaudē vispirms tiek izsaukts tās konstruktora funkcija. Kad visas virsklases ir izveidotas, tad tiek veidota ģeneralizācija starp tiešajām apakšklasēm un virsklasēm, piemēram, ja ir klase *test1, test2,* kas manto no *test1*, un *test3,* kas manto no *test2*, un tiek veidota jauna instance klasei *test3*, tad kopumā generalizācijas veido starp klasem *test1* un *test2*, un *test2* un *test3*.

Attēls, kurā ir teksts

Apraksts ģenerēts automātiski *3.44. att.* **Virsklases pārbaudes secību diagramma**

Asociāciju pārbaudē vispirms tiek izwsauktas to konstruktoru funkcijas. Kad visas asociācijas klases tiek izveidotas, tad katra asociācija tiek pārbaudīta [skat. 3.4.3 nodaļu].



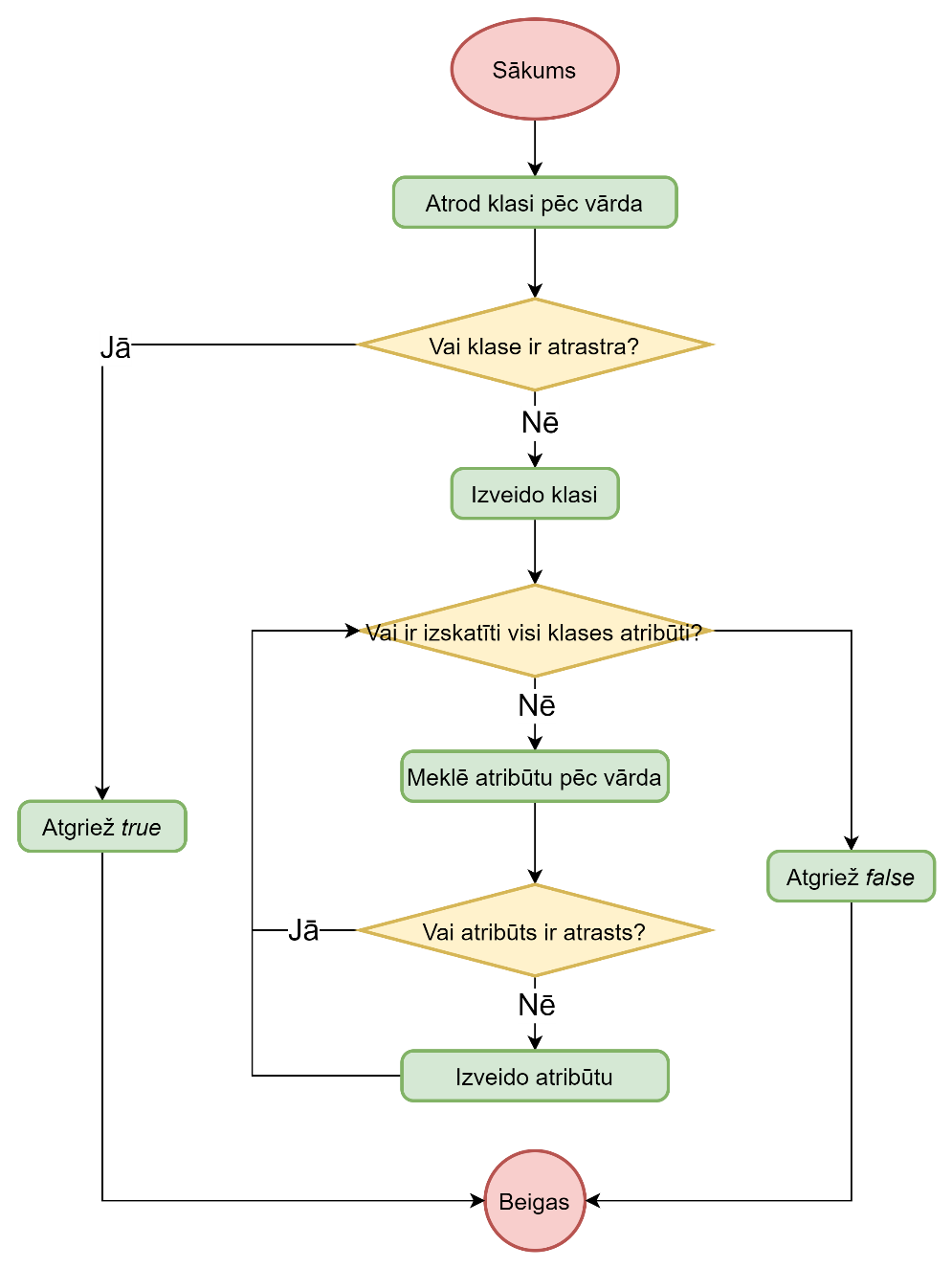
*3.45. att.* **Asociācijas klases pārbaudes secību diagramma.**

### 3.4.2 Klases pārbaudes projektējums

Klases pārbaudei tiek veiktas divas lietas.

1. Cenšas atrast klasi pēc vārda
2. Cenšas atrast klases atribūtus pēc vārdiem (kad klase jau ir atrasta/izveidota).

Gadījumos, kad kaut kā nav (nav klases vai tās atribūta), tad to izveido. Kad tas ir izdarīts, tad tiek izveidots jauns objekts, ja tas ir nepieciešams.



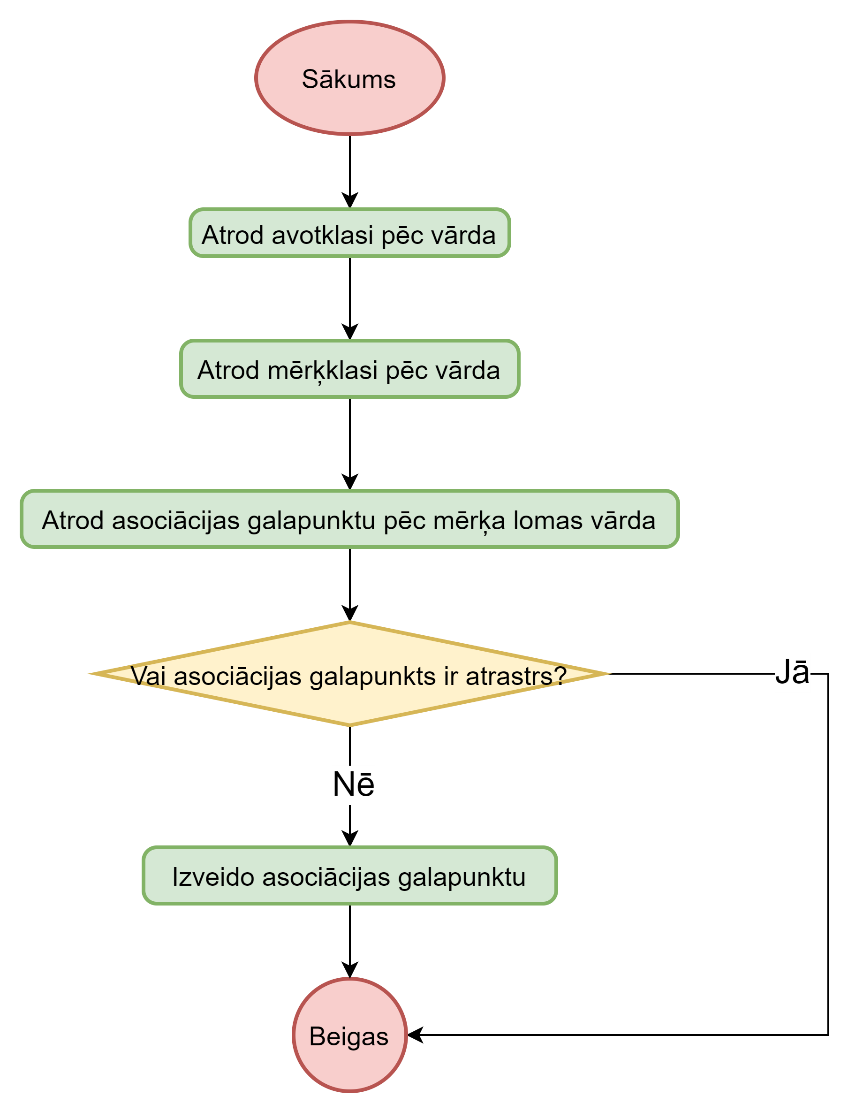
*3. 46. att.* **klases pārbaudes secību diagramma**

### 3.4.3 Asociāciju pārbaudes projektējums

Katras asociācijas pārbaudei tiek veiktas trīs lietas.

1. Atrod avotklasi pēc vārda
2. Atrod mērķklasi pēc vārda
3. Cenšas atrast asociācijas galapunktu avotklasē pēc mērķa lomas vārda.

Ja galapunkts netiek atrasts, tad tas tiek izveidots. Kad tas ir izdarīts, tad tiek atgriezts asociācijas galapunkta objekts, kuru izmanto klases asociāciju īpašībās [skat. 3.2.4 un 3.2.5 nodaļas].



*3.47. att.* **asociācijas pārbaudes secību diagramma**

### 3.4.4 Asociācijas saraksta iegūšanas projektējums

Lai sāktu saraksta iegūšanu, vispirms jāiegūst asociācija. Tad no WebMemory iegūstam sarakstu ar objekta saistītajiem objektiem un izveidojam tukšu sarakstu ar mērķklašu objektiem, kurš ir rezultātsaraksts. Izejam cauri sarakstam, kuru iegūstam no WebMemory un katram saraksta elementam izveidojam objektus ar mērķklases tipu un pievienojam rezultātsarakstam. Kad visiem elementiem ir iziets cauri, tad rezultātsarakstu atgriež.



*3.48. att.* **asociācijas saraksta iegūšanas secību diagramma**

### 3.4.5 Asociācijas saraksta uzstādīšanas projektējums

Šīs funkcijas struktūra ir līdzīga saraksta iegūšanas funkcijai [skat. 3.4.4 nodaļu], tikai saistīto objektu saraksts ir rezultātsaraksts un mērķklases objektu saraksts tiek padots ievaddatos. Kad objektu saraksts ir iegūts pēc mērķklases saraksta izskatīšanas, tad RAAPI tas tiek atjaunots.



*3.49. att.* **asociācijas saraksta uzstādīšanas secību diagramma**

### 3.4.6 Metožu projektējums

Vispirms visus argumentus saglabājam kā JSON simbolu virkni. Tad tiek izsaukta funkcija [skat ? nodaļu], kuras rezultātā iegūstam simbolu virkni JSON formātā. Tad šo simbolu virkni parsējam kā JSON dokumentu, kurā glabājas vai nu funkcijas rezultāts, vai kļūda. Gadījumā, ja nav kļūdas, tad rezultātu atgriež noteiktā datu tipā.



*3.50. att.* **metožu secību diagramma**

## 3.5 WebMemory klašu funkciju projektējums

WebMemory klašu funkciju veidošanai ir paredzēts izmantot zema līmeņa RAAPI, kurš kvalifikācijas darba ietvaros jau ir dots. {atsauce uz RAAPI dokumentāciju}

### 3.5.1 Datu struktūru projektējums

Nodaļā 2.2.2. tika aprakstītas datu struktūras, kuras tiek izmantotas WebMemory. Šajā nodaļā ir izveidota datu struktūru diagramma ar visiem atribūtiem un ir aprakstītas lietas, kuras nav iekļautas iepriekš minētajā nodaļā.

Klasē “WebClass” glabājas divu veidu asociācijas: izejošās un ieejošās. Pieņemsim, ka ir asociācija starp klasēm “sourceClass” un “targetClass”, kur avota galapunktā ir klase “sourceClass” ar lomas vārdu “sourceName” un mērķa galapunktā ir klase “targetClass” ar lomas vārdu “targetName”. Tad klasei “sourceClass” ir izejoša asociācija ar lomas vārdu “targetName” un ieejoša asociācija ar lomas vārdu “sourceName”. Klasei “targetClass” ir izejoša asociācija ar lomas vārdu “sourceName” un ieejoša asociācija ar lomas vārdu “targetName”.

Klasēs glabājas ari atsauce, kuru izmanto elementa meklēsanai iekš RAAPI, pats RAAPI, lai izmantotu tās funkcijas un WebMemory, lai izmantotu augsta līmeņa RAAPI funkcijas.

**

*3.51. att.* **WebMemory datu struktūru klašu diagramma**

### 3.5.2 Vārdnīcu izveides projektējums

Visu vārdnīcu izveidei ir vienāda struktūra [skat. 2.1.8.1 nodaļu]. Vispirms sagatavojam vārdnīcu un no RAAPI iegūstam vēlamo instanču saraksta pirmo iteratoru. Visiem saraksta iteratoriem iziet cauri, kur katrā iterācijā vārdnīcā pievienojam jaunus elementus. Pēc iterēšanas atbrīvojam atmiņu no iteratora un atgriežam visu vārdnīcu. Vārdnīcas tiek izmantotas sarakstu veidošanā [skat. 3.5.2 nodaļu] un elementu atrašanā [skat. 3.5.5 nodaļu]



*3.52. att.* **vārdnīcu izveides secību diagramma**

### 3.5.3 Sarakstu iegūšanas projektējums

Izmantojot iepriekšējā nodaļā aprakstīto funkciju, iegūst vārdnīcu, no kura iegūst sarakstu ar instancēm, kuru atgriež. Saraksti, kuriem šī funkcija ir veidota, ir uzskaitīti 2.1.8.2 nodaļā.



*3.53. att.* **saraksta iegūšanas secību diagramma**

### 3.5.4 Instances izveidošanas projektējums

Iekš RAAPI tiek izveidota jauns elements. Ar elementu izveidošanas funkciju tiek iegūta elementa atsauce, kuru izmanto, lai izveidotu elementa instanci, kuru atgriež. 2.1.8.3 nodaļā ir uzskaitītas klases, kurām šī funkcija tiek izmantota.



*3.54. att.* **Instances izveidošanas secību diagramma**

### 3.5.5 Instances dzēšanas projektējums

Vispirms atrodam instanci, kuru ir jāizdzēš [skat. 3.5.5 nodaļu]. Ja instance ir atrasta, tad to izdzēš, citādāk nekas nenotiek. 2.1.8.4 nodaļā ir uzskaitītas klases, kurām šī funkcija tiek izmantota.

*Attēls, kurā ir teksts

Apraksts ģenerēts automātiski*

*3.55. att.* **instances dzēšanas secību diagramma**

### 3.5.6 Instances atrašanas projektējums

Vispirms ir jāiegūst instances atsauce, ja tiek veikta meklēšana pēc vārda. Tad, izmantojot 3.5.1 nodaļā aprakstīto funkciju, iegūst vārdnīcu, kurā pārbauda, vai eksistē atslēga, kas sakrīt ar iegūto atsauci uz instanci. Ja atsauce eksistē, tad atgriežam instaci, citādāk atgriežam neko. 2.1.8.5 nodaļā ir uzskaitītas klases, kurām šī funkcija tiek izmantota.

**

*3.56. att.* **instances atrašanas secību diagramma**

### 3.5.7 Atribūta vērtības iegūšanas projektējums

Lai iegūtu atribūta vērtību, ir jāiegūst visas objekta klases un starp tām ir jāatrod atribūts. Ja tas ir atrasts, tad atgriežam atribūta vērtību. Šo funkciju izmanto ģenerētais kods atribūtu ‘get’ funkcijā.

**

*3.57. att.* **atribūta vērtības iegūšanas secību diagramma**

### 3.5.8 Atribūta vērtības uzstādīšanas projektējums

Šī funkcija ir līdzīga iepriekšējā nodaļa aprakstītajai funkcijai, vienīgā atšķirība ir tāda, ka vērtība netiek atgriezta, bet gan uzstādīta. Šo funkciju izmanto ģenerētais kods atribūtu ‘set’ funkcijā.

**

*3.58. att.* **atribūta vērtības uzstādīšanas secību diagramma**

### 3.5.9 Objektu sasaistīšanas projektējums

Šajā funkcijā tiek izdzēstas visas vecās saites starp objektiem un tiek uzstādītas jaunas. Saišu dzēšana un izveidošana tiek veikta ar RAAPI iebūvētajām funkcijām.

**

*3.59. att.* **objektu sasaistīšanas secību diagramma**

## 3.6 WebCalls funkciju projektējums

### 3.6.1 Remote funkciju izsaukšana

Nodaļā 2.1.10.1 tika aprakstīts, ka vienīgais, kas tiek veikts, ir funkcijas “*enqueueInnerWebCall*” izsaukšana, kuras rezultātā iegūst JSON simbolu virkni. Šī simbolu virkne funkcijas beigās tiek atgriezta.

*3.60. att.* **Remote funkciju izsaukšanas secību diagramma**

### 3.6.2 C# metodes datu iegūšana

Nodaļā 2.1.9.2 tika aprakstīts, ka metodes URL ir formātā “dotnet:local:namespace.classname#methodname” un ka ir jāiegūst saraksts, kas satur šādus elementus: ”dotnet” , “local” , “namespace.classname” un “methodname”, tāpēc tiek sagatavoti atdalītāji, kuri šajā gadījumā ir “:” un “#”. Izmantojot šos atdalītājus, padoto URL sadalam norādītajās daļās un ievietojam tās sarakstā, kuru beigās atgriežam.

*3.61. att.* **C#** **metodes datu iegūšanas secību diagramma**

### 3.6.3 Lokālā funkciju izsaukšana (Local Web Call)

Izmantojot iepriekšējā nodaļā aprakstīto funkciju no metodes URL iegūstam metodes datus. Ar šiem datiem iegūstam nepieciešamo klasi (izmantojam “namespace.classname”) un klasē atrodam nepieciešamo metodi (izmantojam “methodname”). Ja metode ir atrasta, tad tā tiek izsaukta un tās vērtība tiek atgriezta, citādāk atgriežam JSON virkni, kurā glabājas kļūda, ka metode nav atrasta.

**

*3.62. att.* **Lokālās funkciju izsaukšanas** **secību diagramma**