LATVIJAS UNIVERSITĀTE
FIZIKAS UN MATEMĀTIKAS FAKULTĀTE
DATORIKAS NODAĻA

KONKATENATĪVAS LATVIEŠU VALODAS RUNAS SINTĒZES SISTĒMAS IZVEIDE

BAKALAURA DARBS

Autors: **Mārcis Pinnis**
Stud. apl. Nr. DatZ 040023
Darba vadītājs: *Mg. Sc. Comp.* Normunds Grūzītis

RĪGA 2008

ANOTĀCIJA

Bakalaura darbā ir aprakstīta autora izstrādāta konkatenatīvas latviešu valodas runas sintēzes sistēma. Bakalaura darba ietvaros tika izstrādāta sistēmas arhitektūra un runas sintēzes bibliotēka, kas nodrošina nepieciešamo funkcionalitāti, lai sistēmu būtu iespējams integrēt ārējos risinājumos.

Darbā tiek aprakstīti procesi, kā no teksta latviešu valodā tiek iegūts audio fails, kas satur sākotnējā teksta akustisko reprezentāciju, respektīvi, runu. Darbā tiek arī apskatīti principi, kā no atsevišķiem audio fragmentiem tiek izveidots nepieciešamais audio fails, fragmentus ar dažādām metodēm kombinējot kopā.

Atslēgvārdi: konkatenatīva runas sintēze, digitālo signālu apstrāde.

ABSTRACT

The Bachelor paper describes a concatenative Latvian language speech synthesis system developed by the author. Within the framework of the Bachelor paper the architecture of the system and the speech synthesis library, that provides the necessary functionality, so that the system can be integrated into external systems, was developed.

In the paper there are examined processes of how an audio file, is gained from a text in Latvian, where the resulting audio file contains the acoustic representation of the text, respectively, the speech.

In the paper the author looks at principles of how an audio file is combined together from smaller audio fragments using different concatenation methods.

Keywords: concatenative speech synthesis, digital signal processing.

SATURS

APZĪMĒJUMU SARAKSTS

RIFF	Multivides datu formāts (*Resource Interchange File Format*).
SAPI	*Microsoft* izstrādāta runas apstrādes bibliotēka (*Speech Application Programming Interface*).
T2S	Konkatenatīvas latviešu valodas runas sintēzes sistēmas tehniskais nosaukums; tiek lietots arī kā apzīmējums runas sintēzei (*Text-To-Speech System*).
UML	Universāla modelēšanas valoda, kas tiek lietota programmatūras projektējumu izstrādē (*Unified Modeling Language*).
WAVE	Audio faila formāts, RIFF failu formāta paveids (*Waveform Audio Format*).
WAV	Audio faila formāts (sinonīms apzīmējumam WAVE), apzīmējums tiek lietots arī kā audio failu nosaukumu paplašinājums (*Waveform Audio Format*).
XML	Datu iezīmēšanas valoda (*Extensible Markup Language*).

IEVADS

Konkatenatīva latviešu valodas runas sintēzes sistēma ir latviešu valodai izstrādāta informācijas sistēma, kas spēj latviešu valodā rakstītu tekstu pārvērst tā akustiskajā reprezentācijā, respektīvi runā. Runas signāls tiek veidots kombinējot kopā iepriekš ierakstītus runas fragmentus (vārdus, skaņas) un automātiski ģenerētus fragmentus (klusuma periodus, skaņu pārejas)

Saistībā ar komunikācijas tehnoloģiju uzņēmumu interesi par runas sistēmu (runas analīze, runas sintēze, dialogu sistēmas) attīstību un potenciālo izmantošanu Latvijā, runas sintēze ir kļuvusi par aktuālu tēmu, kurai nepieciešams pievērst uzmanību. Sadarbībā ar SIA „Lattelecom BPO" 2007. gada oktobrī Latvijas Universitātes Matemātikas un informātikas institūta Mākslīgā intelekta laboratorijā tika uzsākts projekts par runas sintēzes un analīzes izpēti un sistēmu izstrādi latviešu valodai. Tā kā darba autors bija atbildīgs par runas sintēzes sistēmas izstrādi, bakalaura darbā tika izvēlēts apskatīt tieši institūtā izstrādāto runas sintēzes sistēmu.

Tā kā pašlaik latviešu valodai ir izveidotas tikai divas runas sintēzes sistēmas, kuras abas ir saistītas ar *Microsoft Windows* operētājsistēmu un nav lietojamas citās operētājsistēmās, svarīgi ir izstrādāt sistēmu, kuru būtu iespējams lietot arī citās operētājsistēmās. Šādam nolūkam tika izstrādāta arhitektūra jaunajai latviešu valodas runas sintēzes sistēmai.

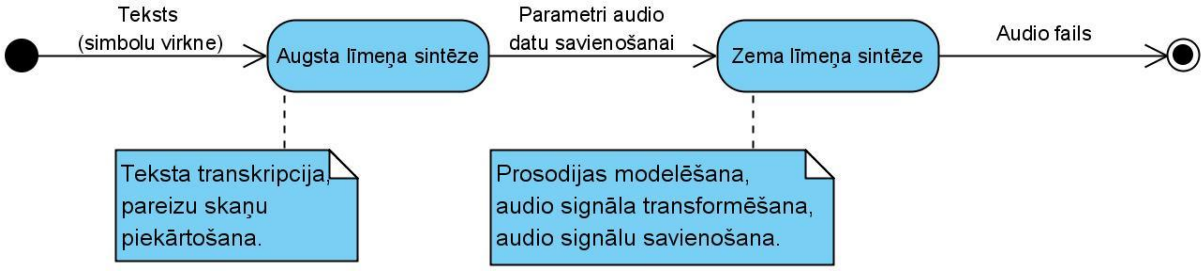
Bakalaura darbu veido četras pamatnodaļas. Pirmajā nodaļā tiek sniegta informācija par to, kas ir runas sintēze un kādi sasniegumi runas sintēzē bijuši Latvijā līdz šim. Otrajā nodaļā aprakstīta autora izstrādātās sistēmas koncepcija, sasniedzamie mērķi un sistēmas varbūtējās implementācijas metodes ar ārējiem risinājumiem. Trešajā nodaļā aprakstīta „Konkatenatīvas latviešu valodas runas sintēzes sistēmas" arhitektūra un izstrādātās funkcionalitātes apraksts. Ceturtajā nodaļā apskatīta sistēmas realizācija, ieskaitot sistēmas izstrādes posmus, sistēmas implementācija viena lietotāja datoram, kā arī izstrādē radušās problēmas un risinājumi.

1. IEVADS RUNAS SINTĒZĒ

Mūsdienās, kad cilvēki paliek aizvien vairāk atkarīgi no datoru klātbūtnes, kad robottehnoloģijas ir attīstītas jau tik tālu, ka cilvēku veidotie tehnoloģiskie šedevri ir spējīgi komunicēt ar cilvēkiem, kad automašīnās ievietotās navigācijas ierīces ir spējīgas cilvēkam nolasīt vēlamo braukšanas maršrutu, kad zinātniskajās laboratorijās aktīvi tiek izstrādātas kompleksas dialogu sistēmas, kas spēj komunicēt ar cilvēku sarunvalodā, nevis izmantojot kādu ierobežotu leksikonu, ir svarīgi saprast, kas ir runas sintēze un kā tā darbojas. Šajā darbā tiks apskatīta konkatentatīvas runas sintēzes sistēmas izveide, bet 1. nodaļā tiks pievērsta uzmanība tam, kādi ir runas sintēzes sistēmu veidi un mērķi, kā arī – kādi sasniegumi līdz šim bijuši Latvijā.

1.1. Kas ir runas sintēze?

Cilvēki, lai komunicētu viens ar otru, izmanto runu. Pateicoties runai iespējams saņemt svarīgu informāciju, kas tālāk tiek apstrādāta pēc nepieciešamības. Tā kā datori nav dzīvas būtnes, tie nespēj komunicēt ar runas palīdzību bez cilvēka radītu risinājumu palīdzības. Procesu, kad ar datora palīdzību tiek radīta mākslīgi veidota cilvēka runa, sauc par runas sintēzi [6]. Runas sintēzi datoros nodrošina īpaši šim mērķim paredzētas runas sintēzes sistēmas. Runas sistēmas atšķiras pēc to uzbūves, specifiskiem darbības principiem un pielietojuma mērķa, bet visām sistēmām ir viena līdzīga uzbūves īpatnība, proti, visām runas sintēzes sistēmām sintēzi sadala divās daļās, t. i., divos procesos – augsta līmeņa runas sintēzē un zema līmeņa runas sintēzē (sk. 1.1. attēlu).



1.1. att. **Runas sintēzes process**

Augsta līmeņa runas sintēzē ietilpst visi apstrādes procesi, kas saistīti ar teksta apstrādi (transkribēšana, skaitļu un saīsinājumu analīze), bet zema līmeņa sintēzē ietilpst tie apstrādes procesi, kas tiešā veidā saistīti ar audio signālu apstrādi (signālu savienošana, signālu ģenerēšana, prosodijas modelēšana utt.), līdz tiek iegūta nepieciešamā runa. Abi apstrādes procesi notiek secīgi viens aiz otra, t. i., vispirms tiek veikta augsta līmeņa sintēze, un tikai

pēc tam tiek veikta zema līmeņa sintēze. Šāda secība tiek ievērota, jo zema līmeņa sintēze ir atkarīga no datiem, kas iegūti augsta līmeņa sintēzē.

1.2. Runas sintēzes veidi

Lielākajai daļai runas sintēzes sistēmu pirmais apstrādes posms (augsta līmeņa sintēze) ir līdzīgs, t. i., no teksta tiek iegūti dati, kas nosaka, kādas skaņas nepieciešamas, kādas pauzes jāievēro, kā arī kādus prosodijas modelēšanas nosacījumus ievērot, bet sistēmām ievērojami atšķiras zema līmeņa sintēze. Atšķirības var būt visai vienkāršas, t. i., principos, kā tiek lietotājam atgriezta runa, bet atšķirības var būt arī ievērojamas, t. i., var atšķirties principi, kā tiek apstrādāti audio signāli un kā tie tiek veidoti. Zema līmeņa sintēzes principiāli atšķirīgās nodrošināšanas dēļ runas sintēzes sistēmas iedala trīs veidos [6].

Pirmais runas sintēzes veids ir konkatenatīva runas sintēze. Runa tiek sintezēta no iepriekš sagatavotiem audio fragmentiem, kas ar dažādām metodēm (pielietojot digitālo signālu apstrādes algoritmus) tiek kombinēti kopā, tādējādi konstruējot runu. Fragmentu kombinēšanas laikā tiek modelēta arī prosodija. Sistēmas, kas ietilpst šajā grupā parasti atšķiras ar to, cik lieli audio fragmenti (fonēmas, difoni, trifoni, veseli vārdi, izteicieni utt.) tiek glabāti datu bāzē runas sintēzes nodrošināšanai. Atkarībā no izvēlēto fragmentu lieluma tiek atbilstoši izstrādāta arī runas sintēzes sistēma, jo dažādu veidu fragmentiem audio datu apstrādes metodes var atšķirties.

Otrs runas sintēzes veids ir formantu sintēze. Formantu sintēzes laikā netiek izmantoti gatavi audio fragmenti, kā tas tika darīts iepriekšējā gadījumā. Katra skaņa sistēmā satur specifiskus parametrus (datu bāzē), kas nosaka īpašības, pēc kurām tiek ģenerēts audio signāls. Tiek norādīts, pie kādām frekvencēm kādi formanti ir sastopami, kādas ir pārejas starp formantiem un skaņām. Modelējot šos datus audio signālā, tiek iegūts vokālais audio signāls. Lai signāls būtu pilnībā uztverams, parasti tiek pievienoti specifiski trokšņi, lai skaņu padarītu cilvēka balsij līdzīgāku [6].

Trešais runas sintēzes veids ir artikulārā sintēze. Artikulārās sintēzes laikā sistēma cenšas modelēt cilvēka balss trakta darbību, tādējādi ģenerējot runas audio signālu. Modelējot pārmaiņas cilvēka balss trakta darbībā, veidojas pārmaiņas audio signālā, kas tiek ģenerēts, nodrošinot pāreju veidošanos starp skaņām, kā arī dažādu trokšņu veidošanu (audio signāla eksplozijas pie skaņām „p”, „k”, „t” utt.).

Sistēmas realizācijai tika izvēlēts konkatenatīvas runas sintēzes sistēmas veids, jo par šo sintēzes veidu jau ir uzkrāta pieredze, kā arī ir iespējams ātrākā laika intervālā iegūt kvalitatīvu rezultātu. Pārējo runas sintēzes veidu realizācijai nepieciešami iepriekš uzkrāti dati

par valodas fonētiskajām īpašībām un skaņām [6]. Tāpēc, ka par latviešu valodu nav uzkrāti nepieciešamie dati, uz doto brīdi ir nelietderīgi uzsākt sistēmu izstrādi, kas realizē šādus runas sintēzes veidus.

1.3. Runas sintēzes mērķi

Runas sintēzes galvenais mērķis ir ļaut cilvēkiem no informācijas sistēmām saņemt informāciju ērtākā veidā – nevis teksta formā, bet runā / audioformā. Tomēr runas sintēzei ir arī citi mērķi, kas ir atkarīgi no runas sintēzes izmantošanas veida. Runas sintēze tiek izmantota ļoti daudzās informācijas sistēmās, kam ir arī atšķirīgi mērķi un funkcijas. Dažas sfēras, kur iespējams izmantot runas sintēzi, ir sekojošas:

- Navigācijas ierīču izstrādē – cilvēks saņem informāciju par maršrutu audioformā, papildus vizuālajam.
- Zvanu centrālēs – ja nepieciešams bieži atkārtot vienu un to pašu tekstu, vieglāk ir ar runas sintēzes palīdzību nodrošināt atbildi zvanītājam.
- Datoru lietojumprogrammās, kas nodrošina teksta nolasi no ekrāna – it sevišķi cilvēkiem ar īpašām vajadzībām (akliem cilvēkiem un cilvēkiem, kas neprot lasīt) tiek padarīta iespējama datora lietošana.
- Dialogu sistēmās – kopā ar runas atpazīšanu tiek nodrošināta datora komunikācija ar cilvēku tikai ar runas palīdzību (izziņu iegūšanai, dažādu ierīču vadīšanai, datoram reaģējot uz cilvēka vēlmēm).
- Datorvārdnīcās – lai izprastu mērķa valodas vārdu pareizu izrunu.

Runas sintēzei pielietojumu ir ļoti daudz un tie ir ļoti dažādi, bet visiem ir viens kopīgs mērķis, t. i., atvieglot cilvēkam dzīvi, padarīt ērtāku datoru izmantošanu.

1.4. Līdzšinējie sasniegumi runas sintēzē Latvijā

Latvijā līdz brīdim, kad tika uzsākts darbs pie autora izstrādātās sistēmas, bija pieejamas divas funkcionējošas runas sintēzes sistēmas. Viena no šīm sistēmām ir Ivara Štrāla 2007. gadā izstrādātā runas sintēzes sistēma „Runātājs” [8], bet otra ir uzņēmuma SIA „Tilde” izstrādāta sistēma „Tildes Visvaris”.

Sistēma „Runātājs” ir veidota kā konkatenatīva runas sintēzes sistēma, kas paredzēta darbam tikai *Microsoft Windows* operētājsistēmā. Sistēma ir izstrādāta ar programmēšanas valodu *Microsoft Visual C++*, izmantojot *Microsoft SAPI 5.1*. Sistēma nodrošina difonu sintēzi, izmantojot tikai minimālu prosodijas modelēšanu (klusuma intervāli starp vārdiem, pieturzīmēm). Līdz ar to sistēmas galvenā darbība saistīta ar teksta transkribēšanu un audio

failu saraksta izveidošanu atskaņošanai. Sistēma ir integrējama *Microsoft Windows* operētājsistēmās, nodrošinot runas sintēzi lietojumprogrammās, kas izmanto *Microsoft Windows* runas sintēzes funkcionalitāti. Sistēma pilnvērtīgi nodrošina tikai augsta līmeņa runas sintēzi, bet nenodrošina zema līmeņa sintēzes iespējas (prosodijas modelēšana). Sistēmas datu bāze ir līdzīga autora izstrādātās sistēmas datu bāzes struktūrai, jo, izstrādājot jauno runas sintēzes sistēmu, tika ņemta vērā iepriekš iegūtā pieredze, protams, to pārstrukturējot, lai nodrošinātu efektīvāku un kvalitatīvāku rezultātu iegūšanu un resursu izmantošanu (bet par to nākamajās nodaļās). Sistēmas datu bāze ir vienīgā sistēmas sastāvdaļa, kas līdzinās jaunizveidotajai sistēmai. Algoritmiskā realizācija jaunizveidotajai sistēmai tika izstrādāta, neietekmējoties no „Runātāja" algoritmiem, jo izstrādātais kods ar tā piesaisti *Microsoft* risinājumiem neatbilst jaunās sistēmas prasībām (operētājsistēmas neatkarīga runas sintēzes nodrošināšana).

SIA „Tilde" izstrādātais runas sintezators arī ir integrējams un pilnībā atkarīgs no Microsoft Windows operētājsistēmas. Tas nozīmē, ka tā izmantošana citās operētājsistēmās nav iespējama. Par to, kā precīzi izstrādāts SIA „Tilde" risinājums, precīza informācija publiski nav pieejama, bet par sistēmas atkarību no *Microsoft Windows* operētājsistēmas liecina *Microsoft SAPI* izmantošana.

Autora izstrādātā konkatenatīvā runas sintēzes sistēma no šīm divām sistēmām atšķiras ar to, ka tiek nodrošināta dažādu garumu (difoni, trifoni, nepilni vārdi, veseli vārdi utt.) audio fragmentu savienošana, tiek nodrošināta sistēmā iekšēja zema līmeņa runas sintēze, kas nozīmē, ka veiksmīgāk iespējams modelēt prosodiju, galvenā atšķirība – sistēma nav piesaistīta pie *Microsoft Windows* operētājsistēmas un ir izmantojama arī veidojot tīklu risinājumus, nevis kā viena lietotāja lietojumprogramma.

2. LATVIEŠU VALODAS RUNAS SINTĒZES SISTĒMAS KONCEPCIJA

2.1. Sistēmas mērķi

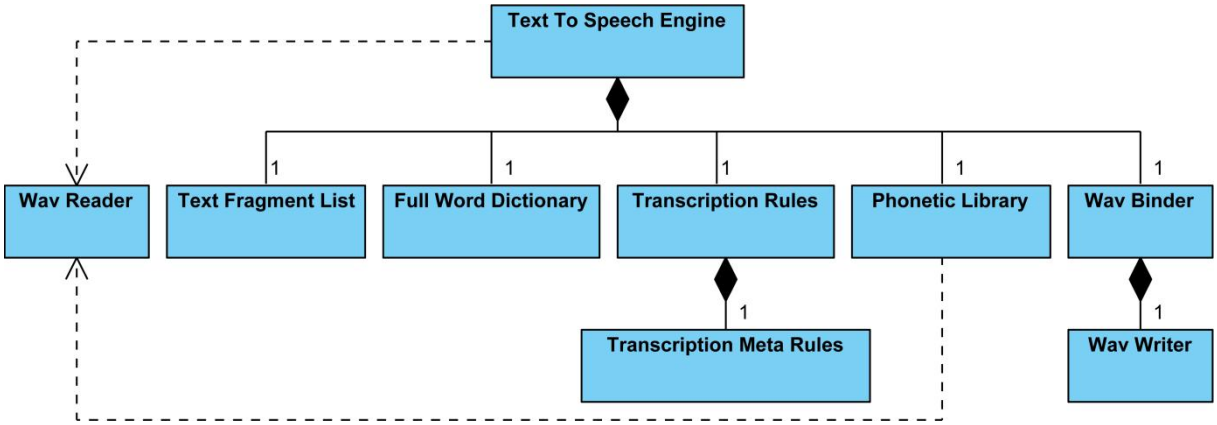
Latviešu valodas runas sintēzes sistēmai ir jānodrošina tekstrunas pārveidotāja (*text-to-speech converter*) funkcionalitāte. Tas nozīmē, ka lietojumprogrammai, saņemot ievaddatos tekstu latviešu valodā, jāspēj izveidot audio failu, kas satur šī teksta reprezentāciju runā. Sistēmas ģenerētajam audio signālam jāsatur uztverama latviešu valodas runa, lai būtu viennozīmīgi saprotams, kādi dati tika padoti lietojumprogrammai ievaddatos.

Kā papildus mērķis runas sintēzes sistēmai tiek izvirzīta prasība izstrādāt kvalitatīvu un lasāmu kodu, ko būtu iespējams atkārtoti izmantot sistēmas jaunāku versiju veidošanā, kā arī pa daļām pārņemt, izstrādājot ar runu saistītus risinājumus.

2.2. Sistēmas uzbūves koncepcija

Lai sasniegtu augstākminētos mērķus, ir izstrādāta konkatenatīva runas sintēzes sistēma latviešu valodai. Tas nozīmē, ka teksta pārveidošana runā tiek realizēta ar gatavu skaņas fragmentu palīdzību, tos kombinējot kopā, izmantojot papildus datu apstrādes posmus veiksmīgai prosodijas modelēšanai un runas uztveramības nodrošināšanai. Sistēma ir projektēta, ņemot vērā objektorientētās programmēšanas principus [7], kā arī ir integrējama citos risinājumos, kā patstāvīgs modulis vai arī kā bibliotēka ar runas sintēzes funkcionalitāti.

Sintezatora uzbūves galvenās klases un to savstarpējo saistību pilnvērtīgai sintēzes nodrošināšanai parāda sistēmas konceptuālā klašu diagramma (sk. 2.1.1. att.).



1.1.1. att. **Sistēmas konceptuālā klašu diagramma**

Sistēma sastāv no vienas pamatklases – tekstrunas pārveidotāja dziņa (*Text To Speech Engine*), kas sastāv no vairākām komponentēm, respektīvi, teksta fragmentu saraksta (*Text Fragment List*), veselu vārdu vārdnīcas (*Full Word Dictionary*), transkribēšanas likumiem (*Transcription Rules*), fonētiskās bibliotēkas (*Phonetic Library*) un audio datu savienotāja

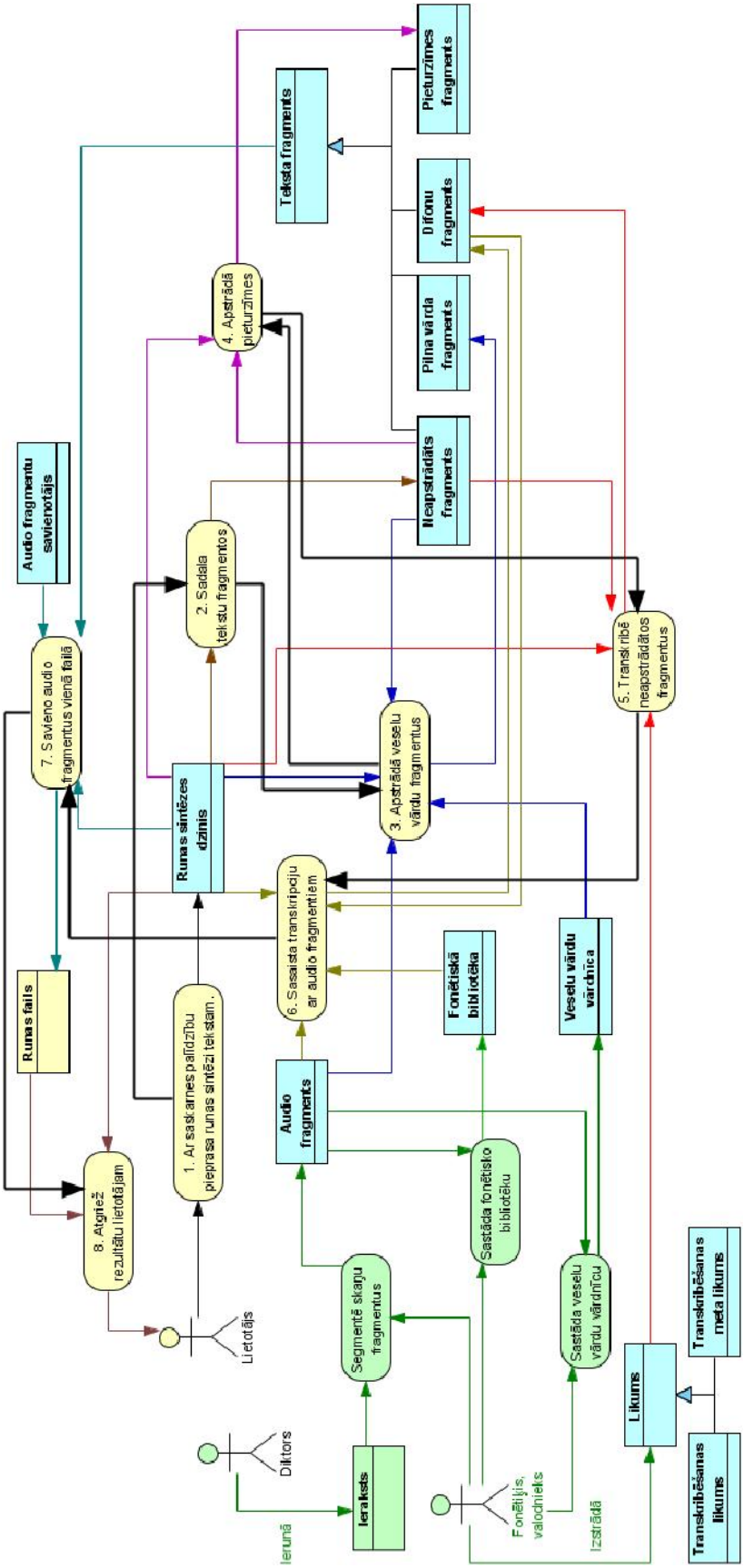
(*Wav Binder*). Audio datu ielādei no failu sistēmas tiek izmantota audio datu lasīšanas komponente (*Wav Reader*). Audio datu savienotāja uzdevums ir izveidot audio failu, līdz ar to tas satur audio datu rakstītāja komponenti (*Wav Writer*).

Atsevišķo sistēmas komponenšu secīga pielietošana teksta apstrādē, rezultātā izveido runu un saglabā to RIFF (*Resource Interchange File Format*) WAVE (*Waveform audio format*) audio failā.

Teksta fragmentu saraksts nodrošina datu uzkrāšanu apstrādes procesā, līdz veiksmīgi izveidots audio fails. Pilno vārdu vārdnīca satur sarakstu ar veseliem vārdiem un to audio reprezentāciju, ieskaitot meklēšanas funkcionalitāti vārdu atrašanai. Transkribēšanas likumu bibliotēka nodrošina latviešu valodas vārdu transkribēšanu fonētiskajā alfabētā. Fonētiskā bibliotēka nodrošina audio datu piekārtošanu atsevišķiem teksta fragmentiem, kas transkribēti fonētiskajā alfabētā. Pēc fonētiskās bibliotēkas izmantošanas, tekstam transkribētajam tekstam atrasta visa nepieciešamā informācija, lai izveidotu sākotnējā teksta runas reprezentāciju.

Svarīgs sistēmas aspekts ir datu bāzes sasaistīšana ar sistēmas komponentēm, bet vispirms datu bāzi ir jāizveido. Par datu bāzes izveidi ir atbildīgi valodnieki un fonētiķi. Sistēmas datu bāzes izveidošanas paredzamie procesi apskatīti 2.2.1. attēlā (iekrāsoti zaļā krāsā). Kā redzams diagrammā, sistēmas teksta apstrādes procesi (dzeltenā krāsā) ir atkarīgi no datiem, kas atrodas datu bāzē. Fonētiķu uzdevums ir segmentēt runas fragmentus, kas iegūti, diktoram ierunājot tekstu ierakstu studijā. Segmentētie fragmenti tiek iekļauti veselu vārdu vārdnīcas un fonētiskās bibliotēkas audio failu datu bāzē. Valodnieku uzdevums ir nodrošināt transkribēšanas likumu izveidošana, lai latviešu valodas tekstu būtu iespējams transkribēt fonētiskajā alfabētā.

2.2.1. attēls konceptuāli parāda arī runas sintēzē iesaistītos procesus. Vispirms lietotājs pieprasa runas sintēzes dzinim runas sintēzi. Lietotāja sintēzei padotais teksts tiek sadalīts fragmentos, pēc tam tiek meklēti un apstrādāti veseli vārdi. Tālāk tiek apstrādāti pieturzīmju fragmenti, pēc kā seko līdz šim neapstrādāto fragmentu transkribēšana fonētiskajā alfabētā un iegūtās transkripcijas segmentēšana mazākos fragmentos, lai piekārtotu skaņu fragmentus no fonētiskās bibliotēkas. Pirmspēdējais process, kas jāizpilda, ir iepriekšējos posmos iegūto datu savienošana, izveidojot rezultāta failu noteiktā adresē failu sistēmā. Kad fails izveidots, lietotājam tiek atgriezts rezultāts tam pieņemamā veidā atkarībā no sistēmas implementācijas veida.

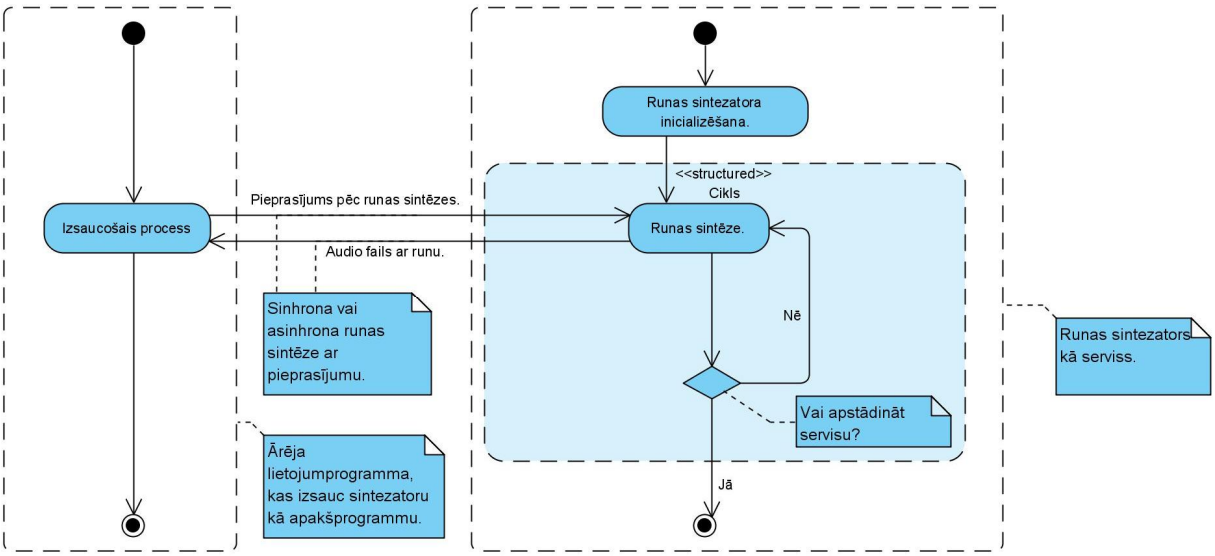


2.2.1. att. **Runas sintēzes sistēmas procesu aktivitāšu diagramma**

Latviešu valodas runas sintēzes sistēmu iespējams integrēt gan lielos, gan mazos risinājumos. Lai nodrošinātu veiksmīgu sistēmas sadarbību ar ārējām lietojumprogrammām, iespējami četri principiāli atšķirīgi runas sintēzes sistēmas realizācijas modeļi.

2.2.1. Servisa realizācija

Latviešu valodas runas sintēzes sistēmas servisa realizācija paredz sistēmas nepārtrauktu darbību, veicot tekstrunas pārveidotāja funkcijas gadījumos, kad tiek saņemts ārējs pieprasījums. Servisa lietojumprogrammas darbību shematiski raksturo attēls 2.2.1.1.

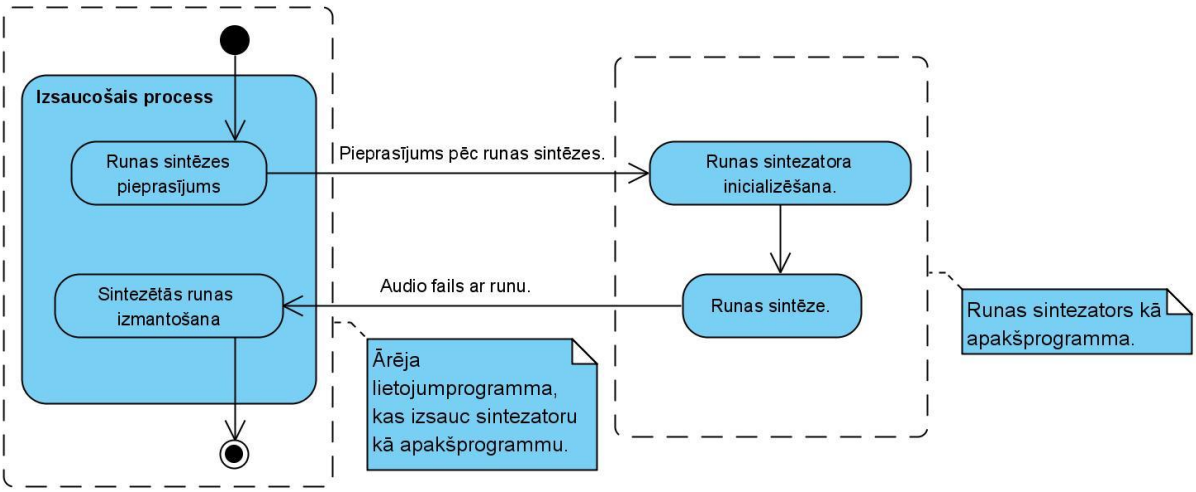


2.2.1.1. att. **Runas sintēzes sistēmas servisa realizācijas modelis**

Kā redzams attēlā, sintezatora serviss tiek ieslēgts citā laikā salīdzinājumā ar lietojumprogrammu, kas izmanto servisa funkcijas. Visbiežāk serviss tiek ieslēgts pirms lietojumprogrammas, lai lietojumprogrammas darbības laikā būtu iespējams izmantot servisa funkcijas. Pretējā gadījumā (serviss tiek ieslēgts pēc lietojumprogrammas) runas sintēze lietojumprogrammai, kurai nepieciešams pārveidot tekstu runā, nebūtu iespējama, jo nebūtu iespējams nodibināt savienojumu ar servisu. Šādas realizācijas gadījumā serviss darbojas kā patstāvīga lietojumprogramma, kura nav saistīta ar programmām, kas izmanto runas sintēzes pakalpojumus. Šāds risinājums ir visizdevīgākais risinājums tādu servera lietojumprogrammu realizācijai, kuru veiksmīgas darbības nodrošināšanai nepieciešama runas sintēze, jo servisa pakalpojumus var izmantot neierobežots skaits lietojumprogrammu, kā arī servisam tikai pie tā ieslēgšanas tiek ielasīta atmiņā datu bāze, kas nepieciešama veiksmīgas funkcionalitātes nodrošināšanai. Vēl viena priekšrocība, izmantojot šādu modeli, ir tāda, ka ir iespējams neierobežots skaits (relatīvi, ņemot vērā sistēmas atmiņas kapacitāti) runas sintēzes pieprasījumu, līdz serviss beidz savu darbību. Serviss beidz darboties tikai gadījumos, kad saņem pavēli par darbības izbeigšanu no ārpuses.

2.2.2. Apakšprogrammas realizācija

Otrs iespējamais latviešu valodas runas sintēzes sistēmas realizācijas modelis (sk. 2.2.2.1. att.) ir veidot sistēmu kā atsevišķu lietojumprogrammu, kas tiek izsaukta katru reizi, kad nepieciešams tekstu pārvērst runā.



2.2.2.1. att. **Runas sintēzes sistēmas kā apakšprogrammas realizācijas modelis**

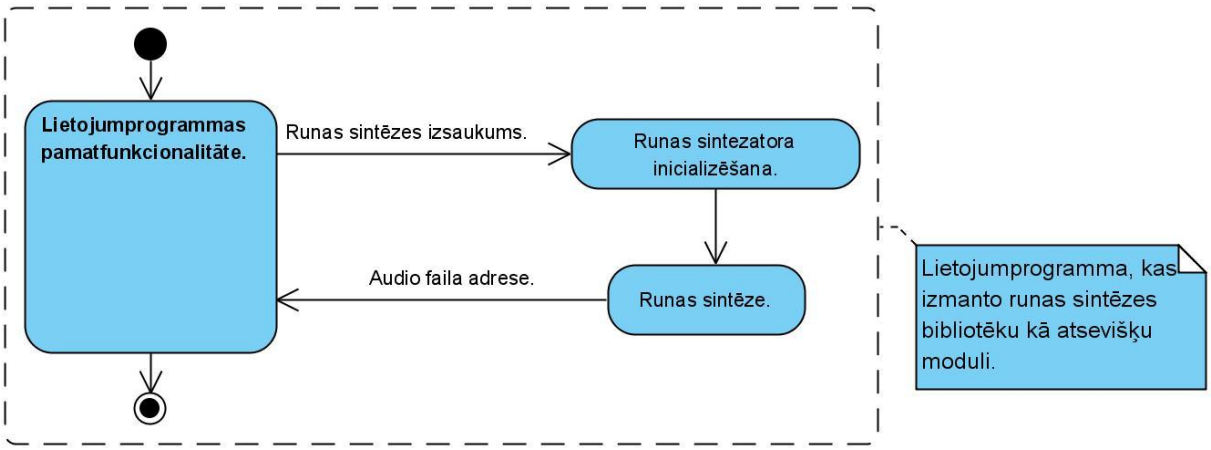
Tas nozīmē, ka gadījumos, kad ārēja lietojumprogramma izsauks runas sintezatoru, patiesībā tiks izsaukta apakšprogramma, kas pārveidos tekstu runā, ņemot vērā parametrus, kurus nodevusi sintezatoram ārējā lietojumprogramma.

Šāds risinājums ir efektīvs, ja runas sintēze nodrošina papildus funkcionalitāti ārējai lietojumprogrammai, bet nav vitāli svarīga tās darbības nodrošināšanai. Kā piemēru var minēt datorvārdnīcas, kurās iespējams noklausīties vārdu izrunu (tā tiek reti lietota un programmu iespējams lietot arī, neklausoties izrunu). Tā kā katra izsaukuma gadījumā tiek veidots jauns process datora atmiņā, kā arī tiek ielasīta visa datu bāze sintezatora pamatfunkciju nodrošināšanai, šāds risinājums nav ieteicams serveru risinājumos un risinājumos, kur lietojumprogrammas darbība ir pilnībā atkarīga no runas sintēzes funkcijām, piemēram, laika ziņu lasītājs, kas neuzrāda tekstu, bet tikai nolasa ziņas. Atgriežoties pie risinājuma efektīvas izmantošanas, šāds modelis ir izdevīgs gadījumos, kad runas sintēze netiek pastāvīgi izmantota, jo lietojumprogramma darbojas un glabā datus atmiņā tikai tik ilgi, cik tiek veikta runas sintēze.

2.2.3. Integrēta moduļa realizācija

Trešais iespējamais latviešu valodas runas sintēzes sistēmas realizācijas modelis (sk. 2.2.3.1. att.), respektīvi, runas sintēzes sistēma, kā integrēts modulis kādā citā lietojumprogrammā, no iepriekšējā modeļa atšķiras ar to, ka veicot izsaukumus netiek veidots

jauns process (gadījumā, ja lietojumprogramma pati neveido jaunus pavedienus), bet sintēzi nodrošina lietojumprogrammas process, kas izsauc runas sintēzi. Līdz ar to tiek nedaudz ietaupīti sistēmas resursi, bet lietojumprogrammas izstrādātājam ir jāpārzina runas sintēzes sistēmas darbības principi, un izstrāde ir limitēta uz vienu un to pašu izstrādes valodu kā runas sintezatoram.



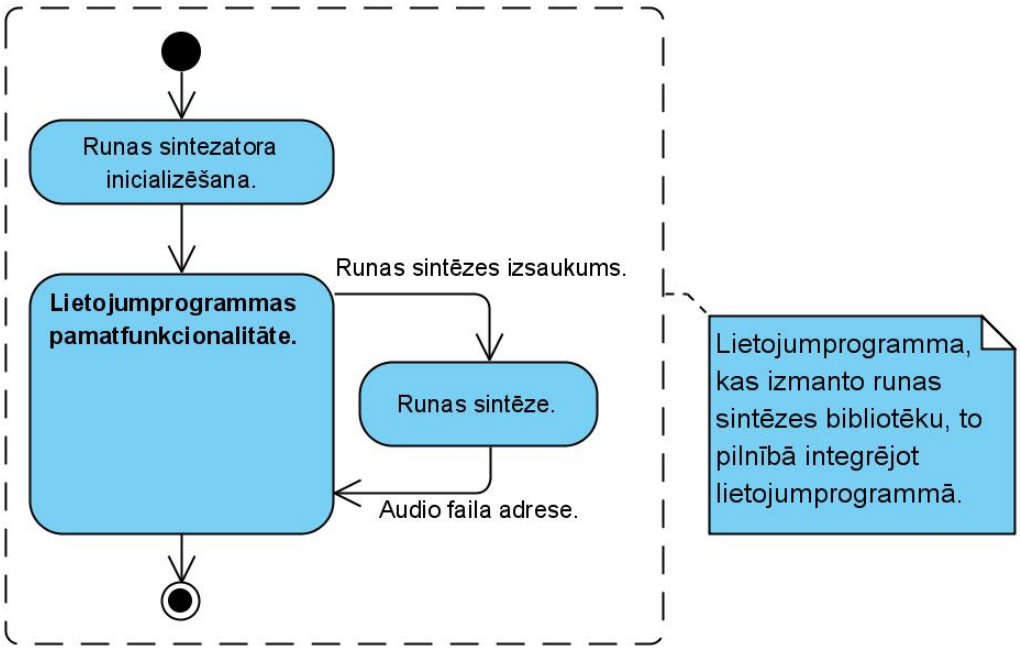
2.2.3.1. att. **Runas sintēzes sistēmas kā integrēta moduļa realizācija**

Katru reizi, kad tiek izsaukta runas sintēzes funkcija lietojumprogrammā, atmiņā tiek ielasīta datu bāze, tiek veikta sintēze, un visbeidzot atmiņa tiek atbrīvota, lai neglabātu liekus datus atmiņā.

Tāpat kā iepriekšējais modelis, arī šis modelis ir efektīvs gadījumos, kad runas sintēze nav vitāli svarīga sistēmas komponente, kā arī runas sintēze tiek veikta atsevišķos gadījumos. Šis realizācijas modelis nav efektīvs gadījumos, kad ir nepieciešama pastāvīga runas sintēze, kā arī, ja runas sintēze nodrošina kāda servera risinājuma darbību.

2.2.4. Integrētas bibliotēkas realizācija

Ceturtais iespējamais latviešu valodas runas sintēzes sistēmas realizācijas modelis (sk. 2.2.4.1. att.) ir runas sintēzes sistēmas bibliotēkas pilnīga integrēšana lietojumprogrammā. Šāda realizācijas modeļa gadījumā runas sintēzes nepieciešamie dati no datu bāzes ir pieejami atmiņā visas konkrētās lietojumprogrammas komponentes darbības laikā. Izsaucot konkrēto lietojumprogrammas komponenti, vai arī, ja lietojumprogramma nav pārāk liela, izsaucot pašu lietojumprogrammu, runas sintēzes sistēma tiek inicializēta, t.i., tiek ielasīta datu bāze pamatfunkciju nodrošināšanai no failu sistēmas. Datu bāze atmiņā glabājas visas lietojumprogrammas vai attiecīgās komponentes darbības laikā.



2.2.4.1. att. **Runas sintēzes sistēmas kā integrētas bibliotēkas realizācija**

Līdz ar to iespējams izsaukt runas sintēzes funkcijas, nepārslogojot aparatūru ar liekām datu ielādes operācijām. Rezultātā iespējams nodrošināt operatīvāku runas sintēzes nodrošināšanu, kā arī iespējams veikt neierobežotu daudzumu secīgus runas sintēzes izsaukumus.

Izmantojot runas sintēzes bibliotēku, lietojumprogrammas izstrādātājam ir jāpārzina bibliotēkas darbības principi tik lielā mērā, lai spētu nodrošināt sava risinājuma veiktspēju un kvalitāti. Viens no ierobežojošajiem faktoriem, izvēloties šādu risinājuma modeli, ir tāds, ka izstrādātājam ir jāizstrādā lietojumprogramma ar tādu pašu izstrādes valodu, kāda lietota latviešu valodas runas sintēzes bibliotēkas izstrādē.

Pretstatot ierobežojumiem un prasībām ieguvumus, šāds realizācijas modelis ir pietiekami efektīvs gan risinājumiem, kas paredzēti darbībai uz serveriem, kā arī risinājumiem, kuri paraudzēti atsevišķu lietotāju darbstacijām gadījumos, kad runas sintēze ir pastāvīgi nepieciešama lietojumprogrammas pamatfunkciju nodrošināšanai. Konkrētais risinājums šādos gadījumos ir efektīvs, jo datu bāzes ielāde atmiņā notiek tikai vienu reizi un runas sintēzes funkcijas pilda pati lietojumprogramma, neveicot izsaukumus ārpus lietojumprogrammas. Šādi iespējams efektīvāk kontrolēt arī kļūdu situācijas, jo ir pieejama informācija par katru kļūdu, kas atgadījusies sintēzes procesā, un ir iespējams attiecīgi reaģēt vai paziņot lietotājam par iemesliem, kāpēc sintēze noritējusi neveiksmīgi.

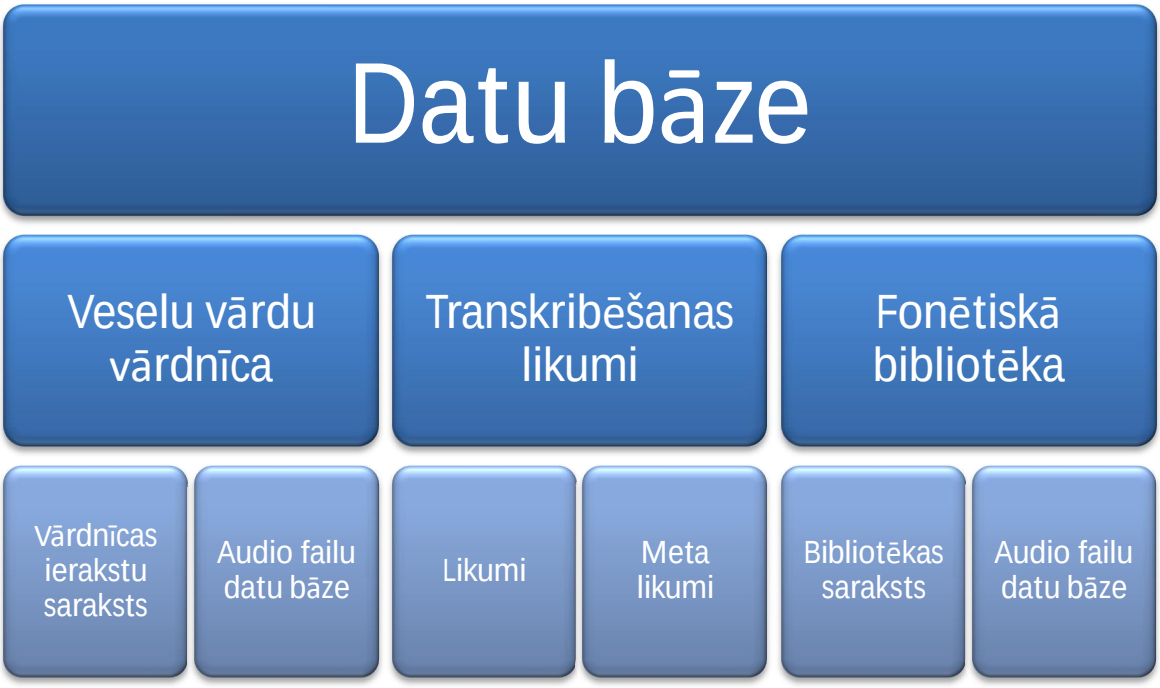
3. RUNAS SINTĒZES SISTĒMAS ARHITEKTŪRA

Iepriekšējā nodaļā tika apskatīta konkatenatīvas runas sintēzes sistēmas koncepcija, t. i., vispārīgi tika aprakstīts, kādu informācijas sistēmu nepieciešams izveidot. Šajā nodaļā tiks apskatīta bakalaura darba autora izstrādātās konkatenatīvās latviešu valodas runas sintēzes sistēmas arhitektūra. Vispirms tiks pievērsta uzmanība sistēmas datu bāzei un pēc tam tiks aprakstīta sistēmas klašu struktūra, ieskaitot galvenos algoritmus, kas nodrošina teksta apstrādi un runas akustiskā signāla izveidi.

3.1. Datu bāzes arhitektūra

Tā kā tika izvēlēta konkatenatīvas runas sintēzes sistēmas realizācija, izmantojot pirms sintēzes izveidotus runas fragmentus, sistēmai nepieciešams prast efektīvi apstrādāt datu bāzi. Lai labāk izprastu sistēmas uzbūvi, vispirms tiks apskatīta datu bāze un tās uzbūves semantika runas sintēzes procesu nodrošināšanā.

Konkatenatīvās latviešu valodas runas sintēzes sistēmas datu bāzi veido sekojošas komponentes (sk. 3.1.1. att.): pilno vārdu vārdnīca, transkripcijas likumi un fonētiskā bibliotēka.

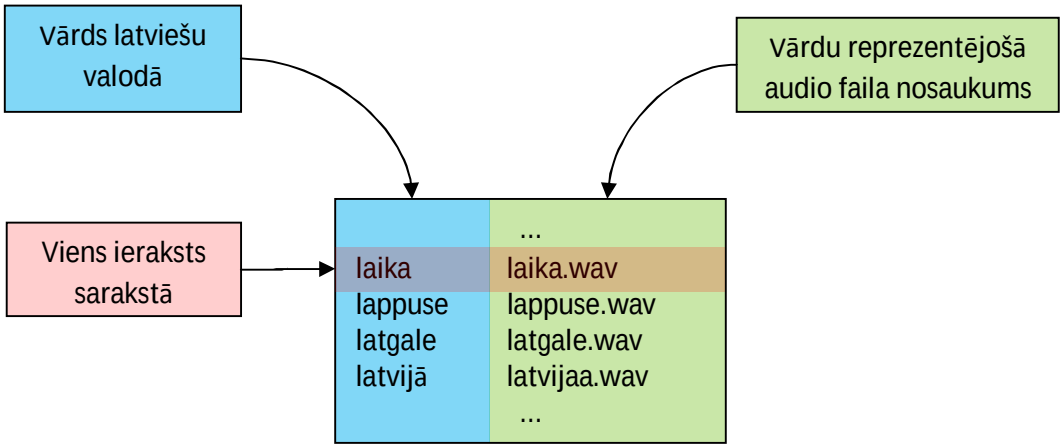


3.1.1. att. **Sistēmas datu bāzes struktūra**

Katra no datu bāzes komponentēm tiek izmantota kādā posmā augstā līmeņa runas sintēzē (sk. nodaļu 1.2.), t. i., teksta apstrādē, lai nodrošinātu pareizu audio fragmentu piekārtošanu attiecīgiem teksta fragmentiem. Tālāk tiks apskatītas detalizētāk atsevišķās datu bāzes komponentes.

3.1.1. Veselu vārdu vārdnīca

Veselu vārdu vārdnīca sastāv no divām daļām. Pirmo daļu veido audio failu datu bāze, kur katrs audio fails satur viena latviešu valodas vārda vai vārdformas akustisko reprezentāciju. Lai nodrošinātu meklēšanas funkcionalitāti pa failu nosaukumiem, ir nepieciešams saraksts, kas sasaista vārdus ar to reprezentējošajiem audio failiem, lai sistēma efektīvi spētu noteikt, vai konkrētam vārdam atbilst kāds audio fails no veselu vārdu vārdnīcas. Saraksts ir realizēts kā teksta fails, kurā vienam ierakstam atbilst simbolu virkņu pāris, t. i., vārds vai vārdforma latviešu valodā un audio faila nosaukums. Saraksta struktūru precīzāk raksturo 3.1.1.1. attēls.



3.1.1.1. att. **Veselu vārdu vārdnīcas ierakstu saraksta struktūra**

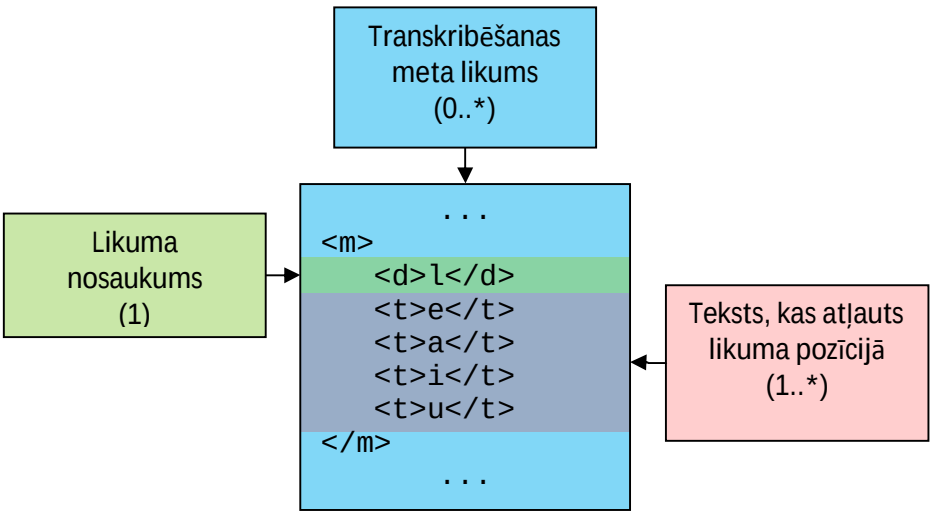
Visi latviešu valodas vārdi sarakstā ir definēti ar mazajiem burtiem, jo, uzsākot teksta apstrādi, sistēma visus lielos burtus apstrādājamajā tekstā aizvietos ar mazajiem burtiem. Šāda teksta apstrādes loģika izvēlēta, lai nodrošinātu ātrāku salīdzināšanas operāciju darbību. Līdz ar to tiek nodrošināta arī burtu lielumu neatkarīga teksta apstrāde.

Saraksts datu bāzē tiek glabāts resursu direktorijā ar nosaukumu „*fwd.txt*”, bet veselo vārdu datu bāze atrodas resursu direktorijas apakšdirektorijā ar nosaukumu „*FullWords*”. Veselu vārdu vārdnīcu var apskatīt pirmajā pielikumā.

3.1.2. Transkribēšanas likumi

Transkribēšanas likumi sastāv no divām daļām [2]. Pirmo daļu veido vispārēji transkribēšanas likumi, kuros definēta loģika teksta pārveidošanai no latviešu alfabēta uz fonētisko alfabētu. Otro daļu veido transkribēšanas meta likumi, kas tiek izmantoti vispārējos likumos, lai norādītu teksta fragmentu grupas, uz kurām var attiekties konkrēts likums. Apskatīsim likumu struktūru, lai precīzāk izprastu saistību starp abiem likumu veidiem.

Likumi datu bāzē tiek realizēti ar XML (*Extensible Markup Language*) palīdzību [9]. Meta likumu struktūru precīzi raksturo 3.1.2.1. attēls.



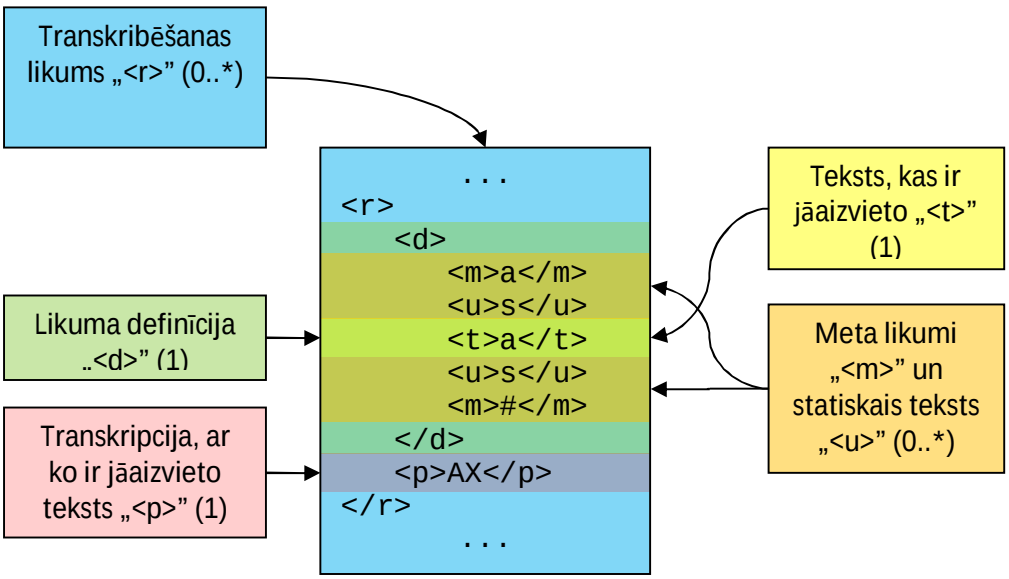
3.1.2.1. att. **Transkribēšanas meta likumu struktūra**

Katru meta likumu veido viena „m" birka. Visas „m" birkas kopā veido transkribēšanas meta likumu datu bāzi. Katrai „m" birkai obligāta prasība ir viens bērna elements „d", kas apzīmē konkrētā meta likuma nosaukumu, kā arī viens vai vairāki bērna elementi „t". Elementi „t" meta likumos definē teksta fragmentus, kas drīkst atrasties transkribējamā teksta pozīcijā, kur tiek pielietots konkrētais meta likums. Līdz ar to semantiski viens meta likums definē teksta fragmentu kopu, kas drīkst atrasties teksta attiecīgajā pozīcijā pirms vai pēc transkribēšanas likuma teksta, kas tiks aizstāts ar transkripciju. Attēlā 3.1.2.1. definētais meta likums nosaka, ka transkribējamajā tekstā pozīcijā, kur tiks pārbaudīta meta likuma izpildīšanās, drīkst atrasties četri teksta fragmenti („e", „a", „i" un „u"). Ja šādi teksta fragmenti tekstā nav sastopami, meta likums neizpildās un transkribēšanas likums, kurā tiek izmantots meta likums, nav derīgs konkrētajā teksta pozīcijā.

Meta likumi tiek izmantoti transkribēšanas likumos, lai definētu specifiskus nosacījumus teksta fragmenta, kas jāaizvieto ar likuma transkripciju, apkārtnei. Transkribēšanas likumu struktūru precīzi raksturo 3.1.2.2. attēls.

Katru transkribēšanas likumu veido viena „r" birka. Visas „r" birkas kopā veido transkribēšanas vispārējo likumu datu bāzi. Katrai „r" birkai obligāta prasība ir divi bērnu elementi: „d" un „p". Birka „d" apraksta likuma definīciju, bet birka „p" – transkripcijas fragmentu.

Likuma definīciju veido teksta fragmenta elements „t", kura iekšējā vērtība tiks aizstāta ar transkripcijas fragmentu no birkas „p". Pirms vai pēc teksta fragmenta drīkst atrasties meta likumi „m" (elements satur konkrēta meta likuma nosaukumu) un statisks teksts „u" (konkrēts teksta fragments, kam obligāti jābūt pirms vai pēc likuma aprakstošā teksta fragmenta). Definīcijas bērnu elementos drīkst būt neierobežots skaits meta likumu elementi un statiska teksta elementi.



3.1.2.2. att. **Transkribēšanas likumu struktūra**

Transkribēšanas likumos ir definēti divu veidu meta likumu elementi iekš vienas birkas. Viens meta likumu veids nosaka skaitu, cik burtu drīkst atrasties konkrētā meta likuma pozīcijā, bet otrs veids nosaka, kādas burtu kombinācijas drīkst atrasties konkrētā meta likuma pozīcijā (šis veids sasaista iepriekš aprakstītos transkribēšanas meta likumus ar transkribēšanas likumiem). To, kāda veida meta likums izmantots, nosaka pēc birkas iekšējās vērtības. Kopā ir četri pirmā veida meta likumi [2]:

„?” – meta likuma pozīcijā jābūt vienam burtam;

„*” – meta likuma pozīcijā drīkst būt jebkāds daudzums burtu;

„#” – meta likuma pozīcijā nedrīkst būt neviens burts (jābūt sasniegtam vārda sākumam vai beigām);

„^” – meta likuma pozīcijā jābūt vismaz vienam burtam.

Viens meta likums transkribēšanas likumam piešķir vispārēju nozīmi, jo viens likums līdz ar to var atbilst vairākiem teksta fragmentiem. Piemēram, meta likums, kas nosaka, ka teksta konkrētajā pozīcijā var atrasties nulle vai vairāk elementu ļauj likumu pielietot, neņemot vērā teksta fragmenta apkārtni, sākot ar konkrēto meta likumu. Pieņemsim, ka ir dots likums „<r><d><m>*</m><t>a<m>*</m></d> A </r>”. Šis likums izpildās, piemēram, vārdiem „mamma” un „galva”. Abiem vārdiem visi burti „a” tiks aizstāti ar fonētiskā alfabēta burtu „A”, kas definēts likuma transkripcijas fragmenta birkā „p”, neņemot vērā, kas par burtiem atrodami apkārt (pa kreisi un pa labi). Bet, ja šiem pašiem vārdiem tiktu pielietots likums „<r><d><m>*</m><t>a<m>#</m></d> A </r>”, tad tiktu aizstāti tikai pēdējie burti „a”, neaiztiekot tos, kas atrodas vārdu vidū vai sākumā. Līdz ar to ir saprotams, ka likumi nav vienlīdz vispārīgi. Likuma vispārības mērs atkarīgs no tā, cik daudz

meta likumi tajā ir izmantoti, cik garu teksta fragmentu tas aizstāj ar transkripciju, kā arī, vai tajā ir izmantoti statiska teksta fragmenti.

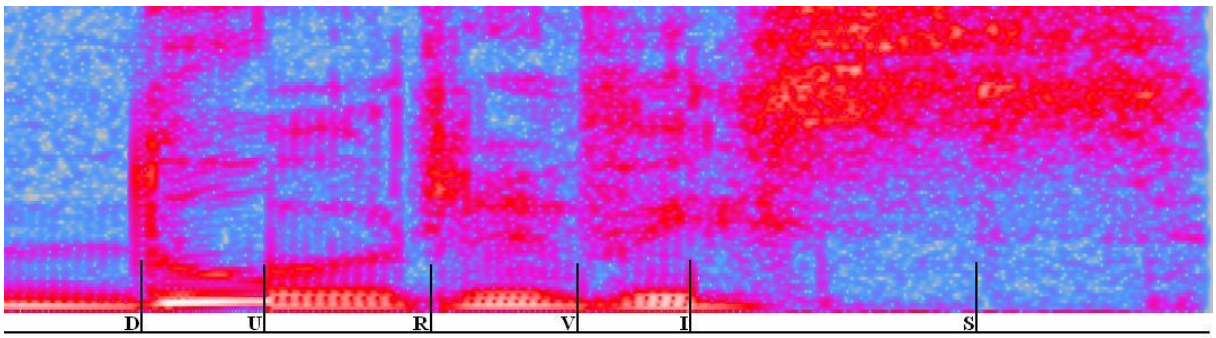
Tā kā vispārīgāki likumi izpildās arī teksta fragmentiem, kuriem izpildās ne tik vispārīgi likumi, ir svarīgi vispārīgākos likumus pielietot tikai tad, ja specifiskie likumi nav izpildījušies. Šāda likumu pielietošanas secība nodrošina, ka visiem teksta fragmentiem jebkurā gadījumā būs atrodams vismaz viens likums, kas būs pielietojams, jo datu bāzē katram latviešu valodas burtam būs atrodams viens likums, kas būs pats vispārīgākais (piemēram, „<r><d><m>*</m><t>a<m>*</m></d> A </r>”). Tiek paredzēts / nodrošināts, ka specifiskas izņēmumu situācijas tiek apstrādātas vispirms, tādējādi teksta fragmentam piekārtojot pareizākus transkripcijas fragmentus.

Transkribēšanas likumi un meta likumi datu bāzē tiek glabāti divos XML failos resursu direktorijā ar attiecīgiem nosaukumiem „*rules.xml*” un „*metas.xml*”. Transkribēšanas likumu datu bāze pieejama pirmajā pielikumā.

3.1.3. Fonētiskā bibliotēka

Fonētiskā bibliotēka, tā pat kā veselu vārdu vārdnīca (sk. nodaļu 3.1.1.), sastāv no divām daļām. Pirmo daļu veido audio failu datu bāze, kur katrs fails satur kādu runas fragmentu. Atšķirībā no veselu vārdu vārdnīcas, fonētiskā bibliotēka nesatur veselus vārdus, bet gan skaņu fragmentus, kurus kombinējot tiks izveidots nepieciešamais vārds.

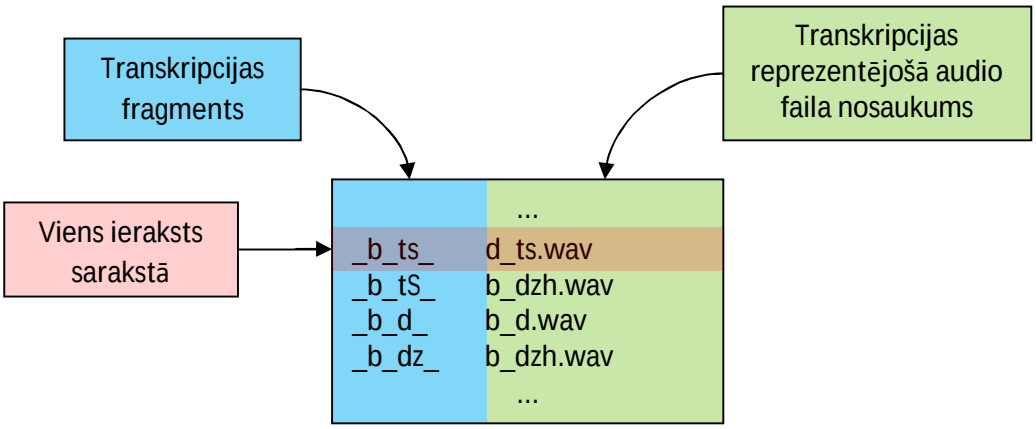
Audio failu datu bāze pārsvarā sastāv no skaņu fragmentiem, kas veido pārejas no vienas fonēmas uz otru – difonus. Difoni ir skaņu fragmenti, kas satur tikai pāreju no vienas fonēmas centra līdz otras fonēmas centram, šādi nodrošinot iespēju fragmentus savienot daudz gludāk, kā savienojot atsevišķas fonēmas. Gadījumā, ja tiktu savienotas atsevišķas fonēmas, rastos nepieciešamība veidot mākslīgas fonēmu pārejas, lai runa būtu uztverama un starp skaņām nebūtu izteikti pārtraukumi (klusumi un lieki trokšņi) [1] Sintezēta vārda spektrogrammā (sk. 3.1.3.1. attēlu) viegli pamanāmas konkatenācijas vietas. Attēlā vertikālās līnijas norāda fonēmu centrus, kuros veikti griezumi. Ja saskaita vertikālās līnijas, var secināt, ka vārds „durvis” savienots no septiņiem audio fragmentiem, no kuriem pieci fragmenti ir satur fonēmu pārejas („d-u”, „u-r”, „r-v”, „v-i”, „i-s”), bet divi fragmenti satur fonēmu puses, respektīvi fonēmas „d” sākuma daļu un fonēmas „s” beigu daļu.



3.1.3.1. att. **Vārda „durvis” spektrogramma (iegūta savienojot difonu fragmentus)**

Neskaitot difonu fragmentus, fonētiskā bibliotēka satur arī garākus fragmentus, kuros iekļautas pilnas fonēmas, veidojot garākas fragmentu konstrukcijas. Garāki fragmenti nepieciešami, ja no difonu fragmentiem nav iespējams izveidot pietiekami gludu (uztveramu, bet liekiem trokšņiem) runu.

Fonētiskās bibliotēkas otru daļu veido saraksts, kas sasaista audio failu datu bāzē esošos failus ar tiem atbilstošajām transkripcijām, lai būtu iespējams vārdam piekārtot visus nepieciešamos audio fragmentus. Saraksta struktūru precīzāk raksturo 3.1.3.2. attēls.



3.1.3.2. att. **Fonētiskās bibliotēkas ierakstu saraksta struktūra**

Saraksts ir veidots kā teksta dokuments, kurā vienu ierakstu definē divu elementu pāris. Pirmais elements ir transkripcijas fragments, bet otrs elements ir audio faila nosaukums. Audio failā glabājas transkripcijas akustiskā reprezentācija.

Transkripcijā katra fonēma ir atdalīta ar apakš svītras simbolu „_”. Ja transkripcijas fragments nesākas ar simbolu „_”,transkripcijas audio fragments var tikt lietots tikai vārda sākumā, Ja transkripcijas fragments nebeidzas ar simbolu „_”,transkripcijas audio fragments var tikt lietots tikai vārda beigās.

Fonētiskās bibliotēkas audio failu datu bāze tiek glabāta resursu direktorijas apakšdirektorijā „*Diphones*”, bet saraksta fails glabājas resursu direktorijas failā „PhoneticLibrary.txt”. Fonētiskās bibliotēkas datu bāze pieejama pirmajā pielikumā.

Fonētiskā bibliotēka tiek izmantota, lai piekārotu transkribētam tekstam audio fragmentus.

3.2. Klašu struktūra un runas sintēzes algoritmi

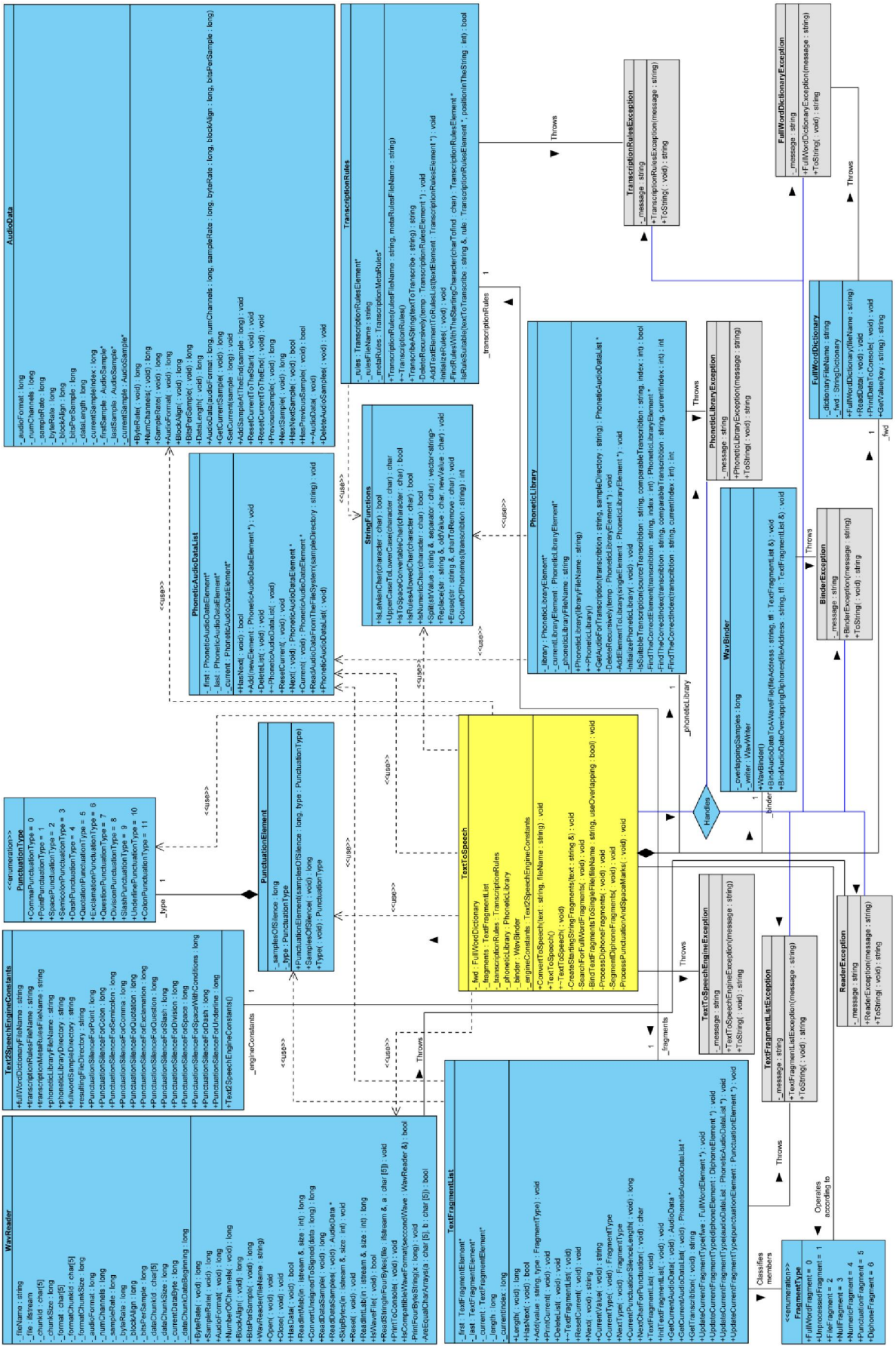
Konkatenatīvas latviešu valodas runas sintēzes sistēma izstrādāta, lietojot objektorientētas programmēšanas principus [7, 10], līdz ar to, lai izprastu sistēmas uzbūvi, ļoti svarīga ir sistēmas klašu struktūra. Sistēmai var izšķirt sešus moduļus, no kuriem viens modulis sasaista kopā visus pārējos moduļus un nodrošina sistēmas kopējo funkcionalitāti, t. i., runas sintēzi. Tālāk tiks apskatīts katrs modulis atsevišķi, sākot ar moduli, kas saista kopā pārējos.

3.2.1. Tekstrunas pārveidotāja dzinis

Tekstrunas pārveidotāja dzinis nodrošina runas sintēzes funkcionalitāti, izmantojot pārējos sistēmas moduļus, kas tiks aprakstīti nākošajās nodaļās, kā arī ir galvenais sistēmas modulis. Klašu diagramma (sk. 3.2.1.1. attēlu) precīzi parāda dziņa klases „*Text2Speech*" metodes un atribūtus, kā arī izmantotās klases nepieciešamās funkcionalitātes nodrošināšanai.

Diagramma neatspoguļo patiesos sistēmas apjomus, bet gan asociācijas saistībā ar dziņa klasi. Pārējās asociācijas, kā arī sistēmas klašu diagramma kopumā tiks apskatīta nedaudz vēlāk.

Runas sintēzes dzinis sastāv no sekojošiem atribūtiem: veselu vārdu vārdnīcas (*FullWordDictionary* - *_fwd*), teksta fragmentu saraksta (*TextFragmentList* - *_fragments*), transkripcijas likumiem (*TranscriptionRules* - *_transcriptionRules*), fonētiskās bibliotēkas (*PhoneticLibrary* - *_phoneticLibrary*), audio datu savienotāja (*WavBinder* - *_binder*) un runas sintēzes dziņa konstantēm (*Text2SpeechEngineConstants* - *_engineConstants*). Inicializējot dzini, tiek inicializēti arī visi dziņa atribūti, līdz ar to dziņa inicializācijas procesā tiek ielasīti operatīvajā atmiņā visi nepieciešamie dati teksta apstrādei. Audio failu datu bāze netiek ielasīta atmiņā, jo tas samazinātu veiktspēju un lielākā daļa audio fragmentu runas sintēzes izsaukumos netiks izmantoti, līdz ar to ielasīšana atmiņā būtu nelietderīga.

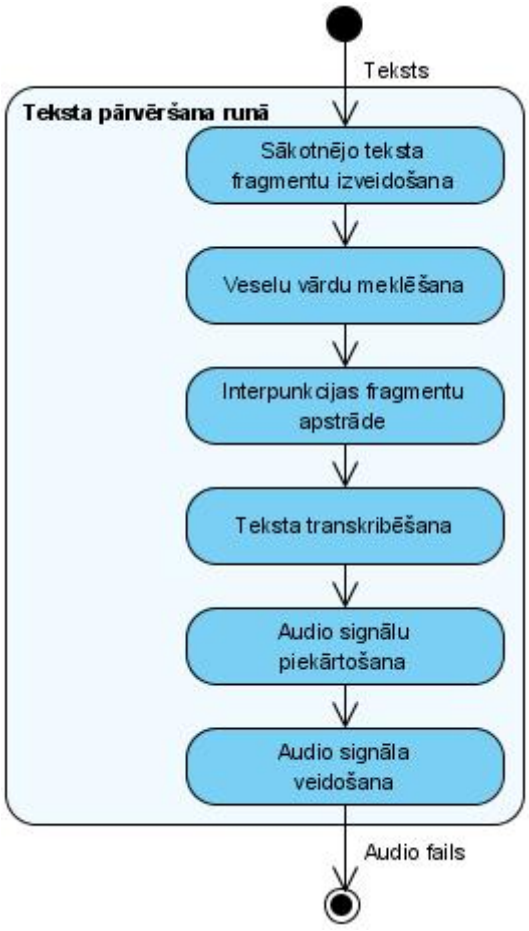


3.2.1.1. att. **Konkatenatīvas latviešu valodas runas sintēzes dziņa klašu diagramma**

Dziņa inicializēšana tiek veikta izsaucot konstruktoru „*TextToSpeech*”. Konstruktors, inicializējot atribūtus, kā parametrus to konstruktoriem, padod konstantes no runas sintēzes dziņa konstantēm. Konstantes ir vienīgie atribūti, kas netiek specifiski inicializētas, jo tām nav nepieciešama informācijas saņemšana no ārpuses.

Tie dati, kas tiek ielasīti atmiņā dziņa inicializācijas procesā, tiek izdzēsti, izsaucot runas sintēzes dziņa destruktoru „*~TextToSpeech*”.

Runas sintēzes dzinis piedāvā vienu publisku metodi, kas nodrošina teksta sintēzi – „*ConvertToSpeech*”. Metode tiek izsaukta ar diviem parametriem: sintezējamo tekstu latviešu valodā un adresi, kura norāda uz vietu, kur jāsaglabā sintezētā runa. Metodes algoritmu un teksta apstrādes posmus, kādi ietilpst runas sintēzes procesā uzskatāmi parāda aktivitāšu diagramma 3.2.1.2. attēlā.



3.2.1.2. att. **Runas sintēzes process**

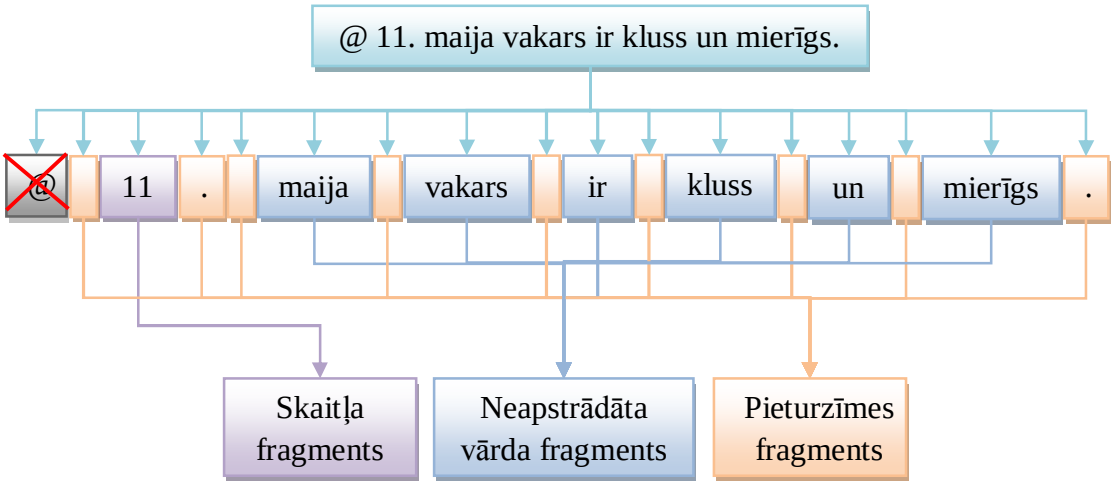
3.2.1.1. Teksta sadalīšana fragmentos

Teksts, kas tiek padots runas sintēzei vispirms tiek sadalīts fragmentos. Teksta sadalīšanu fragmentos veic metode „*CreateStartingStringFragments*”. Ir iespējami trīs sākotnējie teksta fragmentu veidi (*FragmentType*): neapstrādātu vārdu fragmenti (*UnprocessedFragment*), skaitļu fragmenti (*NumericFragment*) un pieturzīmju (ieskaitot

atstarpes) fragmenti (*PunctuationFragment*). Šādi fragmenti nepieciešami, jo to apstrāde atšķiras no pārējo fragmentu apstrādes, piemēram, pieturzīmes un atstarpes norāda uz pauzēm runā, kas patiesībā ir klusuma intervāli, līdz ar to pieturzīmju fragmentiem nepieciešams noteikt tikai pareizo klusuma intervālu, ko ieturēt starp citiem fragmentiem.

Metodes izveidotie fragmenti tiek saglabāti teksta fragmentu sarakstā „_fwd". Šajā posmā no teksta tiek izmesti visi simboli, kas neatbilst latviešu valodai („X", „Y", „$", „@" utt.), kā arī visi lielie burti („A", „Ā", „B", „C" utt.) tiek pārvērsti par mazajiem burtiem, tādējādi nodrošinot burtu lieluma neatkarīgu teksta apstrādi. Šāda burtu lieluma vienādošana ir svarīga, jo sistēma balstās uz teksta salīdzināšanas operācijām, un būtu neefektīvi izstrādāt algoritmus dažādu burtu lielumu apstrādei, ja ir iespējams vienkāršāks un ātrāks (izpildes laikā) risinājums. Manipulācijas ar tekstu (burtu izmešana, burtu lieluma maiņa) nodrošina klase „*StringFunctions*".

Teksta sadalīšanu fragmentos uzskatāmāk var redzēt attēlā 3.2.1.1.1. Redzams, ka, apstrādājot tekstu, zīme „@" netika iekļauta teksta fragmentu sarakstā.



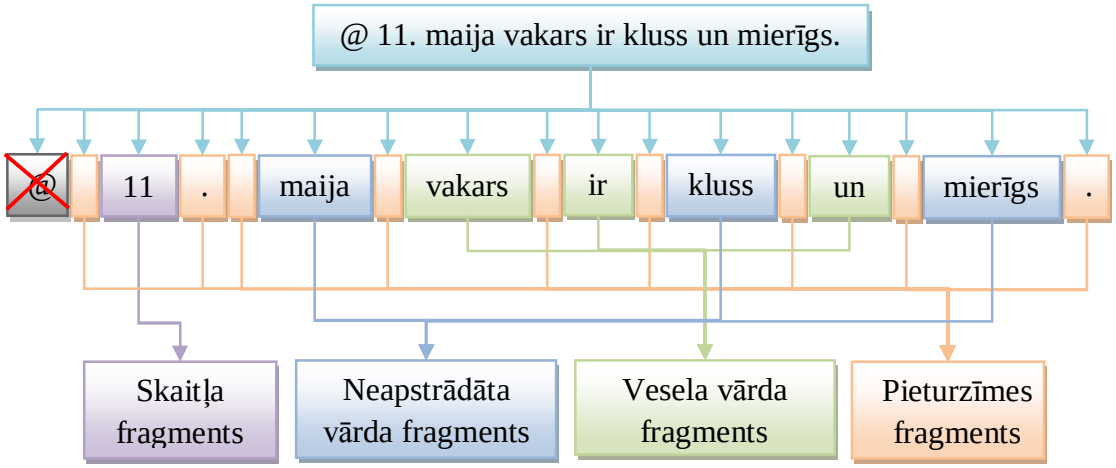
3.2.1.1.1. att. **Sākotnējo teksta fragmentu izveidošanas piemērs**

3.2.1.2. Veselu vārdu meklēšana

Nākošais teksta apstrādes posms pēc teksta fragmentu izveidošanas ir veselu vārdu meklēšana. Šim mērķim teksta sintēzes dzinim izveidota metode „*SearchForFullWordFragments*", pārbauda visus neapstrādāta vārda/teksta fragmentus un noskaidro, vai tie sastopami veselu vārdu vārdnīcā. Ja vārds ir sastopams veselu vārdu vārdnīcā, tad konkrētais teksta fragments tiek pārveidots no neapstrādāta vārda fragmenta par vesela vārda fragmentu (*FullWordFragment*). Fragments tiek papildināts ar vesela vārda elementu (*FullWordElement*), kas satur vārda audio reprezentāciju, kas tiek ielasīta no veselu vārdu vārdnīcas audio failu datu bāzes (plašāk par datu bāzi izskaidrots nodaļā 3.1.1.) ar audio

failu lasītāja (*WavReader*) palīdzību, saglabājot datus operatīvajā atmiņā audio datu objekta (*AudioData*) veidā.

Fragmentiem, kuriem netiek atrasti atbilstoši fragmenti no veselu vārdu vārdnīcas, statuss nemainās, un tie tiek atstāti par neapstrādāta vārda fragmentiem. Savukārt vārdiem, kuriem tika atrasts atbilstošs fragments, statuss tiek nomainīts un tie vairs netiks apstrādāti līdz brīdim, kad visi audio dati tiks vienoti kopā. Iepriekšējā apakšnodaļā apskatītā teksta fragmentu statusi pēc veselu vārdu meklēšanas apskatāmi 3.2.1.2.1. attēlā.



3.2.1.2.1. att. **Veselu vārdu meklēšanas piemērs**

Attēlā redzams, ka trīs neapstrādāta teksta fragmenti tiek pārveidoti par vesela vārda fragmentiem, bet pārējie trīs teksta fragmenti palikuši neapstrādāti, jo veselu vārdu vārdnīcā tiem netika atrasti atbilstoši audio fragmenti.

3.2.1.3. Interpunkcijas fragmentu apstrāde

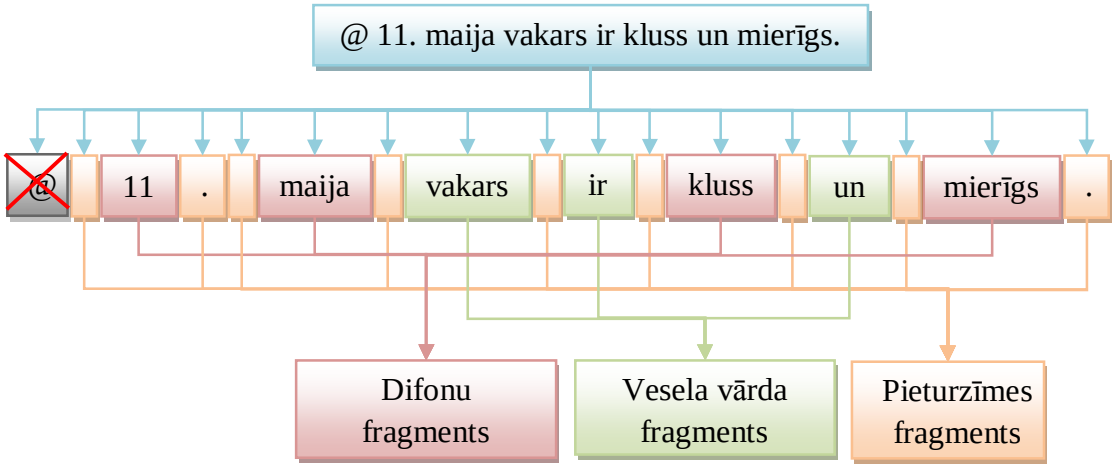
Pieturzīmju fragmentu apstrādi veic metode „*ProcessPunctuationAndSpaceMarks*" , un tajā tiek ņemti vērā tikai pieturzīmju fragmenti. Fragments tiek papildināts ar pieturzīmes elementu (PunctuationElement), kas satur pieturzīmes tipu (*PunctuationType*), kas norāda uz to, kāda veida pieturzīme izmantota. Elements satur arī informāciju par to, cik garš klusuma periods runā ir jāievēro konkrētās pieturzīmes pozīcijā. Piemēram, ja tiek apstrādāts teksta fragments, kas satur punktu, tam tiks piešķirts punkta pieturzīmes tips (*PointPunctuationType*) un klusuma perioda garums, kas vienāds ar 8 820 klusuma punktiem audio signālā pie nosacījuma, ka audio signāls tiek atskaņots pie 22 050 datu punktiem sekundē (*sample rate*).

3.2.1.4. Teksta transkribēšana

Teksta transkribēšanu nodrošina metode „*ProcessDiphoneFragments*". Šī metode apstrādā visus līdz šim neapstrādātos teksta fragmentus un skaitļu fragmentus, piekārtojot

visiem fragmentiem transkripcijas. Teksta transkribēšanu nodrošina transkribēšanas likumu klase (*TranscriptionRules*), izmantojot dziņa atribūtu „_transcriptionRules”.

Visi neapstrādāta vārda fragmenti un skaitļu fragmenti šajā posmā tiek pārveidoti par difonu fragmentiem (*DiphoneFragment*) un tie tiek papildināti ar difonu elementiem (*DiphoneElement*), kas šī posma ietvaros teksta fragmentu papildina ar tā transkripciju fonētiskajā alfabētā. Iepriekšējās apakšnodaļās apskatītā teksta fragmentu statusi pēc teksta transkribēšanas apskatāmi 3.2.1.4.1. attēlā.



3.2.1.4.1. att. **Teksta transkribēšanas piemērs**

Attēlā redzams, ka vairs nav palicis pāri neviens neapstrādāta teksta fragments, kā arī visi skaitļu fragmenti ir pārveidoti par difonu fragmentiem. Attēlā uzrādīts teksta transkribēšanas galvenais rezultāts, t. i., transkripcija. Difonu fragmentiem no piemēra tika piekārtotas šādas transkripcijas:

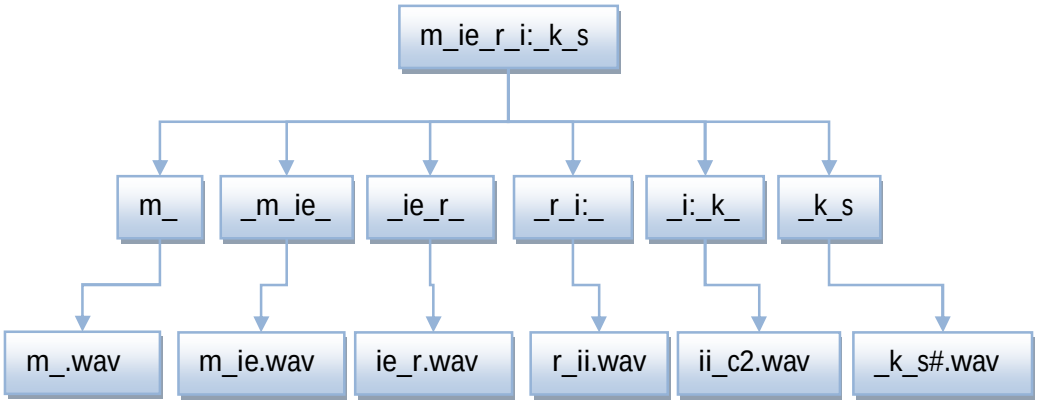
- 11 - v_ie_n_p_A_ts_m_i_t
- maija - m_A_i_j_AX
- kluss - k_l_u_s_s
- mierīgs - m_ie_r_i:_k_s

Skaitļu transkribēšanas procesā tiek transkribēta skaitļa pamatforma, jo, lai noteiktu locījumu, nepieciešama visa teksta semantiskā analīze, kas uz doto brīdi nav iespējama. Papildus, šim ierobežojumam skaitļi, kas garāki par diviem cipariem tiek transkribēti kā ciparu kombinācijas, nevis veseli skaitļi. Piemēram, skaitlis 123 netiks transkribēts kā teksts: „Simt divdesmit trīs.”. Tā vietā teksts tiks transkribēts sekojoši: „Viens, divi, trīs.” Lai minētos ierobežojumus atceltu, jāveido atsevišķs apstrādes modulis, kas pirms teksta sākotnējo fragmentu izveidošanas veiktu teksta semantisko analīzi un izvērstu skaitļus vārdiskā pierakstā.

3.2.1.5. Audio signālu piekārtošana

Audio signālu piekārtošanu nodrošina metode „*SegmentDiphoneFragments*”. Tā kā veselu vārdu fragmentiem jau ir piekārtoti atbilstoši audio fragmenti un pieturzīmju fragmentiem nav nepieciešami audio fragmenti, tad audio datus nepieciešams piekārtot tikai difonu fragmentiem, t. i., fragmentiem ar transkripcijām.

Audio signālu piekārtošana tiek veikta ar fonētiskās bibliotēkas klases (*PhoneticLibrary*) palīdzību, izmantojot runas sintēzes sistēmas dziņa atribūtu „*_phoneticLibrary*”. Kā rezultātu fonētiskā bibliotēka atgriež transkripcijas segmentāciju vairākās daļās, kur katram transkripcijas fragmentam atbilst audio fragments. Segmentētā transkripcija ar tās reprezentējošiem audio fragmentiem glabājas fonētisko audio datu sarakstā (*PhoneticAudioDataList*). Segmentācijas piemērs vārdam „mierīgs” parādīts . attēlā.



3.2.1.5.1. att. **Transkripcijas segmentēšanas piemērs vārdam „mierīgs”**

Vārds „mierīgs”, segmentējot transkripciju, sadalīts sešos fragmentos. Pirmais fragments satur pusfonēmu, nākošie četri fragmenti satur difonus, bet pēdējais fragments satur vienu pilnu fonēmu un vienu pusfonēmu.

3.2.1.6. Audio signāla veidošana

Pēdējais posms runas sintēzes procesā konkatenatīvas latviešu valodas runas sintēzes sistēmā ir audio signāla izveidošanas posms. Šajā posmā visi audio dati, kas ielasīti atmiņā iepriekšējos posmos, tiek savienoti kopā, izveidojot rezultātu, kas ir sākotnējā teksta reprezentācija runā. Runas sintēzes dzinī audio datu savienošanas procesu nodrošina metode „*BindTextFragmentsToSingleFile*”, kas saņem kā parametrus adresi, kur jāsaglabā rezultāts, un patiesumvērtību, kas norāda, vai audio fragmenti jāsavieno ar pārklāšanos, vai bez.

Audio datu savienošana ar pārklāšanos veidota, lai nodrošinātu gludāku dažādu signālu pāreju, tādējādi nodrošinot uztveramāku runu. Rezultāti abu veidu datu savienošanai tikai nedaudz atšķiras, t. i., atšķirību var pamanīt tikai vairākas reizes noklausoties vienu un to pašu

tekstu ar abām datu savienošanas metodēm, līdz ar to kā noklusētais datu savienošanas veids atstāta datu savienošana bez pārklāšanās, kas ir mazāk resursietilpīgs process. Abas datu savienošanas metodes tiks apskatītas nedaudz tālāk.

Audio datu savienošanu nodrošina audio datu savienošanas klase (*WavBinder)*, izmantojot sistēmas dziņa atribūtu „*_binder*".

3.2.1.7. Runas sintēzes dziņa kļūdu apstrāde

Runas sintēzes dzinis pārtver un apstrādā kļūdas, kuras izraisa tā atribūti. Runas sintēzes dzinis pārtver kopā sešus kļūdu veidus un izraisa vienu kļūdas veidu. Kā redzams klašu diagrammā (sk. 3.2.1.1. attēlu), ir veikta neliela atkāpe no standarta UML (*Unified Modeling Language*) klašu diagrammu sintakses, respektīvi, no runas sintēzes dziņa klases (*Text2Speech*) iziet viena asociācija, ar nozīmi, ka dzinis apstrādā kļūdas (diagrammā – „*Handles*"). Šī asociācija sadalās sešās daļās (domātas sešas dažādas asociācijas), norādot kuras kļūdas dzinis apstrādā. Šāda sintakses atkāpe izmantota, lai diagramma būtu kompaktāka un uzskatāmāka.

Runas sintēzes dzinis inicializācijas posmā pārtver veselu vārdu vārdnīcas kļūdu (*FullWordDictionaryException*), transkribēšanas likumu kļūdu (*TranscriptionRulesException*) un fonētiskās bibliotēkas kļūdu (*PhoneticLibraryException*). Ja inicializācijas posmā dzinis pārtver kādu no šīm kļūdām, tiek izraisīta runas sintēzes dziņa kļūda (*Text2SpeechEngineException*) un tālāka sistēmas izmantošana nav iespējama, jo nav pareizi ielasīta datu bāze, kas nozīmē, ka sistēma nevar garantēt, ka sintēze būs veiksmīga. Šādas situācijas var rasties gadījumos, ja datu bāze apzināti ir izkropļota, t. i., ir izdzēstas vitāli svarīgas datu bāzes komponentes kā transkribēšanas meta likumi vai fonētiskās bibliotēkas saraksts.

Veidojot runas sintēzes pieprasījumus, sistēma vairs neizraisa runas sintēzes dziņa kļūdu, jo veicot vairākus pieprasījumus ir svarīgi, lai sistēma turpinātu darboties arī gadījumos, ja nav izdevies pilnvērtīgi nosintezēt vienu pieprasījumu. Šajā posmā dzinis pārtver visas iepriekšējās kļūdas un vēl papildus arī sekojošas kļūdas: teksta fragmentu saraksta kļūdu (*TextFragmentListException*), audio failu lasītāja kļūdu (*ReaderException*) un audio datu savienotāja kļūdu (*BinderException*).

Katrs kļūdas elements satur informāciju par to, kāda kļūda notikusi, un kāpēc notikusi kļūda, lai būtu iespējams izsekot varbūtējās sistēmas algoritmu vai datu bāzes kļūdas.

3.2.2. Teksta fragmentu saraksts

Iepriekšējā apakšnodaļā par runas sintēzes dzini teksta fragmentu saraksts (*TextFragmentList*) bija tikai pieminēts. Šajā apakšnodaļā tiks apskatīts teksta fragmentu saraksta mērķis sistēmā, tā uzbūve un funkcionalitāte, ko tas piedāvā.

Teksta fragmentu saraksta uzdevums ir glabāt pilnīgi visus datus par tekstu, kas jāpārvērš runā. Visi dati, kas tiek uzkrāti visos teksta apstrādes procesos, kas aprakstīti iepriekšējā apakšnodaļā, tiek saglabāti sarakstā, lai būtu iespējams veiksmīgāk veikt runas sintēzi.

3.2.2.1. Teksta fragmentu saraksta datu struktūra

Teksta fragmentu saraksts veidots kā saistīts saraksts, kura elementi ir teksta fragmentu elementi (*TextFragmentElement*). Klašu diagrammā (sk. 3.2.2.1.1. attēlu) redzams, ka teksta fragmentu saraksts satur norādes uz teksta fragmentu elementiem. Semantiski norādes veidotas uz saraksta pirmo elementu „*_first*”, saraksta pēdējo elementu „*_last*” un konkrētajā brīdī apstrādāto elementu „*_current*”. Lai nodrošinātu operatīvāku saraksta darbību, tiek glabāts arī saraksta elementu skaits „*_length*” un konkrētajā brīdī apstrādātā elementa indekss (pozīcija sarakstā) „*_currentIndex*”.

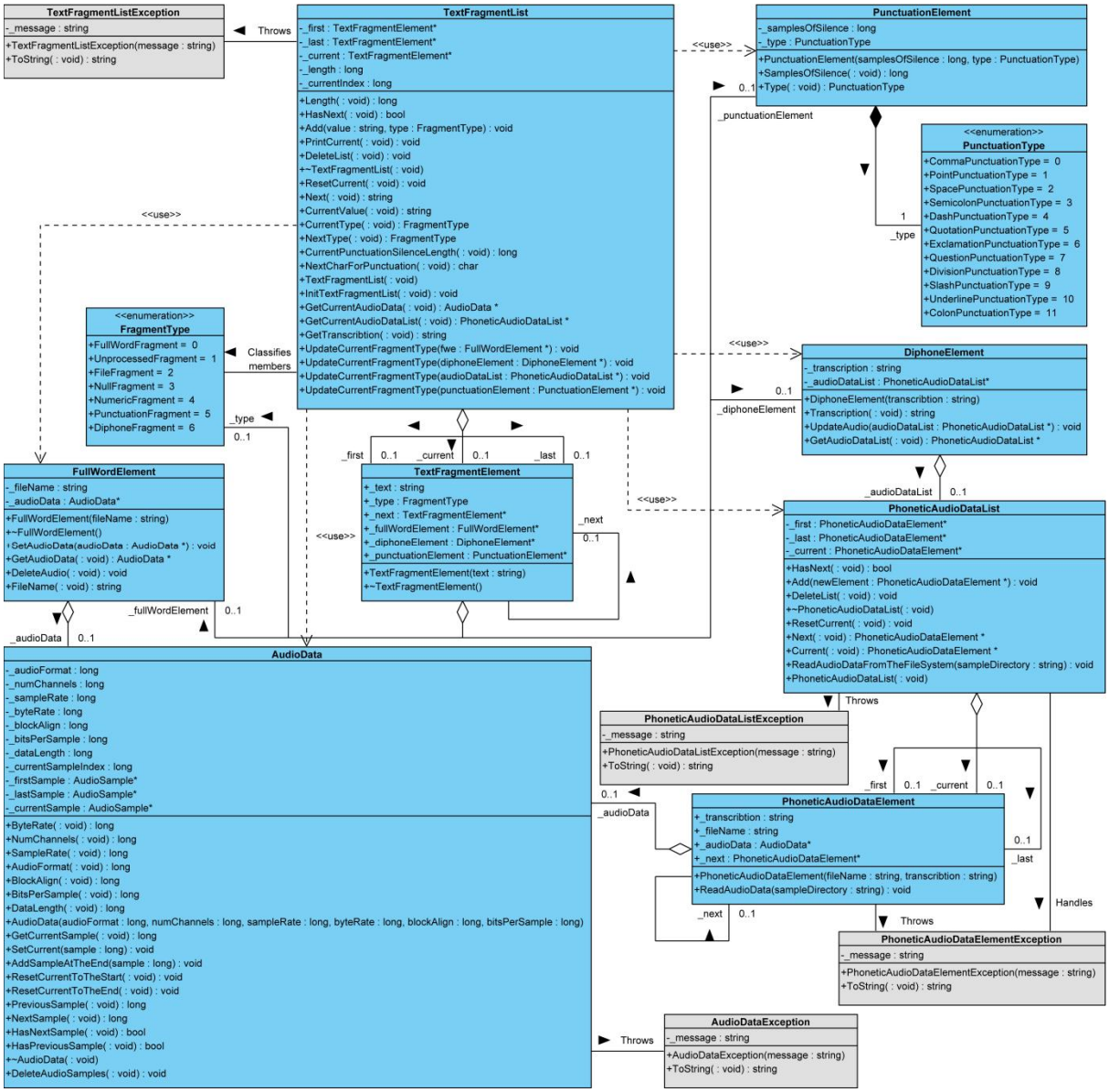
Katram teksta fragmenta elementam ir savs tips (*FragmentType*) „_type”, kas tiek piešķirts kādā no teksta apstrādes posmiem. Kopā iespējami septiņi fragmentu veidi: vesela vārda fragments (*FullWordFragment*), neapstrādāta teksta fragments (*UnprocessedFragment*), faila fragments (*FileFragment*), nulles fragments (*NullFragment*), skaitļa fragments (*NumericFragment*), pieturzīmes fragments (*PunctuationFragment*), difonu fragments (*DiphoneFragment*).

Neatkarībā no tā, kāda veida ir teksta fragmenta elements, tas vienmēr saturēs tekstu „*_text*”. Tā kā secīgi savienoti elementi veido sarakstu, tad katrs elements satur arī norādi uz nākošo teksta fragmenta elementu „*_next*”. Norāde uz nākošo elementu var būt arī tukša (piemēram, saraksta pēdējam elementam).

Atkarībā no tā, kāda veida ir teksta fragmenta elements, tas var saturēt vesela vārda elementu (*FullWordElement* - *_fullWordElement*), difonu elementu (*DiphoneElement* - *_diphoneElement*) vai pieturzīmes elementu (*PunctuationElement* - *_punctuationElement*). Ja teksta fragments satur vesela vārda elementu, tad tas satur arī audio fragmentu audio datu objektā „*_audioData*”, kas reprezentē veselo vārdu, kā arī faila nosaukumu, no kura iegūts audio fragments. Ja teksta fragments satur pieturzīmes elementu, tad tas satur informāciju par

to, kāda veida ir pieturzīmes fragments (*PunctuationType* - *_type*), kā arī, cik garš klusuma periods atbilst konkrētajam fragmentam (*_SamplesOfSilence*).

Gadījumā, ja teksta fragments satur difonu elementu, tas satur arī fragmenta saturošā teksta fonētisko transkripciju. Konkrētais teksta fragments pēc audio signālu piekārtošanas (sk. apakšnodaļu 3.2.1.5.) satur arī fonētisko audio datu sarakstu (*PhoneticAudioDataList*)



3.2.2.1.1. att. **Teksta fragmentu saraksta klašu diagramma**

Fonētisko audio datu saraksts ir speciāli difonu elementam veidota klase, kas domāta, lai atmiņā glabātu audio datus, kas pieder difonu fragmentiem pēc elementa transkripcijas (*_transcription*) segmentēšanas. Tā kā, vienu transkripciju segmentējot, var iegūt ļoti daudz īsākus transkripcijas fragmentus, nepieciešams saraksts, kas satur kopā visus vienas transkripcijas fragmentus. Katrs fonētiskā audio datu saraksta elements (*PhoneticAudioDataElement*) satur transkripcijas fragmentu, audio datus, kas reprezentē

transkripciju, faila nosaukumu, no kurienes iegūti audio dati, kā arī norādi uz nākošo elementu. Fonētisko audio datu saraksts ir projektēts kā saistīts vienvirziena saraksts. Audio datu klase tuvāk tiks apskatīta nodaļā par audio datu savienotāju.

3.2.2.2. Teksta fragmentu saraksta metožu struktūra

Ir apskatīta teksta fragmentu saraksta datu struktūra, līdz ar to, var apskatīt saraksta piedāvāto funkcionalitāti. Teksta fragmentu saraksts piedāvā divu veidu metodes: metodes, kas saistītas ar iterācijas un saraksta veidošanas procesu nodrošināšanu, un metodes, kas saistītas ar saraksta elementu, t. i., teksta fragmentu datu iegūšanu un papildināšanu.

Sarakstā ir sekojošas metodes, kas nodrošina iterācijas un saraksta veidošanas procesus:

- Metode, kas atgriež saraksta garumu (*Length*).
- Metode, kas nosaka, vai sarakstā pēc konkrētajā momentā apstrādātā elementa atrodas vēl kāds elements (*HasNext*).
- Metode, kas pievieno sarakstam jaunu teksta fragmenta elementu (*Add*).
- Metode, kas iztukšo sarakstu (*DeleteList*). Šī metode tiks izsaukta, kad teksts būs pilnībā pārvērsts runā, un rezultāts būs saglabāts failu sistēmā.
- Saraksta destruktors (*~TextFragmentList*). No atmiņas izdzēš sarakstu. Destruktors izsauc metodi, kas iztukšo sarakstu.
- Metode, kas saraksta iteratoru pārvieto uz saraksta sākumu (*ResetCurrent*).
- Metode, kas saraksta iteratoru pārvieto uz nākošo saraksta elementu (*Next*).
- Saraksta konstruktors, kas piešķir sarakstam sākotnējās atribūtu vērtības (*TextFragmentList*).
- Metode, kas pilda saraksta konstruktora funkcijas (*InitTextFragmentList*).

Šīs metodes nodrošina saraksta standarta funkcionalitātes iespējas: saraksta izveidošanu, elementu pievienošanu sarakstam, saraksta elementu skaita noteikšanu, kā arī saraksta un saraksta elementu dzēšanu.

Visas pārējās metodes nodrošina piekļuvi saraksta datiem un metožu darbība ir atkarīga no elementu vērtībām sarakstā. Ja saraksta iterators (*_current*), izsaucot šīs metodes, nenorāda uz kādu elementu, metode var izraisīt kļūdu (ja metode atjauno datus), vai arī atgriezt tukšu vērtību (ja metode atgriež datus). Līdz ar to ir ļoti svarīgi pareizi izmantot iepriekšējā tipa metodes! Metodes, kas nodrošina piekļuvi saraksta konkrētā momentā apstrādātā elementa datiem neatkarībā no elementa veida ir sekojošas:

- Metode konkrētajā momentā apstrādātā teksta fragmenta vērtības (fragmenta reprezentējošais teksts) iegūšanai (*CurrentValue*).

- Metode teksta fragmenta veida noteikšanai (*CurrentType*).
- Metode nākamā teksta fragmenta veida noteikšanai (*NextType*).

Metodes, kas nodrošina piekļuvi datiem atkarībā no tā, kāda veida ir saraksta konkrētajā momentā apstrādātais elements, ir sekojošas:

- Metode, kas atgriež pieturzīmes klusuma perioda garumu gadījumā, ja teksta fragments ir pieturzīme (*CurrentPunctuationSilenceLength*).
- Metode, kas atgriež nākamā teksta fragmenta pirmo burtu (*NextCharForPunctuation*). Metode tiks izmantota, lai noteiktu cik garš klusuma intervāls ir jāievieš atstarpes vietā atkarībā no nākamā teksta fragmenta pirmā burta. Lai nodrošinātu plūstošāku runu, atstarpes vietā pirms vārdiem, kas sākas ar „k", „p" vai „t" ir jāietur lielāks klusuma posms, tāpēc šī metode ir paredzēta, lai uzzinātu, kāds ir vārda pirmais burts.
- Metode teksta fragmenta audio datu iegūšanai (*GetCurrentAudioData*). Metode atgriezīs vesela vārda fragmenta audio datus. Citu teksta fragmenta veidu gadījumā tiks atgriezta tukša vērtība.
- Metode teksta fragmenta fonētiskā audio datu saraksta iegūšanai. Metode atgriezīs difonu fragmenta fonētisko audio datu sarakstu. Citu teksta fragmenta veidu gadījumā tiks atgriezta tukša vērtība.

Pēdējās divas metodes nodrošina piekļuvi audio datiem, kas nepieciešami, lai savienotu visus audio datus vienā rezultātā. Kā redzams audio datus satur tikai vesela vārda fragmenti un difonu fragmenti. Citiem fragmentu veidiem audio dati nav nepieciešami, lai nodrošinātu paredzamo funkcionalitāti.

Neapskatīta vēl ir palikusi viena metode, kas papildina saraksta elementu datus, un, kurai ir iespējami četri pārslogošanas varianti (*UpdateCurrentFragmentType*):

- Metode teksta fragmentu pārvērš par vesela vārda fragmentu, pievienojot vesela vārda elementu.
- Metode teksta fragmentu pārvērš par pieturzīmes fragmentu, pievienojot pieturzīmes elementu.
- Metode teksta fragmentu pārvērš par difonu fragmentu, pievienojot difonu elementu.
- Metode papildina difonu fragmentu ar fonētisko audio datu sarakstu.

Katrā gadījumā metode kā parametrus saņem dažādus datu tipus, tādējādi nodrošinot viennozīmīgu pareizās pārslogotās metodes izsaukumu.

3.2.2.3. Teksta fragmentu saraksta kļūdu apstrāde

Teksta fragmentu saraksts izraisa teksta fragmentu saraksta kļūdas (*TextFragmentListException*) gadījumos, ja tiek veikti pieprasījumi neesošiem datiem, t. i., teksta fragmentu saraksta iterators nenorāda uz nevienu elementu, bet tiek pieprasīta elementa vērtība. Kļūdas tiek izraisītas arī gadījumos, ja tiek mēģināts papildināt elementu datus ar tiem neparedzētiem datiem. Piemēram, vesela vārda teksta fragmentam mēģina pievienot audio datu sarakstu. Šāda rīcība nav atļauta un tiek izraisīta kļūda.

Kļūdas, kuras izraisa teksta fragmentu saraksts satur informāciju par to, kāda kļūda ir notikusi, un kāda bija operācija, kurā kļūda tika izraisīta.

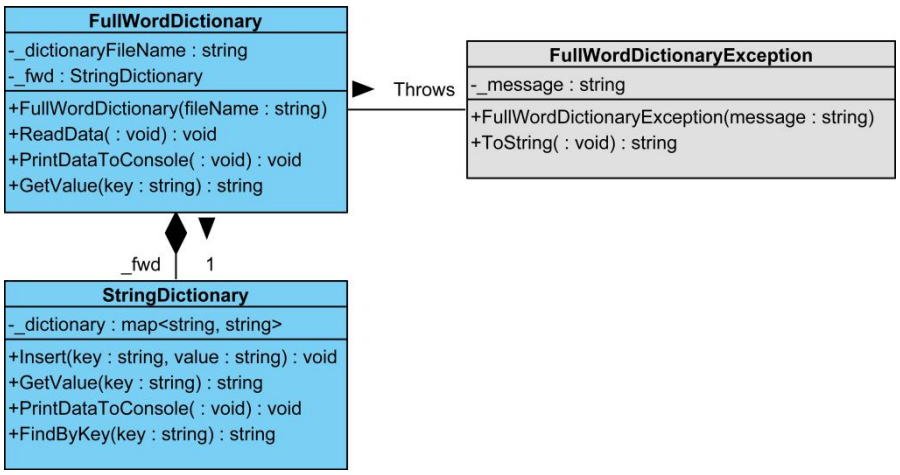
Attēlā 3.2.2.1.1. redzamas arī fonētiskā audio datu saraksta, fonētiskā audio datu elementa un audio datu kļūdas. Tā kā teksta fragmentu saraksts nemanipulē ar datiem, ko satur šie datu objekti, bet tikai uzglabā veselus datu objektus atmiņā (neizsauc metodes un neizmanto atribūtus), tad šīs kļūdas saraksta ietvaros nevar tikt izraisītas, līdz ar to nav jāapstrādā.

3.2.3. Veselu vārdu vārdnīca

Veselu vārdu vārdnīca (klašu diagramma apskatāma attēlā 3.2.3.1.), kā iepriekš tika noskaidrots (sk. nodaļu 3.2.1.2.), tiek izmantota, lai tekstā atrastu veselu vārdu fragmentus, kurus iespējams aizvietot ar audio fragmentiem no veselu vārdu vārdnīcas audio failu datu bāzes.

Lai nodrošinātu nepieciešamo funkcionalitāti, sistēmas operatīvajā atmiņā runas sintēzes dziņa inicializēšanas laikā tiek ielasīts veselu vārdu vārdnīcas ierakstu saraksts. Saraksts tiek glabāts simbolu virkņu vārdnīcā (*StringDictionary*). Simbolu virkņu vārdnīca tiek realizēta tādā veidā, ka viens ieraksts vārdnīcā satur divas simbolu virknes, kur pirmā ir atslēga, bet otra ir vērtība. Atslēgu veido vārds latviešu valodā ar mazajiem burtiem. Mazie burti domāti, lai nodrošinātu burtu lieluma neatkarīgu (*case insensitive*) teksta apstrādi. Vērtību veido audio faila nosaukums, kurā saglabāta atslēgas vārda akustiskā reprezentācija, t. i., runa.

Saraksta ielasi no failu sistēmas atmiņā nodrošina veselu vārdu vārdnīcas klase, kas simbolu virkņu vārdnīcu satur kā atribūtu „*_fwd*”. Ielasi nodrošina metode „ReadData”, kas kā parametru saņem adresi, kas norāda uz datu bāzes failu. Gadījumā, ja vārdnīcas datu ielases procesā parādās kāda kļūda, tiek izraisīta veselu vārdu vārdnīcas kļūda (*FullWordDictionaryException*), kuru tālāk apstrādā runas sintēzes dzinis.



3.2.3.1. att. **Veselu vārdu vārdnīcas klašu diagramma**

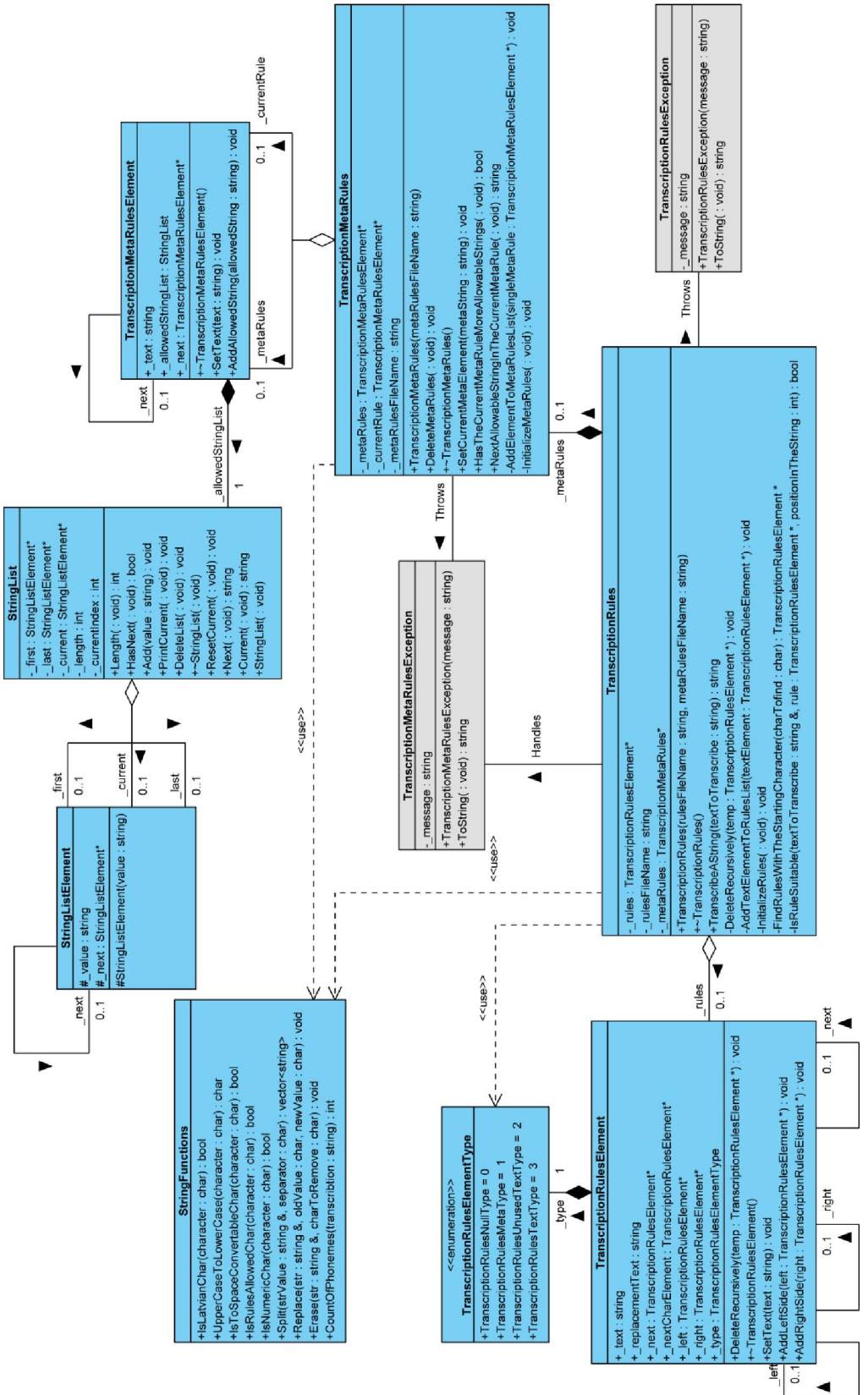
Veselu vārdu meklēšanas procesā teksta fragmentiem (vārdiem) tiek pārbaudīts, vai fragmenti ir sastopami veselu vārdu vārdnīcā ar metodes „*GetValue*" palīdzību. Ja metode atgriež rezultātu, kas nav vienāds ar tukšumu, tad vārds ir sastopams veselu vārdu vārdnīcā, un metodes atgrieztā vērtība ir audio faila nosaukums, kas atbilst meklētajam teksta fragmentam.

3.2.4. Transkribēšanas likumi

Ceturtais posms teksta pārvēršanā par runu ir teksta fragmentu transkribēšana fonētiskajā alfabētā. Fonētiskajā alfabētā rakstīts teksts sniedz informāciju par to, kādas skaņas nepieciešamas, lai tekstu pārvērstu runā. Katrs fonētiskā alfabēta simbols apzīmē vienu skaņu, t. i., vienu fonēmu.

Transkribēšanas likumi nepieciešami, lai tekstam piekārtotu pareizās skaņas, jo atsevišķi burti ne vienmēr viennozīmīgi atbilst kādai konkrētai skaņai [1]. Piemēram, apskatīsim vārdus „aparāts" un „cirks". Pirmais vārds beidzas ar „ts", bet izrunā netiek lietots „t" un „s", t. i., burti netiek izrunāti precīzās skaņās „t" un „s", bet gan citā skaņā, proti, „ts" (fonētiskajā alfabētā burts „c"). Toties vārdā „cirks" tiek viennozīmīgi lietota skaņa „c". Ne vienmēr burtu savienojums „ts" tiks aizstāts ar skaņu „ts". Piemēram, vārdā „atsaukt" tiek lietotas skaņas „t" un „s". Tieši šādiem gadījumiem ir paredzēti transkribēšanas likumi: lai pārvērstu tekstu no latviešu valodas ortogrāfijas rakstības uz fonētisko alfabētu. Fonētiskais alfabēts jau viennozīmīgi nosaka, kāda skaņa tiks konkrētajā pozīcijā izrunāta, līdz ar to pēc transkribēšanas nepieciešams tikai piekārtot skaņu fragmentus, ņemot vērā transkripciju.

Transkribēšana, kā jau iepriekš noskaidrots (sk. nodaļu 3.2.1.4.), tiek pielietota neapstrādātajiem teksta fragmentiem. Transkribēšana ir pēdējais augsta līmeņa teksta sintēzes



posms „Konkatenatīvas latviešu valodas runas sintēzes sistēmā”. Transkribēšanas likumu datu struktūru un funkcionalitāti uzskatāmi parāda klašu diagramma attēlā 3.2.4.1.

3.2.4.1. att. **Transkribēšanas likumu klašu diagramma**

Likumu pielietošanas loģika ir pārņemta no iepriekš institūtā izstrādātās runas sintēzes sistēmas, par ko tika stāstīts pirmajā nodaļā. Datu struktūra un datu bāzes struktūra savukārt ir veidotas no jauna, lai nodrošinātu optimālāku likumu pielietojumu, kā arī, lai padarītu likumus lasāmākus (XML struktūra).

3.2.4.1. Transkribēšanas likumu datu struktūra

Transkribēšanas likumu galvenā klase „*TranscriptionRules*" nodrošina pilnu likumu pielietošanas funkcionalitāti runas sintēzes dzinim, kā arī satur visus datus, kas nepieciešami, lai nodrošinātu transkribēšanu. Tā kā transkribēšanai nepieciešami arī meta likumi (*TranscribtionMetaRules*), lai nodrošinātu pareizāku transkribēšanu, tie tiek glabāti iekš transkribēšanas likumu klases kā atribūts „*_metaRules*".

Transkribēšanas likumi ir veidoti kā trīsdimensionāls saraksts, kur transkribēšanas likumu klase semantiski satur norādi uz saraksta pirmo elementu (*_rules*). Katrs elements ir atsevišķs datu objekts ar nosaukumu „*TranscriptionRulesElement*".

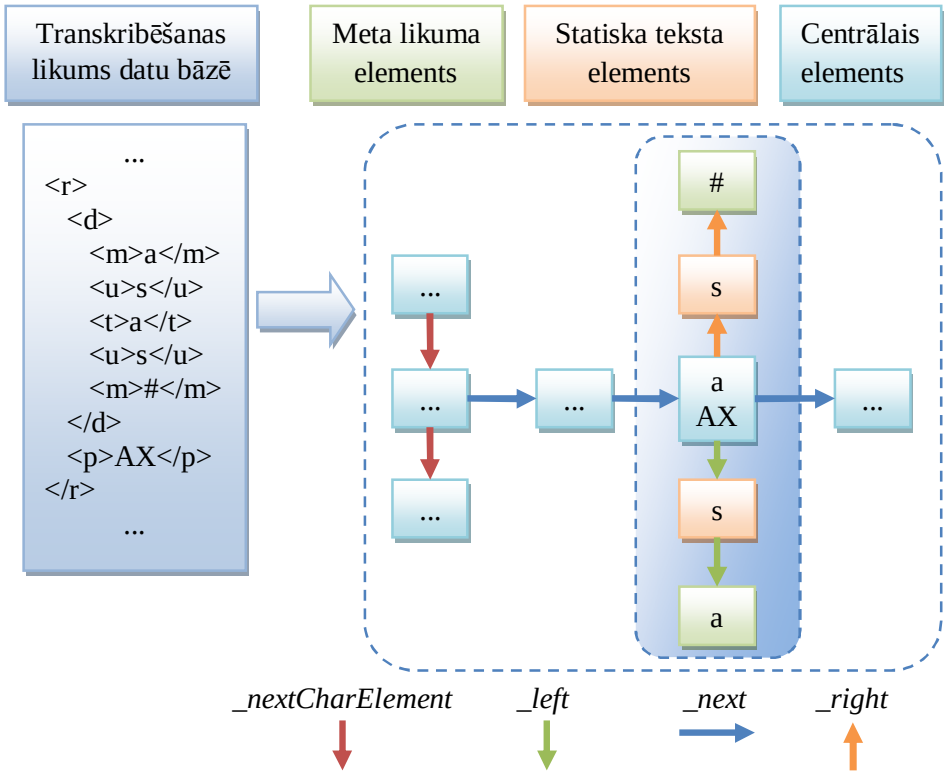
Trīs dimensijas sarakstā veidojas, savienojot elementus trīsdimensionālā ķēdē, kur katrs transkribēšanas likumu elements satur norādi uz nākošo elementu (*_nextCharElement* – pirmā dimensija) ar citādāku likuma teksta pirmo simbolu, norādi uz nākošo elementu (*_next* – otrā dimensija) ar vienādu likuma teksta pirmo simbolu, norādi uz elementu pa kreisi no konkrētā elementa (*_left* – otrās trešās negatīvo vērtību puse) un norādi uz elementu pa labi no konkrētā elementa (*_right* – trešās dimensijas pozitīvo vērtību puse).

Viens likums šādā trīsdimensionālā struktūrā saprotams kā trešā dimensija, t. i., transkribēšanas likumu centrālais elements (tāds, pie kura var nonākt sekojot tikai pirmās dimensijas norādēm „*_nextCharElement*" un otrās dimensijas norādēm „*_next*") ar tā kreiso (*_left*) un labo (*_right*) pusi. Katrs elements kreisajā pusē satur tikai norādes „*_left*" („*_right*", „*_next*" un „*_nextCharElement*" ir tukšas) un katrs elements labajā pusē satur tikai norādes „*_right*" („*_left*", „*_next*" un „*_nextCharElement*" ir tukšas), attiecīgi veidojot sarakstu. Iterācija pa dažādiem likumiem notiek pa pirmo un otro dimensiju, bet iterācija pa vienu likumu notiek pa trešo dimensiju. Norādes galējiem elementiem, protams ir tukšas. Datu struktūru vizuāli parāda diagramma 3.2.4.1.1. attēlā.

Dažādi likumi ir sadalīti pa divām dimensijām, lai optimāli nodrošinātu apstrādes ātrumu. Vispirms tiks atrasts likums, kam ir tāds pats pirmais simbols, kā tekstam konkrētajā pozīcijā (kopā iespējami 54 dažādi burti un cipari), un tad tiks meklēts pirmais likums pa otro dimensiju, kas atbilst teksta konkrētajai pozīcijai. Tā kā uz doto brīdi datu bāzē ir ap 150 likumiem (tas gan nenozīmē, ka nebūs vairāk), kuriem lielākoties teksts ir vienu simbolu garš,

šāda datu izkārtošana divās dimensijās ir optimāls risinājums, lai nodrošinātu efektīvu teksta transkribēšanu.

Transkribēšanas likumu elementus iedala četros veidos. Pirmās dimensijas centrālie elementi, t. i., elementi, kurus no saraksta saknes elementa „*_rules*" klasē „TranscribtionRules" var sasniegt, izmantojot tikai norādes „_next", vienmēr būs teksta tipa elementi (*TranscriptionRulesTextType*). Šāds nosaukums piešķirts, jo centrālie elementi satur teksta fragmentu un transkripciju konkrētajam teksta fragmentam (*_replacementText*) likuma ietvaros. Ja tekstā, kas būs jātranskribē fonētiskajā alfabētā, tiks atrasts konkrētā pozīcijā esošais teksta fragments, un tieši tajā pašā pozīcijā izpildīsies transkribēšanas likums, tad konkrētais teksta fragments tiks aizstāts ar transkripciju. Otrās dimensijas elementi, kas nav centrālie elementi, iedalās divos veidos: meta likuma elementos (*TranscriptionRulesMetaType*) un statiska teksta (*TranscriptionRulesUnusedTextType*) elementos. Kā attēlā 3.2.4.1.1 redzams, abi elementi iegūti no transkribēšanas likumu datu bāzes attiecīgajiem elementiem. Meta likuma elementi norāda uz konkrētu meta likumu, kam teksta konkrētajā pozīcijā ir jāizpildās, bet statiska teksta elementi definē konkrētu teksta fragmentu, kam ir obligāti jāparādās transkribējamajā tekstā konkrētajā pozīcijā.



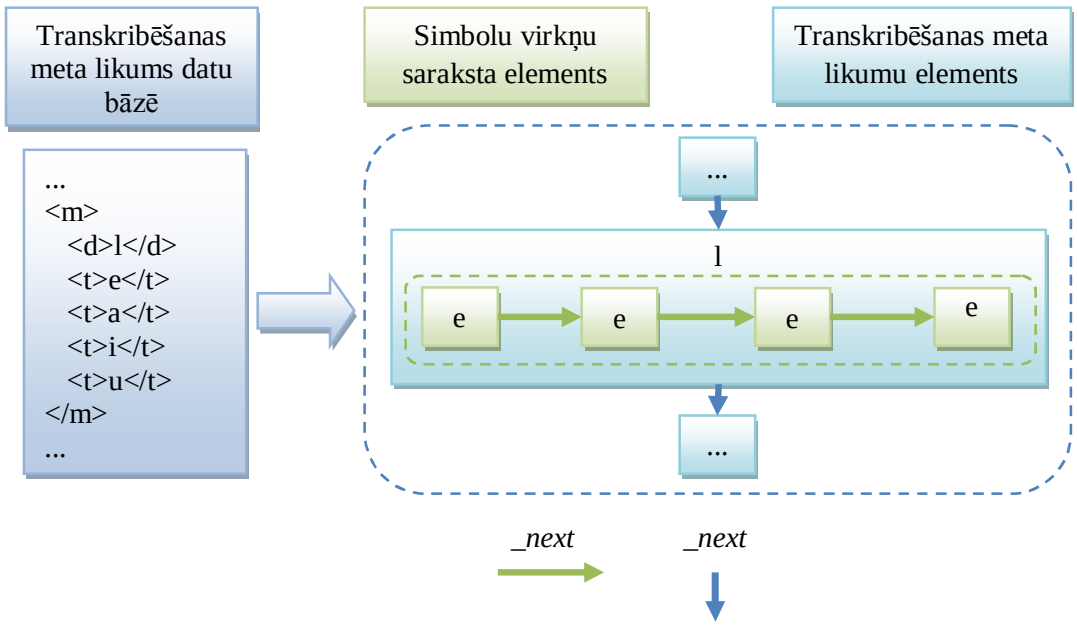
3.2.4.1.1. att. **Transkribēšanas likumu elementu saraksta struktūra**

Pēdējā veida transkribēšanas likumu elementi ir tukšie elementi (*TranscriptionRulesNullType*), t. i., tādi elementi, kuriem nav piešķirts konkrēts tips. Šo tipu elementi saņem pie to izveides, ja netiek norādīts konkrēts elementa tips. Pēc veiksmīgas datu

bāzes ielases šādi elementi neparādīsies. Gadījumā, ja datu bāzes ielasē kāds elements būs bijis kļūdains, tad tas netiks apstrādāts, jo saturēs tukšu elementu.

Transkribēšanas meta likumi veidoti kā saistīts vienvirziena saraksts, kas satur tikai vienu dimensiju, bet katrs meta likumu elements (*TranscriptionMetaRulesElement*) iekšēji satur simbolu virkņu sarakstu (*StringList*) atribūta „*_allowedStringList*" veidā, kas teorētiski (bet ne praktiski) varētu veidot otro dimensiju. Meta likumu datu struktūru vizuāli attēlo 3.2.4.1.2. attēls.

Transkribēšanas meta likumu elements satur pilnībā visu informāciju, kas nepieciešama vienam meta likumam. Elements satur meta likuma nosaukumu (*_text*), likuma paskaidroto teksta fragmentu sarakstu (*_allowedStringList*) un norādi uz nākošo elementu. Secīgi savienoti meta likumu elementi veido meta likumu sarakstu.



3.2.4.1.2. att. **Transkribēšanas meta likumu elementu saraksta struktūra**

3.2.4.2. Transkribēšanas likumu datu izveide

Transkribēšanas likumi tiek inicializēti, kad tiek izsaukts konstruktors (*TranscriptionRules*). Konstruktors definē noklusētos parametrus un izsauc iepriekš apskatītās datu ielases funkcijas gan transkribēšanas likumiem, gan transkribēšanas meta likumiem, izsaucot meta likumu konstruktoru (*TranscriptionMetaRules*). Datu atbrīvošana no atmiņas tiek veikta ar destruktoru palīdzību, t. i., ar metodi „*~TranscriptionRules*" transkribēšanas likumiem un metodi „*~TranscriptionMetaRules*" meta likumiem. Abi destruktori attiecīgi izsauc citas metodes, kas nodrošina sarakstu elementu izdzēšanu no atmiņas. Metode „*DeleteRecursively*" iekš transkribēšanas likumu klases atbrīvo atmiņu no visiem

transkribēšanas likumu elementiem, bet metode „*DeleteMetaRules*” iekš meta likumu klases atbrīvo atmiņu no visiem transkribēšanas meta likumu elementiem.

Transkribēšanas likumu datu objektu veidošana tiek veikta sistēmas inicializācijas posmā. Par datu ielasi no datu bāzes transkribēšanas likumiem ir atbildīga metode „*InitializeRules*”, bet meta likumiem – metode „*InitializeMetaRules*”. Abos gadījumos tiek ielasīti dati no datu bāzes, kas glabājas XML struktūrā. Vienlaicīgi ar datu ielasi, šie dati tiek parsēti, un izveidoti datu objekti, kas atbilst iepriekšējā apakšnodaļā aprakstītajai datu struktūrai.

Katram transkribēšanas likumam vispirms tiek izveidots elementu saraksts, kas pilnīgi reprezentē konkrēto likumu (centrālais elements un ap to saistītie elementi), un tikai tad, kad visi dati par likumu ielasīti, saraksts tiek pievienots pārējiem transkribēšanas likumiem, attiecīgi savienojot likuma centrālos elementus. Šo savienošanas procesu nodrošina metode „*AddTextElementToRulesList*”. Šī metode ir atbildīga par to, kā elementi tiek izkārtoti pareizi pirmajā un otrajā dimensijā.

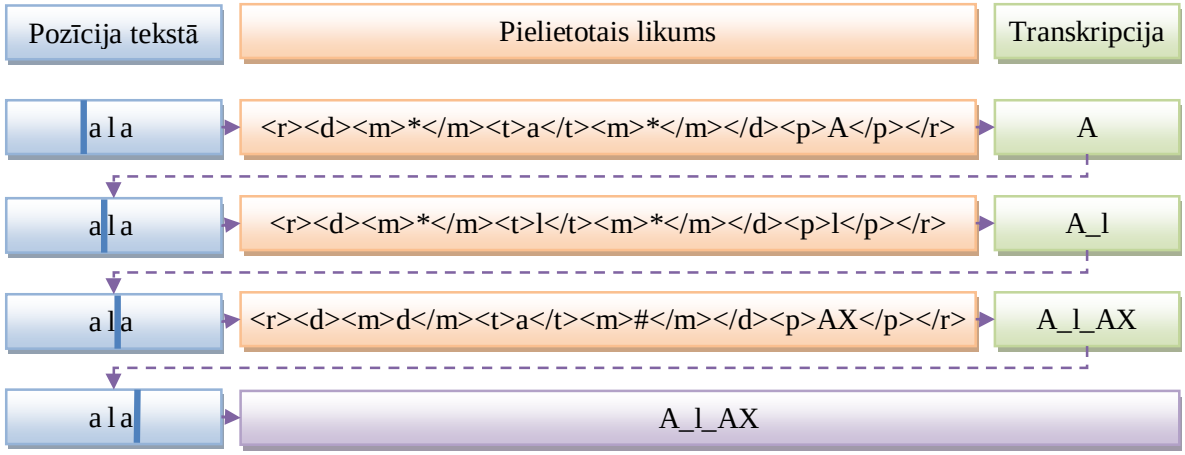
Meta likumiem nav tik sarežģīta datu struktūra, līdz ar to datu izveidošana ir vienkāršāka. Katrs transkribēšanas meta likums tiek atsevišķi izveidots, piešķirot tam nosaukumu un aizpildot atļauto simbolu virkņu sarakstu. Pēc tam tas tiek pievienots sarakstā jau esošajiem elementiem ar metodes „*AddElementToMetaRulesList*” palīdzību. Tā kā meta likumiem nav svarīga secība un to mazais skaits (līdz 10) neprasa optimizāciju, metode vienkārši pievieno elementu saraksta sākumā.

Transkribēšanas likumiem un meta likumiem nav izstrādātas metodes, kas ļautu pakāpeniski iterēt cauri sarakstiem, kā tas bija iespējams teksta fragmentu sarakstam. Iterācija tiek veikta pilnīgi atsevišķās metodēs. Šāds risinājums nodrošināts, jo transkribēšanas likumu dati netiek mainīti sistēmas darbības laikā, bet dati tiek tikai iegūti, pie kam, nav zināms, kur atrodas likums, no kura būs nepieciešama informācija konkrētā teksta apstrādes brīdī.

3.2.4.3. Teksta transkribēšanas funkcionalitāte

Līdz šim tika apskatīta tikai transkribēšanas likumu uzbūve, bet nebija izskaidrots, kādā veidā transkribēšanas likumi tiek pielietoti, lai transkribētu tekstu. „Konkatenatīvas latviešu valodas runas sintēzes sistēmas” ietvaros transkribēti tiks tikai atsevišķi vārdi, jo transkribēšana, kā iepriekš tika noskaidrots (sk. nodaļu 3.2.1.4.), tiks veikta teksta fragmentiem. Līdz ar to kā piemērs tiks apskatīts teksta fragments „ala”. Fragments ir pietiekami liels, lai parādītu, ka tiek piekārtots sākuma transkripcijas fragments, teksta vidus transkripcijas fragments (garākiem teksta fragmentiem būtu vairāki vidus fragmenti) un teksta beigu transkripcijas fragments.

Transkribēšanu nodrošina transkribēšanas likumu metode „*TranscribeAString*”, kas kā parametru saņem transkribējamo tekstu. Lai būtu vieglāk uztverama teksta apstrādes loģika, 3.2.4.3.1. attēlā parādīti rezultāti, kas iegūti pēc katra iterācijas posma cauri tekstam.



3.2.4.3.1. att. **Teksta transkribēšanas piemērs vārdam „ala”**

Lai transkribētu tekstu, tiek iterēts cauri transkribējamajam tekstam un katrā pozīcijā tiek meklēts likums, kas izpildās [1]. Vispirms tiek atrasts pareizais pirmās dimensijas likums ar metodes „*FindRulesWithTheStartingCharacter*” palīdzību, t. i., tiek atrasts likums, kura teksta fragments sākas ar simbolu, kāds ir tekstā konkrētajā pozīcijā, iterējot pa norādēm „*_nextCharElement*”, sākot ar saknes elementu „*_rules*”, kas ir definēts transkribēšanas likumu klasē. Kad atrasts attiecīgā simbola pirmais likums, ar metodes „*IsSuitableRule*” palīdzību tiek noteikts, vai likums konkrētajā pozīcijā ir derīgs, t. i., vai pilnīgi visi likuma nosacījumi konkrētajā pozīcijā izpildās. Ja likums neizpildās, tiek pārbaudīts nākošais likums no otrās dimensijas, iterējot pa norādēm „*_next*”. Likumi ir izstrādāti pēc principa, ka vismaz viens likums katrā dimensijā ir pats vispārīgākais un derēs visos konkrētā simbola gadījumos, t. i., netiks skatīta simbola apkārtne. Līdz ar to visām pozīcijām tiks atrasts likums, bet, protams, ir ieviests arī risinājums gadījumiem, ja netiks atrasts atbilstošs likums (gadījumos, ja kāds ir ļaunprātīgi sabojājis datu bāzi) – tiks izlaists teksta konkrētās pozīcijas simbols un pozīcija tiks pavirzīta pa vienu simbolu uz priekšu. Ja tomēr ir atrasts atbilstošs likums, kas izpildās, transkripcija tiek papildināta ar jaunu fragmentu, kas ņemts no likuma elementa atribūta „*_replacementText*”, un pozīcija tekstā tiek pavirzīta uz priekšu par tik pozīcijām, cik garš ir teksta fragments likuma elementā. Šādi notiek iterācija cauri visam teksta fragmentam, kamēr apstrādes pozīcija ir novietota aiz teksta fragmenta pēdējā simbola. Rezultātā tiek iegūta transkripcija, kas atbilst sākotnējam tekstam.

Apskatot piemēru (sk. 3.2.4.3.1. attēlu) nedaudz precīzāk, var redzēt, ka pielietotie likumi patiešām ir derīgi attiecīgajās teksta pozīcijās. Sākot transkribēšanu pozīcija ir pirms vārda, līdz ar to sistēma ir atradusi likumu, kurš satur teksta fragmentu „a”, kas der vārdam

„ala” sākuma pozīcijā. Izpildās arī meta likumi, kas nosaka, ka pa kreisu un pa labi no teksta fragmenta „a” drīkst atrasties jebkāds skaits jebkādu simbolu. Līdz ar to transkripcija tiek papildināta ar fragmentu „A” un pozīcija tekstā tiek pavirzīta uz priekšu par vienu simbolu. Tiek sākts otrais iterācijas cikls ar pozīciju pirms simbola „l”. Sistēma ir atradusi iepriekšējam likumam līdzīgu vispārīgu likumu, kas izpildās un papildina transkripciju ar fragmentu „l”. Kā redzams, pēdējā izpildītajā iterācijā pirms pēdējā simbola „a” tiek pielietots nedaudz citādāks likums, kā iepriekšējās divās iterācijās. Likums nosaka, ka pirms simbola „a” ir jāizpildās meta likumam „d” un aiz simbola „a” nedrīkst būt neviens cits simbols. Meta likums „d” atļauj tā pozīcijā atrasties teksta fragmentam „l” un aiz simbola „a” patiešām nav neviena cita simbola. Līdz ar to transkribēšanas likums izpildās un transkripcija tiek papildināta ar fragmentu „AX”. Tā kā pozīcija tiek novietota aiz pēdējā vārda „ala” simbola, vairāk iterācijas netiek veiktas, un ir iegūta meklētā transkripcija „A_l_AX”.

3.2.4.4. Transkribēšanas likumu kļūdu apstrāde

Transkribēšanas meta likumu klase var izraisīt transkribēšanas meta likumu kļūdu (TranscriptionMetaRulesException) gadījumos, ja nav iespējams ielasīt no datu bāzes atmiņā meta likumus, un, ja tiek nepareizi izmantotas klases metodes, t. i., nepareizi tiek pieprasīti dati, kurus nav iespējams vispār iegūt (saraksta iterators nav novietots uz kāda elementa).

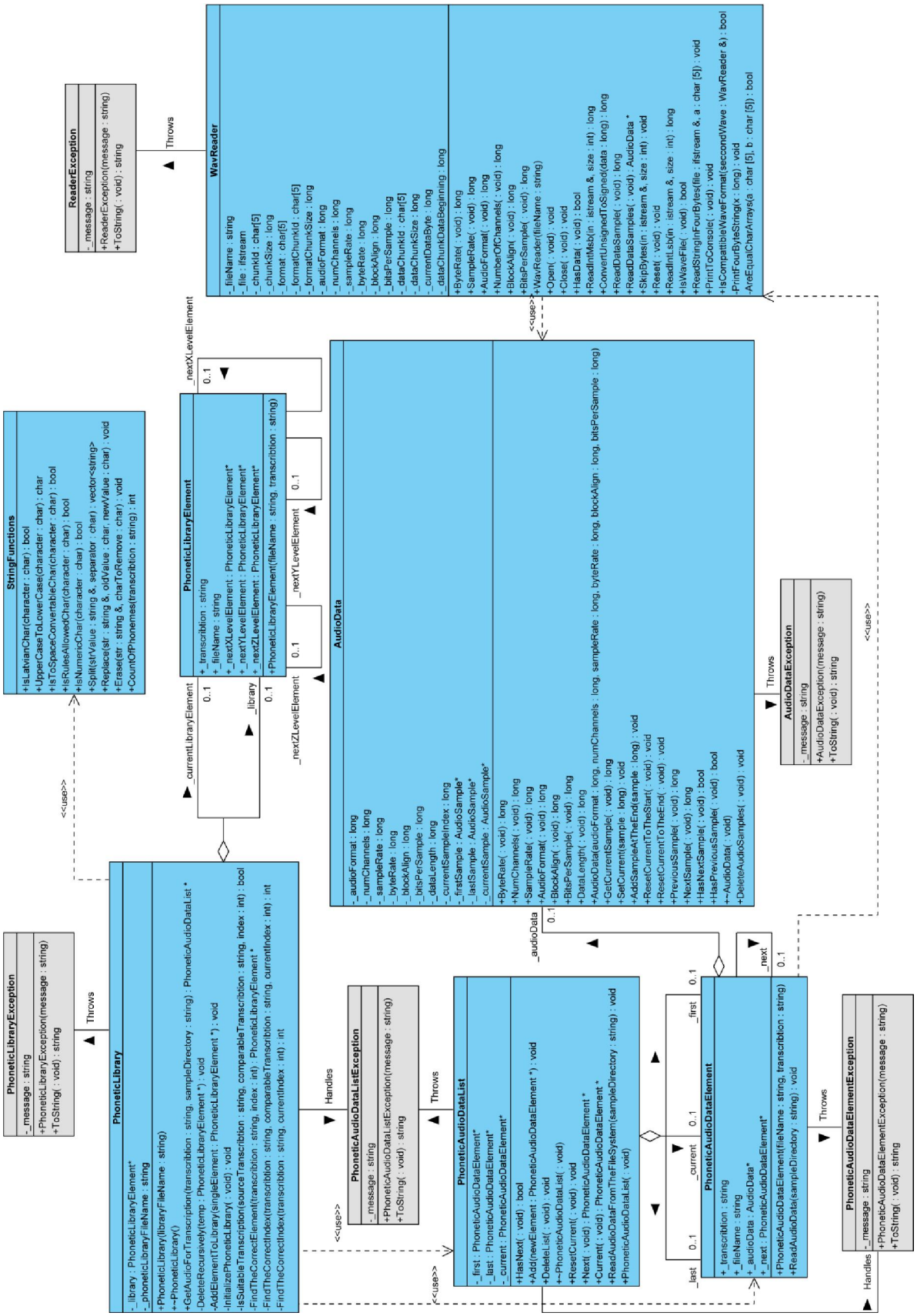
Iepriekšējo kļūdu pārtver un apstrādā transkribēšanas likumu klasē, jo tieši šīs klases metodes izsauc meta likumu metodes, kas izraisa kļūdas. Apstrādes laikā tiek izraisītas transkribēšanas likumu kļūdas (TranscribtionRulesException), kas tiek apstrādātas runas sintēzes dzinī. Varētu teikt, ka apstrāde transkribēšanas likumu klasē ir kļūdas tālāka novirzīšana ar papildus informāciju, kas nebija pieejama meta likumu klasē.

3.2.5. Fonētiskā bibliotēka

Fonētiskā bibliotēka paredzēta, lai nodrošinātu pareizu audio fragmentu piekārtošanu teksta fragmentiem, segmentējot teksta fragmentu transkripcijas vairākos fragmentos. Šī apstrādes posma rezultātā teksts saņems sarakstu ar audio un transkripciju fragmentiem, ko savienojot kopā (nākošajā posmā) tiks iegūta konkrētā teksta fragmenta runa.

3.2.5.1. Fonētiskās bibliotēkas datu struktūra

Fonētiskās bibliotēkas datu struktūru uzskatāmi parāda klašu diagramma 3.2.5.1.1. attēlā.



3.2.5.1.1. att. **Fonētiskās bibliotēkas klašu diagramma**

Fonētiskās bibliotēkas datu bāzes ierakstu saraksta glabāšanai atmiņā un nepieciešamās funkcionalitātes nodrošināšanai tiek izmantots trīsdimensionāls saraksts (*PhoneticLibrary*),

kas tiek veidots no fonētiskās bibliotēkas elementiem (*PhoneticLibraryElement*). Fonētiskās bibliotēkas klase kā atribūtu satur norādi uz saraksta pirmo elementu (*_library*), no kura iespējams sasniegt visus pārējos saraksta elementus.

Fonētiskās bibliotēkas elements sastāv no transkripcijas fragmenta (*_transcription*), audio faila nosaukuma (*_fileName*), kurā atrodama elementa transkripcijas akustiskā reprezentācija, norādes uz nākamo elementu, kura transkripcijas garums ir mazāks par dotā elementa transkripcijas garumu (*_nextXLevelElement* – pirmā dimensija), norādi uz nākamo elementu, kura transkripcijas garums ir vienāds ar dotā elementa transkripcijas garumu, bet transkripcijas otrais simbols ir atšķirīgs (*_nextYLevelElement* – otrā dimensija), un norādi uz nākamo elementu, kura transkripcijas garums ir vienāds ar dotā elementa transkripcijas garumu un transkripcijas otrais simbols arī ir vienāds ar dotā elementa transkripcijas otro simbolu (*_nextZLevelElement* – trešā dimensija).

Pirmā dimensija sadala transkripcijas dažādos garumos, jo ir svarīgi, lai garākie transkripcijas fragmenti apstrādes laikā tiktu aizstāti ar audio fragmentiem pirms tie nav sadalīti mazākās daļās. Labāku sintēzes rezultātu var iegūt, lietojot gatavus vārdu fragmentus, nekā tos sintezējot, piemēram, priedēkli „pār" aizvietojot ar gatavu audio fragmentu nav jādomā, kā nodrošināt veiksmīgu skaņu pārejas prosodiju, jo tā jau ir pieejama gatavajā audio fragmentā. Ja tomēr priedēklis būtu jāsavieno no atsevišķiem fragmentiem („p", „p-ā" un „ā-r"), prosodiju nebūtu iespējams nodrošināt tik skaidru un izteiktu. Līdz ar to ir svarīgi vispirms pārbaudīt, vai nav pieejami garāki fragmenti, kas atbilst transkripcijai, un tikai pēc tam meklēt īsākus fragmentus.

Otrā dimensija, līdzīgi kā transkribēšanas likumu pirmā dimensija, sadala transkripcijas grupās, kuras viena no otras atšķiras ar dažādiem transkripciju otrajiem simboliem. Šoreiz tiek ņemts vērā otrais simbols, jo transkripciju fragmenti no fonētiskās bibliotēkas datu bāzes ierakstu saraksta lielākoties sākas ar „_" simbolu, tādējādi definējot, ka fragments sākas ar pusfonēmas otru (beigu) pusi. Tikai transkripciju fragmenti, kas sākas ar pusfonēmas kreiso pusi (sākumu) nesatur simbolu „_", bet šo transkripciju skaits ir vairākas reizes (apmēram 20) mazāks. Līdz ar to būtu nelietderīgi ņemt vērā pirmo simbolu. Šī dimensija lielākoties nepieciešama, lai samazinātu datu skaitu, caur kuriem būtu jāmeklē atbilstošākais transkripcijas fragments.

Trešajā dimensijā elementi ir izkārtoti ar vienādu transkripciju garumu, un vienādiem otrajiem transkripciju simboliem. Apstrādes secība šajā dimensijā nav būtiska, jo nebūs tādu divu identisku likumu, kas būs vienlīdz derīgi transkripcijas aizstāšanai.

Kā redzams klašu diagrammā (sk. 3.2.73.2.5.1.1. attēlu)ar fonētisko bibliotēku ir saistītas arī citas klases, piemēram, fonētisko audio datu saraksts, un fonētisko audio datu

elements, bet šo klašu struktūra precīzāk tiks apskatīta nākošajā nodaļā par audio datu savienotāju.

3.2.5.2. Fonētiskās bibliotēkas funkcionalitāte

Iepriekš tika noskaidrota fonētiskās bibliotēkas datu struktūra, bet tagad tiks apskatītas iespējas, ko piedāvā fonētiskā bibliotēka.

Vispirms fonētiskā bibliotēka ir jāizveido un jāaizpilda ar datiem. Izveidi nodrošina fonētiskās bibliotēkas konstruktors (*PhoneticLibrary*), kas piešķir klases atribūtiem to noklusētās vērtības. Datu ielasi no datu bāzes nodrošina datu inicializācijas metode (*InitializePhoneticLibrary*). Inicializācijas metode izveido atsevišķus elementus, kas ar saraksta papildināšanas metodi (*AddElementToLibrary*) tiek pievienoti sarakstam attiecīgajā pozīcijā. Saraksta papildināšanas metode ir atbildīga par iepriekšējā nodaļā aprakstītā fonētiskās bibliotēkas datu saraksta trīsdimensionālās struktūras izveidi.

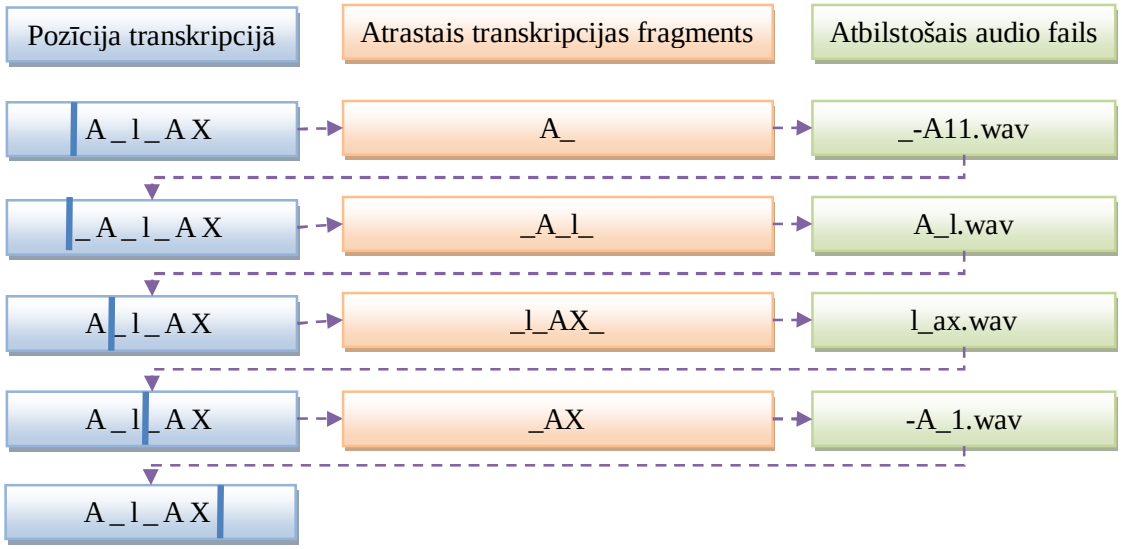
Lai atbrīvotu atmiņu no visiem fonētiskās bibliotēkas elementiem, ir izveidots destruktors (*~PhoneticLibrary*), kas izsauc metodi (*DeleteRecursively*), kas rekursīvi atbrīvo atmiņu no visiem elementiem.

Fonētiskā bibliotēka nodrošina transkripcijas segmentāciju fragmentos un audio datu ielasi no datu bāzes katram atsevišķajam fragmentam. Šo funkcionalitāti nodrošina metode „*GetAudioForTranscription*”, kas kā parametru saņem transkripciju, bet atgriež fonētisko audio datu sarakstu ar segmentēto transkripciju. Līdzīgi, kā teksta transkribēšanas procesā, notiek iterācija pa transkripciju. Šoreiz atšķirība ir tā, ka iterācija notiek tikai pa simbolu „_” pozīcijām starp fonēmām, jo simboli „_” atdala fonēmas, vienu no otras. Konkrētā transkripcijas pozīcijā tiek atrasts atbilstošs fonētiskās bibliotēkas elements ar metodes „*FindTheCorrectElement*” palīdzību. Iekš metodes notiek iterācija pa visām dimensijām, bet tiek pārbaudīti uz atbilstību tikai elementi, kuru transkripcijas otrais simbols sakrīt ar dotās transkripcijas simbolu aiz simbola „_”, pie kam iterācija notiek virzienā no garākām transkripcijām, uz īsākām. Pārbaudi, vai fonētiskās bibliotēkas elementa transkripcija atbilst dotajai transkripcijai konkrētā pozīcijā tiek veikta ar metodes „*IsSuitableTranscription*” palīdzību.

Kad elements ar atbilstošu transkripciju atrasts, tiek izveidots jauns fonētisko audio datu saraksta elements, kam tiek pievienota atrastā transkripcija, tiek ielasīti audio dati no datu bāzes faila un ar metodes „FindTheCorrectIndex” palīdzību tiek noteikta jauna apstrādes pozīcija transkripcijā. Jaunā pozīcija gadījumā, ja atrastā transkripcija bija beigu fragments, tiek novietota aiz dotās transkripcijas. Visos pārējos gadījumos pozīcija tiek novietota pirms iepriekšējā atrastā elementa transkripcijas pēdējās fonēmas. Kamēr netiek sasniegtas dotās

transkripcijas beigas, notiek iterācijas process, meklējot jaunajai pozīcijai jaunu fonētiskās bibliotēkas elementu.

Lai būtu saprotamāka transkripcijas segmentācija, tiks apskatīta iepriekšējās apakšnodaļas piemēra transkripcijas segmentācija (sk. . attēlu). Vārdam „ala” iepriekš tika iegūta transkripcija „A_l_AX”.



3.2.5.2.1. att. **Transkripcijas segmentēšanas piemērs vārdam „ala”**

Kā redzams diagrammā, segmentācija sākas, apskatot pozīciju transkripcijā pirms pirmā simbola. Tā kā pirmo transkripcijas fragmentu apstrādājot, tiek atrasta atbilstoša pusfonēma, tad pozīciju nav iespējams pārvietot pirms pēdējās atrastās fonēmas. Ja netiktu veiktas izmaiņas sākotnējā transkripcijā, sistēma ieciklotos un visu laiku atrastu sākuma pusfonēmas „A_”. Lai nodrošinātu to, ka sistēma vairs sākuma pusfonēmas neatrod, transkripcijas sākumā tiek pievienots simbols „_”, kas norāda, ka konkrētajā pozīcijā der tikai fragments, kas sākas ar fonēmas labo (beigu) pusi. Sistēmā pusfonēmas pieejamas tikai vārdu sākumu un beigu fragmentiem.

Nākošajā iterācijas posmā pozīcija tekstā ir pirms simbola „_” transkripcijas sākumā, bet, kā tika noskaidrots, sākuma fragmenti vairs netiks piekārtoti. Sistēma šajā posmā ir atradusi difonu „_A_l_”. Līdz ar to tiek noteikta jauna pozīcija transkripcijā.

Trešajā iterācijas posmā pozīcija transkripcijā ir pirms fonēmas „l”. Sistēma šajā posmā ir atradusi difonu „_l_AX_”. Šajā situācijā redzams, ka atrastais transkripcijas fragments beidzas ar fonēmas „AX” sākuma pusi, jo transkripcijas fragments beidzas ar simbolu „_”. Ja fragments būtu beidzies ar pilnu fonēmu „AX”, tad nebūtu nepieciešama vairs neviena iterācija, bet tagad nepieciešams piekārtot vēl vienu pusfonēmu, respektīvi, pusfonēmu „_AX”. Tāpēc jaunā pozīcija tiek noteikta pirms fonēmas „AX”.

Pēdējā iterācijas posmā tiek atrasta pusfonēma „_AX”, un tiek noteikta jauna pozīcija transkripcijā, kas ir aiz fonēmas „AX”. Līdz ar to iterācija tiek pārtraukta.

Kā redzams, segmentēšanas procesā tika iegūti četri jauni transkripciju fragmenti, kuriem tika piekārtoti to reprezentējošie audio fragmenti. Visi šie dati tiek atgriezti fonētiskā audio datu saraksta veidā.

3.2.5.3. Fonētiskās bibliotēkas kļūdu apstrāde

Fonētiskā bibliotēka izraisa fonētiskās bibliotēkas kļūdas (*PhoneticLibraryException*) gadījumā, ja nav iespējams ielasīt no datu bāzes fonētiskās bibliotēkas ierakstu sarakstu. Kļūda tālāk tiek apstrādāta runas sintēzes sistēmas dzinī, kas nosaka, ka konkrētā izraisītā kļūda ir kritiska, un tālāka sistēmas darbība nav iespējama.

Fonētiskā bibliotēka apstrādā fonētisko audio datu saraksta elementa kļūdu. Šī kļūda tiek izraisīta, ja nav bijis iespējams veiksmīgi ielasīt audio datus no datu bāzes. Ja kļūda tiek izraisīta, fonētiskā bibliotēka neatgriež prasīto fonētisko audio datu sarakstu konkrētai transkripcijai, bet gan izraisa fonētiskās bibliotēkas kļūdu. Līdz ar to kļūdu gadījumos runā tiks izlaisti vārdi, kurus nebūs iespējams izveidot. Šo kļūdu tālāk apstrādā runas sintēzes dzinis, kas nosaka, ka kļūda ir nebūtiska un tiek apstrādāti atlikušie teksta fragmenti.

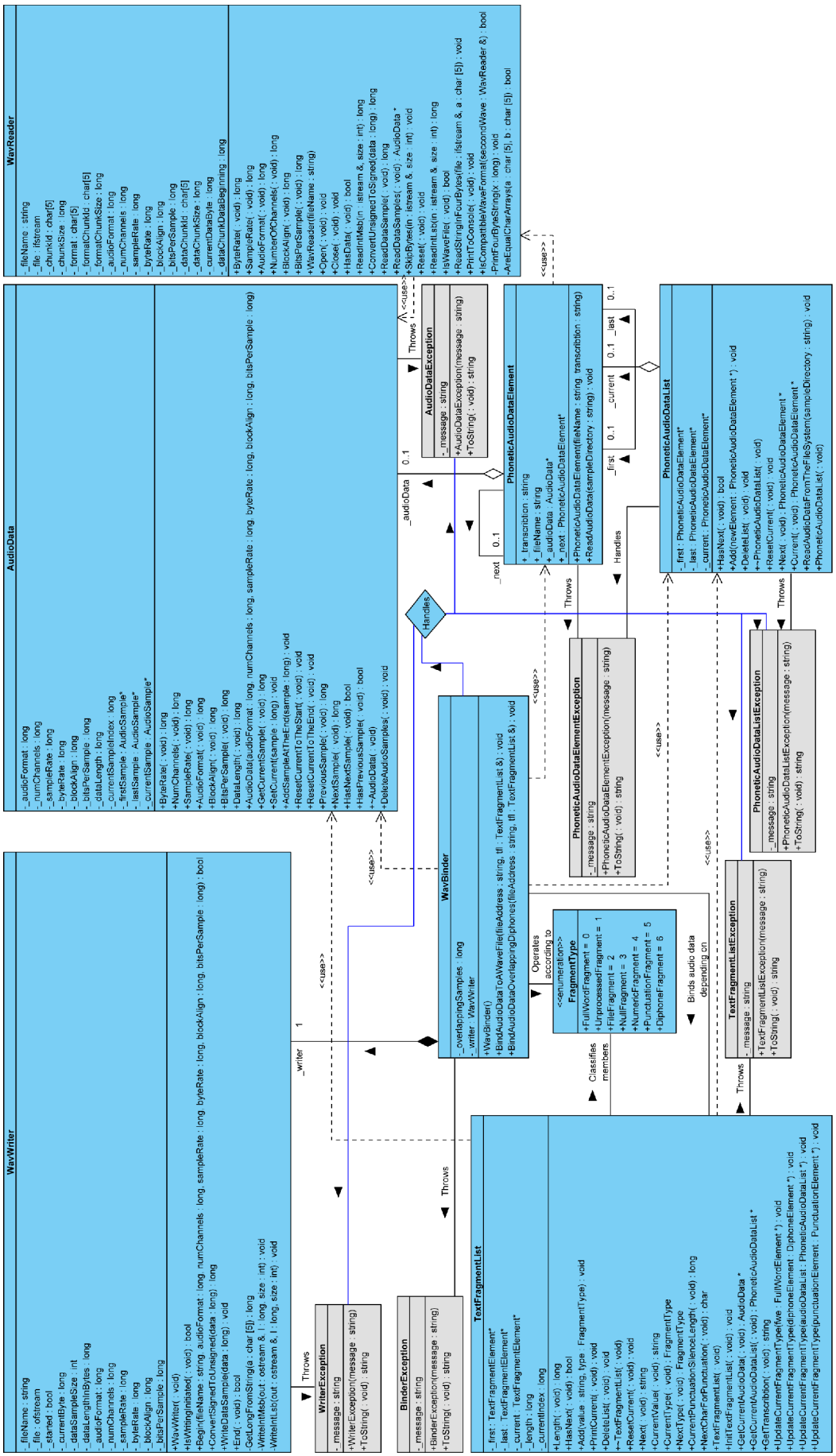
3.2.6. Audio datu savienotājs

Līdz šim padziļināti tika apskatīta augsta līmeņa runas sintēze, kas saistīta ar teksta apstrādi. Tagad uzmanība tiks pievērsta zema līmeņa runas sintēzei, t. i. audio signāla datu apstrādei.

3.2.6.1. Audio datu savienotāja datu struktūra

Audio datu savienotājs (*WavBinder*) ir paredzēts, lai no teksta apstrādes laikā iegūtajiem datiem, izveidotu audio failu, kas reprezentē doto tekstu. Šim nolūkam audio datu savienotājs satur atribūtu, kura funkcijas nodrošina audio failu izveidi, respektīvi audio failu rakstītāju (*WavWirter*). Audio failu rakstītājs no tam padotajiem datiem izveido RIFF WAVE audio failu (paplašinājums „*wav*”) ar tā metožu palīdzību [4, 3].

Kā redzams klašu diagrammā 3.2.6.1.1. attēlā, audio failu lasītāja klase (*WavReader*), audio failu rakstītāja klase un audio datu klase (*AudioData*) satur semantiski līdzīgus datus, kas reprezentē audio signāla parametrus. Šo parametru nosaukumi klasēs var nedaudz atšķirties. Katru klasi un tās struktūru apskatīsim nedaudz vēlāk.



3.2.6.1.1. att. **Audio datu savienotāja klašu diagramma**

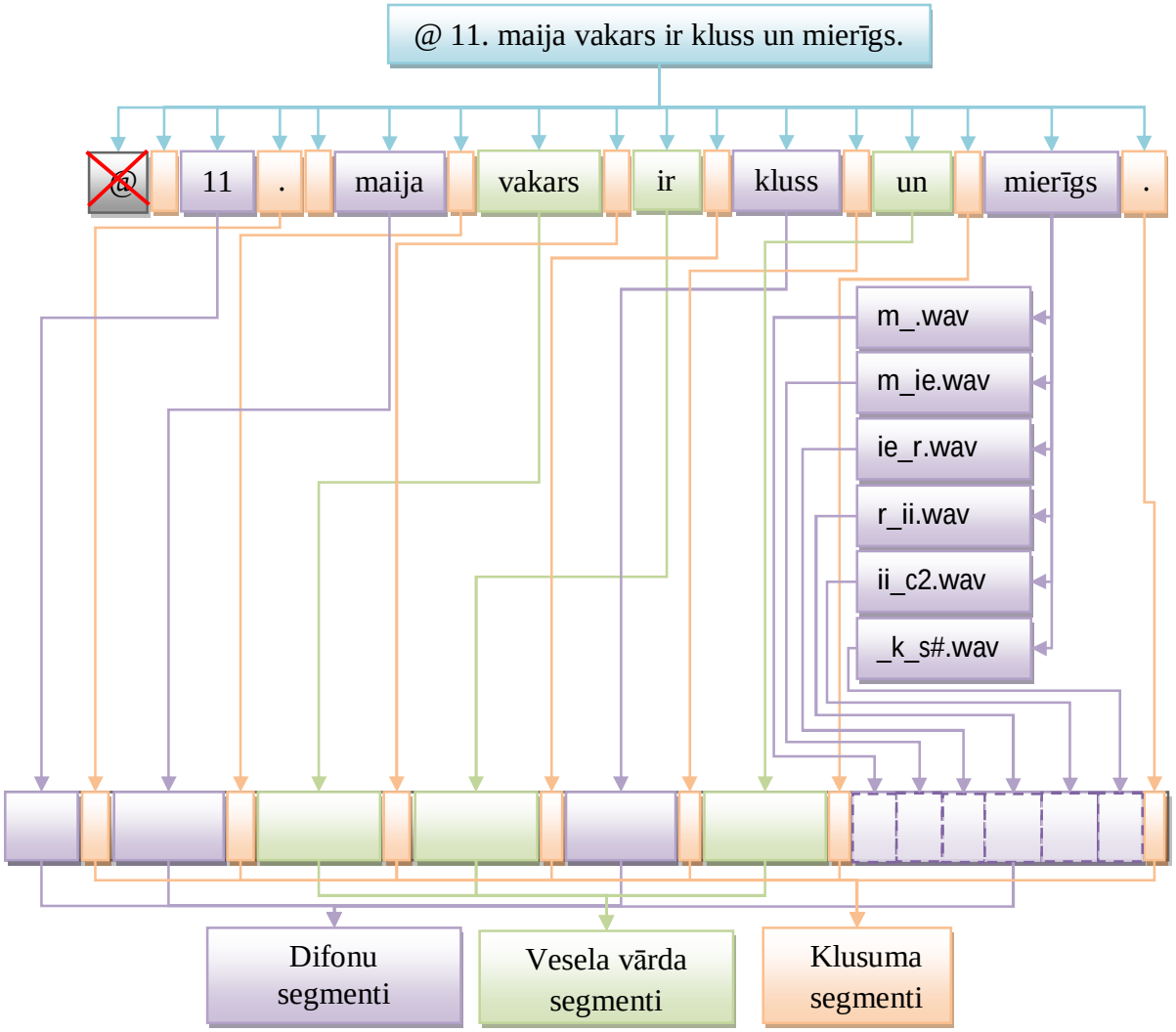
3.2.6.2. Audio datu savienotāja funkcionalitāte

Audio datu savienotāja vienīgais mērķis ir izveidot audio failu. Šī mērķa īstenošanai ir definētas divas publiskas metodes, ja neskaita audio datu savienotāja konstruktoru (*WavBilnder*), kas definē sākotnējās vērtības. Abas metodes nodrošina audio failu veidošanu, bet to algoritmi ir principiāli atšķirīgi.

Pirmā metode (*BindAudioDataToAWaveFile*) nodrošina audio faila veidošanu, kur dažādu audio fragmentu dati tiek vienkārši sakabināti. Metode kā parametrus saņem adresi, kur jāsaglabā izveidotais fails, kā arī teksta fragmentu sarakstu, no kura tiek iegūti audio dati priekš jaunizveidotā faila. Audio datu savienošanas procesā tiks izmantoti tikai tie dati, kas iegūti no audio datu objektiem un pieturzīmes elementiem, kas glabājas teksta fragmentu sarakstā (sk. nodaļu 3.2.2.). Līdz ar to ļoti svarīgi ir teksta fragmentu veidi, no kuriem iespējams uzzināt, kā konkrēto teksta fragmentu ir jāapstrādā un kur tajā ir jāmeklē dati.

Audio datu savienošanai tiek izmantoti sekojoši teksta fragmentu veidi: vesela vārda fragments, difonu fragments un pieturzīmes fragments. No vesela vārda fragmenta atbilstošā vesela vārda elementa tiek iegūts audio datu objekts, kura audio dati tiek saglabāti jaunajā failā. No difonu fragmenta atbilstošā difonu elementa tiek iegūts fonētisko audio datu saraksts, kura elementi satur audio datu objektus, kuru audio dati tiek saglabāti jaunajā failā. No pieturzīmes fragmenta atbilstošā pieturzīmes elementa tiek iegūts pieturzīmes klusuma perioda garums. Klusuma periods nosaka, cik daudz audio datu vērtības (vienādas ar nulli) ir jāsaglabā audio failā konkrētajā pozīcijā.

Metodes algoritms nosaka, ka tiek veikta iterācija pa teksta fragmentiem no teksta fragmentu saraksta. Rezultāta audio fails tiek izveidots tikai gadījumos, ja teksta fragmentu sarakstā vismaz viens elements satur audio datu objektu. Atrodot pirmo teksta fragmentu ar audio datu objektu (difonu fragmentiem vai veselu vārdu fragmentiem) tiek izveidots jauns audio fails, kura formāts atbilst audio datu objektam (iemesls, kāpēc audio datu apstrādes klasēm ir līdzīgi atribūti). Kad fails izveidots, visi audio dati no audio datu objekta tiek saglabāti failā (bet fails netiek aizvērts). Atrodot nākamo audio datu objektu, jauns fails netiek veidots, bet visi audio dati tiek saglabāti iepriekš izveidotajā failā, pievienojot tos faila beigās. Ja pa vidu starp vesela vārda fragmentiem un difonu fragmentiem tiek atrasti pieturzīmju fragmenti, audio failā tiek ierakstīts nepieciešamais klusuma posms. Faila sākumā klusuma posmi nav iespējami, jo nav zināms audio faila formāts, kas atklājas tikai pēc pirmā audio datu objekta apstrādes. Kad visi teksta fragmenti apstrādāti, audio fails tiek aizvērts, vispirms ierakstot attiecīgajās pozīcijās faila segmentu garumu (tiks apskatīts audio failu rakstītāja klases aprakstā).



3.2.6.2.1. att. **Audio faila audio datu struktūras veidošana bez difonu pārklāšanās**

Pirmās audio datu savienošanas metodes izveidoto audio datu struktūra audio failā redzama 3.2.6.2.1. attēlā. Kā redzams, audio fails tiek veidots no iepriekšējās nodaļās apskatītā piemēra teksta fragmentu datiem. Tā kā, veidojot teksta fragmentus, tika izlaisti simboli, kas nav atļauti, pirmais pieturzīmes fragments (atstarpe) netiek iekļauts audio failā. Arī vairāki pieturzīmju fragmenti pēc kārtas netiek iekļauti audio faila audio datos. Tā kā difonu fragmenti patiesībā sastāv no vairākiem mazākiem audio fragmentiem, tad attēlā uzskatāmi parādīts piemērs, ka difonu fragments (attēlā vārdam „mierīgs") tiek segmentēts no sešiem mazākiem audio fragmentiem.

Otrā metode (*BindAudioDataOverlappingDiphones*) pilda tieši tādas pašas funkcijas, kā iepriekšējā metode, bet atšķiras mehānisms, kā tiek savienoti audio dati difonu fragmentiem. Atšķirība datu savienošanā ir tikai difonu fragmentiem, jo starp pārējiem fragmentiem ir klusumu posmi, kas nozīmē, ka nav nepieciešams specifiski savienot datus. Metode difonu fragmentus savieno, tos savstarpēji pārklājot, t. i., tiek sapludinātas kopā divu blakus esošu

difonu audio datu vērtības. Sapludināšanas intervāls ir īpaši noteikts konstants lielums audio datu savienošanas klasē (*_overlappingSamples*).

Lietojot otro metodi tiek uzlabota runas kvalitāte, bet izmaiņas ir visai nelielas (ir labi jāieklausās, lai pamanītu atšķirību), tāpēc kā noklusētā metode datu savienošanai tiek izmantota pirmā metode, kas darbojas ātrāk, jo tai nav nepieciešamas papildus datu kalkulācijas.

3.2.6.3. Audio failu lasītājs

Iepriekšējās nodaļās tika pieminētas datu struktūras, kas nodrošina audio datu glabāšanu atmiņā, kā arī tika pieminētas metodes, kas ielasa audio datus no datu bāzes, bet nevienā no pieminēšanas vietām nebija precīzi izskaidrots, kādi dati tiek ielasīti, un kā tie tiek precīzi glabāti atmiņā. Tāpēc tagad tiks apskatīts audio datu lasītājs.

Lai izveidotu audio failu datu bāzi tika izmantoti tikai RIFF WAVE audio faili, kas visi ir veidoti pēc identiska formāta. Audio failiem ir jābūt veidotiem pēc identiska formāta (datu skaits sekundē, audio kanālu skaits utt.), jo sistēma nenodrošina audio formātu konvertēšanas funkcionalitāti. Audio failu lasītāja funkcijas ir informācijas iegūšana par audio failu formātu, datu daudzumu, kā arī, lai ielasītu atmiņā pašus audio datus izmantojot datu glabāšanai paredzētu datu struktūru – audio datu objektu.

Lai izprastu, kāda ir RIFF WAVE failu struktūra, kā arī, lai izprastu audio failu lasītāja darbības principus, tiks apskatīta viena faila ielase no failu sistēmas.

Lai uzsāktu audio datu ielasi, ar faila atvēršanas metodes (*Open*) palīdzību tiek atvērts audio fails ar failu ievadplūsmas atribūta „_file” palīdzību, un tiek ielasīti audio faila formāta parametri. Tā kā audio fails tiek lasīts kā binārs fails, ir svarīga faila struktūra. Attēlā 3.2.6.3.1. uzrādīta tipiska RIFF WAVE faila struktūra. RIFF WAVE faili sastāv no segmentiem, kas var saturēt citus segmentus [11, 12]. Pašu audio failu definē segments ar nosaukumu „RIFF” (*_chunkId*). Kā redzams attēlā, segmenta nosaukumu veido pirmie četri baiti audio failā. Faila segmentu vēl raksturo segmenta lielums (*_chunkSize*) un segmenta formāts (*_format*), kas nosaka, kāda veida datus satur fails. Katrs parametrs satur četrus baitus.

Iekš faila segmenta iespējams definēt vairākus citus apakšsegmentus. Tipiskam RIFF WAVE failam ir definēti divi apakšsegmenti, bet nav aizliegti vairāk segmenti. Tāpat tipiskam RIFF WAVE failam apakšsegmenti ir sakārtoti pēc principa, ka vispirms definēts faila formāta apakšsegments, un pēc tam definēts audio datu apakšsegments, bet secība var būt arī citādāka.

Pirmais apakšsegments definē audio faila formātu. Segments sastāv no apakšsegmenta nosaukuma (*_formatChunkId*), apakšsegmenta lieluma (*_formatChunkSize*), audio formāta (*_audioFormat* – nosaka formāta apakšegmenta veidu), audio signāla kanālu skaita (*_numChannels*), audio vērtību skaita sekundē (*_sampleRate*), baitu skaita sekundē (*_byteRate*), bloka izlīdzināšanas (*_blockAlign*) un bitu skaita vienā audio vērtībā (*_bitsPerSample*).

Nobīde no faila sākuma	Datu lielums baitos	Datu nosaukums	Datu piemērs
0			
4	4B	Segmenta nosaukums	RIFF
8	4B	Segmenta lielums baitos	207140
12	4B	Segmenta formāts	WAVE
16	4B	Faila formāta apakšsegmenta nosaukums	fmt
20	4B	Formāta apakšsegmenta lielums baitos	16
22	2B	Faila audio formāts	1
24	2B	Audio kanālu skaits	1
28	4B	Audio vērtību skaits sekundē	22050
32	4B	Baitu skaits sekundē	44100
34	2B	Bloka izlīdzināšana baitos	2
36	2B	Biti vienā audio vērtībā	16
40	4B	Audio datu apakšsegmenta nosaukums	data
44	4B	Audio datu apakšsegmenta lielums baitos	207104
	DATA	Audio datu vērtības	...

3.2.6.3.1. att. **Tipiska RIFF WAVE faila struktūra**

Atkarībā no tā, kādas ir ielasīto faila formāta parametru vērtības, tiks lasīti dati no audio datu segmenta. Nepieciešamie parametri pareizai datu ielasei ir: audio kanālu skaits un biti vienā audio vērtībā.

Otrs apakšsegments satur patiesos audio signāla datus. Segments satur nosaukumu „*data*” (*_dataChunkId*), segmenta garumu (*_dataChunkSize*), kā arī datus, kuru garums vienāds ar segmenta garumu bez astoņiem baitiem (nosaukums un garums).

Izsaucot faila atvēršanas metodi tiek ielasīti no faila atmiņā visi tā parametri līdz audio datu vērtībām. Parametri tiek saglabāti audio failu lasītāja attiecīgos atribūtos. Audio datu

vērtību nolasīšanai iespējamas divas dažādas metodes, bet vienlaicīga abu metožu pielietošana nav atļauta, jo metodes pārvieto audio datu lasīšanas galviņu, kā arī atgriež principiāli atšķirīgus rezultātus. Pirmā metode atgriež lietotājam pa vienai audio datu vērtībai katrā izsaukumā (*ReadDataSample*) [4]. Metode paredzēta, ja nepieciešams ielasīt no faila konkrētu daudzumu audio datu vērtības.

Otra metode, kas ielasa audio datu vērtības (*ReadDataSamples*) [4], atgriež audio datu objektu, kas ir lasītā faila pilna reprezentācija atmiņā, t. i., satur visus audio signāla parametrus un visas audio datu vērtības. Parametri, kas tika ielasīti faila atvēršanas laikā, tiek saglabāti jaunizveidotā audio datu objektā. Metode turpina lasīšanu no faila, ielasot visas audio datu vērtības, un saglabājot ielasītās vērtības atmiņā saistīta saraksta veidā, ko veido audio datu vērtības elementi (*AudioSample*), ar audio datu objekta datu pievienošanas metodes palīdzību (*AddSampleAtTheEnd*).

Audio datu saraksts ir divvirzienu saraksts, kur audio datu objekts nodrošina saraksta apstrādes funkcionalitāti (iterēšanu cauri sarakstam, elementu pievienošanu, dzēšanu utt.). Audio datu objekts satur norādes uz saraksta pirmo (*_firstSample*), pēdējo (*_lastSample*) un konkrētajā iterācijas momentā apstrādāto (*_currentSample*) audio datu vērtības elementu.

Kad audio failu lasītāju vairs nav nepieciešams lietot, ir jāaizver dokuments, lai to varētu izmantot atkārtoti. Šo procesu iespējams izdarīt ar aizvēršanas metodi (*Close*). Līdz ar to darbs ar audio failu lasītāju ir identisks, kā ar jebkuru izvades vai ievades plūsmu. Vispirms plūsma tiek atvērta, lai varētu sākt datu ielasi. Pēc tam tiek lasīti nepieciešamie dati, un visbeidzot plūsma tiek aizvērta. Gadījumā, ja plūsma netiktu aizvērta, varētu rasties situācijas, ka nav iespējams nolasīt failus, kuriem plūsmas vēl ir atvērtas, vai arī nav atbrīvotas no atmiņas.

3.2.6.4. Audio failu rakstītājs

Audio failu rakstītājs nodrošina audio failu rakstīšanas funkcionalitāti, un no audio faila lasītāja atšķiras ar to, ka datu plūsma notiek pretējā virzienā, t i., tiek rakstīts failā, nevis lasīti dati. Audio failu lasītājam ir mazāk iespēju, kā audio datu lasītājam, jo rakstītājs izveido tikai viena formāta RIFF WAVE audio failus.

Failu rakstīšana tiek sākta ar jaunā faila izveidošanu un audio faila formāta parametru definēšanu. Šo funkciju nodrošina metode „Begin”, kas izveido failu bināro izvadplūsmu (*_file*), un ieraksta failā 44 baitus ar tukšām vērtībām, respektīvi, nullēm. 44 baitus nepieciešams izlaist, jo tik aizņems audio faila segmenti pirms audio datu vērtībām, pie kam, patiesās parametru vērtības (segmentu izmēri) pilnībā varēs noteikt tikai tad, kad failā visas audio datu vērtības būs jau ierakstītas. Metode kā parametrus saņem arī faila audio formāta

parametrus, kas tiek saglabāti audio faila rakstītāja atribūtos, līdz tiek veidots jauns fails, vai arī tiek atbrīvots no atmiņas viss audio failu rakstītājs. Šie parametri kopā ar tiem, kas tiks iegūti, rakstot datus failā, tiks ierakstīti jaunā audio faila sākuma daļā (44 baitos).

Metode nodrošina arī automātisku atribūtu vērtību izrēķināšanu. Metode no bloka izlīdzināšanas (*_blockAlign*) un audio kanālu skaita (*_numChannels*) izrēķina, cik baitus aizņems katra audio datu vērtība (*_sampleSize*), kā arī pazīmei, kas norāda uz to, vai fails izveidots, un tajā var rakstīt datus, piešķir apstiprinošu vērtību (*_started*).

Kad audio fails veiksmīgi izveidots, var sākt audio datu vērtību rakstīšanu, ko nodrošina metode „*WriteDataSample*”. Metode kā parametru saņem vienu audio datu vērtību, kas tiek saglabāta failā iekš tieši tik baitiem, cik vienai datu vērtībai ir atvēlēts. Tā kā RIFF WAVE faila struktūra nosaka to, ka ir jābūt informācijai par to, cik liels ir katrs faila segments, katru reizi, izsaucot datu rakstīšanas metodi, tiek palielināta vērtība atribūtam, kas domāts audio datu garuma glabāšanai (*_dataLengthInBytes*).

Kad visi audio dati ir ierakstīti failā, nepieciešams aizpildīt faila sākuma 44 baitus, kas sadala failu segmentos, un nosaka, cik liels ir katrs segments. Faila sākuma aizpildīšanu un faila aizvēršanu nodrošina funkcija „End”.

Detalizētāka audio datu savienotāja, audio failu lasītāja, audio failu rakstītāja un audio datu objekta datu struktūra parādīta klašu diagrammā 3.2.6.1.1. attēlā.

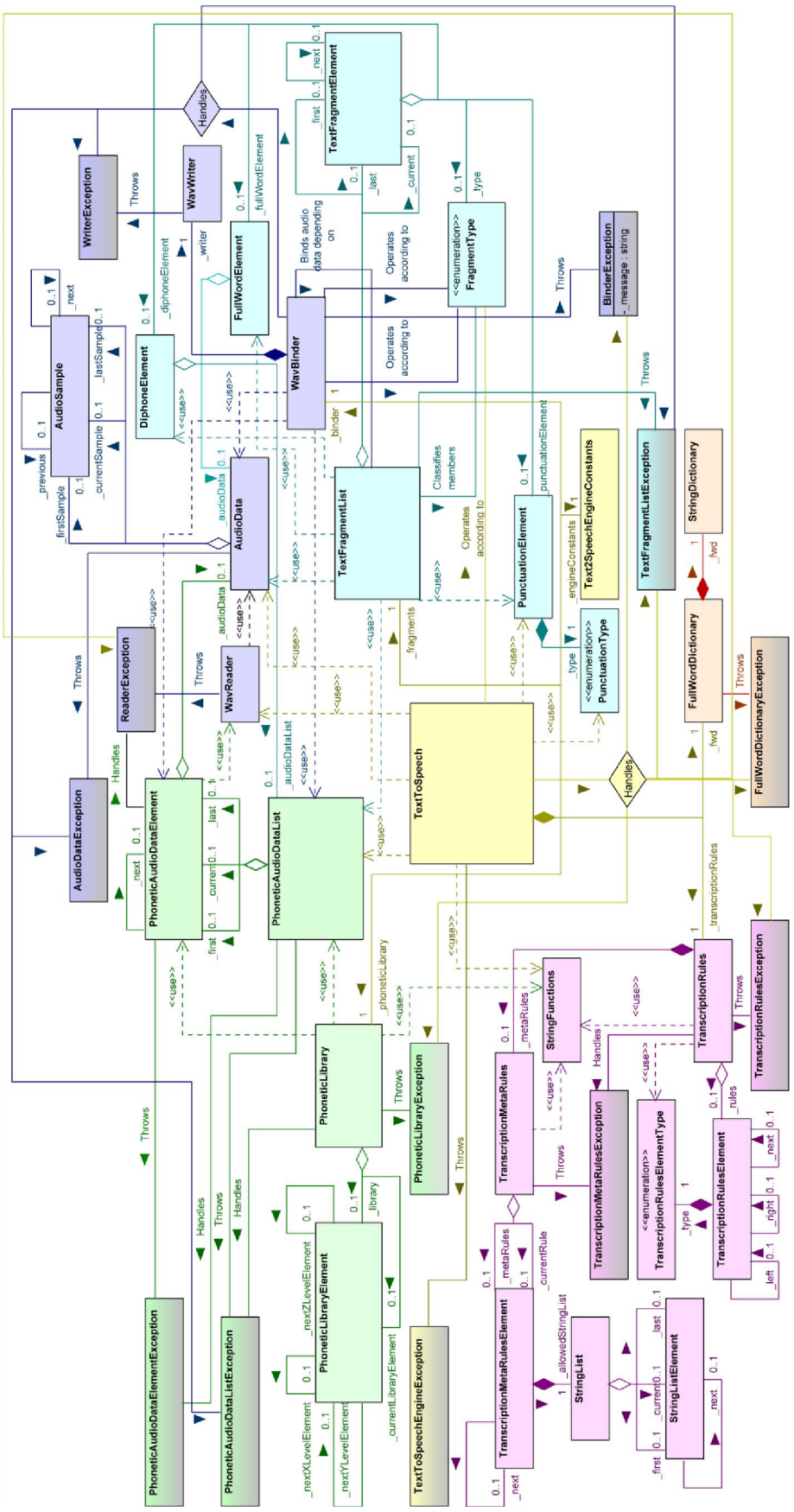
3.2.7. Konkatenatīvas runas sintēzes sistēmas kopējā klašu struktūra

Lai būtu uzskatāmāk redzams sistēmas apjoms un sarežģītība, visi iepriekš aprakstītie moduļi tika savienoti kopā, izveidojot Konkatenatīvas latviešu valodas runas sintēzes sistēmas klašu diagrammu 3.2.7.1. attēlā. Lai diagramma būtu uzskatāma un saprotama, klasēm netiek uzrādīti ne atribūti, ne metodes, kā arī tika iekrāsoti semantiski atšķirīgie runas sintēzes sistēmas moduļi.

Rozā krāsā ir iekrāsoti transkribēšanas likumi, meta likumi un ar tiem saistītās klases. Zaļā krāsā ir iekrāsota fonētiskā bibliotēka un ar to tieši saistītās klases. Violetā krāsā ir iekrāsotas ar audio datu savienotāju saistītas klases, kā arī ar audio datu failu ielasi un rakstīšanu saistītas klases. Zilā krāsā ir iekrāsots teksta fragmentu saraksts un ar to tieši saistītas klases. Sarkanā krāsā ir iekrāsota veselu vārdu vārdnīca un ar to saistītas klases. Visbeidzot dzeltenā krāsā ir iekrāsots centrālais sistēmas modulis, t. i., runas sintēzes sistēmas dzinis un ar to tieši saistītas klases.

Kā redzams diagrammā, klases, kas definē kļūdas, ir iekrāsotas ar krāsu pāreju, kur vienas puses krāsa ir pelēka, bet otra krāsa ir attiecīgā moduļa krāsa, kuram kļūda pieder.

Lai klašu diagrammā būtu viennozīmīgi saprotams, kurām klasēm atbilst konkrētas asociācijas, tās ir iekrāsotas attiecīgos toņos kā klase, no kuras asociācija iziet. Ja arī vietām asociācijas krustojas, jāņem vērā, ka krustojumos asociāciju virziens netiek mainīts, t. i., līnijas krustojumos netiek lauztas par 90 grādiem.



3.2.7.1. att. **Konkatenatīvas latviešu valodas runas sintēzes sistēmas klašu diagramma**

4. SISTĒMAS REALIZĀCIJA

„Konkatenatīvas latviešu valodas runas sintēzes sistēmas" izstrāde tika uzsākta 2007. gada 1. oktobrī Latvijas Universitātes Matemātikas un informātikas institūta Mākslīgā intelekta laboratorijā. Sistēmas izstrādē tika iesaistīti trīs cilvēki: bakalaura darba autors, Dr.philol. Ilze Auziņa un Latvijas Universitātes Fizikas un Matemātikas fakultātes ceturtā kursa studente Sandra Lazukina.

4.1. Atbildības sfēras sistēmas izstrādē

Tā kā runas sintēzes sistēmas izstrādei nepieciešamas zināšanas no tādām sfērām kā valodniecība (it īpaši nepieciešamas zināšanas par fonoloģiju), matemātiskā statistika un datorzinātnes, izstrādē tika iesaistīti cilvēki, kas orientējas šajās sfērās.

Darba autors ir atbildīgs par sistēmas projektēšanu, sistēmas funkcionalitātes nodrošināšanu, sistēmas datu bāzes struktūru (nevis saturu), kā arī par saskarnes izstrādi sistēmas testēšanas nolūkiem.

Dr.philol. Ilze Auziņa ir atbildīga par datu bāzes satura izveidi, t. i., veselu vārdu vārdnīcas audio failu segmentēšanu no runas ierakstiem, veselu vārdu vārdnīcas saraksta izveidi, transkribēšanas likumu izveidi un transkribēšanas meta likumu izveidi atbilstoši datu bāzes struktūrai, fonētiskās bibliotēkas audio fragmentu segmentēšanu, t. i., difonu datu bāzes izveidi, lai sistēma būtu spējīga pilnvērtīgi pildīt tās paredzētās funkcijas. Papildus tam I. Auziņa ir atbildīga par datu bāzes kvalitāti, t. i., lai neveidotos situācijas, ka, sintezējot runu, veidojas iztrūkumi (kāda skaņa izlaista).

Sandra Lazukina ir atbildīga par datu bāzes satura daļēju izveidi, t. i., veselu vārdu vārdnīcas audio failu segmentēšanu no runas ierakstiem, fonētiskās bibliotēkas audio fragmentu segmentēšanu, kā arī statistiskiem pētījumiem, kas nepieciešami, lai veiksmīgāk aizpildītu veselu vārdu vārdnīcu.

Sistēmas pilnvērtīga izstrāde nebūtu iespējama bez iesaistītajām personām no visām trim sfērām.

4.2. Sistēmas izstrādes posmi

Sistēmas izstrāde tika veikta pa trīs mēnešu posmiem, pēc katra posma veidojot atskaites par padarīto darbu klientam, respektīvi SIA „Lattelecom BPO". Līdz 2008. gada maijam tika veiksmīgi nobeigti divi posmi.

4.2.1. Pirmais sistēmas izstrādes posms

Pirmajā sistēmas izstrādes posmā tika uzsākts darbs pie sistēmas arhitektūras projektēšanas, kā arī tika uzsākts darbs pie digitālo signālu, respektīvi, RIFF WAVE failu apstrādes bibliotēkas.

Lai sistēma spētu funkcionēt, bija nepieciešami runas ieraksti, no kuriem varētu segmentēt difonus, kā arī izmantot veselus vārdus. Šī mērķa sasniegšanai profesionālā ierakstu studijā (paralēli sistēmas arhitektūras izstrādei) tika ierakstīti vārdi. Vārdu saraksts tika iepriekš speciāli sagatavots, lai nodrošinātu visu iespējamo latviešu valodas difonu klātbūtni ierakstos. Pēc runas ierakstīšanas tika sākts darbs pie veselu vārdu segmentēšanas.

Pirmā posma nobeigumā tika izstrādāts runas sintēzes sistēmas dzinis ar iespēju apstrādāt pieturzīmes un veselus vārdus, rezultātā izveidojot audio failu, kas satur runu.

Pirmajā posmā izstrādātajai sistēmai bija viens ierobežojums – sintezēt bija iespējams tikai tos vārdus, kas bija pieejami veselu vārdu vārdnīcā.

4.2.2. Otrais sistēmas izstrādes posms

Otrajā sistēmas izstrādes posmā tika izstrādāti teksta transkribēšanas likumu un fonētiskās bibliotēkas moduļi kā arī uzlaboti, visi iepriekš izstrādātie moduļi, lai nodrošinātu pilnvērtīgāku moduļu integrāciju, kā arī, lai paplašinātu funkcionalitāti, nodrošinot ne tikai veselu vārdu sintēzi, bet arī difonu fragmentu runas sintēzi.

Paralēli ar sistēmas risinājumu izstrādi tika papildināta arī datu bāze ar transkribēšanas likumiem, meta likumiem un fonētisko bibliotēku (difoniem un vārdu fragmentiem). Līdz ar to šajā posmā izstrādātā sistēma spēj nodrošināt pilnīgi visu latviešu valodas vārdu sintēzi.

Sistēmas ierobežojumi otrā posma beigās ir saistīti ar nepietiekami gludu difonu konkatenāciju (prosodija vēl netiek pietiekami labi modelēta). Šo problēmu paredzēts risināt nākošajā sistēmas izstrādes posmā.

4.3. Sistēmas izstrādes vides un vadlīnijas

Sistēma izstrādāta programmēšanas valodā C++ [5], lietojot integrēto izstrādes vidi (*integrated development environment - IDE*) „*Code::Blocks 8.02*”. Sistēma izstrādāta pēc objektorientētās programmēšanas principiem, kas nodrošina sistēmas algoritmu atkārtotu izmantošanu citiem mērķiem.

Izstrādes laikā tika pievērsta pastiprināta uzmanība, lai tiktu radīts kvalitatīvs un lasāms kods. Šim nolūkam tika ievērotas vairākas būtiskas vadlīnijas:

- Mainīgo, funkciju un klašu nosaukumi satur semantiski atbilstošus nosaukumus, tādējādi nodrošinot algoritmu vieglāku lasāmību.

- Visas klases, klašu metodes un klašu atribūti satur skaidrojumu, par konkrēto objektu komentāra veidā tieši virs konkrētā objekta definīcijas.
- Metodes tiek semantiski komentētas, lai būtu iespējams izsekot datu plūsmai un, lai metodes algoritms būtu viegli uztverams.
- Kods tiek strukturēts ar atkāpēm (četriem atstarpes simboliem) situācijās, kur tiek lietoti cikli, nosacījumi, metodes, vārdkopas.
- Klašu atribūtu nosaukumi tiek definēti ar simbolu „_" nosaukuma sākumā (piemēram, „*_transcription*").
- Lokālie mainīgie un klašu atribūti tiek rakstīti ar mazo sākuma burtu, bet, ja objekts satur vairāk par vienu vārdu nosaukumā, katrs nākamais vārds tiek uzsākts ar lielo sākuma burtu (piemēram, „*_phoneticLibrary*", „*tempRule*").
- Metožu nosaukumi un klašu nosaukumi tiek rakstīti ar lielo sākuma burtu (piemēram, „*TranscribeAString*").

Audio fragmentu segmentēšanai tika izmantotas tādas audio failu apstrādes programmas kā „*Sony Sound Forge 4*", „*WavePad 3*" un „*Audacity*".

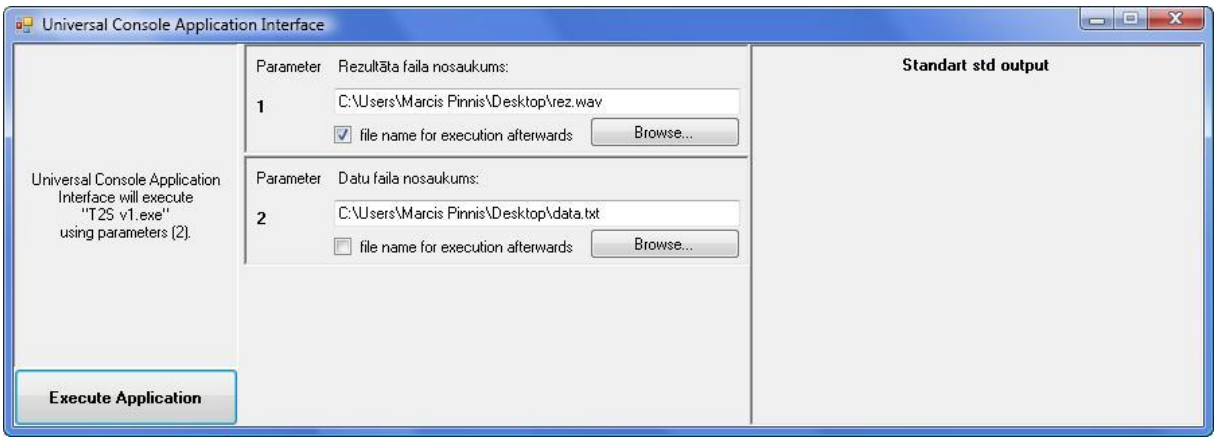
Saskarne, kas nodrošina iespēju izmantot runas sintēzes sistēmu, izstrādāta valodā Visual C# ar integrētās izstrādes vides „*Visual Studio 2008*" palīdzību.

4.4. Runas sintēzes sistēmas saskarne

Tā kā „Konkatenatīvai latviešu valodas runas sintēzes sistēmai" iespējami vairāki implementēšanas mehānismi (sk. nodaļu 2.2.), kuru izmantošana ir atkarīga no sistēmas pielietojuma mērķa, testēšanas nolūkos tika izstrādāta konsoles lietojumprogramma, kas, saņemot attiecīgus parametrus (adresi failam, kur saglabāt rezultātu un adresi failam, no kura jāielasa sintezējamais teksts), veic sintēzes funkcijas. Līdz ar to radās nepieciešamība izstrādāt saskarni, kas ļautu izsaukt konsoles lietojumprogrammu un iespēju robežās demonstrēt rezultātu (izsaukt procesu, kas atskaņo sintezēto tekstu).

4.4.1. Saskarnes apraksts

Iepriekš definētā mērķa sasniegšanai tika radīta universāla *Windows* lietojumprogramma, kas var izsaukt konsoles lietojumprogrammas ar neierobežotu skaitu parametriem. Izveidotās saskarnes nosaukums ir „*Universal Console Application Interface*" (sk. 4.4.1.1. attēlu), kas latviski nozīmē: „Universāla konsoles lietojumprogrammu saskarne."



4.4.1.1. att. **Saskarnes formas paraugs**

Saskarne ir universāla, jo nav paredzēta tikai runas sintēzes sistēmas izsaukšanai. To var viegli ar konfigurācijas faila palīdzību piesaistīt citām konsoles lietojumprogrammām. Tā kā runas sintēzes sistēmas funkcionalitātes testēšana tika veikta ar operētājsistēmas „*Microsoft Windows Vista*" palīdzību, saskarne tika veidota ar šai sistēmai atbilstošu programmatūras izstrādes vidi, proti, „*Visual Studio 2008*". Līdz ar to saskarne ir izstrādāta programmēšanas valodā C#.

4.4.2. Saskarnes konfigurēšana un izmantošana

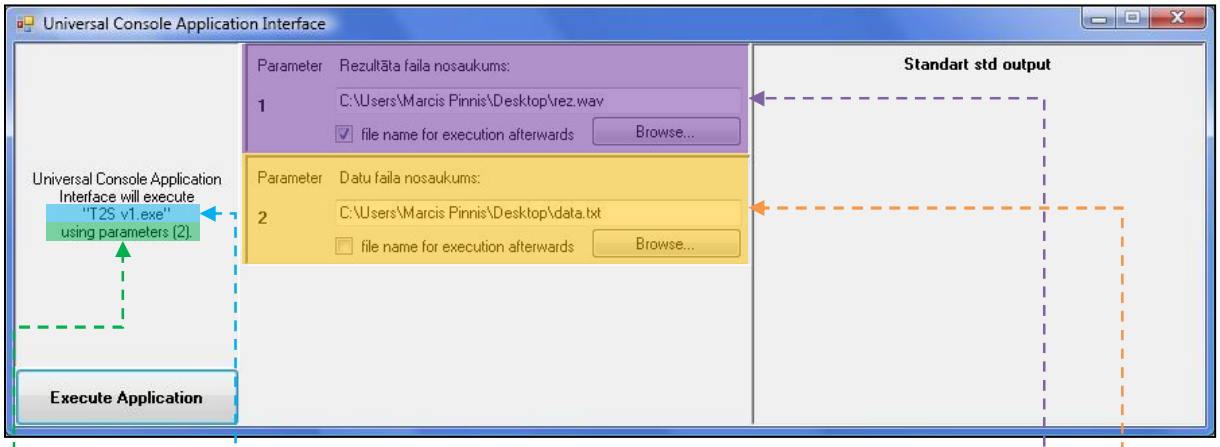
Saskarnes universālo pielietojumu nosaka iespēja saskarni brīvi konfigurēt. Pirmajā pielikumā pieejamā saskarne ir nokonfigurēta, lai izsauktu runas sintēzes sistēmu, kas atrodas tieši tajā pašā direktorijā, kā saskarne. Lai vieglāk izprastu konfigurēšanas principus, tiks apskatīts, kā tiek izveidota saskarnes galvenā forma, balstoties uz konfigurācijas failā „*Universal Console Application Interface.exe.config*" definētajiem parametriem.

Vispirms jāpievērš uzmanība . attēla apakšējai daļai, kur ir redzams konfigurācijas fails, kas satur saskarnes konfigurāciju runas sintēzes sistēmas izsaukumam. Konfigurācijas failā ir iespējamas divu veidu atslēgas: - statiskas un dinamiskas.

Pie statiskām atslēgām pieder konsoles lietojumprogrammas nosaukums (*ApplicationName*), parametru skaits (*ParameterCount*), kā arī direktorijas adrese, kur atrodas konsoles lietojumprogramma. Šīs grupas atslēgas konfigurācijas failā nedrīkst atkārtoties. No statiskajiem parametriem iespējams secināt, ka saskarne izsauks lietojumprogrammu „*T2S v1.exe*" ar diviem parametriem un meklēs šo programmu tieši tajā pašā direktorijā, kur atrodas pati saskarne. Uz identisku lietojumprogrammas direktoriju norāda atslēgas vērtība „*[SAME]*". Šīs vērtības vietā varēja būt jebkādas direktorijas adrese no failu sistēmas.

Pie dinamiskām atslēgām savukārt pieder atslēgu grupa, kas definē vienu konsoles lietojumprogrammas parametru. Šīs atslēgas ir: parametra nosaukums (*ParameterName*),

parametra noklusētā vērtība (*ParameterDefaultValue*), pazīme, vai parametra vērtība ir kāda adrese, kas norāda uz konkrētu failu, tādējādi ļaujot pārlūkot failu sistēmu (*ParameterBrowsable*), pazīme, vai parametra vērtība, ja norāda uz failu norāda uz jaunizveidojamu failu, vai tā norāda uz kādu esošu failu sistēmā (*ParameterIfBrowsableIsSavable*), un pazīme, vai parametra vērtība norāda uz failu, kas ir jāatver pēc konsoles lietojumprogrammas izpildes (*IsParameterExecutable*).




```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <!-- ServerCode - servera kods -->
    <add key="ApplicationName" value="T2S v1.exe"/>
    <add key="ParameterCount" value="2"/>
    <add key="ParameterName1" value="Rezultāta faila nosaukums:"/>
    <add key="ParameterDefaultValue1" value="C:\Users\Marcis
Pinnis\Desktop\rez.wav"/>
    <add key="ParameterBrowsable1" value="true"/>
    <add key="ParameterIfBrowsableIsSaveable1" value="true"/>
    <add key="IsParameterExecutable1" value="1"/>
    <add key="ParameterName2" value="Datu faila nosaukums:"/>
    <add key="ParameterDefaultValue2" value="C:\Users\Marcis
Pinnis\Desktop\data.txt"/>
    <add key="ParameterBrowsable2" value="true"/>
    <add key="ParameterIfBrowsableIsSaveable2" value="false"/>
    <!-- The [SAME] parameter indicates that the directory of the application
is the same as the directory of the interface -->
    <add key="ExecutionAddressForApplication" value="[SAME]"/>
  </appSettings>
</configuration>
```

4.4.2.1. att. **Saskarnes formas ģenerēšana no konfigurācijas datiem**

Otrās grupas atslēgas ir dinamiskas, jo to nosaukumu beigās ir jāpievieno indekss, kas norāda, kuram parametram atslēga atbilst. Attēlā redzams, ka konfigurācija atbilst diviem parametriem, jo parametru skaits ir vienāds ar diviem, kā arī parametru dinamisko atslēgu grupas ir divas (Piemēram, ir divas atslēgas parametru nosaukumiem – „*ParameterName1*”

un „*ParameterName2*”). No dinamiskajām atslēgām obligāta ir tikai parametra nosaukuma atslēga. Pārējās atslēgas var arī nebūt definētas.

Apskatot 4.4.2.1. attēla augšējo daļu, redzams, kā konfigurācijas failā definētās atslēgas ietekmē saskarnes formas datus un uzbūvi. Atkarībā no tā, cik daudz parametru būs definēti konfigurācijas failā, tik daudz lietotāju kontroles (*User Control*) objekti tiks izveidoti parametru vērtību definēšanai.

Kad konfigurācijas fails aizpildīts atbilstoši prasībām, var sākt lietot saskarni. Lai izmantotu saskarni runas sintēzes funkciju nodrošināšanai, nepieciešams ievadīt prasītās adreses abiem parametriem, respektīvi, adresi rezultāta audio failam (paplašinājumam obligāti jābūt „*wav*”) un adresi datu failam, kur glabājas transkribējamais teksts. Ja pie rezultāta parametra bija atzīmēts, ka pēc konsoles lietojumprogrammas izpildes nepieciešams atvērt rezultāta failu, rezultāts tiks atskaņots ar sistēmas noklusēto audio atskaņotāju. Pretējā gadījumā, pēc konsoles lietojumprogrammas izpildes, runa būs saglabāta rezultāta failā.

Pirmajā pielikumā atrodamā ir pievienota „Konkatenatīvās latviešu valodas runas sintēzes sistēmas” aktuālā versija un saskarne, kas ir nokonfigurēta, lai izmantotu runas sintēzes funkcijas.

4.5. Sistēmas izstrādes problēmas un paredzamie risinājumi

Izstrādājot konkatenatīvo latviešu valodas runas sintēzes sistēmu, radās vairākas nozīmīgas problēmas, kas būtu jārisina nākošajos sistēmas izstrādes posmos, lai nodrošinātu kvalitatīvu sistēmas izveidi.

Pirmā problēma, ar kuru autors saskārās, izstrādājot sistēmu, ir difonu fragmentu nepietiekami gludā savienošanās runas sintēzes laikā. Lai gan tika izstrādāts algoritms, kas daļēji novērš fragmentu savienošanas problēmas, fragmentus pārklājot vienu otram virsū, tas tomēr ir nepietiekams, lai pilnvērtīgi nodrošinātu runas prosodijas modelēšanu un pilnvērtīgi uztveramas runas izveidošanu. Līdz ar to, runas uztveramības nodrošināšanai nepieciešams turpmākajos sistēmas izstrādes posmos rast risinājumus audio signāla normalizēšanai, tādējādi atrisinot negludo difonu savienošanas problēmu.

Otra problēma, kas tika atklāta, ir saistīta ar transkribēšanas likumu kļūdainu tekstu transkribēšanu situācijās, kad nav viennozīmīgi skaidrs, kāda skaņa piekārtojama vārda konkrētā pozīcijā. Tā kā latviešu valodā daži burti var apzīmēt vairākas skaņas vienlaicīgi, dažos latviešu valodas vārdos skaņu pielietojumi var atšķirties atkarībā no konteksta. Piemēram, burtam „e” atbilst divas skaņas: šaurais un platais „e”. Kā piemēru var apskatīt teikumu: „Es šodien ēdu, bet vakar ēdu.” Kā redzams, vienam un tam pašam vārdam „ēdu”

atšķiras skaņas, kas tiek izmantotas atkarībā no darbības vārda laika. Diemžēl vārda laika formu nav iespējams noteikt, neveicot teikuma semantisko analīzi, tādējādi nosakot, kāda laika forma patiesībā bija lietota vārdam „ēdu". Šīs problēmas risināšanai sistēmā būtu nepieciešams vārdu semantiskās analīzes modulis, kura rezultātus varētu izmantot transkribējot tekstu, vai arī ieviešot izņēmumsituāciju vārdnīcu. Ne vienmēr vārdu laika forma var pateikt priekšā, kāda skaņa konkrētā pozīcijā izmantota, piemēram, burtam „o" atbilst trīs skaņas. Vārdos „bokss", „ola" un „futbols" parādās visas trīs skaņas, bet algoritmiski nav iespējams izveidot likumus, kas viennozīmīgi noteiktu, kāda skaņa konkrētā pozīcijā lietojama. Šajā gadījumā arī vārda locījumi nepalīdz izvēlēties pareizo skaņu. Lai novērstu šo problēmu, nepieciešams izstrādāt izņēmumsituāciju vārdnīcu, kurā būtu iekļauti to grupu vārdi, kuriem nav iespējams noteikt, kāda skaņa lietojama. Sistēmas izstrādes nākošajā (trešajā) posmā ir paredzēts ieviest izņēmumsituāciju apstrādes moduli.

Trešā problēma, kas tika atklāta un tiks risināta nākošajā sistēmas izstrādes posmā ir datu bāzes audio datu ielases un audio datu rakstīšanas nepietiekamais ātrums. Lai nodrošinātu pilnvērtīgu sintēzi, nepietiek tikai ar pareizu sintēzi. Sintēzei jābūt veiktai pietiekami ātrā laika intervālā. Uz doto brīdi sistēmas apstrādes ātruma lēnākais punkts ir datu ielase un datu rakstīšana failu sistēmā. Līdz šim šai problēmai netika pievērsta uzmanība, jo svarīgāk bija nodrošināt runas sintēzes funkcionalitāti, bet līdz ko šī funkcionalitāte tiks pilnvērtīgi nodrošināta, tiks uzsākts darbs pie sistēmas vispārējas optimizācijas.

Ceturtā problēma ir saistīta ar datu bāzes kvalitāti, respektīvi, veselu vārdu fragmentiem ir pārāk izteikts vārda sākuma uzsvars. Līdz ar to, savienojot vārdus, runa nav plūstoša un pietiekami labi uztverama, jo katrs vārds ir ar izteiktu uzsvaru. Iemesls šai problēmai ir nepietiekami kvalitatīvi ierakstīti runas fragmenti. Risinājums šai problēmai varētu būt atkārtota runas fragmentu ierakstīšana ierakstu studijā, vai arī esošo fragmentu normalizēšana, tādējādi novēršot radušās nepilnības.

Tā kā sistēmas izstrāde vēl nav beigusies, ir loģiski, ka ir problēmas, kas ir jārisina. Iepriekš minētās problēmas ir nozīmīgas, bet nav neatrisināmas. Tāpēc sistēmas izstrāde tiek turpināta – lai nodrošinātu kvalitatīva rezultāta iegūšanu.

NOBEIGUMS UN SECINĀJUMI

Bakalaura darba ietvaros tika izstrādāta Konkatenatīvas latviešu valodas runas sintēzes sistēmas arhitektūra un algoritmu bibliotēka, kas nodrošina runas sintēzes funkcionalitāti. Darba rezultātā uzrakstītais izejas kods atbilst kvalitātes, t. i., lasāmības, atkārtotas izmantojamības un veiktspējas prasībām.

Izstrādātā sistēma nodrošina pilnīgi visu latviešu valodas vārdu runas sintēzi, tādējādi neizvirzot kādus ierobežojumus saistībā ar leksikonu. Sistēmas pielietotā runas sintēzes metodoloģija, izmantojot gan veselus vārdus, gan difonus, gan garākus nepilnu vārdu fragmentus, nodrošina pilnvērtīgu un tajā pašā laikā arī uztveramu runas sintēzi latviešu valodai.

Lai gan darbs pie sistēmas izstrādes tika uzsākts tikai 2007. gada oktobrī, tad līdz šim ir sasniegti augsti rezultāti, jo sistēma ir darboties spējīga un pilda visas paredzētās funkcijas. Tomēr sistēmai nepieciešama pilnveidošana un tālāka attīstība, lai sasniegtu runas kvalitātes atbilstību līdzīgām sistēmām, kas izstrādātas angļu valodai.

Sistēmas turpmākai attīstībai nākošajos izstrādes posmos tiks rasti risinājumi veiktspējas palielināšanai (galvenokārt datu ievades un izvades darbībām) un runas signāla kvalitātes palielināšanai. Runas signāla kvalitāti paredzēts paaugstināt, veidojot valodas izņēmumu apstrādes moduli, kas daļēji atrisinās problēmu saistībā ar burtu daudznozīmību (sk. nodaļu 4.5.), pielietojot audio signālu filtrus, kas normalizē skaņu, reducējot skaņu pārklājumu trokšņus. Visbeidzot paredzēts uzlabot datu bāzes datu kvalitāti, papildinot likumus, manuāli normalizējot skaņu fragmentus un papildinot veselu vārdu vārdnīcu.

PATEICĪBAS

Konkatenatīvas latviešu valodas runas sintēzes sistēmas izstrāde nebūtu iespējama bez projektā iesaistīto personu sadarbības, tāpēc pateicība tiek izteikta:

- Ilzei Auziņai par lielisku sadarbību projekta izstrādes laikā un sistēmas datu bāzes kvalitatīvu pilnveidošanu;
- Andrejam Spektoram par doto iespēju piedalīties runas sintēzes sistēmas izstrādē, kā arī par projekta izcilu vadīšanu;
- Sandrai Lazukinai par kvalitatīvu datu bāzes pilnveidošanu;
- Ralfam Berzinskim par padomiem runas sintēzes problēmu risināšanā.

IZMANTOTĀ LITERATŪRA UN AVOTI

1. **Auziņa, I.**, „Latviešu valodas izrunas datormodelēšana", promocijas darbs, Latvijas Universitāte, Rīga, 2007. 32 lpp.
2. **Auziņa, I.**, „Latviešu valodas grafēmas-fonēmas atbilsmju likumu sistēma", Latvijas Zinātņu Akadēmijas Vēstis, 2004, A daļa, 58. sējums, Nr. 3, 11.–18. lpp.
3. **Embree, P.M.**, **Danieli, D.**, *C++ Algorithms for Digital Signal Processing*, Prentice Hall, 1999, p. 140-165.
4. **Kientzle, T.**, *A Programmer's Guide to Sound*, Addison Wesley, 1998, 464 p.
5. **Kohl, N.**, „*C++ Referrence*", [tiešsaiste] – [atsauce 21.05.2008.]. Pieejams: http://www.cppreference.com.
6. **Lemmetty, S.**, „Review of Speech Synthesis Technology". master's thesis, Dept. Electrical and Communications Engineering, Helsinki University of Technology, 1999, 104 p.
7. **Muller, P.**, „*Introduction To OOP Using C++*", [tiešsaiste] – [atsauce 21.05.2008]. Pieejams: http://oopweb.com/CPP/Documents/Intro2OOP/VolumeFrames.html.
8. **Štrāls, I.**, „Latviešu valodas runas sintezators", kursa darbs, LU Fizikas un matemātikas fakultāte, Latvijas Universitāte, Rīga, 2007. 32 lpp.
9. *Extensible Markup Language (XML) 1.0 (Fourth Edition)* [tiešsaiste] – [atsauce 21.05.2008.]. Pieejams: http://www.w3.org/TR/xml/.
10. *OMG Unified Modeling Language (OMG UML), Infrastructure, V2.1.2* [tiešsaiste] – [atsauce 21.05.2008.]. Pieejams: http://www.omg.org/docs/formal/07-11-04.pdf.
11. *WAVE File Format* [tiešsaiste] – [atsauce 21.05.2008.]. Pieejams: http://www.borg.com/~jglatt/tech/wave.htm.
12. *WAVE PCM soundfile format* [tiešsaiste] – [atsauce 21.05.2008.]. Pieejams: http://ccrma.stanford.edu/courses/422/projects/WaveFormat/.

PIELIKUMI

1. pielikums

Kompaktdisks ar konkatenatīvas runas sintēzes sistēmas izmēģinājuma versiju

Bakalaura darbam pievienots kompaktdisks, kas satur „Konkatenatīvas latviešu valodas runas sintēzes sistēmas" izmēģinājuma versiju „*T2S v1*". Izmēģinājuma versijai pievienota saskarnes lietojumprogramma „*Universal Console Application Interface*", kas nodrošina testa versijas lietošanas iespējas *Microsoft Windows* operētājsistēmās. Izmēģinājuma versija satur pilnvērtīgu datu bāzi, kas ļauj sintezēt visus latviešu valodas vārdus.

Pielikumā pievienota arī „Konkatenatīvas latviešu valodas runas sintēzes sistēmas" izmēģinājuma versijas lietotāja pamācība. Tajā ir izskaidrots, kādas prasības nepieciešamas datoram, uz kura izmēģinājuma versiju domāts izmantot, kā arī ir izskaidrots, kā pareizi izmēģinājuma versija ir jālieto.

Bakalaura darbs „Konkatenatīvas latviešu valodas runas sintēzes sistēmas izveide” izstrādāts LU Fizikas un matemātikas fakultātē.

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

Autors: Mārcis Pinnis

Rekomendēju darbu aizstāvēšanai

Vadītājs: pētnieks *Mg. Sc. Comp.* Normunds Grūzītis

Recenzents: SIA „Tilde” valdes loceklis *Mg. Sc. Comp.* Andrejs Vasiļjevs

Darbs iesniegts Datorikas nodaļā 23.05.2008.

Metodiķe: Ārija Sproģe

Darbs aizstāvēts bakalaura gala pārbaudījuma komisijas sēdē

05.06.2008. prot. Nr. __, vērtējums _____

Komisijas sekretārs: lektors Uldis Straujums