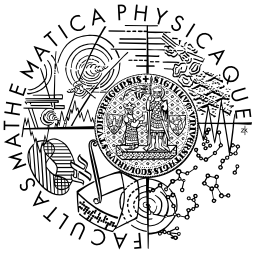


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Radek Strýček

DiaSynth - Syntezátor audia pomocí diagramů, přehrávač a analyzátor

Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: doc. RNDr. Jan Kofroň, Ph.D.
Studijní program: Informatika
Studijní obor: Obecná informatika

Praha 2021

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného titulu.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ....

Podpis autora

Tímto bych rád poděkoval vedoucímu bakalářské práce doc. RNDr. Janu Kofroňovi, Ph.D. za trpělivost, ochotu a pomoc při zpracování mé práce. Dále bych poděkoval rodině za dlouholetou podporu, bez které bych tuto práci nemohl dokončit.

Název práce: DiaSynth - Syntezátor audia pomocí diagramů, přehrávač a analyzátor

Autor: Radek Strýček

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: doc. RNDr. Jan Kofroň, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Cílem práce bylo vytvořit uživatelsky přívětivý program s grafickým uživatelským rozhraním pro práci se zvukem. Práce obsahuje analyzátor, audio přehrávač podobný programu Audacity a část umožňující syntézu zvuku technikou diagramů. Diagram se skládá z generátorů zvukových vln, operátorů a spojení mezi nimi. Výsledkem práce je funkční program napsaný v programovacím jazyce Java, který navíc uživateli umožňuje rozšířit jednotlivé částí pomocí zásuvných modulů (pluginů), což konkrétně znamená následující. Uživatel si může napsat vlastní algoritmy pro analyzátor, vlastní operace pro modifikaci zvukových stop načtených v prohlížeči vln a vlastní operátory a generátory pro syntezátor.

Klíčová slova: Zvuk Audio Syntéza zvuku Přehrávání a úprava zvuku Analýza zvuku

Title: DiaSynth - Diagram Audio Synthesizer and Analyser

Author: Radek Strýček

Department: Department of Distributed and Dependable Systems

Supervisor: doc. RNDr. Jan Kofroň, Ph.D., Department of Distributed and Dependable Systems

Abstract: The main goal of this bachelor's thesis was to create user friendly audio software with graphical user interface. The program contains analyser, audio player in the style of Audacity and sound synthesizer, which is using diagrams to generate audio. Diagram consists of wave generators, operators and connections between them. The result of this work is functional program written in Java programming language. On top of that each part of program can be extended by plugins. To be more specific, it means that there is possibility for user to write own audio analysis algorithms for analyser, own operations for modification of audio tracks loaded in audio player and in case of synthesizer, it means that the user can implement own operators and generators and use them in diagram for sound synthesis.

Keywords: Audio Sound Sound synthesis Audio analysis Digital signal processing

Obsah

1. Úvod

Zvuk a práce s ním je velice důležitá, neboť se po zraku jedná o nejdůležitější smysl, proto se zvuk, jeho chování, vlastnosti a vliv na člověka zkoumá od nepaměti. Pro naší práci je ale nejzásadnější digitální zvuk, který se začal intenzivně zkoumat relativně nedávno, což je samozřejmě dané tím, že do té doby se řešily věci okolo zvuku analogově. Největší výzkum digitální formy zvuku probíhal ve 20. století, kdy i dosáhl svého vrcholu. Výzkum se uskutečňoval jak na univerzitách, tak i v telefonních společnostech a to zejména Bell Labs, které se zajímaly o vše od komprese zvuku až po syntézu. Od té doby uběhlo mnoho let, během nichž došlo k rozšíření počítačů do domácností a obřímu růstu jejich výpočetního výkonu, což mělo za následek vznik spousty nových programů pro práci se zvukem jako je například tento.

Cílem této práce je navrhnout a implementovat snadno použitelnou desktopovou aplikaci pro práci se zvukem, konkrétně umožňující analýzu zvuku, prohlížení a modifikaci zvukových vln a jejich generování za pomoci diagramu složeného z operací a generátorů. Navíc je možné aplikaci rozšířit pomocí zásuvných modulů (pluginů).

Práce může sloužit jako alternativa k již existujícím programům a to jak amatérům, tak i pokročilejším. Může být použita na školách pro ukázku chování zvukových signálů a technik syntézy zvuku, jakou je například frekvenční modulace, a díky existenci pluginů může sloužit i k výuce algoritmů pro práci se zvukem. V neposlední řadě může být program využit pro tvorbu zvukových efektů, například do her.

1.1 Funkcionalita a cíle práce

Aplikace se bude skládat z následujících částí, které budou přístupné skrz záložky (taby) ve spuštěném okně.

1.1.1 Analyzátor

Analyzátor slouží k extrakci informací o zvukovém souboru. Implementovanou funkcionalitu, která zahrnuje algoritmy měřící počet úderů za minutu, tzv. Beat per minute (BPM) algoritmy a základní informace o audio formátu lze rozšiřovat pomocí zásuvných modulů (pluginů), ve kterých má uživatel možnost naprogramovat nové algoritmy pro analýzu zvukových dat.

1.1.2 Přehrávač

Přehrávač, který je inspirovaný programem Audacity, kromě přehrávání mixovaných zvukových stop umožňuje i rychlé prohlížení vlny s možností přiblížení a změny výšky zobrazované vlny. Přehrávač obsahuje referenční časové značky, k jejichž určení bylo nutné vymyslet algoritmus. Navíc přehrávač obsahuje měřič intenzity zvuku a umožňuje modifikaci vln a jejich částí, nejzajímavější implementovanou operací pro modifikaci je low-pass filtr, což je filtr, který propouští

frekvence pouze pod určitou hranicí. Nové operace nad zvukovými stopami lze přidat pomocí zásuvných modulů (pluginů).

1.1.3 Syntezátor

Syntezátor umožňuje provést syntézu zvuku pomocí diagramu složeného ze série generátorů a operátorů. Jedná se o zřejmě nejzajímavější část práce.

Tato část obsahuje grafické uživatelské rozhraní pro tvorbu diagramů, které zahrnuje zobrazení diagramu, možnost pohybu po diagramu a přibližování na části diagramu, navíc bylo nezbytné vymyslet algoritmus řešící zobrazení připojování panelů. Dále bylo nutné umožnit zapnutí, respektive vypnutí přehrávání, zobrazení a nahrávání generované vlny a ačkoliv lze přidat operátory a generátory pomocí pluginů, tak bylo samozřejmě nutné napsat alespoň základní operátory a generátory.

1.2 Pojmy

Pojmem *panel* označujeme v textu GUI komponentu jazyka Java. Prvek diagramu označujeme buď pojmem *výpočetní jednotka*, nebo *panel*.

1.3 Zkratky

Pokud budeme v práci uvádět jméno balíčku, vynecháme `str.rad`, neboť tento prefix mají všechny balíčky. Prefix existuje, aby nedocházelo ke kolizím s jmény balíčku knihoven třetích stran.

1.4 Krátký popis kapitol

Na začátku každé kapitoly najdeme shrnutí o jejím obsahu s odkazy na jednotlivé sekce.

Nejprve si v kapitole Úvod do problematiky digitálního zvuku zavedeme pojmy z oblasti zvuku a podíváme se na základy práce se zvukem. Poté v kapitole Analýza problémů rozebereme problémy, které jsme museli při psaní programu vyřešit. V kapitole Popis řešení se podíváme na objektový návrh a několik hlavních tříd a na řešení vybraných zajímavých problémů. Poté následuje kapitola Uživatelská dokumentace, ve které najdeme jak program používat, jak psát pluginy a kam je vkládat. Najdeme zde i jak program spustit a zkompilovat. V kapitole Související práce se podíváme na programy s podobnou funkcionalitou a v čem je Diasynth výjimečný. V závěru (6.3) zhodnotíme celkovou kvalitu programu a na konec se společně zamyslíme, co by bylo dobré změnit, či přidat do práce v budoucnu.

2. Úvod do problematiky digitálního zvuku

V této části si představíme základní pojmy z oblasti zpracování zvuku, které se hodí znát. Většinu z nich budeme v textu používat. Pokud není přímo u pojmu uvedeno jinak, je definice převzata z volně dostupných zdrojů na internetu.

2.1 Obecné pojmy

Frekvence [12] odpovídá počtu opakování periodického děje za jednotku času. V našem případě je dějem zvukový signál. Typickou jednotkou času je sekunda, pak frekvenci měříme v jednotce Hertz (Hz), tedy počet opakování periody za sekundu. Doba trvání *periody* odpovídá $\frac{1}{frekvence}$.

Amplituda [10] značí maximální možnou absolutní hodnotu vlny. Respektive pokud nemá vlna referenční (neutrální) bod v nule, převedeme jí tak, aby měla a na nové vlně zjistíme amplitudu pomocí absolutní hodnoty.

Fázi [4, str. 38] měříme ve stupních, resp. radiánech a zjednodušeně se dá říct, že udává posun vlny v čase vůči počátečnímu (referenčnímu) bodu vlny. Trvá 360° než dojde k návratu do původního bodu, tedy jinými slovy jedna perioda odpovídá 360°. Tato zjednodušená definice nám bude pro účely práce stačit.

Sample (vzorek) je hodnota vlny v určitém okamžiku, což odpovídá vychýlení od klidného stavu.

Bit depth (bitová hloubka) udává počet bitů použitých na reprezentaci jednoho samplu. V práci budeme mluvit o reprezentaci pomocí celých čísel (typ Integer). Obvykle používaná bitová hloubka činí 16 bitů, občas se používá i 24 bitů pro lepší kvalitu. Čím větší je bitová hloubka, tím více různých hodnot dokážeme reprezentovat, například pro bitovou hloubku o 16 bitech lze reprezentovat $2^{16} = 65536$ různých hodnot. Častěji se na úrovni programu používá termín *sample size*, který mluví o velikosti samplů v bytech.

Sampling rate, častěji *sample rate (vzorkovací frekvence)* udává, kolik samplů najdeme v určitém časovém úseku. Typicky je časovým úsekem jedna sekunda, pak mluvíme o počtu samplů za sekundu a hodnotu měříme v jednotce Hertz (Hz). Značíme f_s. Době mezi samply říkáme *sampling period* a odpovídá hodnotě $\frac{1}{f_s}$. Čím větší je vzorkovací frekvence, tím máme lepší informaci o tvaru vlny a dokážeme reprezentovat i signály o vyšších frekvencích, viz definice 1.

Samplování signálu (Sampling), česky *vzorkování*, je způsob převodu analogové vlny na digitální a to tak, že v pravidelných intervalech pozorujeme hodnoty analogové vlny. Digitální vlna nemusí odpovídat analogové, pokud je nejvyšší frekvence v analogovém signálu vyšší než Nyquistova frekvence (definice 1).

Pulzně kódová modulace (PCM) [13] je název techniky převodu analogového signálu na digitální. Odpovídá vzorkování o dané bitové hloubce. Samplované hodnoty vždy přiřadíme k nejbližší hodnotě z digitálního rozdělení. Počet kroků rozdělení odpovídá bitové hloubce a rychlost vzorkování je určena vzorkovací frekvencí. Pokud jsou kroky rozdělení lineárně rovnoměrné, mluvíme o lineární PCM (LPCM).

Zadefinujeme si pojem Nyquistova frekvence (více se lze dočíst v [4, str. 63]). S definicí přišel Harry Nyquist.

Definice 1. *(Nyquist frequency)*
Nyquistova frekvence odpovídá hodnotě $\frac{f_s}{2}$. Jedná se o teoretický limit na nejvyšší frekvenci reprezentovatelnou v digitálním systému, který pracuje se vzorkovací frekvencí f_s.

Při překročení Nyquistovi frekvence dochází k tzv. *aliasingu*, což je situace kdy máme vlnu s frekvencí nad Nyquistovou. V takovém případě nemáme dostatečnou informaci na reprezentaci vlny, což má za následek, že reprezentovaná vlna neodpovídá původní vlně. Odpovídá úplně jiné, která má frekvenci pod Nyquistovou. Z toho plyne, že vzorkovací frekvence musí být aspoň dvakrát větší než je nejvyšší frekvence obsažena v analogovém signálu, což je známé pod názvem *Nyquist theorem*. Aliasing je důsledek tzv. *undersampling*, což jednoduše znamená, že jsme zvolili příliš malou vzorkovací frekvenci a tedy nejvyšší frekvence v signálu je vyšší než Nyquistova frekvence (Definice 1). Na tento problém si musíme dávat pozor především při převodu na nižší vzorkovací frekvenci.

Definice 2. *(Decibel, [4, str. 63])*
Decibel (dB) je logaritmická bezrozměrná jednotka pro porovnání poměru intenzit 2 (akustických) signálů. Poměr intenzit:

$$R_I = 10 * log_{10}\frac{I_1}{I_2}[dB] \tag{2.1}$$

kde I_1 je intenzita prvního signálu a I_2 druhého. Logaritmus o základu 10 se používá z konvence.

Ovšem v počítačových systémech nemáme informace o fyzických intenzitách signálů, proto se porovnávají jejich amplitudy a tedy získáme tento poměr.

$$R_A = 20 * log_{10}\frac{A_1}{A_2}[dB] \tag{2.2}$$

kde A_1 a A_2 jsou amplitudy porovnávaných signálů. Platí, že zvýšení o 6dB odpovídá zdvojnásobení amplitudy.

Decibel je obzvlášť vhodný pro porovnání intenzit zvuků, neboť lidské vnímání intenzity zvuku je logaritmické.

Definice 3. *(Konvoluce [7, str. 404])*
Konvoluce vektorů $\mathbf{x}$ a $\mathbf{y}$ je $\mathbf{z} = \mathbf{x} * \mathbf{y}$ takový, že $z_j = \sum_k x_k y_{j-k}$, přičemž indexujeme modulo n.

Resp. v našem případě si lze představit digitální signály X a Y. X[i] značí i-tý sample signálu X. V takovém případě je konvoluce definovaná následovně.

$$Z[j] = \sum_k X[k] * Y[j-k] \tag{2.3}$$

Pokud se pohybujeme ve frekvenčním spektru, konvoluci spočteme jako $FZ[k] = FX[k] \cdot FY[k]$, kde FX je výsledek FFT aplikované na pole X.

2.1.1 RMS

Ačkoliv v textu žádná zmínka o hodnotě RMS není, je dobré jí popsat, neboť se jedná o jednu z hodnot, kterou můžeme získat z analyzátoru.

Definice 4. *Root mean square (RMS) odpovídá hodnotě* $\sqrt{\frac{\sum_{i=0}^{LEN} S[i]^2}{LEN}}$*, kde S je pole se samply a LEN je jeho délka. V češtině je pojem známý pod názvem efektivní hodnota.*

Definici RMS můžeme najít na volně dostupných zdrojích z internetu, například na stránce wikipedia [14].

Hodnota RMS se používá jako alternativa k průměru, neboť průměr nám v mnoha případech nedává žádnou informaci o zkoumaném signálu. Například průměr z funkce sinus je roven nule, zatímco RMS odpovídá hodnotě 0.707. Mluvíme o případu, kdy jsou samply typu double s hodnotami mezi -1 a 1. Důvodem je druhá mocnina ve výpočtu, díky které nepočítáme se zápornými hodnotami a tedy nedojde k vynulování. Z tohoto důvodu nám RMS poskytuje lepší informaci o skutečné hladině energie signálu.

2.1.2 Filtry

Ve zkratce řekneme, jak fungují filtry pro zpracování digitálních signálů. Funkcionalitou filtru je nějakým způsobem modifikovat frekvenční spektrum signálu. V práci používáme pouze filtr propuštějící frekvence pod určitou hranicí, tzv. low-pass filtr. Pojem cut-off frekvence označuje výše zmíněnou hranici. Je nutné podotknout, že filtry nežádané frekvence pouze utlumují, nikoliv odstraňují.

Rovnice pro digitální filtry jsou následující.

Rovnice pro výpočet nerekurzivního filtru má tuto podobu [4, str. 203]:

$$y(n) = a_0 \cdot x(n) + a_1 \cdot x(n-1) + ... + a_N \cdot x(n-N) \tag{2.4}$$

N udává historii, tj. počet samplů, které bereme v úvahu pro vypočtení hodnoty současného samplu. a_i jsou koeficienty, kterými násobíme samply ze vstupního signálu x.

Rekurzivní filtry navíc přidají do výpočtu dříve vypočítané hodnoty [4, str. 212]:

$$\begin{aligned} y(n) = a_0 \cdot x(n) + a_1 \cdot x(n-1) + ... + a_N \cdot x(n-N) \\ + b_1 \cdot y(n-1) + ... + b_M \cdot y(n-M) \end{aligned} \tag{2.5}$$

Rekurzivní filtry sice v práci nepoužíváme, ale v sekci 4.4.6 se o nich zmíníme.

2.2 FFT

Pokud se chceme dozvědět o algoritmu FFT něco více, především o matematických vlastnostech algoritmu a jak k němu vlastně přijdeme, lze tak učinit v knihách *Průvodce labyrintem algoritmů* [7, str. 391] a *Understanding Digital Signal Processing* [6, kapitola 3 - DFT a kapitola 4 - FFT]. V rámci textu se pouze stručně zaměříme na to, co výstup FFT pro zvukové signály udává, což nalezneme v druhé z výše zmíněných knih.

Definice 5. *(FFT) Rychlá fourierova transformace (Fast Fourier Transform) je časově efektivní algoritmus pro spočtení diskrétní fourierovy transformace (DFT).*

Algoritmus DFT umožňuje převést signál z časové domény do domény frekvenční, ze které lze vyčíst další vlastnosti zkoumaného zvuku. Navíc ve frekvenční doméně můžeme rychle vykonat konvoluci dvou signálů, viz poznámka pod definicí 3.

Definice 6. *(FFT okno) FFT okno udává počet komplexních čísel na vstupu algoritmu FFT. Základní algoritmus FFT pracuje pouze s okny velikosti* 2^n*. Budeme značit ws.*

Velikost okna nemusí vždy být 2^n, což potvrzuje i použitá knihovna, která podporuje libovolné velikosti FFT oken. Ovšem pro velikosti různé od 2^n se použije jiný, nejspíše pomalejší algoritmus.

Nechť je FX výsledek algoritmu FFT aplikovaného na pole komplexních čísel X, pak FX obsahuje komplexní čísla. Platí, že $FX[k]$ udává informace o frekvenci $k \cdot freqJump$ Hz, $freqJump = \frac{f_s}{ws}$, kde f_s odpovídá *vzorkovací frekvenci* a *ws* je velikost FFT okna [6, str. 52].

Místu uvnitř pole ve výsledku FFT budeme říkat přihrádka (anglicky bin). Například nultá přihrádka odpovídá frekvenci 0 Hz a obsahuje nulté komplexní číslo z výsledku FFT, tj. $FX[0]$. První přihrádka odpovídá frekvenci *freqJump* Hz a obsahuje první komplexní číslo z výsledku FFT, tj. $FX[1]$. $(\frac{ws}{2}+1)$-tá přihrádka odpovídá nyquistově frekvenci, respektive pro lichou velikost FFT okna odpovídá frekvenci $f_n - \frac{freqJump}{2}$ Hz, kde f_n je nyquistova frekvence.

Za zmínku stojí hodnota nulté přihrádky. Udává vychýlení vlny od nuly, mluvíme o tzv. biasu [6, str. 62].

$$\begin{aligned} FX[0].real &= \sum_{i=0}^{LEN-1} X[i] \\ FX[0].imag &= 0 \end{aligned} \tag{2.6}$$

Získaná komplexní čísla poskytují velice těžko uchopitelnou informaci o frekvenčním spektru signálu. Z toho důvodu je převedeme na reálná čísla, která udávají energie daných přihrádek, tzv. magnitudy. Další užitečné hodnoty, které lze získat z komplexního čísla v přihrádce jsou fáze a „power“ (magnituda na druhou) [6, str. 53]. V práci používáme pouze magnitudy.

Definice 7. *(Magnituda) Magnituda odpovídá vzdálenosti komplexního čísla od nuly. Magnitudu i-té přihrádky spočteme následovně*

$$\sqrt{FX[i].real^2 + FX[i].imag^2} \tag{2.7}$$

Použitá knihovna umožňuje provést reálnou FFT, která dostane na vstup pole se zvukovými samply a do stejně velkého pole vrátí výsledek FFT. Tedy na vstupu je n reálných čísel, respektive n komplexních čísel s imaginární částí rovnou nule, proto chybí. Na výstupu máme $\frac{ws}{2}+1$ komplexních čísel, kde poslední komplexní číslo odpovídá nyquistově frekvenci, respektive pro lichou velikost okna frekvenci o něco nižší, viz výše. Některá z výstupních komplexních čísel mají imaginární složku rovnou nule a díky tomu se výsledek vejde do pole délky n. Tuto verzi FFT používáme v práci nejčastěji.

Komplexní varianta FFT dostane na vstup n komplexních čísel a vrátí n komplexních čísel. Pokud tuto FFT zavoláme na komplexní čísla s nulovými imaginárními složkami, což je nejčastější případ, můžeme ve výsledku ignorovat všechna komplexní čísla za $(\frac{ws}{2} + 1)$-tým komplexním číslem. Dávají informace o frekvencích nad nyquistovou, a protože aliasing je symetrický podle nyquistovi frekvence, hodnoty v druhé polovině jsou stejné jako ty v první (tj. FX[k] = FX[n − k]), akorát imaginární složka je vynásobena −1. [6, str. 63].

Pokud máme v signálu frekvenci nespadající do některé z přihrádek, dochází k tzv. úniku („leak“) energie. Výsledkem je únik energie této frekvence do několika okolních přihrádek a tím pádem dojde ke zkreslení výsledného frekvenčního spektra. Pro minimalizaci úniku energie se používá technika „windowing“, což není nic jiného než vynásobení vstupních samplů hodnotami odpovídajícími funkci okna vybraného v rámci „windowing“. Příklady oken jsou rectangle, kdy vstupní samply neměníme, Hamming, Hann, Kaiser, kde poslední z nich je parametrizovaný. Zjednodušeně řečeno použití okna v rámci windowing sníží počet přihrádek, do kterých unikne energie [6, sekce 3.8 - DFT Leakage a sekce 3.9 - Windows, str. 71-88]. Informace o tomto tématu lze najít i v [4, Sekce 7.2 Short-term fourier transform, str. 244-251].

2.3 Práce s digitálním zvukem

Digitální zvuk je série časově od sebe stejně vzdálených samplů, jejichž vzdálenost odpovídá $\frac{1}{f_s}$, kde f_s je vzorkovací frekvence. V rámci práce podporujeme pouze načítání celočíselných samplů z rovnoměrného digitálního rozdělení. Jedná se totiž o nejčastější způsob ukládání zvukových dat. V jazyce Java tento způsob uložení najdeme pod názvem PCM_SIGNED pro znaménkové samply a PCM_UNSIGNED pro bezznaménkové, viz dokumentace jazyka Java [9]. Naprostá většina zvukových dat má znaménkové samply.

Uvnitř programu typicky pro jednoduchost převedeme vstupní vzorky v podobě série bytů na proměnné formátu double, či float, nebo občas i formátu typu int pro maximální rychlost. V případě typu double, respektive float pracujeme s hodnotami v rozmezí mezi -1 a 1, kde 1 je nejvyšší reprezentovatelná hodnota samplem a -1 nejmenší. V případě celých čísel mluvíme o intervalu $[0, 2^{bitDepth} - 1]$ pro bezznaménková čísla a $[-2^{bitDepth-1} + 1, 2^{bitDepth-1}]$ pro znaménková. Při převodu vstupních samplů, které jsou reprezentovány jako série bytů, na interní doubly je nutné mít na vědomí, že příchozí samply mohou být big i little endian a v znaménkovém i bezznaménkovém formátu a samozřejmě je nutné dát si pozor i na velikost samplu v bytech. Totéž platí i při převodu na výstupní formát, do kterého převádíme při zápisu do souboru, či při přehrávání.

3. Analýza problémů

V této kapitole budeme diskutovat volby, které jsme při návrhu a vývoji aplikace učinili. Rozebereme zde mimo jiné podmnožinu požadované funkcionality programu zmíněné v sekci Funkcionalita a cíle práce.

Nejprve se podíváme na volbu programovacího jazyka (3.1), poté zanalyzujeme vlastnosti jazyka Java (3.2). Následně rozebereme, jaké knihovny potřebujeme a z jakých důvodu (3.3). V sekci 3.4 vyřešíme aspekt uživatelského rozhraní, reprezentaci zvukových dat uvnitř programu projdeme v sekci 3.5 a pluginy v sekci 3.6. Dalším aspektem byl formát ukládání projektů aplikace, přesněji se jedná o problémy uložení analyzovaných informací získaných z audio stop 3.7.1 a serializaci diagramů 3.9.1. Dále budeme mluvit o vizualizaci zvukových dat, což zahrnuje volbu algoritmů a jejich implementaci (3.8.1 a 3.8.2). V poslední sekci kapitoly nás čekají následující podsekce: zobrazení diagramu (3.9.2), zobrazení menu pro výběr prvků do diagramu (3.9.3) a způsob jakým budeme připojovat a pohybovat panely (3.9.4).

3.1 Volba jazyka

Na začátku práce bylo nutné vybrat jazyk, ve kterém práci naprogramujeme. Výběr zúžíme na čtyři jazyky, o kterých jsem měl alespoň nějaké základní znalosti a navíc jsem věděl, že jsou k vyřešení problému vhodné: C++, Python, C#, a Java.

C++ se nabízí jako první možnost, neboť se jedná o nejčastěji používaný jazyk pro práci se zvukem a to hlavně díky tomu, že je ze všech alternativ nejrychlejší. Jenže má i několik nevýhod. Ve srovnání s Javou je v určitých věcech nízkoúrovňový a na naprogramování jistých funkcí, například pluginování, by bylo potřeba více času. Navíc na začátku psaní práce jsem neměl dostatečné znalosti o jazyce a nepřišlo mi úplně rozumné vrhnout se do tak velkého projektu s jazykem, o kterém jsem věděl velice málo. Obzvláště pokud se jedná o jazyk C++, jehož úplné pochopení vyžaduje netriviální množství času.

Jazyku Python jsem vyvaroval ze stejného důvodu jako C++, také jsem ho v době začátku psaní práce dostatečně neznal. Navíc Python je pomalejší než Java, ale za to v něm trvá vývoj programů méně času díky velkému množství knihoven a díky tomu, že se jedná oproti ostatním alternativám o skutečně vysokoúrovňový jazyk.

S jazykem C# jsem už měl v době psaní dobré zkušenosti. V mnoha věcech je srovnatelný s jazykem Java a v některých dokonce i lepší. Tedy ve finále byla volba zúžena na jazyky C# a Java. Nakonec jsem se rozhodl pro Javu, protože jsem s ní měl více zkušeností.

3.2 Java

3.2.1 Výhody

Díky obsáhlé standardní knihovně není nutné stahovat mnoho knihoven třetích stran. Standardní knihovna obsahuje metody a třídy jak pro práci se zvukem tak

tvorbu grafického uživatelského rozhraní.

Java je platformně nezávislá, takže při programování není nutné řešit detaily závislé na platformě na rozdíl od C++. Navíc pro distribuci stačí jen jeden .jar soubor místo .exe souboru pro každou platformu.

Java stále patří mezi nejrozšířenější a nejpoužívanější jazyky.

Poslední výhodou je garbage collection, která oprostí programátora od nutnosti starat se o paměť.

3.3 Knihovny

Práce vyžaduje implementaci algoritmu FFT (5), neboť se využívá ve velké části algoritmů pro analýzu. V našem případě je využit v rámci algoritmu pro detekci počtu úderu za minutu. Algoritmus FFT už je několikrát efektivně naprogramovaný, takže nemá smysl ho implementovat znovu a proto jsme využili knihovny JTransforms (`https://github.com/wendykierp/JTransforms`). Hlavními výhodami jsou srozumitelná dokumentace, rychlost, popularita, a především skutečnost, že knihovna je stejně jako zbytek práce psaná v čisté Javě, tím pádem nebude porušena platformní nezávislost aplikace a dojde k zachování důležité vlastnosti programu a to přenositelnosti.

Alternativní volbou byla knihovna The Fastest Fourier Transform in the West (FFTW - `http://www.fftw.org/download.html`), která je sice rychlejší, ale jedná se o C/C++ knihovnu, tedy by bylo potřeba použít nějaký z naprogramovaných wrapperů požívající JNI (Java Native Interface). To by přineslo několik problémů. Za prvé by byla distribuce programu Diasynth o něco složitější a za druhé použití JNI může mít za následek ztrátu platformní nezávislosti a tedy i ztrátu přenositelnosti.

Dále bylo nutné umožnit načíst zvukovou vlnu formátu MP3 ze souboru do programu. Vzhledem k rozšířenosti formátu MP3 je tato funkcionalita nutností. Tento problém jsme vyřešili knihovnou MP3SPI (`http://www.javazoom.net/mp3spi/mp3spi.html`). Načítání ostatních základních formátů zvukových souborů už je přímo ve standardní knihovně. Hlavní výhodou této knihovny je, že využívá tzv. services, česky služeb (`https://docs.oracle.com/javase/tutorial/sound/SPI-providing-sampled.html`), což pro programátora znamená, že pro načtení formátu nepodporovaného standardní knihovnou jazyka Java nemusí nijak upravovat kód. Zjednodušeně řečeno se formát audio souboru, jehož načítání je implementované dodaným poskytovatelem služeb (service provider), přidá k těm ze standardní knihovny. V našem případě je formátem MP3 a poskytovatelem jsou třídy z knihovny.

Naprogramovat si vlastní načítání MP3 souborů nepřipadá k úvahu, neboť standard není volně přístupný a implementace rozhodně není triviální, například kvůli tomu, že součástí MP3 formátu je komprese samplů pomocí Huffmanova kódování. Hledat alternativní knihovnu pro načítání MP3 souborů nemá moc smysl, neboť vybraná knihovna je nejznámější a má výše popsanou příjemnou vlastnost.

3.4 Grafické uživatelské rozhraní (GUI)

V zadání práce bylo řečeno, že program bude obsahovat grafické uživatelské rozhraní. Standardní knihovna obsahuje metody a třídy pro tvorbu GUI, jak již bylo zmíněno u výhod jazyka Java. Ovšem v rámci standardní knihovny existuje více knihoven pro tvorbu GUI. Pro Diasynth jsme vybrali knihovnu Swing, neboť jsem s ní už měl alespoň základní zkušenosti. Jedná se o časem ověřenou knihovnu a navíc je knihovna Swing psaná v čisté Javě, takže je portabilní mezi platformami.

Alternativ existovalo hned několik, první je knihovna AWT, na jejímž základě je knihovna Swing vybudována. AWT pracuje nativně, tedy vše v rámci okna se řeší operačním systémem. Z toho důvodu některé funkce nemusí být přenositelné mezi platformami, což je velká nevýhoda, kvůli které nebyla knihovna AWT zvolena. Kdežto Swing použije AWT pouze k vytvoření okna a věci v rámci okna se řeší kódem standardní knihovny.

Další alternativou je JavaFX, která sice není součástí standardní knihovny, ale je přímo vyvíjena firmou Oracle, respektive dnes je již open-source. JavaFX je novější než Swing a možná bude v některých ohledech i lepší, ale knihovna Swing je dostatečná na to, čeho chceme v grafickém uživatelském rozhraní docílit a navíc s ní na rozdíl od knihovny Swing nemám zkušenosti.

3.5 Zvuk

Samozřejmě bylo nutné určit, jakým způsobem vyřešíme práci se zvukem. Práce se zvukem je stejně jako v případě GUI součástí standardní knihovny jazyka Java. Na rozdíl od situace s GUI je ve standardní knihovně pouze jediný možný způsob práce se zvukem.

Knihovna umožňuje jen úplný základ. V podstatě pouze načítaní, ukládání a přehrávání zvukových dat. Knihovna pracuje pouze se samply ve formátu série bytů, proto musíme v programu vyřešit převod bytů na jednotlivé samply daného zvukového formátu, se kterými už můžeme pracovat. Samply reprezentujeme buď typem int, nebo typem double, kdy samply navíc znormalizujeme a převedeme na hodnoty mezi -1 a 1. V rámci analyzátoru pracujeme s bytovými poli a samply ve formě intů. V syntezátoru a přehrávači pracujeme se samply typu double a převádíme je do typu int a následně do série bytů až při volání metody standardní knihovny pro přehrávání. Problém převodu vyřešíme svépomocí, neboť implementace řešení není až tak složitá. Ačkoliv knihovna poskytuje úplný základ, obsahuje vše, co jsme v rámci práce potřebovali, proto nebylo nutné hledat jinou knihovnu pro práci se zvukem (s výjimkou knihovny pro načítání formátu MP3). Standardní knihovna navíc poskytuje informace o audio formátu, které můžeme následně zobrazit v analyzátoru.

3.6 Pluginy

Pro zvýšení užitečnosti práce byla přidána rozšiřitelnost jednotlivých částí pomocí pluginů. Načítání pluginů zase řešíme pouze třídami a metodami ze standardní knihovny a to především pomocí reflexe a třídy ClassLoader. Na načítání

pluginů existují knihovny, například OSGi (`https://www.osgi.org/`), či Java Plug-in Framework (JPF), ale většina takových knihoven je až moc komplikovaných a jejich funkcionalita daleko převyšuje to, čehož chceme dosáhnout. Pouze chceme při spuštění programu načíst class soubory v adresářích do programu.

Jaký rozhraní musí pluginy implementovat, a kam musíme pluginy vložit, zjistíme v sekci 5.8.

3.7 Analyzátor

3.7.1 Ukládání analyzovaných audio stop

V zadání práce byla zmíněna možnost načíst dříve analyzované soubory, proto musíme vyřešit, v jaké formě je uložíme na perzistentní úložiště.

Data získaná z analýzy audio souborů ukládáme do souboru formátu XML. Vzhledem k tomu, že analyzovaná data jsou dvojice: označení algoritmu použitého pro analýzu a hodnota, tak se pro jejich uložení formát XML velice dobře hodí. Navíc pro práci s XML daty existuje ve standardní knihovně jazyka Java package `org.w3c.dom`, takže v programu musíme řešit pouze průchod XML stromem. Parsování a vytváření XML stromu řeší knihovna za nás. To ovšem není jediný způsob, jakým se dal problém vyřešit. Alternativami byly formáty JSON, či INI, nebo dokonce i vlastní formát, ale to už je zbytečné, neboť výše zmíněné formáty dokonale splňují požadavky na funkcionalitu. Pro formát XML jsem se nakonec rozhodl, protože už jsem s ním v jazyce Java pracoval dříve.

Detailněji rozebereme formát analyzovaných dat do souboru s koncovkou .xml v sekci 4.3.1.

3.8 Přehrávač

3.8.1 Zobrazení vlny

Jednou ze základních částí přehrávače je možnost zobrazit vlnu. V této části postupně dospějeme ke správnému řešení problému vykreslování vlny.

Než se podíváme na algoritmy vykreslování, je dobré si uvědomit dvě věci. Za prvé vlna má typicky mnohem více samplů než kolik je počet dostupných pixelů na šířku.

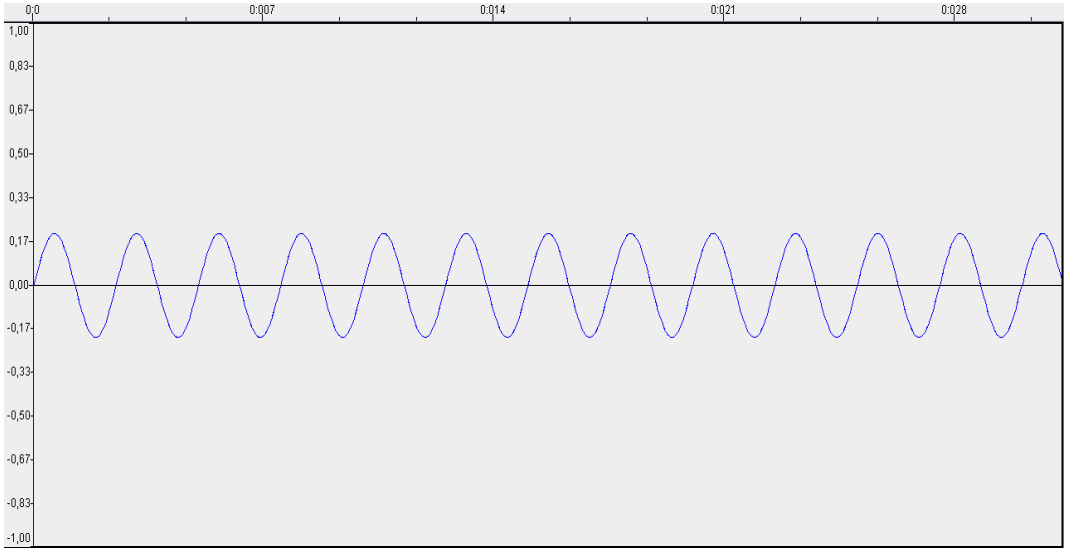
Když je samplů méně než pixelů na šířku, můžeme si dovolit vykreslit jednotlivé samply. Jednotlivé samply zobrazujeme jako úsečku vycházející ze samplu s hodnotou rovnou nule až do hodnoty vykreslovaného samplu. Pozici samplu vyznačíme kruhem.

Pokud máme více samplů než pixelů, je nutné samply agregovat. Chceme získat tolik hodnot, kolik je pixelů. Tyto hodnoty následně vykreslíme. Jako první možnost agregace se nabízí vzít průměr samplů. Tato metoda funguje poměrně dobře v případě, kdy máme jen několik málo samplů na jeden pixel. Typicky jsou hodnoty vedlejších samplů blízko u sebe, takže výsledný průměr je reprezentuje poměrně dobře. Ovšem pro náhodný generátor už nemusí platit, že hodnoty blízkých samplů jsou podobné. Můžeme mít dva samply vedle sebe s generovanými hodnotami x a $-x$, tedy máme průměr rovný nule. Tato hodnota už samply moc

dobře nereprezentuje. Podobný problém nastane, když bereme hodně samplů na pixel. Například pokud pixel odpovídá jedné periodě sinové vlny, je výsledná vykreslená hodnota rovna nule.

Předchozí řešení už nás může navést ke správnému východisku, kterým je vzít minimum a maximum ze samplů. Tyto hodnoty spojíme čárou, respektive kvůli čtvercové vlně a dalším vlnám je občas nutné navíc spojit předchozí maximum se současný minimem, nebo předchozí minimum se současným maximem. Problém spočívá v tom, že u čtvercové vlny nemusí být vykreslena vertikální úsečka mezi 1 a -1. Tato situace nastává, když agregujeme ze stejných hodnot, tj. jedna agregace proběhne na samplech rovných hodnotě 1 a druhá -1. Čtvercová vlna je extrémní případ tohoto jevu. V obecném případě jev nastane, kdykoliv předchozí a současná úsečka nemají společný bod na y-ové souřadnici. V takovém případě musíme úsečky spojit.

Jednotlivá vykreslení můžeme vidět níže na obrazcích 3.1 a 3.2.



Obrázek 3.1: Ukázka vykreslení vlny funkce sinus - agregované hodnoty

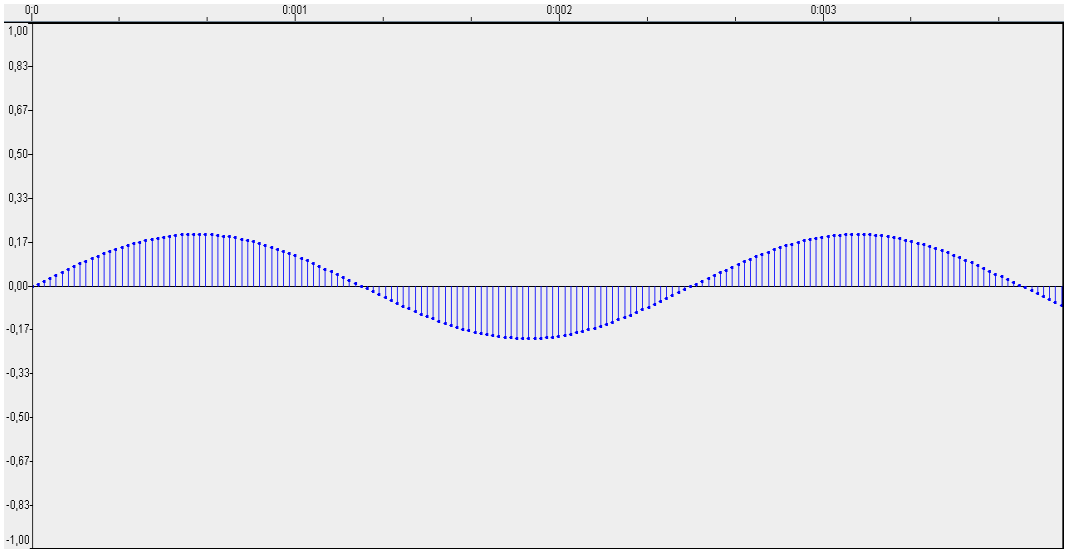
V sekci 4.4.1 se podíváme na datovou strukturu, kterou je nutné implementovat, pokud chceme vykreslování vlny provádět efektivně.

3.8.2 Měření intenzity zvuku

Většina programů umožňující přehrávání obsahuje měřič hlasitosti, proto bylo vhodné přidat ho i do programu Diasynth.

Měřič intenzity zvuku pracuje v decibelech, viz 2.2, tedy potřebujeme referenční hodnotu, vůči které budeme brát poměr hodnot samplů přehrávaného signálu. Když se podíváme do programu Audacity, vidíme, že maximální hodnota v decibelech je 0dB a minimální obvykle -60dB. Z toho můžeme vyvodit, že referenční hodnotou bude maximální možná hodnota samplu. Pokud se pohybujeme v typu double, jedná se o hodnotu jedna.

Nyní nám stačí vložit správné hodnoty do rovnice z definice decibelu. Do



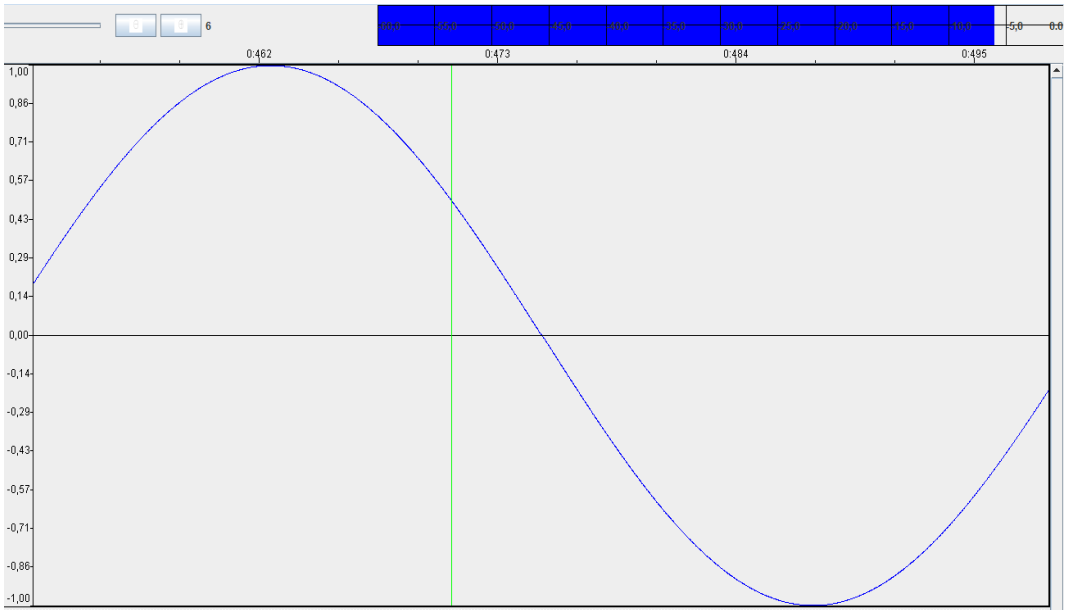
Obrázek 3.2: Ukázka vykreslení vlny funkce sinus - individuální hodnoty

čitatele dáme současný sample a do jmenovatele hodnotu jedna.

$$R_A = 20 * log_{10}\frac{A_1}{A_2}[dB] \tag{3.1}$$

Výše zmíněný výpočet provedeme pro každý kanál. Abychom zbytečně nezatěžovali procesor, počítáme a zobrazujeme intenzity v pravidelných intervalech, čímž i částečně vyřešíme problém příliš rychlého blikání hodnot.

Jak vypadá měřič intenzity zvuku můžeme vidět v obrázku 3.3.



Obrázek 3.3: Měřič intenzity zvuku je v pravém horním rohu. Intenzita odpovídá hodnotě -6dB a hodnota samplu, ze kterého intenzitu počítáme je 0.5. Jedná se o sample, který leží na zelené úsečce.

3.9 Syntezátor

3.9.1 Ukládání diagramů

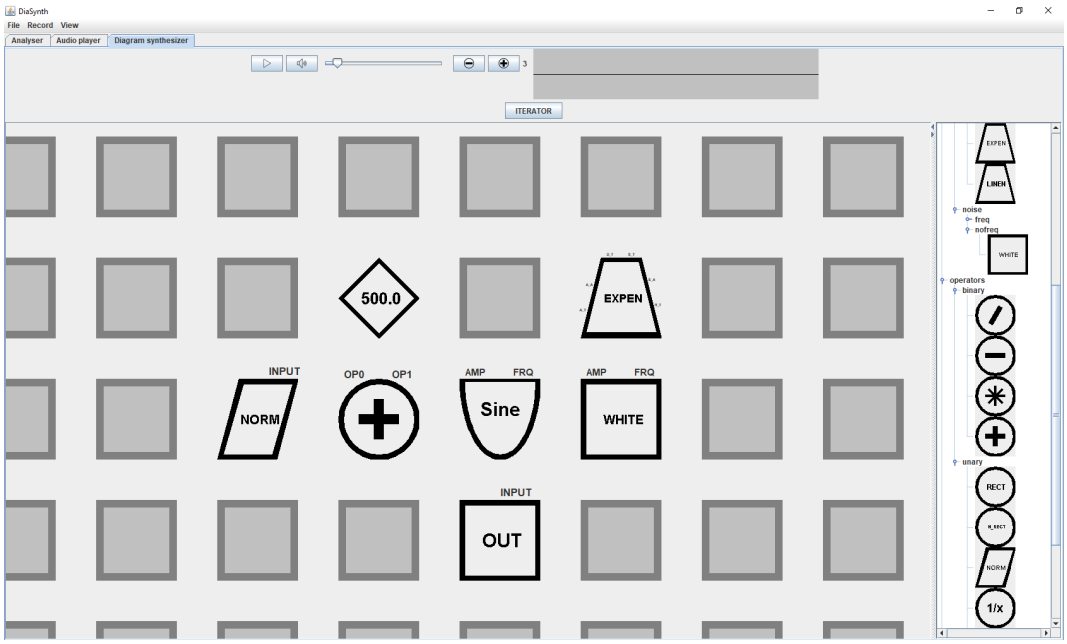
V zadání jsme zmínili, že diagramy bude možné ukládat a načítat.

Problém vyřešíme svépomocí, neboť je velice specifický. Uložený diagram budeme reprezentovat textovým souborem s nově vymyšleným formátem. Mohli jsme použít nějaký z již existujících formátů, například XML, ale v tomto případě je nejlepším řešením vymyslet formát nový, který na problém přesně pasuje. Na detaily se podíváme v sekcích 4.3.3 a 4.3.4.

3.9.2 Zobrazení diagramu a jeho prvků

Vzhledem k tomu, že diagram je grafický, musíme vyřešit jeho podobu.

V programu ukládáme jednotlivé prvky diagramu do předem připravených pozic v mřížce. To má oproti volnému umístění panelů v diagramu výhodu, v tom že výsledek je přehlednější a předvídatelnější, ale zaplatíme za to nevyužitou plochou. Vzhled okna se syntézou můžeme vidět v příloze 3.4.



Obrázek 3.4: Okno se syntezátorem

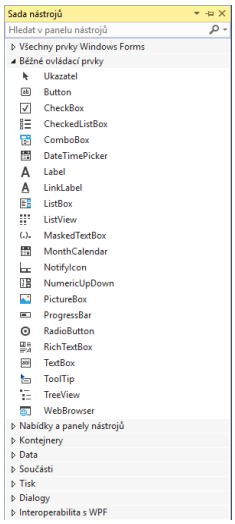
Dále bylo nutné nějakým způsobem odlišit prvky diagramu. Odlišení provádíme nejen tvarem panelů, ale i jejich obsahem.

3.9.3 Menu pro výběr prvků do diagramu

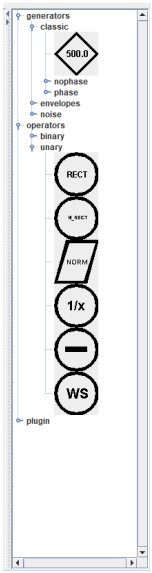
Při návrhu bylo nutné vymyslet, jakým způsobem uživateli zobrazíme přístupné výpočetní jednotky. Rozhodli jsme se implementovat nabídku výpočetních jednotek pomocí hierarchického rozbalovacího menu. Inspirace pro toto řešení pochází z programů pro tvorbu grafických uživatelských rozhraní, například z toho ve vývojovém prostředí Visual Studia, viz příloha 3.5.

Jak vypadá menu v programu Diasynth lze vidět v příloze 3.6.

Konkurenční program Pure Data řeší tento problém pomocí textových záložek ve vrchní části okna, což v kombinaci s ne úplně jasnými jmény výpočetních jednotek a absencí tooltipů může nového uživatele odradit.



Obrázek 3.5: Menu pro výběr prvků v Visual studiu při tvorbě win forms.



Obrázek 3.6: Menu pro výběr komponent do diagramu.

3.9.4 Řešení připojování a pohybu panelů

Při návrhu ovládání diagramu bylo nutné rozhodnout, jakým způsobem necháme uživatele pracovat s panely. První variantou, která se nabízí, je provádět posouvání panelů táhnutím a připojování pomocí kliknutí. Druhou variantou, která byla nakonec použita, je pohyb panelů přes kliknutí a připojování přes dvojklik. Druhé řešení má výhodu především pro uživatele notebooků a proto bylo zvoleno.

Navíc je nutné umožnit uživateli pohyb po diagramu v případě, kdy není možný posun táhnutím myši. To nastává při pohybu panelem a při připojování panelu na vstup jiného panelu. Problém řešíme přidáním lišt na kraje okna obsahující diagram. Lišty slouží k posunu daným směrem. Ještě musíme uživateli dát jasnou vizuální nápovědu, že lišty slouží k pohybu. To zařídíme zvýrazněním lišt modrou barvou a nakreslením šipek.

4. Popis řešení

Nejdříve si v sekci 4.1 připomeneme funkcionalitu jednotlivých částí. Poté se podíváme na objektový návrh 4.2, kde nejdříve zjistíme, jakým způsobem jsou spojeny jednotlivé části dohromady a pak jak jsou navrženy části samotné (Analyzátor, Přehrávač, Syntezátor). Následně blíže prozkoumáme formáty souborů vytvářených v rámci programu 4.3. Kapitolu zakončíme sekcí 4.4, ve které rozebereme řešení vybraných problémů.

Než se přesuneme dále tak, aby nedošlo k nedorozumění, zmíníme metody `paintComponent` a `paint`. Ty v jazyce Java slouží k vykreslování GUI komponent. Ještě si zadefinujeme pojmy operátor a generátor, které rozebereme později v sekci o vybraných problémech. Generátor je výpočetní jednotka, která je buď závislá na čase, nebo nemá vstupní parametry. Operátor je výpočetní jednotka, která na základě svých vstupů vyprodukuje výstup. Výpočetní jednotka je prvek diagramu, někdy jí také nazýváme panel.

4.1 Struktura programu

Program se skládá ze tří hlavních částí: Syntezátor, analyzátor a přehrávač. Níže rozebereme strukturu a funkci jednotlivých částí.

Analyzátor má dvě části. První umožňuje provádět analýzu audio souborů, výsledky analýz se ukládají na perzistentní úložiště. V druhé části si můžeme výsledky analýz zobrazit, případně i vymazat z perzistentního úložiště.

V přehrávači můžeme kromě přehrávání a mixování prohlížet načtené zvukové vlny, změnit výstupní audio formát, uložit výsledek mixování na perzistentní úložiště a v neposlední řadě provést operace na vlny. V přehrávači si můžeme všimnout záložky „`Experimental`“, jak název napovídá, tak funkcionalita v této záložce není stoprocentně odladěná a je v práci tak trochu navíc, proto o ní už nadále nenajdeme v textu zmínku. Na druhou stranu je dostatečně zajímavá na to, aby v programu zůstala.

Poslední částí programu je syntezátor, který slouží pro syntézu zvuku pomocí diagramů. Diagramy se skládají z generátorů a operátorů, které posouváme a připojujeme v rámci okna reprezentujícího syntezátor. V oknu představujícím syntezátor generovanou vlnu nejen přehráváme, ale i zobrazujeme. Generovaný zvuk lze uložit nejen na perzistentní úložiště, ale i přímo do přehrávače. Stejně jako u přehrávače můžeme kontrolovat výstupní formát generovaného zvuku. V neposlední řadě syntezátor umožňuje ukládání a načítání diagramů.

4.2 Objektový návrh

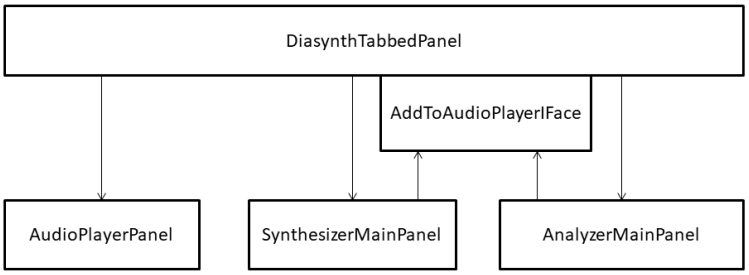
V této kapitole popíšeme objektový návrh programu Diasynth.

Ve všech diagramech udávají šipky směr toku informací, například A $\rightarrow$ B znamená, že informace směřují z A do B. Pokud nedochází k žádné častější výměně informací ani jedním směrem, nebo výměna není dostatečně dobře popsatelná, značíme spojení pouze úsečkou. Slovíčkem extends značíme dědičnost. Vždy je potomkem třída, která je v diagramu níže. Na hodně místech máme obousměr-

nou komunikaci, čemuž se nešlo ve většině případů vyhnout. Pro příklad komunikace mezi přehrávačem (`AudioPlayerPanel`) a panelem s vlnou (`WaveMainPanel`) vypadá následovně. Směrem k přehrávači míří události spojené s kliknutím na nějakou část panelu s vlnou. Směrem k vlně míří informace o pohybu posuvníku prohlížeče vln, přiblížení a žádosti o provedení změny na samplech dané vlny.

Program běží v jednom oknu `DiasynthTabbedPanel`, což je potomek třídy `JTabbedPane`, a na základě zvolené záložky se okno vyplní. Vyplnění zajišťují třídy `AnalyzerMainPanel`, `AudioPlayerPanel` a `SynthesizerMainPanel`, viz diagram 4.1. Tyto třídy musí implementovat rozhraní `TabChangeIFace` s metodou `changedTabAction(boolean hasFocus)`. Při zavolání s hodnotou „`false`“ má třída za úkol po sobě uklidit, což může například znamenat uspání audio vlákna. Při zavolání s hodnotou „`true`“ by se třída měla připravit na zobrazení, což minimálně zahrnuje nastavení vrchních záložek. Nyní popíšeme ve zkratce návrh v rámci těchto tříd. V sekcích 4.2.1, 4.2.2, 4.2.3 půjdeme více do hloubky a popíšeme důležité metody a některé pomocné třídy.

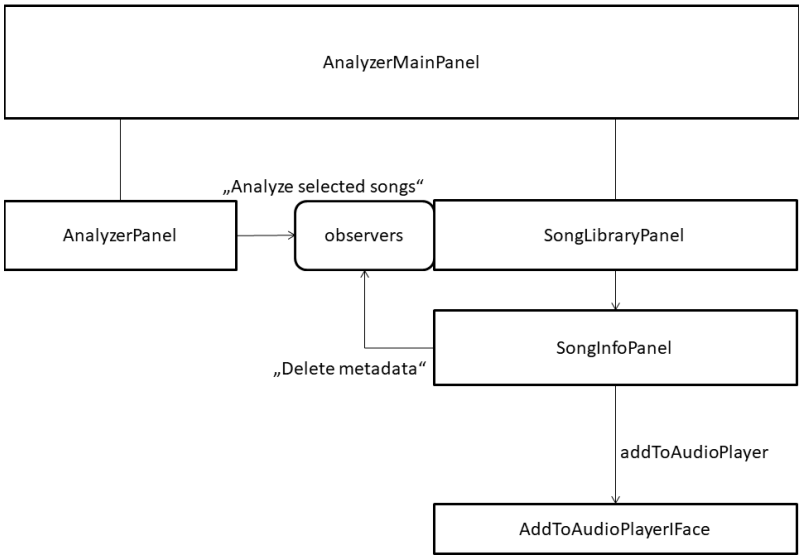
Ještě než se přesuneme na samotný objektový návrh, ujasníme si pár pojmů. Výraz posuvník je český výraz pro „scrollbar“. `JScrollPane` je třída ze standardní knihovny jazyka Java a odpovídá panelu s horizontálním a vertikálním posuvníkem. `JSplitPane` je také třída ze standardní knihovny. Jedná se o panel, který má vrchní a spodní, resp. levou a pravou část. Do částí můžeme vložit libovolné různé panely. Účelem `JSplitPane` je změna výšky, resp. šířky panelu v jedné části na úkor výšky, resp. šířky panelu v části druhé. Změna se provádí tahem myši. Pohybujeme oddělovačem částí.



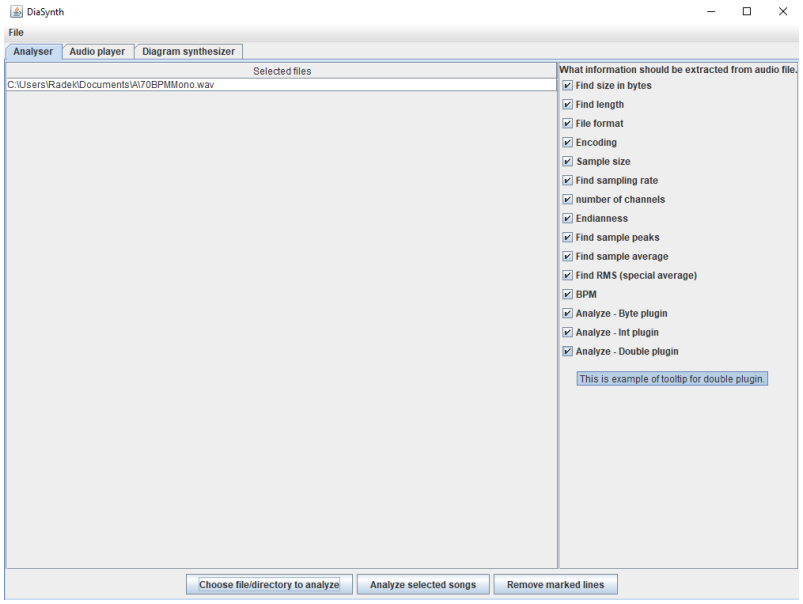
Obrázek 4.1: Diagram popisující hlavní části programu Diasynth

Panel `AnalyzerMainPanel` reprezentuje část s analyzátorem a je vyplněn buď třídou `AnalyzerPanel`, ve které analyzujeme nové písně, viz příloha 4.3, nebo třídou `SongLibraryPanel`, ve které vidíme výsledky analýz, viz příloha 4.4. Pokud se chceme podívat na informace o specifické písničce, vytvoří se instance třídy `SongInfoPanel`. Popis odpovídá diagramu 4.2.

Přehrávač je reprezentovaný třídou `AudioPlayerPanel`. Jak okno s přehráva-



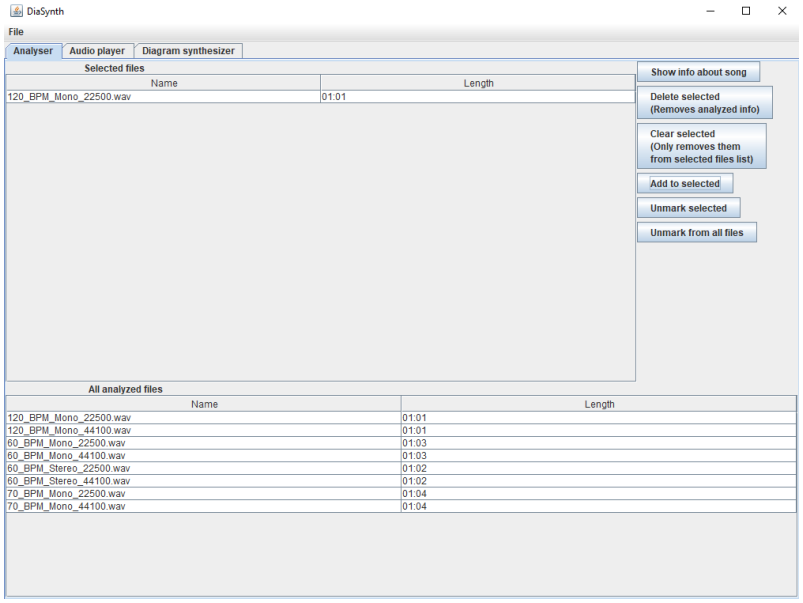
Obrázek 4.2: Diagram popisující analyzátor



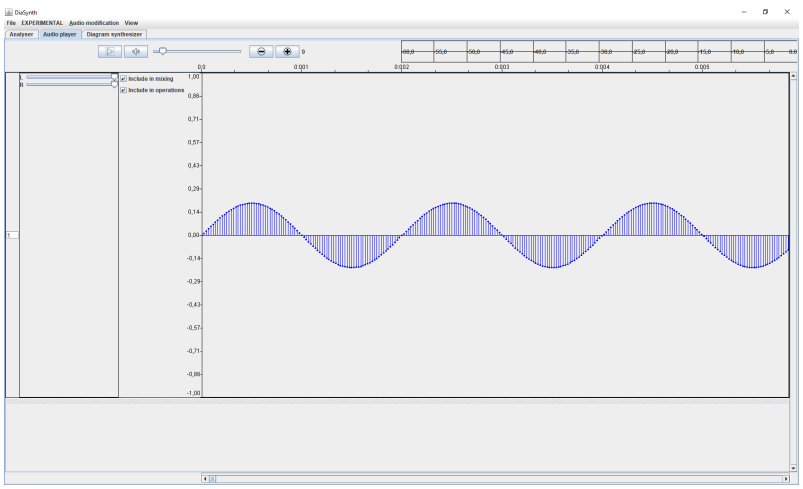
Obrázek 4.3: Okno pro výběr souborů k analýze

čem vypadá, můžeme vidět v obrázku 4.5. Rozmístění komponent v rámci panelu je řešeno pomocí `GridBagLayout`, což je layout manager. Ve vrchní části přehrávače nalezneme panel s tlačítky reprezentovaný třídou `AudioControlPanelWithZoomAndDecibel`. Obsahuje tlačítka pro kontrolu přehrávání, přiblížení a část pro měření decibelů. Pod tlačítky najdeme časové značky reprezentované třídou `TimestampsPanel`. Pod časovými značkami nalezneme JScrollPane, panel s posuvníkem, do kterého vkládáme jednotlivé stopy. Instance třídy JScrollPane je uložená v proměnné „*panelWithWaves*“. Pro lepší představu se můžeme podívat do diagramu 4.6.

Můžeme si všimnout, že panel s vlnami obsahuje pouze vertikální posuvník. Horizontální pohyb totiž musíme řešit zvláštním posuvníkem. K tomu nám po-



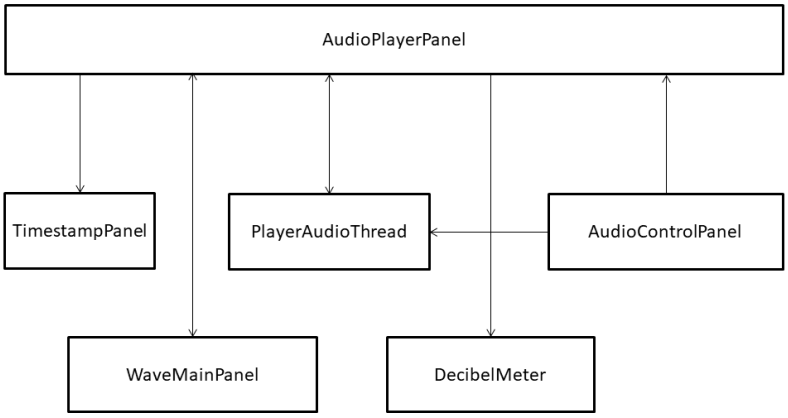
Obrázek 4.4: Okno s knihovnou dříve analyzovaných souborů



Obrázek 4.5: Okno s přehrávačem

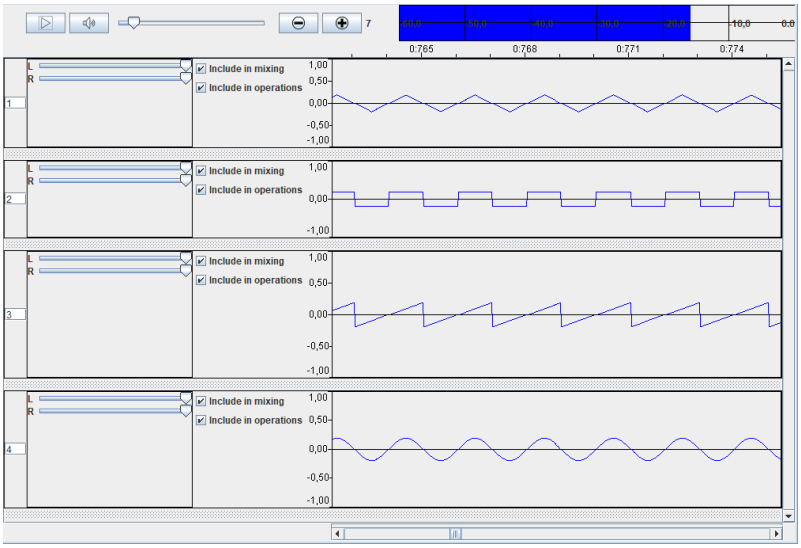
slouží třída `WaveScrollerWrapperPanel`, která se skládá z několika panelů. Jedním z nich je `JScrollPane` reprezentující horizontální posuvník. Na tuto třídu se důkladněji podíváme v rámci popisu přehrávače v sekci 4.2.2. Důvody pro oddělení horizontálního posuvníku jsou dva. Za prvé chceme mít posuvník jen pro pohyb po vlně, posuvník přímo na `JScrollPane` s vlnami by pohyboval i věcmi okolo vlny, tj. komponentami sloužící pro mixování. Druhým a hlavním důvodem je, že z implementačních důvodů nechceme, aby měl panel s vlnou velikost rovnou délce vlny v pixelech. Je lepší, když bude mít velikost odpovídající viditelné části vlny a změnu obsahu provádět na základě absolutní pozice ve vlně, která se mění v závislosti na událostech horizontálního posuvníku.

Vlny vkládáme do `JSplitPane`, aby je bylo možné vertikálně roztahovat. Jazyk Java nepodporuje multi `JSplitPane`, proto skládáme `JSplitPane` do sebe. To konkrétně znamená, že vrchní komponentou `JSplitPane` je `JSplitPane` a spodní komponentou jedna vlna. Platí rekurzivně. Výjimku tvoří první `JSplitPane`, to



Obrázek 4.6: Diagram popisující přehrávač

totiž obsahuje vlnu i ve vrchní části. Poslední `JSplitPane` má jako vrchní komponentu `JSplitPane` a jako spodní komponentu panel nulové velikosti reprezentovaný třídou `EmptyPanelWithoutSetMethod`. Pro lepší pochopení se můžeme podívat na obrázek 4.7. První `JSplitPane` obsahuje ve vrchní části trojúhelníkovou vlnou ve spodní čtvercovou. Další `JSplitPane` obsahuje ve vrchní části první `JSplitPane` a ve spodní pilovitou vlnu. Dále pokračujeme, jak jsme popsali výše.

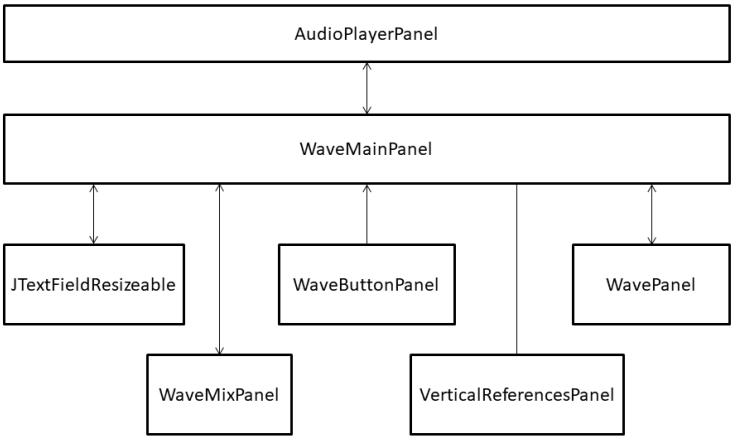


Obrázek 4.7: Přehrávač obsahující několik načtených vln

Skládání má pozitivní vedlejší efekt a to ten, že vlnu můžeme zvětšovat směrem nahoru na úkor vln nad ní.

Samotná vlna se skládá z pěti částí, které zastřešuje třída `WaveMainPanel`, viz diagram 4.8. Části jsou: pořadové číslo vlny v panelu s vlnami (`JTextField-Resizeable`), panel s posuvníky zajišťující mixování (`WaveMixPanel`), panel s kontrolními zaškrtávacími poli (`WaveButtonPanel`), panel referenčních hodnot

(`VerticalReferencesPanel`) a samotná vlna (`WavePanel`).



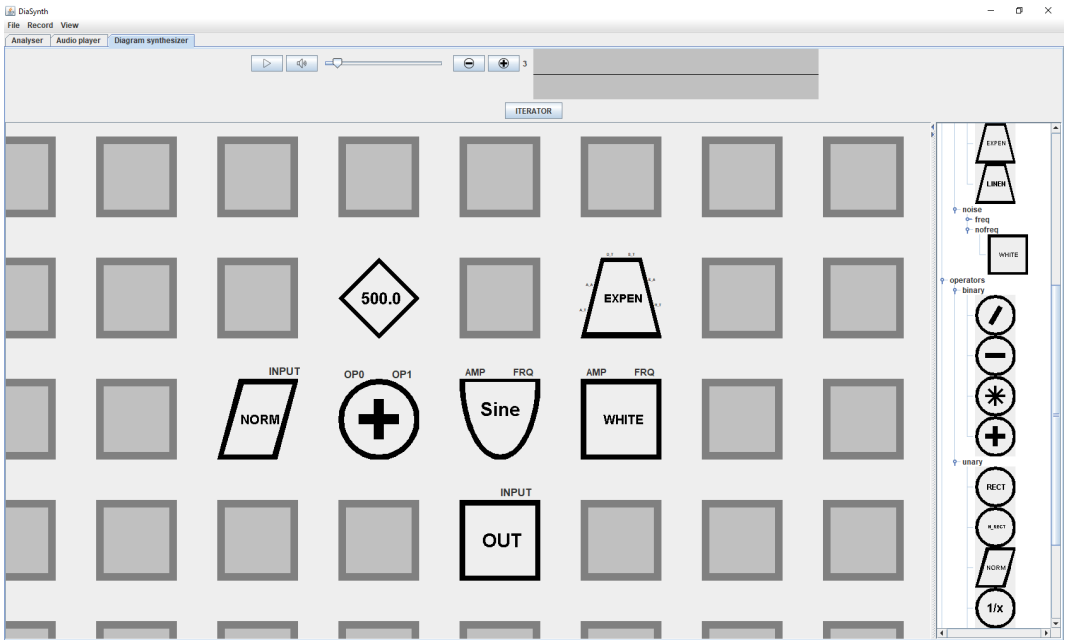
Obrázek 4.8: Diagram popisující všechny části související s vlnou

Audio vlákno je reprezentované třídou `PlayerAudioThread` a k mixování využívá metodu `performMixing`. Způsob mixování je určen implementací metod z rozhraní `AudioMixerIFace`, které má hned několik metod pro mixování samplů. Implementovali jsme více mixerů. Nakonec jsme se rozhodli pro mixování pomocí třídy `MixerWithPostProcessingSumDivision`, která sečtené samply dělí součtem multiplikativních faktorů pro daný kanál. Multiplikativní faktory jsou dány hodnotami posuvníků určených k mixování. Abychom nepočítali součet pokaždé znova, spočítáme ho pouze při změně, tj. když uživatel pohne nějakým z posuvníků pro mixování. Informaci o změně na posuvníku předáváme zavoláním metody `update` na třídě reprezentující mixer.

Část se syntezátorem je reprezentovaná třídou `SynthesizerMainPanel`, potomek třídy `JPanel`. Jak okno se syntezátorem vypadá můžeme vidět na obrázku 4.9. Rozmístění komponent v rámci třídy je stejně jako v případě přehrávače řešeno pomocí `GridBagLayout` a jednotlivé komponenty můžeme vidět v diagramu 4.10.

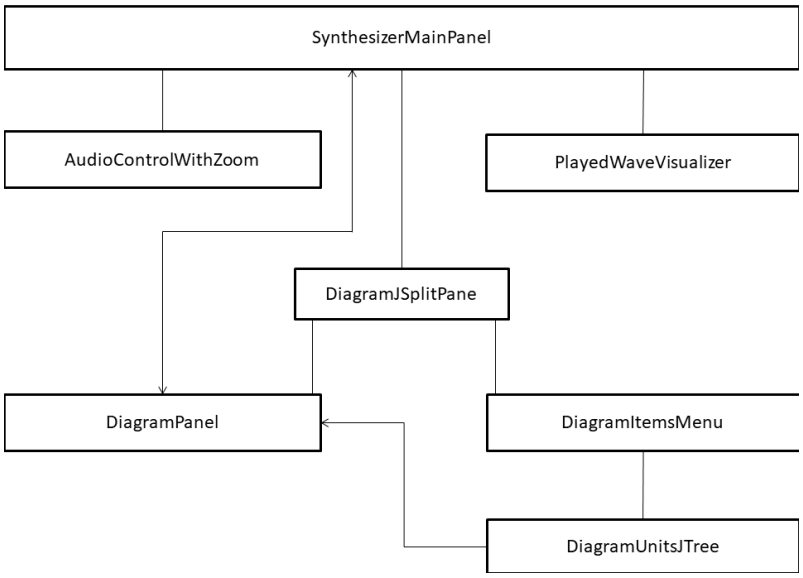
Ve vrchní části najdeme panel s tlačítky reprezentovaný třídou `AudioControlPanelWithZoom`, který se od panelu s tlačítky v přehrávači liší pouze absencí měřiče decibelů. Napravo od tlačítek vidíme instanci třídy `PlayedWaveVisualizer` starající se o vykreslování přehrávané vlny. Níže najdeme tlačítko pro iteraci přes prvky diagramu. Spodní část celého panelu reprezentujeme třídou `DiagramJSplitPane`, potomek třídy `JSplitPane`. Na jeho levé straně najdeme diagram reprezentovaný třídou `DiagramPanel`, potomek třídy `JLayeredPane`. Na jeho pravé straně `DiagramItemsMenu`, potomek třídy `JScrollPane`. Obsah `DiagramItemsMenu` je určen třídou `DiagramUnitsJTree`, která je potomek třídy `JTree`, a stará se o sestavení a zobrazení hierarchie výpočetních jednotek v rámci diagramu.

Třída `DiagramPanel` se stará o mnoho věcí, například vykreslování diagramu a kontrolu správnosti při připojování a přidávání panelů. Zjednodušeně řečeno se stará o vše ohledně diagramu kromě syntézy. Pro uvedení této třídy do kon-

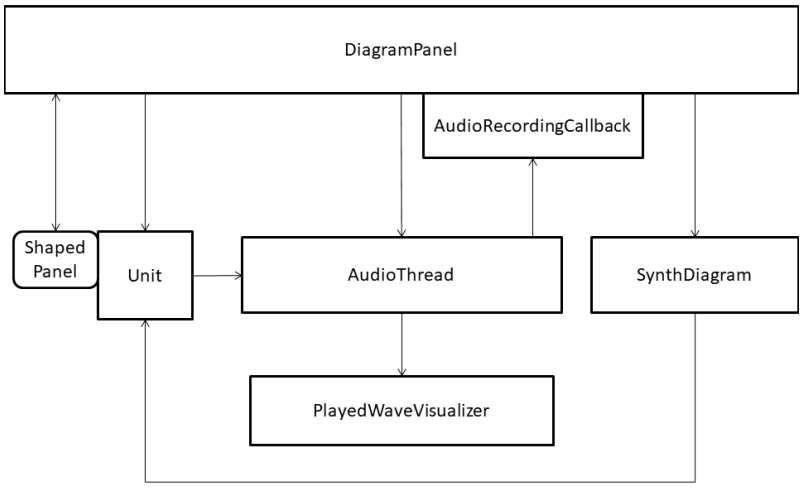


Obrázek 4.9: Okno se syntezátorem

textu ostatních tříd se můžeme podívat do diagramu 4.11, kde `AudioThread` je audio vlákno syntezátoru a `SynthDiagram` je třída starající se o generování hodnot na základě výpočetních jednotek v diagramu. Výpočetní jednotky jsou potomci třídy `Unit`. Třída `MovableJPanel` reprezentuje panel, kterým můžeme v rámci diagramu pohybovat. Dědí od třídy `MovableJPanelBase` stejně jako třída `ReferenceMovableJPanel`, která reprezentuje referenční panel, který nevykreslujeme a používáme ho pro nastavení umístění a velikosti ostatních panelů. Od třídy `MovableJPanel` dědí abstraktní třída `ShapedPanel`, díky které můžeme měnit tvary panelů. Právě potomky třídy `ShapedPanel` používáme v práci pro reprezentaci výpočetních jednotek z hlediska GUI. Diagram znázorňující tuto dědičnost a komunikaci s třídou `Unit` můžeme vidět na obrázku 4.12.



Obrázek 4.10: Diagram popisující syntezátor z pohledu třídy SynthesizerMainPanel

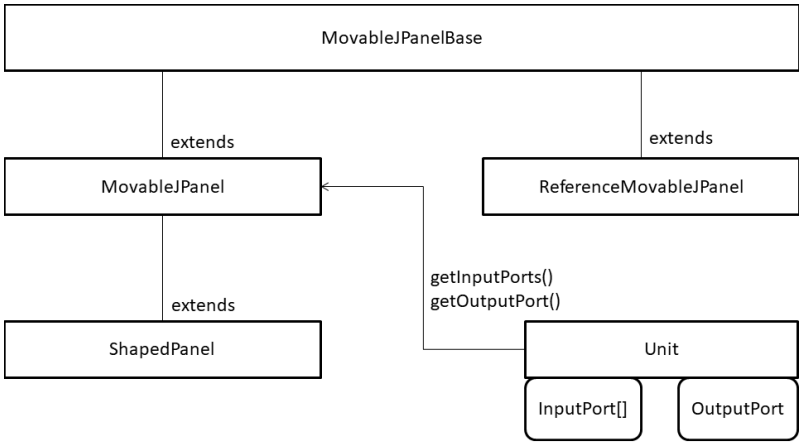


Obrázek 4.11: Diagram popisující syntezátor

Třída `MovableJPanel` je lehce spjatá s třídou `Unit` reprezentující generátor z hlediska syntézy. Od třídy Unit získává informace o připojeních skrze rozhraní `UnitViewForGUIIFace`. K reprezentaci připojení slouží porty, které nejsou nic jiného než koncové body připojení panelů v diagramu. Porty pro připojení reprezentujeme třídami `InputPort` a `OutputPort`.

V souvislosti s porty nesmíme zapomenout zmínit statickou metodu `choosePort` na třídě `PortChooser`, která slouží k zobrazení dialogu pro vybraní portu při připojování, či odstraňování.

Třídy `InputPort` a `OutputPort` dědí od společné třídy Port. Do třídy `InputPort` můžeme připojit maximálně jeden `OutputPort`, který reprezentujeme



Obrázek 4.12: Diagram popisující dědičnost okolo třídy MovableJPanel

proměnnou „*connectedPort*“. `OutputPort` může být připojen na nula až nekonečno jiných vstupních portů, tedy potřebujeme kolekci, kde si je všechny zaznamenáme.

Ještě je dobré zmínit, že máme několik tříd, které dědí od třídy `InputPort`. Takové třídy reprezentují nějaký specifický port, například `FrequencyInputPort` reprezentuje vstupní port, který dostává frekvenci. Od třídy `InputPort` se liší pouze specifikací parametrů v konstruktoru. Parametry jsou tooltip, výchozí hodnota, jméno a zkratka. Všechny implementované vstupní porty můžeme vidět v balíčku `synthesizer.gui.diagram.panels.port.ports`.

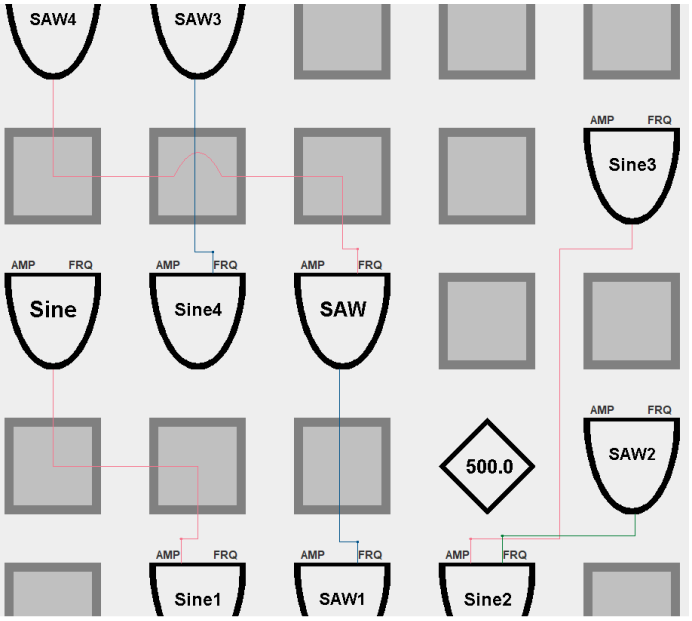
Ovšem nyní jsme mluvili pouze o informaci samotného připojení, ještě potřebujeme vědět kudy připojení povede, k tomu slouží třída `Cable`. Cesty si pamatujeme pouze na třídě `OutputPort`. K samotné reprezentaci cesty uvnitř třídy `Cable` používáme třídu `Path2D` ze standardní knihovny. K vykreslení cest dochází v metodě `paint`, kterou nalezneme uvnitř třídy `DiagramPanel`. Cesty jsou vyvýšeny a různě obarveny v závislosti na počtu překrytí s ostatními cestami. Pokud dojde ke kolizi horizontální části cesty s vertikální částí, vykreslíme oblouk na horizontální části kabelu v místě kolize.

Máme tři typy cest:

```
public enum CableType {
        STRAIGHT_LINE,
        AISLE_ALGORITHM,
        ADVANCED_ALGORITHM
}
```

Všechny typy cest můžeme vidět na obrázku 4.13.

Třída `Cable` obsahuje relativní i absolutní cestu, tím pádem není nutné počítat, kudy vede cesta při libovolném pohybu po ploše diagramu. Pouze stačí přepočítat relativní cestu na absolutní, o což se stará metoda `setAbsolutePathBasedOnRelativePath` na třídě `Cable`.

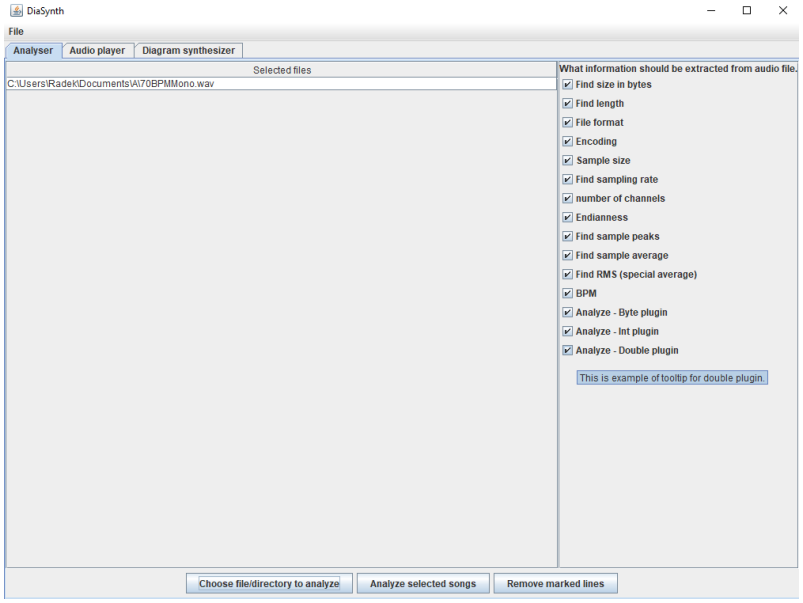


Obrázek 4.13: Ukázky různých cest kabelů spojujících panely. Mezi Saw4 a Saw je cesta známá pod názvem *ADVANCED_ALGORITHM*. Mezi Saw3 a Sine4 je cesta typu *STRAIGHT_LINE* a konečně mezi Sine3 a Sine2 je cesta typu *AISLE_ALGORITHM*.

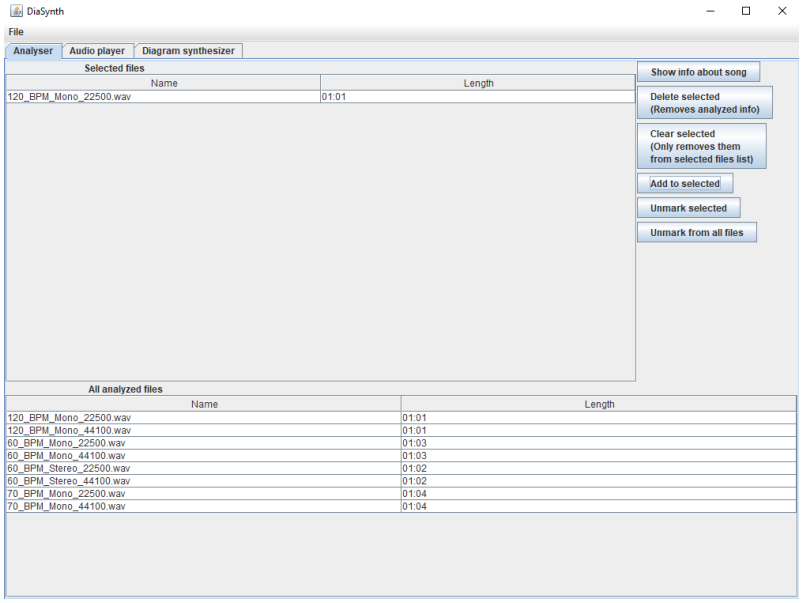
4.2.1 Analyzátor

Analyzátor je z hlediska návrhu velice jednoduchý. `AnalyzerMainPanel` je hlavní třídou, jak jsme mohli vidět na diagramu 4.1 ukazující program Diasynth na nejvyšší úrovni. Tato třída mění svůj obsah na základě toho, jaká záložka je momentálně otevřená, viz diagram níže 4.16. Otevřená záložka je buď `AnalyzerPanel`, nebo `SongLibraryPanel`. Můžeme je vidět na obrázcích 4.14 a 4.15. Před přesunutím na třídy samotné se zmíníme o komunikaci mezi těmito dvěma třídami. Pokud ve třídě `AnalyzerPanel` zanalyzujeme nové písničky, musíme je přidat do knihovny analyzovaných skladeb. K tomu používáme návrhový vzor observer. Třídě `Analyzer` předáme v konstruktoru observery s `update` metodou, kterou zavoláme, kdykoliv dojde k analýze nějakých souborů. O notifikaci observerů se stará třída `DataModelSubject`, přesněji její metoda `notifyObservers`. Observery jsou reprezentované třídou `DataModelObserver`. Komunikace je znázorněná diagramem 4.17.

`AnalyzerPanel` obsahuje `JTable`, do které přidáváme soubory, které chceme analyzovat. Na pravé straně okna najdeme zaškrtávací políčka, které jsou získány buď ze zásuvných modulů, nebo jsou v kódu napevno, což je pozůstatek z dob, kdy ještě nebylo v plánu přidávání nových algoritmů pro analýzu formou zásuvných modulů.



Obrázek 4.14: Okno pro výběr souborů k analýze

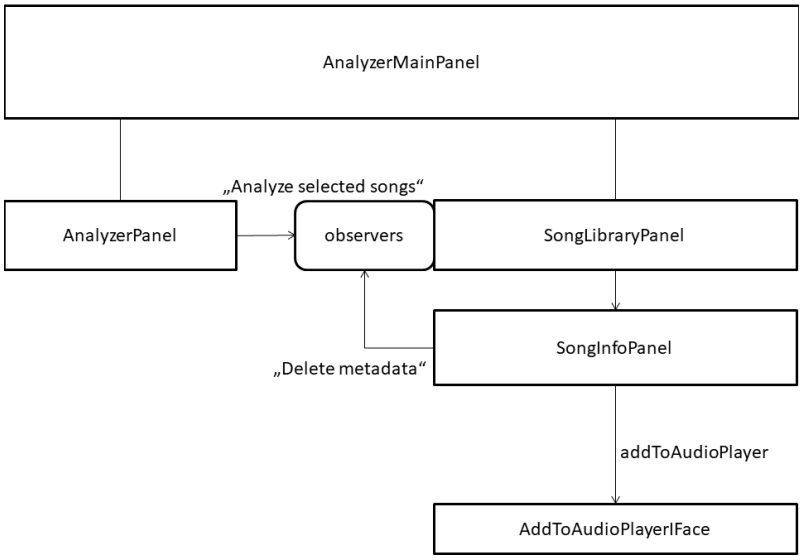


Obrázek 4.15: Okno s knihovnou dříve analyzovaných souborů

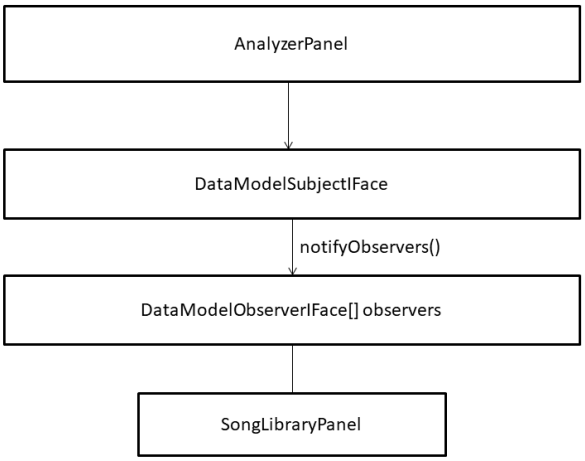
Kód pro přidávání zaškrtávacích políček na základě zásuvných modulů je jednoduchý, varianta s doublovými samply:

```
private void loadPlugins() {
        // ...
        List<AnalyzerDoublePluginIFace> doublePlugins;
        doublePlugins = AnalyzerDoublePluginIFace.loadPlugins();

        for(AnalyzerDoublePluginIFace p : doublePlugins) {
                JCheckBox checkBox = new JCheckBox(p.getName());
                checkBox.setToolTipText(p.getTooltip());
                checkBox.setSelected(true);
```



Obrázek 4.16: Diagram popisující analyzátor



Obrázek 4.17: Diagram popisující předávání informací pomocí návrhového vzoru observer

```
                doublePluginPairs.add(new Pair<>(checkBox, p));
        }
        // ...
}
```

Samotnou analýzu zaškrtnutých políček zásuvných modulů řeší metody `runSelectedPlugins*`. Místo hvězdičky doplníme Byte, Int, nebo Double podle typu zásuvného modulu. Typy se liší v reprezentaci vstupní vlny.

Analýza obecně probíhá tak, že po stisknutí tlačítka „`Analyze selected songs`“ se spustí kód třídy implementující rozhraní `ActionListener`, ve kterém vezmeme soubory z data modelu `JTable` a zavoláme na ně metodu `analyze`, která

bere jako parametr cestu k souboru. Postupně přidáme výsledky v podobě párů (jméno, hodnota) do kolekce list, jejíž obsah vložíme do načteného XML stromu, který následně vložíme do souboru ANALYZED_AUDIO.xml. V rámci přidávání se podíváme, jestli už daný uzel existuje, pokud ano, přepíšeme ho, jinak přidáme nový. Pokud došlo ke změně, upozorníme observery.

```
int rowCount = dataModel.getRowCount();
boolean anyFileAnalyzed = false;
for(int i = 0; i < rowCount; i++) {
        analyze((String)dataModel.getValueAt(0, 0));
        anyFileAnalyzed = true;
        dataModel.removeRow(0);
}
if(anyFileAnalyzed) {
        AnalyzerXML.createXMLFile(ANALYZED_AUDIO_XML_FILENAME,
                        AnalyzerXML.getXMLDoc().getFirstChild(), frame);
        subject.notifyObservers();
}
```

Návrh třídy `SongLibraryPanel` je velice přímočarý a nepotřebuje žádný komentář. Pokud se chceme podívat, jak nějaká akce v rámci této části funguje, stačí si najít odpovídající `ActionListener` pro dané tlačítko. Pouze se zmíníme o tom, že zobrazení analyzovaných informací o souboru řeší třída `SongInfoPanel`. Tato třída si načte informace o uživatelem vybraném souboru z XML stromu, přidá je do data modelu `JTable` a zobrazí. V rámci třídy `SongInfoPanel` můžeme mazat analyzovaná metadata, takže stejně jako v případě třídy `AnalyzerPanel` používáme návrhový vzor observer.

Práci s XML řešíme ve třídě `AnalyzerXML`. Implementace třídy `AnalyzerXML` je technický detail, takže o té se nebudeme zmiňovat, pouze připomeneme, že používáme Java DOM.

4.2.2 Přehrávač

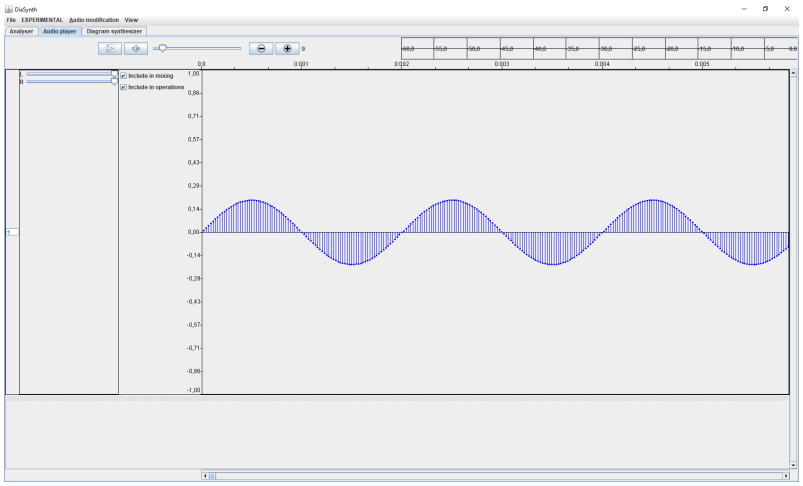
Nyní se podrobněji podíváme na části přehrávače popsané v úvodu sekce. Zaměříme se především na GUI, neboť jeho implementace tvoří většinu kódu, který navíc může být bez vysvětlení v textu poměrně složitý na pochopení. Některé problémy, o kterých se zde zmíníme jen lehce, probereme v sekci o řešení vybraných problémů 4.4 více do hloubky. Přehrávač můžeme vidět na obrázku 4.18.

Uvnitř třídy `AudioPlayerPanel` musíme nějak uchovat všechny vlny, učiníme tak následovně.

```
private List<WaveMainPanel> waves;
private List<JSplitPane> splitters;
```

Pořadí v kolekci reprezentované položkou „*splitters*“ je od nejméně zanořeného `JSplitPane` po nejvíce zanořený. Velikost vertikální mezery mezi vlnami je daná konstantou „*DIVIDER_SIZE*“.

Kvůli skládání `JSplitPane` musíme speciálně řešit přidání první vlny (`addFirstWave`) a přidání ostatních vln (`addNonFirstWave`). Ovšem toto rozdělení



Obrázek 4.18: Okno s přehrávačem

se provede v metodě `addWave`, tedy se jedná spíše implementační detail. Stejně tak u mazání rozlišujeme, jestli mažeme poslední vlnu, nebo ne. To řeší metoda `removeWave`, na jejímž konci voláme metodu `postDeletionAction`, která zajistí aktualizaci proměnných pro mixování a dalších věcí ohledně přehrávání. První vlnou myslíme, že délka kolekce „*waves*“ je rovna jedné, nikoli že se jedná o nultý index. Pro snadnější implementaci a zachování velikostí provádíme při mazání prohození mazané vlny s poslední, což odpovídá volání metody `moveSwapToLastAndRemove`. Prohazujeme postupně s vedlejšími vlnami, takže po vymazání nedojde ke změně pořadí vln.

Pro `JSplitPane` bylo nutné napsat listenery, tentokrát rozlišujeme tři případy. Jedná se buď o první (`FirstSplitterChangeListener`), či poslední vlnu (`CompoundLastSplitterChangeListener`), nebo o vlnu rozdílnou od první a poslední (`CompoundSplitterChangeListener`). Musíme rozlišovat, jestli se jedná o poslední vlnu, nebo ne, neboť v rámci přehrávače chceme podporovat funkcionalitu „nekonečného“ roztažení poslední vlny směrem dolů. Tato funkcionalita způsobila poměrně netriviální množství problémů, protože se nejedná o úplně běžnou funkcionalitu. Z tohoto důvodu jsme se museli starat o velikosti jednotlivých vln sami místo toho, aby je metody standardní knihovny řešily automaticky za nás. Všechny výše zmíněné listenery poslouchají události na rozdělení `JSplitPane`, na tzv. rozdělovači.

Implementace třídy `CompoundLastSplitterChangeListener` nás může zaskočit, neboť volá `mouseReleased` metodu na položce „*lastSplitterMouseAdapter*“. Právě tento `MouseAdapter` se stará o funkcionalitu nekonečného roztažení poslední vlny poslouchání událostí způsobených myší na rozdělovači. V rámci potomka třídy `ChangeListener` totiž není moc dobře možné implementovat danou funkcionalitu, neboť si musíme sami nastavovat pozici rozdělení posledního rozdělovače (`lastSplitter.setDividerLocation`), čímž vytvoříme další událost. Pak vlastně nevíme, co je opravdovým důvodem události, jestli skutečné roztažení, nebo umělé, způsobené zavoláním v rámci metody. Teď nás jistě napadá otázka, proč tedy třída `CompoundLastSplitterChangeListener` vůbec existuje. Důvod je poměrně zajímavý. Totiž v jazyce Java se nachází chyba spočívající v tom, že můžeme roztáhnout `JSplitPane` (ten poslední), i když klikneme jeden pixel pod rozdělovač. Poznáme to tak, že se nezmění kurzor, ale i přesto můžeme rozdělova-

čem `JSplitPane` pohybovat. Ovšem takové roztažení `mouse listener` poslouchající na rozdělovači nezaznamená, tím pádem nenastavíme správně velikosti a dojde ke špatnému vykreslení vlny. V rámci `CompoundLastSplitterChangeListener` pak rozlišujeme, jestli tento případ nastal, nebo šlo o klasické roztažení, v tom případě už nesmíme znovu volat `mouseReleased` metodu.

Pro posouvání při roztahování vlny mimo viditelné okno slouží metody `moveJScrollPaneUp` a `moveJScrollPaneDown`.

O přehrávání se stará třída `PlayerAudioThread` reprezentující audio vlákno. Dědí od třídy `Thread`. Jak mixování funguje a v čem se liší od audio vlákna pro syntezátor uvidíme v sekci 4.4.5. Hlavní metodou vlákna je `run`, uvnitř které zavoláme metodu `playAudioLoop`. Zde ve smyčce provádíme mixování metodou `performMixing`. Výsledky mixování následně zapíšeme do `SourceDataLine`. Navíc musíme posouvat úsečku značící současnou pozici přehrávání, o to se stará následující kód.

```
if(filledWithWaveSamples) {
        if (userClickedWave) {
                switchToUserSelectedSample();
        } else {
                timeLineX += timeLinePixelsPerPlayPart;
        }
}
repaint();
```

Dalšími metodami třídy jsou metody typické pro audio vlákno: `play`, `pause`, `outputFormatChanged`.

Ještě je dobré zmínit, že třída dostává v konstruktoru hodnoty, podle kterých nastaví parametry přehrávání. Jedná se o velikost přehrávaného bufferu a maximální čas, resp. počet samplů, který může být v jeden okamžik uvnitř `SourceDataLine`. První parametr ovlivňuje vytíženost audio vlákna a druhý reakční dobu, tj. za jak dlouho se projeví uživatelem provedená akce. Akce jsou například změna pozice přehrávání, či jeho úplné zastavení. Tyto parametry se převedou na hodnoty, které použijeme pro nalezení správných hodnot pro daný výstupní formát. Viz následující kód:

```
public PlayerAudioThread(boolean shouldPause, int maxPlayTimeInMs,
                         int internalBufferSizeInMs) {
  maxPlayTimeDivFactor = AudioThread.
             convertTimeInMsToDivFactor(maxPlayTimeInMs);
  internalBufferTimeDivFactor = AudioThread.
             convertTimeInMsToDivFactor(internalBufferSizeInMs);
  // ...
}

public void outputFormatChanged() {
  int maxByteCountInAudioLine = AudioThread.
             convertMsToByteLen(sampleRate, frameSize,
                                maxPlayTimeDivFactor);

  int audioArrLen = AudioThread.
```

```
            convertMsToByteLen(sampleRate, frameSize,
                               internalBufferTimeDivFactor);
  // ...
}
```

Dále se podíváme na označování části vlny a přetáčení přehrávání, o což se stará třída `WavePanelMouseListener`, která poslouchá události na panelu s vlnami (`WavePanel`) způsobené myší. Metoda `mouseClicked` řeší změnu místa přehrávání. Metody `mousePressed` a `mouseReleased` řeší označování vlny. V rámci těchto metod měníme proměnné na třídě reprezentující přehrávač. Samozřejmě tak činíme přes metody, nikoliv přímým přístupem na položky. Při označování se jedná o proměnné „*markStartXPixel*“, „*markStartXSample*“, „*markEndXPixel*“ a „*markEndXSample*“. Při změně pozice přehrávání zavoláme metodu `processUserClickedWaveEvent`. V této metodě mimo jiné nastavíme příznak „*userClickedWave*“ kontrolovaný v audio vlákně.

Jednou z vlastností přehrávače je možnost prohazovat pořadí vln a to jak táhnutím, tak pomocí textového okna na levé straně zvukových stop. Prohození pomocí textového okna je zařízeno zavoláním metody `swapSplitterComponents`. Prohazování tažením řeší potomek třídy `MouseAdapter` definovaný uvnitř konstruktoru třídy `WaveMainPanel`. Přesněji se jedná o metodu `mouseDragged`, která volá `tryMoveSwap`.

S označením části vlny souvisí třída `ClipboardWave`, která reprezentuje zkopírovanou vlnu. Pamatujeme si indexy zkopírované vlny. Nemáme v paměti celou vlnu dvakrát. Z toho důvodu při změně vlny dojde k odstranění vlny z „clipboard“.

O přidávání zásuvných modulů pro práci s vlnami se starají metody `addTwoInputWavesPlugins` a `addSingleInputWavePlugins`.

```
private void addPlugin(OperationOnWavesPluginIFace pluginToAdd,
                       JMenu menu) {
   JMenuItem menuItem = new JMenuItem(pluginToAdd.getPluginName());
   withInputWaveMenuItems.add(menuItem);
   menuItem.setToolTipText(pluginToAdd.getPluginTooltip());

   menuItem.addActionListener(new ActionListener() {
      OperationOnWavesPluginIFace plugin = pluginToAdd;

      @Override
      public void actionPerformed(ActionEvent e) {
         // To reset the plugin
         Class<?> clazz = plugin.getClass();
         try {
            Constructor<?> constructor = clazz.getConstructor();
            if (constructor == null) {
               // The log message in code is a bit longer than this
               MyLogger.log("No constructor", 0);
               return;
            }

            plugin = (OperationOnWavesPluginIFace) clazz.newInstance();
         }
```

```
            catch(Exception exception) {
               MyLogger.logException(exception);
               return;
            }
            // To reset the plugin

            boolean canContinueOperation;
            canContinueOperation = loadPluginParameters(plugin, true);
            if(canContinueOperation) {
               stopAndModifyAudio(false, new ModifyAudioIFace() {
                  @Override
                  public void modifyAudio() {
                     performOperationInternal(plugin);
                  }
               }, true, false);
            }
         }
      });

     menu.add(menuItem);
}


private void addPlugins(JMenu menu) {
   addTwoInputWavesPlugins(menu);
   menu.addSeparator();
   addSingleInputWavePlugins(menu);
}

private void addTwoInputWavesPlugins(JMenu menu) {
   List<OperationOnWavesPluginIFace> plugins;
   plugins = OperationOnWavesPluginIFace.loadPlugins();
   for(OperationOnWavesPluginIFace plugin : plugins) {
      addPlugin(plugin, menu);
   }
}
```

Při implementaci `addSingleInputWavePlugins` akorát vyměníme třídu `OperationOnWavesPluginIFace` za třídu `OperationOnWavePluginIFace`. Samotný běh zásuvného modulu po jeho spuštění uživatelem je zařízen metodou `performOperationInternal`. Pluginy voláme uvnitř metody `stopAndModifyAudio`, která zastaví přehrávání, provede vstupní operaci a v závislosti na parametrech buď začne znovu přehrávat, nebo ne. Pro budoucí práci na programu je dobré vědět, že při změně délky stopy je nutné zavolat metodu `postProcessingAfterChangingWaveLength`.

O ukládání vlny se stará metoda `addSaveFileToMenu`, respektive `ActionListener`, který v rámci metody vytvoříme.

Ještě před přesunem na stopy popíšeme horizontální pohyb vln a s ním související přiblížení. Připomínáme, že pokud budeme mluvit o horizontálním posuvníku, myslíme horizontální posuvník uvnitř třídy `WaveScrollerWrapperPanel`,

nikoliv posuvník na `JScrollPane` s vlnami. O přiblížení se stará metoda `updateZoom`, v níž můžeme vidět příznak „*canZoom*“. Ten zaručí, že nedojde k dalšímu přiblížení, pokud předchozí ještě plně neskončilo. O znovu povolení přiblížení se stará položka „*waveScrollerWrapperPanel*“, neboť přiblížení je kombinace zvětšení maximálního možného posunutí a samotného posunutí. Maximální hodnotu horizontálního posuvníku změníme voláním metody `setWaveScrollerPanelsSizes` na třídě `AudioPlayerPanel`. Na zavolání této metody zareaguje `ComponentListener` poslouchající na panelu reprezentující obsah horizontálního posuvníku. Událost posunutí řeší `HorizontalBarAdjustmentListener`, který poslouchá na samotném horizontálním posuvníku uvnitř třídy `WaveScrollerWrapperPanel`. K přibližování se vrátíme, až budeme mluvit o třídě reprezentující vlnu.

Třída `WaveScrollerWrapperPanel` dědí od `JPanel` a najdeme jí hned pod `JScrollPane`, kam umisťujeme vlny. Implementace je poměrně zajímavá. Máme tři prázdné panely s nulovou výškou a šířkou odpovídající daným částem z `JScrollPane`: panel před skutečnou vykreslovanou vlnou, panel na místě vykreslované vlny a panel pro vertikální posuvník přehrávače. Panel na místě vykreslované vlny má velikost totožnou s absolutní délkou vykreslované vlny a je prvkem `JScrollPane`, který reprezentuje horizontální posuvník. Tento `JScrollPane` najdeme jako položku s jménem „*waveScroller*“.

Nyní se konečně můžeme podívat na komponentu reprezentující celou stopu, tedy na třídu `WaveMainPanel`. Tato třída se skládá z několika komponent, které jsou rozmístěny pomocí `GridBagLayout`. Můžeme je vidět v diagramu 4.19, který jsme už mohli spatřit dříve, ale pro pohodlnost ho přikládáme znovu. Postupně popíšeme komponenty zleva doprava.

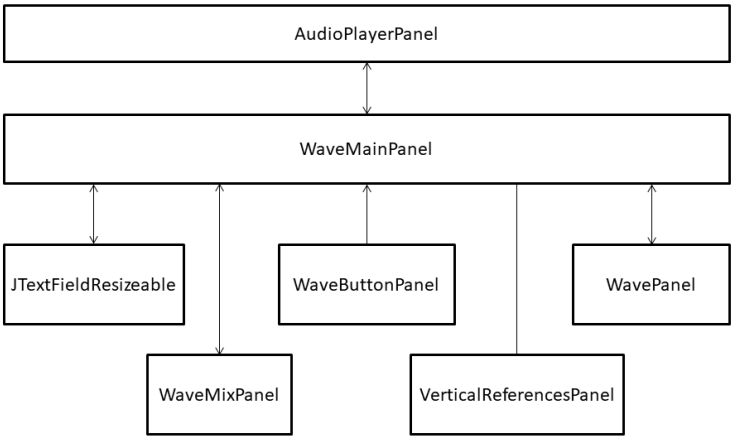
První komponentou je třída `JTextFieldResizeable` reprezentující číslo stopy v rámci přehrávače. Druhou je třída `WaveMixPanel`, která obsahuje posuvníky kontrolující multiplikativní faktor pro výstupní kanály dané stopy. Například pro stereo je levý kanál reprezentovaný prvním posuvníkem a pravý druhým. Při nastavení prvního posuvníku na poloviční hodnotu bude do levého kanálu posílaná vlna s poloviční amplitudou, podobně pro pravý kanál. Další třídou v rámci `WaveMainPanel` je `WaveButtonPanel`, která obsahuje dvě zaškrtávací políčka. První určuje, zda-li máme danou vlnou použít v mixování a druhé, jestli jí máme modifikovat při modifikaci vln. Při vyškrtnutí prvního políčka se na dané vlně nezobrazuje úsečka udávající danou přehrávanou pozici. Při vyškrtnutí druhého políčka se nezobrazuje označení části vlny.

Poslední třídou před samotnou vlnou je třída `VerticalReferencesPanel`, která reprezentuje referenční hodnoty z intervalu [-1, 1] pro vlnu, respektive z intervalu zadaného v konstruktoru.

Poslední třídou je `WavePanel`, která reprezentuje samotnou vlnu. Samply dané vlny jsou uloženy v proměnné „*doubleWave*“ typu `DoubleWave`.

Nyní se podíváme na věci okolo vykreslování hodnot. Pro představu můžeme nahlédnout do diagramu 4.20. Datovou strukturu `ShiftBufferDouble` rozebereme později v sekci 4.4.1. Nyní jen stačí vědět, že se jedná o buffer, který obsahuje hodnoty k vykreslení. Obsahuje nejen právě viditelné hodnoty, ale i hodnoty v blízkém okolí. Okno viditelných hodnot posouváme na základě pohybu posuvníku.

Vykreslované hodnoty jsou uloženy uvnitř následujících tříd:



Obrázek 4.19: Diagram popisující všechny části související s vlnou

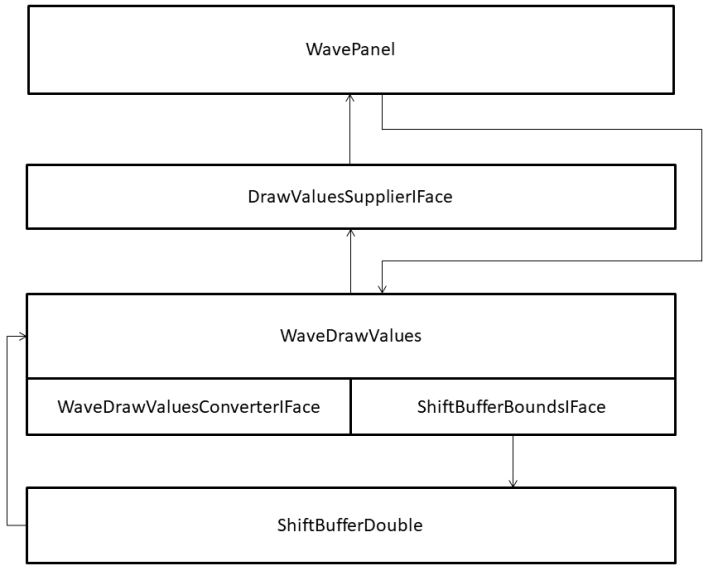
```
private WaveDrawValuesIndividual drawValuesIndividual;
private WaveDrawValuesAggregated drawValuesAggregated;
private WaveDrawValues currentDrawValues;
```

Proměnná „*currentDrawValues*“ ukazuje na jednu ze tříd zmíněných nad ní v závislosti na tom, jestli se momentálně vykreslují agregované hodnoty, nebo individuální samply. Ovšem výše zmíněné třídy potřebují kvůli naplnění novými hodnotami při posouvání dvě následující položky.

```
private DrawValuesSupplierIndividual drawValuesSupplierIndividual;
private DrawValuesSupplierAggregated drawValuesSupplierAggregated;
```

Tyto třídy se starají o převod mezi pixely a samply, což zařizuje metoda `convertFromPixelToIndexInAudio`, a především o naplnění posuvného bufferu novými hodnotami k vykreslení, o což se stará metoda `fillBufferWithValuesToDraw`. V případě `DrawValuesSupplierIndividual` je její implementace triviální a stačí pouze zkopírovat samply z vlny. V případě `DrawValuesSupplierAggregated` musíme spočítat agregační hodnoty, k čemuž využijeme statickou metodu `findExtremesInValues`. Zažádání o nové hodnoty proběhne, kdykoliv viditelné okno přesáhne hranice bufferu. Dalšími hlavními metodami na třídě `WaveDrawValues` jsou `waveResize`, která se zavolá při změně velikosti vlny. `drawSamples`, která vykreslí viditelné hodnoty v bufferu a pak samozřejmě `shiftBuffer`, kterou zavoláme při posunu horizontálního posuvníku.

Samotný posuvný buffer je pak reprezentovaný třídou `ShiftBufferDouble`. Je implementovaný tak, jak bychom čekali, tj. máme „*startIndex*“ a „*endIndex*“ reprezentující viditelné okno a metodu `updateStartIndex` měnící „*startIndex*“ a s ním i „*endIndex*“. Velikost viditelného okna je daná hodnotou `windowSize` a je předána jako parametr v konstruktoru. Celková velikost bufferu je pak dána jako:



Obrázek 4.20: Diagram popisující komunikaci mezi třídami v rámci vlny

```
int bufferSize = windowSize * (2 * windowCountToTheRight + 1)
```

Dále máme proměnné „*minLeftIndex*“ a „*maxRightIndex*“ reprezentující indexy v poli s validními hodnotami a metodu `setBounds`, která tyto hranice v případě potřeby nastaví na korektní hodnoty. Samozřejmě máme buffer a metodu `getBuffer`. Navíc si můžeme všimnout, že třída ke svému chodu potřebuje dostat v konstruktoru instanci třídy `ShiftBufferBoundsIFace`, neboť posuvný buffer musí nějakým způsobem získat nové hodnoty při posunutí viditelného okna mimo hranice bufferu.

Nyní se podíváme, co se děje ve třídě `WavePanel` při posouvání a přibližování.

Při posouvání horizontálního posuvníku zavoláme na třídě `WavePanel` metodu `updateWaveDrawValues`, která na třídě `WaveDrawValues` zavolá metodu `shiftBuffer`. Změnu v pixelech předáme jako parametr metody. Přibližování vlny se řeší voláním metody `updateZoom`, ve které se správně upraví velikosti vlny a spočte absolutní pozice po přiblížení. Přiblížení totiž mění nejen velikost, ale i pozici posuvníku. Pozice je navíc určená podle toho, jestli se přibližujeme na konec, začátek, doprostřed, nebo na pozici úsečky znázorňující právě přehrávanou část stopy. Při přiblížení musíme ještě zkontrolovat, jestli nedošlo ke změně typu vykreslování, tj. z vykreslování agregačních hodnot na vykreslování individuálních samplů, nebo naopak.

Při přiblížení může dojít k bliknutí, neboť na malý okamžik se posuneme na jinou pozici, než na které máme být po skončení přiblížení. Totiž zvětšení velikosti posuvníku a samotný posun jsou dva odlišné jevy, které se neprovádí ve stejný okamžik. Jedná se o vizuálně velice nepříjemný jev, proto ho bylo nutné nějakým způsobem vyřešit. Problém řešíme vytvořením obrázku stavu vlny před přiblížením, který následně zobrazíme místo samotné vlny po celou dobu přibližování. Obrázek je reprezentovaný proměnnou „*zoomBridgeImg*“ typu `Image`.

Samotné vykreslování uvnitř metody `paintComponent` ve třídě `WavePanel` je přímočaré, akorát stojí za zmínku volání metody

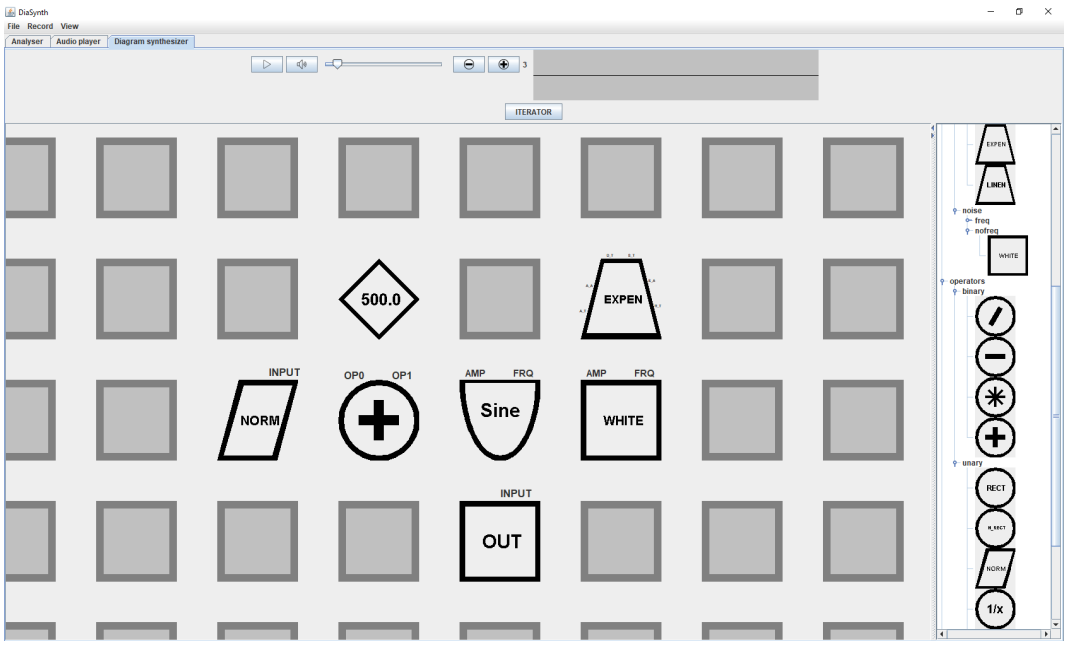
`visibleWidthChangedCallback`, ve které se zkontroluje, jestli nedošlo k nějaké změně viditelné velikosti vlny. Pokud došlo, opravíme velikosti a stav třídy.

Dále se ve třídě `WavePanel` řeší některé operace související se změnou délky vlny, přesněji se jedná o metody: `paste`, `pasteWithOverwriting`, `move` a `remove`.

Poslední třídou, kterou v rámci přehrávače zmíníme, je `WavePanelPopupMenuActionsIFace`. Tato třída reprezentuje menu zobrazené po pravém kliknutí na vlnu.

4.2.3 Syntezátor

V této části popíšeme syntezátor, jehož úlohou je vytvářet zvuk na základě prvků, které uživatel pospojuje dohromady. O jaké prvky se konkrétně jedná a jak vypadají jejich výstupy, popíšeme později v sekcích 4.4.12 a 4.4.13. Na jednotlivé metody těchto výpočetních jednotek se pro účely psaní pluginů podíváme v rámci uživatelské dokumentaci 5.8.3. Syntezátor můžeme vidět na obrázku 4.21.



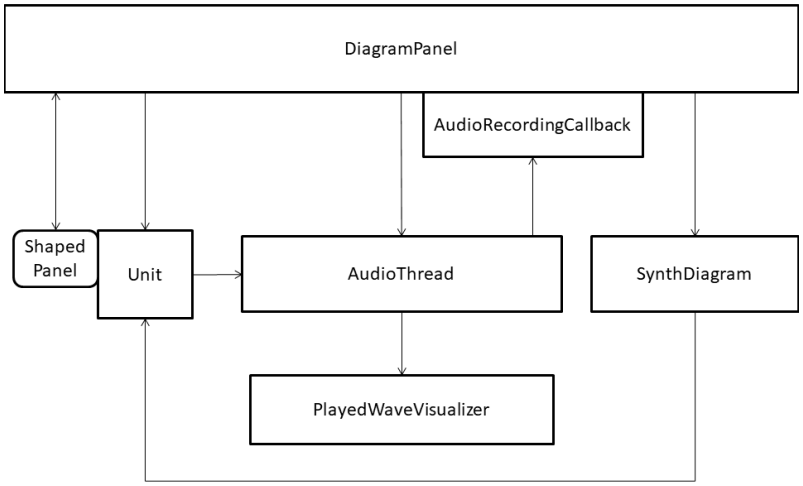
Obrázek 4.21: Okno se syntezátorem

Neprobereme zde důkladně všechny části syntezátoru. Některé zmíníme jen lehce a později je rozebereme důkladněji v sekci o řešení vybraných problémů 4.4.

`SynthesizerMainPanel`, potomek třídy `JPanel`, reprezentuje okno se syntezátorem. Stará se mimo jiné o všechny akce spojené se záložkami.

`SynthesizerMainPanel` se skládá z panelu s tlačítky, který odpovídá instanci třídy `AudioControlPanelWithZoom`. Dědí od `AudioControlPanel`, ke které přidává k dvě tlačítka pro přiblížení. Část s přehrávačem používá ještě více rozšířenou variantu `AudioControlPanelWithZoomAndDecibel`, která navíc obsahuje měřič decibelů. `AudioControlPanel` má tlačítko play/pause, posuvník pro kontrolu hlasitosti a tlačítko pro vypnutí zvuku.

Pro lepší pochopení následujícího textu se můžeme podívat do diagramu 4.22. Už jsme ho mohli spatřit dříve, ale pro lepší orientaci v textu ho přidáváme znovu. Diagram popisuje základní části syntezátoru, o kterých se budeme nyní bavit.



Obrázek 4.22: Diagram popisující syntezátor

Dále panel `SynthesizerMainPanel` obsahuje třídu `PlayedWaveVisualizer`, která slouží pro vizualizaci přehrávané vlny. Stejně jako u vln v přehrávači používáme instanci třídy `WaveDrawValuesAggregated`, ale musíme si napsat jinou implementaci rozhraní `DrawValuesSupplierIFace`.

Vykreslování generované vlny probíhá následujícím způsobem. Máme frontu, do které audio vlákno dává samply těsně před jejich přehráním. Vždy při volání metody `paintComponent` posuneme interní buffer. Kód pro posunutí vypadá následovně.

```
int sampleShift = lastPushedSample[currentChannel] -
                  lastDrawnSample[currentChannel];
drawValuesWrappers[currentChannel].
                  shiftBuffer((int)(sampleShift / samplesPerPixel));
```

Tímto kódem předáváme informaci, kolik pixelů přišlo od posledního vykreslení a podle toho se zachováme v metodě `fillBufferWithValuesToDraw`. Buď buffer posuneme a z části naplníme, pokud od posledního vykreslení neproběhlo více samplů než vykreslujeme v jeden okamžik, nebo ho celý přepíšeme posledními hodnotami, které máme ve frontě.

Kvůli pravidelnému vykreslování necháváme běžet časovač (timer). Ten v pravidelných intervalech volá metodu `repaint` na třídě `JPanel`, která požádá o překreslení panelu s vlnou. K překreslení nemusí dojít okamžitě, ale vždy k němu dojde v nějakém krátkém okamžiku. Časovač zastavíme při stopnutí audio vlákna, tím ušetříme zbytečné vynucování vykreslování.

Funkcionalitu iterátoru přes jednotky umístěné v diagramu implementujeme pomocí třídy `JList`, která dostává za model námi napsanou třídu `ListModelForPanels`.

Poslední částí panelu `SynthesizerMainPanel` je `DiagramJSplitPane`, potomek třídy `JSplitPane`. Na jeho levé straně najdeme třídu `DiagramPanel` reprezentující diagram a na pravé třídu `DiagramItemsMenu`, což je menu s výpočetními jednotkami, které lze do diagramu přidat.

Třída `DiagramItemsMenu` je pouze potomek `JSrollPane`. Její skutečný obsah tvoří třída `DiagramUnitsJTree`, která je potomkem třídy `JTree`. Stará se o zobrazení hierarchie výpočetních jednotek. Mimo jiné obsahuje metody pro načtení pluginů do syntezátoru. Máme dva typy metod pro načítání pluginů. První pro spouštění ze spustitelného souboru .jar, druhý pro klasickou kompilaci s přístupem ke zdrojovým kódům. O vytvoření obsahu `JTree` se stará metoda `createTree`. Uvnitř voláme metodu `getRoot`, která načte zásuvné moduly a vytvoří kořen pro `JTree`.

Nyní konečně můžeme popsat třídu `DiagramPanel`, která reprezentuje diagram.

Na začátku třídy máme definované konstanty, kde asi nejzajímavější je položka „*STATIC_PANEL_MIN_WIDTH*“, která určuje minimální šířku panelu, samozřejmě existuje i „MAX“ varianta.

Mimo jiné zde najdeme metody pro nahrávání, jejichž implementace je triviální, proto je zde nebudeme popisovat. Způsoby nahrávání pouze krátce zmíníme v rámci řešení vybraných problémů 4.4.7.

Pro serializaci diagramu zde máme `save` a `load` metody z rozhraní `SerializeIFace`.

K vytvoření instance výpočetní jednotky z poskytnuté šablony za pomoci reflexe slouží metoda `createUnit`.

Kolekce se všemi výpočetními jednotkami z diagramu se jmenuje „*panels*“.

```
public class UnitArrayListSortedByY extends ArrayList<Unit> {
        public void repairUnitPosition(Unit u) {
                remove(u);
                add(u);
        }
        // ...
}

// Kvůli místu v textu nepíšeme jaké rozhraní implementuje
public class DiagramPanel extends JLayeredPane {
        UnitArrayListSortedByY panels;
}
```

`UnitArrayListSortedByY` je kolekce, která navíc zachovává vlastnost, že jsou všechny panely setříděny podle y-ové a sekundárně podle x-ové souřadnice. Při zavolání `add` se panel přidá na správné místo v kolekci. Ovšem při změně pozice nějakého panelu v rámci diagramu musíme zavolat metodu `repairUnitPosition`, čímž donutíme kolekci znovu přidat prvek a tím zachovat správné utřídění. Implementace není nejefektivnější, což nevadí, neboť tato metoda se bude volat vůči ostatním metodám programu velice zřídka.

K pohybu po diagramu slouží metody `moveX`, `moveY`. Používají je například metody `zoom`, `mouseDragged` a `tryMoveBoardUsingPolling`. Poslední z nich je použita v situaci, kdy se nemůžeme posouvat táhnutím myši, tj. při připojování a pohybu panelu. Převod táhnutí myši na pohyb po diagramu provádí metoda `mouseDragged`.

Metoda `zoom` se nejdříve přesune na pozici, na kterou ukazuje kurzor myši a poté provede přiblížení. Pokud můžeme přibližovat, respektive oddalovat, změníme velikosti všech panelu, posuneme panely a aktualizujeme absolutní cesty

připojení. Posun panelů po přiblížení provádíme na základě absolutní pozice tzv. referenčního panelu, který nevykreslujeme a celou dobu je na pozici [0, 0]. Pozicí [0, 0] myslíme diskrétní souřadnice panelů. Explicitně řečeno se jedná o nultý panel na x-ové a nultý panel na y-ové souřadnici. Pro příklad panel tři pozice napravo od něho má souřadnice [3, 0]. Absolutní pozicí myslíme klasické souřadnice používané pro GUI prvky v jazyce Java.

Samozřejmě je nutné diagram a prvky na něm vykreslit. K tomu slouží metoda `paintComponent`, která vykresluje čtverečky, na které je možné vložit panel. Metoda `paint` slouží pro vykreslení všeho ostatního, tj. panelů, připojení a modrých okrajů při akci s panely. Vykreslení okrajů provádí metoda `drawScrollEdges`.

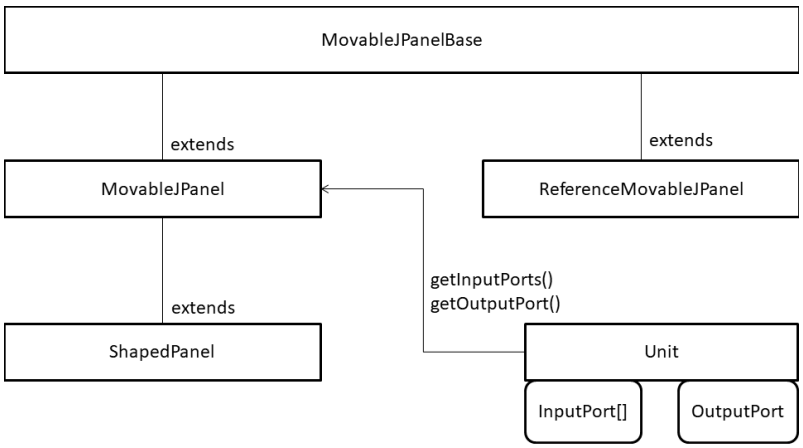
Ještě zmíníme pár metod použitých při připojování a pohybu panelů.

`recalculateAllCables` je volána při přidání nového panelu, či posunutí nějakého existujícího a způsobí vyhledání nových cest pro všechna připojení. O nastavení relativních cest se stará metoda `setRelativeCableConnections`. Relativní cesty jsou nezávislé na přiblížení a pohybu myši po diagramu. Slouží k hledání kolizí cest a vypočítání absolutních cest, které vykreslujeme.

Metoda `findElevation` slouží k vyhledání vyvýšení cesty na základě sdílení částí cest mezi panely.

Pomocná metoda `getStaticPanelLocation` vrátí pro dané absolutní x a y pozici levého horního rohu čtverce, nebo „`null`“, pokud poskytnuté souřadnice x a y ukazují mimo čtverec.

Nyní se můžeme konečně přesunout na popis třídy `MovableJPanel`, která je základní třídou pro panely v diagramu. Pro uvedení do kontextu se můžeme podívat na diagram 4.23, který jsme už mohli vidět dříve.



Obrázek 4.23: Diagram popisující dědičnost okolo třídy MovableJPanel

Nejprve se podíváme na metody související s pohybem panelu. Při zahájení pohybu je na panelu zavolána metoda `startDragging` a při skončení metoda `stoppedDragging`. Metoda `isValidPanelPlacement` kontroluje, jestli je umístění panelu validní. To zahrnuje zjištění, zda-li na dané pozici již není jiný panel, což provedeme metodou `checkForCollisions` na třídě `DiagramPanel`. Ještě musíme

ověřit korektnost připojení panelu. To zahrnuje dvě kontroly. Všechny panely, které se na něho připojují, jsou nad ním a všechny panely, ke kterým z něho vede připojení, jsou pod ním.

Třída `MovableJPanelMouseAdapter` funguje jako mouse listener a navíc v pravidelných intervalech volá callback metody na odpovídajících panelech. Právě v rámci callback metod používáme výše zmíněné metody na detekci kolizí a podle získaných výsledků měníme barvu panelu a tím vytváříme efekt blikání. Callback metody jsou `panelMovementTimerEventCallback`, `connectorInputTimerEventCallback` a `connectorOutputTimerEventCallback`. Z názvů je jasné, co dělají a na jakém panelu jsou volané.

Panel potřebuje několik metod pro správné vykreslení jak sebe, tak i svých připojení. O některých se nyní zmíníme.

Metody se jménem `getNextToLastPoint` jsou použity pro nalezení předposledního bodu připojení, tj. posledního bodu, který neleží na panelu. Metoda `setInputPortLoc` slouží pro nastavení absolutních souřadnic vstupních portů.

Dále ve třídě nalezneme metody pro vykreslování labelu daného portu, které zde nebudeme zmiňovat, protože se jedná o implementační detail.

Metody `getInternals`, `getDistanceFromRectangleBorders`, `reshape`, `getIsPointInsideShape` jsou zajímavé až pro potomky třídy `ShapedPanel`.

Nyní se podíváme na třídu `ShapedPanel`, potomek třídy `MovableJPanel`. Liší se od svého předka především následujícím kódem.

```
protected Shape panelShape;
public Shape getPanelShape() {
        return panelShape;
}
private ShapedPanelInternals internals;
@Override
public ShapedPanelInternals getInternals() {
        return internals;
}
private void setInternals(ShapedPanelInternals internals) {
        this.internals = internals;
}

/**
* Called when resized. Call method reshapeInternals inside.
*/
@Override
abstract public void reshape(Dimension newSize);
```

Pro správný chod třídy je nutné v potomcích nastavit proměnné „*internals*“ a „*shapedPanel*“ a přepsat metody z rozhraní `ShapeSpecificGetMethodsIFace`. Speciálně se jedná o metody `getDistanceFromRectangleBorders`, `getLastPoint` a `getNextToLastPoint`. Občas se může hodit přepsat metodu `paintComponent`, například pokud chceme vykreslovat i hrany. Ovšem nejdůležitější metodou je metoda `reshape`, která je volána při změně velikosti, tedy v této metodě musíme nastavit proměnné „*shapedPanel*“ a „*internals*“ na správné velikosti. Pro nastavení „*internals*“ stačí zavolat metodu `reshapeInternals`.

Rozhraní `ShapedPanelInternals` vypadá následovně.

```
public interface ShapedPanelInternals {
        void reshape(Dimension newSize);
        void draw(Graphics g);

        /**
        * Should create exact copy of class and return it
        * (the copy should not have any shared variables)
        */
        ShapedPanelInternals createCopy();
}
```

Pro příklady implementací se můžeme podívat na `ConstantTextInternals` a `CircleShapedPanel`, které jsou asi nejsnazší na pochopení.

Nyní se přesuneme na porty. Porty nejsou nic jiného než koncové body připojení panelů v diagramu. Slouží tedy k reprezentaci připojení. Rozdělujeme je na vstupní a výstupní.

Nejprve zmíníme metodu, díky které můžeme zvolit port v dialogu, například při připojování. Jedná se o statickou metodu `choosePort` na třídě `PortChooser`. Vytvoří jednoduchý dialog se všemi vstupními porty a povolí volbu těch, co vyhovují filtru. Filtr je předaný jako parametr. Uživatelem zvolený port je vrácen jako návratová hodnota.

```
public static InputPort choosePort(InputPortsGetterIFace portsGetter,
                                   PortFilterIFace filter)
```

Třída `Port` má kromě základních abstraktních metod na vytvoření a odstranění připojení a metody copy i metody z rozhraní `UnitGeneratedValuesInfo`. Třídy `InputPort` a `OutputPort` pak tyto metody implementují.

Rozhraní `UnitGeneratedValuesInfo` obsahuje metody, které používá syntezátor. Pokud je zavoláme na třídě `InputPort`, vrátí odpovídající hodnoty ze třídy `OutputPort`, která je k ní připojená. Pokud je zavoláme na třídě `OutputPort`, vrátí generované hodnoty pro danou výpočetní jednotku. Jednoduše řečeno třída `OutputPort` dává svoje hodnoty na výstup a `InputPort` si je z něho bere.

Ještě nám zbývá popsat návrh v rámci syntézy a máme vše důležité v rámci syntezátoru popsané. Zásadní metodou je `performOneStep`, která provede jeden krok syntézy, což odpovídá jednomu zavolání metody `calculateSamples` na všech výpočetních jednotkách z diagramu. Tuto metodu voláme opakovaně uvnitř metody `run`, což je `run` metoda vlákna pro syntézu. Uspání vlákna zařídíme metodou `pause`, která je zavolána, například když přepneme okno, či přímo zastavíme přehrávání. Výpočetní jednotka `OutputUnit` zásobuje frontu audio vlákna samply v rámci metody `calculateSamples`. Audio vlákno je třída se jménem `AudioThread`. Hlavní metodou je metoda `playAudioLoop`, která se volá v rámci metody `run`. Toto vlákno jsme popsali v sekci 4.4.4.

Všechny výpočetní jednotky diagramu dědí od třídy `Unit`, která na základě vstupních portů uložených v poli „*inputPorts*“ generuje uvnitř metody `calculateSamples` hodnoty. Uložení portů v poli má několik výhod při implementaci, a navíc to uživateli ušetří trochu práce při psaní pluginů. Generované

hodnoty se ukládají do pole uvnitř výstupního portu, ze kterého si je berou připojené panely. Dále si u sebe třída `Unit` pamatuje svůj `ShapedPanel`, který bude zobrazen v diagramu. Více výpočetní jednotky prozkoumáme v rámci uživatelské dokumentace 5.8.3, když se budeme bavit o zásuvných modulech.

Třídy `Unit`, `OutputPort` a `DiagramPanel` implementují rozhraní `SerializeIFace` s metodami `save` a `load`, které používáme při ukládání diagramu na perzistentní úložiště.

4.2.4 Pomocné metody a třídy

Zde se nejprve podíváme na nejdůležitější třídy a jejich metody pro načítání, převod a práci se zvukovými daty. Metody pro převod najdeme ve třídě `util.audio.AudioConverter`. Pomocné metody pro práci se zvukem najdeme ve třídě `util.audio.AudioUtilities`. V závěru sekce zmíníme další v práci často používané metody a třídy, které nesouvisí se zvukem.

Metody pro načítání zvukových dat: `convertStreamToByteArray` a `separateChannelsDouble`. `convertStreamToByteArray` načte vstupní zvuková data ze streamu do bytového pole. `separateChannelsDouble` převede vstupní stream bytů do dvoudimenzionálního pole typu double. Velikost první dimenze odpovídá počtu kanálů. Implementaci řešíme voláním obecnější metody `getEveryNthSampleMoreChannelsDouble`, parametr n nastavíme na počet kanálů.

Ukládání zvukových dat do souboru řeší metoda s názvem `saveAudio`. Podporované formáty audio souborů odpovídají těm, které poskytuje Java. Jedná se o formáty AIFF-C, AIFF, AU, SND, WAVE. Výčet pochází z dokumentace třídy `AudioFileFormat.Type` [8]. Ovšem je nutné poznamenat, že některé formáty nemusí být na daném systém přístupné, například na mém systému Java neposkytuje formáty .aifc a .snd. Nepodporované formáty uživateli nenabízíme. V práci občas ukládáme na perzistentní úložiště i pole s prvky typu `double`, o což se stará metoda `createDoubleWaveFile` na třídě `DoubleWave`. V tomto případě používáme vlastní formát a výsledný soubor má příponu „.dwav“. Soubor obsahuje hlavičku definující formát a za ní následují samply typu double v binární podobě. Použití můžeme vidět ve třídách `WaveShaper` a `Wavetable` při serializaci.

Nyní vypíšeme všechny důležité metody pro převádění zvukových dat, bez kterých by program nemohl fungovat. Všechny následující metody mají viditelnost public a jsou statické. Nebudeme zacházet do implementace a ani je popisovat. Jména by měla být dostatečně výstižná.

```
// Metody na převod samplů mezi různými typy
void convertIntToByteArr(byte[] arr, int numberToConvert,
                         int sampleSize, int startIndex,
                         boolean convertToBigEndian);
void convertDoubleToByteArr(double sampleDouble, int sampleSize,
                            int maxAbsoluteValue, boolean isBigEndian,
                            boolean isSigned,
                            int startIndex, byte[] resultArr);

int convertBytesToInt(byte[] bytes, int sampleSize, int mask,
                      int arrIndex, boolean isBigEndian,
```

```
                        boolean isSigned);
int convertDoubleToInt(double sampleDouble,
                        int maxAbsoluteValue, boolean isSigned);

// Pomocné metody pro převody
int getMaxAbsoluteValueSigned(int sampleSizeInBits) {
        return (1 << (sampleSizeInBits - 1)) - 1;
}
int calculateMask(int sampleSize) throws IOException;

// Metody pro převod mezi poli
int normalizeToDoubles(byte[] byteSamples, double[] outputArr,
                        int sampleSize, int sampleSizeInBits,
                        int arrIndex, int outputStartIndex,
                        int outputLen, boolean isBigEndian,
                        boolean isSigned) throws IOException


void convertDoubleArrToByteArr(double[] doubleArr, byte[] byteArr,
                                int doubleStartInd,
                                int byteStartInd, int len,
                                int sampleSize, int maxAbsoluteValue,
                                boolean isBigEndian, boolean isSigned);
```

Pro převod mezi formáty používáme `convertToMono` a `convertSampleRate`. `convertToMono` zavolaná na třídě `ByteWave` změní načtený zvuk ve formě pole bytů spolu s informacemi o formátu tak, aby měla výsledná datová struktura pouze jeden kanál. To zahrnuje převod pole bytů a změnu proměnných souvisejících s počtem kanálů. Pro převod bytového pole voláme statickou metodu stejného jména, které předáme všechny nutné informace. Metoda `convertSampleRate` slouží pro převod mezi vzorkovacími frekvencemi, nakonec je používaná jen varianta s polem doublů, neboť se snadněji ladila a bytová varianta není potřeba, protože v rámci analyzátoru neprovádíme převod vzorkovacích frekvencí. Tato metoda interně volá buď `convertToHigherSampleRate`, nebo `convertToLowerSampleRateByImmediate`. Pro převod do nižší vzorkovací frekvence máme dvě možnosti. První je námi použitá „immediate“ varianta, ve které využíváme interpolace. Druhou možností je převést signál na vzorkovací frekvenci, která je n-krát větší než cílová frekvence a zároveň je větší než frekvence současná. Z výsledku vezmeme každý n-tý sample a máme převedeno. Hodnotu n samozřejmě volíme minimální.

Pro práci se zvukem nám ještě zbývá metoda `performNonRecursiveFilter` a metody pro práci s FFT, které nalezneme ve třídě `util.audio.FFT`.

Statická Metoda `performNonRecursiveFilter` aplikuje nerekurzivní filtr poskytnutými koeficienty na dané vstupní pole a výsledek uloží do výstupního pole. Metoda funguje, i když jsou vstupní a výstupní pole totožné. Tuto metodu najdeme uvnitř třídy `NonRecursiveFilter`. V programu tuto metodu nepoužíváme přímo, ale voláme jí v rámci metody `runLowPassFilter`, která vytvoří pole koeficientů dané délky odpovídající nějakému low-pass filtru.

Připomínáme, že nerekurzivní filtry jsme definovali rovnicí 2.4.

V programu použitý vzorec pro výpočet koeficientů nalezneme v [4, str. 206]. Pro úplnost uvedeme i použitý kód, ovšem bez popisu.

```
// Calculate coefs for filter
for(int k = 0; k < (coefCount - 1) / (double)2; k++) {
        double currCoef = 0;
        for(int i = 1; i < (coefCount - 1) / (double)2; i++) {
                double tmp = (2 * Math.PI * i / coefCount) *
                (k - (coefCount - 1) / (double)2);
                currCoef += Math.abs(coefForCalc[i]) * Math.cos(tmp);
        }
        coef[k] = coefForCalc[0] + 2 * currCoef;
        coef[k] /= coefCount;
}
```

Zbytek koeficientů je převrácený dle prostředního prvku, tj. $coef[k] = coef[N - k - 1]$, kde N odpovídá parametru „*coefCount*“.

Hodnoty pole `coefForCalc` mají do indexu odpovídající cut-off frekvenci hodnotu jedna, pro prostřední index hodnotu 0.5 a pro zbytek indexů hodnotu 0 [4, str. 208].

Platí, že k-tý index v poli koeficientů odpovídá frekvenci

$$k \cdot \frac{f_s}{N} \tag{4.1}$$

N odpovídá parametru „*coefCount*“, f_s je vzorkovací frekvence.

Použití low-pass filtru je nutné při převodu do nižší vzorkovací frekvence, protože musíme ze signálu odstranit frekvence vyšší než je Nyquistova, respektive je aspoň utlumit. Low-pass filtr je i jedna z implementovaných operací v přehrávači.

FFT metody zde popisovat nebude, jen je dobré vědět, že v práci často používáme typ FFT s reálným vstupem. V knihovně tento typ metody najdeme ve třídě `DoubleFFT_1D` pod názvem `realForward(double[] arr)`. Tento typ FFT nám ušetří místo, ale je nutné mít na vědomí, že výsledek takové FFT má zpřeházené indexy, tj. index v poli nemusí odpovídat dané přihrádce. Jak jsou indexy zpřeházené, můžeme vidět v komentáři na začátku třídy `util.audio.FFT`, který najdeme i v dokumentaci knihovny poskytující FFT. Ve třídě `FFTWindow` najdeme metody pro vytváření hahnova a hammingova okna.

Než opustíme část zabývající se zvukem, zmíníme se o datových strukturách `ByteWave` a `DoubleWave`, které uchovávají zvuková data spolu s audio formátem. Najdeme je v balíčku `util.audio.wave`. V prvním případě jsou samply uloženy jako pole bytů, v druhém jako pole doublů. O načtení zvukových dat ze souboru do třídy `ByteWave` se stará metoda `loadSong`. Instance třídy `DoubleWave` se vytváří buď z pole s prvky typu double[] a audio formátu, nebo ze třídy `ByteWave`.

Načítání zásuvných modulů řeší třída `plugin.util.PluginLoader`.

O vytvoření panelu z anotací se stará třída `AnnotationPanel`. Jakým způsobem se tato funkcionalita používá rozebereme uvnitř uživatelské dokumentaci v sekci o zásuvných modulech pro přehrávač 5.8.2.

Dále zmíníme metody pro práci s labely, které jsou používané tak často, že si aspoň zaslouží zmínku: `drawStringWithSpace`, `getBiggestFontToFitSize`, `findMaxFontSize`, `getBiggestFontToFitMaxHeight`.

Tyto čtyři metody najdeme spolu s dalšími ve třídě `util.swing.SwingUtils`.

V práci na několika místech pracujeme s časem, proto potřebujeme metody starající se o převod vstupního času na výstupní formát, ve kterém jsou jednotlivé časové míry odděleny znakem „:“. Jedná se o metody `convert*ToTime`, kde místo „*“ je časová míra. Všechny tyto metody interně volají obecnou metodu `convertTimeUniversal`. Metody pro práci s časem najdeme ve třídě `util.Time`.

4.3 Vytvářené soubory a jejich formáty

Zde blíže prozkoumáme formáty generovaných souborů v rámci programu.

4.3.1 Ukládání informací z analýzy

Informace z analýzy jsou ukládány do xml souboru, aby je bylo možné později zobrazit.

Na nejvyšší úrovni XML stromu je tag <songs> a pod ním následují podstromy <song>, ve kterých už jsou jednotlivé analyzované parametry, kde atribut name určuje jméno parametru a tag <value> udává jeho analyzovanou hodnotu. Nyní následuje příklad XML souboru s jedním analyzovaným souborem.

```
<songs>
  <song>
    <info name="Name">
      <value>120BPMMono.wav</value>
    </info>
    <info name="Length">
      <value>01:01</value>
    </info>
    <info name="File format">
      <value>WAVE (.wav)</value>
    </info>
    <info name="Encoding">
      <value>PCM_SIGNED</value>
    </info>
    <info name="Sample Size (In bytes)">
      <value>2</value>
    </info>
    <info name="Sample rate">
      <value>22050</value>
    </info>
    <info name="Number of channels">
      <value>1</value>
    </info>
    <info name="Endianness">
      <value>Little endian</value>
    </info>
    <info name="RMS">
      <value>0,17</value>
    </info>
    <info name="BPM (Barycenter part)">
```

```
      <value>119</value>
    </info>
  </song>
</songs>
```

Pokud se podíváme do XML souboru v textovém editoru, vidíme, že jednotlivé prvky jsou odsazeny podobně jako v příkladu výše. Ovšem Java DOM sloužící pro zpracování XML má s načtením takto odsazeného XML souboru problémy, takže je možné, že i jiné knihovny budou mít podobné problémy. Z tohoto důvodu možná v budoucnu odsazení odstraníme a necháme XML soubor v nezměněné podobě, což v tomto případě znamená, že vše bude na jednom řádku.

4.3.2 Audio formáty

Formáty audio souborů není nutné znát, neboť pro jejich načítání a ukládání existují v jazyce Java metody. Jen je dobré vědět, že audio soubory vždy obsahují hlavičku s metadaty a poté samotná data. Z toho důvodu při ukládání samplů ve formátu double nejprve uložíme metadata, která jsou v tomto případě pouze vzorkovací frekvence a počet kanálů, a za ně vložíme samotná audio data.

4.3.3 Ukládání a načítání diagramů

Ukládání a načítání diagramů je realizováno pomocí textových souborů s příponou .dia. Nyní se podíváme na jejich formát.

Na prvním řádku je číslo verze, díky které můžeme v budoucnu dovolit udělat nové verze formátu, ale přesto podporovat načítání všech starých. Samozřejmě pokud bychom v budoucnu provedli zásadní změny v objektovém návrhu, může být se zpětnou kompatibilitou problém.

Na dalším řádku je počet kanálů ve výstupním formátu. Na následujícím řádku je celkový počet panelů v diagramu, včetně výstupních jednotek. Poté následují informace o jednotlivých panelech. Pokud se jedná o výstupní (output) panel, pak je na dalším řádku řetězec „OUTPUT-UNIT“ a na řádku za ním číslo kanálu, který panel reprezentuje, a poté následují informace, které jsou už i u všech ostatních panelů.

Další řádek obsahuje plnou cestu k třídě reprezentující výpočetní jednotku (Cestou myslíme pack.ages.JménoTřídy). Na následujícím řádku najdeme jméno panelu. Na poslední řádce je pozice panelu v diagramu, tj. lokace [x, y]. Souřadnice jsou myšleny na úrovni panelů, nikoliv pixelů.

Poté, co výše zmíněné informace uložíme pro všechny panely, je nutné zapsat informace o připojeních, o což se postará metoda `save` na výstupním portu.

Nejprve napíšeme ke kolika panelům je připojený, a poté na samostatné řádky vypíšeme, o které porty se jedná. Informace jsou uloženy v následujícím formátu: *index_panelu* mezera *index_portu*, kde mezera je znak mezery, *index_portu* udává index vstupního portu v rámci panelu a *index_panelu* udává index, na kterém se panel nachází v kolekci obsahující všechny panely z diagramu. Vzhledem k tomu, že kolekce je setříděná primárně podle y-ové a sekundárně podle x-ové souřadnice, tak rozmístění panelů v kolekci je pro stejné diagramy totožné mezi různými voláními programu.

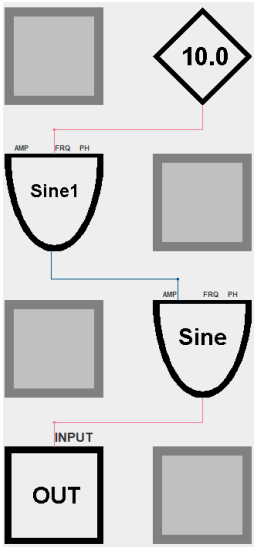
Pro implementační detaily se můžeme podívat do metod `save` a `load` uvnitř třídy `DiagramPanel`.

4.3.4 Příklad souboru s uloženým diagramem

V této sekci se podíváme na příklad serializace. Serializovaný diagram najdeme spolu s textovým souborem reprezentujícím jeho serializaci níže v této sekci.

Jména tříd jsou pro text moc dlouhá, proto jsou rozdělena na dva řádky a stále platí, že vynecháváme prefix str.rad u balíčku, protože ho mají všechny. Navíc jsme do textu přidali komentáře, které jsou za „//“. Tyto komentáře se v souboru reprezentující diagram vyskytovat nesmí. Jsou zde uvedeny pouze pro lepší pochopení.

```
1.0                                     // Verze
1                                       // Počet výstupních panelů
4                                       // Celkový počet jednotek
synthesizer.synth.generators.           // Plné jméno jednotky
classic.ConstantGenerator
CONST                                   // Jméno v rámci diagramu
2 -2                                    // x-ová a y-ová pozice jednotky
10.0                                    // Hodnota const jednotky
synthesizer.synth.generators.
classic.phase.SineGenerator
Sine1
1 -1
synthesizer.synth.generators.
classic.phase.SineGenerator
Sine
2 0
OUTPUT-UNIT                             // Výstupní jednotka
0                                       // Číslo reprezentovaného kanálu
synthesizer.synth.OutputUnit            // Plné jméno výstupní jednotky
OUT                                     // Jméno v rámci diagramu
1 1                                     // Pozice panelu
//
// Nyní následuje popis výstupů jednotlivých panelů.
// Pořadí odpovídá pořadí v tomto souboru,
// tj. nejprve panel CONST, následovaný panely Sine1, Sine a OUT.
// Pozorování: Pořadí v tomto souboru je
// totožné s pořadím panelů v kolekci.
//
1                                     // Počet výstupů CONST generátoru
1 1                                   // 1. panel 1. port
1                                     // Počet výstupů Sine1 generátoru
2 0                                   // 2. panel 0. port
1                                     // Počet výstupů Sine generátoru
3 0                                   // 3. panel 0. port
0                                     // Počet výstupů OUT generátoru
                                      // (Má vždy nula výstupů)
// Poznámka: nultý panel je CONST panel.
```



Obrázek 4.24: Příklad diagramu, jehož serializaci můžeme vidět výše.

4.4 Řešení vybraných problémů

4.4.1 Datová struktura - posuvný buffer

Pro maximálně efektivní prohlížení vln bylo nutné naprogramovat datovou strukturu posuvný buffer, která nám dovolí rychle procházet lokální vykreslované hodnoty.

Idea je jednoduchá. Máme pole, ve kterém se nachází okno a to obsahuje právě viděné, vykreslované hodnoty. Při posunutí vlny pomocí posuvníku okno posuneme. Okno je součástí bufferu, který je typicky větší než je okno samotné, díky tomu stačí pro získání hodnot těsně vedle právě viděných pouze posunout první a poslední index vlny. Pokud je první nebo poslední index mimo platné indexy bufferu, je nutné posunout stávající hodnoty a navíc doplnit buffer novými hodnotami, které se buď načtou z cache souboru na disku obsahujícího vykreslované hodnoty pro současný zoom, nebo se spočítají ze samplů algoritmem ze sekce popisující zobrazení vlny 3.8.1.

Celková délka bufferu závisí na velikosti viditelného okna. Odpovídá

$$bufferLen = windowSize + (2 * p) * windowSize \tag{4.2}$$

p určuje počet oken napravo od viditelného okna bufferu. Stejný počet oken je i nalevo, proto máme v rovnici $2 \cdot$ p. Extrémní případ nastává pro $p = 0$, pak žádné okolí nemáme, a tedy posunutí okna znamená načtení nových hodnot do bufferu, jejichž počet odpovídá buď délce posunutí, nebo velikosti okna, pokud je posunutí větší než je velikost okna.

Při implementaci si musíme uvědomit následující věc. Při vykreslování individuálních hodnot odpovídá jedna hodnota v bufferu jednomu určitému pixelu. Při vykreslování agregovaných hodnot odpovídají dvě hodnoty jednomu pixelu, neboť máme minimum a maximum.

V rámci implementace schováme, jak buffer pracuje interně a místo toho voláme metody posunu a resize na wrapper třídě. To jsme už ovšem do hloubky popsali v rámci objektového návrhu 4.2.

4.4.2 Timestamps (časové značky)

Pro přehrávač stylu Audacity je nutné mít časové značky určující, jaký časový úsek vlny právě zobrazujeme.

Pro správné vykreslení časových značek je nutné vypočítat několik parametrů. Prvním je počet časových štítků viditelných v jeden okamžik, dalším kolik času uběhne mezi těmito časovými značkami a pak samozřejmě kolik pixelů je mezi nimi.

Nejprve si v algoritmu nastavíme počáteční hodnotu celkového počtu labelů, která odpovídá situaci, kdy bereme čtyři labely na jedno viditelné okno.

```
int labelCount;
try {
        labelCount = waveWidth / visibleWaveWidth;
        labelCount *= 4;
} catch (ArithmeticException e) {
        return;
}
```

Poté spočítáme časový skok mezi labely, což na úrovni kódu znamená určit hodnotu proměnné timeJumpInt.

```
double timeJump = numOfSecs / labelCount;
int timestampsMultiples = 5;
int timeJumpInt;
timeJumpInt = (int) timeJump;
if (timeJumpInt >= timestampsMultiples) {
        // We will round the time jump to
        // the closest multiple of timestampsMultiples
        // (for example if timestampsMultiples == 5
        // and timeJumpInt = 7 then timeJumpInt = 10)
        if (timeJumpInt % timestampsMultiples != 0) {
                timeJumpInt += timestampsMultiples -
                                (timeJumpInt % timestampsMultiples);
        }
}
else {
        // Need to calculate time in milliseconds
        if (timeJumpInt < 1) {
                isTimeInSecs = false;
                timeJumpInt = (int) (1000 * timeJump);
                // Convert seconds to milliseconds
                numOfSecs *= 1000;
                // If it is too small then keep lowering the labelCount
                while(timeJumpInt == 0) {
                        labelCount /= 2;
                        // numOfSecs are already milliseconds
                        timeJump = numOfSecs / labelCount;
                        timeJumpInt = (int) timeJump;
                }
        }
}
```

Labely musí mít mezi sebou při vykreslování dostatečně místa. Splnění této podmínky ilustruje následující kód.

```
double pixelJump = 0;
int minimumPixelJump = 20;
int index = 1;
int oldTimeJumpInt = timeJumpInt;
// Now check if it fits the window,
// if it doesn't then take only every index-th label
for (; index <= labelCount; index++) {
        timeJumpInt = index * oldTimeJumpInt;
        // If the song is short enough, then numOfSecs = numOfMillisecs
        pixelJump = timeJumpInt / numOfSecs;
        pixelJump *= waveWidth;
        pixelJump /= MARKS_PER_TIMESTAMP;
        if (pixelJump >= minimumPixelJump) {
                break;
        }
}
```

Dále následuje ještě část s vykreslováním, ale na té už není nic zajímavého, proto zde není přiložen odpovídající kód. Pouze se pohybujeme po „*pixelJump*“ pixelech a vykreslujeme čárky a ke každé „*MARKS_PER_TIMESTAMP*“-té čárce připíšeme časovou značku.

4.4.3 Fronta pro producenta a spotřebitele

V rámci syntezátoru bylo nutné vyřešit problém generování a přehrávání samplů. Vzhledem k tomu, že generování samplů z diagramu může být velice výpočetně náročné, tak jsme generování a přehrávání implementovali v samostatných vláknech a propojili je frontou, což odpovídá problému producenta a spotřebitele. Fronta mezi vlákny je mnohem větší, než kolik si z ní v jeden okamžik bere audio vlákno, neboť generování samplů bude typicky trvat mnohem déle než jejich samotné zpracování v audio vlákně. Audio vlákno totiž pouze vybere data z fronty a vloží je do `SourceDataLine`, případně ještě přehrávaná data předá další třídě, která je zobrazí. Z toho důvodu začneme s přehráváním, až když je ve frontě aspoň nějaký základní počet samplů.

Problém producenta a spotřebitele (producer-consumer problem) je situace, kdy jedno vlákno produkuje data a druhé vlákno tato data přijímá. V našem případě je producentem vlákno, ve kterém se provádí výpočty nad diagramem, a spotřebitelem je audio vlákno, které se stará o přehrávání. Typicky producent vygenerovaná data ukládá do fronty, ze které si je pak spotřebitel vezme, což je právě i náš případ.

Ke vkládání a vybírání dat z fronty dochází poměrně často, a proto jsme zvolili implementaci bez zámků. Implementace pomocí zámků by měla nepříznivý vliv na výkon aplikace, což by mohlo v extrémních případech vyústit až k výpadkům v přehrávání.

Implementace bez zámků je možná, pokud producent data do fronty pouze přidává a spotřebitel je naopak pouze odebírá, což pro řešený problém platí.

Nyní se blíže podíváme na implementaci fronty, kterou můžeme najít ve třídě `CyclicQueueDouble`.

Terminologii budeme používat klasickou, tj. `push` znamená přidání do fronty a `pop` odebrání z fronty.

Jak jsme mohli vyvodit z názvu, fronta je cyklická. To znamená, že jakmile zavoláme `push` a jsme na konci pole reprezentujícího frontu, tak přidáme prvek (resp. prvky) na začátek pole. Ovšem učiníme tak pouze v případě, kdy tím nepřepíšeme z fronty dosud nevybrané prvky. Z toho plyne, že přístup na indexy budeme provádět pomocí operace modulo, proto dává smysl, aby byla délka pole reprezentujícího frontu mocninou dvojky, pak je totiž operace modulo velice levná a odpovídá

```
index & (QUEUE_LEN - 1)
```

Indexy počítáme následujícím způsobem. Budeme mít dvě proměnné typu int: „*popIndex*“ a „*pushIndex*“. S každou operací `pop`, respektive `push` zvětšíme odpovídající proměnnou a na skutečné indexy v poli přistupujeme pomocí modula. Pokud navíc deklarujeme proměnné „*popIndex*“ a „*pushIndex*“ jako volatile, můžeme snadno zjistit počet validních prvků ve frontě v jakémkoliv okamžiku. Modifikátor volatile zajistí, že změny hodnot proměnné jsou vidět ve všech vláknech. Int je v jazyce Java atomický typ, takže se nemusíme bát, že vypočtená hodnota nebude korektní. Nabízí se otázka, jestli neustálým přidáváním k push a pop indexu nedojde k přetečení. To by mělo za následek počítání modula ze záporných čísel, a tedy vyvolání výjimky při přístupu na záporný index pole. Odpověď zní ano. Ovšem při vzorkovací frekvenci o hodnotě 44100Hz se tak stane až za třináct hodin. Navíc při zastavení přehrávání pomocí tlačítka pause vynulujeme pop a push indexy.

Pro zvýšení rychlosti pracují `push` a `pop` metody s poli, ale vždy pracujeme maximálně s tolika prvky, kolik jich je v daném okamžiku dostupných ve frontě pro danou operaci. Metody `push` a `pop` vrací počet přidaných, resp. odebraných prvků. Tyto hodnoty jsou spočteny metodami `getPushLength` a `getPopLength`. Nyní následuje kód se všemi důležitými metodami implementovanými v rámci fronty. Kód je ponechán bez komentáře, neboť je poměrně jednoduchý a samo dokumentující.

```
public int getLen() {
        return pushIndex - popIndex;
}
```

```
public int getPushLength(int startIndex, int endIndex) {
        int availableLen = queue.length - getLen();
        int len = Math.min(endIndex - startIndex, availableLen);
        return len;
}
```

```
public int push(double[] values, int startIndex, int endIndex) {
        int len = getPushLength(startIndex, endIndex);
        for(int index = 0; index < len; index++,
            startIndex++, pushIndex++) {
```

```
                queue[getPushIndexMod()] = values[startIndex];
        }

        return len;
}
```

```
public int pop(double[] values, int startIndex, int endIndex) {
        int len = getPopLength(startIndex, endIndex);
        for(int index = 0; index < len; index++,
            startIndex++, popIndex++) {
                values[startIndex] = queue[getPopIndexMod()];
        }

        return len;
}
```

```
public int getPopLength(int startIndex, int endIndex) {
        return Math.min(endIndex - startIndex, getLen());
}
```

```
public int pop(int n) {
        int len = getPopLength(0, n);
        popIndex += len;
        return len;
}
```

4.4.4 Bližší pohled na audio vlákno syntezátoru

V rámci práce dává smysl mít vlákno starající se o přehrávání zvuku zvlášť, protože běží prakticky pořád, tím pádem není vhodné, aby se přelo o výkon s vláknem pro grafické uživatelské rozhraní.

Audio vlákno běží v nekonečné smyčce a funguje následujícím způsobem.

Vždy na začátku cyklu vlákno zkontroluje, jestli se má zastavit. Pokud k zastavení nedošlo, následuje kontrola, jestli je ve všech frontách aspoň minimální počet samplů pro přehrání, pokud ne, skočíme na začátek smyčky. Pokud ano, tak na základě zbývajícího místa v `SourceDataLine` se rozhodne, kolik samplů se z front přehraje. Mluvíme o frontách, neboť každý kanál má vlastní frontu. Pokud je syntetizovaná vlna vykreslovaná, předají se právě zpracovávané samply typu double třídě starající se o její vykreslovaní. Uvnitř této třídy se přehrávané samply uloží do front. Poté už konečně převedeme double samply do intových samplů výstupního audio formátu a vložíme je v podobě série bytů do `SourceDataLine`, která je přehraje.

4.4.5 Přehrávání a mixování v rámci přehrávače

Přehrávání běží v samostatném audio vlákně. Má dvě hlavní části: mixování a přehrání výsledků mixování. Na rozdíl od syntezátoru řešíme věci ohledně zvuku pouze na jednom samostatném audio vlákně. Kdybychom problém zkoušeli řešit jako producer-consumer a to tak, že producentem by bylo mixující vlákno a

konzumentem audio vlákno, muselo by mixující vlákno poskytnout vláknu přehrávajícímu zvuk informaci o změně přehrávané pozice uživatelem, jinak by došlo k nesprávnému zobrazení přehrávané pozice. Ideálně by se to řešilo přerušením a vyprázdněním fronty, jinak by mohlo dojít k posunu v čase, a tím zase nesprávnému zobrazení přehrávané pozice. Takové řešení je ovšem poměrně těžkopádné. Navíc vzhledem k tomu, že počet mixovaných stop se pohybuje maximálně v řádu desítek, je jedno vlákno více než dostatečně výkonné k tomu, aby se staralo zároveň o mixování a přehrávaní. Přehráváním v této sekci je myšlen převod samplů ve formátu double do výstupního audio formátu a následné naplnění bufferu `SourceDataLine` sérií bytů odpovídající převedeným samplům.

Mixování odpovídá sečtení všech vln vynásobených hodnotami z intervalu [0,1] danými posuvníky na levé straně vln. Posuvníků je tolik, kolik je kanálů ve výstupním formátu. Aby nedošlo k přetečení, tak výsledky pro jednotlivé kanály vydělíme počtem mixovaných vln, respektive pro lepší výsledky se vyplatí vydělit maximální možnou mixovanou hodnotou, která odpovídá součtu multiplikativních faktorů z posuvníků pro daný kanál.

4.4.6 Celkový návrh výpočetního modelu pro diagram

Nyní se podíváme, jak producent vyrábí samply pro audio vlákno. Výpočetní jednotky máme uloženy v kolekci a jsou uspořádány ve směru výpočtu. To znamená, že pro provedení jednoho kroku výpočtu nám stačí kolekci projít od začátku do konce a zavolat na každém prvku metodu pro generování samplů. Výstupní panely v rámci zavolání této metody uloží svůj vstup do fronty, respektive se nejdříve podívají, jestli ho není nutné znormalizovat. Pokud ve frontě není dost místa, počkají, než audio vlákno uvolní místo ve frontě zpracováním dostatečně mnoha samplů. K deadlocku dojít nemůže, neboť pokud dojde k zastavení přehrávání uživatelem, nejdříve zastavíme generující vlákno a až poté audio vlákno. Platí, že generující vlákno se zastaví až poté, co všechny výstupní panely vloží svoje hodnoty do fronty. Mluvíme o výstupních panelech, neboť jeden výstupní panel odpovídá jednomu zvukovému kanálu. Tedy pokud máme formát mono, pak je jen jeden takový panel, pokud stereo tak dva.

Musíme si uvědomit, že při výpočtu diagramu může uživatel nějaké panely odebrat, či změnit jejich pozice. V takovém případě musíme přepočítat výsledek na základě nové situace.

Dále je dobré si uvědomit, že mohou v rámci jednoho cyklu vzniknout inkonzistence. Například pokud odebereme panel, jehož výstup posíláme na vstup dvěma jiným panelům, může se stát, že jeden panel tyto hodnoty dostane a druhý dostane hodnoty výchozí. Totéž platí při změně přípojení. Dá se tedy říct, že inkonzistence odpovídá stavu, kdy výpočetní jednotky v rámci jednoho kroku počítají nad různými diagramy. Ovšem všechny zatím implementované výpočetní jednotky jsou závislé pouze na současných vstupech, případně na nějaké krátké historii příchozích vstupů, tedy inkonzistence ovlivní jen několik málo vypočítaných samplů. Problém nastane, pokud přidáme komponentu, pro kterou inkonzistence ovlivní netriviální množství budoucích samplů. Příkladem takové komponenty je rekurzivní filtr, protože k vypočítání současného samplu se použijí filtrem dříve vypočítané hodnoty, tedy inkonzistence trvá, dokud nezastavíme přehrávání. Rekurzivní filtry jsme definovali rovnicí 2.5.

Toto je inherentní vlastnost rekurzivních filtrů a takové chování dobře odráží jejich funkcionalitu. Pro výpočetní jednotky s inkonzistencí trvající delší dobu jsme přidali do třídy `Unit` abstraktní metodu `copyInternalState(Unit u)`. Instance, na které je metoda zavolána, má za úkol zkopírovat vnitřní stav výpočetní jednotky předané v parametru do sebe. Tuto metodu využijeme v budoucnu pro vyřešení výše zmíněných inkonzistencí.

Nyní se podíváme na současný způsob generování.

```
while(true) {
        try {
                for (Unit unit : units) {
                        if(!unit.getPerformedCalculation()) {
                                unit.calculateSamples();
                                unit.markAsCalculated();
                        }
                }

                if (units.getHasChanged()) {
                        units.setHasChanged(false);
                }
                else {
                        break;
                }
        }
        catch(ConcurrentModificationException |
              NullPointerException e) {
                units.setHasChanged(false);
        }
}

int writtenSamplesCount;
writtenSamplesCount = outputUnitGetter.getOutputUnitWrittenSamples();
timeInSamples += writtenSamplesCount;

// Dále v kódu vynulujeme proměnnou performedCalculation na všech
// jednotkách.
// Učiníme tak zavoláním unmarkAsCalculated na jednotlivých jednotkách.
```

Jeden krok výpočtu, tj. jedno zavolání metody `calculateSamples`, vygeneruje BUFFER_LEN = 512 samplů. Samotný výpočet `calculateSamples` je jednoduchý. Výpočetní Jednotka si vezme hodnoty na svých vstupech a podle toho, o jakou výpočetní jednotku se jedná, s nimi provede odpovídající operaci. U operátorů pouze vezmeme vstupy a provedeme operaci. U generátoru je navíc v metodě `calculateSamples` nutné rozlišit, jestli se jedná o frekvenční modulaci, či nikoliv, viz třída `Generator`.

4.4.7 Nahrávání

V zadání práce bylo zmíněno, že v části obsahující syntezátor bude možné nahrát generovaný zvuk. Program umožňuje jak nahrání vlny do souboru, tak i přímo do přehrávače.

Práce podporuje dva různé způsoby nahrávání.

První, takzvané instantní nahrávání, vygeneruje množství samplů odpovídající zadané délce okamžitě do pole a to přesune do přehrávače nebo do souboru, případně do obou.

Druhým typem nahrávání je real-time nahrávání, které funguje následujícím způsobem. Těsně před tím než zapíšeme samply do `SourceDataLine`, která přehrává zvuk, je uložíme do pole odpovídající nahrávce. Nahrávka se ukončí buď po uběhnutí uživatelem nastaveného maximálního času, nebo manuálním stisknutím tlačítka, kterým jsme nahrávání začali. Výhoda tohoto přístupu spočívá v tom, že v nahrávce jsou zaznamenané i uživatelem provedené změny, například změna frekvence generátoru. Pokud je přehrávání pozastavené (tlačítko pause/play je nastavené na pause), real-time nahrávání neběží a bude pokračovat až po stisknutí tlačítka play.

O tom jak nahrávat se více dozvíme později v sekci popisující ovládání syntezátoru 5.7.3.

4.4.8 Wavetable syntéza

Wavetable syntéza je často používaná technika syntézy zvuku. Základní myšlenkou je uložit vlnu reprezentovanou rovnoměrně rozmístěnými samply v čase do wavetable, která není nic jiného než pole. Tuto tabulku bereme jako jednu periodu vlny, což znamená, že za posledním samplem wavetable následuje sample na indexu nula. Pro jednoduché vlny stačí i velice malá velikost tabulky. Například v případě sinové vlny stačí pro dostatečně dobrou reprezentaci tabulka o 512 prvcích.

Samotná syntéza funguje tak, jak bychom čekali, tedy že v závislosti na tom, o jaké frekvenci chceme generovat vlnu, se pohybujeme po indexech v poli. Čím větší frekvenci generujeme, tím dále se v rámci jednoho kroku po wavetable pohybujeme. Skok pro výpočet dalšího indexu při generování vlny o frekvenci f při výstupním sample rate f_s je určen následující rovnicí [4, str. 77].

$$INDEX\_JUMP = TABLE\_LEN * \frac{f}{f_s} \tag{4.3}$$

Například pro $f_s = 8000$, $f = 8$, $TABLE\_LEN = 500$ se pohybujeme o 0.5 indexu po každém generovaném samplu.

Vzhledem k tomu, že málo kdy se pohybujeme po celých indexech, tak je nutné vyřešit situaci, když skončíme mezi dvěma samply. Častým řešením tohoto problému, které používáme i my, je tzv. interpolace. Chová se jako vážený průměr čísel, mezi které hodnota dopadne. Například pokud máme index 5.932, vezmeme vážený průměr mezi hodnotami v indexech 5 a 6. Vzhledem k tomu, že hodnota leží na 93.2% cesty mezi 5 a 6, tak je výsledkem interpolace hodnota $0.932 \cdot wt[6] + 0.068 \cdot wt[5]$ [4, str. 78].

Výhodou wavetable syntézy oproti generování vln jako funkcí v čase je rychlost a navíc není potřeba v rámci kódu speciálně řešit frekvenční modulaci.

Příklad generování jedné periody funkce sinus do wavetable o velikosti 512 prvků:

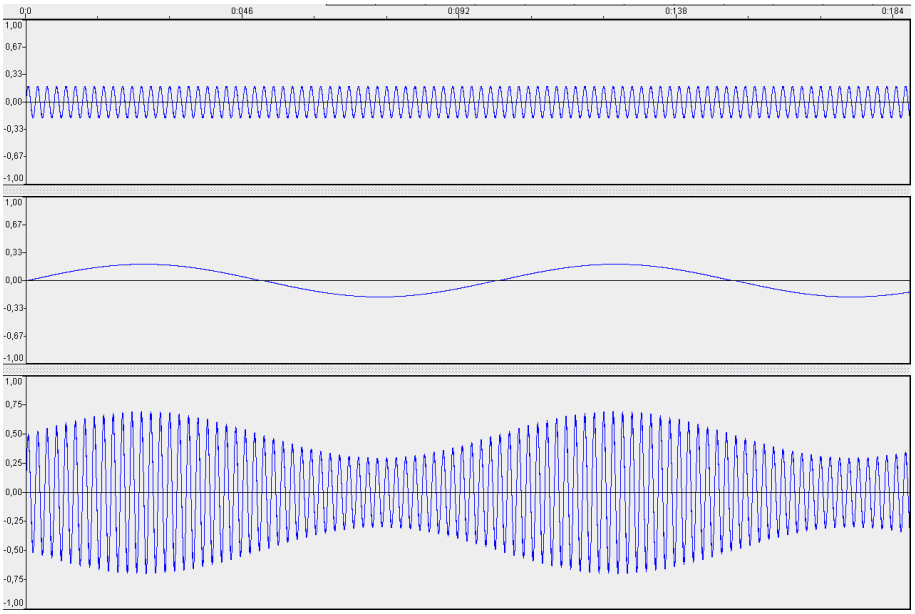
```
double[] sine = new double[512];
for(int i = 0; i < sine.length; i++) {
```

```
        sine[i] = Math.sin(2 * Math.PI * (i / (double)sine.length));
}
```

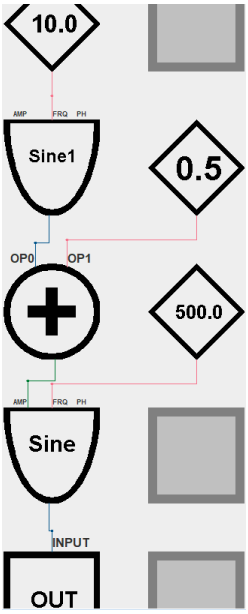
4.4.9 Amplitudová modulace

Definice 8. *(Amplitudová modulace) Amplitudová modulace (amplitude modulation) je situace, kdy je výstup generátoru, respektive posloupnosti operací a generátorů, připojen na binární operátor plus, jehož druhým argumentem je konstanta. Výstup operátoru plus je připojen na vstup kontrolující amplitudu generátoru. Vlnu připojenou na plus nazýváme modulující a generátor, jehož amplitudu modulujeme nazýváme carrier oscilátor. [Více se lze dočíst v 4, str. 90]*

V 4.25 můžeme vidět příklad amplitudové modulace, který je generovaný diagramem 4.26.



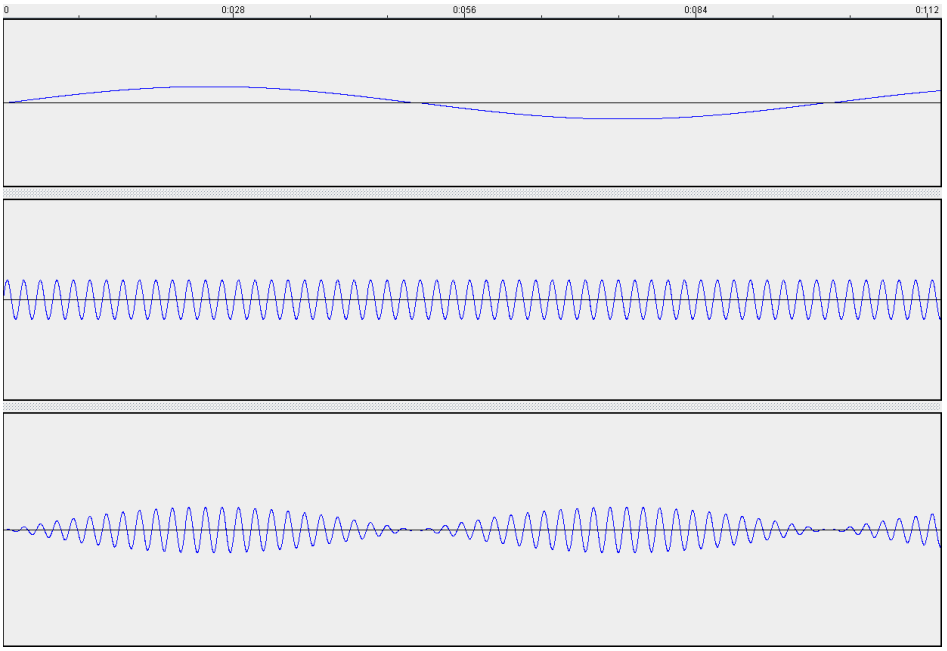
Obrázek 4.25: Amplitudová modulace. Amplituda 500-ti Hz carrier vlny měněna konstantou = 0.5 a 10-ti Hz modulující sinovou vlnou s amplitudou 0.2



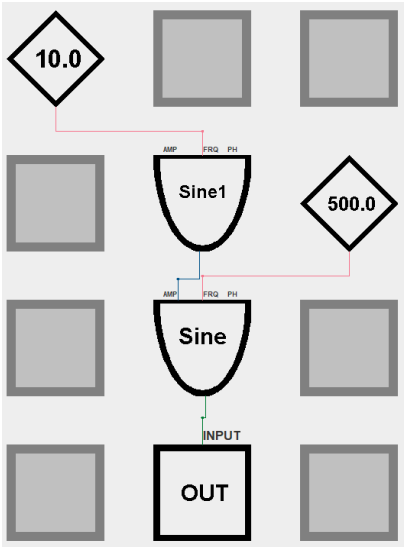
Obrázek 4.26: Diagram pro amplitudovou modulaci, jejíž výsledek lze vidět v 4.25

4.4.10 Ring modulace

Jedná se o speciální případ amplitudové modulace, kdy je konstanta přidaná k amplitudě rovna nule. Vzhledem k tomu, že je konstanta rovna nule, můžeme operátor plus vynechat a připojit generátor rovnou na vstup kontrolující amplitudu [4, str. 92]. V obrázku 4.27 můžeme vidět výsledek ring modulace. Obrázek 4.28 obsahuje diagram použitý pro generování tohoto výsledku.



Obrázek 4.27: Ring modulace, amplituda 500-ti Hz vlny měněna 10-ti Hz sinovou vlnou



Obrázek 4.28: Diagram pro ring modulaci, jejíž výsledek lze vidět v 4.27

4.4.11 Frekvenční modulace

Frekvenční modulace, jak již název napovídá, slouží k modulaci frekvence. Máme carrier oscilátor, jehož příchozí frekvence je výsledek sečtení konstanty, kterou nazýváme carrier frekvence, a výstupu modulující vlny.

Implementace pro generátory, které nepoužívají wavetable syntézu, není až tak jednoduchá, jak se na první pohled může zdát. Nejdříve si musíme uvědomit, že nemůžeme jen vzít příchozí hodnoty na vstupu pro frekvenci a ty použít jako argument do generátoru, tj. kód by byl stejný jako při počítání s konstantní frekvencí. Totiž vlny o různých frekvencích mají v jednom časovém okamžiku rozdílné hodnoty. Například pokud máme 1Hz a 2Hz vlny sinus, pak pro $\frac{f_s}{4}$-tý sample máme u 1Hz vlny hodnotu samplu jedna, zatímco u 2Hz hodnotu nula, viz obrázek: 4.29. To je také důvod, proč nemusíme u wavetable syntézy speciálně řešit frekvenční modulaci, protože tam se v nějakém časovém okamžiku nacházíme na určité pozici vlny a změna frekvence znamená pouze, že se v rámci dalšího kroku přesuneme z dané pozice na jiné místo, takže nenastávají žádné diskontinuity v generovaném signálu.

Správným řešením je využít tzv. fázové modulace, kdy celou dobu generujeme signál o carrier frekvenci, akorát posouváme jeho fázi na základě parametrů a výstupu modulující vlny.

K implementaci využijeme formuli 4.4 z článku od Johna M. Chowinga [5], který je autorem algoritmu frekvenční modulace. Naše testování ukázalo, že první vlnu sinus, tj. carrier vlnu, z rovnice 4.4 můžeme nahradit jakoukoliv vlnou, tedy například čtvercovou, či trojúhelníkovou, či úplně jinou. Ovšem nahrazení modifikující vlny jinou funkcí, než je sinus, nemusí vždy dávat výsledky, jaké jsme čekali. V takovém případě musíme nahradit carrier generátory jejich wavetable alternativami, tj. vložit do tabulek odpovídající vlny. Z toho důvodu by bylo dobré v budoucnu přidat možnost výběru, tj. jestli chceme do diagramu vložit generátor fungující na klasický způsob, nebo použít jeho wavetable variantu. Ovšem jedná

se o vylepšení pro uživatele příjemné, nikoliv nutné.

$$A \cdot sin(CF \cdot T + I \cdot sin(MF \cdot T)) \tag{4.4}$$

A je amplituda vlny, jejíž frekvenci modifikujeme, CF je carrier frekvence, MF je frekvence modulující vlny, T je čas. I je tzv. modulation index, jehož hodnota odpovídá $\frac{d}{MF}$, kde d je peak deviation (největší výchylka) a odpovídá amplitudě modulující vlny.

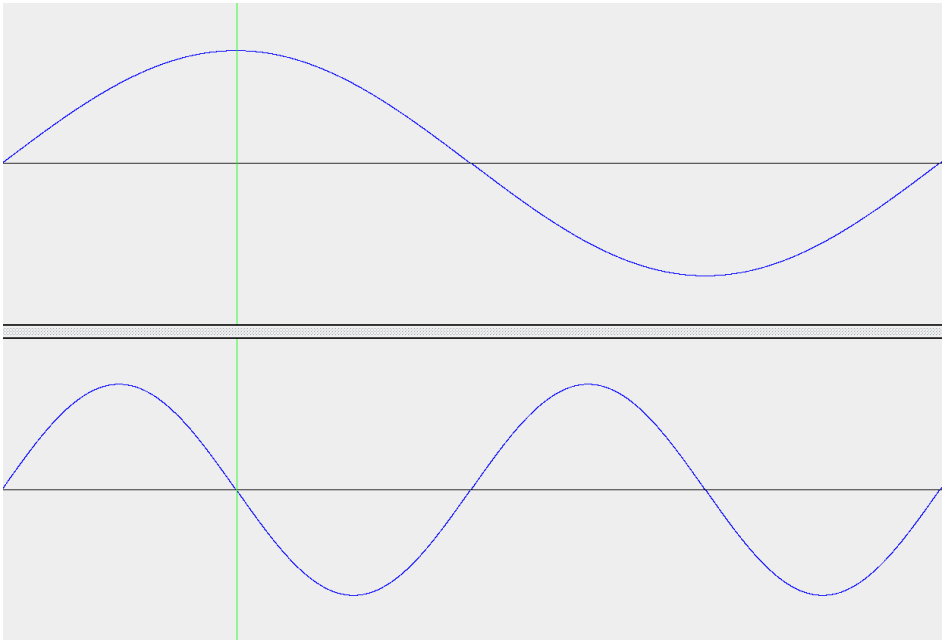
Implementace (metoda `generateSampleFM` uvnitř třídy `Generator`):

```
if(modulatingWaveFreq != 0) {
        phase += modulatingWaveOutValue / modulatingWaveFreq;
}
return generateSampleConst(timeInSecs, diagramFrequency,
                           amp, carrierFreq, phase);
```

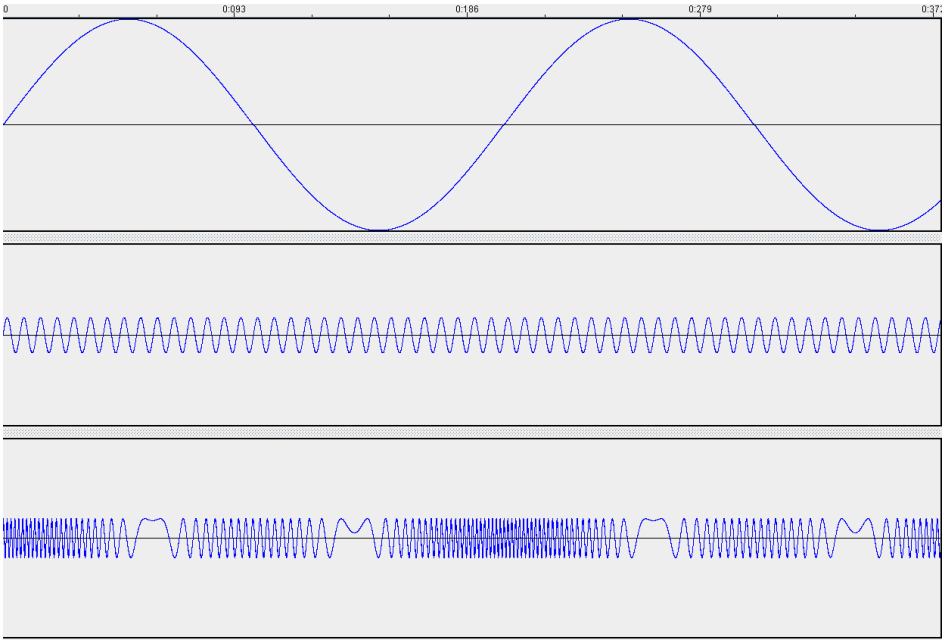
Můžeme si všimnout, že v kódu není explicitně zmíněn modulation index, ale když si uvědomíme, že „*modulatingWaveOutValue*“ je výstupní hodnota modulující vlny a ta už je vynásobena amplitudou, tedy odpovídá 4.5, tak vidíme, že stačí proměnnou „*modulatingWaveOutValue*“ vydělit frekvencí modulující vlny.

$$d \cdot sin(MF \cdot T) \tag{4.5}$$

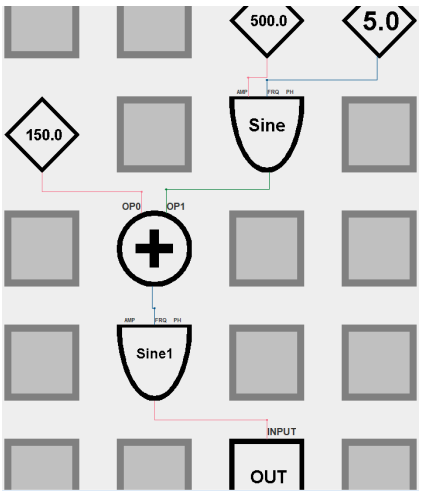
Frekvenční modulaci můžeme vidět v příloze 4.30 a diagram, který jí generuje najdeme v 4.31.



Obrázek 4.29: Obrázek ukazuje dvě sinové vlny o frekvenci jeden a dva hertz, zelená úsečka ukazuje na bod $\frac{f_s}{4}$.



Obrázek 4.30: Frekvenční modulace s parametry: Carrier frekvence = 150 Hz, Frekvence modulující vlny odpovídá hodnotě 5 a amplituda hodnotě 500. První zobrazená vlna, která odpovídá modulující vlně, má hodnoty z intervalu [-1,1] místo intervalu [-500,500].



Obrázek 4.31: Diagram pro frekvenční modulaci, jejíž výsledek lze vidět v 4.30

4.4.12 Generátory v diagramu

Generátory mají na rozdíl od operátorů na vstupu čas, respektive generátor je cokoliv, co je buď závislé na čase, nebo nemá vstupní parametry. V této sekci nebudeme explicitně zmiňovat vstup s časem, neboť ho mají kromě generátoru generující konstantu všechny implementované generátory.

Zatím v práci existují generátory tří typů.

Prvním typem jsou klasické generátory, které mají dva podtypy. První dostane na vstupu amplitudu a frekvenci. Druhý podtyp k těmto vstupům přidá fázi. Fáze je v diagramu zadávána ve stupních, do radiánů se převede až v rámci výpočtu. Typicky dává smysl implementovat jen generátor s fází a generátor bez fáze vyřešit jeho voláním, viz příklady v uživatelské dokumentaci 5.8.3. V práci je pro každý z těchto dvou typů implementováno pět základních generátorů, na které se teď podíváme. Začneme s kódem pro generování funkce sinus.

```
amp * Math.sin(freqToRad(freq) * timeInSecs + phase);
```

Pomocná funkce freqToRad převede frekvenci na úhlovou rychlost a to tak, že jí vynásobí číslem $2 \cdot \pi$.

Oscilátor generující čtvercovou vlnu, pro jehož implementaci využijeme funkci sinus, vypadá následovně:

```
double genVal;
genVal = generateSine(timeInSecs, diagramFrequency,
                      amp, freq, phase);
if (genVal > 0) {
        genVal = amp;
}
else {
        genVal = -amp;
}

return genVal;
```

Pro generování následujících vln je nutné znát funkci Arkus sinus, která odpovídá inverzu funkce sinus. V češtině jí značíme jako sin^{-1}. V kódu je pod označením asin.

Dvě možné implementace pro generování trojúhelníkové vlny. Odpovídají rovnicím ze stránky Wikipedia [16]:

```
double genVal;
// 1 odpovídá amplitudě.
genVal = generateSine(timeInSecs, diagramFrequency, 1, freq, phase);
genVal = amp * 2 / Math.PI * Math.asin(genVal);
return genVal;
```

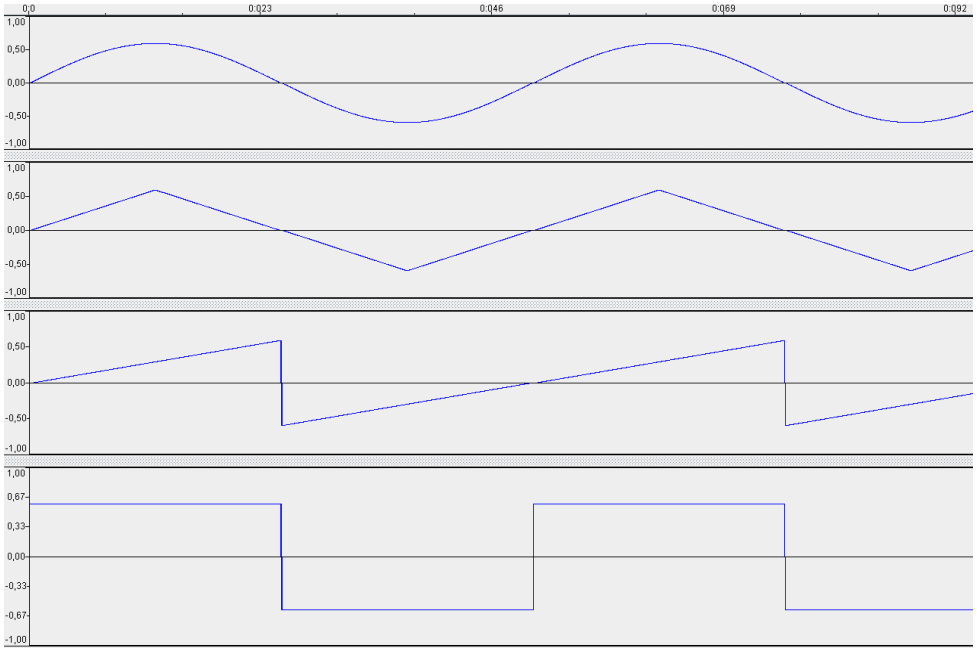
```
double genVal;
double currRad = timeInSecs * freq + phase / (2 * Math.PI);
double floor = Math.floor(currRad + 1 / 2.0);
genVal = amp * 2 * Math.abs(2 * (currRad - floor)) - 1;
return genVal;
```

Pro sawtooth (pilovitou) vlnu máme také dvě různé implementace jejího generování, zase odpovídají rovnicím ze stránky Wikipedia [15]:

```
double genVal;
double currRad = timeInSeconds * freq + phase / (2 * Math.PI);
double floor = Math.floor(currRad + 1 / 2.0);
genVal = amp * (2 * (currRad - floor));
return genVal;
```

```
double genVal;
genVal = -2 * amp / Math.PI *
         Math.atan(calculateCotangent(phase / 2 +
                                      Math.PI * timeInSeconds * freq));
return genVal;
```

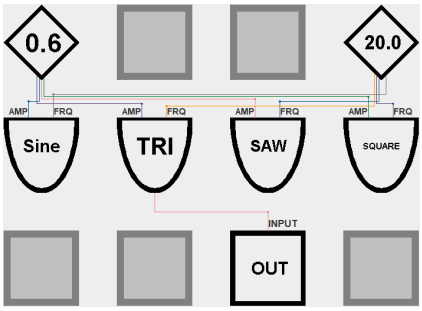
Vlny generované čtyřmi výše popsanými generátory lze najít v příloze 4.32 a jejich generátory v 4.33.



Obrázek 4.32: Můžeme vidět 4 vlny. Všechny o frekvenci 20Hz s amplitudou 0.6. Jedná se v pořadí odshora dolů o vlny sinusovou, trojúhelníkovou, pilovitou a čtvercovou. Anglicky pak sine wave, triangle wave, sawtooth wave a square wave.

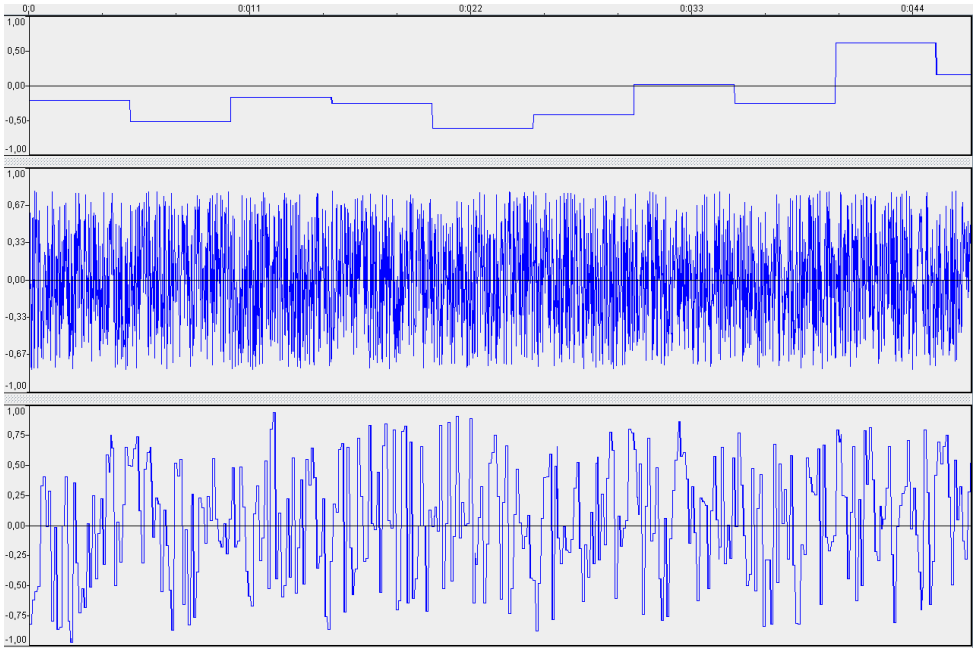
Pátým generátorem je wavetable generátor.

Druhým typem jsou generátory hluku (noise generators). Ty se také rozdělují na dva podtypy. První dostává na vstupu pouze amplitudu a takový generátor generuje novou hodnotu při každém zavolání. Například pokud máme vzorkovací frekvenci $f_s = 44100$, pak generátor hluku vygeneruje za sekundu 44100 různých samplů. Druhý podtyp má navíc vstupní parametr určující frekvenci, která udává, kolik různých samplů má generátor vygenerovat za sekundu. Vzhledem k tomu, že nelze vygenerovat více hodnot za sekundu než je vzorkovací frekvence, tak hodnota dává smysl na intervalu $[1, f_s]$. Po chvíli přemýšlení lze dospět k pozorování,

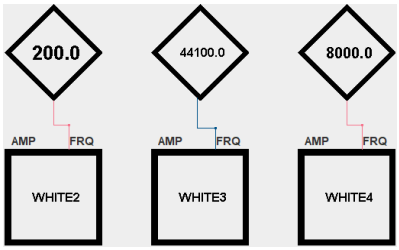


Obrázek 4.33: Prvky diagramu generující vlny z obrázku 4.32. Zleva doprava: sinová, trojúhelníková, sawtooth vlna a čtvercová.

že hodnota $\frac{1}{f}$ udává periodu (budeme značit P) a platí, že hodnota $\mathrm{P} \cdot f_s = \frac{f_s}{f} = n$ nám říká, vygeneruj novou hodnotu každých n samplů. Tuto myšlenku lze najít v [4, str. 97]. Druhým pozorováním je, že generátor bez vstupního parametru frekvence odpovídá generátoru, který má vstupní frekvenci rovnou f_s. V rámci práce je implementovaný pouze white noise generátor, který generuje náhodné hodnoty pomocí libovolného generátoru pseudonáhodných čísel. Pro generování náhodných čísel využijeme metodu `nextDouble` na instanci třídy `Random`, která vrací číslo z intervalu [0,1) a to následně převedeme na číslo z intervalu [-amplituda, amplituda]. Výsledná vlna má při frekvenci rovné f_s stejnou energii ve všech frekvencích, respektive stejně velká frekvenční pásma mají zhruba stejně energie, například pásmo 80-90Hz bude mít přibližně stejnou energii jako pásmo 3080-3090Hz. Příklady výstupů white noise generátorů s různými frekvencemi můžeme vidět v příloze 4.34 a generátory, které je generují v příloze 4.35.



Obrázek 4.34: Můžeme vidět white noise generátory. Všechny mají amplitudu o hodnotě 0.8 a frekvence jsou odshora dolů 200Hz, 44100Hz, 8000Hz. Výstupní vzorkovací frekvence je 44100Hz.

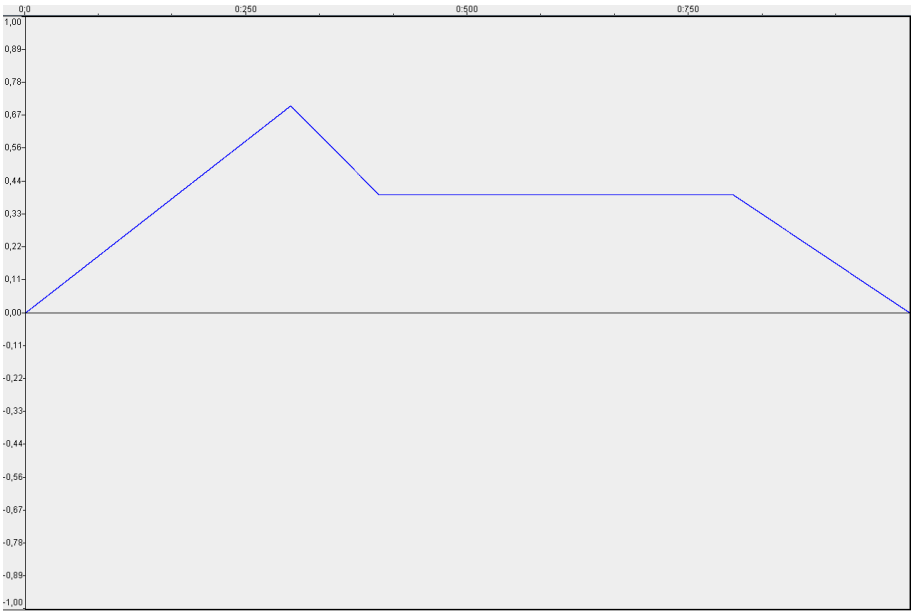


Obrázek 4.35: Prvky diagramu generující jednotlivé vlny z 4.34

Posledním typem jsou envelopes (obálky), také často nazývané ADSR envelopes, kde ADSR znamená Attack, Decay, Sustain, Release. Tyto termíny označují čtyři části obálky. Typicky obálku připojujeme na vstup kontrolující amplitudu generátoru, protože obálky napodobují chování reálných zdrojů zvuků. Nejprve nastane vysoký nárůst energie (attack), poté její snížení (decay) na stabilní hodnotu, která vydrží po dobu hraní (sustain), a když přestaneme, dojde k postupnému snížení energie na nulu (release).

Obálka má šest vstupních parametrů. První udává délku attack fáze, druhý amplitudu, které se na konci attack fáze dosáhne. Třetí udává délku decay fáze, čtvrtý délku sustain fáze, pátý amplitudu sustain fáze a konečně šestý nám určuje dobu trvání release fáze.

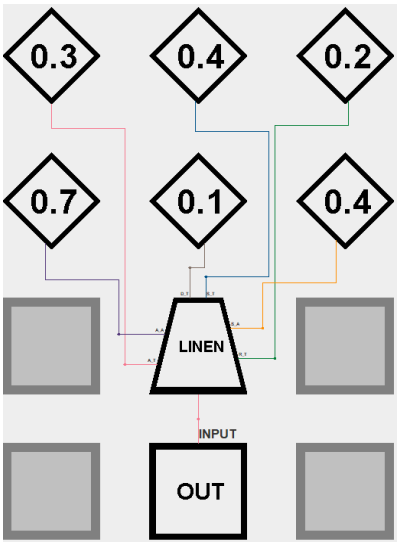
V práci jsou implementovány dva typy obálek. Liší se podle způsobu přechodu mezi fázemi. První typ má přechod lineární a příklad obálky s lineárním přechodem lze spatřit níže v obrázku 4.36 a odpovídající generátor v obrázku 4.37.



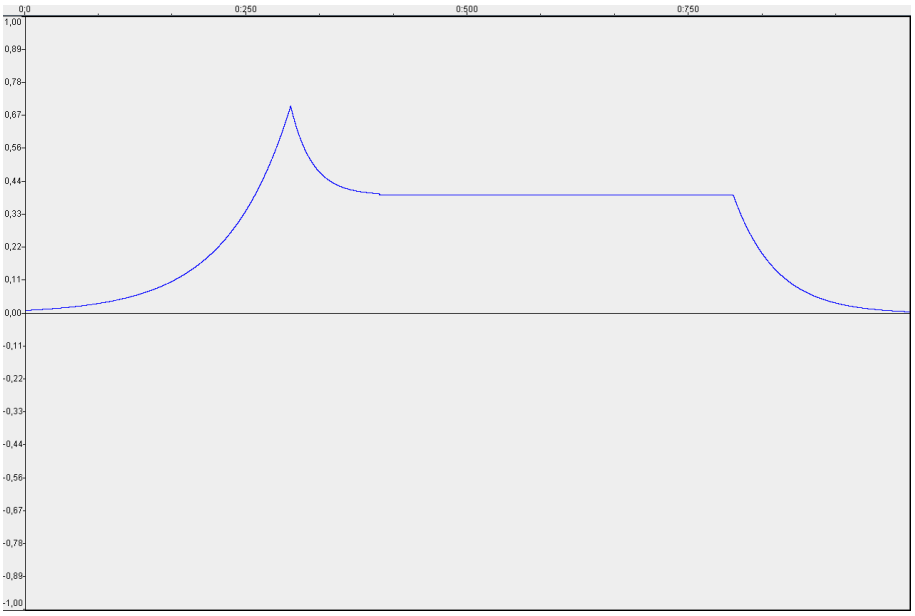
Obrázek 4.36: Envelope (obálka) s lineárními přechody mezi fázemi a následujícími parametry: Attack time = 0.3, Attack amplitude = 0.7, Decay time = 0.1, Sustain time = 0.4, Sustain amplitude = 0.4 a Release time = 0.2

Druhým typem jsou obálky s exponenciálními přechody a příklad obálky tohoto typu lze spatřit níže v obrázku 4.38 a odpovídající generátor v obrázku 4.39.

Obě vlny pod sebou najdeme v obrázku 4.40.



Obrázek 4.37: Diagram generující obálku z obrázku 4.36



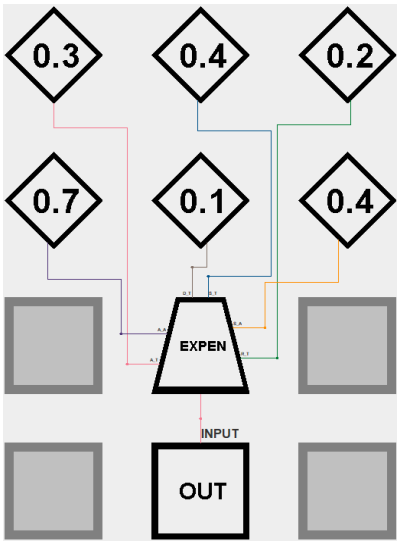
Obrázek 4.38: Envelope (obálka) s exponenciálními přechody mezi fázemi a následujícími parametry: Attack time = 0.3, Attack amplitude = 0.7, Decay time = 0.1, Sustain time = 0.4, Sustain amplitude = 0.4 a Release time = 0.2

4.4.13 Operátory v diagramu

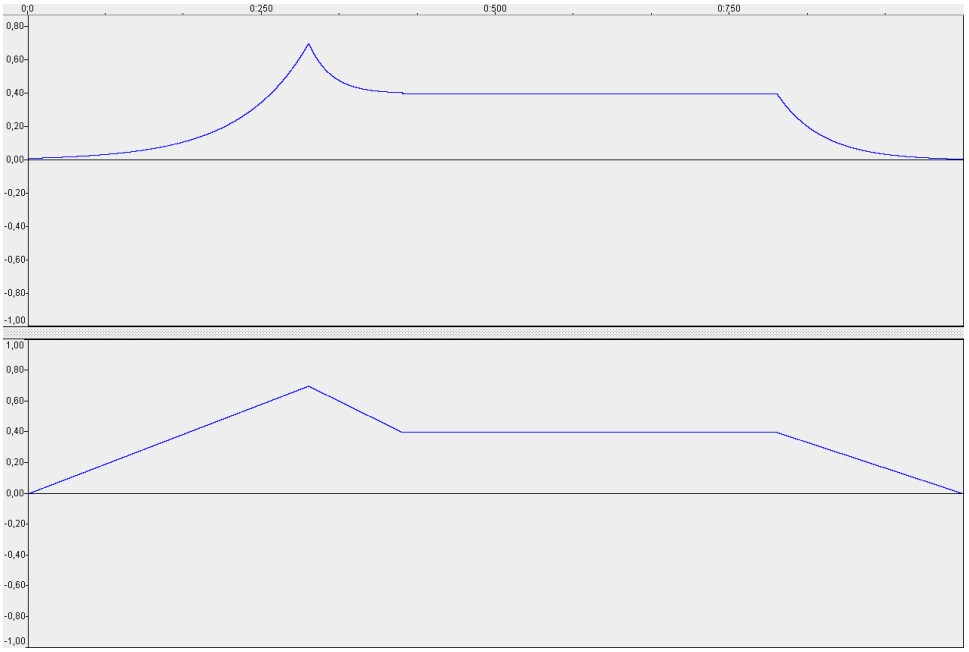
Operátor dostane vstupní hodnoty, které vloží do operace a na výstup dá její výsledek.

V rámci práce implementujeme pouze několik základních operací.

- Binární plus
- Binární minus
- Binární dělení
- Binární násobení



Obrázek 4.39: Diagram generující envelope z 4.38



Obrázek 4.40: 4.38 a 4.36 v jednom obrázku

- Unární minus
- Unární operace převrácení hodnoty (odpovídá $\frac{1}{vstup}$)
- Rectifier (usměrňovač) je unární operátor, který má 2 typy.
  - Half-wave rectification (jednosměrný usměrňovač), který odpovídá kódu

    ```
    return val > 0 ? val : 0;
    ```

    Tedy propouští pouze nezáporné hodnoty, ostatní nastaví na nulu.
  - Full-wave rectification (dvoucestný usměrňovač), který je ekvivalentní absolutní hodnotě.

Výše zmíněné způsoby výpočtu výstupních hodnot usměrňovačů fungují pouze pro znaménkové samply, pro bezznaménkové je nutné posunout neutrální hodnotu z 0 na $\frac{amplituda}{2}$.

- Unární operátor normalizace, který znormalizuje vstupní samply tím, že je vydělí maximální příchozí absolutní hodnotou (amplitudou).

- Unární operátor waveshaper na základě uživatelem zadané funkce transformuje vstupní samply na samply výstupní. V současné verzi programu probíhá zadávání funkce graficky (kreslením). Na x-ové souřadnici máme vstupní samply z intervalu [-1, 1] a na y-ové souřadnici samply výstupní, které jsou také z intervalu [-1, 1]. Z toho plyne, že příchozí samply jsou před vložením do funkce znormalizovány.

Operace obsahující dělení, tj. binární dělení a operace převrácení hodnoty jsou problémové, neboť v rámci diagramu povolujeme operaci normalizace. Předpokladem pro správné fungování normalizace je, že každá jednotka je schopná určit na základě minimálních a maximálních hodnot ze svých vstupů i minimální a maximální hodnotu na svém výstupu. Z nich pak můžeme získat maximální absolutní hodnotu, kterou vydělíme výstup, a tím ho znormalizujeme na hodnoty z intervalu [-1, 1]. Dále budeme pro zjednodušení mluvit jen o maximální absolutní hodnotě.

Ovšem pokud máme při dělení jmenovatele z intervalu (-1, 1), je maximální absolutní hodnota určená vydělením největšího možného čitatele nejmenším možným jmenovatelem v absolutní hodnotě. Zatímco největší číslo v absolutní hodnotě najdeme pro většinu jednotek z diagramu snadno (samozřejmě kromě případu dělení, o kterém teď mluvíme), tak nejmenší možné číslo v absolutní hodnotě nejsme schopni určit ani pro základní jednotky v diagramu. Například pro sinovou vlnu je generovaná maximální hodnota určena maximální absolutní hodnotou, která přijde na vstup reprezentující amplitudu. Zatímco minimální absolutní hodnota je daná kombinací všech vstupů, a tedy je velice nevyzpytatelná a může klesat libovolně blízko k nule. Problém se dal vyřešit dvěma způsoby. Prvním řešením je implementovat klasické dělení, ale pak diagram obsahující dělení nebude mít funkční normalizaci, což je poměrně nekonzistentní se zbytkem práce. Z toho důvodu jsme se rozhodli pro druhé řešení, kdy si vymyslíme nezápornou hodnotu, na kterou zarovnáme všechny příchozí jmenovatele, které jsou v absolutní hodnotě menší než tato uměle daná hodnota. Znaménko hodnoty, na kterou zarovnáváme, je určeno zarovnávanou hodnotou.

Implementace vypadá následovně:

```
public class Reciprocal extends UnaryOperator {
   /**
   * For amplitude = 1 this is the minimum allowed absolute value.
   */
   public static final double MIN_ALLOWED_VAL_FOR_ONE = 0.05;
}

public class BinaryDivision extends BinaryOperator {
   private double minAllowedVal = Reciprocal.MIN_ALLOWED_VAL_FOR_ONE;
   @Override
```

```
    public void calculateSamples() {
        minAllowedVal = Reciprocal.MIN_ALLOWED_VAL_FOR_ONE *
                        inputPorts[1].getMaxAbsValue();
        super.calculateSamples();
    }


    @Override
    public double binaryOperation(double a, double b) {
        if(b > -minAllowedVal && b < minAllowedVal) {
            if(b < 0) {
                b = -minAllowedVal;
            }
            else {
                b = minAllowedVal;
            }
        }

        return a / b;
    }
}
```

Jak jednotlivé operátory vypadají v diagramu, můžeme vidět na obrázku 4.41.

4.4.14 BPM algoritmy

Zde popíšeme jednotlivé algoritmy na hledání počtu úderů za minutu v signálu a jak moc se liší naše algoritmy od algoritmů popsaných na stránce [2], která sloužila jako hlavní zdroj informací pro tuto sekci. Informace o algoritmech 3 a 4 jsme navíc čerpali z internetového zdroje [1], ze kterého čerpala i výše uvedená stránka.

Beat (úder) nastane při výrazném zvýšení energie signálu oproti energiím předchozím, tj. nějaké historii signálu.

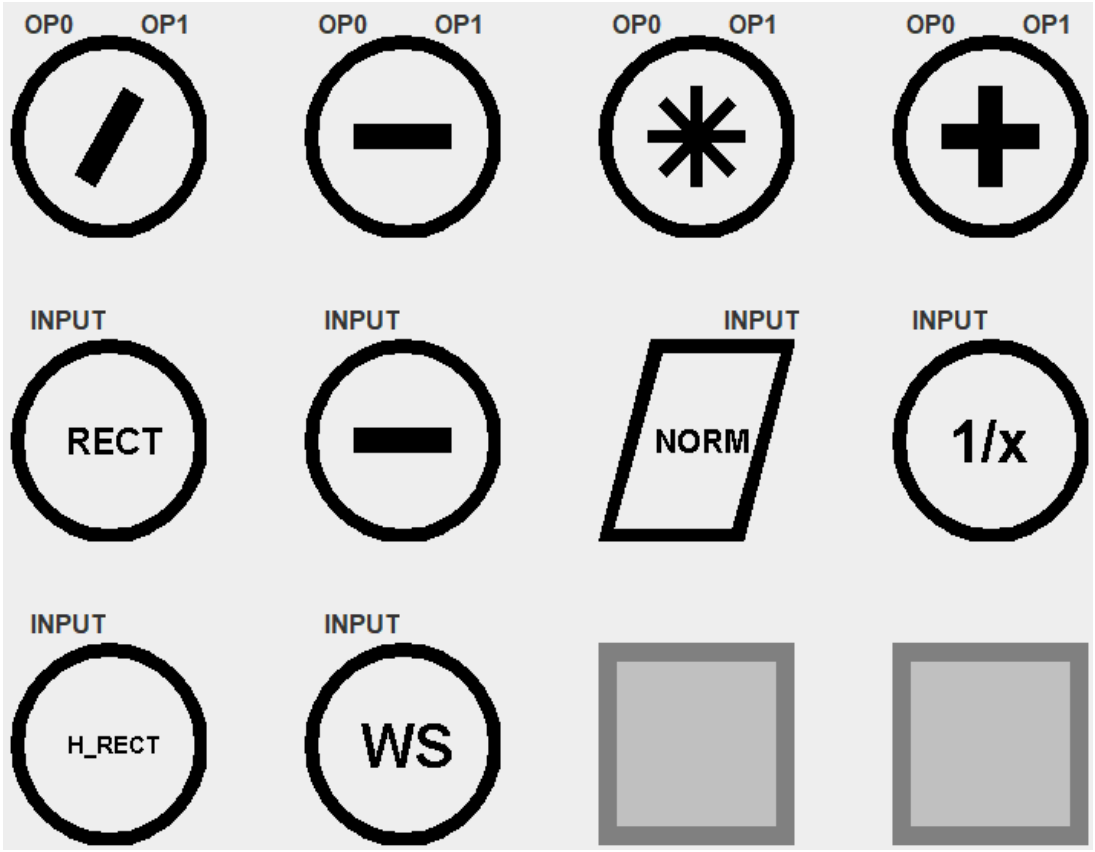
Prvním problémem, který musíme vyřešit, je jak moc velkou historii chceme brát v úvahu. Rozhodně nechceme, aby nastala situace, kdy energie ze začátku dlouhé písničky ovlivní detekci úderů na jejím konci, tj. že vezmeme jako historii celou písničku. Naopak nechceme ani přehnaně krátkou historii, pak budeme nacházet údery příliš často, neboť čím menší je historie, tím snazší je najít výkyv v energii.

Další otázkou je, jak velké budou jednotlivé úseky (okna), ve kterých budeme hledat údery. Historie pak znamená, kolik úseků zpět budeme brát při hledání úderu v úvahu.

Algoritmus 1

Vezmeme úsek signálu a podíváme se, jestli je jeho energie vůči průměru energií z n předchozích úseků větší, respektive průměr z historie ještě vynásobíme vhodným faktorem větším než jedna.

Jako velikost úseku zvolíme, při vzorkovací frekvenci 44100 Hz, 1024 vzorků,



Obrázek 4.41: Základní operátory implementované v programu, zleva doprava postupně. Na prvním řádku binární operátory: dělení, odčítání, násobení a sčítání. Na druhém řádku full-wave rectifier, unární mínus, operace normalizace a převrácení hodnoty. Na řádku posledním najdeme half-wave rectifier a waveshaper. Popis jednotlivých operátorů najdeme v sekci 4.4.13.

což je zhruba $\frac{1}{43}$ sekundy. Jako historii, tj. výše zmíněné n, zvolíme 43 takových úseků, což odpovídá zhruba jedné sekundě.

Energii jednoho úseků získáme takto:

$$E = \sum_{i=k}^{k+1023} X[i]^2 \tag{4.6}$$

Pro výpočet průměru z historie použijeme z důvodu maximální efektivity tzv. posuvný průměr, funguje následovně. Pokud máme spočtený průměr pro jednu historii (v našem případě 43 oken) a chceme vypočítat průměr nový, posunutý o jedno okno, stačí od průměru odečíst nejstarší energii, která je ještě součástí tohoto průměru a následně přičíst energii z nového okna. Přičtená a odečtená energie musí být samozřejmě vyděleny počtem oken zahrnutých v průměru.

Místo konstantního multiplikativního faktoru zvolíme faktor dynamický, ve kterém nějakým způsobem zohledníme velikost výkyvů oproti průměru. Novým faktorem bude rozptyl.

Definice 9. *(Rozptyl)*

$$V = \frac{1}{n} \sum_{i=0}^{LEN} (E[i] - E_{avg})^2 \tag{4.7}$$

LEN je v našem případě rovna 42, E značí energii, E[0] je nejstarší energie započítána v průměru a E[42] nejnovější, E_{avg} je samotný průměr.

Náš nový multiplikativní faktor bude $C = (-0.0025714 \cdot V) + 1.5142857$, tedy rozptyl využijeme k vychýlení od konstanty. Rozptyl je vždy nezáporný, a protože ho násobíme záporným číslem, tak od konstanty vždy odčítáme.

Naše implementace algoritmu se liší právě multiplikativním faktorem. Náš faktor je následující: $C' = (-0.0025714 \cdot V') + 1.8$, kde V' spočteme následovně.

```
V' = V * (varianceMultFactor / maxPossibleValueInVariance)
```

Výsledky algoritmu, který používal rovnici pro výpočet C převzatou z textu, nebyly vůbec uspokojivé. Nejprve jsme posunuli konstantu ve výpočtu na hodnotu 1.8, neboť nově vypočítaná variance V' má větší výkyvy, a proto nebyla konstanta 1.5142857 dostačující. Varianci násobíme hodnotou „*varianceMultFactor*“, neboť získaná maximální variance je moc přísná, proto jsme museli najít vhodnou konstantu, kterou jí vynásobíme. Nakonec jsme zvolili konstantu 10000. Ta dávala pro vzorkovací frekvenci 44100 Hz přijatelné výsledky. Ovšem konstantu musíme škálovat v závislosti na vzorkovací frekvenci. Hodnotu škálování jsme zvolili tak, aby vycházelo zhruba stejné BPM nezávisle na vzorkovací frekvenci.

```
double varianceMultFactor = 10000 * Math.pow(3.75,
                                              44100d / sampleRate - 1);
```

Oproti původnímu textu provádíme ještě jednu změnu. Zaznamenáváme úder nejdříve čtyři úseky po posledním úderu. Čtyři úseky jsou $\frac{4}{43}$ sekundy, což je o něco méně než jedna desetina sekundy. Idea je taková, že v takto malém úseku není v běžné hudbě více než jeden úder, a proto pokud bude nějaký zaznamenán tak rychle po předchozím úderu, jedná se nejspíše o chybu ve výpočtu. Tímto „trikem“ ovšem omezíme maximální možný zaznamenatelný počet úderů za minutu na 645, což je stále mnohem více, než má jakákoliv běžně poslouchaná hudba. Touto změnou došlo k výraznému zpřesnění algoritmu.

I takto jednoduchý algoritmus dává pro většinu signálů uspokojivé výsledky a díky malé výpočetní složitosti ho lze aplikovat na celý signál.

Algoritmus 2

Předchozí algoritmus můžeme vylepšit rozdělením signálu na frekvenční pásma a počítat energie z nich. Naše implementace dává správné výsledky pouze pro některé signály, proto se o tomto algoritmu nebudeme nadále zmiňovat, ale v programu jsme implementaci nechali.

Algoritmus 3

V předchozích algoritmech jsme se dívali na náš problém jako na statistický, nyní zkusíme k problému přistoupit jako k problému z oblasti zpracování signálů.

Myšlenka je následující. Vytvoříme n signálů, které mají k_i úderů za minutu a následně porovnáme, jak moc je náš signál s nimi podobný a vybereme ten nejpodobnější. Signál obsahující k_i úderů se nazývá comb filtr. Výše popsaný proces se nazývá combfilter processing.

Funkce určující podobnost dvou signálů X a Y vypadá následovně:

$$C(\alpha, \beta) = \int_{-\infty}^{\infty} [X(t) \cdot Y(\alpha \cdot (t - \beta))dt] \tag{4.8}$$

Tato funkce kvantifikuje množství energie, které si mohou vyměnit signály $X(t)$ a $Y(t - \beta)$. β má v rovnici za úkol odstranit rozdíly v časech, t udává čas, přes který i integrujeme.

Pokud nastavíme $\alpha = 1$ a přesuneme signály z analogové do digitální domény dostaneme:

$$C[\beta] = \sum_{k=-\infty}^{\infty} X[k] \cdot Y[k - \beta] \tag{4.9}$$

Tato rovnice odpovídá konvoluci, respektive hodnotě indexu (členu) β ve výsledku konvoluce. Vzhledem k tomu, že máme konečné signály, platí, že β jde od 0 do $2 \cdot (n - 1)$, kde n je délka signálu (oba signály jsou stejně dlouhé). Od n odečítáme jedničku, protože když si představíme signál jako polynom, máme na nultém indexu nultý stupeň. Náš signál tedy odpovídá polynomu $(n - 1)$-ního stupně a z algebry víme, že pokud máme součin polynomů o stupních v a w, pak je výsledkem polynom o stupni $v + w$. O polynomech mluvíme, protože násobení polynomů odpovídá konvoluci v diskrétním světě. Rovnice pak vypadá takto:

$$C[\beta] = \sum_{k=0}^{\beta} X[k] \cdot Y[\beta - k] \tag{4.10}$$

Hodnoty polí X a Y jsou mimo jejich hranice rovny nule.

Energii pak spočteme jako konvoluci, kde jednotlivé členy umocníme na druhou:

$$E_Y = \sum_{\beta=0}^{2\cdot(n-1)} C_{XY}[\beta]^2 \tag{4.11}$$

Signály odpovídající jednotlivým úderům za minutu vytvoříme následovně.

Každý T_i-tý sample nastavíme na maximální možnou hodnotu, v případě typu double se jedná o hodnotu jedna. Ostatní samply budou mít hodnotu nula. Tedy T_i určuje počet samplů mezi údery.

Hodnota T_i závisí na vzorkovací frekvenci a na tom, kolik úderů za minutu chceme generovat. Získáme jí následovně.

$$T_i = \frac{60 \cdot f_s}{BPM} \tag{4.12}$$

Pro pochopení této rovnice si stačí pouze uvědomit, že čitatel udává počet samplů v jedné minutě při dané vzorkovací frekvenci.

Rovnice 4.10 je příliš výpočetně náročná, proto se přesuneme do frekvenční domény pomocí FFT a v ní spočteme konvoluci, což je pouze produkt výsledků FFT. Ovšem přesun do frekvenční domény také není levný, z toho důvodu si vybereme n sekund uprostřed písničky, na které algoritmus aplikujeme.

Nejprve si zvolíme délku časového úseku, na který algoritmus aplikujeme. Dává smysl zvolit čas okolo pěti sekund. Poté musíme určit od kolika do kolika úderů za minutu algoritmus poběží a ještě musíme stanovit, jaký bude rozdíl v úderech za minutu mezi vedlejšími BPM poli. Jako spodní hranici zvolíme 60

BPM a 240 BPM jako vrchní. Budeme brát pouze údery za minutu dělitelné deseti.

Nyní už můžeme provést samotný algoritmus, tj. aplikujeme na BPM pole a zkoumaný signál (respektive jeho část) algoritmus FFT a vynásobíme každé BPM pole s naším signálem ve frekvenční doméně. Pokud navíc dáme jednotlivé výsledky do absolutní hodnoty (tj. bereme vzdálenost komplexních čísel od nuly), dá nám jejich součet hledanou energii.

$$E_{BPMc} = \sum_{k=0}^{N} |(ta[k] + i \cdot tb[k]) \cdot (tl[k] + i \cdot tj[k])| \tag{4.13}$$

ta a tl odpovídají reálným částem komplexních čísel získaných z FFT, tb a tl imaginárním.

Z výsledků poté vybereme c takové, že E_{BPMc} je maximální, kde c určuje index pole reprezentujícího dané údery za minutu.

Pro zlepšení výsledků můžeme na původní signál (respektive jeho část) aplikovat derivační filtr, který nám zvýrazní změny amplitud v signálu, a tím usnadní detekci úderů později v algoritmu. Filtr vypadá následovně:

$$Y[i] = \frac{1}{2}(X[i] - X[i-1]) \tag{4.14}$$

Algoritmus 4

Předchozí algoritmus lze ještě vylepšit a to podobně jako jsme učinili při přechodu z algoritmu 1 na algoritmus 2. Totiž pokud nemáme rozdělení na frekvenční pásma, platí, že algoritmus velice těžko rozliší úder činelu a úder bubnu, proto rozdělíme výsledek FFT do pásem.

Jednou z výhod algoritmu s pásmy je, že můžeme upravit faktor, kterým násobíme průměr energie z pásma, a tím dát větší důraz na určité frekvence.

Jaká je ideální velikost pásma a vyplatí se, aby by byla všechna pásma stejně velká? Jak jsme již zmínili v sekci s decibely, lidské vnímání zvuku je logaritmické a to nejen pro intenzity, ale i pro frekvence. Pro člověka je mnohem jednodušší rozeznat rozdíl mezi frekvencemi 80 a 90 Hz než mezi frekvencemi 10000 a 10010 Hz, kdy nejspíše ani většina lidí nepozná, že nějaký rozdíl existuje. To je způsobeno především tím, že většina zvuků, včetně lidské mluvy se pohybuje v nízkých frekvencích. To je mimo jiné důvod, proč stačí pro posílání hlasu poměrně nízké vzorkovací frekvence, čehož například využívaly a zřejmě stále využívají telefonní společnosti, protože tím lze značně redukovat množství dat, které je nutné poslat po telefonní lince. Z tohoto důvodu budeme frekvenční rozsah pásma zvětšovat se zvyšující se frekvencí a to ideálně logaritmicky, například první pásmo bude obsahovat pouze jednu přihrádku, druhé dvě, třetí čtyři a tak dále, čehož ale nemůžeme jednoduše dosáhnout, tak se takovému rozdělení alespoň pokusíme přiblížit.

Použijeme šest pásem. BPM pole do pásem rozdělovat nemusíme.

Kroky algoritmu jsou až do vypočítání FFT stejné, včetně výpočtu FFT. Poté následuje rozdělení výsledků FFT do pásem, provedeme ho následovně. Vezmeme nové vynulované pole stejně dlouhé jako pole obsahující výsledek FFT a do indexů odpovídajících danému frekvenčnímu pásmu zkopírujeme hodnoty z originálního pole s výsledky FFT. Ostatní hodnoty necháme na nule.

Poté pro každé pásmo provedeme výpočet 4.13. Akorát změníme značení na tas a tbs, kde s značí číslo sekce, aby bylo jasné, že se nejedná o původní výsledky FFT:

$$E_{BPMc,s} = \sum_{k=0}^{N} |(tas[k] + i \cdot tbs[k]) \cdot (tl[k] + i \cdot tj[k])| \tag{4.15}$$

Nyní známe pro každé pásmo jeho energii. Teď už jen zbývá učinit rozhodnutí, jakým způsobem z těchto informací získáme BPM. Nejprve se rozhodneme, jestli se zaměříme jen na nějaké frekvence, nebo na všechny. V našem případě se zaměříme na všechny frekvence. Nyní máme několik možností. První je pro každé BPM pole spočítat součet energií ze všech frekvenčních pásem a pak vybrat BPM pole s maximálním součtem. Tento přístup vypadá v kódu následovně. V následujících kódech můžeme vidět dvoudimenzionální pole energies. Jedná se o pole s energiemi pro jednotlivá pásma. První dimenze určuje frekvenční pásmo, druhá, o jaké BPM pole se jedná.

```
public int calculateBPMFromEnergies(double[][] energies, int startBPM,
                                    int jumpBPM, int bpmCount) {
    int maxBPMIndex = 0;
    double maxEnergy = 0;
    double[] totalEnergies = new double[bpmCount];

    for (int bpmIndex = 0; bpmIndex < energies[0].length; bpmIndex++) {
        totalEnergies[bpmIndex] = calculateEnergySum(energies,
                                                     bpmIndex);

        // Multiply here if we want to emphasize some frequencies
        if (totalEnergies[bpmIndex] > maxEnergy) {
            maxEnergy = totalEnergies[bpmIndex];
            maxBPMIndex = bpmIndex;
        }
    }

    return CombFilterBPMGetterIFace.getBPMFromIndex(startBPM,
                                                    jumpBPM,
                                                    maxBPMIndex);
}


private static double calculateEnergySum(double[][] energies,
                                         int bpmIndex) {
    double energy = 0;

    for(int i = 0; i < energies.length; i++) {
        energy += energies[i][bpmIndex];
    }

    return energy;
}
```

Druhou možností je spočítat barycentrum:

```
double maxEnergy;
double maxEnergySum = 0;
double sum = 0;
int maxBpmInd = -1;
int bpm;

for(int i = 0; i < energies.length; i++) {
        maxEnergy = 0;
        for(int bpmInd = 0; bpmInd < energies[i].length; bpmInd++) {
                if(energies[i][bpmInd] > maxEnergy) {
                        maxBpmInd = bpmInd;
                        maxEnergy = energies[i][bpmInd];
                }
        }

        maxEnergySum += maxEnergy;
        bpm = CombFilterBPMGetterIFace.getBPMFromIndex(startBPM,
                                                       jumpBPM,
                                                       maxBpmInd);
        sum += maxEnergy * bpm;
}
return sum / maxEnergySum;
```

Shrnutí

V naší práci implementujeme čtyři algoritmy a to algoritmus 1, algoritmus 2 a dvě verze algoritmu 4, které se liší podle toho, jestli bereme BPM s maximální energií, nebo spočítáme jejich barycentrum. Nazveme je 4A a 4B, kde 4B je algoritmus používající barycentrum. Implementace algoritmu 4A a 4B se liší od původního článku pouze v délce vybraného úseku. Pro algoritmus používající barycentrum používáme časový úsek dlouhý 6.15 sekund. Pro algoritmus 4A máme časový úsek o délce 2.2 sekundy, protože při volbě větší doby má algoritmus tendenci dávat vyšší BPM, což je spíše na škodu. Totiž získaná hodnota může být občas násobek, respektive dělitel správné hodnoty BPM. V takové situaci je lepší získat dělitele, čehož dosáhneme například zkrácením doby na výše zmíněné 2.2 sekundy, navíc tím ušetříme netriviální množství času. Proč k získání nesprávné hodnoty dochází, zmíníme v následujícím odstavci.

Algoritmy 4A a 4B by měly být nejpřesnější, což potvrzuje i naše testování. Vždy byly jejich výsledky velmi blízko korektní hodnotě. Algoritmus 1 je těsně za nimi a i ten dává velice dobré výsledky. Pro určité typy písní dává dokonce i lepší výsledky, ale častěji je spíše méně přesný než algoritmy 4A a 4B. Algoritmus 4A má ten problém, že občas nevrátí správný výsledek. Například někdy dostaneme 60 BPM místo 120 BPM, neboť 60 BPM mělo o trochu více energie. To je z toho důvodu, že výkyvy energií se pohybují na násobcích skutečné hodnoty, resp. na výsledcích dělení mocninou dvojky. Například pokud budeme mít skutečnou hodnotu rovnou 120 BPM, najdeme lokální výkyvy energie na 30, 60, 120, 240, 360 BPM. Samozřejmě můžeme ve výčtu násobků pokračovat. Pro takové případy je dobré porovnat výsledky s algoritmem 4B, u kterého tento problém nenastává.

Výsledky algoritmu 2 jsou, jak již bylo zmíněno výše, správné pouze pro některé signály.

Testování všech algoritmů probíhalo na náhodných písničkách a na několika vygenerovaných audio stopách, ve kterých se po dobu jedné minuty opakoval úder bicích. Testované rytmy byly 60, 70 a 120 úderů za minutu. Stopy byly generovány pomocí zápočtového programu napsaném v jazyce C++ (`https://github.com/RadStr/Music-Keyboard`). Generované audio stopy můžeme nalézt v adresáři test_data.

Implementace algoritmů najdeme ve třídách `BPMSimple`, `BPMSimpleWithFreqBands`, `CombFilterBPMAllSubbandsGetter` a `CombFilterBPMBarycenterGetter` uvnitř metody `computeBPM`.

4.4.15 Testy

Pro určité v rámci práce často používané algoritmy bylo nutné napsat testy.

Jedná se především o algoritmy pro analýzu a algoritmy pro převody samplů, které jsou používané v celé práci. Jejich testování dává smysl, a navíc je lze snadno otestovat na rozdíl od části se syntezátorem, či části s přehrávačem. Implementace je řešena prostým výpisem na standardní výstup, což má dvě hlavní výhody a těmi jsou jednoduchost a skutečnost, že není nutná další knihovna. Sice není výpis výsledků testů na standardní výstup v určitých ohledech úplně ideální, ale svůj účel splňuje. Alternativou bylo použít nějaký z Java frameworků pro testování, například JUnit test. Testy najdeme ve třídě ProgramTest.

5. Uživatelská dokumentace

V této kapitole zjistíme jak program spustit 5.1 ze spustitelného .jar souboru, následně i jak ho zkompilovat ze zdrojových kódů 5.2. V sekci 5.3 se podíváme na alternativu k tomuto přístupu. Zde se dozvíme jak přidat projekt do vývojového prostředí, což typicky chceme, pokud hodláme psát pluginy, či rozšiřovat kód. Následně si v sekci 5.4 ujasníme některé věci ohledně použitých knihoven a v sekci 5.5 se zmíníme o testovacích datech.

V sekci 5.6 zjistíme na co si dát při používání programu pozor. Poté společně zjistíme jak program používat 5.7 a jako poslední nás čeká sekce 5.8, ve které se dozvíme jak psát pluginy a kam je vkládat.

Použitou verzí jazyka je Java 8. Pro správný běh programu se předpokládá, že celý projekt, tj. adresář libs, adresář se zásuvnými moduly a spustitelný soubor .jar v případě, kdy spouštíme program pomocí spustitelného souboru, je uvnitř jednoho adresáře. Pokud projekt kompilujeme, jedná se o adresář libs, adresář se zásuvnými moduly, adresář resources s obrázky použitými v programu a o samotné zdrojové kódy v adresáři src. Adresář se zásuvnými moduly může být samozřejmě vynechán.

5.1 Spouštění

Pro spuštění aplikace bez zásuvných modulů stačí spustit soubor „DiaSynth_Original.jar“.

Pro spuštění aplikace se zásuvnými moduly stačí spustit soubor „Diasynth_Updater.jar“, který vytvoří a spustí soubor „Diasynth_Modified.jar“ obsahující pluginy. „DiaSynth_Original.jar“ a „Diasynth_Updater.jar“ musí být ve stejném adresáři. Jedná se o provizorní řešení, viz úvod do sekce s pluginy 5.8.

Spuštění .jar souboru vyžaduje instalaci platformy Java, přesněji Java Runtime Environment (JRE) obsahující Java Virtual Machine, která se postará o překlad Java bytekódu do strojového kódu cílového počítače.

5.2 Kompilace a sestavení projektu

Pro sestavení používáme nástroj Apache Ant (viz https://ant.apache.org/). Nástroj je nejprve nutné nainstalovat a přidat na system path. Navíc potřebujeme Java Development Kit (JDK) pro kompilaci a následné spuštění. Nástroj ANT se ovládá z příkazové řádky a pokyny pro sestavení a spuštění jsou následující. Příkazy jsou uvedeny v uvozovkách („“).

5.2.1 Postup sestavení pomocí ANT

1. Stáhneme projekt ve formátu .zip a rozbalíme ho.

2. Otevřeme příkazovou řádku a přesuneme se v ní do adresáře, ve kterém se nachází rozbalený projekt a z něho se přesuneme do adresáře obsahujícího soubor build.xml. K přesunutí využijeme příkaz „`cd`".

3. Projekt spustíme příkazem „`ant run`". V rámci tohoto příkazu se zavolají příkazy „`ant compile`" a „`ant jar`". První příkaz projekt zkompiluje, tj. vytvoří .class soubory. Druhý příkaz vyrobí soubor Diasynth_Original.jar z .class souborů. Vyrobený .jar soubor nalezneme ve složce build/jar.

5.3 Práce na projektu ve vývojovém prostředí (IDE)

Po stažení projektu ze SISu nebo repozitáře `https://github.com/RadStr/DiaSynth` je nutné nejprve projekt uložený ve formátu .zip rozbalit. Poté stačí pouze vyřešit odkazy na knihovny, které nalezneme v adresáři libs. Učiníme tak uvnitř vývojového prostředí. Přesné návody pro jednotlivá vývojová prostředí zde nebudeme uvádět.

5.4 Knihovny

Knihovny vkládáme do adresáře libs. Knihovny distribuujeme spolu s programem, proto uvnitř projektu najdeme soubory LICENSE.txt a README.txt, které jsou zahrnuty kvůli licencím použitých knihoven. Pro úplnost znovu uvedeme webové stránky knihoven.

Načítání MP3 souborů: `http://www.javazoom.net/mp3spi/mp3spi.html`
Algoritmus FFT: `https://github.com/wendykierp/JTransforms`.

5.5 Testovací data

Testovací data nalezneme uvnitř adresáře test_data. Obsahuje soubory použité na testování BPM algoritmů a podadresář diagrams, ve kterém jsou uloženy zajímavé diagramy.

Zásuvné moduly jsou uloženy uvnitř adresáře Diasynth-plugins. Slouží pouze pro ukázku funkčnosti pluginování, takže se nejedná o užitečné operace.

5.6 Varování

Nejprve upozorníme na to, že program byl testovaný na operačním systému Windows 10. Protože je Java platformě nezávislá, měl by program fungovat stejně dobře i na ostatních platformách. Ale jak jsem se sám přesvědčil, tak i ve standardní knihovně jsou chyby, takže je miniaturní šance, že program bude mít na jiných platformách odlišné chování.

Jelikož se jedná o práci se zvukem, je nejlepší začínat na minimálních amplitudách a minimální hlasitosti. K tomu se váže další varování. Kontrola hlasitosti na hracím panelu vedle play tlačítka používá master gain control poskytnutý jazykem Java, který neumožňuje úplné zeslabení zvuku, tedy pokud posuneme posuvník na minimální hodnotu, je stále slyšet programem generovaný zvuk. Pro vypnutí je buď nutné nastavit amplitudy násobící výstup na 0, nebo kliknout na tlačítko mute.

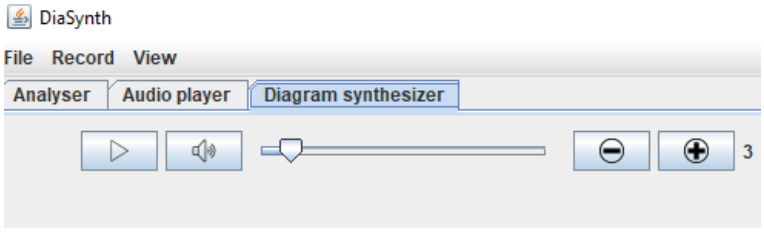
Další varování se týká psaní pluginů pro syntezátor. Ačkoliv byla snaha navrhnout rozhraní tak, aby se už neměnilo, je možné, že v budoucnu dojde k přidání komponenty, která bude vyžadovat poupravení již dříve napsaných pluginů. Pokud taková situace nastane a program bude někým využíván, budou poskytnuty informace o změně spolu s návody a příklady, podle kterých bude možné poupravit pluginy pro nové rozhraní.

Poslední varování se týká zásuvných modulů samotných. Zásuvné moduly mohou obsahovat libovolný kód, tedy i záměrně škodlivý pro uživatele. Varování se samozřejmě týká pouze zásuvných modulů třetích stran. Všechny zásuvné moduly napsané v rámci práce nebudou provádět nic jiného než výpočty odpovídající dané funkcionalitě.

5.7 Ovládání

Ovládání by mělo být přímočaré, navíc většina částí obsahuje popisky s nápovědou („tooltip“), tedy by neměl být problém program ovládat i bez přečtení následujících částí této kapitoly, ale i přesto se přečtení doporučuje.

Mezi komponentami přepínáme pomocí kliknutí na záložky odpovídající jednotlivým částem „`Analyser`“, „`Audio player`“, „`Diagram synthesizer`“, viz 5.1.

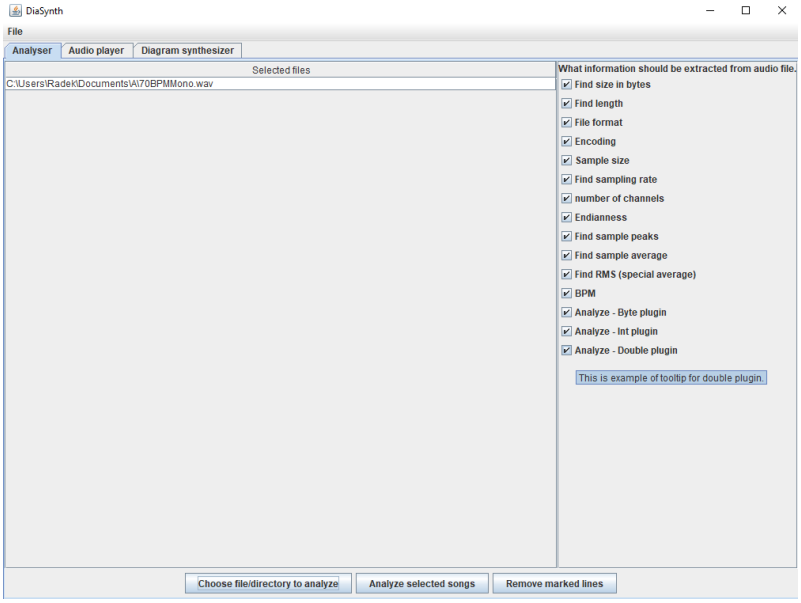


Obrázek 5.1: Obrázek levého horního rohu programu Diasynth

5.7.1 Analyzátor

Nejprve v okně s analyzátorem klikneme vlevo nahoře na záložku „`file`“ a vybereme si, jestli chceme analyzovat nové soubory, nebo zobrazit již dříve analyzované.

Po kliknutí na „`Choose files to analyze`“ v záložce „`file`“ se zobrazí okno sloužící pro analýzu vybraných souborů. Okno můžeme vidět na obrázku 5.2. Na pravé straně si můžeme vybrat, co za informace chceme analyzovat. Pokud chceme vybrat nové soubory k analýze, stiskneme „`Choose file/directory to analyze`“. Pokud chceme soubory, které se zobrazují v „selected files“ analyzovat, stiskneme „`Analyze selected songs`“. Poté je nutné počkat, dokud se soubory nezanalyzují. Program při analýze nereaguje. Pokud jsme některé soubory vybrali omylem, můžeme je označit a zmáčknout tlačítko „`Remove marked lines`“, které je odstraní z listu souborů určených pro analýzu. Při analýze souboru nepodporovaného formátu by měl program bez problému pokračovat. Pokud ovšem budeme mít soubor s hlavičkou odpovídající podporovanému audio souboru, ale



Obrázek 5.2: Okno pro výběr souborů k analýze

data délkou nebudou odpovídat, či nastane nějaká jiná nesrovnalost, lze očekávat nedefinované chování.

Stisknutí „`Show analyzed files`“ v záložce „`file`“ nás přesune do knihovny dříve analyzovaných souborů, které můžeme spatřit ve spodní části. Okno obsahující knihovnu můžeme vidět na obrázku 5.3. Tyto soubory můžeme označit a stisknutím tlačítka „`Add to selected`“ je přesuneme do vrchní části okna. Se soubory ve vrchní části už můžeme provádět další operace, které nyní popíšeme. Slovo označený bude v následujícím textu znamenat označený myší. Akce „`Show info about song`“ zobrazí analyzované informace o prvním označeném souboru. „`Delete Selected`“ vymaže analyzované informace o souborech a vymaže je z listu „Selected files“ a „All analyzed files“. Tlačítko „`Clear selected`“ odstraní všechny soubory z listu „Selected files“, ale nevymaže jejich analyzovaná data. „`Unmark selected`“ vymaže označení ze souborů označených v „Selected files“. „`Unmark from all files`“ udělá totéž pro soubory v listu „All analyzed files“.

5.7.2 Přehrávač

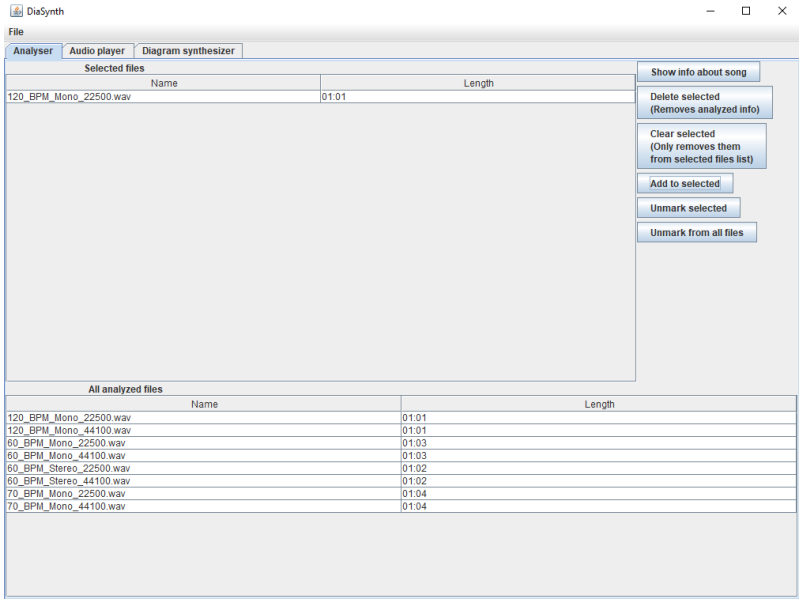
Pro lepší pochopení textu může pomoci obrázek přehrávače 5.4.

V pravé horní části vidíme měřič intenzity zvuku, který můžeme schovat pomocí tlačítka „`Show decibel meter`“ uvnitř záložky „`View`“. Nalevo od měřiče se nachází popořadě: tlačítka pro přibližování/oddalování, následované kontrolou hlasitosti, tlačítkem pro vypnutí/zapnutí zvuku a tlačítkem pause/play.

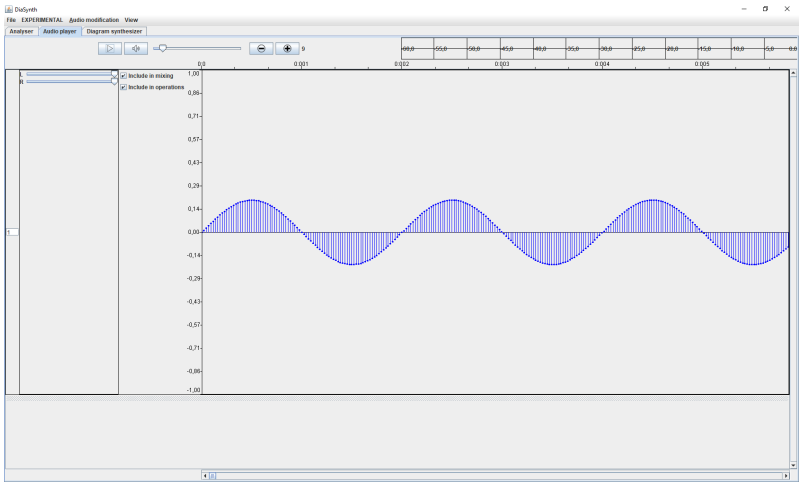
V levém horním rohu máme záložky „`File`“, „`Experimental`“ a „`Audio modification`“, která není přístupná, dokud nenačteme zvukovou stopu.

Nyní se podíváme na záložku „`File`“, jejíž obsah můžeme vidět na obrázku 5.5.

Po stisknutí „`File`“ můžeme přidat nové prázdné vlny pomocí „`Add empty wave(s)`“ a načíst vlny ze souboru pomocí „`Add waves`“. Načteme tolik vln, kolik má formát v audio souboru kanálů. „`Add mono wave`“ načte zvukový soubor,



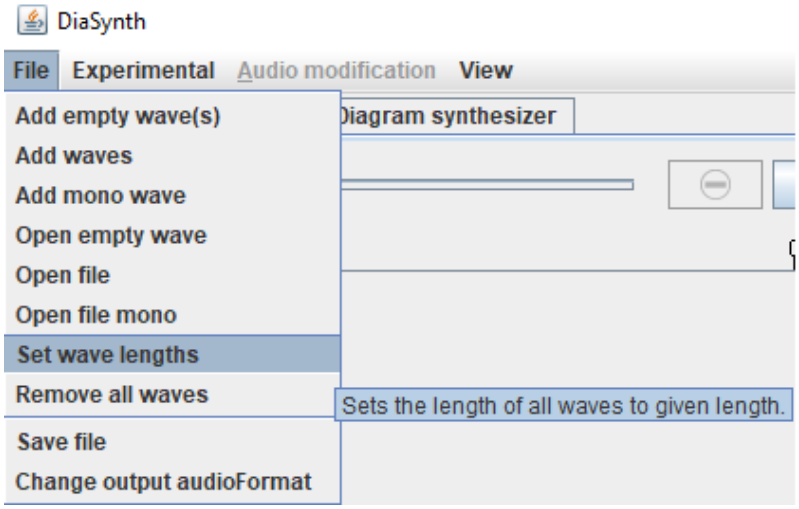
Obrázek 5.3: Okno s knihovnou dříve analyzovaných souborů



Obrázek 5.4: Okno s přehrávačem

ale před jeho přidáním ho převede do formátu mono, tedy do přehrávače následně přibude pouze jedna zvuková stopa. „add“ varianty přidávají nové vlny k již stávajícím, zatímco „open“ varianty všechny vlny kromě nově přidaných vymažou. „`Set wave lengths`“ zkrátí, či prodlouží velikost všech vln na danou délku. „`Remove all waves`“ vymaže všechny vlny z přehrávače.

Pod oddělovačem najdeme tlačítko „`Save file`“, které zajistí uložení mixu všech zvukových stop do souboru ve formátu daném pomocí „`Change output audio format`“. Všechny načtené vlny jsou převedeny do tohoto formátu. Pokud formát změníme, můžeme si vybrat, jestli chceme v přehrávači již načtené stopy převést do nového formátu. Vzhledem k tomu, že interně pracujeme s typem double, jedná se pouze o převod vzorkovací frekvence. Zbylé parametry audio formátu jsou použité až při přehrávání, či ukládání. Pokud bude nově zvolená vzorkovací frekvence menší než původní a neprovedeme převod, výsledné audio bude zpomalené. Pokud naopak bude vzorkovací frekvence větší, bude zrychlené.

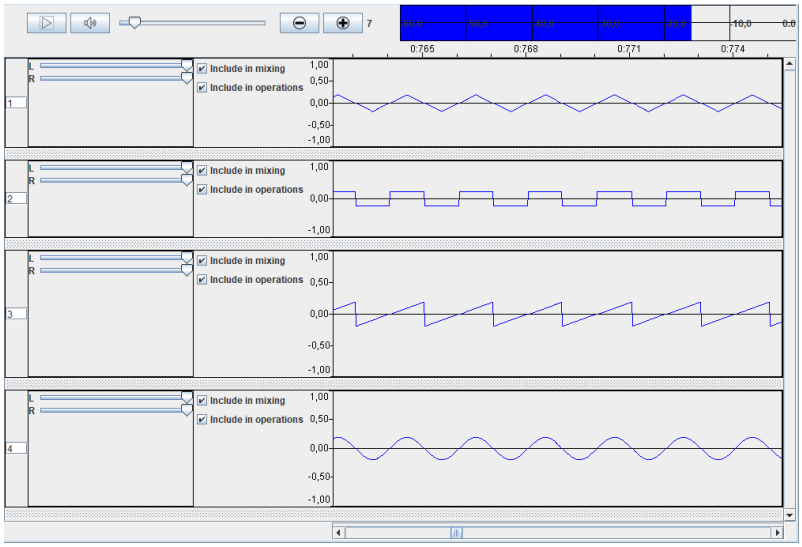


Obrázek 5.5: Záložka „`File`“ uvnitř přehrávače

Je nutné podotknout, že převod na nižší vzorkovací frekvenci způsobuje ztrátu informací o vlně, neboť máme méně samplů na reprezentaci vlny a dle definice 1 dokážeme reprezentovat menší maximální frekvenci v signálu. Z toho plyne, že ideálně chceme buď převést na vyšší vzorkovací frekvenci, nebo mít jistotu, že frekvence v originálním signálu jsou menší než nyquistova frekvence pro novou vzorkovací frekvenci, jinak může vznikat aliasing.

Popis stop

Popis stop jde zleva doprava. Přehrávač s několika načtenými stopami můžeme spatřit na obrázku 5.6.



Obrázek 5.6: Přehrávač obsahující několik načtených vln

Nejprve máme číslo značící pořadové číslo vlny, které můžeme změnit a po potvrzení klávesou „enter“ dojde k prohození daných vln. Pořadí vln můžeme také změnit táhnutím myši. Pro zahájení posouvání musíme kliknout mimo sa-

motnou vlnu a kontrolní tlačítka. Na akci posouvání upozorňujeme rozsvícením pořadového čísla červenou barvou.

Dále následují posuvníky kontrolující amplitudy jednotlivých výstupních kanálů vlny. Funkcionalitu interně realizujeme vynásobením samplů těsně před mixováním číslem $\frac{i}{100}$, kde i je dané pozicí posuvníku a odpovídá celému číslu mezi 0 a 100.

Dále následují zaškrtávací políčka. Pokud zaškrtneme „`Include in mixing`“, bude daná vlna součástí mixování, jinak bude při mixování ignorována. Pokud není políčko zaškrtnuté, nezobrazuje se na dané vlně zelená úsečka značící pozici ve vlně. Políčko „`Include in operations`“ určuje, jestli chceme provést operaci z „`Audio modification`“ na danou vlnu. Pokud není políčko zaškrtnuté, zmizí červené označení vlny určující část, na kterou provedeme operaci.

Zbytek stopy vyplňuje samotná zvuková vlna. Pokud ukážeme na bod ve vlně a chvíli počkáme, zobrazí se nápověda s informacemi o dané pozici.

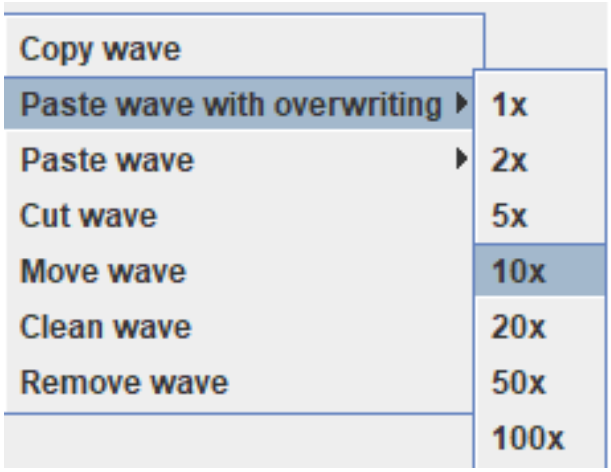
Přibližovat, respektive oddalovat na pozici ve vlně můžeme buď pomocí tlačítek ve vrchní části přehrávače, nebo pokud ukazujeme na vlnu, lze provést přiblížení pomocí kolečka na myši. Bod přiblížení je určen následujícím pseudokódem.

```
if(zelena_usecka) {
        zoom_to_zelena();
}
else {
        if(na_zacatku_vlny) {
                zoom_to_zacatek();
        }
        else if(na_konci_vlny) {
                zoom_to_konec();
        }
        else {
                zoom_to_prostredek();
        }
}
```

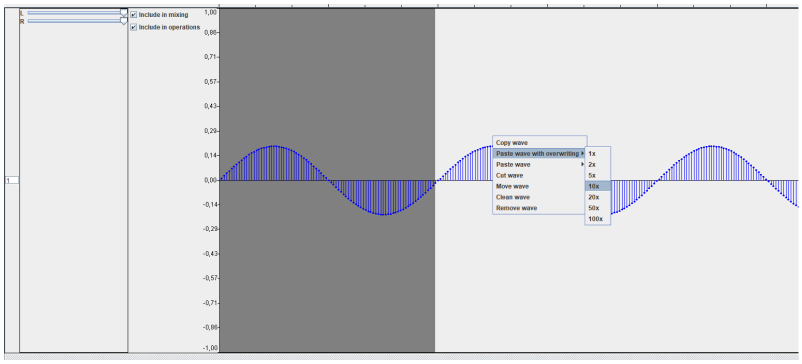
Výšku vlny můžeme změnit chycením spodní části vlny a následným táhnutím myši. Pokud máme v přehrávači načtené aspoň dvě vlny, je možné změnit výšku vybrané vlny i mimo hranice okna. Stačí pokračovat v tahu myší mimo panel s vlnami. Táhnutí směrem dolu je trochu kostrbaté. Musíme provést dostatečně mnoho pohybů myší směrem dolů, aby k roztažení došlo. Pro táhnutí směrem nahoru stačí najet nad panel s vlnami.

Pokud klikneme na vykreslenou vlnu pravým tlačítkem, máme na výběr z několika záložek, které můžeme vidět na obrázku 5.7. „`Copy wave`“ umožní danou vlnu, respektive její označenou část okopírovat. „`Cut wave`“ funguje stejně jako copy, akorát po provedení akce se vyjmutá část vlny nahradí samply s hodnotou 0. Po zkopírování, respektive vyjmutí se daná část označí šedým, respektive růžovým obdélníkem. Označení vlny šedým obdélníkem můžeme vidět na obrázku 5.8. Okopírovanou část lze vložit do další vlny a to buď pomocí „`Paste wave with overwriting`“, čímž přepíšeme samply v cílové vlně kopírovanou vlnou, nebo pomocí „`Paste wave`“, která samply vloží na danou pozici, rozšíří vlnu a k přepisu žádných samplů nedojde. Pozice vložení odpovídá místu, na kterém

jsme stiskli pravé tlačítko myši. Obě tlačítka umožňují určit, kolikrát chceme vlnu vložit, viz obrázek 5.7. „`Move wave`“ funguje tak, že si nejprve označíme část vlny, kterou chceme posunout a poté klikneme pravým tlačítkem myši na pozici, kam jí chceme posunout, a provedeme akci zmíněným tlačítkem. Operace posunutí probíhá v rámci jedné plny a odpovídá kombinaci operací „`Cut wave`“ a „`Paste wave`“. Tlačítko „`Clean wave`“ nahradí označenou část vlny nulami. „`Remove wave`“ danou vlnu úplně vymaže z přehrávače.



Obrázek 5.7: Výčet záložek, který se zobrazí po pravém kliknutí na vlnu.



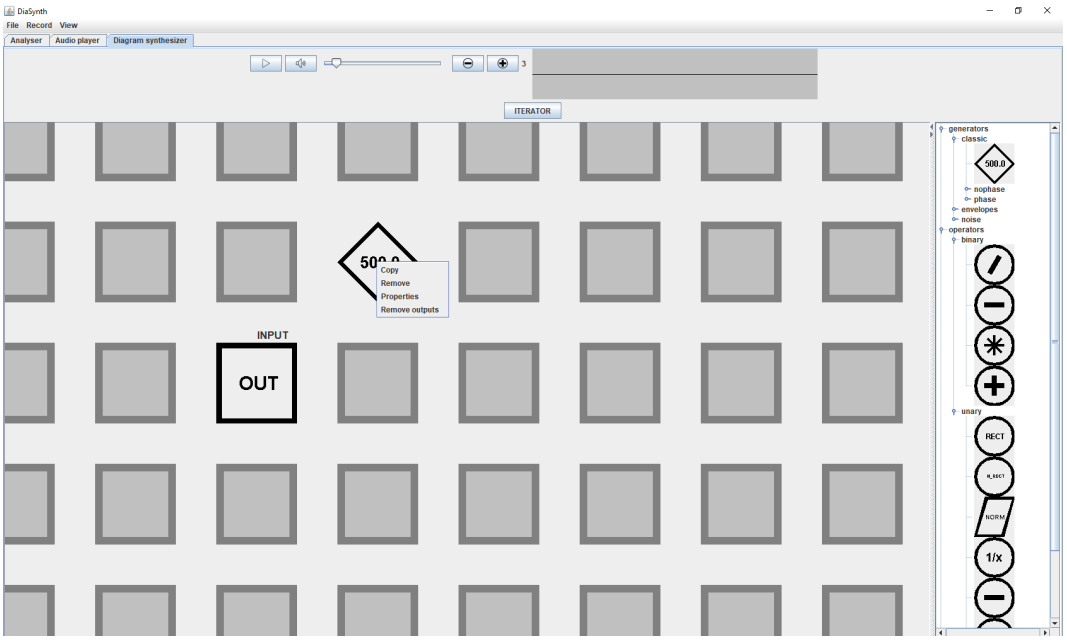
Obrázek 5.8: Označení vlny šedým obdélníkem znamená, že tato část vlny je připravena ke kopírování.

Poslední částí přehrávače je záložka „`Audio modification`“, až na tlačítka pro operace mezi dvěma vlnami není potřeba vysvětlení. Když chceme provést operaci se dvěma vstupními vlnami, označíme jednu část vlny pomocí „copy“, tím vytvoříme vstupní vlnu operace a poté označíme část, kterou chceme použít jako druhý vstup pro operaci. Druhá označená část je i místem pro uložení výsledku. Operaci provádíme po vlnách, tj. pro všechny vlny, které mají zaškrtnuto „`Include in operations`“ kromě vstupní, se provede „out = in op out“. out je výstupní vlna, in vstupní vlna a op je provedená operace. Před provedením se objeví dialog, ve kterém určíme, jakým způsobem chceme zarovnat délky vln. Možnost bez zarovnání funguje tak, že pokud je vstupní vlna kratší, použije se cyklicky. Ostatní možnosti není potřeba vysvětlovat. Zarovnávání se provádí

změnou nejpravějšího označeného indexu, tedy neměníme počáteční bod označení. Obecnou verzi operací, tj. „out = in op in2“, přidáme v budoucnu. Chybí, protože se na ní pozapomnělo.

5.7.3 Syntezátor

Jak syntezátor vypadá, můžeme vidět na obrázku 5.9.



Obrázek 5.9: Okno se syntezátorem

Na pravé straně máme menu obsahující výpočetní jednotky, jehož velikost lze měnit táhnutím myši. Pro přidání panelu do diagramu stačí kliknout na panel z menu a přidat ho do jednoho z volných čtverečků v rámci diagramu.

Pohyb po panelu obsahující diagram probíhá táhnutím myši. Pro rychlý pohyb mezi panely slouží tlačítko iterátor. Pro přibližování, respektive oddalování můžeme použít tlačítka ve vrchní části panelu, která způsobí přiblížení, respektive oddálení do prostředního bodu diagramu. Pro přibližování můžeme použít i kolečko myši, pak je bodem přiblížení kurzor myši.

Změnu pozice panelu v diagramu provedeme takto. Klikneme na panel, přeneseme kurzor myši na cílovou pozici a klikneme znovu. Akce připojování probíhá následujícím způsobem. Klikneme dvakrát na panel, jehož výstup chceme připojit, poté klikneme na cílový panel, ke kterému chceme výstup připojit, a vybereme odpovídající vstup.

Pokud klikneme pravým tlačítkem na panel, pak se v závislosti na typu panelu objeví různá tlačítka. Všechny panely s výjimkou výstupního panelu obsahují tlačítko „`Remove`“, které panel vymaže z diagramu, tlačítko „`Remove outputs`“ a tlačítko „`Copy`“, které vytvoří kopii panelu. Kopie sdílí s originálem pouze vstupy, kopírování výstupů totiž nedává moc smysl.

Některé panely obsahují tlačítko „`Properties`“, po jehož stisknutí se objeví okno, ve kterém lze nastavit určité parametry výpočetní jednotky. Ze základních prvků diagramů obsahují „`Properties`“ pouze čtyři typy panelů. Prvním je panel generující konstantu, tomu lze v properties nastavit generovanou konstantu.

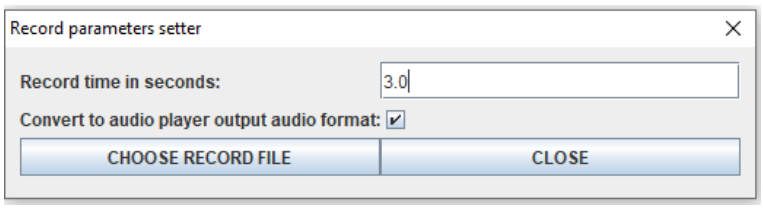
Dále je to pak wavetable panel, kterému lze nastavit interní vlnu odpovídající vlně uložené v audio souborů. Wavetable bez zadané vstupní vlny generuje sinovou vlnu. Třetím panelem je operátor waveshaper, kterému můžeme v properties nastavit funkci použitou pro generování výstupu. Posledním panelem s properties je výstupní panel, kde první prvek v zobrazeném dialogu udává amplitudu výstupního signálu. Pokud je druhý prvek dialogu nastavený na „`false`“, je zajištěno, že výstupní vlna nepřesáhne hodnotu danou prvním parametrem. Zároveň platí, že pokud amplituda vstupní vlny hodnotu parametru nepřesáhne, pak zůstane vstupní vlna nezměněna. Pokud je prvek nastaven na „`true`“, pak první parametr dialogu určuje amplitudu generované vlny, tedy k normalizaci dojde vždy.

Poslední tlačítka, která po pravém kliknutí najdeme, pracují se vstupními porty. Tlačítko „`Remove input`“ vymaže vstup vybraného portu. Tlačítko „`Remove inputs`“ odstraní připojení do všech vstupních portů. Panely bez vstupního portu tyto tlačítka neobsahují.

Ve vrchní části okna můžeme vidět generovanou vlnu, jejíž přiblížení lze měnit kolečkem myši. Po najetí na vlnu se nám zobrazí tooltip s rozměry zobrazované vlny. Vizualizaci vlny lze vypnout v záložce „`View`“ ve vrchní části okna. Dále ve vrchní části panelu najdeme tlačítka pro přibližování a oddalování v diagramu, posuvník kontrolující hlasitost, tlačítko pro ztlumení výstupu a nakonec tlačítko play/pause.

Záložka „`File`“ obsahuje tlačítko new, které odstraní všechny panely kromě výstupních z diagramu. Tlačítko „`Save`“ slouží k uložení diagramu. K načtení diagramu slouží tlačítko „`Load`“. Načtením diagramu se smaže současný diagram. Do budoucna by bylo dobré udělat i načítání, které současný diagram nevymaže. Poslední tlačítko „`Change output audio format`“ mění výstupní audio formát diagramu.

Zbývá záložka „`Record`“ se zaškrtávacími políčky „`Record to player`“ a „`Record to file`“, které určují, zda-li se výsledek nahraje do přehrávače, respektive do souboru. Můžeme zaškrtnout i obě políčka zároveň. „`Record instantly`“ a „`Record in real-time`“ odpovídají popisu v sekci 4.4.7. Jen připomeneme, že nahrávání v reálném čase můžeme ukončit opětovným kliknutím na zaškrtávací políčko „`Record in real-time`“. Záložka „`Set record info`“, kterou najdeme v obrázku 5.10, obsahuje políčko pro nastavení délky nahrávky. V případě instantního nahrávání určuje délku nahrávky a v případě real-time nahrávání určuje maximální možnou délku nahrávky. Pod nastavením času najdeme zaškrtávací políčko určující, zda-li máme převést audio formát do výstupního formátu přehrávače, což samozřejmě neovlivňuje nahrávání do souboru. Spodní tlačítko „`Choose record file`“ udává jméno souboru, do kterého nahrávku uložíme. Parametry zůstanou uloženy po stisknutí křížku, či „`CLOSE`“.



Obrázek 5.10: Dialog pro nastavení parametrů nahrávání

5.8 Pluginy

Popíšeme psaní pluginů pro jednotlivé části, tj. jaká rozhraní je nutné implementovat a kam se vkládají. Zásuvné moduly se řeší formou .class souborů a vkládají se do adresáře Diasynth-plugins, respektive do jeho podadresářů. O jaké podadresáře se jedná zmíníme vždy na začátku popisu jednotlivých částí. V současné verzi musíme do adresáře vložit nejen samotný plugin, ale i celou adresářovou strukturu odpovídající jeho cestě v rámci Java balíčků. Například pro plugin pack.age.plugin musíme do adresáře Diasynth-plugins vložit pack/age/plugin.class.

Původně byl způsob načítání zásuvných modulů úplně jiný, ale kvůli výskytu jistých implementačních problémů velice blízko k datu odevzdání jsme se museli uchýlit k současnému provizornímu řešení.

5.8.1 Analyzátor

Pluginy musí být v balíčku `analyzer.plugin.plugins`. Tedy je musíme vložit do adresáře Diasynth-plugins/str/rad/analyzer/plugin/plugins.

Rozhraní zásuvných modulů analyzátoru jsou `AnalyzerDoublePluginIFace`, `AnalyzerBytePluginIFace`, `AnalyzerIntPluginIFace`, které se od sebe liší formátem, ve kterém je vstupní vlna.

```
public interface AnalyzerDoublePluginIFace extends
                                            AnalyzerBasePluginIFace {
        /**
         * @return Returns pair.
         * First value is the name which will be showed on left.
         * Second value is the analyzed value converted to string.
         */
        Pair<String, String> analyze(DoubleWave wave);
}
```

Zbylá dvě rozhraní se liší pouze jménem rozhraní a signaturou metody `analyze`. Návratový typ také zůstává stejný.

```
// AnalyzerBytePluginIFace
analyze(byte[] samples, int numberOfChannels, int sampleSize,
        int sampleRate, boolean isBigEndian, boolean isSigned);

// AnalyzerIntPluginIFace
analyze(int[] samples, int numberOfChannels, int sampleRate);
```

Všechny tři rozšiřují rozhraní `AnalyzerBasePluginIFace`, které vypadá následovně.

```
public interface AnalyzerBasePluginIFace {
        /**
         * Returns the name of the checkbox, which will be shown
         * to user in analyzer panel.
```

```
         */
        String getName();

        /**
         * Returns the tooltip for the checkbox in analyzer panel.
         */
        String getTooltip();
}
```

Balíček `analyzer.plugin.ifaces` obsahuje všechna výše zmíněná rozhraní.

5.8.2 Přehrávač

Pluginy musí být v balíčku `player.plugin.plugins`. Tedy je musíme vložit do adresáře Diasynth-plugins/str/rad/player/plugin/plugins.

Rozhraní, které je nutné implementovat, závisí na typu implementované operace. Pro operaci pracující se dvěma vlnami je nutné implementovat rozhraní `OperationOnWavesPluginIFace`, které nalezneme v balíčku `player.plugin.ifaces.user.waves`.

Pro operaci pracující na jedné vlně je nutné poskytnout třídu implementující rozhraní `OperationOnWavePluginIFace`, které nalezneme v balíčku `player.plugin.ifaces.user.wave`.

Obě rozhraní rozšiřují rozhraní `AudioPlayerJMenuPluginIFace` a liší se od sebe pouze metodou provádějící operaci. Pro operaci s jednou vstupní vlnou to je následující metoda.

```
void performOperation(DoubleWave audio, int startIndex, int endIndex);
```

Pro operaci se dvěma vstupními vlnami se jedná o tuto metodu.

```
void performOperation(DoubleWave input, DoubleWave output,
                      int inputStartIndex, int inputEndIndex,
                      int outputStartIndex, int outputEndIndex);
```

Rozhraní `AudioPlayerJMenuPluginIFace` vypadá následovně.

```
public interface AudioPlayerJMenuPluginIFace extends PluginBaseIFace {
        /**
        * @return Returns tooltip which will be shown when hovering
        * over the button which will perform the operation.
        */
        String getPluginTooltip();
}

public interface PluginBaseIFace {
        boolean shouldWaitForParametersFromUser();
        boolean isUsingPanelCreatedFromAnnotations();
        String getPluginName();
}
```

Pokud plugin nepřijímá vstup od uživatele, vrací metoda `shouldWaitForParametersFromUser` „false“. Metoda `isUsingPanelCreatedFromAnnotations` je důležitá pouze, pokud předchozí metoda vrací „true“. Pokud metoda `isUsingPanelCreatedFromAnnotations` vrací „true“, pak se vytváří panel z anotovaných položek třídy. Pokud vrací „false“, měl by implementovaný plugin dědit od třídy `JPanel`, neboť se zobrazí v dialogu. Pro většinu zásuvných modulů ovšem stačí panel vytvořený z anotací.

Nyní se podíváme, jak funguje `JPanel` vytvořený pomocí anotací. Program projde třídu reprezentující zásuvný modul a najde položky s anotací `plugin.PluginParameterAnnotation`. Slouží pouze pro anotaci primitivních typů a vypadá následovně.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface PluginParameterAnnotation {
        public static final String UNDEFINED_VAL = "UNDEFINED";

        /**
         * Name which will be shown on the GUI.
         * If not set, then the name of the field will be used.
         */
        public String name() default UNDEFINED_VAL;
        public String lowerBound() default UNDEFINED_VAL;
        public String upperBound() default UNDEFINED_VAL;
        public String defaultValue() default UNDEFINED_VAL;
        public String parameterTooltip() default "";
}
```

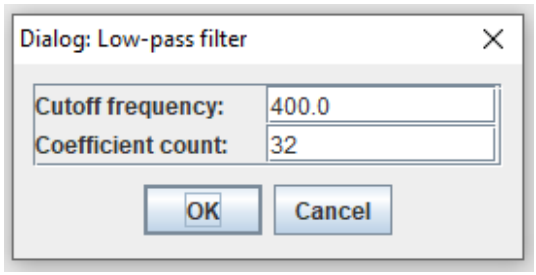
`defaultValue` se nehodí používat, pokud chceme zachovat stav proměnné mezi několika voláními pluginu. V takovém případě je nutné nastavit výchozí hodnotu běžným přiřazením do proměnné.

Nyní se podíváme na příklady. Pro ušetření místa a větší přehlednost změníme řetězce uložené v parametru `parameterTooltip`. Pro operaci filtru ve třídě `player.operations.wave.filters.LowPassFilter` máme položky anotovány následovně.

```
@PluginParameterAnnotation(name = "Cutoff frequency:",
                           lowerBound = "0", defaultValue = "400",
                           parameterTooltip = "Cut-off frequency")
private double cutoffFreq;
@PluginParameterAnnotation(name = "Coefficient count:",
                           lowerBound = "32", defaultValue = "32",
                           parameterTooltip = "Represents the number
                              of the coefficients used for filtering"
private int coefCount;
```

Na obrázku 5.12 můžeme vidět, jak vypadá vytvořený dialog.

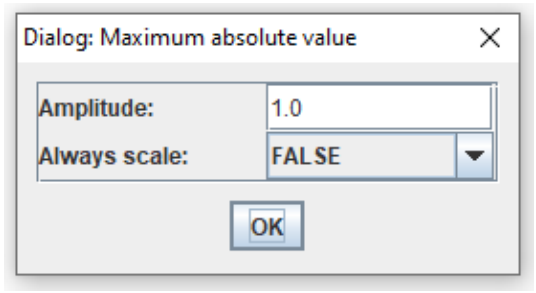
Dalším příkladem je třída `DiagramSynthPackage.Synth.OutputUnit`, která reprezentuje výstupní výpočetní jednotku diagramu.



Obrázek 5.11: Dialog vytvořený před provedením low-pass filtru

```
@PluginParameterAnnotation(name = "Amplitude:", lowerBound = "0",
                           upperBound = "1",
                           parameterTooltip = "Maximum absolute value
                                               allowed in output")
private double maxAbsoluteValue = 1;
@PluginParameterAnnotation(name = "Always scale:",
                           parameterTooltip = "<html>Multiline<br>" +
                                              "tooltip</html>")
private boolean shouldAlwaysSetToMaxAbs = false;
```

Na obrázku 5.12 můžeme vidět, jak vypadá vytvořený dialog.



Obrázek 5.12: Dialog vytvořený v rámci `OutputUnit` generátoru

Pokud chceme použít výčtový typ jako vstup od uživatele, je nutné, aby třída implementovala rozhraní `EnumWrapperForAnnotationPanelIFace`, které má za úkol měnit hodnotu položky výčtového typu uvnitř třídy.

```
/**
* Wrapper interface for the enums.
* FieldName is there to * identify the enum in class.
(There may be more enums in class)
*/
public interface EnumWrapperForAnnotationPanelIFace {
        String[] getEnumsToStrings(String fieldName);
        void setEnumValue(String value, String fieldName);
        void setEnumValueToDefault(String fieldName);
        String getDefaultEnumString(String fieldName);
        String getToolTipForComboBox(String fieldName);
}
```

Metoda `getEnumsToStrings` vrací pole řetězců. Řetězce jsou následně použity v kombinovaném poli (anglicky combobox). Řetězce mohou být libovolné, nemusí se jednat přímo o hodnoty výčtového typu převedené do řetězců, akorát se pak tento fakt musí zohlednit i v metodě `setEnumValue`. Parametr „*value*“ metody `setEnumValue` odpovídá zvolené hodnotě v kombinovaném poli, tj. je to prvek pole získaného z metody `getEnumsToStrings`. `getDefaultEnumString` vrací výchozí řetězec pro kombinované pole.

Příkladem implementace výše zmíněného rozhraní je třída `AlignmentOnWavesOperation`:

```
@PluginParameterAnnotation(name = "Length alignment:",
        parameterTooltip = "The enum which value tells " +
                           "what alignment should be done. " +
                           "Only changes the end indices, " +
                           "not the start ones")
private AlignmentEnum lengthAlignment = AlignmentEnum.NO_ALIGNMENT;


@Override
public String[] getEnumsToStrings(String fieldName) {
    if("lengthAlignment".equals(fieldName)) {
        return AlignmentEnum.getEnumsToStrings();
    }
    return null;
}

@Override
public void setEnumValue(String value, String fieldName) {
    if("lengthAlignment".equals(fieldName)) {
        lengthAlignment = AlignmentEnum.
                                      convertStringToEnumValue(value);
    }
}

@Override
public void setEnumValueToDefault(String fieldName) {
    setEnumValue(getDefaultEnumString(fieldName), fieldName);
}

@Override
public String getDefaultEnumString(String fieldName) {
    if("lengthAlignment".equals(fieldName)) {
        int index = getDefaultIndex(fieldName);
        return AlignmentEnum.getEnumsToStrings()[index];
    }
    return "";
}

private int getDefaultIndex(String fieldName) {
    if("lengthAlignment".equals(fieldName)) {
        return 0;
```

```
    }
    return -1;
}

@Override
public String getToolTipForComboBox(String fieldName) {
    if("lengthAlignment".equals(fieldName)) {
        return "<html>" +
        "NO_ALIGNMENT means that if the output is longer then <br>" +
        "the input will be used more times to fill the output wave." +
        "<br>Other options are self-explaining." +
        "</html>";
    }
    return "";
}
```

Třída `AlignmentEnum`, respektive její pro ukázku důležitá část, vypadá následovně:

```
NO_ALIGNMENT,
ALIGN_TO_SHORTER,
ALIGN_TO_LONGER,
ALIGN_TO_INPUT,
ALIGN_TO_OUTPUT


public static String[] getEnumsToStrings() {
        AlignmentEnum[] values = AlignmentEnum.values();
        String[] strings = new String[values.length];
        for (int i = 0; i < values.length; i++) {
                strings[i] = values[i].toString();
        }

        return strings;
}


public static AlignmentEnum convertStringToEnumValue(String s) {
        AlignmentEnum[] values = AlignmentEnum.values();
        for (AlignmentEnum v : values) {
                if(v.toString().equals(s)) {
                        return v;
                }
        }

        return null;
}
```

5.8.3 Syntezátor

Pluginy musí být v balíčku `synthesizer.synth`. Tedy je musíme vložit do adresáře Diasynth-plugins/str/rad/synthesizer/synth.

Všechny výpočetní jednotky diagramu dědí od třídy `Unit`, ale při psaní pluginu chceme typicky dědit od jedné ze specializovaných tříd. Pro generátory je specializovanou třídou `Generator`, respektive třída `GeneratorNoPhase`, pokud nechceme implementovat algoritmus obsahující fázi.

Ovšem třída `GeneratorNoPhase` i tak potřebuje algoritmus pro frekvenční modulaci, který typicky používá fázi, takže dává smysl dědit od třídy `Generator` a generátor bez fáze implementovat jako potomka varianty s fází, například takto:

```
public class SineGenerator extends Generator {
    public SineGenerator(Unit u) { super(u); }
    public SineGenerator(DiagramPanel panelWithUnits) {
        super(panelWithUnits);
    }

    @Override
    public String getDefaultPanelName() {
        return "Sine";
    }

    @Override
    public void resetToDefaultState() {
        // EMPTY
    }


    @Override
    public double generateSampleConst(double timeInSecs,
                                      int diagramFrequency,
                                      double amp, double freq,
                                      double phase) {
        double genVal;
        genVal = amp * Math.sin(freqToRad(freq) * timeInSecs + phase);
        return genVal;
    }

    @Override
    public String getTooltip() {
        return "Generates sine wave";
    }

    @Override
    public void copyInternalState(Unit copySource) {
        // EMPTY
    }
}
```

```
public class SineGeneratorNoPhase extends SineGenerator {
    public SineGenerator(Unit u) { super(u); }
```

```
    public SineGenerator(DiagramPanel panelWithUnits) {
        super(panelWithUnits);
    }

    @Override
    protected InputPort[] createInputPorts(DiagramPanel panelWithUnits,
                                           double[] neutralValues) {
        return Generator.createInputPorts(this, panelWithUnits,
                                          neutralValues);
    }
}
```

Operátory dědí od abstraktní třídy `Operator`, respektive od abstraktní třídy s odpovídajícím počtem vstupů, tj. `UnaryOperator`, `BinaryOperator` a tak dále. V programu momentálně nalezneme pouze `UnaryOperator` a `BinaryOperator`, neboť v rámci práce nebyl použit žádný ternární operátor. Pro naprogramování abstraktní třídy představující ternární, resp. n-ární operátor se stačí podívat do třídy `BinaryOperator` a změnit metody závisející na počtu vstupů.

Pokud chceme implementovat obálku, pak by měl zásuvný modul dědit od třídy `Envelope`.

Konečně pokud chceme naprogramovat noise generátor, měl by plugin dědit od třídy `NoiseGeneratorNoFreq`, respektive `NoiseGenerator`. Typicky implementujeme `NoiseGeneratorNoFreq` pomocí `NoiseGenerator` podobně jako u generátorů.

Dědění od specializovaných potomků třídy `Unit` nás ušetří od programování určitých metod, například `calculateSamples`, která vezme pole hodnot z jednotlivých vstupů a postupně na ně provede operaci dané výpočetní jednotky. Díky tomu nám při dědění od specializované třídy stačí implementovat pouze operaci, kterou metoda `calculateSamples` n-krát zavolá pro výpočet n výstupních samplů, kde n je velikost pole na vstupu, což je v našem případě 512.

Nyní se podíváme na jednotlivé metody třídy `Unit`, které je nutné implementovat, pokud vytváříme novou výpočetní jednotku. Zmíníme se pouze o metodách, které je nutné implementovat při dědění od nějaké ze specializovaných tříd, protože mě nenapadá důvod, kvůli kterému by bylo dobré dědit přímo od třídy `Unit`. Pokud z nějakého důvodu chceme dědit přímo od třídy Unit, můžeme se podívat do zdrojových kódů na implementace jednotlivých specializovaných abstraktních tříd a podle nich třídu naprogramovat.

Všichni potomci musí implementovat oba konstruktory. Typicky se v konstruktoru pouze zavolá odpovídající konstruktor rodičovské třídy, tj. konstrukce super(parametry). Zavolání rodičovských konstruktorů pomocí konstrukce super je nutné pro správné chování třídy.

```
/**
* Copy constructor.
* Has to be implemented in all deriving classes.
*/
public Unit(Unit u) {
        // CODE
}
```

```
/**
* Constructor used when not copying.
* Has to be implemented in all deriving classes.
*/
public Unit(DiagramPanel panelWithUnits) {
        // CODE
}
```

Občas dává smysl provést i nějakou další akci v copy konstruktoru, například zkopírovat položky třídy. Stejné funkcionality je možné dosáhnout i přepsáním metody `copyPanel`, ale změna konstruktoru mi přijde logičtější. Metoda `copyPanel` interně nevolá metodu `copyInternalState`, neboť to může být v určitých situacích na škodu.

Dále je nutné implementovat následující metodu, neboť její návratová hodnota se použije pro vytvoření unikátního jména, které bude vidět v iterátoru. Jméno můžeme zobrazit i na panelu, pokud se tak rozhodneme.

```
public abstract String getDefaultPanelName();
```

Dále je občas nutné implementovat metodu, která vytvoří vstupní porty. Typy implementovaných portů najdeme v balíčku `synthesizer.gui.diagram.panels.port.ports`.

```
/**
* Should return InputPort[] which will
* be set as unit's input ports.
* It is called inside the constructors.
* If there are no input ports then set it to new InputPort[0].
*/
protected abstract InputPort[] createInputPorts(
                                        DiagramPanel panelWithUnits,
                                        double[] neutralValues);
```

Následující metodu dostatečně vysvětluje komentář.

```
/**
* It is used as parameter to the createInputPorts method.
* It is up to the person who implements the plugin, if
* he takes these values into consideration. He may ignore them and
* just set the input ports to hard-coded default values in
* createInputPorts method.
*
* @return Returns neutral values for ports.
*         If null or if the array is shorter than number of ports
*         then these values should be ignored.
*/
public abstract double[] getNeutralValuesForPorts();
```

Následující dvě metody jsou implementované ve specializovaných abstraktních třídách, ale občas se může hodit, je přepsat. Tyto metody vrátí tvar panelu, který se použije pro vykreslování výpočetní jednotky. Implementované tvary najdeme v balíčku `synthesizer.gui.diagram.panels.shape`.

```
/**
* Creates new shaped panel
*/
protected abstract ShapedPanel createShapedPanel(
                                    DiagramPanel panelWithUnits);

/**
* Creates new shaped panel called with
* corresponding constructor of same signature as this method
* (+ the internals of course)
*/
protected abstract ShapedPanel createShapedPanel(int relativeX,
                                   int relativeY, int w, int h,
                                   DiagramPanel panelWithUnits);
```

Příklad implementace ze třídy Generator:

```
@Override
protected ShapedPanel createShapedPanel(DiagramPanel panelWithUnits) {
        ShapedPanel sp = new ArcShapedPanel(panelWithUnits,
                          new ArcConstantTextInternals(getPanelName()),
                          this);
        return sp;
}

@Override
protected ShapedPanel createShapedPanel(int relativeX, int relativeY,
                                        int w, int h,
                                        DiagramPanel panelWithUnits) {
        ShapedPanel sp = new ArcShapedPanel(relativeX, relativeY,
                          w, h, panelWithUnits,
                          new ArcConstantTextInternals(getPanelName()),
                          this);
        return sp;
}
```

Dále je nutné implementovat metodu, která resetuje stav výpočetní jednotky do stavu před přehráním prvního samplu. Pokud je jednotka bezstavová, pak je tělo této metody prázdné. Příkladem operátoru, který není bezstavový, je filtr.

```
/**
* Resets to the default state
* (as if no sample was ever before played)
*/
public abstract void resetToDefaultState();
```

Když už mluvíme o stavu, tak musíme zmínit metodu `copyInternalState`, která zkopíruje vnitřní stav jednotky v parametru do sebe. O této metodě jsme se bavili v sekci 4.4.6.

Dále je nutné nějakým způsobem předat uživateli, který pracuje v uživatelském rozhraní s panely, informace o panelu. K tomu slouží následující metoda, která vrátí řetězec, jenž se po najetí myší na panel zobrazí.

```
public abstract String getTooltip();
```

Někdy je nutné implementovat následující metod. Tyto metody existují, aby mohla být na výstupním panelu diagramu, nebo přímo na operátoru provádějící normalizaci, provedena normalizace. Vrací minimální, maximální a maximální absolutní hodnotu na výstupu výpočetní jednotky. Metodu `getMaxAbsValue` není nutné implementovat znovu.

```
@Override
public double getMaxAbsValue() {
        double min = getMinValue();
        double max = getMaxValue();
        return Math.max(Math.abs(min), Math.abs(max));
}

@Override
public abstract double getMinValue();
@Override
public abstract double getMaxValue();
```

Pokud chceme do výpočetní jednotky přidat „`properties`“, učiníme tak podobně, jako když implementujeme plugin do přehrávače. Stačí napsat třídu implementující rozhraní `PluginBaseIFace` a v metodě `setPropertiesPanel` nastavit instanci takové třídy do proměnné „*propertiesPanel*“, viz následující kód.

```
// Unit.java
protected abstract void setPropertiesPanel();

// OutputUnit.java
@Override
protected void setPropertiesPanel() {
        propertiesPanel = this;
}
```

Pokud nechceme properties implementovat, pak „*propertiesPanel*“ nastavíme na „`null`“.

Nyní se podíváme, jaké metody musíme implementovat, pokud dědíme od nějaké ze specializovaných tříd. Pro noise generátory stačí naprogramovat následující metody.

```
/**
* Generates noise between 0 and 1.
```

```
* @return
*/
public abstract double generateNoise();
public abstract String getDefaultPanelName();
public abstract String getTooltip();
```

Podobně to vypadá i při dědění od některé z ostatních specializovaných tříd. Vždy implementujeme `getTooltip` a `getDefaultPanelName` a pak metodu, která generuje výstupní sample pro dané vstupy. Pro `Generator` je touto metodou `generateSampleConst`, pro `UnaryOperator` `unaryOperation`, pro `BinaryOperator` `binaryOperation`, pro `Envelope` `generateEnvelopeSample` a pro `NoiseGenerator` je to výše zmíněná metoda `generateNoise`. Občas je navíc nutné vytvořit tvar panelu pomocí `createShapedPanel`.

Kromě těchto metod je nutné ještě implementovat dva konstruktory, jak již bylo zmíněno výše.

Většina věcí ohledně pluginů by měla být nyní jasná. Pro pochopení detailů se můžeme podívat na implementované výpočetní jednotky uvnitř balíčku `synthesizer.synth`.

6. Související práce

V této kapitole se společně podíváme na nejznámější konkurenční programy.

Nejprve projdeme programy umožňující prohlížení zvukových vln (6.1) a to přesněji na programy Audacity a Adobe Audition. Přehrávače umožňující pouze přehrávání audia nemá smysl zmiňovat.

Následně probereme různé syntezátory (6.2). Nejprve ty umožňující syntézu pomocí diagramu s uživatelským rozhraním a to Max/MSP a Pure Data. Poté krátce prozkoumáme programovací jazyk pro syntézu zvuku Csound, který ovšem neobsahuje grafické uživatelské rozhraní. Posledním programem, na který se podíváme, je FL studio, které sice umožňuje syntézu, ale ne formou diagramů. Kapitolu zakončíme sekcí Výjimečnost programu Diasynth, kde zmíníme hlavní výhody programu Diasynth a v čem se liší od konkurence.

Analyzátor nemá moc smysl uvádět do kontextu ostatních programů, protože pokud chceme analyzovat pouze zvukové soubory, budou všechny takové programy vypadat podobně. Respektive jsou dvě možnosti. První možností je ta naše, tj. vybereme soubory, na které provedeme analýzu a výsledky uložíme. Druhou možností je provádět analýzu přímo na stopy v přehrávači, jak to například řeší Audacity. Druhé řešení můžeme v budoucnu přidat i do programu Diasynth.

6.1 Přehrávače

6.1.1 Audacity

Audacity je bezplatný open-source projekt vyvíjený po řadu let mnoha programátory, proto je nemožné se mu touto prací kvalitou vyrovnat ať už stabilitou, či množstvím funkcí. I přesto se náš program konkurenčnímu Audacity v určitých ohledech vyrovná a v některých ho možná i předčí. Například změna výšky vln probíhá v programu Diasynth rychleji. Mixování je dle mého názoru o něco intuitivnější. Přidávání nových vln do programu je mnohem snazší. Diasynth navíc zobrazuje tooltip s informace o jednotlivých samplech po ukázání na vlnu. Tooltipy jsou kvůli minimalizaci počtu alokovaných řetězců řešeny pomocí třídy StringBuilder. Samozřejmě program Audacity má oproti našemu programu spoustu výhod a vyjmenování všech by bylo na dlouho, ale je dobré vědět, že přeci jen má program Diasynth nějaké výhody oproti konkurenci. Na podrobnější porovnání se podíváme v tabulce 6.1.

Odkaz na stránky: `https://www.audacityteam.org/`

Tabulka 6.1: Porovnání přehrávačů

	Diasynth	**Audacity**
Přehrávání	✓	✓
Zobrazení vln	✓	✓
Roztažení vln	✓	✓
Přiblížení vln	✓	✓
Mixování vln	✓ Posuvník pro každý kanál	✓ Jeden posuvník tzv. panning
Modifikace vln	✓ Pár operací	✓ Mnoho operací
Přidání operací přes zásuvné moduly	✓	✓
Měřič intenzity zvuku	✓	✓
FFT	✓ Bez oken Experimentální	✓ S okny
IFFT Kreslení do výsledku FFT	✓ Experimentální	X
Modifikace jednotlivých vzorků	✓ Lze ale složitě	✓
Kreslení zvukové vlny	✓	X Lze přes modifikaci jednotlivých samplů
Spektrogram	X	✓
Uložení stop	Neefektivně Celé v paměti	Efektivně Streamování ze souborů
Tooltip u vlny	✓ Informace o samplu	X
Operace undo	X	✓
Funkcionalita	Základní	Pokročilá

6.1.2 Adobe Audition

Umožňuje především mixování zvuku, přidávání zvukových efektů a analýzu pomocí spektrogramu. Jelikož se na rozdíl od Audacity jedná o placený software, nebudeme programy porovnávat. Program sice obsahuje sedmidenní zkušební verzi, ale časová omezenost neumožňuje snadno zkontrolovat korektnost v textu napsaných informací po uplynutí zkušební doby.

Odkaz na stránky: `https://www.adobe.com/cz/products/audition.html`

6.2 Syntezátory

6.2.1 Max/MSP a Pure Data

Max/MSP je placený software, Pure Data je jeho bezplatná alternativa.

Stejně jako v případě přehrávačů se zaměříme na bezplatnou variantu. Hned po spuštění programu Pure Data je vidět, že program není moc intuitivní. Vůbec není jasné, jakým způsobem máme přidat prvek do diagramu. Když na to konečně přijdeme, tak díky absenci menu nevíme, jakým způsobem můžeme vytvořit jednotlivé generátory. Navíc samotné posouvání a připojování panelů je dle mého názoru v programu Diasynth vyřešeno lépe. Na druhou stranu toho program Pure Data umí mnohem více. Zjišťování funkcionality a to i té základní ovšem zabere spoustu času, což může začátečníka rychle odradit. Naproti tomu v programu Diasynth je hned jasné, co a jak dělat. Ať už se jedná o menu s výběrem panelů, či nápovědy pomocí tooltipů, nebo označení portů popisky, nebo odlišení různých druhů panelů tvarem a vnitřnostmi. Porovnání určitých aspektů můžeme vidět v tabulce 6.2.

Odkaz na stránky (Max/MSP): `https://cycling74.com/`
Odkaz na stránky (Pure Data): `https://puredata.info/`

Tabulka 6.2: Porovnání syntezátorů

	Diasynth	**Pure Data**
Menu s komponentami	intuitivní	neintuitivní a hůře přístupné
Ovládání	Snadné	Složité
Generátory a operátory	Menší počet (Pouze základní)	Větší počet
Funkcionalita	Základní	Pokročilá
LADSPA pluginy	X (Možná v budoucnu)	✓
Vlastní pluginy	✓ (Jednoduše)	✓ (Složitě)

LADSPA plugin je standard, který umožňuje zapojení softwarových audio procesorů a efektů do velkého množství balíčků se syntézou zvuku a nahráváním (přepis úvodního odstavce z oficiální stránky: `https://www.ladspa.org/`). Někdy v budoucnu se podívám, jestli by tento standard nebylo možné napasovat na program Diasynth. Momentálně o tomto standardu nevím dostatečně na to, abych dokázal říct, že to bude možné. Bohužel bude nutné překonat několik překážek například, že je tento standard psaný v jazyce C.

6.2.2 Csound

Pro zajímavost se podíváme na programovací jazyk CSound určený pro generování hudby.

První verze jazyka vznikla v roce 1985. Generování zvuku probíhá na základě dvou typů souborů. Prvním typem souboru je *orchestra*. Udává struktury oscilátorů. Druhým typem je *score* popisující, jak se mají tyto nástroje chovat v čase.

Tyto a další informace najdeme na stránce Wikipedia [11] a i na oficiálních stránkách [3]. CSound má oproti programu Diasynth nevýhodu v tom, že musíme psát kód. Ovšem díky tomu je velice snadná změna hodnot parametrů v čase. Té můžeme v současné verzi programu Diasynth dosáhnout také, ale mnohem složitěji. Psaní kódu navíc výrazně zjednodušuje tvorbu složitých diagramů.

Odkaz na stránky: `https://csound.com/`

6.2.3 FL studio

FL studio poskytuje kromě možnosti mixování na velice vysoké úrovni i syntézu. Syntéza má ale spíše blíže k reálným syntezátorům. Ať už se jedná o ovládání, či samotné možnosti syntézy. Z toho důvodu se tento program hodí především pro uživatele, kteří mají zkušenosti se skutečnými syntezátory. Na druhou stranu syntéza pomocí diagramů je mnohem silnější a umožňuje generovat prakticky jakýkoliv zvuk.

Odkaz na stránky: `https://www.image-line.com/flstudio/`

6.3 Výjimečnost programu Diasynth

Práce má hned tři vlastnosti, které jí odlišují od většiny podobných programů. Práce je napsaná v jazyce Java. Už toto jí odlišuje od naprosté většiny konkurenčního softwaru psaného v jazycích C/C++. To jsme již probrali v sekcích 3.1 a 3.2.

Většina programů poskytuje pouze částečnou funkcionalitu oproti naší práci. Například Audacity obsahuje přehrávač a umožňuje analýzu, ale už neobsahuje možnost generovat audio pomocí diagramu. Zatímco program Pure Data umožňuje syntézu pomocí diagramů, ale zase postrádá zbylé dvě části. Lze namítnout, že to nevadí, neboť spojením výše zmíněných programů dostaneme nejen stejnou funkcionalitu, jakou má Diasynth, ale dokonce jí vysoce převýšíme. V tom se ovšem skrývá další problém. Například program Pure Data může obsáhlou funkcionalitou, složitostí a ne úplně intuitivním ovládáním nového uživatele odradit. Audacity je na tom mnohem lépe, ale i zde je občas složité najít to, co zrovna potřebujeme. Program Diasynth jde opačným směrem. Orientace v programu a jeho ovládání jsou snadné, ale program poskytuje pouze základní funkcionalitu. Omezená funkcionalita je ovšem spíše daná nedostatkem času, nikoliv tím, že jí není možné do programu přidat.

Poslední a asi nejdůležitější výhodou je rozšiřitelnost funkcionality formou pluginů. Stačí pouze implementovat rozhraní a vložit .class soubor do správného adresáře, viz sekce 5.8. Rozhraní pro pluginy jsou navržena tak, aby uživatel musel naprogramovat jen to nejnutnější a byl oproštěn od všeho nesouvisejícího s programováním audio algoritmů, například od nutnosti psát si vlastní uživatelské rozhraní pro zadání vstupních dat algoritmu.

Závěr

Výsledný software splňuje všechny cíle vytyčené v zadání práce.

Mezi hlavní výhody patří již mnohokrát zmiňovaná rozšiřitelnost programu pomocí zásuvných modulů a vysoká stabilita programu, alespoň na mém systému. Dalšími výhodami jsou možnost velice snadného sestavení diagramu pro generování zvuku. Navíc můžeme v rámci pár kliknutí dosáhnout uložení generované vlny ať už do souboru, či přehrávače, ve kterém si lze generovanou vlnu okamžitě prohlédnout. Část s analyzátorem umožňuje snadné ovládání a poskytuje v zadání definovanou funkcionalitu. V neposlední řadě program Diasynth umožňuje provést operace modifikující načtené stopy v přehrávači. Načtené vlny si lze zblízka prohlédnout. Vlny můžeme mixovat dohromady a výsledek mixování uložit do souboru, či přehrát. Přítomnost přehrávače s možností vkládání vln přímo ze syntezátoru navíc maximálně zkrátí počet kroků, které je nutné provést při přechodu mezi syntezátorem a programem pro úpravu zvuku.

Program není ani zdaleka perfektní, ale vzhledem k rozsahu práce a faktu, že se jednalo o můj první netriviální projekt je výsledek velice dobrý a to i přes určitá ne úplně nejlepší rozhodnutí v rámci práce.

Nyní se krátce zamyslíme nad tím, co by bylo dobré do budoucna v práci změnit, respektive do ní přidat, aby se dala označit za kompletní.

- Opravit zbývající chyby v uživatelském rozhraní. V rámci testování byla nalezena jediná chyba. Tato chyba se nachází v přehrávači a spočívá ve špatném nastavení velikosti vlny. Uživatel jí může spravit nastavením všech vln na minimální výšku. Chyba téměř nenastává, proto se mi jí zatím nepodařilo lokalizovat.

- Přidat undo funkci do přehrávače. Tato funkce vrátí stopy do stavu před poslední provedenou operací.

- Přidat další možnosti pluginování do přehrávače, především obecnou variantu pluginu se dvěma vstupními vlnami.

- Změnit způsob zadávání vstupních vln a výstupní vlny pro operace v přehrávači.

- Mimo jiné by bylo dobré umožnit uživateli otevřít generované vlny v programu Audacity, pokud jej uživatel preferuje.

- Změnit načítání zásuvných modulů. Současný přístup sice funguje, ale není ideální. Změna zahrnuje odstranění nutnosti vkládat plugin do adresáře spolu se svojí adresářovou strukturou odpovídající balíčkům v jazyce Java.

- Realizovat načítání audio stop streamováním souborů do bufferů, místo udržování celých zvukových stop v paměti.

Po implementaci výše zmíněných existuje stále spousta dalších rozšíření práce, například:

- Přidat další generátory a operátory do syntezátoru a to především filtry a delay. Delay jednotka čeká n sekund, než pošle svůj vstup na výstup.

- Přidat komponentu vypínač, díky níž bude možné určovat, kdy má daný generátor generovat výstup a kdy ne. Tato komponenta umožní skládání hudby, neboť pak se generátory mohou chovat jako klávesy na klasickém syntezátoru. Komponenta může fungovat například tak, že čte ze souboru události v čase, které budou určovat zapnutí a vypnutí.

- Přidat vytváření diagramů z textových souborů. Textové soubory budou obsahovat kód popisující diagramy. Kód by byl zřejmě podobný jazyku CSound.

- Umožnit zadávání funkce do operátoru waveshaper pomocí textu, například jako $y = 2 \cdot x$.

- A stovky dalších věcí

Seznam použité literatury

[1] Beat detection algorithm. https://www.clear.rice.edu/elec301/Projects01/beat_sync/beatalgo.html, 2001. Navštíveno 26.11.2020.

[2] Beat detection algorithms. http://archive.gamedev.net/archive/reference/programming/features/beatdetection/index.html, 2011. Navštíveno 05.07.2020.

[3] Csound syntax. https://csound.com/get-started.html, 2020. Navštíveno 10.07.2020.

[4] T. A. J. Charles Dodge. *Computer Music: Synthesis, Composition, and Performance*. 2nd Edition. Schirmer/Thomson Learning, 1997.

[5] J. M. Chowning. The synthesis of complex audio spectra by means of frequency modulation. *Journal of the Audio Engineering Society*, 21(7):1, 1973. Digitální verzi lze najít zde https://ccrma.stanford.edu/sites/default/files/user/jc/fm_synthesispaper-2.pdf.

[6] R. G. Lyons. *Understanding Digital Signal Processing*. 1st Edition. Prentice Hall, 1996.

[7] M. a. T. V. MAREŠ. *Průvodce labyrintem algoritmů*. CZ.NIC, z.s.p.o., Praha, 2017.

[8] Oracle. Class audiofileformat.type. https://docs.oracle.com/javase/7/docs/api/javax/sound/sampled/AudioFileFormat.Type.html, 2020. Navštíveno 04.12.2020.

[9] Oracle. Java audio format enconding. https://docs.oracle.com/javase/7/docs/api/javax/sound/sampled/AudioFormat.Encoding.html, 2020. Navštíveno 25.11.2020.

[10] Wikipedia. Amplitude. https://en.wikipedia.org/wiki/Amplitude, 2020. Navštíveno 14.07.2020.

[11] Wikipedia. Csound. https://cs.wikipedia.org/wiki/Csound, 2020. Navštíveno 10.07.2020.

[12] Wikipedia. Frequency. https://en.wikipedia.org/wiki/Frequency, 2020. Navštíveno 13.07.2020.

[13] Wikipedia. Pulse-code modulation. https://en.wikipedia.org/wiki/Pulse-code_modulation, 2020. Navštíveno 25.11.2020.

[14] Wikipedia. Root mean square. https://en.wikipedia.org/wiki/Root_mean_square, 2020. Navštíveno 15.07.2020.

[15] Wikipedia. Sawtooth wave. https://en.wikipedia.org/wiki/Sawtooth_wave, 2020.

[16] Wikipedia. Triangle wave. https://en.wikipedia.org/wiki/Triangle_wave, 2020.