МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ТЕХНОЛОГІЙ ТА ДИЗАЙНУ

Факультет мехатроніки та комп'ютерних технологій

Кафедра інформаційних та комп'ютерних технологій

Дипломна магістерська робота

на тему Комп'ютерно-інтегрована система генерації музичних композицій

Виконав: студент групи МгАк-20
спеціальності
151 – Автоматизація та комп'ютерно-інтегровані технології
за освітньою програмою
Комп'ютерно-інтегровані технологічні процеси і виробництва
Назар КУЦЬ
Керівник к.т.н., доц. Юрій ПИЛИПЕНКО
Рецензент д.т.н., проф. Віктор ЧУПРИНКА

Київ - 2021

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ТЕХНОЛОГІЙ ТА ДИЗАЙНУ

Факультет мехатроніки та комп'ютерних технологій

Кафедра інформаційних та комп'ютерних технологій

Спеціальність 151 Автоматизація та комп'ютерно-інтегровані технології

Освітня програма Комп'ютерно-інтегровані технологічні процеси та виробництва

ЗАТВЕРДЖУЮ

Завідувач кафедри інформаційних та комп'ютерних технологій

_____ Наталія ШИБИЦЬКА

"___" _____ 2021 р.

З А В Д А Н Н Я

НА ДИПЛОМНУ МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТУ

Куцю Назару Миколайовичу

1. Тема роботи «Комп'ютерно-інтегрована система генерації музичних композицій».
Науковий керівник роботи Пилипенко Юрій Миколайович, к.фм.н., доц, затверджені наказом вищого навчального закладу від 4 жовтня 2021 року, № 286-уч.
2. Строк подання студентом роботи - 9 грудня 2021 р.
3. Вихідні дані до роботи: набір шаблонів, що зберігається в базі даних проекту та можуть бути застосовані до різних складових композиції. Кожен шаблон має певний тип і впливає на визначену концептуальну складову треку, таку як: ритм, гармонія, інтервальне зміщення та інші.
4. Зміст дипломної роботи (перелік питань, які потрібно розробити): Вступ. Аналіз проблеми – чим може бути корисним застосування часткової генерації в процесі роботи над музичною композицією. Пошук сфери застосування часткової генерації в процесі творчості. Аналіз існуючих рішень, дослідження історії розвитку музичних технологій. Постановка завдань. Проектування дизайну та архітектури програмного забезпечення для розв'язання поставлених завдань. Пошук ефективних стратегій побудови алгоритмів програми для максимізації продуктивності її застосування. Розробка програми. Тестування. Загальні висновки.

5. Консультанти розділів дипломної магістерської роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Вступ	Пилипенко Ю.М., доцент		
Розділ 1	Пилипенко Ю.М., доцент		
Розділ 2	Пилипенко Ю.М., доцент		
Розділ 3	Пилипенко Ю.М., доцент		
Розділ 4	Пилипенко Ю.М., доцент		
Висновки	Пилипенко Ю.М., доцент		

6. Дата видачі завдання 5 жовтня 2021 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного роботи (проекту)	Строк виконання етапів роботи	Примітка
1	Вступ	19.10. 2021	
2	Розділ 1. Постановка завдання та огляд існуючих рішень	26.10.2021	
3	Розділ 2. Хід розробки	09.11.2021	
4	Розділ 3. Опис ключових алгоритмів	16.11.2021	
5	Розділ 4. Огляд результатів та подальший розвиток програми	23.11.2021	
6	Висновки	28.11.2021	
7	Оформлення дипломної магістерської роботи (чистовий варіант)	29.11.2021	
8	Здача дипломної магістерської роботи на кафедру для рецензування (за 14 днів до захисту)	30.11.2021	
9	Перевірка дипломної магістерської роботи на наявність ознак плагіату (за 10 днів до захисту)	6.12.2021	
10	Подання дипломної магістерської роботи на затвердження завідувачу кафедри (за 7 днів до захисту)	9.12.2021	

Студент _____ (підпис) Назар КУЦЬ

Керівник проекту (роботи) _____ (підпис) Юрій ПИЛИПЕНКО

Директор НМЦУПФ _____ (підпис) Олена ГРИГОРЕВСЬКА

АНОТАЦІЯ

Куць Н.М. Комп'ютерно-інтегрована система генерації музичних композицій

Дипломна магістерська робота за спеціальністю 151 – «Автоматизація та комп'ютерно-інтегровані технології», Київський національний університет технологій та дизайну, Київ, 2021 рік.

Дипломна магістерська робота присвячена підвищенню ефективності в процесі роботи над музичною композицією, шляхом застосування програмних алгоритмів, що дозволяють встановлювати та зберігати концептуальні зв'язки між різними складовими композиції: партіями, їх відрізками та конкретними нотами. Прикладами таких зв'язків можуть служити – ритмічна, гармонійна, інтервальна та інші музичні закономірності, що базуються на музичній теорії та її особливостях, що склалися в процесі тривалого історичного розвитку.

Запропоновано визначити концептуальні зв'язки, які можуть існувати між елементами композиції. Створити шляхи зберігання інформації про них та автоматичного застосування збережених зв'язків при внесенні концептуальних змін на рівні партії чи всієї композиції.

Створена програмна модель з набором алгоритмів, здатних частково чи повністю вирішувати проблему зберігання концептуальних зв'язків між елементами композиції та працювати над нею на рівні контексту.

Ключові слова: генеративне мистецтво, програмні алгоритми, теорія музики, музичний контекст, музична композиція, музичне програмне забезпечення, генерація музики.

АННОТАЦИЯ

Куць Н.Н. Компьютерно-интегрированная система генерации музыкальных композиций

Дипломная магистерская работа по специальности 151 – Автоматизация и компьютерно-интегрированные технологии, Киевский национальный университет технологий и дизайна, Киев, 2021 год.

Дипломная магистерская работа посвящена повышению эффективности в процессе работы над музыкальной композицией путем применения программных алгоритмов, позволяющих устанавливать и сохранять концептуальные связи между разными составляющими композиции: партиями, их отрезками и конкретными нотами. Примерами таких связей могут служить ритмическая, гармоническая, интервальная и другие музыкальные закономерности, основанные на музыкальной теории и ее особенностях, сложившихся в процессе длительного исторического развития.

Предлагается определить концептуальные связи, которые могут существовать между элементами композиции. Создать пути хранения информации и автоматического применения сохраненных связей при внесении концептуальных изменений на уровне партии или целой композиции.

Создана программная модель с набором алгоритмов, способных частично или полностью решать проблему хранения концептуальных связей между элементами композиции и работать над ней на уровне контекста.

Ключевые слова: генеративное искусство, программные алгоритмы, теория музыки, музыкальный контекст, музыкальная композиция, музыкальное программное обеспечение, создание музыки.

ANNOTATION

Kutz N.M. Computer-integrated music generation system

Master's thesis in specialty 151 - "Automation and computer-integrated technologies", Kyiv National University of Technology and Design, Kyiv, 2021.

The master's thesis is devoted to increasing the efficiency of working on musical composition, through the use of software algorithms that allow you to establish, and maintain conceptual relationships between different components of the composition: parts, their segments, and specific notes. Examples of such relationships are rhythmic, harmonic, interval, and other musical patterns based on music theory and its features that have developed over the course of long historical development.

It is suggested to define the conceptual relationships that may exist between the elements of the composition. Create ways to store information about them and automatically apply the saved links when making conceptual changes at the level of the party or the whole composition.

A software model has been created with a set of algorithms that can partially or completely solve the problem of preserving the conceptual relationships between the elements of the composition and working on it at the context level.

Keywords: generative art, software algorithms, music theory, musical context, musical composition, musical software, music generation.

ЗМІСТ

ВСТУП

Актуальність обраної теми дослідження. Написання музики, в першу чергу, креативний процес, в ході якого автор розповідає історію, доступними в цій області способами самовираження.

Процес роботи над музичною композицією можна розділити на три етапи:

1. Формування ідеї
2. Пошук форми
3. Реалізація

Етап формування ідеї залежить від багатьох факторів – життєвого досвіду автора, глибини пізнань в певній області, кругозору. Автор може надихнутися життєвою подією, пейзажом, на нього можуть справити враження твори мистецтва, ідеї з художньої літератури. Причин виникнення ідеї може бути безліч.

Після виникнення ідеї автор може відкинути її або реалізувати у зрозумілій для нього формі. Це можна назвати процесом самовираження. Наприклад, скульптор виражає ідеї створюючи об’ємні фігури, шляхом ліплення, висікання, лиття тощо.

Незалежно від обраної форми далі слідує процес реалізації. На цьому етапі автор спирається на доступні для нього способи самовираження, якими він володіє. Мова йде не про сферу діяльності, як живопис, література, хореографія чи інші, а про інструментарій, яким володіє автор у обраній сфері. Наприклад, в музиці – це володіння музичними інструментами, пізнання в різних областях композиції, написанні оркестровок чи електронної музики. Тобто фактичні навики та знання, користуючись якими автор буде доводити ідею до вигляду закінченого твору.

Проблема сучасних підходів до написанням музики, які пропонують більшість музичних програм, полягає в тому, що всі вони спілкуються з

автором мовою конкретних значень, тоді як сам автор в процесі написання музики розмірковує концепціями, тобто зв’язками, які мають більш абстрактну природу.

Пояснимо більш детально, що означають терміни «конкретні значення» та «музичні концепції».

Якщо не занурюватися в деталі, конкретними значеннями в написанні музики можна вважати ноти. Нота, як одиниця музичного матеріалу, слугує позначенню на письмі сукупності фізичних параметрів звуку – частоти коливання, тривалості коливального процесу, його амплітуди. Нота дозволяє передавати цю інформацію у вигляді простого символу без необхідності заглиблюватися в деталі кожного процесу окремо. Таким чином – нота виступає абстракцією фізичних характеристик звукової хвилі[1].

З сукупності нот будуються фрази, з фраз партії, а з партій цілі композиції.

Важливо що зі сторони ідеї, конкретні ноти не мають такого великого значення, як зв’язки, які можуть між ними виникнути. Це те ж саме що значення окремих слів стає зрозумілим лише в контексті речення, а значення речення в контексті тексту. При цьому між словами в реченні виникає зв’язок. І цей зв’язок є важливіше окремих слів.

Повернемося до музики.

Якщо головна мета взаємодії автора музики та програми для її написання – надання закінченої форми ідеям автора, то досконалою така система могла б стати, якби дозволяла працювати не лише на рівні конкретних значень, а й на рівні контексту. Тобто якби з нею можна було спілкуватися не лише мовою конкретних нот (редагуючи їх параметри), а

1 Зауважимо, що нота як символ не передає інформацію про тембральне забарвлення звуку. Тембр залежить від конкретного джерела, яке породжує звукові коливання (наприклад, для гітари це струни та резонатор).

одразу мовою ідей та концепцій – повідомляючи з яких концепцій має складатися композиція на певному часовому відрізку.

Що може слугувати прикладами концепцій в написанні музики. Концепціями в даному випадку ми називаємо певні закономірності в розвитку музичного твору.

За історію розвитку музики, виникла велика кількість таких концепцій. Різні народи знаходили подібні закономірності, частину з них оформлювали в правила (такі як лади, тональності, ритм і подібне), а інші в структури, які простіше можна назвати «побажаннями» (наприклад, послідовності акордів, зв’язки в цих акордах). Кожна з подібних концепцій надає певних характеристик твору. Наприклад, різні послідовності акордів – можуть звучати сумно або весело, гармонійно чи хвилююче. Тобто кожна така складова – є ніби кистю, яка приносить до загальної композиції свої фарби.

Теорія музики виникла не випадково. Це результат тривалих експериментів з відбором та систематизацією різних закономірностей які виникають при різному комбінуванні простих одиниць – музичних звуків, шумів та людського голосу.

Ці комбінування та систематизація продовжуються донині, утворюючи нові музичні жанри та напрями.

Цікаво, що хоча більшість таких закономірностей виникала хаотично – в різних народів та з різними джерелами звуку, значна кількість з них закріпилася, навіть якщо це відбувалося ненавмисне.

Сучасний автор, працюючи над новою піснею, виражається завдяки відомим йому музичним закономірностям, або експериментує шукаючи нові. В будь-якому разі результат його роботи буде містити набір певних закономірностей, навіть якщо він не знає їх назви, або знайшов їх випадково. Їх ми в даному випадку і називаємо «концепціями».

В чому полягає проблематика використання конкретних значень. Насправді немає нічого поганого в їх використанні. Іноді навіть простіше описати ідею конкретно, ніж намагатися знайти відповідні абстракції. Однак проблема полягає в тому, що рівень роботи з концепціями в сучасних DAW (програми для написанням музики) відсутній взагалі. А його використання могло б дати безліч переваг:

1. Збереження інформації про зв’язки, а не конкретні ноти, дозволило б легше редагувати матеріал при зміні певних концептуальних складових композиції.

2. Автор зміг би працювати на тому рівні, який зараз для нього зручний. Міг би редагувати композицію як на концептуальному рівні так і на рівні деталей реалізації. Це як працювати широкими мазками та більш детальними, до того ж, маючи можливість в будь-який момент замінити одні іншими.

3. Це дозволило б писати музику людям, які мало знайомі з її теорією, оскільки їм не довелося б занурюватися до деталей реалізації, а тільки відбирати концепції та повідомляти про них програмі.

4. Зрозуміло, якщо автор працює лише з концепціями, а програма – з впровадженням змін та реалізацією, − то такий алгоритм дозволив би створювати музику значно швидше.

Мета і задачі досліджень. Метою роботи є пошук та реалізація засобів інтеграції концептуального підходу до програм з написання музики. Так щоб існував зручний інструментарій для роботи як на рівні конкретної реалізації так і на рівні контексту.

Важливо зазначити, що у нас на меті не має створити систему, яка генеруватиме музику «з нуля» самостійно. Таких систем вистачає, однак вони не можуть слугувати надійним інструментом для втілення задумів композитора, оскільки приймають рішення лише на основі жорстко прописаних в них програмних алгоритмів.

Об’єкт дослідження – музичне програмне забезпечення, що реалізує способи зберігання концептуальних даних та їх застосування в процесі роботи над музичною композицією.

Предмет дослідження – знання музичної теорії, засоби генеративного мистецтва та можливості програмних алгоритмів для створення системи, здатної працювати над музичною композицією на рівні контексту.

Методи дослідження. Для вирішення поставленої задачі було проведено дослідження історії генеративного мистецтва і генеративної музики зокрема. Досліджено теорію музики з виокремленням концептуальних складових, на основі яких пізніше будувалися алгоритми (складові такі як тональність, ритмічний малюнок, інтервальна закономірність та гармонійна). В проектуванні системи ми спиралися на принципи побудови чистої архітектури SOLID, можливості багатопоточного програмування, засади об’єктно-орієнтованої та функціональної парадигм програмування.

При розробці алгоритмів ми керувалися задачею зробити їх максимально ефективними, з цією метою, спиралися при їх розбудові на парадигму «розділяй та володарюй», в основі якої – розділення складної задачі на менші з подальшим їх рекурсивним вирішенням.

Для розробки інтерфейсу програми, ми використовували мобільну платформу Android та мову програмування Kotlin.

Зрозуміло, що для вирішення задачі, нам довелося звернутися до математичних обчислень. Вирішити безліч логічних задач, щоб перетворити набір розрізнених концепцій в робочу систему.

Наукова новизна отриманих результатів полягає у створенні нового підходу до написання музики, що базується на використанні концептуальних закономірностей замість використання конкретних значень, в процесі роботи над музичною композицією у віртуальному середовищі.

Практичне значення одержаних результатів. Створена система, здатна встановлювати та зберігати зв’язки між різними складовими композиції. Зв’язки описані елементами, що базуються на музичній теорії. Ця система дозволяє швидше вносити концептуальні зміни – при редагуванні глобальних концепцій, таких як ритмічний малюнок або тональність, зміни на рівні реалізації відбуваються автоматично. За рахунок часткової автоматизації програма дозволяє вносити зміни на необхідному рівні деталізації і дозволяє робити це в десятки або й в сотні разів швидше за ручне редагування.

Апробація результатів магістерської дипломної роботи. Основні положення та результати роботи були представлені та обговорені на XVII Всеукраїнській науковій конференції молодих вчених та студентів «Наукові розробки молоді на сучасному етапі» (м.Київ, КНУТД, 2019) додаток А. Також за результатами дипломної магістерської роботи була підготовлена і готується до друку наукова стаття у фаховому науковому журналі «Інженерія» додаток В.

РОЗДІЛ 1. ПОСТАНОВКА ЗАВДАННЯ ТА ОГЛЯД ІСНУЮЧИХ РІШЕНЬ

1.1. Поняття “генеративна музика”. Напрями розвитку

Генеративна музика (англ. Generative music) - напрямок музичної творчості, в основі якого лежить використання алгоритмів, а також прагнення досягти мінливості в художньому результаті. Термін отримав поширення завдяки британському музиканту-електроннику Браяну Іно, який у співпраці з компанією SSEYO 1996 року випустив програму Koan, призначену для створення "генеративної музики", а також цикл з 12 композицій *"Generative Music 1 with SSEYO Koan software"*, створених цією програмою.

Дослідник Р. Вуллер виокремлює чотири напрями розвитку генеративної музики [1].

1. Лінгвістичний/структурний. Використовує алгоритми, що націлені на генерацію структурно-зв'язного матеріалу, спираючись на досягнення Генеративної граматики Чомскі [2, 3, 4], а також музикознавчі розробки Фреда Лердала і Рея Джекендоффа [5]. Алгоритми базуються на деревовидній структурі.

2. Інтерактивний/поведінковий. Музика породжується системою, що нібито не має входів, і характеризується як "така, що не трансформується" (not transformational). Як приклад наводиться *Generative Music 1* Браяна Іно [6].

3. Креативний/процедурний. Процеси створення музики розробляються або ініціюються композитором. Як приклад наводиться *It's Gonna Rain* Стіва Райха та *In C* Террі Райлі.

4. Біологічний/послідовний. Недетермінована музика [2], або музика, яку не можна відтворити [7]. Автори цієї ідеї порівнюють роботу композитора з вченими-селекціонерами, що займаються виведенням нових

біологічних видів. Як приклад наводиться *"Вірусна симфонія"* Йозефа Нехватала – композиція жанру noise, в якій за основу взято модель розмноження вірусів.

В Україні генеративна музика вперше була представлена на академічній сцені в проекті EM-VISIA 2011 року в рамках фестивалю *Форум музики молодих* під керівництвом композиторки Алли Загайкевич [8]. Програмне забезпечення було застосовано для створення нотного тексту, а озвучення покладено на музикантів-інструменталістів.

Генеративний метод як такий, що лежить в основі віртуального генеративного мистецтва і заснований на факторі випадковості виник задовго до появи персональних комп'ютерів. Так, в 1751 році англійський музикант Вільям Хейс запропонував оригінальний спосіб твору музики та описав його в сатиричному трактаті «Мистецтво складати музику виключно новим методом, придатним для найгірших талантів». «"Винятково новий метод" полягав у тому, що потрібно взяти звичайну щітку (можна зубну), вмочити її в чорнильницю і, провівши пальцем по щетині, розбризкати чорнило на лист нотного паперу. Отримані ляпки повинні позначати положення ноти на нотній лінійці. До них потім залишається додати тактові риси, штилі та ін. Причому це все також вибиралося не за бажанням «композитора», а залежно від карти з колоди, яка потрапляла йому до рук. Після всіх цих творчих мук «твір» готовий до виконання» [9, с. 34].

Думка про випадковість, непередбачуваність підсумків творчого процесу виявилася досить популярною в Західній Європі XVIII століття і породила таке явище, як "Musikalisches Würfelspiel" (нім.) - музична гра в кістки, суть якої зводилася до створення "випадкових" музичних творів завдяки, мабуть, найпоширенішому прототипу генератора випадкових чисел - гральних кубиків. "Musikalisches Würfelspiel", як переконували виробники коробок з грою, дозволяла будь-якому дилетанту написати менует, контрданс,

вальс або марш - без потреби вивчати музичну грамоту [10]. «Ідея полягала в тому, щоб вигадати, наприклад, менует, компонуючи готові, заздалегідь написані музичні частини в порядку, що визначається за допомогою кидання грального кістки. Навіть із одним шестигранним кубиком кількість можливих комбінацій стрімко зростає: при п'яти кидках - 7776 комбінацій, при шести - 46656 [11]. Авторами подібних цікавих ігор виступали маститі композитори. У 1757 Кірнбергером було опубліковано «Посібник до твору полонезів і менуетів за допомогою грального кісток». У 1793 році було видано приписуване Моцарту «Керівництво, як за допомогою двох грального кісток складати вальси в будь-якій кількості, не маючи жодного уявлення про музику та композицію» [10].

У XVIII ст. з'явився перший «секвенсор» – шарманка. «Мелодії» та «акомпанемент» за допомогою спеціальних шпильок наносилися на валик, який при перегортанні відкривав доступ до необхідних труб. «Музиканту залишалось тільки натиснути на «Play», тобто почати крутити ручку, і записана на валику музика починала звучати [12].

У XIX ст. І.Н. Мельцелем був винайдений метроном (1814 р.) і механічний музичний інструмент – пангармонікон (Panharmonicon), для якого Л. Бетховеном була спочатку написана композиція «Битва при Вітторії» (1813 р.) (більш відома як симфонічна п'єса).

У 1837 р. Ч. Пейджем була розроблена «гальванічна музика». «У XVIII ст. знання звуковисотності було зведено до навчання про ладову природу звуковисотної організації. Так, фундаментальні для звукових характеристик поняття консонансу та дисонансу стали поєднуватися з представленням про стійкість та нестійкість, а самі властивості інтервалів розглядались на цьому етапі в тісному зв'язку з ладовими умовами їх застосування. Тоді ж з’явилося представлення про розв’язання дисонансу, яке виконувалося залежно від внутрішньо-ладових тяжінь. Лад розглядався як

єдиний закон звуковисотної логіки, і схожа трактовка предмета дослідження про звуковисотність іноді зустрічається і сьогодні» [12].

В XIX в. першим у світі отримав подобу електронного звуку Гельмгольц, який сконструював апарат, названий резонатором Гельмгольца, і написав кілька фундаментальних робіт, присвячених фізичним і фізіологічним основам музики. Серед них, зокрема, книга «Вчення про звукові відчуття як фізіологічна основа теорії музики» (Lehre von den Tonenefindungen als physiologische Grundlage fur die theorie der Musik). Саме в цій роботі наданий детальний аналіз проблем фізіології слуху, формування тембру звуку, консонансу та дисонансу, питанням організації звуковисотності. Необхідно відміти також, що Гельмгольц висуває принцип щодо консонансу та дисонансу, чим передбачає вчення про гармонію А. Шенберга [13].

Пізніше американським винахідником Е. Греєм був створений інструмент під назвою Harmonic Telegraph. Кожна клавіша цього двооктавного інструменту мала власний електромагнітний генератор відповідної частоти. Звук інструменту був слабким, тембр – схожим на телефонний гудок.

«Метод нарізки» як окремий прояв генеративного методу вплинув на розвиток електронної та експериментальної музики. Так, німецький композитор Карлхайнц Штокхаузен у своїх «Гімнах» компонував «уривки мови, фольклорні звучання, записані розмови, радіоперешкоди, звуки офіційних заходів, маніфестацій, освячення корабля, державного прийому та ін. Весь звуковий колаж існує різного ступеня обробки, об'єднаний прийомами монтажу, де переходи іноді підкреслені, іноді максимально згладжені» [14].

Браян Іно, англійський музикант, батько музичного жанру ambient, також вдавався до генеративного підходу: наприклад, записана на його альбомі "Discreet Music" (1975) однойменна тридцяти-хвилинна композиція є

експериментом з затримкою відлуння (delay), коли дві прості та подібні за структурою мелодії, записані на магніт, повторюються з різною послідовністю і накладаються одна на одну, при цьому відбувається випадкова зміна тембру синтезатора на виході [6].

Ці та інші подібні експерименти започаткували явище, відоме як *sampling* — компонування готових звукових фрагментів (семплів), переведених в цифровий формат, без якого немислима сучасна музична індустрія.

Мабуть, найяскравішим прикладом є творчість американського композитора Джона Кейджа, основоположника алеаторики. Принцип випадковості грав там центральну роль. Твори Кейджа «вимагали свободи переміщення слухачів, їхньої відкритості та розчиненості в акустичних подіях (*happening*), в яких могли змішуватися кілька композицій і які на різних майданчиках звучали по-різному» [15]. Його найзнаменитіша робота − "4,33" (1952) − свого роду вершина музичного генеративного мистецтва. Композитор поставив лише тривалість звучання — 4,33 хвилини, в іншому ж воно цілком випадкове. Без будь-якої участі з боку музикантів, в умовах абсолютної тиші «мелодія» при кожному «виконанні» народжувалася наново з теперішнього моменту — завдяки різним стороннім звукам, чи то шарудінням, чи шумом у залі для глядачів, чи шумом вулиці [16].

Коли композитор за традиційною технологією складає музику, він має в своєму розпорядженні строго обмежений набір інструментів (тембрів). Вважається, що робота композитора в кінцевому рахунку полягає в компонуванні елементів різної висоти, тривалості та гучності. Іншими словами, вважається не настільки важливим, на якому інструменті буде виконаний складений композитором нотний текст, який тембр матиме той чи інший звук. Потім стає очевидною ідея, що темброве забарвлення музики має значну роль при сприйнятті її характеру. Інакше кажучи, одна й та сама

мелодія у виконанні труби та скрипки буде сприйнята слухачем по-різному. З того часу композитори обирали один або кілька конкретних інструментів для втілення своєї ідеї, а симфонічний оркестр аж до нашого часу по праву вважається інструментом з найбагатшими виконавськими можливостями. [1].

Навіть при створенні музики для оркестру, з його досить великим набором інструментів, композитор не має ніякої можливості змінити суворо відоме заздалегідь темброве наповнення відповідно до свого композиторського задуму. Щоб хоч якось розширити ці обмеження, композитори вигадували різні темброві доповнення, так звані «міксти». Спочатку вибиралися найбільш природні «інструменти», що органічно зливаються, потім стали з'являтися все більш незвичайні темброві поєднання.

З появою музично-комп'ютерних технологій (МКТ) [17] та, власне, музичного комп'ютера (МК) [17] композитор отримав можливість створювати та використовувати звук будь-якого бажаного тембру. Сучасні МКТ знімають всі важливі темброві обмеження: обмежувати можуть лише можливості професійного програмного забезпечення та вміння композитора користуватися ними.

У цілому нині можна зазначити, що математичні методи дослідження, у музикознавстві переважно зводяться передусім до синтезування музики з урахуванням закономірностей, отриманих на етапі аналізу композицій, що докладно розглянуто нами у роботах [13]. «В узагальненні (яке могло стати предметом окремої теоретичної дискусії), − пише М.С. Заливадний, − знаходить очевидний вираз як комплексність суспільно-психологічної основи музичної (та іншої художньої) творчості [18, с. 328], і фундаментальний характер самого елемента програмування (чи, точніше, алгоритмізації), що у тій чи іншій формі в усіх галузях людської діяльності» [18, с. 12].

А. Сихра у зв'язку з дослідженням можливостей комплексного наукового підходу до вивчення закономірностей музики (включаючи

застосування ідей та методів кібернетики та теорії інформації) у своїх наукових працях активно використовував елементи низки точних дисциплін (у тому числі кібернетики та теорії інформації), а також досягнення комплексного теоретичного музикознавства XX ст. Так, одна з глав його книги, що стала бестселером «Музика очима науки» [19] називається «Музика і кібернетика».

Таким чином, генеративний метод, заснований на факторі випадковості, що спочатку виник у рамках музичних та літературних експериментів, у другій половині XX століття оформився у самостійний художній напрямок – генеративне мистецтво, що передбачає використання автономних систем – зокрема, генераторів псевдовипадкових чисел – для створення особливих художніх ефектів, несподіваних як для глядача, так і для самого автора. Дане явище має синтетичну природу, оскільки його витоки - в ідеях і принципах класичного російського авангарду, дадаїзму, сюрреалізму, мінімалізму, що послужили благодатним підґрунтям для подальшого творчого пошуку вже з використанням сучасних комп'ютерних технологій.

Генеративне мистецтво — унікальний художній феномен — передбачає творчий процес за допомогою автономних систем, які або виходять за рамки контролю зі сторони художника лише частково, або повністю непідвладні йому.

Для найбільш повного розуміння природи генеративного мистецтва необхідно чітко позначити його кордони і виявити можливі підвиди. Як відомо, подібні спроби вже були і, можливо, найбільш серйозною з них є спроба двох зарубіжних дослідників: професора когнітивних наук в університеті Сассекса, лауреата премії Аллена Ньюелла ACM-AAAI за вклад у філософію когнітивної науки (2018) Маргарет А. Боден і одному з першопрохідців в сфері комп'ютерного мистецтва, володаря нагороди ACM SIGGRAPH 2017 року за видатні досягнення в галузі цифрового мистецтва

Ернеста Едмондса. Обидва автори написали десятки наукових робіт про актуальні арт-практики, існуючі на стику традиційних і пост-інформаційних технік, методів і засобів виразності, і тим самими внесли великий вклад в осмислення питань оцифровування художньої творчості .

В своїй праці під назвою «Що таке генеративне мистецтво?» Боден та Едмондс виокремлюють 11 взаємозалежних видів мистецтва (1): Ele-art (електронне), C-art (комп'ютерне), CA-art (комп'ютерно-опосередковане), D-art (цифрове), G-art (власне генеративне), CG-art (комп'ютерно-генеративне), Evo-art (еволюційне), R-art (робототехнічне), I-art (інтерактивне), CI-art (комп'ютерно-інтерактивне) та VR- art (мистецтво віртуальної реальності) [20].

На підставі класифікації Бодена та Едмондса, В.Лукічев [21] запропонував власну класифікацію (рис.1.1):

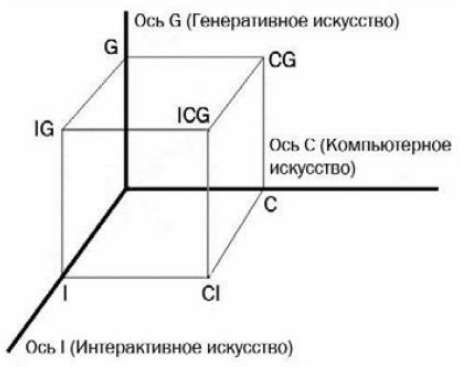

Рис 1.1. Ключові вектори генеративного мистецтва

Класифікацію ген-арту та його контексту В. Лукічев представив у вигляді трьох ключових векторів [21]:

1. Вісь G («генеративна») — описує походження змісту та/або формально-стилістичної сторони згенерованого твору, заснованих на принципі випадковості;

2. Вісь С («комп’ютерна») — описує сукупність апаратних і програмних засобів, використаних при створенні творів;

3. Вісь І («інтерактивна») — описує порядок взаємодії глядача з творінням, можливість участі публіки в його створенні, розвитку, трансформації та ін.

1) G-art (Generative art) — генеративне мистецтво (G=1, C=0, I=0) — сукупність арт-об'єктів, створених принаймні частково деякими автономними процесами фізичних, не віртуальних властивостей, без безпосереднього контролю з боку автора.

2) C-art (Computer art) - Комп'ютерне мистецтво (G = 0, C = 1, I = 0) - сукупність арт-об'єктів, створених із застосуванням комп'ютерних технологій.

Зазначимо, що терміни «комп'ютерне мистецтво» і «цифрове мистецтво» часто виступають синонімами, хоча це не зовсім так: поняття «комп'ютерне мистецтво» явно ширше і передбачає використання навіть аналогових пристроїв, до яких досі вдаються музиканти, бажаючи передати візуальну або аудіо інформацію без погіршення її початкової якості, що неминуче виникає при перетворенні аналогового сигналу до дискретного та навпаки [22].

3) I-art (Interactive art) — інтерактивне мистецтво (G=0, C=0, I=1) — сукупність творів, форма чи зміст яких в значній мірі залежать від участі глядача.

4) CI (Computer Interactive art) — інтерактивне комп'ютерне мистецтво (G=0, C=1, I=1) — сукупність творів, створених на комп'ютері чи іншому мультимедійному пристрої, які передбачають повне чи часткове включення глядача в арт-процес без використання автономних систем. Причому включення може відбуватися різними способами: наприклад, аудиторія може формувати конструктивну чи сюжетну основу твору, впливати на його аудіовізуальні характеристики, змінювати форму, колір, композиційну

структуру, вносити корективи до вже готових варіантів чи створювати їх практично "з нуля".

Прикладів інтерактивного медіа-мистецтва можна навести безліч, оскільки ця категорія є дуже затребуваною: комп'ютерам спочатку властива взаємодія з користувачем на основі інтерфейсів. В зв'язку з цим вмикання глядача в ігровий арт-процес за допомогою мультимедіа відбувається природно, як би саме-собою. Це особливо зрозуміло з позиції сьогоднішнього дня, коли нам стали звичними у повсякденному житті сенсорні екрани і різноманітні Інтернет-сервіси, включаючи соціальні мережі та відео-чати, мобільні додатки, і багато іншого. Приходячи на виставку інтерактивного медіа-мистецтва, глядач уже підготовлений та інтуїтивно розуміє, на яку кнопку слід натиснути, якої частини екрану доторкнутися, де провести рукою тощо.

5) CG-art (Computer-Generated art) - комп'ютерний ген-арт (G = 1, C = 1, I = 0). Комп'ютерний ген-арт — сукупність творів, створених на комп'ютері чи іншому мультимедійному пристрої за допомогою автономного виконання послідовності команд, які написані художником-програмістом та передбачають використання генератора випадкових чисел із наступною візуалізацією отриманих результатів. Елементи інтерактивності в цій категорії не задіяні, як і відображено на нашій схемі: значення по осі I дорівнює нулю.

Митець, запустивши програму, може піти на ланч: комп'ютер, керуючись заданим алгоритмом, самостійно формує арт-об'єкт, ухвалюючи «автономні рішення» там, де автор коду надав таку можливість. Рішення машини ґрунтуються на випадкових числах, що і створює безліч варіацій — автору музики (а можливо співавтору) залишається тільки відібрати найбільш відповідне рішення з усього запропонованого різноманіття.

6) IG-art (Interactive Generative art) — Інтерактивний ген-арт (G=1, C=0, I=1) — сукупність творів, створених за активної участі глядача та залучення «традиційних» (поза-комп'ютерних) автономних систем, що забезпечують той або інший ступінь випадковості без використання мультимедійних технологій.

7) ICG-art (Interactive Computer-Generated art) – інтерактивний комп'ютерний ген-арт (G=1, C=1, I=1) – сукупність творів, створених за допомогою автономних комп'ютерних систем та що передбачають активну взаємодію з глядачем.

Тут, як бачимо, має місце «випадковість у квадраті»: з одного боку, її забезпечує машина з генератором чисел, що видає довільний аудіовізуальний матеріал на основі заданих художником початкових параметрів (як у комп'ютерному ген-арті), а з іншого боку, «гра випадку» створюється поведінкою користувачів, кожен з яких не гірше за генератор випадкових чисел додає хаос та передбачуваність своїми рухами тіла, тембром голосу, діями чи бездіяльністю — залежно від функціоналу тієї чи іншої інтерактивно-генеративної медіа-моделі.

В контексті нашого дослідження, нашу розробку можна віднести до категорії комп'ютерного CI – інтерактивного комп’ютерного мистецтва, а в подальшому до ICG.

Таким чином, ген-арт, перебуваючи в тісній взаємодії з інтерактивним і комп'ютерним мистецтвом, може проявляти властивості одного з них в додачу до власних, або об'єднувати в собі одразу три названі вектори, отримуючи зрештою складну автономну систему, засновану на мультимедійних технологіях, яка передбачає активну взаємодію з глядачем.

1.2. Огляд існуючих рішень

Перш ніж скласти список умов до майбутньої програми та перейти до її проектування, потрібно розглянути існуючі на даний момент рішення, які,

можливо, в тій чи іншій мірі вирішують проблему концептуального підходу (рис. 1.2).

1.2.1. DAW

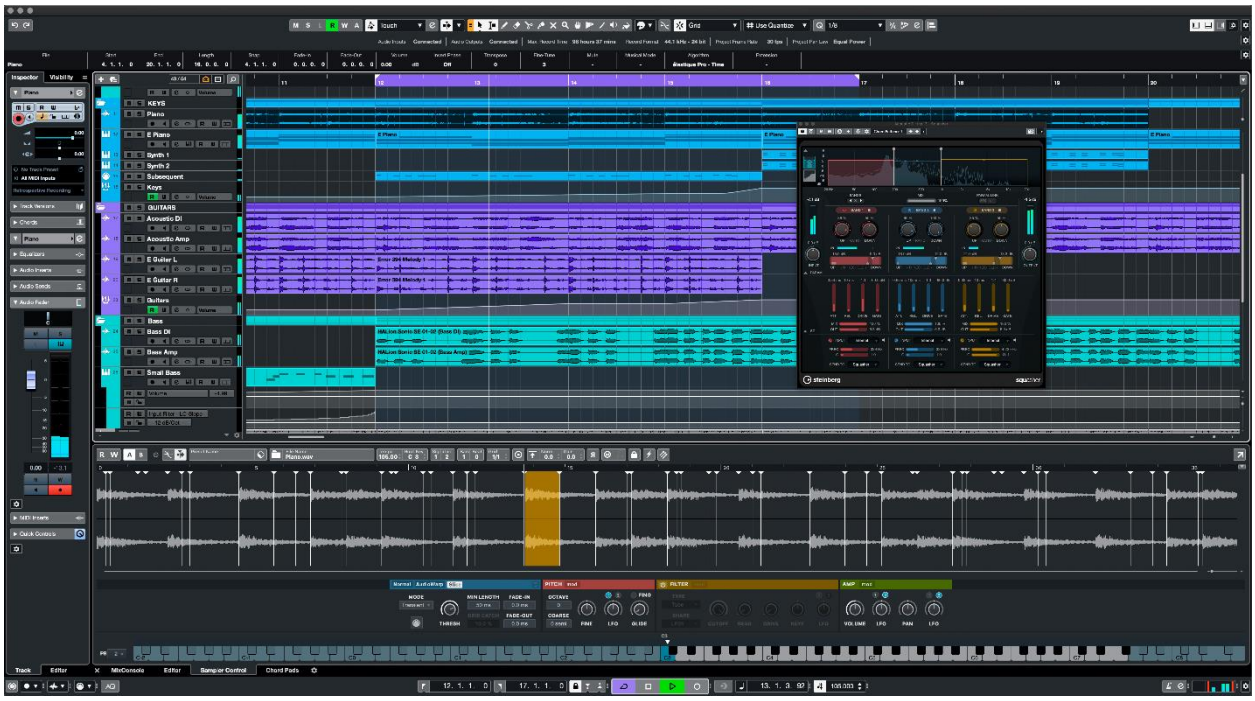

Рис. 1.2. Інтерфейс програми Cubase 10

Цифрова звукова робоча станція (англ. Digital Audio Workstation, DAW) в термінології музичних продюсерів — електронна система, призначена для запису і редагування цифрового аудіо. Основна властивість робочих станцій — можливість вільно маніпулювати із записаним звуком, більшість робочих станцій також підтримують технологію MIDI [23].

Якщо MIDI-редактори вміють маніпулювати MIDI-даними, а аудіо-редактори обробляють аудіо-файли, то DAW уміє все це і багато іншого. Цифрові звукові станції можна назвати універсальними інструментами для написання музики. У них в одному флаконі зібрано все найважливіше, що потрібно для перетворення ідеї в якісний музичний продукт.

Для цього в кожному DAW реалізовано широкий набір як стандартних функцій (редагування MIDI, обробка аудіо тощо) так і унікальних для кожної програми функцій (наприклад, «Grid» – модульна станція синтезу звуку, присутня в програмі Bitwig).

Окрім того функціонал кожної програми завжди можна розширити, користуючись технологією plug-in, яка дозволяє встановлювати та використовувати сторонні програми всередині DAW.

Так можливості DAW можна розширити встановивши додаткові віртуальні інструменти (VSTi) − семплери, віртуальні синтезатори та інші розширення здатні генерувати звук. Також можна встановити розширення для обробки звуку (позначаються просто VST) – різні ефекти, такі як реверберація (ефект приміщення), delay (ефект повторення) та інші.

Серед найпопулярніших DAW: FL Studio, Cubase, Ableton Live, Logic Pro.

Хоча сучасні DAW володіють безліччю потужного функціоналу (особливо це стосується програм які знаходяться на ринку давно), однак досі жодна з них не вирішує проблеми збереження та редагування музичного матеріалу на рівні концепцій.

Звісно, в багатьох реалізовані функції, які дозволяють одночасно редагувати великі об’єми даних, наприклад, змінювати висотне та часове положення музичного відрізку, виділивши його та переміщаючи як одне ціле, або навіть змінити тональність всієї композиції одним повзунком ручки. Однак концептуальні зв’язки між партіями при цьому ніде не зберігаються.

Ймовірно такий функціонал досі не з’явився лише з тієї причини, що не має чіткого розуміння як саме він повинен бути реалізований та як зробити його зручним та зрозумілим для кінцевого споживача.

Основні переваги DAW:

- широкий набір функцій;

- все необхідне для створення музики зібрано в одній програмі;
- можливість максимально детального редагування композиції;
- можливість розширення функцій сторонніми програмами.

Недоліки DAW:

Не зберігають інформації про концептуальні зв’язки між складовими елементами композиції та не мають інструментів для швидкої роботи з ними.

1.2.2. Програми повної генерації

До цієї категорії відносяться програми які генерують музику самостійно «з нуля», або з мінімальною участю оператора-людини. Наприклад, користувач обирає жанр, або список треків, що йому подобаються, а все інше реалізує програма.

Серед таких програм можна виділити, наприклад, онлайн-сервіс Mubert [24] На ньому користувач обирає одну з трьох категорій за якими буде йти подальше уточнення: жанр, настрій чи вид активності. Потім уточнення, наприклад, для категорії «активність» – під-категорії «зосередження», «спорт», «кафе» і подібне. Останнім параметром користувач обирає тривалість треку. Після чого програма керуючись базою семплів (звукових відрізків музичного матеріалу) та програмними алгоритмами, генерує трек заданої тривалості (рис. 1.3).

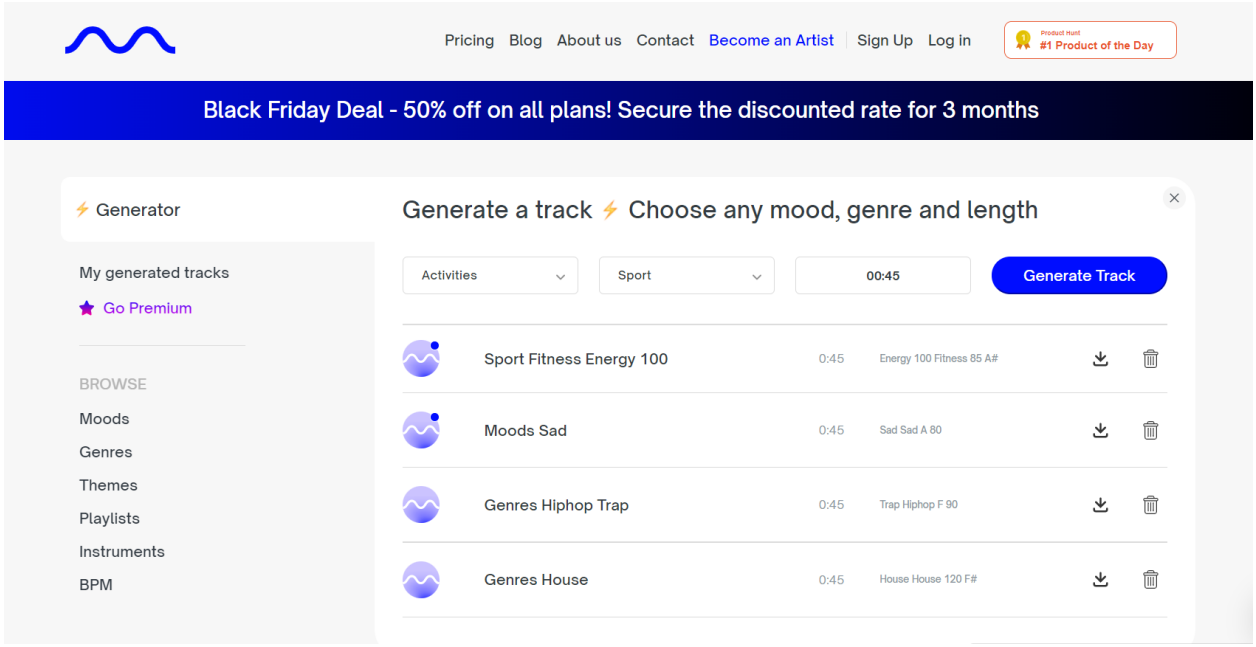

Рис. 1.3. Онлайн-сервіс Mubert

Ще одна програма цієї категорії – віртуальний композитор AIVA [25]. Програма найбільш відома своїми оркестровими композиціями, які складно відрізнити від написаних людиною (рис. 1.4).

Стала першим в світі віртуальним композитором, визнаним музичним товариством SACEM.

На даний момент працює з різними жанрами.

Рис. 1.4. Віртуальний композитор AIVA

Насправді існує велика кількість подібних програм, однак їх суттєвим недоліком є більша орієнтація на «самостійність». Тобто, якщо DAW не вистачає концептуальності, то цим програмам часто не вистачає більш детального налаштування та гнучкості.

Переваги:

- концептуальний підхід;

- в частині реалізована можливість редагування партій по закінченню генерації;

- достойні результати генерації, які складно відрізнити від створених людиною.

Недоліки:

- «самостійні» в процесі прийняття рішень, через що результати їх роботи складно вважати заслугою автора. Більшість потребує вказати лише жанр чи настрій;

- обмежені у варіативності. Всі результати формуються за однаковим принципом, оскільки фундаментальні рішення все одно приймає штучний інтелект;

- більшість не мають функції детального редагування (хоча в AIVA це реалізовано) [25];

- також не зберігають інформації про зв’язки, тому працюють на рівні концепцій лише перед початком генерації, а після неї – тільки на рівні редагування конкретних значень – нот або семплів.

1.2.3. Міді-процесори

Це категорія програм які займаються обробкою MIDI і можуть включати елементи генерації.

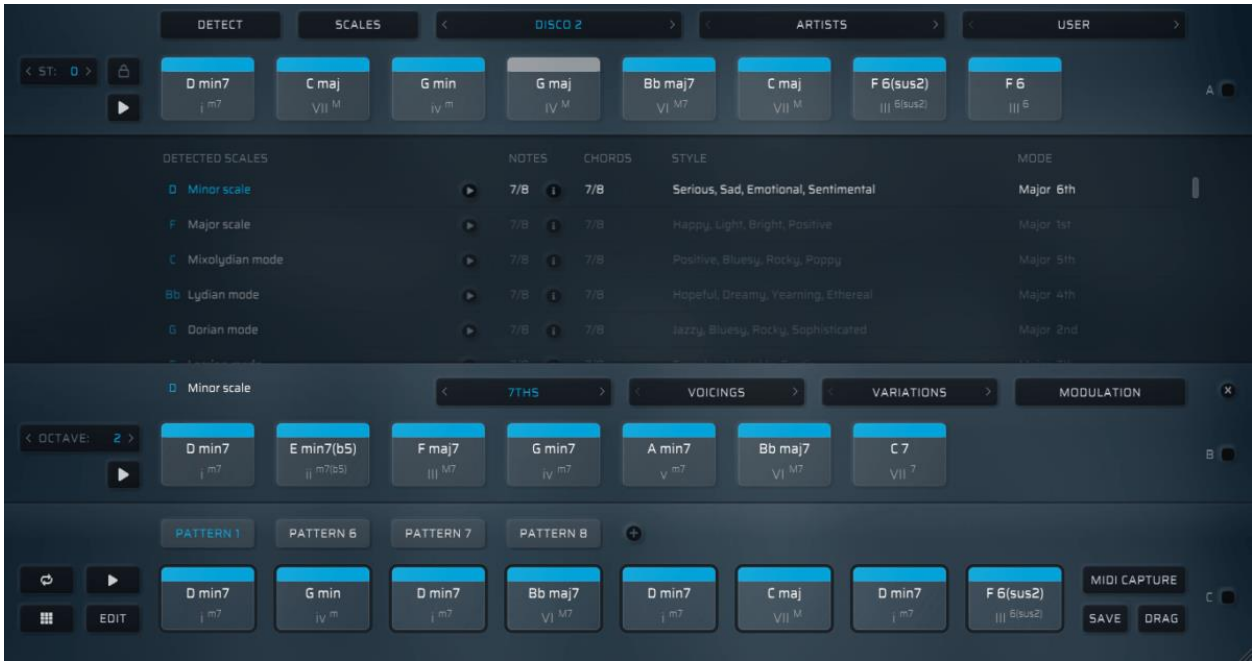

Рис. 1.5. VST-інструмент Scaler 2

Приклад такої програми VST-інструмент Scaler (рис.1.5). Це програма з широким набором функцій. Частина її можливостей:

- аналіз послідовності midi-акордів з визначенням їх типу та тональності, якій вони належать;

- широкі набори вбудованих тональностей та акордів, які можна використовувати для написання власних композицій;

- можливість перетворити партію зіграну одиночними нотами в послідовність акордів, де зіграні ноти будуть базовими нотами акорду;

- можливість відтворення послідовності акордів в різних артикуляціях та з різною ритмікою, наприклад, у вигляді арпеджіо.

За типом роботи Scaler найближче до програми, яку ми хочемо реалізувати. Однак він теж не зберігає інформації про зв’язки між кількома партіями, а приймає участь в процесі першої генерації кожної партії окремо. До того ж, застосовується лише для генерації акордів.

Переваги:

- широкий каталог доступних тональностей та варіацій акордів. На сайті обіцяють понад 1200 варіацій тих та інших.
- можливість обирати артикуляції для відтворення акордів.
- можливість експорту в MIDI.
- можливість використання в сторонніх програмах у вигляді розширення.

Недоліки:

- відсутня можливість установки та збереження зв’язків між партіями.
- неможливо одночасно генерувати кілька партій, кожна має генеруватися окремо.
- сам нічого не генерує, слугує тільки для перетворення обраних послідовностей акордів в міді-послідовності та емуляції їх відтворення.

Висновки до розділу 1

1. Генеративна музика (англ. Generative music) - напрямок музичної творчості, в основі якого лежить використання алгоритмів, а також прагнення досягти мінливості в творчому процесі. Термін отримав поширення завдяки британському електронному музиканту Браяну Іно, який у співпраці з компанією SSEYO 1996 року випустив програму Koan, призначену для створення "генеративної музики", а також цикл з 12 композицій *"Generative Music 1 with SSEYO Koan software"*, створених цією програмою.

2. Можна виокремити чотири напрями розвитку генеративної музики:

а) лінгвістичний/структурний. Використовує алгоритми, що націлені на генерацію структурно-зв'язного матеріалу, спираючись на досягнення Генеративної граматики. В основі алгоритмів − застосування деревовидних структур;

б) інтерактивний/поведінковий. Музика породжується системою з відсутністю вхідних даних, і характеризується як «така, що не має трансформації» (not transformational);

в) креативний/процедурний. Процеси створення музики розробляються або ініціюються композитором;

г) біологічний/послідовний. Недетермінована музика, або музика, яку не можна відтворити.

3. З появою музично-комп'ютерних технологій (МКТ) та, власне, музичного комп'ютера (МК) композитор отримав можливість створювати та використовувати звук будь-якого бажаного тембру. Сучасні МКТ знімають всі важливі темброві обмеження: обмежувати можуть лише можливості професійного програмного забезпечення та вміння композитора користуватися ними.

4. Ми не ставимо за мету створити систему, яка генеруватиме музику повністю самостійно. Таких систем вистачає, однак вони не можуть

слугувати надійним інструментом для втілення задумів композитора, оскільки приймають рішення лише на основі жорстко прописаних в них програмних алгоритмів. Мета роботи – знайти області в написанні музичної композиції, які можна автоматизувати, щоб поліпшити ефективність цього процесу.

5. Аналіз сучасних музично-комп'ютерних технологій (МКТ) показав, що вони володіють широким набором ефектів, інструментів запису та контролю за звуком. За бажання його можна розширювати, додаванням сторонніх бібліотек та незалежно скомпільованих програмних модулів – плагінів. Хоча сучасні технології дають авторам широкий простір творчої свободи, у них є серйозний недолік – вони досі зосереджені на рівні роботи з конкретними значеннями і не володіють функціями для роботи на рівні контексту.

6. Таким чином, метою нашої дипломної роботи – є пошук та реалізація засобів інтеграції концептуального підходу до програм з написання музики. Щоб користувач міг працювати над твором як на рівні конкретної реалізації так і на рівні контексту. В ході роботи ми маємо знайти способи зберігання концептуальних даних та їх застосування в процесі роботи над музичною композицією. Для досягнення цієї мети, ми будемо спиратися на знання з музичної теорії, можливості генеративного мистецтва та програмних алгоритмів.

РОЗДІЛ 2. ХІД РОЗРОБКИ

2.1. Вхідні та вихідні дані програми

Вхідними даними програми є набір умов різного типу, заготовки яких (шаблони) зберігаються в програмі. Користувач відбирає шаблони на основі своїх побажань та додає до різних відрізків (сегментів) партії.

Ці умови мають вигляд заготовлених шаблонів, які впливають на певні характеристики нот вихідної партії.

Вихідними даними програми є набір міді файлів. Кожен міді файл представляє собою послідовність нот вихідної партії. Ці міді файли можна відкрити та використовувати в інших міді-редакторах, таких як FL Studio, Ableton, Cubase та інші.

2.2. Короткий опис MIDI-протоколу

Вихідними даними програми є midi-файли. Таке рішення прийнято з метою зворотної сумісності, щоб результати роботи програми можна було використовувати в інших міді-редакторах, оскільки міді-протокол широко розповсюджений і я стандартом представлення музичних даних на комп’ютері.

Програма базується на роботі з міді, тому потрібно коротко про нього розповісти.

2.2.1. Історія виникнення

MIDI протокол (Musical Instrument Digital Interface – цифровий інтерфейс музичних інструментів), представлений у 1982 році. Передумовою його створення стала потреба у передачі інформації про натискання клавіши між кількома синтезаторами поєднаними у локальну мережу.

Задача була наступною – музикант натискає на одну або кілька клавіш на одному синтезаторі і одночасно ті ж самі ноти звучать на інших синтезаторах, включених послідовно з ним.

З часом застосування MIDI привело до появи синтезаторної пам'яті. Музичний фрагмент зіграний на синтезаторі, записувався у MIDI форматі, і далі міг бути відтворений у будь-який момент часу. На відміну від запису на пластинку чи магнітофон, міді протокол дозволяв зберігати не фрагмент аудіо, а послідовність нот, яку можна неодноразово редагувати без втрати якості.

Наступним важливим етапом розвитку MIDI, стала поява комп'ютерних програм – *midi-редакторів*, які дозволяли прописувати та редагувати фрагменти midi-інформації прямо на екрані комп'ютера, без необхідності підключення синтезатора чи інших інструментів передачі MIDI даних.

Пізніше міді-редактори еволюціонували в DAW (Digital Audio Workstation – Цифрова звукова робоча станція). DAW помістили процес створення музичної композиції в межі однієї програми з додатковими розширеннями – віртуальними аналогами музичних інструментів та ефектів (VST та VSTi). Серед популярних сучасних DAW: Steinberg Cubase, Logic Pro, FL Studio, Ableton Live та інші.

2.2.2. Будова MIDI файлу

Типовий MIDI-файл складається з послідовності команд, що в протоколі носять назву *події*. Кожна подія складається з двох частин – команди, яка описує що саме має відбутися, та мітки часу.

Прикладами подій можуть слугувати наступні:

- *Note on* – стартова позиція звучання ноти;
- *Note off* – фінальна позиція звучання ноти;
- *Program change* − Зміна обраного тембру.

Типів подій в MIDI існує велика кількість. Основні наведені на рис. 2.1:

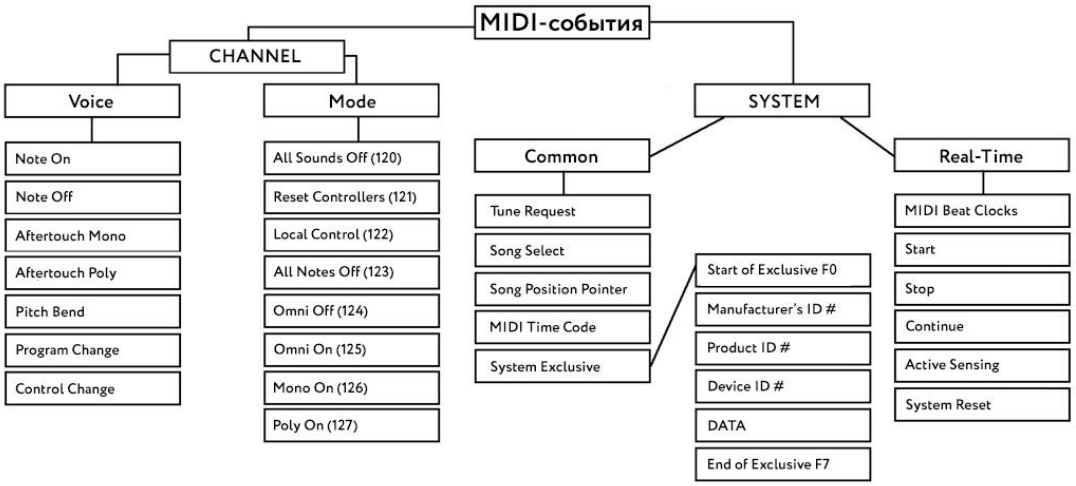

Рис. 2.1. Основні MIDI-події

Вся інформація в міді записується в двійковому форматі. Це означає, що у кожної події є двійковий код, завдяки якому програма міді-редактор може відрізнити одну команду від іншої.

При перегляді в звичайному текстовому форматі міді-файл матиме схожий вигляд (рис. 2.2):

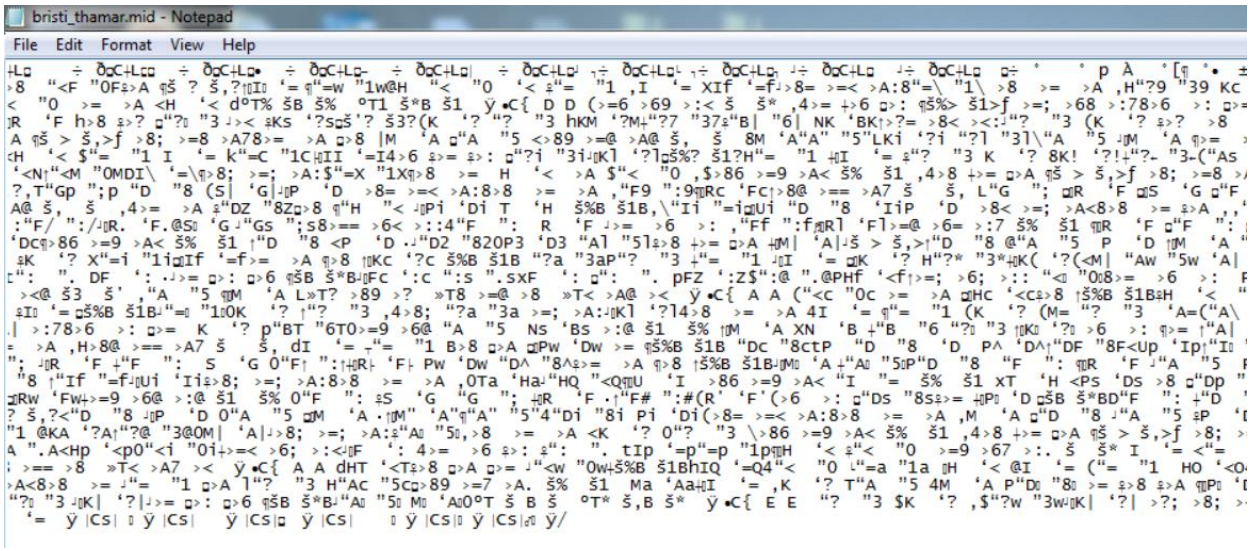

Рис. 2.2. Вигляд міді-файлу відкритого в текстовому редакторі

Вся причина в тому, що текстовий редактор сприймає послідовність двійкових команд як зашифрований набір символів, тоді як в них зашифрована зовсім інша інформація.

При конвертуванні міді-файлу до текстового формату, він мав би схожий вигляд (рис. 2.3):

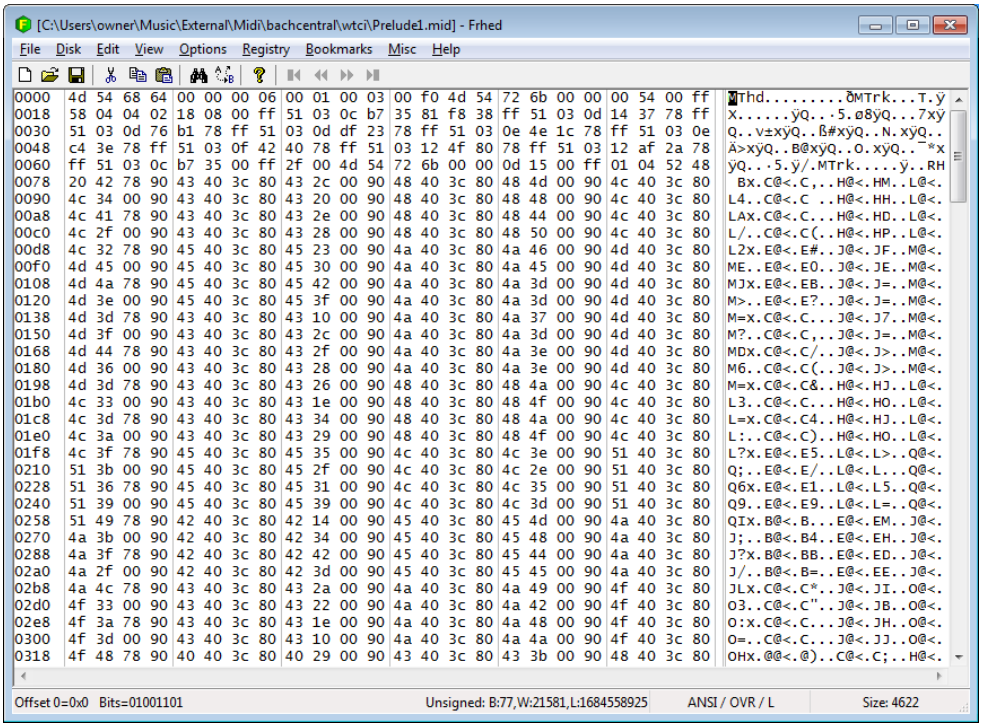

Рис. 2.3. Міді-файл в текстовому представленні

При передачі команди «Note on» чи «Note of», додатковим параметром передається індекс ноти для якої має відбутися подія. Індекс ноти слугує міткою ноти за висотою. Приклад індексів на рис. 2.4:

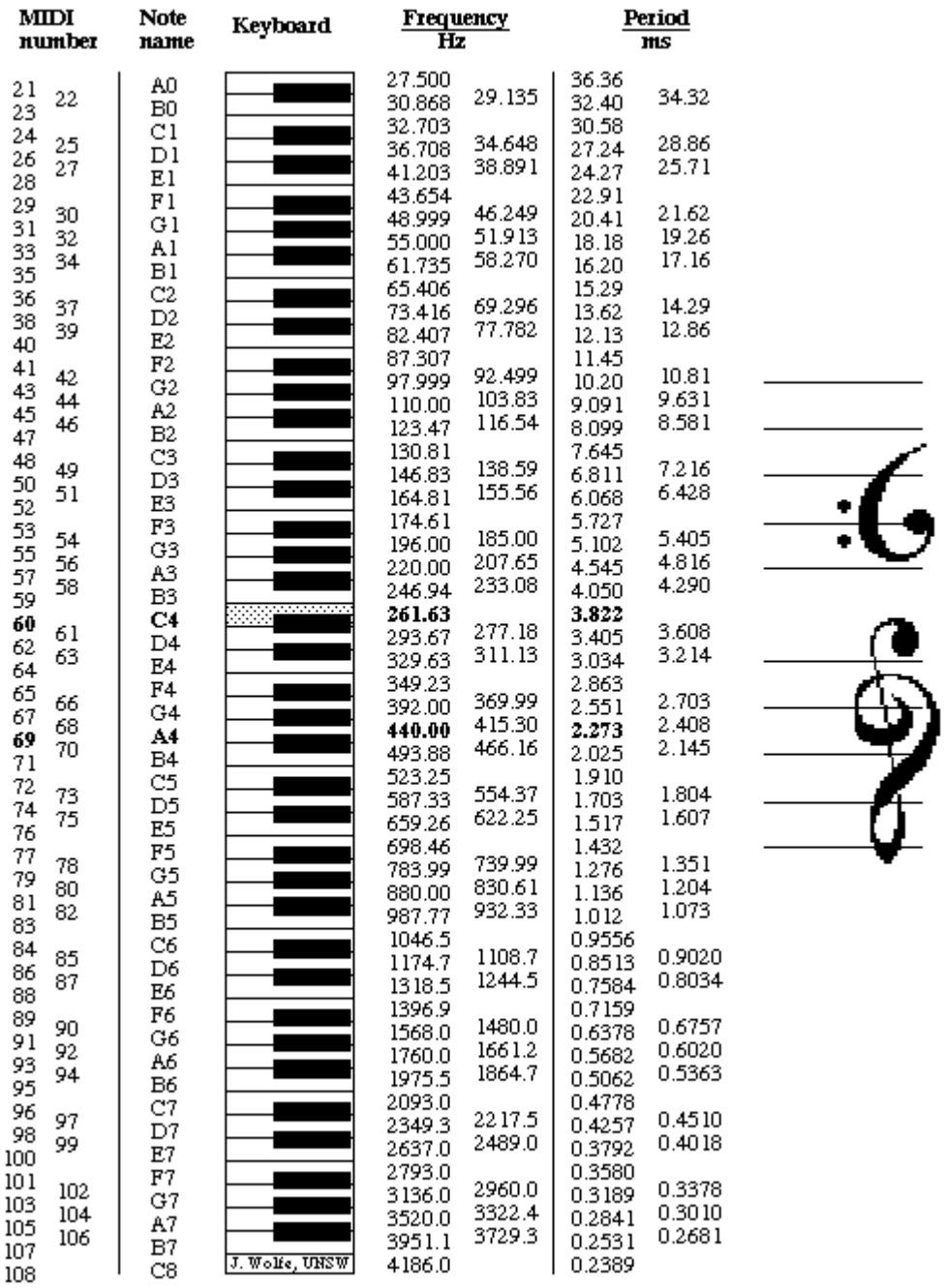

Рис. 2.4. Відповідність індексів висоти нот в MIDI та прийнятих позначень в музичній нотації

Таким чином, послідовність нот в midi-файлі, записується послідовністю подій «*Note on*» та «*Note off*» переданих з міткою висотного індексу ноти та міткою часу, коли клавішу натиснули та відпустили.

Для відкривання та редагування інформації з міді-файлу, використовуються спеціальні програми, що мають назву міді-редактори. У них міді-інформація може бути представлена у вигляді звичного музикантам

нотного письма, або у вигляді кубічної сітки (Piano roll), в якій ноти розміщуються більш зрозуміло для сучасних музикантів. Piano roll складається з двох осей. Положення ноти на осі Y відображає висотне положення ноти, а положення на осі X – часове (рис. 2.5).

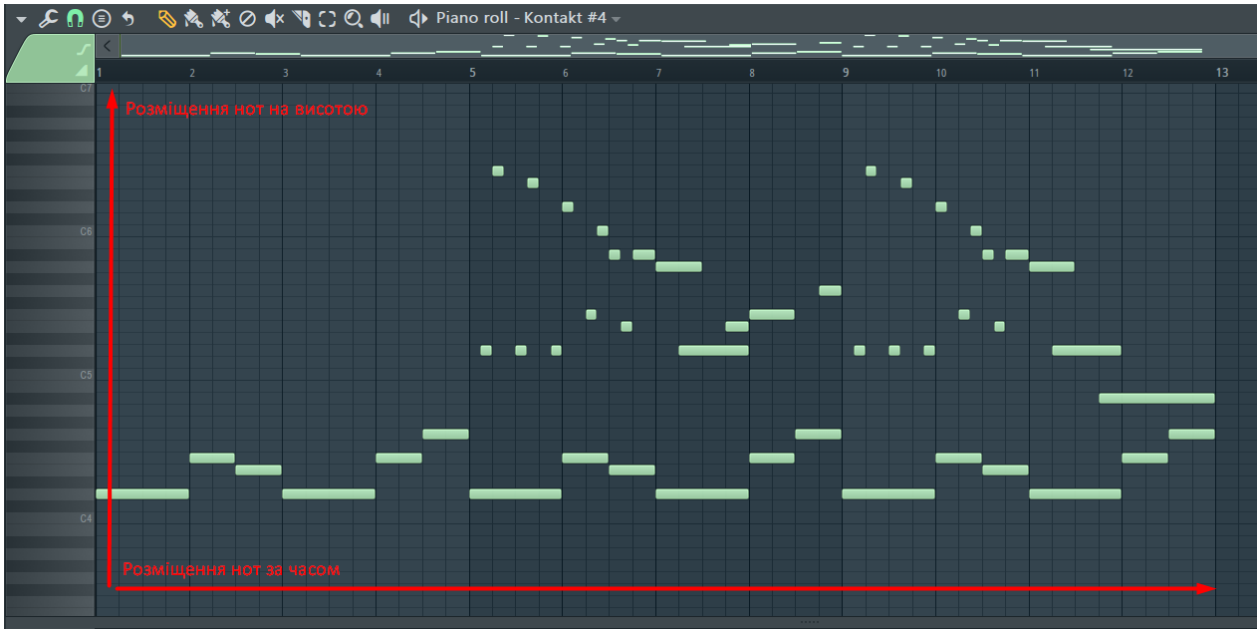

Рис. 2.5. Вигляд midi-файлу в Piano Roll FL Studio

2.2.3. PPQN

В MIDI час кожної події записується не в точних значеннях, таких як секунди чи хвилини, а у відносних, які в протоколі мають назву PPQN.

В академічному музичному записі також використовується набір відносних одиниць для позначення тривалості нот. Такі одиниці носять назву: цілі, половинні, четвертні, восьмі, шістнадцяті і так далі. Реальна ж тривалість ноти залежить від темпу, який позначається в кількості четвертних нот на хвилину та має назву BPM (Beat per minute – дослівно перекладається як «кількість ударів на хвилину»).

Таким чином маючи відносне значення ноти та знаючи темп, можна встановити її фактичну тривалість в конкретних величинах.

PPQN (Pulses Per Quarter Note – «кількість імпульсів в четвертній ноті») – величина що встановлює часове розширення міді-файлу та дробить четвертну ноту на менші величини для більш точного позиціонування кожної події за часом.

PPQN в різних міді-файлах може мати різне значення. Наприклад, в нашій програмі ми встановили його рівним «32», оскільки вважали це значення достатнім для довільного позиціонування за часом. Однак це просто дискретна величина, яка може бути більшою.

Якщо висловлюватися просто, то в midi для кожної події міткою часу служить номер імпульсу який минув від старту файлу до моменту коли подія має відбутися. Знаючи темп BPM та розширення файлу (скільки імпульсів міститься в четвертній ноті), цей час можна перевести в конкретні величини.

2.3. Про вибір платформи та мови програмування

Для написання програми була обрана платформа Андроїд. Все лише задля зручності написання програми, оскільки вже два роки я працюю з цією платформою і вона мені добре знайома.

Всі алгоритми представлені в роботі програми (за виключенням залежних від платформи, таких як збереження до міді) зберігають свою актуальність як для мобільної так і для інших платформ, тому з часом програму можна перенести на комп’ютерну версію.

Базовою мовою програмування вибрано Kotlin – функціональну мову програмування, яка має суттєвий ряд переваг порівняно з Java. Одна з таких переваг – лаконічність – код на Kotlin має менше самоповторень та виглядає компактніше, тому його легше читати і він займає менше місця. Це лише одна з переваг цієї мови. Програмісти знають таких переваг значно більше.

2.4. Набір вимог перед початком написанням програми

Перед написанням програми потрібно чітко встановити вимоги яким вона має відповідати.

Це вимоги до функціоналу, якості коду та інтерфейсу користувача.

Вимоги які ми висували перед написанням програми:

- Програма має виконувати свою головну функцію − допомагати музиканту швидше формувати партії та легше вносити в них зміни. Має бути реалізована можливість вносити зміни на концептуальному рівні так, щоб ці зміни потім автоматично застосовувалися на рівні реалізації окремих партій. В програмі це реалізовано через наслідування партій та можливості накладання та заміни умов, на основі яких будується кожна партія.

- Має бути реалізована можливість збереження результатів програми в форматі, який дозволяє використовувати їх також в інших програмах. На практиці це реалізовано підтримкою міді-формату, для збереження окремих партій створених в програмі.

- Архітектура програми має бути грамотно спроектована, так щоб в подальшому її можна було легко розширяти, додавати нові функції та подібне. В програмі це реалізовано через застосування принципів SOLID, шаблону MVC та використання зрозумілого іменування компонентів програми, їх даних та функцій.

- Програма має бути оптимізованою. Це означає пошук кращих алгоритмів, пошук шляхів для використання меншого об’єму оперативної та довготривалої пам’яті девайса. На практиці це означає, наприклад, що при зміні умов в одній з партій, заново генеруватися має не вся композиція, а лише сегменти, залежні від зміненого сегменту партії.

2.5. Інтерфейс програми

Поговоримо про можливий інтерфейс програми та вимоги які до нього висуваються.

Основними вимогами до інтерфейсу завжди були його дружелюбність (зрозумілість для користувача) та ефективність (необхідність найменшої кількості дій для отримання необхідного функціоналу).

Інтерфейс нашої програми схожий на інші мобільні DAW. За зразок ми взяли інтерфейс програми FL Studio Mobile (рис. 2.6):

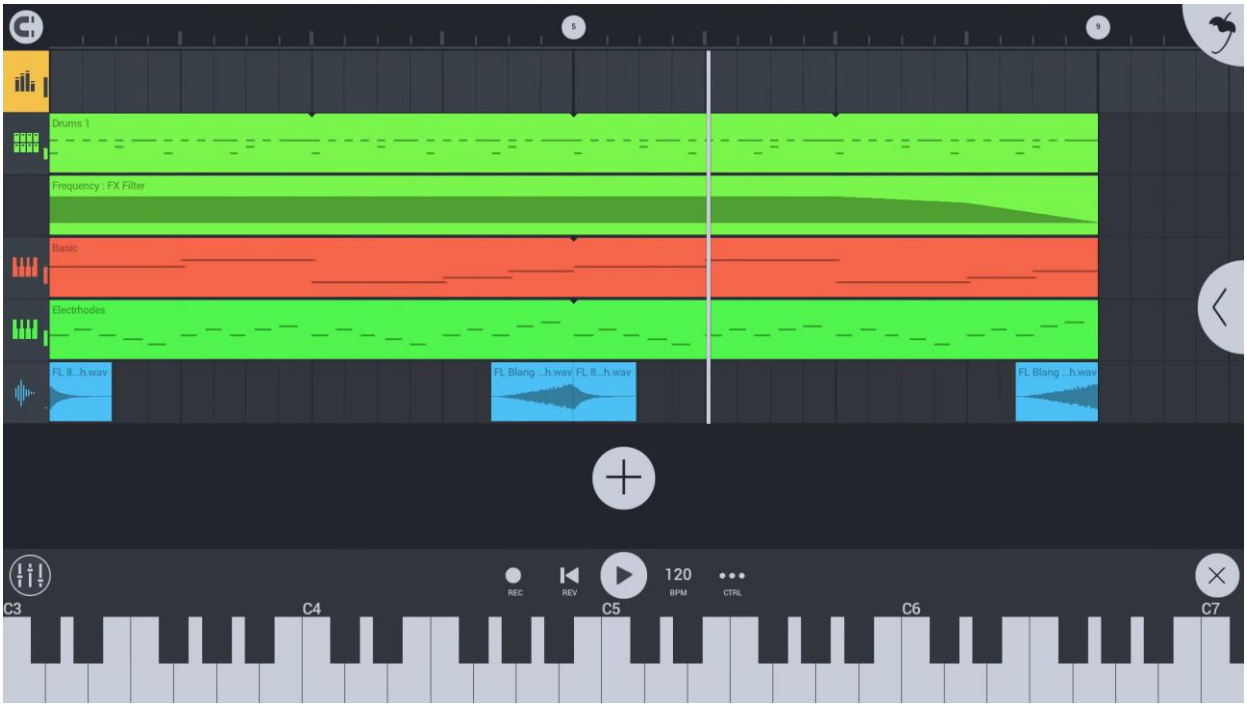

Рис. 2.6. Інтерфейс програми FL Studio Mobile

Чорновий варіант дизайну на папері (рис. 2.7):

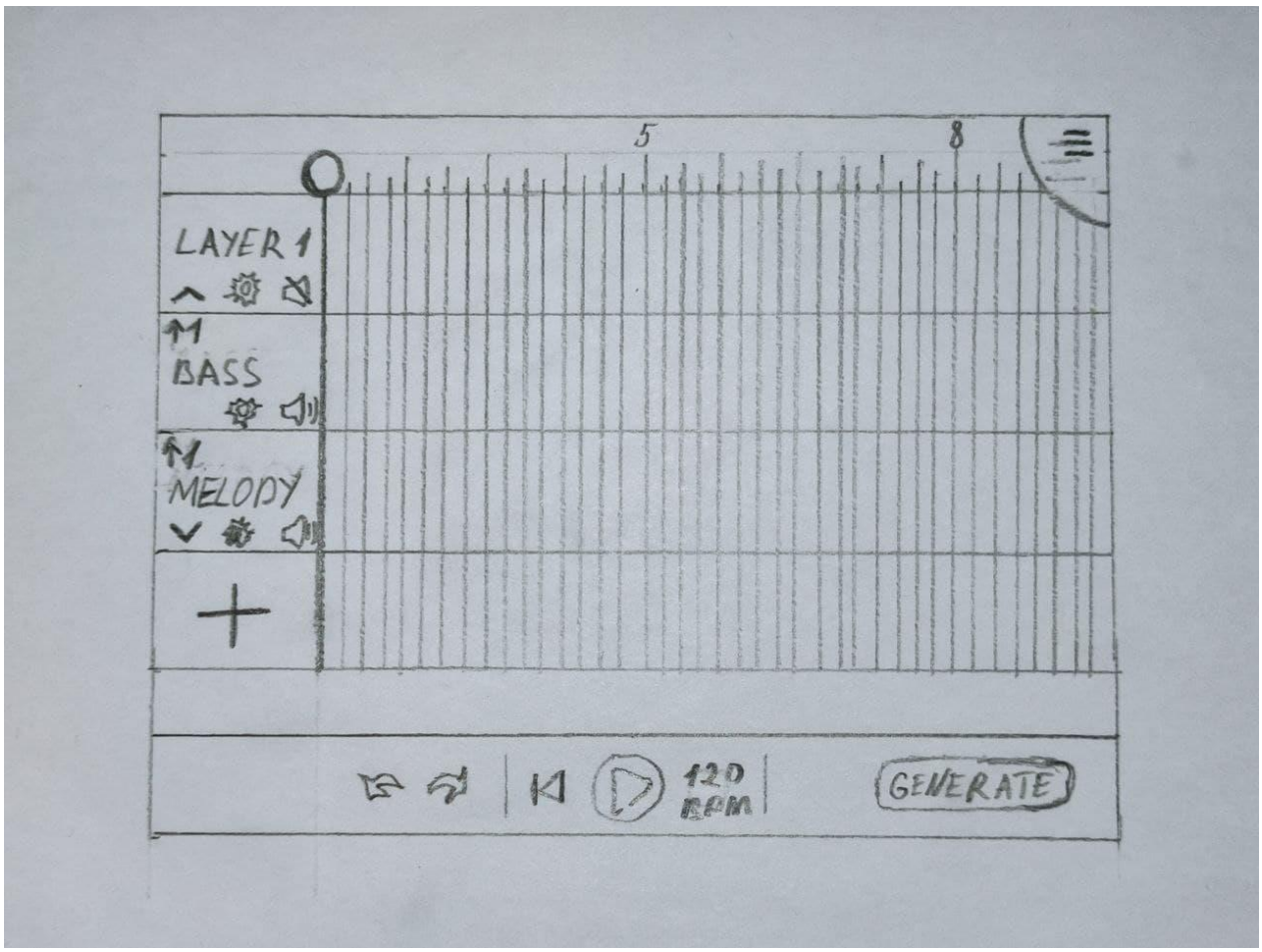

Рис. 2.7. Спрощений дизайн майбутньої програми

2.5.1. Огляд дизайну

Для кращого розуміння програми, корисно розглянути її дизайн.

Головний екран (рис. 2.8):

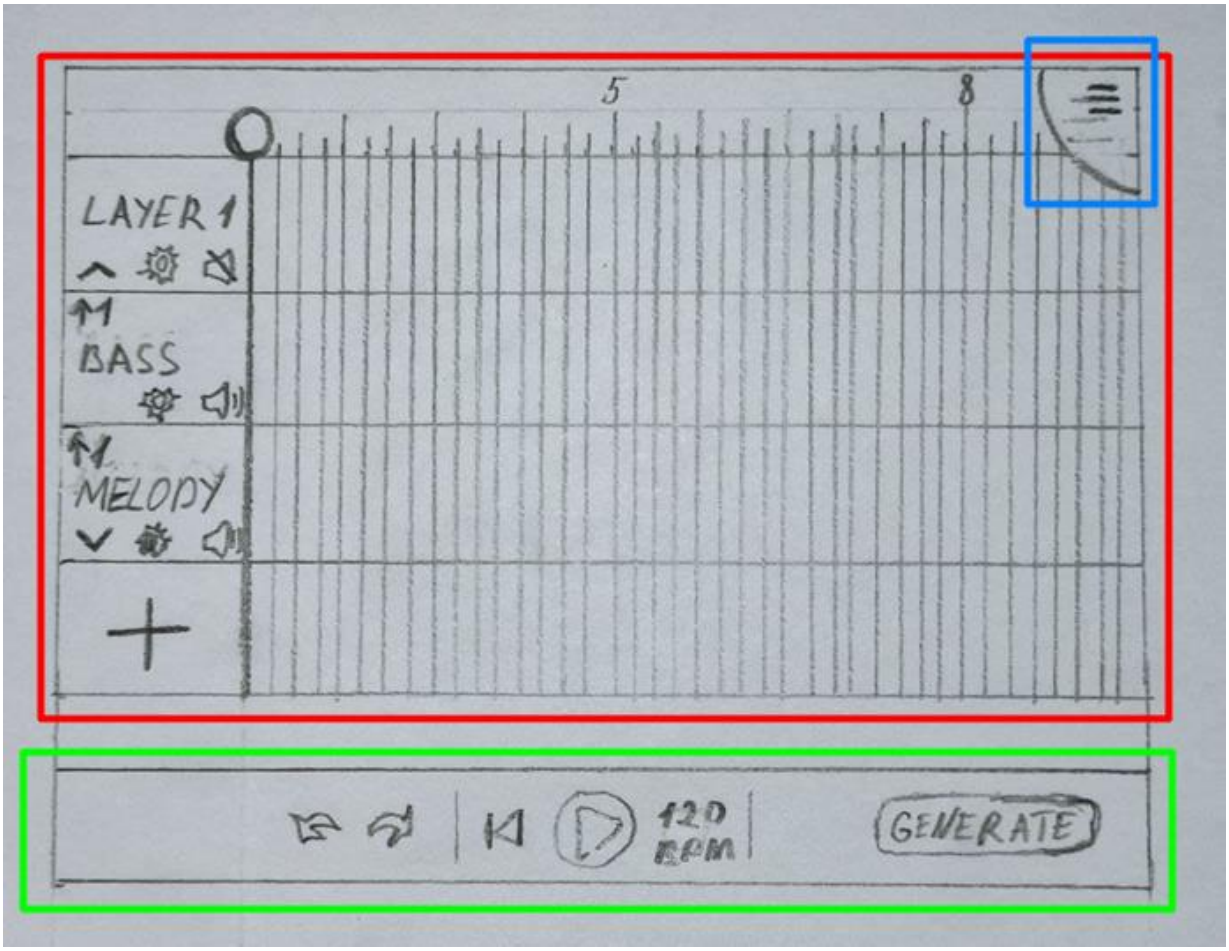

Рис. 2.8. Головний екран програми

Головний екран складається з:

- Панелі управління (зелений колір);
- Області редагування композиції (червоний колір);
- Кнопки виклику бокового меню (синій колір).

Панель управління

- Кнопки «*Undo*» / «*Redo*» – для відміни та повернення виконаних операцій;
- Кнопка «*Go to start*» – повертає курсор до початкової позиції композиції;
- Кнопка «*Play / Pause*» – при першому натисканні запускає відтворення, при повторному – призупиняє його;
- «*BPM*» – дозволяє встановити темп з яким буде відтворюватися композиція. Темп встановлюється в кількості четвертних нот на хвилину;

- Кнопка «*Generate*» − запускає процес генерації. Процес генерації запускається не автоматично після внесення змін, а тільки після натискання на цю кнопку. Вирішили так зробити, оскільки процес генерації може зайняти деякий час і тому користувач може хотіти проводити його комплексно, а не при кожній мінімальній зміні в умовах. Після запуску генерації, кнопка зникає і з’являється лише при внесенні нових змін до умов генерації.

Кнопка виклику бокового меню

При натисканні на цю кнопку, справа екрану буде виїздити бокове меню зі списком операцій:

- New – створити новий проект.

- Open – відкрити існуючий проект.

- Save – зберегти проект зі всіма налаштуваннями.

- Export to midi – відкриється вікно налаштувань збереження до міді.

- Settings – відкривається вікно налаштувань програми (на даний момент задаються за замовчування та не реалізовані).

- Close – закрити програму.

При збереженні до міді, можна обрати які зберігати.

Область редагування композиції (рис. 2.9):

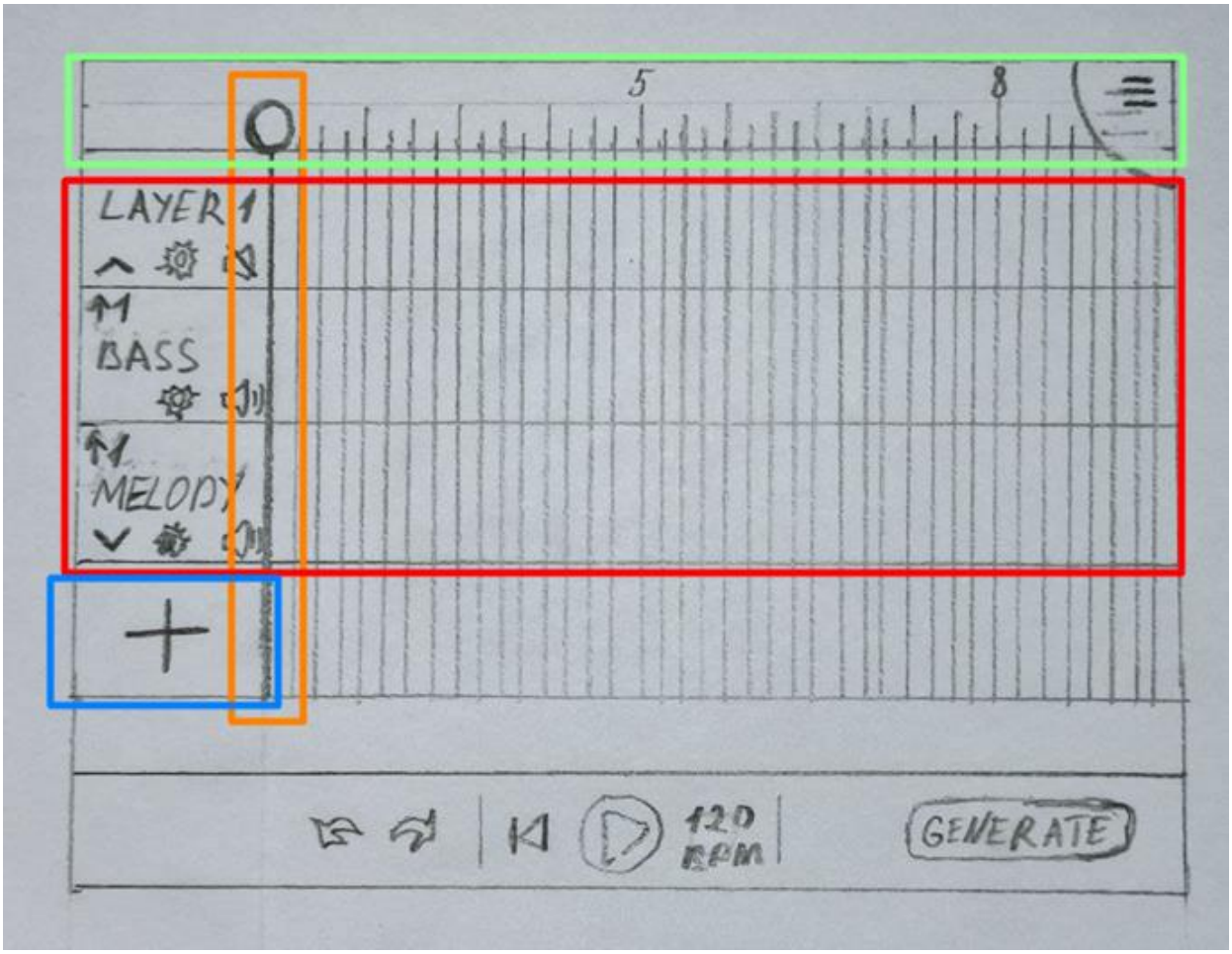

Рис. 2.9. Компоненти області редагування композиції

Складається з наступних елементів:

- Timeline (зелений колір);
- Треків (червоний колір);
- Курсору (помаранчевий колір);
- Кнопки «Add track» (синій колір).

Timeline

Лінійка часу відображає поділки на які ділиться область редагування (вертикальні лінії вздовж всієї області) та номери початку тактів (на малюнку, числа «5» та «8»).

Завдяки розтягуванню timeline можна змінювати масштаб області редагування (так само як і в більшості інших програм).

Курсор

Складається з вертикальної лінії вздовж всієї області редагування та кружка вгорі, за який курсор можна перетягувати по timeline.

Під час відтворення композиції курсор переміщається по області редагування та області timeline. Слугує індикації що зараз відтворюється.

Відтворення завжди розпочинається з поточної позиції курсора.

Кнопка «Add party»

При натисканні відкриває діалогове вікно налаштувань партії.

Я його ще не малював і це не обов’язково поки що. Головне, на тому діалозі можна буде обрати parent-партію для нової, ввести ім’я, обрати музичний інструмент яким партія буде озвучуватися.

Після додавання, нова партія розміщується останньою серед дочірніх партій обраної батьківської партії.

Трек

Трек складається з двох частин: заголовку та належної йому області редагування.

Заголовок треку (рис. 2. 10):

Рис. 2.10. Елементи заголовку треку

- Індикація ланцюжка батьківських партій. Якщо у партії є parent-партія, вгорі заголовку відображається стрілка вгору з числом. Число вказує на кількість наслідувань в ланцюгу наслідування даної партії. Кожна партія може наслідуватися лише від однієї батьківської, але батьківська в свою чергу теж може мати батьківську. Таким чином може існувати ланцюжок наслідування. В даному випадку біля індикації стоїть число «1». Це означає,

що наша партія наслідується від батьківської партії, котра ні від кого більше не наслідується. Якщо партія не має батьківської, індикація наслідування відсутня взагалі.

- Другою лінією відображається ім’я партії.

- На третій лінії відображаються кнопки управління (зліва направо):

«*Collapse*» / «*Expand*» − згортають або розгортають список дочірніх партій. Для економії простору на області редагування наявна можливість приховувати список дочірніх партій. Якщо дочірні партії відсутні, то кнопка також відсутня. Якщо дочірні партії розгорнуті, стрілка показує вгору, якщо згорнуті – вниз.

«*Settings*» − відкриває налаштування партії. В налаштуваннях можна змінити її ім’я, батьківську партію, а також відкрити редагування меж сегментів, про що мова піде пізніше.

«*Mute*» / «*Unmute*» − Вмикають або вимикають звук при відтворенні партії.

Область редагування треків

На області редагування розміщуються сегменти з яких складається композиція. Приклад розміщення сегментів показано на рис. 2.11:

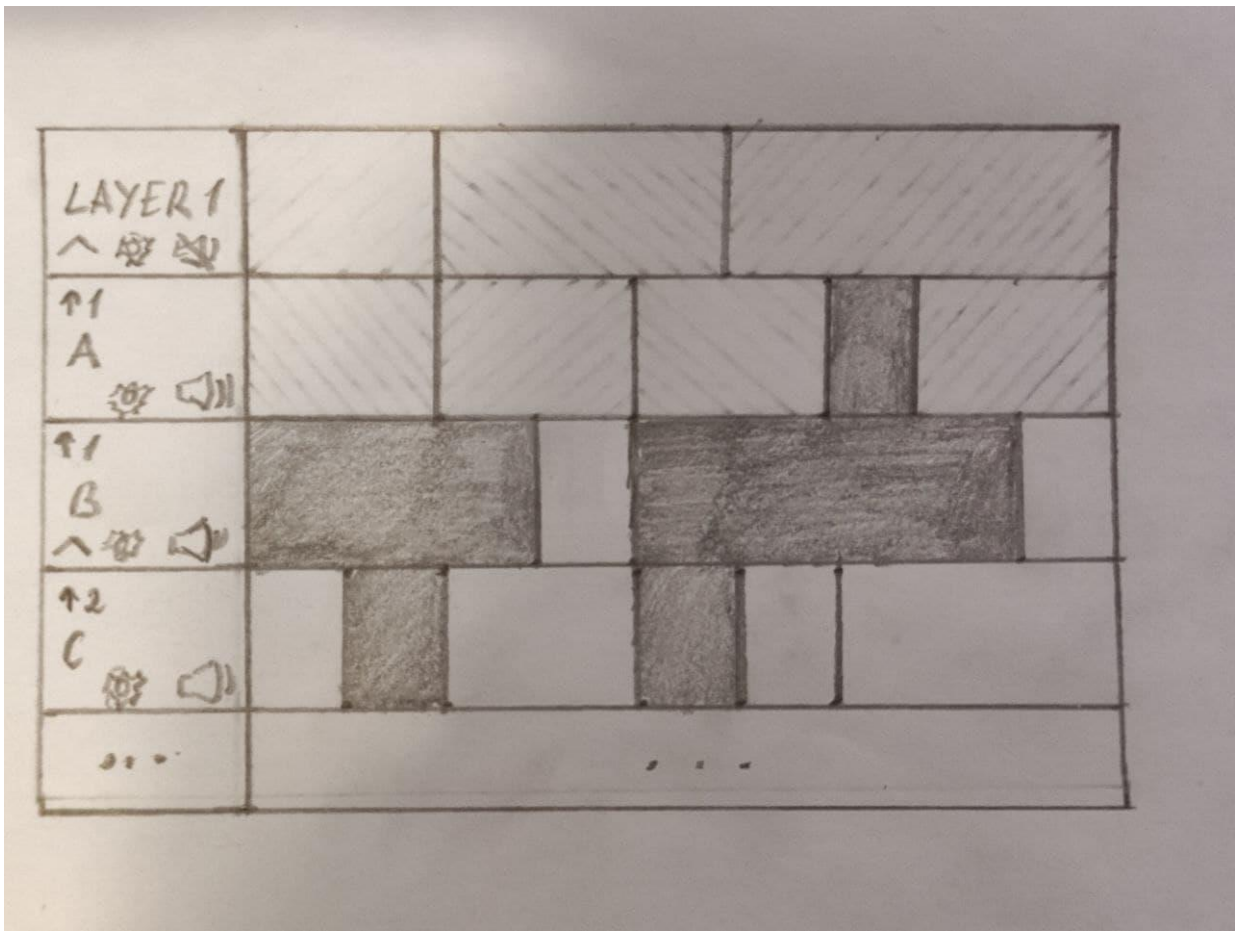

Рис. 2.11. Приклад розміщення сегментів на області редагування

Варто зазначити, що це той самий головний екран, але тут ми намалювали його в спрощеному вигляді. Насправді тут теж мають бути присутніми вертикальні лінії, курсор та лінія часу.

На рисунку вгорі відображаються чотири партії:

Layer 1 – батьківська партія. Оскільки вона не має parent-партії, то індикація наслідування у неї відсутня.

У цієї партії є двоє нащадків: «A» та «B». Від «B» в свою чергу наслідується «C».

Сегмент – це відрізок треку, яким має унікальний набір умов. На малюнку сегменти відображаються незафарбованими або в лінійку.

Повністю чорні області – це відсутність сегментів. Там де сегменти відсутні в партії – тиша, ніяких нот немає і жодної генерації не відбувається.

При створенні нової партії сегменти на ній відсутні. Щоб створити новий сегмент, потрібно перейти до налаштувань партії і там відкрити «редагування сегментів» (на малюнках воно відсутнє). При відкриванні редагування сегментів на головному екрані лишаються тільки заголовки треків, timeline та область редагування. При цьому зникають: курсор, кнопка виклику меню та панель управління. Натомість з’являються кнопки необхідні в процесі редагування, а саме:

- Кнопки «*Undo*» / «*Redo*».

- Кнопка «*Back*» − відміняє всі редагування і повертає головний екран до стандартного вигляду.

- Кнопка «*Apply*» − зберігає зміни та повертає головний екран до стандартного вигляду.

Щоб додати новий сегмент, потрібно просто натиснути на пусту область та тягнути в один з боків, сегмент буде збільшуватися.

Щоб змінити часові межі сегмента, потрібно тягнути за його початок або закінчення. Якщо закінчення одного сегмента повністю збігається з початком нового, то тягнути за межу між ними, їх часові рамки будуть зміщуватися взаємно. Щоб змінювати часові рамки незалежно, перед перетягуванням потрібно двічі натиснути на межу між сегментами і тільки потім тягнути. При цьому між сегментами буде з’являтися пустий простір.

Для видалення сегмента потрібно натиснути на нього. При цьому з’явиться кнопка «видалити» і натиснувши на неї сегмент буде видалено.

Всі зміни завжди можна відвернути користуючись кнопками «*Undo*» / «*Redo*». Також можна змінювати масштаб, користуючись *timeline*.

Накладання умов

Для накладання умов головний екран має бути в звичайному режимі (не в режимі редагування меж сегментів). Щоб відредагувати умови сегменту треку, потрібно натиснути на бажаний сегмент.

При цьому панель управління зникає, а знизу виїздить діалог зі списком вузлів генерації (рис. 2.12).

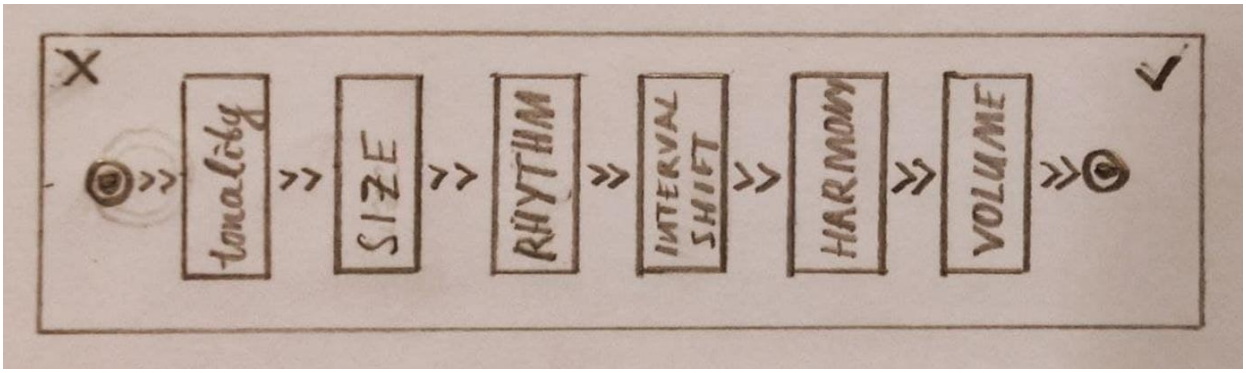

Рис. 2.12. Діалог з вузлами генерації

Детальніше про вузли генерації мова піде далі.

В спрощеному розумінні – вузол генерації – комірка для умов певного типу. На малюнку перераховані всі умови, які збираємося реалізувати в першій версії програми.

Це наступні умови:

1). Тональна

2). Метру

3). Ритмічна

4). Інтервальна

5). Гармонійна

6). Гучності

В дочірніх партіях відсутні вузли тональної та метричної умов, оскільки подібні характеристики на практиці не доводиться змінювати для дочірніх партій. Врешті, це можна зробити створивши ще один незалежний трек.

Розберемо з чого складається діалог вузлів генерації (рис. 2.13):

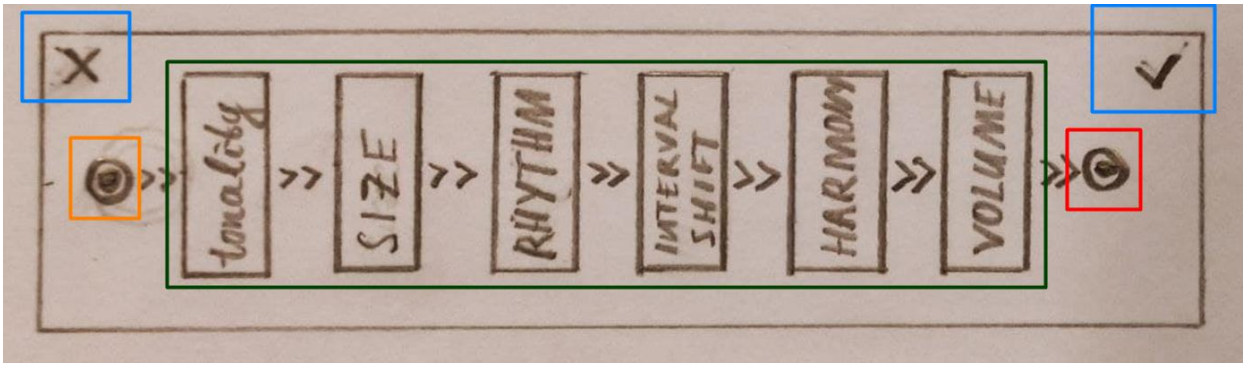

Рис. 2.13. Елементи діалогу вузлів генерації

Кнопки управління (синій колір)

«Close» − відміняє зміни застосовані до сегменту та закриває діалог.

«Apply» − зберігає зміни застосовані до сегменту та закриває діалог.

Вхід до сегменту (помаранчевий) − точка входу на яку подаються результати генерації parent-треку. Якщо у треку parent-трек відсутній, то вхід до сегменту в інтерфейсі теж не відображається.

Вихід з сегменту (червоний) – натиснувши на цей вузол відкривається список дочірніх треків.

Вузли накладання умов (зелений колір).

Кожен вузол накладання умов може містити умову тільки свого формату. Тобто ритмічний вузол не може містити умов інтервального вузла, і навпаки.

Частина вузлів може не застосовуватися і бути вимкненою (як краще вимикати, ми ще поки що не вирішили, але ймовірно можна додати перемикач одразу на блок кожного вузла).

Вузли тональності та метру можна замінити, але вимкнути неможливо, так як вони обов’язкові для застосування інших вузлів. І ці вузли задаються за замовчування. Їх значення за замовчання можна змінити в налаштуваннях програми. За стартового значення вони «До-мажор» та «4/4» відповідно.

Щоб обрати умову, яка буде накладатися у вузлі генерації, потрібно натиснути на вузол. Відкриється інший діалог (рис. 2.14):

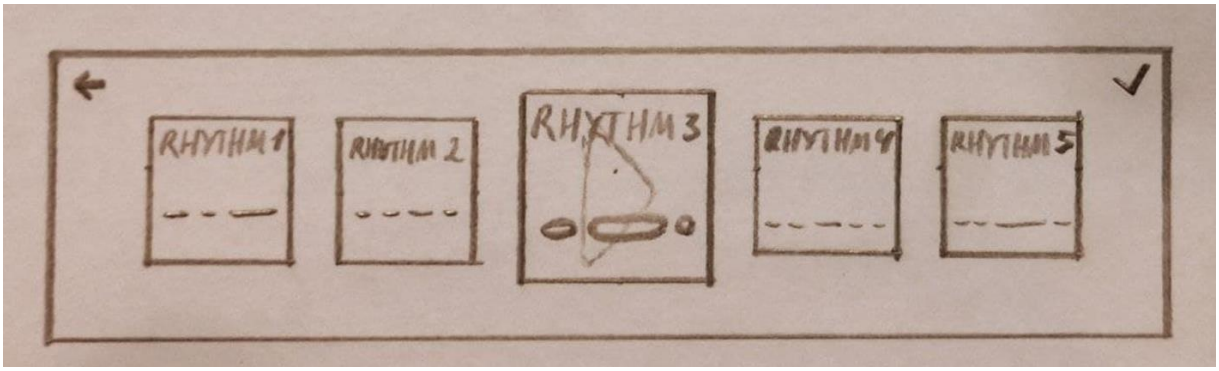

Рис. 2.14. Діалог вибору умови

Він схожий на попередній, але в ньому представлений набір шаблонів для умови яка буде зберігатися у вузлі відповідного типу. В даному випадку на рисунку відображаються ритмічні умови.

Набори умов можуть ділитися на каталоги (на рисунку не показано). Каталог може включати в себе або під-каталоги, або список умов. Щоб обрати умову потрібно помістити її по центру екрана. Це можна зробити натисканням на умову, або прокручуванням списку умов. Поточно обрана умова завжди відображається по центру і має більший розмір порівняно з іншими умовами. Щоб підтвердити вибір потрібно натиснути на кнопку підтвердження.

Коли умова розміщується по центру поверх неї малюється значок «Play», який пропонує відтворити умову.

В подальшому є плани додати для кожної умови можливість редагування та додавання власних умов. А також вікно з перед-прослуховуванням впливу умови.

Після того як умова вибрана повернеться діалог з вузлами. Змінені вузли будуть підсвічуватися іншим кольором. Вимкнені не підсвічуються.

Після закривання ботом шита з вузлами, якщо є зміни, на панелі управління буде відображатися кнопка «Generate». Після натискання на неї вона зникає і починається процес повторної генерації. Кнопка знову з’являється після впровадження будь-яких змін при яких потрібна повторна генерація.

2.6. Проектування архітектури

Backend частину програми можна розділити на два напрямки: компоненти музичної теорії та компоненти генерації.

2.6.1. Класи музичної теорії

Класи-компоненти музичної теорії базовими об’єктами з якими працює програма при накладанні умов, що зберігаються у вузлах генерації.

До компонентів музичної теорії відносяться: тональність, лад, розмір (метр), нота, акорд та партія.

Тональність

Якщо говорити просто, тональність в музиці можна вважати орієнтиром, який вказує, як ноти дозволено розміщувати по висоті.

Тональність в нашій програмі, так само як і в теорії музики, складається з двох ключових складових – ладу та тоніки.

Лад являє собою закономірність (правило) відбору доступних нот зі всього музичного діапазону. Наприклад, мажорний лад має схему – «т т п т т т п» (т – тон, п – півтон), а мінорний – «т п т т п т т». Окрім мінорного та мажорного ладу є безліч інших, наприклад, лідійський, міксолідійський, персидський та багато інших. Лади з’являлися як закономірності в музиці, яка формувалася століттями різними народами і оформилася в характерне звучання, що сприймається з певним забарвленням.

Тоніка – по суті, нота входу від якої починається рахунок ладу. Наприклад, мажорний лад побудований від ноти «До» має назву «До-мажор», а мінорний лад побудований від ноти «Ре», відповідно «Ре-мінор».

В залежності від ладу та тоніки ми відбираємо ноти, які входять в тональність. Ноти, що не входять до тональності будуть звучати

негармонійно, навіть якщо в інших тональностях вони звучать так як потрібно.

По суті лад – це правило гармонійного висотного розміщення звуків, за якого вони формують приємне забарвлення.

Метр

Метр в музиці більше всього співвідноситься з динамікою (пульсацією, побудованою на основі підкреслення одних нот гучніше, а інших – тихіше).

В програмі метр застосовується, в першу чергу, для застосування вузла «Volume».

Метр завжди позначається у формі співвідношення двох чисел, наприклад, «4/4». Нижнє число означає розмір нот, в яких вимірюється метр, а верхнє – їх кількість. Метр також вказує на розмір такту, наприклад, розмір 2/4 означає, що новий такт буде наступати через проміжки часу, рівні тривалості двох четвертних нот. Насправді це не випадково, так як позначення тактів в першу чергу слугує для вказання на сильну метричну долю. Тому такт завжди починається з сильної долі.

Те, що ще важливо знати про ритм, то це акцентування. Це якраз і є те підкреслення гучністю про яке ми говорили раніше. Не всі долі такту мають однаковий акцент, вони розбиваються на «сильні», «відносно сильні» та «слабкі» долі.

Нота

Нота має наступні атрибути: висота, початок звучання, тривалість та гучність.

В проекті ми ще зберігаємо в кожній ноті інформацію про тональність в якій вона була записана та метр. Це необхідно для простішого використання шаблонів.

Висота ноти представлена двома характеристиками – індексом октави та буквою. Завдяки цим значенням можна вирахувати абсолютне значення висоти ноти, так як в акустиці існує закономірність 2:1, де звучання ноти у вищій октаві = звучанню ноти в нинішній октаві помноженій на «2».

В програмі представлені наступні октави:

```
SUBCONTRACT,
CONTROVERSIAL,
LARGE_OCTAVE,
SMALL_OCTAVE,
FIRST_OCTAVE,
SECOND_OCTAVE,
THIRD_OCTAVE,
FOURTH_OCTAVE,
FIFTH_OCTAVE;
```

І наступні позначення нот зі значенням їх висоти в найнижчій октаві:

```
C(16.352),
Ces(17.324),
D(18.354),
Des(19.445),
E(20.602),
F(21.827),
Fes(23.125),
G(24.500),
Ges(25.957),
A(27.500),
Aes(29.135),
H(30.868)
```

Висота ноти визначається числом коливань фізичного тіла на секунду. Якщо тіло коливається не в вакуумі, то коливання тіла повітрям передаються до органів слуху, що зчитує та сприймає їх висоту.

Наприклад, тут помітно, що коливання ноти «Ля» (Європейське позначення «A») становить 27,5 коливань на секунду.

Нота є одиницею музичного матеріалу.

Акорд

Сукупність нот, що звучать одночасно. Ноти акорду можуть починати звучати не одночасно, а по черзі, але на письмі передаються як одне ціле.

В програмі акорди представлені базовою нотою (та нота, на основі якої за гармонійним шаблоном, побудовано акорд) та набором нот, що реалізують саме співзвуччя.

Партія

Набір нот та акордів, розташованих в часі.

2.6.2. Класи умов

Компоненти умов використовуються в алгоритмах генерації для перетворення одного музичного матеріалу, представленого компонентами музичної теорії (ноти, акорди, партії), до іншого.

Умови так само як і вузли генерації реалізовані наступних типів:

1) Ладові
2) Позначення ноти латинськими буквами
3) Метричні
4) Ритмічні
5) Інтервальні
6) Гармонійні
7) Гучності

Всі компоненти умов зберігаються у базі даних і можуть розширятися. Кожен такий компонент являє собою заготовку певного формату, яку можна помістити у вузол генерації того ж формату.

При генерації, вузол проводить процес парсингу (пошук та витягнення необхідної інформації з тексту), після чого накладає умову на партію передану йому в якості аргументу.

Розглянемо в якому форматі зберігаються різні умови в пам’яті програми.

Ладова умова зберігається в наступному форматі, наприклад: «+2|+2|+1|+2|-2|+2|-1». Де кожне число означає зміщення у півтонах.

Умова позначення нот зберігається в самій програмі і не може бути розширеною.

Ритмічний малюнок зашифрований в текстовій формі, має наступний вигляд: «32|64+0:6|8:6|16:6»

Графічно представлений запис виглядає як на рис.2.15:

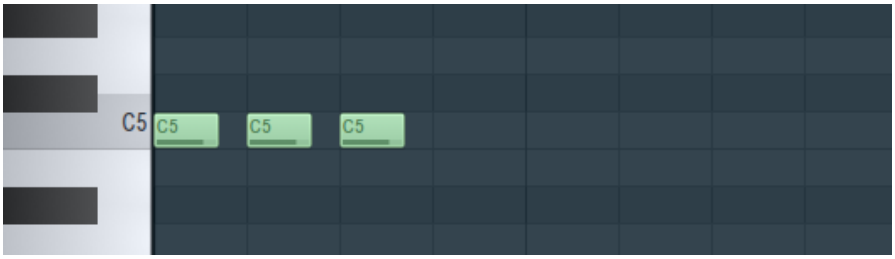

Рис. 2.15. Графічне представлення ритмічної умови «32|64+0:6|8:6|16:6»

Послідовність параметрів нот знаходиться справа від знаку «+»: «0:8|8:8|16:32». Параметри окремої ноти розділяються знаком «|», таким чином блоків опису окремих нот три: «0:8», «8:8», «16:32». В кожному з таких блоків зліва від «:» записана стартова позиція (номер імпульсу в PPQN), коли нота починає звучати, а справа – тривалість її звучання.

Зліва від знаку «+» знаходиться опис розміру партії: «32|64». Ці умови потрібні для забезпечення сумісності в разі незбіжності PPQN, в якому записаний ритмічний малюнок з PPQN, прийнятим в програмі.

Зліва від знаку «|» знаходиться PPQN розширення в якому записаний ритмічний малюнок – «32».

Справа від знаку «|» записана тривалість ритмічного малюнку – «64». Записувати тривалість необхідно, оскільки в ритмічному малюнку не вказуються паузи між нотами, тому не зрозуміло, коли після закінчення останньої ноти, мусить закінчитися ритмічний малюнок. Тривалість записується в тому ж розширенні. Тобто запис «64» свідчить, що ритмічний малюнок займає дві четвертні ноти.

Якщо допустити помилку та записати в умові більше бітів, наприклад, вказати стартову позицію більше 64-х, або тривалість, що виходить за межі 64-х, то при застосуванні умови, такі ноти будуть ігноруватися.

Для гармонійного та ладо-інтервально шаблонів формат зберігання умов однаковий, але одна і та ж інформація використовується в них по-різному. У випадку ладо-інтервального шаблона алгоритм послідовно проходить ноти наявної партії та встановлює їм нову висоту. У випадку

гармонійного шаблону, алгоритм додає нові ноти, що співпадають з нотою з наявної партії за стартом звучання та тривалістю, але зміщені за висотою.

Приклад шифрування умов ЛІ та гармонійного шаблонів: «0:0|0:4|1:0|1:4». Окремі ноти розбиваються знаком «|». Зліва від знаку «:» знаходиться зміщення в октавах відносно октави наявної ноти, справа – зміщення по ступеням. В обох випадках значення можуть бути додатними та від’ємними.

Не слід плутати зміщення в ступенях з інтервальним зміщенням. В даному випадку в записі «0:4» – «4» означає зміщення, а не розмір інтервалу. Тобто це означає, що алгоритм мусить піднятися на 4-ри ступені вгору по дозволеним в тональності нотам. Отриманий результат дорівнює інтервалу квінті, що на письмі позначається цифрою «5».

Також у кожного шаблону поруч з умовами зберігається ім’я, що передує умові та відділяється від неї знаком «*», тому, наприклад, повний ритмічний шаблон для розглянутої умови мав би вигляд:

«some_rhythm_1*32|64+0:6|8:6|16:6»

2.6.3. Класи генерації

Класи генерації використовують шаблони та компоненти музичної теорії і на їх основі створюють вихідний матеріал – партії, які потім конвертуються в міді.

До класів генерації належать: композиція, трек, сегмент, вузол.

Композиція

Композиція – це найвищий рівень генерації. Вона містить в собі ієрархію треків (спеціальну структуру, яка визначає, який трек від якого наслідується). Композиція відповідає також за збереження інформації про те, які саме сегменти треків варто заново згенерувати. І користуючись ієрархією, встановлює послідовність за якою це потрібно зробити.

Трек

Трек містить в собі набір сегментів та партію, яка є вихідним результатом останньої генерації.

Сегмент

Сегмент це контейнер для набору умов. Тобто в кожному сегменті зберігається набір упорядкованих вузлів генерації. сегмент потрібен лише на етапі генерації, так як генерування партії йде послідовно по всім сегментам.

Вузол

Вузол представляє собою робочу одиницю генерації. По своїй суті вузол має власний тип (наприклад, ритмічний) і може працювати лише з умовою, яка має той же тип (наприклад, в ритмічний вузол не можна помістити нічого окрім шаблону ритмічної умови).

При виборі умови в UI, вона поміщається до вузла. Далі в процесі генерації, коли до вузла надходить партія на редагування, ця умова

розшифровується (кожен вузол знає як розпарсити умову, яку зберігає) і накладається на існуючу партію, створюючи нову на її основі.

Загальний цикл генерації

Як він відбувається. Ми послідовно генеруємо всі партії, починаючи з незалежних і далі по ієрархії. Спершу всі незалежні, потім залежні першого рівня і далі. Так ми переконуємося, що необхідна для генерації найнижчих рівнів інформація уже була отримана на ранніх етапах генерації.

Висновки до розділу 2

1. MIDI протокол (Musical Instrument Digital Interface – цифровий інтерфейс музичних інструментів) – основний протокол для зберігання та передачі музичних даних на комп'ютері. Представлений у 1982 році. Інформація в ньому зберігається у вигляді послідовності команд, що в протоколі мають назву *події*. Кожна подія складається з команди, що саме потрібно виконати та часової мітки, яка вказує коли саме команда має спрацювати.

Вся інформація записується до міді файлу у вигляді двійкової послідовності. Для читання та редагування міді-файлів існують спеціальні програми, які мають назву міді-редактори. Всі сучасні віртуальні музичні студії володіють функціональними можливостями міді-редакторів.

2. Вхідними даними програми є набори завчасно заготовлених шаблонів. Кожен шаблон представляє собою певну концептуальну умову – ритмічний малюнок, інтервальну закономірність, гармонійну послідовність і подібне. Шаблони зберігаються в базі даних програми і можу бути в будь-який момент відредаговані чи замінені іншими шаблонами. Шаблони можна вважати конкретною реалізацією в певній області, але вони мають більш концептуальне значення, оскільки впливають лише на параметри певного типу, таким чином зберігаючи зв'язки між елементами в певній вузькій області.

3. Вихідними даними програми є міді файли партій, отриманих в результаті генерації.

4. При проектуванні програми ми висували для кілька основних вимог:

- Програма має уміти працювати з концептуальними зв'язками. Тобто зберігати їх та дозволяти користувачу маніпулювати ними в процесі роботи над композицією.
- Програма мусить реалізувати наслідування партіями.
- Алгоритми роботи програми мають бути ефективними та не виконувати зайвої роботи. Архітектура має бути прозорою та легко розширюваною.

РОЗДІЛ 3. ОПИС КЛЮЧОВИХ АЛГОРИТМІВ

3.1. Основні положення стосовно розробки алгоритмів

Найголовнішою частиною нашої роботи є її алгоритми, оскільки вони вирішують поставлені в роботі задачі.

При розробці алгоритмів ми користувалися положеннями наведеними в серії книг «Досконалий алгоритм» [26].

Основою цих алгоритмів є парадигма «розділяй та володарюй». Емпірично було доведено, що найвищого рівня ефективності алгоритми досягають при розподілі вирішуваної задачі на менші за розміром і їх рекурсивному розв’язанні. За такого підходу кількість операцій, які мають бути проведені для вирішення задачі, зменшується.

На цій парадигмі побудована велика кількість надефективних алгоритмів. Один з прикладів – алгоритм сортування вставками (Merge Sort), який покладений в основу і використовується в стандартних бібліотеках багатьох мов програмування (наприклад, в Java) для сортування елементів масивів, списків та інших структур даних.

При дотриманні парадигми «розділяй та володарюй» задачу розділяють на менші, а їх на ще менші, доки вони не стануть елементарними. Далі елементарні задачі вирішують рекурсивно, починаючи з найменших і підіймаючись вгору (збираючи вихідні дані кожного результату, і далі працюючи з ними) доки задача не буде вирішена повністю.

Ми спробували реалізувати свої алгоритми, спираючись на ідеї наведені в серії книг «Досконалий алгоритм» [26], а також на концепції SOLID-архітектури та чистого коду.

За цими концепціями алгоритми були відділені до окремих блоків і розміщені за призначенням. Таким чином кожен окремий алгоритм впровадження умов виступає незалежною одиницею, яка нічого не знає про інші алгоритми. Це дозволяє змінювати та тестувати кожен алгоритм незалежно від іншого коду.

Хоча будь-яка програма повністю побудована на алгоритмах, нас в першу чергу цікавлять саме алгоритми генерації, оскільки вони вирішують поставлену в дипломі задачу. Тому далі ми будемо розглядати всі головні алгоритми які використовуються в процесі генерації, і почнемо з глобального алгоритму генерації. Далі розглянемо алгоритми, що застосовуються кожним вузлом (як це працює стане ясно далі), а саме – ритмічний, гармонійний, інтервальний та інші. Ми розберемося як в процесі генерації на партію накладаються умови.

Ми не будемо зачіпати алгоритмів не пов’язаних з генерацією, тому пропустимо алгоритми відображення графічного інтерфейсу, взаємодії з базою даних, збереження файлів, конвертації в міді, відтворення міді-інформації та інших.

3.2. Глобальний алгоритм генерації

3.2.1. Реалізація ієрархії наслідування треків

Загальний алгоритм генерації відповідає за послідовну обробку треків композиції, всіх сегментів треку та всіх вузлів сегменту. При цьому треки генеруються в послідовності наслідування. Тобто спершу батьківські, а потім їх дочірні і так далі. Головна мета – забезпечити накладання умов, обраних користувачем, у правильному порядку.

Ми будемо розглядати цей алгоритм послідовно, крок за кроком і в процесі будемо пояснювати, як працює кожен компонент і з чого він складається.

Важливо зазначити, що алгоритм генерації існує в двох форматах – перша генерація, та повторна генерація після внесення змін. Ці алгоритми відрізняються і спершу ми розберемо тільки алгоритм генерації при першому запуску, а далі доповнимо його новими перевірками та умовами, щоб зробити більш ефективним при повторних запусках.

При запуску генерації ми звертаємося до класу Composition. Клас Composition є глобальним класом проекту та існує в єдиному екземплярі. В цьому класі знаходяться дані про треки, ієрархія їх наслідування, а також обслуговуючі функції, такі як *generate* (функція, яка запускає генерацію) та інші.

Тобто, цей клас слугує оболонкою всіх головних даних проекту та обслуговуючих ці дані функцій.

Ієрархія наслідування треків зберігається в окремій структурі **Hierarchy**. Ми вирішили не зберігати даних про батьківські та дочірні треки в об'єктах класу **Track**, оскільки це порушувало б принцип єдиної відповідальності, і в майбутньому могло б привести до складнощів в підтримці та розширенні архітектури. Тому винесли інформацію про наслідування в окремий клас – Hierarchy.

Hierarchy – це деревовидна структура даних, де кожен вузол може мати від нуля до нескінченної кількості дочірніх вузлів та мати або не мати батьківський вузол. Наслідування треків схематично можна уявити в наступному вигляді (рис. 3.1):

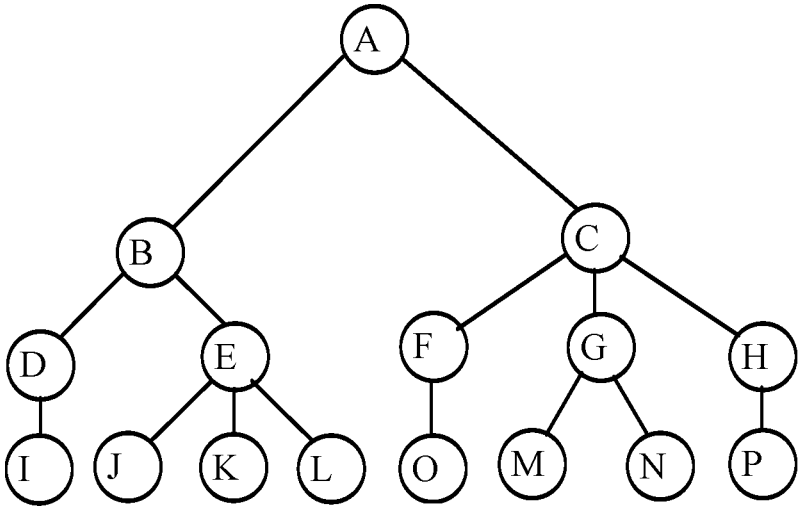

Рис. 3.1. Ієрархія наслідування треків

Де «A» − батьківський трек найвищого рівня. «B» та «C» – дочірні треки від «A». «D» та «E» – дочірні від «B». І так далі.

Екземпляр класу Hierarchy зберігається в Composition і слугує для навігації за треками.

Розглянемо код реалізації.

```
class HierarchyNode(
  val boundTrack: Track,
  var parent: HierarchyNode? = null,
  var children: MutableList<HierarchyNode> = mutableListOf()
) {

  fun addChild(node: HierarchyNode) {
    children.add(node)
  }

  fun removeChild(node: HierarchyNode) {
    children.remove(node)
  }

}
```

HierarchyNode – простий клас, що являє собою вузол дерева ієрархії.

В ньому зберігається посилання на прив'язаний трек *boundTrack*, посилання на батьківський вузол *parent*, який може містити значення *null* (значок «?» після імені типу) і за замовчування проініціалізований саме цим значенням. Список дочірніх вузлів *children*, який за замовчування пустий.

```
class Hierarchy(
    val topNodes: MutableList<HierarchyNode> = mutableListOf(),
    val map: MutableMap<String, HierarchyNode> = mutableMapOf()
) {

    fun findParentTrackOf(track: Track): Track? {...}

    fun add(parent: Track?, track: Track) {...}

    fun remove(track: Track) {...}

    fun changeParent(track: Track, newParent: Track) {...}

    fun getSortedInDeepTracks(): List<Track> {...}

    fun collectNodesInDeep(
        root: HierarchyNode,
        list: MutableList<Track>
    ) {...}

}
```

Рис. 3.2. HierarchyNode – простий клас, що являє собою вузол дерева ієрархії

Клас Hierarchy містить два поля – список вузлів ієрархії найвищого рівня *topNodes* та мапу *map* з ключем – ID треку, та значенням HierarchyNode. Мапу додали для швидшого пошуку треку за ID, оскільки в цій структурі даних отримання значення за ключем здійснюється швидше ніж ручний пошук за ID в ієрархії.

Розглянемо призначення та реалізацію функцій.

Функція findParentTrackOf

Функція повертає батьківський трек для треку переданого у вигляді аргументу. Якщо трек не має батьківського вузла, функція повертає *null*.

```
fun findParentTrackOf(track: Track): Track? {
  val node = map[track.id]
  if (node != null) {
    return node.parent?.boundTrack
  }
```

```
  return null
}
```

Функція add

Функція додає новий трек до ієрархії, встановлюючи для нього батьківський трек. Якщо батьківський трек *null*, новий трек додається до списку незалежних треків *topNodes*:

```
fun add(parent: Track?, track: Track) {
  val node = HierarchyNode(track)
  if (parent == null) {
    topNodes.add(node)
  } else {
    node.parent = map[parent.id]
    node.parent?.addChild(node)
  }
  map[track.id] = node
}
```

Функція remove

Видаляє трек з ієрархії.

```
fun remove(track: Track) {
  val node = map.remove(track.id)
  if (node?.parent != null) {
    node.parent?.removeChild(node)
  }
}
```

Функція changeParent

Змінює parent-трек для треку.

```
fun changeParent(track: Track, newParent: Track) {
  remove(track)
  add(newParent, track)
}
```

Функція getSortedInDeepTracks

Викликає collectNodesInDeep для всіх вузлів верхнього рівня (незалежних треків) та формує список відсортованих «в глибину» вузлів.

```
fun getSortedInDeepTracks(): List<Track> {
  val tracks = mutableListOf<Track>()
  for (node in topNodes) {
    collectNodesInDeep(node, tracks)
  }
  return tracks
}
```

Функція collectNodesInDeep

Рекурсивно додає вузли до списку list, рухаючись «в глибину» по вузлам ієрархії.

```
fun collectNodesInDeep(
  root: HierarchyNode,
  list: MutableList<Track>
) {
  list.add(root.boundTrack)
  for (node in root.children) {
    collectNodesInDeep(node, list)
  }
}
```

Пояснимо що означає термін «відсортованих в глибину». Як вже говорилося, в процесі генерації нам важливо першими згенерувати батьківські партії і тільки потім – дочірні. Це важливо, оскільки, як ми побачимо далі, проміжними даними при генерації дочірніх треків, виступають партії, що зберігаються в батьківських треках (нагадаємо що в кожному треку (*Track*) у вигляді партії (*Party*) зберігається останній результат генерації цього треку).

Таким чином, якщо дочірня партія якимось чином буде згенерована раніше батьківської, то результати її генерації будуть хибними, оскільки спираються на застарілі дані.

Правильну послідовність треків на генерацію можна зібрати різними способами. Якщо повертатися до попереднього рисунку, то сортування в глибину означає, що треки будуть зберігатися в список в такому порядку:

Спершу буде додано глобальний трек «A», оскільки він знаходиться на самій вершині ієрархії.

1. Потім буде додано перший дочірній трек (в нашому випадку «B»).

2. Далі буде перевірка, чи є в цього треку ще дочірні, і якщо є – будуть додані вони, якщо немає, буде доданий наступний трек наслідник «A»

3. Таким чином ми просуваємося в глибину за вузлами, зберігаючи при цьому порядок генерації.

Виходячи з зображення, результуючий список мав би такий вигляд: A, B, D, I, E, J, K, L, C, F, O, G, M, N, H, P.

3.2.2. Генерація на рівні композиції

Для запуску процесу генерації ми звертаємося до функції generate класу Composition (рис. 3.3).

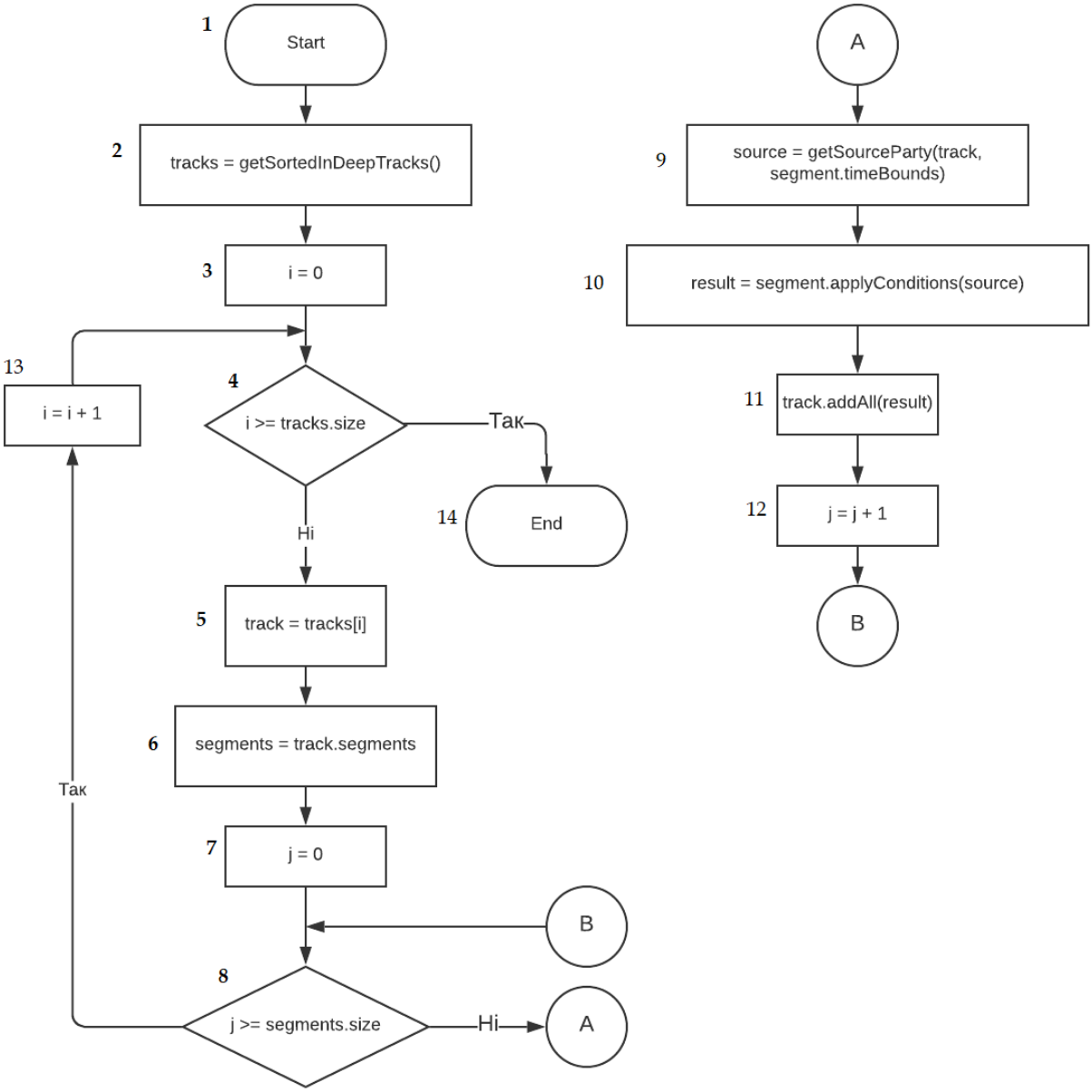

Рис. 3.3. Блок схема алгоритму генерації на рівні Composition

```
fun generate() {
  val tracks: List<Track> = hierarchy.getSortedInDeepTracks()
  for (track in tracks) {
    for (segment in track.segments) {
      val source: Party = getSourceParty(track, segment.timeBounds)
      val result: Party = segment.applyConditions(source)
      track.addAll(result)
    }
  }
}
```

В наведеному фрагменті коду:

Створюємо нову змінну *tracks*, в яку зберігаємо відсортований «в глибину» список треків. В циклі проходимося послідовно по всім трекам. У внутрішньому циклі для кожного треку проходимося по всім його складовим сегментам.

Отримуємо «базову» партію викликаючи функцію *getSourceParty*(…), передаючи до неї аргументами поточний трек *track* та часові обмеження сегмента *segment.timeBounds*. TimeBounds – простий клас обгортка з двома цілочисленними полями *start* та *end*.

```
class TimeBounds(
  var start: Int,
  var end: Int
)
```

Викликаємо функцію applyConditions(source), об’єкта segment, передаючи аргументом партію отриману на минулому кроці. Результат виклику функції зберігаємо у змінну *result*.

Викликаємо *track.addAll(result)*. Функція *addAll* додає всі ноти партії переданої у вигляді аргументу, до внутрішньої партії треку.

В коді ми ще звертаємося до функції *getSourceParty(…)*, яка надає партію для подальшого накладання умов.

```
fun getSourceParty(track: Track, bounds: TimeBounds): Party {
  val parent = hierarchy.findParentTrackOf(track)
  if (parent != null) {
    return TrackUtils.getPartyFragmentOf(track, bounds)
  } else {
    return TrackUtils.generateSingleNoteParty(bounds)
  }
}
```

В коді спершу намагаємося отримати батьківський трек, звертаючись до функції *findParentTrackOf(…)*. Цю функцію ми розглядали раніше і знаємо як вона працює.

Далі перевіряємо *parent*. Якщо він не *null*, звертаємося до допоміжного класу TrackUtils, який не містить жодних даних, а лише набір допоміжних функцій, необхідних в процесі генерації.

Функція *TrackUtils.getPartyFragmentOf(track: Track, bounds: TimeBounds)* «вирізає» з *track* фрагмент його партії, обмежений *bounds* за часом. Якщо нота партії лежить на межі *bounds*, вона обрізається так, щоб повністю знаходитися в межах. Важливо зазначити, що дана функція не повертає список нот з партії треку, а створює нові. Тобто при їх редагуванні, батьківська партія змінюватися не буде.

Якщо *parent* має значення *null*, звертаємося до методу *TrackUtils.generateSingleNoteParty(bounds)*, який генерує партію з єдиною нотою. У цієї ноти присутня інформація лише про початок звучання та тривалість і відсутня вся інша інформація (висота, гучність і подібне). Інформація про часові обмеження ноти дістаються з *bounds*.

Якщо розглядати те що відбувається в функції на концептуальному рівні програми, це означає, що в процесі генерації, для треків найвищого рівня (які не мають parent-треків) за замовчування буде генеруватися «*signle note party*».

Справа в тому, що в процесі накладання умов потрібно їх десь зберігати. Ми вирішили, що всі умови будуть зберігатися в параметрах самої ноти. Таким чином на момент застосування найпершої умови нам уже потрібно мати хоча би одну ноту.

Тривалість же ноти встановлюється за замовчування лише для того, щоб далі застосовувати ритмічний шаблон завжди в одному форматі, а саме – фільтруючому.

В першій версії бекенду програми існувало два різновиди ритмічного шаблону – породжуючий та фільтруючий. Перший створював ритм з нуля, а другий обрізав вхідні ноти за ритмічною умовою. Наявність двох режимів одного вузла вносило плутанину до того ж алгоритм майже не відрізнявся, що зумовлювало наявність повторень в коді.

За нового підходу на самому старті генерації нота має часові рамки, тому режим породжуючого ритму став більше не потрібен.

3.2.3. Генерація на рівні сегмента

Послідовно обходячи сегменти треку ми викликаємо у них функцію *applyConditions(party: Party)*. Розберемося детально що в ній відбувається.

Як вже відомо, треки (*Track*) складаються з сегментів (*Segment*), а сегменти, в свою чергу, з вузлів (*Node*). Генерація на рівні сегменту розпочинається з виклику функції *applyConditions*. Аргументом функції передається відрізок партії. Сегмент не знає що саме це за відрізок і просто послідовно проходить по всім вузлам. Змінна *party* оновлює своє значення

```
fun applyConditions(source: Party): Party {
    var party = source
    for (node in nodes) {
        party = node.execute(party)
    }
    return party
}
```

після застосування умов кожного вузла генерації. В результаті своєї роботи функція повертає партію згенеровану останнім вузлом (рис. 3.4).

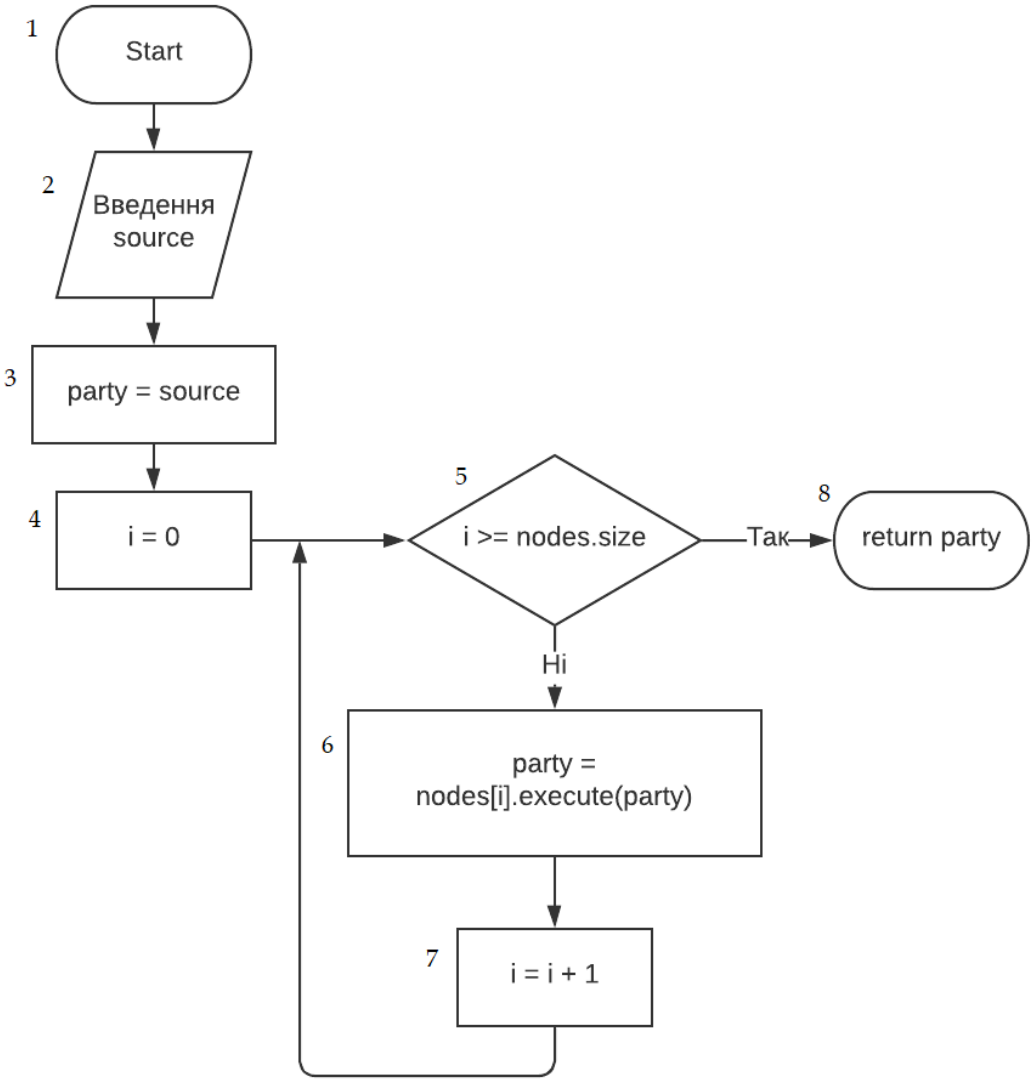

Рис. 3.4. Блок-схема генерації на рівні сегменту

Варто нагадати, що не всі треки мають однаковий набір умов. Сегменти незалежних треків мають додатково два вузла генерації – вузол *тональності* та вузол *метру*, які застосовуються раніше всіх інших і мають значення за замовчанням.

Ці вузли автоматично додаються до списку *nodes*, коли трек встановлюється незалежним і видаляються зі списку, якщо трек набуває статус дочірнього, тому на етапі застосування вузлів додатково фільтрувати їх не потрібно.

3.2.4. Генерація на рівні вузла

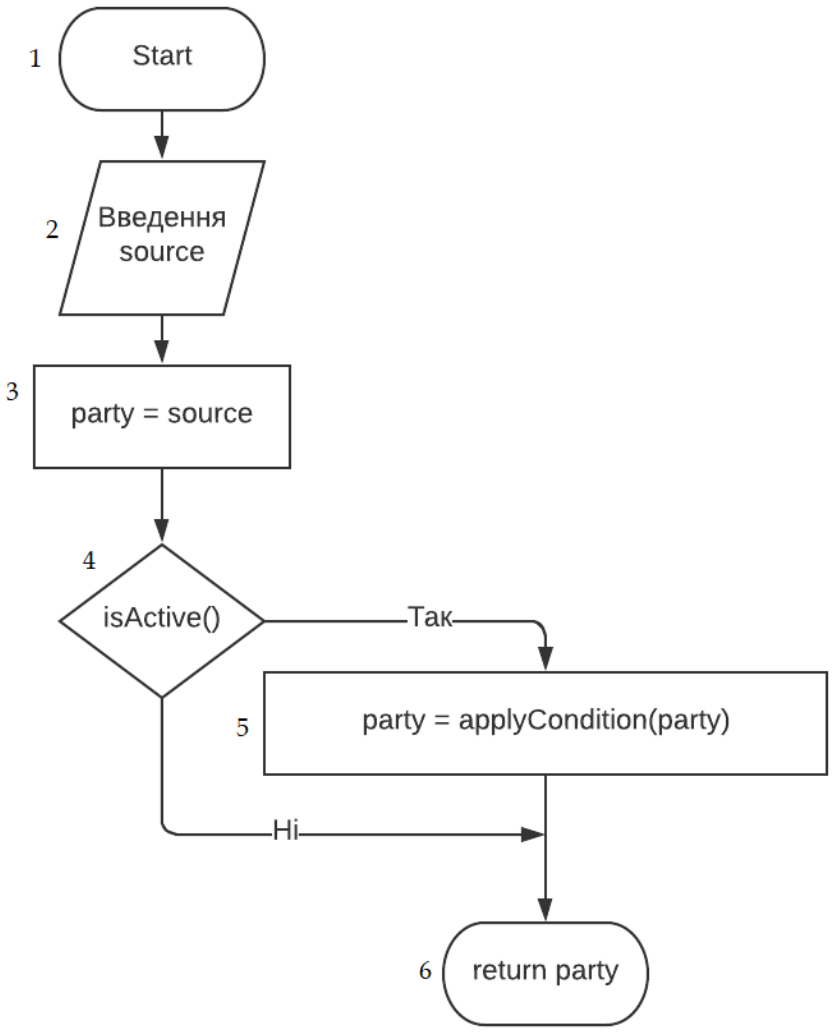

Рис. 3.5. Блок-схема генерації на рівні вузла

На рівні вузла генерація починається з виклику функції *execute* (рис.3.6):

```
fun execute(source: Party): Party {
   if (isActive()) {
      return applyCondition(source)
   }
   return source
}
```

Рис. 3.6. На рівні вузла генерація починається з виклику функції *execute*

Спершу перевіряється активність вузла. Умова активності вузла залежить від двох факторів – чи увімкнено вузол в UI та чи містить вузол певну умову. Якщо вузол не містить умови, він автоматично перестає бути активним і на UI і в процесі генерації. При цьому в процесі генерації вузол ніяк не взаємодіє з партією, що передається йому у вигляді аргументу, а просто повертає її в тому ж вигляді, в якому вона прийшла на вхід.

Якщо ж вузол містить умову та увімкнений на UI, викликається інший його метод – applyCondition, який відрізняється за реалізацією для кожного типу вузла.

Важливо зазначити, що в процесі генерації ми скрізь де маємо вносити зміни, насправді створюємо нову партію на основі існуючої. Цим пояснюється, чому кожен метод накладання умов повертає значення.

Такий підхід заснований на одній з парадигм функціонального програмування, а саме – *незмінюваності даних*. Багато проблем в роботі програм виникає по причині неакуратного використання даних та їх модифікації з різних місць. Тому за можливості дані варто робити незмінюваними. Кожен вузол не змінює партію яку отримує у вигляді аргументу, а створює на її основі нову, копіюючи ноти з *source* та надаючи копіям нові характеристики.

3.2.5. Загальний огляд алгоритму генерації

Підсумуємо все, що дізналися раніше.

Для проекту є глобальна точка входу – це об’єкт класу *Composition*, який містить ієрархію треків та запускає генерацію.

Ієрархія треків зберігається в спеціальній структурі, представленій класом *Hierarchy*. *Hierarchy* складається з вузлів ієрархії – *HierarchyNode*, кожен з яких може мати один батьківський вузол та багато наслідників, і *обов’язково* містить посилання на трек, який представляє в ієрархії.

Генерація розпочинається при виклику метода *generate* об’єкту *Composition* і має наступний алгоритм:

1. Обходимо всі треки по порядку ієрархії.

2. Для кожного трека послідовно проходимо по кожному сегменту.

3. Якщо для треку є parent-трек, дістаємо з нього відрізок партії для редагування — TrackUtils: *getTrackPartyFragment(parentTrack, bounds)*. Якщо трек незалежний, генеруємо для сегмента партію "пустої" ноти, користуючись іншим методом того ж класу TrackUtils: *generateSingleNoteParty(bounds)*.

4. getTrackPartyFragment() — вирізає фрагмент існуючої партії треку, в межах *bounds*.

5.generateSingleNoteParty() — генерує ноту, тривалістю *bounds*, яка не має висоти, гучності та інших параметрів, але містить інформацію про початкову та кінцеву позиції звучання ноти.

6. Отриману партію передаємо до сегменту segment.applyConditions(party). Сегмент постійно заміняє її при кожному виклику вузла генерації, передаючи у вузол параметром.

Результатом обробки сегмент повертає партію, створену останнім вузлом генерації.

Вузол перевіряється на активність, і якщо активний, застосовується умова, якщо не активний — повертається партія передана параметром.

Метод *applyCondition* перевантажується для кожного дочірнього класу, який уміє працювати з умовами певного типу. В кожному з цих класів створюється новий екземпляр класу *Party*, який заповнюється об'єктами *MusicItem* (базовий клас для класів *Note* та *Chord*).

Результат застосування сегмента зберігається в поточний трек: *track.addAll(party)*.

3.3 Алгоритми накладання умов

3.3.1 Шаблони умов

Розберемося докладніше з логікою накладання умов.

В проекті на даний момент представлені 5 типів умов:

- тональна;
- метрична;
- ритмічна;
- інтервально- висотна закономірність;
- гармонійна.

Шаблони всіх умов зберігаються в базі даних в зашифрованому вигляді.

Приклад шифрування ми наводили в попередньому розділі. Для ритмічної послідовності вигляд шаблону в базі даних може мати, наприклад, наступний вигляд:

*some_rhythm_1*32|64+0:6|8:6|16:6*

Це звичайна строка, в якій за певними правилами зашифровано шаблон ритмічного малюнку. Розглянемо з чого вона складається:

some_rhythm_1– ім’я шаблону, яке також служить його унікальним ідентифікатором в базі даних.

32|64 – перша цифра – розширення в якому записана умова (в PPQN). Друга цифра – тривалість фрагменту (також в PPQN).

0:6|8:6|16:6 – ритмічна умова, де окремі ритмічні одиниці розбиті знаком «|». Зліва від знаку «:» знаходиться стартова позиція ритмічної одиниці (PPQN). Справа – її тривалість (PPQN).

Термін «ритмічна одиниця» означає спрощену ноту, в якої немає жодної інформації окрім часових характеристик. В проекті кожна така одиниця конвертується до об’єкту класу TimeBounds.

Коли користувач відкриває каталог умов певного типу (як відбувається вибір умов на UI розповідалося в минулому розділі), ці умови підтягується з БД в зашифрованому вигляді. Далі йде процес дешифрування, в результаті

якого дані зберігаються до відповідного класу, щоб з ними можна було швидко взаємодіяти в подальшому.

Наприклад, ритмічна умова після дешифрування зберігається в класі RhythmCondition, який має наступний вигляд:

```
class RhythmCondition(
  val sequence: List<TimeBounds>,
  val size: Int
)
```

RhythmCondition місить список екземплярів класу *TimeBounds* (ритмічні одиниці) та інформацію про тривалість ритмічного шаблону − *size*. З об'єктом цього класу зручніше взаємодіяти в коді, ніж з зашифрованими даними. Умови інших типів зберігаються у схожих класах-обгортках.

При виборі умови на UI вона зберігається до відповідного вузла, який знає як її обробити. Наприклад, RhythmCondition буде поміщено до RhythmNode (цей клас будемо розглядати пізніше при огляді алгоритму накладання ритмічної умови).

3.3.2. Архітектура базового класу Node

Розглянемо архітектуру базового класу *Node*:

```
abstract class Node<Condition>(
  var condition: Condition? = null,
  var isEnabled: Boolean = false,
) {

  fun execute(source: Party): Party {
    if (isActive()) {
      return applyCondition(source)
    }
    return source
  }

  abstract fun applyCondition(source: Party): Party

  fun isActive() = isEnabled && condition != null

  fun requireCondition() = condition!!
}
```

Почнемо з оголошення імені класу. Як видно з заголовку − *Node* параметризований клас. Про це свідчить передача параметра в трикутних дужках справа від імені класу (<*Condition*>). Це означає, що для кожного класа-наслідника *Node* змінна *condition* матиме різний тип.

Клас *Node* включає два поля:

condition – розшифрована умова в класі-обгортці

isEnabled – логічна змінна, яка індикує чи увімкнений вузол на UI

Розглянемо також методи класу:

execute – функція, що викликається сегментом в процесі генерації. Її ми розглядали раніше

applyCondition – функція, що не має реалізації в базовому класі, але реалізована у всіх наслідників. Відповідає за створення нової партії, на основі вхідної з накладеними умовами.

isActive – індикує чи є вузол активним. Активним вузол вважається, коли він увімкнений в UI (*isEnabled == true*) та містить умову (*condition != null*).

requireCondition – повертає не-нульове значення *condition*. Якщо condition == null, кидає виключення (*NullPointerException*).

В процесі генерації, сегмент послідовно викликає метод *execute* у кожного вузла. Якщо вузол активний, запускається конкретна реалізація методу *applyCondition*.

Далі розглянемо кожен вузол більш детально.

3.3.3. Тональність

Вузол тональності присутній лише в *незалежних треках*. За замовчування він налаштований на тональність *C-maj*.

В проекті представлений класом *TonalityNode* (рис. 3.7):

```
class TonalityNode: Node<TonalityCondition>() {
    override fun applyCondition(source: Party): Party {...}
}
```

```
class TonalityCondition(
    val tonality: Tonality
)
```

```
class TonalityNode: Node<TonalityCondition>() {

  override fun applyCondition(source: Party): Party {
    return Party().apply {
      val condition = requireCondition().tonality
      for (index in 0 until source.items.size) {
        val item = source.items[index].copy()
        item.setTonality(condition)
        item.setHeight(Height(
          FIRST_OCTAVE,
          condition.tonic
        ))
        add(item)
      }
    }
  }
}
```

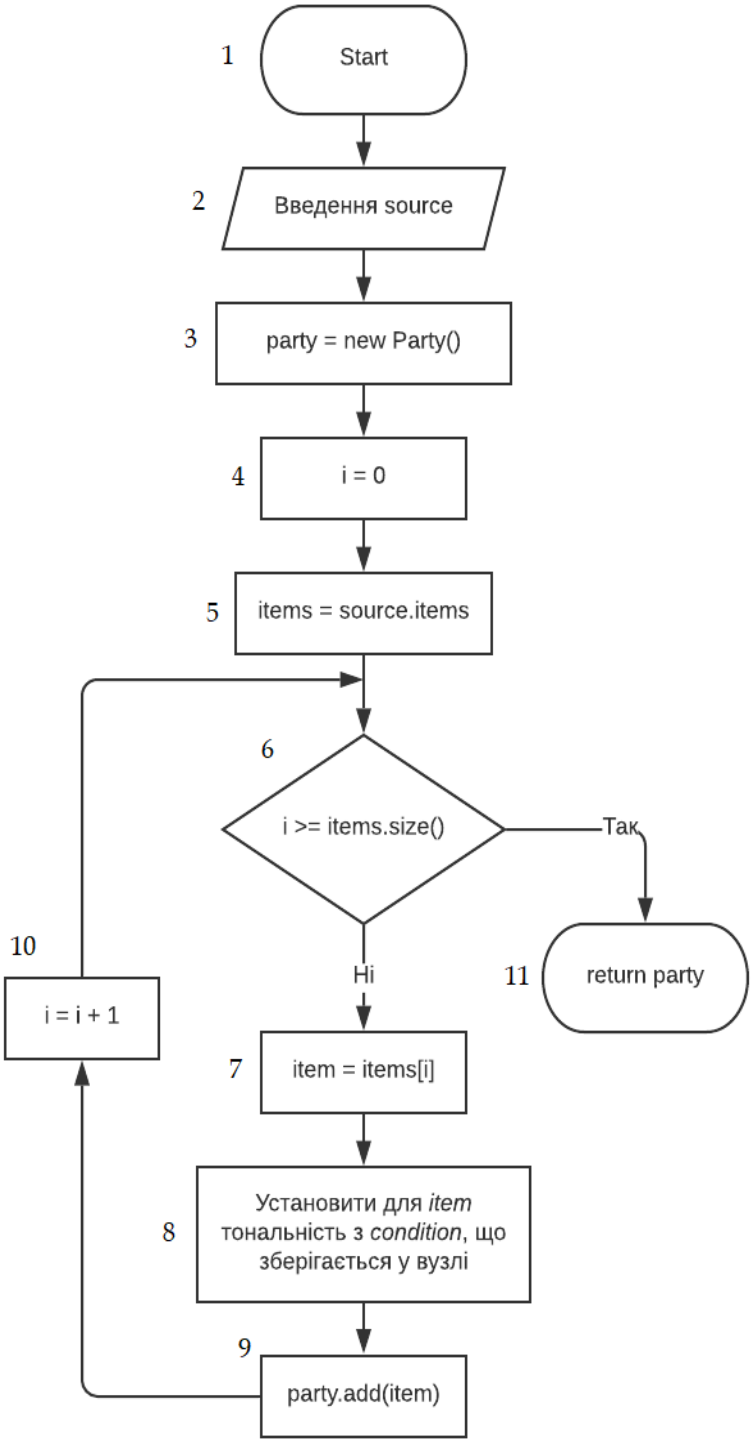

Рис. 3.7. Блок-схема накладання тонального шаблону

Дістаємо умову з класу обгортки.

Послідовно проходимося по всім елементам партії (ноти або акорди).

Кожному елементу встановлюємо тональність, що дістаємо з *condition*.

Кожному елементу встановлюємо висоту за замовчуванням рівну тоніці тональності на першій октаві.

Додаємо ноту до партії.

Коротко поясню як працює *return Party().apply { ... }*.

Функція *apply*, яка слідує одразу за викликом конструктора класу, це нововведення мови Kotlin, яке дозволяє використовувати анонімні-об'єкти одразу на момент створення. Це означає, що ми можемо не зберігати нікуди значення об'єкта, а провести всі необхідні маніпуляції з ним одразу після створення, звертаючись до його методів так, ніби ми знаходимося в ньому.

3.3.4. Метр

```
class MeterNode: Node<MeterCondition>() {

  override fun applyCondition(source: Party): Party {
    return Party().apply {
      val condition = requireCondition().meter
      for (index in 0 until source.items.size) {
        val item = source.items[index].copy()
        item.setMeter(condition)
        add(item)
      }
    }
  }
}

class MeterCondition(
  val meter: Meter
)
```

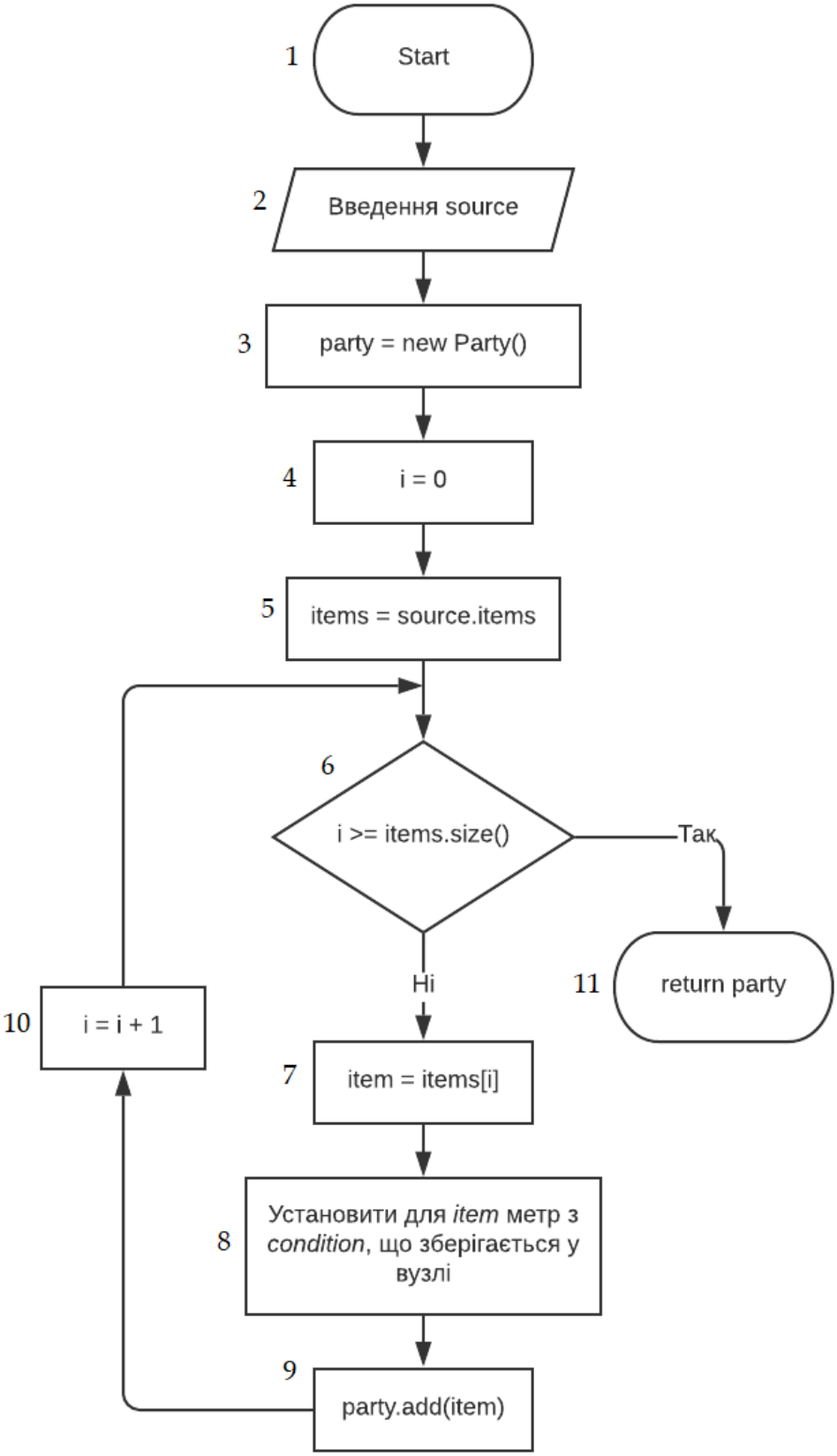

Рис. 3.8. Блок-схема накладання метричного шаблону

1. Дістаємо метр з умови.

2. Послідовно обходимо всі елементи партії-параметра.

3. Встановлюємо їм метр.

4. Додаємо елемент до нової партії.

Важливо, що у всіх вузлах при обході елементів партії *source* ми створюємо копії її елементів, а не видозмінюємо існуючі.

3.3.5. Інтервальна закономірність

Для інтервального та гармонійного вузла застосовується однакова обгортка даних – *IntervalCondition.*

```
class IntervalCondition(
  val sequence: List<HeightShift>
)
```

Це простий клас зі списком «зсувів за висотою» − *HeightShift*:

```
class HeightShift(
  val steps: Int,
  val octave: Int
)
```

Нагадаємо, що висота ноти в класі *Note* зберігається в класі-обгортці – *Height*.

```
class Height(
  var octave: Octave = FIRST_OCTAVE,
  var letter: Letter = C
)
```

Height – це просто обгортка для двох інших полів – обраної октави та букви, що позначає ноту.

Таким чином, *Height* задає абсолютне положення ноти за висотою (октава та ім’я ноти на якій вона знаходиться). А *HeightShift* – нічого не знає про конкретну висоту, натомість в двох його полях записується зміщення, яке має перемістити ноту з її поточної висоти на необхідну кількість октав (*octave*) та ступенів ладу (*steps*). Обидва значення можуть бути додатними чи від’ємними, переміщуючи ноту вгору або вниз.

Розглянемо реалізацію метода applyCondition в IntervalNode (рис. 3.9).

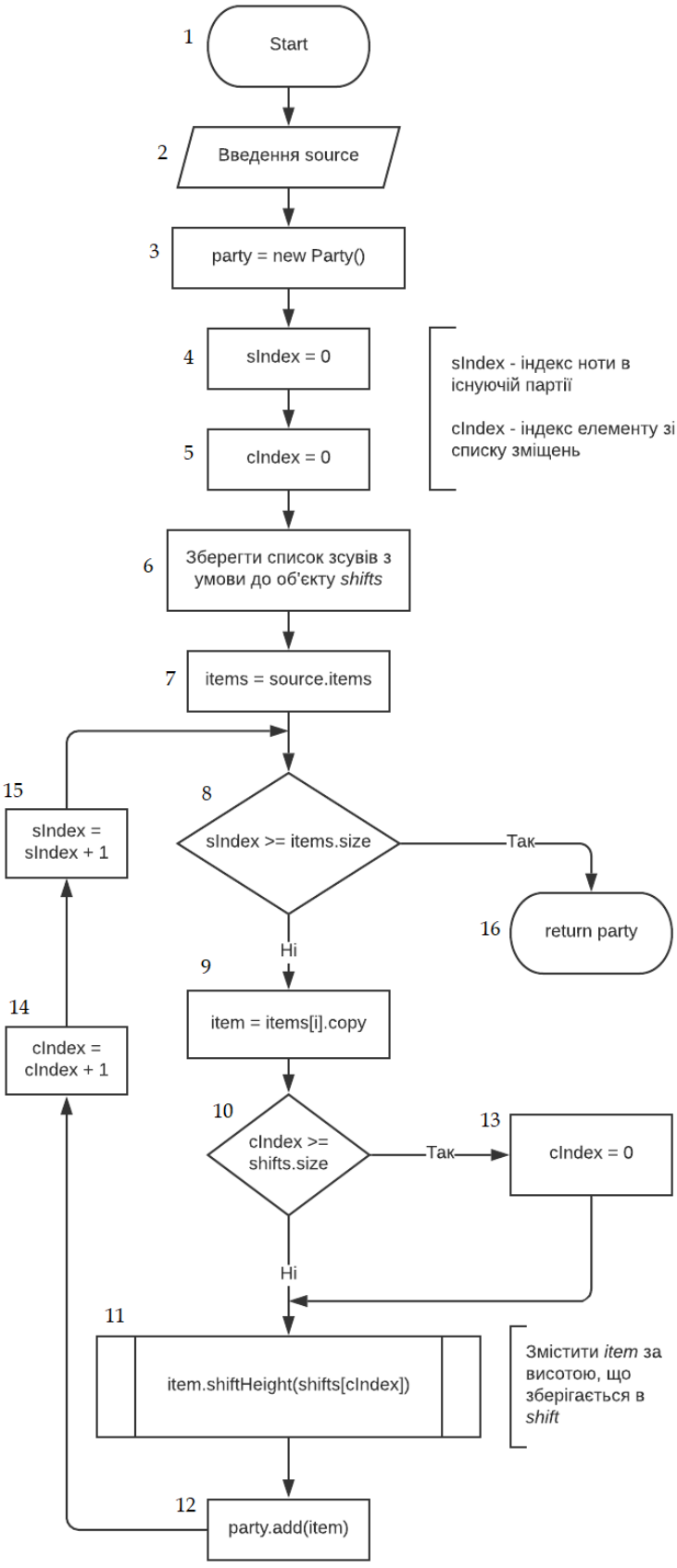

Рис. 3.9. Блок-схема накладання інтервального шаблону

```
class IntervalNode: Node<IntervalCondition>() {
```

```
  override fun applyCondition(source: Party): Party {
    return Party().apply {
      var sIndex = 0 // index of source item
      var cIndex = 0 // index of condition item
      val shifts = requireCondition().sequence

      while (sIndex < source.items.size) {
        val item = source.items[sIndex].copy()
        if (cIndex >= shifts.size) {
          // condition sequence end reached
          // reset condition
          cIndex = 0
        }
        item.shiftHeight(shifts[cIndex])
        add(item)
        sIndex++
      }
    }
  }

}
```

Як має працювати алгоритм?

Маємо вхідні дані – партію, яка може містити ноти або акорди. І маємо умову зі списком зміщень (*IntervalCondition* зі списком *HeightShift*).

Потрібно послідовно обійти всі елементи вхідної партії *source* та змістити кожен елемент (ноту або акорд) на відповідну висоту, записану в списку умов (*sequence* в *IntervalCondition*). Ми зміщуємося по списку *sequence* і по списку елементів з партії *source*.

Процес триває доки не буде досягнуто кінця елементів *source*. Якщо при цьому ми доходимо до кінця *sequence*, то переміщаємося до його початку і починаємо використовувати з першої умови.

Важливо зрозуміти що означає «зміщення за висотою» та як воно реалізовано в програмі.

Висотне зміщення ноти

Зміщення ноти за висотою відбувається по ступеням ладу, на якому базується тональність. Фактично найбільш примітивний алгоритм такого зсуву просто переміщає ноту вгору або вниз (залежно від того яке зміщення – додатне чи від'ємне) рухаючись лише по нотам, дозволеним в цій тональності та оминаючи інші.

Скажімо, якщо маємо тональність Ля-мінор (Am), то нотами, які входять до цієї тональності є ноти A, B, C, D, E, F, G (Ля, Сі, До, Ре, Мі, Фа, Соль), тобто – всі білі клавіші фортепіано.

Тоді, якщо ми маємо ноту, висотою D2 (Нота «Ре» другої октави) та висотне зміщення *octave = -1, interval = 3*, то ми маємо спуститися на октаву нижче і підняти отриману ноту на 3 ступені вгору … E, F, G. Таким чином отримуємо ноту G1 (нота «Соль» першої октави).

В програмі висотне зміщення реалізоване через використання методу *shiftHeight* (рис. 3.10), що належить ще одному допоміжному класу *MusitUtils* (клас, до якого ми винесли всі алгоритми пов'язані з музичною теорією):

```
object MusicUtils {

  //...

  fun shiftHeight(
    height: Height,
    tonality: Tonality,
    shift: HeightShift
  ) {
    val letters = tonality.letters
    var index = letters.indexOf(height.letter)
    var octave = height.octave.ordinal
    val size = letters.size

    octave += shift.steps / size + shift.octave
    index += shift.steps % size

    // check is letter in bounds
    if (index >= size) {
      octave++
      index -= size
    } else if (index < 0) {
      octave--
      index += size
    }

    // check is octave in bounds
    if (octave >= Octave.values().size) {
      octave = Octave.values().size - 1
    } else if (octave < 0) {
      octave = 0
    }

    height.octave = Octave.values()[octave]
    height.letter = letters[index]
  }

  //...

}
```

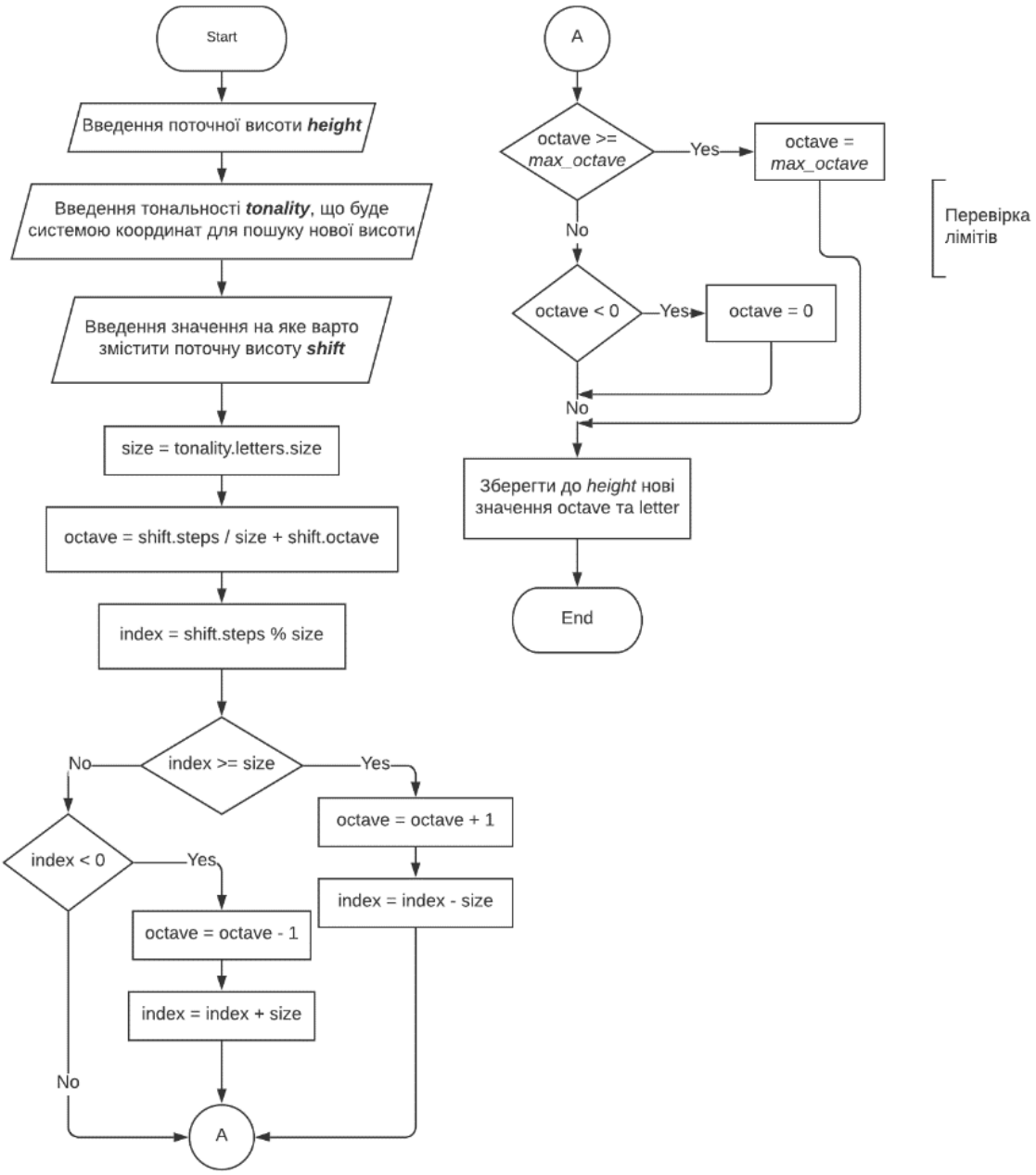

Рис. 3.10. Блок-схема алгоритму висотного зміщення

На вхід функції отримуємо три змінні:

height – об’єкт висоти, яку будемо зміщувати

tonality – тональність, по ступеням якої буде відбуватися зміщення

shift – об’єкт з даними по зсуву, який необхідно здійснити

Дістаємо список нот, які входять до тональності та зберігаємо їх до змінної *letters.*

Дізнаємося порядковий номер буквеного позначення ноти серед нот дозволених в тональності (нагадаємо, що список нот збирається від тональності, рухаючись вгору). Зберігаємо в змінну *index*.

Зберігаємо порядковий номер октави *octave* та кількість нот, які входять до тональності – *size*.

Далі знаходимо нове значення індексу октави – для цього до *octave* приплюсовуємо кількість октав з умови, а також ділимо зсув по ступеням на кількість ступенів в октаві, щоб дізнатися чи є додатковий зсув по октавам і приплюсовуємо це число туди ж.

Знаходимо новий індекс ноти додаючи остаток від ділення *shift.steps* на *size*, щоб зсув за ступенями відбувався в межах однієї октави.

Далі перевіряємо чи знаходиться новий індекс ноти в доступних межах. Якщо він перевищує її, збільшуємо індекс октави на «1», а фактичне значення *index* навпаки зменшуємо на *size* (кількість нот в одній октаві).

Якщо індекс ноти менше допустимого значення, отже ми знаходимося насправді на октаву нижче від потрібної. Тому зменшуємо *octave* на «1», а індекс збільшуємо на кількість нот в октаві.

Наступним кроком перевіряємо октаву за схожим принципом. Якщо октава перевищує індекс доступних значень, встановлюємо максимально можливе. Якщо октава нижче мінімальної – встановлюємо мінімальну.

Зберігаємо оновлені дані до полів об’єкта *height*.

3.3.6. Гармонія

Гармонією в музиці називається послідовність акордів, що акомпанують мелодії.

Гармонійний вузол також взаємодіє з висотним зміщенням, але на відміну від інтервального, не змінює висоти самої ноти, а будує на основі цієї ноти, акорд, копіюючи цю ноту та зміщуючи копії на висоти, зашифровані в послідовності *IntervalCondition* (рис. 3.11).

```
class HarmonyNode: Node<IntervalCondition>() {

  override fun applyCondition(source: Party): Party {
    return Party().apply {
      val shifts = requireCondition().sequence
      for (item in source.items) {
        val baseNote = item.totalNotes()[0].copy()
        baseNote.setHeight(item.getHeight())
        val chordNotes: MutableList<Note> = mutableListOf()
        for (shift in shifts) {
          val copy = baseNote.copy() as Note
          copy.shiftHeight(shift)
          chordNotes.add(copy)
        }
        add(Chord(
          item.getTonality(),
          item.getMeter(),
          item.getHeight(),
          chordNotes
        ))
      }
    }
  }
}
```

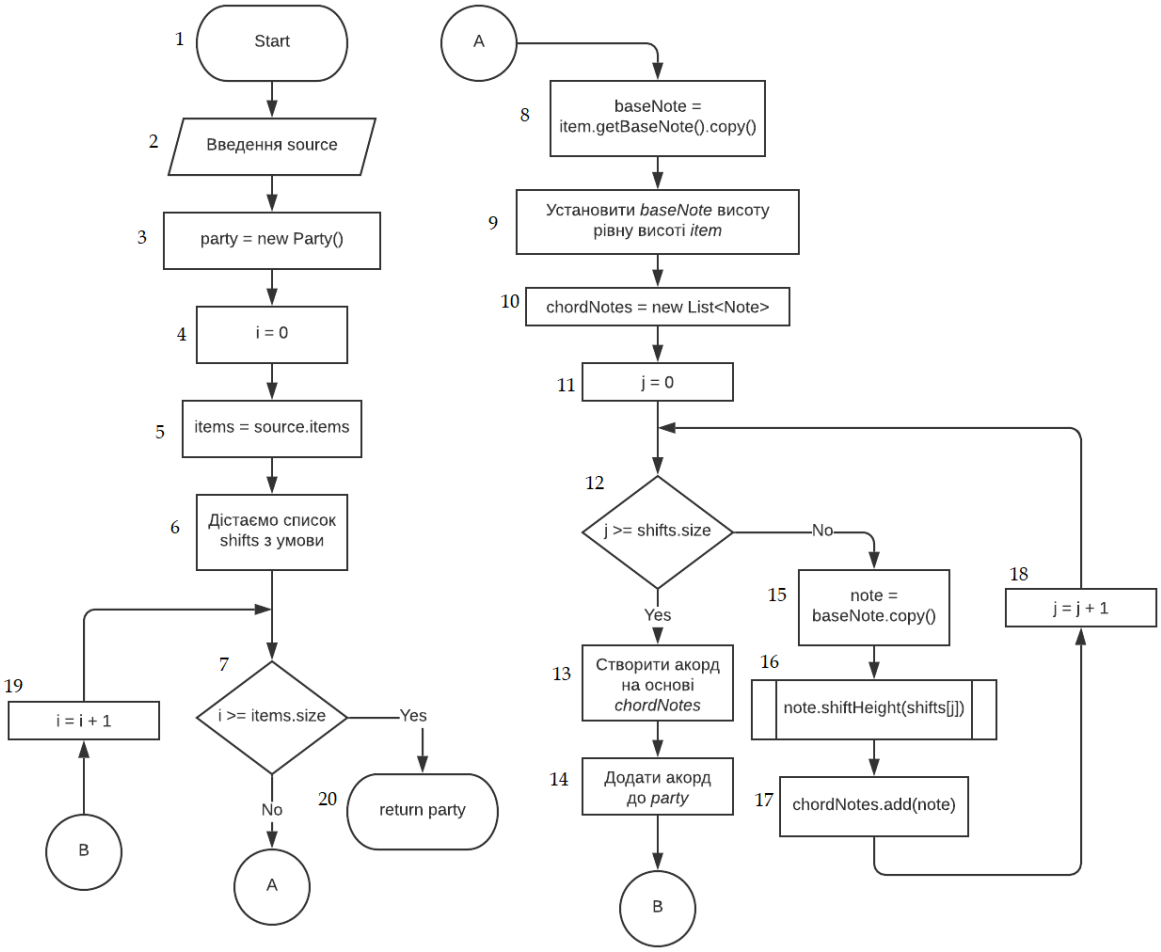

Рис. 3.11. Блок-схема накладання гармонійного шаблону

Дістаємо список зміщень з умови.

Обходимо елементи *source*. Щоб отримати всі характеристики ноти, дістаємо першу ноту з *totalNotes* (функція повертає список нот акорду або одну ноту, якщо це не акорд). Зберігаємо цю ноту в константу *baseNote*.

Встановлюємо базовій ноті висоту елемента (оскільки вона може відрізнятися від тієї що встановлена в першій ноті акорду, якщо елемент виявиться акордом).

Створюємо пустий список для нот акорду *chordNotes*. Далі цей список будемо заповнювати.

В циклі обходимо список зміщень. На кожній ітерації створюємо копію базової ноти, зміщуємо її на висоту *shift*, користуючись знайомим методом з *MusicUtils*, та додаємо до списку нот акорду.

В кінці створюємо новий акорд та додаємо його до партії.

Повторюємо той самий цикл для інших елементів *source*.

3.3.7. Ритм

Розглянемо застосування ритмічного шаблону на візуальному прикладі. Скористаємося piano-roll FL Studio для відображення нот. На висоту нот зважати не будемо. Нас цікавить лише ритміка.

Маємо два ритмічних рисунки:

Рис. 3.12. Ритмічний малюнок нот в партії

Послідовність «1» представляє собою ритмічний малюнок нот в партії. Послідовність «2» − ритмічний малюнок шаблону, який будемо накладати на партію (рис. 3.13).

Фільтруючий ритм працює за принципом кон'юнкції – сигнал на виході буде присутнім, лише якщо він присутній на обох входах. Тобто матимемо згенеровану ноту, тільки в ті періоди, коли одночасно присутні ноти обох ритмічних малюнків:

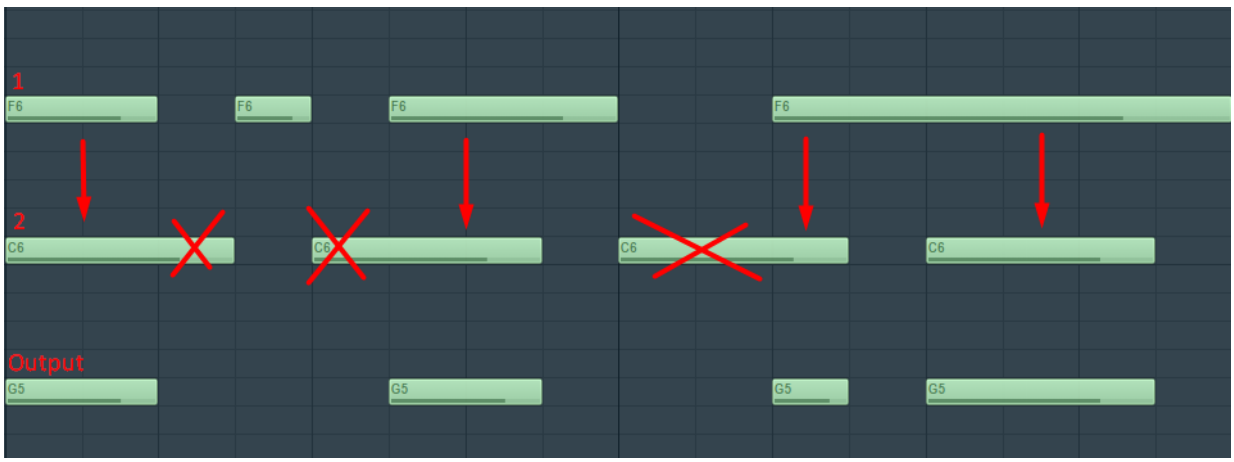

Рис. 3.13. Ритмічний малюнок шаблону, який накладається на партію

Ноти 1-ої послідовності накладаються на ноти 2-ої, там де вони звучать одночасно. Залишки нот, які не пересікаються – просто втрачаються (ми їх закреслили на малюнку).

В результаті отримуємо партію «Output».

```
class RhythmNode: Node<RhythmCondition>() {

  override fun applyCondition(source: Party): Party {
    return Party().apply {

      var n0 = 0 // index of current note in party
      var n1 = 0 // index of current note in condition
            // sequence
      var i1: TimeBounds // item from party
      var i2: TimeBounds // item from condition
      val condition = requireCondition()
      var shift = 0

      // until party finish reached
      while (n0 < source.size()) {

        if (n1 >= condition.sequence.size) {
          // condition sequence end reached
          // reset condition, shift and continue
          n1 = 0
          shift += condition.size
          continue
        }

        val item = source.items[n0].copy()
        i1 = item.getTimeBounds()
        i2 = condition.sequence[n1]

        val i1Start = i1.start
        val i1End = i1.end
        val i2Start = shift + i2.start
        val i2End = shift + i2.end

        // check and update indexes if condition
        // item and party item not intersects
        if (i2End < i1Start) {
          n1++
          continue
```

```
        } else if (i1End < i2Start) {
          n0++
          continue
        }

        val start = max(i1Start, i2Start)
        val end = min(i1End, i2End)
        item.setTimeBounds(TimeBounds(start, end))
        add(item)

        // update indexes
        if (i1End < i2End) {
          n0++
        } else if (i2End < i1End) {
          n1++
        } else {
          n0++
          n1++
        }
      }
    }
  }
}
```

RhythmCondition.kt

```
class RhythmCondition(
  val sequence: List<TimeBounds>,
  val size: Int
)
```

При застосуванні ритмічного шаблону, ми виходимо з того, що на вхід вузла потрапляє партія де всі ноти відсортовані за часом початку звучання, тому ми можемо обходити їх послідовно, знаючи що час зростає разом з індексом поточної ноти. Алгоритм працює наступним чином (рис. 3.14):

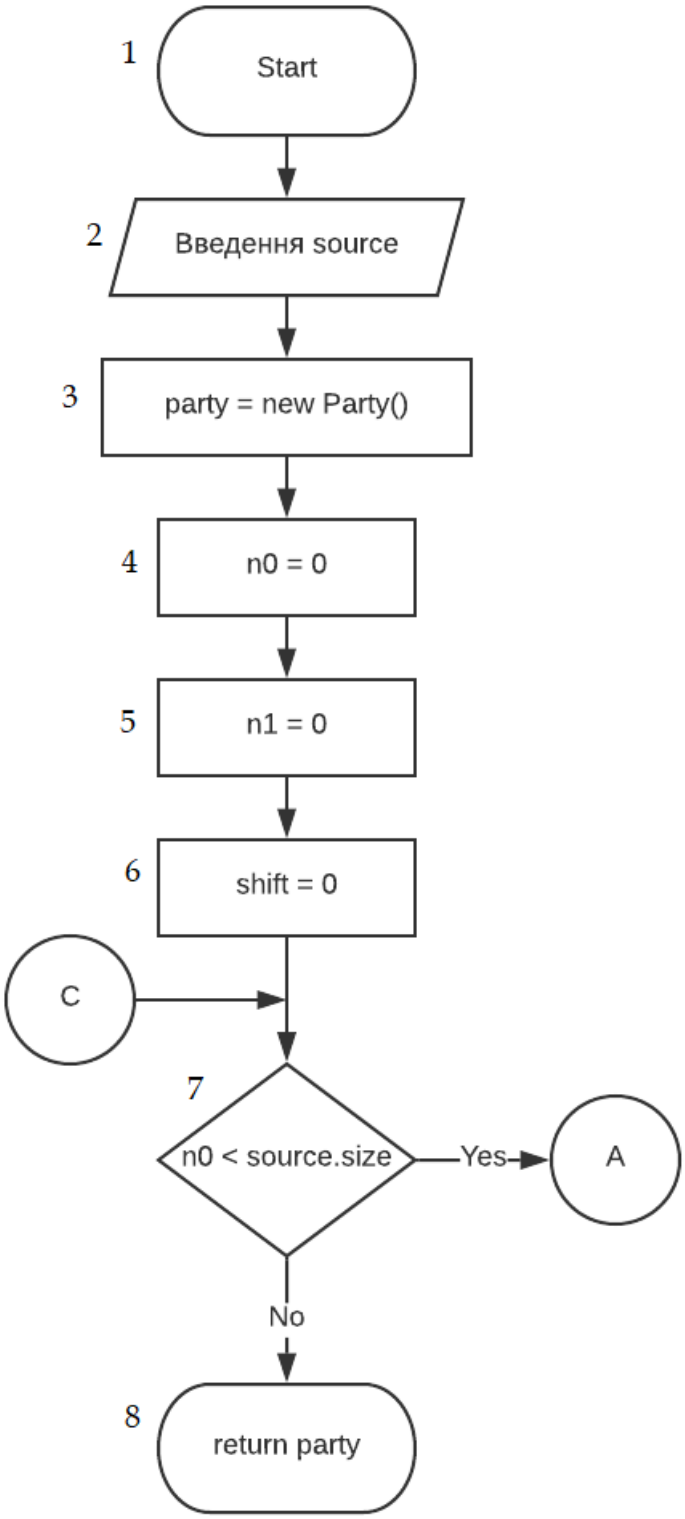

Рис. 3.14. Блок-схема накладання ритмічного шаблону, частина 1

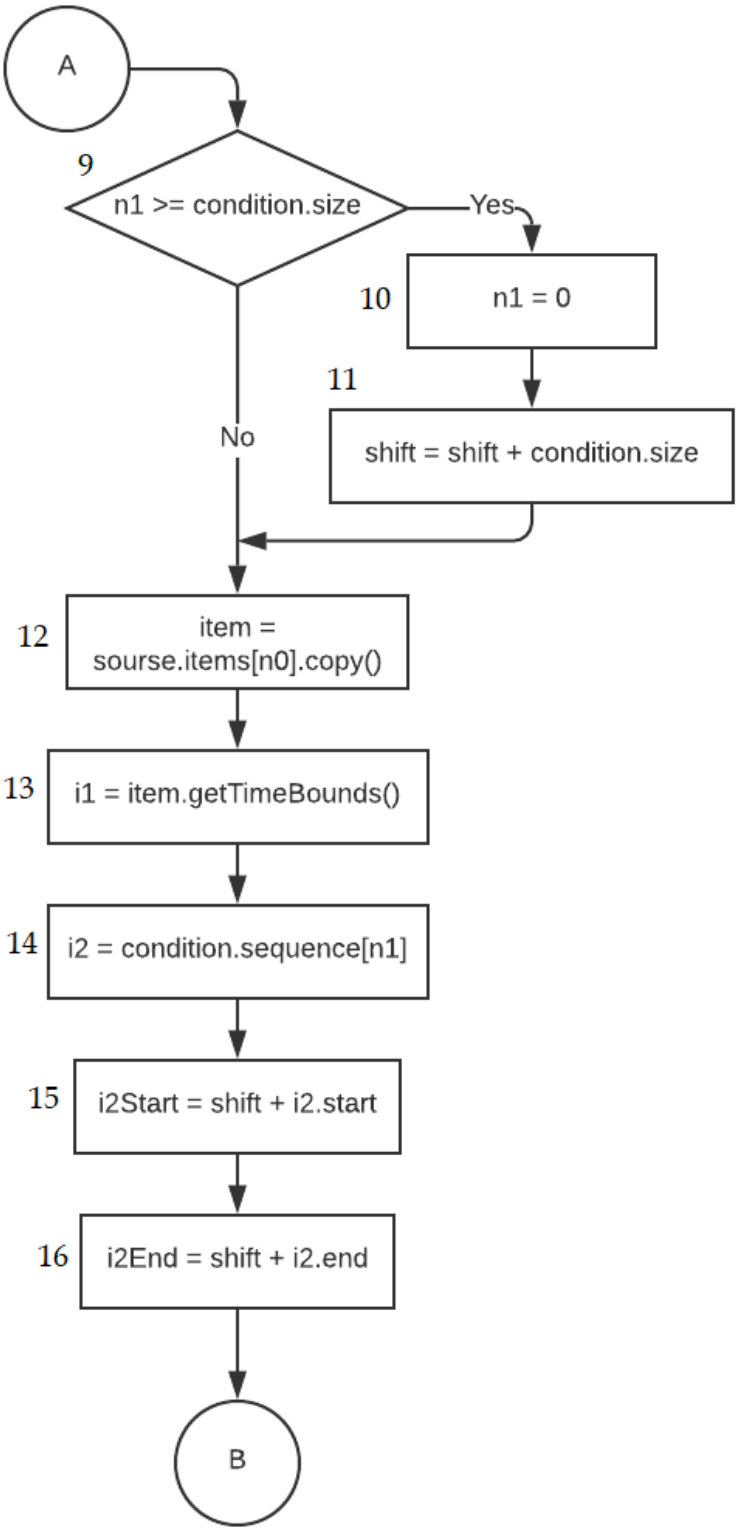

Рис. 3.14. Блок-схема накладання ритмічного шаблону, частина 2

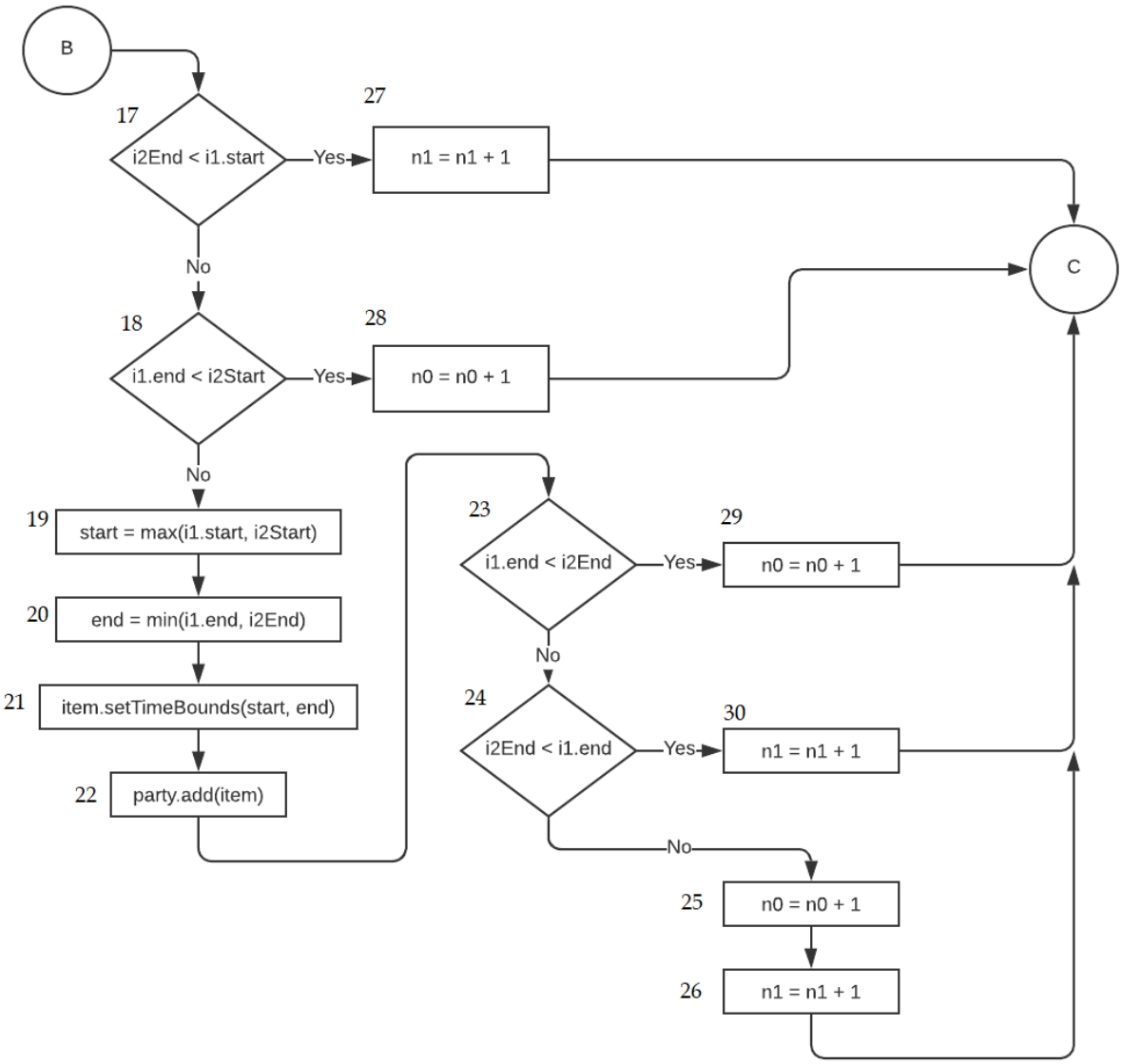

Рис. 3.14. Блок-схема накладання ритмічного шаблону, частина 3

Нам відома тривалість ритмічного шаблону умови (тривалість зберігається в константі *size*, об’єкту *RhythmCondition*). Тому ми ітеративно проходимо кожну таку тривалість. Створюємо змінну *shift*, яка вказує стартову позицію часу від якої буде відраховуватися кожен елемент зі списку ритмічних умов.

Рухаємося в циклі по елементам *source*, доки не досягнемо кінця партії. В циклі дістаємо поточну ноту та поточну умову. Далі потрібно перевірити. Якщо нота партії та умова не пересікаються, то потрібно перейти до наступної ноти партії або до ноти умови. Зробити цю перевірку досить

просто. Дійсно, якщо позиція закінчення елемента умови менше за позицію початку ноти з партії, це означає, що нота умови знаходиться раніше ноти партії і при цьому вони не пересікаються (рис. 3.15).

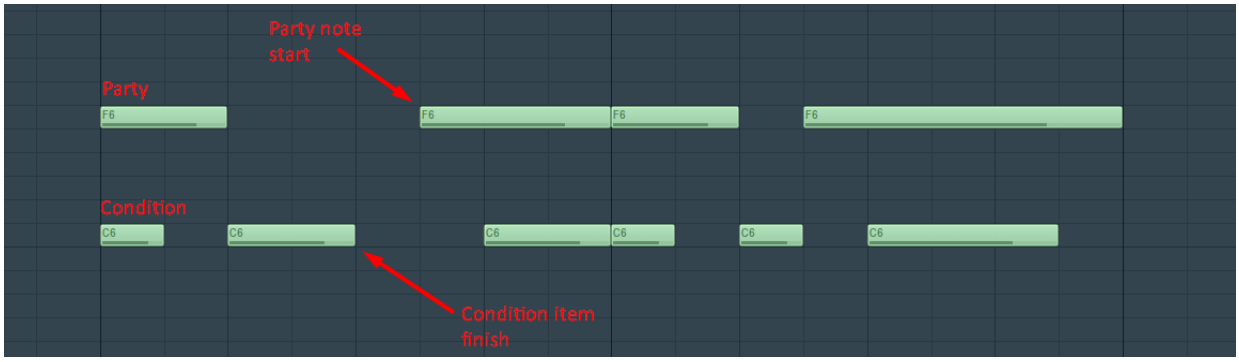

Рис. 3.15. Перевірка взаємодії ноти партії та умови

Тому нам потрібно перейти до наступної ноти з умови та продовжити цикл. Так само перевіряємо чи не закінчується нота партії раніше ніж розпочнеться нота умови (рис. 3.16).

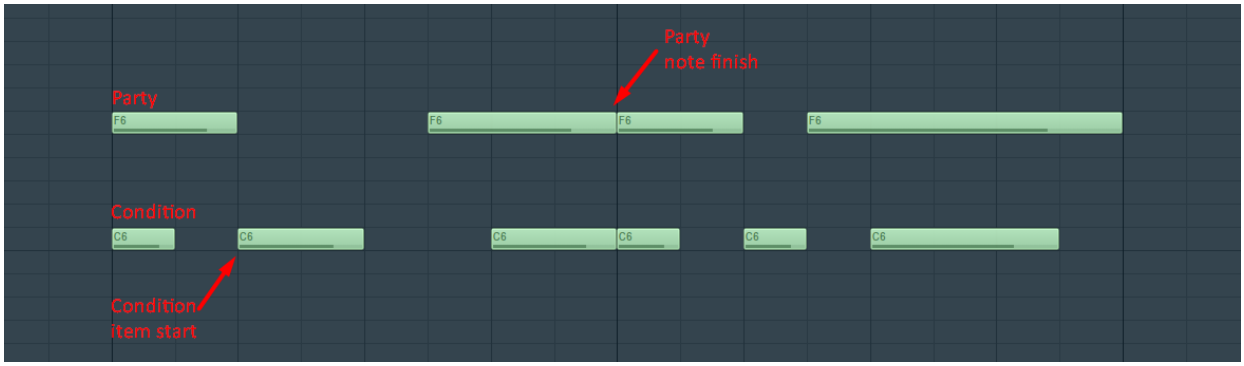

Рис. 3.16. Продовження перевірки взаємодії ноти партії та умови

Якщо це справді так, і вони не пересікаються, ми маємо збільшити індекс поточної ноти партії і продовжити цикл.

Якщо ж ноти пересікаються, стартовою позицією нової ноти буде більше зі значень стартової позиції цих двох нот, а позиція закінчення буде меншим з цих двох чисел. Це й будуть нові часові характеристики ноти.

Після того як часові характеристики нової ноти знайдено, потрібно перейти на наступну ноту в тій послідовності нота якої «закрила» фінальну позицію новоствореної ноти. Щоб установити який індекс потрібно збільшити, варто знову порівняти фінальні позиції і збільшити індекс тієї послідовності, нота якої закінчується раніше, або збільшити індекси обох партій, якщо позиції закінчення їх поточних нот ідентичні.

На старті кожної ітерації ми також перевіряємо чи не виходить поточний індекс ноти з умов, за їх максимальну кількість. Якщо це справді так, потрібно зсунути *shift* на тривалість умови та обнулити індекс умови *n1*. Таким чином ми розбиваємо всю партію на рівні відрізки тривалістю рівною тривалості ритмічного шаблону і працюємо з кожною з них окремо. Це є ще одним впровадженням парадигми «розділяй та володарюй» в нашій програмі.

Підсумовуючи алгоритм можна описати його так:

Розбиваємо всю партію на рівні відрізки, тривалістю рівною тривалості ритмічного малюнку з умови.

Перевіряємо чи пересікаються поточні ноти умови та партії. Якщо ні, переміщуємося вперед, по нотам умови або нотам партії, дивлячись, яка з нот відстає.

Якщо ноти пересікаються, створюємо на основі поточної ноти партії, нову ноту з часовими характеристиками від моменту, коли обидві ноти почали звучати, і до моменту коли, хоча би одна з них закінчилася.

В залежності від того яка нота закінчилася раніше, зміщуємося на наступну ноту в партії або в ритмічному шаблоні, або одразу в обох, якщо вони закінчилися одночасно.

По закінченню нот в ритмічному шаблоні, він розпочинається знову.

Знову ж, всі інші характеристики ноти (висота, тональність, гармонія та інше) дістаються з існуючої ноти партії. А змінюємо ми лише часові характеристики цих нот.

Висновки до розділу 3

1. Для розробки алгоритмів, в проекті застосовується парадигма «розділяй та володарюй», за якої складна задача розбивається на менші, ті в свою чергу на ще менші і так доки не будуть отримані елементарні задачі. Далі задача вирішується рекурсивно – починаючи з елементарних задач і підіймаючись вгору, доки не буде розв’язана повністю. Ця парадигма широко застосовується для проектування складних алгоритмів, оскільки емпірично доведено що такий підхід потребує меншого числа конкретних операцій.

2. Користуючись парадигмою «розділяй та володарюй», а також принципами чистої архітектури, алгоритми були відділені до окремих блоків і розміщені за призначенням. Таким чином кожен окремий алгоритм впровадження умов виступає незалежною одиницею, яка нічого не знає про інші алгоритми. Це дозволяє змінювати та тестувати кожен алгоритм незалежно від іншого коду.

3. Загальний алгоритм генерації відповідає за послідовну обробку треків композиції, всіх сегментів треку та всіх вузлів сегменту. При цьому треки генеруються в послідовності наслідування. Тобто спершу батьківські, потім їх дочірні і так далі. Головна мета – забезпечити накладання умов, обраних користувачем, у правильному порядку.

4. В процесі генерації ми не змінюємо існуючої партії, а в кожному вузлі створюємо нову та заповнюємо її відредагованими параметрами. Такий підхід заснований на одній з парадигм функціонального програмування – *незмінюваності даних*. Багато проблем в роботі програм виникає по причині модифікації даних з різних місць, особливо при застосуванні багато-поточного програмування. Тому за можливості дані варто робити незмінюваними.

РОЗДІЛ 4. ОГЛЯД РЕЗУЛЬТАТІВ ТА ПОДАЛЬШИЙ РОЗВИТОК ПРОГРАМИ

4.1. Приклад застосування програми

Для ілюстрації застосування алгоритмів розглянемо хід генерації на реальному прикладі. Створимо відрізок композиції на 4 такти, який складається з 4-х треків:

1. Базовий трек
2. Бас
3. Акорди
4, Мелодія

Візуальні приклади результатів будемо отримувати з програми FL Studio, так як візуальний інтерфейс нашої програми, попередній перегляд midi поки що не підтримує.

4.1.1. Генерація базового треку

Тривалість базового треку встановимо рівною 4-м тактам. Він буде складатися з єдиного сегменту, тривалістю також 4 такти. Умови сегменту:

- Тональність – Am
- Розмір – 4/4
- Ритмічний малюнок – тривалість в 1 такт
- Інтервальна послідовність – «0|-2|-4|+1»

Нагадаємо, що для *незалежного* треку (який не має *parent-треку*) перед накладанням умов генерується «*пуста*» нота з часовими рамками рівними тривалості сегменту.

На наступних ітераціях на цю ноту накладається тональна умова, а висота ноти встановлюється рівною тоніці тональності в першій октаві. В нашому випадку – нота «Ля» першої октави. Накладання *розміру* на цьому етапі візуально не представлено (рис. 4.1).

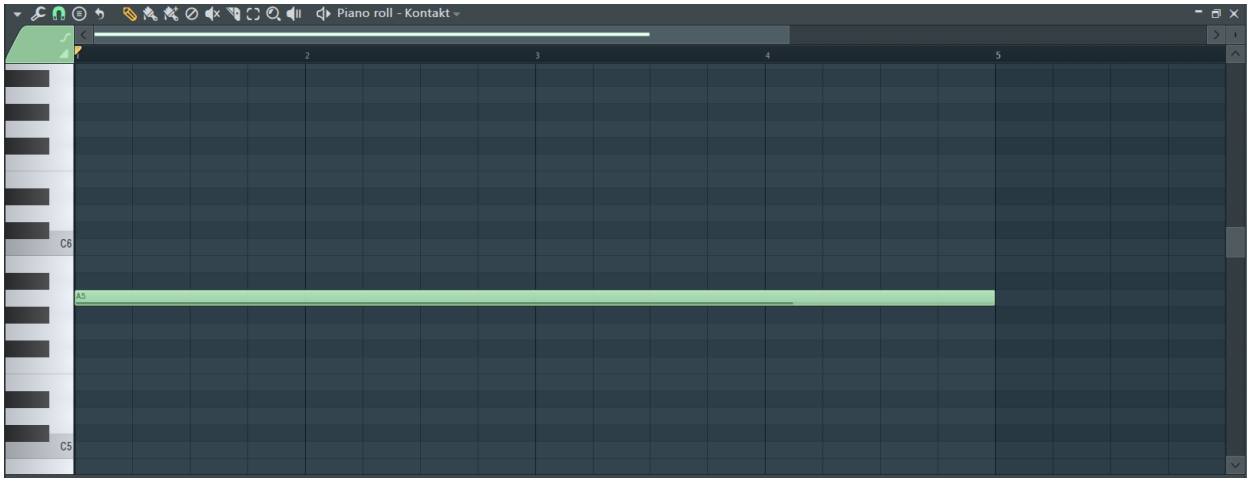

Рис. 4.1. Результат накладання тональності та розміру

При накладанні ритмічного малюнку суцільна нота, створена раніше, розділяється на 4-ри ноти, кожна тривалістю – 1 такт (рис. 4.2):

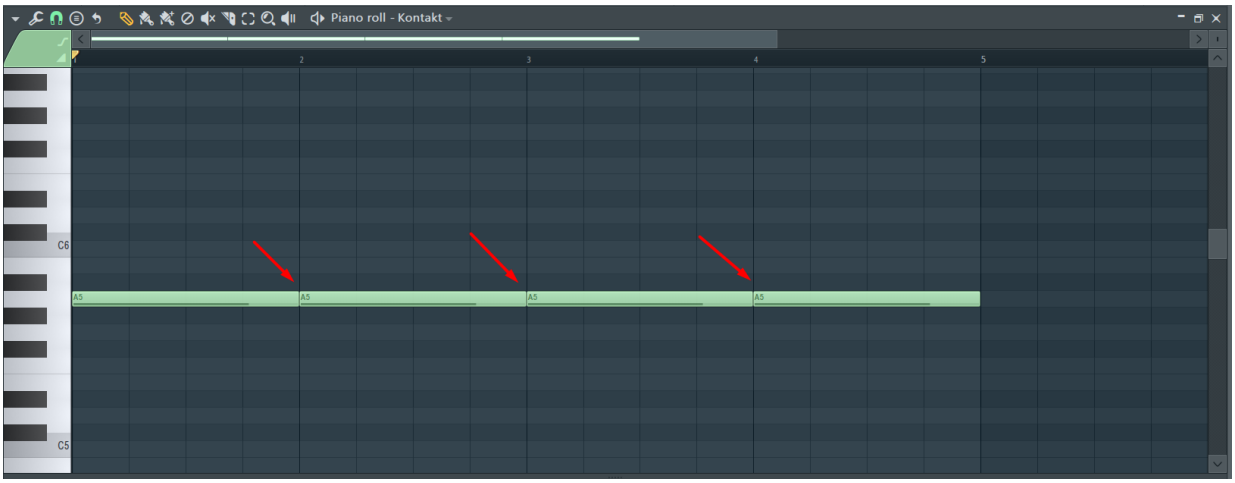

Рис. 4.2. Результат накладання ритмічної умови (стрілками показано межі нот)

На етапі накладання інтервальної умови перша нота залишається на тоніці (зміщення «0»), друга зсувається на дві ступені вниз (зміщення «-2»). Це означає, перейти на дві доступні в тональності ноти вниз. В тональності Ля-мінор, доступні всі ноти, розміщені на білих клавішах фортепіано (і не доступні ті, що розміщені на чорних клавішах).

Таким чином друга нота зміщується на дві ступені вниз (оминаючи чорні клавіші, які представляють ноти, що не входять до тональності, рис. 4.3):

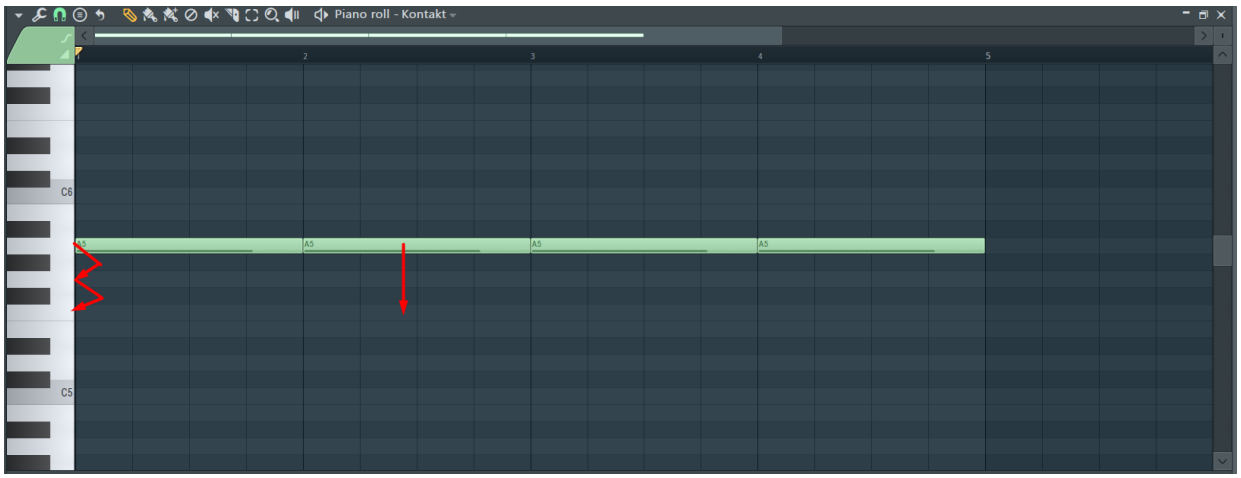

Рис. 4.3. Візуальне представлення процесу накладання інтервальної умови

Третя нота опускається на 4-ри ступені, а остання – підіймається на 1 ступінь вгору (рис. 4.4):

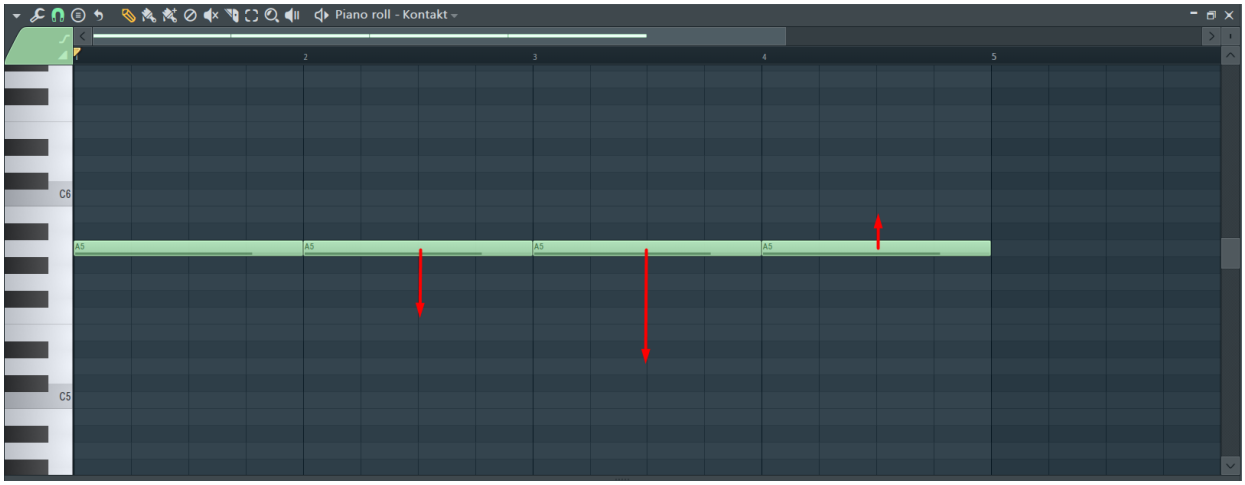

Рис. 4.4. Візуалізація інтервальної закономірності

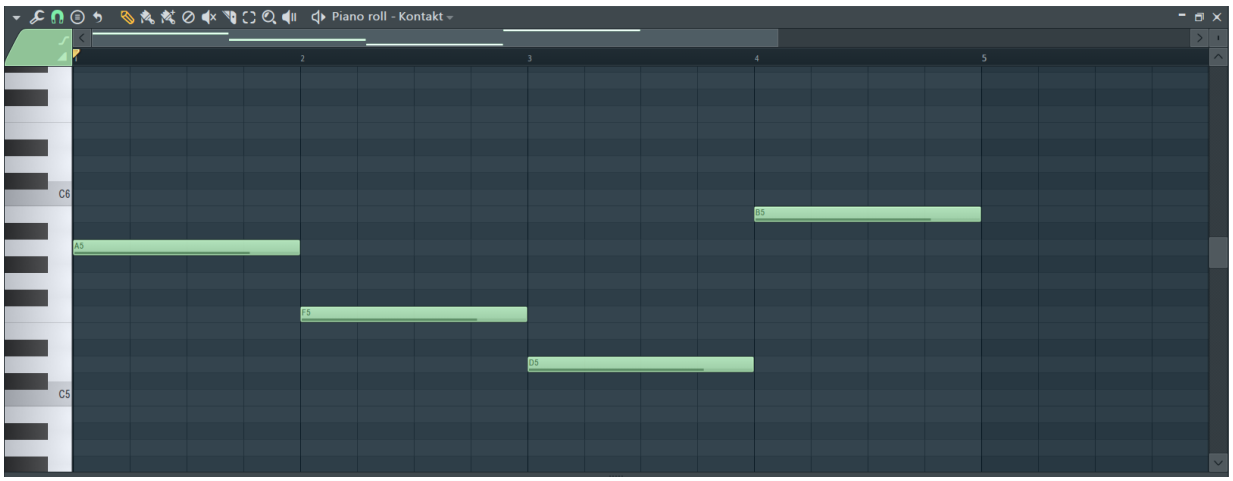

Рис. 4.5. Результат накладання інтервальної закономірності

4.1.2. Генерація треку басу

Нехай басовий трек складається з 2-х сегментів.

Сегмент 1

Тривалість – 3 такти (384 імпульси в PPQN)

Умови:

- Ритмічна – «32|128+0:32|32:64|64:96|112:124»
- Інтервальна – зміщення на октаву вниз («-1:0»).

Сегмент 2

Тривалість – 1 такт (128 імпульсів в PPQN)

Умови:

- Ритмічна – «32|64+0:16|16:32|48:64»
- Інтервальна – зміщення на октаву вниз («-1:0»).

Нагадаємо коротко як розшифровуються записи.

Ритмічний запис «32|128+0:32|32:64|64:96|112:124» означає:

Зліва від знаку «+» розширення: 32 – розмір четвертної ноти в PPQN (скільки імпульсів поміщається в тривалості четвертної ноти), 128 – розмір всього малюнку в PPQN (в даному випадку малюнок має тривалість рівну 4-м четвертним нотам).

Справа від знаку «+» знаходиться сам ритмічний малюнок, де різні ноти розбиті знаком «|», зліва від «:» знаходиться стартова позиція звучання ноти, справа – позиція завершення її звучання. Всі величини записані в PPQN.

Описаний ритмічний малюнок матиме таку форму (рис. 4.6):

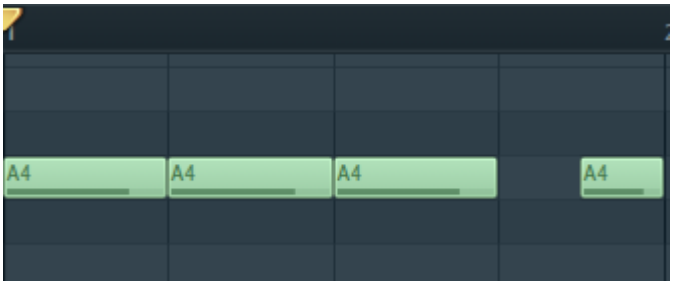

Рис. 4.6. Ритмічний малюнок відображений на конкретних нотах в Piano Roll програми FL Studio

Три четвертні ноти, пауза тривалістю восьмої ноти та одна восьма нота.

Після застосування умов до обох сегментів, маємо на виході наступну партію:

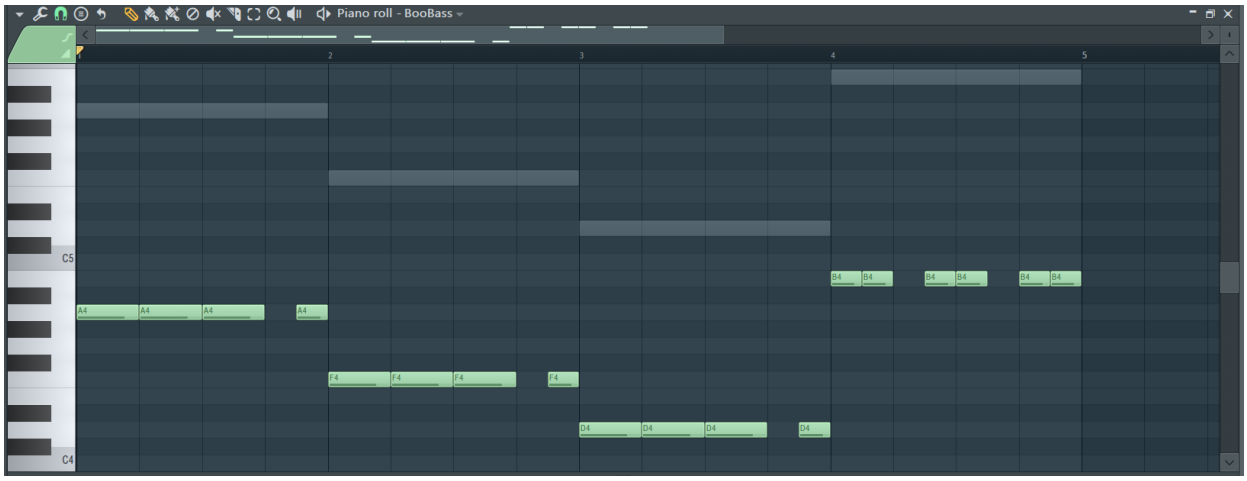

Рис. 4.7. Партія басу сформована на основі базової партії (базова партія позначена сірим кольором)

4.1.3. Генерація треку акордів

Нехай трек акордів складається з 2-х сегментів, кожен тривалістю 1,5 такту. Перший сегмент стартує з початку треку, другий − з середини.

Умови на обох сегментах однакові:

Гармонійна – «-1:0|-3|0|+2»

При генерації алгоритм спирається на ноти базової партії, створюючи в один момент часу 4-ри ноти на основі однієї. Таким чином утворюючи акорди. Перша нота зміщується на октаву вниз, друга на 3 ступені вниз (добираючись до квінти акорду, але розміщеної знизу). Наступною до акорду додається тоніка (зміщення «0»), і нарешті терція зверху (зміщення «+2»).

Простір між сегментами лишається не заповненим (рис. 4.8).

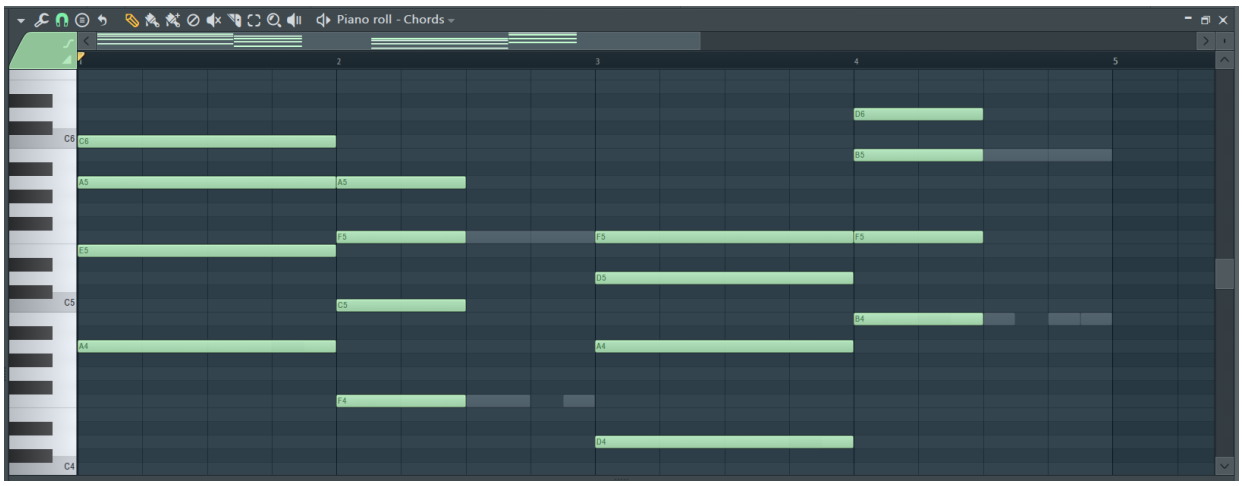

Рис. 4.8. Результат генерації треку акордів (сірим кольором позначені видимі ноти партій базового та басового треків)

4.1.4. Генерація треку мелодії

Мелодію теж не будемо робити складною. Зробимо один сегмент, тривалістю 4-ри такти. З наступними умовами:

- Ритмічна – «32|128+0:16|16:32|32:48|48:96|96:128»
- Інтервальна – «0|+2|+1|+1|-1»

Результат генерації на рис. 4.9:

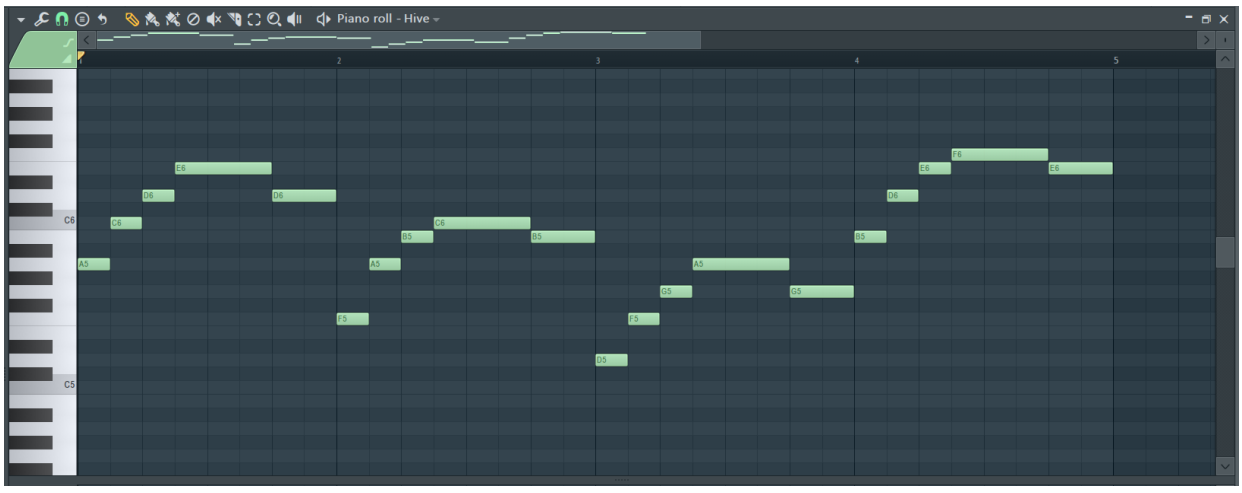

Рис. 4.9. Результат генерації треку мелодії

При оцінці результатів, ми помітили, що останній акорд звучить не гармонійно. Як виявилося ми випадково встановили останньою нотою базової послідовності 2-гу ступінь тональності. Відомо, що друга ступінь

тональності звучить недостатньо гармонійно. Замінимо її на 3-ю ступінь. Для цього просто переходимо до базового треку та в інтервальній умові заміняємо останню цифру послідовності «0|-2|-4|+1» з «+1» до «+2».

Запускаємо повторну генерацію. Всі дочірні партії підтягнули зміни. Повторна генерація зайняла 0,007 секунди (7 Мілі секунд).

В результаті маємо наступні партії (рис. 4.10, 4.11, 4.12, 4.13):

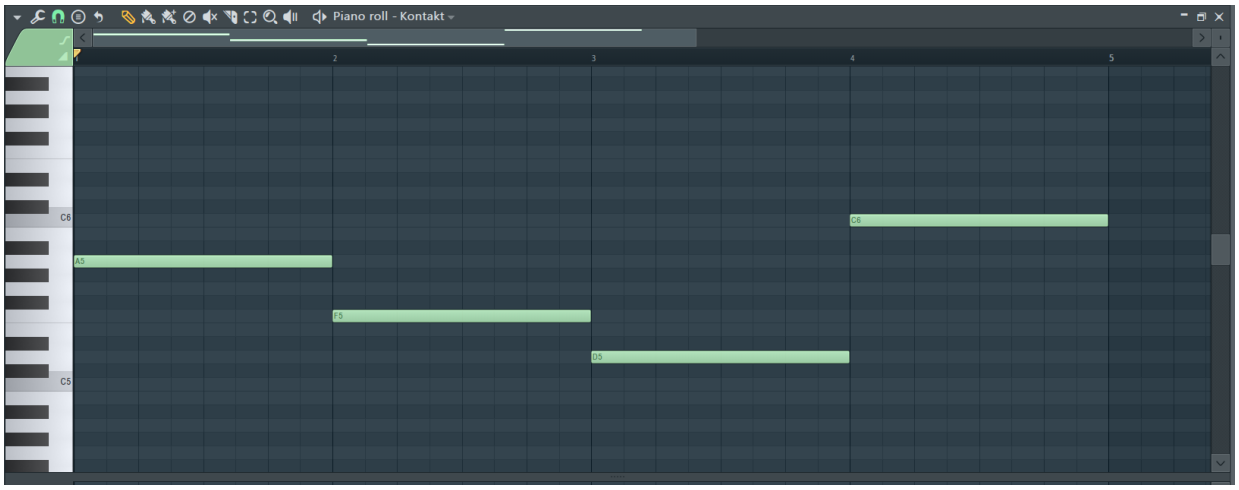

Рис. 4.10. Базова партія

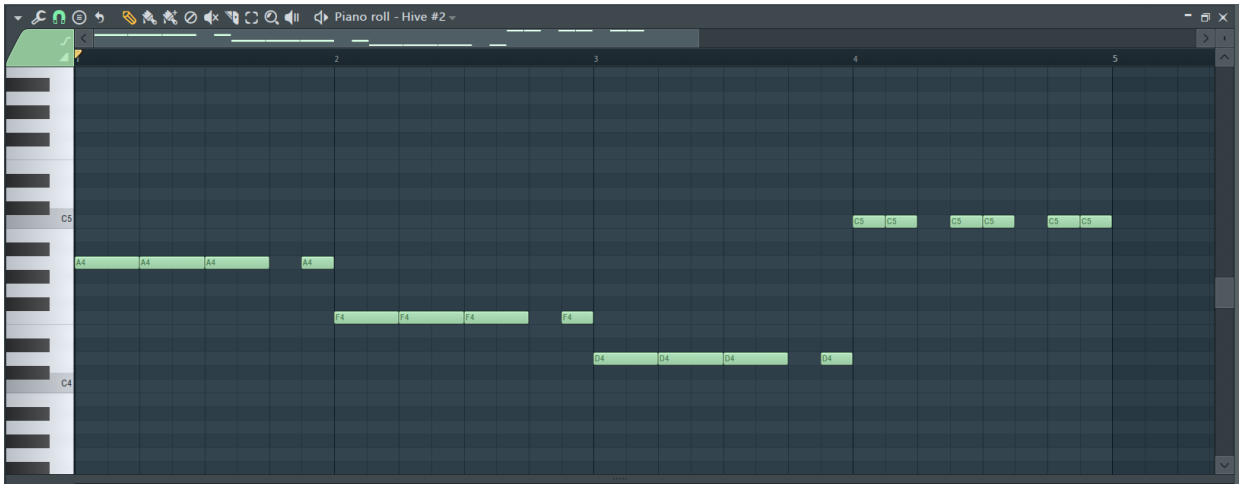

Рис. 4.11. Партія басу

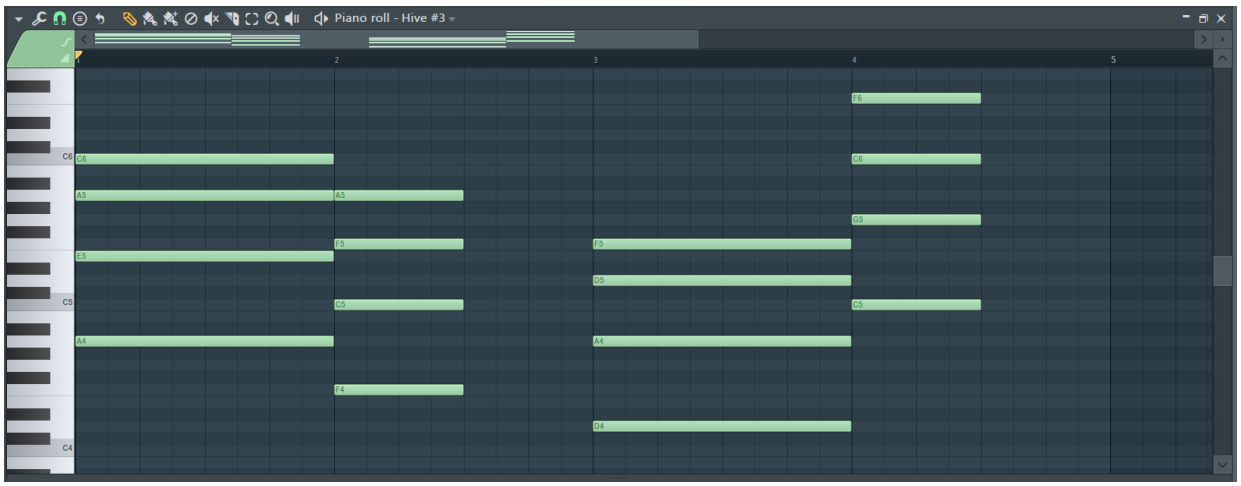

Рис. 4.12. Партія акордів

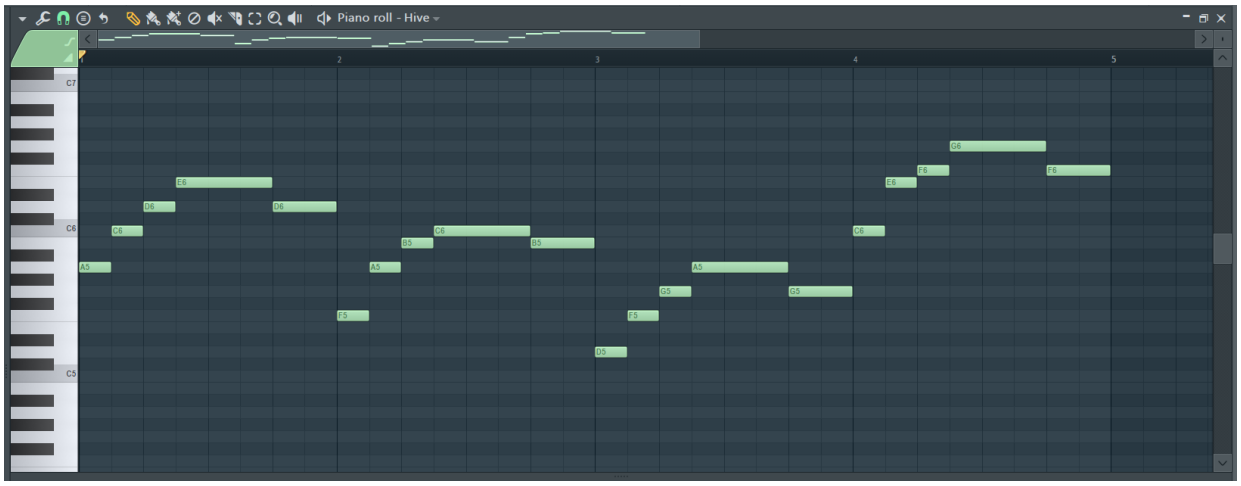

Рис. 4.13. Партія мелодії

4.2. Аналіз приросту продуктивності при внесенні змін

Провести подібну оцінку складно, оскільки різні композиції складаються з різної кількості треків. Також швидкість роботи алгоритмів залежить від характеристик комп’ютера, в першу чергу від характеристик процесора.

До того ж складно об’єктивно оцінити швидкість внесення змін людиною. Різні продюсери застосовують різні інструменти для редагування midi. Те що для одного може зайняти кілька хвилин, для іншого займе одну, або навіть менше. Важливо враховувати також в якому ритмі працює людина.

Зрозуміло що при спробі додати зміни «на швидкість», темп роботи може відрізнятися від звичного.

Однак для демонстрації приросту швидкості при внесенні змін, ми все ж таки вирішили провести невелике дослідження.

Ми взяли отриману в результаті генерації композицію з прикладу і почали змінювати в ній різні умови. При цьому зміни вносили вручну та програмно.

Зміни були наступними – змінити тривалість однієї ноти базової партії, змінити її висоту. Оскільки базова партія складається з єдиного сегменту, то при запуску повторної генерації згенеруються всі треки.

Тестування програми проводили на комп’ютері з процесором Intel Core i7-8750H 2.21 GHz (табл. 4.1).

Таблиця 4.1. Результати тестування програми

№	Ручне редагування, с	Програма, с
1	34	0,003
2	25	0,004
3	31	0,003
4	21	0,003
5	22	0,002
6	40	0,004
7	39	0,002
8	12	0,001
9	31	0,004
10	22	0,001

Власне, середнє тривалість внесення правок при ручному редагуванні становить *27,7 с*.

Тоді як середня тривалість повторної генерації всіх чотирьох треків при внесенні програмних змін, становить приблизно *0,003 с* (*3 мс*).

Тут ми не враховуємо час, необхідний на зміну умови в програмі. Однак цей час за всіх експериментів не перевищував *15 с*. До того ж, важливо розуміти що наведена композиція досить проста. У ній всього 4-ри треки з

нескладними умовами. Якби ми працювали зі складнішими зв’язками, то ручні зміни зайняли б значно більше часу.

Ми повторили експеримент з композицією на 10 треків та більш складними умовами. В результаті, на ручне редагування пішло – *2 хв 3 с*, на редагування умови в програмі – *14 с*, на повторну генерацію – *0,003 с*. Таким чином приріст швидкості у впровадженні змін очевидний.

Втім знову ж, потрібно розуміти що в цьому процесі фігурує багато змінних і в кожному випадку приріст в швидкості порівняно з ручним редагуванням може бути різним, однак зі зростанням складності, продуктивність завжди зростатиме в рази.

4.3. Подальші шляхи розвитку програми

Серед можливих варіантів розвитку програми ми розглядаємо два основних:

1. Перетворення програми до VST та використання в інших DAW.

2. Перетворення програми до повноцінної DAW з впровадженням схожих підходів до обробки аудіо, підбору інструментів та ефектів обробки.

Скоріше за все на даному етапі ми підемо першим шляхом.

Стосовно розвитку та покращення функціоналу ми збираємося:

1. Покращити алгоритми накладання умов, щоб зробити їх ще простіше та зрозуміліше для користувача.

2. Додати можливість аналізу існуючих міді-партій зі збереженням виявлених закономірностей для використання в подальших редагуваннях.

3. Додати особливий різновид шаблонів умов, які не матимуть чіткої реалізації, а описуватимуть певний характер. Конкретна ж реалізація шаблону буде генеруватися ітеративно з можливістю змінити, зберігши характер шаблону.

Висновки до розділу 4

1. Ми розглянули приклад роботи програми та переконалися, що внесення змін з її застосуванням значно виграє порівняно з ручним редагуванням. В експерименті було задіяно 4-ри пов’язаних треки – один батьківський та три дочірні. При внесенні концептуальних змін вручну, процес в середньому займав 27,7 с, тоді як впровадження автоматичних змін займало 0,003 с, не враховуючи часу на ручне редагування умови (яке у всіх експериментах не перевищувало 15 с).

2. Варто розуміти, що отримані результати в швидкості будуть значно відрізнятися для більш складних випадків концептуальних зв’язків. Якщо зв’язків між партіями значно більше і вони складніше і якщо партій буде більше, то програма зможе працювати ще більш ефективно, оскільки всі зв’язки будуть зберігатися і враховуватися при змінах автоматично.

3. Подальшими шляхами розвитку програми ми вбачаємо перетворення до VST інструменту для використання в сторонніх DAW. Іншим напрямком є створення повноцінної DAW та впровадження аналогічних технологій в інших складових роботи над музичною композицією, а саме – обробці звуку, підборі інструментів та інших складових, з метою створення повністю концептуальної системи для створення музики.

ЗАГАЛЬНІ ВИСНОВКИ

1. Для кращого розуміння процесів, які можна покращити впровадженням автоматизації, ми провели дослідження розвитку музичного мистецтва, музичного запису та контексту. Встановили, що за багатовікову історію розвитку музики, виникали правила та закономірності, які з часом були оформлені до теорії музики. Різні автори використовують ці закономірності в процесі своєї роботи, навіть якщо роблять це не свідомо.

2. В сприйнятті музики, як і будь-якого іншого мистецтва, важливішим є контекст, який виникає у вигляді зв’язків між складовими елементами. Так слово набуває сенсу в реченні, а речення – в тексті. Працюючи над композицією, будь-який автор розмірковує більш концептуально. Тобто не лише окремими елементами, а радше зв’язками між ними, що передають той чи інший характер звучання, несуть в собі певну історію. Тому важливо надати автору можливість працювати не лише на рівні конкретної реалізації, а також на рівні контексту.

3. Більшість сучасних програм для створення музики не вміють працювати на концептуальному рівні, а працюють лише на рівні реалізації. Це надає автору широкий рівень творчої свободи, оскільки кожен параметр можна редагувати повністю незалежно від інших. Але також змушує більше перебувати на рівні реалізації і не дозволяє вносити концептуальні значення окремо та швидко їх редагувати. Було прийнято рішення розробити таку систему.

4. Щоб розібратися як вона має працювати та виглядати, було проведено дослідження розвитку музичної теорії, а також генеративного мистецтва, оскільки воно найбільше базується на концептуальних зв’язках. Однак програми-генератори музики також не можна використовувати як надійний інструмент для роботи над композицією на рівні контексту, оскільки вони володіють недостатнім рівнем гнучкості, що означає їх більшу «самостійність» в процесі прийняття концептуальних рішень. Вони

приймають такі рішення керуючись не побажаннями автора, а жорстко прописаними в них алгоритмами.

5. На основі отриманих в процесі дослідження результатів був складений список вимог до майбутньої програми:

- Програма має уміти зберігати та застосовувати концептуальні зв’язки в процесі роботи над музичною композицією.

- Зв’язки мають зберігатися як на рівні окремих партій так і на рівні цілої композиції (тобто встановлюватися не лише між частинами партіями, але і між самими партіями).

- Програма має володіти достатнім рівнем гнучкості. Це означає, що автор зможе працювати як на рівні більш конкретної реалізації так і на рівні контексту, при цьому концептуальні зв’язки між елементами композиції будуть зберігатися в будь-якому випадку.

- Програма має бути ефективною, тобто не виконувати зайвої роботи.

6. На основі висунутих вимог була спроектована програма, яка частково чи повністю реалізує концептуальний підхід до створення музики. Програма базується на застосуванні шаблонів – попередньо заготовлених умов, що впливають на певну концептуальну складову музики – ритміку, динаміку, інтервальну закономірність чи гармонію. Кожен шаблон представляє собою набір зв’язків, які хоча і складаються з конкретних значень, але тільки в певній області.

7. Шаблони зберігаються в базі даних програми і можуть бути використані та модифіковані в будь-який момент часу. Компонування шаблонів як окремих концептуальних одиниць, дозволяє в результаті генерувати конкретні значення автоматично.

8. Для реалізації установки та збереження зв’язків між елементами композиції було додано можливість наслідування партіями. Наслідування в даному випадку означає, що партія як результат генерації може слугувати сировиною для наступних маніпуляцій, в процесі накладання нових умов. Враховуючи що для кожної партії можна встановити різну кількість таких

умов, повний алгоритм генерації можна вважати послідовним ланцюжком накладання умов різного типу на попередньо генеровані ноти.

9. В розробці алгоритмів ми спиралися на парадигму «розділяй та володарюй», яка є домінуючою на даний момент в сфері розробки алгоритмів. За цією парадигмою – будь-який алгоритм досягає вищого рівня ефективності при розподілі задачі на менші, а ті, в свою чергу, на ще менші, доки задачі для вирішення не стануть елементарними. Далі елементарні задачі розв’язують рекурсивно, підіймаючись від найменших складових до вирішення задачі найвищого рівня.

10. Керуючись парадигмою «розділяй та володарюй», а також принципами чистої архітектури SOLID, ми розробили набір алгоритмів для накладання умов різного типу та відділили ці алгоритми в окремі компоненти для їх простої модифікації та тестування.

11. Отримана в результаті програма дозволяє зберігати концептуальні зв’язки між елементами композиції, такими як окремі ноти та партії, завдяки чому швидше вносити концептуальні зміни. Це означає, що якщо нам в якийсь момент потрібно змінити ритмічний малюнок на певному відрізку партії, нам не потрібно редагувати окремо кожну ноту, а достатньо відредагувати або замінити умову, яка відповідає за цей параметр. Всі необхідні дії на рівні конкретної реалізації програма виконає самостійно.

12. Проведено тестування розроблених алгоритмів з дослідженням приросту ефективності запропонованого підходу порівняно з загальноприйнятим. Встановлено, що швидкість впровадження змін з застосуванням програмних алгоритмів може зростати в десятки, сотні та тисячі разів, порівняно з ручним редагуванням.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Wooller, R., A. R. Brown, et al. (2005). A framework for comparison of processes in algorithmic music systems. Generative Arts Practice, Sydney, Creativity and Cognition Studios Press, pp. 109-124.

2. Biles, A. 2002a. GenJam in Transition: from Genetic Jammer to Generative Jammer. In International Conference on Generative Art, Milan, Italy.

3. Chomsky, N. 1956. Three models for the description of language. IRE Transations on Information Theory, 2: 113-124.

4. Chomsky, N. 1957. Syntactic structures. The Hague, the Netherlands: Mouton.

5. Lerdahl, F. and R. Jackendoff. 1983. A generative theory of tonal music. Cambridge, Mass: MIT Press.

6. Eno, B. 1996. Generative Music. http://www.inmotionmagazine.com/eno1.html (accessed 27.10.2021).

7. Dorin, A. 2001. Generative processes and the electronic arts. Organised Sound, 6 (1): 47-53.

8. EM-VISIA 2011. Програмка концерту 15 травня 2011 у Великому залі НМАУ.

9. Зарипов Р.Х. Кибернетика и музыка. Москва : Стереотип, 2019. 238 с.

10. Hideo N. Mozart Musical Game in C K. 516f*. 28.12.1997. URL: http://www.asahi-net.or.jp/~rb5h-ngc/e/k516f.htm (accessed: 09.11.2021).

11. Pearson M. Generative art: a practical guide using Processing. 2011. *Manning publications co.* URL: http://www.manning.com/pearson/GenArt-Sample-Chapter-1.pdf (accessed: 08.11.2021).

12. Беличенко В.В., Горбунова И.Б. Феномен музыкально-компьютерных технологий в обучении информатике музыканта (в условиях перехода на новые образовательные стандарты) : монография. СПб., 2012. 220 с.

13. Горбунова И.Б., Заливадный М.С., Кибиткина Э.В. Музыкальное программирование : учеб. пособие. СПб., 2012. 195 с.

14. Савенко С. Карлхайнц Штокхаузен. 12.08.2006. URL: http://opentextnn.ru/music/personalia/schtokhauzen/ (дата звернення: 07.11.2021).

15. Eldred M. The Quivering of Propriation: A Parallel Way to Music. URL: http://www.arte-fact.org/qvrpropn.html (accessed: 08.11.2021).

16. Ihmels T., Riedel J. The methodology of generative art. Media Art Net, Germany [Website] URL: http://www.medienkunstnetz.de/themes/generative-tools/generative-art/ (accessed: 08.11.2021).

17. Горбунова И. Б. «Автоматические композиции» как предшественники применения кибернетики в музыке. *Общество: философия, история, культура*. 2016. №9. С. 97-101.

18. Заливадный М.С. Теоретические проблемы компьютеризации музыкальной деятельности (опыт комплексной характеристики) : автореф. дис. ... канд. искусствоведения. СПб., 2000. 24 с.

19. Sychra A. Hudba ocima vedy. Praha, 1965. 300 p.

20. Boden M.A. What is generative art? University of Technology Sydney: URL: http://research.it.uts.edu.au/creative/eae/intart/pdfs/generative-art.pdf (accessed: 08.11.2021).

21. Лукичев Р. В. К проблеме классификации generative art: разработка понятийного аппарат. *Наука телевидения.* 2019. №15.3. DOI: 10.30628/1994-9529-2019-15.3-11-31.

22. Марков Б.В., Ярочкин Д.А. Музыка перед вызовами цифрового общества. *Журнал СФУ. Гуманитарные науки.* 2021. №6. С. 810-821.

23. Цифрова звукова робоча станція (англ. Digital Audio Workstation, DAW). URL: https://en.wikipedia.org/wiki/Digital_audio_workstation. (дата звернення: 07.11.2021)

24. Mubert online service. URL: https://mubert.com/.(accessed: 08.11.2021).

25. AIWA online-resource. URL: https://www.aiva.ai/.(дата звернення: 07.11.2021)

26. Рафгарден Т. Совершенный алгоритм. Спб : Питер, 2021. 304 с.

27. Гурин Н. И., Крылова Т. А. Интернет-приложение на платформе Node для электронной биржи перевозок. *Труды БГТУ.* Серия 3: Физико-математические науки и информатика. 2016. №6 (188). С. 178-180.

28. Мавлютов А. Р., Выдрин Д. Ф., Махнёва А. О. Самые востребованные языки программирования. *Academy*. 2017. №1 (16). С. 12-14.

29. Чепегин И. Д. Серверный JavaScript - преимущества и недостатки node. JS. *Вестник науки и образования.* 2020. №12-1 (90). С. 18-19.

30. Cope, D. 1996. Experiments in Musical Intelligence. Madison, Winconsin: A-R Editions.

31. Cope, D. and D. R. Hofstadter. 2001. Virtual music : computer synthesis of musical style. Cambridge, Mass.: MIT Press.

32. Desain, P. and H. Honing. 1992. Music, mind and machine : studies in computer music, music cognition and artificial intelligence. Amsterdam: Thesis Publishers.

33. Developer Survey Results 2019. URL: https://insights.stackoverflow.com/survey/2019/ (accessed: 08.11.2021).

34. Dodge, C. and T. A. Jerse. 1997. Computer Music. 2nd ed. New York: Schirmer Books.

35. Hiller, L. and L. Isaacson. 1958. Musical Composition with a High-Speed Digital Computer. In Machine Models of Music.

36. IBM-Computer-Music-Center. 1999. Music Sketcher. IBM. (accessed 12th of December, 2004).

37. Laurson, M. 1996. Patchwork. Helsinki: Sibelius Academy

38. Lippe, C. 1997. Music for piano and computer: A description. Information Processing Society of Japa SIG Notes, 97 (122): 33-38.

39. Loy, G. and C. Abbott. 1985. Programming languages for computer music synthesis, performance and composition. ACM Computing Surveys (CSUR), 17 (2): 235-265.

40. Microsoft. 2001. DirectMusic Producer 5.3.0.900. Microsoft. http://www.musicmachines.net/ (accessed 8/03/04).

41. Mozer, M. 1994. Neural network music composition by prediction: Exploring the benefits of psychoacoustic constraints and multiscale processing. Connection Science.

42. Pachet, F. 2002b. Playing with Virtual Musicians: the Continuator in Practice. IEEE Multimedia, 9 (3): 77-82

43. Papadopoulos, G. and G. Wiggins. 1999. AI Methods for Algorithmic Composition: A Survey, A Critical View, and Future Prospects. In AISB'99 Symposium on Musical Creativity, Edinburgh.

44. Puckette, M. and D. Zicarelli. 1990. MAX. Opcode. http://www.cycling74.com (accessed: 08.11.2021).

45. Roads, C. 1996. The Computer Music Tutorial. Cambridge, Massachusetts: MIT Press.

46. Russell, S. and P. Norvig. 2004. Artificial Intelligence: A Modern Approach. 2 ed. New Jersey: Prentice Hall.

47. Truax, B. 1976. A Communicational Approach to Computer Sound Programs. Journal of Music Theory, 20 (2): 227-300.

Winkler, T. 1998. Composing Interactive Music. Cambridge, Massachusetts: MIT Press.

ДОДАТКИ

Додаток А

Тезиси з XVII Всеукраїнській науковій конференції молодих вчених та студентів «Наукові розробки молоді на сучасному етапі» (м.Київ, КНУТД, 2019)

УДК 004.8

ПЕРСПЕКТИВИ ШТУЧНОГО ІНТЕЛЕКТУ ТА КОМП'ЮТЕРИЗАЦІЇ В МИСТЕЦТВІ

студ. Н.М. Куць, гр. МгАК-18.
науковий керівник доцент к.ф.-м.н. Ю.М. Пилипенко
Київський національний університет технологій та дизайну

Мета і завдання. Складно знайти сфери людського життя на які не вплинула поява комп'ютера. Мистецтво не стало виключенням. Велика кількість програмного забезпечення була створена з метою зекономити час людям пов'язаним з творчою діяльністю. Раніше композитору, довелося б протягом місяця чекати музикантів, що розучують партії та винаймати зал, аби вперше насолодитися звучанням своєї симфонії. Зараз все, що необхідно – потужний комп'ютер зі встановленим програмним забезпеченням та вдосталь вільного часу.

Об'єкт та предмет дослідження. Сучасні програми навчилися виконувати надскладні операції, над якими раніше працювали команди професіоналів. Складність алгоритмів які комп'ютер може опрацювати вражає. Ще більших результатів дозволило досягнути впровадження штучного інтелекту («Ш І»). Сучасний «Ш І» побудований на основі нейронних мереж, котрі аналізують вхідну інформацію та знаходять в ній закономірності, неочевидні для людини. Такий підхід дозволяє запам'ятовувати зв'язки між різними об'єктами і в майбутньому використовувати їх для створення прогнозів чи генерації нових даних. Це дозволяє використовувати надскладні алгоритми без їх фактичного опису.

Результати дослідження. З впровадженням «Ш І», програми почали виконувати операції нового рівня складності. У 2017 році компанія Adobe презентувала функцію для видалення зайвих фрагментів на відео. Нейронна мережа аналізує зображення та видаляє обрані елементи, заповнюючи прогалини фоном, що автоматично генерується на основі контексту зображення. Можна лише уявити наскільки складною була б реалізація того ж алгоритму без використання нейронних мереж.

Одним із перспективних напрямків розвитку «Ш І» є «генеративне» мистецтво. Цей напрям не новий, однак з розповсюдженням «Ш І», набуває широкої популярності. Нейронні мережі аналізують існуючі твори, знаходять закономірності та на їх основі створюють нові. Часто результати не поступаються оригіналу і, що не менше важливо, лишаються унікальними. Але чи можна називати результати роботи «Ш І» витвором мистецтва? Більшість сучасних мистецтвознавців та філософів, вважають що ні. Основних причин дві.

По-перше, для мистецтва важлива ідея автора, його послання до інших. Потрібне усвідомлення, яке в «Ш І» відсутнє.

По-друге, «Ш І» не має комплексного бачення. Він обмежений алгоритмом і набором даних з яким працює. Таким чином, результати його роботи завжди знаходитимуться в межах алгоритму.

З появою, так званого, «сильного Ш І», який повинен отримати самосвідомість, обидва обмеження мусять зникнути, однак, за даних умов, в генеративній творчості, витворами мистецтва вважають самі алгоритми, а не результати їх роботи. Однак це не означає, що так мусить бути завжди.

Американський психолог Колін Мартіндейл, займався дослідженнями креативного мислення та розділив його на два процеси: первинний та вторинний.

Первинний процес служить пошуку нового. Він пов'язаний з мріями та зануренням в роздуми. В ході цього процесу, розум, явно або неявно, будує асоціації в межах відомих концепцій, порівнює їх, поєднує та генерує нові. В первинному процесі значну роль відіграють емоції.

Вторинний процес відноситься до звичного нам стану свідомості. Його характеристиками виступають абстрактність, логіка, орієнтація на вимоги зовнішньої реальності. В процесі творчості первинне мислення генерує «сирий» матеріал, а вторинне допомагає надати йому бажаної форми.

Комп'ютерні алгоритми відмінно справляються з задачами вторинного процесу мислення, тому їм можна повністю делегувати реалізацію. Це дозволить програмі виконати увесь об'єм робіт автоматично, після нетривалого планування. Ця концепція особливо корисна в сфері мистецтва, де задумка відома до реалізації.

Розглянемо перспективи такого рішення на прикладі музики. Існує багато програм для створення композицій. Вони володіють широким спектром корисних функцій, здатних прискорити втілення ідей. Однак, сам процес досі покладається на автора. Наприклад, композитор пише музику до фільму. Він визначився з розвитком ідеї та знає, де напруга зростатиме, де спаде, а де залишиться грати один інструмент. До моменту реалізації, йому відомо, що повинно вийти в результаті. Він вирішив почати з підбору інструментів, потім перейшов до написання партій і нарешті до обробки, аби все зазвучало гармонійно. На кожному етапі він працює алгоритмічно, послідовно втілюючи задумане в реальність. Якщо поєднати ідеї автора та здатність «Ш І» генерувати музику, можна повністю автоматизувати процес реалізації. Це нагадує ситуацію спілкування двох композиторів, де один дає настанови, а інший їх виконує. Враховуючи можливості обчислювальної техніки, швидкість такого процесу зросте в рази.

Для опису ідеї підходить принцип обмежень. Обмеженнями слугуватимуть аспекти музики: гармонія, підбір музичних інструментів, ефекти і т.п. Від глибини опрацювання кожного з цих параметрів, залежить відповідність результату до задуманого.

Ще одна з перспектив – створення персональної музики. В майбутньому музика буде підлаштовуватися під вас аби покращити ваш стан та задати «правильний» настрій. Це може бути як корисним так і шкідливим, в залежності від того, на основі чого буде прийматися рішення про «правильність», того чи іншого настрою.

Таким чином, поєднання «Ш І» з можливостями програмного забезпечення, стане наступним кроком в сфері автоматизації креативних процесів. Провідне значення в цьому випадку займе ідея, а можливості експериментів та нових відкриттів багатократно зростуть.

Висновки. У висновку відзначимо, що мета роботи – не позбавити музикантів шматка хліба, а створити для них вдосконалений інструмент здатний зробити їх працю простішою та ефективнішою. За цього підходу, ідея композиції мусить зайняти провідне значення, а реалізувати її стане на порядок простіше.

Ключові слова: штучний інтелект, нейронні мережі, «генеративне» мистецтво.

Список використаних джерел

1. Шакла Нишант Машинное обучение и TensorFlow. / Нишант Шакла. – М. «Питер». – 2019. – 336 с.

2. Студенческий научный форум: Анализ теории креативности Колина Мартиндейла — Режим доступа: https://scienceforum.ru/2015/article/2015011510

3. Официальный сайт научно-популярного издания N+1— Режим доступа: https://nplus1.ru/news/2017/10/24/Adobe-Cloak/amp

Додаток В

Текст статті, що готується до друку у журналі «Інженерія»

УДК 004.8

КОМП'ЮТЕРНО-ІНТЕГРОВАНА СИСТЕМА ГЕНЕРАЦІЇ МУЗИЧНИХ КОМПОЗИЦІЙ

Куць Н. М., Пилипенко Ю. М.

Київський національний університет технологій та дизайну

Мета. *Пошук та реалізація засобів інтеграції концептуального підходу до програм з написанням музики.*

Методика. *Проведено дослідження розвитку музичної теорії та генеративного мистецтва. Виокремлено набір концептуальних складових музичної композиції. В проектуванні програмної системи використано принципи чистої архітектури SOLID. В проектуванні алгоритмів покладено в основу парадигму «розділяй та володарюй».*

Результати. *Розроблено програму, яка дозволяє працювати над створенням музичної композиції на концептуальному рівні – редагуючи концептуальні зв'язки, а не конкретні параметри нот.*

Наукова новизна. *Запропоновано новий підхід до написанням музики на комп'ютері, що базується на використанні концептуальних закономірностей замість використання конкретних значень, в процесі роботи над музичною композицією.*

Практична значимість. *Створено систему, здатну встановлювати та зберігати зв'язки між різними складовими композиції. Зв'язки описані елементами, що базуються на музичній теорії. Ця система дозволяє швидше вносити концептуальні зміни – при редагуванні глобальних концепцій, таких як ритмічний малюнок або тональність, зміни на рівні реалізації відбуваються автоматично. За рахунок часткової автоматизації програма дозволяє вносити зміни на необхідному рівні деталізації і дозволяє робити це набагато швидше за ручне редагування.*

***Ключові слова**: : генеративне мистецтво, програмні алгоритми, теорія музики, музичний контекст, музична композиція, музичне програмне забезпечення, генерація музики.*

Написання музики – креативний процес, в ході якого композитор розповідає історію, доступними в музиці засобами самовираження.

Від появи першого музичного запису і до нашого часу, технології створення музики значно еволюціонували. На це вплинули безліч процесів, в тому числі глобалізація та комп'ютеризація. З появою протоколу MIDI [1], збереження та редагування нотної інформації стало значно простішим процесом ніж за використання писемної нотації. Також це дозволило відтворювати записані нотні фрагменти різними тембрами, не втрачаючи при цьому якості.

З часом з'явилися спеціальні комп'ютерні програми, які дозволяли редагувати MIDI-фрагменти. Ці програми отримали назву MIDI-редактори. Пізніше MIDI-редактори еволюціонували в програми нового типу – DAW (англ. «Digital Audio Workstation» – «цифрова звукова робоча станція») [2]. DAW зібрали в одному флаконі все необхідне для створення музики на комп'ютері. Однак попри широкий функціонал, ці програми досі не є ідеальними.

Проблема сучасних підходів до написанням музики, які пропонують більшість музичних програм, полягає у відсутності в них засобів, які б дозволяли працювати над композицією не лише на рівні конкретних нот, а й на рівні музичного контексту, тобто одразу з концептуальними даними. Всі сучасні програми для написання музики взаємодіють з композитором мовою конкретних значень, тоді як сам автор в процесі реалізації ідеї розмірковує концепціями, тобто зв'язками, які мають більш абстрактну природу.

Конкретними значеннями в музиці можна вважати ноти. Нота, як одиниця музичного матеріалу, слугує позначенню на письмі сукупності фізичних параметрів звуку – частоти коливання, тривалості коливального процесу, його амплітуди. Нота дозволяє передавати цю інформацію у вигляді простого символу без необхідності заглиблюватися в деталі кожного процесу окремо. Таким чином – нота виступає абстракцією фізичних характеристик звукової хвилі .

З сукупності нот будуються музичні фрази, з фраз − партії, а з партій − цілі композиції.

Важливо що зі сторони ідеї, конкретні ноти не мають такого великого значення, як зв’язки, які встановлюються між ними шляхом сприйняття музики в часі. Це те ж саме що значення окремих слів стає зрозумілим лише в контексті речення, а значення речення − в контексті тексту. При цьому між словами в реченні виникає зв’язок.

Якщо головна мета взаємодії автора музики та програми для її написання – надання закінченої форми ідеям автора, то досконалою така система могла б стати, якби дозволяла працювати не лише на рівні конкретних значень, а й на рівні контексту. Тобто якби з нею можна було спілкуватися не лише мовою конкретних нот (редагуючи їх параметри), а одразу мовою ідей та концепцій – повідомляючи з яких концепцій має складатися композиція на певному часовому відрізку.

Постановка завдання

Знайти шляхи інтеграції концептуального підходу до комп’ютерних програм з написання музики. Дослідити існуючі на даний момент програми, їх можливості та обмеження. Виокремити концептуальні складові на які можна спиратися в ході роботи над музичною композицією. Розробити програмну модель, яка б дозволяла працювати над композицією як на рівні деталей реалізації, так і на рівні музичного контексту.

Результати досліджень

Проведено дослідження розвитку музичної теорії та відібрано концептуальні складові музичної композиції. Прикладами музичних концепцій можуть бути ритмічна, інтервальна, гармонійна закономірності, тощо.

За історію розвитку музики, виникла велика кількість таких концепцій, які з часом були оформлені до музичної теорії. Ці концепції в теорії музики представлені у вигляді ладів, тональностей, ритму, акордів, їх послідовності тощо. Кожна подібна концепція надає певних характеристик твору. Наприклад, різні послідовності акордів можуть звучати сумно або весело, спокійно чи хвилююче. Тобто кожна така складова – ніби пензлик, що приносить до загальної картини свої фарби.

Сучасне програмне забезпечення для написання музики дозволяє працювати з частиною цих закономірностей, однак більшість роботи досі знаходиться на відповідальності автора. Йому самостійно доводиться писати всі партії, редагуючи кожну

окремо взяту ноту. Звісно, в кожній програмі реалізовано власний набір функцій, які здатні полегшити цей процес. Однак концептуальні зв’язки досі зберігаються лише в уяві автора, а програма ніяк їх не інтерпретує. Тому, хоча ці функції і є корисними, однак впровадження прямої роботи з концепціями дозволило б досягнути значно більшої ефективності.

Ближче всього до реалізації концептуального підходу знаходяться програми з автоматичного написання музики – музичні генератори.

Генеративна музика (англ. Generative music) − напрямок музичної творчості, в основі якого лежить використання алгоритмів, а також прагнення досягти різноманітності в художньому творі.

За Р. Вуллером можна виокремити чотири напрями розвитку генеративної музики[3]:

1. Лінгвістичний/структурний. Використовує алгоритми, що націлені на генерацію структурно-зв'язного матеріалу, спираючись на досягнення Генеративної граматики Чомскі [5], а також музикознавчі розробки Фреда Лердала і Рея Джекендоффа. Алгоритми базуються на деревовидній структурі.

2. Інтерактивний/поведінковий. Музика породжується системою, що нібито не має входів, і характеризується як "така, що не трансформується" (not transformational). Як приклад наводиться «Generative Music 1» Браяна Іно [7].

3. Креативний/процедурний. Процеси створення музики розробляються або ініціюються композитором. Як приклад наводиться It's Gonna Rain Стіва Райха та In C Террі Райлі.

4. Біологічний/послідовний. Недетермінована музика [4], або музика, яку не можна відтворити. Автори цієї ідеї порівнюють роботу композитора з вченими-селекціонерами, що займаються виведенням нових біологічних видів. Як приклад наводиться "Вірусна симфонія" Йозефа Нехвatала – композиція жанру noise, в якій за основу взято модель розмноження вірусів.

Програми генератори музики створюють композицію самостійно «з нуля», або з мінімальною участю оператора-людини. Наприклад, користувач обирає жанр, або список творів, що йому подобаються, а все інше реалізує програма.

Перевагами таких програм є:

- Концептуальний підхід. В процесі генерації вони завжди спираються на певні музичні закономірності, з якими уміють працювати.

- Можливість редагування результатів по закінченню генерації. Що додає їм більше творчої свободи. Однак ця можливість реалізована не у всіх програмах.

- Достойні результати генерації, які складно відрізнити від написаних людиною.

Однак попри переваги ці програми мають і недоліки:

- Такі програми «самостійні» в прийнятті рішень, тому результати їх роботи складно вважати заслугою автора. Більшість програм потребує вказати лише жанр чи настрій майбутньої композиції, а все інше виконають алгоритми.

- Обмежені у варіативності. Всі результати формуються за однаковим принципом, оскільки більшість рішень все одно приймає штучний інтелект. Це також слугує причиною, чому на академічному рівні твором вважається сама програма здатна генерувати музику, а не результати її роботи.

- Не зберігається інформація про зв’язки, тому на концептуальному рівні такі програми працюють лише перед запуском генерації. Після неї тільки на рівні редагування конкретних значень – параметрів окремих нот або семплів.

Таким чином програми повної генерації складно вважати надійним інструментом для роботи над композицією на концептуальному рівні, оскільки в більшості вони керуються не волею автора, а жорстко прописаними в них програмними алгоритмами.

Щоб розробити модель яка впроваджуватиме концептуальний підхід і при цьому буде вдосталь гнучкою до застосування, ми склали список вимог до такої системи, з метою в подальшому реалізувати програму, яка їх дотримуватиметься. Ось список цих вимог:

- Програма має зберігати концептуальні зв’язки між різними елементами композиції.

- При взаємодії з програмою автор повинен мати можливість взаємодіяти з концептуальними елементами і управляти ними та налаштовувати так само просто як і редагування окремо взятих нот.

- Зв’язки мають зберігатися як на рівні окремих партій так і на рівні цілої композиції. Тобто встановлюватися як на відрізках партій (між окремими нотами) так і між партіями.

- Програма має володіти достатнім рівнем гнучкості. Це означає, що автор зможе працювати як на рівні більш конкретної реалізації так і на рівні контексту, при цьому концептуальні зв’язки між елементами композиції будуть зберігатися в будь-якому випадку.

- Програма має бути ефективною, тобто не виконувати зайвої роботи. Це більшою мірою стосується алгоритмів і означає, що при редагуванні певних параметрів, повторна генерація має застосовуватися не до всіх складових композиції, а тільки пов’язаних з відредагованими.

В ході роботи над програмою, ми розробили кілька підходів, які стали ключовими.

Робота на концептуальному рівні означає відбір та налаштування певних закономірностей у вузько відібраній області. Ми дослідили область музичної теорії та виділили закономірності, які можуть стати концептуальними складовими. Такими закономірностями в програмі стали: тональна, метрична, інтервальна, гармонійна, ритмічна та динамічна.

Закономірності було відділено до наборів умов. Кожну умову можна вважати попередньо заготовленим шаблоном – ритмічний малюнок, інтервальна послідовність тощо. Кожна така складова має зберігатися в програмі. В процесі роботи над композицією, користувач відбирає серед наявних шаблонів ті, які краще розкривають суть його ідеї.

Ці шаблони він переміщує до відповідних комірок, кожна з яких приймає умови тільки свого типу. Наприклад, ритмічна умова може бути розміщена в комірці для ритмічних умов. Такі комірки в термінах програми отримали назву *вузли* (*Nodes*) генерації.

Шаблон умови певного типу є концептуальним представленням зв’язків між згенерованими на його основі нотами. Для встановлення зв’язків між партіями реалізовано принцип наслідування. Наслідування в даному контексті означає, що існує партія, яка має певні умови генерації (набір шаблонів різного типу). Якщо ця партія має партію-наслідника, то після генерації батьківської партії, ноти отримані в результаті будуть передаватися у вигляді вхідних даних, на які будуть накладатися умови уже з дочірньої партії.

Таким чином процес генерації відбувається послідовно – спершу генеруються батьківські партії, потім їх дочірні, і потім дочірні дочірніх і т. д. Головний принцип – спершу мають згенеруватися партії вищого порядку, від яких залежить результат генерації наслідників.

Програма реалізована мовою програмування Kotlin. Розглянемо архітектуру проекту.

Найвищим рівнем з яким працює програма є клас композиції. Він включає в себе список треків та обслуговуючі методи. Кожен трек складається з сегментів – певних

часових відрізків треку. Сегменти в свою чергу складаються з вузлів генерації, а кожен вузол містить або не містить умову для генерації.

Таким чином в процесі генерації відбувається послідовне обходження треків композиції в порядку ієрархії (від батьківських до найглибших дочірніх). Для кожного треку послідовно обходяться всі сегменти, а в сегментах – вузли, − які відповідають за накладання умови певного типу.

У вузлі кожного типу реалізовано алгоритм, який знає як застосувати умову до нотного відрізку, переданого на обробку.

Розглянемо загальний алгоритм генерації.

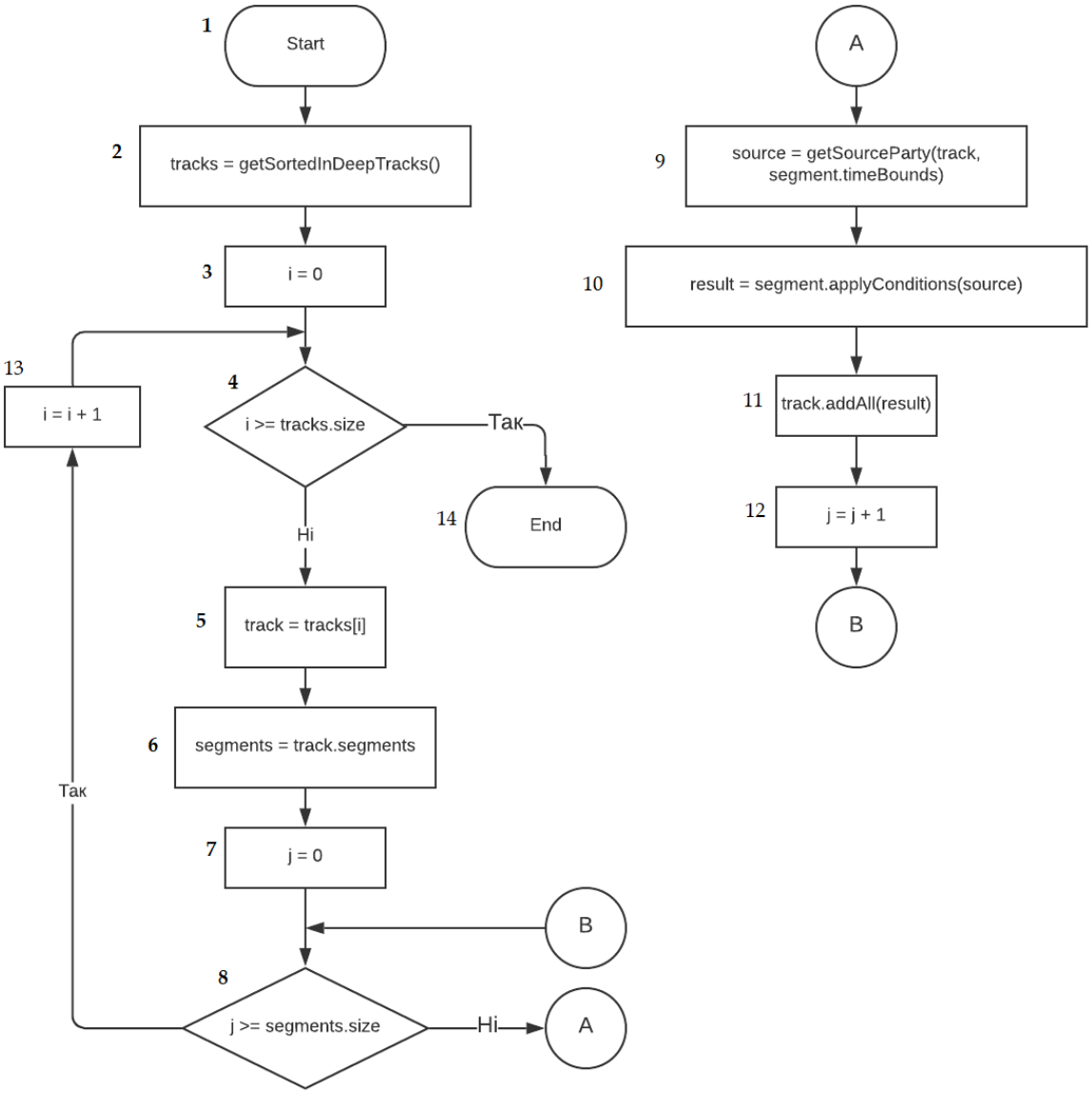

Рис. 1. Блок схема алгоритму генерації на рівні композиції

В блок-схемі алгоритму генерації на рівні композиції (рис. 1) створюємо нову змінну *tracks*, в яку зберігаємо відсортований «в глибину» список треків. В циклі проходимося послідовно по всім трекам. У внутрішньому циклі для кожного треку проходимося по всім його складовим сегментам.

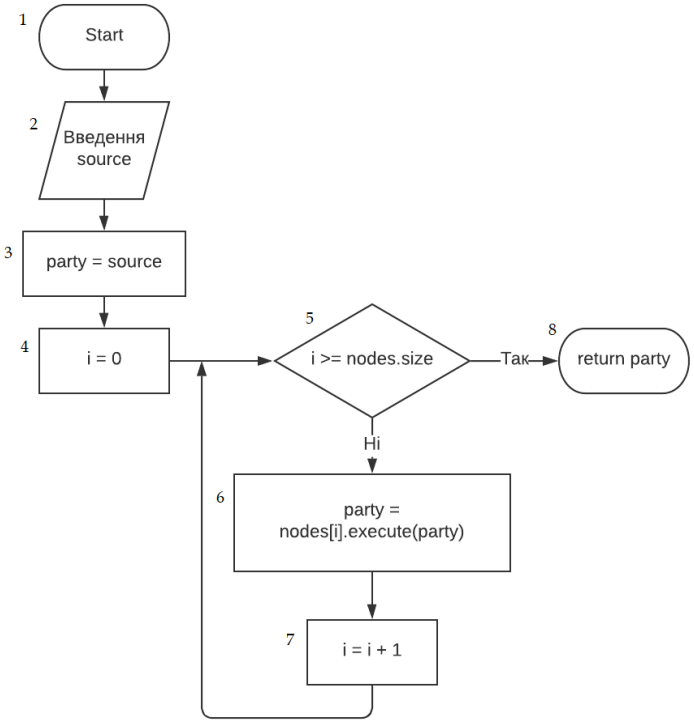

Рис. 2. Блок-схема генерації на рівні сегменту

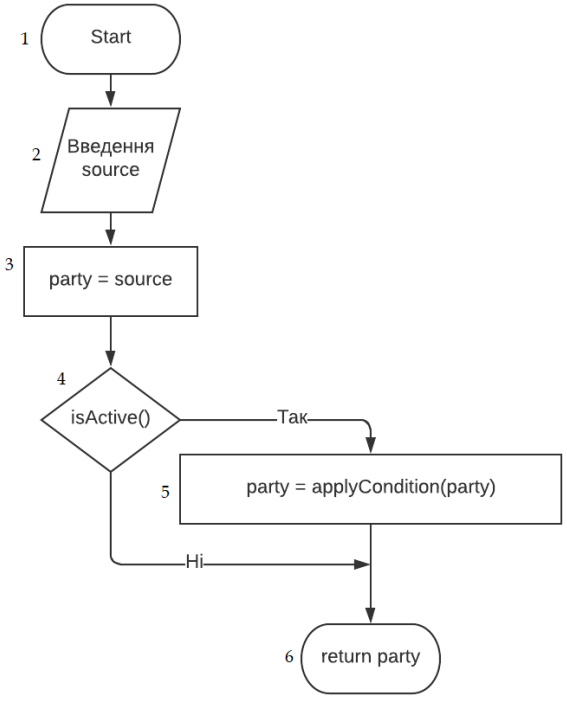

Рис. 3. Блок-схема генерації на рівні вузла

Отримуємо «базову» партію викликаючи функцію *getSourceParty(...)*, передаючи до неї аргументами поточний трек *track* та часові обмеження сегмента *segment.timeBounds*. *TimeBounds* – простий клас обгортка з двома цілочисленними полями *start* та *end*.

Викликаємо функцію *applyConditions(source),* об'єкта *segment*, передаючи аргументом партію отриману на минулому кроці. Результат виклику функції зберігаємо у змінну *result*.

Викликаємо *track.addAll(result)*. Функція *addAll* додає всі ноти партії переданої у вигляді аргументу, до внутрішньої партії треку.

В коді ми ще звертаємося до функції *getSourceParty(...)*, яка надає партію для подальшого накладання умов.

Генерація на рівні сегменту розпочинається з виклику функції *applyConditions*. Аргументом функції передається відрізок партії. Сегмент не знає що це за відрізок і послідовно проходить по всім вузлам. Змінна *party* оновлює своє значення після застосування умов кожного вузла генерації. В результаті своєї роботи функція повертає партію згенеровану останнім вузлом (рис. 2).

На рівні вузла генерація починається з виклику функції *execute* (рис. 3).

Спершу перевіряється активність вузла. Умова активності вузла залежить від двох факторів – чи увімкнено вузол в інтерфейсі користувача та чи містить вузол шаблон умови. Якщо вузол не містить умови, він автоматично перестає бути активним і на інтерфейсі користувача і в процесі генерації. При цьому в процесі генерації вузол ніяк не взаємодіє з партією, що передається йому у вигляді аргументу, а просто повертає її в тому ж вигляді, в якому вона прийшла на вхід.

Тут ми не будемо розглядати конкретних алгоритмів накладання умов, оскільки це об'ємний розділ який потребує окремого розгляду.

Висновки

Запропоновано підхід до написання музики, який базується на роботі з концептуальними даними, в доповнення до роботи на рівні конкретних нот. Реалізовано програму яка слугує ілюстрацією реалізації цього підходу. Програма показала себе ефективною в тестах при внесенні концептуальних змін до композиції, таких як ритмічний малюнок або інтервальна закономірність. Автоматичне внесення змін програмою виконувалося значно швидше ніж внесення тих самих змін при ручному редагуванні.

Список використаних джерел

1. MIDI.
URL: https://uk.wikipedia.org/wiki/MIDI.

2. Цифрова звукова робоча станція (англ. Digital Audio Workstation, DAW). URL: https://en.wikipedia.org/wiki/Digital_audio_workstation. (дата звернення: 07.11.2021)

3. Wooller, R., A. R. Brown, et al. (2005). A framework for comparison of processes in algorithmic music systems. Generative Arts Practice, Sydney, Creativity and Cognition Studios Press, pp. 109-124.

4. Biles, A. 2002a. GenJam in Transition: from Genetic Jammer to Generative Jammer. In International Conference on Generative Art, Milan, Italy.

5. Chomsky, N. 1956. Three models for the description of language. IRE Transations on Information Theory, 2: 113-124.

6. Eno, B. 1996. Generative Music. http://www.inmotionmagazine.com/eno1.html (accessed 27.10.2021)

References

1. MIDI.
URL: https://uk.wikipedia.org/wiki/MIDI.
[in English]

2. Digital Audio Workstation, DAW. URL: https://en.wikipedia.org/wiki/Digital_audio_workstation. [in English]

3. Wooller, R., A. R. Brown, et al. (2005). A framework for comparison of processes in algorithmic music systems. Generative Arts Practice, Sydney, Creativity and Cognition Studios Press, pp. 109-124. [in English]

4. Biles, A. 2002a. GenJam in Transition: from Genetic Jammer to Generative Jammer. In International Conference on Generative Art, Milan, Italy. [in English]

5. Chomsky, N. 1956. Three models for the description of language. IRE Transations on Information Theory, 2: 113-124. [in English]

6. Eno, B. 1996. Generative Music. http://www.inmotionmagazine.com/eno1.html (accessed 27.10.2021) [in English]

Pylypenko Yurii
ORCID: http://orcid.org/0000–0003-4093–7298
pyl20453@gmail.com
Kyiv National University of Technologies and Design

Kutz Nazar
vitaminkuna@gmail.com
Kyiv National University of Technologies and Design

Компьютерно-интегрированная система генерации музыкальных композиций

Куць Н.Н., Пилипенко Ю. Н.
Киевский национальный университет технологий и дизайна

Цель. *Поиск и реализация способов интеграции концептуального подхода к программам для создания музыки.*

Методика. *Проведены исследования развития музыкальной теории и генеративного искусства. Выделен набор концептуальных составляющих музыкальной композиции. В проектировании программной системы использованы принципы чистой архитектуры SOLID. В проектировании алгоритмов положено в основу парадигму «разделяй и властвуй».*

Результаты. *Разработана программа, позволяющая работать над созданием музыкальной композиции на концептуальном уровне – редактируя концептуальные связи, а не конкретные параметры отдельных нот.*

Научная новизна. *Предложен новый подход к написанию музыки на компьютере, основанный на использовании концептуальных закономерностей вместо использования конкретных значений, в процессе работы над музыкальной композицией.*

Практическая значимость. *Создана система, способная устанавливать и сохранять связи между разными составляющими композиции. Связи описаны элементами, основанными на музыкальной теории. Эта система позволяет быстрее вносить концептуальные изменения – при редактировании глобальных концепций, таких как ритмический рисунок или тональность, изменения на уровне реализации происходят автоматически. За счет частичной автоматизации программа позволяет вносить изменения на необходимом уровне детализации и позволяет делать это гораздо быстрее чем при ручном редактировании.*

Ключевые слова: *генеративное искусство, программные алгоритмы, теория музыки, музыкальный контекст, музыкальная композиция, музыкальное программное обеспечение, создание музыки.*

The computer-integrated music generation system

Kutz N.M., Pylypenko Y.N.

Kyiv National University of Technology & Design

Purpose. *Finding and implementing ways to integrate a conceptual approach to music production programs.*

Methodology*. Research into the development of musical theory and generative art. A set of conceptual components of a musical composition is selected. The design of the software system uses the principles of pure SOLID architecture. Algorithm design is based on the "divide and conquers" paradigm.*

Findings. *Created a program allows working on creating a musical composition at the conceptual level - by editing conceptual connections, rather than specific parameters of notes.*

Originality. *A program has been developed that allows you to work on creating a musical composition at a conceptual level - to edit conceptual relationships, and not specific parameters of individual notes, although the latter option also remains.*

Practical value. *A system has been created that can establish and maintain relationships between different components of the composition. The relationships are described by elements based on musical theory. This system allows you to make conceptual changes faster - when editing global concepts such as rhythm or tonality, changes at the implementation level occur automatically. Due to partial automation, the program allows you to make changes at the required level of detail and allows you to do this much faster than with manual editing.*

Keywords: *generative art, software algorithms, music theory, musical context, musical composition, musical software, music generation.*