https://doi.org/10.30857/2786-5371.2022.6.3

УДК 004.89

КУЦЬ Н. М., ПИЛИПЕНКО Ю. М.
Київський національний університет технологій та дизайну, Україна

КОМП'ЮТЕРНА ПРОГРАМА ГЕНЕРАЦІЇ МУЗИЧНИХ КОМПОЗИЦІЙ НА ОСНОВІ МЕЛОДІЇ АВТОРА

Мета. *Пошук та реалізація засобів інтеграції концептуального підходу до створення нових музичних творів.*

Методика. *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. *Розроблені алгоритми, написана та відлагоджена програма на мові програмування Kotlin, яка дозволяє працювати над редагуванням музичної композиції на концептуальному рівні – редагуючи концептуальні зв'язки, а не конкретні параметри нот. Об'єм програми близько 16 тисяч операторів. Тестування програми показало, що її використання дає можливість прискорити музичну обробку композиції приблизно в 10000 разів у порівнянні з ручним редагуванням. Зрозуміло, що остаточне рішення про вдалість відповідного редагування приймає автор, але в нього тепер з'являється можливість прослуховування різних варіантів обробленої композиції, на створення яких він, по суті, не витрачає час.*

Наукова новизна. *Запропоновано новий підхід до використання комп'ютеру при написані музичних композицій, коли на основі мелодії автору комп'ютерна програма генерує, згідно закладених шаблонів, дещо змінені акценти звучання твору (наприклад, по тембру, тональності, ритму і т. д.). Цей підхід реалізовано в вигляді комп'ютерної програми, використання якої в тисячі разів прискорює генерацію музичних творів, в яких звучить мелодія автора.*

Практична значимість. *Написаний програмний продукт дає змогу композитору ефективно використовувати нові можливості створення музичних композицій на основі мелодії автора.*

Ключові слова*: музичне програмне забезпечення; генерація музики; програмні алгоритми; об'єктно-орієнтоване програмування; мова програмування Kotlin.*

Вступ. 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.

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 місті все необхідне для створення музики на комп'ютері.

Проблема підходів до написанням музики, які пропонують більшість сучасних DAW, полягає у відсутності в арсеналі їх функцій засобів, які б дозволяли працювати над композицією не лише на рівні конкретних нот, а й на рівні музичного контексту. Нині DAW взаємодіють з композитором мовою конкретних значень, тоді як сам автор в процесі реалізації ідеї розмірковує концепціями, тобто зв'язками, які мають більш абстрактну природу. Саме такі закономірності, ми в даному випадку і називаємо «концептуальними зв'язками», а іншими словами – мелодією автора.

Конкретними значеннями в музиці можна вважати ноти. Нота виступає абстракцією фізичних характеристик звукової хвилі. З сукупності нот будуються музичні фрази, з фраз – партії, з партій – композиція. Важливо що зі сторони ідеї, конкретні ноти не мають такого великого значення, як зв'язки, які виникають 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.

Постановка завдання. Якщо головна мета взаємодії автора музики та програми для її написання – надання закінченої форми музичним ідеям, то така система могла б стати більш ефективною, якби дозволяла працювати не лише на рівні конкретних значень, а й на рівні контексту. Тобто дозволяла б редагувати закономірності, які пов'язують ноти як єдиний об'єкт, а не лише кожну ноту окремо.

Метою даної роботи було 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 дозволятиме працювати над композицією як на рівні деталей реалізації, так і на рівні музичного контексту.

Подальші впровадження повністю базуються на ідеї, що відділивши конкретні закономірності до різних областей і редагуючи та заміняючи ці закономірності окремо від інших (наприклад, окремо редагуючи ритмічний малюнок або змінюючи тональність), робота над композицією переміщується на концептуальний рівень, тобто не потребує від автора редагування кожної окремої ноти. Це стає можливим завдяки автоматизації рутинних процесів, оскільки програма самостійно редагує параметри нот, базуючись на встановлених закономірностях, ***не порушуючи при цьому задум композитора.***

При цьому зберігається можливість автора безпосередньо впливати на творчий процес, оскільки саме він відбирає та редагує складові закономірності, а програма лише виконує рутинні процеси з їх впровадження.

Повне створення нових творів які можна віднести до витворів мистецтва, будь то в живопису, літературі, музиці – творчий процес, який ніколи не стане рутинною справою, і яку зможе зробити лише обдарована Богом людина.

Результати досліджень. За історію розвитку музики, виникла велика кількість закономірностей, які з часом були оформлені у вигляді музичної теорії [3–7]. Ці концепції в теорії музики представлені у вигляді ладів, тональностей, ритму, акордів, їх послідовності тощо. Кожна подібна закономірність надає твору особисті характеристики. Наприклад, різні типи та послідовності акордів можуть мати різний настрій.

Ми вважаємо, що сімейства концептуальних закономірностей, які можуть стати складовими операторами програми стали: тональні, метричні, інтервальні, гармонійні, ритмічні та динамічні.

Був розроблений алгоритм та створена програма, що дозволяє встановлювати, зберігати та редагувати концептуальні зв'язки між елементами композиції, змінюючи їх по відповідним умовам, для знаходження оптимального звучання твору в цілому.

Програму написано мовою програмування Kotlin [8] на мобільній платформі Android [9]. На даний момент програма включає близько 17 000 рядків коду.

Для кожного сімейства умов в програмі реалізовано набори заготовок, які в термінах програми отримали назву *шаблони*. Шаблон представляє собою умову певного типу. Наприклад, ритмічний шаблон являє собою послідовність часових характеристик нот, а гармонійний – схему побудови акорду у вигляді висотного зміщення нот по ступеням ладу.

Програма реалізована таким чином, що в процесі роботи над композицією, користувач відбирає те звучання твору, яке найкраще відповідає його представленню, прослуховуючи звучання при використанні того чи іншого шаблону.

Зі сторони інтерфейсу користувача алгоритм роботи з програмою має наступну послідовність дій:

1. Користувач натискає на сегмент треку, який буде відредаговано;

2. Відкривається діалог зі списком вузлів генерації. Користувач має натиснути на один з вузлів, щоб відкрити новий діалог для вибору умови, яка буде зберігатися у вузлі.

3. Відкривається діалог вибору умови. Користувач може обрати один з шаблонів, що надає програма або реалізувати власний.

4. Після того як умова буде вибрана, воно збережеться до відповідного вузла (Node) генерації.

5. Далі, коли всі умови будуть відредаговані користувач може запустити нову генерацію натиснувши на кнопку «Generate», яка знаходиться на нижній панелі інструментів програми.

6. При натисканні на цю кнопку запускається глобальний алгоритм генерації, про який мова буде йти далі.

Звертаємо увагу, що саме автор, прослуховуючи відповідні партії після генерації, вибирає той варіант, який, на його думку, ближче до його задуму або подобається йому більше.

Шаблон умови є концептуальним представленням зв'язків між згенерованими на його основі нотами. Для установки концептуальних зв'язків між окремими партіями композиції використовується механізм наслідування треками. Термін «наслідування» в даному випадку не пов'язаний з ООП, а означає, що результат генерації одного треку (ми називаємо його батьківським) виступає вхідними даними для інших треків (дочірніх).

Розглянемо основні компоненти, з яких складається композиція, що дозволить нам краще розібратися з алгоритмами написаної програми.

На глобальному рівні композицію можна уявити як набір треків зі збереженням інформації про порядок їх наслідування. Трек складається з сегментів – обмежених в часі фрагментів з унікальним набором умов. Кожна умова зберігається в окремій комірці – вузлі генерації. В поточній реалізації кожен сегмент може включати лише по одному вузлу визначеного типу. При цьому вузли мають чітко визначену послідовність і при генерації спрацьовують в такому порядку: тональний, метричний, інтервальний, гармонійний, ритмічний і динамічний відповідно.

На рис. 1 представлено візуальний вигляд елементів композиції. Композиція виділена голубим кольорами, трек – червоним, окремі сегменти – зеленим.

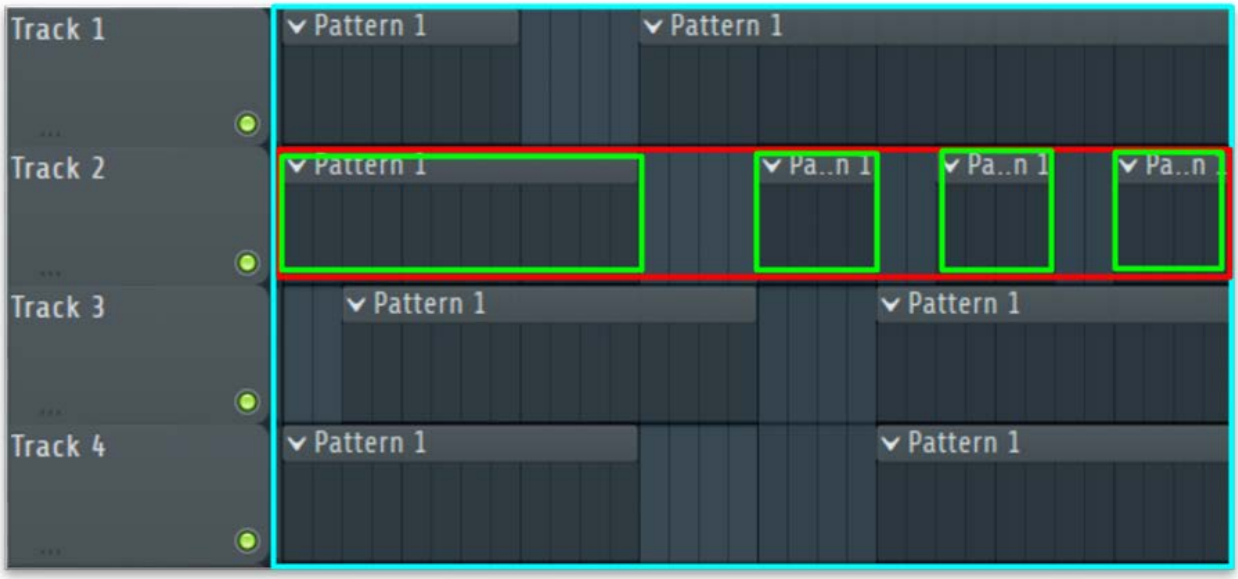

Рис. 1. **Візуальний вигляд компонентів композиції**

Алгоритми програми. Найголовнішою частиною програми є її алгоритми, оскільки вони вирішують поставлену в роботі задачу.

При розробці алгоритмів використовувалися положення наведені в серії книг «Досконалий алгоритм» [10].

В програмі розроблена та реалізована велика кількість алгоритмів, які відповідають за відображення інтерфейсу користувача, збереження даних, конвертацію в MIDI та багато інших. Однак найважливішими в рамках поставленої задачі є алгоритми генерації.

В рамках статті ми опишемо загальний алгоритм генерації та алгоритм накладання одного з типів умов.

Генерація на рівні композиції. 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. При запуску генерації у класу Composition викликається функція generate():

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

В наведеному фрагменті коду послідовно обходяться всі треки композиції. Першими будуть згенеровані батьківські треки, потім їх дочірні і так далі. Це забезпечується викликом hierarchy.getSortedInDeepTracks(), який повертає список треків відсортованих таким чином, що враховує послідовність їх наслідування.

Для кожного треку послідовно обходяться всі його складові сегменти.

Щоб отримати «базові ноти» на які будуть накладатися умови з вузлів генерації, викликається функція getSourceParty(). Ця функція отримує у вигляді аргументів поточний трек для якого відбувається генерація та часові обмеження сегменту – початок та завершення його звучання в форматі PPQN – базовому форматі для представлення часових характеристик в MIDI-протоколі.

Функція getSourceParty() знаходить батьківський трек для поточного треку та «вирізає» з нього фрагмент нот, обмежений часовими рамками сегменту. Якщо батьківський трек відсутній, функція генерує партію з єдиною нотою. Ця нота має часові характеристики, що співпадають з тривалістю поточного сегменту, інші характеристики задаються за замовчування.

Функція getSourceParty() повертає об’єкт Party, що є класом-обгорткою для масиву нот, відсортованих за початком звучання (подія «note on» в MIDI-протоколі). Отримана партія зберігається до змінної *source*.

На наступному кроці викликається функція класу Segment – applyConditions(), яка приймає у вигляді аргументу партію *source* отриману на минулому кроці. В середині цієї функції партія *source* видозмінюється у відповідності до умов, що зберігаються у вузлах сегменту. Результат виклику функції зберігається до змінної *result*, яка теж має тип Party.

На фінальному кроці ноти партії *result* зберігаються до поточного треку викликом track.addAll(result). Функція addAll() додає до треку список нот отриманих в результаті генерації поточного сегменту.

Таким чином при завершенні функції generate() в кожному треку зберігається партія нот, отриманих в процесі останньої генерації.

Генерація на рівні сегменту. На рівні сегменту генерація розпочинається з виклику функції applyConditions():

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

Спершу значення партії *source* зберігається до тимчасової змінної *party*.

Далі послідовно обходяться всі вузли генерації. Для кожного вузла викликається функція execute(), що отримує у вигляді аргументу поточну версію змінної *party*. Функція execute() видозмінює отриману партію та повертає її нову версію, зберігаючи до змінної *party* нове значення.

Після застосування всіх вузлів функція applyConditions() повертає змінну *party*, як результат своєї роботи.

Генерація на рівні вузла. На рівні вузла генерація стартує з виклику функції execute():

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

Перевіряється активність вузла, викликом функції isActive(). Вузол активний, якщо містить шаблон умови та активований користувачем на стороні інтерфейсу.

Якщо вузол активний, повертається результат виклику функції applyCondition(), якщо не активний – партія повертається без змін.

Функція applyCondition() не має загального вигляду, а реалізується для кожного типу вузла окремо. В ній міститься алгоритм застосування визначеного типу умов, залежно від якого редагуються певні характеристики існуючих нот або генеруються нові ноти.

Конкретна реалізація алгоритмів накладання умов. Клас Node виступає абстрактним класом для представлення загальних принципів накладання шаблонів всіх сімейств умов. Для окремого сімейства умов реалізована власна версія класу-наслідника базового класу Node.

Розглянемо реалізацію класу-наслідника Node на прикладі ритмічного типу умов.

Реалізація застосування ритмічної умови. Маємо два ритмічні малюнки: рис. 2 та рис. 3.

Рис. 2. **Ритмічні послідовності елементів партії та елементів умов**

Послідовність «1» представляє собою ритмічний малюнок нот існуючої партії. Послідовність «2» – ритмічний малюнок шаблону, що зберігається у вузлі.

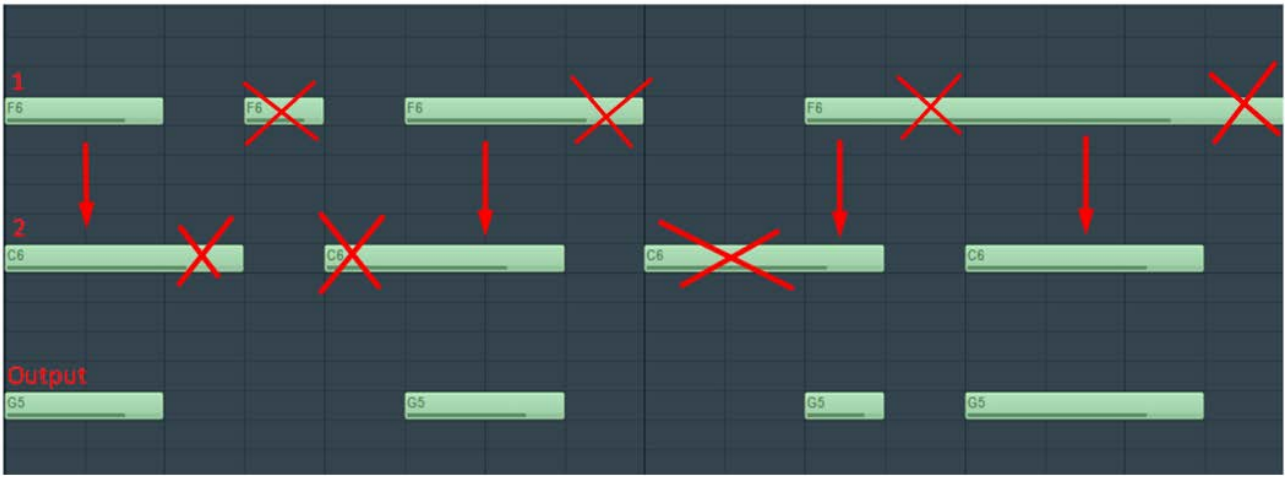

Рис. 3. **Накладання ритмічного шаблону**

Фільтруючий ритм працює за принципом кон'юнкції – сигнал на виході буде присутнім, лише якщо присутні сигнали на обох входах:

Ноти 1-ої послідовності накладаються на ноти 2-ої, там де вони звучать одночасно. Залишки нот, які не пересікаються – втрачаються (на малюнку їх перекреслено). В результаті отримуємо партію «Output».

При застосуванні ритмічного шаблону на вхід вузла потрапляє партія де всі ноти відсортовані за часом початку звучання, тому їх можна обходити послідовно, знаючи що час зростає разом з індексом поточної ноти в масиві.

Розглянемо код реалізації алгоритму:

```
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++
}
}
}
}

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

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 спочатку.

Аналіз приросту продуктивності. Для демонстрації приросту ефективності при внесенні змін, проведено експеримент.

Створено композицію на 16 тактів та 4 треки. Один трек виступає в ролі базового, решта – його дочірні.

До базового треку вручну та програмно вносили зміни наступного характеру: змінювали висоту або тривалість однієї з нот базової партії, змінювали ритмічний малюнок або формат побудови акордів.

Тестування програми проводили на комп'ютері з процесором Intel Core i7-8750H 2.21 GHz.

Результати тестування наведені в табл. 1.

Таблиця 1

Результати тестування

№	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

Середнє тривалість внесення змін при ручному редагуванні становила 27,7 с. Тоді як середня тривалість повторної генерації всіх чотирьох треків при внесенні змін програмою, становила близько 0,003 с (3 мс). Час на ручну заміну умови в програмі у всіх експериментах не перевищував 15 с.

Повторний експеримент з композицією на 10 треків та більш складними умовами дав наступні результати:

- на ручне редагування витрачено 2 хв 3 с;
- на повторну генерацію – 0,003 с.

Висновки. Запропоновано підхід до редагування музичного твору, який базується на роботі з концептуальними закономірностями, в доповнення до роботи на рівні конкретних нот. Створено програму, яка реалізує цей підхід. Програма показала себе ефективною в тестах при внесенні концептуальних змін. Автоматичне внесення змін програмою виконувалося значно швидше ніж внесення тих самих змін при ручному редагуванні. Остаточне звучання музичної композиції – прерогатива автору.

References

1. MIDI. URL: https://uk.wikipedia.org/wiki/MIDI.
2. Digital Audio Workstation, DAW. URL: https://en.wikipedia.org/wiki/Digital_audio_workstation.
3. Sposobin, I. V. (1996). Elementary theory of music. Moscow: Kifara. 208 p. [in Russian].
4. Wooller, R., Brown, A. R. et al. (2005). A framework for comparison of processes in algorithmic music systems. In: *Generative Arts Practice* (pp. 109–124). Sydney, Creativity and Cognition Studios Press.
5. Biles, A. (2002). GenJam in Transition: from Genetic Jammer to Generative Jammer. In: *International Conference on Generative Art*, Milan, Italy.
6. Chomsky, N. (1956). Three models for the description of language. *IRE Transations on Information Theory*, 2: 113–124.
7. Eno, B. (1996). Generative Music. URL: http://www.inmotionmagazine.com/eno1.html.
8. Kotlin language. URL: https://kotlinlang.org/
9. Android for developers. URL: https://developer.android.com/
10. Rafgarden, T. (2021). Perfect algorithm. St. Petersburg: Peter [in Russian].

Література

1. MIDI. URL: https://uk.wikipedia.org/wiki/MIDI.
2. Digital Audio Workstation – DAW. URL: https://en.wikipedia.org/wiki/Digital_audio_workstation.
3. Способін І. В. Елементарна теорія музики. Москва: Кифара, 1996. 208 с.
4. Wooller R., Brown A. R. et al. A framework for comparison of processes in algorithmic music systems. Generative Arts Practice, Sydney, Creativity and Cognition Studios Press, 2005. Pp. 109–124.
5. Biles A. GenJam in Transition: from Genetic Jammer to Generative Jammer. In: *International Conference on Generative Art*, Milan, Italy, 2002.
6. Chomsky N. Three models for the description of language. *IRE Transations on Information Theory*. 1956. No. 2. P. 113–124.
7. Eno B. Generative Music. 1996. URL: http://www.inmotionmagazine.com/eno1.html.
8. Kotlin language. URL: https://kotlinlang.org/
9. Android for developers. URL: https://developer.android.com/
10. Рафгарден Т. Совершенный алгоритм. СПб: Питер, 2021.

PYLYPENKO YURII
Candidate of Physical and Mathematical Sciences, Associate Professor, Department of Information and Computer Technologies, Kyiv National University of Technologies and Design, Ukraine
http://orcid.org/0000-0003-4093-7298
E-mail: pyl20453@gmail.com

Kutz Nazar
Student, Department of Information and Computer Technologies, Kyiv National University of Technologies and Design, Ukraine
E-mail: vitaminkuna@gmail.com

KUTZ N. M., PYLYPENKO Y. M.
Kyiv National University of Technologies and Design, Ukraine

COMPUTER PROGRAM FOR GENERATING MUSICAL COMPOSITIONS BASED ON THE AUTHOR'S MELODY

Purpose. *Finding and implementing ways to integrate a conceptual approach to create new musical compositions.*

Methodology*. A set of conceptual components of a musical composition is singled out. In designing the software system, the principle of pure SOLID architecture was used. The design of algorithms is based on the "divide and conquer" paradigm.*

Findings. *Algorithms have been developed, a program in the Kotlin programming language has been written and debugged, which allows working on editing a musical composition at a conceptual level - editing conceptual relationships, rather than specific note parameters. The volume of the program is about 16 thousand operators. Testing of the program showed that using it allows you to speed up the musical processing*

of a composition by about 10,000 times compared to manual editing. It is clear that the final decision on the success of editing is made by the author, but now he has the opportunity to listen to various versions of the processed composition, which, in fact, he does not spend time creating.

Originality. *A new approach to the use of a computer when writing musical compositions is proposed, when, based on the author's melody, a computer program generates, according to the laid down templates, somewhat modified accents of the sound of the work (for example, in terms of timbre, key, rhythm, etc.). This approach is implemented in the form of a computer program, the use of which accelerates the generation of musical works in which the author's melody sounds thousands of times.*

Practical value. *The written software product allows the composer to effectively use the new possibilities of creating musical compositions based on the author's melody.*

Keywords: *music software; music generation; software algorithms; object-oriented programming; Kotlin programming language.*