НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:
Завідувач кафедри
_____ Сергій СТІРЕНКО
(підпис)
“__” _____2023 р.

Дипломний проєкт

на здобуття ступеня бакалавра
за освітньо-професійною програмою “Інженерія програмного забезпечення комп’ютерних систем”
спеціальності 121 “Інженерія програмного забезпечення”

на тему: Система для розробки користувацького інтерфейсу

Виконав : студент 4 курсу, групи ІП-94
(шифр групи)

Рекечинський Дмитро Олександрович
(прізвище, ім’я, по батькові) (підпис)

Керівник асистент Молчанова А. А.
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант (нормоконтроль) доцент, к.т.н. Волокита А. М.
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент ас., доцент кафедри СП і СКС Сергієнко П. А.
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2023 р.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Сергій СТІРЕНКО
(підпис)
“__” _____2023 р.

ЗАВДАННЯ
на бакалаврський дипломний проєкт студента

Рекечинського Дмитра Олександровича

1. Тема проєкту *Система для розробки користувацького інтерфейсу*
керівник проєкту Молчанова Анастасія Анатоліївна, асистент
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від 31 травня 2023 року №2102-с

2. Термін здачі студентом закінченого проєкту *8 червня 2023 р.*

3. Вихідні дані до проєкту *технічна документація, теоретичні дані.*

4. Зміст розрахунково-пояснювальної записки (перелік питань, які розробляються)

Розділ 1. Системи для розробки користувацького інтерфейсу.

Розділ 2. Вибір і обґрунтування засобів реалізації системи для розробки користувацького інтерфейсу.

Розділ 3. Розробка системи для розробки користувацького інтерфейсу.

Розділ 4. Тестування розробленої системи.

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень) компоненти системи (структурна схема), діаграма класів (функціональна схема), алгоритм дій програмного забезпечення (принципова схема).

6. Консультанта проєкту, з вказівкою розділів проєкту, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Волокита А. М.		

7. Дата видачі завдання «28» вересня 2022 р.

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	*Затвердження теми проєкту*	*26.12.2022-29.12.2022*	
2.	*Вивчення та аналіз завдання*	*02.12.2022-23.04.2023*	
3.	*Розробка архітектури та загальної структури системи*	*17.04.2023-07.05.2023*	
4.	*Розробка структур окремих підсистем*	*24.04.2023-14.05.2023*	
5.	*Програмна реалізація системи*	*24.04.2023-17.05.2023*	
6.	*Оформлення пояснювальної записки*	*07.04.2023-05.05.2023*	
7.	*Захист програмного продукту*	*08.06.2023*	
8.	*Передзахист*	*13.06.2023*	
9.	*Захист*	*20.06.2023*	

Студент-дипломник _____ Дмитро РЕКЕЧИНСЬКИЙ
(підпис)

Керівник проєкту _____ Анастасія МОЛЧАНОВА
(підпис)

АНОТАЦІЯ

У даному дипломному проєкті було досліджено існуючі системи для розробки користувацького інтерфейсу, виявлено їх сильні та слабкі сторони. Результатом роботи є система для розробки користувацького інтерфейсу, яка здатна компонувати програми для кількох платформ з однаковою структурою та ідентичним виглядом. Створено концепт універсальних графічних компонент, який дозволяє створювати нові компоненти самостійно та розширювати їх можливості, використовуючи мову програмування JavaScript. Розроблено мову розмітки NFML, яка відповідає за опис структури інтерфейсу програми.

Програмний продукт було розроблено на мові програмування JavaScript.

Ключові слова: розробка користувацького інтерфейсу, графічний інтерфейс, JavaScript, ANTLR.

ANNOTATION

In this project for a Bachelor’s Degree, existing user interface development systems were explored, their strong and weak sides were revealed. As a result of work, the user interface development system was implemented, which is able to compose programs for several platforms with the same structure and identical look. The concept of universal graphical components was created, which allows to create new components independently and extend their possibilities using JavaScript programming language. The NFML markup language was developed, which is responsible for program interface structure description.

The software product was implemented in JavaScript programming language.

Keywords: user interface development, graphical interface, JavaScript, ANTLR.

справки	Формат	*Значення*	*Найменування*	Кіл. листів	№ екземпля	*Додаток*
			Документація загальна			
			Знову розроблена			
	А4	***ІАЛЦ.467200.002 ТЗ***	Система для розробки	4		
			користувацького інтерфейсу			
			Технічне завдання			
	А4	***ІАЛЦ.467200.003 ПЗ***	Система для розробки	77		
			користувацького інтерфейсу			
			Пояснювальна записка			
	А4	***ІАЛЦ.467200.004 Д1***	Система для розробки	1		
			користувацького інтерфейсу			
			Компоненти системи			
			(структурна схема)			
	А4	***ІАЛЦ.4672008.005 Д2***	Система для розробки	1		
			користувацького інтерфейсу			
			Діаграма класів			
			(функціональна схема)			
	А4	***ІАЛЦ.467200.006 Д3***	Система для розробки	1		
			користувацького інтерфейсу			
			Алгоритм дій програмного			
			забезпечення (принципова			
			схема)			
	А4	***ІАЛЦ.467200.007 Д4***	Система для розробки	19		
			користувацького інтерфейсу			
			Текст програмного коду			

					ІАЛЦ.467200.001 ОА	
Зм	***Лист***	***№ докум.***	***Підп***	***Дата***		
Розроб		Рекечинський Д. О.			*Система для розробки користувацького інтерфейсу* Опис альбому	Літ. / Аркуш 1 / Аркушів 1
Перев		Молчанова А.А.				***КПІ ім. Ігоря Сікорського, ФІОТ, ІП-94***

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Система для розробки користувацького інтерфейсу*»

Київ – 2023

ЗМІСТ

					ІАЛЦ.467200.002 ТЗ		
		№ докум.	Підпис	Дата			
Розробив		Рекечинський Д. О.			**Система для розробки користувацького інтерфейсу** Технічне завдання	Літ.	Аркуш 1 / Аркушів 4
Перевірив		Молчанова А. А.					
						КПІ ім. Ігоря Сікорського, ФІОТ, ІП-94	
Н. Контр.		Волокита А. М.					
Затвердив							

1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання поширюється на розробку системи для розробки користувацького інтерфейсу, а також на подальшу підтримку та вдосконалення розробленої системи.

Областю застосування цієї системи є проектування графічного інтерфейсу застосунків на різних платформах для особистих та комерційних цілей.

2 ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки даної системи є завдання для виконання роботи кваліфікаційно-освітнього рівня «бакалавр інженерії програмного забезпечення», який був затверджений факультетом “Інформатики та обчислювальної техніки” кафедрою обчислювальної техніки Національного технічного Університету України «Київський Політехнічний інститут ім. Ігоря Сікорського».

3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою та призначенням даної роботи є розробка системи для розробки користувацького інтерфейсу, яка дозволяє розширюватись для різних платформ за рахунок створення та адаптування універсальних графічних компонентів.

4 ДЖЕРЕЛА РОЗРОБКИ

Джерелом розробки даного дипломного проекту є офіційні документації проектів систем для розробки користувацького інтерфейсу, пов’язані публікації та статті в мережі Інтернет, науково-технічна література.

5 ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розробленого продукту

Розроблена система має виконувати такі вимоги:

- Простий та зрозумілий для користувача інтерфейс системи.
- Надати можливість користувачам генерувати програми для різних платформ на основі одного документу, написаного на мові розмітки NFML.
- Надати можливість користувачам власноруч створювати універсальні графічні компоненти, які будуть застосовуватись при генеруванні програми.
- Надати вичерпну та зрозумілу документацію для розробленого продукту.

5.2. Вимоги до програмного забезпечення

- ОС Windows, Mac чи Linux.
- Node.js версії 16.19 або вище.
- NPM версії 8.20 або вище.
- ANTLR версії 4.13 або вище.

5.3. Вимоги до апаратної частини

- ЦП не менше ніж Intel® Core (TM) i3-2100T.
- ROM не менше ніж 64 ГБ.
- RAM не менше ніж 4 ГБ.

6 ЕТАПИ РОЗРОБКИ

Назва етапів виконання	Термін виконання
Затвердження теми роботи	26.12.2022-29.12.2022
Вивчення та аналіз завдання	02.12.2022-23.04.2023
Розробка архітектури та загальної структури системи	17.04.2023-07.05.2023
Розробка структур окремих частин системи	24.04.2023-14.05.2023
Програмна реалізація системи	24.04.2023-17.05.2023
Виправлення помилок	29.04.2023-20.05.2023
Оформлення пояснювальної записки	07.04.2023-05.05.2023

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Система для розробки користувацького інтерфейсу*»

Київ – 2023

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

UI	(User Interface) Користувацький інтерфейс.
NFML	(Novice-Friendly Markup Language) Мова розмітки, яка дружня до новачків
Фреймворк	Програмне забезпечення для полегшення розробки складних систем.
Node.js	Середовище виконання програм на JavaScript.
NPM	(Node Package Manager) Менеджер пакетів Node.js.
ANTLR	(ANother Tool for Language Recognition) “Ще один інструмент для розпізнавання мови”, інструмент для генерування лексичних аналізаторів.

ВСТУП

Сучасний світ тісно пов’язаний із технологіями, які щодня використовуються на наших пристроях: мобільних телефонах, комп’ютерах тощо. Впродовж історії розвитку обчислювальної техніки спостерігаються різноманітні форми взаємодії людини та комп’ютера: перфокарти, лампи, перемикачі.

Сьогодні існує доволі зручний та інтуїтивно зрозумілий спосіб взаємодіяти з обчислювальною технікою, і це — графічний інтерфейс. Цей спосіб надання інформації про поточний стан програм значно спростив сприймання даних та керування ними. Кожного дня середньостатистична людина за допомогою графічного інтерфейсу сприймає текстові дані, переглядає фотографії, відео та ділиться враженнями з усім світом.

Вкрай рідко спостерігаються програми, що не застосовують графічний інтерфейс, особливо, коли мова йде про портативні пристрої. Існує безліч інструментів, які спрощують задачу розробки інтерфейсу, поліпшують вигляд, додають яскраві анімації.

Однак, коли мова йде про програмне забезпечення, яке повинне підтримуватись на декількох платформах (наприклад, операційні системи Windows, Linux, Android), задача розробки уніфікованого інтерфейсу стає більш складною, особливо тоді, коли різні платформи мають різну специфіку створення графічних елементів програми. Чимало коштів йде на утримання спеціалістів різних галузей, щоб підтримувати програму для певної платформи в актуальному стані. Іноді трапляється так, що через обмеження певної платформи починають відбуватись невідповідності між версіями програми на різних платформах, що створює дискомфорт для користувачів.

Вирішенням цього питання є створення програмного забезпечення, яке дає змогу розробляти графічний інтерфейс програм рівноцінно для усіх застосовуваних платформ. Крім цього, програмне забезпечення повинне підтримувати можливість створювати та модифікувати графічні компоненти, адже це дає змогу адаптуватись до змін.

Система для розробки користувацького інтерфейсу, яка розроблена в межах дипломного проєкту, несе мету покращити розробку інтерфейсу програмного забезпечення, яке буде використовуватись для декількох платформ. Це досягається за рахунок створення спрощеної мови розмітки NFML та системи, яка перетворює розмітку інтерфейсу програми на код для цільової платформи.

РОЗДІЛ 1. СИСТЕМИ ДЛЯ РОЗРОБКИ КОРИСТУВАЦЬКОГО ІНТЕРФЕЙСУ

1.1 Аналіз потреби в системах для розробки користувацького інтерфейсу

Розробка користувацького інтерфейсу — це важлива та відповідальна задача, від якої прямо залежить враження клієнтів від досвіду користування програмою. В наш час користувацький інтерфейс є основою для більшості програм, що розроблені для популярних платформ.

Процес створення користувацького інтерфейсу включає створення макету із розташуванням графічних елементів, схеми взаємодії між елементами тощо. Вигляд макету залежатиме від задач, на які програма розрахована та вибору найкращої стратегії взаємодій елементів та компонент між собою.

Також, якщо програма орієнтована на підтримку багатьох платформ, необхідно врахувати особливості з кожної цільової платформи. Розробка користувацького інтерфейсу потребує особливого підходу залежно від платформи, на якій програма буде виконуватись, адже кожна платформа має власну реалізацію графічних елементів та взаємодії між ними. Це створює складнощі, оскільки з кожною зміною вимоги щодо основної програми необхідно врахувати особливості кожної з цільових платформ.

Розглянемо існуючі системи для розробки користувацького інтерфейсу, які покликані вирішити проблему різноманітності платформ виконання програми.

1.2 Сучасні системи для розробки користувацького інтерфейсу

1.2.1 Qt

Qt — це кросплатформний фреймворк для розробки додатків з графічним інтерфейсом. Платформи, які підтримуються, включають Linux, OS X, Windows, Android, iOS та інші.

Цей проект було створено в 1990 році норвезькою компанією Trolltech. Наразі проект є власністю компанії The Qt Company [1], а розробка ведеться світовою спільнотою, що складається із зацікавлених в проекті людей. Кожен може долучитись до розробки та навіть отримати достатньо високий рівень доступу до проекту, щоб вирішувати подальші кроки розвитку Qt [2]. Код проекту є відкритим для перегляду всім людям [3].

Qt написаний на мові програмування C++. Робота фреймворку полягає у розширенні мови C++ для використання додаткових абстракцій, як сигнали, слоти тощо. Безпосередньо Qt можуть використовувати мови програмування, що сумісні з C++.

Приклад коду програми на C++, що використовує Qt:

```
#include <QApplication>
#include <QPushButton>

int main(int argc, char **argv)
{
  QApplication app (argc, argv);
  QPushButton button ("Hello, world!");
  button.setWindowTitle("Qt app example");
  button.resize(300, 300);
  button.show();
```

```
  return app.exec();
}
```

Результат програми зображений на рис. 1.1.

Рисунок 1.1 — Приклад програми із застосуванням Qt

Для інших мов було розроблено проміжну мову розмітки QML. QML — це мультипарадигмальна мова для створення програм. З її допомогою можна описати загальний графічний інтерфейс програми, а також її поведінку. JavaScript може використовуватись для опису взаємодій між графічними компоненами безпосередньо у файлі QML [4].

Приклад програми на QML, яка відображає текст “Hello, QML!” в червоному квадраті розміром 200 на 200 пікселів [5]:

```
import QtQuick 2.0

Rectangle {
    width: 200
    height: 200
    color: "red"
```

```
    Text {
        anchors.centerIn: parent
        text: "Hello, QML!"
    }
}
```

Припускаємо, що приклад програми на QML збережений у файлі example.qml. За таких умов, результат виконання команди qmlscene example.qml зображено на рис. 1.2.

Рисунок 1.2 — Приклад програми із застосуванням QML

Однак, це працює лише тоді, коли існує бібліотека для прив'язки Qt (чи QML) до певної мови програмування. Офіційно спільнотою Qt підтримується прив'язка до мови Python, усі інші розробляються окремими спільнотами [6].

Існують випадки, коли проект, покликаний надати підтримку Qt для певної мови програмування, закинуто та розробка не ведеться більше як 3 роки. Наприклад, проект Brig, який надає підтримку QML для Node.js, має останні зміни у коді 2 жовтня 2017 року [7]. Або проект qmlrs, який надає підтримку QML для мови програмування Rust, який має останні зміни у коді 12 грудня 2017 року [8].

Також, Qt має підтримку розробки для застосування у проектах з Web-технологіями, а саме WebAssembly. Обмеженням є те, що програма на виході є статичною, тобто, її не можна розширити додатковим кодом після компіляції [9].

Таким чином, Qt не є достатньо гнучкою системою в плані використання мови програмування.

1.2.2 Flutter

Flutter — це портативний набір інструментів для розробки UI, розроблений на мові програмування Dart компанією Google. Flutter підтримує декілька платформ виконання: Android, iOS, Windows, Linux та веб-браузери [10].

Серед переваг Flutter розробники підкреслюють саме високу продуктивність, об’єктно-орієнтованість та зручність для розробки.

Висока продуктивність досягається за рахунок компільованості мови, нічого не інтерпретується під час запуску. Розробники гарантують, що Flutter розроблений таким чином, щоб програма надавала 60 кадрів на секунду у постійному режимі.

В основі Flutter полягає об’єктно-орієнтований стиль програмування, оскільки, згідно з документацією від Google, галузь розробки користувацького інтерфейсу покладається на фреймворки, що застосовують цей стиль. Різка зміна стилю програмування означала би перевинахід колеса для вирішення складних задач.

Для досягнення зручності розробки Flutter підтримує гарячий перезапуск зі збереженням стану програми (після кожної перекомпіляції не доведеться з нуля відтворювати хід програми для відтворення необхідного стану). Також для

мобільних платформ Flutter має спільний інтерфейс, що прискорює розробку для платформ iOS та Android [11].

Приклад простої програми на Flutter, яка виводить на екран напис “Hello, world” [12]:

```
import 'package:flutter/material.dart';

void main() {
  runApp(
    const Center(
      child: Text(
        'Hello, world!',
        textDirection: TextDirection.ltr,
      ),
    ),
  );
}
```

Результат виконання програми зображено на рис. 1.3.

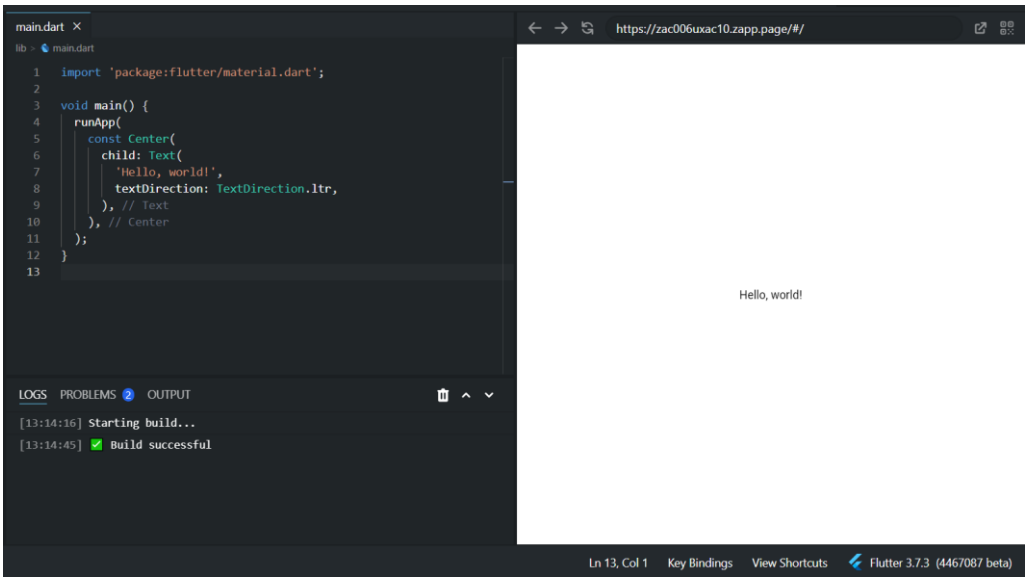


Рисунок 1.3 — Приклад програми із застосуванням Flutter

Для побудови графічних елементів Flutter не використовує елементи безпосередньо з API платформи, на якій Flutter запущено. Натомість, Flutter має свій рушій для відображення графічних елементів. З одного боку, це гарантує

те, що програма матиме вигляд, спільний для усіх платформ. З іншого боку, швидкодія на певній платформі цілком залежатиме від рішення Google [11].

Щодо здатності до підтримки мов програмування, Flutter імпортується лише у Dart. Для комунікації з іншими мовами програмування використовується канал обміну інформації. Мови програмування, для яких підтримується ця функція: Java, Kotlin, Objective-C, Swift, C++, C та JavaScript [13].

1.2.3 Electron

Electron — це фреймворк для розробки програмного забезпечення, яке застосовується на стаціонарних та портативних комп'ютерах. Вперше було випущено 15 липня 2013 року, причому не як безпосередньо фреймворк, а як допоміжний інструмент для інтегрованого середовища програмування Atom [14].

Платформи виконання, які офіційно підтримуються: Windows версії 10 та вище, macOS версії High Sierra та вище, Linux (Ubuntu 14.04 та вище) [15].

Electron працює за рахунок вмонтовування рушія веб-браузера Chromium та Node.js, міжплатформного середовища виконання коду JavaScript. Таким чином, програми, розроблені для Web із використанням HTML, CSS та JavaScript, можуть бути запущені як самостійна програма [16].

Приклад коду для створення програми на Electron [17]:

```
const { app, BrowserWindow } = require('electron');
const path = require('path');

const createWindow = () => {
    const mainWindow = new BrowserWindow({
        width: 800,
        height: 600,
        webPreferences: {
```

```
            preload: path.join(__dirname, 'preload.js')
        }
    });

    mainWindow.loadFile('index.html');
}

app.whenReady().then(() => {
    createWindow();
});

app.on('window-all-closed', () => {
    app.quit();
});
```

Результат виконання програми зображено на рис. 1.4.

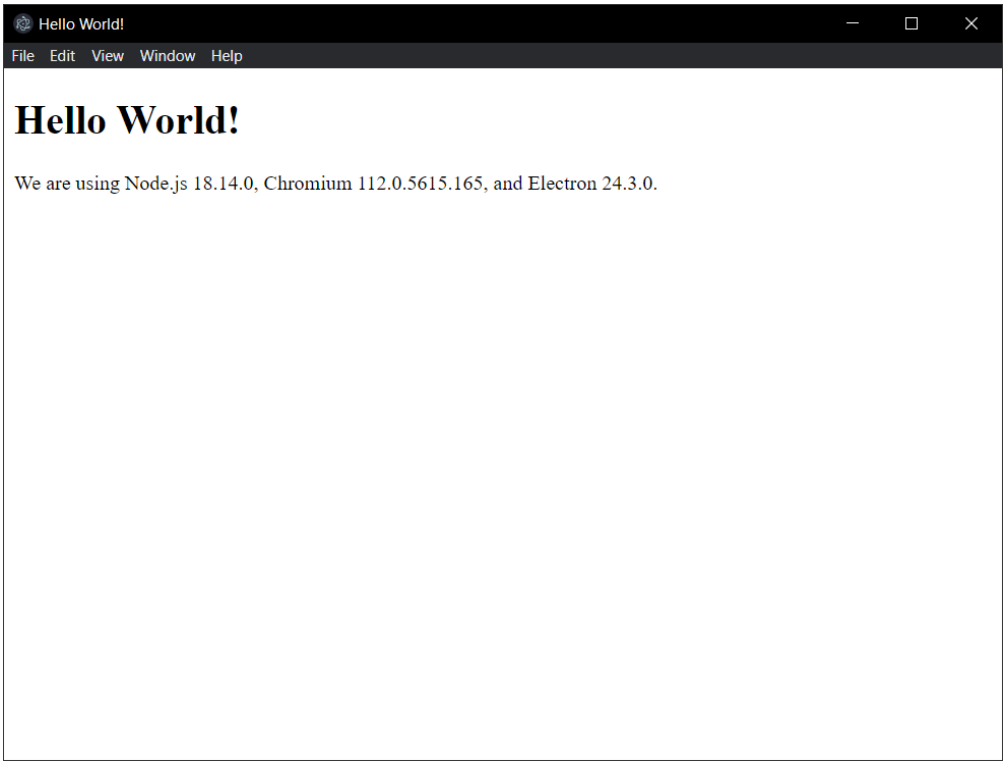


Рисунок 1.4 — Приклад програми із застосуванням Electron

Цей підхід є доволі цікавим, оскільки для веб-технологій існує чимало інструментів, які є допоміжними при розробці користувацького інтерфейсу. Однак, з деякими інструментами можуть виникнути проблеми.

Наприклад, Electron може конфліктувати з веб-фреймворками, які модифікують зарезервовані об’єкти та функції JavaScript, наприклад, require, exports тощо. Такими веб-фреймворками є jQuery, Meteor, AngularJS. Щоб вирішити цю проблему, розробники пропонують вимкнути підтримку Node.JS або змінити ідентифікатори зарезервованих об’єктів всередині головного HTML файлу [18].

1.2.4 React Native

React Native — це фреймворк для розробки користувацького інтерфейсу, який дозволяє використовувати React для розробки нативного програмного забезпечення, яке застосовується на Android та iOS. Створений компанією Facebook в 2015 році, в розробці по цей день. Спільнота React Native розробляє додаткове програмне забезпечення для підтримки платформ Windows та macOS [19].

React Native націлений на те, щоб використовувати синтаксис веб-фреймворку React, та, разом з тим, використовувати графічні елементи цільової платформи. За це відповідають компоненти, що містяться в NPM-модулі “react-native”. Під час виконання програми компоненти React Native підмінюються графічними елементами платформи. За рахунок цього програми краще інтегровані з платформою та мають високу продуктивність [20].

Приклад коду для створення програми на React Native [20]:

```
import React from 'react';
import {Text, View} from 'react-native';

const YourApp = () => {
  return (
    <View
      style={{
        flex: 1,
```

```
        justifyContent: 'center',
        alignItems: 'center',
      }}>
      <Text>Hello, world!</Text>
    </View>
  );
};

export default YourApp;
```

Результат виконання програми зображено на рис. 1.5.

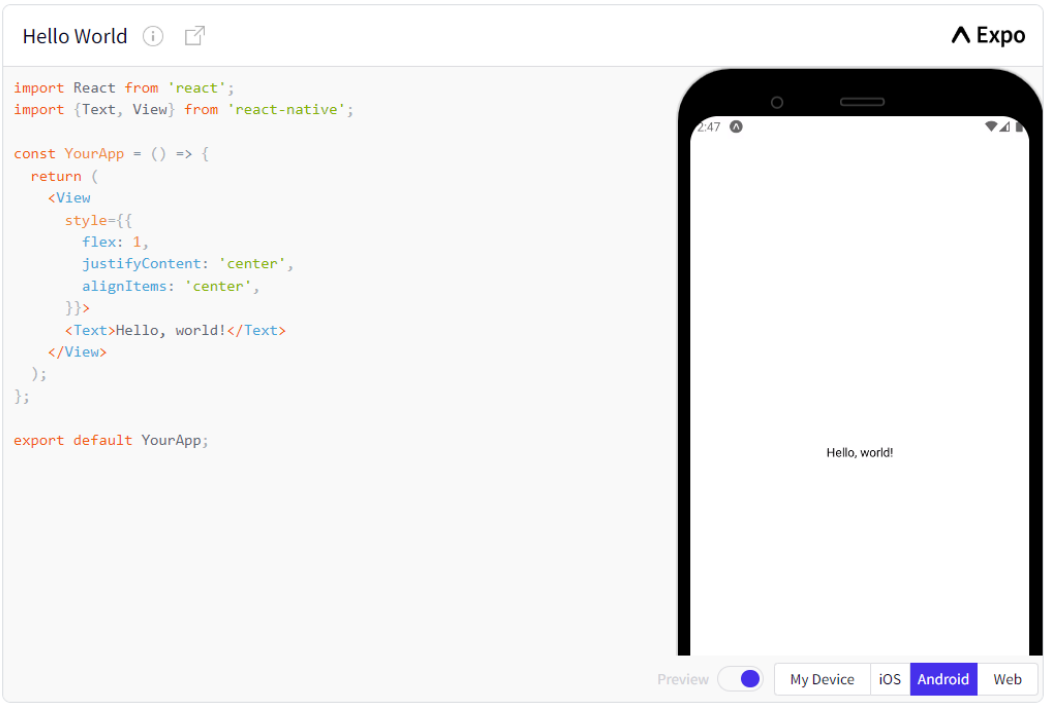


Рисунок 1.5 — Приклад програми із застосуванням React Native

Програмний код фреймворку React Native включає в себе декілька мов програмування: Java, C++, JavaScript, Objective-C, Ruby та інші. Це пов’язано з тим, що фреймворку необхідно підтримувати декілька цільових платформ [21].

Із недоліків фреймворку можна зазначити абсолютну залежність від компанії Facebook, так як проект теоретично може бути кардинально змінений або закритий. Також складно розширювати список підтримуваних платформ,

оскільки фреймворк має велику кодову базу, а також не має стабільного релізу. Остання версія станом на 17 липня 2023 року — 0.71.8 [21].

1.3 Порівняння систем для розробки користувацького інтерфейсу

В цій частині в таблиці 1.1 буде розглянуто функціональність, яка підтримується чи не підтримується для певної системи. В таблиці 1.2 буде описано загальні переваги та недоліки кожної з систем. В перелік систем для розробки користувацького інтерфейсу включено Qt, Flutter, Electron та React Native.

Таблиця 1.1 — Порівняння функціональності систем для розробки користувацького інтерфейсу

Функція	Qt	Flutter	Electron	React Native
Підтримка декількох платформ виконання	+	+	+	+
Підтримка більше, ніж однієї мови програмування	+	+/-	-	-
Використання графічних елементів цільової платформи	-	-	-	+
Відкритий код реалізації	+	+	+	+
Наявність проектів, які додають підтримку певної платформи	+	-	+	+
Стандартна функція розширення підтримуваних платформ виконання	-	-	-	-

Примітка до таблиці 1.1: підтримка Flutter більше, ніж однієї мови програмування є неповною, оскільки взаємодія з іншою мовою програмування відбувається через абстракцію у вигляді каналу.

Таблиця 1.2 — Переваги та недоліки систем для розробки користувацького інтерфейсу

Система	Переваги	Недоліки
Qt	Широка підтримка цільових платформ, мов програмування. Багата бібліотека графічних елементів.	Є складною в плані розширення існуючих платформ. Для кожної платформи необхідно розробляти свою розмітку.
Flutter	Спільний інтерфейс мобільних платформ Android та iOS. Збереження стану програми при розробці.	Контакт з іншими мовами програмування не є безпосереднім. Не використовує нативні графічні елементи платформи.
Electron	Можливість застосування веб-технологій при розробці програми.	Конфлікт із множиною веб-фреймворків. Неповна підтримка ряду платформ.
React Native	Використовує графічні елементи цільової платформи. Спільний інтерфейс для багатьох цільових платформ.	Підтримка лише однієї мови програмування. Відсутність стабільної версії продукту.

Як видно з таблиці 1.1, більшість систем розроблені з підтримкою декількох платформ виконання. Більшість з них мають обмеження щодо мови програмування, оскільки це — частина їхнього дизайну. Також більшість систем не використовують графічні елементи цільової платформи. Спільною проблемою для всіх систем для розробки графічного інтерфейсу є складність у

розширенні, як щодо платформ виконання, так і мов програмування для реалізації програми.

Таблиця 1.2 надає цілковиту картину про переваги та недоліки кожної з систем. Виходячи з фактів, наведених у цій таблиці, щоб вирішити проблему розробки користувацького інтерфейсу, необхідно створити систему з універсальним синтаксисом та розширювальною множинною підтримкою платформ виконання та мов програмування.

У розробленій системі буде комбіновано переваги існуючих систем. Наприклад, система надаватиме нативні графічні елементи (React Native), маючи спільний інтерфейс для цільових платформ (Flutter).

ВИСНОВОК ДО РОЗДІЛУ 1

Розробка користувацького інтерфейсу є важливою задачею в сучасному світі. Більшість програм мають графічний інтерфейс, який безпосередньо впливає на взаємодію між програмою та користувачем. Ця частина програми є актуальною для більшості користувачів незалежно від роду діяльності.

Було розглянуто декілька існуючих систем для розробки користувацького інтерфейсу, наведено їх принцип роботи, переваги та недоліки. Також проведено порівняння, в результаті якого виявлено проблеми, яка жодна з систем поки що не здатна вирішити повністю, а саме можливість використовувати декілька мов програмування та при цьому динамічно розширювати кількість платформ виконання.

Зважаючи на наведені вище факти, створення системи для розробки користувацького інтерфейсу є важливим для вирішення наявних проблем. Така система може використовуватись при розробці програмного забезпечення, яке націлене на декілька платформ із можливістю розширюватись в майбутньому.

РОЗДІЛ 2. ВИБІР І ОБҐРУНТУВАННЯ ЗАСОБІВ РЕАЛІЗАЦІЇ СИСТЕМИ ДЛЯ РОЗРОБКИ КОРИСТУВАЦЬКОГО ІНТЕРФЕЙСУ

2.1 Визначення вимог до засобів реалізації

Спираючись на зібрані дані у розділі 1, складено вимоги до програмного забезпечення в межах дипломного проєкту:

- Множинна підтримка платформ виконання програми
- Проста та зручна мова розмітки для побудови інтерфейсу програмного забезпечення
- Універсальний підхід до розробки користувацького інтерфейсу, “один шаблон — всі платформи”
- Модульна система елементів для забезпечення легкого розширення підтримуваних платформ
- Мінімізувати незручності при розробці функціональності, підтримуваної лише для певних платформ

2.2 Вибір засобів реалізації

В першу чергу, для реалізації системи для розробки користувацького інтерфейсу необхідно обрати мову розмітки, за допомогою якої буде описуватись ієрархія графічних елементів програми. Існує декілька основних мов розмітки: XML, JSON та YAML.

XML не є зручним для розробки, оскільки цій мові притаманна надлишковість, що спричиняє незручності при створенні великих за розміром макетів.

JSON не є зручним форматом для створення розмітки користувацького інтерфейсу, оскільки для усіх ідентифікаторів та текстових даних необхідні подвійні лапки з обох боків. Також, в JSON не існує способу описати багаторядковий текст без літерального значення символу переносу рядка \n [22].

YAML є незручним в тому плані, що ця мова розмітки залежна від відступу, оскільки межі елементу визначаються саме відступом. Таким чином, якщо є певні порушення щодо відступу, файл не зможе бути правильно прочитаним.

Згідно з вищенаведеними факторами, було вирішено створити власну мову розмітки для опису інтерфейсу під назвою NFML, тобто, Novice-Friendly Markup Language, мова розмітки, яка дружня до новачків. Її особливості та властивості наведені у підрозділі 2.6.

Для парсингу цієї мови обрано генератор парсеру ANTLR, який дозволяє з легкістю створити синтаксис мови та обробляти її.

Враховуючи вимогу підтримки багатьох платформ виконання, необхідно, щоб система для розробки користувацького інтерфейсу могла запускатись на декількох платформах, які використовуються для розробки програмного забезпечення. Відповідно, цьому повинна сприяти мова програмування, за допомогою якої буде реалізовано цей проект.

Також, щоб мінімізувати незручності при розробці функціональності, підтримуваної лише для певних платформ, мова програмування повинна мати слабку типізацію, яка допоможе легко розширювати елементи інтерфейсу в майбутньому.

І не менш важливим фактором для мови програмування є підтримка ANTLR [23], оскільки без цього компоненту буде складно розробити цілісну систему.

Цим вимогам цілком відповідає мова програмування JavaScript, сучасна популярна мова програмування зі слабкою динамічною типізацією та багатою підтримкою різноманітних бібліотек. За запуск JavaScript безпосередньо на комп'ютері відповідатиме Node.js, який в комплекті має менеджер пакетів NPM.

Таким чином, для розробки системи було обрано такі засоби реалізації:

- Мова програмування JavaScript
- Середовище виконання Node.js
- Генератор парсеру ANTLR
- Мова розмітки NFML, яка створена в процесі розробки

2.3 Мова програмування JavaScript

Згідно з документацією MDN Web Docs, JavaScript — це легка інтерпретована мова програмування, яка підтримує декілька парадигм програмування, зокрема імперативну, декларативну та об'єктно-орієнтовану (ООП) [24]. В цій роботі основна увага приділяється саме об'єктно-орієнтованій парадигмі, тому що вона допоможе скласти ієрархію елементів програмного забезпечення дипломного проєкту.

JavaScript на початку свого існування мав доволі обмежене ООП: конструктор класу описувався у вигляді функції, а методи класу присвоювались як властивості функції-конструктору. Для наочності нижче наведено зразок коду за старим стандартом JavaScript, який створює клас Circle [25]:

```
function Circle(radius) {
    this.radius = radius;
```

```
    Circle.circlesMade++;
}

Circle.draw = function draw(circle, canvas) {
    /* Canvas drawing code */
}
```

Однак, стандарт ES6 надав можливість створювати класи через безпосередню конструкцію, яка звична в ООП-орієнтованих мовах (Java, C#). Підтримуються наслідування, статичні методи та приватні поля [26]. Приклад коду, наведений вище, можна подати у такому вигляді [25]:

```
class Circle {
    constructor(radius) {
        this.radius = radius;
        Circle.circlesMade++;
    };

    static draw(circle, canvas) {
        // Canvas drawing code
    };
}
```

Також, JavaScript доволі гнучка мова в плані доступу до атрибутів об’єкту. Наприклад, якщо такого атрибуту не існує, помилки не виникне. Натомість, із виразу буде повернено значення undefined, що буквально означає “не визначене значення”. До атрибуту можна мати доступ через крапкову нотацію (object.attribute), або через квадратні дужки (object[“attribute”]). Це доволі зручно, якщо необхідно мати доступ до атрибуту, ідентифікатор якого збережений у вигляді String [27].

Не менш важливе значення має підтримка обробки текстових даних, адже основна робота програмного забезпечення полягає у перетворенні текстових

даних з однієї форми в іншу. Одним із потужних інструментів JavaScript є метод String.replace, який дозволяє замінити одну частину тексту на іншу. Підтримується пошук не лише за повним співпадінням, але й за регулярним виразом, що спрощує задачу, коли йде мова про велику кількість різноманітних замінюваних виразів [28].

2.4 Середовище виконання Node.js

Node.js — це середовище виконання коду JavaScript, яке здатне запускатись на декількох платформах. Цей інструмент є дуже популярним, особливо для проектів, у яких основна кодова база на мові програмування JavaScript. Для виконання коду використовується рушій V8, який розробляється спеціально для браузера Google Chrome.

Однією з ключових переваг Node.js є не просто підтримка асинхронних неблокуючих операцій, а безпосередня реалізація стандартних бібліотек з таким принципом. Тобто, замість витрачання циклів процесора на очікування завершення операції (наприклад, зчитування файлу з файлової системи), Node.js продовжує виконання по отриманню результату операції [29].

Node.js постачається разом з потужним інструментом, менеджером пакетів NPM. Пакет — це окремий програмний модуль, який постачається через мережу Інтернет. Пакетом може бути бібліотека, фреймворк або самостійна програма. За допомогою пакетів можна розробити програмне забезпечення швидше та ефективніше. Станом на вересень 2022 року, понад 2 мільйони пакетів було внесено до реєстру NPM. Таким чином, радше за все, пакет, в якому є потреба та необхідність, уже існує [30].

NPM допомагає керувати залежностями. Це працює за рахунок створення файлу package.json, в якому міститься інформація про назву проекту, його залежності та навіть скрипти. Наприклад, існує велика за обсягом команда, яка

відповідає за компіляцію проекту. Її можна зберегти у файлі package.json як команду з назвою compile та запускати кожен раз за допомогою команди npm run compile.

Важливим фактором є те, що у реєстрі NPM наявний пакет, який відповідає за зв'язок генератору парсеру ANTLR та JavaScript. Це значно спростить створення власної мови розмітки та її обробку [31].

2.5 Генератор парсеру ANTLR

ANTLR (ANother Tool for Language Recognition, «ще один інструмент для розпізнавання мови») — це потужний генератор парсеру для зчитування та обробки структурованого тексту. Цей інструмент розповсюджено використовується для побудови мови або пов'язаного фреймворку. На основі попередньо розробленого словника ANTLR генерує парсер, який будує абстрактне дерево, а також генерує інтерфейс граматичного аналізатора, що дає змогу легко реагувати на розпізнавання певних лінгвістичних конструкцій. ANTLR підтримує 10 цільових мов (серед яких присутній JavaScript), і основною ціллю розробки є збереження узгодженості між ними [32].

Поточна версія ANTLR — четверта. Використовуючи термін ANTLR4, люди посилаються саме на четверту версію ANTLR, а не окремий інструмент чи продукт. Четверта версія є найбільш рекомендованою з огляду на те, що третя версія в процесі розробки стала більш заплутана. Друга версія мала нечітко сформульовану ліцензію відкритого коду, через що вона не могла бути включена як залежність в IDE Eclipse. Також, автор створив власний алгоритм парсингу під назвою «адаптивний LL(*) алгоритм». Його особливістю є те, що він виконує граматичний аналіз під час виконання програми. Таким чином, синтаксичний аналізатор в процесі виконання стає все швидшим і швидшим, що дає перевагу над статичним LL(*) алгоритмом аналізу [33].

Файл граматики ANTLR має розширення g4 (ймовірно, походить від grammar version 4). У ньому містяться назва словника, опції, правила парсера та правила лексера. Назви правил парсера починаються з маленької літери, а назви правил лексера — з великої, що дає змогу відрізняти, в якому середовищі ці правила будуть застосовані. Правила лексера підтримують повний пропуск певного правила (тобто, деякі символи не будуть включені в аналіз парсером), або перенаправлення в інший канал, що дає змогу не аналізувати ці символи, але принаймні вони будуть доступні. Більше того, підтримується умовне виконання певного правила, однак вираз для перевірки залежний від цільової мови програмування. Наприклад, якщо цільова мова — JavaScript, то реалізація умовної конструкції також повинна бути реалізована на JavaScript [33].

2.6 Мова розмітки NFML

NFML (Novice-Friendly Markup Language) — це авторська декларативна мова розмітки, яка відповідає за опис користувацького інтерфейсу програми. Її особливістю є простота в користуванні невелика кількість лінгвістичних конструкцій та легкість при читанні.

Існує всього чотири типи даних в цій мові:

- Об’єкт
- String (текстовий тип даних)
- Список
- Масив

Об’єкт — це основний тип даних. На головному рівні (глобальна область видимості), файл NFML може містити лише об’єкт, причому лише один. Він є екземпляром заздалегідь визначеного класу, що відповідає за компонент інтерфейсу. Всередині об’єкт містить атрибути у вигляді пар ключ-значення.

Приклад об’єкту:

```
button {
    label: Click me!
}
```

String — текстовий тип даних, який має два види представлення: однорядковий та багаторядковий. Однорядковий тип String обмежується двокрапкою після ідентифікатора ключа та переносом на новий рядок, причому символи табуляції та пробілу не враховуються до початку будь-якого символу. Багаторядковий тип String обмежується символами ---, як в прикладі:

```
text: ---
    Так! Я буду крізь сльози сміятись,
    Серед лиха співати пісні,
    Без надії таки сподіватись,
    Буду жити! Геть думи сумні!
---
```

Символи пробілу та табуляції не враховуються до початку рядка. Однак, якщо необхідно їх включити до тексту як початок, необхідно на початок додати символ оберненої косої лінії \. При цьому, сам символ не буде включений у текст.

Список — це перелік елементів типу однорядкового String. Зазвичай застосовуються для переліку однозначних опцій, наприклад, тема оформлення програми, варіант алгоритму тощо. Має такий вигляд:

```
theme: [
    Blue
    Yellow
    Light
]
```

Масив — це перелік об’єктів. Зазвичай застосовується при описі дочірніх елементів, таким чином, можливо контролювати ієрархію графічних елементів. Має такий вигляд:

```
children: [[
    label: {
        text: Age
    }
    input {
        type: text
    }
]]
```

NFML також підтримує коментарі. Коментар — це фрагмент коду, який не буде включено в остаточний результат. Рядок коментаря починається зі знаку # та закінчується за переносі на новий рядок. Перед коментарем допустимо ставити знаки пробілу та табуляції. Приклад коментаря:

```
button {
    # Comment example
    label: Click me!
}
```

Усі атрибути (пари ключ-значення) розділені між собою переносом на новий рядок, що допомагає не нагромаджувати їх у одному рядку. Таким чином, це допомагає підтримувати легкість читання та розробки.

Крім цього, у NFML допускаються порожні об’єкт, список та масив, однак, є обмеження. Знаки, для відкривання та закривання повинні бути на різних рядках, інакше, це буде зараховано як String. Також, допускаються порожні рядки та коментарі всередині порожнього об'єкту, списку чи масиву. В іншому разі, це би ускладнило розробку документу в рази, оскільки чернетковий варіант документу NFML був би помилковий з точки зору системи.

ВИСНОВОК ДО РОЗДІЛУ 2

У цьому розділі було обрано технології та інструменти для реалізації системи для розробки користувацького інтерфейсу. Також, було обґрунтовано їх доцільність, описано їх переваги та функціональність.

Як мову програмування для реалізації дипломного проєкту, було обрано JavaScript, так як вона себе добре рекомендує з огляду на поставлені вимоги. Разом з тим, як середовище виконання коду, було обрано Node.js, який є найкращим для JavaScript.

Як мова розмітки інтерфейсу, було обрано авторську мову NFML. Було описано її функціональність, особливості та можливості. Було обґрунтовано доцільність створення нової мови замість використання існуючої мови розмітки.

Також, для полегшення розробки, було обрано генератор парсеру ANTLR, який є ефективним засобом для створення лексичного аналізатору на основі наведеного словника.

Разом усі ці інструменти створюють базу для системи розробки користувацького інтерфейсу. Вимоги щодо інструментів, які наведені на початку розділу 2, вони цілком задовольняють. Таким чином, завдання щодо вибору технологій для реалізації, яке було поставлене, успішно виконано.

РОЗДІЛ 3. РОЗРОБКА СИСТЕМИ ДЛЯ РОЗРОБКИ КОРИСТУВАЦЬКОГО ІНТЕРФЕЙСУ

3.1 Реалізація мови розмітки NFML у сфері застосування ANTLR

ANTLR — це потужний інструмент для створення парсера на основі граматичного словника. Однак, для того, щоб парсер функціонував в точності так, як очікується, необхідно створити точні та однозначні граматичні правила для кожного з аспектів мови та типів даних.

Як відомо, словник створюється у файлі з розширенням g4. Таким чином, для словника створено файл nfml.g4 та початковим виразом, який визначає ім’я словника (воно повинне співпадати з ім’ям файлу):

```
grammar nfml;
```

Тепер визначимо фундаментальні граматичні правила мови. Файл може містити в собі лише один об’єкт. Усі атрибути об’єкта — це набори ключ-значення, які розділені двокрапкою. Клас об’єкта та ключ атрибуту є ідентифікаторами, які можуть складатись лише з латинських літер нижнього регістру, розділених не більше ніж одним знаком дефісу. Значенням може бути об’єкт, текст (String), список та масив. Ці правила наведені у файлі g4 таким чином:

```
nfml: object EOF;

pair: identifier COLON value NEWLINE+;

identifier: ID_LETTER+? (ID_SEPARATOR ID_LETTER+?)*;
```

```
value: object | string | multiline_string | list | array;
```

Цей формат запису дещо нагадує існуючий формат опису граматичних особливостей мови BNF (Backus-Naur form, форма Бекуса-Наура). Він є зручним для опису, оскільки він включає в себе умовні конструкції та регулярні вирази, що значно спрощує процес створення граматичних словників.

Після того, як було визначено основні правила NFML, необхідно визначити основні типи даних. З визначенням правил для списку та масиву проблем не виникло, оскільки ці значення обмежені знаками для початку та кінця, а також внутрішніми елементами всередині них:

```
list: LIST_OPEN NEWLINE+ (string NEWLINE+)* LIST_CLOSE;

array: ARRAY_OPEN NEWLINE+ (object NEWLINE+)* ARRAY_CLOSE;
```

Однак, текстовий тип даних String може набувати двох форм: однорядковий та багаторядковий. Визначити одне й те саме правило для цих двох форм є складною задачею, оскільки для них не лише різні правила, але й обробка після парсингу відбувається по-різному. Згідно з обставинами, які склалися при спробі створення правила для цих двох форм, було вирішено побудувати два окремі правила для одного й того ж типу даних.

Однорядкова форма обмежується лише знаком переносу на новий рядок, тому правило для однорядкової форми виглядає таким чином:

```
string: (~NEWLINE)*;
```

Багаторядкова форма тексту обмежується знаками, а також допускає порожні рядки, які потім не будуть включені в результат. Для цього можна перевикористати існуюче правило string таким чином:

```
multiline_string: MULTILINE_STRING_DELIMETER NEWLINE+ (string NEWLINE)+? MULTILINE_STRING_DELIMETER;
```

Причому застосовується оператор нежадібного підбору шаблону +? для того, щоб лексична одиниця MULTILINE_STRING_DELIMETER не була зарахована в якості string.

Правило для об’єкта виглядає таким чином:

```
object: NEWLINE* identifier OBJECT_OPEN NEWLINE+ pair* OBJECT_CLOSE NEWLINE*;
```

На початку та в кінці допускаються символи переносу на новий рядок, інакше файл буде вважатись недійсним. Така поведінка була б небажаною для мови розмітки NFML.

Всі лексичні правила, які були застосовані у наведених вище правилах, мають такі значення:

```
ID_LETTER: [a-z];

ID_SEPARATOR: '-';

COLON: ':';

MULTILINE_STRING_DELIMETER: '---';

OBJECT_OPEN: '{';

OBJECT_CLOSE: '}';

LIST_OPEN: '[';

LIST_CLOSE: ']';

ARRAY_OPEN: '[[';

ARRAY_CLOSE: ']]';

NEWLINE: ('\r'? '\n');
```

Однак, у граматичних правилах не було жодної згадки відступи на початку рядка, хоча вони використовуються при форматуванні документу NFML. Виникає логічне запитання: яким чином обробляються відступи на початку рядка?

Для початку, допускаємо, що для відступів на початку рядка допускаються символи пробілу та табуляції. Тоді, можна перевести ці символи у прихований канал передачі символів:

```
WHITESPACE: [ \t]+ -> channel(HIDDEN);
```

Ці символи не будуть оброблятись, однак, вони не зникають із обробленого синтаксичного дерева. Таким чином, якщо у String будуть пробіли, вони можуть бути відновлені у початковий текст.

Останнім аспектом мови є коментар. Правило є таким: якщо на початку рядка (включаючи відступ) є знак решітки (#), то до кінця поточного рядка усі символи ігноруються при обробці. Це лексичне правило реалізовано таким чином:

```
COMMENT: NEWLINE [ \t]* '#' ~('\r' | '\n')* -> skip;
```

На початку має бути знак переносу на новий рядок, оскільки штатними засобами неможливо відстежити початок рядка. Однак, виникає проблема, якщо коментар на початку файлу, оскільки перед ним немає такого знаку. На щастя, ANTLR підтримує умовні конструкції для правил. Тобто, правило спрацьовує лише тоді, коли умова, поставлена на початку, виконується. Виглядає це так:

```
COMMENT_AT_BEGINNING_OF_FILE: {this.column === 0}? [ \t]* '#' ~('\r' | '\n')* -> skip;
```

Тобто, якщо поточна позиція символа нульова (є початком файлу), тоді, якщо на початку рядка, не враховуючи відступ, є знак решітки, поточний рядок ігнорується.

Варто зауважити, в кінці правил COMMENT та COMMENT_AT_BEGINNING_OF_FILE замість правила NEWFILE застосоване його буквальне значення. Це є наслідком того, що в лексичному правилі неможливо перевикористати інше правило, якщо його необхідно заперечити.

Також, якщо символ не згаданий у словнику, він не буде розпізнаний. Для підтримки усіх можливих символів Unicode буде застосоване це лексичне правило:

```
UNICODE_SET: [\u0000-\uFFFF];
```

Воно не застосовується в жодному правилі, але воно дає можливість використовувати всі можливі символи.

Словник для мови сформовано, тепер необхідно розробити логіку для обробки вхідних значень та перетворення у структуру цільової платформи.

3.2 Загальна архітектура системи

Система для розробки користувацького інтерфейсу створена у формі окремого застосунку, який запускається за допомогою Node.js та має залежність від NPM-пакету antlr4. Загальна структура проекту зображена на рис. 3.1.

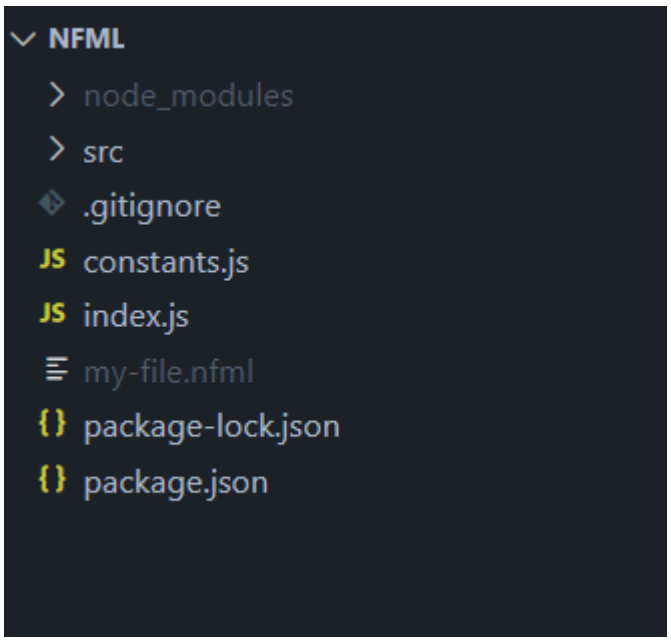


Рисунок 3.1 — Загальна структура файлів проекту

На вищому (кореневому) рівні директорії проекту розміщені файли, які відповідають за інформацію про проект та запуск системи.

Директорія node_modules містить залежності проекту, такі як бібліотеки, фреймворки тощо. Вона створена автоматично, як результат команди npm install, яка відповідає за встановлення вказаних залежностей.

Файл .gitignore створений для того, щоб вказати файли чи директорії, які не повинні бути відстежені системою контролю версій Git. У ньому вказані директорія node_modules та файли з розширенням nfml.

Файл index.js є головним файлом. Він приймає аргументи командного рядка та надає відповідний результат. До нього підв’язаний файл constants.js, який містить літеральні вирази, які використовуються в програмі. Також, до головного файлу підключені скрипти з директорії src.

Файл package.json містить загальну інформацію про проект, команди для запуску окремих утиліт та залежності, які використовуються для цього проекту.

Файл package-lock.json містить інформацію про версії встановлених залежностей. Це необхідно для того, щоб можна було точно встановити версії необхідних залежностей, оскільки інші версії можуть потенційно нести зміни, які є загрозливими для працездатності проекту.

Директорія src містить початковий код системи, який цілком відповідає за обробку введеного тексту та перетворення у інший вигляд. На рис. 3.2 зображено структуру файлів всередині цієї директорії.

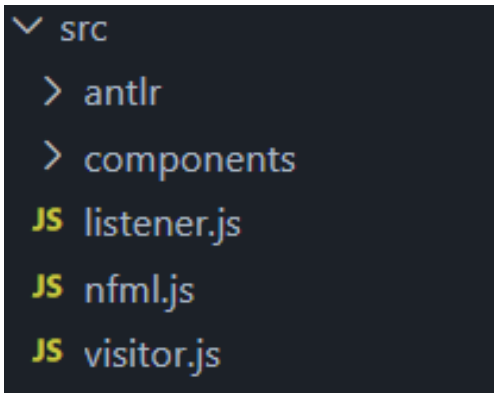


Рисунок 3.2 — Структура файлів директорії src

Директорія antlr містить словник g4 та файли, які автоматично згенеровані утилітою ANTLR. Директорія components містить графічні компоненти, які доступні в межах системи та файли для роботи з ними. Файли listener.js та visitor.js додають логіки до згенерованого парсеру nfml та вказують, як правильно десеріалізувати вхідний файл NFML. Файл nfml.js об’єднує усі модулі та відповідає за компіляцію вхідного файлу.

3.3 Файли, які пов’язані з ANTLR

Файл, який містить словник NFML, збережений з назвою nfml.g4 у директорії antlr. Однак, сам по собі словник не має здатності виконуватись та, відповідно, парсити текст, тому необхідно на його основі згенерувати всі корисні для системи файли. Це робиться за допомогою команди antlr -Dlanguage=JavaScript nfml.g4, таким чином, файли генеруються для мови програмування JavaScript. На рис. 3.3 зображено список файлів, які було згенеровано внаслідок цієї команди.

Рисунок 3.3 — Список файлів, згенерованих утилітою ANTLR

Файли nfmlLexer та nfmlParser представляють собою логіку для лексера та парсера, відповідно. Файл nfmlListener містить інтерфейс у вигляді не реалізованих методів класу NfmlListener. Це дає змогу реалізувати власну логіку для обробки вхідних лексичних одиниць, контекстів тощо. Файли з розширенням interp та tokens використовуються скриптами JavaScript для аналізу даних та перетворення у синтаксичне дерево.

Скрипти для аналізу даних на основі NFML тепер наявні, однак вони не здатні до десеріалізації, тобто, повноцінного перетворення текстових даних у програмну абстракцію. Це можливо реалізувати, створивши власний клас Listener, який є нащадком NfmlListener та доповнити його логікою для сприйняття вхідних даних. Він розміщений у файлі listener.js в директорії src (зображено на рис. 3.2). Поля класу, їх властивості та призначення описані в таблиці 3.1. Методи класу, їх аргументи, принцип роботи та кінцевий результат описані в таблиці 3.2.

Таблиця 3.1 — Поля класу Listener

Ідентифікатор поля	Властивості	Призначення
#document	Доступ: приватне Тип даних: об’єкт	Це поле містить цілісне відображення документу NFML в якості об’єкта в JavaScript.
#modeStack	Доступ: приватне Тип даних: масив	Це поле є стеком режимів поточного контексту (багаторядковий String, масив, список, об’єкт). Ці режими визначають поведінку при записі даних.

Продовження таблиці 3.1

Ідентифікатор поля	Властивості	Призначення
#identifierStack	Доступ: приватне Тип даних: масив	Це поле є стеком ідентифікаторів. Річ у тому, що ідентифікатори з’являються раніше, ніж значення (в парі ключ-значення). Тому необхідно записувати ідентифікатори в стек, із якого, в разі потреби, можна вилучити останнє записане значення для його присвоєння за ключем до об’єкта, який є відображенням документу.
#currentObjectReference	Доступ: приватне Тип даних: об’єкт	Це поле є посиланням на поточний об’єкт.

Продовження таблиці 3.1

Ідентифікатор поля	Властивості	Призначення
#objectReferenceStack	Доступ: приватне Тип даних: масив	Це поле є стеком об’єктів, але використовується як масив вказівників на об’єкти. У JavaScript об’єкти присвоюються за посиланням, а не за значенням. Однак, взаємодія з ними відбувається без абстракції “вказівник”, тобто, в контексті мови JavaScript вони є повноцінними об’єктами. Це поле використовується для повернення із внутрішнього контексту об’єкту (або масиву) до зовнішнього (батьківського).

Кінець таблиці 3.1

Ідентифікатор поля	Властивості	Призначення
#multilineStringBuffer	Доступ: приватне Тип даних: текстовий (String)	Це поле є буфером для багаторядкового виду типу даних String. По завершенню запису для одного значення та його присвоєння до документу, буфер очищується та є готовим для запису іншого значення.

Примітка. В мові програмування JavaScript, щоб позначити поле класу як приватне, необхідно на початок ідентифікатора написати знак решітки (#).

Таблиця 3.2 — Методи класу Listener

Ідентифікатор методу	Аргументи	Опис, результати
getDocument	Відсутні для поточного методу	Повертає значення поля #getDocument. Поле #getDocument є приватним, тому неможливо отримати його значення напряму.

Продовження таблиці 3.2

Ідентифікатор методу	Аргументи	Опис, результати
#getCurrentMode	Відсутні для поточного методу	Цей метод є приватним. Повертає активний режим поточного контексту.
enterNfml	ctx, тип даних: ParserRuleContext	Викликається аналізатором ANTLR на початку документу NFML. Ініціалізує об’єкт документа та наводить вказівник поточного об’єкта на нього.
enterIdentifier	ctx, тип даних: ParserRuleContext	Викликається аналізатором NFML при знаходженні будь-якого ідентифікатора. Цей метод зчитує значення ідентифікатора та розміщує його у відповідний стек.
enterMultiline_string	ctx, тип даних: ParserRuleContext	Викликається аналізатором NFML на початку багаторядкового виду типу даних String. Встановлює відповідний режим поточного контексту.

Продовження таблиці 3.2

Ідентифікатор методу	Аргументи	Опис, результати
exitMultiline_string	ctx, тип даних: ParserRuleContext	Викликається аналізатором NFML в кінці багаторядкового виду типу даних String. Записує значення у документ, очищає буфер та встановлює попередній режим поточного контексту.
enterString	ctx, тип даних: ParserRuleContext	Викликається аналізатором NFML при знаходженні однорядкового виду типу даних String, або рядка багаторядкового виду типу даних String, або елементу списку. Зчитує рядок та записує його значення у документ залежно від режиму поточного контексту.
enterList	ctx, тип даних: ParserRuleContext	Викликається аналізатором NFML на початку списку. Ініціалізує список, записує його за ідентифікатором до поточного об’єкта, переводить вказівник поточного об’єкта на список та встановлює відповідний режим поточного контексту.

Продовження таблиці 3.2

Ідентифікатор методу	Аргументи	Опис, результати
exitList	ctx, тип даних: ParserRuleContext	Викликається аналізатором NFML в кінці списку. Встановлює попередній режим поточного контексту та переводить вказівник поточного об’єкта на батьківський об’єкт.
enterArray	ctx, тип даних: ParserRuleContext	Викликається аналізатором NFML на початку масиву. Ініціалізує масив, записує його за ідентифікатором до поточного об’єкта, переводить вказівник поточного об’єкта на масив та встановлює відповідний режим поточного контексту.
exitArray	ctx, тип даних: ParserRuleContext	Викликається аналізатором NFML в кінці масиву. Встановлює попередній режим поточного контексту та переводить вказівник поточного об’єкта на батьківський об’єкт.

Кінець таблиці 3.2

Ідентифікатор методу	Аргументи	Опис, результати
enterObject	ctx, тип даних: ParserRuleContext	Викликається аналізатором NFML на початку об’єкта. Якщо це кореневий об’єкт (найвищий), новий об’єкт не створюється. В іншому випадку, ініціалізує об’єкт, записує його за ідентифікатором до поточного об’єкта чи масиву, переводить вказівник поточного об’єкта на новоутворений та встановлює відповідний режим поточного контексту.
exitObject	ctx, тип даних: ParserRuleContext	Викликається аналізатором NFML в кінці об’єкта. Встановлює попередній режим поточного контексту та переводить вказівник поточного об’єкта на батьківський об’єкт. У разі, якщо це кінець кореневого об’єкту, покажчик на поточний об’єкт не змінюється.

Примітка до таблиці 3.2: В мові програмування JavaScript, щоб позначити метод класу як приватний, необхідно на початок ідентифікатора написати знак решітки (#).

Таким чином, усі поля класу Listener є приватними. Це дає змогу обмежити доступ до критичних даних, спотворення яких може призвести до неправильної десеріалізації документу. Натомість, більшість методів класу Listener є публічними, оскільки вони повинні бути доступними для інших модулів програми, наприклад, аналізатора, згенерованого ANTLR.

По завершенню аналізу вхідного документу NFML, атрибут document містить повністю перетворені дані, які можуть бути обробленими. Таблиця 3.3 відображає вхідний тип даних NFML та його відображення у мові програмування JavaScript.

Таблиця 3.3 — Відображення типів даних NFML у мові програмування JavaScript

Тип даних NFML	Відображення у JavaScript
Об’єкт	Об’єкт JavaScript (Object) типу ключ-значення. Клас елементу записаний у атрибут _class.
Текстовий (String)	Тип даних String. Рядки в багаторядковому форматі розділені знаком переносу на новий рядок, таким чином, щоб зберегти їх як одне цілісне значення.
Список	Масив JavaScript, який містить елементи типу String.
Масив	Масив JavaScript, який містить елементи типу Object.

Для попереднього аналізу на синтаксичні помилки, застосовується клас Visitor. Його вміст збережено у файл visitor.js в директорії src (зображено на рис. 3.2). Реалізація такого класу є стандартною для ANTLR в межах мови програмування JavaScript [34].

3.4 Концепція універсальних графічних компонент

Найскладнішою частиною вирішення проблеми створення системи для розробки користувацького інтерфейсу є створення архітектури графічних компонент, при якій елементи можуть використовуватись на будь-якій платформі з однаковим результатом. Головною проблемою є різниця в особливостях кожної платформи та семантиці опису програми, графічних елементів тощо.

Цю проблему може вирішити концепція універсальних графічних компонент. Їх особливість в тому, що у них логіка розділена залежно від платформи, але інтерфейс вхідних даних спільний, що дає змогу гарантувати однаковий результат. Також, графічні компоненти повинні підтримувати розширюваність платформ, тобто, вони повинні бути легкі у підтримуванні. Найефективніший спосіб це реалізувати — це представити компоненти у вигляді класів JavaScript, у яких визначена логіка перетворення для окремого компоненту в текстовий вигляд шляхом підміни значень атрибутів графічних компонент в шаблоні. Для того, щоб не перевантажувати вміст класу, шаблони компонент будуть розміщені в окремих файлах.

В першу чергу, необхідно визначити клас Component, який є батьківським класом для усіх компонент. Це дасть змогу визначити інтерфейс, спільний для

компонент та скоротити кількість коду за рахунок перевикористання логіки. На таблиці 3.4 перечислені методи класу та їх цільове призначення.

Таблиця 3.4 — Методи батьківського класу Component

Ідентифікатор методу	Аргументи	Опис, результати
constructor	data, тип даних: Object Аргумент data містить усі дані та атрибути, зібрані з документу NFML. targetPlatform, тип даних: String	Конструктор класу Component. Якщо targetPlatform має невідповідне значення (такої платформи не існує), система надасть інформацію про помилку.
traverseChildren	componentMap, тип даних: Object Аргумент componentMap містить прив'язку назв компонент до їх реалізації.	Цей метод відповідає за ініціалізацію компонент, які є нащадками поточного компоненту. Якщо нащадків у поточного компоненту немає, тоді виконання коду цього методу ігнорується.

Продовження таблиці 3.4

Ідентифікатор методу	Аргументи	Опис, результати
prepare	Відсутні для поточного методу	Цей метод відповідає за перетворення окремого компоненту у текстове відображення, яке є частиною скомпонованої графічної програми. В контексті реалізації класу Component цей метод є інтерфейсом, тобто, кожен нащадок класу Component повинен його реалізувати. Також, цей метод є асинхронним (неблокуючим), що дає змогу прискорити процес перетворення компонент.
readTemplateFile	absolutePath, тип даних: String folderName, тип даних: String filename, тип даних: String	Цей метод відповідає за зчитування файлу за шляхом, який компонується зі значень аргументів absolutePath, folderName та filename у послідовному порядку. Зчитаний файл буде використано компонентою як шаблон.

Кінець таблиці 3.4

Ідентифікатор методу	Аргументи	Опис, результати
retrieveIdentificator	identificatorLength, тип даних: Number, значення за замовчуванням: 8	Цей метод відповідає за отримання ідентифікатора компоненту. Якщо ідентифікатор компоненту не був вказаний явно за допомогою атрибуту id, він автоматично генерується із набору латинських літер нижнього регістру. Згенерований ідентифікатор матиме довжину, рівну значенню аргументу identificatorLength.

Наступним кроком реалізації системи є визначення компонент, які необхідно створити та платформ для перевірки концепції цієї системи.

Для прикладу було взято такі платформи запуску програми:

- HTML (платформа Web)
- Java (графічна бібліотека JavaFX)

Щодо графічних компонент, було обрано такі:

- Document (головний компонент)
- Label

- Button
- Input
- Link

Усі ці компоненти будуть успадковані від класу Component та використані в межах системи для розробки користувацького інтерфейсу. Головним елементом є Document, який вміщатиме в себе всі інші елементи. Таблиця 3.5 відображає атрибути елементів та їх відображення в цільових платформах.

Таблиця 3.5 — Відображення графічних елементів у цільових платформах

Елемент, його атрибути	Відображення у Web	Відображення у Java
Document title, тип даних: String children, тип даних: масив	Документ HTML з назвою title, який містить елементи, які описані у масиві children	Програма з класом Example, яка використовує бібліотеку JavaFX. Вікно має назву title та містить елементи всередині children.
Label title, тип даних: String id, тип даних: String	Елемент <p>, вмістом якого є значення атрибуту title.	Екземпляр класу javafx.scene.control.Label, вмістом якого є значення атрибуту title. Ідентифікатор змінної дорівнює id.

Кінець таблиці 3.5

Елемент, його атрибути	Відображення у Web	Відображення у Java
Button title, тип даних: String id, тип даних: String	Елемент <button>, вмістом якого є значення атрибуту title.	Екземпляр класу javafx.scene.control.Button, вмістом якого є значення атрибуту title. Ідентифікатор змінної дорівнює id.
Input title, тип даних: String id, тип даних: String	Елемент <input>, вмістом якого є значення атрибуту title.	Екземпляр класу TextField (пакет javafx.scene.control), вмістом якого є значення атрибуту title. Ідентифікатор змінної дорівнює id.
Link title, тип даних: String id, тип даних: String url, тип даних: String	Елемент <a>, який посилається на url (атрибут href дорівнює значенню атрибуту url), вмістом якого є значення атрибуту title.	Екземпляр класу Hyperlink (пакет javafx.scene.control), вмістом якого є значення атрибуту title. Ідентифікатор змінної дорівнює id. При натисканні відкриває посилання url у браузері.

Таким чином, за допомогою цих елементів можна скласти базову програму, яка відображатиме обрані для прикладу графічні елементи. Їх реалізація має спільну для всіх компонент структуру: всередині директорії src/components створюється директорія, яка має назву компонента. Всередині неї створюється скрипт index.js, у якому є клас, який є нащадком Component. Крім цього скрипту, створюються папки з шаблонами для підміни значень. Приклад для компоненту Button зображено на рисунку 3.4.

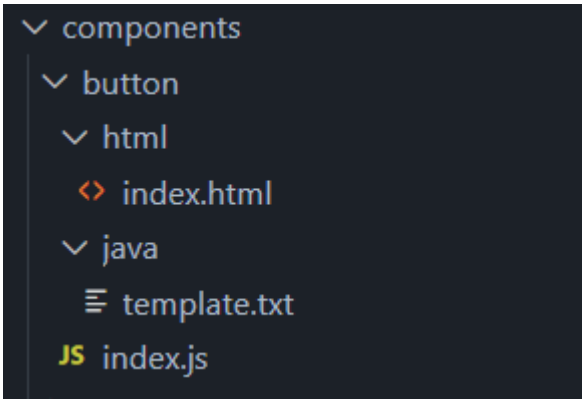


Рисунок 3.4 — Ієрархія файлів для компоненту Button

У файлі index.js реалізовано клас Button, який успадковується від Component. Таким чином, достатньо реалізувати лише два методи: конструктор класу та prepare, який відповідає за перетворення даних у текстовий вигляд компоненту відповідно до цільової платформи. Папки html та java містять шаблони для підміни значень. Метод prepare завантажує необхідний шаблон та підмінює підготовлені частини тексту на відповідні значення. Так само це реалізовано для елементів Label, Input та Link.

Щодо компоненту Document, то його реалізація особлива тим, що він повинен містити не лише інформацію про себе, але й інформацію про нащадків та загальну структуру програми відповідно до цільової платформи. Метод

підміни і в цьому випадку дозволяє реалізувати цей компонент, оскільки ми можемо зчитати вміст компонент-нащадків та розмістити його у шаблоні.

Такий підхід дозволяє за потреби створювати нові компоненти, а також розширювати список підтримуваних платформ.

3.5 Процес перетворення документу NFML у цільовий документ

Логіка перетворення тексту в об’єкт JavaScript та ієрархія графічних компонент були створені. Необхідно створити логіку, яка їх об’єднує, а саме перетворює об’єкт у відповідні екземпляри класів компонент. Алгоритм, який реалізує цю дію, такий:

1. Зчитати атрибут _class.
2. Якщо існує компонент, який відповідає значенню _class, ініціалізувати компонент на основі даних, наявних у об’єкті. В іншому разі, видати помилку.
3. Якщо в об’єкті наявний атрибут children і він не є порожнім, повторити дії 1-3 для всіх вкладених елементів. В іншому разі, ігнорувати крок 3.

В класі Component наявний метод traverseChildren, який відповідає за перетворення вкладених компонент. Скрипт traverseDocument, який розміщений у директорії src/components (рис. 3.5), відповідає за ініціалізацію головного компоненту за запуск процесу перетворення компонент-нащадків. Для інформації про наявні компоненти використовується файл ComponentMap.js.

Файл nfml.js, який розміщений у директорії src, містить функцію compile, яка приймає аргументи input (вміст вхідного документу NFML) та targetPlatform (цільова платформа). Функція compile обробляє вхідний вміст, використовуючи засоби ANTLR, класи Visitor та Listener та перетворює у готову програму за допомогою виклику функції traverseDocument. Залежно від того, чи успішно виконався процес обробки вхідного вмісту, може бути повернений або вміст вихідного файлу, або статус про помилку.

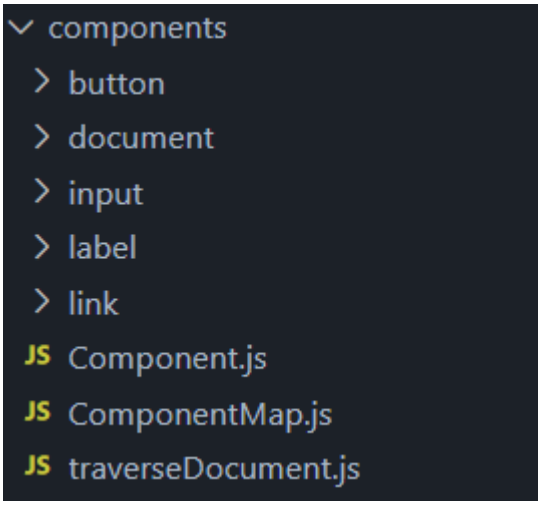


Рисунок 3.5 — Ієрархія файлів директорії src/components

Для взаємодії з користувачем було реалізовано головний модуль — index.js, який розміщений на кореневому рівні проекту. Метод взаємодії - інтерфейс командного рядка. На таблиці 3.6 зображені аргументи модулю та їх значення.

Таблиця 3.6 — Аргументи головного модулю

Назва аргументу	Скорочений варіант	Значення в програмі
--file	-f	Шлях до файлу, в якому розміщений документ NFML. Обов’зяковий параметр.
--platform	-p	Цільова платформа. Якщо не вказано, то, за умовчуванням, набуває значення HTML.
--output	-o	Файл, в якому буде міститись код програми для цільової платформи. Якщо не вказано, код буде виведено у інтерфейс командного рядка.
--help	-h	Надати короткий довідник з параметрів головного файлу. Якщо вказано, усі інші параметри ігноруються, а процес перетворення NFML документу не відбувається.

На таблиці 3.7 наведені приклади команд при використанні системи та їх очікувані результати.

Таблиця 3.7 — Приклади команд при використанні системи

Приклад команди	Очікуваний результат
node index.js	На екран виведено короткий довідник з параметрів, оскільки шлях до вхідного файлу не було надано.
node index.js --file my-file.nfml	Вхідний файл оброблено, результат виведено на екран. Цільова платформа: HTML.
node index.js --file my-file.nfml --platform Java	Вхідний файл оброблено, результат виведено на екран. Цільова платформа: Java.
node index.js --file my-file.nfml --output test.html	Вхідний файл оброблено, результат виведено у файл test.html. Цільова платформа: HTML.
node index.js --file my-file.nfml --output Example.java --platform Java	Вхідний файл оброблено, результат виведено у файл Example.java. Цільова платформа: Java.
node index.js --help	На екран виведено короткий довідник з параметрів.
node index.js --file my-file.nfml --output Example.java --platform Java --help	На екран виведено короткий довідник з параметрів. Усі інші аргументи було проігноровано.

Таким чином, створено систему із інтерфейсом командного рядка. Це дозволяє керувати процесом в будь-якому середовищі, яке підтримує такий інтерфейс. Приклад запуску команди зображено на рисунку 3.6.

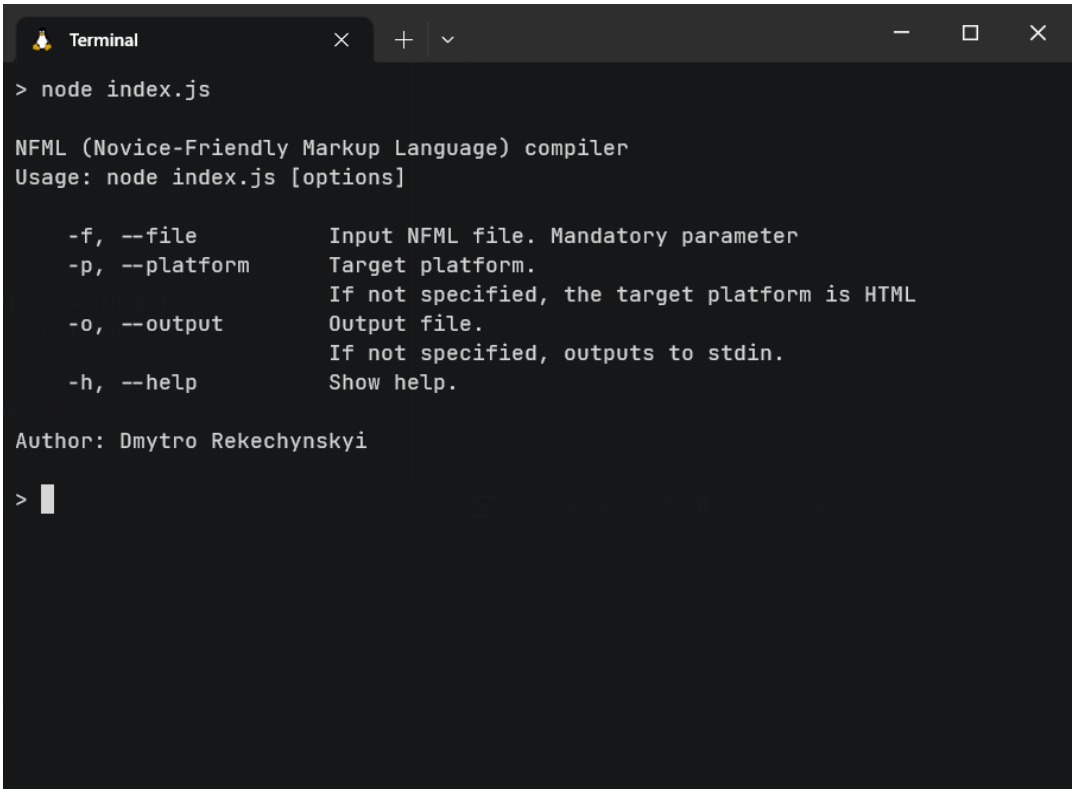


Рисунок 3.6 — Приклад запуску системи

Система взаємодіє з користувачем, надаючи інформацію про статус виконаної команди. Також, в разі необхідності, вона показує короткий довідник з параметрів, їх опис та вплив на результат виконання. Таким чином, система для розробки користувацького інтерфейсу є простою та зручною в користуванні.

ВИСНОВОК ДО РОЗДІЛУ 3

У цьому розділі було описано загальну структуру проєкту, підхід до архітектури реалізації та зв’язки між окремими компонентами. Систему розроблено на основі інструментів, наведених у розділі 2.

Для аналізу вхідного тексту NFML було розроблено граматичний словник ANTLR та граматичний аналізатор Listener на базі інтерфейсу згенерованого класу NfmlListener.

Щоб задовольнити мету перетворення отриманих даних у повноцінний код цільової платформи, було створено концепцію універсальних графічних компонент, які мають окрему логіку та спільний інтерфейс. Логіка перетворення компонент описується мовою JavaScript, що дозволяє розширювати список підтримуваних компонент та платформ за необхідності.

Головний файл системи має інтерфейс командного рядка та потребує ввід параметрів для надання бажаного результату. Було описано усі можливі сценарії виконання.

Таким чином, задача створення системи для розробки користувацького інтерфейсу із задоволенням вимог, поставлених у розділах 1, 2 була успішно виконана.

РОЗДІЛ 4. ТЕСТУВАННЯ РОЗРОБЛЕНОЇ СИСТЕМИ

4.1 Загальний огляд роботи системи

Для перевірки системи на коректну роботу необхідно створити приклад документу NFML. Його вміст зображено на рис. 4.1. Припускаємо, що файл, у якому збережено цей документ, збережено на кореневому рівні проекту, а також він має назву my-file.nfml.

```
# Приклад документу NFML

document {
    title: Дипломний проєкт

    children: [[
        label {
            title: Напис всередині документу. ウクライナに栄光あれ！
        }

        button {
            title: Кнопка
        }

        # Link element example
        link {
            title: Посилання на чудовий сайт
            url: https://google.com
        }
    ]]
}

```

Рисунок 4.1 — Вміст тестового документу NFML

За таких умов, достатньо запустити команду node index.js --file my-file.nfml, щоб перевірити, чи система перетворює документ NFML у код на мові

розмітки HTML. Оскільки файл, у який буде збережено результат перетворення, не вказано, вміст HTML буде зображено в командному рядку. Команда та результат виконання системи зображено на рисунку 4.2.

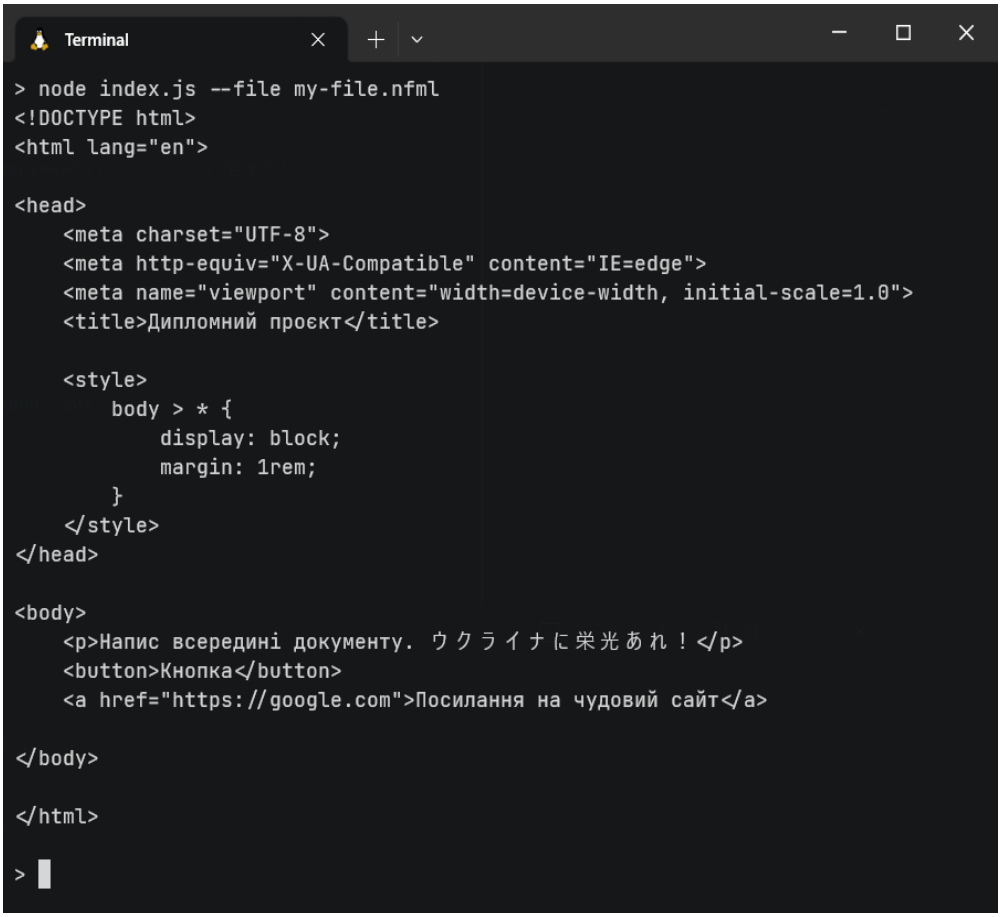


Рисунок 4.2 — Результат виконання команди node index.js --file my-file.nfml

Виконання було виконано успішно, а вміст згенерованого коду на мові HTML відповідає тому, що вказано у документі NFML. Тепер, спробуємо зберегти цей результат у файл. Запустимо команду node index.js --file my-file.nfml --output test.html. При цьому статус про успішний запис у файл має бути висвітлено у командному рядку (рис. 4.3).

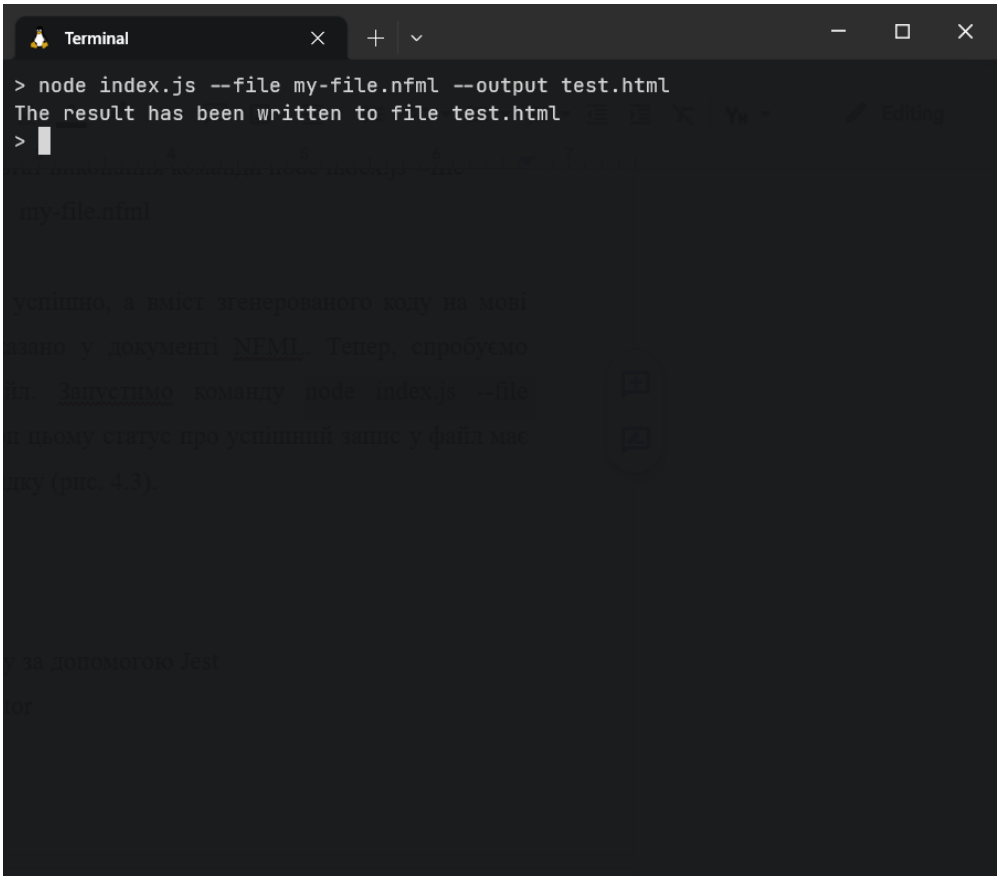


Рисунок 4.3 — Результат виконання команди node index.js --file my-file.nfml --output test.html

Після успішного запису в файл необхідно відкрити test.html у браузері для того, щоб переконатись, що усі елементи відображаються коректно (рис. 4.4).

Згідно зі структурою елементів у документі NFML (див. рис. 3.1), сайт повинен мати назву “Дипломний проєкт”, містити напис, кнопку та посилання. В такому разі, система була здатна коректно спроектувати елементи документу NFML на файл HTML.

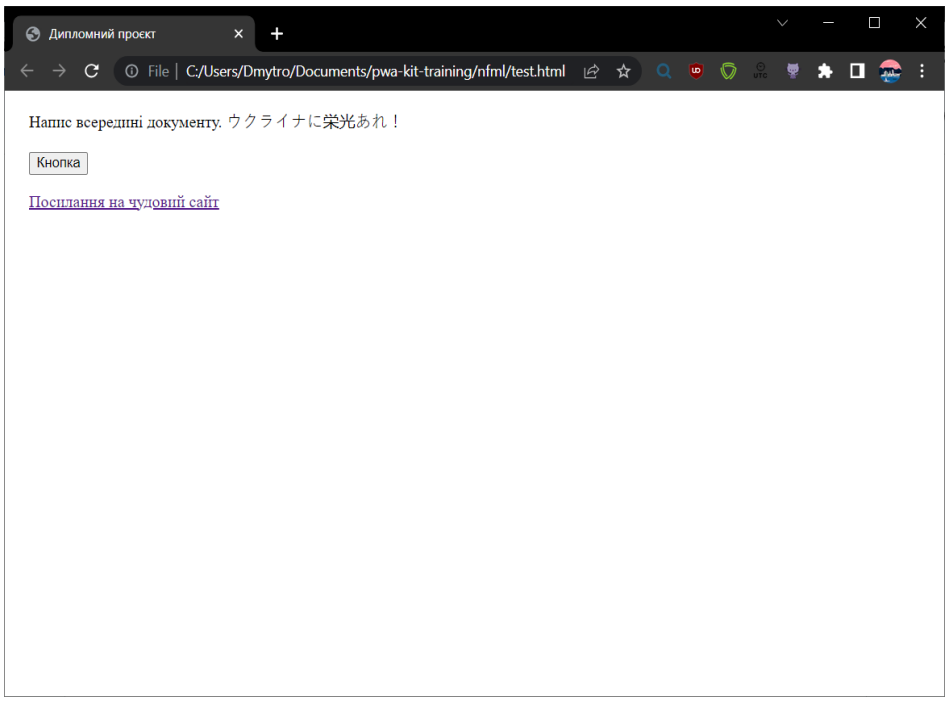


Рисунок 4.4 — Результат перетворення документу NFML для платформи HTML (Web)

Також, необхідно запустити команду node index.js --file my-file.nfml --output Example.java --platform Java для того, щоб перевірити роботоздатність на платформі Java (рис. 4.5).

Рисунок 4.5 — Результат перетворення документу NFML для платформи Java

Як бачимо, програми мають однакову структуру, яка відповідає документу, ідентичний вигляд та безпомилкове виконання. Тепер слід перевірити, як реагує система на структуру документу, яка не відповідає специфікації NFML. Найпростіший спосіб — пошкодити документ за рахунок вставлення знаку решітки перед лінією коду, яка є важливою для виконання (рис. 4.6).

```
# Приклад документу NFML

document {
    title: Дипломний проєкт

    # children: [[
        label {
            title: Напис всередині документу. ウクライナに栄光あれ！
        }

        button {
            title: Кнопка
        }

        # Link element example
        link {
            title: Посилання на чудовий сайт
            url: https://google.com
        }
    ]]
}

```

Рисунок 4.6 — Пошкоджений документ NFML

Очікується, що після запуску команди node index.js --file my-file.nfml будуть повідомлення про помилку, яка виникла під час спроби прочитати та аналізувати документ NFML (рис. 4.7).

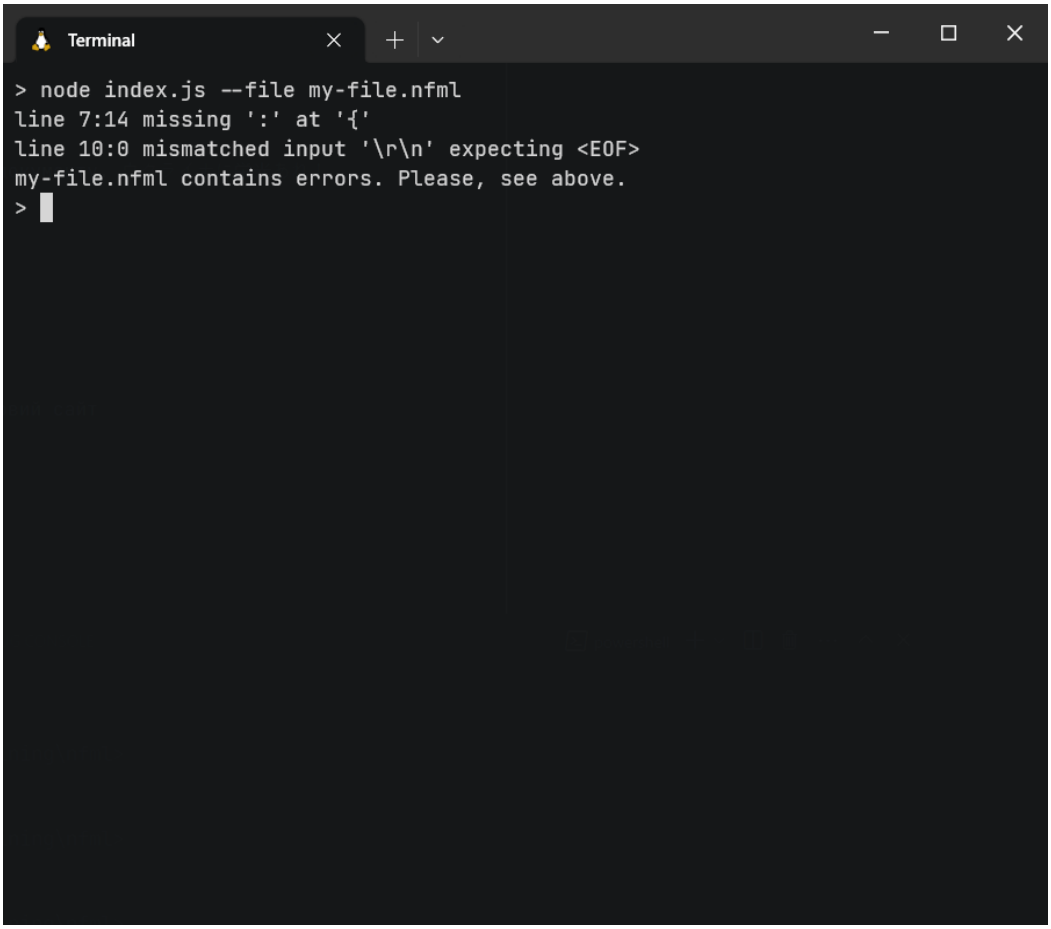


Рисунок 4.7 — Результат виконання програми при пошкодженому документі NFML

Таким чином, було перевірено систему на відповідну поведінку залежно від переданих даних. Результати співпали з очікуваннями щодо поведінки системи, що свідчить про коректність роботи розробленої системи.

4.2 Тестування ділянок коду за допомогою Jest

Для тестування ділянок коду було обрано фреймворк Jest, який позиціонується розробниками як платформа для тестування коду на JavaScript,

яка зосереджена на простоті. Із його переваг виділяються зручність в користуванні, багата документація, ізольованість станів та паралелізація тестів [35].

Також, не менш важливою функцією є можливість відстеження викликів функцій та їх аргументів. Таким чином, можна відстежити, що було передано у консоль, відстеживши функцію console.log.

Необхідно покрити критично важливі ділянки коду. Однак, є частини коду, які тестуванню не підлягають. Код, автоматично згенерований інструментом ANTLR, не залежить від нашої розробки. Також, неможливо перевірити поведінку та стан Listener залежно від аргументів, оскільки усі поля в цьому класі приватні.

Таким чином, тести для програмного коду будуть реалізовані для таких складових системи:

- Клас Visitor
- Клас Component
- Модулі системи traverseDocument.js та nfml.js
- Головний модуль системи (index.js)

4.2.1 Тестування класу Visitor

У підрозділі 3.3 було згадано, що реалізація цього класу є стандартною в межах ANTLR. Однак, з часом розробки функціонал може розширюватись, що може призвести до змін у логіці цього класу, тому важливо його відстежувати поведінку.

Тестові випадки для методу visitChildren(ctx):

- Повинен повернути undefined, коли ctx дорівнює null
- Повинен повернути порожній масив, коли ctx не має нащадків
- Повинен повернути масив дочірніх вузлів, коли ctx має нащадків без вкладених нащадків
- Повинен повернути масив дочірніх вузлів, коли ctx має нащадків з вкладеними нащадками

4.2.2 Тестування класу Component

Тестові випадки для конструктора:

- Повинен створити новий екземпляр Component, якщо аргументи — валідні
- Повинен викинути помилку, якщо цільова платформа — невалідна

Тестові випадки для методу traverseChildren:

- Повинен перетворювати валідні компоненти-нащадки
- Повинен викинути помилку, якщо компоненти-нащадки — невалідні

Тестові випадки для методу retrieveIdentificator:

- Повинен витягнути існуючий id
- Повинен згенерувати id, якщо немає існуючого

Тестові випадки для методу readTemplateFile:

- Повинен прочитати файл, якщо він існує
- Повинен викинути помилку, якщо файлу не існує

4.2.3 Тестування модулів системи traverseDocument.js та nfml.js

Тестові випадки для функції traverseDocument (файл traverseDocument.js):

- Повинен повернути контент для валідного документу
- Повинен викинути помилку, якщо документ — невалідний
- Повинен викинути помилку, якщо цільова платформа — невалідна
- Повинен повернути контент для валідного документу з множинною кількістю компонент-нащадків

Тестові випадки для методу compile (файл nfml.js):

- Повинен повернути скомпільований контент для валідного документу
- Повинен викинути помилку, якщо документ — невалідний
- Повинен викинути помилку, якщо цільова платформа — невалідна

4.2.4 Тестування головного модулю системи

Тестові випадки для головного модулю програми:

- Повинен показати короткий довідник, якщо не вказано параметр --file
- Повинен скомпілювати вхідний файл та вивести на екран
- Повинен викинути помилку, якщо надано невалідний аргумент
- Повинен показати короткий довідник, якщо надано аргумент --help

4.2.5 Результати кодового тестування

Для тестування достатньо команди jest, однак хорошою практикою є вказати цю команду як скрипт “test” у файлі package.json. Таким чином, тести запускаються через команду npm test, що є більш універсальною командою в сфері Node.js. Результати запуску тестів зображено на рисунку 4.8.

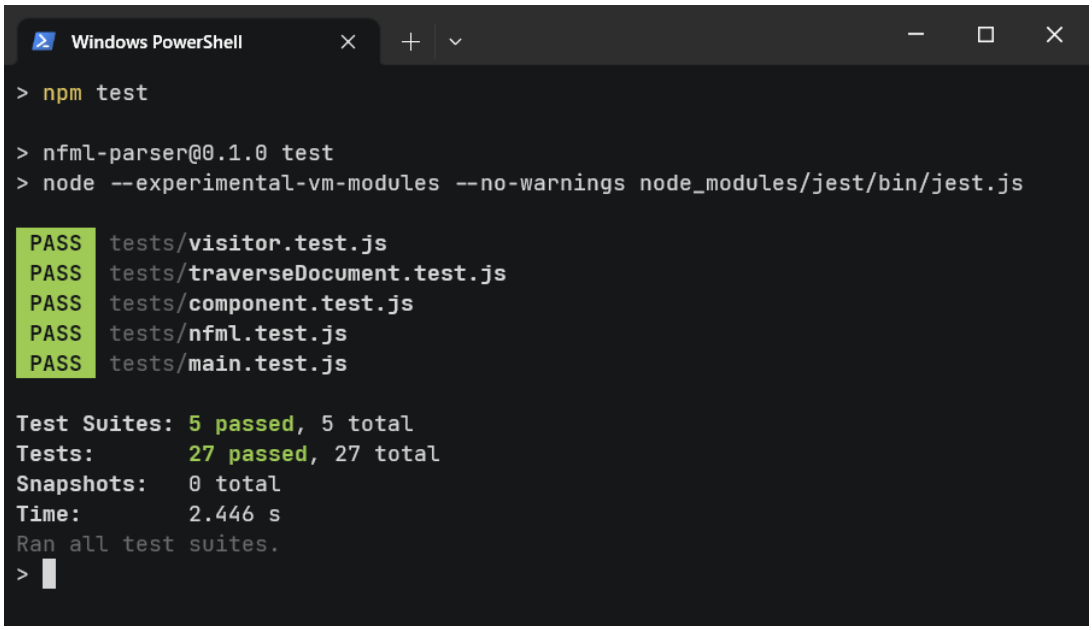


Рисунок 4.8 — Результати тестів

Згідно з результатами тестів Jest, із 5 тестових наборів (27 тестових випадків) було пройдено успішно 5 тестових наборів, що свідчить про відповідність системи щодо очікуваної поведінки.

ВИСНОВОК ДО РОЗДІЛУ 4

У цьому розділі було продемонстровано роботу системи, оглянуто її поведінку при наданні тих чи інших даних. Система здатна генерувати файли для кількох цільових платформ, надаючи однакову структуру програм та ідентичний вигляд.

Також, розроблено тести для модулів програми, враховано можливі сценарії виконання та визначено очікувану поведінку. При запуску тестів система надавала очікувані результати, що свідчить про стабільність роботи системи. Кожен компонент системи було розроблено з увагою до деталей, оскільки тести були виконані успішно навіть при сценаріях, які зазвичай не передбачені при використанні системи. Наприклад, при введенні некоректних аргументів система реагує відповідно, не виконуючи зайвих дій. При пошкодженому документі NFML система повідомляє про прогалини у файлі, вказуючи на відповідні місця.

Завдання продемонструвати роботу системи для розробки користувацького інтерфейсу та перевірити на працездатність було успішно виконано.

ВИСНОВКИ

При виконанні дипломної роботи було створено систему для розробки користувацького інтерфейсу. За мету було поставлено створити систему, яка дозволяє бути легко розширюваною та мати єдиний інтерфейс для всіх платформ. Користувач має змогу створити власний графічний компонент та адаптувати його для декількох цільових платформ.

У першому розділі було описано існуючі рішення для розробки користувацького інтерфейсу, виявлено їх переваги та недоліки. Спільною проблемою для всіх систем була неможливість використовувати декілька мов програмування та при цьому динамічно розширювати кількість платформ виконання. Ґрунтуючись на зібраних даних в першому розділі, було створено вимоги щодо створюваної системи.

У другому розділі було обрано інструменти для реалізації системи для розробки користувацького інтерфейсу. Як мову програмування було обрано JavaScript, так як вона себе добре рекомендує з огляду на поставлені вимоги. Разом з тим, як середовище виконання коду, було обрано Node.js. Як мова розмітки інтерфейсу, було обрано авторську мову NFML. Також було обрано генератор парсеру ANTLR, який є ефективним засобом для створення лексичного аналізатору на основі наведеного словника.

У третьому розділі було описано загальну структуру проєкту, підхід до архітектури реалізації та зв’язки між окремими компонентами. Для аналізу вхідного тексту NFML було розроблено граматичний словник ANTLR. Щоб задовольнити мету перетворення отриманих даних у повноцінний код цільової платформи, було створено концепцію універсальних графічних компонент. Головний файл системи має інтерфейс командного рядка та потребує ввід параметрів для надання бажаного результату.

У четвертому розділі було продемонстровано роботу системи, оглянуто її поведінку при наданні тих чи інших даних. Також, розроблено тести для модулів програми, враховано можливі сценарії виконання та визначено очікувану поведінку. При запуску тестів система надавала очікувані результати, що свідчить про стабільність роботи системи.

Задача створити систему для розробки користувацького інтерфейсу, яка несе за мету бути адаптивною до цільових платформ, було успішно виконано. Створена система вдало створює програму для абсолютно різних за семантикою платформ HTML та Java, зберігаючи при цьому ідентичний вигляд. Розроблена система вирізняється з-поміж наявних систем тим, що дає змогу легко створювати графічні компоненти програмних систем, зменшуючи витрати часу та зусиль розробників на проєктування та модернізацію цих систем.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. About QT — QT Wiki [Електронний ресурс] – Режим доступу до ресурсу: https://wiki.qt.io/About_Qt (дата звернення 29.04.2023).
2. The Qt Governance Model — Qt Wiki [Електронний ресурс] – Режим доступу до ресурсу: https://wiki.qt.io/The_Qt_Governance_Model (дата звернення 29.04.2023).
3. Організація Qt — GitHub [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/qt (дата звернення 29.04.2023).
4. Qt 6 QML / J. Thelin, J. Bocklage-Ryannel, C. Lorquet., 2022. – 728 с.
5. QML Syntax Basics | Qt QML 6.5.0 [Електронний ресурс] – Режим доступу до ресурсу: https://doc.qt.io/qt-6/qtqml-syntax-basics.html (дата звернення 29.04.2023).
6. Language Bindings — Qt Wiki [Електронний ресурс] – Режим доступу до ресурсу: https://wiki.qt.io/Language_Bindings (дата звернення 29.04.2023).
7. Репозиторій BrigJS/brig — GitHub [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/BrigJS/brig (дата звернення 29.04.2023).
8. Репозиторій flanfly/qmlrs — GitHub [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/flanfly/qmlrs (дата звернення 29.04.2023).
9. Qt WebAssembly Platform Notes | Qt 5.15 [Електронний ресурс] – Режим доступу до ресурсу: https://doc.qt.io/qt-5/qtwebassembly-platform-notes.html (дата звернення 21.05.2023).
10. Supported deployment platforms — Flutter documentation [Електронний ресурс] – Режим доступу до ресурсу: https://docs.flutter.dev/reference/supported-platforms (дата звернення 02.05.2023).

11. FAQ — Flutter documentation [Електронний ресурс] – Режим доступу до ресурсу: https://docs.flutter.dev/resources/faq (дата звернення 02.05.2023).
12. Introduction to widgets — Flutter documentation [Електронний ресурс] – Режим доступу до ресурсу: https://docs.flutter.dev/ui/widgets-intro (дата звернення 02.05.2023).
13. Flutter in Action / Eric Windmill. – Shelter Island: Manning Publications, Co., 2020. – 368 с. – (1).
14. Репозиторій electron/electron, версія v0.1.0 — GitHub [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/electron/electron/tree/v0.1.0 (дата звернення 03.05.2023).
15. Репозиторій electron/electron — GitHub [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/electron/electron (дата звернення 03.05.2023).
16. Prerequisites | Electron [Електронний ресурс] – Режим доступу до ресурсу: https://electronjs.org/docs/latest/tutorial/tutorial-prerequisites (дата звернення 03.05.2023).
17. Quick Start | Electron [Електронний ресурс] – Режим доступу до ресурсу: https://electronjs.org/docs/latest/tutorial/quick-start (дата звернення 03.05.2023).
18. Electron FAQ | Electron [Електронний ресурс] – Режим доступу до ресурсу: https://electronjs.org/docs/latest/faq (дата звернення 03.05.2023).
19. React Native · Learn once, write anywhere [Електронний ресурс] – Режим доступу до ресурсу: https://reactnative.dev/ (дата звернення 04.05.2023).
20. Core Components and Native Components · React Native [Електронний ресурс] – Режим доступу до ресурсу: https://reactnative.dev/docs/intro-react-native-components (дата звернення 04.05.2023).
21. Репозиторій facebook/react-native — GitHub [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/facebook/react-native (дата звернення 04.05.2023).

22. Are multi-line strings allowed in JSON? — Stack Overflow [Електронний ресурс] – Режим доступу до ресурсу: https://stackoverflow.com/questions/2392766/are-multi-line-strings-allowed-in-json (дата звернення 05.05.2023)
23. Runtime Libraries and Code Generation Targets, репозиторій antlr/antlr4 — GitHub [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/antlr/antlr4/blob/master/doc/targets.md (дата звернення 05.05.2023)
24. JavaScript | MDN [Електронний ресурс] – Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Web/JavaScript (дата звернення 05.05.2023).
25. ES6 In Depth: Classes – Mozilla Hacks — the Web developer blog [Електронний ресурс] – Режим доступу до ресурсу: https://hacks.mozilla.org/2015/07/es6-in-depth-classes (дата звернення 05.05.2023).
26. Classes — JavaScript | MDN [Електронний ресурс] – Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes (дата звернення 05.05.2023).
27. Property accessors — JavaScript | MDN [Електронний ресурс] – Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Property_accessors (дата звернення 05.05.2023).
28. String.prototype.replace() — JavaScript | MDN [Електронний ресурс] – Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/replace (дата звернення 05.05.2023).
29. Introduction to Node.js [Електронний ресурс] – Режим доступу до ресурсу: https://nodejs.dev/en/learn/ (дата звернення 07.05.2023).

30. An introduction to the NPM package manager [Електронний ресурс] – Режим доступу до ресурсу: https://nodejs.dev/en/learn/an-introduction-to-the-npm-package-manager/ (дата звернення 07.05.2023).
31. antlr4 [Електронний ресурс] – Режим доступу до ресурсу: https://www.npmjs.com/package/antlr4 (дата звернення 09.05.2023).
32. Репозиторій antlr/antlr4 — GitHub [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/antlr/antlr4 (дата звернення 09.05.2023).
33. The Definitive ANTLR 4 Reference / Terence Parr. – Dallas, Texas: The Pragmatic Bookshelf, 2012. – 322 с.
34. JavaScript, репозиторій antlr/antlr4 — GitHub [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/antlr/antlr4/blob/master/doc/javascript-target.md (дата звернення 19.05.2023).
35. Документація Jest [Електронний ресурс] – Режим доступу до ресурсу: https://jestjs.io/ (дата звернення 25.05.2023).

ДОДАТОК 1

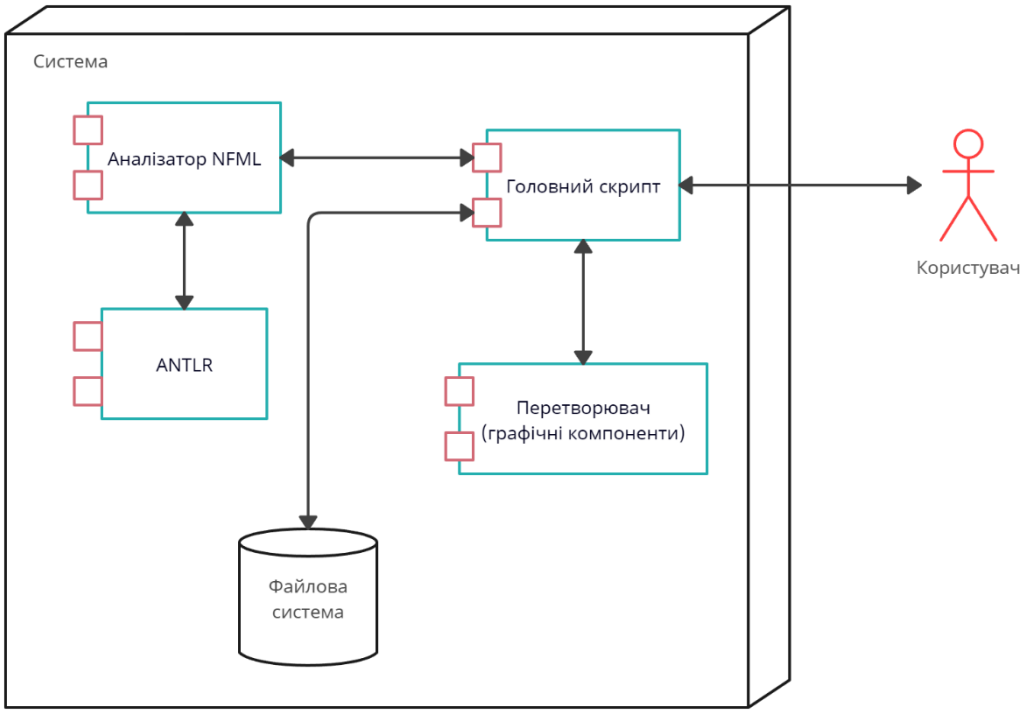
Система для розробки користувацького інтерфейсу

Компоненти системи (структурна схема)

ІАЛЦ.467200.004 Д1

Аркушів 1

Київ 2023 р



					ІАЛЦ.467200.004 Д1	
		№ докум.	Підпис	Дата		
Розробив		Рекечинський Д. О.			**Система для розробки користувацького інтерфейсу** Компоненти системи (структурна схема)	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Молчанова А. А.				
Н. Контр.		Волокита А. М.				КПІ ім. Ігоря Сікорського, ФІОТ, ІП-94
Затвердив						

ДОДАТОК 2

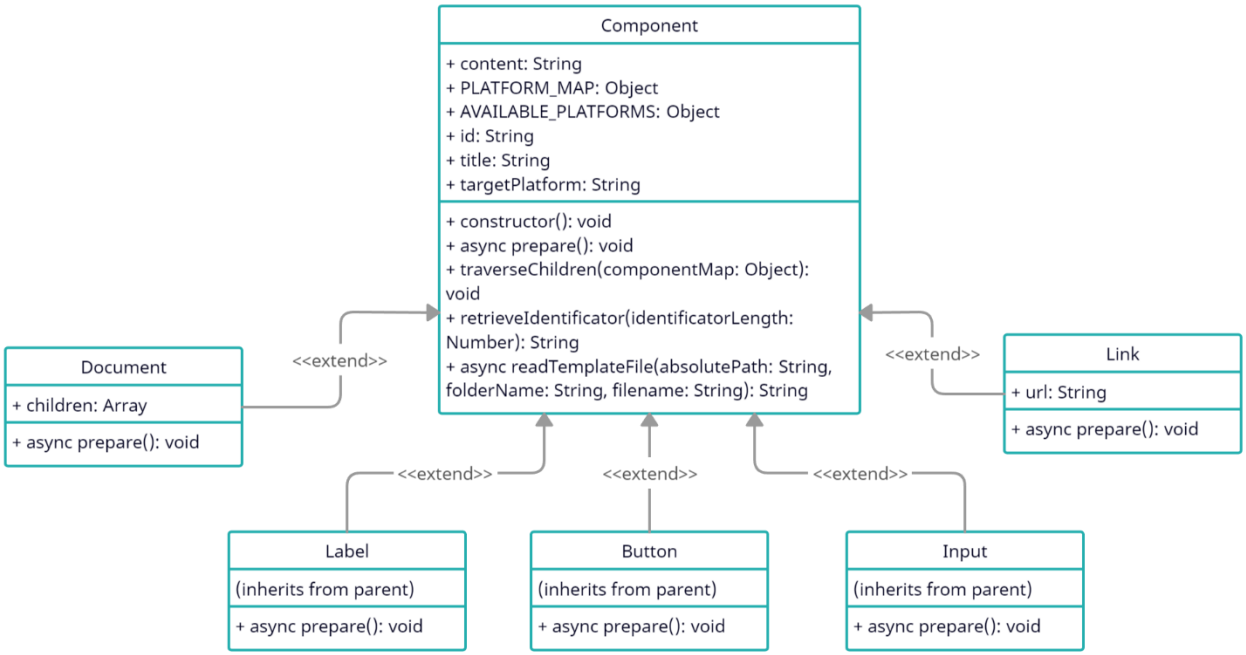
Система для розробки користувацького інтерфейсу

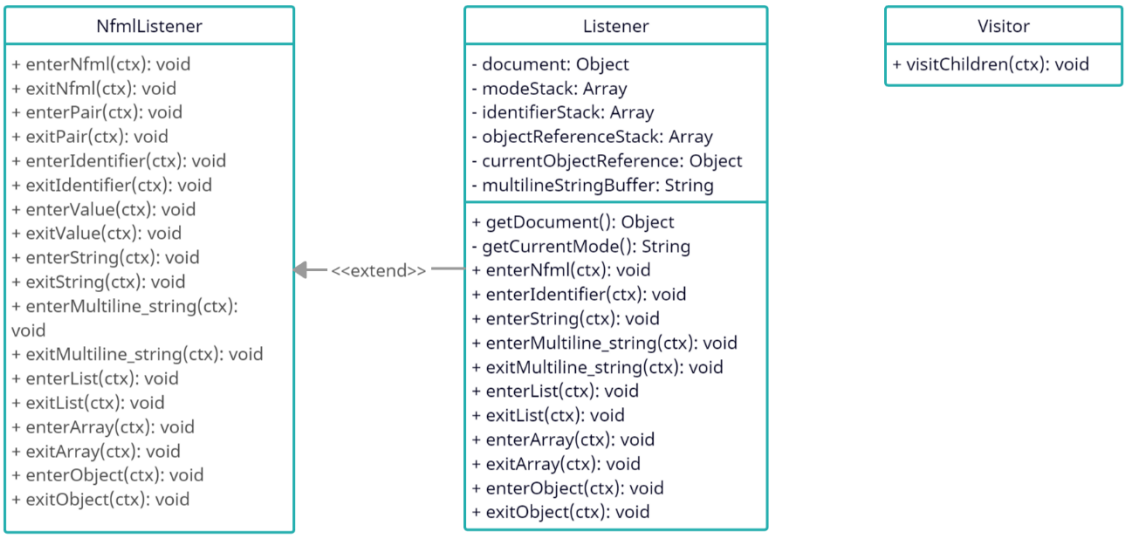
Діаграма класів (функціональна схема)

ІАЛЦ.467200.005 Д2

Аркушів 1

Київ 2023 р





					ІАЛЦ.467200.005 Д2	
		№ докум.	Підпис	Дата		
Розробив		Рекечинський Д. О.			**Система для розробки користувацького інтерфейсу** Діаграма класів (функціональна схема)	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Молчанова А. А.				
Н. Контр.		Волокита А. М.				КПІ ім. Ігоря Сікорського, ФІОТ, ІП-94
Затвердив						

ДОДАТОК 3

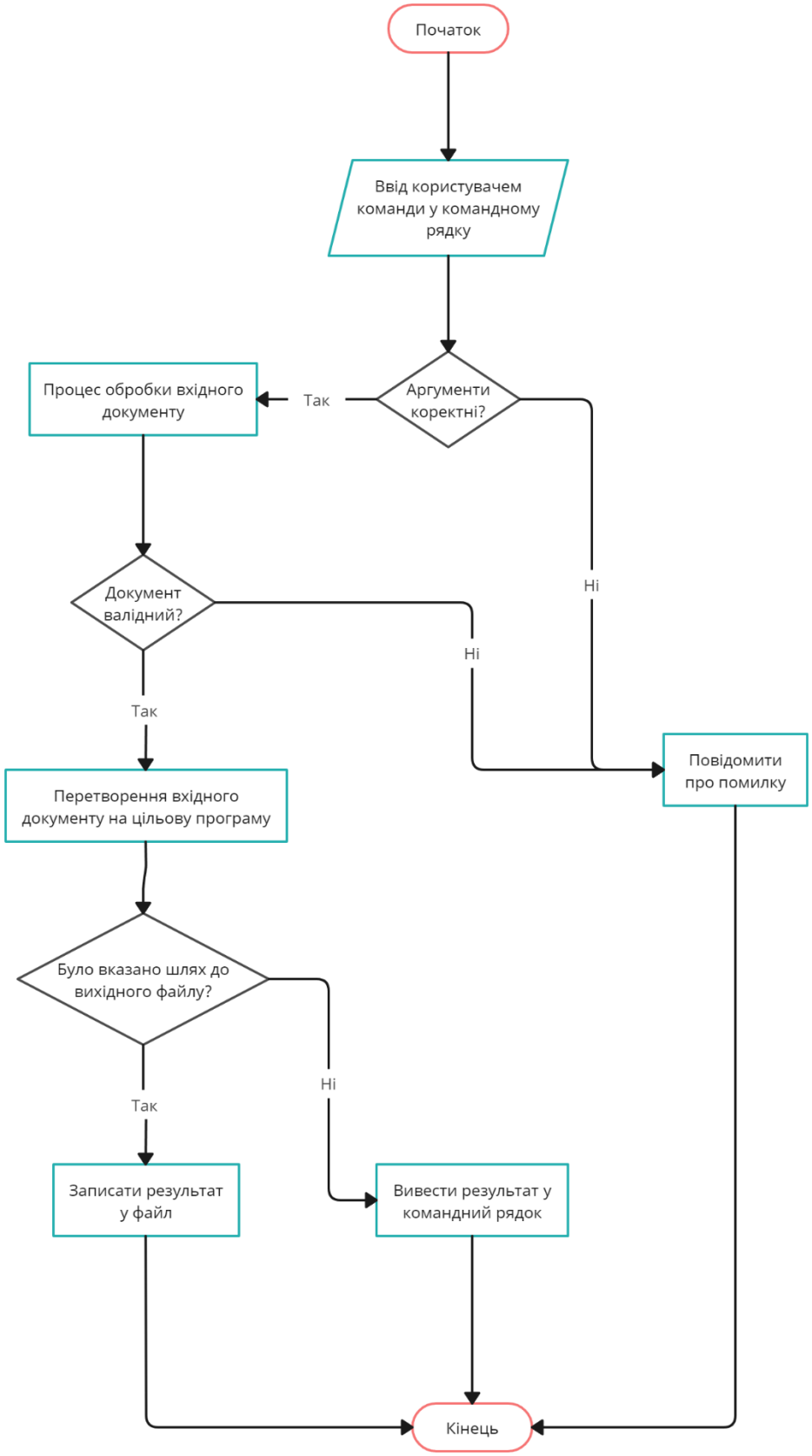
Система для розробки користувацького інтерфейсу

Алгоритм дій програмного забезпечення (принципова схема)

ІАЛЦ.467200.006 Д3

Аркушів 1

Київ 2023 р



					ІАЛЦ.467200.006 Д3	
		№ докум.	Підпис	Дата		
Розробив		Рекечинський Д. О.			**Система для розробки користувацького інтерфейсу** Алгоритм дій програмного забезпечення (принципова схема)	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Молчанова А. А.				
Н. Контр.		Волокита А. М.				КПІ ім. Ігоря Сікорського, ФІОТ, ІП-94
Затвердив						

ДОДАТОК 4

Система для розробки користувацького інтерфейсу

Текст програмного коду

ІАЛЦ.467200.007 Д4

Аркушів 19

Київ 2023 р

```
// index.js

import * as fs from 'fs/promises';
import { compile } from './src/nfml.js';
import { HELP_MODE, ARGUMENT_MAP, MODE_MAP, HELP_TEXT } from './constants.js';

const parsedArguments = {
    platform: 'HTML'
};

const fileSystemOptions = {
    encoding: 'utf8'
};

let argumentMode = '';
let foundHelpArgument = false;

export const main = async () => {
    // Ignore first two arguments, since they are "node" and "index.js"
    const inputArguments = process.argv.slice(2);

    for (const argument of inputArguments) {
        if (argumentMode) {
            const argumentProperty = MODE_MAP[argumentMode];
            parsedArguments[argumentProperty] = argument;
            argumentMode = '';
        } else {
            argumentMode = ARGUMENT_MAP[argument];

            if (!argumentMode) {
                throw new Error(`Argument ${argument} is not expectable`);
            } else if (argumentMode === HELP_MODE) {
                foundHelpArgument = true;
                break;
            }
        }
    }

    if (foundHelpArgument || !parsedArguments.filename) {
        console.log(HELP_TEXT);
        return;
    }

    const { filename, platform, outputFile } = parsedArguments;
    const input = await fs.readFile(filename, fileSystemOptions);

    const { error, compiledOutput } = await compile(input, platform);

    if (error) {
        console.error(`${filename} contains errors. Please, see above.`);
        process.exitCode = 1;
    } else if (outputFile) {
        await fs.writeFile(outputFile, compiledOutput, fileSystemOptions);
        console.log(`The result has been written to file ${outputFile}`);
    } else {
        console.log(compiledOutput);
    }
}

// If this is loaded as a main file
if (import.meta.url.endsWith(process.argv[1])) {
    main();
}
```

// constants.js

```
const FILENAME_MODE = 'mode:file';
const PLATFORM_MODE = 'mode:platform';
const OUTPUT_MODE = 'mode:output';
export const HELP_MODE = 'mode:help';

export const ARGUMENT_MAP = {
    '--file': FILENAME_MODE,
    '-f': FILENAME_MODE,
    '--platform': PLATFORM_MODE,
    '-p': PLATFORM_MODE,
    '--output': OUTPUT_MODE,
    '-o': OUTPUT_MODE,
    '--help': HELP_MODE,
    '-h': HELP_MODE
};

export const MODE_MAP = {
    [FILENAME_MODE]: 'filename',
    [PLATFORM_MODE]: 'platform',
    [OUTPUT_MODE]: 'outputFile'
};

export const HELP_TEXT = `
NFML (Novice-Friendly Markup Language) compiler
Usage: node index.js [options]

    -f, --file          Input NFML file. Mandatory parameter
    -p, --platform      Target platform.
                        If not specified, the target platform is HTML
    -o, --output        Output file.
                        If not specified, outputs to stdin.
    -h, --help          Show help.

Author: Dmytro Rekechynskyi
`;
```

// package.json

```
{
  "name": "nfml-parser",
  "version": "0.1.0",
  "description": "NFML markup language parser",
  "main": "index.js",
  "type": "module",
  "scripts": {
```

```
    "test": "node --experimental-vm-modules --no-warnings node_modules/jest/bin/jest.js"
  },
  "keywords": [
    "nfml",
    "markup",
    "parser"
  ],
  "author": "Dmytro Rekechynskyi",
  "license": "GPL-3.0-only",
  "dependencies": {
    "antlr4": "^4.13.0",
    "jest": "^29.5.0"
  }
}
```

// src/visitor.js

```
export default class Visitor {
    visitChildren(ctx) {
        if (!ctx) {
            return;
        }

        if (ctx.children) {
            const result = ctx.children.map(child => {
                if (child.children && child.children.length != 0) {
                    return child.accept(this);
                } else {
                    return child.getText();
                }
            });

            return result;
        }
    }
}
```

// src/nfml.js

```
import antlr from 'antlr4';
import NfmlLexer from './antlr/nfmlLexer.js';
import NfmlParser from './antlr/nfmlParser.js'
import Listener from './listener.js';
import Visitor from './visitor.js';
import { traverseDocument } from './components/traverseDocument.js';
```

```
export const compile = async (input, targetPlatform) => {
    const chars = new antlr.InputStream(input);
    const lexer = new NfmlLexer(chars);
    const tokens = new antlr.CommonTokenStream(lexer);
    const parser = new NfmlParser(tokens);
    parser.buildParseTrees = true;

    const tree = parser.nfml();
    tree.accept(new Visitor());

    if (tree.parser._syntaxErrors > 0) {
        return { error: true };
    }

    const listener = new Listener();
    antlr.tree.ParseTreeWalker.DEFAULT.walk(listener, tree);

    const document = listener.getDocument();

    const contents = await traverseDocument(document, targetPlatform);
    return { compiledOutput: contents };
};

// src/listener.js

// Base listener class
import NfmlListener from './antlr/nfmlListener.js';

const mode = {
    objectContext: 'OBJECT',
    multilineStringContext: 'MULTILINE_STRING',
    listContext: 'LIST',
    arrayContext: 'ARRAY'
};

const kebabToCamelCase = (string) =>
    string.replace(/-./g, (x) => x[1].toUpperCase());

const capitalize = (string) => string[0].toUpperCase() + string.slice(1);

export default class Listener extends NfmlListener {
    #document;
    #modeStack = [];
    #identifierStack = [];

    #objectReferenceStack = [];
    #currentObjectReference;
```

```
#multilineStringBuffer = '';

getDocument() {
    return this.#document;
}

#getCurrentMode() {
    const modeStack = this.#modeStack;

    if (modeStack.length === 0) {
        return null;
    }

    return modeStack[modeStack.length - 1];
}

// Enter a parse tree produced by nfmlParser#nfml.
enterNfml(ctx) {
    this.#document = {};
    this.#currentObjectReference = this.#document;
}

// Enter a parse tree produced by nfmlParser#identifier.
enterIdentifier(ctx) {
    const identifier = ctx.getText();
    const camelCaseId = kebabToCamelCase(identifier);
    this.#identifierStack.push(camelCaseId);
}

// Enter a parse tree produced by nfmlParser#string.
enterString(ctx) {
    const {start, stop} = ctx;
    let startIndex = -1;
    let stopIndex = -1;

    if (start && stop) {
        startIndex = start.start;
        stopIndex = stop.stop;
    }

    let value;

    if (startIndex < 0 && stopIndex < 0) {
        // Fallback
        value = ctx.getText();
    } else {
```

```
            // The purpose of doing like this is including whitespace
            // into string.
            // The thing is whitespace is redirected into channel HIDDEN,
            // so we need to get it back.
            // The most obvious solution is to read the whole token manually,
            // so it includes all the needed characters.
            value = ctx.start.getInputStream().getText(startIndex, stopIndex);
        }

        // If the string is "raw"
        if (value[0] === '\\') {
            // Replace the leading sign
            value = value.replace('\\', '');
        }

        const currentMode = this.#getCurrentMode();

        switch (currentMode) {
            case mode.multilineStringContext:
                this.#multilineStringBuffer += value + '\n';
                break;
            case mode.listContext:
                this.#currentObjectReference.push(value);
                break;
            default:
                const identifier = this.#identifierStack.pop();
                this.#currentObjectReference[identifier] = value;
        }
    }

    // Enter a parse tree produced by nfmlParser#multiline_string.
    enterMultiline_string(ctx) {
        this.#modeStack.push(mode.multilineStringContext);
    }

    // Exit a parse tree produced by nfmlParser#multiline_string.
    exitMultiline_string(ctx) {
        const identifier = this.#identifierStack.pop();
        const value = this.#multilineStringBuffer;
        this.#currentObjectReference[identifier] = value;

        this.#modeStack.pop();
        this.#multilineStringBuffer = '';
    }


    // Enter a parse tree produced by nfmlParser#list.
```

```
enterList(ctx) {
    const identifier = this.#identifierStack.pop();

    const currentObject = this.#currentObjectReference;
    const newObject = [];
    currentObject[identifier] = newObject;

    this.#objectReferenceStack.push(currentObject);
    this.#currentObjectReference = newObject;

    this.#modeStack.push(mode.listContext);
}

// Exit a parse tree produced by nfmlParser#list.
exitList(ctx) {
    const rootObject = this.#objectReferenceStack.pop();
    this.#currentObjectReference = rootObject;

    this.#modeStack.pop();
}


// Enter a parse tree produced by nfmlParser#array.
enterArray(ctx) {
    const identifier = this.#identifierStack.pop();

    const currentObject = this.#currentObjectReference;
    const newObject = [];
    currentObject[identifier] = newObject;

    this.#objectReferenceStack.push(currentObject);
    this.#currentObjectReference = newObject;

    this.#modeStack.push(mode.arrayContext);
}

// Exit a parse tree produced by nfmlParser#array.
exitArray(ctx) {
    const rootObject = this.#objectReferenceStack.pop();
    this.#currentObjectReference = rootObject;

    this.#modeStack.pop();
}


// Enter a parse tree produced by nfmlParser#object.
enterObject(ctx) {
```

```
        const currentMode = this.#getCurrentMode();

        if (currentMode === mode.arrayContext) {
            const currentObject = this.#currentObjectReference;
            const newObject = {};
            currentObject.push(newObject);

            this.#objectReferenceStack.push(currentObject);
            this.#currentObjectReference = newObject;
        } else {
            const identifier = this.#identifierStack.pop();

            if (identifier) {
                const currentObject = this.#currentObjectReference;
                const newObject = {};
                currentObject[identifier] = newObject;

                this.#objectReferenceStack.push(currentObject);
                this.#currentObjectReference = newObject;
            }
        }

        this.#modeStack.push(mode.objectContext);
    }

    // Exit a parse tree produced by nfmlParser#object.
    exitObject(ctx) {
        const identifier = this.#identifierStack.pop();
        const classId = capitalize(identifier);
        this.#currentObjectReference._class = classId;

        const rootObject = this.#objectReferenceStack.pop();
        if (rootObject) {
            this.#currentObjectReference = rootObject;
        }

        this.#modeStack.pop();
    }
}
```

// src/components/traverseDocument.js

```
import { ComponentMap } from "./ComponentMap.js";

export const traverseDocument = async (document, targetPlatform) => {
    const RootComponent = ComponentMap[document._class];
    const instance = new RootComponent(document, targetPlatform);
```

```
    instance.traverseChildren(ComponentMap);
    await instance.prepare();
    return instance.content;
};

// src/components/link/java/template.txt

Hyperlink {identifier} = new Hyperlink("{title}");
{identifier}.setOnAction(event -> {
    getHostServices().showDocument("{url}");
    event.consume();
});

// src/components/link/index.js

import Component from '../Component.js';

export default class Link extends Component {
    constructor (data, targetPlatform) {
        super(data, targetPlatform);

        this.url = data.url;
    }

    async prepare() {
        switch (this.targetPlatform) {
            case this.PLATFORM_MAP.html:
                const htmlFile = await this.readTemplateFile(import.meta.url, 'html',
'index.html');

                const preparedPage = htmlFile
                    .replace('{title}', this.title)
                    .replace('{url}', this.url);

                this.content = preparedPage;
                break;
            case this.PLATFORM_MAP.java:
                const javaFile = await this.readTemplateFile(import.meta.url, 'java',
'template.txt');
                const variableName = this.retrieveIdentificator();

                this.variableName = variableName;

                const preparedComponent = javaFile
                    .replace('{title}', this.title)
                    .replaceAll('{identifier}', variableName)
                    .replace('{url}', this.url);
```

```
                this.content = preparedComponent;
        }
    }
}

// src/components/link/html/index.html

    <a href="{url}">{title}</a>

// src/components/label/java/template.txt

Label {identifier} = new Label("{title}");

// src/components/label/index.js

import Component from '../Component.js';

export default class Label extends Component {
    constructor (data, targetPlatform) {
        super(data, targetPlatform);
    }

    async prepare() {
        switch (this.targetPlatform) {
            case this.PLATFORM_MAP.html:
                const htmlFile = await this.readTemplateFile(import.meta.url, 'html',
'index.html');

                const preparedPage = htmlFile
                    .replace('{title}', this.title);

                this.content = preparedPage;
                break;
            case this.PLATFORM_MAP.java:
                const javaFile = await this.readTemplateFile(import.meta.url, 'java',
'template.txt');
                const variableName = this.retrieveIdentificator();

                this.variableName = variableName;

                const preparedComponent = javaFile
                    .replace('{title}', this.title)
                    .replace('{identifier}', variableName);

                this.content = preparedComponent;
        }
```

```
    }
}
```

// src/components/label/html/index.html

```
    <p>{title}</p>
```

// src/components/input/java/template.txt

```
TextField {identifier} = new TextField();
{identifier}.setPromptText("{title}");
```

// src/components/input/index.js

```
import Component from '../Component.js';

export default class Input extends Component {
    constructor (data, targetPlatform) {
        super(data, targetPlatform);
    }

    async prepare() {
        switch (this.targetPlatform) {
            case this.PLATFORM_MAP.html:
                const htmlFile = await this.readTemplateFile(import.meta.url, 'html',
'index.html');

                const preparedPage = htmlFile
                    .replace('{title}', this.title);

                this.content = preparedPage;
                break;
            case this.PLATFORM_MAP.java:
                const javaFile = await this.readTemplateFile(import.meta.url, 'java',
'template.txt');
                const variableName = this.retrieveIdentificator();

                this.variableName = variableName;

                const preparedComponent = javaFile
                    .replace('{title}', this.title)
                    .replaceAll('{identifier}', variableName);

                this.content = preparedComponent;
        }
    }
}
```

```
// src/components/input/html/index.html

    <input type="text" placeholder="{title}">

// src/components/document/java/Example.java

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

// For screen calculation
import javafx.geometry.Rectangle2D;
import javafx.stage.Screen;

public class Example extends Application {
    @Override
    public void start(Stage primaryStage) {
//children
        VBox DOCUMENT_ROOT_NAME = new VBox();

//childrenAssignment
        Rectangle2D screenBounds = Screen.getPrimary().getBounds();
        double screenHeight = screenBounds.getHeight();
        double screenWidth = screenBounds.getWidth();

        long windowHeight = Math.round(screenHeight * 0.6);
        long windowWidth = Math.round(screenWidth * 0.6);

        Scene scene = new Scene(DOCUMENT_ROOT_NAME, windowWidth, windowHeight);
        primaryStage.setTitle("{title}");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}

// src/components/document/index.js

import Component from '../Component.js';

export default class Document extends Component {
```

```
    constructor (data, targetPlatform) {
        super(data, targetPlatform);
        this.children = data.children;
        this.rootName = "root";
    }

    async prepare() {
        const children = this.children;
        let childrenContent = '';

        // Let all the childs to be prepared
        await Promise.allSettled(children.map((child) => child.prepare()));

        switch (this.targetPlatform) {
            case this.PLATFORM_MAP.html:
                for (const child of children) {
                    childrenContent += child.content;
                }

                const htmlFile = await this.readTemplateFile(import.meta.url, 'html',
'index.html');

                const preparedPage = htmlFile
                    .replace('{title}', this.title)
                    .replace('{children}', childrenContent);

                this.content = preparedPage;
                break;
            case this.PLATFORM_MAP.java:
                let childrenAssignment = '';

                const rootAssignment = (childName) =>
                    `${this.rootName}.getChildren().add(${childName});\n`;

                const addIdentation = (text, identation = 8) => {
                    const identationPart = ' '.repeat(identation);
                    return identationPart + text.replace(/\n(?!$)/g, '\n' +
identationPart);
                };

                for (const child of children) {
                    childrenContent += addIdentation(child.content);
                    childrenAssignment                                          +=
addIdentation(rootAssignment(child.variableName));
                }
```

```
                const javaFile = await this.readTemplateFile(import.meta.url, 'java',
'Example.java');

                const preparedApp = javaFile
                    .replace('{title}', this.title)
                    .replaceAll('DOCUMENT_ROOT_NAME', this.rootName)
                    .replace('//children', childrenContent)
                    .replace('//childrenAssignment', childrenAssignment);

                this.content = preparedApp
                break;
        }
    }
}
```

// src/components/document/html/index.html

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{title}</title>

    <style>
        body > * {
            display: block;
            margin: 1rem;
        }
    </style>
</head>

<body>
{children}
</body>

</html>
```

// src/components/button/java/template.txt

```
Button {identifier} = new Button("{title}");
```

// src/components/button/index.js

```
import Component from '../Component.js';
```

```
export default class Button extends Component {
    constructor (data, targetPlatform) {
        super(data, targetPlatform);
    }

    async prepare() {
        switch (this.targetPlatform) {
            case this.PLATFORM_MAP.html:
                const htmlFile = await this.readTemplateFile(import.meta.url, 'html',
'index.html');

                const preparedPage = htmlFile
                    .replace('{title}', this.title);

                this.content = preparedPage;
                break;
            case this.PLATFORM_MAP.java:
                const javaFile = await this.readTemplateFile(import.meta.url, 'java',
'template.txt');
                const variableName = this.retrieveIdentificator();

                this.variableName = variableName;

                const preparedComponent = javaFile
                    .replace('{title}', this.title)
                    .replace('{identifier}', variableName);

                this.content = preparedComponent;
        }
    }
}
```

// src/components/button/html/index.html

```
    <button>{title}</button>
```

// src/components/ComponentMap.js

```
import Button from "./button/index.js";
import Document from "./document/index.js";
import Input from "./input/index.js";
import Link from "./link/index.js";
import Label from "./label/index.js";

export const ComponentMap = {
    Button,
```

```
    Document,
    Input,
    Link,
    Label
};

// src/components/Component.js

import { fileURLToPath } from 'url';
import * as fs from 'fs/promises';
import path from 'path';

export default class Component {
    content = '';
    PLATFORM_MAP = {
        html: 'HTML',
        java: 'Java'
    };
    AVAILABLE_PLATFORMS = Object.values(this.PLATFORM_MAP);

    constructor (data, targetPlatform) {
        if (!this.AVAILABLE_PLATFORMS.includes(targetPlatform)) {
            throw new Error(`The system does not support ${targetPlatform}`);
        }

        this.id = data.id;
        this.title = data.title;
        this.targetPlatform = targetPlatform;
    }

    async prepare () {}

    traverseChildren(componentMap) {
        if (this.children) {
            const result = [];

            for (const child of this.children) {
                const {_class, ...data} = child;
                const CurrentComponent = componentMap[_class];

                if (!CurrentComponent) {
                    throw new Error(`Component ${_class} was not found`);
                }

                const instance = new CurrentComponent(data, this.targetPlatform);
                instance.traverseChildren(componentMap);
```

```
                result.push(instance);
            }

            this.children = result;
        }
    }

    retrieveIdentificator(identificatorLength = 8) {
        if (this.id) {
            return this.id;
        }

        const FIRST_LETTER_CODE = 97;
        const LETTER_AMOUNT = 25;

        const randomCharacter = () => {
            const letterPosition = Math.round(LETTER_AMOUNT * Math.random());
            return String.fromCharCode(FIRST_LETTER_CODE + letterPosition);
        }

        let identificator = '';

        for (let i = 0; i < identificatorLength; i++) {
            identificator += randomCharacter();
        }

        return identificator;
    }

    async readTemplateFile(absolutePath, folderName, filename) {
        const directoryPath = path.dirname(fileURLToPath(absolutePath));
        const filePath = path.resolve(directoryPath, folderName, filename);
        const file = await fs.readFile(filePath, {encoding: 'utf8'});

        return file;
    }
}

// src/antlr/nfml.g4

// Grammar for NFML
grammar nfml;

/********************
* Fundamental rules
*********************/
```

```
nfml: object EOF;

pair: identifier COLON value NEWLINE+;

identifier: ID_LETTER+? (ID_SEPARATOR ID_LETTER+?)*;

value: object | string | multiline_string | list | array;

/********************
* Types of data
*********************/

string: (~NEWLINE)*;

multiline_string
    : MULTILINE_STRING_DELIMETER NEWLINE+ (string NEWLINE)+? MULTILINE_STRING_DELIMETER
    ;

list: LIST_OPEN NEWLINE+ (string NEWLINE+)* LIST_CLOSE;

array: ARRAY_OPEN NEWLINE+ (object NEWLINE+)* ARRAY_CLOSE;

object: NEWLINE* identifier OBJECT_OPEN NEWLINE+ pair* OBJECT_CLOSE NEWLINE*;

/********************
* Lexer rules
*********************/

// Literal values
ID_LETTER: [a-z];
ID_SEPARATOR: '-';
COLON: ':';

MULTILINE_STRING_DELIMETER: '---';

OBJECT_OPEN: '{';
OBJECT_CLOSE: '}';

LIST_OPEN: '[';
LIST_CLOSE: ']';

ARRAY_OPEN: '[[';
ARRAY_CLOSE: ']]';

NEWLINE: ('\r'? '\n');

// Comment rule
```

```
COMMENT: NEWLINE [ \t]* '#' ~('\r' | '\n')* -> skip;

// Comment at the beginning of file hack
COMMENT_AT_BEGINNING_OF_FILE:
    {this.column === 0}? [ \t]* '#' ~('\r' | '\n')* -> skip
    ;

// Ignore indentation whitespace
WHITESPACE: [ \t]+ -> channel(HIDDEN);

// Enable Unicode support
UNICODE_SET: [\u0000-\uFFFF];
```