Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)
Кафедра Інформатики
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

РОЗРОБКА ТЕРМІНАЛЬНОГО ТЕКСТОВОГО РЕДАКТОРА З ІНТЕРАКТИВНИМ ВИКОНАННЯМ КОДУ
(тема)

Виконав:
студент 4 курсу, групи ІТІНФ-20-2
Зіменко Н.А.
(прізвище, ініціали)
Спеціальності 122 Комп'ютерні науки
(код і повна назва спеціальності)
Тип програми освітньо-професійна
Освітня програма Інформатика
(повна назва освітньої програми)
Керівник ст. викл. Кіношенко Д.К.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри ____________ Кобилін О.А.
(підпис) (прізвище, ініціали)

2024 р.

Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)

Кафедра Інформатики
(повна назва)

Рівень вищої освіти перший (бакалаврський)

Спеціальність 122 Комп’ютерні науки
(код і повна назва)

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

ЗАТВЕРДЖУЮ:
Зав. кафедри ________________
(підпис)
«____»________________2024 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Зіменку Нікіті Андрійовичу
(прізвище, ім’я, по батькові)

1. Тема роботи Розробка термінального текстового редактора з інтерактивним виконанням коду

затверджена наказом по університету від 20 травня 2024 року № 464 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 28 травня 2024 р.

3. Вихідні дані до роботи наукова-технічна та науково-методична література, матеріали конференцій, мова програмування Go, середовище розробки GoLand.

4. Перелік питань, що потрібно опрацювати в роботі

1. Аналіз існуючих засобів розробки текстових користувацьких інтерфейсів.
2. Моделювання графічних компонентів, алгоритмів та підходів.
3. Проєктування графічних компонентів.
4. Програмна реалізація графічних компонентів.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) Актуальність проблеми побудови текстових користувацьких інтерфейсів, постановка задачі.

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата
Консультант з дотримання діючих стандартів та норм			

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Отримання завдання на кваліфікаційну роботу	08.04.2024	
2	Аналіз завдання, підбір літератури	09.04.2024-12.04.2024	
3	Аналіз літератури з досліджуваної проблеми	12.04.2024-18.04.2024	
4	Аналіз засобів розробки текстових інтерфейсів	18.04.2024-20.04.2024	
5	Проєктування графічних компонентів	20.04.2024-02.05.2024	
6	Програмна реалізація	02.05.2024-19.05.2024	
7	Оформлення пояснювальної записки	19.05.2024-24.05.2024	
8	Перевірка на плагіат	24.05.2024-28.05.2024	
9	Рецензування	28.05.2024-31.05.2024	
10	Підготовка презентації та доповіді	01.06.2024-05.06.2024	
11	Занесення роботи в електронний архів	05.06.2024	
12	Попередній захист кваліфікаційної роботи	05.06.2024	

Дата видачі завдання 8 квітня 2024 р.

Студент ____________________
(підпис)

Керівник роботи ____________________ ст. викл. Кіношенко Д.К.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка до кваліфікаційної роботи: 63 с., 3 табл., 39 рис., 30 джерел.

КОРИСТУВАЦЬКИЙ ІНТЕРФЕЙС, ГРАФІЧНИЙ ІНТЕРФЕЙС, ТЕКСТОВИЙ ІНТЕРФЕЙС, ТЕРМІНАЛЬНИЙ ЕМУЛЯТОР, ШАБЛОН ПРОЄКТУВАННЯ.

Об’єкт роботи – задача побудови користувацьких інтерфейсів.

Мета роботи – створення та програмна реалізація гнучкої і розширюваної системи побудови складних термінальних текстових інтерфейсів, що дозволить полегшити перехід між командними інтерфейсами користувача та графічними.

В процесі виконання кваліфікаційної роботи було проведено аналіз предметної області, спроєктовано та програмно реалізовано архітектурний підхід побудови програмних застосунків, а також базові компоненти графічних користувацьких інтерфейсів.

USER INTERFACE, GRAPHICAL INTERFACE, TEXTUAL INTERFACE, TERMINAL EMULATOR, DESIGN TEMPLATE.

The object of work is the task of building user interfaces.

The purpose of the work is to create and programmatically implement a flexible and extensible system for building complex terminal text interfaces that will facilitate the transition between command user interfaces and graphical user interfaces.

In the course of the qualification work, the subject area was analyzed, an architectural approach to building software applications, as well as the basic components of graphical user interfaces were designed and programmatically implemented.

ЗМІСТ

Перелік умовних позначень, символів, одиниць, скорочень і термінів............. 7
Вступ.. 8
1 Аналіз предметної області та постановка задачі... 10
1.1 Загальний огляд середовища виконання користувацьких інтерфейсів. 10
1.2 Загальний огляд користувацьких інтерфейсів....................................... 11
1.3 Загальний огляд процесу моделювання інтерактивності...................... 12
1.4 Загальний огляд підходів опису відображення і поведінки...................13
1.5 Оцінка існуючих інструментів для створення термінальних текстових користувацьких інтерфейсів... 15
1.6 Аналіз вимог до інструменту створення термінальних текстових користувацьких інтерфейсів... 17
1.7 Постановка задачі.. 18
2 Аналіз і моделювання шаблонів проєктування програмних застосунків..... 20
2.1 Загальний огляд принципів роботи термінальних емуляторів...............20
2.1.1 Алгоритми відображення... 20
2.1.2 Алгоритми опрацювання подій...24
2.2 Загальний огляд архітектурних шаблонів програмних застосунків...... 26
2.2.1 Аналіз шаблону проєктування Model View Controller................... 26
2.2.2 Аналіз шаблону проєктування Model View Presenter.....................28
2.2.3 Аналіз шаблону проєктування Model View ViewModel.................30
2.2.4 Загальне порівняння наведених архітектурних шаблонів............. 32
2.3 Розробка шаблону проєктування програмних застосунків DCSM........ 33
3 Програмна реалізація і тестування... 35
3.1 Розробка користувацького інтерфейсу програмного застосунку........... 35
3.2 Програмна реалізація шару взаємодії з даними..................................... 39

3.3 Програмна реалізація алгоритмів і компонентів відображення............42
3.4 Програмна реалізація процесу взаємодії користувача........................... 45
3.5 Програмна реалізація примітивів базових компонентів інтерфейсу..... 47
3.5.1 Компоненти візуалізації...48
3.5.2 Компоненти композиції... 51
3.6 Тестування...54
Висновки...60
Перелік джерел посилання.. 61

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

API – Application Programming Interface (програмний інтерфейс застосунку)

XML – eXtensible Markup Language (розширювана мова розмітки)

Composable – властивість системи чи компонентів, що дозволяє їх комбінувати та повторно використовувати в різних контекстах або застосунках.

TUI – Text User Interface (текстовий інтерфейс користувача)

ТТКІ – текстовий термінальний користувацький інтерфейс

GUI – Graphical User Interface (графічний інтерфейс користувача)

UI – User Interface (користувацький інтерфейс)

ВСТУП

Перші електронно-обчислювальні машини були не тільки технологічно складними, а й механічно вимогливими системами. Вважалось, що оператор має підлаштовуватись під комп'ютер, а не навпаки, що разом з обмеженими обчислювальними можливостями, вплинуло на сприйняття користувацьких інтерфейсів як зайве ускладнення системи, що зменшило б вже недостатні ресурси. Користувач не мав доступу до машини в реальному часі; всі інструкції попередньо завантажувались з перфокарт, які комп'ютер мав послідовно опрацювати і виконати.

Командні інтерфейси можна вважати першим поштовхом до розвитку процесу взаємодії з обчислювальними машинами. Транзакційна система з конкретними командами позитивно вплинула як на швидкість роботи оператора, так і на гнучкість усієї системи, проте, не стала легше у використанні. Потреба пам'ятати значну кількість команд, відповідних опціональних параметрів-значень до них і правильно складати транзакції – вимагала значних зусиль навіть у досвідчених користувачів. Продуктивність збільшилась, але через необхідність мануального вводу кожної інструкції кількість помилок тільки зростала.

Стрімкий розвиток інформаційних систем і розширення сфер їх застосування сильно вплинули на процес побудови користувацьких інтерфейсів. З кожним новим етапом взаємодія оператора і техніки стала більш інтуїтивною, змінивши конкретні команди, на комплексні графічні системи, що сховали внутрішні механізми виконання операцій.

Користувацькі інтерфейси пройшли довгий шлях від своєї появи, до сучасного вигляду. Створено уніфіковані дизайн-системи, що мають на меті полегшити процес їх побудови. Незважаючи на це, частина інструментів, в тому числі для створення інформаційних систем, залишились на етапі командного вводу. Це змушує розробників витрачати значний час на

ознайомлення з технологією і її налаштування, навіть якщо вона займає лише малу частину кінцевого продукту.

Актуальність даної роботи полягає у необхідності дослідження проблеми переходу термінальних інструментів з командних на текстові користувацькі інтерфейси. У кваліфікаційній роботі дана проблема розглядається на прикладі розробки термінального середовища виконання програмного коду.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Загальний огляд середовища виконання користувацьких інтерфейсів.

При введені користувачем команд для виконання, з ним взаємодіє не ядро операційної системи, а посередник, що називається командним інтерпретатором або оболонкою.

Командний інтерпретатор, або оболонка – це користувацький інтерфейс для доступу до служб операційної системи [1, 2]. Він інтерпретує команди, введені користувачем, зазвичай через текстовий інтерфейс, і перетворює їх на дії, які може виконати операційна система. Оболонки дозволяють користувачам запускати програми, керувати файлами і каталогами та виконувати інші команди через інтерфейс командного рядка.

По суті, оболонка виступає одночасно як інтерпретатор мови і міст між користувачем і операційною системою, полегшуючи пряме спілкування без необхідності взаємодії через графічні елементи [3]. Таким чином, вона безпосередньо надає оператору протокол для взаємодії з функціями системи, в свою чергу не звертаючись до ядра напряму, адже вона не є прямими інтерфейсом взаємодії з ним, а оперує через API ядра [4, 5], як і інші програмні застосунки (рис. 1.1).

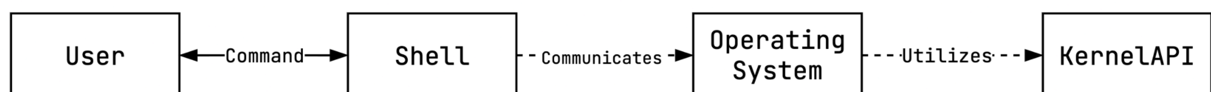

Рисунок 1.1 – Діаграма схеми взаємодії інтерпретатора команд

Дане розділення доступу є важливим, адже таким в такому разі забезпечується захист, який надає модель операційної системи, забезпечуючи,

що програми та користувацькі команди працюють в безпечному та контрольованому середовищі.

1.2 Загальний огляд користувацьких інтерфейсів

Користувацький інтерфейс – частина інформаційної системи, що призначена для взаємодії з користувачем. Вона впроваджує зрозумілі та зручні механізми не тільки відображення інформації, а й доповнення її користувачем. Різні типи інтерфейсів дозволяють виконувати ці завдання на різних рівнях абстракцій та з різною зручністю для користувача. В загальному сенсі виокремлюється два типи даних інтерфейсів:

– графічний;

– текстовий.

Графічний інтерфейс користувача – це тип користувацького інтерфейсу, який дозволяє користувачам взаємодіяти з електронними пристроями за допомогою графічних піктограм і візуальних індикаторів, таких як вторинні позначення, на відміну від текстових інтерфейсів, друкованих командних міток або текстової навігації. Графічні інтерфейси були розроблені, щоб зробити комп'ютери та інші пристрої доступнішими і простішими у використанні для широких верств населення, ефективно розширюючи коло тих, хто може ефективно взаємодіяти з програмним і апаратним забезпеченням [6].

Розглядаючи тему графічних користувацьких інтерфейсів потрібно зазначити важливий пункт, що саме вони були попередниками, з яких пізніше з’явились текстові. Незважаючи на зручність та очевидні переваги саме графічних інтерфейсів, це приклад технології, що з’явилась раніше свого часу і просто не могла коректно функціонувати на більшості тогочасних обчислювальних машин через брак ресурсів, тому її важливість полягає якраз у введені в обіг абстракцій, що стали основою всіх сучасних користувацьких інтерфейсів [7].

Текстовий інтерфейс є способом взаємодії між користувачем і обчислювальною машиною, і формуються шляхом використання композиції буквено-цифрових символів, а також створені з них елементи – графічні примітиви. Даний тип інтерфейсу дозволяє реалізувати всі базові аспекти інтерфейсу такі як: меню, кнопки, поля для тексту і вводу; а також компоненти їх організації, що виключає потребу знання специфічних команд, адже вони заміняються наведеними абстракціями.

1.3 Загальний огляд процесу моделювання інтерактивності

Інтерактивність – це явище, що визначає ступінь взаємодії між об'єктами операційної системи. В широкому розумінні, можна сказати, що інтерактивність досягається можливістю отримуючи інформацію в одному місці інформаційної системи, реагувати на неї в іншому, описаним заздалегідь способом. Таким чином, система надає користувачу можливість досягти бажаного результати, яким може бути як вивід потрібних вихідних даних, так і зміни стану зовнішнього або внутрішнього середовища застосунку. Саме на явищі інтерактивності будуються сучасні користувацькі інтерфейси.

В свою чергу, інформацією виступає певна подія. В контексті операційної системи це будь яка дія, що була зафіксована; вона може походити як від користувача, так і від самої системи або сторонніх джерел. Кожна подія має набір метаданих, які передаються спершу з циклу подій до ядра операційної системи, а далі звідти у оболонку, з якою взаємодіє користувач, і яка надає інформацію про події кожному застосунку, що її може опрацювати [8].

Процес моделювання інтеракцій починається з визначення спектру подій операційної системи, на які застосунок має реагувати, для того щоб користувачі могли досягати поставленої мети при взаємодії з інформаційною системою. Ефективне моделювання вимагає балансу між гнучкістю та

контролем. Інтерфейси мають бути достатньо гнучкими, щоб обробляти різноманітні взаємодії користувачів, але водночас контрольованими.

1.4 Загальний огляд підходів опису відображення і поведінки

При побудові користувацьких інтерфейсів процес опису порядку і форми відображення та його поведінки має фундаментальну роль. Отже, маємо на меті обрати підхід, що не буде страждати на зайву багатослівність і виходящу з цього заплутаність, матиме чітку структуру і буде максимально наближеним до звичного для оператора середовища розробки. Таким чином, основними підходами для опису відображення і поведінки є:

– метамова;

– предметно-орієнтована мова.

Метамова надає структуру для опису даних, але не накладає на ці дані жодних обмежень, щодо предметної області або поведінки при обробці. Вона визначає синтаксис розмітки, який можна використовувати для структурування інформації в ієрархічній формі. Основний сенс і обробку цих структурованих даних надають саме застосунки, що інтерпретують метамову. І вже саме ці застосунки є предметно-орієнтованими. Наприклад, можна привести мову XML, що мала широку популярність в розробці користувацьких інтерфейсів мобільних застосунків.

Лістинг 1.1 Приклад опису користувацького інтерфейсу мовою XML для Android застосунку:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
```

```
    android:orientation="vertical"
    android:padding="16dp">

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, welcome to our app!"
</LinearLayout>
```

Лістинг 1.2 Приклад опису користувацького інтерфейсу специфічною метамовою:

```
vstack [width: 30, height: 10]

    expand
        border
            expand
                text "Hello, welcome to our app!"
```

Мова, що має на меті опис користувацьких інтерфейсів, особливо якщо вони складаються з великої кількості активних екранів, повинна мати лаконічний і чітко структурований вигляд. На прикладі двох фрагментів коду можна зазначити наслідки недотримання приведених аксіом. Якраз багатослівність і заплутаність XML-подібних мов привели до еволюційного переходу до використання предметно-орієнтованих мов.

Якщо не розглядати деталі реалізації конкретної проблемно- або предметно-орієнтованої мови, можна сказати, що вона є спрощеною формою мови програмування, в більшості випадків неповною за Тьюрингом, чия структура і принцип побудови відповідає проблемі, яку вона призначена

вирішити. Виходячи з назви, зрозуміла концепція і сфера застосування – це вирішення конкретних проблем специфічної предметної області, зазвичай, розробки програмних застосунків. Наприклад, класичним представником предметно-орієнтованої мови є декларативний інтерфейс бібліотеки Jetpack Compose. Вона дозволяє будувати комплексні користувацькі інтерфейси з підтримкою лінійних або реактивних моделей інтерактивності. Для реалізації вже представленого тривіального екрану застосунку, вище імплементованим з використанням метамови потрібно визначити функцію з специфічною анотацією «Composable» [9, 10], в тілі функції потрібно зазначити які компоненти інтерфейсу буде використано, як вони будуть розміщуватись і якщо потрібно, як будуть реагувати на зовнішні події [11, 12].

Лістинг 1.3 Приклад опису користувацького інтерфейсу предметно-орієнтованою мовою Jetpack Compose:

```
@Composable
fun JetpackCompose() {
    Column(
        modifier = Modifier.padding(16.dp)
    ) {
        Text(
            text = "Hello, welcome to our app!",
            fontSize = 18.sp
        )
    }
}
```

Таким чином, можна визначити, що саме такий вид опису програмних компонентів можна вважати еталонним для вирішення проблем актуальної предметної області.

1.5 Оцінка існуючих інструментів для створення термінальних текстових користувацьких інтерфейсів

Оцінка існуючих інструментів для створення термінальних текстових користувацьких інтерфейсів є важливою для розуміння ландшафту доступних технологій, їхніх можливостей та обмежень [13]. Різні інструменти та бібліотеки були розроблені для полегшення створення TUI, кожен з яких пропонує унікальні функції та можливості. Досліджуючи ці інструменти, ми можемо виявити найкращі практики, поширені проблеми та області, де необхідні інновації. Ця оцінка також надасть уявлення про вподобання та очікування користувачів, що буде мотивувати розробку нового, більш ефективного інструменту.

Одним із найвідоміших інструментів у цій області є ncurses. NCurses є широко використовуваною бібліотекою для створення TUI у Unix-подібних системах [14]. Вона надає функції для маніпулювання екраном терміналу, обробки введення користувача та управління вікнами і підвікнами. Хоча ncurses є потужною та гнучкою, вона має круту криву навчання та вимагає від розробників гарного розуміння низькорівневої обробки терміналу. Незважаючи на складність, вона залишається популярним вибором завдяки своїй надійності та обширній документації.

Іншим значущим інструментом є Termbox, який пропонує простішу альтернативу NCurses. Termbox зосереджується на наданні мінімалістичного API, який легко використовувати та розуміти. Він підтримує базові функції, такі як відображення тексту, обробка введення та управління кольорами. Однак, простота Termbox обмежує його можливості порівняно з NCurses. Він підходить для проєктів малого та середнього розміру, але може не відповідати вимогам складніших застосувань, які потребують розширених можливостей.

На завершення, оцінка існуючих інструментів для створення термінальних текстових користувацьких інтерфейсів виявляє різноманітний ландшафт рішень, кожне з яких має свої сильні та слабкі сторони. Ncurses та

Termbox обслуговують різні рівні складності, тоді як Blessed та blessed-contrib приносять сучасні практики розробки у створення TUI. Аналізуючи ці інструменти, ми можемо виявити прогалини та можливості для інновацій, що в кінцевому підсумку спрямовуватиме розробку нового інструменту, який поєднує зручність використання, гнучкість та продуктивність для задоволення потреб сучасних розробників [15-19].

1.6 Аналіз вимог до інструменту створення термінальних текстових користувацьких інтерфейсів

Створення термінальних текстових користувацьких інтерфейсів вимагає інструментів, які задовольняють різні технічні та функціональні вимоги. Важливою вимогою є висока адаптивність і конфігурованість, що дозволяє розробникам легко модифікувати інтерфейс під специфічні потреби різних застосунків [20-21]. Інструмент повинен підтримувати різноманітні операційні системи, забезпечуючи широке використання термінальних інтерфейсів на різних платформах, в той час бути достатньо оптимізованим, щоб система витрачала якомога менше обчислювальних ресурсів [22].

API, що пропонується інструментом, має бути інтуїтивно зрозумілим та легким у використанні, що сприяє швидкій розробці та інтеграції інтерфейсів без глибоких знань специфікацій.

Ефективність і продуктивність інструменту також є ключовими, оскільки термінальні інтерфейси часто використовуються на обладнанні з обмеженими ресурсами. Інструмент має забезпечувати швидку реакцію та високу продуктивність, що є важливим для забезпечення задовільного користувацького досвіду.

Додатково, інструменти для створення ТТКІ повинні дозволяти розширення функціональності через опис додаткових компонентів інтерфейсу, що будуть задовольняти вимогам розробників. Це забезпечує гнучкість у впровадженні нових функцій та інтеграції з іншими системами та

бібліотеками. Здатність відображати складні дані через гнучкі візуальні компоненти важлива для ефективної візуалізації інформації, що підтримує більш складні інтеракції із користувачем.

Підтримка інтерактивності є фундаментальною, оскільки інтерфейси мають забезпечувати не лише відображення інформації, а й активну взаємодію з користувачем. Це включає можливість користувача взаємодіяти з різними елементами інтерфейсу, такими як меню, списки, панелі прокручування тощо.

1.7 Постановка задачі

Виходячи з аналізу області текстових користувацьких інтерфейсів і можливостей подальшого їх використання, а також проблематики реалізації окремих аспектів їх керування, розширення і відображення, необхідно спроєктувати загальний підхід побудови текстових термінальних інтерфейсів і програмно втілити відповідний інструмент розробки, що буде задовольняти описані вимоги.

Об’єктом роботи є задача побудови користувацьких інтерфейсів.

Мета роботи є створення та програмна реалізація гнучкої і розширюваної системи побудови складних термінальних текстових інтерфейсів, що дозволить полегшити перехід між командними інтерфейсами користувача та графічними.

Для інструменту створення термінальних текстових користувацьких інтерфейсів можна зробити такі висновки:

– інструмент має забезпечувати високу адаптивність і конфігурованість;

– підтримка різних операційних систем є обов'язковою, оскільки термінальні інтерфейси часто використовуються на різних платформах;

– API інструменту має бути інтуїтивно зрозумілим та легким у використанні;

– інструмент повинен забезпечувати високу продуктивність, особливо на обладнанні з обмеженими ресурсами;

– можливість розширення функціональності через плагіни або модулі, що дає розробникам можливість адаптувати інструмент під змінювані потреби;

– інструмент має дозволяти відображення складних даних через гнучкі візуальні компоненти, важливі для ефективної візуалізації інформації;

– підтримка інтерактивності є критичною.

2 АНАЛІЗ І МОДЕЛЮВАННЯ ШАБЛОНІВ ПРОЄКТУВАННЯ ПРОГРАМНИХ ЗАСТОСУНКІВ

2.1 Загальний огляд принципів роботи термінальних емуляторів

Термінальний емулятор – це складне програмне забезпечення, яке імітує функціональність апаратних терміналів у сучасному комп'ютерному середовищі, забезпечуючи інтерфейс для взаємодії з операційною системою через текстові команди.

2.1.1 Алгоритми відображення

Процес відтворення в термінальному емуляторі починається з приймання даних, які містять як прямий текстовий вивід, так і різні керівні послідовності. Ці дані можуть надходити як безпосередньо від користувача, так і через мережеві з'єднання, якщо термінальний емулятор використовується для віддаленого доступу.

Керуючі послідовності відіграють ключову роль у визначенні того, як текст має бути представлений. Вони можуть містити інструкції зі зміни кольору тексту та фону, переміщення курсору, очищення частини екрана або всього екрана, а також стилізації тексту, такими як жирний або курсив. Дані послідовності обробляються термінальним емулятором для активації відповідних функцій відображення.

Коли термінальний емулятор інтерпретує ці послідовності, він оновлює своє внутрішнє уявлення екрана. Це включає в себе визначення місця розташування кожного символу на екрані, що вимагає точного слідування поточному положенню курсору, яке змінюється з кожною новою командою.

Застосування стилів (рис. 2.1), таких як кольори тексту (табл. 2.2) і фону (табл. 2.3), жирність або курсив (табл. 2.1), також оперуються через

керуючі коди, що дає змогу термінальному эмулятору динамічно адаптуватися до завдань візуалізації тексту.

Рисунок 2.1 – Приклад встановлення стилю для заданого тексту

Таблиця 2.1 – ANSI коди-модифікатори стилю ASCII символів

Код	**Атрибут**	**Призначення**
1	**2**	**3**
Reset	0	Скинути всі атрибути
Bold	1	Встановити жирний шрифт
Dim	2	Встановити напівпрозорий шрифт
Italic	3	Встановити курсивний шрифт
Underline	4	Встановити підкреслений шрифт
Blink	5	Мигання курсору (повільне)
Reverse	7	Поміняти місцями колір тексту і фону

Продовження таблиці 2.1

1	2	3
Hidden	8	Приховати курсор
Strikethrough	9	Встановити закреслений шрифт

Таблиця 2.2 – Цифрові коди-модифікатори кольору ASCII символів

Код	Колір	Опис
Black	30	Чорний колір
Red	31	Червоний колір
Green	32	Зелений колір
Yellow	33	Жовтий колір
Blue	34	Синій колір
Magenta	35	Фіолетовий колір
Cyan	36	Блакитний колір
White	37	Білий
Default	39	Стандартний для терміналу колір
Bright Black	90	Сірий колір
Bright Red	91	Світло-червоний колір
Bright Green	92	Світло-зелений колір
Bright Yellow	93	Світло-жовтий колір
Bright Blue	94	Світло-синій колір
Bright Magenta	95	Світло-фіолетовий колір
Bright Cyan	96	Світло-блакитний колір
Bright White	97	Світло-білий колір

Таблиця 2.3 – Цифрові коди-модифікатори кольору ASCII символів

Код	**Колір**	**Опис**
Black	40	Чорний колір
Red	41	Червоний колір
Green	42	Зелений колір
Yellow	43	Жовтий колір
Blue	44	Синій колір
Magenta	45	Фіолетовий колір
Cyan	46	Блакитний колір
White	47	Білий
Default	49	Стандартний для терміналу колір
Bright Black	100	Сірий колір
Bright Red	101	Світло-червоний колір
Bright Green	102	Світло-зелений колір
Bright Yellow	103	Світло-жовтий колір
Bright Blue	104	Світло-синій колір
Bright Magenta	105	Світло-фіолетовий колір
Bright Cyan	106	Світло-блакитний колір
Bright White	107	Світло-білий колір

Крім того, процес візуалізації в термінальних емуляторах включає в себе оптимізацію для забезпечення чіткості відображення та читабельності тексту. Це означає, що при кожній зміні даних або при введенні нових команд, термінальний емулятор повинен швидко перемальовувати екран, щоб відображати нову інформацію. Такий підхід вимагає високої продуктивності від алгоритмів відтворення, щоб забезпечити низьку затримку і високу чуйність інтерфейсу. Один із підходів, який допомагає досягти потрібної продуктивності, – це використання алгоритму буферизації тексту [23].

На початку процесу, всі дані, введені користувачем або отримані від додатка, спочатку накопичуються в тимчасовому буфері. Це дає змогу системі згрупувати інформацію та обробити її порціями, що значно ефективніше порівняно з обробкою кожного символу окремо. У цьому буфері дані аналізуються на предмет керуючих послідовностей, які можуть містити команди на зміну кольору, стилю тексту або переміщення курсору. Ці команди застосовуються до того, як текст буде відтворено на екрані.

Коли настає момент відтворення, система визначає, які ділянки буфера змінилися з моменту останнього відтворення. Це дає змогу перемалювати тільки ті частини екрана, де відбулися зміни, мінімізуючи таким чином кількість операцій відтворення. Такий підхід, відомий як метод «брудних прямокутників», дає змогу значно знизити навантаження на процесор і прискорити відображення тексту. Процес візуалізації також пов'язаний із технічними складнощами, такими як коректне відображення нестандартних символів або символів різних мов, що вимагає підтримки широкого спектра кодувань і шрифтів.

Таким чином, сучасні термінальні емулятори є результатом просунутої інженерії, що забезпечує гнучкість і потужність у відображенні текстової інформації в найрізноманітніших сценаріях використання.

2.1.2 Алгоритми опрацювання подій

Термінальні емулятори ефективно керують взаємодією між користувачем і операційною системою, інтерпретуючи і обробляючи різні типи подій введення і системні повідомлення. Це включає не тільки пряме текстове введення з клавіатури, а й складніші форми інтерактивності, як-от події вводу з миші або клавіатури та системні сигнали.

Коли користувач вводить дані з клавіатури, кожне натискання клавіші транслюється в символи або команди, які потім передаються в командну оболонку або іншу програму, що працює в емуляторі. Це основний механізм

взаємодії користувача з системою через термінальний емулятор, і його ефективність критично важлива для забезпечення швидкого і точного відгуку на команди користувача.

Додатково, сучасні термінальні емулятори можуть обробляти події миші, що розширює їхню функціональність за межі традиційного текстового введення. Підтримка подій миші включає в себе реакцію на натискання і рухи миші, що може бути використане для управління текстовим курсором, виділення тексту для копіювання і вставки, а також для інтерактивного управління призначеним для користувача інтерфейсом, наприклад, в текстових редакторах або додатках з меню.

Також термінальні емулятори повинні коректно реагувати на системні сигнали, такі як зміна розмірів вікна. Коли розмір вікна терміналу змінюється, це вимагає від емулятора перерахувати, як інформація має бути відображена, що може включати в себе зміни в розташуванні тексту, перерозподіл рядків і стовпців, а також адаптацію активного вмісту до нового розміру вікна.

Обробка вхідного виводу від шела або інших програм також є важливою функцією термінального емулятора. Виведення може містити текстові дані, які мають бути відображені, а також керівні послідовності, які емулятор має інтерпретувати для правильного форматування тексту, включно з кольорами, стилями та розташуванням елементів інтерфейсу.

Ефективність забезпечення даних процесів, вимагає від емулятора не тільки відображати необхідну інформацію, а й динамічно інтерпретувати та застосовувати комплексні керуючі команди в реальному часі, щоб підтримувати актуальність і читабельність виводу.

Таким чином, термінальні емулятори є багатофункціональними інструментами, які забезпечують міст між користувачем і комп'ютерною системою, адаптуючись до різноманітних форм введення і управління для забезпечення гладкої та інтуїтивно зрозумілої роботи.

2.2 Загальний огляд архітектурних шаблонів програмних застосунків

2.2.1 Аналіз шаблону проєктування Model View Controller

MVC є найпоширенішим шаблоном у розробці будь-якого програмного забезпечення (рис. 2.2). Попри зручність для побудови невеликих проєктів, не спрямованих на велике навантаження, даний шаблон в наш час є непридатним для використання. Перш за все, це обумовлено некоректним сучасним трактуванням його суті, адже на момент формулювання концепції, поняття графічного інтерфейсу ще не увійшло в обіг [24].

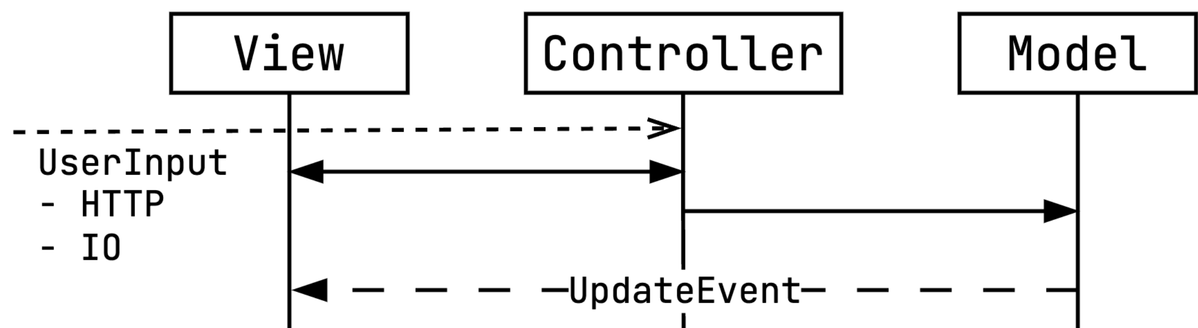

Рисунок 2.2 – Діаграма еталонного представлення шаблону MVC

В основі ідеї MVC, як і ключовим аспектом для всіх наступних відгалужень, є «відокремлене представлення». Сенс відокремленого представлення полягає в тому, щоб провести чітку межу між доменними об'єктами, що відображають реальний світ, і об'єктами подання, якими є GUI елементи на екрані користувача.

Доменні об'єкти, мають бути повністю незалежними і працювати без посилань на представлення, вони повинні мати можливість підтримувати множинні представлення, можливо навіть одночасно. Дана теоретична модель, яка була першочергово запропонована до використання, не набула популярності в загальному використанні, в свою чергу, актуальною стала

повна її протилежність, аспект відокремленості компонентів було відкинуто на вимогу зменшення складності й пришвидшення розробки (рис. 2.3).

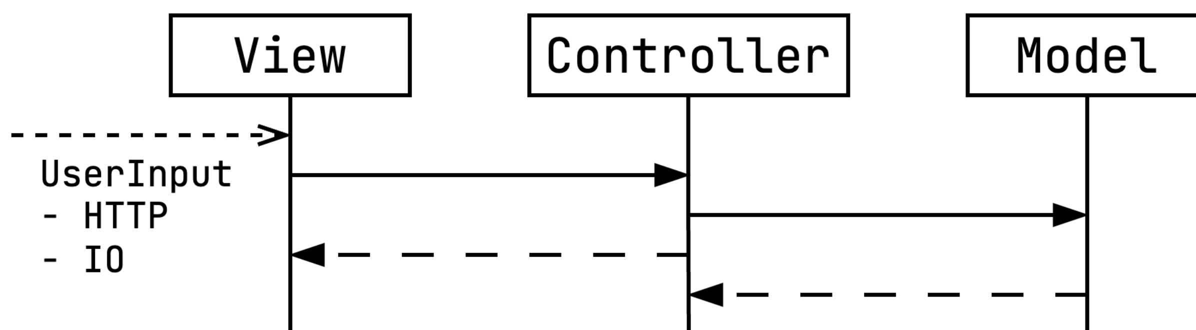

Рисунок 2.3 – Діаграма загального представлення шаблону MVC

Некоректна модель довгий час широко використовувалась, особливо в області веб-розробки, доки не була заміщена, через суттєві недоліки, основними з яких є:

– складність масштабування;

– складності тестування;

– дублювання логіки.

Складність масштабування великих додатків, побудованих на основі шаблону MVC є його найбільшою проблемою. Породжена реалізація MVC описує, що всі компоненти (модель, відображення і контролер) тісно пов’язані між собою, що експоненційно збільшує залежність між ними в процесі збільшення обсягу застосунку. Наприклад, зміни в моделі можуть вимагати каскадних змін в контролерах та видах, що часто унеможливлює внесення оновлень та додавання нових функцій без порушення існуючої функціональності та значних змін.

Масштабування суттєво погіршує загальну продуктивність, оскільки збільшення кількості компонентів може призвести до зниження швидкодії за рахунок збільшення обсягу взаємодії між шарами. Великі додатки часто вимагають більшого розділення компонентів та розподілу обробки даних, що

може бути важко досягти у рамках стандартної моделі MVC без значного перепроєктування архітектури.

У рамках MVC тісна взаємозалежність компонентів ускладнює процес модульного тестування. Кожен контролер, який залежить від певної моделі та взаємодіє з певним видом, вимагає включення всіх цих компонентів під час моделювання його роботи. Необхідність ізоляції контролера від зовнішніх залежностей збільшує кількість ресурсів на проведення загального тестування, за рахунок потреби у створенні імітаторів поведінки шарів моделей і відображення. Крім того, часті зміни в одному компоненті можуть вимагати повторних багаторазових перевірок інших.

Дублювання логіки у MVC часто виникає через те, що різні контролери мають схожу логіку обробки даних користувача або взаємодії з моделлю. Наприклад, якщо декілька контролерів обробляють введення даних користувача, логіка валідації даних може бути відтворена у багатьох місцях, замість того щоб бути уніфікованою в одному компоненті. Це призводить до збільшення витрат на утримання програмного коду, оскільки однакові зміни потрібно вносити в кілька місць, збільшуючи ймовірність помилок і ускладнюючи рефакторинг. Дублювання може також ускладнити внесення змін в логіку, особливо коли ці зміни потрібно синхронізувати міжількома частинами додатка, що розподілені між багатьма контролерами. Це вимагає додаткового обережного планування та тестування для забезпечення консистентності поведінки додатка.

2.2.2 Аналіз шаблону проєктування Model View Presenter

MVP є одним з нащадків MVC, що отримав суттєву відмінність у процесі взаємодії з діями користувача, саме тому дана модель є більш досконалою з погляду на реалізацію представлення, що виконує тільки свою безпосередню функцію. Користувач може взаємодіяти з формою віджетів, але

якщо віджет стосується логіки інтерфейсу, View буде передавати подію модулю Presenter.

В свою чергу, Presenter (рис. 2.4) відповідає за всю логіку користувацького інтерфейсу та за синхронізацію моделі і представлення. Таким чином, ми маємо, що цикл виконання починається в представленні, звідки він передає подію в Presenter, який вже вирішує як має оновлюватись модель і після цього синхронізує всі зміни між моделлю і представленням. Важливо зазначити те, що Presenter не звертається до представлення напряму, замість цього використовується певний інтерфейс.

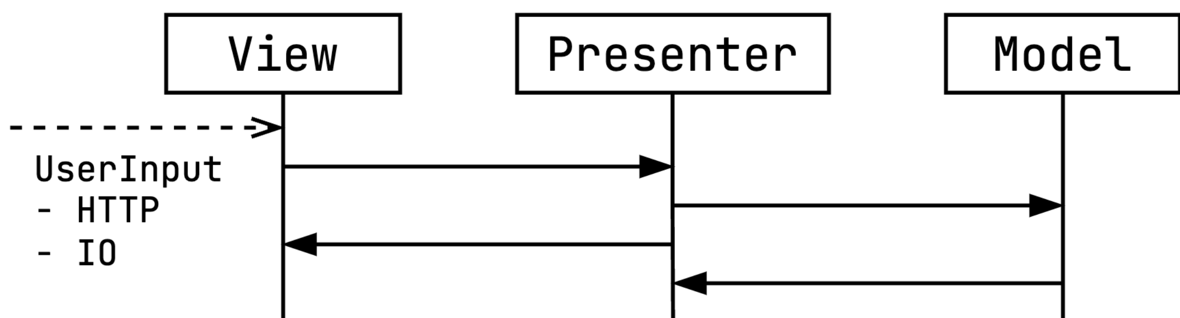

Рисунок 2.4 – Діаграма загального представлення шаблону MVP

Незважаючи на певні переваги, такі як покращене розділення відповідальності та легшість тестування, MVP також стикається з рядом проблем, що обмежують його застосування у сучасних розробках. Основними можна виділити:

– проблеми з гнучкістю взаємодії;

– обмежена підтримка сучасних вимог до користувацьких інтерфейсів.

Жорстке визначення ролей між компонентами, що може ускладнити реагування на зміни в користувацьких вимогах або технологічних умовах. Presenter часто відповідає за взаємодію між Model та View, що робить його перевантаженим і важким для модифікації без впливу на інші компоненти системи. Така централізація може вести до складнощів при необхідності

адаптувати або масштабувати додаток. Крім того, у деяких сценаріях, особливо в сучасних додатках з різноманітними і динамічними користувацькими інтерфейсами, потреба в постійному обміні інформацією між View і Presenter може призводити до зайвої складності та підвищення навантаження на розробників при додаванні нових функцій.

MVP може не завжди ефективно справлятися з сучасними вимогами до динамічних і багатофункціональних користувацьких інтерфейсів. Presenter, який виступає в ролі посередника між Model та View, часто не здатний коректно і швидко обробляти зміни або події, які відбуваються в інтерфейсі, що може призвести до втрати синхронізації між компонентами віджетів інтерфейсу і поточного стану даних або логіки додатку. Дана проблема особливо актуальна в ситуаціях, де потрібна коректна інтерактивність інтерфейсу та швидке оновлення даних, в реальному часі.

2.2.3 Аналіз шаблону проєктування Model View ViewModel

MVVM є архітектурним шаблоном, критично відмінним від MVC та MVP завдяки акценту на зв'язуванні даних. Ця особливість дозволяє автоматично оновлювати інтерфейс користувача при змінах у моделі, що спрощує синхронізацію між моделлю та представленням без прямого втручання. Зв'язування даних в MVVM сприяє створенню гнучких і легко адаптованих додатків, що особливо важливо для складних користувацьких інтерфейсів.

У MVVM, ViewModel не взаємодіє напряму з View, подібно до MVP, де Presenter також уникає прямого зв'язку з View. Це розділення покращує тестування та підтримку коду, оскільки зменшує залежність між компонентами системи. View використовує зв'язування для прямого з'єднання своїх властивостей з властивостями ViewModel, що автоматизує багато процесів взаємодії (рис. 2.5).

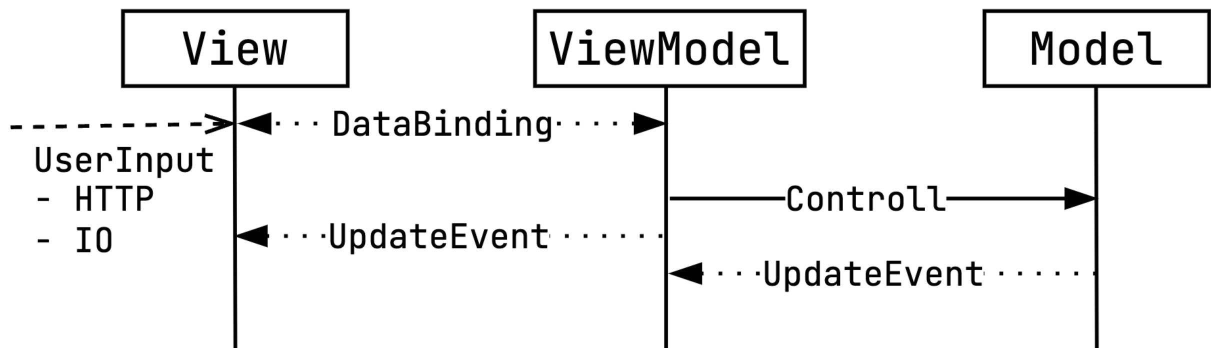

Рисунок 2.5 – Діаграма загального представлення шаблону MVVM

Взаємодії в MVVM часто ґрунтується на подіях, що дозволяє ViewModel обробляти зміни даних та користувацькі взаємодії без прямого втручання у компоненти інтерфейсу. Використання шини повідомлень може бути доречним у складних системах для синхронізації поведінки між різними частинами архітектури, хоча це не є обов'язковою частиною MVVM і може бути адаптовано під конкретні потреби додатку.

Однією з головних переваг MVVM є покращення можливостей тестування компонентів. Завдяки тому, що ViewModel не залежить від View і взаємодіє тільки через зв'язування даних, розробники можуть тестувати бізнес-логіку без необхідності створення складних тестів GUI. Це значно спрощує процес розробки та забезпечує більш високу якість продукту.

Проте, шаблон MVVM не позбавлено недоліків. Основним з яких є проблема контролю станів в процесі виконання застосунку. Управління станом у шаблоні MVVM передбачає відповідальність шару ViewModel за збереження стану відображення за умови, що цей стан має точно представляти базову модель, проте синхронізація станів між ViewModel і View залишається нестабільною, що частково призведено двосторонньою прив’язкою даних.

Двостороння прив’язка даних є компромісом проєктування, що мав на меті надати полегшену взаємодію між інтерфейсом користувача і логікою застосунку, проте призвів до сильного порушення консистентності даних між

компонентами і ускладненням відслідковування життєвого циклу відображення.

Таким чином, MVVM повністю або частково вирішує всі проблеми проєктування, які мали його попередники, проте, його використання досі сумісне з серйозними складнощами не тільки в процесі розробки і підтримки застосунків, а й забезпеченні їх коректної роботи.

2.2.4 Загальне порівняння наведених архітектурних шаблонів

MVC, один з найстаріших шаблонів проєктування GUI, чітко розділяє роботу моделі, відображення користувацького інтерфейсу та логіку маршрутизації команд. Даний шаблон підходить для застосунків, де чітке розділення логіки дозволяє розробникам легко управляти та оновлювати окремі частини системи. Однак, велика залежність між View та Controller може ускладнити розробку та тестування.

MVP є ітерацією MVC, де Presenter замінює Controller, взаємодіючи безпосередньо з Model і управляючи всіма змінами у View. Це робить View більш пасивним та спрощує автоматизацію тестування, оскільки Presenter можна легко емулювати в тестовому середовищі. Незважаючи на це, MVP все ще може зіткнутися з проблемами при масштабуванні проєктів через складнощі в управлінні зв'язками між компонентами.

MVVM є прикладом найбільш вдалого вирішення проблеми проєктування програмних застосунків, він вирішує більшу частину проблем, що були властиві MVC та MVP, проте досі залишається недосконалим у аспектах відстеження потоку даних між модулями і контролі життєвого циклу відображення стану інтерфейсу.

В результаті аналізу, можна зробити висновок, що розробка програмного забезпечення вимагає вибору доцільного архітектурного шаблону. Шаблон проєктування для розробки програмних застосунків, що ґрунтуються на графічних користувацьких інтерфейсів потребує чіткого

розподілу власності на інформацію між кожним з модулів, мінімальний рівень з’язності між модулями і забезпечення синхронізації станів під час виконання.

2.3 Розробка шаблону проєктування програмних застосунків DCSM

Враховуючи пераховані недоліки вже існуючих MV-подібних архітектурних шаблонів маємо на меті вивести більш досконалих підхід проєктування програмних застосунків (рис. 2.6) [25, 26].

Drawer виконує безпосередню функцію відображення актуального стану, що був зібраний у модулі Service. Важливий аспектом є виділення логіки опрацювання подій, ініційованих користувачем, в модуль Controller. Для цього, після відображення наданого стану інтерфейсу, представлення надає розмітку зон взаємодії для контроллера.

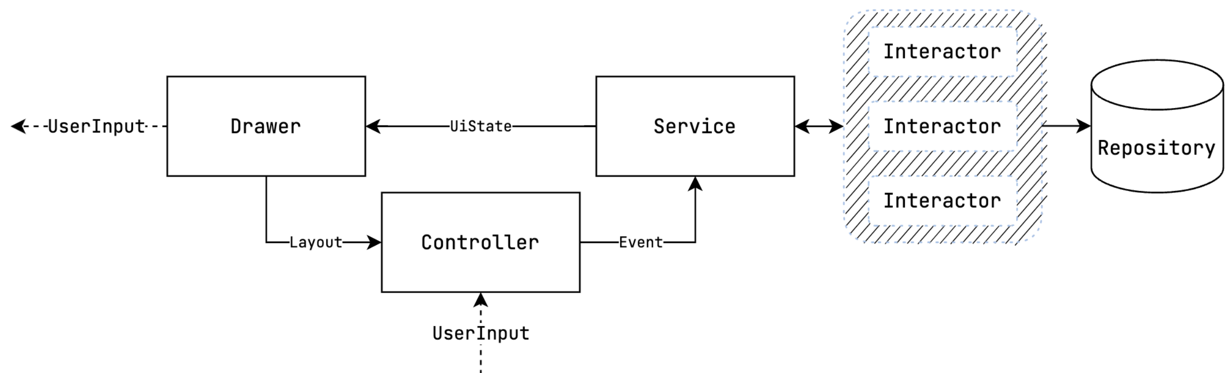

Рисунок 2.6 – Діаграма загального представлення шаблону DCSM

Controller виконує функцію опрацювання і маршрутизації користувацького вводу, для чого вводиться абстракція Подій, що буде визначати поведінку для конкретних типів користувацьких інтеракцій. Кожна подія має специфічний контракт, що є узгодженим для сервісного шару.

Service має на меті формування стану інтерфейсу, таким чином маємо специфічну реалізацію ViewModel, за відмінністю, що в даному випадку

мутація є глобальною, на весь View компонент, яким буде замінено старий. Для отримання даних з моделі, якими буде заповнено стану графічного інтерфейсу, використовується набір реалізацій Interactor, що будуть викликатись в залежності від необхідних даних з моделі, а при отриманні даних будуть зіставляти отримані з моделі дані, згідно узгодженого контракту з сервісом. Таким чином, ми зменшуємо зв'язність між сервісом і репозиторієм, що призведе до значного спрощення логіки сервісу, полегшить тестування й значно покращить загальну гнучкість системи.

Repository є узагальненим модулем, що залежить виключно від реалізації, основною метою якого є виключно отримання даних, ніяких маніпуляцій або мутацій даних на даному шарі відбуватись не повинно.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ І ТЕСТУВАННЯ

3.1 Розробка користувацького інтерфейсу програмного застосунку

Основною частиною програмного застосунку є редактор коду (рис. 3.1), тому моделювання і розробка користувацького інтерфейсу має починатись саме з нього. Таким чином, потрібно визначити:

– графічні компоненти;

– розмітку розміщення графічних компонентів;

– специфічні особливості розміщення графічних компонентів.

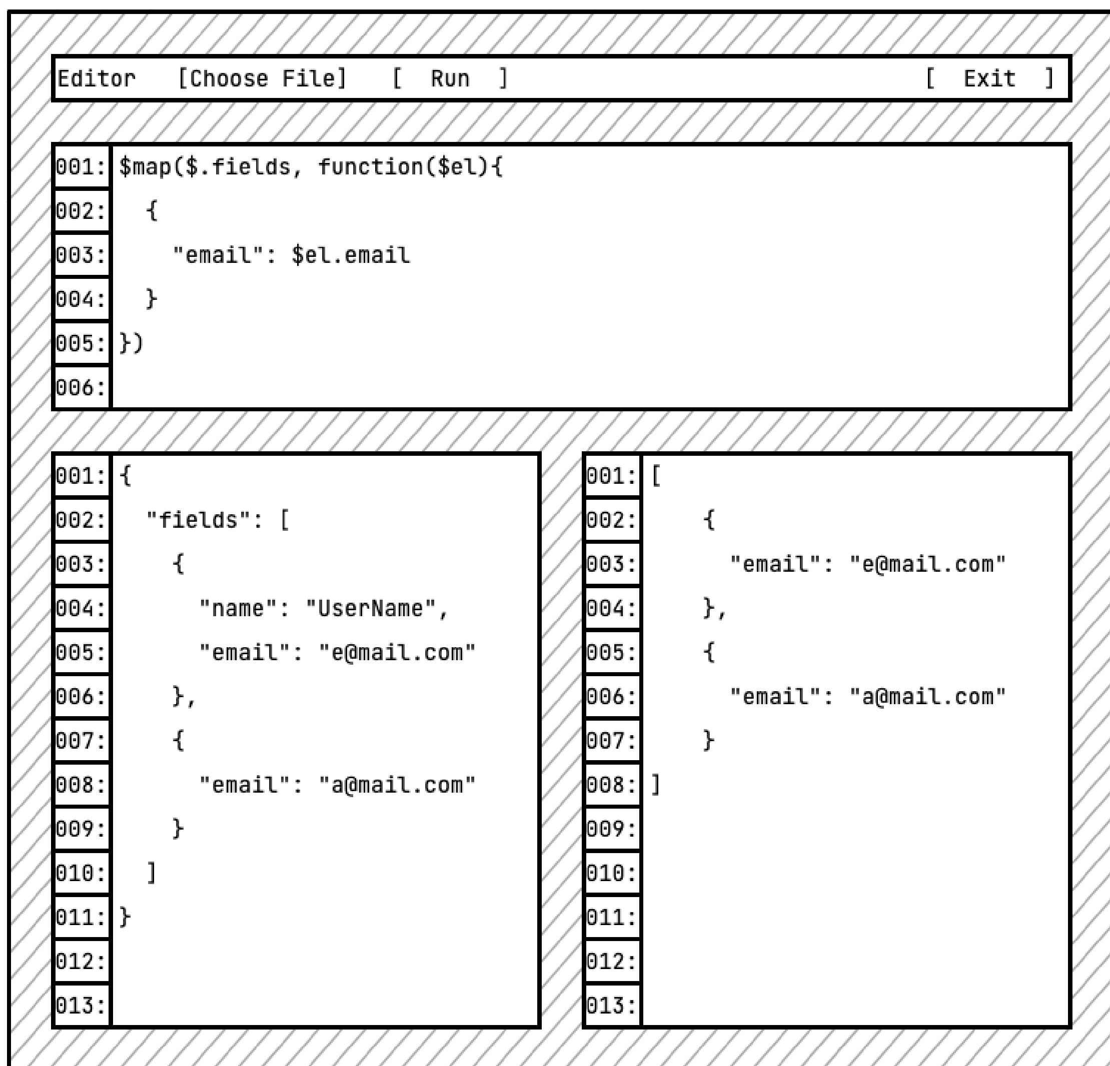

Рисунок 3.1 – Схема користувацького інтерфейсу для екрану редактора

Як представлено на схемі, головний екран, а саме редактор коду, має чотири основних елементи:

– навігаційну панель;

– багаторядкове поле вводу програмного коду;

– багаторядкове поле відображення вихідного файлу;

– багаторядкове поле відображення результату виконання програмного коду.

Кожен компонент має власну функцію, відповідно якої, він отримує конкретні додаткові властивості. Наприклад, ми маємо два елементи, що відповідають за відображення тексту. Даний текст представлено у форматі JSON, для якого характерно мати великий обсяг, але кожен елемент має невелику ширину. Отже, нам важливо, щоб ці два компоненти займали якнайбільше простору у вертикальній площині.

Окремо потрібно розглянути навігаційну панель (рис. 3.2), що відповідає не за одну, а одразу за декілька важливих функцій застосунку:

– однорядкове текстове поле;

– кнопку переходу на екран вибору вихідного файлу;

– кнопку виконання програмного коду;

– кнопку виходу з застосунку.

Навігація на даному екрані відбувається за допомогою трьох кнопок, з якими може взаємодіяти користувач. Для цього, ми використовуємо композицію декількох компонентів:

Рисунок 3.2 – Схема розмітки навігаційної панелі на складові компоненти

Наступним екраном застосунку є файловий провідник, який дозволить користувачу в довільному порядку обирати звідки редактор коду буде

отримувати вихідні дані, відносно яких буде виконуватись описаний програмний код (рис. 3.3).

Аналогічно до головного екрану, ми маємо наступні компоненти:

– навігаційна панель;

– вікно файлової системи.

На відміну від головного екрану, файловий провідник має не один складний компонент.

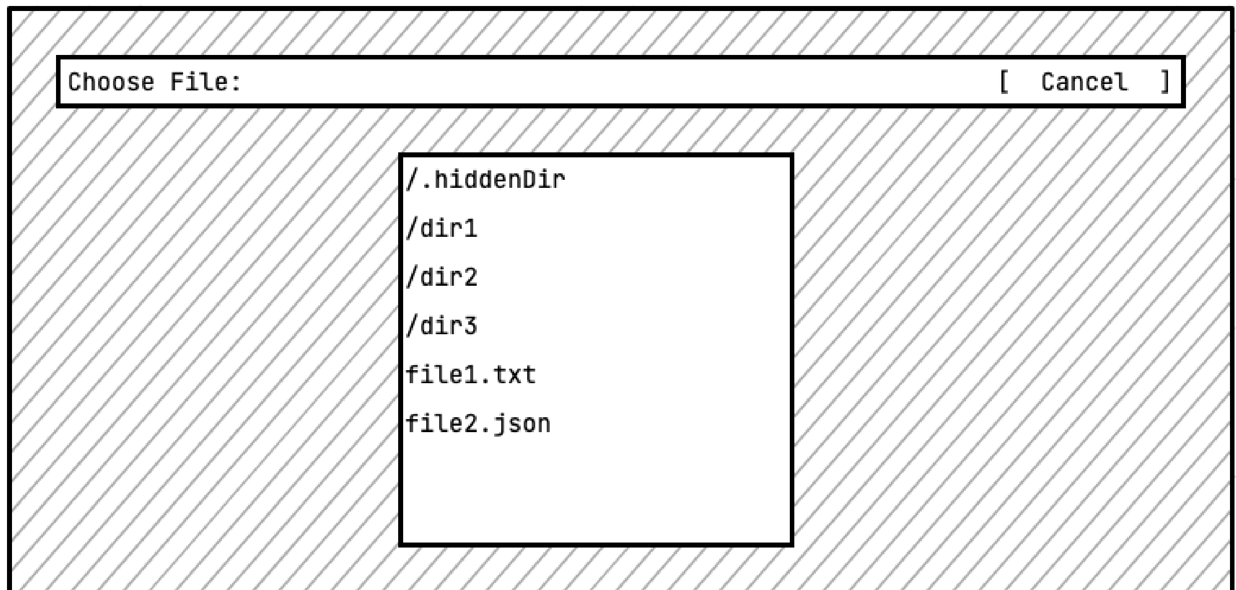

Рисунок 3.3 – Схема користувацького інтерфейсу для екрану файлового менеджера (у кореневій директорії)

Навігаційна панель (рис. 3.4) містить спільні риси з попередньою, за виключенням меншої кількості блоків керування.

Рисунок 3.4 – Схема розмітки навігаційної панелі на складові компоненти

Вона містить тільки одну кнопка, що забезпечує повернення до екрану редактора програмного коду (рис. 3.5).

Рисунок 3.5 – Схема розмітки вікна огляду файлової системи на складові компоненти

Важливо зазначити, поведінка компоненту може змінюватись в залежності від дій користувача, таким чином, порядок елементів компоненту файлового провідника може отримати додатковий пункт для переходу на попередній шар (рис. 3.6).

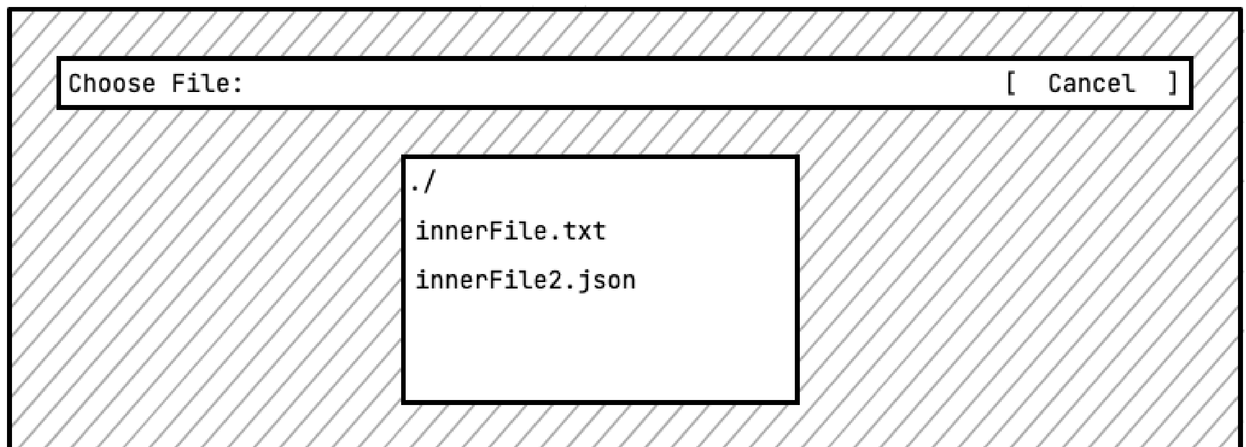

Рисунок 3.6 – Схема користувацького інтерфейсу для екрану файлового менеджера (у вкладеній директорії)

У такий спосіб, можна зробити висновок, що навіть використовуючи мінімальну кількість базових графічних компонентів можна побудувати

комплексний графічний інтерфейс, що буде задовольняти всі можливі потреби користувача.

3.2 Програмна реалізація шару взаємодії з даними

Основна частина логіки програмного застосунку зосереджена на двох процесах:

– взаємодія з файловою системою пристрою;

– виконання програмного коду.

Перш за все для опису і реалізації процесу взаємодії з файловою системою потрібно визначити якими сутностями вона маніпулює, а саме файли і директорії. Різниця між ними доволі суттєва, але в даному випадку, більшою частиною з них можна знехтувати, нас цікавить лише невелика низка з них, що дозволяє коректно відображати кожен запис із загального списку елементів; тому головна різниця між ними зводиться до значення булевого маркеру екземпляру (рис. 3.7) [27].

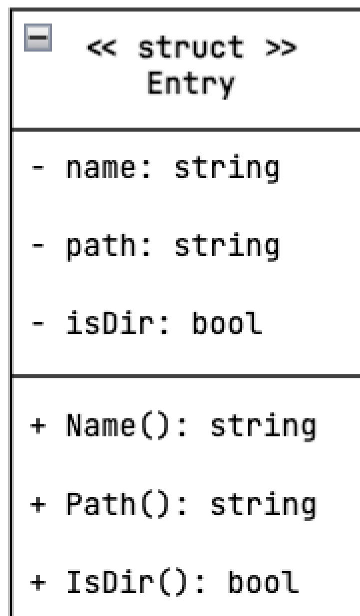

Рисунок 3.7 – Діаграма представлення сутності об’єкту файлової системи

На основі даної структури і буде описуватись поведінка одного з ключових репозиторіїв:

Репозиторій виконує функцію взаємодії із певним сховищем даних, яке може мати будь-який формат, від хмарних сховищ до, як в даному випадку, файлової системи пристрою, що надасть доступ до перегляду конкретної частини загального каталогу даних.

Важливою особливістю даної реалізації репозиторію є система каналів даних, що повертають інформацію до сервісного шару, який дозволяє незалежно від дій користувача оновлювати актуальний стан інтерфейсу. Наприклад, метод «Entries» (рис. 3.8) повертає канал, в який будуть повертатись всі записи на актуальному рівні вкладеності у провіднику, при оновленні даних у файловій системі пристрою, підтверджені зміни будуть відправлені в шину даних і далі перейдуть на рівень сервісу.

В свою чергу, це не залишається єдиним шляхом повернення даних, так само можна повернути звичайні примітивні або складні типи даних, метод «Content» (рис. 3.8).

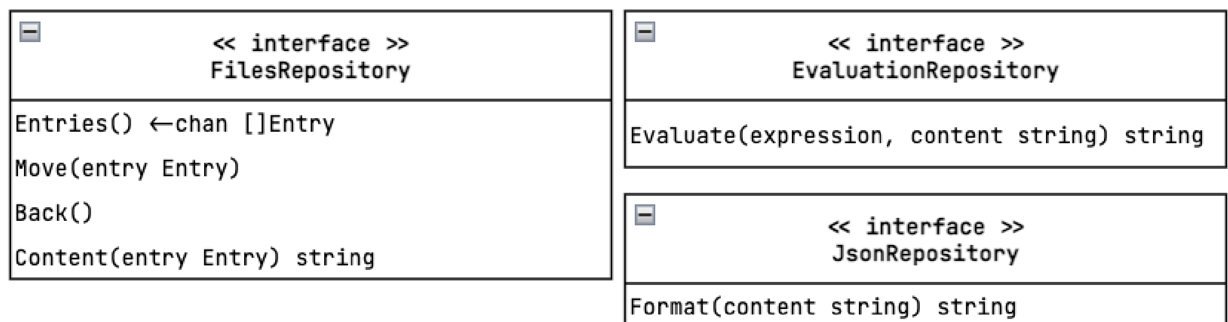

Рисунок 3.8 – Діаграма репозиторіїв програмного застосунку

Оглядово, можна зазначити, що «FilesRepository» відповідає за всю логіку взаємодії програмного застосунку з файловим каталогом операційної системи, надає вичерпний інтерфейс взаємодії для перегляду в глибину та пошуку потрібного запису. «JsonRepository» відповідає за всю логіку роботи з JSON об’єктами, а саме форматування вихідних даних, що далі будуть

відображені в інтерфейсі екрану редактора програмного коду. «EvaluationRepository» надає можливість виконувати обчислення і приведення програмного коду, що був введений користувачем, спираючись на опрацьовані вихідні дані.

Для взаємодії між сервісом і репозиторієм з’являється інша проміжна абстракція – інтерактор. Кожен інтерактор створюється під той набір методів, що надає репозиторій, таким чином ми не тільки повністю уникаємо зв’язності між компонентами архітектури, а й значно зменшуємо навантаження на кожен сервіс.

Можна сказати, що кожен інтерактор створюється під конкретну потребу бізнес-логіки, відповідно до доступних методів описаних в використаних репозиторіях (рис. 3.9), і по-суті являє собою обгортку для них.

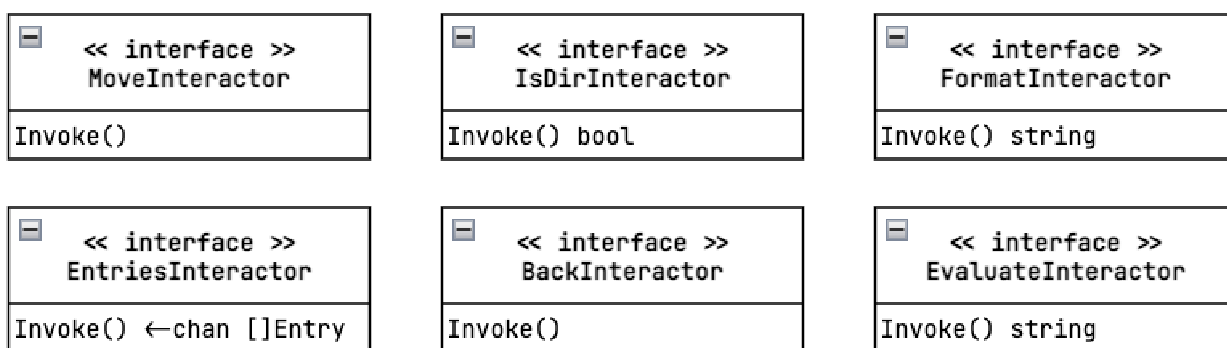

Рисунок 3.9 – Діаграма компонентів взаємодії між сервісом і сховищем даних

Ініціалізація моделі відбувається шляхом використання фабричного шаблону (рис. 3.10), в інтерфейсі якого описані методи створення екземплярів конкретних сутностей. В даному випадку цими сутностями є:

– репозиторій;

– інтерактор.

При створенні кожного інтерактора нам потрібно зазначити, якими сутностями він має оперувати, тобто за обробку яких даних він відповідає, це дозволить уникнути повторень і уніфікувати підхід виклику інтеракторів.

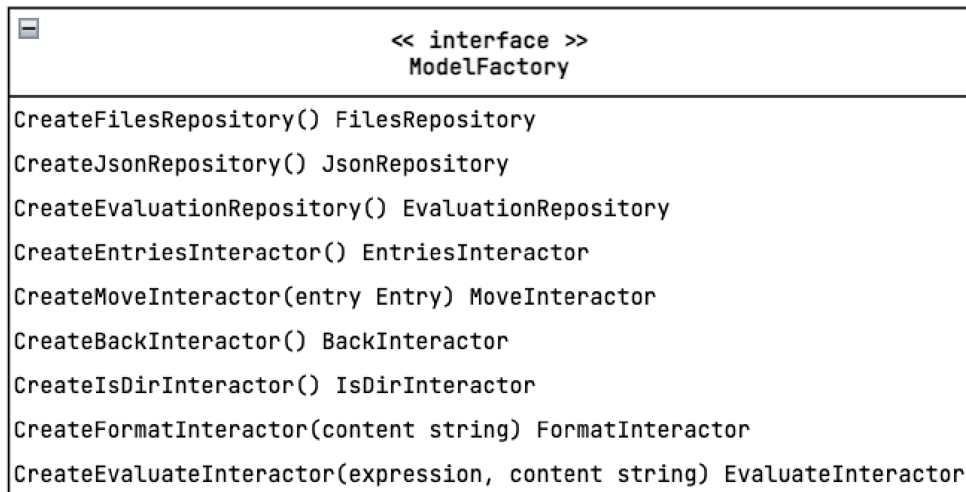

Рисунок 3.10 – Діаграма фабрики для ініціалізації моделі

Строк життя їх екземплярів можна вважати одним з найбільших, за виключенням малоймовірних змін під час виконання програми, кожен з екземплярів буде доступним весь період роботи застосунку.

3.3 Програмна реалізація алгоритмів і компонентів відображення

Відображення інформації є останнім етапом виконання будь-якої програми, що в першу чергу потребує моделювання способу її збереження. Потоком виводу інформації про актуальний стан інтерфейсу є термінальне вікно, тому саме на нього потрібно опиратися при виборі способу збереження проміжних даних. Простір екрану терміналу поділяється на:

$$n = h \times w \quad , \qquad (3.1)$$

де, n – загальна кількість комірок;

h – висота екрану;

w – ширина екрану.

Таким чином, необхідно описати сутність, що буде виконувати роль маски екрану вікна терміналу, в свою чергу, забезпечуючи можливість збереження інформації конкретного типу. Для цього було реалізовано сутність

Canvas (рис. 3.11), унікальний екземпляр даного типу буде жити доти, доки не буде змінено стан екрану терміналу, а саме змінено розмір, в такому випадку, Canvas буде створено заново з іншими вихідними даними і заповнено.

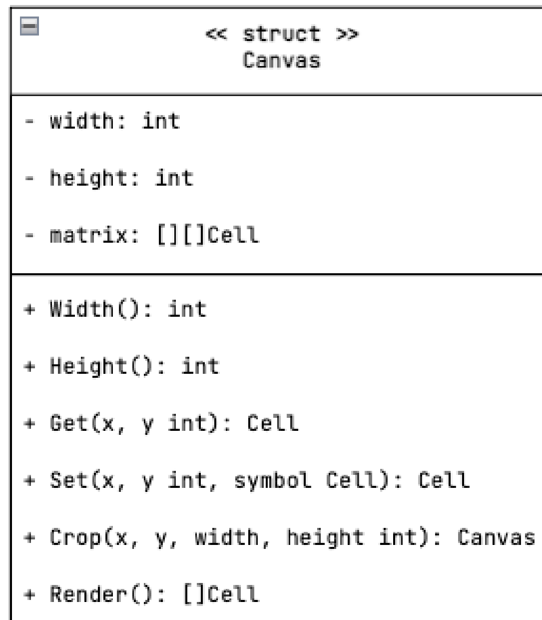

Рисунок 3.11 – Діаграма представлення сутності Canvas

Кожна комірка, займає фіксований розмір, що значно полегшує процес збереження, адже не потрібно закладати можливий динамічний розмір символу, що зберігається (рис. 3.12).

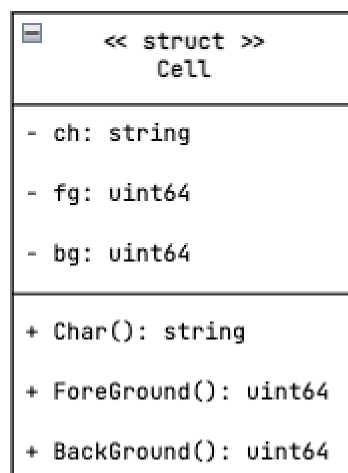

Рисунок 3.12 – Діаграма представлення сутності Cell

Потрібно враховувати те, що комірка є не тільки контейнером збереження самого символьного значення, але й його модифікаторів, – колір символьного значення, колір фону. Таким чином, бажано вивести окремий тип даних, що не тільки полегшить розуміння програмного коду стороннім розробником, а й значно розширить можливі сценарії модифікації поведінки.

Кожен графічний компонент повинен мати представлений набір методів, що має забезпечувати не тільки роботу логіки відображення, а й управління компонентом, його реакцію на дії користувача (рис. 3.13). Перед відображенням візуальний елемент проходить чіткий цикл, який завжди повинен йти в визначеному порядку і не повинен пропускати жодного кроку.

Першим етапом завжди йде виклик методу Measure, що поверне виміри конкретного компоненту, в залежності від його налаштувань. Другим етапом завжди йде Layout, що буде визначати розмітку позицій внутрішніх елементів (за наявності). Останнім етапом є заповнення виділеного полотна вмістом View елемента.

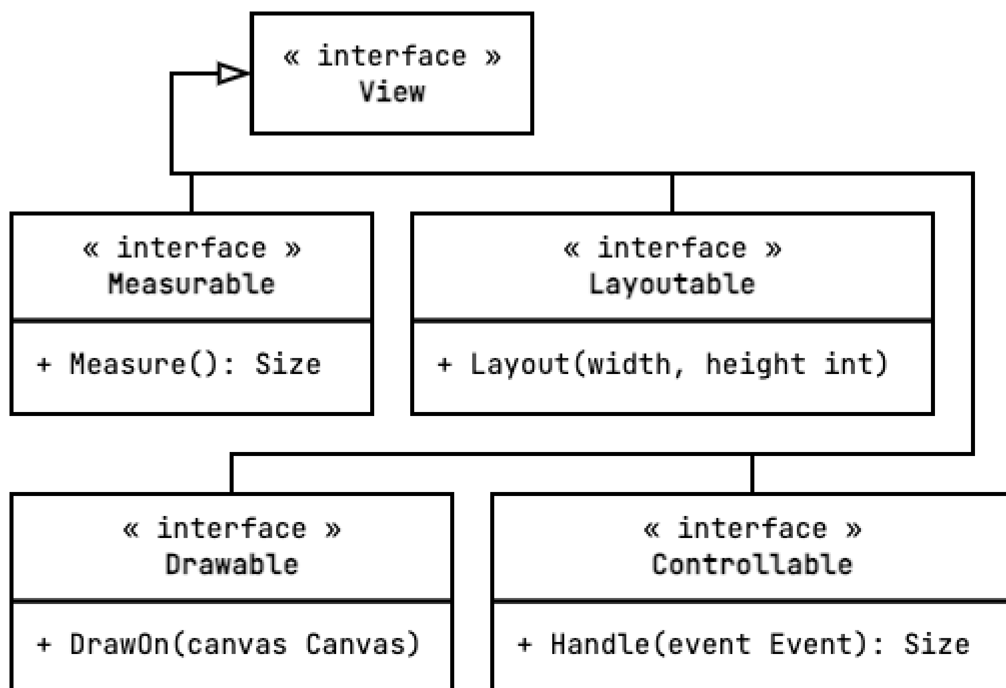

Рисунок 3.13 – Діаграма представлення View

3.4 Програмна реалізація процесу взаємодії користувача

Внутрішня реалізація View інкапсулює в собі логіку роботи трьох взаємопов'язаних компонентів, кожен з яких виконує відповідну частину роботи (рис. 3.14).

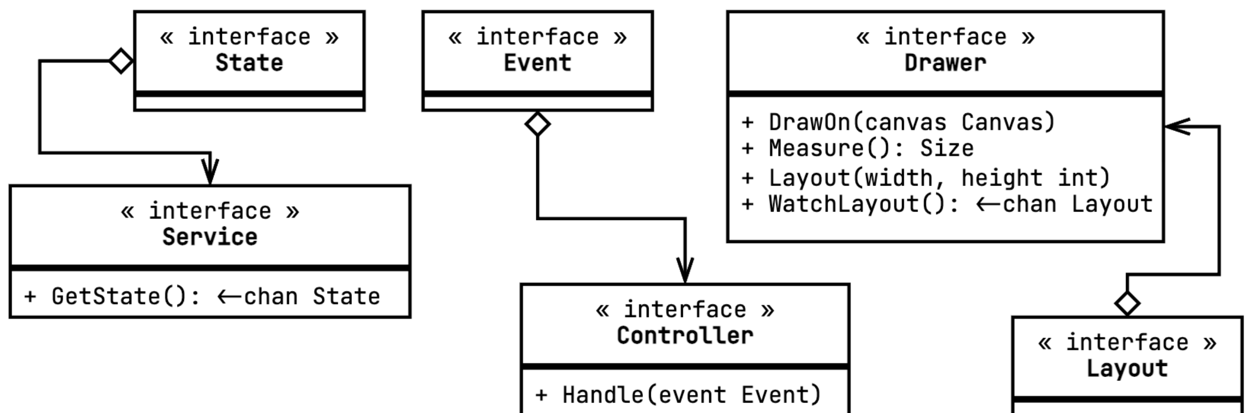

Рисунок 3.14 – Діаграма компонентів системи користувацької взаємодії

Даний шар архітектури оперує двома можливими варіантами подій, що надходять за таких умов:

– зміна стану репозиторію;

– дія користувача.

Формально, описати можливі зміни стану сховища можливо, але не доцільно через надто велику зв'язність з безпосередньою реалізацією, проте, можливі операції користувача є передбачуваними (рис. 3.15). Умовно, події пов'язані з командами клієнта можна мають спільне джерело походження – це певний зовнішній пристрій вводу. Різниця між подіями полягає лише в реалізації протоколу пакування даних операційною системою в залежності від пристрою. Таким чином, можна казати, що маршрутизація команд користувача відбувається за рахунок метаданих, які контролер отримує від операційної системи. В свою чергу, реакції і процес передачі цих даних залежить виключно від внутрішньої реалізації компонентів побудови.

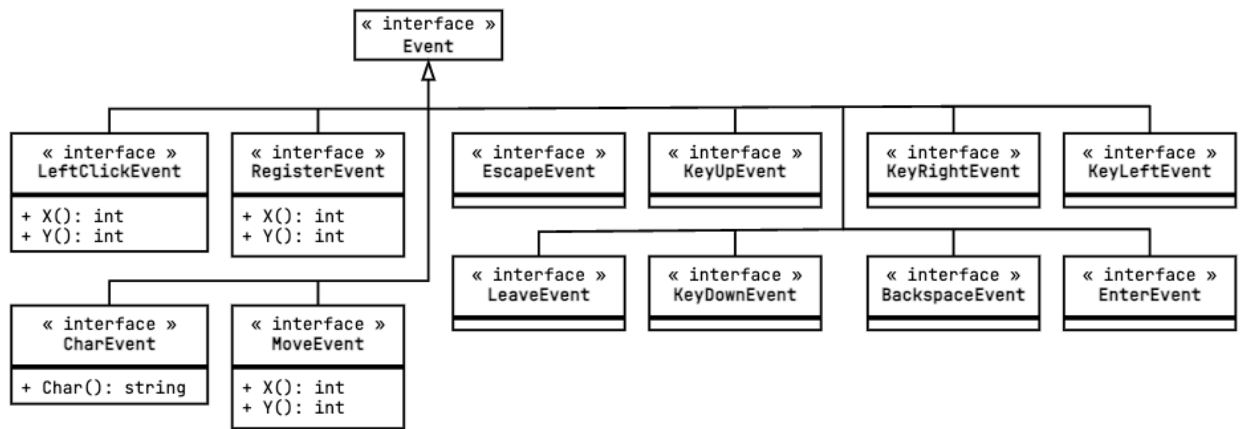

Рисунок 3.15 – Діаграма доступного списку користувацьких подій

Специфікою термінального середовища, є відсутність чіткого розмежування подій натискання і відтискання клавіш, що відповідають за введення літер формального алфавіту мови і спеціальних символів, по типу «Enter», «Escape» тощо. Таким чином, вимушеним кроком є категоризація сутностей подій не тільки за їх прямим призначенням, а й за класифікацією їх певного набору метаданих. Як приклад можна навести «CharEvent» і «EnterEvent» обидва є відображенням конкретної операції клавіатури, але лише одна з них потребує наявності додаткового атрибуту. Концепція події є критичною для подальшого розуміння принципу взаємодії основоположних компонентів.

Оскільки події можуть надходити з двох джерел, то і процес обробки і взаємодії відрізняється, проте не критично, у випадку коли подія пов’язана з діями клієнту, вона потрапляє до контроллеру, що буде викликати необхідний метод сервісу, що в свою чергу буде викликати необхідний набір інтеракторів, що будуть виконувати певний набір операцій мутації або отримання даних з репозиторію. Відповідно, якщо подія пов’язана з прямою зміною об’кту або їх групи в репозиторії, то описані кроки пропускаються і далій йде спільна поведінка для обох випадків, а саме послідовне повідомлення кожного компоненту.

Першим інформацію про зміни отримує сервіс, що в свою чергу викликає чергу операцій у шарі відображення, який в рекурсивному

алгоритмі перераховує виміри всіх вкладених сутностей, проводить нову розмітку доступного простору, повідомляє контроллер про зміни в розмітці і відображає оновлений стан графічного інтерфейсу (рис. 3.16).

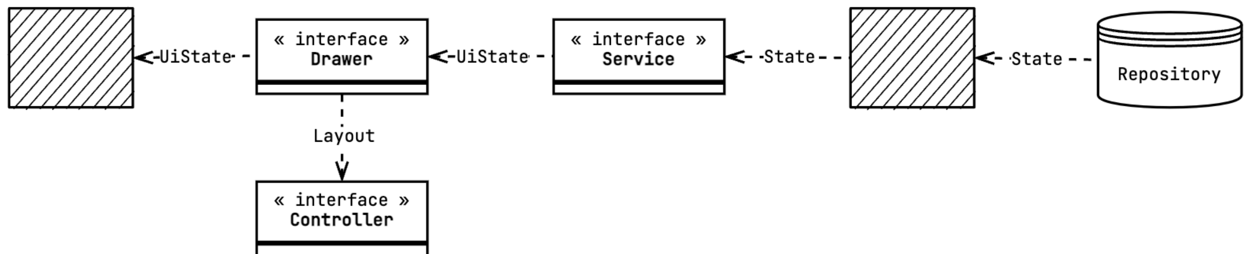

Рисунок 3.16 – Діаграма схеми оновлень компонентів системи відображення

3.5 Програмна реалізація примітивів базових компонентів інтерфейсу

Інформаційну систему будь-якої складності можна розібрати на складові компоненти, що її утворюють, за таким самим принципом відбуваються процеси і в області користувацьких інтерфейсів. Кожен екран програмного застосунку має специфічний набір компонентів, від примітивних, до складних, що можна порівняти з системою типів у мовах програмування. В яких для виконання задач вищої складності використовується об’єднаний тип даних, утворений композицією примітивних. Усі елементи побудови інтерфейсу поділяються на дві категорії, а саме на компоненти композиції і компоненти візуалізації (рис. 3.17).

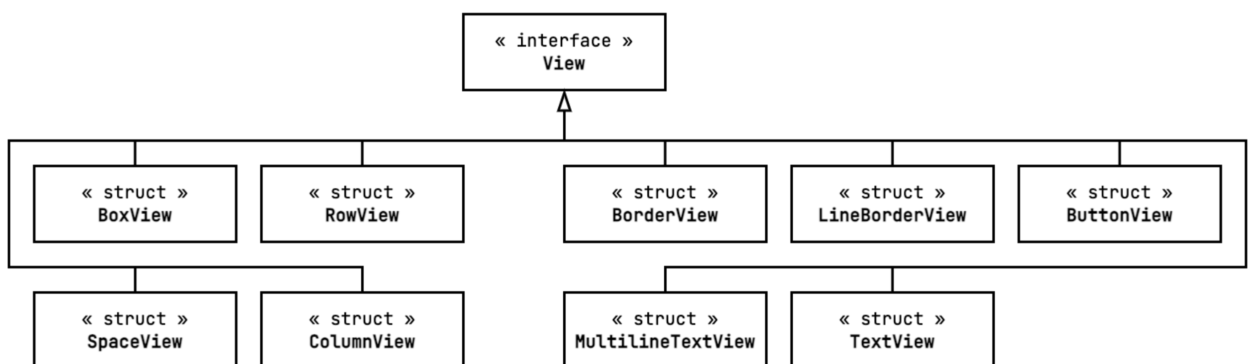

Рисунок 3.17 – Діаграма схеми усіх базових компонентів інтерфейсу

3.5.1 Компоненти візуалізації

Найпростішим елементом графічного інтерфейсу є текст (рис. 3.18). Текстове поле, що виконує виключно функцію відображення заданого однорядкового тексту.

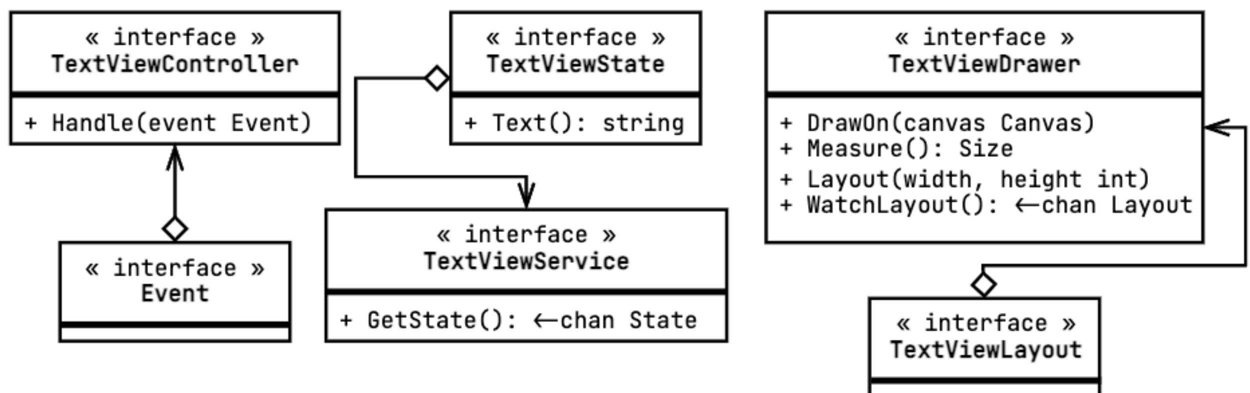

Рисунок 3.18 – Діаграма реалізації шаблону для компоненту TextView

Поведінка даного компоненту залежить від декількох факторів. Якщо обмеження ширини не були задані, тобот компонент може зайняти всю надану ширину, то текст буде виставлено по-центру рядку, все інше місце залишиться порожнім. Якщо обмеження знято не було, то у випадку, коли компонент запросив ширину меншу за довжину тексту, останні буде виведено тільки частину тексту до заданої ширини, при чому останні два символи будуть замінені на крапки, щоб явно показати, що текст було виведено не весь, показано на рисунку 3.19.

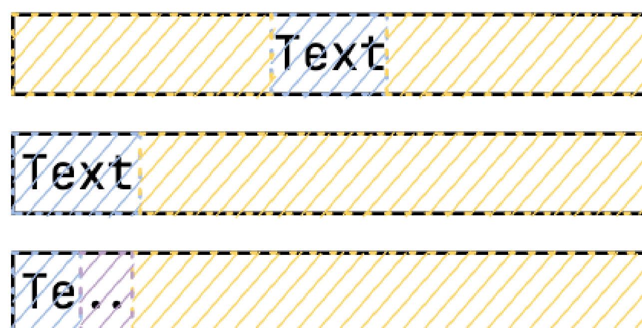

Рисунок 3.19 – Діаграма відображення поведінки компоненту

Через обмеженість можливого використання TextView, а саме відсутність можливості для користувача модифікувати заданий текст, що сильно звужує область можливого використання.

Задля вирішення даної проблеми було введено додатковий графічний елемент MultilineTextView (рис. 3.20).

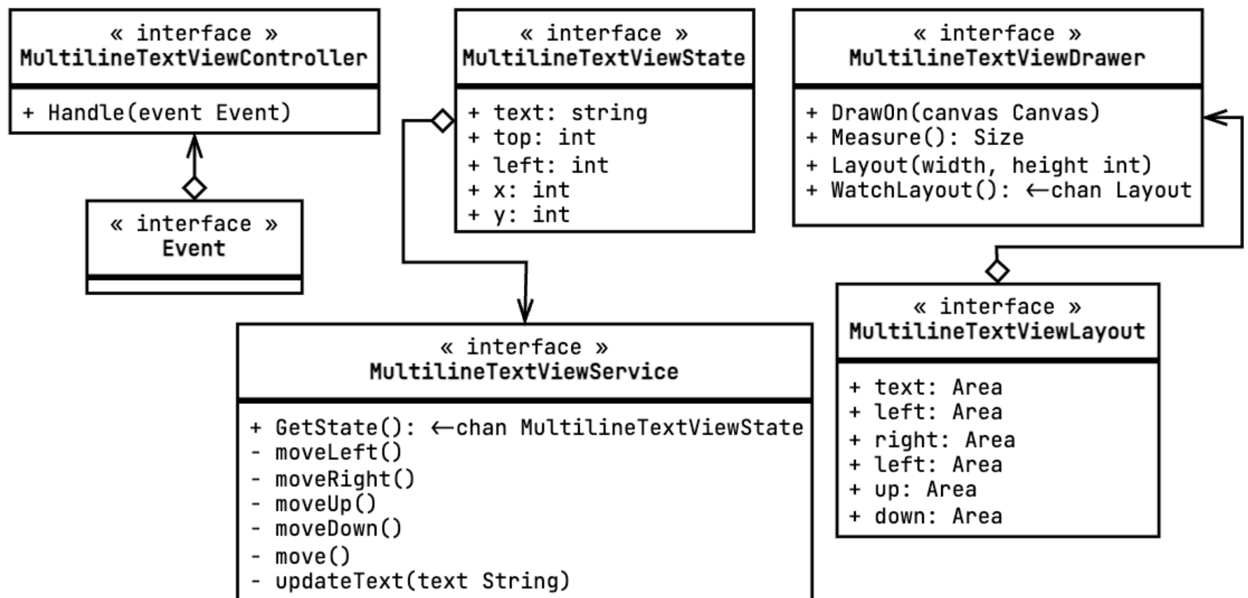

Рисунок 3.20 – Діаграма реалізації шаблону для компоненту MultilineTextView

Багаторядковий текстовий ввод є комплексним графічним компонентом, що вимагає реалізації декількох критичних аспектів, а саме: відслідковування позиції курсору, можливість його переміщення не тільки вздовж одного рядка, а і переходити між ними зі збереженням відносного положення, таким чином, щоб при переміщенні з довшого рядка курсор виставлявся відносно довжини нового. Положення курсору може змінюватись як посимвольно, за використання клавіатури, так і в межах усього компоненту з використанням миші [28].

Основною функцією є можливість вводу тексту, що має не тільки враховувати актуальне положення курсору, а й відносне відображення тексту,

тобто чи було задіяно опцію прокручування, – горизонтального або вертикального.

Чітке розмежування графічних компонентів на екрані є важливою складовою користувацького інтерфейсу, що дозволяє клієнту візуально відрізняти окремі елементи, наприклад звичайний текст і кнопку. Задля досягнення даної мети, було реалізовано два типи рамочних обгорток (рис. 3.21) для візуальних компонентів (рис. 3.22):

– однорядкові;

– багаторядкові.

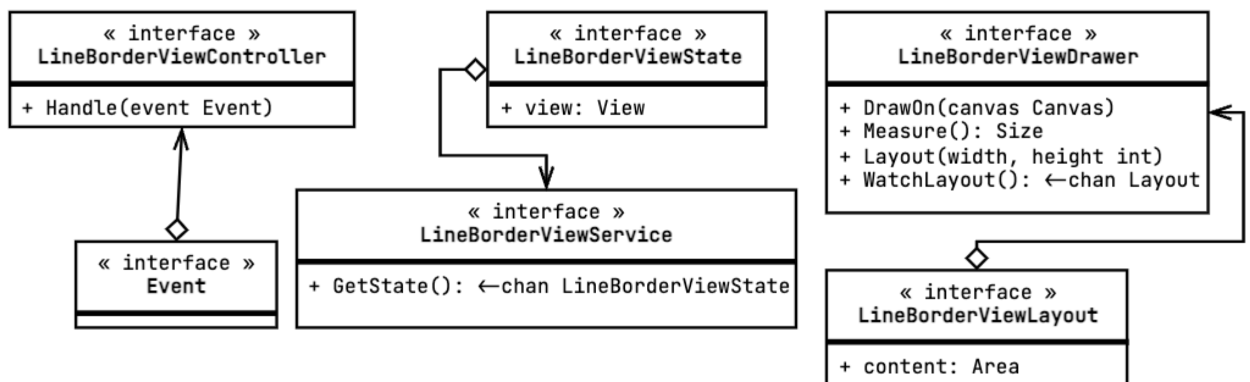

Рисунок 3.21 – Діаграма реалізації шаблону для однорядкового обгорткового компоненту

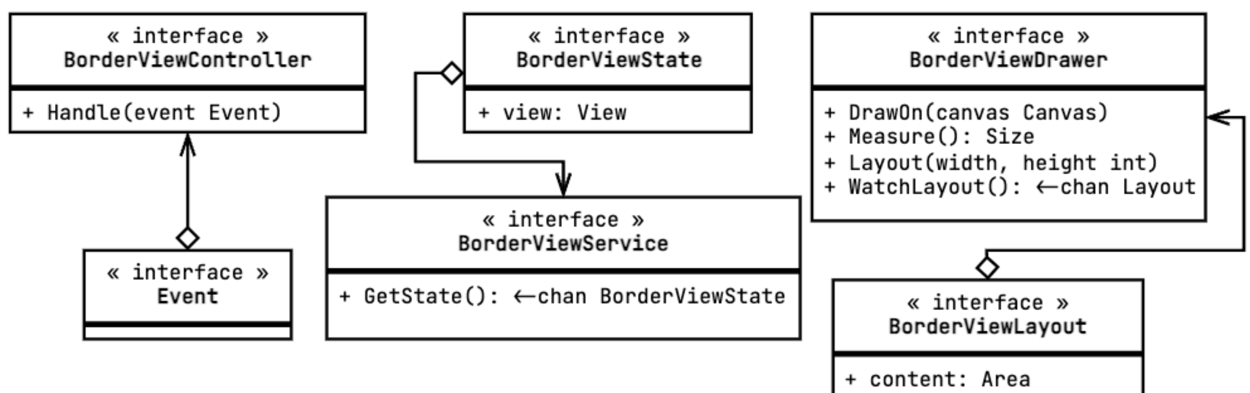

Рисунок 3.22 – Діаграма реалізації шаблону для багаторядкового обгорткового компоненту

Загальна поведінка обох компонентів прямолінійна – обгортка елементу

інтерфейсу у рамку, після вирахування вимірів внутрішнього View до них додаються власні виміри рамки, що в результаті й визначає простір, який буде зайнято. Важливо зазначити, що в рамку можна огорнути взагалі будь-який елемент, що значно вплине на гнучкість у побудові користувацького інтерфейсу.

3.5.2 Компоненти композиції

Компоненти композиції є ключовим фактором, що й впливає на фінальний вигляд будь-якого інтерфейсу, адже саме за допомогою них можливо правильно розставити кожен примітив відображення.

Основним елементом для композиції компонентів є тип BoxView (рис. 3.23), що виконує роль контейнера для одного вкладеного View, проте важливо зазначити, що View не обов'язково буде візуальним, це може бути ще один композиційний елемент, по типу RowView або ColumnView. Сам BoxView не тільки забезпечує механізм зберігання і доступу до внутрішніх компонентів, а й відповідає за масштабування, – прокрутку елементів, у випадку недостатнього розміру екрану для відображення.

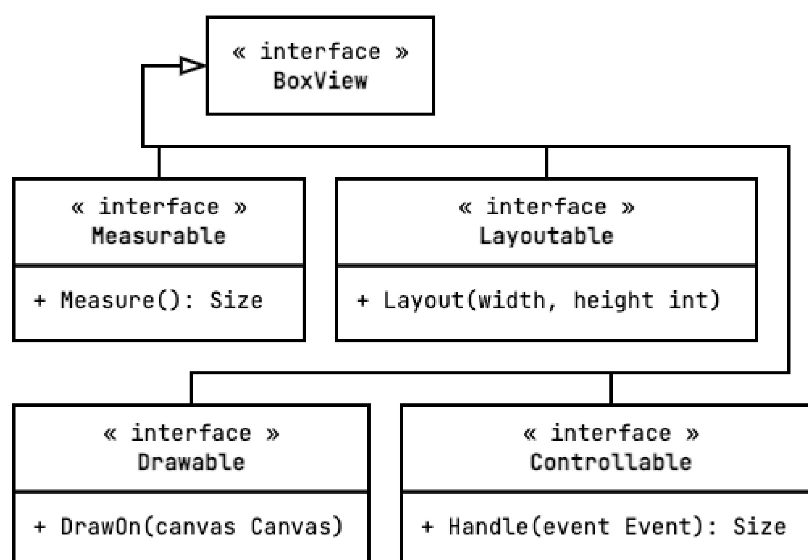

Рисунок 3.23 – Діаграма представлення компоненту BoxView

Наявність опції горизонтального і вертикального прокручування є дуже важливим фактором забезпечення гнучкості усієї системи, що буде гарантувати коректне відображення на віртуальному або фізичному екрані будь-якого розміру.

Важливим нюансом є умова появи елементів прокручування – а саме відповідні обмеження розміру батьківського екрану, тобто до моменту, поки користувацький графічний інтерфейс не почне виходити за визначені рамки, панель горизонтального і вертикального скролів будуть не активними, і відображатись можуть як разом, так і окремо, все залежить від параметрів екрану.

Якщо BoxView описує загальну наявність певного View у користувацькому інтерфейсі, то RowView (рис. 3.24) і ColumnView (рис. 3.25) відповідають за структуризацію компонентів в просторі.

Поведінка обох елементів композиції відбувається за схожим алгоритмом, тільки в різних площинах. RowView має на меті розміщувати елементи в рядок, організовувати відступи між ними, тощо.

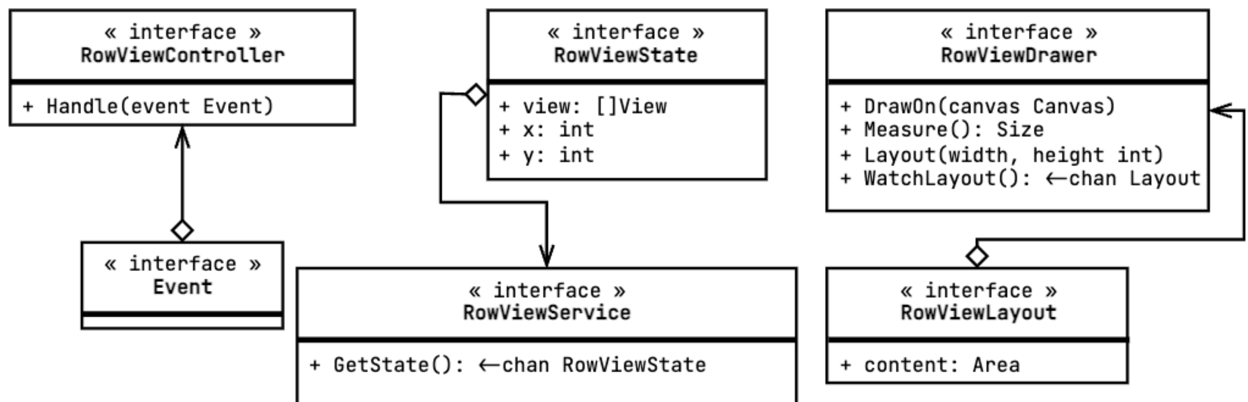

Рисунок 3.24 – Діаграма представлення компоненту RowView

Функціонування компонента залежить від наявності обмежень на розширення ширини внутрішніх об’єктів. Якщо RowView має дозвіл на розширення до максимально доступної ширини, то всі внутрішні View елементи будуть рівномірно розтягуватись в ширину.

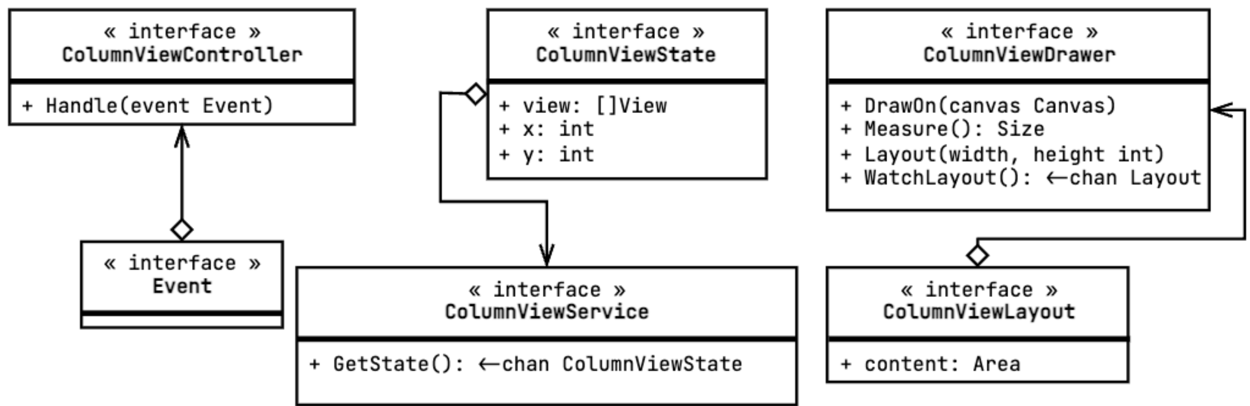

Рисунок 3.25 – Діаграма представлення компоненту ColumnView

Як зазначено, ColumnView виконує подібну функцію, але розміщення відбувається у вертикальному просторі. Поведінка компоненту, в свою чергу, модифікується у випадку відсутності обмеження на розширення в висоту. Таким чином, відповідно до рядку, кожен вкладений елемент колонки буде рівномірно збільшено у висоті.

Дані особливості вимірювання компонентів дозволяють полегшити процес заповнення елементами інтерфейсу усього вільного простору екрану.

Останнім елементом композиції є компонент відступів між елементами є SpaceView (рис. 3.26). Дана сутність забезпечує наявність певного порожнього простору між іншими компонентами.

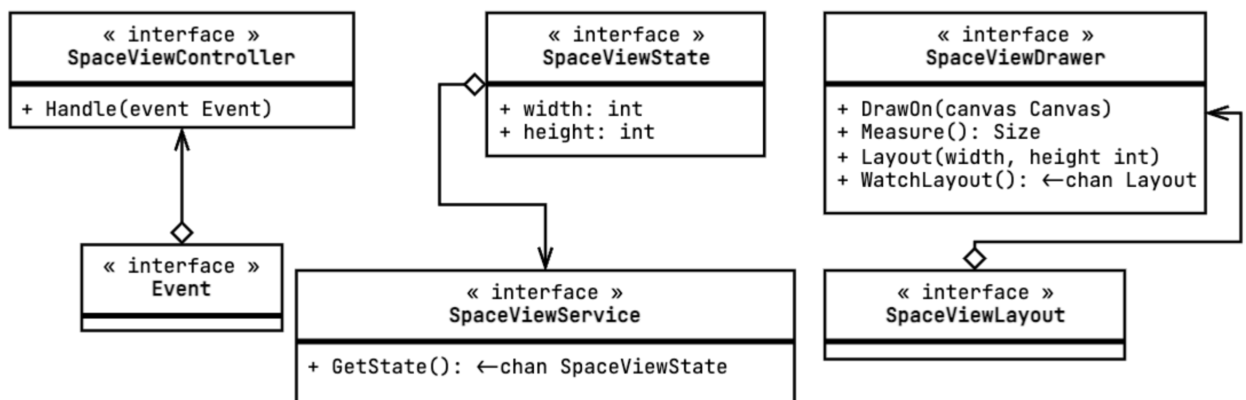

Рисунок 3.26 – Діаграма представлення компоненту SpaceView

Наприклад, якщо для екземпляру типу SpaceView зняти обмеження

розмірів, а саме зміна ширини, то можна розмістити другий компонент в правій або лівій частині екрану.

3.6 Тестування

Для підтвердження справедливості наведеної концепції побудови користувацьких текстових інтерфейсів і програмних застосунків, було розроблено редактор програмного коду мови трансформації і взаємодії з JSON об'єктами – JSONata.

Розроблений термінальний редактор (рис. 3.27) має не тільки безпосередню функцію виконання програмного коду, а й доступ до кореневої директорії, для пошуку файлу, що зберігає вихідні дані для опрацювання. Завантажений JSON об'єкт буде відображено на одній з багаторядкових текстових панелей [29, 30].

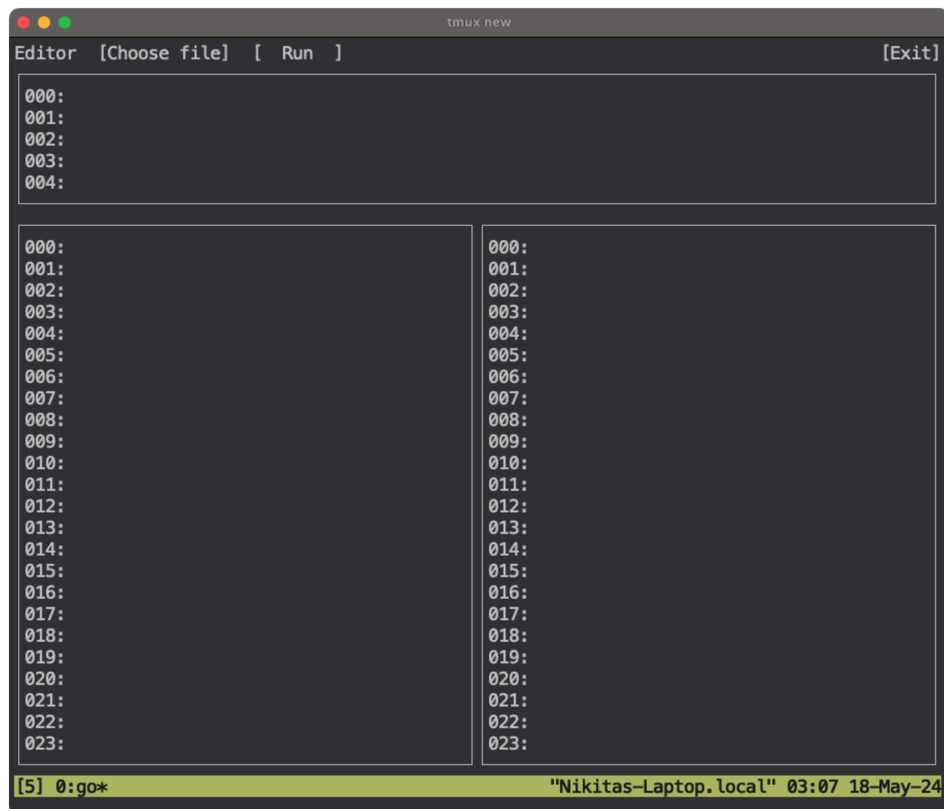

Рисунок 3.27 – Результат виконання програмного коду створення екрану текстового редактору

Лістинг 3.1 Приклад реалізації екрану текстового редактору:

```
editor := tui.NewMultilineTextView(expression, 5, 5, false, true)
view := tui.NewColumnView([]tui.View{
	tui.NewRowView([]tui.View{
		tui.NewTextView("Editor", 6, true),
		tui.NewSpaceView(" ", 2, 1, true, true),

tui.NewLineBorder(tui.NewButtonView(tui.NewTextView("Choose file", 11, true),
func() error {
			tui.Terminal.StartForResult(&FilesFrame{})
			return nil
		})),
		tui.NewSpaceView(" ", 2, 1, true, true),
		tui.NewLineBorder(tui.NewButtonView(tui.NewTextView("Run",
7, true), func() error {
			var data any
			err := json.Unmarshal([]byte(fileContent), &data)
			if err != nil {
				return err
			}
			e := jsonata.MustCompile(editor.String())
			res, err := e.Eval(data)
			if err != nil {
				tui.LoggerInstance.Write(res)
				return err
			}

			bytes, err := json.MarshalIndent(res, "", " ")
			if err != nil {
				tui.LoggerInstance.Write(string(bytes))
				return err
			}
			view := p.buildView(editor.String(), fileContent,
string(bytes))
			p.Update(view)
			return nil
		})),
```

tui.NewSpaceView(" ", 1, 1, false, true),
tui.NewLineBorder(tui.NewButtonView(tui.NewTextView("Exit", 4, true), func() error {
tui.Terminal.Exit()
return nil
})),
}, false, true),
tui.NewBorderView(editor, true),
tui.NewRowView([]tui.View{
tui.NewBorderView(tui.NewMultilineTextView(fileContent, 5, 5, false, false), true),

tui.NewBorderView(tui.NewMultilineTextView(expressionResult, 5, 5, false, false), true),
}, false, false),
})

Так само було розроблено екран файлового провіднику (рис. 3.28), що підтримує можливість переходу між директоріями (рис. 3.29).

Рисунок 3.28 – Результат виконання програмного коду створення екрану файлового провідника

Лістинг 3.2 Приклад реалізації екрану файлового провідника:

tui.NewColumnView([]tui.View{
tui.NewRowView([]tui.View{

```
                tui.NewTextView("Choose file:", 12, true),
                tui.NewSpaceView(" ", 2, 1, false, true),
tui.NewLineBorder(tui.NewButtonView(tui.NewTextView("Cancel", 6, true), func() error {
                        tui.Terminal.Exit()
                        return nil
                })),
            }, false, true),
            tui.NewRowView([]tui.View{
                tui.NewSpaceView(" ", 1, 1, false, true),
                tui.NewBoxView(
                        tui.NewBorderView(tui.NewColumnView(buttons), true),
                        1,
                        1,
                        false,
                        false,
                ),
                tui.NewSpaceView(" ", 1, 1, false, true),
            }, false, false),
        })
```

Рисунок 3.29 – Результат виконання програмного коду створення екрану файлового провідника при спробі перейти в іншу директорію

Розроблений підхід до опису елементів користувацького інтерфейсу дозволяє клієнту не тільки використовувати вже описані графічні елементи, а й визначати власні View об’єкти. На тривіальному прикладі вибору директорії, було справджено, що система обробки подій правильно опрацювали подію натискання кнопки миші і змінила стан екрану.

При виборі потрібного файлу, можна побачити, що екран файлового провідника автоматично закривається і програмний застосунок повертається на початкову сторінку, з єдиною зміною – одна з багаторядкових панелей тепер відображає JSON об’єкт (рис. 3.30).

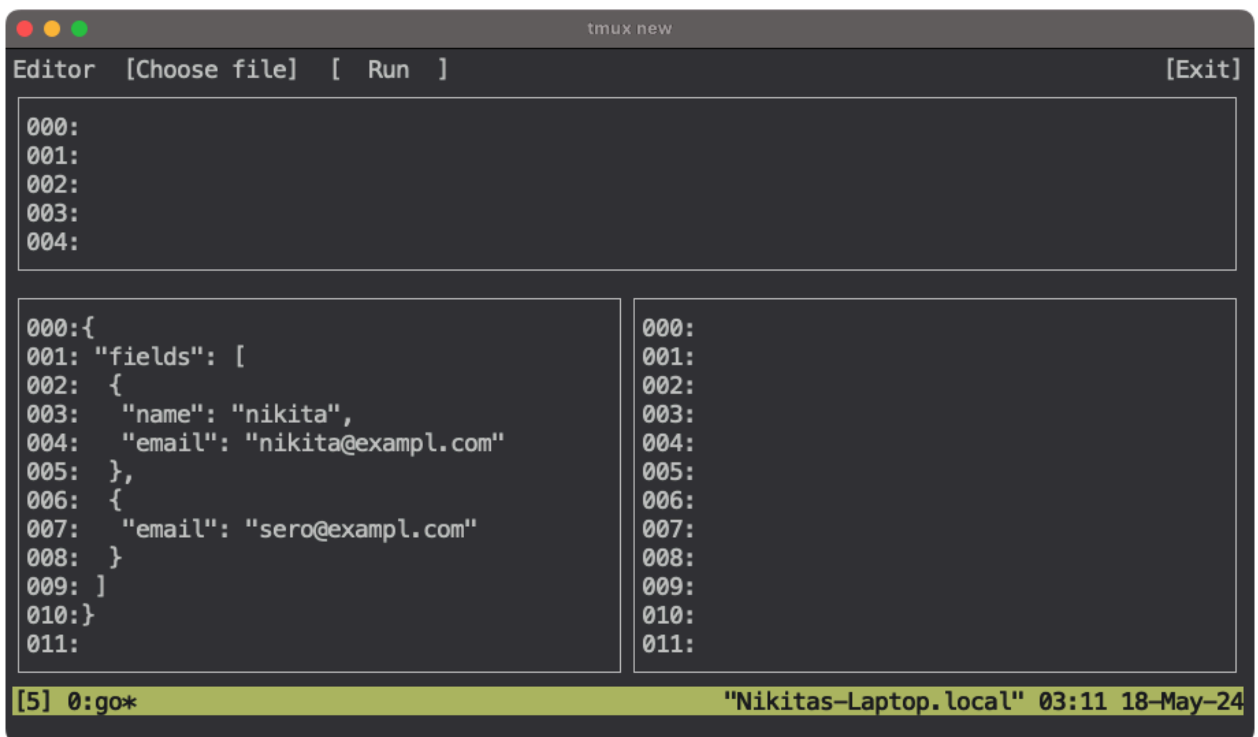

Рисунок 3.30 – Результат вибору вихідного файлу в файловому провіднику

Важливо, щоб програмний застосунок виконував поставлено задачу не тільки в ідеальних умовах. Можливим сценарієм роботи застосунку є недостатня кількість простору для повноцінного відображення користувацького інтерфейсу. В такому випадку, необхідно активувати елементи управління, а саме можливість посунути екран в горизонтальному і вертикальному напрямку (рис. 3.31).

```
Editor  [Choose file]  [  Run  ]                                    [Exit]
000:$map($.fields, function($el){
001: {
002:  "email": $el.email
003: }
004:})

000:{                                    000:[
001: "fields": [                         001: {
002:  {                                  002:  "email": "nikita@exampl.com"
003:   "name": "nikita",                 003: },
004:   "email": "nikita@exampl.com"      004: {
005:  },                                 005:  "email": "sero@exampl.com"
006:  {                                  006: }
007:   "email": "sero@exampl.com"        007:]
008:  }                                  008:
009: ]                                   009:
010:}                                    010:
011:                                     011:
[5] 0:go*                    "Nikitas-Laptop.local" 03:47 18-May-24
```

Рисунок 3.31 – Результат виконання програмного коду

На цьому етапі можна перевірити не тільки процес вводу програмного коду, а й результат його виконання (рис. 3.32).

```
Editor  [Choose file]  [
000:$map($.fields, func
001: {
002:  "email": $el.emai
003: }
004:})

000:{                000
001: "fields": [     001
002:  {              002D
                        R
[5] ocal" 03:48 18-May-24
```

Рисунок 3.32 – Результат відображення користувацького інтерфейсу при недостатньому просторі екрану

ВИСНОВКИ

В межах виконання кваліфікаційної роботи було проаналізовано предметну область і об'єкт роботи з огляду на загально прийняті напрацювання в областях інформатики та інженерії програмного забезпечення, а також пройдено ряд етапів для досягнення мети роботи:

– спроєктовано архітектурний шаблон побудови програмних застосунків;

– спроєктовано й реалізовано підхід роботи з графічними компонентами;

– спроєктовано й розроблено базові компоненти користувацьких інтерфейсів;

– спроєктовано й розроблено підхід взаємодії і опрацювання термінальних подій;

– програмно реалізовано програмний застосунок, на основі попередніх досягнень.

Результат кваліфікаційної роботи можна охарактеризувати як успішний, оскільки спроєктовані і реалізовані програмно підходи, алгоритми та інструменти, що дозволяютю значно розширити подальшу сферу використання текстових користувацьких інтерфейсів. Розроблений інструмент компонування інтерфейсів дозволяє використовувати інтуїтивний декларативний стиль користувачем, що дозволяє знизити не тільки ступінь входу для нових розробників, а й значно спростити загальний процес розробки і тестування.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Kernighan, B. W. & Pike R. (1983). The Unix Programming Environment. Prentice-Hall.

2. Stevens W., Rago S. (2013). Advanced Programming in the UNIX Environment. Addison-Wesley Professional.

3. Hughes J. & McGuire M. & Sklar D. & Foley J. & Feiner S. & Akeley K. (2013). Computer Graphics: Principles and Practice. Addison-Wesley Professional.

4. Tanenbaum S. A. & Woodhull A. (2006). Operating Systems Design and Implementation. Pearson.

5. Tanenbaum A. S. & Bos H. (2022). Modern Operating Systems. Pearson.

6. Rojas, R., & Hashagen, U. (Eds.). (2002). The first computers: History and architectures. MIT press.

7. Ceruzzi, P. E. (2003). A history of modern computing. MIT press.

8. Kerrisk M. (2010). The Linux Programming Interface: A Linux and UNIX System Programming Handbook. No Starch Press.

9. Moskala, M., & Wojda, I. (2017). Android Development with Kotlin. Packt Publishing Ltd.

10. Cheng, Y., & Domínguez, A. O. (2019). Advanced android app architecture: real-world app architecture in Kotlin 1.3. Razeware LLC.

11. Martin, R. C. (2009). Clean code: a handbook of agile software craftsmanship. Pearson Education.

12. Martin, R. C. (2017). Clean Architecture: A Craftsman's Guide to Software Structure and Design. Robert C. Martin.

13. McConnell S. (2004). Code Complete: A Practical Handbook of Software Construction. Microsoft Press.

14. Gookin D. (2007). Programmer's Guide to NCurses. Wiley Publishing, Inc.

15. Kinoshenko, D., Mashtalir, V., & Yegorova, E. (2006). Clustering method for fast content based image retrieval. In Computer Vision and Graphics: International Conference, ICCVG 2004, Warsaw, Poland, September 2004, Proceedings (pp. 946-952). Springer Netherlands.

16. Kinoshenko, D., Mashtalir, V., Shlyakhov, V., & Yegorova, E. (2012). nested Partitions Properties for spatial content Image retrieval. In Multimedia Storage and Retrieval Innovations for Digital Library Systems (pp. 240-269). IGI Global.

17. Kinoshenko, D., Mashtalir, V., Yegorova, E., & Vinarsky, V. (2005). Hierarchical partitions for content image retrieval from large-scale database. In Machine Learning and Data Mining in Pattern Recognition: 4th International Conference, MLDM 2005, Leipzig, Germany, July 9-11, 2005. Proceedings 4 (pp. 445-455). Springer Berlin Heidelberg.

18. Kinoshenko, D., Mashtalir, V., & Shlyakhov, V. (2007). A partition metric for clustering features analysis.

19. Bodyanskiy, Y., Kinoshenko, D., Mashtalir, S., & Mikhnova, O. (2012). On-line video segmentation using methods of fault detection in multidimensional time sequences. International Journal of Electronic Commerce Studies, 3(1), 1-20.

20. Chupikov, A., Kinoshenko, D., Mashtalir, V., & Shcherbinin, K. (2007). Image retrieval with segmentation-based query. In Adaptive Multimedia Retrieval: User, Context, and Feedback: 4th International Workshop, AMR 2006, Geneva, Switzerland, July 27-28, 2006, Revised Selected Papers 4 (pp. 207-221). Springer Berlin Heidelberg.

21. Kinoshenko, D., Mashtalir, V., Shlyakhov, V., & Yegorova, E. (2011). Metrical properties of nested partitions for image retrieval. In Machine Learning Techniques for Adaptive Multimedia Retrieval: Technologies Applications and Perspectives (pp. 18-49). IGI Global.

22. Abelson H., & Sussman G. J. (1996). Structure and interpretation of computer programs (p. 688). The MIT Press.

23. Cormen T. H. & Leiserson C. E. & Rivest R. L. & Stein C. (2022). Introduction to Algorithms, fourth edition. The MIT Press.

24. Fowler, M. (2002). Patterns of Enterprise Application Architecture. Addison-Wesley Professional.

25. Gamma E. & Helm R. & Johnson R. & Vlissides J. & Booch G. (). Robert Nystrom (2021). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional.

26. Fowler M. (2018). Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional.

27. Ousterhout J. (2021). A Philosophy of Software Design. Yaknyam Press

28. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1993). Design patterns: Abstraction and reuse of object-oriented design. In ECOOP'93—Object-Oriented Programming: 7th European Conference Kaiserslautern, Germany, July 26–30, 1993 Proceedings 7 (pp. 406-431). Springer Berlin Heidelberg.

29. Donovan A. & Kernighan B. (2015). Go Programming Language. Addison-Wesley Professional Computing.

30. Kennedy W. & Ketelsen B. & Martin E. & Thomas M. (2018) Go in Action. Manning Publications.