Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)
Кафедра програмної інженерії
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

бакалавр
(назва освітнього ступеня)

на тему: Розробка додатку для моніторингу продуктивності комп’ютера з використанням фреймворку Qt

Виконав: студент IV курсу, групи СП-42

спеціальності 121 Інженерія програмного забезпечення
(шифр і назва спеціальності)

_____ Семенюк О.Б.
(підпис) (прізвище та ініціали)

Керівник _____ Стоянов Ю.М.
(підпис) (прізвище та ініціали)

Нормоконтроль _____ Стоянов Ю.М.
(підпис) (прізвище та ініціали)

Завідувач кафедри _____ Петрик М.Р.
(підпис) (прізвище та ініціали)

Рецензент _____ _____
(підпис) (прізвище та ініціали)

Тернопіль
2024

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)

Кафедра програмної інженерії
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Петрик М. Р.
(підпис) (прізвище та ініціали)
« » 2024 р.

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня бакалавр
(назва освітнього ступеня)

за спеціальністю 121 Інженерія програмного забезпечення
(шифр і назва спеціальності)

студенту Семенюк Олександр Богданович
(прізвище, ім'я, по батькові)

1. Тема роботи Розробка додатку для моніторингу продуктивності комп'ютера з використанням фреймворку Qt.

Керівник роботи к.т.н., ст. викладач каф. ПІ Стоянов Ю.М.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від «15» квітня 2024 року № 4/7-342.

2. Термін подання студентом завершеної роботи

3. Вихідні дані до роботи

4. Зміст роботи (перелік питань, які потрібно розробити)

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Безпека життєдіяльності, основи охорони праці	Лазарюк В.В., к.т.н., доцент кафедри МТ		

7. Дата видачі завдання _____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1.	Ознайомлення з завданням до кваліфікаційної роботи		
2.	Підбір джерел інформації щодо створювання десктопних додатків на базі C++ і Qt		
3.	Переклад та опрацювання інформації з вибраних джерел		
4.	Розробка додатку на основі отриманої інформації		
5.	Оформлення звіту по виконаній роботі		
6.	Виконання завдання до підрозділу «Безпека життєдіяльності»		
7.	Виконання завдання до підрозділу «Основи охорони праці»		
8.	Підготовка презентації і доповіді		
9.	Нормоконтроль		
10.	Перевірка на плагіат		
11.	Попередній захист кваліфікаційної роботи		
12.	Захист кваліфікаційної роботи		

Студент _____ Семенюк О. Б.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Стоянов Ю. М.
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Розробка додатку для моніторингу продуктивності комп’ютера з використанням фреймворку Qt // Кваліфікаційна робота освітнього рівня «Бакалавр» // Семенюк Олександр Богданович // Тернопільський національний технічний університет імені Івана Пулюя, факультет комп’ютерно-інформаційних систем і програмної інженерії, кафедра прогограмної інженерії, група СП-42 // Тернопіль, 2024 // С. 69, рис. – 43, табл. – 0 , кресл. – 0, додат. – 2, бібліогр. – 9.

Ключові слова: C++, Qt, Qt Creator, Qt Designer, UML додаток, користувач, лістинг, діаграма.

В першому розділі кваліфікаційної роботи проаналізовано готові програмні рішення конкурентів, розглянуто актуальність розробки додатку для моніторингу продуктивності комп’ютера та технічний аспект проблеми .

В другому розділі кваліфікаційної роботи здійснено дослідження патернів програмування для додатку. Розробку загальної структури на основі модифікації обраного патерну та вигляду додатку, спроектовано архітектуру додатку та відображено її у вигляді UML діаграми.

В третьому розділі кваліфікаційної роботи реалізовано ключові класи додатку, проведено тестування додатку.

ANNOTATION

Development of an application for monitoring computer performance using the Qt framework // Qualification work of the educational level "Bachelor" // Semenyuk Oleksandr Bogdanovych // Ivan Pulyu Ternopil National Technical University, Faculty of Computer Information Systems and Software Engineering, Department of Software Engineering , group SP-42 // Ternopil, 2024 // C. 69, fig. – 43, tab. - 0, chair. - 0, add. – 2, bibliography - 9.

Keywords: C++, Qt, Qt Creator, Qt Designer, UML application, user, listing, diagram.

In the first section of the qualification work, ready-made software solutions of competitors were analyzed, the relevance of developing an application for monitoring computer performance and the technical aspect of the problem were considered.

In the second section of the qualification work, a study of programming patterns for the application was carried out. The development of the general structure based on the modification of the selected pattern and the appearance of the application, the architecture of the application was designed and displayed in the form of a UML diagram.

In the third section of the qualification work, the key classes of the application were implemented, the application was tested.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ПК – персональний комп’ютер.

ЦП – центральний процесор.

ОЗП – оперативна запам'ятовуюча пам'ять.

ГП – графічний процесор.

C++ – мова програмування загального призначення з підтримкою кількох парадигм програмування: об'єктно-орієнтованої, узагальненої, процедурної та ін.

Qt – фреймворк для створення додатків, написаний на мові C++.

Qt Creator – інтегроване середовище розробки для створення застосунків на основі Qt.

Qt Designer – інструмент для розробки графічного інтерфейсу користувача з використанням фреймворку Qt.

UML – мова візуального представлення ідей розробника.

API (Application Programming Interface) – інтерфейс програмування додатків, програмний інтерфейс програми.

IDE (Integrated Development Environment) – інтегроване середовище розробки програмного забезпечення.

GUI (Graphical User Interface) – графічний інтерфейс користувача.

Патерн – шаблон проектування, рішення загальної проблеми у проектуванні програмного забезпечення.

ЗМІСТ

ВСТУП

У сучасному світі комп'ютери відіграють ключову роль у практично кожній сфері людської діяльності, від підприємництва до науки і освіти. Від їхньої продуктивності напряму залежить ефективність роботи користувачів та успішність виконання завдань. Тому критично важливо мати засоби для моніторингу та аналізу роботи комп'ютерних систем з метою своєчасного виявлення потенційних проблем і підвищення їхньої надійності.

Програми для моніторингу продуктивності комп'ютера застосовуються у широкому спектрі сфер, включаючи системне адміністрування, розробку програмного забезпечення, тестування ПЗ та моніторинг серверів. Вони надають можливість стежити за роботою ключових компонентів комп'ютера, таких як система охолодження та використання ресурсів, тим самим забезпечуючи оптимальну продуктивність і зменшуючи ймовірність виникнення неполадок.

Розробка програмного забезпечення для моніторингу продуктивності вимагає використання потужних інструментів і технологій. У даній бакалаврській роботі пропонується розробка додатку з інтуїтивно зрозумілим інтерфейсом, що дозволить користувачам ефективно контролювати та оптимізувати роботу їхніх комп'ютерів. Використання мови програмування C++ сприятиме підвищенню продуктивності додатку, завдяки її швидкодії та підтримці паралельного програмування. Фреймворк Qt, у свою чергу, забезпечить швидку розробку графічного інтерфейсу та забезпечить його працездатність на різних платформах.

Очікується, що розроблений проект значно полегшить процес моніторингу та управління продуктивністю комп'ютерів для різних категорій користувачів, що сприятиме підвищенню їхньої ефективності та зниженню часу на виявлення та вирішення проблем.

Розробка такого додатку має величезний потенціал у полегшенні життя користувачів і підвищенні ефективності їхньої роботи. Працюючи над проектом,

планується не лише створення інструменту для моніторингу, але й розширення його функціоналу для включення додаткових аналітичних можливостей. Наприклад, аналізу пам'яті та обчислювальних потужностей для оптимізації ресурсів і підвищення продуктивності.

Платформа Linux стала однією з найбільш популярних операційних систем серед тих, хто цінує стабільність, безпеку та відкритість коду. Її розвиток і поширення в останні роки свідчать про те, що Linux визнаний як надійна основа для різноманітних комп'ютерних систем, від вбудованих пристроїв до серверів великих дата-центрів

Розробка цього додатку відповідає попиту на інструменти для операційної системи Linux, яка продовжує набирати популярність. Основною метою проекту є створення інтуїтивно зрозумілого інструменту для моніторингу та управління продуктивністю комп'ютерів, що працюють під цією платформою. Використання мови програмування C++ та фреймворку Qt забезпечить не лише високу продуктивність додатку, але й його ефективність в управлінні ресурсами, що є критичним для користувачів Linux, які цінують ефективне використання системних ресурсів.

Особливу увагу слід приділити необхідності такого додатку для нових користувачів Linux. Цей інструмент стане незамінним помічником для тих, хто тільки починає ознайомлення з операційною системою Linux, допомагаючи їм ефективно використовувати потенціал своїх комп'ютерів і знижувати поріг входу у складні технологічні процеси. Таким чином, розробка цього додатку не лише відповідає сучасним вимогам інформаційних технологій, але й сприятиме подальшому зростанню популярності Linux як операційної системи, доступної для широкого кола користувачів у різних сферах життя.

РОЗДІЛ 1. ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧ ДЛЯ РОЗРОБКИ ДОДАТКУ «SympleSysMonitor»

1.1 Огляд конкурентів

При розробці програмного забезпечення важливо перевірити наявність аналогічних рішень на ринку. Проведення аналізу конкурентів дозволяє визначити їхні слабкі та сильні сторони, що допоможе внести покращення у власний проєкт. Це, у свою чергу, підвищить його привабливість для користувачів.

На ринку існують продукти, які виконують цю задачу. Найбільш відомою програмою є AIDA64. Розробником цього програмного забезпечення є компанія FinalWire, заснована у 2010 році. Вони відомі своїми програмами для моніторингу ресурсів комп'ютерів, такими як AIDA32, AIDA64, Everest та їхні модифіковані версії. FinalWire співпрацює з багатьма відомими компаніями, такими як AMD, Asus, Intel та іншими. Найпопулярнішим їхнім продуктом є AIDA64, яка навіть стала частиною їхнього логотипу (рис. 1.1) [1].

Рис. 1.1 – Логотип компанії конкурента FinalWire Ltd

AIDA64 є зручним і зрозумілим застосунком для просунутих користувачів. Програма містить багато інструментів, що дозволяють отримати всю інформацію про комп'ютер та його компоненти, а також здійснювати стрес-тестування.

Для покращення роботи з великою кількістю параметрів показників комп'ютера, AIDA64 має графічний інтерфейс (рис. 1.2). Графічний інтерфейс надає зручний доступ до показників комп'ютера, розділяючи їх по категоріях відповідно до особливостей параметрів, залежно від того, чи вони стосуються

окремих фізичних компонентів комп'ютера або певного програмного забезпечення [1].

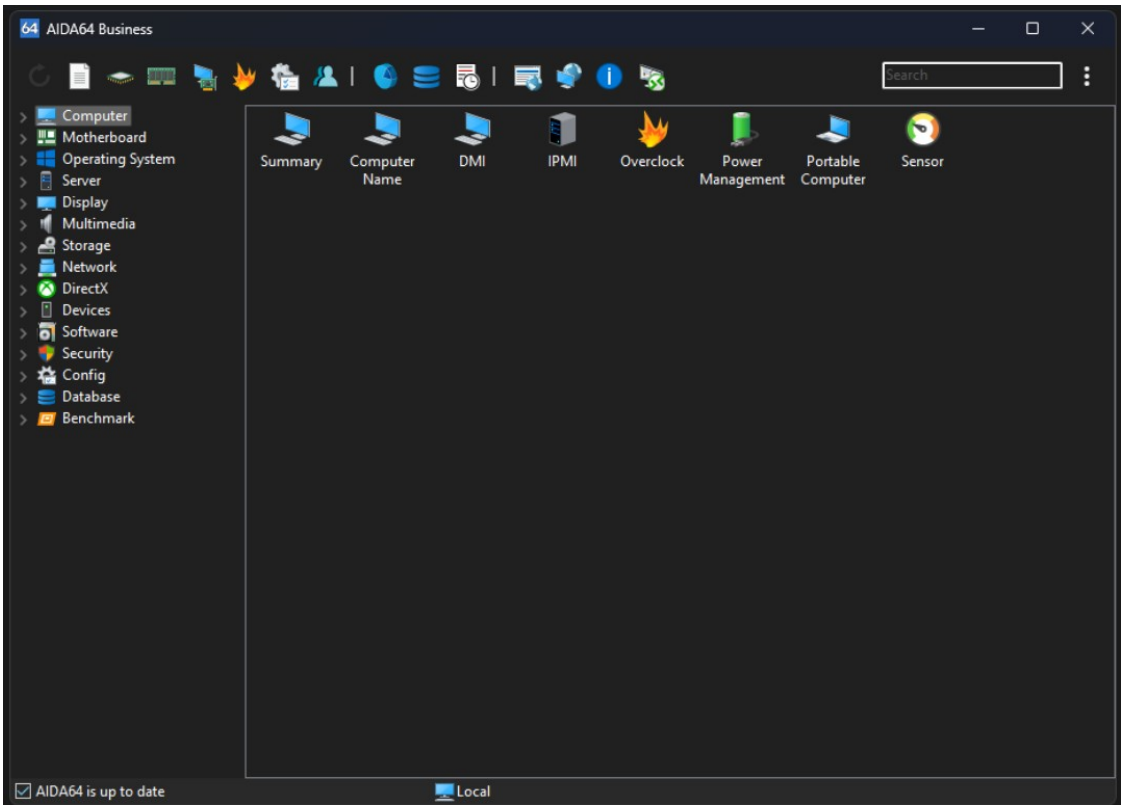

Рис. 1.2 – Інтерфейс програми AIDA64

Дана програма є платною, але надає безкоштовний пробний період. Проте для звичайних користувачів вона може містити забагато зайвої інформації, яку вони не використовують через нерозуміння або відсутність потреби в ній. Крім того, користувачі деяких дистрибутивів операційної системи Linux можуть зіткнутися з проблемами під час встановлення цієї програми. Найчастіше з цією проблемою стикаються користувачі, які є початківцями в освоєнні системи Linux.

1.2 Обґрунтування вибору напрямку дослідження

При переході з ОС Windows на ОС Ubuntu я зіткнувся з відсутністю зручних стандартних програм для моніторингу продуктивності компонентів комп'ютера. Це викликало інтерес до пошуку застосунків для вирішення цієї проблеми. Для ОС Ubuntu існує обмежена кількість програм для моніторингу продуктивності компонентів комп'ютера.

Одним із таких додатків є AIDA64, який є одним із небагатьох додатків із графічним інтерфейсом. Проте встановлення AIDA64 викликало складнощі, і в результаті програму не вдалося завершити встановлення. Інші програми з подібними функціями мають інтерфейс, який надмірно навантажений великою кількістю параметрів. Тому характерною особливістю усіх цих програм є складний інтерфейс для звичайних користувачів та іноді складність при встановленні.

У зв'язку з цим я обрав цей напрямок для створення простої та зручної програми для моніторингу продуктивності компонентів комп'ютера. Спочатку додаток розроблятиметься для платформи та операційної системи Ubuntu, але під час розробки буде передбачено можливість розширення додатку для інших платформ. Архітектура додатку буде розроблена так, щоб можна було легко скомпілювати програму з використанням різних ресурсних файлів та функцій для різних операційних систем.

Після аналізу додатків конкурентів вирішено спочатку реалізувати мінімальну кількість функцій, які показуватимуть найважливіші параметри продуктивності комп'ютера.

1.3 Технічний аспект проблеми

З урахуванням цих даних було обрано мову програмування C++ (рис. 1.3), щоб забезпечити високу ефективність додатку та зменшити споживання ресурсів. C++ — це мультипарадигмова мова програмування, яка дозволяє використовувати об'єктно-орієнтований та процедурний підходи при створенні додатків. Завдяки цьому можна полегшити створення архітектури проєкту [2].

Стандартна шаблонна бібліотека містить великий набір інструментів. Найважливішими з них при виконанні цього проєкту є бібліотеки для роботи з файлами. З їхньою допомогою функції програми зможуть звертатися до системних файлів для отримання інформації про різні параметри комп'ютера [2].

Рис. 1.3 – Логотип логотип мови програмування C++

Вибір мови програмування C++ дозволяє використовувати код, написаний на мові C, оскільки ці мови є сумісними і практично не мають значних проблем із переходом на нову версію. Спільноти розробників, які використовують ці мови, продовж багатьох років вже створили багато бібліотек та рішень. Це дозволяє повторно використовувати функції та бібліотеки з різних актуальних ресурсів для створення і покращення додатку. Якщо все ж виникають проблеми у зв’язку з різними версіями мов програмування, то зазвичай зміна коду не потребуватиме багато часу [2].

Додатковим інструментом було обрано фреймворк Qt (рис. 1.4), оскільки він є зручним і ефективним інструментом для створення графічних інтерфейсів. Qt надає велику кількість вбудованих інструментів, шаблонів та готових рішень, що робить його дуже зручним для розробників [3].

Рис. 1.4 – Логотип логотип фреймворку Qt

Особливістю фреймворку Qt є можливість розробки для трьох мов програмування: Python, C та C++. У випадку несумісності коду C та C++ можна створити окрему підпрограму з використанням C. Завдяки регулярним оновленням та виправленням помилок цей фреймворк залишається актуальним та продовжує набирати популярність. Для забезпечення швидкодії та мінімальних затрат ресурсів комп’ютера буде використовуватись лише C++ та C [3].

Існують великі спільноти розробників, які працюють з C, C++ та Qt, і можуть допомогти початківцям ознайомитися з цими технологіями. Є велика кількість курсів та документацій для кожного класу. Також у цих спільнотах вже існує багато готових рішень, накопичених протягом багатьох років, які можуть бути використані під час розробки проєкту.

Для ефективної розробки програмного забезпечення важливим є вибір відповідного інтегрованого середовища розробки (IDE). Правильний вибір IDE може значно підвищити продуктивність і спростити процес розробки. У даному проєкті було вирішено використовувати Qt Creator.

Qt Creator — це мультиплатформене IDE, яке забезпечує потужні можливості для розробки програмного забезпечення на C++ з використанням фреймворку Qt [3].

Інтерфейс Qt Creator (рис. 1.5) є зручним та інтуїтивно зрозумілим, що полегшує його використання як новачкам, так і досвідченим розробникам. Крім того, інтерфейс можна налаштувати під свої потреби, що дозволяє кожному розробнику працювати в комфортних умовах.

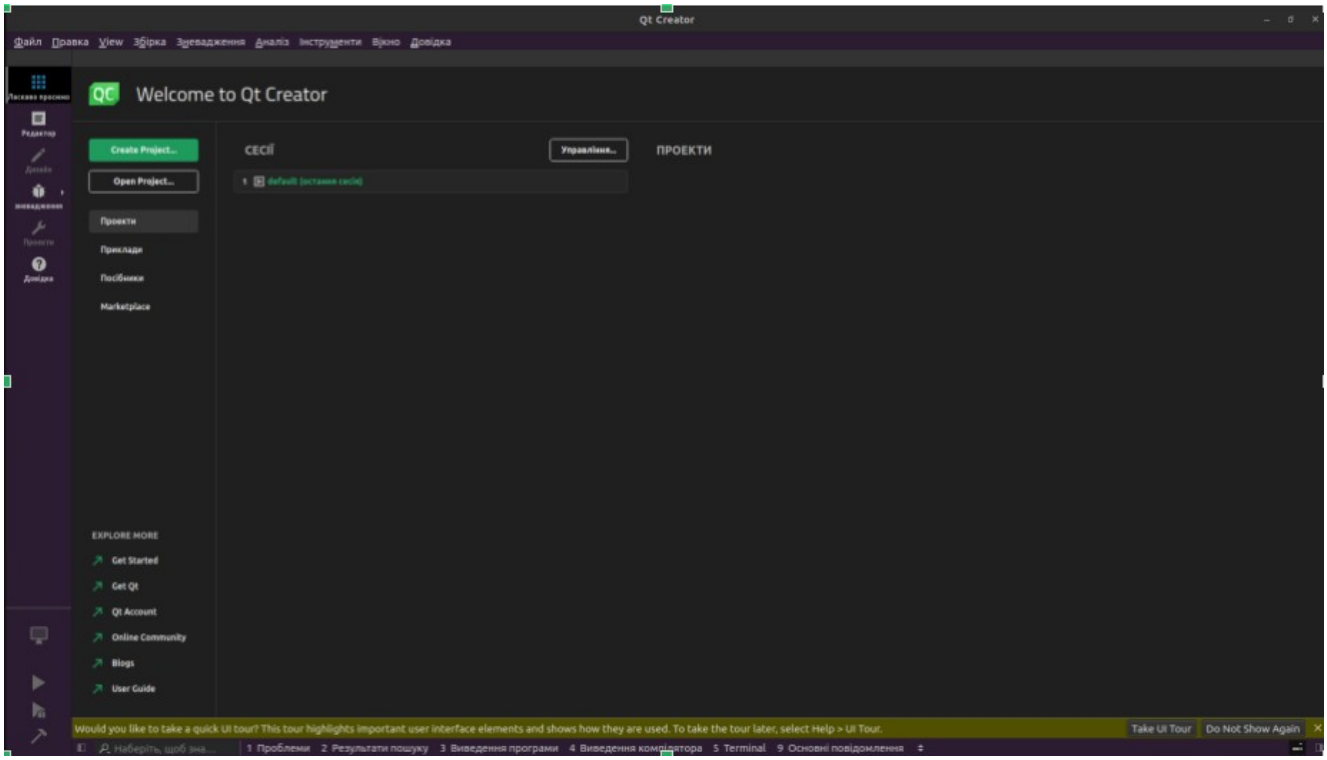

Рис. 1.5 – Інтерфейс Qt Creator

Qt Creator має вбудований розділ "Довідка" (рис. 1.6), який містить детальну документацію до класів фреймворку Qt та системи збірки CMake. Це забезпечує

швидкий доступ до необхідної інформації, що значно прискорює процес розробки та вирішення проблем.

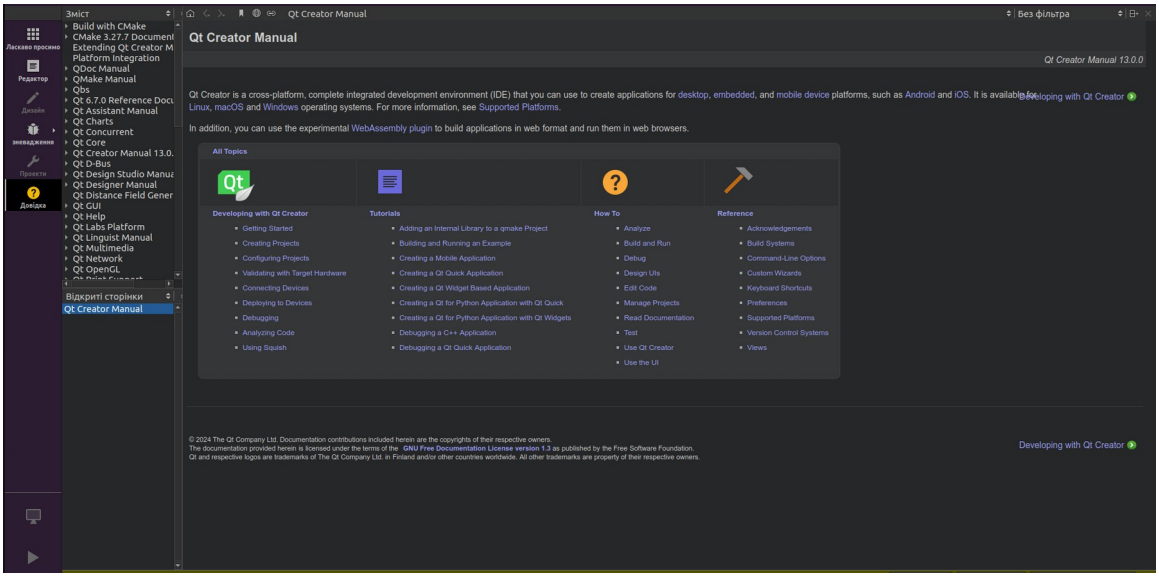

Рис. 1.6 – Розділ “Довідка” у Qt Creator

Редактор (рис. 1.7) коду в Qt Creator має такі функції, як автодоповнення, підказки та знаходження помилок. Ці інструменти допомагають розробникам швидше писати код та уникати поширених помилок [3].

Рис. 1.7 – Редактор у Qt Creator

Вбудований у Qt Creator графічний редактор Qt Designer (рис. 1.8) надає інструменти для швидкого створення інтерфейсу програми. Створений інтерфейс зберігається у файлах формату .ui, які містять XML-код. Цей код використовується під час роботи програми для відображення користувацького інтерфейсу [3].

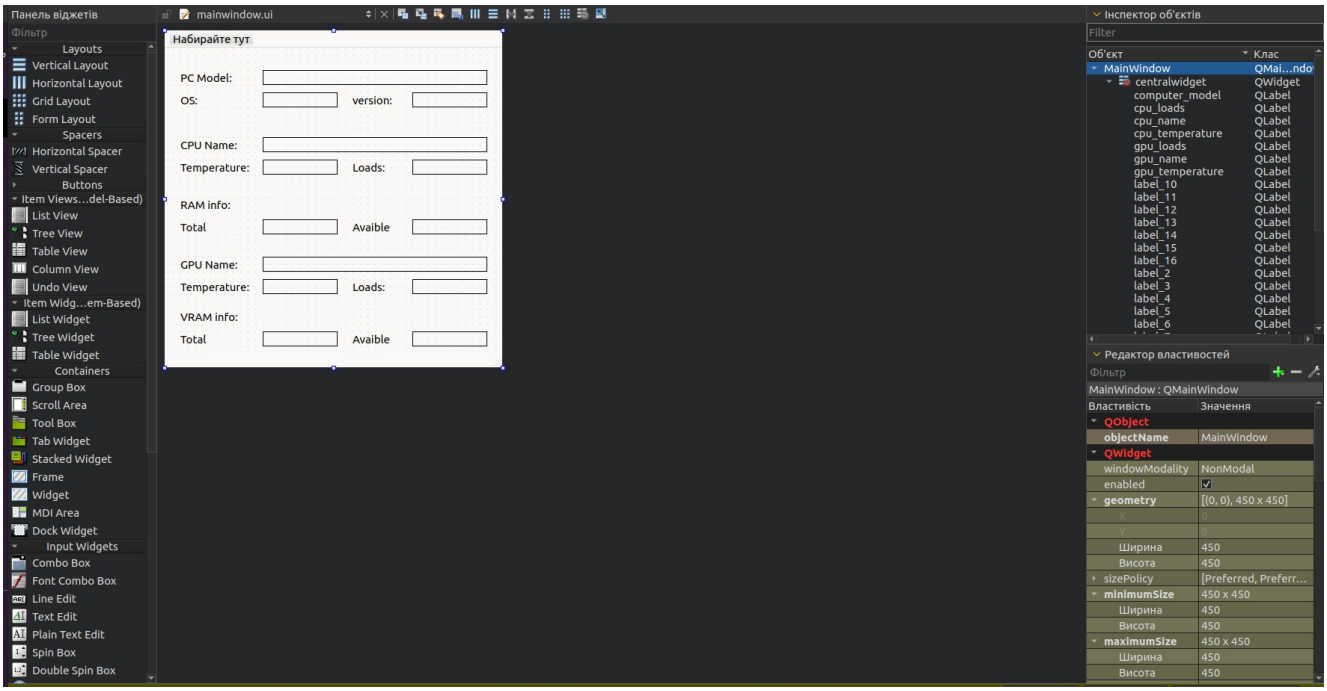

Рис. 1.8 – Qt Designer

IDE підтримує різні системи збірки, включаючи CMake та QMake, що забезпечує гнучкість у виборі інструментів для збірки проєкту [3].

Вибір Qt Creator як основного IDE для цього проєкту обґрунтований його потужними можливостями, зручністю використання та підтримкою фреймворку Qt. Використання цього середовища забезпечує високу продуктивність розробки, швидке створення якісних графічних інтерфейсів та легке керування кодом. Крім того, велика спільнота розробників та наявність детальної документації роблять Qt Creator відмінним вибором для розробки сучасного програмного забезпечення.

РОЗДІЛ 2. ПРОЕКТУВАННЯ ДОДАТКУ «SympleSysMonitor»

2.1 Розробка загальної структури та вигляду додатку

Програмний додаток «SimpleSysMonitor» [4] складатиметься з одного віджету, в якому знаходиться кілька основних розділів, зв'язаних між собою, доступ до яких можна отримати через простий та інтуїтивно зрозумілий інтерфейс. На головній сторінці додатку користувач побачить зведену інформацію про основні параметри системи, такі як модель ПК, операційна система, центральний процесор (CPU), оперативна пам'ять (RAM), графічний процесор (GPU) та відеопам'ять (VRAM).

Після запуску додатку користувач зможе ознайомитися з такими розділами:

- Інформація про систему - дані про модель ПК та версію встановленої операційної системи Ubuntu.
- Відомості про CPU - назва процесора, поточна температура та навантаженість у відсотках.
- Відомості про RAM: Показує загальний обсяг оперативної пам'яті та доступну пам'ять у реальному часі.
- Відомості про GPU: - назву графічного процесора, його температуру та навантаженість у відсотках.
- Відомості про VRAM: Відображає загальний та доступний обсяг відеопам'яті.

Користувач може налаштовувати відображення інформації відповідно до своїх потреб. Додаток «SimpleSysMonitor» [4] створений для зручного та швидкого моніторингу стану системи, що дозволяє вчасно виявляти потенційні проблеми та підтримувати стабільну роботу ПК.

Основними кольорами додатку будуть білий та чорний, відповідно до теми встановленій в ОС користувача.

Для відображення можливостей користувача було створено діаграму варіантів використання. Ця діаграма допомагає візуалізувати, як користувач взаємодіє з додатком «SimpleSysMonitor» [4] для отримання інформації про різні апаратні компоненти системи. Вона дозволяє краще розуміти структуру додатку та його функціональні можливості.

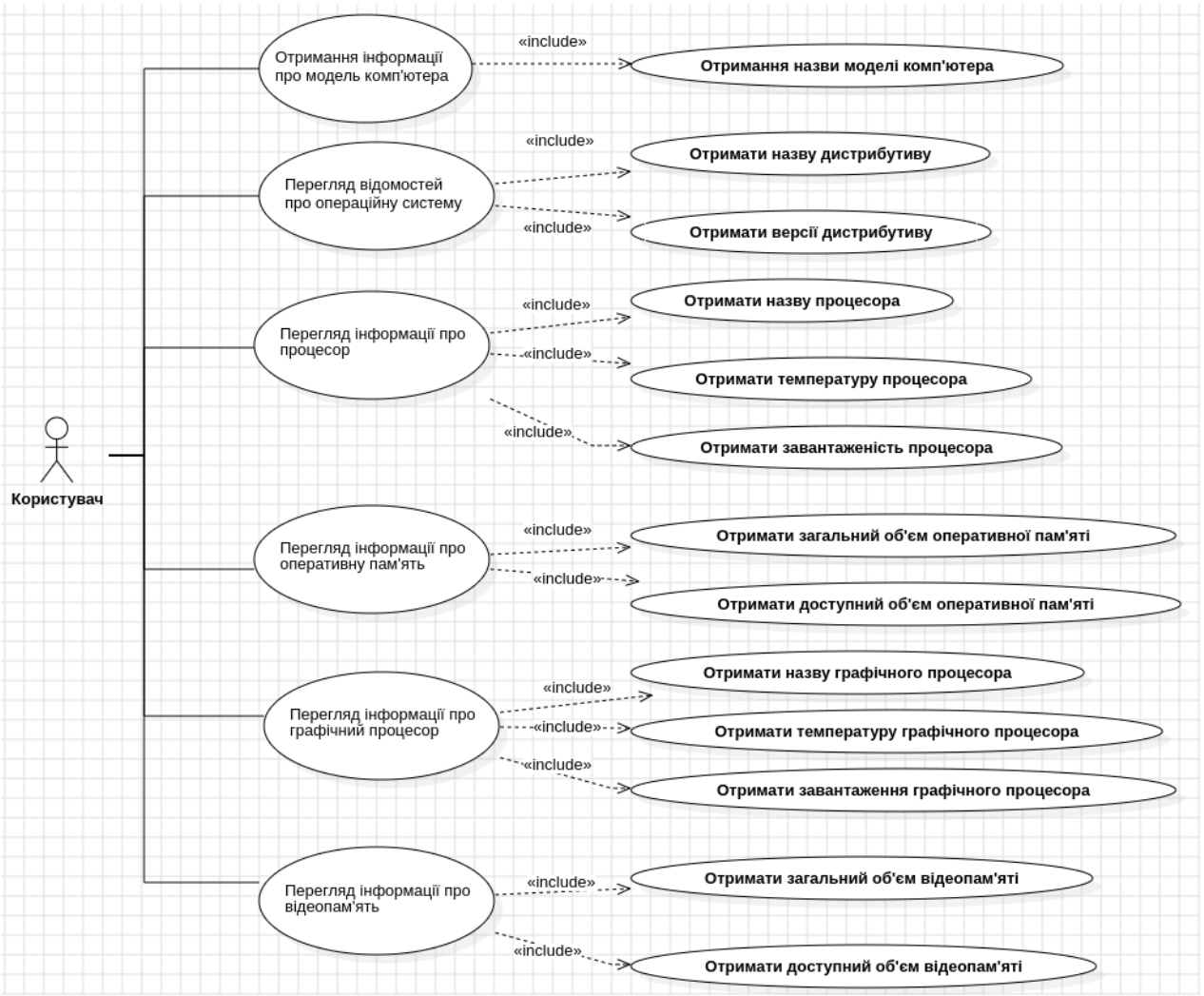

Рис. 2.1 – Діаграма варіантів використання додатку користувачем

Як можна побачити з діаграми користувач має шість основних можливостей, які включають в собі 2-3 сценарії. Кожен з цих сценаріїв є окремим випадком використання, що детально описує конкретну функціональність додатку. Зв'язки типу «include» показують, що певні дії включають в себе виконання піддій для досягнення кінцевого результату.

2.2 Проектування архітектури

Отримавши відомості про майбутній додаток та особливості технологій, які будуть використовуватись, можна перейти до проектування архітектури. Це включає визначення необхідного патерну програмування для побудови ієрархії класів. Важливо обрати патерн, який дозволить розділити графічну та логічну частини програми, оскільки це сприятиме розширенню функціональності додатку та збільшенню кількості підтримуваних платформ. Крім того, логічна частина програми не потребує взаємодії з користувачем.

При виборі патерну необхідно враховувати такі фактори:

- Розділення логічної та графічної складової програми.
- Можливість легкого розширення функціональності.
- Мінімальна взаємодія користувача з логічною частиною програми.
- Поділ логічної частини на окремі класи з розподілом функціональності програми.

Після аналізу патернів програмування було обрано три основні патерни: Observer, Bridge та Model-View-Controller (MVC). Розглянемо особливості

Observer (Спостерігач) - це поведінковий патерн проектування, який створює механізм підписки, що дає змогу одним об’єктам стежити й реагувати на події, які відбуваються в інших об’єктах [5].

Bridge (Міст) - це структурний патерн проектування, який розділяє один або кілька класів на дві окремі ієрархії — абстракцію та реалізацію, дозволяючи змінювати код в одній гілці класів, незалежно від іншої [5].

Model-View-Controller (MVC) це модель дизайну програмного забезпечення, яка зазвичай використовується для розвитку інтерфейсів користувача, що розділяє пов'язану логіку програми на три взаємопов'язані елементи: модель (внутрішнє подання інформації), вид (інтерфейс, який представляє інформацію та приймає її

від користувача), та контролер (програмне забезпечення, що пов'язує модель та вид) [5].

При виборі патерну для загальної структури програми необхідно враховувати особливості фреймворку Qt. Для побудови окремих сукупностей класів можна використати будь-який патерн, але важливо вибрати патерн, який буде найкраще поєднуватися із стандартним кодом Qt. Тому розглянемо особливості обраних патернів у поєднанні із структурою Qt.

Model-View-Controller (MVC): Незважаючи на те, що патерн MVC не повністю підходить до ієрархії класів Qt, концепція цього патерну є найбільш схожою до тої, що необхідна для проєкту [5].

Observer (Спостерігач): Цей патерн часто використовується в Qt за допомогою механізму сигналів і слотів, що забезпечує взаємодію між об'єктами [5].

Bridge (Міст): Патерн Міст дозволяє легко розділити логічну та графічну частину програми, але його ієрархія класів не завжди поєднується з ієрархією класів стандартної графічної програми Qt [5].

Для реалізації необхідно знайти модифікацію патерну MVC, яка краще поєднується із стандартним кодом Qt. Під час розробки графічного інтерфейсу у дизайнері Qt Creator середовище автоматично створює .ui файл, що містить XML код. Під час компіляції цей код перетворюється в клас, який відповідає за створення графічного інтерфейсу, тобто виконує роль View у моделі MVC.

Клас MainWindow зазвичай відповідає за зміну інтерфейсу програми та інформації, що подається там. Він виконує роль посередника між графічною та логічною частинами програми, тобто є випадком View-Controller класу, який одночасно виконує ролі контролера та виду.

Клас Model представляє сукупність класів, які отримують дані про параметри компонентів комп’ютера.

Таким чином, модифікація патерну MVC для даного проєкту матиме таку структуру:

- View (слабкий) - представлений .ui файлом, який створює графічний інтерфейс.
- View-Controller - впливає на інтерфейс після його створення і виконує роль посередника між графічною та логічною частинами.
- Model - сукупність класів, що відповідають за обробку даних.

Маючи патерн, за яким буде побудована ієрархія класів, та відомості про систему з попередніх розділів, можна приступити до визначення, які класи будуть присутні в системі.

Отже можна виділити такі класи які потрібні системі, а саме:

- UI::MainWindow збереження інформації про структуру вікна, та її початкове створення.
- MainWindow який частково відповідатиме за створення графічного інтерфейсу, повністю контролюватиме його зміни та буде посередником між графічною та логічною частиною програми, клас
- CPU_info відповідатиме за отримання інформації про процесор комп’ютера,
- RAM_info — отримання інформації про оперативну пам’ять,
- GPU_info — отримання інформації про відеокарту.

Маючи структуру та класи програми можна приступити до побудови діаграм класів для візуалізації відомостей про класи та їх ієрархію. Для побудови діаграми буде використано мову візуалізації додатків UML [7]. На сьогоднішній день ця мова є основною для представлення додатків. В цій мові є певний набір графічних інструментів, який допомагає розробникам у передачі своїх ідей, та концепцій.

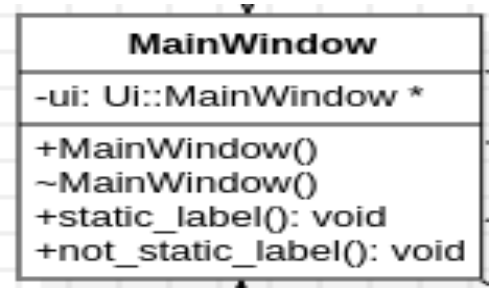

Рис. 2.2 – Клас MainWindow

На рисунку 2.2 зображено клас MainWindow, який відповідає за створення головного вікна програми. Він наслідує функціональність класу QMainWindow і забезпечує основні можливості інтерфейсу користувача. А також зберігає посилання на клас UI::MainWindow, який використовується для відтворення інтерфейсу програми за xml кодом.

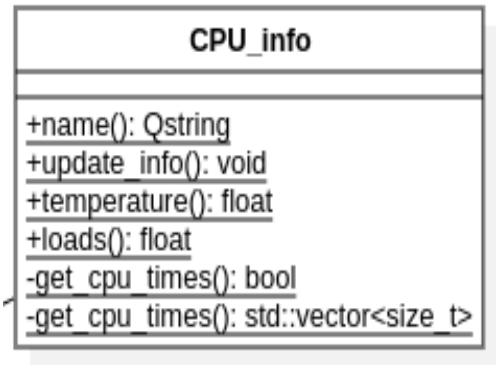

Рис. 2.3 – Клас CPU_info

На рисунку 2.3 зображено клас CPU_info, який відповідає за отримання та обробку інформації про центральний процесор. Цей клас містить лише статичні функції, що дозволяє викликати їх без необхідності створення екземпляра класу.

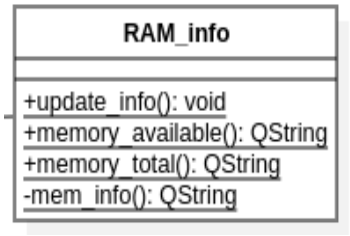

Рис. 2.4 – Клас RAM_info

На рисунку 2.4 зображено клас RAM_info, який відповідає за отримання та оновлення інформації про оперативну пам'ять системи. Цей клас містить лише статичні функції, що дозволяє викликати їх без необхідності створення екземпляра класу.

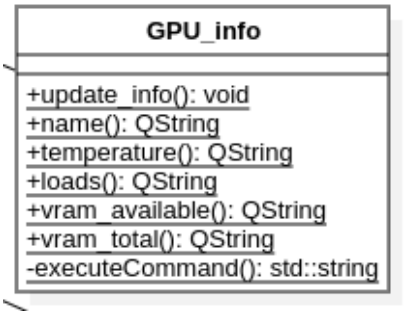

Рис. 2.5 – Клас GPU_info

На рисунку 2.5 зображено клас GPU_info, як видно з наведеного коду, відповідає за отримання та оновлення інформації про графічний процесор (GPU). Цей клас містить лише статичні функції, що дозволяє викликати їх без необхідності створення екземпляра класу.

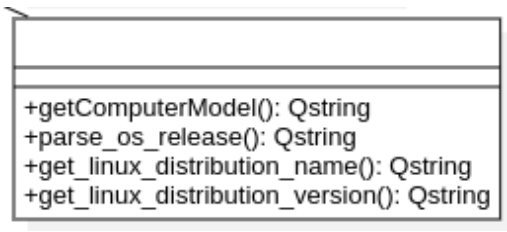

Рис. 2.6 – Додаткові функції програми

На рисунку 2.6 зображено додаткові функції програми які не були поміщені в окремий клас. Дані функції відповідають за отримання назви моделі комп’ютера, та відомостей про операційну систему.

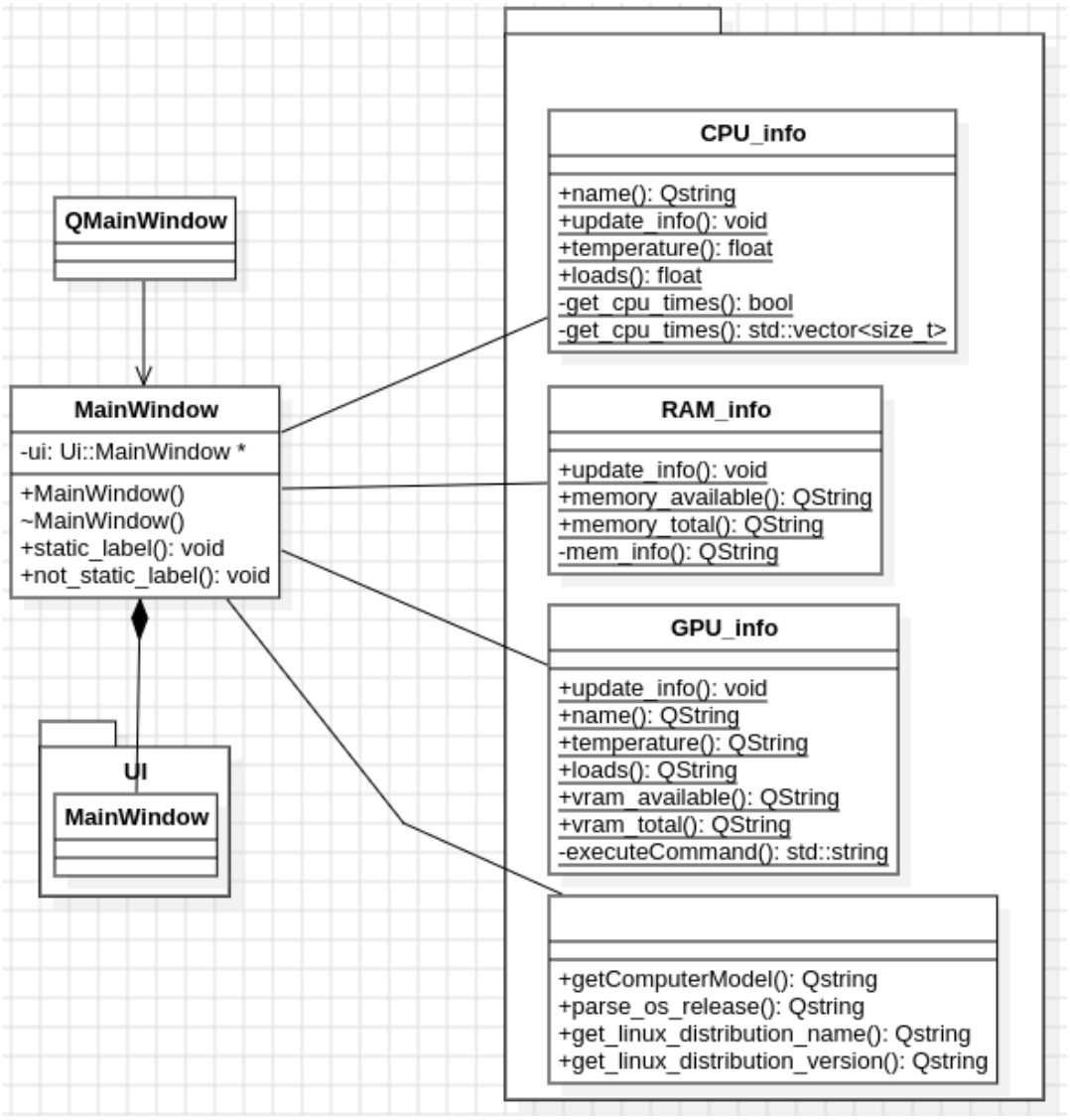

Рис. 2.7 – Діаграма класів додатку «SimpleSysMonitor»

На рисунку 2.7 зображено загальну структуру додатку, відношення класів, та важливі класи фреймворку від яких успадковуються деякі класи додатку. Також як можна помітити частина класів містять лише статичні поля. Дане рішення було використано для того щоб згрупувати функції, а також обмежити доступ до деяких функцій.

РОЗДІЛ 3. КОНСТРУЮВАННЯ ТА ТЕСТУВАННЯ ДОДАТКУ «SimpleSysMonitor»

3.1 Реалізація ключових класів

Для реалізації ключових класів спершу необхідно створити графічний інтерфейс програми, тобто той елемент який відповідатиме View (слабкий) в обраному патерні програмування. Для його створення було використано інструменти Qt Designer. Графічний інтерфейс передбачає в собі усі поля згадані у попередніх розділах. Результат створення графічного інтерфейсу показано на рисунку 3.1.

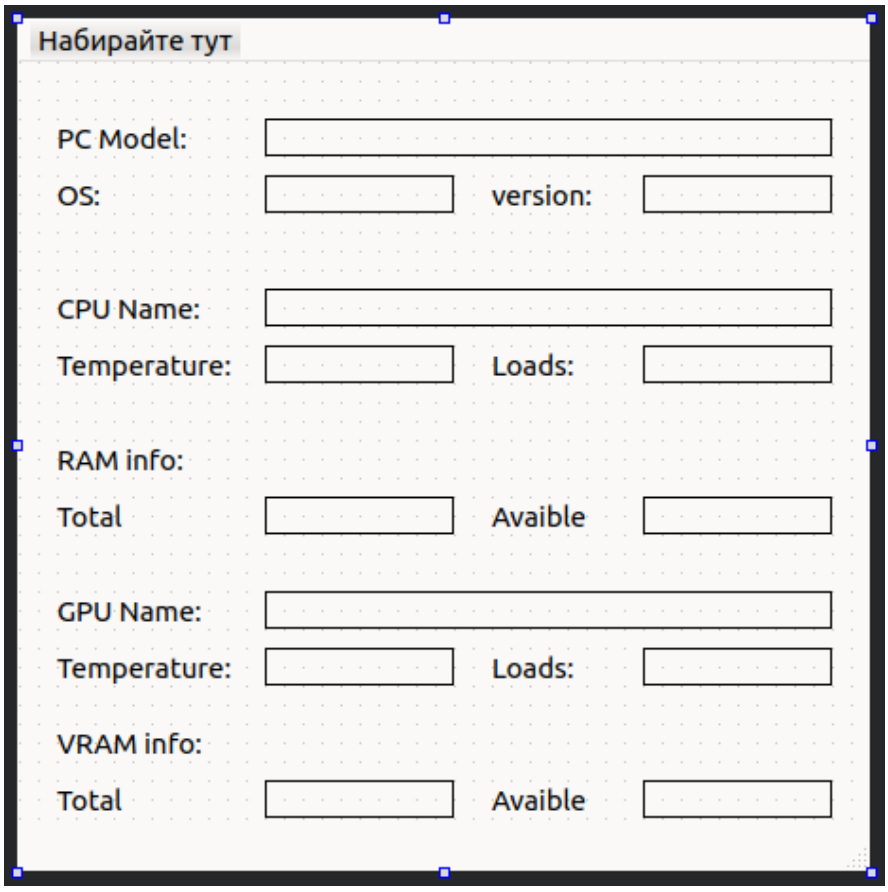

Рис. 3.1 – Графічний інтерфейс додатку «SimpleSysMonitor» у середовищі Qt Designer

Даний графічний інтерфейс зберігається засобами Qt Designer у вигляді xml коду(рис 3.2). Даний xml код використовується середовищем Qt Creator для генерації C++ коду для побудови вікна програми та її елементів. Отриманий C++ код являє собою Ui::MainWindow клас [3].

```xml
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
 <class>MainWindow</class>
 <widget class="QMainWindow" name="MainWindow">
  <property name="geometry">
   <rect>
    <x>0</x>
    <y>0</y>
    <width>450</width>
    <height>450</height>
   </rect>
  </property>
  <property name="minimumSize">
   <size>
    <width>450</width>
    <height>450</height>
   </size>
  </property>
  <property name="maximumSize">
   <size>
    <width>450</width>
    <height>450</height>
   </size>
  </property>
```

Рис. 3.2 – Приклад xml коду створеного в Qt Designer

Згенерований клас надалі компілюється та приєднається до проєкту. На рисунку 3.3 зображено попереднє оголошення класу Ui::MainWindow.

```
QT_BEGIN_NAMESPACE
namespace Ui {
class MainWindow;
}
QT_END_NAMESPACE
```

Рис. 3.3 – Оголошення класу Ui::MainWindow

Наступним елементом із обраного патерну, який буде реалізований, буде елемент View-Controller, представлений у класі MainWindow. Даний клас також використовується для побудови графічного інтерфейсу, а також відповідає за відображення даних. Він успадковується від класу QMainWindow [3].

Головне вікно забезпечує основу для створення інтерфейсу користувача програми. Qt має QMainWindow та його related classes для управління основними вікнами. QMainWindow має власний макет, до якого можна додати QToolBarc, QDockWidgetc, a QMenuBar, і a QStatusBar. Макет має центральну зону, яку може займати будь-який віджет. Тому для створення класу MainWindow було обрано QMainWindow як батьківський клас. Ви можете побачити зображення макета QMainWindow нижче на рисунку 3.4

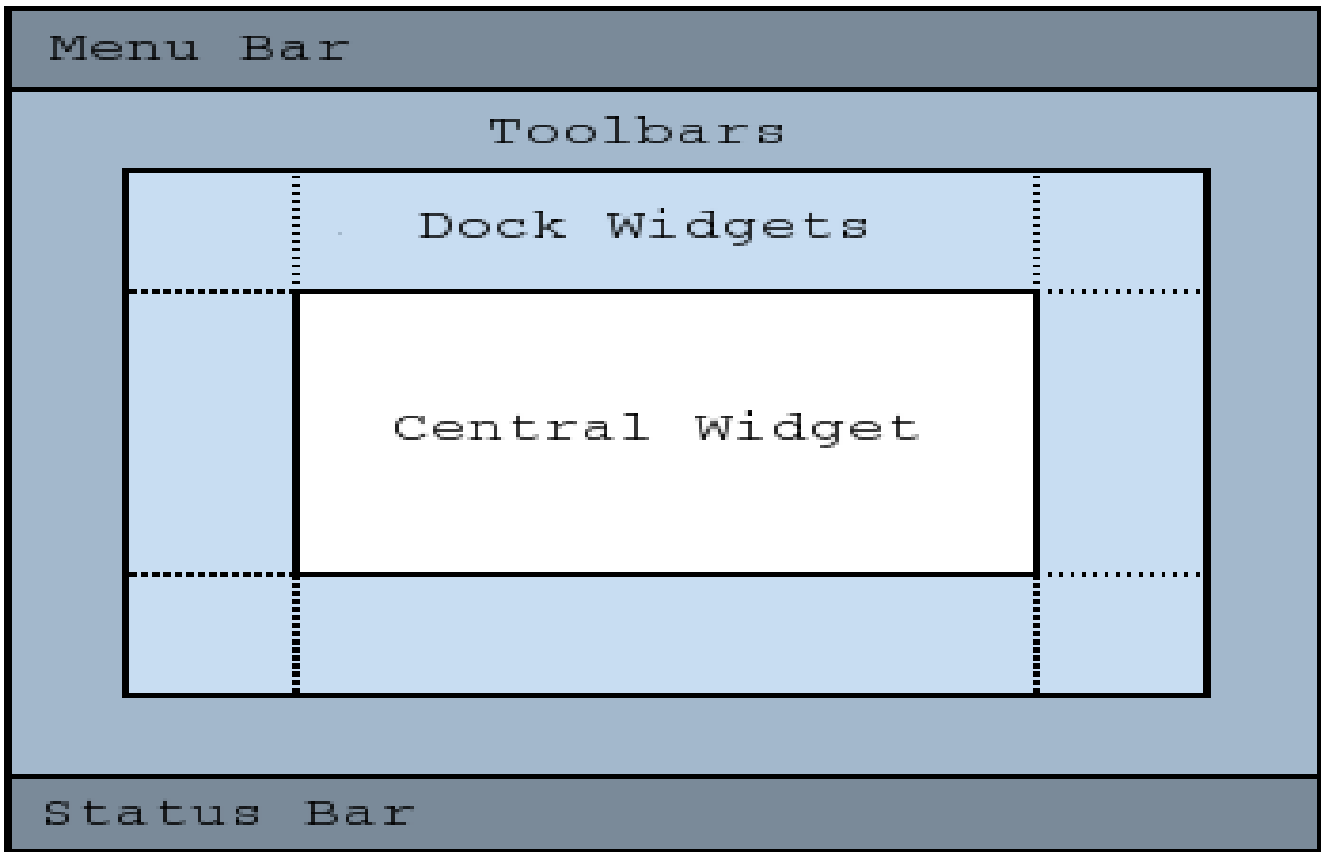

Рис. 3.4 – Макет QMainWindow

На рисунку 3.5 зображено код з оголошенням класу MainWindow. Як можна побачити це є стандартний код який генерується редактором Qt Creatr, за винятком двох методів, а саме static_label() і not_static_label(). Конструктор запускає код з класу Ui::MainWindow для створення графічного інтерфейсу програми. Деструктор відповідає за звільнення динамічної пам’яті.

```
class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();
    void static_label();
    void not_static_label();


private:
    Ui::MainWindow *ui;
};
```

Рис. 3.5 – код класу MainWindow

Розглянемо детальніше методи які були додані до даного класу. Почнемо із static_label(), код якого зображено на рисунку 3.6. Даний метод викликається лише раз. Він отримує вказівники на елементи інтерфейсу, та відправляє їх у функції та методи класів для отримання даних які не будуть оновлюватись впродовж роботи програми. Наприклад назва моделі комп’ютера, назва та версія операційної системи, назва центрального та графічного процесорів, загальний обсяг оперативної та відео пам’яті.

```
void MainWindow::static_label(){

    get_computer_model(ui->computer_model);

    get_linux_distribution_name(ui->os_name);
    get_linux_distribution_version(ui->os_version);

    ui->cpu_name->setText(CPU_info::name());

    ui->ram_total_memory->setText(RAM_info::memory_total());

    ui->gpu_name->setText(GPU_info::name());
    ui->vram_total_memory->setText(GPU_info::vram_total());

}
```

Рис. 3.6 – код методу static_label()

Розглянемо наступним метод not_static_label(). Даний метод також викликається лише раз, але на відміну від static_label() він працює увесь час роботи програми. Як ми можемо побачити на рисунку 3.7 даний метод використовує клас QFuture для створення трьох паралельних потоків. Кожен із потоків відповідає за оновлення даних у певних полях. Використання паралельних потоків дозволяє прискорити програму, оскільки всі функції виконуються одночасно і не має потреби очікувати завершення усіх функцій для оновлення даних. Також використання паралельних процесів дозволяє уникати помилок які виникають з використанням вічного циклу та послідовного виконання програм. Поля які оновлюються з використанням даного методу містять змінні дані, такі як температура та завантаженість якихось компонентів комп'ютера.

```
void MainWindow::not_static_label(){
    QFuture<void> result1 =
        QtConcurrent::run(CPU_info::update_info, ui->cpu_temperature, ui->cpu_loads);

    QFuture<void> result2 =
        QtConcurrent::run(RAM_info::update_info, ui->ram_avaible_memory);

    QFuture<void> result3 =
        QtConcurrent::run(GPU_info::update_info, ui->gpu_temperature, ui->gpu_loads, ui->vram_avaible_memory);

}
```

Рис. 3.7 – код методу not_static_label()

Оскільки елементи програми які відповідають за компоненти View та View-Controller у обраному патерні вже створені, то тепер можна почати створення класів сукупність яких буде відповідати компоненту Model. Як було описано у попередньому розділі всі класи містять лише статичні методи. Це необхідно для групування функції та обмеження доступу до деяких із них щоб забезпечити інкапсуляцію.

Ознайомимось із класом CPU_info (рис. 3.8). Даний клас не потребує створення об’єкта оскільки містить лише статичні методи. Метою даного класу є збереження функцій для доступу до інформації пов’язаної із процесором.

```
class CPU_info{
public:
    static QString name();

    static void update_info(QLabel *temperature_label, QLabel *loads_label);

    static float temperature ();
    static float loads ();

private:
    static bool get_cpu_times(size_t &idle_time, size_t &total_time);
    static std::vector<size_t> get_cpu_times();
};
```

Рис. 3.8 – код класу CPU_info

Першим методом класу CPU_info є метод name() (рис. 3.9) даний метод використовує звернення до системного файлу “cpuinfo”. Після отримання певної інформації відбувається пошук параметру “model name”, знайшовши даний параметр зчитується частина стрічки в якій був даний параметр. Отриманий рядок повертається результатом виконання даного методу.

```
QString CPU_info::name(){
    std::string line;
    std::ifstream finfo("/proc/cpuinfo");
    while(getline(finfo,line)) {
        std::stringstream str(line);
        std::string itype;
        std::string info;
        if ( getline( str, itype, ':' ) && getline(str,info) && itype.substr(0,10) == "model name" ) {
            return QString::fromStdString(info);
        }
    }
}                                                    Δ Non-void function does not return
```

Рис. 3.9 – код методу name()

Наступним методом класу CPU_info є метод update_info() (рис. 3.10). Даний метод викликається в окремому паралельному потоці ще при запуску програми. Даний метод повинен працювати вічно поки увімкнена програма. Саме тому всередині цього методу використовується вічний цикл. В середині цього вічного циклу відбувається оновлення даних в отриманих полях, з використанням методів які отримують відповідну інформацію температуру та завантаженість процесора. Дані методи будуть розглянуті пізніше. Ще важливою деталлю цього циклу є зупинка потоку з використанням функції sleep(). Дана функція зупиняє час на одну секунду це забезпечить більш плавну зміну даних у програмі, а також зменшить навантаження на процесор. Тому функція sleep() оптимізує програму, адже без неї процесор буде виконувати безліч операцій з оновленням даних, за дуже короткий термін

```
void CPU_info::update_info(QLabel *temperature_label, QLabel *loads_label)
{
    while(true){

        temperature_label->setText(QString::number(temperature()) + " °C");

        loads_label->setText(QString::number(loads()) + " %");

        sleep(1);
    }
}
```

Рис. 3.10 – код методу update_info()

Метод класу CPU_info temperature() (рис. 3.11) переважно використовує код на мові C для доступу до системного файлу thermal_zone0, для знаходження відомостей про поточну температуру процесора та повертає її в градусах Цельсія.

```
float CPU_info::temperature (){
    float systemp, millideg;
    FILE *thermal;
    [[maybe_unused]]int n;

    thermal = fopen("/sys/class/thermal/thermal_zone0/temp","r");
    n = fscanf(thermal,"%f",&millideg);
    fclose(thermal);
    systemp = millideg / 1000;

    return systemp;
}
```

Рис. 3.11 – код методу themperature()

Методи в класі CPU_info використовуються для отримання інформації про завантаженість процесора. Метод loads() (рис. 3.12) обчислює завантаженість процесора в процентах, визначаючи різницю в часі простою та загальному часі між двома вимірюваннями з інтервалом у 1 секунду, після чого повертає цю величину. Для цього він використовує допоміжні методи get_cpu_times. Ці два методи мають однакову назву, але є перевантаженими.

```
float CPU_info::loads (){
    size_t previous_idle_time=0, previous_total_time=0;
    for (size_t idle_time, total_time; get_cpu_times(idle_time, total_time); sleep(1)) {
        const float idle_time_delta = idle_time - previous_idle_time;
        const float total_time_delta = total_time - previous_total_time;
        const float utilization = 100.0 * (1.0 - idle_time_delta / total_time_delta);

        return utilization;

        previous_idle_time = idle_time;
        previous_total_time = total_time;
    }
}                                                    Δ Non-void function d
```

Рис. 3.12 – код методу loads()

Метод `std::vector<size_t> get_cpu_times()` (рис. 3.13) читає файл `/proc/stat`, щоб отримати значення часу для кожного стану процесора і повертає їх у векторі. Метод `bool get_cpu_times()` (рис. 3.13) викликає попередній метод, щоб отримати ці значення, встановлює час простою та загальний час, а потім повертає true, якщо вдалося отримати необхідні дані, або false, якщо даних недостатньо.

```
std::vector<size_t> CPU_info::get_cpu_times() {
    std::ifstream proc_stat("/proc/stat");
    proc_stat.ignore(5, ' ');
    std::vector<size_t> times;
    for (size_t time; proc_stat >> time; times.push_back(time));
    return times;
}

bool CPU_info::get_cpu_times(size_t &idle_time, size_t &total_time) {
    const std::vector<size_t> cpu_times = get_cpu_times();
    if (cpu_times.size() < 4)
        return false;
    idle_time = cpu_times[3];
    total_time = std::accumulate(cpu_times.begin(), cpu_times.end(), 0);
    return true;
}
```

Рис. 3.13 – код методів get_cpu_times()

Розглянемо клас RAM_info (рис. 3.14). Даний клас не потребує створення об’єкта оскільки містить лише статичні методи. Метою даного класу є збереження функцій для доступу до інформації пов’язаної із оперативною пам'яттю.

```
class RAM_info{
public:
    static void update_info(QLabel *memory_available_label);

    static QString memory_available();
    static QString memory_total();
private:
    static QString mem_info(std::string mem_name);
};
```

Рис. 3.14 – код класу RAM_info

Першим методом класу RAM_info є метод update_info() (рис. 3.15). Даний метод еквівалентний відповідному класу із CPU_info(), але на відміну від нього, даний метод відповідає лише за оновлення даних про доступну оперативну пам’ять.

```
void RAM_info::update_info(QLabel *memory_available_label)
{
    while(true){
        memory_available_label->setText(memory_available());
        sleep(1);
    }
}
```

Рис. 3.15 – код методу not_static_label()

Найважливішим методом класу RAM_info є метод mem_info(рис. 3.16). Даний метод необхідний для взаємодії із системним файлом “meminfo”. Він приймає від інших методів цього ж класу рядки із фрагментом команди для отримання того чи іншого параметру, та повертає результат цього звернення до файлу.

```
QString RAM_info::mem_info(std::string mem_name){
    double mem = -1;
    std::ifstream meminfo("/proc/meminfo");
    std::string line;
    while (std::getline(meminfo, line))
    {
        if (line.find(mem_name+":") != std::string::npos)
        {
            const std::size_t firstWhiteSpacePos = line.find_first_of(' ');
            const std::size_t firstNonWhiteSpaceChar = line.find_first_not_of(' ', firstWhiteSpacePos);
            const std::size_t nextWhiteSpace = line.find_first_of(' ', firstNonWhiteSpaceChar);
            const std::size_t numChars = nextWhiteSpace - firstNonWhiteSpaceChar;
            const std::string memAvailableStr = line.substr(firstNonWhiteSpaceChar, numChars);
            mem = std::stoll(memAvailableStr);
            break;
        }
    }

    return QString::number(mem/1024/1024)+" Gb";
}
```

Рис. 3.16 – код методу mem_info()

Решта методів даного класу є однотипними. Між ними усіма існує лише дві відмінності, а саме назва та команда яка використовується для звернення до бібліотеки. Приклад коду наведено на рисунку 3.17.

```
QString RAM_info::memory_available()
{
    return mem_info("MemAvailable");
}

QString RAM_info::memory_total()
{
    return mem_info("MemTotal");
}
```

Рис. 3.17 – код методу not_static_label()

Ознайомимось із класом GPU_info (рис. 3.18). Даний клас не потребує створення об’єкта оскільки містить лише статичні методи. Метою даного класу є збереження функцій для доступу до інформації пов’язаної із процесором. Також на відміну від попередніх класів, даний клас не здатен отримати необхідну інформацію пов’язану із відеокартою, оскільки для таких дій необхідно прописувати драйвери, або ж використовувати уже наявні бібліотеки. В даному проєкті було використано бібліотеку “nvidia-smi”.

```
class GPU_info{
public:
    static void update_info(QLabel *temperature_label, QLabel *loads_label, QLabel *vram_available_label);

    static QString name();

    static QString temperature();
    static QString loads ();
    static QString vram_available();
    static QString vram_total();

private:
    static std::string executeCommand(const std::string& command);
};
```

Рис. 3.18 – код класу GPU_info

Першим методом класу GPU_info є метод update_info() (рис. 3.19). Даний метод еквівалентний відповідному класу із CPU_info(), але на відміну від нього, даний метод також відповідає за оновлення даних про відео пам’ять.

```
void GPU_info::update_info(QLabel *temperature_label, QLabel *loads_label, QLabel *vram_available_label)
{
    while(true){
        temperature_label->setText(temperature());
        loads_label->setText(loads());
        vram_available_label->setText(vram_available());
        sleep(1);
    }
}
```

Рис. 3.19 – код методу update_info()

Найважливішим методом даного класу GPU_info є метод executeCommand() на рисунку 3.20. Даний метод необхідний для взаємодії із бібліотекою “nvidia-smi”. Він приймає від інших методів цього ж класу рядки із командами для отримання того чи іншого параметру, та повертає результат цього звернення до бібліотеки.

```
std::string GPU_info::executeCommand(const std::string& command) {
    std::array<char, 128> buffer;
    std::string result;
    std::unique_ptr<FILE, decltype(&pclose)> pipe(popen(command.c_str(), "r"), pclose);

    if (!pipe) {
        return "ERROR ";
    }

    while (fgets(buffer.data(), buffer.size(), pipe.get()) != nullptr) {
        result += buffer.data();
    }

    return result;
}
```

Рис. 3.20 – код методу executeCommand()

Решта методів даного класу є однотипними. Між ними усіма існує лише дві відмінності, а саме назва та команда яка використовується для звернення до бібліотеки. Тому буде наведено приклад коду лиш одного методу на рисунку 3.21

```
QString GPU_info::name() {
    std::string command = "nvidia-smi --query-gpu=name --format=csv,noheader";

    QString result = QString::fromStdString(executeCommand(command));
    result.chop(1);
    return result;
}
```

Рис. 3.21 – код методу name()

Також до проєкту було додано кілька функцій які мали б надати допоміжну інформацію. Дані функції не були віднесені до якогось наявного чи окремого класу.

Першою із таких функцій є функція з отриманням назви моделі комп’ютера get_computer_model() (рис. 3.22). Для цього функція звертається до системного файлу product_name.

```
void get_computer_model(QLabel *label) {
    std::ifstream file("/sys/devices/virtual/dmi/id/product_name");

    std::string model;
    std::getline(file, model);
    file.close();
    label->setText(QString::fromStdString(model));

}
```

Рис. 3.22 – код функції not_static_label()

Також важливою функцією є функція parse_os_release() (рис. 3.23). Дана функція відповідає за доступ до системного файлу “os-release”. Вона необхідна для двох наступних функцій.

```cpp
std::unordered_map<std::string, std::string> parse_os_release() {
    std::unordered_map<std::string, std::string> info;
    std::ifstream file("/etc/os-release");

    if (!file.is_open()) {
        throw std::runtime_error("Unable to open /etc/os-release");
    }

    std::string line;
    while (std::getline(file, line)) {
        auto delimiter_pos = line.find('=');
        if (delimiter_pos == std::string::npos) {
            continue;
        }

        std::string key = line.substr(0, delimiter_pos);
        std::string value = line.substr(delimiter_pos + 1);

        // Remove surrounding quotes from the value
        if (value.size() > 1 && value.front() == '"' && value.back() == '"') {
            value = value.substr(1, value.size() - 2);
        }

        info[key] = value;
    }

    return info;
}
```

Рис. 3.23 – код функції parse_os_release()

Функції get_linux_distribution_name і get_linux_distribution_version використовують параметри за якими відбувається пошук необхідних даних функцією parse_os_release() (рис. 3.24). Функції get_linux_distribution_name і get_linux_distribution_version повертають інформацію про назву та версію операційної системи відповідно.

```cpp
void get_linux_distribution_name(QLabel *label) {
    try {
        auto info = parse_os_release();
        label->setText(QString::fromStdString(info["NAME"]));
    } catch (const std::exception& e) {
        std::cerr << "Error: " << e.what() << std::endl;
        label->setText("Unknown");
    }
}

void get_linux_distribution_version(QLabel *label) {
    try {
        auto info = parse_os_release();
        label->setText(QString::fromStdString(info["VERSION_ID"]));
    } catch (const std::exception& e) {
        std::cerr << "Error: " << e.what() << std::endl;
        label->setText("Unknown");
    }
}
```

Рис. 3.24 – код функцій get_linux_distribution_name() і get_linux_distribution_version()

3.2 Тестування та результат розробки

На етапі тестування мною було отримано результати співпраці та поведінку окремих класів програми. Метою тестування було виявлення помилок у програмі, для подальшого виправлення, та представлення кінцевого проєкту під час захисту роботи.

З урахуванням особливостей програми було виділено три основні напрямки тестування: правильність інформації, стабільність, сумісність.

Для визначення чи програма подає правдиву інформацію було використано комбінацію двох типів тестування, а саме тестування білої скриньки та компонентне тестування [7].

Оскільки було відомо що програма звертається до системних файлів, задля отримання інформації, тому під час тестування також відбувалось звернення до системних файлів з використанням консольних команд. Даним методом було перевірено кожен параметр програми окремо, з різною завантаженістю комп’ютера.

Результат даного тестування показав спів-падіння показників отриманих у програмі та в консолі. Отже програма виконується правильно.

Перевірка стабільності потребувала тестування програми впродовж довгого часу, з періодами різного навантаження комп’ютера [7]. Даний метод тестування показав, що після тривалої роботи програми, її поля не зависають, та продовжують оновлювати дані. Витоків пам’яті не було зафіксовано.

Перевірка сумісності має на меті перевірку на сумісність з різними операційними системами [7].

Оскільки першочергово додаток був створений для системи Ubuntu, тому додаток розрахований на архітектуру системних файлів яка використовується в системі Ubuntu. Система Ubuntu була створена на основі системи Debian, це означає що проєкт повинен працювати на всіх дистрибутивах Лінукс похідних від Debian.

Результат тестування сумісності показав роботу програми на всіх дистрибутивах сімейства Debian.

Часткова сумісність програма також передбачає що у комп’ютері буде встановлено відеокарту фірми NVIDIA, у інших випадках програма повідомляє про помилку у полях з параметрами пов’язаними із відеокартою.

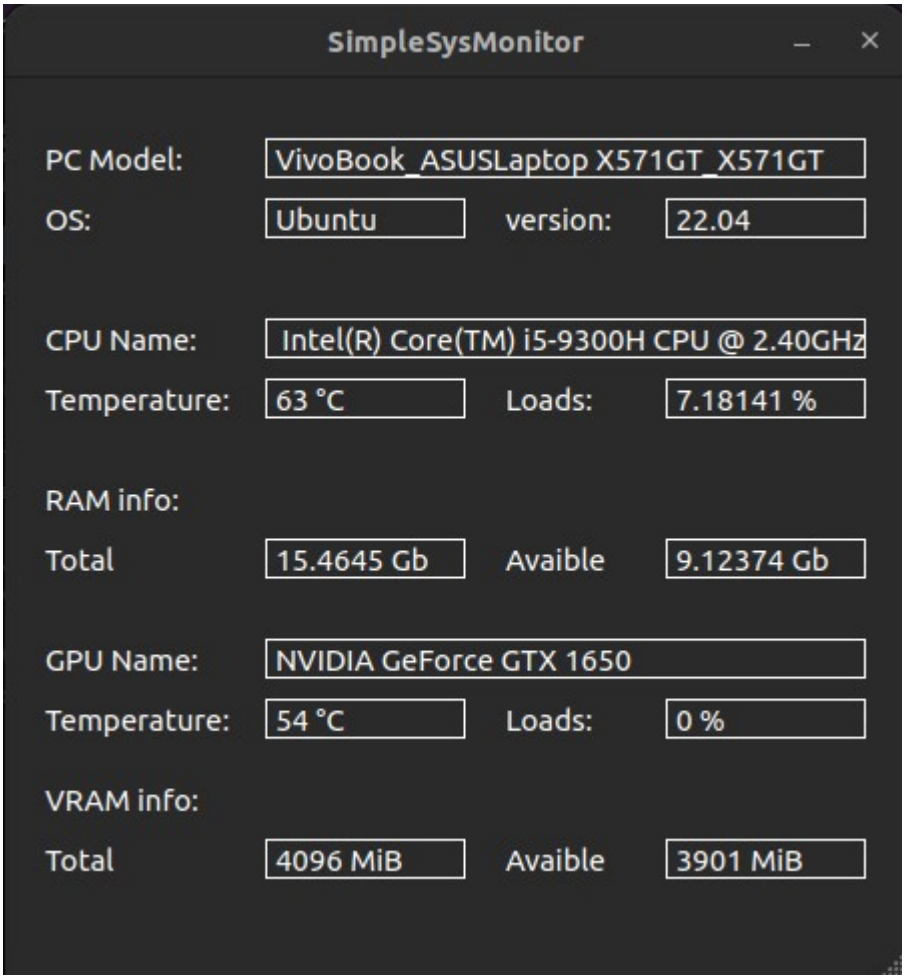

Рис. 3.25 – кінцевий вигляд програми

РОЗДІЛ 4. БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ ТА ОСНОВИ ОХОРОНИ ПРАЦІ

4.1 Долікарська допомога при опіках.

За даними ВООЗ, термічні ураження займають третє місце серед усіх інших видів травм. Щорічно у світі на кожний мільйон жителів припадає 300 осіб, які потребують лікування внаслідок термічних опіків або їх ускладнень. Загалом, 70% уражень виникають в домашніх умовах, з яких 30–40% стосуються дітей до 5 років. Полум'я (50%), гарячі пари і рідини (30%) та розпечені предмети (5%) є типовими джерелами термічних уражень. 80% опіків призводять до пошкодження кінцівок. Летальність внаслідок опіків становить 20% серед дітей і 2% серед людей похилого віку. Збільшується кількість масових опіків, важкість уражень, тривалість лікування та рівень смертності. Україна має коефіцієнт 21,3 опіків на 10 000 населення [8].

Опік, це ураження тканин, що виникає через вплив зовнішньої енергії. Він може бути термічним, хімічним, електричним або випромінювальним в залежності від джерела. Кожен випадок термічного опіку потребує оцінки його глибини, розміру та ступеня важкості, що є ключовими критеріями для визначення стратегії лікування та передбачення наслідків [8].

Глибина опіку визначається обсягом пошкоджень шкіри та суміжних тканин. Для оцінки існує чотириступенева класифікація:

- I - гіперемія та набряк шкіри;
- II - некроз та відшарування зовнішніх шарів епідермісу, формування міхурів;
- III А - частковий некроз шкіри зі збереженням мальпігієвого шару та дериватів;
- III Б - повний некроз шкіри та її дериватів;
- IV - некроз шкіри та глибоких тканин.

Опіки I–IIIA вважаються поверхневими, коли є можливість самоугоєння за рахунок збереження певних шарів шкіри. Опіки IIIБ–IV розглядаються як глибокі, де самоугоєння неможливе через повне ураження всіх шарів шкіри, тому потрібне хірургічне втручання для лікування [8].

Під площею опіку розуміють виражене у відсотках співвідношення площі опікової поверхні до загальної площі тіла.

Для поєднання класифікацій опіків з урахуванням ступеню опіку, площі та інших параметрів було створено діаграму на рисунку 4.1

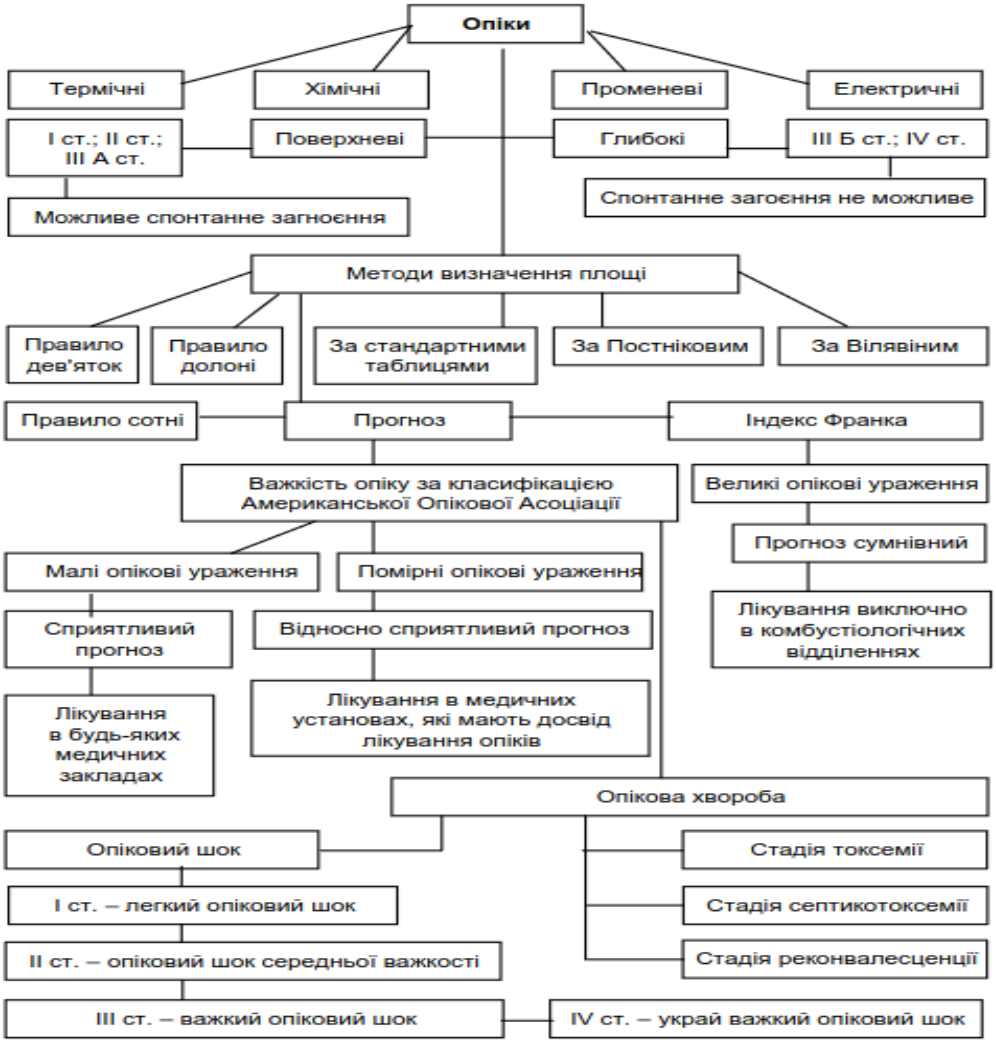

Рисунок 4.1 – Структурно-логічна схема класифікації опіків

Перша допомога при термічних опіках передбачає:

- припинення дії термічного агента;
- евакуацію з замкнутих приміщень;
- забезпечення доступу повітря;
- серцево-легеневу реанімацію;
- зняття/зрізання одягу з уражених ділянок;
- знеболення;
- накладання сухої стерильної пов’язки (опіки I–III ступеня доцільніше обробляти аерозолями).

При значних опіках часто розвивається шок. У цьому випадку обов’язково проводяться протишокові заходи.

Протишокові заходи (обкутування, тепле пиття, підняття кінцівок вище площини тіла) і транспортування потерпілого в найближчий медичний заклад. Водночас категорично не дозволяється очищати опікову поверхню від фіксованих сторонніх тіл, відривати шматки приклеєного до рани одягу, пошкоджувати булли, наносити на рану мазі, накладати ватні пов'язки, застосовувати для місцевого охолодження лід або холодну воду (нижче 10 °С) [8].

До групи пацієнтів, які потребують невідкладної стаціонарної медичної допомоги, відносять обпечених у важкому, шоковому та термінальному станах, з комбінованими ураженнями і опіками дихальних шляхів, обличчя та очей, з ознаками отруєння чадним газом, циркулярними глибокими опіками тулуба, кінцівок і шиї, значним післяопіковим набряком кінцівок [8].

Хімічні опіки виникають внаслідок контакту тканин організму з хімічними речовинами. Основними агентами кислоти (азотна, сірчана та ін.), луги (негашене вапно, їдкий натр та ін.), продукти переробки нафти (бензин, гас), солі важких металів (азотнокисле срібло, хлористий цинк) і летючі масла (корицеве, бергамотове). Окрім того, хімічні опіки можуть спричинятися термохімічними (напалм, пірогель) і бойовими отруйними(люїзит, іприт) речовинами [8].

Оскільки глибина опікового ураження і об'єм першої допомоги при хімічних пошкодженнях зумовлюються видом пошкоджуючого агенту, першочергового значення набуває встановлення характеру хімічної речовини, що спричинила опік. Основну увагу слід спрямувати на визначення вигляду і консистенції струпа та його фіксації до оточуючих тканин, фізичного стану і запаху пошкоджуючої речовини, виявленню ознак загальної інтоксикації. Кислоти, солі важких металів та летючі масла спричиняють коагуляційний некроз, що має вигляд сухого, різного кольору струпа з чіткими межами, фіксованого в тканинах, що певною мірою обмежує подальше проникнення хімічного агента глибше в тканини. Під дією лугів виникає колікваційний некроз – м'який, пухкий, з нечіткими контурами струп [8].

Головною метою першої допомоги при хімічних опіках є зменшення концентрації і термінове видалення хімічного агента з ураженої ділянки – найкращим і найдоступнішим способом є негайне інтенсивне обмивання ділянки опіку водою впродовж 5–10 хв, а також нейтралізація відповідною речовиною (при опіках кислотами – розчин соди, при опіках лугами – розчин оцтової чи іншої кислоти), а також застосування відповідних антидотів [8].

В Україні надання першої медичної допомоги при опіках регулюється державними стандартами та протоколами, затвердженими Міністерством охорони здоров'я (МОЗ) України. Основні положення щодо надання першої допомоги при опіках містяться в таких нормативних документах:

Наказ МОЗ України від 29 березня 2013 року № 221 - "Про затвердження клінічних протоколів надання медичної допомоги з хірургічних захворювань у закладах охорони здоров'я України". У цьому наказі затверджено клінічні протоколи, які регулюють надання допомоги при опіках, включаючи алгоритми надання першої допомоги та подальшого лікування.

Отож, чинні норми та стандарти надання першої медичної допомоги при опіках встановлюють чіткий план дій лікування та запобігання ускладнень у постраждалих.

4.2 Показники ефективності та заходи щодо покращенню умов та охорони праці.

Виконання сучасних завдань охорони праці повинно базуватися на системному підході, що передбачає об'єднання окремих заходів у єдину систему цілеспрямованих і постійно здійснюваних дій на всіх рівнях і етапах управління господарською діяльністю в галузі. Системний ефект досягається шляхом підвищення рівня організації виробничої системи, що сприяє її більшій інтеграції через поєднання багатьох зусиль в одне ціле. Проведення аудиту професійної безпеки робочих місць, лабораторних досліджень умов праці, оцінка технічного стану виробничого устаткування та атестація робочих місць сприятимуть покращенню охорони праці та безпеки на робочих місцях [9].

Показники ефективності та заходи щодо покращення умов і охорони праці є важливими елементами управління трудовими ресурсами та забезпечення безпеки на робочих місцях. В Україні ці питання регулюються законами та нормативними актами, які встановлюють стандарти для забезпечення безпеки праці. Основні показники ефективності та заходи щодо покращення умов і охорони праці включають:

1. Рівень виробничого травматизму:
    - Кількість нещасних випадків на виробництві.
    - Кількість смертельних випадків на виробництві.
2. Рівень професійних захворювань:
    - Кількість зареєстрованих професійних захворювань.
3. Частота та тяжкість нещасних випадків:
    - Кількість нещасних випадків на 1000 працівників.
    - Коефіцієнт тяжкості (сума днів непрацездатності на 1000 працівників).
4. Рівень виконання плану заходів з охорони праці:

- Відсоток виконаних заходів з плану охорони праці.

5. Фінансові витрати на охорону праці:
    - Загальна сума витрат на заходи з охорони праці.
    - Витрати на охорону праці на одного працівника.

Контроль, облік, аналіз, звітність, комунікація, аудит та моніторинг – це функції управління, які призначені для визначення фактичного стану та виконання завдань з охорони праці у галузі (або підрозділах). Ці функції передбачають порівняння результатів з планами та чинними нормативними актами, визначення наслідків відхилень, фактичних даних, періодичне складання статистичних звітів за встановленою формою, а також прийняття рішень щодо включення заходів з охорони праці у плани робіт на наступні роки [9].

Контроль, аудит та моніторинг сприяють виконанню посадовими особами планів робіт, дотриманню законодавства та інших нормативних актів з охорони праці. Ці функції повинні визначати ступінь виконання посадовими особами їхніх обов'язків щодо охорони праці [9].

У процесі аудиту необхідно перевірити виконання висновків попередніх аудитів й усунення встановлених недоліків. Результати аудиту надаються керівництву підприємства.

Ефективність заходів щодо поліпшення умов і охорони праці оцінюється в першу чергу за показниками соціальної ефективності, які передбачають створення умов праці, що відповідають санітарним нормам і вимогам правил безпеки [9].

Отже покращення умов і охорони праці приводить до зменшення кількості виробничих травм, загальної і професійної захворюваності; до скорочення чисельності працівників, що працюють в умовах, які не відповідають санітарно-гігієнічним нормам; зменшення кількості випадків виходу на пенсію за інвалідністю внаслідок травматизму чи професійної захворюваності; скорочення плинності кадрів через незадовільні умови праці тощо.

ВИСНОВОК

У ході виконання кваліфікаційної роботи на тему "Розробка додатку для моніторингу продуктивності комп’ютера з використанням фреймворку Qt" було досягнуто основну мету – створення ефективного інструменту для моніторингу продуктивності комп’ютерів. Розроблений додаток, названий "SimpleSysMonitor" [4], дозволяє користувачам стежити за ключовими показниками роботи комп'ютера, такими як використання процесора, пам'яті та графічного процесора.

Вибір мови програмування C++ та фреймворку Qt дозволив створити високопродуктивне програмне забезпечення з кросплатформенним інтерфейсом. Це забезпечило високу швидкодію додатку та його стабільну роботу на різних операційних системах Linux сімейства Debian.

Було досліджено різні патерни програмування, та їх модифікації. На основі кращого із них було спроектовано загальну структуру та архітектуру додатку. Основні класи та модулі були реалізовані з використанням об'єктно-орієнтованих принципів програмування. Графічний інтерфейс додатку забезпечує зручну взаємодію користувача з програмою. Також структура додатку дозволить в майбутньому додавати новий функціонал та підтримку нових платформ.

Розроблений додаток "SimpleSysMonitor" [4] має інтуїтивно зрозумілий інтерфейс, що робить його доступним для новачків. Завдяки використанню фреймворку Qt додаток може бути успішно використаний на різних операційних системах Linux сімейства Debian

Таким чином, виконана робота не лише підтверджує актуальність та доцільність розробки подібних інструментів, але й демонструє можливість створення ефективного програмного забезпечення з використанням сучасних технологій. Розроблений додаток сприятиме підвищенню продуктивності та надійності роботи комп’ютерних систем, що позитивно вплине на ефективність роботи користувачів у різних сферах діяльності.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. AIDA64 [Електронний ресурс] – Режим доступу до ресурсу: https://www.aida64.com/.

2. C++ [Електронний ресурс] – Режим доступу до ресурсу: https://en.cppreference.com/w/.

3. Qt Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://doc.qt.io/.

4. Проект додатку SimpleSysMonitor [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/Satheyn/SympleSysMonitor.

5. Патерни проектування [Електронний ресурс] – Режим доступу до ресурсу: https://refactoring.guru/uk/design-patterns.

6. UML [Електронний ресурс] – Режим доступу до ресурсу: https://docs.kde.org/trunk5/uk/umbrello/umbrello/uml-basics.html.

7. Тестування програмного забезпечення [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Тестування_програмного_забезпечення.

8. Опіки. Класифікація, клініка. Перша допомога при різних видах опіку. Ураження світловим випромінюванням ядерного вибуху. Опіки, викликанні запалювальними сумішами : метод. вказ. до практ. занять та самост. роботи студентів 3-го курсу II та IV мед. фак-тів з дисципліни "Загальна хірургія" / упоряд. В. О. Сипливий, В. В. Доценко, Г. Д. Петренко та ін. – Харків : ХНМУ, 2020. – 20 с.

9. Методичні вказівки до виконання розрахунково-графічного завдання «Техніко-економічна ефективність заходів, спрямованих на підвищення охорони праці на виробництві» з навчальної дисципліни “Організаційно-технічне забезпечення аудиту з професійної безпеки робочих місць галузі охорони праці” / уклад. Євтушенко Н. С., Твердохлєбова Н. Є. – Харків: НТУ “ХПІ”, 2019. – 28 с.

ДОДАТКИ

Додаток А

Лістинг коду основних сутностей ігрового додатку

Лістинг 1 — код mainwindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QFuture>
#include <QtConcurrent/QtConcurrent>

#include "MyFunction.h"

QT_BEGIN_NAMESPACE
namespace Ui {
class MainWindow;
}
QT_END_NAMESPACE

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();
    void static_label();
    void not_static_label();


private:
    Ui::MainWindow *ui;
};
#endif // MAINWINDOW_H
```

Лістинг 2 — код mainwindow.cpp

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);
}

MainWindow::~MainWindow()
{
    delete ui;
}
```

```
void MainWindow::static_label(){

    get_computer_model(ui->computer_model);

    get_linux_distribution_name(ui->os_name);
    get_linux_distribution_version(ui->os_version);

    ui->cpu_name->setText(CPU_info::name());

    ui->ram_total_memory->setText(RAM_info::memory_total());

    ui->gpu_name->setText(GPU_info::name());
    ui->vram_total_memory->setText(GPU_info::vram_total());

}

void MainWindow::not_static_label(){
    QFuture<void> result1 =
        QtConcurrent::run(CPU_info::update_info, ui->cpu_temperature, ui-
>cpu_loads);

    QFuture<void> result2 =
        QtConcurrent::run(RAM_info::update_info, ui->ram_avaible_memory);

    QFuture<void> result3 =
        QtConcurrent::run(GPU_info::update_info, ui->gpu_temperature, ui-
>gpu_loads, ui->vram_avaible_memory);

}
```

Лістинг 3 — код model.h

```
#ifndef MODEL_H
#define MODEL_H


#include <QLabel>
#include <QThread>
#include <qrandom.h>
#include <stdio.h>

#include <fstream>
#include <iostream>
#include <numeric>
#include <unistd.h>
#include <vector>

#include <string>
#include <sstream>
```

```
#include <unordered_map>

void get_computer_model(QLabel *label);
std::unordered_map<std::string, std::string> parse_os_release();
void get_linux_distribution_name(QLabel *label);
void get_linux_distribution_version(QLabel *label);

class CPU_info{
public:
    static QString name();

    static void update_info(QLabel *temperature_label, QLabel
*loads_label);

    static float temperature ();
    static float loads ();

private:
    static bool get_cpu_times(size_t &idle_time, size_t &total_time);
    static std::vector<size_t> get_cpu_times();
};

class RAM_info{
public:
    static void update_info(QLabel *memory_available_label);

    static QString memory_available();
    static QString memory_total();
private:
    static QString mem_info(std::string mem_name);
};

class GPU_info{
public:
    static void update_info(QLabel *temperature_label, QLabel *loads_label,
QLabel *vram_available_label);

    static QString name();

    static QString temperature();
    static QString loads ();
    static QString vram_available();
    static QString vram_total();

private:
    static std::string executeCommand(const std::string& command);
};

#endif // MODEL_H
```

Лістинг 4 — код other_function.cpp

```
#include "other_function.h"
```

```
void get_computer_model(QLabel *label) {
    std::ifstream file("/sys/devices/virtual/dmi/id/product_name");

    std::string model;
    std::getline(file, model);
    file.close();
    label->setText(QString::fromStdString(model));

}

std::unordered_map<std::string, std::string> parse_os_release() {
    std::unordered_map<std::string, std::string> info;
    std::ifstream file("/etc/os-release");

    if (!file.is_open()) {
        throw std::runtime_error("Unable to open /etc/os-release");
    }

    std::string line;
    while (std::getline(file, line)) {
        auto delimiter_pos = line.find('=');
        if (delimiter_pos == std::string::npos) {
            continue;
        }

        std::string key = line.substr(0, delimiter_pos);
        std::string value = line.substr(delimiter_pos + 1);

        // Remove surrounding quotes from the value
        if (value.size() > 1 && value.front() == '"' && value.back() ==
'"') {
            value = value.substr(1, value.size() - 2);
        }

        info[key] = value;
    }

    return info;
}

void get_linux_distribution_name(QLabel *label) {
    try {
        auto info = parse_os_release();
        label->setText(QString::fromStdString(info["NAME"]));
    } catch (const std::exception& e) {
        std::cerr << "Error: " << e.what() << std::endl;
        label->setText("Unknown");
    }
}

void get_linux_distribution_version(QLabel *label) {
    try {
        auto info = parse_os_release();
        label->setText(QString::fromStdString(info["VERSION_ID"]));
```

```
    } catch (const std::exception& e) {
        std::cerr << "Error: " << e.what() << std::endl;
        label->setText("Unknown");
    }
}
```

Лістинг 5 — код cpu_info.cpp

```
#include "other_function.h"
QString CPU_info::name(){
    std::string line;
    std::ifstream finfo("/proc/cpuinfo");
    while(getline(finfo,line)) {
        std::stringstream str(line);
        std::string itype;
        std::string info;
        if ( getline( str, itype, ':' ) && getline(str,info) &&
itype.substr(0,10) == "model name" ) {
            return QString::fromStdString(info);
        }
    }
}

void CPU_info::update_info(QLabel *temperature_label, QLabel *loads_label)
{
    while(true){

        temperature_label->setText(QString::number(temperature()) + " °C");

        loads_label->setText(QString::number(loads()) + " %");

        sleep(1);
    }
}

float CPU_info::temperature (){
    float systemp, millideg;
    FILE *thermal;
    [[maybe_unused]]int n;

    thermal = fopen("/sys/class/thermal/thermal_zone0/temp","r");
    n = fscanf(thermal,"%f",&millideg);
    fclose(thermal);
    systemp = millideg / 1000;

    return systemp;
}

float CPU_info::loads (){
    size_t previous_idle_time=0, previous_total_time=0;
    for (size_t idle_time, total_time; get_cpu_times(idle_time,
total_time); sleep(1)) {
        const float idle_time_delta = idle_time - previous_idle_time;
        const float total_time_delta = total_time - previous_total_time;
```

```
        const float utilization = 100.0 * (1.0 - idle_time_delta /
total_time_delta);

        return utilization;

        previous_idle_time = idle_time;
        previous_total_time = total_time;
    }
}

std::vector<size_t> CPU_info::get_cpu_times() {
    std::ifstream proc_stat("/proc/stat");
    proc_stat.ignore(5, ' ');
    std::vector<size_t> times;
    for (size_t time; proc_stat >> time; times.push_back(time));
    return times;
}

bool CPU_info::get_cpu_times(size_t &idle_time, size_t &total_time) {
    const std::vector<size_t> cpu_times = get_cpu_times();
    if (cpu_times.size() < 4)
        return false;
    idle_time = cpu_times[3];
    total_time = std::accumulate(cpu_times.begin(), cpu_times.end(), 0);
    return true;
}
```

Лістинг 6 — код gpu_info.cpp

```
#include"other_function.h"

void GPU_info::update_info(QLabel *temperature_label, QLabel *loads_label,
QLabel *vram_available_label)
{
    while(true){
        temperature_label->setText(temperature());
        loads_label->setText(loads());
        vram_available_label->setText(vram_available());
        sleep(1);
    }
}

std::string GPU_info::executeCommand(const std::string& command) {
    std::array<char, 128> buffer;
    std::string result;
    std::unique_ptr<FILE, decltype(&pclose)> pipe(popen(command.c_str(),
"r"), pclose);

    if (!pipe) {
        return "ERROR ";
    }

    while (fgets(buffer.data(), buffer.size(), pipe.get()) != nullptr) {
        result += buffer.data();
    }
```

```
    return result;
}

QString GPU_info::name() {
    std::string command = "nvidia-smi --query-gpu=name --
format=csv,noheader";

    QString result = QString::fromStdString(executeCommand(command));
    result.chop(1);
    return result;
}


QString GPU_info::temperature() {
    std::string command = "nvidia-smi --query-gpu=temperature.gpu --
format=csv,noheader";

    QString result = QString::fromStdString(executeCommand(command));
    result.chop(1);
    return result;
}

QString GPU_info::loads() {
    std::string command = "nvidia-smi --query-gpu=utilization.gpu --
format=csv,noheader";

    QString result = QString::fromStdString(executeCommand(command));
    result.chop(1);
    return result;
}

QString GPU_info::vram_available() {
    std::string command = "nvidia-smi --query-gpu=memory.free --
format=csv,noheader";

    QString result = QString::fromStdString(executeCommand(command));
    result.chop(1);
    return result;
}

QString GPU_info::vram_total() {
    std::string command = "nvidia-smi --query-gpu=memory.total --
format=csv,noheader";

    QString result = QString::fromStdString(executeCommand(command));
    result.chop(1);
    return result;
}
```

Лістинг 7 — код ram_info.cpp

```
#include"other_function.h"
```

```
void RAM_info::update_info(QLabel *memory_available_label)
{
    while(true){
        memory_available_label->setText(memory_available());
        sleep(1);
    }
}

QString RAM_info::memory_available()
{
    return mem_info("MemAvailable");
}

QString RAM_info::memory_total()
{
    return mem_info("MemTotal");
}

QString RAM_info::mem_info(std::string mem_name){
    double mem = -1;
    std::ifstream meminfo("/proc/meminfo");
    std::string line;
    while (std::getline(meminfo, line))
    {
        if (line.find(mem_name+":") != std::string::npos)
        {
            const std::size_t firstWhiteSpacePos = line.find_first_of(' ');
            const std::size_t firstNonWhiteSpaceChar =
line.find_first_not_of(' ', firstWhiteSpacePos);
            const std::size_t nextWhiteSpace = line.find_first_of(' ',
firstNonWhiteSpaceChar);
            const std::size_t numChars = nextWhiteSpace -
firstNonWhiteSpaceChar;
            const std::string memAvailableStr =
line.substr(firstNonWhiteSpaceChar, numChars);
            mem = std::stoll(memAvailableStr);
            break;
        }
    }

    return QString::number(mem/1024/1024)+" Gb";
}
```

Лістинг 8 — код main.cpp

```
#include "mainwindow.h"

#include <QApplication>
#include <QTimer>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();
```

```
    QTimer::singleShot(0, &w, &MainWindow::static_label);
    QTimer::singleShot(0, &w, &MainWindow::not_static_label);
    return a.exec();
}
```

Лістинг 9 — код mainwindow.ui

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
 <class>MainWindow</class>
 <widget class="QMainWindow" name="MainWindow">
  <property name="geometry">
   <rect>
    <x>0</x>
    <y>0</y>
    <width>450</width>
    <height>450</height>
   </rect>
  </property>
  <property name="minimumSize">
   <size>
    <width>450</width>
    <height>450</height>
   </size>
  </property>
  <property name="maximumSize">
   <size>
    <width>450</width>
    <height>450</height>
   </size>
  </property>
  <property name="windowTitle">
   <string>SimpleSysMonitor</string>
  </property>
  <property name="autoFillBackground">
   <bool>false</bool>
  </property>
  <widget class="QWidget" name="centralwidget">
   <widget class="QLabel" name="label_2">
    <property name="geometry">
     <rect>
      <x>20</x>
      <y>150</y>
      <width>100</width>
      <height>20</height>
     </rect>
    </property>
    <property name="text">
     <string>Temperature: </string>
    </property>
   </widget>
   <widget class="QLabel" name="label_3">
    <property name="geometry">
     <rect>
      <x>250</x>
      <y>150</y>
```

```
    <width>80</width>
    <height>20</height>
   </rect>
  </property>
  <property name="text">
   <string>Loads: </string>
  </property>
 </widget>
 <widget class="QLabel" name="cpu_temperature">
  <property name="geometry">
   <rect>
    <x>130</x>
    <y>150</y>
    <width>100</width>
    <height>20</height>
   </rect>
  </property>
  <property name="frameShape">
   <enum>QFrame::Box</enum>
  </property>
  <property name="text">
   <string/>
  </property>
 </widget>
 <widget class="QLabel" name="cpu_loads">
  <property name="geometry">
   <rect>
    <x>330</x>
    <y>150</y>
    <width>100</width>
    <height>20</height>
   </rect>
  </property>
  <property name="frameShape">
   <enum>QFrame::Box</enum>
  </property>
  <property name="text">
   <string/>
  </property>
  <property name="margin">
   <number>0</number>
  </property>
 </widget>
 <widget class="QLabel" name="label_4">
  <property name="geometry">
   <rect>
    <x>20</x>
    <y>120</y>
    <width>100</width>
    <height>20</height>
   </rect>
  </property>
  <property name="text">
   <string>CPU Name:</string>
  </property>
 </widget>
```

```
<widget class="QLabel" name="cpu_name">
 <property name="geometry">
  <rect>
   <x>130</x>
   <y>120</y>
   <width>300</width>
   <height>20</height>
  </rect>
 </property>
 <property name="frameShape">
  <enum>QFrame::Box</enum>
 </property>
 <property name="text">
  <string/>
 </property>
</widget>
<widget class="QLabel" name="label_5">
 <property name="geometry">
  <rect>
   <x>20</x>
   <y>30</y>
   <width>100</width>
   <height>20</height>
  </rect>
 </property>
 <property name="text">
  <string>PC Model: </string>
 </property>
</widget>
<widget class="QLabel" name="computer_model">
 <property name="geometry">
  <rect>
   <x>130</x>
   <y>30</y>
   <width>300</width>
   <height>20</height>
  </rect>
 </property>
 <property name="frameShape">
  <enum>QFrame::Box</enum>
 </property>
 <property name="text">
  <string/>
 </property>
</widget>
<widget class="QLabel" name="label_7">
 <property name="geometry">
  <rect>
   <x>20</x>
   <y>200</y>
   <width>100</width>
   <height>20</height>
  </rect>
 </property>
 <property name="text">
  <string>RAM info:</string>
```

```
 </property>
</widget>
<widget class="QLabel" name="label_8">
 <property name="geometry">
  <rect>
   <x>20</x>
   <y>230</y>
   <width>100</width>
   <height>20</height>
  </rect>
 </property>
 <property name="text">
  <string>Total</string>
 </property>
</widget>
<widget class="QLabel" name="ram_total_memory">
 <property name="geometry">
  <rect>
   <x>130</x>
   <y>230</y>
   <width>100</width>
   <height>20</height>
  </rect>
 </property>
 <property name="font">
  <font>
   <stylestrategy>PreferDefault</stylestrategy>
  </font>
 </property>
 <property name="frameShape">
  <enum>QFrame::Box</enum>
 </property>
 <property name="lineWidth">
  <number>1</number>
 </property>
 <property name="text">
  <string/>
 </property>
</widget>
<widget class="QLabel" name="label_9">
 <property name="geometry">
  <rect>
   <x>250</x>
   <y>230</y>
   <width>80</width>
   <height>20</height>
  </rect>
 </property>
 <property name="text">
  <string>Avaible</string>
 </property>
</widget>
<widget class="QLabel" name="ram_avaible_memory">
 <property name="geometry">
  <rect>
   <x>330</x>
```

```
    <y>230</y>
    <width>100</width>
    <height>20</height>
   </rect>
  </property>
  <property name="frameShape">
   <enum>QFrame::Box</enum>
  </property>
  <property name="text">
   <string/>
  </property>
 </widget>
 <widget class="QLabel" name="label_10">
  <property name="geometry">
   <rect>
    <x>20</x>
    <y>280</y>
    <width>80</width>
    <height>20</height>
   </rect>
  </property>
  <property name="text">
   <string>GPU Name:</string>
  </property>
 </widget>
 <widget class="QLabel" name="gpu_name">
  <property name="geometry">
   <rect>
    <x>130</x>
    <y>280</y>
    <width>300</width>
    <height>20</height>
   </rect>
  </property>
  <property name="frameShape">
   <enum>QFrame::Box</enum>
  </property>
  <property name="text">
   <string/>
  </property>
 </widget>
 <widget class="QLabel" name="gpu_temperature">
  <property name="geometry">
   <rect>
    <x>130</x>
    <y>310</y>
    <width>100</width>
    <height>20</height>
   </rect>
  </property>
  <property name="frameShape">
   <enum>QFrame::Box</enum>
  </property>
  <property name="text">
   <string/>
  </property>
```

```
</widget>
<widget class="QLabel" name="label_12">
 <property name="geometry">
  <rect>
   <x>20</x>
   <y>310</y>
   <width>100</width>
   <height>20</height>
  </rect>
 </property>
 <property name="text">
  <string>Temperature: </string>
 </property>
</widget>
<widget class="QLabel" name="os_name">
 <property name="geometry">
  <rect>
   <x>130</x>
   <y>60</y>
   <width>100</width>
   <height>20</height>
  </rect>
 </property>
 <property name="frameShape">
  <enum>QFrame::Box</enum>
 </property>
 <property name="text">
  <string/>
 </property>
</widget>
<widget class="QLabel" name="label_14">
 <property name="geometry">
  <rect>
   <x>20</x>
   <y>60</y>
   <width>100</width>
   <height>20</height>
  </rect>
 </property>
 <property name="text">
  <string>OS: </string>
 </property>
</widget>
<widget class="QLabel" name="os_version">
 <property name="geometry">
  <rect>
   <x>330</x>
   <y>60</y>
   <width>100</width>
   <height>20</height>
  </rect>
 </property>
 <property name="frameShape">
  <enum>QFrame::Box</enum>
 </property>
 <property name="text">
```

```
  <string/>
 </property>
</widget>
<widget class="QLabel" name="label_15">
 <property name="geometry">
  <rect>
   <x>250</x>
   <y>60</y>
   <width>80</width>
   <height>20</height>
  </rect>
 </property>
 <property name="text">
  <string>version: </string>
 </property>
</widget>
<widget class="QLabel" name="label_11">
 <property name="geometry">
  <rect>
   <x>20</x>
   <y>350</y>
   <width>100</width>
   <height>20</height>
  </rect>
 </property>
 <property name="text">
  <string>VRAM info:</string>
 </property>
</widget>
<widget class="QLabel" name="label_13">
 <property name="geometry">
  <rect>
   <x>250</x>
   <y>380</y>
   <width>80</width>
   <height>20</height>
  </rect>
 </property>
 <property name="text">
  <string>Avaible</string>
 </property>
</widget>
<widget class="QLabel" name="vram_avaible_memory">
 <property name="geometry">
  <rect>
   <x>330</x>
   <y>380</y>
   <width>100</width>
   <height>20</height>
  </rect>
 </property>
 <property name="frameShape">
  <enum>QFrame::Box</enum>
 </property>
 <property name="text">
  <string/>
```

```
 </property>
</widget>
<widget class="QLabel" name="label_16">
 <property name="geometry">
  <rect>
   <x>20</x>
   <y>380</y>
   <width>100</width>
   <height>20</height>
  </rect>
 </property>
 <property name="text">
  <string>Total</string>
 </property>
</widget>
<widget class="QLabel" name="vram_total_memory">
 <property name="geometry">
  <rect>
   <x>130</x>
   <y>380</y>
   <width>100</width>
   <height>20</height>
  </rect>
 </property>
 <property name="font">
  <font>
   <stylestrategy>PreferDefault</stylestrategy>
  </font>
 </property>
 <property name="frameShape">
  <enum>QFrame::Box</enum>
 </property>
 <property name="lineWidth">
  <number>1</number>
 </property>
 <property name="text">
  <string/>
 </property>
</widget>
<widget class="QLabel" name="gpu_loads">
 <property name="geometry">
  <rect>
   <x>330</x>
   <y>310</y>
   <width>100</width>
   <height>20</height>
  </rect>
 </property>
 <property name="frameShape">
  <enum>QFrame::Box</enum>
 </property>
 <property name="text">
  <string/>
 </property>
 <property name="margin">
  <number>0</number>
```

```
    </property>
   </widget>
   <widget class="QLabel" name="label_6">
    <property name="geometry">
     <rect>
      <x>250</x>
      <y>310</y>
      <width>80</width>
      <height>20</height>
     </rect>
    </property>
    <property name="text">
     <string>Loads: </string>
    </property>
   </widget>
  </widget>
  <widget class="QMenuBar" name="menubar">
   <property name="geometry">
    <rect>
     <x>0</x>
     <y>0</y>
     <width>450</width>
     <height>22</height>
    </rect>
   </property>
  </widget>
  <widget class="QStatusBar" name="statusbar"/>
 </widget>
 <resources/>
 <connections/>
</ui>
```

Додаток Б – Диск