Розробка текстового опису дизайну програмного рішення, який включає:

a. високорівневий опис реалізації майбутнього програмного рішення згідно

розроблених вимог;

b. структуру програмного рішення; архітектуру класів, їх призначення і взаємодію;

опис функцій;

c. першу спробу реалізації каркасу програмного рішення з ієрархіями класів і їх

членами, але без тіла методів (наприклад розробка \*.h файлів); цей каркас нівякому разі

не розцінюється як остаточний і однозначно буде мінятися пізніше при реалізації програмного рішення.

**1. Високорівневий опис реалізації майбутнього програмного рішення**

Програмне рішення є багатопотоковою графічною програмою, яка моніторить і відображає в реальному часі використання ресурсів комп’ютера, таких як процесор (CPU), оперативна пам’ять (RAM), графічний процесор (GPU), дисковий простір, мережева активність та системна інформація. Основні функціональні вимоги:

1. **Моніторинг ресурсів:**

- Збір даних про використання CPU (загальне та по ядрах), RAM, GPU (якщо доступно), дискового простору, I/O операцій диска, мережевої активності та часу роботи системи.

- Оновлення даних кожні 3 секунди для зменшення навантаження на систему.

- Зберігання історії даних (до 60 точок) для побудови графіків.

2. **Візуалізація даних:**

- Графічний інтерфейс користувача (GUI) на основі бібліотеки Tkinter.

- Використання вкладок (Notebook) для відображення інформації про CPU, RAM, GPU, диск, мережу та системну інформацію.

- Графіки, побудовані за допомогою Matplotlib, для відображення динаміки використання ресурсів.

- Таблиці (Treeview) для відображення списків процесів і мережевих з’єднань.

3. **Оповіщення та рекомендації:**

- Сповіщення користувача при перевищенні порогових значень (наприклад, CPU > 90%, RAM > 90%, температура GPU > 85°C тощо).

- Рекомендації для зменшення навантаження (наприклад, закриття ресурсоємних процесів).

4. **Управління процесами:**

- Можливість завершення вибраних процесів через GUI.

- Відображення детальної інформації про процеси (PID, ім’я, використання пам’яті, CPU).

5. **Експорт даних:**

- Автоматичний експорт звітів кожні 5 хвилин у текстовий файл.

- Ручний експорт даних за вибраний період часу.

- Копіювання системної інформації в буфер обміну.

6. **Логування:**

- Збереження діагностичних повідомлень у файл `monitor.log` для відлагодження.

7. **Платформна сумісність:**

- Підтримка Windows і Linux (з частковими обмеженнями для macOS).

- Використання бібліотек `psutil`, `GPUtil` (опціонально), `wmi` (Windows), `pySMART` (опціонально) для доступу до апаратних даних.

Програма використовує багатопотоковість (модуль `threading`) для паралельного моніторингу різних компонентів, що забезпечує плавне оновлення GUI без блокування інтерфейсу. Для зменшення виведення в консоль перенаправляється `stdout` у `/dev/null`.

**2. Структура програмного рішення**

Програмне рішення реалізовано в єдиному класі `SystemMonitor`, який інкапсулює всю логіку моніторингу, GUI та управління потоками. Структура включає:

2.1. Основний клас

- \*\*SystemMonitor\*\*: Відповідає за ініціалізацію GUI, запуск потоків моніторингу, обробку подій та завершення програми.

2.2. Модулі моніторингу

- Кожен компонент (CPU, RAM, GPU, диск, мережа, час роботи) має окремий метод моніторингу (`monitor\_cpu`, `monitor\_ram` тощо), який працює в окремому потоці.

- Методи оновлення GUI (`update\_cpu\_gui`, `update\_ram\_gui` тощо) викликаються через `root.after` для безпечного доступу до Tkinter.

2.3. GUI компоненти

- Використовується `ttk.Notebook` для організації вкладок.

- Кожна вкладка містить:

- Мітки (`ttk.Label`) для відображення поточних значень.

- Графіки (`FigureCanvasTkAgg`) для візуалізації історії.

- Таблиці (`ttk.Treeview`) для списків процесів або драйверів.

- Кнопки (`ttk.Button`) для взаємодії (наприклад, завершення процесу або експорт даних).

2.4. Архітектура класів, їх призначення та взаємодія

- Клас `SystemMonitor`:

- Призначення: Центральний клас, який координує моніторинг, відображення та взаємодію з користувачем.

- Атрибути:

- `root`: Об’єкт Tkinter для GUI.

- `stop\_event`: Об’єкт `threading.Event` для зупинки потоків.

- Історичні дані (`cpu\_usage\_history`, `ram\_usage\_history` тощо) для графіків.

- Порогові значення (`cpu\_threshold`, `ram\_threshold` тощо) для сповіщень.

- Об’єкти GUI (мітки, графіки, таблиці).

- Методи:

- Ініціалізація: `setup\_gui`, `start\_monitoring`.

- Моніторинг: `monitor\_cpu`, `monitor\_ram`, `monitor\_gpu`, `monitor\_disk`, `monitor\_network`, `monitor\_uptime`.

- Оновлення GUI: `update\_cpu\_gui`, `update\_ram\_gui` тощо.

- Управління процесами: `kill\_process`, `update\_process\_list`, `update\_net\_process\_list`.

- Експорт даних: `export\_data`, `manual\_export\_data`, `auto\_export\_data`.

- Обробка подій: `on\_resize`, `on\_closing`, `sort\_treeview`, `copy\_system\_info`.

- Взаємодія:

- Клас використовує бібліотеки `psutil`, `GPUtil`, `wmi`, `pySMART` для збору даних.

- Потоки моніторингу періодично викликають методи оновлення GUI через `root.after` для уникнення конфліктів із головним потоком Tkinter.

- Дані з потоків моніторингу записуються в історичні списки, які використовуються для побудови графіків.

- Користувацькі дії (сортування таблиць, завершення процесів, експорт) обробляються через методи, прив’язані до подій GUI.

2.5. Опис функцій

- `\_\_init\_\_`: Ініціалізує GUI, історичні списки, порогові значення та перенаправляє `stdout`.

- `setup\_gui`: Створює вкладки, мітки, графіки, таблиці та кнопки.

- `start\_monitoring`: Запускає потоки моніторингу та планує автоматичний експорт.

- `monitor\_\*` (наприклад, `monitor\_cpu`)*: Збирають дані з апаратних компонентів і викликають методи оновлення GUI.*

*-* `update\_ \_gui` (наприклад, `update\_cpu\_gui`): Оновлюють мітки, графіки та таблиці, перевіряють порогові значення.

- `kill\_process`: Завершує вибраний процес після підтвердження користувача.

- `export\_data`: Створює текстовий звіт із даними системи та історії.

- `on\_closing`: Координує завершення програми, зупиняючи потоки та закриваючи GUI.

- `sort\_treeview` / `sort\_net\_treeview`: Сортують таблиці за вибраним стовпцем.

- `copy\_system\_info`: Копіює системну інформацію в буфер обміну.

Зображення, що містить текст, знімок екрана, Шрифт, чорно-білий

Вміст, створений ШІ, може бути неправильним.

3. Перша спроба реалізації каркасу програмного рішення

Нижче наведено каркас у вигляді псевдокоду, написаний на мові програмування Python (замість \*.h файлів, оскільки код написаний на Python). Цей каркас коду є не остаточним і може змінюватися.

# system\_monitor.py

import tkinter as tk

from threading import Event

from typing import List, Dict, Any

import matplotlib.figure

class ISystemMonitor(ABC):

"""Абстрактний інтерфейс для моніторингу системи."""

def setup\_gui(self) -> None:

"""Ініціалізує графічний інтерфейс."""

pass

def start\_monitoring(self) -> None:

"""Запускає потоки моніторингу."""

pass

def monitor\_cpu(self) -> None:

"""Моніторить використання CPU."""

pass

def update\_cpu\_gui(self, total\_cpu: float, cpu\_percent: List[float]) -> None:

"""Оновлює GUI для CPU."""

pass

def monitor\_ram(self) -> None:

"""Моніторить використання RAM."""

pass

def update\_ram\_gui(self, ram: Any) -> None:

"""Оновлює GUI для RAM."""

pass

def monitor\_gpu(self) -> None:

"""Моніторить використання GPU."""

pass

def update\_gpu\_gui(self, gpu\_usage: float, gpu\_memory\_used: float,

gpu\_memory\_total: float, gpu\_memory\_percent: float,

gpu\_temp: float) -> None:

"""Оновлює GUI для GPU."""

pass

def monitor\_disk(self) -> None:

"""Моніторить дисковий простір та I/O."""

pass

def update\_disk\_gui(self, disk: Any, read\_speed: float, write\_speed: float,

disk\_temp: str, disk\_health: str) -> None:

"""Оновлює GUI для диска."""

pass

def monitor\_network(self) -> None:

"""Моніторить мережеву активність."""

pass

def update\_network\_gui(self, download\_speed: float, upload\_speed: float) -> None:

"""Оновлює GUI для мережі."""

pass

def monitor\_uptime(self) -> None:

"""Моніторить час роботи системи."""

pass

def update\_uptime\_gui(self, uptime\_seconds: int, days: int, hours: int, minutes: int) -> None:

"""Оновлює GUI для часу роботи."""

pass

def kill\_process(self) -> None:

"""Завершує вибраний процес."""

pass

def export\_data(self, time\_range\_minutes: int = None) -> str:

"""Експортує дані у файл."""

pass

def on\_closing(self) -> None:

"""Обробляє закриття програми."""

pass

class SystemMonitor(ISystemMonitor):

"""Реалізація системи моніторингу."""

def \_\_init\_\_(self):

self.root: tk.Tk = tk.Tk()

self.stop\_event: Event = Event()

self.cpu\_usage\_history: List[List[float]] = []

self.ram\_usage\_history: List[float] = []

self.gpu\_usage\_history: List[float] = []

self.disk\_io\_history\_read: List[float] = []

self.net\_download\_history: List[float] = []

self.process\_data: Dict[int, Dict[str, Any]] = {}

self.alert\_log: List[str] = []

self.cpu\_threshold: float = 90.0

self.cpu\_label: tk.Label = None

self.cpu\_fig: matplotlib.figure.Figure = None

self.after\_ids: List[int] = []

def setup\_gui(self) -> None:

pass

def start\_monitoring(self) -> None:

pass

def monitor\_cpu(self) -> None:

pass

def update\_cpu\_gui(self, total\_cpu: float, cpu\_percent: List[float]) -> None:

pass

def monitor\_ram(self) -> None:

pass

def update\_ram\_gui(self, ram: Any) -> None:

pass

def monitor\_gpu(self) -> None:

pass

def update\_gpu\_gui(self, gpu\_usage: float, gpu\_memory\_used: float,

gpu\_memory\_total: float, gpu\_memory\_percent: float,

gpu\_temp: float) -> None:

pass

def monitor\_disk(self) -> None:

pass

def update\_disk\_gui(self, disk: Any, read\_speed: float, write\_speed: float,

disk\_temp: str, disk\_health: str) -> None:

pass

def monitor\_network(self) -> None:

pass

def update\_network\_gui(self, download\_speed: float, upload\_speed: float) -> None:

pass

def monitor\_uptime(self) -> None:

pass

def update\_uptime\_gui(self, uptime\_seconds: int, days: int, hours: int, minutes: int) -> None:

pass

def kill\_process(self) -> None:

pass

def export\_data(self, time\_range\_minutes: int = None) -> str:

pass

def on\_closing(self) -> None:

pass