HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

KHOA CÔNG NGHỆ THÔNG TIN

BỘ MÔN LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

BÁO CÁO BÀI TẬP LỚN

LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

ĐỀ TÀI: PHÁT TRIỂN ỨNG DỤNG QUẢN LÝ THỜI GIAN VÀ CÔNG VIỆC

Nhóm lớp: 16

Nhóm BTL: 11

Thành viên: Bùi Quang Vinh - B23DCCN926

Phan Thanh Bình - B23DCCN087

Trần Bá Hoàng - B22DCKH044

Nguyễn Thanh Phong - B23DCCN646

Đàm Thuận Toàn - B23DCCN828

Hà Nội – 2025

Mục Lục

BÁO CÁO TỔNG HỢP
PHÁT TRIỂN ỨNG DỤNG
QUẢN LÝ THỜI GIAN VÀ CÔNG VIỆC

I. Giới thiệu dự án

- Dự án Phát triển Ứng dụng quản lý thời gian và công việc được thiết kế nhằm giúp người dùng tổ chức và theo dõi các công việc hàng ngày một cách hiệu quả. Ứng dụng cho phép người dùng thêm, xóa, và quản lý các nhiệm vụ theo từng ngày, với khả năng thiết lập thời gian thực hiện công việc, mức độ quan trọng của từng công việc kết hợp với bộ đếm thời gian Pomodoro luân phiên giữa các phiên làm việc và nghỉ.

- Mục tiêu chính của dự án là cung cấp một công cụ tiện ích hỗ trợ người dùng tối ưu hóa thời gian và năng suất làm việc, đồng thời giảm thiểu căng thẳng trong việc quản lý công việc hàng ngày. Ứng dụng cũng bao gồm các chức năng theo dõi tiến độ hoàn thành công việc thông qua thanh tiến độ, giúp người dùng dễ dàng nhận biết trạng thái hiện tại của các nhiệm vụ và điều chỉnh kế hoạch của mình một cách hợp lý.

II. Kỹ thuật, thư viện ngôn ngữ, công nghệ sử dụng

1. Ngôn ngữ sử dụng:

- Java: ngôn ngữ lập trình hướng đối tượng, dùng để xây dựng toàn bộ logic và cấu trúc dữ liệu của ứng dụng.

2. Giao diện người dùng:

- JavaFX: nền tảng xây dựng giao diện đồ họa chính. Ứng dụng sử dụng các thành phần như Scene, Stage, Pane, VBox, HBox, ListView, ProgressBar, Dialog để tạo bố cục và giao diện trực quan.

- FXML: tách phần định nghĩa giao diện ra khỏi mã nguồn xử lý, giúp tổ chức file rõ ràng.

- CSS JavaFX: dùng để tùy chỉnh màu sắc, theme, hình nền, và cải thiện trải nghiệm người dùng.

- ControlsFX: thư viện hỗ trợ các thành phần giao diện mở rộng như thông báo popup, dialog nâng cao.

- Media / AudioClip: hỗ trợ phát âm thanh báo hiệu khi kết thúc phiên làm việc hoặc nghỉ trong Pomodoro.

3. Xử lý logic:

- Java OOP: sử dụng lớp, đối tượng, đóng gói, kế thừa và các nguyên lý OOP để xây dựng các thành phần: Calendar, Week, Day, Task, Music.

- Date/Time API (LocalDate, LocalTime, Duration, DateTimeFormatter): xử lý ngày, giờ, thời lượng và định dạng thời gian.

- JavaFX Timeline và ScheduledService:

  - Timeline cập nhật đếm giờ Pomodoro theo từng giây.
  - ScheduledService xử lý chạy ngầm, kiểm tra các task sắp đến hạn và kích hoạt thông báo định kỳ.

- ObservableList: đảm bảo dữ liệu thay đổi ở Model được cập nhật tức thời lên giao diện.

- Java Collection: dùng để sắp xếp, cập nhật và lọc danh sách task.

4. Lưu trữ dữ liệu:

- Java Serializable: tuần tự hóa các đối tượng Calendar, Week, Day, Task thành file .dat.

- Lưu trữ theo tuần: mỗi tuần được lưu vào một tệp riêng, giúp quản lý dữ liệu dễ dàng và giảm xung đột.

- ArrayList + Serializable: dùng để lưu danh sách task mỗi ngày khi ghi file.

5. Công cụ quản lý dự án:

- Maven: quản lý thư viện, cấu trúc dự án và quá trình build.

6. Mô hình MVC:

- Mô hình MVC (Model-View-Controller) là một mô hình thiết kế phần mềm phổ biến trong phát triển ứng dụng. Mô hình này tách biệt các thành phần chính của ứng dụng, bao gồm dữ liệu (Model), giao diện người dùng (View) và logic xử lý (Controller). Mô hình MVC giúp tăng tính mở rộng, bảo trì và quản lý mã nguồn dễ dàng hơn.

- Các thành phần của mô hình MVC:

❖ **View (Giao diện người dùng)**

View sẽ là phần giao diện người dùng, nơi người dùng có thể tương tác với ứng dụng. View sẽ nhận dữ liệu từ Model và hiển thị chúng theo cách thân thiện với người dùng.

- **Model (Mô hình)**

  Model trong ứng dụng quản lý công việc sẽ quản lý tất cả các dữ liệu và logic nghiệp vụ liên quan đến các công việc.

- **Controller (Bộ điều khiển)**

  Controller sẽ là cầu nối giữa Model và View, xử lý các yêu cầu từ người dùng và điều phối các thao tác.

- **Utils(Tiện ích hỗ trợ)**

  Utils là nơi chứa các lớp hỗ trợ dùng chung trong ứng dụng, không thuộc trực tiếp vào Model, View hoặc Controller. Các lớp này giúp đơn giản hóa mã nguồn, tránh lặp lại logic và hỗ trợ các chức năng phụ như thiết lập giao diện.

III. Cấu trúc dự án

- Khi ứng dụng khởi động, nó tạo một đối tượng calendar để lưu trữ và quản lý lịch trình theo từng tuần. Dữ liệu của các tuần được tổ chức trong các đối tượng week, mỗi week chứa các thông tin công việc theo từng ngày. Các đối tượng week được lưu trữ trong một cấu trúc map để truy cập nhanh chóng theo thời gian.

- Khi ứng dụng mở, các tuần đã lưu trước đó sẽ được tải lên từ bộ nhớ, giúp người dùng tiếp tục công việc và xem lại các công việc đã lưu từ các phiên trước. Khi đóng ứng dụng, toàn bộ thông tin của calendar, bao gồm các tuần và các công việc chi tiết bên trong, sẽ được lưu lại để đảm bảo dữ liệu không bị mất.

- Ứng dụng cho phép người dùng dễ dàng thêm, sửa, xóa các công việc theo ngày và tuần, đồng thời tự động sắp xếp lịch trình để hiển thị một cách trực quan và dễ theo dõi.

Dưới đây là các phần chính của dự án:

1. App và AppManager

Trong mô hình MVC với JavaFX, lớp App kế thừa từ Application là nơi khởi động ứng dụng. Lớp này thiết lập Stage chính, tạo, hiển thị giao diện ban đầu của ứng dụng và khởi tạo đối tượng calendar mới khi mở ứng dụng cũng như gọi phương thức trong calendar để lưu các đối tượng lại khi đóng ứng dụng.

```java
package taskmanagement;

import ...

public class App extends Application {

    @Override
    public void init() {
        AppManager.calendar = new Calendar();
        // Khởi động dịch vụ chạy ngầm ngay khi ứng dụng bắt đầu
        AppManager.startStatusService();
    }

    @Override
    public void start(Stage stage) throws IOException {
        AppManager.stage = stage;
        FXMLLoader fxmlLoader = new FXMLLoader(App.class.getResource( name: "/Fxml/main-window.fxml"));
        AppManager.mainWindow = new Scene(fxmlLoader.load());
        AppManager.switchToMainWindow();
    }

    @Override
    public void stop() throws IOException {
        // Tắt dịch vụ chạy ngầm trước khi đóng ứng dụng
        AppManager.stopStatusService();
        // Lưu dữ liệu tuần sau khi đã tắt service
        AppManager.calendar.saveWeeksToFile();
    }

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

AppManager được sử dụng để chuyển đổi giữa các cửa sổ làm việc của ứng dụng. Đồng thời ở đây cũng cung cấp các biến toàn cục để quản lý tập trung instance hiện tại của model và quản lý việc bật tắt dịch vụ chạy ngầm.

```java
public class AppManager {
    public static Stage stage;
    public static Scene mainWindow;
    public static Calendar calendar;
    public static Day selectedDay;
    public static Task selectedTask;

    // Giữ duy nhất instance service
    private static final StatusUpdateService statusUpdateService = new StatusUpdateService();

    // Khởi động service MỘT LẦN
    public static void startStatusService() {
        if (statusUpdateService.getState() == Worker.State.READY) {
            statusUpdateService.start();
        } else if (statusUpdateService.getState() == Worker.State.CANCELLED) {
            statusUpdateService.reset();
            statusUpdateService.start();
        }
    }

    // Dừng service MỘT LẦN
    public static void stopStatusService() {
        if (statusUpdateService.isRunning()) {
            statusUpdateService.cancel();
        }
    }

    public static void switchToDayWindow() throws IOException {
        startStatusService();
        loadAndSetScene( fxmlPath: "/Fxml/day-window.fxml");
    }

    public static void switchToPomodoroWindow() throws IOException {
        stopStatusService();
        loadAndSetScene( fxmlPath: "/Fxml/pomodoro-window.fxml");
    }

    public static void switchToMainWindow() {
        stopStatusService();

        // Lưu lại kích thước hiện tại của stage
        double width = stage.getWidth();
        double height = stage.getHeight();
        boolean isMaximized = stage.isMaximized();

        // Gán lại scene chính
        stage.setScene(mainWindow);

        // Giữ nguyên kích thước / trạng thái phóng to
        stage.setWidth(width);
        stage.setHeight(height);
        stage.setMaximized(isMaximized);

        stage.show();
    }
```

```
    private static void loadAndSetScene(String fxmlPath) throws IOException {
        double width = 800;  // fallback mặc định
        double height = 600;
        boolean isMaximized = stage.isMaximized();

        // Nếu đang có scene hiện tại, lấy kích thước thực tế từ đó
        if (stage.getScene() != null) {
            width = stage.getScene().getWidth();
            height = stage.getScene().getHeight();
        }

        FXMLLoader loader = new FXMLLoader(AppManager.class.getResource(fxmlPath));
        Parent root = loader.load();

        Scene scene = new Scene(root, width, height);
        stage.setScene(scene);

        // Giữ trạng thái phóng to nếu trước đó có
        stage.setMaximized(isMaximized);

        stage.show();
    }
}
```

2. Views (Giao diện)

a. Cửa sổ lịch (MainWindow)

Cửa sổ lịch là giao diện chính của ứng dụng, nơi người dùng có thể xem và quản lý các công việc theo tuần. Các thành phần chính của cửa sổ lịch bao gồm:

- **ListView cho mỗi ngày trong tuần**: Mỗi ngày trong tuần được đại diện bởi một ListView và Label, hiển thị danh sách các công việc của ngày đó. Các ListView này được tùy chỉnh bằng lớp TaskCellCalendarWindow để hiển thị thông tin công việc một cách trực quan và dễ hiểu.
- **Nút điều hướng tuần**: Người dùng có thể chuyển đổi giữa các tuần bằng cách sử dụng các nút “Sau” và “Trước”. Khi tuần hiện tại thay đổi, các công việc trong tuần mới sẽ được nạp lại vào các ListView tương ứng.
- **DatePicker**: Người dùng có thể chọn một ngày cụ thể để chuyển đến tuần chứa ngày đó. DatePicker giúp người dùng dễ dàng điều hướng đến các tuần khác nhau mà không cần phải sử dụng các nút điều hướng tuần.

b. Cửa sổ ngày (DayWindow)

Cửa sổ ngày cung cấp giao diện chi tiết hơn về các công việc trong một ngày cụ thể. Các thành phần chính của cửa sổ ngày bao gồm:

- **ListView hiển thị công việc**: Danh sách các công việc trong ngày được hiển thị trong một ListView, với mỗi công việc được tùy chỉnh bằng lớp TaskCellDayWindow để hiển thị thông tin chi tiết như thời gian bắt đầu, thời gian làm việc và nghỉ ngơi, và trạng thái hiện tại của công việc.
- **Thông tin về công việc**: Mỗi ô trong ListView hiển thị thông tin chi tiết về công việc, bao gồm tên công việc, thời gian bắt đầu, thời gian làm việc và nghỉ ngơi, và trạng thái hiện tại. Các thông tin này được hiển thị trong một bảng để đảm bảo bố cục rõ ràng và dễ đọc.
- **Màu nền tùy chỉnh**: Màu nền của mỗi ô trong ListView được tùy chỉnh dựa trên mức độ quan trọng của công việc, giúp người dùng dễ dàng nhận biết các công việc quan trọng hơn.
- **Thanh theo dõi tiến độ hoàn thành công việc** : Giúp người dùng biết được mình đã hoàn thành bao nhiêu phần trăm công việc.
- **Nút thoát/thêm/xoá/chạy** : quay trở về cửa sổ lịch,thêm/xoá/chạy các task

c. Cửa sổ Pomodoro (PomodoroWindow)

Cửa sổ Pomodoro cung cấp giao diện chi tiết hơn về việc theo dõi thời gian làm việc và nghỉ ngơi của một công việc cụ thể. Các thành phần chính của cửa sổ Pomodoro bao gồm:

- **Nút Exit** : Quay trở về giao diện ngày.
- **Bộ đếm thời gian** : Dùng để theo dõi thời gian còn lại trong các phiên làm việc và nghỉ
- **Nhãn tổng thời gian tập trung**: Hiển thị ở phía dưới, hiển thị tổng thời gian n gười dùng đã tập trung làm việc.
- **Nhãn Mode**: Cho biết người dùng đang trong thời gian nghỉ hay làm việc, với n ền xanh nhạt.
- **Nút Start và Stop**: Dùng để bắt đầu hoặc dừng phiên hiện tại
- **ComboBox chọn bài hát**: Nằm gần phần trên trung tâm của cửa sổ, cho phép c họn bài hát để nghe khi làm việc.

3. Model (Mô hình)

a. Lớp Calendar

Lớp này chịu trách nhiệm quản lý các tuần và lưu trữ chúng. Lớp Calendar có các chức năng chính như:

- **Khởi tạo và cập nhật tuần hiện tại**: Đặt startOfCurrentWeek là đầu tuần hiện tại và cập nhật bản đồ tuần.

```
public class Calendar implements Serializable {
    @Serial
    private static final long serialVersionUID = 1L;

    private final Map<LocalDate, Week> weeks = new HashMap<>();
    private LocalDate startOfCurrentWeek;

    public Calendar() {
        this.startOfCurrentWeek = LocalDate.now().with(DayOfWeek.MONDAY);
        updateWeekMap();
    }

    public void updateWeekMap() {
        if (!weeks.containsKey(startOfCurrentWeek)) {
            weeks.put(startOfCurrentWeek, loadWeek());
        }
    }
```

- **Tải và lưu trữ tuần**: Tải tuần từ file hoặc tạo mới nếu chưa có, và lưu trữ các tuần trong map thành các file. Tải lại tuần hiện tại.

```
private Week loadWeek() {
    File file = new File(getWeekFilePath(startOfCurrentWeek));
    if (!file.exists()) return new Week(startOfCurrentWeek);
    try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream(file))) {
        return (Week) ois.readObject();
    } catch (IOException | ClassNotFoundException e) {
        return new Week(startOfCurrentWeek);
    }
}

public void saveWeeksToFile() throws IOException {
    Path directoryPath = Paths.get(System.getProperty("user.home"), ...more: "Documents", "saved-weeks");
    if (Files.notExists(directoryPath)) Files.createDirectories(directoryPath);

    for (Map.Entry<LocalDate, Week> entry : weeks.entrySet()) {
        LocalDate weekStart = entry.getKey();
        Week week = entry.getValue();
        try (ObjectOutputStream oos = new ObjectOutputStream(
                new FileOutputStream(getWeekFilePath(weekStart)))) {
            oos.writeObject(week);
        }
    }
}
```

```
private String getWeekFilePath(LocalDate weekStart) {
    return Paths.get(System.getProperty("user.home"),
            ...more: "Documents",
            "saved-weeks",
            weekStart.format(DateTimeFormatter.ofPattern("yyyy-MM-dd")) + ".dat").toString();
}
```

- **Chuyển đổi giữa các tuần**: Chuyển đổi giữa tuần hiện tại, tuần trước và tuần sau.

```
public void setToNextWeek() {
    startOfCurrentWeek = startOfCurrentWeek.plusWeeks( weeksToAdd: 1);
    updateWeekMap();
}

public void setToPreviousWeek() {
    startOfCurrentWeek = startOfCurrentWeek.minusWeeks( weeksToSubtract: 1);
    updateWeekMap();
}

public void setToAnotherWeek(LocalDate date) {
    startOfCurrentWeek = date.with(DayOfWeek.MONDAY);
    updateWeekMap();
}

public LocalDate getStartOfCurrentWeek() { return startOfCurrentWeek; }

public Week getCurrentWeek() { return weeks.get(startOfCurrentWeek); }
```

b. Lớp Week

Lớp Week có các chức năng chính như:

- **Khởi tạo tuần**: Tạo mới các đối tượng Day cho mỗi ngày trong tuần, bắt đầu từ ngày khởi tạo.

```
public class Week implements Serializable {
    @Serial
    private static final long serialVersionUID = 1L;

    private final List<Day> dayList;
    private final LocalDate startDate;

    public Week(LocalDate startDate) {
        this.startDate = startDate;
        this.dayList = new ArrayList<>();

        for (int i = 0; i < 7; i++) {
            LocalDate dayDate = startDate.plusDays(i);
            dayList.add(new Day(dayDate));
        }
    }
```

- **Quản lý tuần**: Lưu trữ ngày bắt đầu của tuần và danh sách các ngày trong tuần.

```
public List<Day> getDayList() { return dayList; }

public LocalDate getStartDate() { return startDate; }
```

c. Lớp Day

Lớp này chịu trách nhiệm quản lý các công việc trong một ngày cụ thể. Lớp Day có các chức năng chính sau:

- **Quản lý công việc danh sách**: Lưu trữ ngày cụ thể và các công việc liên quan.

```
public class Day implements Serializable {
    @Serial
    private static final long serialVersionUID = 1L;

    private final LocalDate date;

    private transient ObservableList<Task> taskObservableList;
    private List<Task> serializableList;

    public Day(LocalDate date) {
        this.date = date;
        this.taskObservableList = FXCollections.observableArrayList();
    }

    public LocalDate getDate() { return date; }
```

- **Thêm, xóa, sắp xếp công việc theo thời gian bắt đầu**: Cho phép thực hiện ba thao tác.

```
public void addTask(Task task) {
    taskObservableList.add(task);
    sortTasksByTime();
}

private void sortTasksByTime() {
    taskObservableList.sort(Comparator.comparing(taskmanagement.Models.Task::getStartTime));
}

public void removeTask(Task task) { taskObservableList.remove(task); }

public ObservableList<Task> getTaskObservableList() { return taskObservableList; }
```

- **Lưu trữ và khôi phục**: Sử dụng ObservableList để tự động cập nhật giao diện và dung 1 list bình thường để lưu trữ khi tuần tự hóa .

```java
@Serial
private void writeObject(ObjectOutputStream oos) throws IOException {
    serializableList = new ArrayList<>(taskObservableList);
    oos.defaultWriteObject();
}

// Tạo lại observable list sau khi deserialize
@Serial
private void readObject(ObjectInputStream ois) throws IOException, ClassNotFoundException {
    ois.defaultReadObject();
    taskObservableList = FXCollections.observableArrayList(serializableList);
}
```

d. Lớp Task

Lớp này chịu trách nhiệm quản lý các thông tin cơ bản về công việc, bao gồm thời điểm bắt đầu, các quãng làm và nghỉ, thời gian bắt buộc, và trạng thái hiện tại của công việc.

Lớp Task có các chức năng chính như:

- **Quản lý thông tin công việc**: Lưu trữ các thông tin cơ bản về công việc như tên, thời gian bắt đầu, thời gian làm việc và nghỉ ngơi, mức độ quan trọng, và thời gian bắt buộc, trạng thái công việc.

```java
public class Task implements Serializable {
    @Serial
    private static final long serialVersionUID = 1L;

    public enum State { READY, FOCUS, BREAK, STOPPED, FAIL, DONE }
    public enum Priority { Thấp, Trung, Cao }

    private final String taskName;
    private final LocalTime startTime;
    private final Duration focusTime;
    private final Duration breakTime;
    private final Priority importanceLevel;
    private final Duration mandatoryTime; // Thời gian bắt buộc thực hiện task
    private State currentState;
    private transient ObjectProperty<State> currentStateProperty;
    private Duration sessionElapsedTime = Duration.ZERO; // Thời gian trôi qua trong 1 session
    private Duration totalFocusTime = Duration.ZERO; // Tổng thời gian đã focus

    // Cờ để theo dõi thông báo
    private boolean notificationSent = false;
```

```
public Task(String taskName, LocalTime startTime, Duration focusTime,
            Duration breakTime, Priority importanceLevel, Duration mandatoryTime) {
    this.taskName = taskName;
    this.startTime = startTime;
    this.focusTime = focusTime;
    this.breakTime = breakTime;
    this.importanceLevel = importanceLevel;
    this.mandatoryTime = mandatoryTime;
    this.currentState = State.READY;
    this.currentStateProperty = new SimpleObjectProperty<>(currentState);
}
```

- **Quản lý thời gian**: Theo dõi và cập nhật tổng thời gian làm việc và thời gian trôi qua trong các quãng làm việc và nghỉ ngơi.

```
public void updateTime(Duration duration) {
    if (isRunning()) {
        sessionElapsedTime = sessionElapsedTime.add(duration);
        if (isFocus()) totalFocusTime = totalFocusTime.add(duration);
        if (totalFocusTime.greaterThanOrEqualTo(mandatoryTime)) setTaskDone();
    }
}
```

- **Chuyển đổi trạng thái**: Cho phép bắt đầu, dừng, và chuyển đổi giữa các trạng thái làm việc và nghỉ ngơi.

```
public void startFocus() { resetSession(State.FOCUS); }
public void startBreak() { resetSession(State.BREAK); }

private void resetSession(State state) {
    setCurrentState(state);
    sessionElapsedTime = Duration.ZERO;
}

public void stop() {
    setCurrentState(State.STOPPED);
    sessionElapsedTime = Duration.ZERO;
    totalFocusTime = Duration.ZERO;
}
```

```
public void updateTime(Duration duration) {
    if (isRunning()) {
        sessionElapsedTime = sessionElapsedTime.add(duration);
        if (isFocus()) totalFocusTime = totalFocusTime.add(duration);
        if (totalFocusTime.greaterThanOrEqualTo(mandatoryTime)) setTaskDone();
    }
}
```

```
public Duration getSessionRemainingTime() {
    Duration time = isBreak() ? breakTime : focusTime;
    return time.subtract(sessionElapsedTime);
}
```

- **Kiểm tra trạng thái và các phương thức get, set**

```
public boolean isFinishedSession() { return getSessionRemainingTime().lessThanOrEqualTo(Duration.ZERO); }
public boolean isDone() { return currentState == State.DONE; }
public boolean isReady() { return currentState == State.READY; }
public boolean isRunning() {
    return currentState == State.FOCUS || currentState == State.BREAK || currentState == State.DONE;
}
public boolean isFocus() { return currentState == State.FOCUS; }
public boolean isBreak() { return currentState == State.BREAK; }
public boolean isFail() { return currentState == State.FAIL; }

public void setTaskDone() { setCurrentState(State.DONE); }
public void setTaskFailed() { setCurrentState(State.FAIL); }

// Getter/setter cờ thông báo
public boolean hasBeenNotified() { return notificationSent; }
public void setNotificationSent() { this.notificationSent = true; }

public Priority getImportanceLevel() { return importanceLevel; }
public LocalTime getStartTime() { return startTime; }
public Duration getBreakTime() { return breakTime; }
public String getTaskName() { return taskName; }
public Duration getTotalFocusTime() { return totalFocusTime; }
public Duration getMandatoryTime() { return mandatoryTime; }
public Duration getFocusTime() { return focusTime; }
```

e. Lớp Music

Lớp này chịu trách nhiệm quản lý danh sách các bài hát và trạng thái phát nhạc. Lớp Music có các chức năng chính như:

- **Quản lý danh sách bài hát**: Tải danh sách các bài hát từ thư mục và lưu trữ chúng.

```
public class Music {
    private final List<File> songs;
    private State playingState = State.STOPPED;

    public enum State {
        PLAYING, STOPPED
    }

    public Music() { this.songs = loadSongsFromDirectory(); }

    private List<File> loadSongsFromDirectory() {
        try {
            URL resource = getClass().getClassLoader().getResource( name: "Music");
            if (resource != null) {
                File directory = new File(resource.getFile());
                File[] files = directory.listFiles();
                return files != null ? new ArrayList<>(Arrays.asList(files)) : new ArrayList<>();
            }
            return new ArrayList<>();
        } catch (Exception e) {
            return new ArrayList<>();
        }
    }

    public List<File> getSongs() { return songs; }
```

- **Trạng thái phát nhạc**: Quản lý trạng thái phát nhạc (đang phát hoặc dừng).

```
public boolean isPlaying() { return playingState == State.PLAYING; }

public void stopPlaying() { playingState = State.STOPPED; }
```

- **Bắt đầu và dừng phát nhạc**: Cho phép bắt đầu và dừng phát nhạc.

```
public void startPlaying() { playingState = State.PLAYING; }
```

4. Controller (Bộ điều khiển)

a. Lớp CalendarWindowController

Lớp này chịu trách nhiệm quản lý giao diện lịch, bao gồm việc hiển thị các công việc theo tuần và xử lý các sự kiện người dùng.

Lớp CalendarWindowController có các chức năng chính như:

- **Quản lý giao diện lịch**: Hiển thị các công việc theo tuần và cập nhật giao diện khi tuần hiện tại thay đổi.

```
@Override
public void initialize(URL url, ResourceBundle resourceBundle) {
    calendar = AppManager.calendar;
    datePicker.setValue(calendar.getStartOfCurrentWeek());

    listViews = List.of(mondayListView, tuesdayListView, wednesdayListView,
            thursdayListView, fridayListView, saturdayListView, sundayListView);
    labels = List.of(mondayLabel, tuesdayLabel, wednesdayLabel,
            thursdayLabel, fridayLabel, saturdayLabel, sundayLabel);

    setupListViewCellFactories();
    updateListViews();
    setupListViewWidths();

    // Cập nhật khi thay đổi kích thước cửa sổ
    rootPane.widthProperty().addListener(( ObservableValue<extends Number> obs, Number oldWidth, Number newWidth) -> setupListViewWidths());

    // Áp dụng nền
    BackgroundManager.applyBackground(rootPane);
}

private void setupListViewCellFactories() {
    listViews.forEach( ListView<Task> listView -> listView.setCellFactory( ListView<Task> _ -> new TaskCellCalendarWindow()));
}

public void updateListViews() {
    List<Day> dayList = calendar.getCurrentWeek().getDayList();
    IntStream.range(0, listViews.size()).forEach( int i ->
            listViews.get(i).setItems(dayList.get(i).getTaskObservableList()));

    updateDayLabels();
    highlightToday();
}
```

```
private void updateDayLabels() {
    List<LocalDate> days = calendar.getCurrentWeek().getDayList()
            .stream().map(Day::getDate).toList();

    java.time.format.DateTimeFormatter formatter =
            java.time.format.DateTimeFormatter.ofPattern("dd/MM");

    String[] weekdays = {"Thứ 2", "Thứ 3", "Thứ 4", "Thứ 5", "Thứ 6", "Thứ 7", "Chủ Nhật"};

    IntStream.range(0, labels.size()).forEach( int i -> {
        labels.get(i).setText(weekdays[i] + " - " + days.get(i).format(formatter));
    });
}

private void highlightToday() {
    LocalDate today = LocalDate.now();
    List<Day> dayList = calendar.getCurrentWeek().getDayList();

    IntStream.range(0, labels.size()).forEach( int i -> {
        LocalDate date = dayList.get(i).getDate();
        if (date.equals(today)) {
            labels.get(i).setStyle("-fx-background-color: #4CAF50; -fx-font-weight: bold; -fx-border-color: black;");
        } else {
            labels.get(i).setStyle("-fx-background-color: #FFFFFF; -fx-font-weight: bold; -fx-border-color: black;");
        }
    });
}
```

- **Tính toán kích thước giao diện**: Điều chỉnh kích thước các list view và label để phù hợp với kích thước cửa sổ.

```
private void setupListViewWidths() {
    double width = rootPane.getWidth() / listViews.size();
    IntStream.range(0, listViews.size()).forEach( int i -> {
        double x = i * width;
        labels.get(i).setPrefWidth(width);
        labels.get(i).setLayoutX(x);
        labels.get(i).setLayoutY(80);

        listViews.get(i).setPrefWidth(width);
        listViews.get(i).setLayoutX(x);
        listViews.get(i).setLayoutY(110);
    });
}
```

- **Xử lý sự kiện người dùng**: Chuyển đổi giữa các tuần và chuyển sang cửa sổ ngày tương ứng khi người dùng click vào các list view; chuyển đổi sang cửa sổ thay đổi hình nền

```java
@FXML
public void mondayClicked() throws IOException { handleDayClick( dayIndex: 0); }
@FXML
public void tuesdayClicked() throws IOException { handleDayClick( dayIndex: 1); }
@FXML
public void wednesdayClicked() throws IOException { handleDayClick( dayIndex: 2); }
@FXML
public void thursdayClicked() throws IOException { handleDayClick( dayIndex: 3); }
@FXML
public void fridayClicked() throws IOException { handleDayClick( dayIndex: 4); }
@FXML
public void saturdayClicked() throws IOException { handleDayClick( dayIndex: 5); }
@FXML
public void sundayClicked() throws IOException { handleDayClick( dayIndex: 6); }

private void handleDayClick(int dayIndex) throws IOException {
    AppManager.selectedDay = calendar.getCurrentWeek().getDayList().get(dayIndex);
    AppManager.switchToDayWindow();
    listViews.get(dayIndex).getSelectionModel().clearSelection();
}

@FXML
private void handleChangeDate() {
    if (!isUpdatingDatePicker && datePicker.getValue() != null) {
        calendar.setToAnotherWeek(datePicker.getValue());
        updateListViews();
    }
}
```

```java
@FXML
private void handleNextButtonAction() {
    calendar.setToNextWeek();
    updateDatePicker();
    updateListViews();
}

@FXML
private void handlePreviousButtonAction() {
    calendar.setToPreviousWeek();
    updateDatePicker();
    updateListViews();
}

private void updateDatePicker() {
    isUpdatingDatePicker = true;
    datePicker.setValue(calendar.getStartOfCurrentWeek());
    isUpdatingDatePicker = false;
}
```

```
@FXML
private void handleUploadBackground() {
    FileChooser fileChooser = new FileChooser();
    fileChooser.setTitle("Chọn ảnh làm nền");
    fileChooser.getExtensionFilters().addAll(
            new FileChooser.ExtensionFilter( s: "Image Files", ...strings: "*.png", "*.jpg", "*.jpeg")
    );

    File file = fileChooser.showOpenDialog( window: null);
    if (file != null) {
        try {
            String imagePath = file.toURI().toString();
            BackgroundManager.saveBackground(imagePath);
            BackgroundManager.applyBackground(rootPane);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

b. Lớp TaskCellCalendarWindow

Lớp này kế thừa ListCell<Task> chịu trách nhiệm tùy chỉnh các ô hiển thị trong danh sách công việc của cửa sổ chính.

```
public class TaskCellCalendarWindow extends ListCell<Task> {

    @Override
    protected void updateItem(Task task, boolean empty) {
        super.updateItem(task, empty);

        if (empty || task == null) {
            setGraphic(null);
        } else {
            // Tạo nhãn và container
            Label taskNameLabel = new Label(task.getTaskName());
            taskNameLabel.setWrapText(true);
            taskNameLabel.setTextAlignment(TextAlignment.CENTER);
            taskNameLabel.setMaxWidth(getListView().getWidth() - 40);

            HBox container = new HBox(taskNameLabel);
            container.setAlignment(Pos.CENTER);
            container.setPadding(new Insets( v: 5));

            // Màu nền và viền dựa trên mức độ ưu tiên
            Color backgroundColor;
            Color borderColor;
```

```
        switch (task.getImportanceLevel()) {
            case Thấp -> {
                backgroundColor = Color.web( s: "#A8E6CF"); // xanh nhạt
                borderColor = Color.web( s: "#00FF00");    // viền xanh
            }
            case Trung -> {
                backgroundColor = Color.web( s: "#FFFF99"); // vàng nhạt
                borderColor = Color.web( s: "#FF9800");    // viền cam
            }
            case Cao -> {
                backgroundColor = Color.web( s: "#FFAB91"); // cam đỏ
                borderColor = Color.web( s: "#FF0000");    // viền đỏ
            }
            default -> {
                backgroundColor = Color.LIGHTGRAY;
                borderColor = Color.GRAY;
            }
        }

        // Set background và border
        container.setBackground(new Background(
                new BackgroundFill(backgroundColor, new CornerRadii( v: 5), Insets.EMPTY)
        ));
        container.setBorder(new Border(
                new BorderStroke(borderColor, BorderStrokeStyle.SOLID, new CornerRadii( v: 5), new BorderWidths( v: 2))
        ));

        setGraphic(container);
    }
```

c. Lớp DayViewController

Lớp này chủ yếu xử lý các hoạt động liên quan đến việc hiển thị và quản lý các công việc trong một ngày cụ thể.

Lớp DayViewController có các chức năng chính sau:

- **Khởi tạo và Cập nhật giao diện**: Khi lớp được khởi tạo, nó lấy ngày hiện tại từ AppManager và thiết lập danh sách công việc cho ListView. Nó cũng cập nhật nhãn hiển thị ngày tháng và thêm biểu tượng cho mức độ quan trọng của các công việc.

```
    public void initialize(URL url, ResourceBundle resourceBundle) {
        day = AppManager.selectedDay;

        listView.setCellFactory( ListView<Task> _ -> new TaskCellDayWindow());
        listView.setItems(day.getTaskObservableList());

        dateLabel.setText(day.getDate().toString());
        dateLabel.setFont(Font.font( s: "System", FontWeight.BOLD,  v: 12));

        BackgroundManager.applyBackground(rootPane);

        updateCompletion();
        addImportanceLegend();
    }
```

- **Thêm Công việc**:

➢ Tạo một hộp thoại để người dùng nhập thông tin cho công việc mới, bao gồm tên công việc, thời gian bắt đầu, thời gian tập trung, thời gian nghỉ, mức độ quan trọng và thời gian bắt buộc.

```
public void handleAddTask() {
    Dialog<Task> dialog = new Dialog<>();
    dialog.setTitle("Thêm công việc mới");
    dialog.setHeaderText("Nhập chi tiết cho công việc mới");
    ButtonType addButtonType = new ButtonType( s: "Thêm", ButtonBar.ButtonData.OK_DONE);
    dialog.getDialogPane().getButtonTypes().addAll(addButtonType, ButtonType.CANCEL);

    TextField taskNameField = new TextField();
    taskNameField.setPromptText("Tên công việc");

    TextField startTimeField = new TextField();
    startTimeField.setPromptText("Thời điểm bắt đầu (H:mm)");

    TextField focusTimeField = new TextField();
    focusTimeField.setPromptText("Quãng tập trung (phút)");

    TextField breakTimeField = new TextField();
    breakTimeField.setPromptText("Quãng nghỉ (phút)");

    ChoiceBox<Task.Priority> importanceLevelChoiceBox = new ChoiceBox<>();
    importanceLevelChoiceBox.setItems(FXCollections.observableArrayList(Task.Priority.values()));
    importanceLevelChoiceBox.getSelectionModel().selectFirst();

    TextField mandatoryTimeField = new TextField();
    mandatoryTimeField.setPromptText("Thời gian bắt buộc (phút)");

    VBox content = new VBox( v: 10);
    content.getChildren().addAll(
            new Label( s: "Tên công việc:"), taskNameField,
            new Label( s: "Thời gian bắt đầu:"), startTimeField,
            new Label( s: "Thời gian tập trung:"), focusTimeField,
            new Label( s: "Thời gian nghỉ:"), breakTimeField,
            new Label( s: "Mức độ ưu tiên:"), importanceLevelChoiceBox,
            new Label( s: "Thời gian bắt buộc:"), mandatoryTimeField
    );

    dialog.getDialogPane().setContent(content);
```

➢ Nó kiểm tra tính hợp lệ của các thông tin nhập vào, ngăn chặn việc thêm công việc nếu thời gian bắt đầu là trước thời điểm hiện tại hoặc nếu có xung đột thời gian với các công việc đã tồn tại.

```
        final Button addButton = (Button) dialog.getDialogPane().lookupButton(addButtonType);
        addButton.addEventFilter(ActionEvent.ACTION, ActionEvent event -> {
            try {
                String taskName = taskNameField.getText();
                LocalTime startTime = LocalTime.parse(startTimeField.getText(), DateTimeFormatter.ofPattern("H:mm"));
                Duration focusTime = Duration.minutes(Integer.parseInt(focusTimeField.getText()));
                Duration breakTime = Duration.minutes(Integer.parseInt(breakTimeField.getText()));
                Task.Priority importanceLevel = importanceLevelChoiceBox.getValue();
                Duration mandatoryTime = Duration.minutes(Integer.parseInt(mandatoryTimeField.getText()));

                Task newTask = new Task(taskName, startTime, focusTime, breakTime, importanceLevel, mandatoryTime);
                boolean valid = true;
                boolean isBeforeToday = day.getDate().isBefore(LocalDate.now());
                boolean isTodayButBeforeNow = day.getDate().isEqual(LocalDate.now())
                        && newTask.getStartTime().isBefore(LocalTime.now());
                if (isBeforeToday || isTodayButBeforeNow) {
                    Alert alert = new Alert(Alert.AlertType.WARNING);
                    alert.setHeaderText("Đã quá thời điểm bắt đầu task");
                    alert.setContentText("Không thể tạo công việc với thời gian bắt đầu trước thời điểm hiện tại.");
                    alert.showAndWait();
                    event.consume();
                    valid = false;
                }
                if (isTimeConflict(newTask)) {
                    Alert alert = new Alert(Alert.AlertType.CONFIRMATION);
                    alert.setHeaderText("Thời gian công việc trùng với công việc khác");
                    alert.setContentText("Bạn có chắc chắn muốn tạo task này không?");
                    Optional<ButtonType> result = alert.showAndWait();
                    if (result.isEmpty() || result.get() != ButtonType.OK) {
                        event.consume();
                        valid = false;
                    }
                }
                if (valid) day.addTask(newTask);
            } catch (DateTimeParseException e) {
                showErrorDialog( header: "Lỗi định dạng",  content: "Thời gian bắt đầu phải theo định dạng HH:mm.");
                event.consume();
            } catch (NumberFormatException e) {
                showErrorDialog( header: "Lỗi định dạng",  content: "Thời gian tập trung, nghỉ và bắt buộc phải là số nguyên.");
                event.consume();
            } catch (Exception e) {
                showErrorDialog( header: "Dữ liệu không hợp lệ",  content: "Vui lòng đảm bảo tất cả các trường đều được nhập đúng định dạng.");
                event.consume();
            }
        });
        dialog.showAndWait();
    }

    private boolean isTimeConflict(Task newTask) {
        LocalTime newStart = newTask.getStartTime();
        LocalTime newEnd = newStart.plusSeconds((long) newTask.getMandatoryTime().toSeconds());

        return day.getTaskObservableList().stream().anyMatch( Task existingTask -> {
            LocalTime existingStart = existingTask.getStartTime();
            LocalTime existingEnd = existingStart.plusSeconds((long) existingTask.getMandatoryTime().toSeconds());
            return newStart.isBefore(existingEnd) && newEnd.isAfter(existingStart);
        });
    }
```

- **Xóa Công việc**: Kiểm tra xem có công việc nào được chọn trong ListView hay không. Nếu có, nó hiển thị một hộp thoại xác nhận trước khi xóa công việc được chọn khỏi danh sách.

```
public void handleDeleteTask() {
    if (listView.getSelectionModel().getSelectedItem() != null) {
        Alert alert = new Alert(Alert.AlertType.CONFIRMATION);
        alert.setTitle("Xác nhận xóa");
        alert.setHeaderText("Xóa công việc");
        alert.setContentText("Bạn có chắc muốn xóa công việc này không?");
        Optional<ButtonType> result = alert.showAndWait();
        if (result.isPresent() && result.get() == ButtonType.OK) {
            day.removeTask(listView.getSelectionModel().getSelectedItem());
        }
        listView.getSelectionModel().clearSelection();
    }
}
```

- **Bắt đầu Công việc**: Kiểm tra xem có công việc nào được chọn hay không. Nếu có, nó xác nhận với người dùng trước khi chuyển sang cửa sổ Pomodoro cho công việc đang chọn, đồng thời hiển thị thông báo nếu công việc đã hoàn thành hoặc đã quá hạn.

```
public void handleStartTask() throws IOException {
    if (listView.getSelectionModel().getSelectedItem() == null) return;

    if (AppManager.selectedTask.isReady()) {
        Alert alert = new Alert(Alert.AlertType.CONFIRMATION);
        alert.setContentText("Xác nhận bắt đầu làm việc");
        Optional<ButtonType> result = alert.showAndWait();
        if (result.isPresent() && result.get() == ButtonType.OK) {
            AppManager.switchToPomodoroWindow();
        }
    } else if (AppManager.selectedTask.isDone()) {
        Alert alert = new Alert(Alert.AlertType.INFORMATION);
        alert.setHeaderText("TASK DONE");
        alert.setContentText("Bạn đã hoàn thành công việc này rồi");
        alert.show();
    } else if (AppManager.selectedTask.isFail()) {
        Alert alert = new Alert(Alert.AlertType.INFORMATION);
        alert.setHeaderText("TASK FAILED");
        alert.setContentText("Đã quá hạn để làm việc này");
        alert.show();
    }
    listView.getSelectionModel().clearSelection();
}
```

- **Xử lý thoát**: Cho phép người dùng thoát khỏi màn hình hiện tại và quay lại màn hình chính của ứng dụng.

```
@FXML
public void handleExitButton() { AppManager.switchToMainWindow(); }
```

- **Cập nhật Tiến độ hoàn thành**: Tính toán và cập nhật phần trăm hoàn thành của các công việc trong ngày, hiển thị thông tin này trên thanh tiến độ và nhãn phần trăm hoàn thành.

```
private void updateCompletion() {
    long completedTasks = day.getTaskObservableList().stream()
            .filter(Task::isDone)
            .count();
    long totalTasks = day.getTaskObservableList().size();
    double completion = (totalTasks > 0) ? (double) completedTasks / totalTasks : 0;

    completionBar.setProgress(completion);
    completionPercentage.setText(String.format("%.0f%%", completion * 100));
}
```

- **Thêm mô tả về Mức độ quan trọng**: Tạo và thêm mô tả về mức độ quan trọng của các công việc vào giao diện, sử dụng các màu sắc khác nhau để phân biệt các mức độ quan trọng như thấp, trung bình và cao.

```
private void addImportanceLegend() {
    Label text = new Label( s: "Mức ưu tiên");

    HBox lowPriority = createPriorityBox( labelText: "THẤP", Color.web( s: "#A8E6CF"));
    HBox mediumPriority = createPriorityBox( labelText: "TRUNG", Color.web( s: "#FFD54F"));
    HBox highPriority = createPriorityBox( labelText: "CAO", Color.web( s: "#FFAB91"));

    lowPriority.setPadding(new Insets( v: 5));
    mediumPriority.setPadding(new Insets( v: 5));
    highPriority.setPadding(new Insets( v: 5));
    text.setPadding(new Insets( v: 5,  v1: 5,  v2: 0,  v3: 5));

    importanceLegendBox.getChildren().addAll(text, lowPriority, mediumPriority, highPriority);
}

private HBox createPriorityBox(String labelText, Color color) {
    Label label = new Label(labelText);
    Rectangle colorBox = new Rectangle( v: 15,  v1: 15, color);
    colorBox.setStroke(Color.BLACK);
    HBox box = new HBox(colorBox, label);
    box.setSpacing(5);
    return box;
}
```

d. Lớp TaskCellDayWindow

Lớp này kế thừa ListCell<Task> chịu trách nhiệm tùy chỉnh các ô hiển thị trong danh sách công việc của cửa sổ chính.

Lớp TaskCellDayWindow có các chức năng chính như:

- **Thiết lập thông tin về công việc :** Trong phương thức updateItem, thông tin công việc được thêm vào một GridPane, bao gồm tên công việc, thời gian bắt đầu, thời gian tối thiểu, khoảng thời gian tập trung và khoảng thời gian nghỉ.

```
    HBox cell = new HBox();
    cell.setSpacing(10);

    GridPane taskInfo = new GridPane();
    taskInfo.setVgap(5);
    taskInfo.setHgap(10);

    addTaskInfo(taskInfo, item.getTaskName(), colIndex: 0, rowIndex: 0, FontWeight.BOLD);
    addTaskInfo(taskInfo, text: "Thời gian bắt đầu: " + item.getStartTime(), colIndex: 0, rowIndex: 1, FontWeight.NORMAL);
    addTaskInfo(taskInfo, text: "Thời gian tối thiểu: " + item.getMandatoryTime().toMinutes() + " phút", colIndex: 1, rowIndex: 1, FontWeight.NORMAL);
    addTaskInfo(taskInfo, text: "Quãng tập trung: " + item.getFocusTime().toMinutes() + " phút", colIndex: 0, rowIndex: 2, FontWeight.NORMAL);
    addTaskInfo(taskInfo, text: "Quãng nghỉ: " + item.getBreakTime().toMinutes() + " phút", colIndex: 1, rowIndex: 2, FontWeight.NORMAL);
```

```
private void addTaskInfo(GridPane taskInfo, String text, int colIndex, int rowIndex, FontWeight fontWeight) {
    Text textNode = new Text(text);
    textNode.setFont(Font.font( s: "System", fontWeight, v: 12));
    textNode.setFill(Color.BLACK);
    taskInfo.add(textNode, colIndex, rowIndex);
}
```

- **Phân loại công việc bằng màu sắc theo mức độ quan trọng**: Gán màu nền của ô công việc dựa trên mức độ quan trọng (THẤP, TRUNG, CAO) với các màu sắc khác nhau.

```
// Màu nền dựa trên mức độ quan trọng
BackgroundFill backgroundFill = switch (item.getImportanceLevel()) {
    case Thấp -> new BackgroundFill(Color.web( s: "#A8E6CF"), new CornerRadii( v: 5), Insets.EMPTY);
    case Trung -> new BackgroundFill(Color.web( s: "#FFD54F"), new CornerRadii( v: 5), Insets.EMPTY);
    case Cao -> new BackgroundFill(Color.web( s: "#FFAB91"), new CornerRadii( v: 5), Insets.EMPTY);
};
cell.setBackground(new Background(backgroundFill));
```

- **Cập nhật giao diện khi được chọn**: Thay đổi màu sắc của ô công việc khi được chọn.

```
// Hiệu ứng khi chọn
if (isSelected()) {
    ColorAdjust colorAdjust = new ColorAdjust();
    colorAdjust.setBrightness(-0.2);
    cell.setEffect(colorAdjust);
    AppManager.selectedTask = item;
} else {
    cell.setEffect(null);
}
```

- **Hiển thị trạng thái công việc**: Gán nhãn trạng thái cho các công việc và cập nhật trạng thái theo thời gian thực.

```
Label statusLabel = new Label();
statusLabel.setFont(Font.font( s: "System", FontWeight.BOLD,  v: 12));
statusLabel.setBackground(new Background(new BackgroundFill(Color.web( s: "#F0F0F0"), new CornerRadii( v: 5), Insets.EMPTY)));
statusLabel.setPadding(new Insets( v: 5));
statusLabel.setTextFill(Color.BLACK);

StringBinding statusBinding = Bindings.createStringBinding(() -> {
    Task.State state = item.currentStateProperty().get();
    return switch (state) {
        case DONE -> "HOÀN THÀNH";
        case FAIL -> "THẤT BẠI";
        case READY -> "SẴN SÀNG";
        case FOCUS -> "TẬP TRUNG";
        case BREAK -> "NGHỈ NGƠI";
        case STOPPED -> "DỪNG";
    };
}, item.currentStateProperty());
statusLabel.textProperty().bind(statusBinding);
```

e. Lớp PomodoroController

Lớp này chịu trách nhiệm xử lý các chức năng liên quan đến phương pháp Pomodoro, bao gồm quản lý thời gian làm việc, phát nhạc và xử lý các sự kiện từ người dùng.

Lớp PomodoroController có các chức năng chính sau:

- **Khởi tạo lớp**

Khởi tạo biến task từ AppManager.selectedTask để xác định nhiệm vụ hiện tại. Thiết lập nhãn đếm ngược (countdownLabel) với thời gian làm việc ban đầu.

```
public void initialize(URL url, ResourceBundle resourceBundle) {
    this.task = AppManager.selectedTask;
    if (this.task == null) {
        System.err.println("Lỗi: PomodoroController được tải mà không có task nào được chọn.");
        Alert alert = new Alert(Alert.AlertType.ERROR);
        alert.setTitle("Lỗi");
        alert.setHeaderText("Không thể bắt đầu Pomodoro");
        alert.setContentText("Không có công việc nào được chọn. Vui lòng quay lại và chọn một công việc.");
        alert.showAndWait();
        return;
    }

    countdownLabel.setText(formatTime((int) task.getFocusTime().toSeconds()));
```

Tạo một Timeline để cập nhật đồng hồ đếm ngược mỗi giây.

```
    timeline = new Timeline(new KeyFrame(Duration.seconds( v: 1),  ActionEvent _ -> updateCountdown()));
    timeline.setCycleCount(Timeline.INDEFINITE);
```

Tải âm thanh thông báo khi kết thúc phiên.

```
    notificationSound = new AudioClip(
            Objects.requireNonNull(getClass().getResource(NOTIFICATION_SOUND_PATH)).toString()
    );
```

Thiết lập danh sách bài hát từ lớp Music vào ComboBox nếu có bài hát.

```
this.music = new Music();
if (!music.getSongs().isEmpty()) {
    setUpListSong();
}
```

Đăng ký xử lý sự kiện thoát ứng dụng khi người dùng cố gắng đóng cửa sổ.

```
AppManager.stage.setOnCloseRequest( WindowEvent event -> {
    event.consume();
    try {
        handleExit();
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
});
```

- **Cập nhật đếm ngược**

  Cập nhật thời gian còn lại của nhiệm vụ bằng cách giảm thời gian mỗi giây. Nếu phiên làm việc kết thúc, phát âm thanh thông báo và chuyển sang trạng thái "Break" hoặc "Focus" tùy thuộc vào trạng thái hiện tại của nhiệm vụ.

```
private void updateCountdown() {
    task.updateTime(Duration.seconds( v: 1));
    if (task.isFinishedSession()) {
        notificationSound.play();
        if (task.isBreak()) {
            task.startFocus();
            modeLabel.setText("Tập trung");
        } else {
            task.startBreak();
            modeLabel.setText("Nghỉ");
        }
    }
    updateTimeLabel();
}
```

- **Bắt đầu và Dừng phiên Pomodoro**

  Bắt đầu phiên làm việc nếu chưa bắt đầu. Thay đổi trạng thái modeLabel thành "Focus" và phát Timeline để bắt đầu đếm thời gian.

```
@FXML
public void startPomodoroButton() {
    if (!task.isRunning()) {
        task.startFocus();
        updateTimeLabel();
        modeLabel.setText("Tập trung");
        timeline.play();
    }
}
```

Dừng phiên làm việc nếu nó đang chạy. Dừng Timeline và thay đổi trạng thái modeLabel thành "Stop".

```
@FXML
public void stopPomodoroButton() {
    if (task.isRunning()) {
        task.stop();
        timeline.stop();
        modeLabel.setText("Nghỉ");
    }
}
```

- **Cập nhật thời gian hiển thị**

  Cập nhật nhãn đếm ngược và nhãn tổng thời gian đã làm việc. Thời gian được định dạng thành chuỗi với định dạng "HH:mm" hoặc "mm".

```
private void updateTimeLabel() {
    countdownLabel.setText(formatTime((int) task.getSessionRemainingTime().toSeconds()));
    totalTimeLabel.setText("Thời gian tập trung: " + formatTime((int) task.getTotalFocusTime().toSeconds()));
}

private String formatTime(int seconds) {
    int hours = seconds / 3600;
    int minutes = (seconds % 3600) / 60;
    seconds = seconds % 60;
    if (hours > 0) return String.format("%02d:%02d:%02d", hours, minutes, seconds);
    else return String.format("%02d:%02d", minutes, seconds);
}
```

- **Xử lý thoát ứng dụng.**

  Trước khi thoát, kiểm tra xem nhiệm vụ đã hoàn thành hay chưa. Nếu chưa hoàn thành, hiển thị hộp thoại xác nhận, thông báo cho người dùng về thời gian còn lại để hoàn thành nhiệm vụ.

```
private void handleExit() throws IOException {
    if (task.isDone()) {
        exitToDayWindow();
    } else {
        Alert alert = new Alert(Alert.AlertType.CONFIRMATION);
        alert.setTitle("Xác nhận thoát");
        int totalRemainTime = (int) task.getMandatoryTime().subtract(task.getTotalFocusTime()).toSeconds();
        int minutes = totalRemainTime / 60, seconds = totalRemainTime % 60;
        alert.setContentText(String.format(
                "Còn %d phút %d giây nữa để hoàn thành\nTask sẽ tính là không hoàn thành nếu bạn thoát bây giờ",
                minutes, seconds
        ));
        Optional<ButtonType> result = alert.showAndWait();
        if (result.isPresent() && result.get() == ButtonType.OK) {
            task.setTaskFailed();
            exitToDayWindow();
        }
    }
}
```

```
private void exitToDayWindow() throws IOException {
    if (mediaPlayer != null) mediaPlayer.stop();
    AppManager.switchToDayWindow();
    AppManager.stage.setOnCloseRequest(null);
    if (mediaPlayer != null) mediaPlayer.stop();
}
```

- **Thiết lập danh sách bài hát:** Nạp danh sách bài hát vào ComboBox và thiết lập bài hát đầu tiên làm bài hát hiện tại. Đăng ký sự kiện khi người dùng thay đổi bài hát chọn.

```
private void setUpListSong() {
    for (File song : music.getSongs()) {
        songPicker.getItems().add(song.getName().substring(0, song.getName().lastIndexOf( ch: '.')));
    }
    songPicker.getSelectionModel().selectFirst();
    loadSelectedSong();

    songPicker.getSelectionModel().selectedIndexProperty().addListener(( ObservableValue<extends Number> _, Number _, Number newVal) -> {
        songNumber = newVal.intValue();
        loadSelectedSong();
        playPauseButton.setText("Bắt đầu");
    });
}
```

- **Tải bài hát chọn:** Dừng bài hát hiện tại nếu có và tải bài hát được chọn từ danh sách để chuẩn bị phát.

```
private void loadSelectedSong() {
    if (mediaPlayer != null && music.isPlaying()) {
        mediaPlayer.stop();
        mediaPlayer.dispose();
        music.stopPlaying();
    }
    File selectedSong = music.getSongs().get(songNumber);
    Media media = new Media(selectedSong.toURI().toString());
    mediaPlayer = new MediaPlayer(media);
    mediaPlayer.setCycleCount(MediaPlayer.INDEFINITE);
}
```

- **Điều khiển phát nhạc:** Điều khiển việc phát hoặc dừng bài hát hiện tại. Thay đổi nút từ "Play" thành "Pause" và ngược lại dựa trên trạng thái của bài hát.

```
public void handlePlayPause() {
    if (music.getSongs().isEmpty()) return;

    if (music.isPlaying()) {
        mediaPlayer.pause();
        music.stopPlaying();
        playPauseButton.setText("Bắt đầu");
    } else {
        mediaPlayer.play();
        music.startPlaying();
        playPauseButton.setText("Dừng");
    }
}
```

f. Lớp StatusUpdateService

Lớp này chịu trách nhiệm chạy dịch vụ ngầm để đánh dấu các công việc đã quá hạn theo thời gian thực và gửi thông báo pop-up trên màn hình trước 1 phút của task chuẩn bị bắt đầu.

```
public class StatusUpdateService extends ScheduledService<Void> {

    // Thời gian quét định kỳ (giây)
    private static final long CHECK_INTERVAL_SECONDS = 10;
    // Thời gian thông báo trước khi task bắt đầu (phút)
    private static final long NOTIFY_BEFORE_MINUTES = 5;

    public StatusUpdateService() { setPeriod(Duration.seconds(CHECK_INTERVAL_SECONDS)); }

    @Override
    protected javafx.concurrent.Task<Void> createTask() {
        return () -> {
                updateFailedStatus(AppManager.selectedDay);
                checkForNotifications();
                return null;
        };
    }
```

```
    /** Cập nhật trạng thái FAIL cho các task đã quá hạn (ngày đang xem) */
    private void updateFailedStatus(Day selectedDay) {
        if (selectedDay == null) return;

        LocalDate currentDate = LocalDate.now();
        LocalTime currentTime = LocalTime.now();

        selectedDay.getTaskObservableList().stream()
                .filter( Task modelTask -> modelTask.isReady())
                .forEach( Task modelTask -> {
                    boolean isBeforeToday = selectedDay.getDate().isBefore(currentDate);
                    boolean isTodayAndExpired = selectedDay.getDate().isEqual(currentDate)
                            && modelTask.getStartTime().plusMinutes( minutesToAdd: 5).isBefore(currentTime);

                    if (isBeforeToday || isTodayAndExpired) {
                        Platform.runLater(modelTask::setTaskFailed);
                    }
                });
    }
```

```java
/** Gửi thông báo pop-up cho các task sắp bắt đầu (dựa trên ngày hôm nay) */
private void checkForNotifications() {
    if (AppManager.calendar == null) return;

    LocalDate todayDate = LocalDate.now();
    LocalTime now = LocalTime.now();

    AppManager.calendar.setToAnotherWeek(todayDate); // đảm bảo tuần hiện tại được tải
    Day today = AppManager.calendar.getCurrentWeek().getDayList().stream()
            .filter( Day d -> d.getDate().isEqual(todayDate))
            .findFirst().orElse( other: null);

    if (today == null) return;

    today.getTaskObservableList().stream()
            .filter( Task modelTask -> modelTask.isReady() && !modelTask.hasBeenNotified())
            .forEach( Task modelTask -> {
                long secondsUntilStart = java.time.Duration.between(now, modelTask.getStartTime()).getSeconds();
                if (secondsUntilStart > 0 && secondsUntilStart <= (NOTIFY_BEFORE_MINUTES * 60)) {
                    modelTask.setNotificationSent();
                    Platform.runLater(() -> {
                        Notifications.create()
                                .title("Công việc sắp bắt đầu!")
                                .text(String.format("Công việc '%s' sẽ bắt đầu lúc %s.",
                                        modelTask.getTaskName(), modelTask.getStartTime().toString()))
                                .position(Pos.BOTTOM_RIGHT)
                                .hideAfter(Duration.seconds( v: 10))
                                .showInformation();
                    });
                }
            });
}
```

5. Utils (Tiện ích hỗ trợ)

a. BackgroundManager

BackgroundManager giúp quản lý hình nền, giúp ghi nhớ và tải lại hình nền cho giao diện JavaFX

- Lưu đường dẫn của ảnh nền mới nếu người dùng thay đổi ảnh nền

```java
public static void saveBackground(String imagePath) {
    Preferences prefs = Preferences.userNodeForPackage(BackgroundManager.class);
    prefs.put(KEY, imagePath);
}
```

- Thay đổi ảnh nền

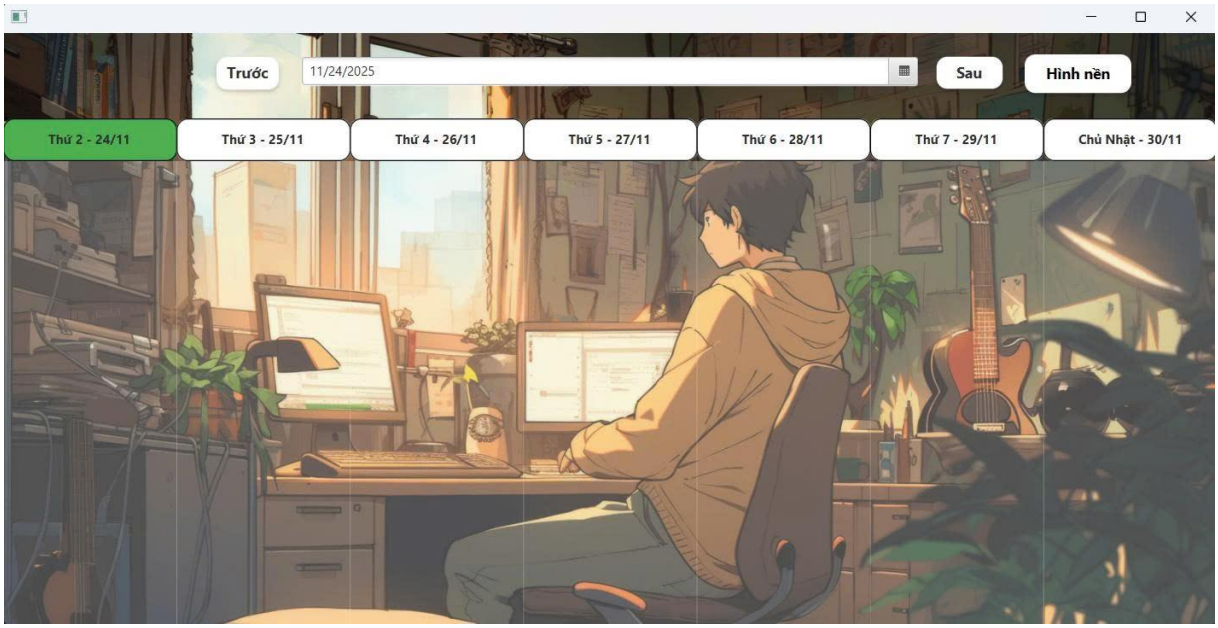
```
public static void applyBackground(Pane pane) {
    Preferences prefs = Preferences.userNodeForPackage(BackgroundManager.class);
    String path = prefs.get(KEY,  def: null);

    if (path != null) {
        pane.setStyle(
                "-fx-background-image: url('" + path + "');" +
                        "-fx-background-size: cover;" +
                        "-fx-background-position: center;" +
                        "-fx-background-repeat: no-repeat;"
        );
    }
}
```

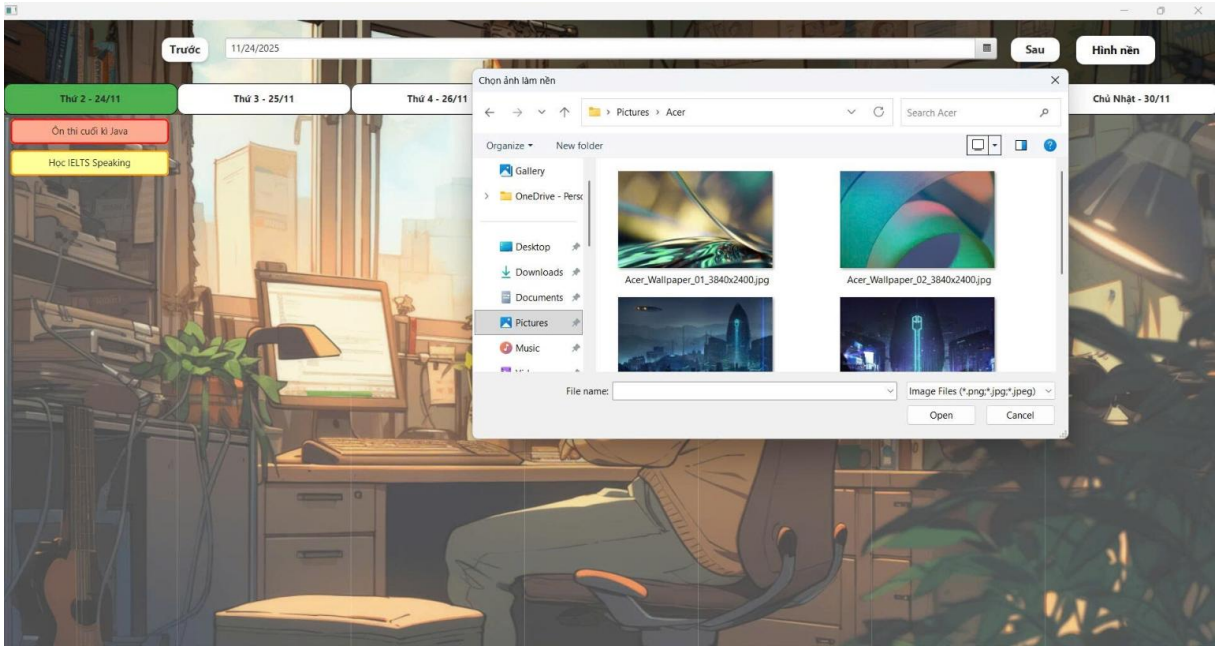
IV. Kết quả đạt được

1. Giao diện người dùng trực quan và dễ sử dụng

Ứng dụng đã phát triển một giao diện dễ sử dụng, giúp người dùng dễ dàng tạo, quản lý và theo dõi các công việc. Giao diện lịch và danh sách công việc được thiết kế rõ ràng, với các màu sắc và biểu tượng giúp phân biệt mức độ quan trọng của các công việc. Ngoài ra ứng dụng còn giúp người dùng dễ dàng thay đổi giao diện màn hình chính bằng cách có thể tự import ảnh từ máy tính lên.



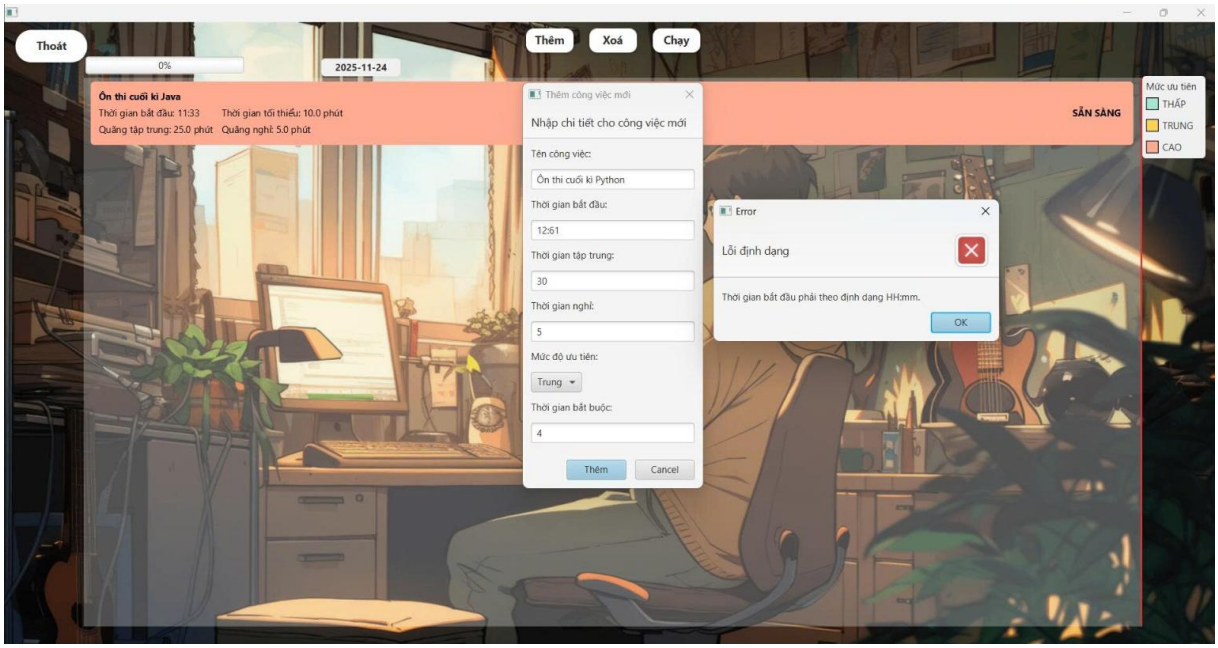
Hình 1. Giao diện chính của ứng dụng quản lý thời gian và công việc



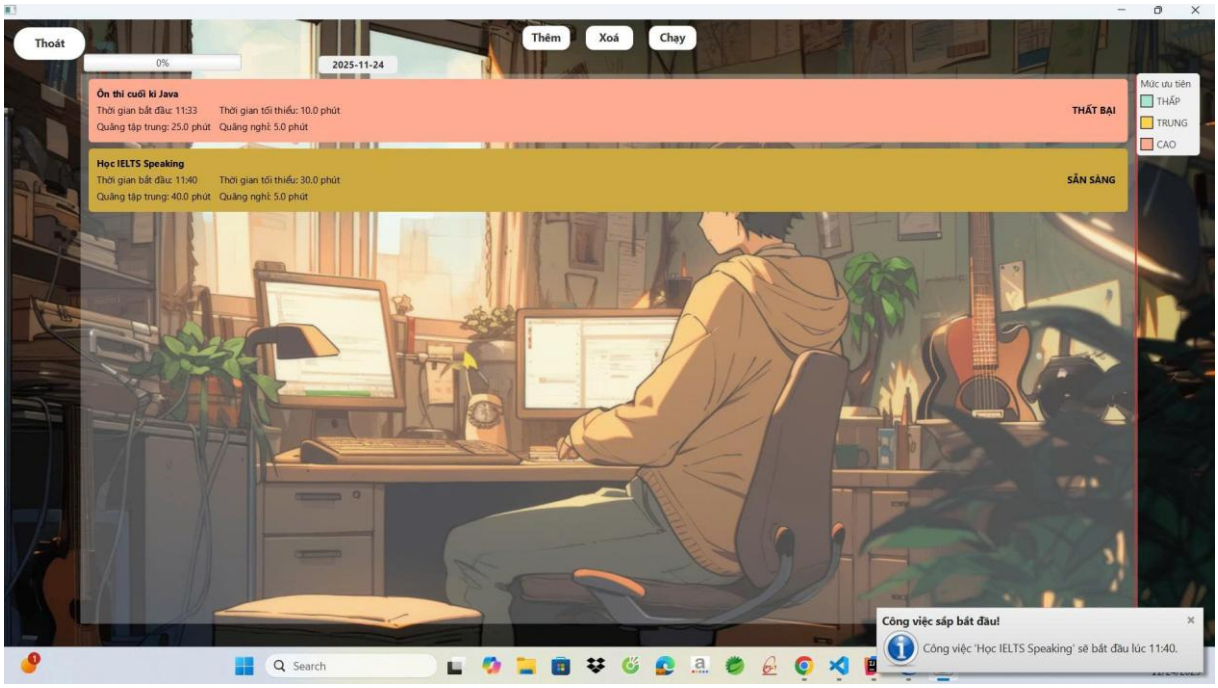
Hình 2. Giao diện cho phép người dùng có thể tải ảnh từ máy tính lên thay đổi giao diện tùy ý

2. Quản lý công việc hiệu quả

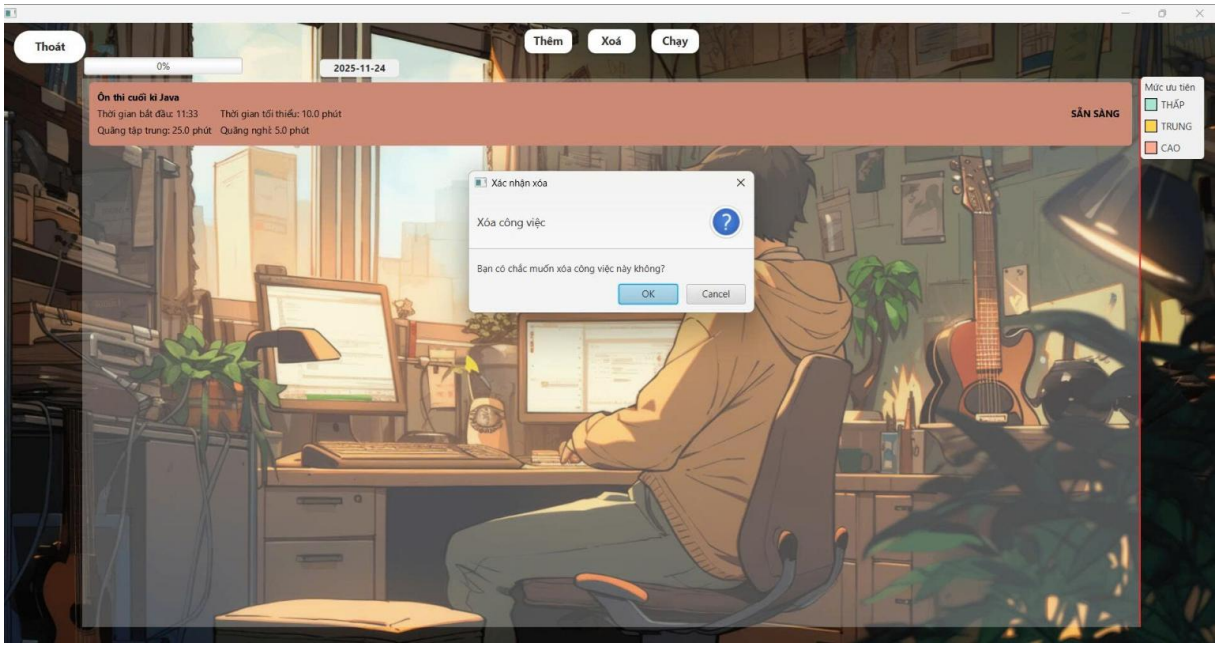
Người dùng có thể tạo mới, chỉnh sửa và xóa các công việc một cách dễ dàng. Các công việc được sắp xếp theo thời gian và mức độ quan trọng, giúp người dùng quản lý thời gian và công việc hiệu quả hơn. Trước 1 phút khi chuẩn bị bắt đầu 1 task nào đó, sẽ xuất hiện thông báo pop-up trên màn hình thông báo công việc chuẩn bị bắt đầu.



Hình 3. Giao diện cho phép người dùng thêm công việc kèm với thời gian, nếu lỗi định dạng HH:mm thì sẽ hiện thông báo “Lỗi định dạng”

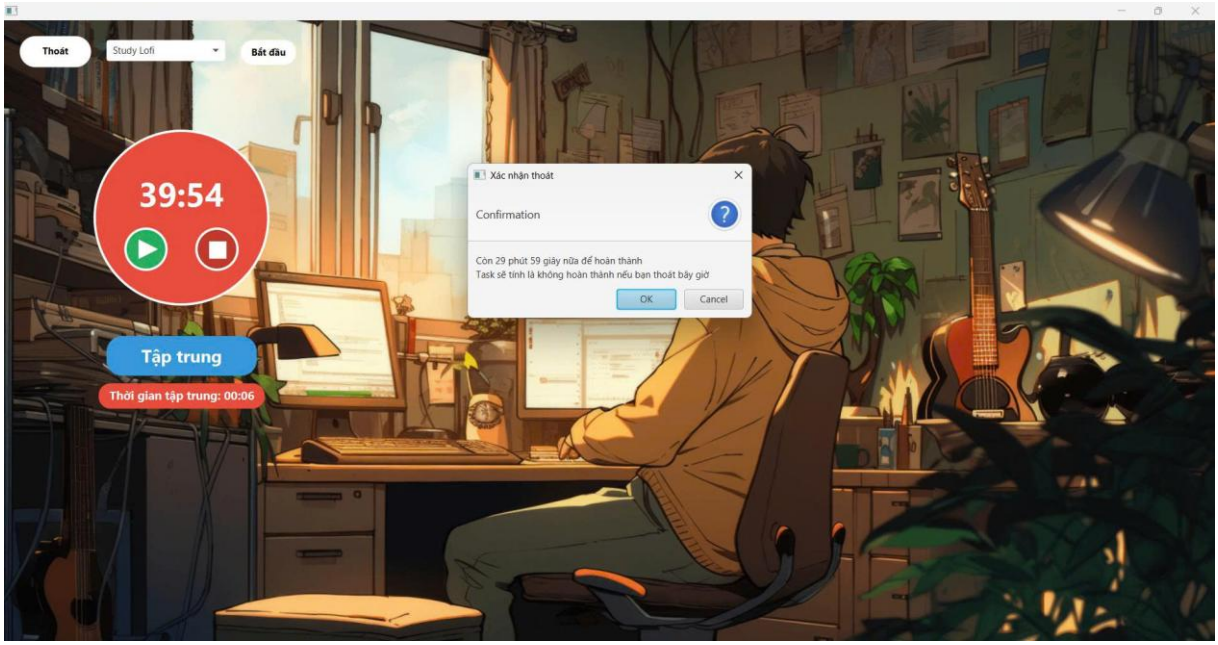


Hình 4. Trước 1 phút khi task nào đó bắt đầu, sẽ xuất hiện thông báo pop-up trên màn hình.



Hình 5. Thao tác xóa công việc

Hình 6: Sau khi xác nhận bắt đầu công việc sẽ xuất hiện cửa sổ Pomodoro



Hình 7: Sau khi xác nhận thoát công việc sẽ hiện thông báo

V. Tổng kết

Trong quá trình xây dựng ứng dụng nhóm em cơ bản đã đạt được các mục tiêu quan trọng bao gồm:

- Tổ chức mã nguồn theo đúng nguyên tắc của lập trình hướng đối tượng (OOP), các thành phần được tổ chức thành các gói và lớp riêng biệt và theo sát cấu trúc mô hình MVC.

- Đã sử dụng được các kiến thức của ngôn ngữ Java như các cấu trúc dữ liệu list, map, triển khai các interface như Initializable, Serializable,...
- Đã áp dụng được JavaFX trong việc xử lý các sự kiện và xây dựng giao diện bằng các thành phần UI như Button, Label, List View,…
- Triển khai mã nguồn đúng với các yêu cầu logic đề ra ban đầu và hoàn thiện ứng dụng.

Mã nguồn của dự án: https://github.com/PhanBinh18/OOP-BTL.git