Notexio Text Editor
Project Report

Project name: Notexio Text Editor
Course: CSE323 (Operating Systems Design)
Student name: Nafisa Tabassum
Student ID: 2221466042
Section: 14
Instructor: Dr. Safat Siddiqui
Semester: Fall 2025
Submission date: 23.12.25
GitHub repository: https://github.com/nafisatabassum30/Notexio
GitHub Pages project page: https://nafisatabassum30.github.io/Notexio/
Demo video: https://youtu.be/5XnIDQdSf2A?si=8Hs_EIlXaYCzMQQs

1. Executive Summary

Notexio is a lightweight, customizable text editor built with Python + Tkinter to demonstrate operating-system concepts through a practical GUI application. The project focuses on file I/O, state consistency (unsaved changes), crash recovery via recovery files, cross-platform behavior (Windows/Linux input + printing), and safe user experience patterns (warnings, recent files, settings persistence).

2. Project Overview

2.1 Goals

- Build a usable text editor with core “Notepad-like” operations.
- Apply OS concepts in a real program: file handling, recovery, background work, resource management, and cross-platform compatibility.
- Provide a clean and modern UX: toolbar, status bar, themes, shortcuts.

2.2 Key Features Implemented

- File I/O: New/Open/Save/Save As, recent files list, unsaved-changes warning.

- Editing: Undo/Redo, find/replace, go-to-line, clipboard operations.
- View/UX: Zoom, fullscreen, optional line numbers, status bar showing cursor position and counts.
- Safety: Recovery files + cleanup, optional auto-save loop.
- Themes: Light/dark mode and customizable colors.
- Export: Export document as **PDF** using ReportLab.
- Platform support: Windows drag-and-drop (optional `tkinterdnd2`), OS-specific printing (`win32print` on Windows, `lp/lpr` on Linux/macOS).

3. Architecture & Module Responsibilities

Notexio is organized as a modular Tkinter application where UI events call into feature managers:

- `main.py`: App composition and menu/shortcut wiring.
- `src/editor.py`: Main text widget, modified-state tracking, and core UI container.
- `src/file_manager.py`: File open/save logic + recent files persistence.
- `src/safety_features.py`: Recovery file creation/cleanup + background auto-save loop.
- `src/theme_manager.py`: Theme application across widgets.
- `src/ui_components.py`: Toolbar, status bar, line numbers UI.
- `src/edit_operations.py`: Find/replace/go-to-line and clipboard operations.
- `src/misc_features.py`: Print / print preview / export as PDF / drag & drop.
- `src/settings_manager.py`: JSON config read/write (`config/settings.json`).

*Design choice: This separation makes each feature set isolated and testable, and mirrors OS design thinking (clear responsibilities, controlled interaction points).

4. Operating Systems Concepts Demonstrated

4.1 *File I/O, Encoding, and Error Handling*

Notexio reads and writes files using explicit encoding (`utf-8`). This touches OS-level concerns:

- File descriptors / handles: OS resources that must be opened/closed correctly (Python context managers handle this reliably).
- Encoding correctness: Prevents corrupted text and ensures consistent storage across machines.
- Failure modes: Permission errors, missing files, locked files — surfaced to the user as dialogs.

4.2 Concurrency and UI Thread Safety (Auto-save / Recovery)

Tkinter requires UI updates to happen on the main thread. Notexio's auto-save runs in a background thread (daemon), which periodically writes recovery files.

Theory: GUI toolkits generally are not thread-safe because widget state is shared; touching it from multiple threads can cause race conditions or crashes. A safe design is:
- Background thread does file I/O only and avoids mutating widgets directly.
- UI changes should be scheduled using the event loop (e.g., `root.after(...)`) if needed.

4.3 Crash Recovery and Data Durability

Recovery files are a simplified durability mechanism:

- Regular snapshots of unsaved content are stored in `recovery/`.
- On startup, the app scans for `.recovery` files and offers restore.
- Old recovery files are cleaned up to avoid disk growth.

*Theory: This imitates OS/file-system reliability ideas (periodic checkpoints; reducing loss after abnormal termination).

4.4 Cross-platform Differences (Input + Printing)

- Mouse wheel events differ across OSs (`event.delta` vs `Button-4/5`).
- Printing pipelines differ: Windows typically uses Win32 APIs; Linux/macOS uses `lp/lpr`.

This is an OS-design reality: identical user features often require platform-specific implementations.

5. Challenges & Fixes (STAR Format)

The course requirement asks for challenges described using STAR:
- Situation (context)
- Task (what needed to be done)
- Action (what I did)
- Result (outcome)

Challenge 1 — Recent Files menu opened the wrong file (late-binding lambda bug)

- Situation: I added an “Open Recent” submenu. Clicking a recent file sometimes opened the last file in the list, not the one clicked.
- Task: Fix the menu so every item opens its correct path.
- Action: I fixed the classic Python late-binding closure issue by capturing the current filepath in the lambda default argument:
  - `command=lambda fp=filepath: open_file(fp)` instead of `command=lambda: open_file(filepath)`
- Result: Each menu entry consistently opens the correct file, improving usability and correctness.

Theory (why it happens): In Python, closures capture variables by reference. When the callback runs, `filepath` has already changed to the last loop value unless you bind it at definition time.

Challenge 2 — PDF export crashed on special characters like `&` or `<`

- Situation: Exporting to PDF failed on some documents; ReportLab raised parsing errors when the text had special characters.
- Task: Make “Export as PDF” robust for real text content.
- Action: I escaped reserved XML/HTML characters before feeding text into `reportlab.platypus.Paragraph`:
  - `&` → `&amp;`, `<` → `&lt;`, `>` → `&gt;`
- Result: PDF export became stable for normal programming text and notes that include symbols.

Theory: ReportLab’s `Paragraph` expects XML-like markup; unescaped characters can break parsing.

Challenge 3 — Auto-save could freeze or behave unpredictably (UI thread-safety)

- Situation: I added auto-save / recovery to protect user work. Early versions risked touching UI state from a background thread.
- Task: Keep auto-save reliable without breaking Tkinter’s single-thread UI rules.
- Action: I designed auto-save as a background loop that performs file writes only, keeping UI rendering and widget updates inside the main event loop.
- Result: Auto-save runs without UI freezes, and recovery snapshots are produced in the background.

Theory: Tkinter widgets are not thread-safe; UI changes must happen on the main thread to avoid race conditions and crashes.

Challenge 4 — Mouse wheel scrolling didn’t work consistently across Windows/Linux

- Situation: Scrolling worked on Windows but failed on Linux (or scrolled in the wrong direction).
- Task: Support scrolling across platforms with consistent behavior.
- Action: I implemented dual handling:
  - Windows/macOS: `event.delta`
  - Linux: `Button-4` and `Button-5`
- Result: Scrolling works across platforms and improves the "feels like Notepad" UX.

Challenge 5 — Recovery files could grow unbounded and waste disk space

- Situation: Recovery snapshots are valuable, but frequent snapshots can flood disk storage over time.
- Task: Keep recovery useful while avoiding uncontrolled disk growth.
- Action: I implemented cleanup logic to keep only the newest N recovery files (default 10), deleting older ones based on modification time.
- Result: Recovery stays effective without filling storage, matching responsible OS resource management.

Theory: Storage is a finite OS resource. Good applications apply retention policies and clean up old artifacts.

Challenge 6 — Modified/unsaved state management (false positives and UX correctness)

- Situation: In text editors, "unsaved changes" must be accurate; false warnings annoy users, and missing warnings can cause data loss.
- Task: Track modified state correctly and integrate it with window title and exit prompts.
- Action: I used Tkinter's `<<Modified>>` virtual event and maintained an `is_modified` flag to:
  - add `*` to the title when content changes
  - prompt on exit or before opening a new file
- Result: Users get correct warnings and visual indicators, reducing accidental data loss.

6. Testing & Validation (What I Verified)

- Open/Save/Save As for `.txt` and arbitrary extensions.
- Recent files list updates and open the correct file.
- Unsaved-changes prompt appears when expected.
- Recovery files are created and older ones are cleaned up.
- Theme toggles apply to major UI components.

- Export as PDF works for normal text and text containing `<`, `>`, `&`.
- Mouse wheel works on Windows/Linux event models.

7. Limitations & Future Work

- Add atomic-save strategy (write to temp + rename) to avoid partial writes if the app crashes mid-save.
- Add unit tests for file operations and settings persistence.
- Improve print preview to render like a real page layout.
- Package for distribution (PyInstaller) with icon + assets bundled.

8. Submission Checklist (What included on GitHub)

- `docs/index.md` (GitHub Pages landing page)
- `docs/Notexio_CSE323_Report.pdf` (the report PDF)
- `docs/report.md` (same report in Markdown for quick reading)
- A demo video link (YouTube) embedded/linked on `docs/index.md`

9. References

- Tkinter documentation (event loop and `<<Modified>>` usage)
- ReportLab documentation (`SimpleDocTemplate`, `Paragraph`)
- OS printing concepts (`lp/lpr`, Win32 print pipeline)

10. Conclusion

Notexio demonstrates OS design concepts through a real GUI application: safe file I/O, graceful OS integration, and reliability-focused features like recovery and safe shutdown. Key lessons were event-driven design, cross-platform behavior, and building safeguards that mirror professional desktop editors.