Notexio — Project Report (CSE323: Operating Systems Design)

Project name: Notexio Text Editor **Course:** CSE323 (Operating Systems Design) **Semester:** _(fill)_ **Student:** _(fill)_ **ID:** _(fill)_ **Instructor:** _(fill)_ **Repository:** _(fill: GitHub repo link)_ **GitHub Pages:** _(fill: published link)_ **Demo video (2–5 min):** _(fill: YouTube/Drive link)_

Short Introduction (What I Built)

Notexio is a lightweight, customizable text editor built with **Python + Tkinter**. While it looks like a productivity app, it was designed as an **Operating Systems (OS) course project** to demonstrate practical OS-facing concepts such as:

- **File I/O and persistence** (open/save, safe writes, encoding decisions)
- **Process/runtime behavior** through event-driven UI loops
- **Concurrency pitfalls** and safe scheduling (auto-save/recovery)
- **Fault tolerance & recovery** (crash/exit recovery files and cleanup)
- **System integration** (printing, PDF export, OS-specific drag-and-drop behavior)

Key features implemented include file management (new/open/save/save-as/recent files), editing tools (find/replace/go-to-line), formatting (fonts/colors/bold/italic/underline), view options (zoom, word wrap, fullscreen, line numbers), safety features (unsaved-change warnings, recovery files, optional auto-save), and utilities (statistics, duplicate-word highlight, extra-space removal).

Project Structure (High-Level Architecture)

The application is modular: each feature family is isolated into its own manager class and connected through the main app:

- `main.py`: application composition, menu wiring, global bindings
- `src/editor.py`: core `Text` widget, modification tracking, UI update hooks
- `src/file_manager.py`: open/save/recent files, unsaved-change workflow
- `src/safety_features.py`: recovery files and auto-save scheduling
- `src/misc_features.py`: print/preview, PDF export, drag-and-drop fallback
- `src/ui_components.py`: toolbar, status bar, line numbers UI

- `src/settings_manager.py`: JSON config persistence (`config/settings.json`)

This separation helped keep the codebase maintainable and made debugging easier (each “OS concept” has a logical home).

Challenges Faced & How I Solved Them (STAR Format)

Below are the most important problems I faced during development, written using the **STAR** method (Situation, Task, Action, Result). I focused on technical issues that connect directly to OS-level design ideas: correctness under concurrency, I/O correctness, robustness, and user-facing reliability.

1) Auto-save caused intermittent crashes / undefined behavior (Thread-safety + UI concurrency)

Situation: I implemented auto-save using a background thread to avoid blocking the UI. However, the thread accessed the Tkinter `Text` widget to read content for recovery files. This caused intermittent crashes or odd behavior, especially when typing while auto-save ran.

Task: Make auto-save reliable without freezing the UI and without unsafe cross-thread UI access.

Action: I redesigned auto-save to run **entirely on Tkinter’s event loop** using `root.after(...)` scheduling. This ensures all widget reads happen on the **main/UI thread**, which is the only thread Tkinter safely supports.

Result: Auto-save became stable and deterministic—no random UI crashes, no race-y reads, and recovery files are generated safely at the configured interval.

Theoretical note (why this matters in OS terms):

- Tkinter is similar to a single-threaded event-driven process: UI objects are not protected by locks and are not thread-safe.
- In OS design terms, the earlier approach was a classic **concurrency bug**: shared mutable state (UI widget internals) was accessed by multiple execution contexts without synchronization.
- The fix is equivalent to enforcing an “ownership” rule: all UI state belongs to the UI thread, and periodic work is scheduled cooperatively.

2) Line numbers didn’t scroll with the document (UI state synchronization bug)

Situation: I added a line numbers sidebar. The numbers updated when text changed, but when I scrolled the document, the line number panel stayed "stuck," creating a mismatch between the visible text and the visible line numbers.

Task: Keep the line-number view synchronized with scroll events while preserving smooth performance.

Action: I updated the mouse-wheel scroll handler to refresh line numbers whenever scrolling occurs (only when line numbers are enabled). This keeps both views aligned without heavy recomputation in the common case where line numbers are off.

Result: Line numbers now scroll in sync with the text view and remain visually correct, improving usability and correctness.

Theoretical note:

- This is a synchronization problem between two views of the same underlying state (document content and viewport position).
- In OS UI/tooling terms, it resembles keeping two "buffers" consistent under user-driven events.

3) Recent Files menu opened the wrong file (Closure capture / late binding bug)

Situation: When populating the "Open Recent" menu, clicking any recent item sometimes opened the **same** file (often the last file in the list). This happened because the menu commands were created inside a loop.

Task: Ensure each menu item reliably opens its own file path.

Action: I fixed the closure capture issue by binding the loop variable as a **default argument** to the lambda (e.g., `lambda fp=filepath: open_file(fp)`), preventing Python's late binding from reusing the final loop value.

Result: Each recent-menu item now opens the correct file every time.

Theoretical note:

- The root issue was *language/runtime behavior* (late binding) rather than OS behavior, but it impacted correctness of user-facing file I/O.
- Reliability in OS tooling depends on correct binding between UI actions and file system operations.

4) “Unsaved Changes” logic had edge cases (State machine correctness)

Situation: A text editor has a small but critical state machine: *clean vs modified*, and *file path set vs not set*. I saw edge cases where the status bar/window title didn’t always match the real state after operations like New/Open/Save.

Task: Make state transitions consistent and ensure the UI always reflects the true document state.

Action: I centralized modification tracking in the editor (via Tk’s `&lt;&lt;Modified&gt;&gt;` event), then ensured file operations explicitly reset the modified flag and refresh the title/status bar after New/Open/Save. I treated the editor state like a state machine:

- `current_file: None | path`
- `is_modified: True | False`
- UI title indicator `*` derived from `is_modified`

Result: The “*” unsaved indicator, window title, and status bar remain consistent across workflows, reducing user confusion and preventing accidental data loss.

Theoretical note:

- State-machine correctness is a common OS concern (e.g., file descriptor lifecycle, process states).
- This editor applies the same idea: transitions must be explicit and UI must reflect real state.

5) Recovery files needed cleanup and naming discipline (Storage management)

Situation: Recovery files are helpful, but without cleanup they can fill disk space over time and become hard to manage. Also, if multiple recovery files exist, users need a sensible restore order.

Task: Implement a recovery system that is safe, bounded, and user-friendly.

Action: I generated recovery files with timestamps and maintained a rolling cleanup policy (keep most recent N recovery files). On startup, the app scans the recovery directory, sorts by modification time, and offers to restore the newest file.

Result: Recovery is reliable and disk usage is bounded. Users get a simple “restore most recent?” choice instead of a confusing pile of files.

Theoretical note:

- This mirrors OS log rotation and crash recovery patterns: keep recent checkpoints, discard old ones, and define deterministic restore selection.

6) PDF export broke on special characters (Escaping + document formatting)

Situation: PDF export uses ReportLab `Paragraph`, which interprets input as a markup-like format. If the document contains `&amp;`, `&lt;`, or `&gt;`, export can fail or render incorrectly.

Task: Ensure PDF export works for arbitrary text content.

Action: I sanitized paragraph text by escaping reserved characters before creating `Paragraph` objects. I also preserved line breaks by splitting into paragraphs and spacing them consistently for readability.

Result: PDF export works with real-world text (including code snippets and symbols), producing clean output without crashing.

Theoretical note:

- This is analogous to safe boundary handling between a “user space” buffer and an output subsystem: you must validate/escape inputs before handing them to a formatter.

7) Cross-platform behavior differences (OS integration constraints)

Situation: Features like printing and drag-and-drop behave differently across Windows/Linux/macOS. Some capabilities require optional dependencies (e.g., `pywin32`, `tkinterdnd2`) and system commands (`lp`/`lpr`).

Task: Provide best-effort OS integration while keeping the core editor portable.

Action: I implemented a capability-based approach:

- Use Windows APIs when available (via `pywin32`)
- Use platform-native print commands on Linux/macOS
- Make drag-and-drop optional and gracefully degrade when the dependency is missing
- Keep the core editor independent from platform-specific modules

Result: The editor remains functional on all platforms, while advanced integration features activate when the OS and dependencies support them.

Verification / Test Plan (What I Did to Validate)

- Open/Save/Save As: verify UTF-8 reading/writing and correct filename/title updates
- Unsaved changes prompt: verify Yes/No/Cancel paths
- Recent files: verify correct file opens from menu list
- Recovery: create edits → close without saving → confirm recovery file → restore at startup
- Line numbers: toggle on → scroll and type → ensure sync and correctness
- PDF export: include special characters (`&amp; &lt; &gt;`) and verify export succeeds

GitHub Submission Requirements (From the Course)

GitHub folder submission (what to include)

Include these in your repository:

- `docs/Notexio_CSE323_Project_Report.md` (this report source)
- `docs/Notexio_CSE323_Project_Report.pdf` (final PDF)
- `docs/index.md` (GitHub Pages landing page)

GitHub Pages link submission (how to publish)

1. Push the `docs/` folder to GitHub. 2. In GitHub repo settings: **Settings** → **Pages** 3. Set **Source** to “Deploy from a branch” 4. Select your default branch and choose folder: `/docs` 5. Your GitHub Pages URL will become: `https://&lt;username&gt;.github.io/&lt;repository&gt;/`

On the landing page, link to the PDF so your instructor can open it in one click.

2–5 Minute Demo Video (Suggested Script)

0:00–0:20 — Intro

- “Hi, this is Notexio, a Python Tkinter text editor for CSE323 OS Design.”
- “I focused on OS concepts: file I/O, recovery, safe scheduling, and system integration.”

0:20–1:20 — Core workflow

- New → type text → show unsaved `*`
- Save As → reopen → show recent files menu

1:20–2:20 — OS-focused features

- Trigger recovery: modify file, close, reopen, restore
- Turn on line numbers and scroll
- Show statistics + remove extra spaces

2:20–3:10 — Export/Integration

- Export as PDF and open the output
- Mention print support and cross-platform behavior

3:10–4:30 — “Challenges & Fixes” highlight

- Explain the auto-save thread-safety issue and the fix (Tk `after`)
- Mention recent-files closure bug fix

4:30–5:00 — Wrap up

- Summary of what you learned (I/O correctness, event loops, recovery design)
- Show GitHub Pages link with the report PDF

Conclusion

Notexio demonstrates OS design principles through a practical tool: reliable file I/O, safe scheduling, recovery design, and platform-aware integrations. The largest learning outcome was treating “small UI features” with OS-grade discipline—correctness, robustness, and predictable state transitions.

Future Improvements (Optional)

- Add atomic “safe save” (write temp file + rename) to prevent partial writes on crash
- Add per-file encoding detection and selectable encoding in UI
- Add tabbed editing support end-to-end (the UI scaffolding exists)
- Add structured logging for easier diagnostics