Notexio — CSE323 (Operating Systems Design) Project Report

Project: Notexio Text Editor (Python + Tkinter)

Course: CSE323 — Operating Systems Design

Author: _(your name)_

Date: 20 Dec 2025

Repository: _(paste your GitHub repo link here)_

1) Short Introduction (What is Notexio?)

Notexio is a lightweight, Windows Notepad–style text editor built using Python and Tkinter. The project was developed as an OS-focused application to demonstrate practical operating-system interactions through:

- File I/O (open/save/save-as, encoding handling)
- Persistent configuration (JSON settings + recent files)
- Crash safety (recovery snapshots + restore)
- Cross-platform behavior differences (Windows vs Linux/macOS printing, drag & drop)
- Event-driven programming (Tkinter main loop, UI events, background timers)

Notexio is structured into focused modules under `src/` (editor, file manager, safety features, tools, view manager, theme manager, etc.), with settings stored in `config/settings.json` and recovery snapshots stored in `recovery/`.

2) Key Features Implemented (Technical Summary)

File system operations (OS I/O concepts)

- **Open/Save/Save As** with UTF-8 encoding (`src/file_manager.py`)
- **Unsaved-changes detection** before destructive actions (new/open/exit)
- **Recent files list** persisted to JSON settings

Safety + reliability (fault tolerance)

- **Recovery snapshots** written to `recovery/` using timestamps (`src/safety_features.py`)
- **Recovery restore** prompt at startup (wired in `main.py`)

- **Auto-save timer** implemented via a background thread that schedules safe UI-thread snapshots

View + UX

- Word wrap toggle (`src/view_manager.py`)
- Zoom in/out/reset with a persistent base font size (View ↔ Formatter integration)
- Optional line numbers sidebar (`src/ui_components.py`)
- Theme support (light/dark/custom) with status bar + toolbar recoloring (`src/theme_manager.py`)

Tools

- Word/character/line count, reading-time estimate (`src/tools.py`)
- Duplicate-word highlighter using regex + tag-based highlighting

PDF export + printing (system integration)

- Export to PDF using ReportLab (`src/misc_features.py`)
- Print support using platform-specific paths:
- Windows: `pywin32` (ShellExecute printing)
- Linux/macOS: `lp`/`lpr` system commands

3) Operating Systems Concepts Demonstrated

3.1 File I/O and persistence

Notexio uses classic OS file operations:

- Opening files via paths returned by the OS file dialog
- Reading and writing with explicit encoding (`utf-8`)
- Handling exceptions (permission errors, missing files, invalid paths)

It also persists “process state” across runs through `config/settings.json`:

- window size
- theme preference
- recent files list
- auto-save preferences
- view preferences (word wrap, line numbers)

3.2 Processes/threads and the UI event loop

Tkinter is **single-threaded**: its widgets are not thread-safe. Notexio uses:

- The **Tk main loop** for event-driven UI updates (keypress, clicks, Modified events)
- A **background thread** only for timing (auto-save interval), and it schedules UI-safe work using `root.after(...)`.

This mirrors a common OS design principle: background “workers” must not directly mutate UI state; instead they pass work to the main event loop (a message-passing approach).

3.3 Reliability (recovery files)

Recovery files are a simplified crash-recovery approach:

- A snapshot is written periodically (or on exit) to a separate recovery directory.
- On startup, the app scans the recovery directory and can restore the most recent snapshot.

Conceptually, this is similar to:

- **journaling** / write-ahead logs (WAL): keep a safe copy of state to recover from crashes
- “last known good state” strategies

4) Challenges Faced & How I Solved Them (STAR Format)

This section lists the most important problems I faced during development and how I resolved them, using the **STAR** format.

Challenge 1 — Unsaved changes causing accidental data loss

- **Situation**: While testing open/new/exit flows, I repeatedly lost edits because nothing prevented users from discarding changes.
- **Task**: Add safe-guards like real editors (Notepad) so destructive actions warn the user and optionally trigger a save.
- **Action**: Implemented a single check path (`check_unsaved_changes`) and called it from new/open/exit workflows. The dialog supports **Yes/No/Cancel**, and “Yes” routes into Save/Save-As as needed.
- **Result**: Data loss was prevented, and the behavior became predictable and professional—matching user expectations for desktop editors.

Challenge 2 — Auto-save + Tkinter thread-safety (hard crash / random exceptions)

- **Situation**: I initially implemented auto-save using a background thread that directly read the Tk text widget. During longer sessions, I saw intermittent UI crashes or “Tcl” errors.
- **Task**: Make auto-save reliable without freezing the UI.
- **Action**: Learned that Tkinter widgets must only be accessed from the main thread. Kept the background thread **only for timing**, then used the Tk event loop (`root.after(0, ...)`) to execute snapshot creation safely on the UI thread.
- **Theory**: Many GUI toolkits (Tk, Qt, Win32 UI) enforce a single UI-thread model. Accessing UI objects from other threads is undefined and can corrupt internal widget state. Using an event loop callback is effectively message passing.
- **Result**: Auto-save became stable and never blocked typing. Recovery snapshots can now be created safely at intervals.

Challenge 3 — Recovery file design (keeping snapshots useful without filling disk)

- **Situation**: A naive recovery strategy can generate infinite files and waste storage.
- **Task**: Keep recovery helpful while controlling disk usage.
- **Action**: Designed recovery snapshots to include a timestamp and (when available) original filename. Implemented cleanup to keep only the most recent N snapshots.
- Stored snapshots in a dedicated `recovery/` directory for easy discovery and isolation from user documents.
- **Result**: Recovery is effective and low-risk: users can restore recent work, and storage remains bounded.

Challenge 4 — Cross-platform printing differences (Windows vs Linux/macOS)

- **Situation**: Printing is not a single portable API across OSes. Windows provides a “Shell print” path, while Linux/macOS typically rely on CUPS commands.
- **Task**: Implement printing in a way that works on multiple OS environments.
- **Action**: Implemented platform branching:
- Windows: optional `pywin32` to print via `ShellExecute`.
- Linux/macOS: write to a temp file and call `lp`/`lpr`.
- Added error/warning paths when dependencies are missing.
- **Theory**: OS design differs in how device I/O is exposed. Windows printing is tightly integrated into the GUI shell; Unix-like systems often expose printing via command-line spooling systems.
- **Result**: Printing works where supported and degrades gracefully with clear user guidance.

Challenge 5 — Maintaining a clean architecture as features grew

- **Situation**: Adding features (themes, tools, PDF export, line numbers, zoom) risked turning the code into one large file with tangled state.
- **Task**: Keep the project maintainable and readable for a course deliverable.
- **Action**: Split responsibilities into modules (editor, file manager, view manager, tools, theme manager, safety features, UI components). Used shared references intentionally for cross-feature coordination (e.g., ViewManager updates the status bar zoom label).
- **Result**: The codebase is easier to navigate, test, and explain in a demo, and each feature has a clear "home."

Challenge 6 — PDF export and special-character correctness

- **Situation**: PDF export initially broke when the document contained characters like &, <, or >, because ReportLab's `Paragraph` interprets a subset of HTML/XML.
- **Task**: Export documents reliably regardless of content.
- **Action**: Escaped special characters (&, <, >) before creating `Paragraph` objects so user text is treated as content, not markup.
- **Theory**: Many document pipelines treat text as structured markup. When raw user input is interpreted as markup, it can cause parse errors or unexpected rendering. Escaping converts reserved tokens into safe literals.
- **Result**: PDF export became robust for typical user text and consistent across documents.

5) What I Would Improve Next (Optional Enhancements)

- **Atomic saves**: write to a temporary file and rename to reduce corruption risk if the process is killed mid-write.
- **Encoding detection**: attempt UTF-8 first, then fall back to other encodings with a clear UI choice.
- **Stronger recovery semantics**: store recovery metadata (original path hash, checksum) to prevent restoring the wrong snapshot.
- **More unit tests** around file operations and settings serialization.

6) Submission Requirements (From Course Sheet)

GitHub page link submission

- **GitHub Pages link**: _(TODO: paste your GitHub Pages URL here if you host a project page)_

Video demo (2–5 minutes)

- **Video link**: _(TODO: paste unlisted YouTube / Drive link)_

- **Suggested demo flow**:
- Show open/save/save-as + recent files
- Toggle theme + zoom + word wrap + line numbers
- Demonstrate find/replace and tools (word count, duplicate highlight)
- Trigger recovery: modify, close without saving, reopen and restore
- Export as PDF and show resulting file

GitHub folder submission

- This report is included in the repository under `docs/`.