Notexio

CSE323 — Operating Systems Design

Project Report (STAR-format challenges)

Project name	Notexio Text Editor
Course	CSE323 — Operating Systems Design
Semester/Section	[Fill]
Student name	[Fill]
Student ID	[Fill]
Instructor	[Fill]
Date	2025-12-20

This report follows the course submission requirements: it includes project challenges in STAR format and provides a GitHub submission checklist (report PDF, GitHub Pages link placeholder, and demo video plan).

1) Abstract

Notexio is a lightweight text editor built with Python and Tkinter. The project was designed as an OS-oriented application to practice file I/O, data persistence, event-driven GUI programming, and safe recovery mechanisms in the presence of crashes, forced exits, or unexpected shutdowns. The final system supports open/save, undo/redo, find/replace, usability features (status bar, line numbers, zoom, themes), and OS-adjacent integrations (printing via platform tools and exporting to PDF).

2) Project Overview

2.1 Goals

• Build a stable, user-friendly editor that behaves like a modern Notepad-style app.<br/>• Demonstrate OS concepts through real features: file system interaction (open/save), concurrency considerations (auto-save without blocking UI), process/tool invocation (printing), and fault tolerance (recovery files).

2.2 Key Features Implemented

• File operations: New/Open/Save/Save As, Open Recent, unsaved-change prompts<br/>• Editing: Undo/Redo, clipboard operations, Find/Replace, Go To Line<br/>• View: Zoom, word wrap, fullscreen, optional line numbers<br/>• Tools: statistics, reading time, duplicate-word highlight, remove extra spaces<br/>• Themes: light/dark/custom<br/>• Safety: optional auto-save, recovery files, warn on exit<br/>• Export/print: export as PDF, print preview, printing (platform-dependent)

3) System Design & Architecture

Notexio uses a modular architecture. main.py wires together the components and passes shared references so modules can coordinate without duplicating state.

3.1 Modules (high-signal responsibilities)

• src/editor.py: Tk root + main text widget; modification tracking and title updates<br/>• src/file_manager.py: open/save, recent files, unsaved-change prompts<br/>• src/edit_operations.py: find/replace/go-to-line + clipboard + undo/redo<br/>• src/formatter.py: font and visual formatting controls<br/>• src/view_manager.py: zoom/word-wrap/fullscreen logic<br/>• src/tools.py: statistics + cleanup utilities<br/>• src/theme_manager.py: theme propagation across UI<br/>• src/safety_features.py: auto-save + recovery snapshots<br/>• src/misc_features.py: printing and PDF export<br/>• src/settings_manager.py: JSON persistence (config/settings.json)

3.2 OS concepts reflected

• File I/O + persistence: explicit open/read/write; recent files and preferences persisted in JSON<br/>• Reliability: recovery files act as journaling-lite for user text<br/>• Concurrency model: GUI event loop; background timing must not block UI<br/>• System integration: printing delegates to OS tools/APIs

4) Challenges Faced (STAR format)

Challenge 1 — Dirty state incorrect after programmatic loads/saves

<b>Situation:</b> Tkinter’s Text widget can remain marked as “modified” after code-driven inserts/clears, causing false unsaved-change prompts.

<b>Task:</b> Make the “modified” indicator reflect real user edits only.

<b>Action:</b> Reset Tk’s internal modified flag after open/new/save operations and keep the app-level dirty flag synchronized with title/status.

<b>Result:</b> Opening/saving no longer triggers false prompts; the title shows * only when the user actually edits.

<i>Theory:</i> Widgets maintain internal state for change events; applications must explicitly acknowledge when changes are intentional (file load) vs user edits.

Challenge 2 — Auto-save instability due to thread-unsafe UI access

<b>Situation:</b> Auto-save used a background thread for timing, but reading Tkinter widgets from non-UI threads is unsafe and can crash intermittently.

<b>Task:</b> Keep UI responsive while ensuring recovery snapshots are created safely.

<b>Action:</b> Use root.after(...) to schedule snapshot creation on the UI event loop; the worker thread only sleeps and triggers scheduling.

<b>Result:</b> Auto-save remains non-blocking while eliminating thread-safety crashes.

<i>Theory:</i> GUI frameworks often require thread confinement: all widget access must occur on the event-loop thread; after() safely queues work there.

Challenge 3 — Recovery files could grow without bounds

<b>Situation:</b> Recovery snapshots are intentionally redundant, but without retention they can fill disk over time.

<b>Task:</b> Keep recovery useful while preventing uncontrolled storage growth.

<b>Action:</b> Implemented a retention policy that keeps only the most recent recovery files (sorted by modification time) and removes older ones.

<b>Result:</b> Recovery remains available for recent work while storage stays bounded.

<i>Theory:</i> Bounded logs and log rotation are common OS patterns to prevent resource exhaustion.

Challenge 4 — PDF export failed on special characters

<b>Situation:</b> ReportLab Paragraph accepts markup-like text; unescaped &, <, > can break parsing for normal user content.

<b>Task:</b> Make PDF export robust for code snippets and symbol-heavy text.

<b>Action:</b> Escaped reserved characters before creating Paragraph objects and built a simple paragraph/spacing layout.

<b>Result:</b> PDF export works reliably across typical user text.

<i>Theory:</i> When a renderer supports markup, raw user text must be escaped to avoid accidental interpretation.

Challenge 5 — Printing behavior differed by OS

<b>Situation:</b> Windows printing typically uses Win32 APIs, while Linux/macOS often use lp/lpr; dependencies may be missing.

<b>Task:</b> Provide printing with graceful fallbacks and clear feedback.

<b>Action:</b> Used OS detection: pywin32 path on Windows when available; otherwise invoke lp/lpr on Unix-like systems and show guidance if unavailable.

<b>Result:</b> Printing works where supported; users get actionable guidance if tooling/dependencies are missing.

<i>Theory:</i> This reflects a common OS abstraction boundary: portable UI logic with OS-specific system services selected at runtime.

Challenge 6 — Status bar column index mismatched user expectations

<b>Situation:</b> Tkinter reports cursor column indices as 0-based, but Notepad-style UIs show 1-based columns.

<b>Task:</b> Match user expectations for professional UX.

<b>Action:</b> Converted internal column values to 1-based for display.

<b>Result:</b> Status bar now shows Ln 1, Col 1 at document start, matching Notepad behavior.

5) Validation / Test Plan (manual)

• File I/O: open/edit/save/re-open; Save As default extension; recent files update<br/>• Safety: unsaved-change prompt; recovery restore on startup; auto-save snapshot creation<br/>• Export/print: export PDF containing < & >; print preview displays content<br/>• UX: status bar line/column and * indicator; zoom label updates; dark mode theming

6) How to Run

pip install -r requirements.txt<br/>python main.py

7) Submission Package (matches course requirements)

• GitHub folder submission: docs/Notexio_CSE323_Project_Report.pdf (and optional .md source)<br/>• GitHub page link submission: enable GitHub Pages and paste link here: [Fill after enabling]<br/>• Submission video demo (2–5 minutes): show open/save, find/replace, theme toggle, export PDF, recovery behavior; paste link: [Fill]<br/>• Short intro: Notexio is a Notepad-style editor emphasizing OS concepts: persistence, reliability, and system tool integration.