**Project Reflection**

**Successes**

The making of my project zomboid modding tool uses modularity, separating the core python logic from platform specific scripts and operations making it easy to maintain. Cross platform compatibility was done by using dynamic script selection based on the operating system pyhton is running on.

**Challenges and Solutions**

Several technical came up during development. Path issues happened due to differences between Windows and Linux file systems. This was resolved by adopting Python’s os.path for consistent path handling. Subprocess execution errors occasionally crashed mod installations; these were fixed through error trapping and detailed logging. Another challenge involved registry or script corruption during unexpected shutdowns which was fixed by more error handling.

**Key Lessons Learned**

This project shows the importance of user-centered design. Early versions of the tool wasn’t as easy to use and vague. So the help system and interactive workflows were expanded. Logging, became critical for diagnosing issues. Also a lot of edge-case testing particularly for path validation and special character handling prevented many potential failures.

**Future Development Directions**

While the current version meets core requirements, several enhancements could expand its use. A graphical user interface would make the tool accessible to non-technical users. Support for mod dependencies—such as automatic installation of required assets—would streamline complex mod setups. Integration with the Steam Workshop API could enable direct publishing and downloading of mods.

**Evolution of Design**

The system underwent significant architectural changes during development. The initial monolithic design, which combined all functionality in a single script, was refactored into discrete components:

* The ModManager class for core operations.
* Platform-specific scripts for installation and file management.
* External JSON and log files for data persistence.

Another major revision involved mod type configuration. Originally, supported mod categories (e.g., Textures, Sounds) were hardcoded. This was replaced with a configurable MOD\_TYPES dictionary, allowing users to define custom content structures.

**Conclusion**

This project evolved from a basic automation script into a robust mod management tool through constant refinement. The experience highlighted the balance required between automation and flexibility—users needed prescriptive guidance for common tasks while retaining control for advanced customization. Future efforts would benefit from earlier focus on error handling and user experience testing.