GNUStep Architectural Enhancement

Group 14: ArchiTECHS

Kamana Chapagain 21kc77@queensu.ca
Sareena Shrestha 21ss377@queensu.ca
Yashasvi Pradhan 21yp26@queensu.ca
Justin Li 21jjl13@queensu.ca
Saachi Singh 22ss62@queensu.ca
Shreya Menon 22sm47@queensu.ca

CISC 322/326
Assignment 3: Proposal for Enhancement
April 4th, 2025

Table of Contents

Abstract

In this report, we propose an architectural enhancement to GNUstep. This is done through introducing a new subsystem, libs-autosave, which enables version history and an autosave functionality. We identified the need for this subsystem because GNUstep currently lacks built-in support for autosaving or restoring previous versions of the user's work. This causes users to rely on external tools in case progress is lost due to crashes or human error. Our proposed enhancement addresses this by periodically and automatically saving file states and allowing users to browse and restore previous versions within the GNUstep environment.

We analyze two approaches to implement this. The first involves extending the existing core components, i.e., libs-base and libs-gui, while the second involves introducing a new subsystem (libs-autosave) responsible for all version control functionality. To compare these approaches, we perform a SAAM analysis focused on stakeholders (users and developers) and non-functional requirements (performance, maintainability, testability, usability, evolvability, reusability, and portability)

Following the analysis, we conclude that the new subsystem approach is superior. We then examine the effects this enhancement introduces. We learn that it affects both high- and low-level architecture by introducing new dependencies between libs-autosave, libs-base, libs-gui, and apps-gorm while maintaining GNUstep's layered structure. Subsystem interactions and future performance implications are also taken into consideration. We use two use cases - automatic versioning after edits and restoring a previous version - via the interface to understand the workings of the feature better. Additionally, we go over concurrency and understand that this enhancement introduces no new concurrency requirements as autosave operations are triggered synchronously and rely on existing thread-safe mechanisms. Finally, we consider potential risks, testing strategies and limitations. This gives us a comprehensive overview of the subsystem's impact on GNUstep.

Introduction

Enhancing a complex system like GNUstep involves careful architectural design and planning. This is why planning is essential. We recognized the importance of evaluating different design approaches, determining interactions between subsystems, introducing dependencies between existing components and looking at how these changes impact both developers and users.

Once the most suitable approach is selected, its impact on the architecture is carefully examined. This means we needed to determine any new components or dependencies, understand how the enhancement integrates with existing subsystems and how it impacts various files, directories etc. We also consider whether the change introduces any concurrency requirements. Finally, analyzing the use cases helps us know how the feature will function under real operation. It is also important to establish a clear testing strategy, explore any risks introduced address potential limitations

This report walks through the planning and evaluation process for our proposed enhancement to GNUstep. We aim to eliminate the reliance on manual saving or external tools by providing users with the ability to automatically save progress and recover previous versions

Proposed Enhancement and Motivation

When working with development tools like GNUstep, one common frustration users face is losing their progress due to accidental closures, unexpected crashes, or simply forgetting to save files regularly. Currently, GNUstep does not offer an autosave feature, meaning users must manually save their work to avoid losing progress. Additionally, if a mistake occurs or unwanted changes are made, there's no straightforward way to revert to an earlier version without external version control tools. While external tools do exist, integrating these functionalities directly into GNUstep would substantially enhance usability and efficiency.

Our proposed solution is to implement a "Version History Autosave" feature directly within GNUstep. This feature would automatically save the user's work either periodically or whenever significant edits are made, similar to the version control functionality in tools like Google Docs. Users could then easily revert to previous versions of their projects directly from within the GNUstep interface, eliminating the need to depend on external tools.

Incorporating an integrated autosave and version history feature would greatly enhance the user experience by reducing frustration and minimizing the risk of losing important work. It would allow developers to concentrate more on coding and less on managing manual saves or recovering lost data. This enhancement would also simplify troubleshooting and experimentation, as users could confidently make changes knowing they can quickly revert to a previous version if necessary. By integrating version control directly into GNUstep, developers would benefit from a more seamless workflow and overall, an improved development experience.

Approach A

The first approach is to introduce a new subsystem called *libs-autosave* to handle autosave and version history functionality. This component would manage automatic versioning by saving user changes at regular intervals or when meaningful edits occur. These versions would be stored and made accessible within GNUstep, allowing users to view and restore earlier states of their work. Existing components like libs-base and libs-gui would interact with *libs-autosave* to trigger saves and retrieve version data when needed. Since GNUstep already follows a layered, object-oriented architecture, this addition naturally fits into the system. The responsibilities would remain well-separated, and the object-oriented design would ensure clean communication between modules using defined interfaces, allowing the new feature to be integrated without disrupting the architecture's clarity or maintainability.

Approach B

The second approach is to embed autosave and version history functionality directly into existing components, primarily libs-base and libs-gui. In this case, file handling and event-related classes within libs-base could be expanded to detect and save changes periodically, while libs-gui could incorporate interface elements for browsing and restoring previous versions. This approach builds directly on the existing structure without introducing a new subsystem, using the current subsystems to support the added functionality. GNUstep's object-oriented architecture supports these kinds of extensions, allowing updates to existing subsystems without altering the architectural style. This method would distribute the version control responsibilities across components that already manage related behaviors, offering an alternative way to support the enhancement within the system's current structure.

SAAM Analysis

The Software Architecture Analysis Method (SAAM) is a technique used to assess the impact of architectural decisions on a software system's quality attributes, particularly focusing on how well the proposed features address the needs of stakeholders and support non-functional requirements (NFRs). In this analysis, we compare two different architectural approaches for implementing the proposed Version History Autosave feature in GNUstep. This feature automatically saves previous states of user projects or interface designs, allowing users to restore or browse past versions.

Stakeholder Analysis

End users

The primary users of GNUstep are the end users. This means developers and designers who use and interact with Gorm and/or GNUstep libraries to build their own software. Their experience with the system depends heavily on experience-based factors such as usability and performance.

For users, the autosave system should be simple and non-intrusive. The ability to recover an earlier version of a design or project should feel straightforward as well, and users shouldn't have to manually manage versions or worry about data loss in case of a crash.

From a usability standpoint, it's important that version history integrates naturally into the GUI, possibly through contextual menus or a version timeline without cluttering the interface. Performance is also critical, therefore, any background saving process should not cause a noticeable lag in the interface or interrupt ongoing tasks. If versioning causes Gorm or other applications to freeze up or respond slowly, the feature could end up causing more frustration than benefit.

Since GNUstep is cross-platform, the behavior of this new feature should also be portable. Users should expect consistent behavior across macOS, Linux, and Windows systems. Lastly, storage

efficiency and control become relevant if many versions are stored per project. While users may appreciate having access to older versions, they don't want the system to silently consume gigabytes of space without limits or control.

Maintainers/Developers

The secondary stakeholder group is the GNUstep maintainers and contributors. For them, non-functional concerns like maintainability, testability, and evolvability are the most important.

A poorly designed versioning system could quickly become complicated with existing GUI or base libraries, increasing coupling and making debugging or testing harder. Developers need the feature to be modular and self-contained so that any future fixes or upgrades to autosave/version control don't require modifying foundational components like libs-base or libs-gui.

Testability is also a concern. Being able to write unit tests for versioning logic (e.g. detecting changes, creating snapshots, or restoring versions) is much easier if the logic is encapsulated in its own component. If versioning behavior is spread across multiple layers, testing becomes more fragile and prone to regressions.

Another aspect developers may also be thinking about is reusability, meaning if the version control logic is cleanly separated, it could be reused in other GNUstep tools, like project editors or IDE-style environments. This would reduce code duplication and speed up future development. Evolvability matters too—it should be easy to extend the versioning system to support more features like branching, timestamped checkpoints, or even integration with remote storage systems, without touching unrelated parts of the platform.

Non-Functional Requirements

NFR	Approach A: New subsystem (libs-autosave)	Approach B: Integrate into existing core components (libs-base/libs-gui)
Usability	**High:** Allows users to access version history through clear, distinct UI affordances or menus.	**High:** Through abstraction, the feature will appear the same to the end user, even if the underlying implementation is more complex.
Performance	**High:** A dedicated subsystem avoids bloating existing subsystems, likely keeping performance stable and predictable.	**Medium:** May degrade slightly due to added responsibilities to existing subsystems, which were not designed without the new features in mind.
Maintainability	**High:** A standalone subsystem is relatively easier to debug, log, and maintain.	**Low:** Changes to the feature are very likely to require modifying

		libs-base/libs-gui, making the system harder to understand, maintain, and modify.
Testability	**High:** A standalone subsystem is easier to test and document with less entanglement in other subsystems.	**Low:** The feature would be more difficult to isolate and test independently when embedded within core components.
Evolvability	**High:** Any future features can more easily be added safely in a separate subsystem.	**Low:** Future development would not be as straightforward due to higher coupling with core components.
Reusability	**Low:** In terms of short-term convenience, a new subsystem would be developed ground up, taking longer initially.	**High:** Integrating into already existing subsystems would easily make use of existing software, saving time earlier on.
Portability	**High:** GNUstep is already a cross-platform framework, hence the new subsystem must be designed with the same properties from the start.	**High:** Integrating into existing subsystems may inherit their already cross-platform properties.

Non-functional Requirement (NFR) Analysis

Based on the above analyses of our two suggested approaches, we conclude that Approach A, which introduces a new subsystem (libs-autosave), is the better choice for implementing version history autosave in GNUstep. It outperforms Approach B in nearly every non-functional requirement, especially in areas important to our primary stakeholder, the end user, as well as NFRs critical to long-term system health such as maintainability, testability, and evolvability.

For users, Approach A keeps things simple and intuitive. Version control stays neatly separated from the existing save/load interface, making it easier to understand and use without introducing any confusion. It also keeps performance consistent, since the autosave logic can be optimized on its own without slowing anything else down, as it is a standalone subsystem. Most importantly, users can rely on it to work smoothly and predictably without interfering with other parts of the system.

For developers, Approach A is a significantly more sustainable and maintainable solution. Since the feature lives in its own dedicated component, it reduces any unnecessary complication of core libraries such as libs-base and libs-gui. This modularity makes the feature easier to debug, test, and evolve over time. Future enhancements, for instance, branching history or more advanced recovery tools, can be implemented within the subsystem without entangling unrelated

parts of the system. Furthermore, the isolated design opens up reusability potential across other GNUstep tools or Objective-C applications needing similar functionality.

While Approach B might seem simpler to implement by integrating directly into existing components, this comes with trade-offs. It increases coupling, reduces clarity, and introduces long-term maintenance risks. It may offer short-term convenience, but over time, it would generate technical debt and make future improvements harder to manage. Thus, Approach A aligns with the architectural principles of GNUstep and provides a clean, robust, and evolvable architectural solution that aligns well with both user expectations and developer needs, making it the best overall choice for our proposal of the version history autosave feature in GNUstep.

Effects of the Enhancement

Current State of the System

GNUStep follows a layered, object-oriented architecture with subsystems like libs-base (core utilities), libs-gui (user interface), and apps-gorm (interface builder). The system currently lacks autosave or version history capabilities, requiring users to manually save files and rely on external tools for version control. The proposed enhancement introduces a new libs-autosave subsystem. The current system's simplicity is traded for improved usability and reliability, with minimal architectural disruption due to GNUstep's modular design.

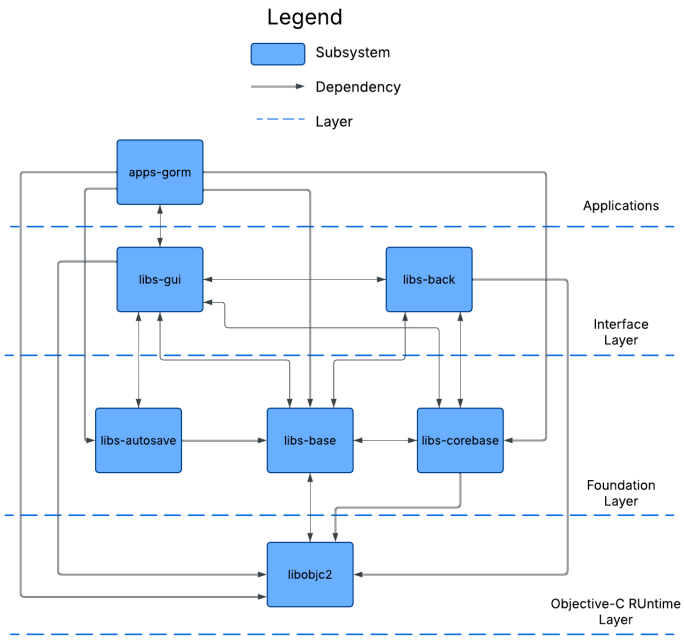

Figure 1: Conceptual Architecture of GNUstep with libs-autosave

Our enhanced GNUstep architecture introduces carefully designed dependencies to support the new autosave functionality while preserving the system's layered structure. We've established

three key relationships: (1) libs-gui → libs-autosave for UI updates, following the Dependency Inversion Principle by having presentation layer depend on version control abstractions; (2) apps-gorm → libs-autosave for edit notifications using the Observer pattern to maintain loose coupling; and (3) libs-autosave → libs-base to leverage existing file operations while keeping version management distinct from core utilities. We strategically positioned libs-autosave in the Foundation Layer because it provides fundamental services that build directly on core file operations while remaining independent of GUI logic, adhering to the Stable Dependencies Principle. This placement yields significant architectural benefits, including improved maintainability (through contained versioning logic), enhanced testability (via defined interfaces), and future extensibility (allowing additions like cloud sync without layer violations). These carefully considered modifications integrate version control as a transparent enhancement that respects GNUstep's existing layered architecture while adding valuable functionality through minimally invasive, well-abstracted dependencies.

Effect On the High and Low Level Architecture

The high-level architecture of GNUstep will be extended by adding a new libs-autosave subsystem between libs-base and libs-gui. This creates new dependencies from both libs-gui and apps-gorm to libs-autosave for version control functionality while maintaining GNUstep's existing layered structure.

At the low level, libs-autosave will introduce new components for managing automatic versioning and file recovery. The subsystem will interact with libs-base for file operations and libs-gui for UI updates, following GNUstep's established design patterns while adding the new autosave capability.

Interactions With Other Features and Subsystems

The new libs-autosave component will integrate seamlessly with several existing GNUstep features. It connects most with libs-base's file handling system to save and load different versions of files. When changes happen in apps-gorm (the interface builder), libs-autosave will automatically create backup copies using libs-base's file operations.

For the user interface side, libs-autosave will add new menu options in libs-gui's window system. These menus will let users view past versions and undo changes. The existing edit history in Gorm will be enhanced to show the autosave versions alongside regular edits.

The feature also ties into GNUstep's existing document system. When a programmer uses NSDocument to save files, libs-autosave will quietly make extra backup copies in the background. This happens without changing how the normal save feature works.

Tools like the debugger will get new options to examine autosave files when tracking down problems. The existing preferences system will gain new settings to control how often autosaves happen and how many versions to keep.

All these connections use GNUstep's standard ways for components to communicate, so the changes fit naturally with how the system already works. The autosave feature acts like an extra layer that supports existing functions rather than replacing them. This creates a responsive user experience while maintaining system stability, as all disk operations are handled through libs-base's robust file management capabilities.

Performance, Evolvability, Testability, and Maintainability

Performance

The proposed enhancement introduces periodic autosaving and version control, which may slightly increase memory usage and CPU activity during save operations. However, because autosaves are only triggered after edits, the impact on the overall system performance is expected to be minimal. Background-saving mechanisms can be optimized to avoid the user's workflow and UI responsiveness to avoid a negative impact on user performance.

Evolvability

Adding the auto-save and version history as a separate subsystem supports long-term evolvability. Future updates, such as supporting branching or merging, could be added without major changes to existing components. The clear separation of this new feature from the rest of the system allows it to evolve independently, giving it plenty of room for future growth and expansion.

Testability

Creating a new subsystem to handle this function makes it easier to isolate and test. Developers can write unit tests for the functions without needing to test the entire system. This modular approach works to allow the testing of different scenarios without disrupting the existing subsystems/

Maintainability

By maintaining this new feature using a modular approach, future bug fixes or optimizations can be made in a targeted way without affecting unrelated parts of the codebase. The use of well-defined interfaces between libs-autosave and other subsystems ensures that maintenance efforts remain localized and manageable, which aligns well with GNUstep's object-oriented design.

Impacted Directories and Files

The implementation will introduce a new subsystem directory, libs-autosave, containing the core version control functionality. Modifications will be made to libs-base, including updates to file handling utilities (such as NSFileManager) and data serialization components to support version tracking. In libs-gui, changes will be made to NSDocument as well as menu templates and window controllers to integrate version history display. These changes maintain compatibility

with GNUstep's existing architecture while implementing the new autosave functionality in a way that aligns with GNUstep's design principles.

Concurrency

There are no new concurrency requirements for the proposed autosave enhancement. Autosave operations are triggered synchronously immediately after user edits (eg, in apps-gorm), meaning no background threads or asynchronous scheduling. File writes and version metadata updates are handled by libs-base, which already employs thread-safe mechanisms for disk operations, such as file locking, ensuring data integrity during saves. User-initiated version restores (e.g., via libs-gui) also operate sequentially, avoiding race conditions.

Use Cases

Use Case 1: Auto-save after any changes

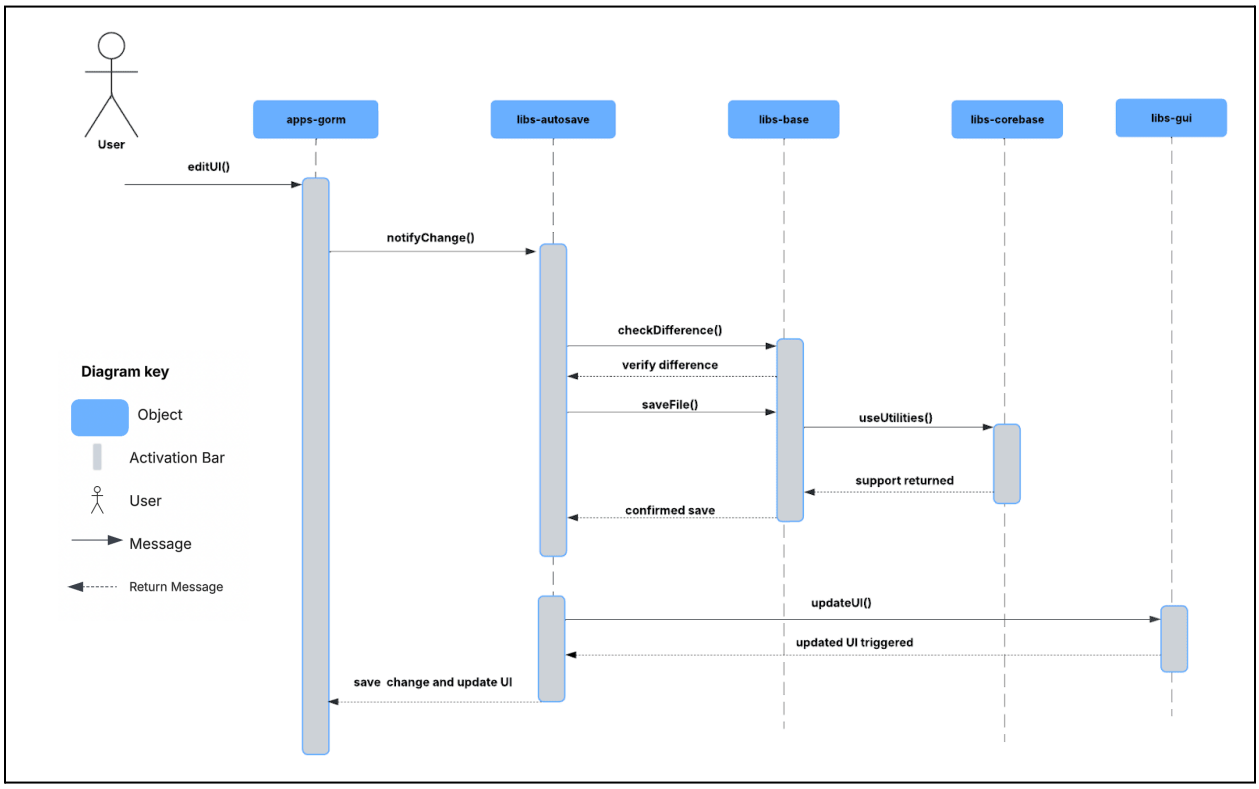

Figure 2: Sequence Diagram for use case of the auto-save function after changes

This sequence diagram above displays the use case in which a user makes a change to their edit using apps-gorm, and GNUstep automatically recognizes this and auto-saves the new change. When the user interacts with the UI, such as adjusting a layout or modifying a component, apps-gorm receives the event through editUI() and immediately notifies the new libs-autosave component via notifyChange(). libs-autosave checks for any differences by calling checkDifference() on libs-base, which compares the current and previous versions. If a change is detected, libs-autosave initiates the save process through saveFile(). libs-base then uses utility

support from libs-corebase (via useUtilities()) to carry out file-saving tasks, such as writing the change to disk and updating version metadata. Once the save is confirmed, libs-base returns this confirmation to libs-autosave. At this point, libs-autosave triggers updateUI() in libs-gui to reflect the new saved state in the interface. libs-gui responds once the UI is updated, and finally, libs-autosave sends a saveAndUpdateUI() confirmation back to apps-gorm. This ensures the user's changes are automatically saved and the interface UI is immediately updated to show the user confirmation of the auto-save.

Use Case 2: Restore previous version

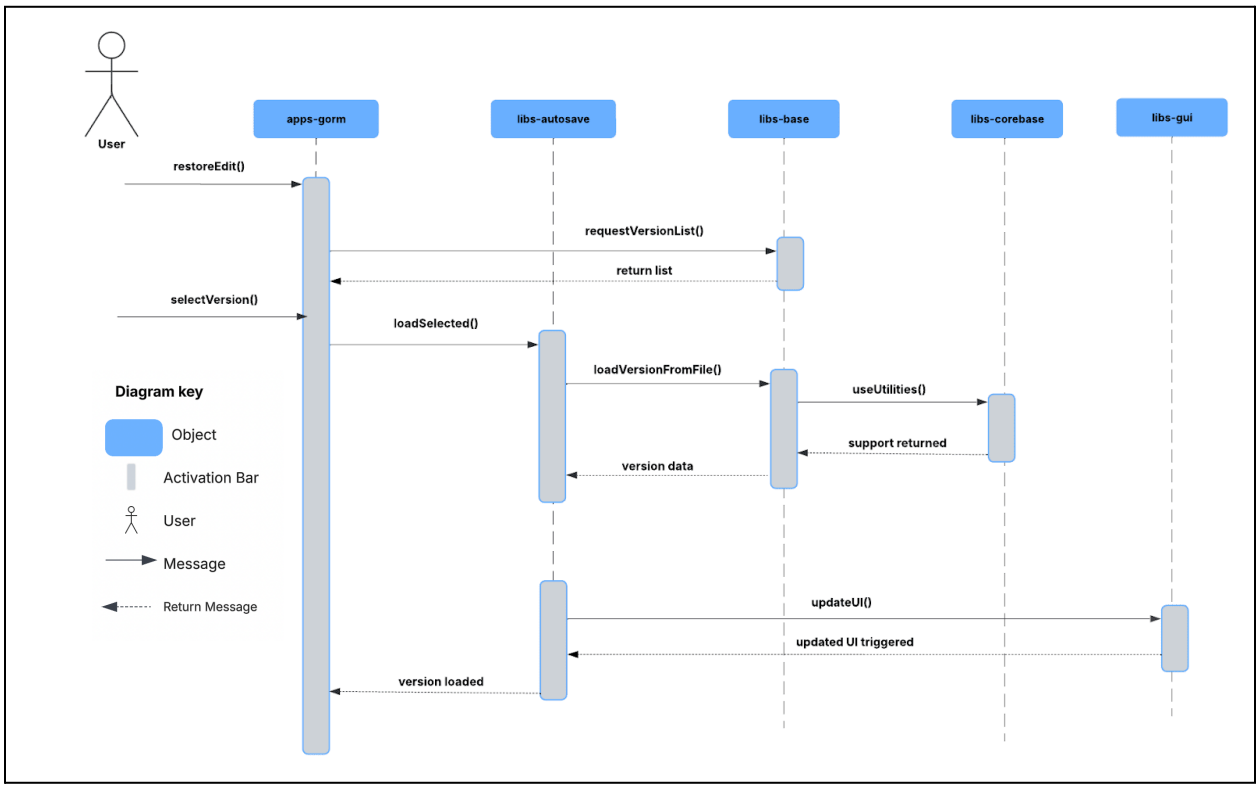

Figure 3: Sequence Diagram for use case of restoring to a previous version of edits

This sequence diagram above displays the use case in which a user chooses to restore a previous version of their UI edits using the version control feature integrated into GNUstep. The process begins when the user triggers restoreEdit() through the apps-gorm interface. This prompts apps-gorm to contact libs-autosave, which is responsible for handling version control. libs-autosave sends a requestVersionList() to libs-base, which retrieves available saved versions and returns the list. The user then selects a version, which triggers loadSelected(), prompting libs-autosave to handle the request by calling loadVersionFromFile() to libs-base. To complete the request, libs-base uses low-level support from libs-corebase by calling useUtilities() to process file access or formatting as needed. At this point, libs-autosave calls updateUI() on libs-gui to visually reflect the restored version. After the UI is updated, libs-gui notifies libs-autosave that the update has been triggered, and libs-autosave confirms the restored version back to apps-gorm.

Potential Risks

Although introducing a subsystem like libs-autosave has many benefits, there are potential risks involved that must be acknowledged.

Performance

Saving large files frequently might result in performance degradation. Despite the autosave operations getting triggered synchronously to avoid concurrency issues, small delays during this could affect the responsiveness of the user interface.

Maintainability

The libs-autosave subsystem interacts with multiple components like libs-base, libs-gui and apps-gorm. This results in changes in one component requiring corresponding updates in the other. This increases the maintenance time and without strict interface abstraction and documentation, coupling between these modules will become harder to manage over time.

Security

Storing multiple versions of files that contain sensitive user data introduces security concerns if it is not encrypted properly. Although GNUstep does not typically handle sensitive information, this should be acknowledged for broader use cases.

Storage Consumption

Saving multiple versions of a file can lead to excessive storage usage. This is especially the case for large or complex files. If not managed properly through potential version limits, this could consume a significant amount of disk space and degrade the performance of the system.

Testing

To ensure seamless integration of the autosave feature, a multi-layered testing strategy will validate interactions between the new libs-autosave subsystem and existing components. Integration testing will simulate real-world workflows, such as triggering edits in apps-gorm and verifying that libs-autosave correctly coordinates with libs-base to save versions and update the UI via libs-gui. Tools can be used to mock file operations to isolate dependencies, while metrics save latency and UI responsiveness will quantify performance impacts. Unit tests will focus on core functionality to ensure robustness at the component level. Cross-platform testing will confirm consistent behaviors across all systems, particularly for file paths and permissions. TO mitigate risks, tests will emulate edge cases such as crashes during autosave (validating recovering integrity) and storage limits (e.g., automatic pruning of old versions). This comprehensive approach ensures reliability while aligning with GNUstep’s modular design philosophy.

Lessons Learned and Limitations

One challenge we faced was the uncertainty around whether our proposed autosave feature was implemented correctly and efficiently. Since this feature doesn't currently exist in GNUstep and wasn't based on existing documentation or references, we had no concrete example to follow. This made it harder to validate our architectural decisions, and we often had to rely on internal reasoning and trial-and-error during design discussions.

One lesson we learned during our discussion process was the importance of teamwork. Whenever we got stuck or disagreed on how to approach a problem, whether it was placing a dependency or deciding where to integrate a new feature, talking it out together usually helped us find the right direction. Sometimes, one of us would catch something the others hadn't thought of, or suggest a simpler way to solve an issue. Having multiple perspectives made our final design stronger and helped us work through uncertainties more confidently.

Another lesson we learned was the importance of using the SAAM to address non-functional requirements in our enhancement. It helped us evaluate different architectural solutions based on performance, security, and scalability. By considering the perspectives of various stakeholders, we better understood their concerns and how non-functional requirements relate to them. This approach allowed us to make more informed decisions and create a solution that balances both functional and non-functional needs.

Conclusion

The new libs-autosave subsystem brings essential version control directly into GNUstep, automatically protecting work through periodic saves and easy version recovery. By building on GNUstep's existing architecture, using libs-base for file operations and libs-gui for interface integration, it delivers a seamless safety net against crashes or accidental changes without disrupting familiar workflows. Developers and designers gain peace of mind knowing their progress is preserved, while the intuitive version history eliminates tedious manual saving and external backup tools. This enhancement stands by GNUstep's modular design philosophy, offering both immediate productivity benefits and a foundation for scalable improvements.

Data Dictionary

Coupling: The degree of dependence between software components. Low coupling means less dependencies, making the system easier to maintain and modify.
Layered Architecture: A design pattern where the system is organized into layers, each with a specific role (e.g., UI, business logic, data access), and each layer only interacts with the one directly below it.
Non-Functional Requirement: A constraint that defines how a system should perform rather than what the system should do.
Object-oriented architecture: A design pattern that structures systems as collections of interacting objects, which bundle data and behavior together using classes and inheritance.

Subsystem: A smaller part of a larger system that performs a specific function and can often be developed, tested, and maintained independently.
Technical Debt: The extra work or risk that builds up when shortcuts are taken in design or implementation. It can make future changes harder, costlier, or more error-prone.
Version Control: A system that tracks and stores changes to files over time, allowing users to view, restore, or revert to previous versions of their work.

Naming Conventions

GORM: GNUstep Object Relationship Modeler (an interface builder application in GNUstep)
GUI: Graphical User Interface
NFR: Non-functional requirement
SAAM: Software Architecture Analysis Method

References

GNUstep concurrency. GitHub. (n.d.). https://github.com/search?q=org%3Agnustep%20concurrency&type=code

GNUstep libs-back. GitHub. (n.d.). https://github.com/gnustep/libs-back