CISC 322/326 - Winter 2025

GNUstep Proposal for Enhancement Report

April 4th, 2025

Infinite Loops - Authors

Ray Huezo 21tch5@queensu.ca

Anneth Sivakumar 21as221@queensu.ca

Vedsai Pandla 21vp23@queensu.ca

Rachel Narda 22rn3@queensu.ca

Jonathan Thompson 20jmt1@queensu.ca

Princess Viernes 20pejv@queensu.ca

Table of Contents

1.0 Abstract

This report proposes architectural enhancements for GNUstep , an open-source framework for developing cross-platform Objective-C applications. Building upon previous conceptual and concrete analyses of GNUstep, we propose the addition of a translation layer, *app-translate*, to enable macOS applications to operate seamlessly on Linux and Windows platforms. This enhancement, inspired by compatibility technologies such as WINE and Darling, would transform GNUstep from a development-only framework to a runtime environment capable of running macOS programs across multiple platforms.

Two implementation strategies are explored: one involving multiple subsystem dependencies to maximize integration and compatibility, and another minimalistic approach that prioritizes modularity and reduced architectural impact. Both designs are evaluated through the Software Architecture Analysis Method (SAAM), with an emphasis on the implications for key stakeholders such as developers, users, and investors in terms of non-functional criteria like performance, usability, and cross-compatibility.

A comparative analysis reveals that while the first approach offers great reliability and interoperability, it increases architectural complexity. The second option provides simpler integration but at the cost of limited fracture coverage. Ultimately, the proposed translation layer enhancement to GNUstep utility aligns with its modular object-oriented architecture, positioning it as a comprehensive tool for cross-platform application creation and deployment.

2.0 Introduction and Overview

The report represents the third and final stage of our architectural analysis of GNUstep, an open-source framework that allows for cross-form development of Objective-C applications [8]. Building on the conceptual architecture and the concrete, implementation-based analysis, this study proposes an architecture enhancement to extend GNUstep's capabilities. This enhancement aims to address an existing gap in functionality while remaining consistent with the system's modular and object-oriented architectural style.

The proposed enhancement is introducing a translation layer, referred to as *app-translate*, which enables GNUstep to execute completed macOS applications on Linux and Windows platforms. This addition, inspired by existing platforms such as WINE and Darling, allows users to run Windows and MacOS applications, on different operating systems and would provide developers with a powerful tool to ensure full runtime compatibility across platforms [3][7]. By combining the ideas behind WINE and Darling, *apps-translate* aims to bridge the gap between development and deployment, transforming GNUstep into a cross-platform runtime framework as well as a cross-platform development environment.

The remainder of this paper details the architectural modifications required to implement the proposed feature. Two implementation strategies are presented and evaluated: one that introduces multiple subsystem dependencies to facilitate comprehensive integration and a second that adopts a more lightweight design to minimize architectural impact. For each, we examine the architectural changes, interface consequences, and trade-offs.

Additionally, a Software Architecture Analysis Method (SAAM) evaluation is conducted to determine how well each implementation approach meets key non-functional requirements from the perspective of major stakeholders. This comparison analysis informs our proposal for the most effective design approach.

Ultimately, this proposal seeks to improve GNUstep's utility, portability, and relevance in the modern software landscape by enabling seamless application execution across macOS, Linux, and Windows platforms. In doing so, this strengthens GNUstep's role as a comprehensive solution for cross-platform Objective-C development and deployment.

3.0 Enhancement Proposal

We are proposing the addition of a translation layer, which would allow users to run their MacOS applications on Linux and Windows. This idea draws from two different projects, WINE and Darling. WINE is a compatibility layer that allows users to run Windows applications on various POSIX-compliant operating systems [7], and Darling is a translation layer that enables users to run macOS software on Linux [3]. Combining these ideas, we propose that GNUstep implements a translation layer, similar to WINE and Darling, so that users can make their completed macOS applications compatible with Linux and Windows.

4.0 Implementation

4.1 Implementation 1

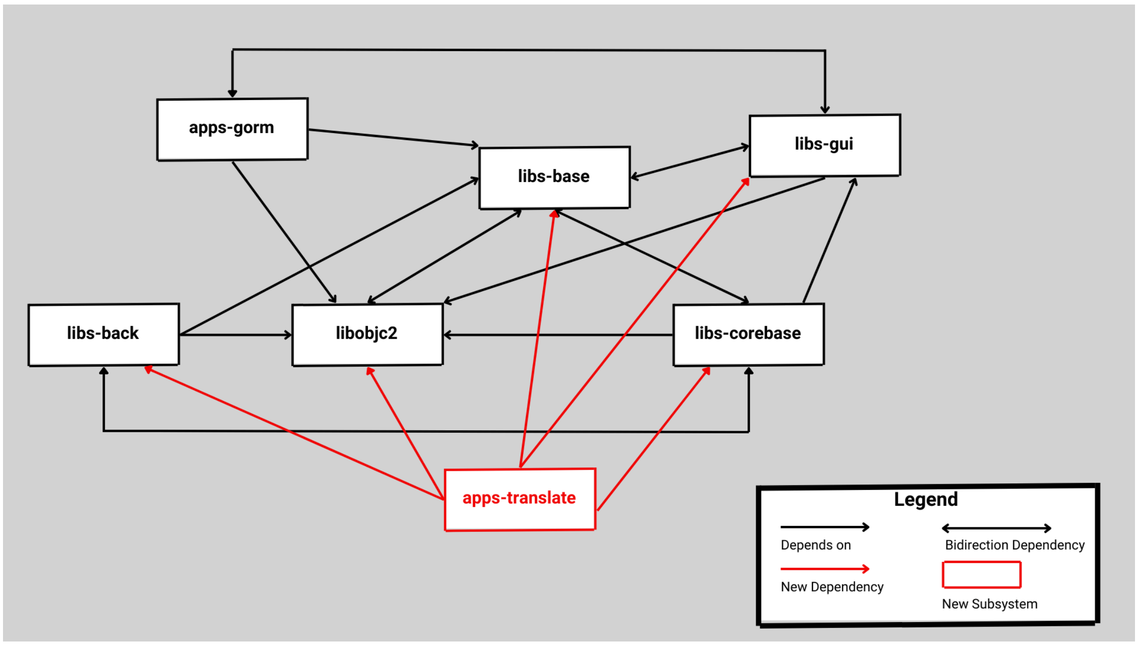

Figure 1: *Implementation 1*

The first approach to implement apps-translate can be seen in Figure 1. This approach would add five new dependencies to the high-level architecture. Apps-translate would draw from the Objective-C resources and utilize the APIs of the OpenStep Foundation and Apple's Core Foundation implemented in libs-base and libs-corebase respectively [5] [6]. As well, it would use libs-back to aid in the translation of GUIs [4]. Apps-translate would also rely on libobjc2 as an Objective-C runtime library to aid in the translation. Lastly, it would use libs-gui for UI management [2].

4.2 Implementation 2

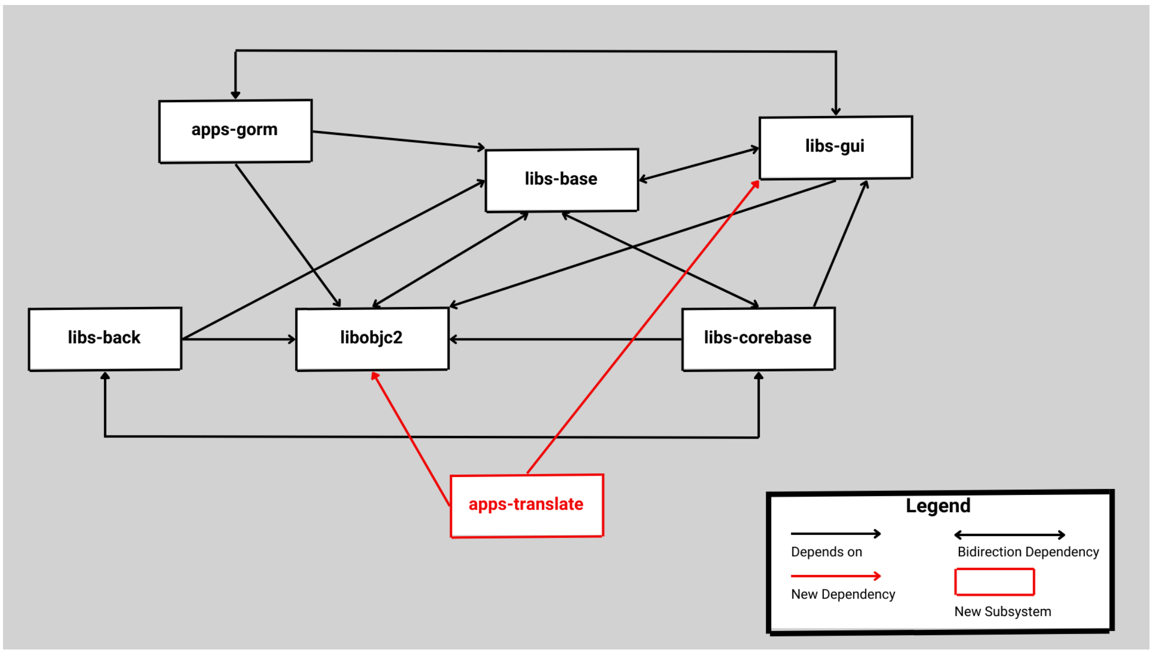

Figure 2: *Implementation 2*

The second approach to implement apps-translate can be seen in Figure 2. This approach only adds two new dependencies to the high-level architecture. In this implementation, apps-translate would continue to depend on libs-gui, but would only rely on libobjc2 for aid in translation. This means apps-translation would need to include its own implementation of the APIs of the OpenStep Foundation and Apple's Core Foundation, as well as its own library for Objective-C resources.

5.0 SEI SAAM Analysis

The Software Engineering Institute (SEI) developed the Software Architecture Analysis Method (SAAM) to evaluate a system's architecture [1]. In this report, SAAM is applied to GNUstep's enhanced system to identify the relations between the stakeholders and the non-functional requirements (NFRs), and the enhancement's effect on the stakeholders.

5.1 Stakeholders

The enhancement would involve three major stakeholders: developers, investors, and users. Developers have an important role to design, build, test, and maintain software systems and the enhancement would involve several types of developers. For instance, the GNUstep developers could hire and/or work with Darling developers to combine their knowledge and expertise to integrate this transition layer component. The developers would also be supported by investors that provide funding for the project and/or mentorship. Users also can provide feedback during the development phase, but are impacted more after the release date since the enhancement is created primarily for them.

5.2 NFRs

For the enhancement, each stakeholder would expect specific NFRs from the completed project. The users would expect reliability, performance, and usability to be able to develop their applications easily that are consistent and efficient. The investors would expect cross-compatibility, performance, and reliability to ensure the enhancement would provide a valuable system product to bring in more users. The developers would need to take the NFRs from the other stakeholders into account to ensure they can work towards an enhancement that can benefit them, but also allow any modifications to the NFRs that would be needed to develop the added component. The developers would focus on achieving cross-compatibility, performance, and interoperability to preserve the GNUstep design while integrating Darling features and producing a high-quality system.

5.3 Effect of Enhancement

The effects from the enhancement would affect each stakeholder differently which results in specific NFRs of each stakeholder. The users would benefit the most with the ability to apply their macOS applications to more OS while putting in little effort into the development. If the amount of users increases from this enhancement, the investors benefit from more profitable opportunities as the system advances. However, the users and investors would be negatively impacted from any low-quality performance and/or unexpected features/issues that would depart the users from using GNUstep. The enhancement would also impact all the stakeholders' time depending on several factors (i.e. resource allocation, expected completion date, etc.).

The developers are greatly impacted by the amount of work they have to put into the project. Although they are working towards adding a new advancement for GNUstep, they face more risks than the other stakeholders. Depending on the current financials during the development phase, the amount of developers depends on the available resources allocated towards their salaries. It is not guaranteed that there would be enough investors to support the project. There is also uncertainty that the original developer team would be hired again, and would most likely need a balanced amount of GNUstep and Darling developers. After the project is released, the developer team must also maintain the system and act on any bug fixes or criticism from the users. However, if there are enough resources, the enhancement would be a great opportunity for developers to earn money while working towards the adding new GNUstep capabilities.

6.0 Implementation Comparison

In the first enhancement implementation, the greater number of new dependencies would positively impact the users, investors, and developers. More dependencies would allow more API coverage from *libs-base* and *libs-back* and control to ensure reliability and cross-compatibility within the system for the users and investors. The dependency on *libs-gui* and *libs-core* would give more GUI tools for usability and lower the translation overhead for performance among the users and investors. For the developers, the added dependencies allow direct access to each component to ensure cross-compatibility with greater control on the overall system to optimize performance. This implementation would also target interoperability with GNUstep and Darling tightly integrated for seamless interactions between the components.

6.1 Architectural Comparison

Implementation 1 adopts a layered architectural style, deeply integrating with GNUstep's existing subsystems. It introduces five dependencies: libs-base (OpenStep Foundation APIs), libs-corebase (Apple Core Foundation APIs), libs-back (GUI rendering), libobjc2 (Objective-C runtime), and libs-gui (UI management). This approach maximizes compatibility with macOS APIs by leveraging GNUstep's mature libraries, ensuring broad coverage of foundational and GUI-related features. However, the reliance on multiple subsystems increases architectural complexity, as updates to any dependency could require coordinated maintenance. Performance may also face moderate overhead due to interactions between layered components.

Implementation 2 follows a minimalist, microkernel-like design, depending only on libobjc2 and libs-gui. To compensate for the lack of dependencies, it reimplements critical APIs like OpenStep and Core Foundation within apps-translate. This reduces architectural complexity and avoids tight coupling with GNUstep's subsystems, enabling faster iteration and lighter resource usage. However, limited macOS API coverage risks compatibility gaps, particularly for applications relying on advanced macOS-specific features. While performance benefits from reduced overhead, maintaining custom code alongside GNUstep's upstream updates introduces dual maintenance burdens.

6.2 SAAM Analysis Comparison

The SAAM evaluation highlights stakeholder-centric trade-offs:

- **Reliability**:
  Implementation 1's dependency on proven subsystems like libs-base ensures stability for users and investors, minimizing untested edge cases. Implementation 2's custom code introduces reliability risks, especially for macOS-specific APIs.

- **Performance**:
  While Implementation 2 reduces overhead through minimalism, Implementation 1's use of libs-gui and libs-back optimizes GUI rendering efficiency, indirectly lowering translation latency for users.
- **Cross-Compatibility**:
  Implementation 1's broad API coverage satisfies investors by aligning with macOS standards. Implementation 2's reimplemented logic struggles to match this parity, limiting its utility.
- **Interoperability**:
  Implementation 1's shared dependencies with Darling streamline macOS integration, whereas Implementation 2's isolated design risks fragmentation.
- **Maintainability**:
  Implementation 1 aligns with GNUstep's roadmap, easing long-term updates despite dependency management complexity. Implementation 2's custom code demands parallel maintenance effort.

6.3 Concluding Remarks

Implementation 1 emerges as the superior choice, balancing stakeholder needs with technical robustness. Its expanded dependencies empower users with reliable cross-compatibility, investors with macOS-aligned performance, and developers with precise control over system behavior. While architectural complexity increases, the trade-off ensures a future-proof foundation for GNUstep as a runtime environment. Implementation 2 suits niche scenarios prioritizing minimalism over macOS feature parity but falters in compatibility and maintainability.

7.0 Potential Risks and Limitation

The proposed enhancement introduces valuable cross-platform capabilities to GNUstep, but it also has some risks and limitations. Despite the potential advantages of leveraging compatibility layers like Wine and Darling to extend GNUstep's cross-platform capabilities, this approach introduces several important risks and limitations. First, compatibility and stability remain key concerns. These translation layers, while powerful, do not perfectly replicate the behavior of native operating systems. Many macOS applications may behave unpredictably or fail entirely due to incomplete API coverage or unsupported system calls, particularly as macOS is constantly evolving. This risk is especially high when supporting newer or more complex applications.

Additionally, security is another critical concern. By introducing a layer that interprets or bridges native system calls, the system's attack surface is widened, potentially exposing users to vulnerabilities that are specific to the compatibility layer itself [9]. Malicious or poorly designed applications might exploit these translation systems in ways that would not be possible on a native system.

Also, performance overhead also presents a significant limitation. The use of a compatibility layer inherently introduces latency and computational overhead, especially in applications that require intensive system resources [9]. This may be acceptable for simpler productivity applications but can be a major drawback for graphics-heavy or real-time software. In addition, while Wine has a mature and active developer community, Darling is still relatively new and lacking the strong support architecture that developers rely on for troubleshooting and enhancement.

Furthermore, legal and licensing risks are also potential risks of the proposed architecture. Copying Apple's proprietary APIs, even in open-source tools like Darling, can venture into legal areas, depending on how these APIs are used and distributed. Any deployment of such solutions in a commercial or public environment would need to be carefully vetted to ensure compliance with software licensing laws. Finally, there are concerns related to system integration. Applications running via Wine or Darling may fail to fully integrate with the original environment of the host operating system, leading to inconsistent user experiences or limited functionality compared to other applications.

8.0 Use Case: Run a Cross-Platform macOS App

The use case in *Figure 3.0* involves a *User*, the primary actor, to create a new macOS application using GNUstep and Darling that is cross-compatible with Windows and Linux. The process begins with the user running the app on Darling which triggers a request sent to *apps-translate*. *apps-translate* initializes the runtime using *libobjc2* to load the Objective-C classes. *apps-translate* loads the foundational and additional logic using *libs-base* and *libs-corebase* to be able to load the gorm UI.

apps-translate loads the UI using *apps-gorm* that consists of instances of UI objects created from *libobjc2*. Then, *apps-gorm* creates the hierarchy of interface objects and initializes actions using *libs-gui* which also calls *libs-back* to create the window. After the gorm UI loads, *apps-translate* initializes the event loop with *libs-gui* which is rendered with *libs-back*. When the event loop is ready, *apps-translate* is ready to translate for Darling and Darling displays the app for the user.

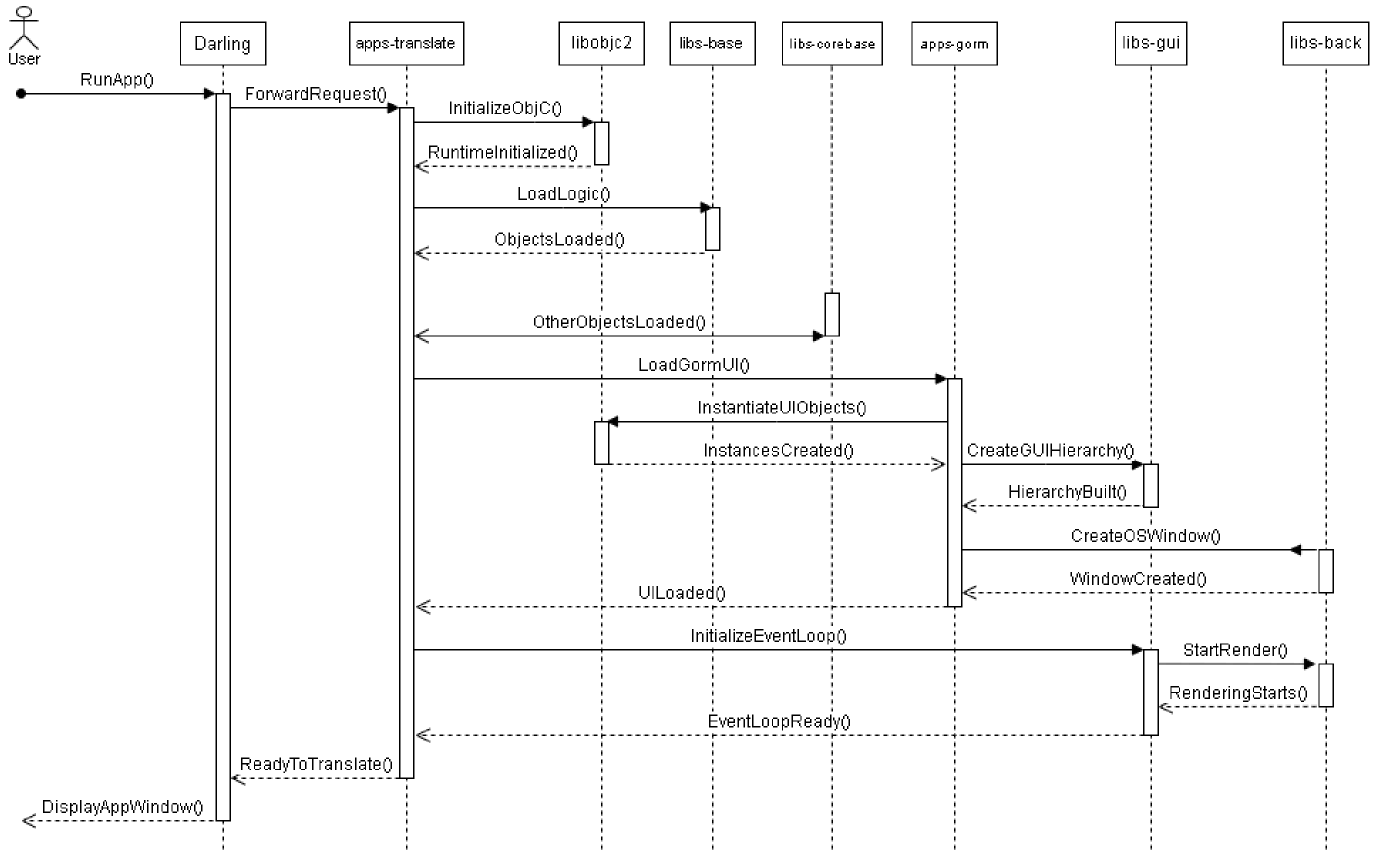

Figure 3.0 - Use Case: Run a Cross-Platform macOS App

9.0 Lessons Learned

Overall, working on this enhancement project for GNUstep has provided many insights into the architectural and practical challenges of cross-platform system design. One of the most important lessons learned is that cross-platform compatibility is far more complex than initially assumed. While tools like Wine and Darling offer a starting point, replicating system behaviors across fundamentally different operating systems requires extensive technical knowledge and continuous testing.

Moreover, the importance of community support and documentation in selecting architectural tools [10]. Wine's mature system, extensive documentation, and active user base contrast with the limited resources available for Darling. This highlighted how community maturity can significantly affect the feasibility and efficiency of implementing a system enhancement.

Another key takeaway was the value of modular architectural design. Both Wine and Darling rely on well-structured subsystems to isolate concerns like syscall translation, library emulation, and application execution [7]. This modularity not only helps manage complexity but also enables more targeted debugging and maintenance. It reinforced the importance of architectural clarity when designing scalable enhancements.

The necessity of thorough cross-environment testing is another key takeaway. Differences in system behavior, user interface conventions, and underlying libraries across operating systems can easily introduce subtle bugs that are difficult to trace without extensive testing. Ensuring strong functionality requires deliberate test planning across multiple platforms and use cases.

Lastly, this project emphasized the importance of considering legal and ethical dimensions in software architecture. Creating compatibility layers for proprietary systems requires a thorough grasp of license agreements and legal limitations [10]. A solution must not only be technically viable, but also ethically and legally sound. This experience has increased our understanding of the larger environment in which software systems work, as well as the responsibilities associated with software development.

10.0 Conclusion

This report introduced a proposed enhancement to GNUstep: a translation layer, apps-translate, that enables macOS applications to run on Linux and Windows platforms. Building on our prior conceptual and concrete architectural analyses, this enhancement seeks to turn GNUstep from a development-only framework into a fully cross-platform runtime environment.

We proposed and evaluated two architectural implementations of this translation layer. The first implementation is designed to support a wide range of macOS API compatibility by integrating deeply with multiple GNUstep subsystems. The second implementation focused on modularity and simplicity by limiting dependencies and re-implementing essential macOS APIs internally.

Each approach was analyzed using the Software Architecture Analysis Method (SAAM) to assess trade-offs for key stakeholders, including users, developers, and investors. While the second implementation offered reduced architectural complexity and greater independence, the first implementation aligned more closely with stakeholder priorities (cross-compatibility, performance, and interoperability).

Based on this analysis, Implementation 1 is recommended as the more effective solution. Its layered integration with well-established GNUstep libraries offers a strong foundation for application compatibility, while remaining consistent with GNUstep's object-oriented architecture. However, it introduces added dependency management and potential maintenance challenges, the long-term benefits of API coverage and strong system interaction outweigh these concerns.

Ultimately, the apps-translate enhancement positions GNUstep as a comprehensive Object-C platform, supporting both cross-platform development and deployment. As modern software systems continue to demand cross-environment support, this proposal advances GNUstep's relevance in today's landscape.

11.0 Data Dictionary

apps-gorm - Contains Gorm, a visual interface builder for designing GUI layouts via drag-and-drop.

libobjc2 - A subsystem that implements the Objective-C runtime. Used to handle key tasks such as dynamic method dispatch and memory management.

libs-back – A rendering system library responsible for abstracting different backend implementations (e.g., X11, Windows GDI).

libs-base – A core library providing fundamental Objective-C classes for collections, threading, and system interaction.

libs-corebase – A compatibility layer library ensuring integration with Apple's Core Foundation framework.

libs-gui – A library providing graphical user interface components such as windows, buttons, and menus.

apps-translate - A subsystem modeled after Wine and Darling to run macOS apps on Linux/Windows.

Translation layer - A system component that intercepts macOS application calls and remaps them to equivalent Linux or Windows os.

Wine - A compatibility layer that allows windows applications to run on POSIX systems.

Darling - A compatibility layer that enables macOS applications to run on Linux by translating Darwin/macOS APIs into Linux equivalents.

Cross-Compatibility - The ability of software to operate across multiple OS.

Interoperability - The capacity of different systems or subsystems to work together .

SAAM (Software Architecture Analysis Method) - A method used to evaluate software architecture.

12.0 Naming Conventions

apps - Used for application-level modules

libs - Used for core library subsystems

NS - Stands for NextStep, a prefix used for class names and operations inherited from OpenStep (e.g NSThread and NSResponder).

NS* - Prefix for Foundation classes

libs-* - prefix for GNUstep libraries

13.0 References

[1] Software architecture analysis method. (2024, March 26). In Wikipedia. https://en.wikipedia.org/wiki/Software_architecture_analysis_method

[2] *AppKit*. AppKit - GNUstepWiki. (n.d.). https://mediawiki.gnustep.org/index.php/AppKit

[3] Darling. (n.d.). https://www.darlinghq.org/

[4] Gnustep. (n.d.-a). *Libs-back/announce at master · GNUSTEP/Libs-back*. GitHub. https://github.com/gnustep/libs-back/blob/master/ANNOUNCE

[5] Gnustep. (n.d.-b). *Libs-base/documentation/manual/baselibrary.texi at master · GNUSTEP/Libs-Base*. GitHub. https://github.com/gnustep/libs-base/blob/master/Documentation/manual/BaseLibrary.texi

[6] Gnustep. (n.d.-c). *Libs-corebase/documentation/mainpage.dox at master · GNUSTEP/Libs-corebase*. GitHub. https://github.com/gnustep/libs-corebase/blob/master/Documentation/MainPage.dox

[7] *What is wine?*. WineHQ. (n.d.). https://www.winehq.org/

[8] Botto, F., Frith-Macdonald, R., Pero, N., & Robert, A. (2001-2004). *Objective-C language and GNUstep base library programming manual*. Free Software Foundation. Retrieved from http://andrewd.ces.clemson.edu/courses/cpsc102/notes/GNUStep-manual.pdf

[9] Majerowicz, Lucas. "Software Architecture Pitfalls: High Risk-Low Reward Designs." Medium, August 25, 2021. https://medium.com/@lucas.majerowicz/software-architecture-pitfalls-high-risk-low-reward-designs-2d12b890ab2a.

[10] Nguyen, Taylor. "10 Common Risks in Software Development: How to Minimize?" Rikkeisoft, May 19, 2023. https://rikkeisoft.com/blog/software-development-risks/.