GNUStep Concrete Architecture

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 2: Concrete Architecture Report

March 14th, 2025

Table of Contents

Abstract

This report provides a detailed description of the GNUstep architecture, with a particular focus on how the concrete layout of its pieces is derived through a methodical process of dependency analysis using the Understand tool. The system was initially conceived as an object-oriented and layered system with certain key subsystems such as libs-base, libs-corebase, libs-gui, libs-back, and Gorm. However, the concrete analysis revealed an additional important component, libobjc2. Libobjc2, the Objective-C runtime library, facilitates object-oriented features like dynamic method invocation and message passing between all of the major subsystems, making it a root dependency for the entire system. Some bidirectional dependencies were also discovered in the analysis, representing a more dynamic and interdependent system than initially assumed.

Aside from the further architectural review, we examined the subsystem libs-back's internal dependencies and revealed that its design is both interactive and modular. Rendering and platform-dependent communications are addressed by libs-back and its dependents on libs-base for low-level utility and libs-gui for the graphical user interface. With reflexion analysis, we gained a more precise architectural model representing the condition of the system structure, correcting initial assumptions on relationships and dependencies. The outcome confirms that GNUstep's architecture is not as simple and linear as the conceptual model suggested, with libobjc2 being the central component and bidirectional dependencies playing a supreme role in the system's functionality. This increased insight is a reflection of a better comprehension of GNUstep's architecture and its relations.

Introduction

The purpose of this report is to analyze the concrete architecture of GNUstep. In our previous report, we examined conceptual architecture, focusing on the high-level design and structure of the system based on our understanding of the system at that time. In this report, we will focus on analyzing the actual implemented structure of GNUstep by using the Understand tool to extract and organize system dependencies.

We will first begin by describing the step-by-step method we followed to create our concrete architecture, which consisted of conducting independent research and then analyzing dependencies using Understand. We will then discuss the conceptual and concrete architecture. The conceptual architecture which we had originally analyzed in our first presented a high-level, layered, and object-oriented design where each subsystem had a clear and distinct role. In this assignment, we discuss the minor changes we made to our conceptual model, including the addition of another dependency as well as the separation of our components into layers. Through further analysis, we discover that the concrete architecture of GNUstep is more complex and interconnected. Particularly due to the many two-way dependencies between the components as well as an additional component called libobjc2, which we did not recognize during our first report. Following this, we will perform two reflection analyses: one to compare our high-level conceptual architecture with the concrete architecture, identify discrepancies with subsystems and their dependencies as well as provide explanations for these differences. The other reflection analysis will be on the 2nd-level subsystems and inner architecture of libs-back. In this analysis, we compare both the conceptual and concrete architecture views and highlight how the conceptual and concrete architectures differ and what these differences reveal about how libs-back truly functions in GNUstep.

Finally, we will conclude our report by discussing the use cases that illustrate how GNUstep operates, such as running the GNUstep software and saving a file using an application built on GNU-step. We will also be exploring aspects of concurrency, reflecting on the limitations and lessons learned and ending with a conclusion summarizing our insights.

Derivation Process

To derive the concrete architecture of GNUstep, we followed a structured approach; each team member independently worked on examining the system's structure using the Understand tool before consolidating our findings and refining them as a group. After our analysis using the Understand software, we were able to identify an additional key subsystem, libobjc2. This was understood by visualizing dependencies by importing source files, classifying subsystems, and expressing relationships through diagrams.

We were also able to identify discrepancies in how dependencies were originally represented as we refined our conceptual architecture. When we analyzed the system using Understand, we realized that our conceptual model oversimplified the relationships between the subsystems. By using Understand, we identified several bidirectional dependencies. Additionally, libobjc2 was found to be a central dependency rather than a secondary component, requiring adjustments to our diagram. Another key observation was that Gorm has dependencies beyond just GUI components, which let us know that while Gorm is primarily a UI design tool, it also interacts with the Objective-C runtime and core libraries, which are necessary for its operation.

After reviewing these relationships, we were able to create an updated dependency diagram that accurately represents how subsystems interact in GNUstep. This was done by adding the newly discovered dependencies between subsystems and adding libobjc2 as a main subsystem. Despite all of the adjustments we had to make, we still stand by our decision that GNUstep follows both a layered and object-oriented architectural style. Overall, the system still follows a layered organization, with higher-level libraries building on lower-level ones. Additionally, GNUstep's consistent use of Objective-C classes and modular components confirms that object-oriented design remains a core part of its architecture.

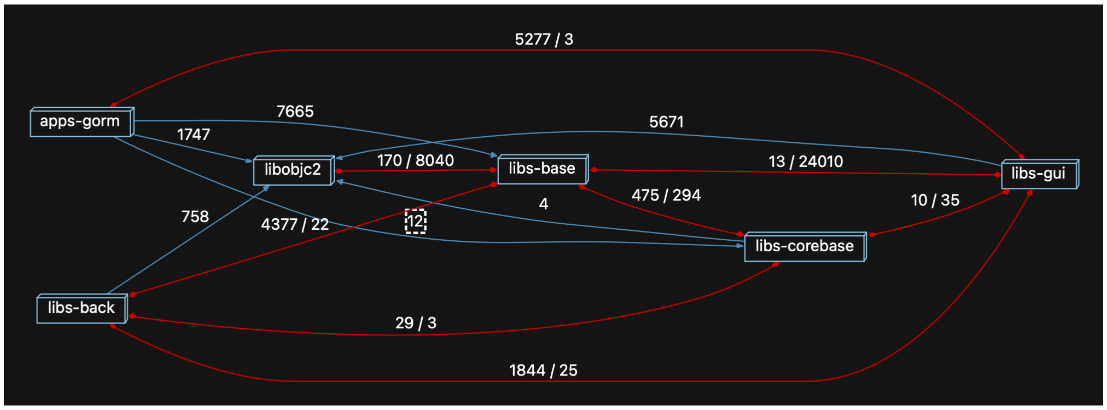

Figure 1: Dependency diagram generated by Understand

Conceptual Architecture

Figure 2 shows the conceptual architecture of GNUstep. This structure reflects a layered and object-oriented design. For this report, we made two small updates to clarify our model. First, we added a direct dependency from Gorm to libs-base because Gorm relies on essential system services provided by libs-base to function properly. For example, when Gorm saves or loads interface files, it needs file I/O, memory management, and basic data structures, all of which are part of libs-base. Second, we separated the components into layers, such as Application, Interface, and Foundation, to better show how these subsystems are organized.

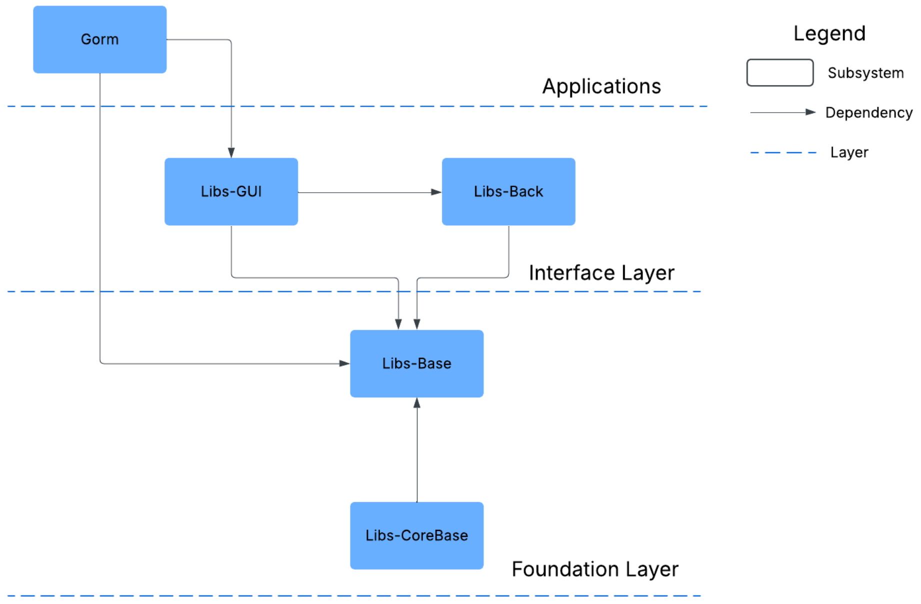

Figure 2: Proposed Conceptual Architecture of GNUstep

Concrete Architecture

GNUstep follows a layered architecture with object-oriented design. The system is divided into distinct layers, where lower layers provide fundamental system services and higher layers use them as a foundation to implement functionality. Modularity is guaranteed, and the system becomes more maintainable and extendable by dividing them like this. Additionally, GNUstep is built around object-oriented principles, primarily with the use of Objective-C. These are used to provide encapsulation, inheritance, and polymorphism, which enable reusable and adaptable components.

The proposed concrete architecture of GNUstep, as shown in Figure 3, reflects how the system is actually structured based on our analysis using Understand. While our conceptual architecture focused on how we expected the subsystems to interact based on various online sources that were mentioned in our first report, this diagram shows the real dependencies between components.

Libobjc2 is the modern Objective-C runtime that makes it possible for GNUstep to support newer features of the language. Although we did not include libobjc2 in our conceptual architecture, we discovered that, since every part of GNUstep is built on Objective-C, libobjc2 is essential to how the whole system works. This allows it to fully support object-oriented programming (FreshPorts, n.d.). Additionally, libobjc2 and libs-base share a two-way dependency.

Libs-base provides the core system services for GNUstep, including file input/output, memory management, threading, and foundational data structures. While we initially expected it to serve as an independent foundation for other components, we discovered that libs-base actually has two-way dependencies with libobjc2, libs-corebase, libs-back, and libs-gui.

Libs-corebase offers low-level utilities and system functions that extend the capabilities of libs-base. In our conceptual model, we only identified a one-way dependency from libs-corebase to libs-base, but we discovered that this relationship is actually two-way, meaning that libs-base also depends on libs-corebase. We also found two-way dependencies between libs-back and libs-gui, which we had not proposed in our conceptual architecture.

Libs-gui provides GNUstep's graphical user interface components, including windows, menus, and buttons. In our conceptual model, we only identified a one-way dependency from libs-GUI to libs-base and libs-back, but we discovered that these relationships are actually two-way dependencies. We also found two-way dependencies between libs-corebase and apps-gorm.

Libs-back serves as GNUstep's backend, handling rendering and interfacing with system graphics APIs. We found that the one-way dependency from libs-back to libs-base is actually two-way, meaning that libs-base also depends on libs-back. We also discovered that libs-back has two-way dependencies with libs-corebase and libs-GUI.

Gorm is GNUstep's interface builder for creating application UIs. As expected from our proposed conceptual architecture, Gorm depends on libs-base and libs-corebase to access core system services and low-level utilities. However, we also discovered a two-way dependency between Gorm and libs-gui.

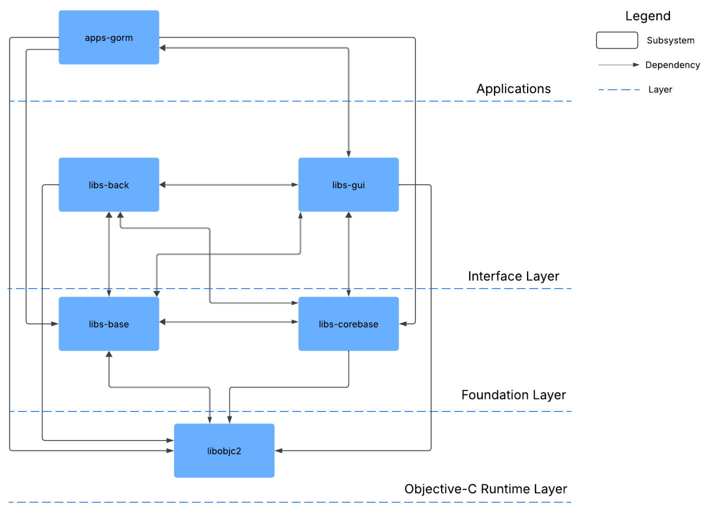

Figure 3: Proposed Concrete Architecture of GNUstep

Reflexion Analysis of GNUstep

Figure 4 shows our reflexion model. The initial model in our conceptual report was designed with a layered and object-oriented architecture, where subsystems such as libs-base, libs-gui, and libs-back were expected to follow a strict hierarchy with clear one-way dependencies. However, the concrete analysis identified several two-way dependencies, additional interactions between components, and a new subsystem, libobjc2, which serves as a foundational runtime component for the entire framework. It is important to note that although libobjc2 was not initially included in our conceptual architecture, using Understand has revealed it as a core component of GNUstep. Therefore, while all subsystems directly depend on libobjc2, we do not consider these dependencies as divergences but rather as naturally expected connections within our concrete architecture.

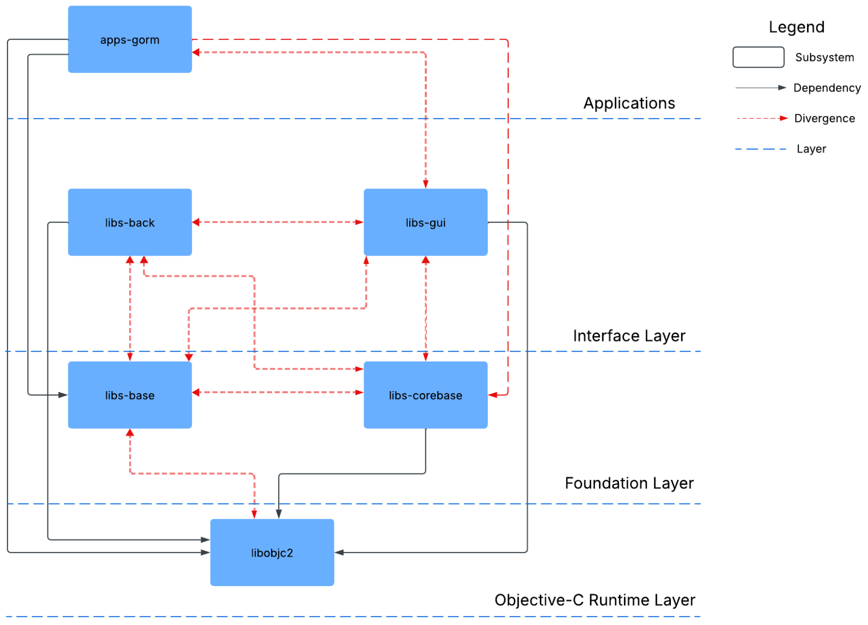

Figure 4: Proposed Reflexion Model of GNUstep

libobjc2

As previously mentioned, libobjc2 was not explicitly included in the conceptual model. Initially, it was assumed to be an underlying runtime rather than a separate component of GNUstep's layered architecture. However, the concrete architecture shows that every major subsystem in GNUstep directly depends on libobjc2, making it a foundational component that must be recognized. Without libobjc2, key object-oriented features would not function, which explains its direct interactions with libs-base, libs-gui, and libs-corebase.

Additionally, there is a two-way dependency between libobjc2 and libs-base, which was not accounted for in the original model. While it was initially assumed that libobjc2 would provide services only to higher-level libraries, the real implementation shows that lib-base also provides system-level utilities that libobjc2 relies on. This dependency suggests that libs-base includes fundamental functionality that even the Objective-C runtime requires, potentially for memory management or thread synchronization.

libs-base

We positioned libs-base as the core system library in our conceptual architecture, responsible for providing services such as memory management, file I/O, and concurrency control to all other GNUstep components. This expectation was generally validated in the concrete analysis, as libs-base is a major dependency for all other libraries. However, it was initially assumed that libs-base would function as an independent foundational layer, yet the concrete implementation reveals several two-way dependencies that indicate more complex interactions rather than the one dependency in our conceptual architecture, such as with libobjc2, libs-back, libs-corebase, and libs-gui, and a direct dependency from Gorm. For instance, the NSInvocation class allows for the packaging of a message and its arguments, which can be invoked at a later time (GNUstep libs-base Repository, n.d.). To function correctly, NSInvocation relies on runtime functions such as objc_msgSend and method_getImplementation from libobjc2. Conversely, libobjc2 uses utility functions from libs-base for tasks like memory management and string manipulation.

libs-corebase

Our original conceptual model positioned libs-corebase as an optional extension to libs-base, providing additional low-level utilities rather than core system services. In theory, we expected a one-way dependency from libs-corebase to libs-base, reinforcing the hierarchy where corebase functions as a helper library rather than a more important system component. However, our concrete analysis revealed that libs-base also depends on libs-corebase, forming another two-way dependency that was not accounted for in our conceptual architecture. This suggests that certain low-level optimizations or additional utility functions provided by libs-corebase are essential to its functioning. One possible explanation is that corebase may provide some sort of system abstraction or performance optimizations that are used by other parts of the system. Thus, rather than simply enhancing functionality, corebase may have become a required component over time, resulting in a two-way relationship with libs-base.

libs-gui

libs-gui was expected to function as a graphical user interface layer, relying on libs-base for core functionality and libs-back for rendering. Though these dependencies were in fact true in the implementation, the concrete architecture revealed additional unexpected dependencies. For one, libs-gui had a two-way dependency with both Gorm and libs-back. There was another two-way dependency between libs-gui and libs-corebase, meaning certain GUI operations may require extended system functions, such as custom memory management or font handling, which are provided by corebase. Then, the two-way relationship between libs-gui and libs-back indicates that GUI rendering is not a simple top-down process but involves dynamic communication between the interface layer and the rendering engine.

libs-back

Our conceptual model correctly identified libs-back as the backend rendering system, responsible for translating GUI commands into low-level graphics operations. However, here again, we originally assumed a one-way dependency, where libs-back would rely on libs-base for system functionality but would not provide services back to it. In reality, the concrete architecture demonstrates multiple two-way dependencies—with libs-base, lib-corebase, and libs-gui.

Gorm

Our conceptual model positioned Gorm as an application-layer tool, interacting primarily with libs-gui for GUI design. We noticed an additional direct dependency between Gorm and libs-base. This suggests that Gorm interacts directly with core system functions, likely for file handling, object persistence, or other project management tasks. Rather than Gorm being purely an interface design tool, Gorm interacting with system-level functionality means it likely stores and manages UI elements persistently. This divergence from the conceptual model suggests that Gorm is more tightly integrated into the GNUstep ecosystem than initially believed, functioning not just as a front-end tool but as a system-aware application. More specifically, there is a direct dependency on libs-base through its use of the NSArchiver and NSUnarchiver classes (GNUstep Archiver classes, n.d.). These classes are responsible for encoding and decoding objects, allowing Gorm to save and load user interface designs.

Internal Dependencies of the libs-back Subsystem

The libs-back subsystem is a core part of the GNUstep project, providing the infrastructure for rendering graphical user interfaces (GUIs), managing windows, handling user input, and interacting with platform-specific systems (GNUstep libs-back, n.d). Its architecture is modular, with distinct functional areas that work together to create a portable, flexible, and efficient GUI framework. This part of the report explores the conceptual and concrete architecture of libs-back, focusing on its key features and their interactions.

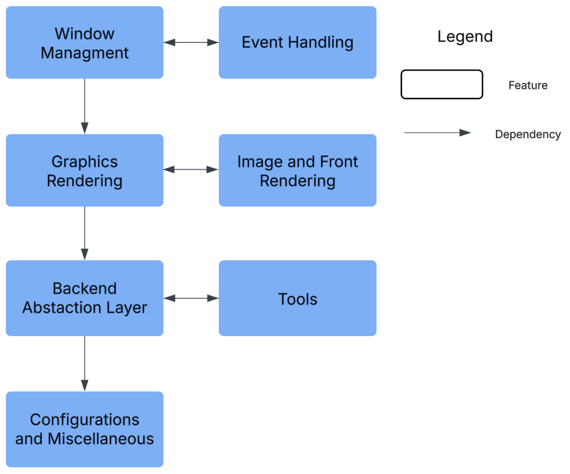

Figure 5: Proposed Conceptual Architecture of libs-back

In our conceptual architecture we find that libs-back is designed for modularity, portability, and clear separation of concerns. At the base, Configuration and Miscellaneous handle system settings and utilities. The Backend Abstraction Layer builds on this, providing a unified interface for platform-specific operations. The Graphics Rendering layer depends on the backend for drawing, while Image and Font Rendering handles images and text. Window Management relies on the backend for window operations and works with Event Handling to process user interactions. Tools assist in development and configuration. This layered, modular design ensures cross-platform compatibility and maintainability, with each component focusing on a specific role while working together to deliver a cohesive GUI framework.

The Graphics Rendering feature is responsible for drawing all visual elements on the screen, including shapes, lines, colors, and textures. It manages the drawing environment, such as creating drawing contexts, handling coordinate systems, and compositing visual elements. For example, it renders UI controls like buttons and sliders. The Backend Abstraction layer abstracts platform-specific details, enabling the GUI library to run on multiple platforms without significant changes to higher-level code. It provides a unified interface for creating windows, processing input events, and rendering graphics. The Window Management feature handles the creation, display, and lifecycle of windows and their associated views. It configures window properties (e.g., size and position) and manages the hierarchy of views within a window. The Event Handling layer manages user interactions, such as mouse clicks and keyboard input, and dispatches them to the appropriate UI elements. It represents user input as events (e.g., NSEvent) and routes them through a responder chain. The Image and Font Rendering feature focuses on rendering images (e.g., icons, photos) and text (e.g., labels, paragraphs) within the GUI. It builds on the Graphics Rendering layer to draw images and text, adding specialized functionality for image decoding, font management, and text layout. The Tools feature includes utility programs and scripts for configuring, testing, and debugging the GUI library. These tools handle tasks like managing resources and optimizing performance. The Configuration and Miscellaneous feature handles system settings, low-level utilities, and other tasks that don't fit into the primary functional areas. It manages front-end and back-end configuration and provides utilities for memory management, debugging, and runtime initialization.

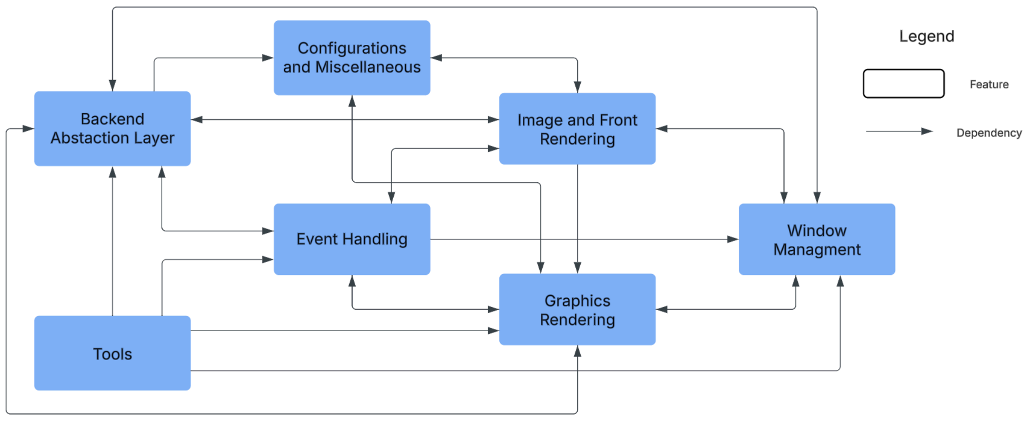

Figure 6: Concrete Architecture of libs-back

Comparing the conceptual and concrete architectures reveals key structural differences that impact the system's modularity, dependencies, and overall functionality. The concrete architecture provides a more detailed and nuanced understanding of the system's interactions compared to the conceptual view. While the conceptual architecture offers a high-level overview, the concrete architecture reveals additional dependencies and interactions that refine our understanding of how the system operates.

For example, in the conceptual view, Tools were thought to depend only on the Backend Abstraction Layer. However, the concrete architecture shows that Tools also depend on Event Handling, Graphics Rendering, and Window Management. This indicates that tools interact with multiple layers to configure, test, and debug the GUI library, requiring access to event processing, rendering, and window management functionalities. This discrepancy can be seen because Tools often need direct access to UI rendering and event handling for testing purposes (e.g., debugging how UI elements react to user input) (Model–View–Controller, n.d.). Since Tools assist in managing system settings, they naturally extend into Window Management. In practice, tools often interact with multiple layers rather than just the backend, leading to these extra dependencies.

Similarly, the relationship between Graphics Rendering and Image and Font Rendering is clarified in the concrete architecture. While the conceptual view suggested mutual dependency, implying that both modules work interchangeably, the concrete view shows that only Image and Font Rendering depend on Graphics Rendering. Graphics Rendering is a more general layer, and is responsible for drawing all UI elements, while Image and Font Rendering is a specialized subset that only deals with images and text. This creates a one-way dependency – Image and Font Rendering needs the Graphics layer to draw text and images, but Graphics Rendering does not necessarily rely on Image and Font Rendering. A mutual dependency that was assumed in the conceptual design would create coupling issues, making the system harder to maintain and optimize (Larman, 2004).

Additionally, the concrete architecture reveals that Image and Font Rendering has dependencies beyond Graphics Rendering, such as Window Management and Event Handling. This is because rendering text and images is not purely graphical – it is also tied to window position, scaling, and layering, meaning it naturally depends on Window Management to ensure correct placement. The interaction with Event Handling is required for dynamic text updates (e.g., typing in a text field, resizing fonts) to adjust the UI in response to user actions (Shneiderman, B., & Plaisant, C).

The Event Handling layer, which was initially thought to depend only on the Backend Abstraction Layer and Window Management, is shown in the concrete architecture to also depend on Graphics Rendering. To ensure a seamless user experience, UI components must visually change when clicked, hovered over, or interacted with (Shneiderman, B., & Plaisant, C). This requires Event Handling to trigger Graphics Rendering updates. Modern GUI frameworks minimize unnecessary redraws by re-rendering only the affected parts of the UI when an event occurs, meaning Event Handling must have a direct link to Graphics Rendering.
Finally, Window Management is revealed to be more interconnected than initially thought. In addition to depending on the Backend Abstraction Layer and Graphics Rendering, it also interacts with Event Handling and Image and Font Rendering. This can be explained as windows must react to events, such as Window resizing, closing, or moving, which requires direct communication with Event Handling (Kruchten, 1995). Not to mention, if a window resizes, text and images must also be re-rendered to fit new dimensions. This dependency ensures that texts and images dynamically adapt to window changes.

Overall, the libs-back subsystem is a modular and well-organized architecture that forms the foundation of the GNUstep GUI library. Its features Graphics Rendering, Backend Abstraction Layer, Window Management, Event Handling, Image and Font Rendering, Tools, and Configuration and Miscellaneous work together seamlessly to deliver a portable, flexible, and powerful GUI framework. The concrete architecture differs from the conceptual model in a few ways: Practical implementation requires tighter integration between components, optimizations for event-driven rendering reduce unnecessary computations, and GUI frameworks often involve overlapping dependencies for responsiveness. By analyzing the why behind these changes, we understand that concrete architecture prioritizes efficiency and maintainability over strict modularity. The additional dependencies reflect practical needs that were not fully considered in the conceptual phase.

Use Cases

When a user starts the application, it begins to prepare for the user to create or edit a graphical user interface. Gorm can create Objective-C objects directly using libobjc2. Gorm relies on libs-gui to initiate the graphical interface framework, as libs-gui is responsible for all the buttons, windows, etc. Once the GUI components have been set up, libs-gui sends "render" instructions to libs-back, from which libs-back translates these abstract components into visible graphics on the screen. Libs-back may require system utilities like handling file paths or managing strings, which is all done by libs-base. When libs-base needs to create or manage Objective-C system objects, it calls libobjc2, and then libobjc2 returns the necessary objects to libs-base, allowing it to complete the system service request. libs-base returns the required system data to libs-back, which it will need to finish rendering. Finally, libs-gui gives Gorm the prepared interface that is now visible on the screen and ready for user interaction.

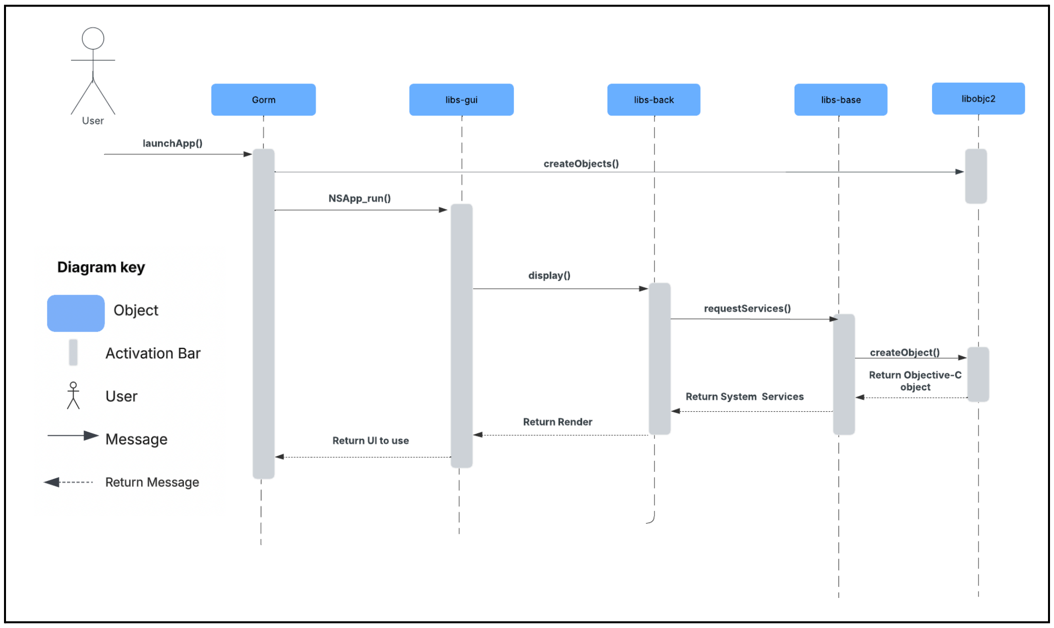

Figure 7. Sequence Diagram for Use Case to Run GNUstep

The user clicks the "Save" button on the application's graphical interface. The libs-gui captures this event and prepares to handle it. It forwards the event to the application's controller, which is responsible for processing what "Save" means. The application's controller formats the data in the correct file type to be saved (e.g., JSON, .txt, etc), and then it calls libs-base to handle the system-level task of writing the data to a file. libs-base then need a file path, so it asks libobjc2 to create an NSString object (which is an Objective-C object) for the file path, and libobjc2 returns the NSString file path to libs-base. libs-base then attempts to write to the file and returns a success or fail status to the App controller. The app controller prepares a log entry internally that tracks the file (e.g., File saved at 8:48 am). The app then updates the status label in the GUI with a success/fail message. Finally, libs-gui tells libs-back to render the updated label and any other changes, and libs-back finishes rendering and notifies libs-gui.

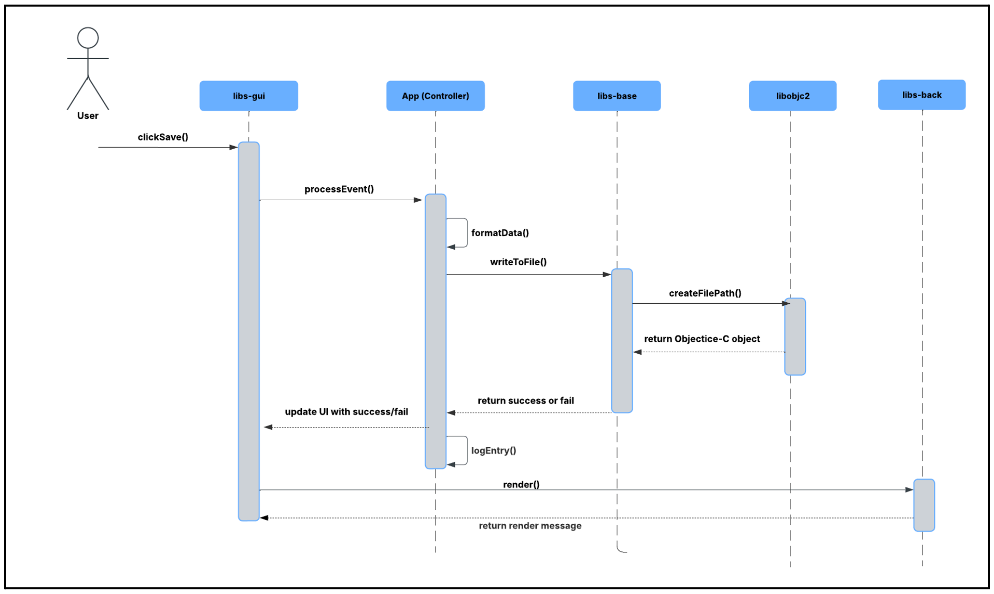

Figure 8: Sequence diagram for use case of saving a file on an application

Concurrency

GNUstep uses a multi-threaded architecture for efficient concurrency, facilitated by the libs-base library. This library enables the creation and management of threads, allowing applications to perform background tasks like network requests or file I/O without blocking the main thread. It also supports synchronization mechanisms, such as locks and semaphores, to prevent race conditions and ensure safe access to shared resources (GitHub, n.d.). For GUI applications, long-running tasks are offloaded to background threads, maintaining a responsive user interface. While concurrency can improve application performance, poor design choices, such as excessive context switching or incorrect thread management, can lead to performance bottlenecks (GeeksforGeeks, n.d.).

Lessons Learned and Limitations

During this report, we learned how to use the tool Understand to analyze software systems, specifically GNUstep and extract dependencies of the subsystems. By using Understand, we were able to visualize the architecture clearly, which gave us a better picture of the system's internal structure. However, we encountered a few unexpected dependencies, which we found challenging to analyze due to a lack of documentation between subsystems. We attempted to do additional research for these dependencies and relied on our knowledge and inferences.

As far as working together, we recognize that there is still room for improvement, particularly when it comes to starting our work earlier. In this assignment, several subsections were interdependent; for instance, the sequence diagrams relied heavily on having a completed concrete architecture. This meant that if one part was delayed, it directly impacted the progress of other sections. Additionally, because we had to learn to use new software tools like Understand, this process took more time than we initially expected and wasn't fully accounted for in our planning. Had we started earlier, we could have avoided last-minute rushing and allowed ourselves more time to navigate the learning curve of the tool. One key takeaway from this report is the importance of beginning work well in advance, not only to meet deadlines but to give ourselves the necessary buffer time for unexpected challenges, such as learning new tools or analyzing complex dependencies. Moving forward, we aim to prioritize early starts to improve both the quality of our work and our overall workflow as a team.

Conclusion

Through our analysis, we gained valuable insights into GNUstep's architecture and system dependencies, refining our understanding from our initial conceptual model. Discovering the key subsystem libobjc2, which was not included in our conceptual model, revealed its foundational role with bidirectional dependencies across multiple subsystems. We also identified that apps-gorm interacts with core libraries in addition to GUI components.

Our reflexion analysis highlighted the importance of validation in software architecture. The discovery of two-way dependencies between libs-base, libs-corebase, libs-gui, and libs-back showed that GNUstep follows a more interdependent system rather than a hierarchical one. We also identified challenges related to limited documentation and unexpected dependencies that required additional research and inferences.

Ultimately, this paper provided us with a deeper understanding of GNUstep's architecture and how it differed from the conceptual model. Moving forward, exploring concurrency mechanisms and performance optimizations could enhance GNUstep's maintainability and efficiency.

Data Dictionary

Open-source: A code base that is publicly accessible to be used or contributed to
Runtime environment: Software component(s) that provide the necessary infrastructure for executing applications

Naming Conventions

API: Application Programming Interface
ARC: Automatic Reference Counting
Gorm: Graphical Object Relationship Modeller

GUI: Graphical User Interface
I/O: Input/Output
IPC: Inter-Process Communication (IPC)
UI: User Interface
XML: Extensible Markup Language

References

GeeksforGeeks. Synchronization in Java. (n.d.). https://www.geeksforgeeks.org/synchronization-in-java/

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

Kruchten, P. (1995). Architectural blueprints—The "4+1" view model of software architecture. *IEEE Software, 12*(6), 42–50. https://www.researchgate.net/profile/Philippe-Kruchten/publication/238381956_A_41_View_Model_of_Software_Architecture/links/570ec85208aee76b9dadff07/A-4-1-View-Model-of-Software-Architecture.pdf

Lang/Libobjc2: Replacement objective-C runtime supporting modern objective-C features. FreshPorts. (n.d.). https://www.freshports.org/lang/libobjc2/

Larman, C. (2004). *Applying UML and patterns: An introduction to object-oriented analysis and design and iterative development* (3rd ed.). Prentice Hall. https://personal.utdallas.edu/~chung/SP/applying-uml-and-patterns.pdf

Model–view–controller. Wikipedia. (n.d.). https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller?utm_source=chatgpt.com

NSArchiver class reference. GNUstep. *(n.d.)* https://www.gnustep.org/resources/documentation/Developer/Base/Reference/NSArchiver.html

NSInvocation.m source file. GitHub. *(n.d.).* https://github.com/gnustep/libs-base/blob/9c6bd9ed97c835a14d53abe2a92fe5443b513def/Source/NSInvocation.m

NSThread class reference. GNUstep. (n.d.). https://www.gnustep.org/resources/documentation/Developer/Base/Reference/NSThread.html#class$NSThread

Shneiderman, B., & Plaisant, C. (2010). *Designing the user interface: Strategies for effective human-computer interaction* (5th ed.). Pearson. http://seu1.org/files/level5/IT201/Book%20-%20Ben%20Shneiderman-Designing%20the%20User%20Interface-4th%20Edition.pdf

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