Concrete Architecture of GNUstep

February 13th, 2025

Group #8: 500 - Internal Server Error

Qiu Xing (Nathan) Cai - 21qxc@queensuca

Wanting Huang - 20wh18@queensu.ca

Haoxi Yang - 21hy11@queensu.ca

Yuda Hu - 21yh29@queensu.ca

Nicholas Wang - 22nw3@queensu.ca

Amethyst Shen - 21YC121@QueensU.ca

Table of Contents

Abstract

In our previous report, we investigated the conceptual architecture of GNUstep, identifying it as a layered system designed for modularity and maintainability. This report builds upon that foundation by analyzing its concrete architecture using SciTools Understand. Through dependency extraction, we uncovered new subsystem relationships, some of which deviated from our initial conceptual model. Unexpected dependencies emerged, while some predicted interactions were absent, leading to a refined understanding of GNUstep's internal structure.

Our findings confirm the layered design of GNUstep but highlight variations in subsystem interactions and dependencies. By comparing conceptual and concrete architectures, we identified key structural discrepancies and updated our conceptual model accordingly. This study provides a deeper insight into GNUstep's real-world implementation, offering a more accurate representation of its architecture and the implications of its design.

1.0 Introduction and Overview

We established that GNUstep employs a layered architecture to ensure modularity and maintainability, with a strong foundation in object-oriented design principles. This report extends that analysis by investigating the concrete architecture, verifying subsystem interactions, and identifying deviations between the conceptual and actual implementation. Through dependency analysis using SciTools Understand, we extracted relationships between key components, refining our architectural understanding and uncovering unexpected dependencies.

This report is structured as follows. Section 2 presents an updated conceptual architecture, incorporating insights gained from analyzing the concrete structure. We then describe our methodology for extracting dependencies and grouping subsystems, detailing the rationale behind our classification. Section 2.3 explores the top-level concrete architecture, focusing on subsystem interactions, architectural styles, and alternative approaches. A dedicated subsystem analysis in Section 2.4 examines the conceptual and concrete views of a selected component, followed by a reflection analysis comparing its expected and actual structure.

Section 3 provides an overview of GNUstep's external interfaces, covering its interaction with compilers, debugging tools, file systems, and rendering backends. In Section 4, we present two use cases with corresponding sequence diagrams, illustrating key execution flows and the dynamic behavior of the system. Section 5 conducts a reflexion analysis, contrasting our refined conceptual architecture with the actual implementation to evaluate structural discrepancies.

Finally, Section 6 summarizes our findings, while Section 7 discusses lessons learned throughout the dependency extraction and architectural analysis process.

Supporting materials, including a data dictionary and naming conventions, are included in Sections 8 and 9. This study provides a refined understanding of GNUstep's real-world implementation, validating its adherence to software architecture principles and highlighting areas for potential improvement.

2.0 Architecture Analysis

2.1 Updated Conceptual Architecture

For conceptual architecture, we still believe that GNUstep is designed with a Layered Architecture to ensure modularity and maintainability, while also following Object-Oriented Design Principles.

The architecture consists of five key subsystems, each with a well-defined role and interaction model. The base layer (libs-base and libs-corebase) provides fundamental system services, including memory management, file I/O, and networking. The application logic layer (libs-gui) manages UI components and event processing. The rendering layer (libs-back) ensures platform-independent rendering, allowing the system to work across multiple environments such as X11, Windows GDI, and OpenGL. Finally, Gorm acts as a visual interface builder, allowing developers to design GUI applications efficiently.

The diagram below visually represents the structure and interactions between these key components.

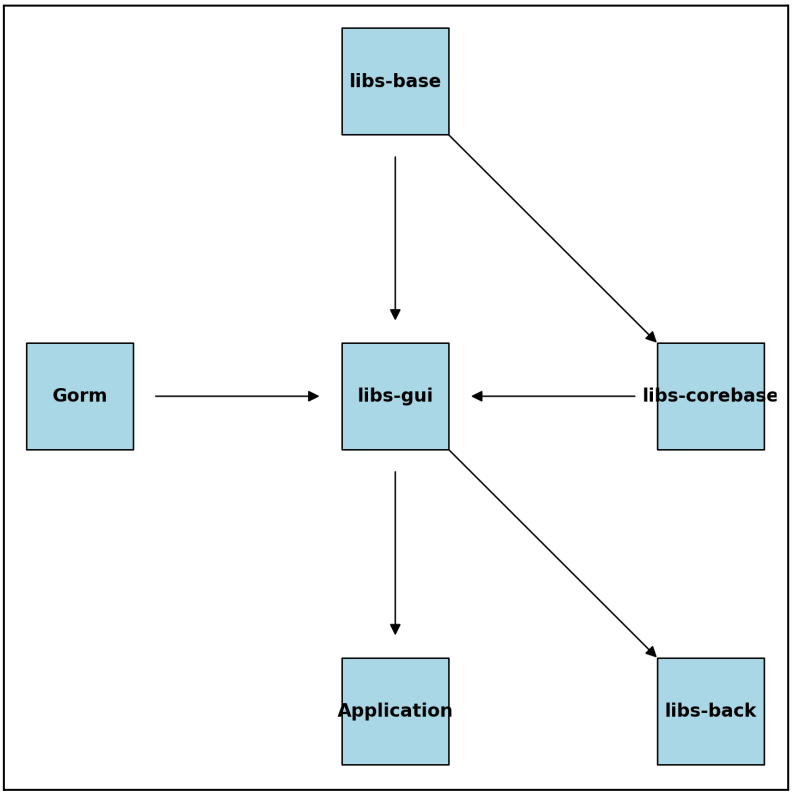

Figure 1: Conceptual Architecture Dependency Diagram, by Top-Level Subsystems

2.2 Architecture Derivation Process

2.2.1 Methodology for Extracting Dependencies

To analyze the dependencies within the GNUstep project, we employed the Understand SciTools, a static analysis tool widely utilized for comprehensively identifying module interdependencies.

Initially, a pre-built Understand project specifically tailored for GNUstep was loaded to guarantee accuracy and consistency in our analysis.

Next, we generated a Dependency Graph within Understand. The tool clearly illustrated both static (blue lines) and dynamic or indirect (red lines) dependencies between the subsystems. Static dependencies typically represent direct function calls or references, while dynamic dependencies involve indirect interactions, possibly through dynamic linking mechanisms.

For instance, Understand distinctly highlighted the significant dependency of libs-gui on libs-base, as well as the strong reliance of libs-base on libobjc2. Although we attempted to generate Call Graphs for deeper granularity, Understand encountered performance issues due to the project's complexity. Thus, our analysis was primarily based on the Dependency Graph below.

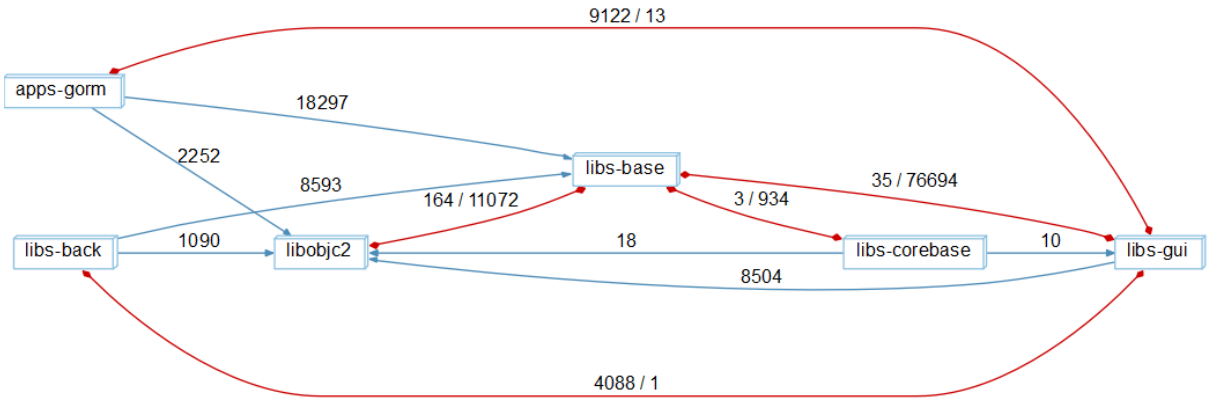

Figure 2: Concrete Architecture Dependency Graph, by Top-Level Subsystems

2.2.2 Process of Grouping Subsystems

The GNUstep subsystems were categorized primarily based on functional similarities, aligning closely with GNUstep's inherent architectural design and principles. Such categorization simplifies comprehension and aids in maintainability.

Subsystems were grouped as follows:

GUI Component Subsystem:

> **libs-gui**: Supplies essential graphical user interface functionalities, including models, color selection, printing, and related UI elements.
> **apps-gorm:** Offers advanced GUI functionalities and user interface design capabilities such as InterfaceBuilder and GormCore.

Core and Runtime Subsystem:

libs-base: Provides foundational components, including essential data structures and utilities.
libs-corebase: Supports core low-level functionalities and essential structures.
libobjc2: Fundamental Objective-C runtime, facilitating GNUstep's basic execution environment.

Backend Support Subsystem:

libs-back: Handles backend rendering functionalities, providing abstraction from the underlying hardware and operating systems.

Subsystem groups were derived from visual inspection of Understand dependency graphs, directory structures, and GNUstep's established module categorizations.

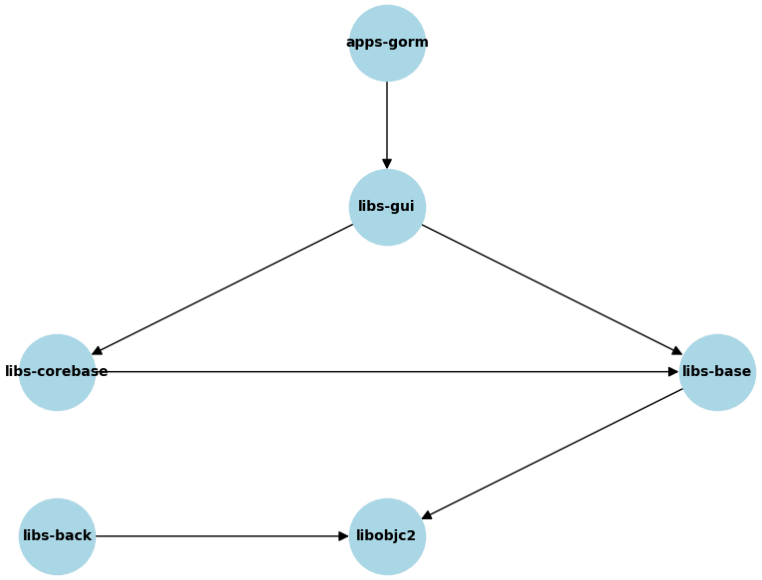

Figure 3: Subsystem Grouping Diagram

2.2.3 Justification for Architecture Decisions

This subsystem grouping strategy was selected after careful consideration of the following critical aspects:

Clear Functional Segmentation: By organizing subsystems according to their functional roles, we established a clear delineation of responsibilities. This approach enhances readability, simplifies the architecture, and facilitates easier future maintenance and functional extensions.
Reduced Coupling: Grouping subsystems based on functionality naturally reduced inter-subsystem complexity and dependencies. As evidenced by the Understand dependency graph, the subsystems exhibit straightforward and manageable interrelations, making modifications and enhancements less complicated.
Maintainability and Scalability: The chosen architecture simplifies future

development efforts by allowing individual subsystems to be upgraded or modified independently, thereby supporting scalability and easier integration of new functionalities.
Community Alignment and Official Standards: This grouping aligns with the standards recommended by the GNUstep community and official documentation, ensuring consistency, ease of adoption, and effective collaboration within the development community.

2.3 Top-Level Concrete Architecture

2.3.1 Identification of Concrete Subsystems

We used SciTools Understand to understand the real implementation of GNU steps. We used the tool to extract concrete subsystems based on the dependency graph. Each system in the software frame of GNUstep has its independent role, it can ensure the modular and maintainable structure.

libs-base: we can provide the basic service system, such as handling film, thread and memory management.
libs-corebase:support the extra core function.
libs-base: we can support extended utilities for apps;
libs-gui: The main is UI framework, it can handle the event management, rendering the components.
apps-gorm: it is a GUI build tool to design interfaces visually for the allowed developers.
libs-back:it is a platform independent UI rendering that can handle the graph rendering and make sure X11, Windows GDI and OpenGl.
libobjc2: when running the Objective-C, it can promote the object-oriented capabilities of GNUstep. (Objective-C Runtime, n.d)

2.3.2 Interactions Between Subsystems

According to the dependency analysis generated by the we find the follow key interaction.libs-gui heavily depends on the libs-base, it uses basic service to apply the application logic and UI event handling. It also interacts for the rendering with libs-back, and uses delegating graphics to ensure cross-platform support. libs-gui is close to the apps-gorm, and apps-gorm depends on libs-gui to define interface components. libobjc2 can support the memory management when it is running and object-oriented functionalities. Although most of the dependencies align with our conceptual architecture, we noticed some unexpected couplings where some UI operations bypass libs-gui and interact with libs-base.(Objective-C Runtime, n.d)

2.3.3 Architecture Style and Design Choices

GUNstep follows a Layered architecture, where each subsystem can provide a clear abstraction and builds it upon the lower-level services. This design enhances maintainability, modularity, and platform independence.

Additionally, GUNstep uses the Object-Oriented Design principles, it uses Objective-C to emphasize encapsulation and modularity. The interesting thing is the model-View_Controller pattern is to select the corn design of libs-gui, it can ensure a clear separation between model, view and controller. For model, application data handled by libs-base. For view, UI components are handled by lib-gui. For controllers, user interaction management is handled by libs-gui. Another key design principle depends on inversion where higher-level modules such as libs-gui, apps-gorm rely on abstracted lower-level functions such as libs-base and libs-back.(Taylor,Medvidović, Dashofy, 2010)

2.3.4 Alternative Architecture Styles

Although GUNstep predominantly follows a layered architecture, we realized the potential alternative styles and their impact for the system:

1.Microkernel Architecture: for this architecture, we use to break the functionalities into small, independently managed services to increase modularity. However, because messages pass between microservices, it might cause performance overhead. The reason of not choosing this because GUNstep performance is very relies on fast communication between components (Taylor,Medvidović, Dashofy, 2010)

2. Component-Based Architecture: this architecture separates the libs-gui and libs-back. Also, it can make the rendering logic and replacement easier. But the purpose increases the complexity of managing interdependencies, especially in event handling.The reason for not choosing it is that the layered method can be kept easily, and ensure the architecture and structured event driven interactions remain efficient. (Taylor,Medvidović, Dashofy, 2010)

3. Service-Oriented Architecture (SOA):this architecture allows the ibs-base and libs-gui function to be independent service to enhance the reusability.However, introducing REST or RPC-based service communication would add overhead and complexity. We choose layered architecture but not this is because SOA is more adapted for distributed applications, but GUNstep is designed for tightly integrated application frameworks. (Taylor,Medvidović, Dashofy, 2010)

Finally, the layered architecture is the best selected for GUNstep, it can keep the balance of maintainability and efficiency. It ensures clear separation of concerns, it allows tight integration where necessary for performance optimization at same time.

2.3.5 High-Level Reflexion Analysis

There are some divergences we have observed, for example, libs-back should be dependent on libs-base and libs-corebase but it turns out those two subsystems also depend on libs-back even though those two subsystems provide the core dependencies for all of GNUStep.

libs-corebase→libs-back

There is a special macro for TRUE and FALSE under libs-back/source/xdps/parseAFM.c that's use in libs-corebase as an equivalent for boolean data types like the return type for the function CFGregorianDateIsValid in libs-corebase/Source/CFDate.c

libs-base→libs-back

The boolean macro is also used in libs-base like it is in libs-corebase like for libs-base/Source/libgnustep-base-entry.m using it on line 62.

We believe for situations like this where there's a macro that serves as the boolean data type, which will likely be used everywhere, it would be a better design choice to put it under libs-base or libs-corebase since it is such a core functionality.

2.4 Subsystem Analysis

We chose to do an analysis on the libs-gui subsystem, covering its conceptual and concrete views, followed by a reflection analysis.

2.4.1 Conceptual View

The libs-gui subsystem is the graphical user interface (GUI) framework in GNUstep, also known as the GNUstep GUI Library, which is responsible for managing UI components, handling user interactions, and abstracting rendering operations. It provides the building blocks for application interfaces, ensuring that developers can create interactive and visually consistent applications.

At a high level, libs-gui follows a layered design. It defines UI elements like buttons, text fields, and menus while delegating rendering to libs-back. This separation ensures that UI logic remains independent of platform-specific drawing operations, allowing GNUstep applications to be cross-platform. The subsystem also manages event processing, ensuring that mouse clicks, keyboard inputs, and system interactions are correctly handled and dispatched. The Model-View-Controller (MVC) pattern is central to libs-gui. The Model represents application data and logic, the View manages UI components, and the Controller processes user input. This separation improves modularity and maintainability, preventing UI logic from being tightly coupled with business logic.

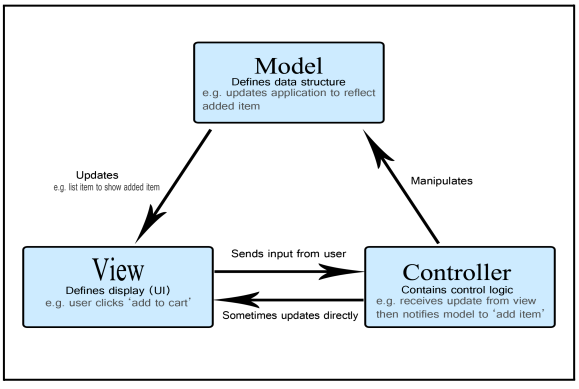

Figure 4: Illustration of MVC Pattern

While libs-gui provides high-level UI functionalities, it relies on other subsystems for critical operations. libs-base supplies core utilities, libobjc2 enables runtime object management, and libs-back handles all graphical rendering. These interactions ensure that libs-gui remains focused on UI structure and event management, rather than low-level drawing operations.

By maintaining clear abstraction layers, reusable components, and a modular structure, libs-gui serves as a flexible and scalable framework for GNUstep applications. Its architecture ensures UI consistency, cross-platform compatibility, and efficient event handling, making it a foundational component of the system.

libs-gui is divided into two sections, a frontend which contains code relating to which is independent of platform and display system. Then there is a backend which handles all display system dependents code, such as calls to the OS[1].

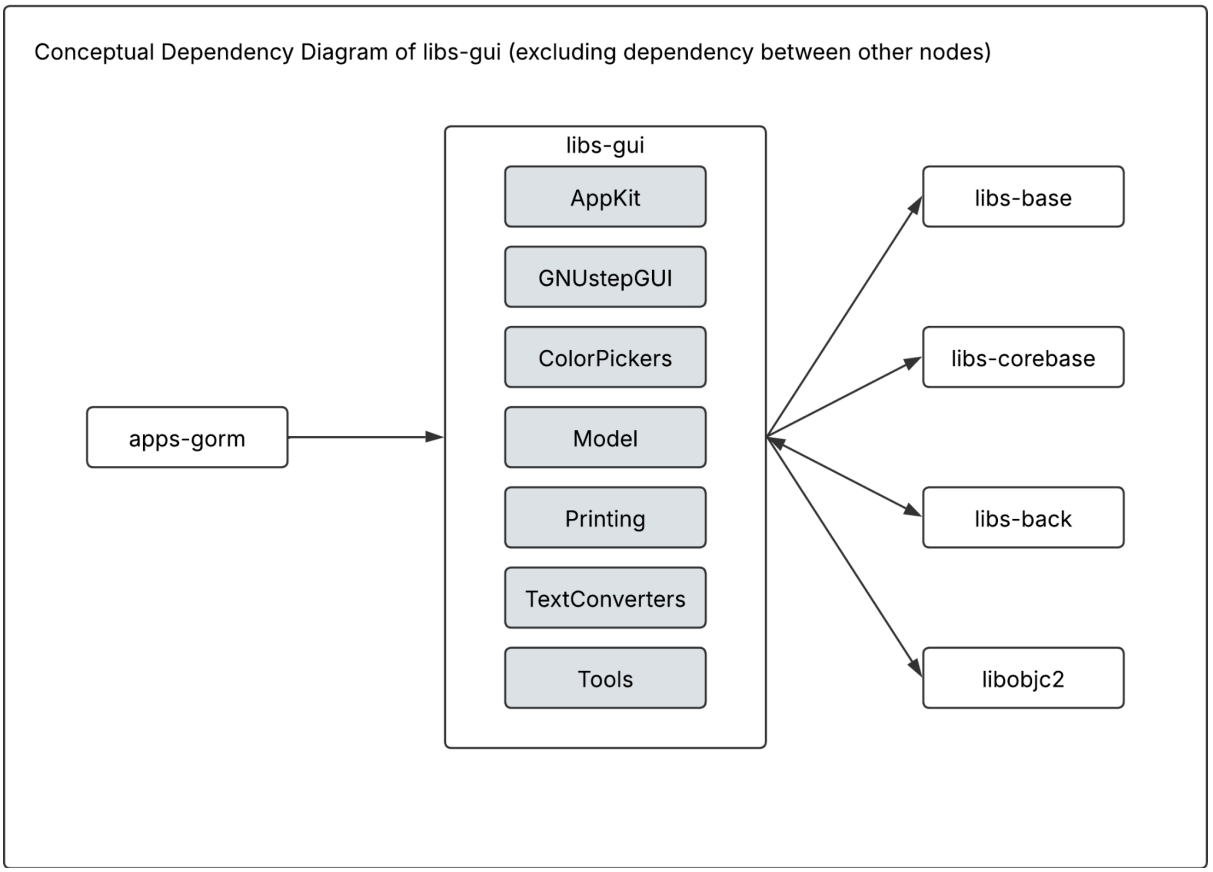

Figure 5: Conceptual Architecture Dependency Graph, by Top-Level Subsystems, focusing on libs-gui only

2.4.2 Concrete View

The libs-gui subsystem is organized as a collection of practical components that deliver specific functionalities for graphical interfaces within the GNUstep project. Internally, it includes clearly defined modules such as UI controls, event handling mechanisms, printing utilities, text conversion, and testing frameworks. Each module has clear responsibilities: for example, the Model handles data representation, while modules like ColorPickers and Printing address specialized GUI functionalities.

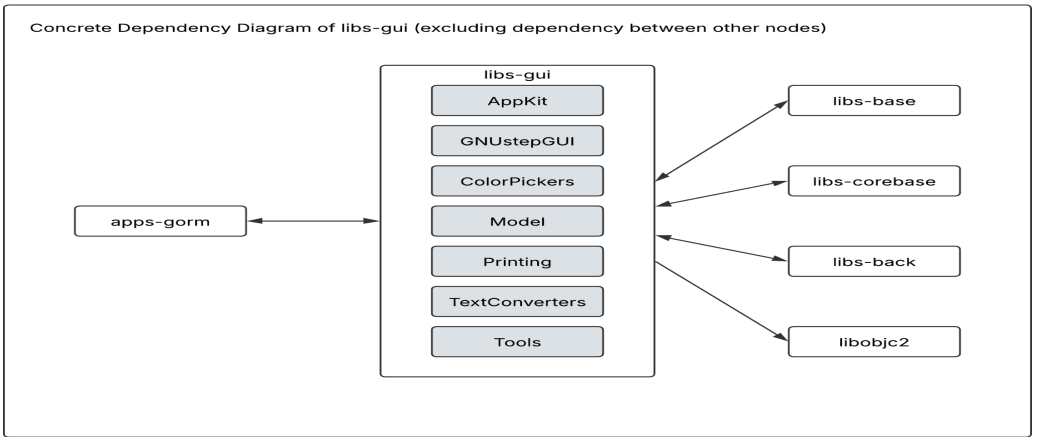

Figure 8: Concrete Architecture Dependency Graph, by Top-Level Subsystems, focusing on libs-gui only

Central to libs-gui is the NSView component, which provides a foundation for visual elements. Components such as NSButtonCell, NSTextField, and NSMenuItem indirectly inherit from NSView to define interactive UI elements. These components encapsulate their behaviors and interfaces clearly, promoting reuse and ease of customization across applications.

Event management within libs-gui relies primarily on the NSEvent component, which captures and processes user interactions like mouse clicks and keystrokes. While most rendering tasks are delegated to the dedicated libs-back subsystem, certain elements within libs-gui still retain minor rendering responsibilities, causing some unnecessary complexity and redundancy in the subsystem's implementation.

In practice, libs-gui extensively utilizes functionality provided by other GNUstep subsystems. It leverages libs-base for essential system-level operations and interacts closely with the Objective-C runtime (libobjc2) for dynamic object and memory management. Tools such as apps-gorm, the GUI builder, use libs-gui directly to visually construct user interfaces, further demonstrating its central role.

2.4.3 Reflection Analysis for 2nd level Subsystem

As we can see, there is a clear difference, in the conceptual view, there were no bi-dependencies between any of the subsystems of GNUStep, but in the concrete view, they are all bi-directional except for libobjc2. So there are several divergences but no absences.

There are several subsystems that use each other as dependencies multiple times for different reasons, so we'll investigate one case for each unexpected dependency.
libs-base → libs-gui
One example is in the file libs-base/Source/NSMessagePortNameServer.m in lines 522 and 626, libs-base retrieves the port name through the NSMessagePort class's property several times which is from libs-gui/Source
libs-gui → apps-gorm
libs-gui overrides some of the methods of apps-gorm/GormCore such as toolbarSelectableItemIdentifiers in libs-gui/Source/NSTabViewController.m
libs-corebase → libs-gui

libs-corebase includes the config.h file several times from libs-gui/Source such as in libs/corebase/Source/CFRunloop.c

3.0 External Interfaces

GNUstep interacts with various external systems to support application development, debugging, and execution across different environments. It facilitates interprocess communication through Distributed Objects, enabling seamless interaction between separate processes by allowing method invocation across them. Additionally, Mulle XML-RPC provides remote procedure call (RPC) capabilities over HTTP, allowing applications to integrate with web-based services efficiently.

For compilation and build management, GNUstep-make ensures platform-specific configurations, defining linking rules and handling dependencies. Debugging support is available through tools like GDB and LLDB, which allow developers to analyze program execution, set breakpoints, and inspect runtime behavior. Logging is managed via NSLog, capturing critical system events during execution.

The file system interface, managed by NSFileManager, enables applications to handle file operations, resource management, and persistent storage, including configuration files in .plist format. GNUstep also supports database interaction, allowing structured data storage by mapping Objective-C objects to relational databases. Additionally, platform-specific rendering backends ensure that graphical output remains consistent across different operating systems, including X11, Windows GDI+, and Core Graphics.

These interfaces collectively enhance modularity and cross-platform compatibility, allowing GNUstep applications to integrate smoothly with external services, manage data efficiently, and provide a stable development environment.

4.0 Use Cases & Sequence Diagrams

4.1 Use Case 1: Menu Action Execution in GNUstep

At run-time, GNUstep Gorm handles user interactions through menu actions that trigger predefined operations. This use case illustrates the process of selecting a menu option and executing the associated command.

The process begins when the user clicks on an option from a menu, for example "Open...". The NSMenu (the “view” in MVC) component’s predefined action will be triggered, which requests the NSDocumentController (the “controller” in MVC) to load a document via the selector mechanism.

Once the document controller receives the request, it invokes the corresponding methods of NSDocument (the “model” in MVC), to initialize a document and then display it as demanded. The NSDocument, when required to perform display, will initialize a NSWindowController, set it properly, and activate it, hence assigning the responsibility of displaying, which is not a business of a document, to the window controller.

This process highlights the structured event-driven design of GNUstep, where actions are dynamically resolved at runtime using selectors, allowing flexible handling of UI commands.

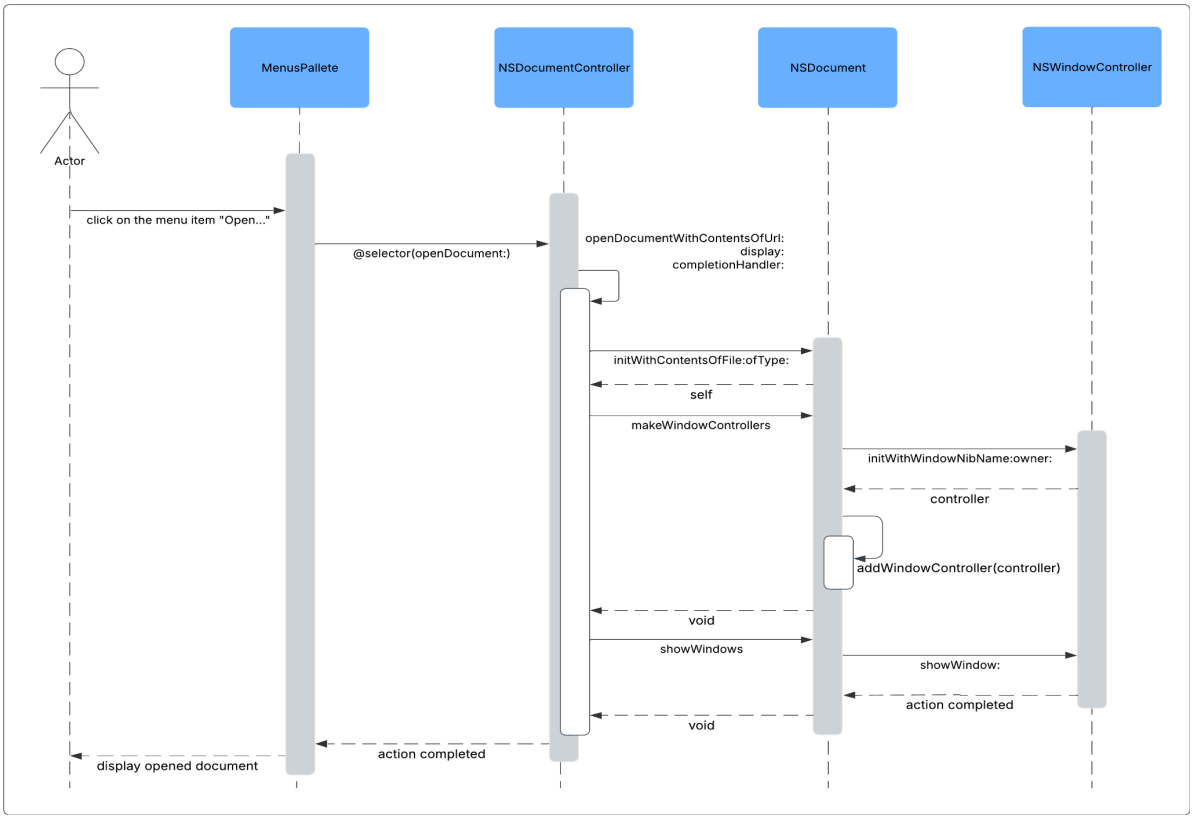

Figure 9: Sequence Diagram for Menu Action Execution

4.2 Use Case 2: Drag-and-Drop Image Upload

This use case focuses on GNUstep's drag-and-drop event processing, demonstrating how dragging handling is done.

The sequence starts when the user drags an image and drops it onto an object, including but not limited to NSButton, NSImageView, and NSMenuItem, in the Gorm application. The GormViewEditor will respond to dragging events and then call the responsible delegate, while ControlsPallete registered itself as a delegate for image. The methods of ControlsPallete will be called, to verify if the dragged content (the image) is accepted, and if it is, deposit the image onto the object that the user is editing.

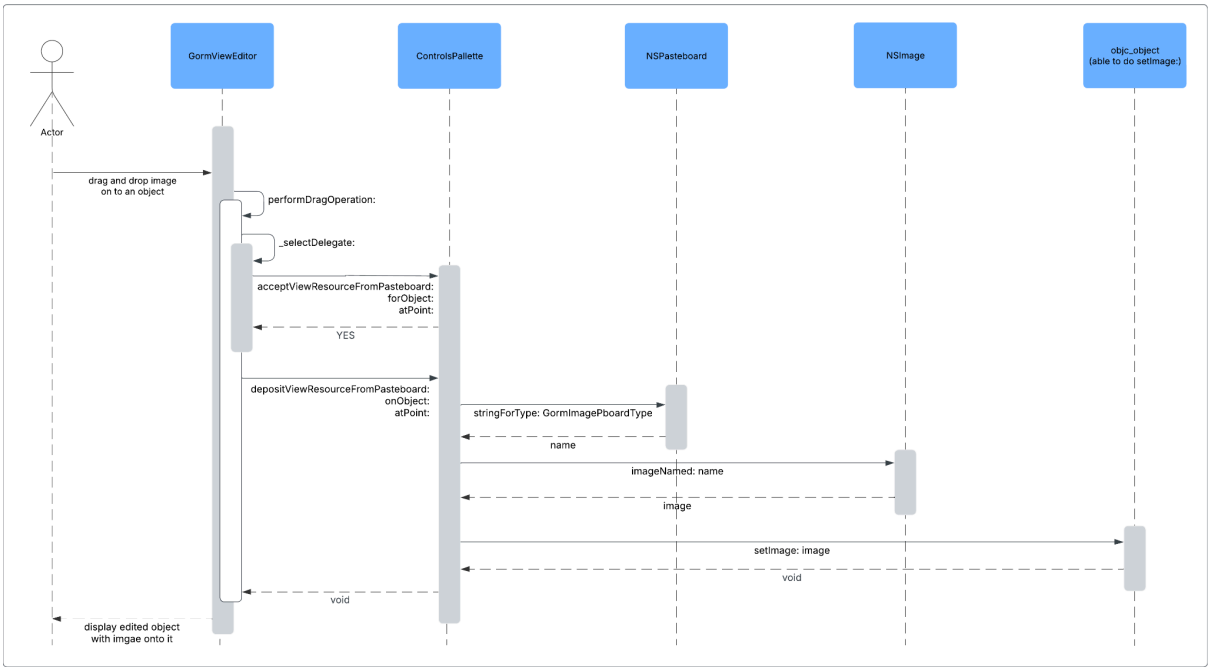

Figure 10: Sequence Diagram for Drag-and-Drop Image Upload

5.0 Conclusion

Our report analyzed the concrete architecture of GNUstep, refining the conceptual understanding established in our previous study. We identified key interactions, unexpected dependencies, and deviations from our initial conceptual model. The findings reaffirm GNUstep's layered architecture, ensuring modularity and maintainability while revealing bidirectional dependencies between certain subsystems, such as libs-gui and libs-back, that challenge strict modular separation.

Our reflexion analysis highlighted structural discrepancies between the conceptual and concrete architectures, leading to updates in our subsystem classification. Additionally, an examination of external interfaces demonstrated how GNUstep integrates with compilers, debugging tools, file systems, and rendering backends, ensuring platform compatibility. The use case analysis and sequence diagrams further illustrated key runtime interactions, reinforcing the event-driven nature of the system.

Overall, this study provided a deeper understanding of GNUstep's real-world implementation. While the architecture effectively supports cross-platform development, areas for improvement include reducing unnecessary dependencies and improving documentation to enhance maintainability and future development.

6.0 Lessons Learned

Through this project, our group gained a deeper understanding of how conceptual and concrete architectures can differ in practice. Initially, we believed our conceptual model captured the core structure of GNUstep, but after performing dependency analysis with SciTools Understand, we realized that real-world implementations often

have unexpected complexities. Some subsystems had bidirectional dependencies we didn't anticipate, which made us rethink the strict separation between layers.

One of the biggest takeaways was how difficult it can be to accurately map software architecture without the right tools. At first, navigating the dependency graphs was overwhelming, especially when trying to distinguish between direct and indirect dependencies. However, by refining our approach and focusing on key subsystems, we were able to make sense of the structure and improve our understanding of how GNUstep components interact.

Another lesson was the importance of external interfaces. While we initially focused on GNUstep's internal components, we soon realized how much it relies on external systems like compilers, debugging tools, and platform-specific rendering backends. These connections are just as important as the internal structure because they impact how the system runs across different environments.

Lastly, working with sequence diagrams helped us visualize how the system actually processes user interactions. Instead of just looking at static dependencies, seeing how different components communicate during runtime gave us a much clearer picture of how GNUstep operates.

7.0 Data Dictionary

1. libs-base: Provides fundamental system functionalities, including string handling, collections, and memory management.
2. libs-gui: Manages graphical user interface components and user interactions.
3. libs-back: The rendering backend that interacts with platform-specific graphics APIs like X11, Windows GDI+, and Core Graphics.
4. Gorm: A graphical interface builder that enables the design and manipulation of UI components.
5. NSApplication: The main application object responsible for managing the event loop and application lifecycle.
6. NSRunLoop: A continuous loop that listens for and processes input events, ensuring UI responsiveness.
7. NSFileManager: Facilitates file system interactions such as reading, writing, and resource management.
8. Distributed Objects: Enables interprocess communication by allowing objects to invoke methods remotely across different applications.
9. XML-RPC: A remote procedure call (RPC) protocol that allows GNUstep applications to interact with web-based services over HTTP.

8.0 Naming Conventions

1. NS (NeXTSTEP Prefix): Many Objective-C classes in GNUstep retain the NS prefix (e.g., NSString, NSWindow), a convention inherited from NeXTSTEP.
2. MVC (Model-View-Controller): The architectural pattern used in libs-gui, separating data models, user interface components, and business logic.
3. API (Application Programming Interface): The set of functions and protocols used for interacting with GNUstep libraries.

1. GNUstep-make (Build System Naming): A tool that follows standardized naming for makefiles, defining build configurations.
2. NSLog (Logging Convention): The function used for runtime logging, following Apple's convention of prefixing logging utilities with NS.
3. Rendering Backends (Platform-Specific Naming): The libs-back module contains implementations like x11, winlib, and cairo to denote different graphical rendering backends.

9.0 References

- [1] https://github.com/gnustep/libs-gui/blob/master/ANNOUNCE
- [2]*Objective-C runtime*. Apple Developer Documentation. (n.d.). https://developer.apple.com/documentation/objectivec
- [3] https://github.com/gnustep/libs-gui
- [4]Taylor, R. N., Medvidović, N., & Dashofy, E. M. (2010). *Software architecture: Foundations, theory, and Practice*. Wiley.