A2 Report: **GNUstep Concrete Architecture**
CISC 322/326 – Winter 2025
Group 4 – GNUstep Architecture

Authors:

- Seyed Ebrahim Haghshenas (21seh22@queensu.ca)
- Masih Hashemi (masih.h@queensu.ca)
- Kiarash Mirkamandari (kiarash.mirkamandari@queensu.ca)
- Bahar Rahimivarposhti (23vs27@queensu.ca)
- Kiarash Soleimani Roozbahani (21ksr5@queensu.ca)
- Andrew Ternopolsky (21at103@queensu.ca)

1. Introduction

In this A2 report, we examine the concrete architecture of GNUstep as implemented in the source code. Our goal is to contrast this real structure with the conceptual design documented in A1 and identify any key discrepancies or refinements. GNUstep, as an open-source reimplementation of the OpenStep standard, is comprised of multiple libraries—namely libs-base, libs-corebase, libs-gui, and libs-back—along with a modern Objective-C runtime (libobjc2). While these components were introduced in the conceptual architecture, the code-level analysis conducted here reveals additional details, such as partial usage of libs-corebase and the dynamic loading of multiple backends.

To help clarify the system's true layout, this report centers on four major sections: Section 2 presents the recovered Concrete Architecture, describing subsystem responsibilities and dependencies; Section 3 conducts a Reflexion Analysis, comparing the conceptual vs. concrete architectures; Section 4 focuses on Use Case Analysis, employing sequence diagrams to illustrate runtime interactions; and Section 5 provides Lessons Learned from performing this analysis in depth. Ultimately, we wrap up with concluding remarks in Section 6, highlighting implications and opportunities for future enhancements.

Through this process, we aim to provide a deeper understanding of how GNUstep's stated design objectives materialize in real code, as well as the practical challenges or evolutionary factors that shaped these libraries. Special attention is given to libs-gui, the subsystem implementing the GUI (AppKit) layer, which interacts heavily with the libs-back rendering system. In the upcoming sections, placeholders for diagrams are included to mark where architectural and sequence diagrams would further illustrate relationships and data flow. By bridging high-level theory with

low-level implementation, this A2 report lays the groundwork for a more robust, evidence-based approach to software architecture decisions.

---

2. Concrete Architecture

2.1 Overview and Subsystem Responsibilities

The concrete architecture of GNUstep can be described as a multilayer library stack that sequentially implements the OpenStep specification. At the bottom level, there exist Core libraries that provide core data structures and system application programming interfaces; above these, there is the GUI library with user interface classes, which in turn outsource the actual rendering process to what is termed the backend. This design style supports GNUstep in isolating platform-independent logic from implementations that depend on the operating system. While such a separation has long been promoted, a look at the level of implementation comes with many sub-libraries and operating system interaction adding further worth to the nominal layers.

1. **libs-corebase**
   The libs-corebase library is theoretically similar to CoreFoundation by Apple since it provides core C-based features like CFArray, CFString, run loops, and memory management. However, in practical usage, it is limited in a way that some parts of the Foundation level (libs-base) bypass libs-corebase when core functions lack implementation or implementation is lacking. While libs-corebase is important for specific low-level features, it does not solely offer all core base functions.

2. **libs-base**
   Operating within the GNUstep Foundation framework, libs-base includes fundamental classes such as NSObject, NSString, and NSArray, in addition to offering concurrency support through NSRunLoop and NSThread. Additionally, it supports integration with operating system services in regards to file operations, networking, and memory management. As a design principle, all the components above libs-base rely on these classes for basic operations. Reflexion analysis reveals that libs-base sometimes calls OS libraries directly, thus bypassing libs-corebase.

3. **libs-gui**
   Based on Apple's AppKit, libs-gui defines core elements in the user interface, like

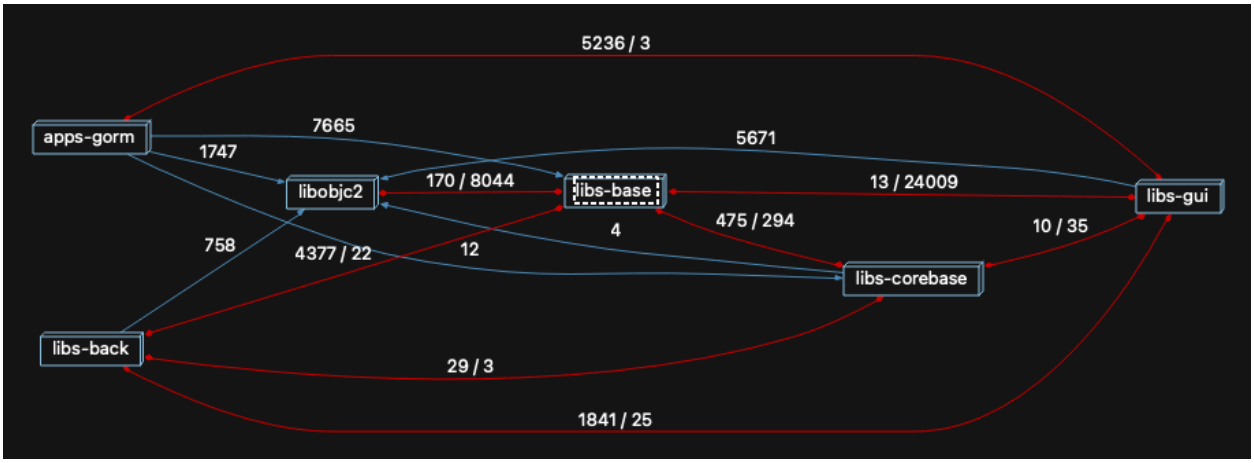
NSWindow, NSView, and NSButton, alongside their corresponding event management frameworks. Based in Model-View-Controller (MVC) architecture, it supports the division between the Model (data) and the View (user interface), with the interaction being coordinated by the Controllers. Being the most comprehensive subsystem in GNUstep, libs-gui includes all the code that is relevant to the user-facing widgets, organization in layout, and application-level event processing loops.

4. **libs-back**

   The subsystem is commonly termed as “gnustep-back” and is responsible for bridging libs-gui with the windowing application programming interface (API) of the underlying operating environment, like X11 in Unix, Win32 in Windows, or any display server. It executes drawing requests from libs-gui and converts them into platform-specific native APIs (e.g., XCreateWindow, Win32 GDI, or Cairo for rendering). The text states that several backend implementations exist, which are dynamically loaded depending on the platform or user settings, thus allowing cross-rendering without requiring a recoding of libs-gui.

5. **libobjc2**

   GNUstep relies on a sophisticated Objective-C runtime, libobjc2, that supports the richer features necessary for application construction, similar to Cocoa. While often forgotten in theory, libobjc2 plays a central role in supporting memory management semantics, message sending, categories, blocks, and reflection. A study of reflection uncovers that libs-base and libs-gui expressly rely on the dynamic dispatch abilities provided by this



runtime.

Figure 1 – High-Level Dependency Diagram of GNUstep

In Figure 1 (placeholder), arrows denote dependencies that are realized at compile-time or runtime, depicting the optional dependence of libs-base on libs-corebase, the heavy dependence of libs-gui on libs-base, and the call from the GUI subsystem to libs-back for operating system-specific rendering. The runtime environment beneath, libobjc2, underpins all of these components. This is consistent with our theoretical model of a layered architecture, albeit with finer-grained interrelationships.

2.2 Subsystem Interactions

Along with their divergent functions, the various library components interact with each other through complex mechanisms. As a point in fact, libs-gui manages the user event loop but relies on libs-base in order to run the underlying run loop logic. When libs-gui is ready to make a window appear, it calls libs-back, which in turn calls the operating systems' application-level APIs in order to display the window. A look through the source tells us that backends can be swapped depending on the environment—using dynamic linking—demonstrating a strong concern with portability.

It is interesting that direct calls from libs-base to the system library can be mediated by libs-corebase, demonstrating a real-world solution: if a frequently-used function or type is implemented poorly in libs-corebase, developers would be likely to make a system call. At an architecture level, this creates a chain that does not perfectly match the optimal pattern proposed. The resulting trade-off seems to balance both execution optimization and developer convenience, which has been widely adopted as a best practice in the codebase.

2.2.1 The libs-gui Focus

As libs-gui plays a central role in the GNUstep user environment, libs-gui was fully scrutinized. In addition to providing window and control classes, libs-gui is charged with enabling the following features:

- **Event Routing**: NSApplication handles events from the operating system, routing them to the proper NSWindow or NSView.
- **MVC Support**: Specific classes or patterns within libs-gui encourage separation of concerns. For example, NSTableView expects a data source (model) and optionally a delegate (controller).
- **Drawing Abstractions**: High-level drawing commands (e.g., NSBezierPath) are provided in libs-gui, but the actual rendering instructions are handed off to the backend (e.g., via Cairo or Win32 calls).

- **Target-Action Mechanisms**: Individual procedures in the codebase of the application can be associated with different user interactions, for example, menu choices and button clicks. Based on the dynamic nature of Objective-C, the libs-gui library calls performSelector: at runtime on the target specified.

The combination of these traits highlights that libs-gui is an object-oriented, event-driven framework. At the same time, the reflexion analysis in Section 3 shows that many conceptual assumptions about modularity are preserved, even though there may be additional bridging layers, macros, or direct system calls in the actual implementation to handle edge cases.

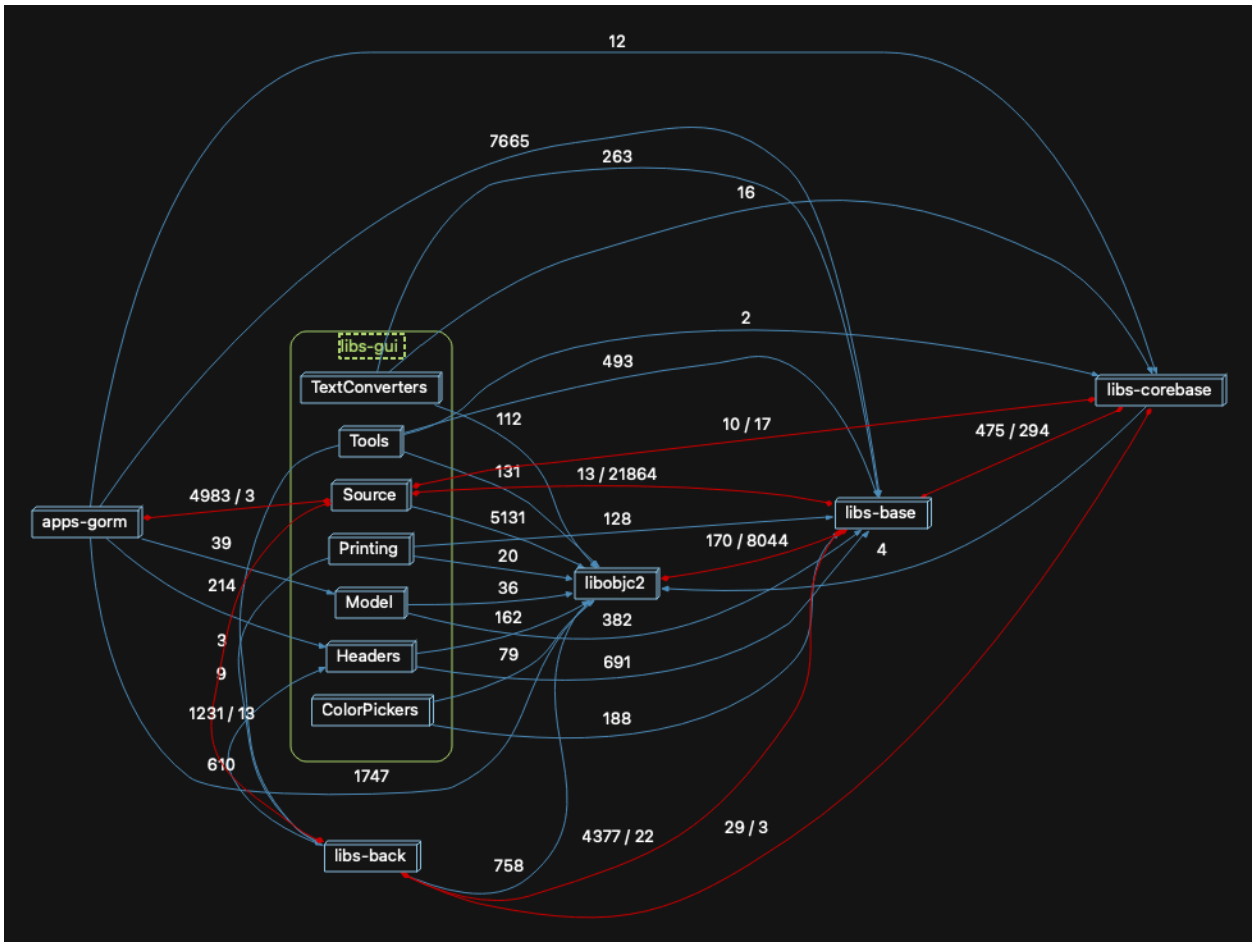


Figure 2 – libs-gui and libs-back Structural Overview

The conceptual diagram in Figure 2 captures the reliance of libs-gui classes on plugin-based back-end architecture. A circle or rectangle in this diagram represents a principal class or set of classes in libs-gui, while the connecting lines that lead to libs-back represent the areas that

contain dynamic calls. The phases align with the theoretical model but illustrate increased accuracy in the actual implementation in terms of code.

---

3. Comparison with Conceptual Architecture

3.1 Overview of Reflexion Approach

A reflective analysis was made by mapping the conceptual modules, as defined in A1, against the current source directories and symbol references. Differences were detected when new relations, or lack thereof, emerged among the dependency graphs at the level of code. In actual implementation, partial static analysis and code review were utilized in combination in order to support or refute each expected link. The outcome was a better comprehension of the architecture that accurately represents how the libraries are built and run at runtime.

3.2 Identified Discrepancies

1. **Objective-C Runtime (Extra Module)**

   The theoretical model proposed by A1 treated the Objective-C runtime as one coherent, unified whole, rarely as a separate subsystem. The reflexion model, by contrast, demonstrates that libobjc2 is a stand-alone component that is invoked by other libraries for linkage. This finding goes beyond implementation details; in fact, design patterns like delegation, target-action, and dynamic property introspection rely heavily upon the services provided by the runtime.

2. **Multiple Backends vs. Single “libs-back”**
   Theoretically, we assumed that there was one monolithic "backend" library that acts as a bridge between libs-gui and the operating system. In reality, we had several implementations (e.g., Cairo and Win32), and each is instantiated dynamically. This design supports platform compatibility. The theoretical model didn't show that environments or users would be able to switch between these backends at runtime, nor that separate codebases exist for each platform. The implementation reveals that libs-back essentially is an interface with several backends included.

3. **CoreBase Incompleteness**
   The suggested conceptual model is that libs-base would fully build upon CoreBase, in the

same way that Apple’s Foundation is built atop CoreFoundation. In reality, however, the implementation sometimes works around libs-corebase, instead calling operating system APIs directly for file i/o or concurrency functionality. This is due to parts of libs-corebase that are either unfinished or obsolete. Thus, the layering is not as strictly adhered to, making the call graph more tangled than the ideal layering would suggest.

4. **Performance-Driven Shortcuts**
   Another subtlety is the presence of Performance-Driven Shortcuts. Individual habits that are conceptually placed in libs-gui could be situated partly in libs-back in order to be faster or in order to directly integrate with the operating system. These direct calls can be located by reflection analysis, revealing that the code is not strictly compartmentalised according to conceptual boundaries.

   —----------

3.3 Rationale for Divergences

Every occurrence of discrepancy is supported by a reason. As an example, consistency in supporting various implementation backends is built on a cross-platform design that cannot be fully summarized in one conceptual arrow labeled as “Back.” Likewise, the specific calls from operating system functions in libs-base can be read as a historical record: because libs-corebase was lacking, people worked around the weaknesses in order to be able to advance. At the same time, the separate runtime is typical in C-based object-oriented systems like Objective-C; this feature, however, seems never properly illustrated in the conceptual figures, likely in order to make them brief.

These practical restrictions, far from diminishing the theoretical model, represent its augmentation, enlargement, or "adaptation" in order to ensure that GNUstep is stable, efficient, and adaptable. The remainder of this section complements these static observations by examining the behavior at runtime through two example scenarios.

---

4. Use Case Analysis

4.1 Launching a GNUstep GUI Application

In the first case, a user starts a GNUstep-based GUI application by choosing an icon or by invoking a specific command. When the operating system starts the application, the main() function is invoked, thereby invoking NSApplicationMain from the libs-gui library. This specific function is what initializes the singleton object of type NSApplication, sets up the main event loop, and tries to load a compatible backend from the libs-back library.

1. **Application Startup**: The process is started by the operating system, which connects to libs-base, libs-gui, and, indirectly, libs-back.
2. **GUI Initialization**: NSApplicationMain or +[NSApplication sharedApplication] creates global objects, sets up the UI environment, and configures the event loop with assistance from libs-base (through classes like NSRunLoop).
3. **Backend Registration**: libs-back is loaded—statically or dynamically—based on user or system defaults. For instance, on a Linux system with X11, a Cairo-based backend might be chosen. The library calls system APIs (XOpenDisplay) to connect with the window server.
4. **Main Window**: If the application's Info.plist file or the main nib file (named.gorm files in GNUstep jargon) defines a main interface, libs-gui uses backend function calls to instantiate it. The operating system then draws the newly created window on the user's screen.
5. **Event Loop**: When the main window is presented, the application enters an execution loop. Later, events like keyboard inputs and mouse buttons get forwarded from the operating system through libs-back, which then forwards them through libs-gui, enabling their routing to the target window or view object.

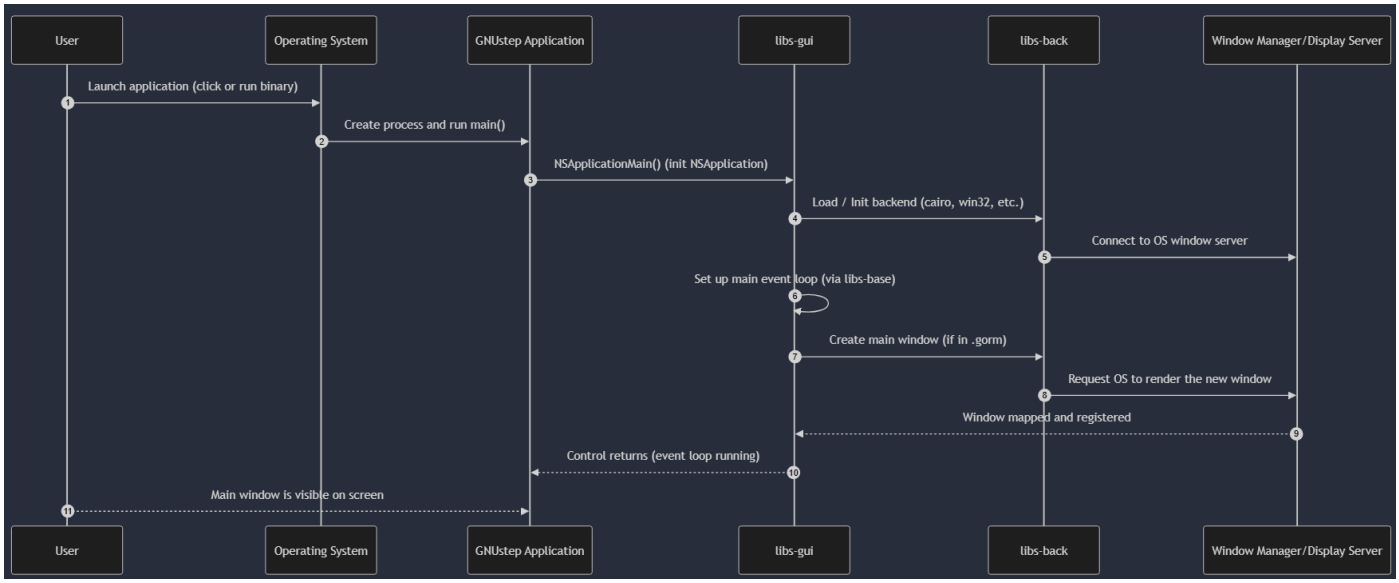


Figure 4 – Sequence Diagram for Launching a GNUstep GUI Application

Architecturally, this process follows the principles of conceptual layering; the application code is in contact with libs-gui, instead of directly dealing with operating system calls. At the same time, libs-gui sends platform-specific work through the specified plugin interface to libs-back. In this way, the user has a unified and working application without being aware of the underlying layered architecture.

4.2 Rendering a New Window at Runtime

Another example demonstrates the application in operation during execution, as it works by creating a new window. A user might, for instance, click "New Window" from a list, while in another example, the dialog might be invoked by a native event.

1. **Event Capture**: In the traditional event loop model framework, user events like button presses or menu selection are gathered by libs-back, forwarded to libs-gui, and eventually sent to the destination NSMenuItem or NSView object.
2. **Application-Defined Action**: If the user's click is tied to an action method, like myController.openNewWindow: Objective-C uses dynamic dispatch to call that method on the target object.
3. **NSWindow Creation**: When the application performs [newWindow makeKeyAndOrderFront:nil], the GUI library passes on the task of creating a suitable operating system-level window to the backend.
4. **On-Screen Display**: The operating system window manager is charged with displaying the rendered window, with corresponding events reported back through the same channel. The developer does not need to deal with operating system calls; the pre-defined layers ensure that platform-specific details are handled by libs-back.

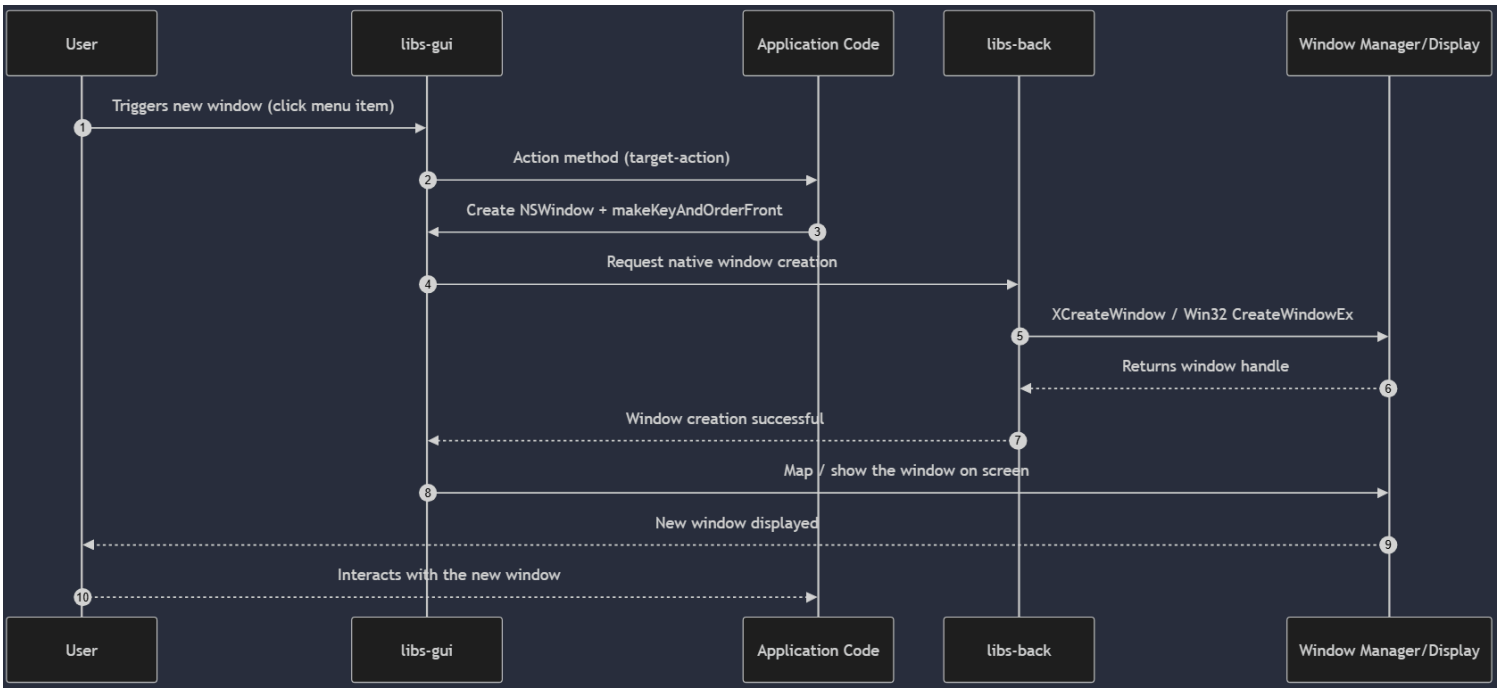


Figure 5 – Sequence Diagram for Window Rendering at Runtime

The information gathered from these dynamic flows demonstrates the importance that objective runtime (libobjc2) has in supporting target-action and delegation approaches. While the theoretical framework supports event-driven interaction, the observations made from such interactions clarify the division of labor among the different libraries: the main library creates run loops and data structures, the graphical user interface library creates objects and initiates the establishment of backend connections, and the backend utilizes operating system services for rendering.

---

5. Lessons Learned

5.1 Architectural Insights

A major finding is that the architecture of GNUstep, while adhering to a layered architectural pattern, provides enough flexibility to support either multiple or selective backends, and partial or optional inclusion of libs-corebase and advanced Objective-C features. Far from being an rigid framework, this architecture is a dynamic system shaped by cross-platform concerns, legacy

decisions, and performance constraints. This underscores the imperative of balancing theoretical design and actual implementation since many "clean" designs require modifications in practice to maintain their utility in the long run.

We further verified that the decoupling of the graphical user interface from OS rendering increases both modularity and portability, while adding complexity within. Dynamic linking combined with event-driven patterns permits developers to write code once, with libs-back dealing with operating system-specific requirements. At the same time, selective workarounds around libs-corebase demonstrate that the "perfect" layering is sometimes sacrificed for reality. Understanding such trade-offs is important in order to make future recommendations for improvement or refactoring.

5.2 Challenges and Observations

Restoring this architecture was a major challenge. Much of the documentation related to GNUstep is scattered, and the existing codebase is large and continually being updated. Reflected analysis required conceptual modules to be mapped onto their corresponding physical files, and inter-library dependency checking. A number of bridging or "assistant" modules that were omitted through simplifications in conceptual design were detected. These include, for instance, classes in libs-gui that call system-level APIs directly in order to utilize advanced text rendering, short-circuiting the conceptual pipe for the purpose of optimization.

In addition, recognition of libobjc2 as a separate, modern runtime demonstrates the strong influence that language-level artifacts can have in shaping architecture. While the theoretical model would place such artifacts in the category "compiler or runtime," the actual implementation is very much rooted in strong reflection and dispatch features, thus highlighting a very important difference between theoretical constructs and practical architectural design.

5.3 Implications for Future Architectural Changes

These revelations provide a platform for future enhancements. Specifically, if the GNUstep community were interested in adding or further developing the lib-corebase, a change that defines directly invoked operating system calls and ones that can be wrapped would make layering more feasible. Or, in supporting innovative platforms like Wayland in Linux, the plugin architecture in the backend can be extended without weakening libs-gui, essentially supporting the critical modular design. Understanding the subtleties in libobjc2 that control delegate calls, as well as memory management, would further help in supporting additional programming language integration, or in developing more advanced tools. In short, knowing particular details dispels oversimplified assumptions, allowing for exact amendments. The conclusions drawn from

the A2 analysis emphasize that architectural transparency depends on the coupling of theoretical models with practical coding realities faced every day.

---

6. Conclusion

This A2 report documents the concrete architecture of GNUstep, providing an in-depth comparison to the conceptual architecture formulated in A1. The layered structure—CoreBase, Base, GUI, Back—fundamentally holds, yet the real system includes pragmatic shortcuts, incomplete CoreBase reliance, multiple possible backends, and a standalone Objective-C runtime. By tracing two representative use cases (launching an application and creating a new window), we illustrated how these subsystems interact dynamically, confirming that the design remains robust despite historical evolutions.

Moreover, reflexion analysis reveals that differences between the original conceptual model and the code are frequently driven by performance, cross-platform ambitions, or historical constraints. Instead of contradicting the underlying goals of OpenStep reimplementation, these divergences reflect an architecture that has matured over time. The lessons learned section highlights that bridging conceptual and actual architectures is an iterative process, one that helps maintainers refine or reaffirm structural decisions. With these findings, we are now poised to propose targeted improvements, ensuring GNUstep retains the balance between conceptual elegance and practical adaptability.

References

1. GNUstep Official Docs: https://www.gnustep.org
2. GitHub – gnustep/libs-base: Source code for Foundation classes
3. GitHub – gnustep/libs-gui: Source code for GUI (AppKit) classes
4. GitHub – gnustep/libs-back: Source code for rendering backends
5. GitHub – gnustep/libobjc2: Modern Objective-C runtime for GNUstep
6. Group 4 – Conceptual Architecture of GNUstep (A1 Report, 2025)