Concrete Architecture of GNUstep

Team GNUtered

Logan Philip - 21lfp1@queensu.ca
Alex Nguyen - 22thn@queensu.ca
Ryan Waterson - 22rdw@queensu.ca
William Shaver - 21wgs6@queensu.ca
Nguyen Nguyen - 22nkn@queensu.ca
Gia Nguyen - 21hghn1@queensu.ca

Abstract
In this report, we document our methodical investigation into the concrete architecture of GNUstep, beginning with the unchanged conceptual architecture diagram as the foundation. Building on this baseline, we derive a detailed concrete architecture, complete with an updated diagram, that captures GNUstep's real-world implementation by meticulously tracing source code dependencies using the Understand tool. A comprehensive reflexion analysis then compares this concrete model against our original conceptual design, highlighting both expected and 5 unforeseen interdependencies. In parallel, our subsystem analysis on the Gorm subsystem details the dependencies of the conceptual and concrete architectures within the component. External interfaces are analyzed to illustrate how GNUstep interacts with operating systems and external tools. Finally, two use cases are presented to demonstrate practical scenarios, linking our theoretical findings to real-world applications.

Introduction and Overview
GNUstep is a multifaceted, open-source framework that has grown through the diverse efforts of a global developer community. The original conceptual architecture offers a high-level vision of how GNUstep should ideally operate. In this report, we delve into the concrete architecture to reveal the system's actual structure as implemented in the source code. To achieve this, we used the Understand tool to meticulously trace the dependencies of subsystems, allowing us to uncover both the expected and unexpected dependencies that have accumulated over time. This process not only validates our conceptual model, but also highlights areas where the real system deviates from the original design.

Our analysis begins by looking at the original conceptual diagram, which serves as a reference framework for our investigation. We then detail the derivation process of the concrete architecture, presenting a new, comprehensive diagram that reflects GNUstep's real-world implementation. This diagram illustrates how various subsystems interact, demonstrating that while some dependencies align with our initial expectations, others have emerged due to individual developer contributions and evolving coding practices. A critical part of our work is the "reflexion analysis", where we systematically compare the concrete architecture with the conceptual blueprint. This comparison exposes key divergences, showing why certain dependencies exist and how they impact the overall system.

In addition, our report includes an in-depth subsystem analysis, breaking down an individual component to examine its specific roles and interactions. We explore the architectural style in use and describe how these paradigms are manifested in both the conceptual and concrete models. The discussion extends to external interfaces, where we analyze how GNUstep interacts with underlying operating systems and external tools.

Derivation Process

To create our architecture, we began by installing Understand, setting it up, and learning how to use it through the posted tutorial. We learned about several features that would be very helpful to us when creating our architecture. We were already somewhat familiar with GNUstep's source code, as we had examined it through the Github repositories to create our conceptual architecture in Assignment 1, but we reviewed it further going into Assignment 2. We had an understanding of GNUsteps different layers, being the applications, interface, windowing system, foundation, and operating system layers. By using the source code on Github, as well as our other sources like GNUsteps documentation, we understood the dependencies and interactions between components to create our conceptual architecture.

For our concrete architecture, we used Understand to view the source code of GNUstep. This allowed us to have an easier time looking through it, and see how it's organized. A very useful tool to us was Understands feature that allows you to create a graph showing the dependencies between components. This revealed a lot to us, and showed us more about GNUsteps architecture that we did not previously know. While our conceptual architecture was similar, we found more dependencies between components that we have added to our concrete architecture. In addition, we have added in the component libobjc2 that is in the foundation layer and interacts with many of the existing components.

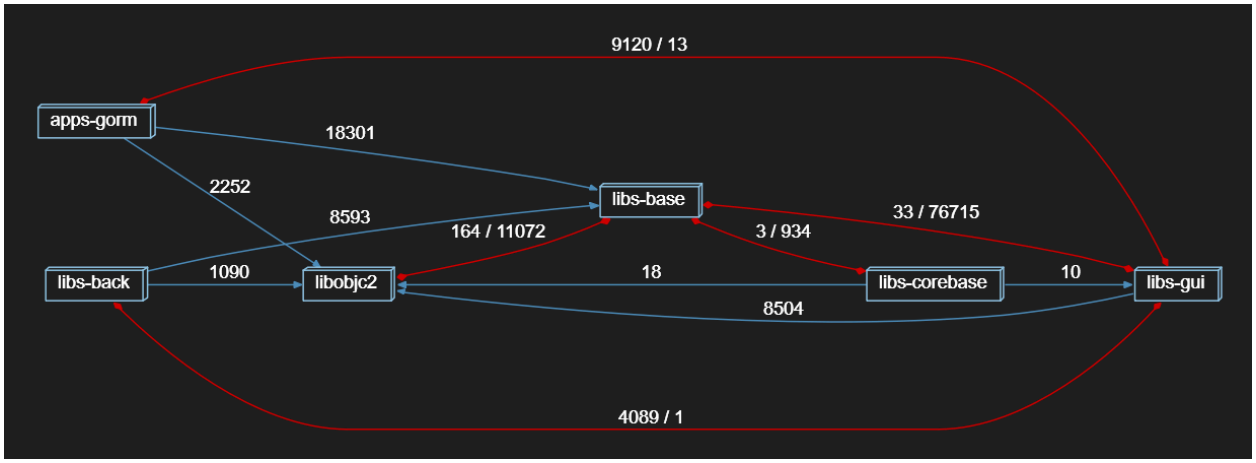

Figure 1. The Understand Architecture of GNUstep.

Conceptual Architecture

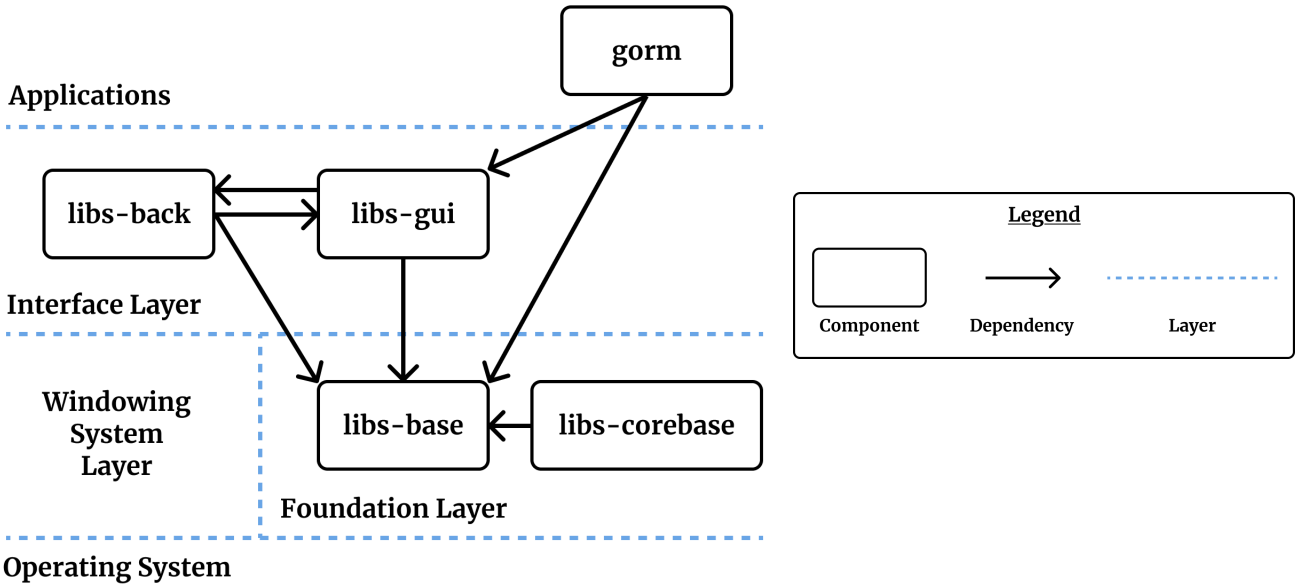

Figure 2. The proposed conceptual architecture of GNUstep from A1.

Concrete Architecture

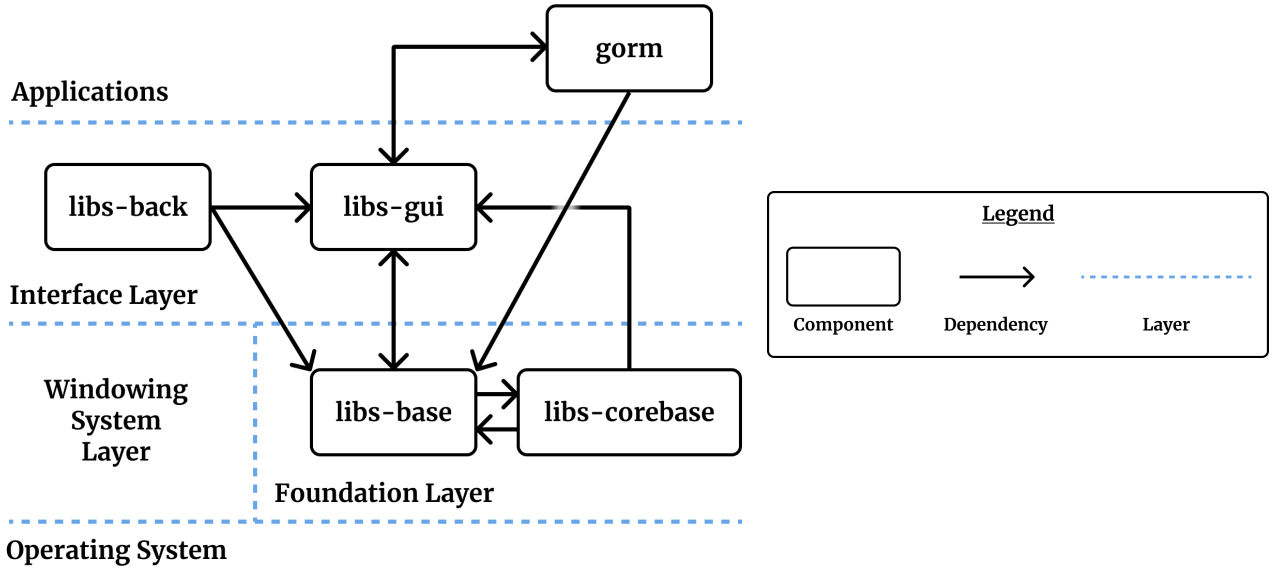

Figure 3. The proposed concrete architecture of GNUstep.

The layered architectural style from our conceptual design largely remains in place in the concrete architecture of GNUstep. However, after a closer examination of the source code and runtime interactions, we realized that GNUstep's components aren't as strictly hierarchical as we initially thought. In particular, the Foundation layer (libs-base) occasionally calls into CoreBase (libs-corebase) for convenience, and CoreBase and the

GUI layer (libs-gui) use callbacks to communicate with higher-level components **(Figure 3)**. We also found that new bidirectional dependencies appear once we consider both compile-time and runtime relationships, leading us to annotate our diagram with additional arrows that were not present in our conceptual design. Below, we detail each component's role in the concrete architecture, focusing on the updated dependencies and the rationale behind them.

Libs-base (Foundation Layer)
In GNUStep, Libs-base component provides non-graphical classes with the goal of mirroring Apple's Foundation framework. It offers core data structures (e.g., NSString, NSArray), concurrency (NSOperation), and other system abstractions (e.g., NSFileManager). As expected, most GNUstep components rely on libs-base for fundamental Objective-C functionality. However, in our concrete analysis, we discovered that libs-base sometimes calls functions from libs-corebase and libs-gui, particularly for tasks such as URL handling and string bridging. Since CoreBase was introduced after libs-base was already mature, we believe that this design choice is intentional as it allows libs-base to reuse existing CoreBase code rather than re-implementing similar features, which improves both efficiency and maintainability.

Libs-corebase (Foundation Layer)
The libs-corebase component is designed to mimic Apple's CoreFoundation in a pure C form, providing fundamental types such as CFString, CFURL, and CFArray. Looking at the design, we thought it was a lower-level library that libs-base would build on. In reality, the libs-base component came first and is the more mature component. As a result, libs-base both calls certain libs-corebase functions and supports Objective-C interoperability through toll-free bridging. However, Libs-corebase is optional, which means projects that don't use CoreFoundation APIs can exclude it. While libs-base can call libs-corebase, this dependency is not required in all configurations, keeping the architecture flexible. We also found that libs-corebase uses callbacks to access the higher-level component libs-gui, and represented this with a new arrow.

libs-gui (Application Kit Layer)
libs-gui implements the AppKit-like functionality of GNUstep, including windows, views, controls (e.g., NSTableView, NSButton), and event handling. As expected, libs-gui depends on libs-base for fundamental data structures and classes. However, we also discovered runtime callbacks from libs-gui to code in higher-level applications. For instance, NSTableView calls the object "ValueForTableColumn:row" on whichever object is set as its data source. Because this method is invoked dynamically at runtime, we represented this with a bidirectional arrow where it made sense, showing that libs-gui and the application communicate both ways. That said, from a compile-time perspective, libs-gui doesn't actually depend on Gorm as it just interacts with any data source that follows the expected structure to fetch table contents.

libs-back (Windowing System Layer)
As expected and as illustrated in our conceptual design, the libs-back component is beneath the libs-gui component and handles low-level interactions with the window

system. At compile time, libs-back implements interfaces from libs-gui, while at runtime, libs-gui calls back-end methods to create windows and render graphics. This two-way interaction validates our original design while showing that the dependency is more flexible than a strict top-down structure.

Gorm (Application Layer)
Gorm is an Interface Builder, like application for designing graphical user interfaces. Initially, we recognized a straightforward compile-time dependency of Gorm on libs-gui and libs-base. However, deeper analysis shows a runtime callback as well. Once Gorm registers itself as the data source or delegate for certain GUI objects (e.g., NSTableView), libs-gui will call methods on Gorm's classes. Consequently, our final diagram shows a bidirectional arrow between Gorm and libs-gui to capture the compile-time requirement (Gorm linking against the GUI) and the runtime method invocations (the GUI querying Gorm for data). This adjustment better represents the actual relationship, which is more dynamic than our original one-way arrow in the conceptual design

Reflexion Analysis

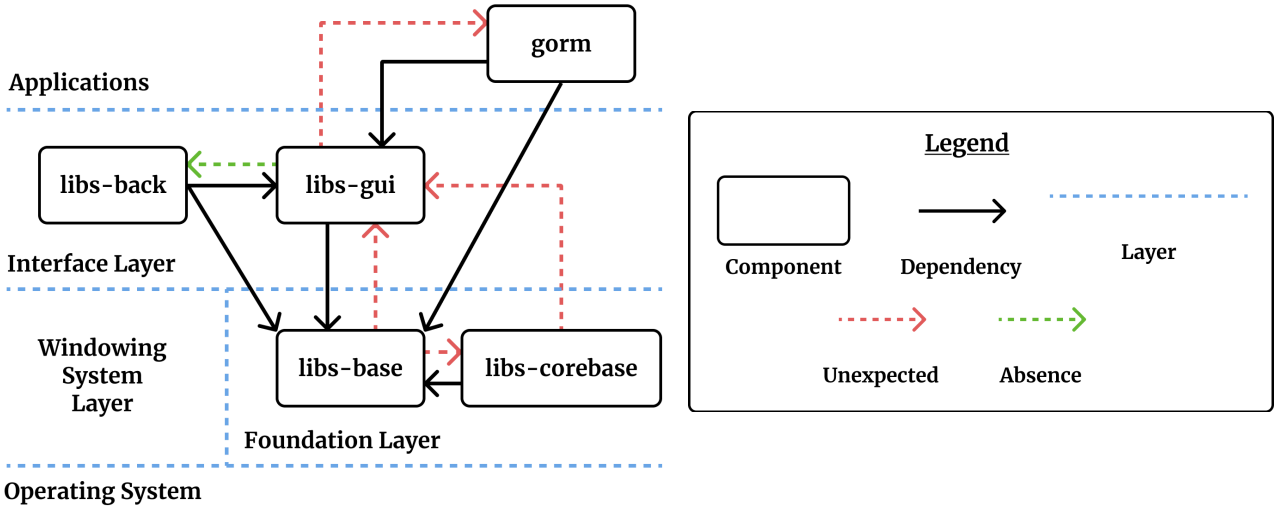

Figure 4. The reflexion analysis for GNUstep.

During our reflexion analysis, we discovered several inconsistencies between the conceptual and concrete architectures. We found that our concrete architecture contains 4 unexpected dependencies, along with 1 absence.

Unexpected Dependencies

libs-corebase → libs-gui: Our concrete architecture discovered an unexpected dependency from libs-corebase to libs-gui. As it turns out, libs-corebase calls upon the config.h file from libs-gui to determine when to execute certain code. For example, GSUnicode.c in libs-corebase only executes strlen() functions if the HAVE_STRING_H flag is set within config.h.

libs-base → libs-gui: We also discovered an unexpected dependency from libs-base to libs-gui. Libs-base, at the very least, uses configuration fields from the libs-gui file NSMenuView.m in order to determine if release or retain operations are needed for items within the libs-base file GSIArray.h.

libs-base → libs-corebase: libs-base only utilizes libs-core briefly. Like stated earlier, libs-base uses libs-corebase for URL handling and string bridging. For example, libs-base contains a function which returns a CoreFoundation URL path for efficiency by calling upon libs-corebase.

libs-gui → gorm: The final unexpected dependency is from libs-gui to gorm. libs-gui, specifically NSToolbar.m, accesses toolbar item identifiers from gorm. For example, NSToolbar.m gets which toolbar items should be selectable by calling "toolbarSelectableItemIdentifiers" from gorm.

Absent Dependencies

libs-gui → libs-back: Our conceptual architecture, which was derived from the GNUstep documentation, determined that libs-gui depended on libs-back in order to ensure that UI and backend worked properly together. As it turns out, however, no such dependency from libs-gui to libs-back actually exists. Any of this functionality that was initially assumed within the conceptual architecture likely takes place by its dependency on libs-base instead.

Gorm Subsystem Analysis

The subsystem we decided to analyze is the Gorm component, otherwise known as the "Graphical Object Relationship Modeller", GNUstep's GUI building application. The conceptual architecture was found by looking at the documentation for Gorm [1], and the concrete architecture was found using Understand. Both the conceptual and concrete use the object-oriented architecture style, but the concrete architecture has 3 unexpected dependencies not present in the conceptual architecture.

Conceptual and Concrete Architecture

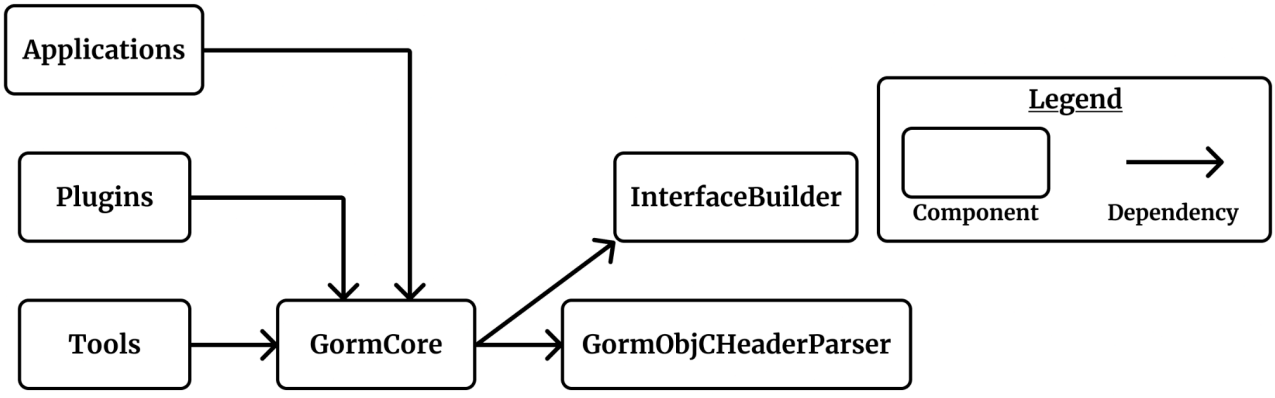

Figure 5. The proposed conceptual architecture for the Gorm subsystem.

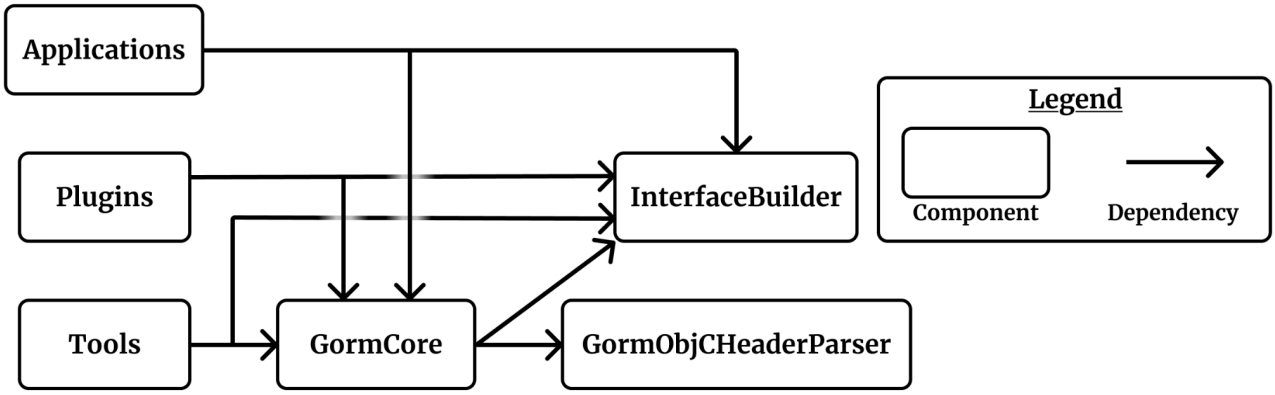

Figure 6. The proposed concrete architecture for the Gorm subsystem.

The architecture of the Gorm subsystem contains six separate components that add functionality to Gorm: GormCore, InterfaceBuilder, GormObjBuilder, Applications, Plugins, and Tools **(Figure 5, 6)**.

GormCore
The GormCore subsystem is the core framework for Gorm. It provides fundamental functionalities and classes needed for other Gorm subsystems. This framework allows the use of Gorm's features within other applications or plugins. GormCore is dependent on the InterfaceBuilder and GormObjCHeaderParser subsystems.

InterfaceBuilder
The InterfaceBuilder subsystem facilitates the creation of custom palettes and inspectors, enabling outside applications an interface for which they can access Gorm. Internally, other subsystems depend on the InterfaceBuilder for the plethora of interface-related types and objects needed throughout Gorm. InterfaceBuilder does not depend on anything within Gorm.

GormObjCHeaderParser
The GormObjCHeaderParser subsystem allows Gorm to parse Objective C information from class header files. This subsystem does not depend on any other subsystems either.

Applications
The Applications subsystem contains the Gorm application itself. Since the Gorm application makes fundamental use of the frameworks and objects from other subsystems, it is dependent on the GormCore and InterfaceBuilder subsystems.

Plugins
The Plugins subsystem holds onto the implementation of various Gorm plugins like 'GModel', 'Nib', and 'Xib'. These plugins cause dependencies from GormCore and InterfaceBuilder.

Tools
The Tools subsystem is the home of the ‘gormtool’ command’s implementation, which enables the use of some Gorm features from the command line. The Tools subsystem, by way of gormtool, is dependent on both GormCore, as well as InterfaceBuilder.

Reflexion Analysis
Performing reflexion analysis on the conceptual and concrete architectures of the Gorm subsystem reveals a few inconsistencies, notably regarding the InterfaceBuilder subcomponent. Based on the documentation, the only subcomponent that depends on InterfaceBuilder is GormCore, but after determining the concrete architecture, it reveals that the following dependencies are missing: **Applications → InterfaceBuilder**, **Plugins → InterfaceBuilder**, and **Tools → InterfaceBuilder**. We found that all three subcomponents rely on InterfaceBuilder in some form or another. Tools and Plugins depend on it in the files AppDelegate.m and GormGormWrapperLoader.m respectively in order to use types defined in InterfaceBuilder. Applications makes use of InterfaceBuilder almost 70 times, notably for the interface building functionalities required by the Gorm application.

External Interfaces
GNUstep has several external interfaces that it uses for cross platform functionality.

Build and Dependency Management:
This looks at the GNUstep make tool, which interfaces with compilers, like GCC or Clang, as well as package managers in order to automate builds. In order for cross-platform compilation to work, it resolves dependencies like libobjc2 and libffi, so that developers can create one GNUmakefile and it will be adapted for Windows DLLs, Linux RPMs, or macOS frameworks.

Graphical user interface:
The important components here are the libs-gui library and gorm, the GUI builder, as they are needed in user interaction. When a developer is designing an app, they can use the drag and drop tools from gorm, which will generate objective-c code for various UI elements such as windows, like NSWindow, and buttons, like NSButton. These components then depend on the libs-back library, which renders the graphics for various platforms like Windows, X11, and Quartz.

Graphics and Rendering Libraries:
When creating complex visuals Cairo and OpenGL are used for vector graphics and 3D acceleration. Examples of these could be NSTestView using Cairo for anti-aliased text, or animations in gorm using OpenGL shaders. These libraries are important for GNUsteps cross platform functionality, as they ensure consistent output across different hardware.

Input Handling:
To handle user inputs, these inputs, such as keystrokes or mouse clicks, are first captured by the native windowing system, such as X11 events for X11, and are then converted to NSEvent objects by the libs-back library. Next, the libs-gui library uses NSResponder chains to process these events.

Operating System Abstraction:
OS-specific tasks are abstracted by the libs-base and libs-corebase libraries, which translates GNUsteps APIs to POSIX calls for linux, Win32 for Windows, or Cocoa for MacOS. It can then be used for file I/O with NSFileManager, memory management with NSAutoreleasePool, and threading with NSThread.

Windowing System Backends:
The libs-back library is used to connect GNUstep to the native windowing service, where it converts NSview rendering commands into operations that can be understood by a specific platform. For X11 it uses Xlib or Cairo to draw UI elements. For Win32 it uses GDI for window management and Direct2D for graphics. For MacOS it uses Quartz for pixel perfect rendering. Through this, an NSWindow declaration can work on any OS.

Use Cases

Use Case 1: Designing an Interface with Gorm

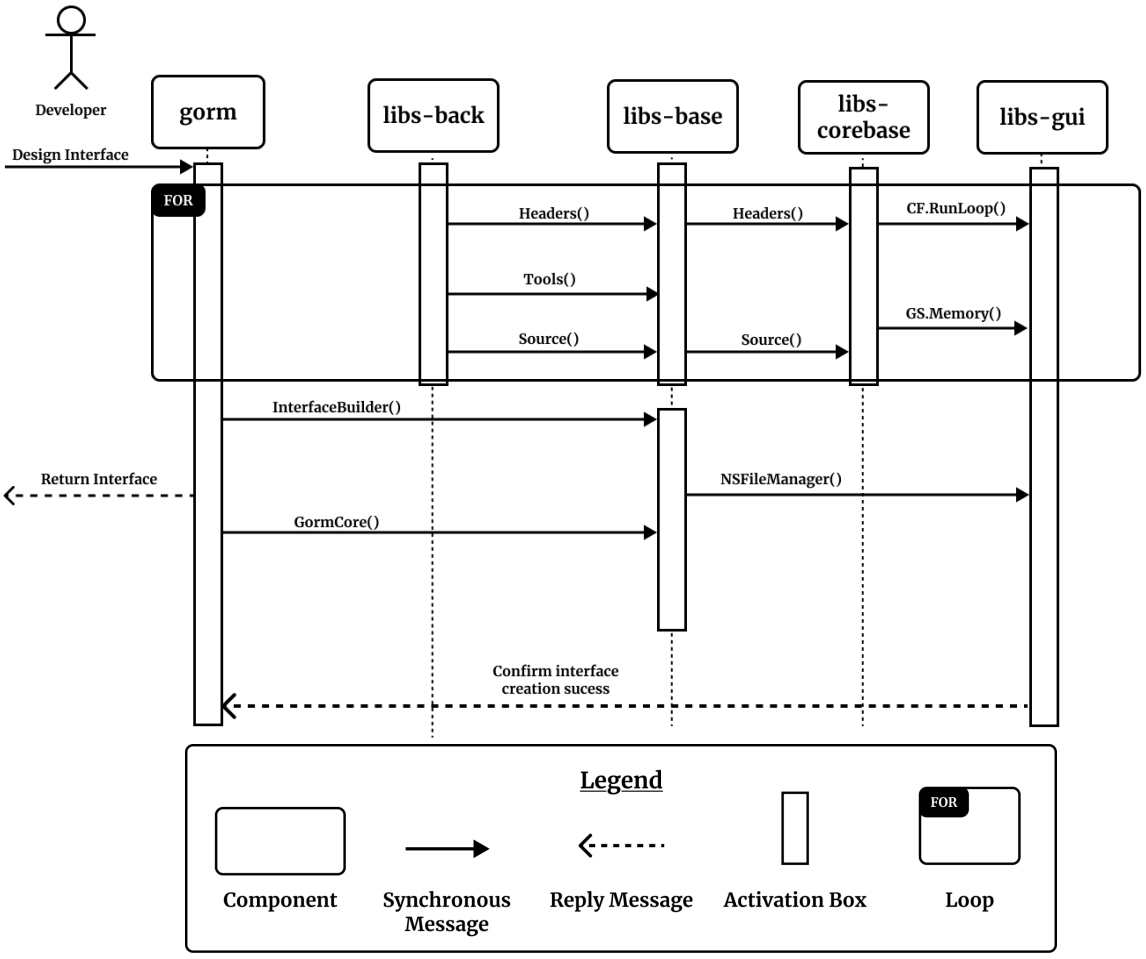

Figure 7. Sequence diagram for the use case of designing an interface using Gorm.

This use case involves designing an interface using Gorm within the GNUstep environment, allowing the developer to visually create and manage UI components **(Figure 7)**. This interaction is crucial since Gorm serves as the interface builder for GNUstep applications, providing a graphical way to define UI elements and their relationships.

The process of designing an interface begins when the developer opens Gorm and initiates a new interface design. The Gorm tool communicates with the libs-back subsystem by requesting Headers(), Tools(), and Source(), enabling access to necessary components. These calls are forwarded to libs-base, which further interacts with libs-corebase to manage runtime and memory functions through CF.RunLoop() and GS.Memory(). Once the required dependencies are in place, Gorm calls InterfaceBuilder(), initiating the UI construction process. The GormCore() function is then triggered to finalize the structure and apply necessary configurations. Upon successful interface creation, Gorm confirms the process, and the developer can proceed with connecting UI components to logic and testing the interface within GNUstep.

Use Case 2: Parsing a Property List with libs-corebase

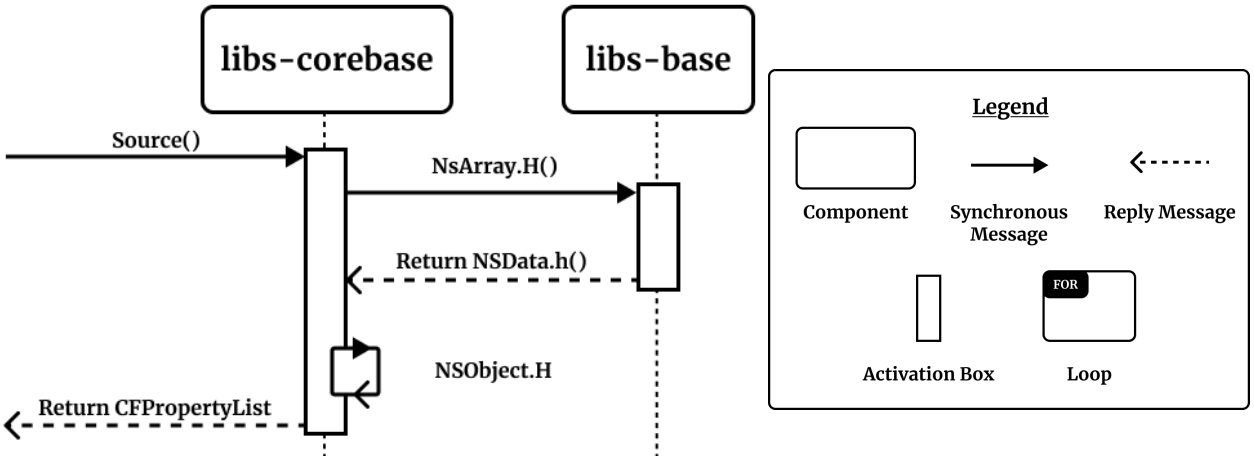

Figure 8. Sequence diagram for the use case of designing an interface using Gorm.

This sequence diagram represents the process of parsing a property list (plist) using libs-corebase in GNUstep **(Figure 8)**. The process begins when libs-corebase calls Source(), initiating the request to retrieve and process the property list. To achieve this, libs-corebase interacts with libs-base by requesting NSArray.h(), which is responsible for handling collections such as arrays and dictionaries within the property list. libs-base then returns NSData.h(), indicating that the plist data is being processed in a raw format before conversion. Additionally, NSObject.h is accessed to manage the parsed objects. Finally, after processing and structuring the data, libs-corebase returns a CFPropertyList, representing the parsed plist in a format compatible with Core Foundation functions. This interaction enables efficient handling of plist files in GNUstep applications.

Concurrency

The use of concurrency in GNUstep is primarily facilitated by the use of the libs-base files NSThread.m and NSLock.m. NSThread handles threading, whereas NSLock handles synchronization of threads by utilizing mutex locks. One such example of concurrency is the file diningPhilosophers.m in libs-base, which tests and demonstrates a practical use of NSLock.

Data Dictionary

Conceptual architecture: A high-level overview of a system's structure that shows how its different parts interact, without describing implementation details. This provides a framework for understanding the overall design and functionality.

Concrete Architecture: The detailed, implemented structure of a software system, outlining how components are built, integrated, and operate. Unlike conceptual architecture, concrete architecture includes implementation details, specific technologies, and coding practices that realize the system's design.

Windowing System: A graphical user interface that manages the layout and display of information on screens. It organizes and controls windows, handling user input and screen rendering.

Foundation Layer: A layer in GNUstep's architecture that supplies essential libraries for data management, ensuring cross-platform consistency and facilitating build automation. It underpins core functionalities such as file I/O, memory management, and collection handling.

Interface Layer: The component of GNUstep that connects user applications to system-level services. It is responsible for rendering graphics and managing user interactions, bridging the gap between underlying system processes and the user's experience.

Applications Layer: The topmost layer that hosts software developed using GNUstep. This layer leverages the capabilities provided by lower layers (like the interface and foundation layers) to deliver fully functional applications.

Quartz: The core graphics and windowing system used in macOS, which is responsible for rendering 2D graphics, managing windows, and applying visual effects. It serves as an example of a platform-specific technology abstracted by GNUstep.

Widget: A small, dynamic element of a user interface that displays information or enables user interaction. Widgets can range from buttons and menus to other interactive controls that form the building blocks of a GUI.

Open source: Software that is made publicly available for anyone to use, modify, and distribute freely. Open-source projects typically foster collaboration and community-driven development, as seen with GNUstep.

Understand (Tool): A software utility designed to analyze and visualize the structure and dependencies of code. It helps developers comprehend complex codebases by mapping out interactions, relationships, and architecture, aiding in maintenance and debugging.

Naming Conventions

GUI: “Graphical user interface”, allows the user to interact with the system without having to rely on text commands.

UI: “User interface”, how a user interacts with a computer.

OS: “Operating system”, an operating system is the software that handles resource management and the hardware for a computer.

Gorm: “Graphical Object Relationship Modeller”, GNUstep’s GUI builder.

CF: “CoreFoundation”, CoreFoundation is a framework to provide software services to applications, and their services and environments. It can also be used for abstracting some data types, along with some other utilities. [2]

GS: “GNUstep”, an open-source, cross-platform framework for desktop application development—the focus of this report.

NS: “NeXTSTEP”, Nextstep is an object-oriented, multitasking operating system that has since been discontinued.

plist: “Property list”, a “uniform and architecture-independent means of organizing, storing, and accessing data for Mac apps”. [3]

libobjc2: A package that enables Objective C runtime. [4]

libffi: A library for foreign function interface. Foreign function interface allows code written in one language to call on code that has been written in another language. [5]

Conclusions

In conclusion, our conceptual architecture was quite similar to what we are now proposing as our concrete architecture. Our concrete architecture keeps a similar layered style to our conceptual architecture, but with less of a strict hierarchy. We found several new dependencies, including some bidirectional dependencies, making the components of GNUstep much more interconnected than we previously thought, which is now shown in our concrete architecture.

We analyzed the Gorm subsystem, which is GNUstep’s GUI builder. Gorm has an object oriented architecture style, which we found in both our conceptual and concrete architecture for it. We did, however, find 4 new dependencies that were not captured in our conceptual architecture, along with 1 dependency absence, when compared to our conceptual architecture. Our use cases have been revised as well, and while they keep a similar structure we now have a deeper understanding of GNUstep. This allowed us to include more detail in our use cases, and more accurately depict how the system works.

Lessons Learned

When analyzing the dependencies using Understand, our group encountered difficulties in interpreting unexpected dependencies within the codebase. Many of these dependencies were not initially accounted for in our conceptual architecture, leading to some confusion and uncertainty about whether we were correctly mapping the structure. Additionally, commit messages were often vague and did not provide enough context to explain why certain files interacted the way they did. Learning to use Understand effectively was also challenging due to its complexity, as it provided a vast amount of information that was sometimes difficult to filter and interpret. One major takeaway from this experience was the importance of clear documentation and meaningful commit messages in large-scale projects to aid future maintainers.

Despite these challenges, our team's communication played a key role in navigating the complexities of this report. By holding weekly meetings and maintaining active discussions over Discord, we ensured that everyone remained aligned on progress and roadblocks. We also improved task distribution by assigning specific sections of the report early on, which allowed us to work more efficiently. This structured approach helped prevent last-minute rushes and ensured that each team member had enough time to research and contribute effectively. Our experience reinforced the importance of proactive communication and delegation in collaborative projects.

One of the key takeaways from this project was the importance of staying on top of course material. Early on, a significant amount of time was spent catching up on past concepts and understanding the software, which delayed our analysis. Balancing this report with other coursework made time management even more challenging. In hindsight, dedicating more effort earlier in the semester to understanding dependencies and familiarizing ourselves with the tools would have made the process much smoother. Moving forward, we recognize the value of pacing our workload more effectively and ensuring that we allocate enough time for thorough analysis rather than leaving it to the last few weeks.

References

[1] “apps-gorm Repository Documentation.” GitHub, https://github.com/gnustep/apps-gorm.

[2] “Core Foundation.” *Apple Developer*, https://developer.apple.com/documentation/corefoundation.

[3] “Property Lists.” *Apply Developer,* https://developer.apple.com/library/archive/documentation/CoreFoundation/Conceptual/CFPropertyLists/CFPropertyLists.html

[4] “libobjc2 Repository.” *Github*, https://github.com/gnustep/libobjc2.

[5] “libffi Repository.” *Github*, https://github.com/libffi/libffi.