Concrete Architecture Report for GNUStep

Group 3: Gamers Never Give Up

Zoya Zarei-Joorshari — 19zzj@queensu.ca

Meagan Mann — 20mmm27@queensu.ca

Jawad Ahmed — 19jaa14@queensu.ca

Kim Hyun Bin — 24vln2@queensu.ca

Ripley Visentin — 22rknv@queensu.ca

Ethan Nguyen — 20hen@queensu.ca

March 24, 2025

Contents

Abstract

This report aims to provide insights into the concrete architecture of GNUStep and to extend the findings of our previous conceptual architecture report. After performing relfexion analysis on the source code of GNUStep, the chosen architectural style has shifted from Objected-Oriented to Layered: this architecture allows the components to be separated into layers which improves the dependency management, modularity, and maintainability of the code. This report will also discuss the discrepancies found between the conceptual and concrete architecture. Further, This report will closely examine the apps-gorm component, providing further insights into a subsystem of GNUStep by revisiting the conceptual architecture and extending it through reflexion analysis of the source code.

1 Introduction

GNUStep is an open source framework created for development of GUI applications. Written in Objective-C, it is a free, object-oriented, cross-platform, development environment. It inherits from NeXTSTEP-like applications and was designed in mind for cross-compatibility between legacy and modern Apple environments. GNUStep has many key components that have their own set of specific responsibilities[9].

In our first report, we aimed to illustrate the conceptual architecture of GNUStep through documentation, online sources and an analysis of the system's overall structure. This second report seeks to illustrate the concrete architecture of GNUStep via a thorough analysis of the system's underlying implementation rather than the idealized conceptual framework previously written. As a result, our analysis involves an examination of the open-source code repository of GNUStep, the dependency structure of the components, and an in-depth exploration of the subsystem *Gorm*. Dependency analysis was conducted using the Understand, an effective tool to isolate subsystems and visualize dependencies and interactions between components within a larger overall software architecture.

With this knowledge of GNUStep's underlying implementation, we performed a reflexion analysis between our conceptual and concrete reports. This includes a change in perspective from the object-oriented architectural style to a layered architectural style. This was redefined based on our new understanding of the organizational hierarchy of layers within GNUStep and the collection of procedure and function calls between them.

In addition, we highlight unexpected dependencies between components in GNUStep and update our use cases/sequence diagrams to match the modified concrete architecture. Towards the end, we end with naming conventions and lessons learned while performing a concrete architecture report.

2 Concrete Architecture

2.1 Derivation Process

Through Understand, we can obtain a better visualization of the core components of GNUStep and the correlations with their respective dependencies. It was similar to how we initially viewed the module relationships, through a conceptual perspective, yet provided a more detailed and intuitive structure. As illustrated in *Figure 1*, the GNUStep module dependencies can be one-way or two-way, indicating whether a module merely depends on another for its functionality or if both modules rely on each other. libs-base, for instance, appears to be in the middle of most interactions, pointing to lower-level libraries like libs-objc2 and referencing higher-level components like libs-gui. Meanwhile, application-level modules like apps-gorm depend on the core libraries for their essential services but also contribute back to the system workflow. As we visualize these dependencies, we observe how GNUStep's modules communicate with one another, validating the final architecture, and also determining tightly coupled or autonomous modules that must be examined for maintainability and extensibility.

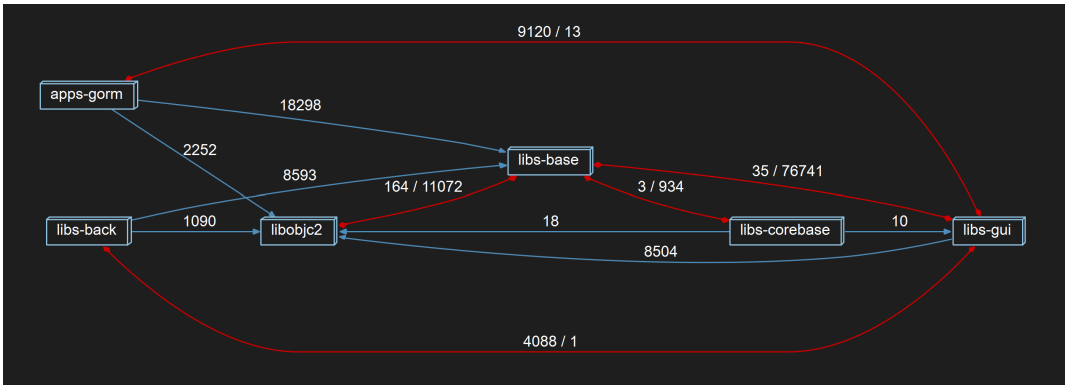

Figure 1: Dependency diagram for GNUStep from Understand

Despite the dependency diagram displaying the number of dependencies between the components, the relation between libs-objc2 and libs-base functions through polymorphic behavior, rather than static dependency. This, more dynamic approach, allows the modules to communicate without strict coupling, demonstrating how the GNUStep design applies polymorphism to give the system more flexibility.

2.2 Architectural Style

The shift from Object-Oriented to Layered Architecture, covered in Section 4, reorganized GNUstep's components into distinct layers with clear separation of concerns. This change improved dependency management, modularity, and maintainability by introducing a more structured approach than the initial Object-Oriented model.

3 apps-gorm

3.1 Conceptual Architecture

Gorm is an application used to design graphical user interfaces for applications [1]. As discussed in our previous report, the components of this subsystem are GormCore, InterfaceBuilder, GormOb- jCHeaderParder, the Gorm application, and gormtool[5]. Within the application there are drag and drop objects (NSWindow objects) and editors (NSMenu objects) attached to them [2]. Editors are responsible for handling the selection of their respective objects, meaning that editors must be aware of the identity of their respective objects [2]: this leads to the conclusion that Gorm uses an Objected-Oriented architecture. Further, Gorm depends on libs-gui to display its UI elements.

3.2 Concrete Architecture

3.2.1 Gorm

This is the main GUI application with which developers interact and is central to the apps-gorm subsystem. It is located within application directory which holds the Gorm application itself and any other apps which might be written using the framework Gorm provides. The directory contains all the necessary files to build and compile Gorm effectively[7]. It also holds the developers palettes. Pallettes allows developers to add their own objects to Gorm. This component of the subsystem is the user-facing interface where developers design UIs[8]. Thus, it can be viewed as the GUI layer managing UI components. It is powered by GormCore in the layer below handling system utilities and lower-level management.

3.2.2 GormCore

GormCore is the core framework that handles the underlying functionality of Gorm. This component holds all of the Objective-C/header files that essentially allow the Gorm application to function. Examples include: *GormViewEditor.m, GormSoundEditor.m, GormClassInspector.m, GormClassEditor.m* and their respective header files. Essentially, it contains all of the classes needed to interact with a Gorm file[7]. This component illustrate "objected-orientedness" with its many Objective-C classes being implemented for modularity and reusability in the context of UI development. With GormCore being such a crucial aspect of GNUStep, it is a reason why we denoted GNUStep as having an objected-oriented architectural style in our conceptual report. This assumption has been modified in Section 4.

3.2.3 GormObjCHeaderParser

GormObjCHeaderParser parses the Objective-C header files to extract things like class definitions, methods, and properties. It is a library that is a basic recursive descent parser that handles ObjC syntax. GormCore requires this component to register and map the varying Objective-C classes within it[7]. GormObjCHeaderParser also allows developers to associate custom-built Objective-C classes in the Gorm application itself.

The communication this component has with the other components of Gorm is in another abstract layer compared to GormCore, as the layers both components exist in have different functions (compilation and object management respectively). These layers provides a substrate for communication.

3.2.4 InterfaceBuilder

InterfaceBuilder was originally an application descended from the NeXTSTEP development software of the same name to build pallette; user-interface objects for GUI development. As a result, the tool is very similar to Gorm. In fact, it has many of the same functionalities and restrictions[10]. In the context of the apps-gorm subsystem, this component is a clone of the InterfaceBuilder framework. Its primary purpose is to allow the creation of custom palettes and inspectors outside of Gorm, essentially providing a compatibility layer that mimics the behavior of the old Interface Builder. This makes it easier to port applications from legacy software.

3.2.5 Plugins

These plugins extends the functionality of the apps-gorm subsystem by adding additional UI components, behaviors, or features. For example, it contains a Nib subdirectory. Nib files are resource files that store the user interfaces of iOS and Mac apps, highlighting the extension of app-gorm's functionality to include newer Objective-C software applications[3].

3.2.6 Tools

This component allows the user to access certain features of Gorm from the command line. It is a "hybrid" tool/headless application. Some tools interact with GormCore to modify UI files programmatically. The Tools component is currently in its experimental phase[7].

3.3 Subsystem Reflexion Analysis

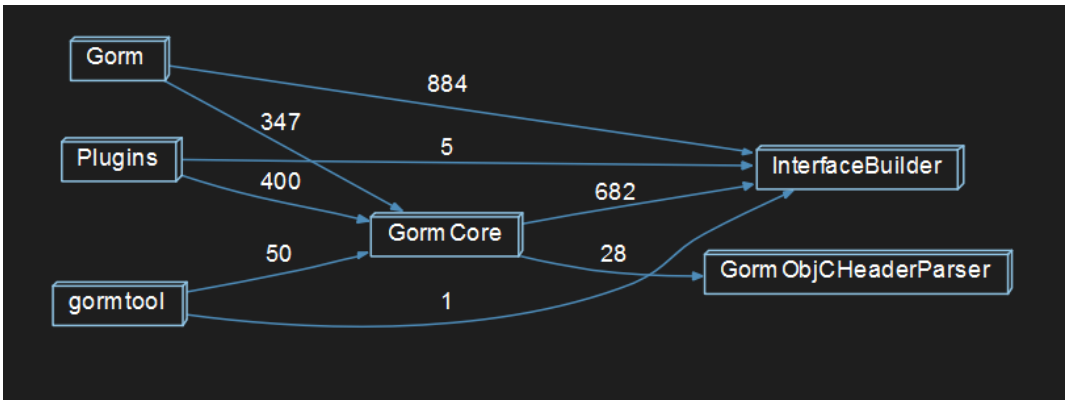

Figure 2: Dependency graph of Gorm from Understand

3.3.1 Gorm → InterfaceBuilder

Since Gorm (within Applications) focuses mostly on the end user's interaction with GNUStep and the Gorm subsystem, Gorm depends on the Interfacebuilder library to provide functionality for cut/paste operations, palettes, document opening

3.3.2 GormCore → InterfaceBuilder

InterfaceBuilder provides GormCore with the framework for resource management, plugin interactions, and element resizing and inspection.

3.3.3 Plugins → GormCore → InterfaceBuilder

InterfaceBuilder necessitates the usage of various plugins by GormCore which allow it to recognize XIB, Nib, Gorm, and GModel Files.

3.3.4 Gorm → GormCore

GormCore handles the logic necessary for many GUI operations such as file and interface element management. Gormcore also manages the data models that are essential in configuring the properties of the UI elements.

3.3.5 GormCore → GormObjCHeaderParser

GormObjCHeaderParser allows GormCore to read information from the required Objective-C classes for GUI design elements and their behaviour.

3.3.6 gormtool → GormCore

GormCore provides gormtool with the framework to import and export classes, as well as read and write to files from the command line. Its definition of file types and metadata associated with Gorm files also enable gormtool to process the various file types used in the Gorm subsystem (Gmodel, XIB, NIB).

4 Updated Architecture

4.1 High-Level Reflexion Analysis

4.1.1 Recap of Conceptual Architecture

Originally, we identified GNUstep's architecture as having an Object-Oriented Style, where multiple autonomous components interact dynamically without strict hierarchical constraints. Each component has specific functionality, invoking methods from other components as needed. This assessment was derived from the codebase, documentation [6], and our initial architectural analysis (Figure 3).

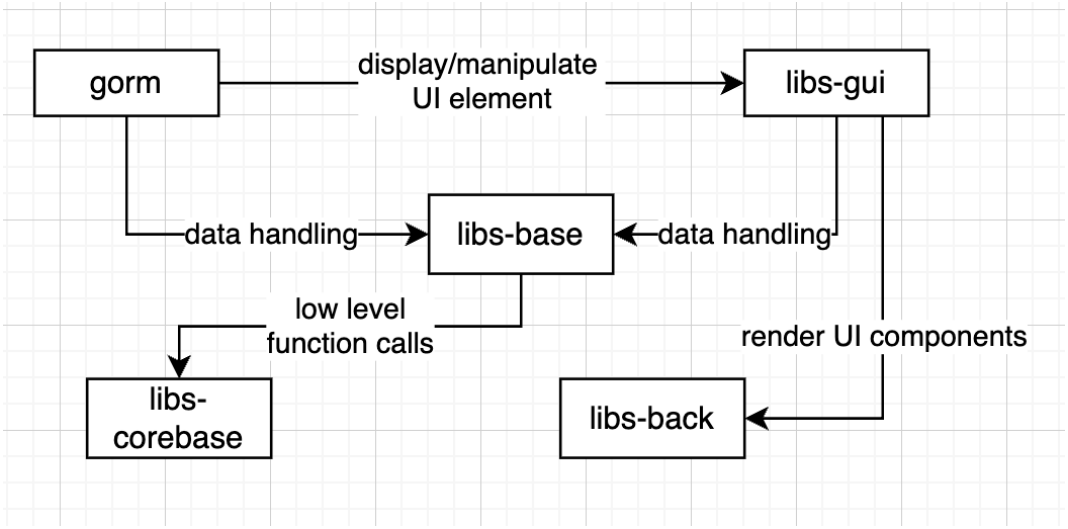

Figure 3: Conceptual architecture for GNUStep from the conceptual architecture report

However, while this Object-Oriented perspective captured the modular nature of GNUstep, it did not fully account for structured dependencies between components. Upon further examination, we found that GNUstep's architecture aligns more closely with a Layered Architecture, which organizes components into distinct hierarchical layers where lower levels provide core functionality for higher layers [4].

4.1.2 Modified Layered Architecture

We now redefine our conceptual understanding based on a Layered Architecture (Figure 4), which introduces a clear separation of concerns to better structure GNUstep's components. In this model, the Graphical User Interface Layer manages UI components and user interactions, including libs-gui and gorm, ensuring a structured approach to rendering and event handling. The Core Runtime and Utility Layer is responsible for core processing and system utilities, with libs-base and libs-corebase handling essential operations and data management. Finally, the System Abstraction Layer oversees low-level rendering and platform-specific operations, with libs-back serving as the bridge between the graphical components and the underlying hardware. This revised conceptual view enforces structured interactions and controlled dependencies, which were previously not explicitly maintained in the original Object-Oriented model.

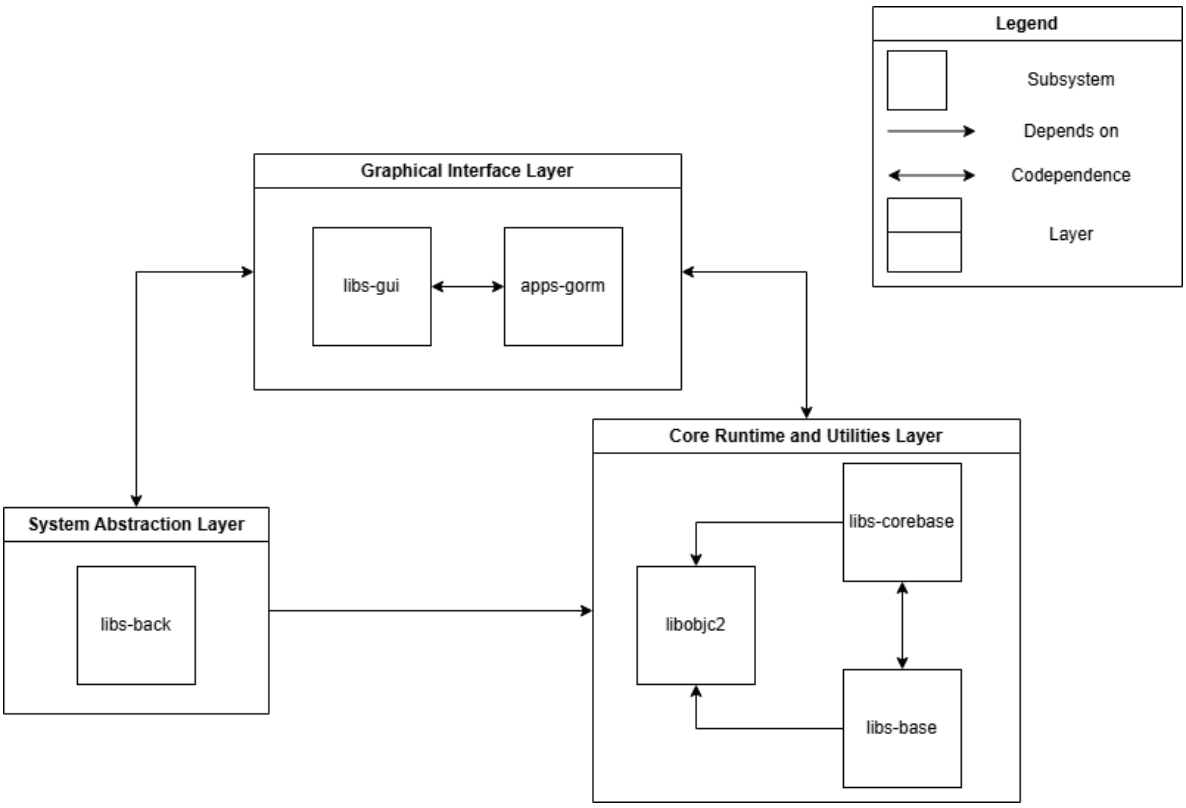

Figure 4: Concrete architecture diagram for GNUStep

The modified architecture introduces several key adjustments to enhance modularity and maintain separation of concerns. One significant change is the enforcement of strict layering, ensuring that lower-level components do not directly depend on higher-level ones. Additionally, the libs-back is now restricted to handling only rendering operations, preventing it from interacting with non-rendering components. Another adjustment involves ensuring that apps-GORM interacts solely with UI components, without affecting core libraries directly. These restructuring efforts help bridge the gap between the Object-Oriented origins and the intended Layered Architecture, reinforcing the system's modularity and improving the overall separation of concerns.

4.1.3 Concrete Architecture Differences

When analyzing GNUstep's actual implementation, we discovered significant deviations from the original Object-Oriented model. As shown in our dependency analysis (Figure 1), we identified several key findings that highlight deviations from the expected layered structure. Some components exhibit direct dependencies that were originally expected to remain isolated, leading to a more interconnected system than anticipated. Additionally, we observed cross-layer interactions, where lower-level components unexpectedly interact with higher-level ones. External dependencies, such as libobjc2, introduce additional complexity that was not initially accounted for in the conceptual model. See section 4.2 for a detailed description of unexpected dependencies.

4.2 Unexpected Dependencies

4.2.1 All GNUStep Components → libobjc2

libobjc2 is a libarary that provides custom Objective-C utilties and tools for the purposes of GNUStep. These tools are used by the other GNUStep components to perform their tasks.

4.2.2 libs-base → libs-gui

libs-base uses libs-gui for threading purposes. Further libs-base uses GSIArray objects (a custom array implementation for GNUStep) in which lib-gui defines functions for: this seems to be done for optimization purposes.

4.2.3 libs-corebase → libs-base

libs-corebase provides added behavior to the to the classes defined in libs-base. It extends higher-level functionality to the more basic classes of libs-base. It also provides more datatypes based on datatypes defined in libs-base.

4.2.4 libs-corebase → libs-gui

libs-corebase uses libs-gui to check strings. It checks to see if there are strings or if strings contain unicode characters: this allows libs-corebase to import functionality or perform string operations as needed.

4.2.5 libs-back → libs-gui

libs-back is the the connector for front end (libs-gui) to the platform-specific backend display mechanism . It uses classes and their associated data from libs-gui and translates them into rendering instructions based on the platform. This dependency facilitates the portability of GNUStep.

4.2.6 libs-back → libs-base

libs-back depends on libs-base for necessary functionality and data types. It depends on libs-base core functionalities, such as memory management and event handling. Further, libs-back also relies on the basic classes defined in libs-base to properly process and translate data from libs-gui.

4.2.7 libs-gui → apps-gorm

apps-gorm provides libs-gui with the ability to generate and keep track of rows and columns. Further, apps-gorm allows libs-gui to manage the toolbar within the Gorm application.

5 Sequence Diagrams

5.1 Use Case 1: Modernizing Legacy OpenStep Applications

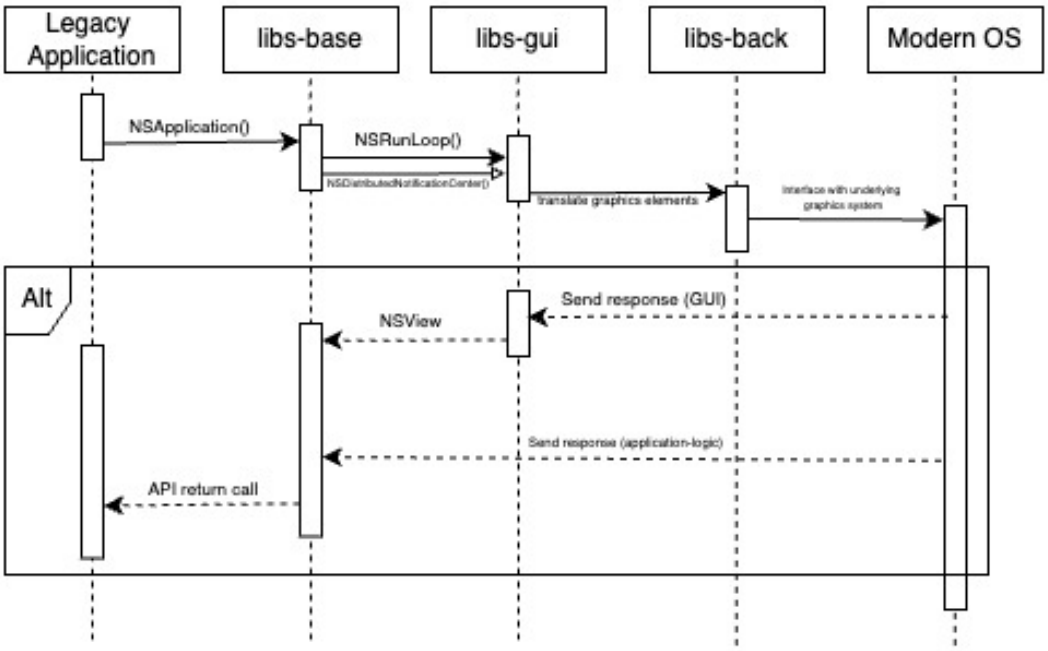

Figure 5: Sequence diagram for use case modernizing legacy OpenStep applications

This illustrates the modernization process of legacy OpenStep applications by integrating them with a modern operating system. It depicts the interaction between different components, including the legacy application, various libraries (libs-base, libs-gui, libs-back), and the modern OS. The sequence begins with the legacy application invoking functions in the NSApplication class which initiates the main event loop. The NSRunLoop() function in libs-base is initiatied to process input sources. This is proceeded by further processing, including distributing notifications and translating graphical elements in libs-gui via the distributed notification center. This distributed notification center is said to provides a versatile yet simple mechanism for objects in different processes to communicate effectively while knowing very little about each others' internals. The libs-back layer interfaces with the underlying graphics system of the modern OS. This structured approach ensures that legacy OpenStep applications can operate efficiently on contemporary systems while maintaining compatibility with modern graphical and system architectures.

5.2 Use Case 2: Cross-Platform Development With Objective-C

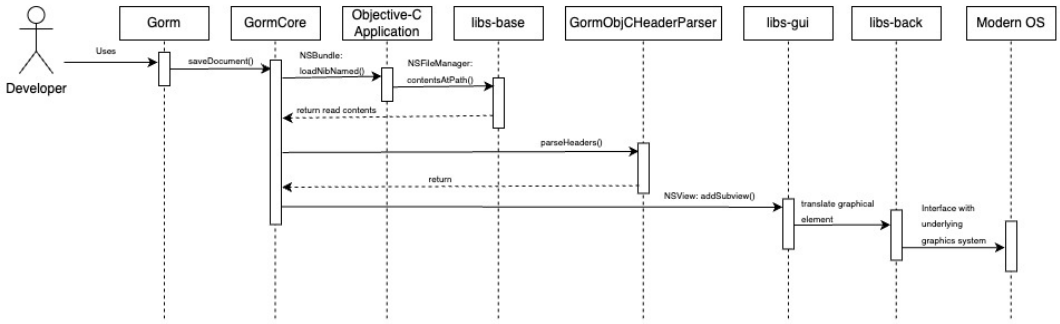

Figure 6: Sequence diagram for use case cross-platform development with Objective- C

This diagram illustrates the cross-platform development workflow using Objective-C, showcasing interactions between various components. The process begins with a developer using Gorm, a graphical interface builder, to save a document. GormCore then loads the necessary Nib files using NSBundle: loadNibNamed(), retrieving their contents. The Objective-C application processes file paths via NSFileManager: contentsAtPath(), returning the relevant data. The GormObjCHeaderParser component parses the headers before functions such as NSView: addSubview() in libs-gui translates graphical elements. Finally, libs-back interfaces with the modern operating system's graphics system to render the UI. This sequence ensures compatibility across platforms, allowing Objective-C applications to integrate seamlessly with modern systems while leveraging Gorm for UI development.

6 Lesson Learned

1. **Conceptual vs Concrete Architecture**
   GNUstep's conceptual architecture is its ideal design, built around modularity and object-oriented principles. However, when implemented, practical constraints such as performance considerations and real-world dependencies require deviations from this ideal. The shift from an Object-Oriented Architecture to a Layered Architecture shows this evolution. While the conceptual model emphasizes reusability, the concrete architecture introduces structured layers to better manage dependencies and improve maintainability. This transition highlights the challenge of balancing theoretical design goals with the realities of software engineering.

2. **Importance of Dependency Analysis**
   Using dependency analysis tools revealed unexpected cross-layer dependencies, emphasizing the value of automated analysis in understanding complex system interactions.

3. **Documenting Architectural Decisions**
   Keeping detailed records of both the intended design and the actual implementation using reflexion analysis helped uncover inconsistencies between the two. This process made it easier to understand deviations, refine the architecture, and guide future improvements, ultimately ensuring a more maintainable and scalable system.

7 Conclusion

This report examines the Concrete Architecture of GNUStep. The architecture was established with the Understand tool to analyze the dependencies between the various subsystems and components of the system architecture. We were able to identify convergences, divergences, and absences through reflexion analysis. While we decided upon an object-oriented style in our conceptual report from A1, we ultimately adopted a layered style. A thorough examination of the apps-gorm subsystem, supported by updated sequence diagrams that reflect the flow of function calls and interactions, further validated the merits of this layered structure.

References

[1] GNUstep Developer Tools: Gorm.

[2] Guide to the gorm application.

[3] Apple Developer Documentation. Nib file, 2024. Accessed: 2025-03-11.

[4] P.U. Chavan, M. Murugan, and P.P. Chavan. A review on software architecture styles with layered robotic software architecture. In *2015 International Conference on Computing Communication Control and Automation*, pages 827–831, 2015.

[5] Gnustep. Gnustep/apps-gorm: Gorm is a clone of the cocoa (openstep/nextstep) 'interface builder' application for GNUstep.

[6] GNUstep contributors. GNUstep Information, 2025. Accessed: 2025-03-12.

[7] GNUstep Developers. Gorm - gnustep interface builder repository, 2024. Accessed: 2025-03-11.

[8] GNUstep Project. *Gorm: GNUstep's Interface Builder*, 2024. Accessed: 2025-03-11.

[9] Wikipedia contributors. GNUstep, 2025. Accessed: 2025-03-12.

[10] Wikipedia contributors. Interface builder, 2025. Accessed: 2025-03-11.