CISC 322/326 - Winter 2025

GNUstep Concrete Architecture Report

March 14th, 2025

Infinite Loops - Authors

Ray Huezo	21tch5@queensu.ca
Anneth Sivakumar	21as221@queensu.ca
Vedsai Pandla	21vp23@queensu.ca
Rachel Narda	22rn3@queensu.ca
Jonathan Thompson	20jmt1@queensu.ca
Princess Viernes	20pejv@queensu.ca

Table of Contents

1.0 Abstract

This report examines the concrete architecture of GNUstep, an open-source framework that offers a cross-platform development environment for Objective-C applications. GNUstep supports the OpenStep specification, ensuring compatibility with Apple's Cocoa API and facilitating the development of graphical and server-side applications. This report builds on our conceptual analysis by examining the as-built system, which is derived from source code dependencies and architectural visualization tools. The derivation process involved using Understand, a software visualization tool, to extract system dependencies and classify GNUstep's components into hierarchical subsystems. A detailed examination of the apps-gorm subsystem was conducted, focusing on its components, interactions, and architectural styles. Additionally, a reflexion analysis was performed to identify differences between the conceptual and concrete architectures, highlighting any unexpected dependencies and structural variations. The study delves deeper into GNUstep's external interfaces, including its API, graphical interface, command-line tools, and system integration mechanisms, which support cross-platform functionality and modular development. The findings reveal that GNUstep maintains a highly modular and object-oriented design that closely aligns with the original architectural purpose while including additional dependencies, such as libobjc2, for runtime support. By comparing the conceptual and concrete architectures, this report provides a comprehensive evaluation of GNUstep's design, offering insights into its implementation, architectural trade-offs, and areas for refinement.

2.0 Introduction & Overview

GNUstep is an open-source framework that provides a cross-platform development environment for Objective-C applications [1]. It implements the OpenStep specification, allowing compatibility with Apple's Cocoa API and enabling the development of portable graphical and server-side applications [2] . While Assignment 1 examined the conceptual architecture of GNUstep, this report focuses on its concrete architecture—the as-built system as extracted from its source code and dependencies.

The objective of this report is to derive the actual structure of GNUstep by analyzing its dependencies and system organization. To accomplish this, we leveraged the extracted dependency file and the software visualization tool, Understand, to examine the overall organization of GNUstep, as well as, the physical components within each subsystem. We were able to analyze the internal processes and interactions between objects from different subsystems to develop the top-level concrete structure of GNUstep. This process allows us to compare the actual implementation with the previously defined conceptual architecture.

Beyond presenting the top-level architecture, this report delves into the internal organization of apps-gorm, a main subsystem within GNUstep. We analyzed apps-gorm's objects, interactions, and design and composed a concrete architecture of the subsystem. Furthermore, we performed a reflexion analysis to identify any discrepancies between the conceptual and concrete architectures of apps-gorm, while investigating the underlying causes of these divergences.

The architectural documentation process involved not only dependency extraction but also manual verification through source code analysis and architectural mapping. Through this, we

highlight the major control flows, interdependencies, and architectural styles employed within GNUstep. This report also includes concrete architecture diagrams that have been refined to improve readability, ensuring that the system's structure is presented clearly and logically.

By comparing GNUstep's conceptual architecture with its concrete implementation, this report aims to provide a comprehensive view of the system's actual design. The findings offer insights into how well GNUstep aligns with its intended design goals and where modifications in conceptual understanding might be necessary to reflect the real-world implementation accurately. This comparison ultimately enhances our understanding of GNUstep's design principles and their practical application.

3.0 Derivation Process

In order to derive the concrete architecture for GNUstep we used the Understand tool to visualize the dependencies among subsystems. We compared the actual implementation to our revised conceptual architecture from Assignment 1. This allowed us to identify any subsystems and/or dependencies that were overlooked in our previous report.

To derive the concrete architecture of a second-level subsystem, we first needed to decide which subsystem to focus on. We picked the apps-gorm subsystem due to its unique components, as many subsystems in GNUstep share a similar component structure. We began by examining the implementation of apps-gorm to gain an understanding of the various components and interactions. From there, a conceptual architecture was formed and compared with the concrete architecture shown by the Understand tool.

4.0 Concrete Architecture

4.1 Conceptual Architecture Review

In our initial report on the conceptual architecture of GNUstep, we believed that the system followed a layered, object-oriented style with certain aspects of the publish-subscribe style. However, after receiving feedback on the report, we have updated our previous architectural style and concluded that GNUstep follows solely an object-oriented architectural style.

The conceptual architecture of GNUstep in Figure 1.0 consisted of 5 main subsystems: apps-gorm, libs-back, libs-gui, libs-corebase, and libs-base. Each subsystem is composed of objects that interact with other objects through method invocation. [1]. An object can independently function on its own when requested by other objects, but also consist of methods that act on instance variables (data) in response to messages [1]. The system also offers distributed objects that allow communication with other objects between different processes running on the same machine or on different machines on the same network [5].

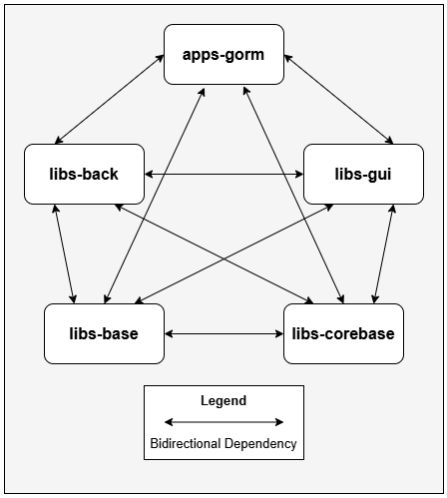


Figure 1.0: *Updated Conceptual Architecture*

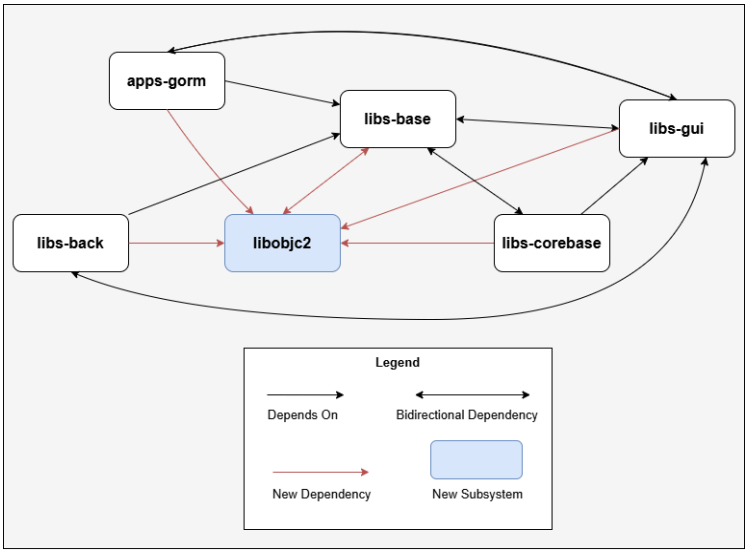


Figure 1.1: *Concrete Architecture*

4.2 Concrete Architecture

We have examined the physical system of GNUstep and its contents using Understand to derive a concrete diagram of the system presented as Figure 1.1.

Within the main directory, there are six subsystems: apps-gorm, libobjc2, libs-back, libs-base, libs-corebase, and libs-gui. Each subsystem contains folders and/or files that depend on files from other subsystems. The concrete architecture is consistent with the object-oriented architectural style derived from the conceptual architecture along with the following subsystems.

apps-gorm: A visual GUI builder that provides an interface for developers to create applications using a drag-and-drop feature [4]. Gorm mainly depends on libs-gui to generate and manage UI components. apps-gorm also depends on libs-base, specifically on classes within the Headers, Source, and config folders for fundamental objects/classes and internal processing.

libs-gui: A front-end component for libs-back that offers classes that include graphical objects such as buttons, text fields, etc. and support for handling user events [2]. Source objects depend on objects in the Applications and GormCore folders from apps-gorm to control a display of the UI components. A file in the Source folder also depends on libs-back to initialize the backend for the application. libs-gui also depends on libs-base, specifically on classes within the Headers, Source, and config folders for fundamental objects/classes and internal processing.

libs-back: A back-end component for libs-gui and converts high-level UI commands into system-specific rendering instructions [3]. The subsystem works for platforms using the X-Window System or Window's Systems [6]. libs-back also depends on the objects from the Headers, Source, config, and MacOSfolders located within libs-base for fundamental objects/classes and internal processing.

libs-base: This library consists of general-purpose, non-graphical Objective C objects and implements the API of the OpenStep Foundation Kit for data management, network and file interaction, and more [7]. It includes classes for fundamental data structures like strings, invocations, notifications, notification dispatchers, distributed objects, and more [8]. There are objects in the Headers and Source folders that depend on objects from libs-gui and libs-corebase for internal processing that correspond with the Foundation Kit implementation.

libs-corebase: This library works similarly and close to libs-base where it provides general-purpose, non-graphical C objects and implements the API of Apple's Core Foundation framework [9]. libs-corebase provides abstractions to data types, supports internationalization, and provides utilities property lists, run loops, and more [9]. It primarily depends on libs-base to create abstractions of the same objects in libs-base. Some objects also depend on the config object from libs-gui for internal processing.

The concrete architecture also introduces a new subsystem, libobjc2. In the reflection analysis below, we provide a more in-depth analysis of libobjc2, along with the removed and added dependencies between the conceptual design and the concrete design of GNUstep.

5.0 Reflexion Analysis

5.1 New Subsystem

In the conceptual architecture, the absence of a subsystem was found. This new subsystem is libobjc2. libobjc2, often referred to as the "GNUstep runtime," is an Objective-C runtime library that underpins modern Objective-C features on non-Apple platforms. It handles the core mechanics of the programming language, which includes dynamic method dispatching (objc_msgSend), object creation, and memory management. By providing support for language additions such as Automatic Reference Counting (ARC), blocks (closures), and property introspection, libobjc2 enables GNUstep to stay compatible with Apple's evolving Objective-C ecosystem while running on operating systems like Linux, BSD, and Windows.

Within GNUstep's architecture, libobjc2 typically sits beneath higher-level libraries like libs-base and libs-gui. While most developers interact with classes and frameworks provided by these higher layers, libobjc2 is crucial behind the scenes: it enforces the object-oriented model, handles memory management, and facilitates modern Objective-C features. Its tight coupling with Clang ensures that advanced language constructs, such as ARC or blocks, are correctly compiled and executed. From a layered perspective, one might view libobjc2 as part of the runtime and toolchain layer, invisible to application developers but essential for GNUstep's full functionality.

5.2 Removed and Unexpected Dependencies

There were critical differences between the conceptual architecture of GNUstep and its concrete implementation, underscoring how theoretical design assumptions diverge from practical system realities. These discrepancies arise from runtime necessities, interoperability requirements, and the inherent complexity of maintaining cross-platform compatibility.

In the conceptual architecture, libs-corebase (Core Foundation compatibility layer), was assumed to directly depend on libs-gui (graphical frontend) for foundational data type support. However, the concrete implementation revealed no such direct dependency. Instead, libs-corebase interacts exclusively with libs-base (Foundation Kit), which acts as an intermediary to fulfill GUI-related data needs. This structural adjustment reflects a deliberate separation of concerns, ensuring libs-corebase remains decoupled from graphical subsystems to preserve its role as a platform-agnostic compatibility layer. By delegating GUI-specific tasks to libs-base, the architecture enhances modularity, allowing Core Foundation utilities to function independently of rendering logic.

The conceptual model overlooked the libobjc2 subsystem, which surfaced as a pivotal component in the concrete architecture. libobjc2 provides the Objective-C runtime, handling memory management (e.g., Automatic Reference Counting), dynamic method dispatch (objc_msgSend), and modern language features like blocks. This subsystem underpins libs-base and libs-gui, enabling them to leverage advanced Objective-C capabilities. Its omission in the conceptual architecture highlights a gap in anticipating the runtime's foundational role, particularly in bridging high-level abstractions with low-language mechanics. The dependency of libs-base on libs-objc2 underscores the runtime's necessity for enforcing object lifecycle rules and message-passing semantics, which are critical for GNUstep's alignment with Apple's Objective-C framework.

The conceptual model posited a unidirectional relationship where libs-corebase depended on libs-base. However, the concrete architecture revealed a bidirectional coupling. For instance, NSRunLoop (from libs-base) relies on CFRunLoop (from libs-corebase) for event loop management, while CFDictionary (Core Foundation) is internally used by NSDictionary (Foundation) for performance-critical operations. This mutual dependency ensures seamless interoperability between Apple's Core Foundation and GNUstep's Foundation Kit, enabling "toll-free bridging" where Core Foundation types can be interchangeably used with their Foundation counterparts. While this tight integration enhances functionality, it introduces challenges in modular maintenance, as changes to one subsystem risk unintended impacts on the other.

The rendering backend (libs-back) was initially assumed to operate independently of Core Foundation utilities. However, the concrete analysis showed that libs-back depends on libs-corebase for data buffer management. For example, the art backend uses CFDataRef to handle pixel buffers, leveraging libs-corebase's memory-efficient abstractions. This dependency was not anticipated in the conceptual model but is justified by the need for high-performance data handling in rendering pipelines. By utilizing libs-corebase, libs-back avoids reinventing low-level data structures, ensuring consistency with GNUstep's broader data management strategies.

6.0 Concrete Architecture of *apps-gorm*

The subsystem apps-gorm contains Gorm, which stands for Graphical Object Relationship modeler. apps-gorm consists of libraries and frameworks such as GormCore, InterfaceBuilder, and GormObjCHeaderParser as well as additional directories like Application, Tool, Plugins, and Documentation. The Application directory holds the Gorm application as well as any other apps written using the framework provided by Gorm. The Tool directory holds gormtool, which is used to allow users access to certain Gorm features from the command line. The Plugins directory contains Gorm extensions. The Documentation directory contains the documentation for the Gorm application

and all frameworks used in its functionality. The GormCore framework contains all the classes needed to interact with a Gorm file. This framework is used by the application, as well as gormtool. The Plugins directory also uses this framework as it provides a way for other applications to use Gorm features indirectly. GormObjCHeaderParser is a library that operates as a basic recursive descent parser. It handles Objective-C syntax so that Gorm can gain information about a class from its header file. By being a library, other applications and tools can use it. The InterfaceBuilder framework is a clone of Apple's Cocoa Interface Builder framework. It allows the creation of custom palettes and inspectors from outside of Gorm [10].

Below are the diagrams showing the conceptual architecture and concrete architecture of apps-gorm. The conceptual diagram was created by reviewing the documentation for each component to gain an understanding of how the components interacted. The concrete diagram is a reconfigured version of the diagram provided by Understand.

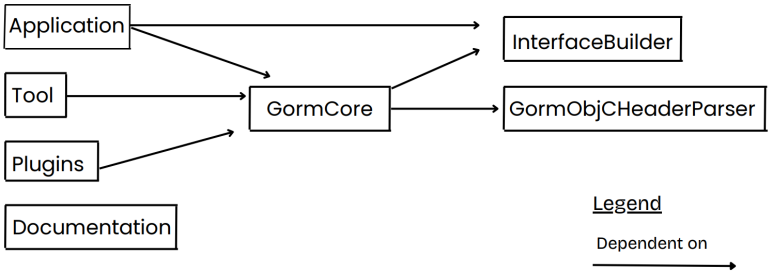


Figure 2.0: *Conceptual architecture of the subsystem apps-gorm*

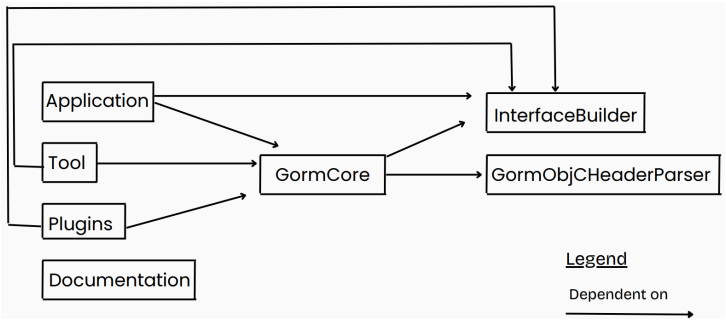


Figure 2.1: *Concrete architecture of the subsystem apps-gorm*

As seen in the figures above, there were two new dependencies not depicted in the conceptual architecture. The Tool and Plugins directories both depend on the InterfaceBuilder framework. This is due to the framework being essential for Gorm's functionality [10]. The Tool and Plugins directories both hold applications that apply to, and/or use Gorm, which means they will depend on the InterfaceBuilder for their functionality as well.

7.0 External Interface

GNUstep includes a number of external interfaces that make it easier to interact with its core libraries, development tools, and external systems. These interfaces provide communication between components, allowing applications to execute smoothly across various platforms. The primary

external interfaces include the Application Programming Interface (API), graphical user interface (GUI), command-line tools, system integration mechanisms, and development utilities.

The GNUstep framework adheres to the OpenStep specification, which ensures compatibility with Objective-C-based software development [2]. The API is divided into several core libraries, including the GNUstep Base library, which contains essential Objective-C classes for data structures, string manipulation, threading, and system interaction [8]. The GNUstep GUI package includes widgets, windows, buttons, and menus that allow developers to create graphical user interfaces [2]. Additionally, the GNUstep Back library acts as an abstraction layer for producing graphics on a variety of systems, such as X11 and Windows GDI [11]. The GNUstep CoreBase library integrates with Apple's Core Foundation framework, allowing macOS interoperability [12]. These libraries work together to provide a well-defined API that enables developers to create cross-platform applications while maintaining compatibility with Cocoa.

GNUstep's external interfaces are closely tied to its concrete architecture. The apps-gorm subsystem, which provides an interface for graphical user interface design, is built using the GUI and Base libraries [4]. It relies on the GUI library to render user interface elements and the Base library to manage essential system interactions [13]. The structure of apps-gorm reflects the object-oriented design principles observed in GNUstep's architecture, where modular components interact via method calls and shared data structures [4]. Additionally, apps-gorm works with the GormCore framework, which provides methods for interacting with interface files, allowing applications to be developed dynamically using established UI layouts [4].

In addition to GUI-based tools, GNUstep includes a suite of command-line utilities for development, debugging, and deployment [2]. The gnustep-make tool acts as a build system, simplifying the compilation and installation of GNUstep applications [14]. ProjectCenter serves as an integrated development environment (IDE) for managing Objective-C projects [15]. Furthermore, various command-line interfaces are available for managing application configurations, dependencies, and debugging information, providing developers with a comprehensive set of tools for efficient program development.

GNUstep is designed to operate seamlessly on different platforms, ensuring compatibility with a variety of operating systems. System integration mechanisms include a well-structured filesystem hierarchy that categorizes system files into predetermined domains such as System, Local, Network, and User, ensuring a logical and efficient arrangement. The libs-back subsystem provides abstraction layers for rendering user interfaces on different platforms, allowing applications to adapt to the underlying operating system without requiring significant modifications [6]. The framework also ensures cross-platform compatibility, allowing applications to run on Linux, BSD, Windows, and other Unix-like environments without requiring significant modifications. Additionally, GNUstep supports inter-process communication (IPC) via NSDistributedNotificationCenter, which enables processes to exchange messages efficiently, while ensuring seamless data flow between system components [2].

To further enhance software creation and maintenance, GNUstep provides several development tools. Gorm's Interface Builder is a graphical tool for designing application interfaces, which helps to streamline the development process [4]. The framework also includes code compilation and debugging tools, integrating with compilers such as Clang and GCC to build

Objective-C applications [16]. Additionally, GNUstep also simplifies project management with version control and dependency management tools, allowing developers to track changes and manage libraries more easily.

GNUstep provides development utilities that improve the software engineering process. Gorm's Interface Builder enables developers to design and manage application interfaces visually, reducing the need for extensive manual coding [4]. The framework also contains debugging and profiling tools to ensure that applications are performance-optimized [4]. By leveraging these utilities, developers can build and maintain GNUstep applications efficiently while adhering to best architectural practices.

By integrating its external interfaces—ranging from the API and graphical tools to command-line utilities and system integration mechanisms—GNUstep maintains a robust and versatile development environment. Its structured filesystem, support for inter-process communication, and platform-independent UI rendering through libs-back ensure seamless cross-platform operation. Development tools such as gnustep-make, ProjectCenter, and Gorm's Interface Builder streamline software creation, debugging, and project management. The reliance on object-oriented principles, modularity, and adherence to standard APIs ensures that applications built with GNUstep remain scalable, efficient, and well-integrated across diverse operating systems.

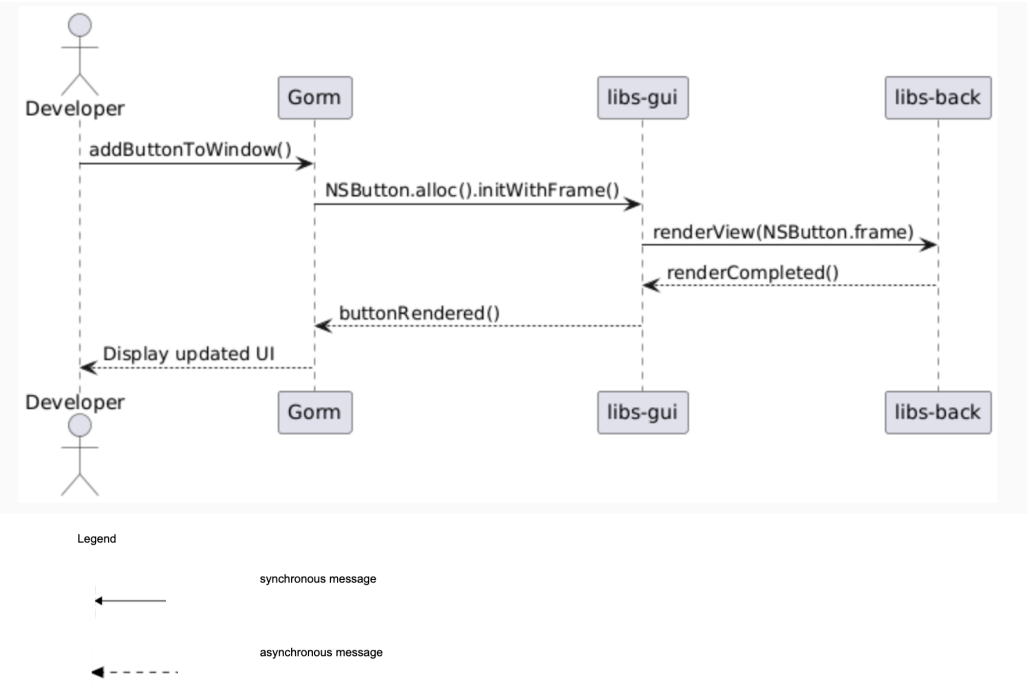
8.0 Use Cases

8.1 Use Case #1: Designing a GUI Window in Gorm

In this use case, a developer uses Gorm, GNUstep's graphical interface builder, to design a window containing a button. The process begins when the developer drags a button from Gorm's component palette onto a blank window canvas. Gorm, as part of the apps-gorm subsystem, responds by instantiating a NSButton object through the libs-gui subsystem, which provides the graphical frontend components. The NSButton is initialized with properties such as its frame dimensions, title, and target-action pair.

Once the button is configured, libs-gui triggers a rendering request to the libs-back subsystem, which handles platform-specific graphics operations. For example, on a Linux system using the X Window System, libs-back translates the button's abstract representation into Xlib calls like XDrawRectangle and XFillArc to render the button's shape and label. After the rendering is completed, libs-back sends confirmation back to libs-gui, which in turn notifies Gorm. Finally, Gorm updates its visual preview to reflect the newly added button, allowing the developer to interact with the updated interface.

This workflow highlights the object-oriented interaction between subsystems, where method calls like NSButton.alloc() and renderView() enforce encapsulation and modularity. The absence of direct involvement from libs-corebase or libs-base in this use case underscores the separation between graphical rendering and foundational utilities.

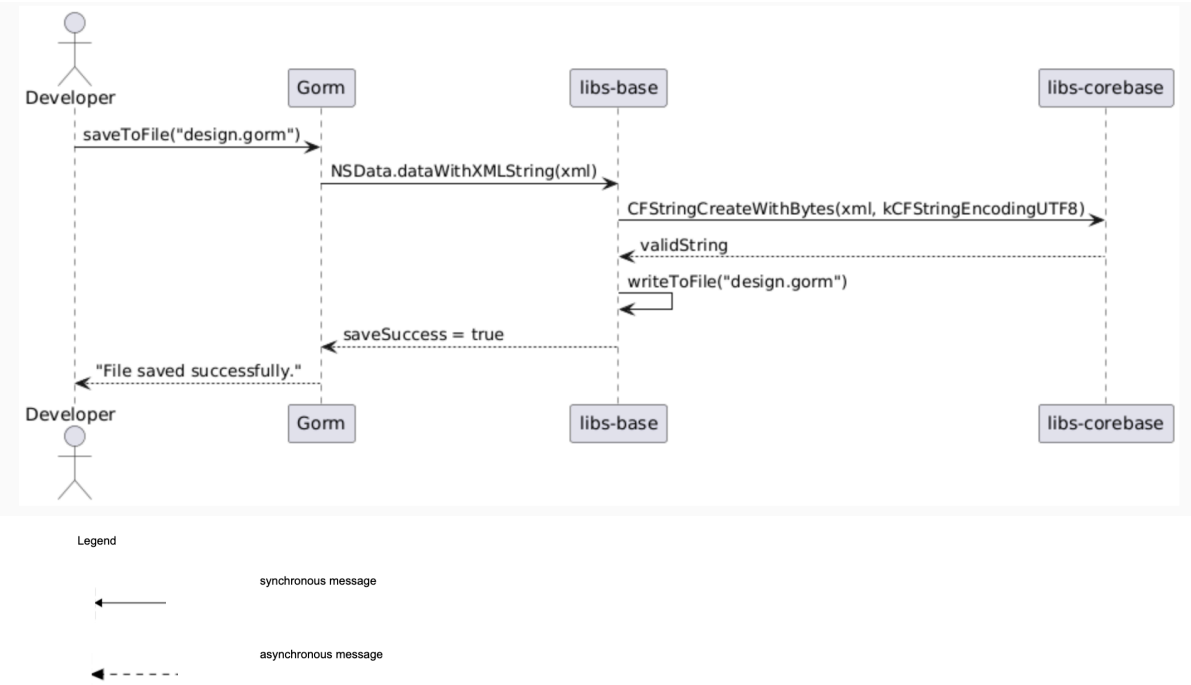


8.2 Use Case #2: Saving a .gorm File with Internationalized Text

This use case involves saving a GUI layout designed in Gorm as a .gorm file, where the interface includes components with Unicode text, such as labels in Japanese or Arabic. When the developer selects "Save" from Gorm's menu, the apps-gorm subsystem serializes the UI elements into an XML format. During serialization, Gorm leverages the libs-base subsystem to convert the XML string into binary data using the NSData class, a Foundation Kit utility for byte stream management.

To ensure the Unicode text is correctly encoded, libs-base invokes the CFStringCreateWithBytes() function from the libs-corebase subsystem. This Core Foundation function validates the byte stream against the specified encoding standard (e.g., UTF-8) and returns a validated CFString object. Once validated, libs-base writes the binary data to disk using NSData.writeToFile(), a method that handles file I/O operations. Upon successful writing, Gorm displays a confirmation dialog to the developer.

This use case reveals the bidirectional dependency between libs-base and libs-corebase, since the Foundation's NSData object relies on the Core Foundation's CFString object for encoding validation. It also demonstrates how apps-gorm indirectly depends on libs-corebase through libs-base, a deviation from the conceptual architecture that assumed Gorm's independence from the Core Foundation. The process exemplifies GNUstep's pragmatic approach to data handling, where cross-subsystem collaboration ensures robust support for internationalization.



9.0 Limitations & Lessons Learned

While studying the concrete architecture of GNUstep, we came across some limitations that affect the system's maintainability, performance, and cross-platform compatibility. One of the key limitations of GNUstep's architecture is the complexity of dependency management. The intricate relationships between subsystems, such as the bidirectional dependencies between libs-base and libs-corebase or the role of libobjc2, make maintenance challenging. Modifying one subsystem can have unintended ripple effects across the architecture. Additionally, while GNUstep provides documentation for individual libraries, there is a lack of resources detailing how these subsystems interact in practice. This documentation gap made it difficult to anticipate certain dependencies during our analysis.

Another limitation includes the cross-platform compatibility. While GNUstep aims to be portable across various operating systems, certain graphical rendering operations within libs-back are still dependent on system-specific implementations, such as X11 for Linux or Windows GDI. Hence, to ensure uniform behavior across platforms, additional development efforts are required. Furthermore, while GNUstep follows OpenStep principles, modern Objective-C enhancements and runtime optimizations have led to deviations from the original specification, potentially causing compatibility issues for legacy applications. Lastly, the layered structure of GNUstep, though modular, introduces performance overhead. The reliance on multiple abstraction layers such as libs-gui depending on libs-back, which then interfaces with platform-specific backends, could result in increased processing time, particularly in graphical applications.

Nonetheless, despite these limitations, we learned several important lessons about concrete architecture and system design. One key takeaway is the importance of reflection analysis in verifying architectural assumptions. While the conceptual model provided a useful reference, our analysis revealed significant discrepancies between theoretical design and actual implementation, demonstrating the need for empirical validation through tools like Understand. Additionally, while modularity enhances maintainability, it must be carefully designed to avoid unnecessary coupling between components, as seen in the unexpected dependencies between libs-corebase and libs-back.

Another important lesson was the runtime of the libraries. Our initial conceptual model did not account for libobjc2, demonstrating how critical runtime components are to a system's functionality. This omission reinforced the importance of understanding underlying system dependencies, particularly when working with languages like Objective-C, which rely heavily on dynamic runtime features. Moreover, our analysis demonstrated the ongoing challenges of cross-platform development. Although GNUstep is designed for platform independence, the differences in backend implementations require continuous adaptation and testing to maintain consistent functionality across operating systems.

Lastly, our study highlighted the importance of thorough and up-to-date documentation. Many of the challenges we encountered in tracing dependencies and understanding system interactions came from incomplete or scattered documentation. Improved architectural documentation would immensely help future developers in navigating, maintaining, and extending GNUstep. Overall, this analysis highlighted the importance of aligning conceptual models with real-world implementations to keep software architecture accurate and adaptable.

10.0 Conclusion

This report provided an in-depth examination of GNUstep's concrete architecture, comparing it to the conceptual architecture derived from our previous report. Through the use of dependency extraction tools like Understand, we analyzed GNUstep's system organization, identified its key subsystems, and dependencies. Our findings concluded that GNUstep adheres to an object-oriented architecture, while also revealing structural differences between the conceptual architecture and the actual implementation.

One of the key insights was the presence of the libobjc2 subsystem, which was overlooked in the conceptual model. libobjc2 plays a critical role in enabling dynamic method dispatch, memory management, and compatibility with Apple's Objective-C framework. Additionally, discrepancies between assumed and actual dependencies, like the removed dependency from libs-base to libs-back, highlighted the complexities between the real-world model and theoretical model.

A detailed examination of the apps-gorm subsystem further demonstrated how GNUstep facilitates application development. The discovery of new dependencies, such as those between InterfaceBuilder and the Tool and Plugins directories, displays the importance of analyzing the concrete design of a subsystem to improve our understanding of the overall system. Moreover, the

study of GNUstep's external interfaces revealed how its API, GUI, and system integration mechanisms work together to ensure cross-platform functionality.

Through this report, we gained a deeper understanding of the architectural trade-offs between GNUstep's conceptual and concrete designs. While the system mainly aligns with the conceptual design, there were practical constraints that led to structural modifications to improve interoperability, performance, and maintainability. This study reinforces the importance of validating conceptual architectures against concrete architectures to better understand a system's real-world behavior.

11.0 Glossary

11.1 Data Dictionary

apps-gorm - Contains Gorm, a visual interface builder for designing GUI layouts via drag-and-drop.

Concrete Architecture - The actual, as-built structure found through code and dependency analysis (as opposed to conceptual)

GormCore - A framework within the apps-gorm subsystem that provides the core classes for processing and interacting with gorm files.

GormObjCHeaderParser - A library within apps-gorm that implements a recursive descent parser for Objective-C header files, used to extract class information.

InterfaceBuilder - A framework that mimics Cocoa's Interface Builder, allowing the creation of custom palettes and inspector for designing GUI components.

libobjc2 - A subsystem that implements the Objective-C runtime. Used to handle key tasks such as dynamic method dispatch and memory management.

libs-back – A rendering system library responsible for abstracting different backend implementations (e.g., X11, Windows GDI).

libs-base – A core library providing fundamental Objective-C classes for collections, threading, and system interaction.

libs-corebase – A compatibility layer library ensuring integration with Apple's Core Foundation framework.

libs-gui – A library providing graphical user interface components such as windows, buttons, and menus.

Naming Conventions - The specific rules for naming classes, libraries, etc.

OpenStep Specification - The set of API standards that GNUstep implements to maintain compatibility with Apple's Cocoa API and support cross-platform Objective-C development.

Reflexion Analysis - A comparison of the conceptual architecture versus the concrete architecture to identify discrepancies.

Understand - A static analysis and visualization tool used to derive the system's concrete architecture.

11.2 Naming Conventions

apps - Used for application-level modules

libobjc2 - A special naming for the Objective-C runtime system

libs - Used for core library subsystems

NS – Stands for NextStep, a prefix used for class names and operations inherited from OpenStep (e.g NSThread and NSResponder).

12.0 References

[1] Botto, F., Frith-Macdonald, R., Pero, N., & Robert, A. (2001-2004). *Objective-C language and GNUstep base library programming manual*. Free Software Foundation. Retrieved from http://andrewd.ces.clemson.edu/courses/cpsc102/notes/GNUStep-manual.pdf

[2] GNUstep. (2024). *GNUstep API documentation*. Retrieved from https://www.gnustep.org

[3] GNUstep. (2024). *AppKit documentation*. GNUstep MediaWiki. Retrieved from https://mediawiki.gnustep.org/index.php/AppKit

[4] GNUstep. (2024). *Gorm.* GNUstep. Retrieved from https://www.gnustep.org/experience/Gorm.html

[5] Pero, N. (2010). *1 What are Distributed Objects (DO)*. GNUstep Distributed Objects. Retrieved from https://www.gnustep.org/nicola/Tutorials/DistributedObjects/node1.html

[6] GNUstep. (2025). *GNUstep: libs-back* (Version 0.32.0). GitHub. https://github.com/gnustep/libs-back/blob/master/ANNOUNCE

[7] GNUstep. (2024). *GNUstep: libs-base* (Version 1.30.1). GitHub. https://github.com/gnustep/libs-base/blob/master/Documentation/manual/BaseLibrary.texi

[8] GNUstep. (2016). *GNUstep: libs-base* (Version 1.31.1). GitHub. https://github.com/gnustep/libs-base

[9] GNUstep. (2012). *GNUstep: libs-corebase* (Version 0.2). GitHub.https://github.com/gnustep/libs-corebase/blob/master/Documentation/MainPage.dox

[10] GNUstep. (n.d.). *Gnustep/apps-gorm*. GitHub. https://github.com/gnustep/apps-gorm

[11] GNUstep. (2024). *Back - GNUstep Wiki*. GNUstep MediaWiki. https://mediawiki.gnustep.org/index.php/Back

[12] GitHub. (n.d.). *gnustep/libs-corebase*. GitHub. https://github.com/gnustep/libs-corebase

[13] GNUstep. (n.d.). *GNUstep: libs-gui* (Version 0.32.0). GitHub. https://github.com/gnustep/libs-gui

[14] GNUstep. (n.d.). *GNUstep: tools-make* (Version 2.9.3). GitHub. https://github.com/gnustep/tools-make

[15] GNUstep. (2024). *ProjectCenter.app documentation*. GNUstep MediaWiki. Retrieved from https://mediawiki.gnustep.org/index.php/ProjectCenter.app

[16] GNUstep. (2025). *GNUstep: tools-windows-msvc*. GitHub. https://github.com/gnustep/tools-windows-msvc