Concrete Architecture Report: GNUStep

March 14th, 2025

CISC 326/322
Group 34: Yin Yang
Aatif Mohammad: 22am37@queensu.ca
Madison MacNeil: 20mkm17@queensu.ca
Avery McLean: 21arcm@queensu.ca
Aden Wong: wong.aden@queensu.ca
Greg Costigan: 21gjc10@queensu.ca
Ryan Vuscan: ryan.vuscan@queensu.ca

Table of Contents

1. Abstract

GNUstep is a free, open-source implementation of the NextStep and OpenStep frameworks, providing a strong platform for cross-compatible application development in an object-oriented environment. This report explores the concrete architecture of GNUstep, analyzing its as-built structure and how its components interact at a system level. We will have a particular focus on Gorm, GNUstep's graphical interface builder, describing its role within the system and its internal architectural design.

To construct an understanding of the concrete architecture, we analyze the system's dependency structure, assigning top-level entities into subsystems and identifying architectural styles and design patterns. This report also compares the concrete architecture against our previously determined conceptual architecture, performing an analysis to highlight discrepancies and their rationales. Through this comparison, we aim to provide a comprehensive understanding of the GNUstep implementation, its modular design, and its adherence to software engineering principles.

2. Introduction and Overview

This report aims to recover the concrete architecture of GNUstep through an analysis of its software dependencies, subsystem interactions, and architectural style. The investigation is centered around the use of the "Understand" tool (developed by SciTools), an analysis tool that helps to examine file dependency, import patterns, and inter-component interactions in a graphical display. By analyzing the results of this tool, we arrive at a clearer picture of how GNUstep is structured at the implementation level.

Gorm is a major focus of this analysis. As GNUstep's graphical interface builder, it plays a vital role in facilitating the development of user interfaces within the system. The report will explore its internal architecture, subsystem interactions/dependencies, and how its design corresponds with GNUstep's object-oriented, modular nature.

We also conduct a reflexion analysis to validate our findings. This involves comparing the recovered concrete architecture with our previously determined conceptual model, highlighting discrepancies that were not apparent in the conceptual analysis.

The report also presents an outline of the concrete architecture, specifically the top-level subsystems and their interactions/dependencies. We document the derivation process used to map out the system's architecture. The report will ultimately provide a comprehensive architectural analysis of GNUstep, its structural organization, design principles, and implementation.

3. Architecture Derivation

The file dependency graphs in Scitools Understand software were explored extensively to recover the concrete architecture of GNUstep, beginning with the highest level directory of the codebase and progressively delving into the inner components of each subsystem by double-clicking to reveal the components that exist within them and their internal and external dependencies in the graphical view. This allowed a view of which dependency relationships were expected versus unexpected. Examining the highest level dependencies provided initial insight into areas requiring more granular exploration to understand component interactions. For example, when viewing the calls/called-by graph, a dependency between libs-base and libs-corebase could be observed, but further investigation was needed to determine the specific nature of this dependency: Which function/class in libs-base calls which function/class in libs-corebase?

Inspection of the file structure of the codebase was used to determine how classes are organized and grouped within the system and what import patterns exist between components. The Dependency Browser was used to see which specific classes in which folders were being accessed. This systematic approach to architecture discovery helped reveal both the intended hierarchical structure of GNUStep and the practical cross-layer and upward dependencies that reflect its real-world implementation requirements.

4. Top-Level Subsystems

4.1 Conceptual Architecture Review

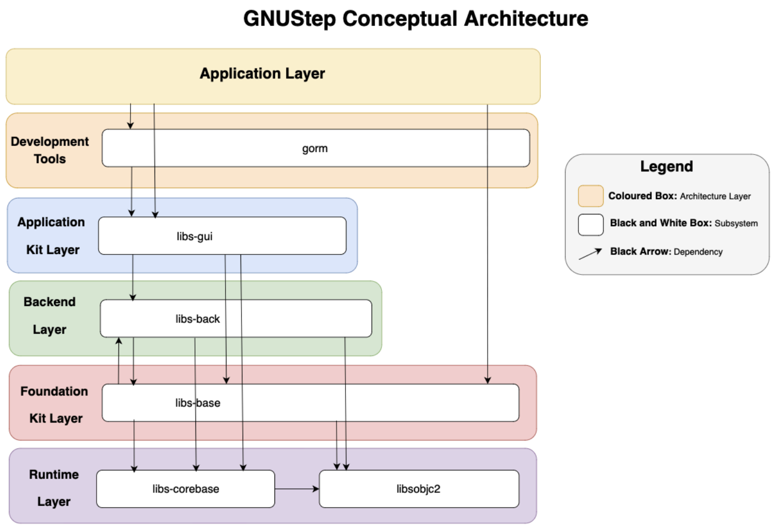


Figure 1: Layered Conceptual Architecture of GNUstep
Please Note: This conceptual architecture is the same as in the first report, but the formatting has been altered to remain consistent with the formatting of the concrete architecture diagram in this second report.

GNUstep makes use of a couple different architectural styles to allow for easy, seamless construction of software for its users. Its layered architecture contains the development tools layer, which contains an UI environment; using Gorm, it allows developers to drag and drop elements to create graphical interfaces. The application kit layer manages how UI elements are created and how events are handled at runtime with the GUI library. The backend layer exists as a backend to abstract the UI objects created by the GUI library, making GNUstep compatible with any OS. The application kit interacts with the backend layer to provide base services and system commands. Finally, the runtime layer provides core libraries as well as objective-c runtime.

GNUstep also incorporates an object-oriented style based on the use of Objective-C. This style allows for structuring the components in a modular way, which allows for greater scalability and flexibility when incorporating new features. GNUstep makes use of Obj-C's protocols and categories. This architecture simplifies the development of applications and supports the creation of dynamic and maintainable software in GNUstep.

4.2 Concrete Architecture

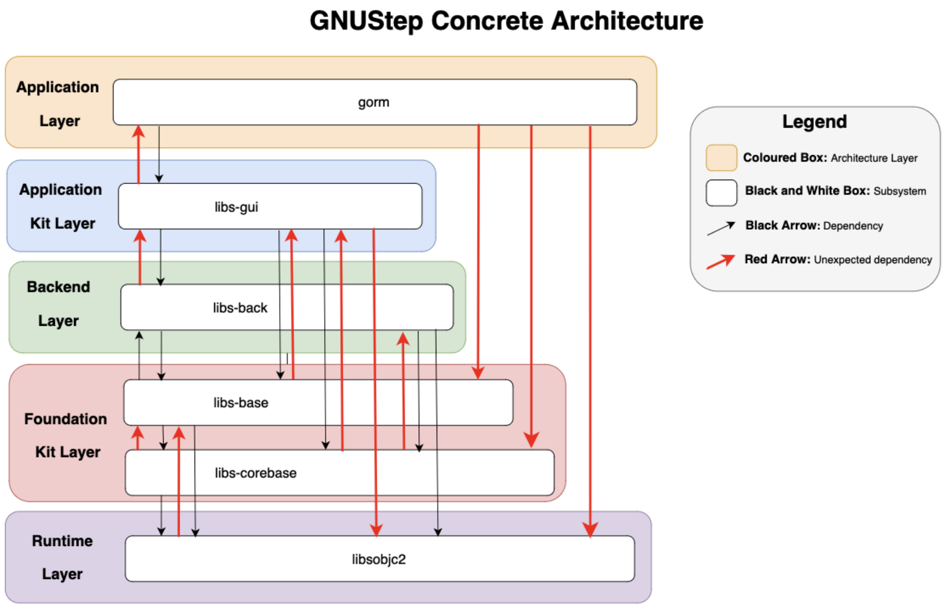


Figure 2: Layered Concrete Architecture of GNUstep
Please Note: This figure has been redrawn from an Understand diagram, as to best demonstrate the layers that exist in GNUsteps architecture.

The concrete architecture of GNUStep reveals a complex web of dependencies between top-level subsystems that goes beyond the initially described conceptual architecture. These dependencies reflect the sophisticated modularity of the GNUStep system, where layers aren't strictly sequential in their dependencies. GNUStep is fundamentally object-oriented, with each subsystem implemented as a library of Objective-C classes and C interfaces. This object-oriented design enables rich interactions between the subsystems across different architectural layers:

Hierarchical Layer Dependencies

GNUstep establishes a primary downward flow where each layer depends on services provided by layers below it. The Application Layer (gorm) uses the Application Kit Layer (libs-gui) for GUI component rendering, which relies on the Backend Layer (libs-back) for windowing and graphics library integration, which in turn uses the Foundation Kit Layer (libs-base/libs-corebase) for core functionality, with everything ultimately depending on the Runtime Layer (libsobjc2) for Objective-C management at runtime.

Cross Layer Dependencies

Many components bypass intermediate layers to directly access libs-base and libs-corebase because these libraries provide fundamental capabilities needed throughout the system. The libs-gui component depends directly on libs-base and libs-corebase (bypassing libs-back) because GUI components need foundational data structures and utilities independent of windowing system integration. For example, a text field needs string handling capabilities from libs-base regardless of which windowing system renders it. Gorm depends directly on libs-base, libs-corebase, and libsobjc2 because not all of its functionality requires graphical components. For instance, gormtool accesses Gorm functionality from the command line, bypassing GUI requirements but still needing foundational classes.

All subsystems depend directly on libsobjc2 because libsobjc2 manages the dynamic nature of Objective-C at runtime, resolving class and method references that aren't fully determined at compile time.It handles Objective-C's message-passing mechanism, which is fundamental to how objects interact in the entire system. libsobjc2 provides capabilities for examining and modifying the structure of classes and objects at runtime, which is essential for many GNUStep features.

Upward Layer Dependencies

The upward-layer dependencies in the GNUStep architecture exist generally because lower-level layers sometimes need to access information about or communicate with higher-level layers. Lower layers often need to notify higher layers about events (like user input or system changes). Lower layers sometimes need to query the state of higher-level components to make appropriate decisions. For example, a rendering backend might need to know the current visual state of GUI elements.

4.3 Reflexion Analysis

In our analysis of GNUstep's concrete architecture, we found that while our conceptual architecture report accurately captured several key aspects, there are notable differences between the initial conceptual view and the actual implementation. This reflexion analysis aims to highlight and explain these divergences by examining the rationale for each difference.

Gorm Relocation to the Application Layer

In our conceptual analysis, we placed Gorm in the Development Tools layer, thinking of it as a tool to help developers with creating software. After looking at the concrete architecture, we realized Gorm's role is more important in the functioning of the application. Gorm is integral for how the application runs, handling data storage and retrieval, which are essential. Gorm is responsible for interacting with the database, which is crucial to GNUstep's functionality. Because of this, it makes more sense for Gorm to be a part of the application layer, as it directly affects how the application functions.

Libs-corebase Relocation to Foundation Kit Layer

Libs-corebase is not part of the runtime layer; rather, it provides a set of C interfaces that act as wrappers for the Objective-C classes defined in Libs-base. Since Objective-C is a superset of C, it retains full compatibility with C while introducing object-oriented capabilities. These C interfaces in Libs-corebase are essential for enabling GNUstep to interact with low-level programs, libraries, or system components that only support C and lack native Objective-C support. Instead of performing translation dynamically during execution, this Objective-C-to-C bridging occurs at the interface level (compile time) ensuring better performance and predictable behavior at runtime.

In contrast, Libs-objc2 is a core part of the Objective-C runtime layer. Its primary function is to manage the dynamic nature of Objective-C, particularly resolving class and method ambiguities that are not detected at compile time. Objective-C's runtime flexibility allows method dispatching, message forwarding, and class modifications to occur dynamically, which introduces potential risks when integrating with lower-level systems. A key issue it addresses is type-dependent dispatch. Traditionally, Objective-C resolves methods by name alone, which can cause type mismatches—e.g., if two methods share a name but expect different argument types, incorrect register access can lead to memory corruption. Libs-objc2 helps mitigate these issues by ensuring proper resolution of such conflicts, maintaining type safety, and preserving the integrity of method calls and memory access during execution.

Unexpected Dependencies

→ uni-directional dependency
↔ bi-directional dependency

Gorm → Objective C

Gorm depends on Objective-C because it relies on the Objective-C runtime for dynamic object creation, class loading, and message parsing. It uses the Appkit frameworks and Foundation from Objective-C to decode and manipulate UI elements at runtime, extend classes through categories, and dynamically load bundles. This enables Gorm to work with GNUstep's graphical models and construct UI components.

Gorm → Libs-base

Gorm also depends on libs-base, which is part of the GNUstep base library, providing Objective-C runtime functions, collections, and utilities. A key part of this dependency is the GormGModelWrapperLoader, which processes .gmodel files. Unlike typical files, these require the GModelDecoder to dynamically interpret and reconstruct UI elements. This process relies heavily on the library's unarchiving mechanisms to properly restore the interface components.

Libs-back → libs-gui

Libs-back depends on libs-gui for functions from the GUI library for rendering or graphical processing tasks. It specifically uses components such as DPSclip, DPSimage and DPSstroke, suggesting that libs-back uses GUI-related functionality for handling images and rendering.

Objective-C ↔ libs-base

We correctly stated that libs-base depended on objective-C, but it is actually a mutual dependency. The main reason for Objective-C depending on libs-base is runtime.c, which is largely responsible for handling dynamic class modifications, like adding instance variables, methods and protocols at runtime. These functions are core to Objective-C's runtime type system.

Libs-base ↔ libs-gui

We had also correctly stated the dependency of libs-gui to libs-base, but did not realize it is also a mutual dependency. Libs-base depends on libs-gui for higher-level UI and system functionality, including sorting, collections and file operations. Additionally it also interacts with libs-gui's internal data structures via macros, bypassing abstraction layers for efficiency.

Libs-Corebase ↔ libs-base

We correctly identified the dependency between libs-base and libs-corebase but failed to realize it was a mutual dependency. Libs-corebase depends on libs-base for Obj-C runtime support, error handling, and threading between C and Objective-C API's.

Libs-Corebase → libs-gui

Corebase depends on libs-gui for processing, Unicode handling and date formatting, as well as networking and resource management. It also uses libs-gui for event loops, socket communication memory management and Obj-C runtime features for GUI-related interactions.

Libs-gui ↔ gorm

We said that gorm heavily relies on libs-gui however, libs-gui also depends on gorm for Core foundation-style data structures and memory management with the GUI framework.

Libs-gui → Objective C

Libs-gui relies on libobjc2 mainly because of the Automatic Reference Counting (ARC) and NSAutoreleasePool classes. The main function of these classes is to handle memory allocation for

Objective-C objects, especially in scenarios where memory needs to be managed before ARC can be applied.

Libs-Corebase → libs-back

Corebase depends on back by calling CFGregorianDateIsValid to verify that certain dates are valid, probably for date based computations or rules.

5. 2nd Level Sub-System

5.1 Conceptual Architecture

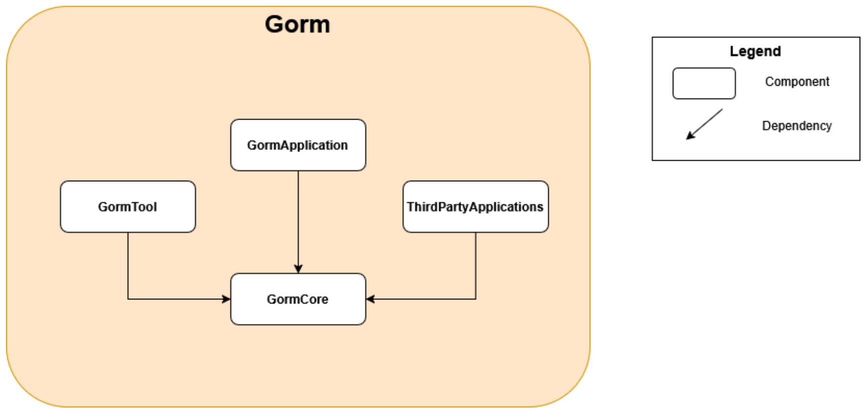


Figure 3: Conceptual architecture of Gorm

Gorm is conceptually designed to be a tool that allows developers to visually design UI without having to code the elements manually. It resides in the highest level of abstraction providing users with visual representations of lower-level components. It creates a bridge between the visual design process and the underlying component architecture. At its core, Gorm follows a modular design philosophy that highlights separation of concerns and component reusability. This approach allows for flexibility, extensibility, and maintenance advantages, enabling developers to enhance functionality without disrupting the core system. The conceptual architecture divides Gorm into distinct modules with well-defined responsibilities and interfaces between them. Gorm follows an object oriented design approach being a part of GNUStep, but mainly has a component based architecture.

GormApplication serves as the primary visual interface component, providing an interactive design environment for users to create and manipulate graphical user interfaces. It depends on the GormCore for core functionalities.

GormTools functions as a command-line utility, offering automation capabilities for interface design tasks. This component extends Gorm's functionality to non-graphical environments. It depends on GormCore for functionality as well.

GormCore acts as the central interface design engine and core component of the architecture. GormCore handles the essential operations for managing interface objects, their properties, and relationships. It serves as the foundation upon which other components build upon and interact.

ThirdPartyApplications serve as the extension mechanism for custom functionality. The component allows third-party developers to add new capabilities to Gorm without modifying the core codebase. It relies on GormCore for functionality.

5.2 Concrete Architecture

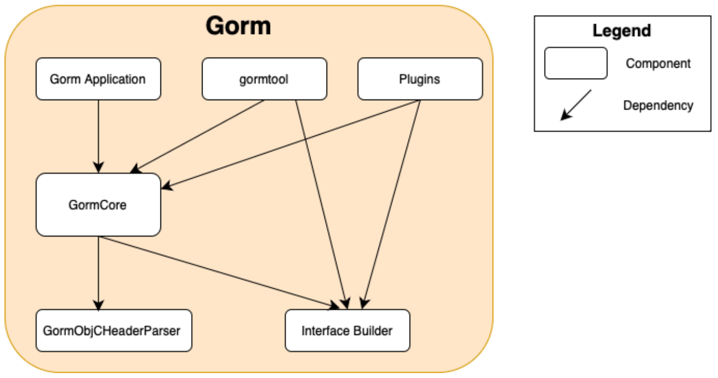


Figure 4: A Visualization of the Inner Concrete Architecture of Gorm

Gorm, which stands for "Graphical Object Relationship Modeller," is a GNUstep GUI application used for designing and managing user interfaces. Its architecture is built to facilitate the creation, manipulation, and interaction of graphical elements within an object-oriented environment. Gorm allows developers to dynamically load UI components, establish object relationships, and inspect properties through an intuitive visual interface.

The software is designed with an emphasis on modular structure to ensure extensibility and flexibility. The Gorm App, which provides a graphical interface for drag-and-drop design, object manipulation, and interactive testing, is a core component of Gorm. One important feature of the Gorm App is the palette, a sort of "toolbox" that contains all the reusable UI elements like buttons, text fields, sliders, and other widgets that can be dragged and dropped onto the design canvas. Another key feature is the Inspector tool, which allows developers to inspect and modify properties of the UI components they are implementing in their interface.

The second user-facing component is Gormtool, a command-line utility that extends Gorm's functionality headlessly for file manipulation. It is particularly useful for automation, scripting, and integrating Gorm into larger workflows.

Both the Gorm App and Gormtool depend on GormCore, Gorm's core engine, which handles all the internal logic and data structures for interface development, for backend functionality. It contains all the essential classes for interacting with .gorm files, manages UI components, their properties, and interactions. GormCore is responsible for saving UI structures to .gorm files and reloading them. Through the use of plugins, third party applications are able to integrate with Gorm through GormCore, insofar as any Gorm Plugins depend directly on GormCore for functionality.

GormCore then relies on GormObjCHeaderParser, a library that parses Objective-C header files to extract relevant information about classes and methods. This functionality is separated into a library to allow other applications or tools to make use of it, as is explicitly described on the GNUStep Github repository.

Gormtool and any potential plugins a user has chosen to include, depend on InterfaceBuilder, a clone of Apple's InterfaceBuilder framework. InterfaceBuilder enables the creation of custom palettes and inspectors outside of Gorm's core functionalities. The inclusion of InterfaceBuilder within Gorm's architecture allows Gorm to be extended without altering its core features, enabling third-party applications or plugins to unify with Gorm seamlessly. These external tools rely on GormCore for core functionality and InterfaceBuilder for creating custom classes.

5.3 Reflexion Analysis

Our concrete architecture of GNUstep's Gorm reveals several key discrepancies from the conceptual model initially derived. By analyzing these differences, we can better understand the structural and functional challenges encountered during implementation.

The conceptual architecture presented a clean, modular hierarchy where GormApplication, GormTool, and ThirdPartyApplications interact exclusively with GormCore. However, the concrete architecture introduced additional components and dependencies that were not initially accounted for. The first major divergence was the introduction of GormObjCHeaderParser, which was required to extract Objective-C class and method information. This component was absent in the conceptual model because it was initially assumed that interface object relationships could be managed solely by GormCore. However, in practice, dynamic Objective-C integration was necessary for handling GNUstep's runtime environment, leading to its inclusion. Another significant change was the integration of InterfaceBuilder, which became a critical component for enabling extensibility through third-party plugins. The conceptual model assumed that extensions would interact with GormCore directly, but in reality, a structured mechanism for managing UI extensions was required. InterfaceBuilder facilitated this by allowing external developers to create custom palettes and inspectors without modifying the core system, leading to its unexpected presence in the final architecture.

Additionally, interdependencies between components became more complex than originally anticipated. While the conceptual model maintained strict separation, the concrete implementation shows multiple bidirectional dependencies, reflecting tighter integration needs. For example, GormTool, which was designed as an independent command-line utility, required deeper integration with InterfaceBuilder and GormCore to manipulate UI elements effectively. These structural shifts demonstrate how real-world constraints led to a more interconnected system.

The unexpected dependencies seen in the concrete architecture were primarily driven by practical needs rather than initial design principles. The inclusion of GormObjCHeaderParser was essential for maintaining compatibility with GNUstep's Objective-C-based infrastructure, which the conceptual architecture did not fully consider. Similarly, InterfaceBuilder was required to ensure third-party extensibility, something that the initial modular design did not accommodate adequately. Another notable divergence is the presence of dependencies that bypassed intended hierarchical layers. In theory, Gorm was supposed to interact only with high-level application components, but in practice, it required direct dependencies on lower layers of GNUstep's framework, such as foundation and runtime libraries. These changes reflect how theoretical modularity often gives way to functional necessity when working within a complex software ecosystem.

By comparing the conceptual and concrete architectures, we see how the idealized design of Gorm evolved to accommodate integration challenges, extensibility needs, and real-world constraints. The unexpected dependencies highlight the balance between maintaining architectural clarity and ensuring practical usability within the GNUstep environment.

6. Use Case 1 - Instantiating a Window Object in Gorm

The window creation process utilizes our understanding of the layered architecture, beginning in the Application Layer where Gorm initiates the sequence by establishing memory management through an NSAutoreleasePool in the Foundation Kit Layer. Gorm then creates a window by instantiating NSWindow with specific dimensions and style masks. During initialization, NSWindow creates a content view using NSView and sets a window title via NSString, demonstrating interaction with other Application Kit and Foundation Kit components. The window then connects to the Backend Layer by obtaining a GSDisplayServer instance through GSCurrentServer() - an important architectural detail where an abstract class defined in libs-gui is implemented by concrete backend-specific subclasses. The display server creates the actual window and assigns it a unique window number, after which NSWindow configures properties like window level. When Gorm calls makeKeyAndOrderFront:, NSWindow processes this through a chain of methods (orderFrontRegardless and orderWindow:relativeTo:) that ultimately communicate with the display server to make the window visible on screen. Throughout this process, the Runtime Layer facilitates object messaging via objc_msgSend.

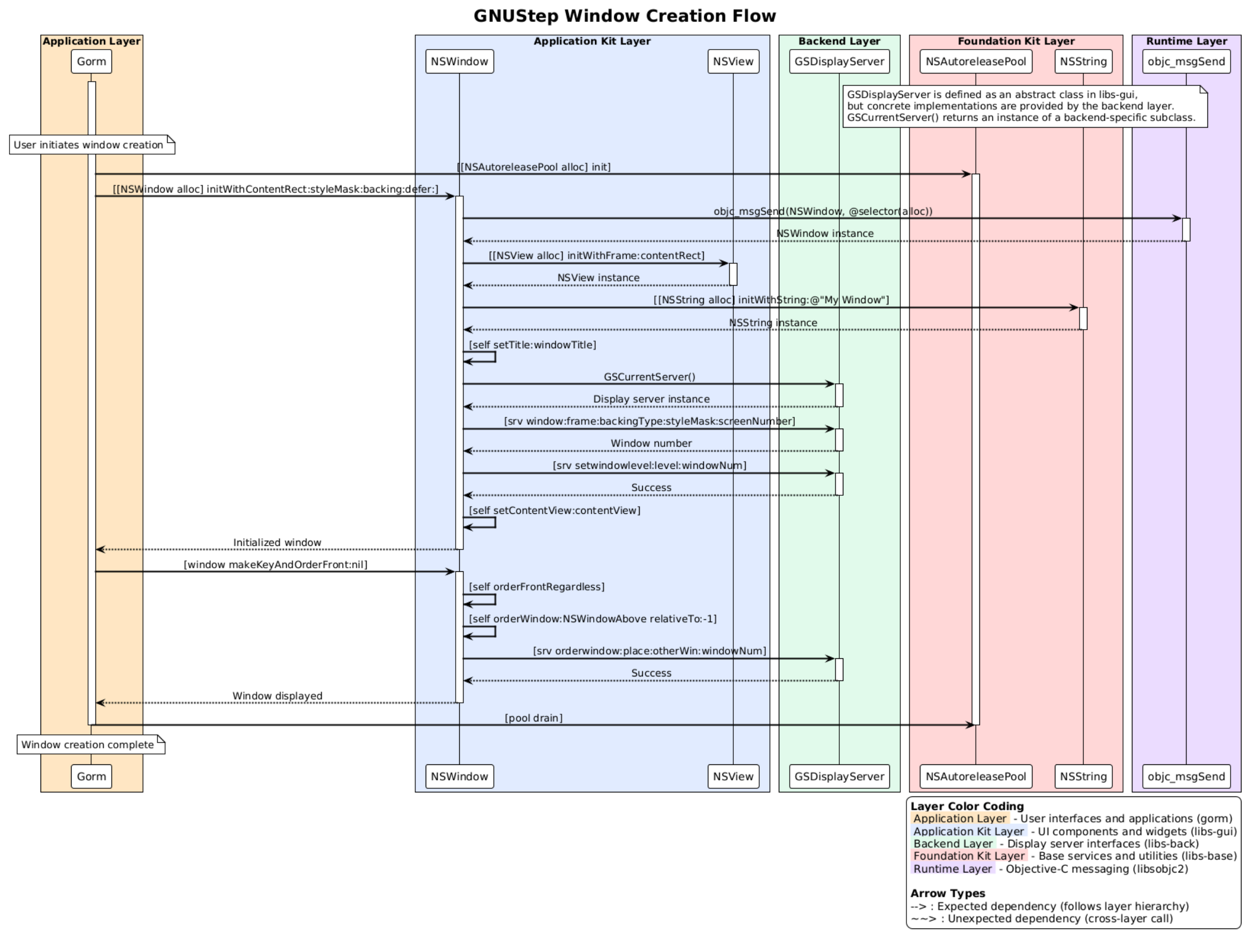
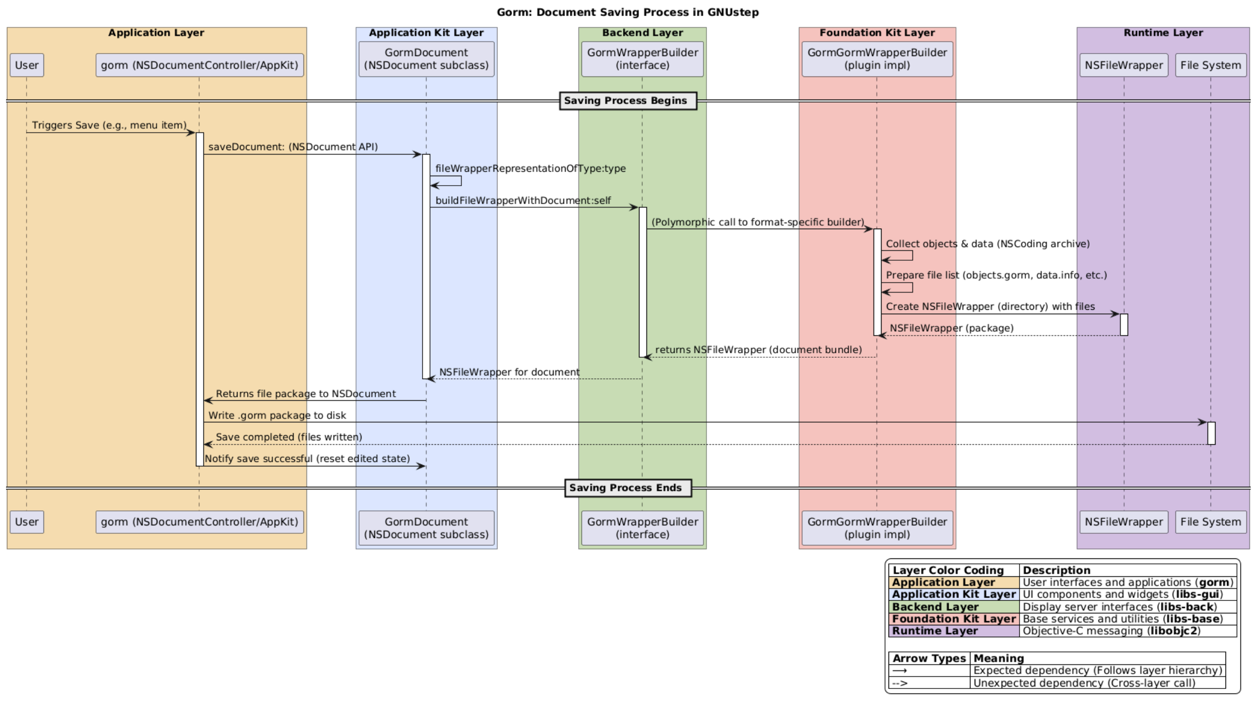

GNUStep Window Creation Flow
Application Layer
Application Kit Layer
Backend Layer
Foundation Kit Layer
Runtime Layer
Gorm
NSWindow
NSView
GSDisplayServer
NSAutoreleasePool
NSString
objc_msgSend
GSDisplayServer is defined as an abstract class in libs-gui,
but concrete implementations are provided by the backend layer.
GSCurrentServer() returns an instance of a backend-specific subclass.
User initiates window creation
[[NSAutoreleasePool alloc] init]
[[NSWindow alloc] initWithContentRect:styleMask:backing:defer:]
objc_msgSend(NSWindow, @selector(alloc))
NSWindow instance
[[NSView alloc] initWithFrame:contentRect]
NSView instance
[[NSString alloc] initWithString:@"My Window"]
NSString instance
[self setTitle:windowTitle]
GSCurrentServer()
Display server instance
[srv window:frame:backingType:styleMask:screenNumber]
Window number
[srv setwindowlevel:level:windowNum]
Success
[self setContentView:contentView]
Initialized window
[window makeKeyAndOrderFront:nil]
[self orderFrontRegardless]
[self orderWindow:NSWindowAbove relativeTo:-1]
[srv orderwindow:place:otherWin:windowNum]
Success
Window displayed
[pool drain]
Window creation complete
Layer Color Coding
Application Layer - User interfaces and applications (gorm)
Application Kit Layer - UI components and widgets (libs-gui)
Backend Layer - Display server interfaces (libs-back)
Foundation Kit Layer - Base services and utilities (libs-base)
Runtime Layer - Objective-C messaging (libsobjc2)
Arrow Types
--> : Expected dependency (follows layer hierarchy)
~~> : Unexpected dependency (cross-layer call)

Figure 5: A sequence diagram of the window creation process.

7. Use Case 2 - Saving a document in Gorm

The document saving process in Gorm follows the structured hierarchy of the GNUstep architecture, beginning in the Application Layer where the user triggers a save action via Gorm's NSDocumentController. This prompts GormDocument, a subclass of NSDocument in the Application Kit Layer, to generate a file representation through fileWrapperRepresentationOfType:. The request is delegated to the Backend Layer, where GormWrapperBuilder constructs an NSFileWrapper package, utilizing a format-specific implementation in the Foundation Kit Layer via GormGormWrapperBuilder. This builder serializes objects and metadata, assembling components such as objects.gorm and data.info. The Runtime Layer facilitates storage through NSFileWrapper, which interacts with the file system to write the structured package to disk. Once the write operation completes, control returns to GormDocument, which resets the edited state and notifies NSDocumentController of a successful save, ensuring consistency across the application.


Gorm: Document Saving Process in GNUstep
Application Layer
Application Kit Layer
Backend Layer
Foundation Kit Layer
Runtime Layer
User
gorm (NSDocumentController/AppKit)
GormDocument (NSDocument subclass)
GormWrapperBuilder (interface)
GormGormWrapperBuilder (plugin impl)
NSFileWrapper
File System
Saving Process Begins
Triggers Save (e.g., menu item)
saveDocument: (NSDocument API)
fileWrapperRepresentationOfType:type
buildFileWrapperWithDocument:self
(Polymorphic call to format-specific builder)
Collect objects & data (NSCoding archive)
Prepare file list (objects.gorm, data.info, etc.)
Create NSFileWrapper (directory) with files
NSFileWrapper (package)
returns NSFileWrapper (document bundle)
NSFileWrapper for document
Returns file package to NSDocument
Write .gorm package to disk
Save completed (files written)
Notify save successful (reset edited state)
Saving Process Ends

Layer Color Coding	Description
Application Layer	User interfaces and applications (**gorm**)
Application Kit Layer	UI components and widgets (**libs-gui**)
Backend Layer	Display server interfaces (**libs-back**)
Foundation Kit Layer	Base services and utilities (**libs-base**)
Runtime Layer	Objective-C messaging (**libobjc2**)

Arrow Types	Meaning
⟶	Expected dependency (Follows layer hierarchy)
-->	Unexpected dependency (Cross-layer call)

Figure 6: A sequence diagram of saving a document in Gorm

8. Lessons Learned

One of our main challenges we faced while writing this report was working with the Understand tool. While it is extremely useful for analyzing dependencies and visualizing software architectures, it has a steep learning curve. Initially it was difficult to navigate and would often have errors and crash with no explanation, making the process frustrating and time consuming. It took a while to learn how to extract meaningful insights from Understand, which slowed us down in the early portion of the assignment. In hindsight, we should have started earlier, as we spent a significant amount of time just getting comfortable using Understand. Additionally, our group's communication was less consistent than the previous report, which further contributed to delays. In hindsights, better coordination and an earlier start would have helped us manage our time more effectively.

Beyond the technical challenges from the assignment we learned about the importance of modularity in software design. We saw firsthand how tools like Gorm, as a standalone component in the application layer, benefit from being modular, making them easier to develop and maintain. Additionally, from analyzing GNUstep's concrete architecture we saw how bidirectional dependencies and dependencies to non-adjacent layers add complexity to software but are often the case in real world scenarios when building software. This analysis reinforced why careful architectural design is so important in large software systems.

9. Conclusions

In conclusion, our report of GNUstep's concrete architecture reveals a far more complex dependency structure than initially predicted. The conceptual architecture envisioned a clean, modular hierarchy, but the actual implementation includes bidirectional dependencies, cross-layer interactions, and more components previously not accounted for.

Gorm, GNUstep's graphical interface builder and the primary focus of our report, facilitates the development of user interfaces within the system via a modular design philosophy and a primarily component based architecture. Gorm was originally assigned to the Development Tools Layer but after further analysis was reassigned to the Application Layer because of its critical role in the functioning of applications, managing data retrieval, and more.

This analysis allows us to conclude that Gorm's role is more than just a development tool but an integral part of the system's runtime and application execution. It also allows us to conclude GNUStep as a highly interconnected system in which dependencies evolve based on necessity. It is worth noting that our group initially expected a neatly structured top-down framework. This has helped us understand just how important adaptability is in large-scale software systems, where maintaining strict hierarchy is often less practical than enabling efficient cross-component interactions.

10. Data Dictionary

API - A set of rules and protocols that enables software applications to communicate and exchange data, allowing them to interact and share functionalities

11. Naming Convention

UI - User Interface
GUI - Graphical User Interface
OS - Operating System
API - Application Programming Interface
ARC - Automatic Reference Counting

References

Building Apps with GORM — GNUstep. (2024). Github.io. https://gnustep.github.io/Guides/App/index.html

Casamento, G., & Frith-Macdonald, R. (2000). *Guide to the Gorm Application Version 0.9.2*. https://www.gnustep.org/resources/documentation/Gorm.pdf

GNUstep. (2025). Gnustep.org. https://www.gnustep.org/

Manuals for Understand. (2025). SciTools Support. https://support.scitools.com/support/solutions/articles/70000583171-manuals-for-understand